ECE 60146: HW2

Jingsong Lin lin1311@purdue.edu

Section 2

Compare the scaing method used in slide 26 and 28:

If the batch of images which the pixel value are limited to range from **0 to 32** are used, the scaling results using slide 26 and 28 are **different** (as shown in fig1). But if the batch of images which the pixel value are range from **0 to 255** are used, the scaling results using slide 26 and 28 are the **same** (as shown in fig2).

This suggests that *tvt.ToTensor()* will always scale the images by dividing 255 which is the maximum possible pixel value in uint8 format no matter the maximum value in the image. But if we manually scale the image, we will scale the images by diving the maximum value in the images which is not necessary 255. This explain the difference when using the image with the pixel value limited to range from 0 to 32.

The code for the implementation is shown in fig3.

```
result for using code in slide 26
tensor([[0.6250, 0.9688, 0.9688, 0.9375, 0.5625, 0.5625, 0.1875, 0.1562, 0.8125],
        [0.1875, 0.1875, 0.5938, 0.1562, 0.5625, 0.1562, 0.4375, 0.5938, 0.5000],
        [0.2812, 0.3125, 0.7500, 0.4375, 0.1250, 0.2188, 0.1562, 0.2188, 0.9375],
        [0.9375, 0.8438, 0.0000, 0.5312, 0.2188, 0.1875, 0.8125, 0.1250, 0.4688],
        [0.3750, 0.9062, 0.6250, 0.9688, 0.9062, 0.0000, 0.3125, 0.8750, 0.3438]])
result for using code in slide 28
tensor([[0.0784, 0.1216, 0.1216, 0.1176, 0.0706, 0.0706, 0.0235, 0.0196, 0.1020],
        [0.0235, 0.0235, 0.0745, 0.0196, 0.0706, 0.0196, 0.0549, 0.0745, 0.0627],
        [0.0353, 0.0392, 0.0941, 0.0549, 0.0157, 0.0275, 0.0196, 0.0275, 0.1176],
        [0.1176, 0.1059, 0.0000, 0.0667, 0.0275, 0.0235, 0.1020, 0.0157, 0.0588],
        [0.0471, 0.1137, 0.0784, 0.1216, 0.1137, 0.0000, 0.0392, 0.1098, 0.0431]])
```

Fig1 – result for pixel value range from 0 to 32

```
result for using code in slide 26
tensor([[0.6353, 0.3686, 0.8118, 0.4196, 0.3451, 0.8314, 0.3098, 0.3686, 0.8941],
        [0.2706, 0.6078, 0.9647, 0.2902, 0.5333, 0.7961, 0.2314, 0.0353, 0.5255],
        [0.6941, 0.2471, 0.0824, 0.2078, 0.1961, 0.9176, 0.7216, 1.0000, 0.5725],
        [0.0941, 0.7608, 0.2588, 0.8275, 0.9451, 0.7373, 0.6118, 0.1569, 0.8078],
        [0.5529, 0.8235, 0.2549, 0.4549, 0.9961, 0.3882, 0.1373, 0.9020, 0.1294]])
result for using code in slide 28
tensor([[0.6353, 0.3686, 0.8118, 0.4196, 0.3451, 0.8314, 0.3098, 0.3686, 0.8941],
        [0.2706, 0.6078, 0.9647, 0.2902, 0.5333, 0.7961, 0.2314, 0.0353, 0.5255],
        [0.6941, 0.2471, 0.0824, 0.2078, 0.1961, 0.9176, 0.7216, 1.0000, 0.5725],
        [0.0941, 0.7608, 0.2588, 0.8275, 0.9451, 0.7373, 0.6118, 0.1569, 0.8078],
        [0.5529, 0.8235, 0.2549, 0.4549, 0.9961, 0.3882, 0.1373, 0.9020, 0.1294]])
```

Fig2 – result for pixel value range from 0 to 255

```
## Section2

#create random image with the scale from 0 to 32
#images = torch.randint(0,33,(4,3,5,9)).type(torch.uint8)

#create random image with the scale from 0 to 255
images = torch.randint(0,256,(4,3,5,9)).type(torch.uint8)

# code in slide 26
images_scaled = images / images.max().float()

# code in slide 28
images_scaled_2 = torch.zeros_like(images).float()
for i in range(images.shape[0]):
    images_scaled_2[i] = tvt.ToTensor()(np.transpose(images[i].numpy(),(1,2,0)))

print('result for using code in slide 26')
print(images_scaled[0,1,:,:])
print('result for using code in slide 28')
print(images_scaled_2[0,1,:,:])
```

Fig3

Section 2.1

The results for dividing max value and 255 are the **same** since the maximum value in this image is 255.

