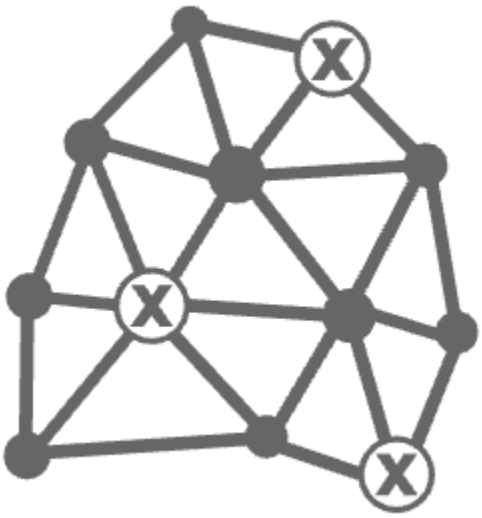


CloudMesh Final Report

By

Joon Kang (j222kang)

Palaash Kolhe (pkolhe)

Jordan Mao (j52mao)

Rayaq Siddiqui (m72siddi)

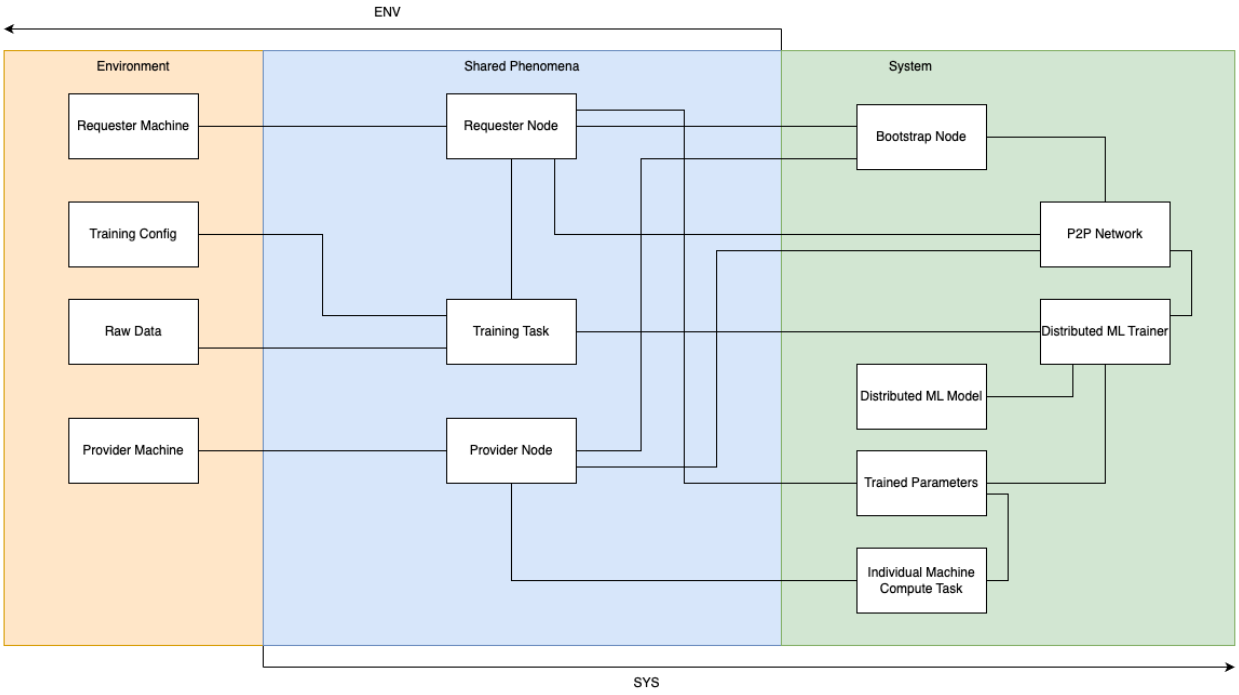
Tony Sun (w225sun)

Group 18

Table of Contents

Requirements Specification

1. Domain Model



2. Use Case Diagram

2.1 For Requester Machine

InitializeRequester()
ConnectToP2PNetwork()
SetGeographicLocation(location)
SetTrainingData(training_data_path)
SetModelConfiguration(model_configuration_path)
SetBatchSize({1, …, n})

- Sets the batch size for the training loop at each iteration.

SetLearningRate(rate)

- Sets the learning rate parameter for the gradient update for the model training loop.

SetNumEpochs({1, …, n})
SetModelType({SimpleCNN, Resnet18, …, Custom})
SetDesiredDeviceType({CPU, GPU, MPS})

- Sets what kind of resources the requester wants from the provider

SetNumberDesiredProviderPeers(num_peers)
LeaveNetwork()
StartTrainingWorkflow()
ReceiveTrainingResults()

- Start polling for the results from the leader provider

2.2 For Provider Machine

InitializeProvider()
ConnectToP2PNetwork()
SetGeographicLocation(location)
SetAvailableRam(ram)
SetAvailableCores(num_cores)
SetDeviceType({CPU, GPU, MPS})
AcceptRequests()

- Accept incoming requests from a provider peer
- Once a request is accepted, no more requests will be accepted until the current request is completed

LeaveNetwork()

3. List of assumptions, exceptions, and variations

3.1 Domain Assumptions

1. The requester trusts that the providers will accurately execute the training task based on the specified configurations and return a viable model.
2. The requester may be providing sensitive data to the provider nodes to train the models. The requester needs to trust that the provider is not corrupting the data.
3. The provider trusts that the system is providing reasonable sized work loads that will not cause harm to the provider machine. The provider will want to ensure that the workflow is a part of using the provider machine equitably and not causing harm to the machine.
4. The network connection between Requester, Providers, and the bootstrap node is stable enough for uninterrupted data transfer. The architecture assumes that the network is consistent and there isn't network related corruption in the architecture
5. Geographic location will not slow down the training process and be a bottleneck in the training workflow. The requester will want to ensure that the provider nodes are in a close proximity so there is a reasonable bandwidth in the training jobs.
6. Data is valid, specified correctly when starting training jobs, and formatted correctly. The ML training architecture just takes the data under the assumption it is formatted in a specified way. We would want to keep the data structured similarly
7. The bootstrap node will consistently handle node registration and network connections without failure. The bootstrap node is the main entrypoint into the architecture and we would want to ensure that the bootstrap node constantly stays up.

3.2 Exceptions

1. A provider goes offline or leaves the network before the training workflow completes, due to reasons such as network outages, machine powering off, or system crashes.
2. The requester and provider disconnect during data or model transmission, resulting in incomplete/corrupted data. This can be due to similar reasons like network outages, machine powering off, or system crashes.
3. NAT/firewalls may prevent other machines from initiating communication with requesters and/or providers behind these layers, making them unreachable from the outside unless outbound requests are made from these machines first.
4. Provider hardware does not match the specified resources broadcasted to the network. For example, the provider may have less RAM than specified, or the provider might not have an available GPU when it specified the availability of one.
5. Training data format or model configuration transmitted does not match the expected input. Errors in the data/model configuration formatting from requester could cause issues on the provider side when parsing and using them for training.
6. Provider's resources are insufficient to handle the requested training task, causing training delays or failures. In the case that a provider runs out of resources to handle the

requested training task, such as running out of RAM or overloading the CPUs or GPUs due to other processes running on the machines that compete for the same resources.
7. Insufficient number of available provider peers on the network may delay a pending training task from being started.
8. Bootstrap node server going down would prevent requester and provider peers from joining the network.

3.3 Variations

1. In a set of provider peers (followers), some nodes may be faulty, either due to slower GPU's, or unexpected departures from the network. As a result, the generated training result might only be from a subset of provider peers rather than all of them. This creates a partial result, but since we used asynchronous ML patterns where each follower's results are averaged as soon as they return to the leader, we can still achieve a final result comparable to one involving all provider peers.
2. Use a service other than ngrok to transfer data from one node to another. The same training result would be accomplished as a result of this, the only difference would be the speed in which it is achieved. By removing dependency on an out of house service, we might be able to speed up the data transfer.
3. Different provider peers are selected based on hardware capabilities. We might get training results that vary as a result of this. Training that is performed on a hardware accelerated M1 chip might produce different results than another chip.
4. Different provider peers selected based on geographic proximity or latency will produce different training times. Slow computers that are closer might take longer to train compared to fast computers that are farther away.
5. The training task itself could be some other form of ML task. It might be a computer vision training task instead of the current classification tasks. It might be a NLP task too. All of these are variations of the current product, but with different tasks.
6. The number of epochs and the number of partitions can be varied. A larger number of epochs and partitions will produce results that vary from results produced with a smaller number of epochs and partitions.
7. A varying number of peers can be used. More peers will produce worse results since the data will be spread very thinly across the peers. Less peers will increase the total training time.
8. The available cores and RAM allowed by a provider will produce different training times for each provider. More cores and more RAM will produce faster results than less cores and RAM.

4. Functional Requirements

4.1 InitializeRequester(port)

4.1.1 Typical Scenario

The user joins the program as a requester. This function is called to initialize the user's machine. A port is specified here to set up the requester's communication channels. This port will be used later to connect to the P2P network.

4.1.2 Exceptional Scenario

Initialization fails due to an error. The error could be due to -

- Missing or out-of-date dependencies or the user's computer turning off mid-initialization.
- the port specified might be outside of the allowed range for ports (1–65535).

4.1.3 Alternative Scenario

Initialization is delayed since the user's computer is undergoing a system check or slowing down.

4.2 ConnectToP2PNetwork(port)

4.2.1 Typical Scenario

The requester is successfully initialized. It now attempts to join the P2P network by broadcasting a connection request to the bootstrap node, which contains data such as the requester's IP address and the port. It waits for an acknowledgement from the bootstrap peer after this. Once the connection is established, the requester is ready to send and receive data.

4.2.2 Exceptional Scenario

The connection may fail if there is a firewall that prevents the sending or receiving of data. The router itself must open up an internet-facing port to receive information through TCP. There could also be a weak internet connection, which would cause packet loss.

The port may also be occupied by another service or application, causing a conflict. This will create an exception and exit the program.

