Optimizing Byzantine Adversary-Resistant Federated Learning

Nidhi Jain
nidhij1@cs.cmu.edu

Yuan (Cindy) Jiang
yjiang2@andrew.cmu.edu

Neha Nishikant
nnishika@cs.cmu.edu

***Abstract*—Today's importance of per-device data privacy has led to a growing predominance of federated learning. Training models in this distributed fashion leaves models vulnerable to possibly malicious, or Byzantine, devices. Immediate solutions to this issue involve numerous replicas re-performing the same calculations followed by majority answer selection, resulting in a heavy network communication burden. In this paper, we formulate several Byzantine identification techniques in this distributed training setting to maintain high model performance - without the added network traffic cost.**

I. Introduction

Distributed machine learning distributes data and model training to a cluster of servers to enhance scalability. In the age of big data and increased high-compute resource costs, distributed machine learning is more prevalent than ever. In particular, we consider the federated learning scheme. In federated learning, large amounts of training data are spread across many physically separate clients. A centralized server trains a model across these clients as shown in Figure 1. Due to the impracticality of sending large amounts of data across the network, the clients instead locally train a model on their data and communicate this model to the centralized server. The server aggregates these models and sends back a unified version to all clients. The clients pick up from this starting point and train again. This goes on for a number of rounds until the server is satisfied.

Generally, each client communicates its model to the server by sending the gradients for all of the weights of this model. The server can average these gradients and send them back to the client who can then use these gradients to update their weights. However, in our simulation, we simulate the client's local data by giving it $\frac{|D|}{n}$ randomly selected samples from the training set, where $|D|$ is the size of the training set and n is the number of clients. Thus distribution of data is, in expectation, the same for all the clients. It therefore suffices for the clients to communicate the actual weights themselves to the server instead of the gradients for the weights since the weights across all clients should eventually converge. Our methodology in no way relies on sending weights instead of gradients and can easily be converted to communicate gradients instead for use cases where the data distribution may not be the same across all clients.

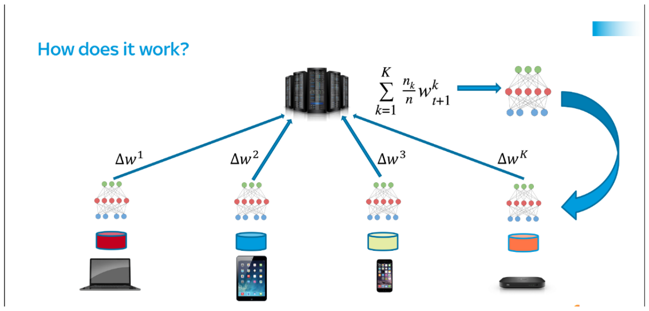


Fig. 1. Federated Learning Methodology

Now, there are many applications for federated learning. For example, it is used in mobile applications where clients are the devices and each client collects data from each user and trains on its local data. It is also used in healthcare, autonomous vehicles, IoT devices, among many other applications.

However, distributed machine learning algorithms often fail to account for fault tolerance and Byzantine generals and "common implementations of these HPC-inspired patterns ... lack fault-tolerance completely" [10]. Byzantine clients can use malicious messages to jeopardize training in the server and hinder convergence. We are motivated to work on Byzantine failures because they are rarely studied in distributed machine learning systems. Byzantine failures are far more dangerous than fail-safe errors due to their detection difficulty, thus emphasizing the importance of making our distributed learning model robust and resilient to these kind of behaviors.

To solve this problem, we propose adding redundancy to our data to accommodate any malfunctioning nodes or lost data in communication. However, with the additional data redundancy, network communication could be a bottleneck of performance and seriously limit the practicality of the system. For example, a common technique for byzantine tolerance may be a consensus based algorithm, but this requires many rounds of communication [5]. Further, since the data is different across each client, results of local model training won't be the same, even between two honest clients, thus limiting the effectiveness of consensus based methods.

