11-695: AI Engineering
Assignment 3: Distributed Optimization

Spring 2020

Abstract

In this assignment, you will implement a distributed optimization algorithm in Pytorch [Paszke et al., 2017]. Although the distributed environment required is a single machine with many parallel processes, the assignment will help you understand better how to implement a distributed program in practice and evaluate related overheads. The starter code is provided for you, with all the data loading mechanisms and training driver implemented. Your job is to understand the starter code and implement the correct training routine for the SVRG algorithm [Johnson and Zhang, 2013] in distributed mode, and then compare it with the distributed SGD.

1 Code and Dataset

Install Pytorch (of version at least 1.0 and recommended 1.4) If you have not already, please navigate to Pytorch's site and follow their instructions to install the framework. The instructions are at

https://pytorch.org

This assignment only requires you to use CPU, but you can also install the GPU version with no harm. As in the previous assignments, you can quickly use an AWS AMI for quick and convenient setup.

Datasets. There are 2 datasets for this assignment. One is MNIST, which will be automatically downloaded and processed for you. The other is the **abalone** dataset from the UCI repository[1]. The driver for this dataset is provided to you, so you can focus completely on the SVRG algorithm.

Starter code. The starter code for your project is available at

http://aieng.cs.cmu.edu/assignment.html

After downloading and unzipping the code and the data, you can navigate to your code in the folder `src`. Same as the last assignment, you will see the `TODOs` and hints which suggest what you should or should not do. You will be mainly working the file `d_svrg.py`.

Compared to the previous assignments, this one is less intensive in terms of coding, but requires you to thoroughly understand the algorithm and how to do the communication in a distributed setting first. After that, only about less than 100 lines of code (excluding bells-and-whistles ones) are needed to complete this assignment.

[1] http://archive.ics.uci.edu/ml/index.php

SVRG code structure. First, you need to understand the structure of the algorithm illustrated in Figure 1. Our implementation uses 2 same models (having independent weights, of course) and 2 separate optimizers respectively.

About distributed manifestation of this algorithm in `d_svrg.py`, we select the node with rank 0 to be the parameter server, and also a client node itself. Likewise, rank-0 node will be responsible for gathering all gradients information, as well as other statistics such as loss values and accuracies. You might find this one different from what we studied in class but there is no contradiction at all. Nonetheless, if you find this setting troublesome, we would encourage you to review the provided distributed SGD implementation carefully before moving on.

Next, if you take a closer look at the SVRG algorithm, the main communication happens when you switch from the outer loop (indexed by s) to the inner loop (indexed by t) and vice versa. For each time of such switch, the communication has to happen between 2 models. The communication is, however, more complicated when you might need point-to-point exchange between the randomly chosen node and the master rank-0 node. If by any reason it is difficult to debug or implement, you can simply use a collective operation, *e.g.* `all_reduce` instead of `reduce`. In practice, however, it is not always the case that client nodes can contact with each other due to privacy restriction, which we assume not to have here.

Procedure SVRG

Parameters update frequency m and learning rate η
Initialize $\tilde{w}_0$
Iterate: for $s = 1, 2, \ldots$
  $\tilde{w} = \tilde{w}_{s-1}$
  $\tilde{\mu} = \frac{1}{n} \sum_{i=1}^{n} \nabla \psi_i(\tilde{w})$
  $w_0 = \tilde{w}$
  Iterate: for $t = 1, 2, \ldots, m$
    Randomly pick $i_t \in \{1, \ldots, n\}$ and update weight
      $w_t = w_{t-1} - \eta(\nabla \psi_{i_t}(w_{t-1}) - \nabla \psi_{i_t}(\tilde{w}) + \tilde{\mu})$
  end
  option I: set $\tilde{w}_s = w_m$
  option II: set $\tilde{w}_s = w_t$ for randomly chosen $t \in \{0, \ldots, m-1\}$
end

Figure 1: The original SVRG algorithm [Johnson and Zhang, 2013]. Legend: s denotes the incremental number of epochs, m is the total number of inner loops per epoch, global weights at epoch s are denoted as $\tilde{w}_s$ while local (to the node) weights are w_t for inner loop t.

2 Requirements

The only entry point of the program is in the file `train.py` that will call relevant modules. It is your responsibility to read the code and figure out all the relevant points. We also provided some `TODOs` and hints, however, to make your tasks less challenging.

First, you need to understand the SVRG algorithm, which might be confusing at first but in fact not so complicated. See the legend carefully in Figure 1 for details, and pay a careful attention to the global (outer-loop) and local (inner-loop) variables.

Your jobs are as follows:

1. **Finish distributed SVRG on `d_svrg.py`**: we have provided the guidelines along the code with `TODOs` items so that you can follow and finish conveniently.

2. **Consistency of the initial loss** It is required for you to have the training loss curve plots, in which the losses of SGD and SVRG both start at the same point. The reason is that in optimization, people care about how low the algorithm can bring down this loss, and so make them start at the same point will make fair comparisons. As a result, for this task, you have to implement a separate operation where you initialize the weights for each of your models, and save it into disks. To make it convenient for you, we have already provided the code to load those weights (ee `init_model()` in `utils.py` for more detail). We expect you to store your weights in the folder `weights`, but you can change however you want (and make sure to change the respective parts in the code to reflect this change). In short, you can implement this in the provided file `create_init_weights_for_all.py` or any where you want.

3. **Evaluate your distributed SVRG**. In evaluation, you need to test it with different settings, such as number of nodes, number of batches, ... It is required that you have to compare it with the distributed SGD algorithm for each setting you consider. To make it convenient for you, we already set up the output folder for you at the same level as `src` and `data`, with the form `output_xx` in the code.

Implementation Note: You don't have to follow our implementation structure; you can do it your way, even if you want to wipe out everything and re-implement it. The only requirement if you decide to re-implement, however, is that it has to be distributed with multiple nodes and has to include communications of either point-to-point or collective types to exchange gradients and loss/accuracies.

Hint: Many functions in `utils.py` should be handy.

3 Assessment

This assignment tests the concepts rather than implementation, so we do not require difficult baselines for you. Passing those simple baselines will hence earn you a complete 100 points. In detail, you are only required to pass the following **baselines** for your **best** model (but remember to record the hyper-parameters for this best model in your report):

1. MNIST: 91% test accuracy for SVRG (yours must be higher).

2. Abalone: 30.0 test loss for SVRG (yours must be lower).

You can run as many epochs as you want to pass those baselines. If by any reason you cannot pass any baseline, we will deduct points based on the percentage of the gap from that specific baseline.

Extra credits: We will provide (up to maximum 10) bonus points in all possible categories combined if you can do extra work as–but not limited to–follows:

- Re-implement SVRG completely your way, which is different from the provided one (of course you also have to pass the baselines)

- Explore other algorithms besides SGD and SVRG. The other similar versions of them are not counted, *e.g.* Adam or Adagrad for SGD, to name a few. Check with the course staff first if you are not sure about this.

- Migrate both SGD and SVRG to GPU, or still with CPU but evaluate it with multiple physical nodes (where you have to re-make the initialization of the cluster environment properly).

4 Reports

As usual, you are required to submit your log files of training, your statistics on any method on any dataset that you work on. The limit is maximum 2 pages. We will give more credits if you have a high-quality, well-written one with exhaustive explorations with different settings, accompanied by your reasoning and analyses.

5 Gradings

The grading breakdowns are as follows with totally 110 points:

1. Pass the baseline for MNIST for SVRG: **40 points**
2. Pass the baseline for Abalone for SVRG: **40 points**
3. Report: **20 points**
4. Extra credits: up to **10 points** as desribed above in section 3 and 4. As usual, we will not limit any creative ideas from you; feel free to apply any idea that you see fit. If the idea is interesting, negative results would be interesting too (although you are required to provide your reasoning about such results).

6 Submission

All submissions must be made to Canvas individually, including:

- Your folder of code excluding data files or folder.
- Your real log of running each of your models, and also the output directories.
- Your report.

Note: Your code must be runnable directly with Python3 and Pytorch (of version at least 1.0). Any extra packages might be super useful but is prohibited, and should not be needed. But if you insist to use any of new packages on the internet, check with the instructors first. Furthermore, if your code has a special instruction to run, please detail it. We will give you zero for each part where your code is not runnable. The TAs will be running each of your code so please be considerate to them as well.

7 Academic Integrity.

As usual, you are encouraged to discuss with your friends and the instructors. Anything they tell you, you can use. However, looking at other people's codes should not happen at all cost.

References

Rie Johnson and Tong Zhang. Accelerating stochastic gradient descent using predictive variance reduction. In *Advances in neural information processing systems*, pages 315–323, 2013.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

Revision

1. Apr 05: First release.