Assignment 09

Please refer to MyCourses/Gradescope for assignment deadlines.

The goal of this assignment is to give you hands-on practice with the following concepts:

- Motion and optical flow
- Getting started with running code on GPUs using PyTorch

Part 1 of 3: DIY coarse-to-fine optical flow

To start, go to [https://vision.middlebury.edu/flow/data/] and download the `other-color-twoframes.zip` file. This contains two-image pairs of frames that we will use to compute optical flow. The images are part of a public dataset that is well-known in the field.

We are going to implement our own coarse-to-fine optical flow algorithm using only numpy and low-level OpenCV functions. Here's a high-level view of the logic:

1. `solve_optical_flow_constraint_equation_lucas_kanade(Ix, Iy, It, window_size, alpha)`
    - This function is given to you. It uses numpy's `solve` function to solve for `[u, v]` in the optical flow constraint equation.
    - The 'sliding window' feature is implemented in vectorized form using a numpy indexing trick called `sliding_window_view`. This is a useful trick for vectorizing operations over a 2D array. (Think back to assignment 3 and how this would have been a fast solution for correlation!)
    - The input gradients (Ix, Iy, It) describe how pixel intensity changes in space and time, and are passed in as `(h, w)` arrays.
2. `warp_by_vector_field(image, vectors_uv)`
    - This function warps an image based on the provided vector field. In other words, the vector field describes the motion of each pixel in the image, and this function should calculate a new image where each pixel is displaced according to the vector field.
    - Use `cv.remap` with `cv.INTER_LINEAR` here (you need to figure out the appropriate inputs)
3. `calculate_gradients(image1, image2)`
    - Computes spatial (Ix, Iy) and temporal (It) gradients between two images.
    - To be consistent with the answerkey, you should

- use `cv.Sobel` with 3x3 filters for the x and y gradients, averaging across the two frames.
- pay close attention to the `scale` argument to the `Sobel` function. Back when we did edge detection, we cared much less about the units of the gradients. Here, the optical flow constraint equation is very sensitive to the scale of the spatial and temporal gradients. Make sure the spatial gradients are scaled so they have units of "change in brightness per pixel"
- We cannot use Sobel for the temporal gradient because we have only two frames. This actually makes it easier in some ways – we just need to estimate the "change in brightness per change in time", where the change in frame Δt is simply 1. However, for consistency with the spatial gradients, you should still use a 3x3 tent over the spatial dimensions (i.e. a 3x3 gaussian blur) to smooth out the time gradient estimates.

4. `def coarse_to_fine_optical_flow(image1, image2, levels, window, alpha)`

   - This function does the equivalent of `cv.calcOpticalFlowPyrLK()`, but using the functions above. It should take two images and compute the optical flow between them using a coarse-to-fine approach. This function is called by `calculate_optical_flow_pyr_lk_cross_check()`, which does the consistency check, so this function by itself is expected to calculate *some* flow vector for every pixel.
   - Logically, the pseudo-code here is something like

```
1. create a Gaussian pyramid for each image
2. starting at the top (smallest) level of the pyramid, calculate the
   optical flow.
3. For each subsequent level,
   1. upsample the flow from the previous level (pay attention to scale!)
   2. *warp* image1 using the upsampled flow from the previous level
   3. calculate the "residual" optical flow at this level after warping
   4. update the total flow estimate
4. Finally, rescale the flow to the original image size (because the
  gaussian pyramid may have included some upscaling). Again pay attention
  to how rescaling the height/width of the flow field requires also
  adjusting the vectors themselves.
```

5. `def calculate_optical_flow_pyr_lk_cross_check(...)`

   - This function calls `coarse_to_fine_optical_flow` twice, once in the forward direction (image1 -> image2) and once in the reverse direction (image2 -> image1). It then

checks that the flow vectors are consistent in both directions. If they are not, it sets the flow vector to `nan`. This is a form of "cross-check" to ensure that the correspondence problem is solved in both directions at once. Where it isn't, we assume that the flow is not valid.

