landmark

December 15, 2021

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for Landmark Classification

In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested. Sections that begin with **'(IMPLEMENTATION)'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section, and the specifics of the implementation are marked in the code block with a 'TODO' statement. Please be sure to read the instructions carefully!

> **Note**: Once you have completed all the code implementations, you need to finalize your work by exporting the Jupyter Notebook as an HTML document. Before exporting the notebook to HTML, all the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to **File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.

Step 0: Download Datasets and Install Python Modules

Note: if you are using the Udacity workspace, *YOU CAN SKIP THIS STEP*. The dataset can be found in the `/data` folder and all required Python modules have been installed in the workspace.

Download the landmark dataset. Unzip the folder and place it in this project's home directory, at the location `/landmark_images`.

Install the following Python modules: * cv2 * matplotlib * numpy * PIL * torch * torchvision
Step 1: Create a CNN to Classify Landmarks (from Scratch)

In this step, you will create a CNN that classifies landmarks. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 20%.

Although 20% may seem low at first glance, it seems more reasonable after realizing how difficult of a problem this is. Many times, an image that is taken at a landmark captures a fairly mundane image of an animal or plant, like in the following picture.

Just by looking at that image alone, would you have been able to guess that it was taken at the Haleakal National Park in Hawaii?

An accuracy of 20% is significantly better than random guessing, which would provide an accuracy of just 2%. In Step 2 of this notebook, you will have the opportunity to greatly improve accuracy by using transfer learning to create a CNN.

Remember that practice is far ahead of theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

1.1.1 (IMPLEMENTATION) Specify Data Loaders for the Landmark Dataset

Use the code cell below to create three separate data loaders: one for training data, one for validation data, and one for test data. Randomly split the images located at `landmark_images/train` to create the train and validation data loaders, and use the images located at `landmark_images/test` to create the test data loader.

Note: Remember that the dataset can be found at `/data/landmark_images/` in the workspace.

All three of your data loaders should be accessible via a dictionary named `loaders_scratch`. Your train data loader should be at `loaders_scratch['train']`, your validation data loader should be at `loaders_scratch['valid']`, and your test data loader should be at `loaders_scratch['test']`.

You may find this documentation on custom datasets to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of transforms!

In [1]:
```
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
import os
import numpy as np
import torch
from torchvision import datasets
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler



batch_size= 20 # how many samples the CNN sees and learn from at a time
valid_size = 0.2

# define training and test data directories
data_dir = '/data/landmark_images/'
train_dir = os.path.join(data_dir, 'train')
test_dir = os.path.join(data_dir, 'test')
```

```
data_transform = transforms.Compose([transforms.RandomResizedCrop(256),
                                     transforms.ToTensor(),
                                     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.

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

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

num_train = len(train_data)
indices = list(range(num_train)) # indices of the enire dataset
np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))  # take 20% of training set size
train_idx, valid_idx = indices[split:], indices[:split]

train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

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


# allow us to iterate data once batch at a time
loaders_scratch = {'train':train_loader ,'valid': valid_loader, 'test':test_loader }


#print(train_data.classes)
classes = [classes_name.split(".")[1] for classes_name in train_data.classes]
#print(classes[49])
```

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

Question 1: Describe your chosen procedure for preprocessing the data. - How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why? - Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

Answer: I randomly resized crop the images to 256x256 pixels for training and testing. My research led me to choose 256 pixels since it appears to be a fairly typical and suitable size for an image Then, the image is transformed to a tensor and normalized.

Did you decide to augment the dataset? No, because I got the needed accuracy without augmentaion. Also, It takes extra time.

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
        import random

        ## 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`)

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


        fig = plt.figure(figsize=(20,2*8))
        for index in range(12):
            ax = fig.add_subplot(4, 4, index+1, xticks=[], yticks=[])
            rand_img = random.randint(0, len(train_data))
            img = imshow(train_data[rand_img][0]) # unnormalize
            class_name = classes[train_data[rand_img][1]]
            ax.set_title(class_name)
```

1.1.3 Initialize use_cuda variable

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

1.1.4 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [3]: import torch.optim as optim
        import torch.nn as nn

        ## TODO: select loss function
        criterion_scratch = nn.CrossEntropyLoss()

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

1.1.5 (IMPLEMENTATION) Model Architecture

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

In [4]:
```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
class Net(nn.Module):
    ## TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        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(64 * 32 * 32 , 256)
        self.fc2 = nn.Linear(256, 50)
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.conv1(x))) # size 128
        x = self.pool(F.relu(self.conv2(x))) # size 64
        x = self.pool(F.relu(self.conv3(x))) # size 32
        x = x.view(-1, 64 * 32 * 32 )
        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 start by doing some research, and I found this helpful article https://towardsdatascience.com/a-guide-to-an-efficient-way-to-build-neural-network-architectures-part-ii-hyper-parameter-42efca01e5d7, it explains some general tips. In order to satisfy the task, I concentrated on designing a simple architecture as much as possible. I used 3 CNN with RELU activation function, and Max pooling using a 2x2 kernel between them to focus on the main target features via dividing the image by a factor of 2. In the fully connected layers, I used two linear layers and 0.3 dropouts to avoid overfitting.

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 [3]: 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: # load them in parallel
                        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
                    optimizer.zero_grad()
                    output = model(data)
                    loss = criterion(output, target)
                    loss.backward() # calculate gradient
                    optimizer.step() # update wieghts
                    train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data.item() - train

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

            ## 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
                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 [6]: def custom_weight_init(m):
            ## TODO: implement a weight initialization strategy

            classname = m.__class__.__name__
            # for the two Linear layers
            if classname.find('Linear') != -1:
                num_inputs = m.in_features
                y= 1.0/np.sqrt(num_inputs) # general rule
                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: 3.907488         Validation Loss: 3.899762
Validation loss decreased (inf --> 3.899762).  Saving model ...
Epoch: 2         Training Loss: 3.872485         Validation Loss: 3.840784
Validation loss decreased (3.899762 --> 3.840784).  Saving model ...
Epoch: 3         Training Loss: 3.811376         Validation Loss: 3.785992
Validation loss decreased (3.840784 --> 3.785992).  Saving model ...
Epoch: 4         Training Loss: 3.759716         Validation Loss: 3.764054
Validation loss decreased (3.785992 --> 3.764054).  Saving model ...
Epoch: 5         Training Loss: 3.709446         Validation Loss: 3.736488
Validation loss decreased (3.764054 --> 3.736488).  Saving model ...
Epoch: 6         Training Loss: 3.640216         Validation Loss: 3.646607
```

```
Validation loss decreased (3.736488 --> 3.646607).  Saving model ...



    ---------------------------------------------------------------------------

    KeyboardInterrupt                         Traceback (most recent call last)

    <ipython-input-6-a3055780c100> in <module>()
     14 model_scratch.apply(custom_weight_init)
     15 model_scratch = train(20, loaders_scratch, model_scratch, get_optimizer_scratch(mode
---> 16                       criterion_scratch, use_cuda, 'ignore.pt')


    <ipython-input-5-ef58a952dbdc> in train(n_epochs, loaders, model, optimizer, criterion,
     14         # set the module to training mode
     15         model.train()
---> 16         for batch_idx, (data, target) in enumerate(loaders['train']):
     17             # move to GPU
     18             if use_cuda: # load them in parallel


    /opt/conda/lib/python3.6/site-packages/torch/utils/data/dataloader.py in __next__(self)
    262         if self.num_workers == 0:  # same-process loading
    263             indices = next(self.sample_iter)  # may raise StopIteration
--> 264             batch = self.collate_fn([self.dataset[i] for i in indices])
    265             if self.pin_memory:
    266                 batch = pin_memory_batch(batch)


    /opt/conda/lib/python3.6/site-packages/torch/utils/data/dataloader.py in <listcomp>(.0)
    262         if self.num_workers == 0:  # same-process loading
    263             indices = next(self.sample_iter)  # may raise StopIteration
--> 264             batch = self.collate_fn([self.dataset[i] for i in indices])
    265             if self.pin_memory:
    266                 batch = pin_memory_batch(batch)


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/datasets/
     99         """
    100         path, target = self.samples[index]
--> 101         sample = self.loader(path)
    102         if self.transform is not None:
    103             sample = self.transform(sample)


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/datasets/
    145         return accimage_loader(path)
```

```
    146     else:
--> 147         return pil_loader(path)
    148 
    149 


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/datasets/
    128     with open(path, 'rb') as f:
    129         img = Image.open(f)
--> 130         return img.convert('RGB')
    131 
    132 


