landmark

November 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 Haleakalā 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.

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!

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

# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 20
# percentage of training set to use as validation
```

```
valid_size = 0.2

img_size = 32
n_out = 50

#ImageNet standards
mean = torch.tensor([0.4915, 0.4823, 0.4468])
std = torch.tensor([0.2470, 0.2435, 0.2616])
normalize = transforms.Normalize(mean.tolist(), std.tolist())
denormalize = transforms.Normalize((-mean / std).tolist(), (1.0 / std).tolist())

train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.2),
    transforms.RandomRotation(10),
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.RandomChoice([
            transforms.ColorJitter(hue=0.1),
            transforms.ColorJitter(brightness=0.2),
            transforms.ColorJitter(saturation=0.2),
            transforms.ColorJitter(contrast=0.2),
        ]),
    # -------------
    transforms.ToTensor(),
    transforms.Normalize(mean,std),
])

test_transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean,std),
])

# Define datasets for training and testing
train_data = datasets.ImageFolder('landmark_images/train',␣
 ↪transform=train_transform)
test_data = datasets.ImageFolder('landmark_images/test',␣
 ↪transform=test_transform)

# obtain training indices that will be used for validation
num_train = len(train_data)
indices = list(range(num_train))
np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]
```

```
# define samplers for obtaining training and validation batches
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

# prepare data loaders (combine dataset and sampler)
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
    sampler=train_sampler, num_workers=num_workers)
valid_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
    sampler=valid_sampler, num_workers=num_workers)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
    num_workers=num_workers)

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

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

Answer: A common feature of handheld pictures from phones are small rotations and horizontal flips due to the selfie camera. To generalize over them, I've taken random horizontal flip and rotation (mean=10 deg) transformations. To have a lightweight model, I've reduced the number of pixels to 32, while resizing and cropping accordingly to avoid the padding produced by the small rotations.

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.

[2]:
```
import matplotlib.pyplot as plt
%matplotlib inline

## TODO: visualize a batch of the train data loader

# helper function to un-normalize and display an image
def imshow(img):
    plt.imshow(denormalize(img).permute(1,2,0))  # convert from Tensor image

dataiter = iter(train_loader)
images, labels = dataiter.next()

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

```
classes = [item[3:].replace("_", " ") for item in train_data.classes]

# plot the images in the batch, along with the corresponding labels
fig = plt.figure(figsize=(15, 4))
# display 20 images
for idx in np.arange(10):
    ax = fig.add_subplot(2, 5, idx+1, xticks=[], yticks=[])
    imshow(images[idx])
    ax.set_title(classes[labels[idx]])
```

1.1.3 Initialize use_cuda variable

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

1.1.4 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

[4]:
```
## TODO: select loss function
from torch import optim
from torch import nn
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.

[5]:
```
import torch.nn.functional as F

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

        ## Define layers of a CNN
        # convolutional layer (sees 32x32x3 image tensor)
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        # convolutional layer (sees 16x16x16 tensor)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        # convolutional layer (sees 8x8x32 tensor)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        # max pooling layer
        self.pool = nn.MaxPool2d(2, 2)
        # linear layer (64 * 28 * 28 -> 256)
        self.fc1 = nn.Linear(64 * 28 * 28, 256)
        # linear layer (256 -> 50)
        self.fc2 = nn.Linear(256, 50)
        # dropout layer (p=0.25)
        self.dropout = nn.Dropout(0.25)

        # Batch norm
        self.batch_norm2d = nn.BatchNorm2d(32)
        self.batch_norm1d = nn.BatchNorm1d(256)




    def forward(self, x):
        ## Define forward behavior
        # add sequence of convolutional and max pooling layers
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.batch_norm2d(x)
        x = self.pool(F.relu(self.conv3(x)))
        # flatten image input
        x = x.view(-1, 64 * 28 * 28)
        # add dropout layer
        x = self.dropout(x)
        # add 1st hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        x = self.batch_norm1d(x)
        # add dropout layer
        x = self.dropout(x)
```

```
        # add 2nd hidden layer, with relu activation function
        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: Since the goal was to acheive 20% accuracy, I started with the architecture provided in the lectures, which turned out to do the job. It presents 3 convolution + ReLU layers, which is the common structure in popular CNNs like VGG16. The dropout applied to the two last ReLU layers reduce 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`.

[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
            ## record the average training loss, using something like
```

```
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data.
↪item() - train_loss))
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            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
            output = model(data)
            loss = criterion(output, target)
            valid_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 stored in save_path
        if valid_loss <= valid_loss_min:
            print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model␣
↪...'.format(
            valid_loss_min,
            valid_loss))
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss
```

```
    return model
```

1.1.7 (IMPLEMENTATION) Experiment with the Weight Initialization

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

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

[7]:
```
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.BatchNorm2d):
        m.weight.data.fill_(1)
        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(10, loaders_scratch, model_scratch,
 ↪get_optimizer_scratch(model_scratch),
                      criterion_scratch, use_cuda, 'ignore.pt')
```

```
/home/manuel/anaconda3/envs/dl-course/lib/python3.7/site-
packages/torch/nn/functional.py:718: UserWarning: Named tensors and all their
associated APIs are an experimental feature and subject to change. Please do not
use them for anything important until they are released as stable. (Triggered
internally at  /tmp/pip-req-build-qq9311m_/c10/core/TensorImpl.h:1156.)
  return torch.max_pool2d(input, kernel_size, stride, padding, dilation,
ceil_mode)
```

```
Epoch: 1        Training Loss: 3.680342         Validation Loss: 3.383841
Validation loss decreased (inf --> 3.383841).  Saving model …
Epoch: 2        Training Loss: 3.220806         Validation Loss: 3.227353
Validation loss decreased (3.383841 --> 3.227353).  Saving model …
Epoch: 3        Training Loss: 2.962814         Validation Loss: 3.125516
Validation loss decreased (3.227353 --> 3.125516).  Saving model …
```

```
Epoch: 4        Training Loss: 2.759545         Validation Loss: 3.028284
Validation loss decreased (3.125516 --> 3.028284).  Saving model …
Epoch: 5        Training Loss: 2.600085         Validation Loss: 2.976320
Validation loss decreased (3.028284 --> 2.976320).  Saving model …
Epoch: 6        Training Loss: 2.498665         Validation Loss: 3.260376
Epoch: 7        Training Loss: 2.446877         Validation Loss: 2.941050
Validation loss decreased (2.976320 --> 2.941050).  Saving model …
Epoch: 8        Training Loss: 2.224681         Validation Loss: 2.873510
Validation loss decreased (2.941050 --> 2.873510).  Saving model …
Epoch: 9        Training Loss: 2.088446         Validation Loss: 2.964906
Epoch: 10       Training Loss: 1.930027         Validation Loss: 2.951501
```

1.1.8 (IMPLEMENTATION) Train and Validate the Model

Run the next code cell to train your model.

[8]:
```
## TODO: you may change the number of epochs if you'd like,
## but changing it is not required
num_epochs = 10

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

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

# reset the model parameters
model_scratch.apply(default_weight_init)

# train the model
model_scratch = train(num_epochs, loaders_scratch, model_scratch,␣
 ↪get_optimizer_scratch(model_scratch),
                      criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1        Training Loss: 3.595787         Validation Loss: 3.403126
Validation loss decreased (inf --> 3.403126).  Saving model …
Epoch: 2        Training Loss: 3.228036         Validation Loss: 3.277391
Validation loss decreased (3.403126 --> 3.277391).  Saving model …
Epoch: 3        Training Loss: 3.004406         Validation Loss: 3.126817
Validation loss decreased (3.277391 --> 3.126817).  Saving model …
Epoch: 4        Training Loss: 2.820500         Validation Loss: 3.056196
Validation loss decreased (3.126817 --> 3.056196).  Saving model …
Epoch: 5        Training Loss: 2.632769         Validation Loss: 2.997713
Validation loss decreased (3.056196 --> 2.997713).  Saving model …
Epoch: 6        Training Loss: 2.491124         Validation Loss: 3.103961
Epoch: 7        Training Loss: 2.381711         Validation Loss: 2.952242
```

```
Validation loss decreased (2.997713 --> 2.952242).  Saving model …
Epoch: 8        Training Loss: 2.246535         Validation Loss: 2.988248
Epoch: 9        Training Loss: 2.120590         Validation Loss: 2.920899
Validation loss decreased (2.952242 --> 2.920899).  Saving model …
Epoch: 10       Training Loss: 1.995600         Validation Loss: 2.867684
Validation loss decreased (2.920899 --> 2.867684).  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%.

[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: 2.719281


Test Accuracy: 33% (417/1250)
```

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.

