January 25, 2023

Part 2: Understanding Data Normalization

The reason why pixel value scaling on a per image basis and scaling by dividing the entire batch by the maximum value of the entire batch result in the same output is because in both cases, the images are divided by 255. This is hardcoded, so, no matter what the actual maximum value the image contains, every image will have every pixel divided by 255. To see this in action, one can look at the class ToTensor in the PyTorch documentation (https://pytorch.org/vision/main/_modules/torchvision/transforms/transforms.html#ToTensor), the __call__ method returns the output of the Functional.to_tensor() method. In the Functional.to_tensor() method, there are checks for the correct dimensionality (at least 2 or 3 dimensions for the input to the function) and for the correct object type (numpy array with unit8 datatype or a PIL object). If those checks pass, then, the entire image is divided by 255, as seen in line 155 of the Functional code on the PyTorch github (https://github.com/pytorch/vision/blob/5dd95944c609ac399743fa843ddb7b83780512b3/torchvision/transforms/functional.py#L125).

Part 3: Programming Tasks

3.1: Setting Up Your Conda Environment

I installed pytorch version 1.13 as only 1.12 and above has mac GPU support. The full environment.yml file is included in the zip file with the submission.

3.2 Becoming Familiar with torchvision.transforms:

Plot of projective transformations is below

January 25, 2023

In order to make the image on the top left of the plot appear as close to the image on the top right of the plot, I used a perspective transform. In the perspective transform, I picked 4 start points that correspond to locations around the stop sign (a point to the top left of the stop sign, the top of the lamp on the right of the stop sign, the bottom of the lamp on the right of the stop sign, and a point on the bottom left of the stop sign). I then picked four end points that correspond to similar locations around the stop sign in the target image. The four points were a point on the top left of the stop sign, a point on the upper right of the target image but pretty far from the stop sign, and two points on the ground below the stop sign. There was a fair bit of

experimentation, but the best I did was to get the angle of the pole and stop sign similar to the angle of those features in the target image. Unfortunately, the Wasserstein distance did not go up but I believe that is due to the black region in the transformed image and the fact that the stop sign only makes up a small portion of the image. Next time, I would try to do this with more of the stop sign inside the frame.

3.3 Creating Your Own Dataset Class

The Dataset class was implemented given the instructions in the lab. The transformations in the tvt.Compose pipeline were:

1. Converting the PIL image to a tensor using ToTensor()
2. Resizing the image to 256 x 256 using Resize()
3. Normalizing the images from -1 to 1 using Normalize(). The mean was 0 and the standard deviation was 1.
4. The augmentations were color jitter, random horizontal flip, and random rotation. These were implemented using ColorJitter(0.75, 0.75), RandomHorizontalFlip(p = 0.75), and RandomRotation(degrees = 45).

The methods were chosen to provide invariance to different lighting and rotations that can be seen in the real world. For example, one of my images (as seen below) is a mechanical pencil and earphones case. In theory, one can take a picture of the pencil and case from any angle, which rotates the tensor loaded into the training network. Horizontal flip and random rotations provide invariance for that case. It was important to ponder that rotation does not semantically change the meaning of the actual object, like it would in an MNIST dataset (think about a 6 being rotated - it would look like a 9 in some cases). However, that was not the case for any of the objects in my small dataset, so, it made sense to perform the rotations and flips. The color jitter was done because images can be taken with different lighting conditions. As one can see in the images below, the pencil and earphones case seems to have brighter lighting conditions while the picture of my slippers seems to be taken with less light in my augmented image. The main goal of the color jitter is therefore to hopefully augment the same semantic object with a different lighting condition that may be seen in the real world.

