ECE 60146 Deep Learning Homework 2

Mehmet Berk Sahin, sahinm@purdue.edu, 34740048

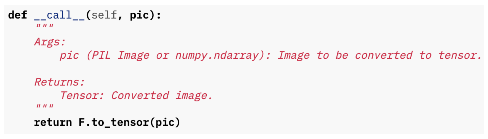
January 26, 2023

1 Introduction

In this homework, we covered the pieces needed to implement a custom image dataset wrapped to PyTorch dataloader for training and testing neural networks. To achieve that, I familiarized myself with image representations such as PIL and torch tensor, and the data augmentation processes, which are both color-related and geometry-related transformations. They are implemented using the torchvision library with the addition of os and numpy. In this report, I will answer each question by sharing relevant code snippets and by explaining how the algorithm solves the problem.

2 Understanding Data Normalization

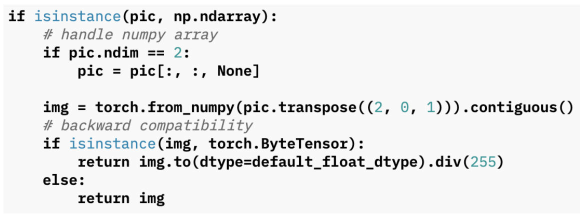
This question is about the 'mystery' in the Slides 19 through 36 and given in the homework so I will not rephrase it here. From the Slides 26 and 28, it seems that `tvt.ToTensor` divides the images with their maximum values but that's not quite true because if that was true, we would see 1.0 at every image. To see what is going on here, I checked the source code of `tvt.ToTensor` class. In Slide 28, we create an instance of `tvt.ToTensor` and called it as a function, in which we input an image as a parameter. So, I checked the `__call__` instance method of `tvt.ToTensor`. It is given below:



```
def __call__(self, pic):
    """
    Args:
        pic (PIL Image or numpy.ndarray): Image to be converted to tensor.

    Returns:
        Tensor: Converted image.
    """
    return F.to_tensor(pic)
```

Figure 1: Call instance method of ToTensor class

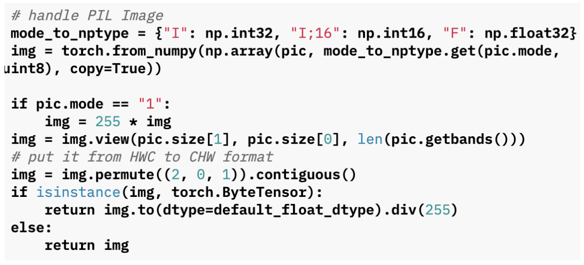
It returns the output of `tvt.functional.to_tensor` method. So, I checked the source code of this function too. Since there are lot of lines, I cropped the necessary parts from the source code.



```
if isinstance(pic, np.ndarray):
    # handle numpy array
    if pic.ndim == 2:
        pic = pic[:, :, None]

    img = torch.from_numpy(pic.transpose((2, 0, 1))).contiguous()
    # backward compatibility
    if isinstance(img, torch.ByteTensor):
        return img.to(dtype=default_float_dtype).div(255)
    else:
        return img
```

Figure 2: Converting NumPy image to PyTorch tensor

In Figure 2, if the input is a `NumPy` array, it adjusts the image size according to `(C, H, W)`. And if the data type of the entries in the array is not `np.uint8`, it returns the tensor without scaling. Else, it returns the image by **dividing it by 255** (because of `.div(255)` call). So, instead of the maximum value in an image, `tvt.ToTensor` scales the images by 255, which is the maximum value of `RGB` range. In Slide 26, since the maximum value over the batch is 255, the results of Slides 26 and 28 were exactly the same. Lastly, note that before the division by 255, tensor entries are converted to float-type numbers to not get zeroes after the division. For the `PIL` images, we have the same scaling as can be seen in Figure 3.



```
# handle PIL Image
mode_to_nptype = {"I": np.int32, "I;16": np.int16, "F": np.float32}
img = torch.from_numpy(np.array(pic, mode_to_nptype.get(pic.mode,
uint8), copy=True))

if pic.mode == "1":
    img = 255 * img
img = img.view(pic.size[1], pic.size[0], len(pic.getbands()))
# put it from HWC to CHW format
img = img.permute((2, 0, 1)).contiguous()
if isinstance(img, torch.ByteTensor):
    return img.to(dtype=default_float_dtype).div(255)
else:
    return img
```

Figure 3: Converting PIL image to PyTorch tensor

Similar to `NumPy`, `PIL` images are converted to tensor and then permuted to get `(C, H, W)`. Then, if the entries are `torch.uint8`, entries are converted to float-type and divided by 255. Otherwise, the image tensor is returned without any scaling.

3 Programming Tasks

3.1 Setting Up Your Conda Environment

I have submitted my `environment.yml` with the `.zip` file I have uploaded to Brightspace. And I have followed the instructions given in the assignment. You can check my submission. Note that since I am using macOS, I couldn't install `cudatoolkit`, which is fine as mentioned by TA.

3.2 Becoming Familiar with `torchvision.transforms`

After reviewing the material on Slide 37 through 47 of the Week 2 slides on `torchvision`, I have followed the instructions in the assignment part `3.2`. I have taken a photo of a stop sign near Purdue University, EE building. The first picture was taken from a front view and the second one was taken from an oblique angle. They are given in Figure 4.

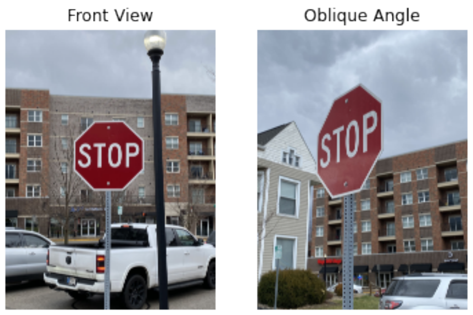
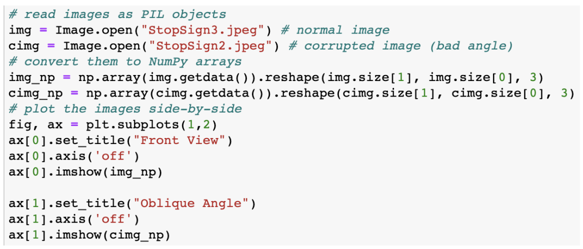


Figure 4: Stop Signs from Front View and Oblique Angle

And the code snippet that loads these images and displays them are given in Figure 5.



```
# read images as PIL objects
img = Image.open("StopSign3.jpeg") # normal image
cimg = Image.open("StopSign2.jpeg") # corrupted image (bad angle)
# convert them to NumPy arrays
img_np = np.array(img.getdata()).reshape(img.size[1], img.size[0], 3)
cimg_np = np.array(cimg.getdata()).reshape(cimg.size[1], cimg.size[0], 3)
# plot the images side-by-side
fig, ax = plt.subplots(1,2)
ax[0].set_title("Front View")
ax[0].axis('off')
ax[0].imshow(img_np)

ax[1].set_title("Oblique Angle")
ax[1].axis('off')
ax[1].imshow(cimg_np)
```

Figure 5: Code Snippet for displaying the previous result

It loads the `.jpeg` images as a `PIL` image. Then it converts the `PIL` instances to `NumPy` instance to be plotted via `matplotlib.pyplot`. And last 8 lines plot the result.

Now, I experimented with the callable instances `tvt.RandomAffine` and `tvt.functional.perspective()` that are mentioned on Slides 46 and 47 of Week 2 in order to transform one image into other as much as possible. I plotted the transformed image, which is the most similar to the target image, with the target stop sign side-by-side. Result is given in Figure 6.

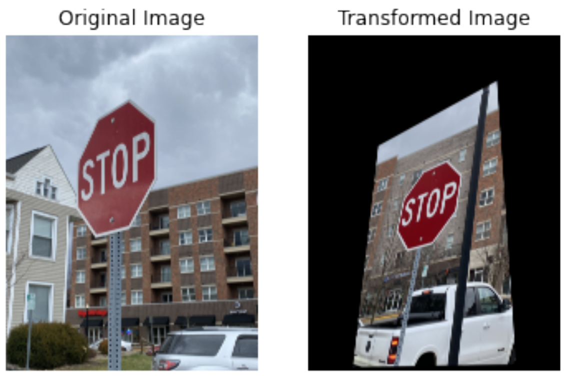
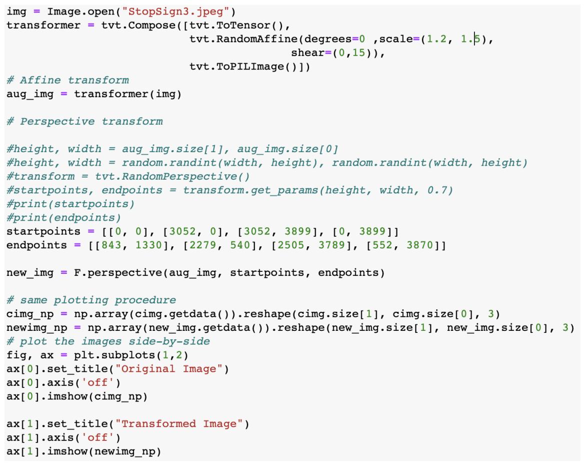


Figure 6: Target Image vs. Transformed Image

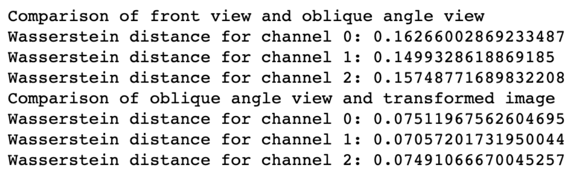
I achieved this by using both affine and projective transformations. I used `tvt.RandomAffine` and `tvt.RandomPerspective` functions to generate random affine and perspective transformation on the original image, which is front-view, until I get a similar image to the target image. While I was applying random transformations, I saved the parameters so that if I see a successful transformation, I keep the parameters and reuse them. In `tvt.RandomAffine` function, I randomly choose scaling from $(1.2, 1.5)$ and shearing from $(0, 15)$. I think these two parameters are essential to give a angled view to a stop-sign. Moreover, to add a different perspective to an image, I used `tvt.RandomAffine` method. You can check the code snippet in Figure 7. I used `tvt.Compose` to call `tvt.ToTensor`, `tvt.RandomAffine`, and `tvt.ToPILImage` callable instances at once. And I used the `tvt.RandomPerspective` seperately to have a access to the random parameters, which are passed to the `torchvision.transform.functional.perspective` function, to display the final result. Lastly, I converted the final results to `NumPy` arrays to plot them.



```
img = Image.open("StopSign3.jpeg")
transformer = tvt.Compose([tvt.ToTensor(),
                           tvt.RandomAffine(degrees=0 ,scale=(1.2, 1.5),
                                            shear=(0,15)),
                           tvt.ToPILImage()])
# Affine transform
aug_img = transformer(img)

# Perspective transform

#height, width = aug_img.size[1], aug_img.size[0]
#height, width = random.randint(width, height), random.randint(width, height)
#transform = tvt.RandomPerspective()
#startpoints, endpoints = transform.get_params(height, width, 0.7)
#print(startpoints)
#print(endpoints)
startpoints = [[0, 0], [3052, 0], [3052, 3899], [0, 3899]]
endpoints = [[843, 1330], [2279, 540], [2505, 3789], [552, 3870]]

new_img = F.perspective(aug_img, startpoints, endpoints)

# same plotting procedure
cimg_np = np.array(cimg.getdata()).reshape(cimg.size[1], cimg.size[0], 3)
newimg_np = np.array(new_img.getdata()).reshape(new_img.size[1], new_img.size[0], 3)
# plot the images side-by-side
fig, ax = plt.subplots(1,2)
ax[0].set_title("Original Image")
ax[0].axis('off')
ax[0].imshow(cimg_np)

ax[1].set_title("Transformed Image")
ax[1].axis('off')
ax[1].imshow(newimg_np)
```

Figure 7: Code Snippet for Augmentation

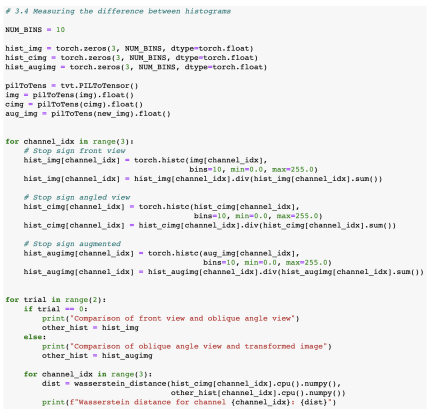
To measure the similarity between two images of the stop sign, I created the histogram of the images and calculate the **wasserstein distance** between the histograms. I have calculate the wasserstein distance *per-channel basis* as explained in the slides and I have used `wasserstein_distance()` function of `scipy.stats` package.



```
Comparison of front view and oblique angle view
Wasserstein distance for channel 0: 0.16266002869233487
Wasserstein distance for channel 1: 0.1499328618869185
Wasserstein distance for channel 2: 0.15748771689832208
Comparison of oblique angle view and transformed image
Wasserstein distance for channel 0: 0.07511967562604695
Wasserstein distance for channel 1: 0.07057201731950044
Wasserstein distance for channel 2: 0.07491066670045257
```

Figure 8: Wasserstein Distances for Front View - Oblique Angle and Oblique Angle - Transformed

As you can see in Figure 8, Wasserstein distance decreases after the transformation. This suggests us that image becomes more similar to the target (oblique angle) image. However, one should keep in mind that due to the perspective effect, we have more zeroes in the transformed image, which is an another reason why the difference is small. I asked this to TA and s/he said no problem. The code snippet for this part is given in Figure 9.



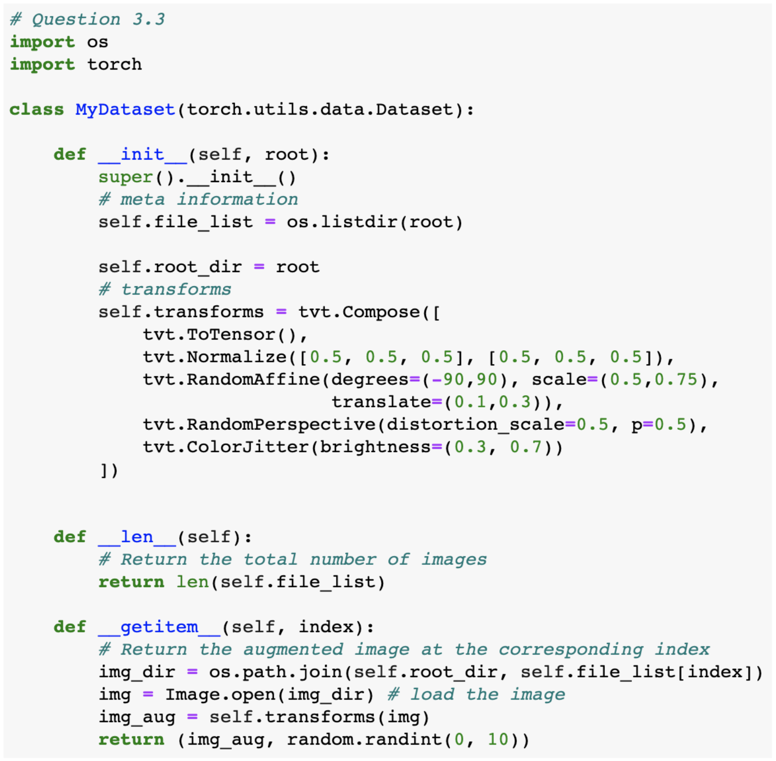
```
# 3.4 Measuring the difference between histograms

NUM_BINS = 10

hist_img = torch.zeros(3, NUM_BINS, dtype=torch.float)
hist_cimg = torch.zeros(3, NUM_BINS, dtype=torch.float)
hist_augimg = torch.zeros(3, NUM_BINS, dtype=torch.float)

pilToTens = tvt.PILToTensor()
img = pilToTens(img).float()
cimg = pilToTens(cimg).float()
aug_img = pilToTens(new_img).float()


for channel_idx in range(3):
    # Stop sign front view
    hist_img[channel_idx] = torch.histc(img[channel_idx],
                                        bins=10, min=0.0, max=255.0)
    hist_img[channel_idx] = hist_img[channel_idx].div(hist_img[channel_idx].sum())

    # Stop sign angled view
    hist_cimg[channel_idx] = torch.histc(hist_cimg[channel_idx],
                                         bins=10, min=0.0, max=255.0)
    hist_cimg[channel_idx] = hist_cimg[channel_idx].div(hist_cimg[channel_idx].sum())

    # Stop sign augmented
    hist_augimg[channel_idx] = torch.histc(aug_img[channel_idx],
                                           bins=10, min=0.0, max=255.0)
    hist_augimg[channel_idx] = hist_augimg[channel_idx].div(hist_augimg[channel_idx].sum())


for trial in range(2):
    if trial == 0:
        print("Comparison of front view and oblique angle view")
        other_hist = hist_img
    else:
        print("Comparison of oblique angle view and transformed image")
        other_hist = hist_augimg

    for channel_idx in range(3):
        dist = wasserstein_distance(hist_cimg[channel_idx].cpu().numpy(),
                                    other_hist[channel_idx].cpu().numpy())
        print(f"Wasserstein distance for channel {channel_idx}: {dist}")
```

Figure 9: Code Snippet for Wasserstein Distance

In the code, I converted the `PIL` images to tensors whose entries are float-point number. Then, tensors at each channels are converted to histograms and saved into the zero tensors, which are initialized at the beginning. Then, wasserstein distances were calculated and printed as a result.

3.3 Creating Your Own Dataset Class

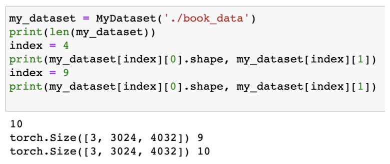
In this section, I implemented a custom dataset class with the desired properties given in the homework. I will explain the implementation through the code, which is given in Figure **??**



```python
# Question 3.3
import os
import torch

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

    def __init__(self, root):
        super().__init__()
        # meta information
        self.file_list = os.listdir(root)

        self.root_dir = root
        # transforms
        self.transforms = tvt.Compose([
            tvt.ToTensor(),
            tvt.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]),
            tvt.RandomAffine(degrees=(-90,90), scale=(0.5,0.75),
                             translate=(0.1,0.3)),
            tvt.RandomPerspective(distortion_scale=0.5, p=0.5),
            tvt.ColorJitter(brightness=(0.3, 0.7))
        ])


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

    def __getitem__(self, index):
        # Return the augmented image at the corresponding index
        img_dir = os.path.join(self.root_dir, self.file_list[index])
        img = Image.open(img_dir) # load the image
        img_aug = self.transforms(img)
        return (img_aug, random.randint(0, 10))
```

Figure 10: MyDataset Class Implementation

This class inherited the properties of `torch.utils.data.Dataset` class. In the `__init__(self, root)`, three instance variables are initialized. `self.file_list` keeps the image file names, which are uploaded by me (10 images mentioned in the homework), in a list. `self.root_dir` is the path as a string to the data folder. And `self.transforms` keeps the callable instances of transformation objects in a list and passes an image through the callable instances sequentially. `__len__(self)` returns the number of images in the data file. Lastly, `__getitem__(self, index)` loads the image corresponding to the given index from the data folder at the root path. It applies the transformations and then returns a tuple consisting of transformed image and its random label. I took a photo of a book from 10 different views. And as a transformation, I have used `tvt.RandomAffine`, `tvt.RandomPerspective` and `tvt.ColorJitter`. The first two of them are geometry-based and the last one is color-based. I thought to detect a book, both lightning and angle or perspective are important so I picked them. Code snippet given in Figure 11 works as in the example given in the homework.



```
my_dataset = MyDataset('./book_data')
print(len(my_dataset))
index = 4
print(my_dataset[index][0].shape, my_dataset[index][1])
index = 9
print(my_dataset[index][0].shape, my_dataset[index][1])
```

```
10
torch.Size([3, 3024, 4032]) 9
torch.Size([3, 3024, 4032]) 10
```

Figure 11: Code Snippet for MyDataset

I chose three images randomly and plot their original versions with their augmented versions side-by-side.

