Distributed Training System PA1 – Design Document

Coordinator Node Design

1. Workflow & Task Dispatching

When the client calls the train() method in the Coordinator's Thrift service, the coordinator starts the training process. It first reads the list of available compute nodes from a configuration file (compute_nodes.txt), which contains details like hostnames and ports of the nodes. It also constructs the MLP model based on parameters like H (hidden layer size) and K (output size), which are passed by the client.

The coordinator now reads all the training files, which are to be processed. It puts these files in a thread-safe queue so that more than one worker thread can read and process them simultaneously. Training is carried out in various rounds as the client has requested, with each round including training on all the available files and model update by employing new weights prior to going into the next round.

For efficient task delegation, the coordinator uses a round-robin style of mapping training files to compute nodes. The worker threads pull a training file from the queue and try sending it over Thrift RPC to a compute node. After taking the task by a compute node, the thread waits for a response. But if the node cannot accept the task due to high load or is unavailable, the thread tries with another available node. If all nodes decline the task, the file is requeued for retry later.

2. Multi-threading & Synchronization

To efficiently handle multiple training tasks at once, the coordinator employs Python's threading module so that tasks are executed in parallel. Instead of executing training files sequentially, the coordinator creates a thread pool where the number of threads is two more than the number of compute nodes. This ensures that while some of the threads are busy waiting for responses from compute nodes, there are always some other threads available to run new tasks. A thread chooses a file from the common task queue and assigns it to a compute node. If the node happens to be idle and grabs the task, the thread waits for the reply, ensuring efficient execution of tasks. If the node is unreachable or occupied, the thread selects another node, and if the task is denied by all the nodes, the file is inserted again in the queue to be processed later.

To prevent race conditions and update shared data structures correctly, locks are employed for synchronization by the coordinator. The Round-Robin Lock ensures that threads select compute nodes in controlled and non-conflicting manner. Furthermore, when threads update the shared list of results where computed gradients are kept, a Results Lock avoids inconsistency due to

write races. Once all the tasks for a round are completed, the gradients collected are added up, model weights are updated, and the subsequent round begins with the updated model parameters. The approach offers parallel processing efficiency with data consistency and without losing any task because of temporary failures.

3. Model Aggregation & Validation

Once all training tasks of a round have been completed, the coordinator adds up gradients from the compute nodes. It accomplishes this by adding gradients from successful training jobs and then dividing the sum by the number of successfully completed tasks. The updated new weights are then saved into the model.

When the model is updated, the coordinator evaluates the performance by testing it on an independent validation set. Error on validation is computed and recorded to track the improvement of the model. This is then iterated for the specified number of rounds, each round using the new weights of the previous iteration.

4. Handling Failures & Retries

Since compute nodes can refuse work because they are loaded, the coordinator is designed to fail gracefully. When a node cannot handle a request, the file is queued so it can be tried again sometime later. The system is constructed so that all training files will be handled eventually, although initial attempts will fail.

If all the rounds are completed successfully, the coordinator returns the last validation error to the client. This is the trained MLP model after a number of rounds of training and aggregation.

Compute Node Design

1. Workflow & Task Execution

When the Coordinator assigns a training task to a Compute Node, it does so based on the train_with_weights method, which is invoked with parameters such as initial model parameters, training file, learning rate parameter (eta), number of hidden and output units (h, k), and number of epochs. The compute node would first emulate the system load prior to deciding whether it will run the task. When the task is received, the compute node initializes the MLP model and trains it using the provided weights and dataset. Training is performed via the ml.py script from PA1, with the core logic for training a neural network incorporated within it. When training is complete, the compute node calculates the gradient updates by obtaining the difference between the initial weights and final weights and sends back this information to the Coordinator using the TrainResult structure.

2. Load Simulation & Task Scheduling

In order to emulate a realistic load, the Compute Node also has an emulated load condition where, based on a randomly calculated number against load_probability, the node can reject the job in order to emulate real-world server overload. When the task is rejected, the Coordinator reassigns it to a different node or resubmits later. The scheduling policy determines the allocation of tasks: when loaded as load-based scheduling, rejection probability increases with load_probability to balance dynamically loads across multiple compute nodes. Heavily loaded nodes tend to receive fewer new tasks, and this allows the system to efficiently balance workload.

3. Model Training & Gradient Computation

Once a task is received, the Compute Node initializes the model using init_training_model and trains using the train function. While training, the model's weights are updated iteratively over multiple epochs. The final updated weights are compared with the initial weights to compute the gradient updates. The gradients are propagated back to the Coordinator, which aggregates them across all compute nodes to update the global model. Compute Node does not have any persistent state except for the current training session so that each task is run independently without affecting the next rounds.

4. Communication & Response Handling

The Compute Node interacts with the Coordinator using Apache Thrift RPC to issue efficient remote procedure calls. Upon training, the TrainResult structure is populated with the training error, new weights, and an acceptance flag. The response is returned to the Coordinator, which combines responses from multiple compute nodes and updates the global model accordingly. This interaction keeps the distributed training workflow in sync across all the nodes.

Client Architecture

The client begins the training process by establishing communication with the coordinator and sending the necessary parameters for model training. It is performed using a CLI command that accepts arguments such as the IP address of the coordinator, port, dataset directory, number of rounds of training, epochs, size of the hidden layer (H), and learning rate (eta). The client understands these arguments and then creates a TCP socket connection with the coordinator by using the provided IP address and port. This allows the client to call the RPC methods of the coordinator and continue with the training procedure.

Upon connection, the client invokes the train function defined in the coordinator's Thrift file. This function is in charge of the entire training process and ultimately returns the final validation error once all training cycles are complete. Besides this function call, the client passes main parameters such as the directory containing training files, the number of output units, and the learning rate for the model. For the purpose of seamless communication, the client employs Thrift-generated Python stubs, which make the client, coordinator, and compute nodes compatible. This framework supports a structured and uniform flow, where the client can be the entry point for initiating and managing distributed training.

Thrift Interface Design

Weights Struct
The Weights struct holds two lists of doubles, V and W, as the model parameters in a typical multi-layer perceptron (MLP). This allows for every node to exchange the model's current state with the coordinator and updated gradients without manually parsing arrays.

TrainResult Struct
TrainResult struct holds details of a training task. It possesses a boolean field, accepted, to indicate whether a compute node accepted or refused the task due to load. It also stores the previous training error which is helpful to monitor progress and to return the result. The coordinator relies on this data to see if the compute node decided to accept the job and present the final answer or reject the job.

Coordinator Service
The Coordinator service has the train() method, the entry point of distributed training. Clients call this method to begin training, providing parameters such as the directory with training data files, number of rounds and epochs, learning rate and hidden layer size. The method returns the final validation error after training has completed.

ComputeNode Service
The ComputeNode service provides the train_with_weights() method, which is invoked by the coordinator to assign training tasks. Apart from the training file path and initial weights, the coordinator also passes parameters like k, h, and eta, which define the model's configuration and learning rate. The method also accepts a scheduling_policy indicator, which defines how tasks are accepted or rejected according to varying loads. Upon completion, the compute node will return a TrainResult object, which indicates whether the task was accepted with any gradients and training error.

Testing Scenarios

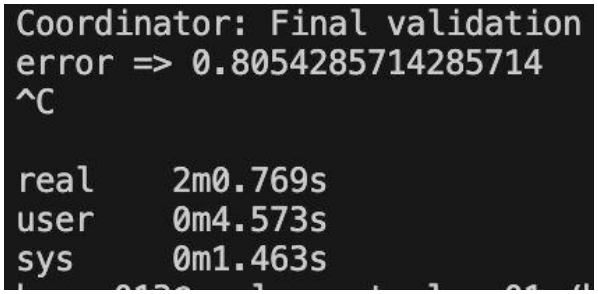
We have conducted various test cases with random scheduling algorithm and load balancing algorithm for the distributed system training.

Below are the description of the test cases

Test Case Number	No. of Compute Nodes	Scheduling policy	Load probability	Epoch	H	Eta	No. of rounds	Real time	User time	System time	Validation error
1	4	Load Balancing	[0.2,0.2,0.2,0.2]	15	20	0.0001	5	2m 0s 769ms	4s 573ms	1s 463ms	0.805428571428571
2	4	Load Balancing	[0.8,0.8,0.8,0.8]	15	20	0.0001	5	5m 29s 160ms	5s 6ms	1s 503ms	0.805428571428571
3	4	Load Balancing	[0.2,0.4,0.6,0.8]	15	20	0.0001	5	2m 35s 695ms	4s 609ms	1s 515ms	0.805428571428571
4	4	Random	NA	15	20	0.0001	5	2m 6s 902ms	4s 485ms	1s 353ms	0.805428571428571
5	2	Random	NA	15	20	0.0001	5	2m 49s 911ms	4s 593ms	1s 507ms	0.805428571428571
6	2-4	Random	NA	15	20	0.0001	5	2m 13s 833ms	4s 537ms	1s 478ms	0.805428571428571
7	4	Random	NA	15	20	0.0001	25	10m 38s 634ms	18s 287ms	5s 196ms	0.324
8	4	Load Balancing	[0.5,0.5,0.5,0.5]	50	24	0.0001	15	18m 46s 705ms	11s 430ms	3s 414ms	0.27

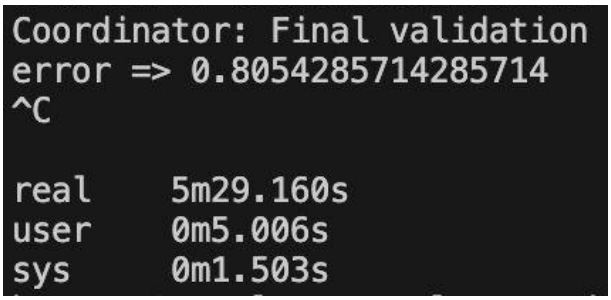
Test Case 1

Description:
All four nodes use load-balancing with a 20% probability of load for all of them. Those tasks which are encountered with a busy node are re-queued until they are accepted. We train for 15 epochs on each accepted task with 20 hidden units and eta=0.0001, over 5 rounds in total. We have low rejections here because the probability of load is 20% for all of the nodes and thus the task is done earlier in only 2 minutes. Although moderate rejection, real time is relatively low (2 m 0 s). The final verification error of 0.8054 attests convergent behavior under uniform moderate loading.



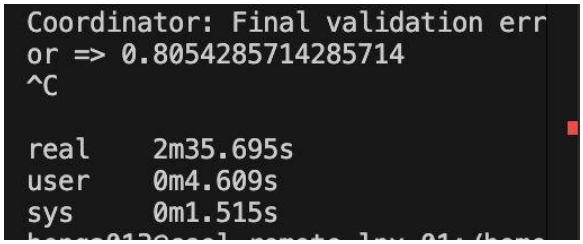
Test Case 2

Description:
In this scenario, the load probability for each of the 4 nodes is 0.8, simulating high loading. We run 15 epochs, 20 hidden, and eta=0.0001 for 5 iterations. As all the nodes reject jobs 80% of the time, real time takes 5 m 29 s. Even on repeated rejections and re-queues, the final validation error is 0.8054285714285714, illustrating that repeated retry and gradient accumulation can cope with high rejection.



