

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.

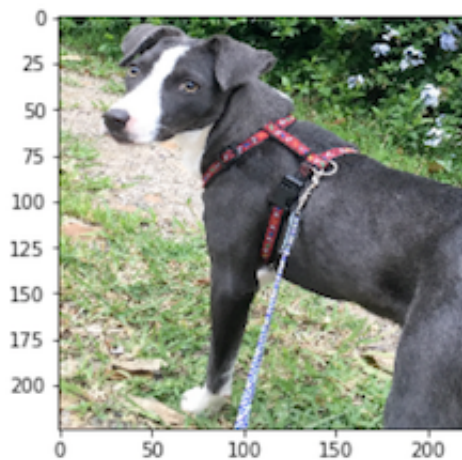
The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.

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!).

```
hello, dog!  
your predicted breed is ...  
American Staffordshire terrier
```



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
- [Step 1](#): Detect Humans
- [Step 2](#): Detect Dogs
- [Step 3](#): Create a CNN to Classify Dog Breeds (from Scratch)
- [Step 4](#): Create a CNN to Classify Dog Breeds (using Transfer Learning)
- [Step 5](#): Write your Algorithm
- [Step 6](#): Test Your Algorithm

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) (<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) (<https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip>). 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 \(http://www.7-zip.org/\)](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 [243]: import numpy as np
          from glob import glob

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

          # 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 \(http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html\)](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 \(https://github.com/opencv/opencv/tree/master/data/haarcascades\)](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 [244]: import cv2
import matplotlib.pyplot as plt
%matplotlib inline

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

# load color (BGR) image
img = cv2.imread(human_files[57])
# 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))

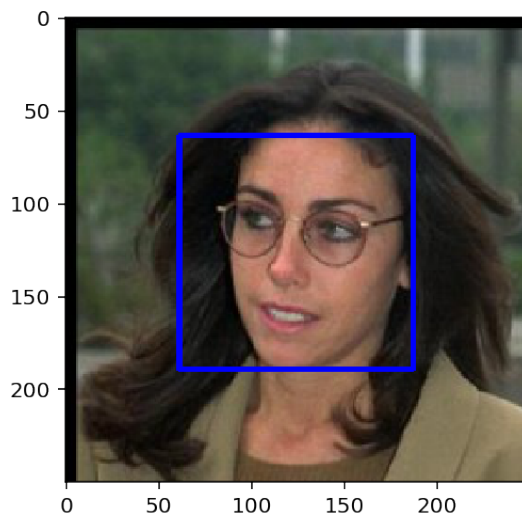
# get bounding box for each detected face
for (x,y,w,h) in faces:
    # add bounding box to color image
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

# 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 [245]: # 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 [246]: 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.

## TODO NO. 1 of
# Step One - create a function
def face_detection_test(files):
    # create two count files
    detection_count = 0; # initial count of zero
    # create total count from the length of file
    total_count = len(files)

    ## Create a loop which adds for every face detected
    ## through the function face_detector(img_path)
    for x in files:
        detection_count += face_detector(x)

    return detection_count, total_count
```

```
In [247]: run the function on both parameters
feature-based cascade classifiers - Face Detection")
face in human_files: {} / {}".format(face_detection_test(human_files_sho
face in dog_files: {} / {}".format(face_detection_test(dog_files_short))[

Haar feature-based cascade classifiers - Face Detection
Detect face in human_files: 99 / 100
Detect face in dog_files: 18 / 100
```

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 [248]: ### (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](http://pytorch.org/docs/master/torchvision/models.html) (<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/) (<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) (<https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a>).

```
In [249]: 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()

print("cuda available? {}".format(use_cuda))

cuda available? False
```

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

```

In [250]: from PIL import Image
import torchvision.transforms as transforms

# Set PIL to be tolerant of image files that are truncated.
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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 -- STEP ONE
    ## Return the *index* of the predicted class for that image -- STEP

    # STEP ONE - Load and pre-process the image from given img_path
    # open the image
    image = Image.open(img_path)

    # transform the image
    # VGG16 takes 244,244 image size as input,
    size = 224

    # Create a data transform process - resize the input, normalize it a
    data_transform = transforms.Compose([
        transforms.RandomResizedCrop(size),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(45),
        transforms.ToTensor(), # transform it into a to
        transforms.Normalize(mean=[0.485, 0.456, 0.406],
                             std=[0.229, 0.224, 0.225]))

