ECE60146: Homework 2

Manish Kumar Krishne Gowda, 0033682812

(Spring 2024)

1 Introduction

In this assignment,programming tasks related to torchvision, random tensors, numpy and tensor interconversion, histograms, image transformations, etc. were performed using Python. It is intended to provide an insight on image handling and image processing using pytorch module. The modules covered in this homework will provide a base for working with deep neural networks like converting images to tensor data type (which is the default pytorch format for working with deep neural networks), manipulating the available image input to better train the network, etc.

2 Understanding Pixel Value Scaling and Normalization

```
from google.colab import drive
drive.mount('/content/drive')
%cd /content/drive/MyDrive/ece60146_hw2/

from PIL import Image # pillow fork
import os
import time
import torch
import torchvision.transforms as tvt
import numpy as np
import random #for random seed
import matplotlib.pyplot as plt #matplotlib for visualisation
from scipy.stats import wasserstein_distance

def manual_scaling(images):
  images_scaled = images/images.max().float()
  return images_scaled

def tvt_scaling(images):
  images_scaled = torch.zeros_like(images).float()
  for i in range(images.shape[0]):
    images_scaled[i] = tvt.ToTensor()(np.transpose(images[i].numpy(), (1,2,0)))
  return images_scaled

images_v1 = torch.randint(0, 32, (4, 3, 5, 9)).type(torch.uint8) #version1 images
                                          -> pixel values limited to (0,32)
images_v2 = torch.randint(0, 256, (4, 3, 5, 9)).type(torch.uint8) #version2 images
                                           -> pixel values in full one byte range

#v1 manual pixel scaling vs automated (tvt.ToTensor) comparison
manual_images_v1 = manual_scaling(images_v1)
auto_images_v1 = tvt_scaling(images_v1)
print("v1 image first batch comparison output : ")
print((manual_images_v1[0] == auto_images_v1[0]))

#v2 manual pixel scaling vs automated (tvt.ToTensor) comparison
manual_images_v2 = manual_scaling(images_v2)
auto_images_v2 = tvt_scaling(images_v2)
print("v2 image first batch comparison output : ")
```

```
print((manual_images_v2[0] == auto_images_v2[0]))
```

As per the task instructions manual pixel-value scaling using **max()** function is compared with a more "automated" pixel-value scaling as provided by **tvt.ToTensor()** method. Two different versions **images_v1** and **images_v2** of a simulated batch of images were created using **torch.randint** method. Here **images_v1** image pixels are limited to the range 0 through 31, while **images_v2** image pixels spans the full one-byte range (i.e. 0 through 255). For both these images, values we get with the manual approach of scaling with the values we get with **tvt.ToTensor** approach of scaling are compared. Specifically, the function **manual_scaling()** performs scaling of former type, using the maximum pixel value in the image (obtained using **max()** function). The function **tvt_scaling()** performs the latter automated version of scaling.

The output of the manual scaling comparison is shown in Figure 1 while output of the automated scaling comparison is shown in Figure 2

```
v1 image first batch comparison output :
tensor([[[False, False, False, False, False, False, False, False, False],
         [False, False, False, False, False,  True, False, False, False],
         [False, False, False, False, False, False, False, False, False],
         [False, False, False, False, False, False, False, False, False],
         [False, False, False, False,  True, False, False, False, False]],

        [[False, False, False, False, False, False, False, False, False],
         [False, False, False, False, False, False, False, False, False],
         [False, False, False, False, False, False, False,  True, False],
         [False, False, False, False, False, False, False, False, False],
         [False, False, False, False, False, False, False, False, False]],

        [[False, False, False, False, False, False, False, False, False],
         [False, False, False, False, False, False, False, False, False],
         [False, False,  True, False, False, False, False, False, False],
         [False, False, False, False, False, False, False, False, False],
         [False, False, False, False, False,  True, False, False, False]]])
```

Figure 1: Low Illumination Image Comparison Results

For the Low Illumination Image (i.e. image with pixels restricted to the range (0,32)), the manual and automated scaling yield different results in each pixel (except in the pixels that are equal to zero in the original image, the scaling in both cases will lead to a value of zero. Thus we can see sparse "True" values in the comparision output). This is because, while **max()** function scales the image by the maximum pixel value in the image matrix, the **tvt.ToTensor** always scales by 255, irrespective of the max value. Similarly the Full pixel range Image Comparison Results yields "True" in all pixel comparisions. But, its interesting to note that for the **images_v1** matrix which this comparion is based on, the max pixel value is 255. If the **torch.randint** function samples a matrix where the max pixel value is not 255 (say 254 or 253 or less), then the scaling in the two types will yield different results. To show the output in minimal space, only the first channel of each image is compared.

