Data-Driven, Bottom-up, Asynchronous Federated Learning

Michelle Lim\textsuperscript{1}, Weitai Ting\textsuperscript{1}, Yuki de Pourbaix\textsuperscript{1}, Franck Le\textsuperscript{2}, Shiqiang Wang\textsuperscript{2}, Y. Richard Yang\textsuperscript{1}

\textsuperscript{1}Yale University, \textsuperscript{2}IBM T. J. Watson Research Center

\{michelle.lim, weitai.ting, yuki.depourbaix, yang.r.yang\}@yale.edu, fle@us.ibm.com

Abstract

Federated learning (FL) is a distributed machine learning approach that can train a model on decentralized sets of data without the data ever leaving their host devices. The predominant FL model is the Google FL protocol (GFL), which is a top-down, synchronous, and time-division resource allocation system. It faces several issues, such as increased bias in selecting data and inability to scale beyond a small number of devices. In this paper, we design and investigate an alternative FL architecture to address the challenges of GFL. Our system is a bottom-up, data-driven, asynchronous resource allocation system, that maximizes key resources consisting of samples, computation, and bandwidth. The performance of our system is evaluated with the MNIST dataset on a networked prototype system. The experimental results show that our system improves fairness and scalability without compromising convergence efficiency.

1 Introduction

Federated Learning (FL) is a distributed machine learning strategy that allows the learning to be done on the devices that created the data, instead of in a centralized data center, or set of data centers. The reasons for training on decentralized data include enhanced data privacy, reduced latency, and increased efficiency.

The predominant FL model is the Google FL protocol (GFL, Bonawitz \textit{et al.} [2019]), a Federated Learning Protocol 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 shows great success in many areas, it also presents many open challenges. Our observation is that GFL can be considered a top-down, synchronous, time-division resource allocation system, leading to the following challenges:

1. \textbf{Device Selection Bias:} Only devices that have the computational power and network resources are allowed to contribute to the model. The GFL paper reveals in their evaluations that 22\% of the time, model updates are dropped because the device does not report back in time. Faster phones can always contribute their data, but slower phones’ data are ignored.

As seen in Figure 1, in the Selection round, 2 of the devices are rejected because of the device count constraints. In the Reporting Round, 1 of the devices trains for too long and is not included in the aggregation.

2. \textbf{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. Scaling requires more machines to be running in the server, and even then the number of devices that can join each round is limited to about ten percent of the total number of devices available.

Our Contributions

In this paper, we design and investigate an alternative FL architecture to address the challenges of GFL. We maximize the key resources of a learning system including data from slower devices while bounding the fairness of the data used. Our main contributions in this paper are as follows:

1. We design a data-driven asynchronous learning system, where the creation of new data triggers learning, and whereby participating devices can participate asynchronously.

2. We design an algorithm to bound the fairness of the model, ensuring all devices commit examples.

3. To minimize the bandwidth requirements and maximize scalability, we design clustering algorithms to organize
the topology of the network of the learning system.

4. We evaluate the performance of the proposed architecture via extensive experiments using the MNIST dataset. The experiments confirm that our proposed approach provides the same performance as GFL but with stronger fairness and scalability guarantees.

2 The BiasControl Aggregation Algorithm

2.1 Background: Federated Averaging

GFL’s FederatedAvg algorithm is synchronous. On each device, the algorithm first iterates the local minibatch gradient descent \( w_k \leftarrow w_k - \eta \nabla \ell(w_k) \) a set number of times before sending \( w_k \) to a central aggregator. The central aggregator takes a weighted average of the all \( w_k \)'s received, weighted by the number of samples each device \( k \) has: \( w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{K} w_{t+1}^k \). The weight coefficient \( (\alpha_k) \) given to \( w_{k}^t + 1 \) is \( \frac{n_k}{K} \).

During aggregation, all \( w_k \)'s that are aggregated are the result of the same number of epochs. The GFL weight coefficient of \( \alpha_k \) thus does not extend to the asynchronous model, where aggregation will entail combining \( w_k \)'s that are the result of different numbers of epochs. We cannot use the GFL weight coefficient of \( \alpha_k \) in the asynchronous setting also because it would be very coarse in measuring the relative influence of each \( w_k \). Unlike in GFL whereby all devices share the same \( w_k \) at the start of each round of FedAvg, in an asynchronous setting each device’s aggregated \( w_k \) is the result of different combinations of contributions from different devices. Federated learning in such an asynchronous setting is challenging especially when the devices have heterogeneous speeds and the data is non-IID distributed at different devices, as discussed in Wang et al. [2019].

2.2 Problem Formulation

We have to find an aggregation method that considers the influence of examples and duration of training, in assigning a weight. That is, given a set of models \( \{w_1, w_2, w_3...w_m\} \) at device \( d \), whereby multiple models can come from the same device, find the k-vector \( \alpha \) to form the new \( w_{t+1} \) such that we are minimizing some fairness constraint on the influence of each device. \( \alpha_i \) refers to the ith entry of \( \alpha \).

\[
w = \sum_{m=1}^{M} \alpha_m w_m
\]

To measure the influence of examples and duration of training, we need to attach metadata \( v \) to each model. \( v \) is a metadata \( n \)-tuple, where each entry represents the number of contributing examples it has ever seen from devices 1 to \( n \). We have a a set of metadata tuples \( \{v_1, v_2, v_3...v_m\} \) at device \( d \), where \( v_i \) corresponds to the metadata for \( w_i \).

First, let us define how we will combine the metadata tuples to form a new metadata tuple:

\[
v = \sum_{i=1}^{M} \alpha_i v_i
\]

Let \( f(v) \) denote the fairness constraint measuring the fairness of weighted contribution from each model. Since \( v \) represents the weighted contribution from each model, we design \( f \) to be a function of \( v \). \( f(v) \) is user-defined. For simplicity, it could be:

\[
f(v) = \frac{\max_i (||\alpha_i v_i||_2)}{\min_i (||\alpha_i v_i||_2)}
\]

The goal is to find \( \alpha \) to optimize fairness \( f(v) \):

\[
\min_{\alpha} f(v) \\
\text{subject to } \alpha_i \geq 0, \sum_{i=1}^{M} \alpha_i = 1
\]

2.3 Algorithm

Since we know the value of \( \alpha_i v_i \) will always be positive, the smallest value possible for \( f(v) \) is 1. But if it is the case that if \( f(v) \) is 1, then the max and min of \( \alpha_i v_i \) are the same, and therefore all of the \( \alpha_i v_i \) have the same value. Further, since it must be the case that the \( \alpha_i \) sum to 1, the value of each \( \alpha_i v_i \) must be \( \frac{1}{M} \). Thus the algorithm only has to compute:

\[
\alpha_i = \frac{1}{M||v_i||_2}
\]

3 Architecture

Our system consists of a set of key components (Figure 2) that manages key resources, namely samples and computation. (1) The Receiver and Sender Queues on a device allows it to receive requests from the Model’s and send produced Model’s asynchronously. (2) The Smart Sample Store is the source of new samples for the learning system. (3) The Device State consists of the device’s model \( w \) and its metadata \( v \). (4) The Aggregate-Update Job Scheduler schedules aggregation jobs or local update jobs based on the status of the receiver queues.

![Figure 2: Key Components of our system](image)

3.1 Receiver and Sender Queues

The receiver and sender queues are message queues that store Model’s. The receiver queues receive from the device’s neighbors, and the sender queues send to the device’s neighbors. Thus, the receiver and sender queue systems each consists of a queue for each of the device’s neighbors. The leader device also has to send and receive from other leader devices. Thus, the leader device’s sender and receiver queues also include queue for each of the other leader devices.
The receiver queues are consumed by the aggregation jobs. The local update jobs update the local model in the smart sample store, and enqueues a copy of the new model onto the sender queues for the recipients.

3.2 Smart Sample Store

The smart sample store is a queue of samples that is consumed by the local update jobs. When the application creates a new sample, it enqueues the sample onto the smart sample store. For instance, in an image classification learning task, when the end-user takes a photo and labels it, the photo and its label is enqueued onto the store.

3.3 Device State

The device state consists of the device’s model \( w \) and its metadata \( v \).

3.4 Aggregate-Update Job Scheduler

The Aggregate-Update job scheduler arbitrates whether to run an aggregation job or local update job based on the status of the receiver queues.

Aggregation refers to the process of combining Model’s received from other devices with the Model in the Device State to apply onto the device’s model \( w \) and \( v \) in the Device State. (See Algorithm 1)

```
Algorithm 1 Aggregation Job at Device d
1: ListOfModels = Dequeue all Model’s from ReceiverQueues //\{w_1,v_1\}, \{w_2,v_2\},...\{w_m,v_m\}
2: ListOfModels.add((w_d,v_d))
3: Calculate α using (5)
4: w = Aggregate ListOfModels \( w \)'s using (1)
5: v = Aggregate ListOfModels \( v \)'s using (2)
```

The local update job refers to the process of running back-propagation on new samples and then updating the local model. Backpropagation refers to the calculation of the gradient of the error function with respect to the neural network’s weights. The local update job first dequeues a minibatch from the Smart Sample Store and runs backpropagation to produce a ModelUpdate stored in the Sample Store and sent to all neighbors. (See Algorithm 2)

```
Algorithm 2 Local Update Job at Device d
1: \( x_i \) = Dequeue from SmartSampleStore
2: \( w = w - \eta \nabla \ell(w,x_i) \)
3: \( v = v + \delta_i 1 \) //Add 1 to entry in \( v \) representing self’s device examples
4: return
```

We want to prioritize aggregating at all times as long as the receiver queues are non-empty. This is because we wish to make use of the latest models instead of running backpropagation on an older model. (See Algorithm 3).

4 Network

For a fully decentralized system, we rely on peer-to-peer communications in an ad hoc network to update a distributed global model. Every device that joins must be able to integrate itself into the network without direction.

4.1 Network as a Graph

The network used for evaluations is a partially connected network with no partitions, such that every device can reach every other device on the network. We visualize the network as a connected, undirected graph \( G \) such that \( G = (V,E) \) in which \( V = V \) is the set of devices with new data and there is an edge \( u,v \) in \( E \) if there is a bidirectional connection between devices \( u \) and \( v \) where each can receive the other’s transmissions. All devices without data are not considered vertices but may act as carriers between devices on the network.

4.2 Peer-to-Peer communication

There are many existing gossip algorithms designed for P2P networks that synchronize all devices on the network TODO: citations. Though gossip algorithms focus on efficient dispersion of data across all devices on a network, this is not our use case. As we intend very frequent updates, every device sending to every other device is unscaleable for bandwidth limitations. We therefore split our network into clusters.

4.3 Cluster

A cluster is considered a set of devices on the network who frequently update one another’s queues. They must have good connections with high throughput and low latency. Each cluster consists of a single cluster head, with every other member of the cluster holding a direct connection to its cluster head. Every device sends its update to the cluster head as soon as the update is calculated (as controlled by the Bias Control Algorithm). The cluster head then floods the members of its cluster with that update. If the model of the cluster head diverges from the cached global model by some threshold, it sends the new model to its neighbour clusters. After a cluster head has received updates from every other cluster head, it sends a 1-bit message to all neighbour clusters who relay the message to their neighbours etc. until the full network has reached the message. On reception of the message, the cluster pauses and waits to receive the new global model. The cluster head who sent the message floods its neighbour clusters with the new global model and then restarts training. Every cluster head waits to receives the new global model, floods it to all members of its cluster and its neighbouring cluster heads, then restarts training. This ensures global synchronization motivated by the availability of data without flooding
the larger network too often. There are several heuristics we consider applying for when to update the global model, with continuing research in the area.

5 Methodology

5.1 Experiment Platform
We deploy 3 clusters with 10 devices in total, each device running on a Mininet network. Each device runs a Flask app that communicates with other devices via the HTTP protocol.

5.2 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 LeCun and Cortes [2010], 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, to train a simple 3-layer perceptron that learns the relationship between images and the digits they represent. The hidden layer has 500 units and a ReLU activation.

5.3 Convergence Criteria
Our convergence criteria is slightly modified from that chosen by Hsieh et al. [2017]: the algorithm has converged if the loss value changes by less than 2% over the course of 100 minibatch iterations rather than 10 batch gradient descent iterations. This is because a batch gradient descent is larger than our system’s choice of the minibatch gradient descent iteration.

5.4 Settings
We compare the convergence speed and accuracy of the system using our BiasControl during aggregation and using the GFL FederatedAveraging algorithm. We study the following settings:

1. Balanced class distribution: All devices have same number of images per class.
2. Unbalanced class distribution: Where each device only has one type of image.
3. Synchronous federated learning system: Where we wait for the slowest node to complete per round. Each device sends out one update and sleeps. The leader decides to wake all devices up when all 10 updates arrive.

The first two help us explore the superiority of algorithms in highly non-IID data. The thirds helps us explore the performance of the algorithms in a synchronous setting versus asynchronous setting.

We consider minibatch stochastic gradient (SGD) descent, whereby we run each SGD iteration on one minibatch. We run the algorithm on 5 minibatches before sending the model to other devices. We use the same initial random seed at all nodes to initialize each layer of the neural network using a normal distribution.

6 Experimentation Results

6.1 Distribution of Data
We partition the MNIST data over 2 clients as follows: IID, where the data is shuffled, and then partitioned into 2 clients each receiving 30,000 examples and Non-IID, where we first sort the data by digit label, and extract 5400 images from each of them. Then, we assign the labelled images “0” to “4” to the first device, and the images “5” to “9” to the second device. When run our system with and without the BiasControl weighting (“fairness_weights” and naive_averaging”) and compare them to the case where there is no communication (“no_communication”).

The results are reported in Figure 3. Using our system provides a much bigger improvement over non-communication in non-iid distribution than in iid distribution.

Because federated learning data tends to be non-iid, and our system was designed for the non-iid distribution in mind, the rest of our experiments will use the non-iid distribution setting.

6.2 Relative Speed of Devices
In non-iid distributions, we found that using our system allows us to meaningfully include the contributions of slower devices. We ran a cluster of size two where one of the devices was made to sleep for 0.5 seconds before sending its update. In such a setting (Figure 4), including the contribution of such a slow device using our asynchronous system resulted in 56.70% accuracy, which is a 12% performance improvement on the model that does not include such a device (“no_communication”). This 12% improvement consists of each model learning to recognize digits that did not exist to its originating device.

Conclusion: We also found that in clusters whose speeds are mostly similar, we can obtain a high accuracy of 80.00% in non-iid distributions. This is because the models are both in the same stage, and so there is no stale model that will cancel out the learning from the fresher model.

\[\text{There are only 5400 images in label 5}\]
6.3 Comparison with Synchronous Federated Learning

GFL is a synchronous system that waits for all devices to send updates to an aggregator before aggregating the updates. Until the aggregator is done, all nodes that have sent updates to it during the round have to stop. To have model this synchronicity in our system, we block aggregation until all devices’ updates have been received. We also forbid backpropagation until all devices’ updates are aggregated. The results are reported in Figures 5 and 6.

In a scenario where the cluster had the same speeds, the asynchronous system takes a much longer time but gets a better accuracy. In a scenario where the cluster had different speeds, the asynchronous system takes lesser time and gets a better accuracy. Our system took lesser time since it may maximize the faster nodes’ clock cycles to augment the model. Even though Wang et al. [2019] suggested that an asynchronous system will perform worse on non-iid distribution because of overfitting on faster nodes, the BiasControl algorithm probably reduced this overfitting. We conclude that it is better to use our system with its synchronicity in a setting with heterogeneous devices.

7 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.

Acknowledgments

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, and Tim Adamson for setting up Docker.

References


Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.