    /opt/conda/lib/python3.6/site-packages/PIL/Image.py in convert(self, mode, matrix, dithe
    890         """
    891 
--> 892         self.load()
    893 
    894         if not mode and self.mode == "P":


    /opt/conda/lib/python3.6/site-packages/PIL/ImageFile.py in load(self)
    233 
    234                             b = b + s
--> 235                             n, err_code = decoder.decode(b)
    236                             if n < 0:
    237                                 break


    KeyboardInterrupt:
```

1.1.8 (IMPLEMENTATION) Train and Validate the Model

Run the next code cell to train your model.

In [8]:
```
## TODO: you may change the number of epochs if you'd like,
## but changing it is not required
num_epochs = 100

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

# function to re-initialize a model with pytorch's default weight initialization
def default_weight_init(m):
    reset_parameters = getattr(m, 'reset_parameters', None)
    if callable(reset_parameters):
        m.reset_parameters()
```

```python
# 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: 3.909505         Validation Loss: 3.895505
Validation loss decreased (inf --> 3.895505).  Saving model ...
Epoch: 2        Training Loss: 3.868633         Validation Loss: 3.817610
Validation loss decreased (3.895505 --> 3.817610).  Saving model ...
Epoch: 3        Training Loss: 3.795987         Validation Loss: 3.747426
Validation loss decreased (3.817610 --> 3.747426).  Saving model ...
Epoch: 4        Training Loss: 3.752184         Validation Loss: 3.711107
Validation loss decreased (3.747426 --> 3.711107).  Saving model ...
Epoch: 5        Training Loss: 3.683661         Validation Loss: 3.602775
Validation loss decreased (3.711107 --> 3.602775).  Saving model ...
Epoch: 6        Training Loss: 3.633052         Validation Loss: 3.563989
Validation loss decreased (3.602775 --> 3.563989).  Saving model ...
Epoch: 7        Training Loss: 3.577597         Validation Loss: 3.525716
Validation loss decreased (3.563989 --> 3.525716).  Saving model ...
Epoch: 8        Training Loss: 3.532984         Validation Loss: 3.471752
Validation loss decreased (3.525716 --> 3.471752).  Saving model ...
Epoch: 9        Training Loss: 3.522616         Validation Loss: 3.500795
Epoch: 10        Training Loss: 3.483000         Validation Loss: 3.433374
Validation loss decreased (3.471752 --> 3.433374).  Saving model ...
Epoch: 11        Training Loss: 3.445436         Validation Loss: 3.410617
Validation loss decreased (3.433374 --> 3.410617).  Saving model ...
Epoch: 12        Training Loss: 3.414833         Validation Loss: 3.397876
Validation loss decreased (3.410617 --> 3.397876).  Saving model ...
Epoch: 13        Training Loss: 3.401638         Validation Loss: 3.381579
Validation loss decreased (3.397876 --> 3.381579).  Saving model ...
Epoch: 14        Training Loss: 3.362428         Validation Loss: 3.313298
Validation loss decreased (3.381579 --> 3.313298).  Saving model ...
Epoch: 15        Training Loss: 3.332376         Validation Loss: 3.299838
Validation loss decreased (3.313298 --> 3.299838).  Saving model ...
Epoch: 16        Training Loss: 3.295439         Validation Loss: 3.280938
Validation loss decreased (3.299838 --> 3.280938).  Saving model ...
Epoch: 17        Training Loss: 3.295731         Validation Loss: 3.255529
Validation loss decreased (3.280938 --> 3.255529).  Saving model ...
Epoch: 18        Training Loss: 3.263695         Validation Loss: 3.176454
Validation loss decreased (3.255529 --> 3.176454).  Saving model ...
Epoch: 19        Training Loss: 3.214501         Validation Loss: 3.369181
Epoch: 20        Training Loss: 3.182446         Validation Loss: 3.202065
Epoch: 21        Training Loss: 3.157708         Validation Loss: 3.153024
Validation loss decreased (3.176454 --> 3.153024).  Saving model ...
Epoch: 22        Training Loss: 3.116857         Validation Loss: 3.171619
```

```
Epoch: 23        Training Loss: 3.083092        Validation Loss: 3.157139
Epoch: 24        Training Loss: 3.078749        Validation Loss: 3.060971
Validation loss decreased (3.153024 --> 3.060971).  Saving model ...
Epoch: 25        Training Loss: 3.054673        Validation Loss: 2.998141
Validation loss decreased (3.060971 --> 2.998141).  Saving model ...
Epoch: 26        Training Loss: 3.026882        Validation Loss: 3.061116
Epoch: 27        Training Loss: 2.976199        Validation Loss: 3.053724
Epoch: 28        Training Loss: 2.944025        Validation Loss: 3.022949
Epoch: 29        Training Loss: 2.946410        Validation Loss: 2.967309
Validation loss decreased (2.998141 --> 2.967309).  Saving model ...
Epoch: 30        Training Loss: 2.933103        Validation Loss: 3.035238
Epoch: 31        Training Loss: 2.889539        Validation Loss: 3.019790
Epoch: 32        Training Loss: 2.863601        Validation Loss: 2.863687
Validation loss decreased (2.967309 --> 2.863687).  Saving model ...
Epoch: 33        Training Loss: 2.824453        Validation Loss: 3.046966
Epoch: 34        Training Loss: 2.831050        Validation Loss: 2.940507
Epoch: 35        Training Loss: 2.804018        Validation Loss: 2.886432
Epoch: 36        Training Loss: 2.780912        Validation Loss: 2.974003
Epoch: 37        Training Loss: 2.738892        Validation Loss: 2.862343
Validation loss decreased (2.863687 --> 2.862343).  Saving model ...
Epoch: 38        Training Loss: 2.712267        Validation Loss: 2.868053
Epoch: 39        Training Loss: 2.726476        Validation Loss: 2.848291
Validation loss decreased (2.862343 --> 2.848291).  Saving model ...
Epoch: 40        Training Loss: 2.671203        Validation Loss: 2.871970
Epoch: 41        Training Loss: 2.652282        Validation Loss: 2.824695
Validation loss decreased (2.848291 --> 2.824695).  Saving model ...
Epoch: 42        Training Loss: 2.653142        Validation Loss: 2.804786
Validation loss decreased (2.824695 --> 2.804786).  Saving model ...
Epoch: 43        Training Loss: 2.621647        Validation Loss: 2.899562
Epoch: 44        Training Loss: 2.605801        Validation Loss: 2.859537
Epoch: 45        Training Loss: 2.587340        Validation Loss: 2.820272
Epoch: 46        Training Loss: 2.597420        Validation Loss: 2.741832
Validation loss decreased (2.804786 --> 2.741832).  Saving model ...
Epoch: 47        Training Loss: 2.559875        Validation Loss: 2.727482
Validation loss decreased (2.741832 --> 2.727482).  Saving model ...
Epoch: 48        Training Loss: 2.523615        Validation Loss: 2.805245
Epoch: 49        Training Loss: 2.503593        Validation Loss: 2.802748
Epoch: 50        Training Loss: 2.506984        Validation Loss: 2.863890
Epoch: 51        Training Loss: 2.474814        Validation Loss: 2.789474
Epoch: 52        Training Loss: 2.476566        Validation Loss: 2.697211
Validation loss decreased (2.727482 --> 2.697211).  Saving model ...
Epoch: 53        Training Loss: 2.452291        Validation Loss: 2.784248
Epoch: 54        Training Loss: 2.403779        Validation Loss: 2.712603
Epoch: 55        Training Loss: 2.409839        Validation Loss: 2.811819
Epoch: 56        Training Loss: 2.368833        Validation Loss: 2.708915
Epoch: 57        Training Loss: 2.354452        Validation Loss: 2.697857
Epoch: 58        Training Loss: 2.361934        Validation Loss: 2.715126
Epoch: 59        Training Loss: 2.334514        Validation Loss: 2.626867
```

```
Validation loss decreased (2.697211 --> 2.626867).  Saving model ...
Epoch: 60        Training Loss: 2.330992        Validation Loss: 2.805471
Epoch: 61        Training Loss: 2.290376        Validation Loss: 2.746309
Epoch: 62        Training Loss: 2.282030        Validation Loss: 2.777620
Epoch: 63        Training Loss: 2.263074        Validation Loss: 2.674854
Epoch: 64        Training Loss: 2.247675        Validation Loss: 2.673689
Epoch: 65        Training Loss: 2.252676        Validation Loss: 2.668916
Epoch: 66        Training Loss: 2.216201        Validation Loss: 2.728748
Epoch: 67        Training Loss: 2.182983        Validation Loss: 2.616902
Validation loss decreased (2.626867 --> 2.616902).  Saving model ...
Epoch: 68        Training Loss: 2.155512        Validation Loss: 2.699193
Epoch: 69        Training Loss: 2.141415        Validation Loss: 2.718259
Epoch: 70        Training Loss: 2.183408        Validation Loss: 2.649160
Epoch: 71        Training Loss: 2.116867        Validation Loss: 2.606816
Validation loss decreased (2.616902 --> 2.606816).  Saving model ...
Epoch: 72        Training Loss: 2.143877        Validation Loss: 2.642533
Epoch: 73        Training Loss: 2.089066        Validation Loss: 2.638836
Epoch: 74        Training Loss: 2.120855        Validation Loss: 2.635272
Epoch: 75        Training Loss: 2.054554        Validation Loss: 2.718017
Epoch: 76        Training Loss: 2.068479        Validation Loss: 2.686291
Epoch: 77        Training Loss: 2.063130        Validation Loss: 2.641442
Epoch: 78        Training Loss: 2.046640        Validation Loss: 2.640541
Epoch: 79        Training Loss: 2.019077        Validation Loss: 2.720374
Epoch: 80        Training Loss: 1.998615        Validation Loss: 2.647174
Epoch: 81        Training Loss: 1.994461        Validation Loss: 2.619648
Epoch: 82        Training Loss: 2.010665        Validation Loss: 2.608289
Epoch: 83        Training Loss: 1.956347        Validation Loss: 2.592226
Validation loss decreased (2.606816 --> 2.592226).  Saving model ...
Epoch: 84        Training Loss: 1.953790        Validation Loss: 2.717095
Epoch: 85        Training Loss: 1.942105        Validation Loss: 2.614144
Epoch: 86        Training Loss: 1.941722        Validation Loss: 2.706060
Epoch: 87        Training Loss: 1.916081        Validation Loss: 2.627805
Epoch: 88        Training Loss: 1.888493        Validation Loss: 2.588948
Validation loss decreased (2.592226 --> 2.588948).  Saving model ...
Epoch: 89        Training Loss: 1.871697        Validation Loss: 2.586426
Validation loss decreased (2.588948 --> 2.586426).  Saving model ...
Epoch: 90        Training Loss: 1.851914        Validation Loss: 2.587875
Epoch: 91        Training Loss: 1.903486        Validation Loss: 2.681633
Epoch: 92        Training Loss: 1.859662        Validation Loss: 2.627298
Epoch: 93        Training Loss: 1.864360        Validation Loss: 2.663803
Epoch: 94        Training Loss: 1.832581        Validation Loss: 2.642565
Epoch: 95        Training Loss: 1.795311        Validation Loss: 2.592579
Epoch: 96        Training Loss: 1.813892        Validation Loss: 2.550806
Validation loss decreased (2.586426 --> 2.550806).  Saving model ...
Epoch: 97        Training Loss: 1.816315        Validation Loss: 2.648030
Epoch: 98        Training Loss: 1.792597        Validation Loss: 2.623763
Epoch: 99        Training Loss: 1.790171        Validation Loss: 2.643060
Epoch: 100        Training Loss: 1.778031        Validation Loss: 2.630555
```

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 [12]: 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)
```

```
        ---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call last)
```

```
<ipython-input-12-aeacc80f0572> in <module>()
     31
     32 # load the model that got the best validation accuracy
---> 33 model_scratch.load_state_dict(torch.load('model_scratch.pt'))
     34 test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)


NameError: name 'model_scratch' is not defined
```

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 [7]:
```
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes

batch_size= 20 # how many samples the CNN sees and learn from at a time
valid_size=0.2

# define training and test data directories
data_dir = '/data/landmark_images/'
train_dir = os.path.join(data_dir, 'train')
test_dir = os.path.join(data_dir, 'test')

data_transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                     transforms.ToTensor()])

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

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