2.1 Investigation of provided npy image

```
test_img_np_array = np.load('images/test_image.npy')
plt.imshow(test_img_np_array)
```

```
v2 image first batch comparison output :
tensor([[[True, True, True, True, True, True, True, True, True],
         [True, True, True, True, True, True, True, True, True],
         [True, True, True, True, True, True, True, True, True],
         [True, True, True, True, True, True, True, True, True],
         [True, True, True, True, True, True, True, True, True]],

        [[True, True, True, True, True, True, True, True, True],
         [True, True, True, True, True, True, True, True, True],
         [True, True, True, True, True, True, True, True, True],
         [True, True, True, True, True, True, True, True, True],
         [True, True, True, True, True, True, True, True, True]],

        [[True, True, True, True, True, True, True, True, True],
         [True, True, True, True, True, True, True, True, True],
         [True, True, True, True, True, True, True, True, True],
         [True, True, True, True, True, True, True, True, True],
         [True, True, True, True, True, True, True, True, True]]])
```

Figure 2: Full pixel range Image Comparison Results

```
print(f'max value in given npy image = {np.amax(test_img_np_array)}')  #print max
                                        value in the np array
print(f'min value in given npy image = {np.amin(test_img_np_array)}')  #print min
                                        value in the nd array

images_scale_max = manual_scaling(torch.from_numpy(np.transpose(test_img_np_array
                                        ,(2,0,1)))) #manual scaling scales the
                                        image by the max value i.e. 219
images_scale_255 = tvt.ToTensor()(test_img_np_array) #avoiding tvt_scaling
                                        function as it iterates through the
                                        batches

print("manual scaling output : ")
print(images_scale_max[0])
print()
print("tvt.ToTensor scaling output : ")
print(images_scale_255[0])

#alternatively we can use foll. lines for direct operations on np array
#print(test_img_np_array[0]/np.amax(test_img_np_array))
#print(test_img_np_array[0]/255)s
```

The given image was investigated and the results are shown in Figure 3. In order to perform the scaling of the given npy matrix by the max value and again by the max possible value (i.e. 255), the given numpy matrix was transformed to the tensor datatype using **from_numpy** inbuilt method and the function logics defined for the previous task was reused. The observed results are consistent with the scaling explained in previous section.

In most practical image datasets encountered in deep learning, the images will span a full range (0,255) for each channel. Hence both manual scaling and tvt.ToTensor based scaling can be used. However, the **tvt.ToTensor** based scaling is more efficient and is preferred as we work with

large image datasets.

3 Programming Tasks

3.1 Setting Up Your Conda Environment

This task is skippped as I am working with google colab. Google Colab comes with many pre-installed libraries and packages, including some commonly used ones in the data science and machine learning community. I can use the built-in package management system in Colab to install additional packages using commands like **!pip install**

3.2 Becoming Familiar with torchvision.transforms

```
#3.2 Becoming Familiar with torchvision.transforms
calculator_direct = Image.open('images/calculator_direct.jpeg')
calculator_oblique = Image.open('images/calculator_oblique.jpeg')
my_bins = 10
fig, axes = plt.subplots(1, 2, figsize=(6, 5), sharey=False)
axes[0].imshow(calculator_direct)
axes[1].imshow(calculator_oblique)
plt.show()

#calculating histograms
hist_calculator_direct = torch.histc(tvt.ToTensor()(calculator_direct), bins=
                                     my_bins, min=0.0, max=1.0)
hist_calculator_direct = hist_calculator_direct.div(hist_calculator_direct.sum())
hist_calculator_oblique = torch.histc(tvt.ToTensor()(calculator_oblique), bins=
                                     my_bins, min=0.0, max=1.0)
hist_calculator_oblique = hist_calculator_oblique.div(hist_calculator_oblique.sum
                                     ())

no_transfomer = tvt.RandomAffine(degrees=(0,0))
img_without_transform = no_transfomer(calculator_oblique)

x_shear = np.linspace(-40, 40, 9)
y_shear = np.linspace(-40, 40, 9)
min_affine_dist = 1e10 #setting value to inf for initial comparison

#calculating Wasserstein Dist
def cal_wasserstein_dist(calculator_trans_img, num_bins = 10):
  hist_calculator_trans_img = torch.histc(tvt.ToTensor()(calculator_trans_img),
                                     bins=num_bins, min=0.0, max=1.0)
  hist_calculator_trans_img = hist_calculator_trans_img.div(
                                     hist_calculator_trans_img.sum())
  w_dist = wasserstein_distance(hist_calculator_direct.cpu().numpy(),
                                     hist_calculator_trans_img.cpu().numpy()
                                     )
  return w_dist,hist_calculator_trans_img

#Looping throgh different degrees and shear values
for degree in range(-60,60,20):
  for i, x_shear_idx in enumerate(x_shear):
    for j, y_shear_idx in enumerate(y_shear):
      affine_transformer = tvt.RandomAffine(degrees=(degree,degree), translate=(0,
                                          0),scale=(1,1),shear = [x_shear_idx
```

```
                                                , x_shear_idx+5, y_shear_idx,
                                                y_shear_idx+5])
      transformed_img = affine_transformer(calculator_oblique)
      w_dist,hist_calculator_trans_img = cal_wasserstein_dist(transformed_img,
                                                num_bins=my_bins)
      if w_dist < min_affine_dist:
        min_affine_dist = w_dist
        best_affine_img = transformed_img
        selected_deg = degree
        selected_x_shear = x_shear_idx
        selected_y_shear = y_shear_idx
        hist_trans_img_affine = hist_calculator_trans_img

min_pers_dist = 1e9
W, H = calculator_direct.size
for i in range(15):
  startpoints, endpoints = tvt.RandomPerspective().get_params(W, H, 0.3)
  transformed_img = tvt.functional.perspective(calculator_direct, startpoints,
                                            endpoints)
  w_dist,hist_calculator_trans_img = cal_wasserstein_dist(transformed_img,num_bins
                                            =my_bins)
  # Comparing the Wassertein distance with the minimum value
  if w_dist < min_pers_dist:
    min_pers_dist = w_dist
    best_pers_img = transformed_img
    selected_startpoints = startpoints
    selected_endpoints = endpoints
    hist_trans_img_pers = hist_calculator_trans_img

w_dist_obliqe,_ = cal_wasserstein_dist(calculator_oblique,num_bins=my_bins)
print("the selected affine transformer parameters are : degree={}, x_shear=[{},{}]
                                            & y_shear=[{},{}]".format(selected_deg,
                                           selected_x_shear,selected_x_shear+5,
                                           selected_y_shear,selected_y_shear+5))
print("the selected perspective transformer parameters are : startpoints={}
                                           endpoints={}".format(selected_startpoints
                                           ,selected_endpoints))
print("wasserstein_dist bw orig direct image and orig oblique img = {}".format(
                                           w_dist_obliqe))
print("wasserstein_dist bw orig direct image and affine transformed img = {}".
                                           format(min_affine_dist))
print("wasserstein_dist bw orig direct image and perspective transformed img = {}"
                                           .format(min_pers_dist))
# Plot Best Image after Affine Transform
fig, axes = plt.subplots(1, 4, figsize=(8, 5), sharey=True)
axes[0].imshow(calculator_direct)
axes[0].set_title('Orig Direct Img')
axes[1].imshow(calculator_oblique)
axes[1].set_title('Orig Oblique Img')
axes[2].imshow(best_affine_img)
axes[2].set_title('Best Aff-Trans Img')
axes[3].imshow(best_pers_img)
axes[3].set_title('Best Persp-Trans Img')
plt.show()

#histogram plots
x = range(my_bins)
fig, axes = plt.subplots(1, 4, figsize=(8, 5), sharey=False)
axes[0].bar(x, hist_calculator_direct, align='center')
```

```
axes[1].bar(x, hist_calculator_oblique, align='center')
axes[2].bar(x, hist_trans_img_affine, align='center')
axes[3].bar(x, hist_trans_img_pers, align='center')

axes[0].set_title('Hist Direct Img')
axes[1].set_title('Hist Oblique Img')
axes[2].set_title('Hist Affine. Img')
axes[3].set_title('Hist Persp. Img')
```

