ASSIGNMENT 3

MPI Collective Communication and Asynchronous Communication

Due Date: Thursday 19/01/2023 23:59

- Problems with connecting to servers should only be sent to daniel.noor@campus.technion.ac.il

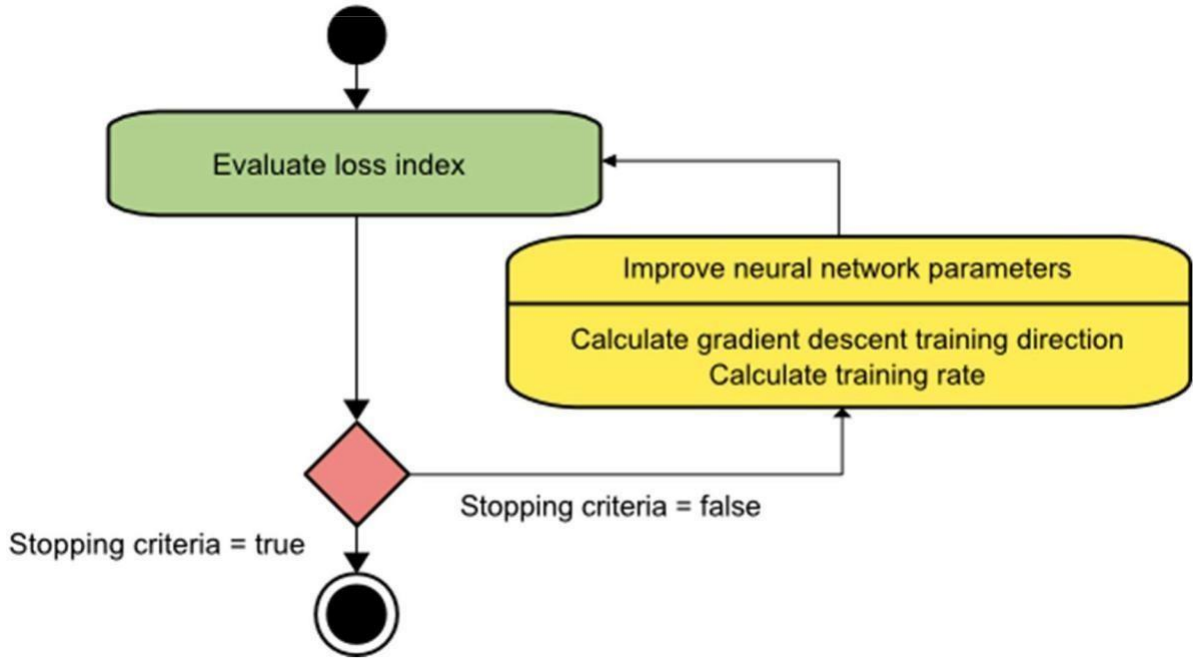
- Postponements can only be authorized by the TA in charge daniel.noor@campus.technion.ac.il

  (remember the late submission policy published on the course website)

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 will 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 $gradient\_sum[j] = \sum_{i \in Workers} my\_gradients_i[j]$.

1a. Using MPI

In sync_network_collective.py, complete the implementation of:

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

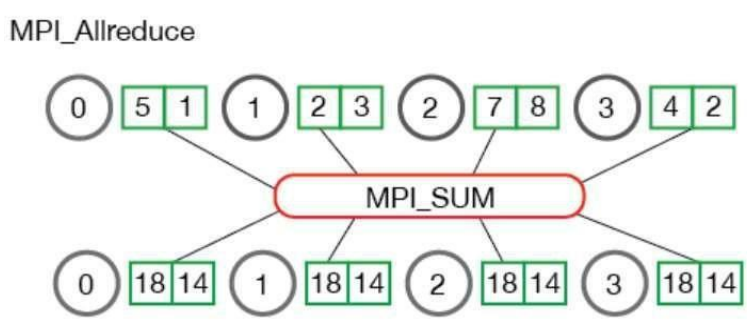
Every process does forward and backward propagations to calculate gradients my_nabla_w and my_nabla_b. Each is a **list** (of size equal to the number of layers) **of numpy arrays**.

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 communication routines you have seen in class is MPI Allreduce.
It takes an array of input elements from each process and returns an array of output elements to all the processes.



Use only p2p communication in this part.
You should use Asynchronous communication when able.

In my_naive_allreduce, implement:

```
def allreduce(send, recv, comm, op):
```

Implement a naïve version of allreduce, in which every process sends all the other process its array and reduces all arrays.

You may assume that send and recv are numpy arrays of the same shape.

op is the reduce operator which means it is associative and commutative.

Example:

```
def op_sum(x, y):
    return x*y
```

In my_ring_allreduce implement

```
def ringallreduce(send, recv, comm, op):
```

Implement allreduce again, this time use the Ring All Reduce algorithm you have seen in the lecture.

You may assume that send and recv are numpy arrays of the same shape.

Finally, **complete the implementation of fit in sync_network.py with your Ring All Reduce**.

Note that you need to use the correct operator for this task.

Part 2 - Asynchronous

In this part, you will 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 (using Wait is still allowed).

In async_network, complete the following functions:

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 oversees.

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 its 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 oversees).
        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. Give a detailed explanation of your implementation in both parts.
2. Run the synchronous neural network implementation (sync_network.py, not sync_network_collective.py) with 4, 8 and 16 cores.
3. Compare the run time between different number of cores and the original neural network implementation. Include screenshots and a short explanation about the results.
4. According to what approach does the current synchronous algorithm gain speedup (Amdahl or Gustafson)? What changes need to be made to gain speedup in the other approach.
5. Run the asynchronous neural network implementation with 2 masters and 4 and 8 cores and then 4 masters with 8 and 16 cores.
6. 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.
7. Why are we splitting the parameter server (master) across multiple machines in the asynchronous approach?
8. Show graphs of final accuracy as a function of the number of workers for both synchronous and asynchronous (with 2 and 4 masters) implementations.
9. Explain why the asynchronous implementation diverges when training on a large number of workers.
10. Compare the synchronous approach with the asynchronous approach. What are the benefits of each approach? What are the disadvantages?
11. 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.
12. What is the complexity of naïve all reduce (how much data every process sends and how much data in total is sent)?
13. 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.
3. Document your code thoroughly.
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.Finiallize().
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.) expect 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.

Server

Assuming you already configured the server in previous HW's, setting up the server is done using this script:

```
cd ~/

python3 -m pip install mpi4py
```

Inside the server, running your script should be done like this:

```
srun -K -c 4 -n X -- mpi=pmi2 -- pty python3 main.py sync or

srun -K -c 4 -n X -- mpi=pmi2 -- pty python3 main.py async M or

srun -K -c 4 -n X -- mpi=pmi2 -- 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.

Submission

Submit a hw3.zip with the following files only:

1. sync_network_collective.py, using MPI collective communication.
2. sync_network.py, using own your implementation of Ring All Reduce.
3. async_network.py with your implementation.
4. my_naive_allreduce.py with your implementation.
5. my_ring_allreduce.py with your implementation.
6. A hw3.pdf report of performance analysis of **maximum 5 pages, make it concise.**