landmark

August 5, 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

# I'm using the code from Lesson 1

import os #cv2
from PIL import Image #PIL
import torch
import numpy as np

import torchvision
from torchvision import datasets, models
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
import matplotlib.pyplot as plt # to visualize data

# image = Image.open("/data/landmark_images/test/37.Atomium/5ecb74282baee5aa.jpg")
```

```python
# width, height = image.size
# print(width, height)



# 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

# convert data to torch.FloatTensor
# transform following
train_transform = transforms.Compose([transforms.RandomRotation(30),
                                      transforms.Resize(256),
                                      transforms.CenterCrop(224),
                                      transforms.RandomHorizontalFlip(),
                                      transforms.ToTensor(),
                                      transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.22
                                     ])
test_valid_transform = transforms.Compose([transforms.Resize(256),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.22
                                     ])

# choose the training and test datasets
train_data = datasets.ImageFolder("/data/landmark_images/train", transform = train_trans
test_data = datasets.ImageFolder("/data/landmark_images/test", transform = test_valid_tr
valid_data = datasets.ImageFolder("/data/landmark_images/train", transform = test_valid_

# 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]
# print("Count Train pics: ", len(train_idx))
# print("Count Validation pics: ", len(valid_idx))
# print("Count Test pics: ", len(test_data))


# 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=train_sampler, num_workers=num_workers)
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size,
    sampler=valid_sampler, num_workers=num_workers)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
    num_workers=num_workers)

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

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: My code rotates the image randomly by 30 degrees, then resizes the image into a size of (256,256), and then doing a centered cropping into a size of (224, 224), and then finally randomly flipping the image horizontally. I also normalized the images. I mainly followed the network from Lesson 1 as my guide. I picked 224x224 as my input tensor size, this is because later on in Step 2 I will be using a VGG16 model, which has 224x224 as input tensor size. I decided to crop my image to a 224x224 to save on time on latter parts.

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
        std = torch.tensor([0.229, 0.224, 0.225])
        mean = torch.tensor([0.485, 0.456, 0.406])


        ## 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`)

        # I'm using the sample code from MNIST notebook

        # obtain one batch of training images
        classes = train_data.classes
        print("Length of Classes: ", len(classes))

        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=(50, 25))
for idx in np.arange(5): # generate 5 images
    ax = fig.add_subplot(1, 5, idx+1, xticks=[], yticks=[])
    # un-normalize (using example in Lesson 1)
    plt.imshow(np.transpose(images[idx] * std[:, None, None] + mean[:, None, None], (1,
    # print out the correct label for each image
    # .item() gets the value contained in a Tensor
    ax.set_title(classes[labels[idx]])
```

```
Length of Classes:  50
```

1.1.3 Initialize use_cuda variable

```
In [3]: # useful variable that tells us whether we should use the GPU
        use_cuda = torch.cuda.is_available()
```

1.1.4 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [4]: ## TODO: select loss function

        import torch.nn as nn
        import torch.nn.functional as F
        import torch.optim as optim

        # following example in Lecture, using Cross Entropy Loss and Adam Optimizer

        criterion_scratch = nn.CrossEntropyLoss()

        def get_optimizer_scratch(model):
            ## TODO: select and return an optimizer
            # optimizer = optim.Adam(model.parameters(), lr = 0.01)
            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 [5]: import torch.nn as nn

        # define the CNN architecture
        class Net(nn.Module):
            ## TODO: choose an architecture, and complete the class
            # I'm using the example from lecture (CIFAR-10)
            def __init__(self):
                super(Net, self).__init__()

                ## Define layers of a CNN
                # convolutional layer (sees 32x32x3 image tensor)
                self.conv1 = nn.Conv2d(3, 16, 3, padding = 1)

                # convolutional layer (sees 16x16x16 tensor)
                self.conv2 = nn.Conv2d(16, 32, 3, padding=1)

                # convolutional layer (sees 8x8x32 tensor)
                self.conv3 = nn.Conv2d(32, 64, 3, padding=1)

                # conv layer
                # self.conv4 = nn.Conv2d(64, 128, 3, padding = 1)

                # max pooling layer
                self.pool = nn.MaxPool2d(2, 2)

                # linear layer (64 * 32 * 32 -> 500), fully connected layer
                # self.fc1 = nn.Linear(128 * 14 * 14, 256)
                self.fc1 = nn.Linear(64*28*28, 256)

                # linear layer (500 -> 50)
                self.fc2 = nn.Linear(256, 50) # We know len(classes) = 50

                # dropout layer (p=0.25)
                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)



            def forward(self, x):
```

```python
        ## Define forward behavior

        # add sequence of convolutional and max pooling layers
        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.conv4(x)))
        x = self.pool(F.relu(self.conv3(x)))

        # print(x.shape)

        # flatten image input
        # x = x.view(-1, 128 * 14 * 14)
        x = x.view(-1, 64*28*28)

        #print(x.shape)
        # add dropout later
        x = self.dropout(x)

        # add first hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
        x = self.dropout(x)

        # add 2nd hidden layer
        x = self.fc2(x)
        # no need to add dropout or relu coz its output layer
        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))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=50176, 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)
)
```

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

Answer:

I'm using the network that was used in lecture (Lesson 1, CIFAR-10 CNN Notebook). In the lecture, we know that a simple ConvNet for CIFAR-10 could have the architechture (INPUT - CONV - RELU - POOL - FC). I tried using this Network to see if this would map my data good enough, as our project is similar to the CIFAR-10 (except with 50 labels instead of 10). I used 3 Conv layers, as I want to capture more details with each CONV layers I have. I also added Batch Normalization layers.

1. Firstly INPUT (224x224x3) holds the raw pixel values of the image (width,height,RGB)
2. Apply first CONV layer (in order to capture shallow details), then Apply Batch Norm
3. Apply ReLU to ensure our output are positive
4. Apply maxpool to reduce the size of the image (helps with extracting sharp and smooth textures to reduce variance and computations)
5. Repeat step 2 ,3, and 4 exactly 2 times (we are not applying it to the last CONV layer)
6. Add a dropout layer to prevent overfitting of the data
7. Use a fully connected layer, followed by ReLU
8. A second Fully Connected layer is used, generating an output of 50 (for each image, since we have 50 labels)

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 [6]:
```python
from tqdm import tqdm
#from torch.optim.lr_scheduler import StepLR

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

    #scheduler = StepLR(optimizer, step_size = 20, gamma = 0.1)

    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

    # again, we're trying out the CIFAR-10 code from lecture
    # clear the gradients of all optimized variables
    optimizer.zero_grad()
    # forward pass: compute predicted outputs by passing inputs to the model
    output = model(data)
    # calculate the batch loss
    loss = criterion(output, target)
    # backward pass: compute gradient of the loss with respect to model paramete
    loss.backward()
    # perform a single optimization step (parameter update)
    optimizer.step()
    # update training loss
    # unlike the CIFAR-10 example, I'm following hint given above
    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
    # forward pass: compute predicted outputs by passing inputs to the model
    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
```

```
            # 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(), save_path)
                valid_loss_min = valid_loss



        return model
```

