P2PFL: Peer to Peer Federated Learning System

Matt Laws
Williams College
mdl4@williams.edu

Nathan Vosburg
Williams College
njv1@williams.edu

Abstract

Federated learning (FL) has emerged as a promising approach to enable machine learning (ML) training on distributed, privacy-sensitive datasets without requiring centralized data collection. However, traditional client-server FL systems often rely on costly centralized infrastructure, raising concerns about accessibility, scalability, and single points of failure. In this work, we propose P2PFL, a peer-to-peer federated learning system that leverages decentralized networks to overcome these challenges. Our system enables peers to train ML models locally using their private data and periodically exchange model updates with a random subset of other peers in the network. This decentralized approach democratizes ML by distributing computation across participants and improves privacy by ensuring that raw data never leaves local devices. Using the MNIST dataset, we evaluate P2PFL on both latency and model accuracy, demonstrating significant improvements over non-federated techniques and achieving accuracy above 90% despite the non-i.i.d. data distribution. We highlight the potential of P2PFL to make ML systems more accessible, secure, and scalable while maintaining high performance.

1 Introduction

In recent years there has been an explosion in the Machine learning (ML) field accompanied by a surplus of new research and startups all focused on developing new ML technology or leveraging it for a specific task. With this growing demand for ML, it is important the underlying systems are able to support such high computation in an efficient and scalable manner. Companies like OpenAI and Athropic have **gigantic** data centers that help them manage all of their computation. However, this is not possible for everyone. It is important to develop systems that allow for easier utilization of these tools to improve the accessibility and democratization of ML.

Another common concern is privacy. Especially with text or image data—data that can often contain sensitive or personal information—, we want to be able to keep our data separate from a larger corpus. That said, we still want all the advantages of the cool new ML model, but if no one is willing to provide data, how do we get it?

Addressing these two concerns we introduce P2PFL, our Peer-to-peer Federated learning system. Our systems allows peers to join the network and receive a model. From there the peer will use their own data to train a localized model and periodically share/recieve their model with a random subset of its peers. This methodology allows for us to distribute the training cost across machines and create an aggregate model that outperforms each individual.

2 Background

2.1 Fully Connected Neural Networks

A fully connected neural network (NN) is a network that contains an input layer, 1 or more hidden layers, and an output layer. The input layer contains one node per feature of the input. In the case of multi-class classification, the output layer has a node for each of the potential classes that represents the probability that a given input belongs to that class. A NN can be represented as a directed acyclic graph[1] where nodes in layer L_i can only have parents from layer L_{i-1}. In a fully connected NN L_i has all nodes in L_{i-1} as parents. Each edge represents a learnable weight that forms a weight matrix θ that defines the values of the nodes of layer L_i as a function of the previous layer L_{i-1} by the following formula:

$$L_i = \text{Activation}(L_{i-1}[\theta]^{L_{i-1} \to L_i})$$

[1] a directed acyclic graph is defined as a graph G where G is simple, contains only directed edges, and there are no directed cycles

Where we use ReLU as our Activation and $[\theta]^{L_{i-1} \to L_i}$ is the vector of weights between layers L_{i-1} and L_i. The goal of NNs is to learn a vector θ that contains all $[\theta]^{L_{i-1} \to L_i}$ such that given some input X, after applying the weights of the NN the distance of the prediction ($\hat{Y}$) to the true value (Y) is minimized.

2.2 Convolutional Neural Network

A convolutional neural network (CNN) is a network that uses convolutional layers to augment fully connected layers. CNNs are effective at learning features from the data using locality. Overall CNNs use a combination of learned parameters to obtain a prediction vector $\hat{Y}$ given some input input X that estimates the true value Y. A convolutional layers relies on the convolution function to compute weights. The convolution of functions f and g is defined by the following:

For each $X_{i:i+k,j:j+k}$ in a image X defined as a $n \times m$ vector of pixels with 1-indexed domain: $1 \leq i \leq n-3; 1 \leq j \leq m-3$. A $k \times k$ filter is applied as seen below:

$$c_{p,q} = \sum_{i=1}^{k} \sum_{j=1}^{k} w_{i,j} \cdot x_{i+p,j+q} + \text{bias}$$

here (p, q) is the coordinate of the filter's top left corner and the weights ($w_{i,j}$) are learned through training the network. The output of a convolutional layer – which is also a two dimensional vector – is an aggregation of all the $c_{p,q}$ such that the feature at position (p, q) in the output layer is set to $c_{p,q}$.

An important feature of CNNs is that each feature in the new layer only sees a portion of the input, namely the size of the filter. By aggregating smaller sections of the image together into a single node, the network is able to learn features in a broader space than a single pixel. These learned features are then passed to a standard NN to complete the learning.

2.3 Federated Learning

Federated learning (FL) is a technique for training ML models on local resources with separate datasets without the need for information to be exchanged between them. FL is particularly motivated when each resource corresponds to a device. Consider an autocorrect model as an example. In this case, each client (or user) is sent an initial model by the server, then periodically the clients receive a request from the server for the model accuracy. In a perfect world, each client receives the request, sends back its gradients, and the central server retrains and pushes an updated model back to the clients see figure 1 for diagram.

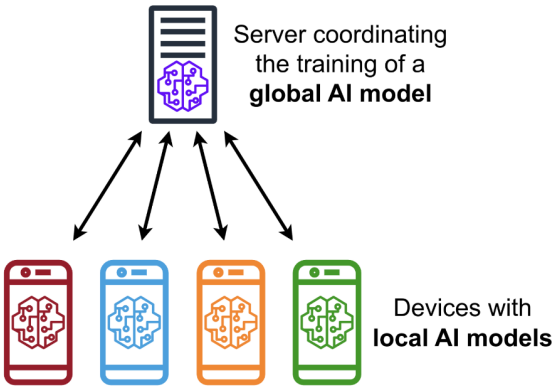

Figure 1: Diagram of Client-Server Federated Learning

2.4 Peer-to-Peer Networks

Standard FL uses a client-server model for its training; however, this still requires a centralized company (or some entity) maintaining and paying for servers. Peer-to-peer (P2P) networks are decentralized systems in which every participating node, or "peer," acts as both a client and a server. This allows them to share resources and responsibilities without relying on a central authority. Unlike standard client-server architectures, P2P networks distribute the workload among all participants, creating a more resilient and scalable infrastructure. This also allows them to utilize the compute of many devices to aid in a single task. Peers in the network communicate directly with one another, allowing for efficient transfer of information. P2P networks are also highly fault-tolerant due to their decentralized nature, as there is no single point of failure. Applying a P2P methodology to FL allows us to achieve both our goals of democratization and information security.

3 Data

The MNIST (Modified National Institute of Standards and Technology) dataset is a benchmark dataset widely used in machine learning and computer vision for image classification tasks. It consists of grayscale images of handwritten digits ranging from 0 to 9, each represented as a 28x28 pixel grid. MNIST provides 60,000 examples and we

use a 80-20 train dev split to divide our data. From their we partition the data between 10 different peers allowing each to have independent data. Notably while we partition the data, we only allow each client to see 7 of the 10 numbers (thus why we rarely see accuracy lower than 70%. This **non-i.i.d.** partition allows us to test the functionality of federated learning—without sharing data, it should be nearly impossible to achieve significantly higher than a 70% accuracy on the dev data because we have never seen 3 of the 10 numbers being tested. Thus if we demonstrate accuracy over 70% we can show that the methodology works. Due to the extreme non-i.i.d. nature of the data, it is unlikely that the models will every perfectly converge to a near 100% accuracy, but we hope to show significant improvement.

We chose to use images as our data because they well represent potentially private data. If some company, say Apple, wanted to train a model to help better sort our pictures—that would be great, but we wouldn't want all of our photos to be sent to a centralized server. Hence why image classification makes a good FL task.

4 Design

Our design ships a python and a Go script along with a user interface (UI) for running the code see figure 3. Our UI allows a new peer to specify which port they want to open a connection over, the peer they are going to use to boot into the system and a model that they want to submit for training, they can also opt to purely receive the model and then use their data to train. Then python and Go scripts (which are running on the same device) make use of the file system for communication. Upon joining the network, a folder maintaining the state of the peer in created. A sketch of our design with 4 peers is shown in figure 2.

4.1 Python Design

Our python scripts are designed to run indefinitely and continue training until terminated. The python clients handle all of the FL leaning work and the flow of the program. The client also maintains a timer for when to send an aggregation request. When the timer triggers (after completing a training cycle) a request is sent to the go client to ask its peers for their models. The models are then placed in a "aggregation" folder in the state directory, and the models are aggregated. In order to improve privacy we don't send a full model file we only send the state dict which only posses the weights. Thus without knowing the base model there is no way of recovering the model from the data being sent. While this is not a perfect solution it gives us some level of protection.

4.2 Go Design

The Go client's design centers around providing a robust networking layer for our P2PFL system while keeping the Python client focused solely on model training. At a high level, the client is designed to handle peer discovery, maintain network connections, and facilitate model transfers between peers. The client is designed to maintain an up-to-date list of active peers while providing mechanisms for both serving its local model and requesting models from other peers for aggregation.

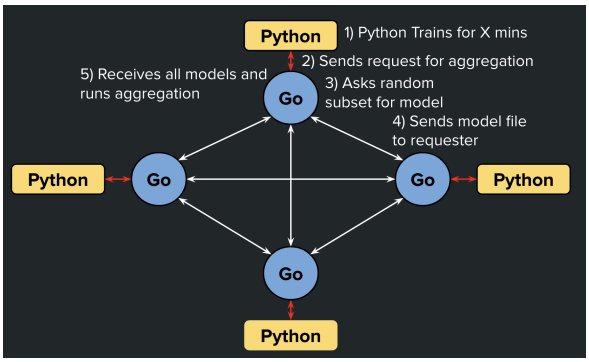

Figure 2: Sketch of our design in a network with 4 peers. All peers know about other peers but all peers do not interact with all peers to aggregate thus allowing for scalability.

5 Implementation

5.1 Python Implementation

For the implementation of our python script we sought to use generalizable methods as opposed to specialized HuggingFace models so that users could easily interchange what models could be trained across the system. Thus we use torch to handle model definition and training. For training on our data we use the Cross Entropy Loss as our criterion and a CNN convolutional neural network as our network, with two convolutional layers with pooling and two fully connected layers.

5.2 Go Implementation

When a new peer boots into the system, it first establishes a connection with any other node and

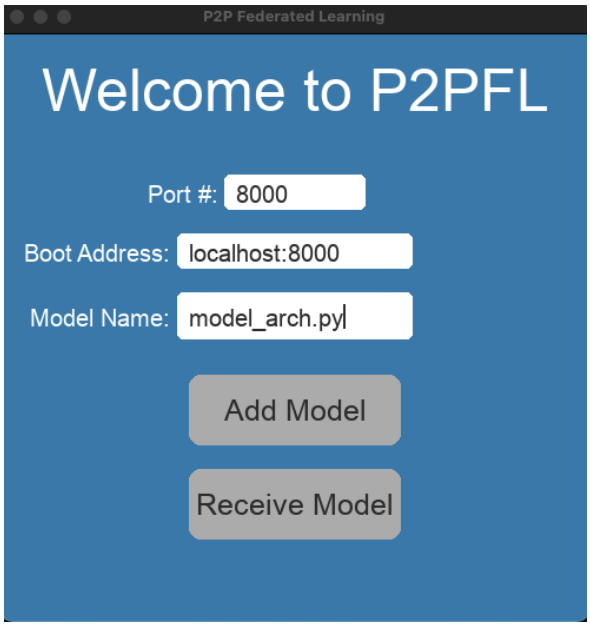

Figure 3: UI for new peers connecting to the network allows users to add a new model architecture to train or just receive exisiting models.

