Record and Reward Federated Learning Contributions with Blockchain

Ismael Martinez*
Department of Computer Science and Operations Research
University of Montreal
Montreal, Canada
ismael.martinez@umontreal.ca

Sreya Francis*
Montreal Institute of Learning Algorithms
University of Montreal
Montreal, Canada
sreya.francis@umontreal.ca

Abdelhakim Senhaji Hafid
Department of Computer Science and Operations Research
University of Montreal
Montreal, Canada
ahafid@iro.umontreal.ca

***Abstract*—Although Federated Learning allows for participants to contribute their local data without it being revealed, it faces issues in data security and in accurately paying participants for quality data contributions. In this paper, we propose an EOS Blockchain design and workflow to establish data security, a novel validation error based metric upon which we qualify gradient uploads for payment, and implement a small example of our blockchain Federated Learning model to analyze its performance.**

***Index Terms*—blockchain, Federated Learning, distributed machine learning, class sampled validation error**

I. Introduction

In today's data market, users generate data in various forms including social media behaviour, purchasing patterns, and health care records, which is then collected by firms and used either for sale or for in-house data analytics and machine learning. As a result, each of us is giving away a personal resource for no reward. In addition, these organizations have full access to our data, which can be a major invasion of privacy depending on the type of data collected. One proposed method of mitigating this issue of ownership and privacy when the purpose of the data is proprietary machine learning is *Federated Learning* [1] [2], where an owner sends the training model to users who train on their local data and send back only the updated weights of the model. By doing this, a user never unveils his data to the owner, and keeps ownership of his data. A secondary result of this type of training is that users with sensitive data such as health care data are more likely to partake in the training, meaning the owner also receives more data to use for training.

There still remains the concern of handing out our data, a useful resource to organizational training models, for free. We propose the use of blockchain to facilitate the uploading and tracking of updates from users, as well as rewarding users for the data they used in computation. An additional benefit to using a blockchain is that it renders the updates immutable and thus secure. The combination of data privacy and security coupled with rewards for uploads renders this system desirable to a larger scope of people, allowing organizers to collect a larger pool of data from a wider set of users.

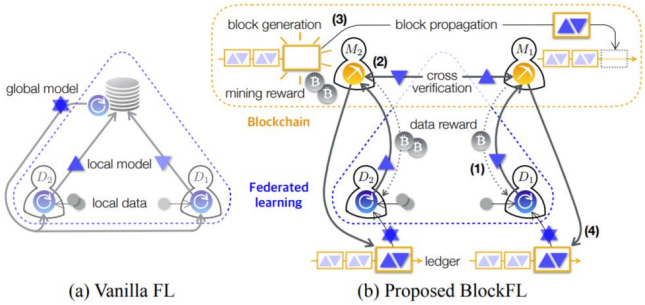


Fig. 1: As described in the proposal for BlockFL [3], the architecture of BlockFL compared to "Vanilla" Federated Learning [2].

BlockFL [3] uses blockchain to reward users for their local updates proportional to how many local data points are used as shown in Fig. 1. The payment to devices is left to the miner to pay "out-of-pocket", which is not a lasting solution if miners pay devices more than they are rewarded for blocks. This device reward benefits an honest node; however, this value may be inflated by a malicious node seeking higher reward.

DeepChain [4] proposes an incentive-based blockchain mechanism to reward honest participants and penalize dishonest participants. The blockwise-BA consensus protocol proposed relies on cryptographically selecting a worker to create a block which is validated by a committee; this method relies on choosing an honest committee, and for the random algorithm to be negligibly close to perfectly random, both issues which may not be true in practice.

Kurtulmus and Daniel [5] proposed an Ethereum blockchain implementation of machine learning to reward users for pro-

*Equal contribution

ducing trained models for organizers. Given an organizer's published dataset and evaluation function, users compete to produce the first or the best training model that maximizes this evaluation function. One large problem that arises with this system is that all model evaluations are done on the blockchain which yields large gas costs; many users must each pay gas for their models to be evaluated, however only one or two users are paid out. Users needing to pay large gas prices in addition to effort and time into building and training a machine learning model for submission without a guarantee of repayment does not yield a sustainable system.

Another study [6] looks into Distributed Machine Learning and Federated Learning with an Ethereum blockchain reward based on the evaluation of the trained model. Participants train a global model with local data and upload the model parameters via IPFS; if the evaluation of the uploaded model passes a predefined Minimum Acceptable Fitness Rate (MAFR) threshold, the participant is rewarded in Ether. Although it is stated a participant is rewarded based on the value of their contribution, both the evaluation and the amount are not well defined nor tested, and a public knowledge of a low MAFR threshold could lead to participants purposely contributing their data in smaller batches across more training rounds to maximize reward.

Similarly, [7] attempts to make a distributed machine learning marketplace and Federated Learning training system with users rewarded in DML tokens through Ethereum Smart Contracts. The choice to use a proprietary token with the sole purpose of interacting, buying, and selling within the system may be a deterrent to users seeking a more stable cryptocurreny.

The limitations of the related work can be summarized as inaccuracy or inefficiency in rewarding user contributions, and lack of scalability of data on the blockchain.

The main contributions of this paper can be summarized as follows:

- Merging Federated Learning with blockchain to ensure both data privacy [1] [2] and security, and thus motivate more user contributions.
- Using EOS Blockchain and IPFS to *record* uploaded updates in a sclalable manner and *reward* users based on training data cost.
- Proposal of a Class-Sampled Validation-Error Scheme (CSVES) for validating and rewarding only valuable uploaded updates via Smart Contracts. Two validating criteria of error trend (CSVES-Tr) and error threshold (CSVES-Th) are examined and evaluated.
- Simple implementation with Python and Hyperledger Fabric to verify the feasibility of the system, with plans to implement a PoC in EOS at a later date.

The rest of this paper is organized as follows. In Section II, we propose an architecture for achieving Federated Learning with an EOS Blockchain. In Section III, we implement a version of our solution with assumptions using both Python and Hyperledger Fabric. In Section IV, we look at future work in research and experimentation. Finally, we conclude the paper in Section V.

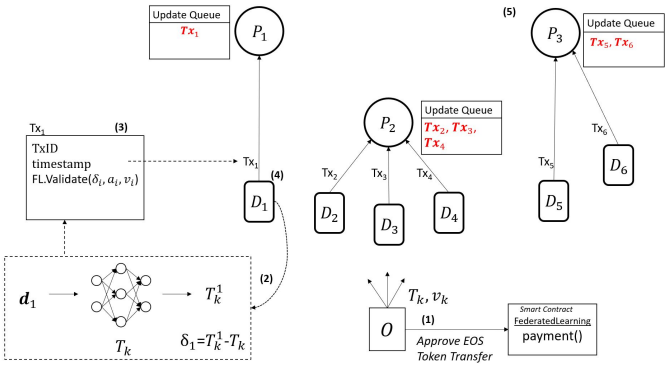


Fig. 2: The workflow of calculating and uploading update values δ for validation, as explained in Section II-D.

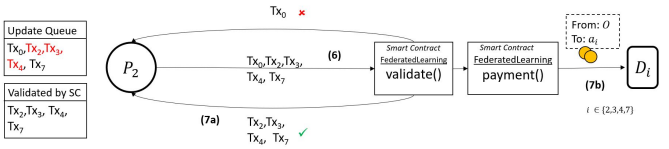


Fig. 3: The continued workflow of validating the δ via Smart Contracts and paying successful candidates, as explained in Section II-D.

II. Proposed Design and Architecture

Taking related work into consideration, we choose to make adjustments to improve privacy, access control and storage; the architecture and workflow of the proposed system are shown in Fig. 2 and 3.

The following assumptions are made regarding the devices and data in the system.

- A smartphone device has enough storage to store the current global model during training.
- A smartphone device does not necessarily have enough extra storage to store the entire blockchain.
- The training data for the model is homogeneous across different devices.

A. System and Blockchain Architecture

For our system design, we are using EOS Blockchain, a public blockchain with no transaction fees which further incentivizes its use by users [9] [10]. EOS uses a set of 21 *producers* to create blocks simultaneously, creating an extremely scalable blockchain able to process millions of transactions per second. In our system, the model owner $\mathcal{O}$ has full liability of payment for the device and producer work, as opposed to devices D needing to pay for their transactions [5], or miners to reward devices out-of-pocket [3].

Our base implementation of Federated Learning is built with smartphone devices in mind who act as the users performing the training and sending in δ values. The model owner $\mathcal{O}$ defines the initial model and distributes the reward.

TABLE I: Parameters required in a transaction upload from device D_i.

Parameter	Purpose	Size	Section
TxID	Unique identifier of the transaction	256 bits	
a_i	Address of device D_i	160 bits	
$H(\delta_i)$	SHA256 Hash of binary representation of training model weight updates	˜256 bits	Section II-D
n_i	The data cost – number of data points used to calculate the model update	16 bits	Section II-D
v_i	The current version k of T_k	16 bits	Section II-B
v, r, s	Signature of hash of the transaction	65 bits	

A transaction in our system carries the information needed for the Federated Learning process. For a user D_i and a global model T_k, we define these transaction parameters below.

The *gradient upload* δ_i is the binary representation of the weight updates to the model. For a received model T_k, T_k^i is the result of training the model on local data $\boldsymbol{d_i}$ and our gradient is the difference $\delta_i = T_k^i - T_k$. Regardless of what data is used, the value of δ_i will be quite large. For example, the single channel MNIST image data [11] is 1 MB per update δ_i. The way we record these values within the blockchain is to store signed transactions in a table off-chain within the IPFS file system [12], and record only the hash of the gradient value $H(\delta_i)$ on-chain. This process is shown in Fig. 5. This means that when the Smart Contract validates the format of δ, it must use an oracle to access the value in the table off-chain. When the owner updates the model, it must grab the gradients from this same off-chain table by querying IPFS for the document with the same on-chain gradient hash.

The *data cost* $|\boldsymbol{d_i}| = n_i$ of D_i is the number of datapoints used for training the model T_k to obtain δ_i. The amount rewarded to D_i for its gradient δ_i is proportional to the data cost n_i. For a simulation of users training on the MNIST dataset, we can see in Fig. 4 that the number of datapoints is approximately linear to the training time, justifying our choice to reward users proportional to number of datapoints used as it has a strong linear relationship with training effort.

The *version* of the model being used is the integer value k of the current Global Model T_k. If the uploaded version v_i does not match the current version k, either the update value δ_i needs to be adjusted, or the value δ_i should be dropped.

The *address* a_i is the network address of the device D_i where payment is to be sent.

In addition to these values, each transaction has a *transaction id* to uniquely identify a transaction, and a *signature* created by the user's private and public key pair and denoted by {v, r, s} values. A summary of the transaction values is available in Table I.

B. Global Model

The *initial model* T_0 has all weights initialized to normally distributed values with mean 0 and variance 1; therefore, we define each weight $w \sim \mathcal{N}(0, 1)$.

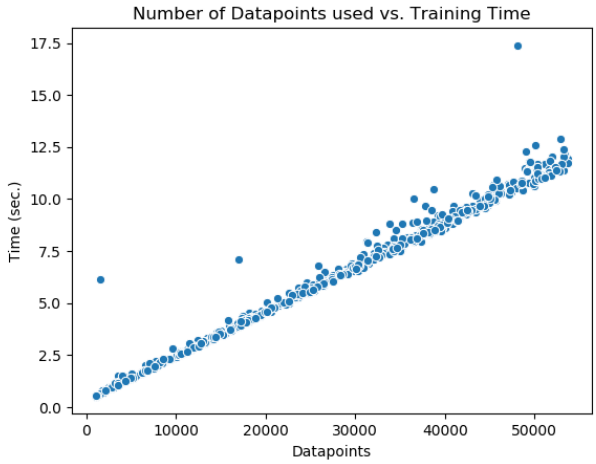


Fig. 4: Simulation of 500 users training with the MNIST dataset with number of points in [320, 60,000] shows a strong linear relationship between number of points used for training and training time.

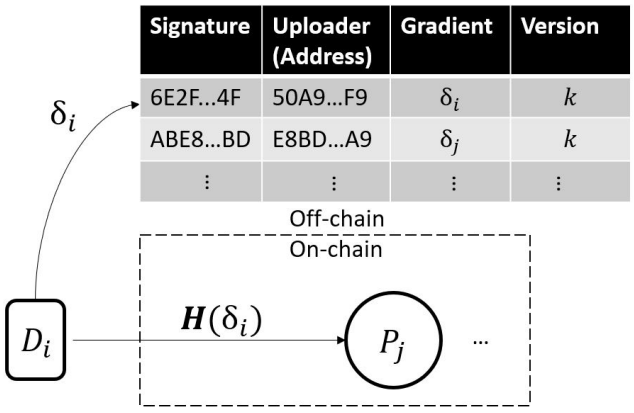


Fig. 5: A device uploads the gradient value to an off-chain table within the IPFS file system where it is later accessed by the Smart Contract for validation, and the owner for gradient aggregation. Only the hash of the gradient remains on-chain.

To calculate T_1, the owner $\mathcal{O}$ applies the aggregate of all δ_i updates from the blockchain where version $v_i = 1$. We denote the number of such updates by b_1. We define w_l as being the lth weight in T_0, and $\delta_{i,l}$ as the lth weight of the ith gradient value; the training model will apply the update

$$w_l \leftarrow w_l + \eta \cdot \bar{\delta}_l = w_l + \eta \cdot \frac{1}{b_1} \sum_{i=1,\ldots,b_1} \delta_{i,l}$$

to each $w_l \in T_0$, where η is the *learning rate* [13].

Similarly, to calculate T_{k+1}, the owner $\mathcal{O}$ applies the update

$$w_l \leftarrow w_l + \eta \cdot \bar{\delta}_l = w_l + \eta \cdot \frac{1}{b_{k+1}} \sum_{i=1,\ldots,b_{k+1}} \delta_{i,l}$$

to each $w_l \in T_k$. If the most recent version for which the uploads have been aggregated is version K, then T_K is known as the *Global Model*.

C. Smart Contracts

We propose the creation and usage of three Smart Contracts in our system.

`UploadGradient` – This Smart Contract verifies that every parameter as described in Table I is present in the transaction. It compares the hash of the transaction update value δ_i with the same value on the off-chain IPFS record as per Fig. 5, ensures that the hashes are equal, that the true value of δ_i follows the required size and format requirements for T_k as set by $\mathcal{O}$, and verifies the submitted version v_i is the same value as the last version value v_k sent out by $\mathcal{O}$ (Section II-A). Once a transaction is validated as being a proper update transaction, the Smart Contract returns `True`.

`Payment` – Once a transaction from device $D_i \in \mathcal{D}$ is validated, this Smart Contract is triggered to send payment proportional to the data cost n_i to address a_i. Since we are using EOS, this payment would take the form of an EOS tokens of which $\mathcal{O}$ has given prior approval to the Smart Contract to transfer to devices.

`FederatedLearning` – This is the only Smart Contract a user may call directly. This function forwards the user's transaction to the `UploadGradient` Smart Contract; if `True` is returned, this function then makes a call to `Payment()`. The parameters required by this function for a device D_i are $H(\delta_i), n_i, v_i$ and a_i, and are forwarded to calls to `UploadGradient()` and `Payment()`.

D. System Design and Workflow

We define $\mathcal{D}$ to be the set of all devices and $\mathcal{P}$ to be the set of all producers. As in Fig. 2, step (1), each device $D_i \in \mathcal{D}$ has a copy of T_k given to it by $\mathcal{O}$, along with the current version v_k; this is defined as *round* k. We walk-through the process of training the next model, validating the transactions and paying the users, referring to Fig. 2 and 3.

For each device $D_i \in \mathcal{D}$ upon receiving the kth model T_k, (2) D_i trains the model T_k off-chain using it's local data $\boldsymbol{d_i}$ of size n_i to get an updated model T_k^i. The gradient is then calculated as $\delta_i = T_k^i - T_k$. (3) The values of δ_i, the size of the dataset n_i, the device address a_i and the version v_k are set as parameters to the Smart Contract `FederatedLearning()` together with a TxID and a digital signature. (4) The device D_i sends this transaction on-chain to any producer P_j to which D_i is connected. (5) Each producer $P_j \in \mathcal{P}$ adds the transactions received by the devices to their transaction queue (i.e., pending transactions) to be verified for the next block. (6) Each producer executes each transaction in its queue, which involves a call to `FederatedLearning()` to validate the user role, then a call to `UploadGradient()` where details about each transaction are validated, such as the correct format for δ_i, the correct version v_i, and that the address a_i exists. (7a) Once validated, this transaction is added to the next block; (7b) the device $D_i \in \mathcal{D}$ who submitted a valid transaction is rewarded through a call to `Payment()` an amount proportional to the data cost n_i via the submitted address a_i.

E. Data Validity and Quality

The validation check we have defined earlier requires producers to trust that the data cost a device claims to have used is correct. We propose a new concept of tailoring a validation set to a device's data breakdown which we will call a *Class-Sampled Validation Error Scheme* (CSVES).

This proposed method, shown in Fig. 6, begins prior to training. We define the set of all classes available as $C = \{C_1, C_2, ..., C_p\}$ for p classes. For a device $D \in \mathcal{D}$, we define the set C_D as the set of all classes for which D has data. Then, $C_D \subseteq C$ because there may exist a class $C_i \in C$ such that for all local datapoints $d \in \boldsymbol{d}$, $d \notin C_i$. D sends the set C_D to $\mathcal{O}$ and receives a validation set with datapoints chosen only from the classes in C_D from an off-chain dataset belonging to $\mathcal{O}$. Once received, D begins to train model T_k with the local training dataset $\boldsymbol{d}$ and the received validation set. During training, the model will intermittently apply the validation set to the training model and record the validation error; if the model is improving, we expect the validation error to decrease. At the end of training, D will send the validation errors alongside other parameters for the function call `UploadGradient()`. We modify this function to look at the general trend of the validation errors over training time; if the validation errors are decreasing, we say δ is a valuable and valid gradient update, and we reward D based on its data cost n. The CSVES algorithm from the perspective of a device, the owner and the producer are outlined in Algorithm 1, 2 and 3 respectively. Another method of verifying the validation error can be done using a threshold τ set by $\mathcal{O}$ – this method affects only the producer and is described in Algorithm 4. We will hereby specify the CSVES variants of validating error over trend and by threshold as CSVES-Tr and CSVES-Th respectively.

This algorithm does not require any trust of the devices who can inflate their data cost n for a higher reward; however, it is possible for either a faulty gradient δ to appear valuable and thus be rewarded, or for a valid gradient δ to have an increasing validation error if the datapoints used by D don't reflect the data in the received validation set and thus not be rewarded.

The most obvious issue with CSVES is that it is only defined here for classification problems; it would be useful to find other similar schemes for tailoring validation sets based on non-classification data. Simulation results in Table II show that CSVES-Tr validates user data quality %60 of the time, whereas Table III shows CSVES-Th validates proper data at a near perfect rate for the given threshold $\tau = 0.1$. Faulty data was simulated for in Table IV and Table V for CSVES-Tr and CSVES-Th respectively; the acceptance rates for CSVES-Tr and CSVES-Th average at %33 and %75 respectively, suggesting the CSVES algorithm to be a good starting point for developing a stronger data quality acceptance scheme, but

Algorithm 1: Class-Sampled Validation-Error Scheme: Device

Result: *Hash*(δ), v_E
Parameter: Device $D \in \mathcal{D}$
// Assign c_i based on the number of datapoints in class i
for $i \leftarrow 1$ **to** p **do**
  if $|\boldsymbol{d}_i| > 0$ **then**
    $c_i \leftarrow 1$;
  else
    $c_i \leftarrow 0$;
  end
end
Send $\{c_1, \ldots, c_p\}$ to $\mathcal{O}$;
// Await $\boldsymbol{v}$ from $\mathcal{O}$
$T'_k, v_E \leftarrow (T_k, \boldsymbol{v})$;
// v_E is the resulting validation error vector from training
$\delta \leftarrow T'_k - T_k$
Send *Hash*(δ), v_E, and dataset size n to nearest Producer

Algorithm 2: Class-Sampled Validation-Error Scheme: Owner

Result: Vector $\{c_1, \ldots, c_p\}$
Parameter: Vector $\{c_1, \ldots, c_p\}_D$, m
$\boldsymbol{v} \leftarrow$ empty vector size p;
for $i \leftarrow 1$ **to** p **do**
  if $c_i = 1$ **then**
    // Take a random sample of size m from off-chain dataset for class i
    $\boldsymbol{v}[i] \leftarrow$ randomSample(m, i);
  end
end
Send $\boldsymbol{v}$ to D

Algorithm 3: Class-Sampled Validation-Error Scheme: Producer - Error Trend

Result: Vector $\{c_1, \ldots, c_p\}$
Parameter: *Hash*(δ), v_E, n
// Determine if the overall trend of the validation error is decreasing
Given the vector v_E, determine the line of best-fit $y = \beta_0 + \beta_1 x$;
if $\beta_1 < 0$ **then**
  Send Payment(n) to D
end
.
Upload δ to the Blockchain.

Algorithm 4: Class-Sampled Validation-Error Scheme: Producer - Error Threshold

Result: Vector $\{c_1, \ldots, c_p\}$
Parameter: *Hash*(δ), v_E, n
// Determine if the final validation error metric surpasses a threshold τ
Given the vector v_E, determine if the last value $v_E[-1] \leq \tau$;
if $v_E[-1] < \tau$ **then**
  Send Payment(n) to D
end
.
Upload δ to the Blockchain.

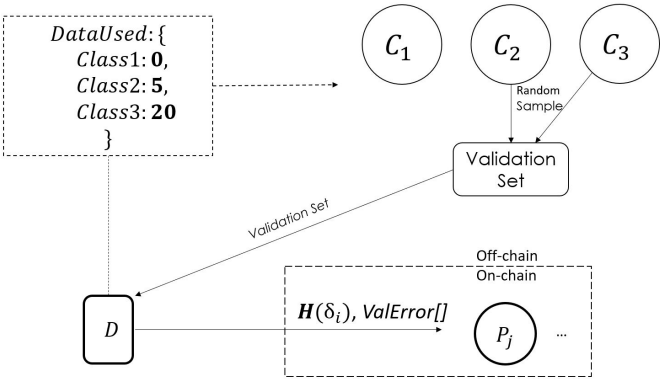


Fig. 6: Prior to training, a device $D \in \mathcal{D}$ sends a list of the classes for which it has data, and receives a validation set containing data from only those classes. Once the validation set is received, training proceeds and the set of validation errors throughout training is sent along with the gradient δ.

not a great approach on its own. Furthermore, this scheme does not *prove* that the data cost n of a user is correct, only that the data itself is valuable enough to merit the requested reward. Similar schemes based on a combination of validation error trend and threshold should be explored to validate data quality, and new schemes should be developed to prove the data cost n of a user.

F. Restrictions on Users via Smart Contracts

As noted in [14] and [15], even when only uploading the gradient it is still possible for the owner $\mathcal{O}$ or other parties to infer a user's data from the uploaded gradient. Although we are using a public blockchain, neither the full gradients nor the training model are on the blockchain, keeping them both hidden from the public. Training Model privacy is achieved through the use of Paillier's Cryptosystem, a homomorphic encryption scheme commonly used in distributed machine learning [4]; gradient privacy is achieved by uploading the full gradient to an off-chain table on IPFS only accessible by $\mathcal{O}$ as defined in Section II-A.

This blockchain system is intended to restrict the actions taken by users participating in the Federated Learning process.

TABLE II: Simulation of valid data acceptance rates via CSVES-Tr.

# of Classes	Instances	Accepted	% Accepted
1	8	0	%0.000
2	49	27	%55.102
3	122	81	%66.393
4	196	121	%61.135
5	266	108	%59.399
6	188	114	%60.600
7	110	64	%69.182
8	50	31	%62.000
9	11	7	%63.636

[a] Simulation of training 1000 instances on the MNIST dataset with 5 epochs each of 1000 training points and 100 validation points.

TABLE III: Simulation of valid data acceptance rates via CSVES-Th with $\tau = 0.1$.

# of Classes	Instances	Accepted	% Accepted
1	9	5	%55.555
2	41	41	%100.000
3	112	112	%100.000
4	211	211	%100.000
5	240	238	%99.167
6	203	203	%100.000
7	119	119	%100.000
8	50	50	%100.000
9	13	12	%92.308

[a] Simulation of training 1000 instances on the MNIST dataset with 5 epochs each of 1000 training points and 100 validation points.

TABLE IV: Simulation of faulty data acceptance rates via CSVES-Tr.

# of Classes	Instances	Accepted	% Accepted
1	14	3	%21.429
2	44	9	%20.455
3	121	36	%29.752
4	194	54	%27.835
5	272	98	%36.029
6	191	75	%39.267
7	117	41	%35.047
8	35	20	%57.143
9	9	3	%33.333

[a] Simulation of training 1000 instances on the MNIST dataset with 5 epochs each of 1000 training points and 100 validation points.

TABLE V: Simulation of faulty data acceptance rates via CSVES-Th with $\tau = 0.1$.

# of Classes	Instances	Accepted	% Accepted
1	11	2	%18.181
2	61	28	%45.902
3	122	88	%72.131
4	196	166	%84.694
5	240	212	%88.333
6	220	185	%84.091
7	111	102	%91.892
8	34	32	%94.118
9	5	5	%100.00

[a] Simulation of training 1000 instances on the MNIST dataset with 5 epochs each of 1000 training points and 100 validation points.

We define a *member* x of the blockchain as either a device $x \in \mathcal{D}$, a producer $x \in \mathcal{P}$ or the model owner $x = \mathcal{O}$. The only restriction we put is that $\mathcal{O} \cap \mathcal{D} = \varnothing$, meaning $\mathcal{O}$ doesn't take part in the Federated Learning training process and none of the devices take part in the model aggregation process.

To achieve this, we make use of Smart Contracts to validate a user's role against a set of rules prior to forwarding its transaction to the appropriate Federated Learning Smart Contracts. In this Smart Contract called `FederatedLearning()`, we define the owner $\mathcal{O}$ as not being able to submit any transactions that call the `UploadGradient()` Smart Contract, thus impeding it from contributing its own data to the process. Furthermore, we define a rule to limit a device's only action as submitting a transaction with a call to `UploadGradient()`. Since it is left to the owner $\mathcal{O}$ to trigger the next training round, $\mathcal{O}$ retains control of deciding for how many rounds the training takes place.

We also want to restrict each device from uploading more than one gradient to the model for each version v_k. One method is to assign a nonce to each user upload per version in `FederatedLearning()` to keep track of whether a user has already uploaded for a particular version. In practice, we want a device to upload the gradient obtained from their best dataset; due to our restriction, it is suggested that devices are confident with their off-chain dataset and training process before uploading their one and only transaction for round k.

III. Proof of Concept

We made a small implementation of our design in order to test out the general workflow in practice. For this implementation, we used 15 training rounds of 10 local `Device` participants who performed the model training off-chain in Python, and sent transactions to the Hyperledger Fabric blockchain. Future plans include a larger implementation of this system with EOS blockchain.

The Proof of Concept seeks to answer whether a blockchain could work with Federated Learning implementation in Python to record and reward gradient uploads; since we're using a REST API to interact with Hyperledger Fabric, then any

programming language that supports API calls can interact with our blockchain system.

A. Hyperledger Fabric - REST API

For this implementation on Hyperledger Fabric we made use of Hyperledger Fabric Composer where we defined the files `model.cto` – where we define the *participants*, *assets* and *transactions*, `logic.js` – where we define the Smart Contract chain code, and `permissions.acl` – where we define the ruleset restricting/allowing the actions of the participants.

The `model.cto` defines the participants, assets, and functions of our system. We can have multiple `Device` participants, which make up the users in our Federated Learning process. Although training of the model occurs off-chain, we have a single instance of a `TrainingModel` asset which we use to keep track for which *version* we are currently uploading gradient values. A `Gradient` asset is linked to a `Device` participant, and has the *hash* of the gradient as in Fig. 5, the current training *version*, and the *dataCost* claimed by the *Device* participant. The `Token` asset is also linked to a `Device` participant, has the current training *version*, and a *value* which is equal to the claimed *dataCost*. The `UploadGradient()` transaction is the parameter description of our Smart Contract, requires a `Device` participant instance and the `TrainingModel` asset instance, and has the *hash*, *dataCost*, and *version* parameters which we've seen in the other assets.

The `logic.js` file is the *chaincode* of our application which is HyperLedger Fabric's version of a Smart Contract written in pure JavaScript. In this function, we have decided to combine the `Payment()` functionality within the same `UploadGradient()` function – the script validates the submitted parameters, creates the necessary assets, and rewards the user for their upload. The script follows these steps:

`UploadGradient(`**`tx`**`.{Device, TrainingModel,` *version*, *hash*, *dataCost*`})`

1) *Validation*
   a) Verify that the uploaded **`tx`**.*version* is equal to the current **`tx`**.`TrainingModel`.*version* attribute.
2) *Create new Gradient*
   a) Create a blank asset of type `Gradient` with unique id <**`tx`**.`Device`.*deviceId*_**`tx`**.*version*>.
   b) `Gradient`.*device* ← **`tx`**.`Device`
   c) `Gradient`.*version* ← **`tx`**.*version*
   d) `Gradient`.*hash* ← **`tx`**.*hash*
3) *Pay user via a Token*
   a) Create a blank asset of type `Token` with unique id **`tx`**.`Device.`*deviceId*_**`tx`**.*version*.
   b) `Token`.*value* ← **`tx`**.*dataCost*
   c) `Token.Device` ← **`tx`**.`Device`

Within Hyperledger Fabric, the *deviceId* of a `Device` is used in the same way as the *address* a_i from our design. We leave it for future work to both implement and test the Class-Sample Validation Error Scheme and to use an Oracle to check the off-chain format of the gradient δ, both as part of the *Validation* phase.

Our `permission.acl` file defines the *allowance* of participants of which actions they can perform. We set the following rules:

- Devices have no access to create or modify `Gradient` assets unless submitting a transaction to `UploadGradient()`
- Devices have no access to create or modify `Token` assets unless submitting a transaction to `UploadGradient()`
- All participants have READ access to the all `Gradient` assets.

This ruleset ensures that participants only see the information they need to see, which are at most all the gradients being uploaded; they do not need to see the rewarded tokens.

To run and interact with the blockchain locally, we deployed the blockchain with a local REST API. Then, we could perform off-chain training and data storage with Python while sending calls to the API to read the blockchain data, getting the list of active participants for whom we need to train, and posting transactions with calls to `UploadGradient()`.

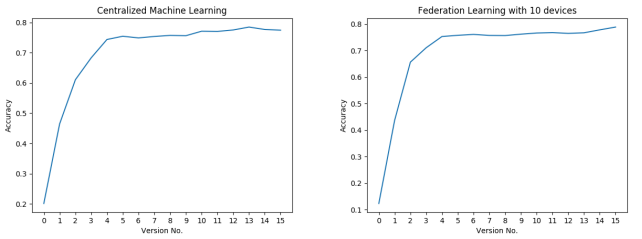
B. Implementation Workflow

We have Python and Hyperledger Fabric working together to achieve Federated Learning using the following workflow.

1) a) Create the Initial Model in Python as per Section II-B.
   b) Create the `TrainingModel` asset in Hyperledger Fabric with *version*: 0.
2) a) Create D local devices in Python, each with a unique id.
   b) Create D `Device` participants with the same id values as in Python.
3) For version $v = k, k \in \{0, ..., V\}$, perform the Federated Learning Process:
   [H] $D \in \mathcal{D}$ $T'_k \leftarrow \texttt{train}(T_k)$
   $\delta \leftarrow T'_k - T_k$
   · Write δ off-chain with *version* and *dataCost*
   · Upload δ, *version* and *dataCost* to Hyperledger Fabric, creating a `Gradient` asset and a `Token` asset
4) Average all gradients with *version* $v = k$ to obtain $\bar{\delta}_k$.
5) Apply the federated aggregation on T_k to obtain
$$T_{k+1} \leftarrow T_k + \eta \cdot \bar{\delta}_k$$
   where $\eta \in (0, 1]$ is the *aggregation factor*.
6) Increment `TrainingModel`.*version* $v \leftarrow k + 1$ on Hyperledger Fabric.

C. Results

For $n = 10$ clients, each with an uneven and overlapping split within the range of 0.5 to 0.1 of the MNIST dataset, and running 15 rounds, we obtain the accuracy metrics of the Global Model shown in Fig. 7. We can see from Fig. 7b that the model improves but quickly plateau with small dips in accuracy without deviating significantly from a centralized approach in Fig 7a. This confirms that the



(a) Centralized dataset. (b) Decentralized dataset over 10 devices.

Fig. 7: Graphical results of training accuracy over 15 versions of a (a) centralized dataset (b) decentralized dataset over 10 devices.

blockchain does not interfere with the Federated Learning process, while recording and rewarding devices for their data in Hyperledger Fabric which we can easily view from the browser's REST API dashboard. This plateau could be due to the lack of diversity of the devices per round, or the lack of diversity in the datasets of each device per round. Ideally, the Federated Learning training model would see new data every round; this type of experimentation is left for future work. Another reason may be that the number of steps taken by the training model is not optimal; we attempted to make every device train the same way by fixing the number of training iterations to do so. The small dips in accuracy could be due to over-fitting on the part of each device as the model become more accurate. For these two reasons, more optimal methods of enforcing a standard training process to avoid over-fitting and to reach training optimization are left for future work [16].

IV. Future Work

Moving forward, further implementation, testing and analysis of variations on CSVES are left as future work to evaluate whether CVES can be altered to be a more accurate scheme for determining the quality or usefulness of local data used for training the model. We also plan to investigate other validation schemes that can accurately determine how much payment a gradient upload should be rewarded, either based on verified number of data points or on evaluated data model improvement. Finally, we acknowledged the value of seeking a standardized form of training such that two devices with the same data calculate the same gradients; such a standard will lead to consistency in uploaded results and fairness in rewards.

V. Conclusion

In this proposal, we addressed the problems of data privacy, security, and fair reward in distributed machine learning using blockchain and Federated Learning. An in-depth workflow was presented for scalable recording and rewarding of gradients using a combination of blockchain and off-chain databases of records. We have also proposed CSVES to validate and verify gradients to determine a reasonable device reward. We implemented a Proof of Concept with a small set of clients and rounds to demonstrate that the blockchain does not interfere with the federated learning aggregation, while limiting the number of uploads and validating the claimed data cost per device. Finally, we composed a list of aspects of Federated Learning and Blockchain that require more in-depth study for implementation as part of future work. With the proposed system, individuals benefit by retaining ownership and receiving incentives for their data, and model owners benefit from access to a larger and more diverse set of client data, leading to more robust and higher performing Machine Learning models.

References

[1] J. Koneny, H. B. McMahan, F. X. Yu, P. Richtrik, A. T. Suresh, and D. Bacon."Federated learning: Strategies for improving communication efficiency", arXiv preprint arXiv:1610.05492, 2016.

[2] B. McMahan and D. Ramage, "Federated Learning: Collaborative machine learning without centralized training data", Google Research Blog, 2017.

[3] H. Kim, J. Park, M. Bennis, and S.-L. Kim. "On-device federated learning via blockchain and its latency analysis", arXiv preprint arXiv:1808.03949, 2018.

[4] J. Weng, J. Weng, J. Zhang, M. Li, Y. Zhang, and W. Luo, "Deepchain: Auditable and privacy-preserving deep learning with blockchain-based incentive", Cryptology ePrint Archive, Report 2018/679, 2018

[5] B. Kurtulmus and K. Daniel. "Trustless machine learning contracts;evaluating and exchanging machine learning models on the ethereum blockchain", arXivpreprint arXiv:1802.10185, 2018.

[6] G. J. Mendis, M. Sabounchi, J. Wei, and R. Roche, "Blockchain as a Service: An autonomous, privacy preserving, decentralized architecture for Deep Learning", arXiv preprint arXiv:1807.02515, 2018

[7] Decentralized Machine Learning, "Decentralized Machine Learning White Paper", https://decentralizedml.com, cited May 2019.

[8] H. B. McMahan, D. Ramage, K. Talwar, H. Zhang, "Learning differentially private language models without losing accuracy", arXiv:1710.06963, 2017.

[9] I. Grigg, "EOS - An introduction', Whitepaper iang.org/papers/EOSAnIntroduction.pdf, 2017.

[10] EOSIO."Eos.io". technicalwhitePaperV2, https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md, cited May 2019

[11] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]", IEEE Signal Processing Magazine 29(6):141142, 2012.

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

[13] T. Ben-Nun, T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis", 2018.

[14] R. C. Geyer, T. Klein, and M. Nabi, "Differentially Private Federated Learning: A Client Level Perspective", arXiv:1712.07557, 2017.

[15] M. Fredrikson, S. Jha ,T. Ristenpart, "Model inversion attacks that exploit confidence information and basic countermeasures", arXiv:1812.03288v1 [cs.LG], 2018.

[16] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, V. Shmatikov, "How to backdoor federated learning", arXiv preprint arXiv:1807.00459v2, 2018.