CS744 Assignment 2

Due: Oct 5, 2020, 10pm Central Time

Overview

This assignment is designed to build on your in-class understanding of how distributed training of machine learning algorithms is performed. You will get on hands-on experience in using PyTorch (https://pytorch.org/) in conjunction with MPI like communication frameworks like Gloo (https://github.com/facebookincubator/gloo) and OpenMPI (https://www.open-mpi.org/). You will understand the tradeoffs of different approaches of performing distributed training.

Learning Outcomes

After completing this programming assignment, you should be able to:

- Deploy and Configure Distributed ML Training frameworks.
- Write distributed training applications with PyTorch
- Understand the trade-off between different methods of performing distributed training
- Describe how Machine Learning frameworks(PyTorch) interact with collective communication frameworks (e.g., OpenMPI, Gloo)

Environment Setup

Like previous assignment you will complete this assignment on Cloudlab. See Assignment 0 (https://pages.cs.wisc.edu/~shivaram/cs744-fa20/assignment-zero.html) to learn how to use CloudLab. You should only create one experiment per group and work together. An experiment lasts 16 hours, which is very quick. So, set a time frame that all your group members can sit together and focus on the project, or make sure to extend the experiment when it is necessary.

In this assignment, we provide you a CloudLab profile called “cs744-fa20-assignment2” under “UWMadison744-F20” project for you to start your experiment. The profile is a simple 4-node cluster with Ubuntu installed on each machine.

While launching the experiment make sure to choose the right group name. You get full control of the machines once the experiment is created, so feel free to download any missing packages you need in the assignment.

As the first step, you should run the following command on every VM:

1. `sudo apt-get update --fix-missing`
2. `sudo apt install python-pip`
3. Install numpy using: `pip install intel-numpy`
4. Install Pytorch for CPU using

```
pip install torch==1.4.0+cpu torchvision==0.5.0+cpu -f https://download.pytorch.org/whl/torch
_stable.html
```

5. Install future using: `pip install future`

6. Gloo comes built in with PyTorch, therefore requiring no installation.

Note 1: For installation using pip may be asked for root permission. Ideally when setting things up you should use virtual environment but since we have dedicated instances for the project you are free to use `sudo pip` . Note 2: If you are familiar feel free to use Anaconda for setup.

For this assignment using the home directory is enough, you will not need to use any extra disk.

Part 1: Training VGG-11 on Cifar10

We have provided a base script for you to start with, which provides a model setup (model.py) and training setup(main.py) to train on VGG-11 network with Cifar10 dataset.

You can find the base training scripts to modify here (https://pages.cs.wisc.edu/~shivaram/cs744-fa20/assignment2-template.tar.gz)

Task1: Fill in the standard training loop of forward pass, backward pass, loss computation and optimizer step in main.py. Make sure to print the loss value after every 20 iterations. Run training for a total of 1 epoch (i.e., until every example has been seen once) with batch size 256. (NOTE: If CIFAR-10 has 50,000 examples then you will finish 1 epoch after 196 iterations).

There are several examples for training that describe these four steps. Some good resources include the PyTorch examples repository (https://github.com/pytorch/examples) and the Pytorch tutorials (https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html). This script is also a starting point for later parts of the assignment. Familiarize yourself with the script and run training for 1 epoch on a single machine.

Part 2 - Distributed Data Parallel Training

Next you will modify the script used in Part 1, to enable distributed data parallel training. There are primarily two ways distributed training is performed i) Data Parallel, ii) Model Parallel. In case of Data parallel each of the participating workers train the same network, but on different data points from the dataset. After each iteration (forward and backward pass) the workers average their local gradients to come up with a single update. In model parallel training the model is partitioned among number of workers. Each worker performs training on part of the model and sends it output to the worker which has the next partition during forward pass and vice-versa in backward pass. Model parallel is usually used when the size of the network is very large and doesn't fit on a single worker. In this assignment we solely focus on Data Parallel Training. For data parallel training you will need to partition the data among other nodes. Look at the FAQ's to find details on how to partition the data.

Part 2a: Sync gradient with gather and scatter call using Gloo backend

PyTorch comes with the Gloo backend built-in. We will use this to implement gradient aggregation using the gather and scatter calls.

(i) Set Pytorch up in distributed mode using the distributed module of PyTorch. For details look here (https://pytorch.org/docs/stable/distributed.html). Initialize the distributed environment using init_process_group (https://pytorch.org/docs/stable/distributed.html).

(ii) Next, to perform gradient aggregation you will need to read the gradients after backward pass for each layer. Pytorch performs gradient computation using auto grad when you call `.backward` on a computation graph. The gradient is stored in `.grad` attribute of the parameters. The parameters can be accessed using `model.parameters()` .

(iii) Finally, to perform the aggregation you will you use gather and scatter communication collectives. Specifically Rank 0 in the group will gather the gradients from all the participating workers and perform elementwise mean and then scatter the mean vector. The workers update the grad variable with the received vector and then continue training.

Task2a: Implement gradient sync using gather and scatter. Verify that you are using the same total batch size, where total batch size = batch size on one machine * num_of machines. With the same total batch size you should get similar loss values as in the single node training case. Remember you trained with total batch size of 256 in Task 1.

Part 2b:Sync gradient with allreduce using Gloo backend

Ring Reduce is an extremely scalable technique for performing gradient synchronization. Read here (https://tech.preferred.jp/en/blog/technologies-behind-distributed-deep-learning-allreduce/) about how ring reduce has been applied to distributed machine learning. Instead of using the scatter and gather collectives separately, you will next use the built in `allreduce` collective to sync gradients among different nodes. Again read the gradients after backward pass layer by layer and perform allreduce on the gradient of each layer. Note the PyTorch allreduce call doesn't have an 'average' mode. You can use the 'sum' operation and then get the average on each node by dividing with number of workers participating. After averaging, update the gradients of the model as in the previous part.

Task2b: Implement gradient sync using allreduce collective in Gloo. In this case if you have set the same random seed (see FAQ), you should see the same loss value as in Task2a, while using the same total batch size of 256.

Part 3: Distributed Data Parallel Training using Built in Module

Now instead of writing your own gradient synchronization, use the distributed functionality provided by PyTorch. Register your model with distributed data parallel (https://pytorch.org/docs/master/generated/torch.nn.parallel.DistributedDataParallel.html#torch.nn.parallel.DistributedDataParallel) and perform distributed training. Unlike in Part 1 and Part 2 you will not need to read the gradients for each layer as DistributedDataParallel performs these steps automatically. For more details read here (https://pytorch.org/tutorials/intermediate/ddp_tutorial.html).

Deliverables

You should submit a tar.gz file to Canvas, which consists of a brief report (filename: groupx.pdf) and the code of each task. In the report include the following contents:

- Discard the timings of first iteration and report *avg time per iteration* for the remaining 9 iterations for each task (1, 2a, 2b, 3).
- In the context of the PyTorch distributed paper (https://arxiv.org/pdf/2006.15704.pdf), reason about the difference or lack of difference among different setups.
- Comment on the scalability of distributed machine learning based on your results.
- Add contributions of each member of the group.
- Code for each part should be in different folders.
- All your codes should be runnable using the following command-

```
python main.py --master-ip $ip_address$ --num-nodes 4 --rank $rank$
```

Look at python argparse to provide this functionality. where IP-address, num-nodes and rank are command line parameters supplied by the grader at run time.

- Provide any other implementation details

FAQ's

Testing Programs We suggest you to write small programs to test the functionality of communication collectives at first. For example, create a tensor and send it to another nodes. Next try to perform all reduce on it. This will help you get comfortable with communication collectives.

Using Experiment Network Same as previous assignments please use the experimental network for this assignment. This means you need to make sure the machines listen on 10.10.1.* interfaces.

Example of distributed setup Look at PyTorch Imagenet (https://github.com/pytorch/examples/blob/master/imagenet/main.py) and distributed (https://github.com/pytorch/examples/tree/master/distributed/ddp) examples.

Setting up distributed init In this setup we will use init-method as "tcp://10.10.1.1:6585". Instead of a shared file system we want to use TCP to communicate. In this example I am using 10.10.1.1 is the IP-address for rank 0. Port has to be a non-privileged port, i.e. greater than 1023.

Running the program If you are using an MPI setup you can use `mpi run` command to launch multiple workers. Since we have a small number of nodes we will manually start our program on the 4 nodes and keep all arguments same except the rank. That is, ssh into each node and run `python main.py --rank <fill-in-appropriate-rank>`

Rank and World Size Rank is indexed from 0, World size is number of worker nodes. So in our case Rank will be from 0 to 3 and World Size will be 4

Setting up random seed You will need to setup the random seed before you initialize the model, since the model is initialized randomly. In Data Parallel setting you need to make sure you start from the same model on all the workers. Look at `torch.manual_seed()` and `numpy.random.seed()` .

Data Partitioning In case of data parallel training, the workers train on non-overlapping data partitions. You will use the distributed sampler to distribute the data among workers. For more details look at torch.utils.data.distributed_sampler (https://pytorch.org/docs/stable/data.html#torch.utils.data.distributed.DistributedSampler)