unibz

Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Fakultät für Informatik
Facoltà di Scienze e Tecnologie informatiche
Faculty of Computer Science

Master in Computational Data Science

Master Thesis

Federated Reinforcement Learning on the Edge

Candidate: Hannes Wiedenhofer

Supervisor: Dr. Roberto Confalonieri

Co-Supervisor: Prof. Antonio Liotta

September, 2021

Abstract

Motivation As both Internet of Things and the preservation of privacy become more important, implementing these technologies in the field of machine learning brings various advantages. It allows us to process the collected data on edge devices without sending it over the network, decreasing the communication volume and decreasing the threat of privacy breaches.

Problem statement The goal is to create a conceptual federated reinforcement learning framework that works on the edge. Memory and computational restrictions of edge devices must be taken into consideration, while the resulting model quality should be as high as possible.

Approach This thesis developed a reinforcement learning system in different increasingly complex modelling rounds, each of them representing a different structural approach.

Results The result is a conceptual federated reinforcement learning framework that works on the edge. Additionally, the thesis presents a comparison of the resulting model quality, advantages and disadvantages of different structural approaches.

Conclusions The result of this thesis serves as a basis for implementing a fully federated reinforcement learning on the edge.

Acknowledgements

I would like to thank my supervisors, Dr. Confalonieri and Prof. Liotta for their availability, advice, continuous support and patience during the process of writing this thesis.

Also, I would like to thank my colleagues, friends, and family for being understanding and supporting me throughout my studies.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Edge computing and federated learning represent some of the most exciting and promising technologies of the last years. With the emergence of Internet of Things and the rise of privacy-consciousness, federated learning on the edge represents an important technology whose popularity may increase in the next years. This thesis combines federated learning on the edge with reinforcement learning for solving a rebalancing problem. In particular, we aim at solving a bike rebalancing problem within a bike sharing system. As bike sharing systems expand and increase the number of bikes and stations asymmetric shifts of bikes occur from popular origin stations to common destinations. Intelligent systems are required to rebalance the bikes and make sure each station has the required number of bikes.

1.1 Motivation

Reinforcement learning [1] is a machine learning technique that allows an agent to learn by trial and error. It is one of the three basic machine learning paradigms next to supervised learning and unsupervised learning. Typically, reinforcement learning is based on Markov Decision Processes. Different from supervised learning, reinforcement learning does not require labelled input/output pairs. The focus is on finding a balance between exploration (undiscovered knowledge) and exploitation (current knowledge). Reinforcement Learning is a general framework that can solve complex tasks without any prior knowledge. That is why it is used in many different applications such as robotics, business strategy planning, data processing, and aircraft control.

Edge Computing [7] is the decentralized data processing on so called "edge devices". Edge devices are devices on the edge of the network, as opposed to devices in the center of the network such as servers. Examples for edge devices are smartphones, IoT devices, and sensors. Edge computing enables us to process and analyze data in real-time without having to send large amounts of data over the network.

Federated Learning [8] is a decentralized way of performing machine learning. A global model is trained using multiple devices. Each device has its own data that is not shared with anybody else. Instead, only model parameters are shared. Using these parameters from all devices, the server aggregates them and updates the global model. This is a way of training machine learning models while ensuring data privacy, data security, and data access control.

A combination of these technologies enables us to create a solid machine learning model for multiple agents in a potentially complex network while respecting the principles of data

privacy as federated learning does not share data explicitly over the network. Data can still be inferred by intercepting the exchanged model parameters, but federated learning makes this more difficult for attackers. Data privacy can be increased by implementing additional privacy preserving technologies. Similar approaches may be necessary in the future as people are becoming more aware of data privacy and the implied consequences.

This thesis offers a comparison between different reinforcement learning approaches and provides the basis for implementing a fully federated reinforcement learning system on the edge.

1.2 Objective

The objective of this paper is to provide a conceptual federated reinforcement learning on the edge. This involves three different areas: federated learning, reinforcement learning, and edge computing. This paper also analyzes the problem from a practical point of view, examining existing federated learning frameworks and highlighting their strengths and weaknesses. The paper proposes a new federated learning framework for a specific use case, namely bike rebalancing within a bike sharing system. For this goal, different approaches are presented and compared. Finally, this paper lays the ground work for developing a fully federated reinforcement learning system on the edge.

1.3 Approach

We reviewed the state of the art, analyzing different papers and comparing their approaches. The most common federated learning frameworks were analyzed and compared. We attempted to use these frameworks implementing simple machine learning models, without any success because they are in an early phase of development and therefore contain bugs and do not provide a decent documentation. Therefore, we decided to implement our own framework based on code from a github repository[1]. We started with the implementation of a centralized approach, then extended it for a distributed approach, and finally we implemented the basis for a federated approach. We evaluated the three approaches, comparing their strengths, weaknesses and most importantly the resulting model quality.

1.4 Structure of the thesis

This thesis is divided into eight chapters:

- Chapter 1 introduces the problem and describes the objective of the thesis
- Chapter 2 defines the different technologies we used
- Chapter 3 describes the problem we want to solve
- Chapter 4 reviews similar papers and analyzes and compares different federated learning framework
- Chapter 5 describes our proposed solution in detail, Chapter 6 evaluates and analyzes the results of our proposed solution

[1] https://www.citibikenyc.com/system-data

- Chapter 7 summarizes our work
- Chapter 8 shows the implication of our work on future studies and implementations

Chapter 2

Background Information

This thesis combines various different technologies within the realm of Machine Learning. In the following we give an overview of the used technologies, defining terminology, and describing key concepts.

2.1 Reinforcement Learning

Reinforcement Learning is an area of Machine Learning which enables an agent to learn by trial and error in a potentially complex environment. Each action has a consequence in terms of reward. Based on the rewards given by the environment, the agent aims to learn the optimal policy within this environment. Terminology:

- Agent: acts within the environment and learns based on its actions.
- Environment: everything the agent interacts with, either directly or indirectly. It takes the agent's current state and action as input and returns the agent's reward and its next state.
- Action: the agent's means of interacting with the environment. Every action yields a reward from the environment.
- State: the situation an agent finds itself in.
- Reward: the feedback given from the environment to the agent. Can be positive or negative. The reward affects the agent's behaviour.
- Policy: a mapping from a state to the probabilities of selecting each possible action given that state.

Reinforcement Learning algorithms can be divided into two types: on-policy and off-policy. On-policy learning algorithms evaluate and improve the same policy which is being used to select actions. In off-policy settings the agent's experience is added to a data buffer, and each new policy collects additional data. All of this data is used to train an updated new policy.

The first mention of reinforcement learning probably dates back to [9], released in 1963. The author proposed a "trial-and-error device" which learns to play the game of Tic-tac-toe. This device was initially constructed using matchboxes and subsequently simulated

on a Pegasus 2 computer. The author's work builds on concepts of psychology that suggest that the strength of reinforcement becomes less as we go back further in time. Whenever the device performed well, positive rewards were assigned to it. Otherwise, the device was assigned negative rewards. This way, the device was able to obtain better results with every iteration.

The most seminal piece of literature about reinforcement learning is considered to be [1]. It represents the reinforcement learning environment in form of Finite Markov Decision Processes. The authors describe MDPs as "formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations or states, and through those future rewards". Figure 2.1 shows the interaction between the agent and the environment in a reinforcement learning setting. The agent performs an action within the environment. Based on this action, the environment's state changes and it assigns a reward to the agent. Based on the new state and the reward received, the agent executes a new action. This cycle continues until a certain goal or stopping condition is met.

