SPRING 2023 ECE 60146 – Homework 2

Sahithi Kodali – 34789866
kodali1@purdue.edu

THEORY QUESTION

Understanding Data Normalization

In slide 26, the pixel-value scaling is being done by dividing every image in the batch (or can be termed as batch basis) with the maximum pixel 255 which is occurred in another image of the same batch. In spite of the maximum pixel 255 not occurring in the image, since it is batch basis scaling, all the images in the batch are divided by the max value 255 so that the values can be scaled between the range [0,1].

However in slide 28, where the pixel-value scaling is done per-image basis using *tvt.ToTensor*, each image pixel is divided by the maximum pixel value possible i.e. 255 for the 8-bit images. If every image is divided by the maximum pixel value 255, irrespective of the maximum pixel is in the image or not, the scaling is performed by 255 to obtain pixel values in the range [0,1].

Hence, for the above reasons performing pixel-value scaling on a per-image basis in Slide 28 and on a batch basis in Slide 26 would yield exactly the same results.

PROGRAMMING TASKS

1. Setting up Conda environment

The Conda environment *ece60146* is created following the instructions and attached as a *'environment.yml'* file along with this report.

2. Data Augmentation with torchvision.transforms

To perform data augmentation, two images of 'stop sign' - one from the front angle and the other from an oblique angle - are pictured. The similarity between these two images has been measured using histograms with Wasserstein distance, to take this as a standard for further comparison with the augmented image that needs to be similar to the front angled stop image.

Following the requirements and intuition an affine transformation using *tvt.RandomAffine* with a 2 degrees turn has been performed on the image. The image with affine rectification is further rectified using projective transformation by *tvt.perspective* giving the start and end points for transformation. The end points are chosen by trial and error and intuition in a loop and the Histogram distance using Wasserstein is taken to compare the similarity and obtain the best image possible which is similar to the target front angled stop image. The augmented images and the histogram similarities can be seen in the figures 1,2 and 3 respectively.

