landmark

June 19, 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 [2]:
```
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes

#import needed libraries and modules
from torchvision import datasets, transforms, models
import torch
import numpy as np
from torch.utils.data.sampler import SubsetRandomSampler
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.nn.functional as F

#use transforms to process the data pipeline
transform_train = transforms.Compose([transforms.RandomRotation(30),
                transforms.RandomResizedCrop(224),
                transforms.RandomHorizontalFlip(),
                transforms.ToTensor(),
```

```
                transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                     std=[0.229, 0.224, 0.225])])
transform_test = transforms.Compose([transforms.Resize(255),
                transforms.CenterCrop(224),
                transforms.ToTensor(),
                transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                     std=[0.229, 0.224, 0.225])])
#specify path of data
root_dir = '/data/landmark_images'
train_data = datasets.ImageFolder(root_dir+'/train', transform=transform_train)
test_data = datasets.ImageFolder(root_dir+'/test', transform=transform_test)

# percentage of training set to use as validation
valid_size = 0.2

# obtain training indices that will be used for validation as used in toturials.
#*p.s: I tried different method using torch.utils.data.random_split in my device but I d
#and thats why I just used a SubsetRandomSampler
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)

# define trainloaders
trainloader = torch.utils.data.DataLoader(train_data, batch_size=32, sampler=train_sampl
validloader = torch.utils.data.DataLoader(train_data, batch_size=32, sampler=valid_sampl
testloader = torch.utils.data.DataLoader(test_data, batch_size=32)

loaders_scratch = {'train': trainloader, 'valid': validloader, 'test': testloader}
```

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: Besically, I defined different transformers for data processing and thten I split the data and made the data loaders that will be used in training/validation/testing. 1- my code resize all of them to different reasons, first to make their sizes equal, second to train faster (since image sizes are small), and third to use the same code again later in transfer-learning since Pytorch models have minimum size of (224) and the defined normalization values above. 2- yes, for train and validation sets I decided to do random rotation/resize/flip to generalize the model more. As for test dataset, I have done only resize and center crop since they are besically for testing not training. In both cases, I added normalization with the value recommended by Pytorch (for imagenet dataset I think?).

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 [3]: %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`)

        #based on https://matplotlib.org/stable/gallery/images_contours_and_fields/image_demo.ht

        #make an iterator and get a batch of data
        data_iter = iter(loaders_scratch['train'])
        images, labels = next(data_iter)

        #define image processing function
        def fixed_image(image):
            image = image.numpy().transpose((1, 2, 0))
            mean = np.array([0.485, 0.456, 0.406])
            std = np.array([0.229, 0.224, 0.225])
            image = std * image + mean
            image = np.clip(image, 0, 1)
            return image

        #plot the images in the figure with their titles
        fig, axs = plt.subplots(1, 5, figsize=(20, 5))
        for ax, i in zip(axs, [i for i in range(5)]):
            ax.imshow(fixed_image(images[i]))
            ax.set_title(train_data.classes[labels[i]][3:].replace('_', ' '))

        plt.show()
```

1.1.3 Initialize use_cuda variable

```
In [4]: # useful variable that tells us whether we should use the GPU
        use_cuda = torch.cuda.is_available()
        use_cuda

Out[4]: 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 [4]: ## TODO: select loss function
        criterion_scratch = torch.nn.CrossEntropyLoss()

        def get_optimizer_scratch(model):
            return torch.optim.Adam(model.parameters())
```

1.1.5 (IMPLEMENTATION) Model Architecture

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

```
In [5]: # 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
                #sees 224x224x3
                self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
                #sees 112x112x32
                self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
                #sees 56x56x64
                self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
                #sees 28x28x128
                self.conv4 = nn.Conv2d(128, 256, 3, padding=1)
                #sees 14x14x256
                self.conv5 = nn.Conv2d(256, 512, 3, padding=1)
                self.pooling = nn.MaxPool2d(2,2)
                self.fc1 = nn.Linear(7*7*512, 1024)
                self.fc2 = nn.Linear(1024, 50)
                self.Drop = nn.Dropout(0.25)

            def forward(self, x):
                ## Define forward behavior
                x = self.pooling(F.relu(self.conv1(x)))
                x = self.pooling(F.relu(self.conv2(x)))
```

```
            x = self.pooling(F.relu(self.conv3(x)))
            x = self.pooling(F.relu(self.conv4(x)))
            x = self.pooling(F.relu(self.conv5(x)))
            x = x.view(-1, 7*7*512)
            x = self.Drop(x)
            x = F.relu(self.fc1(x))
            x = self.Drop(x)
            x = self.fc2(x)
            return x

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

    # instantiate the CNN
    model_scratch = Net()

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

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

Answer: interesting question since it took me sometime to define a "good" architecture :). As for the network above, besically it is a simple CNN that uses conv layers with pooling to make useful feature-maps for the network and pooling is used to reduce the dimentionality of the feature-maps so can the model train faster and finally there is a fully connected layer used for prediction. A relu function is used as activation function since that it is a good activation function that does the job, computionally not expensive (the model trains faster), and it has good reputition in CNN models. Also, there is a drop-out function to reduce the possibility of overfitting. Also, since I used CrossEntropyLoss, I dont have to define softmax function at the end. CrossEntropyLoss is being used because I'm working on multi-class classification problem. As for my optimizer choose, I decided to go with Adam since it is a very nice momentum optimizer and does the job pretty well in CNNs from my previous knowledge on the topic. In my "initial model", I tried to start with low number of feature-maps such as Conv2d(3, 4) but it confused me since the model never learns I though I had problem in my training code. Then, I decided to look into state-of-art models in classification problems since the project is about that (here https://paperswithcode.com/sota/image-classification-on-imagenet), then I've opened some papers like Inception or Resnet and seen that most of them start with Conv2d(3, 32) or Conv2d(3, 64), I'm not sure why but it looks like the model trains well with good such numbers.

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

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

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

        ## TODO: find the loss and update the model parameters accordingly
        #clear gradient
        optimizer.zero_grad()
        #forward pass
        output = model(data)
        #batch loss
        loss = criterion(output, target)
        #backward pass
        loss.backward()
        #update weights
        optimizer.step()
        ## record the average training loss, using something like
        ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data.item() - tr
        train_loss +=((1 / (batch_idx + 1)) * (loss.data.item() - train_loss))



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

        ## TODO: update average validation loss

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

```
            #batch loss
            loss = criterion(output, target)
            #val loss
            valid_loss +=((1 / (batch_idx + 1)) * (loss.data.item() - valid_loss))




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

        ## TODO: if the validation loss has decreased, save the model at the filepath st
        if valid_loss<=valid_loss_min:
            print('Validation loss decreased from {:.3f} to {:.3f}.  Saving...'.format(v
            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
            #from Udacity weight init lectures
            #I wanted to use normal dist because it yields the best result :D
            classname = m.__class__.__name__
            # iterate over linear layers
            if (classname.find('Linear') != -1):
                # get inputs in that layer
                n = m.in_features
                #y will be used to define range of values
                y = (1.0/np.sqrt(n))
                #fill data with normal dist
                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(15, loaders_scratch, model_scratch, get_optimizer_scratch(model_sc
                              criterion_scratch, use_cuda, 'ignore.pt')
```

```
Epoch: 1         Training Loss: 3.883174         Validation Loss: 3.817678
Validation loss decreased from inf to 3.818.  Saving...
Epoch: 2         Training Loss: 3.792716         Validation Loss: 3.802810
Validation loss decreased from 3.818 to 3.803.  Saving...
Epoch: 3         Training Loss: 3.728430         Validation Loss: 3.696549
Validation loss decreased from 3.803 to 3.697.  Saving...
Epoch: 4         Training Loss: 3.643141         Validation Loss: 3.571628
Validation loss decreased from 3.697 to 3.572.  Saving...
Epoch: 5         Training Loss: 3.530134         Validation Loss: 3.537065
Validation loss decreased from 3.572 to 3.537.  Saving...
Epoch: 6         Training Loss: 3.466899         Validation Loss: 3.419910
Validation loss decreased from 3.537 to 3.420.  Saving...
Epoch: 7         Training Loss: 3.365408         Validation Loss: 3.322824
Validation loss decreased from 3.420 to 3.323.  Saving...
Epoch: 8         Training Loss: 3.269600         Validation Loss: 3.269552
Validation loss decreased from 3.323 to 3.270.  Saving...
Epoch: 9         Training Loss: 3.177768         Validation Loss: 3.149099
Validation loss decreased from 3.270 to 3.149.  Saving...
Epoch: 10         Training Loss: 3.076113         Validation Loss: 3.157675
Epoch: 11         Training Loss: 3.013116         Validation Loss: 3.047112
Validation loss decreased from 3.149 to 3.047.  Saving...
Epoch: 12         Training Loss: 2.941075         Validation Loss: 3.048334
Epoch: 13         Training Loss: 2.868963         Validation Loss: 2.946725
Validation loss decreased from 3.047 to 2.947.  Saving...
Epoch: 14         Training Loss: 2.797405         Validation Loss: 2.946489
Validation loss decreased from 2.947 to 2.946.  Saving...
Epoch: 15         Training Loss: 2.731584         Validation Loss: 2.857864
Validation loss decreased from 2.946 to 2.858.  Saving...
```

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 = 15

        #-#-# 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: 3.896508         Validation Loss: 3.840067
Validation loss decreased from inf to 3.840.  Saving...
Epoch: 2         Training Loss: 3.781824         Validation Loss: 3.721947
Validation loss decreased from 3.840 to 3.722.  Saving...
Epoch: 3         Training Loss: 3.670157         Validation Loss: 3.612874
Validation loss decreased from 3.722 to 3.613.  Saving...
Epoch: 4         Training Loss: 3.584044         Validation Loss: 3.579917
Validation loss decreased from 3.613 to 3.580.  Saving...
Epoch: 5         Training Loss: 3.500155         Validation Loss: 3.457118
Validation loss decreased from 3.580 to 3.457.  Saving...
Epoch: 6         Training Loss: 3.412047         Validation Loss: 3.409316
Validation loss decreased from 3.457 to 3.409.  Saving...
Epoch: 7         Training Loss: 3.325495         Validation Loss: 3.364747
Validation loss decreased from 3.409 to 3.365.  Saving...
Epoch: 8         Training Loss: 3.205619         Validation Loss: 3.146691
Validation loss decreased from 3.365 to 3.147.  Saving...
Epoch: 9         Training Loss: 3.101047         Validation Loss: 3.073852
Validation loss decreased from 3.147 to 3.074.  Saving...
Epoch: 10         Training Loss: 3.050317         Validation Loss: 3.110825
Epoch: 11         Training Loss: 2.978258         Validation Loss: 3.003998
Validation loss decreased from 3.074 to 3.004.  Saving...
Epoch: 12         Training Loss: 2.916844         Validation Loss: 2.987180
Validation loss decreased from 3.004 to 2.987.  Saving...
Epoch: 13         Training Loss: 2.874545         Validation Loss: 2.936947
Validation loss decreased from 2.987 to 2.937.  Saving...
Epoch: 14         Training Loss: 2.835343         Validation Loss: 2.893687
Validation loss decreased from 2.937 to 2.894.  Saving...
Epoch: 15         Training Loss: 2.712956         Validation Loss: 2.923934
```

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: 2.570421


Test Accuracy: 34% (433/1250)
```

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': trainloader, 'valid': validloader, 'test': testloader}
```

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 = torch.nn.CrossEntropyLoss()

         def get_optimizer_transfer(model):
             ## TODO: select and return optimizer
             return torch.optim.Adam(model.fc.parameters())
```

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 [5]: ## TODO: Specify model architecture
        model_transfer = models.resnet50(pretrained=True)
        n_inputs = model_transfer.fc.in_features
        last_layer = nn.Linear(n_inputs, 50)
        model_transfer.fc = last_layer
        print(model_transfer)
        #-#-# Do NOT modify the code below this line. #-#-#

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

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:02<00:00, 40911009.12it/s]


ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
```

```
(maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
(layer1): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
)
(layer2): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
```

```
  (1): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (3): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
)
(layer3): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (3): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (4): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (5): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
)
(layer4): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=2048, 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: hmm, since the problem I'm working on is for image classification, I besically checked https://pytorch.org/vision/stable/models.html and https://paperswithcode.com/sota/image-classification-on-imagenet to find a good state-of-art model that can work on my problem. I tried ResNet-152 and ResNet-101 but got out-of-memory error, and then I just decreased the batch-size and used ResNet-50 to make it work. As for optimizer and loss choose, it is for the same reason as I said in scratch model. Furthermore, I have changed the last layer in the model because in my problem I'm classifing 50 classes not 1000. Finally, in this transfer-learning process, I decided to freeze the whole model and just train the last fc layer. Actually, I wanted to try this (https://imgur.com/a/XLqknmU) as we have learned in the lectures because I think the ImageNet data set does not contain places (Resnet trained on objects?). But, I didnt do it since I feel Resnet above is very complex and I'm not sure which layer to remove... if I have had faster parallel GPUs and more time in production problem I might try different combinations of removing last layers before fc layer and compare the result.

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.
         model_transfer = train(15, loaders_transfer, model_transfer, get_optimizer_transfer(mod
                                criterion_transfer, use_cuda, 'model_transfer.pt')
```

```
Epoch: 1         Training Loss: 2.773623         Validation Loss: 1.970645
Validation loss decreased from inf to 1.971.  Saving...
Epoch: 2         Training Loss: 1.690221         Validation Loss: 1.637305
Validation loss decreased from 1.971 to 1.637.  Saving...
Epoch: 3         Training Loss: 1.466101         Validation Loss: 1.493998
Validation loss decreased from 1.637 to 1.494.  Saving...
Epoch: 4         Training Loss: 1.329941         Validation Loss: 1.386191
Validation loss decreased from 1.494 to 1.386.  Saving...
Epoch: 5         Training Loss: 1.228864         Validation Loss: 1.382530
Validation loss decreased from 1.386 to 1.383.  Saving...
Epoch: 6         Training Loss: 1.178012         Validation Loss: 1.405610
Epoch: 7         Training Loss: 1.110163         Validation Loss: 1.302858
Validation loss decreased from 1.383 to 1.303.  Saving...
Epoch: 8         Training Loss: 1.092655         Validation Loss: 1.321890
Epoch: 9         Training Loss: 1.009292         Validation Loss: 1.358791
Epoch: 10         Training Loss: 1.043428         Validation Loss: 1.371527
Epoch: 11         Training Loss: 1.008477         Validation Loss: 1.341259
Epoch: 12         Training Loss: 0.954402         Validation Loss: 1.338015
Epoch: 13         Training Loss: 0.969632         Validation Loss: 1.309250
Epoch: 14         Training Loss: 0.968984         Validation Loss: 1.275512
Validation loss decreased from 1.303 to 1.276.  Saving...
Epoch: 15         Training Loss: 0.915982         Validation Loss: 1.310834
```

```
In [6]: #-#-# 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'))
```

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

```
Test Loss: 0.894907


Test Accuracy: 78% (977/1250)
```

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 [7]: import cv2
        from PIL import Image

        def predict_landmarks(img_path, k):
            ## TODO: return the names of the top k landmarks predicted by the transfer learned C
            image = Image.open(img_path)
            image = transform_test(image)
            image = image.unsqueeze_(0)
            image = image.cuda()
            with torch.no_grad():
                output = model_transfer(image)
            soft_max_output = F.softmax(output.squeeze(), dim=0)
            top = soft_max_output.topk(k)
            print(top)
            top_numpy = top[1].to('cpu').numpy()
            classes = np.array(train_data.classes)
            result = classes.take(top_numpy).tolist()
            for i in range(len(result)):
                result[i]=result[i][3:].replace("_", " ")
            return result

        ## the class names can be accessed at the `classes` attribute
        ## of your dataset object (e.g., `train_dataset.classes`)
        model_transfer.eval()
        # test on a sample image
        image = 'images/test/09.Golden_Gate_Bridge/190f3bae17c32c37.jpg'
        k = 7
        predict_landmarks(image, k)
```

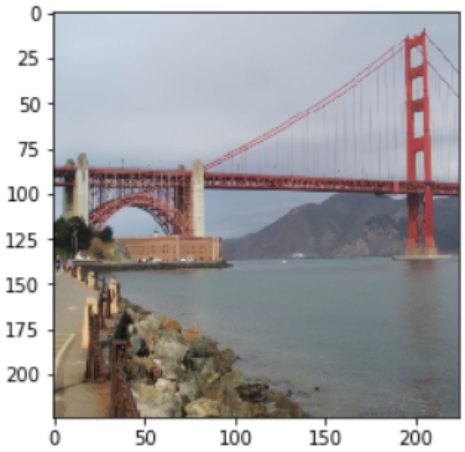
```
(tensor([ 0.7182,  0.1835,  0.0521,  0.0208,  0.0157,  0.0030,  0.0022], device='cuda:0'), tenso
```

```
Out[7]: ['Golden Gate Bridge',
         'Forth Bridge',
         'Brooklyn Bridge',
         'Sydney Opera House',
         'Sydney Harbour Bridge',
         'Niagara Falls',
         'Pont du Gard']
```

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!



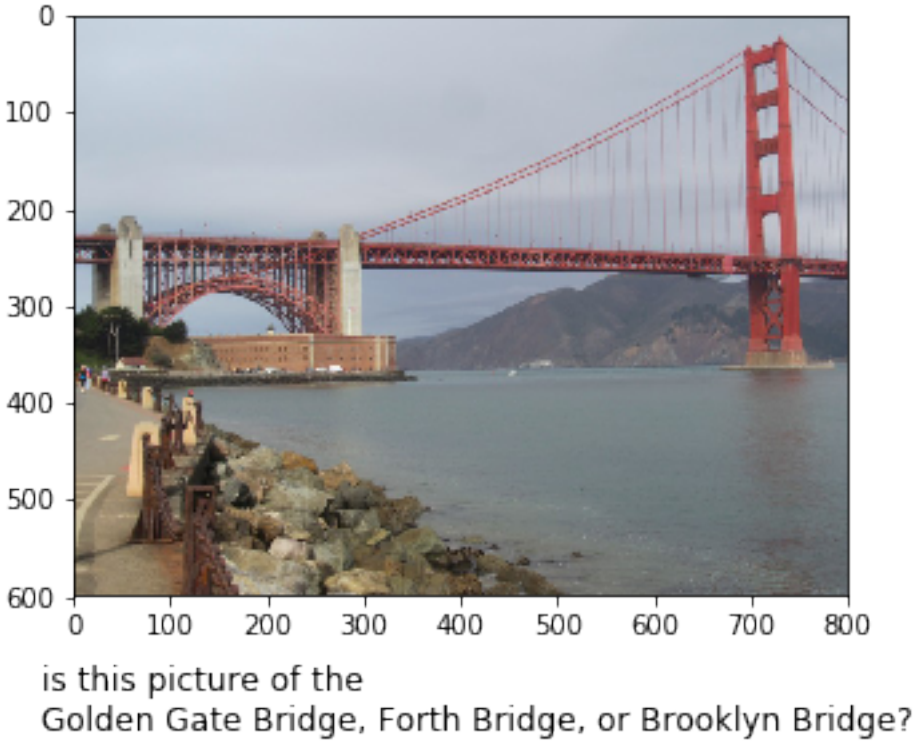

```
Is this picture of the
Golden Gate Bridge, Brooklyn Bridge, or Sydney Harbour Bridge?
```

```
In [8]: 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)
            text ='is this picture of the \n'
            for landmark in predicted_landmarks:
                if(landmark == predicted_landmarks[-1]):
                    text+="or "+landmark+"?"
                else:
                    text+=landmark+", "
            plt.figtext(0.15, -0.05, text, fontsize=12)
            plt.show()
```

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

```
(tensor([ 0.7182,  0.1835,  0.0521], device='cuda:0'), tensor([  9,  38,  30], device='cuda:0'))
```



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.

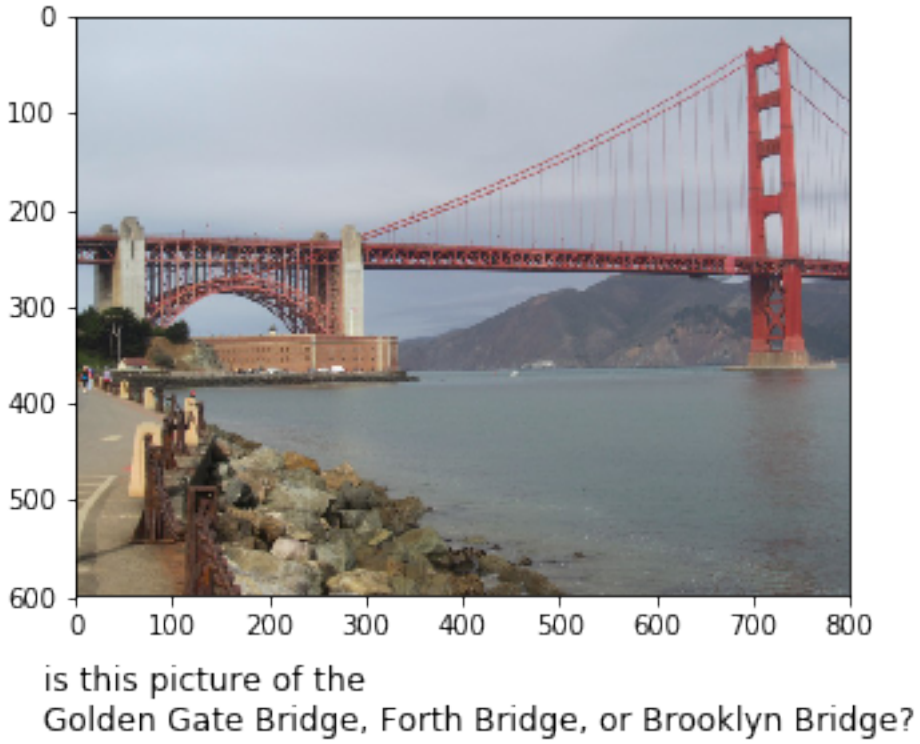
Question 4: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: the output is better than I expected, it seems like my model works pretty well on available test set :). As for improvements in the model, I think... 1- it could have been trained for longer time (more epochs) since I just trained it for like 15 epochs and there is more potential. 2- we could have improved the data quality (collected more data or more variant of the same class) that has been taken in different times/angles/etc... 3- I think using more complex model like (inception_v3/deeper resnet) would improve the result of the model. We could also use ensambling by averaging the result of different models, which I think could improve the result. 4- the model is limited only to the classes in the data, it would be nice improvement to make it work on more landmarks in the world!

```
In [9]: # In my computer, I tried some different images in the "test" folder and wrote the answe
        #The below images are different from the ones I tested in my computer,
```

