BME646 and ECE60146: Homework 2

Spring 2023

Arghadip Das

das169@purdue.edu

1. Introduction

The aim of this homework is to make us familiarize with the image representations such as PIL and torch tensor. It also introduces the necessary concepts to implement an image dataloader within PyTorch framework.

2. Understanding Data Normalization

The results in Slides 26 and 28 are same although the methods are different. We obtain the results in Slide 26 through manual computation (dividing the pixel values in ALL of the batch images by the max value of the entire batch, i.e., 255) for every image. However, *tvt.ToTensor* is used to get the results in Slide 28. It appears that *tvt.ToTensor* divides the pixel values of an image by the max pixel value in that image. As it is operating on per image basis (due to the for loop), but the maximum value (255) appears ONLY in second channel of the third image, the answers in two previously mentioned slides should be different.
I believe, the "mystery" is *tvt.ToTensor* always divides the pixel values with the maximum possible pixel value in *int8* format, i.e. 255. It does not matter if the maximum value (255) is present in the image channels or not. That is the reason why two results are same.

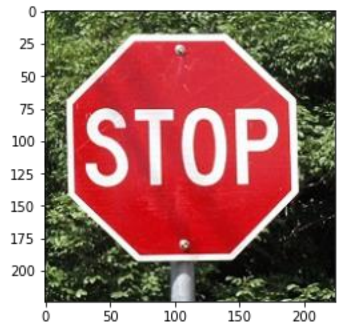
3. Programming Tasks

3.1. *Setting Up Your Conda Environment*

As I am using Google Colab for this assignment, this step is skipped.

3.2. *Becoming Familiar with torchvision.transforms*

Two captured images of the stop sign are given below. The image dimensions are (224, 224).

(a) 

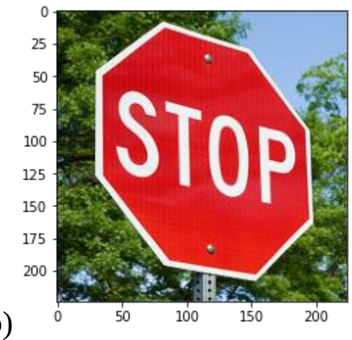
(b) 

Fig. 1. (a) Direct (target), and (b) oblique images of a stop sign

Source code for loading and displaying the images in Fig. 1:

```
# Importing necessary libraries for all programming tasks
import torch                                    # PyTorch
import torchvision.transforms as tvt            # Torchvision transforms
import numpy                                    # Numpy for miscellaneous tasks
from PIL import Image                           # Pillow for images
import random                                   # random for random numbers
import os                                       # os for proper directory paths
import matplotlib.pyplot as plt                 # for displaying the images
from scipy.stats import wasserstein_distance   # Calculate the distance between histograms

# Proper seed setting for reproducibility of the results
# Taken from Slide 73
seed = 0
random.seed(seed)
torch.manual_seed(seed)
torch.cuda.manual_seed(seed)
numpy.random.seed(seed)
torch.backends.cudnn.deterministic=True
torch.backends.cudnn.benchmarks=False
os.environ['PYTHONHASHSEED'] = str(seed)

path_direct = "/content/drive/MyDrive/Arghadip/DL/stop_direct.jpg"     # Path for direct image
im_direct = Image.open(path_direct)                                     # Load as PIL object
plt.imshow(im_direct)                                                   # To show image
plt.show()                                                              # Display on screen

path_oblique = "/content/drive/MyDrive/Arghadip/DL/stop_oblique.jpg"     # Path for Oblique image
```

```
im_oblique = Image.open(path_oblique)                                          # Load as PIL object
plt.imshow(im_oblique)                                                         # To show image
plt.show()                                                                     # Display on screen

# Histogram computation of direct stop sign
hist_direct = torch.histc(tvt.ToTensor()(im_direct), bins=10, min=0.0, max=1.0)
hist_direct = hist_direct.div(hist_direct.sum())
# print(hist_direct)
```

Best transformed images:

1. Using affine parameters:

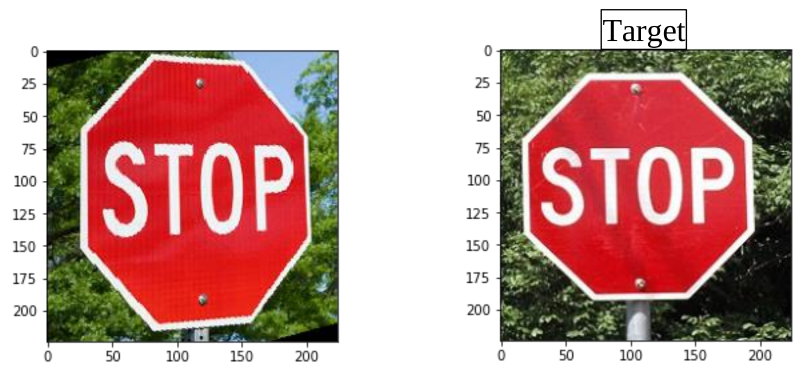


Fig. 2. Best transformed image using *tvt.RandomAffine()*. The parameters are *degree = (-15,-15), translate = (0.01,0), scale = (1.1,1.1), shear = [15,15,0,0]*. The Wasserstein distance between the direct image (Fig. 1(a)) and the transformed image (Fig. 2) is 0.012.

The exploration space of parameters is given below.

- degree: {(-25,-25), (-20,-20), (-15,-15)}
- translate: {(0.01,0), (0.02,0), (0.03,0)}
- scale: {(1,1), (1.1,1.1), (1.2,1.2)}
- shear: {[20,20,5,5], [15,15,0,0], [25,25,0,0]}

<u>Approach:</u>

Fig. 3. Affine transform (straight and parallel lines hold their behavior after the transform)

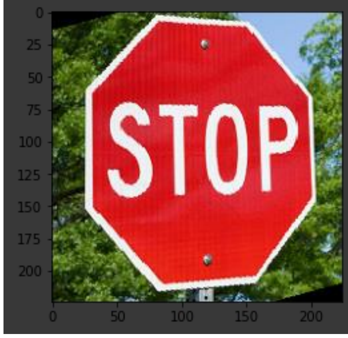
The oblique image in Fig. 1(b) needs to be rotated anti-clockwise in order to get the target image in Fig. 1(a). That's why the *degree* parameter is set to negative values. If carefully observed, the oblique image is also slightly shifted along horizontal direction. Here the *translate* parameter comes to our rescue. In a similar fashion, *scale* and *shear* parameters are also properly chosen after few trials and errors.

Source code to select the best parameters by minimizing the distance with the target image:

```
list_of_dist = []   # Empty list initialization to store the Wasserstein distance for diff params
list_of_params = [] # Empty list initialization to store different combination of parameters

for degree in range(-25,-14,5):                              # Start loop for parameter "degree"
  for translate_x in [0.01,0.02,0.03]:                       # Start loop for parameter "translate"
    for scale in [1,1.1,1.2]:                                # Start loop for parameter "scale"
      for shear in [[20,20,5,5],[15,15,0,0],[25,25,0,0]]: # Start loop for parameter "shear"

        # Creating the transformer with the proper parameters
        affine_transfomer = tvt.RandomAffine(degrees=(degree,degree), translate=(translate_x,0), 
scale=(scale,scale), shear=shear)
        affine_img = affine_transfomer(im_oblique)  # Transformed image

        # Calculate the Wasserstein distance
        hist_affine = torch.histc(tvt.ToTensor()(affine_img), bins=10, min=0.0, max=1.0)    # 
Histogram with 10 bins
        hist_affine = hist_affine.div(hist_affine.sum())                                    # 
Normalize the histogram
        dist = wasserstein_distance(hist_direct.cpu().numpy(), hist_affine.cpu().numpy())   # 
Wasserstein distance

        # Appending the "dist" to the list of distances
        list_of_dist.append(dist)
        # Appending the parameters to the list of parameters
        list_of_params.append((degree, translate_x, scale, shear))

# Finding the parameters for which the distance is minimum
min_index = numpy.argmin(list_of_dist)    # Index for which the Wasserstein distance is minimum
best_degree, best_translate_x, best_scale, best_shear = list_of_params[min_index] # Set of 
parameters corresponding to min_index
print('Min. Wasserstein distance= ', list_of_dist[min_index], '\nBest Parameters:\nDegree= ', 
best_degree, '| translate_x= ', best_translate_x, '| scale= ', best_scale, '| shear= ', 
best_shear)  # Print the parameters

# Creating the transformer with the best parameters
best_affine_transfomer = tvt.RandomAffine(degrees=(best_degree,best_degree), 
translate=(best_translate_x,0), scale=(best_scale,best_scale), shear=best_shear)
best_affine_img = best_affine_transfomer(im_oblique)  # Transformed image that best resembles 
with the direct image
# Display the best image
plt.imshow(best_affine_img)
plt.show()
```

