ECE 60146 Deep Learning
Homework 2
01/25/2023

Name: *Nikita Ravi (ravi30)*

1 Introduction

The purpose of this homework assignment is to understand the different transformation functions available to alter and modify the images so that it could be a better input to neural networks. In addition, through this assignment we gained experience in creating a dataset class and dataloaders which are essential in preparing the data needed for the neural network to a tensor format and process them in parallel.

2 Understanding Data Normalization

The mystery question is: How are the results the same when conducting the pixel-value scaling on a per-image basis or per-batch basis?

Answer: After reading the source code for tvt.ToTensor() [1], I understood why the results of pixel-value scaling per-image basis is the same as per-batch basis. The reason is because tvt.ToTensor() converts the entire image array to a tensor type and then divides the values in the tensor by 255. On the other hand, in the per-batch approach, the max value that the entire batch of image arrays is divided by will always be 255. This is because the pixels of an image will always be in the range [0, 255].

3 Task 1: Transformation of Images

Before passing the images into the neural network as an input, it is vital we transform the images so that the potential distortion (i.e. lighting, angles, rotation) in the images can be resolved and the model can analyze it better. The two transformations used for this assignment are perspective and random affine transformations. The inputs images used are as follows:

```
def get_images(path, resize=True, rotate=True):
    image_files = [file for file in os.listdir(path) if file != ".DS_Store
   "]
    image_files = [img for img in sorted(image_files, key=lambda x: int(re
   .sub(r'\D', '', x)))]

    images_pil = [Image.open(path + img) for img in image_files]

    if(resize):
        new_size = (300, 300)
        images_pil = [image.resize(new_size) for image in images_pil]
    images_pil_rotated = [image.rotate(270) for image in images_pil]

    return image_files, images_pil_rotated
```

(a) Original

(b) Oblique Effect

Figure 1: Task 1 Inputs

```
13
14 def transform_tensor_to_PIL(tensor_image):
15     transform = tvt.ToPILImage()
16     img = transform(tensor_image)
17
18     return img
19
20 def transform_PIL_to_tensor(image_pil):
21     transform = tvt.ToTensor()
22     img = transform(image_pil)
23
24     return img
25
26 def create_histogram(tensor_image, bins=10):
27     hist = list(map(lambda c: torch.histc(tensor_image[c,:,:], bins=bins,
    min=-1, max=1), list(range(3))))
28     return hist
29
30 def compute_distance(hist1, hist2):
31     distance_per_channel = list(map(lambda c: wasserstein_distance(torch.
    squeeze(hist1[c]).cpu().numpy(), torch.squeeze(hist2[c]).cpu().numpy())
    , list(range(3))))
32     return distance_per_channel
33
34 _, images_pil = get_images("./Task 1 Images/", resize=False)
35
36 image_tensor_original = transform_PIL_to_tensor(images_pil[0])
37 hist_original = create_histogram(image_tensor_original, bins=10)
38
39 image_tensor_oblique = transform_PIL_to_tensor(images_pil[1])
40 hist_oblique = create_histogram(image_tensor_oblique, bins=10)
41
42 # Compute the distance between hist_oblique and hist_original
```

```
distance_between_original_oblique = compute_distance(hist_original,
    hist_oblique)
print(distance_between_original_oblique)
print(sum(distance_between_original_oblique))
```

Listing 1: Preparations for Applying Transformations

The Wasserstein distance in RGB between the original image and the oblique effect image is as follows

R	G	B
71635.99999999999	101849.60000000002	28705.800000000007

Table 1: Wasserstein distance between the original image and image with the oblique effect

```
_, images_pil = get_images("./Task 1 Images/", resize=False)

image_tensor_original = transform_PIL_to_tensor(images_pil[0])
hist_original = create_histogram(image_tensor_original, bins=10)

image_tensor_oblique = transform_PIL_to_tensor(images_pil[1])
hist_oblique = create_histogram(image_tensor_oblique, bins=10)

# Compute the distance between hist_oblique and hist_original
distance_between_original_oblique = compute_distance(hist_original, hist_oblique)
print(distance_between_original_oblique)
print(sum(distance_between_original_oblique))
```

```
[71635.99999999999, 101849.60000000002, 28705.800000000007]
202191.40000000002
```

