IPLS: A Framework for Decentralized Federated Learning

Christodoulos Pappas
University of Thessaly
chrpappas@uth.gr

Dimitris Chatzopoulos
HKUST
dcab@cse.ust.hk

Spyros Lalis
University of Thessaly
lalis@uth.gr

Manolis Vavalis
University of Thessaly
mav@uth.gr

***Abstract*—The proliferation of resourceful mobile devices that store rich, multidimensional and privacy-sensitive user data motivate federated learning, a paradigm that enables mobile devices to produce a machine-learning model without sharing their data. However, the majority of the existing federated frameworks follow a centralized approach. In this work, we introduce IPLS, a fully decentralized federated learning framework that is partially based on the interplanetary file system (IPFS). By using IPLS and connecting into the corresponding private IPFS network, any party can initiate the training process of a machine-learning model or join an ongoing training process that has been started by another party. IPLS scales with the number of participants, is robust against intermittent connectivity and dynamic participant departures/arrivals, requires minimal resources and guarantees that the accuracy of the trained model quickly converges to that of a centralized federated learning framework with a negligible accuracy drop of less than 1‰.**

***Index Terms*—machine learning, federated learning, interplanetary file system, mobile computing**

I. Introduction

Federated learning (FL) is a recently proposed machine learning (ML) paradigm that allows parties holding potentially privacy-sensitive data to collectively train a model without having to disclose their local data to a third party [1]. The most prominent example is Google Keyboard that uses metadata from users' typing to propose next words or to auto-correct typed words, while preserving user privacy [2].

Conventional FL works in a *centralised* way, whereby a single server orchestrates the training process. More specifically, the server determines the type of the model (e.g., a deep neural network) to be trained as well as the loss function and optimisation algorithm to be used (e.g., stochastic gradient descent [3]), registers agents that wish to participate in the training processes and records their contact information in order to be able to communicate with them directly, randomly samples a subset of the agents for the next training round, sends to each of these agents the most updated values of the global model parameters, and aggregates the individual agent contributions in order to update the global model parameters that will be used in the next training round.

The choice of the model, loss function and algorithm as well as the registration of the interested agents, are done as part of an initialisation process, which takes place before the actual training starts. The training process, depicted in Figure 1a, takes place in rounds, until the global parameters converge. In each round, the chosen agents receive the global parameters from the server, execute the optimisation algorithm for a predetermined period (specified in time units or number of iterations) using only their locally stored (private) data. When the period expires, each agent calculates the difference between the locally trained model and the global model that was received from the server, and reports this difference back to the server.

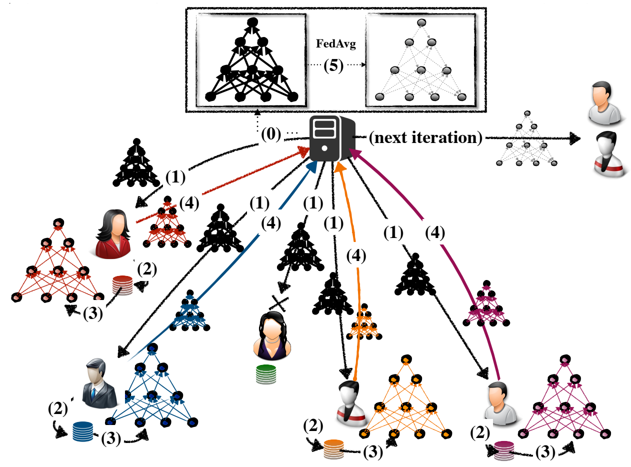
It is obvious that this approach introduces a single point of failure and any unavailability of the central server may disrupt the training process. Also, the server needs to have reliable and high-bandwidth communication links with the agents in order to support the transfer of potentially voluminous data with all of them. Additionally, a server with access to individually trained models may be able to extract privacy-sensitive information [4]–[6].

An alternative is to use a *decentralised* approach where the participating agents train the model while relying only on their own resources and without sharing all the model parameters with a single agent. In such a decentralised version of FL, illustrated in Figure 1b, the agents train a model in a peer-to-peer fashion, without the assistance of a server. Any agent can initiate the training process by specifying the model, the loss function and the training algorithm to be used. Then, interested agents may register and participate in the training process.

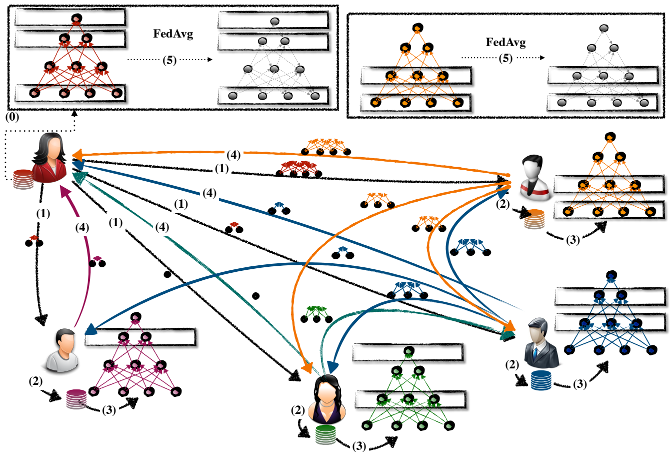
In contrast to the centralised approach, where only the server is responsible for storing, updating and broadcasting the model to the participating agents, in decentralised FL, the model is split in multiple partitions that are replicated on multiple agents. For example, a model using a neural network of 100 layers [7] can be split in 10 partitions of 10 layers each. As a result, each agent is responsible for storing a part of the model, updating the corresponding parameters and communicating them to the agents working on the other partitions.

Also note that multiple agents may be responsible for the same partition of the model, in which case, after each training round, they ideally need to synchronize in order to agree on the same values, by running a suitable aggregation protocol [8]. However, this synchronization may introduce significant delays, especially when the connections between the agents are not reliable. For this reason, in this work, we consider more relaxed, *asynchronous* aggregation protocols

ISBN 978-3-903176-39-3© 2021 IFIP



(a) Centralised FL.



(b) Decentralised FL.

Fig. 1: In centralized FL (a), each agent sends the updated model to the server, the server produces the new model, and begins a new training phase. In IPLS (b), each agent is responsible for some partitions of the model and agents interact with a limited number of other agents by exchanging partitioned gradients or model updates.

that do not require (full) synchronization after each round. We show that this does not impact significantly the accuracy of the trained model.

Contributions. Inspired by the design and functionality of the Interplanetary File System (IPFS) [9], this paper introduces a decentralized FL framework, named Interplanetary Learning System (IPLS)[1], which allows a large number of agents to collaborate in the training of a model without relying on any central entity. The main contributions are: (i) We propose a new approach towards fully decentralized and scalable FL. (ii) We completely relax the synchronization requirements between agents by the use of asynchronous stochastic gradient descent (SGD), without this having a negative effect on the accuracy of the trained model. (iii) We present a concrete implementation, in the form of a middleware atop IPFS, which can be used through a structured API by anyone who wishes to train an ML model, without having to employ a centralized service. (iv) We evaluate our implementation via a series of simulation experiments, showing that it achieves very good accuracy and convergence compared to a centralized approach.

Why IPFS. We build our implementation on top of IPFS, which allows devices of any type to exchange files without the requirement of a having a direct persistent connection between them. This way, the agents participating in an IPLS-based training process can communicate, indirectly by uploading and downloading files into IPFS. As part of our future work we intend to further improve IPLS by exploiting additional IPFS services (e.g., IPNS, CRDTs) as well as to examine other alternatives such as gossip learning protocols.

The rest of the paper is structured as follows: Section II introduces IPLS in detail. In Section III, we evaluate the performance of IPLS. Section IV compares IPLS to related work. Finally, Section V concludes the paper and points to future research directions.

[1]The current implementation of IPLS is publicly available here: https://github.com/ChristodoulosPappas/IPLS-Java-API

II. Interplanetary Learning System

The design of IPLS is based on two assumptions in order to guarantee four desirable properties.

Assumptions. We assume that any agent that participates in the training of a model using IPLS may get temporarily disconnected from its peers or unilaterally decide to terminate and depart from the ongoing training process to save resources. This may happen at any point in time. However, we assume that agents remain disconnected only for a while, unless they exhibit a permanent failure or leave the training process.

Properties. We design IPLS in such a way to guarantee the following properties:

Convergence. The global parameters of the trained model converge to a fixed set of values. Also, the accuracy of the resulting model is close to that of a model trained in a centralised fashion with the same data.

Scalability. The produced traffic increases sub-linearly to the number of participating agents. Moreover, the number of participants does not affect the total amount of data that needs to be sent/received over the network by each individual agent.

Lightweight storage requirements. Besides the local (private) data each agent owns and uses during training, IPLS itself requires relatively little extra space in order to store part of the model during the training process.

Fault-tolerance. Even if some of the agents leave unexpectedly, the training process terminates successfully without harming convergence.

A. Training a model with IPLS

Given a model M with weight parameters W, and a set of agents $\mathcal{A}$ with each agent $a_i \in \mathcal{A}$ owning a private dataset d_i,

Symbol	Meaning	Symbol	Meaning
M	model	W	model parameters
K	partitions	w_k	parameters in partition k
d_i	dataset of agent a_i	k_i	partitions stored by a_i
π	min partitions per agent	ρ	max replications per partition

TABLE I: Notation table.

we next describe how IPLS trains M in a decentralised way. The introduced notation is summarized in Table I.

Initialisation phase. Any agent can initiate the training process by determining the characteristics of M, i.e, the topology of the model (e.g., ResNet [7]), an optimisation algorithm, and a loss function that will be used to optimise the weights W of M. IPLS uses the pub/sub module [10] of IPFS to notify agents about the initialisation of a training process and invite them to express their interest.

Model partitioning and distribution. Depending on the model size, W can be split into partitions that are assigned to different agents. Each agent can be assigned multiple partitions. In addition, the same partition can be replicated on multiple agents. Formally, let W be split into K partitions, $W = \bigcup_{k=1}^{K} w_k$. Also, let k_i denote the partitions that have been assigned to agent a_i. Note that a_i and a_j can be assigned non-disjoint partitions, $k_i \cap k_j \neq \varnothing$. The distribution of the partitions is based on two tuning parameters π and ρ, for the minimum number of partitions an agent can store and the maximum number of times a partition can be replicated, respectively. In the beginning, the agent that initiates the training process stores all the partitions. When a new agent joins, she gets π partitions from the agent that is currently responsible for the most partitions. If there are several such agents, the new agent selects the π least replicated partitions. Model partitioning and assignment occur as part of the initialisation phase. Once this completes, each agent knows which agents are responsible for each partition and their addresses.

Partitioning example. Agent a_1 initiates the process with $K = 6$ partitions, let $k_1 = \{1, 2, 3, 4, 5, 6\}$, It also sets $\pi = 4$ and $\rho = 2$. When agent a_2 joins, it will store 4 partitions, let $k_2 = \{3, 4, 5, 6\}$, while a_1 remains responsible for partitions $k_1 = \{1, 2, 3, 4\}$. Next, agent a_3 joins and stores partitions $k_3 = \{1, 2, 5, 6\}$. Any other agent that wishes to participate will not be considered as it is of no use. This is because $\rho = 2$ and all partitions have already been (fully) replicated twice.

Training phase. During the training phase each agent $a_i \in \mathcal{A}$ initially contacts a sufficiently large subset of agents so as to be able to collect all global parameters. The number of the contacts depends on the number of the partitions stored locally and the rest of the partitions needed in order to have the whole model. Next, the agent uses the locally stored data, d_i, the predetermined optimisation algorithm and the loss function to update the model parameters by running the algorithm for a given number of iterations. Finally, the agent calculates the difference between the locally updated parameters and the global values received before starting the training round, and informs the contact agents responsible for each partition (not stored locally). In turn, these agents receive the individual local updates from several other agents, calculate, and send back the new corresponding global parameters of the model.

The impact of π. The value of π determines the degree of distribution for the partitions and the amount of communication that needs to take place among the agents for the partitions that are not stored locally. For example, if $\pi = 1$ and $\rho = 1$, each of the K partitions will be assigned to a different agent (assuming a sufficient number of participants), and at the end of a training round every agent needs to communicate with the rest of the $K - 1$ agents in order to inform them about its local updates and receive the corresponding global parameters from them. In the extreme case where $\pi = K$ and $\rho = 1$, all partitions will be assigned to a single agent that will become the equivalent of a centralized server.

The impact of ρ. Larger values of ρ increase robustness: whenever an agent for a given partition is not available, the other agents can get the corresponding updated global parameters from any other agent that is responsible for it. Furthermore, having more agents responsible for the same partition, distributes the load of receiving and processing corresponding local updates. For example, if ρ = 2 then each agent responsible for a given partition will receive updates from only half the agents. However, this increases the amount of communication between the agents used to replicate a given partition in order to reach a consensus about the corresponding new global parameters. Ideally, all replicator agents should synchronize with each other, after each training iteration, to produce a consistent global update for this partition before this is disseminated to the other agents. IPLS relaxes this strict synchronization requirement. More specifically, an agent waits to synchronize with its peer replicators only for a configurable amount of time. After this timeout, the agent proceeds with the dissemination of the global update and the next iteration as usual. Delayed synchronization messages that arrive out of context, are ignored.

Scalability and storage requirements. The data sent and received by each agent is constant because on each communication round it sends and receives data of at most of the size of the model. Assuming $\rho = 1$ and K partitions of equal size (i.e., $|w_1| = \ldots = |w_K| = w$), each agent a_i sends to every other agent a_j an update of size $k_j \times w$. Thus, the differences (gradients) send by agent a_i after each training round are of size $data_i^s = \sum_{j \neq i} k_j \times w = (K - k_i) \times w < W$. The same holds for the received updates $data_i^r$. Thus, the total amount of data each agent a_i exchanges with other agents in each round is $data_i^s + data_i^r \leq 2 \times W$. It follows that the total amount of data communicated in each round is $\sum_i = data_i^s + data_i^r \leq 2 \times |\mathcal{A}| \times W$, which is the same volume as in conventional centralized FL. As the value of ρ increases and the same partition is replicated on a larger number of agents, each agent communicates with fewer agents in order to receive gradients and send back the respective updates. However, the agents that store the same partition

Algorithm 1: Operation of agent $a_i \in \mathcal{A}$ using the IPLS primitives.

```
$IPLS.Init(filepath, bootstrapper)$
while $accuracy < Threshold$ do
    $M \leftarrow IPLS.LoadModel()$
    $\Delta W \leftarrow M.fit(d_i, SGD)$
    $IPLS.UpdateModel(\Delta W)$
end
```

need to exchange additional synchronization messages for the calculation of the global partition parameters. In terms of storage, IPLS has lightweight requirements as agents only need to store the models in which they participate in their training.

B. IPLS API

IPLS is build atop IPFS [9], a fully decentralized peer-to-peer file system with a pub/sub functionality that assists agents on communicating with each other. IPLS offers an API of four methods: $Init$, $UpdateModel$, $LoadModel$ and $Terminate$ Algorithm 1 shows how the first three methods are used during the training of the model. The fourth method is used only when the agent wishes to quit the training process prematurely.

First, the agent invokes the IPLS $Init$ method. This initializes the IPFS daemon, retrieves the model M, the loss function, π, ρ and the optimisation/training algorithm to be used, and broadcasts the local communication addresses via the pub/sub service of IPFS. It then waits for responses from already participating agents, containing their communication addresses and the partitions for which they are responsible. Once responses have been collected from a sufficient number of agents, the method selects the partitions for which the local agent will be responsible for, and communicates this information to all other participants. Finally, a daemon is started that take care of all the interactions that occur in the background while the agent is busy training the model.

Before starting the next training iteration, the agent needs to call the $LoadModel$ method in order to retrieve the current model. In the very first invocation, this call returns the model that is retrieved during initialiation. In all subsequent invocations, it returns the updated local model, which is combined with the global updates that were received from agents responsible for the corresponding partitions at the end of the previous training round.

At the end of each training iteration, the agent calls the $UpdateModel$ method in order to update the local model. For each partition, a lookup is performed to find agents that are responsible for it. There can be many criteria for choosing the suitable agent, such as locality, connectivity, trust, load and energy. The current implementation does not employ any such optimization criterion, but can be extended to do so. After selecting the appropriate agents for each partition, a request is sent to each one of them, containing the partition identifier and the gradients sub-vector that resulted from local training, and a reply is received with the updated global sub-vector that is used to update the local sub-vector of the model. Each agent also performs the necessary actions for the partitions which is responsible. More specifically, upon receiving a gradient update δ_k for partition k, the corresponding model parameter w_k is updated by subtracting the gradient with a weight factor ϵ and send back to the agent, to $w_k \leftarrow w_k - \epsilon \times \delta_k$. Also, the weight factor itself is updated according to the number r of local gradient updates received, to $\epsilon \leftarrow \alpha \times \epsilon + (1 - \alpha) \times \frac{1}{r}$, where $\alpha \in (0, 1)$ set during the initialization phase.

Finally, when an agent wishes to stop participating in the model training process, it calls the $Terminate$ method. As a result, IPLS uploads to IPFS a file containing the model partitions for which the leaving agent was responsible along with the current global parameters. It then looks up for other agents that could assume these responsibilities, and broadcasts a message with the corresponding partition assignment. Upon receiving such message, these agents update the partitions for which they are responsible, download the corresponding partition parameters and aggregate them with their own local weights in order to form a new global sub-vector. Note that this is done in the background, and any changes in the partitioning are communicated to the application layer of the agent the next time it invokes the $LoadModel$ method.

III. Performance Evaluation

We have conducted various experiments to examine the performance of our implementation. In this Section, we present the experimental setup and discuss the results of indicative experiments in terms of model training convergence, fault-tolerance and scalability.

A. Setup

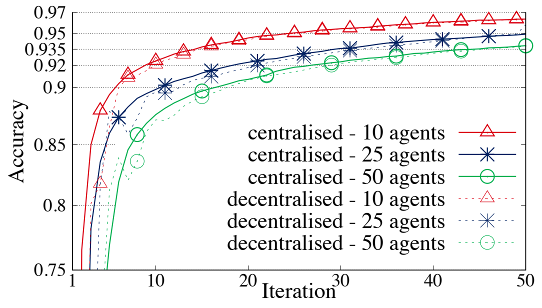
For the simulation of the connectivity between the agents we use *mininet*[2]. Each mininet node is an agent that uses IPLS in order to participate in the training of a model. Additionally, we set up a private IPFS network where every node runs as part of an IPLS agent.

We use the MNIST dataset [11] that contains 60000 images of digits that are categorised in 10 classes (i.e., $0-9$). We use MNIST to train a four-layer ($785 \times 500 \times 100 \times 10$) neural network that can classify an image that contains a digit. We split MNIST into $|\mathcal{A}|$ parts, with uniformly distributed labels and assign to each agent a_i a dataset of $d_i = 60000/|\mathcal{A}|$ samples. For instance, when using 10 agents, each agent stores 6000 samples and the probability of each such sample to belong to a particular class is the same for every agent.

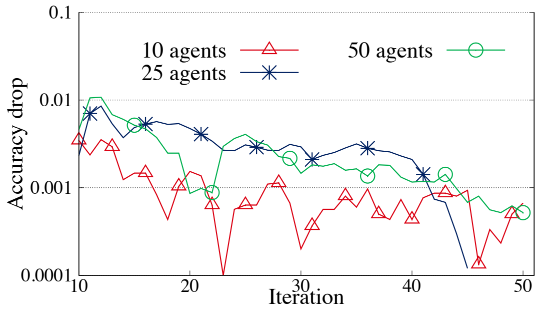
B. Experiments

Model training convergence. First, we examine the convergence of the model training process when using IPLS (decentralized) vs conventional (centralized) FL, for three scenarios with 10, 25 and 50 agents. In these experiments, we set $\pi = 1$ and $\rho = 1$. Figure 2a depicts the accuracy increase as a function of the number of iterations performed. It can be

[2]http://mininet.org/



(a) Accuracy of decentralized vs centralised FL.



(b) Accuracy loss due to decentralisation.

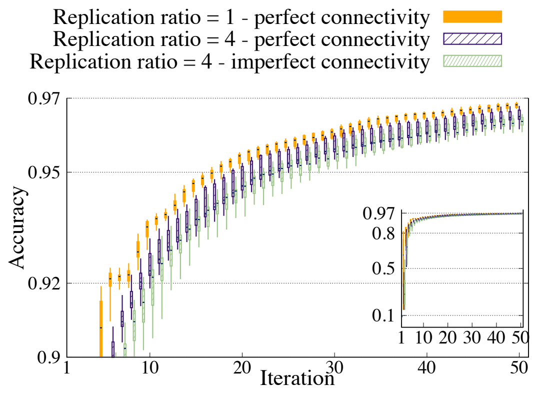
Fig. 2: Convergence of IPLS (decentralized) vs conventional (centralized) FL when using a different number of agents.

seen that IPLS has similar convergence to conventional FL. Additionally, we confirm that if a fixed dataset is partitioned in fewer parts and given to less agents, the accuracy of the model is higher. This is explained by the fact that each agent has more data when updating her local model. Figure 2b shows the loss of accuracy due to decentralisation. After 40 iterations, this becomes less than 1‰, which is practically negligible for most applications.

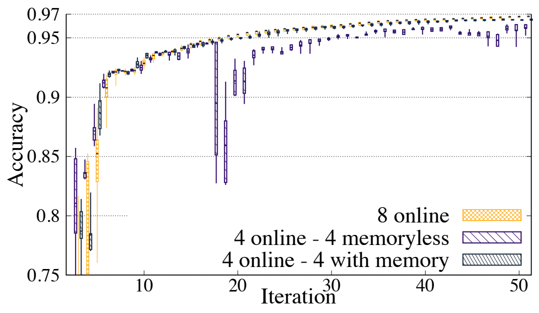
Fault-tolerance. Next, we examine how the partition replication ratio impacts accuracy. We run experiments with 8 agents and $\rho = 1, 4$ for perfect and imperfect connectivity. The latter is simulated by de-activating and then re-activating the agent's network access. Figure 3a depicts the results. It can be seen that increasing the replication ratio decreases the accuracy achieved. This is due to the fact that the agents who are responsible for the same partition do not synchronise in a strict manner to produce the correct global parameters after each iteration. Interestingly, imperfect connectivity does not have a grave impact on accuracy or convergence, which demonstrates the ability of IPLS to handle such scenarios in a graceful way. We note that the behavior of the no-replication configuration ($\rho = 1$) is the same way as in the perfect connectivity scenario and is not shown separately in the figure. The reason is that each agent has to receive the global update for each partition that is not stored locally from the single responsible agent, waiting for it to re-connect (if needed). This introduces an extra delay, but it does not affect convergence.

Further, we examine the robustness of IPLS for dynamic agent departures and participation. More specifically, we investigate a scenario with 8 agents, half of which leave and then re-join the training process. We consider the case where a joining agent starts completely memoryless without retrieving the current model or can continue with the model it had locally available when it left (with memory). In these experiments, we set $\pi = 1$ and $\rho = 1$. As shown in Figure 3b, the accuracy of the trained model does not deteriorate significantly vs the case where all agents participate throughout the entire training process. In fact, if the joining agents have memory, convergence is practically identical to the case where there are no departures at all. Joining without memory introduces some inaccuracy, but this is repaired to a large degree, given a sufficient number of iterations.

Communication Complexity. Table II shows the data traffic of IPLS for different configurations. This is reported at the layer of IPLS as well as at the network/transport layer (measured using wireshark). As discussed in Section II, for $\rho = 1$ each agent has to send/receive approximately two



(a) Different replication and connectivity scenarios.



(b) Different agent departure/join scenarios.

Fig. 3: Robustness of IPLS in the presence of intermittent connectivity and dynamic agent departures/arrivals.

Number of agents	4	5	6	7	8
$\rho = 1$ in network	7.14	7.6	7.9	8.16	8.26
$\rho = 1$ in middleware	6.4	6.67	6.9	7.1	7.2
$\rho = 2$ in network	7.9	8.56	8.72	8.83	8.86
$\rho = 2$ in middleware	6.91	7.096	7.2	7.4	7.42

TABLE II: Data (MB) sent by each agent per iteration.

times the size of the model. For our model of 3.55 MB, this amounts to 7.1 MB. We can see that this assumption is valid, despite the extra communication overhead of the IPFS daemon. Note that for $\rho = 2$ the amount of data exchanged over the network is larger than for $\rho = 1$. This is because of the extra data exchange that takes place to synchronize the agents responsible for the same (replicated) partitions. However, the overall impact is relatively limited due to the small replication ratio.

IV. Related Work

Existing decentralized FL systems are mostly based on gossiping schemes. For example, the authors of [12] and [13] implement the classic decentralized ML algorithm on which agents download the model from multiple neighbouring agents. An alternative approach is proposed by Ramanan *et al.* [14] who use a blockchain to aggregate agents' updates. However, their approach has several limitations related to the gas costs and the data size of blockchain-based transactions.

Although the work of Hu *et al* [15] is close to IPLS, since it also partitions the model into non overlapping segments, it differs heavily from IPLS because it is based on gossiping, and not on a distributed memory abstraction. Moreover, IPLS differs from [12]–[14] because it does not download the entire model from selected peers but only partitions of that model. The disadvantage of [12]–[14] compared to IPLS, is that in order to gain better accuracy agents have to download the same partition from different agents. Compared to the aforementioned works, IPLS not only transmits significantly less data over the internet, but also reaches approximately the same convergence rate and accuracy as our centralized rival. Moreover given that IPLS is based on distributed shared memory, gives the API users more freedom to apply classic parallel optimization algorithms such as [16] which can heavily reduce the communication complexity.

V. Conclusion and Future work

The unavailability of a decentralized federated learning framework that can be used directly in mobile devices and especially smartphones motivated the development of IPLS. Although in an early stage, IPLS can be used to train models with the same convergence rate and the same traffic, as traditional FL frameworks.

There are multiple directions towards which IPLS can be further developed. First of all, it needs to be installed in heterogeneous devices in order to analyse its needs and performance. Similarly, we plan to examine its feasibility in training as many as possible state-of-the-art models. Furthermore, a more sophisticated algorithm that allows agents to change the partitions for which they are responsible based on their bandwidth and their available resources can increase significantly the performance of IPLS because more updates will be delivered on time. Additionally, split learning techniques [17], [18] can be integrated to IPLS in order to enable devices with limited computational capabilities to participate. Last but not least, IPLS should incorporate an incentive mechanism, similar to Filecoin [19] and Flopcoin [20], to motivate mobile users to share their resources.

Acknowledgement

This research has been co-financed by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH-CREATE-INNOVATE (project code: T1EDK-02161).

References

[1] Q. Yang, Y. Liu, T. Chen, and Y. Tong, "Federated machine learning: Concept and applications," *ACM Trans. Intell. Syst. Technol.*, vol. 10, no. 2, Jan. 2019.

[2] T. Yang, G. Andrew, H. Eichner, H. Sun, W. Li, N. Kong, D. Ramage, and F. Beaufays, "Applied federated learning: Improving google keyboard query suggestions," *CoRR*, vol. abs/1812.02903, 2018.

[3] H. Robbins and S. Monro, "A stochastic approximation method," *The annals of mathematical statistics*, pp. 400–407, 1951.

[4] M. Fredrikson, S. Jha, and T. Ristenpart, "Model inversion attacks that exploit confidence information and basic countermeasures," in *Proc. of CCS*, 2015, p. 1322–1333.

[5] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 3–18.

[6] X. Yuan, P. He, Q. Zhu, and X. Li, "Adversarial examples: Attacks and defenses for deep learning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 9, pp. 2805–2824, 2019.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[8] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical secure aggregation for privacy-preserving machine learning," in *Proc. of ACM CCS*, 2017, p. 1175–1191.

[9] J. Benet, "Ipfs-content addressed, versioned, p2p file system," *arXiv preprint arXiv:1407.3561*, 2014.

[10] K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems," in *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, ser. SOSP '87, 1987, p. 123–138.

[11] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[12] A. Koloskova, S. U. Stich, and M. Jaggi, "Decentralized stochastic optimization and gossip algorithms with compressed communication," *arXiv preprint arXiv:1902.00340*, 2019.

[13] A. G. Roy, S. Siddiqui, S. Pölsterl, N. Navab, and C. Wachinger, "Braintorrent: A peer-to-peer environment for decentralized federated learning," *CoRR*, vol. abs/1905.06731, 2019.

[14] P. Ramanan, K. Nakayama, and R. Sharma, "BAFFLE : Blockchain based aggregator free federated learning," *CoRR*, vol. abs/1909.07452, 2019.

[15] C. Hu, J. Jiang, and Z. Wang, "Decentralized federated learning: A segmented gossip approach," *CoRR*, vol. abs/1908.07782, 2019.

[16] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in neural information processing systems*, 2011, pp. 693–701.

[17] K. Palanisamy, V. Khimani, M. H. Moti, and D. Chatzopoulos, "Spliteasy: A practical approach for training ml models on mobile devices," in *Proc. of ACM HotMobile*, 2021, p. 37–43.

[18] P. Vepakomma, O. Gupta, T. Swedish, and R. Raskar, "Split learning for health: Distributed deep learning without sharing raw patient data," *CoRR*, vol. abs/1812.00564, 2018.

[19] J. Benet, "Filecoin research roadmap for 2017," 2017.

[20] D. Chatzopoulos, M. Ahmadi, S. Kosta, and P. Hui, "Flopcoin: A cryptocurrency for computation offloading," *IEEE Transactions on Mobile Computing*, vol. 17, no. 5, pp. 1062–1075, 2017.