The comparison result is shown in fig4 and the code for implementation is shown in fig5.

```
the shape of image is (64, 64, 3)
the max value in the image is 255.0
the min value in the image is 0.0
result for dividing the max value
tensor([[0.0745, 0.8392, 0.2941, 0.5647, 0.2157, 0.4941, 0.6314, 0.8980, 0.3216],
        [0.6353, 0.3686, 0.8118, 0.4196, 0.3451, 0.8314, 0.3098, 0.3686, 0.8941],
        [0.0275, 0.5686, 0.1882, 0.0745, 0.7725, 0.3216, 0.0275, 0.6667, 0.0706]])
result for dividing 255
tensor([[0.0745, 0.8392, 0.2941, 0.5647, 0.2157, 0.4941, 0.6314, 0.8980, 0.3216],
        [0.6353, 0.3686, 0.8118, 0.4196, 0.3451, 0.8314, 0.3098, 0.3686, 0.8941],
        [0.0275, 0.5686, 0.1882, 0.0745, 0.7725, 0.3216, 0.0275, 0.6667, 0.0706]])
```

Fig4

```
## Section2.1

image = np.load('test_image.npy')
print(f'the shape of image is {image.shape}')
image_max = images.max().float()
image_min = images.min().float()
print(f'the max value in the image is {image_max}')
print(f'the min value in the image is {image_min}')

images_scaled = images / images.max().float()
images_scaled_2 = images / 255.0

print('result for dividing the max value')
print(images_scaled[0,:,0])
print('result for dividing 255')
print(images_scaled_2[0,:,0])
```

Fig5

Section 3.1

The conda environment has been set up and the environment.yml is attached. Note that since I use mac with gpu does not support cuda, the cudatoolkit cannot be installed.

Section 3.2

Fig6 shows the front and oblique images that I took.

(a)Front image

(b) Oblique image

Fig6

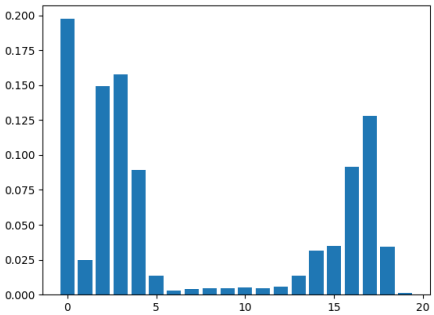
I will try to **transform the front image to the oblique image** using ***tvt.RandomAffine().*** The result is shown in Fig7 which have the comparison with the oblique image.

(a) Best transform image

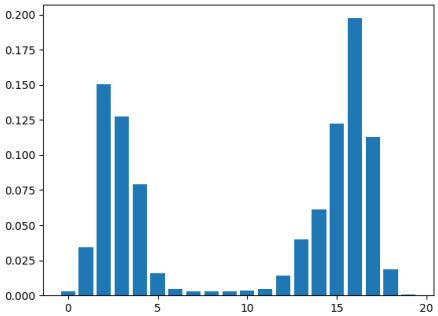
(b) Oblique image(target image)

Fig7

The histograms of the images shown in Fig7 are shown in Fig8 (The histogram have 20 bins in the range of [0.0,1.0]). The **Wasserstein distance** between the histograms of targeted and best transform image is **0.0059**. Note that the Wasserstein distance between the histograms of front and oblique image (the original images) is 0.0081.



