用Blockchain实现了VFL，和其他研究激励的不同，这篇文章是借助blockchain的透明性来实现开放FL平台，和我的综述最接近！作者将FL训练抽象为三个角色，用IPFS存储模型，用一个scorer来控制模型整合，任务发布者owner存储label-id。作者还提供了代码和实现细节。美中不足是只实现了图片VFL场景并且仅4 clients。

BlockLearning: A Modular Framework for Blockchain-Based Vertical Federated Learning

Henrique Dias(✉) and Nirvana Meratnia(✉)

Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven 5600MB, The Netherlands
mail@hacdias.com, n.meratnia@tue.nl

Abstract. Federated Learning allows multiple distributed clients to collaborate on training the same Machine Learning model. Blockchain-based Federated Learning has emerged in recent years to improve its transparency, traceability, auditability, authentication, persistency, and information safety. Various Blockchain-based Horizontal Federated Learning models are to be found in the literature. However, to the best of our knowledge, no solution for Blockchain-based Vertical Federated Learning exists. In this paper, we introduce BlockLearning, an extensible and modular framework that supports Vertical Federated Learning and different types of blockchain related algorithms. We also present performance evaluation results in terms of execution time, transaction cost, transaction latency, model accuracy and convergence, as well as communication and computation costs when BlockLearning is applied to vertically partitioned data.

Keywords: Blockchain · Blockchain-based federated learning · Horizontal federated learning · Vertical federated learning

1 Introduction

Federated Learning (FL), introduced by Google researchers in 2016 [19], allows multiple clients, in different locations, to collaborate on training a global Machine Learning (ML) model without sharing their own data with each other. Instead of sharing the raw data, clients only share their model parameters, such as weights. The first benefit of FL is that, by not sharing raw data, models can preserve clients' data privacy. In addition, since model parameters are usually much smaller than the raw data, this leads to less data being transported over the networks. Finally, since the data is distributed among different clients, a single powerful server is not required to train the model, as usually training models with smaller amounts of data is computationally less expensive.

According to [26,33], FL techniques can be broadly divided into three categories: horizontal, vertical, and transfer federated learning. In Horizontal Federated Learning (HFL), the different data sets in the different clients share the same feature space, but not the sample space. In Vertical Federated Learning (VFL), clients share an intersecting sample space, but different feature spaces.

Most FL solutions include a central server that coordinates the federated training process and aggregates the model weights from each of the clients into a single model. This central coordinator is a single point of failure, since it is required to always be online and behave correctly [16,30]. To address this, Blockchain-based Federated Learning (BFL) techniques have been proposed.

By combining Blockchain with the Federated Learning, not only can the central orchestrator be eliminated, but also the federated training process can be made more transparent. In the blockchain, each transaction is recorded in the distributed ledger. These transactions record information such as local updates, scores, aggregations, among others. Having this information in a public ledger allows for a transparent training process and reward distribution [16], as well as traceability, auditability, persistency, and authentication.

While the combination of blockchain and Horizontal Federated Learning has received enough attention from the research community, to the best of our knowledge, there is no work giving a practical solution on how to implement Vertical Federated Learning in the context of Blockchain-based Federated Learning. To fill this gap, we propose a modular and extensible framework for Blockchain-based Federated Learning, called BlockLearning, that supports Vertical Federated Learning. In addition to presenting the design and implementation aspects of BlockLearning, we also present its performance evaluation results in terms of execution time, transaction cost, transaction latency, model accuracy and convergence, as well as communication and computation costs when applied to vertically partitioned data.

The remainder of this paper is structured as follows. Section 2 presents a short overview of the existing work regarding BFL frameworks. Sections 3 and 4 describe the design and implementation of our framework BlockLearning, respectively, focusing on Vertical Federated Learning. Section 5 presents our performance evaluation and experiments. Section 6 presents discussion of the results. Finally, Sect. 7 gives a conclusion and future directions.

2 Related Work

As far as use of consensus algorithms in BFL is concerned, most authors have used an already existing consensus algorithm, such as Proof of Work [13,21,35], Proof of Stake [6,9,17] and Proof of Authority [14,27,34]. In addition, in the majority of existing works that used an already existing consensus algorithm, the BFL system is built on top of an already existing blockchain platform.

