Assignment 10

Please refer to MyCourses/Gradescope for assignment deadlines.

The goal of this assignment is to get practice with gradient descent in PyTorch to solve two different kinds of problem:

1. Triangulation: an optimization problem for depth inference on a single image. We'll infer the 3d location of points by gradient descent on reprojection error.
2. Logistic regression: the simplest machine learning model we'll train by gradient descent for handwritten digit recognition.

Part 1 of 3: Triangulation by gradient descent

In this part of the assignment, you'll implement a simple triangulation algorithm to infer the 3D location of points in a scene from correspondences in two 2D images. In Assignment 08, you used `cv.triangulatePoints`, which uses some linear algebra tricks to try to solve for 3D locations from 2D points in closed form. This is *fast* because it doesn't require any iterative adjustements, but it is not always numerically stable.

The file `triangulate.py` is organized into (1) `Camera` class and (2) the triangulation logic. The camera class holds intrinsic and extrinsic parameters of a simple pinhole camera model (if you want to use something like this in practice, you should include the nonlinear lens distortion parameters as well and infer the camera parameters with a calibration procedure).

Recall the perspective projection equation:

$$\tilde{\mathbf{x}}_s = \mathbf{K} \begin{bmatrix} \mathbf{R} & \mathbf{t} \end{bmatrix} \bar{\mathbf{p}} \,. \tag{1}$$

where $\mathbf{p}$ is a point in world coordinates, $\tilde{\mathbf{x}}_s$ is the point in image coordinates, $\mathbf{K}$ is the intrinsic camear calibration matrix, and $\mathbf{R}$ and $\mathbf{t}$ are the camera extrinsics. We'll adopt the OpenCV convention here and use a 3-parameter calibration matrix

$$\mathbf{K} = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

where f_x is the focal length in pixels, and (c_x, c_y) is the image center in pixels. We'll also adopt the OpenCV convention (unlike the book) of pointing the optical axis in the positive Z direction and thinking of the Y axis as pointing "down" to match the image coordinate system.

The `triangulation.py` script will run (and animate!) gradient descent on the reprojection error for a synthetic scene where we know the ground truth 3D points and camera parameters. This lets us simulate minimization of reprojection error while guaranteeing that we've solved the correspondence problem exactly. The main exercise here is about using torch and playing around with some optimization tricks.

Expected outputs

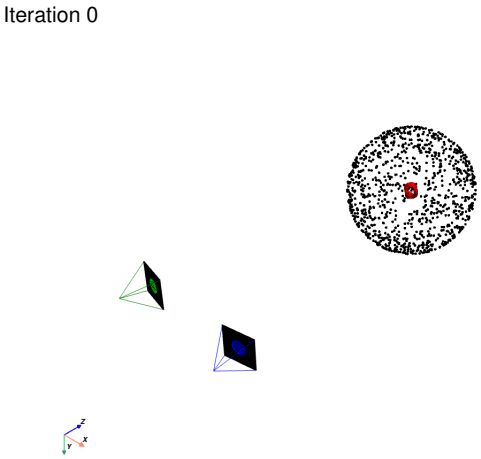
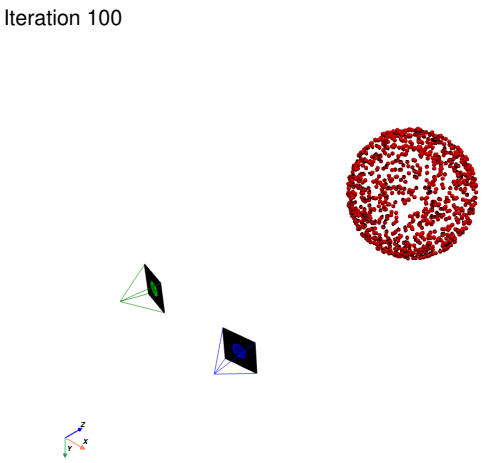
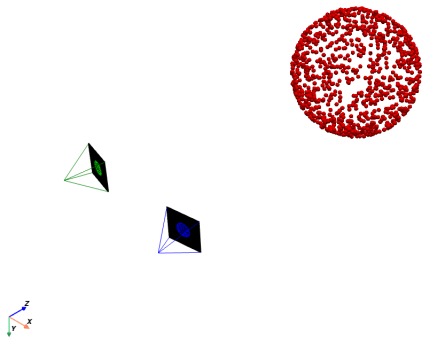


