Homework 2

Alexandre Olivé Pellicer

1. Introduction

2. Understanding Pixel Value Scaling and Normalization

I first create the simulated batch of images in which the pixel values are limited between 0 and 32:

```
images_limited = torch.randint(0, 33, (4, 3, 5, 9)).type(torch.uint8)
```

And then I apply the 2 different scaling methods:

- manual scaling

```
images_limited_scaled_manual = images_limited / images_limited.max().float()
print(images_limited_scaled_manual[0])
```

```
tensor([[[0.3750, 0.5312, 0.1875, 0.2188, 0.4062, 0.2812, 0.8750, 0.2188,
          0.8125],
         [0.2500, 0.0625, 0.6875, 0.0938, 0.6250, 0.6562, 0.4688, 0.6875,
          1.0000],
         [0.1562, 0.2500, 0.7500, 0.3750, 0.1250, 0.6875, 0.0625, 1.0000,
          0.6875],
         [0.6250, 0.8438, 0.3438, 1.0000, 0.8125, 0.8125, 0.4688, 0.5312,
          0.9062],
         [0.4375, 0.5312, 0.1250, 0.5625, 1.0000, 0.6562, 0.0625, 0.0312,
          0.6562]],

        [[0.7188, 0.4375, 0.2188, 0.6875, 0.1562, 0.5000, 0.5000, 0.1875,
```

- "automated" scaling based on tvt.ToTensor()

```
images_limited_scaled_automated = torch.zeros_like(images_limited).float()
for i in range(images_limited.shape[0]):
    images_limited_scaled_automated[i] = tvt.ToTensor()(numpy.transpose(images_limited[i].numpy(), (1,2,0)))
print(images_limited_scaled_automated[0])
```

```
tensor([[[0.0471, 0.0667, 0.0235, 0.0275, 0.0510, 0.0353, 0.1098, 0.0275,
          0.1020],
         [0.0314, 0.0078, 0.0863, 0.0118, 0.0784, 0.0824, 0.0588, 0.0863,
          0.1255],
         [0.0196, 0.0314, 0.0941, 0.0471, 0.0157, 0.0863, 0.0078, 0.1255,
          0.0863],
         [0.0784, 0.1059, 0.0431, 0.1255, 0.1020, 0.1020, 0.0588, 0.0667,
          0.1137],
         [0.0549, 0.0667, 0.0157, 0.0706, 0.1255, 0.0824, 0.0078, 0.0039,
          0.0824]],
```

From the obtained results we can see that when working with an image that has values between 0 and 32 the resulting image after scaling is different when it is done manually compared to when it is "automated".

Nevertheless, if we repeat the same experiment with a simulated batch of images in which the pixel values are limited between 0 and 255:

```
images_spanded = torch.randint(0, 256, (4, 3, 5, 9)).type(torch.uint8)
```

- manual scaling

```
images_spanded_scaled_manual = images_spanded / images_spanded.max().float()
print(images_spanded_scaled_manual[0])
```

```
tensor([[[0.8235, 0.0431, 0.7961, 0.3020, 0.3020, 0.0353, 0.2471, 0.7137,
          0.3490],
         [0.4000, 0.7373, 0.4157, 0.0667, 0.0471, 0.6392, 0.3216, 0.4745,
          0.1490],
         [0.9882, 0.8745, 0.7176, 0.7176, 0.5961, 0.6157, 0.5137, 0.9882,
          0.3059],
         [0.9961, 0.4824, 0.2784, 0.9804, 0.0627, 0.3882, 0.1725, 0.5373,
          0.8431],
         [0.3882, 0.5804, 0.5255, 0.6196, 0.4196, 0.8745, 0.2275, 0.4157,
          0.5961]],

        [[0.0667, 0.3804, 0.4471, 0.0000, 0.1412, 0.1765, 0.0784, 0.5686,
```

- "automated" scaling based on tvt.ToTensor()

```
images_spanded_scaled_automated = torch.zeros_like(images_spanded).float()
for i in range(images_spanded.shape[0]):
    images_spanded_scaled_automated[i] = tvt.ToTensor()(numpy.transpose(images_spanded[i].numpy(), (1,2,0)))
print(images_spanded_scaled_automated[0])
```

```
tensor([[[0.8235, 0.0431, 0.7961, 0.3020, 0.3020, 0.0353, 0.2471, 0.7137,
          0.3490],
         [0.4000, 0.7373, 0.4157, 0.0667, 0.0471, 0.6392, 0.3216, 0.4745,
          0.1490],
         [0.9882, 0.8745, 0.7176, 0.7176, 0.5961, 0.6157, 0.5137, 0.9882,
          0.3059],
         [0.9961, 0.4824, 0.2784, 0.9804, 0.0627, 0.3882, 0.1725, 0.5373,
          0.8431],
         [0.3882, 0.5804, 0.5255, 0.6196, 0.4196, 0.8745, 0.2275, 0.4157,
          0.5961]],

        [[0.0667, 0.3804, 0.4471, 0.0000, 0.1412, 0.1765, 0.0784, 0.5686,
```

we see that in this case the result is the same.

This is because of the implementation of the __call__() function of the class "ToTensor". In this method, the method "F.to_tensor(pic)" is called. Inside this method we see the following lines of code:

```
146
147     if isinstance(pic, np.ndarray):
148         # handle numpy array
149         if pic.ndim == 2:
150             pic = pic[:, :, None]
151
152         img = torch.from_numpy(pic.transpose((2, 0, 1))).contiguous()
153         # backward compatibility
154         if isinstance(img, torch.ByteTensor):
155             return img.to(dtype=default_float_dtype).div(255)
156         else:
157             return img
158
```

As we can see in line 155, the image is divided by 255. So, when creating the batch with values between 0 and 32, when scaling manually, we were dividing by 32. But, when using the tvt.ToTensor(), we were dividing by 255.

2.1. Try it yourself

```python
img_array = numpy.load('test_image.npy')
max_value = numpy.max(img_array)
min_value = numpy.min(img_array)
print('max value: ', max_value, " min value: ", min_value)

img_array_devided_max = img_array / max_value
print('max value when dividing by max: ', numpy.max(img_array_devided_max), ' min value when dividing by max: ', numpy.min(img_array_devided_max))
img_array_devided_255 = img_array / 255
print('max value when dividing by 255: ', numpy.max(img_array_devided_255), ' min value when dividing by 255: ', numpy.min(img_array_devided_255))
```

[100] ✓ 0.0s Python

```
max value:  219  min value:  0
max value when dividing by max:  1.0  min value when dividing by max:  0.0
max value when dividing by 255:  0.8588235294117647  min value when dividing by 255:  0.0
```

We can see that when we divide the image by its max value, the resulting image, the scaled one, occupies the full range from 0 to 1. This is the desired behavior. Nevertheless, when dividing by 255, the maximum value is 0.85 so we are not taking advantage of the full range between 0 and 1. This happens because the max value of the image is not 255, it is 219.

3. Programming Tasks

3.1. Setting Up Your Conda Environment

3.2. Becoming Familiar with torchvision.transforms

Fig 1: Original front Image

Fig 2: Original oblique Image

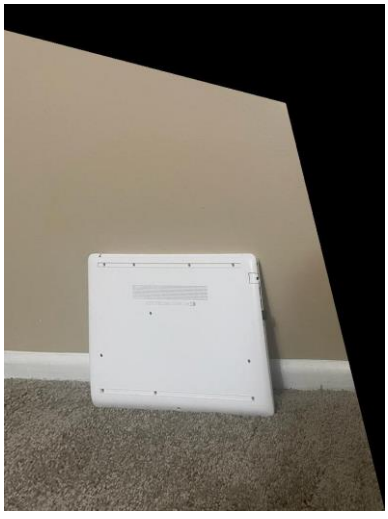


Fig 3: Original front Image Fig 4: Transformed Image

Although thinking about implementing the system to keep trying different projective parameters in a loop until finding the parameters that will make one image look reasonably similar to the other, we found that the number of parameters that needed to be tried was very large, since we would have to iterate 8 different parameters using the perspective transformation. This would take a lot of time. For this reason, we decided to iterate manually. We have used the projective transformation and we set the parameters in function of the distance metric obtained and evaluating visually the similarity between the original front image and the transformed oblique image. The final parameters used are the following:

```
endpoints = [[0-200,0], [width-200, 0+200], [width, height], [0,
height+520]]
perspective_img = tvt.functional.perspective(oblique, startpoints,
endpoints, interpolation=3)
```

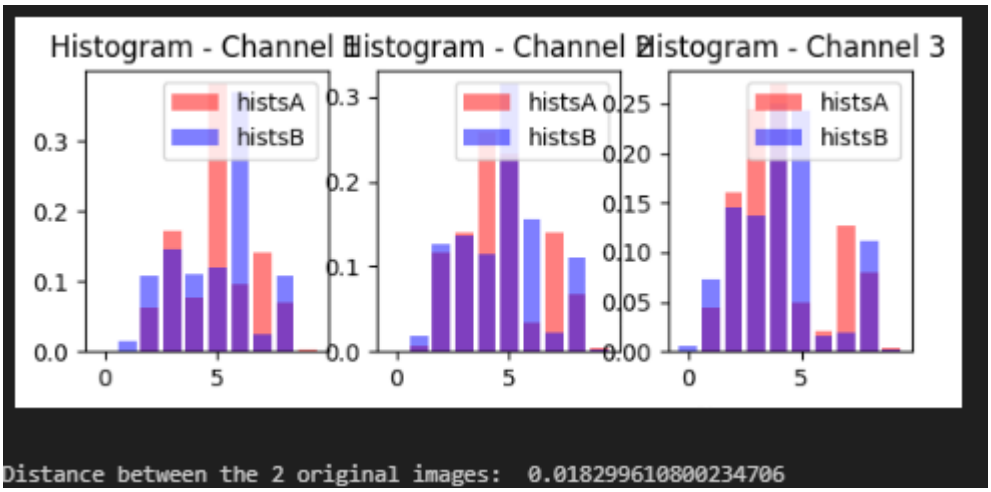
The code used is the following:

```python
from scipy.stats import wasserstein_distance
import matplotlib.pyplot as plt


# Compute the similarity of the histograms of 2 images and plot the histograms
def compute_similarity(A, B):
    num_bins = 10

    A_to_tensor = tvt.ToTensor()(A)
    B_to_tensor = tvt.ToTensor()(B)

    color_channels_A = [A_to_tensor[ch] for ch in range(3)]
    color_channels_B = [B_to_tensor[ch] for ch in range(3)]
    histsA = [torch.histc(color_channels_A[ch],bins=num_bins) for ch in range(3)]
    histsA = [histsA[ch].div(histsA[ch].sum()) for ch in range(3)]
    histsB = [torch.histc(color_channels_B[ch],bins=num_bins) for ch in range(3)]
    histsB = [histsB[ch].div(histsB[ch].sum()) for ch in range(3)]

    dist_r = wasserstein_distance( histsA[0].cpu().numpy(), histsB[0].cpu().numpy() )
    dist_g = wasserstein_distance( histsA[1].cpu().numpy(), histsB[1].cpu().numpy() )
    dist_b = wasserstein_distance( histsA[2].cpu().numpy(), histsB[2].cpu().numpy() )

    dist = (dist_r + dist_g + dist_b)/3

    # Plotting histsA and histsB with titles
    for ch in range(3):
        plt.subplot(2, 3, ch + 1)
        plt.bar(range(num_bins), histsA[ch].cpu().numpy(), alpha=0.5, color='r', label='histsA')
        plt.bar(range(num_bins), histsB[ch].cpu().numpy(), alpha=0.5, color='b', label='histsB')
        plt.title(f'Histogram - Channel {ch + 1}')
        plt.legend()

    plt.show()

    return dist

width, height = front.size
startpoints = [[0,0], [width, 0], [width, height], [0, height]]
#Set the points of the transformation
endpoints = [[0-200,0], [width-200, 0+200], [width, height], [0, height+520]]
perspective_img = tvt.functional.perspective(oblique, startpoints, endpoints, interpolation=3)

display(perspective_img)
perspective_img.save(f"new.jpg")

dist  = compute_similarity(front, oblique)
print("Distance between the 2 original images: ", dist)
dist  = compute_similarity(front, perspective_img)
print("Distance between the front image and the transformed oblique image", dist)
```

Comparing the histograms (Red, Green, Blue) of the 2 original images (Fig 1 and Fig 2) and computing the Wasserstein distance we obtain the following result:

histsA = Front Image

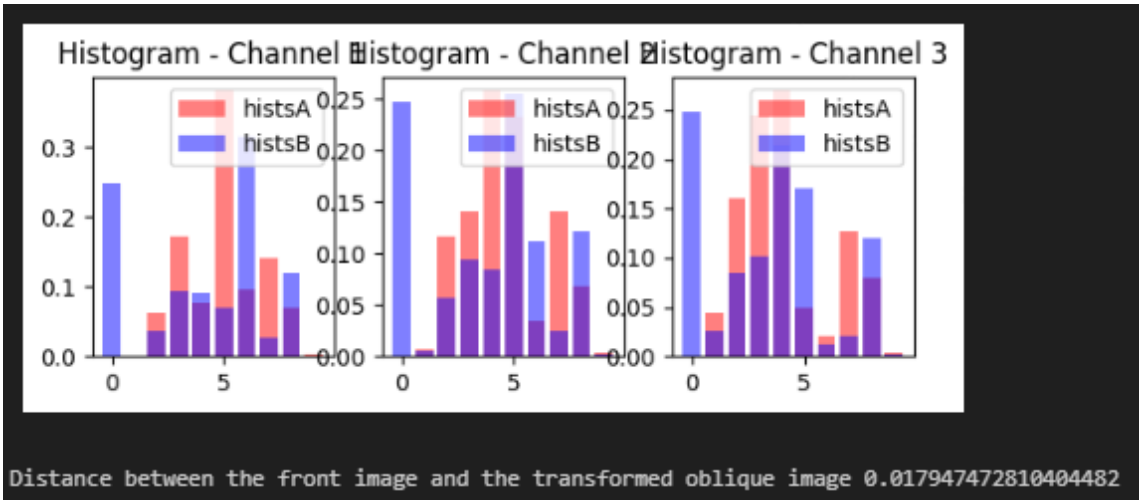
histsB = Oblique Image



Comparing the histograms (Red, Green, Blue) of the original front image and the transformed oblique image (Fig 3 and Fig 4) and computing the Wasserstein distance we obtain the following result:

histsA = Front Image

histsB = Transformed Oblique Image



We can see that by applying the transformation, the distance between both images has been reduced from 0.01829 to 0.01794

3.3. Creating Your Own Dataset Class

```python
class MyDataset ( torch.utils.data.Dataset ):

    def __init__ ( self , root ):
        super().__init__()
        # Obtain meta information (e.g. list of file names )
        self.root = root
        self.filenames = os.listdir(self.root)

        # Initialize data augmentation transforms , etc.
        self.transform = tvt.Compose([
            tvt.ToTensor(),
            tvt.Normalize([0], [1]),
            tvt.RandomRotation(degrees = 45),
            tvt.RandomHorizontalFlip( p = 0.5),
            tvt.RandomVerticalFlip( p = 0.5),
            tvt.ColorJitter(brightness=.4, hue=.2)
        ])

    def __len__ ( self ):
        # Return the total number of images
        # the number is a place holder only
        return len(self.filenames)

    def __getitem__ ( self , index ):
        # Read an image at index and perform augmentations
        path = os.path.join(self.root, self.filenames[index])
        img = Image.open(path)

        #Apply augmentation
        transformed_img = self.transform(img)

        # Return the tuple : ( augmented tensor , integer label )
        return transformed_img, random.randint(0, 10)
```

We have applied the following transforms:

- ToTensor() and Normalize([0],[1]): to convert the PIL image to a tensor and normalize the images between -1 and 1. We do this since Neural Networks like seeing the input data in this format.
- RandomRotation(), RandomHorizontalFlip() and tvt.RandomVerticalFlip(): to provided invariance to different rotations and pictures taken from different angles.
- ColorJitter(): to provided invariance to different lighting conditions.

We run the code provided by the assignment to check that the code works correctly:

```
# Based on the previous minimal example
my_dataset = MyDataset ("bottle_images")
print(len( my_dataset ) )
index = 6
print( my_dataset[ index ][0].shape, my_dataset[ index ][1])

index = 8
print( my_dataset[ index ][0].shape, my_dataset[ index ][1])
```

✓ 2.1s

```
10
torch.Size([3, 1600, 1200]) 5
torch.Size([3, 1600, 1200]) 1
```

Examples of images that have been modified as part of the data augmentation process:

Fig 5: Original Image 1

Fig 6: Original Image 2

Fig 7: Original Image 3

Fig 8: Augmented Image 1

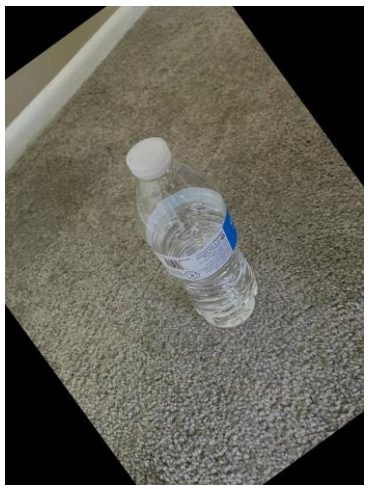


Fig 9: Augmented Image 2

Fig 10: Augmented Image 3

3.4. Generating Data in Parallel

```python
from torch.utils.data import DataLoader

my_dataset = MyDataset ("bottle_images")

batch_size = 4
dataloader = DataLoader(my_dataset, batch_size = batch_size, shuffle = True )

# Plot the first batch of images
for batch in dataloader :
    images , labels = batch
    for image in images:
        display(tvt.ToPILImage()(image))
```

4 images from the same batch as returned by my dataloader:




Fig 11: Batch Image 1 Fig 12: Image 2 Fig 13: Image 3 Fig 14: Batch Image 4

Comparison of the performance gain by using the multi-threaded DataLoader v.s. just using Dataset:

I. Calculating how long it takes to load and augment 1000 random images in your dataset (with replacement) by calling my_dataset.__getitem__ 1000 times:

```python
import time

my_dataset = MyDataset ("bottle_images")

# By just using Dataset
start_time = time.time()
for _ in range(1000):
    index = random.randint(0, 9)
    img_tensor = my_dataset[ index ][0]

end_time = time.time()
elapsed_time = end_time - start_time
print(f"By just using Dataset: {elapsed_time} seconds")
```

```
✓ 7m 3.5s
By just using Dataset: 423.51303124427795 seconds
```

We can see that it took 423 seconds.

II. Calculating the time needed by dataloader to process 1000 random images.

a. To do it, we first create a dataset of 1000 images. These images are just copies of the 10 original images of bottles used before.
b. Once we have the new directory with 1000 images, we can record the time:

```
# By using Dataloader
batch_size = 4
num_workers = 4

my_1000_dataset = MyDataset ("1000_bottle_images")

dataloader = DataLoader(my_1000_dataset, batch_size = batch_size, shuffle = True, num_workers=num_workers)
start_time_2 = time.time()
it = 0
for batch in dataloader :
    images , labels = batch
    it += 1
    if it >= (1000/batch_size):
        end_time_2 = time.time()

elapsed_time_2 = end_time_2 - start_time_2
print(f"By using Dataloader: {elapsed_time_2} seconds")
```

✓ 4m 47.9s

```
By using Dataloader: 287.22305822372437 seconds
```

The results obtained by using different values for “batch_size” and “num_workers” are the following:

	Num_workers = 2	Num_workers = 4
Batch_size = 4	418 seconds	287 seconds
Batch_size = 8	407seconds	273 seconds

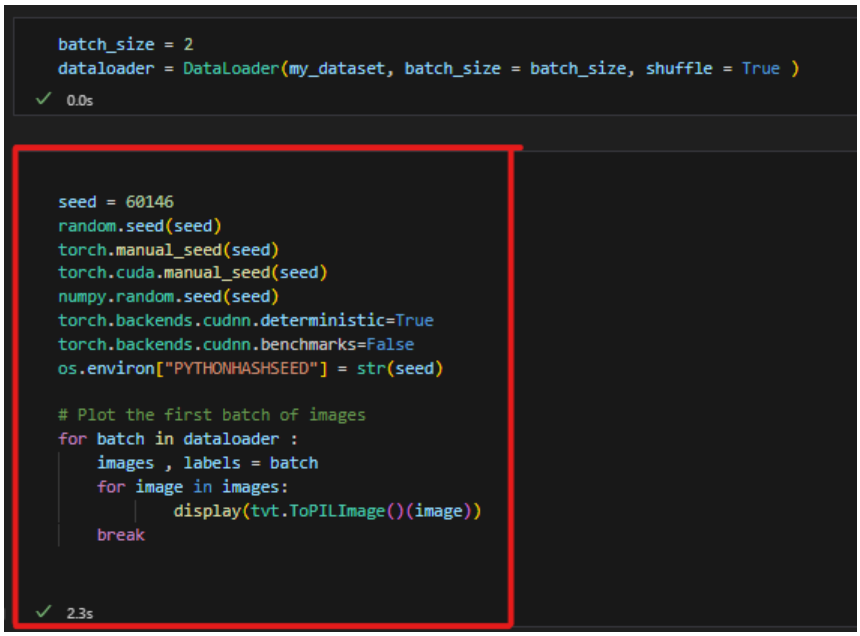
We can see that using a Dataloader the images are processed faster. Furthermore, we can see that by increasing the batch_size and num_workers the Dataloader can process the images faster.

3.5. Random Seed

```
batch_size = 2
dataloader = DataLoader(my_dataset, batch_size = batch_size, shuffle = True )
```

```
# Plot the first batch of images
for batch in dataloader :
    images , labels = batch
    for image in images:
        display(tvt.ToPILImage()(image))
    break
```

When running the red cell for the second time, the two displayed images are different from the images displayed for the first time. This is because, since I'm using "shuffle=True" and there is no seed set, every time I iterate over the dataloader the order is randomly set.
Nevertheless, when setting a seed this behavior changes:



```
batch_size = 2
dataloader = DataLoader(my_dataset, batch_size = batch_size, shuffle = True )
```

✓ 0.0s

```
seed = 60146
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ["PYTHONHASHSEED"] = str(seed)

# Plot the first batch of images
for batch in dataloader :
    images , labels = batch
    for image in images:
            display(tvt.ToPILImage()(image))
    break
```

✓ 2.3s

Now, when running the red cell for the second time, the displayed images are the same that were displayed the first time. This is because we have set a seed.