According to the literature, the FL model parameters may either be stored on-chain, i.e., in the blockchain itself [14,27,34], or off-chain, i.e., in a separate storage provider [2,4,18]. Even though most implementations prefer an on-chain storage, they also use custom blockchain implementations [5,13,31], which means that they can implement a platform that has different restrictions on how much data a smart contract can handle. When it comes to using already existing blockchain platforms such as Ethereum, most implementations prefer off-chain storage using a system such as the InterPlanetary File System [18,21,25].

Several works address Horizontal BFL frameworks [10,18,23,30]. However, only [22] discusses the possibility of implementing a Vertical BFS system, but provides no practical solution. In addition, very few of these frameworks provide the source code or build an extensible framework.

3 BlockLearning Framework's Design

Our modular framework, called BlockLearning, is designed in such a way that modules can be easily added, removed or changed, to accommodate both vertical and horizontal partitioned data and different blockchain-related algorithms. In this paper, we focus on its support for Vertical Federated Learning. To see how it supports Horizontal Federated Learning, one is referred to [11].

In BlockLearning, devices, identified by the address of their account in the blockchain, can be classified into three categories: *trainers*, *aggregators* and *scorers*. Additionally, the entity that deploys the contract and is responsible for starting and terminating the rounds is called *model owner*. A device, i.e., a client or a server, can be categorized as one or more categories. By allowing each device to play more than one role, the framework provides flexibility to support different architectures and algorithms.

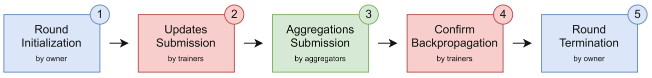


Fig. 1. BlockLearning's execution Flow

The framework supports a modular sequential flow presented in Fig. 1 and has the following phases:

1. The model owner initializes the round, during which, depending on the participant selection algorithm, the trainers that will participate may have been selected already, or not.
2. The trainers retrieve the information such as the global weights from the last round and train the model using their local data. Then, the trainers submit their model updates.
3. The aggregators retrieve the model updates and execute the aggregation algorithm and submit the aggregation results and the backpropagation gradients to the blockchain.
4. The trainers receive the gradients and backpropagate to their local model. Then, the trainers send a message to the blockchain to confirm the backpropagation.

5. Finally, the model owner sends a transaction to the blockchain in order to terminate the round. At this point, the smart contract checks if the majority of the aggregators agreed on the aggregation. If so, the round is marked as terminated. Otherwise, the round fails, indicating that the aggregators did not reach an agreement, which may indicate that some of the aggregators are compromised.

In the last phase of the execution flow, the smart contract checks if the majority of the aggregators agree on the aggregation. The majority is defined by *at least 50%*. Therefore, the framework offers a 50% threat model. However, the threat threshold can be changed, changing the threat model.

3.1 Structure and Modules

The framework is divided into three main software components: the smart contracts, the library, and the testbed. The structure of the framework, as well as its components and their corresponding modules, are depicted in Fig. 2. Each of the components plays a different role in the overall system in order to support the logical flow shown in Fig. 1. In what follows, we explain each of the components in more detail. One should note the unexplained components are not relevant for Vertical Federated Learning and are extensions to support the Horizontal Federated Learning.

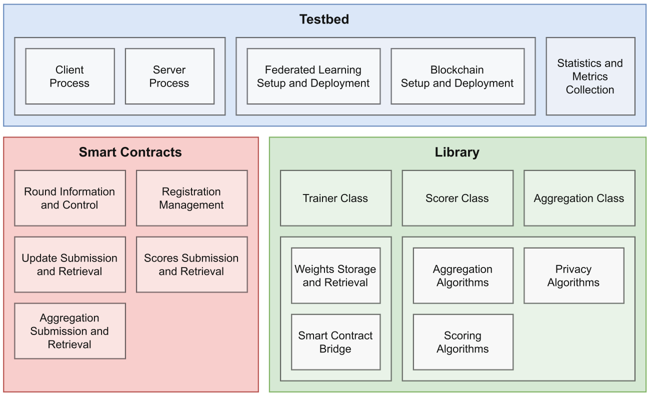

Testbed
Client Process
Server Process
Federated Learning Setup and Deployment
Blockchain Setup and Deployment
Statistics and Metrics Collection
Smart Contracts
Round Information and Control
Registration Management
Update Submission and Retrieval
Scores Submission and Retrieval
Aggregation Submission and Retrieval
Library
Trainer Class
Scorer Class
Aggregation Class
Weights Storage and Retrieval
Smart Contract Bridge
Aggregation Algorithms
Privacy Algorithms
Scoring Algorithms

Fig. 2. BlockLearning's structure and modules

Smart Contracts live on the blockchain and are the main means of communication between FL clients and servers. In addition, they hold information regarding the current status of the round, as well as the updates, scores, aggregations, among others. The smart contracts provide the following functionality:

- *Round Information and Control*: the smart contract provides information about whether the round is ongoing and which phase, i.e., scoring, aggregation, or termination phase, it is in. It allows for flexibility such that new phases can be added in the future, such as the backpropagation confirmation phase we need for our vertical model. In addition, it allows for rounds to be started and marked as terminated. Round phase advancements are defined through pre-defined conditions that, once met, automatically move the round to the next phase.
- *Registration Management*: the smart contract allows devices to register themselves as trainers, aggregators, or scorers in the system. Finally, the smart contract provides information about which devices participate in each round.
- *Update Submission and Retrieval*: the smart contract allows trainers to submit their updates, which must include a pointer to the model weights and the amount of data points that were used to train the model. In addition, it includes the training accuracy and testing accuracy for each individual trainer. The submissions must be accessible.

返回模型参数的指针？

- *Aggregation Submission and Retrieval*: the smart contract allows aggregators to submit the aggregations, which contain a pointer to the weights. The aggregations must be accessible.

Library encodes the algorithms, utilities, and building blocks necessary to implement the scripts that run on the clients and the servers. It includes:

- *Aggregation Algorithms*: implementation of the different aggregation algorithms with a common interface, such that adding new algorithms is easy and simple and they are interchangeable.
- *Weights Storage and Retrieval*: utilities to load and store weights on the decentralized storage provider. These provide an interface in order to make it easy to change the storage provider by providing a different implementation.

IPFS

- *Smart Contract Bridge*: a contract class that provides an interface to the smart contract that lives on the blockchain. With this class, it should be possible to call the smart contract functions as if they were local functions.
- *Trainer* and *Aggregator Classes*: a class per each device category. This class must register the devices as their category upon initialization. It must also provide methods to execute the training, scoring and aggregation tasks, respectively.

Testbed provides the platform to conduct the experiments in a reproducible way, for instance by setting static seeds for randomness. The testbed includes:

- *Client, Server* and *Owner Scripts*: scripts that will be run at the clients, the servers, and at the model owner, respectively. These scripts will use the library in order to perform the right tasks according to which algorithm is being used.
- *Federated Learning Setup and Deployment*: scripts and tools to easily deploy the client and server machines in a test environment, such as containers.

– *Blockchain Setup and Deployment*: scripts and tools to easily deploy the blockchain network in a test environment using the different consensus algorithms, and to deploy the contract to such network.

In addition, the testbed includes tools to collect the required statistics and logs to retrieve the metrics necessary for the impact analysis.

4 BlockLearning Framework's Implementation

In this section, we go over the implementation details of the BlockLearning framework. The complete implementation is publicly available on GitHub[1].

Smart Contracts: We use the Ethereum [32] blockchain platform as it is the most popular and compatible with all techniques we use for our experiments and comparison with the related work. Therefore, the smart contracts must be implemented in a programming language that supports Ethereum. For this, we chose Solidity [8] as it is the most well-known with the widest support.

Our framework provides a smart contract, named `Base`, that provides the common data structures and functionality. Then, other smart contracts can derive from `Base` and provide additional functionality. In the case for Vertical Federated Learning, we have a `VerticalSplitCNN` smart contract which provides the additional functionality required to support Vertical Federated Learning through the blockchain with the Split-CNN model, which is introduced in the following section.

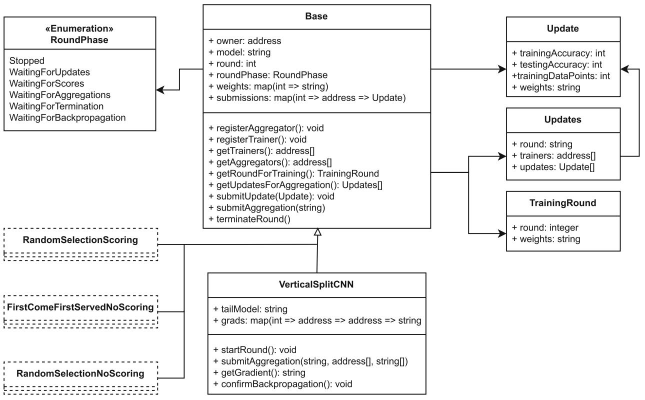


Fig. 3. Smart Contracts Class Diagram

A class diagram with the public interfaces of the contracts, as well as the data types, is depicted in Fig. 3 (adapted from [11]). It can clearly be seen that

[1] https://github.com/hacdias/blocklearning.

the smart contract provides round information and control, registration management, updates submission and retrieval, as well as aggregation submission and retrieval.

An interesting implementation detail to note is that score and accuracy values are stored as integers. Currently, Solidity does not support floating point numbers. To preserve fidelity, the original values are multiplied by a large integer, 10^{18}. Then, when the values are retrieved from the smart contract, they are divided by the same value in order to get the original value.

Library: is implemented in the Python [29] programming language. The main motivation for using Python is that many well-known Machine Learning libraries, such as TensorFlow [1] and PyTorch [24] are implemented in Python, as well as many data processing tools.

Aggregation Algorithms: are the first component of the library and they provide a common interface to which each algorithm must conform to. By having a common interface, we can easily implement new algorithms, or change existing ones. In the case of the aggregation algorithms, we require the following interface:

$$\texttt{aggregate(trainers, updates, scorers, scores)} \rightarrow \texttt{weights}$$

The aggregators provides a function `aggregate` that receives an array with the trainer addresses, an array with the updates sorted by the same order as the trainers, an array with the scorers and an array with the scores sorted by the same order as the scorers. It is important to note that the scorers and the scores are optional arguments and related to Horizontal Federated Learning. The function returns an array with the aggregated weights.

Weights Storage and Retrieval: is the second component of the library, consisting on the utilities to store and retrieve the weights. The weights storage class also provides a common interface such that change of storage providers is possible. For our implementation, we use the InterPlanetary File System (IPFS) [3] as our decentralized storage provider since it is widely used by the community.

Smart Contract Bridge: is the third component, consisting on the smart contract bridge class. The smart contract bridge is implemented using the `Web3.py`[2] library, which provides utilities to call the functions of the smart contracts. The contract bridge class provides 1:1 functions for each functions of the smart contract.

Trainer and Aggregator Classes: implement the main flow of each of these procedures using the modules aforementioned described. For example, the trainer class is initialized with the contract bridge, the weights storage, the model, the data and an optional privacy mechanism. Then, it provides a method `train()` that executes the training procedure, and a method `backward()` that executes the backward propagation. Similarly, the aggregator class provides `aggregate()`.

[2] https://github.com/ethereum/web3.py.

Testbed: is the platform to conduct the experiments. It is mostly implemented using the aforementioned library and Docker [20]. Docker is a platform that allows to easily deploy applications in an isolated setting through what is called a container, allowing us to simulate multiple devices in the same network. Each container runs a piece of software called an image. In the testbed, we have the following major components:

Client, Server and Owner Scripts: are the processes that will run at the client, server and model owner, respectively. These are implemented using the BlockLearning library. In each of these scripts, we first load the required data, such as the data set in the clients, and initialize the required algorithms. Algorithm 1 presents the main loop of the client script, for Vertical Federated Learning with the Split-CNN model.

将两个vertical client的Hidden拼接后输入到server的后端layer中决策

Algorithm 1. Client Script Main Loop for Split-CNN

```
T ← Initialize Split-CNN Trainer
while True do
    P ← Get Phase From Smart Contract
    if P is Waiting For Updates then
        Execute Training Procedure T.train()
    else if P is Waiting For Backpropagation then
        Execute Backpropagation Procedure T.backward()
    end if
end while
```

Blockchain Setup and Deployment: is implemented using already existing tools and our library. As previously mentioned, we use Docker containers in order to run the experiments. Moreover, we use Docker Compose in order to deploy multiple containers at once and orchestrate the deployment process.

We use different Ethereum implementations, depending on the consensus algorithm since they are not all available within the sample implementation. Ethereum's main implementation, `go-ethereum`[3], provides PoA and PoW. For QBFT, we use a fork called `quorum` [7], which is mostly identical to `go-ethereum` but supports QBFT. Moreover, the Blockchain setup and deployment follows the following steps:

1. *Generate Accounts.* In first place, the Ethereum accounts for the clients and servers are generated using the provided `go-ethereum` toolkit. Each account is pre-loaded with 100 ETH, the Ethereum currency, so that clients or servers will not run out of currency to submit their transactions.
2. *Build Images.* In second place, we build the Docker images that will be used to deploy the Blockchain network. This images are based on the images provided by each of the Ethereum's implementations that we use. In addition, they

[3] https://github.com/ethereum/go-ethereum.

pre-load the account information, as well as some additional configuration to ensure that all nodes are connected when the network is bootstrapped.
3. *Deploy Network.* In third place, the network is deployed using Docker Compose and the configured amount of nodes.
4. *Deploy Contract.* Finally, the contract is deployed to the network using Truffle, which is a tool designed to help developers developing and deploying smart contracts.

Federated Learning Setup and Deployment: similarly to the Blockchain setup and deployment, we also use Docker Compose for the Federated Learning system. The process is identical as in the previous section, except that we only build the images and deploy the Federated Learning network.

Statistics and Metrics Collection: the different components of the library, such as the `Trainer` and `Aggregator` classes, produce logs. These logs contain information related to timestamps and round number, and events that happen at certain points of the execution, such as: *aggregation started*, *aggregation ended*, among others. These logs are retrieved from the containers using command-line tools implemented into a script called `toolkit.py`. In addition, resource-related statistics, such as RAM usage, CPU usage, and network traffic, are collected directly from the Docker, through the `docker stats` command.

5 Experimental Setup and Evaluation

In this section, we provide information regarding the experimental setup and performance evaluation.

Dataset, Client Sampling, and Machine Learning Model: for our experiments, we used the MNIST [15] dataset, which includes 70,000 images of handwritten digits from 0 to 9, where each image is 28 by 28 pixels. The MINST data set is not only a well-known dataset but also widely used by the majority of the reviewed works. For the vertical data partition, we used the method of [28]. Firstly, we chose number of samples to be assigned to each client. We chose 20,000 samples in order to match the original work [28] we will be comparing with. The samples were randomly chosen from the original data set. Subsequently, each sample was assigned a unique identifier (ID) that is used as label when giving the data samples to each client. Only the servers have access to the ground truth labels. After assigning the IDs, the feature space F was divided into C parts F_c, where C is the number of clients. Finally, the features F_c, with $c \in C$ was assigned to each of the clients.

For the Vertical FL, we used a dual-headed, or four-headed, Split-CNN [12, 28], depending on whether we have two or four clients. To use this model, the model owner is expected to have the labels, while the clients are expected to have some features of each sample. For the MNIST data set, the features are vertical fragments of the image. To divide a 28 by 28 image sample between 2 clients, for example, we split the image into two 14 by 28 segments, as depicted

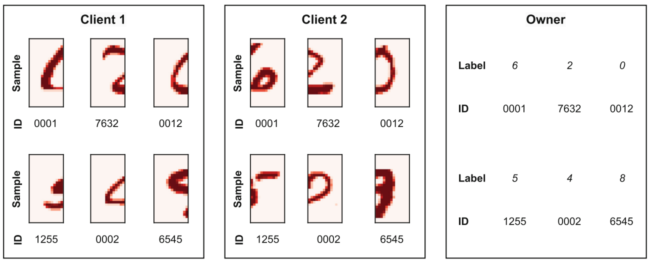


Fig. 4. Vertical Data Distribution for 2 Clients

in Fig. 4. The model at the clients is the head model, while the model at the servers is the tail model. To train this model, each client gives its input data to the models and collects the output of the last layer. Then, this intermediate output is sent to the servers, which are then given to the tail model. The servers calculate the gradients, which are then backpropagated to the clients. For more details, please consult the original works where the workings of this model are given in more detail. The architecture is depicted in Fig. 5 (adapted from [12]).

拼接后传给后面的FC层

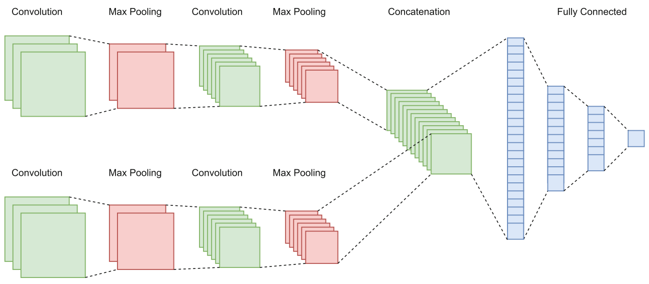


Fig. 5. Split-CNN Model Architecture

Hardware and Software Specifications: the experiments were executed on a remote machine, with a AMD Ryzen 5 3600 6-Core 4.2 GHz CPU, 64 GB of RAM, and a 500 GB NVMe disk. Use of GPUs was not needed, since if we consider that FL systems are being executed in IoT clients, it is unlikely that such resource-constrained devices would have a GPU available. In addition, the MNIST data set and the models we used are relatively simple, which means that they can be easily trained using CPUs. Nonetheless, it is worth mentioning that the training process would likely be faster on machines with GPUs.

6 Results and Discussion

We performed experiments to validate whether our implementation was successful and whether the Vertical BFL can be supported. We ran two experiments with two different number of clients: 2 and 4. The decision to use 2 clients was motivated by [12], where the Split-CNN was introduced for Vertical FL without blockchain for the first time. In addition, we also performed experiments with 4 clients.

Execution Time, Transaction Cost, and Transaction Latency: as it can be seen from Table 1, the experiments take longer with 4 clients than with 2. Regarding the transaction latency, it can be seen that it does not vary considerably with the number of clients. Similarly, the transaction costs do not present significant changes as the number of clients is relatively low. However, as expected, the transaction costs are slightly higher with 4 clients.

Table 1. Execution time, transaction cost, and transaction latency per number of clients

	2	4
E2E time (m)	18.08	24.30
Mean round time (s)	21.68	29.15
Mean transaction latency (s)	1.482	1.418
Mean transaction cost (Gas)	138659	141013

Model Accuracy and Convergence: Figure 6 illustrates the model accuracy of our experiments as well as those of Romanini et al. [28], where a Split-CNN model without blockchain was used with the MNIST data set. It can be seen that model accuracy of [28] is higher, which may be related to implementation differences like the Machine Learning library used, which is not known.

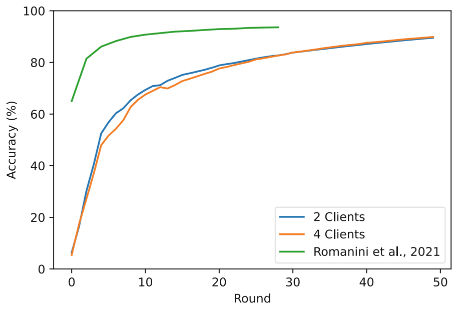


Fig. 6. Model Accuracy Per Number of Clients

Communication Costs: we can observe from Fig. 7 that at the clients there is no major difference of traffic when the number of clients increases. This can be explained by the fact that, by using a Split-CNN, each client is only required to upload its own intermediate results and downloads the gradient updates, which are similar in size. At the servers, the costs are higher as the number of clients increases. Since the higher number of clients leads to higher number of heads in the Split-CNN model, the servers are required to download more intermediate results and to upload more gradient updates. Therefore, the network traffic at the servers increases with the number of clients.

On the blockchain, the difference of number of clients is not significant to make a significant difference on traffic, since these experiments were executed with a very low number of clients.

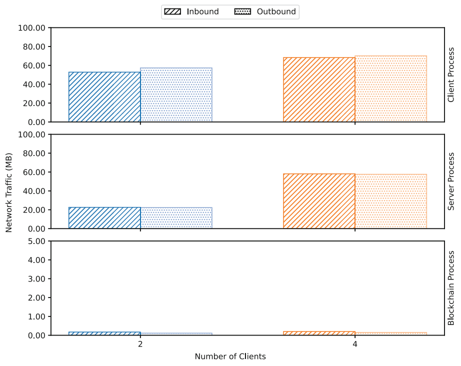
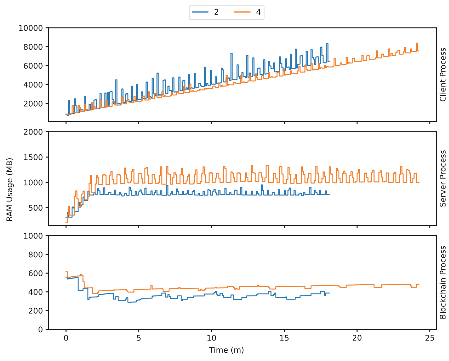


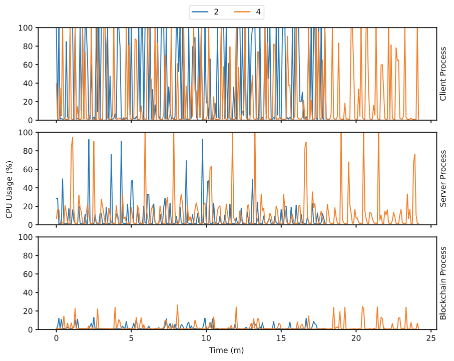
Fig. 7. Network traffic per round per number of clients

Computation Costs: Computation costs, namely RAM usage and CPU usage, are depicted in Figs. 8a and 8b, respectively. Regarding the RAM usage, we observe that with a higher number of clients, there is a higher RAM usage on the serves and the blockchain processes. This is caused by the fact that more data is being stored in-memory due to the higher amount of intermediate results that the servers store in-memory, as well as the number of blockchain transactions in the blockchain. At the clients, however, the opposite happens. This can be explained by the fact that when there are more clients, each client has less features as per the data partitioning, as explained before.

Regarding the CPU usage, we see similar results as to the RAM usage, which are explained by the same reasons.



(a) RAM Usage



(b) CPU Usage

Fig. 8. Computation costs per number of clients

7 Conclusions and Future Directions

In this paper, we presented the design and implementation of a modular and extensible framework for Blockchain-based Federated Learning that supports Vertical Federated Learning. In addition, we presented its performance evaluations results in terms of execution time, transaction cost, transaction latency, model accuracy and convergence, as well as communication and computation costs.

In the future, it would be interesting to investigate how to make BlockLearning more generic in order to support other Vertical Federated Learning models. In addition, it would be valuable to incorporate the Private Set Intersection phase into our framework, allowing it to be directly applied to cases where the clients have intersecting, but not equal, sample spaces.

References

1. Abadi, M., et al.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015). https://www.tensorflow.org
2. Awan, S., Li, F., Luo, B., Liu, M.: Poster: a reliable and accountable privacy-preserving federated learning framework using the blockchain. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pp. 2561–2563. CCS 2019, Association for Computing Machinery, New York, NY, USA (2019)
3. Benet, J.: IPFS - content addressed, versioned, p2p file system (2014)
4. Cai, H., Rueckert, D., Passerat-Palmbach, J.: 2CP: decentralized protocols to transparently evaluate contributivity in blockchain federated learning environments (2020)
5. Cao, M., Zhang, L., Cao, B.: Toward on-device federated learning: a direct acyclic graph-based blockchain approach. IEEE Trans. Neural Netw. Learn. Syst. 1–15 (2021)

6. Chen, H., Asif, S.A., Park, J., Shen, C.C., Bennis, M.: Robust blockchained federated learning with model validation and proof-of-stake inspired consensus (2021)
7. ConsenSys: Consensys/quorum: A permissioned implementation of ethereum supporting data privacy
8. Contributors, S.: Solidity 0.8.15 documentation (2021)
9. Cui, L., et al.: CREAT: blockchain-assisted compression algorithm of federated learning for content caching in edge computing. IEEE Internet of Things J. **9**, 14151–14161 (2020)
10. Desai, H.B., Ozdayi, M.S., Kantarcioglu, M.: Blockfla: accountable federated learning via hybrid blockchain architecture. In: Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy, pp. 101–112. CODASPY 2021, Association for Computing Machinery, New York, NY, USA (2021)
11. Dias, H.: Impact Analysis of Different Consensus, Participant Selection and Scoring Algorithms in Blockchain-based Federated Learning Systems Using a Modular Framework. Master's thesis, TU Eindhoven (2022)
12. Jin, T., Hong, S.: Split-CNN: splitting window-based operations in convolutional neural networks for memory system optimization. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 835–847. ASPLOS 2019, Association for Computing Machinery, New York, NY, USA (2019)
13. Kim, H., Park, J., Bennis, M., Kim, S.L.: Blockchained on-device federated learning. IEEE Commun. Lett. **24**(6), 1279–1283 (2020)
14. Korkmaz, C., Kocas, H.E., Uysal, A., Masry, A., Ozkasap, O., Akgun, B.: Chain FL: decentralized federated machine learning via blockchain. In: 2020 Second International Conference on Blockchain Computing and Applications (BCCA), pp. 140–146 (2020)
15. LeCun, Y., Cortes, C., Burges, C.: MNIST handwritten digit database. ATT Labs (2010). http://yann.lecun.com/exdb/mnist
16. Li, D., et al.: Blockchain for federated learning toward secure distributed machine learning systems: a systemic survey. Soft Comput. **26**, 4423–4440 (2021)
17. Lu, Y., Huang, X., Zhang, K., Maharjan, S., Zhang, Y.: Blockchain empowered asynchronous federated learning for secure data sharing in internet of vehicles. IEEE Trans. Veh. Technol. **69**(4), 4298–4311 (2020)
18. Martinez, I., Francis, S., Hafid, A.S.: Record and reward federated learning contributions with blockchain. In: 2019 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), pp. 50–57 (2019)
19. McMahan, H.B., Moore, E., Ramage, D., Hampson, S., Arcas, B.A.: Communication-efficient learning of deep networks from decentralized data. In: Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, vol. 54, pp. 1273–1282 (2017)
20. Merkel, D.: Docker: lightweight Linux containers for consistent development and deployment. Linux J. **2014**(239), 2 (2014)
21. Mugunthan, V., Rahman, R., Kagal, L.: Blockflow: an accountable and privacy-preserving solution for federated learning. ArXiv (2020)
22. Nagar, A.: Privacy-preserving blockchain based federated learning with differential data sharing (2019)
23. Passerat-Palmbach, J., Farnan, T., Miller, R., Gross, M.S., Flannery, H.L., Gleim, B.: A blockchain-orchestrated federated learning architecture for healthcare consortia (2019)

24. Paszke, A., et al.: Pytorch: an imperative style, high-performance deep learning library. In: Wallach, H., Larochelle, H., Beygelzimer, A., d' Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 32, pp. 8024–8035. Curran Associates, Inc. (2019)
25. Peyvandi, A., Majidi, B., Peyvandi, S., Patra, J.C.: Privacy-preserving federated learning for scalable and high data quality computational-intelligence-as-a-service in society 5.0. Multimed. Tools Appl. **81**, 25029–25050 (2022)
26. Pfitzner, B., Steckhan, N., Arnrich, B.: Federated learning in a medical context: a systematic literature review. ACM Trans. Internet Technol. **21**(2), 1–31 (2021)
27. Ramanan, P., Nakayama, K.: Baffle: blockchain based aggregator free federated learning. In: 2020 IEEE International Conference on Blockchain (Blockchain), pp. 72–81 (2020)
28. Romanini, D., et al.: PyVertical: a vertical federated learning framework for multi-headed SplitNN (2021)
29. Van Rossum, G., Drake, F.L.: Python 3 Reference Manual. CreateSpace, Scotts Valley, CA (2009)
30. Wang, Z., Hu, Q.: Blockchain-based federated learning: a comprehensive survey (2021)
31. Weng, J., Weng, J., Zhang, J., Li, M., Zhang, Y., Luo, W.: Deepchain: auditable and privacy-preserving deep learning with blockchain-based incentive. IEEE Trans. Dependab. Secur. Comput. **18**(5), 2438–2455 (2021)
32. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. Ethereum Proj. Yellow Paper. **151**, 1–32 (2014)
33. Yang, Q., Liu, Y., Chen, T., Tong, Y.: Federated machine learning: concept and applications. ACM Trans. Intell. Syst. Technol. **10**(2), 1–9 (2019)
34. Zhang, Q., Palacharla, P., Sekiya, M., Suga, J., Katagiri, T.: Demo: a blockchain based protocol for federated learning. In: 2020 IEEE 28th International Conference on Network Protocols (ICNP), pp. 1–2 (2020)
35. Zhang, W., et al.: Blockchain-based federated learning for device failure detection in industrial IoT. IEEE Internet Things J. **8**(7), 5926–5937 (2021)