landmark

May 11, 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 [8]:
```
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
import os
import numpy as np
import torch
from torchvision import datasets
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
%matplotlib inline\

bs =32

validation_percentage = 0.1
data_dir = '/data/landmark_images/'
train_dir = os.path.join(data_dir, 'train')
test_dir = os.path.join(data_dir, 'test')
```

```
rand_trans = transforms.RandomApply([ transforms.RandomRotation(degrees=120),transforms.

data_transform = transforms.Compose([transforms.Resize(256),
                                     transforms.CenterCrop(256),
                                        rand_trans,
                                     transforms.ToTensor()])


#transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))

train_data = datasets.ImageFolder(train_dir, transform=data_transform)
test_data = datasets.ImageFolder(test_dir, transform=data_transform)

# print out some data stats
print('Num training images: ', len(train_data))
print('Num test images: ', len(test_data))

validation_indices= np.random.choice(len(train_data),(int)(len(train_data)*validation_pe
train_indices = list(set(range(len(train_data)))-set(validation_indices))



val_sample = SubsetRandomSampler(validation_indices)
train_sample = SubsetRandomSampler(train_indices)


train_loader = torch.utils.data.DataLoader(dataset=train_data, batch_size=bs, num_worker
val_loader = torch.utils.data.DataLoader(dataset=train_data, batch_size=bs, num_workers=


test_loader = torch.utils.data.DataLoader(dataset=test_data, batch_size=bs, num_workers=
loaders_scratch = {'train': train_loader, 'valid': val_loader, 'test': test_loader}


len(train_data.classes)
```

```
Num training images:  4996
Num test images:  1250
```

```
Out[8]: 50
```

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 defined some random transformations which include random rotations of 120 degrees and horizontal flips that will happen with probability 0.4 on each image. This way, the algorithm I will define later on will be robust against angle changes of the objects in the pictures. I also resized the images and cropped them so the model will recognize them more easily.

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

## 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`)


# obtain one batch of training images
dataiter = iter(train_loader)
images, labels = dataiter.next()
images = images.numpy()


# plot the images in the batch, along with the corresponding labels
fig = plt.figure(figsize=(35, 6))
for idx in np.arange(20):

    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])

    ax.imshow(np.squeeze(images[idx].T), cmap='ocean')

    ax.set_title(train_data.classes[labels[idx].item()])
```

1.1.3 Initialize use_cuda variable

```
In [5]: # useful variable that tells us whether we should use the GPU
        import torch
        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 [10]: ## TODO: select loss function
         import torch.nn as nn
         criterion_scratch = nn.CrossEntropyLoss()

         def get_optimizer_scratch(model):
             return torch.optim.SGD(model.parameters(), lr=0.01)
```

1.1.5 (IMPLEMENTATION) Model Architecture

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

```
In [6]: 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, 4, stride=2, padding=1)

                self.conv2 = nn.Conv2d(16, 32, 4, stride=2, padding=1)

                self.conv3 = nn.Conv2d(32, 64, 4, stride=2, padding=1)

                self.fc1 = nn.Linear(64 * 4 * 4, 500)

                self.fc2 = nn.Linear(500, 50)
                self.dropout_conv = nn.Dropout(0.2)
                self.dropout_fc = nn.Dropout(0.4)

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

```
    def forward(self, x):

        x= self.pool(F.relu(self.conv1(x)))
        x=self.dropout_conv(x)
        x= self.pool(F.relu(self.conv2(x)))
        x=self.dropout_conv(x)
        x= self.pool(F.relu(self.conv3(x)))
        x=self.dropout_conv(x)
        x = x.view(-1, 64 * 4 * 4)
        x= F.relu(self.fc1(x))
        x=self.dropout_fc(x)
        x= F.log_softmax(self.fc2(x),-1)

        ## Define forward behavior




        return x

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

# instantiate the CNN
model_scratch = Net()

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

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

Answer: The input images are 256x256. The reason why is that anything smaller would give me blurry pictures with low resolution and I thought this would decrease the performance of my CNN. Since this is quite large, I wanted to shrink the image sizes a lot (to 4x4, which I found that works well). I also know that MaxPool layers with input 2,2 works quite well. Therefore, since I wanted to use 2-3 conv layers, I also needed to shrink the images sizes by half at the convoluational layers. I was able to achieve this with stride=2, padding=1 and filter size=4. The number of channels are not too big and not too small, they are increasing as we move towards the fully connected layers, which is a good rule of thumb I found after doing some research. I used relu activation for the hidden layers as it was established that it was a good idea to do so by researchers. Since this is a classification task, the last layer outputs softmax probabilities of each class.

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

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

                 ###################
                 # train the model #
                 ###################
                 # 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()
                     optimizer.zero_grad()
                     output = model(data)
                     loss_train =criterion(output, target)
                     loss_train.backward()

                     optimizer.step()
                     train_loss += loss_train.item()*data.size(0)
                     ## 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



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

```
            loss_valid = criterion(output, target)
            valid_loss += loss_valid.item()*data.size(0)
            ## TODO: update average validation loss




        train_loss = train_loss/len(loaders['train'].sampler)
        valid_loss = valid_loss/len(loaders['valid'].sampler)
        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
            epoch,
            train_loss,
            valid_loss
            ))

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




    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 [7]: def custom_weight_init(m):

        classname = m.__class__.__name__
        # for every Linear layer in a model..
        if classname.find('Linear') != -1:
            # get the number of the inputs
            n = m.in_features
            y = (1.0/np.sqrt(n))
            m.weight.data.normal_(0, 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: 3.917403         Validation Loss: 3.912827
Validation loss decreased (inf --> 3.912827).  Saving model ...
Epoch: 2         Training Loss: 3.912786         Validation Loss: 3.913783
Epoch: 3         Training Loss: 3.911848         Validation Loss: 3.913150
Epoch: 4         Training Loss: 3.911920         Validation Loss: 3.913664
Epoch: 5         Training Loss: 3.910806         Validation Loss: 3.913195
Epoch: 6         Training Loss: 3.909389         Validation Loss: 3.912908
Epoch: 7         Training Loss: 3.908630         Validation Loss: 3.913267
Epoch: 8         Training Loss: 3.905667         Validation Loss: 3.912913
Epoch: 9         Training Loss: 3.906516         Validation Loss: 3.911979
Validation loss decreased (3.912827 --> 3.911979).  Saving model ...
Epoch: 10         Training Loss: 3.905401         Validation Loss: 3.911543
Validation loss decreased (3.911979 --> 3.911543).  Saving model ...
Epoch: 11         Training Loss: 3.903623         Validation Loss: 3.908658
Validation loss decreased (3.911543 --> 3.908658).  Saving model ...
Epoch: 12         Training Loss: 3.900033         Validation Loss: 3.907975
Validation loss decreased (3.908658 --> 3.907975).  Saving model ...
Epoch: 13         Training Loss: 3.897384         Validation Loss: 3.904930
Validation loss decreased (3.907975 --> 3.904930).  Saving model ...
Epoch: 14         Training Loss: 3.892976         Validation Loss: 3.899331
Validation loss decreased (3.904930 --> 3.899331).  Saving model ...
Epoch: 15         Training Loss: 3.887071         Validation Loss: 3.895616
Validation loss decreased (3.899331 --> 3.895616).  Saving model ...
Epoch: 16         Training Loss: 3.877583         Validation Loss: 3.885141
Validation loss decreased (3.895616 --> 3.885141).  Saving model ...
Epoch: 17         Training Loss: 3.867724         Validation Loss: 3.875566
Validation loss decreased (3.885141 --> 3.875566).  Saving model ...
Epoch: 18         Training Loss: 3.854391         Validation Loss: 3.861173
Validation loss decreased (3.875566 --> 3.861173).  Saving model ...
Epoch: 19         Training Loss: 3.834960         Validation Loss: 3.838820
Validation loss decreased (3.861173 --> 3.838820).  Saving model ...
Epoch: 20         Training Loss: 3.817158         Validation Loss: 3.817924
Validation loss decreased (3.838820 --> 3.817924).  Saving model ...
```

1.1.8 (IMPLEMENTATION) Train and Validate the Model

Run the next code cell to train your model.

In [8]:
```
## 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: 3.913108         Validation Loss: 3.911742
Validation loss decreased (inf --> 3.911742).  Saving model ...
Epoch: 2         Training Loss: 3.912395         Validation Loss: 3.912450
Epoch: 3         Training Loss: 3.912192         Validation Loss: 3.912679
Epoch: 4         Training Loss: 3.911504         Validation Loss: 3.912954
Epoch: 5         Training Loss: 3.910554         Validation Loss: 3.913313
Epoch: 6         Training Loss: 3.910354         Validation Loss: 3.913450
Epoch: 7         Training Loss: 3.909443         Validation Loss: 3.913878
Epoch: 8         Training Loss: 3.908899         Validation Loss: 3.913795
Epoch: 9         Training Loss: 3.909522         Validation Loss: 3.913274
Epoch: 10         Training Loss: 3.906681         Validation Loss: 3.913225
Epoch: 11         Training Loss: 3.905901         Validation Loss: 3.912534
Epoch: 12         Training Loss: 3.905820         Validation Loss: 3.911076
Validation loss decreased (3.911742 --> 3.911076).  Saving model ...
Epoch: 13         Training Loss: 3.902034         Validation Loss: 3.909670
Validation loss decreased (3.911076 --> 3.909670).  Saving model ...
Epoch: 14         Training Loss: 3.898165         Validation Loss: 3.906887
Validation loss decreased (3.909670 --> 3.906887).  Saving model ...
Epoch: 15         Training Loss: 3.896517         Validation Loss: 3.901667
Validation loss decreased (3.906887 --> 3.901667).  Saving model ...
Epoch: 16         Training Loss: 3.889351         Validation Loss: 3.896628
Validation loss decreased (3.901667 --> 3.896628).  Saving model ...
Epoch: 17         Training Loss: 3.882898         Validation Loss: 3.889695
Validation loss decreased (3.896628 --> 3.889695).  Saving model ...
Epoch: 18         Training Loss: 3.873829         Validation Loss: 3.879044
Validation loss decreased (3.889695 --> 3.879044).  Saving model ...
Epoch: 19         Training Loss: 3.857265         Validation Loss: 3.857816
Validation loss decreased (3.879044 --> 3.857816).  Saving model ...
Epoch: 20         Training Loss: 3.829709         Validation Loss: 3.828133
Validation loss decreased (3.857816 --> 3.828133).  Saving model ...
Epoch: 21         Training Loss: 3.793444         Validation Loss: 3.780047
```

```
Validation loss decreased (3.828133 --> 3.780047).  Saving model ...
Epoch: 22         Training Loss: 3.769166         Validation Loss: 3.749107
Validation loss decreased (3.780047 --> 3.749107).  Saving model ...
Epoch: 23         Training Loss: 3.740516         Validation Loss: 3.732328
Validation loss decreased (3.749107 --> 3.732328).  Saving model ...
Epoch: 24         Training Loss: 3.718624         Validation Loss: 3.706744
Validation loss decreased (3.732328 --> 3.706744).  Saving model ...
Epoch: 25         Training Loss: 3.703160         Validation Loss: 3.699284
Validation loss decreased (3.706744 --> 3.699284).  Saving model ...
Epoch: 26         Training Loss: 3.694438         Validation Loss: 3.676962
Validation loss decreased (3.699284 --> 3.676962).  Saving model ...
Epoch: 27         Training Loss: 3.675538         Validation Loss: 3.648266
Validation loss decreased (3.676962 --> 3.648266).  Saving model ...
Epoch: 28         Training Loss: 3.678811         Validation Loss: 3.686300
Epoch: 29         Training Loss: 3.650593         Validation Loss: 3.631205
Validation loss decreased (3.648266 --> 3.631205).  Saving model ...
Epoch: 30         Training Loss: 3.634835         Validation Loss: 3.622129
Validation loss decreased (3.631205 --> 3.622129).  Saving model ...
Epoch: 31         Training Loss: 3.628786         Validation Loss: 3.651247
Epoch: 32         Training Loss: 3.620309         Validation Loss: 3.609519
Validation loss decreased (3.622129 --> 3.609519).  Saving model ...
Epoch: 33         Training Loss: 3.602008         Validation Loss: 3.619942
Epoch: 34         Training Loss: 3.572397         Validation Loss: 3.600091
Validation loss decreased (3.609519 --> 3.600091).  Saving model ...
Epoch: 35         Training Loss: 3.566942         Validation Loss: 3.570428
Validation loss decreased (3.600091 --> 3.570428).  Saving model ...
Epoch: 36         Training Loss: 3.549710         Validation Loss: 3.518174
Validation loss decreased (3.570428 --> 3.518174).  Saving model ...
Epoch: 37         Training Loss: 3.537355         Validation Loss: 3.531786
Epoch: 38         Training Loss: 3.525239         Validation Loss: 3.552638
Epoch: 39         Training Loss: 3.508358         Validation Loss: 3.549731
Epoch: 40         Training Loss: 3.496640         Validation Loss: 3.473099
Validation loss decreased (3.518174 --> 3.473099).  Saving model ...
Epoch: 41         Training Loss: 3.474533         Validation Loss: 3.626767
Epoch: 42         Training Loss: 3.472121         Validation Loss: 3.464428
Validation loss decreased (3.473099 --> 3.464428).  Saving model ...
Epoch: 43         Training Loss: 3.452722         Validation Loss: 3.471341
Epoch: 44         Training Loss: 3.422054         Validation Loss: 3.471556
Epoch: 45         Training Loss: 3.420322         Validation Loss: 3.451356
Validation loss decreased (3.464428 --> 3.451356).  Saving model ...
Epoch: 46         Training Loss: 3.416286         Validation Loss: 3.411610
Validation loss decreased (3.451356 --> 3.411610).  Saving model ...
Epoch: 47         Training Loss: 3.392075         Validation Loss: 3.423788
Epoch: 48         Training Loss: 3.373356         Validation Loss: 3.351322
Validation loss decreased (3.411610 --> 3.351322).  Saving model ...
Epoch: 49         Training Loss: 3.347766         Validation Loss: 3.387995
Epoch: 50         Training Loss: 3.333390         Validation Loss: 3.358404
Epoch: 51         Training Loss: 3.342010         Validation Loss: 3.352396
```

```
Epoch: 52         Training Loss: 3.307847         Validation Loss: 3.337360
Validation loss decreased (3.351322 --> 3.337360).  Saving model ...
Epoch: 53         Training Loss: 3.297266         Validation Loss: 3.345652
Epoch: 54         Training Loss: 3.286739         Validation Loss: 3.373181
Epoch: 55         Training Loss: 3.271923         Validation Loss: 3.319673
Validation loss decreased (3.337360 --> 3.319673).  Saving model ...
Epoch: 56         Training Loss: 3.275303         Validation Loss: 3.273119
Validation loss decreased (3.319673 --> 3.273119).  Saving model ...
Epoch: 57         Training Loss: 3.249388         Validation Loss: 3.290715
Epoch: 58         Training Loss: 3.227256         Validation Loss: 3.290188
Epoch: 59         Training Loss: 3.205438         Validation Loss: 3.262897
Validation loss decreased (3.273119 --> 3.262897).  Saving model ...
Epoch: 60         Training Loss: 3.185375         Validation Loss: 3.267635
Epoch: 61         Training Loss: 3.196579         Validation Loss: 3.339338
Epoch: 62         Training Loss: 3.179892         Validation Loss: 3.267692
Epoch: 63         Training Loss: 3.168377         Validation Loss: 3.223944
Validation loss decreased (3.262897 --> 3.223944).  Saving model ...
Epoch: 64         Training Loss: 3.165520         Validation Loss: 3.228703
Epoch: 65         Training Loss: 3.132627         Validation Loss: 3.199172
Validation loss decreased (3.223944 --> 3.199172).  Saving model ...
Epoch: 66         Training Loss: 3.120271         Validation Loss: 3.198268
Validation loss decreased (3.199172 --> 3.198268).  Saving model ...
Epoch: 67         Training Loss: 3.111350         Validation Loss: 3.293342
Epoch: 68         Training Loss: 3.100813         Validation Loss: 3.245912
Epoch: 69         Training Loss: 3.074499         Validation Loss: 3.222383
Epoch: 70         Training Loss: 3.056954         Validation Loss: 3.235256
Epoch: 71         Training Loss: 3.052018         Validation Loss: 3.184625
Validation loss decreased (3.198268 --> 3.184625).  Saving model ...
Epoch: 72         Training Loss: 3.040391         Validation Loss: 3.202398
Epoch: 73         Training Loss: 2.996821         Validation Loss: 3.165954
Validation loss decreased (3.184625 --> 3.165954).  Saving model ...
Epoch: 74         Training Loss: 3.028141         Validation Loss: 3.140491
Validation loss decreased (3.165954 --> 3.140491).  Saving model ...
Epoch: 75         Training Loss: 2.985444         Validation Loss: 3.193607
Epoch: 76         Training Loss: 2.976217         Validation Loss: 3.191226
Epoch: 77         Training Loss: 2.969220         Validation Loss: 3.138085
Validation loss decreased (3.140491 --> 3.138085).  Saving model ...
Epoch: 78         Training Loss: 2.965296         Validation Loss: 3.120912
Validation loss decreased (3.138085 --> 3.120912).  Saving model ...
Epoch: 79         Training Loss: 2.939505         Validation Loss: 3.159173
Epoch: 80         Training Loss: 2.918778         Validation Loss: 3.099154
Validation loss decreased (3.120912 --> 3.099154).  Saving model ...
Epoch: 81         Training Loss: 2.921077         Validation Loss: 3.114194
Epoch: 82         Training Loss: 2.891597         Validation Loss: 3.129225
Epoch: 83         Training Loss: 2.898538         Validation Loss: 3.079901
Validation loss decreased (3.099154 --> 3.079901).  Saving model ...
Epoch: 84         Training Loss: 2.865020         Validation Loss: 3.222037
Epoch: 85         Training Loss: 2.855529         Validation Loss: 3.116891
```

```
Epoch: 86         Training Loss: 2.836380         Validation Loss: 3.078824
Validation loss decreased (3.079901 --> 3.078824).  Saving model ...
Epoch: 87         Training Loss: 2.813975         Validation Loss: 3.121116
Epoch: 88         Training Loss: 2.787430         Validation Loss: 3.073910
Validation loss decreased (3.078824 --> 3.073910).  Saving model ...
Epoch: 89         Training Loss: 2.808462         Validation Loss: 3.140983
Epoch: 90         Training Loss: 2.805036         Validation Loss: 3.119817
Epoch: 91         Training Loss: 2.774046         Validation Loss: 3.129371
Epoch: 92         Training Loss: 2.744580         Validation Loss: 3.020874
Validation loss decreased (3.073910 --> 3.020874).  Saving model ...
Epoch: 93         Training Loss: 2.742188         Validation Loss: 3.113821
Epoch: 94         Training Loss: 2.726771         Validation Loss: 3.048110
Epoch: 95         Training Loss: 2.717246         Validation Loss: 3.064472
Epoch: 96         Training Loss: 2.698709         Validation Loss: 3.176665
Epoch: 97         Training Loss: 2.690188         Validation Loss: 3.088131
Epoch: 98         Training Loss: 2.664875         Validation Loss: 2.999595
Validation loss decreased (3.020874 --> 2.999595).  Saving model ...
Epoch: 99         Training Loss: 2.639918         Validation Loss: 3.117057
Epoch: 100         Training Loss: 2.654259         Validation Loss: 3.134155
```

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: 2.974028


Test Accuracy: 26% (332/1250)
```

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 [17]:
```
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
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.nn as nn
```

```
data_dir = '/data/landmark_images/'
train_dir = os.path.join(data_dir, 'train')
test_dir = os.path.join(data_dir, 'test')

rand_trans = transforms.RandomApply([ transforms.RandomRotation(degrees=120),transforms
preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

bs=32

validation_percentage =0.2
train_data = datasets.ImageFolder(train_dir, transform=preprocess)
test_data = datasets.ImageFolder(test_dir, transform=preprocess)


validation_indices_trans= np.random.choice(len(train_data),(int)(len(train_data)*valida
train_indices_trans = list(set(range(len(train_data)))-set(validation_indices_trans))



val_sample_trans = SubsetRandomSampler(validation_indices_trans)
train_sample_trans=SubsetRandomSampler(train_indices_trans)

train_loader_trans = torch.utils.data.DataLoader(dataset=train_data, batch_size=bs, num
val_loader_trans = torch.utils.data.DataLoader(dataset=train_data, batch_size=bs, num_w


test_loader_trans = torch.utils.data.DataLoader(dataset=test_data, batch_size=bs, num_w


loaders_transfer = {'train': train_loader_trans, 'valid': val_loader_trans, 'test': tes



use_cuda = torch.cuda.is_available()
```

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


         def get_optimizer_transfer(model):
             return torch.optim.Adam(model.parameters(), lr=0.001)
```

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 [19]: ## TODO: Specify model architecture
         import torchvision.models as models
         model_transfer = models.resnet50(pretrained=True)




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

         if use_cuda:
             model_transfer = model_transfer.cuda()

         # print(model_transfer)
```

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: Resnet18 is a powerful, yet smaller architecture in comparison to many models offered by torchvision. At first, I wanted to give it a try. In the past, I used it for CIFAR100 dataset and it worked well. Then, I changed my mind and I decided that since Resnet50 is a bit larger and has better generalization capabilities, I'd give it a shot.

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

         n_inputs = model_transfer.fc.in_features
         last_layer = nn.Linear(n_inputs, 50)
         model_transfer.fc = last_layer

         model_transfer= model_transfer.to("cuda")
         model_to_save = train(15, loaders_transfer, model_transfer, get_optimizer_transfer(mode
```