1.1.7 (IMPLEMENTATION) Experiment with the Weight Initialization

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

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

```
In [7]: def custom_weight_init(m):
        ## TODO: implement a weight initialization strategy
        # Using Lesson 5 Weight Initialization Notebook example
        # general rule for weight initialization is to start weights in the range of [-y,y]
        # y = 1/ sqrt(n), n = number of inputs
        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
                      criterion_scratch, use_cuda, 'ignore.pt')
```

```
Epoch: 1         Training Loss: 3.868338        Validation Loss: 3.774506
Validation loss decreased (inf --> 3.774506).  Saving model ...
Epoch: 2         Training Loss: 3.663238        Validation Loss: 3.590045
Validation loss decreased (3.774506 --> 3.590045).  Saving model ...
Epoch: 3         Training Loss: 3.492836        Validation Loss: 3.520766
Validation loss decreased (3.590045 --> 3.520766).  Saving model ...
Epoch: 4         Training Loss: 3.356481        Validation Loss: 3.295196
Validation loss decreased (3.520766 --> 3.295196).  Saving model ...
Epoch: 5         Training Loss: 3.200615        Validation Loss: 3.236304
Validation loss decreased (3.295196 --> 3.236304).  Saving model ...
Epoch: 6         Training Loss: 3.106119        Validation Loss: 3.065451
Validation loss decreased (3.236304 --> 3.065451).  Saving model ...
Epoch: 7         Training Loss: 2.989924        Validation Loss: 3.073928
Epoch: 8         Training Loss: 2.933932        Validation Loss: 3.007524
Validation loss decreased (3.065451 --> 3.007524).  Saving model ...
Epoch: 9         Training Loss: 2.859475        Validation Loss: 3.034158
Epoch: 10         Training Loss: 2.791857        Validation Loss: 2.997532
Validation loss decreased (3.007524 --> 2.997532).  Saving model ...
Epoch: 11         Training Loss: 2.714964        Validation Loss: 2.998771
Epoch: 12         Training Loss: 2.668300        Validation Loss: 2.889353
Validation loss decreased (2.997532 --> 2.889353).  Saving model ...
Epoch: 13         Training Loss: 2.600997        Validation Loss: 2.850174
Validation loss decreased (2.889353 --> 2.850174).  Saving model ...
Epoch: 14         Training Loss: 2.545321        Validation Loss: 2.838924
Validation loss decreased (2.850174 --> 2.838924).  Saving model ...
Epoch: 15         Training Loss: 2.469181        Validation Loss: 2.807817
Validation loss decreased (2.838924 --> 2.807817).  Saving model ...
Epoch: 16         Training Loss: 2.420879        Validation Loss: 2.858339
Epoch: 17         Training Loss: 2.356749        Validation Loss: 2.812252
Epoch: 18         Training Loss: 2.285262        Validation Loss: 2.807979
Epoch: 19         Training Loss: 2.243528        Validation Loss: 2.754631
Validation loss decreased (2.807817 --> 2.754631).  Saving model ...
Epoch: 20         Training Loss: 2.168799        Validation Loss: 2.824948
```

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 = 10

#-#-# 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: 3.883210         Validation Loss: 3.817197
Validation loss decreased (inf --> 3.817197).  Saving model ...
Epoch: 2         Training Loss: 3.702794         Validation Loss: 3.595353
Validation loss decreased (3.817197 --> 3.595353).  Saving model ...
Epoch: 3         Training Loss: 3.514483         Validation Loss: 3.423400
Validation loss decreased (3.595353 --> 3.423400).  Saving model ...
Epoch: 4         Training Loss: 3.385166         Validation Loss: 3.367055
Validation loss decreased (3.423400 --> 3.367055).  Saving model ...
Epoch: 5         Training Loss: 3.278983         Validation Loss: 3.261766
Validation loss decreased (3.367055 --> 3.261766).  Saving model ...
Epoch: 6         Training Loss: 3.164944         Validation Loss: 3.160344
Validation loss decreased (3.261766 --> 3.160344).  Saving model ...
Epoch: 7         Training Loss: 3.053925         Validation Loss: 3.168199
Epoch: 8         Training Loss: 2.974912         Validation Loss: 3.063055
Validation loss decreased (3.160344 --> 3.063055).  Saving model ...
Epoch: 9         Training Loss: 2.905925         Validation Loss: 3.030259
Validation loss decreased (3.063055 --> 3.030259).  Saving model ...
Epoch: 10         Training Loss: 2.820557         Validation Loss: 2.978847
Validation loss decreased (3.030259 --> 2.978847).  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 [9]: 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: 2.812334


Test Accuracy: 28% (362/1250)
```

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

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

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


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

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 [12]: ## TODO: Specify model architecture
         # following lecture example, we will use VGG16

         model_transfer = models.vgg16(pretrained = True)

         print(model_transfer.classifier[6].in_features)
         print(model_transfer.classifier[6].out_features)

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

         # replace final layer with a new one
         n_inputs = model_transfer.classifier[6].in_features

         # add last linear layer that maps n_inputs -> 50  classes
         # new layers will automatically have required_grad = True
         last_layer = nn.Linear(n_inputs, 50)

         model_transfer.classifier[6] = last_layer # replace model's last layer with our new lay
```

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

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 [00:04<00:00, 111937314.14it/s]
```

```
4096
1000
```

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:
Before, we used VGG16 in Lecture to help us classify Flower images. I think that classifying Landmark images is a similar task, and therefore I used a VGG16. Not only that, the model was already pretrained on numerous images, so all I had to do was freeze the feature layers in order to use pretrained parameters. After that, I replace the last FC layer with our desired count of labels (50).

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 [13]:
```
# TODO: train the model and save the best model parameters at filepath 'model_transfer.
# copy pasting from the From Scratch implementation

#train(num_epochs, loaders_scratch, model_scratch, get_optimizer_scratch(model_scratch)
#tried 10 epoch, validation loss stagnate after epoch 3. so I'm changing epoch to 4.

train(3, loaders_transfer, model_transfer, get_optimizer_transfer(model_transfer), crit


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

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

```
Epoch: 1         Training Loss: 2.230566         Validation Loss: 1.391592
Validation loss decreased (inf --> 1.391592).  Saving model ...
Epoch: 2         Training Loss: 1.293656         Validation Loss: 1.163378
Validation loss decreased (1.391592 --> 1.163378).  Saving model ...
```

```
Epoch: 3        Training Loss: 1.059555         Validation Loss: 1.077736
Validation loss decreased (1.163378 --> 1.077736).  Saving model ...
```

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

```
Test Loss: 0.954673


Test Accuracy: 74% (926/1250)
```

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 [26]: import cv2
         from PIL import Image

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

         def predict_landmarks(img_path, k):
             ## TODO: return the names of the top k landmarks predicted by the transfer learned
             image = Image.open(img_path)
             transform = transforms.Compose([transforms.Resize(256),
                                             transforms.CenterCrop(224),
                                             transforms.ToTensor(),
                                             transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.2
                                            ])
             #transform the image
```

```python
    trans_img = transform(image)

    # https://medium.com/@josh_2774/deep-learning-with-pytorch-9574e74d17ad
    trans_img.unsqueeze_(0)

    #move pic to GPU
    if use_cuda:
        trans_img = trans_img.cuda()

    # load model
    model_transfer.load_state_dict(torch.load('model_transfer.pt'))
    model_transfer.eval()

    #put image through the network
    pred_tensor = model_transfer(trans_img)

    # https://medium.com/@josh_2774/deep-learning-with-pytorch-9574e74d17ad
    # pred_tensor = pred_tensor.unsqueeze(0)


    # https://pytorch.org/docs/stable/generated/torch.topk.html
    top_3_values, top_3_indices = torch.topk(pred_tensor, k = 3)

    top_probs = top_3_values.cpu().detach().numpy().tolist()[0]
    top_labs = top_3_indices.cpu().detach().numpy().tolist()[0]

    # now convert stuff into labels
    landmark_names = []
    for i in top_labs:
        landmark_names.append(classes[i])

    return landmark_names
```

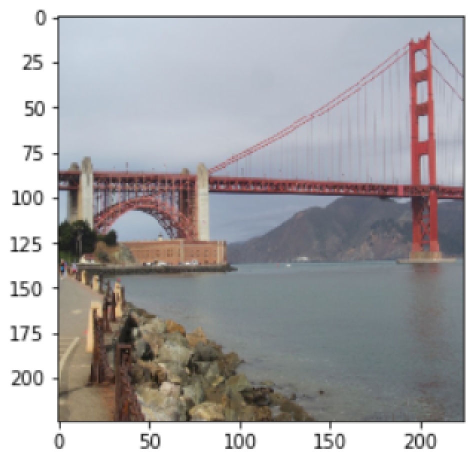
```python
# test on a sample image
predict_landmarks('images/test/09.Golden_Gate_Bridge/190f3bae17c32c37.jpg', 5)
```

Out[26]: ['09.Golden_Gate_Bridge', '38.Forth_Bridge', '30.Brooklyn_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 [27]:
```python
def suggest_locations(img_path):
    # get landmark predictions
    predicted_landmarks = predict_landmarks(img_path, 3)

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

    preds = predict_landmarks(img_path, 3)

    pred1 = preds[0].split(".")[1].replace("_"," ")
    pred2 = preds[1].split(".")[1].replace("_"," ")
    pred3 = preds[2].split(".")[1].replace("_"," ")
    print("Top predictions: " + pred1 + ", " + pred2 + ", " + pred3)



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


```
Top predictions: Golden Gate Bridge, Forth Bridge, Brooklyn Bridge
```