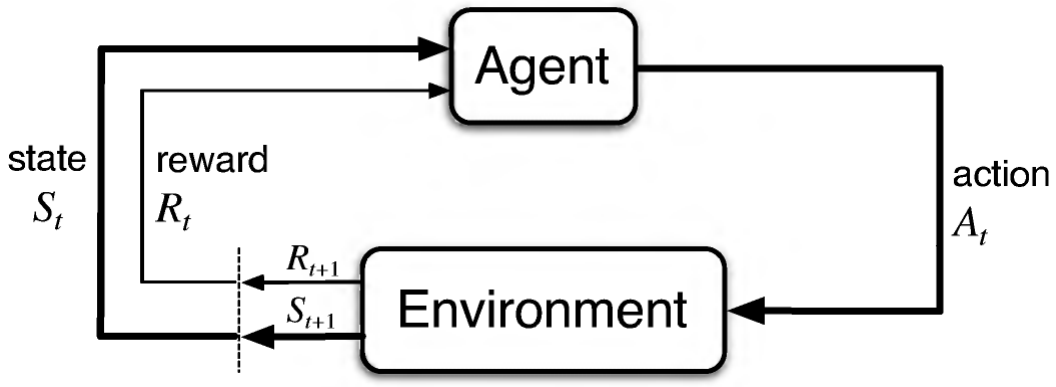

Figure 2.1: The agent-environment interaction in a Markov decision process (Taken from [1]).

2.2 Q-Learning

Q-learning was first introduced by [10]. It is an off-policy reinforcement learning algorithm for discrete action and state spaces. It does not require a model of the environment and is therefore "model-free". For any finite Markov decision process it finds a policy that is optimal in the sense that it maximizes the expected value of the total reward over any and all successive steps, starting from the current state. The core of the algorithm is the Bellman equation as represented in Figure 2.3. This is what the "Q" in Q-learning refers to.

An example for a problem that can be solved by q-learning is the so called "taxi problem" in which at a random state, our job is to drive the taxi to the passenger's location, pick up the passenger, drive to the passenger's desired destination, and drop the passenger off. At any time, the taxi, which represents the agent, can perform six actions: 0 - move south, 1 - move north, 2 - move east, 3 - move west, 4 - pick up passenger, 5 - drop off passenger. As can be seen in Figure 2.2 the taxi moves in a grid world with 5 rows and 5 columns. The letters represent pick-up and drop-off locations. The green lines represent walls and there-

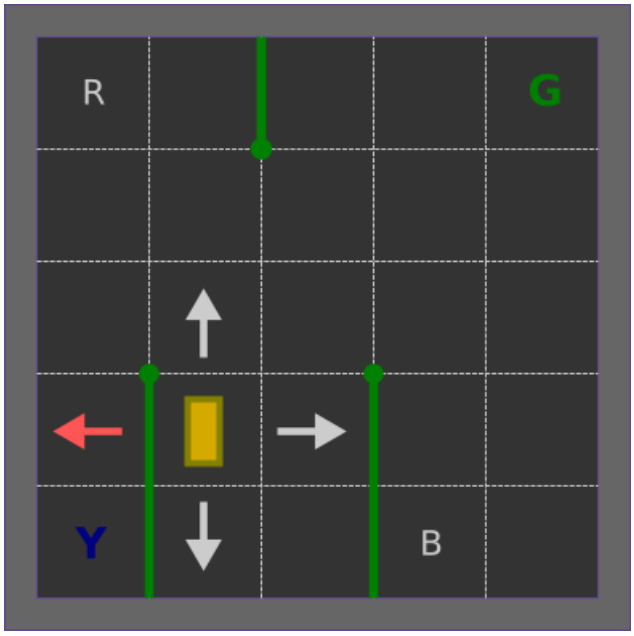

Figure 2.2: OpenAI taxi-V3 environment(Image by Guillaume Androz from towardsdata-science).

$$Q^{new}(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}}^{\text{new value (temporal difference target)}} - \underbrace{Q(s_t,a_t)}_{\text{old value}}\Big)}^{\text{temporal difference}}$$

Figure 2.3: The Bellman equation (Adapted from [1]).

fore introduce illegal actions. The passenger can be located in 4 points of the grid world, and the passenger's destination can be located in 4 different points. Each state is defined by a tuple with 4 entries: (taxi row, taxi col, passenger location, destination). This results in 400 possible states. The taxi receives +20 points for a successful drop-off, and loses 1 point of every time step it takes. Illegal drop-offs and pick-ups are punished by a reward of -10. An example for a q-table for this problem can be seen in Figure 2.4.

It is a weighted sum of the old q-value and the new information, where r_t is the reward received by moving from state s_t to state s_{t+1}.$Q^{new}(s_t,a_t)$ is the sum of three factors:

- $(1-\alpha)Q(s_t,a_t)$: the old value weighted by the learning rate. Values of the learning rate close to 1 make the changes in Q more rapid.
- αr_t: the reward $r_t = r(s_t,a_t)$ to obtain if action α_t is taken when in state s_t, weighted by the learning rate
- $\alpha\gamma \max_a Q(s_{t+1},a)$: the maximum reward that can be obtained from state s_{t+1}, weighted by the learning rate and the discount factor)

When s_{t+1} is a final or terminal state, the episode ends.

2.2.1 Variables

- Learning rate: defines the relation between exploitation of prior knowledge and exploration of new possibilities. When set to 0, the agent does not learn anything new and relies on prior knowledge. When set to 1, the agent does not take into consideration prior knowledge. In practice, often a constant learning rate is used, such as 0.1.

- Discount factor: determines how much weight is given to future rewards. When set to 0 the agent will only consider current rewards, while a number close to 1 will make it strive for a high reward in the long-term.

2.2.2 Q-Tables

The values calculated using the Bellman equation from Figure 2.3 are inserted into q-tables. The first column of a q-table represents the state space. The state space is the set of all states the agent encounters. In standard q-learning the state space must be discrete. If the state space is continuous, function approximation should be applied. The other columns represent the action space. The action space is the set of all possible actions the agent can execute. Figure 2.4 represents the-qtable for the taxi problem described earlier. The action space consists of the following actions: South(0), North(1), East(2), West(3), Pickup(4), Dropoff(5). The q-table is initialized with all values set to 0. Then, these values are updated based on the agent's actions and the assigned rewards via the Bellman equation. The opti-

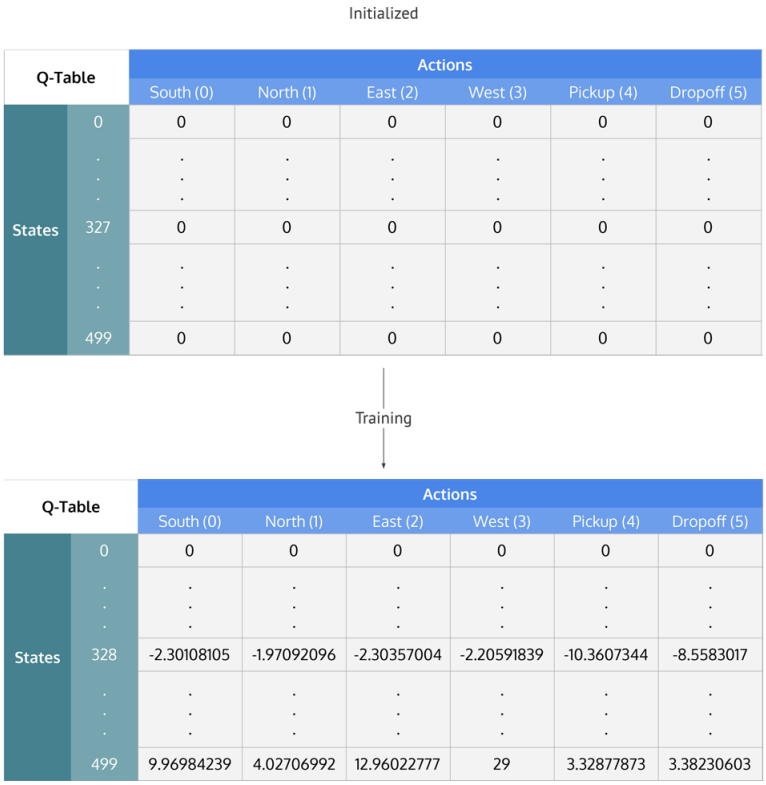

Figure 2.4: Example of a Q-table (Image by LearnDataSci on Wikimedia).

mal policy consists of choosing, given a state, the highest corresponding value in the state's row and executing the associated action.