(a) Histogram of best transform image



(b) Histogram of target image

Fig8

How to find the "best" transform image:

For the best transform image shown in Fig7, I have use ***tvt.RandomAffine(degrees=(-9.0,-9.0),scale=(0.9,0.9))*** . Specifically, the best degree parameter that I use is -9.0 and the best scaling parameter that I use is 0.9. Other parameters are as default.

To find the best transform image, I have used the grid search: degrees parameter is search from range [-45,45] and the scale parameter is search from range [0.9,1.2]. All the combination will be tried and the one which gives the minimum Wasserstein distance

between histograms of the targeted and transform images will be considered the best parameters set.

The code for implementation is shown in Fig9.

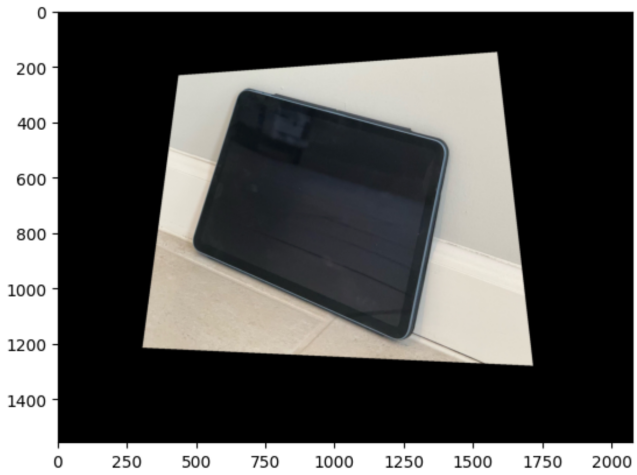
```
image_front = Image.open("front.jpg")
image_oblique = Image.open("oblique.jpg")

hist_front = hist_nor(tvt.ToTensor()(image_front))
hist_oblique = hist_nor(tvt.ToTensor()(image_oblique))
dist = wasserstein_distance(hist_front.cpu().numpy(),hist_oblique.cpu().numpy())
print(f'distance between front and oblique image is {dist}')

degree_list = np.arange(-45.0,45.0,1.0)
scale_list = np.arange(0.8,1.2,0.05)

best_dist = dist

for degree in degree_list:
    for scale_par in scale_list:
        image_affine = tvt.RandomAffine(degrees=(degree,degree),scale=(scale_par,scale_par))(image_front)
        hist_affine = hist_nor(tvt.ToTensor()(image_affine))
        dist = wasserstein_distance(hist_oblique.cpu().numpy(),hist_affine.cpu().numpy())
        if dist <= best_dist:
            best_degree = degree
            best_scale = scale_par
            best_dist = dist


print(f'the distance bewteen front and best affine image is {best_dist}')
print(f'the best degree {best_degree}')
print(f'the best scale {best_scale}')
image_best_affine = tvt.RandomAffine(degrees=(best_degree,best_degree),scale=(best_scale,best_scale))(image_front)
hist_best_affine = hist_nor(tvt.ToTensor()(image_best_affine))
```

```
#plot and save the image
plt.figure()
plt.imshow(image_best_affine)
plt.show()
plt.close()
image_best_affine = image_best_affine.save("best affine.jpg")

# plot histogram
x = range(20)
plt.figure()
plt.bar(x, hist_best_affine, align='center')
plt.xticks(np.linspace(0,20,5))
plt.savefig('best_affine_hist.png')
plt.show()
plt.close()

plt.figure()
plt.bar(x, hist_oblique, align='center')
plt.xticks(np.linspace(0,20,5))
plt.savefig('oblique_hist.png')
plt.show()
plt.close()
```

Fig9

The function tvt.RandomPerspective() has also been tried. The following figure shows the code and the result.

```
image_front = Image.open("front.jpg")
image_oblique = Image.open("oblique.jpg")
image_perspective = tvt.RandomPerspective(distortion_scale=0.5, p=1.0)(image_oblique)
plt.figure()
plt.imshow(image_perspective)
plt.show()
plt.close()
image_perspective = image_perspective.save("perspective.jpg")
```



Section 3.3

The class ***MyDataset*** is defined. I have chose 3 image transform: tvt.RandomRotation, tvt.ColorJitter, tvt.RandomAffine.