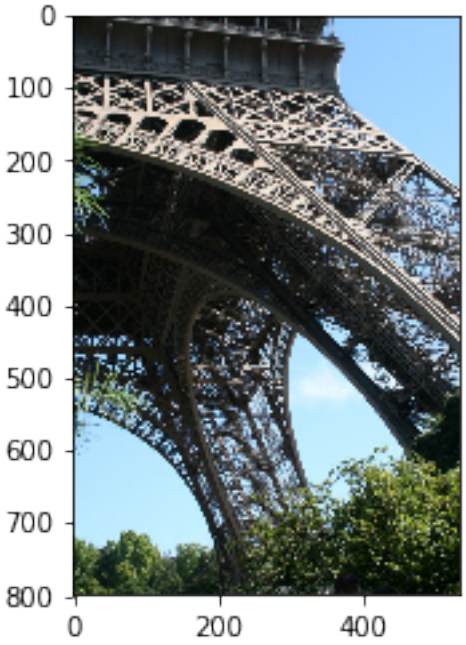
```
        #but I just used them here since they are available in the VM workspace for the answer.
        suggest_locations('images/test/09.Golden_Gate_Bridge/190f3bae17c32c37.jpg')
```

```
(tensor([ 0.7182,  0.1835,  0.0521], device='cuda:0'), tensor([  9,  38,  30], device='cuda:0'))
```



In [10]:
```
suggest_locations('images/test/16.Eiffel_Tower/3828627c8730f160.jpg')
```

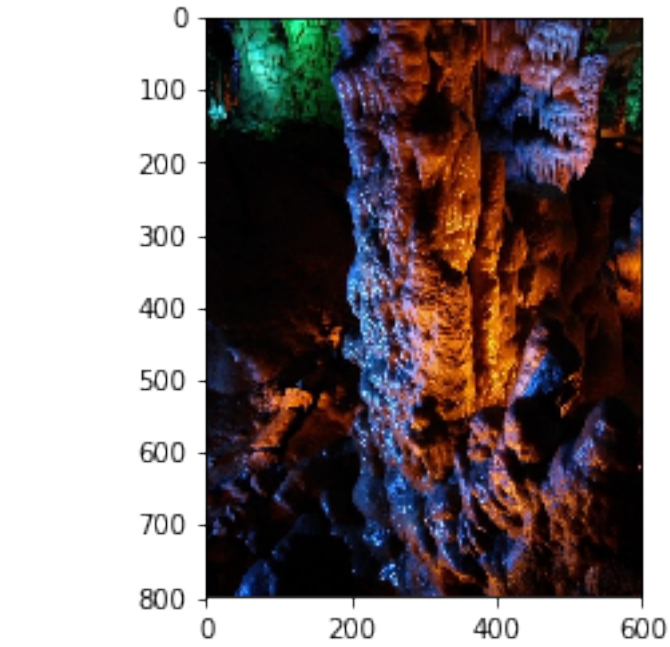
```
(tensor([ 0.5091,  0.4854,  0.0029], device='cuda:0'), tensor([ 28,  16,  26], device='cuda:0'))
```



is this picture of the
Sydney Harbour Bridge, Eiffel Tower, or Pont du Gard?