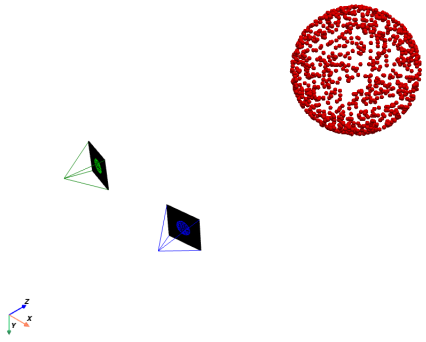
Figure 1: Example of animated triangulation, Step 0. Black points are 'true' locations, red points are the inferred 3D locations, being updated by gradient descent.



Iteration 200



Iteration 300



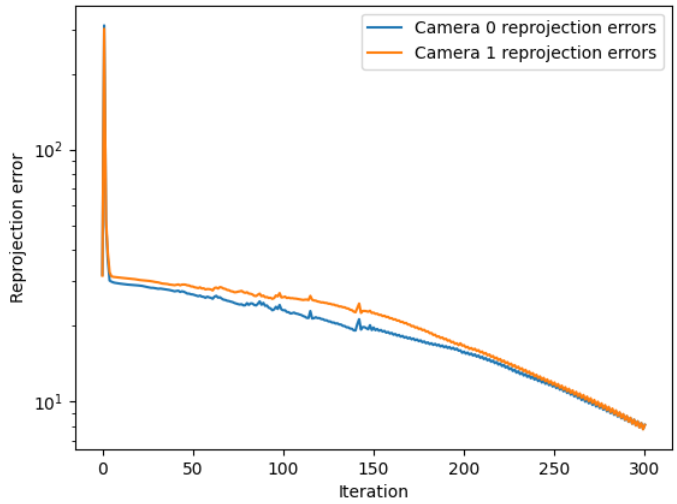


Figure 2: Loss curves WITHOUT gradient clipping (note that it diverges at first)

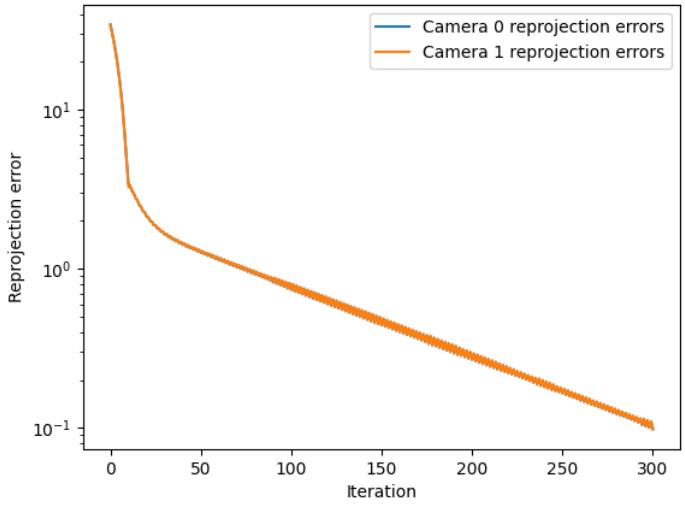


Figure 3: Loss curves WITH gradient clipping

Part 1.1

The first thing you need to do is implement `Camera.perpective_projection`. This function must implement equation (1) using torch functions. Note that, like numpy, you can use basic python operations like `+`, `-`, `@`, etc. and this will work with torch tensors.

The cool thing about implementing the optics in torch is that it is differentiable with respect to any of its inputs. Here, we're going to take the camera parameters as *given* and solve for the 3D coordinates of the points, but in principle we could also run gradient descent taking the 3D points as given and solving for the camera parameters!

