Homework 2 - Deep Learning (ECE 60146)

Souradip Pal
pal43@purdue.edu
PUID 0034772329

January 25, 2023

1 Introduction

In this homework, the problems give an overview of how to implement an image data loader using PyTorch and understand the different image representations like PIL, tensor, numpy, etc. It also gives an overview of the data augmentation process using several transformations available in *torchvision* library. Wasserstein distance was also introduced to measure the distance between image histograms for comparing two images.

2 Theory Question

2.1 Task 2: Understanding Data Normalization

In this theory question, it is required to explain why the batch of scaled images was obtained using different methods - (i) by dividing the pixel values in all the batch images by the max pixel value in any of the images in the batch and (ii) using *tvt.ToTensor()* in a *for-loop* for each image separately, produces the same result as per Slides 12-36 presented by Prof. Kak. The main reason why the pixel values of the scaled images turned out to be the same in this particular case is due to the fact that the max pixel value in one of the images in the batch was **255**. By default, *tvt.ToTensor()* divides the pixel values by **255** to scale them between 0 to 1.0 as given in this link. Thus the max pixel value used for scaling the images in both the approaches were same in this case. If the max pixel value in any one of the images in the batch would not have been **255**, then the two approaches would have given different results.

3 Implementation and Results

3.1 Task 3: Programming Tasks

3.1.1 Task 3.1

A python virtual environment was setup using conda as per the instructions and necessary packages were installed. The *environment.yml* file has been submitted in the zip file. [Note: The tasks were mostly performed in Google Collaboratory. However, the codes were also run on the local machine with the mentioned conda environment.]

3.1.2 Task 3.2

- Firstly, a photo of a stop sign was taken standing directly in front of the sign with the camera straight as shown in Fig 1.
- Next, another photo of the stop sign was taken by pointing the camera towards the stop sign but standing off to a side as shown in Fig 1
- To transform one image of the stop sign to an oblique one, the first image was transformed by calling the *tvt.RandomAffine()* callable instance with different parameters. The values of the following parameter were varied in a loop as mentioned (i) degree = $(-5, 5)$, (ii) translate = $[0.0, 0.1, 0.2, 0.3, 0.4]$ and (iii) shear = $[-60, -45, ..., 45, 60]$. The similarity between each of the transformed images and the second image was calculated by taking the L2 norm of the Wasserstein distance of the three color channels as presented in Slides 65 to 73. The image which had the least distance was then identified as the best-transformed image. Fig 2 shows the best-transformed image obtained using the Affine transformation.
- Similarly, the first image was transformed by calling *tvt.function.perspective()* function to get different projective transforms of the straight image. Before calling *tvt.function.perspective()*, the method *tvt.RandomPerspective().get_params()* was called for a number of iterations with a distortion scale of 0.3 to get the various random combinations of start and endpoint which were then used in the function *tvt.function.perspective()*. The similarity was also calculated using Wasserstein distance. The image which had the least distance was then identified as the best-transformed image. Fig 3 shows the best-transformed image obtained using the Projective transformation.
- The image obtained after Projective transformation is visually quite similar to the original image taken at an oblique angle. The edges of the stop sign in the original straight image are parallel to each other but in the image taken from the oblique angle, it is not. Thus using Affine transformation it is not possible to get a transformed image similar to the oblique image since parallel lines remain parallel in Affine transformation. However, using Projective transformation, a visually similar image was obtained. It is to be noted that the Wasserstein distance for the image with the best Projective transform exceeds that of the Affine one because of the fact that the projective image had many zero pixels which increased the distance measure based on the histogram.