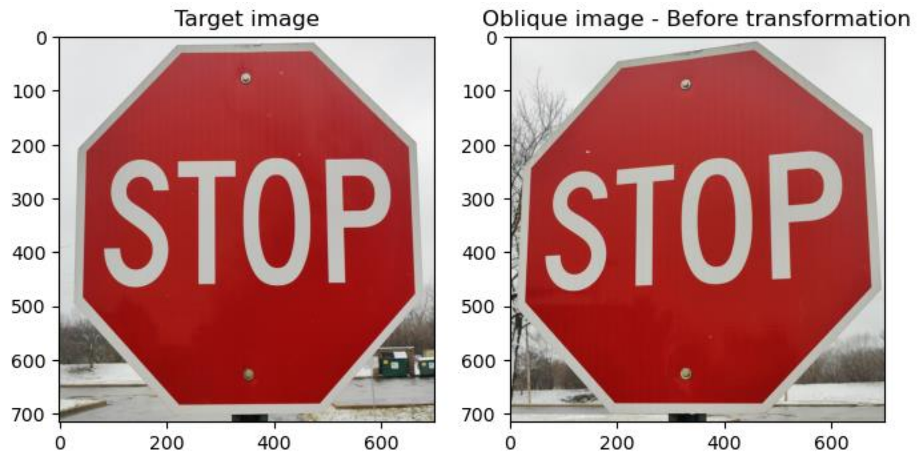


Figure 1: Stop sign images – Before Transformation

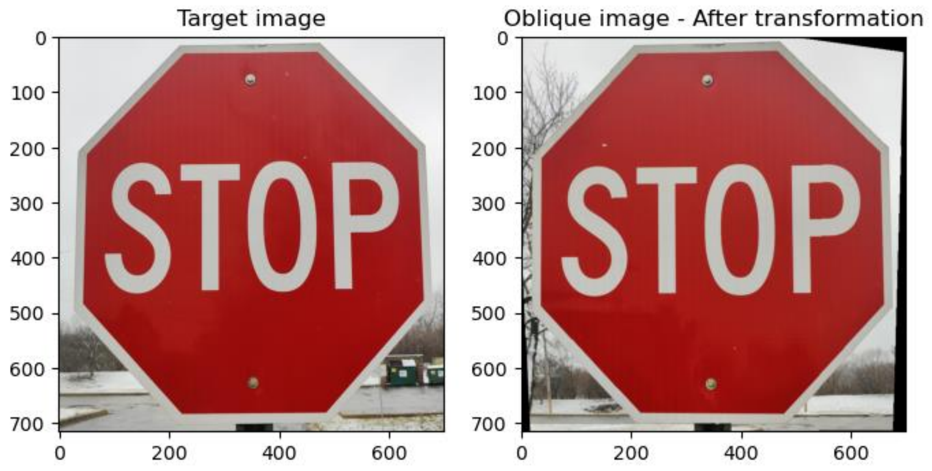
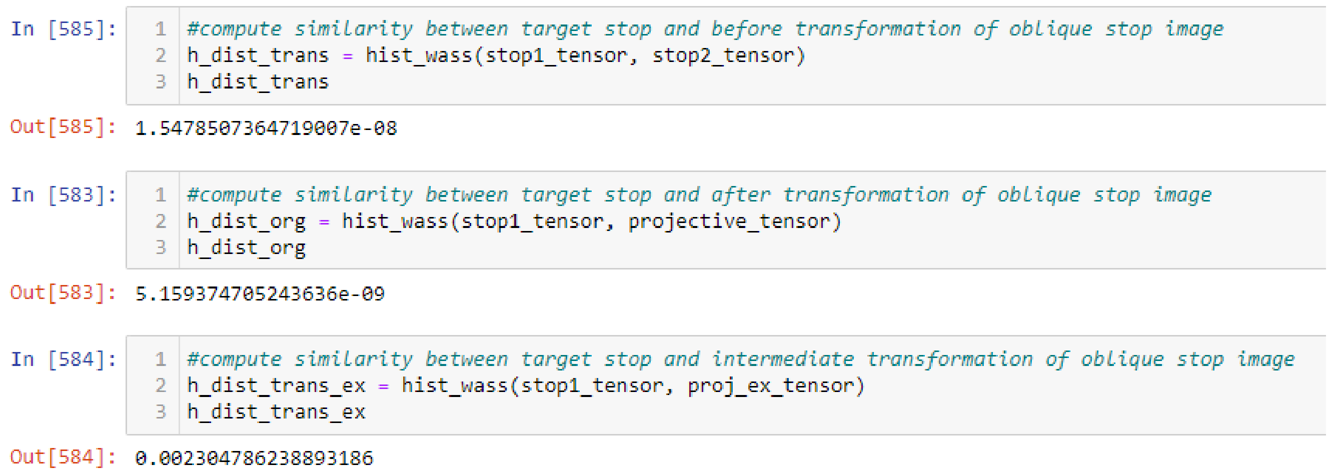


Figure 2: Stop sign images – After Transformation



```
In [585]:
#compute similarity between target stop and before transformation of oblique stop image
h_dist_trans = hist_wass(stop1_tensor, stop2_tensor)
h_dist_trans

Out[585]: 1.5478507364719007e-08
```

```
In [583]:
#compute similarity between target stop and after transformation of oblique stop image
h_dist_org = hist_wass(stop1_tensor, projective_tensor)
h_dist_org

Out[583]: 5.159374705243636e-09
```

```
In [584]:
#compute similarity between target stop and intermediate transformation of oblique stop image
h_dist_trans_ex = hist_wass(stop1_tensor, proj_ex_tensor)
h_dist_trans_ex

Out[584]: 0.002304786238893186
```

Figure 3: Histogram based similarity. The first input block shows the original distance similarity, the second input block shows distance similarity after transformation and third input block shows distance similarity with an intermediate image during transformation.

In figure 3, we can observe the distance to be least (5.159374705243636e-09) after complete transformation of the image when compared to before transformation (1.5478507364719007e-08). Since smaller distance means greater similarity of images, we can say that the transformed image yields much similar image to target image. It can also be observed that one of the intermediate images during transformation yields higher distance which can be expected.

3. Creating Own Dataset Class

The own dataset class with methods *__init__*, *__len__* and *__getitem__* is created. The *__getitem__* method takes an index to call that image indexed in the dataset and perform the three augmentations that are initialized using *tvt.compose* in *__init__*. For image index greater than 10, a mod by 10 is performed to avoid any index error. Figure 4 shows the working of the own dataset class as shown in the snippet given in HW.

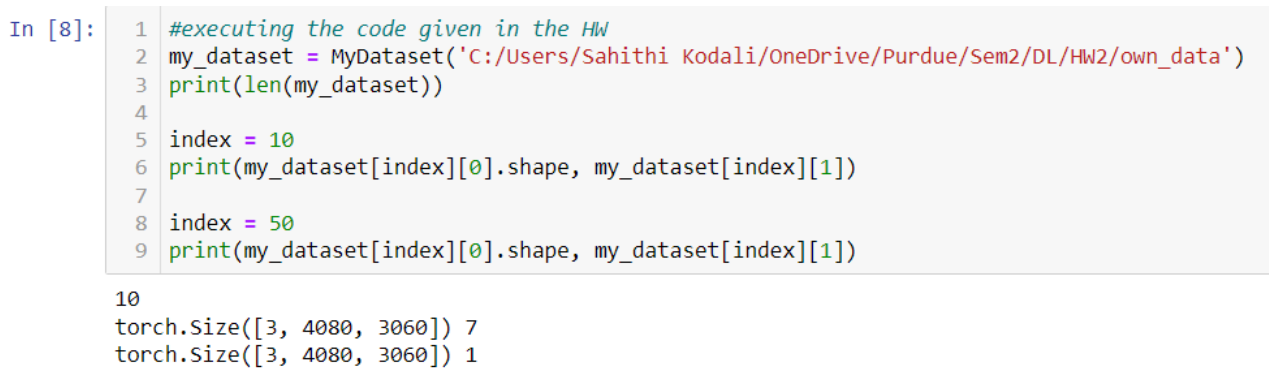
The three augmentation transforms chosen are described below,

i. tvt.RandomRotation(45): Randomly rotates the image by 45 degrees. This is chosen since it is quite possible for an image to look slightly rotated/tilted like the projective distortion w.r.t the object movement.