    # Apply the above - transformation
    image_tensor = data_transform(image)

    # STEP TWO - return the *index*
    # Current image shape and size = (number of color channels, height a
    # Pytorch required = (num input imgs, number of color channels, heig
    # # fake batch dimension required to fit network's input dimensions
    image_tensor = image_tensor.unsqueeze(0)

    # check for CUDA/GPU
    if use_cuda:
        image_tensor = image_tensor.cuda()

```



```

# run the tensor through VGG16
image_tensor = VGG16(image_tensor)

# Returns a Tensor of shape (batch, num class labels)
prediction_tensor = image_tensor.data.numpy().argmax()

return prediction_tensor # predicted class index

```

```

In [251]: # Random Code Check - for personal inquiry
# This returns the index corresponding to the ImageNet class
# .. that is predicted by the pre-trained VGG-16 model.
VGG16_predict(dog_files_short[57])

```

Out[251]: 205

(IMPLEMENTATION) Write a Dog Detector

While looking at the [dictionary \(https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a\)](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 [252]: ### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    class_index = VGG16_predict(img_path)

    if class_index >= 151 and class_index <= 268:
        return True
    else:
        return False

```

```

In [253]: ## TEST dog_detector
dog_detector(dog_files_short[57])

```

Out[253]: False

```

In [254]: ## TEST dog_detector
dog_detector(human_files_short[34])

```

Out[254]: False

(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 [255]: ### 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_count = 0;  
    total_count = len(files)  
    for x in files:  
        detection_count += dog_detector(x)  
    x = detection_count/total_count  
    x = x * 100  
    return x
```

```
In [256]: print('VGG-16 Prediction')  
print('The percentage of the detected dog - Humans: {} %'.format(dog_det  
print('The percentage of the detected dog - Dogs: {} %'.format(dog_detec
```

VGG-16 Prediction

The percentage of the detected dog - Humans: 2.0 %

The percentage of the detected dog - Dogs: 72.0 %

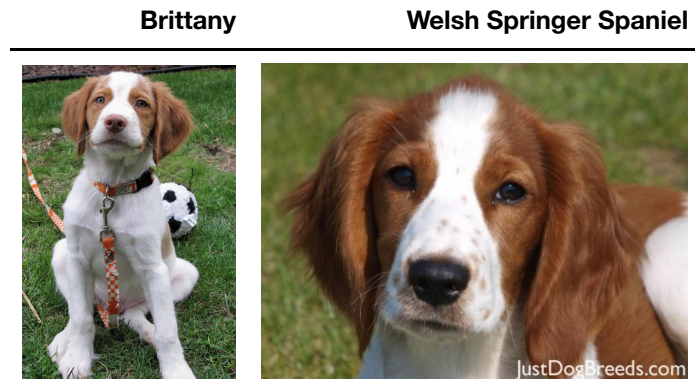
We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as [Inception-v3](http://pytorch.org/docs/master/torchvision/models.html#inception-v3) (<http://pytorch.org/docs/master/torchvision/models.html#inception-v3>), [ResNet-50](http://pytorch.org/docs/master/torchvision/models.html#id3) (<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 [257]: ### (Optional)  
### TODO: Report the performance of another pre-trained network.  
### Feel free to use as many code cells as needed.
```

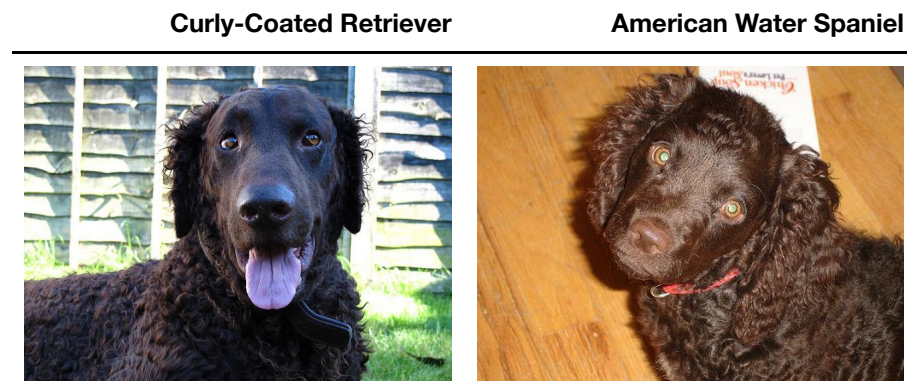
Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

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.