3.1.3 Task 3.3

- Firstly, 10 images were captured and stored in a single directory. These images act as the data for the custom dataset. A custom dataset class called *MyDataset* was implemented using PyTorch's built-in *torch.utils.data.Dataset* base class. In its constructor, some meta information was stored like the image paths, root directory path, and the transforms chosen for data augmentation. Here, three data augmentation transforms *tvt.RandomHorizontalFlip()*, *tvt.RandomResizedCrop()* and *tvt.GaussianBlur()* was chosen along with *tvt.ToTensor()* and were chained using *tvt.Compose()*.
- Moreover, the *__get_item__* method in the class was implemented where each image was loaded from the disc and a tuple of the image tensor and a random label was returned after applying the augmentation transforms. Shown in Fig 4 is the plot of three images along with their transformed versions.
- The transform *tvt.RandomHorizontalFlip()* was chosen as it will generally add more images that are flipped so that the model understands that flipped images generally contain the same object. *tvt.RandomResizedCrop()* crops the image randomly with a scale factor between 0.8 and 1 and resizes images to **(256, 256)** as set in the size parameter so that each image can be processed

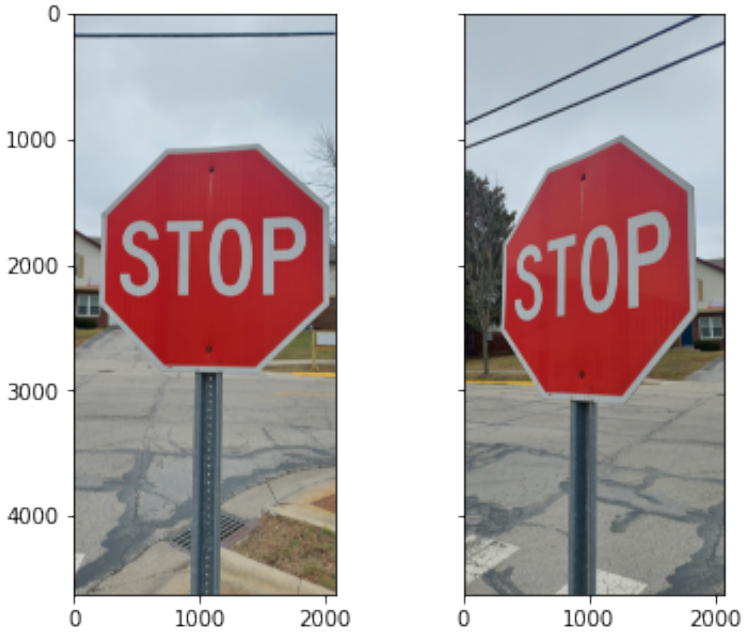

Figure 1: Images of stop sign taken in straight and oblique angles respectively

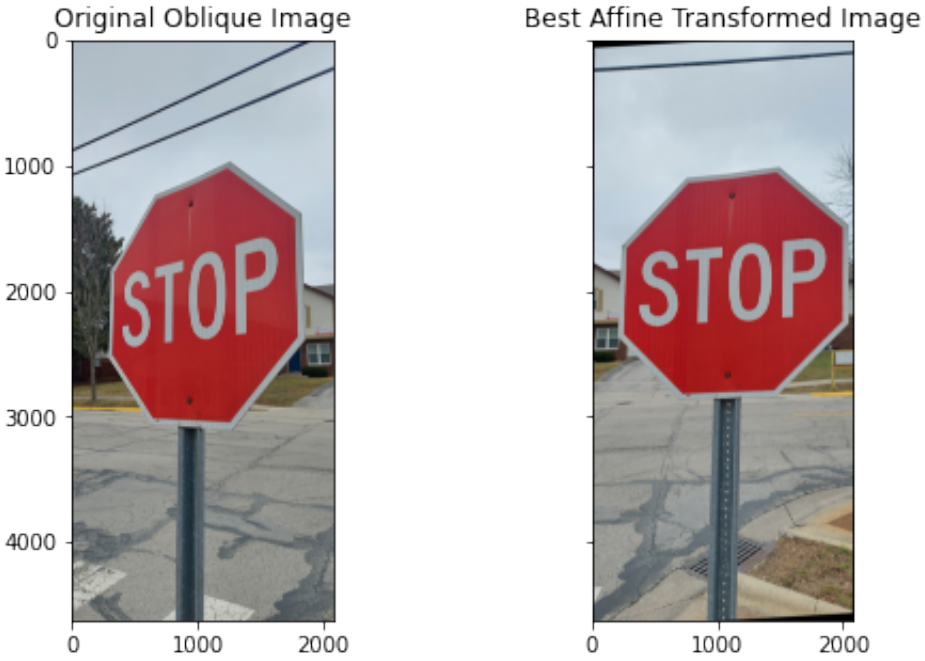

Figure 2: Original oblique image & best Affine transformed image. Wasserstein Distance = 0.0144

consistently by the Dataloader. Also, *tvt.GaussianBlur()* is applied to make the model more robust against blurry images.