2.3 Federated Learning

Federated leaning[8] is a decentralized form of Machine Learning. It aims at training machine learning algorithms on multiple local datasets without explicitly exchanging data. Multiple local models are trained and parameters are exchanged between them to create a single global model shared by all nodes. This enables multiple actors to build a common model without sharing data while respecting data privacy, data access rights, and data security. There exist two common types of federated learning as you can see in Figure 2.5.

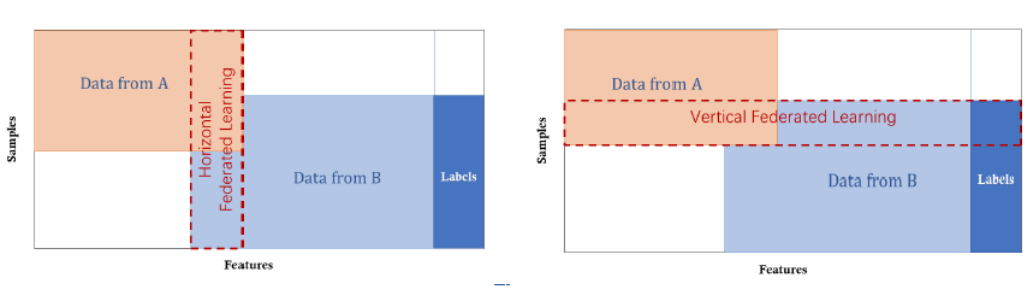

Figure 2.5: Horizontal and Vertical Federated Learning.

- Horizontal Federated Learning: all the devices have datasets with the same feature space, but different sample space. An example is a bank that operates in two different regions. The bank will collect the same data from clients in both regions, but the clients in one region will not be the same as the clients in the other region. This is the most common federated learning approach.

- Vertical Federated Learning: all the devices have datasets with the same sample space, but different feature space. An example is an online streaming service and an online book store that have the same customer, and use their data to recommend the customer new items based on the customer's history on both platforms.

An example of applied federated learning is Google's Gboard application[1]. It is a keyboard for mobile devices. In [2] the authors apply federated learning for keyboard predictions. As we can see in Figure 2.6, the goal is to suggest words based on the words already typed by the user. In this case, for the words "I love you", words like "so much", "too", and "and" are suggested.

The next word prediction is achieved by training a recurrent neural network using a federated learning framework. In Figure 2.7 we see the workflow of the federated learning framework. In A, the client devices compute stochastic gradient descent updates on locally-stored date. In B, a server aggregates the client updates to build a new global model. In C, the new model is sent back to the clients, and the process is repeated. In this way the model

[1] https://play.google.com/store/apps/details?id=com.google.android.inputmethod.latin&hl=en&gl=US

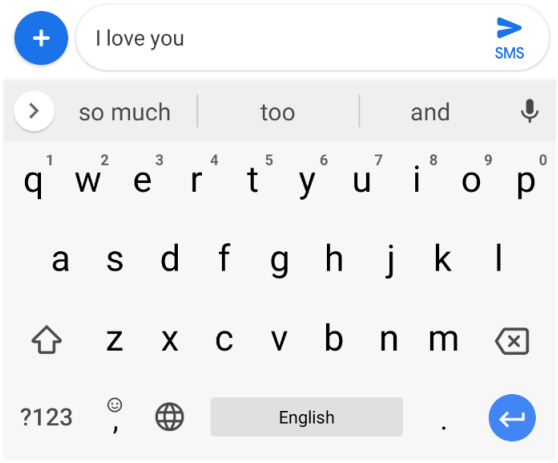

Figure 2.6: Next word predictions in Gboard (Taken from [2]).

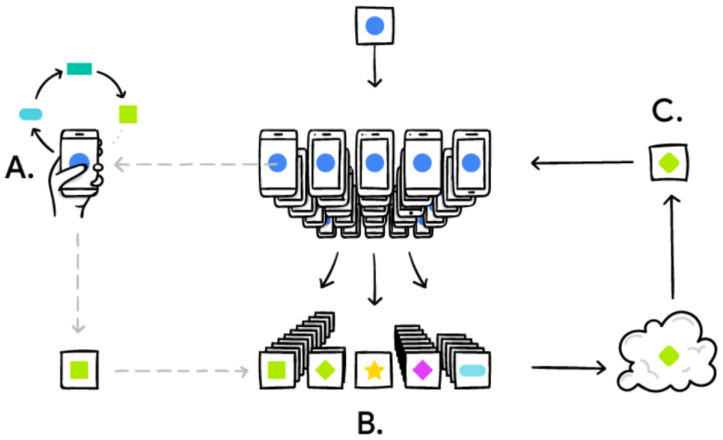

Figure 2.7: An illustration of the federated learning process (Taken from [2]).

stays up to date and is able to predict the next words based on all users' behavior while still respecting everybody's privacy. The authors showed that their federated model can outperform an identical server-trained model. This represents one of the first applications of federated language modeling in a commercial setting.

Another example of the Gboard keyboard shows the data privacy aspect of federated learning as can be seen in Figure 2.8. The engineer wants to know the frequency of the words "hello" and "world" among all the Gboard users. He does not need to know the frequency for a single user. The word count of each user are collected by the server, combined and summed. This way the engineer does not have access to a single user's word count data, but only to the aggregated data from all the users.

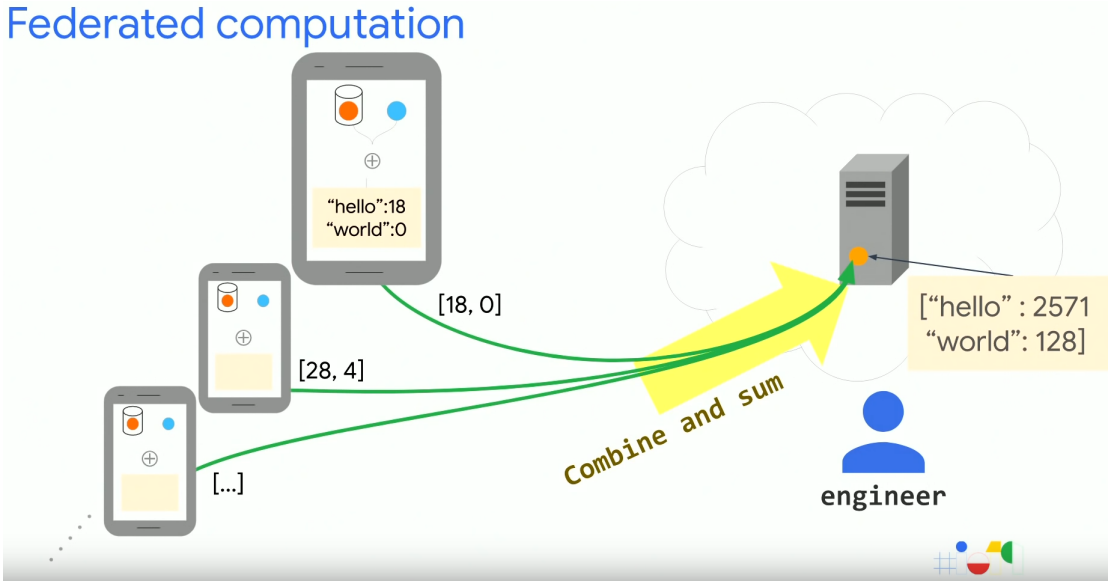

Figure 2.8: Federated computation word count example (Taken from Tensorflow's YouTube channel[11]).