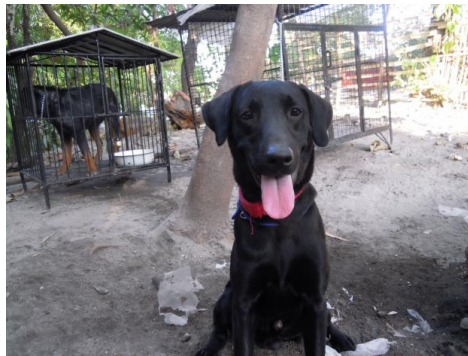
It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).



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 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!

(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>)!

```

In [258]: import os
          from torchvision import datasets, utils

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

          # Step One - Define (pre-hyper)parameters
          batch_size = 20 # Number of samples loaded for one batch
          num_workers = 0 # Number of subprocesses, if it's 0, it uses the main pr
          size = 224 # size required for the Image input for VGG16

          # Step Two - Create path for data
          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/')

          # Step Three - Create VGG16 standard normalization
          # Normalize images because the values of images should be loaded between
          standard_normalization = transforms.Normalize(mean=[0.485, 0.456, 0.406]
                                                         std=[0.229, 0.224, 0.225])

          # Step Four - built data_transform
          # Transform for Training Set
          transform_train = transforms.Compose([
              transforms.RandomHorizontalFlip(), # Augment - Random Horizontal Fli
              transforms.RandomRotation(45),     # Augment - Random Rotation
              transforms.RandomResizedCrop(size), # Augment - Random Resize Crop
              transforms.ToTensor(),
              standard_normalization
          ])

          # Transform for Validation Set - MINIMUM Augment - CenterCrop
          # Validation data required for Validation Check - No Augmentation
          transform_valid = transforms.Compose([
              transforms.Resize(size),
              transforms.CenterCrop(size),
              transforms.ToTensor(),
              standard_normalization
          ])

          # Transform for Test Set - NO Augment - only resize image
          transform_test = transforms.Compose([
              transforms.Resize((224,224)),
              transforms.ToTensor(),
              standard_normalization
          ])

          # Step Five - Data Batching for - test, train, and valid
          # Load the datasets with ImageFolder
          train_data = datasets.ImageFolder(train_dir, transform=transform_train)
          valid_data = datasets.ImageFolder(valid_dir, transform=transform_valid)

```

```

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

# Step Six - Data Loading
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,
                                           shuffle=False,
                                           num_workers=num_workers)

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

# Step Seven - Check the size of the dataset and Loaders
print("Size of Train Dataset = " + str(len(train_data)))
print("Size of Valid Dataset = " + str(len(valid_data)))
print("Size of Test Dataset = " + str(len(test_data)))
print("\n***** \n")
print("Length of Train Loader - " + str(len(train_loader)))
print("Length of Valid Loader - " + str(len(valid_loader)))
print("Length of Test Loader - " + str(len(test_loader)))

```

```

Size of Train Dataset = 6680
Size of Valid Dataset = 835
Size of Test Dataset = 836

```

```

*****

```

```

Length of Train Loader - 334
Length of Valid Loader - 42
Length of Test Loader - 42

```

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:

(IMPLEMENTATION) Model Architecture

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

```
In [276]: import torch.nn as nn
import torch.nn.functional as F

num_classes = 133 # total classes of dog breeds
dropout = nn.Dropout(0.3) # dropout layer (p=0.3)

# 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 """
        """ ARCHITECTURE """
        # convolutional layer 1 == (sees 224x224x3 image tensor, where
        # (input depth = 3, desired depth output = 16, kernel size = 3x3)
        self.conv1 = nn.Conv2d( 3, 16, 3)
        # max pooling downsize (222x222x16) to (111x111x16)

        # convolutional layer 2 == (sees 111x111x16 image tensor, 16 =
        # (input depth = 3, desired depth output = 32, kernel size = 3x3)
        self.conv2 = nn.Conv2d(16, 32, 3)
        # max pooling downsize (109x109x32) to (54x54x32)

        # convolutional layer 3 == (sees 54x54x32 image tensor, 32 =
        # (input depth = 32, desired depth output = 64, kernel size = 3x3)
        self.conv3 = nn.Conv2d(32, 64, 3)
        # max pooling downsize (52x52x64) to (26x26x64)

        # convolutional layer 4 == (sees 26x26x64 image tensor, 64 =
        # (input depth = 64, desired depth output = 128, kernel size = 3x3)
        self.conv4 = nn.Conv2d(64, 128, 3)
        # max pooling downsize (24x24x128) to (12x12x128)

        # convolutional layer 5 == (sees 12x12x128 image tensor, 128 =
        # (input depth = 128, desired depth output = 256, kernel size =
        self.conv5 = nn.Conv2d(128, 256, 3)
        # max pooling downsize (10x10x256) to (5x5x256)

        # max pooling layer
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # fully connected - - first linear layer (5x5x256) to (500)
        self.fc1 = nn.Linear( 256 * 5 * 5, 500)

        # fully connected - - second linear layer (500 to num_classes)
        self.fc2 = nn.Linear(500, num_classes) # Last layer (output)

        # dropout layer (p=0.05)
```

```

        # self.dropout = dropout

    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(F.relu(self.conv4(x)))
        x = self.pool(F.relu(self.conv5(x)))

        # flatten image input (transfer 2d image from CNN to linear layer)
        x = x.view(-1, 5 * 5 * 256)

        # add 1st hidden layer, with relu activation function
        x = F.relu(self.fc1(x))

        # add 2nd hidden layer, with relu activation function
        x = self.fc2(x)

    return x

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

# instantiate the CNN
model_scratch = Net()

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

```

```

In [277]: ## Check the model
print(model_scratch)

```

```

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)
  (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:

Input: 224x224x3, Kernel Size: 3x3, Padding: 1, dropout = 0.25, num_classes = 133

Convolutional_layer_1: load in (224x224x3) tensors and convert to (222x222x16) tensors

Maxpooling: convert the output from Convolutional_layer_1 to (111x111x16)

Convolutional_layer_2: load in (111x111x16) tensors and convert to (109x109x32) tensors

Maxpooling: convert the output from Convolutional_layer_2 to (54x54x32)

Convolutional_layer_3: load in (54x54x32) tensors and convert to (52x52x64) tensors

Maxpooling: convert the output from Convolutional_layer_3 to (26x26x64)

Convolutional_layer_4: load in (26x26x64) tensors and convert to (24x24x128) tensors

Maxpooling: convert the output from Convolutional_layer_4 to (12x12x128)

Convolutional_layer_5: load in (12x12x128) tensors and convert to (10x10x256) tensors

Maxpooling: convert the output from Convolutional_layer_5 to (5x5x256)

Flatten: image input and feed it into 2 hidden layers, with ReLu activation.

Last layer (output) - 133 different labels, the output should include 133 classes (num_classes)

(IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [278]: import torch.optim as optim

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

          # Create Parameter == Learning Rate
          learning_rate = 0.05

          ### TODO: select optimizer
          # Alternate 1
          # optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=learning_

          ## ALTERNATE 2
          optimizer_scratch = optim.Adam(model_scratch.parameters())
```

(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) (<http://pytorch.org/docs/master/notes/serialization.html>) at filepath 'model_scratch.pt' .

```
In [279]: # the following import is required for training to be robust to truncate
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

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

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

            ## Step One - clear the gradients
            optimizer.zero_grad()

            # Step Two - forward pass through the model
            output = model(data)

            # Step Three - Calculate the batch loss
            loss = criterion(output, target)

            # Step Four - Calculate backward pass
            loss.backward()

            # Step Five - perform a single optimization
            optimizer.step()

            # Step Six - update training loss
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.dat

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

```

(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

    # Step One - clear the gradient
    ## NO NEED

    # Step Two - forward pass
    output = model(data)

    # Step Three - calculate the batch loss
    loss = criterion(output, target)

    # Step Four - backward pass
    ## NO NEED

    # Step Five - optimization
    ## NO NEED

    # Step Six - update validation loss
    valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data

# Calculate average losses
train_loss = train_loss/len(train_loader.dataset)
valid_loss = valid_loss/len(valid_loader.dataset)

# print training/validation statistics
print('\n Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {
    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
        valid_loss_min,
        valid_loss))
    torch.save(model.state_dict(), save_path)
    valid_loss_min = valid_loss

# return trained model
return model

```

```
In [280]: # Set Number of epochs
n_epochs = 17

# train the model
model_scratch = train(n_epochs, loaders_scratch, model_scratch, optimize
                      criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 13      Training Loss: 0.000586      Validation Loss: 0.004
431
```

