ASSIGNMENT 3

MPI Collective Communication and Asynchronous Communication

Due Date: 22/01/2019 23:59

Questions:

- Questions regarding this assignment should only be sent to Anny.
- Problems with connecting to servers should only be sent to Or.
- Postponements can only be authorized by the TA in charge Ido.
Brief Background

In this exercise, we’ll dive deeper into the *Fit* (train) function of the Neural Network. In every cycle (epoch) of fit, we first divide the training data into batches with the infamous function `create_batches`. We iterate over and over on the batches and execute forward and backward propagations on the data. During this process, we can improve our parameters (weights and biases) and therefore renovate our Neural Network.

(A diagram which describes how fit function works)

The “Evaluate loss” stage (which contains the forward and backward propagations) is considered as a time-consuming stage. That is because we are multiplying large matrices.

In this exercise, you will divide the batches between different workers. This methodology can improve performance.

You’ll implement the fit function in two complement manners:

- Synchronous Fit
- Asynchronous Fit

And later, you’ll compare runtime & performance between the implementations and deduce your conclusions.
Part 1 - Synchronous

In this part, every worker will work on a smaller size batch and calculates gradients for weights and biases. Consequently, you’ll sum the gradients and recalculate new weights and biases according to the sum.

Meaning, if worker $i$ calculated the gradients $my\_gradients_i$ then the sum of gradients between the workers will hold that $gradients\_sum[j] = \Sigma_{i\in\text{workers}} my\_gradients_i[j]$.

1a. Using MPI

In synch_networy.py, complete the implementation of:

```python
def fit(self, training_data, validation_data=None):
```

Every process does forward and backward propagations to calculate gradients $my\_nabla\_w$ and $my\_nabla\_b$. Each is a list (of size equal to the number of layers) of numpy arrays.

You need to use MPI collective communication so that every process will hold the sums of the gradients as explained above in the lists $nabla\_w$ and $nabla\_b$.

To be clear, the shape of $nabla\_w$ and $nabla\_b$ needs to be the same as $my\_nabla\_w$ and $my\_nabla\_b$ respectively.

1b. Implementing your own Allreduce

One of the collective communications routines you’ve seen in class is MPI Allreduce. It takes an array of input elements on each process and returns an array of output elements to all the processes.

![MPI_Allreduce Diagram]

Use only p2p communication in this part. You should use Asynchronous communication when able.
In my naïve allreduce implement

```python
def allreduce(send, recv, comm):
    # Implement a naïve version of allreduce, in which every process sends all the other
    # process it’s array and reduces all arrays.
    # You may assume that send and recv are numpy arrays of the same shape.
    # The reduce function is always sum.
```

In my ring allreduce implement

```python
def ringallreduce(send, recv, comm):
    # Implement allreduce again, this time use the Ring All Reduce algorithm you’ve seen in
    # the lecture.
    # You may assume that send and recv are numpy arrays of the same shape.
    # The reduce function is always sum.
```
Part 2 - Asynchronous

In this part, you'll implement an asynchronous approach. There will be two components: worker(s) and master(s).

Note, you may use only asynchronous MPI p2p communication functions in this part.

In asynch_network complete the following functions:

```python
def do_worker(self, training_data):
    Every worker acts according to the next algorithm:
    1. Divide number of batches between the workers (In total, all the workers should execute number_of_batches in every epoch).
    2. For every epoch, create mini batches.
       a. For every batch, execute forward_prop and backward_prop to calculate gradients.
          i. Send the gradients (weights and biases) of each layer to the master in charge of the layer.
          ii. Receive from each master the new weights and biases of the layers the master is in charge of.

def do_master(self, validation_data):
    Every master acts according to the next algorithm:
    1. Divide the layers between the masters (each master will receive the gradients that corresponds to it’s layers and calculate the weights and biases of those layers)
    2. For every epoch,
       a. For every batch,
          i. Wait for any worker to send gradients (of weights and biases of the layers the master is in charge of).
          ii. Calculate the new weights and biases for the layers in charge of using the gradients received.
          iii. Send the worker the new weights and biases the master calculated.
    3. Send the weights and biases the master calculated to process 0
```
Report

1. Run the synchronous neural network implementation with 4, 8 and 16 cores.
2. Compare the run time between different number of cores and the original neural network implementation. Include screenshots and a short explanation about the results.
3. According to what approach the current synchronous algorithm gain speedup (Amdahl or Gustafson)? What changes need to be made in order to gain speedup in the other approach.
4. Run the asynchronous neural network implementation with 2 masters and 4 and 8 cores and then 4 masters with 8 and 16 cores.
5. Compare the run time between the different runs and the accuracy of the training between them. Include screenshots and a short explanation about the results.
6. Why are we splitting the parameter server (master) across multiple machines in the asynchronous approach?
7. Show graphs of final accuracy as a function of the number of workers for both synchronous and asynchronous (with 2 and 4 masters) implementations.
8. Explain why the asynchronous implementation diverge when training on a large number of workers.
9. Compare the synchronous approach with the asynchronous approach. What are the benefits of each approach? What are the disadvantages?
10. Run allreduce_test.py with 2, 4 and 8 cores. Compare the results of the naïve all reduce implementation and the ring implementation. Include screenshots.
11. What is the complexity of naïve all reduce (how much data every process sends and how much data in total is sent)?
12. What is the complexity of a ring all reduce?
Notes and Tips

1. You can add variables and prints as you need, but your code must be clear and organized.
2. Don’t remove anything already in the code, adhere to instruction comments.
4. In the tutorial, you were told the mpi4py processes are initialized when MPI is imported. In this exercise we have disabled this automatic feature (using mpi4py.rc) and therefore we are doing it manually using MPI.Init() and MPI.Finialize().
5. When you use mpi4py library, always use the function which begins with capital letter (If you won’t, it may behave different than expected).
6. Functions of the mpi4py library (as described in 2.) expects numpy arrays.
7. You can change the flag --pty in your run with -o out%t to get output of each process in separate files named outX. It might help with debugging.
8. Use your utils.py from the 1st exercise. You don’t need to submit it again.

Server

Install the necessary packages before you start working (through bash shell, only once):

pip3 install --user mpi4py

Running your script on the server should be done like this:

srun -K -p 236370 -c 4 -n X --pty python3 main.py sync

or

srun -K -p 236370 -c 4 -n X --pty python3 main.py async M

or

srun -K -p 236370 -c 4 -n X --pty python3 allreduce_test.py

where X is the number of cores you want to use, and M is the number of masters in async run.

Notice the space between ‘c’ and ‘4’!

Submission

Submit a hw3.zip with the following files only:

1. sync_network.py with your implementation.
2. async_network.py with your implementation.
3. my_naive_allreduce.py with your implementation.
4. my_ring_allreduce.py with your implementation.
5. A hw3.pdf report of performance analysis of **maximum 3 pages, make it concise.**