- Logically, you need to do three things:
  - Treat each `uv` value as a 2D vector. You have a `uv` in the forward direction and a `uv` in the reverse direction. when the flow is *good*, we expect `uv_forward = -uv_reverse`, or close to it. You should calculate the norm of the "difference" between the two flows, where I'm putting "difference" in quotes because the minus-sign is important. The `np.linalg.norm` function will calculate the norm for you.
  - Wherever the difference is greater than the `goodness_threshold`, set the output flow to `nan`
  - Everywhere else, set the output flow to the "average" of the two flows. This is a form of smoothing that helps to reduce noise in the flow field. I've again put "average" in quotes because you should be careful about the signs of the vectors.

If it works, you should see an output like this when passing the `Hydrangea` images from the Middlebury dataset:

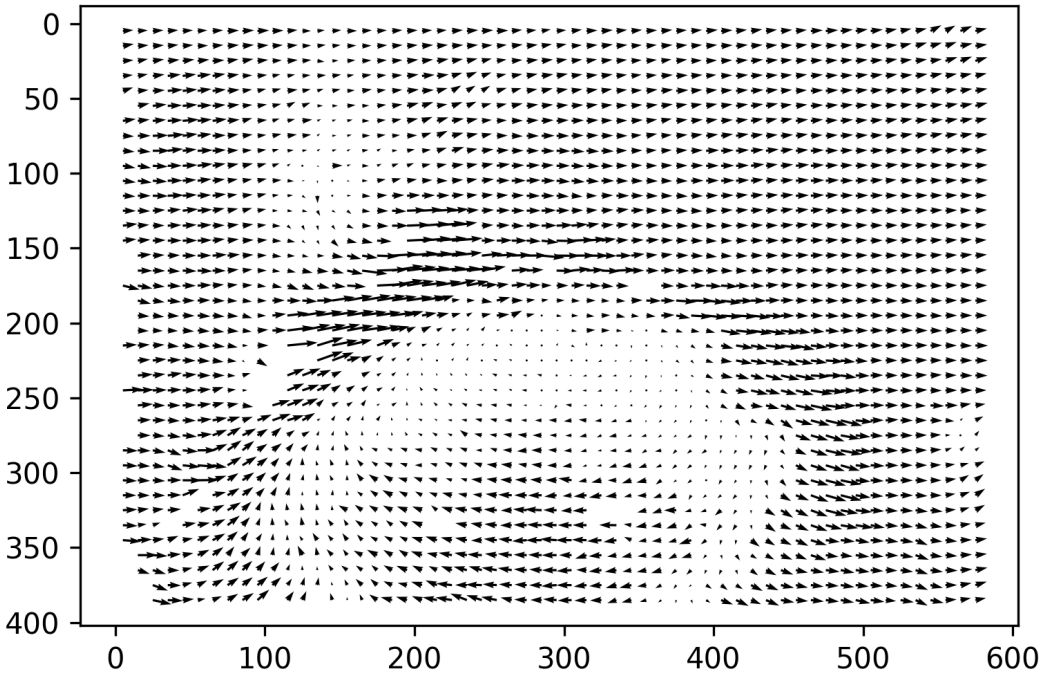

Figure 1: Hydrangea quiver plot

Note that we've provided you two different ways to visualize flow fields in `utils.py`, one with HSV and one with a quiver plot.

Part 2 of 3: Connect your IDE to the CS department servers

Moving forward, we're going to start using PyTorch and running code on GPUs. This can in principle be done on your personal computer (if it has CUDA support), but regardless of the hardware you have, it's a useful skill to know how to configure your IDE for local development and remote deployment.

You should have access to the following servers in the CS department:

- `granger.cs.rit.edu`
- `weasley.cs.rit.edu`
- `lovegood.cs.rit.edu`

These servers are a shared resources with your classmates and with students in other classes. **It is up to you to use good judgment and not abuse these resources.** For starters, I recommend flipping a 3-sided coin to randomly decide which server you'll use by default as a matter of load-balancing.

We'll assume you know the basics of connecting over SSH and running simple linux commands for navigating the filesystem, creating directories, editing plain text files in the shell, etc. If you don't, you should take a moment to familiarize yourself with these concepts.