The reason to choose tvt.RandomRotation: This function will rotate the image by a certain degree. This is a geometry-related transform. People might observe the object at different angles but the classification of this object should not change. This function can make classifier robust to this scenario.

The reason to choose tvt.ColorJitter: This function will randomly change the brightness, contrast, saturation and hue of an image. This is a color-related transform. People might observe the object in different brightness or environment but the classification of this object should not change. This function can make classifier robust to this scenario.

The reason to choose tvt.RandomAffine: This is a geometry-related transform. People might observe the object in different viewpoint but the classification of this object should not change. This function can make classifier robust to this scenario.

Table 1 compare the original and augmented images for 3 different object.

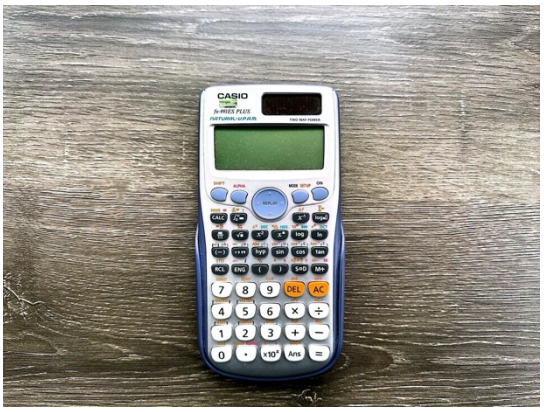
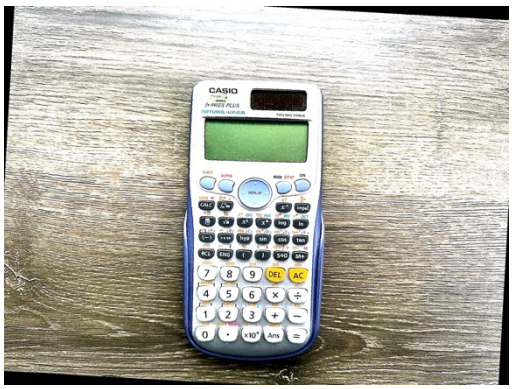
Original image	Augmented image
	
	
	

Table1

The code for implement ***MyDataset*** class is shown in Fig10 and the test case is shown in Fig11.

```python
## Section 3.3
class MyDataset(torch.utils.data.Dataset):
    def __init__(self, root):
        super().__init__()
        # Obtain meta information (e.g. list of file names)
        # Initialize data augmentation transforms , etc.
        self.path = root
        self.transform = tvt.Compose([
            tvt.RandomRotation(degrees=(0, 45)),
            tvt.ColorJitter(brightness=.5, hue=.3),
            tvt.RandomAffine(degrees=(-5.0,5.0)),
        ])

    def __len__(self):
        # Return the total number of images
        # the number is a place holder only
        return 100

    def __getitem__(self, index):
        # Read an image at index and perform augmentations
        # Return the tuple: (augmented tensor, integer label)
        index = index % 10 # only 10 images in the folder
        image_path = os.path.join(self.path,str(index)+'.jpg')
        image = Image.open(image_path)
        image = self.transform(image)
        return (image, index)
```

Fig10

```python
## Section 3.3 test case
my_dataset = MyDataset('./image_folder/')
print(len(my_dataset))

index = 50
image_tuple_1 = my_dataset[index]
print(image_tuple_1[0].size, image_tuple_1[1])

index = 22
image_tuple_2 = my_dataset[index]
print(image_tuple_2[0].size, image_tuple_2[1])

index = 34
image_tuple_3 = my_dataset[index]
print(image_tuple_3[0].size, image_tuple_3[1])
```

```
100
(2076, 1556) 0
(2076, 1556) 2
(2076, 1556) 4
```

Fig11

Section 3.4

Using **DataLoader** to get a batch of images with batch size 4. Note that to use Dataloader, the image need to convert to tensor (I previously use PIL image as input and output). Also the images need to be the same size.

Fig12 shows the 4 images returned by my dataloader and Fig13 shows the code to implement this.

