BME646 and ECE60146: Homework 2

Spring 2023
Due Date: 11:59pm, Jan 25, 2023

Turn in typed solutions via BrightSpace. Additional instructions can be found at BrightSpace. Late submissions will be accepted with penalty: **-10 points per-late-day, up to 5 days**.

1 Introduction

The goal of this homework is to introduce you to the pieces needed to implement an image dataloader within the PyTorch framework for training or testing your deep neural networks. To do so, you will familiarize yourself with image representations such as PIL and torch tensor, and the data augmentation process. Upon completing this homework, you will be able to construct your own image dataloader for training deep neural networks using the `torchvision` library. For more information regarding the image representations and the usage of the `torchvision` library, you can refer to Prof. Kak's tutorial [5]. Note that this homework contains a "theory" task (Sec. 2) followed by the programming tasks (Sec. 3).

2 Understanding Data Normalization

This task is about solving what will appear to be a "mystery" in the Slides 19 through 36 of your instructor's Week 2 lecture slides [5]. In your report, explain in one or two paragraphs what you think would be the explanation to the mystery as described in the rest of this section.

Slides 19 though 36 are meant to give you a deeper understanding of what exactly is computed by the two commonly used callable instances of `tvt.ToTensor` and `tvt.Normalize` for the basic transformations that are typically applied to the images before they can be fed into a neural network. The short name `tvt` stands for the module `torchvision.transforms`.

The Python statement at the bottom of Slide 19 declares a batch of 4 "images", each with 3 channels, and each of size 5×9. As you would expect, the pixel values in these images are unsigned one-byte integers in the range $[0, 255]$, as you can see in the printout for the first image of the batch on Slide 22.

To prepare the 4 images of the batch for input to a neural network, we must first *scale* the pixel values from the $[0, 255]$ integer range to the $[0, 1.0]$ floating-point range. And then *normalize* the floating-point values to the $[-1.0, 1.0]$ range. As to what exactly is meant by normalization will be explained in class.

What you see at the bottom of Slide 26 is a result of manual pixel-value scaling as obtained by dividing all the pixels in the batch by the maximum pixel value in the batch. You will see exactly the same values at the bottom of Slide 28 where the pixel-value scaling was carried out with the `tvt.ToTensor` callable instance.

You will notice that the output shown on Slide 26 was obtained by dividing the pixel values in ALL of the batch images by the max value in the entire batch. As shown by Line (6) in Slide 23, the max value in the batch is in the image indexed 2 (meaning the third image) for its channel indexed 1 (the Green channel) and at location $(3, 3)$.

The similarity of the results shown on Slides 26 and 28 would appear to imply that the pixel-value scaling achieved by `tvt.ToTensor` is nothing more than what you get by dividing the pixel values in all the batch images by the max pixel value in any of the images.

Now here comes the "mystery": The piece of code shown on Slide 28 carries out pixel-value scaling on a per-image basis in the batch. The `for` loop ensures that `tvt.ToTensor` is called on each image separately. So you would think the scaling achieved by this code fragment would divide all the pixel values in an image by the max value in that image. As shown on Slide 23, the largest value in the entire batch does NOT occur in any of the channels in the first image; it occurs in the second channel of the third image (of index 2). So if the pixel values in the first image were to be scaled by the max value in that image, you would NOT get exactly the same answer as shown on Slide 26.

The mystery 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?

3 Programming Tasks

3.1 Setting Up Your Conda Environment

Before writing any code, you will first need to set up an Anaconda [1] environment, in which PyTorch and other necessary packages will be installed.

You should familiarize yourself with the basics of using conda for package management. Nonetheless, what is outlined below will help you get started:

1. A very useful cheatsheet on the conda commands can be found here [2].
2. If you are used to using pip, execute the following to download Anaconda:

   `sudo pip install conda`

   For alternatives to pip, follow the instructions here [3] for installation.
3. Create your ECE60146 conda environment:

   `conda create --name ece60146 python=3.8`
4. Activate your new conda environment:

   `conda activate ece60146`
5. Install the necessary packages (e.g. PyTorch, torchvision) for your solutions

   `conda install pytorch==1.10.0 torchvision==0.11.0 cudatoolkit=10.2 -c pytorch`

   Note that the command above is specifically for a GPU-enabled installation of PyTorch version 1.10. Depending on your own hardware specifications and the drivers installed, the command will vary. You can find more about such commands for installing PyTorch here [4] While GPU capabilities are not required for this homework, you will need them for later homeworks.
6. After you have created the conda environment and installed the all the dependencies, use the following command to export a snapshot of the package dependencies in your current conda environment:

   `conda env export > environment.yml`
7. Submit your `environment.yml` file to demonstrate that your conda enviroment has been properly set up.

3.2 Becoming Familiar with `torchvision.transforms`

This task is about the Data Augmentation material on Slide 37 through 47 of the Week 2 slides on `torchvision`. Review those slides carefully and execute the following steps:

1. Take a photo of a stop sign with your cellphone camera while you are standing directly in front of the sign and the camera is pointing straight at it. Alternatively, you can also take a picture of a photograph of a stop sign instead.

2. Take another photo of the same stop sign, but this time from a very oblique angle. There are two ways you can create the oblique effect: (1) Your camera continues to point in a direction that parallels the road while you stand off to a side. And (2) You continue to point your camera towards the stop sign while you are standing off to a side.

3. Now experiment with applying the callable instance `tvt.RandomAffine` and the function `tvt.functional.perspective()` that are mentioned on Slides 46 and 47 of Week 2 to see if you can transform one image into the other.