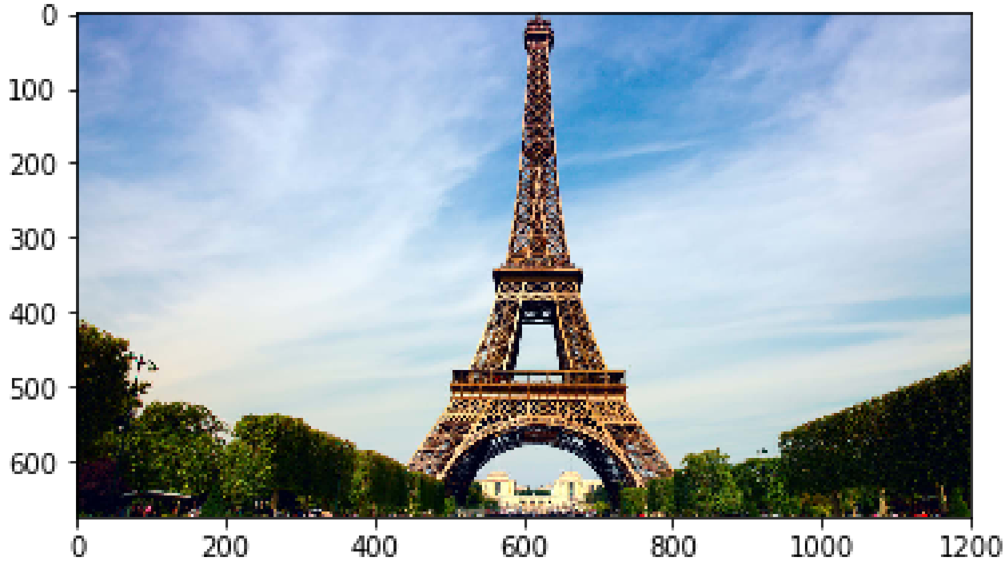
1.1.17 (IMPLEMENTATION) Test Your Algorithm

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.

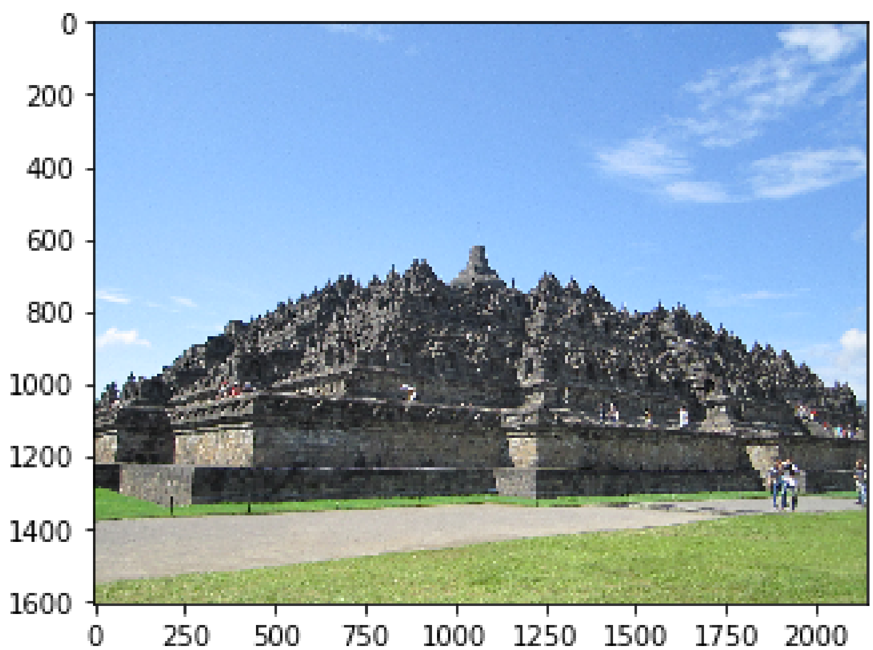
Question 4: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement) 1. Increasing the amount of images we have for training 2. Introduce more augmentation to our data. Possibly doing non centered crops, changing hue and color, etc 3. Try out transfer learning with other networks

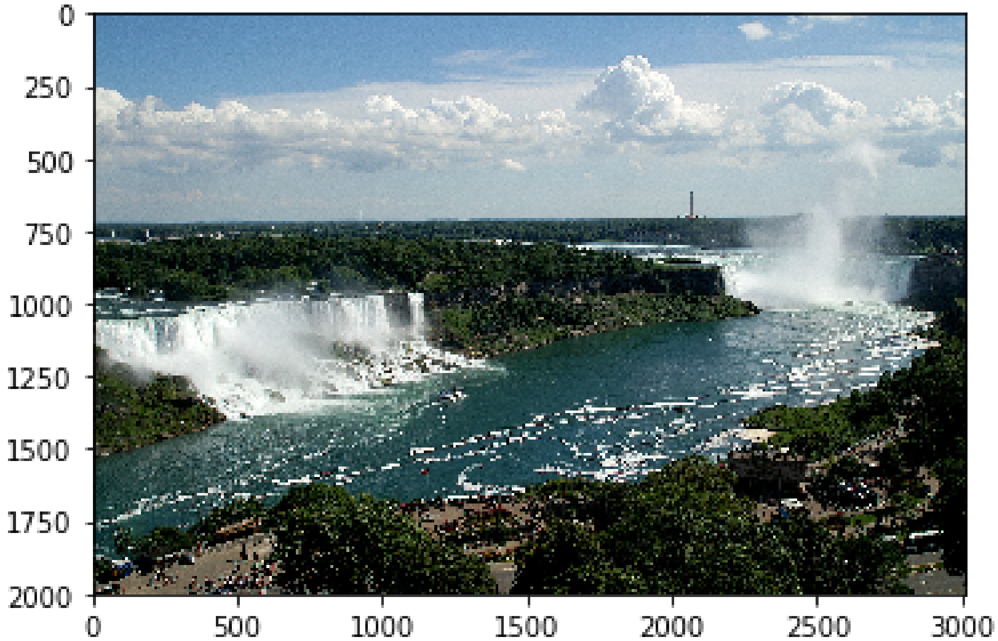
```
In [29]: ## TODO: Execute the `suggest_locations` function on
         ## at least 4 images on your computer.
         ## Feel free to use as many code cells as needed.
         for path in os.listdir('mine'):
             suggest_locations(os.path.join('mine', path))
```



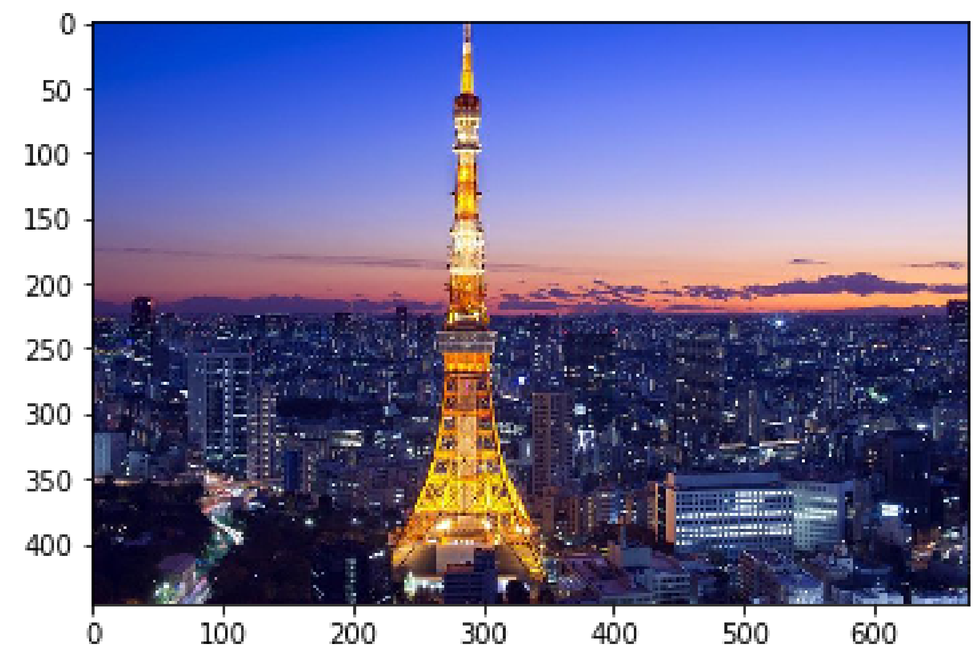
Top predictions: Eiffel Tower, Prague Astronomical Clock, Terminal Tower



Top predictions: Edinburgh Castle, Vienna City Hall, Whitby Abbey



Top predictions: Niagara Falls, Gullfoss Falls, Yellowstone National Park



```
Top predictions: Eiffel Tower, Terminal Tower, Vienna City Hall
```

```
In [ ]:
```