Proof of Gradient Descent: new approach for distributed machine learning implemented in blockchain.

Eric Anderson

Email: ericsoncrypto87@gmail.com

Bitcoin mining is a power-consuming and resource-intensive process. To add a block of transactions to the blockchain, miners spend a significant amount of energy. The Bitcoin protocol, which is named 'proof of work' (PoW), resembles a lottery and the underlying computational work is not useful otherwise. In this paper, we describe a new type of 'proof of useful work' – proof of Gradient Descent (PoGD) protocol. PoGD is based on training a machine learning model on the blockchain. In this artificially created but stable ecosystem miners get a chance to create new Tensor coins after performing honest and useful ML training work. Clients submit tasks and pay to all contributors of training. This is an extra stimulus to participate in the network because the system does not rely only on the lottery procedure anymore. Using our consensus protocol, interested parties can order, complete, and verify useful work in a distributed environment. We outline mechanisms to reward useful work and punish malicious actors. We aim to build better AI systems using the security and the transparency of the blockchain.

Keywords: Machine learning, blockchain, proof of useful work, mining, Tensor coin, project Tensor, artificial intelligence.

1. INTRODUCTION

Bitcoin [1] miners are spending an enormous amount of electricity to add a new block to the blockchain. A blockchain is a sequence of blocks, where each block contains some amount of transactions. Except other fields, the header of a block contains a cryptographic hash of the previous block, a target value, and a nonce. In the classical proof of work consensus (PoW) protocol, to add a new block the double SHA-256 hash of the block's header must match the target value which is known to all network participants. The nonce is varied to obtain different hash values. For each successfully mined block, a miner is allowed to create out of nothing a specific amount of new coins (the block subsidy) and also to collect the fees for transactions included in the block. Sometimes new blocks are created on top of the same previous block, which can cause chain splitting. According to the consensus rule, miners should build on the longest chain to discard shorter forks. So, as we see, PoW is easy to verify, but hard to produce[2]. Repeated hashing (by varying the nonce) is not useful. PoW is a pure lottery and does not guarantee fair payment for all participants that spend their computational resources. We would like to reward only productive nodes. To do so, PoW should be combined with beneficial work (machine learning) and miners should compete to provide a proof of useful work (PoUW) – proof of gradient descent (PoGD).

There are several obstacles in designing a PoUW system using machine learning (ML) as the basis for useful work. If we will make a comparison between tasks of ML and traditional hashing, we will see that ML tasks are a lot more complex and diverse. The Bitcoin puzzle [2] grows in complexity over time, but in our scenario, the client's ML problem dictates the complexity. So, due to the heterogeneity of ML tasks, it is difficult to verify if an actor performed honest work. There are many steps in the ML algorithms where low-performing actors can avoid work, perform cheap work, or behave maliciously. Despite of this problem, there is probably the greatest problem of this concept - blockchain is not designed to hold the large amounts of data which is needed for ML training, and most of this data is not interesting for the majority of actors. It is also hard to distribute and coordinate a ML training process in a trustless environment as the blockchain's peer-to-peer (P2P) network.

Some of the commercial machine learning platforms work with Big Data that requires distribution of workloads and orchestration across multiple nodes. These systems use data parallelism: each node holds an internal replica of the trained ML model, while data is divided across worker nodes. The blockchain offers access to more computational resources than a standard cloud ML service.

Our research was motivated by the questions: *Can*

we build better AI systems using blockchain's security?Can we provide a better blockchain protocol based on PoGD?

We used infrastructure of the blockchain to coordinate the training of a deep neural network (DNN). We designed a decentralised network with a consensus protocol to perform and verify useful work based on ML. At the core of the protocol lies an original way to create nonces derived from useful work.

Our paper is organised in the following way: In Paragraph 2 we briefly present related efforts to combine artificial intelligence with the blockchain. We provide an overview of our environment and the protocol in Paragraph 3. We show the detailed steps of the ML training in Paragraph 4. In the section about the proof of useful work (Paragraph 5), we describe the mining and the verification processes. We are also working on a proof of concept (PoC) whose details are given in Paragraph 6. In Paragraph 7, we discuss potential performance and security concerns related to our proposal. We provide conclusions and details about future work in Paragraph 8.

2. RELATED WORK

A series of different PoUW protocols have been proposed, all of them with limited practicality: Ball et al. [2] proposed methods for solving the *orthogonal vectors*, *all-pairs shortest path* and *3SUM* problems on blockchain; in PrimeCoin miners search for Cunningham chains of prime numbers [3].

DML [4] and SingularityNET [5] are marketplaces for distributed AI services utilizing smart contracts. SingularityNET requires curation of services. None of them have a tight integration with the blockchain, keeping AI as a separate service.

Gridcoin [6] is an open-source proof-of-stake (PoS) blockchain for solving tasks on Berkeley Open Infrastructure for Network Computing and their system is centralised (whitelists, central server, centralised verification).

Coin.AI [7] is just a theoretical proposal where miners train separately a DNN. The architecture of the deep neural network is generated based on the last mined block and the verification involves checking the model's performance. Their system is prone to security risks, e.g. miners doing cheap work or no work.

CrowdBC [8] is a crowdsourcing framework using Ethereum smart contracts. Participants deposit a certain monetary amount to mitigate for potential bad behaviour, but miners do not perform useful work.

According to our knowledge, our PoUW proposal is the only protocol that tightly integrates with the blockchain. It does not require smart contracts. The miners perform useful computational work that can be deterministically verified. Our PoUW facilitates better AI through decentralisation, enhanced security and the right incentives.

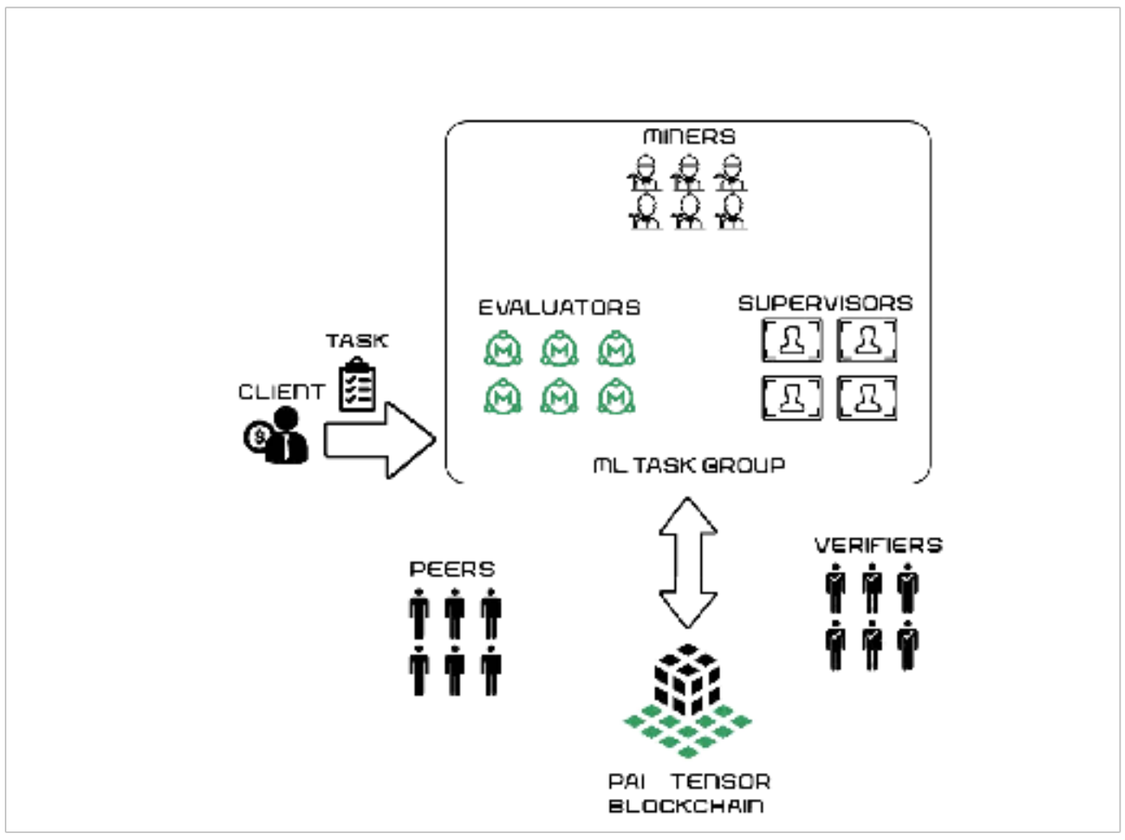

FIGURE 1: **Environment overview.** The client submits a ML task to the Tensor network. Worker nodes perform the training and evaluators decide how to pay. The Tensor blockchain ensures the security of the ML process.

3. SYSTEM OVERVIEW

3.1. Environment

The environment is the Tensor (Personalised Artificial Intelligence) blockchain, a hybrid Proof of Work/Proof of Stake (PoW/PoS) blockchain. It is a P2P decentralised network (1) composed of:

Clients: Nodes that pay to train their models on the Tensor blockchain.

Miners: Nodes that perform the training. They can mine a new block with special nonces obtained after each iteration. The training is distributed and all miners collaborate by sharing updates about their local model.

Supervisors: Actors that record all messages during a task in a log called 'message history'. They also guard against malicious behaviour during the training because the environment may also contain Byzantine nodes.

Evaluators: Independent nodes that test the final models from each miner and send the best one to the client. They also split the client's fee and pay all nodes accordingly.

Verifiers: Nodes that verify if a block is valid. We need them because verification is computationally expensive and it is not carried out by all nodes.

Peers: Nodes that do not have any of the aforementioned roles. They are using regular blockchain transactions.

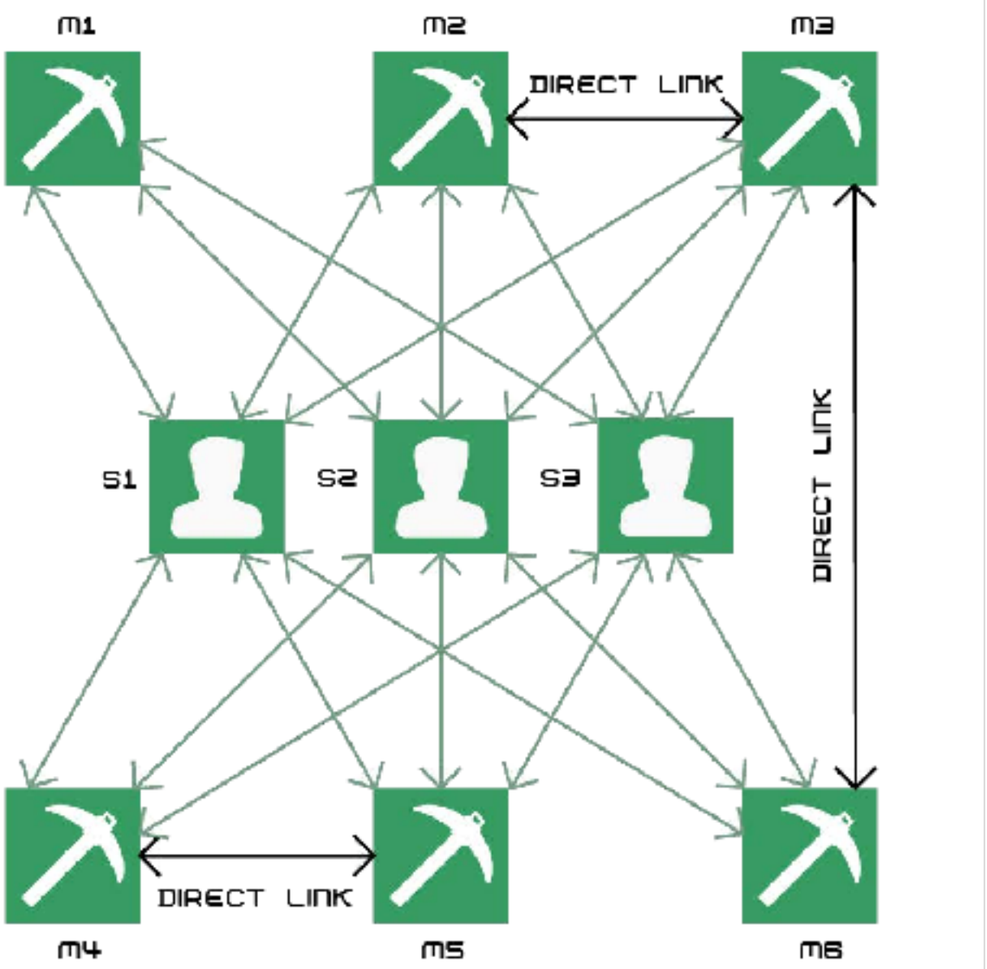

FIGURE 2: **A communication topology example.** Miners (M1-M6) are fully connected with the supervisors (S1-S3) and send/query data to/from them. A few miners established direct links in-between them (e.g. M2-M3, M4-M5, M3-M6).

Miners and supervisors are called *worker nodes* because they actively participate in the training. Worker nodes communicate by using fast message channels. To facilitate direct communication, we recommend running all worker nodes in a *virtual private network* (VPN) with a full VPN mesh topology (e.g. PeerVPN – `https://peervpn.net`). Miners send messages to supervisors that record the message history. Directly sending messages between miners is optional (see Fig. 2).

3.2. Transactions

A Bitcoin transaction is a digitally signed data structure that contains versioning, inputs and outputs. A transaction is valid if its inputs refer to other unspent outputs (UTXOs). **OP_RETURN** is a script operation field (80 bytes in length in Bitcoin) that marks a transaction output invalid, but can be used to store arbitrary data in the blockchain.

In the Tensor blockchain, we use special transactions to handle the training, verification, evaluation and payment of the ML tasks. To do so, we encode all the required extra information in the OP_RETURN code (160 bytes in our protocol) of one of the transaction's outputs [3].

Before inclusion in blocks, transactions wait in the *mempool*, a buffer of pending transactions. Nodes relay transactions from the mempool to each other and eventually they propagate across the whole P2P network.

Off-chain messages and own transactions are signed using the participant's private ECDSA key. When we need multi-party transactions we use *Boneh-Lynn-Shacham* (BLS) keys and signatures ([9]), for example when supervisors must agree on a common decision. To reduce the number of special transactions, we require that a *leader* posts a multi-party transaction.

3.3. Staking

Except regular peers, all nodes must first deposit coins as a collateral. We call this process '*staking*'. The deposits represent locked money that are later returned along with extra fees if the participants finish properly their work.

Staking means buying *tickets*. Nodes can issue *BUY_TICKETS* transactions, which are special transactions containing the desired role (miner, supervisor etc) and preferences for specific tasks (encoded in the OP_RETURN field). Tickets become 'live' after they are included in the blockchain and buried under a preset number of blocks (e.g. 128). There are around 40960 tickets in the *mempool* and their number is kept constant by updating ticket prices after every 144 mined blocks. We use Decred's algorithm ([10]) to adjust prices. No more than 50 tickets can be included in a mined block.

Another type of stake transaction is *PAY_FOR_TASK*, which is issued by clients when they submit a new task. It includes a task description and a training fee.

Staking is a way to ensure against malicious behaviour because the stakes of dishonest actors are captured and re-distributed to fair players. At the end of training, the evaluators decide how to split the client's fee and punish the bad actors. Evaluators issue a *CHARGE_FOR_TASK* transaction to pay the honest participants. The client's fee is returned if the task could not be executed. The stakes of honest worker nodes are also returned if the task was compromised by malicious actors. Each stake has an expiration time after which, if the holder has not been assigned to do work, the amount is fully returned. Offline nodes that are selected to perform work lose a part of their stake (e.g. 10%).

3.4. Tasks

A client submits a task definition T and a fee F as a special transaction to the blockchain (*PAY_FOR_TASK*), containing:

- A description of the trained model: the model type (e.g.: multi-layer perceptron, convolutional neural network) and its architecture: the layers

[3] We provided an example implementation for special transactions in the Supplementary material in Appendix A, and in Appendix B we detailed a method to shorten the task wait time.

(e.g. dense, dropout, convolutional, batch normalisation, pooling) with the number of neurons, the activation functions (e.g. *relu*, *sigmoid*, *tanh*), the loss function (e.g. softmax cross-entropy loss).

- The optimiser (e.g. SGD, Adam) and its parameters (e.g. the initialiser, the learning rate).
- The stopping criterion (early-stopping, preset no. of epochs etc)
- The validation strategy (e.g. cross-fold validation, holdout) and the percentage of data from the training dataset set apart for validation. – $D_{pct}^{(val)}$.
- Metrics of interest K (e.g. accuracy, loss); the client will pay the best model based on these metrics on the test dataset.
- Dataset information: format (e.g. CSV, MNIST [11]), mini-batch size, training dataset percent out of the whole dataset ($D_{pct}^{(tr')}$), the hashes of at least 10 equally divided data batches from the original dataset D and the size of the whole dataset D.
- Performance: expected training time (t_{exp}), hardware preferences (e.g. CPU vs GPU).

The fee F is split between honest worker nodes and evaluators following a reward scheme R: a part is received by miners based on performance, a part by supervisors and another part by evaluators. All nodes are incentivised to participate in the Tensor blockchain: clients receive a trained model, miners can receive block rewards and a fraction of the client's fee, supervisors and evaluators also receive a part of the client's fee. Miners compensate verifiers with a share of the block's reward.

3.5. Protocol

In our system, a client utilises the Tensor blockchain to train a ML model and to pay for this service. After the client broadcasts the task to the Tensor network, the miners and the supervisors are matched randomly by the network based on the worker nodes' preferences P vs the task definition T.

The dataset D is first split into:

- A training dataset, further revealed to the miners to perform ML work on it.
- A validation dataset selected from the initial training dataset for ML validation.
- A test dataset revealed to the evaluators when the final model should be tested.

The training dataset is further split into equally-sized mini-batches. Typical sizes for a mini-batch range from 10 to 1000 records. A mini-batch is processed in each iteration. An epoch is a full training cycle, i.e. all mini-batches from the training dataset are processed.

Once training is started, miners iteratively improve their local models based on their work on mini-batches and based on messages they receive from fellow miners. Every miner shares the modifications to his/her model with other task participants. They also mine with several nonces obtained after each iteration.

The supervisors record all the messages during a task and look after malicious behaviours. Evaluators test the final models, select the best model for the client and distribute the client's fee.

Miners build blocks as in the Bitcoin protocol, but with the addition of useful work, i.e. honestly executing an iteration of the ML task. A miner scans the mempool, collects transactions, creates the block and adds additional information (extra fields) for the proof of useful work. In our PoUW, the nonce is a SHA-256 hash occupying 32 bytes.

To validate a block, verifiers will receive the input data, re-run the lucky iteration and check if the outputs are reproducible. A miner will send to verifiers all data and context necessary for the *proof of useful work*.

4. WORKFLOW

There are four ML task workflow stages: registration, initialisation, training and finalisation.

4.1. Registration

In the registration phase, the client submits a PAY_FOR_TASK transaction. This transaction is included in a block called the *task definition block*. Clients can revoke a task if it has not started yet by sending a REVOKE_TASK transaction referencing the initial submission. The corresponding PAY_FOR_TASK transaction is thus annulled and the stake released. In case they want to adjust their bids, clients may also re-post their tasks before they start or expire. After updating, a task's maturity time is reinitialized.

At least 2 miners and 3 supervisors are required to start a task. The number of supervisors can be in the interval $|S| \in [3, \max(3, sqrt|M|)]$, while the number of miners can be in the interval $[2, \omega_D * disk\_size(D) + \omega_t * t_{exp} + \omega_F * F]$, where $disk\_size$ is a function that returns the size of the dataset on the disk (in kB), $|S|$ is the number of the supervisors, $|M|$ is the number of miners, while ω_D, ω_t and ω_F are network-wide coefficients. A task definition block not satisfying these requirements is deemed invalid.

A worker node calculates if any of its live tickets is selected for participation in training. A ticket bought after task submission cannot participate in the training. For each ticket, the procedure is as follows:

1. calculate the cosine similarity between a ticket's preferences P and the corresponding maximal subset of possible task preferences, $P_{max} \subseteq T$ (P

and P_{max} are encoded as vectors with non-negative values): $s(P, P_{max}) = \frac{\sum_{i=1}^{n} P_i P_{max\,i}}{\sqrt{\sum_{i=1}^{n} P_i^2}\sqrt{\sum_{i=1}^{n} {P_{max}}_i^2}}$; $s(P, P_{max}) \in [0, 1]$

2. using a verifiable random function (VRF) [12], using the hash of the task definition block ($hash_{TDB}$) and the role on the ticket (supervisor, miner, etc) as inputs, produce a hash and a proof: $(hash, proof) = VRF_{sk}(hash_{TDB}||role)$ (see also [13]). Note: *The hash looks random, but it is dependent on a secret key, sk, and the input string. Knowing the public key, pk, corresponding to sk and the proof, one can verify if the hash was generated correctly.*

3. if $s(P, P_{max}) \geq \omega_S \frac{hash}{2^{hashlen-1}}$, the ticket is selected to participate in task T. ω_S is a network-wide parameter, *hashlen* is the bit-length of the hash.

Selected worker nodes will send special transactions named *applications* (*JOIN_TASK* transactions) that will be included in a subsequent block, called the *participation block*. We seed the random number generator with the task definition block's hash to reproduce/verify the selection process.

4.2. Initialisation

4.2.1. Key exchange and generation

Worker nodes exchange their public keys to verify the received messages and the special transactions.

Supervisors use *t-of-n threshold BLS signatures* to reach consensus, where $n = |S|$ and t represents a 2/3 threshold ($t = \frac{2}{3}n$). Any subset of t up to n supervisors are able to sign a transaction and make it valid, but less than t signers would render the transaction invalid. Supervisors run a modified version of the Joint-Feldman distributed key generation (DKG) protocol ([14]) to collectively generate their private BLS keys (see Supplementary material, Appendix C and Appendix D). Due to the BLS threshold signature properties, for any multi-party transaction tx, as soon as any t signature shares are collected, a leader can reconstitute the global signature on the transaction and verify it as if the global private key (which is unknown) had been used for signing. At the end of the DKG protocol, all supervisors must post *DKG_SUCCESSFUL* transactions to the blockchain signed with their ECDSA secret keys containing their locally calculated t-of-n public key. If all supervisors post transactions with the same public key during a predefined time window, then the protocol is successful and the parties should proceed to the next phase. Otherwise, faulty supervisors are replaced, their stakes confiscated and the DKG protocol is restarted.

4.2.2. Data preparation

Before submitting a task, the client privately splits the dataset D into $p \geq 10$ consecutive fragments $(d_1, d_2..d_p)$, such that $|d_1| = |d_2| = .. = |d_{p-1}|$; and $|d_p| = |D| - (p-1)|d_1| < |d_1|$, then hashes the fragments, gets $H(d_1), H(d_2)..H(d_p)$ and appends the hashes to the dataset section in the task definition. Hashes are public, but D is known only to the client.

During the initialisation phase, the client and the worker nodes use the task definition block hash as a random seed to permute the hashes from D. The first $D_{pct}^{(tr')}$ % of hashes will correspond to the initial training dataset and the client will reveal it to the worker nodes, while the rest will correspond to the test dataset, which is kept secret until the ML task is finished.

The *validation dataset* is independently and deterministically derived by the miners from the initial training dataset based on the validation strategy. For example, in the case of *holdout*, the validation dataset $D^{(val)}$ contains the last $D_{pct}^{(val)}$ % mini-batches of the initial training dataset $D^{(tr')}$. The final training dataset is obtained by subtracting the validation dataset from the initial training dataset: $D^{(tr)} = D^{(tr')} \setminus D^{(val)}$.

The final training dataset is further split into m mini-batches $b_1...b_m; |b_1| = |b_2| = .. = |b_{m-1}|$ and $|b_m| = |D^{(tr)}| - (m-1)|b_1| \leq |b_1|$ (size specified by the client in the task preferences). We use one of the following two methods to assign batches to miners:

Consistent hashing with bounded loads , a method inspired by [15] and [16], in which one computes the hashes of mini-batches con-catenated with the current epoch number ξ: $H(b_1||\xi)..H(b_n||\xi)$. Then, maps every hash to a *consistent hash ring* of size 2^κ using the modulo function: $H \bmod 2^\kappa$, where κ is the exponent of the mapping space (chosen network-wide, e.g. $\kappa = 10$). Each miner should process the mini-batches with the hashes between his/her hash and his/her successor's hash on the hash ring. To prevent discrepancies between the amount of assigned work, we never let a miner process more than $cm/|M|$ mini-batches, where $1 \leq c \leq 2$ is a public constant. We do so by assigning the cur-rent mini-batch to the next miner until the above condition is met.

Interleaved parts , a procedure in which miners and mini-batches are sorted lexicographically by their IDs ($M = \{M_1..M_n\}$) and their hashes, respectively. The allocation procedure specifies that the first miner gets the first mini-batch, the second one gets the second and so on, until all $|M|$ miners are assigned to the first $|M|$ mini-batches. Then, the first miner receives the $|M|+1$-th mini-batch and so on, until all m mini-batches are assigned. For each epoch ξ, a miner M_i is first assigned to mini-batch $b_{i+\xi-1}$, then the assignment continues with $b_{i+\xi-1+|M|}$, $b_{i+\xi-1+2|M|}$... in a rotating manner to also include the first initially uncovered positions $[1..i+\xi]$ (the mini-batches are

seen as a circular data structure).

If a miner quits or a new one joins a task, the above assignment is re-evaluated. Using a random number generator seed based on the hash of the task definition block, we ensure that the miners know in advance which mini-batches should be processed and the order. The steps can also be reproduced during the verification.

4.2.3. Data storage

Supervisors store the data during the ML training. They can use one of the following redundancy schemes:

Full replicas Every supervisor keeps a copy of $D^{(tr')}$. This has a high redundancy factor ([17], p. 36]) equal to the number of supervisors $\beta = \frac{|data_{red}|}{|data|} = |S|$, and a miner can selectively download mini-batches.

Reed-Solomon The supervisors store $D^{(tr')}$ using a $(k+h)$ *Reed-Solomon storage scheme* ([18]). They split the training dataset into $k+h$ equal fragments, where the total number of the supervisors is roughly $|S| \approx k + h$ and $k \approx [1/3..2/3] * |S|$. To get the full dataset, a miner should download data from k supervisors and re-construct it. The redundancy is low $\beta = \frac{k+h}{k} = [1.5..3]$, but miners perform extra computation to re-create the dataset.

4.3. Training

Miners start working on a ML task by initialising the weights and biases ($\boldsymbol{\theta}$) of their local models. At each iteration they fetch a mini-batch, perform *stochastic gradient descent* (SGD) on it ([19, p. 147]), then communicate what they changed in $\boldsymbol{\theta}$ (weight updates) to the other worker nodes. Cost functions in most ML algorithms are sums of loss functions L over training records and their global gradient is an expectation. To minimise the global loss, instead of using all records, one could sample a mini-batch b_i and calculate a partial gradient $\mathbf{g^{(i)}}$ that approximates the global gradient: $\mathbf{g^{(i)}} = \frac{1}{|b_i|} \nabla \sum_{j=1}^{|b_i|} L(x^{(j)}, y^{(j)}, \theta)$, where x and y represent the features and targets per record. Using this gradient, the classical SGD algorithm updates the model as follows: $\theta \leftarrow \theta - \epsilon \mathbf{g^{(i)}}$, where ϵ is the learning rate.

The size of a gradient is equal to the size of the local model, so it is impractical for a miner to send all the gradient updates to the network. To overcome this issue, we use the "dead-reckoning" scheme from [4], in which we update only weights that under/over-flow a certain τ value and communicate the coordinates of these updates in the local model. Gradients under/over τ accumulate and are applied later. Learning is not negatively impacted by this delay.

An iteration consists of the steps described in **Algorithm 1**, adapted from [4]. See Table 2 for notation.

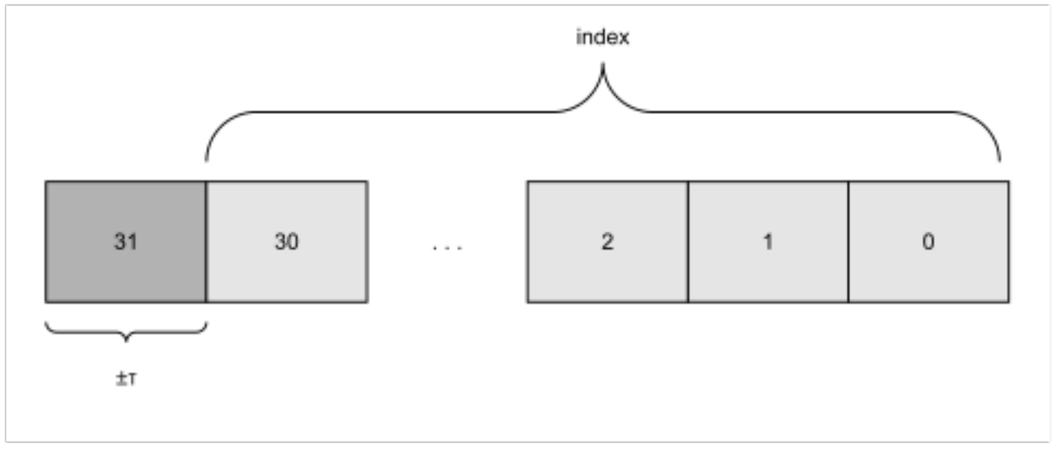

FIGURE 3: **Message map.** Leftmost bit is for τ, the rest is for the index.