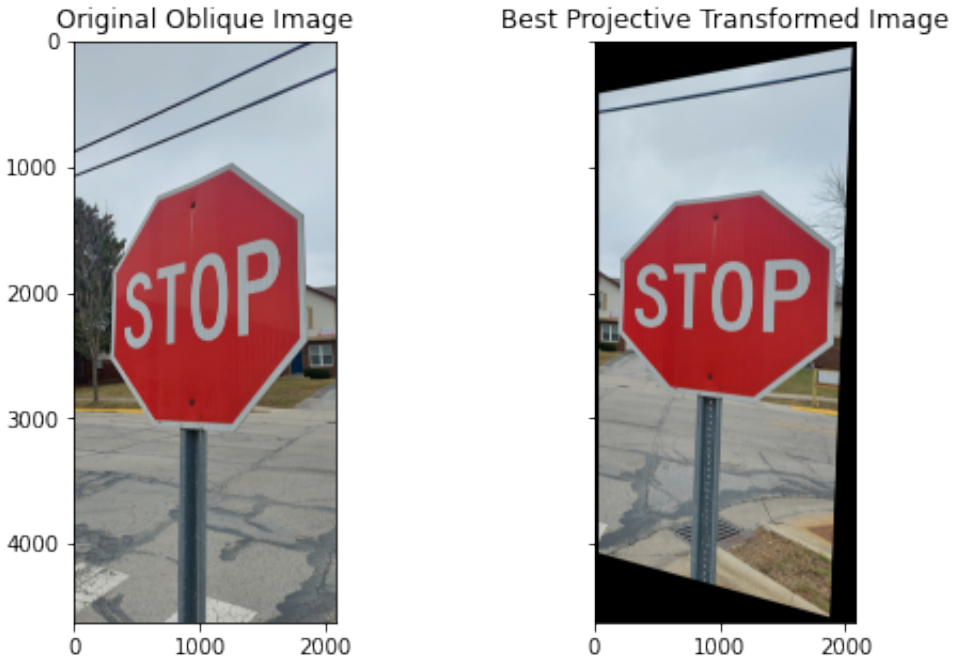

Figure 3: Original oblique image & best Projective transformed image. Wasserstein Distance = 0.0369

Running the code snippet in the task gives the following outputs:

```
>> my_dataset = MyDataset('/ECE60146/HW2/data/')
>> print(len(my_dataset))
>> 10
>> index = 10
>> print(my_dataset[index][0].shape, my_dataset[index][1])
>> torch.Size([3, 256, 256]) 0
>> index = 50
>> print(my_dataset[index][0].shape, my_dataset[index][1])
>> torch.Size([3, 256, 256]) 0
```

3.1.4 Task 3.4

- In this task, the custom dataset was wrapped in a *torch.utils.data.DataLoader* class provided by PyTorch for processing the images in batches. Initially, the batch size was set as 4 and a batch of images was obtained from the Dataloader and plotted as shown in Fig 5
- To compare the performance of Dataset and Dataloader, firstly the Dataset *__get_item__* method was called for processing 1000 images with randomly generator index values and its corresponding execution time was recorded. On average, the processing time for the dataset was found to be **261.38 s**. Next, the dataloaders were initialized with different combinations of batch size and a number of worker threads(all greater than 1), and the processing time of 1000 images was recorded. Table 1 shows the performance in different settings of the dataloader. It is seen that processing time decreases using a large batch size. Also, if the number of threads used for processing is more than the batch size the performance decreases.