[10]:
```
### 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.

[11]:
```
## TODO: select loss function
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`.

[12]:
```
## TODO: Specify model architecture
from torchvision import models

model_transfer = models.vgg16(pretrained=True)

# Freeze training for all "features" layers
for param in model_transfer.features.parameters():
    param.requires_grad = False
# Change last layer (requires_grad True by default)
n_inputs = model_transfer.classifier[6].in_features
last_layer = nn.Linear(n_inputs, len(classes))
model_transfer.classifier[6] = last_layer
#-#-# Do NOT modify the code below this line. #-#-#

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

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: Similarly to the lecture developments, I took a pretrained VGG16 due to its nice balance between tractability and performance. Then, I froze the original weights and replaced the last fully connected layer to only retrain that last layer of the network.

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'`.

[13]:
```
# TODO: train the model and save the best model parameters at filepath␣
↪'model_transfer.pt'
num_epochs = 20
model_transfer = train(num_epochs, loaders_transfer, model_transfer,␣
↪get_optimizer_transfer(model_transfer),
                      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: 2.136046         Validation Loss: 1.357144
Validation loss decreased (inf --> 1.357144).  Saving model …
Epoch: 2        Training Loss: 1.187221         Validation Loss: 1.160573
```

```
Validation loss decreased (1.357144 --> 1.160573).  Saving model …
Epoch: 3        Training Loss: 0.910623         Validation Loss: 1.108385
Validation loss decreased (1.160573 --> 1.108385).  Saving model …
Epoch: 4        Training Loss: 0.720761         Validation Loss: 1.004999
Validation loss decreased (1.108385 --> 1.004999).  Saving model …
Epoch: 5        Training Loss: 0.593541         Validation Loss: 0.991274
Validation loss decreased (1.004999 --> 0.991274).  Saving model …
Epoch: 6        Training Loss: 0.479454         Validation Loss: 1.018056
Epoch: 7        Training Loss: 0.418789         Validation Loss: 1.039585
Epoch: 8        Training Loss: 0.327667         Validation Loss: 1.014092
Epoch: 9        Training Loss: 0.274942         Validation Loss: 1.038353
Epoch: 10       Training Loss: 0.263151         Validation Loss: 1.031971
Epoch: 11       Training Loss: 0.230219         Validation Loss: 1.004532
Epoch: 12       Training Loss: 0.190237         Validation Loss: 1.010533
Epoch: 13       Training Loss: 0.164766         Validation Loss: 1.028117
Epoch: 14       Training Loss: 0.151496         Validation Loss: 1.055496
Epoch: 15       Training Loss: 0.133682         Validation Loss: 1.033857
Epoch: 16       Training Loss: 0.123344         Validation Loss: 1.034674
Epoch: 17       Training Loss: 0.101116         Validation Loss: 1.081887
Epoch: 18       Training Loss: 0.100154         Validation Loss: 1.073098
Epoch: 19       Training Loss: 0.095090         Validation Loss: 1.102523
Epoch: 20       Training Loss: 0.084613         Validation Loss: 1.063520
```

[13]:
```
<All keys matched successfully>
```

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%.

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

```
Test Loss: 0.830721


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']
```

[15]:

```python
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 CNN
    img = Image.open(img_path)
    img = test_transform(img)
    img.unsqueeze_(0)

    if use_cuda:
        img = img.cuda()

    output = model_transfer(img)
    top_p, top_idx = torch.topk(output,k)
    idx = np.squeeze(top_idx.numpy()) if not use_cuda else np.squeeze(top_idx.
 ↪cpu().numpy())

    places =[]
    for i in idx:
        places.append(classes[i])

    return places

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

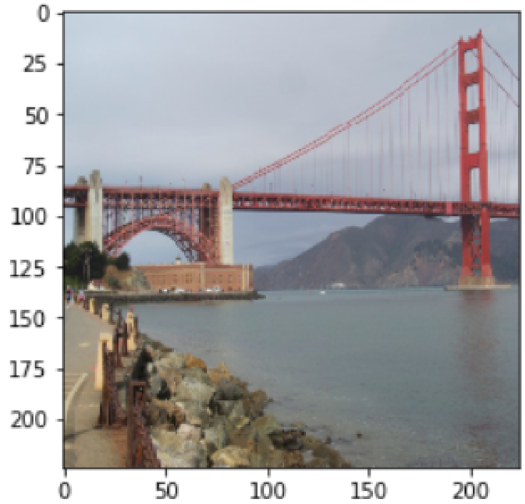
[15]:

```
['Golden Gate Bridge',
 'Forth Bridge',
 'Brooklyn Bridge',
 'Sydney Harbour Bridge',
 'Niagara Falls']
```

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!