4. Note that for measuring the similarity between two images of the stop sign, you can measure the distance between the two corresponding histograms, as explained on Slides 65 through 73.

5. One possible way of solving this problem is to keep trying different affine (or projective) parameters in a loop until you find the parameters that will make one image look reasonably similar to the other.

6. In your report, first plot your two images of the stop sign side-by-side. Subsequently, display your best transformed image, that is the most similar to the target image, using either the affine or projective parameters. Explain in one or two paragraphs on how you have solved this task.

3.3 Creating Your Own Dataset Class

Now that you have become familiar with implementing transforms using `torchvision`, the next step is to learn how to create a custom dataset class that is based on the `torch.utils.data.Dataset` class for your own images. Your custom dataset class will store the meta information about your dataset and implement the method that loads and augments your images. The code snippet below provides a minimal example of a custom dataset within the PyTorch framework:

```
1 import torch
2
3 class MyDataset(torch.utils.data.Dataset):
```

```
4
5     def __init__(self, root):
6         super().__init__()
7         # Obtain meta information (e.g. list of file names)
8         # Initialize data augmentation transforms, etc.
9         pass
10
11    def __len__(self):
12        # Return the total number of images
13        return 100
14
15    def __getitem__(self, index):
16        # Read an image at index and perform augmentations
17        # Return the tuple: (augmented tensor, integer label)
18        return torch.rand((3, 256, 256)), random.randint(0, 10)
```

Before proceeding, take ten images with your cellphone camera of any object and store them together within a single folder. Now, based on the code snippet above, implement a custom dataset class that handles your own images. More specifically, your `__getitem__` method should:

1. Read from disk the image corresponding to the input index as a PIL image.

2. Subsequently, assuming that you are using your custom dataset to train a classifier, augment your image with three different transforms of your choice that you think will make your classifier more robust. Note that a suitable transform could be either color-related or geometry-related. Note that you should use `tvt.Compose` to chain your augmentations into a single callable instance.

3. Finally, return a tuple, with the first item being the tensor representation of your augmented image and the second the class label. For now, you can just use a random integer as your class label.

The code below demonstrates the expected usage of your custom dataset class:

```
# Based on the previous minimal example
my_dataset = MyDataset('./path/to/your/folder')
print(len(my_dataset))  # 100
index = 10
print(my_dataset[index][0].shape, my_dataset[index][1])
# torch.Size([3, 256, 256]) 6

index = 50
```

```
print(my_dataset[index][0].shape, my_dataset[index][1])
# torch.Size([3, 256, 256]) 8
```

In your report, for at least three of your own images, plot the original version side-by-side with its augmented version. Also briefly explain the rationale behind your chosen augmentation transforms.

3.4 Generating Data in Parallel

For reasons that will become clear later in this class, training a deep neural network in practice requires the training samples to be fed in batches. Since calling `__getitem__` will return you a single training sample, you now need to build a dataloader class that will yield you a *batch* of training samples per iteration. More importantly, by using a dataloader, the loading and augmentation of your training samples is done efficiently in a multi-threaded fashion.

For the programming part, wrap an instance of your custom dataset class within the `torch.utils.data.DataLoader` class so that your images for training can be processed in parallel and are returned in batches. In your report, set your batch size to 4 and plot all 4 images together from the same batch as returned by your dataloader.

Additionally, compare and discuss the performance gain by using the multi-threaded `DataLoader` v.s. just using `Dataset`. First, record the time needed to load and augment 1000 random images in your dataset (with replacement) by calling `my_dataset.__getitem__` 1000 times. Then, record the time needed by `my_dataloader` to process 1000 random images. Note that for this comparison to work, you should set both your `batch_size` and `num_workers` to values greater than 1. In your report, tabulate your findings on the timings and experiment with different settings of the `batch_size` and `num_workers` parameters.

4 Submission Instructions

Include a typed report explaining how did you solve the given programming tasks.

1. Turn in a zipped file, it should include (a) a typed self-contained pdf report with source code and results and (b) source code files (only .py files are accepted). Rename your .zip file as hw2_<First Name><Last Name>.zip and follow the same file naming convention for your pdf report too.

2. For all homeworks, you are encouraged to use `.ipynb` for development and the report. If you use `.ipynb`, please convert it to `.py` and submit that as source code.

3. You can resubmit a homework assignment as many times as you want up to the deadline. Each submission will overwrite any previous submission. **If you are submitting late, do it only once on BrightSpace.** Otherwise, we cannot guarantee that your latest submission will be pulled for grading and will not accept related regrade requests.

4. The sample solutions from previous years are for reference only. **Your code and final report must be your own work.**

5. Your pdf must include a description of

   - Your explanation to the mystery as described in Sec. 2.
   - The various plots and descriptions as instructed by the subsections in Sec. 3.
   - Your source code. Make sure that your source code files are adequately commented and cleaned up.

References

[1] Anaconda, . URL https://www.anaconda.com/.

[2] Conda Cheat Sheet, . URL https://docs.conda.io/projects/conda/en/4.6.0/_downloads/52a95608c49671267e40c689e0bc00ca/conda-cheatsheet.pdf.

[3] Conda Installation, . URL https://conda.io/projects/conda/en/latest/user-guide/install/index.html.

[4] Installing Previous Versions of PyTorch. URL https://pytorch.org/get-started/previous-versions/.

[5] Torchvision and Random Tensors. URL https://engineering.purdue.edu/DeepLearn/pdf-kak/Torchvision.pdf.