dog_app

July 8, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested. Sections that begin with **'(IMPLEMENTATION)'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section, and the specifics of the implementation are marked in the code block with a 'TODO' statement. Please be sure to read the instructions carefully!

> **Note**: Once you have completed all of the code implementations, you need to finalize your work by exporting the Jupyter Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to **File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

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

Step 0: Import Datasets

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

Note: if you are using the Udacity workspace, you *DO NOT* need to re-download these - they can be found in the `/data` folder as noted in the cell below.

- Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dog_images`.
- Download the human dataset. Unzip the folder and place it in the home directory, at location `/lfw`.

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

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

```
In [1]: import numpy as np
        from glob import glob

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

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

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

Step 1: Detect Humans

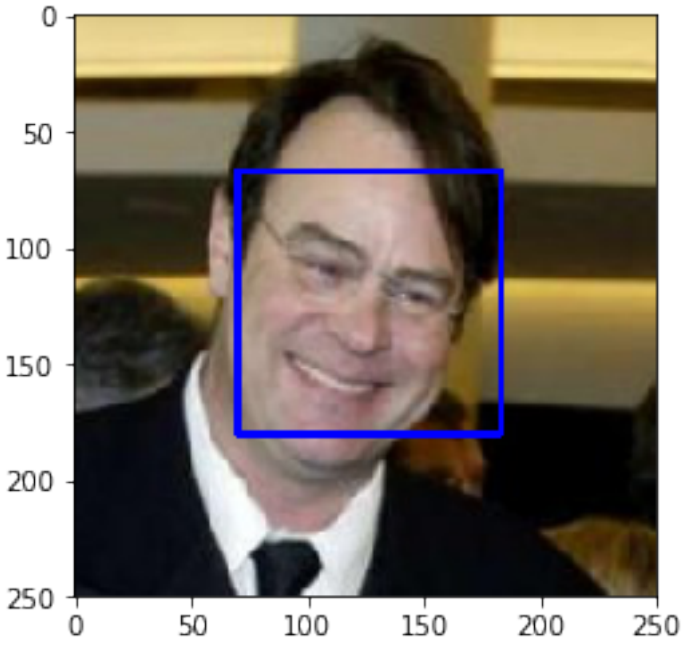
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

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

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

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

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

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

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

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

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

```
Number of faces detected: 1
```



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

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

1.1.1 Write a Human Face Detector

We can use this procedure to write a function that returns `True` if a human face is detected in an image and `False` otherwise. This function, aptly named `face_detector`, takes a string-valued file path to an image as input and appears in the code block below.

```
In [3]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray)
            return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer: Using the above implemented face_detector function: percentage of Human faces detected in Human files : 98% percenatge of Human faces detected in Dog files : 17%

```
In [4]: from tqdm import tqdm

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

        #-#-# Do NOT modify the code above this line. #-#-#

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.

        def detect(detector, files):
            # np.mean(list(map(detector, files))) # Or this method
            return np.mean([detector(f) for f in files])
        print('human: {:.2%}'.format(detect(face_detector, human_files_short)))
        print('dog: {:.2%}'.format(detect(face_detector, dog_files_short)))


human: 98.00%
dog: 17.00%
```

We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make use of deep learning :). Please use the code cell below to design and test your own face detection

algorithm. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

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

Step 2: Detect Dogs

In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

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

```
In [5]: import torch
        import torchvision.models as models

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:06<00:00, 91994498.12it/s]
```

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

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

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

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

In [10]:
```
from PIL import Image
import torchvision.transforms as transforms

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

    #transforms image to image_tensor as needed for input to VGG16 model.


    '''
    As described on https://pytorch.org/hub/pytorch_vision_vgg/
    All pre-trained models expect input images normalized in the same way,
    i.e. mini-batches of 3-channel RGB images of shape (3 x H x W),
    where H and W are expected to be at least 224.
    The images have to be loaded in to a range of [0, 1] and
    then normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225].
    So used this sample code right from there.
    '''

    input_image = Image.open(img_path)
    preprocess = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
    ])
    input_tensor = preprocess(input_image)
    input_batch = input_tensor.unsqueeze(0) # create a mini-batch as expected by the mo

    # move the input to GPU for speed if available
    if use_cuda:
        input_batch = input_batch.cuda()

    with torch.no_grad():
        output = VGG16(input_batch)
```

```
    # convert output to predicted index
    _, index = torch.max(output, 1)
    pred_index = np.squeeze(index.numpy()) if not use_cuda else np.squeeze(index.cpu().

    return int(pred_index) # predicted class index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary, you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from `'Chihuahua'` to `'Mexican hairless'`. Thus, in order to check to see if an image is predicted to contain a dog by the pre-trained VGG-16 model, we need only check if the pre-trained model predicts an index between 151 and 268 (inclusive).

Use these ideas to complete the `dog_detector` function below, which returns `True` if a dog is detected in an image (and `False` if not).

```
In [11]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             pred_index = VGG16_predict(img_path)

             return 151 <= pred_index <= 268 # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer:

percentage of Dog faces detected in Human files : 0% percenatge of Dog faces detected in Dog files : 100%

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

         print('human: {:.2%}'.format(detect(dog_detector, human_files_short)))
         print('dog: {:.2%}'.format(detect(dog_detector, dog_files_short)))
```

```
human: 0.00%
dog: 100.00%
```

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

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

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

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

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

In [13]:
```
import os
import random
import requests
import time
import ast
import numpy as np
from glob import glob
import cv2
from tqdm import tqdm
from PIL import Image, ImageFile

import torch
import torchvision
from torchvision import datasets
import torchvision.transforms as transforms
import torch.optim as optim
import torchvision.models as models

import matplotlib.pyplot as plt


ImageFile.LOAD_TRUNCATED_IMAGES = True

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

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


"""
Resource help for dataloaders and augmentation has been taken from
https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html
"""

# Data augmentation for training and normalization for validation and test
data_transforms = {
                        'train': transforms.Compose([
                            transforms.RandomResizedCrop(224),
                            transforms.RandomHorizontalFlip(),
                            transforms.RandomRotation(10),
                            transforms.ToTensor(),
                            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
                        ]),
                        'valid': transforms.Compose([
                            transforms.Resize(256),
                            transforms.CenterCrop(224),
                            transforms.ToTensor(),
```

```
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
        ]),
        'test': transforms.Compose([
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
        ]),
    }


data_dir = '/data/dog_images'

image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                          data_transforms[x])
                  for x in ['train', 'valid', 'test']}
loaders_scratch = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=16,
                                                  shuffle=True, num_workers=2)
                   for x in ['train', 'valid', 'test']}

dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'valid', 'test']}
```

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:

1. Training set has been rescaled to 224*224. I have used 224 as a resize value, so that it can be used in other prerained models too. Like VGG16 needs input image size to be at least 224.
2. Yes, I have augmented the training set through Random Flip and Rotation. Validation and test set is just normalized. This will reduce overfitting

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [14]: import torch.nn as nn
         import torch.nn.functional as F

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             """
             Resource help has been taken from below link :
             https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
             """
             def __init__(self):
                 super(Net, self).__init__()
```

```
        ## Define layers of a CNN
        self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
        self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
        self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
        self.conv4 = nn.Conv2d(256, 512, 3, padding=1)
        self.conv5 = nn.Conv2d(512, 1024, 3, padding=1)
        self.conv6 = nn.Conv2d(1024, 2048, 3, padding=1)
        self.conv7 = nn.Conv2d(2048, 4096, 3, padding=1)

        self.bn1 = nn.BatchNorm2d(64)
        self.bn2 = nn.BatchNorm2d(128)
        self.bn3 = nn.BatchNorm2d(256)
        self.bn4 = nn.BatchNorm2d(512)
        self.bn5 = nn.BatchNorm2d(1024)
        self.bn6 = nn.BatchNorm2d(2048)
        self.bn7 = nn.BatchNorm2d(4096)

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

        self.fc1 = nn.Linear(4096, 2048)
        self.fc2 = nn.Linear(2048, 133)

        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        ## Define forward behavior
        x = self.bn1(self.pool(F.relu(self.conv1(x))))
        x = self.bn2(self.pool(F.relu(self.conv2(x))))
        x = self.bn3(self.pool(F.relu(self.conv3(x))))
        x = self.bn4(self.pool(F.relu(self.conv4(x))))
        x = self.bn5(self.pool(F.relu(self.conv5(x))))
        x = self.bn6(self.pool(F.relu(self.conv6(x))))
        x = self.bn7(self.pool(F.relu(self.conv7(x))))

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

# move tensors to GPU if CUDA is available
if use_cuda:
```

```
model_scratch.cuda()
```

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

Answer:

Thinking of ResNet101, i tried the concept of using more stacked layers to captures fine details, so to gain a goood accuracy. Incorporating mentor suggestions ended up with having 7 conv2d layer, 1 maxpool layer, 2 fully connected layer and 1dropout layer. This can be tuned more to acheive a good accuracy.

Steps: 1. Added 7 conv2d layers with alternate BatchNormaliZation2D and dropout after each layer to overcome overfitting. 2. Added MaxPool2D layer to downsample as its best choice for this kind of classification. 3. Added FC layers to produce 133-dim output.

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_scratch`, and the optimizer as `optimizer_scratch` below.

```
In [15]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.02)
```

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

```
In [16]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf



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

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

```
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        ## find the loss and update the model parameters accordingly
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model paramet
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()

        ## record the average training loss, using something like
        train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - 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()

        # forward pass: compute predicted outputs by passing inputs to the model

        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        ## update the average validation loss
        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:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
        valid_loss_min,
        valid_loss))
        torch.save(model.state_dict(), save_path)
        valid_loss_min = valid_loss
```

```
        # return trained model
        return model


# train the model
model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 4.650801         Validation Loss: 4.376101
Validation loss decreased (inf --> 4.376101).  Saving model ...
Epoch: 2         Training Loss: 4.399663         Validation Loss: 4.081033
Validation loss decreased (4.376101 --> 4.081033).  Saving model ...
Epoch: 3         Training Loss: 4.216371         Validation Loss: 3.884148
Validation loss decreased (4.081033 --> 3.884148).  Saving model ...
Epoch: 5         Training Loss: 3.950665         Validation Loss: 3.908316
Epoch: 6         Training Loss: 3.838559         Validation Loss: 3.459666
Validation loss decreased (3.884148 --> 3.459666).  Saving model ...
Epoch: 7         Training Loss: 3.724799         Validation Loss: 3.504665
Epoch: 8         Training Loss: 3.625448         Validation Loss: 3.242447
Validation loss decreased (3.459666 --> 3.242447).  Saving model ...
Epoch: 9         Training Loss: 3.527767         Validation Loss: 3.145574
Validation loss decreased (3.242447 --> 3.145574).  Saving model ...
Epoch: 10         Training Loss: 3.426196         Validation Loss: 3.834436
Epoch: 11         Training Loss: 3.308857         Validation Loss: 3.037142
Validation loss decreased (3.145574 --> 3.037142).  Saving model ...
Epoch: 12         Training Loss: 3.285589         Validation Loss: 2.751651
Validation loss decreased (3.037142 --> 2.751651).  Saving model ...
Epoch: 13         Training Loss: 3.177968         Validation Loss: 2.834353
Epoch: 14         Training Loss: 3.087807         Validation Loss: 2.577734
Validation loss decreased (2.751651 --> 2.577734).  Saving model ...
Epoch: 15         Training Loss: 2.989716         Validation Loss: 2.603860
Epoch: 16         Training Loss: 2.924501         Validation Loss: 2.502494
Validation loss decreased (2.577734 --> 2.502494).  Saving model ...
Epoch: 17         Training Loss: 2.847238         Validation Loss: 2.448747
Validation loss decreased (2.502494 --> 2.448747).  Saving model ...
Epoch: 18         Training Loss: 2.786272         Validation Loss: 2.333966
Validation loss decreased (2.448747 --> 2.333966).  Saving model ...
Epoch: 19         Training Loss: 2.737456         Validation Loss: 2.204324
Validation loss decreased (2.333966 --> 2.204324).  Saving model ...
Epoch: 20         Training Loss: 2.642419         Validation Loss: 2.437210
```

1.1.11 (IMPLEMENTATION) Test the Model

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

```
In [17]: def test(loaders, model, criterion, use_cuda):

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

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

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

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

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

```
Test Loss: 2.240855


Test Accuracy: 38% (325/836)
```

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

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

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

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

```
In [18]: ## TODO: Specify data loaders
         loaders_transfer = loaders_scratch
```

1.1.13 (IMPLEMENTATION) Model Architecture

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

```
In [19]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture
         model_transfer = models.resnet101(pretrained=True)

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

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

         for param in fc_parameters:
             param.requires_grad = True


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

```
Downloading: "https://download.pytorch.org/models/resnet101-5d3b4d8f.pth" to /root/.torch/models
100%|| 178728960/178728960 [00:02<00:00, 78337779.83it/s]
```

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:

Steps: 1. import resnet101 model 2. just change its final FC layer to meet our requiremnts to output to 133 classes. 3. Used CrossEntropyLoss and SGD optimizer with learning rate of 0.001

I used resnet101 for thi task because resNet gains good accuracy with increased depths. As it has 101 layers, it capures very fine details. And it also easy to optimize. Now as its already trained on huge dataset, its an advantage of transfer learning to already have base weights. This results in good accuracy.

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [20]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr=0.001)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [21]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf



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

                 ###################
                 # train the model #
                 ###################
                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     ## find the loss and update the model parameters accordingly
                     # clear the gradients of all optimized variables
                     optimizer.zero_grad()
                     # forward pass: compute predicted outputs by passing inputs to the model
                     output = model(data)
                     # calculate the batch loss
                     loss = criterion(output, target)
                     # backward pass: compute gradient of the loss with respect to model paramet
                     loss.backward()
                     # perform a single optimization step (parameter update)
                     optimizer.step()

                     ## record the average training loss, using something like
                     train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - 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()

                    # forward pass: compute predicted outputs by passing inputs to the model

                    output = model(data)
                    # calculate the batch loss
                    loss = criterion(output, target)
                    ## update the average validation loss
                    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:
                    print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
                    valid_loss_min,
                    valid_loss))
                    torch.save(model.state_dict(), save_path)
                    valid_loss_min = valid_loss

            # return trained model
            return model
        # train the model
        model_transfer = train(15, loaders_transfer, model_transfer, optimizer_transfer, criter

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

```
Epoch: 1         Training Loss: 4.799244         Validation Loss: 4.594285
Validation loss decreased (inf --> 4.594285).  Saving model ...
Epoch: 2         Training Loss: 4.529651         Validation Loss: 4.277665
Validation loss decreased (4.594285 --> 4.277665).  Saving model ...
Epoch: 3         Training Loss: 4.301835         Validation Loss: 3.998369
Validation loss decreased (4.277665 --> 3.998369).  Saving model ...
Epoch: 4         Training Loss: 4.089093         Validation Loss: 3.717441
Validation loss decreased (3.998369 --> 3.717441).  Saving model ...
Epoch: 5         Training Loss: 3.878784         Validation Loss: 3.506076
```

```
Validation loss decreased (3.717441 --> 3.506076).  Saving model ...
Epoch: 6         Training Loss: 3.680005         Validation Loss: 3.228996
Validation loss decreased (3.506076 --> 3.228996).  Saving model ...
Epoch: 7         Training Loss: 3.502679         Validation Loss: 3.064612
Validation loss decreased (3.228996 --> 3.064612).  Saving model ...
Epoch: 8         Training Loss: 3.335519         Validation Loss: 2.867664
Validation loss decreased (3.064612 --> 2.867664).  Saving model ...
Epoch: 9         Training Loss: 3.184310         Validation Loss: 2.647885
Validation loss decreased (2.867664 --> 2.647885).  Saving model ...
Epoch: 10         Training Loss: 3.046098         Validation Loss: 2.483310
Validation loss decreased (2.647885 --> 2.483310).  Saving model ...
Epoch: 11         Training Loss: 2.895712         Validation Loss: 2.323523
Validation loss decreased (2.483310 --> 2.323523).  Saving model ...
Epoch: 12         Training Loss: 2.786886         Validation Loss: 2.178687
Validation loss decreased (2.323523 --> 2.178687).  Saving model ...
Epoch: 13         Training Loss: 2.679495         Validation Loss: 2.135894
Validation loss decreased (2.178687 --> 2.135894).  Saving model ...
Epoch: 14         Training Loss: 2.582039         Validation Loss: 2.000326
Validation loss decreased (2.135894 --> 2.000326).  Saving model ...
Epoch: 15         Training Loss: 2.490747         Validation Loss: 1.853126
Validation loss decreased (2.000326 --> 1.853126).  Saving model ...
```

1.1.16 (IMPLEMENTATION) Test the Model

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

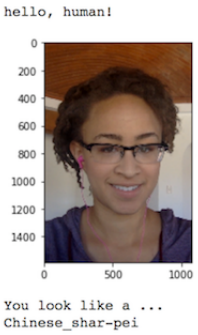
```
In [22]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.802369


Test Accuracy: 74% (626/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan hound`, etc) that is predicted by your model.

```
In [23]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.
         data_transfer = image_datasets
         # list of class names by index, i.e. a name can be accessed like class_names[0]
         class_names = [item[4:].replace("_", " ") for item in data_transfer['train'].classes]

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
```



Sample Human Output

```
input_image = Image.open(img_path)
preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])
input_tensor = preprocess(input_image)
input_batch = input_tensor.unsqueeze(0) # create a mini-batch as expected by the mo

# move the input to GPU for speed if available
if use_cuda:
    input_batch = input_batch.cuda()

with torch.no_grad():
    output = model_transfer(input_batch)

# convert output to predicted index
index = torch.max(output,1)[1].item()

return class_names[index]
```

Step 5: Write your Algorithm

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

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

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

1.1.18 (IMPLEMENTATION) Write your Algorithm

In [24]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
def plot_image(img_path):
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()


def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    if face_detector(img_path):
        print ("hello, human!")
        predicted_breed = predict_breed_transfer(img_path)
        plot_image(img_path)
        print("You look like a ...: ",predicted_breed)

    elif dog_detector(img_path):
        print ("hello, Doggie!")
        predicted_breed = predict_breed_transfer(img_path)
        plot_image(img_path)
        print("You look like a ...: ",predicted_breed)

    else:
        print ("Unable to identify...")
        plot_image(img_path)
```

Step 6: Test Your Algorithm

In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that *you* look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

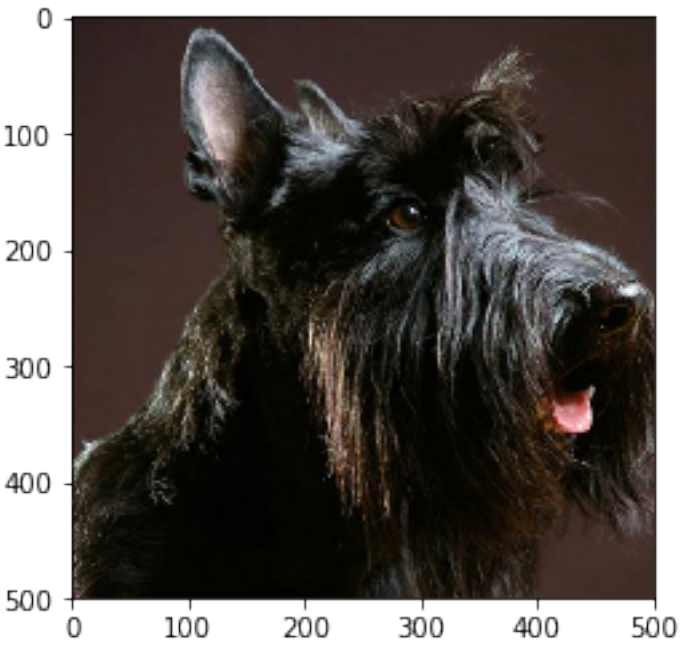
Answer: (Three possible points for improvement)

Improvement points: 1. proper image preprocessing needs to be done. i.e not only resizing or flip. 2. More data needs for training and validation. 3. Hyper parameter tuning. As during training i tried running with batch size 32, with learning rate 0.001, and performance decreased. Where as with batch size 16 and learning rate 0.02, it gave good result.

In [25]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
```

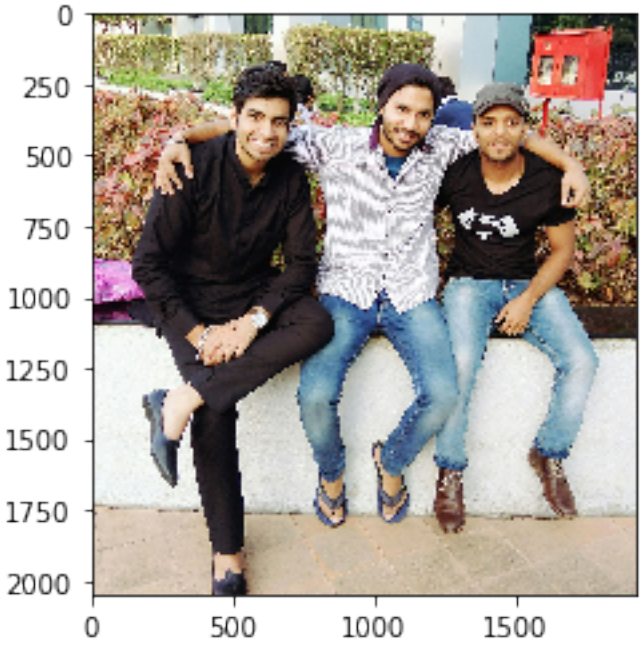
```
## Feel free to use as many code cells as needed.

import os
import glob
img_dir = "testImages" # Enter Directory of all images
data_path = os.path.join(img_dir,'*g')
files = glob.glob(data_path)
## suggested code, below
for file in files:
    run_app(file)
```

```
hello, Doggie!
```

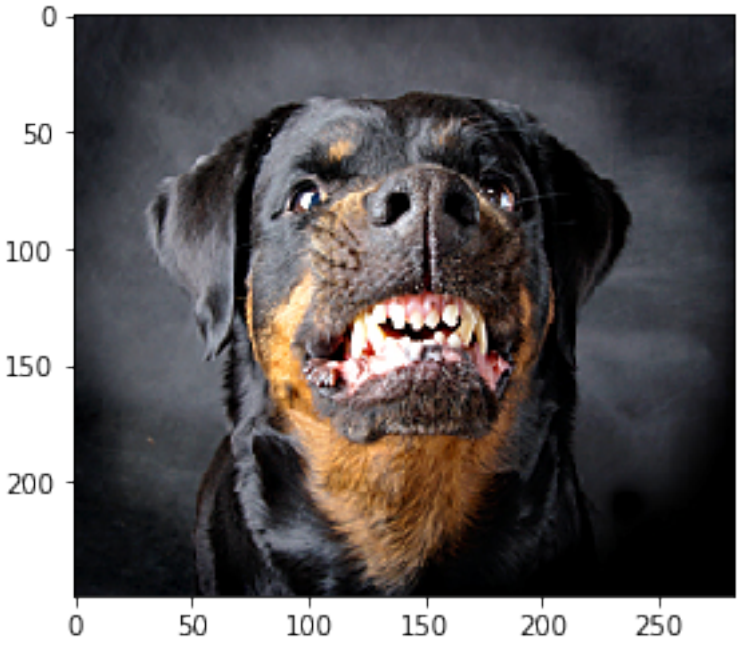



```
You look like a ...:  Affenpinscher
hello, human!
```

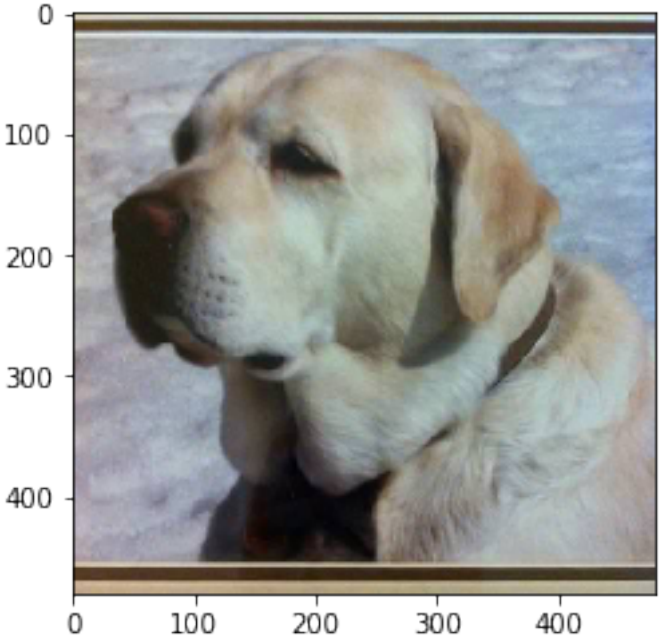

```
You look like a ...:  Akita
hello, human!
```



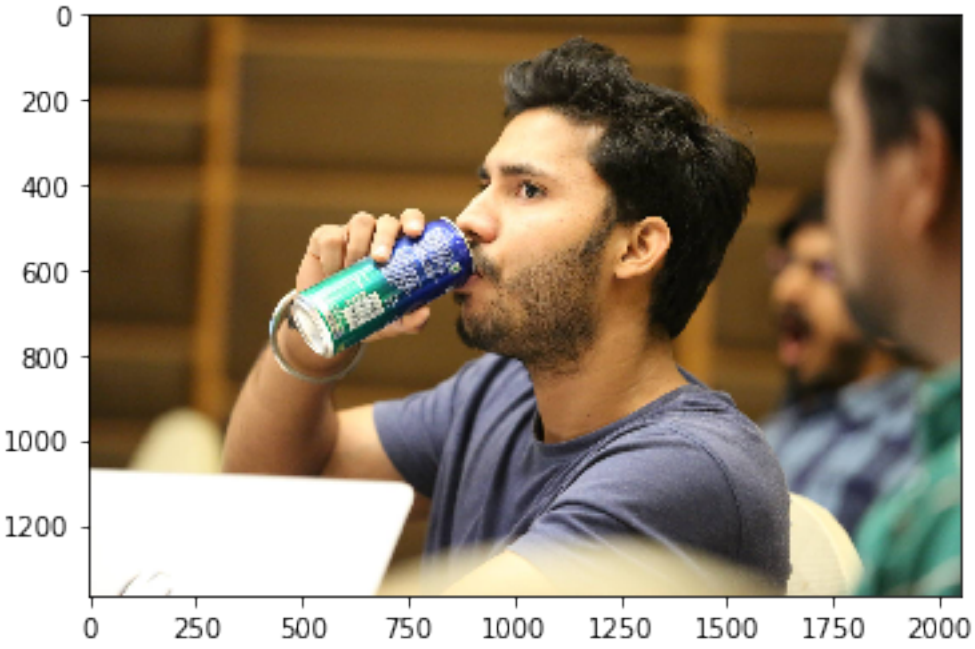
```
You look like a ...:  Dalmatian
hello, Doggie!
```




```
You look like a ...:  Beauceron
hello, Doggie!
```

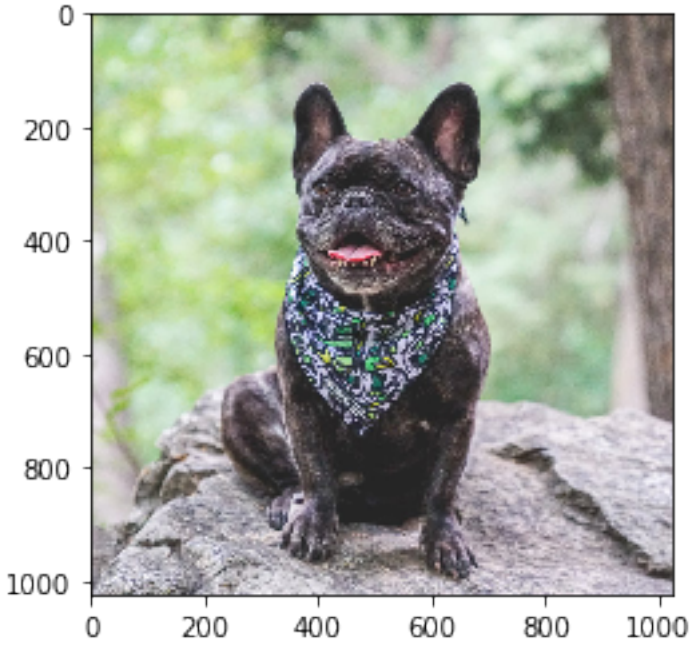


```
You look like a ...:  Golden retriever
Unable to identify...
```




```
hello, human!
```


```
You look like a ...:  Bull terrier
hello, Doggie!
```



```
You look like a ...:  French bulldog
```

```
In [27]: """
         References :
             1. https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
             2. https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html
             3. https://pytorch.org/docs/stable/nn.html#linear-layers
         """
```

```
Out[27]: '\nReferences :\n    1. https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.h
```

```
In [ ]:
```