Output:

```
Min. Wasserstein distance=  0.0117758819833339785
Best Parameters:
Degree=  -15 | translate_x=  0.01 | scale=  1.1 | shear=  [15, 15, 0, 0]
```



2. **Using projective parameters:**

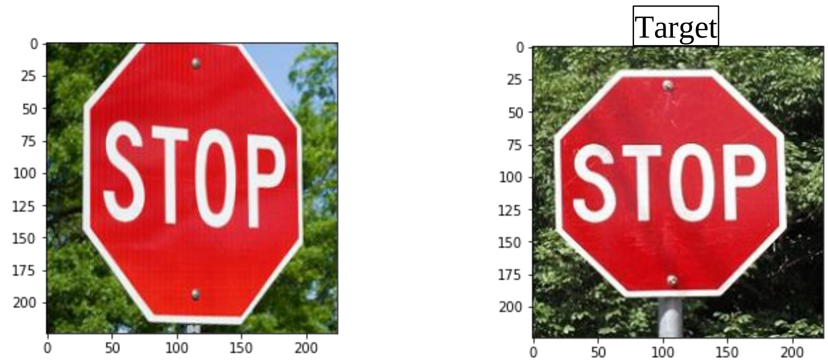


Fig. 4. Best transformed image using *tvt.functonal.perspective()*. The parameters are *startpoints = [[0,0], [223,0], [223,223], [0,223]]*, *endpoints=[[0,0], [230,-60], [223,223], [0,260]]*. The Wasserstein distance between the direct image (Fig. 1(a)) and the transformed image (Fig. 2) is 0.011.

Both the *startpoint* and *endpoint* parameters consist of four corners, *top-left*, *top-right*, *bottom-right*, *bottom-left*, respectively. Each corner is a list of two integers (x, y), e.g., *top-left = [top-left-x, top-left-y]*.

The exploration space of parameters is given below.

- *top-right-x*: {230, 240, 250}
- *top-right-y*: {-80, -70, -60}
- *bottom-left-x*: {-20, -10, 0}
- *bottom-left-y*: {260, 270, 280}

Approach:

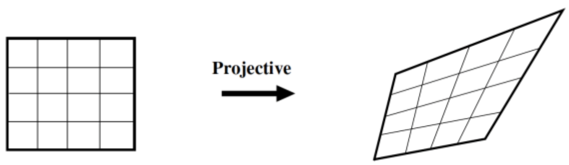


Fig. 5. Projective transform (straight lines hold their behavior after the transform)

Observing Fig. 1(b) reveals that the *top-left* and *bottom-right* corners of it almost resembles the target image in Fig. 1(a). Therefore, by playing with *top-right* and *bottom-left* corners Fig. 1(a) can be obtained from Fig. 1(b). For example, the *top-right* corner of the image in 1(b) needs to be moved along right-upward (north-east) direction to obtain the target image. The *top-right-x* > 224 and *top-right-y* < 0 will satisfy our requirements. Similarly, the parameters are chosen for *bottom-left* corner to move it along the left-downward (south-west) direction.