Part 1.2

Next, you need to implement `reprojection_error`, which uses `Camera.perspective_projection` to project 3D points to 2D and then returns the Euclidean distance on the image plane between the given 2D points and the projected points. This is the "cost function" that we want to minimize per point and per camera. We require that this function returns a tensor of size `(N,)`, i.e. it should give one error per input point. The caller will then have the job of "reducing" the set of errors per point and per camera to a single scalar value to be optimized.

Part 1.3

Next, you need to fill in the missing parts of `triangulate_by_gradient_descent`. The core of this function is an **optimization loop**. It iteratively refines the 3D points using gradient descent. For the purposes of this exercise, we've set up the pre- and post-processing and history-tracking for you. For the purposes of this assignment, you're not allowed to change the optimizer we're using. We'd prefer that you go through the exercise of gradient clipping and learning rate scheduling to get the process to converge. You need to implement the following in the missing code block

- Aggregate all the errors per-point and per-camera into a single scalar. This is called "reduction" of the loss function in ML lingo. Here, we insist that you use the `sum` of the errors.
- Use torch's `backward` function to compute the gradients of the aggregate cost with respect to the things we are optimizing (i.e. the 3D points).
- Clip the gradients (see part 1.4 below)
- Update the 3D points using `Optimizer.step()` (where the `Optimizer` we've set up is an instance of `torch.optim.SGD`).

At this point, you should have a semi-working optimization loop, but you'll find that it is prone to diverging or very sensitive to your choice of step size.

Part 1.4: Make convergence more reliable

The last two "tricks" you need to implement to make the optimization more robust are:

- Use a "Learning rate scheduler" to reduce the step size over time. Try

```
scheduler = torch.optim.lr_scheduler.StepLR(opt, step_size=1, gamma=0.99)
```

and calling `scheduler.step()` after each call to `opt.step()`. This will reduce the learning rate by a factor of 0.99 every step. This is a classic trick in optimization: by taking smaller and smaller steps, an optimizer stops 'bouncing around' the solution and starts to settle to a good minimum. You'll see this reduction in 'bouncing' in the animation or in the loss plot.

- Implement `clip_gradient_norm` by following its docstring. The idea is that sometimes the `param.grad` values can be very large, resulting in very big updates to the parameters. Combatting this just by reducing the step size might just slow down the process. Instead, a common practice is to "clip" the gradient vectors to a maximum length.

  Once you've implemented `clip_gradient_norm`, you should figure out where to call it in the optimization loop (hint: after gradients are calculated, but before the parameters are updated).

Part 2 of 3: Datasets and DataLoaders

In this part of the assignment, you'll briefly poke around the MNIST and CIFAR-10 datasets using the `torchvision` library. You'll write two functions whose job it is to simply print out some basic information about a given dataset.

In PyTorch, a `Dataset` class is a simple way to represent a collection of data samples. Custom `Datasets` are super easy to implement; essentially, any object that subclasses `torch.utils.data.Dataset` and provides an implementation of the `__len__` and `__getitem__` methods can be used as a dataset. For instance, you could write a custom `Dataset` class to load data from a CSV file, or from a directory of images, or from a database. As easy as that is, we're going to do something even easier and use some of the built-in datasets that PyTorch (specifically the `torchvision` library) provides.

Let's say you call `ds = MyDataset()`. Then, you could access the `i`-th sample in the dataset by calling `ds[i]`, which would call the `__getitem__` method of the `MyDataset` class. However, this isn't how we use datasets in practice, for a few reasons:

1. we'll want to get an entire *batch* of samples at a time,
2. it's useful to shuffle the dataset so that we don't always see the samples in the same order, and
3. if the `__getitem__` method is slow (e.g., if it's reading from disk), we'd ideally run the slow i/o operations for the next batch in the background while the model is training on the current batch.

All of this is handled by the `DataLoader` class in PyTorch. The `DataLoader` class wraps a `Dataset` object and provides an iterator that returns batches of samples. It also handles shuffling, batching, and (if you set `num_workers` to a value greater than 0) running the slow `__getitem__` method in parallel with the model training. It looks like this:

```
dl = DataLoader(ds, batch_size=32, shuffle=True, num_workers=4,
↪  pin_memory=True)
```

The `pin_memory` argument is useful if you're using a GPU and can speed up data transfer to the GPU. You can read about how it works **here** if you're curious.