```
        #-#-# 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: 3.238208         Validation Loss: 3.482963
Validation loss decreased (inf --> 3.482963).  Saving model ...
Epoch: 2         Training Loss: 2.438487         Validation Loss: 2.440921
Validation loss decreased (3.482963 --> 2.440921).  Saving model ...
Epoch: 3         Training Loss: 1.928435         Validation Loss: 2.224661
Validation loss decreased (2.440921 --> 2.224661).  Saving model ...
Epoch: 4         Training Loss: 1.649205         Validation Loss: 1.981084
Validation loss decreased (2.224661 --> 1.981084).  Saving model ...
Epoch: 5         Training Loss: 1.348152         Validation Loss: 2.711083
Epoch: 6         Training Loss: 1.229485         Validation Loss: 2.402664
Epoch: 7         Training Loss: 1.046359         Validation Loss: 1.914852
Validation loss decreased (1.981084 --> 1.914852).  Saving model ...
Epoch: 8         Training Loss: 0.854319         Validation Loss: 1.854457
Validation loss decreased (1.914852 --> 1.854457).  Saving model ...
Epoch: 9         Training Loss: 0.691123         Validation Loss: 1.979537
Epoch: 10         Training Loss: 0.612787         Validation Loss: 2.794797
Epoch: 11         Training Loss: 0.548776         Validation Loss: 1.970711
Epoch: 12         Training Loss: 0.412441         Validation Loss: 1.923302
Epoch: 13         Training Loss: 0.346307         Validation Loss: 2.261581
Epoch: 14         Training Loss: 0.518306         Validation Loss: 2.298184
Epoch: 15         Training Loss: 0.283708         Validation Loss: 1.902477
```

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

```
Test Loss: 1.537252


Test Accuracy: 62% (776/1250)
```

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']
```

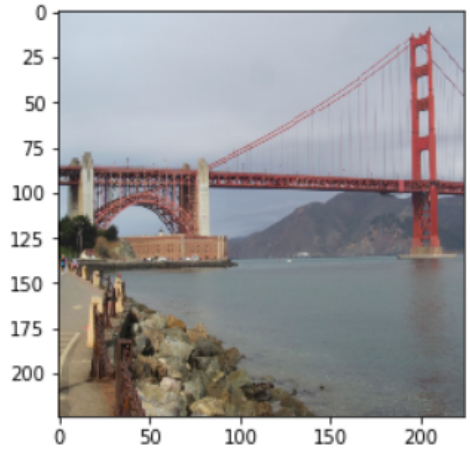
```
In [24]: import cv2
         from PIL import Image
         from torchvision.transforms import ToTensor
         ## the class names can be accessed at the `classes` attribute
         ## of your dataset object (e.g., `train_dataset.classes`)

         def predict_landmarks(img_path, k):
             im=Image.open(img_path)#.unsqueeze(0)

             preprocess = transforms.Compose([
                             transforms.Resize(256),
                             transforms.CenterCrop(224),
                             transforms.ToTensor(),
                             transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
                             ])

             img = preprocess(im)
             img = img.to("cuda")
             img = img.reshape(1, 3, 224,224)
             preds, indices=  torch.topk(model_transfer(img),5)
         #     print(indices.shape)
             res =[]
             for i in range(k):
                 res.append(train_data.classes[indices[0][i]])

             return res
             ## TODO: return the names of the top k landmarks predicted by the transfer learned



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

```
Out[24]: ['09.Golden_Gate_Bridge',
          '30.Brooklyn_Bridge',
          '03.Dead_Sea',
          '38.Forth_Bridge',
          '22.Moscow_Raceway']
```

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`.

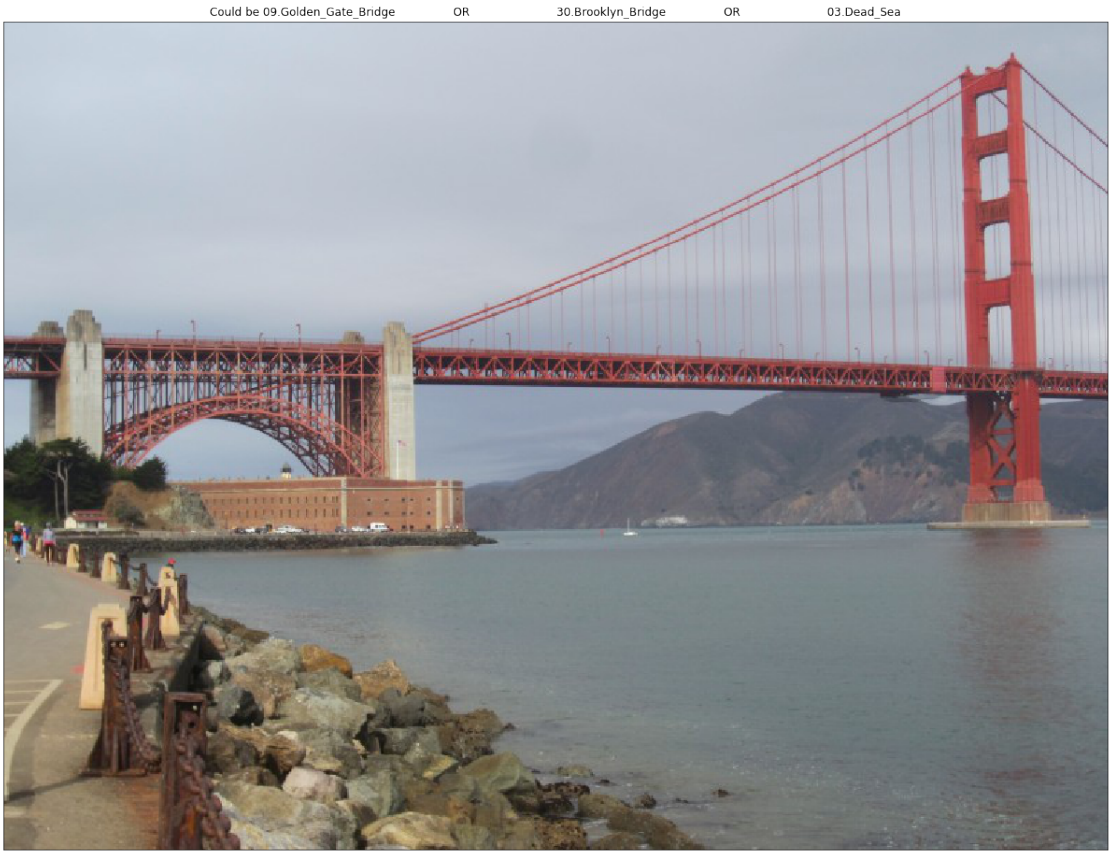
Some sample output for `suggest_locations` is provided below, but feel free to design your own user experience!




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