ii. tvt.ColorJitter(brightness = [0.4,0.8], contrast = [0.4,0.8], saturation = [0.4,0.8], hue = [0.2,0.5]): This augmentation changes the properties of image i.e. brightness, contrast, saturation ad hue in the given range of numbers. This is chosen since based on the illumination and angle there is a high possibility for these factors of the image to look differently w.r.t object movement.

iii. tvt.RandomInvert(p=0.2): This augmentation inverts the color of images by the given probability 0.2. Following the above explanation, in addition to basic properties the colors can look inverted depending on the object movement, external conditions and hence this can be useful.

In figures 5,6,7; random 3 images from a 10 image own dataset are displayed along with their augmented versions after applying the transformations described above.



```
In [8]:
#executing the code given in the HW
my_dataset = MyDataset('C:/Users/Sahithi Kodali/OneDrive/Purdue/Sem2/DL/HW2/own_data')
print(len(my_dataset))

index = 10
print(my_dataset[index][0].shape, my_dataset[index][1])

index = 50
print(my_dataset[index][0].shape, my_dataset[index][1])
```

```
10
torch.Size([3, 4080, 3060]) 7
torch.Size([3, 4080, 3060]) 1
```

Figure 4: Working of the own dataset class – As per given snippet in HW.

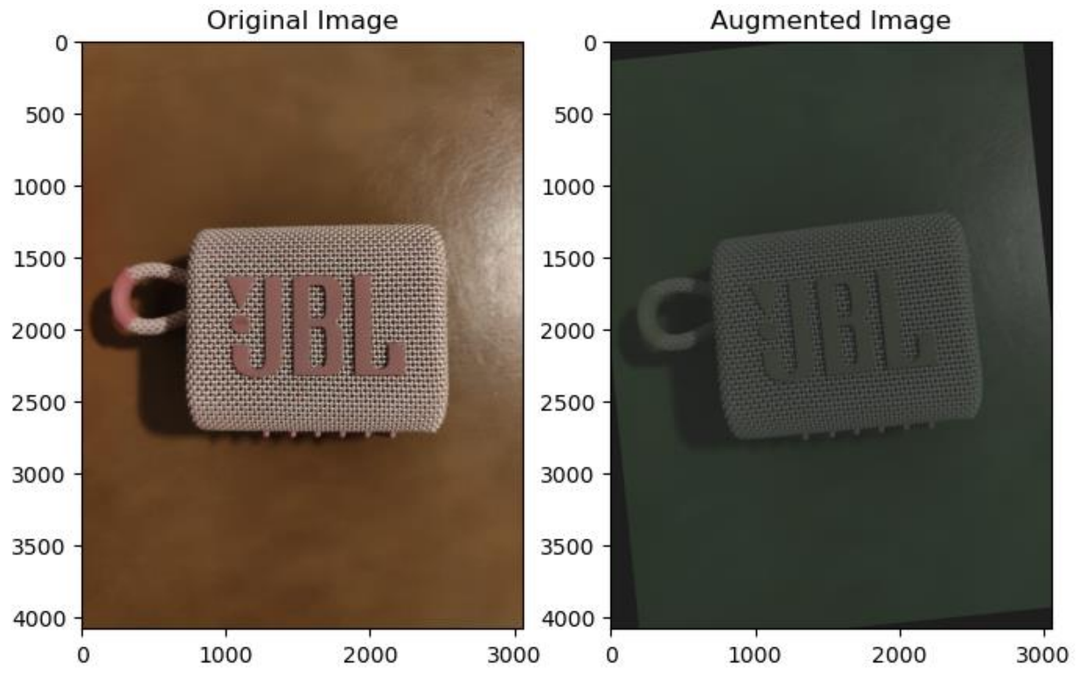


Figure 5: Plotting original vs augmented version of the Image 3 from own dataset.

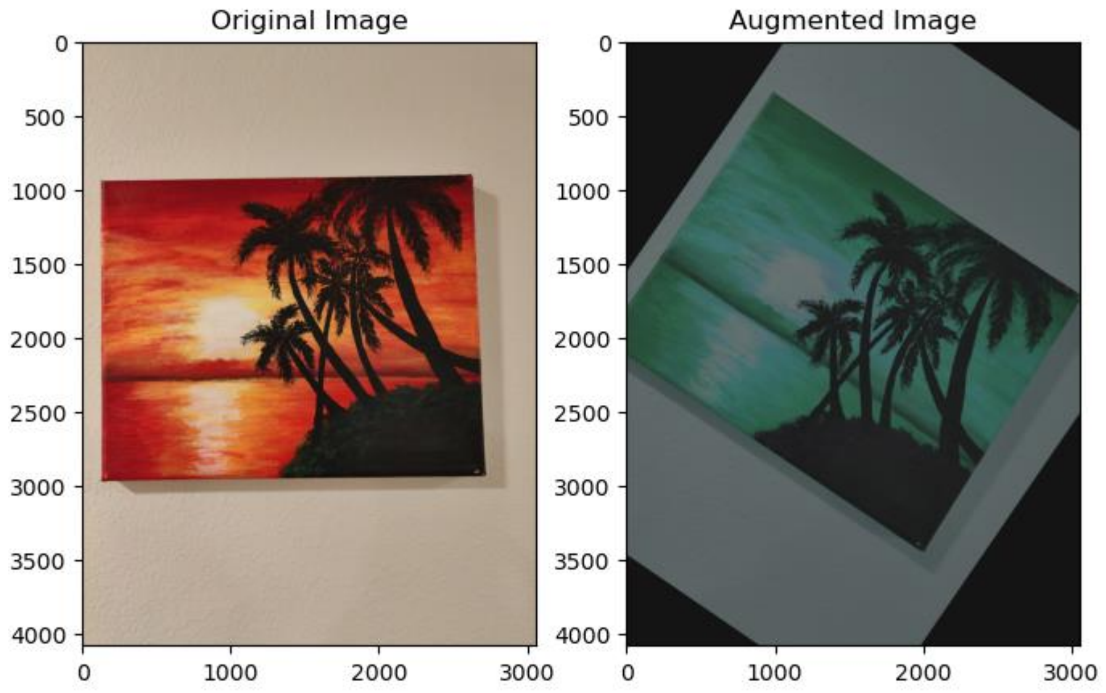


Figure 6: Plotting original vs augmented version of the Image 5 from own dataset.

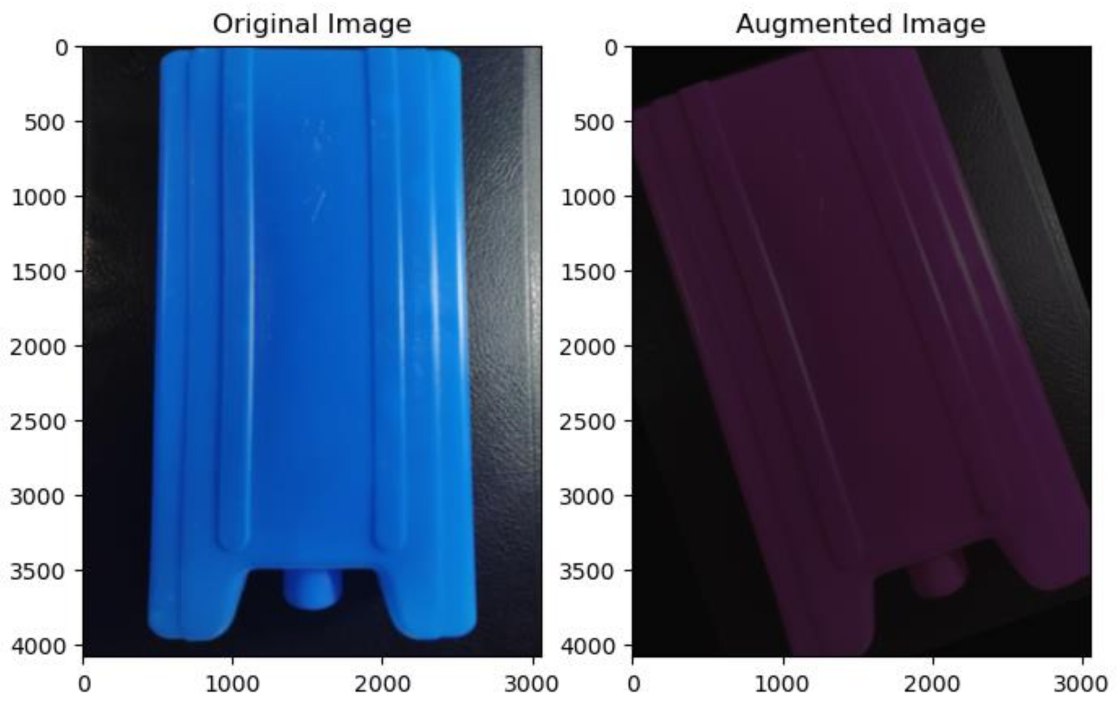
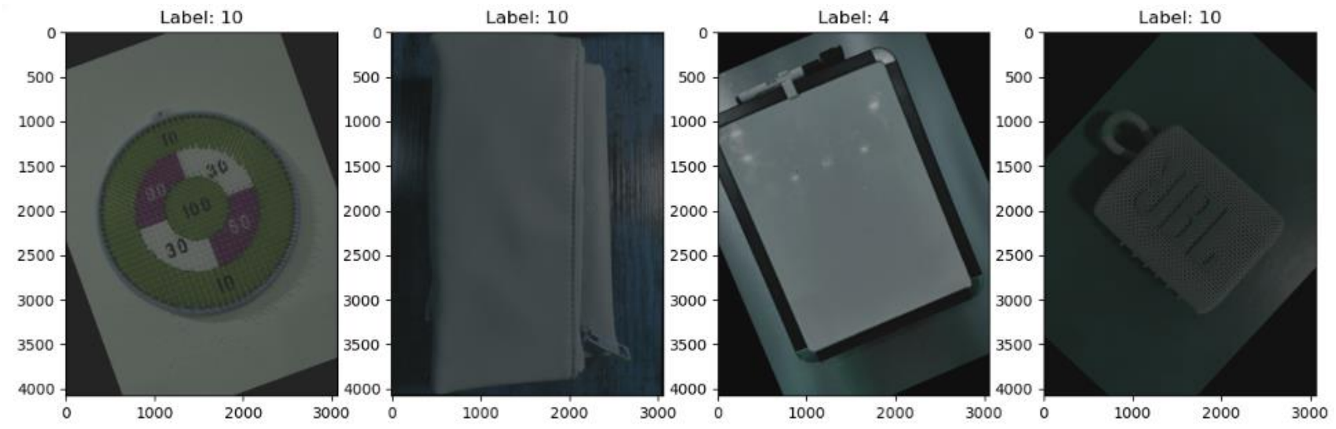


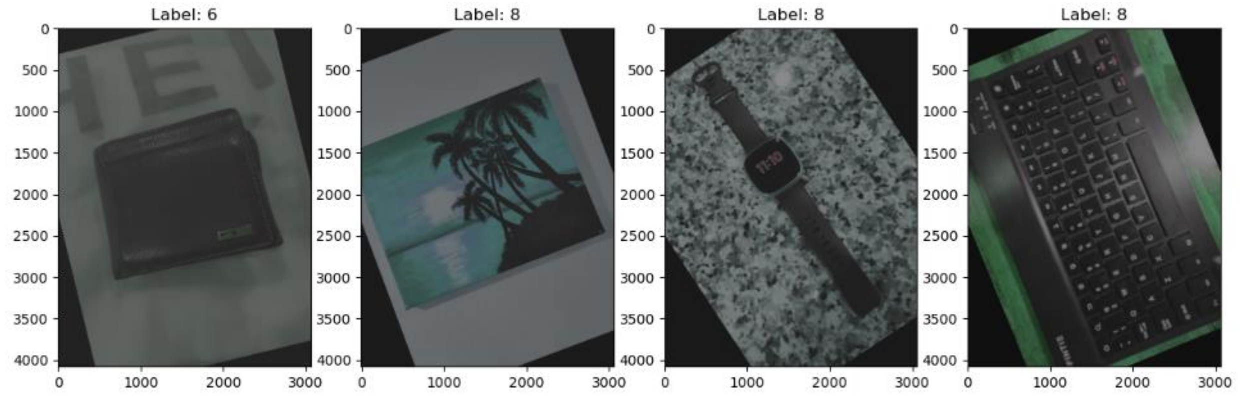
Figure 7: Plotting original vs augmented version of the Image 8 from own dataset.

4. Generating data in parallel

To generate data in parallel *torch.utils.data.DataLoader* is used for the custom dataset class written above. The batch size is set to 4 and the number of workers is set as 2. Two batch samples of 4 images in each and a snippet of a few batches is shown in the plots of Figure 8. Google collab is used for this task since the local system did not support more than zero number of workers.



Batch 1



Batch 2

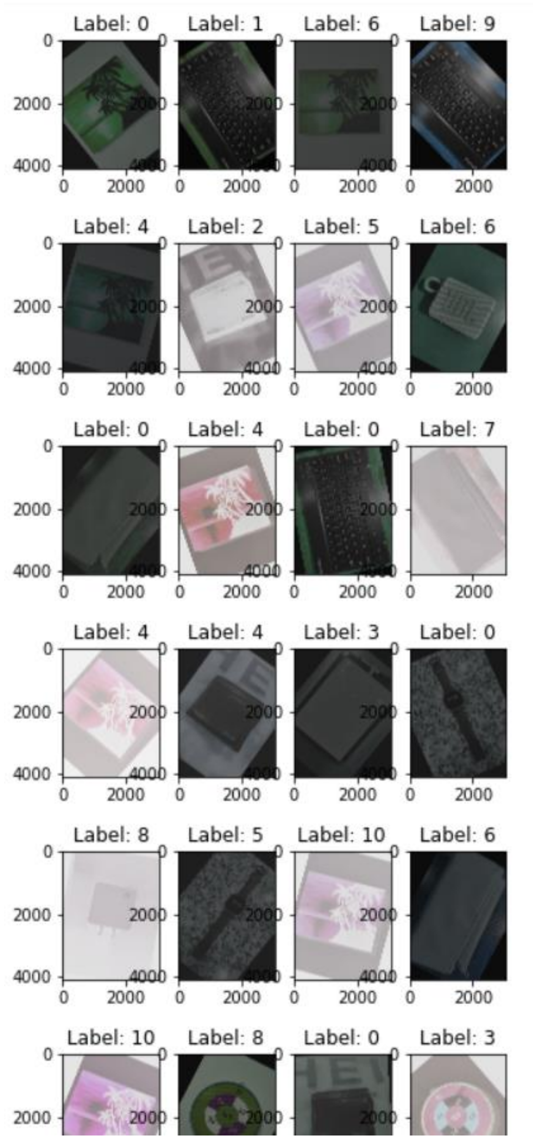
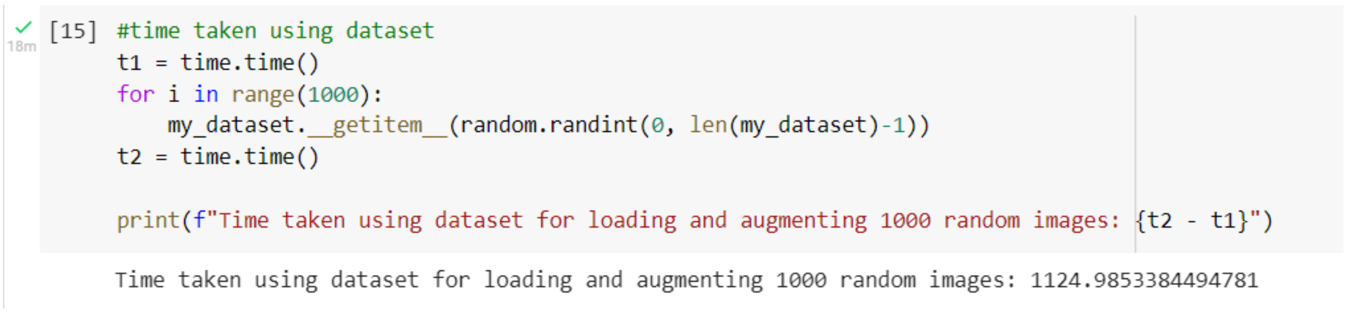


Figure 8: Plotting 4 images together per batch. 2 sample batch images and a snippet of a few batches are shown.

The time taken by calling *my_dataset.__getitem__* 1000 times to load and augment 1000 random images can be seen in Figure 9. The time taken to load the 1000 random images using *my_dataloader* can be seen in Figure 10. A sampler is used to get 1000 images from a 10-image dataset. We can observe that the *dataset takes about 18 minutes (1124 secs) while the dataloader takes 10 second*s for loading 1000 images, thus showcasing the speed, performance and efficient usage of dataloader's multi-threading in deep learning tasks.

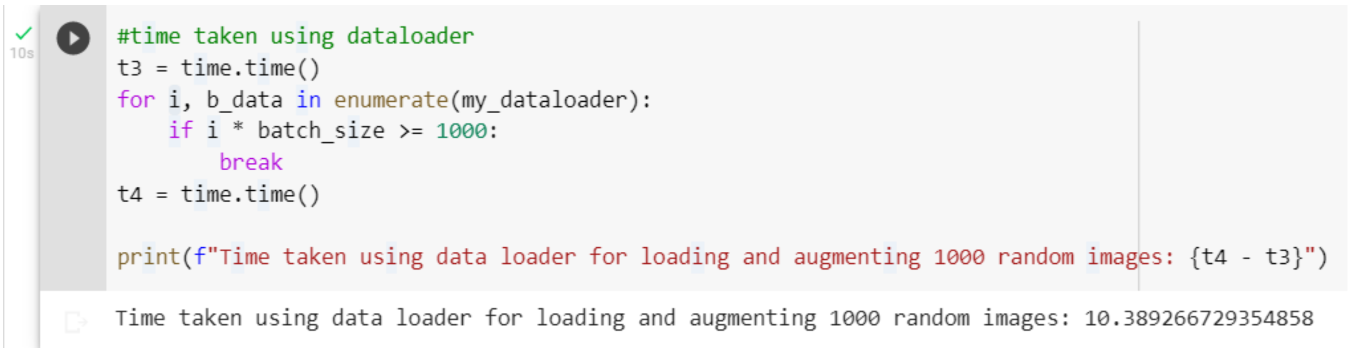


```
[15] #time taken using dataset
     t1 = time.time()
     for i in range(1000):
         my_dataset.__getitem__(random.randint(0, len(my_dataset)-1))
     t2 = time.time()

     print(f"Time taken using dataset for loading and augmenting 1000 random images: {t2 - t1}")

     Time taken using dataset for loading and augmenting 1000 random images: 1124.9853384494781
```

Figure 9: Time taken by the dataset.__getitem__ to load and augment 1000 random images.



```
#time taken using dataloader
t3 = time.time()
for i, b_data in enumerate(my_dataloader):
    if i * batch_size >= 1000:
        break
t4 = time.time()

print(f"Time taken using data loader for loading and augmenting 1000 random images: {t4 - t3}")

Time taken using data loader for loading and augmenting 1000 random images: 10.389266729354858
```

Figure 10: Time taken by the data loader to load and augment 1000 random images.

Further, different batch sizes of [4, 8, 16, 32] and the number of workers of [2, 4, 6, 8] were experimented to check the timings of dataloader in different settings. These results are tabulated in Table 1.

We can observe from Table 1 that the dataloader performed nearly with similar timings in most of the settings, with the lowest time marked for a batch size of 32 and number of workers 8. This can be chosen for similar experiments to obtain better performance gain. Also, taking bigger batch size and number of workers might help in increasing time significantly.

Table 1: Time taken by dataloader for different batch size and number of workers.

Batch size	Number of workers	Time (in secs)
4	2	10.112257480621338
4	4	10.143130540847778
4	6	9.897129774093628
4	8	10.16693377494812
8	2	10.25710678100586
8	4	10.283414125442505
8	6	10.15097427368164
8	8	10.382701396942139
16	2	10.237225770950317
16	4	10.41330361366272
16	6	10.226925134658813
16	8	10.268909215927124
32	2	10.136833429336548
32	4	10.043468952178955
32	6	10.260425567626953
32	8	9.761768341064453

SOURCE CODE:

```
#!/usr/bin/env python
# coding: utf-8

# <h2><center>ECE60146 Deep Learning</center></h2>
# <h3><center>Homework - 2 </center></h3>
# <h3><center>Sahithi Kodali - 34789866</center></h3>
# <h3><center>kodali1@purdue.edu</center></h3>

# ### 2. Data Augmentation

# In[106]:


# Import libraries needed
import numpy as np
import PIL
from PIL import Image
import torchvision.transforms as tvt
import torch
```

```
from scipy.stats import wasserstein_distance


# In[58]:


# Read the images

#get PIL images
stop1_pilImg = Image.open('stop1.jpg')
stop2_pilImg = Image.open('stop2.jpg')
stop2_pilImg.show()

#get tensors of the PIL images
stop1_tensor = tvt.ToTensor()(stop1_pilImg)
stop2_tensor = tvt.ToTensor()(stop2_pilImg)


# In[120]:


affine = tvt.RandomAffine(degrees = 2, translate=None, scale=None, shear=None)
aff_img = affine(stop2_pilImg)
aff_tensor = tvt.ToTensor()(aff_img)
copy_aff = aff_img.copy()


# In[565]:


# Compute similarity of histograms using wasserstein distance

def hist_wass(img1_tensor, img2_tensor):

    #considering only red channel
    img1_red = img1_tensor[0,:,:]
    img2_red = img2_tensor[0,:,:]

    #remove any non-zero pixels
    img1_t = img1_red[img1_red != 0]
    img2_t = img2_red[img2_red != 0]

    #compute histograms and normalize
    hist1 = torch.histc(img1_t, bins=256, min=0, max=255)
    hist1 = hist1/hist1.sum()
    hist2 = torch.histc(img2_t, bins=256, min=0, max=255)
    hist2 = hist2/hist2.sum()

    #find wasserstein distance
    dist = wasserstein_distance(hist1.numpy(), hist2.numpy())
    return dist


# In[578]:


#use the tvt.perspective function to convert the affine image to target image

#give start and end points
startpoints = torch.tensor([[0,716], [700,716], [700,0], [0,0]], dtype =
torch.float32)
endpoints = torch.tensor([[15,716], [675, 716], [695,30], [-20,-68]], dtype =
torch.float32)

#apply projective transformation
projective = tvt.functional.perspective(copy_aff, startpoints, endpoints)
projective.show()

#convert image to tensor
projective_tensor = tvt.ToTensor()(projective)


# In[576]:


#plot target image vs oblique image before transformation
figure, axs = plt.subplots(1, 2, figsize = (8,8))
axs[0].set_title('Target image')
axs[0].imshow(stop1_pilImg)
axs[1].set_title('Oblique image - Before transformation')
axs[1].imshow(stop2_pilImg)
plt.show()


# In[579]:


#plot target image vs oblique image after transformation
figure, axs = plt.subplots(1, 2, figsize = (8,8))
axs[0].set_title('Target image')
axs[0].imshow(stop1_pilImg)
```

```
axs[1].set_title('Oblique image - After transformation')
axs[1].imshow(projective)
plt.show()


# In[582]:


#load one intermediate projective image
proj_ex = Image.open('stop_transf.PNG')
proj_ex_tensor = tvt.ToTensor()(proj_ex)


# In[585]:


#compute similarity between target stop and before transformation of oblique stop
image
h_dist_trans = hist_wass(stop1_tensor, stop2_tensor)
h_dist_trans


# In[583]:


#compute similarity between target stop and after transformation of oblique stop
image
h_dist_org = hist_wass(stop1_tensor, projective_tensor)
h_dist_org


# In[584]:


#compute similarity between target stop and intermediate transformation of
oblique stop image
h_dist_trans_ex = hist_wass(stop1_tensor, proj_ex_tensor)
