Deep Learning HW2

Shen Chang Jan 25 2023

Section 2

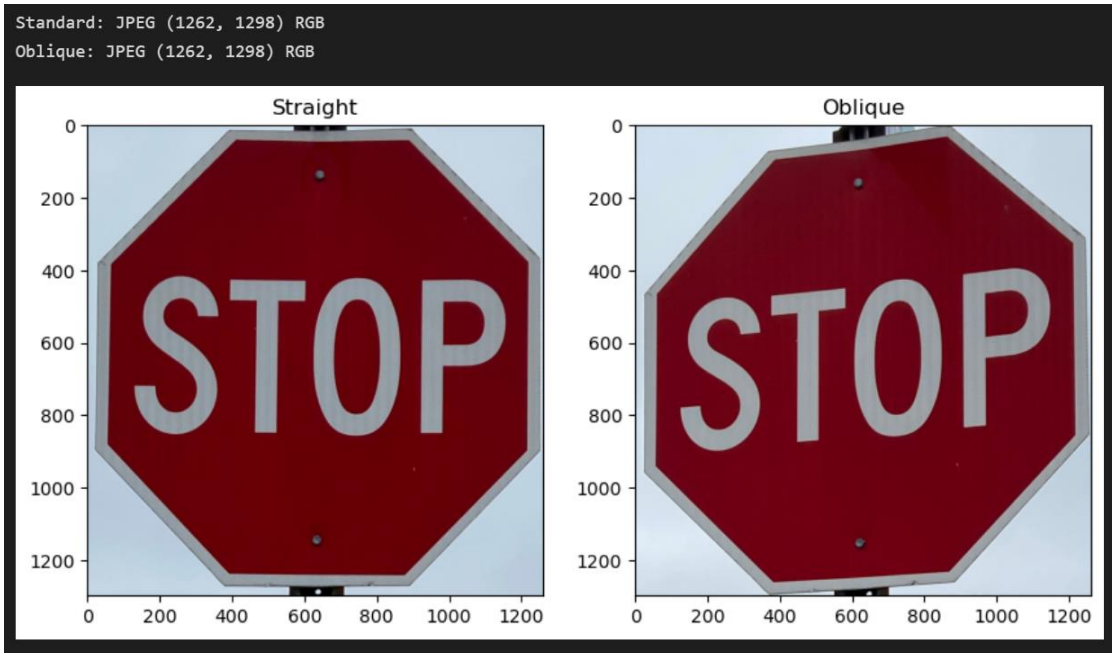
Question: If the pixel-value scaling by the piece of code in Slide 28 is on a per-image basis and if the same by the code shown on Slide 26 is on a batch basis, how come the two results are exactly the same?

Answer: Slide 26's scaling for the image is based on the maximum value of all images, found using "images.max()." The tvt.ToTensor() function in Slide 28 applies this same maximum value, i.e, the global maximum in the batch, to all four images (images_format = (4, 3, 5, 9)) during scaling, resulting in identical outputs when printing images_scaled[0] using both methods.

Section 3

3.2

Figure 1. Photos input of a Stop Sign



To find out the optimal set of parameters for **tvt.RandomAffine** and the **tvt.functional.perspective()**, I generated 100 list sets with different parameters for attributes “degree” 、”scale” in RandomAffine and “distortion_scale” in RandomPerspective. Note that instead of using **tvt.functional.perspective()** for perspective transform, here I employed the **tvt.RandomPerspective()** for transformation to shrink the amount of parameters. My goal is to transform the Straight image into the Oblique one.