Chapter 3

Problem Statement

Edge Computing [7] refers to the technologies allowing computation to be performed on the edge of the network. It brings the computation to the devices where the data is being gathered and therefore does not require communication with a central server. The advantage is that this reduces latency issues that can affect an application's performance. "Edge" can be defined as any computing and network resources along the path between data sources and cloud data centers. Examples for edge devices are smartphones or sensors. Performing Machine Learning on the Edge solves both security concerns and reduces network traffic. It also enables the processing of data in real-time. The edge computing paradigm offers cost savings for companies who do not have the resources to build dedicated data centers for their operations [12]. Engineers can build a reliable network of smaller and cheaper edge devices. Given the features of edge computing and federated learning, edge computing is a suitable environment to perform federated learning in.

Figure 3.1 represents a typical federated learning scenario on the edge. The central cloud server creates a global model, the edge devices download the model parameters. Then, the edge devices perform local training on their data and update the downloaded model. Each device uploads its new model parameters to the cloud server. The server aggregates all the client updates and updates the global model. This procedure is repeated until a stopping condition is met.

Developing a federated reinforcement learning system on the edge presents various challenges. Restrictions and complications from different areas must be taken into consideration. Some of the main challenges are:

- Memory restrictions: edge devices have a very limited amount of memory. The more complex a Machine Learning model is, the more memory is required to train and apply it. As for q-learning, the complexity is determined by the action and state spaces and the resulting q-table. The outcome of the training is stored in the q-table in the form of the optimal policy. It is crucial to keep the q-table's dimensions as small as possible in order for being able to perform reinforcement learning on edge devices.

- Computational restrictions: edge devices have a limited amount of computational resources. Usually, edge devices are not developed for handling large amounts of computation. They usually just take the information detected by a sensor and send it to a server. As with memory restrictions, it is important to keep the q-table's dimensions as small as possible. The bigger the q-table, the more expensive the computation associated with training a model.

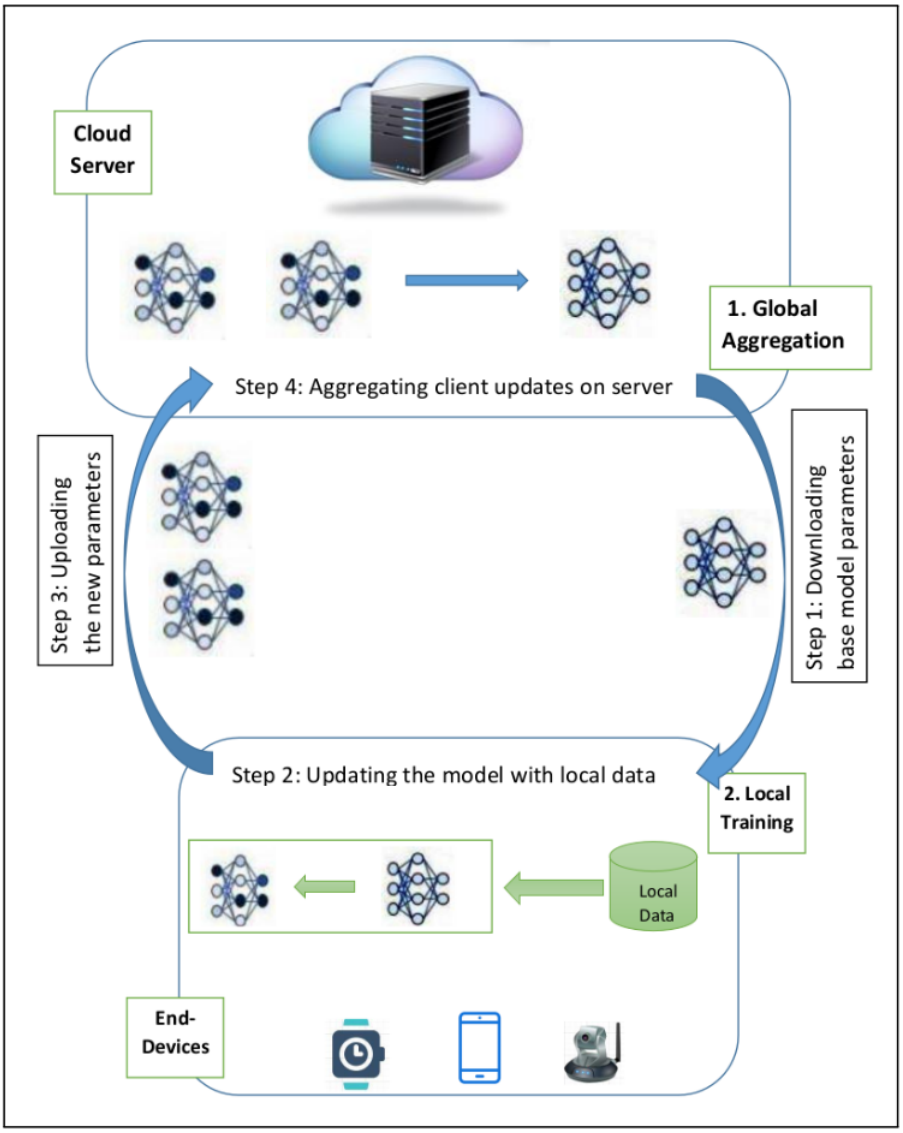

Figure 3.1: An example for a federated learning scenario on the edge (Taken from [3]).

After analyzing these challenges, it is clear that the q-table must be kept as small as possible. However, we still want to achieve decent quality models. That means we need to find a way to make the q-tables as small as possible while still obtaining good models. To achieve this goal, different strategies are implemented and evaluated:

- Centralized approach: this is the basis to start from. The assumption is that there is a central server that knows the information of all the devices. There are no storage or computational restrictions for the server. This is the simplest scenario and should yield the best results.
- Distributed approach: the assumption is that there is no central entity, only edge devices with limited memory and computational power. Each edge device has only information about itself. This approach may lead to disappointing results, because there is no communication between the devices.

- Federated approach: this approach is a compromise between the first two. There is no central server overseeing the whole system, but the edge devices are able to communicate with each other. There will be a trade-off between the amount of communication and the resulting model quality.

Chapter 4

State of the art

For the research for related work we used Google, IEEE Xplore, and Github. The main keywords for the search we used are "federated reinforcement learning", "distributed q-learning", and "reinforcement learning on the edge". The results of the research are listed below.

4.1 Related Work

The problem of our paper is closely related to load balancing and resource allocation. The type of resource can be anything from computational tasks to physical objects such as bikes. The following papers are related to our work:

In [4], the authors propose a spatio-temporal reinforcement learning based bike reposition model. They cluster the bike stations to reduce the problem complexity. Then a deep reinforcement learning model is trained for each cluster. The limitation here is that the bikes can only be repositioned within each cluster. That could be a problem when the customer's behaviour changes in the future, leading to an insufficient number of bikes in one cluster and too many bikes in another cluster. Figure 4.1 shows the proposed framework. It consists of an offline learning process and an online reposition process. The offline learning process has three components: a clustering algorithm, system simulator generation and a Spatio-Temporal Reinforcement Learning model (STRL) for each cluster.

The clustering algorithm first clusters stations which are close to each other and have similar transitions. Then, these clusters are clustered again into groups based on their inter-region transition patterns.

The system simulator is used for training and evaluating the reposition model. It is based on two predictors: an O-Model which predicts the rent and an I-Model which predicts the return demand at each region.

The STRL learns an optimal inner-cluster reposition policy. As the state and action spaces are very large, the authors implement a deep neural network to estimate the optimal long-term value function for each STRL.