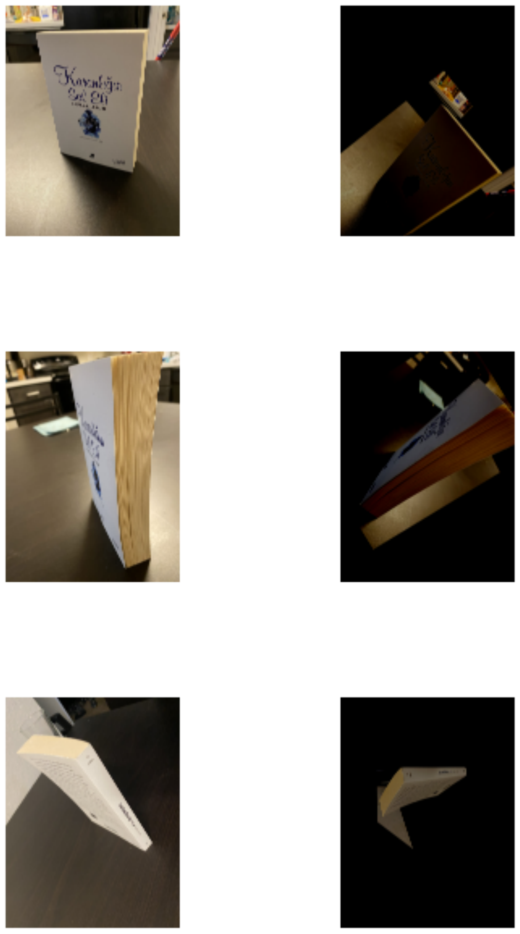
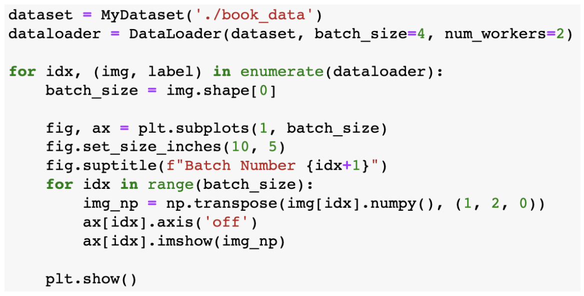


Figure 12: Original Images vs. Augmented Images

Images at the left column consist of the original images taken from the local disk and the images at the right column are their augmented versions. As it can be seen in the plots in Figure 12, brightness of the images changed as I wanted. This will increase the generalization power of the network because all of the dataset consists of images, which are taken under good light conditions. By adding poor light conditions, I increase the variability of the dataset. Also, similar to the stop sign example, images were augmented such that different perspectives and angles were generated. This is useful to increase the variability thereby generalization capability of the network.

3.4 Generating Data in Parallel

The purpose of a dataloader is to load and to augment the training samples efficiently in a multi-threaded fashion. To achieve that, I wrote the following code snippet.



```
dataset = MyDataset('./book_data')
dataloader = DataLoader(dataset, batch_size=4, num_workers=2)

for idx, (img, label) in enumerate(dataloader):
    batch_size = img.shape[0]

    fig, ax = plt.subplots(1, batch_size)
    fig.set_size_inches(10, 5)
    fig.suptitle(f"Batch Number {idx+1}")
    for idx in range(batch_size):
        img_np = np.transpose(img[idx].numpy(), (1, 2, 0))
        ax[idx].axis('off')
        ax[idx].imshow(img_np)

    plt.show()
```

Figure 13: Code Snippet for Implementing DataLoader with Custom Dataset

In Figure 13, I wrap an instance of my custom dataset class within the `torch.utils.data.DataLoader` class so that my images can be processed in parallel and are returned batches. Output of this dataset can be seen below:

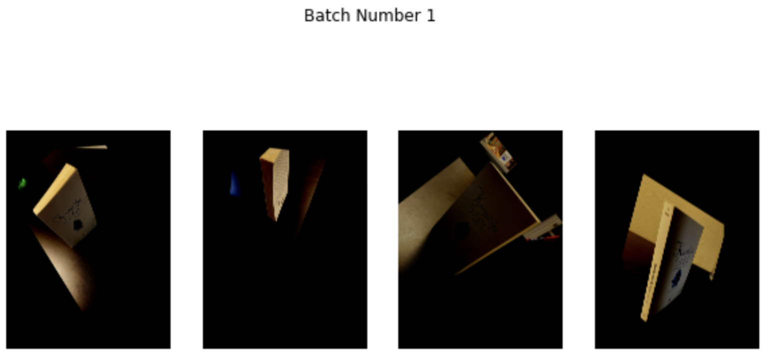


Figure 14: First Batch of the DataLoader

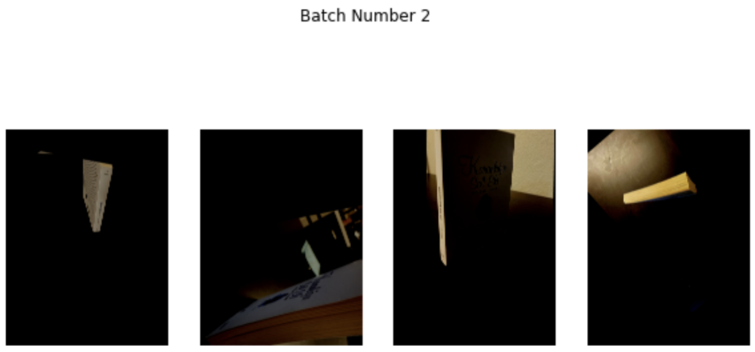


Figure 15: Second Batch of the DataLoader

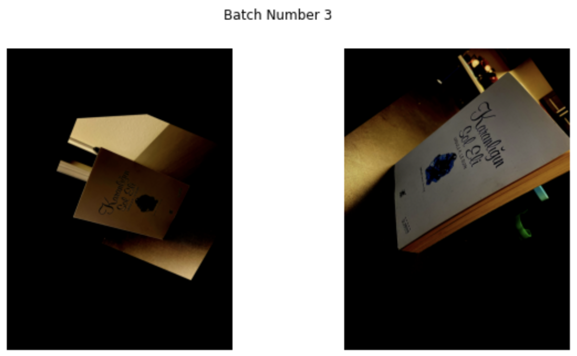
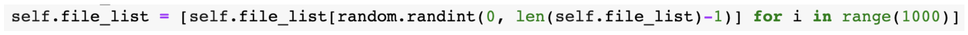


Figure 16: Third Batch of the DataLoader

As can be seen from the images, original book images are transformed into augmented forms. In addition to the geometric changes, light in the images has also changed.

