ECE60146 DL HW2

Akshita Kamsali, akamsali@purdue.edu

January 24, 2023

Note: bold lower case letters indicate vectors and bold upper case letters indicate matrices

1 Theory

The mystery of how converting to float and dividing by max value yields the same result as `ToTensor` from `torchvision.transforms` can be solved from looking at the source code for the implementation of latter.

In fig 1 we can see that the image is reshape to $\mathbf{C} \times \mathbf{H} \times \mathbf{W}$ and then divided by 255 to normalise. This is based on the assumption that PIL images are 8 byte integers of shape $\mathbf{H} \times \mathbf{W}$ with C channel number of tuples at every position. Since the maximum value an 8 byte integer can take is 255, it is hardcoded as such.

2 Programming

2.1 Task 1: Applying Affine/Perspective transform

In the field computer vision, homography is used a fundamental tool to perform many image based tasks. When training a neural network, we want the model to learn to identify these projective distortions and illumination differences. To show one such transformation we pick points from a front view image of a stop sign and oblique view image. Then we try to transform the oblique view into a front view with either `RandomAffine` or `perspective` functions available to us in `torchvision.transforms`.

Figure 2 shows RandomAffine and perspective transform to tranform oblique view to front view. Figures 3 and 4 show zoomed version from 2b and 2c respectively.

```
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 1: Source code from PyTorch GitHub Repo

Method	Runtime
MyDataset	635.8
Dataloader ($batchsize = 4, nw = 1$)	672
Dataloader ($batchsize = 4, nw = 2$)	415
Dataloader ($batchsize = 4, nw = 4$)	311
Dataloader ($batchsize = 8, nw = 2$)	350
Dataloader ($batchsize = 8, nw = 4$)	300

Table 1: Runtimes for my implementation and Pytorch Dataloader with varying workers (nw)

Similarity measurement Then we generate histograms from two images to measure similarity using wasserstain distance. Wasserstein distance between the two images is 0.001020476357306882. Histograms are shown in 5

2.2 Task 2: Using torch Dataloader

Resize and Normalise I chose to resize images to (1024, 1024), followed by conversion to tensors using `ToTensor()`, then `Normalise`.

Gaussian blur to enhance features at scales After that I made the design choice of using `Gaussianblur`. I made this choice to enhance image structures at different scales, and simultaneously reduce noise.

Affine and Perspective transforms Then I chose to apply `Affine` and `Perspective` transforms to introduce affine and projective tranformations at random to the images in dataset. This to done to augment more types of scenarios where the images may be taken from oblique angles and distorted due to varying perceptions.

2.2.1 Comparision of `__getitem__` and Parallel Dataloader

Time taken for 1000 calls of `__getitem__` 635.8037867546082.
now with batch size fixed at 4:
Time taken for 1000 calls of torch Dataloader with one worker: 671.9216966629028
Time taken for 1000 calls of torch Dataloader with two workers: 415.3375608921051
Time taken for 1000 calls of torch Dataloader with nw = 4: 311.74111437797546
Table 1 shows runtimes.

I observe that absolute computation times vary depending on hardware. However, the pattern looks very similar with computation time reducing with increasing batch size and number of workers. Figures 6 and 7 show images from `__getitem__` and torch Dataloader calls.

(a) Front view and Oblique view

(b) Front view and Oblique view with Affine transformed applied

(c) Front view and Oblique view with Affine and perspective transformed applied

Figure 2: Task 1 figures

Figure 3: Oblique view with Affine transformation

Figure 4: Oblique view with Affine followed by perspective transformation

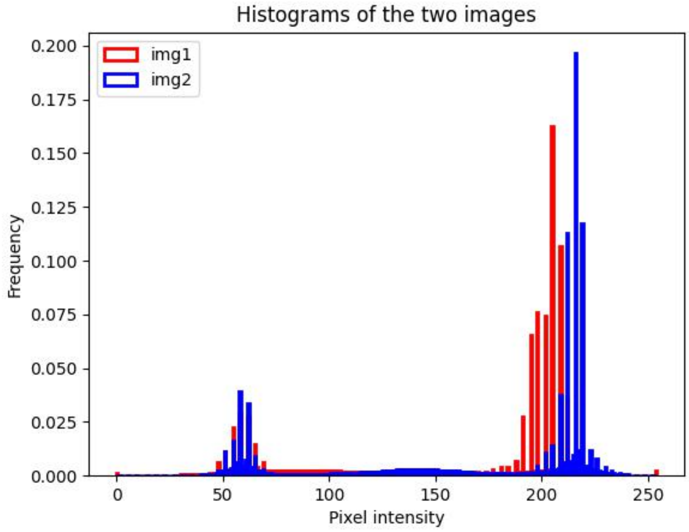
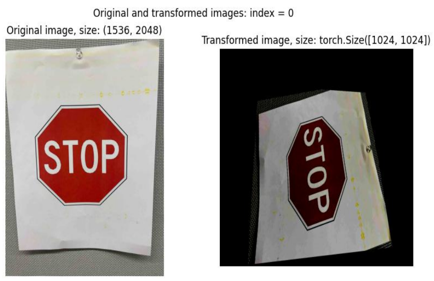


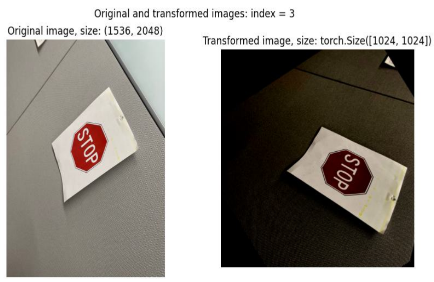
Figure 5: Histogram showing histograms for two different perspectives after transformation



(a) Index = 0

(b) Index = 1

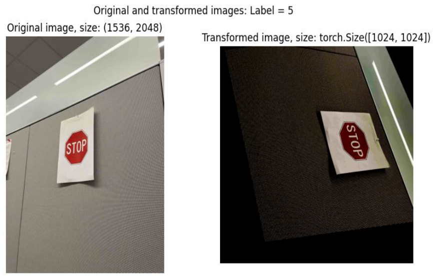
(c) Index = 2



(d) Index = 3

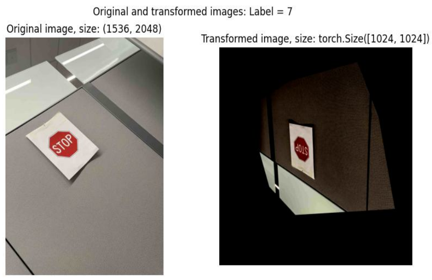
Figure 6: Plots with 4 calls to `__getitem__`, image before and after transformation

(a) Label = 3



(b) Label = 5

(c) Label = 10



(d) Label = 7

Figure 7: Plots with iterating over over first batch of the Dataloader with batchsize 4

3 Code

I have written my code for `MyDataset` in a seperate .py file and call it from my main file.

3.1 MyDataset