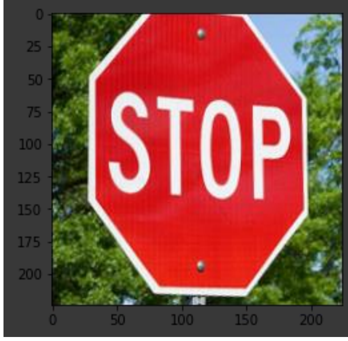
<u>Source code to select the best parameters by minimizing the distance with the target image:</u>

```
list_of_dist = []   # Empty list initialization to store the Wasserstein distance for diff params
list_of_params = [] # Empty list initialization to store different combination of parameters

for top_right_x in range(230,260,10):           # Start loop for the parameter top-right x
coordinate
  for top_right_y in range(-80,-50,10):         # Start loop for the parameter top-right y
coordinate
    for bottom_left_x in range(-20,10,10):      # Start loop for the parameter bottom-left x
coordinate
      for bottom_left_y in range(260,280,10):   # Start loop for the parameter bottom-left y
coordinate

        # Transformed image with proper perspective transform parameters
        perspective_img = tvt.functional.perspective(img=im_oblique, startpoints=[[0,0], [223,0],
[223,223], [0,223]], endpoints=[[0,0], [top_right_x,top_right_y], [223,223],
[bottom_left_x,bottom_left_y]])

        # Calculate the Wasserstein distance
        hist_projective = torch.histc(tvt.ToTensor()(perspective_img), bins=10, min=0.0, max=1.0)
# Histogram with 10 bins
        hist_projective = hist_projective.div(hist_projective.sum())
# Normalize the histogram
        dist = wasserstein_distance(hist_direct.cpu().numpy(), hist_projective.cpu().numpy())
# Wasserstein distance

        # Appending the dist to the list of distances
        list_of_dist.append(dist)
        # Appending the parameters to the list of parameters
        list_of_params.append((top_right_x, top_right_y, bottom_left_x, bottom_left_y))

# Finding the parameters for which the distance is minimum
min_index = numpy.argmin(list_of_dist)    # Index for which the Wasserstein distance is minimum
best_top_right_x, best_top_right_y, best_bottom_left_x, best_bottom_left_y =
list_of_params[min_index]  # Set of parameters corresponding to min_index
```

```
print('Min. Wasserstein distance= ', list_of_dist[min_index], '\nBest Parameters:\ntop_right_x= ', best_top_right_x, '| top_right_y= ', best_top_right_y, '| bottom_left_x= ', best_bottom_left_x, '| bottom_left_y= ', best_bottom_left_y)  # Print the parameters

# Transformed image that best resembles with the original image
best_perspective_img = tvt.functional.perspective(img=im_oblique, startpoints=[[0,0], [223,0], [223,223], [0,223]], endpoints=[[0,0], [best_top_right_x,best_top_right_y], [223,223], [best_bottom_left_x,best_bottom_left_y]])

# Display the best image
plt.imshow(best_perspective_img)
plt.show()
```

Output:

```
Min. Wasserstein distance=  0.011340085603296754
Best Parameters:
top_right_x=  230 | top_right_y=  -60 | bottom_left_x=  0 | bottom_left_y=
260
```



3.3. Creating Our Own Dataset Class

Ten images of different objects are captured using the mobile phone and uploaded to a Google Drive folder. The images are resized to 256×256 and named from ‘0.jpg’ to ‘9.jpg’. The *__len__()* and *__getitem__()* functions are modified as per the instructions. Two important things to be noted here. Although we have only 10 images, in order to create an illusion for the dataloader that we have 1000 images,

- the *__len__()* method returns 1000 (NOT 10);
- and to avoid indexing error while image loading from the disk, *index%10* is used in *__getitem__()* function.

The chosen augmentation transformations, which are suitable for image classification tasks, are *tvt.ColorJitter*, *tvt.RandomGrayscale*, *tvt.RandomHorizintalFlip*.

Source code for custom Dataset class implementation:

```
# Custom dataset class definition
class MyDataset(torch.utils.data.Dataset):
  def __init__(self, root='/content/drive/MyDrive/Arghadip/DL/HW2/Dataset'):
    super().__init__()  # Part of the definition is obtained from parent class
    # Obtain meta information i.e. location of image files
    self.path = root
    # Initialize data augmentation transforms , etc.
    # tvt.Compose collates multiple transforms and perform them sequentially
    self.xform = tvt.Compose([
        # ColorJitter deals with altering the color properties of an image by changing its pixel
values.
        tvt.ColorJitter(brightness=1, contrast=0, saturation=0, hue=0),
        # Converted into grayscale with probability 0.5 for augmentation.
        tvt.RandomGrayscale(p=0.5),
        # Flipped horizontally with probability 0.5
        tvt.RandomHorizontalFlip(p=0.5),
        # Conversion from PIL to floating point Tensor
        tvt.ToTensor()
    ])

  def __len__ (self):
    # Return the total number of images
    # IMP: Although we have only 10 images in our directory, still we want to
    # evaluate the performance of parallel loading of 1000 images using a
    # dataloader object. Therefore the total length is returned as 1000.
    return 1000

  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
    path = os.path.join(self.path, str(index%10) + '.jpg')
    image = Image.open(path)                                    # Load image as PIL object
    image = self.xform(image)                                   # Apply transform
    return (image, random.randint(0,10))                        # Return the image tensor and label
```