An image of a calculator placed against a black wall was used for this task. Sallable instance **tvt.RandomAffine** and the function **tvt.functional.perspective()** mentioned on Slides 46 and 47 of Week 2 were experimented with to find the best transformed version of the original image.

For affine transformation, different degrees and shear values were tested in a for loop. Translation and scale values were kept default, as the oblique image can be found to be almost similar size to the original image.

For Perspective transformation, the function **RandomPerspective** was used to sample random start and end points for the original image and the perspective transformation was obtained.

In both cases Wasserstein distance was used as comparision metric between the original image and the transformed image. Finally, the transformed image with the least Wasserstein distance among the all the transformed images was taken to be the best image for each type. The affine transformer parameters of selected shear and degrees, the perspective transformer parameters of the selected start and end points and the selected Wasserstein distance of the final selected image are captured and reported. Note that the startpoints and endpoints for Perspective transformer is a List containing [top-left, top-right, bottom-right, bottom-left] for the original image and transformed image respectively. Further, the histogram of the images are calculated using **torch.histc** function. All Affine Homographies are Projective Homographies, but not the other way around. This could explain a possible reason for better performance of the perspective transformer. The results are shown in Figure 4 and Figure 5.

3.3 Creating Your Own Dataset Class

```
#3.3 Creating Your Own Dataset Class
class MyDataset(torch.utils.data.Dataset):
  def __init__(self, root):
    super().__init__()
    # Obtain file names
    # perform data augmentation transforms, etc.
    self.root_dir = root
    self.image_paths = os.listdir(self.root_dir)
    self.transforms = tvt.Compose([tvt.ToTensor(),
                                   tvt.RandomGrayscale(p=0.5),
                                   tvt.RandomResizedCrop(256, scale=(0.9, 1.0)),
                                   tvt.GaussianBlur(5, sigma=(0.5, 2.0))])
  def __len__(self):
    # Return the total number of images
    return len(self.image_paths)

  def __getitem__(self, index):
    # Read an image at index and perform augmentations
    # Return the tuple : ( augmented tensor , integer label )
    # Get the path of the image
    # As we have only 10 images, we used "index % 10" to cover the cases when
                                                index >= 10
    index = index % len(self.image_paths)
```

```
    image_name = self.image_paths[index]
    full_image_path = os.path.join(self.root_dir, image_name)
    my_image = Image.open(full_image_path)
    transformed_image = self.transforms(my_image)
    return (transformed_image, index)

#test input demo
my_dataset = MyDataset(root = '/content/drive/MyDrive/ece60146_hw2/images/
                                        ten_images')
print(len(my_dataset))
index = 10
print(my_dataset[index][0].shape, my_dataset[index][1])
index = 55
print(my_dataset[index][0].shape, my_dataset[index][1])

# PLot three original images with augement versions
fig, axes = plt.subplots(3, 2, figsize=(10, 12), sharey=True)
rand_indices = np.random.randint(0, len(my_dataset), 3) #obtain 3 random images
                                        from our image set
for i in range(0,3):
  index = rand_indices[i]
  img, label = my_dataset[index]
  original_image = Image.open(os.path.join(my_dataset.root_dir, my_dataset.
                                          image_paths[index]))
  resized_image = original_image.resize((256, 256))
  axes[i][0].imshow(resized_image)
  axes[i][0].set_title('Original Image')
  axes[i][1].imshow(np.array(img).transpose(1,2,0))
  axes[i][1].set_title('Augmented Image')
```

10 images were captured using camera and are presented in the report folder. Three transforms, namely **RandomResizedCrop**, **RandomGrayscale**, and **GaussianBlur** were used on the original images. The first is a popular transform used to normalize the input image size while training neural networks. The other two are used to add random noise effects on the original images, providing addinal data to train our network on, so that it persorms well on unseen inputs. Output of 3 images are shown in Figure 6

3.4 Generating Data in Parallel

```
batch_size = 4
my_dataloader = torch.utils.data.DataLoader(dataset=my_dataset, batch_size=
                                        batch_size, shuffle=True, num_workers = 2
                                        )
iterator = iter(my_dataloader) #iter is a special function defined in Dataloader
                                        class, hence it is not overridden in
                                        MyDataset class
batch = next(iterator)
fig, axes = plt.subplots(1, batch_size, figsize=(12, 8), sharey=True)
for i in range(0,batch_size):
  image_in_batch = batch[0][i]
  axes[i].imshow(np.array(image_in_batch).transpose(1,2,0))
plt.show()

#comparing the performance gain by using the multi-threaded DataLoader v.s. just
                                        using Dataset
num_iters = 1000
batch_size_list = [32,64]
```

```
num_workers_list = [2, 4]

#performance gain by using the MyDataset __get_item__
rand_indices = np.random.randint(len(my_dataset), size = num_iters)
start_time = time.time()
for i in rand_indices:
  rand_image, label = my_dataset[i]
end_time = time.time()
elapsed_time = end_time - start_time
print(f'Time taken to process {num_iters} images in the dataset using __get_item__
                                        : {elapsed_time} seconds\n')

#performance gain by using the multi-threaded DataLoader
for bsize in batch_size_list :
  for nworkers in num_workers_list:
    print(f'Performance of dataloader with batch size {bsize} and {nworkers}
                                            num_workers:')
    dataloader = torch.utils.data.DataLoader(dataset=my_dataset,
                batch_size=bsize, shuffle=True, num_workers = nworkers)
    iterator = iter(dataloader)
    start_time = time.time()
    for i in range(int(num_iters/bsize)):
      try:
        image, label = next(iterator)
      except StopIteration:
        iterator = iter(dataloader)
        image, label = next(iterator)
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f'Time taken to process {num_iters} images in the dataset using
                                            Dataloder: {elapsed_time} seconds]\n'
                                            )
```

Output of 4 images together from the same batch as returned by the dataloader is shown in Figure 7. Time taken to process 1000 images in the dataset using __get_item__ was found to be 83.28334498405457 seconds. The Time taken to process 1000 images using Dataloader class iterator is reported in the Table. It is observed that the processing time decreases when using a larger batch size. Additionally, it should be noted that if the number of threads used for processing exceeds the batch size, there is a decrease in performance. The batch size use will be more apparant with larger image sizes.

batch_size	num_workers	time taken
32	2	35.016271
32	4	36.15959
64	2	16.19922
64	4	17.402995

3.5 Random Seed

```
batch_size = 2
dataloader = torch.utils.data.DataLoader(my_dataset, batch_size = batch_size,
                                        shuffle = True)

#Plot the first batch of images
for batch in dataloader :
  images,labels = batch
```

```
  fig, axes = plt.subplots(1, batch_size, figsize=(8, 8), sharey=True)
  for i in range(0,batch_size):
    image_in_batch = images[i]
    axes[i].imshow(np.array(image_in_batch).transpose(1,2,0))
  plt.show()
  break

# Rerun the iterator
for batch in dataloader :
  images,labels = batch
  fig, axes = plt.subplots(1, batch_size, figsize=(8, 8), sharey=True)
  for i in range(0,batch_size):
    image_in_batch = images[i]
    axes[i].imshow(np.array(image_in_batch).transpose(1,2,0))
  plt.show()
  break

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
os.environ['PYTHONHASHSEED'] = str(seed)

batch_size = 2
dataloader = torch.utils.data.DataLoader(my_dataset, batch_size = batch_size,
                                         shuffle = True)
#Plot the first batch of images
for batch in dataloader :
  images,labels = batch
  fig, axes = plt.subplots(1, batch_size, figsize=(8, 8), sharey=True)
  for i in range(0,batch_size):
    image_in_batch = images[i]
    axes[i].imshow(np.array(image_in_batch).transpose(1,2,0))
  plt.show()
  break


# Rerun the iterator
for batch in dataloader :
  images,labels = batch
  fig, axes = plt.subplots(1, batch_size, figsize=(8, 8), sharey=True)
  for i in range(0,batch_size):
    image_in_batch = images[i]
    axes[i].imshow(np.array(image_in_batch).transpose(1,2,0))
  plt.show()
  break
```

In this task the use of Random seed is investigated. For both cases the two images obtained in subsequent trials of the iterator from the same batch is different. This is because in the second sample of the dataloader, the dataloader shuffles the input and returns a different sample of the 1000 random images. The use of "random_seed" however is to reproduce the results across different platforms. For example, an iterator call with the random seed enabled, immediately after the seed set will output the same images. This is briefly shown in the code. The output of the iterator with and without seed setting can be seen in Figure 8 and 9.

4 Conclusion

Data augmentation proves beneficial for expanding the training dataset, particularly in scenarios with limited training data. The study reveals successful mapping of straight images to oblique ones through Projective transformation. Moreover, custom datasets can be tailored to specific data formats and efficiently processed in mini-batches using an appropriate batch size and number of workers within a Dataloader class. This not only accelerates the training process but also enhances its stability.

```
max value in given npy image = 219
min value in given npy image = 0
manual scaling output :
tensor([[0.6164, 0.6210, 0.6256,  ..., 0.7671, 0.5936, 0.6256],
        [0.6164, 0.6164, 0.6210,  ..., 0.8402, 0.6256, 0.6301],
        [0.6301, 0.6393, 0.6484,  ..., 0.9132, 0.6484, 0.6119],
        ...,
        [0.5936, 0.5936, 0.5936,  ..., 0.5571, 0.5525, 0.5525],
        [0.5342, 0.5434, 0.5571,  ..., 0.5525, 0.5616, 0.5434],
        [0.6027, 0.5936, 0.5936,  ..., 0.5616, 0.5753, 0.5342]])

tvt.ToTensor scaling output :
tensor([[0.5294, 0.5333, 0.5373,  ..., 0.6588, 0.5098, 0.5373],
        [0.5294, 0.5294, 0.5333,  ..., 0.7216, 0.5373, 0.5412],
        [0.5412, 0.5490, 0.5569,  ..., 0.7843, 0.5569, 0.5255],
        ...,
        [0.5098, 0.5098, 0.5098,  ..., 0.4784, 0.4745, 0.4745],
        [0.4588, 0.4667, 0.4784,  ..., 0.4745, 0.4824, 0.4667],
        [0.5176, 0.5098, 0.5098,  ..., 0.4824, 0.4941, 0.4588]])
```

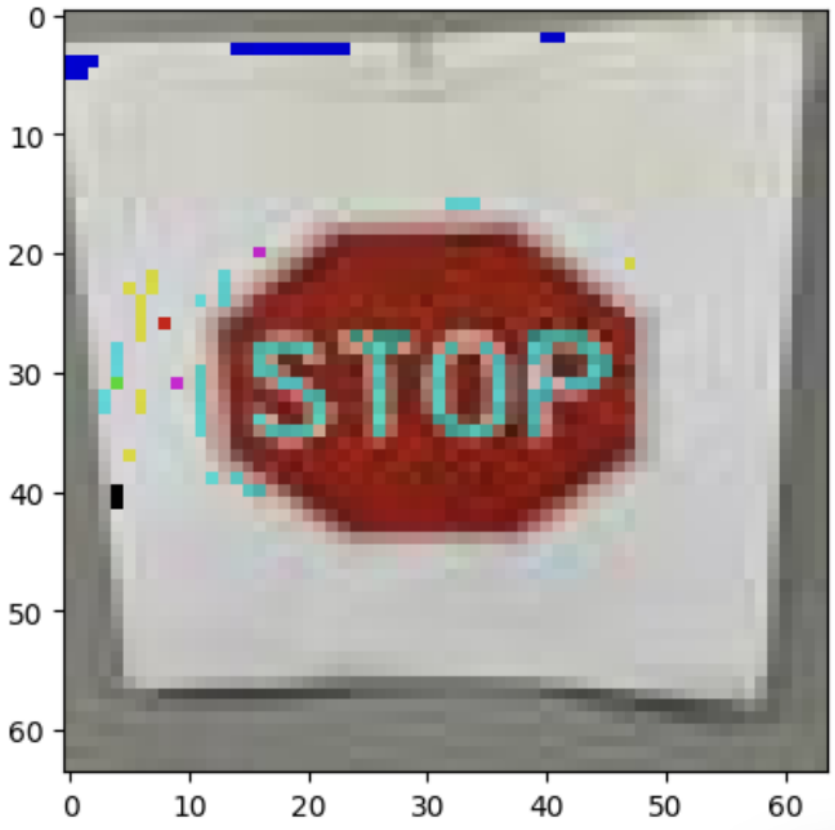

Figure 3: Given npy file investigation results (Task 2.1)

```
the selected affine transformer parameters are : degree=-20, x_shear=[20.0,25.0] & y_shear=[-20.0,-15.0]
the selected perspective transformer parameters are : startpoints=[[0, 0], [1199, 0], [1199, 1599], [0, 1599]]
                                                     endpoints=[[22, 61], [1187, 211], [1171, 1592], [63, 1507]]
wasserstein_dist bw orig direct image and orig oblique img = 0.02238031635060906
wasserstein_dist bw orig direct image and affine transformed img = 0.022371878009289498
wasserstein_dist bw orig direct image and perspective transformed img = 0.01571083981543779
```

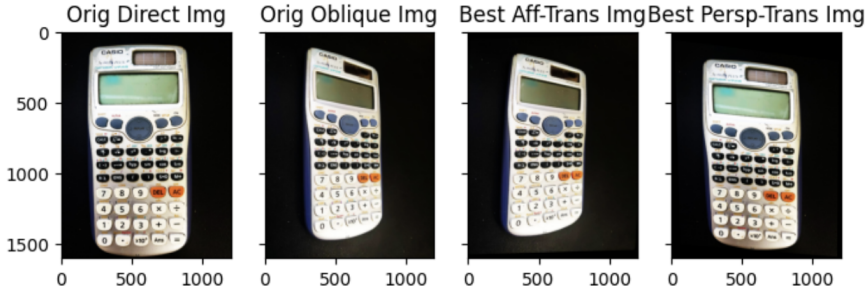

Figure 4: Original and Transformed Images

```
Text(0.5, 1.0, 'Hist Persp. Img')
```

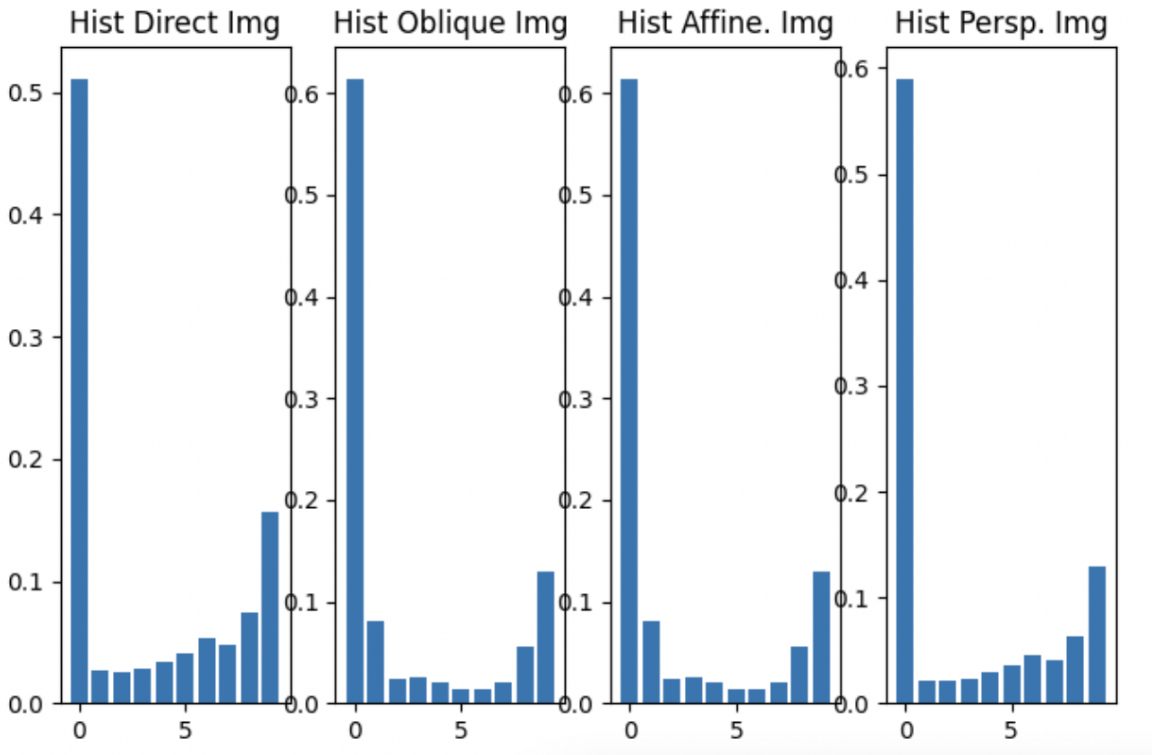

Figure 5: Histograms of the Original and Transformed Images

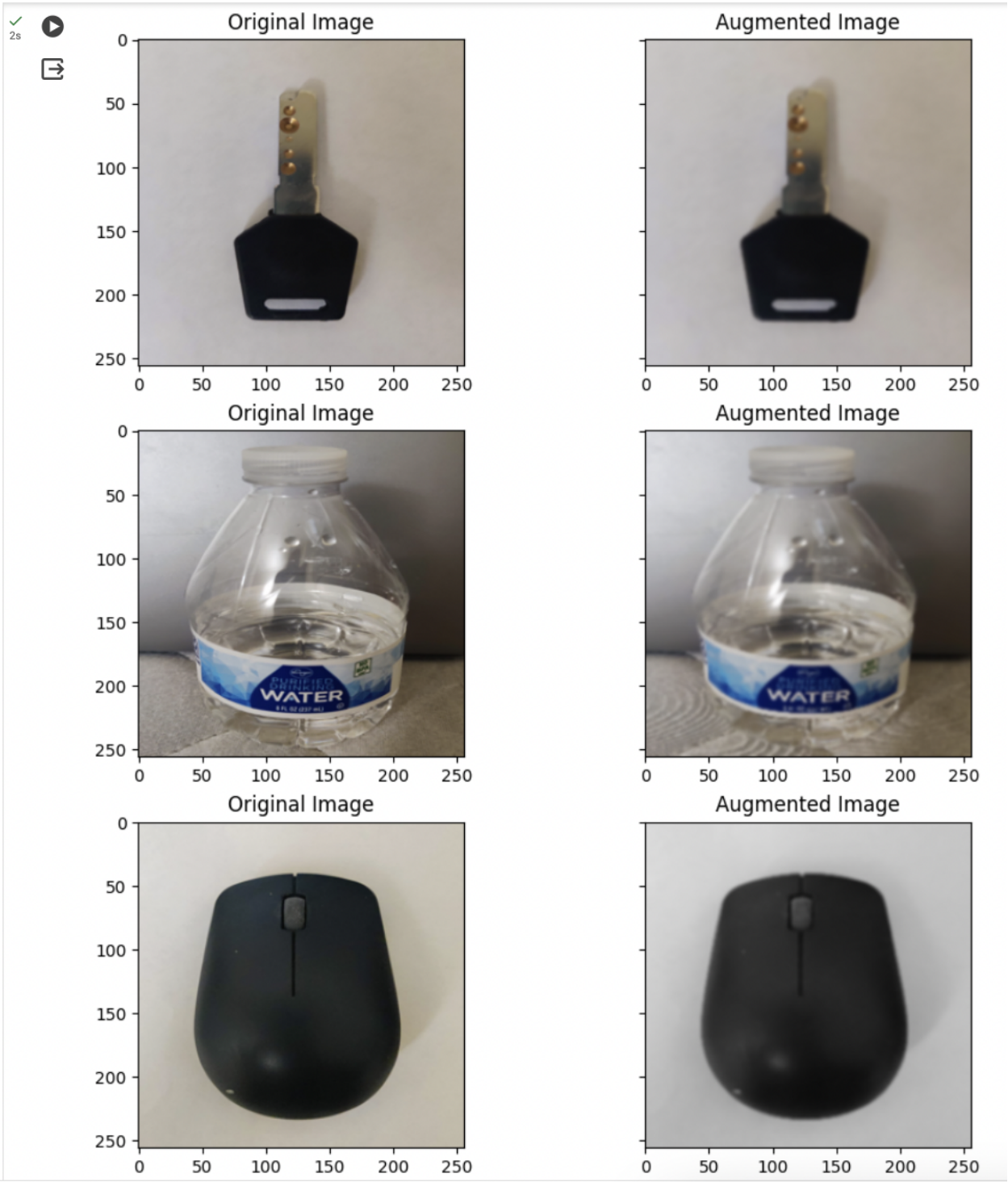

Figure 6: Output of original and augmented images

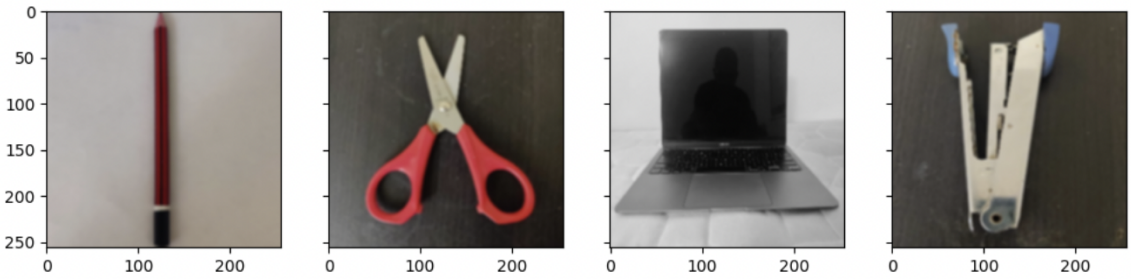

Figure 7: 4 images of same batch

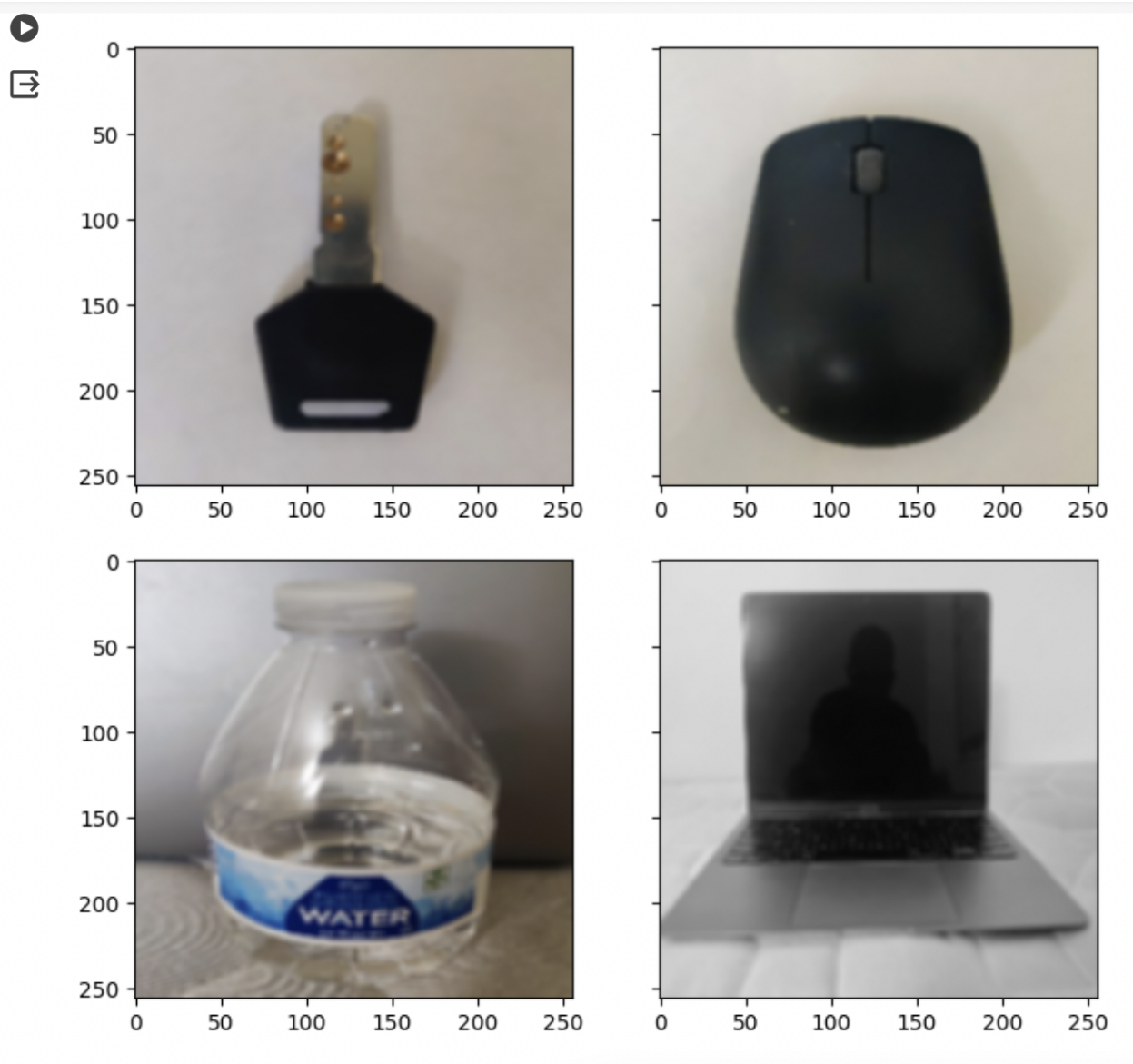

Figure 8: Image output without Random Seed

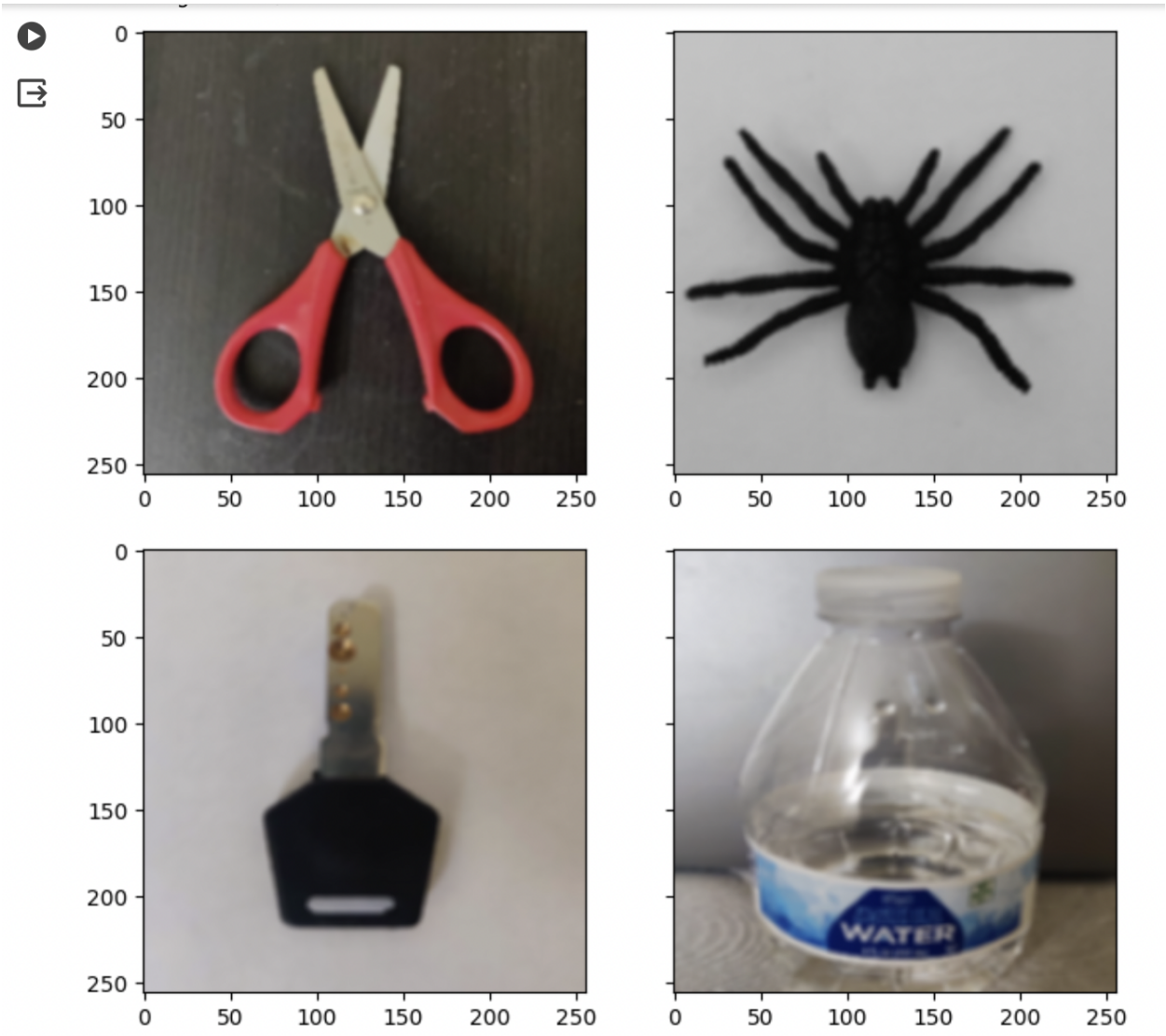

Figure 9: Image output with Random Seed set