Instructions

You must run this code on the CS department servers. The datasets have already been downloaded for you and placed in `/local/sandbox/csci631/datasets`. **Do not download your own copy of these datasets!** The location of the dataset files is configured in `config.py`, which you should not modify except perhaps temporarily for local debugging purposes on your own machine.

Fill in the missing code in `dataset_info.py` to print out useful diagnostic information about a dataset or dataloader. The string formatting is done for you, you just need to inspect the properties of the `Dataset` and `DataLoader` object. *Do not modify the string in what you submit so that we can unit-test it*. Run the `dataset_info.py` script with `--dataset mnist` or `--dataset cifar10` to see some information about each one.

Example output

Output from running `python dataset_info.py --dataset mnist`:

Dataset with 60000 samples, 10 classes, image shape torch.Size([1, 28, 28]) and dtype torch.float32. DataLoader with 60000 total samples split across 1875 batches of size 32. Batch shape is torch.Size([32, 1, 28, 28]).

Part 3 of 3

In this part of the assignment, you'll train a simple **logistic regression** model to classify small hand-written digits (MNIST dataset) and small color images (CIFAR-10 dataset). This is one of the simplest model types we can use for image classification. We're using it here so that

1. You can get experience with PyTorch's image handling and training process without the added complexity of big fancy models, and
2. You get a good sense of the baseline performance of a simple model on these datasets, so that later when we use more complex models, you have a reference point to compare to.

Instructions

1. Fill in the missing code in `models.py`. You must implement the `__init__` and `forward` methods of the `LogisticRegression` class. The `__init__` method should initialize the weights and biases of the model, and the `forward` method should compute the output of the model given an input tensor. Logistic regression from data X to labels y is given by the equation $p(y) = \sigma(z)$, where $z = WX + b$ and σ is the softmax function. This maps from an input vector X to a probability distribution over class labels. Importantly, a single `nn.Linear` layer accomplishes the $WX + b$ part of the equation, and we're actually going to return z from the `forward` method, *not* returning probabilities, so the model code is actually a lot simpler than the math looks.

2. Fill in the missing code in `train.py`, and study the code that is already provided for you there. Your primary task is to implement the `train_single_epoch` function, which is called by the `train` function. The `train_single_epoch` function should train the model for one epoch (i.e., one loop over the dataloader). It should log the accuracy and loss of the model on the training set in the provided Tensorboard `SummaryWriter` object. Use the `evaluate()` function as a reference. The primary differences between `train_single_epoch` and `evaluate` are that

   - `train_single_epoch` should call `model.train()` at the start rather than `model.eval()`
   - it should call `optimizer.zero_grad()` at the start of each batch
   - it should call `loss.backward()` and `optimizer.step()` at the end of each batch
   - it should log the loss and accuracy to the `SummaryWriter` every batch
   - it should increment `step` each batch
   - it should not have a `with torch.no_grad()` block (we do want grads while training!)

3. Run the training script **at least** four times: twice for MNIST and twice for CIFAR-10, using at least two different learning rates. You can use the `--dataset` and `--lr` command-line flags to specify the dataset and learning rate, respectively. For instance, to train on CIFAR-10 with a learning rate of 0.01, you would run `python train.py --dataset cifar10 --lr 0.01`. You can also specify the number of epochs to train for with the `--epochs` flag. Remember, the goal at this stage is not to make the best or fanciest model. It's to get comfortable with the training process and to get a sense of how well a simple model can do on these datasets.

   Earlier, you used a Tensorboard `SummaryWriter` object to log the loss and accuracy of the model over the course of training. Tensorboard is a great tool for visualizing how your model training is progressing. (Other tools for this include `WandB`, `MLFlow`, etc. We're using tensorboard here because it's one of the easier ones to get started with.) It works by writing logs to a specified directory, which you can then view as interactive graphs in a browser window. This

requires launching a tensorboard server on the CS machine. To start the tensorboard server, SSH into the CS department server, activate your virtual environment where tensorboard is installed, and run

```
tensorboard --bind_all --logdir=logs --port=6006
```

Then, open a browser window and navigate to `http://<servername>.cs.rit.edu:6006` to see the Tensorboard UI. **Note: only one person can run a tensorboard server on a given port at a time. If you get an error message about the port being in use, try a value other than 6006. When you're done, kill your tensorboard server to free up ports and processing for other students!** Port 6006 is the default for tensorboard, but you can use 6007, 6008, etc. if someone else is already using the others.