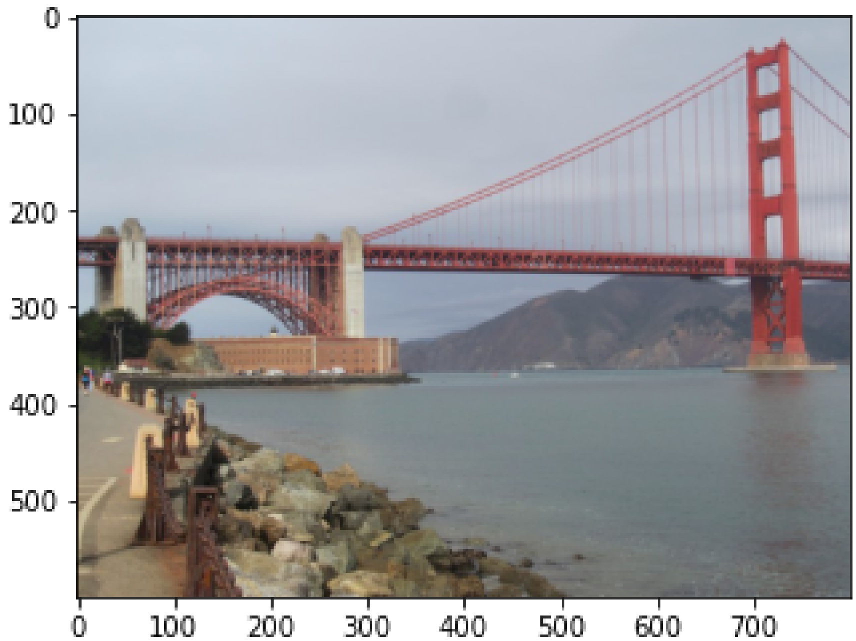



```
Is this picture of the
Golden Gate Bridge, Brooklyn Bridge, or Sydney Harbour Bridge?
```