```
In [25]: 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
             im=Image.open(img_path)#.unsqueeze(0)

             preprocess = transforms.Compose([
                             transforms.Resize(256),
                             transforms.CenterCrop(224),
                             transforms.ToTensor(),
                             transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224,
                             ])

             img = preprocess(im)
             fig = plt.figure(figsize=(250, 70))


             ax = fig.add_subplot(2, 20/2, 1, xticks=[], yticks=[])

             ax.imshow(np.squeeze(im), cmap='ocean')
             title ="Could be "
```

```
    for i in range(len(predicted_landmarks)):
        title += predicted_landmarks[i]
        if i<len(predicted_landmarks)-1:
            title+="                OR                        "
    ax.set_title(title)


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



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.

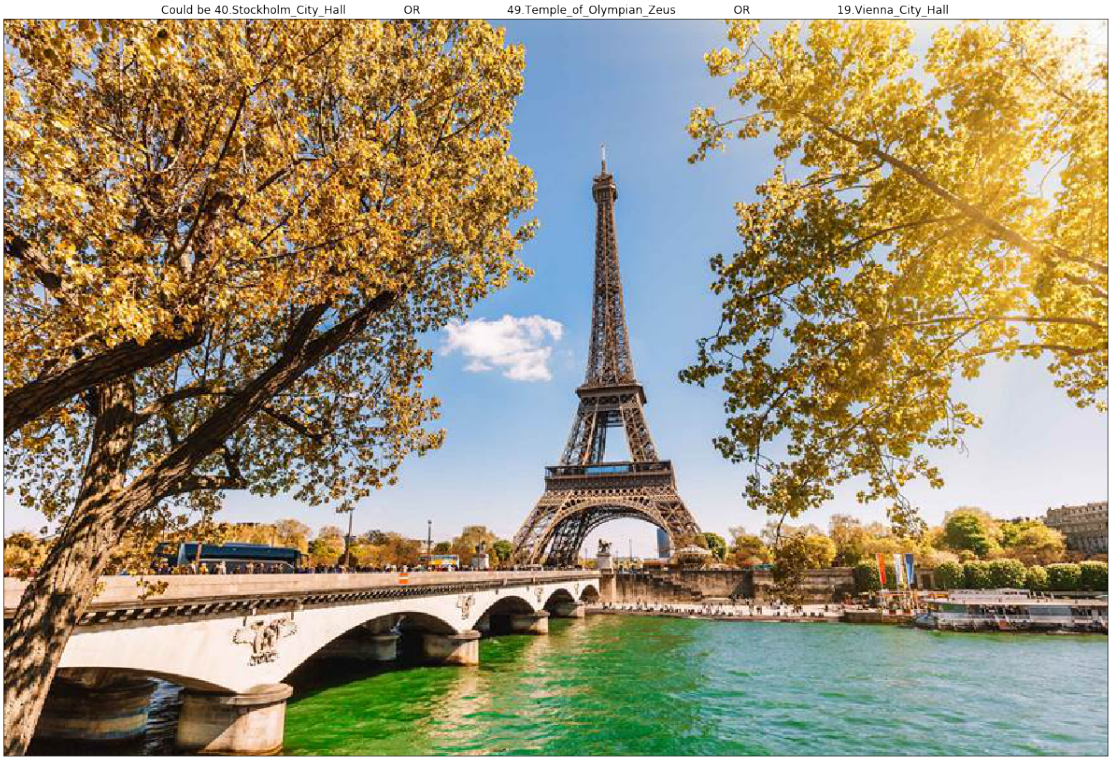
Answer: The algorithm was unable to classify a picture of the Eiffel Tower taken far away with trees in the picture and thought it was either Vienna or Stockholm city hall, both of which are pointy cathedrals. The third prediction was Temple of Olympian Zeus, which has long arches. The algorithm needs to learn to classify the long arhces better, which can be achieved with a more

complex architecture. Also, even though I applied some data transformations, the algorithm still could not recognize the picture of the tower taken far away. The dataset needs a more varied representations of the images.

Getting inspired by this output, I tried out pictures of vienna and stockholm city halls, which were very well recognized. I also tried a funny picture of the Temple of Olympian Zeus, which the algorithm was able to correctly identify. So overall, I am happy with the outcome.

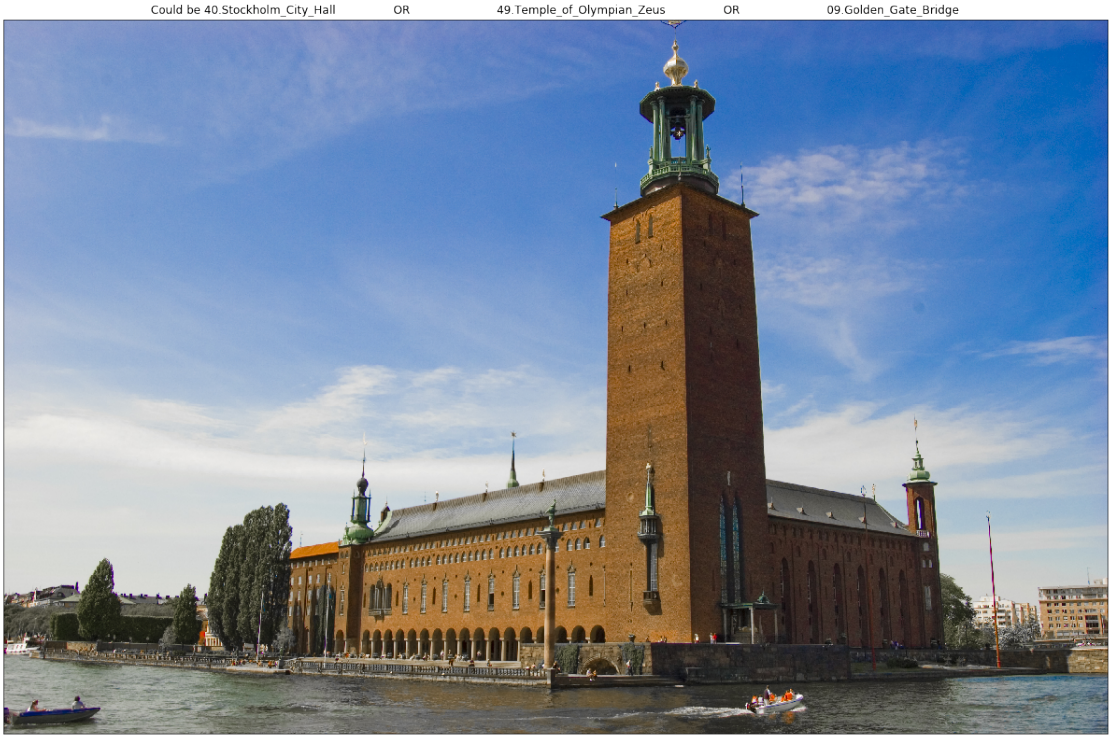
In summary, my takeaways are: 1. Procure a more representative dataset if possible 2. Apply more data augmentation and transformation techniques on the data 3. Use a deeper network.

In [26]:
```
## TODO: Execute the `suggest_locations` function on
## at least 4 images on your computer.
## Feel free to use as many code cells as needed.
# print(train_data.classes)
suggest_locations('france-eiffel-tower-paris.jpg')
suggest_locations('Wien.jpg')
suggest_locations('145.jpg')
suggest_locations("Townhallstockholm.jpg")
```



Could be 19.Vienna_City_Hall OR 40.Stockholm_City_Hall OR 39.Gateway_of_India

Could be 49.Temple_of_Olympian_Zeus OR 19.Vienna_City_Hall OR 40.Stockholm_City_Hall


Could be 40.Stockholm_City_Hall OR 49.Temple_of_Olympian_Zeus OR 09.Golden_Gate_Bridge

In [ ]: