Convolutional Neural Networks

Project: Write an Algorithm for Landmark Classification

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 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 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.

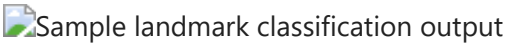
The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.

Why We're Here

Photo sharing and photo storage services like to have location data for each photo that is uploaded. With the location data, these services can build advanced features, such as automatic suggestion of relevant tags or automatic photo organization, which help provide a compelling user experience. Although a photo's location can often be obtained by looking at the photo's metadata, many photos uploaded to these services will not have location metadata available. This can happen when, for example, the camera capturing the picture does not have GPS or if a photo's metadata is scrubbed due to privacy concerns.

If no location metadata for an image is available, one way to infer the location is to detect and classify a discernible landmark in the image. Given the large number of landmarks across the world and the immense volume of images that are uploaded to photo sharing services, using human judgement to classify these landmarks would not be feasible.

In this notebook, you will take the first steps towards addressing this problem by building models to automatically predict the location of the image based on any landmarks depicted in the image. At the end of this project, your code will accept any user-supplied image as input and suggest the top k most relevant landmarks from 50 possible landmarks from across the world. The image below displays a potential sample output of your finished project.

 Sample landmark classification output

The Road Ahead

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

Step 0: Download Datasets and Install Python Modules

Note: if you are using the Udacity workspace, *YOU CAN SKIP THIS STEP*. The dataset can be found in the `/data` folder and all required Python modules have been installed in the workspace.

Download the landmark dataset. Unzip the folder and place it in this project's home directory, at the location `/landmark_images`.

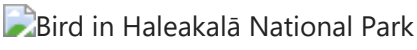
Install the following Python modules:

- cv2
- matplotlib
- numpy
- PIL
- torch
- torchvision

Step 1: Create a CNN to Classify Landmarks (from Scratch)

In this step, you will create a CNN that classifies landmarks. 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 20%.

Although 20% may seem low at first glance, it seems more reasonable after realizing how difficult of a problem this is. Many times, an image that is taken at a landmark captures a fairly mundane image of an animal or plant, like in the following picture.


Bird in Haleakalā National Park

Just by looking at that image alone, would you have been able to guess that it was taken at the Haleakalā National Park in Hawaii?

An accuracy of 20% is significantly better than random guessing, which would provide an accuracy of just 2%. In Step 2 of this notebook, you will have the opportunity to greatly improve accuracy by using transfer learning to create a CNN.

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

(IMPLEMENTATION) Specify Data Loaders for the Landmark Dataset

Use the code cell below to create three separate data loaders: one for training data, one for validation data, and one for test data. Randomly split the images located at `landmark_images/train` to create the train and validation data loaders, and use the images located at `landmark_images/test` to create the test data loader.

All three of your data loaders should be accessible via a dictionary named `loaders_scratch`. Your train data loader should be at `loaders_scratch['train']`, your validation data loader should be at `loaders_scratch['valid']`, and your test data loader should be at `loaders_scratch['test']`.

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 [ ]:
```
# import stuff

%matplotlib inline
%config InlineBackend.figure_format = 'retina'

import os

import matplotlib.pyplot as plt

import torch
nn = torch.nn
F = torch.nn.functional
print('pytorch version: ',torch.__version__)

from torchvision import datasets, transforms

import numpy as np
```

```
np.random.seed(13) #static random numbers

import cv2

import splitfolders


import inspect
from datetime import datetime
```

```
pytorch version:  1.9.0+cu111
```

In [ ]:
```
torch.cuda.empty_cache()
```

In [ ]:
```
try:
    import playsound
except:
    print('no playsound module')


def training_done():
    try:
        playsound.playsound('Krypton.mp3')
    except:
        print('no sound')
```

In [ ]:
```
data_train = 'landmark_images/train' #this is where the raw data is

dir_train = 'images/train'
dir_test = 'images/test'
dir_val= 'images/val'

if os.path.exists(dir_train) == False:
    splitfolders.ratio(data_train, output="images", seed=69, ratio=(.8, .1, .1), group_
```

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

# mean = [0.485, 0.456, 0.406]
# std = [0.229, 0.224, 0.225]

size = 128
mean = np.array([0.485, 0.456, 0.406])
std = np.array([0.229, 0.224, 0.225])

#https://pytorch.org/vision/stable/transforms.html
# (0.485, 0.456, 0.406), (0.229, 0.224, 0.225)

# train and valid
# maybe add a new more transforms
# https://pytorch.org/vision/stable/transforms.html
# transform0 = transforms.Compose([
#     transforms.RandomRotation(10),
#     transforms.Resize(size=(512,512)),
#     transforms.RandomResizedCrop(size=(256,256)),
#     # transforms.Resize(size=(128,128)),
```

```
#     # transforms.Resize(size=(64,64)),
#     # transforms.Grayscale(),
#     # transforms.RandomAdjustSharpness(1, p=1),
#     # transforms.RandomVerticalFlip(),
#     transforms.RandomHorizontalFlip(p=0.5),
#     transforms.ToTensor(),
#     # transforms.Normalize([0.5]*3,[0.5]*3),
#     ])

# # test
# transform1 = transforms.Compose([
#     # transforms.Resize(300),
#     transforms.Resize(size=(256,256)),
#     # transforms.Resize(size=(128,128)),
#     # transforms.Resize(size=(64,64)),
#     # transforms.Grayscale(),
#     transforms.ToTensor(),
#     # transforms.Normalize([0.5]*3,[0.5]*3),
#     ])

transform0 = transforms.Compose([
    transforms.RandomRotation(10),
    transforms.RandomResizedCrop(size),
    transforms.RandomHorizontalFlip(p=0.1),
    transforms.ColorJitter(brightness=0.05,contrast=0.05,saturation=0.5,hue=0.05),
    # transforms.ColorJitter(contrast=0.05),
    # transforms.ColorJitter(saturation=0.05),
    # transforms.ColorJitter(hue=0.05),
    transforms.ToTensor(),
    transforms.Normalize(mean,std)
])

transform1 = transforms.Compose([
    transforms.CenterCrop((size, size)),
    transforms.ToTensor(),
    transforms.Normalize(mean,std)
])

loaders_scratch = {}
loaders = [
        {
            'name':'train',
            'dir': dir_train,
            'transform': transform0,
        },
        {
            'name':'valid',
            'dir': dir_val,
            'transform': transform1,
        },
        {
            'name':'test',
            'dir': dir_test,
            'transform': transform1,
        },
    ]


for l in loaders:
    dataset = datasets.ImageFolder(root=l['dir'], transform=l['transform'])
    print(l['name'],len(dataset))
```

```
    loaders_scratch[l['name']] = torch.utils.data.DataLoader(dataset, batch_size=64, sh


print(*loaders_scratch)
```

```
train 3996
valid 499
test 501
train valid test
```

In [ ]:

```
dataset = datasets.ImageFolder(root=dir_train, transform=transform0)
# print('\n'.join(dataset.classes))

label_map = list(dataset.classes)
print(*label_map,sep='\n')
```

```
00.Haleakala_National_Park
01.Mount_Rainier_National_Park
02.Ljubljana_Castle
03.Dead_Sea
04.Wroclaws_Dwarves
05.London_Olympic_Stadium
06.Niagara_Falls
07.Stonehenge
08.Grand_Canyon
09.Golden_Gate_Bridge
10.Edinburgh_Castle
11.Mount_Rushmore_National_Memorial
12.Kantanagar_Temple
13.Yellowstone_National_Park
14.Terminal_Tower
15.Central_Park
16.Eiffel_Tower
17.Changdeokgung
18.Delicate_Arch
19.Vienna_City_Hall
20.Matterhorn
21.Taj_Mahal
22.Moscow_Raceway
23.Externsteine
24.Soreq_Cave
25.Banff_National_Park
26.Pont_du_Gard
27.Seattle_Japanese_Garden
28.Sydney_Harbour_Bridge
29.Petronas_Towers
30.Brooklyn_Bridge
31.Washington_Monument
32.Hanging_Temple
33.Sydney_Opera_House
34.Great_Barrier_Reef
35.Monumento_a_la_Revolucion
36.Badlands_National_Park
37.Atomium
38.Forth_Bridge
39.Gateway_of_India
40.Stockholm_City_Hall
41.Machu_Picchu
42.Death_Valley_National_Park
43.Gullfoss_Falls
```

```
44.Trevi_Fountain
45.Temple_of_Heaven
46.Great_Wall_of_China
47.Prague_Astronomical_Clock
48.Whitby_Abbey
49.Temple_of_Olympian_Zeus
```

resource for rendering image with plt

https://github.com/udacity/deep-learning-v2-pytorch/blob/master/convolutional-neural-networks/cifar-cnn/cifar10_cnn_solution.ipynb

resouec for splitting the training and validation

https://stackoverflow.com/questions/53074712/how-to-split-folder-of-images-into-test-training-validation-sets-with-stratified

In [ ]:

```
# helper function from
# https://github.com/udacity/deep-learning-v2-pytorch/blob/master/convolutional-neural-

# helper function to un-normalize and display an image
def imshow(image):
    # img = img / 2 #+ 0.5  # unnormalize ..don't need this line
    image = image.numpy() * std[:, None, None] + mean[:, None, None]
    plt.imshow(np.transpose(image, (1, 2, 0)))  # convert from Tensor image

def show_images(images,labels,title=None,figsize=(20,5) ,image_limit = 10,color_map=Non

    # unnormalize image
    images = images.numpy() * std[:, None, None] + mean[:, None, None]
    fig = plt.figure(figsize=figsize,facecolor='white')
    fig.suptitle(title, fontsize=16)

    rows = int(np.ceil(image_limit/5))

    for idx in range(image_limit):
        ax = fig.add_subplot(rows, 5, idx+1, xticks=[], yticks=[])
        # imshow(images[idx])


        plt.imshow(np.transpose(images[idx], (1, 2, 0)),cmap=color_map)

        ax.set_title(label_map[labels[idx].item()], fontsize = 12)
```

Question 1: 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:

- How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why?
    - i'm using the transforms within the torchvision library
        - i'm using
            - random rotation -10 and 10
            - random resize and crop (256 size)
            - random horizontal flip ~~* grey scaling them .. i don't think color will matter that much~~
            - random color jitter (brightness,contrast, etc)
            - and normalize ~~* random vertical flip~~
    - i'm cropping to a 256 square, because it's a square and divisible by 2.
        - we need to make sure all images are the same size for our NN, since our inputs have to be the same length.
- Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?
    - yes, we are rotating a bit, resize cropping and flipping... this makes it so our AI doesn't get too used to the same exact image.

(IMPLEMENTATION) Visualize a Batch of Training Data

Use the code cell below to retrieve a batch of images from your train data loader, display at least 5 images simultaneously, and label each displayed image with its class name (e.g., "Golden Gate Bridge").

Visualizing the output of your data loader is a great way to ensure that your data loading and preprocessing are working as expected.

In [ ]:

```
import matplotlib.pyplot as plt
%matplotlib inline

## TODO: visualize a batch of the train data loader

## the class names can be accessed at the `classes` attribute
## of your dataset object (e.g., `train_dataset.classes`)

# train data

images, labels = next(iter(loaders_scratch['train']))
show_images(images,labels,title='Train',color_map=None)

# images, labels = next(iter(loaders_scratch['test']))
# show_images(images,labels,title='Test')

## used for random...but we are already shuffling
# r = np.random.choice(range(images.shape[0]))
# print(r)
# print(images[r].shape)
# imshow(images[r])
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
[0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
[0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
[0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
[0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
[0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
[0..255] for integers).
```

Initialize use_cuda variable

Steps

1. download & install the cuda toolkit
   - https://developer.nvidia.com/cuda-toolkit-archive
   - most newer versions should be fine
2. get the specific version of pytorch
   - https://pytorch.org/get-started/locally/
   - pick your OS, Cuda version, and etc
   - run the provided command

notes:

Cuda is only for specific Nvidia GPUs

In [ ]:
```
# useful variable that tells us whether we should use the GPU
use_cuda = torch.cuda.is_available()
print(use_cuda)
```

```
True
```

(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 fill in the function `get_optimizer_scratch` below.

In [ ]:
```
## TODO: select loss function
criterion_scratch = None

# #i do not have inputs of targets, yet
```

```
# #so i'll use this function when the time comes
# def get_criterion_scratch(input,target):
#     criterion_scratch = F.nll_loss(input=input,target=target)

# we used this one in the lesson
criterion_scratch = nn.CrossEntropyLoss()

#others to try
# criterion_scratch = nn.L1Loss()
# criterion_scratch = nn.NLLLoss()
# criterion_scratch = nn.MSELoss()
# criterion_scratch = nn.HingeEmbeddingLoss()


# def get_optimizer_scratch(model, learn_rate=0.001):
def get_optimizer_scratch(model):
    ## TODO: select and return an optimizer
    # optimizer = torch.optim.SGD(model.parameters(), lr = 0.001)

    # others to try
    # optimizer = torch.optim.RMSprop(model.parameters(),lr=0.001)
    # optimizer = torch.optim.Adadelta(model.parameters(),lr=0.001)
    optimizer = torch.optim.Adam(model.parameters(),lr=0.001)
    # optimizer = torch.optim.Adagrad(model.parameters(),lr=0.001)

    return optimizer
```

(IMPLEMENTATION) Model Architecture

Create a CNN to classify images of landmarks. Use the template in the code cell below.

In [ ]:

```
import torch.nn as nn

# 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

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

        # # input is 3 (RGB)
        # # output channels to 9 (how thick out tensor will be)
        # # kernel is (3,3) ...so we will be looking at a 3x3 pixels at a time
        # self.conv1 = nn.Conv2d(3,64,kernel_size=(3,3),stride=1,padding=1)
        # self.conv2 = nn.Conv2d(64,64,kernel_size=(3,3),stride=1,padding=1)
        # self.conv3 = nn.Conv2d(64,128,kernel_size=(3,3),stride=1,padding=1)
        # self.conv4 = nn.Conv2d(128,128,kernel_size=(3,3),stride=1,padding=1)
        # self.conv5 = nn.Conv2d(128,256,kernel_size=(3,3),stride=1,padding=1)
        # self.conv6 = nn.Conv2d(256,256,kernel_size=(3,3),stride=1,padding=1)
        # self.conv7 = nn.Conv2d(256,512,kernel_size=(3,3),stride=1,padding=1)
        # self.conv8 = nn.Conv2d(512,512,kernel_size=(3,3),stride=1,padding=1)

        self.conv1 = nn.Conv2d(3,16,kernel_size=(3,3),stride=1,padding=1)
```

```
        self.conv2 = nn.Conv2d(16,32,kernel_size=(3,3),stride=1,padding=1)
        self.conv3 = nn.Conv2d(32,64,kernel_size=(3,3),stride=1,padding=1)
        # self.conv4 = nn.Conv2d(64,128,kernel_size=(3,3),stride=1,padding=1)

        # self.fc1 = nn.Linear(131072,65536)
        # self.fc2 = nn.Linear(65536,50)

        self.fc1 = nn.Linear(16384,256)
        self.fc2 = nn.Linear(256,128)
        self.fc3 = nn.Linear(128,50)

        self.dropout = nn.Dropout(0.2)

        self.printed_x = False

    def print_forward(self):
        return str(inspect.getsource(self.forward))

    def print_model(self):
        return str(self)

    def forward(self, x):
        ## Define forward behavior

        x = self.conv1(x)
        x = F.leaky_relu(x)
        x = self.pool(x)

        x = self.conv2(x)
        x = F.leaky_relu(x)
        x = self.pool(x)

        x = self.conv3(x)
        x = F.leaky_relu(x)
        x = self.pool(x)

        # x = self.conv4(x)
        # x = self.pool(x)
        # x = F.leaky_relu(x)

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

        if self.printed_x == False:
            print('x.shape: ',x.shape)
            self.printed_x = True
        #needed to find the correct shape for line ^24

        x = self.dropout(x)
        x = F.leaky_relu(self.fc1(x),negative_slope=0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc2(x),negative_slope=0.2)
        x = self.dropout(x)
        x = F.leaky_relu(self.fc3(x),negative_slope=0.2)

        # x = F.relu(self.fc2(x))
        x = F.log_softmax(x,dim=1)

        return x
```

```
#-#-# Do NOT 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 2: Outline the steps you took to get to your final CNN architecture and your reasoning at each step.

Answer:
my first few models just doubled the convultional layer's output and then had one fully connected layer, however this did not give me over 20%.

after looking at vgg and resnet i tried to mimic them. if these models had good scores and i should be able to mimic them and get a better output myself. so i added more fully connected layers, pools and dropouts.

after from trail and error i got to a 28% accuracy on the test function.

(IMPLEMENTATION) Implement the Training Algorithm

Implement your training algorithm in the code cell below. Save the final model parameters at the filepath stored in the variable `save_path`.

In [ ]:

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

            if use_cuda:
                torch.cuda.empty_cache()
                data, target = data.cuda(), target.cuda()

            ## TODO: find the loss and update the model parameters accordingly
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data.item() - t

            # this resets before back propagation
            optimizer.zero_grad()
```

```
            #get model output
            output = model(data)

            #calculate loss
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

            #train based on loss
            train_loss += ((1 / (batch_idx + 1)) * (loss.data.item() - train_loss))


        ######################
        # validate the model #
        ######################
        # set the model to evaluation mode
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                torch.cuda.empty_cache()
                data, target = data.cuda(), target.cuda()

            ## TODO: update average validation loss
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # update average validation loss
            valid_loss += ((1 / (batch_idx + 1)) * (loss.data.item() - valid_loss))


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

        ## TODO: if the validation loss has decreased, save the model at the filepath s
        if valid_loss <= valid_loss_min:

            # print('valid_loss: {:.10f}'.format(valid_loss))
            valid_loss_min = valid_loss
            print('valid_loss_min: {:.10f}'.format(valid_loss_min))

            # print('saving model')

            #save model
            torch.save(model.state_dict(), save_path)
            print('saved: ', save_path)




    return model
```

(IMPLEMENTATION) Experiment with the Weight Initialization

Use the code cell below to define a custom weight initialization, and then train with your weight initialization for a few epochs. Make sure that neither the training loss nor validation loss is `nan`.

Later on, you will be able to see how this compares to training with PyTorch's default weight initialization.

In [ ]:

```
def custom_weight_init(m):
    ## TODO: implement a weight initialization strategy

    #get the name of the layer
    name = m.__class__.__name__
    print(name)
    if name.find('Linear') != -1:

        # get the number of the inputs
        y = (1.0/np.sqrt(m.in_features))

        m.weight.data.normal_(0, y)
        m.bias.data.fill_(0)

print(model_scratch)

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

model_scratch.apply(custom_weight_init)
model_scratch = train(20, loaders_scratch, model_scratch, get_optimizer_scratch(model_s
    criterion_scratch, use_cuda, 'ignore.pt')
# model_scratch = train(1, loaders_scratch, model_scratch, get_optimizer_scratch(model_
#     criterion_scratch, use_cuda, 'ignore.pt')

training_done()
```

```
Net(
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=16384, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=128, bias=True)
  (fc3): Linear(in_features=128, out_features=50, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
MaxPool2d
Conv2d
Conv2d
Conv2d
Linear
Linear
Linear
Dropout
Net
Epoch: 1        Training Loss: 3.547239         Validation Loss: 3.537234
valid_loss_min: 3.5372340977
saved:  ignore.pt
Epoch: 2        Training Loss: 2.995708         Validation Loss: 3.318236
valid_loss_min: 3.3182357848
saved:  ignore.pt
Epoch: 3        Training Loss: 2.753861         Validation Loss: 3.383212
Epoch: 4        Training Loss: 2.623263         Validation Loss: 3.209296
```

```
valid_loss_min: 3.2092956901
saved:  ignore.pt
Epoch: 5        Training Loss: 2.536053         Validation Loss: 3.364543
Epoch: 6        Training Loss: 2.435655         Validation Loss: 3.158963
valid_loss_min: 3.1589625180
saved:  ignore.pt
Epoch: 7        Training Loss: 2.359009         Validation Loss: 3.177190
Epoch: 8        Training Loss: 2.325564         Validation Loss: 3.142543
valid_loss_min: 3.1425430179
saved:  ignore.pt
Epoch: 9        Training Loss: 2.230478         Validation Loss: 3.268974
Epoch: 10       Training Loss: 2.202910         Validation Loss: 3.285772
Epoch: 11       Training Loss: 2.175727         Validation Loss: 3.255863
Epoch: 12       Training Loss: 2.184670         Validation Loss: 3.155661
Epoch: 13       Training Loss: 2.128167         Validation Loss: 3.124331
valid_loss_min: 3.1243307889
saved:  ignore.pt
Epoch: 14       Training Loss: 2.076077         Validation Loss: 3.252947
Epoch: 15       Training Loss: 2.073512         Validation Loss: 3.264821
Epoch: 16       Training Loss: 2.025115         Validation Loss: 3.266159
Epoch: 17       Training Loss: 2.006955         Validation Loss: 3.238283
Epoch: 18       Training Loss: 1.945982         Validation Loss: 3.219480
Epoch: 19       Training Loss: 1.948232         Validation Loss: 3.205019
Epoch: 20       Training Loss: 1.976405         Validation Loss: 3.189481
```

(IMPLEMENTATION) Train and Validate the Model

Run the next code cell to train your model.

In [ ]:

```
#this allowed me to save information about my model

def save_model_specs(m):
    dttm = datetime.now().strftime("%Y%m%d%H%M%S")
    name = 'model_spects_' + dttm + '.txt'
    with open(name, 'w') as f:
        f.write(model_scratch.print_model())
        f.write('\n\n')
        f.write(model_scratch.print_forward())
    return name


# save_model_specs(model_scratch)
```

In [ ]:

```
## TODO: you may change the number of epochs if you'd like,
## but changing it is not required
num_epochs = 200

#save information about this model
ms = save_model_specs(model_scratch)

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

# function to re-initialize a model with pytorch's default weight initialization
def default_weight_init(m):
    reset_parameters = getattr(m, 'reset_parameters', None)
    if callable(reset_parameters):
        m.reset_parameters()
```

```
# reset the model parameters
model_scratch.apply(default_weight_init)

# train the model
model_scratch = train(num_epochs, loaders_scratch, model_scratch, get_optimizer_scratch
                      criterion_scratch, use_cuda, 'model_scratch.pt')

training_done()
```

```
C:\Users\JGarza\miniconda3\envs\lm3\lib\site-packages\torch\nn\functional.py:718: UserWa
rning: Named tensors and all their associated APIs are an experimental feature and subje
ct to change. Please do not use them for anything important until they are released as s
table. (Triggered internally at  ..\c10/core/TensorImpl.h:1156.)
  return torch.max_pool2d(input, kernel_size, stride, padding, dilation, ceil_mode)
x.shape:  torch.Size([64, 16384])
Epoch: 1        Training Loss: 3.873153         Validation Loss: 3.923265
valid_loss_min: 3.9232650101
saved:  model_scratch.pt
Epoch: 2        Training Loss: 3.742517         Validation Loss: 3.779026
valid_loss_min: 3.7790262401
saved:  model_scratch.pt
Epoch: 3        Training Loss: 3.610291         Validation Loss: 3.705194
valid_loss_min: 3.7051938176
saved:  model_scratch.pt
Epoch: 4        Training Loss: 3.483709         Validation Loss: 3.653688
valid_loss_min: 3.6536877751
saved:  model_scratch.pt
Epoch: 5        Training Loss: 3.383897         Validation Loss: 3.612159
valid_loss_min: 3.6121587157
saved:  model_scratch.pt
Epoch: 6        Training Loss: 3.297176         Validation Loss: 3.596963
valid_loss_min: 3.5969630182
saved:  model_scratch.pt
Epoch: 7        Training Loss: 3.195979         Validation Loss: 3.573895
valid_loss_min: 3.5738947093
saved:  model_scratch.pt
Epoch: 8        Training Loss: 3.142290         Validation Loss: 3.531185
valid_loss_min: 3.5311848819
saved:  model_scratch.pt
Epoch: 9        Training Loss: 3.080629         Validation Loss: 3.584989
Epoch: 10       Training Loss: 3.001242         Validation Loss: 3.446130
valid_loss_min: 3.4461302757
saved:  model_scratch.pt
Epoch: 11       Training Loss: 2.946821         Validation Loss: 3.443898
valid_loss_min: 3.4438978732
saved:  model_scratch.pt
Epoch: 12       Training Loss: 2.871822         Validation Loss: 3.474834
Epoch: 13       Training Loss: 2.848054         Validation Loss: 3.517810
Epoch: 14       Training Loss: 2.815887         Validation Loss: 3.506477
Epoch: 15       Training Loss: 2.761858         Validation Loss: 3.399811
valid_loss_min: 3.3998109698
saved:  model_scratch.pt
Epoch: 16       Training Loss: 2.735817         Validation Loss: 3.385872
valid_loss_min: 3.3858715892
saved:  model_scratch.pt
Epoch: 17       Training Loss: 2.676215         Validation Loss: 3.453596
Epoch: 18       Training Loss: 2.618847         Validation Loss: 3.448229
Epoch: 19       Training Loss: 2.593927         Validation Loss: 3.383564
valid_loss_min: 3.3835643530
saved:  model_scratch.pt
```

```
Epoch: 20       Training Loss: 2.579453         Validation Loss: 3.354790
valid_loss_min: 3.3547903299
saved:  model_scratch.pt
Epoch: 21       Training Loss: 2.563262         Validation Loss: 3.322260
valid_loss_min: 3.3222599030
saved:  model_scratch.pt
Epoch: 22       Training Loss: 2.531193         Validation Loss: 3.354297
Epoch: 23       Training Loss: 2.506894         Validation Loss: 3.433205
Epoch: 24       Training Loss: 2.436046         Validation Loss: 3.337532
Epoch: 25       Training Loss: 2.461715         Validation Loss: 3.268325
valid_loss_min: 3.2683249116
saved:  model_scratch.pt
Epoch: 26       Training Loss: 2.396691         Validation Loss: 3.381772
Epoch: 27       Training Loss: 2.396133         Validation Loss: 3.382679
Epoch: 28       Training Loss: 2.388649         Validation Loss: 3.332398
Epoch: 29       Training Loss: 2.340199         Validation Loss: 3.232119
valid_loss_min: 3.2321191430
saved:  model_scratch.pt
Epoch: 30       Training Loss: 2.304878         Validation Loss: 3.313002
Epoch: 31       Training Loss: 2.296762         Validation Loss: 3.344792
Epoch: 32       Training Loss: 2.300024         Validation Loss: 3.358517
Epoch: 33       Training Loss: 2.259769         Validation Loss: 3.370558
Epoch: 34       Training Loss: 2.245814         Validation Loss: 3.281126
Epoch: 35       Training Loss: 2.210682         Validation Loss: 3.246249
Epoch: 36       Training Loss: 2.175223         Validation Loss: 3.217550
valid_loss_min: 3.2175504863
saved:  model_scratch.pt
Epoch: 37       Training Loss: 2.171326         Validation Loss: 3.314161
Epoch: 38       Training Loss: 2.172530         Validation Loss: 3.329392
Epoch: 39       Training Loss: 2.173602         Validation Loss: 3.257709
Epoch: 40       Training Loss: 2.128484         Validation Loss: 3.305284
Epoch: 41       Training Loss: 2.155317         Validation Loss: 3.215880
valid_loss_min: 3.2158801556
saved:  model_scratch.pt
Epoch: 42       Training Loss: 2.100343         Validation Loss: 3.345874
Epoch: 43       Training Loss: 2.029278         Validation Loss: 3.320589
Epoch: 44       Training Loss: 2.075221         Validation Loss: 3.314872
Epoch: 45       Training Loss: 2.063778         Validation Loss: 3.245261
Epoch: 46       Training Loss: 2.041589         Validation Loss: 3.247769
Epoch: 47       Training Loss: 2.045937         Validation Loss: 3.357670
Epoch: 48       Training Loss: 2.036675         Validation Loss: 3.336010
Epoch: 49       Training Loss: 1.999043         Validation Loss: 3.264808
Epoch: 50       Training Loss: 1.984028         Validation Loss: 3.300476
Epoch: 51       Training Loss: 1.943146         Validation Loss: 3.340531
Epoch: 52       Training Loss: 1.976503         Validation Loss: 3.290643
Epoch: 53       Training Loss: 1.994231         Validation Loss: 3.235343
Epoch: 54       Training Loss: 1.939168         Validation Loss: 3.267083
Epoch: 55       Training Loss: 1.916948         Validation Loss: 3.239802
Epoch: 56       Training Loss: 1.989915         Validation Loss: 3.222888
Epoch: 57       Training Loss: 1.932901         Validation Loss: 3.275119
Epoch: 58       Training Loss: 1.916238         Validation Loss: 3.323514
Epoch: 59       Training Loss: 1.940147         Validation Loss: 3.313124
Epoch: 60       Training Loss: 1.907206         Validation Loss: 3.290156
Epoch: 61       Training Loss: 1.942205         Validation Loss: 3.293176
Epoch: 62       Training Loss: 1.903101         Validation Loss: 3.185369
valid_loss_min: 3.1853693724
saved:  model_scratch.pt
Epoch: 63       Training Loss: 1.884316         Validation Loss: 3.267317
Epoch: 64       Training Loss: 1.833592         Validation Loss: 3.346903
Epoch: 65       Training Loss: 1.840247         Validation Loss: 3.452234
Epoch: 66       Training Loss: 1.847975         Validation Loss: 3.248863
```

```
Epoch: 67       Training Loss: 1.835166         Validation Loss: 3.438620
Epoch: 68       Training Loss: 1.821640         Validation Loss: 3.320481
Epoch: 69       Training Loss: 1.837836         Validation Loss: 3.206787
Epoch: 70       Training Loss: 1.803670         Validation Loss: 3.399221
Epoch: 71       Training Loss: 1.831343         Validation Loss: 3.320581
Epoch: 72       Training Loss: 1.855379         Validation Loss: 3.279020
Epoch: 73       Training Loss: 1.785960         Validation Loss: 3.265032
Epoch: 74       Training Loss: 1.767898         Validation Loss: 3.314181
Epoch: 75       Training Loss: 1.786411         Validation Loss: 3.370351
Epoch: 76       Training Loss: 1.802403         Validation Loss: 3.350020
Epoch: 77       Training Loss: 1.725970         Validation Loss: 3.279673
Epoch: 78       Training Loss: 1.757657         Validation Loss: 3.254071
Epoch: 79       Training Loss: 1.721502         Validation Loss: 3.215011
Epoch: 80       Training Loss: 1.713209         Validation Loss: 3.346990
Epoch: 81       Training Loss: 1.701474         Validation Loss: 3.351555
Epoch: 82       Training Loss: 1.697753         Validation Loss: 3.249437
Epoch: 83       Training Loss: 1.764958         Validation Loss: 3.219951
Epoch: 84       Training Loss: 1.717125         Validation Loss: 3.282340
Epoch: 85       Training Loss: 1.679295         Validation Loss: 3.331624
Epoch: 86       Training Loss: 1.686862         Validation Loss: 3.340589
Epoch: 87       Training Loss: 1.658542         Validation Loss: 3.330867
Epoch: 88       Training Loss: 1.716238         Validation Loss: 3.301521
Epoch: 89       Training Loss: 1.655747         Validation Loss: 3.258401
Epoch: 90       Training Loss: 1.711830         Validation Loss: 3.256185
Epoch: 91       Training Loss: 1.698082         Validation Loss: 3.204439
Epoch: 92       Training Loss: 1.670694         Validation Loss: 3.312573
Epoch: 93       Training Loss: 1.686687         Validation Loss: 3.341612
Epoch: 94       Training Loss: 1.650589         Validation Loss: 3.136930
valid_loss_min: 3.1369304061
saved:  model_scratch.pt
Epoch: 95       Training Loss: 1.644793         Validation Loss: 3.304775
Epoch: 96       Training Loss: 1.668745         Validation Loss: 3.396186
Epoch: 97       Training Loss: 1.619471         Validation Loss: 3.188164
Epoch: 98       Training Loss: 1.626114         Validation Loss: 3.251437
Epoch: 99       Training Loss: 1.594419         Validation Loss: 3.270999
Epoch: 100      Training Loss: 1.604230         Validation Loss: 3.199820
Epoch: 101      Training Loss: 1.625020         Validation Loss: 3.243446
Epoch: 102      Training Loss: 1.637467         Validation Loss: 3.493325
Epoch: 103      Training Loss: 1.624048         Validation Loss: 3.343145
Epoch: 104      Training Loss: 1.580080         Validation Loss: 3.235943
Epoch: 105      Training Loss: 1.626861         Validation Loss: 3.318541
Epoch: 106      Training Loss: 1.608351         Validation Loss: 3.351922
Epoch: 107      Training Loss: 1.627345         Validation Loss: 3.319736
Epoch: 108      Training Loss: 1.618487         Validation Loss: 3.294760
Epoch: 109      Training Loss: 1.581394         Validation Loss: 3.363090
Epoch: 110      Training Loss: 1.582254         Validation Loss: 3.323109
Epoch: 111      Training Loss: 1.553928         Validation Loss: 3.387664
Epoch: 112      Training Loss: 1.578295         Validation Loss: 3.402485
Epoch: 113      Training Loss: 1.562038         Validation Loss: 3.319520
Epoch: 114      Training Loss: 1.602672         Validation Loss: 3.337283
Epoch: 115      Training Loss: 1.566514         Validation Loss: 3.341435
Epoch: 116      Training Loss: 1.574522         Validation Loss: 3.320448
Epoch: 117      Training Loss: 1.544316         Validation Loss: 3.348370
Epoch: 118      Training Loss: 1.553516         Validation Loss: 3.383344
Epoch: 119      Training Loss: 1.550338         Validation Loss: 3.277851
Epoch: 120      Training Loss: 1.536621         Validation Loss: 3.298728
Epoch: 121      Training Loss: 1.517157         Validation Loss: 3.275847
Epoch: 122      Training Loss: 1.523779         Validation Loss: 3.397568
Epoch: 123      Training Loss: 1.510676         Validation Loss: 3.460572
Epoch: 124      Training Loss: 1.530161         Validation Loss: 3.241450
Epoch: 125      Training Loss: 1.502001         Validation Loss: 3.590507
```

```
Epoch: 126      Training Loss: 1.507606         Validation Loss: 3.355292
Epoch: 127      Training Loss: 1.504227         Validation Loss: 3.275490
Epoch: 128      Training Loss: 1.467714         Validation Loss: 3.334708
Epoch: 129      Training Loss: 1.481764         Validation Loss: 3.398428
Epoch: 130      Training Loss: 1.526085         Validation Loss: 3.467800
Epoch: 131      Training Loss: 1.494790         Validation Loss: 3.347629
Epoch: 132      Training Loss: 1.505155         Validation Loss: 3.201993
Epoch: 133      Training Loss: 1.531405         Validation Loss: 3.359160
Epoch: 134      Training Loss: 1.470475         Validation Loss: 3.418785
Epoch: 135      Training Loss: 1.460317         Validation Loss: 3.313516
Epoch: 136      Training Loss: 1.482611         Validation Loss: 3.276762
Epoch: 137      Training Loss: 1.484042         Validation Loss: 3.368085
Epoch: 138      Training Loss: 1.483218         Validation Loss: 3.506397
Epoch: 139      Training Loss: 1.468984         Validation Loss: 3.635416
Epoch: 140      Training Loss: 1.503921         Validation Loss: 3.301408
Epoch: 141      Training Loss: 1.453242         Validation Loss: 3.343073
Epoch: 142      Training Loss: 1.454843         Validation Loss: 3.313228
Epoch: 143      Training Loss: 1.470952         Validation Loss: 3.533153
Epoch: 144      Training Loss: 1.467885         Validation Loss: 3.395523
Epoch: 145      Training Loss: 1.443691         Validation Loss: 3.250160
Epoch: 146      Training Loss: 1.470001         Validation Loss: 3.413421
Epoch: 147      Training Loss: 1.379275         Validation Loss: 3.517614
Epoch: 148      Training Loss: 1.422496         Validation Loss: 3.417979
Epoch: 149      Training Loss: 1.434433         Validation Loss: 3.286021
Epoch: 150      Training Loss: 1.457590         Validation Loss: 3.449401
Epoch: 151      Training Loss: 1.465470         Validation Loss: 3.335582
Epoch: 152      Training Loss: 1.414177         Validation Loss: 3.283046
Epoch: 153      Training Loss: 1.406939         Validation Loss: 3.409252
Epoch: 154      Training Loss: 1.402871         Validation Loss: 3.460306
Epoch: 155      Training Loss: 1.437788         Validation Loss: 3.324561
Epoch: 156      Training Loss: 1.391447         Validation Loss: 3.299711
Epoch: 157      Training Loss: 1.465664         Validation Loss: 3.403409
Epoch: 158      Training Loss: 1.507976         Validation Loss: 3.264173
Epoch: 159      Training Loss: 1.384741         Validation Loss: 3.289305
Epoch: 160      Training Loss: 1.447772         Validation Loss: 3.285047
Epoch: 161      Training Loss: 1.358992         Validation Loss: 3.377420
Epoch: 162      Training Loss: 1.430509         Validation Loss: 3.429964
Epoch: 163      Training Loss: 1.375167         Validation Loss: 3.294994
Epoch: 164      Training Loss: 1.434617         Validation Loss: 3.304188
Epoch: 165      Training Loss: 1.427945         Validation Loss: 3.301825
Epoch: 166      Training Loss: 1.427787         Validation Loss: 3.387916
Epoch: 167      Training Loss: 1.378402         Validation Loss: 3.339757
Epoch: 168      Training Loss: 1.414967         Validation Loss: 3.387617
Epoch: 169      Training Loss: 1.397306         Validation Loss: 3.156665
Epoch: 170      Training Loss: 1.379892         Validation Loss: 3.386069
Epoch: 171      Training Loss: 1.421549         Validation Loss: 3.396428
Epoch: 172      Training Loss: 1.399397         Validation Loss: 3.203363
Epoch: 173      Training Loss: 1.409170         Validation Loss: 3.362925
Epoch: 174      Training Loss: 1.369631         Validation Loss: 3.303527
Epoch: 175      Training Loss: 1.353078         Validation Loss: 3.311549
Epoch: 176      Training Loss: 1.344728         Validation Loss: 3.378529
Epoch: 177      Training Loss: 1.388703         Validation Loss: 3.326973
Epoch: 178      Training Loss: 1.385569         Validation Loss: 3.391251
Epoch: 179      Training Loss: 1.378590         Validation Loss: 3.386482
Epoch: 180      Training Loss: 1.358225         Validation Loss: 3.192385
Epoch: 181      Training Loss: 1.374148         Validation Loss: 3.430165
Epoch: 182      Training Loss: 1.345461         Validation Loss: 3.312873
Epoch: 183      Training Loss: 1.392624         Validation Loss: 3.359835
Epoch: 184      Training Loss: 1.377776         Validation Loss: 3.293821
Epoch: 185      Training Loss: 1.403242         Validation Loss: 3.373683
Epoch: 186      Training Loss: 1.362348         Validation Loss: 3.293807
```

```
Epoch: 187      Training Loss: 1.370734         Validation Loss: 3.209247
Epoch: 188      Training Loss: 1.363375         Validation Loss: 3.318186
Epoch: 189      Training Loss: 1.332865         Validation Loss: 3.364515
Epoch: 190      Training Loss: 1.359444         Validation Loss: 3.462594
Epoch: 191      Training Loss: 1.326493         Validation Loss: 3.332458
Epoch: 192      Training Loss: 1.374573         Validation Loss: 3.553651
Epoch: 193      Training Loss: 1.374436         Validation Loss: 3.408561
Epoch: 194      Training Loss: 1.385739         Validation Loss: 3.306505
Epoch: 195      Training Loss: 1.375677         Validation Loss: 3.529334
Epoch: 196      Training Loss: 1.346959         Validation Loss: 3.308462
Epoch: 197      Training Loss: 1.323029         Validation Loss: 3.339844
Epoch: 198      Training Loss: 1.332034         Validation Loss: 3.374526
Epoch: 199      Training Loss: 1.318532         Validation Loss: 3.416389
Epoch: 200      Training Loss: 1.382229         Validation Loss: 3.215977
```

(IMPLEMENTATION) Test the Model

Run the code cell below to try out your model on the test dataset of landmark images. Run the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 20%.

In [ ]:

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

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

    # set the module to evaluation mode
    model.eval()

    for batch_idx, (data, target) in enumerate(loaders['test']):
        # move to GPU
        if use_cuda:
            torch.cuda.empty_cache()
            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.item() - 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))

    #if the model specs file is still there i can add the test scores to it
    try:
        with open(ms,'a') as f:
            f.write('\n\n')
            f.write('Test Loss: {:.6f}\n'.format(test_loss))
            f.write('\nTest Accuracy: %2d%% (%2d/%2d)' % (
```

```
            100. * correct / total, correct, total))
    except:
        print('no ms')
        pass

# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.197581


Test Accuracy: 28% (144/501)
no ms
```

Step 2: Create a CNN to Classify Landmarks (using Transfer Learning)

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

(IMPLEMENTATION) Specify Data Loaders for the Landmark Dataset

Use the code cell below to create three separate data loaders: one for training data, one for validation data, and one for test data. Randomly split the images located at `landmark_images/train` to create the train and validation data loaders, and use the images located at `landmark_images/test` to create the test data loader.

All three of your data loaders should be accessible via a dictionary named `loaders_transfer`. Your train data loader should be at `loaders_transfer['train']`, your validation data loader should be at `loaders_transfer['valid']`, and your test data loader should be at `loaders_transfer['test']`.

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

In [ ]:

```
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
import os
import numpy as np
import torch
import torch.nn as nn

import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt

%matplotlib inline
```

```
# loaders_transfer = {'train': None, 'valid': None, 'test': None}
# loaders_transfer = loaders_scratch
```

In [ ]:

```
data_train = 'landmark_images/train' #this is where the raw data is

dir_train = 'images/train'
dir_test = 'images/test'
dir_val= 'images/val'

if os.path.exists(dir_train) == False:
    splitfolders.ratio(data_train, output="images", seed=69, ratio=(.8, .1, .1), group_
```

In [ ]:

```
size = 256
mean = np.array([0.485, 0.456, 0.406])
std = np.array([0.229, 0.224, 0.225])


transform0 = transforms.Compose([
    transforms.RandomRotation(30),
    transforms.RandomResizedCrop(size),
    transforms.RandomHorizontalFlip(p=0.1),
    transforms.ColorJitter(brightness=0.05,contrast=0.05,saturation=0.5,hue=0.05),
    # transforms.ColorJitter(contrast=0.05),
    # transforms.ColorJitter(saturation=0.05),
    # transforms.ColorJitter(hue=0.05),
    transforms.ToTensor(),
    transforms.Normalize(mean,std)
])

transform1 = transforms.Compose([
    transforms.CenterCrop((size, size)),
    transforms.ToTensor(),
    transforms.Normalize(mean,std)
])

loaders_transfer = {}
loaders = [
        {
            'name':'train',
            'dir': dir_train,
            'transform': transform0,
        },
        {
            'name':'valid',
            'dir': dir_val,
            'transform': transform1,
        },
        {
            'name':'test',
            'dir': dir_test,
            'transform': transform1,
        },
    ]


for l in loaders:
    dataset = datasets.ImageFolder(root=l['dir'], transform=l['transform'])
    print(l['name'],len(dataset))
```

```
    loaders_transfer[l['name']] = torch.utils.data.DataLoader(dataset, batch_size=16, s


print(*loaders_transfer)
```

```
train 3996
valid 499
test 501
train valid test
```

(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 fill in the function `get_optimizer_transfer` below.

In [ ]:

```
## TODO: select loss function
# criterion_scratch = None

# we used this one in the lesson
criterion_transfer = nn.CrossEntropyLoss()

#others to try
# criterion_transfer = nn.L1Loss()
# criterion_transfer = nn.NLLLoss()
# criterion_transfer = nn.MSELoss()
# criterion_transfer = nn.HingeEmbeddingLoss()


# def get_optimizer_scratch(model, learn_rate=0.001):
def get_optimizer_transfer(model):
    ## TODO: select and return an optimizer
    optimizer = torch.optim.SGD(model.parameters(), lr = 0.001)

    # others to try
    # optimizer = torch.optim.RMSprop(model.parameters(),lr=0.001)
    # optimizer = torch.optim.Adadelta(model.parameters(),lr=0.001)
    # optimizer = torch.optim.Adam(model.parameters(),lr=0.001)
    # optimizer = torch.optim.Adagrad(model.parameters(),lr=0.001)

    return optimizer
```

(IMPLEMENTATION) Model Architecture

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

In [ ]:

```
# useful variable that tells us whether we should use the GPU
use_cuda = torch.cuda.is_available()
print(use_cuda)
```

```
True
```

In [ ]:

```
torch.cuda.empty_cache()
```

In [ ]:

```
## TODO: Specify model architecture


# https://pytorch.org/vision/stable/models.html
# model_transfer = models.vgg16(pretrained=True) #not working only about 52%

model_transfer = models.vgg19_bn(pretrained=True)

# model_transfer = models.vgg11(pretrained=True)

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

num_inputs = model_transfer.classifier[6].in_features
final_layer = nn.Linear(num_inputs, 50)
model_transfer.classifier[6] = final_layer


# model_transfer = models.resnet50(pretrained=True) #resnet is difficult to get working
# num_features = model_transfer.fc.in_features
# model_transfer.fc = nn.Linear(num_features, 50)

torch.save(model_transfer.state_dict(), 'model_transfer.pt')

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

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

```
Downloading: "https://download.pytorch.org/models/vgg19_bn-c79401a0.pth" to C:\Users\JGa
rza/.cache\torch\hub\checkpoints\vgg19_bn-c79401a0.pth
100.0%
```

Question 3: 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:
~~we used the vgg16 in one of the demos, so i tried that one first and it seems to work ok.

after importing vgg16 we had to set the number of inputs based on the number of features, and then we set the outputs to 50 for our classes.~~

I tried a few models, and some were a bit difficult to set up. but after i saw the training on the vgg models i was able to get that model working.

vgg19 architecture is able to scale down the image in multiple layers...this allows it to retain the important parts of the image, and remove the noise. in addition the 3 fully connected layers are able to make more sense out of a larger number of inputs then 1 fully connected layer alone.

(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 [ ]:

```
# TODO: train the model and save the best model parameters at filepath 'model_transfer.

model_transfer = train(100, loaders_transfer, model_transfer, get_optimizer_transfer(mo

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

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

```
Epoch: 1        Training Loss: 3.801741         Validation Loss: 3.542806
valid_loss_min: 3.5428062156
saved:  model_transfer.pt
Epoch: 2        Training Loss: 3.452338         Validation Loss: 3.166952
valid_loss_min: 3.1669523418
saved:  model_transfer.pt
Epoch: 3        Training Loss: 3.122536         Validation Loss: 2.797232
valid_loss_min: 2.7972324118
saved:  model_transfer.pt
Epoch: 4        Training Loss: 2.807328         Validation Loss: 2.502496
valid_loss_min: 2.5024958029
saved:  model_transfer.pt
Epoch: 5        Training Loss: 2.514951         Validation Loss: 2.214901
valid_loss_min: 2.2149012610
saved:  model_transfer.pt
Epoch: 6        Training Loss: 2.319168         Validation Loss: 2.071080
valid_loss_min: 2.0710796006
saved:  model_transfer.pt
Epoch: 7        Training Loss: 2.146826         Validation Loss: 1.901805
valid_loss_min: 1.9018054828
saved:  model_transfer.pt
Epoch: 8        Training Loss: 2.039973         Validation Loss: 1.811968
valid_loss_min: 1.8119676039
saved:  model_transfer.pt
Epoch: 9        Training Loss: 1.941100         Validation Loss: 1.759248
valid_loss_min: 1.7592480369
saved:  model_transfer.pt
Epoch: 10       Training Loss: 1.868742         Validation Loss: 1.674663
valid_loss_min: 1.6746630333
saved:  model_transfer.pt
Epoch: 11       Training Loss: 1.834154         Validation Loss: 1.628738
valid_loss_min: 1.6287379581
saved:  model_transfer.pt
Epoch: 12       Training Loss: 1.755940         Validation Loss: 1.598254
valid_loss_min: 1.5982540529
saved:  model_transfer.pt
Epoch: 13       Training Loss: 1.745586         Validation Loss: 1.604606
Epoch: 14       Training Loss: 1.708091         Validation Loss: 1.532228
valid_loss_min: 1.5322275423
saved:  model_transfer.pt
Epoch: 15       Training Loss: 1.661741         Validation Loss: 1.530211
valid_loss_min: 1.5302106105
saved:  model_transfer.pt
Epoch: 16       Training Loss: 1.671059         Validation Loss: 1.499456
valid_loss_min: 1.4994556680
saved:  model_transfer.pt
Epoch: 17       Training Loss: 1.602576         Validation Loss: 1.505682
Epoch: 18       Training Loss: 1.589733         Validation Loss: 1.481229
valid_loss_min: 1.4812291171
saved:  model_transfer.pt
```

```
Epoch: 19       Training Loss: 1.567645         Validation Loss: 1.486745
Epoch: 20       Training Loss: 1.526702         Validation Loss: 1.447163
valid_loss_min: 1.4471627055
saved:  model_transfer.pt
Epoch: 21       Training Loss: 1.530097         Validation Loss: 1.472251
Epoch: 22       Training Loss: 1.550866         Validation Loss: 1.424655
valid_loss_min: 1.4246554282
saved:  model_transfer.pt
Epoch: 23       Training Loss: 1.524984         Validation Loss: 1.441221
Epoch: 24       Training Loss: 1.513371         Validation Loss: 1.396163
valid_loss_min: 1.3961628648
saved:  model_transfer.pt
Epoch: 25       Training Loss: 1.467710         Validation Loss: 1.396367
Epoch: 26       Training Loss: 1.482894         Validation Loss: 1.398675
Epoch: 27       Training Loss: 1.450101         Validation Loss: 1.400391
Epoch: 28       Training Loss: 1.465930         Validation Loss: 1.418110
Epoch: 29       Training Loss: 1.450469         Validation Loss: 1.411157
Epoch: 30       Training Loss: 1.402869         Validation Loss: 1.443262
Epoch: 31       Training Loss: 1.399429         Validation Loss: 1.370469
valid_loss_min: 1.3704686575
saved:  model_transfer.pt
Epoch: 32       Training Loss: 1.371501         Validation Loss: 1.384919
Epoch: 33       Training Loss: 1.372575         Validation Loss: 1.364801
valid_loss_min: 1.3648011656
saved:  model_transfer.pt
Epoch: 34       Training Loss: 1.368450         Validation Loss: 1.390200
Epoch: 35       Training Loss: 1.349220         Validation Loss: 1.377157
Epoch: 36       Training Loss: 1.377531         Validation Loss: 1.415021
Epoch: 37       Training Loss: 1.360668         Validation Loss: 1.382934
Epoch: 38       Training Loss: 1.344022         Validation Loss: 1.354084
valid_loss_min: 1.3540841844
saved:  model_transfer.pt
Epoch: 39       Training Loss: 1.354069         Validation Loss: 1.420224
Epoch: 40       Training Loss: 1.344322         Validation Loss: 1.352469
valid_loss_min: 1.3524686769
saved:  model_transfer.pt
Epoch: 41       Training Loss: 1.309006         Validation Loss: 1.371570
Epoch: 42       Training Loss: 1.310723         Validation Loss: 1.356746
Epoch: 43       Training Loss: 1.306272         Validation Loss: 1.415458
Epoch: 44       Training Loss: 1.308766         Validation Loss: 1.363453
Epoch: 45       Training Loss: 1.284993         Validation Loss: 1.358798
Epoch: 46       Training Loss: 1.316673         Validation Loss: 1.362160
Epoch: 47       Training Loss: 1.305400         Validation Loss: 1.363035
Epoch: 48       Training Loss: 1.274133         Validation Loss: 1.386637
Epoch: 49       Training Loss: 1.284394         Validation Loss: 1.387577
Epoch: 50       Training Loss: 1.261272         Validation Loss: 1.352565
Epoch: 51       Training Loss: 1.292438         Validation Loss: 1.370316
Epoch: 52       Training Loss: 1.233448         Validation Loss: 1.305114
valid_loss_min: 1.3051141967
saved:  model_transfer.pt
Epoch: 53       Training Loss: 1.247346         Validation Loss: 1.343543
Epoch: 54       Training Loss: 1.245966         Validation Loss: 1.326711
Epoch: 55       Training Loss: 1.258307         Validation Loss: 1.291160
valid_loss_min: 1.2911600727
saved:  model_transfer.pt
Epoch: 56       Training Loss: 1.242302         Validation Loss: 1.308448
Epoch: 57       Training Loss: 1.253893         Validation Loss: 1.333049
Epoch: 58       Training Loss: 1.237516         Validation Loss: 1.301190
Epoch: 59       Training Loss: 1.225134         Validation Loss: 1.316601
Epoch: 60       Training Loss: 1.220613         Validation Loss: 1.343164
Epoch: 61       Training Loss: 1.210511         Validation Loss: 1.337858
```

```
Epoch: 62       Training Loss: 1.200453         Validation Loss: 1.310924
Epoch: 63       Training Loss: 1.207261         Validation Loss: 1.316450
Epoch: 64       Training Loss: 1.212559         Validation Loss: 1.352561
Epoch: 65       Training Loss: 1.187202         Validation Loss: 1.315956
Epoch: 66       Training Loss: 1.178044         Validation Loss: 1.316500
Epoch: 67       Training Loss: 1.201407         Validation Loss: 1.337132
Epoch: 68       Training Loss: 1.155121         Validation Loss: 1.338544
Epoch: 69       Training Loss: 1.193997         Validation Loss: 1.365828
Epoch: 70       Training Loss: 1.186053         Validation Loss: 1.286068
valid_loss_min: 1.2860682206
saved:  model_transfer.pt
Epoch: 71       Training Loss: 1.167307         Validation Loss: 1.289281
Epoch: 72       Training Loss: 1.195489         Validation Loss: 1.272583
valid_loss_min: 1.2725828513
saved:  model_transfer.pt
Epoch: 73       Training Loss: 1.167043         Validation Loss: 1.318778
Epoch: 74       Training Loss: 1.139039         Validation Loss: 1.339745
Epoch: 75       Training Loss: 1.142560         Validation Loss: 1.337402
Epoch: 76       Training Loss: 1.179030         Validation Loss: 1.290750
Epoch: 77       Training Loss: 1.136127         Validation Loss: 1.275186
Epoch: 78       Training Loss: 1.170927         Validation Loss: 1.277169
Epoch: 79       Training Loss: 1.117672         Validation Loss: 1.299860
Epoch: 80       Training Loss: 1.129766         Validation Loss: 1.320111
Epoch: 81       Training Loss: 1.134553         Validation Loss: 1.290523
Epoch: 82       Training Loss: 1.109167         Validation Loss: 1.309192
Epoch: 83       Training Loss: 1.158824         Validation Loss: 1.353063
Epoch: 84       Training Loss: 1.127334         Validation Loss: 1.359118
Epoch: 85       Training Loss: 1.111767         Validation Loss: 1.297525
Epoch: 86       Training Loss: 1.103297         Validation Loss: 1.294292
Epoch: 87       Training Loss: 1.123373         Validation Loss: 1.292377
Epoch: 88       Training Loss: 1.115293         Validation Loss: 1.304395
Epoch: 89       Training Loss: 1.096108         Validation Loss: 1.291939
Epoch: 90       Training Loss: 1.095606         Validation Loss: 1.293328
Epoch: 91       Training Loss: 1.079823         Validation Loss: 1.302960
Epoch: 92       Training Loss: 1.079580         Validation Loss: 1.302771
Epoch: 93       Training Loss: 1.088439         Validation Loss: 1.322767
Epoch: 94       Training Loss: 1.097192         Validation Loss: 1.290362
Epoch: 95       Training Loss: 1.119136         Validation Loss: 1.268183
valid_loss_min: 1.2681831997
saved:  model_transfer.pt
Epoch: 96       Training Loss: 1.066409         Validation Loss: 1.262901
valid_loss_min: 1.2629005546
saved:  model_transfer.pt
Epoch: 97       Training Loss: 1.075925         Validation Loss: 1.260939
valid_loss_min: 1.2609385019
saved:  model_transfer.pt
Epoch: 98       Training Loss: 1.084537         Validation Loss: 1.270316
Epoch: 99       Training Loss: 1.072272         Validation Loss: 1.286101
Epoch: 100      Training Loss: 1.060633         Validation Loss: 1.265338
<All keys matched successfully>
```

Out[ ]:

(IMPLEMENTATION) Test the Model

Try out your model on the test dataset of landmark 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 [ ]:

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

```
Test Loss: 1.343670


Test Accuracy: 65% (328/501)
no ms
```

Step 3: Write Your Landmark Prediction Algorithm

Great job creating your CNN models! Now that you have put in all the hard work of creating accurate classifiers, let's define some functions to make it easy for others to use your classifiers.

(IMPLEMENTATION) Write Your Algorithm, Part 1

Implement the function `predict_landmarks`, which accepts a file path to an image and an integer k, and then predicts the **top k most likely landmarks**. You are **required** to use your transfer learned CNN from Step 2 to predict the landmarks.

An example of the expected behavior of `predict_landmarks`:

```
>>> predicted_landmarks = predict_landmarks('example_image.jpg', 3)
>>> print(predicted_landmarks)
['Golden Gate Bridge', 'Brooklyn Bridge', 'Sydney Harbour Bridge']
```

In [ ]:

```
import cv2
from PIL import Image

## the class names can be accessed at the `classes` attribute
## of your dataset object (e.g., `train_dataset.classes`)

def predict_landmarks(img_path, k):
    ## TODO: return the names of the top k landmarks predicted by the transfer learned

    image = Image.open(img_path)
    i = transform0(image)
    # imshow(i)
    i = i[None,:]
    if use_cuda:
        i = i.cuda()
    p = model_transfer(i).cpu()
    p = p.detach().numpy()[0]

    p_sorted = p.copy()
    p_sorted.sort()
    p_sorted = p_sorted[-k:]
    p_sorted = np.flip(p_sorted)

    # print(p)
    # print(p_sorted)

    targets = list(range(k))
    for index,i in enumerate(p):
        if i in p_sorted:
            pos = np.where(p_sorted == i)[0]
            # print(pos)
```

```
            targets[pos[0]] = {'position':pos[0],'index':index,'value':i,'label':label_
    # print(*targets,sep='\n')
    return targets




# test on a sample image
predict_landmarks('images/test/09.Golden_Gate_Bridge/06dcc95353f874d9.jpg', 5)
```

Out[ ]:
```
[{'position': 0,
  'index': 9,
  'value': 14.619126,
  'label': '09.Golden_Gate_Bridge'},
 {'position': 1,
  'index': 30,
  'value': 9.5372715,
  'label': '30.Brooklyn_Bridge'},
 {'position': 2, 'index': 38, 'value': 4.770165, 'label': '38.Forth_Bridge'},
 {'position': 3,
  'index': 28,
  'value': 4.251987,
  'label': '28.Sydney_Harbour_Bridge'},
 {'position': 4,
  'index': 35,
  'value': 3.1836703,
  'label': '35.Monumento_a_la_Revolucion'}]
```

(IMPLEMENTATION) Write Your Algorithm, Part 2

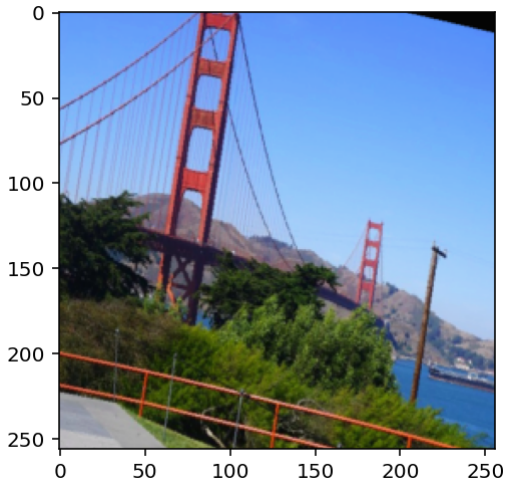
In the code cell below, implement the function `suggest_locations`, which accepts a file path to an image as input, and then displays the image and the **top 3 most likely landmarks** as predicted by `predict_landmarks`.

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

In [ ]:
```
def suggest_locations(img_path):
    # get landmark predictions

    print('img_path: ',img_path)

    predicted_landmarks_list = predict_landmarks(img_path, 3)
    # print(type(predicted_landmarks_list))

    ## TODO: display image and display landmark predictions
    image = Image.open(img_path)
    i = transform0(image)
    imshow(i)

    print(*predicted_landmarks_list,sep='\n')
```

```
# test on a sample image
suggest_locations('images/test/09.Golden_Gate_Bridge/06dcc95353f874d9.jpg')
```

```
img_path:  images/test/09.Golden_Gate_Bridge/06dcc95353f874d9.jpg
{'position': 0, 'index': 9, 'value': 16.038969, 'label': '09.Golden_Gate_Bridge'}
{'position': 1, 'index': 30, 'value': 9.797019, 'label': '30.Brooklyn_Bridge'}
{'position': 2, 'index': 38, 'value': 6.713119, 'label': '38.Forth_Bridge'}
```



(IMPLEMENTATION) Test Your Algorithm

Test your algorithm by running the `suggest_locations` function on at least four images on your computer. Feel free to use any images you like.

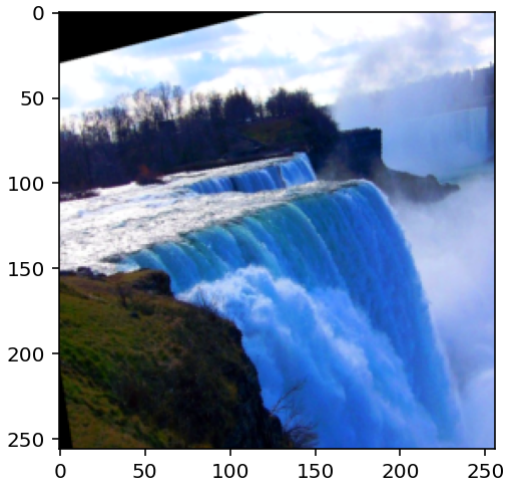
Question 4: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

- i might be shrinking down the images too much on the transforms should it be 64? 128? ...more?
- my random adjustments might be too much
- i might not have enough convd layers
- i might need to have multiple fully Connected layers

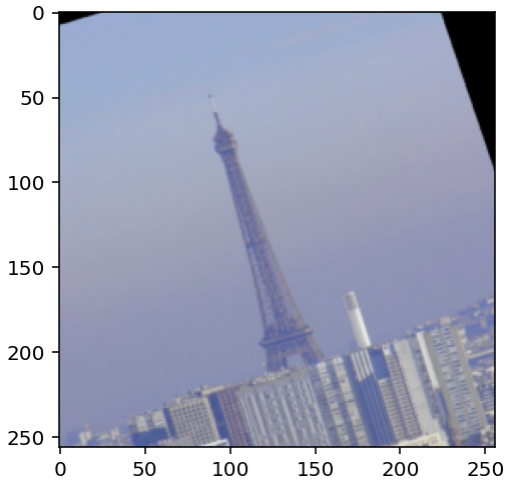
In [ ]:
```
## TODO: Execute the `suggest_locations` function on
## at least 4 images on your computer.
## Feel free to use as many code cells as needed.

suggest_locations('images/test/06.Niagara_Falls/35a6283e9c0a5d40.jpg')
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
[0..255] for integers).
img_path:  images/test/06.Niagara_Falls/35a6283e9c0a5d40.jpg
{'position': 0, 'index': 6, 'value': 10.27085, 'label': '06.Niagara_Falls'}
{'position': 1, 'index': 43, 'value': 9.590778, 'label': '43.Gullfoss_Falls'}
{'position': 2, 'index': 25, 'value': 5.131311, 'label': '25.Banff_National_Park'}
```



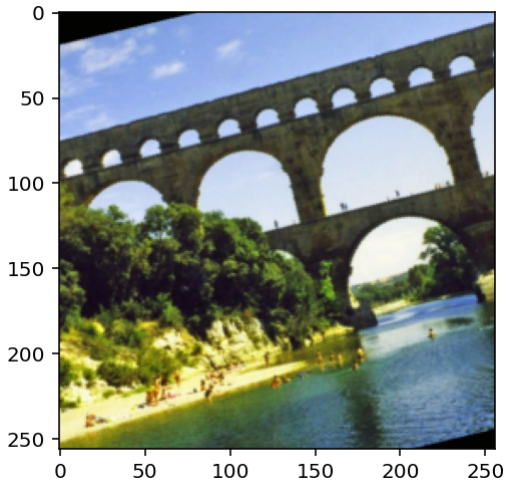
In [ ]:
```
suggest_locations('images/test/16.Eiffel_Tower/5cd6b1c34778255c.jpg')
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
[0..255] for integers).
img_path:  images/test/16.Eiffel_Tower/5cd6b1c34778255c.jpg
{'position': 0, 'index': 31, 'value': 7.397161, 'label': '31.Washington_Monument'}
{'position': 1, 'index': 16, 'value': 6.9582233, 'label': '16.Eiffel_Tower'}
{'position': 2, 'index': 29, 'value': 6.919037, 'label': '29.Petronas_Towers'}
```



In [ ]:
```
suggest_locations('images/test/26.Pont_du_Gard/1f9ee209316a09ba.jpg')
```

```
img_path:  images/test/26.Pont_du_Gard/1f9ee209316a09ba.jpg
{'position': 0, 'index': 26, 'value': 19.572773, 'label': '26.Pont_du_Gard'}
{'position': 1, 'index': 48, 'value': 9.511307, 'label': '48.Whitby_Abbey'}
{'position': 2, 'index': 38, 'value': 9.128225, 'label': '38.Forth_Bridge'}
```



In [ ]:
```
suggest_locations('images/test/36.Badlands_National_Park/0b82cd7b4b0a16d7.jpg')
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or
[0..255] for integers).
img_path:  images/test/36.Badlands_National_Park/0b82cd7b4b0a16d7.jpg
{'position': 0, 'index': 18, 'value': 12.210916, 'label': '18.Delicate_Arch'}
{'position': 1, 'index': 36, 'value': 11.308897, 'label': '36.Badlands_National_Park'}
{'position': 2, 'index': 42, 'value': 8.565559, 'label': '42.Death_Valley_National_Par
k'}
```