requests a list of active peers in the network. This initial connection is crucial for joining the P2P network, as it provides the new peer with knowledge of other participants. The boot node shares its peer list, which contains the IP addresses and ports of other active peers in the network. This peer list is maintained in a thread-safe map structure that gets dynamically updated as new peers join or leave the network, ensuring concurrent access is handled safely. The client implements a gRPC streaming server that handles two primary types of requests: serving the local model to other peers and collecting models from peers for aggregation. When serving its model, the client streams the model file in chunks to requesting peers, allowing for efficient transfer of potentially large model files. For model aggregation, the client randomly selects a subset of peers from its known peer list and requests their models in parallel. This random selection helps distribute network load and ensures the system remains resilient even if some peers become unavailable. Furthermore, it provides key benefits for the machine learning process - by randomly selecting different peers each time, we prevent overfitting to any particular peer's local dataset and introduce helpful stochasticity to the optimization process.

As new peers join the network, they automatically become available for model sharing and aggregation. This allows the network to grow organically while maintaining its distributed nature. Each peer maintains its own view of the network through its peer list, and no single peer is critical to the network's operation. This decentralized structure, combined with the random peer selection for aggregation, ensures the system can continue functioning even as peers join, leave, or temporarily disconnect from the network.

5.3 Integration

To communicate between the two clients, we use gRPC, a modern RPC framework that allows us to make calls from the python scripts to run Go functions. To pass data between the python and Go clients we use the shared file systems; however it is important to make sure that since we are reading and writing files we properly lock and unlock. Since it is difficult to maintain a system level lock across separate processes, we use a lock file approach. The way it works is simple and we will explain via an example. When first joining the network, the Go client needs to boot and run some setup commands before starting the python script, to enforce this ordering the python script immediately busy waits until it sees a ".BOOT_DONE" file written to its state data. The python script checks every second for 15 seconds and if it sees the script it removes it and continues to run its code. If it doesn't see the script in 15 seconds we assume the boot failed for some reason and timeout. The Go client on the other hand simply runs its boot script and at the very end writes the file. This locking method is also used for waiting on the aggregation. With gRPC, the shared file systems, and locking we now have ways for the two halves to communicate and share data.

6 Evaluation

We evaluated P2PFL on several different metrics but mainly focus on latency and accuracy. To measure accuracy—and to run a full test of the system, we ran our script on 10 different 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz with 8 CPUs (lab computers). For efficiency we did not use the UI to run, instead calling the run script directly across many different ssh connections—see figure4

6.1 Federated Learning Results

We tested with a shared CNN model as described in section 5.1 and the data described in sec 3. We ran

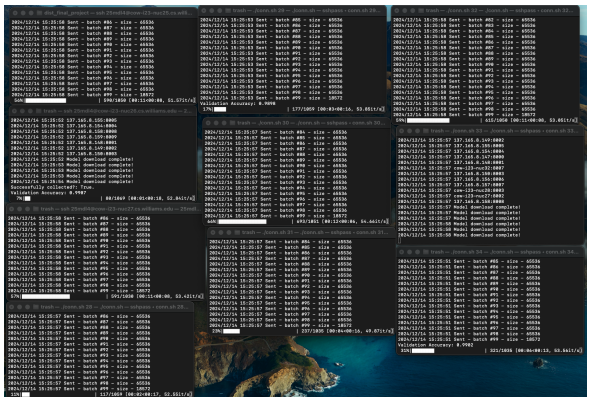

Figure 4: Testing setup:ssh across 10 lab computers.

our tests for approximately 20 minutes, introducing new peers periodically for about 5 minutes until we reached 10. All peers were booted to the one that connected previous to them. Accuracy results of development accuracy (a set containing all 10 numbers) are shown in figure 5. We remind readers than any accuracy over 70% shows improvement via federated learning. This graph notably shows a few things. The first is that once all the peers have entered into the network we start to see a think band of peers all starting to reach near 100% accuracy, the results remain slightly unstable, occasionally dropping down a bit but overall achieve strong results. After significant training time, all models consistently achieve above 90% accuracy showning a $> 20\%$ improvement over un-federated techniques. We also note than all fit lines for the data are steadily increasing, including the average fit. Finally, the outlier around (100, 0.45) is likely due to a model transfer failing, relatively uncommon (as seen by only one in 10x20mins) and recoverable.

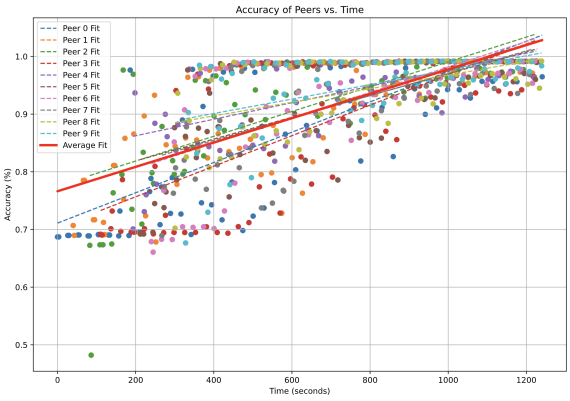

Figure 5: Point validation accuracy of every training instance of each client along with best fit lines and a global best fit line. Note the upward trend and convergence to above 90% accuracy.

6.2 System Results

We also measured training cycles over time and show results in figure 6. Since all peers were run on identical hardware, it follows that they all take proportionate runtime as shown by the roughly parallel lines. We also note the linearity of these results indicating that our runtime **does not** scale with the addition of new peers. This is important if we ever wanted to deploy this network full scale.

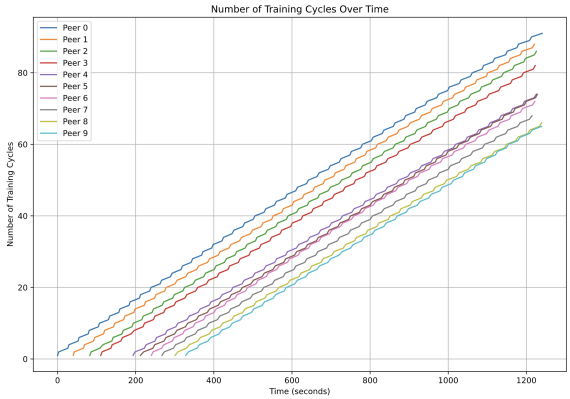

Figure 6: Number of training cycles over time. Note the X-intercepts are different because the clients were started sequentially.

6.3 Peer Discovery

We measured how quickly peers discover other peers in the network and show results in figure 7. The data shows a consistent linear increase in peer connectivity as new peers join the network, with each peer taking roughly the same amount of time to discover the full network regardless of when they joined. This linear growth pattern suggests our peer discovery mechanism is stable and predictable, with new peers being effectively integrated into the network at a constant rate. The consistency of this pattern across different peers, despite their varying initialization times and initial peer lists, demonstrates the robustness of our peer discovery approach. All peers eventually converge to a complete view of the network, showing that our system successfully achieves full connectivity while avoiding the network congestion that could occur from simultaneous peer discovery attempts.

6.4 Network Scalability

We measured model aggregation time as a function of network size and show results in figure 8. The blue dots represent individual aggregation events, while the red line shows the average aggregation

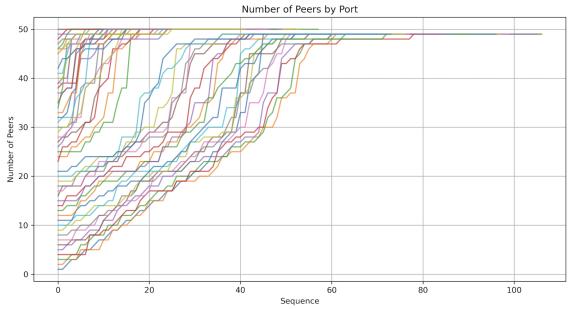

Figure 7: Peer discovery of training instances during testing. Note the linear increases in connectivity.

time. For networks of larger than 20 peers, we observe that aggregation time remains relatively stable around 200-400ms, with some expected variance. This stability is achieved through our random peer selection strategy - by only aggregating with a subset of peers rather than the entire network, we maintain consistent performance regardless of network size. The noticeable peak after 45 peers is due to complications in the testing environment due to peer failure, leading to a large degree of timeouts. Under normal operation, the data suggests our system maintains stable performance even as the network grows, indicating that our design choice of random peer selection effectively prevents scaling issues that would typically arise in larger networks.

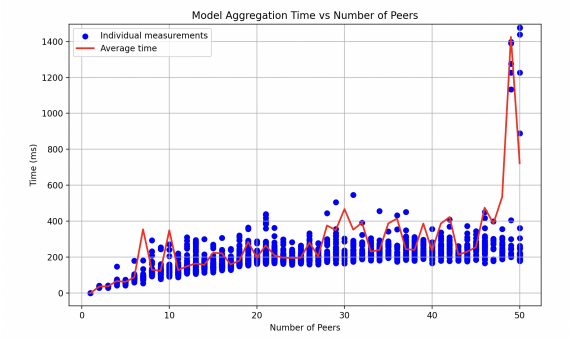

Figure 8: Aggregation times. Note the stability of times after 20 peers are in the network.

7 Future Work

There are two main avenues of future work we would like to consider. The first is a more dynamic program that allows for multiple models to be trained at once and better specification of a data file, training resources, etc. While the model we developed has all of the tools to train a P2PFL system, it lacks some of the ease of use of a fully deployable model. Moving forward we would like to work out all of these issues from a generalizable and human focused lens to make our model widely usable. This includes improvement to the UI, better multi-modal management (currently the best way to do it is boot a new network on a different port), and dynamic data allocation of ingrained computer data—such as messages or photos. We are excited to purse these avenues and want to see where else we can scale this project too.

We also want to explore different methods for sharing and storing state data. While using the file system is a very solid approach it still incurs the heavy cost of I/O which we could look at trying to eliminate.

Finally there are a million different things we could look at to making models more secure. A common problem in P2P networks is malicious users. Since there is no centralize service it is hard to catch these users and then can mess up results for all. For example if a peer sent back a poisoned model (one with intentionally incorrect weights) that would mess up the convergence of our model. There are many recent lines of research that seek to address these problems and exploring these and integrating them would be a nice direction as well.

8 Conclusion

In this paper we introduced P2PFL, our peer-to-peer federated learning system that allows users to train ML models without giving up their data or requiring expensive servers. Using the MNIST dataset, we demonstrated that P2PFL works really well - even when peers only had access to 7 out of 10 numbers, our system consistently achieved above 90% accuracy through collaboration. This is a significant improvement over the 70% ceiling that would be possible without federated learning.

Our tests showed that P2PFL scales effectively too. When we added more peers to the network, the system kept running smoothly with aggregation times staying between 200-400ms for networks up to 45 peers. The peer discovery system worked reliably, with new peers quickly finding and connecting to the network in a predictable way.

While there's still work to be done on making the system more user-friendly and secure, P2PFL shows that we can build effective ML systems without relying on expensive centralized infrastructure. Our system makes it possible for people to work together on ML problems while keeping their data private and sharing the computational load. Build-

ing this system challenged us to combine neural networks, distributed systems, and networking in ways we hadn't before, especially in making Go and Python work together seamlessly. The project deepened our appreciation for both the complexity and potential of distributed ML systems.