- The miner fetches a mini-batch (b_i) according to a method from subsection 4.2.2.
- Updates the local model with weight updates received from peers.
- Derives local gradients using backpropagation and adds them to a residual gradient.
- Prepares a message map, a list $\boldsymbol{\delta^{(\ell)}}$ containing the addresses (indices) of the gradients whose values exceed a threshold value $\pm\tau$ in the residual gradient, where τ is given by the client. An element $e(0|1, i)$ of the list has 4 bytes. The leftmost bit is 0 when the gradient residual at index i is less than $-\tau$ and 1 when it is greater than $+\tau$). The other 31 bits are for the index i (unsigned integer from 0 to $2^{31} - 1$) (see Fig. 3).
- Applies the weight updates from the message map to the local replica of the DNN.
- Receives and uncompresses peers' messages.
- Communicates to the network a message containing: version of current implementation, ML task ID, message type (IT_RES), epoch – ξ, number of peer updates, message map of weight updates – $\delta^{(\ell)}$, values of metrics – $K^{(tr)}$, start and finish times – $t^{(s)}$, $t^{(f)}$, hash of mini-batch, hash of the list with peer messages applied at current iteration, hash of initial model state θ, hash of initial gradient residual, hash of zero-nonce block intended to be mined k iterations in the future – $hash(ZNB)$, hash of list with uncompressed peer messages to be used in the next iteration (received in step 20) and the DER-encoded ECDSA signature of the miner (see Table 1).
- Mines with several nonces.

If a miner finishes earlier the work for an epoch, then he/she will wait for the majority of his/her peers to also complete and will apply their peer updates *ad interim*.

For each epoch ξ, a leader supervisor is chosen in a round-robin fashion using the formula: $(i - 1)$

Algorithm 1 Work done during an iteration

procedure ITERATION()
  $\mathbf{X}^{(\mathbf{tr})}, \mathbf{y}^{(\mathbf{tr})} \leftarrow Load(b_i)$ ▷ Load a mini-batch
  for each $\boldsymbol{\Delta}_i^{(p)} \in \boldsymbol{\Delta}^{(p)}$ **do**
    $\boldsymbol{\theta'} \leftarrow \boldsymbol{\theta} - \epsilon\boldsymbol{\Delta}_i^{(p)}$ ▷ Update local replica with peer values
  $\boldsymbol{g}^{(\ell)} \leftarrow Backprop()$ ▷ Compute local gradients using backpropagation
  $\mathbf{g}^{(\mathbf{r})'} \leftarrow \mathbf{g}^{(\mathbf{r})} + \boldsymbol{g}^{(\ell)}$ ▷ Update gradient residual
  $\mathbf{g}^{(\mathbf{r})''} \leftarrow \mathbf{g}^{(\mathbf{r})'}; \boldsymbol{\Delta}^{(\ell)} \leftarrow \mathbf{0}_{\boldsymbol{n}}; \boldsymbol{\delta}^{(\ell)} \leftarrow$ ▷ Initialise final residual gradient, local weights list and message map (n – length of the gradient)
  for each $g_i^{(r)'} \in \mathbf{g}^{(\mathbf{r})'}$ **do**
    if $g_i^{(r)'} > +\tau$ **then**
      $\boldsymbol{\delta}^{(\ell)} \leftarrow \boldsymbol{\delta}^{(\ell)} \cup e(\mathbf{1}, i)$
      $\Delta_i^{(\ell)} \leftarrow +\tau$
      $g_i^{(r)''} \leftarrow g_i^{(r)'} - \tau$
    else if $g_i^{(r)'} < -\tau$ **then**
      $\boldsymbol{\delta}^{(\ell)} \leftarrow \boldsymbol{\delta}^{(\ell)} \cup e(\mathbf{0}, i)$
      $\Delta_i^{(\ell)} \leftarrow -\tau$
      $g_i^{(r)''} \leftarrow g_i^{(r)'} + \tau$
  $\boldsymbol{\theta''} \leftarrow \boldsymbol{\theta'} - \epsilon\boldsymbol{\Delta}^{(\ell)}$ ▷ Update local replica with local values
  $\boldsymbol{K}^{(\boldsymbol{tr})} \leftarrow FwdPass(\boldsymbol{\theta''}, \mathbf{X}^{(\mathbf{tr})}, \mathbf{y}^{(\mathbf{tr})})$ ▷ Evaluate metrics on the training mini-batch
  $\boldsymbol{\delta}^{(\boldsymbol{p})} \leftarrow$ *Get peers message maps()* ▷ Receive, uncompress peer messages and extract the message maps.
  $\boldsymbol{\Delta}^{(\boldsymbol{p})} \leftarrow$ *Rebuild gradients*$(\boldsymbol{\delta}^{(\boldsymbol{p})})$
  $msg \leftarrow$ *Build message* (serialise the variables in Table 1 in an IT_RES message, sign and compress)
  Send(msg) ▷ Broadcast message to network
  Mine$(...)$ ▷ Mine

mod $\xi + 1$, where i is the rank of the supervisor's ID after sorting all supervisors in lexicographical order. If the next potential leader (index $(i-1) \bmod \xi + 2$) submits a valid multi-party *REPLACE_LEADER* special transaction backed by at least 2/3 of supervisors then the current leader is replaced by the next one. The transaction contains the reason why the leader was replaced (offline, too slow, malicious etc). This procedure can continue until a good leader is found.

Supervisors help miners to prove that they performed honest ML work by recording the messages sent during the training, the *message history*. Verifiers are only interested in the segments of the message history that can demonstrate useful work. Therefore, during an epoch the leader will pack sets of consecutive messages into *slots*. Supervisors agree on the exact order of the messages in a slot using preferential voting (e.g. Schulze method [20]). After the votes are cast, the leader computes the result per slot and publishes the hash of it as a *MESSAGE_HISTORY* transaction. The slot raw

TABLE 1: Payload of an IT_RES message.

field	bytes	type	notes
version	2	short	message version
task_id	16	uuid	task id
msg_type	1	char	IT_RES
ξ	2	ushort	epoch
$\lvert\delta^{(\ell)}\rvert$	2	ushort	no. of peer updates
$\delta^{(\ell)}$	$4 * \lvert\delta^{(\ell)}\rvert$	list	see Fig. 3
$\lvert K^{(tr)}\rvert$	2	ushort	no. of metrics
$K^{(tr)}$	$4 * \lvert K^{(tr)}\rvert$	list	metrics
$t^{(s)}$	4	uint	UNIX timestamp
$t^{(f)}$	4	uint	UNIX timestamp
$hash(b_i)$	32	uint256	SHA256
$hash(\Delta^{(p)})$	32	uint256	SHA256
$hash(\theta)$	32	uint256	SHA256
$hash(g^{(r)})$	32	uint256	SHA256
$hash(ZNB)$	32	uint256	SHA256
$hash(\delta^{(p)})$	32	uint256	SHA256
sgn	65	string	DER-encoded ECDSA

data is kept by all supervisors and can be downloaded by miners or verifiers.

Worker nodes lose their stake if they skip assigned iterations (5% for miners and 10% for supervisors). A worker node rejoining a task has to catch up with the latest weight updates. Supervisors can add another miner to the group if the number of miners drops below 80% of the initial size. A supervisor is replaced when he/she fails to record more than 10% of the iterations. The leader initiates the replacement procedure by publishing a special transaction called *RECRUIT_WORKER_NODE*. We provide all the details about the detection of offline nodes and the replacement procedure in the Supplementary material (Appendix E).

4.4. Finalisation

The client provides the test dataset to the evaluators and they will verify if the client did not cheat (i.e. the hashes match the withheld test mini-batches). They check the model from each miner on the test dataset and they send the model with the best performance to the client. Evaluators are selected using a matching algorithm similar to the one used for worker nodes.

TABLE 2: Notation used in Algorithm 1.

symbol	meaning
$\mathbf{X}^{(\mathbf{tr})}$	features in the mini-batch
$y^{(tr)}$	targets in the mini-batch
θ	model's state (weights and biases) at the beginning of iteration
θ'	model's state (weights and biases) after applying peer updates
θ''	model's state (weights and biases) after applying local updates
ϵ	learning rate
$\boldsymbol{\Delta}^{(p)}$	weight updates from peers
$\boldsymbol{\Delta}^{(\ell)}$	local weight updates
$\boldsymbol{g}^{(\ell)}$	local gradient
$\mathbf{g}^{(\mathbf{r})}$, $\mathbf{g}^{(\mathbf{r})'}$, $\mathbf{g}^{(\mathbf{r})''}$	residual gradient at different steps
$\boldsymbol{\delta}^{(\ell)}$	message map
$\boldsymbol{\delta}^{(p)}$	message maps from peers
$K^{(tr)}$	metrics on mini-batch
$t^{(s)}$	iteration start time
$t^{(f)}$	iteration end time

Evaluators must produce identical conclusions containing ML metrics. A conclusion is published using the special *CHARGE_FOR_TASK* transaction. If less than 2/3 of the evaluators produce identical reports, subsequent rounds of evaluators are drawn until a 2/3 consensus is reached. An evaluator can cheat by waiting to see the conclusions of others appearing in the mempool, and then publish an identical one. To prevent this, we break the procedure in two phases: 1) all evaluators post conclusions to the mempool in an encrypted form; 2) once they see that all conclusions have been posted, each of them then posts the decryption key. We call this method *commit-then-reveal*.

We summarised all the above steps in Fig. 4.

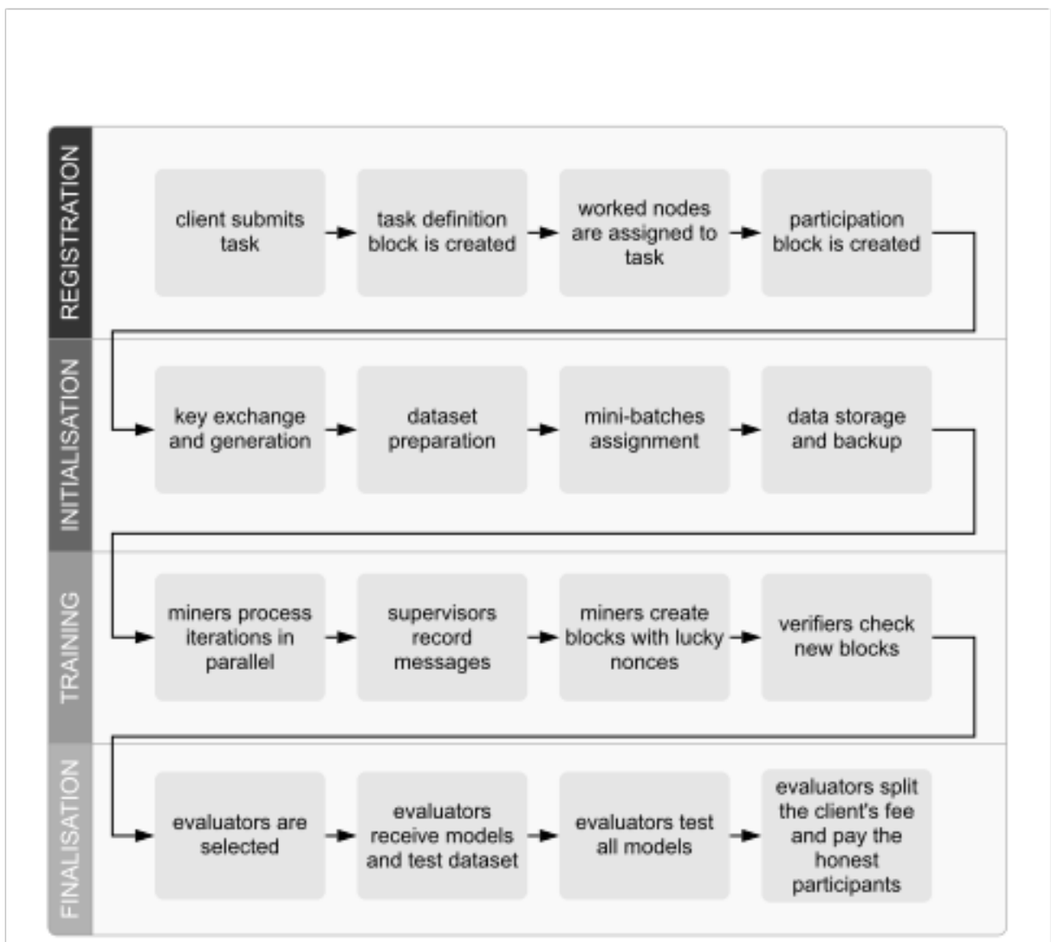

FIGURE 4: **ML task steps.** Summary of main events during a typical ML training procedure in the Tensor blockchain.

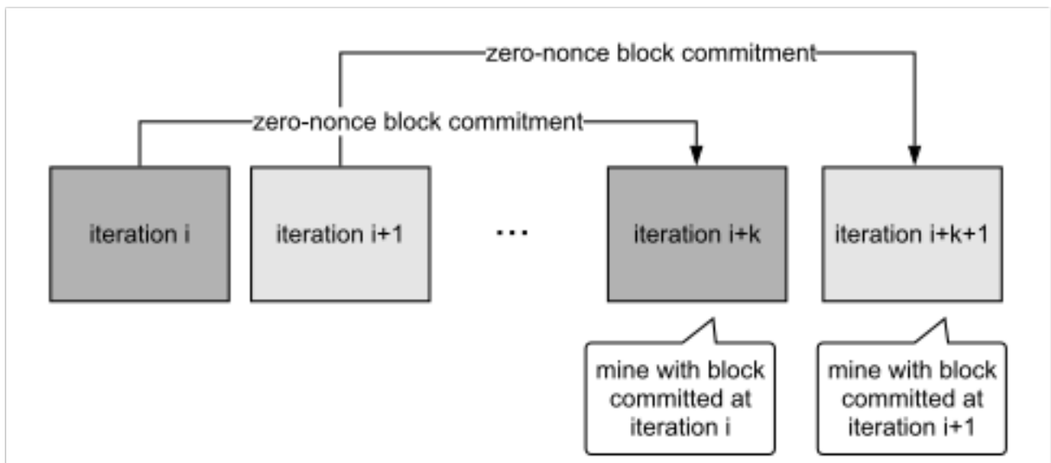

FIGURE 5: **Zero-nonce block commitment.** A miner announces k iterations in advance the zero-nonce block to be mined.

5. PROOF OF GRADIENT DESCENT

5.1. Mining

After each iteration, a miner has the right to mine a block. In the classical Bitcoin, a miner can obtain different hashes of the block header by varying the nonce. We limit the number of nonces to $a = \omega_B * disk\_size(b_i) + \omega_M * model\_size(\boldsymbol{\theta''})$, where $model\_size$ is a function returning the number of weighs and biases in the model, while ω_B and ω_M are network-wide coefficients. We want to ensure that hashing is insignificant and that most computing power is spent for ML training.

A miner might also generate billions of hashes by manipulating the transactions included in the block (e.g. changing the timestamp). Therefore, we require that k iterations before, the miner should "commit" to a *zero-nonce block* (ZNB), as illustrated in Fig. 5. A ZNB is built using a fixed header and a fixed set of transactions, with the nonce and other auxiliary fields necessary for PoUW set to zero. A miner should include the ZNB hash in the IT_RES message at iteration i. At iteration $i + k$, the miner replaces the nonce from the ZNB with the nonces obtained with by-products of ML training, as described in Algorithm 2. The nonce is a double hash of the concatenation of the local model at the end of the iteration (θ'') and the local gradients. We call the first hash obtained with the by-products of ML work the *nonce precursor*.

Algorithm 2 Mining operation

procedure MINE(...)
  $nonce_{precursor} \leftarrow hash(\boldsymbol{\theta''}|\Delta^{(\ell)})$ ▷ Build nonce precursor
  $nonce \leftarrow hash(nonce_{precursor})$ ▷ Build nonce
  $a = \omega_B * disk\_size(b_i) + \omega_M * model\_size(\boldsymbol{\theta''})$ ▷ No. of nonces allowed
  for $j \leftarrow 0$ to $a - 1$ **do**
    $success = MineWithNonce(nonce + j)$
    if *success* **then**
      $Store(\boldsymbol{\theta}, b_i, g^{(r)}, \Delta^{(p)}, h_i)$ ▷ Miner stores: model from beginning of iteration, mini-batch, gradient residual, peer updates and the relevant message history IDs
      $block \leftarrow MakeBlock(...nonce, hash(h_i), hash(msg)...)$ ▷ New block with PoUW fields.
      $AddToTensorBlockChain(block)$ ▷ Adds the block to the Tensor blockchain.

Upon successful mining, the miner stores the initial model state (weights and biases), the mini-batch, the initial gradient residual and the peer updates, but also downloads and stores the relevant message history slots from the supervisors. Furthermore, creates a new block with $hash(h_i)$ and $hash(msg)$ as extra fields for lookup, where $hash(h_i)$ refers to the Merkle tree of the message history slots containing the IT RES messages from iteration i up to iteration $i + k$, while $hash(msg)$ is the hash of the IT RES message corresponding to the lucky iteration. After successful verification, the block is added to the Tensor blockchain. Fig. 6 shows the block header particularities of PoUW vs. the Bitcoin.

For settlement, transactions are added from the mempool to the blockchain through mining. When there are periods without client tasks, to ensure continuous mining, miners will work on several network-wide predefined tasks, such as protein structure prediction using DNN. Miners would not get a client's fee, only the block's reward and the transactions' fees. Exceptionally, supervisors, verifiers and evaluators would get paid from the block's reward. After all blocks with non-zero subsidy have been mined, they will be paid from the transactions' fees. To prevent a shortage of miners, the network should have a limited number of dedicated mining agents belonging to Project Tensor.

5.2. Verification.

Verification of a mined block is delegated to the *verifier* because re-running an iteration is computationally expensive. Verification has the following steps:

- Ten *verifiers* are automatically selected based on the mined block hash (as a random seed).
- Verifiers first check if the hash of the block is under the network target T (as in Bitcoin).

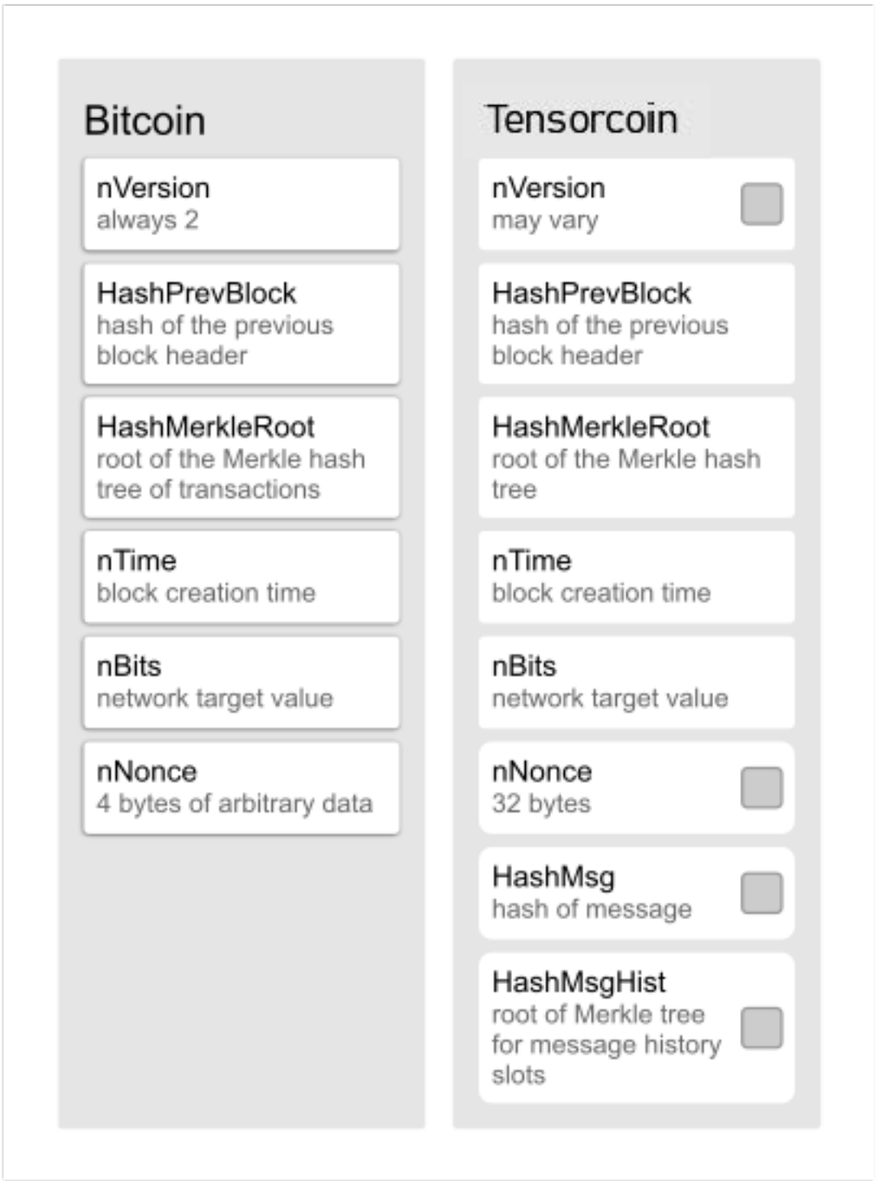

FIGURE 6: **Block header differences between Bitcoin and Tensor.** Differing fields are marked by a square sign. We used the same notation and block structure as in [21] .

- They extract the fields from the block: $nonce, hash(h_i)$ and $hash(msg)$ and receive the mini-batch, the local model from the start of the iteration, the gradient residual, the peers' weight updates and the relevant message history part.
- After extracting information from the message history, they can check if the miner is legit and if the ticket and the ML task are valid.
- A verifier compares the hashes and decides if the message history slot was registered as a special transaction in a previous block and if the hash of the starting model, the gradient residual and the peers' updates were announced in the previous iteration.
- They verify if batch b_i exists and if it should have been processed at that particular time in that order, as given by the preset order of messages from the initial allocation.
- From message history, verifiers can check if the $i + k$ commitment is valid, by comparing it with the zero-nonce block version of the currently mined block.
- Given the mini-batch, the starting local model, the peers' updates and the gradient residual, verifiers re-run steps 2-18 from the Algorithm 1 and check if the obtained metrics coincide with the ones sent to the network.

- Verifiers compress, hash the peer messages and check if the results match the hash reported in the previous message.
- With the end iteration model state and the local weights map calculated, a verifier will reconstruct the nonce precursor and the nonce and compare it with the one provided in the mined block.
- Verifiers will also check if the metrics improved over a time window Δt.
- Each verifier will post an encrypted digest (as a transaction) containing the nonce precursor to the mempool.
- The miner watches for digests to appear and decides at some point to post a *COLLECT_VERIFICATIONS* special transaction. The miner is interested to get as many confirmations as possible because the next block subsidy is proportional to them (in increments of 10%, up to 100% of the usual Bitcoin subsidy per block when 10 approvals are collected) and he/she wants other miners to build on top of his block in the future.
- Verifiers will reveal their digests by providing the encryption key as another transaction. The digests also contain the metrics reported on the task.

Regular peer nodes check if the block hash is under the network target as in the Bitcoin protocol, but they also watch for verifiers' digests to verify if the nonce was generated from the nonce precursor. The transactions containing digests for the previous block are included in the current block to prove the subsidy amount in the *coinbase transaction* (the transaction that pays the miner for finding the block).

6. IMPLEMENTATION

We implemented an early stage proof-of-concept (PoC) of PoGD.

PoGD Core contains the code related to the disributed ML training (Algorithm 1), verification (Subsection 5.2) and mining (see Algorithm 2). For the ML training, we used MXNet [26], which is suited for high performance machine learning and has support for dynamic computational graphs. According to our preliminary tests, MXNet is generally faster than several other similar ML frame-works. MXNet is also used by Amazon in their ML cloud offerings and it is an open-source project, part of the Apache Incubator.

PoGD Blockchain is the underlying blockchain used for our PoC and it is a modified version ofProject PAI's blockchain. It contains the block header modifications, special transactions and other blockchain logic. PoGD Core cannot function without PoGD Blockchain and viceversa.

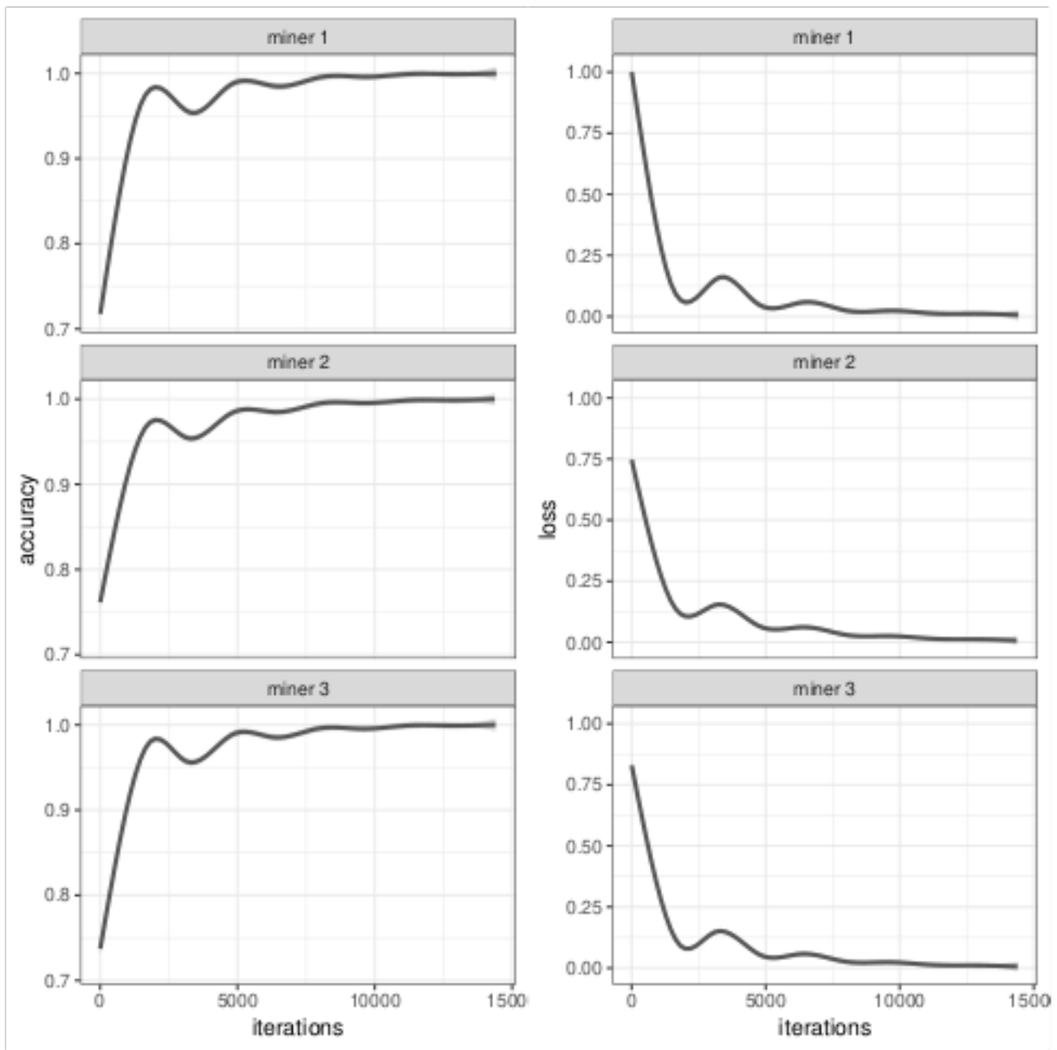

FIGURE 7: **Convergence.** In this example, a distributed ML system with 3 miners improves the accuracy and minimises the loss as the training is progressing.

PoGD Simulation is a project based on Kubernetes [23] to simulate a PoUW environment with different actors for how to setup a simulation). It can be used to study a Tensor network on a local or cloud machine. Fig. 7 shows an example of how our ML system is improving the metrics during the training. To collect these metrics, 3 miners trained a fully-connected DNN using the standard MNIST dataset from [11] on a NC6 Microsoft Azure machine (Intel Xeon E5-2690v3 CPU, 56 GB RAM and 1 x K80 GPU). The convergence topic and how the distributed system behaves with different numbers of training nodes on various data sets are outside the scope of this paper (see [4] for more information on these matters).

PoUW is now part of the Project Tensor initiative.

7. DISCUSSION

Our protocol brings several advantages over the classical Bitcoin. We use a modified hybrid PoW/PoS consensus with better security and we provide more rewards to network participants. However, due to its inherent complexity, our solution takes into account

and mitigates potential performance and security risks. In the following subsections, we provide an economical analysis and discuss about several performance optimisations and adversarial scenarios important for the security of the system.

7.1. Economical analysis

Our PoUW solution is more profitable than Bitcoin mining and it is cheaper than ML cloud solutions. To prove it, we compare the costs of a client using a cloud ML solution vs. ours, and the return on investment (ROI) for miners in two scenarios: Bitcoin and PoUW mining. We assume that a client will not pay an hourly fee higher than what cloud providers are charging for a ML-capable virtual machine. Miners would not participate if their investment and operating costs exceed their profits. Miners would switch to Bitcoin mining if that would be more profitable.

