FLADS: Federated Learning with an Asynchronous Distributed System

Aman Bansal
Stanford University
aman0456@stanford.edu

Aditya Chandrasekar
Stanford University
adichand@stanford.edu

Gabriel Mudel
Stanford University
gmudel@stanford.edu

***Abstract*—As we go further into this century, the questions surrounding data and privacy are going to get more prominent and harder to answer. There has already been substantial outcry regarding misuse of private data by corporations around the world. This has exacerbated the need for machine learning methods which ensure data privacy. One of the major cause of such concerns are the small devices that many individuals use in their everyday lives. These devices can range from a virtual assistant like Amazon's Alexa to a self-driving car. There are long-standing concerns around corporations listening to your conversations and monetizing them, all done in the name of improving their products. In this paper, we introduce *FLADS*, a tool which combines Federated Learning concepts with an Asynchronous Distributed System to create a network of node which don't communicate with any centralized server. Rather, they facilitate learning by sharing their data among themselves. *FLADS* is novel in that it proposes a trade-off between accuracy and training throughput by allowing for stale gradients. It also combines the modules of federated learning and distributed systems in a plug-and-play way; any of the algorithms can be independently changed and plugged to our implementation provided they follow the interface. We also run experiments to quantify how well *FLADS* performs and the trade-offs it offers. From the experiments, we conclude that allowing for stale gradients can improve overall model training throughput with only marginal impacts to accuracy, and may even improve accuracy in a very skewed data distribution.**

I. Introduction

Currently, most of the smart devices that we use owe their "intelligence" to the data that they collect and the machine learning algorithm that they run to use that data and learn from it, thereby improving future performance. These products have immensely changed the way we live our life for the better, but they have also come with a cost. Almost all such products are owned and run by a central entity which stores and has access to the data collected by these products. It's undeniable that advancements in technologies such as autonomous vehicles would not have occurred in lieu of abundantly available data. However, the current setting of these products suffer from some major drawbacks:

- The user data is accessible by the corporation which can constitute violation of user privacy. This data can further get leaked or hacked [1], leading to even larger concerns.
- The whole system is dependent on a central power. They can stop any particular user's product from improving by using data.

In this paper, we present *FLADS*, a tool that tackles these drawbacks. The problem statement that *FLADS* solves is as follows:

Problem Statement: Given some nodes (which correspond to smart devices) that continuously collect data and use it to perform model updates, ensure that every node is able to learn from every node's data while following these restrictions:

- **Fault Tolerant**: No single point of power/failure.
- **Privacy Preserving**: No node shares its data with any other node.

We first give related works in section II, design details in section III, implementation details in section IV, experiments in section IV, and then conclude in section VI.

II. Related Work

There's a significant field of Machine Learning focused on Distributed ML research [2], [3], [4]. This area of research is focused on using multiple compute machines to reduce the training time of a neural network. The majority of works that we have found in our review do not concern themselves with data safety and privacy handling.

Federated Learning [5], [6], captures the idea of training an ML in a private way on compute-constrained machines such as mobile devices. These methods are privacy-preserving but centralized – ultimately, data is being sent back to a central server for training. We did an extensive search for a federated learning paper [7] with fault-tolerance as a design goal. During our search, we found [8], a Learning algorithm designed to be robust against byzantine faults. However, this assumption was stronger than the environment we were designing for, especially since Byzantine failure models can only handle $\lfloor \frac{n}{3} \rfloor$ failures in a n-node network.

III. Design

We begin the description of *FLADS*'s design by enumerating the different components that go into building it. We first explain the design of a node and then explain how that node interacts with other nodes to form a working distributed system.

A. Node

A node is a symbolic representation of a smart device as defined in the problem statement. We define its properties below:

- It has a compute environment.
- It has a Deep Neural Network (DNN) which is used for taking the "smart" action.
- It has the ability to collect data to perform DNN training locally.
- It has a network card/interface for it to receive and send messages over standard protocols like TCP/UDP. This will be used to communicate with other nodes in the system.