The authors of [5] present a reinforcement learning approach for performing nightly offline rebalancing operations in an electric car sharing systems. In this scenario shuttles are used to transport drivers to the electric cars. The drivers then bring the cars to a charging station. A central controller built on recurrent neural network using a policy gradient method determines the routing policies of a fleet of shuttles. This approach works only if the central entity knows the whole environment. All the computation is done by a central

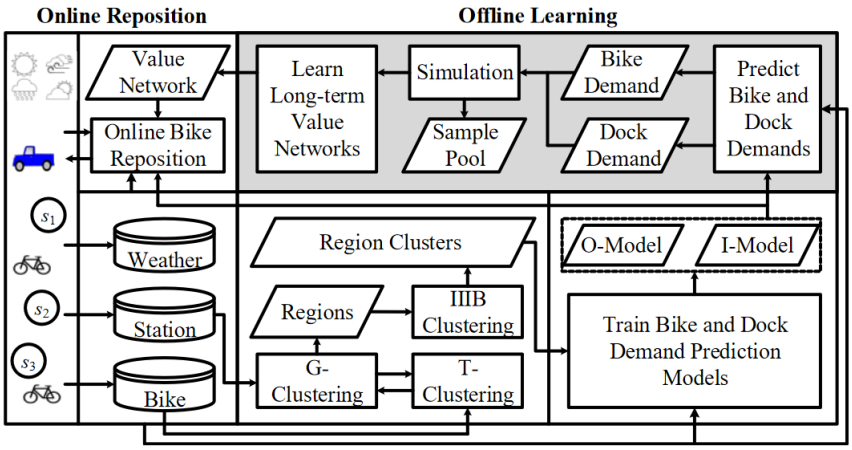

Fig 3. Dynamic bike reposition framework

Figure 4.1: Dynamic Bike Reposition Framework (Taken from [4]).

node. This could potentially lead to scalability problems. Figure 4.2 represents the proposed framework. It consists of three parts: the free-floating electric vehicle sharing system (FFEVSS) simulator, the actor module, and the critic module. The FFEVSS simulator represents the dynamics in the network caused by movements of shuttles. It also updates a masking scheme according to the current state of the network. The actor consists of a convolutional layer and an LSTM for each time step. It produces the solution and the total completion time.

The critic produces a baseline. With the received total reward for the selected actions botht the actor and the critic networks are updated.

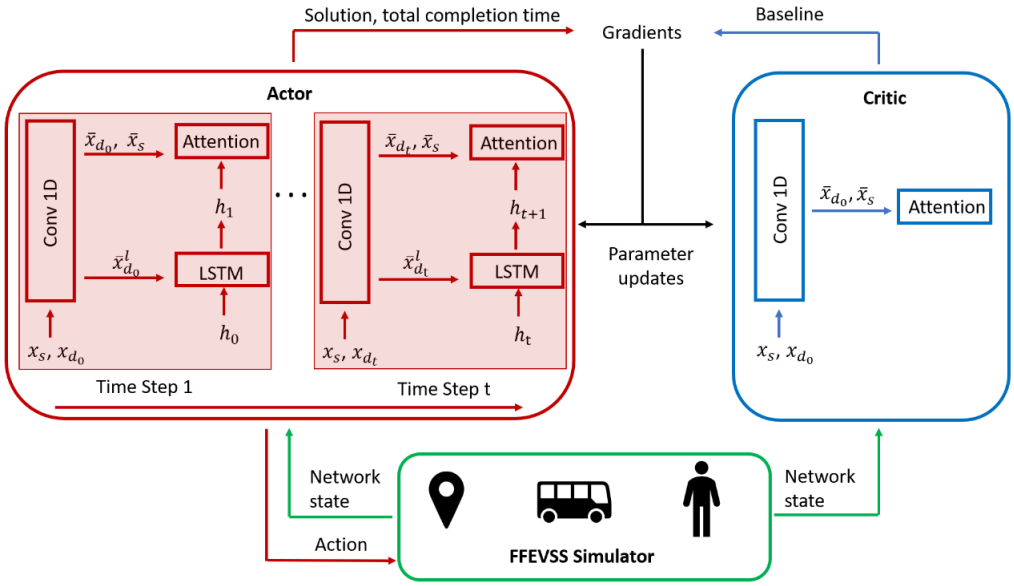

Figure 4.2: An overview of the reinforcement learning model. (Taken from [5]).

In [6], the authors present a deep reinforcement learning framework for rebalancing

dockless bike sharing systems. Dockless bike sharing systems do not have fixed stations. Bikes can be left wherever the customer chooses to. This makes the problem of rebalancing even more complex. Taking into consideration both spatial and temporal data, the authors developed a new algorithm called Hierarchical Reinforcement Pricing which builds upon the Deep Deterministic Policy Gradient algorithm. The model was tested on a Chinese dockless bike sharing system and outperformed state of the art solutions. Figure 4.3 shows an overview of the proposed framework. The rebalancing is not performed by trucks, but users of the bike sharing system are encouraged to place bikes in certain locations by offering them monetary incentives. The service level represents the reward assigned to each action. The service level equals to the total number of satisfied requests. A state consists of supply, demand, and arrival. Supply is the current number of bikes in a region, demand is the number of bikes requested in the region, and arrival is the number of bikes arrived in the current time step in a region.

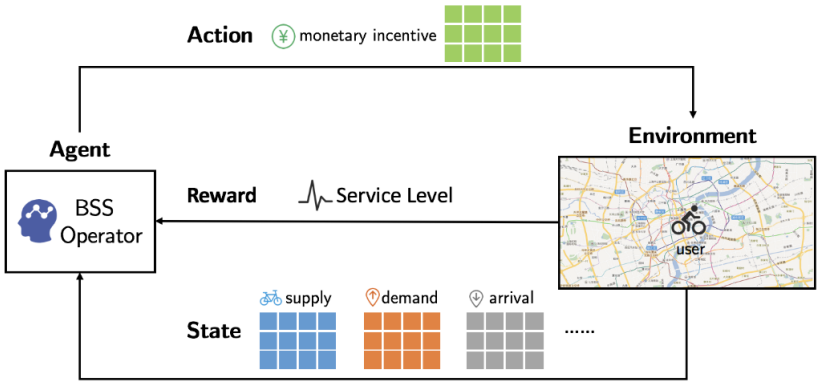

Figure 4.3: An overview of the deep reinforcement learning framework for rebalancing dockless bikesharing systems. (Taken from [6]).

In [13], the authors present a novel federated learning framework based on reinforcement learning which learns a private Q-network policy for each agent by sharing limited encrypted data between the agents. The framework assumes states cannot be shared between agents which complicates the implementation but provides security and privacy advantages. It functions in three phases:

1. each agent collects encrypted (Gaussian differentials) output values of Q-networks from other agents

2. each agent builds a neural network (MLP) to compute a global Q-network output with its own Q-network output and the encrypted values as input

3. each agent updates both the MLP and its own Q-network based on the global Q-network output

These steps are constantly repeated, keeping the model updated. The paper shows that the proposed framework is effective in building high-quality policies for agents given that the data cannot be shared between agents.

[14] focuses on federated reinforcement learning in IoT networks. The authors try to minimize delay and energy consumption by implementing a Double Deep Q-Network (DDQN).

The IoT network consists of three types of devices: IoT devices, MCC (mobile cloud computing server), and MEC (mobile edge computing server). Reinforcement Learning is used to decide whether an IoT device should offload computation to one of the servers or whether to do the computation locally on the device. This decision is affected by communication channel condition, size of the computational task, available transmission power, and local resources. The DDQN consists of an online network for selecting an action and a target network for evaluating the action taken. This approach achieved a scalable, privacy-preserving, and computationally efficient framework for resource allocation and offloading decision optimization.

In [15], the authors solve the problem of resource allocation in mobility-aware federated learning networks by implementing deep reinforcement learning. The network consists of workers, server, power beacon, and different communication channels. The algorithm decides the amount of energy to be recharged to the workers. For this task two channels are available: the default channel and a more stable default channel. The goal of the algorithm is to maximize the number of successful transmissions while minimizing energy and channel costs. The results show that the reward obtained by the proposed algorithm is higher the rewards obtained by conventional algorithms.

In [16], the authors propose a hierarchical federated deep learning framework for reducing content access time in IoT networks and thus optimizing content transfer for more storage resources. The framework categorizes edge devices hierarchically. By minimizing the redundancy of content storage and latency, the proposed framework improves the performance for each local base station network individually and the global network.

4.2 Federated Learning Frameworks

There exist various federated learning frameworks[17]. Some of the most popular are:

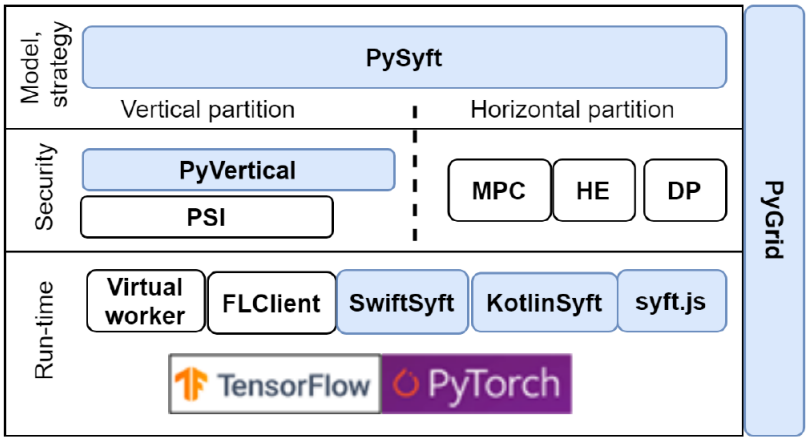

Figure 4.4: Architecture of the OpenMined ecosystem.

- PySyft: an open-source framework for private deep learning written in Python with an MIT license. It is part of the OpenMined[1] ecosystem, which also includes the following projects:

[1] https://www.openmined.org/

- PyGrid: a peer-to-peer network of data owners and data scientists who collectively train analysis models using PySyft
- KotlinSyft: a project to train and inference PySyft models on Android
- SwiftSyft: a project to train and inference PySyft models on iOS
- Syft.js: a web interface for the components listed above

As you can see in Figure 4.4, PySyft supports both vertically and horizontally partitioned data. The available security mechanisms are Multi Party Computation, Homomorphic Encryption and Differential Privacy. PySyft supports both the PyTorch and the TensorFlow libraries. It is recommended to use PySyft in a virtual environment with the Anaconda package manager. PySyft applies the principles of differential privacy. PySyft only works in simulation mode, so the actual deployment on multiple devices is not supported. Like most of OpenMined's projects, PySyft is under active development and by no means a stable or reliable framework. However, this framework has the largest community of contributors (over 250 developers), so it can be expected to have rapid development.

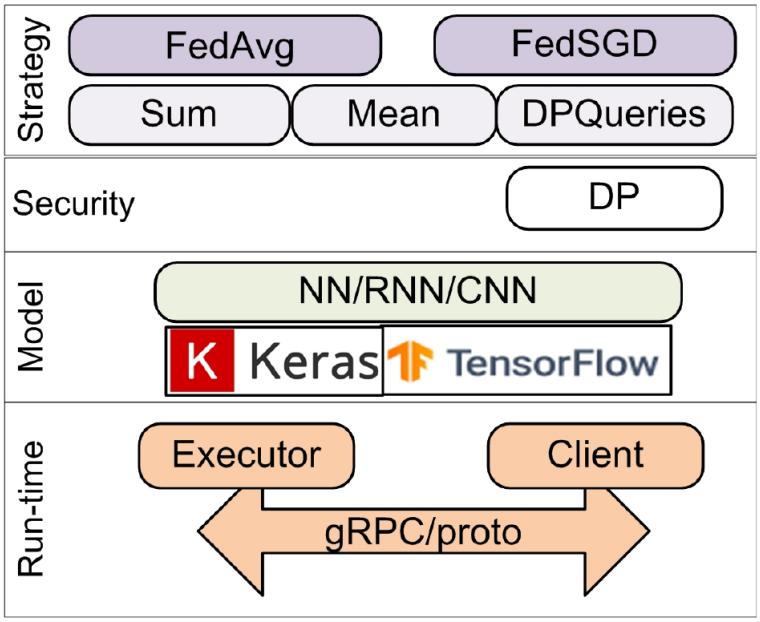

Figure 4.5: Architecture of TensorFlow Federated.

- Tensorflow Federated: an open-source framework for deep learning on decentralized data. It is part of Google's Tensorflow[2] ecosystem. It is distributed under the Apache 2.0 license. It builds on Tensorflow's standard structures. TFF implements classes for the FedAvg and federated stochastic gradient descent algorithms. They can use the following aggregation functions for aggregating the client model updates on the server:
  - sum: sums the client values and outputs the sum located at the server

[2]https://www.tensorflow.org/

- mean: computes a weighted mean of the client values and outputs the mean at the server
- differentially private: aggregates the client values in a privacy-preserving manner based on differential privacy algorithms and outputs the result at the server.

Tensorflow Federated's architecture can be seen in Figure 4.5. Similar to PySyft, it only works in simulation mode. The current version of Tensorflow Federated is not complete and still lacks decent documentation, and features required for practical communication of the framework.

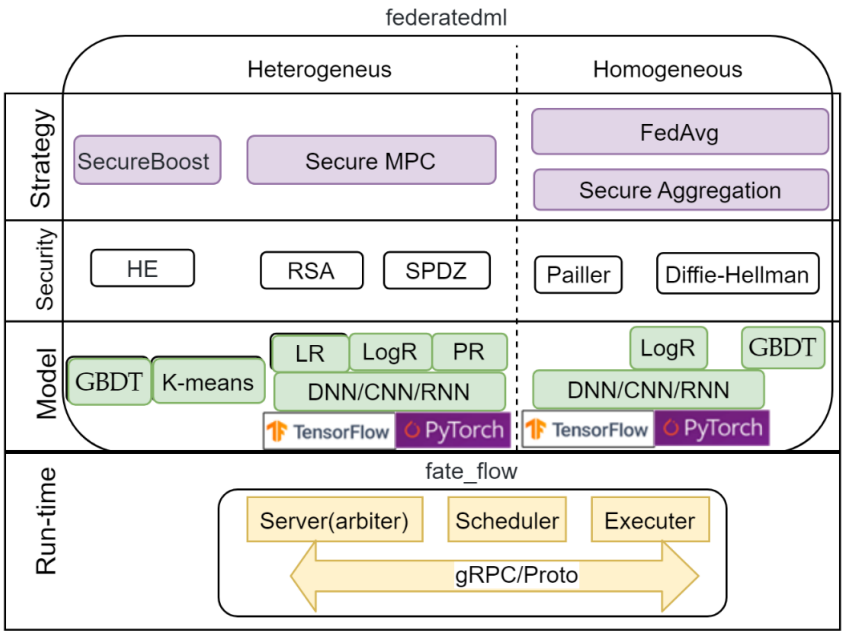

Figure 4.6: Architecture of Federated AI Technology Enabler (FATE).

- FATE: Federated AI Technology Enabler Framework (FATE) is an open-source secure computational federated learning framework by Webank. It is distributed under the Apache 2.0 license. It is based on TensorFlow models. FATE supports both horizontal and vertical data partitions. The available aggregation algorithms are FedAvg and SecAgg. To ensure security, FATE supports various methods. It provides homomorphic encryption, RSA encryption, and MPC protocols. FATE only implements a centralized scheme, which includes:
  - a server performing model aggregation and training
  - a scheduler scheduling the federated learning process
  - an executer that implements the federated learning algorithm

  FATE's architecture can be seen in Figure 4.6. It shows the available models, the encryption algorithms, the possible aggregation algorithms and the run-time explained above. Unlike PySyft and TFF, FATE has a release version and is ready to be used in production. However, it currently only supports a centralized architecture. Also, it does not have a core API. This means developers have to modify the source code in order to implement their own federated learning algorithms.

4.3 Attempt to use Federated Learning Frameworks

We attempted to use both PySyft and Tensorflow Federated, we did not manage to obtain decent results from any of them.

PySyft works with encrypted tensors. We tried to establish a communication between two agents. However, we were not able to obtain the values from the shared tensor. We were able to run training on the tensor, but we did not manage to access the results. It was not clear on which agent the training was performed. We consulted the documentation, but we found it to be a documentation for an old version of PySyft that managed the communication between agents in a completely different way. The communication between agents is managed via an interface called Duet. This interface was only introduced in the more recent versions of PySyft and is completely different compared to previous versions. An agent has to request a tensor from another agent in order to be able to work on it. The other agent must accept the request. Then the tensor is shared while remaining encrypted. We were not able to decrypt the tensor and whenever we performed an algorithm on it, we could not determine on which agent the computation was performed.

We tried to use Tensorflow Federated by implementing a simple recommender system using the movielens dataset. We were able to follow an example notebook and managed to do the training and evaluation. Unfortunately, the resulting mean average error of the recommendations was a lot higher than expected and due to the lack of the documentation we did not find the cause of the error. Tensorflow Federated requires the dataset to be in a particular form, so we converted our dataset into a TFF dataset. The next step was to create a model. TFF works with basic keras models, so this did not represent any difficulty. Then, functions for updating the client and the server must be implemented. The orchestration logic consists of initializing the server and the model, updating the client, updating the server, a function to proceed to the next learning round, and an evaluation function. We followed the provided tutorial, but the results were not as desired.

Based on this experience we decided to implement our own conceptual federated learning framework.

4.4 Review

After analyzing similar papers and different federated learning frameworks we decided to focus on the most popular federated learning frameworks and try to see if they are suitable for our type of problem. We tried to implement simple example notebooks in both PySyft and Tensorflow Federated. Tensorflow yielded unsatisfying results even for simple problems and there was no way of finding why that is the case because there was no documentation available. PySyft works with encrypted tensors and requires the programmer to follow a specific workflow. Again, the only documentation offered was for old versions of the framework and did not apply to the new one in any way. Due to the lack of documentation and presence of bugs, we were not able to obtain satisfying results and therefore decided to develop our own q-learning based federated solution for the problem. These shortcomings are due to the early development stage of both frameworks. Developing our own framework from scratch required more time than using a framework as initially planned.

Chapter 5

Problem Solution

We want to develop a federated reinforcement learning system that works on edge devices and maintains a satisfying performance level. Our solution is split into different steps, ranging from a simple baseline to a more complex approach that comes close to a real federated learning framework. We propose a conceptual model for federated learning on the edge. The code repository can be found at https://gitlab.inf.unibz.it/Hannes.Wiedenhofer/thesis/.

The dataset we used for developing this system is obtained from the Citi Bike dataset[1]. It contains data about a bike sharing system in New York City. The data presents itself in the form of trips taken from one station to another. Based on this data and a starting stock of 20 bikes for each station, the stock history for each station is calculated for a period of 24 hours. Throughout these 24 hours the proposed system tries to maintain the bike stock for each station between the lower limit of 0 bikes and the upper limit of 50 bikes. The solution is based on an implementation from a Github repository[2] created for a project for the course "Machine Learning for Cities" in 2018 at NYU by Ian Xiao, Alex Shannon, Prince Abunku, and Brenton Arnabold. Their implementation focused on a single station and only provided the possibility to remove bikes from the stations' stock. For our solution we took the structure of this implementation and extended it in order to be able to simulate an arbitrary number of agents/stations, so actual real scenarios can be simulated that contain a network of stations. Also, we enabled the stations to move bikes from one station to another, again leading to a system that is closer to real bike sharing environments. Based on this we implemented a system with one central agent overseeing the stock of all stations, a system with distributed agents which do not communicate with each other, and a system that can serve as a base for a federated approach where agents communicate with each other.

5.1 Assumptions

For this project the following assumptions are made:

- At each hour, each agent can move 0, 1, 3, 5, or 10 bikes from one station to another
- The system aims at maintaining a bike stock between 0 and 50 at all times for each station

[1] https://www.citibikenyc.com/system-data

[2] `https://github.com/ianxxiao/reinforcement_learning_project`

- Whenever n bikes are removed from one station, those n bikes must be assigned to another station
- The initial bike stock for each station is set to 20. All stock calculations start from a stock of 20 at 0 am.

The solution is divided into three different approaches. All of them are based on Q-learning. Each approach consists of:

- an agent which manages its q-table, chooses actions based on the q-table, and learns its policy as a consequence of the chosen actions.
- an environment which assigns the rewards to the agent(s), updates the stations' bike stocks, and updates the simulation hour.
- a helper which reads the csv file with the Citi Bike trip data and transforms it into the station history.
- a trainer which calls all the different elements of the system and manages the communication between them. It also stores the results.

While all of the approaches have these elements in common, they differ significantly in number of agents, communication logic and structure of the q-table. In the following these differences are explained in detail.

5.2 Reward System and Metrics

The goal is to keep the stock of every station between 0 (lower limit) and 50 (upper limit) throughout an entire day (24 hours). At each hour the agent(s) perform an action. Based on the chosen action the environment assigns a reward. The reward can be positive or negative. For every bike repositioning the environment assigns a small negative reward. This leads to the agent preferring not to move any bikes if not necessary. Additionally, whenever one station's stock is not within the defined limits, a bigger negative reward is added. At the end of the day, if a station's stock is within the limits, a positive reward is given, otherwise a negative one. The reward system is based on the implementation by the NYU students[3] and extended to function in an environment with multiple stations. Each time a station's stock exceeds either the upper or the lower limit, a reward of -30 is assigned to its agent. At the end of the day, a reward of 20 is assigned to each station if its stock is between the limits, otherwise a reward of -20 is assigned. For each station the following rewards are assigned:

- -30 if bike stock falls outside the range [0, 50] at each hour
- -0.5 times the number of bikes removed at each hour
- +20 if bike stock in [0, 50] at 23 hours; else -20

The evaluation metric represents the percentage of time the stock of all stations is within the defined limits. It can be seen in Figure 5.1. The number of hours where the number of stock exceeds the higher limit and the number of hours where the number of stock exceeds

[3] `https://github.com/ianxxiao/reinforcement_learning_project`

the lower limit is subtracted from the total number of hours (in the case of 2 stations equal to 2*24=48), and then it is converted into percent by multiplying the result by 100 and dividing it by the total number of hours. For each station the number of hours in which the stock is within the limits and the number of hours for which the stock is not within the limits are counted. Then, these numbers are added together for all stations and the percentage is calculated.

$$success\_rate = \frac{(\#total\_h - \#h\_overstock - \#h\_understock) * 100}{\#total\_h} \tag{5.1}$$

where

- $\#total\_h$ represents the total number of hours
- $\#h\_overstock$ is the number of hours the stock is above the upper limit
- $\#h\_understock$ is the number of hours the stock is below the lower limit

5.3 Parameters

The system's behaviour can be altered by changing different parameters. These variables were helpful when developing the system and represent different development steps. Depending on the variables' values, the system can become relatively simple or more complex. These are the most important parameters:

- Remove only
- Stock Mode
- Predictions

5.3.1 Remove only

If this flag is set, the system will no longer be able to move bikes from one station to another, but only to remove bikes from a station. The removed bikes are not assigned to anybody and disappear. While this may not be useful in a real scenario, it is helpful for determining whether the trained model works correctly. When this flag is set, it is not possible to obtain satisfying results when the original stock of a station is below the lower limit at any point in time. The original stock is the number of bikes in a station before the model interacts with it. When this flag is not set, the stations will not be able to remove bikes and make them disappear, instead each removed bike from one station must be assigned to another station.