```
Validation loss decreased (0.004553 --> 0.004431). Saving model ...
```

```
Batch 1, Loss: 4.298153
Batch 101, Loss: 3.844600
Batch 201, Loss: 3.859962
Batch 301, Loss: 3.848164
```

```
Epoch: 14      Training Loss: 0.000576      Validation Loss: 0.004
461
```

```
Batch 1, Loss: 3.838127
Batch 101, Loss: 3.757249
Batch 201, Loss: 3.791903
Batch 301, Loss: 3.800731
```

```
Epoch: 15      Training Loss: 0.000569      Validation Loss: 0.004
358
```

```
Validation loss decreased (0.004431 --> 0.004358). Saving model ...
```

```
In [281]: # load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

(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 [282]: 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
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - te
        # 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))))
        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: 3.702281

Test Accuracy: 12% (103/836)

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 [283]: ## 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 [287]: import torchvision.models as models
import torch.nn as nn

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

# freeze all pretrained model parameters
for param in model_transfer.parameters():
    param.requires_grad = False

# get the last layer of the model
n_inputs = model_transfer.fc.in_features

# Assigning last layer of the transferred model output with 133
last_layer = nn.Linear(n_inputs, 133, bias=True)

# Assign the output as last layer
model_transfer.fc = last_layer

for param in model_transfer.fc.parameters():
    param.require_grad = True

# check to see that your last layer produces the expected number of outp
print(model_transfer.fc.out_features)
print("*****")
print(model_transfer)

if use_cuda:
    model_transfer = model_transfer.cuda()
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    )
    (2): Bottleneck(
        (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bi

```

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 decide to keep most of the pre-trained parameters and just change the last layers to match the total out-feature as in this project.

Resnet50 is suitable for the current problem - because, i was reading a paper on it, and felt it was gaining attention. Reference: Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds. Masafumi Yamazaki, Akihiko Kasagi, et.al source - <https://arxiv.org/abs/1903.12650> (<https://arxiv.org/abs/1903.12650>)

(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) (<http://pytorch.org/docs/master/nn.html#loss-functions>) and [optimizer](http://pytorch.org/docs/master/optim.html) (<http://pytorch.org/docs/master/optim.html>). Save the chosen loss function as `criterion_transfer`, and the optimizer as `optimizer_transfer` below.

In [288]:

```
import torch.optim as optim

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

# Create Parameter == Learning Rate
learning_rate = 0.001

### TODO: select optimizer
optimizer_transfer = optim.SGD(model_transfer.parameters(), lr=learning_
```

(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) (<http://pytorch.org/docs/master/notes/serialization.html>) at filepath `'model_transfer.pt'`.


```
In [289]: n_epochs = 15

# train the model
model_transfer = train(n_epochs, loaders_transfer, model_transfer, optim
                       criterion_transfer, use_cuda, 'model_transfer.pt')

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

Batch 1, Loss: 3.334375
Batch 101, Loss: 3.353011
Batch 201, Loss: 3.338034

Epoch: 12      Training Loss: 0.000499      Validation Loss: 0.003
385
Validation loss decreased (0.003506 --> 0.003385).  Saving model ...

Batch 1, Loss: 3.208863
Batch 101, Loss: 3.240035
Batch 201, Loss: 3.235894
Batch 301, Loss: 3.230571

Epoch: 13      Training Loss: 0.000483      Validation Loss: 0.003
239
Validation loss decreased (0.003385 --> 0.003239).  Saving model ...

Batch 1, Loss: 2.704275
Batch 101, Loss: 3.142478
Batch 201, Loss: 3.150944
Batch 301, Loss: 3.140066
```

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

```
Test Loss: 2.512493
```

```
Test Accuracy: 65% (546/836)
```

(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 [291]: ### 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_n  
class_names = [item[4:].replace("_", " ") for item in loaders_transfer['train']]
```

```
In [292]: loaders_transfer['train'].dataset.classes[:10]
```

```
Out[292]: ['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 [293]: class_names[:10]
```

```
Out[293]: ['Affenpinscher',  
          'Afghan hound',  
          'Airedale terrier',  
          'Akita',  
          'Alaskan malamute',  
          'American eskimo dog',  
          'American foxhound',  
          'American staffordshire terrier',  
          'American water spaniel',  
          'Anatolian shepherd dog']
```

```
In [294]: from PIL import Image
import torchvision.transforms as transforms

def predict_breed_transfer(img_path):

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

    # Pre-process the image
    normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                     std=[0.229, 0.224, 0.225])

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

    # Get the image tensor
    img_tensor = transform(img)

    # Unsqueeze the image
    # Why? refer above - to add one more column, make compatible
    img_tensor.unsqueeze_(0)

    # Send the image to GPU, if available
    if use_cuda:
        img_tensor = img_tensor.cuda()

    # Send the image through the pre-trained model
    output = model_transfer(img_tensor)

    _, prediction = torch.max(output.data,1)

    # Get the predicted breed name for the image
    breed_name = class_names[prediction-1]

    # Return the breed_name for the given image
    return breed_name
```

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 `dog_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!

```
hello, human!
```



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

(IMPLEMENTATION) Write your Algorithm

```
In [295]: ### 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) == True:
        prediction = predict_breed_transfer(img_path)
        print("Dogs Detected!\nIt looks like a: {}".format(prediction))
    elif face_detector(img_path) == True:
        prediction = predict_breed_transfer(img_path)
        print("Human Detected!\nYou look like a: {}".format(prediction))
    else:
        print("No Dogs or Humans Detected")
```

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: (Three possible points for improvement)

Output is not as I expected. I could do better. I can improve on my results via

1. More Training Datasets (more images)
2. Hyper-parameters tweaking (Dropout, Learning Rate, Optimizers, Batch Size)
3. More tweaking to pre-processed data - (Rotation, Center Crop, Shuffle, etc)

```
In [300]: ## 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.
test_figure = np.array(glob('myImages/*'))
test_figure_path = [i for i in test_figure]
#print(len(test_figure_path))
```

```
# suggested code, below
fig = plt.figure()
fig_index = 0
for i in range(len(test_figure_path)):

    file = test_figure_path[i]
    pred = run_app(file)

    fig_index += 1

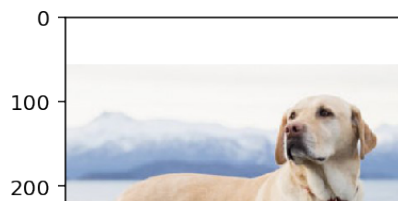
    img=np.asarray(Image.open(file))
```

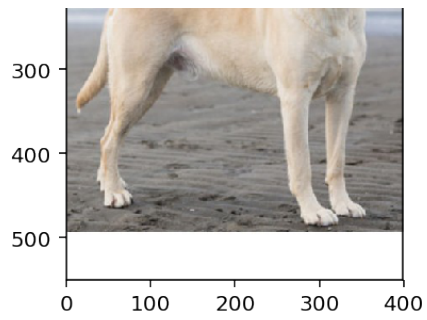


Dogs Detected!
It looks like a: Borzoi

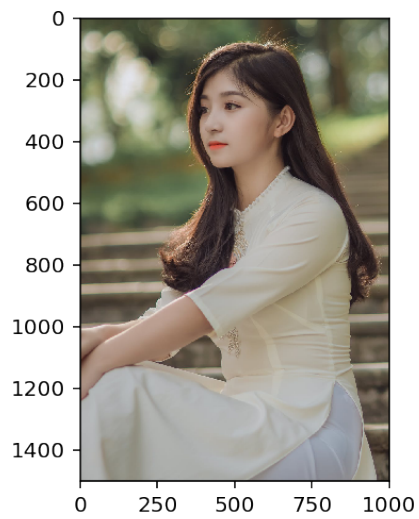


Human Detected!
You look like a: Black russian terrier





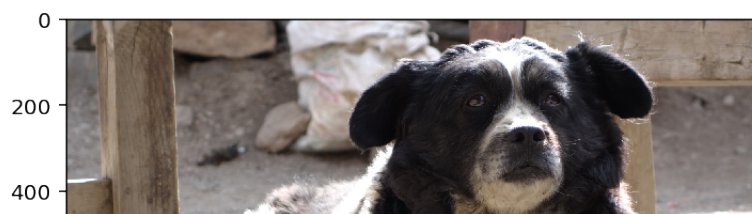
Dogs Detected!
It looks like a: Glen of imaal terrier



Human Detected!
You look like a: Lowchen



No Dogs or Humans Detected





No Dogs or Humans Detected

In []:

In []: