landmark

December 1, 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 [4]:
```
import torch
from torchvision import datasets, transforms
import numpy as np
import splitfolders
```

In [3]:
```
! pip install split-folders
```

```
Collecting split-folders
  Downloading https://files.pythonhosted.org/packages/b8/5f/3c2b2f7ea5e047c8cdc3bb00ae582c5438fc
Installing collected packages: split-folders
Successfully installed split-folders-0.4.3
```

In [5]:
```
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes

train_transform = transforms.Compose([
    transforms.RandomResizedCrop(224),
```

```
    transforms.RandomHorizontalFlip(p=0.3),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

test_val_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

splitfolders.ratio("/data/landmark_images/train", output="train_val", seed=1337, ratio=(
train_data = datasets.ImageFolder('./train_val/train', transform=train_transform)
valid_data = datasets.ImageFolder('./train_val/val', transform=test_val_transform)
test_data = datasets.ImageFolder('./landmark_images/test', transform=test_val_transform)


n_classes = len(train_data.classes)
classes = [class_.split(".")[1].replace("_", " ") for class_ in train_data.classes]

batch_size = 32

train_loader = torch.utils.data.DataLoader(train_data, batch_size, shuffle=True)
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size, shuffle=True)


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

```
Copying files: 4996 files [00:12, 400.44 files/s]
```

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: A1: in the training by random cropping and for testing by normal resize then center cropping. the choosen input size was 224 because it is most use in this task for classification and like vgg16 with imagenet.

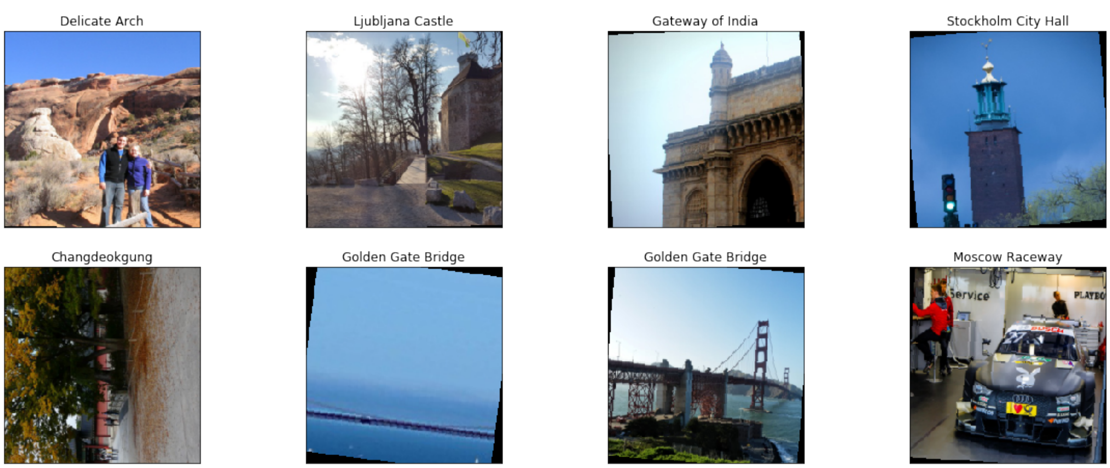
A2: yes, only for training: random horizontal flip with probabilty of 30% and random rotation for 10 degrees.

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 [24]: 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`)

         import random

         def unnormlize(img, s, m):
             return img * s[:, None, None] + m[:, None, None]

         fig = plt.figure(figsize=(20,2*8))
         for idx in range(8):
             ax = fig.add_subplot(4, 4, idx+1, xticks=[], yticks=[], )
             rand_img = random.randint(0, len(train_data))

             img = unnormlize(train_data[rand_img][0], torch.Tensor([0.229, 0.224, 0.225]), torc
             plt.imshow(np.transpose(img.numpy(), (1, 2, 0)))
             class_name = classes[train_data[rand_img][1]]
             ax.set_title(class_name)
```



1.1.3 Initialize use_cuda variable

```
In [6]: # 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 [7]:
```
import torch.nn as nn
import torch.optim as optim
```

In [8]:
```
## TODO: select loss function
criterion_scratch = nn.CrossEntropyLoss()

def get_optimizer_scratch(model):
    ## TODO: select and return an optimizer
    optimizer = optim.SGD(model.parameters(), lr=0.02)
    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 [9]:
```
import torch.nn.functional as F
```

In [10]:
```
# define the CNN architecture
class Net(nn.Module):
    ## TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()

        ## Define layers of a CNN

        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(28 * 28 * 64, 256)
        self.fc2 = nn.Linear(256, n_classes)
        self.dropout = nn.Dropout(0.3)



    def forward(self, x):
        ## Define forward behavior

        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 28 * 28 * 64)
        x = self.dropout(x)
```

```
            x = F.relu(self.fc1(x))
            x = self.dropout(x)
            x = self.fc2(x)


            return x

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

    # instantiate the CNN
    model_scratch = Net()

    # 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: I returned to the first and second lessons in CNN chapter and review it and getting more understanding how CNN works and finally I searched for articles about CNN in image classification and got this article: https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-ii-hyper-parameter-42efca01e5d7 and I follow that guide without batch normalization.

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

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

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

```
        ## TODO: find the loss and update the model parameters accordingly
        ## record the average training loss, using something like
        ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data.item() - t

        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data.item() - trai




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

        ## TODO: update average validation loss
        output = model(data)
        loss = criterion(output, target)
        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 s
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased.  Saving model ...')
        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 [16]: def custom_weight_init(m):
             ## TODO: implement a weight initialization strategy
             if isinstance(m, nn.Conv2d):
                 n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                 m.weight.data.normal_(0, np.sqrt(2. / n))
                 if m.bias is not None:
                     m.bias.data.zero_()
             elif isinstance(m, nn.Linear):
                 n = m.in_features
                 y = 1.0/np.sqrt(n)
                 m.weight.data.normal_(0, y)
                 m.bias.data.zero_()


         #-#-# 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_s
                               criterion_scratch, use_cuda, 'ignore.pt')
```

```
Epoch: 1        Training Loss: 3.890368        Validation Loss: 3.791037
Validation loss decreased.  Saving model ...
Epoch: 2        Training Loss: 3.766836        Validation Loss: 3.700719
Validation loss decreased.  Saving model ...
Epoch: 3        Training Loss: 3.676332        Validation Loss: 3.560715
Validation loss decreased.  Saving model ...
Epoch: 4        Training Loss: 3.594732        Validation Loss: 3.484435
Validation loss decreased.  Saving model ...
Epoch: 5        Training Loss: 3.520298        Validation Loss: 3.455525
Validation loss decreased.  Saving model ...
Epoch: 6        Training Loss: 3.462143        Validation Loss: 3.417543
Validation loss decreased.  Saving model ...
Epoch: 7        Training Loss: 3.394466        Validation Loss: 3.348084
Validation loss decreased.  Saving model ...
Epoch: 8        Training Loss: 3.363833        Validation Loss: 3.293525
Validation loss decreased.  Saving model ...
Epoch: 9        Training Loss: 3.321818        Validation Loss: 3.280778
```

```
Validation loss decreased.  Saving model ...
Epoch: 10         Training Loss: 3.279316         Validation Loss: 3.233558
Validation loss decreased.  Saving model ...
Epoch: 11         Training Loss: 3.250193         Validation Loss: 3.308343
Epoch: 12         Training Loss: 3.212363         Validation Loss: 3.133894
Validation loss decreased.  Saving model ...
Epoch: 13         Training Loss: 3.202331         Validation Loss: 3.152840
Epoch: 14         Training Loss: 3.161348         Validation Loss: 3.230696
Epoch: 15         Training Loss: 3.149108         Validation Loss: 3.178567
Epoch: 16         Training Loss: 3.108228         Validation Loss: 3.145370
Epoch: 17         Training Loss: 3.073907         Validation Loss: 3.012860
Validation loss decreased.  Saving model ...
Epoch: 18         Training Loss: 3.027281         Validation Loss: 3.130322
Epoch: 19         Training Loss: 2.994791         Validation Loss: 3.000735
Validation loss decreased.  Saving model ...
Epoch: 20         Training Loss: 2.946450         Validation Loss: 3.068411
```

1.1.8 (IMPLEMENTATION) Train and Validate the Model

Run the next code cell to train your model.

In [19]:
```python
## TODO: you may change the number of epochs if you'd like,
## but changing it is not required
num_epochs = 30

#-#-# 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: 3.906564         Validation Loss: 3.872541
Validation loss decreased.  Saving model ...
Epoch: 2         Training Loss: 3.840918         Validation Loss: 3.763925
Validation loss decreased.  Saving model ...
Epoch: 3         Training Loss: 3.755440         Validation Loss: 3.672594
Validation loss decreased.  Saving model ...
Epoch: 4         Training Loss: 3.691984         Validation Loss: 3.613972
```

```
Validation loss decreased.  Saving model ...
Epoch: 5        Training Loss: 3.626201        Validation Loss: 3.558258
Validation loss decreased.  Saving model ...
Epoch: 6        Training Loss: 3.557535        Validation Loss: 3.598085
Epoch: 7        Training Loss: 3.516012        Validation Loss: 3.465958
Validation loss decreased.  Saving model ...
Epoch: 8        Training Loss: 3.434233        Validation Loss: 3.342068
Validation loss decreased.  Saving model ...
Epoch: 9        Training Loss: 3.407582        Validation Loss: 3.337504
Validation loss decreased.  Saving model ...
Epoch: 10        Training Loss: 3.355127        Validation Loss: 3.430574
Epoch: 11        Training Loss: 3.290676        Validation Loss: 3.225321
Validation loss decreased.  Saving model ...
Epoch: 12        Training Loss: 3.265308        Validation Loss: 3.270146
Epoch: 13        Training Loss: 3.253887        Validation Loss: 3.177811
Validation loss decreased.  Saving model ...
Epoch: 14        Training Loss: 3.194455        Validation Loss: 3.262294
Epoch: 15        Training Loss: 3.207428        Validation Loss: 3.149806
Validation loss decreased.  Saving model ...
Epoch: 16        Training Loss: 3.165356        Validation Loss: 3.122023
Validation loss decreased.  Saving model ...
Epoch: 17        Training Loss: 3.140834        Validation Loss: 3.167165
Epoch: 18        Training Loss: 3.086922        Validation Loss: 3.280803
Epoch: 19        Training Loss: 3.091530        Validation Loss: 3.175477
Epoch: 20        Training Loss: 3.037606        Validation Loss: 3.169945
Epoch: 21        Training Loss: 3.026002        Validation Loss: 3.001220
Validation loss decreased.  Saving model ...
Epoch: 22        Training Loss: 3.004746        Validation Loss: 3.422950
Epoch: 23        Training Loss: 3.005199        Validation Loss: 3.002218
Epoch: 24        Training Loss: 2.962083        Validation Loss: 3.014511
Epoch: 25        Training Loss: 2.956298        Validation Loss: 3.218424
Epoch: 26        Training Loss: 2.925776        Validation Loss: 3.132196
Epoch: 27        Training Loss: 2.885943        Validation Loss: 2.918003
Validation loss decreased.  Saving model ...
Epoch: 28        Training Loss: 2.868252        Validation Loss: 3.046173
Epoch: 29        Training Loss: 2.835162        Validation Loss: 2.943342
Epoch: 30        Training Loss: 2.841954        Validation Loss: 2.905801
Validation loss decreased.  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 [20]: 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: 2.829261


Test Accuracy: 28% (351/1250)
```

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

         loaders_transfer = loaders_scratch.copy()
```

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 [13]: ## 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=0.02)
             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 [14]: from torchvision import models

In [15]: ## TODO: Specify model architecture

         model_transfer = models.vgg16(pretrained=True)

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

         num_input = model_transfer.classifier[6].in_features
         model_transfer.classifier[6] = nn.Linear(num_input, n_classes)


         #-#-# 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 [00:05<00:00, 108677324.21it/s]
```

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: I use vgg16 because it had been trained with a large dataset and with little similar data. I replaced last fully connected layer with same input size but 50 output size to address our problem

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 [16]: # TODO: train the model and save the best model parameters at filepath 'model_transfer.
         train(15, loaders_transfer, model_transfer, get_optimizer_transfer(model_transfer), cri
               use_cuda, 'model_transfer.pt')



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

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

```
Epoch: 1         Training Loss: 2.554012        Validation Loss: 1.670675
Validation loss decreased.  Saving model ...
Epoch: 2         Training Loss: 1.698070        Validation Loss: 1.332399
Validation loss decreased.  Saving model ...
Epoch: 3         Training Loss: 1.511131        Validation Loss: 1.235777
Validation loss decreased.  Saving model ...
Epoch: 4         Training Loss: 1.370468        Validation Loss: 1.158766
Validation loss decreased.  Saving model ...
Epoch: 5         Training Loss: 1.285819        Validation Loss: 1.148329
Validation loss decreased.  Saving model ...
Epoch: 6         Training Loss: 1.214011        Validation Loss: 1.123043
Validation loss decreased.  Saving model ...
Epoch: 7         Training Loss: 1.145267        Validation Loss: 1.080372
Validation loss decreased.  Saving model ...
Epoch: 8         Training Loss: 1.126196        Validation Loss: 1.086837
Epoch: 9         Training Loss: 1.062500        Validation Loss: 1.066760
Validation loss decreased.  Saving model ...
Epoch: 10         Training Loss: 1.007532        Validation Loss: 1.027337
Validation loss decreased.  Saving model ...
```

```
Epoch: 11         Training Loss: 0.969979        Validation Loss: 1.021818
Validation loss decreased.  Saving model ...
Epoch: 12         Training Loss: 0.949582        Validation Loss: 1.019601
Validation loss decreased.  Saving model ...
Epoch: 13         Training Loss: 0.902821        Validation Loss: 1.015920
Validation loss decreased.  Saving model ...
Epoch: 14         Training Loss: 0.894579        Validation Loss: 1.080869
Epoch: 15         Training Loss: 0.823666        Validation Loss: 1.007445
Validation loss decreased.  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 [21]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.844298


Test Accuracy: 77% (967/1250)
```

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 [22]: import cv2
         from PIL import Image

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

         def predict_landmarks(img_path, k):
             ## TODO: return the names of the top k landmarks predicted by the transfer learned
```

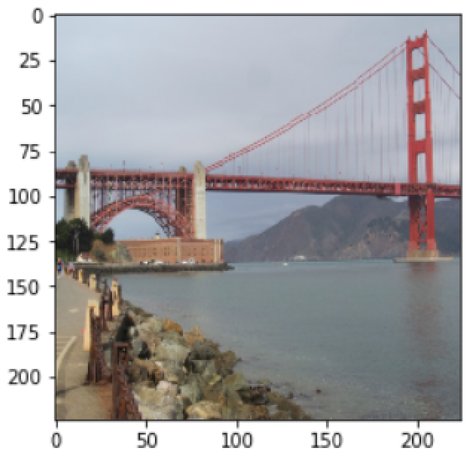
```
        img = Image.open(img_path).convert('RGB')

        transform = transforms.Compose([
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])

        img = transform(img)
        img.unsqueeze_(0)

        if use_cuda:
            img = img.cuda()

        model_transfer.eval()
        output = model_transfer(img)
        top_values, top_idx = output.topk(k)

        near_classes = [classes[i] for i in top_idx[0].tolist()]
        return near_classes


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

```
Out[22]: ['Golden Gate Bridge',
          'Forth Bridge',
          'Brooklyn Bridge',
          'Eiffel Tower',
          'Sydney Harbour 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 [25]: def suggest_locations(img_path):
             # get landmark predictions
             predicted_landmarks = predict_landmarks(img_path, 3)

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

             print(f"Actual Label: {img_path.split('/')[2][3:].replace('_',' ').split('.')[0]}")
             print(f"Predicted Label in order: Is this picture of the\n {predicted_landmarks[0]}



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


```
Actual Label: Golden Gate Bridge
Predicted Label in order: Is this picture of the
 Golden Gate Bridge, Forth Bridge, or Brooklyn Bridge?
```

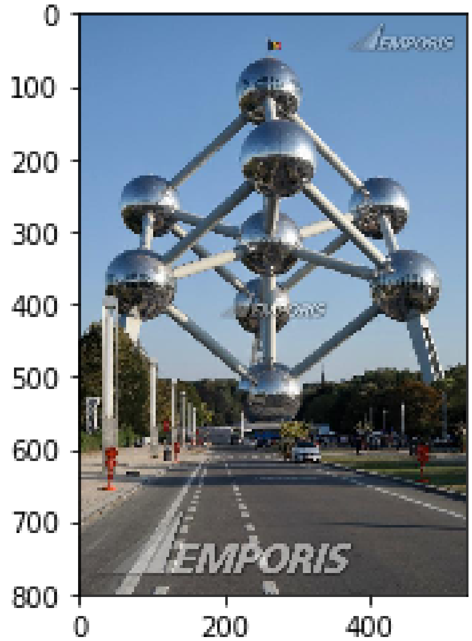
1.1.17 (IMPLEMENTATION) Test Your Algorithm

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.

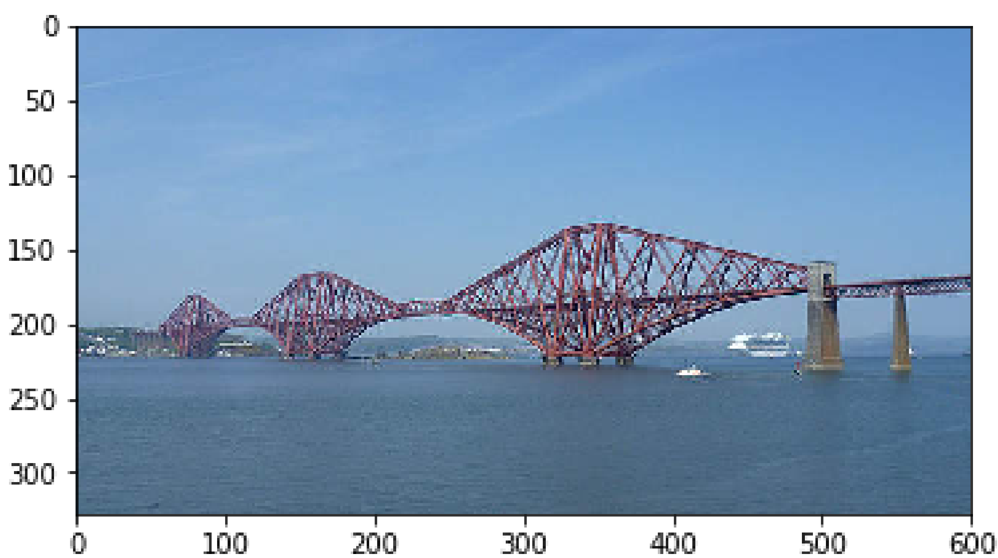
Answer: (Three possible points for improvement) 1 - train vgg16 with large dataset related to our problem 2 - clean the data 3 - tune the hyperparameters

```
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.

         import os

In [27]:
         suggest_locations('./test_algorithm/01.Atomium.jpg')
```



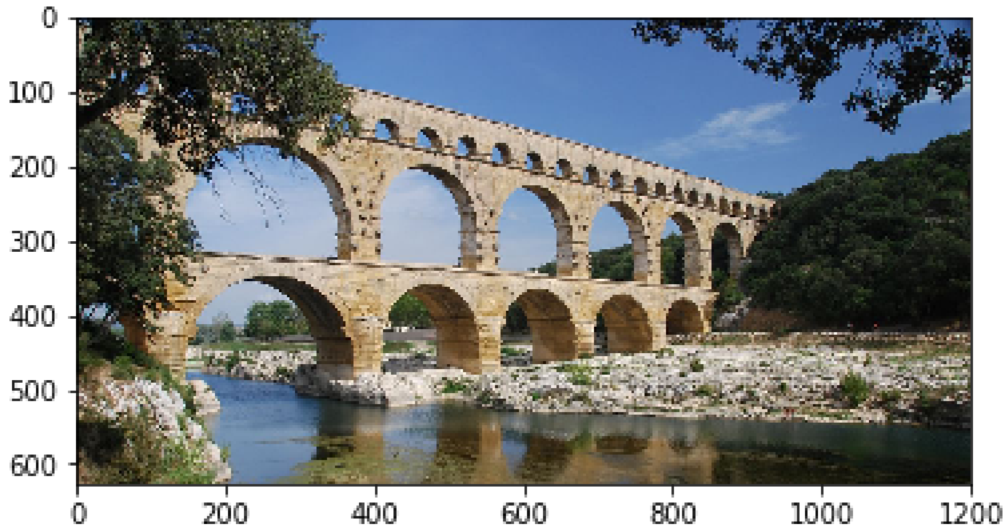

```
Actual Label: Atomium
Predicted Label in order: Is this picture of the
 Atomium, Sydney Harbour Bridge, or Monumento a la Revolucion?
```

```
In [29]: suggest_locations('./test_algorithm/02.Forth_Bridge.jpg')
```



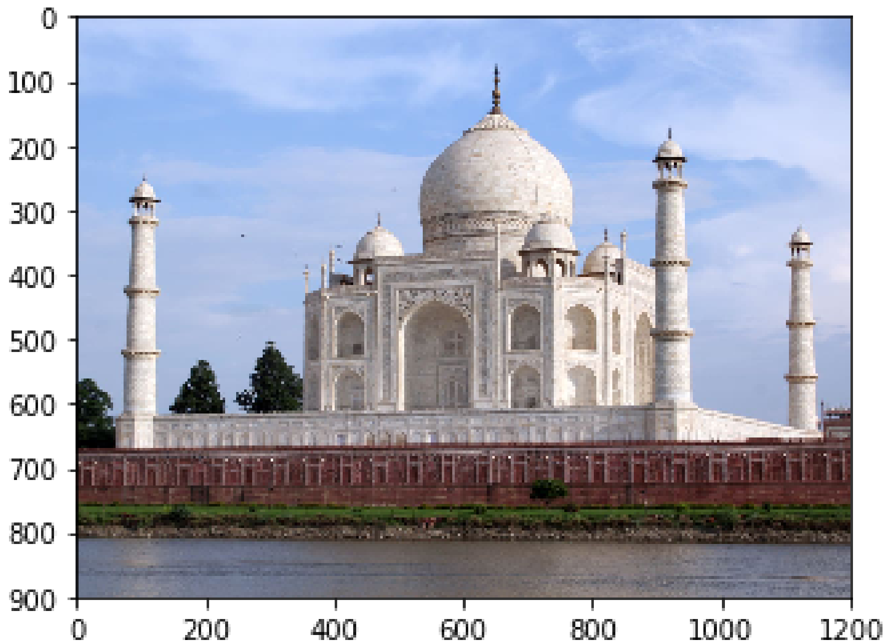
```
Actual Label: Forth Bridge
Predicted Label in order: Is this picture of the
 Forth Bridge, Sydney Harbour Bridge, or Eiffel Tower?
```

```
In [30]: suggest_locations('./test_algorithm/03.Pont_du_Gard.jpg')
```




```
Actual Label: Pont du Gard
Predicted Label in order: Is this picture of the
 Pont du Gard, Taj Mahal, or Ljubljana Castle?
```

```
In [31]: suggest_locations('./test_algorithm/04.Taj_Mahal.jpg')
```




```
Actual Label: Taj Mahal
Predicted Label in order: Is this picture of the
 Taj Mahal, Niagara Falls, or Stockholm City Hall?
```

```
In [ ]:
```