B. The Distributed System

Our system consists of a network of multiple such nodes. All the nodes in the system are initialized so that they have the same initial DNN state. The nodes then perform DNN forward passes and communicate with each other is to achieve the following goal:

Goal: Train the DNNs of all the nodes in the system using data from as many nodes as possible and keep the state of the DNN consistent across the nodes.

To ensure DNN model consistency across nodes, *FLADS* uses a consensus protocol. We use a modified version of the Zookeeper Atomic Broadcast Protocol (ZAB) for this task. Our implementation of ZAB is explained in Section IV. We now detail the high-level functionality of our system:

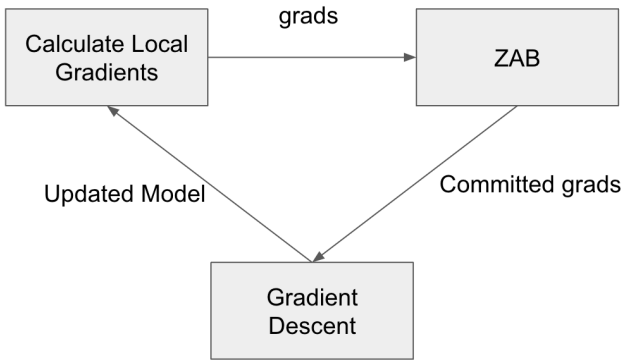

Fig. 1. Summary of *FLADS*

1) All of the nodes perform a modified DNN training with their own data. They choose a mini-batch and use it to perform a forward and backward pass over the DNN. The nodes then store the gradients calculated after the backward pass but don't update their model with these gradients.
2) The gradients are then passed to the ZAB protocol as a client request. The protocol, treating the gradients as a client request, enforces a consensus among all the nodes to agree on which order these gradients should be applied in.
3) Every node now does a gradient descent with the gradients in the consensus order.
4) After the gradient descent updates, the nodes calculate the next set of gradients on the updated DNN and return to step 1.

The process is summarized in Figure 1. The correctness of this process follows from the total order guarantee given by the ZAB protocol.

IV. Implementation

FLADS is implemented in a modular way. The ML component is completely independent of the distributed system component, allowing *FLADS* to have a plug-and-play functionality. Any gradient-based algorithm can be plugged into the rest of the system.

We start by discussing the implementation of individual components and then describe how they are combined together. The GitHub repository for our implementation can be found here. We have implemented our code in *Go*.

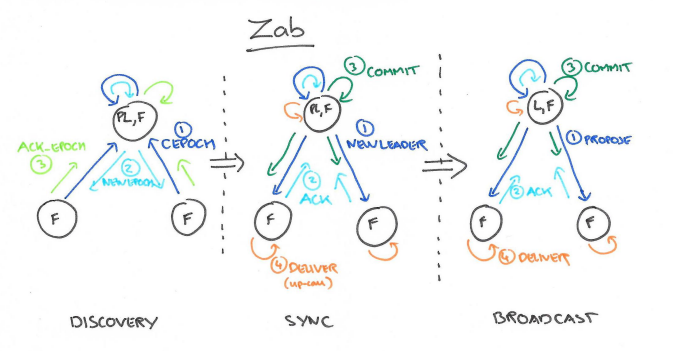

Fig. 2. Summary of ZAB Protocol [10]

A. Machine Learning Implementation

We used *gotorch* which extends support for PyTorch-like Tensor operations. The DNN architecture used for this project was a 3-layer feedforward multilayer perceptron (MLP) trained with vanilla stochastic gradient descent (SGD) on the well-known MNIST dataset. The manner in which the DNN parameters are updated is a key difference between our implementation and that of a "vanilla" DNN. Where a vanilla, non-distributed DNN would perform a gradient update immediately after each minibatch of training, ours waits to achieve consensus among the nodes before performing the update. The reason for this is that we would like our models' states to not diverge. If more than a small amount of model divergence were to occur, then the gradients from one model may hurt the performance of another model.

B. System Implementation

We use an adaptation of the Zookeper Atomic Broadcast (ZAB) protocol given in [9] for our distributed system algorithm. Figure 2 summarizes the ZAB protocol with all its phases.

While from a technical standpoint we implement exactly the protocol given in [9], we made a few different choices to tune the implementation for our use case. Below, we give our implementation choices and the reasoning for those choices.

- *No Self Messages*: Some ZAB implementations assume that the leader node is also like a follower node and also participates in all the protocol phases as follower by sending messages to itself (also shown in Figure 2). We have not treated leader as a follower and we just adjust the message counts by 1 when checking for quorum instead of sending a self message.
- *Next Leader Choice*: The ZAB protocol leaves the choice of next leader in case of a view change open until all the nodes get the same next leader ID. We set the next leader ID to be simply the node whose ID is one greater than the former leader (modulo the numer of nodes in the network). In failure cases, we manually tell the crashed node who the current leader is upon boot. A more sophisticated leader election protocol is left to future work.
- *Heartbeats*: Each follower sends the leader a heartbeat, and the leader broadcasts its heartbeat to all followers over a separate UDP connection (Zab messages are sent over TCP to preserve ordering).

We also had some optimizations that we had thought of but did not include in our implementation.

- *Send Model instead of Gradients*: A single update request can send multiple gradients from a client. Instead of sending all of these gradients as part of a new proposal, the leader could simply calculate the final model state after applying these incoming gradients and send a proposal with the new model state. Since the size of the model and the size of a single gradient are the same, sending the whole model state in a proposal may be much more efficient than sending multiple gradients.
- *Batch Gradients*: Instead of sending a request to the leader as soon as the gradients are ready, wait for `GRAD_BATCH_SIZE` gradients and send them together to the leader. Combined with the above optimization, this could significantly reduce network congestion.

C. Combining the Two

Combining the Machine Learning (ML) and the Distributed System (DS) modules may look like a simple task but it poses an important question: when should we run the next batch iteration in the DNN? We initially planned to explore the following 3 ways of picking the next batch iteration. We discuss our three implementations in terms of gradient *staleness*, which refers to how old the gradients with respect to the current state of SGD training.

1) **No Staleness**: SGD is an inherently sequential algorithm, so applying gradients out of order (i.e. stale gradients) can lead the model states to diverge. No staleness means that we only allow committing gradients which have been calculated using the current model's weight. This is a very difficult synchronization requirement to enforce - effectively asking the ML module to continuously wait for the

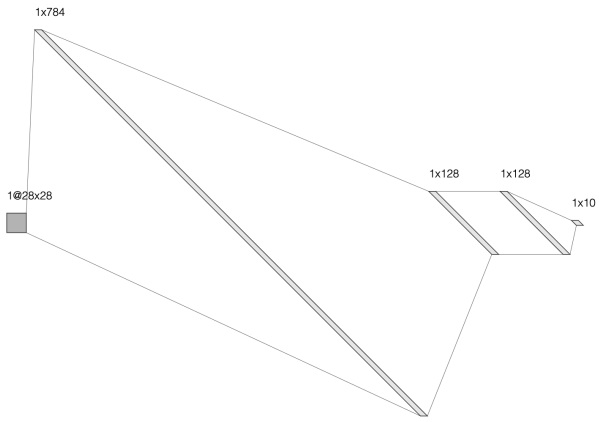

Fig. 3. SmallNN Architecture

DS module - and will significantly hurt the training efficiency.

2) **Strong Staleness**: Strong staleness means that we're allowing any gradient to be committed independent of how many SGD epochs before they were calculated. We expect the accuracy of this variation to be very low since gradients older than a certain number of batch iterations will essentially be random noise. We do not include these in our plots, but we conjecture that accuracy would tend towards that of a model which guesses randomly.
3) **Weak Staleness**: Weak staleness means that we're allowing some stale gradients but not beyond a certain number of batch iterations. We find that this variation yields a reasonable trade-off between accuracy and training time. This is the version we have implemented.

V. Experiments

We run experiments to focus on both the modules, ML and DS.

A. Accuracy

The details of the experiments are:

- Dataset: MNIST (image size = 28*28 grayscale, number of classes = 10)
- Model: A feedforward MLP with 2 hidden layers, both with size 128. See Figure 3.
- Number of Nodes: 3
- Metric: Validation Accuracy vs. Epoch
- Dataset Split across nodes:
  - Uniform: The dataset uniformly randomly divided into 3 parts for each node, and
  - Range-Partitioned: in which the dataset is divided based on the target label - node 0 has images with labels 0, 1, 2, and 3; node 1 has 4, 5, and 6; and node 2 has 7, 8, and 9. This dataset is meant to simulate a scenario in which our nodes have data from drastically different distributions – a common setting in federated learning.

- Experiment Plots:
  - No staleness: Equivalent to training one node with the data of all nodes. This is the theoretical optimal performance.
  - Weak staleness: Run using our Zab implementation.
  - Baseline: Each node trains on its own local dataset with no communication.

The plots for the Uniform and Range-Partitioned data split are given in Figures 4 and 5.

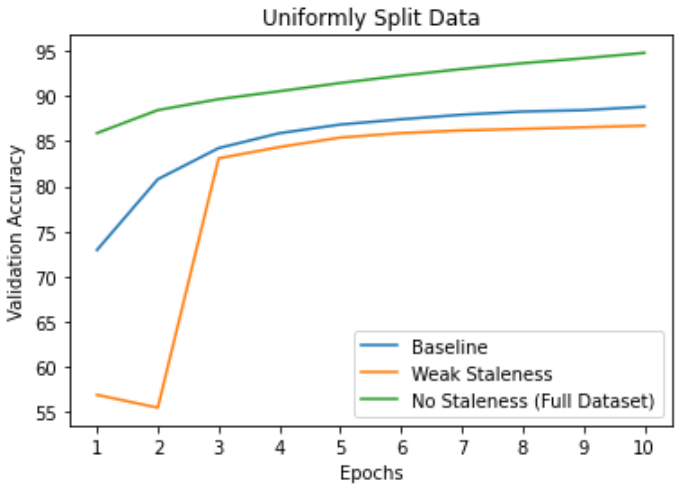

Fig. 4. Machine Learning Experiment Plot for Uniform Data Split

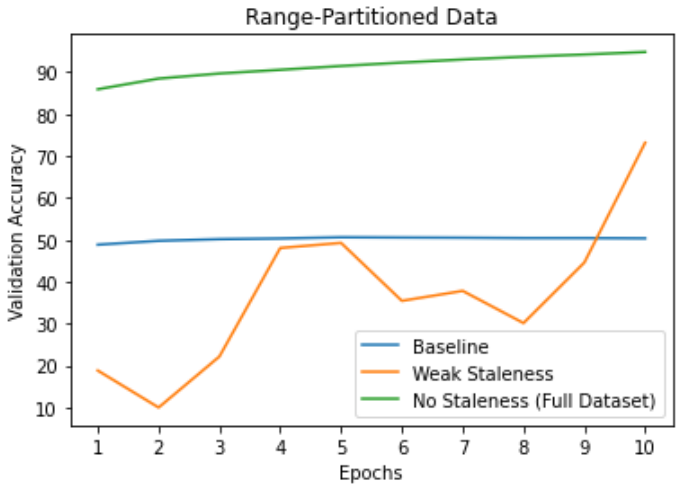

Fig. 5. Machine Learning Experiment Plot for Range-Partitioned Data

Analysis: The plots are consistent with the general trend that we expected them to follow:

- In Uniform split, 'no staleness' performs better than 'baseline' because the latter doesn't see the whole data missing out on important features. 'Weak staleness' has lower accuracy than 'baseline' because of allowing stale gradients.
- In Range-Partitioned data, 'no staleness' is the best by a huge margin because it is not affected by the range partition. 'Weak staleness' becomes noticeably better than baseline by the end of training because its advantage of learning from other node's data exceeds the disadvantage of allowing stale data. We also note the somewhat erratic behavior of training performance and attribute this to stale gradients being applied throughout the training process.

B. Network Communication Cost

In Figure 6, we plot the communication cost vs time to get a better idea of the scale of cost and its trade-off with accuracy. In this experiment:

- Uniform or Range-Partitioned will have identical network costs.
- Baseline variation will have 0 communication cost while no staleness and weak staleness will both have the same cost but weak staleness will have significantly more throughput.

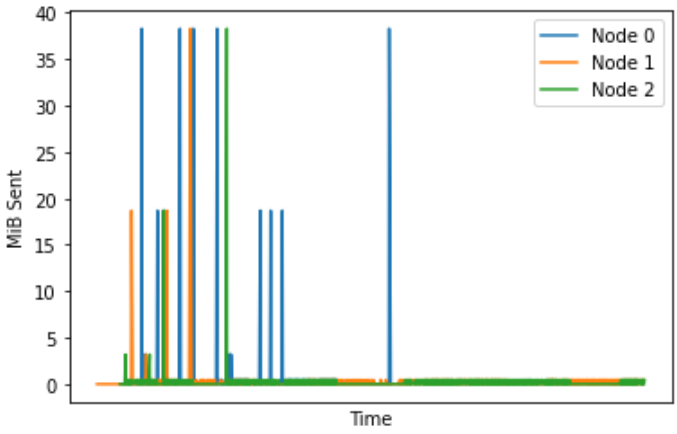

Fig. 6. Communication Cost Plot

Analysis: The plot shows the small but frequent heartbeat messages as well as the infrequent but heavy gradient messages. The gradient message size is reaching almost 40 MiB. Due to these extreme communication costs, we couldn't use deeper neural networks for our experiments. This is a big disadvantage of *FLADS*. While we do suggest some routes for optimization in Section IV, we leave the question of further optimization to future work.

VI. Conclusion

In this paper, we introduced *FLADS*, and discussed its design and implementation details, as well as the choices made during implementation specific to our use case. We introduced the idea of relaxing the constraints on ML model training by allowing stale gradients to be used for SGD.

We ran experiments with different data splits among nodes and different staleness variations to confirm our intuition. The results strongly suggest that using *FLADS* retains the advantage of high training throughput and privacy preservation that we would like to achieve in federated learning scenarios with only marginal decrease in accuracy vs a baseline model. With a skewed data split, the accuracy might even improve as is shown by the Range-Partitioned data split. Such splits are quite common in practice.

References

[1] Hicham Hammouchi, Othmane Cherqi, Ghita Mezzour, Mounir Ghogho, and Mohammed El Koutbi. Digging deeper into data breaches: An exploratory data analysis of hacking breaches over time. *Procedia Computer Science*, 151:1004–1009, 2019. The 10th International Conference on Ambient Systems, Networks and Technologies (ANT 2019) / The 2nd International

Conference on Emerging Data and Industry 4.0 (EDI40 2019) / Affiliated Workshops.

[2] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc' aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc Le, and Andrew Ng. Large scale distributed deep networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[3] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. A survey on distributed machine learning, 2019.

[4] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent, 2011.

[5] Jakub Konečný, H. Brendan McMahan, Felix X. Yu, Peter Richtarik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. In *NIPS Workshop on Private Multi-Party Machine Learning*, 2016.

[6] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data, 2017.

[7] Alireza Fallah, Aryan Mokhtari, and Asuman Ozdaglar. Personalized federated learning: A meta-learning approach, 2020.

[8] Chunjiang Che, Xiaoli Li, Chuan Chen, Xiaoyu He, and Zibin Zheng. A decentralized federated learning framework via committee mechanism with convergence guarantee, 2021.

[9] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256, 2011.

[10] ADRIAN COLYER. Zab: High-performance broadcast for primary-backup systems.