If using PyCharm, you need to fix a configuration bug before proceeding. Follow these steps:

1. SSH into one of the above servers
2. Open your `~/.bashrc` file in a text editor
3. Paste the following block of code at the end of the file:

```
# PYCHARM-OVER-SSH-FIX.
# The problem: if you run PyCharm on your local machine but configure
↪  it to
# use an "SSH interpreter" (run python here on the server), it
↪  configures
# things by default in /tmp/cache/. However, if one person configures
↪  things
# in /tmp/cache/, then nobody else will have write permissions there,
↪  and
# PyCharm will fail to set up the remote interpreter correctly.
# The solution: set a separate TMPDIR environment variable for each
↪  user
# *and make sure the directory exists using mkdir*.
export TMPDIR=/tmp/${USER}
mkdir -p $TMPDIR
```

If you are using a different IDE (Spyder, VSCode, etc.), you may or may not need to do something similar. Your instructor uses PyCharm, so you're on your own for other IDEs.

Next, you should configure your IDE with a "remote interpreter" and automatic upload of files. The idea is that any changes you make to `.py` files on your local machine will be automatically uploaded to the server[1], and any time you "run" or "debug" a script, the code will be executed on the server. This is super useful when doing things like training machine learning models using server hardware but stepping through line by line in your IDE's debugger.

Here's what this configuration looks like in PyCharm (again, other IDEs will have a similar but different process, for which you're on your own):

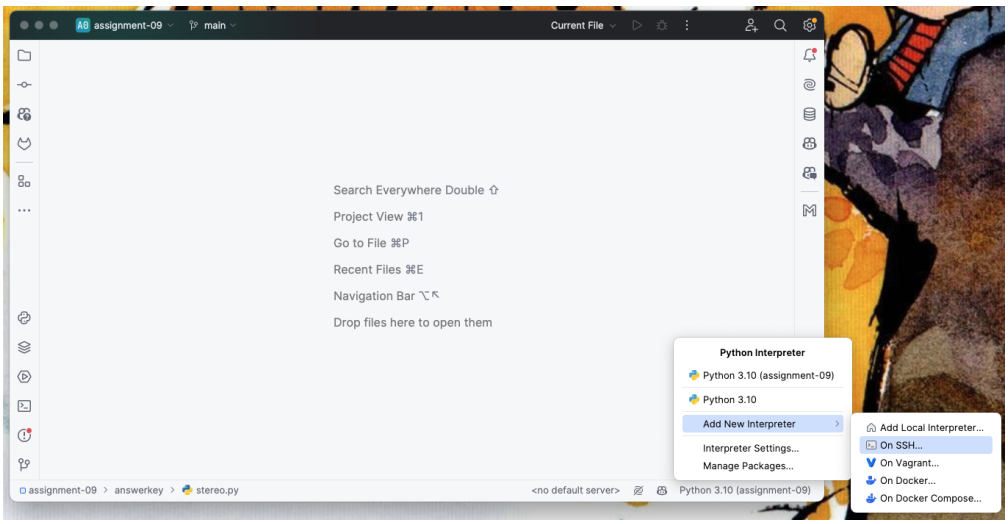

Figure 2: PyCharm remote interpreter configuration Step 1

[1] careful – this is often configured asymmetrically, so changes on the server are not necessarily automatically downloaded back to your machine!

