Efficient Detection of Byzantine Attacks in Federated Learning using Last Layer Biases

Najeeb Jebreel, Alberto Blanco-Justicia, David Sánchez, and Josep Domingo-Ferrer

Universitat Rovira i Virgili,
Department of Computer Engineering and Mathematics,
CYBERCAT-Center for Cybersecurity Research of Catalonia,
UNESCO Chair in Data Privacy,
Av. Paos Catalans 26, 43007 Tarragona, Catalonia
{najeebmoharramsalim.jebreel,alberto.blanco,david.sanchez,
josep.domingo}@urv.cat

Abstract. Federated learning (FL) is an alternative to centralized machine learning (ML) that builds a model across multiple decentralized edge devices (a.k.a. workers) that own the training data. This has two advantages: i) the data used for training are not uploaded to the server and ii) the server can distribute the training load across the workers instead of using its own resources. However, due to the distributed nature of FL, the server has no control over the behaviors of the workers. Malicious workers can, therefore, orchestrate different kinds of attacks against FL. Byzantine attacks are amongst the most common and straightforward attacks on FL. They try to prevent FL models from converging by uploading random updates. Several techniques have been proposed to detect such kind of attacks, but they usually entail a high cost for the server. This hampers one of the main benefits of FL, which is load reduction. In this work, we propose a highly efficient approach to detect workers that try to perform Byzantine FL attacks. In particular, we analyze the last layer biases of deep learning (DL) models on the server side to detect malicious workers. We evaluate our approach with two deep learning models on the MNIST and CIFAR-10 data sets. Experimental results show that our approach significantly outperforms current methods in runtime while providing similar attack detection accuracy.

Keywords: Federated learning · Security · Byzantine attacks

1 Introduction

Machine learning (ML) is extensively employed in areas such as computer vision, voice processing or natural language processing [11]. However, ML algorithms are highly *data-hungry*; that is, they require a large number of observations to reach high accuracy levels [10], and thus, substantial training costs. Intelligent devices, such as smartphones, produce rich raw data in terms of quantity and quality and have ample unused storage and computing capabilities. Federated

learning (FL) is an alternative to centralized ML that leverages the local data and the free computing resources of edge devices to train models in a distributed way [9]. Specifically, in FL a global shared model is learned cooperatively by edge devices (a.k.a. workers) without sharing their local private data. In this way, FL significantly decreases the training costs at the server side while providing privacy by design, because the clients' data do not need to be uploaded to the server.

An example application of FL is the generation of predictive text for smartphone keyboard apps. In this scenario, a service provider (the server) trains a predictive text model from a publicly available text corpus and distributes it among its users (the workers), along with the keyboard app. The keyboard app is capable of adapting (in other words, updating) its locally stored model with the user's inputs, so that it is able to predict the text that best resembles the way each user writes. From time to time, the service provider will randomly select a subset of its user base and collect their updated models. Then, the service provider will aggregate these updated models using some aggregation algorithm to obtain a new predictive text model which better matches the way its user base writes (the way groups of people write changes over time). The federated averaging (*FedAvg*) algorithm is widely used, which takes either the average of the difference between the global model and the locally updated models or the average of the local models themselves [5,9,13]. This training process is repeated over time, potentially indefinitely.

Although FL systems provide privacy and autonomy by design for workers, that same autonomy may be exploited by malicious adversaries. Since the server has no control over the behavior of the workers, any of them may deviate from the prescribed training protocol to conduct attacks such as Byzantine attacks [2]. In Byzantine attacks, attackers try to prevent the model from converging by sending maliciously crafted updates. In [3, 4] the authors show that a single malicious worker can significantly degrade the accuracy of the model and even prevent the model from converging.

Several methods have been proposed to counter Byzantine attacks targeting FL. In [12] the authors propose to use the coordinate-wise median of the workers' updates to filter out malicious updates. Specifically, let w_t be the global model parameters at time t and $w_t^{(k)}$ be the local model parameters from worker k at time t. Additionally, let $w[i]$ refer to the value of w at coordinate i (a specific weight or bias). The server sets each global parameter as $w_t[i] = median(\{w_t^{(k)}[i]\}_{k=1}^{m})$, where m is the number of participating workers at time t. Alternative methods, named *Krum* and *Multi-Krum*, were proposed in [4]. The Krum method obtains a new global model by selecting the local model that is most similar to the other local models received at a certain training round. For each $w^{(k)}$ (and assuming m' malicious workers in the federated learning system), Krum computes the sum of the distances between $w^{(k)}$ and its closest $m - m'$ local models. Then, Krum selects the local model with the smallest sum of distances as the new global model. In Multi-Krum, the $(m-m')$

local models with the smallest sum of distances are aggregated to yield the new global model.

All these defense methods are robust against Byzantine attacks up to a certain number of malicious workers. However, they incur high computational costs at the server side for large ML models due to the number of parameters to be analyzed (for example, the number of parameters in the ResNet deep learning model is about 25.6 million [6]). On top of that, the number of workers involved in federated learning systems may also reach tens of millions. Since these methods analyze *all* the model parameters for each worker update, the cost at the server side grows unaffordably with the size of the models and the number of involved workers. This goes against one of the fundamental motivations of FL, which is to alleviate the computational cost at the server side.

Contributions and plan of this paper

In this paper, we introduce a novel approach for detecting Byzantine attacks in FL with minimum computational cost at the server side. Our approach aims to detect and eliminate malicious workers' updates at every training round by using only the last layer biases of deep learning models. The intuitive idea is that, instead of analyzing the whole updates, it is possible to just analyze a small fraction of the updates that holds enough information to differentiate honest from malicious workers. In deep learning models (e.g. image classification models), the early layers extract features, while the final decision depends on the latter layers [6]. We exploit this to develop a robust aggregation method that achieves accurate detection of malicious workers with low computational cost. Specifically, for each worker, whatever the worker's update size is, we only compute the distance between the worker's biases and the geometric median of all workers' last layer biases. We use these distances to detect and filter out a malicious worker's update. As far as we know, this is the first study that applies a detection approach using the last layer biases in deep learning models.

We tested our method with two well-known data sets (MNIST and CIFAR-10) and by using two different architectures of deep learning models. We also compared our approach with three state-of-the-art methods: Median, Krum and Multi-Krum. Our results show that our method achieves similar attack detection performance as the best baseline (Multi-Krum) while significantly reducing the runtime required to verify and aggregate updates at the server side at each training round.

The remainder of the paper is organized as follows. Section 2 gives background on federated learning and deep learning. Section 3 details our approach. Section 4 describes and reports the results of our experiments. Finally, Section 5 gathers conclusions and proposes several lines of future research.

2 Background

2.1 Federated averaging

In FL, the federated averaging algorithm (*FedAvg*) [9] iteratively computes a new shared global model, which is the result of averaging local models that have been trained on the workers' remote devices. The training process is coordinated by a central server that updates and distributes the shared global model w_t. More specifically, the server starts by randomly initializing the global model w_t. Then, at each training round of *FedAvg*, the server selects a subset of workers S of size $C \cdot K \geq 1$ where K is the total number of workers in the system and C is the fraction of workers that are selected in the training round. After that, the server distributes the current global model w_t to all workers in S. Besides the global model, the server sends a set of hyper-parameters to be used at the workers' side to train their model locally. The server sends the number of local epochs E, the local batch size BS and a learning rate η. After receiving the new shared model w_t, each worker divides its local data into batches of size BS and performs E local training epochs of stochastic gradient descent (SGD). Finally, workers upload their updated local models $w_{t+1}^{(k)}$ to the server, which then computes the new global model w_{t+1} by averaging all received local models. The aggregation framework in *FedAvg* is given by:

$$w_{t+1} = \sum_{k=1}^{K} \frac{n^{(k)}}{n} w_{t+1}^{(k)}. \tag{1}$$

Here, n denotes the total number of data points at K workers such that $\sum_{k=1}^{K} n^{(k)} = n$, and $n^{(k)}$ refers to the number of data points at the worker k.

2.2 Bias in neural networks

The biological neuron is known to fire only when its processed input exceeds a certain threshold value. The same mechanism is present in artificial neural networks [7], where the bias (a.k.a. threshold) is used to control the triggering of the activation function, i.e, it is used to tune the output along with the weighted sum of the inputs to the neuron as shown in Figure 1. Thus, the bias helps the model in a way that it can best fit the given data.

Accordingly, biases for models sharing the same architecture and trained by several workers are assumed to be similar. This assumption is based on the similarity of the distribution of the local private data of each worker, and on the identical structure of the model and the hyper-parameters used during the training to optimize the model parameters. In our work, we take advantage of the similarity of the last layer biases of the honest workers' neural networks, which we use to differentiate malicious workers from honest ones. This is illustrated in Figure 2, which shows how the biases for honest workers are close together, while the biases of malicious workers providing random updates follow a random and uncorrelated pattern.

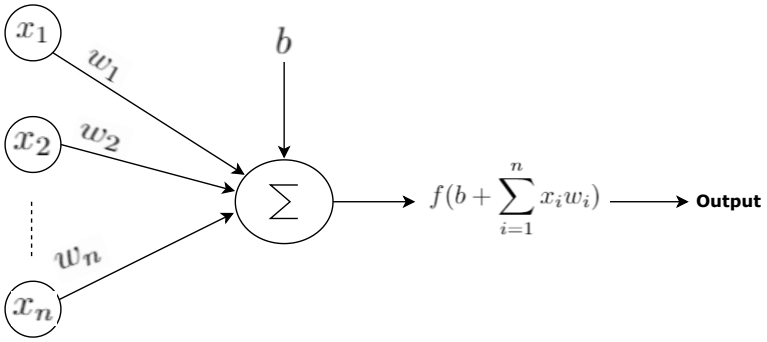


Fig. 1. Example of a neuron, showing inputs $x_1, ..., x_n$, weights $w_1, ..., w_n$, bias b and an activation function f applied to the weighted sum to get the output.

We assume that, if there is a set $\{w^{(1)}, w^{(2)}, \ldots, w^{(K)}\}$ of workers' model updates and each $w^{(k)}$ has $L^{(k)}$ layers, there is a last layer $l_{-1}^{(k)} \in L^{(k)}$ that contains n neurons and a bias vector $b^{(k)} \in \mathbb{R}^n$. Then, let B be the set that contains all the last layer vectors $b^{(1)}, b^{(2)}, \ldots, b^{(K)}$ of the K workers' updates.

2.3 Geometric median

For a given set of m points $x_1, x_2, ..., x_m$ with $x_i \in \mathbb{R}^n$, the geometric median is defined as:

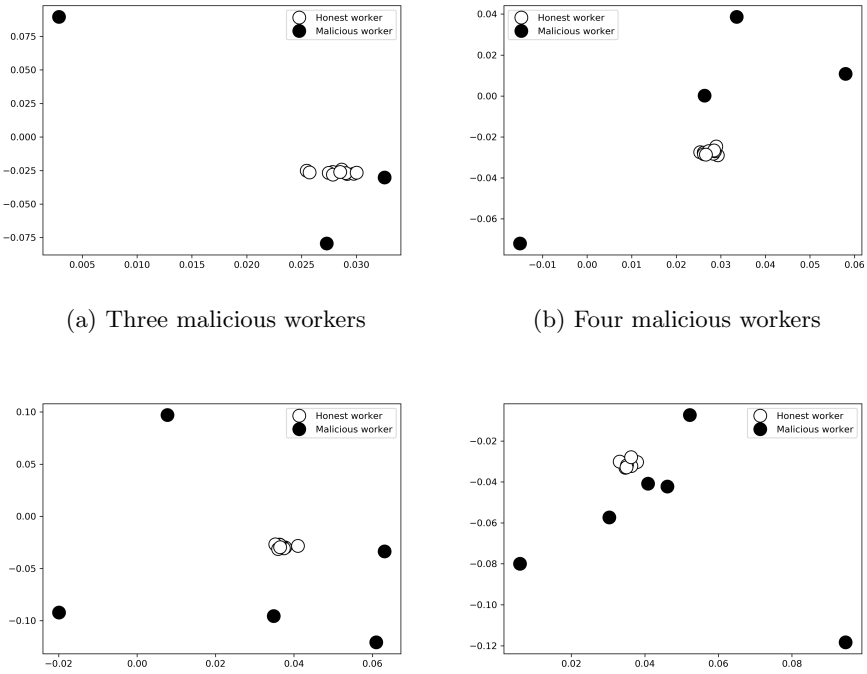
$$\underset{y \in \mathbb{R}^n}{\operatorname{argmin}} \sum_{i=1}^{m} \|x_i - y\|_2 , \tag{2}$$

where the value of the argument y is the point that minimizes the sum of Euclidean distances from the x_i's to y. The geometric median does not change if up to half of the sample data are arbitrarily corrupted [8]. Therefore, it provides a robust way for identifying the biases of honest workers, because the geometric median will be closer to the majority who are honest.

3 Efficient detection of Byzantine attacks

We consider a typical FL scenario where there is one server who uses the *FedAvg* algorithm [9] to coordinate the training process of K workers. The workers cooperatively train a shared global model w by using their local data. A fraction of such workers may be malicious attackers. Honest workers compute their updates correctly and send the result to the server. However, malicious workers may deviate from the prescribed protocols and send random updates aiming at degrading the global model.

The key idea of our approach is that, at each global training round, the server assigns a score to each participating worker and uses that score to consider or



(a) Three malicious workers

(b) Four malicious workers

(c) Five malicious workers

(d) Six malicious workers

Fig. 2. Graphical representation of last layer biases

not the worker's update at the aggregation phase. Unlike state-of-art methods, we do not analyze the entire workers' updates. Instead, we focus only on the last layer biases of workers' updates to filter out malicious updates.

Algorithm 1 formalizes the methodology we propose. The algorithm has a set of hyper-parameters K, C, BS, E, η and τ. K is the number of participating workers, C is the fraction of workers to be selected at each round, BS is the local batch size, E the number of local epochs, η is the learning rate and τ is the confidence value used to detect malicious workers.

At the first global training round, the server starts a federated learning task by initializing the global model w_0. Then it selects a random set S of m workers, where $m = max(C \cdot K, 1)$, transfers w_t to each worker in S, and asks them to train the model (locally) using the defined hyper-parameters. Each worker trains the model using her private local data and transfers the updates $w_{t+1}^{(k)}$ to the server, who aggregates updates to create a new global model for the next training round w_{t+1}.

The server then decides whether a worker is honest or malicious by using a scoring function. The scoring function **GetScores** receives the last layer biases

B of all the updates of the workers, and computes the geometric median $GeoMed$ of the layer biases $b^{(k)}$ for each $k \in S$. After that, it finds the set of distances $Dist$ between the $GeoMed$ and each $b^{(k)}$. Then, the server sorts $Dist$ and computes $Q1, Q3, IQR = Q3 - Q1$. Next, by using the hyper-parameter τ, the scoring function assigns 1 to a worker k if $dist_k \leq Q3 + \tau \times IQR$; else 0.

Once the server has obtained the set $Scores$, it performs the aggregation phase of updates $w_{t+1}^{(k)}$ for each $k \in S$ such that $score^{(k)} = 1$. We propose replacing Equation (1) in *FedAvg* by $w_{t+1} = \sum_{k \in S, score^{(k)}=1} \frac{n^{(k)}}{sn} w_{t+1}^{(k)}$, where $sn = \sum_{k \in S, score^{(k)}=1} n^{(k)}$. Note that, since $\sum_{k \in S, score^{(k)}=1} \frac{n^{(k)}}{sn} = 1$, the convergence of the proposed averaging procedure at the server side is guaranteed as long as *FedAvg* converges.

4 Experimental results

Experiments have been carried out on two public data sets: MNIST and CIFAR-10. MNIST consists of 70,000 images of handwritten digits (which correspond to classes), from 0 to 9. Each image is gray-scaled and 28×28 pixels in size. We used 60,000 of them for training and the remaining 10,000 for testing. The CIFAR-10 dataset contains of 60,000 32×32 colored images of 10 classes. Each class has 6,000 images. We used 50,000 images for training and the remaining 10,000 for testing. For both datasets, we divided the training data set into 100 equally sized shards, which were distributed among 100 simulated workers.

For MNIST, we used a small DL model with 21,840 trainable parameters. The earlier layers were convolutional layers that extracted the features and passed them to the last two fully connected layers: fully connected layer 1 (FC1), and fully connected layer 2 (FC2). FC1 contained 50 neurons with 1 bias for each neuron, while FC2 contained 10 neurons with 1 bias for each neuron. The output of FC2 was fed to a SoftMax layer that introduced the final probabilities of the 10 classes. For CIFAR-10, we used a large DL model that contained 14,728,266 trainable parameters in total. The earlier layers extracted the features and they had only one fully connected layer (FC1) that contained 10 neurons with 1 bias for each neuron. The output of the fully connected layer was fed to a SoftMax layer that introduced the final probabilities of the 10 classes.

We trained the small DL model for 30 global rounds. Each worker trained the model locally using the stochastic gradient descent (SGD) with 5 local epochs, local batch size = 20, learning rate = 0.001, and momentum = 0.9. We trained the large DL model for 60 global rounds. Each worker trained the model locally using the stochastic gradient descent (SGD) with 5 local epochs, local batch size = 50, learning rate = 0.01 and momentum = 0.9.

For both models, we used the precision as a performance metric. Precision is the amount of true positives TP divided by the sum of true positives and false positives FP, that is $TP/(TP + FP)$.

In the experiments the server selected a fraction $C = 15\%$ of workers at each round. We limited the number of malicious updates per round between

Algorithm 1: Byzantine-attack robust aggregation of updates using last layer biases scores.

Input: K, C, BS, E, η, τ
initialize w_0
for *each round t = 0, 1,...* **do**
  $m \leftarrow \max([C \cdot K], 1)$
  $S =$ random set of m workers.
  for *each worker* $k \in S$ ***in parallel*** **do**
    $w_{t+1}^{(k)} =$ `WorkerUpdate(`k, w_t`)`
  end
  $Scores =$ `GetScores(`S`)`
  $w_{t+1} = \sum_{k \in S, score^{(k)}=1} \frac{n^{(k)}}{sn} w_{t+1}^{(k)}$ where $sn = \sum_{k \in S, score^{(k)}=1} n^{(k)}$
end
Function *WorkerUpdate*(k, w_t)
  $w \leftarrow w_t$
  for *each local epoch* $i = 1, 2, \ldots E$ **do**
    for *each batch* β *of size* BS **do**
      $w \leftarrow w - \eta \nabla L(w, \beta)$
    end
  end
  return w
end
Function *GetScores*(S)
  Let B be the set of biases of the last layers for each worker $k \in S$
  $GeoMed =$ *GeometricMedian*(B)
  $Dist =$ the distances between the $GeoMed$ and each $b^{(k)} \in B$
  Compute $Q1, Q3$ of $Dist$
  $IQR = Q3 - Q1$
  $Scores = [\,]$
  for *each* $k \in S$ **do**
    if $dist_k \leq Q3 + \tau \times IQR$ **then**
      Add(1, Scores)
    else
      Add(0, Scores)
    end
  end
  return Scores
end

1 to 6. This is consistent with the Byzantine resilience property, which states that a method is considered to be Byzantine resilient if it is robust against m' Byzantine workers where $(2m' + 2) < m$ [1].

Figure 3 shows the evolution of the classification precision achieved by the small DL model at each round when no detection is performed. The dashed line shows the precision when all the workers acted honestly. The global model achieves a precision around 96% after 30 rounds. In addition to the all-honest

case, we considered different scenarios where a fraction of the workers were malicious. In our experiments, malicious workers trained their model honestly and, after that, they added some random noise to each parameter in their model from a Gaussian distribution with mean 0 and standard deviation 0.5 for the small DL model, and with mean of 0 and standard deviation 0.2 for the large DL model. It is noticeable that the precision of the model decreases as the number of malicious updates per round increases.

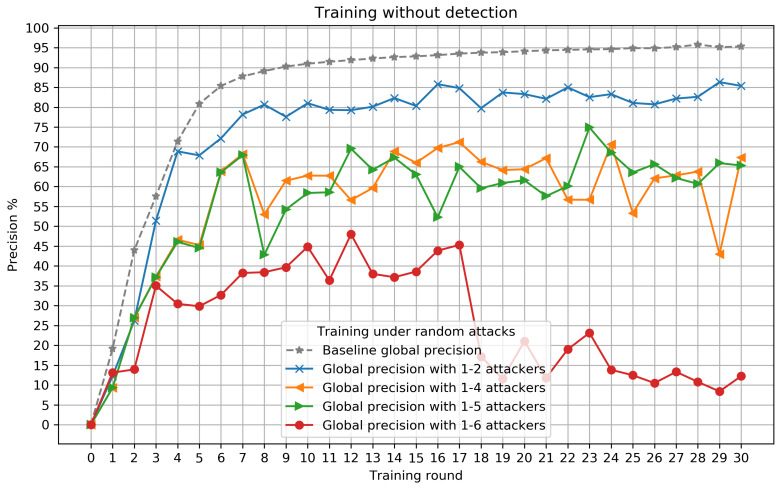


Fig. 3. Evolution of the small DL model precision with MNIST under random attacks when no detection is performed

We next considered training with detection of malicious workers. We chose τ to be -0.5 in this case, that is, we consider each worker has a distance greater than the median to be malicious. We compared the results of our method with three state-of-art methods: Median, Krum, and Multi-Krum. Figure 4 shows the precision of the small DL model when subjected to up to 6 random updates per round. We can see that both Median and Krum have similar performance, and that both Multi-Krum and our method outperform them and obtain virtually identical results. The reason behind this identical performance is that our method takes the most honest set of updates, like Multi-Krum.

In Figure 5 we show the performance of our method with the large DL model and CIFAR-10 w.r.t. the values of τ. The dashed line shows the baseline precision when all workers acted honestly, which is around 84% after 60 rounds. With $\tau = 1$ our method filtered out most of the malicious workers when they were far enough from the majority. However, we can see that between training rounds 22 and 31

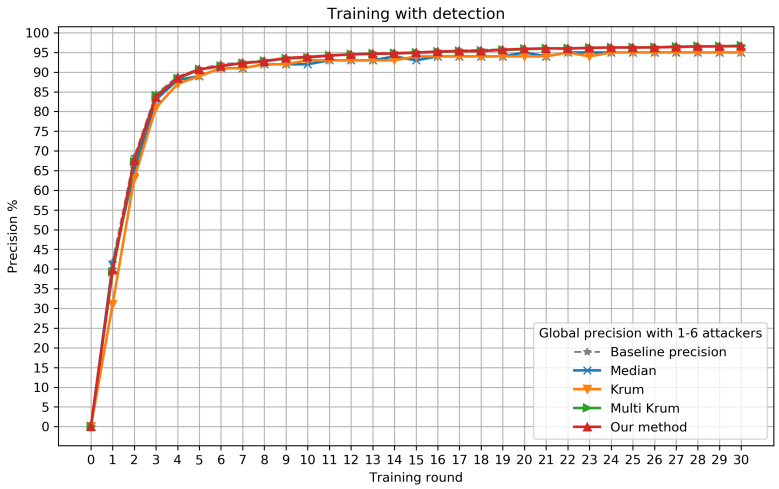


Fig. 4. Evolution of the small DL model precision with MNIST under random attacks with detection

some malicious workers were able to escape detection, and that their effect in the model precision was devastating. With $\tau = 0.5$ most malicious workers were eliminated and with $\tau = -0.5$, all of them were eliminated.

Even more relevant, Figure 6 shows that, in addition to our method offering detection as good as state-of-the-art methods, it is also significantly more *computationally efficient* than its counterparts. The reason is that for each worker our method only focuses on the last layer parameters.

5 Conclusions and Future Work

We have shown that by focusing on a specific aspect of the learning model (biases) we can efficiently detect Byzantine attacks in FL. As shown in the experiments, our method achieves state-of-the-art accuracy while significantly decreasing the runtime of the attack detection at the server side. The scalability of our method is particularly useful when using large DL models with millions of parameters and workers. Although our results are promising, choosing the value of τ is challenging. The increase or the decrease of the value of τ must depend on the proportion of the malicious workers in the system. The higher the expected number of malicious workers the smaller the value of τ.

As future work, we plan to study the possibility of using our method to detect other kinds of attacks to FL, such as targeted and stealthy attacks.

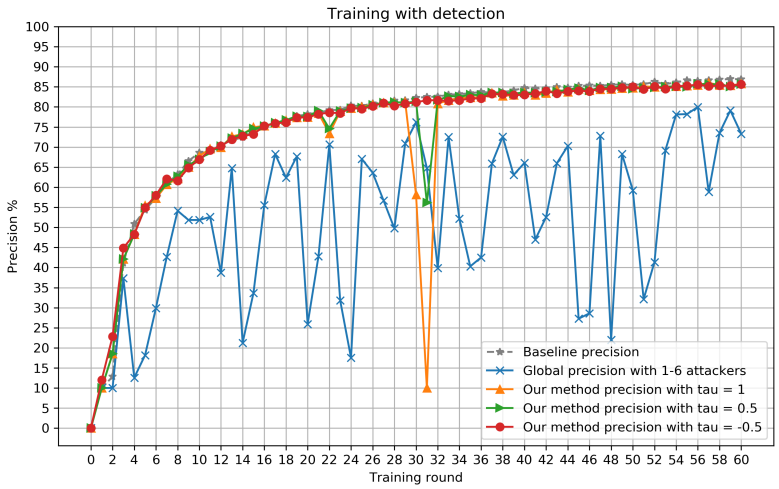


Fig. 5. Evolution of the large DL model precision with CIFAR-10 under random attacks with detection

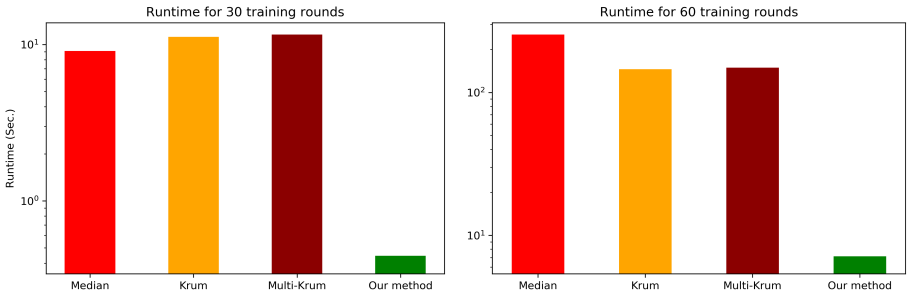


Fig. 6. Runtime (in seconds) for each method: small DL model on the left and large DL model on the right.

Acknowledgements

Partial support is acknowledged from the European Commission (project H2020-871042 SoBigData++), the Government of Catalonia (ICREA Acadèmia Prize to J. Domingo-Ferrer and grant 2017 SGR 705), and the Spanish Government (projects RTI2018-095094-B-C21 "Consent" and TIN2016-80250-R "Sec-MCloud"). The authors are with the UNESCO Chair in Data Privacy, but the views in this paper are their own and are not necessarily shared by UNESCO.

References

1. Alistarh, D., Allen-Zhu, Z., Li, J.: Byzantine stochastic gradient descent. In: Advances in Neural Information Processing Systems. pp. 4613–4623 (2018)
2. Bagdasaryan, E., Veit, A., Hua, Y., Estrin, D., Shmatikov, V.: How to backdoor federated learning. arXiv preprint arXiv:1807.00459 (2018)
3. Bhagoji, A.N., Chakraborty, S., Mittal, P., Calo, S.: Analyzing federated learning through an adversarial lens. arXiv preprint arXiv:1811.12470 (2018)
4. Blanchard, P., Guerraoui, R., Stainer, J., et al.: Machine learning with adversaries: Byzantine tolerant gradient descent. In: Advances in Neural Information Processing Systems. pp. 119–129 (2017)
5. Chen, X., Chen, T., Sun, H., Wu, Z.S., Hong, M.: Distributed training with heterogeneous data: Bridging median and mean based algorithms. arXiv preprint arXiv:1906.01736 (2019)
6. Khan, A., Sohail, A., Zahoora, U., Qureshi, A.S.: A survey of the recent architectures of deep convolutional neural networks. arXiv preprint arXiv:1901.06032 (2019)
7. Leshno, M., Lin, V.Y., Pinkus, A., Schocken, S.: Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. Neural Networks **6**(6), 861–867 (1993)
8. Lopuhaa, H.P.: On the relation between s-estimators and m-estimators of multivariate location and covariance. The Annals of Statistics **17**(4), 1662–1683 (1989)
9. McMahan, H.B., Moore, E., Ramage, D., Hampson, S., et al.: Communication-efficient learning of deep networks from decentralized data. arXiv preprint arXiv:1602.05629 (2016)
10. Obermeyer, Z., Emanuel, E.J.: Predicting the futurebig data, machine learning, and clinical medicine. The New England Journal of Medicine **375**(13), 1216 (2016)
11. Wang, X., Han, Y., Wang, C., Zhao, Q., Chen, X., Chen, M.: In-edge ai: Intelligentizing mobile edge computing, caching and communication by federated learning. IEEE Network **33**(5), 156–165 (2019)
12. Yin, D., Chen, Y., Ramchandran, K., Bartlett, P.: Byzantine-robust distributed learning: Towards optimal statistical rates. arXiv preprint arXiv:1803.01498 (2018)
13. Yu, H., Yang, S., Zhu, S.: Parallel restarted sgd with faster convergence and less communication: Demystifying why model averaging works for deep learning. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 33, pp. 5693–5700 (2019)