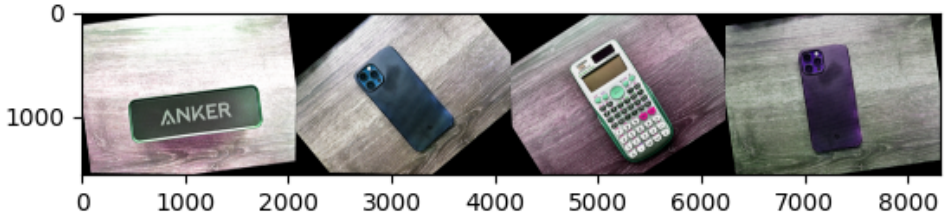


Fig12

```
## Section 3.4
my_dataset = MyDataset('./image_folder/')
batch_size = 4
dataloader = DataLoader(my_dataset,batch_size=batch_size,shuffle=True)

images, labels = next(iter(dataloader)) # get a batch
image_combine = torchvision.utils.make_grid(images)
image_combine = image_combine.numpy()
image_combine = np.transpose(image_combine,(1,2,0))
plt.figure()
plt.imshow(image_combine)
plt.savefig('image_combine.png')
plt.show()
plt.close()
```

Fig13

Compare the performance using multi-threaded **DataLoader** and using **Dataset**:

First I will compute the time to return 1000 images by calling my_dataset.__getitem__ directly. Note that I only use **tvt.ColorJitter** transform in this and the following experiment. The intention is to save time but this will still be the fair comparison.

Fig14 shows the code and the running time. The time to return 1000 images by calling my_dataset.__getitem__ directly is **179.02 sec**

```python
## Section 3.4
#calculate the time to get 1000 images using __getitem__
my_dataset = MyDataset('./image_folder/')
start_time = time.time()

for index in range(1000):
    image_tuple_1 = my_dataset[index]

print(f'time for getting 1000 images using __getitem__ is {time.time()-start_time}')
```

```
time for getting 1000 images using __getitem__ is 179.01815342903137
```

Fig14

Next I will compute the time by using Dataloader. I will first fix the batch size to be 4 and change the number of workers. Fig15 shows the code and the run time. The run time is also summarize in table2.

```python
## Section 3.4
#calculate the time to get 1000 images using dataloder
my_dataset = MyDataset('./image_folder/')

# fix the batch size
batch_size = 4
num_workers_list = [2,4,6]

for num_workers in num_workers_list:
    dataloader = DataLoader(my_dataset,batch_size=batch_size,shuffle=True,num_workers=num_workers)
    dataiter = iter(dataloader)

    start_time = time.time()
    for index in range(int(1000/batch_size)):
        try:
            image_tuple = next(dataiter)
        except StopIteration:
            pass

    print(f'time for getting 1000 images using dataloader with batch size {batch_size} and number of workers {num_workers} is {time.time()-start_time}')
```

```
time for getting 1000 images using dataloader with batch size 4 and number of workers 2 is 21.247254848480225
time for getting 1000 images using dataloader with batch size 4 and number of workers 4 is 14.989679098129272
time for getting 1000 images using dataloader with batch size 4 and number of workers 6 is 15.784927606582642
```

Fig15

Batch size	Number of workers	Running time(s)
4	2	21.25
4	4	14.99
4	6	15.78

Table2

It is expected that using Dataloader is much faster than using __getitem__ directly. This is due to the parallelism. Also, if the number of worker is smaller than the batch size, as the number of workers increase, the running time decrease. But when the number of worker is larger than the batch size, the running time will not decrease as the number of worker increase.

Finally, I will fix the number of workers to be 6 and increase the batch size. Fig16 shows the code and the run time. The run time is also summarize in table3.

```python
## Section 3.4
#calculate the time to get 1000 images using dataloder
my_dataset = MyDataset('./image_folder/')

# fix the batch size
batch_size_list = [2,4,8,10,50]
num_workers = 6

for batch_size in batch_size_list:
    dataloader = DataLoader(my_dataset,batch_size=batch_size,shuffle=True,num_workers=num_workers)
    dataiter = iter(dataloader)

    start_time = time.time()
    for index in range(int(1000/batch_size)):
        try:
            image_tuple = next(dataiter)
        except StopIteration:
            pass

    print(f'time for getting 1000 images using dataloader with batch size {batch_size} and number of workers {num_workers} is {time.time()-start_time}')
```

```
time for getting 1000 images using dataloader with batch size 2 and number of workers 6 is 13.980159997940063
time for getting 1000 images using dataloader with batch size 4 and number of workers 6 is 14.10300350189209
time for getting 1000 images using dataloader with batch size 8 and number of workers 6 is 14.488687515258789
time for getting 1000 images using dataloader with batch size 10 and number of workers 6 is 13.96829104423523
ERROR: Unexpected bus error encountered in worker. This might be caused by insufficient shared memory (shm).
```

Fig16

Number of workers	Batch size	Running time(s)
6	2	13.98
6	4	14.10
6	8	14.48
6	10	13.97

Table3

As the batch size increase, the running time remain approximately the same if the number of workers is fixed. When the batch size is too big, we might run into insufficient memory problem (this issue is shown in fig16 when I set the batch size to be 50).

Section 3.5

By running the code shown in Fig17. Without setting the seed, the return from the iterator are not the same for two different time (result shown in Fig18).

The reason that we have different results in different times is we shuffle the data differently in different time. Without setting the seed, we will expect the shuffle results will be different.

```python
## Section 3.5

def plot_image(images,index):
    image_combine = torchvision.utils.make_grid(images)
    image_combine = image_combine.numpy()
    image_combine = np.transpose(image_combine,(1,2,0))
    plt.figure()
    plt.imshow(image_combine)
    plt.savefig(f'image_combine{index}.png')
    plt.show()
    plt.close()

my_dataset = MyDataset('./image_folder/')
batch_size = 2
dataloader = DataLoader(my_dataset,batch_size=batch_size,shuffle=True)

for batch in dataloader:
    images,labels = batch
    print('the first time')
    plot_image(images,1)
    break

for batch in dataloader:
    images,labels = batch
    print('the second time')
    plot_image(images,2)
    break
```