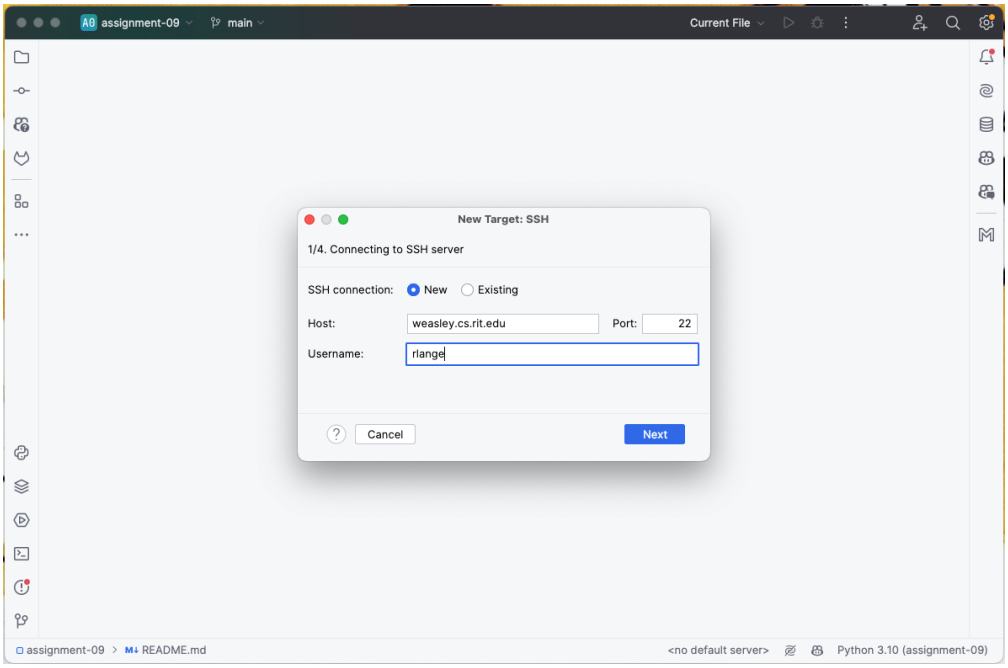

Figure 3: PyCharm remote interpreter configuration Step 2a ("New" SSH configuation if it's your first time connecting to this server)

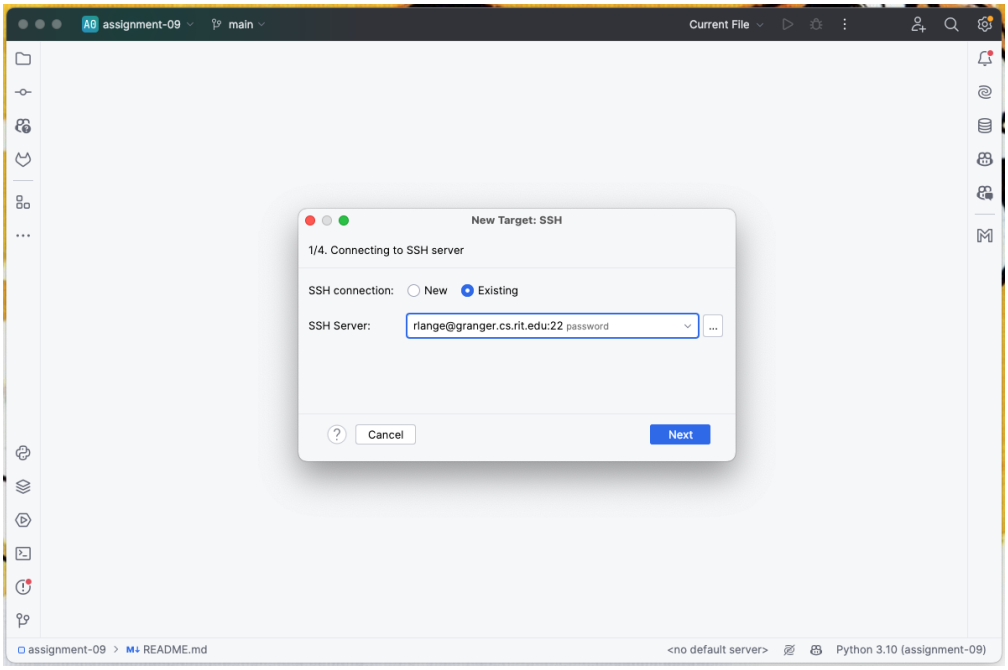

Figure 4: PyCharm remote interpreter configuration Step 2b ("Existing" SSH configuration if you've used this server before (including in other projects))

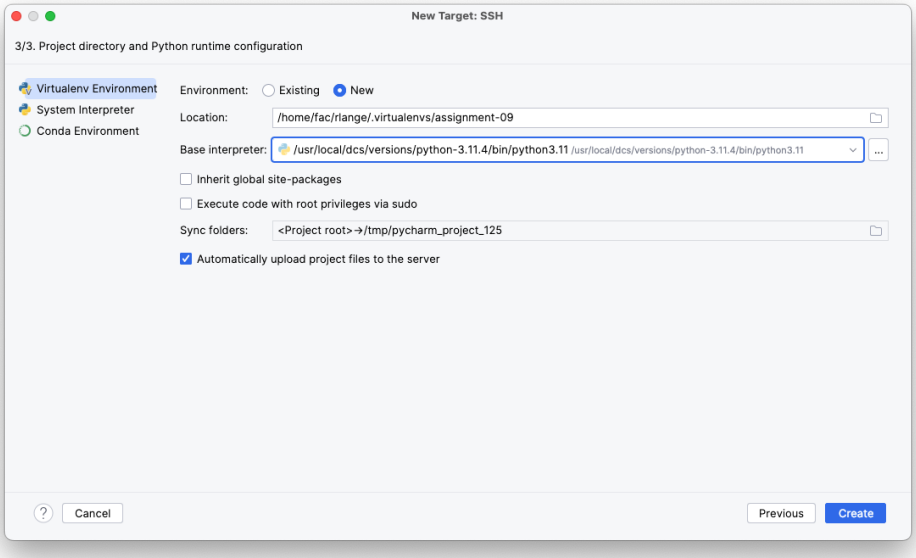

Figure 5: PyCharm remote interpreter configuration Step 3. **Note that the "base interpreter" is not the system default. You should use Python 3.10 or Python 3.11 for this course, both of which are available on the servers in `/usr/local/dcs/versions` as shown here.** Also note that the "Automatic upload" box is checked. This ensures that when you save changes to a file locally, they are automatically sent to the server. Finally, the "sync folders" option defaults to a random `/tmp/` directory on the server, but can be reconfigured to your home directory. This tells PyCharm where to upload files, and where to base its relative paths off of.

Troubleshooting

Server configuration doesn't always go smoothly. Here's what you need to know about PyCharm: the steps above result in three changes to your IDE configuration, each of which can be tweaked after the fact. In PyCharm's settings, you should check issues in three places:

1. Check that you're actually connecting over SSH. To do this, you can open the "Tools > SSH Configurations" OR the "Build, Execution, Deployment > Deployment" tab in the settings and click the "Test Connection" button.

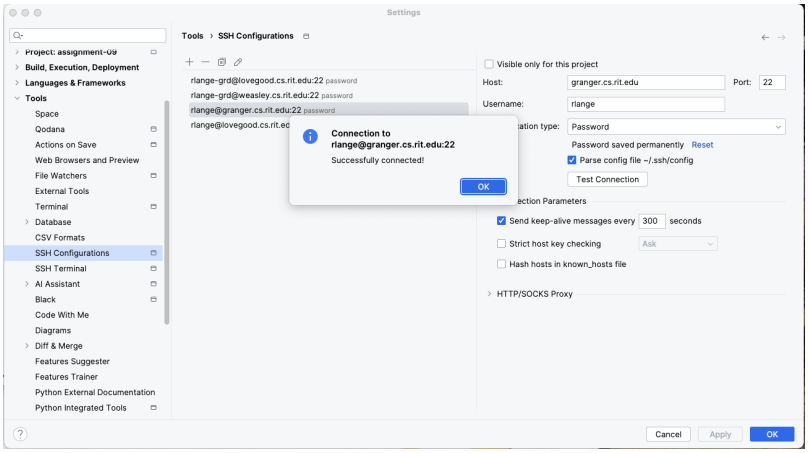

Figure 6: SSH Configuration Test Connection

2. Check that your “Deployment” configuration is correct. This is where you set up where files are uploaded and manage the automatic upload settings. The “Mappings” tab defines which local folders are synced with which remote folders.

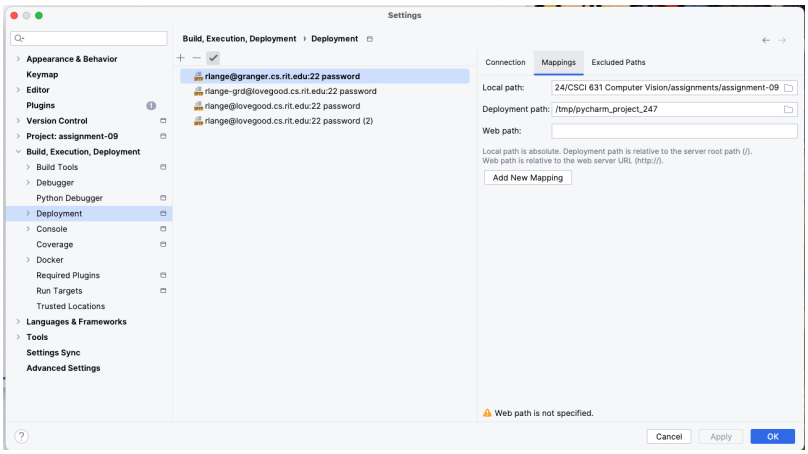

Figure 7: Deployment Configuration

3. Check that your “Python Interpreter” is set up correctly. This is where you set the Python interpreter that will be used to run your code. You can find this in the “Project: > Python Interpreter” settings.

