ECOLE
POLYTECHNIQUE
DE BRUXELLES

ULB

UNIVERSITÉ LIBRE DE BRUXELLES

Federated Learning in Swarm Robotics using Blockchain Smart Contracts

Mémoire présenté en vue de l'obtention du diplôme
d'Ingénieur Civil en informatique à finalité spécialisée

Sébastien De Vos

Directeur
Professeur Marco Dorigo

Superviseur
Dr Andreagiovanni Reina

Co-Superviseur
Dr Volker Strobel
Alexandre Pacheco

Service
IRIDIA

Année académique
2022 - 2023

ULB UNIVERSITÉ LIBRE DE BRUXELLES

ULB UNIVERSITÉ LIBRE DE BRUXELLES

Exemplaire à apposer sur le mémoire ou travail de fin d'études,
au verso de la première page de couverture.

Fait en deux exemplaires, **Bruxelles, le** 02/06/2023

Signature

Sébastien De Vos

Réservé au secrétariat :	**Mémoire réussi***	**OUI**
		NON

CONSULTATION DU MEMOIRE/TRAVAIL DE FIN D'ETUDES

Je soussigné

NOM :
De Vos

PRENOM :
Sébastien

TITRE du travail :
Federated Learning in Swarm Robotics using Blockchain Smart Contracts

AUTORISE*

~~**REFUSE***~~

la consultation du présent mémoire/travail de fin d'études par les utilisateurs des bibliothèques de l'Université libre de Bruxelles.

Si la consultation est autorisée, le soussigné concède par la présente à l'Université libre de Bruxelles, pour toute la durée légale de protection de l'œuvre, une licence gratuite et non exclusive de reproduction et de communication au public de son œuvre précisée ci-dessus, sur supports graphiques ou électroniques, afin d'en permettre la consultation par les utilisateurs des bibliothèques de l'ULB et d'autres institutions dans les limites du prêt inter-bibliothèques.

Formulaire version 04/2015

** Biffer la mention inutile*

** Biffer la mention inutile*

Résumé

Auteur : Sébastien De Vos
Titre du master : Ingénieur Civil en Informatique à Finalité Spécialisée
Année académique : 2022-2023
Titre du mémoire : Federated Learning in Swarm Robotics using Blockchain Smart Contracts

Le federated learning est une nouvelle approche de machine learning distribué. Il offre des avantages tels que la conservation des données localement et la distribution des calculs. Plus précisément, le federated learning signifie que chaque machine entraîne un modèle localement puis échange uniquement les paramètres appris. Cependant, pour agréger les paramètres entraînés localement, les travaux existants utilisent principalement des serveurs centralisés. Bien que cette approche montre un grand potentiel dans les systèmes multi-robots, elle ne convient pas aux systèmes de robotique en essaim qui, inspirés par les comportements collectifs des systèmes biologiques, visent à créer des systèmes décentralisés où des robots autonomes peuvent s'auto-organiser et atteindre des objectifs sans autorité centrale. Dans cette thèse, nous présentons un *proof-of-concept* de federated learning dans un essaim de robots ne compromettant pas la décentralisation. Pour ce faire, nous utilisons la technologie de la blockchain pour permettre à notre essaim de robots de synchroniser un modèle partagé qui est l'agrégation des modèles individuels sans dépendre d'un serveur central. Cette implémentation protège également l'entrainement des modèles contre des attaques de robots défectueux et malveillants, appelés robots Byzantins, pouvant survenir dans les réseaux décentralisés. Nos expériences sont menées dans ARGoS, un simulateur basé sur la physique de robotique en essaim, tout en utilisant le protocole blockchain Ethereum exécuté par chaque robot simulé. Nous montrons qu'introduire un seul robot Byzantin peut perturber considérablement le processus d'apprentissage des modèles. À cet égard, nous avons conçu deux mécanismes de protection qui empêchent efficacement l'impact des Byzantins. Nous montrons également les inconvénients et les défis actuels de ces mécanismes. Nous espérons qu'avec les solutions proposées, cette thèse pourra ouvrir la voie à une méthodologie sécurisée de federated learning dans les travaux futurs.

Mots-clés : federated learning, robotique en essaim, byzantin, blockchain, smart contract.

Abstract

Federated learning is a new approach to distributed machine learning. It offers advantages such as keeping data locally and distributing computation. Specifically, federated learning means that each machine trains a model locally and then exchanges only the learned parameters. However, to aggregate the locally trained parameters, existing work mainly uses centralized servers. Although the approach shows great promise in multi-robot systems, it does not lend itself easily to swarm robotics systems which, inspired by collective behaviors in biological systems, aim to create decentralized systems where autonomous robots can self-organize and achieve objectives without a central authority. In this thesis, we present a proof-of-concept implementation of federated learning in a robot swarm that does not compromise decentralization. To do so, we use blockchain technology to enable our robot swarm to synchronize a shared model that is the aggregation of the individual models without relying on a central server. This implementation also protects the learning process from different Byzantine faults or attacks that may occur in decentralized networks. Our experiments are conducted in ARGoS, a physics-based simulator for swarm robotics, using the Ethereum blockchain protocol which is executed by each simulated robot. We show that introducing only a single Byzantine can heavily disrupt the training process. As such, we devise two protection mechanisms that effectively prevent the Byzantine's impact. We also show the drawbacks and current challenges of these mechanisms. We expect that with the solutions provided, this thesis can pave the way to a secure federated learning methodology in future works.

Keywords: federated learning, swarm robotics, byzantine, blockchain, smart contract.

Acknowledgments

I would like to express my sincere gratitude to Andreagiovanni Reina, Volker Strobel, and Alexandre Pacheco, without whom this thesis would have never been possible. Their invaluable support and guidance, both technically and personally, have been instrumental throughout this journey. I am truly thankful to them for their continuous assistance. I would also like to express my gratitude to Marco Dorigo and the IRIDIA group for allowing me to conduct my work in their laboratory.

Furthermore, I would like to extend my appreciation to my friends who accompanied me on this lengthy academic voyage in the field of engineering.

I am also deeply grateful to my family, especially my parents, for their unwavering love, comfort, and assistance whenever I needed it. Their constant support has been a source of strength throughout my endeavors. Additionally, I would like to thank my brother, Julien, and my sister, Émilie, for their encouragement and positive mindset, which helped me overcome challenges along the way. Lastly, I would like to express my heartfelt appreciation to Élodie, for her continuous encouragement and source of motivation to push the utmost out of me.

Acronyms

AI	Artificial Intelligence
ANN	Artificial Neural Network
DNN	Deep Neural Network
FL	Federated Learning
LSTM	Long Short-Term Memory
MAE	Mean-Absolute Error
ML	Machine Learning
MSE	Mean-Squared Error
PoA	Proof-of-Authority
PoS	Proof-of-Stake
PoW	Proof-of-Work
RaB	Range and Bearing
SC	Smart Contract
SGD	Stochastic Gradient Descent
SSE	Sum of the Squared Error
TCP	Transmission Control Protocol
TF	TensorFlow

Contents

List of Figures

Chapter 1

Introduction

Swarm robotics takes its roots from collective behavior in biological systems such as bee or ant colonies. As a group, they are capable of solving complex problems, such as finding safe nesting locations through individual decision-making and peer-to-peer interactions. Swarm robotics aims to create decentralized systems in which the robots in a swarm can self-organize and complete goals effectively. This means that there is no centralized authority deciding the robots' behavior nor infrastructure that allows for global communications. Rather, self-organization is an emerging property of the simple interactions between individuals and the environment. Additionally, most swarms consist of cooperative robots, similarly to how swarms of insects are cooperative by nature because they share the same goal. However, recent research has shown that the actions of damaged or malfunctioning robots may propagate errors that lead to undesired group behaviors. In more extreme cases, robots may be hacked and actively try to harm the swarm. Robots with these characteristics are referred to as Byzantine robots. Recent research by Strobel et al. [35] and Pacheco et al. [28, 27] introduces blockchain technology to swarm robotics in order to address these security issues.

In the field of Artificial Intelligence (AI), Federated Learning (FL) is a novel approach to distributed machine learning. It has the advantage of increasing data privacy by keeping data locally and sharing computation. In FL, each agent performs local training (for example, on a smartphone) on the data it has collected and shares the trained model with a centralized server that aggregates all the individual models. FL shares a similar foundation with swarm robotics, as both involve agents performing small tasks to achieve a common objective. However, FL still relies on a centralized server.

This thesis explores the implementation of federated learning in swarm robotics. Each robot of the swarm locally trains a model with the data they have collected. However, in contrast to classical FL, which uses a central server to aggregate the parameters of the local models, a decentralized and secure data structure will be utilized. This data structure is the Ethereum Blockchain, which is maintained by the robot swarm. The Ethereum Blockchain supports Smart Contracts (SC) which allows programs to be executed in a decentralized blockchain network and reach consensus on the outcome of the programs. Additionally, this thesis explores resilience to malfunctioning or malicious robots, specifically Byzantine robots that may send incorrect model parameters. Therefore, using a blockchain serves a dual purpose: as a distributed computation platform that enables the synchronization of an aggregated model; and as a means to implement security mecha-

nisms that address the risks posed by Byzantine robots for the model training.

In this thesis, we begin by providing an overview of the background and employed technologies in Chapter 2. Next, in Chapter 3, we present the experimental implementation, including the network architecture that we implemented, the weights transmitted by each Byzantine robot, and the protective measures that we designed. Subsequently, we analyze the results of each experiment and outline areas for future investigation in Chapter 4. Finally, we conclude this thesis with a discussion on the results and findings of our study, its implications, and potential future developments in Chapter 5.

1.1 Objectives

It has been shown that FL can be used by a robot swarm to perform decentralized and collective learning of a model for trajectory prediction [21]. Unfortunately, we show that this method is fragile to the introduction of a single Byzantine robot that sends randomized parameters (similar to the initialization parameters of this model).

Our objective is thus to present an implementation of the same federated learning mechanism as a smart contract on the blockchain. This migration is done because the blockchain mechanism has proven to be effective when dealing with Byzantine robots in swarm robotics [35, 27].

We then show a smart contract implementation that prevents the harm caused by the same Byzantine robots and enables the secure federated learning of the model in the presence of Byzantine robots.

1.2 Related Work

This thesis builds upon the paper *Flow-FL: Data-Driven Federated Learning for Spatio-Temporal Predictions in Multi-Robot Systems* [21]. We re-implement this paper and change the aggregation mechanism to be on the blockchain and the training to take place during the experiment rather than post-experiment. Since this thesis combines 3 fields: swarm robotics, federated learning and blockchain, we divide this section in 3 part. First we discuss the current implementation of federated learning in swarm robotics. Next, we present the ongoing work related to federated learning in blockchain. Finally, we end with a discussion over the implementation of blockchain in swarm robotics.

Federated learning saw its debut with McMahan et al. [22] at Google in 2016 to train the Gboard on smartphones [11] while keeping all the training data on the user's device, increasing data privacy. Then FL started to gain interest in the swarm robotics fields. Majcherczyk et al. [21] implemented FL for trajectory prediction in swarm robotics and decentralized the aggregation process in a driven-data approach which they call Flow-FL. Na et al. [23] proposed a federated reinforcement learning strategy for automatic controller design of robot swarms. Finally, Zhu et al. [49] explores the implementation of a decentralized deep reinforcement learning for swarms using the blockchain as the aggregation mechanism, which bears similarities to the approach we implement in this thesis.

Federated learning on the blockchain is also a novel approach. Wang and Hu [46] were among the first to explore this concept and conducted a comprehensive survey on the application of blockchain in federated learning. In their paper, they discussed multiple architectures, and the one this thesis implements is the fully coupled blockchain FL, where the agents of federated learning serve as the nodes of the blockchain. To incentivize participants to contribute, appropriate reward mechanisms have been proposed. Kim et al. [17] introduced BlockFL, a blockchain-based approach that rewards users proportionally to the number of samples used for training. However, this mechanism poses a threat where the value of samples may be exaggerated by malicious nodes. In this thesis, we explore the aspect of malicious nodes and implement a similar reward mechanism. Other reward mechanisms, such as the *Shapley value* method [33], exist to assess the contribution of federated learning participants fairly. However, calculating the Shapley value is time-consuming. Liu et al. [20] implemented the Shapley value method in their blockchain called FedCoin, a peer-to-peer payment system for federated learning that enables feasible profit distribution based on the Shapley value. In Fedcoin, blockchain consensus entities calculate the Shapley value and a new block is created based on the proof of Shapley (PoSap) protocol. Wang et al. [45] and Song et al. [33] also implemented their version of the Shapley value as a base for their reward mechanism. Lastly, Kang et al. [16] implemented another effective incentive mechanism. They proposed reputation as a metric to evaluate reliability and trustworthiness. They then effectively created an incentive mechanism combining reputation with contract theory to motivate high-reputation mobile devices with high-quality data to participate in model training.

Finally, blockchain was first introduced to swarm robotics by Castelló Ferrer [7]. He described a range of applications for blockchain in robot swarms, including secure communication, data logging, and consensus agreement. Later, as a first proof of concept, Strobel and Dorigo [36] designed a system to protect a swarm of robots from Byzantine where the task was to reach a consensus on the most present colored tile in an arena. This system used the Ethereum blockchain framework [6] and the robot swarm simulator ARGoS [29]. The same researcher then extended this work to determine the relative frequency of white tiles and presented the ARGoS-blockchain interface [35]. Lastly, Pacheco et al. [27] implemented the same collective decision task in a physical robot swarm. They used a mobile ad-hoc network protocol to share data between robots and switched the blockchain consensus algorithm to proof-of-authority [38] which is much less computationally intensive than the initially implemented proof-of-work consensus algorithm [24]. With these changes, they effectively demonstrated the viability of the Ethereum blockchain approach for the protection of physical robot swarm to Byzantines in real world. This thesis uses the ARGoS-blockchain interface with Ethereum and the proof-of-authority algorithm.

Chapter 2

Background

This thesis relies on three main technologies: Artificial Neural Networks, Blockchain, and Swarm Robotics. Each technology will be reviewed to describe what is the current state of research and the tools that have been used to conduct this thesis.

2.1 Artificial Neural Networks

Artificial neural networks (ANN) can be traced back to the field of artificial intelligence (AI) and the early research on modeling the human brain [32]. Today they are used as powerful machine learning tools, like GPT-4 which is a generative AI capable of generating text based on a user's queries (through text) [26]. The task of the neural network is to match as best as possible an input to an output. Let x be the input, y be the desired output, $M(\cdot)$ (for model) the neural network, and the d weights associated w. The goal of the model is defined as:

$$y = M(x; w) \tag{2.1}$$

Typically, we call the output of a model a *prediction*, and is defined as $\hat{y}$. *Training* a model is the equivalent of solving an optimization problem where the objective is to minimize a loss function $\mathcal{L}(w)$ by tuning the weights.

$$\underset{w \in \mathbb{R}^d}{min} \quad \mathcal{L}(w) \quad \text{with} \quad \mathcal{L}(w) = \frac{1}{N} \sum_{i=1}^{N} \ell(y_i, x_i; w) \tag{2.2}$$

Where $\ell(y_i, x_i; w)$ corresponds to the loss of the prediction for sample (x_i, y_i) with weights w. For this minimization to be effective, a large amount of samples (x_i, y_i) are needed, this is what we refer as the data set which, in the case of Eq. 2.2, is of size N. Many loss functions exist, such as the Mean Squared Error (MSE) which is used in regression ANN problems [41].

To further explain the principles behind ANN, this section is divided into four parts. The first two sections, feed-forward and back-propagation, aim to explain how the model is used and trained. Next, in Section 2.1.3, we introduce the federated learning (FL) algorithm. Finally, in Section 2.1.4, we present the Long-Short Term Memory (LSTM) layer, which is a different kind of layer than the classical dense (neurons) layer, as it is used in the model for trajectory prediction.

2.1.1 Feed-forward Mechanism

An ANN can be viewed as layers of neurons. Each neuron of a layer is connected to every neuron of the next layer with its associate weight [4]. Typically, an ANN will be composed of an input layer, a hidden layer, and an output layer (Figure 2.1). If there are multiple hidden layers, the model is usually referred to as a Deep Neural Network (DNN).

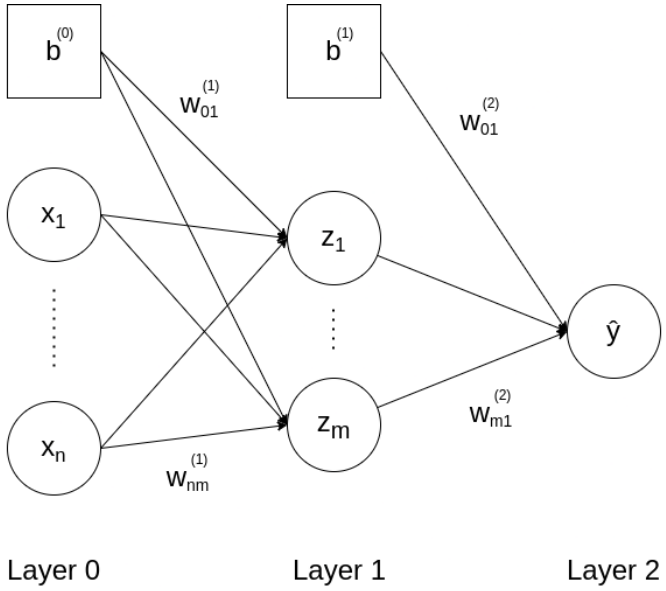


Figure 2.1: Feed-forward ANN with an input layer x, a hidden layer z and an output layer $\hat{y}$.

From Figure 2.1, let[1]:

- n the number of inputs.
- L the number of layers.
- $m^{(l)}$ the number of hidden neurons of the l^{th} layer $\in \{1, ..., L\}$.
- $w_{ij}^{(l)}$ the weight connecting the i^{th} neuron of layer $l-1$ to the j^{th} neuron of layer l.
- $z_j^{(l)}$ with $j \in \{1, ..., m^{(l)}\}$ the output of the j^{th} hidden neuron of the l^{th} layer.
- $b^{(l)}$ the bias of the l^{th} layer.
- Finally we denote $m^{(0)} = n$ and $z_j^{(0)} = x_j$ with $j \in \{1, ..., n\}$.

The output of neuron $z_j^{(l)}$ for $l \geq 1$ is composed of a linear and non-linear part (which is why we call ANN non-linear algorithms). Let's first consider the linear part of $z_j^{(l)}$ as $v_j^{(l)}$ which is given by the linear combination of the weights and the neurons of the previous layer:

$$v_j^{(l)} = \sum_{k=1}^{m^{(l-1)}} z_k^{(l-1)} \cdot w_{kj}^{(l)} + b^{(l-1)} \cdot w_{0j}^{(l)} \quad \text{for } j \in \{1, ..., m^{(l)}\} \tag{2.3}$$

[1]This section is inspired from Gianluca Bontempi's Handbook chapter 8.1.1 on Artificial Neural Network [4]

The non-linear part is called an *activation function*. Multiple activation functions exist, with the most popular ones being the sigmoid, ReLu (Rectified Linear unit), or hyperbolic tangent function [39]. For example, the ReLu function is defined as:

$$ReLu(x) = max(0, x) \tag{2.4}$$

Let us call $a^{(l)}(\cdot)$ the activation function of layer l, $z_j^{(l)}$ can be obtained by processing $v_j^{(l)}$ through the activation function:

$$z_j^{(l)} = a^{(l)}(v_j^{(l)}) \tag{2.5}$$

The output, for example of a 2 layers ANN, is thus associated with its input with the following equation:

$$\hat{y} = a^{(2)}(v_1^{(2)}) = a^{(2)}\left(\sum_{k=1}^{m^{(1)}} z_k^{(1)} \cdot w_{k1}^{(2)} + b^{(1)} \cdot w_{01}^{(2)}\right), \tag{2.6}$$

where

$$z_k^{(1)} = a^{(1)}\left(\sum_{j=1}^{n} x_j \cdot w_{jk}^{(1)} + b^{(0)} \cdot w_{0k}^{(1)}\right) \quad \text{for } k \in \{1, ..., m^{(1)}\}. \tag{2.7}$$

2.1.2 Back-propagation

The back-propagation algorithm is used to find the optimal weights. It is a gradient descent-based algorithm with the goal of minimizing the non-convex loss function [4]. In this thesis, the loss is computed as the Mean-Squared Error (MSE) over a dataset of N samples.

$$\mathcal{L}(w) = \frac{1}{N}\sum_{i=1}^{N}(y_i - M(x_i; w))^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i})^2 \tag{2.8}$$

The back-propagation computes the gradient of the loss function with respect to the weights and updates the weights using a learning rate η. The basic approach of gradient descent is called the iterative gradient descent[4] or more commonly Stochastic Gradient Descent (SGD) [37].

$$w(t+1) \leftarrow w(t) - \eta \frac{\partial \mathcal{L}(w(t))}{\partial w(t)}, \tag{2.9}$$

where $w(t)$ is the weight vector at iteration t.

More advanced gradient descent techniques have been later introduced such as the Root Mean Square Propagation[2] (RMSProp) [31] and Adam[3] [18]. Both RMSprop and Adam are improved optimization algorithms of SGD which have faster convergence speeds. Still, SGD will be used for this thesis as it is the foundation of the Federated Averaging Algorithm (see next section) which will be used and was also used in Flow-FL[4].

[2]RMSProp was first introduced as class note slides:
http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

[3]Adam is not an acronym but is derived from adaptive moment estimation. Adam adjusts the learning rate for each weight by utilizing estimations of the first and second moments of the gradient.[2]

[4]In their paper [21] they mention they used RMSprop, yet in their code SGD was the optimizer used, this is why we use SGD.

2.1.3 Federated Learning

Federated Learning (FL) is a machine learning technique that enables multiple devices to collaboratively train a model without the need to share data with a centralized server. It is an iterative algorithm that involves sharing the model between clients and server instead of clients sharing their data with the server, Figure 2.2.

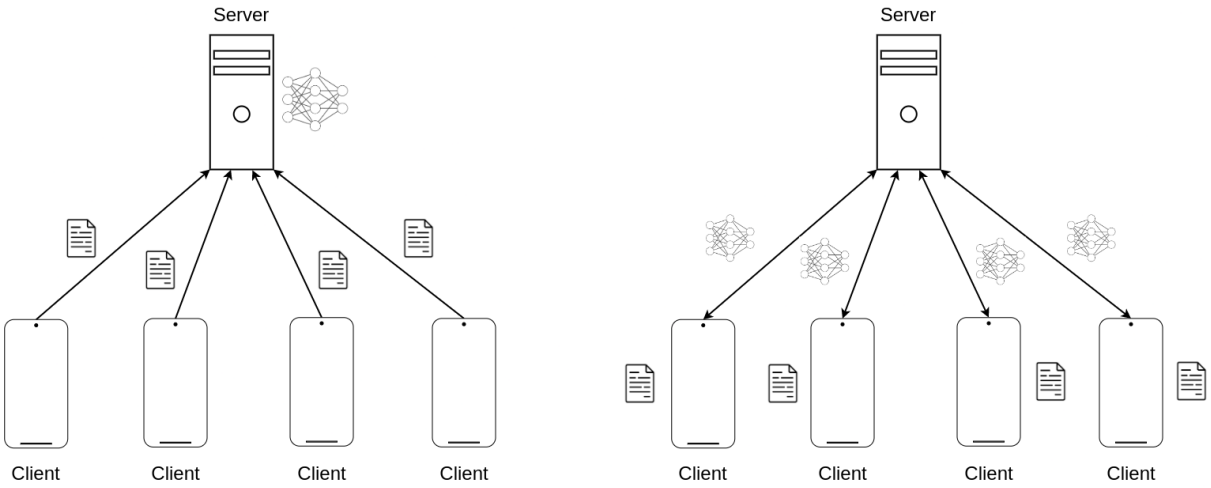


Figure 2.2: Classical (left) vs centralized federated (right) machine learning[5]. The clients are represented with phones, and the file logo represents the data of each client. The model is represented with the network of neurons next to the server on the left image and between clients and server on the right image. Lastly, the server is represented with a computer and is on the top of both images.

To set up FL, the server first decides on the architecture of the ANN/DNN it wants and initializes the weights. Weights can be initialized randomly [14] with different distributions, such as a normal distribution [40]. Once the server has its initial model it selects from the available clients the ones that will perform local training. Once locally trained the client sends back the model and the server aggregates them. This is called an aggregation round and this whole process (apart from setting the weights initially) is repeated iteratively until convergence of the model.

There are multiple aggregation mechanisms, such as FedAwo [48], FedProx [19], FedNova [44], FedDyn [1], or FedAdp [47]. However, in this thesis, we focus on Federated Averaging (or FedAvg) [22], a simple yet effective algorithm that works well with LSTM layers. The FedAvg algorithm is an optimization problem with a similar goal to the original ANN.

$$\min_{w \in \mathbb{R}^d} \quad \mathcal{L}(w) \quad \text{with} \quad \mathcal{L}(w) = \sum_{k=1}^{K} \frac{n_k}{N} \mathcal{L}_k(w), \tag{2.10}$$

where

$$\mathcal{L}_k(w) = \frac{1}{n_k} \sum_{i \in \mathcal{P}_k} \ell(y_i, x_i; w), \tag{2.11}$$

[5]We created this image using Alex Lenail model generator for the neural network https://alexlenail.me/NN-SVG/index.html and using the file logo from https://www.iconpacks.net/free-icon/file-1453.html.

and let:

- K the number of clients.
- C the fraction of selected clients.
- $\mathcal{P}_k$ the data set of client k
- n_k the data set size of client k.
- B the local mini-batch size.
- E the local epoch count.
- R the number of aggregation rounds.

In a scenario where $C = 1$ (so using all clients) with learning rate η, each client will compute their local training and update their weights

$$w^k(t+1) \leftarrow w^k(t) - \eta \frac{\partial \mathcal{L}_k(w^k(t))}{\partial w^k(t)}, \tag{2.12}$$

where $w^k(t)$ corresponds to the weights of robot k at time t. Before any update has been made locally the weights $w^k(t) = w(t)$. Also, if the number of batches is > 1 (due to $B < n_k$) or the number of epochs $E > 1$ then the weights are updated multiple times locally, and only at the last update are the weights assigned to $w^k(t+1)$. All $w^k(t+1)$ are then aggregated with a weighted mean based on the number of samples they have trained on:

$$w(t+1) \leftarrow \sum_{k=1}^{K} \frac{n_k}{N} w^k(t+1). \tag{2.13}$$

2.1.4 Long Short-Term Memory Layer

Long Short-Term Memory (LSTM) is an ANN layer that enables the learning of time-series data using a carousel mechanism. It was a response to the vanishing/exploding gradient issue of Recurrent Neural Networks with long time-series data that made the training of such ANN much longer [15]. In 2017, another improvement in time-series data processing for ANN was introduced by Vaswani et al. [43]: The Transformer. Transformers are powerful tools that made GPT[6]-4 possible. One issue of transformers is their computational and memory cost (they are heavier to run than LSTM). Since this thesis's training dataset input size is short (32 points long sequences for the input), transformers are unnecessary. Instead, we use LSTMs because they are the lighter option while still being capable of solving the ANN problem we present in this thesis.

LSTMs are made of 4 gates: the *forget* gate, the *input* gate, the *state candidate* gate and the *output* gate, Figure 2.3.

[6]GPT stands for Generative Pre-trained Transformer.

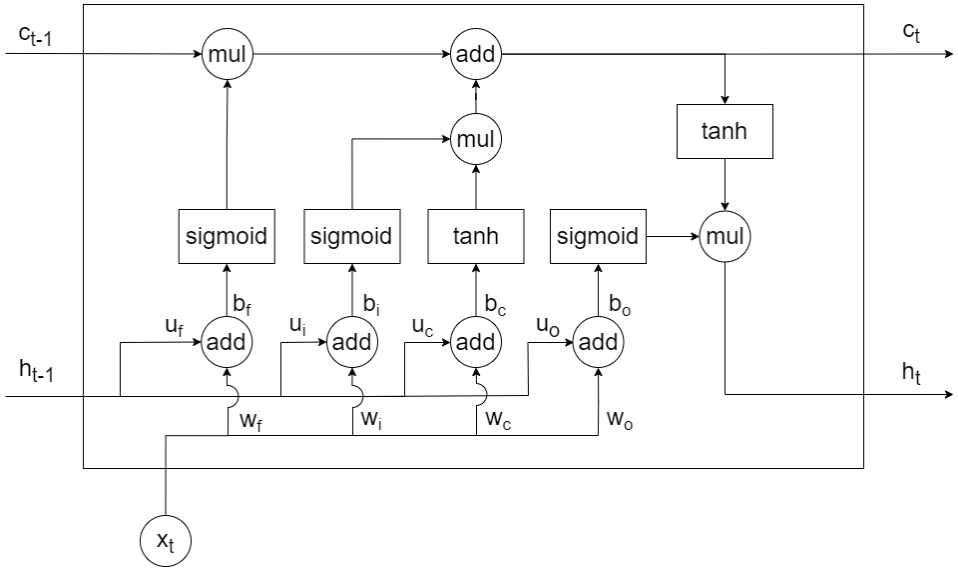


Figure 2.3: Representation of an LSTM with weights w, u, and bias b using the sigmoid and hyperbolic tangent (tanh) activation functions . The *add* symbol corresponds to a matrix addition and the *mul* symbol corresponds to a Hadamard product $\odot$ [34].

LSTMs have two main branches, here depicted with c_t and h_t, which respectively correspond to the long-term memory and the short-term memory [34]. They carry the information of the previous input (x_{t-1}) to the next (x_t) of a given sequence. h_t is also the final output of the LSTM when the whole sequence has been processed. To compute c_t and h_t, two different activation functions are used in this LSTM: the sigmoid $\sigma(x)$ and the hyperbolic tangent (tanh) $\tau(x)$,

$$\sigma(x) = \frac{1}{1 - e^{-x}}, \qquad \tau(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{2.14}$$

With these activation functions we can define the respective gates equations:

$$f_t = \sigma(w_f x_t + h_{t-1} u_f + bf) \tag{2.15}$$
$$i_t = \sigma(w_i x_t + h_{t-1} u_i + bi) \tag{2.16}$$
$$s_t = \tau(w_c x_c + h_{t-1} u_c + bc) \tag{2.17}$$
$$o_t = \sigma(w_o x_t + h_{t-1} u_o + bo) \tag{2.18}$$

Using these gate equations, c_t and h_t can be computed as

$$c_t = c_{t-1} \odot f_t + i_t \odot s_t \tag{2.19}$$
$$h_t = \tau(c_t) \odot o_t. \tag{2.20}$$

The weights of a LSTM layer are identical for all elements of the sequence. Only c_t and h_t are updated when getting the next input x_t of the sequence. In the same way as the neurons of Section 2.1.1, LSTMs weights are updated by applying the back-propagation algorithm of Section 2.1.2.

2.2 Blockchain Protocols

Blockchain technology made its debut in 2008 with the Bitcoin cryptocurrency created by Satoshi Nakamoto[7] [24] as a way to record digital transactions in a decentralized way. This means that, for the first time, it became possible to send money across the internet without the need for governmental or banking institutions to manage user accounts and funds. Instead, these transactions are recorded on a distributed public ledger that relies on trustless verification and consensus systems that ensure that double-spending funds is not feasible.

Ethereum [6] is another cryptocurrency and blockchain protocol which launched in 2014. It extends upon Bitcoin in the sense that the transactions that go on its ledger are not only monetary: when sending Ethereum transactions it is possible to execute code on a public *computing platform* known as the Ethereum Virtual Machine. The computer programs that users can interact with are known as Smart Contracts (SCs). In this thesis, we leverage Ethereum SCs to execute code synchronously in a network of robots.

Throughout the remainder of this section, we present the fundamentals of blockchain technology, starting with the Bitcoin blockchain as a monetary ledger, and then the Ethereum blockchain as a distributed computing platform.

2.2.1 Monetary Ledger

Bitcoin, in the first place, is a way of transferring a form of digital money known as a cryptocurrency. All transactions are stored in a ledger. A transaction is, for example, Alice sends Bob 5 Bitcoin. For a transaction to be accepted it needs to be digitally signed by Alice with her private key as money is being transferred from her account to another account. To prevent Bob from copy-pasting Alice's transaction multiple times a hash of the previous transaction (in this case Alice's transaction) is included in the creation of the signature, Figure 2.4. In Bitcoin and most blockchain frameworks, all the data is public and accessible by everyone using the blockchain.

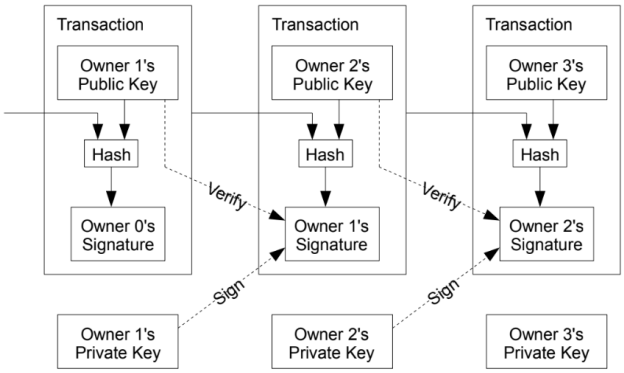


Figure 2.4: Transaction scheme of the Bitcoin Blockchain [24]. Looking at the second transaction, owner 2 would be Bob, and owner 1 would be Alice, as Alice is transferring Bitcoins to Bob. The transaction is digitally signed by using the hash of the previous transaction to or from Alice and the public key of the next owner (Bob). Anyone can then verify this signature using the public key of the previous owner (Alice).

[7]Satoshi Nakamoto is a pseudonym, and their real identity remains unknown to this day.

In the second place, Bitcoin is a decentralized system, which means that every participant keeps a copy of the blockchain. Then, for a transaction to be added to the blockchain it needs to be included in a new *block*, that will be appended to the blockchain. To achieve this, users broadcast their transactions to other participants which keep a local buffer of transactions known as a *mempool*.

A block is thus a collection of transactions that are collected from the mempool of the node that issued that block. New blocks are issued through a process called *mining*. The mining process is at the core of the consensus protocol that Bitcoin uses: *proof-of-work* (PoW). A consensus protocol is a mechanism used to enable multiple participants or nodes to agree on the state of a system. It ensures that all participants reach a consistent view of the shared data, even in the presence of failures, delays, or malicious actors. Indeed, as the blockchain is decentralized, participants have to choose and agree on a single version of the blockchain.

Proof-of-work

PoW is the consensus mechanism that enabled decentralized systems to prevent double-spending (a situation where the same Bitcoin would be used for 2 transactions). Mining consists in hashing the block with the hash of the previous block (which is what creates the *chain* of the blockchain) and a nonce[8], Figure 2.5. The goal is to change the nonce such that the hash begins with a certain number of zero bits.

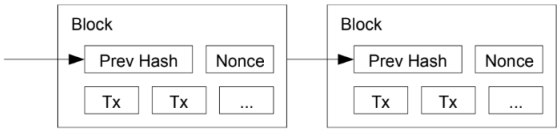


Figure 2.5: PoW mining mechanism of a new block [24].

Finding the nonce is exponentially harder as a function of the number of zeros required. While finding the nonce is hard, verifying it is quick and simple and it can be done by executing a hash function. Trying to cheat by changing the transaction inside a block would require re-mining an old block and this is a hard operation since finding a nonce before every other miner in the blockchain is highly unlikely, therefore cheating is discouraged. Miners are rewarded with Bitcoins and transaction fees[9] when their block is added to the blockchain. This is what motivates miners to mine as fast as possible and this requires high computing power. Finally, when there are conflicting blockchains from different nodes, the longer blockchain (meaning the one with the most proof of work) is the chosen blockchain. The other blockchain being discarded means that all the extra blocks that are not on the longest blockchain aren't taken into consideration anymore and miners lose all rewards they previously received, this is what also incentivizes miners to always mine on the longest blockchain.

[8] A nonce is a number used only once.

[9] The transaction fee is a small compensation the senders add to motivate miners to include their transactions.

2.2.2 Distributed Computing Platform

Ethereum is another blockchain that introduced its cryptocurrency, *Ether*, in 2014. It allows for the execution of Turing-complete programming code on the blockchain, enabling the use of SCs. These contracts enable participants to execute code on the blockchain and accept the outcome without requiring trust or supervision. One can interact with SCs in two ways: through *call* functions or *transact* functions. The call function is used to access functions that do not change the state of the smart contract and are therefore used for making queries or accessing information on the blockchain. On the other hand, transact functions are functions that will change the state of the blockchain, and thus require users to send a transaction and pay a certain fee depending on the complexity of the code to be executed. Similarly to regular transactions in Section 2.2.1, transactions that interact with SCs are first stored in the local mempool of the nodes, and once they are included in a block and appended to the blockchain, the state of the SC is updated definitely.

Proof-of-stake

Recently, in 2022, Ethereum changed its consensus algorithm and switched to *proof-of-stake* (PoS) [25]. PoS is a response to PoW's main flaw which is its high amount of power consumption required to mine blocks. Also, PoW gives more rewards to people with better equipment (having a higher hash rate[10]) as the higher the hash rate the higher the chances are of being the one to create the next block. This also led people to start working together, which is called a mining pool, where they combine their mining power and distribute the reward between them. This effectively had the effect of making PoW more centralized.

In PoS, the people validating the blocks are not called miners anymore but *validators* and validators *forge* new blocks[25]. PoS's point of view was that letting everyone try to solve the problem is wasteful, instead it uses an election process in which one node is selected to forge the next block. To become a validator, a node has to deposit a certain amount of cryptotokens (like Ether) in the network as stake. The size of the stake determines the probability of a validator being chosen to forge the next block. The chances increase linearly with the deposit amount[11]. If a node is selected to forge the next block, it has to ensure that the transactions it adds to the block are valid. Afterward, the node signs the block, adds it to its blockchain, and broadcasts it to the network. As a reward the node receives the fees that are associated with the transactions inside this block. On the other hand, if the validators approve fraudulent transactions, they will lose a part of their stake. Validators can be trusted to correctly validate transactions as long as their stake is higher than the sum of the rewarded transaction fees. In the end, this is a financial motivation for good validators and discourages fraudulent behavior as they will lose more than they gain. When a node stops being a validator, its stake plus all the transaction fees that it received will be released after some time.

[10]The hash rate is the hashing speed of a system, i.e., the number of hashes a system can compute per second.

[11]Here an analogy could be made with PoW as having more powerful equipment (thus more expensive) leads to having higher chances of finding the solution. Effectively making the rich more favorable to be rewarded.

PoS is thus a solution that is more decentralized than PoW (due to its mining pools) and also encourages more people of being nodes of the network as it is much less expensive when compared to PoW (which requires expensive equipment) making it even more decentralized.

PoS is not a perfect solution either. It has issues with the fact that if an individual reaches 51% of the total amount of stake of the network he can effectively control the blockchain[12]. Additionally, the validator election cannot be perfectly random[30], as it would create a snowball effect where the richest validators would become richer and gain more chances of being elected, perpetuating the imbalance. Thus, PoS also brings its own risk compared to PoW.

Proof-of-authority

The framework used for this thesis implements a different consensus algorithm with the Ethereum blockchain: *proof-of-authority* (PoA) [38]. PoA proved itself as a good consensus mechanism for swarm robotics using blockchain technology [27].

In PoA, the validators are called *sealers* and are the nodes that can create blocks with their signature and add them to the blockchain. Sealers are pre-approved nodes by the network administrator, and the election, similar to the PoS consensus election mechanism, is biased towards a preferred sealer selected in a round-robin[13] manner. When the preferred sealer signs the block, it is called an *in-turn* signature, and if another sealer signs it then it is an *out-of-turn* signature [10].

For a block to be valid, certain conditions have to be met, notably:

- The timestamp of the new block must be at least t seconds, after the previous one. In this thesis, the time t is set to t=10 s and is explained in Chapter 3.
- The sealer can only sign one block in $\lfloor \frac{N}{2} \rfloor + 1$ where N is the number of sealers. This is to ensure the 51% majority rule.
- The sealer must correctly sign the block with its private key and the hash of the previous block.

Finally, the block is broadcasted through all the nodes which can then verify the block. As it is the case with previous consensus protocols, the chain with the highest *sealing priority* is the one the nodes agree on. In the case of PoA, blocks signed with in-turn signatures have a sealing priority of 2, whereas out-of-turn signatures have a sealing priority of 1.

PoA is able to scale easily, as it can either keep a core of trusted sealers or add and remove some based on a majority vote. Nonetheless, PoA is more inclined to private

[12]It is called the 51% rule and this is still quite unlikely for Bitcoin or Ethereum because it would require to have billions worth of dollars/euro in mining hardware or Ether, respectively, on the blockchain.

[13]Round robin is a method of sequentially selecting all elements of a group in a fair and systematic manner, where each element is chosen in turn and the selection process starts over again from the beginning of the list once all elements have been chosen. This order is typically rational and consistent, such as starting from the top of the list and proceeding downwards.

networks, in the sense that only allowed nodes can seal blocks. But this is not an issue in the case of this thesis as it is desired that the network is private and that only selected robots be sealers.

2.3 Swarm Robotics

Swarm robotics is the field that designs groups of robots following the principles of swarm intelligence [9, 12]. Swarm intelligence is a research field that focuses on the collective behaviors exhibited by decentralized and self-organized systems composed of multiple agents [3, 8]. Swarm intelligence draws inspiration from the behavior of social insects, like ants and bees, which do not have a central authority and, instead, focus on the resulting collective behavior originating from local interactions between the individuals themselves and the environment.

Similarly, in robot swarms, collective behaviors emerge from the local communication among robots, and from the interactions between robots and their environment. In particular, robot swarms act in an autonomous and self-organized way.

Indeed, one characteristic of robot swarms is the concept of localized interaction, wherein each robot has a finite range of communication and perception. Consequently, at any given moment, each robot directly interacts exclusively with its neighboring robots. The primary implication of this is that individual robots remain unaware of the overall size of the swarm and are unaffected by it.

A robot swarm can consist only of robots with identical hardware and control software. This is known as a homogeneous swarm. Alternatively, a robot swarm can include robots with different hardware and/or control software from different classes. This is known as a heterogeneous swarm. Both homogeneous and heterogeneous swarms share a common characteristic: they exhibit a high level of redundancy. Within these swarms, multiple robots possess the capability to perform each individual action necessary to fulfill the given mission. Essentially, no single robot is essential or irreplaceable.

The aforementioned characteristics (i.e., locality and redundancy) are highly valued due to their recognized ability to enhance resiliency, scalability, and flexibility [12].

Firstly, with a high redundancy and the lack of a single point of failure (because no robot is irreplaceable), robot swarms are resilient. This promotes a system that is robust to the failures of individual robots.

Moreover, the locality of interaction within the swarm plays a key role in achieving scalability. Due to this locality, robots in the swarm are unaware of the swarm's size. Given that the density does not vary significantly, adding or removing robots will not qualitatively alter the behavior of the swarm. This makes the swarm scalable as it is not required to modify the behavior of individual robots when there are more or fewer robots.

Finally, the ability to perform parallel execution (each robot executing the same code) enables the swarm of robots to react to contingencies, modification of the environment,

and changes in the working conditions. This flexibility enables the swarm to dynamically respond to evolving situations.

Chapter 3

Methodology

The federated learning scenario using the Ethereum blockchain is implemented in ARGoS [29], a physics-based simulator tailored for research in swarm robotics. The interface between Ethereum and ARGoS was originally employed for studying blockchain-based robot swarms[1][28, 35], and the authors also extended the framework to real-robots [27].

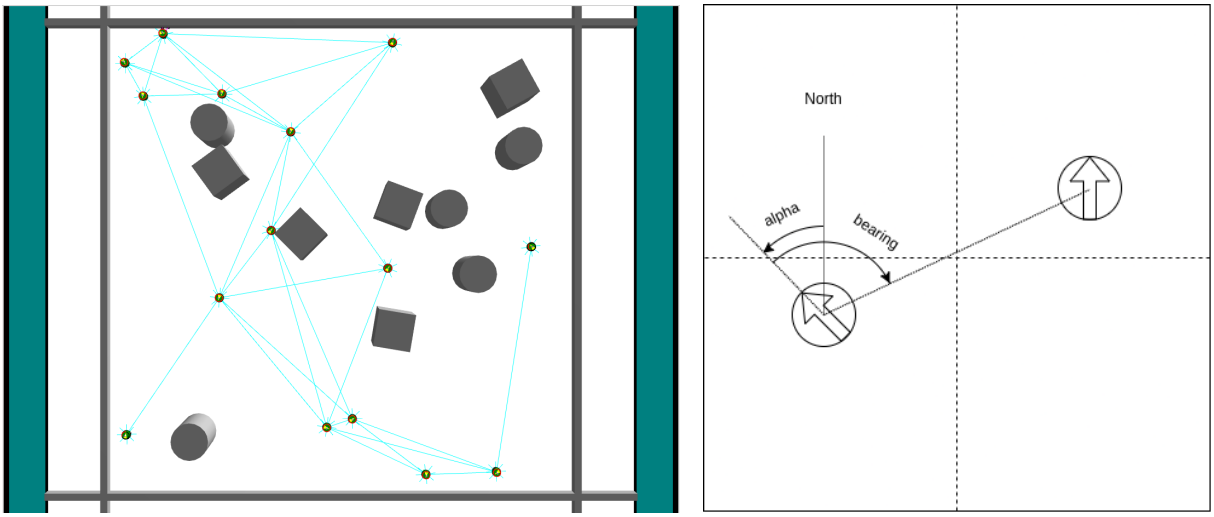
In this thesis, the mission of the robots is to utilize federated learning to train a neural network that can predict the trajectories the robots will follow, based on previously recorded positions within a 2D space filled with obstacles. The process involves the robots keeping their data and collectively training a shared model, which is subsequently shared back on the blockchain for aggregation. We present in this chapter how we implemented this mission.

We start by presenting the simulation environment in Section 3.1. In the same section, we describe the robot controller and the data collection methodology. In Section 3.2, we present the neural network architecture and the implementation of federated learning on the blockchain. Finally, in Section 3.3 we explain the details behind the security layers and the different weights the Byzantines send in each experiment.

3.1 Simulation Environment

The arena is a 2D closed square of dimension $5000 \times 5000\,mm^2$ populated with 15 robots. The experiment has a time limit of 50 000 steps, with 10 steps executed per second (effectively making the simulations 5000 s long). The arena is uniformly filled with 5 cylinders of radius $150\,mm$ and height $500\,mm$, and 5 boxes in the form of rectangular parallelepipeds with a base size of $300 \times 300\,mm^2$ and a height of $500\,mm$. A visual representation of the arena is shown in Figure 3.1a.

[1]The source code of this thesis is available at: https://github.com/teksander/geth-argos



(a) Experimental arena **(b)** Compass and bearing (not to scale)

Figure 3.1: Experimental arena in which robots perform obstacle avoidance (left) & Robot's compass and bearing representation (right). Alpha is the angle between the robot's orientation and the North, and the bearing the is angle between the robot's orientation and the position of the neighboring robot.

The Robots we use are e-pucks [13] that are augmented with a range-and-bearing module and a GPS. E-pucks have a diameter $d = 75\,mm$ and move at a speed of $2.5\,mm{\cdot}step^{-1}$, which is the distance moved per simulation step, in the arena. The range of communication between robots is $r = 2500\,mm$ and the bearing $\beta \in [-\pi, \pi[$ is the angle between the direction of the robot and the relative position of a neighboring robot, see Figure 3.1b. Robots have access to GPS coordinates, meaning that they know their (x,y) position at any given time where $x, y \in [-2500, 2500]$. They also have access to a compass where the North is at the top of the arena. The angle $\alpha \in [-\pi, \pi[$ measures the orientation of the robots around the z-axis, where the positive α values are on the left side (i.e. anti-clockwise direction starting from the North) of Figure 3.1b. In this context, an angle of 0 represents the North[2].

Knowing α, β, r, d and, the position of a robot, which we call here (x_r, y_r) for *reference* robot, we can compute the estimated position $(\hat{x}, \hat{y})$ of a neighbor as:

$$\begin{cases} \hat{x} = x_r + (r + d) \cdot cos(\alpha + \beta) \\ \hat{y} = y_r + (r + d) \cdot sin(\alpha + \beta) \end{cases} \tag{3.1}$$

3.1.1 Robot Controller

During the experiments, each robot continuously performs obstacle avoidance and collects data (a detailed explanation on the data collection methodology is in Section 3.1.2). In parallel, every 100 s, robots will enter the training state. In the training state, they first call the blockchain to access the latest weights (see Section 3.2 for a detailed explanation

[2]All these parameters are the same as the one used in the Flow-FL paper [21] to enable comparison of the results. However, the original article used Khepera robots. Our choice to use E-pucks is motivated by the fact that E-pucks have been equipped with a Raspberry Pi computer and have been shown to be capable to run blockchain software [27], thus facilitating a future extension of this work to a swarm of physical robots.

of the training and model description). They then train a neural network with these weights and finally submit the weights back on the blockchain.

Obstacle avoidance

At every time step, robots move in the arena. To move, we either increase or decrease the speed of each wheels. The e-puck has 2 wheels, one on its left and one on its right. The E-puck also has 8 infra-red proximity sensors as shown in Figure 3.2.

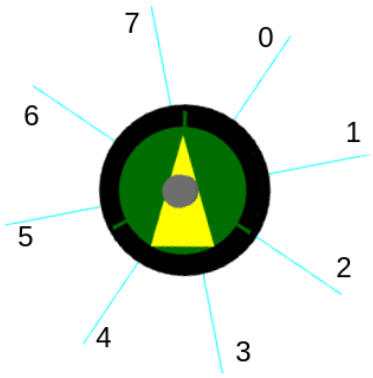


Figure 3.2: E-puck robot in ARGoS simulator with its 8 infra-red proximity sensors.

The sensors with indexes 0, 1, 6, and 7 are utilized in constructing our obstacle avoidance algorithm, which is why the weights assigned to these indexes in the algorithm are $\{-10, 10\}$. The sensor returns a value $\in [0, 1]$. A value of 0 indicates the absence of an obstacle in front of the sensor, while a value between 0 and 1 represents the proximity between the robot and the obstacle. Based on this information, we employ the following obstacle avoidance function.

Algorithm 1 ***avoid*** with *left* and *right* respectively the speed of the left and right wheel of the robot, *speed* a constant equal to 2.5, and thresh_ir a threshold value equal to 0.2.

```
obstacle, avoid_left, avoid_right ← 0
readings = epuck_proximity.get_readings()        ▷ The infra-red sensor values
weights = [−10, −10, 0, 0, 0, 0, 10, 10]
for i = 0 to 7 do
    if reading > thresh_ir then
        obstacle = True
        avoid_left += weights[i] · readings[i]
        avoid_right -= weights[i] · readings[i]
if obstacle then
    left = speed/2 + avoid_left
    right = speed/2 + avoid_right
return left, right
```

Finally, before we set the speed values for each wheel, the values are saturated so as to never exceed 2.5 speed per wheel.

Algorithm 2 ***saturate*** with *left* and *right* respectively the speed of the left and right wheel of the robot and *speed* a constant equal to 2.5.

$m = max(|left|, |right|)$
if $m > speed$ **then**
  $left = \frac{left \cdot speed}{m}$
  $right = \frac{right \cdot speed}{m}$
return $left$, $right$

Interaction with the blockchain

The implementation of the blockchain for each robot is made possible through the use of docker containers, each running *ethereum/client-go:v1.10.2*. Each robot has thus its own docker container associated. Originally, we wanted each docker container to have its own instance of TensorFlow[3] (TF). Unfortunately, due to difficulties in creating a docker image that includes both TensorFlow and Go-Ethereum simultaneously, a separate server has been built with the sole purpose of running TensorFlow to train the weights. This means that when robots want to train their NN, they connect to the TensorFlow server via TCP[4] connection and send their weights and their ID. This way, the server can access the data of the robot contacting it. This is thus a *central* server for training that was deployed for technical reasons, but if an actual implementation for real robots would take place, each robot would have its own TensorFlow instance.

Since training takes less than 4s (this value comes from the test with 1250 s expiration time) and a maximum of 15 robots perform training, the queue will never exceed the 100 s. This queue is also mitigated through time as robots wait 100 seconds after training before recontacting the server, this effectively spreads the robots in the 100 s interval of training. A visual representation of the training sequence is in Figure 3.3.

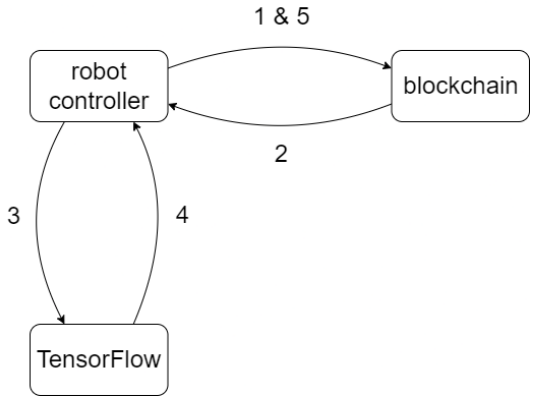


Figure 3.3: Training sequence of actions of a robot in the docker container.

In Figure 3.3 is the sequence of action corresponding to:

1. Robot sends a query to the blockchain to access the latest weights.
2. The blockchain sends the latest aggregated weights.

[3]TensorFlow is the library used in Python to train the neural network in this thesis.
[4]TCP is a communication standard used for communication over the network.

3. The robot uses TensorFlow to train its data with its weights.

4. TensorFlow returns the trained weights.

5. The robot submits its weights on the blockchain.

When it comes to the block sealing period on the blockchain, it is set to 10 seconds. This means that transactions are sealed every 10 seconds. We settled on this value as a compromise between a too-short or too-long sealing time. On one hand, if the sealing time was shorter, there would be numerous blocks mined without any transactions included. On the other hand, if the sealing time was longer, many transactions and even aggregation rounds would have to wait to be sealed, which could potentially delay the aggregation process.

3.1.2 Dataset creation

A trajectory is a combination of 100 positions over time (thus trajectories of 10 s) and each position is computed using Eq. 3.1. If the connection with a neighboring robot is interrupted during the recording, the trajectory is discarded. When robots collect these trajectories, they store them into *.csv* files. This data collection methodology originates from the Flow-FL paper [21] and has been translated from C++ to Python. A robot stores its trajectories every time it collects 10 complete trajectories to minimize file opening and closing.

In this implementation, we made the decision to assign an expiration time to the data, which is a tuning parameter determined in the first experiment in Section 4.1. This choice was made to emulate a realistic scenario where it would not be conceivable to, for example, store ever-increasing amounts of data on a robot.

To reduce the data loading process when training the neural networks, trajectories are saved in different files every 50 s. Specifically, the first 50 s of trajectories are stored in the file *traj1.csv*, the next 50 s are stored in *traj2.csv*, and so on. The last file being *traj100.csv* since each experiment takes 5000 seconds. This way, when we say that data has a 750 seconds expiration time, it means that only the 15 last trajectory files are opened. This prevents from processing large amounts of data that would not be used.

When the data of the trajectory files are loaded they are split into an 80% training set and 20% validation set.

3.2 Federated Learning & Blockchain

3.2.1 Neural Network Architecture

The neural network architecture is the same one as the one in the Flow-FL paper [21] and is a sequential model composed of:

- First layer: 16 LSTM with input size 32x2
- Second layer: Dropout of 20%

- Third layer: Dense layer of 96 (48x2) neurons.
- Final layer: Reshape layer that sets the output with shape 48x2.

As a result, there are a total of 2848 parameters in the model. Additionally, with an input size of 32x2, it corresponds to an input duration of 3.2 seconds. Similarly, the output size of 48x2 corresponds to a predicted trajectory output duration of 4.8 seconds. Consequently, we use only 8 s out of the recorded 10 s duration. This has been kept as is to compare the results with the Flow-FL paper [21]. Additionally, the optimizer used is SGD as discussed in section 2.1.3 and the loss function is the MSE.

3.2.2 Federated Learning implementation on Blockchain

The SC we use in this thesis is in the Solidity language and runs on version ^0.8.0. On the blockchain, there is a *gas* price associated with computational operations performed within the smart contract. This gas price is proportional to the amount of computation executed. Additionally, there are typically limits imposed on gas usage to prevent excessive computational load on the blockchain. However, in this thesis, we do not delve into this aspect, and all limits have been removed to facilitate aggregations within smart contracts.

The SC provides two functions to the robots (excluding the monitoring function): *submitWeights* for sending trained weights and *getWeights* for retrieving the currently aggregated weights. At the start of each experiment, initial weights are set on the blockchain. These initial weights are obtained through random initialization performed by TensorFlow for our specific model. It's important to note that the Solidity language does not support floating-point numbers. To address this limitation, we multiply the weights by 10^9 when sending them to the blockchain and then divide them by 10^9 when retrieving the weights from the blockchain.

When the weights are submitted on the blockchain, to reduce computation, a rolling average is performed. This means that only 2 lists are stored on the blockchain: *currentWeights* and *nextWeights*. Effectively they are assigned as follows:

$$nextWeights_i \mathrel{+}= n_k \cdot w_{k,i} \quad \forall i \in \{1, 2, ..., 2848\}, \tag{3.2}$$

$$currentWeights_i = \frac{nextWeights_i}{N} \quad \forall i \in \{1, 2, ..., 2848\}, \tag{3.3}$$

where n_k is the number of samples the robot k has trained the weights on and w_k is the weights of robot k after training them and N is the total number of samples (so the sum of the n_k) used in an aggregation round.

The number of participants (the quorum size) shouldn't be too high as indicated by the paper on FedAvg [22]. Thus the number of participants has been set to 50% of the number of robots, which comes to 7.

Additionally, to minimize memory usage, the weights are constrained to the *int48* data type. This decision is based on the fact that weights always remain within the interval $\in [-1, 1]$ which is then scaled to $\in [-10^9, 10^9]$. Also, in the case of a 1250 s

expiration time (as shown in Section 4.1, Figure 4.2), the average number of samples is approximately 330. Since the rolling average involves the summation of the product between the number of samples used and the weight, the theoretical limit is equal to $10^9 \cdot 330 \cdot 7 = 2.31 \cdot 10^{12} < 2^{47} = 1.41 \cdot 10^{14}$.

Moreover, considering that we have 2848 weights of size *int48*, the weight list occupies $2848 \cdot 48 = 133.5$ kilobits (Kb), or 16.7 kilobytes (KB). Therefore, whenever the robots send their weights, the blockchain memory increases by approximately 16.7 KB.

Both the batch size and epoch count are set to 20, following the parameters used in the Flow-FL paper[5]. Finally, the learning rate has been kept at its default value of 0.001 and the number R of aggregation rounds is not set as we try to do as much as possible in the 5000 s available per experience.

In summary, the FedAvg [22] has been adapted on the blockchain to be:

Algorithm 3 ***FedAvg*** with $K = 15$ total clients; $C = 50\%$ fraction of clients participating; $B = 20$ local mini-batch size; $E = 20$ local epochs; R aggregation rounds and learning rate $\eta = 0.001$.

Blockchain executes:

initialize $w(0)$
for round $t = 1$ to R **do**
  $p \leftarrow max(C \cdot K, 1)$
  $S_t \leftarrow$ (random set of p clients)
  $N \leftarrow 0$
  for client $k \in S_t$ **do**
    $n_k \leftarrow$ (number of samples of robot k)
    $N \leftarrow N + n_k$
    $w^k(t+1) \leftarrow \text{RobotTrain}(k, w(t))$
  $w(t+1) \leftarrow \sum_{k \in S_t} \frac{n_k}{N} \cdot w^k(t+1)$

RobotTrain(k, w)**:**

$\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ in batches of size B)
for epoch $e = 1$ to E **do**
  for batch $b \in \mathcal{B}$ **do**
    $w \leftarrow w - \eta \nabla \ell(b; w)$
return w

3.3 Byzantines Behaviors and Security Layers

We implement two smart contracts. A basic one that has no security against Byzantine robots, i.e., it functions only as a distributed database and model aggregator. We use this smart contract to test the convergence of the model and to aid in the choice of appropriate expiration time. And a more advanced one with two security layers that require robots

[5]In the paper they mention they do only one epoch but when looking at their code they were replicating the data 20 times for their epoch which is the equivalent of 20 epochs.

to make a transaction of ether when submitting their weights. The transaction fee is set to 5 ether and the number of ether each robot starts with is 21 ether. In the case of the robot's weights being completely discarded (due to the first security layer), this effectively lets them lose the equivalent of four submissions before not being able to submit anymore (the extra ether is to take care of the gas fee when executing the SC). For the other robots, this lets them lose and gain money without making them be instantly discarded the moment their weights are considered bad (this refers to the second security layer).

3.3.1 First and Second Security Layer

The first security layer is based on a threshold. Every time a robot submits its weights on the blockchain, they are compared with the previous aggregated ones. The comparison is a distance function, The Mean-Absolute Error (MAE), which sums the distance of each weight respectively.

$$\text{MAE}(w_s, w_a) = \frac{1}{N}\sum_{i=1}^{N}|w_{s,i} - w_{a,i}| \tag{3.4}$$

In Eq. 3.4, w_s are the submitted weights, w_a are the previously aggregated weights, and N is the total number of trainable weights (in this case 2848). During training, the peak MAE robots achieved on average was 10^7 (due to the multiplication by 10^9). The threshold above which weights are not taken into account is thus set to five times this average. This means that if a robot sends weights that generate a MAE greater than $5 \cdot 10^7$, it loses the money it submitted.

The second security layer consists of a ranking function. Since the aggregation process requires seven robots, there is seven ranks. The robots are ranked based on their distance (measured using MAE) from the previously aggregated weights. The robot with the distance that corresponds to the median of all the distances of the seven robots is ranked first. Then the following ranks are based on the absolute difference between a robot's distance and the ranked first robot's distance. Robots with a distance similar to the median are ranked high, whilst robots with a distance much different (leading to a high absolute difference) are poorly ranked.

For example, let's assume the seven robots have these MAE: $[33, 54, 22, 19, 44, 77, 61]$. The median value of this list is 44, thus the fifth robot would be ranked first. Hence, the current ranking is $[\_, \_, \_, \_, 0, \_, \_]$ where we put 0 for the robot ranked first, and 6 for the robot ranked last. Then we compute the absolute difference between the robot's MAE and the MAE of the robot ranked first. This gives the following list: $[11, 10, 22, 25, 0, 23, 15]$. We then rank the robots based on these distances, with the shorter having a good rank, and the higher having a bad rank. The final ranks are thus: $[2, 1, 4, 6, 0, 5, 3]$.

Each rank has its own weight corresponding to a higher or lower reward. The weights are of 2 categories, positive and negative. The positive weights are for the first (in ranking) n robots, which are positively rewarded, and the last robots (in ranking) have negative weights, which indicates a penalty in their reward. To compute the reward/penalty both the weights and the number of samples are taken into account. The number of samples is also considered since they also take part in the aggregation. The idea of this combination

came from four thoughts:

1. High ranking with a high number of samples should lead to a good reward.
2. High ranking with a low number of samples should lead to a reward.
3. Low ranking with a low number of samples should be penalized.
4. Low ranking with a high number of samples should be severely penalized.

The rank weights for this thesis are $[1, 1, 1, 1, 1, -1, -1]$, consisting of 5 equal positive weights, and 2 equal negative weights. This thus rewards the first robots and penalizes the 2 last robots. Note that if there are only good robots that send weights, they should have an equal probability for any rank since they send on average 200 samples. This leads good robots to eventually cancel the penalty with the rewards.

The sum of the reward is equal to the sum of the penalties which is equal to the submission price: 5 ether. Finally, each robot is rewarded/penalized a fraction of this reward/penalty. This fraction is computed as follows:

First, we compute the product between the robot's rank weight (rw) and the number of samples (s) it has trained on:

$$w_i = rw_i \cdot s_i \quad \forall i \in \{1, 2, ..., 7\}. \tag{3.5}$$

Then the positive weights are normalized between them only, and the same goes for the negative ones:

$$w_{f,i} = \frac{w_i}{\sum_{i \in \mathcal{P}} w_i} \quad \forall i \in \mathcal{P}, \tag{3.6}$$

$$w_{f,j} = \frac{w_j}{|\sum_{j \in \mathcal{N}} w_j|} \quad \forall j \in \mathcal{N}, \tag{3.7}$$

where $\mathcal{P}$ and $\mathcal{N}$ represent the set of positive and negative weights respectively, and w_f is the final weight. The amount of ether robot i is getting back is then:

$$m_i = (1 + w_{f,i}) \cdot 5\, ether \quad \forall i \in \{1, 2, ..., 7\}. \tag{3.8}$$

Since the price of submitting weights is 5 ether, this effectively means that positively ranked robots get $w_f \cdot 5$ ether in addition to the submission price (they thus make a profit). On the other hand, negatively ranked robots lose $w_f \cdot 5$ ether from their submission price.

3.3.2 Three Byzantine Behaviors

In experiments two to five, we introduce Byzantine robots. There are 3 types of Byzantines. The first one is considered to be faulty and sends random weights using the *random.randint()* function provided by Python, which generates random numbers following a uniform distribution. It has uniformly distributed weights in the interval $[-0.5, 0.5]$ which are then multiplied by 10^9 for the blockchain. We use this Byzantine robot in experiments 2 and 3, respectively in Section 4.2 and 4.3.

The second Byzantine robot is considered to be a malicious robot (for example, because it has been hacked) that sends as trained weights the currently aggregated one. This is to slow down the training and is in response to the first security that prevents vastly different weights from being accepted. We use this Byzantine robot for the fourth experiment in Section 4.4.

Finally, the third Byzantine robot is a *smart* hacked robot. Its objective is to always be ranked first, which would make it rich and the other robots poor. This is to try and bypass the second security layer. To be ranked first, it needs to know what is the median MAE. To estimate it, it tracks over time what are the current aggregated weights $w(t)$ and what are the previous aggregated weights $w(t-1)$. Then it computes the MAE between the two lists of weights. This MAE is an indicator of how much it needs to change the current weights to try to be ranked first. The weights the Byzantine robot sends are thus:

$$w_i^k(t+1) = w_i(t) + r \cdot MAE \cdot 2 \tag{3.9}$$

Where r is a uniformly distributed random value between 0 and 1 which is regenerated for every weight $w_i(t)$. Since it is uniformly distributed, it is multiplied by 2 such that on average r will be equal to 1, which effectively generates weights with a bias that have the same MAE. This bias will harm the network and at the same time make the robot richer. This Byzantine robot is used in the fifth experiment in Section 4.5.

Chapter 4

Results & Discussion

This chapter presents the results of five experiments that we conducted. The first experiment, discussed in Section 4.1, was conducted as a means to establish the data quantity that is suitable for federated learning on the blockchain. The second experiment, in Section 4.2, we introduce a faulty Byzantine robot (that sends random weights), but is, however, capable to impede collective learning. The third experiment, in Section 4.3, presents the results of our first security layer, which is shown to be capable of managing faulty Byzantine robots. Section 4.4 is the fourth experiment and presents the results of our second security layer, and how it fares against a malicious type of Byzantine robot (that uses the previous aggregated weights as its trained weights). And the fifth experiment, in Section 4.5 presents the results of a smart Byzantine robot (see 3.3.2) that manages to bypass both security layers and take the money of honest robots. Finally, a discussion of all the experiments is done in Section 4.6. Note that all the results discuss a loss (or average loss), and this loss specifically refers to the validation loss.

4.1 Data Quantity

The first experiment consists of 15 well-functioning robots performing obstacle avoidance in the arena. This experiment will thus verify if we managed to apply federated learning on the blockchain using a smart contract. Also, since all the robots are collaborative, the only variation in each configuration will be the amount of data used for training. The objective is to determine the *data expiration time* (i.e., the minimum amount of data) that is needed for optimal convergence speed and final results.

This is important because it can be expensive to store large amounts of data, as robots may have limited storage capabilities. Additionally, training data can become outdated in dynamic scenarios. Similar reasoning would also be valid in different scenarios, for example, in federated learning deployed on people's smartphones, it would not be tolerable to store all the data and take up a considerable amount of the phone's memory for the sake of training. Therefore, our solution is to discard data older than a given expiration time. This experiment is designed to measure how the expiration time influences the collective learning performance on what is a suitable value for our next experiments.

For this experiment, we executed 5 runs[1] for each configuration of data expiration

[1]Only 5 runs are performed per configuration as we only want to select the data expiration time to fix a parameter for the following experiments.

time, which ranges from 250 s to 1250 s.

Figure 4.1 shows that as the data expiration times increase, the number of aggregation rounds increases. This is potentially due to the blockchain receiving too many transactions per second (or the block sealing period of 10 s being too high) with a small data expiration time where the training on the server is shorter (taking maximum 2 seconds). Also one could have expected the number of aggregations to reach:

$$aggregations = \frac{15}{7} \cdot \frac{5000}{100} = 107 \tag{4.1}$$

As there are a total of $\frac{5000}{100}$ training periods, 15 robots in the arena, and 7 robots required for aggregation. However, this theoretical limit is hard to reach as communication limitations and blockchain conflicts lead to fewer aggregations taking place in the end.

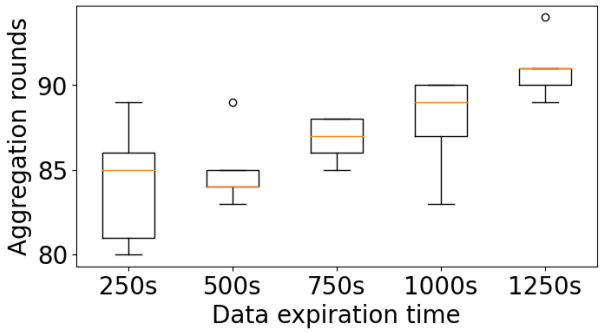


Figure 4.1: 1^{st} experiment's aggregation rounds for each data expiration time.

Figure 4.2, presents the number of samples (the number of trajectories) each robot uses when performing local training at given aggregation rounds. We can see that higher expiration times lead to a higher number of samples. For example, with 250 s the number of samples available for training averages 65, and for 750 s it averages 200. This also indicates that the number of samples available increases linearly with the expiration time, 3 times more expiration time leads to 3 times more data for training. We can also see that a certain delay is required to reach the peak number of samples for a given expiration time. This is because robots always start with zero data, and have to create their dataset along the experiment.

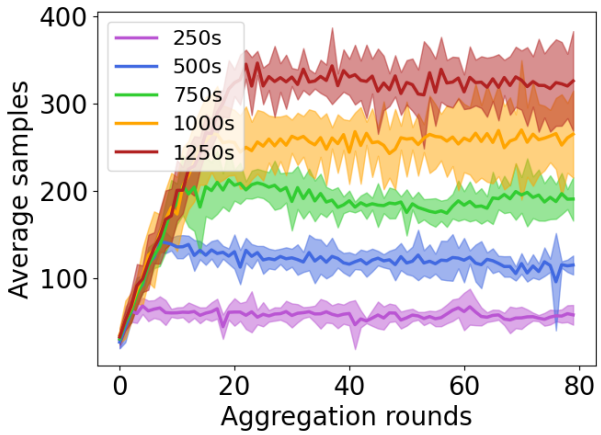
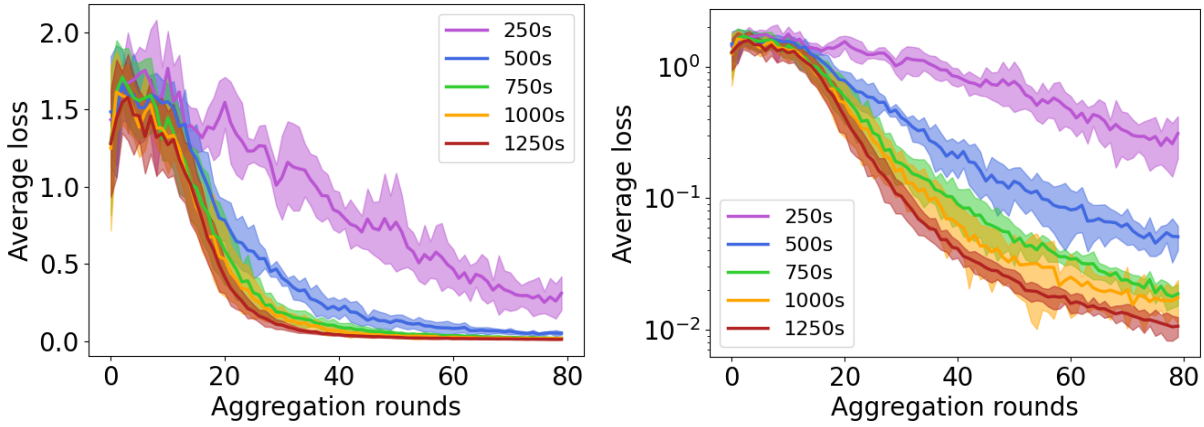


Figure 4.2: 1^{st} experiment's number of samples used for training for each expiration time in function of the aggregation round.

The results of Figure 4.3 are computed as follows: the loss of the general model is obtained by taking a weighted average of the loss of each robot taking part in the aggregation round. As for the aggregations, the weights depend on the number of samples each robot has locally trained the model on. Additionally, since 5 runs were executed for each configuration, the average (the darker line) of the 5 runs is displayed with its 95% confidence interval (the lighter area of one's color tone).

On Figure 4.3, 2 graphs are present, one in linear scale and one in logarithmic scale. The reason why both are shown is that usually, loss graphs in a neural network are with a linear scale. In the case of this thesis, experiments will have really similar results to the point that graphs wouldn't be readable on a linear scale. This (and the next Figure 4.4) will thus be the only linear scale loss graphs presented.



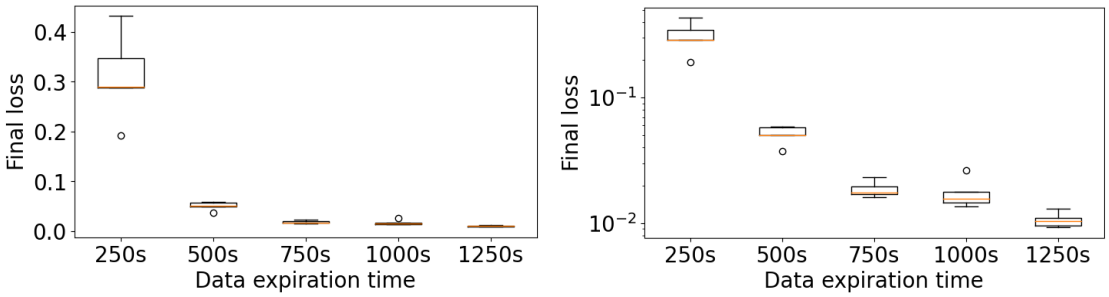
(a) Linear scale **(b)** Logarithmic scale

Figure 4.3: 1^{st} experiment's average loss in function of the aggregation rounds with only collaborating robots for each expiration time in function of aggregation rounds.

When looking at Figure 4.3, visually 250 s and 500 s prove to be insufficient expiration times for training the model, as the convergence speed (i.e. the speed of average loss' decrease in function of the aggregation rounds) of these expiration times is significantly

impacted. On the other hand, starting at 750 s, increasing the data leads to diminishing returns, refer to Figure 4.4. Indeed, even though an expiration time of 1250 s is 500 s longer than an expiration time of 750 s, its improvement (in terms of average loss reduction) is much less than the improvement we see when increasing from 250 s to 750 s. Comparing our results with the ones obtained in the original article that proposed the Flow-FL framework [21], the convergence speed of our system (Figure 4.3) is quantitatively different but qualitatively the same. By that, we mean that we reach the same final results with a loss of 10^{-2} for the higher expiration times, but we do not have the same convergence speed. In particular, our system has a slower convergence speed, which may be caused by the use of a different parameter set (as it was not possible to identify all parameters used in the original Flow-FL article [21]).

In Figure 4.4 we present the final loss (i.e., the loss at the end of an experiment)[2] in the form of boxplots. These boxplots and all the other graphs with boxplots show the following: the lower extreme, lower quartile, median, upper quartile, and upper extreme with the whiskers length being 1.5 times the interquartile range.



(a) Linear scale **(b)** Logarithmic scale

Figure 4.4: 1^{st} experiment's final loss with only collaborative robots for each expiration time.

To quantify why we select the 750 s data expiration time, we present in Table 4.1 a relative performance by setting 1250 s as the best performer. All experiments start with an initial loss, on average, of 1.5. We thus compute the relative performance with:

$$rp = \frac{1.5 - \ell_i}{1.5 - \ell_{1250}} \tag{4.2}$$

Expiration Time [s]	Average Final Loss	Relative Performance
250	0.3095	79.93%
500	0.0507	97.30%
750	0.0187	99.46%
1000	0.0175	99.54%
1250	0.0106	100%

Table 4.1: Relative performance table of the final average loss for each data expiration time.

Based on Table 4.1, augmenting the data beyond 750 s only results in marginal improvement, with 750 s managing to achieve 99.46% of the results obtained with 1250 s,

[2]In this subsection, we stop the experiments at 80 aggregation rounds, to enable a comparison with the experiment with 250s data expiration time, whose minimum number of aggregation rounds is 80.

while also exhibiting a similar convergence speed (as shown in Figure 4.3). Therefore, we selected the 750 s expiration time as the optimal data expiration time for the subsequent experiments.

4.2 Introduction of Byzantine Robots

As we have shown, the swarm is capable of training on the blockchain with 15 honest robots. We can now test its resilience to Byzantine robots. In this experiment, no layer of security is present. This means that if a robot sends bad weights along with a high number of samples, it will be taken into account in the aggregation process.

In this context, Byzantine robots are considered faulty robots, as their sensors could be blocked or malfunctioning. To simulate this, the Byzantine robot sends random weights. The weights are uniformly distributed between -0.5 and 0.5, as this is the interval TensorFlow [40] uses when generating random weight at initialization. The number of samples they send along the weights is 200 as this is the average number of samples robots train on after the first experiment, see the 750 s curve of Figure 4.2 in Section 4.1.

We perform 10 runs for both configurations with zero Byzantine robots and one Byzantine robot. We restrict the tests to one Byzantine robot because, as depicted by Figure 4.5, having only one Byzantine is already sufficient to destroy the model's training.

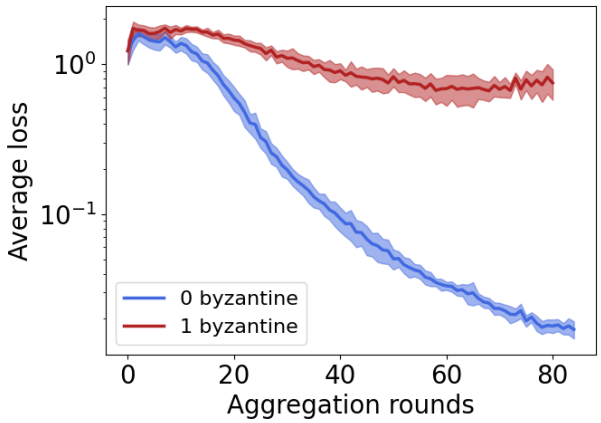


Figure 4.5: 2^{nd} experiment's average loss in function of aggregation rounds with 1 Byzantine robot and no security.

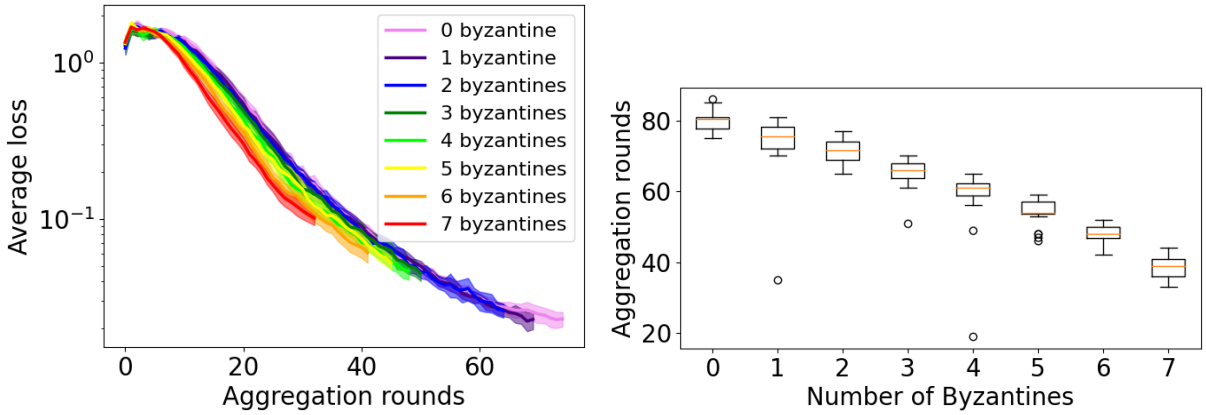
Figure 4.5 shows that having no security makes the system vulnerable to a single malfunctioning or malicious robot. Introducing security layers to face faulty or malicious robots becomes mandatory and is the focus of all the next experiments.

4.3 First Security Layer

In this experiment, we introduce a revised Smart Contract, which is an evolution of the previous one that only implemented the aggregation mechanism without any security layers. The new Smart Contract requires robots to send 5 ether to submit their weights. Robots start with 21 ether and if they fall below 5 ether they can not participate anymore.

We use two security layers (see Section 3.3 which present these security layers) that are designed to cause Byzantine robots to lose ethers. In this section, we present the results of the first layer.

We conduct twenty runs for each configuration with Byzantine numbers ranging from 0 to 7 (which is half of the swarm and also the number of participants needed to complete an aggregation round). The Byzantine robots send the same weights as the previous experiment, of Section 4.2. The objective of this experiment is to verify if the first security layer works. Namely, weights with an MAE (see Section 3.3 which presents the first security layer) greater[3] than $5 \cdot 10^7$ should not be taken into account.



(a) Average loss in function of aggregation rounds.

(b) Aggregation rounds with Byzantines.

Figure 4.6: 3^{th} experiment's average loss for each configuration of Byzantine robots in function of aggregation rounds (left) and aggregation rounds for each configuration of Byzantine robots (right)[4].

There are 3 observations we can make out of Figure 4.6a:

1. The first security layer works, the training is not broken as in all configurations the loss curve converges.

2. The more Byzantine there are the fewer the number of aggregation rounds.

3. When there are more Byzantines, it appears that the training is faster (when comparing round aggregation-wise). For instance, in the case of 7 Byzantines, the curve is shifted to the left, occurring earlier in the aggregation rounds, in contrast to the scenario without Byzantines.

[3]Since we multiply the weights by 10^9 this is the equivalent of requiring an MAE < 0.05 when weights have initial uniform random distribution $\in [-0.5; 0.5]$

[4]Note that the graph in Figure 4.6a the curves present an aggregation count that differs compared to Figure 4.6b which the number of aggregations for each configuration. The reason is that the length of the curve (i.e. the number of aggregations it reaches in Figure 4.6a) is actually limited to the minimum number of aggregations a configuration has reached. This is because we want to keep a confidence interval of a maximum number of samples throughout the whole curve. (Note that the hard outliers are not taken into account otherwise the curve with 4 Byzantines would have stopped at 20 aggregations on Figure 4.6a). The same is done for the next experiments, and it is thus important to only look at the aggregation rounds box plots to actually get a good analysis of the aggregations. The loss curve graphs are only presented to look at the convergence speed.

To better visualize the decrease in aggregation rounds when there are more Byzantine robots, a box plot with the number of aggregations for each configuration is shown in Figure 4.6b.

Additionally to the decrease in aggregation rounds with Byzantine robots in Figure 4.6b, we can also observe that with no Byzantine the number of aggregations also reduces compared to the previous experiments. In Section 4.1, when there was no security with 750 s expiration time, we had the median at 87 aggregations. Now, with the added security layers and the same expiration time, we have the median at 80 aggregations. We can see that these added security layers come at the cost of fewer aggregation taking place. The reason behind this reduction in aggregation rounds is given in Section 4.3.1 where we discuss the ether distribution among robots.

We can also notice in Figure 4.6b that there are hard outliers, namely in configurations with 1 and 4. The cause of this issue is related to the blockchain itself and the PoA consensus mechanism which in some rare cases can get stuck and no more blocks are able to be sealed. This behavior is not explored in this thesis and we only concentrate on the results which are not related to this issue.

To explain the third observation from Figure 4.6a, it is necessary to examine the number of samples used for each aggregation round.

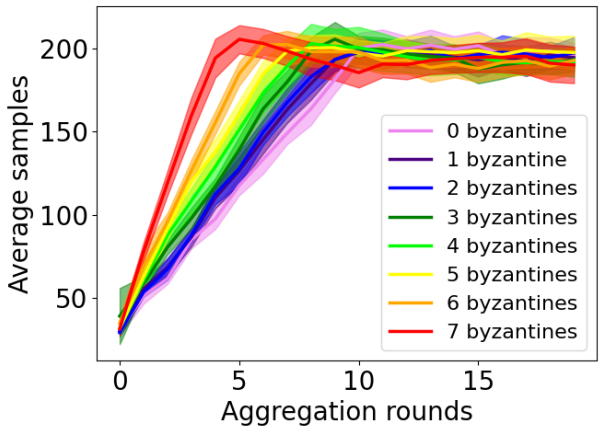


Figure 4.7: 3^{rd} experiment's number of samples used for training for each configuration of Byzantine robots in function of aggregation rounds.

Through Figure 4.7, it becomes easier to understand why configurations with more Byzantine robots exhibit faster training, even though this might not be immediately apparent. When more Byzantines are present, the aggregation process occurs less frequently. For example, the 5^{th} aggregation with 7 Byzantines can happen after 1000 seconds, while the 5^{th} aggregation with 0 Byzantine may happen in less than 500 seconds[5]

Additionally, a certain delay is needed to reach the required 200 samples per training cycle. And as we have seen in the first experiment in Section 4.1 with Figure 4.3, the convergence speed depends on the amount of data used. So with more Byzantines, Figure 4.6a appears to show that training is faster because at the 5^{th} aggregation with 7

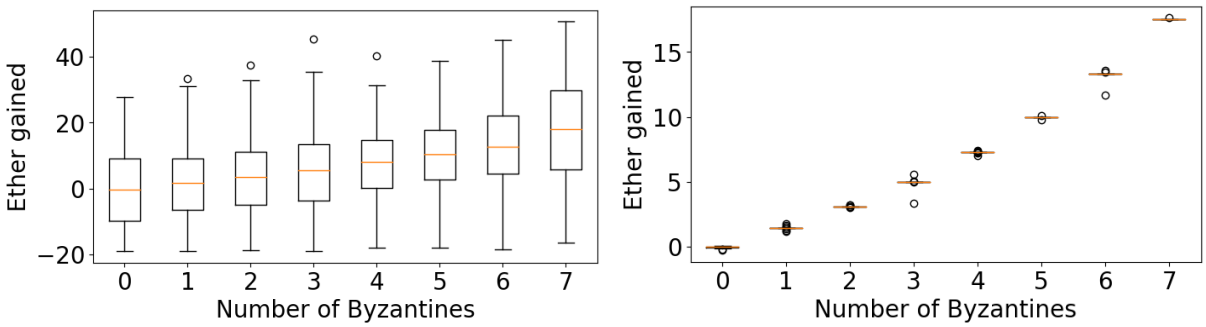
[5]These values were used for an example and are not actual results.

Byzantines, the robots are already training with 200 samples, whilst with 0 Byzantine, they start training with 200 samples only after the 10^{th} aggregation.

This should not be confused by assuming that the x-axis has the wrong units (i.e., aggregation instead of time). If the x-axis is set to time (s), the curves (with Byzantines) would more closely resemble the curve obtained in the first experiment with a 250 s expiration time. Indeed, the final loss is much higher at the end of the experiment due to significantly fewer aggregations taking place.

4.3.1 Ether distribution

The smart contract now requires 5 ether for weights to be submitted on the Blockchain. Hence, the desired behavior is that honest robots become rich whereas bad ones become poor.



(a) Ether gained by honest robots. **(b)** Average ether gained by honest robots.

Figure 4.8: 3^{rd} experiment's ether of honest robots for each configuration of Byzantine.

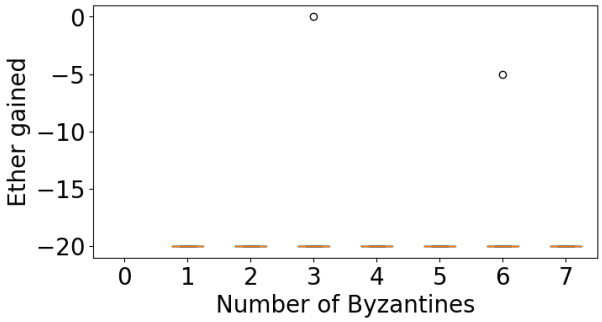


Figure 4.9: 3^{rd} experiment's ether of Byzantine robots for each configuration of Byzantine.

From Figure 4.8b and 4.9, Byzantine robots effectively lose their ether and make honest robots richer on average. This indicates that the first security layer is not rewarding the bad robots and is actually giving their ether to the honest robots.

Unfortunately, when examining Figure 4.8a, which presents a box plot illustrating the distribution of ether among all honest robots (as opposed to the average per experiment shown in 4.8b), we observe a significant disparity. Some robots become affluent, some maintain stability, while others become impoverished to the extent that they can

no longer participate. This discrepancy arises from the ranking system, the second layer of security, which rewards the top 5 robots and penalizes the bottom 2. The ranking compels the robots to adhere to a specific pattern, implying that they should be trained on similar amounts and types of data. For instance, if the majority of trajectories consist of straight lines and one robot suddenly only observes other robots avoiding obstacles, it will train its neural network differently. Consequently, this deviation may result in a significantly different MAE compared to others, leading to its frequent placement as the last-ranked robot, despite its good work to improve the neural network.

This issue is also the cause of the fewer aggregation rounds taking place compared to the previous experiments, like in Section 4.1. Since fewer robots can afford to participate (i.e., due to the Byzantines and the ones whose training diverges losing ethers), fewer aggregations will take place overall.

This implementation thus harbors a weakness, originating from the challenge of identifying which weights are truly problematic (i.e., the MAE computation may be too basic). This problem is akin to the more widely recognized issue of the black box in neural networks. The term *black box* stems from the difficulty in comprehending the correspondence between specific weights and their impact on the quality of the model.

4.4 Second Security Layer

In the previous experiment, Byzantine robots were sending weights that were instantly evicted due to being too different, because they generated an MAE higher than the threshold. This time, to be integrated into the aggregation rounds, malicious Byzantine robots send the same weights as the current aggregated weights on the blockchain. This is to try and slow down the training. To prevent this type of attack, which is much less harmful than the previous one, a second security layer is introduced. Refer to Section 3.3 for further details on the security layers. The configuration is the same as Section 4.3 with 20 runs for Byzantine ranging from 0 to 7.

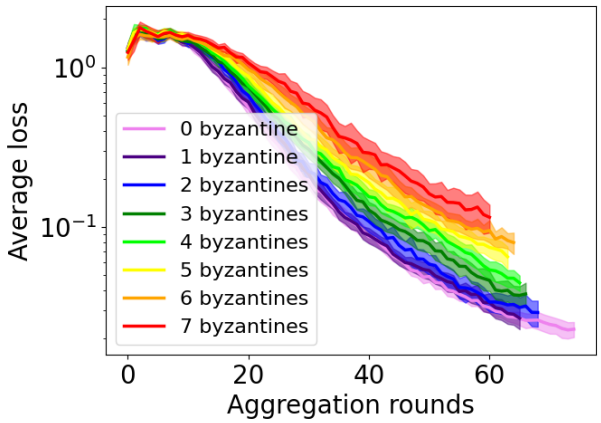


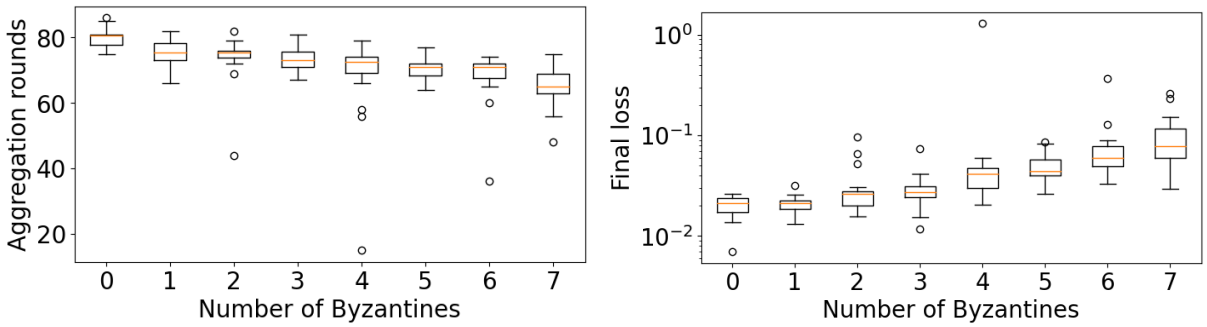
Figure 4.10: 4^{th} experiment's average loss for each configuration of Byzantine robots in function of aggregation rounds.

There are 3 observations we can make out of Figure 4.10:

1. The second security layer works as models converge.
2. Compared to the previous experiment (section 4.3) more aggregation rounds are taking place with Byzantine.
3. The convergence speed is now slower aggregation-wise when there are more Byzantine.

The reason behind the second observation is that Byzantine robots are not simply evicted anymore but actually take part in the aggregations. Note that the aggregation count still decreases but this reduction is slower than in the previous experiment in Section 4.3. This decrease in aggregation rounds is shown in Figure 4.11a.

The third observation is directly linked to the second one; since Byzantines are accepted, bad weights, which do not help the training, are taken into account which slows down the convergence speed. As a consequence, the final loss reached decreases with the amount of Byzantines, see Figure 4.11b.



(a) Aggregation rounds with Byzantines

(b) Final loss with Byzantines

Figure 4.11: 4^{th} experiment's aggregation rounds and final loss for each configuration of Byzantine robots.

Since more aggregations occur because Byzantines are being accepted in the aggregation rounds, we do not observe the same effect of training becoming faster with an increasing number of Byzantines, as we had in the previous experiment, in Section 4.3. In Figure 4.12 the curves overlap, emphasizing the point that all configurations have access to the same amount of data aggregation-wise.

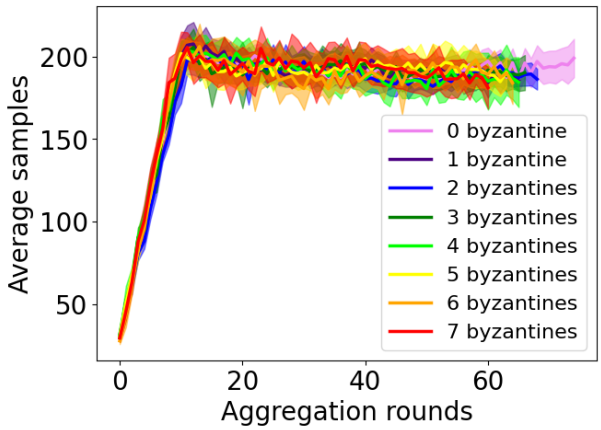
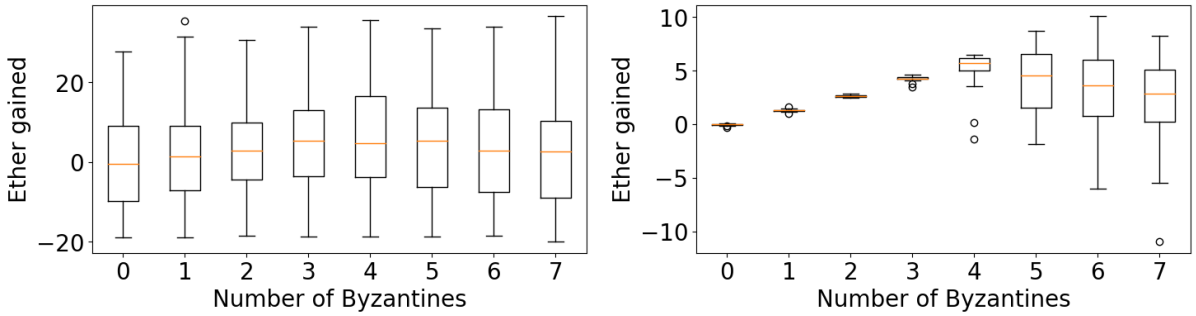


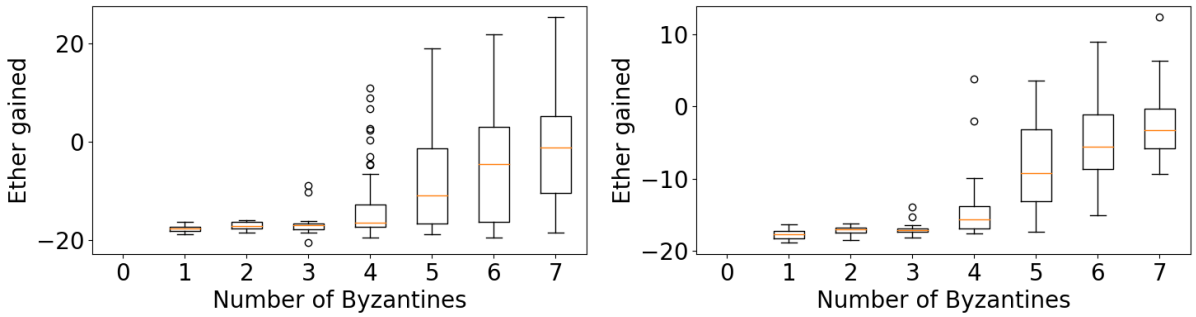
Figure 4.12: 4^{th} experiment's number of samples used for training for each configuration of Byzantine robots in function of aggregation rounds.

4.4.1 Ether distribution



(a) Ether gained by honest robots **(b)** Average ether gained by honest robots

Figure 4.13: 4^{th} experiment's ether of honest robots for each configuration of Byzantine.



(a) Ether gained by Byzantine robots **(b)** Average ether gained by Byzantine robots

Figure 4.14: 4^{th} experiment's ether of Byzantine robots for each configuration of Byzantine.

Let's first analyze Figure 4.13b and 4.14b, which represent the average ether gained by honest and Byzantine robots. We see an interesting pattern where up until 3 Byzantines, which is less than half of the quorum size (the quorum size being 7), the honest robots gain ether and the Byzantines lose all their ether. However, the moment there are more

Byzantine than half of the quorum size the second security layer starts to fail and Byzantines manage to keep their ether.

If we now look at Figure 4.13a and 4.14a, we still see the limitation that honest robots may lose all their ether. This limitation is the same as the one discussed in Section 4.3.1. The high variance in the ether repartition among honest robots in this case is the same, but in addition to that, we now also have the fact that with more Byzantine the honest robots do not increase their ether on average.

4.5 Smart Byzantines and Current Limitation

In this last experiment, Byzantine robots try to be ranked first by sending weights with an MAE that follows the current trend of the training. For more detailed information on the weights' generation, refer to Section 3.3.2 on the last Byzantine presented. The configuration is the same as Section 4.3 with 18 runs[6] for Byzantine ranging from 0 to 7.

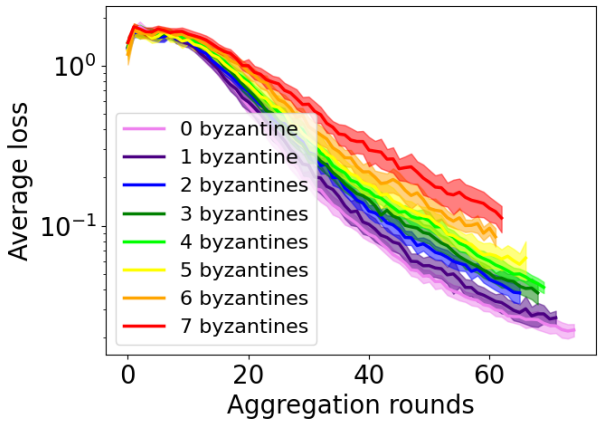
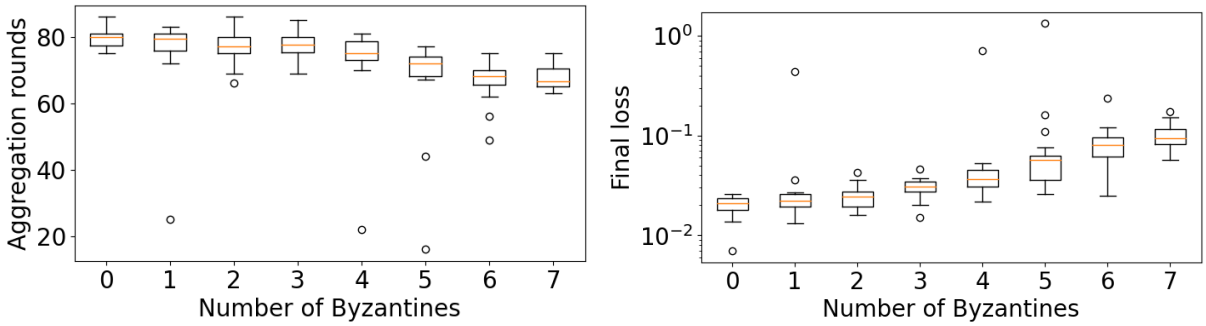


Figure 4.15: 5^{th} experiment's average loss for each configuration of Byzantine robots in function of aggregation rounds.

By looking at Figure 4.15, we can see that it follows the same pattern as the loss curve observed in Section 4.4 with Figure 4.10. The more Byzantine there are the slower the training curve. But the same story can not be told with the aggregation rounds. As Byzantines do not get eliminated the number of aggregation doesn't directly starts decreasing. This is better represented in Figure 4.16. Indeed, it is only when there are more than 4 Byzantines that the number of aggregation starts to decrease, and this is actually because honest robots start to get evicted (this is understandable through the ether graphs in Section 4.5.1).

[6]We restricted the simulations of the 5^{th} to 18 runs due to time constraints.

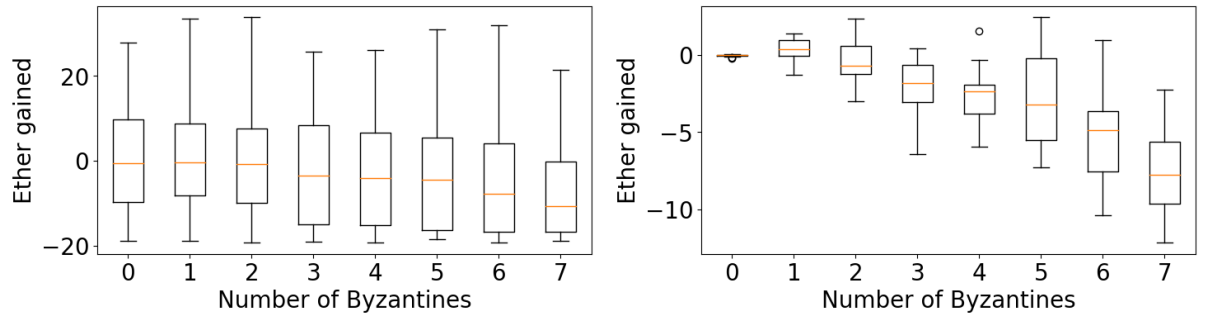


(a) Aggregation rounds with Byzantines **(b)** Final loss with Byzantines

Figure 4.16: 5^{th} experiment's aggregation rounds and final loss for each configuration of Byzantine robots.

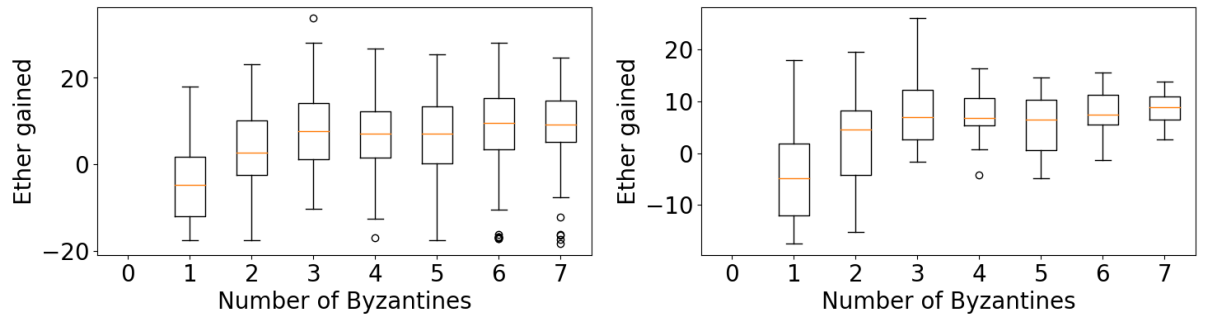
In the end, by only looking at Figure 4.15 and 4.16b, we could think that this *smart* Byzantine robots isn't actually doing much better than the previous Byzantine robot from Section 4.4. But this story is much different if we start looking at the ether distribution between honest and Byzantine robots.

4.5.1 Ether distribution



(a) Ether gained by honest robots **(b)** Average ether gained by honest robots

Figure 4.17: 5^{th} experiment's ether of honest robots for each configuration of Byzantine.



(a) Ether gained by Byzantine robots **(b)** Average ether gained by Byzantine robots

Figure 4.18: 5^{th} experiment's ether of Byzantine robots for each configuration of Byzantine.

Through Figure 4.17 and 4.18 we can see that, on the one hand, honest robots lose ether the more Byzantine robots there are, and on the other hand, Byzantine robots gain more

ether. This means that Byzantine robots could eventually prevent honest robots from being able to participate. At this moment, Byzantine robots can start harming the network with MAE much closer to the Threshold and thus prevent the proper training, similar to the scenario of 4.2. This is also a limitation of the current implementation.

Note that they reach a plateau[7] around 3 and 4 Byzantine robots. This is because of 2 reasons:

1. The more Byzantines there are the more ether on average each honest robots lose.
2. The more Byzantine robots there are, the more they have to share among themselves, and the fewer honest robots there are providing the ether.

Since the ether gained by Byzantine robots can be computed as:

$$e_b^+ = \frac{n_h}{n_b} \cdot e_h^- \tag{4.3}$$

where e_b^+ is the average ether that each Byzantine robot gains, e_h^- is the average ether lost by each of the honest robots, n_h is the number of honest robots, and n_b is the number of Byzantine robots. Having the fraction of honest robots decrease and the ether lost increase leads to this plateau.

4.6 Discussion

Through these different experiments, we showed that it is possible to actually perform federated learning on the blockchain using smart contracts in Section 4.1. We also presented the threat Byzantine robots pose to the actual training of the network with experiment 2 in Section 4.2. We thus implemented a security layer to prevent this attack and presented the results in Section 4.3. This protective layer worked as intended and enabled the network to converge again. We can observe that the first security layer, even though quite basic, is actually the one that prevents the most damage to the network as it prevents high disturbance in the network weights.

Similarly, we thought of other attacks as we were building the first security layer and thus introduced the second security layer in Section 4.4 to motivate robots to follow a certain trend. This approach prevented a new attack which consisted of sending weights that the robot did not obtaining through training. The trade-off of this second security layer is the reduction in the number of aggregation rounds, even in the cases with no Byzantines. Unfortunately, it also had the consequence of potentially leading to some honest robots losing their ether. Finally, in Section 4.5, we present a way in which Byzantine robots can exploit the system in order to take the ether of the honest robots, and eventually, using the gained ether to control the convergence of the network.

This implementation thus harbors two weaknesses. The first one is the distance function (MAE) used for the ranking in the second security layer: it can be vulnerable to

[7]This is actually a maximum and not a plateau since if we further increase the number of Byzantine, there will be no more honest robots to take ether from. With 15 Byzantines robots, the average ether gained will be 0 as they all send the same weights and the same amount of samples.

some attacks by a smart or malicious robots, while incentivizing robots to provide models that follow a certain trend, thus removing some freedom in the training of the model.

Second, the current implementation of the blockchain is also a limitation. Making the aggregation take place on the blockchain forces high amounts of data to be transferred. This means that the blockchain needs to perform computations of minimum $O(n)$ with n being the total number of weights. Also, on the memory side, since the blockchain stores transactions (which contain the weights) in blocks, every aggregated weight submitted by a robot will be stored on the blockchain.

Figure 4.19 shows the memory used by the blockchain throughout the experiments. We can see that it linearly increases with time and reaches at the end 100 MB of memory usage. This is with a neural network of only 2848 weights. This architecture is actually on the smaller side and if this implementation would be used on larger models[8] the memory usage would rapidly increase which is not practical (especially on smartphone that generally have gigabytes of memory available).

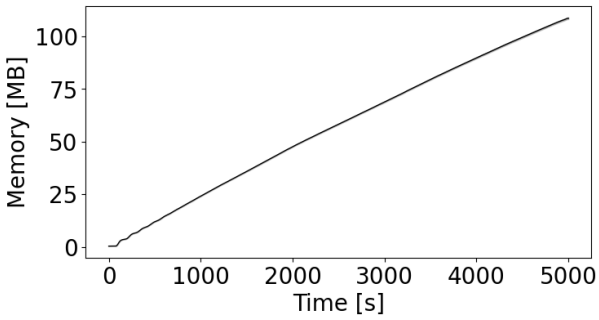


Figure 4.19: Memory usage in megabytes of the blockchain in function of time.

[8]To put in perspective, Chat-GPT, the large language model, has $1.75 \cdot 10^{11}$ weights [5]. (This is at the opposite end of the spectrum as it is arguably one of the largest models currently in existence)

Chapter 5

Future Work & Conclusion

5.1 Future Work

There are two main issues with the current implementation. The first one comes from the distance function (MAE) used for the ranking system of the second security layer, and the second one is the blockchain implementation.

To solve both issues aforementioned multiple options are possible. We believe that the most important one to explore would be the development of a better distance function, surpassing the Mean Absolute Error (MAE) approach. However, it is important to consider that the underlying issue is the difficulty in identifying what are good and bad weights/parameters when training a neural network. For instance, Unterthiner et al. [42] provides a method to estimate the performance of a neural network by looking only at its weights. Changing from our MAE distance function approach to their proposed metric can help address two issues faced by the second security layer: firstly, it would prevent honest robots from losing ether due to being unfairly ranked despite their honest work; secondly, it would eliminate the security breach that allows Byzantine robots to take ether from honest robots. Another method would be to use the more popular Shapley value, used by Liu et al. [20] in the FedCoin project, which allows for assessing the contribution of federated learning participants.

Concerning the limitation of our blockchain implementation, it is common practice to send the hash of large information instead of the information itself on the blockchain. To adapt this method to the scenario of this thesis, the robots would initially start with the same initial weights, and the hash of these initial weights would be stored on the blockchain. When a robot wishes to submit new weights, it would submit the hash of the new weights on the blockchain, while sharing the actual weights with neighboring robots off-chain (i.e., without resorting to a blockchain transaction). The neighboring robots can verify the authenticity of the weights by computing the hash of the received weights and cross-checking with the hash on-chain. Then, once enough robots sent the hash of their weights on the blockchain, the robots compute the aggregation locally and then share the hash of the aggregated weights on the blockchain. In this way, only hashes are stored on the blockchain and the memory usage wouldn't depend on the size of the network anymore, while the blockchain is still being leveraged to ensure the authentication of messages, synchronization of the aggregated model between all participants, and to protect the system from devastating attacks such as Sybil attacks or DoS (by requiring

ethers alongside the submission of the hashed weights).

5.2 Conclusion

Our objective was to investigate the implementation of federated learning in swarm robotics using blockchain smart contracts. Through the proof of concept in the present thesis, we show that this is indeed feasible. Subsequently, our focus shifted from a feasibility study, towards highlighting the vulnerability of federated learning to attacks (such as robots transmitting random weights) and illustrating the use of smart contracts to mitigate such attacks. Although these protective measures are effective, they do involve certain trade-offs, such as reduced aggregation over time and the possibility of penalizing honest robots.

We proceeded to explore additional types of attacks and showcased the current limitations of our implementation. These limitations include, in the first place, a weak ability to distinguish between good and bad parameters. Secondly, and closely related to the previous limitation, we show that there is a particular attack that can exploit this weakness, thus enabling the siphoning of ether from honest robots to Byzantines. Lastly, the current smart contract implementation lacks scalability when applied to larger networks with a higher number of parameters. These issues have been discussed in Section 4.6 and potential solutions and areas of research have been proposed in Section 5.1.

In conclusion, while federated learning has shown promise in various fields, including multi-robot systems, its application in the context of swarm robotics is hampered by challenges such as synchronizing a distributed database and achieving secure peer-to-peer communications. By incorporating blockchain technology into swarm robotics, we offer a promising solution to address these challenges. Our approach leverages the database synchronization and security features of blockchain, enabling robot swarms to overcome these challenges without compromising their critical properties: autonomy, decentralization, and scalability. Although this proof of concept has demonstrated its potential, further research and future work are needed to refine and enhance the proposed solutions.

Bibliography

[1] Durmus Alp Emre Acar et al. *Federated Learning Based on Dynamic Regularization*. Nov. 9, 2021. DOI: 10.48550/arXiv.2111.04263. arXiv: 2111.04263[cs]. URL: http://arxiv.org/abs/2111.04263 (visited on 05/05/2023).

[2] *Adam - Cornell University Computational Optimization Open Textbook - Optimization Wiki*. URL: https://optimization.cbe.cornell.edu/index.php?title=Adam (visited on 05/05/2023).

[3] Gerardo Beni. "From Swarm Intelligence to Swarm Robotics". In: *Swarm Robotics*. Ed. by Erol Şahin and William M. Spears. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 1–9. ISBN: 978-3-540-30552-1. DOI: 10.1007/978-3-540-30552-1_1.

[4] Gianluca Bontempi. *"Statistical foundations of machine learning: the handbook"*. Feb. 15, 2022.

[5] Tom B. Brown et al. *Language Models are Few-Shot Learners*. July 22, 2020. DOI: 10.48550/arXiv.2005.14165. arXiv: 2005.14165[cs]. URL: http://arxiv.org/abs/2005.14165 (visited on 06/01/2023).

[6] Vitalik Buterin. "A NEXT GENERATION SMART CONTRACT & DECENTRALIZED APPLICATION PLATFORM". In: ().

[7] Eduardo Castelló Ferrer. "The Blockchain: A New Framework for Robotic Swarm Systems". In: *Proceedings of the Future Technologies Conference (FTC) 2018*. Ed. by Kohei Arai, Rahul Bhatia, and Supriya Kapoor. Advances in Intelligent Systems and Computing. Cham: Springer International Publishing, 2019, pp. 1037–1058. ISBN: 978-3-030-02683-7. DOI: 10.1007/978-3-030-02683-7_77.

[8] Marco Dorigo and Mauro Birattari. "Swarm intelligence". In: *Scholarpedia* 2.9 (Sept. 29, 2007), p. 1462. ISSN: 1941-6016. DOI: 10.4249/scholarpedia.1462. URL: http://www.scholarpedia.org/article/Swarm_intelligence.

[9] Marco Dorigo, Mauro Birattari, and Manuele Brambilla. "Swarm robotics". In: *Scholarpedia* 9.1 (Jan. 14, 2014), p. 1463. ISSN: 1941-6016. DOI: 10.4249/scholarpedia.1463. URL: http://www.scholarpedia.org/article/Swarm_robotics.

[10] *EIP-225: Clique proof-of-authority consensus protocol*. Ethereum Improvement Proposals. URL: https://eips.ethereum.org/EIPS/eip-225 (visited on 05/28/2023).

[11] *Federated Learning: Collaborative Machine Learning without Centralized Training Data*. Apr. 6, 2017. URL: https://ai.googleblog.com/2017/04/federated-learning-collaborative.html (visited on 05/23/2023).

[12] Lorenzo Garattoni and Mauro Birattari. "Swarm Robotics". In: *Wiley Encyclopedia of Electrical and Electronics Engineering*. John Wiley & Sons, Ltd, 2016, pp. 1–19. ISBN: 978-0-471-34608-1. DOI: 10.1002/047134608X.W8312. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/047134608X.W8312.

[13] Paulo Gonçalves et al. "The e-puck, a Robot Designed for Education in Engineering". In: *Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions* 1 (Jan. 1, 2009).

[14] Ekaterina Gracheva. "Trainless model performance estimation based on random weights initialisations for neural architecture search". In: *Array* 12 (Dec. 1, 2021), p. 100082. ISSN: 2590-0056. DOI: 10.1016/j.array.2021.100082. URL: https://www.sciencedirect.com/science/article/pii/S2590005621000308.

[15] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1, 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[16] Jiawen Kang et al. "Incentive Mechanism for Reliable Federated Learning: A Joint Optimization Approach to Combining Reputation and Contract Theory". In: *IEEE Internet of Things Journal* 6.6 (Dec. 2019). Conference Name: IEEE Internet of Things Journal, pp. 10700–10714. ISSN: 2327-4662. DOI: 10.1109/JIOT.2019.2940820.

[17] Hyesung Kim et al. *Blockchained On-Device Federated Learning*. July 1, 2019. DOI: 10.48550/arXiv.1808.03949. arXiv: 1808.03949[cs,math]. URL: http://arxiv.org/abs/1808.03949 (visited on 05/24/2023).

[18] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. Jan. 29, 2017. DOI: 10.48550/arXiv.1412.6980. arXiv: 1412.6980[cs]. URL: http://arxiv.org/abs/1412.6980.

[19] Tian Li et al. *Federated Optimization in Heterogeneous Networks*. Apr. 21, 2020. DOI: 10.48550/arXiv.1812.06127. arXiv: 1812.06127[cs,stat]. URL: http://arxiv.org/abs/1812.06127 (visited on 05/05/2023).

[20] Yuan Liu et al. *FedCoin: A Peer-to-Peer Payment System for Federated Learning*. Feb. 25, 2020. DOI: 10.48550/arXiv.2002.11711. arXiv: 2002.11711[cs,stat]. URL: http://arxiv.org/abs/2002.11711 (visited on 05/24/2023).

[21] Nathalie Majcherczyk, Nishan Srishankar, and Carlo Pinciroli. "Flow-FL: Data-Driven Federated Learning for Spatio-Temporal Predictions in Multi-Robot Systems". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. 2021 IEEE International Conference on Robotics and Automation (ICRA). ISSN: 2577-087X. May 2021, pp. 8836–8842. DOI: 10.1109/ICRA48506.2021.9560791.

[22] H. Brendan McMahan et al. *Communication-Efficient Learning of Deep Networks from Decentralized Data*. Jan. 26, 2023. DOI: 10.48550/arXiv.1602.05629. arXiv: 1602.05629[cs]. URL: http://arxiv.org/abs/1602.05629.

[23] Seongin Na et al. *Federated Reinforcement Learning for Collective Navigation of Robotic Swarms*. Sept. 11, 2022. arXiv: 2202.01141[cs]. URL: http://arxiv.org/abs/2202.01141 (visited on 05/23/2023).

[24] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System". In: ().

[25] Cong Nguyen et al. "Proof-of-Stake Consensus Mechanisms for Future Blockchain Networks: Fundamentals, Applications and Opportunities". In: *IEEE Access* PP (June 26, 2019), pp. 1–1. DOI: 10.1109/ACCESS.2019.2925010.

[26] OpenAI. *GPT-4 Technical Report*. Mar. 27, 2023. DOI: 10.48550/arXiv.2303.08774. arXiv: 2303.08774[cs]. URL: http://arxiv.org/abs/2303.08774.

[27] Alexandre Pacheco, Volker Strobel, and Marco Dorigo. "A Blockchain-Controlled Physical Robot Swarm Communicating via an Ad-Hoc Network". In: *Swarm Intelligence*. Ed. by Marco Dorigo et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 3–15. ISBN: 978-3-030-60376-2. DOI: 10.1007/978-3-030-60376-2_1.

[28] Alexandre Pacheco et al. "Real-Time Coordination of a Foraging Robot Swarm Using Blockchain Smart Contracts". In: *Swarm Intelligence*. Ed. by Marco Dorigo et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 196–208. ISBN: 978-3-031-20176-9. DOI: 10.1007/978-3-031-20176-9_16.

[29] Carlo Pinciroli et al. "ARGoS: a modular, parallel, multi-engine simulator for multi-robot systems". In: *Swarm Intelligence* 6.4 (Dec. 1, 2012), pp. 271–295. ISSN: 1935-3820. DOI: 10.1007/s11721-012-0072-5. URL: https://doi.org/10.1007/s11721-012-0072-5 (visited on 05/15/2023).

[30] *Proof-of-stake (PoS)*. ethereum.org. URL: https://ethereum.org (visited on 05/28/2023).

[31] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. June 15, 2017. arXiv: 1609.04747[cs]. URL: http://arxiv.org/abs/1609.04747.

[32] Juergen Schmidhuber. *Annotated History of Modern AI and Deep Learning*. Dec. 29, 2022. DOI: 10.48550/arXiv.2212.11279. arXiv: 2212.11279[cs]. URL: http://arxiv.org/abs/2212.11279.

[33] Tianshu Song, Yongxin Tong, and Shuyue Wei. "Profit Allocation for Federated Learning". In: *2019 IEEE International Conference on Big Data (Big Data)*. 2019 IEEE International Conference on Big Data (Big Data). Dec. 2019, pp. 2577–2586. DOI: 10.1109/BigData47090.2019.9006327.

[34] Ralf C. Staudemeyer and Eric Rothstein Morris. *Understanding LSTM – a tutorial into Long Short-Term Memory Recurrent Neural Networks*. Sept. 12, 2019. DOI: 10.48550/arXiv.1909.09586. arXiv: 1909.09586[cs]. URL: http://arxiv.org/abs/1909.09586.

[35] Volker Strobel, Eduardo Castelló Ferrer, and Marco Dorigo. "Blockchain Technology Secures Robot Swarms: A Comparison of Consensus Protocols and Their Resilience to Byzantine Robots". In: *Frontiers in Robotics and AI* 7 (2020). ISSN: 2296-9144. URL: https://www.frontiersin.org/articles/10.3389/frobt.2020.00054 (visited on 05/23/2023).

[36] Volker Strobel and Marco Dorigo. "Blockchain technology for robot swarms: A shared knowledge and reputation management system for collective estimation". In: ().

[37] Ilya Sutskever, James Martens, and George Dahl. "On the importance of initialization and momentum in deep learning". In: ().

[38] Péter Szilágyi. *EIP-225: Clique proof-of-authority consensus protocol*. Ethereum Improvement Proposals. Mar. 6, 2017. URL: https://eips.ethereum.org/EIPS/eip-225 (visited on 05/09/2023).

[39] Keras Team. *Keras documentation: Layer activation functions*. URL: https://keras.io/api/layers/activations/ (visited on 05/03/2023).

[40] Keras Team. *Keras documentation: Layer weight initializers*. URL: https://keras.io/api/layers/initializers/ (visited on 05/05/2023).

[41] Keras Team. *Keras documentation: Losses*. URL: https://keras.io/api/losses/ (visited on 05/03/2023).

[42] Thomas Unterthiner et al. *Predicting Neural Network Accuracy from Weights*. Apr. 9, 2021. DOI: 10.48550/arXiv.2002.11448. arXiv: 2002.11448[cs,stat]. URL: http://arxiv.org/abs/2002.11448 (visited on 05/20/2023).

[43] Ashish Vaswani et al. *Attention Is All You Need*. Dec. 5, 2017. DOI: 10.48550/arXiv.1706.03762.

[44] Jianyu Wang et al. "A Novel Framework for the Analysis and Design of Heterogeneous Federated Learning". In: *IEEE Transactions on Signal Processing* 69 (2021). Publisher: Institute of Electrical and Electronics Engineers Inc., pp. 5234–5249. ISSN: 1053-587X. DOI: 10.1109/TSP.2021.3106104. URL: https://collaborate.princeton.edu/en/publications/a-novel-framework-for-the-analysis-and-design-of-heterogeneous-fe (visited on 05/05/2023).

[45] Zexin Wang, Biwei Yan, and Anming Dong. "Blockchain Empowered Federated Learning for Data Sharing Incentive Mechanism". In: *Procedia Computer Science*. International Conference on Identification, Information and Knowledge in the internet of Things, 2021 202 (Jan. 1, 2022), pp. 348–353. ISSN: 1877-0509. DOI: 10.1016/j.procs.2022.04.047. URL: https://www.sciencedirect.com/science/article/pii/S1877050922005816 (visited on 05/24/2023).

[46] Zhilin Wang and Qin Hu. *Blockchain-based Federated Learning: A Comprehensive Survey*. Oct. 5, 2021. arXiv: 2110.02182[cs]. URL: http://arxiv.org/abs/2110.02182 (visited on 05/23/2023).

[47] Hongda Wu and Ping Wang. *Fast-Convergent Federated Learning with Adaptive Weighting*. Apr. 5, 2021. arXiv: 2012.00661[cs]. URL: http://arxiv.org/abs/2012.00661 (visited on 05/05/2023).

[48] Xi Yu et al. "Federated Learning Optimization Algorithm for Automatic Weight Optimal". In: *Computational Intelligence and Neuroscience* 2022 (Nov. 9, 2022). Publisher: Hindawi, e8342638. ISSN: 1687-5265. DOI: 10.1155/2022/8342638. URL: https://www.hindawi.com/journals/cin/2022/8342638/.

[49] Xudong Zhu, Fan Zhang, and Hui Li. "Swarm Deep Reinforcement Learning for Robotic Manipulation". In: *Procedia Computer Science*. 12th International Conference on Emerging Ubiquitous Systems and Pervasive Networks / 11th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare 198 (Jan. 1, 2022), pp. 472–479. ISSN: 1877-0509. DOI: 10.1016/j.procs.2021.12.272. URL: https://www.sciencedirect.com/science/article/pii/S1877050921025114 (visited on 05/23/2023).