Fig17

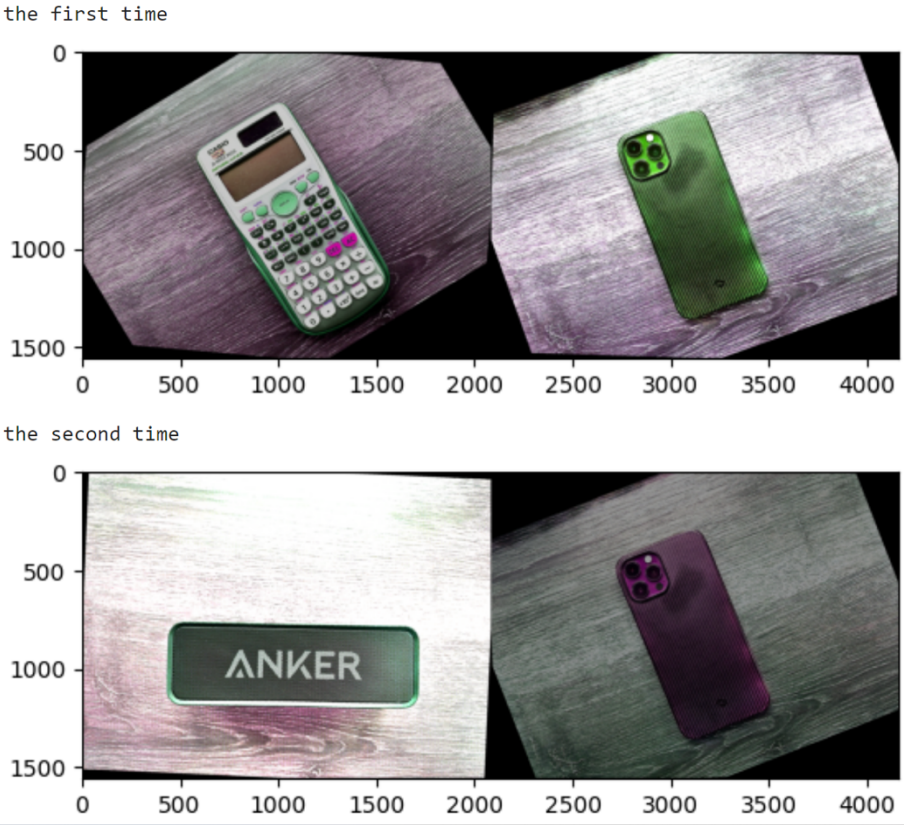


Fig18

By setting the seed below the import (as shown in Fig19) and also setting the seed before running the iterator (as shown in Fig20). The return images will **remain the same** (as shown in Fig21).

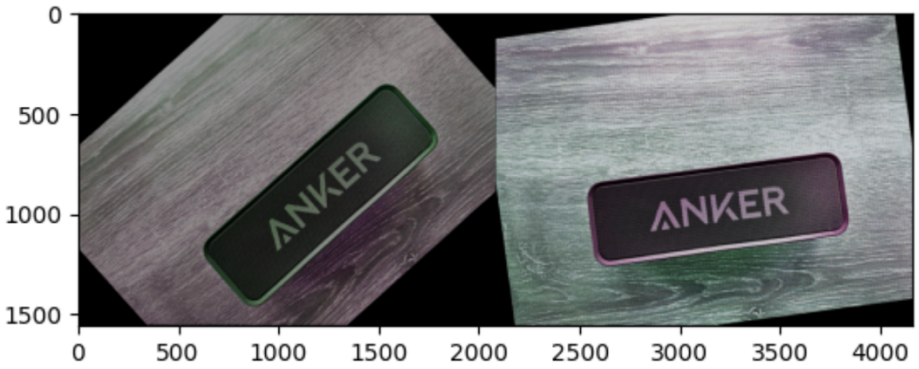
```
import torch
import torchvision
import torchvision.transforms as tvt
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
from scipy.stats import wasserstein_distance
import os
from torch.utils.data import DataLoader
import time
import random

#setting the seed as shown in the lecture slice
seed = 60146
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
np.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
```

Fig19

```
## Section 3.5

def plot_image(images,index):
    image_combine = torchvision.utils.make_grid(images)
    image_combine = image_combine.numpy()
    image_combine = np.transpose(image_combine,(1,2,0))
    plt.figure()
    plt.imshow(image_combine)
    plt.savefig(f'image_combine{index}.png')
    plt.show()
    plt.close()

my_dataset = MyDataset('./image_folder/')
batch_size = 2
dataloader = DataLoader(my_dataset,batch_size=batch_size,shuffle=True)

torch.manual_seed(seed)
for batch in dataloader:
    images,labels = batch
    print('the first time')
    plot_image(images,1)
    break

torch.manual_seed(seed)
for batch in dataloader:
    images,labels = batch
    print('the second time')
    plot_image(images,2)
    break
```

Fig20

the first time



the second time

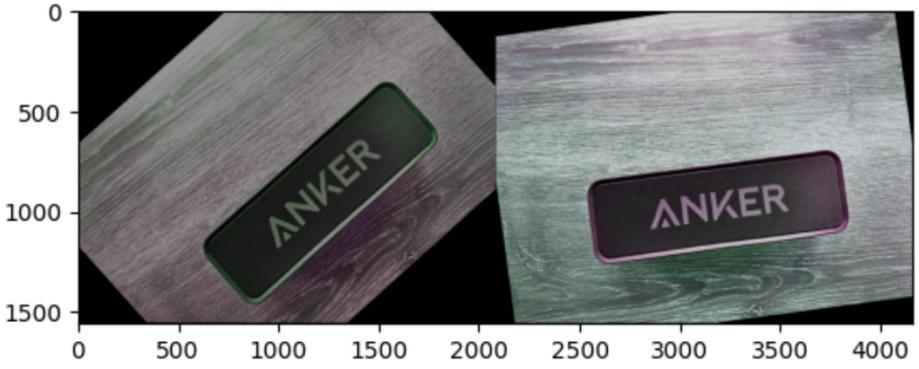


Fig21