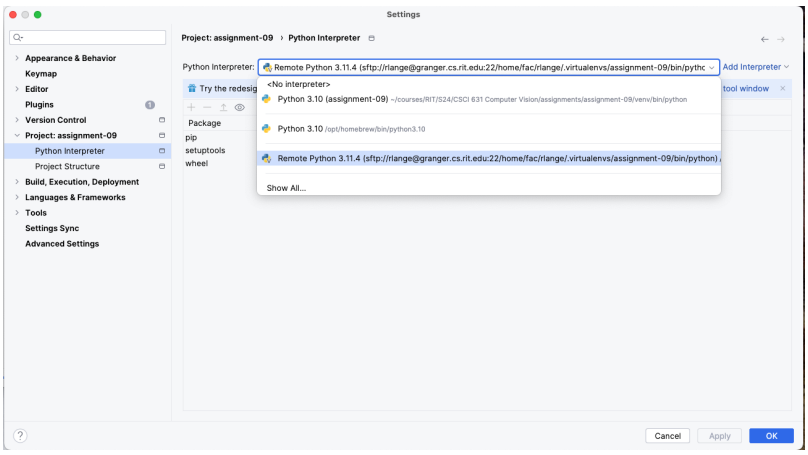

Figure 8: Python Interpreter Configuration

Testing your configuration / proving it works

Once you have your IDE set up to run code on the servers, you should test it. Using your IDE *running locally on your machine, but with the code executing on the server*, run the `gpu_test.py` scipt that we've provided. Then, take a screenshot showing your (local) IDE executing the code remotely and place that screenshot in an `images/` directory.

When you run `make submission.zip`, all `.png` and `.jpg` files in `images/` will automatically be included and uploaded to Gradescope.

Part 3 of 3: Getting started with PyTorch

Now that you have your IDE set up to run code on the servers, you're going to start using PyTorch. PyTorch can run on your machine's CPU, but we need the servers to enable GPU usage. Think of PyTorch as a fancy version of numpy (in fact, they're syntatically almost identical) that can also (1) run on GPUs, and (2) automatically compute gradients for you (more on this later).

If your personal machine also has GPU/CUDA support, we still require you to demonstrate that you can run code on the servers over SSH. This is a useful skill.

Outputs: No python code needs to be submitted for this one, but you'll be writing and running short snippets of code for it. You'll be placing answers in the file `torch_snippets_and_errors.txt`. The first few lines are already filled to get you started.

Instructions: For each of the following bugs/errors/behaviors, write a snippet of code that does it, then write a line in `torch_snippets_and_errors.txt` saying what the error message is (if there is an error message) or what the behavior is (if there is no error message). There are essentially three possible outcomes of each snippet:

- **It runs without error and produces the expected output.** In this case, you should write a line in `torch_snippets_and_errors.txt` saying what the correct and expected behavior was.
- **It runs without error but produces unexpected output.** In this case, you should write a line in `torch_snippets_and_errors.txt` saying what you expected and what the unexpected outcome was.
- **It produces an error message.** In this case, you should write a line in `torch_snippets_and_errors.txt` with the last line of the error message (following the examples in `torch_snippets_and_errors.txt`).

Snippets to write:

1. Add two PyTorch tensors of shape (3, 4) to a PyTorch tensor of shape (4, 3). For example, you could do `torch.randn(3, 4) + torch.randn(4, 3)`
2. Add a PyTorch tensor of shape (3, 4) to a PyTorch tensor of shape (4,). For example, you could do `torch.randn(3, 4) + torch.randn(4)`
3. Add a PyTorch tensor of shape (3, 4) to a PyTorch tensor of shape (4, 1). For example, you could do `torch.randn(3, 4) + torch.randn(4, 1)`
4. Add a pytorch tensor of dtype `torch.float32` to a pytorch tensor of dtype `torch.float64`. For example, you could do `torch.tensor([0, 1, 2], dtype=torch.float32) + torch.tensor([3, 4, 5], dtype=torch.float64)`

5. Create a tensor of all zeros of dtype `torch.uint8`, then try to assign it a value of `-3.14`. For example, you could do `x = torch.zeros(3, dtype=torch.uint8); x[0] = -3.14`
6. Add two pytorch tensors of the same shape and dtype but on different devices (e.g. CPU and GPU). For example, you could do `torch.randn(3, 4) + torch.randn(3, 4).cuda()`
7. Generate some random data to make a scatter plot, then try to plot it with the matplotlib function `plt.scatter`. For example, you could do `plt.scatter(torch.randn(100), torch.randn(100)); plt.show()`.