```python
import glob
from PIL import Image
from torchvision import transforms as tvt
import torch


class MyDataset(torch.utils.data.Dataset):
    def __init__(self, root: "str") -> None:
        super().__init__()
        # get file list with glob.glob
        self.file_list = sorted(glob.glob(root + "/*.jpg"))
        # set transform with tvt.Compose
        self.transform = tvt.Compose(
            [
                tvt.Resize((1024, 1024)),
                tvt.ToTensor(),
                tvt.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5]),
                tvt.GaussianBlur(3, sigma=(0.1, 2.0)),
                tvt.RandomAffine(
                    degrees=(0, 90),
                    translate=(0.1, 0.1),
                    scale=(0.8, 1.2),
                    shear=10,
                    fill=0,
                ),
                tvt.RandomPerspective(distortion_scale=0.5, p=0.5, fill=0),
            ]
        )

    def __len__(self) -> int:
        return len(self.file_list)

    def __getitem__(self, index: int):
        # read image with PIL.Image.open
        img = Image.open(self.file_list[index])
        # transform image with self.transform
        img = self.transform(img)
        label = self.file_list[index].split("/")[-1].split(".")[0]

        return img, int(label)
```

mydataset.py

3.2 main

```python
from mydataset import MyDataset

import torch
from torch.utils.data import DataLoader
import torchvision.transforms as tvt

```

```
from scipy.stats import wasserstein_distance
from PIL import Image

import numpy as np
import matplotlib.pyplot as plt
from time import time

import matplotlib.pyplot as plt

img1 = Image.open("data/1.jpg")
img2 = Image.open("data/2.jpg")


def concatenate_img(img1, img2, name="concat.jpg"):
    concat_img = Image.new("RGB", (img1.size[0] * 2, img1.size[1]), "white")

    # pasting the first image (image_name, (position))
    concat_img.paste(img1, (0, 0))
    # pasting the second image (image_name, (position))
    concat_img.paste(img2, (img1.size[0], 0))

    # save the concatenated image
    plt.imshow(concat_img)
    plt.axis("off")
    plt.savefig(f"solutions/{name}")


concatenate_img(img1, img2)


# apply affine transform to img2
affine_tr = tvt.RandomAffine((30, 40))
img2 = affine_tr(img2)

# save the transformed image
img2.save("solutions/2_aff.jpg")
concatenate_img(img1, img2, name="concat_aff.jpg")


# apply perspective transform to img2
start_pts = [[851, 728], [973, 683], [851, 1171], [961, 1110]]
end_pts = [[546, 572], [917, 578], [579, 1431], [925, 1431]]

transformed_image = tvt.functional.perspective(
    img2, startpoints=start_pts, endpoints=end_pts
)
transformed_image.save("solutions/2_aff_persp.jpg")
concatenate_img(img1, transformed_image, name="concat_aff_pers.jpg")


# generate histograms
def get_hist(img, bins=10):
    tensor_img = tvt.functional.pil_to_tensor(img.convert('L')).float()
    hist_img = torch.histc(tensor_img, bins=bins, min=0, max=255)
    hist_img = hist_img.div(hist_img.sum()) # normalize the histogram for probability
    return hist_img

# get histograms of the two images
bins = 255
hist_img1 = get_hist(img1, bins=bins)
```

```
hist_img2 = get_hist(transformed_image, bins=bins)

# plot the histograms

plt.bar(np.arange(bins), hist_img1, fill=False, edgecolor='red', linewidth=2, label='img1')
plt.bar(np.arange(bins), hist_img2, fill=False, edgecolor='blue', linewidth=2, label='img2')
plt.legend()
plt.title('Histograms of the two images')
plt.ylabel('Frequency')
plt.xlabel('Pixel intensity')
plt.savefig('solutions/hist.jpg')


# compute wasserstein distance
dist = wasserstein_distance(hist_img1.cpu().numpy(),
                            hist_img2.cpu().numpy())
print(f'Wasserstein distance between the two images is {dist}')


my_dataset = MyDataset('data')
print(len(my_dataset))
index = 9
print(my_dataset[index][0].shape, my_dataset[index][1])

def plot_og_tr_img(og_img, tr_img, index=9):
    fig, ax = plt.subplots(1, 2, figsize=(10, 5))
    ax[0].imshow(og_img)
    ax[0].axis('off')
    ax[0].set_title(f'Original image, size: {og_img.size}')
    ax[1].imshow(tr_img)
    ax[1].set_title(f'Transformed image, size: {tr_img.size()[:2]}')
    ax[1].axis('off')
    plt.suptitle(f'Original and transformed images: Label = {index}')
    plt.savefig(f'solutions/dl_og_tr_img_{index}.jpg')

# plot images from getitem
for i in range(10):
    og_img = Image.open(my_dataset.file_list[i])
    tr_img = my_dataset[i][0].permute(1,2,0)
    plot_og_tr_img(og_img, tr_img, index=i)

# plot images from dataloader
batch_size = 4
train_dataloader = DataLoader(my_dataset, batch_size=batch_size, shuffle=True, num_workers=2)

train_data, labels = next(iter(train_dataloader))

#plot images from a batch in train dataloader
for i in range(batch_size):
    og_img = Image.open(my_dataset.file_list[labels[i].item() - 1])
    tr_img = train_data[i].permute(1,2,0)
    plot_og_tr_img(og_img, tr_img, index=labels[i])



# Time taken for 1000 calls of __getitem__
start = time()
```

```python
num_tot_images = len(my_dataset)
for i in range(1000):
    my_dataset.__getitem__(i%num_tot_images)

print("Time taken for 1000 calls of __getitem__", time() - start)

# Time taken for 1000 image accumulations in torch Dataloader
batch_size = 4
train_dataloader = DataLoader(
    my_dataset, batch_size=batch_size, shuffle=True, num_workers=2
)  # change num_workers to 0/1/4 to see the difference

start = time()

count = 0
while True:
    for img, labels in train_dataloader:
        count += labels.shape[0]
    if count >= 1000:
        break

print("Time taken for 1000 calls of torch Dataloader", time() - start)
```

main.py