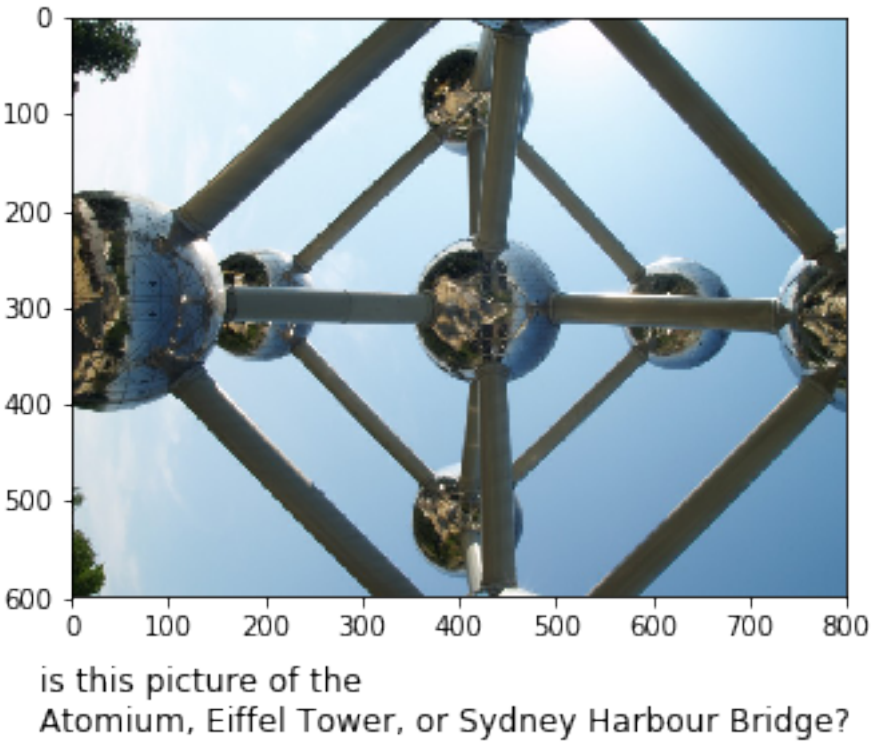
```
In [11]: suggest_locations('images/test/24.Soreq_Cave/18dbbad48a83a742.jpg')

(tensor([ 0.6837,  0.1248,  0.0770], device='cuda:0'), tensor([ 24,  18,   8], device='cuda:0'))
```



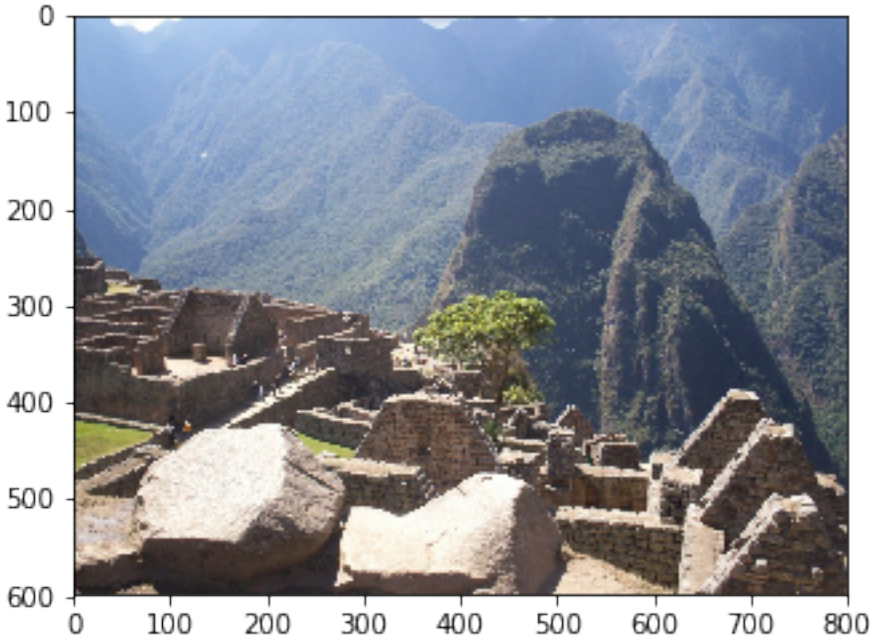

```
In [12]: suggest_locations('images/test/37.Atomium/5ecb74282baee5aa.jpg')

(tensor([ 0.9998,  0.0001,  0.0000], device='cuda:0'), tensor([ 37,  16,  28], device='cuda:0'))
```




```
In [13]: suggest_locations('images/test/41.Machu_Picchu/4336abf3179202f2.jpg')
```

```
(tensor([ 0.6696,  0.3049,  0.0187], device='cuda:0'), tensor([ 41,  46,  32], device='cuda:0'))
```



is this picture of the
Machu Picchu, Great Wall of China, or Hanging Temple?

In [ ]: