Distributed Systems Project 1

Jun-Ting Hsu (hsu00191@umn.edu), Thomas Knickerbocker (knick073@umn.edu)

PHASE 2 - RUNNING CODE (additional details in the README file):

In Coordinator Terminal: python3 coordinator.py 9090 1

In Node Terminals: python3 compute_node.py <portNo> 0.0
^where portNo is 9091, 9092, 9093, and/or 9094

In Client Terminal: python3 client.py localhost 9090 ./ML/letters 30 10 20 0.0001

PHASE 2 - SYSTEM DESIGN:

Our Architecture:

Basic Description:
The coordinator handles incoming requests from the client. Client requests to train an MLP.

Coordinator sends jobs out to compute nodes in accordance with its scheduling policy, and will attempt to reschedule jobs upon rejection from some node up to <max_retries> number of times.

A lock is used to keep our node load values (which are utilized in scheduling policy 2's node selection) thread-safe as nodes are selected and acquired.

A different lock is used to control the updating of our gradients, which are shared amongst compute nodes at the beginning of each round of computation, during which nodes compute new gradients based on their subset of the data, and those new gradients are averaged by the coordinator, and then used to update the parameters of the coordinator's model.

Logs of each job are maintained by the coordinator. The timing, node selection attempts, epochs, and validation errors of models are logged as they are trained.

The in-order operations of the coordinator and a single compute node are described below.

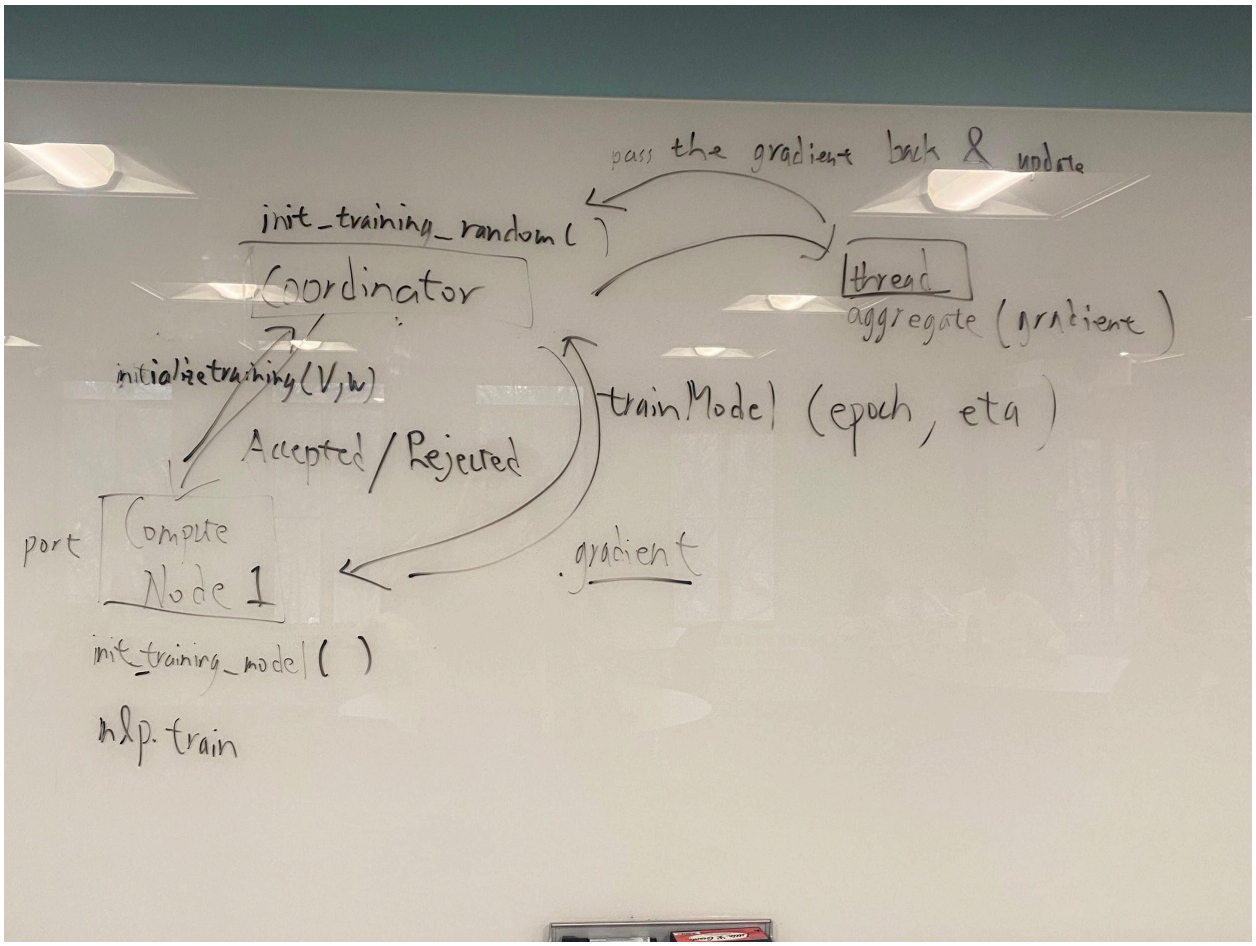
Coordinator:

1. call init_training_random() to initialize the model weight
2. call get_weights()
3. In thread()
    a. pass the V, W to Compute_nodes(initializeTraining())
    b. receive the TaskStatus, if ACCEPTED call trainModel()

    c. receive {gradient, error_rate}.
    d. Update shared_gradient
4. Average the gradient
5. Update the weights
6. self.mlp_model.validate()

Compute_node:

1. Listen on Task(get initializeTraining(), got the V, W)
2. See whether should reject the task or not
3. If No, see if should inject load(sleep for 3 sec), then call init_training_model(V, W)
4. If init_training_model() return false(error), reject the task
5. If the task is well accepted, receive trainModel(), this will call self.mlp_model.train() to train the model
6. Calculate the gradient
7. Pass it back to Coordinator



PHASE 2 - TEST CASES:

NOTE: Every run used the training parameters:
Rounds: 30, Epochs: 10, H: 20, eta: 0.0001

Initiated via the command `python3 client.py localhost 9090 ./ML/letters 30 10 20 0.0001` in the client terminal

Details	Validation Accuracy	Time Taken
4 compute nodes with load probabilities of 0.2 and a scheduling policy of 1 (random)	32.80%	504.74 seconds
4 compute nodes with load probabilities of 0.8 and a scheduling policy of 1 (random)	32.80%	675.31 seconds
4 compute nodes with load probabilities of 0.2 and a scheduling policy of 2 (least load)	32.80%	457.90 seconds
4 compute nodes with load probabilities of 0.8 and a scheduling policy of 2 (least load)	32.80%	1010.11 seconds
4 compute nodes with load probabilities of 0.2, 0.4, 0.6, & 0.8 and a scheduling policy of 1 (random)	32.80%	595.61 seconds
4 compute nodes with load probabilities of 0.2, 0.5, 0.9, & 0.9 and a scheduling policy of 1 (random)	32.80%	629.21 seconds
4 compute nodes with load probabilities of 0.2, 0.4, 0.6, & 0.8 and a scheduling policy of 2 (least load)	32.80%	678.76 seconds
4 compute nodes with load probabilities of 0.2, 0.5, 0.9, & 0.9 and a scheduling policy of 2 (least load)	32.80%	686.22 seconds

Analysis:
In the first 4 tests, it is demonstrated that increasing the load probability is directly related to the time required to complete a job This is because a higher load probability translates to a larger chance of individual compute node sleeping when receiving a job, which delays its computation. When a least_load algorithm is used, one would naturally expect that the amount of time taken on jobs would decrease, however, in our testing, we did not find this to be the case.

In the second four tests, the random algorithm outperformed least_load in both cases.

We conjecture that a significant portion of the variance between testing runs (particularly the variance seen in run #4, which took 1010.11 seconds to complete) can be attributed to using computing environment that is shared, with other University members, whom may have been running code on the same machines as us at the same time, effectively reducing the share of compute that we are able to get from our scheduler and nodes. Some amount of the variance could also be attributed to luck, since probabilities are significantly at play in this system (probability of sleeping, selecting a ‘busy’ node, etc.). Finally, we note that in the least_load algorithm, the number of active jobs is the only thing taken into account by the scheduler, meaning that nodes without active jobs (which typically have a high load probability, since they are more likely to reject jobs that are sent to them and thus frequently appear to be ‘more available’) may take up scheduler time, particularly as our scheduler is locking when attempting to estab lish a connection to those nodes.

Accuracy stayed the same across all runs, implying that our scheduler is fairly fault-tolerant with respect to scheduling policy and load probabilities. Since we have a max-retry here, and we try to assign the task to least-loaded node, the coordinator managed to find an available node every time, thus the system is fault-tolerant, and the accuracies are consistent.
.