DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA DELL'INFORMAZIONE

Towards Trustworthy Collaborative Machine Learning: A Blockchain-Driven Federated Approach

Relatore
Prof. Perin Giovanni

Laureanda
Antonello Ilenia

ANNO ACCADEMICO 2023-2024

Data di laurea 19/03/2024

Abstract

The purpose of this thesis is to explore how integrating Blockchain (BC) technology with Federated Learning (FL) can improve the security and reliability of standard FL systems. To achieve this, the principles of FL and BC are first reviewed. Next, various Blockchain Enabled Federated Learning (BCFL) architectures are theoretically examined, weighing their strengths and weaknesses. Finally, a straightforward implementation of one of the discussed architectures is conducted to assess its efficiency and efficacy. The simulation highlights that while the computational overhead associated with BC technology is significant, it does not affect the model's quality. Despite this overhead, the integrity added to FL by BC makes it a promising solution for securing decentralized and distributed machine learning systems.

Lo scopo di questa tesi è studiare come integrare la tecnologia Blockchain (BC) con il Federated Learning (FL) possa migliorare la sicurezza e l'affidabilità dei sistemi FL standard. Per raggiungere questo obiettivo, vengono prima esaminati i principi di FL e BC. Successivamente, varie architetture di Blockchain-Enabled Federated Learning (BCFL) vengono esaminate teoricamente, valutandone i punti di forza e debolezze. Infine, viene condotta un'implementazione di una delle architetture discusse per valutarne l'efficienza e l'efficacia. La simulazione evidenzia che mentre il sovraccarico computazionale associato alla tecnologia BC è notevole, non ha effetto sulla qualità del modello. Nonostante questo sovraccarico, l'integrità aggiunta al FL dalla BC la rende una soluzione promettente per la sicurezza di sistemi di apprendimento automatico decentralizzati e distribuiti.

Contents

1 Introduction **7**
1.1 Federated Learning 7
1.1.1 Principles 7
1.1.2 Advantages and challenges 9
1.2 Blockchain 9
1.2.1 Principles 9
1.2.2 Security Properties 11

2 Blockchain-Enabled Federated Learning **13**
2.1 Integration of Blockchain and Federated Learning 13
2.2 Advantages and challenges 15

3 Implementation **17**
3.1 Description of the environment and tools used 17
3.2 Overview of the Blockchain Implementation 18
3.3 Overview of the Federated Learning Implementation 22
3.4 Interaction between Blockchain and Federated Learning Structures 23

4 Results **27**

5 Conclusions **33**

Chapter 1

Introduction

1.1 Federated Learning

In today's world, personal devices such as smartphones generate and store vast amounts of data. This data is vital in improving user experience by powering intelligent applications. However, the private nature of the data raises security concerns when stored remotely. Federated Learning (FL) was introduced to address these concerns.

1.1.1 Principles

Federated learning is a machine learning technique that sees a federation of users who collectively solve a training task coordinated by a server. Each client has its dataset, which is never uploaded but used to train a local model and compute an upgrade, which will be sent to the server. FL trains models using non-IID datasets that vary significantly in size depending on the device characteristics and usage.

In this thesis, we consider the Federated Averaging (FedAvg) algorithm, which utilizes stochastic gradient descent (SGD) [2]. The traditional FL approach involves client selection, local model training using predefined algorithms such as SGD, and aggregation of updates by the server. While this approach is computationally efficient, it requires multiple rounds of communication and several users to train effective models.

Algorithm 1 `FederatedAveraging`. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate.

Server executes:
  initialize w_0
  for each round $t = 1, 2, \dots$ **do**
    $m \leftarrow \max(C \cdot K, 1)$
    $S_t \leftarrow$ (random set of m clients)
    for each client $k \in S_t$ **in parallel do**
      $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$
    $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$

ClientUpdate(k, w): *// Run on client k*
  $\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size B)
  for each local epoch i from 1 to E **do**
    for batch $b \in \mathcal{B}$ **do**
      $w \leftarrow w - \eta \nabla \ell(w; b)$
  return w to server

Figure 1.1: FedAvg algorithm.[2]

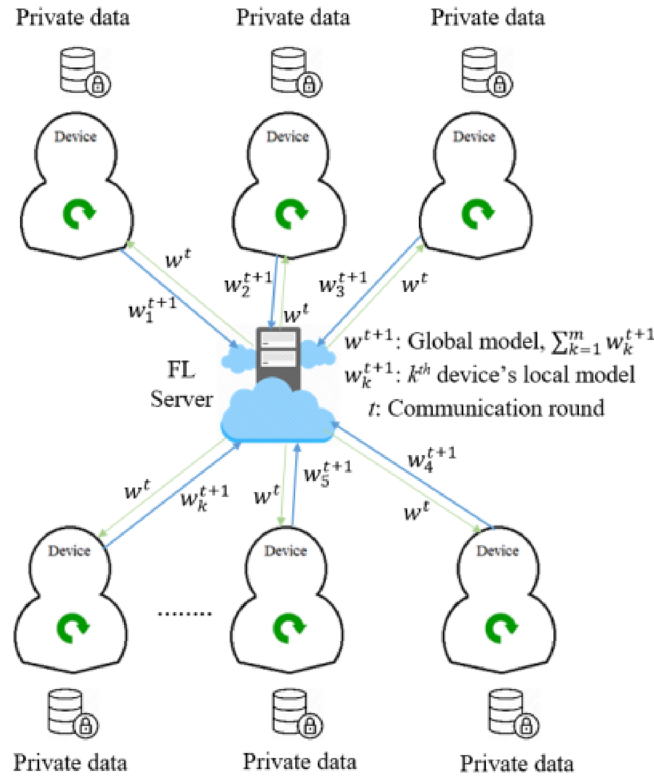


Figure 1.2: Standard FL architecture [4].

1.1.2 Advantages and challenges

This federated approach offers several advantages, including:

- Safe data transfer: The central server does not store any raw data, minimizing the potential for data breaches and reducing concerns for the service provider.
- Communications efficiency: Users only need to send upgrades, which reduces communication costs.

However, this approach also presents several challenges, such as:

- Single point of failure: If the aggregator fails, the entire system could be compromised due to a malicious attack or connection issues.
- Lack of incentives: Users are typically assumed to contribute their computational power without payment, which could result in participants providing unreliable data or dropping out of the system entirely, compromising the overall model quality.
- Lack of validity checks: There is no built-in mechanism to prevent clients from poisoning the shared model, either with malicious updates or by editing the parameters.
- User unreliability: Each participating device is inherently unreliable due to potential network issues or device power autonomy.

These deficiencies make FL not as efficient and reliable as desired.

1.2 Blockchain

1.2.1 Principles

Blockchain (BC) was released as the underlying technology for Bitcoin 1.0 in 2008 [1]. As suggested by its name, BC is made of a succession of virtually appended blocks.

Each block is built from the following components:

- Hash of the previous block header, responsible for the virtual link between the blocks. The hash can be obtained using secure hashing functions such as SHA256.
- Timestamp, measured in seconds since the initial time.
- Difficulty value, which represents the current hashing target. This is updated depending on the network's overall power to ensure a steady production of new blocks.

- Nonce, the result of the proof-of-work (POW).
- Block body, containing the transaction data.

A particular block in the chain, called the genesis block, contains information such as the protocols and incentives system.

Some nodes in the network are called miners, whose role is to add each new block to the distributed database. To create a new block, miners must find a nonce that, hashed with the current block's content, gives a result with some leading zeros, defined by the difficulty value.

After a block has been created, the nodes will verify all the transactions in the block and add it to their database.

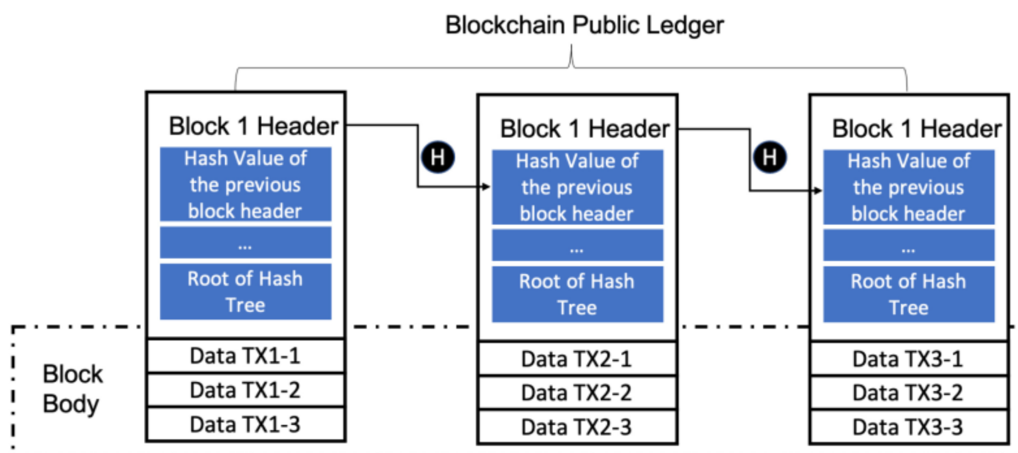


Figure 1.3: Blockchain structure [4].

From the perspective of permission, BC can be defined as:

- Public, when it allows everyone access. These are decentralized since they require no protection from outside attacks.
- Private, when only a selected group of people have access. These are centralized since the organization that owns them dictates who has access and writing permission.
- Consortium, which is a hybrid of the other two. These are partially decentralized, with the consensus mechanism managed by a set of assigned nodes.

Regardless of the type of BC, their workflow is the same.

1.2.2 Security Properties

While BC comes with a built-in incentive mechanism, the dedicated security properties are the focus of this thesis. The main features can be detailed as follows [4]:

- Authentication: This comes from the built-in verification mechanisms.
- Traceability: All transaction information is stored in a block, which is added to the shared ledger.
- High availability: Devices can join and participate whenever.
- Decentralization: The involvement of a central entity is minimal, if not unnecessary, depending on the type of BC.
- Persistence: All transaction data is validated and stored in a public ledger. While it is possible to falsify it, it is not feasible due to the built-in mechanisms.

To better understand the persistence property, let us imagine a malicious user who decides to modify the data contained in block X. By doing this, the block's hash would change, meaning that the previous hash recorded in block X+1 would be inconsistent. This change would be promptly detected once the verification process is triggered to add a new block. To make this kind of attack work, the malicious user would need to recompute the nonce of block X and update the previous hash stored in block X+1, which would change that block's hash, and so on. This means that the attacker would need to update and recompute data for every block after X, which would require a lot of computational power.

This issue is known as the fifty-one percent power attack. The implications of this attack go beyond altering transaction data. By acquiring control over the network, malicious users can double spend, withhold blocks, and overall impede normal functions. Specific consensus mechanisms are needed to prevent the attack, although the details of this go beyond the scope of this work.

Chapter 2

Blockchain-Enabled Federated Learning

As previously discussed, traditional FL does not function as efficiently and reliably as desired: a potential solution is to combine FL with BC. This chapter outlines several possible BCFL architectures, discussing their strengths and weaknesses.

2.1 Integration of Blockchain and Federated Learning

BCFL architectures can vary depending on the type of blockchain used and how the coupling happens.

First, let us discuss architectures depending on the BC type [3] :

- Public chain: This type of blockchain system allows all clients and miners to engage freely, resulting in more potential participants and transparency. To ensure the quality of the model, this system requires validation protocols to prevent any malicious upgrades from being aggregated.

- Private chain: This type of blockchain system allows access only to selected clients based on various factors such as computational resources, past performance, and overall reliability.

Furthermore, the possible couplings are [5]:

1. Fully coupled BCFL: In this model, the clients of FL are also miners in the BC network. Therefore, they provide computational power to train local models, but they also need to mine blocks. It is a fully decentralized approach since the blockchain can be the aggregator. When the global model needs to be updated, a group of users can participate in the aggregation process.

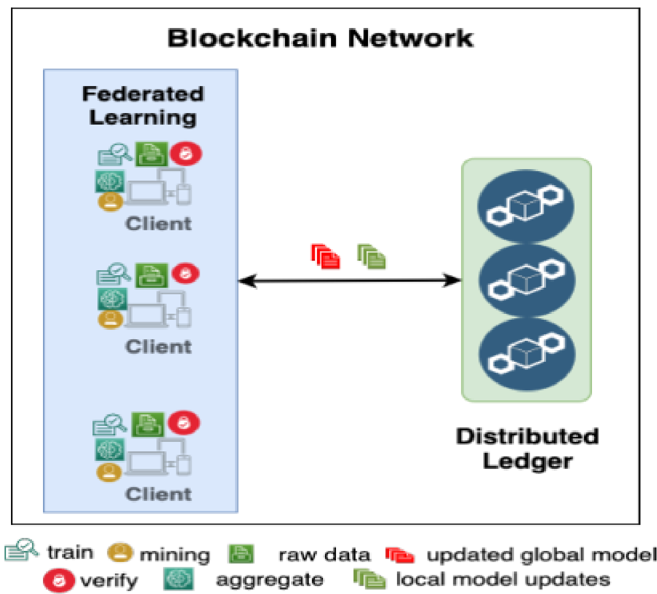


Figure 2.1: Fully coupled BCFL architecture [5].

2. Flexibly coupled BCFL: In this instance, the BC and FL systems are separate networks. The clients only train the local model and compute upgrades, while miners in the BC network aggregate and mine new blocks. An entity is needed to coordinate the two networks.

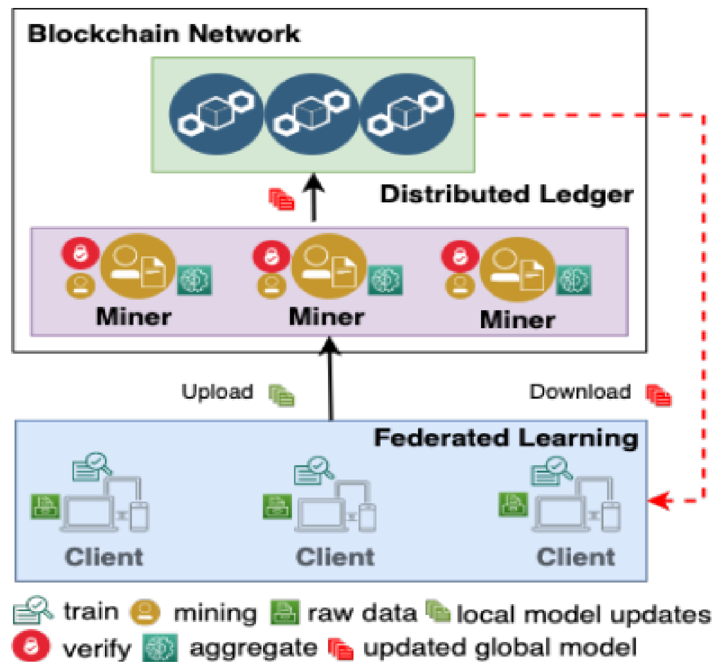


Figure 2.2: Flexibly coupled BCFL architecture [5].

3. Loosely coupled BCFL: In this case, the BC network verifies model updates and keeps track of user reputations. It is a standard FL architecture with reputation and verification features.

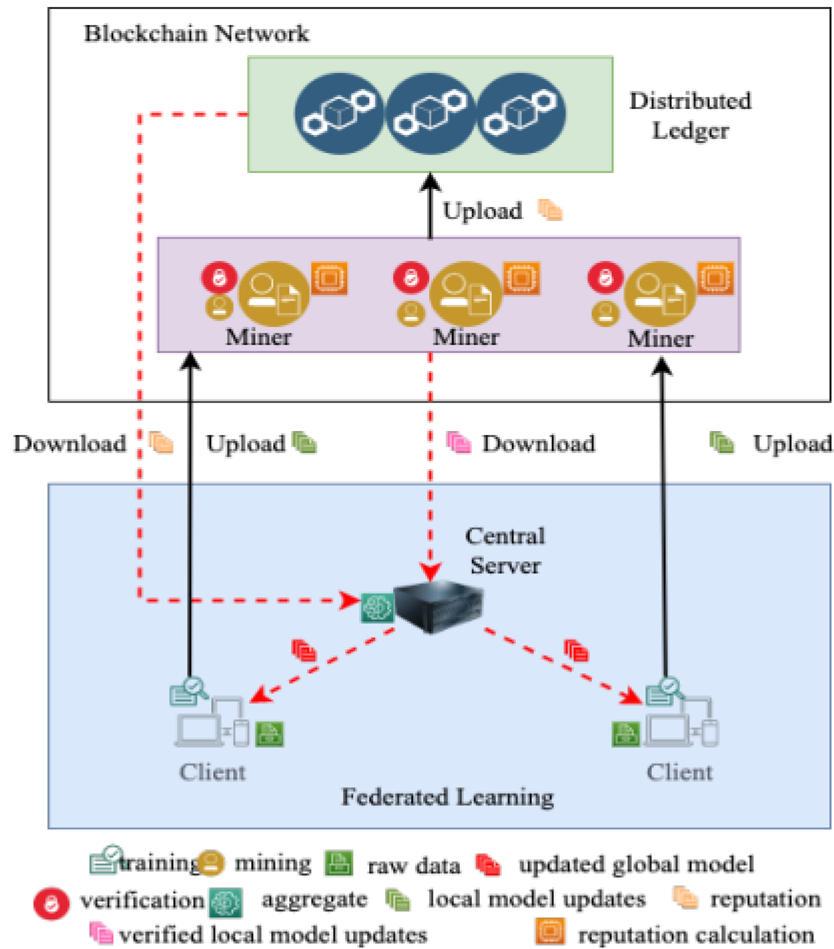


Figure 2.3: Loosely coupled BCFL architecture [5].

Organizations can choose the most appropriate architecture based on their requirements and available resources. For example, public blockchain is optimal if an organization aims to attract as many participants as possible.

2.2 Advantages and challenges

After considering different architectures, it is clear that none offer a solution to all the problems previously listed for FL. Therefore, there is no perfect solution. Let us now discuss the advantages and disadvantages of each possible listed solution:

1. Public blockchain: Allowing anyone to access the network ensures the wide availability of participants, who bring training data and computational power. Furthermore, it allows for full transparency. The downside is that specific validation protocols are needed to mitigate issues related to malicious updates. Moreover, with many potential miners, the

value of the mining difficulty must be increased to ensure steady production of blocks, causing significant consumption of computing resources.

2. Private blockchain: Since only a selected group of users can participate, the need for specific validity and consensus mechanisms is reduced, meaning lighter protocols can be deployed, saving resources. However, this type of system can suffer from a lack of participants and training data.

3. Fully coupled BCFL: This architecture lacks central authority, making it safe from single-point failure. Furthermore, since the transaction data is distributed, should any device disconnect, nothing will be lost, meaning the network is not sensible to each user's unreliability. Nevertheless, the fact that users not only train the local model but also mine blocks add significant computational overhead to the system. This is because the expected users of FL are smartphones and other personal devices, which do not hold much computing power, making complex computing tasks such as proof of work prohibitive.

4. Flexibly coupled BCFL: Since users are no longer expected to mine blocks, allowing miners to complete the task, the computational overhead mentioned in the third point is significantly reduced. This architecture needs a central entity to coordinate, making it vulnerable to single-point failure. Furthermore, latency issues may arise between the two networks.

5. Loosely coupled BCFL: In this case, the BC is only used for validity checks and user reputation management, meaning that FL still relies on a central server. Therefore, single-point failure and secure storage of the global model are not solved. Furthermore, maintaining the two systems separately results in inefficient utilization of resources.

To summarize, when designing a BCFL system, it is crucial to determine the appropriate type of blockchain and architecture to use. This decision should be based on the specific goals and requirements of the system, as well as the expected number of users and their roles.

The third and fourth architectures use the blockchain as a storage for the global model. The model parameters are stored as transaction data, which provides a significant advantage over traditional FL since the BC records the parameters as learning progresses. Additionally, the parameters are easily accessible to all users in the network. Moreover, the BC can be utilized to manage user reputation, prevent single-point failure, and attract more participants, all of which are weaknesses that were previously highlighted when discussing traditional FL. However, using a BC entails significant computation overheads, which increases the overall cost of training models, particularly in the third architecture.

Chapter 3

Implementation

3.1 Description of the environment and tools used

The goal of the simulation is to explore the interaction between the BC and FL, showing some of the features previously discussed. The project operates entirely in a local environment and consists of four classes:

- Block, which handles each block and its mining.
- Blockchain, which handles the virtual links between blocks. Moreover, it manages the verification of the blocks to be added.
- User, detailing personal datasets and training the local model.
- Federated Learning, performing one round of client selection and aggregation of the updates.

The simulation also includes a ”main” file that employs the implemented classes to train the global model over several rounds and a ”.env” file to store environment variables.

Python was used to develop this project due to its widespread support for scientific computing and machine learning libraries. The libraries used are:

1. Numpy 1.26.2: For efficient array and matrix computation, fundamental to the operations of our algorithms.
2. Scikit-learn 1.3.2: In particular, the SGDRegressor module, to leverage its capabilities in regression analysis with stochastic gradient descent.
3. Python-dotenv: Used for managing environment variables.

4. Hashlib: Integral to our Blockchain implementation, providing secure hashing functions necessary for block creation and validation.

5. Matplotlib 3.8.0: For visualizing data and model performance.

The personal computing environment has the following hardware and software specifications:

- Hardware:
  - CPU: 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz
  - GPU: Nvidia RTX 3050, 4GB GDDR6. Though available, it was not utilized for computations.
  - RAM: 16GB DDR4 3200MHz
  - Storage: 512GB SSD
- Software:
  - Operating System: Windows 11 Home
  - Python Version: 3.11
  - Library Versions: Listed above, important for ensuring reproducibility and consistency in project execution.
  - Development Tools: The code was primarily developed using Pycharm 2023.2.5.
  - Conda: Deployed for environment and dependency management.

3.2 Overview of the Blockchain Implementation

Our blockchain implementation consists of a sequence of blocks, each containing the fields discussed in Chapter 1. The transaction data field is used to store the model's parameters. The BC implementation employs two classes: Block and Blockchain, which will be detailed below. First of all, let us discuss the Block class. When first created, each block has nonce set to zero, and the hash is calculated accordingly using the "calculate hash" method. If the nonce value changes, the hash will not be automatically calculated. Therefore, whoever completes the POW will also need to recalculate it.

```
def __init__(self, index, timestamp,
                 data, previous_hash):
    self.index = index
```

```
        self.timestamp = timestamp
        self.data = data
        self.previousHash = previous_hash
        self.nonce = 0
        self.hash = self.calculate_hash()

    def calculate_hash(self):
        hash_object = hashlib.sha256()

        # Convert each string to base64 before hashing
        hash_object.update(base64.b64encode(
            str(self.index).encode()))
        hash_object.update(base64.b64encode(
            str(self.timestamp).encode()))
        hash_object.update(base64.b64encode(
            str(self.data).encode()))
        hash_object.update(base64.b64encode(
            str(self.previousHash).encode()))
        hash_object.update(base64.b64encode(
            str(self.nonce).encode()))

        calculated_hash = hash_object.hexdigest()
        return calculated_hash
```

The POW method computes the block's nonce depending on the given difficulty value. Moreover, it returns the total time spent to complete the task, which will be used for performance analysis.

```
    def proof_of_work(self, difficulty):
        start = time.time()
        while self.hash[:difficulty] != "0" * difficulty:
            self.nonce += 1
            self.hash = self.calculate_hash()
        end = time.time()
        return end - start
```

At last, there are two more methods: one for printing the block, which can be useful for testing, and one to obtain the data stored by the block.

```
def print_block(self):
```

```
        print("Block " + str(self.index) + ":")
        print("Timestamp: " + str(self.timestamp))
        print("Data: " + str(self.data))
        print("Nonce: " + str(self.nonce))
        print("Hash: " + str(self.hash))
        print("Previous hash: " + str(self.previousHash))
        print()

    def get_data(self):
        return self.data
```

As for the BC, when first created it only contains the genesis block, which is obtained by calling the relevant method.

```
    def __init__(self, timestamp, data, difficulty):
        self.difficulty = difficulty
        self.chain = []
        self.create_genesis_block(timestamp, data)

    def create_genesis_block(self, timestamp, data):
        self.chain = np.array(
                    [Block(0, timestamp, data, "0")])
        self.chain[0].proof_of_work(self.difficulty)
```

New blocks can be added to the BC by calling the "add block" method, which follows these steps:

1. Block creation.
2. Block mining.
3. Chain validation. The new block gets added only if the check succeeds.

```
    def add_block(self, time, data):
        new_block = Block(self.get_chain_length(), time,
                    data, self.get_latest_block().hash)
        run_time = new_block.proof_of_work(self.difficulty)
        if self.is_chain_valid():
            self.chain = np.append(self.chain, new_block)
        return run_time
```

The chain gets validated through the relevant method, which checks the consistency of each block's hash, and the correspondence between each block's previous hash parameter and the previous block. If either of those were to fail, a message would be printed.

```
def is_chain_valid(self):
        for i in range(1, self.get_chain_length()):
            current_block = self.chain[i]
            previous_block = self.chain[i - 1]
            calc_current_hash = current_block.calculate_hash()
            if current_block.hash != calc_current_hash:
                print("Invalid hash for block " + str(i))
                return False
            if current_block.previousHash != previous_block.
               hash:
                print("Invalid previous hash for block " + str(
                    i))
                return False
        return True
```

At last, two more methods return the latest block in the chain and the chain's length, respectively.

```
def get_latest_block(self):
        return self.chain[self.get_chain_length() - 1]

    def get_chain_length(self):
        return len(self.chain)
```

Before discussing the FL implementation, it is necessary to understand some of the relevant features of the implemented BC. Firstly, the proof of work method is called when a block is created. However, it does not identify who is involved in completing the task. In real-life scenarios, multiple miners compete to complete the task and receive rewards according to their contribution. For simplicity, this implementation does not handle the interaction between miners. Instead, it allows what could be pictured as one miner to complete the entire process. Moreover, the chain validation method scans the entire chain linearly, verifying every transaction. This was done for simplicity, as in a realistic BC, transaction data is distributed among the nodes in the network and not stored in a central array like in this implementation. Furthermore, if the chain isn't validated a message is printed, but no action is taken. At last, the implementation does not deploy a method for dynamically updating the difficulty value, making it a constant. This is not the case in a realistic BC, where the difficulty gets updated to ensure steady block production. Overall, these simplifications were made to shift the focus of the work towards the

interaction between BC and FL rather than on the specifics of BC. Nonetheless, these issues need to be addressed to deploy a realistic implementation.

3.3 Overview of the Federated Learning Implementation

First of all, let us discuss the implementation of the User class. Each user object contains three fields: a unique identifier, the dataset, and the local model.

```
def __init__(self, id, dataset):
        self.dataset = dataset
        self.id = id
        self.local_model = SGDRegressor()
```

The training method handles the training of the local model, performing a partial fit over the available dataset.

```
    def train(self):
        local_epochs = user_dataset_size // 2
        for i in range(local_epochs):
            self.local_model.partial_fit(self.dataset[:, 0].
               reshape(-1, 1), self.dataset[:, 1].ravel())
```

At last, there is a method to obtain the local model's parameters.

```
    def get_parameters(self):
        return self.local_model.coef_, self.local_model.
           intercept_
```

Let us proceed to the Federated Learning class, which includes two methods: standard FL and BCFL. For now, we will only discuss the standard one. The method follows these steps:

1. User selection: Performed at random.
2. Local model training.
3. Update generation.

```
def federated_learning(global_model, users):
    labels = np.arange(len(users))
    # train model with m users participating over each epoch

   m = np.random.randint(2, len(users))
```

```python
    users_to_participate = np.random.choice(labels, m, replace=
        False)

    # Create arrays to store the sum of the predictions of each
        User, shaped like the global model
    predict_a = 0.0
    predict_b = 0.0

    # train each User
    for j in users_to_participate:
        users[j].train()
        # Add the difference between the local model and the
            global model to the prediction
        predict_a += users[j].local_model.coef_ - global_model.
            coef_
        predict_b += users[j].local_model.intercept_ -
            global_model.intercept_

    return predict_a / m, predict_b / m
```

3.4 Interaction between Blockchain and Federated Learning Structures

The architecture used in this simulation is flexibly coupled BCFL (figure 2.2). We will assume, for simplicity, that:

- Users are selected to ensure their reliability, where a user is considered reliable if they provide truthful data and will not disconnect during model training.
- The size of users' datasets is the same.
- The BC is private, which allows us to control who has access to the network. This enables us to prevent malicious entities from accessing. Moreover, it allows user selection depending on computational power, which helps prevent the fifty-one percent power attack, making the network secure without requiring other consensus mechanisms.

The simulation runs, as previously stated, on a fully local environment and makes use of the following environment variables:

- Epochs: Representing the number of BCFL rounds that will be performed.
- User dataset size.
- Max number of users.

The BCFL method executes one round of FL, following the steps:

1. Retrieval of the global model parameters, which are stored in the latest block.
2. Model training, by calling the standard FL method.
3. Aggregation: Performed by the entity coordinating the two networks. In this case, whoever is running the simulation acts as the coordinator.
4. Addition of the new block to the BC.

```
def block_chain_federated_learning(block_chain, users):
    params = block_chain.get_latest_block().get_data()
    global_model = SGDRegressor()
    global_model.coef_ = params[0]
    global_model.intercept_ = params[1]
    coef_update, intercept_update = federated_learning(
        global_model, users)
    runTime = block_chain.add_block(time.time(),
                                    [global_model.coef_ +
                                        coef_update,
                                        global_model.intercept_
                                        + intercept_update])
    return block_chain, runTime
```

The simulation starts automatically by running the "main" function. The process involves the following steps:

1. BC creation and initialization: The genesis block contains an array of zeros as data. This is an arbitrary choice and could have been set to any value.
2. User generation: The details of this will be explained later.
3. Model training.

Furthermore, as training proceeds computational times are recorded. These will be used for performance analysis.

```
if __name__ == '__main__':
    # create blockchain
    block_chain = BlockChain(time.time(), np.array([0.0, 0.0]))

    users, a, b = generate_users()

    start = time.time()
    total_pow_time = 0
    # perform federated learning
    for current_epoch in range(epochs):
        block_chain, pow_time = block_chain_federated_learning(
            block_chain, users)
        total_pow_time += pow_time

    end = time.time()
    print("Time taken: " + str(round(end - start, 2)) + "
       seconds")
    print("Time taken for proof of work: " + str(round(
       total_pow_time, 2)) + " seconds")
    print("Proof of work overhead: " + str(round(total_pow_time
        / (end - start), 2) * 100) + "%\n")
```

The user generation step is performed by using another available method, which:

1. Selects the number of users to participate.
2. Generates the goal parameters.
3. Generates a dataset.
4. Creates and stores an array of users, splitting the dataset among them.
5. Returns the goal parameters and an array containing all the users.

```
def generate_users():
    # Generate an array of users of random size n
    n = np.random.randint(1, max_number_of_users)

    # generate random a and b, y = a*x + b
    a = np.random.rand()
```

```
    b = np.random.rand()

    # generate a dataset to be split among users
    x = np.random.rand(user_dataset_size * n, 1)
    y = a * x + b + np.random.normal(0, 0.1, (user_dataset_size
        * n, 1))
    dataset = np.stack((x, y), axis=1)

    users = []
    for i in range(n):
        users = np.append(users, User(i, dataset[i *
            user_dataset_size:(i + 1) * user_dataset_size]))
    return users, a, b
```

At last, there is a method that can be used to attempt to attack the BC, intending to test the BC's immutability property. After a block has been modified the change will be detected by the validation method, which will be called when attempting to add a new block.

```
def attack_blockchain(block_chain, index):
    print(”Do you wish to attack the blockchain? (y/n)”)
    if input() == 'y':
        block_chain.chain[index].data = np.array([-100, -100])
        print(”Do you also wish to re-compute the nonce of the
            block? (y/n)”)
        if input() == 'y':
            block_chain.chain[index].hash = block_chain.chain[
                index].calculate_hash()
            block_chain.chain[index].proof_of_work(block_chain.
                difficulty)
            print(”In this case the change will be detected
                when checking the next block (” + str(index + 1)
                 + ”)”)
        else:
            print(”Without re-calculating the nonce the change
                will be detected when checking the hacked block
                itself.”)
        print(”Block ” + str(index) + ” has been attacked”)
    return block_chain
```

Chapter 4

Results

To ensure that the results presented in this chapter are reproducible, we used the following values to create the graphs:

```
epochs=40
user_dataset_size=10
max_number_of_users=100
```

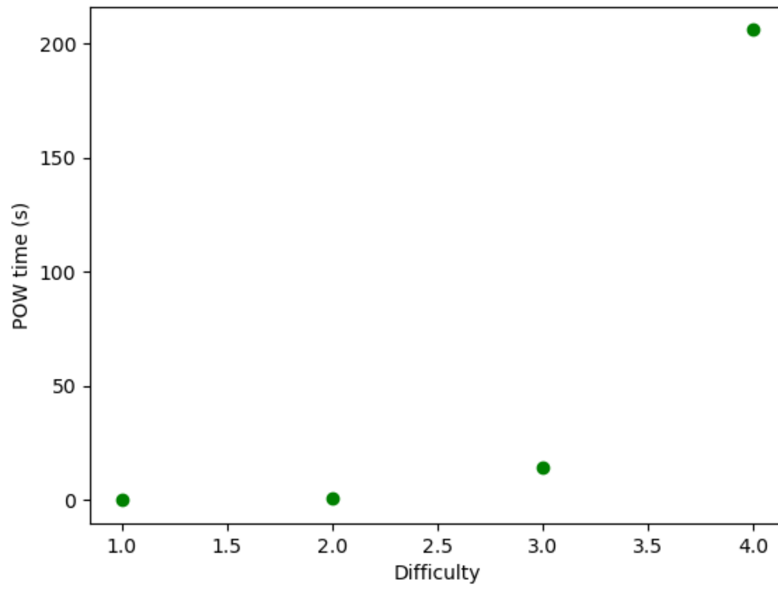
The dataset used in our simulations was created by considering the linear relationship $y = ax + b$, where "a" and "b" were generated at random. To make the dataset more realistic, normal distributed noise was added. The features were generated using a uniform distribution, and their corresponding targets were calculated using the linear relationship, resulting in a linear regression problem.

As for preprocessing, we stacked the features and targets together to form a unified dataset, which is later split equally among the users. Since the dataset was specifically generated, no further preprocessing was required.

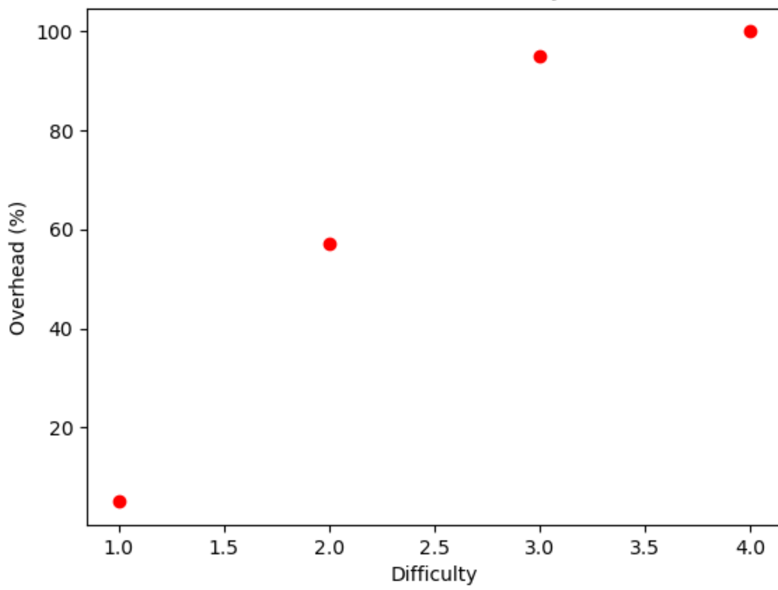
At last, before moving on to the results, let us review the meaning of the difficulty value in this context. The difficulty value indicates the number of leading zeros each block's hash must have once POW is completed. Increasing the difficulty makes finding a suitable nonce a more lengthy task.

The exploration of BCFL through our implementation sheds light on several aspects previously discussed. One of the most striking observations is the POW's impact on total computation time, which we found to increase exponentially with the difficulty level, as shown in Figure 4.1a. Meanwhile, the computation time associated with standard FL remains relatively constant. This distinction highlights how the mining process introduces a significant overhead as the difficulty increases, as shown in Figure 4.1b.

This increase in runtime does not impact the model's quality. This is evidenced when evaluating its Mean Statistical Error (MSE) across varying difficulty levels. Plot 4.2a reveals a consis-



(a) POW time.



(b) POW overhead.

Figure 4.1: Effect of POW over runtime.

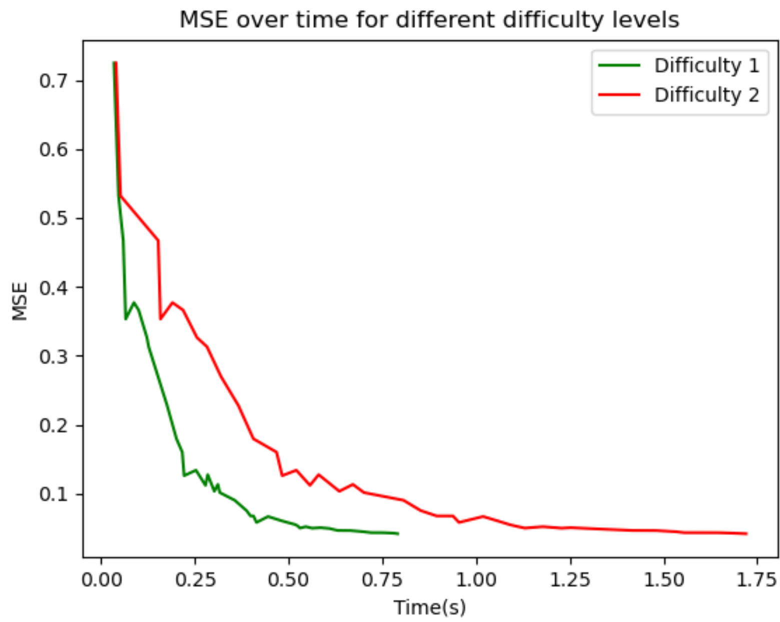
tent start and progression across the models, regardless of the difficulty level; the differentiating factor lies solely in the time taken to achieve the result. Furthermore, plot 4.2b underlines the weight this added computation time has on the process: the network with difficulty equal to four takes significantly longer to complete the same task.

Beyond the additional computation time, Blockchain brings Federated Learning some indispensable attributes. Within our implementation, the most significant is immutability – Blockchain acts as an immutable record of the model's progression over time. To verify the property, we employed the "attack blockchain" method. In the following example, we tried to alter the content of the fifth block while also recomputing its nonce and hash accordingly. Upon attempting to add a new block, the validation process verifies the consistency of the entire BC. As expected, no issue arises when checking the hacked block since all its data is technically correct. The problem is detected when checking block number six: its previous hash parameter does not correspond to the one stored in block five, letting the network know there has been an attack.

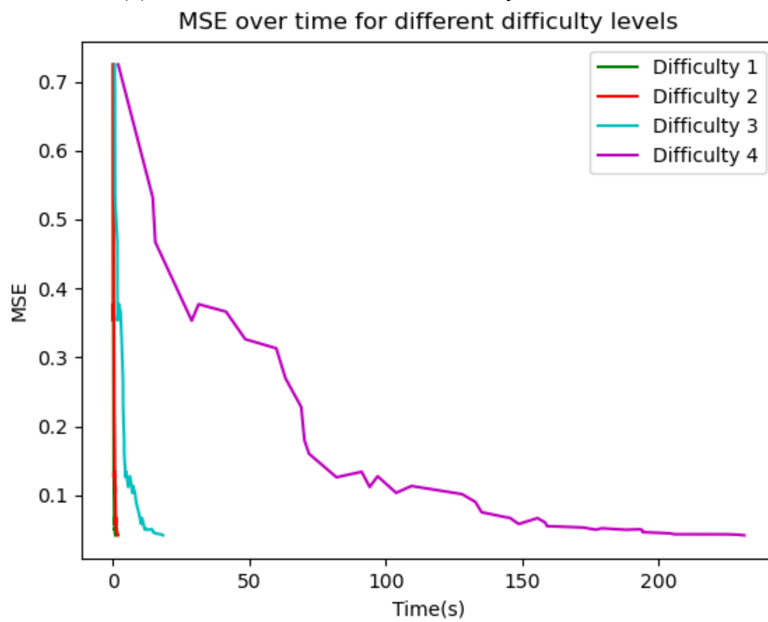
```
Attacking block 5
Block before attack:
Block 5:
Timestamp: 1708452872.5109034
Data: [0.29934605], [0.53717539]
Nonce: 13
Hash:
    00ba02d9cef36402982bdfa6fc6593bf8f790a40fe0b058160eedf74de1e468e
Previous hash:
    00ee705f42ad04f7bcd95367a6cc0ac4f52de06ac5b6cf0d740784126102854d

Block after attack:
Block 5:
Timestamp: 1708452872.5109034
Data: [-100], [-100]
Nonce: 801
Hash:
    00e11037e002b3b8b4a9f67b32d2da688ff3e7fc126340f95a657c7438820b3b
Previous hash:
    00ee705f42ad04f7bcd95367a6cc0ac4f52de06ac5b6cf0d740784126102854d

Invalid previous hash for block 6
```



(a) Plot of the MSE over difficulty one and two.



(b) Plot of the MSE over difficulty up to four.

Figure 4.2: MSE plots.

```
Block 6:
Timestamp: 1708452872.5459032
Data: [0.32329815], [0.57810183]
Nonce: 159
Hash:
    005a4fdfab0b5ce0e5eb501d1cb4139a9a8df9dd9d65fd306813b4be1e122049
Previous hash:
    00ba02d9cef36402982bdfa6fc6593bf8f790a40fe0b058160eedf74de1e468e
```

This immutable record fortifies the overall system against manipulative attempts, thus ensuring data integrity within the FL paradigm.

In summary, this simulation reveals that while undoubtedly integrating BC into FL adds significant computational overheads, it does not detract from the model's quality. Moreover, BC introduces FL to an invaluable layer of security, protecting the model's parameters from manipulation.

Chapter 5

Conclusions

This thesis has explored the integration of Blockchain technology with Federated Learning, highlighting the potential to enhance the security and decentralization of machine learning models. Through the simplified yet comprehensive implementation of both systems, we have demonstrated the effectiveness of their interaction. Moreover, the trade-off between the cost to complete the Proof of Work and the potential brought by Blockchain was highlighted.

For future research, it is imperative to explore strategies to ensure the reliability of users. Furthermore, deploying the simulation outside of a local environment would allow further insight into the potential communication delays and issues between the two systems.

Bibliography

[1] Nakamoto S Bitcoin. *Bitcoin: A peer-to-peer electronic cash system*. 2008.

[2] Brendan McMahan et al. “Communication-Efficient Learning of Deep Networks from Decentralized Data”. In: *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. Ed. by Aarti Singh and Jerry Zhu. Vol. 54. Proceedings of Machine Learning Research. PMLR, 20–22 Apr 2017, pp. 1273–1282. url: `https://proceedings.mlr.press/v54/mcmahan17a.html`.

[3] Dinh C. Nguyen et al. “Federated Learning Meets Blockchain in Edge Computing: Opportunities and Challenges”. In: *IEEE Internet of Things Journal* 8.16 (2021), pp. 12806–12825. doi: `10.1109/JIOT.2021.3072611`.

[4] Youyang Qu et al. “Blockchain-enabled Federated Learning: A Survey”. In: *ACM Comput. Surv.* 55.4 (Nov. 2022). issn: 0360-0300. doi: `10.1145/3524104`. url: `https://doi.org/10.1145/3524104`.

[5] Zhilin Wang and Qin Hu. “Blockchain-based federated learning: A comprehensive survey”. In: *arXiv preprint arXiv:2110.02182* (2021).