4.2.3 Alternative Scenario

The requester will attempt to connect to the internet. If the port provided is already occupied, then the system will automatically choose another open port. This process will be logged to the user. The connection will still succeed, just with a different port.

4.3 SetGeographicLocation(location)

4.3.1 Typical Scenario

The requester machine connects to the network and the geographic location associated with their connection is passed into this function. This is now stored in the system for a singular connection.

4.3.2 Exceptional Scenario

A requester machine connects to the network with a VPN, and their login location is incorrect as a result of this. Due to this, they will join an incorrect cluster of peers, which will increase the overall network latency for that cluster.

4.3.3 Alternative Scenario

A requester machine specifies the location from where they are joining in from. They can manually enter which location they would like to join.

4.4 SetTrainingData(training_data_path)

4.4.1 Typical Scenario

The requester machine will connect to the network and the training data path is passed into the training task through this function. This function will check the completeness of the data, i.e. train / test data folders in path, and update the task config.

4.4.2 Exceptional Scenario

The requester machine will connect to the network and the training data path is passed into the training task through this function, but the data is incomplete, i.e. missing train data folder. The task config will be updated, but the training task will eventually fail.

4.4.3 Alternative Scenario

The requester machine provides some cloud storage bucket with the training data and is passed into the task config. The training task will succeed if the form of cloud storage is supported, else, it will fail.

4.5 SetModelConfiguration(model_configuration_path)

4.5.1 Typical Scenario

The requester machine will connect to the network and pass in the model configuration to the training task through this function. This model configuration file will contain a variety of different hyperparameters, such as batch size, learning rate, num epochs, num provider peers, model type, data path.

4.5.2 Exceptional Scenario

The requester will connect to the network and pass in a model configuration for some unreasonable specification. Like a 100B+ parameter model with a large learning rate, small batch size, large num epochs, and a large number of provider peers. The provider peer network would definitely crash and not be able to handle this

4.5.3 Alternative Scenario

The requester machine will connect to the network and pass in None, to specify the default configurations. We will also have some default configurations for vision and language models they can pass in as well.

4.6 SetBatchSize({1, ..., n})

4.6.1 Typical Scenario

The requester will connect to the network and pass in the target batch size to the training task through this function. This batch size is used in training and determines how many samples will be used per iteration. Batch size is in the domain [1, n]

4.6.2 Exceptional Scenario

The requester will connect to the network and pass in a really large or small target batch size to the training task through this function. The provider machine will either take too long with small

batch sizes or will crash due to not having enough RAM if large batch sizes. We can clip the batch size depending on the provider machine.

4.6.3 Alternative Scenario

The requester will connect to the network and pass in a batch size outside of [1,n]. The provider peer will overwrite this batch size and default it to the system global batch size of 16.

4.7 SetLearningRate(rate)

4.7.1 Typical Scenario

The requester will connect to the network and pass in the learning rate to the training task through this function. The learning rate will determine the size of the gradient update per iteration. The typical learning rates are 0.001, 0.01, 0.1.

4.7.2 Exceptional Scenario

The requester will connect to the network and pass in the learning rate to the training task through this function. The learning rate is large (i.e. 100). This may cause numerical instability and cause the weights to overflow and cause issues on the provider peer and crash the training workflow.

4.7.3 Alternative Scenario

The requester will connect to the network and pass in the learning rate to the training task and it is less than 0. The provider peer will overwrite this learning rate to the default of 0.001.

4.8 SetNumEpochs({1, ..., n})

4.8.1 Typical Scenario

The requester will connect to the network and pass in the num epochs to the training task through this function. The number of epochs will determine the duration the model will train for. The typical num epochs is 100, 500, 1000.

4.8.2 Exceptional Scenario

The requester will connect to the network and pass in the num epochs to the training task through this function. The number of epochs is some float or unreasonably large. This would never release the provider peer from its training flow. We will either cast it to an integer or clip the value to some reasonable upper bound.

4.9 SetModelType({SimpleCNN, Resnet18, …, Custom})

4.9.1 Typical Scenario

The requester machine will connect to the network and pass in the model type to the training task through this function. This model file will contain a PyTorch “Model” object in our ml architecture.

4.9.2 Exceptional Scenario

The requester will connect to the network and pass in a model type for a 100B+ parameter LLM. Very few peers will be able to support this model and if sent to a provider peer, it will likely cause the provider machine to crash since it is not a reasonable size workload.

4.9.3 Alternative Scenario

The requester machine will connect to the network and pass in some default model type name to the training task through this function. This is for a user who wants to train a type of model, such as vision, but does not know how to make custom models and uses the default based on our documentation.

4.10 SetDesiredDeviceType({CPU, GPU, MPS})

4.10.1 Typical Scenario

The requester machine will connect to the network and pass in the desired device type for the training to occur on through this function. The network will search for peers with only this device type.

4.10.2 Exceptional Scenario

The requester will connect to the network and pass in a device type that is infrequent in our network or not supported and request some number of peers. The bootstrap node might fail finding the peers or the nodes might be far apart geographically, which will create latency in the peer to peer network.

4.10.3 Alternative Scenario

The requester machine will connect to the network and pass in None. The function will default to using the most frequently available device in the local network. Thus decreasing latency and increasing the chances of the bootstrap node being able to create a network,

4.11 SetNumberDesiredProviderPeers(num_peers)

4.11.1 Typical Scenario

After the requester connects to the network, it specifies the desired number of provider peers it would like to distribute the workload efficiently. The bootstrap node finds the specified number of provider peers and returns their IP addresses and ports. The requester peer then establishes a connection with these providers.

4.11.2 Exceptional Scenario

The requester sets a number of provider peers that is greater than the amount of provider peers available on the network. This will log an error to the requester, and ask the requester to set the desired provider peers to the maximum number of providers available on the network.

4.11.3 Alternative Scenario

The system dynamically sets the number of provider peers based on how many are available in the network. It will automatically reduce the requested provider peers to the total available peers in the system, and log this to the requester.

4.12 LeaveNetwork()

4.12.1 Typical Scenario

The requester is completed with its work on the P2P network. It is now ready to leave the network. It sends a “Leave” message to the bootstrap node, notifying it of its intent to leave the network. This will cause the bootstrap node to remove the requester peer’s entry from its mapping table, which associates peers with their IP addresses and ports. Upon successful removal, a confirmation message is returned to the requester.

4.12.2 Exceptional Scenario

Disconnection fails due to network issues. The requester peer's "Leave" message is lost in transit, the bootstrap encounters an error while processing the request, or the confirmation message is lost in transit back to the requester. Both the requester and the bootstrap node are designed to attempt a limited number of retries in this situation. If a confirmation message is not returned in the maximum number of retries, the requester will terminate on its end. While this may result in "ghost" nodes existing temporarily in the bootstrap node's mapping table, such entries will be cleared during the routine weekly table reset to maintain network integrity.

4.12.3 Alternative Scenario

The system queues the disconnection request in the middle of processing data, allowing current tasks to finish executing before disconnecting from the network. This prevents data corruption, and still accomplishes the task of disconnecting from the network.

4.13 StartTrainingWorkflow()

4.13.1 Typical Scenario

The requester will initiate the training workflow among the provider peers assigned the task. It is then up to the provider peers to coordinate among themselves (through the leader peer) to synchronize parameter and gradient updates for the distributed training process.

4.13.2 Exceptional Scenario

A provider peer disconnects from the network after it has been assigned to this task but before the requester initiates the training workflow. The requester will be unable to communicate with the missing provider when this unexpected behaviour occurs.

4.13.3 Alternative Scenario

The requester will look for additional provider peers again through SetNumberDesiredProvderPeers(num_peers) to regain a desired set number of providers to assign to its training task.

4.14 ReceiveTrainingResults()

4.14.1 Typical Scenario

The requester will check if the training task is completed by requesting the results from the leader provider peer. The leader provider peer should then return the trained model to the requester, and all the provider peers assigned to that training task should mark themselves as available in the network.

4.14.2 Exceptional Scenario

It is possible that the leader provider peer leaves the network before the requester reconnects after some duration to ask for the training results.

4.14.3 Alternative Scenario

If the requester is unsuccessful in polling the leader provider peer for training results, it can go through the list of follower peers to request the training results, since each of them should have a copy of the training results.

4.15 InitializeProvider()

4.15.1 Typical Scenario

The user joins the program as a provider. This function is called to initialize the user's machine. A port is specified here to set up the requester's communication channels. This port will be used later to connect to the P2P network.

4.15.2 Exceptional Scenario

Initialization fails due to an error. The error could be due to -

- missing or out-of-date dependencies or the user's computer turning off mid-initialization.
- the port specified might be outside of the allowed range for ports (1–65535).
- lacks adequate CPU, memory, or disk space

4.15.3 Alternative Scenario

Initialization is delayed since the user's computer is undergoing a system check or it is slowing down. There could be background tasks occurring, which will cause minimal resources to be allocated to CloudMesh.

4.16 ConnectToP2PNetwork(port)

4.16.1 Typical Scenario

The provider is successfully initialized. It now attempts to join the P2P network by broadcasting a connection request to the bootstrap node, which contains data such as the provider's IP address and the port. It waits for an acknowledgement from the bootstrap peer after this. Once the connection is established, the provider is ready to send and receive data.

4.16.2 Exceptional Scenario

The connection may fail if there is a firewall that prevents the sending or receiving of data. The router itself must open up an internet-facing port to receive information through TCP. There could also be a weak internet connection, which would cause packet loss.

The port may also be occupied by another service or application, causing a conflict. This will create an exception and exit the program.

4.16.3 Alternative Scenario

The provider will attempt to connect to the internet. If the port provided is already occupied, then the system will automatically choose another open port. This process will be logged to the user. The connection will still succeed, just with a different port.

4.17 SetGeographicLocation(location)

4.17.1 Typical Scenario

The provider machine connects to the network and the geographic location associated with their connection is passed into this function. This is now stored in the system for a singular connection.

4.17.2 Exceptional Scenario

A provider machine connects to the network with a VPN, and their login location is incorrect as a result of this. Due to this, they will join an incorrect cluster of peers, which will increase the overall network latency for that cluster.

4.17.3 Alternative Scenario

A provider machine specifies the location from where they are joining in from. They can manually enter which location they would like to join.

4.18 SetAvailableRam(ram)

4.18.1 Typical Scenario

The provider peer sets the available RAM of the machine being used for the training task. The average expected RAM falls between 8-32GB. This allows the requester to pick the providers with the highest-performance machines.

4.18.2 Exceptional Scenario

The provider peer declares an amount of available RAM that is too low to support machine learning (such as 4GB), which would cause it to be a bottleneck in the distributed training workflow, where its slowness would cause it to work with consistently outdated parameters and gradients.

4.18.3 Alternative Scenario

The provider peer user can leave the available RAM specified as empty, allowing our system to declare a default minimum of 8 GB available for use.

4.19 SetAvailableCores(num_cores)

4.19.1 Typical Scenario

The provider peer sets the available number of cores of the machine being used for the training task. The average expected number of CPU cores for training ML models is expected to be at least 16. This allows the requester to pick the providers with the highest-performance machines.

4.19.2 Exceptional Scenario

The provider peer declares a number of cores that is too low to support machine learning (such as 8 cores), which would cause it to be a bottleneck in the distributed training workflow, where its slowness would cause it to work with consistently outdated parameters and gradients.

4.19.3 Alternative Scenario

The provider peer user can leave the available number of cores specified as empty, allowing our system to declare a default minimum of 16 cores available for use.

4.20 SetDeviceType({CPU, GPU, MPS})

4.20.1 Typical Scenario

The provider peer sets the device type of the machine being used for the training task, which is either CPU, GPU, or MPS (allows accelerated pytorch training on MacOS). This allows the requester to pick the providers with the highest-performance machines in order of GPU>MPS>CPU.

4.20.2 Exceptional Scenario

If a provider incorrectly declares its device type, the training code will produce faults when trying to leverage nonexistent resources. For example, if a provider declares that it has access to a GPU when it actually does not, the training code will be unable to use CUDA through GPUs for accelerated training.

4.20.3 Alternative Scenario

If the provider produces faults when trying to leverage computation acceleration tools from GPU or MPS, it may be an indication that the provider incorrectly declared the device type. The provider should then default to using the CPU to complete the training task.

4.21 AcceptRequests()

4.21.1 Typical Scenario

The provider peer marks itself as ready to accept training task requests, enabling requesters to assign the provider to a training task to start on.

4.21.2 Exceptional Scenario

If a provider peer is ready to accept requests and two requesters attempt to assign the same provider to two separate training tasks, a race condition occurs, in which only one requester will have obtained the provider.

4.21.3 Alternative Scenario

If a requester finds a provider marked as ready to accept requests, attempts to assign it a training task, but fails due to a race condition where a different requester obtained the provider at the same time, the requester will look for other provider peers and try again.

4.22 LeaveNetwork()

4.22.1 Typical Scenario

The provider has completed its work on the P2P network. It is now ready to leave the network. It sends a “Leave” message to the bootstrap node, notifying it of its intent to leave the network. This will cause the bootstrap node to remove the provider peer’s entry from its mapping table, which associates peers with their IP addresses and ports. The provider also notifies other connected peers to remove its entry in their respective mapping tables. Upon successful removal, a confirmation message is returned to the provider.

4.22.2 Exceptional Scenario

Disconnection fails due to network issues. The provider peer’s “Leave” message is lost in transit, the bootstrap encounters an error while processing the request, or the confirmation message is lost in transit back to the provider. Both the provider and the bootstrap node are designed to attempt a limited number of retries in this situation. If a confirmation message is not returned in the maximum number of retries, the provider will terminate on its end. While this may result in “ghost” nodes existing temporarily in the bootstrap node’s mapping table, such entries will be cleared during the routine weekly table reset to maintain network integrity.

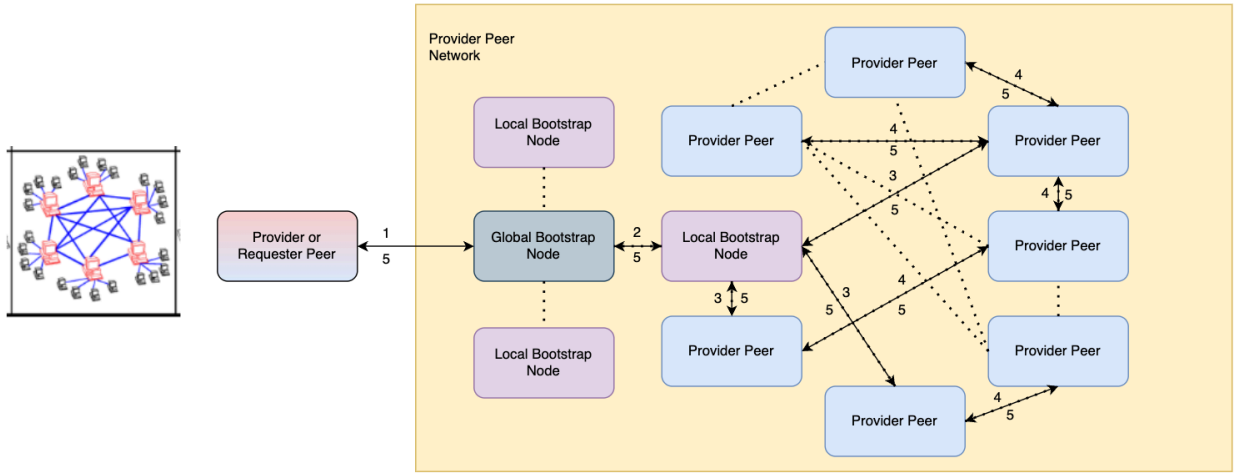
4.22.3 Alternative Scenario

The system queues the disconnection request in the middle of processing data, allowing current tasks to finish executing before disconnecting from the network. This prevents data corruption, and still accomplishes the task of disconnecting from the network.

Design Docs

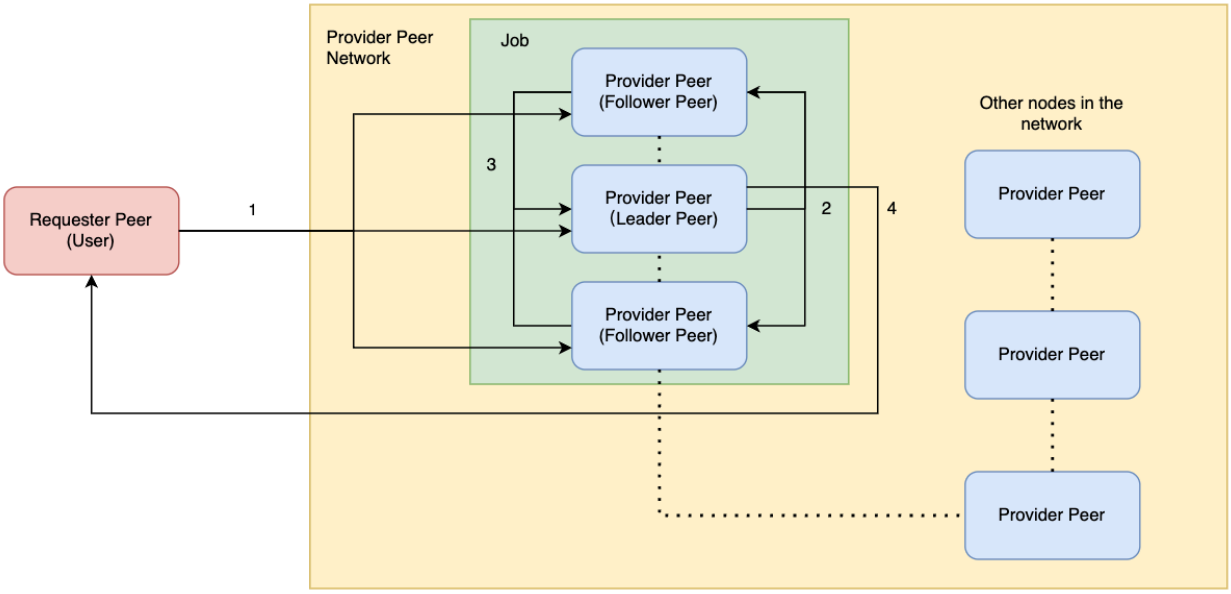
5. High-Level Architecture

5.1 Peer Discovery



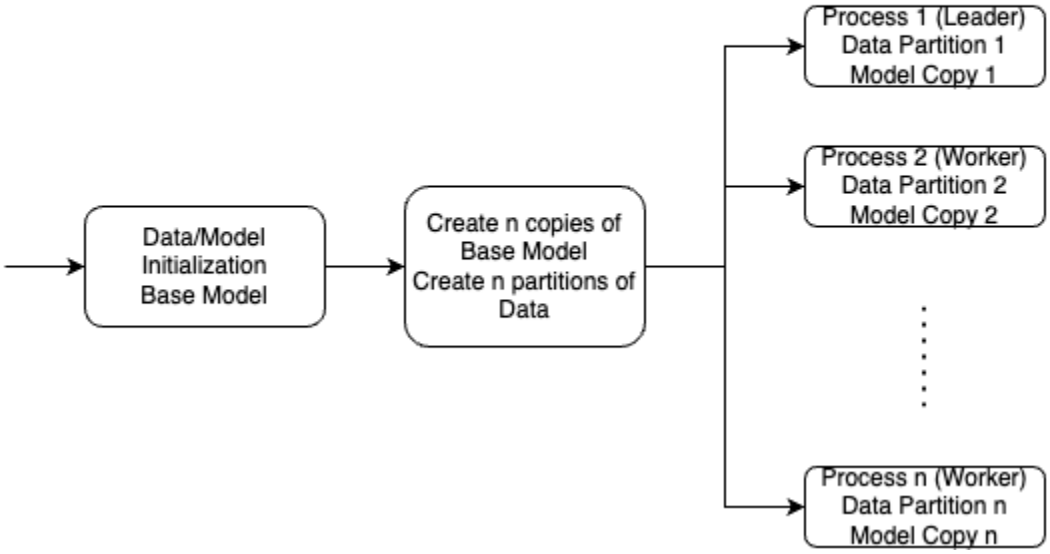
1. A new peer that wants to join the network submits a join request to the global bootstrap node. The address of this node is static.
2. The global bootstrap node redirects the request to the local bootstrap node based on geographical location.
3. The local bootstrap node compiles a list of provider peers that include its neighbours, alongside the neighbours of neighbours.
4. We continue searching across the neighbours of each node until we believe a sufficient amount of provider peers have been found.
5. We return a list of all visited provider peers that fall within a certain criteria (i.e. nodes within a certain physical distance away) and update the list of neighbours for each node as we propagate back to the provider/requester peer that wants to join our network.

5.2 Task Workflow



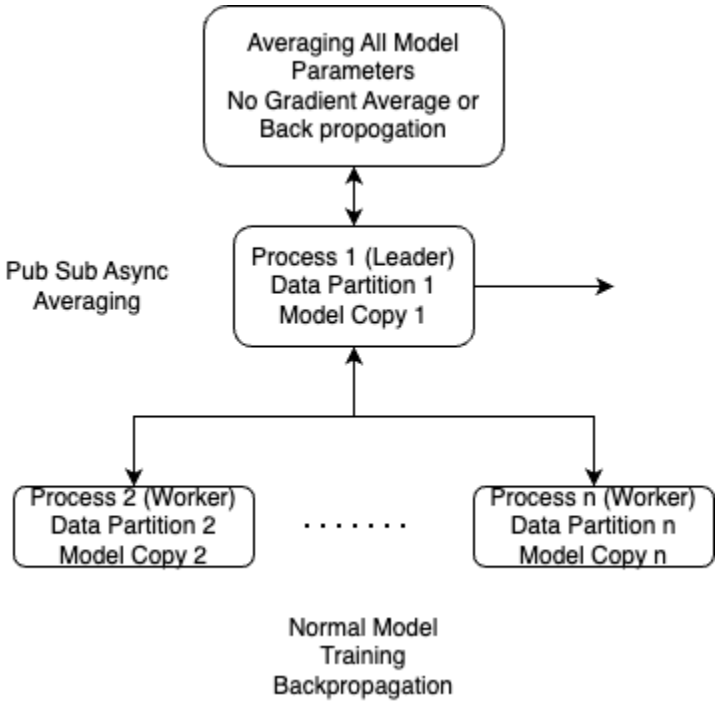
1. The requester peer submits a task to a list of provider peers alongside the data it needs to train the model. One of these provider peers is designated to be the leader peer who has the additional responsibility of aggregating the model.
2. The leader peer will send to the provider peers the current iteration of the trained model after averaging out the gradient descents.
3. Each individual provider peer will calculate its own gradient descent with the training data and send it to the leader peer for averaging.
4. Steps 2 and 3 will repeat until the model has completed its training process.
5. The leader peer sends the model back to the requester peer after it has finished training.

5.3 Distributed Model Training Initialization



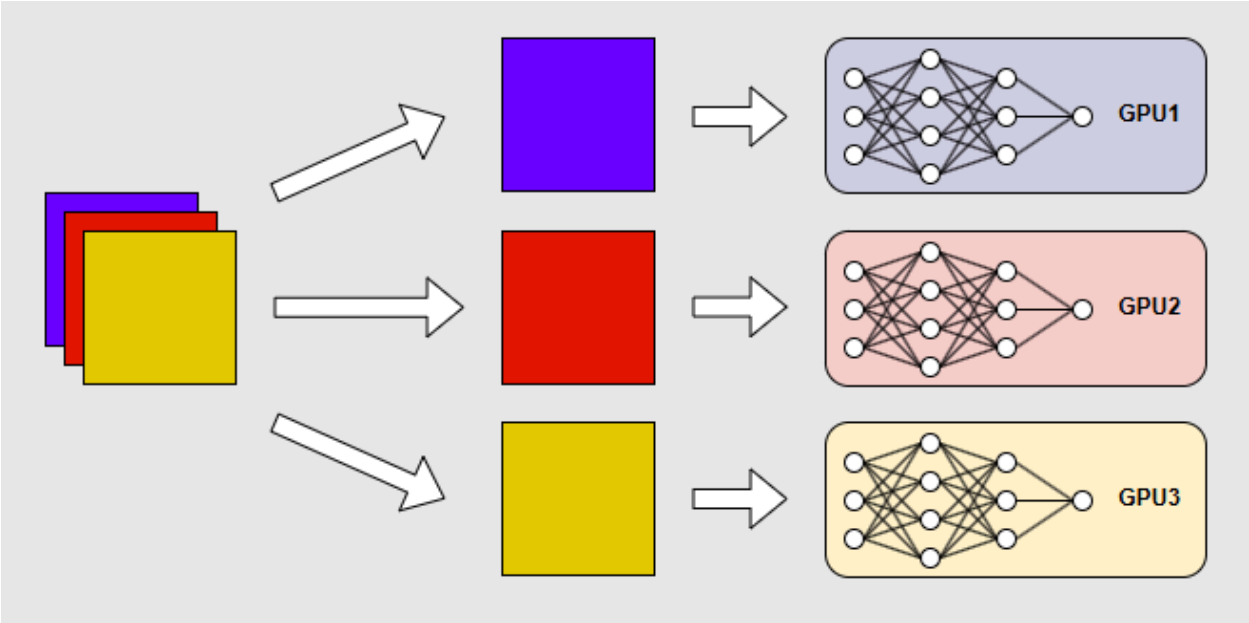
1. We start by initializing some base model as specified by the user and setting up the data loading architecture.
2. Creating n copies of the base model to be distributed over the n peers (i.e. partitions). We also create n partitions of the data which are distributed over the n peers.
3. Each of the n peers (workers and leaders) are initialized with the data and the models. This is loading the model in memory and locating the data locally before the training begins

5.4 Distributed Model Training Workflow



1. We ensure that each of the peers are initialized (the workers and leaders) with their data and model. We make sure the model is loaded in memory and that the data is loaded with the data loader.
2. We begin the asynchronous training cycle. The workers are doing the majority of the training (iterating over the data for the specified number of epochs).
3. After each worker peer has been training for a predetermined number of iterations, we send it to the leader peer for averaging.
4. The leader peer begins averaging after it receives at least one aggregation cycle. We are doing parameter based averaging.
5. The worker peers receive the result of the averaging and update their local model and continue the training cycle.

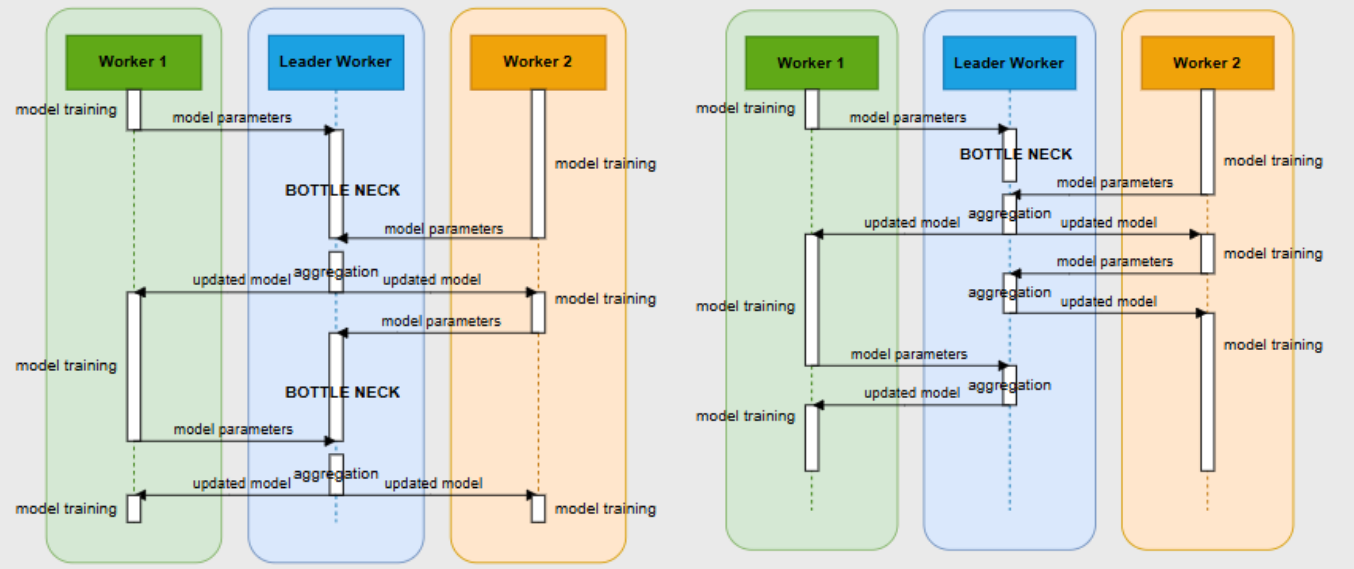
5.5. Data Parallelism



Data parallelism is a technique that involves splitting a large task into smaller, independent subtasks that are processed simultaneously. In the context of our project, each provider peer is responsible for training using a subset of the dataset. While training, the model parameters are periodically sent to the leader provider peer for aggregation to ensure the models from each provider peer don’t diverge. Specifics about the aggregation process can be found in 6.1.

6. Detailed Design

6.1. Sync vs Async Model Aggregation



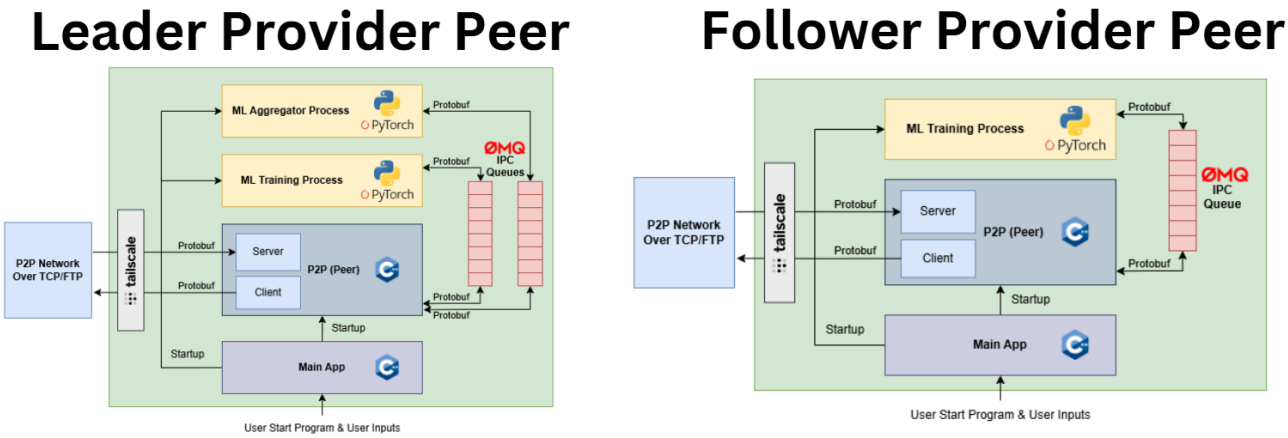
Synchronous Aggregation

Asynchronous Aggregation

As mentioned in 5.5, model parameters need to be periodically aggregated to avoid model divergence. To avoid the bottleneck of waiting for every worker to reach the same point of training (i.e. finishing an epoch), model aggregation is done asynchronously. Once a worker is finished training a certain workload (i.e. one epoch, 64 batches, configurable), known as an aggregation cycle, it sends its model to the leader worker to be aggregated. The leader aggregates using the latest model from each worker and returns the updated model, allowing the worker to continue training immediately.

Synchronous aggregation (left diagram) has a barrier waiting for all workers at the end of every aggregation cycle. This causes all workers to bottleneck on the slowest worker often. On the other hand, asynchronous aggregation (right diagram) only has one barrier at the first aggregation cycle. Afterwards, a worker never has to wait on any other worker. Although this results in a higher number of aggregations, and workers training on potentially outdated models, we found that the model accuracy remained largely unchanged with frequent aggregation, while training speeds reduced significantly.

6.2. Integrating the P2P Architecture with the ML Training Processes



When a job is sent out from the requester, all provider peers launch an ML training process, while the leader additionally launches an aggregator process. Interprocess communication (IPC) between the processes is performed using ZeroMQ and Protobufs.

6.2.1. Protobufs

We have created 1 base proto message type called PayloadMessage, which contains one of 8 payload objects, with the following signature:

```
message PayloadMessage {
    string id = 1;
    string senderId = 2;
    utility.IpAddress senderIpAddress = 3;
    PayloadType payloadType = 4;

    oneof payloadObject {
        DiscoveryRequest discoveryRequest = 6;
        DiscoveryResponse discoveryResponse = 7;
        TaskRequest taskRequest = 8;
        TaskResponse taskResponse = 9;
        Acknowledgement acknowledgement = 10;
        Registration registration = 11;
        RegistrationResponse registrationResponse = 12;
        ModelStateDictParams modelStateDictParams = 13;
    }
}
```

DiscoveryRequest and DiscoveryResponse are used by all peers to communicate with the bootstrap node. They facilitate peer discovery by specifying the number of peers requested and providing a list of available peers in response.

Registration and RegistrationResponse are used by all peers to communicate with the bootstrap node. These messages handle peer registration, allowing new nodes to join the CloudMesh network.

TaskRequest is sent from a requester peer to multiple provider peers and initiates the training workflow. This proto contains metadata about the job, such as the number of assigned workers, the leader UUID, the IP addresses of the assigned workers, the number of epochs, and the glob pattern information used for FTP-based file distribution.

TaskResponse contains the serialized model state dictionary in byte format and a boolean flag that indicates if a job is completed. This is sent from the provider peers back to the requester pee upon job completion.

ModelStateDictParams is used to transfer the model's state dictionary from the ML process to the P2P architecture. This wraps the output of a pickle.dump() as a bytes array and is exchanged between follower and leader peers during the model training process.

Together, these protobuf messages work together to assist in facilitating standardized communication between all the peers.

6.2.2. ZeroMQ

We utilize the ZeroMQ queueing system to perform inter-process communication. The system operates in a publisher-consumer manner, where processes can publish protobuf messages to the queue, and other processes can consume these messages as needed. Both the ML process and the P2P process act as publishers and consumers, enabling bidirectional communication and message exchange through the shared queue.

6.3. FTP

To facilitate the transfer of large training data sets required for distributed machine learning tasks, we implemented a simplified version of the File Transfer Protocol (FTP).

Each peer within the CloudMesh network runs a lightweight FTP server responsible for "hosting" and distributing data files. When a requester peer initiates a training task, it leverages FTP to efficiently serve training data files to each provider peer assigned to the task. Provider peers subsequently download the datasets directly via FTP. This process ensures a robust mechanism for handling large file sizes typical in ML workflows while maintaining simplicity and compatibility across diverse network conditions.

6.4. TailScale

TailScale is integrated into CloudMesh to manage peer-to-peer connections across devices behind NATs and firewalls. Specifically, TailScale establishes secure, private, peer-to-peer VPN tunnels between peers, overcoming common network obstacles such as NAT traversal and firewall restrictions through automatic hole-punching in a private subnet. By abstracting away complex network configurations, TailScale allows peers running on different networks or geographical locations to discover and securely communicate with one another.

User Manual and Deployment Information

Our program is mainly built for Mac and Linux systems, and as such, the instructions will follow suit. The program is compatible with Windows systems, but following the instructions below will not work due to system differences. Detailed instructions will not be available on how to install the dependencies or how to run on Windows.

7.1. Dependency Installation

Bazel

- Bazel is used to compile our C++ code.
- Instructions to install can be found here: https://bazel.build/install

Tailscale

- Tailscale is used to set up a P2P vpn network to connect the machines.
- Instructions to install can be found here: https://tailscale.com/kb/1347/installation

ZMQ

- Run the following terminal commands in the root directory
- To install libzmq:

```
cd third_party
git clone https://github.com/zeromq/libzmq
cd libzmq
mkdir build
cd build
sudo cmake .. -DENABLE_DRAFT=ON -DENABLE_CURVE=OFF -DENABLE_WSS=OFF
sudo make install
```

- To install cppzmq:

```
cd third_party
git clone https://github.com/zeromq/cppzmq
cd cppzmq
mkdir build
cd build
sudo cmake ..
sudo make install
```

7.2. Code Compilation

Before compiling the code, first figure out the IP address of the bootstrap node you wish to use.

Then run the following command, replacing {BOOTSTRAP_IP_ADDRESS} with the correct IP.

```
export BOOTSTRAP_HOST={BOOTSTRAP_IP_ADDRESS}
```

To build the code, we use bazel.

Linux

```
bazel build //... --experimental_google_legacy_api
```

MacOS

```
bazel build //... --experimental_google_legacy_api --config=macos
```

7.3. User Manual

This is a quick-start guide to get a simple demo working on multiple computers. This guide will require 4 computers, which play the role of 1 provider, 2 requesters, and 1 bootstrap node.

Clone the CloudMesh repository from here: https://github.com/DCP-CloudMesh/CloudMesh

This repository already contains the CIFAR10 dataset that will be transmitted from the requester node to the provider nodes.

Bootstrap Node (Computer 1)

- Start the bootstrap service - `./bazel-bin/bootstrap`

Set the environment variable for the BOOTSTRAP_HOST to Computer 1's TailScale IP for all nodes

First Provider Node (Computer 2)

- Initialize a *Provider Node* by typing `./bazel-bin/provider -p 8081` into the terminal
    a. This will launch the provider on port 8081
    b. This initialization will open up 4 ZMQ ports, which will be listed in the output
        i. 2 to communicate with the ML process
        ii. 2 to communicate with the aggregator process
- Start the ML process - `python3 ml/main_simpleCNN.py --port_rec <port> –port_send <port>`
    a. Enter the ports from step 1 that pertain to the ML process
- Run `python3 ml/aggregator.py --port_rec <port> –port_send <port> --num_peers <num_peers>`
    a. Enter the ports from step 1 that pertain to the aggregator process
    b. Enter the number of provider peers that are being used in this test training job. In this quick start scenario, that would be 2 peers

Second Provider Node (Computer 3)

- Repeat exactly the same steps as in Section 2, plugging in the new ZeroMQ ports when running the ML and aggregator scripts.

Once all the above steps are completed, we can begin the requester workflow.

Requester Node (Computer 4)

- Initialize a *Requester Node* by typing `./bazel-bin/requester -w <num_workers> -e <num_epochs> -p <port> -m <mode>`
    a. Num_workers stands for how many provider peers are desired for the current training job. Set this to 2 for our case
    b. Num_epochs stands for how many epochs we would like the system to run for. Set this value to 10
    c. Port can be set to 8082, as this will open up the requester on port 8082
    d. Mode can be either compute (c) or request (r). Set this to c
- Now we wait until the 10 epochs are completed
- Finally, we can run `./bazel-bin/requester -p 8082 -m r` to receive the results

8. Deployment Information

Deployment of CloudMesh requires the following components and considerations:

- **Bootstrap Node Deployment:**
    - Deploy the bootstrap node locally or on a cloud provider with a static, publicly accessible IP address.
    - Ensure the port (ie: 8080) is open and accessible through firewall configurations.
    - Environment variables `BOOTSTRAP_HOST` and optionally `BOOTSTRAP_PORT` must be configured on all peers.
- **Provider and Requester Deployment:**
    - Peers must install all dependencies, including ZeroMQ and TailScale, as described in the dependencies section in the project README.md.
    - Ensure all peer machines have sufficient computational resources (RAM, CPU cores, GPU as required) for intended ML tasks.
    - Set up and authenticate TailScale on each peer to establish secure VPN tunnels and handle NAT traversal automatically.
    - Providers should execute the `aggregator` and `training` processes in addition to the
- **Network Configuration:**
    - Verify correct TailScale configurations for peer discovery and secure communication (use `ping` or similar).
    - Regularly test and maintain the VPN tunnels to ensure consistent peer-to-peer connectivity.

For local development, testing and deployment, see DCP-CloudMesh/CloudMesh.

Verification and Validation

9. Results

The verification and validation consist of a variety of different experiments. Firstly, the random shuffle experiment - which consists of partially-uniform random partitions of the data, similar to k-folds cross validation, and evaluating the training pipeline over all data distributions. Secondly, the baseline, normal pytorch training loop experiment to understand if the model architecture works well for the problem domain. Then a centralized federated learning experiment, which simulated the distributed training and model aggregation without the decentralization. Finally, concluded by our CloudMesh metrics.

For the metrics, we will consider a variety of different values. Primarily,

1. Loss - measure of the difference between predicted and actual values
2. Accuracy - measure of how often a machine learning model predicts the correct value
3. F1 Score - an accuracy metric that takes into account Precision and Recall
4. Precision - measure of how often a machine learning model correctly predicts the positive class
5. Recall - measure of how often a model correctly identifies positive instances from all actual positive samples
6. Confusion Matrix - an n-dimension representation of the predicted vs actual class of a sample
7. Time - measure of how long a model takes to train end to end

Apart from just evaluating metrics, we have considered evaluating the ground truth vs the inference results. By looking at the images and their labels, we can qualitatively analyze the inference results in comparison to the baseline labels.

9.1. Random Shuffle Experiment

9.1.1. CIFAR10 Dataset

The CIFAR10 Dataset is a collection of 60,000 32x32 coloured images, evenly split across 10 classes. The classes being very distinct, as follows:

1. Airplane
2. Automobile
3. Bird
4. Cat
5. Deer
6. Dog
7. Frog

8. Horse
9. Ship
10. Truck

We are training a specialized convolutional neural network architecture with 3 Conv2dLayers, 1 max pool layer, and 2 fully connected layers. This can be defined as the following

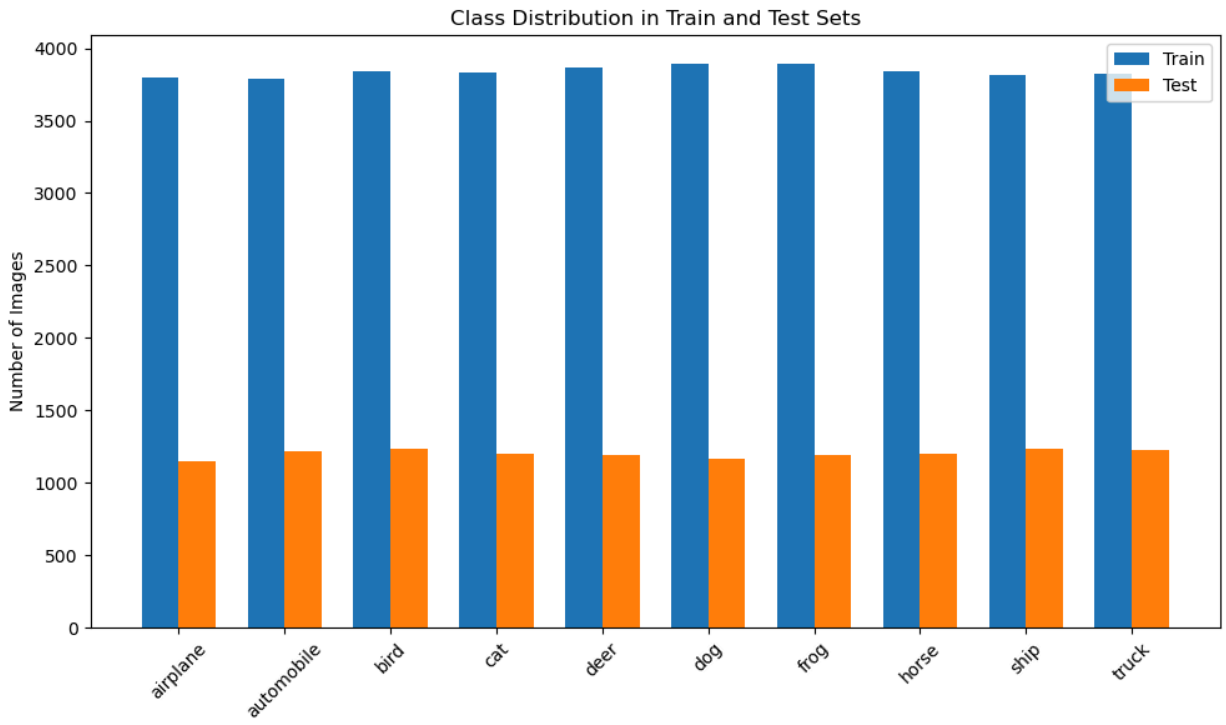
```python
class SimpleCNN(nn.Module):
    def __init__(self, output_dim=10):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.fc2 = nn.Linear(512, output_dim)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = self.pool(torch.relu(self.conv3(x)))
        x = x.view(-1, 128 * 4 * 4)  # Flatten the tensor
        x = torch.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)
```

SimpleCNN Model Architecture

We specifically use a CNN architecture because we are dealing with an image classification problem. Taking an image as an input and producing a class (categorical) output. The CNN architecture is able to take spatial information from the image input and process the information, which is fed into the fully connected linear layers. The linear layer is the neural network which is able to understand the information from the CNN and determine the output class.

9.1.2. Random Shuffle (RS) Experiment

When looking at the CIFAR10 dataset, we note that it is divided into one partition of 50,000 train images and 10,000 test images. Both of the partitions are uniformly distributed among all 10 classes. This produced questions about the robustness of our training pipeline. In particular, if we are able to train our model and produce accurate results for an unbalanced dataset. We Shuffled the training and testing data together and created brand new partitions with non-uniform distributions and split them into a train and test set. We can see the distribution for 1 of the 10 experiments below.

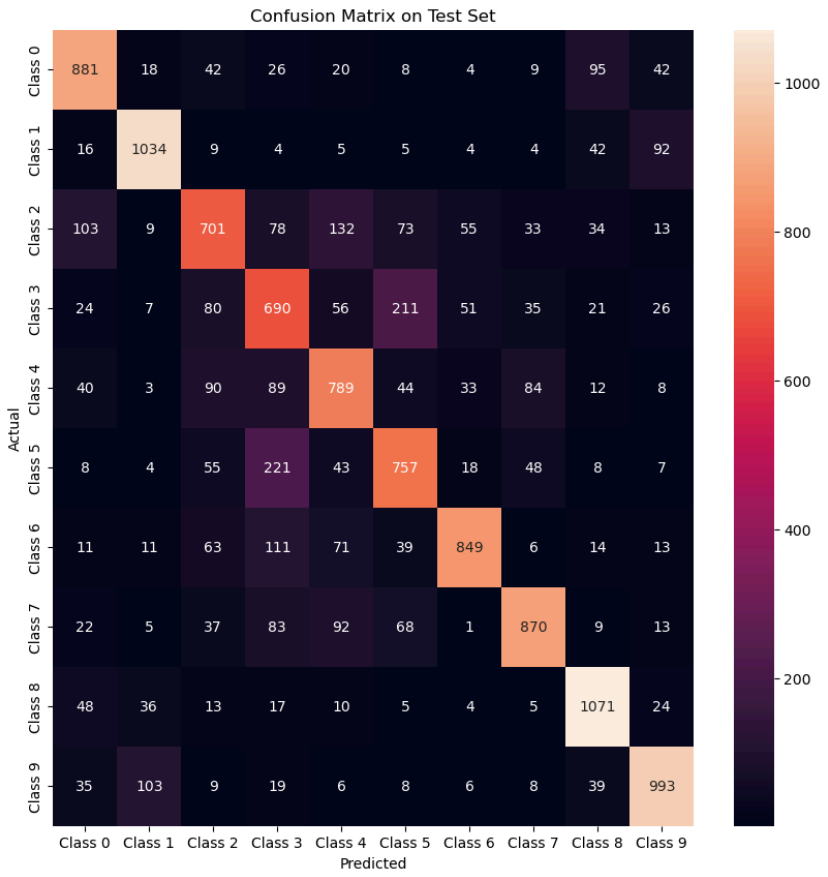


Distribution for Experiment 10 in RS Experiment

We can see that the class distribution is relatively equal, but completely different than the original dataset. It is also imbalanced.

For the final metrics of Experiment 10, we see the following

Metric	Value
Loss	0.012489
Accuracy	71.958%
Precision	0.722261
Recall	0.719212
F1 Score	0.719603

Confusion Matrix	

In comparison to the metrics listed below, we see that we maintain very similar metrics. In terms of accuracy and F1 Score, the model training pipeline is robust to changes in the dataset. We are able to handle these very common edge cases. Thus showing that the distribution of the dataset is not a limitation of our training aggregation technique.

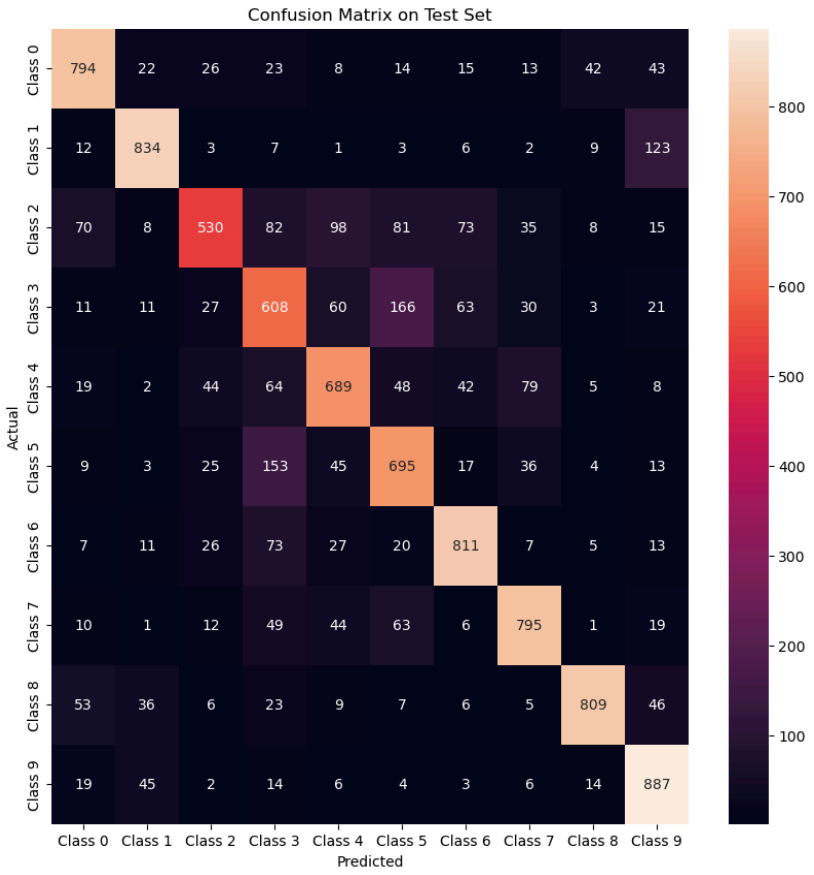
We conduct 10 experiments overall which are listed in the DistributedML repo in the directory linked: [random_shuffle_distributed]. From the 10 experiments, we see the various types of CIFAR10 training / testing distribution and we note that the metrics for all of the experiments are similar with a relatively small standard deviation. The metrics are producing values in an expected range.

Thus, we can say that our machine learning training pipeline and the parameter based aggregation technique that we are using is robust to variations in the dataset. We are able to handle imbalanced datasets and unique training situations and produce very accurate metrics.

9.2. Baseline (Normal PyTorch Training Loop) Experiment

When creating CloudMesh, we wanted to have baseline metrics from a non-distributed, single machine PyTorch training loop. We want to train a simple CNN model using normal pytorch techniques. We are using the standard CIFAR10 data distribution.

These are the best metrics we produced for the baseline training process over 10 epochs of training.

Metric	Value
Loss	0.0135166
Accuracy	74.52%
Precision	0.7500629
Recall	0.7452
F1 Score	0.7444758
Confusion Matrix	
Time	10.3 minutes

The notable metrics in these experiments are the 74.52% accuracy and the 0.7445 F1 Score and 10.3 mins for time. These are the benchmarks we aim to simulate in the Centralized Federated Learning Experiment and what we hope to achieve in the CloudMesh final pipeline.

These metrics can be found here: simpleCNN_10ep_002.

9.3. Centralized Federated Learning Experiment

The centralized federated learning experiment was the first step towards the final CloudMesh product. Essentially, CloudMesh is a huge distributed data parallelism problem. While we were building out the decentralized distributed network in C++, we needed a simulation to see if the machine learning logic works and produces correct results. We needed the results to be accurate and efficient. Thus, introducing the centralized federated learning experiment.

All of the code for this experiment can be seen here: main_distributed_iteration_pub_sub.

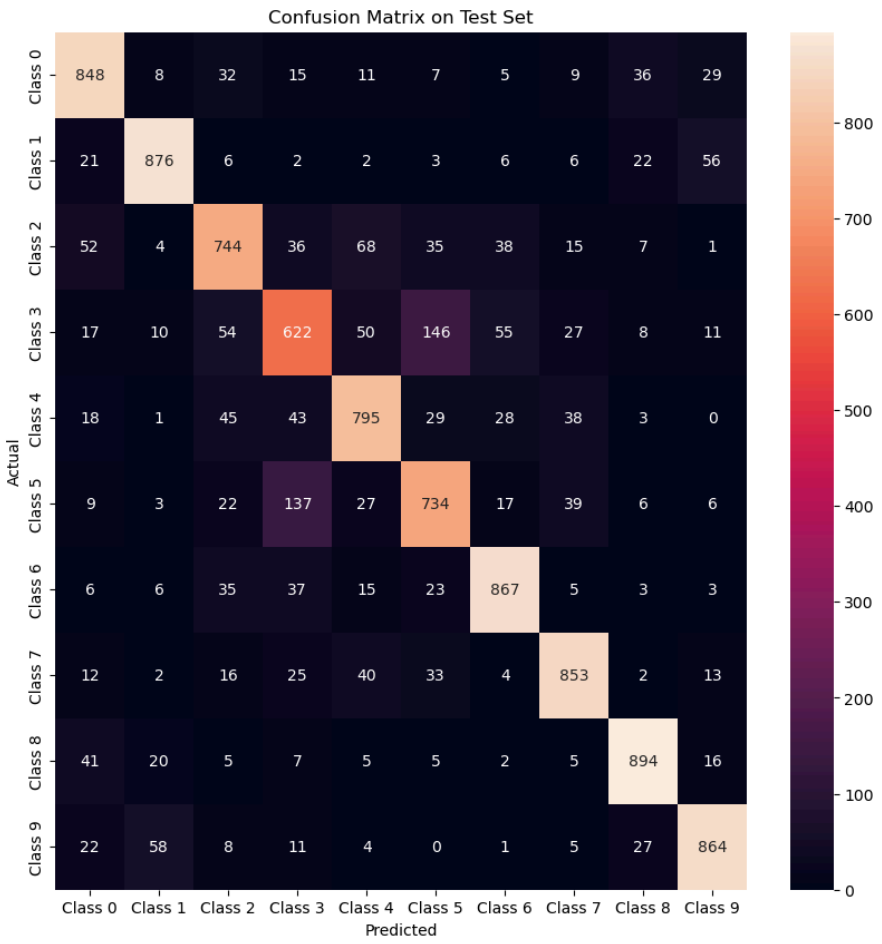
Essentially, we are using multiprocessing to simulate these fake peers. The multiprocessing library in python has this concept of a Manager. Essentially, it is able to communicate with all of the peers and do some shared workload. This manager was essentially our leader peer, where we built out the aggregation of our pipeline. Specifically, we experimented with parameter and gradient based averaging in the ModulPublisher for the Manager. We settled with parameter based averaging which essentially is doing a torch stack over all of the parameters and then taking the mean over all of the models parameters.

For the multiprocessing itself, we spin up training processes which communicate centrally to this manager. Each process is given a copy of the base model and a partition of its dataset (for data parallelism). Since each process is responsible for their own dataset partition, we are essentially doing a centralized form of federated learning.

With this simulation of the final CloudMesh pipeline, we can determine the performance of the final pipeline and tweak it early. This allowed us to experiment early and experiment a lot to get the best ML results possible. Especially because building a fast, efficient and scalable Peer to Peer Network in C++ takes quite a bit of time.

These are the best metrics we produced for the centralized federated learning experiment over 10 epochs of training.

Metric	Value
Loss	0.0125613
Accuracy	80.97%
Precision	0.8087416
Recall	0.8097
F1 Score	0.8090595

Confusion Matrix	
Time	41.6 mins

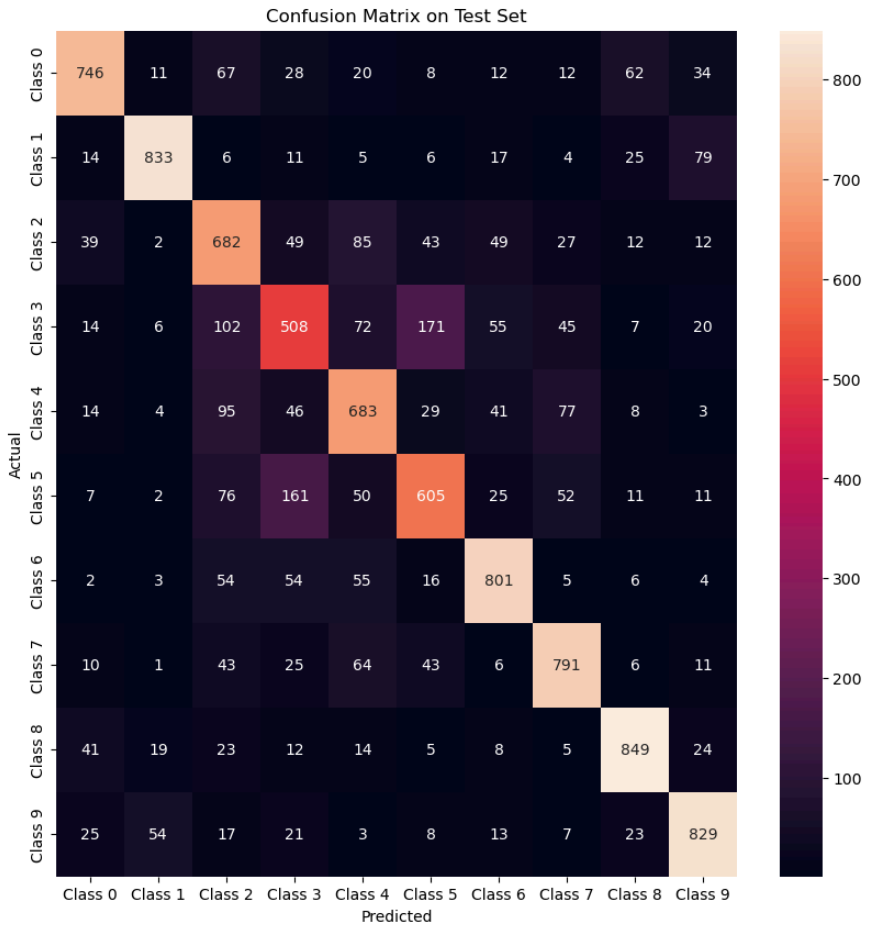
The notable metrics here are the 80.97% accuracy and the 0.8091 F1 score and 41.6 mins for time. These are significantly better than the baseline metrics in the same amount of iterations. However, the pipeline takes a lot more time to run. This would make sense since the multiprocessing library takes a lot more overhead and the pipeline is restricted to compute only on one device.

Thus, we can prove the training technique proposed for CloudMesh produces amazing accuracy metrics in the same amount of work, but takes significantly longer to train. We hope to improve the speed with the distributed system and highly performant C++ networking code.

9.4. CloudMesh Metrics

Finally, we are validating the CloudMesh architecture. We are using the asynchronous, decentralized, distributed training pipeline as described by all of the design documents earlier in this report. More details about the design of our architecture are in sections 5 and 6.

These are the best metrics we produced for the CloudMesh system over 10 epochs of training.

Metric	Value
Loss	0.031591
Accuracy	73.6952%
Precision	0.738710
Recall	0.737379
F1 Score	0.735747
Confusion Matrix	
Time	8.8 mins

The notable metrics to consider here are

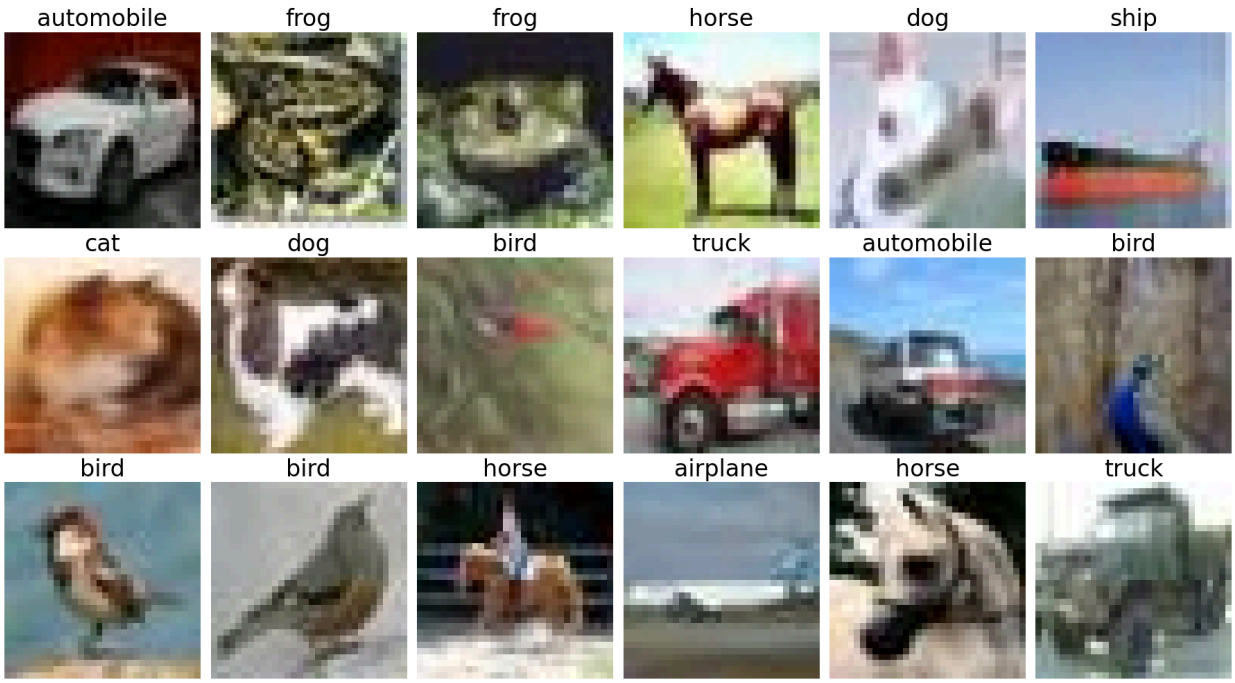
- 73.69% accuracy
- 0.7357 f1 score
- 8.8 mins for time

In terms of comparing these metrics to the baseline, we note that CloudMesh overall performs slightly worse. However, these metrics are still in the expected range for a successful product in our experiments. This loss in performance can be explained because we decided to aggregate only once per epoch as opposed to multiple times per epoch while collecting these results. The F1 score is analyzed very similarly to the accuracy. In terms of time, it is faster than BOTH the baseline pytorch training and the centralized federated learning training.

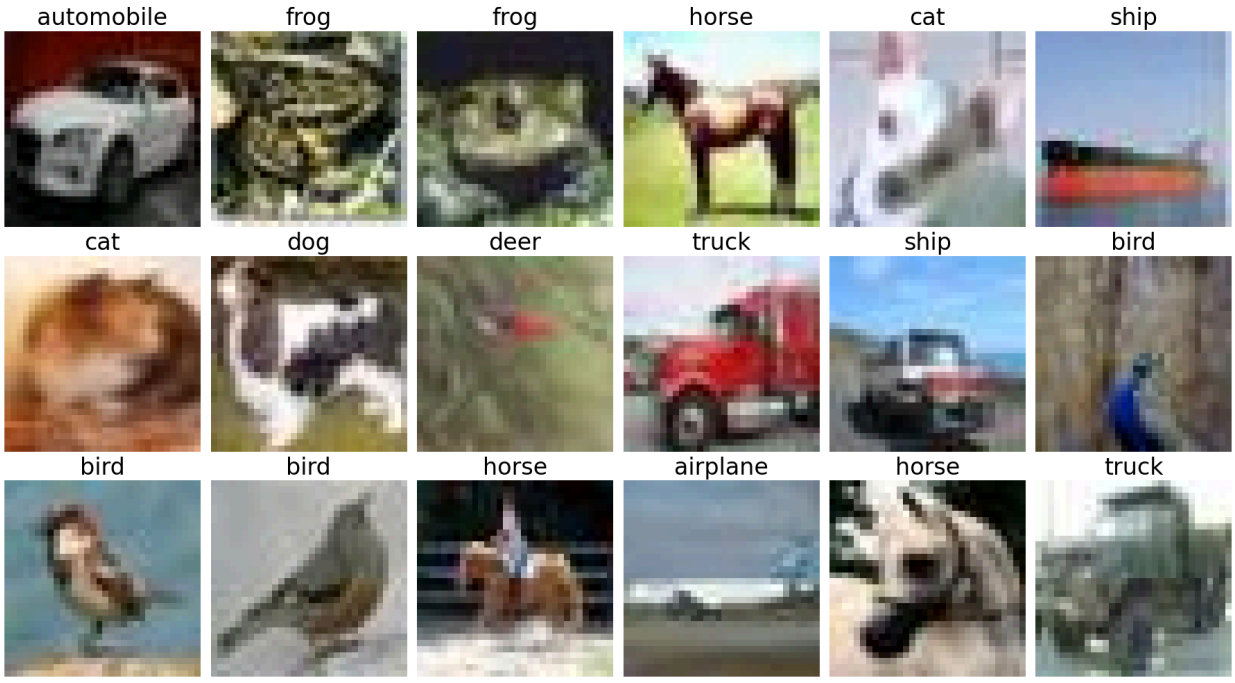
These are very notable results as we produce very similar accuracy within 10 epochs of training and we are faster than both experiments. This opens a lot of doors for people with low personal compute - people are able to training models as they would normally and achieve state of the art metrics and timings.

9.5. Ground Truth vs Inferencing Results

This last analysis is a qualitative comparison between the ground truth labels and the predicted labels. This allows us to see how well our models are actually performing apart from looking at numbers that we produced.



Ground Truth Labels



Inferencing Labels

We can see inference labels are almost the exact same as the ground truth labels. For the predictions that are wrong, we can see that the images themselves are very low quality and very little detail. These are interesting to analyze because we are able to see how the trained end to end model performs on real image input and mapping to the outputs. We can see qualitatively that our pipeline produces very accurate and performant models.

10. Glossary of Terms

1. **Epoch** - A cycle of training where a model iterates over all of the data once. An epoch consists of multiple iterations.
2. **Iterations** - A cycle where the model runs inferencing, calculates the loss, does gradient updates and backpropagation.
3. **Peer** - A member of the P2P network. This can be either a requester peer or a provider peer.
4. **Bootstrap Node** - Entry Point into network with a static IP address that peers connect to in the first stage.
5. **Training Task** - The packaged workload that includes the model to train, partitioned training data, training configurations, assigned provider peers, and additional metadata.
6. **Provider Peer** - The peer that provides compute resources to complete ML training task.
7. **Leader Provider** - A specialized provider peer that has additional responsibilities of averaging and broadcasting updated gradients and parameters among assigned provider peer group for a given training task.
8. **Follower Provider** - A simple provider peer that trains the model based on the data it is assigned. It sends this data back to the leader peer after every epoch.
9. **Requester Peer** - The peer that requests to train its model based on its training data.
10. **P2P Network** - The network which holds the requester nodes, the bootstrap nodes, and provider nodes. The bootstrap node exposes the network to the public and holds the network together, as it holds information of all provider peers and requester peers in its region.
11. **Hyperparameter** - some user defined parameter used for loss functions, training loops, optimization, and other parts of ML training.
12. **Gradient Descent** - parameter update technique used to modify the weights during a training iteration.
13. **Batch Size** - the number of samples used in an iteration for parameter updates
14. **Learning Rate** - the rate of change to the parameters based on the gradient update calculations.
15. **Overfitting** - When a model is too complex and memorizes the training data and has a high variance.
16. **Underfitting** - When a model is not complex enough and is not representative of the training data and has high bias.
17. **Distributed Training** - When the training is performed across multiple computers and is combined at the end.
18. **Federated Training** - One centralized peer that performs aggregation based on the results from all of the other nodes.
19. **Gradient Averaging -** A process where the leader peer computes an average of the gradients received from follower peers to update the global model.
20. **Cluster -** A group of interconnected peers in the P2P network, determined by geographic or network proximity.
21. **Ghost Nodes -** Residual entries in the bootstrap node's mapping table representing peers that have disconnected without being deleted from the mapping table.

22. **Partition** - A subset of the training data distributed to a provider peer during distributed training.
23. **Model Aggregation** - the process of combining updated model parameters or gradients from multiple peers to produce a global model.
24. **Asynchronous Training** - a training paradigm where all peers perform computations independently, and updates are aggregated at irregular intervals.
25. **Synchronous Training** - a training paradigm where all peers complete a computation step before aggregating updates and proceeding to the next step.
26. **Latency** - the delay experience in a network during data transmissions between peers.
27. **Backpropagation** - a process in a neural network where gradients are calculated and propagated backward to update model weights.
28. **Inference** - the process of making predictions using a trained model.
29. **Data Loader** - a component responsible for loading and preprocessing data before it is used in training iterations or validation / testing.