Demonstration of MyDataset class:

Test code:

```
my_dataset = MyDataset('/content/drive/MyDrive/Arghadip/DL/HW2/Dataset')  # Creating an instance
print(len(my_dataset))                                                    # Total number of
images
index = 10
print(my_dataset[index][0].shape, my_dataset[index][1])

index = 50
print(my_dataset[index][0].shape, my_dataset[index][1])
```

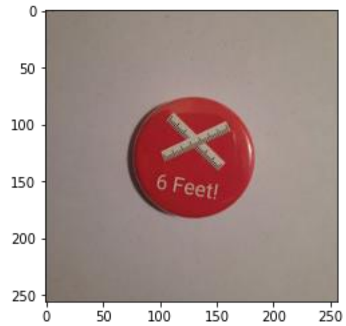
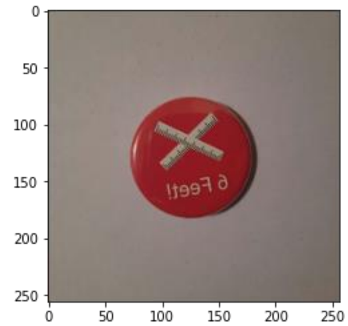
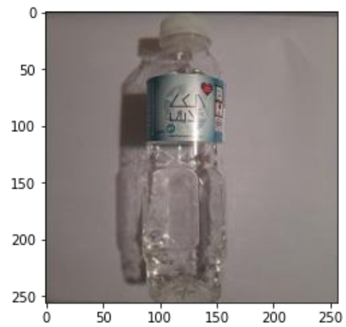
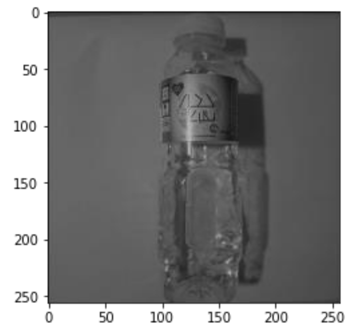
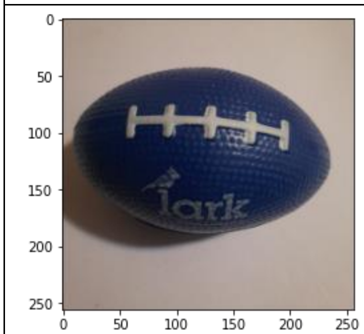
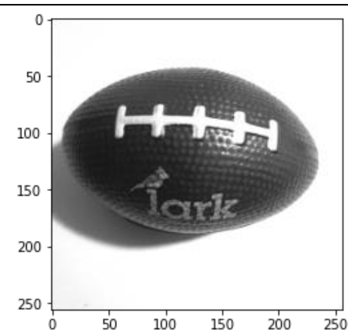
Output:

```
1000
torch.Size([3, 256, 256]) 6
torch.Size([3, 256, 256]) 4
```

Original and augmented images:

Code to plot images:

```
# Code for plotting original and augmented version of images
index = 7   # Change the index to plot a different image

# Plot Original
image = Image.open(os.path.join('/content/drive/MyDrive/Arghadip/DL/HW2/Dataset', str(index) +
'.jpg'))
plt.imshow(image)
plt.show()

# Plot augmented version
plt.imshow(tvt.ToPILImage()(my_dataset[index][0]))
plt.show()
```

Original version	***Augmented version***
	
	
	

Rationale behind the chosen transformations:

Transformation	***Rationale***
tvt.ColorJitter()	It alters the color properties of an image by changing its pixel values. In a real-life scenario, the different illumination leads to different brightness of a captured image. Other parameters like contrast, saturation and hue also varies. Therefore, this augmentation helps in better training of the image classifier network.
tvt.RandomGrayscale()	It converts color images (3 channels, RGB) to gray scale images (1 channel). The image classifier must be able to classify from all types of input images, not only the color images. This type of color augmentation helps in that aspect of traning.
tvt.RandomHorizontalFlip()	This is a geometry-related transform. The image classifier is better trained if it sees images of objects captured from different angles. Therefore, horizontal flip helps in learning those features.

3.4. Generating Data in Parallel

The instance of MyDataset class is wrapped within the *torch.utils.data.DataLoader* class so that the images can be processed in a multi-threaded fashion.

Plotting all images from a batch of 4:

Source code:

```
# Code to plot a batch of 4
import torchvision
# function to show a batch of images
def imshow(img):
    npimg = img.numpy()   # Convert tensor to numpy array
    plt.imshow(numpy.transpose(npimg, (1, 2, 0))) # shaping of array to plot properly
    plt.show()

batch_size = 4    # Setting the batch size = 4
my_dataloader = DataLoader(my_dataset, batch_size=batch_size, shuffle=True, num_workers=0)  #
Creating the data loader
my_dataiter = iter(my_dataloader)   # Creating an iterator from the data loader
data, label = next(my_dataiter)     # Load one batch of data and corresponding labels
imshow(torchvision.utils.make_grid(data))   # Call imshow()
```

Output:

Comparison: multi-threaded DataLoader vs. only Dataset

1. **Time needed to load and augment 1000 images by calling *my_dataset.__getitem__()*:**

   1. ***Source code:***

```
# Getting time for 1000 individual loading using __getitem__()
start_time = time.time()      # Start timer

for i in range(Total_images): # Loop for 1000 images
  my_dataset.__getitem__(i)

end_time = time.time()        # Stop timer

print('Load time (just using Dataset)= ', end_time - start_time, ' seconds')
```

   2. ***Output:***

```
Load time (just using Dataset)=  7.096383094787598  seconds
```

2. **Time needed by *my_dataloader* to process 1000 random images (across different batch sizes and number of workers):**

I create a data loader and an iterator object from that data loader once for a certain *batch_size* and *num_workers*. Then in a *for* loop the iterator is called for *(1000/batch_size)* times. The considered batch sizes and number of workers are shown in the following source code.

   1. ***Source code***

```
# Comparison and perfomance gain for DataLoader
for batch_size in [1,10,20,50,100]:   # Start loop for different batch sizes
  for num_workers in [0,2,4]:         # Start loop for different num_workers
    # Wrapping Dataset instance within a DataLoader
    my_dataloader = DataLoader(my_dataset, batch_size=batch_size, shuffle=True, 
num_workers=num_workers)
    # Creating an iterable from DataLoader object
    my_dataiter = iter(my_dataloader)

    start_time = time.time()          # Start timer

    for i in range(int(Total_images/batch_size)): # Run loop for 1000/batch_size number of 
batches
      next(my_dataiter)               # Iterate through the iterator

    end_time = time.time()            # Stop timer
```

```
    print('Batch size = ', batch_size, '| num_workers= ', num_workers, '| Load time = ', end_time
- start_time, ' seconds')
```

2. ***Output***

```
Batch size =  1 | num_workers=  0 | Load time =  7.533124685287476  seconds
Batch size =  1 | num_workers=  2 | Load time =  6.805963039398193  seconds
Batch size =  1 | num_workers=  4 | Load time =  6.515792608261108  seconds
Batch size =  10 | num_workers=  0 | Load time =  5.805272102355957  seconds
Batch size =  10 | num_workers=  2 | Load time =  4.883432865142822  seconds
Batch size =  10 | num_workers=  4 | Load time =  4.461414575576782  seconds
Batch size =  20 | num_workers=  0 | Load time =  5.8049585819244385  seconds
Batch size =  20 | num_workers=  2 | Load time =  4.582255601882935  seconds
Batch size =  20 | num_workers=  4 | Load time =  4.592945575714111  seconds
Batch size =  50 | num_workers=  0 | Load time =  5.819510459899902  seconds
Batch size =  50 | num_workers=  2 | Load time =  4.447611570358276  seconds
Batch size =  50 | num_workers=  4 | Load time =  4.46168065071106  seconds
Batch size =  100 | num_workers=  0 | Load time =  5.795533895492554  seconds
Batch size =  100 | num_workers=  2 | Load time =  4.647416591644287  seconds
Batch size =  100 | num_workers=  4 | Load time =  4.61956000328064  seconds
```

Load times across different batch_size and num_workers are shown in the table and plot.

batch_size	num_workers	Load time (s)
1	0	4.80918
1	2	4.72898
1	4	5.49303
2	0	4.73202
2	2	4.11033
2	4	3.65303
5	0	4.73659
5	2	3.73318
5	4	3.30234
10	0	4.80384
10	2	3.59341
10	4	3.07502
20	0	4.77154
20	2	3.42828
20	4	3.03153
50	0	4.69958
50	2	3.4184
50	4	3.01213
100	0	4.63261
100	2	3.45138
100	4	3.19065

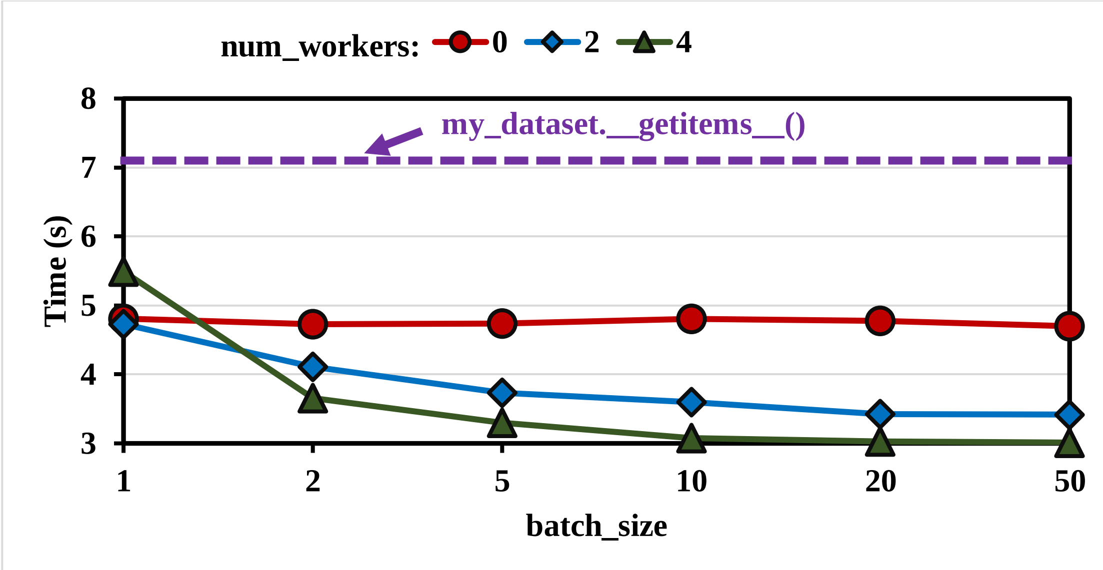


Fig. 6. Load time vs. *batch_size* plot across different *num_workers*

Discussion:

From the plot, as the batch size increases, the load time first reduces and then remains almost same. Increasing batch size means more images are packed together. Therefore, it reduces the number of iterations in the *for* loop. This leads to reduced overhead and thereby reduces the load and processing time. Also, for a certain *batch_size*, if the *num_workers* are increased, then the load time reduces. As more threads join the process, the increased parallelism reduces the load and augmentation time. The results for *batch_size* = 1 shows some random behavior when *num_workers* are changed. However, we still see significant improvement in processing time (7.1s vs. 5.5s) over *my_dataset.__getitem__()* when data loader is used. This can be attributed to the efficient PyTorch back-end implementation of data loaders.

If we compare the data loader results with the previously obtained result for *my_dataset.__getitem__()*, we see significant improvement when *batch_size* and *num_workers* are significantly greater than 1. We get ***~2.33X*** (7.1s vs. 3.0s) speed-up (performance gain) by using data loader with *batch_size = 50 and num_workers = 4*.

4. Lessons Learned

In this homework, we get familiarized with the image representations such as PIL and torch tensor. We also learn the necessary concepts to implement a custom dataset and an image dataloader for parallel loading and processing of data from disk using PyTorch framework.

--- End of the document ---