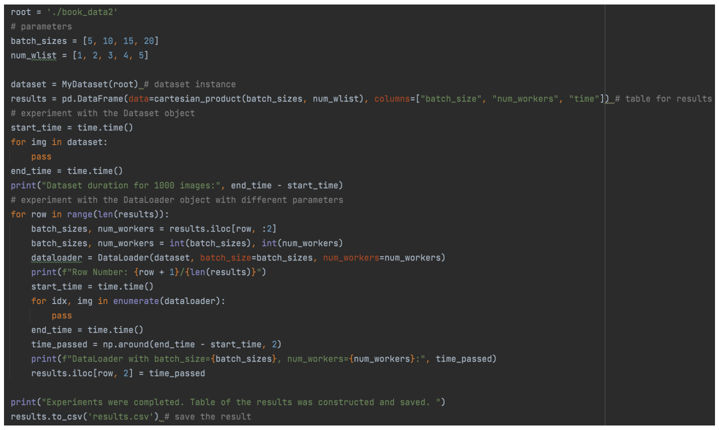
In the last part, the performance between the multi-threaded `DataLoader` v.s. just using `Dataset` classes were compared. To make that comparison, I have added the following code line to the class definition to fully utilize the dataloader.



```
self.file_list = [self.file_list[random.randint(0, len(self.file_list)-1)] for i in range(1000)]
```

Figure 17: Added Line to the MyDataset Class Definition

I choose 1000 random images from my dataset, in other words, I obtained 1000 image file names. I did this because if I did not do that, I would have 2 images at every third batch of the dataloader. So, dataloader's performance will not be fully demonstrated. First, I recorded the time needed to load and augment 1000 random images in your dataset by using `Dataset` class. Then, I compare its performance to the performance of `DataLoader` with different `batch_size` and `num_workers` settings. I created two lists for the parameters: `batch_size = [5,10,15,20]` and `num_workers = [1,2,3,4,5]`. I run all possible combinations so in total, I did 20 experiments with `DataLoader` over 1000 images. Lastly, in the previous sections, I used the raw image data that comes from my iPhone. So, the size of the images is 4032×3024. Thus, transformations of those images take a lot of time (around 1 hour for 1000 images). To reduce this time, I downscale the size of the images to $1024x784$. The implementation of this can be found in Figure 18.



```
root = './book_data2'
# parameters
batch_sizes = [5, 10, 15, 20]
num_wlist = [1, 2, 3, 4, 5]

dataset = MyDataset(root) # dataset instance
results = pd.DataFrame(data=cartesian_product(batch_sizes, num_wlist), columns=["batch_size", "num_workers", "time"]) # table for results
# experiment with the Dataset object
start_time = time.time()
for img in dataset:
    pass
end_time = time.time()
print("Dataset duration for 1000 images:", end_time - start_time)
# experiment with the DataLoader object with different parameters
for row in range(len(results)):
    batch_sizes, num_workers = results.iloc[row, :2]
    batch_sizes, num_workers = int(batch_sizes), int(num_workers)
    dataloader = DataLoader(dataset, batch_size=batch_sizes, num_workers=num_workers)
    print(f"Row Number: {row + 1}/{len(results)}")
    start_time = time.time()
    for idx, img in enumerate(dataloader):
        pass
    end_time = time.time()
    time_passed = np.around(end_time - start_time, 2)
    print(f"DataLoader with batch_size={batch_sizes}, num_workers={num_workers}:", time_passed)
    results.iloc[row, 2] = time_passed

print("Experiments were completed. Table of the results was constructed and saved. ")
results.to_csv('results.csv') # save the result
```

Figure 18: Dataset vs. DataLoader Comparison

I used `time` class as a stopwatch. I initialized the parameters and created the result as a `pandas.DataFrame` object, which saves the results for the `DataLoader` experiments. And in the for loops, I did not do anything because while iterating through the for loop, the program calls `dataset.__getitem__`, which is what we want to measure. At each iteration, I printed and saved the results to the table. After all the experiments are done, the table is saved as a `.csv` file to the current folder that the program is running. Results can be seen below:

Batch_size	**Num_workers**	**Time**
5.0	1.0	152.85
5.0	2.0	89.08
5.0	3.0	74.27
5.0	4.0	69.81
5.0	5.0	69.25
10.0	1.0	153.92
10.0	2.0	90.44
10.0	3.0	75.33
10.0	4.0	70.53
10.0	5.0	70.15
15.0	1.0	154.46
15.0	2.0	90.83
15.0	3.0	76.52
15.0	4.0	71.32
15.0	5.0	70.7
20.0	1.0	156.53
20.0	2.0	95.51
20.0	3.0	84.11
20.0	4.0	77.34
20.0	5.0	74.66

Table 1: Times that `DataLoader` processes 1000 random images with different hyperparameters.