h_dist_trans_ex


# ### 3. Creating own dataset class

# In[397]:


import os
import random
import matplotlib.pyplot as plt


# In[484]:


#create the own dataset class
class MyDataset(torch.utils.data.Dataset):

    #intializations
    def __init__(self, root):
        super().__init__()
        self.root = root
        self.data = os.listdir(root)
        #transformations chosen for image
        self.transformations = tvt.Compose([tvt.RandomRotation(45),
                                            tvt.ColorJitter(brightness =
[0.4,0.8], contrast = [0.4,0.8], saturation = [0.4,0.8], hue = [0.2, 0.5]),
                                            tvt.RandomInvert(p = 0.2)])

    #compute length of dataset
    def __len__(self):
        return len(self.data)

    #apply transformations for the image chosen by index
    def __getitem__(self, index):
        index = index%10
        img = Image.open(os.path.join(self.root, self.data[index]))
        label = random.randint(0,10)
        if self.transformations:
            img_transformed = self.transformations(img)
        return tvt.ToTensor()(img_transformed), label


# In[485]:


#executing the code given in the HW
my_dataset = MyDataset('C:/Users/Sahithi
Kodali/OneDrive/Purdue/Sem2/DL/HW2/own_data')
print(len(my_dataset))
index = 10
print(my_dataset[index][0].shape, my_dataset[index][1])

index = 50
print(my_dataset[index][0].shape, my_dataset[index][1])
```

```
# In[486]:


#function to plot original vs augmented image
def plot_orgVSaug(org_img, aug_img):
    figure, axs = plt.subplots(1, 2, figsize = (8,8))
    axs[0].set_title('Original Image')
    axs[0].imshow(org_img)
    axs[1].set_title('Augmented Image')
    axs[1].imshow(aug_img)
    plt.show()


# In[491]:


#get the augmented image tensors using the custom dataset class and convert to
PIL
toPIL = tvt.ToPILImage()

img_3_aug = toPIL(my_dataset[3][0])
img_5_aug = toPIL(my_dataset[5][0])
img_8_aug = toPIL(my_dataset[8][0])


# In[492]:


#get the original images in the dataset
img3 = Image.open(r"C:\Users\Sahithi
Kodali\OneDrive\Purdue\Sem2\DL\HW2\own_data\3.jpg")
img5 = Image.open(r"C:\Users\Sahithi
Kodali\OneDrive\Purdue\Sem2\DL\HW2\own_data\5.jpg")
img8 = Image.open(r"C:\Users\Sahithi
Kodali\OneDrive\Purdue\Sem2\DL\HW2\own_data\8.jpg")


# In[493]:


#make the plots for three such images
plot_orgVSaug(img3, img_3_aug)
```

```
# In[494]:


plot_orgVSaug(img5, img_5_aug)


# In[495]:


plot_orgVSaug(img8, img_8_aug)


# ### 4. Generating data in parallel

# In[496]:


import time
from torch.utils.data import DataLoader, sampler


# In[500]:


#give batch size and number of workers
batch_size = 4
num_workers = 2

# Creating a dataloader instance
my_sampler = sampler.RandomSampler(my_dataset, replacement = True, num_samples =
1000)
my_dataloader = DataLoader(my_dataset, batch_size = batch_size, num_workers =
num_workers, sampler = my_sampler)



# In[501]:


#time taken using dataset
t1 = time.time()
for i in range(1000):
    my_dataset.__getitem__(i)
t2 = time.time()

print(f"Time taken using dataset for loading and augmenting 1000 random images:
{t2 - t1}")
```

```
# In[502]:


#time taken using dataloader
t3 = time.time()
for i, b_data in enumerate(my_dataloader):
    if i * batch_size >= 1000:
        break
t4 = time.time()

print(f"Time taken using data loader for loading and augmenting 1000 random
images: {t4 - t3}")


# In[588]:


# Plot all the four images from each batch
for i, b_data in enumerate(my_dataloader):
    imgs, labels = b_data
    figure, axs = plt.subplots(1, batch_size, figsize=(5, 5))

    for i in range(batch_size):
        axs[i].imshow(toPIL(imgs[i]))
        axs[i].set_title(f'Label: {labels[i]}')

    plt.show()


# In[589]:


#check times taken for different batch sizes and number of workers
batches = [4, 8, 16, 32]
workers = [2, 4, 6, 8]
times = []

for b in batches:
    for w in workers:
        my_dataloader = DataLoader(my_dataset, batch_size=batch_size,
num_workers=num_workers, sampler = my_sampler)


        #time taken using dataloader
        t3 = time.time()
        for i, b_data in enumerate(my_dataloader):
            if i * batch_size >= 1000:
                break
        t4 = time.time()
        diff = t4 - t3
        times.append((diff, b, w))

    for t in times:
        print(t)
```

*****