landmark

September 21, 2021

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]:
```
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
import numpy as np
import pandas as pd
import torch
import PIL
from torchvision import datasets
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
from torch.optim.lr_scheduler import StepLR

# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 20
# percentage of training set to use as validation
valid_size = 0.2
```

```
#Defining transform
m = [0.485, 0.456, 0.406]
stdev = [0.229, 0.224, 0.225]
transform_train = transforms.Compose([
                                #transforms.Resize((128, 128)),
                                transforms.RandomHorizontalFlip(),
                                transforms.RandomRotation(10),
                                transforms.RandomResizedCrop(128),
                                transforms.ToTensor(),
                                #transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                                transforms.Normalize(m, stdev)
                               ])

transform_test = transforms.Compose([transforms.Resize(128),
                                     transforms.CenterCrop(128),
                                     transforms.ToTensor(),
                                     transforms.Normalize(m, stdev)])

path_train = '/data/landmark_images/train'
path_test = '/data/landmark_images/test'
train_data = datasets.ImageFolder(root=path_train,  transform=transform_train)
test_data = datasets.ImageFolder(root=path_test,  transform=transform_test)
valid_data = datasets.ImageFolder(root=path_train,  transform=transform_test)

print(len(train_data))
print(len(test_data))

# obtain training indices that will be used for validation
num_train = len(train_data)
indices = list(range(num_train))
np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

# define samplers for obtaining training and validation batches
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,sampler=tra
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, num_workers=
valid_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,sampler=val

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

```
4996
1250
```

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: - I selected 128 as the size of image to start with as I think going to 256 may result in a too complicated model. By doing some search, I realized that I can use CenterCrop only on the test data and for trainig I should use RandomResizedCrop. - Yes, I decided to apply augumentation to make the model less dependant to rotation and scale. This also help to increase the accuracy and prevent from overfitting.

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

#def imshow(img):
    #img = img / 2 + 0.5  # unnormalize
    #plt.imshow(np.transpose(img, (1, 2, 0)))  # convert from Tensor image

## TODO: visualize a batch of the train data loader
# obtain one batch of training images


dataiter = iter(loaders_scratch['train'])
images, labels = dataiter.next()
images = images.numpy()
mn = images.min()
mx = images.max()
## the class names can be accessed at the `classes` attribute
## of your dataset object (e.g., `train_dataset.classes`)
# plot the images in the batch, along with the corresponding labels
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(10):
    ax = fig.add_subplot(2, 10/2, idx+1, xticks=[], yticks=[])
    #img = random.randint(0, training_size)
    plt.imshow((images[idx].transpose(1,2,0) - mn)/(mx-mn))
    ax.set_title(train_data.classes[labels[idx]])
```

1.1.3 Initialize use_cuda variable

```
In [2]: # useful variable that tells us whether we should use the GPU
        use_cuda = torch.cuda.is_available()
        use_cuda

Out[2]: True
```

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 [3]: ## TODO: select loss function
        import torch.nn as nn
        import torch.optim as optim
        criterion_scratch = nn.CrossEntropyLoss()

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

1.1.5 (IMPLEMENTATION) Model Architecture

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

```
In [4]: 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
                #Convolutional layers
                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.conv4 = nn.Conv2d(64, 128, 3, padding=1)
            # max pool layer
            self.pool = nn.MaxPool2d(2,2)
            # Linear layer
            self.fc1 = nn.Linear(128*8*8, 256)
            self.fc2 = nn.Linear(256, 50)
            # Dropout layer
            self.dropout = nn.Dropout(0.25)
                    # batch norm
            self.batchnorm1 = nn.BatchNorm2d(16)
            self.batchnorm2 = nn.BatchNorm2d(32)
            self.batchnorm3 = nn.BatchNorm2d(64)
            self.batchnorm4 = nn.BatchNorm2d(128)
            self.batchnorm5 = nn.BatchNorm1d(256)
            #self.batchnorm6 = nn.BatchNorm2d(50)



        def forward(self, x):
            ## Define forward behavior
            x = self.pool(F.relu(self.batchnorm1(self.conv1(x)))) # maxpool(relu(conv(x)))
            x = self.pool(F.relu(self.batchnorm2(self.conv2(x))))
            x = self.pool(F.relu(self.batchnorm3(self.conv3(x))))
            x = self.pool(F.relu(self.batchnorm4(self.conv4(x))))
            #flatten the output to be used in the first linear layer
            #print(x.shape)
            x = x.view(-1, 128*8*8)
            x = self.dropout(x)
            #x = F.relu(self.fc1(x))
            x = F.relu(self.batchnorm5(self.fc1(x)))
            x = self.dropout(x)
            x = self.fc2(x)
            return x

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

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


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

```
Net(
```

```
  (conv1): Conv2d(3, 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))
  (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=8192, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=50, bias=True)
  (dropout): Dropout(p=0.25)
  (batchnorm1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (batchnorm2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (batchnorm3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (batchnorm4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (batchnorm5): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
```

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

Answer:

I implemented four convolutional layer and two fully connected layers. This step required a little bit of troubleshooting :) By mentor help, I sussefully adjusted the size for each layer. In the first attempt, I had used BatchNorm only on the Covolitional layer but realized that it's necessary to apply it for the linear section as well.

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 [5]: optimizer = get_optimizer_scratch(model_scratch)
        scheduler = StepLR(optimizer, step_size = 2, gamma = 0.1)
        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):
                scheduler.step()
                # 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

        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model)
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update training loss
        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)
        # calculate the batch loss
        loss = criterion(output, target)
        # update average validation loss
        valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data.item() - valid


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



    # 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
                        valid_loss_min,
                        valid_loss))
                    #torch.save(model.state_dict(), 'model_cifar.pt')
                    torch.save(model.state_dict(), save_path)
                    valid_loss_min = valid_loss



            return model
```

In [15]:
```
        ################################################################
        ######### This section is for TROUBLESHOOTING ONLY ##########
        ################################################################

        valid_loss_min = np.Inf
        epoch = 20
        import torch.nn.functional as F


        for batch_idx, (data, target) in enumerate(loaders_scratch['train']):
            if batch_idx == 0:
                x=data
                print(batch_idx)
                print(data.shape)
                conv1 = nn.Conv2d(3, 16, 3, padding=1)
                conv2 = nn.Conv2d(16, 32, 3, padding=1)
                conv3 = nn.Conv2d(32, 64, 3, padding=1)
                conv4 = nn.Conv2d(64, 128, 3, padding=1)
                # max pool layer
                pool = nn.MaxPool2d(2,2)
                # Linear layer
                fc1 = nn.Linear(128*8*8, 256)
                fc2 = nn.Linear(256, 50)
                # Dropout layer
                dropout = nn.Dropout(0.25)
                        # batch norm
                batchnorm1 = nn.BatchNorm2d(16)
                batchnorm2 = nn.BatchNorm2d(32)
                batchnorm3 = nn.BatchNorm2d(64)
                batchnorm4 = nn.BatchNorm2d(128)
                batchnorm5 = nn.BatchNorm1d(256)
```

```
            x = pool(F.relu(batchnorm1(conv1(x))))
            print('Tensor size after first conv is   ', x.shape)
            x = pool(F.relu(batchnorm2(conv2(x))))
            print('Tensor size after second conv is   ', x.shape)
            x = pool(F.relu(batchnorm3(conv3(x))))
            print('Tensor size after thirs conv is   ', x.shape)
            x = pool(F.relu(conv4(x)))
            print('Tensor size after fourth conv is   ', x.shape)

            x = x.view(-1, 128*8*8)
            print('Tensor size after resizing is  ', x.shape)
            y = F.relu(fc1(x))
            print('Tensor size after first linear w/o drop is ', y.shape)
            x = dropout(x)
            #x = F.relu(fc1(x))
            x = batchnorm5(F.relu(fc1(x)))
            print('Tensor size after first linear is   ', x.shape)
            x = dropout(x)
            x = fc2(x)
            print('Tensor size after second linear is   ', x.shape)
            print('Target shap is   ', target.shape)

        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
```

```
0
torch.Size([20, 3, 128, 128])
Tensor size after first conv is    torch.Size([20, 16, 64, 64])
Tensor size after second conv is    torch.Size([20, 32, 32, 32])
Tensor size after thirs conv is    torch.Size([20, 64, 16, 16])
Tensor size after fourth conv is    torch.Size([20, 128, 8, 8])
Tensor size after resizing is   torch.Size([20, 8192])
Tensor size after first linear w/o drop is  torch.Size([20, 256])
Tensor size after first linear is    torch.Size([20, 256])
Tensor size after second linear is    torch.Size([20, 50])
Target shap is    torch.Size([20])
```

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 [7]: def custom_weight_init(m):
```

```
            ## TODO: implement a weight initialization strategy
            classname = m.__class__.__name__

            if classname.find('Linear') != -1:
                n = m.in_features
                y = 1.0/np.sqrt(n)
                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(20, loaders_scratch, model_scratch, get_optimizer_scratch(model_sc
```

```
Epoch: 1         Training Loss: 3.786397         Validation Loss: 3.667379
Validation loss decreased (inf --> 3.667379).  Saving model ...
Epoch: 2         Training Loss: 3.615521         Validation Loss: 3.539209
Validation loss decreased (3.667379 --> 3.539209).  Saving model ...
Epoch: 3         Training Loss: 3.501212         Validation Loss: 3.463693
Validation loss decreased (3.539209 --> 3.463693).  Saving model ...
Epoch: 4         Training Loss: 3.426472         Validation Loss: 3.400615
Validation loss decreased (3.463693 --> 3.400615).  Saving model ...
Epoch: 5         Training Loss: 3.368268         Validation Loss: 3.335250
Validation loss decreased (3.400615 --> 3.335250).  Saving model ...
Epoch: 6         Training Loss: 3.302021         Validation Loss: 3.256348
Validation loss decreased (3.335250 --> 3.256348).  Saving model ...
Epoch: 7         Training Loss: 3.251411         Validation Loss: 3.236152
Validation loss decreased (3.256348 --> 3.236152).  Saving model ...
Epoch: 8         Training Loss: 3.194193         Validation Loss: 3.253657
Epoch: 9         Training Loss: 3.162100         Validation Loss: 3.116988
Validation loss decreased (3.236152 --> 3.116988).  Saving model ...
Epoch: 10         Training Loss: 3.095539         Validation Loss: 3.088632
Validation loss decreased (3.116988 --> 3.088632).  Saving model ...
Epoch: 11         Training Loss: 3.038417         Validation Loss: 3.053650
Validation loss decreased (3.088632 --> 3.053650).  Saving model ...
Epoch: 12         Training Loss: 3.022779         Validation Loss: 3.046879
Validation loss decreased (3.053650 --> 3.046879).  Saving model ...
Epoch: 13         Training Loss: 2.968057         Validation Loss: 3.004952
Validation loss decreased (3.046879 --> 3.004952).  Saving model ...
Epoch: 14         Training Loss: 2.927338         Validation Loss: 3.044567
Epoch: 15         Training Loss: 2.898842         Validation Loss: 2.877755
Validation loss decreased (3.004952 --> 2.877755).  Saving model ...
Epoch: 16         Training Loss: 2.867038         Validation Loss: 2.927622
Epoch: 17         Training Loss: 2.831774         Validation Loss: 2.914153
Epoch: 18         Training Loss: 2.800319         Validation Loss: 2.896927
Epoch: 19         Training Loss: 2.806604         Validation Loss: 2.849400
Validation loss decreased (2.877755 --> 2.849400).  Saving model ...
Epoch: 20         Training Loss: 2.764293         Validation Loss: 2.883046
```

1.1.8 (IMPLEMENTATION) Train and Validate the Model

Run the next code cell to train your model.

```
In [6]: ## TODO: you may change the number of epochs if you'd like,
        ## but changing it is not required
        num_epochs = 40
        #-#-# 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: 3.803716         Validation Loss: 3.682843
Validation loss decreased (inf --> 3.682843).  Saving model ...
Epoch: 2         Training Loss: 3.645629         Validation Loss: 3.571265
Validation loss decreased (3.682843 --> 3.571265).  Saving model ...
Epoch: 3         Training Loss: 3.546929         Validation Loss: 3.497993
Validation loss decreased (3.571265 --> 3.497993).  Saving model ...
Epoch: 4         Training Loss: 3.469458         Validation Loss: 3.422358
Validation loss decreased (3.497993 --> 3.422358).  Saving model ...
Epoch: 5         Training Loss: 3.389051         Validation Loss: 3.380331
Validation loss decreased (3.422358 --> 3.380331).  Saving model ...
Epoch: 6         Training Loss: 3.327836         Validation Loss: 3.301177
Validation loss decreased (3.380331 --> 3.301177).  Saving model ...
Epoch: 7         Training Loss: 3.286047         Validation Loss: 3.276750
Validation loss decreased (3.301177 --> 3.276750).  Saving model ...
Epoch: 8         Training Loss: 3.236435         Validation Loss: 3.213975
Validation loss decreased (3.276750 --> 3.213975).  Saving model ...
Epoch: 9         Training Loss: 3.160006         Validation Loss: 3.174861
Validation loss decreased (3.213975 --> 3.174861).  Saving model ...
Epoch: 10         Training Loss: 3.122537         Validation Loss: 3.199974
Epoch: 11         Training Loss: 3.079614         Validation Loss: 3.123021
Validation loss decreased (3.174861 --> 3.123021).  Saving model ...
Epoch: 12         Training Loss: 3.034275         Validation Loss: 3.065593
Validation loss decreased (3.123021 --> 3.065593).  Saving model ...
Epoch: 13         Training Loss: 3.020999         Validation Loss: 3.043473
```

```
Validation loss decreased (3.065593 --> 3.043473).  Saving model ...
Epoch: 14         Training Loss: 2.942043         Validation Loss: 3.024624
Validation loss decreased (3.043473 --> 3.024624).  Saving model ...
Epoch: 15         Training Loss: 2.920042         Validation Loss: 2.960574
Validation loss decreased (3.024624 --> 2.960574).  Saving model ...
Epoch: 16         Training Loss: 2.879425         Validation Loss: 2.909041
Validation loss decreased (2.960574 --> 2.909041).  Saving model ...
Epoch: 17         Training Loss: 2.882089         Validation Loss: 2.910191
Epoch: 18         Training Loss: 2.848674         Validation Loss: 2.912469
Epoch: 19         Training Loss: 2.823501         Validation Loss: 2.879029
Validation loss decreased (2.909041 --> 2.879029).  Saving model ...
Epoch: 20         Training Loss: 2.805261         Validation Loss: 2.828003
Validation loss decreased (2.879029 --> 2.828003).  Saving model ...
Epoch: 21         Training Loss: 2.770945         Validation Loss: 2.748629
Validation loss decreased (2.828003 --> 2.748629).  Saving model ...
Epoch: 22         Training Loss: 2.737035         Validation Loss: 2.761140
Epoch: 23         Training Loss: 2.721711         Validation Loss: 2.880503
Epoch: 24         Training Loss: 2.704365         Validation Loss: 2.827674
Epoch: 25         Training Loss: 2.656323         Validation Loss: 2.724364
Validation loss decreased (2.748629 --> 2.724364).  Saving model ...
Epoch: 26         Training Loss: 2.656662         Validation Loss: 2.702653
Validation loss decreased (2.724364 --> 2.702653).  Saving model ...
Epoch: 27         Training Loss: 2.637547         Validation Loss: 2.741078
Epoch: 28         Training Loss: 2.590465         Validation Loss: 2.712598
Epoch: 29         Training Loss: 2.589237         Validation Loss: 2.658913
Validation loss decreased (2.702653 --> 2.658913).  Saving model ...
Epoch: 30         Training Loss: 2.563687         Validation Loss: 2.702556
Epoch: 31         Training Loss: 2.558216         Validation Loss: 2.745108
Epoch: 32         Training Loss: 2.558704         Validation Loss: 2.591881
Validation loss decreased (2.658913 --> 2.591881).  Saving model ...
Epoch: 33         Training Loss: 2.490823         Validation Loss: 2.673256
Epoch: 34         Training Loss: 2.491973         Validation Loss: 2.751335
Epoch: 35         Training Loss: 2.461594         Validation Loss: 2.643678
Epoch: 36         Training Loss: 2.448619         Validation Loss: 2.846660
Epoch: 37         Training Loss: 2.438218         Validation Loss: 2.644690
Epoch: 38         Training Loss: 2.447191         Validation Loss: 2.561375
Validation loss decreased (2.591881 --> 2.561375).  Saving model ...
Epoch: 39         Training Loss: 2.403877         Validation Loss: 2.575696
Epoch: 40         Training Loss: 2.396264         Validation Loss: 2.553553
Validation loss decreased (2.561375 --> 2.553553).  Saving model ...
```

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 [7]: 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: 2.163731


Test Accuracy: 44% (558/1250)
```

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 [8]: ### TODO: Write data loaders for training, validation, and test sets
        ## Specify appropriate transforms, and batch_sizes
        import numpy as np
        import pandas as pd
        import torch
        import PIL
        from torchvision import datasets
        import torchvision.transforms as transforms
        from torch.utils.data.sampler import SubsetRandomSampler
        from torch.optim.lr_scheduler import StepLR

        # number of subprocesses to use for data loading
        num_workers = 0
        # how many samples per batch to load
        batch_size = 20
        # percentage of training set to use as validation
        valid_size = 0.2
        #Defining transform
        m = [0.485, 0.456, 0.406]
        stdev = [0.229, 0.224, 0.225]
        transform_train = transforms.Compose([
                                        #transforms.Resize((128, 128)),
                                        transforms.RandomHorizontalFlip(),
                                        transforms.RandomRotation(10),
                                        transforms.RandomResizedCrop(224),
                                        transforms.ToTensor(),
                                        #transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                                        transforms.Normalize(m, stdev)
                                       ])

        transform_test = transforms.Compose([transforms.Resize(224),
                                             transforms.CenterCrop(224),
                                             transforms.ToTensor(),
                                             transforms.Normalize(m, stdev)])
```

```
path_train = '/data/landmark_images/train'
path_test = '/data/landmark_images/test'
train_data = datasets.ImageFolder(root=path_train,  transform=transform_train)
test_data = datasets.ImageFolder(root=path_test,  transform=transform_test)
valid_data = datasets.ImageFolder(root=path_train,  transform=transform_test)

print(len(train_data))
print(len(test_data))

# obtain training indices that will be used for validation
num_train = len(train_data)
indices = list(range(num_train))
np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

# define samplers for obtaining training and validation batches
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,sampler=tra
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, num_workers=
valid_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,sampler=val

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

```
4996
1250
```

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 [9]:
```
## TODO: select loss function
import torch.nn as nn
import torch.optim as optim

criterion_transfer = nn.CrossEntropyLoss()
```

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 [10]:
```
## TODO: Specify model architecture
import torchvision
```

```
model_transfer = torchvision.models.vgg16(pretrained=True)

# Freeze training for all "features" layers
for param in model_transfer.features.parameters():
    param.requires_grad = False

n_inputs = model_transfer.classifier[6].in_features
model_transfer.classifier[6] = torch.nn.Linear(n_inputs,50)
print(model_transfer.classifier[6])

def get_optimizer_transfer(model):
    optimizer = optim.Adam(model.classifier.parameters(), lr=0.001)
    return optimizer

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:05<00:00, 104326870.20it/s]


Linear(in_features=4096, out_features=50, bias=True)
```

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:

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 [11]: # TODO: train the model and save the best model parameters at filepath 'model_transfer.

         # In here I am using the same traing as the scrach model.
         optimizer = get_optimizer_transfer(model_transfer)
         scheduler = StepLR(optimizer, step_size = 2, gamma = 0.1)
         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):
                 scheduler.step()
                 # 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() - t

    # clear the gradients of all optimized variables
    optimizer.zero_grad()
    # forward pass: compute predicted outputs by passing inputs to the model)
    output = model(data)
    loss = criterion(output, target)
    loss.backward()
    # perform a single optimization step (parameter update)
    optimizer.step()
    # update training loss
    train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data.item() - trai

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


# calculate average losses--
```

```
                # This section caused the losses to be very low. By mentor suggestion, I commen
                #train_loss = train_loss/len(train_loader.dataset)
                #valid_loss = valid_loss/len(valid_loader.dataset)



                # 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('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
                        valid_loss_min,
                        valid_loss))
                    #torch.save(model.state_dict(), 'model_cifar.pt')
                    torch.save(model.state_dict(), save_path)
                    valid_loss_min = valid_loss



            return model
```

In [12]:
```
num_epochs=15

model_transfer = train(num_epochs, loaders_transfer, model_transfer, get_optimizer_tran
                       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: 3.112411         Validation Loss: 2.324054
Validation loss decreased (inf --> 2.324054).  Saving model ...
Epoch: 2         Training Loss: 2.846022         Validation Loss: 1.987040
Validation loss decreased (2.324054 --> 1.987040).  Saving model ...
Epoch: 3         Training Loss: 2.762999         Validation Loss: 1.989357
Epoch: 4         Training Loss: 2.690881         Validation Loss: 1.950255
Validation loss decreased (1.987040 --> 1.950255).  Saving model ...
Epoch: 5         Training Loss: 2.550693         Validation Loss: 1.917758
Validation loss decreased (1.950255 --> 1.917758).  Saving model ...
Epoch: 6         Training Loss: 2.435336         Validation Loss: 1.953481
Epoch: 7         Training Loss: 2.512679         Validation Loss: 1.906693
Validation loss decreased (1.917758 --> 1.906693).  Saving model ...
```

```
Epoch: 8         Training Loss: 2.450354         Validation Loss: 1.995157
Epoch: 9         Training Loss: 2.300106         Validation Loss: 1.981727
Epoch: 10         Training Loss: 2.310773         Validation Loss: 1.924283
Epoch: 11         Training Loss: 2.272328         Validation Loss: 1.857130
Validation loss decreased (1.906693 --> 1.857130).  Saving model ...
Epoch: 12         Training Loss: 2.312359         Validation Loss: 1.942193
Epoch: 13         Training Loss: 2.275644         Validation Loss: 1.826124
Validation loss decreased (1.857130 --> 1.826124).  Saving model ...
Epoch: 14         Training Loss: 2.342832         Validation Loss: 1.899710
Epoch: 15         Training Loss: 2.304256         Validation Loss: 1.969837
```

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 [13]:
```
def test(loaders, model, criterion, use_cuda):

    # monitor test loss and accuracy
    test_loss = 0.0
    correct = 0.0
    total = 0.0

    # 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))
```

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

```
Test Loss: 1.365151


Test Accuracy: 66% (830/1250)
```

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 [14]: 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
             img = Image.open(img_path)
             img_trans = transform_test(img).unsqueeze(0)
             if use_cuda:
                 transformed_img = img_trans.cuda()

             output = model_transfer(transformed_img)
             output_cpu = output.cpu() # Errors out w/o moving to CPU
             _,x = torch.topk(output_cpu,k)
             d = _.detach().numpy()
             p = x.detach().numpy()
             classes = train_data.classes
             j = []
             for i ,k in enumerate(classes):
                 if i in p:
```

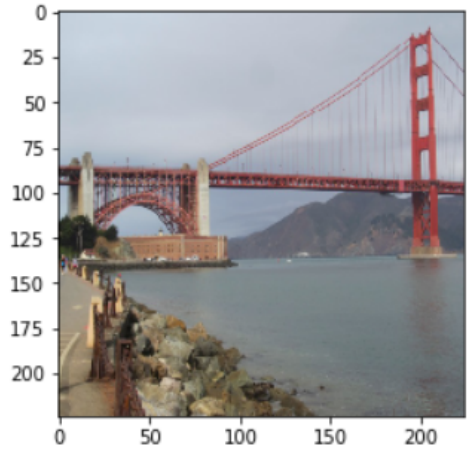
```
                j.append(classes[i])
        return j

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

```
Out[14]: ['03.Dead_Sea',
          '09.Golden_Gate_Bridge',
          '28.Sydney_Harbour_Bridge',
          '30.Brooklyn_Bridge',
          '38.Forth_Bridge']
```

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`.

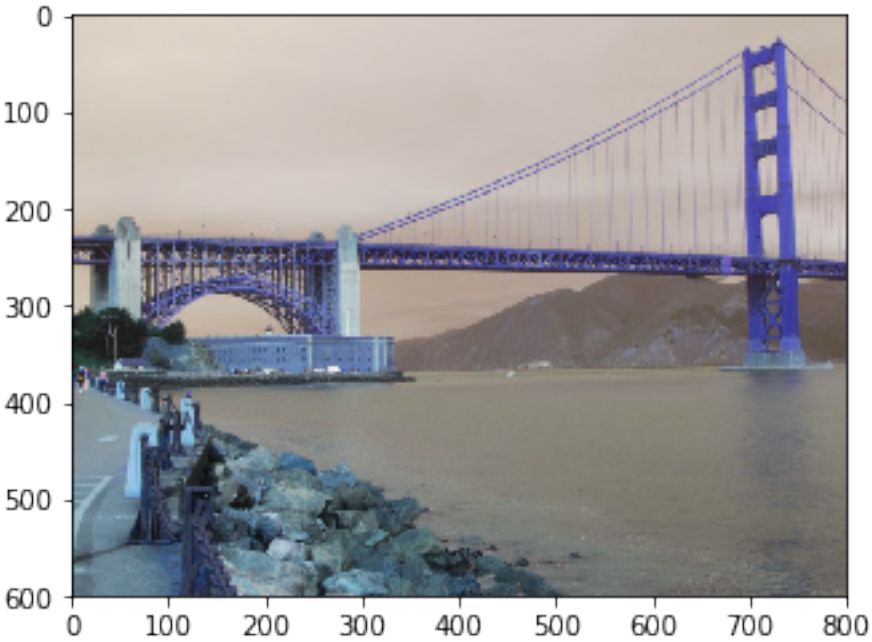
Some sample output for `suggest_locations` is provided below, but feel free to design your own user experience!




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

```
In [26]: import matplotlib.pyplot as plt
         def suggest_locations(img_path):
             # get landmark predictions
             ## TODO: display image and display landmark predictions
             x = cv2.imread(img_path)
             plt.imshow(x)
             plt.show()
             predicted_landmarks = predict_landmarks(img_path, 3)

             for predicted_landmark in predicted_landmarks:
                 print(predicted_landmark)
```

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




```
09.Golden_Gate_Bridge
28.Sydney_Harbour_Bridge
38.Forth_Bridge
```

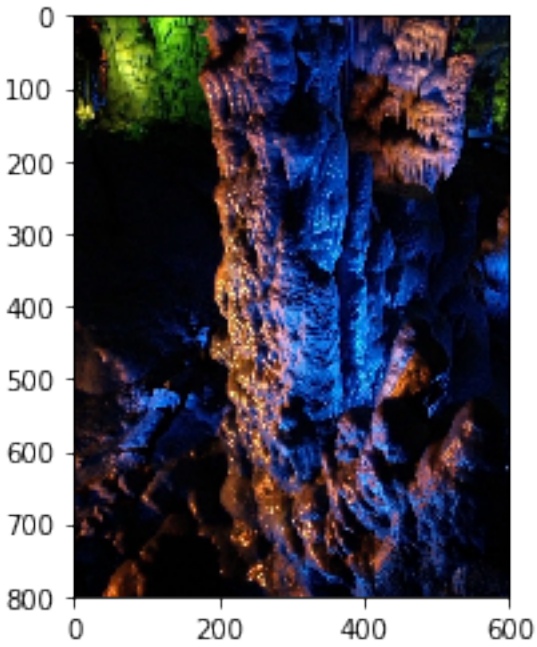
1.1.17 (IMPLEMENTATION) Test Your Algorithm

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.

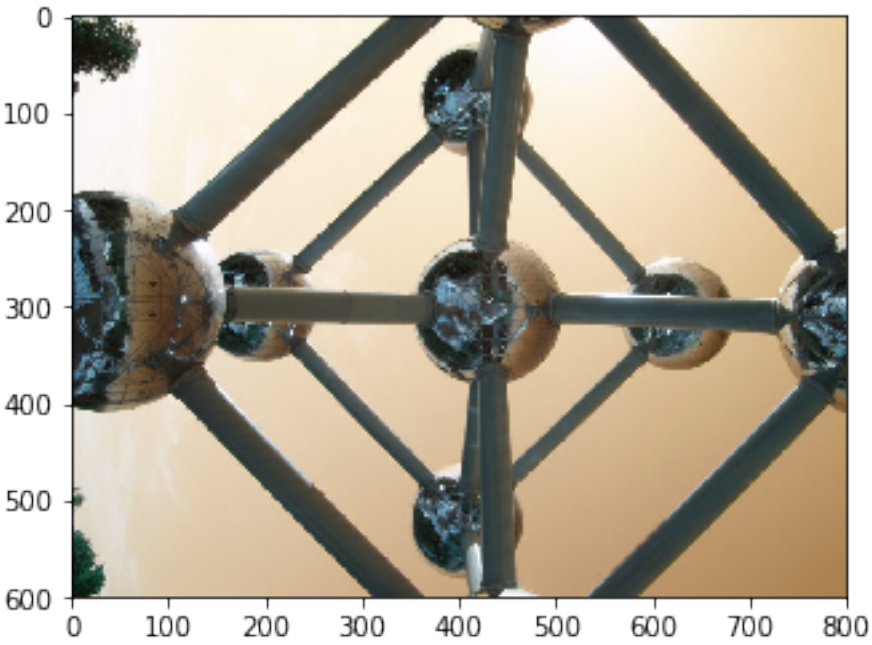
Answer: The ouput matchs with my expectation ;) However, I think adding more linear layer will help in better prediaction. Also, increasing the number of epoch and having more data in training datset will help the model to learn better.

```
In [21]: ## TODO: Execute the `suggest_locations` function on
         ## at least 4 images on your computer.
         ## Feel free to use as many code cells as needed.
         # test on a sample image
         suggest_locations('images/test/24.Soreq_Cave/18dbbad48a83a742.jpg')
```



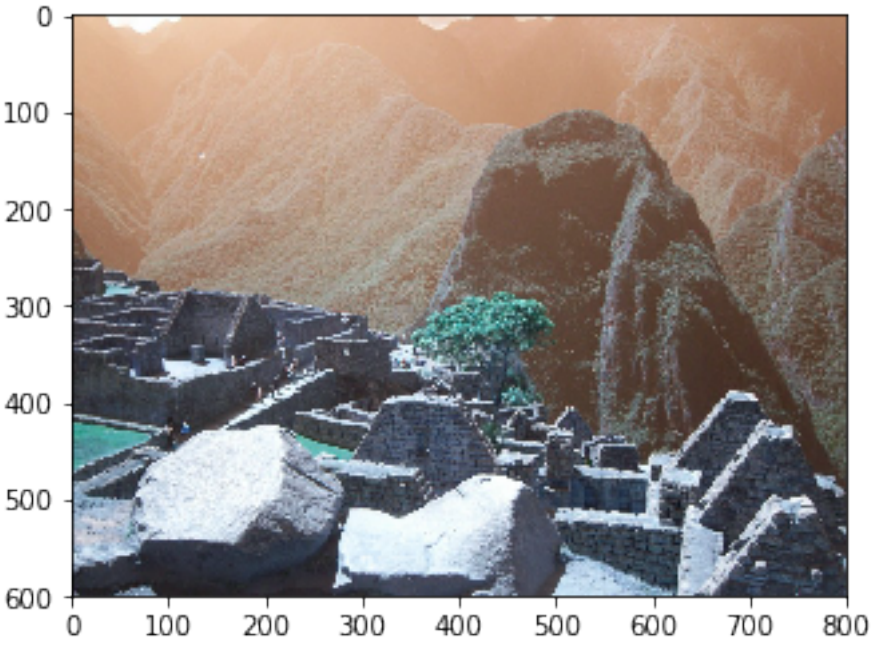

```
['05.London_Olympic_Stadium', '24.Soreq_Cave', '34.Great_Barrier_Reef']
```

```
In [28]: suggest_locations('images/test/37.Atomium/5ecb74282baee5aa.jpg')
```



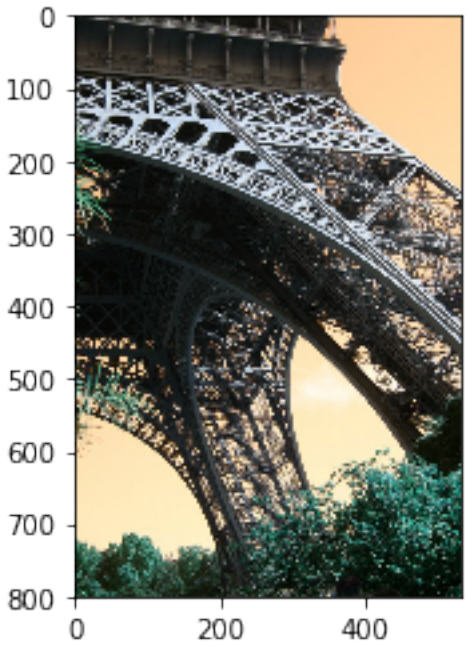
```
04.Wroclaws_Dwarves
05.London_Olympic_Stadium
37.Atomium
```

```
In [29]: suggest_locations('images/test/41.Machu_Picchu/4336abf3179202f2.jpg')
```




```
08.Grand_Canyon
41.Machu_Picchu
46.Great_Wall_of_China
```

```
In [30]: suggest_locations('images/test/16.Eiffel_Tower/3828627c8730f160.jpg')
```




```
16.Eiffel_Tower
28.Sydney_Harbour_Bridge
38.Forth_Bridge
```

In [ ]: