HW 2: Pytorch Image Processing

David E. Farache, Email ID: dfarache@purdue.edu

January 25, 2023

1 Introduction

This homework focuses on the implementation of image processing methods within the PyTorch library. This includes loading in images as distinct data types, methods to transform the images once properly converted, and validation techniques implemented to grant a numeric evaluation.

2 Methodology

For this assignment, 2 stop sign images were taken with intention of transitioning one image to another using random affine and perspective transformation. Whereby Affine homography maintains straight and parallel lines while perspective homography only maintains straight lines. This involved: first loading in an image via PIL library, then transferring that image into a tensor using tvt.ToTensor, and feeding that input into the transformation. This is then evaluated using the wasserstein distance calculation per RGB channel of the images.

For the remainder of the assignment, 10 images were taken for developing a class called my_dataloader. This object would perform transformation upon the images that would then be stored along its index to be called on later and have the ability to enumerate through. This was then fed into the Dataloader function of PyTorch to compare to looping through the data structure via a standard python for a loop. This expectation is that using DataLoader would outperform as it would parallelize processing and run through faster language, C.

3 Task 1

Question: Why does normalizing the RGB channels of an image in the batch with the max value of all channels, grant the same result as normalizing with the max value of each channel image?

Answer: The reason is due to the selection of method, that being tvt.ToTensor will constantly scale with max value of RGB, 255, leading each color panel to have the same result when normalizing. The reason for max value of 255 is that all RGB pixel values are within a range [0, 255]. The reason why the sample had the same result was that the max value of each individual color panel had the same max value as that of the entire set but only one max was reported when using the max() function.

4 Task 2

4.1 Image Processing and Normalization

For processing, the images are first transferred from a jpeg image into a PIL file and then converted into a normalized PyTorch tensor between 0 and 1.

```
# Import Images
original_file='/Users/davidfarache/Documents/ECE60146/HW2/StopSignImages/IMG_0172.jpg'
target_file='/Users/davidfarache/Documents/ECE60146/HW2/StopSignImages/IMG_0173.jpg'

# Load images as PIL
original_pil = Image.open(original_file) # load image as PIL
target_pil = Image.open(target_file) # load image as PIL

# Convert to tensor and normalize
original_to_tensor = tvt.ToTensor()(original_pil) ## for conversion to [0,1] range
target_to_tensor = tvt.ToTensor()(target_pil) ## for conversion to [0,1] range
```

Listing 1: *Import and transfer PIL to images to normalized PyTorch tensors*

The 2 stop sign images can be seen below with the left being a straight angle that is trying to be converted to the oblique angle image.

(a) *Straight angle* **(b)** *Oblique angle*

Figure 1: *Original and target stop sign images*

The initial step in the transformation process is understanding the extent of the difference between the distinct angles using the Wasserstein distance. The code performs this action by in-taking two Pytorch tensors, converting them to histograms, and applying the wasserstein_distance function from scipy as seen below:

```
def computeDistance(original_to_tensor, target_to_tensor, num_bins):
```

```
    #create empty tensor to store new RGB normalized data
    histTensorA = torch.zeros( target_to_tensor.shape[0], num_bins, dtype=torch.float )
    histTensorB = torch.zeros( target_to_tensor.shape[0], num_bins, dtype=torch.float )

    # Convert to histogram and normalize
    histsA = [torch.histc(original_to_tensor[ch],bins=num_bins,min=-3.0,max=3.0) for ch in range(3)]
    histsA = [histsA[ch].div(histsA[ch].sum()) for ch in range(3)] ## (13)
    histsB = [torch.histc(target_to_tensor[ch],bins=num_bins,min=-3.0,max=3.0) for ch in range(3)]
    histsB = [histsB[ch].div(histsB[ch].sum()) for ch in range(3)] ## (15)

    # Save into array with idx batch for processing distance
    for ch in range(3):
        histTensorA[ch] = histsA[ch]
        histTensorB[ch] = histsB[ch]

    # Store distance information
    BatchImageRGBDistance = []
    for ch in range(3):
        dist = wasserstein_distance( torch.squeeze( histTensorA[ch] ).cpu().numpy(),
        torch.squeeze( histTensorB[ch] ).cpu().numpy() )
        #print("\n Wasserstein distance for channel %d: " % ch, dist)
        BatchImageRGBDistance.append([ch, dist])

    return BatchImageRGBDistance #return distance RGB

batchRGB = computeDistance(original_to_tensor, target_to_tensor, 10)

print('Red: ' + str(batchRGB[0][1]))
print('Green: ' + str(batchRGB[1][1]))
print('Blue: ' + str(batchRGB[2][1]))
```

Listing 2: *Convert tensor to histogram and then return wasserstein distance between RGB values*

The output received from the code for the distances of the channels red, blue, and green channels of the straight angle to the oblique angle is shown below:

Figure 2: *RGB Distance results*

```
Red: 0.0130065083503723 11
Green: 0.018287110328674312
Blue: 0.01489518582820892
```

4.2 Random Affine Transformation

The random-affine transformation was applied on image 1 (a) to appear as image 1 (b) using the PyTorch method tvt.RandomAffine.

```
#Perform affine transformation
def randAffineTransform(image_to_tensor, degree, trans, scale, shear):
    randAffine = tvt.RandomAffine(degree, translate=trans, scale=scale, shear=
    shear)
    transformImage = randAffine(image_to_tensor)

    img = tvt.ToPILImage()(transformImage)
    img.show()

    return transformImage
```

Listing 3: *Code to encat RandomAffine transformation upon tensor*

This function intakes different parameters in order to optimize the change. The code will loop through the parameter within a pre-designated range for scale, translation, degree, and shear. The selected values are those with the lowest sum distance of the RGB channels of the image.

```
# Loop for best affine parameters
degParam = [0, 0]
transParam = [1, 1]
scaleParam = [1, 1]
shearParam = [0, 0]
total = 10 ** 12

#Solve for best scale using distance calc
for scale2 in range(1, 10, 1):
    for scale1 in range(scale2, 10, 1):
        transformFig = randAffineTransform(original_to_tensor, (degParam[1],
    degParam[0]), (transParam[1], transParam[0]), (scale2/10, scale1/10), (
    shearParam[1], shearParam[0]))
        val = computeDistance(transformFig, target_to_tensor, 10)
        if float(val[0][1] + val[1][1] + val[2][1]) < total:
            total =  float(val[0][1] + val[1][1] + val[2][1])
            scaleParam[0] = scale1/10
            scaleParam[1] = scale2/10

print(scaleParam)
img = tvt.ToPILImage()(transformFig)
img.show()

#Solve for best translation using distance calc
for trans2 in range(0, 10, 1):
    for trans1 in range(trans2+1, 10, 1):
        transformFig = randAffineTransform(original_to_tensor, (degParam[1],
    degParam[0]), (trans2/10, trans1/10), (scaleParam[1], scaleParam[0]), (
    shearParam[1], shearParam[0]))
        val = computeDistance(transformFig, target_to_tensor, 10)
        if float(val[0][1] + val[1][1] + val[2][1]) < total:
            total =  float(val[0][1] + val[1][1] + val[2][1])
            transParam[0] = trans1 / 10
            transParam[1] = trans2 / 10

print(transParam)
img = tvt.ToPILImage()(transformFig)
img.show()

#Solve for best degree using distance calc
for shear2 in range(0,720, 10):
```

```
    for shear1 in range(shear2,720, 10):
        transformFig = randAffineTransform(original_to_tensor, (degParam[1],
   degParam[0]), (transParam[1], transParam[0]), (scaleParam[1], scaleParam[0]),
    (shear2, shear1))
        val = computeDistance(transformFig, target_to_tensor, 10)
        if float(val[0][1] + val[1][1] + val[2][1]) < total:
            total =  float(val[0][1] + val[1][1] + val[2][1])
            shearParam[0] = shear1
            shearParam[1] = shear2

print(shearParam)
img = tvt.ToPILImage()(transformFig)
img.show()

#Solve for best degree using distance calc
for deg2 in range(0, 360*4, 10):
    for deg1 in range(deg2, 360*4, 10):
        transformFig = randAffineTransform(original_to_tensor, (deg2, deg1), (
   transParam[1], transParam[0]), (scaleParam[1], scaleParam[0]), (shearParam
   [1], shearParam[0]))
        val = computeDistance(transformFig, target_to_tensor, 10)
        if float(val[0][1] + val[1][1] + val[2][1]) < total:
            total =  float(val[0][1] + val[1][1] + val[2][1])
            degParam[0] = deg1
            degParam[1] = deg2

print(degParam)
img = tvt.ToPILImage()(transformFig)
img.show()
```

Listing 4: *Code to loop through parameters for RandomAffine transformation upon tensor*

Below is the final image produced via transformation with the original on the left, the transformed image in the center, and the target on the right with parameters in accordance to a previous order (scale = [0.8, 0.8], translate = [0, 0.1], degree = [350, 1160], shear = [80, 550]):

(a) *Straight picture*

(b) *Transformed Image*

(c) *Oblique angle*

Figure 3: *Apply random-affine transformation from original to target transformation*

4.3 Perspective Transformation

For perspective transformation, the perspectiveTransformeImage was used. This transformation takes in points of comparison from the original to the target image which were found using pisxqy which can be seen below in the code.

```
# Perspective Transformation
endpoints = [[142, 132], [140, 225], [325, 57], [325, 169]]
startpoints = [[110, 120], [105, 225], [268, 125], [369, 231]]

perspectiveTransformeImage = tvt.functional.perspective(original_to_tensor,
    startpoints=startpoints, endpoints=endpoints, interpolation=tvt.
    InterpolationMode.BILINEAR)
# hist_perspective_transformed = create_histogram(perspective_transformed_image,
     bins=10)

tvt.ToPILImage()(perspective_transformed_image).show()
```

Listing 5: *Apply perspective transformation from original to target transformation*

Below is the final image produced via transformation with the original on the left, the transformed image in the center, and the target on the right:

(a) *Straight angle* **(b)** *Transformed Image* **(c)** *Oblique angle*

Figure 4: *Perspective transformation with original and target images*

5 Task 3

For this assignment, a data-set of 10 images was required for processing, which was taken on an iPhone camera and converted to jpegs, as to reduce processing time.

```
import os
import matplotlib.pyplot as plt

# Load files and retirn filename and image
def loadFile(path):
    image = []
    fileName = []
    os.chdir(path)
    for file in os.listdir():
        if '.jpg' in file:
```

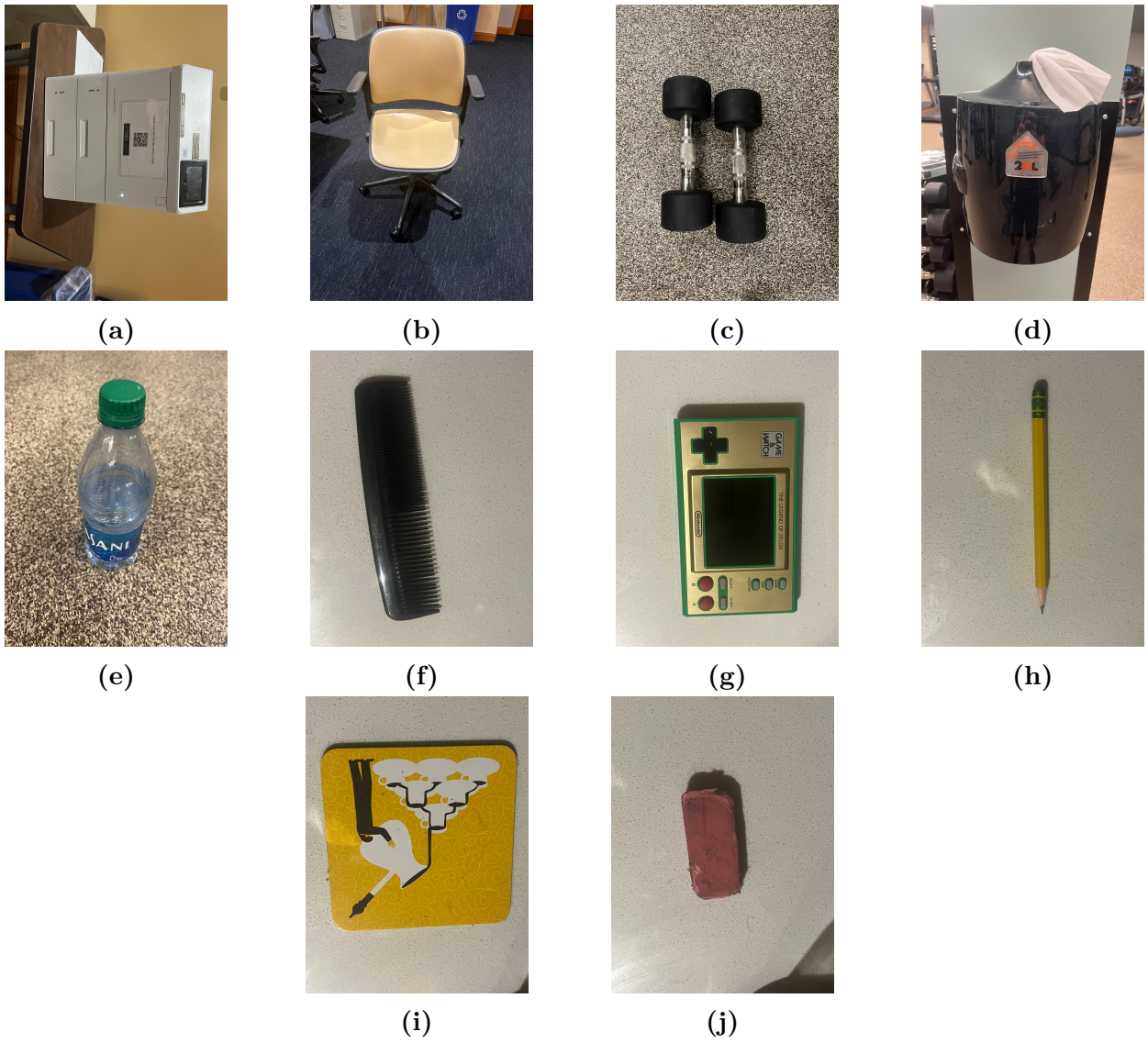


Figure 5: *MyDataset Pictures Taken*

```
            fileName.append(file)
            image.append(Image.open(file))

    return fileName, image
```

The code utilizes the function above to process the images based on a directory given, converting to PIL images and collecting file names. The object below will then convert the PIL to Pytorch tensor and apply pre-decided transformation in _getitem_. The chosen transformations were: tvt.RandomAffine to readjust the scale to better fit the main object of the image, tvt.transforms.CenterCrop(1500) to remove the noise around the desired object in the image, and tvt.ColorJitter plays with brightness, contrast, and saturation to better exaggerate the image.

```
import torch
class MyDataset(torch.utils.data.Dataset):
    # Obtain meta information (e.g. list of file names) # Initialize data
   augmentation transforms , etc. pass

    def __init__(self, root):
```

```
        super().__init__()
        self.root = root
        self.transforms = tvt.Compose([ tvt.RandomAffine(0, scale=(0.7, 1)),
                                        tvt.ColorJitter(brightness=(0.5, 1),
    contrast=(0.5, 1), saturation=(0.5, 1)),
                                        tvt.transforms.CenterCrop(1500),
                                        tvt.ToTensor()])
        self.fileName, self.im = loadFile(root)

    def __len__(self):
        return len(self.im)
    # Return the total number of images return 100

    def __getitem__(self, index):
        images = self.transforms(self.im[index])
        classLabel = index+1
        return images, classLabel

    # Read an image at index and perform augmentations
    # Return the tuple: (augmented tensor, integer label) return torch.rand((3,
    256, 256)), random.randint(0, 10)
```

```
# Based on the previous minimal example
my_dataset = MyDataset('/Users/davidfarache/Documents/ECE60146/HW2/
    MyDatasetFolder/')
print(len(my_dataset)) # 10
index = 1
print(my_dataset[index][0].shape, my_dataset[index][1]) # torch.Size([3, 256,
    256]) 6
index = 5
print(my_dataset[index][0].shape, my_dataset[index][1]) # torch.Size([3, 256,
    256]) 8
```

The picture was taken from the code to prove the result desired was output-ed.

Figure 6: *Ojbect, data-strcture desired output*

```
10
torch.Size([3, 1500, 1500]) 2
torch.Size([3, 1500, 1500]) 6
```