The time that `Dataset` processes 1000 random images in my dataset was found to be `120.5`. From Table 1, we can say that using `num_workers=1` is worse than using `Dataset` instance. Because we just created one process and there is no multi-threading. In fact, wrapping `Dataset` instance with `DataLoader`, we lose more time. One important observation is that as the number of `num_workers` increases, processing time decreases nonlinearly. This also makes sense because `num_workers` is the number of subprocesses to load the data. So, we utilize multi-threading more as we increase the number of workers, which will result in faster processing. One important point about the number of workers is the following although it seems that `num_workers` and processing time are negatively correlated this is not the case in general. I have experimented with very large values of `num_workers` and I saw that too much `num_workers` results in slower processing times. I think this is because of the computer's hardware power. The more powerful the computer is, the more parallelization one can achieve. Another observation in Table 1 is for the constant `num_workers` but increasing `batch_size`, processing time increases. I think the reason for this is the more data we want to process at each iteration, the larger the queue for the processing, which slows the processing. But, there is not big loss between different batches and I think this can be considered as one benefit of using `DataLoader`. Lastly, all cases which utilize multi-threading are faster than `Dataset`. So, it should be used in the training or evaluation of the Deep Learning model.

```python
# Author: Mehmet Berk Sahin
# Email: sahinm@purdue.edu
# ID: 34740048

# imports
from torch.utils.data import DataLoader
import pandas as pd
import time
from PIL import Image
import numpy as np
import torchvision.transforms as tvt
import random
import torchvision.transforms.functional as F
import matplotlib.pyplot as plt
from scipy.stats import wasserstein_distance
from torch.utils.data import DataLoader
import os
import torch

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

    def __init__(self, root):
        super().__init__()
        # meta information
        self.file_list = os.listdir(root)
        if root == "./book_data2":
            self.file_list = [self.file_list[random.randint(0, len(self.file_list)-1)] for i in range(1000)]

        self.root_dir = root
        # transforms
        self.transforms = tvt.Compose([
            tvt.ToTensor(),
            tvt.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]),
            tvt.RandomAffine(degrees=(-90, 90), scale=(0.5, 0.75),
                             translate=(0.1, 0.3)),
            tvt.RandomPerspective(distortion_scale=0.5, p=0.5),
            tvt.ColorJitter(brightness=(0.3, 0.7))
        ])

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

    def __getitem__(self, index):
        # Return the augmented image at the corresponding index
        img_dir = os.path.join(self.root_dir, self.file_list[index])
        img = Image.open(img_dir)  # load the image
        img_aug = self.transforms(img)
        return (img_aug, random.randint(0, 10))


def cartesian_product(list1, list2):
    result = np.zeros(shape=(len(list1) * len(list2), 3))
    counter = 0
    for i in range(len(list1)):
        for j in range(len(list2)):
            result[counter, 0], result[counter, 1] = list1[i], list2[j]
            counter += 1
    return result



if __name__ == '__main__':

    # DISCLAIMER: DATASET FILE NAMES SHOULD BE "book_data" AND "book_data2"
    # OTHERWISE PROGRAM DOES NOT WORK PROPERLY. THE FORMER ONE INCLUDES OR-
    # IGINAL IMAGES FROM IPHONE. THE SECOND ONE IS DOWNSCALED VERSION.

    print("Packages were imported succesfully.")

    # For reproducibility seed = 0 (taken from weekly slides 2) !TAKEN FROM WEEK 2 SLIDES!
    seed = 3
    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

    ########### QUESTION 3.2 ################
    # read images as PIL objects
    img = Image.open("StopSign3.jpeg")  # normal image
    cimg = Image.open("StopSign2.jpeg")  # corrupted image (bad angle)
    # convert them to NumPy arrays
    img_np = np.array(img.getdata()).reshape(img.size[1], img.size[0], 3)
    cimg_np = np.array(cimg.getdata()).reshape(cimg.size[1], cimg.size[0], 3)
    # plot the images side-by-side
    fig, ax = plt.subplots(1, 2)
    ax[0].set_title("Front View")
    ax[0].axis('off')
    ax[0].imshow(img_np)
    ax[1].set_title("Oblique Angle")
    ax[1].axis('off')
    ax[1].imshow(cimg_np)
    plt.show()

    img = Image.open("StopSign3.jpeg")
    transformer = tvt.Compose([tvt.ToTensor(),
                               tvt.RandomAffine(degrees=0, scale=(1.2, 1.5),
                                                shear=(0, 15)),
                               tvt.ToPILImage()])
    # Affine transform
    aug_img = transformer(img)

    # Perspective transform

    # height, width = aug_img.size[1], aug_img.size[0]
    # height, width = random.randint(width, height), random.randint(width, height)
    # transform = tvt.RandomPerspective()
    # startpoints, endpoints = transform.get_params(height, width, 0.7)
    # print(startpoints)
    # print(endpoints)
    startpoints = [[0, 0], [3052, 0], [3052, 3899], [0, 3899]]
    endpoints = [[843, 1330], [2279, 540], [2505, 3789], [552, 3870]]
    new_img = F.perspective(aug_img, startpoints, endpoints)

    # same plotting procedure
    cimg_np = np.array(cimg.getdata()).reshape(cimg.size[1], cimg.size[0], 3)
    newimg_np = np.array(new_img.getdata()).reshape(new_img.size[1], new_img.size[0], 3)
    # plot the images side-by-side
    fig, ax = plt.subplots(1, 2)
    ax[0].set_title("Original Image")
    ax[0].axis('off')
    ax[0].imshow(cimg_np)

    ax[1].set_title("Transformed Image")
    ax[1].axis('off')
    ax[1].imshow(newimg_np)
    plt.show()

    # measuring the difference between histograms
    NUM_BINS = 10
    # creating arrays for histograms for each channel
    hist_img = torch.zeros(3, NUM_BINS, dtype=torch.float)
    hist_cimg = torch.zeros(3, NUM_BINS, dtype=torch.float)
    hist_augimg = torch.zeros(3, NUM_BINS, dtype=torch.float)

    pilToTens = tvt.PILToTensor()
    img = pilToTens(img).float()
    cimg = pilToTens(cimg).float()
    aug_img = pilToTens(new_img).float()

    for channel_idx in range(3):
        # stop sign front view
        hist_img[channel_idx] = torch.histc(img[channel_idx],
                                            bins=10, min=0.0, max=255.0)
        hist_img[channel_idx] = hist_img[channel_idx].div(hist_img[channel_idx].sum())

        # stop sign angled view
        hist_cimg[channel_idx] = torch.histc(hist_cimg[channel_idx],
                                             bins=10, min=0.0, max=255.0)
        hist_cimg[channel_idx] = hist_cimg[channel_idx].div(hist_cimg[channel_idx].sum())

        # stop sign augmented
        hist_augimg[channel_idx] = torch.histc(aug_img[channel_idx],
                                               bins=10, min=0.0, max=255.0)
        hist_augimg[channel_idx] = hist_augimg[channel_idx].div(hist_augimg[channel_idx].sum())

    for trial in range(2):
        if trial == 0:
            print("Comparison of front view and oblique angle view")
            other_hist = hist_img
        else:
            print("Comparison of oblique angle view and transformed image")
            other_hist = hist_augimg

        for channel_idx in range(3):
            dist = wasserstein_distance(hist_cimg[channel_idx].cpu().numpy(),
                                        other_hist[channel_idx].cpu().numpy())
            print(f"Wasserstein distance for channel {channel_idx}: {dist}")

    ########### QUESTION 3.3 ################
    root = './book_data'
    # example similar to the example in the assignment
    my_dataset = MyDataset(root)
    print(len(my_dataset))
    index = 4
    print(my_dataset[index][0].shape, my_dataset[index][1])
    index = 9
    print(my_dataset[index][0].shape, my_dataset[index][1])

    # choose three images randomly and display their original and augmented versions
    file_names = os.listdir(root)
```

```python
    # sample random numbers
    rand_idxs = torch.randint(0, 10, size=(3,))
    # plot configurations
    fig, ax = plt.subplots(3, 2)
    fig.set_size_inches(18.5, 10.5)
    fig.subplots_adjust(wspace=-.75, hspace=0.5)
    # go through random images
    for counter, idx in enumerate(rand_idxs):
        org_img = Image.open(os.path.join(root, file_names[idx]))  # load img
        aug_img = my_dataset[idx][0].numpy()  # take the augmented img as np
        aug_img = np.transpose(aug_img, (1, 2, 0))
        org_img = np.array(org_img.getdata()).reshape(org_img.size[1], org_img.size[0], 3)

        ax[counter, 0].axis('off')
        ax[counter, 0].imshow(org_img)
        ax[counter, 1].axis('off')
        ax[counter, 1].imshow(aug_img)
    plt.suptitle("Original Images vs. Augmented Images")
    plt.show()

    dataset = MyDataset('./book_data')
    dataloader = DataLoader(dataset, batch_size=4, num_workers=2)
    # go through the dataset via dataloader
    for idx, (img, label) in enumerate(dataloader):
        batch_size = img.shape[0]
        fig, ax = plt.subplots(1, batch_size)
        fig.set_size_inches(10, 5)
        fig.suptitle(f"Batch Number {idx+1}")
        # display images in batch
        for idx in range(batch_size):
            img_np = np.transpose(img[idx].numpy(), (1, 2, 0))
            ax[idx].axis('off')
            ax[idx].imshow(img_np)
        plt.show()

    ########### QUESTION 3.4 ###########
    root = './book_data2'
    # parameters
    batch_sizes = [5, 10, 15, 20]
    num_wlist = [1, 2, 3, 4, 5]

    dataset = MyDataset(root) # dataset instance
    results = pd.DataFrame(data=cartesian_product(batch_sizes, num_wlist), columns=["batch_size", "num_workers", "time"]) # table for results
    # experiment with the Dataset object
    start_time = time.time()
    for img in dataset:
        pass
    end_time = time.time()
    print("Dataset duration for 1000 images:", end_time - start_time)

    # experiment with the DataLoader object with different parameters
    for row in range(len(results)):
        batch_sizes, num_workers = results.iloc[row, :2]
        batch_sizes, num_workers = int(batch_sizes), int(num_workers)
        dataloader = DataLoader(dataset, batch_size=batch_sizes, num_workers=num_workers)
        print(f"Row Number: {row + 1}/{len(results)}")
        start_time = time.time()
        for idx, img in enumerate(dataloader):
            pass
        end_time = time.time()
        time_passed = np.around(end_time - start_time, 2)
        print(f"DataLoader with batch_size={batch_sizes}, num_workers={num_workers}:", time_passed)
        results.iloc[row, 2] = time_passed

    print("Experiments were completed. Table of the results was constructed and saved. ")
    results.to_csv('results.csv') # save the result
```