Figure 2: Distance between Original and Oblique

3.1 Perspective Transformation

The perspective transformation is a type of transformation that warps one image into the space of another image by manually identifying the starting and ending points. The transformation was applied on the original image to transform it into the image with oblique effect.

```
# Perspective Transformation
endpoints = [[785, 947], [2033, 916], [2032, 1440], [789, 1474]]
startpoints = [[1530, 735], [2710, 912], [2675, 1432], [1528, 1318]]

perspective_transformed_image = tvt.functional.perspective(
    image_tensor_original, startpoints=startpoints, endpoints=endpoints,
    interpolation=tvt.InterpolationMode.BILINEAR)
hist_perspective_transformed = create_histogram(
    perspective_transformed_image, bins=10)
```

```
7
8 transform_tensor_to_PIL(perspective_transformed_image).save("./Task 1
    Results/perspective.jpg")
9
10 # Compute the distance between hist_oblique and
    hist_perspective_transformed
11 distance_between_transformed_oblique = compute_distance(
    hist_perspective_transformed, hist_oblique)
12 print(distance_between_transformed_oblique)
13 print(sum(distance_between_transformed_oblique))
```

Listing 2: Perspective Transformation

The result of the transformation is shown in figure 3

(a) Original (b) Perspective Transformed (c) Oblique Effect

Figure 3: Result of Perspective Transformation

The Wasserstein distance in RGB between the perspective transformed image to the image with the oblique effect is as follows:

R	G	B
63273.399999999994	98036.00000000003	46156.600000000006

Table 2: Wasserstein distance between the perspective transformed image and image with the oblique effect

```
# Perspective Transformation
endpoints = [[785, 947], [2033, 916], [2032, 1440], [789, 1474]]
startpoints = [[1530, 735], [2710, 912], [2675, 1432], [1528, 1318]]

perspective_transformed_image = tvt.functional.perspective(image_tensor_original, startpoints=startpoints, endpoints=endpoints, interpolation=tvt.Ir
hist_perspective_transformed = create_histogram(perspective_transformed_image, bins=10)

transform_tensor_to_PIL(perspective_transformed_image).save("./Task 1 Results/perspective.jpg")

# Compute the distance between hist_oblique and hist_perspective_transformed
distance_between_transformed_oblique = compute_distance(hist_perspective_transformed, hist_oblique)
print(distance_between_transformed_oblique)
print(sum(distance_between_transformed_oblique))
```

Python

```
[63273.399999999994, 98036.00000000003, 46156.600000000006]
207466.00000000003
```

Figure 4: Distance between Perspective and Oblique

3.2 Random Affine Transformation

The random affine transformation is a type of transformation that randomly chooses a parameter within the range provided while maintaining all the paralle lines. The transformation was applied on the original image to transform it into the image with oblique effect.

```python
# Random Affine Transformation
random_affine_transformation = tvt.RandomAffine(degrees=(30, 70), translate=(0.2, 0.8), scale=(0.5, 0.75))

random_affine_transformed_image = random_affine_transformation(image_tensor_original)
hist_affine_transformed = create_histogram(random_affine_transformed_image, bins=10)

transform_tensor_to_PIL(random_affine_transformed_image).save("./Task 1 Results/affine.jpg")

# Compute the distance between hist_oblique and hist_affine_transformed
distance_between_transformed_oblique = compute_distance(hist_affine_transformed, hist_oblique)
print(distance_between_transformed_oblique)
print(sum(distance_between_transformed_oblique))
```

Listing 3: Random Affine

The result of the transformation is shown in figure 5

(a) Original (b) Affine Transformed (c) Oblique Effect

Figure 5: Result of Affine Transformation

The Wasserstein distance in RGB between the affine transformed image to the image with the oblique effect is as follows:

R	G	B
958115.0	909755.0	904064.5999999999

Table 3: Wasserstein distance between the affine transformed image and image with the oblique effect

```python
# Random Affine Transformation
random_affine_transformation = tvt.RandomAffine(degrees=(30, 70), translate=(0.2, 0.8), scale=(0.5, 0.75))

random_affine_transformed_image = random_affine_transformation(image_tensor_original)
hist_affine_transformed = create_histogram(random_affine_transformed_image, bins=10)

transform_tensor_to_PIL(random_affine_transformed_image).save("./Task 1 Results/affine.jpg")

# Compute the distance between hist_oblique and hist_affine_transformed
distance_between_transformed_oblique = compute_distance(hist_affine_transformed, hist_oblique)
print(distance_between_transformed_oblique)
print(sum(distance_between_transformed_oblique))
```

Python

```
[958115.0, 909755.0, 904064.5999999999]
2771934.5999999996
```

Figure 6: Distance between Random Affine and Oblique

4 Task 2: Creating your own Dataset Class

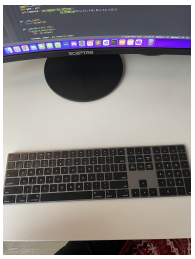
The purpose of this task is to create a custom dataset class for our images. The custom dataset class will store the meta information about the dataset and implement that loads and augments our images.

The input images are shown in 7

(a) Input 1

(b) Input 2

(c) Input 3



(d) Input 4

(e) Input 5

(f) Input 6

(g) Input 7

(h) Input 8

(i) Input 9

(j) Input 10

Figure 7: Task 2 Inputs

The source code below is how we create a dataset class in PyTorch. The transformations used were random affine, random perspective, and color jitter.

```
class MyDataset(torch.utils.data.Dataset):
    def __init__(self, path):
        super().__init__()
        self.path = path
        self.transform = tvt.Compose([tvt.RandomAffine(degrees=(25, 45)),
                                      tvt.RandomPerspective(
   distortion_scale=0.6, p=0.5),
                                      tvt.ColorJitter(saturation=0.8,
   hue=0.2),
                                      tvt.ToTensor()])

        self.filenames, self.images = get_images(self.path)

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

    def __getitem__(self, idx):
        image = self.transform(self.images[idx])
        label = int(self.filenames[idx].strip(".JPG"))

```

```
19          return image, label
```

Listing 4: Dataset Class

Figure 8 illustrate the augmentation of three input images

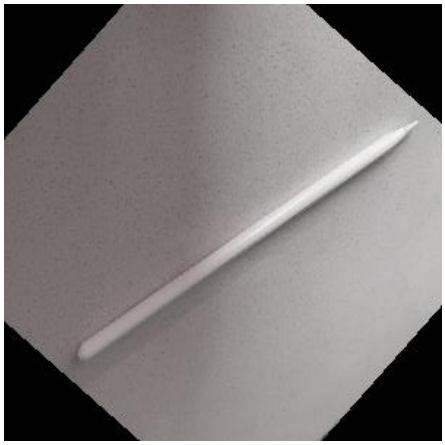
(a) Original Input 1

(b) Augmented Input 1

(c) Original Input 2

(d) Augmented Input 2

(e) Original Input 3



(f) Augmented Input 3

Figure 8: Task 2 Results

5 Task 3: Generating Data in Parallel

Since calling the ___getitem___ function will only return a single training sample, we build a dataloader class which yields a batch of the training samples in a multi-thread fashion.

```
dataloader = torch.utils.data.DataLoader(my_dataset, batch_size=4)
fig, ax = plt.subplots(3, 4)
for batch_idx, (images, labels) in enumerate(dataloader):
    for idx in range(len(labels)):
        image = transform_tensor_to_PIL(images[idx])
        ax[batch_idx, idx].imshow(image)

        label = str(labels[idx])
        image.save("./Task 3 Results/" + label + ".jpg")
```