Processing with 100 samples

```python
from pyDOE import lhs
from tqdm import tqdm
import numpy.matlib as npm

nsamples = 100
reruns = 1
nparams = 3

# Set up parameter array
params = npm.repmat(lhs(nparams, samples = nsamples),reruns,1)
# Each row is a new parameter set
```

Python

```python
#Shrink the parameter ranges
DT = list(params[:,0]*0.2+0.1) # distortion from scale (0.1 to 0.3)
SC= list(params[:,1]*0.2+0.9) # Image scaling from scale (0.9 to 1.1)
D = list(params[:,2]*20) # Image rotation from degree (0 to 20)
# translations in the x and y directions is not required in this example
```

Python

```python
Distance = []
for i in tqdm(range(0,nsamples*reruns)):
    distortion = DT[i];  scaling = SC[i]; degree = D[i];

    ## Image Transformations (Affine & Project Distortions)
    transform1 = T.Compose([ # tranform standard.jpg PIL image to tensor and do the image transformation
    T.ToTensor(),
    T.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    T.RandomAffine(degrees=(degree,degree), translate=(0, 0), scale=(scaling, scaling)), #  Affine Transform
    T.RandomPerspective(distortion_scale=distortion, p=1.0), #  Project Distortions Transform
    ])

    transform2 = T.Compose([  # tranform oblique.jpg PIL image to tensor
    T.ToTensor(),
    T.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])
    img = transform1(im_st)
    im = transform2(im_ob)

    Distance.append(Wasserstein_Distance(im,img)) # Wasserstein Distance
```

Python

```
100%|██████████| 100/100 [00:42<00:00,  2.38it/s]
```

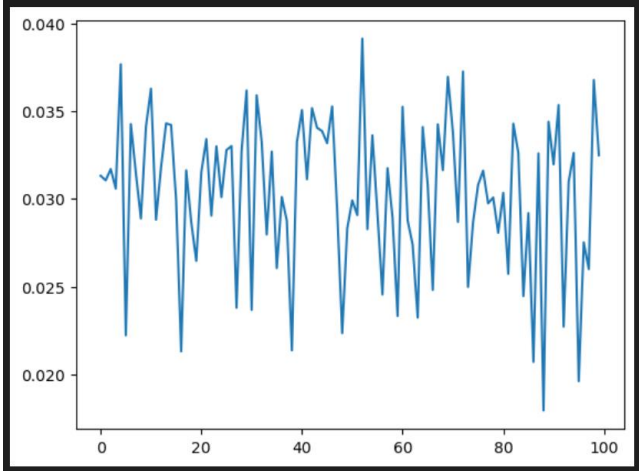
Function for Calculating Wasserstein Distance

```python
from scipy.stats import wasserstein_distance

def Wasserstein_Distance(im,img): # im & img should be tensor input
    #  Construct Per-Channel Histograms
    hist1 = torch.histc(img, bins=10, min=-1.0, max=1.0).float()
    hist2 = torch.histc(im, bins=10, min=-1.0, max=1.0).float()
    # Normalized histograms
    hist1 = hist1.div(hist1.sum())
    hist2 = hist2.div(hist2.sum())
     # Caculate Wasserstein Distance
    dist = wasserstein_distance( torch.squeeze( hist1 ).cpu().numpy(),
    torch.squeeze(hist2).cpu().numpy() )
    return dist
```

Python

Result

```
min = 88
Optimum fitting occurs with with distortion: 0.10392588866948436, scaling: 1.011570658663367, rotation: 2.298447008721491 degree
```



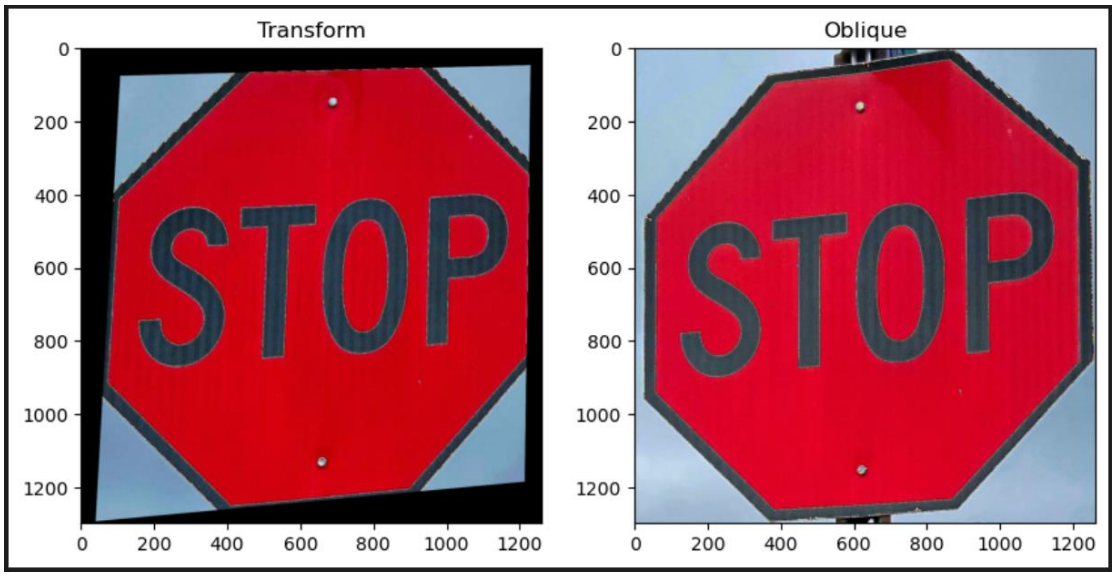
Wasserstein Distance Between Channel Histograms

```python
#  Construct Per-Channel Histograms
color_channels_img = [img[ch] for ch in range(3)]
color_channels_im = [im[ch] for ch in range(3)]
hist1 = [torch.histc(color_channels_img[ch], bins=10, min=-1.0, max=1.0).float() for ch in range(3)]
hist2 = [torch.histc(color_channels_im[ch], bins=10, min=-1.0, max=1.0).float() for ch in range(3)]
# Normalized histograms
hist1 = [hist1[ch].div(hist1[ch].sum()) for ch in range(3)]
hist2 = [hist2[ch].div(hist2[ch].sum()) for ch in range(3)]
for ch in range(3):
    dist = wasserstein_distance( torch.squeeze( hist1[ch]).cpu().numpy(),
    torch.squeeze( hist2[ch] ).cpu().numpy() )
    print("\n Wasserstein distance for channel %d: " % ch, dist)
```

Python

```
Wasserstein distance for channel 0:  0.01182553350035505

Wasserstein distance for channel 1:  0.013388020798447541

Wasserstein distance for channel 2:  0.018439192359801383
```

Figure 2. The Result of the Optimal Transformation



Note that I have applied “tvt.ToTensor” to both images for generating their histograms by “torch.histc.” However, there is a tradeoff that we can’t successfully get the same image after transferring the tenor back to PIL format due to the maximum problem discussed in Section 2.

3.3

Custom Dataset Class

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

    def __init__(self, root):
        super().__init__()
        self.root = root
        self.images = os.listdir(root) # Obtain meta information (e.g. list of file names)
        self.transform =  T.Compose([ # tranform standard.jpg PIL image to tensor and do the image transformation
                                T.ToTensor(),
                                T.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                                T.RandomAffine(degrees=30, translate=(0.0, 0.0), scale=(0.8, 1.2)), #  Affine Transform
                                T.RandomPerspective(distortion_scale=0.2, p=1.0), #  Perspective Transform
                                T.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5) #  Color Transform
                                ])

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

    def __getitem__(self, index):
        img_path = os.path.join(self.root, self.images[index-1]) #  Read an image at index  !!!!!!
        img = Image.open(img_path)
        img_t = self.transform(img)

        return img_t, np.random.randint (0 ,10) # Return the tuple : ( augmented tensor , integer label )
```

Python

Result