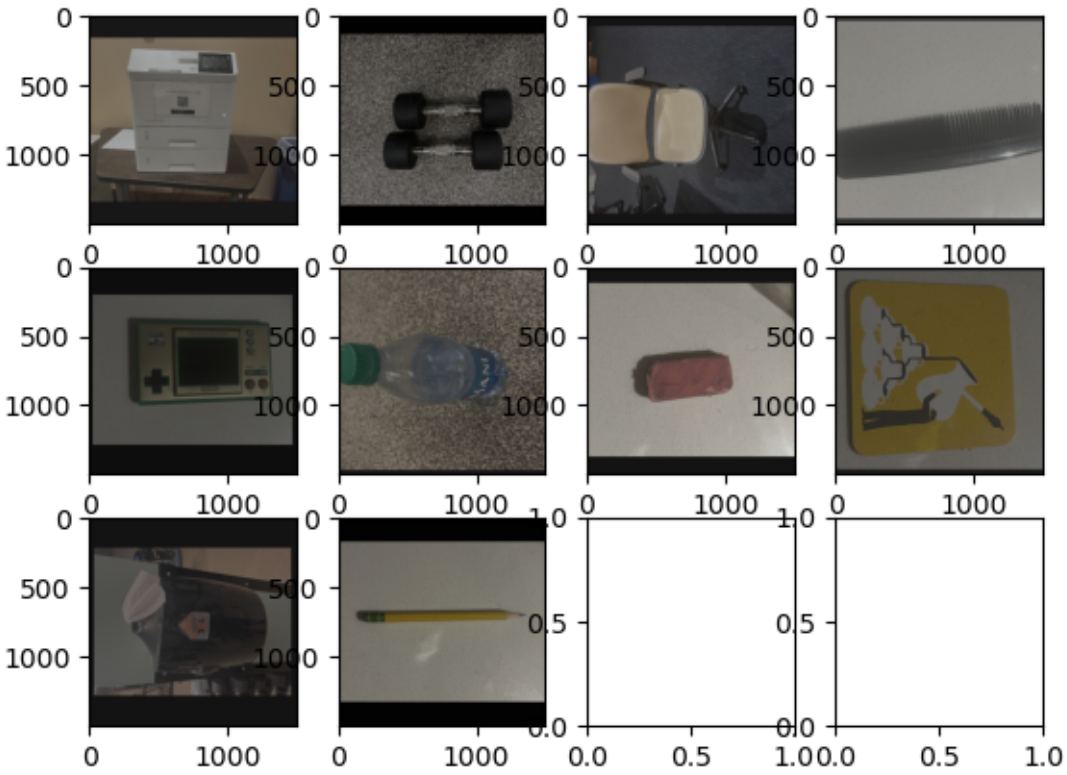
6 Task 4

The code below accesses the object of my_dataset to get image information as a tensor and the file name. The transformations applied by the _get_item are displayed below.

```
import time

dataloader = torch.utils.data.DataLoader(my_dataset, batch_size=4)

fig, ax = plt.subplots(3, 4)
```

```
for batch_index, (images, labels) in enumerate(dataloader):
    for i in range(len(labels)):
        image = tvt.ToPILImage()(images[i])
        ax[batch_index, i].imshow(image)

        fileName = str(labels[i])
        image.save("/Users/davidfarache/Documents/ECE60146/HW2/
    TrasnformedMyDataset/"+fileName)
```

Figure 7: *Transformed images using the MyDataset object*



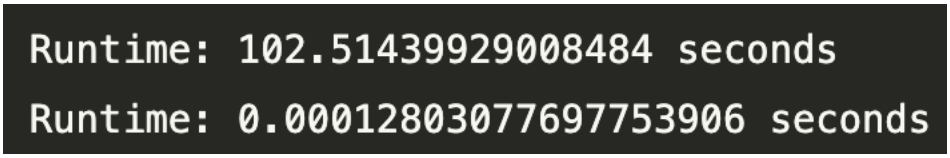
The following code is the comparison of data processing via a python for loop and using the DataLoader function of PyTorch which was timed via the time() function. What can be seen is that the DataLoader function is significantly faster than looping through the dataset and saving the information.

```
start_time = time.time()

for i in range(1000):
    num = random.randint(0, 9)
    augmented_image, label = my_dataset[num][0], my_dataset[num][1]

print("Runtime: %s seconds" % (time.time() - start_time))

start_time = time.time()
my_dataloader = torch.utils.data.DataLoader(my_dataset, batch_size=1000, shuffle
    =True, num_workers=4)

print("Runtime: %s seconds" % (time.time() - start_time))
```

Listing 6: *Timing of looping versus using the DataLoader*

The picture was taken from the code to prove the result desired was output-ed.

Figure 8: *Timing of loop vs DataLoader*



7 Lessons learned

This assignment was an introduction to computer vision where transferring of data types was shown, the use of transformers was learned, and the application of DataLoader and the speed optimization it provides was made clear.