```
num_train = len(train_data)
indices = list(range(num_train)) # indices of the enire dataset
np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))  # take 20% of training set size
train_idx, valid_idx = indices[split:], indices[:split]

train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

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


# allow us to iterate data once batch at a time
loaders_transfer = {'train':train_loader ,'valid': valid_loader, 'test':test_loader }


#print(train_data.classes)
classes = [classes_name.split(".")[1] for classes_name in train_data.classes]
#print(classes[49])
```

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

1.1.11 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

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

        criterion_transfer = nn.CrossEntropyLoss()

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

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 [4]: ## TODO: Specify model architecture
        import torch.nn as nn
        from torchvision import models

        model_transfer = models.vgg16(pretrained=True)

        #freezing features- weights
        for param in model_transfer.features.parameters():
            param.require_grad =False

        # replace last layer
        model_transfer.classifier[6] = nn.Linear( model_transfer.classifier[6].in_features , len

        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/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:05<00:00, 104223793.91it/s]


VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
```

```
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=50, bias=True)
  )
)
```

Question 3: Outline the steps you took to get to your final CNN architecture and your reasoning at each step. Describe why you think the architecture is suitable for the current problem.

Answer: Since the VGG-16 model has been trained on millions of images, I used it as a pre-trained model. We only need to replace the final fully connected layer of the model with our own problem to output 50 classes because we have a small dataset and similar data. Also, The parameters of all the feature layers of the model were also frozen.

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 [5]:
```
import signal

from contextlib import contextmanager

import requests


DELAY = INTERVAL = 4 * 60  # interval time in seconds
MIN_DELAY = MIN_INTERVAL = 2 * 60
KEEPALIVE_URL = "https://nebula.udacity.com/api/v1/remote/keep-alive"
TOKEN_URL = "http://metadata.google.internal/computeMetadata/v1/instance/attributes/keep
TOKEN_HEADERS = {"Metadata-Flavor":"Google"}


def _request_handler(headers):
```

```python
    def _handler(signum, frame):
        requests.request("POST", KEEPALIVE_URL, headers=headers)
    return _handler


@contextmanager
def active_session(delay=DELAY, interval=INTERVAL):
    """
    Example:

    from workspace_utils import active session

    with active_session():
        # do long-running work here
    """
    token = requests.request("GET", TOKEN_URL, headers=TOKEN_HEADERS).text
    headers = {'Authorization': "STAR " + token}
    delay = max(delay, MIN_DELAY)
    interval = max(interval, MIN_INTERVAL)
    original_handler = signal.getsignal(signal.SIGALRM)
    try:
        signal.signal(signal.SIGALRM, _request_handler(headers))
        signal.setitimer(signal.ITIMER_REAL, delay, interval)
        yield
    finally:
        signal.signal(signal.SIGALRM, original_handler)
        signal.setitimer(signal.ITIMER_REAL, 0)


def keep_awake(iterable, delay=DELAY, interval=INTERVAL):
    """
    Example:

    from workspace_utils import keep_awake

    for i in keep_awake(range(5)):
        # do iteration with lots of work here
    """
    with active_session(delay, interval): yield from iterable
```

In [10]:
```python
# TODO: train the model and save the best model parameters at filepath 'model_transfer.
num_epochs = 13

with active_session():
# train the model
    model_transfer = train(num_epochs, loaders_transfer, model_transfer, get_optimizer_
                          criterion_transfer, use_cuda, 'model_transfer.pt')
```

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

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

```
Epoch: 1         Training Loss: 2.717466         Validation Loss: 2.052474
Validation loss decreased (inf --> 2.052474).  Saving model ...
Epoch: 2         Training Loss: 1.897653         Validation Loss: 1.759560
Validation loss decreased (2.052474 --> 1.759560).  Saving model ...
Epoch: 3         Training Loss: 1.649123         Validation Loss: 1.691449
Validation loss decreased (1.759560 --> 1.691449).  Saving model ...
Epoch: 4         Training Loss: 1.533880         Validation Loss: 1.613452
Validation loss decreased (1.691449 --> 1.613452).  Saving model ...
Epoch: 5         Training Loss: 1.445309         Validation Loss: 1.533401
Validation loss decreased (1.613452 --> 1.533401).  Saving model ...
Epoch: 6         Training Loss: 1.349331         Validation Loss: 1.505527
Validation loss decreased (1.533401 --> 1.505527).  Saving model ...
Epoch: 7         Training Loss: 1.258682         Validation Loss: 1.499583
Validation loss decreased (1.505527 --> 1.499583).  Saving model ...
Epoch: 8         Training Loss: 1.201682         Validation Loss: 1.448802
Validation loss decreased (1.499583 --> 1.448802).  Saving model ...
Epoch: 9         Training Loss: 1.167291         Validation Loss: 1.456493
Epoch: 10         Training Loss: 1.107164         Validation Loss: 1.462213
Epoch: 11         Training Loss: 1.043663         Validation Loss: 1.495822
Epoch: 12         Training Loss: 0.999197         Validation Loss: 1.451352
Epoch: 13         Training Loss: 0.967533         Validation Loss: 1.401765
Validation loss decreased (1.448802 --> 1.401765).  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 [13]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.215043


Test Accuracy: 68% (853/1250)
```

Step 3: Write Your Landmark Prediction Algorithm
Great job creating your CNN models! Now that you have put in all the hard work of creating accurate classifiers, let's define some functions to make it easy for others to use your classifiers.

1.1.15 (IMPLEMENTATION) Write Your Algorithm, Part 1

Implement the function `predict_landmarks`, which accepts a file path to an image and an integer k, and then predicts the **top k most likely landmarks**. You are **required** to use your transfer learned CNN from Step 2 to predict the landmarks.

An example of the expected behavior of `predict_landmarks`:

```
>>> predicted_landmarks = predict_landmarks('example_image.jpg', 3)
>>> print(predicted_landmarks)
['Golden Gate Bridge', 'Brooklyn Bridge', 'Sydney Harbour Bridge']
```

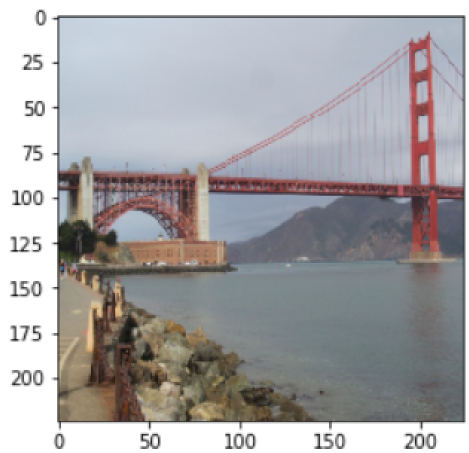
```
In [14]: import cv2
         from PIL import Image

         ## the class names can be accessed at the `classes` attribute
         ## of your dataset object (e.g., `train_dataset.classes`)
         def predict_landmarks(img_path, k):
             ## TODO: return the names of the top k landmarks predicted by the transfer learned
             image = Image.open(img_path)

             transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                             transforms.ToTensor()])

             image= transform(image)
             image.unsqueeze_(0)

             if use_cuda:
                 image = image.cuda()

             model_transfer.eval()

             output = model_transfer(image)
             values, indices = output.topk(k)

             top_k_classes = []

             for i in indices[0].tolist():
                 top_k_classes.append(classes[i])

             model_transfer.train()

             return top_k_classes

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

```
['Golden_Gate_Bridge', 'Forth_Bridge', 'Brooklyn_Bridge', 'Sydney_Harbour_Bridge', 'Niagara_Fall
```

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 [16]: import matplotlib.pyplot as plt
         %matplotlib inline

         def suggest_locations(img_path):
             # get landmark predictions
             predicted_landmarks = predict_landmarks(img_path, 3)
             ## TODO: display image and display landmark predictions

             img = Image.open(img_path)
             plt.imshow(img)
             plt.show()
             print('Is this picture of the',predicted_landmarks[0],',', predicted_landmarks[1],'

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