```python
path = os.path.join(os.getcwd(),'Dataset')
my_dataset = MyDataset (path)
print (len(my_dataset))
index = 10
print ( my_dataset[index][0].shape , my_dataset[index][1])
```

Python

```
10
torch.Size([3, 256, 256]) 6
```

To better visualize the transformation, I built up another class without the “tvt.ToTensor” in the tvt.Compose and applied the Affine, Perspective, and the Color transformation with RandomAffine, RandomPerspective, and ColorJitter.

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

    def __init__(self, root, transform = None):
        super().__init__()
        self.root = root
        self.images = os.listdir(root)
        self.transform =  transform

    def __len__(self):
        return len(self.images)

    def __getitem__(self, index):
        img_path = os.path.join(self.root, self.images[index-1]) #  Read an image at index
        img = Image.open(img_path)

        if self.transform: # Do specific transformation
            img_t = self.transform(img)

        return img, img_t
```

Python

```python
Affine = T.Compose([T.RandomAffine(degrees=220, translate=(0.0, 0.0), scale=(0.8, 1.2)) ]) #  Affine Transform
Perspective = T.Compose([T.RandomPerspective(distortion_scale=0.2, p=1.0)]) #  Perspective Transform
Color = T.Compose([T.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5)]) #  Color Transform
```

Python

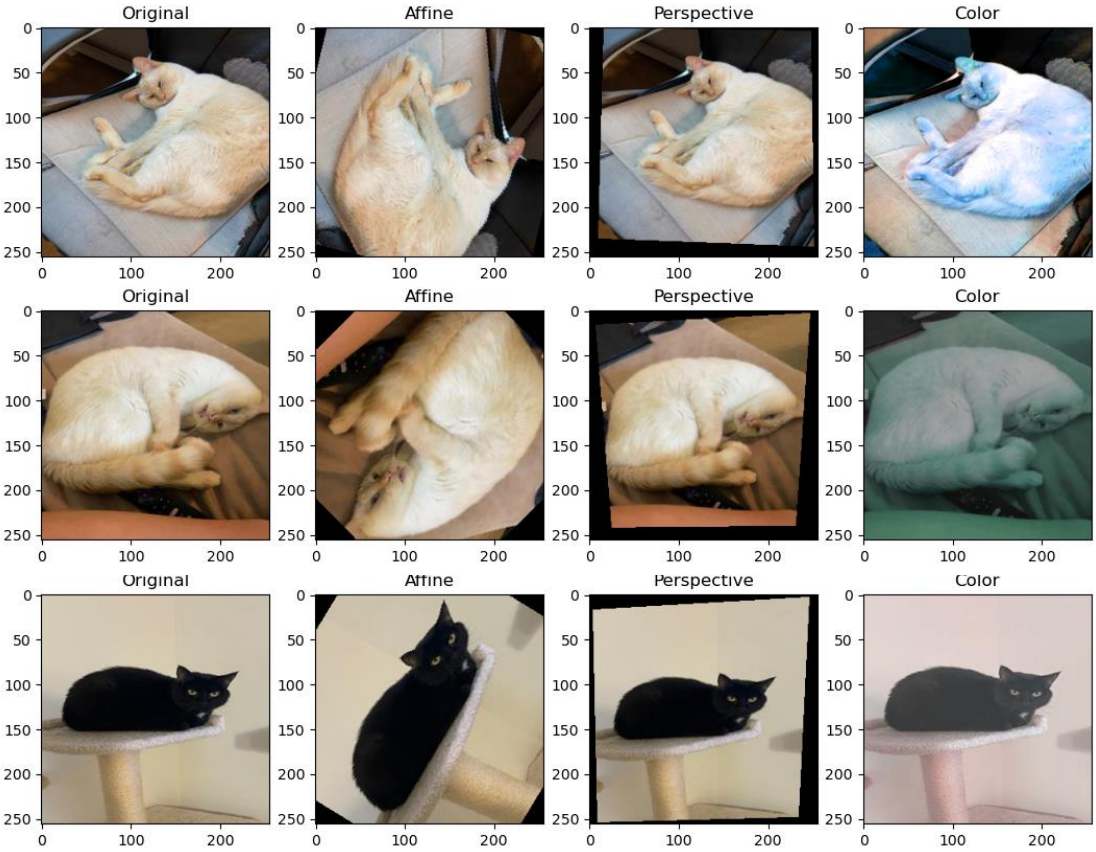
Reason for selecting these transformations for augmentation:

- **Affine transformation:** It can simulate various real-world scenarios such as rotation, translation, scaling, and skewing, which can make the model more robust and less prone to overfitting. It also increases the overall diversity of the data, allowing the model to generalize better to unseen data.

- **Perspective transformation:** It can simulate changes in the viewing angle of an image, such as when an object is viewed from different positions or distances. Perspective transformation can also help the model to learn about the 3D structure of an object, which can be useful for tasks such as object detection or image segmentation.

- **Color transformation:** It can simulate changes in the lighting conditions or the color balance by randomly altering the brightness, contrast, hue, saturation and gamma correction of the image. Additionally, color transformation can also help to the changes in color representation, like in different devices, cameras, lighting conditions, etc.

Figure 3. Three Photos with Three Different Types of Transformation



3.4

Record the Timing with Calling __getitem__ 1000 Times

```python
import random
import time

random_list = [random.randint(0,9) for _ in range(1000)] # build up a list with 1000 integers between (0,9)
start_time = time.time() # start the timer
my_dataset = MyDataset(path)
dataset = [my_dataset.__getitem__(i) for i in random_list] # create a dataset with __getitem__
print("--- %s seconds ---" % (time.time() - start_time))  # end the timer
```

Python

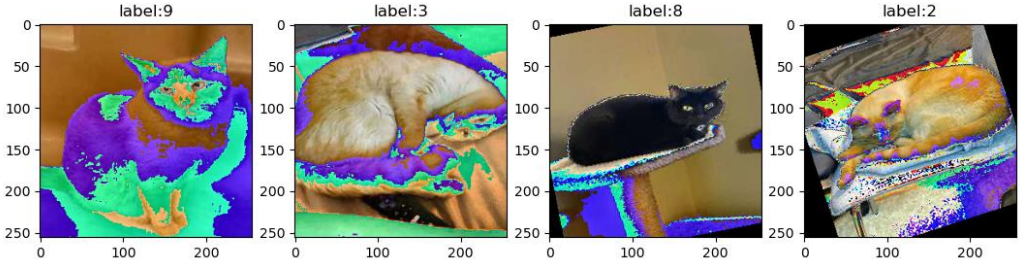
```
--- 10.581845998764038 seconds ---
```

DataLoader with batch_size = 4

```python
dataloader = torch.utils.data.DataLoader(dataset, batch_size=4, shuffle=False, num_workers=4)  # Create the DataLoader, passing the dataset
for data in dataloader:
    x,y = data[0], data[1] # x,y will be the input tuple of the model f(x,y)

# Plot images in a batch, here I will take out the last batch
fig = plt.figure(figsize=(13, 10))
for i in range(len(x)):
    img_ = T.ToPILImage()(x[i])
    fig.add_subplot(1, len(x), i+1)
    plt.imshow(img_)
    plt.title('label:'+ str(y[i].numpy()))
```

Python

Figure 4. Four Images in a Batch Generated by DataLoader



Record the Timing Using DataLoader

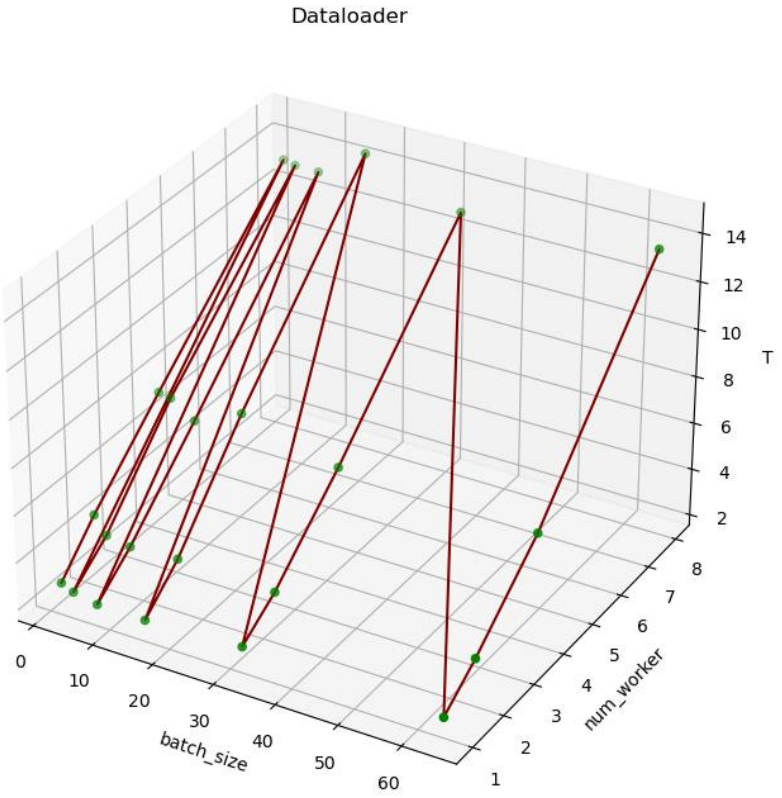
```python
# Create the DataLoader, passing the dataset and record the timing
batch_sizes = [2, 4, 8, 16, 32, 64]
num_workers= [1, 2, 4, 8]

times = {'batch_size':[], 'num_worker':[], 'time':[]}
for bs in batch_sizes:
    for n in num_workers:
        dataloader = torch.utils.data.DataLoader(dataset, batch_size=bs, shuffle=False, num_workers=n)
        start_time = time.time()
        for data in dataloader:
            x,y = data[0], data[1]
            # begin your model(x,y) computations
        times['time'].append(time.time() - start_time)
        times['batch_size'].append(bs)
        times['num_worker'].append(n)
```

Python

Tabulation

	batch_size	num_worker	time
0	2	1	2.904463
1	2	2	4.604993
2	2	4	7.524226
3	2	8	13.206249
4	4	1	2.672436
5	4	2	3.875655
6	4	4	7.423990
7	4	8	13.103817
8	8	1	2.465467
9	8	2	3.705628
10	8	4	6.744732
11	8	8	13.079484
12	16	1	2.450431
13	16	2	3.820676
14	16	4	7.651614
15	16	8	14.373039
16	32	1	2.653872
17	32	2	3.705731
18	32	4	6.577157
19	32	8	12.954106
20	64	1	2.407194
21	64	2	3.561763
22	64	4	6.305084
23	64	8	13.570806

Figure 5. The Timing Variation According to Different Parameters



Discussion

- “**batch_size**”: When I increase the **batch size** in the Dataloader, the time it takes to complete one iteration may decrease. This is because larger batch sizes allow the model to make better use of the hardware, such as a GPU, by making more efficient use of memory and allowing for more parallelization. Thus, when using the Pytorch Dataloader, increasing the batch size can lead to fewer number of iterations needed to go through the entire dataset, which can also speed up the overall training time. However, it's not always the case, increasing the batch size too much can lead to memory issues.
- “**num_workers**”: When I increase the number of '**num_workers**' in a parallel processing setting, the time it takes to complete a task may increase because of the overhead associated with managing and coordinating multiple worker processes. Additionally, if the task is not easily parallelizable, increasing the number of workers may not lead to a proportional decrease in completion time.