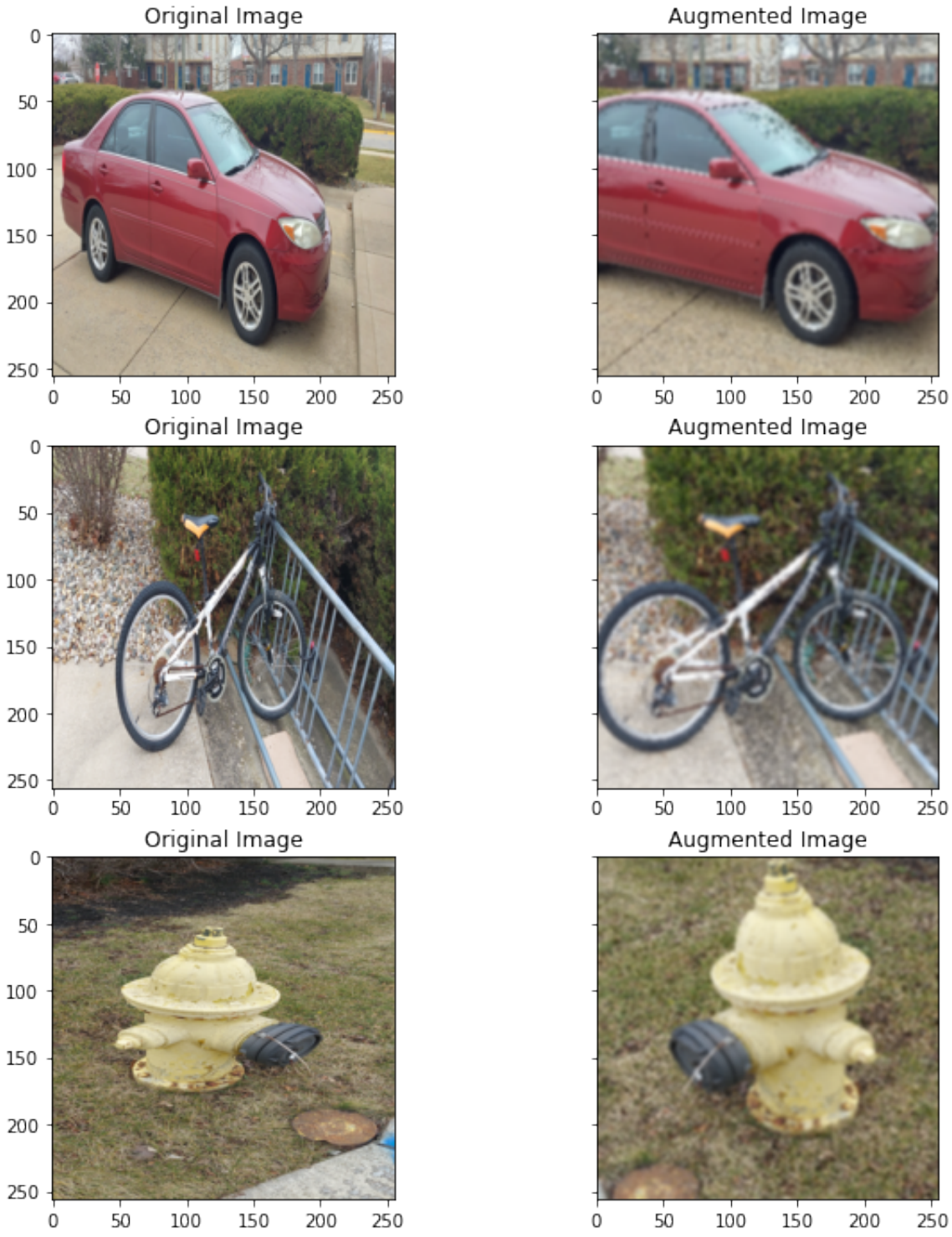

Figure 4: Three images with their augmented versions

Figure 5: Batch of 4 images returned by DataLoader

batch_size	num_workers	time(s)
2	2	221.42
2	3	228.16
2	4	243.02
4	2	183.46
4	3	188.53
4	4	188.66
8	2	154.69
8	3	156.39
8	4	156.77

Table 1: Performance comparison of Dataloader for different batch sizes and number of threads

4 Conclusion

In conclusion, *PIL* library can be used to load images from the disc and various transformations and augmentations can be applied to those images using the *torchvision.transforms* library for pre-processing the image data before training. This step converts the image data into normalized tensors which are appropriate for processing in GPU. Also, data augmentation helps in increasing the number of training samples in those cases where training data is present in limited amounts. It is found that the straight image was able to be mapped to the oblique image using Projective transformation. Also, custom datasets can be implemented as per the format of the data which can then be wrapped in a Dataloader class for faster processing of images in chunks of mini-batches using the appropriate batch size and the number of workers. This not only makes the training process faster but also stabilizes it.