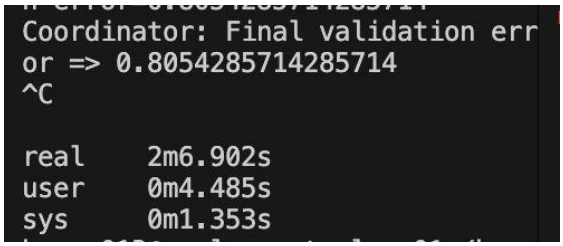
Test Case 3

Description:
All four nodes are load-balancing but with different probabilities—0.2, 0.4, 0.6, and 0.8—so that some nodes reject more than others. The training uses 15 epochs, 20 hidden units, and eta=0.0001 for 5 rounds. Tasks on the higher-load nodes are rejected more, but re-queues ensure acceptance in the long run. Actual time is 2 m 35 s, and last validation error is 0.8529, which shows successful training under heterogeneous load. This is approximating the moderate load for the system as a whole and thus the time to perform the task is between that of the low loaded and high loaded systems.



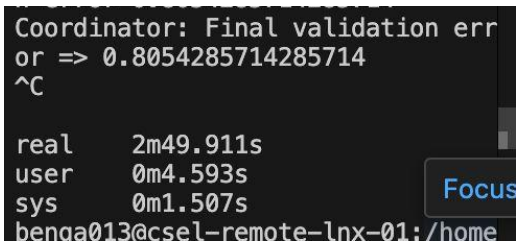
Test Case 4

Description:
Random scheduling is transformed to random distribution of tasks across the 4 nodes without any rejections. All accepted jobs are run 15 epochs with 20 hidden units and eta=0.0001, 5 times in total. The real time is 2 m 6 s-shorter compared to heavily loaded conditions-and the final validation error, 0.8054, shows that random assignment with no rejections can converge properly.



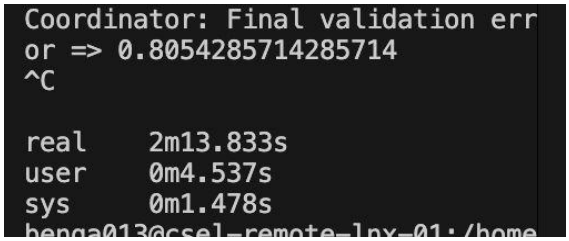
Test Case 5

Description:
On 2 nodes with random scheduling, tasks are always accepted. We train for 15 epochs on each with 20 hidden units and eta=0.0001 for 3 cycles. More nodes increase the parallel speedup with real time still at 2 m 33 s, with last step validation error being 0.8054, indicating better accuracy on more parallel resources. We have only 2 nodes working here and therefore it is slower than test case 4 as it has 4 nodes since there are only 2 nodes available to execute the job.



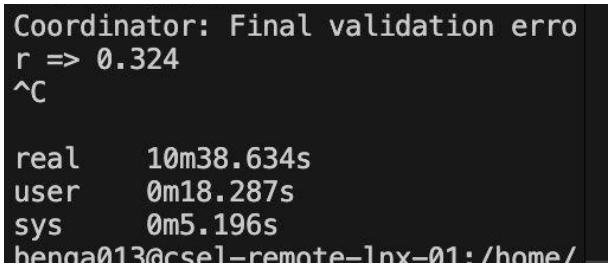
Test Case 6

Description:
Now only 2 nodes with random scheduling are being used initially. And then we add 2 more nodes after a certain duration. They imitate the feature that nodes can come in during execution and take up the tasks. Here the time of the whole system comes in between the time of 2 and 4 nodes respectively. Every accepted task is trained for 15 epochs, 20 hidden units, eta=0.0001, for a total of 5 rounds. The real time is 2 m 13 s, slightly shorter since there are fewer tasks in total, and the final validation error of 0.8054 is evidence that even with a small cluster, random assignment can lead to good convergence.



Test Case 7

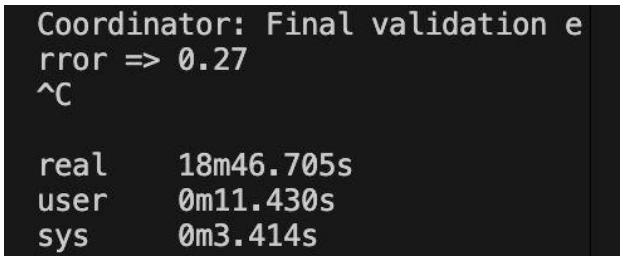
Description:
4 nodes, random scheduling to prevent rejection. We train for 15 epochs with 20 hidden units, eta=0.0001, and 25 rounds. Real time becomes 10 m 41 s. The final validation error of 0.324 shows that the model is well trained with the right set of parameters.



Test Case 8

Description:
With 4 nodes with 50% chance of load, jobs get rejected and re-queue quite frequently. Each file

is trained for 50 epochs, 24 hidden units, and eta=0.0001, for 15 rounds. Real time taken is 18 m 10 s, with validation error at last being 0.27, the minimum of these cases, which indicates how effective number of rounds and parameters used like epochs etc. are. Here the time taken is more because there are rejections that can take place, and even if it does, the system manages well by training all the files and reducing the validation error.



Conclusion

These results show how scheduling policies (Random vs. Load Balancing) and load probabilities affect the actual completion time, as well as the final model's validation error. In cases where compute nodes have higher or more varied load probabilities, load balancing can result in better throughput and potentially more balanced validation errors, though random scheduling is still adequate when loads are low or equal.

By systematically varying these parameters, we demonstrate the capability of the system to handle homogeneous and heterogeneous workloads and present insights into how different configurations may influence performance and training outcomes.

References

We partially relied on ChatGPT for code suggestions in two areas of our distributed system:

1. Connecting to compute nodes using sockets
2. Reading port and hostname information from a file

Python Software Foundation, *"Queue Objects,"* Python 3 Documentation, accessed March 31, 2023, https://docs.python.org/3/library/queue.html#queue-objects.

Python Software Foundation, *"Thread Objects,"* Python 3 Documentation, accessed March 31, 2023, https://docs.python.org/3/library/threading.html#thread-objects.

Python Software Foundation, *"Lock Objects,"* Python 3 Documentation, accessed March 31, 2023, https://docs.python.org/3/library/threading.html#lock-objects.

Steps to execute

1. **Unzip the Folder**
    a. Extract the zipped folder (e.g., distributed_training.zip) into a local directory of your choice.
    b. Navigate to phase2/ML/ within the unzipped folder.
2. **Create Virtual Environment and install all the requirements**
    a. python3 -m venv myenv
    b. source myenv/bin/activate
    c. pip install -r requirements.txt
    d. Execute the compute_nodes, coordinator_node and client in the virtual environment
3. **Configure Compute Nodes**
    a. Open the file compute_nodes.txt (if used) and ensure it lists the host and port for each compute node you plan to run.

    For example:
    Localhost,9091
    Localhost,9092
    Localhost,9093

4. **Start the Compute Nodes**
    Open additional terminals—one for each compute node.
    python3 compute_node.py <port> <load_probability>
        Example:
        python3 compute_node.py 9091 0.2
        python3 compute_node.py 9092 0.4

    Each node will start listening on its respective port and be ready to accept tasks from the coordinator.

5. **Start the Coordinator:**
    python3 coordinator_node.py <port> <scheduling_policy>
    Example: python3 coordinator_node.py 9095 2
    (Here, 9095 is the coordinator's port, and 2 represent the load-balancing policy.)

6. **Run the Client**
    Once the coordinator and compute nodes are running, open another terminal for the client.
        Run:
        python3 client.py <coordinator_ip> <coordinator_port> <dir_path> <rounds> <epochs> <h> <eta>
    Example:

```
python3 client.py localhost 9095 letters 3 75 24 0.0001
```

7. You can find the final Validation after training in the client console.