Once you've successfully trained at least 2 models on each dataset with different hyperparameters, take a screenshot of the Tensorboard UI and include it in your submission in the `images` directory.

Expected output

Here's what a functioning tensorboard UI might look like after training two MNIST models with different learning rates:

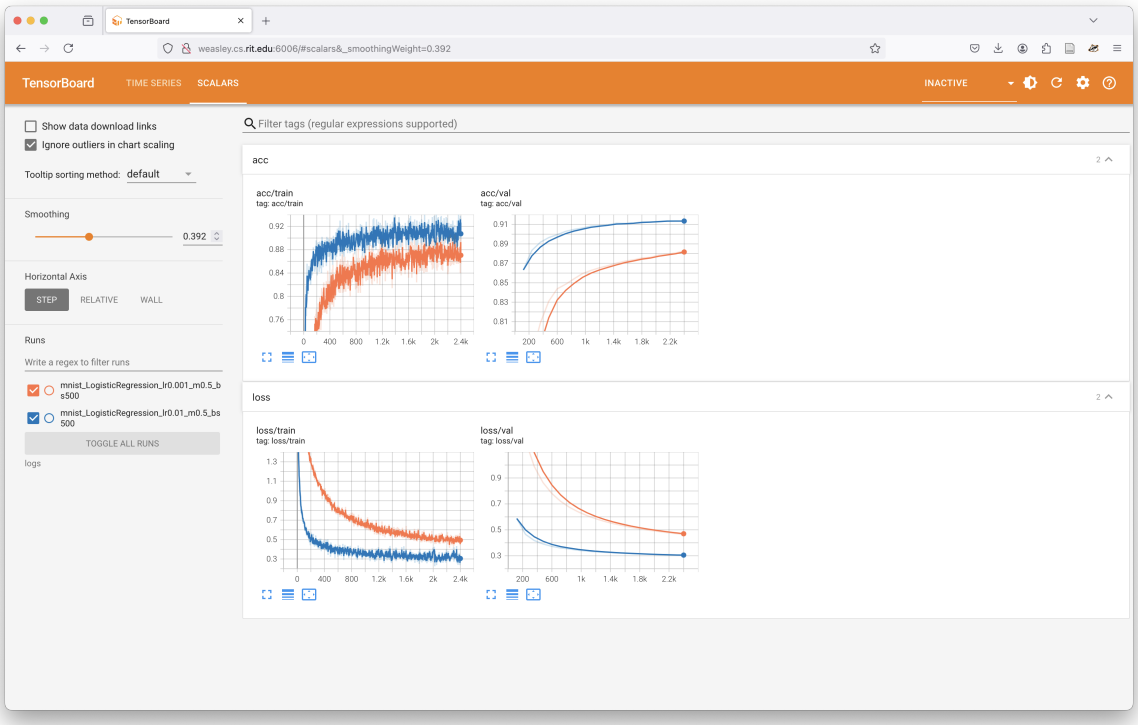


Figure 4: Example tensorboard screenshot

Your screenshot(s) must also include loss and accuracy plots for logistic regression trained on the CIFAR-10 dataset.

Collaboration and Generative AI disclosure and Assignment Reflection

Did you collaborate with anyone? Did you use any Generative AI tools? How did the assignment go? Briefly explain what you did in the `collaboration-disclosure.txt` file. **Completing the collaboration disclosure for each assignment is required.** Completing the reflection part of the file is optional, but it is a good way to give feedback to the instructor and grader about how the course is going.

Submitting

As described above, use `make submission.zip` to generate a zip file that you can upload to Gradescope. You can upload as many times as you like before the deadline. To account for late penalties, contact the instructor or grader directly if you plan on uploading any further revisions after the deadline.