This paper focuses on creating a practical byzantine fault tolerant system for distributed machine learning while optimizing for accuracy and maintaining low network communication costs. In particular, we limit all added network communication from the fault free baseline to only server-to-replica communication. There is no client-to-client communication or replica-to-server communication. Further, we don't add any extra rounds of communication between the clients and the servers apart from the baseline. In addition, we reduce computational costs by designating replicas to be "data-only". They do not perform any training computation and simply exist to replicate their primary's model state at given checkpoints.

We introduce three techniques we use for byzantine client handling. First, for Byzantine clients that delay messages, we use a timeout threshold to determine if a client's message is delayed to the server. Second, to detect Byzantine clients which sends malicious messages, we will use anomaly detection techniques to find the clients with model parameters that deviate from the benign clients. Third, we use a importance weighing scheme to adjust the weights of information from different clients, and differentiate the weights of potential Byzantine clients from normal clients.

For the following of this paper, Section II reviews the related works on distributed SGD and Byzantine-fault tolerant federated learning. Section III describes the system overview and its implementation, and discusses in detail the algorithms we use to detect and handle Byzantine clients. Section IV discusses our evaluation plan and explains the experiments and results. Section V concludes the paper.

II. Related Work

Stochastic Gradient Descent (SGD) is a powerful optimization technique that is widely used for solving large-scale machine learning tasks. Training a machine learning model with large datasets could be impractical because it is too slow or does not scale on a single machine. One approach is to distribute the data and training model across a cluster of machines to improve the efficiency and scalability of a machine learning system. TensorFlow is a scalable, distributed training and inference system which can be used for computations of up to hundreds of machines and thousands of devices [1]. The distributed implementation of TensorFlow supports an environment where the client, the master, and the workers can all be in different processes on different machines. Other examples including CNTK use one central server to manage the shared state and parameter updates from different training processes [8].

Modern distributed networks like mobile devices and autonomous vehicles generate a great amount of data every day. As the storage and computational capabilities of devices grow, it is increasingly attractive to leverage these enhanced local resources on devices. In addition to the concerns over transmitting private raw data over the network, there has been growing interest in storing and training data locally on remote devices [7]. Federated learning is a machine learning setting where many clients collaboratively train a model [4]. With significant practical potential, federated learning has recently boomed and highlighted the derived efficiency of pushing model computation to the edge. While keeping data localized, federated learning also eliminates network communication overhead due to transferring datasets. Mobile devices are now used as compute resources to train and generate predictions base off of their local data. Not only does this provide security benefits, but also improve efficiency and limit network communication in the ideal case [9].

While federated learning has many potential benefits, there are still certain challenges to overcome in order to make federated learning robust in a practical setting. A federated learning system is vulnerable to attacks by malicious clients which may send incorrect model updates to degrade learning accuracy or enforce targeted model poisoning attacks to modify the model's behavior [6]. Previous works have developed Byzantine-tolerant algorithms to defend these attacks. For instance, Chen et al. proposed a variant of the classical gradient descent method to tolerate Byzantine failures up to a certain threshold [3]. Other examples include new robust aggregation rules for distributed SGD under a general Byzantine failure model [11]. However, these algorithms are unable to achieve satisfactory model accuracy in the federated learning setting [7]. One possible reason is that these methods do not differentiate the malicious updates from the normal ones, but aim to mitigate the effects of malicious updates. Our goal is to design a machine learning system that can tackle Byzantine-adversary attacks in federated learning while maintaining high training performance.

Blanchard et al. proposed an algorithm for Byzantine-tolerant distributed SGD that is *not* replication-based [2]. The authors dismiss replication-based Byzantine-tolerance due to the potential high communication cost from the added replicas in the system. However, we propose a number of optimizations to reduce network communication cost and hypothesize that in practice, our optimizations will outweigh the additional communication cost due to replication. The advantage our approach has is that the replication-based methods are much easier to implement and analyze compared to the complex algorithm presented in the paper.

III. Methods

A. System Overview

We build a simulation to represent a system capable of performing distributed machine learning. This simulation contains three main components - servers, clients, and replicas (Figure 2). Playing the role of a central node, this server is responsible for spawning the clients, or initiating message

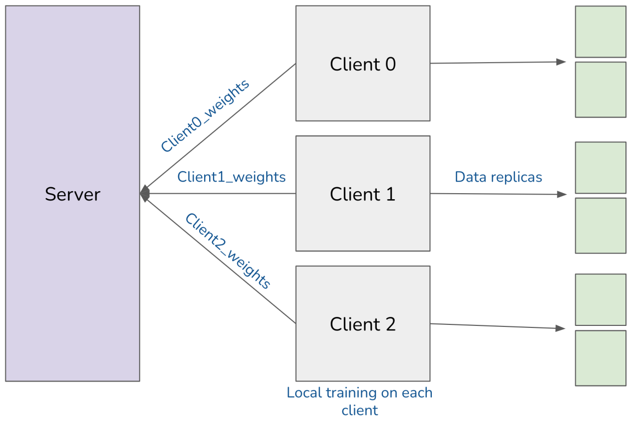


Fig. 2. We utilize a simulation to represent a distributed system. Each client contains a unique subset of data, as well as corresponding replicas that have copies of these respective data subsets. Given messages from these clients sent over a simulated network, the main server that performs aggregation and Byzantine node detection.

communication channels with them. Each client contains its own unique subset of training data. Note that we make an assumption here that each client's data is drawn from the same overall data distribution. In addition to unique data, each client also has corresponding replicas. These replicas are responsible for storing additional copies of the client data and do not perform any computation unless they are labeled as the 'primary.' Upon initialization of the system, clients are chosen at random to be Byzantine without any knowledge of the server. This system serves as the backbone for all computation and communication during the distributed training process.

We now describe the simulation steps of a typical model training event. Upon beginning the simulation, each client trains the machine learning model on its own local data in parallel before sending model results back to the server. In order to sufficiently summarize the model's learning in a given client, the message sent from the client to the server must include the final calculated weights for every parameter in the model. Each client sends one of these carefully crafted messages back to the server after local training. Using Pytorch's Multiprocessing library, we simulate these messages from all the clients being sent in parallel. From the server side, the server now has received model information from each of the clients. The server now uses later-discussed techniques to identify Byzantine clients, and performs an aggregation of all the non-Byzantine clients' weights. The baseline aggregation method is a simple parameter-wise averaging. At this point, the server takes its computed aggregated parameters and sends them all back in parallel to each of the clients. The clients use these weights as their model starting point before retraining locally on their own subsets of data. All adapted client hyperparameters remain frozen between the sending calculated weights and receiving aggregated weights. So, as client local training restarts, the same adapted hyperparameters are used. Upon several rounds of local client training and server aggregation, the model eventually converges. The server then outputs metrics to determine the model success.

B. *Byzantine Client Assumptions*

A byzantine client is capable of adding arbitrarily long delays on any message it sends, and can even indefinitely delay it. On top of that, a byzantine client can send incorrect messages to the server to degrade the model accuracy during aggregation. Incorrect messages are simulated as random weights generated by the Byzantine client.

C. *Byzantine Client Detection - Handling Indefinitely Delayed Clients*

This first technique helps to handle byzantine clients with delayed messages. This is malicious because the server waits for all clients to send their locally trained models before averaging them. A delayed client could cause the server to wait indefinitely and effectively halt the training process.

The simplest solution here is to have a timeout threshold. The server begins the timer once it sends the averaged model back to all the clients. All the honest clients will begin their local training then. At the end of the timeout, if a given client has not communicated their local model back to the server, the server stops waiting for that client's model and instead averages the models it already has received.

The interesting thing here is choosing the timeout threshold and the tradeoffs that come with that choice. Longer timeout thresholds obviously increase the latency of the entire algorithm. However, a timeout that is too short might go off too quickly and miss a honest node that hadn't finished training. That's now one less model for the server to average at this round, and the information loss could hurt accuracy.

We simulate this timeout threshold with a "drop probability" p. p is the probability that, for a given round, a client times out and we do not consider it in the model aggregation. For the pre-selected byzantine nodes, this p is higher to simulate its ability to use arbitraily long delays. For honest nodes, the p is lower. As the timeout threshold increases, p decreases. This is because increasing timeout thresholds increase leniency and decrease the probability that a given client may time out. As we adjust this timeout threshold, the byzantine p and honest p scale together.

D. *Byzantine Client Detection - Identifying Malicious Messages*

The server uses statistical anomaly detection techniques to identify potentially malicious messages sent by clients. We operate under the assumption that Byzantine clients can maliciously alter messages by sending completely randomized parameter weights instead of the expected local training weights. Upon receiving model weights from each of the clients, the server averages the weights for each of the model parameters. Then, for a given client parameter, the server quantifies the Euclidian distance between the client's calculated weights and the average overall for that parameter. We coin this calculated Euclidian distance to be the $deviation$ of the given client's parameter from the average. We repeat this calculation for each of the parameters, for each of the clients, eventually having a list of n deviations per client, where n is the number of model parameters. A given parameter p_i for a client is considered a 'problem' parameter if this parameter's deviation from the average of all clients' respective p_is lies in the top t percent of all client deviations for this given parameter. To classify a client as Byzantine, the client must have at least k problem parameters. We establish t to be a constant value of $t = 0.3$ based on empirical analysis.

To determine k, we consider the expected behavior of the clients' weights. Initially, as training begins, we expect that the weights across clients will be fairly random, and thus expect more parameters of a given honest client to exceed k. However, as training continues and weights reach convergence, we expect that even across clients, weights will be far more similar. Thus, we expect a lower number of parameters of an honest client to exceed k. In order to take advantage of this expected phenomenon, we treat k as an adaptive threshold. As training iterations go on, k decreases, thus increasing harshness and making it more likely for a high weight deviation to flag its corresponding client as Byzantine. We establish k to have a minimum and maximum threshold, and treat these two thresholds as hyperparameters.

Upon identifying a client as Byzantine, the server swaps out the client with a replica, and now establishes this replica as the new primary. The server assumes that this new primary is honest unless future messages flag it as Byzantine via the mentioned identification techniques.

E. Byzantine Client Detection - Importance Weighing Scheme

Importance weighing is a technique we use to adjust the weights of information received from different clients, and differentiate the weights of potential Byzantine clients from honest clients. One challenge we face in removing potential Byzantine clients detected by our server is that if we directly drop the clients with outlier parameters, we may remove potential honest clients which happen to have unsatisfactory trained model parameters in that single round, and our server will lose these honest clients' model parameters during aggregation. To overcome this problem, we provide a varying importance weighing scheme which aggregates the information received from clients with different weights. If a client is identified as an outlier for its model parameters, the server decreases the weights of its parameters in every round. On the other hand, normal clients' weights are kept unchanged over each round.

The major concern this optimization tries to solve is that if our server is too strict with dropping clients with outlier weights, we might lose honest clients which are valuable to achieve satisfactory model training performance. However, this leniency may let our server tolerate actual byzantine clients, therefore our optimization is to lower the weight of model parameters from these outlier clients so that their results would not interfere with the aggregation of remaining clients by much and our system can still achieve promising learning performance upon convergence.

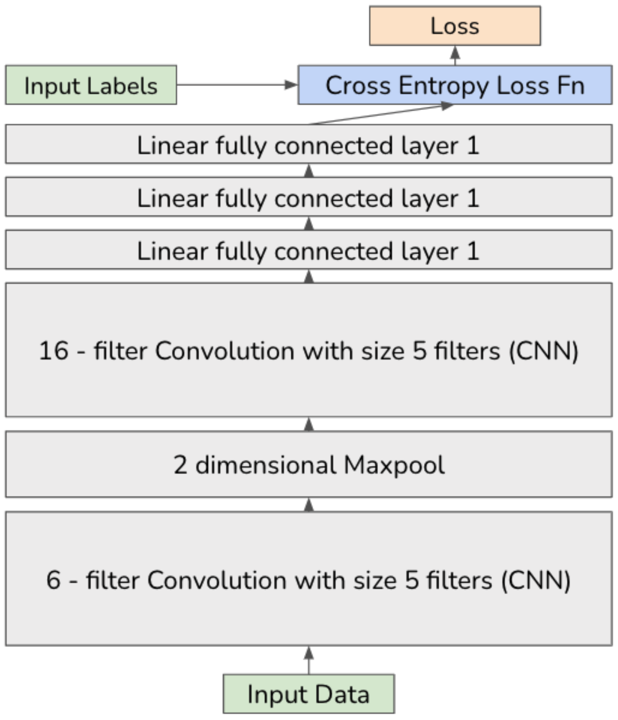

Loss
Input Labels
Cross Entropy Loss Fn
Linear fully connected layer 1
Linear fully connected layer 1
Linear fully connected layer 1
16 - filter Convolution with size 5 filters (CNN)
2 dimensional Maxpool
6 - filter Convolution with size 5 filters (CNN)
Input Data

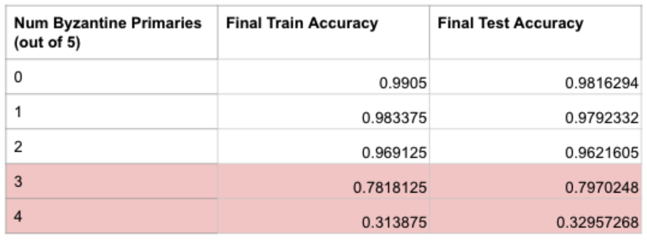
Fig. 3. We use a multi layer deep learning model for training.

IV. Evaluation

A. Model and Dataset

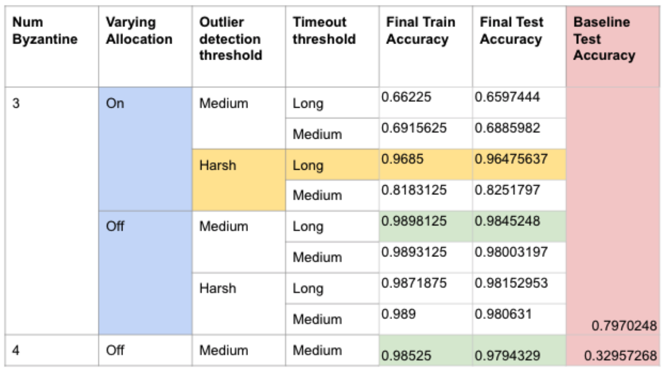
The use case we chose to focus on is the MNIST dataset for handwritten digit recognition. MNIST contains $60,000$ training examples, and $10,000$ testing examples. The data contains input as normalized and centered images of handwritten digits, and labels as the ground truth integer values corresponding to the images. MNIST is a widely used and a very popular basic benchmark due to its simplicity in preprocessing. Despite its dataset simplicity, model training still remains a challenging task that requires deep learning methods. As a result, we choose to use this dataset due to both its medium task complexity that may resemble typical use cases more as well as the dataset's ease of use.

Our model is a 6 layer architecture (Figure 3) with 2 CNNs, 3 fully connected layers, and 1 max pool layer. We train using Cross Entropy Loss with a Stochastic Gradient Descent optimizer and initial learning rate of 0.01, batch size of 32, and 20 epochs across 5 primary clients. We use $16,000$



Num Byzantine Primaries (out of 5)	Final Train Accuracy	Final Test Accuracy
0	0.9905	0.9816294
1	0.983375	0.9792332
2	0.969125	0.9621605
3	0.7818125	0.7970248
4	0.313875	0.32957268

Fig. 4. We experiment with a varying number of Byzantine clients out of 5 without optimizations and observe the model performance.



Num Byzantine	Varying Allocation	Outlier detection threshold	Timeout threshold	Final Train Accuracy	Final Test Accuracy	Baseline Test Accuracy
3	On	Medium	Long	0.66225	0.6597444	
			Medium	0.6915625	0.6885982	
		Harsh	Long	0.9685	0.96475637	
			Medium	0.8183125	0.8251797	
	Off	Medium	Long	0.9898125	0.9845248	
			Medium	0.9893125	0.98003197	
		Harsh	Long	0.9871875	0.98152953	
			Medium	0.989	0.980631	0.7970248
4	Off	Medium	Medium	0.98525	0.9794329	0.32957268

Fig. 5. We perform several experiments that each involve a different set of tuned optimization hyperparameters.

training samples and test on $10,000$ samples. We do not use any additional model optimizations ($i.e.$ momentum, weight decay).

B. Adjustable hyperparameters

For all of our experiments, we use 5 clients and 5 total training rounds. Each round runs 10 epochs per client.

One of the most important adjustable hyperparameter is the number of Byzantine clients in the system. These Byzantine clients are the primaries of the replicas. We allow 0-all Byzantine clients in the system.

For "Handling Indefinitely Delayed Clients", there is a timeout threshold for detecting Byzantine clients that are delaying messages, the criteria for our experiments range from short, medium, or long. These categories correspond with byzantine node drop probabilities of 0.3, 0.5, and 1.0. If a client's message is delayed for more than the amount of threshold, the client is identified as Byzantine.

For "Identifying Malicious Messages", there is an outlier detection threshold. While this threshold is adaptive within a given run, the final threshold it comes to can be varied. We perform experiments with low, medium, and high final outlier harshness thresholds. This corresponds to marking an entire *client* as an outlier once we hit 70%, 50%, and 30% (respectively) outlying *parameters* for that client. In our case, we have 10 parameters in our model so this corresponds to 7, 5, and 3 outlying parameters respectively.

For "Importance Weighting Scheme", we can either toggle that to be on or off. If it is off, then if a client times out or is considered an outlier, it is immediately classified as byzantine and thus not included in the aggregation of that round as well as dropped in favor of its replica for the next rounds. If it is on, then the client isn't dropped and its weight in the weighted aggregation simply goes from 1 to 0.1.

C. Baselines

In our baselines (Fig 4), we experiment with a varying number of Byzantine clients out of a total of 5 clients. We also measure success via the final train and test accuracy.

We observe model performance with a varying number of Byzantine clients without any optimizations. We notice that before hitting 3 Byzantine clients, we are able to achieve decent accuracy, although decreasing. After 3 Byzantine clients, however, we see a steep drop. Note that due to the non deterministic nature of training via weight randomization, the number of Byzantine clients before the steep drop varies between 2 and 3. We focus our remaining experiments to evaluate our optimization techniques on the cases of 3 and 4 Byzantine primaries.

D. Improvements

In Figure 5, we highlight the green cells to show the model's performance after our byzantine detection. As seen from the comparison between the baseline (red column) and a given configuration from our model (green cells), we show serious improvement, comparable to having no byzantine nodes.

Another observation to note of is the blue column. Interestingly, having the importance weighting scheme off produces overall better results than having it on. Within having it on, the yellow cells highlight the best configuration. With a harsh outlier detection threshold and a long timeout threshold, the accuracy is back up in comparison to the other configurations with importance weighting scheme off. Our proposed explanation for this is that since we are being more lenient once we find a byzantine node (since we only lower the weight instead of fully dropping it), we must be stricter in finding the byzantine nodes. Thus the harsh outlier detection performs well. Further, we cannot afford to lose honest nodes for aggregation due to this leniency with byzantine nodes, which explains the high performance with a long timeout threshold. In the future, a different scheme to lower the weights might prove to have better results. For example, we may choose to half the weight each time a node is considered byzantine and eventually drop it once it falls beneath a certain threshold. This introduces yet another hyperparameter subject to tuning.

E. Accuracy figure

Figure 6 shows the train accuracies for a given client for each round, before and after training. The train accuracy is

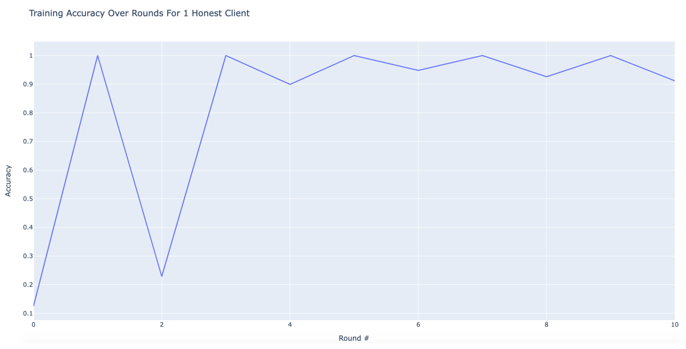


Fig. 6. Local train accuracies for client 1030. Last point is final train accuracy on entire train set

computed on the client's subset of the training data. The accuracy starts out very low. Then it climbs fast within a round, before experiencing an expected drop after aggregation. We expect this drop to occur because the model is now more optimized for a general set rather than for the client's specific subset. This drop is reflected in the accuracy before the next round of training starts. The drops become lower and lower as the rounds increase since the weights across clients converge. The last dip is attributed to the very last aggregation and the fact that it is the train accuracy on the entire train set.

V. Conclusion

The paper presented the design and implementation of a distributed machine learning system which can tolerate Byzantine clients. These malicious clients can delay, possibly indefinitely, messages or send corrupted messages to the server to degrade model learning performance. We developed an algorithm to detect Byzantine clients and swap out these clients with their replicas. We implemented three optimizations to perform byzantine detection without exploding network communication that could come with the addition of replicas. Our model achieves high training and testing accuracy while maintaining low network overhead.

Future work could explore if it is possible to simulate byzantine nodes with intelligently calculated weights instead of random weights. If so, how does that affect performance? Further, server failures could be explored and protected against using a decentralized structure and a consensus algorithm for checking aggregation. This would make our system even more robust to failure and adversaries.

Acknowledgment

We would like to thank our professor Phil Gibbons and our TA Juncheng (Jason) Yang for their guidance and support throughout this project.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Z. Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Christopher Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *ArXiv*, abs/1603.04467, 2016.

[2] Peva Blanchard, El Mahdi El Mhamdi, Rachid Guerraoui, and Julien Stainer. Machine learning with adversaries: Byzantine tolerant gradient descent. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[3] Yudong Chen, Lili Su, and Jiaming Xu. Distributed statistical machine learning in adversarial settings: Byzantine gradient descent. *ACM SIGMETRICS Performance Evaluation Review*, 2019.

[4] Peter Kairouz, H. Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, Rafael G. L. D'Oliveira, Hubert Eichner, Salim El Rouayheb, David Evans, Josh Gardner, Zachary Garrett, Adrià Gascón, Badih Ghazi, Phillip B. Gibbons, Marco Gruteser, Zaid Harchaoui, Chaoyang He, Lie He, Zhouyuan Huo, Ben Hutchinson, Justin Hsu, Martin Jaggi, Tara Javidi, Gauri Joshi, Mikhail Khodak, Jakub Konecný, Aleksandra Korolova, Farinaz Koushanfar, Sanmi Koyejo, Tancrède Lepoint, Yang Liu, Prateek Mittal, Mehryar Mohri, Richard Nock, Ayfer Özgür, Rasmus Pagh, Hang Qi, Daniel Ramage, Ramesh Raskar, Mariana Raykova, Dawn Song, Weikang Song, Sebastian U. Stich, Ziteng Sun, Ananda Theertha Suresh, Florian Tramèr, Praneeth Vepakomma, Jianyu Wang, Li Xiong, Zheng Xu, Qiang Yang, Felix X. Yu, Han Yu, and Sen Zhao. Advances and open problems in federated learning. *Foundations and Trends® in Machine Learning*, 14(1–2):1–210, 2021.

[5] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, December 2001.

[6] Suyi Li, Yong Cheng, Wei Wang, Yang Liu, and Tianjian Chen. Learning to detect malicious clients for robust federated learning. *arXiv preprint arXiv:2002.00211*, 2020.

[7] Tian Li, Anit Kumar Sahu, Ameet S. Talwalkar, and Virginia Smith. Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine*, 37:50–60, 2020.

[8] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In *NIPS*, 2017.

[9] Brendan McMahan and Daniel Ramage. Federated learning: Collaborative machine learning without centralized training data, Apr 2017.

[10] Joost Verbraeken, Matthijs Wolting, Jonathan Katzy, Jeroen Kloppenburg, Tim Verbelen, and Jan S. Rellermeyer. A survey on distributed machine learning. *CoRR*, abs/1912.09789, 2019.

[11] Cong Xie, Oluwasanmi Koyejo, and Indranil Gupta. Generalized byzantine-tolerant sgd. *ArXiv*, abs/1802.10116, 2018.