[16]:
```
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')
    print('{}, {}, or {}?'.format(predicted_landmarks[0],␣
 ↪predicted_landmarks[1],predicted_landmarks[2]))


# 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

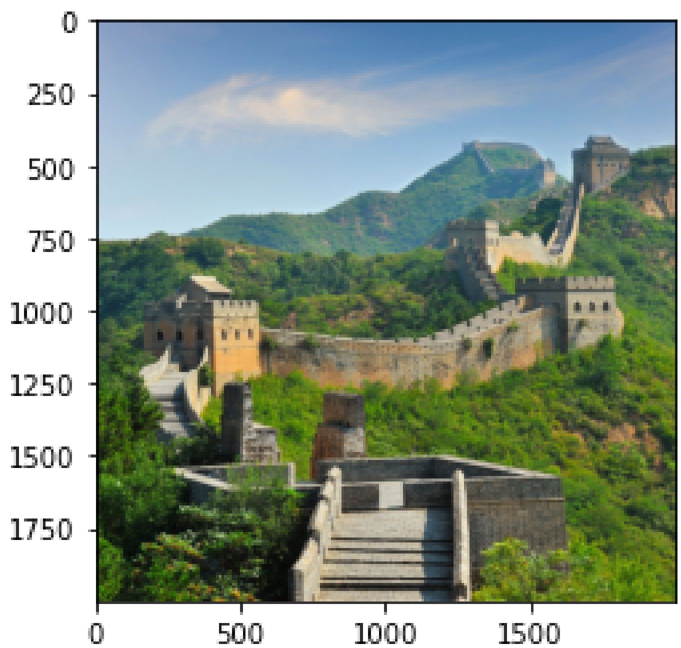
Test your algorithm by running the `suggest_locations` function on at least four images on your computer. Feel free to use any images you like.

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

Answer: It is tremendously impressive compared to traditional computer vision. However, it is still far from performing reliably. This could be improved by using a larger dataset, retraining the whole network or using a more powerfull model such as ResNet.

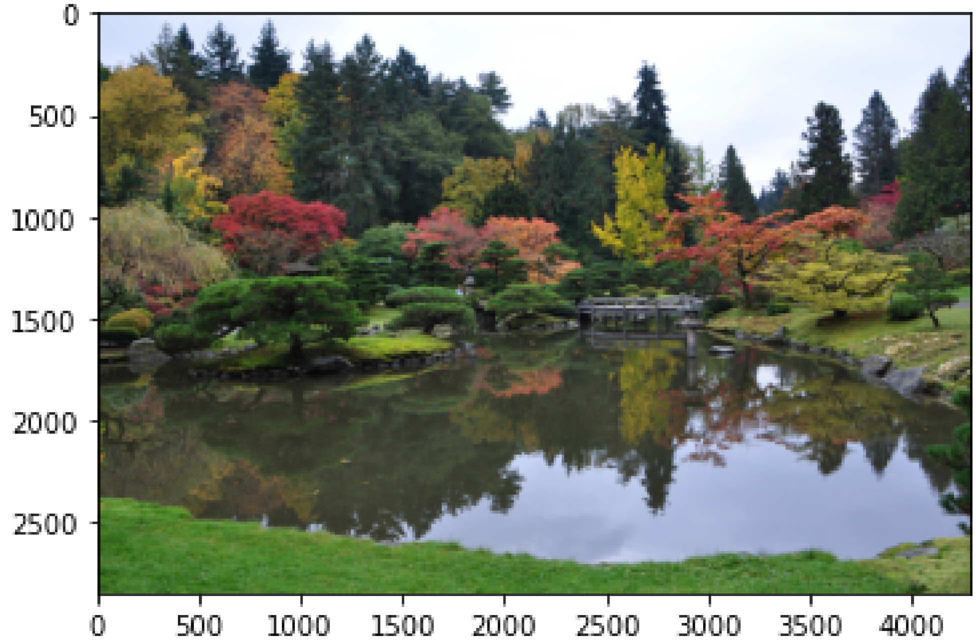
[17]:
```
## TODO: Execute the `suggest_locations` function on
## at least 4 images on your computer.
## Feel free to use as many code cells as needed.
suggest_locations('samples/niagara.jpg')
suggest_locations('samples/china_wall.jpg')
suggest_locations('samples/seattle_japanese_garden.jpg')
suggest_locations('samples/taj_mahal.jpeg')
suggest_locations('samples/wroclaw_dwarves.jpg')
```

Is this picture of the
Niagara Falls, Yellowstone National Park, or Gullfoss Falls?

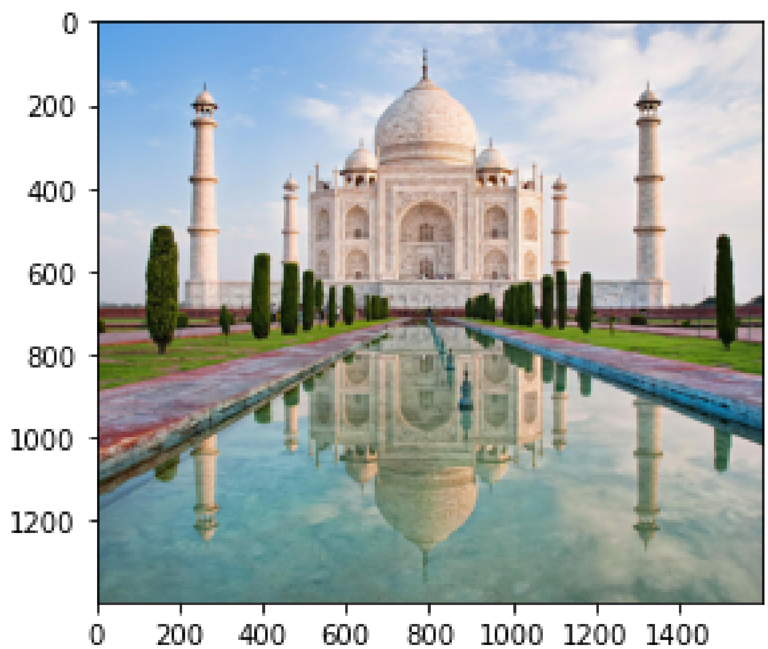


Is this picture of the

Edinburgh Castle, Ljubljana Castle, or Great Wall of China?



Is this picture of the
Seattle Japanese Garden, Central Park, or Taj Mahal?



Is this picture of the
Taj Mahal, Vienna City Hall, or Stockholm City Hall?

Is this picture of the
Wroclaws Dwarves, Delicate Arch, or Machu Picchu?