```
Is this picture of the Golden_Gate_Bridge , Forth_Bridge , or Brooklyn_Bridge
```

1.1.17 (IMPLEMENTATION) Test Your Algorithm

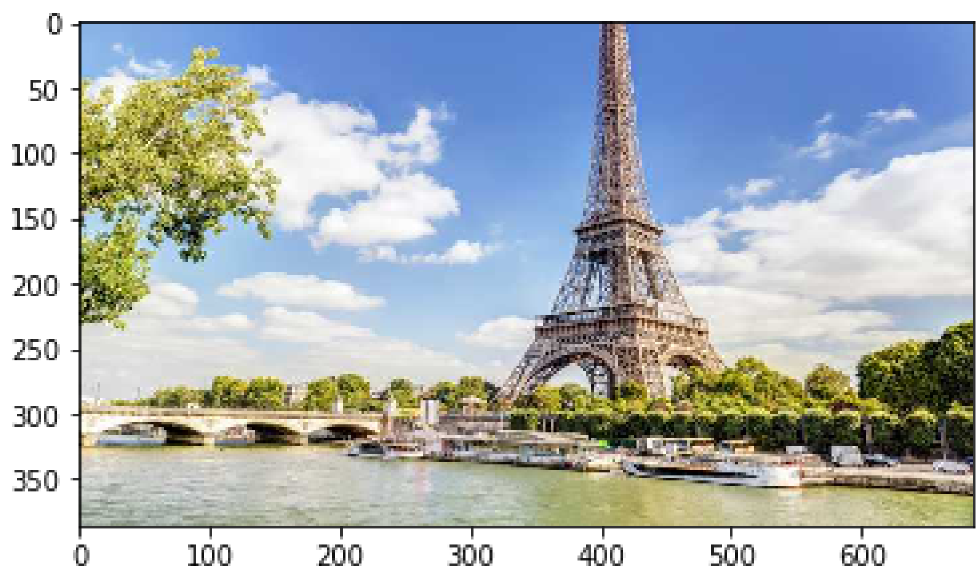
Test your algorithm by running the `suggest_locations` function on at least four images on your computer. Feel free to use any images you like.

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) In general, the outputs are better than what I expected. Possible points for improvement:- More related training data should be fed into the model. Trying some changes in the model architecture, like adding more fully connected layers Trying other hyperparameter values

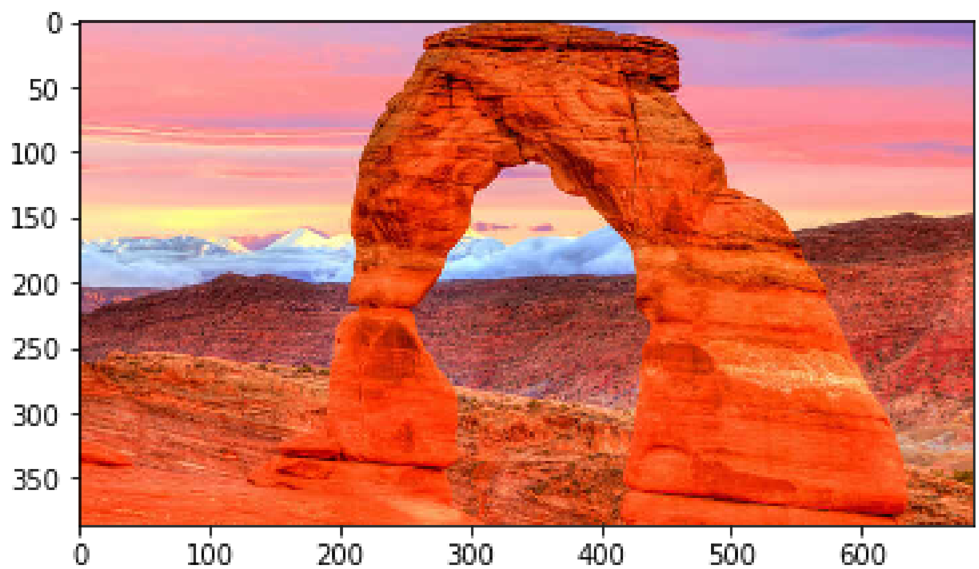
```
In [19]: ## TODO: Execute the `suggest_locations` function on
         ## at least 4 images on your computer.
         ## Feel free to use as many code cells as needed.

         suggest_locations('myimages/pic1.jpg')


         suggest_locations('myimages/pic2.jpg')

         suggest_locations('myimages/pic3.jpg')

         suggest_locations('myimages/pic4.jpg')
```

Is this picture of the Edinburgh_Castle , Machu_Picchu , or Ljubljana_Castle



Is this picture of the Eiffel_Tower , Vienna_City_Hall , or Terminal_Tower

Is this picture of the Stonehenge , Taj_Mahal , or Wroclaws_Dwarves



Is this picture of the Delicate_Arch , Badlands_National_Park , or Grand_Canyon