Data-driven, Bottom-up, Asynchronous approach to Federated Machine Learning for Fairness and Scalability

Michelle Lim

Yale University

Abstract

Federated learning (FL) is a distributed machine learning approach that can train a model on a decentralized set of data without the data ever leaving the devices. The predominant FL model in production is the Google FL protocol (GFL), which is a top-down and synchronous and time-division resource allocation system. Despite its many benefits, it has led to multiple challenges, mainly increased bias in selecting training data and challenges in scaling beyond a small number of devices. In this paper, we design and investigate an alternative FL architecture to address the challenges of GFL and at the same time reap the benefits of FL. Our system is a bottom-up data-driven, asynchronous resource allocation system, that maximizes key resources consisting of samples, computation, and bandwidth. Our system can improve fairness and scalability without compromising convergence efficiency.

1 Introduction

Federated Learning (FL) is a distributed machine learning strategy which allows the learning to be done on the devices that created the data, instead of in a centralized data center, or a set of data centers. The reasons for training on decentralized data include enhanced data privacy, reduced latency, and increased efficiency. Instead of bringing the data to a centralized model, the model is brought to the data, and the model parameters are then shared between the devices as they each train on their own piece of the data. Training on the device also saves much bandwidth when the data being trained on is larger than the model being trained, which is the case in many applications.

A major paper in the field of FL, (GFL, [1]), developed a Federated Learning Protocol (FLP) which allows devices to communicate with a central server to train deep neural networks over the course of many rounds, and store the trained model in the server’s persistent storage. Though this model showed great success in many areas, it also presented many open challenges in Section 11 of GFL. Two major challenges that the paper presented are as follows:

Device Selection Bias: Though all devices (mobile phones) that met a set of eligibility criteria were allowed to join in training, only the devices that had the computational power faster than a certain threshold were allowed to actually contribute to the model. Once devices began training on their own data in the
configuration phase, there were given a set amount of time in which they could report with their model updates. If they were too slow to report back in that time window, then their model update was dropped. The GFL paper revealed in their evaluations that 22% of the time, model updates were dropped because the device did not report back in time. This results in faster and more expensive phones always having their data contribute to the model whenever they are selected, and slower and less expensive phone having their data largely ignored.

![Google FL protocol](taken from Google FL)

As seen in Figure 1, in the Selection round, many of the devices were rejected because of either the network or because of the device count constraints. In the Reporting Round, 1 of the devices trained for too long and so was not included in the aggregation.

**Figure 1: Google FL protocol**

**Scalability:** Requiring all the devices to receive the model from a central server and report back to the central server limits the scalability of the FL. Though the GFL server was designed to scale as more devices joined, the scaling still required more machines to be running in the server, and even then the number of devices that could join each round was still often limited to about ten percent of the total number of devices available. This presents a major underutilization of resources, especially considering that the requirements for a device to be considered available are that it is idle, charging, and connected to an unmetered network.

**Our contribution**

In this paper, we design and investigate an alternative FL architecture to address the challenges of GFL and at the same time obtain the benefits of FL. We designed our system to utilize the key resources of a learning system, data, storage, and compute slots, while simultaneously bounding the amount of bias introduced into the system by slower devices. Our observation is that GFL can be considered as a top-down, synchronous, time-division resources allocation system, leading to the aforementioned challenges. BitFL, on the other hand, is a bottom-up, data-driven, asynchronous resource allocation system.

**The Fairness Metric**

A fair federated learning framework is one in which the computation and bandwidth of the device a sample resides in should not impact whether it is included in the training. This is equivalent to reaching this goal:

\[
\| M_f^* - M_c^* \| < Var (| M_c^{* \text{D}_{20\%}} - M_c^* |)
\]

where \( M_f^* \) refers to the result of our FL system with decentralized data and \( M_c^* \) refers to the result of centralized machine learning. \( M_c^{* \text{D}_{20\%}} \) refers to the result of centralized machine learning where a random 20% of the data is dropped out.

The distance in accuracy between our system which includes all the contributions of all training data, and a centralized system that includes all training data, should be closer than the distance between a centralized system that excludes 20% of the data and a centralized system with all the data. We picked 20% because the GFL model drops 20% of their clients in each round of
training (Appendix A of [1]). We will empirically measure these distances in our evaluations.

2 Architecture

Our system consists of a set of key components that manage key resources, namely samples, computation, and bandwidth. Our job scheduler makes training and aggregation asynchronous. Our bias controller (which is baked into the receiver and sender queues) coordinates update arrival rate so no device gets too much influence. Our model synchronizer synchronizes a global model among all nodes while distributing the network burden.

Asynchronous Job scheduler: Receiver queues, Aggregator, Backpropagator and Sender queue

![Figure 2: Neighborhood cluster of 3 devices running our system](image)

They broadcast their SGD updates (with their corresponding color) to each other and aggregate the updates via an aggregator.

To design for asynchronicity, we have created (1) a receiver queue system to queue model updates from other devices; (2) an aggregator that consumes the receiver queue system (picking from queues randomly) and applies the dequeued model updates to the local model. (3) A backpropagator then runs backpropagation using the updated model and produces a model update. This model update is enqueued onto both the sender queue and applied onto the model; (4) The sender queue queues model updates received from all other devices.

The backpropagation should run for a fixed number of epochs with the model after each aggregation. We have to keep the number of epochs fixed so as to maintain fairness across devices, such that no model update will contain a disproportionate amount of training on a particular device’s data.

Bias controller (in sender and receiver queues)

The bias controller coordinates update arrival rate so no device gets too much influence. Without a bias controller, and assuming random dequeuing, the queue with the infinite update arrival rate will dominate the model trained.

The mechanism for controlling update arrival rate is as follows: reject enqueue requests from a peer device when a certain fairness rule is violated. The ML programmer can pick between two fairness rules:

1. All queues can only be at most k times longer than the current shortest queue length:
   \[
   \frac{\max_i(\text{length}(\text{queue}_i))}{\min_i(\text{length}(\text{queue}_i))} \leq k
   \]

2. Set a threshold for the minimum queue length required (\(\alpha\)). There must be at most a set number (e.g. 80%) of queues whose length is above \(\alpha\) at any point in time.
   \[
   |\{\text{queues s.t. } \text{length}(\text{queue}) \geq \alpha\}| \leq 80\%
   \]
The rejection message can be implemented via throwing a 405 Error, and this will signal to the peer device to slow down their sending rate.

Model Synchronizer

It is important to synchronize the model regularly, for every set number ($\Delta_t$) of clock cycles. The model synchronizer will be a single node that has computed the model from the most number of contributing examples. The time for model synchronization will be global and synchronized as all devices will be using the same clock.

We will select this node through a looser version of the leader election component of the Raft protocol [2]. To begin an election, a device votes for itself and sends RequestVote RPCs from all other servers in the neighborhood. In our system, these RequestVote RPCs include information on how many devices and contributing examples their model has included. To eliminate devices with the most biased model, voting devices with fairer models will not vote for devices with a more biased model than themselves.

Once a leader is established, the leader sends its entire model to all devices in the neighborhood. We have adapted the BitTorrent protocol [3] to leverage all devices to relay the model to all devices, hence minimizing the bandwidth requirements from the leader device.

3 Methodology

Experiment Platform

We deploy a neighborhood of 10 servers, each running on a Docker image on our local machine. They are Flask apps that communicate with each other via the HTTP protocol. Our local machine is a MacBook Pro (macOS mojave) running on 2 GHz Intel Core i5 processing power. Later, we hope to run these Docker images on container net.

Application

We evaluate our system using the popular ML application of image classification, which classifies images into categories. Here, we are using the popular MNIST dataset of handwritten digits [4], which has a training set of 60,000 examples, and a test set of 10,000 examples. Each image is labeled as one of the 10 digits. We implemented stochastic gradient descent with backpropagation using Facebook’s PyTorch [5], to train a CNN that learns the relationship

4 Evaluation Results

We evaluate our system in terms of its convergence rate. We will present the results at a later date.

5 Conclusion

We introduce a data-driven, bottom-up, asynchronous approach to federated machine learning, with an emphasis on improving fairness and scalability. Our system removes GFL’s rigidity of having to drop slow devices. We allow devices to aggregate updates asynchronously, so the slow devices’ updates will still be considered in the training process. We introduce a novel and direct way of controlling for bias from devices and their examples. We also introduce a novel peer-to-peer model synchronization model, that picks out the least biased but most well-informed device as the leader, and then efficiently utilizes the scarce and heterogeneous bandwidth of neighboring devices to spread the model to every device. Our evaluation shows that our model convergences quickly, while including
data from slow devices. We conclude that our system is an effective and fair asynchronous FL system.

6 Acknowledgements

I would like to thank my teammates Weitai Ting, Yuki De Pourbaix and Tim Adamson for working on this project with me. We would like to thank Professor Richard Yang for being our project advisor, and for his mentorship, guidance, and suggestions. We would also like to thank George Saussy for helping us set up some parts of the infrastructure of our code base.

7 Supplementary Information

Code for our project can be found at https://github.com/michellelimxuanli/gaia2.

8 References