5 Source Code

```
# -*- coding: utf-8 -*-
"""hw2_SouradipPal.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/12_MkHvjiZGw2zP_nPxMbJJJbBNipHOz-
"""
```

```
from google.colab import drive
drive.mount('/content/drive')

from PIL import Image, ExifTags
import torch
import torchvision.transforms as tvt
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import wasserstein_distance

# Method to load PIL Image from file
def load_image(filepath):
    img = Image.open(filepath)

    # Image orientation issue resolved as given in
    # https://stackoverflow.com/questions/13872331
    try:
        for orientation in ExifTags.TAGS.keys():
            if ExifTags.TAGS[orientation]=='Orientation':
                break

        exif = img._getexif()
        if exif == None:
            return img

        if exif[orientation] == 3:
            img = img.rotate(180, expand=True)
        elif exif[orientation] == 6:
            img = img.rotate(270, expand=True)
        elif exif[orientation] == 8:
            img = img.rotate(90, expand=True)
        return img

    except (AttributeError, KeyError, IndexError):
        return img

img1 = load_image('/content/drive/MyDrive/Purdue/ECE60146/HW2/data/stop_sign_001.jpg')
img2 = load_image('/content/drive/MyDrive/Purdue/ECE60146/HW2/data/stop_sign_002.jpg')

fig, axes = plt.subplots(1, 2, figsize=(6, 5), sharey=True)
axes[0].imshow(img1)
axes[1].imshow(img2)
plt.show()

transforms = tvt.Compose([tvt.ToTensor(), tvt.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5))])

img1_transformed = transforms(img1)
img2_transformed = transforms(img2)

# Logic taken from previous year solution
# https://engineering.purdue.edu/DeepLearn/2_best_solutions/2022/hw2_s22_sol1.pdf
```

```
# calculate Wasserstein distance between two images
def calculate_dist(im1, im2, num_bins = 10):
    channels_im1 = [im1[ch] for ch in range(3)]
    channels_im2 = [im2[ch] for ch in range(3)]
    histTensor1 = torch.zeros(3, num_bins, dtype=torch.float)
    histTensor2 = torch.zeros(3, num_bins, dtype=torch.float)
    hists1 = [torch.histc(channels_im1[ch], num_bins, -3.0, 3.0) for ch in range(3)]
    hists1 = [hists1[ch].div(hists1[ch].sum()) for ch in range(3)]
    hists2 = [torch.histc(channels_im2[ch], num_bins, -3.0, 3.0) for ch in range(3)]
    hists2 = [hists2[ch].div(hists2[ch].sum()) for ch in range(3)]

    d = []
    channels = ["R", "G", "B"]
    for ch in range(3):
        histTensor1[ch] = hists1[ch]
        histTensor2[ch] = hists2[ch]
        dist = wasserstein_distance(torch.squeeze(histTensor1[ch]).cpu().numpy(),
                                    torch.squeeze(histTensor2[ch]).cpu().numpy())
        d.append(dist)
    return d

calculate_dist(img1_transformed, img2_transformed)

# Experimenting Affine transformation using degree, translate, shear parameters
trans = np.linspace(0, 0.4, 5)
shear = np.linspace(-60, 60, 9)
min_affine_d = 1e9

for i, s in enumerate(shear):
    for j, t in enumerate(trans):
        affine_transformer = tvt.RandomAffine(degrees=(-5, 5), translate=(t, t),
                                              shear = [0, 0, s, s+5])
        img_t = affine_transformer(img1)
        img_t_transformed = transforms(img_t)
        d = calculate_dist(img_t_transformed, img2_transformed)
        # Comparing the L2 norm of Wassertein distance of the three channels
        if np.linalg.norm(d) < min_affine_d:
            min_affine_d = np.linalg.norm(d)
            img_affine = img_t
plt.show()

# Plot Best Image after Affine Transform
fig, axes = plt.subplots(1, 2, figsize=(8, 5), sharey=True)
axes[0].imshow(img2)
axes[0].set_title('Original Oblique Image')
axes[1].imshow(img_affine)
axes[1].set_title('Best Affine Transformed Image')

plt.show()

# Experimenting Projective transformation using distortion parameter
```

```
min_pers_d = 1e9

W, H  = img1.size

for i in range(20):
    startpoints, endpoints = tvt.RandomPerspective().get_params(W, H, 0.3)
    img_t = tvt.functional.perspective(img1, startpoints, endpoints)
    img_t_transformed = transforms(img_t)
    d = calculate_dist(img_t_transformed, img2_transformed)
    # Comparing the L2 norm of Wassertein distance of the three channels
    if np.linalg.norm(d) < min_pers_d:
        min_pers_d = np.linalg.norm(d)
        img_perspective = img_t
plt.show()

# Plot Best Image after Projective Transform
fig, axes = plt.subplots(1, 2, figsize=(8, 5), sharey=True)
axes[0].imshow(img2)
axes[0].set_title('Original Oblique Image')
axes[1].imshow(img_perspective)
axes[1].set_title('Best Projective Transformed Image')
plt.show()

print(min_affine_d, min_pers_d)

import os
import torch

# Creating custom dataset class
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.root_dir = root
        self.img_paths = os.listdir(self.root_dir)
        self.transforms = tvt.Compose([tvt.ToTensor(), tvt.RandomHorizontalFlip(),
                                       tvt.RandomResizedCrop(256, scale=(0.9, 1.0)),
                                       tvt.GaussianBlur(5, sigma=(1.0, 2.0))])

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

    def __getitem__(self, index):
        # Read an image at index and perform augmentations
        # Return the tuple : (augmented tensor, integer label)
        index = index % len(self.img_paths)
        img_path = self.img_paths[index]
        path = os.path.join(self.root_dir, img_path)
        im = load_image(path)
```

```
        im_transformed = self.transforms(im)
        return im_transformed, index

# Example Output
my_dataset = MyDataset(root = '/content/drive/MyDrive/Purdue/ECE60146/HW2/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])

# PLot three original images with augement versions
fig, axes = plt.subplots(3, 2, figsize=(10, 12), sharey=True)
indices = np.random.randint(0, len(my_dataset), 3)
for i in range(3):
    index = indices[i]
    img, label = my_dataset[index]
    img_orig = load_image(os.path.join(my_dataset.root_dir, my_dataset.img_paths[index]))
    img_orig = img_orig.resize((256, 256))
    axes[i][0].imshow(img_orig)
    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')

plt.show()

# Defining dataloader
my_dataloader = torch.utils.data.DataLoader(dataset=my_dataset, batch_size=4,
                                            shuffle=True, num_workers = 2)

# Get a batch of augmented images and plot it
iterator = iter(my_dataloader)
batch = next(iterator)
fig, axes = plt.subplots(1, 4, figsize=(12, 8), sharey=True)
for i in range(4):
    img = batch[0][i]
    axes[i].imshow(np.array(img).transpose(1,2,0))
plt.show()

import time

# Compare dataset and dataloader performance
def compare_perf(dataset, num_iters = 1000):
    n = len(dataset)
    indices = np.random.randint(n, size = num_iters)
    start = time.time()
    for i in indices:
        img, label = dataset[i]
    end = time.time()
    print(f'Time to process 1000 images in Dataset using __get_item__: {end - start} s')
```

```
    for bsize in [2,4,8]:
        for nworkers in [2, 3, 4]:
            print(f'Performance of dataloader with batch size {bsize}, num_workers {nworkers}:')
            dataloader = torch.utils.data.DataLoader(dataset=dataset,
                                                     batch_size=bsize,
                                                     shuffle=True,
                                                     num_workers = nworkers)
            iterator = iter(dataloader)
            start = time.time()
            # multiple iteration of dataloader (https://stackoverflow.com/questions/47714643/)
            for i in range(int(num_iters/bsize)):
                try:
                    img, label = next(iterator)
                except StopIteration:
                    iterator = iter(dataloader)
                    img, label = next(iterator)
            end = time.time()
            print(f'Time to process {num_iters} images in DataLoader: {end - start} s')

compare_perf(my_dataset, num_iters = 1000)
```