To study the ROI and the profitability, we published an online PoUW cost calculator available online on the project page under the 'ROI Calculator' link. The variables' names and formulas are explained in the online document. We briefly describe the calculator:

- The first sheet contains global parameters, such as the client's hourly fee (in USD), the average number of paid participants per task, the ML distributed system efficiency, the price of the Tensor coin and the Bitcoin, the electricity price, no. of active miners in the Tensor network and the Tensor block revenue.
- The second sheet is a price study containing profits for the most popular Bitcoin mining rigs.
- The third sheet contains a comparison between using cloud ML training solutions from Amazon, Microsoft and Google vs. four local deep learning workstation configurations (two hardware configurations from [24] and two from Exxact [25].
- The fourth sheet shows the client's profit if he/she uses our system.

We assume a PoUW miner invests initial capital to buy ML hardware that is amortised in 3 years. It is cheaper for a miner to buy and use a local workstation than use similar cloud machine configurations. The total costs after 1 and 3 years for local vs. cloud scenarios are shown in Fig. 8. The formula for the *return on investment* (ROI) as a percentage is:

$$R = 100(\frac{g\left(\frac{F_h}{Q} + 6\frac{W}{U}\right)}{\frac{C_H}{26280} + E} - 1),$$

where g is the number of GPUs in the configuration, F_h is the client hourly fee (*fee_usd_client_1h* from the cost calculator), Q is the average number of paid participants per task (*paid_participants_per_task*), W – miner's block _ _ revenue (*revenue_tensor_block* * *price_usd_tensorcoin*, U – average number of miners in the network at any moment (*miners active*), C_H –initial cost of hardware _ (*cost usd hwd*), E – electricity price per hour (*power kwh 1h_* cost usd 1h*). To assess how ROI changes in respect to the input variables, we provide its partial derivatives:

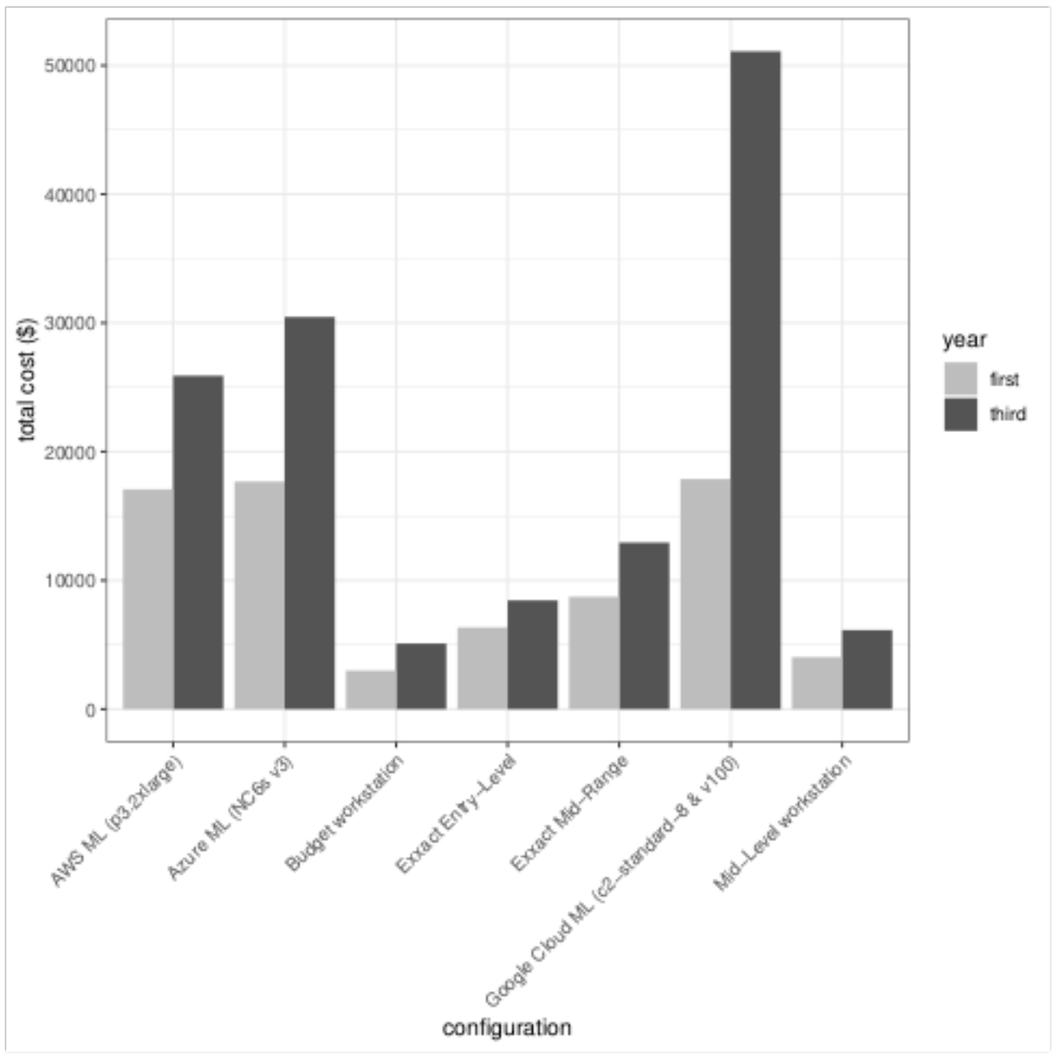

FIGURE 8: **Total costs at 1 and 3 years.** Budget, Mid-Level and Exxact configurations are local solutions. AWS, Azure and Google are cloud solutions.

$$\frac{dR}{dF_h} = \frac{2628000g}{Q\left(C_H + 26280E\right)}$$
$$\frac{dR}{dQ} = -\frac{2628000g \cdot F_h}{Q^2\left(C_H + 26280E\right)}$$
$$\frac{dR}{dW} = \frac{15768000g}{U\left(C_H + 26280E\right)}$$
$$\frac{dR}{dU} = -\frac{15768000g \cdot W}{U^2\left(C_H + 26280E\right)}$$
$$\frac{dR}{dC_H} = -\frac{2628000g \cdot \left(F_h \cdot U + 6 \cdot Q \cdot W\right)}{QU\left(C_H + 26280E\right)^2}$$
$$\frac{dR}{dE} = -\frac{69063840000g\left(F_h \cdot U + 6 \cdot Q \cdot W\right)}{QU\left(C_H + 26280 \cdot E\right)^2}$$

To quantify the importance of the ROI variables, we need to take into account real-world constraints: the electricity price is in an interval, the block reward is halving and there are static hardware configurations. In the case when $F_h = 0.50$ \$, $Q = 20$, $W = 1155$ \$ (revenue_tensorblock=1500 and price usd tensorcoin=0.77\$, the mean all-time price of Tensor), $U = 10000$, $C_H = 1945$ \$ and 0.12 \$ for 1 kWh, we have obtained the importance scores (in percentages) outlined in Fig. 9. We can see that electricity is the variable whose change has the most influence on the profitability. Fig. 10

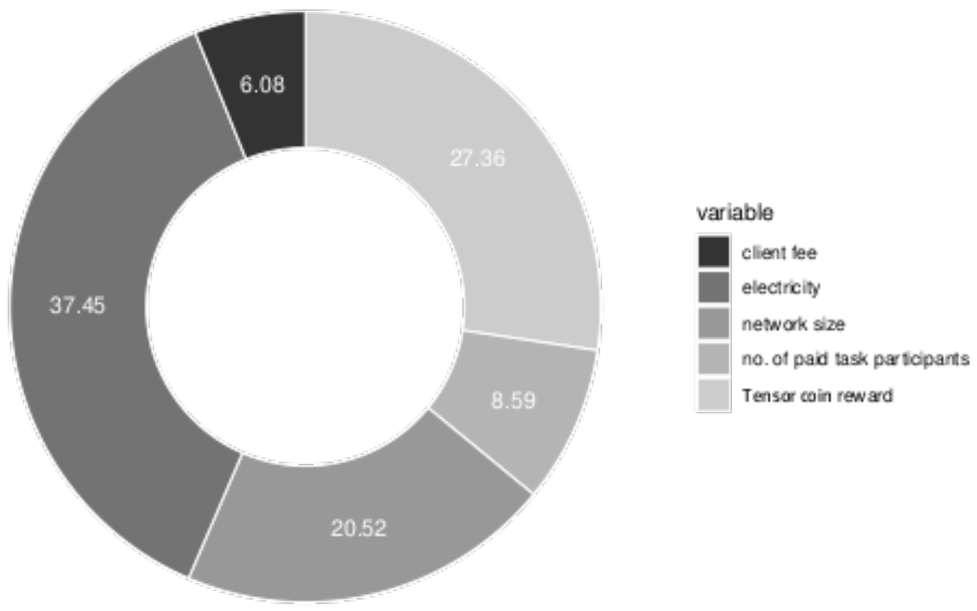

FIGURE 9: **Importance of variables.** We plotted the most important ROI variables (others are <1%). Variables were assessed by their absolute values.

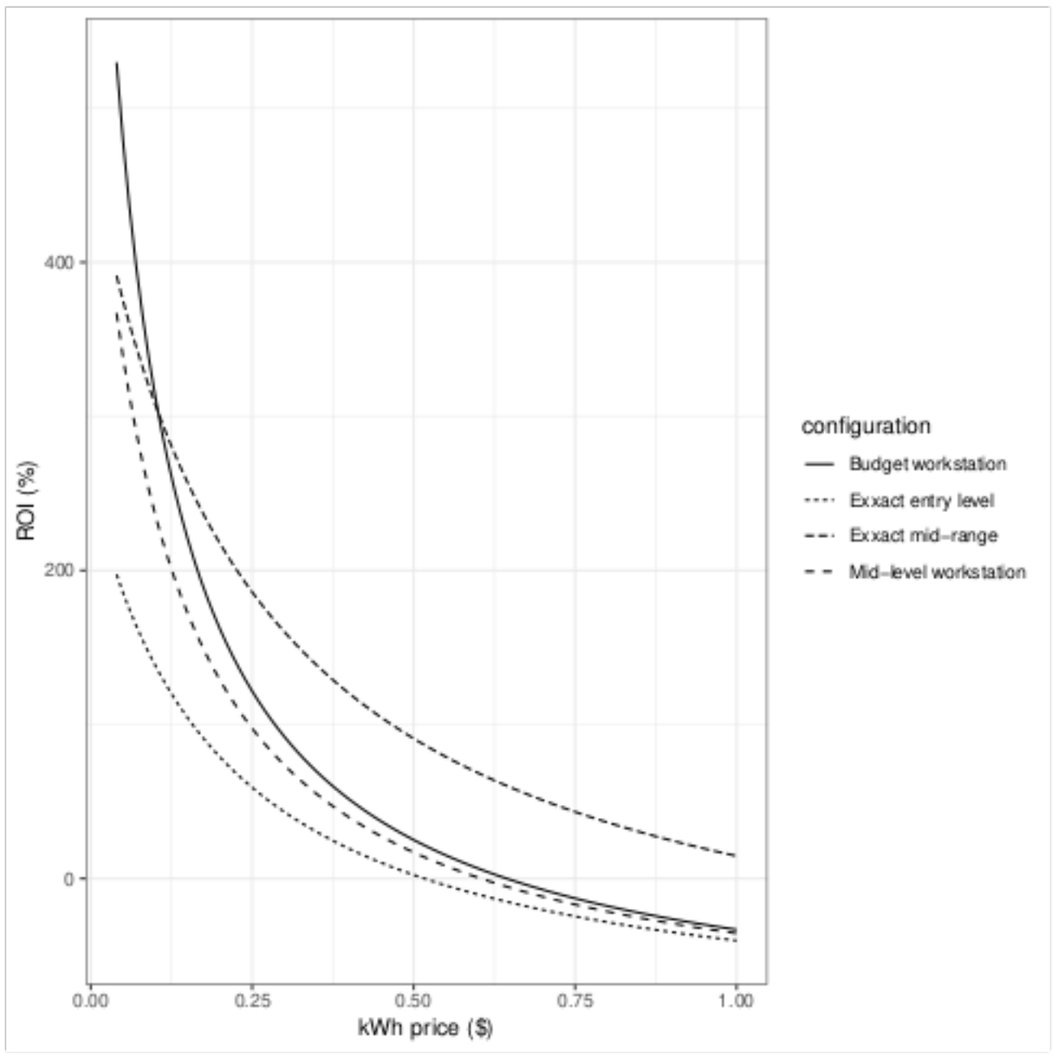

FIGURE 10: **ROI (%) as a function of electricity price.**

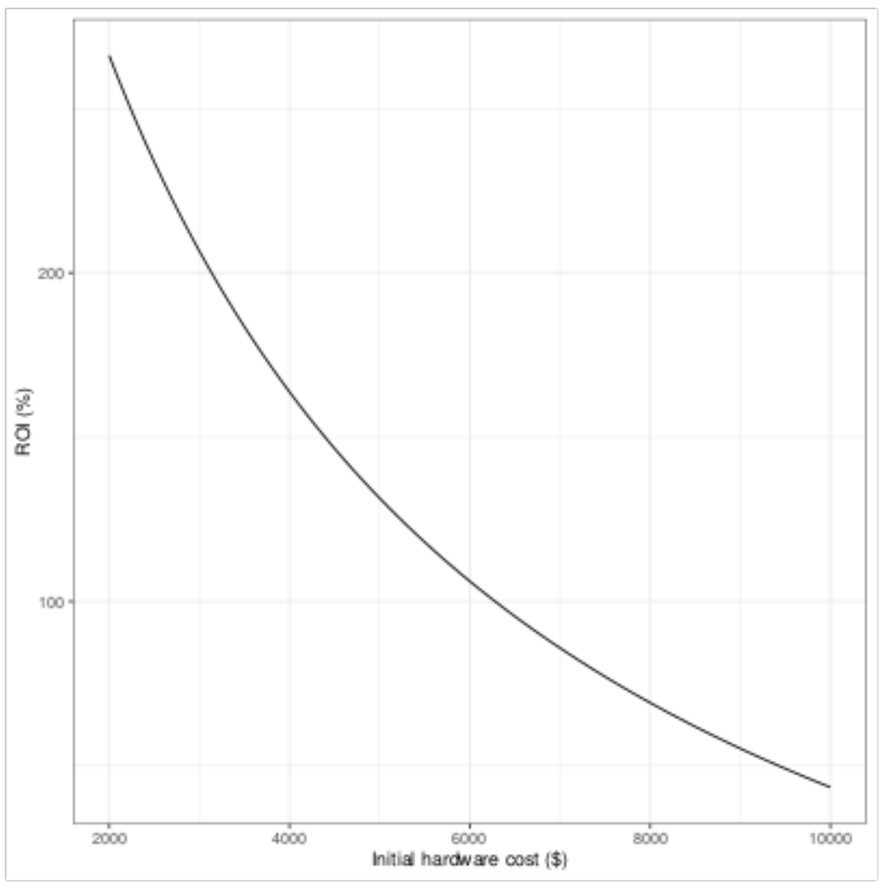

FIGURE 11: **ROI (%) as a function of initial hardware cost.**

shows a rapid decrease in profits as the price of a kWh increases. The same is true for the classical Bitcoin. The Tensor reward (composed of the block reward multiplied by the Tensor price) is the second most important variable, closely followed by the network size. Other variables that matter are the average number of paid task participants (miners, supervisors and evaluators) and the client's fee.

However, Fig. 9 does not capture the impact of the initial hardware investment. From Fig. 11, it might seem that PoGD profitability is higher for miners that invest less in their hardware configurations. To mitigate against this kind of risk, our solution has a matching mechanism between a specific task hardware preferences and the miners' systems capabilities (see Subsection 4.1). More expensive hardware comes with more GPUs ($g > 1$), and the miner can participate in several tasks simultaneously (one GPU per task). Additionally, the reward scheme R should allocate a higher compensation for the better hardware.

A miner participates in our network if his/her hourly profit $P_h^{(tensor)}$ is positive and greater than the profit obtained by mining Bitcoin using the best mining rig on the market ($P_h^{(btc)}$): $P_h^{(tensor)} > P_h^{(btc)}$ and $P_h^{(tensor)} > 0$. The hourly profit is calculated as: $P^{(tensor)}{}_h = g\left(\frac{F_h}{Q} + 6\frac{W}{U} - \frac{\not{C}_H}{26280} + E\right)$.

Clients use our PoGD system if they pay less than the cheapest cloud option $F_{h_{min}}^{(cloud)}$ (in our case, GCP is $F_{h_{min}}^{(cloud)} = 2.36$ \$), attenuated by a variable that describes the distributed system's efficiency, E_f, such that: $F_h < F_{h_{min}}^{(cloud)} \cdot E_f$. The advantage (%) gained by a client using our solution is $100\left(1 - \frac{F_h}{F_{h_{min}}^{(cloud)} \cdot E_f}\right)$.

With the price of Bitcoin being \$8243.41 (as of October 2019), only the best Bitcoin mining rig on the market (Antminer S17 Pro) achieved a 18% ROI in our calculations, while the PoUW miner configurations achieved a mean ROI of around 200% using the previously stated parameter values. Therefore, **Tensor coin** PoUW mining was approx. 10x more profitable than Bitcoin. For clients, we also obtained reduced fees by approx. 30% vs. using the

cheapest cloud ML option.

7.2. Performance considerations

A distributed solution will never be as efficient as running the whole training on a local machine, but there are several optimisations which can improve the performance: miners can receive gradients asynchronously during the whole iteration, the mini-batches can be preloaded, building the message map can be done in parallel. Mining can be performed in a different thread or process. Worker nodes can discard unnecessary data. Only verifiers carry out full verifications, regular peer nodes are not overwhelmed with expensive computations. In our Supplementary Material, we offer more information about performance characteristics and how we can increase the efficiency E_f.

To optimise bandwidth, we use the "dead-reckoning" scheme from Strom. The technique achieves a reduction of 99.6 % in data transfers for a model with 14.6 million parameters.

7.3. Security

We assume that the majority of participants is honest. We designed our PoUW protocol to avert various threats from Byzantine actors. We list the strategies to disincentivize and make cheating impossible in the following paragraphs:

Client uses pre-trained models
In this scenario, a client has pre-trained a model with good performance. He/she submits the ML task to the Tensor network and then acts as a miner. If the malicious actor is selected to work on the task, he/she could submit his/her pre-trained model without doing ML work, while mining for Tensor coins.

Homomorphic encryption would ensure adequate protection, but it is computationally infeasible [26]. To mitigate this risk, each miner must send weight updates at every iteration and must provide the inputs for lucky nonces. A malicious actor cannot perform cheap work because his/her progress depends on peer updates.

Client uses a test set from another distribution
The client might do this to obtain a trained model without having to pay for it because it would have a bad performance on the test dataset. The training and the test dataset should be identically distributed.

In our protocol, the client submits the whole dataset that is shuffled and split into a training and a test dataset based on the hash of the task definition block (as described in subsection 4.2.2). The task definition block is unknown, it doesn't exist before task submission and subsequent mining.

Client submits a malformed task definition
We allow the worker nodes to inspect and report if the task T is malformed. In that case, the client's fee is confiscated and distributed to worker nodes and to evaluators.

Gradient poisoning attack
Gradient poisoning is a type of attack in which a miner tries to skew the learning process by sending huge or fake gradients. Sending the same message multiple times is also a type of poisoning (spam). Blanchard et al. have proven in [27] that only one Byzantine actor could significantly affect the ML process and proposed the *Krum function* for detection. Damaskinos et al. also proposed the *Kardam filter* [28] for dealing with this threat.

Supervisors watch for this attack and add the malicious worker node to a blacklist which they expose publicly. Fellow nodes will ignore the gradient updates from the bad miners and evaluators will confiscate their stakes. Also, miners will not apply multiple IT_RES messages corresponding to the same iteration. To turn away miners that do not make progress, validators require that a lucky miner must prove that his/her local model improved over previous iterations.

Miner performs only mining
We constrain the number of nonces to make classical mining insignificant. A miner that would do bogus ML work and focus only on mining would be unable to prove the validity of the produced blocks. Bogus ML work includes: echoing received weight updates, leaving the task before completion or not following the steps in the ML training. It is economically damaging to the miners to engage in such behaviours because they would lose their stake and wouldn't receive any fee from the client anyway.

Sybil attacks
Bad actors can setup several Sybils on the network to collectively generate a bad model. They could also perform cheap work by replicating only one unit of work across all controlled nodes. To avert this attack, worker nodes are not allowed to pick tasks themselves, they only state their preferences (see Subsection 4.1). By doing so, they also cannot pick easy tasks.

Byzantine leader
If the leader of the supervisors delays publishing the *MESSAGE_HISTORY* transactions, then another leader is immediately elected. If the leader publishes invalid MESSAGE_HISTORY transactions, then he/she is added to the blacklist and replaced.

DOS attacks
Worker nodes may suspect that a DOS attack takes place when they do not receive enough peer updates or

when the training process is very slow (or stalled). They can pause the training process and resume it when the attack is over.

The procedure to defend against DOS attacks is: several worker nodes issue a *CONSIDER_PAUSE* transaction with the reason *DOS_ATTACK*. When a majority is considering to pause within a predefined time-frame, then honest workers emit a *PAUSE* transaction and everybody pauses. In pause mode, every concerned node will send *HEARTBEAT* off-chain messages to the former nodes in the ML task group. When a new majority of active nodes is formed, they can publish *CONSIDER_RESUME* messages and finally, *RESUME* transactions to continue the training process.

A verifier can reject a block mined during a DOS attack if there are enough elements to suspect that it was mined by an attacker.

Blockchain spam
A malicious miner could flood the blockchain with bogus blocks and determine honest verifiers to spend a considerable effort to validate them. This is also a DoS attack because it is hard to validate these blocks very fast. We adopt the following countermeasures:

- We prioritise the less expensive verification operations to be run first.
- A miner's stake gets confiscated and the miner is blacklisted if he/she submits invalid blocks.
- We limit the number of blocks a miner can publish during a predefined time interval.

Long-range attacks
In a *long range attack*, an attacker forks a large number of blocks or the entire blockchain starting with the genesis block. If the attacker has a high computational power, he/she can even outpace the main chain and publish his/her alternative chain.

Our underlying blockchain is a hybrid PoS/PoW blockchain protected by the longest chain rule. New blocks are created by spending a considerable amount of energy on useful work. We also require that at every 1024 blocks a checkpoint is created: everything before the checkpoint is truly immutable (i.e. no change can be done later to parts of the blockchain deeper than 1024 blocks). To establish a checkpoint, a set of 12 validators are randomly chosen to vote. At least 9 out of 12 votes are needed to establish a checkpoint.

8. CONCLUSION

We presented a novel proof of useful work concept using a distributed and decentralised machine learning system on blockchain. Our proposal can be easily extended to other AI algorithms.

We briefly reviewed the related work and outlined the unique characteristics of our solution. We conceived a different blockchain framework in which miners get compensation for doing useful work and we elaborated mechanisms to deter and punish bad actors. We presented the roles, the environment and the consensus protocol that combines machine learning with blockchain mining to create and reward useful work.

In our system, a client can train a ML model using a distributed network of worker nodes. After they perform a small unit of pre-assigned work, miners can mine new blocks with special *nonces*. At each iteration, nonces are obtained with a formula that takes into consideration inputs and by-products of the ML training. If a miner finds a lucky nonce, he/she must prove that he/she executed honestly the iteration so that his block will be accepted by the rest of the network. Verification means re-running the lucky iteration. Because miners are using data parallelism to train their models, they need to exchange information quickly using off-chain messages. Most of these messages carry data about updates that a miner performed to his/her local model. These updates are replicated across the task group by fellow worker nodes. A message history is recorded and will serve later in the verification. Compared to other blockchains, a node always receives compensation from the client; solving the blockchain puzzle is a bonus. Although we constrain the nonces to several values, the target difficulty in the network is very low in order to mine a block every 10 minutes as in the Bitcoin protocol. We shift the mining process towards ML training, while the actual hashing is insignificant.

We also implemented a proof of concept for PoUW. We showed that our PoUW solution is more cost-friendly to a client than regular cloud ML training, but also more profitable to miners compared to Bitcoin mining. Our approach also shows that ML models can be trained collectively with good performance using commodity hardware owned by individuals. We believe that such a system would democratise artificial intelligence using the security of the blockchain technologies.

In this paper, we described the particular case of training a deep neural network (DNN), but the principles can easily generalise to most AI iteration-based algorithms. Future work includes adding multi-party secure computation to the protocol and a production-ready implementation of the system described in this paper.

ACKNOWLEDGMENTS

The authors would like to thank Muhammad Naveed, Assistant Professor of Computer Science at the University of Southern California for his extraordinary assistance and for reviewing this work.

REFERENCES

1. Nakamoto, S. (2009). Bitcoin: A peer-to-peer electronic cash system.
2. Ball, M., Rosen, A., Sabin, M., and Vasudevan, P. N. (2017) Proofs of useful work. IACR Cryptology ePrint Archive, 2018, 203.
3. King, S. (2013).Primecoin: Cryptocurrency with prime number proof-of-work. http://primecoin.io/bin/primecoin-paper.pdf.
4. Decentralized machine learning: whitepaper. **https: //decentralizedml.com**.
5. SingularityNET: A decentralized, open market and inter-network for AIs. **https://public. singularitynet.io/whitepaper.pdf**.
6. Gridcoin whitepaper. **https://gridcoin.us/assets/img/whitepaper.pdf**.
7. Baldominos, A. and Saez, Y. (2019) Coin.AI: A proof-of-useful-work scheme for blockchain-based distributed deep learning. *CoRR*, **arXiv**.
8. Li, M., Weng, J., Yang, A., Lu, W., Zhang, Y., Hou, L., Liu, J., Xiang, Y., and Deng, R. H. (2019) Crowdbc: A blockchain-based decentralized framework for crowdsourcing. *IEEE Transactions on Parallel and Distributed Systems*, **30**, 1251–1266.
9. Boneh, D., Lynn, B., and Shacham, H. (2004) Short signatures from the weil pairing. *Journal of Cryptology*, **17**, 297–319.
10. developers, D. Decred documentation. **https://docs.decred.org/**.
11. LeCun, Y. and Cortes, C. (2010). MNIST handwritten digit database.
12. Micali, S., Rabin, M., and Vadhan, S. (1999) Verifiable random functions. *Proceedings of the 40th Annual Symposium on the Foundations of Computer Science*, New York, NY, October, pp. 120–130. IEEE.
13. Gilad, Y., Hemo, R., Micali, S., Vlachos, G., and Zeldovich, N. (2017) Algorand: Scaling byzantine agreements for cryptocurrencies. *Proceedings of the 26th Symposium on Operating Systems Principles*, New York, NY, USA SOSP '17, pp. 51–68. ACM.
14. Pedersen, T. P. (1991) A threshold cryptosystem without a trusted party. *Proceedings of the 10th Annual International Conference on Theory and Application of Cryptographic Techniques*, Berlin, Heidelberg EUROCRYPT'91, pp. 522–526. Springer-Verlag.
15. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. (1997) Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing, New York, NY, USA STOC '97, pp. 654–663. ACM.
16. Mirrokni, V., Thorup, M., and Zadimoghaddam,
    a. (2018) Consistent hashing with bounded loads. Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 587–604.
17. Duminuco, A. (2009) Data Redundancy and Mainte-nance for Peer-to-Peer File Backup Systems. Phd the-sis T´el´ecom ParisTech.
18. Reed, I. and Solomon, G. (1960) Polynomial codes over certain finite fields. Journal of the Society for Industrial and Applied Mathematics, 8, 300–304.
19. Goodfellow, I., Bengio, Y., and Courville, A. (2016) Deep Learning. The MIT Press.
20. Schulze, M. (2011) A new monotonic, clone-independent, reversal symmetric, and condorcet-consistent single-winner election method. Social Choice and Welfare, 36, 267–303.
21. Okupski, K. (2014). Bitcoin developer reference.
22. Chen, T., Li, M., Cmu, U. W., Li, Y., Lin, M., Wang, N., Wang, M., Xu, B., Zhang, C., Zhang, Z., and Alberta, U. MXNet : A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiV, 2015, 1–6.
23. What is Kubernetes. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.
24. Chen, J. (2019). How to build the perfect Deep Learning Computer and save thousands of dol-lars. https://medium.com/the-mission/how-to-build-the-perfect-deep-learning-\ computer-and-save-thousands-of-dollars-\ 9ec3b2eb4ce2.
25. Exxact Corp – The Best Deep Learning Workstations in the Business. https://www.exxactcorp.com/Deep-Learning-NVIDIA-GPU-Systems.
26. Rist, L. Jan Encrypt your Machine Learning. Medium, 2018.
27. Blanchard, P., El Mhamdi, E. M., Guerraoui, R., and Stainer, J. (2017) Machine learning with adversaries: Byzantine tolerant gradient descent. Proceedings of the 31st International Conference on Neural Information Processing Systems, Red Hook, NY, USA NIPS'17 118–128. Curran Associates Inc.
28. Damaskinos, G., Mhamdi, E. M. E., Guerraoui, R., Patra, R., and Taziki, M. Asynchronous Byzantine Machine Learning. arXiv, 2018.
29. Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. (2002) Seti@home: An experiment in public-resource computing. Commun. ACM, 45, 56–61.
30. Miller, A., Juels, A., Shi, E., Parno, B., and Katz, J.(2014) Permacoin: Repurposing bitcoin work for data preservation. Proceedings of the IEEE Symposium on Security and Privacy, May. IEEE.
31. Labs, P. (2017). Filecoin: A decentralized storage network. https://filecoin.io/filecoin.pdf .~
32. Vorick, D. and Champine, L. (2014). Sia: Simple decentralized storage. https://sia.tech/sia.pdf.
33. Schrijvers, O., Bonneau, J., Boneh, D., and Roughgar-den, T. (2017) Incentive compatibility of bitcoin mining pool reward functions. In Grossklags, J. and Preneel, (eds.), Financial Cryptography and Data Security, Berlin, Heidelberg, pp. 477–498. Springer Berlin Hei-delberg.
34. Miller, A., Juels, A., Shi, E., Parno, B., and Katz, (2014) Permacoin: Repurposing bitcoin work for data preservation. Proceedings - IEEE Symposium on Security and Privacy, may, pp. 475–490. IEEE.
35. (2020) Bitcoin Core version 0.9.0 released.
36. Du, J. (2018) PoUW Research Report.
37. Zhang, B. and Srihari, S. N. (2003). Properties of binary vector dissimilarity measures.
38. Prechelt, L. (2012) Early Stopping — But When? In Montavon, G., Orr, G. B., and M¨uller, K.-R. (eds.), Neural Networks: Tricks of the Trade: Second Edition. Springer Berlin Heidelberg, Berlin, Heidelberg.

39. Chiu, J. and Koeppl, T. V. (2018). Incentive compatibility on the blockchain Waters, A., Harvilla, M., and Gerzanics, P. (2018) PAI Data Storage and Sharing.
40. Castro, M. and Liskov, B. (1999) Practical byzantine fault tolerance. Proceedings of the Third Symposium on Operating Systems Design and Implementation, Berkeley, CA, USA OSDI '99, pp. 173–186. USENIX Association.
41. Das, A., Gupta, I., and Motivala, A. (2002) Swim: Scalable weakly-consistent infection-style process group membership protocol.Proceedings of the 2002 International Conference on Dependable Systems and Networks, Washington, DC, USA
42. Backes, M. and Cachin, C. (2003) Reliable broadcast in a computational hybrid model with byzantine faults, crashes, and recoveries. In Proc. of Intl. Conference on Dependable Systems and Networks (DSN-2003, pp. 37–46. IEEE.

TABLE A.1: BUY_TICKETS

transaction	description
txin[0..n]	references to existing UTXOs
txout[0]	contains a value that is a multiple of the current price of tickets of the according type (miner, supervisor etc.); script field is empty!
txout[1]	returns change to the staker as P2PKH; this is the address to which mining remuneration will be paid
txout[2]	an OP_RETURN that declares ticket type and preferences: "TICKET-TYPE:preferences"

TABLE A.2: PAY_FOR_TASK

transaction	description
txin[0..n]	these reference existing UTXOs
txout[0]	declaring only the amount staked; script field is empty!
txout[1]	returns change to the staker
txout[2]	an OP_RETURN that describes the task

APPENDIX A. ANATOMY OF TRANSACTIONS

Tensor nodes recognise transaction types by reading their last TXOUT's. They contain descriptions in their OP RETURN scripts. Here is an example of a possible implementation. Details are given in Tables A.1, A.2 and A.3.

The specification of the Bitcoin OP_RETURN outputs states that the first opcode is OP_RETURN and it is followed by a push opcode. We introduced a richer format called *structured data outputs*, which are recognized as having OP_RETURN fields followed by special OP_STRUCT opcodes, which in turn are followed by an array of data items. The specification for the structured data format also states that the first data item must specify the version, followed by transaction specifics. For example, for PAY_FOR_TASK, the next fields contain details about the dataset, validation, optimiser etc.

APPENDIX B. SHORTENING TASK WAIT TIME

We aim to shorten the waiting time of starting a task. We require that the timestamp of the task is earlier than the block containing it and the task must eventually appear in all blockchain forks.

A node keeps a local list of tasks along with their detection time (which differs across nodes by seconds). To add new elements to it, each node searches the chain starting from task's timestamp, including orphaned forks in the traversal. Any mined block containing the task submission is a task definition block, but the nodes should recognise only one of them. Therefore, the oldest block is chosen, regardless of the branch. In case there is more than one block with an identical timestamp containing the same task submission, the one with the lowest hash is picked.

Selected workers start joining by sending JOIN_TASK transactions, in which they include the hash of their perceived task definition block and the task hash. A majority with a specific task definition hash will emerge and only those selected by it will start the training. Those who are not selected are ignored if they try to participate.

TABLE A.3: CHARGE_FOR_TASK

transaction	description
txin[0..n]	these reference task-related stakes (client's stake as well as captured stakes of participants who transgressed); script fields are empty!
txout[0..n]	these are classic P2PKH outputs that pay participants; note that only one CHARGE_FOR_TASK transaction is spendable
txout[n+1]	an OP_RETURN script that publishes the best model for the client to download: "RESULT:taskHash:url"

When training is completed, the part of the ledger around task submission moment will have been stabilised. Nodes have to find the task stake block to reference the PAY_FOR_TASK stake. They search the chain starting from the task's timestamp, traversing only the active branch and disregarding stalled forks. The first block containing the task submission is the task stake block.

Using this procedure the network can safely start working on tasks long before the stakes are confirmed.

APPENDIX C. A DISTRIBUTED KEY GENERATION (DKG) SCHEME

In our PoGD system, the private key shares are constructed using a distributed key generation (DKG) protocol during task initialisation, which is a modified version of the Joint-Feldman protocol ([18]). It contains the following steps:

- Each supervisor creates an t − 1-degree polynomial $S_i(x) = s_{i,0} + s_{i,1}x + ... + s_{i,t-1}x^{t-1}$, where all the coefficients except $s_{i,0}$ are randomly generated private keys. $s_{i,0} = s_{k_i}$ is the private BLS key of the participant.

- Based on their $S_i(x)$, all supervisors calculate and broadcast their own $P_i(x) = p_{i,0} + p_{i,1}x + ... + p_{i,t-1}x^{t-1}$ which is another polynomial of the same degree, that holds the corresponding public keys, such that $p_{i,j} = g_i \times s_{i,j}$, j ∈ –0, ..t − 1″.

- Every party evaluates S(x) for all participants and for itself, by replacing x with the corresponding index of each supervisor. For example, in case we are dealing with a 3-of-5 scheme, supervisor 2 will calculate the following:

$$
\begin{aligned}
S_2(1) &= s_{2,0} + s_{2,1} \cdot 1 + s_{2,2} \cdot 1 \\
S_2(2) &= s_{2,0} + s_{2,1} \cdot 2 + s_{2,2} \cdot 2^2 \\
S_2(3) &= s_{2,0} + s_{2,1} \cdot 3 + s_{2,2} \cdot 3^2 \\
S_2(4) &= s_{2,0} + s_{2,1} \cdot 4 + s_{2,2} \cdot 4^2 \\
S_2(5) &= s_{2,0} + s_{2,1} \cdot 5 + s_{2,2} \cdot 5^2
\end{aligned}
$$

- Every supervisor will encrypt and send every $S_i(x)$ to the corresponding parties. E.g. supervisor 2 will use supervisor 1's public key to encrypt $S_2(1)$ and send it to supervisor 1. In case one party does not receive all the private shares during a pre-set time window, he/she will complain against the senders by sending DKG COMPLAINT transactions to the blockchain containing the indexes of the senders.

- Every private share is decrypted upon arrival and verified using P(x) by replacing x with his/her own identifier/ index. If the result does not match the public key derived from the received private share, the node will fill a complaint against the sender by publishing a DKG COMPLAINT transaction on the blockchain, that contains the identity of the culprit and the received private share.

- To obtain the global public key ($\mathcal{P}_\mathbf{k}$), all supervisors will aggregate the free terms of all polynomials $\mathcal{P}(x)$: $\mathcal{P}_\mathbf{k} = \sum_{i=1}^{n} p_{i,0} = p_{1,0} + p_{2,0} + ... + p_{n,0}$, which are publicly known.

- The global private key $\mathcal{S}_\mathbf{k} = \sum_{i=1}^{n} s_{i,0} = s_{1,0} + s_{2,0} + ... + s_{n,0}$ is unknown to any party.

- At the end of the protocol, all supervisors must post *DKG_SUCCESSFUL* transactions containing the locally calculated *t-of-n* public key $\mathcal{P}_\mathbf{k}$. When n such transactions with the same public key are observed during a predefined time window Δt, then the DKG protocol is considered successful and the parties can proceed to the next phase.

A DKG protocol runs in the initialisation phase of a ML task (after the key exchange), but also whenever the supervisory committee changes. Supervisors that produce faults during DKG are banned from the network and their stakes are confiscated.

It is not easy to detect if a node received a wrong share or the node received a correct share but pretends that he/she didn't in order to exclude another node. If 2/3 of the nodes complain against one node, then that supervisor is automatically disqualified; however, if only one node complains against another, then the other nodes will vote based on the current evidence which of the nodes will be excluded: the sender or the receiver. The DKG is restarted with the existing non-faulty members if their reduced number still satisfies the size requirements or with new members replacing faulty ones if there would be less worker nodes than required. The selection procedure for additional worker nodes is the same as the one provided in the task registration phase. Miners will include JOIN_TASK transactions for the extra nodes in a second participation block that should reference the first participation block.

APPENDIX D. SIGNING *T-OF-N* TRANSACTIONS

Each supervisor has several secret key shares that are used to sign *t-of-n* transactions. For a transaction tx, each supervisor will follow these steps:

1. Compute $H(tx)$, which is the hash of the transaction to the BLS curve.
2. Publishes $Sig_i(tx) = \sum_{j=1}^{n} \mathcal{S}_j(i) \times H(tx)$, where i is the index of the current node and j indexes the private shares. Please note that each node i calculates the aggregation of its private key shares as $\sum_{j=1}^{n} \mathcal{S}_j(i)$ and uses it to sign the transactions.
3. The epoch leader collects at least t signature shares.

For a given transaction tx, as soon as any t signature shares are collected, due to the BLS threshold signature properties, the leader can reconstitute the global signature on the transaction ($Sig(tx)$) by performing Lagrange interpolation, as if the global private had been used to sign the transaction ($Sig(tx) = (s_{1,0}+s_{2,0}+...+s_{n,0}) \times H(tx)$). The global signature validates against the free coefficient of the global public key $\mathcal{P}(x)$.

APPENDIX E. CRASH-RECOVERY MODEL

We use a crash-recovery model to detect when and which nodes go offline. It is inspired from [47], a framework in which nodes may crash and recover repeatedly, while some may go offline permanently. In real-life, although there are potential network problems or software/hardware glitches, nodes eventually go back online and continue the ML training.

Appendix E.1. Offline detection

If a worker node suspects that another worker node (miner or supervisor) becomes too slow or does not send the expected messages in a reasonable amount of time t_r, then he/she launches a test to detect if the node is offline (crashed). The probing algorithm (Algorithm 3) is inspired from an algorithm called SWIM ([46]). We modified the algorithm to become Byzantine Fault Tolerant (BFT). As in PBFT ([45]), we require that more than 1/3 of the nodes (the maximum accepted number of possible faulty nodes in a BFT system) should declare that a particular node is online so that the entire task group considers that the node is online. We assume a weak synchrony as in [45], i.e. network faults are eventually repaired and most nodes are coming back from the offline mode fairly quickly.

Any worker node w_i keeps a local list of known active worker nodes W. As outlined in Algorithm 3, a supervisor s_i pings another suspected worker node w_j. If no response is received in a time interval Δt then s_i will randomly select a subset of k worker nodes ($W_k \subsetneq W$) nodes and ask them in parallel to also ping the node in question. To avoid any bias, the k chosen nodes are determined by a random number generator seeded with the hash of the last participation block. The number of "alive" responses are counted. If more than 1/3 of the enquired nodes declare that the node is alive then all nodes must keep it in their local list. Each response is signed by the sender. Otherwise, the suspecting worker node will publicly contact the leader and ask him/her to decide whether the node should be replaced (ReportOfflineNode procedure). To do so, the suspecting node will publish a transaction (*CHECK_NODE*) containing the responses from peers and the reason for investigation (e.g. "offline"). The leader runs an election in which supervisors must vote on whether the node should be declared offline or online. Each supervisor will directly ping the suspected offline node and if no response is received in a specified time interval, then the supervisor will vote it as offline. The leader will publish a *NODE_STATUS t-of-n* transaction with all the votes. If the final status is offline (2/3 or more of the supervisors voted "offline"), then the leader will publish a *RECRUIT_WORKER_NODE* transaction containing the ID of the replaced worker node and the reason for replacement. The working nodes will remove the offline node from their lists.

Algorithm 4 is similar to the Algorithm 3, but it is run only by supervisors.

A node that is going in offline-online mode too often must be removed from the task working group using the *CHECK_NODE* and *NODE_STATUS* transactions, with "offline-online" as the reason. Malicious nodes are reported and verified in the same way using different reasons (e.g. "denial-of-service attack", "gradient poisoning" – sending wrong updates to derail the ML training etc.).

Appendix E.2. Offline supervisors

A supervisor could go offline because of a faulty network connection. If he/she loses more than 10% of the training iterations, then he/she loses his/her stake and cannot rejoin the task.

Algorithm 3 Algorithm used by a worker node w_i to detect if a suspected node w_j is offline.

procedure DETECTOFFLINENODE(...)
  $r \xleftarrow{\Delta t} ping(w_j)$ ▷ Ping w_j and wait for a time Δt for a response
  if $r == online$ **then**
    $W \leftarrow W \cup w_j$ ▷ Keep or add it to the list
  else ▷ No response
    $W_k \leftarrow rnd(W \setminus \{w_i, w_j\})$ ▷ Pick k random worker nodes (a subset of W).
    $c \leftarrow 0$ ▷ Online counter.
    $R \leftarrow \{r\}$ ▷ Responses.
    parfor $w_k \in W_k$ **do** ▷ In parallel.
      $R_k \xleftarrow{\Delta t} ping\_req(w_k)$ ▷ Send ping requests to each chosen node.
      if $status(R_k) == online$ **then**
        $c \leftarrow c + 1$
    end parfor
    if $c > |W|/3$ **then** ▷ If more than 1/3 report the node as online
      $W \leftarrow W \cup w_j$ ▷ The node is alive.
    else
      $s \leftarrow ReportOfflineNode(w_j, W_k, R)$ ▷ Report the k chosen nodes and the responses.
      if $s ==$ offline **then**
        $W \leftarrow W \setminus w_j$ ▷ Remove the node from the active list.

Algorithm 4 Algorithm run by the leader to determine if he/she should declare a worker node as offline.

procedure INVESTIGATENODESTATUS(...)
  $c_x \leftarrow 0$ ▷ Offline counter.
  $r \xleftarrow{\Delta t} ping(w_j)$ ▷ Ping w_j and wait for a time Δt for a response
  if $r \neq online$ **then**
    $c_x \leftarrow 1$ ▷ Offline counter.
  $R \leftarrow \{r\}$ ▷ Responses.
  parfor $s_k \in S_k \setminus \{s_i\}$ **do** ▷ In parallel.
    $R_k \xleftarrow{\Delta t} ping\_req(s_k)$ ▷ Send ping requests to each supervisor.
    if $status(R_k) == offline$ **then**
      $c_x \leftarrow c_x + 1$
  end parfor
  if $c_x \geq 2/3|S|$ **then** ▷ If more than 2/3 report the node as offline
    $Tr(RECRUIT\_WORKER\_NODE, R)$ ▷ Issue transaction to recruit a new worker node
  else
    $Tr(NODE\_STATUS\_ONLINE, R)$

If a supervisor comes back online and he/she didn't lose over 10% of the iterations, then he/she can synchronise with the other supervisors. The supervisor will read from the public streams/databases of the other supervisors and will get up-to-date.

If a supervisor is missing for periods of more than 10% of the training, then the other supervisors will initiate a recruitment process to replace the absent supervisor. The leader will post a *RECRUIT_WORKER_NODE* transaction which will specify that a new supervisor is needed.

Appendix E.3. Offline miners

If a miner does not participate in more than 5% of all training operations, he/she will lose his/her stake. Otherwise, he/she can rejoin the task. No model synchronisation is needed, yet the miner has to catch up with the latest weight updates.

Another miner is called in by the supervisors when the total number of miners drops to less than 80% of the original size. The leader will publish a special transaction called *RECRUIT_WORKER_NODE*. The stake of the lost miner will be transferred to the task's stake and redistributed by the evaluators at the end of the training.

Appendix E.4. Recovery

A supervisor or miner that comes back online but did not lose enough iterations in order to be replaces will synchronise his/her internal database with the rest of the group. No voting to re-accept the old member is required.

APPENDIX F. PERFORMANCE OF THE ML DISTRIBUTED SYSTEM

The efficiency of a ML distributed system is less than the one of running the training on a single machine. We consider $E_f = 1$ when the training is run entirely on a single machine. In case of a distributed system, E_f is much lower, but can be increased by parallelising different operations, as shown in F.1.

In the left pane of the figure, we show the operations that would take place on a single machine. In the middle, we have an unoptimised scenario in which all operations are executed serially. In the right pane, we improved the execution: peer messages are asynchronously received during the whole iteration, the peer updates are summed and applied once, while the message map is done concomitantly with other related operations.

We measured the execution times for various steps in the main algorithm on a machine that used only the CPU and on another machine equipped with a GPU. The first machine was a MacBook Pro 2017 with 2,8 GHz Quad-Core Intel Core i7 CPU and 16 GB of RAM. The GPU machine was a NC6 Microsft Azure machine (E5-2690v3 Intel Xeon CPU with 56 GB RAM and 1 x K80 GPU). We provided the measurements in F.2 and F.3, respectively. As expected, most of the steps that involve ML operations (e.g. backpropagation, model updates) are executed faster on the GPU machine.

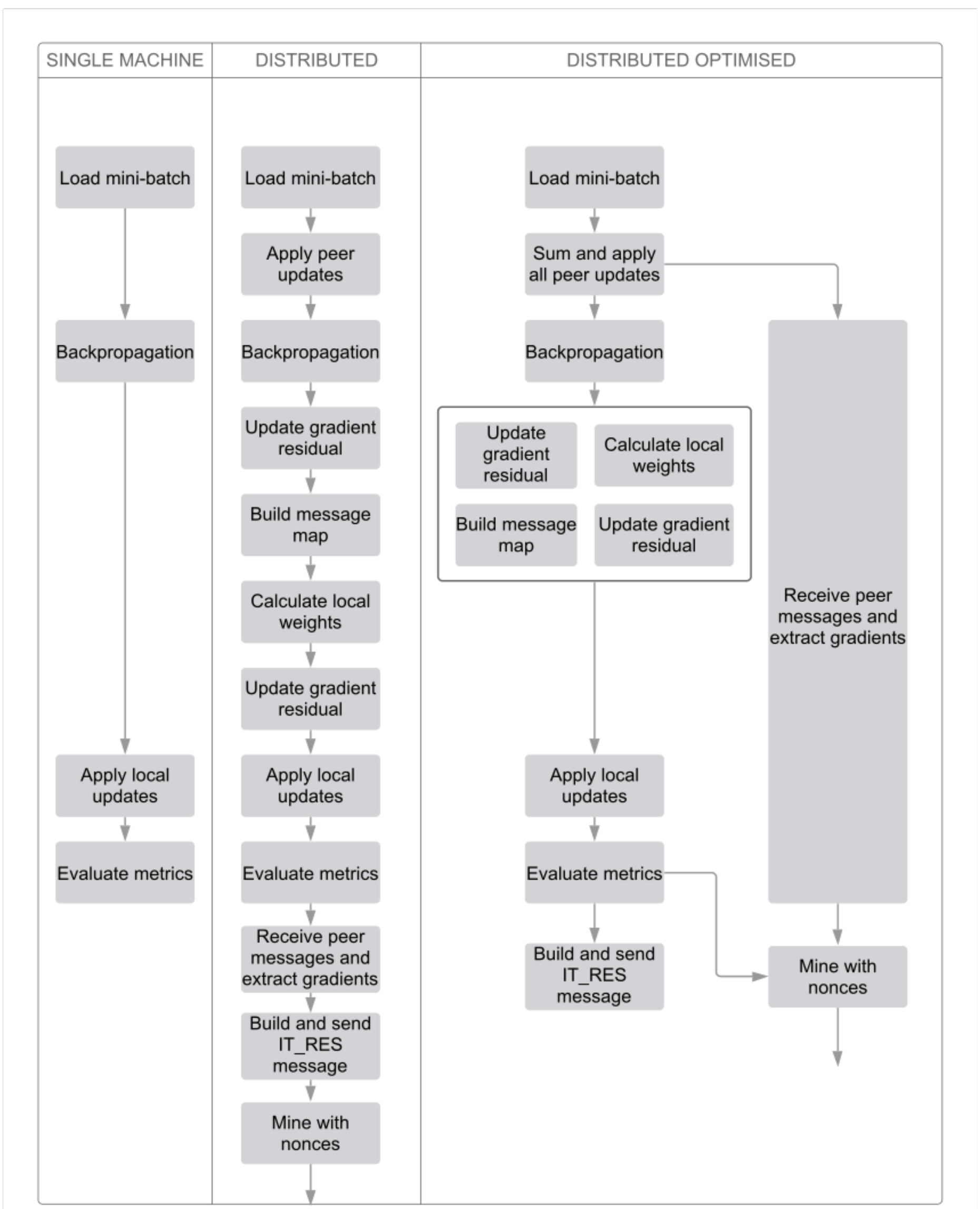

FIGURE F.1: **Performance optimisations.**

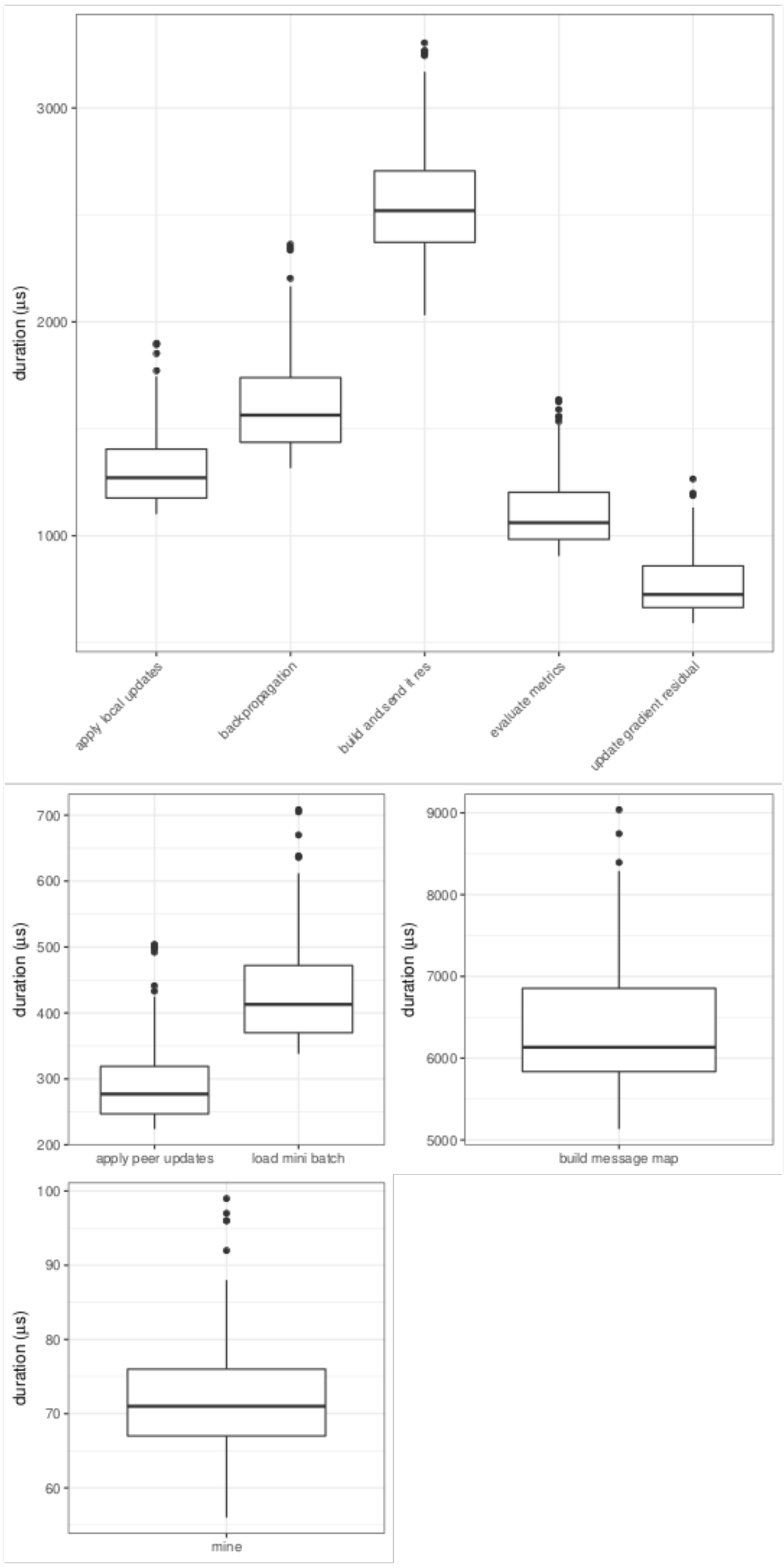

FIGURE F.2: **Time measurements for different steps on CPU.**

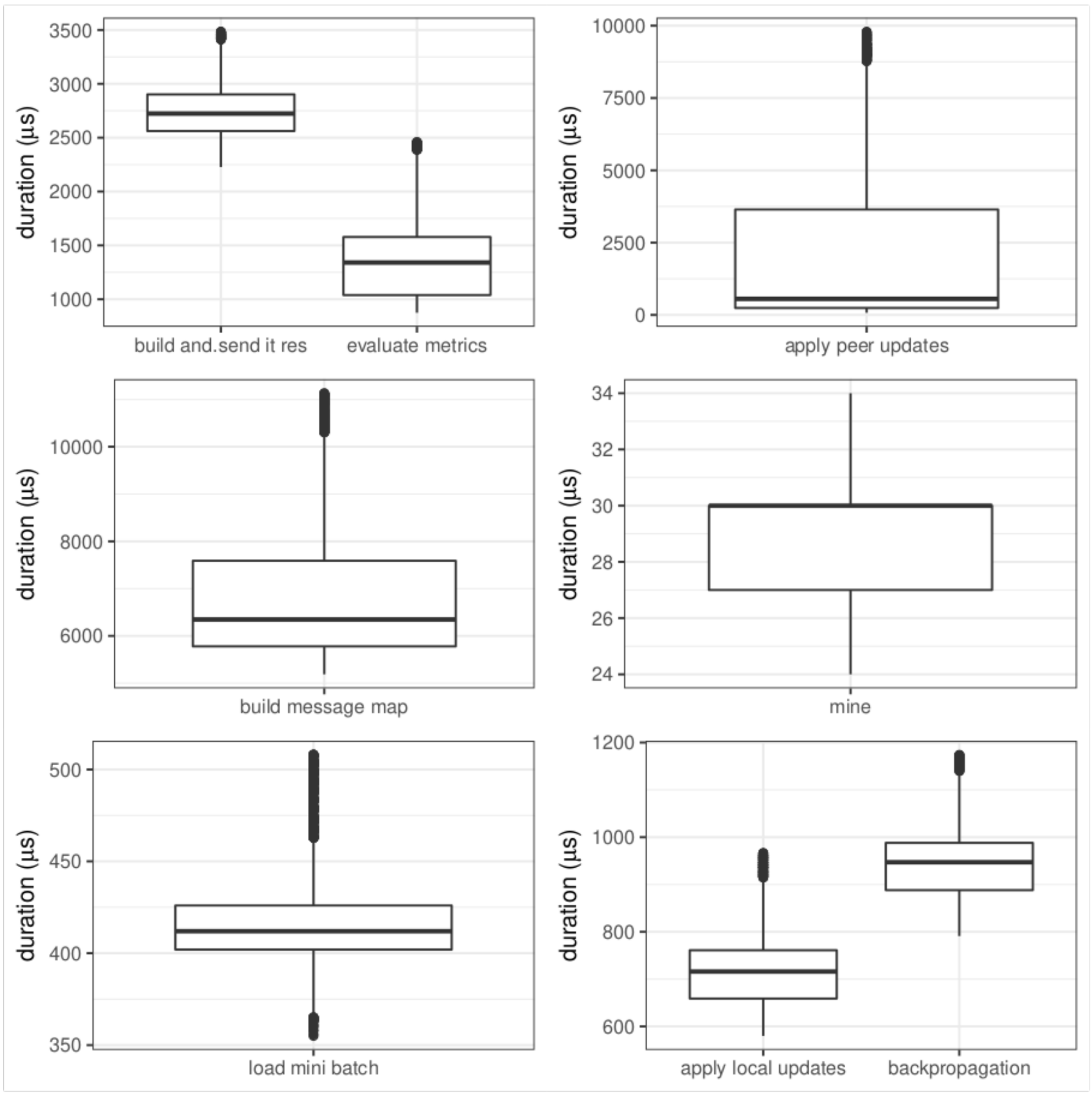

FIGURE F.3: **Time measurements for different steps on GPU.**