landmark

August 12, 2022

1 Convolutional Neural Networks

1.1 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.

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.

Just by looking at that image alone, would you have been able to guess that it was taken at the Haleakal 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!

1.1.1 (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.

Note: Remember that the dataset can be found at `/data/landmark_images/` in the workspace.

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 [1]:
```
# Importing necessary libraries
from PIL import Image
import os
import numpy as np
import torch
from torchvision import datasets
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
import torch.optim as optim
import torch.nn as nn
```

In [2]:
```
# Let's see the sizes of our images
# External Reference - https://stackoverflow.com/questions/6444548/how-do-i-get-the-pict
Image.open('/data/landmark_images/train/00.Haleakala_National_Park/084c2aa50d0a9249.jpg'
```

Out[2]:
```
(800, 600)
```

In [3]:
```
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes

# Defining data directories
data_directory = '/data/landmark_images/'
train_directory = os.path.join(data_directory, 'train')
test_directory = os.path.join(data_directory, 'test')

# Defining the batch and validation sizes
batch_size= 15
validation_size = 0.3

# External reference consulted - https://www.analyticsvidhya.com/blog/2021/04/10-pytorch
# Added GaussianBlur as well but had to remove it as I was getting error as torchvision.
# called 'GaussianBlur'

train_data_transform = transforms.Compose([transforms.RandomResizedCrop(256),
                                           transforms.RandomRotation(degrees=20),
                                           transforms.RandomHorizontalFlip(p=0.1),
                                           transforms.RandomVerticalFlip(p=0.1),
                                           transforms.RandomGrayscale(p=0.1),
                                           transforms.ToTensor(),
                                           transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.

validation_data_transform = transforms.Compose([transforms.Resize(256),
                                           transforms.CenterCrop(256),
                                           transforms.ToTensor(),
                                           transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.

train_data = datasets.ImageFolder(train_directory, transform=train_data_transform)
test_data = datasets.ImageFolder(test_directory, transform=validation_data_transform)


# We are using sabset random sampler.
# We would be distributing the data by shuffling and by maintaining the ratio of validat


# Preparing the loader with sampler
number_of_training_samples = len(train_data)
indices_of_entire_ds = list(range(number_of_training_samples))
np.random.shuffle(indices_of_entire_ds)
split_index = int(np.floor(validation_size * number_of_training_samples))
train_idx, valid_idx = indices_of_entire_ds[split_index:], indices_of_entire_ds[:split_i
train_random_sampler = SubsetRandomSampler(train_idx)
validation_random_sampler = SubsetRandomSampler(valid_idx)
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,sampler=tra
validation_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,sample
test_loader =  torch.utils.data.DataLoader(test_data, batch_size=batch_size, num_workers
```

```
loaders_scratch = {'train': train_loader, 'valid': validation_loader, 'test': test_loade
```

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?

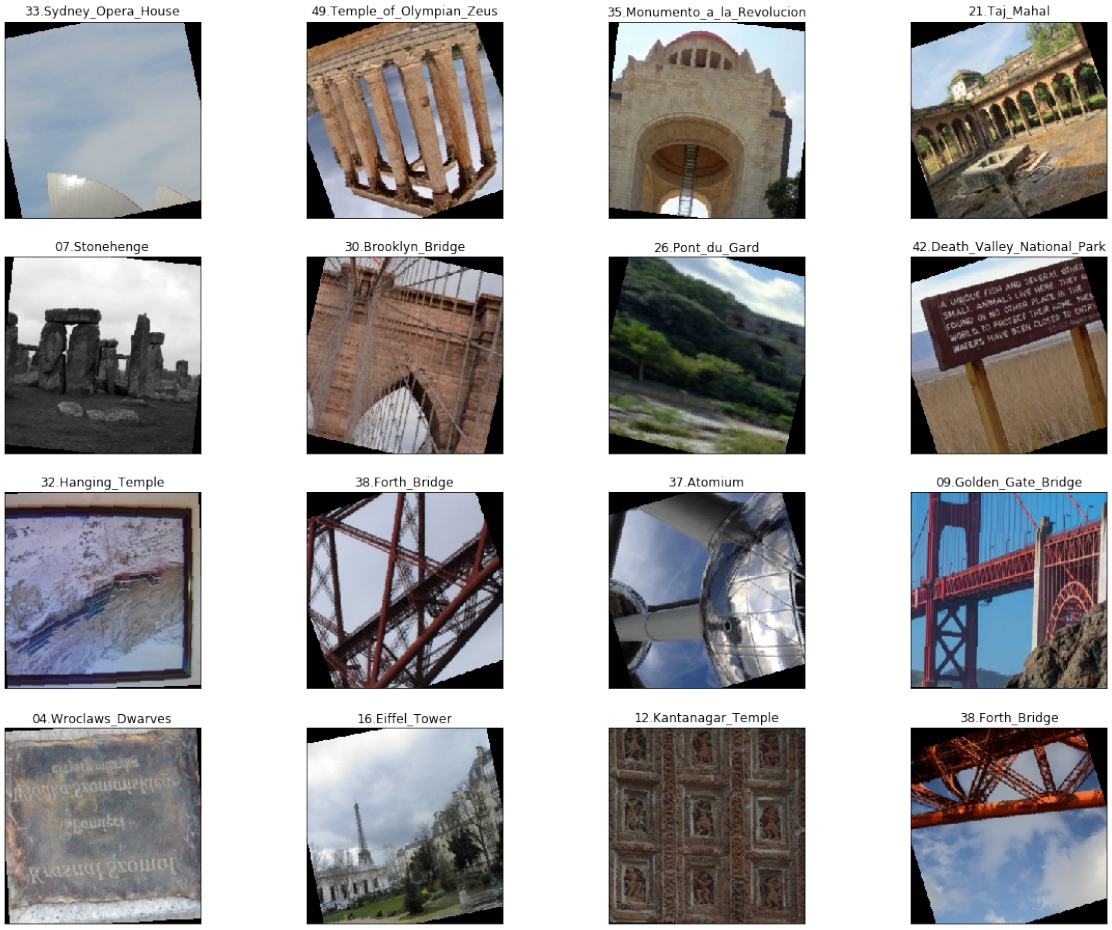
Answer: We are working on 256 * 256 images. On doing trial and error with various sizes, this gave better performance. As part of image augmentation, we have used, random resizing, center cropping, random rotation, random horizontal flip and normalising for training cases, and random resizing, center cropping and normalising for validation cases.

1.1.2 (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 [4]: import matplotlib.pyplot as plt
        %matplotlib inline
        import random

        ## 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`)

        # Internal source - Udacity helper code during training videos
        def imshow(img):
            img = img / 2 + 0.5
            plt.imshow(np.transpose(img.numpy(), (1, 2, 0)))
            return img

        fig = plt.figure(figsize=(20,16))
        classes = [classes_name.split(".")[1] for classes_name in train_data.classes]
        for index in range(16):
            ax = fig.add_subplot(4, 4, index+1, xticks=[], yticks=[])
            rand_img = random.randint(0, len(train_data))
            img = imshow(train_data[rand_img][0])
            ax.set_title(train_data.classes[train_data[rand_img][1]])
```



1.1.3 Initialize use_cuda variable

```
In [5]: # useful variable that tells us whether we should use the GPU
        use_cuda = torch.cuda.is_available()
```

1.1.4 (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 [6]: ## TODO: select loss function

        # Used the cross entropy loss since the case is multi class classification
        criterion_scratch = nn.CrossEntropyLoss()

        def get_optimizer_scratch(model):
            ## TODO: select and return an optimizer
            return optim.SGD(model.parameters(), lr=0.02)
```

1.1.5 (IMPLEMENTATION) Model Architecture

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

In [7]:
```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
class Net(nn.Module):
    ## TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()

        ## Define layers of a CNN
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64*32*32 , 1024)
        self.fc2 = nn.Linear(1024, 256)
        self.fc3 = nn.Linear(256, 128)
        self.fc4 = nn.Linear(128, 50)
        self.dropout = nn.Dropout(0.4)




    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 64*32*32 )
        x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = F.relu(self.fc3(x))
        x = self.dropout(x)
        x = self.fc4(x)
```

```
            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()
```

```
In [8]: print(model_scratch)
```

```
Net(
  (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))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=65536, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=256, bias=True)
  (fc3): Linear(in_features=256, out_features=128, bias=True)
  (fc4): Linear(in_features=128, out_features=50, bias=True)
  (dropout): Dropout(p=0.4)
)
```

Question 2: Outline the steps you took to get to your final CNN architecture and your reasoning at each step.

Answer: We have used three convolution layers followed by four fully connected layers. we tried with two and four convolutons and also tried with three linear (fully connected) layers, but in those cases, could not reach 20 % accuracy in test data. We used sgd as optimizer, actually we tried with adam earlier with comparable learning rate. In case of using adam, the error converging rate for this dataset was very low, that means it would take a lot longer to converge. SGD with learning rate of 0.02 gave a better convergence than with 0.01, obviously while operating on random snapshot of the total data (as we shuffled and randomized the indices for the images). I thought, later we can try with more complex networks and may be with a bigger number of epochs with a callback mechanism (where we could be specifying the minimum loss decrease on pre-defined number of epochs, in order for to proceed etc). Using dropouts we could see that accuracy improved by 10% more than which were without dropouts. We used standard max pool layer of 2 * 2 with strides as 2. This halved the hight and width of the input. We have applied a number image augmentation techniques of pytorch transforms, we can see applying more varied set of transformations and altering the probabilities to see the effect in future.

1.1.6 (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 [9]: 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:
            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() - tr
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data.item() - train



    ######################
    # 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:
            data, target = data.cuda(), target.cuda()

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

```
            # 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 st
            if valid_loss <= valid_loss_min:
                print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.for
                torch.save(model.state_dict(), save_path)
                valid_loss_min = valid_loss



        return model
```

1.1.7 (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 [43]: def custom_weight_init(m):
             ## TODO: implement a weight initialization strategy
             classname = m.__class__.__name__

             if classname.find('Linear') != -1:
                 num_inputs = m.in_features
                 y= 1.0/np.sqrt(num_inputs)
                 m.weight.data.uniform_(-y , y)
                 m.bias.data.fill_(0)



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

         model_scratch.apply(custom_weight_init)
         model_scratch = train(8, loaders_scratch, model_scratch, get_optimizer_scratch(model_sc
                               criterion_scratch, use_cuda, 'ignore.pt')
```

```
Epoch: 1         Training Loss: 3.912922         Validation Loss: 3.913076
Validation loss decreased (inf --> 3.913076).  Saving model ...
Epoch: 2         Training Loss: 3.912261         Validation Loss: 3.913062
Validation loss decreased (3.913076 --> 3.913062).  Saving model ...
```

```
Epoch: 3         Training Loss: 3.910082         Validation Loss: 3.912290
Validation loss decreased (3.913062 --> 3.912290).  Saving model ...
Epoch: 4         Training Loss: 3.906258         Validation Loss: 3.904962
Validation loss decreased (3.912290 --> 3.904962).  Saving model ...
Epoch: 5         Training Loss: 3.895269         Validation Loss: 3.876913
Validation loss decreased (3.904962 --> 3.876913).  Saving model ...
Epoch: 6         Training Loss: 3.867520         Validation Loss: 3.849769
Validation loss decreased (3.876913 --> 3.849769).  Saving model ...
Epoch: 7         Training Loss: 3.852977         Validation Loss: 3.831681
Validation loss decreased (3.849769 --> 3.831681).  Saving model ...
Epoch: 8         Training Loss: 3.838297         Validation Loss: 3.816555
Validation loss decreased (3.831681 --> 3.816555).  Saving model ...
```

1.1.8 (IMPLEMENTATION) Train and Validate the Model

Run the next code cell to train your model.

```
In [10]: ## TODO: you may change the number of epochs if you'd like,
         ## but changing it is not required
         num_epochs = 100

         #-#-# 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')
```

```
Epoch: 1         Training Loss: 3.913423         Validation Loss: 3.913874
Validation loss decreased (inf --> 3.913874).  Saving model ...
Epoch: 2         Training Loss: 3.911869         Validation Loss: 3.913885
Epoch: 3         Training Loss: 3.909173         Validation Loss: 3.912154
Validation loss decreased (3.913874 --> 3.912154).  Saving model ...
Epoch: 4         Training Loss: 3.899386         Validation Loss: 3.896779
Validation loss decreased (3.912154 --> 3.896779).  Saving model ...
Epoch: 5         Training Loss: 3.875019         Validation Loss: 3.865647
Validation loss decreased (3.896779 --> 3.865647).  Saving model ...
Epoch: 6         Training Loss: 3.842585         Validation Loss: 3.867139
Epoch: 7         Training Loss: 3.831042         Validation Loss: 3.890847
```

```
Epoch: 8         Training Loss: 3.832695         Validation Loss: 3.844286
Validation loss decreased (3.865647 --> 3.844286).  Saving model ...
Epoch: 9         Training Loss: 3.806385         Validation Loss: 3.846126
Epoch: 10         Training Loss: 3.807946         Validation Loss: 3.836652
Validation loss decreased (3.844286 --> 3.836652).  Saving model ...
Epoch: 11         Training Loss: 3.796006         Validation Loss: 3.861960
Epoch: 12         Training Loss: 3.789496         Validation Loss: 3.801566
Validation loss decreased (3.836652 --> 3.801566).  Saving model ...
Epoch: 13         Training Loss: 3.786601         Validation Loss: 3.803515
Epoch: 14         Training Loss: 3.783884         Validation Loss: 3.794180
Validation loss decreased (3.801566 --> 3.794180).  Saving model ...
Epoch: 15         Training Loss: 3.774569         Validation Loss: 3.784166
Validation loss decreased (3.794180 --> 3.784166).  Saving model ...
Epoch: 16         Training Loss: 3.777193         Validation Loss: 3.784369
Epoch: 17         Training Loss: 3.761792         Validation Loss: 3.777437
Validation loss decreased (3.784166 --> 3.777437).  Saving model ...
Epoch: 18         Training Loss: 3.763835         Validation Loss: 3.850805
Epoch: 19         Training Loss: 3.747069         Validation Loss: 3.927807
Epoch: 20         Training Loss: 3.745840         Validation Loss: 3.823922
Epoch: 21         Training Loss: 3.739717         Validation Loss: 3.803420
Epoch: 22         Training Loss: 3.733029         Validation Loss: 3.767324
Validation loss decreased (3.777437 --> 3.767324).  Saving model ...
Epoch: 23         Training Loss: 3.709407         Validation Loss: 3.737296
Validation loss decreased (3.767324 --> 3.737296).  Saving model ...
Epoch: 24         Training Loss: 3.713738         Validation Loss: 3.750854
Epoch: 25         Training Loss: 3.674470         Validation Loss: 3.715262
Validation loss decreased (3.737296 --> 3.715262).  Saving model ...
Epoch: 26         Training Loss: 3.694244         Validation Loss: 3.704538
Validation loss decreased (3.715262 --> 3.704538).  Saving model ...
Epoch: 27         Training Loss: 3.676139         Validation Loss: 3.714035
Epoch: 28         Training Loss: 3.662277         Validation Loss: 3.698381
Validation loss decreased (3.704538 --> 3.698381).  Saving model ...
Epoch: 29         Training Loss: 3.646394         Validation Loss: 3.782303
Epoch: 30         Training Loss: 3.622972         Validation Loss: 3.657739
Validation loss decreased (3.698381 --> 3.657739).  Saving model ...
Epoch: 31         Training Loss: 3.631088         Validation Loss: 3.769034
Epoch: 32         Training Loss: 3.605301         Validation Loss: 3.677366
Epoch: 33         Training Loss: 3.588285         Validation Loss: 3.645385
Validation loss decreased (3.657739 --> 3.645385).  Saving model ...
Epoch: 34         Training Loss: 3.598676         Validation Loss: 3.601982
Validation loss decreased (3.645385 --> 3.601982).  Saving model ...
Epoch: 35         Training Loss: 3.566523         Validation Loss: 4.678338
Epoch: 36         Training Loss: 3.559583         Validation Loss: 3.892051
Epoch: 37         Training Loss: 3.555884         Validation Loss: 3.695909
Epoch: 38         Training Loss: 3.541166         Validation Loss: 3.591787
Validation loss decreased (3.601982 --> 3.591787).  Saving model ...
Epoch: 39         Training Loss: 3.514702         Validation Loss: 3.573704
Validation loss decreased (3.591787 --> 3.573704).  Saving model ...
```

```
Epoch: 40         Training Loss: 3.512603         Validation Loss: 3.548648
Validation loss decreased (3.573704 --> 3.548648).  Saving model ...
Epoch: 41         Training Loss: 3.514374         Validation Loss: 3.660301
Epoch: 42         Training Loss: 3.501249         Validation Loss: 3.533611
Validation loss decreased (3.548648 --> 3.533611).  Saving model ...
Epoch: 43         Training Loss: 3.505938         Validation Loss: 3.516864
Validation loss decreased (3.533611 --> 3.516864).  Saving model ...
Epoch: 44         Training Loss: 3.477929         Validation Loss: 3.552981
Epoch: 45         Training Loss: 3.460357         Validation Loss: 3.696497
Epoch: 46         Training Loss: 3.466994         Validation Loss: 3.634275
Epoch: 47         Training Loss: 3.433603         Validation Loss: 3.483739
Validation loss decreased (3.516864 --> 3.483739).  Saving model ...
Epoch: 48         Training Loss: 3.428722         Validation Loss: 3.511147
Epoch: 49         Training Loss: 3.419873         Validation Loss: 3.477794
Validation loss decreased (3.483739 --> 3.477794).  Saving model ...
Epoch: 50         Training Loss: 3.408594         Validation Loss: 3.499345
Epoch: 51         Training Loss: 3.407236         Validation Loss: 3.475628
Validation loss decreased (3.477794 --> 3.475628).  Saving model ...
Epoch: 52         Training Loss: 3.364330         Validation Loss: 3.481857
Epoch: 53         Training Loss: 3.374996         Validation Loss: 3.369084
Validation loss decreased (3.475628 --> 3.369084).  Saving model ...
Epoch: 54         Training Loss: 3.347294         Validation Loss: 3.400375
Epoch: 55         Training Loss: 3.341270         Validation Loss: 3.364926
Validation loss decreased (3.369084 --> 3.364926).  Saving model ...
Epoch: 56         Training Loss: 3.323968         Validation Loss: 3.347680
Validation loss decreased (3.364926 --> 3.347680).  Saving model ...
Epoch: 57         Training Loss: 3.316387         Validation Loss: 3.323234
Validation loss decreased (3.347680 --> 3.323234).  Saving model ...
Epoch: 58         Training Loss: 3.293698         Validation Loss: 3.308047
Validation loss decreased (3.323234 --> 3.308047).  Saving model ...
Epoch: 59         Training Loss: 3.296340         Validation Loss: 3.387789
Epoch: 60         Training Loss: 3.268534         Validation Loss: 3.295505
Validation loss decreased (3.308047 --> 3.295505).  Saving model ...
Epoch: 61         Training Loss: 3.274253         Validation Loss: 3.424173
Epoch: 62         Training Loss: 3.260137         Validation Loss: 3.225644
Validation loss decreased (3.295505 --> 3.225644).  Saving model ...
Epoch: 63         Training Loss: 3.243905         Validation Loss: 3.218765
Validation loss decreased (3.225644 --> 3.218765).  Saving model ...
Epoch: 64         Training Loss: 3.241777         Validation Loss: 3.492232
Epoch: 65         Training Loss: 3.230778         Validation Loss: 3.494298
Epoch: 66         Training Loss: 3.222287         Validation Loss: 3.245509
Epoch: 67         Training Loss: 3.194039         Validation Loss: 3.215393
Validation loss decreased (3.218765 --> 3.215393).  Saving model ...
Epoch: 68         Training Loss: 3.199134         Validation Loss: 3.269355
Epoch: 69         Training Loss: 3.176635         Validation Loss: 3.563009
Epoch: 70         Training Loss: 3.183355         Validation Loss: 3.183092
Validation loss decreased (3.215393 --> 3.183092).  Saving model ...
Epoch: 71         Training Loss: 3.168424         Validation Loss: 3.251602
```

```
Epoch: 72         Training Loss: 3.145338         Validation Loss: 4.014598
Epoch: 73         Training Loss: 3.147232         Validation Loss: 3.162912
Validation loss decreased (3.183092 --> 3.162912).  Saving model ...
Epoch: 74         Training Loss: 3.125444         Validation Loss: 3.162050
Validation loss decreased (3.162912 --> 3.162050).  Saving model ...
Epoch: 75         Training Loss: 3.115264         Validation Loss: 3.164838
Epoch: 76         Training Loss: 3.123929         Validation Loss: 3.154771
Validation loss decreased (3.162050 --> 3.154771).  Saving model ...
Epoch: 77         Training Loss: 3.095851         Validation Loss: 3.189995
Epoch: 78         Training Loss: 3.107980         Validation Loss: 3.229875
Epoch: 79         Training Loss: 3.080715         Validation Loss: 3.108867
Validation loss decreased (3.154771 --> 3.108867).  Saving model ...
Epoch: 80         Training Loss: 3.057520         Validation Loss: 3.169122
Epoch: 81         Training Loss: 3.080780         Validation Loss: 3.102261
Validation loss decreased (3.108867 --> 3.102261).  Saving model ...
Epoch: 82         Training Loss: 3.043155         Validation Loss: 3.139609
Epoch: 83         Training Loss: 3.039091         Validation Loss: 3.053910
Validation loss decreased (3.102261 --> 3.053910).  Saving model ...
Epoch: 84         Training Loss: 3.019383         Validation Loss: 3.142816
Epoch: 85         Training Loss: 3.032163         Validation Loss: 3.103264
Epoch: 86         Training Loss: 2.998780         Validation Loss: 3.067236
Epoch: 87         Training Loss: 3.001264         Validation Loss: 3.127470
Epoch: 88         Training Loss: 3.003762         Validation Loss: 3.157178
Epoch: 89         Training Loss: 2.974841         Validation Loss: 3.121075
Epoch: 90         Training Loss: 2.973026         Validation Loss: 3.185410
Epoch: 91         Training Loss: 2.951642         Validation Loss: 3.264283
Epoch: 92         Training Loss: 2.945361         Validation Loss: 3.008729
Validation loss decreased (3.053910 --> 3.008729).  Saving model ...
Epoch: 93         Training Loss: 2.931276         Validation Loss: 3.167958
Epoch: 94         Training Loss: 2.938843         Validation Loss: 3.058688
Epoch: 95         Training Loss: 2.953446         Validation Loss: 3.130493
Epoch: 96         Training Loss: 2.907232         Validation Loss: 3.094703
Epoch: 97         Training Loss: 2.908273         Validation Loss: 3.066957
Epoch: 98         Training Loss: 2.897566         Validation Loss: 3.319558
Epoch: 99         Training Loss: 2.878703         Validation Loss: 2.975676
Validation loss decreased (3.008729 --> 2.975676).  Saving model ...
Epoch: 100         Training Loss: 2.897788         Validation Loss: 2.991184
```

1.1.9 (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 [11]: 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:
                    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))

        # 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: 2.556358


Test Accuracy: 34% (427/1250)
```

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.

1.1.10 (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 [6]:
```
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes

from PIL import Image
import os
import numpy as np
import torch
from torchvision import datasets
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
import torch.optim as optim
import torch.nn as nn
from torchvision import models


data_directory = '/data/landmark_images/'
train_directory = os.path.join(data_directory, 'train')
test_directory = os.path.join(data_directory, 'test')
batch_size= 15
validation_size = 0.3

train_data_transform = transforms.Compose([transforms.Resize(256),
                                      transforms.CenterCrop(256),
                                      transforms.ToTensor(),
                                      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.

validation_data_transform = transforms.Compose([transforms.Resize(256),
                                      transforms.CenterCrop(256),
                                      transforms.ToTensor(),
                                      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.

train_data = datasets.ImageFolder(train_directory, transform=train_data_transform)
test_data = datasets.ImageFolder(test_directory, transform=validation_data_transform)

number_of_training_samples = len(train_data)
indices_of_entire_ds = list(range(number_of_training_samples))
np.random.shuffle(indices_of_entire_ds)
split_index = int(np.floor(validation_size * number_of_training_samples))
train_idx, valid_idx = indices_of_entire_ds[split_index:], indices_of_entire_ds[:split_i
```

```
train_random_sampler = SubsetRandomSampler(train_idx)
validation_random_sampler = SubsetRandomSampler(valid_idx)
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,sampler=tra
validation_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,sample
test_loader =  torch.utils.data.DataLoader(test_data, batch_size=batch_size, num_workers
classes = [classes_name.split(".")[1] for classes_name in train_data.classes]

loaders_transfer = {'train': train_loader, 'valid': validation_loader, 'test': test_load
```

1.1.11 (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 [7]:
```
## TODO: select loss function
criterion_transfer = nn.CrossEntropyLoss()


def get_optimizer_transfer(model):
    ## TODO: select and return optimizer
    return optim.SGD(model.classifier.parameters(), lr=0.01)
```

1.1.12 (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 [8]:
```
import torch.nn.functional as F

## TODO: Specify model architecture
use_cuda = torch.cuda.is_available()

model_transfer = models.vgg16(pretrained=True)

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

model_transfer.classifier[6] = nn.Linear( model_transfer.classifier[6].in_features , len


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

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

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 have used vgg16 as its widely used for basic image classification tasks without having much complexity. Since our data volume is small and relatively similar to the data set on which vgg16 were originally trained, we have only touched the final layer as per our need (number of classes).

But it seemed like torchvision 0.2.1 thats inbuilt in this workspace has issues around using transfer learning for similar cases, with similar kind of pre-trained models. We took the external reference as the following, and accordingly changed the code in train, validation and test methods. We tried installing torchvision 0.2.2 but has a lot of dependencies on other already installed packages, for example, moviepy, numpy etc. https://stackoverflow.com/questions/55145561/implementation-of-vgg16-on-pytorch-giving-size-mismatch-error

https://github.com/pytorch/vision/commit/83b2dfb2ebcd1b0694d46e3006ca96183c303706

1.1.13 (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 [9]: import torch.nn.functional as F
        # TODO: train the model and save the best model parameters at filepath 'model_transfer.p
        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: # load them in parallel
                        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() - tr
                    optimizer.zero_grad()

                    feat = model.features(data)
                    avg = F.adaptive_avg_pool2d(feat, (7,7))
```

```
            avg = avg.view(avg.size(0), -1)
            output = model.classifier(avg)

            #output = model(data)
            loss = criterion(output, target)
            loss.backward() # calculate gradient
            optimizer.step() # update wieghts
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data.item() - train

        ######################
        # 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:
                data, target = data.cuda(), target.cuda()
            ## TODO: update average validation loss

            feat = model.features(data)
            avg = F.adaptive_avg_pool2d(feat, (7,7))
            avg = avg.view(avg.size(0), -1)
            output = model.classifier(avg)

            #output = model(data)
            loss = criterion(output, target)
            valid_loss =valid_loss + ((1 / (batch_idx + 1)) * (loss.data.item() - valid_


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

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


    return model



num_epochs = 13
```

```
        model_transfer = train(num_epochs, loaders_transfer, model_transfer, get_optimizer_trans
                               criterion_transfer, use_cuda, 'model_transfer.pt')


        #-#-# 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: 2.166322         Validation Loss: 1.626524
Validation loss decreased (inf --> 1.626524).  Saving model ...
Epoch: 2         Training Loss: 1.161654         Validation Loss: 1.195760
Validation loss decreased (1.626524 --> 1.195760).  Saving model ...
Epoch: 3         Training Loss: 0.834795         Validation Loss: 1.376087
Epoch: 4         Training Loss: 0.609353         Validation Loss: 1.118006
Validation loss decreased (1.195760 --> 1.118006).  Saving model ...
Epoch: 5         Training Loss: 0.461519         Validation Loss: 1.104085
Validation loss decreased (1.118006 --> 1.104085).  Saving model ...
Epoch: 6         Training Loss: 0.349375         Validation Loss: 1.146080
Epoch: 7         Training Loss: 0.256757         Validation Loss: 1.165316
Epoch: 8         Training Loss: 0.203531         Validation Loss: 1.084754
Validation loss decreased (1.104085 --> 1.084754).  Saving model ...
Epoch: 9         Training Loss: 0.148263         Validation Loss: 1.286422
Epoch: 10         Training Loss: 0.135936         Validation Loss: 1.134916
Epoch: 11         Training Loss: 0.098735         Validation Loss: 1.158100
Epoch: 12         Training Loss: 0.084891         Validation Loss: 1.192996
Epoch: 13         Training Loss: 0.068694         Validation Loss: 1.173679
```

1.1.14 (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 [11]: 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:
                     data, target = data.cuda(), target.cuda()
```

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

            feat = model.features(data)
            avg = F.adaptive_avg_pool2d(feat, (7,7))
            avg = avg.view(avg.size(0), -1)
            output = model.classifier(avg)

            #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))

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

```
Test Loss: 0.900930


Test Accuracy: 77% (969/1250)
```

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.

1.1.15 (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`:

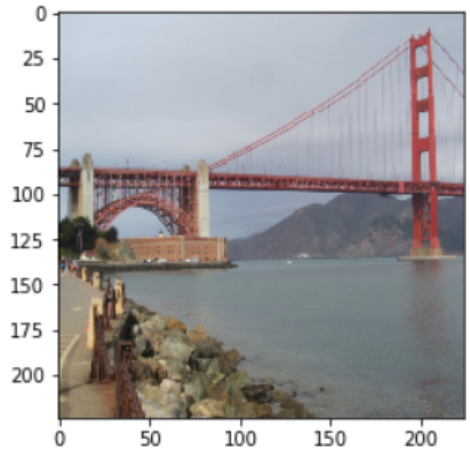
```
>>> predicted_landmarks = predict_landmarks('example_image.jpg', 3)
>>> print(predicted_landmarks)
['Golden Gate Bridge', 'Brooklyn Bridge', 'Sydney Harbour Bridge']
```

```
In [12]: 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
             top_k_classes = []
             model_transfer.eval()

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

             image= transform(Image.open(img_path))
             image.unsqueeze_(0) # doing unsqueezeing in-place

             if use_cuda:
                 image = image.cuda()


             output = model_transfer(image)
             values, indices = output.topk(k)

             for i in indices[0].tolist():
                 top_k_classes.append(classes[i])

             model_transfer.train()

             return top_k_classes


         # test on a sample image
         predict_landmarks('images/test/09.Golden_Gate_Bridge/190f3bae17c32c37.jpg', 5)

Out[12]: ['Golden_Gate_Bridge',
          'Sydney_Harbour_Bridge',
          'Forth_Bridge',
          'Brooklyn_Bridge',
          'Niagara_Falls']
```

1.1.16 (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!




```
Is this picture of the
Golden Gate Bridge, Brooklyn Bridge, or Sydney Harbour Bridge?
```

```
In [14]: import matplotlib.pyplot as plt
         %matplotlib inline

         def suggest_locations(img_path):
             # get landmark predictions
             predicted_landmarks = predict_landmarks(img_path, 3)

             ## TODO: display image and display landmark predictions
             plt.imshow(Image.open(img_path))
             plt.show()
             print('Possibly this picture is of the (amongst)',predicted_landmarks[0],',', predi



         # test on a sample image
         suggest_locations('images/test/09.Golden_Gate_Bridge/190f3bae17c32c37.jpg')
```


```
Possibly this picture is of the (amongst) Golden_Gate_Bridge , Forth_Bridge , or Sydney_Harbour_
```

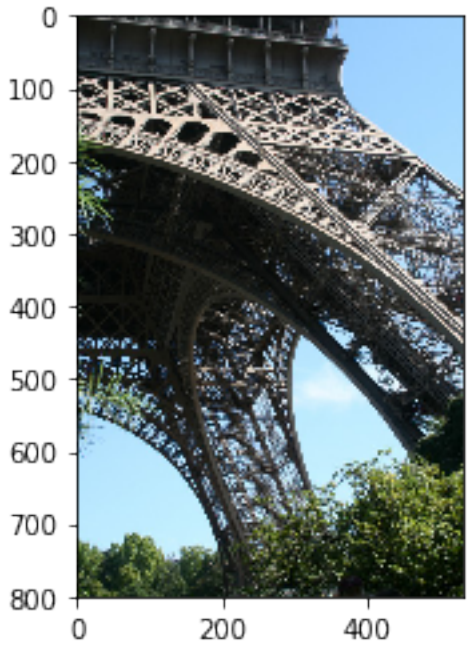
1.1.17 (IMPLEMENTATION) Test Your Algorithm

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.

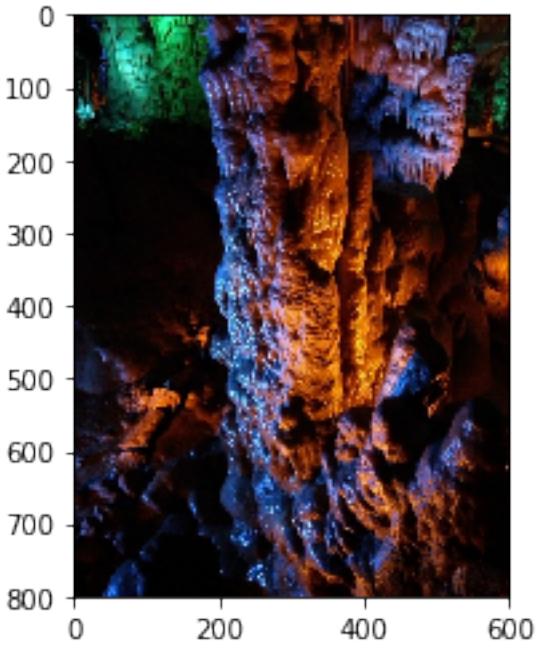
Question 4: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement) I think the output is better than what I expected. Possible improvements would include - 1. Collecting more training images on all possible classes and train the model. 2. Trying with varying number of transforms. 3. May be we can try with more sophisticated models like vgg19 or resnet50. 4. Obviously we can try training it with more number of epochs, or with a number of different regularization techniques.

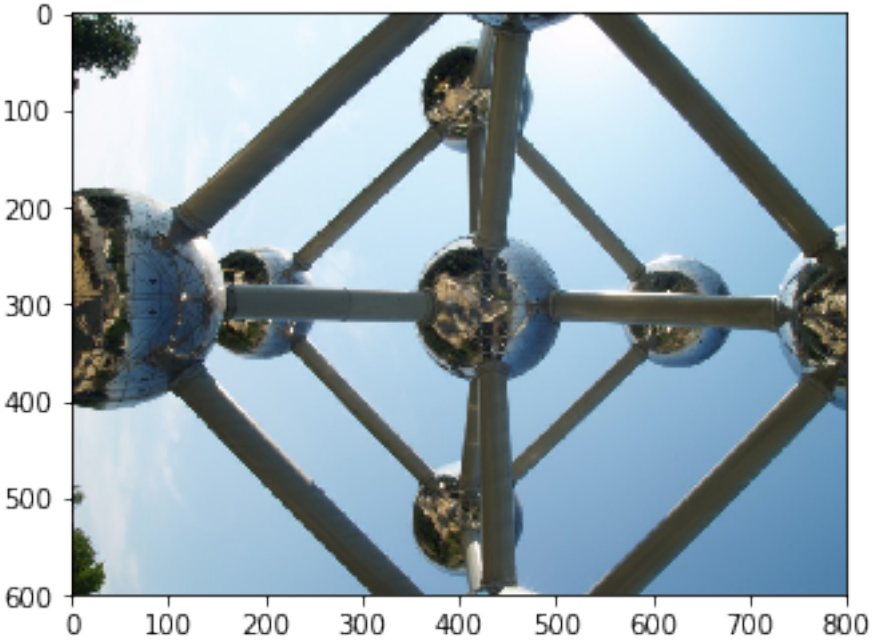
```
In [19]: ## TODO: Execute the `suggest_locations` function on
         ## at least 4 images on your computer.
         ## Feel free to use as many code cells as needed.

         suggest_locations('images/test/16.Eiffel_Tower/3828627c8730f160.jpg')
         suggest_locations('images/test/24.Soreq_Cave/18dbbad48a83a742.jpg')
         suggest_locations('images/test/37.Atomium/5ecb74282baee5aa.jpg')
         suggest_locations('images/test/41.Machu_Picchu/4336abf3179202f2.jpg')
```



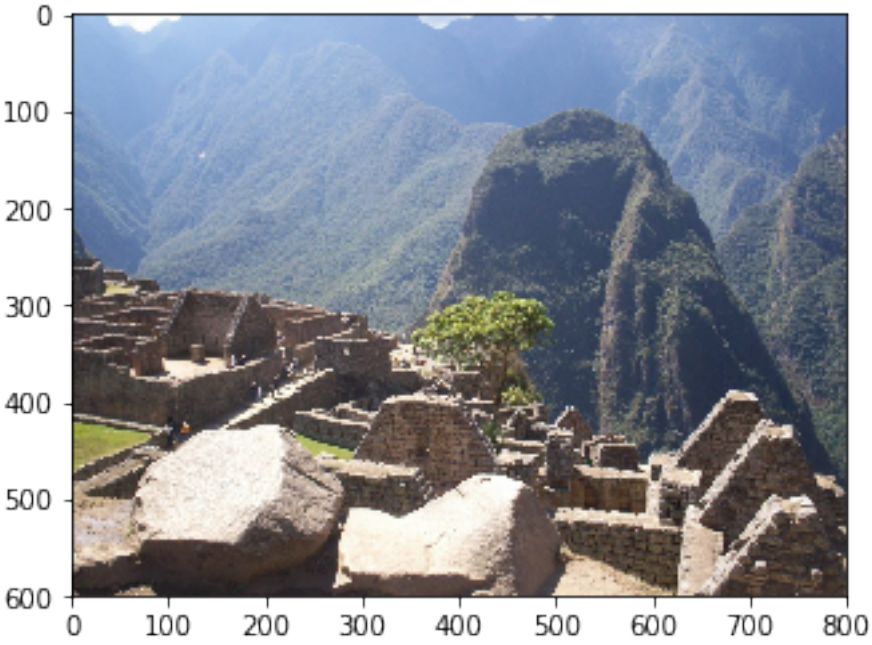

```
Possibly this picture is of the (amongst) Eiffel_Tower , Sydney_Harbour_Bridge , or Monumento_a_
```



```
Possibly this picture is of the (amongst) Soreq_Cave , Matterhorn , or Great_Barrier_Reef
```




```
Possibly this picture is of the (amongst) Atomium , Sydney_Harbour_Bridge , or Wroclaws_Dwarves
```



Possibly this picture is of the (amongst) Great_Wall_of_China , Machu_Picchu , or Niagara_Falls

In [ ]: