UPPSALA
UNIVERSITET

PROJECT REPORT

Enhancing Blockchain Based Federated Learning

Jacob Tiensuu, Jialun Song, Fredrik Örn, Maja Linderholm

Project in Computational Science: Report

26 January 2021

Abstract

Federated learning extends traditional machine learning by enabling collaborative learning without having to pool all data in one place. With the use of the so-called edge computing each trainer in the collaborative network preforms training with its local data on a machine learning model, then the model parameters of all new obtained models are aggregated together. This method is useful when collaboratively training a machine learning model on privacy sensitive data since the data isn't shared and stays private. All trainers in the network are connected by a central server which leaves a single point of failure. In order to ensure correct results, the server must be stable, resistant to attacks and trusted by all trainers. If these conditions aren't met the central server can be replaced with a blockchain which is a more tamperproof solution. This is done in Felix Morsbach's master thesis ”Hardened Model Aggregation for Federated Learning backed by Distributed Trust” [5], where the Ethereum blockchain framework replaces a central server. This project extends Morsbach's thesis by changing how aggregation of the new models can be made in a more efficient way with less computations on the blockchain. Our implementation manages to save transaction costs and reduces the overhead without compromising the accuracy of the model or requirements on the hardware.

Contents

1 Introduction **3**

2 Background **4**
- 2.1 Federated Learning 4
- 2.2 Blockchain Technology 5
- 2.3 Previous Work 7
- 2.4 Problem Formulation 8

3 Development tools **9**
- 3.1 Containerisation 9
- 3.2 Machine learning model 9
- 3.3 Tools for Decentralisation 9

4 Implementation **10**
- 4.1 Consensus Policy 10
- 4.2 Secure Voting 13

5 Evaluation **15**
- 5.1 Gas Consumption 15
- 5.2 CPU and Memory Usage 16
- 5.3 Effect of Local Training 17

6 Conclusion **18**
- 6.1 Further work 19

7 Appendix **20**

8 References **20**

1 Introduction

In recent years, terms like artificial intelligence and machine learning are seemingly everywhere and the interest for the subject goes far beyond just the world of data science [1] [2]. Even though the applications seem endless, most machine learning models rely on having access to big data sets for training the model in order to make good predictions. Depending on where the data is coming from, collecting it all in one place might be an issue. One increasingly common source of data is mobile devices and their user data, which often is privacy sensitive. With lots of local data on the mobile devices, it can also be a challenge to transmit all the data to a central server. To tackle this machine learning problem, a group of Google researchers proposed federated learning [3].

Federated learning makes use of edge computing which is distributed computing that brings computations closer to the user, and collaboration by training a machine learning model on each device which send their local model to a central server while the data stays on the device. Then all local model updates are aggregated together, creating a new global model. This process decentralises machine learning by enabling multiple devices to learn collaboratively. By collaborating, all participants can benefit from a model trained on all data while keeping their data private. It also enables the ability for real time predictions because the model makes predictions on a local device in contrast to making predictions on a server and sending it back to the device [3].

While federated learning provides decentralisation into the machine learning field it still uses a central server. This leaves a single point of failure, which means that the central server must be stable, resistant to attacks and trusted by all participants/trainers. If these conditions can't be met further decentralisation is essential [4]. In Felix Morsbach's master thesis "Hardened Model Aggregation for Federated Learning backed by Distributed Trust" [5], Morsbach decentralises the server by using Ethereum blockchain framework. In contrast to letting the central server perform the aggregation, with the use of blockchain the trainers will have that responsibility, resulting in a more tamperproof solution [4]. The Ethereum blockchain is a paid service and uses the cryptocurrency ETH (Ether) which users pay with when executing computations on the blockchain [6]

Morsbach's implementation achieves the goal of removing the central server, but the total communication in the network is increased and a real world application running on the Ethereum blockchain would quickly become costly [5]. The purpose of this project is to improve Morsbach's implementation with a smart policy that allows the network to agree on a new global model without all trainers having to submit their aggregated local updates. This decreases the ETH cost, since less computations are needed on the Ethereum blockchain, and can reduce the total communication cost for the participating devices.

2 Background

2.1 Federated Learning

When training a machine learning model, it is desirable to possess a big data set with a broad range of data. In traditional machine learning a pool of data uses a central server which stores a machine learning model that can make predictions. This architecture requires all data to be centralised in a single location which can be problematic when dealing with classified data which contains sensitive information that only certain parties have access to (e.g medical journals) [3].

In 2017 a group of researchers at Google introduced federated learning which sought to handle problems with having to pool all data in one place, where traditional machine learning was not sufficient. Federated learning deals with these problems by using edge computing which in this case means letting the devices compute and update the model themselves in contrast to letting a central server do it as in traditional machine learning. The data possessed by multiple trainers may be evenly or unevenly distributed between them and is therefore either IID (independent and identically distributed) or non-IID. Federated learning can handle both IID data and non-IID data between the trainers [3].

One iteration in the federated learning process is represented in Figure 1 and further described by the processes 1-3.

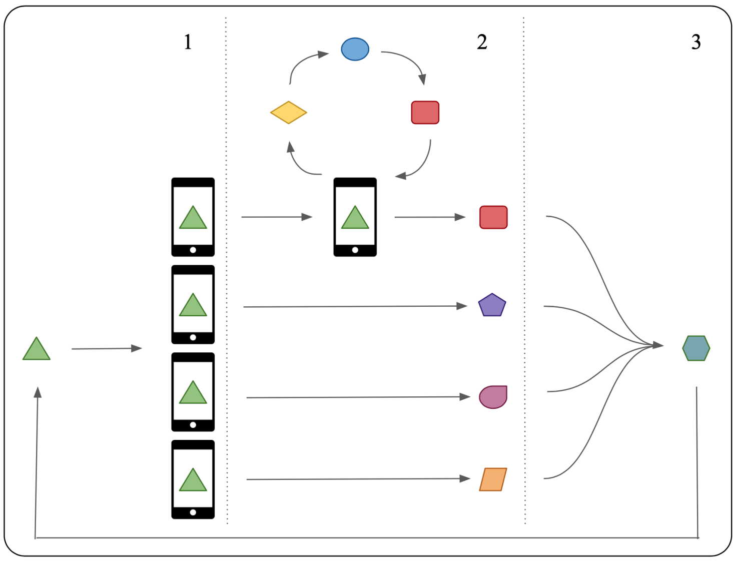

Figure 1: One iteration of the federated learning process. A machine learning model is represented by a geometrical shape and a trainer is represented by a phone. 1) Download the model. 2) Train local models. 3) Aggregate the local models.

1. All trainers download the current global model from a central entity.
2. Using the local data on each trainer device the model will be further trained which will result in a new model.

3. When all trainers have obtained a new model they will be aggregated together by e.g. averaging the model weights. The aggregated model will become the new global model and will be used in step 1 for the next iteration [3].

One iteration of the federated learning process is referred to as a global epoch. At the end of each global epoch a new global model is achieved. In each global epoch we have one or multiple local epochs which occur separately at each trainer during training on local data. In Figure 1 process 1-3 represents a global epoch and process 2 represents one or multiple local epochs (in this figure three local epochs are used) [3].

Federated learning can be suitable for machine learning problems where many trainers possess privacy sensitive data while using the same machine learning model. It may also be suitable for cases where you have multiple devices using the same machine learning model that aims to make predictions in real time with minimum delay. For the second case it is sufficient to have a model that occupies less memory than the data since if the goal is to reduce the execution time it's preferable to send as little data as possible between devices and server. [3].

Federated learning extends traditional machine learning by enabling extra features, but it comes with a bottleneck, the dependency of a central server. The central server which stores the global machine learning model will leave a single point of failure. This means that the central server needs to be stable, resistant to attacks and trusted by all trainers in order to ensure correct results. If the conditions are not met further decentralisation is essential [4]. In the master thesis by Morsbach [5] a blockchain is used to remove the need of a central server and decentralises regular federated learning even further.

2.2 Blockchain Technology

The blockchain concept was first introduced with Bitcoin, the famous first cryptocurrency that made peer-to-peer digital money transactions possible [7]. The blockchain achieves this by removing the need for a trusted central authority, for example a bank, and replacing it with a distributed transaction ledger. The ledger holds all timestamped transactions ever made, stored in ”blocks” by the network and visible to anyone. To make sure transactions are approved by the account making them, digital signatures are used.

To add new transactions to the ledger, a new block needs to be created and added to the chain. The authenticity of the new block is indicated by a cryptographic proof (called proof-of-work), instead of being verified by the trusted authority. The proof uses a hash function that takes the hash of the previous block, the information added in the new block and a random number, called a nonce, as inputs. The inputs are then encoded into to a hexadecimal number of fixed length - the hash. Every unique combination of inputs gives an output that is extremely unlikely to get from another input and the algorithm is one-way, so that the input cannot be calculated from the output.

Block hashes are required to start with a given number of zeros, which makes it harder to find the correct input for a new block. To mine a new block, the random number in the input (the nonce) is varied until it (together with the other inputs) gives an output hash that starts with the correct number of zeros in the beginning. Once a blockchain participant (called a miner) has found such a nonce, the block is added to the chain. The fact that the new block hash starts with correct number of zeros is the proof-of-work [7, 8].

The proof is considered trustworthy since it is very computationally demanding to find the correct hash, there is no better way than to randomly guess numbers for the nonce. To change the data of one block, old or new, also results in changing the block hash. Since the blocks are linked together, all the following blocks must also be changed for the changed chain to look like the real one. For attackers to succeed they need to control a majority of the CPU power in the network. Hence, there is no longer a single point of failure and for large networks, attacks become unfeasible.

2.2.1 Ethereum, Smart Contracts and Gas

Since Bitcoin's origin, a lot of different cryptocurrencies have been introduced and the blockchain infrastructure has been proposed as a way to decentralise other things than money transactions. With the creation of Ethereum, a platform for developing programs that run on blockchain became a reality. Ethereum uses ”smart contracts” which basically is code that is stored and executed on the blockchain. With the use of smart contracts, programmers can define all sorts of applications where actors interact with each other and certain conditions need to be met before functions in the contract can be used. For example, a contract could be used to transfer ownership or to replace the central server in federated learning, which is the case in this project. By keeping the contract on the Ethereum blockchain, it cannot be deleted, it can be clearly stated what kind of updates one could make to the contract and, thanks to digital signatures, who should be able to make them. At the same time the model's security is provided by the immutable blockchain. [9]

(It is worth noting that Ethereum will is switching validation algorithm *proof-of-work* to *proof-of-stake* which means that new blocks are validated in a new way. *Proof-of-stake* is an interesting alternative, but it will not affect the main ideas needed for this project. Too read more on how the algorithm works, see e.g [10].)

In order to work, the Ethereum blockchain (and blockchains in general) need to provide some incentive for miners to add new blocks. Without incentive, there will be very few miners which in turn leads to very little total CPU-power, resulting in the blockchain being sensitive to attacks. The incentive is given partly by a reward given to the miner who mines the new block and partly by transaction fees. The transaction fees are based on the amount of computational steps needed to complete a transaction, measured in ”gas”, where one gas is roughly equivalent to one computational step. When the transaction is complete, you pay for the gas used with ETH, Ethereum's own cryptocurrency. The same computation will always use the same amount of gas, but the ETH price needed to pay for the gas used can vary, see, e.g. [9]. One of the key objectives of this project is to reduce the number of computations on the blockchain needed for the decentralised federated learning model to work, and hence reduce the total gas used.

2.2.2 Decentralised Storage with IPFS

Since one of the key features of the blockchain is that all transactions are stored (forever), it is not suitable for storing large amounts of data. Therefore, storage on the chain is costly, [11]. An alternative to direct storage on the blockchain is to upload the data to an external storage and store the address to the data location on the blockchain. This project uses the InterPlanetary File System (IPFS), a peer-to-peer file system that allows all trainers to share content with each other directly via content addresses, compared to the ordinary web that uses location addresses. That means that when you search for some content it does not matter where it is hosted, IPFS will find others in the network that have the content you are looking for and you will download it from them directly, much like how torrents work. When the content is kept on multiple locations, there is no longer a single point of failure for the data storage and the need for central server to store the model has been removed. This is enough background on IPFS to understand this project, for more details, see [12].

2.3 Previous Work

In [5], a decentralised federate learning system is developed. The implementation combines an Ethereum based blockchain [9] with the decentralised storage system IPFS [13]. In this implementation a smart contract is deployed to the blockchain. The smart contract governs and sets up rules on how the federated learning system should behave, which machine learning method should be used and how trainers interact with each other and the blockchain.

Every trainer is always in one of three possible states: training, aggregation or finished. One round of training and aggregation is called a global epoch. The total number of global epochs are stored on the smart contract, the switching between training and aggregation is repeated until all global epochs has been completed. After the final global epoch, the states are switched to finished for all trainers. A global epoch begins with all trainers in training state where they train a model on the data they own locally. This local model is henceforth referred to as a submodel. Hash-coded address for each submodel are stored on the blockchain and then shared through the IPFS network. When all trainers have obtained their own submodel the IPFS address of each submodel is submitted to the blockchain and the state is switched to aggregation. During aggregation every trainer retrieves the hash-code addresses to every other submodel from the blockchain and downloads them through the IPFS network. Every trainer takes the submodels and combines them into a new model by averaging the weights. Trainers now vote on the new model, by sending a hash-coded address of the model to the blockchain. In Morsbach's implementation all trainers are needed for the process to continue. The smart contract compares the hash-coded address sent by different trainers. If all trainers have sent the same hash-code, they all agree on the same new model. The states are switched to training and a new global epoch can begin. The scheme of switching between training and aggregation is repeated a preset number of times, finally the states are switched to finished.

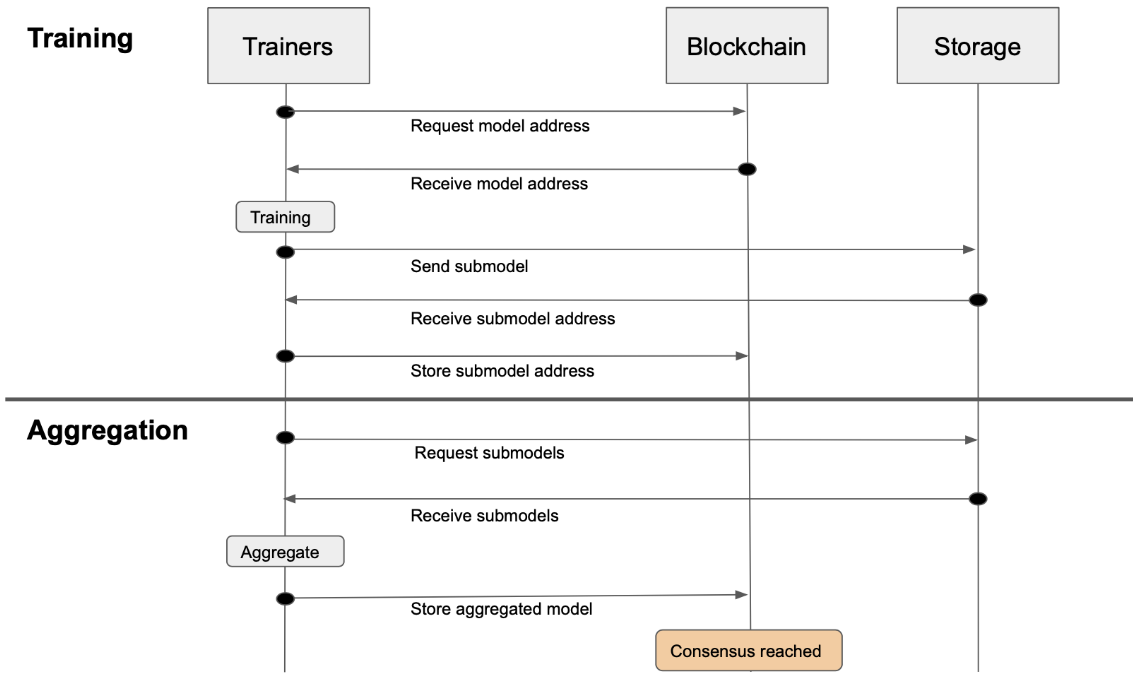

Figure 2: Visualisation of one training state followed by one aggregation state. Together they form one epoch and is repeated a preset number of times.

In Morsbach's implementation [5] every trainer needs to agree on the new model in order for the process to continue. Since the authority to affect the progress of the system is distributed among all trainers, the whole system has very high integrity and any trainer can not tamper with the system for personal gain. However, the cost of the voting process is high and can be delayed if a trainer can not aggregate or send their vote.

The system is implemented to run on both a local scenario and a real world scenario. The local scenario uses a simulated Ethereum blockchain and Redis network [17]. The real world scenario connects to the actual Ethereum and IPFS network. For development purposes the local scenario is used throughout this project.

2.4 Problem Formulation

For this project the voting system is revised by introducing a consensus threshold based solution with the goal of reducing the gas consumption. The consensus threshold allows a smaller group of trainers to verify the new model and commit it to the chain. The solution should reduce the total number of updates needed on the blockchain each global epoch, resulting in less gas consumption. It is desired that the implementation does not affect the required computer resources significantly. Therefore, the total gas consumption, CPU and memory are analysed to evaluate the performance. The voting system is also stabilised, preventing trainers from voting several times during the voting phase.

3 Development tools

In this section, the core technologies and tools applied in this project are introduced. As noted in the previous section, in [5], two scenarios are considered, *local* and *multi-node*, in which two sets of technologies are used respectively besides the basic programming languages. Since the current project focuses on improvements in the local scenario, most of the tools are retained from the previous work, including the simulation method, machine learning model and tools for decentralisation.

3.1 Containerisation

Docker, an OS-level virtualisation software, is used for the local simulation of blockchain-based federated learning. The local set-up uses a virtual network to connect an Ethereum blockchain simulator, a data storage and several computing nodes. By adopting Docker, all these required components can be virtualised and well managed on one computer, so we can focus on the implementation of core logic without putting much effort into the maintenance of the development environment. Furthermore, Docker uses YAML files to explicitly configure all provisions and services, those files are naturally involved in version control, so we can flexibly switch between different set-ups to conduct experiments. Last but not least, as the number of trainers increases in large experiments, the provisioning can be easily achieved by Docker without extra machine-provisions or cluster management.

3.2 Machine learning model

The machine learning part algorithms are not the main focus of the project, and the same model setup has been used as in Morsbach's thesis, with TensorFlow's standard four layered neural network for the MNIST data set [5]. The only part that has been changed directly in the machine learning model is the number of local optimisation epochs each worker makes, since it might reduce the total communication needed in the network, further discussed in Section 5.3.

3.3 Tools for Decentralisation

3.3.1 Smart Contracts in Solidity

To run the smart contract on the Ethereum Virtual Machine, the program must be in bytecode. However, it is easier to write the code in some other language that compiles to bytecode. In this project, the language Solidity is used (as in many other smart contracts), see e.g. [14]. For the workers (based on Go code) to be able to call functions in the contract Go Ethereum, the official Go implementation is used. From Go Ehereum, the source code generator `abigen` allows conversion of smart contracts written in Solidity into Go-packages that can be compiled. For documentation, see [15].

3.3.2 Redis

In order to simulate the IPFS network in the local scenario redis is used [17]. With redis all information is stored with a content address on the local machine.

3.3.3 Blockchain

Making commits to the actual Ethereum blockchain is expensive and therefore we've used Ganache-cli [16]. Ganache-cli is a command line tool to locally simulate an Ethereum blockchain [16]. With the tool we can deploy contracts and commit transaction to the chain. In order to estimate transactions costs the python package Web3 is used to query transactions and extract gas consumption's [18].

4 Implementation

4.1 Consensus Policy

To alleviate the high overhead in the original aggregation process implemented in [5], a consensus policy is introduced into the contract to allow the aggregation result to be decided by a certain proportion of trainers. The policy can be adjusted in different computing environments to reach a sound balance between security and efficiency.

Modifications on both instantiations of the contract and trainer are made to implement this consensus policy. Visualisations from both perspectives are provided in this section to show the qualitative change of the new process. In addition, several examples are given to present how the consensus policy functions in different test cases.

4.1.1 Consensus Threshold in Contract

In the old practice of the contract, it is mandatory to count aggregation results from all the trainers to reach a consensus, which could be interpreted as a requirement of 100% voter turnout. The new policy uses a threshold strategy to lift that requirement, so as long as a candidate receiving enough votes, passing the threshold, the consensus is said to be reached and the process proceeds, as shown in Figure 3. This step of implementation mainly consists of two parts.

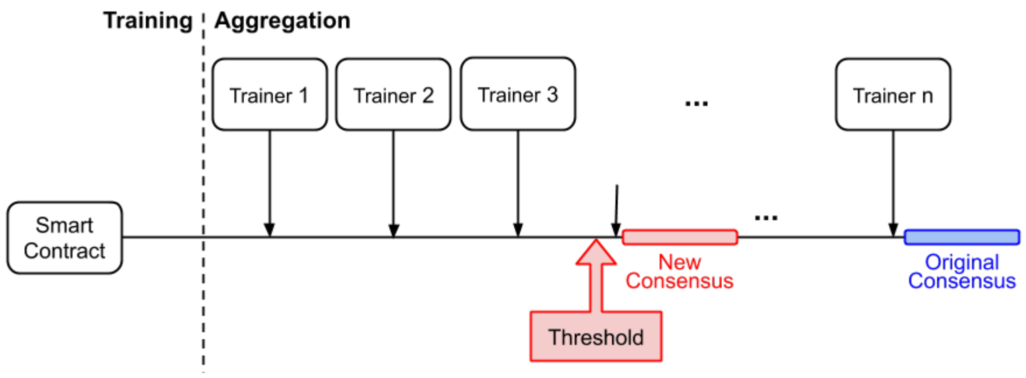

Figure 3: The original consensus must be reached after all trainers' submissions, whereas the new policy starts reaching a consensus right after the number of submissions pass a given threshold.

First, a threshold variable T is added to the configuration of the contract. Before deploying a contract, the administrator can accordingly assign a fraction number to T, for example, $T = 0.3$ in a relatively trustable situation or $T = 0.6$ for meeting a stricter requirement. The threshold can be further adjusted in the process according to the situation.

The second part is the logic condition whether a consensus is reached. Let V_m represent the vote count of the most voted aggregation candidate and N represent the total number of trainers, then passing the threshold can be represented as $V_m \geq N \times T$. However, this formula might not be met at all because of a high threshold or highly corrupted environment. In such a situation, the candidate with the most votes M is chosen. Also, if a tie happens as two or more candidates get equal number of votes, the first candidate is assigned to M. To consider all possible situations, let V represent the number of votes, the logic condition of the new policy can be shown as the pseudo-code below.

```
if (Vm>=N*T or V==N) {result = M}
```

Now, the contract is able to decide the aggregation result as soon as either condition is fulfilled in each aggregation round.

4.1.2 Adaptation of Trainer

The overhead in the aggregation process can be further reduced by adapting the trainers to the new consensus policy. Each submission of an aggregation result from a trainer to the contract is a transaction, which costs gas. In the original solution, once a trainer starts an aggregation process, which is time-consuming, the submission is to be made. This becomes an issue if consensus is reached while the left trainers are still calculating the aggregation locally. These trainers will go through the submission process even though it is doomed.

To address this issue, a state checking is added right before the submission. If the contract is not in the aggregation state, which means the consensus has been reached, then this submission step is skipped and no transaction is made. Depending on the threshold rate, the total transaction costs in the aggregation process can be proportionally reduced.

However, in some certain situations, an aggregation might be submitted exactly when a consensus being reached by other submissions. In this case, the generated transaction will be ignored by the contract and cost only a smaller amount of gas. So even though this kind of transaction cannot be prevented, the overhead is lower thanks to the new consensus policy. Additionally, more detailed log information is generated during the runs to enable the monitoring of the process and cost. The new logs record when and how each submission of aggregation is dealt with by the contract. A visualisation based on the logs of a test run is shown in Figure 4.

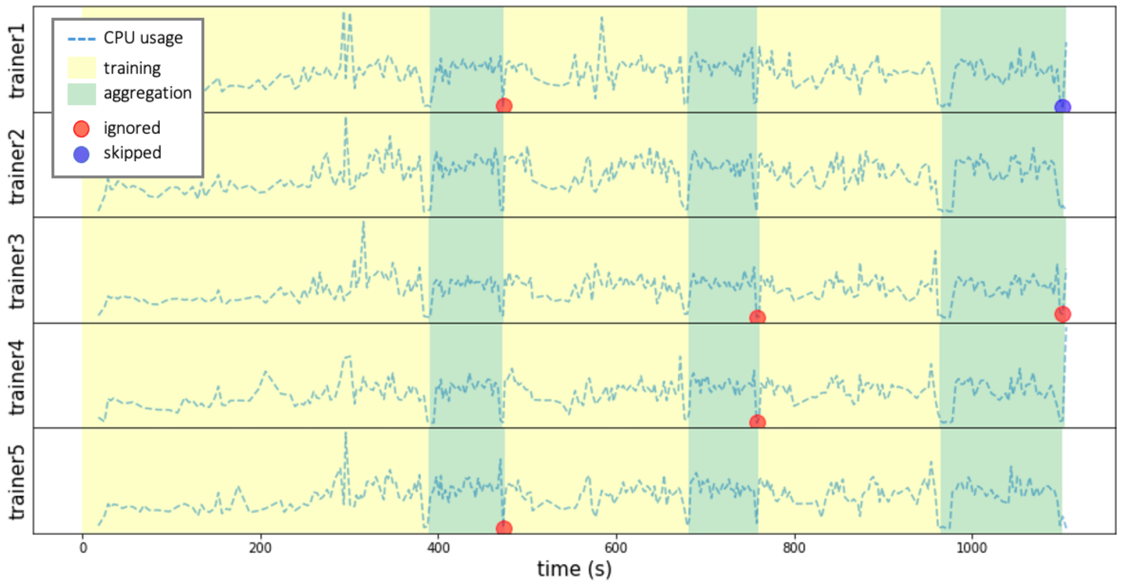

Figure 4: A visualisation of consensus points in an example 3-epoch run with 5 trainers and 0.6 as the consensus threshold. The dashed lines show the trainers' CPU usages during the whole run, the yellow/green backgrounds distinguish the training/aggregation phases while the red/blue dots represent whether the transaction is ignored or skipped due to the new consensus policy.

In this example run, the number of trainers N is set to 5 and the consensus threshold T is equal to 0.6. Horizontally, we can see there are three rounds of training and aggregation, and the consensus is always reached at the end of aggregation phases. According to the passing-threshold condition $V_m \geq N \times T$, the consensus can be reached when $V_m \geq 3.0$. Since no corruption happened during this example run, three submissions could reach a consensus in each round and two submissions were skipped or ignored. So, vertically, we can see from the plot that there are always two dots in each aggregation phase.

We note, this example run was conducted in an IID way. Every trainer had an equal amount of data and the same capability to process, so it is reasonable for all trainers to perform the same action almost simultaneously. So, in Figure 4, most of the dots are red and there is no clear pattern of which trainers' transactions are prone to be saved.

In a non-IID scenario, which is a common real-life set-up, the new consensus policy will be more effective from two aspects. First, the incident of simultaneous submissions across different trainers may seldom happen, so most unnecessary transactions will be skipped instead of being ignored, therefore the gas cost can be further reduced. Secondly, the time consumption of the aggregation step can also be reduced since the slowest trainers are no longer required to be counted after the consensus is reached. Although the security of a real-life situation can be challenging, the reaching of consensus can be secured as long as the threshold value is cautiously conditioned.

4.1.3 Test Cases

To guarantee that the new consensus policy works correctly, several test cases were programmed in the implementation. These cases also provide simple and clear ideas on how the policy is expected to function. Four example cases are given in Table 1, all with a consensus threshold of 60%. Each case mimics a sequence of submissions by a certain number of trainers in one aggregation round where each trainer submits 0 or 1 as its aggregation result. Two assessments are made for each case - whether the consensus is reached at the expected point in time and whether the consensus result is correct.

Table 1: Test cases for the new consensus policy

number of	**submission sequence**				**consensus**	
trainers	1st	2nd	3rd	4th	reached at	result
3	0	0	1	-	2nd	0
3	0	1	0	-	3rd	0
3	0	1	1	-	3rd	1
4	0	1	0	1	4th	0

The first three test cases verify the first condition of the policy, that the consensus must be reached at the submission which brings the votes counting above the threshold, and the result must be the value with the most votes. While the last case verifies the second condition of the policy, that even the threshold can be met, a consensus is to be reached after all trainers make submissions, and the result must be either the dominant one or the first candidate being voted in a tie.

4.2 Secure Voting

Decentralised federated learning is supposed to use the blockchain to remove the need for a trusted authority. However, the chain on its own does not guarantee that the code running on it is written without loopholes. The original code that this project builds upon [5] is so far only used for research and testing, but to use it in a real world scenario the voting process must be rigid. To take a step towards a more rigid voting structure in the aggregation, a branch (secureVoting) with code aimed at improving it has been added in the GitHub repository. Since the code in the branch changes the aggregation voting and weights submission, it does not pass all the test cases. Tests for the new voting logic have not been developed yet but it behaves as expected and only fails the tests that goes against the new voting logic. It also results in the same performance on the MNIST data set as the previous voting mechanism, indicating correctness. We note that results in the other parts of the report have not been produced using this code. Nevertheless, the idea is presented below.

In the original code, the consensus of aggregation submissions was designed to guarantee security/correctness even if a small amount of aggregations are wrong. However, it does not keep track of how many local weight updates (which the aggregation is based upon) a trainer makes or which trainers are voting each time. For example, the contract may receive several submissions from one trainer due to corruption or bad internet condition, etc. Therefore, an update that prevents trainers from submitting multiple local updates (and thus make their updates a bigger part of what will be aggregated into the new global model) or voting for more than one aggregation candidate has been added to the smart contract.

This is done by adding the mapping variable `rightToVote` to the smart contract. It maps each trainer to a `bool` value that is `true` until the trainer makes its first submission and it switches to `false`. All the values are then reset when the contract moves to the news state, as seen in Figure 5. The contract checks the voting right of all trainers before allowing them to use the functions for submitting updates or voting for aggregation and hence fraud is prevented.

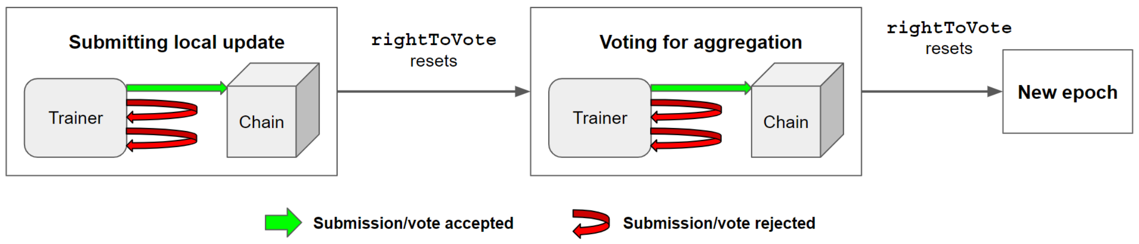

Figure 5: Basic principle of secure voting. The `rightToVote` variable makes sure that each trainer only makes one commission in each step and resets when all local updates are submitted and when the aggregation is finished and the model starts a new global epoch.

The new voting system succeeds to remove some security problems, but still leaves a few open issues. There is still nothing preventing new trainers to be added by the existing trainers at any time. To get more votes for their aggregation a trainer can therefore add new trainers that will vote in their favour. It is a bit trickier, but still doable and a clear security compromise. This could be handled by adding a similar condition as described above to the function that adds trainers. One would then need to decide who can add trainers and when.

Before implementing this in a real life scenario, one should also remember that a single trainer can always be corrupted and that is not something that can be prevented similarly in the contract. Still, the updated voting scheme in the secureVoting branch is a good base to build from if the applications should be launched in a real scenario.

5 Evaluation

In order to evaluate the performance of the enhanced implementation. We decided to look at total gas consumption for different number of trainers, CPU and memory usage of trainers and chain. We have also investigated ways to make the federated learning models converge quicker. All results presented in this section is gathered from the local scenario using 3 global epoch Using the MNIST dataset. All trainers have equal partitions of the data and between each simulation the system was rebuilt from scratch.

All results are obtained by using a local simulation on one device.

5.1 Gas Consumption

With the introduction of a consensus threshold for how many trainers that need to agree on the new model we except the total gas consumption to be reduced. We also expect that the amount of reduction is dependent on the choice of threshold. The total gas consumption consists of all gas consumed by the trainers when they have made transactions to the chain during the training and aggregation phases. In the results we have neglected the transactions that unavoidable, such as deploying the smart contracts and trainers to the chain. In this section we have calculated the total gas consumption for different threshold with different number of trainers. The results are then compared to the original implementation in [5]. The results are gathered on a *MacBook Pro with an 1,4 GHz Intel Core i5* processor with four independent cores.

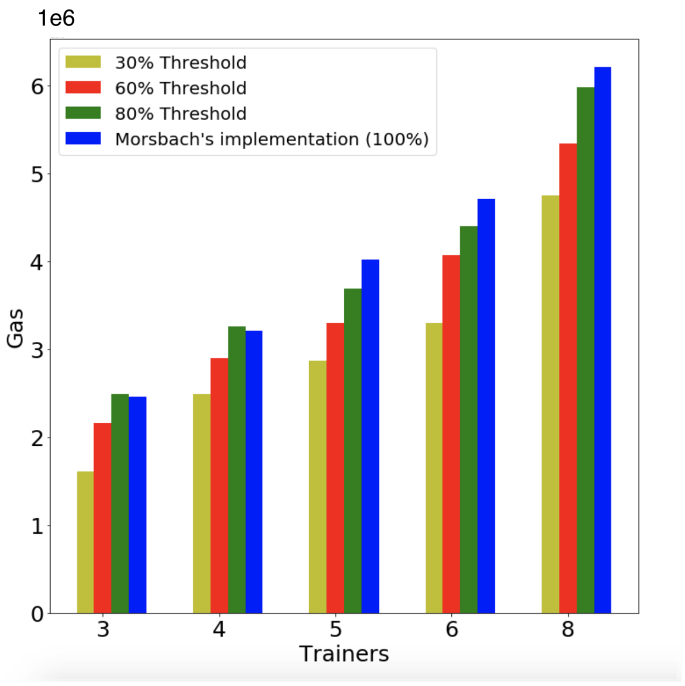

Figure 6: The total gas consumption for simulations using different number of trainers and consensus thresholds using three global epochs.

As expected, when a lower threshold is used the total gas consumption is reduced. Specifically, the gas consumed during the aggregation phase can be proportionally reduced according to the threshold, also affected by the number of skipped and ignored transactions. The reduction also increased when more trainers participate. The total gas consumption is decreased by 12.3% for three trainers with a 60% threshold and 14.3% for eight trainers compared to Morsbach's original implementation. We expect this absolute reduction to grow with the number of trainers. Since the threshold is rounded upwards, a diminishing return exists when the threshold is increased. For example, using an 80% threshold on three trainers will require 2.6, rounded to three trainers to vote for the same new model which is equivalent to a 100% threshold. This behaviour can be seen in Figure 6 and the total gas consumption actually becomes slightly higher for an 80% threshold compared to the original implementation for three and four trainers. This is caused by the smart contract having to do more logical operation in the enhanced implementation compared to the original implementation.

From the results we can conclude that the enhanced implementation is able to reduce the total gas consumption and that it is expected to grow with the number of trainers. However, the results depend on which threshold is used.

5.2 CPU and Memory Usage

The enhanced implementation extends the original implementation in [5] by using a consensus policy with the purpose of making the application more efficient. It is of importance that the extra feature doesn't compromise the functionality of the original implementation. When investigating the CPU and memory usage of the enhanced implementation it's undesired to obtain a significant increase in comparison to the original implementation. In Figure 7 and 8 a comparison between the CPU and memory usage for the chain and trainers respectively is made by executing the application three times using five trainers, three global epochs, a 60% consensus threshold and accumulating the data over time. The results are gathered on a *MacBook Pro with an 3 GHz Intel Core i7* processor with two independent cores.

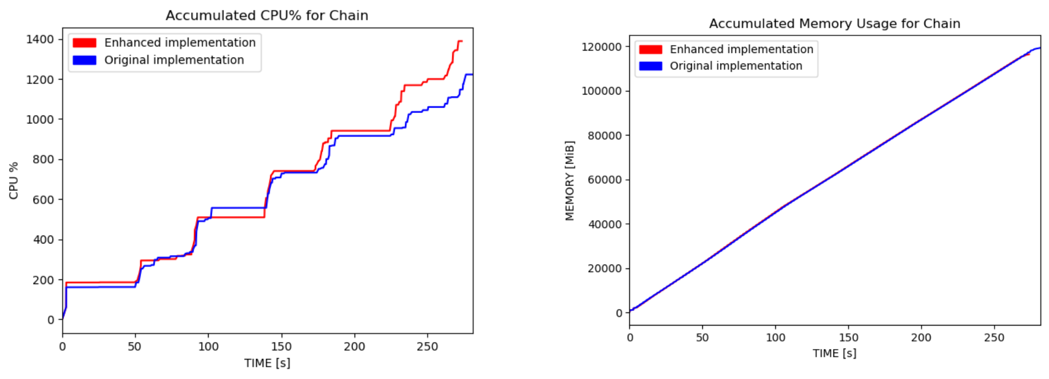

Figure 7: Accumulated CPU and memory usage for the chain during three simulations

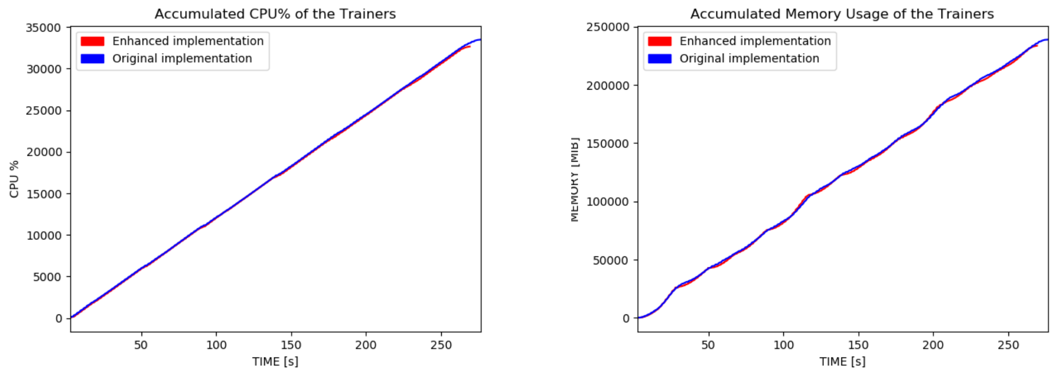

Figure 8: Accumulated CPU and memory usage for five trainers during three simulations

Figure 7 and Figure 8 visualise that over time the CPU usage for the chain, and the CPU and memory usage for the trainers is almost identical for the original and enhanced implementation. In Figure 7 the CPU usage for the chain in the enhanced implementation is slightly higher than in the original implementation. This could be due to the fact that measurements of a smaller load on a CPU can result in inaccurate measurements or that the enhanced implementation is in fact using a bigger load on the CPU. Considering the overall results of the graphs it can be concluded that the enhanced implementation isn't using significantly much more resources than the original implementation and isn't compromising how and on what machine the application can be used. Investigating the accuracy of the machine learning models also suggests that the accuracy isn't affected by the new consensus policy.

5.3 Effect of Local Training

One way that can make federated learning models converge quicker is to let trainers do more training locally, before submitting their updates and calculating the new model. This is a way to reduce the communication costs, which dominate in the federated learning setup, compared to non-federated setups where computational costs dominate, see [3].

Replacing the central server with a decentralised blockchain setup should not affect this general federated learning idea. In this case it would not only decrease the total communication cost, but also the total number of operations needed on the blockchain, hence reducing the transaction cost as well. To test if this could be used to further enhance the implementation, the convergence rate was measured for different numbers of local epochs, as shown in Figure 9.

For the algorithm used in this project, more local training means increasing the amount of epochs that the trainers run locally (changed in the file `utils.py`).

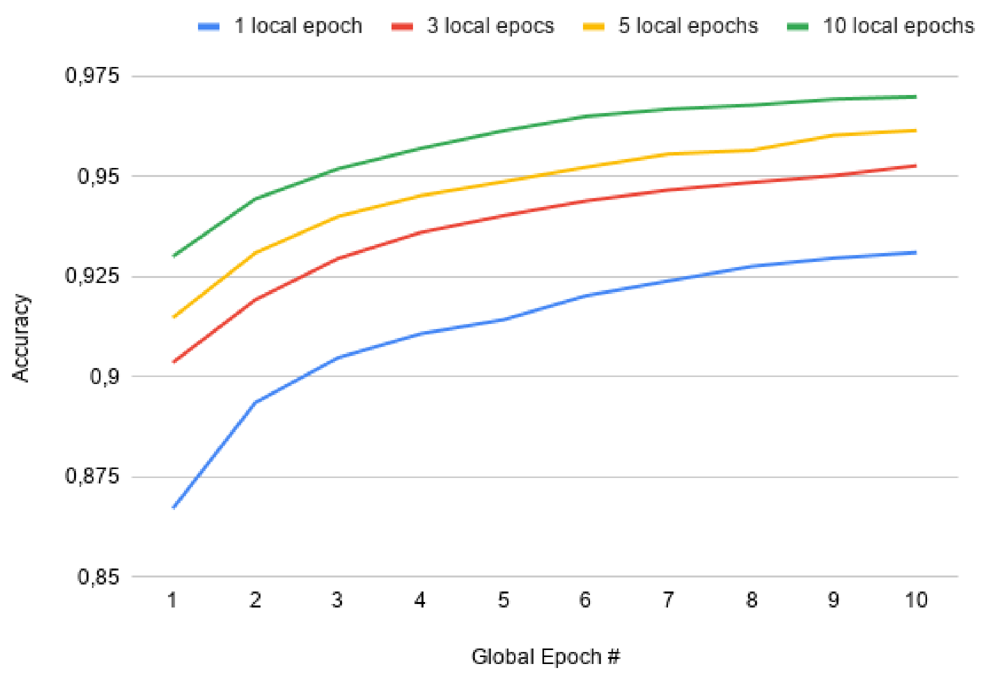

Figure 9: Model's convergence rate variation depending on the amount of training done locally. Training is done on the MNIST data set split up equally, since the data is not sorted and IID data can be assumed.

As Figure 9 indicates, more local training does achieve a higher accuracy quicker for this implementation. However, this behaviour cannot be expected to go on for ever, since the local trainers will overfit (train so much that it improves its results on the training data at the expense of worse performance on unseen data) their models to their data, which happens at around 20 local epochs in [3]. Something similar should be expected here, since decentralising federated learning does not change the machine learning concepts of it. It is also worth noticing that this experiment is done with the data equally split between the trainers and that the MNIST data set is not sorted. This gives an IID data setup which might not be realistic for most real life applications. The good convergence result is also affected by the fact that all trainers are initialised with the same weights and should hence take optimisation steps towards the same local minima. That might not be the case for different initialisation on a non-convex problem, where there are multiple local minima and the trainers could find different ones when optimising. In that case, the average of their weights could end up far from any local minima, giving a very bad model.

6 Conclusion

When using privacy sensitive data or seeking to reduce the communication delay in a machine learning application used by several trainers, federated learning can be a good option. However, it leaves a single point of failure which in [5] is solved by using blockchain to decentralise it even further.

The enhanced implementation we propose successfully introduces a voting system based on a consensus threshold. Through testing, it is verified to give the same, correct federated learning model. The threshold results in reduced total gas consumption, proportional to how low the threshold is set. Since bigger networks use more gas in total, the amount of saved gas also increases with the number of trainers. In a large decentralised federated learning scenario, this would correspond to a lot of money being saved in ETH-transaction costs. For the case with eight workers and a 60% threshold (as in 5.1) where the total gas consumption is reduced with around 14%, this would save about 900 SEK (using the gas price and ETH exchange rate of 2021-01-22) [19] [20]. For the CPU and memory usage the enhanced implementation demands almost equally as much computer resources as the original implementation.

6.1 Further work

The current implementation uses a simple averaging scheme when the new model is aggregated. Every trainer shares their weights with everyone and the averaging is done by everyone locally. This policy gives high trust but comes with high overhead, since all trainers make the exact same calculation. Another more efficient approach would be to implement an incremental federated averaging scheme [21] where the average is calculated in steps. This could decrease the communication costs and the number of operations on the blockchain, but the challenge is to do it without the need to trust each individual trainer. There is also potential for a gossip based model, like GossipGraD [22], where trainers split their weights into segments and send the segments to a few other random trainers. In this way, the whole network will benefit from all the data after a while. However, the structure for model updating will have to change significantly compared to the current implementation.

7 Appendix

All code can be found in this GitHub-repository: `https://github.com/fredrikorn/DecFL`

8 References

References

[1] Jiang Weiwen, Jinjun Xiong, and Yiyu Shi. “When Machine Learning Meets Quantum Computers: A Case Study.” ArXiv:2012.10360 [Quant-Ph], December 18, 2020, 1. https://doi.org/10.1145/3394885.3431629.

[2] Zou Hongbo, Guangjing Chen, Pengtao Xie, Sean Chen, Yongtian He, Hochih Huang, Zheng Nie, et al. “Validate and Enable Machine Learning in Industrial AI.” ArXiv:2012.09610 [Cs], October 30, 2020, 1.

[3] McMahan, H. Brendan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. “Communication-Efficient Learning of Deep Networks from Decentralized Data.” ArXiv:1602.05629 [Cs], February 28, 2017. http://arxiv.org/abs/1602.05629.

[4] Jun Li, Yumeng Shao, Ming Ding, Chuan Ma, Kang Wei, Zhu Han, and H. Vincent Poor. “Blockchain Assisted Decentralized Federated Learning (BLADE-FL) with Lazy Clients.” ArXiv:2012.02044 [Cs], December 2, 2020, 1–2.

[5] Felix Johannes, Morsbach. Hardened Model Aggregation for Federated Learning Backed by Distributed Trust Towards Decentralizing Federated Learning Using a Blockchain, 2020. http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-423621.

[6] ethereum.org. “Ethereum Whitepaper.” Accessed December 22, 2020. https://ethereum.org.

[7] Nakamoto, Satoshi. “Bitcoin: A Peer-to-Peer Electronic Cash System.” Cryptography Mailing List at Https://Metzdowd.Com, March 24, 2009.

[8] Dang, Quynh H. “Secure Hash Standard.” NIST, August 4, 2015. https://www.nist.gov/publications/secure-hash-standard.

[9] Buterin, V. “Ethereum Whitepaper”, 2014. ethereum.org. Accessed December 17, 2020. https://ethereum.org.

[10] ethereum.org. “Proof-of-Stake (PoS).” Accessed February 8, 2021. https://ethereum.org.

[11] Raval, Siraj. Decentralized Applications: Harnessing Bitcoin’s Blockchain Technology. O’Reilly Media, Inc., 2016.

[12] Benet, Juan. “IPFS - Content Addressed, Versioned, P2P File System.” ArXiv:1407.3561 [Cs], July 14, 2014. http://arxiv.org/abs/1407.3561.

[13] N. Nizamuddin, K. Salah, M. Ajmal Azad, J. Arshad, and M.H. Rehman. Decentralized document version control using ethereum block- chain and IPFS. Computers Electrical Engineering, 76:183–197, June 2019. ISSN 00457906. doi: 10.1016/j.compeleceng.2019.03.014.

[14] Bhargavan, Karthikeyan, Antoine Delignat-Lavaud, Cedric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Beguelin ´. “Short Paper: Formal Verification of Smart Contracts,” 2016. http://www.cs.umd.edu/ aseem/solidetherplas.pdf.

[15] Ethereum/Go-Ethereum. Go. 2013. Reprint, ethereum, 2021. https://github.com/ethereum/go-ethereum.

[16] Trufflesuite/Ganache-Cli. JavaScript. 2016. Reprint, Truffle Suite, 2021. https://github.com/trufflesuite/ganache-cli.

[17] Redis/Redis. C. 2009. Reprint, Redis, 2021. https://github.com/redis/redis.

[18] “Introduction — Web3.Py 5.15.0 Documentation.” Accessed January 26, 2021. https://web3py.readthedocs.io/en/stable/.

[19] “Ethereum Average Gas Price.” Accessed January 25, 2021. https://ycharts.com/indicators/ethereum_average_gas_price.

[20] “Ethereum Price Chart (ETH) — Coinbase.” Accessed January 25, 2021. https://www.coinbase.com/price/ethereum.

[21] The Ubuntu Incident. “Calculating the Average Incrementally,” April 25, 2012. https://ubuntuincident.wordpress.com/2012/04/25/calculating-the-average-incrementally/.

[22] Daily, Jeff, Abhinav Vishnu, Charles Siegel, Thomas Warfel, and Vinay Amatya. “GossipGraD: Scalable Deep Learning Using Gossip Communication Based Asynchronous Gradient Descent.” ArXiv:1803.05880 [Cs], March 15, 2018. http://arxiv.org/abs/1803.05880.