   No need to include the plot anywhere. Just write down whether it worked or not (do you expect matplotlib, which was designed to plot numpy arrays, to know how to work with pytorch tensors?)
8. Repeat number 7 but where the tensors are on the GPU. For example, as a one-liner you could do `plt.scatter(torch.randn(100).cuda(), torch.randn(100).cuda()); plt.show()`
9. Repeat number 7 but where the tensors have the `requires_grad` attribute set to `True`. For example, you could do

   ```
   x = torch.randn(100, requires_grad=True)
   y = torch.randn(100, requires_grad=True)
   plt.scatter(x, y)
   plt.show()
   ```
10. Repeat number 9 but use `plt.scatter(x.detach(), y.detach())`
11. Create a PyTorch tensor and increment it in-place using the `+=` operator. For example, you could do `x = torch.zeros(3); x += 1`
12. Repeat number 11, but with a tensor that has the `requires_grad` attribute set to `True`. For example, you could do `x = torch.zeros(3, requires_grad=True); x += 1`
13. Calculate the derivative dy/dx where $y = x^2$ at the point $x = 100$. For example, you could do `x = torch.tensor([100.], requires_grad=True); y = x**2; print(torch.autograd.grad(y, x))`. The expected behavior here is that you get a tensor whose value is the derivative of y with respect to x at the point `x=100`. From calculus, we know $d/dx(x^2) = 2x$, so the expected output is `[200.]`.
14. Repeat number 13 but where `x` is a tensor with more than one element, for example `x = torch.tensor([100., 200., 300.], requires_grad=True)`
15. Repeat number 13 but where `y = torch.log(torch.exp(x))`. From calculus, we know that $d/dx(\log(e^x)) = 1$. Is that what you get?

16. Try to create a roughly 20GB array on CUDA. A single float32 has 4 bytes, so you'll need to create an array with 5 billion elements, such as a tensor of shape (5000, 1000, 1000). For example, you could do `torch.zeros(5000, 1000, 1000, dtype=torch.float32, device='cuda')`.

    (Note: since GPUs are shared, this one has the potential to break *other* people's code, and you should test it out only enough to understand what's going on, then stop.)

17. PyTorch shapes images differently than OpenCV. In OpenCV we saw arrays of dtype np.uint8 and shape `(h,w,c)` where `c=1` for grayscale images and `c=3` for BGR or other color spaces. In PyTorch, we'll more often be dealing with tensors of dtype `torch.float32` and values that range in `[0.0, 1.0]` (just like the `uint8_to_float` conversion we've been doing in numpy up until now). In PyTorch, the shape will be `(c,h,w)` instead of `(h,w,c)`. Let's look at an example… I downloaded a famous small color image dataset called CIFAR-10 to the `/local/sandbox/csci631/datasets/cifar10/` directory on all three course servers. You should have read access but not write access to everything in `/local/sandbox/csci631/`. Run the following (all of this **should** work and if it doesn't, we might need to fix some server configuration things)

    ```
    import torchvision
    import matplotlib.pyplot as plt

    transform = torchvision.transforms.ToTensor()
    dataset = torchvision.datasets.CIFAR10(
        "/local/sandbox/csci631/datasets/cifar10",
        train=True,
        transform=transform
    )
    img, label = dataset[0]
    print(img.shape, img.dtype, img.min(), img.max(), label)
    ```

    You don't need to add anything to `torch_snippets_and_errors.txt` for the above, but you should run it and make sure it works. Note that for CIFAR-10, the images are 32x32 pixels. Now, let's try to display the image with matplotlib. Run the following:

    ```
    plt.imshow(img)
    plt.show()
    ```

    This should give an error. Put the error as entry 17 in `torch_snippets_and_errors.txt`.

18. Now, let's try to fix it. Run the following:

    ```
    plt.imshow(img.permute(1, 2, 0))
    plt.show()
    ```

Write down what you see in `torch_snippets_and_errors.txt` as entry 18 (something like "I see a picture of…").

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.