SIMON D. LEVY

Computer Science Department, Washington and Lee University

CSCI 315 Assignment #4

Due 11:59PM Monday 6 March, via Github

Goal

The goal of this assignment is to become familiar with PyTorch, one of the two most popular software packages for deep learning. We will also step up our game, moving from Mozer's simplified 14×14-pixel MNIST digit set to the full-sized 28×28 set. Scaling up to real-world tools and datasets may also mean that you will have to switch from working on your laptop to working on a workstation. The computers in the Advanced Lab (Parmly 413) have been set up with PyTorch to help with this.

Getting Started

First, follow the instructions I showed you about using pytorch.org for installing PyTorch on your laptop if you have one.

Most of your coding for this assignment will be copy/paste/modify — in my experience, the next best thing to writing it yourself from scratch — and often the only practical option!

The author's Github repository for the book contains code in Jupyter notebook (ipynb) form, corresponding to the Chapter 5 section **Building the MNIST Classifier in PyTorch (**starting on p. 148). Find that code in the repository and copy/paste it into a file **mlp.py** (one section a time is safest), all the way through the `train()` and `test()` calls at the bottom. With no modification to the code, I was able to get a figure and test results very similar to what's shown in the notebook: a nice smooth descending error and a testing accuracy close to 91%. I also saw that the `train()` function saved (pickled) the trained network to file **mnist.pt**, which is nice. I did get some weird *Unable to init server ... Connection refused* messages, as well as something about "Gdk" on the machines in

P413, but that didn't affect my ability to run the code. In my experience, these kinds of minor annoyances are pretty common with Deep Learning packages and other large open-source software projects that are evolving so rapidly.

Pickling

As in our previous assignment, we want to get used to the habit of separating our training and testing code into two separate programs, enabling us to run a trained network on new data. So, copy/paste your **mlp.py** script into **mlp_train.py** and **mlp_test.py**. Then edit your three files so that **mlp.py** has just the network class code, **mlp_train.py** runs the training, and **mlp_test.py** runs the test. Running **mlp_train.py** will produce the **mnist.pt** (pickled sate dictionary) file as before. I also found it helpful for **train()** to print a little message telling the user *Saving network in mnist.plt*. To figure out how to load this file into **mlp_test.py**, I found this documentation useful. Good coding practice also dictates that you should remove unnecessary imports (e.g., MNIST data imports from **mlp.py** and **mlp_test.py**). At this point you should probably also comment-out the plt plotting code to save yourself the trouble of the plot window popping up.

You'll probably notice at this point that in both scripts you need to specify this size of the network (`in_dim, feature_dim, out_dim`). This isn't terrible, but it will slow you down when you want to experiment with different network shapes in the next section. Since this is the MNIST data set, we know that the number of inputs is always 784 (28×28 pixels) and the number outputs (digit classifications) is 10. Plus, as in our previous assignment, there's information in the trained network (params, weights) that enables you to determine the other size without having to store or specify it explicitly. So add a little code to **mlp_test.py** to extract and use this information.

Hyper-parameters

91% accuracy seems like an awesome result on a serious data set like MNIST. Looking at the code, you'll see what look like pretty arbitrary decisions about the standard hyperparameters (training iterations, hidden units, eta). In this part we'll try a little "Goldilocks and the Three Bears" experimentation to see whether the values we're using are "just right" (i.e., a good tradeoff between training time and testing generalization).

First let's look at training iterations. Another name for these is ***epochs***, which you'll see in your **mlp_train.py** script as 40. We could mess around all day with different values, but

the Goldilocks approach suggests simply trying something significantly lower and significantly higher. So try 20 and 60 instead and note whether you get obviously better or worse testing results. You'll put these values into a little plot or table in your writeup, so for the time being you can just write them down or save them in a spreadsheet.

Next let's play with the size of the hidden layer. As usual, people use different terminology for the same concept. Based on the work you did in the pickling part, it should be pretty obvious what variable name corresponds to the hidden-layer size. Once you've got that, modify the code to stop training not after a fixed number of epochs, but instead after a particular loss value. Then see if you can test the claim in our lecture notes about more hidden units providing faster training but worse generalization (testing). (FWIW, I wasn't able to get anything consistent here.)