Listing 5: Dataloader

Figure 9 illustrates the augmented images obtained in batches

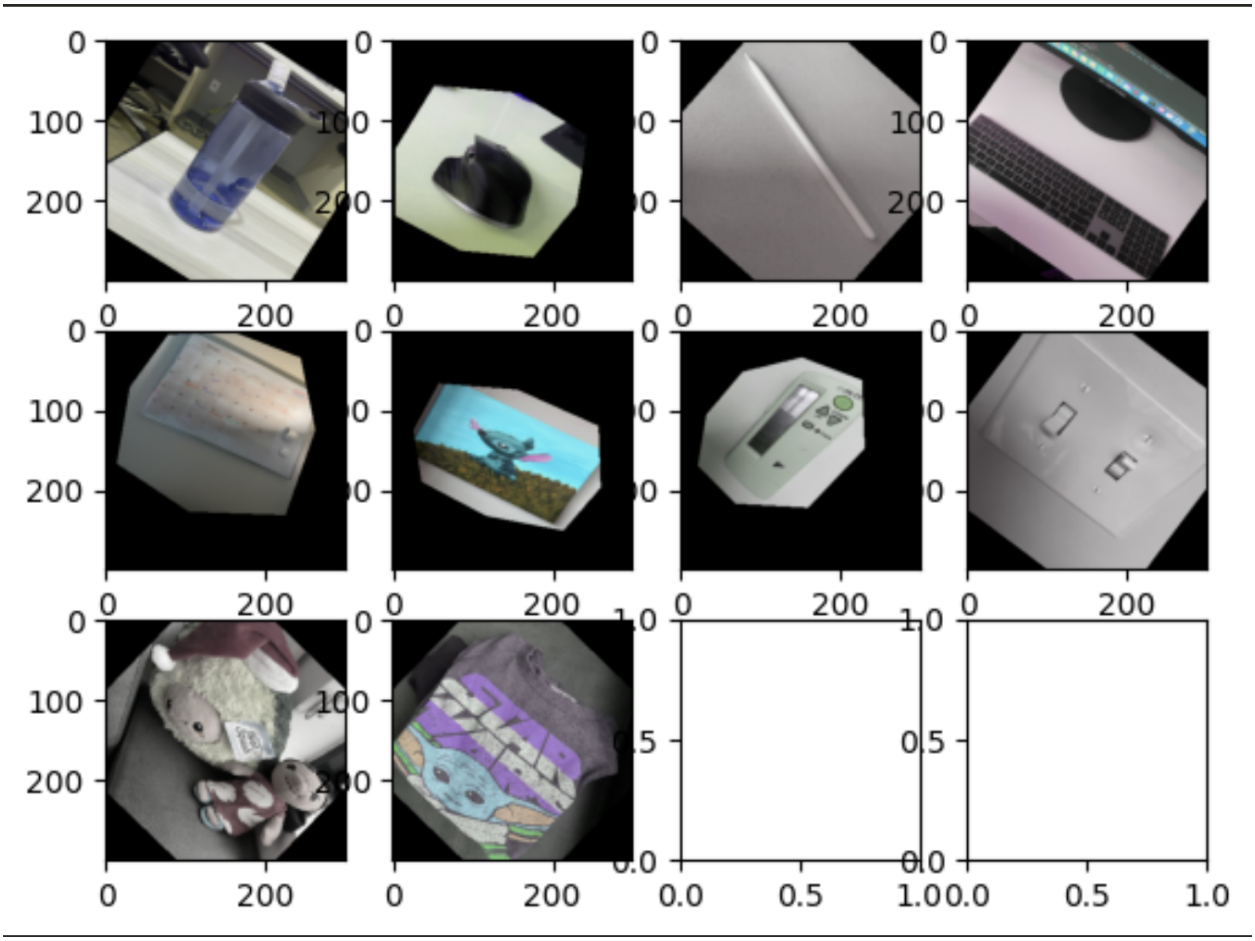


Figure 9: Images obtained in batches

To confirm that the multi-threading dataloader approach is faster than calling the ___getitem___ function of the dataset class, we ran the code in listing 6 and listing 7 and compared the runtime.

```
start_time = time.time()
for idx in range(1000):
    index = random.randint(0, 9)
    augmented_image, label = my_dataset[index][0], my_dataset[index][1]

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

Listing 6: Calling Dataset Class 1000 Times

```
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 7: Dataloader Batch Size of 1000

Approach	Runtime
Dataset Class	18.03 seconds
Dataloader Class	0.00023 seconds

Table 4: Runtime difference between the Dataset and Dataloader Approach

Figure 10: Runtime Difference

From Table 4 and Figure 10, we can clearly see that the dataloader approach is quicker than calling the get_item function multiple times because of its multithreading ability.

References

[1] PyTorch, "Source code for torchvision.transforms", https://pytorch.org/docs/0.2.0/_modules/torch