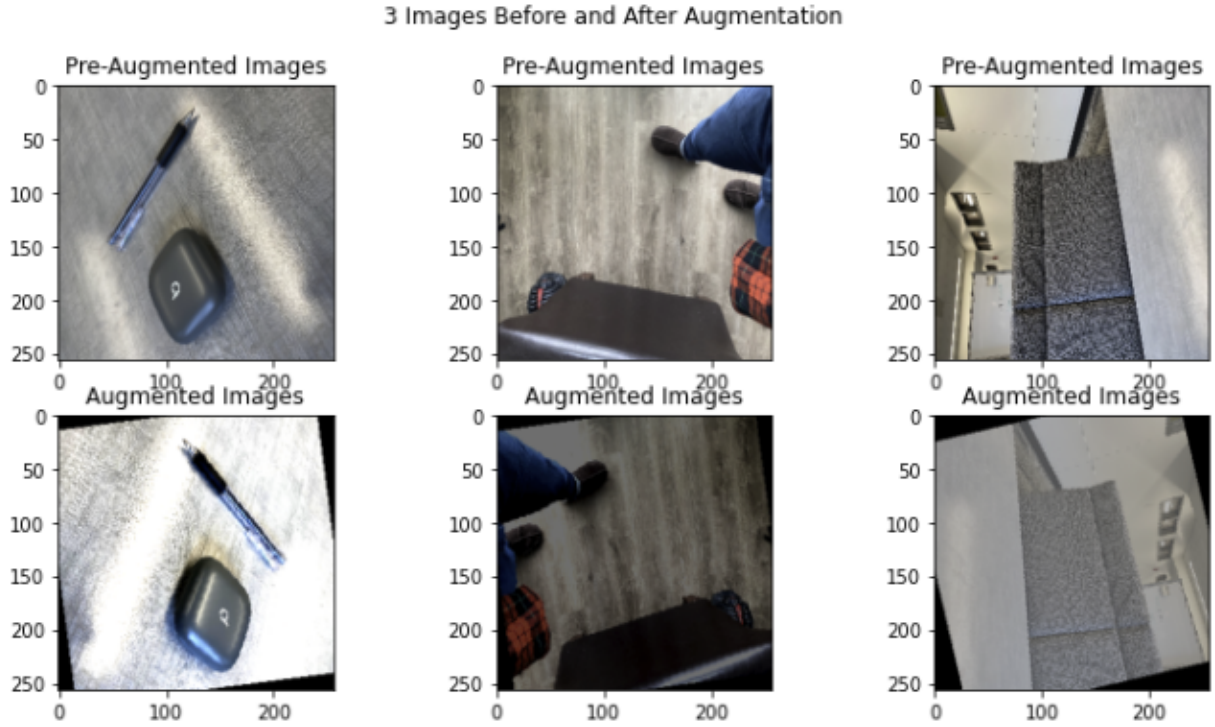
Below is the code of the class and the output of the instantiated MyDataset class.

January 25, 2023

```
root = '10_HW2_photos/'
import random
class MyDataset(torch.utils.data.Dataset):
    def __init__(self, root):
        super(MyDataset).__init__()
        self.root = (dir_root + root)
        self.filenames = []
        for filename in os.listdir(self.root):
            self.filenames.append(filename)
        self.augmentation = tvt.Compose([tvt.ToTensor(), tvt.Resize((256,256)),
                                         tvt.Normalize([0], [1]) ,
                                         tvt.ColorJitter(0.75, 0.75) ,
                                         tvt.RandomHorizontalFlip( p = 0.75),
                                         tvt.RandomRotation(degrees = 45) ])
        self.resize = tvt.Compose([tvt.ToTensor(), tvt.Normalize([0], [1]), tvt.Resize((256, 256))])

    def __len__(self):
        return len(self.filenames)

    def __getitem__(self, index):
        item = Image.open(self.root + self.filenames[index])
        #print('index is', index)
        aug_img = self.augmentation(item)
        class_label = random.randint(0,10)
        return aug_img, class_label

    def convertOriginal_to_np(self, num_images_to_display):
        count = 0
        orig = []
        for img in self.filenames:
            full_img_file_path = self.root + img
            img_tensor = self.resize(Image.open(full_img_file_path))
            img_np = np.transpose(np.array(img_tensor), (1, 2, 0))
            orig.append(img_np)
            if count >= num_images_to_display - 1:
                break
            else:
                count = count + 1
        return orig

my_dataset = MyDataset(root)
print(len(my_dataset))
index = 2
print(my_dataset[index][0].shape, my_dataset[index][1])

```

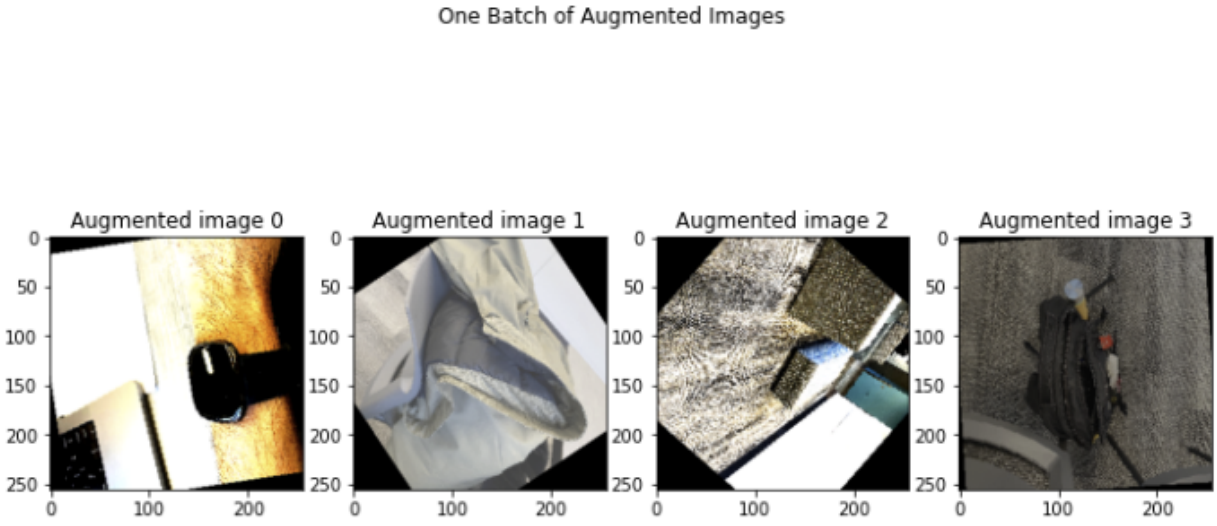
```
10
torch.Size([3, 256, 256]) 1
```

Below are 3 original images in the local directory and the same three that have gone through the augmentation transforms:



Part 3.4: Generating Data in Parallel

The instantiated MyDataset class object was wrapped in the torch.utils.data.DataLoader class. The first goal of this section is to display one batch of 4 images. The second goal of this section was to understand the speed of loading 1000 images using the __getitem__ method in the MyDataset class versus looping through a batch of images in the instantiated MyDataset with the functionality provided in the DataLoader. The results are below:



January 25, 2023

Below is the experiment results for using __getitem__ 1000 times to process 1000 images versus using the DataLoader batch loading to process 1000 images:

Time to Process 1000 images: DataLoader vs Dataset in PyTorch					
		Number of Workers			
		1	2	4	6
Batch Size	1	109.4956629	N/A	N/A	N/A
	4	N/A	64.17058492	33.85705996	23.88801575
	8	N/A	65.36338878	34.51232004	24.1432941
	10	N/A	65.39961028	33.93644595	24.48525906
Note that all times are in seconds. Batch size 1 and number of workers 1 corresponds to using the __getitem__ method in the Dataset class					

The best parameters for processing 1000 images was using 6 workers and a batch size of 4, although a batch size of 8 or 10 would have done about as well using 6 workers! The __getitem__ method was more than 4 times slower than the fastest dataloader configuration!

Below is the code of the experiments which show the actual results:

Experiment for __getitem__:

```
import time
seed = 0
random.seed(seed)
##### START MEASURING TIME FOR my_dataset ########
start_time = time.time()
for i in range(1000):
    my_dataset.__getitem__(index = random.randint(0,9))
    if (i % 100 == 0):
        end_time = time.time()
print("Total time for fetching 1000 images without a dataloader was", end_time - start_time, " seconds")
##### END MEASURING TIME FOR my_dataset

```

```
Total time for fetching 1000 images without a dataloader was 109.49566292762756  seconds
```

January 25, 2023

Experiment for DataLoader:

```
batch_steps = [4, 8, 10]
num_worker_steps = [2, 4, 6]
results = np.zeros((3,3))
img1_test = MyDataset(root)
#print(results)
num_items = 1000
for m, i in enumerate(batch_steps):
    for n, j in enumerate(num_worker_steps):
        print("num_workers is", j, "and batch_size is", i)
        my_dataloader_test = torch.utils.data.DataLoader(img1_test, batch_size = i, num_workers = j)
        ##### START MEASURING TIME FOR my_dataloader ######
        start_time = time.time()
        count = 0
        batch_size = i
        for batch in my_dataloader_test:
            #print('in for loop with ', i, 'as ' , i, 'and j as', j)
            count+=1
            #print(count)
            if count >= (num_items/batch_size):
                end_time = time.time()
        ##### END MEASURING TIME FOR my_dataloader ######
                results[m][n] = (end_time - start_time)
                print("With batch size", batch_size, "and num_workers", j,  "loading", num_items,
                      "images took", results[m][n], "seconds")


print("Table of total time to load images. Each row is batch_size and each column is num_workers")
print("The best performance was ", results.min(), "seconds")
print(results)
```

```
num_workers is 2 and batch_size is 4
With batch size 4 and num_workers 2 loading 1000 images took 64.17058491706848 seconds
num_workers is 4 and batch_size is 4
With batch size 4 and num_workers 4 loading 1000 images took 33.857059955596924 seconds
num_workers is 6 and batch_size is 4
With batch size 4 and num_workers 6 loading 1000 images took 23.888015747070312 seconds
num_workers is 2 and batch_size is 8
With batch size 8 and num_workers 2 loading 1000 images took 65.36338877677917 seconds
num_workers is 4 and batch_size is 8
With batch size 8 and num_workers 4 loading 1000 images took 34.512320041656494 seconds
num_workers is 6 and batch_size is 8
With batch size 8 and num_workers 6 loading 1000 images took 24.143294095993042 seconds
num_workers is 2 and batch_size is 10
With batch size 10 and num_workers 2 loading 1000 images took 65.3996102809906 seconds
num_workers is 4 and batch_size is 10
With batch size 10 and num_workers 4 loading 1000 images took 33.93644595146179 seconds
num_workers is 6 and batch_size is 10
With batch size 10 and num_workers 6 loading 1000 images took 24.48525905609131 seconds
Table of total time to load images. Each row is batch_size and each column is num_workers
The best performance was  23.888015747070312 seconds
[[64.17058492 33.85705996 23.88801575]
 [65.36338878 34.51232004 24.1432941 ]
 [65.39961028 33.93644595 24.48525906]]
```