Finally, look for the variable corresponding to the learning rate that we called η. Again, try something significantly bigger and smaller to see if you get any noticeable difference in training duration and testing accuracy. And again, your results are valuable (perhaps more so!) even if they don't match expectations: maybe you can get the network to train faster *and* get better test results?

Confusion Matrix

As before, a confusion matrix is a lot more informative than a simple accuracy rate. What you'll do on this part is grab your confusion matrix code from the previous assignment and copy/paste it into your **mlp_test.py** script, then figure out how to reconcile it with the existing code in **mlp_test.py**.

Looking at that code, I saw some obvious candidates, `pred` and `targets`. At first the shapes and contents those tensors didn't make sense, but looking at them in more detail, I realized how they were formatted, and how to use them to populate my confusion matrix. Once I figured that out, I was able to add just two lines to do this!

Direct logistic regression (no Hidden Layer)

As I mentioned in class, the previous time I gave this assignment (in TensorFlow) we found that the logistic regression being use on the output layer of the MNIST network was so powerful that we got good results without needing a hidden layer; i.e., with a single-layer perceptron. To see how well we can do with an SLP, copy your three scripts to three new

scripts **slp.py**, **slp_train.py**, and **slp_test.py**. Then modify the code to use only one layer of weights. Since there is no more hidden layer, you'll have to experiment with the other hyper-params (number of epochs, learning rate) to try and match the performance of your MLP. *Hint*: A common trick to get quickly to a good solution is to keep doubling the size of the value (e.g., epochs) until you get something that works.

CPU vs. GPU showdown

As we discussed, a big (supposed) advantage of PyTorch is the ease with which it lets you run your training on the GPU via the CUDA software libraries. Since you're unlikely to have a CUDA-capable GPU on your laptop (and it's nontrivial to install CUDA anyway in my experience), I suggest doing this final part of the assignment on one of the machines in the Advanced Lab (Parmly 413). The only trick is that you have to specify the location of Python you're using (**/usr/bin/python3**), since only one of the versions of Python in the Lab has torch installed. For example, at the command prompt:

```
/usr/bin/python3 mlp_train.py
```

Unable to locate anything helpful in our textbooks about PyTorch + CUDA, I found various tutorials online, but none got me to a complete solution. After much trial and error I came up with the following recipe. The point of doing it this way is that you'll automatically run on CUDA if it's available, and on the ordinary CPU otherwise:

First, at the top of your training script, set a variable to name what device you want to run on, based on whether CUDA is available:

```
dev = "cuda:0" if torch.cuda.is_available() else "cpu"
print("Running on " + dev)
```

Next, immediately after constructing your classifier, convert it to use the device:

```
classifier = classifier.to(dev)
```

Finally, in the training loop, do the same `.to(dev)` trick on your `data` and target `tensors`.

Once you've got this working, it's time for a head-to-head comparison between GPU and CPU. First, we'll time it on the GPU. From the command line:

```
time python3 mlp_train.py
```

Do this a few times, noting down the **real** component of the timing result, in case it varies between trials. Next, override the automatic device choice, forcing the program to run on the CPU:

```
dev = "cpu" # "cuda:0" if torch.cuda.is_available() else "cpu"
```

To my disappointment, I found that the GPU ran at the same speed (time) or *slower* than the CPU! If you've taken a course on parallel computing (or google around a bit on this complaint), you'll know that the typical explanation is that you're running a relatively small model and transferring the data to the GPU more frequently than you might need to. At this point I was happy with my ability to run on the GPU for future work and didn't try harder to get the GPU to win as expected. So an excellent extra-credit opportunity would be to modify your model or training code enough to get a noticeable speedup on the GPU.

What to submit to Github

Do a little PDF write-up (a single page should be sufficient; two at most) with a brief description of your results in each part, including the confusion matrix. That's the main thing I'll look at, but please also submit your six Python scripts as well, in case I want to check your results.