5.3.2 Stock Mode

The stock mode can be set to "linear", "random", or "actual". It determines the original stock for the stations. When this variable is set to "linear", every station's stock is increased by the same amount of bikes at every hour. The default value is 3 bikes per hour. This results in a constant growth in terms of bike stock. "Random" is similar to "linear", but the number of bikes added at each hour is not constant, but random. These two options may not be helpful in a real environment, but they are useful to verify the performance of the model. The stock mode "actual" uses the data from the csv file in order to create hourly

stock data for each station based on the trips present in the csv file. The resulting stocks are not necessarily always positive. It is possible for stations to have a negative stock mode at certain points in time. This stock mode is the closest to a possible real scenario. In Figure 5.1 we see a graphical representation of the different stock modes.

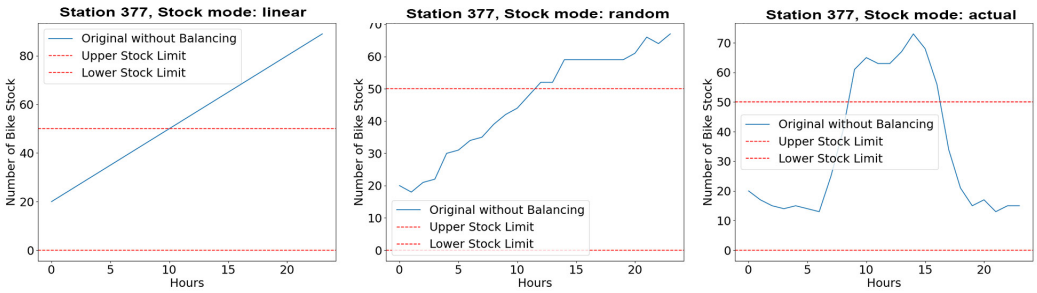

Figure 5.1: Stock modes.

5.3.3 Predictions

When set, this flag enables the system to take into consideration predictions for the stations' bike stock. We took the forecasts from the previously mention github repository[4]. Random Forests models were built to predict the hourly net flow of bikes. For each station, its own model was created. Net flow is the number of bikes arriving minus the number of bikes departing a station. The net flow predictions were added to a station's current balance to predict the expected balance in the future. For training the model trip data from August to October 2016 and July to August 2017 was used. The trip data for September 2017 was used as test set. For each station there were 3696 training records (24 hours times 154 days). The variables for the model were a mix of autoregressive features and weather data scraped from Weather Underground's API. The model's ability to predict the bike stock varied greatly from station to station. Some stations were easily predictable, others difficult. Stations with low bike flow were especially difficult to predict. These predictions enable the agents to take an approximate look into the future and could potentially be of advantage to the model quality.

5.4 Centralized Approach

This is the approach that is based on the simplest structure. There exists only one central agent which oversees all the stations and is aware of all their bike stocks at every point in time. The architecture can be seen in Figure 5.2.

The agent is attached to the central server and this way can access the information of the whole system. This knowledge helps the agent to choose the best action for every situation and therefore to obtain the optimal policy. The q-table contains all information about every station at every point in time. As you can see in the example of a two station environment in Figure 5.3, the state space (first column) contains the stock number for each station, the other columns represent the action space. For example, the action (515,377,1) represents the movement of 1 bike from station 515 to 377. The values are calculated using the Bellman

[4] `https://github.com/ianxxiao/reinforcement_learning_project/blob/master/code/EXPECTED_BALANCES.json`

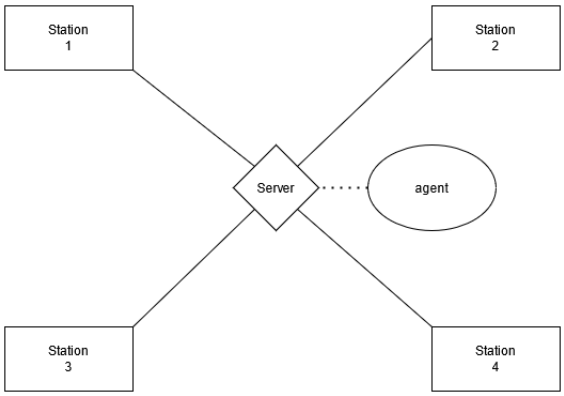

Figure 5.2: Architecture of the centralized approach.

Stock	(515, 377, 1)	(377, 515, 1)	(515, 377, 3)	(377, 515, 3)	(515, 377, 5)	(377, 515, 5)	(515, 377, 10)	(377, 515, 10)	(0, 0, 0)
{"377": 41, "515": -15}	-2,31766E+15	-2,03804E+13	-1,7556E+11	-3,13985E+15	-1,7556E+11	-1,18212E+16	-2,86854E+16	-1,7556E+11	-1,18212E+16
{"377": 53, "515": -27}	-178206	0	-2,36424E+16	-3,51119E+11	-6	-1194	-2,36424E+16	-6	-3,51119E+11
{"377": 74, "515": -27}	-178206	-2940597006	-6	-1194	-1194	0	-178206	0	-2,36424E+16
{"377": 77, "515": -26}	-6	0	0	-6	-6	-6,2797E+15	-178206	-1194	-1194
{"377": 70, "515": -18}	-2,36424E+16	-178206	-2940597006	-1194	0	-1194	-2940597006	-178206	-178206
{"377": 71, "515": -22}	-178206	-1194	0	-6	-178206	-2,36424E+16	-1194	-1194	-1194
{"377": 75, "515": -18}	-178206	-2940597006	-1194	-1194	0	-6	0	-1194	-1194
{"377": 80, "515": -14}	-1194	-178206	-3,51119E+11	-1194	-3,51119E+11	-1194	-2940597006	-2940597006	-178206
{"377": 65, "515": 1}	-1,18212E+16	-1,7556E+11	-1470298503	-1,18212E+16	-597	-597	-2,03804E+13	-1,7556E+11	-1,18212E+16
{"377": 63, "515": -13}	-4,07608E+13	0	-6	-1194	-178206	-178206	-6	0	-1194
{"377": 29, "515": -1	-3,1398[illegible]+15	1,182[illegible]E+16	-2,8685[illegible]E+16	-3,408[illegible]5E+15	-1470[illegible]98503	-2,3176[illegible]6E+15	-2,31[illegible]6E+15	-2,[illegible]804E+1	-1,7[illegible]56E+11
[illegible]	[illegible]03	-3	[illegible]	[illegible]0	[illegible]3103	[illegible]	[illegible]-59	[illegible]	[illegible]29103

Figure 5.3: Example Q-table for centralized approach.

equation. Let us see how the calculation works based on an example. We assume this is the first episode of the algorithm. Therefore, the old value $Q(s_t, a_t)$ will be 0. The learning rate is set to 0.01 because we want the algorithm to take into consideration the knowledge learnt in the past. Let us assume this episode was not successful at keeping the bike stock between the limits and therefore received a reward of -270. The discount factor is set to 0.9 because we do not want the algorithm to focus too much on the current reward, but we want it to achieve high rewards in the long run. Let us assume the algorithm estimated the optimal future value to be 180. With these values the Bellman equation looks like this:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{0}_{\text{old value}} + \underbrace{0.01}_{\text{learning rate}} \cdot \overbrace{(\underbrace{-270}_{\text{reward}} + \underbrace{0.9}_{\text{discount factor}} \cdot \underbrace{180}_{\text{estimate of optimal future value}} - \underbrace{0}_{\text{old value}})}^{\text{temporal difference}}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

This would lead to a $Q^{new}(s_t, a_t)$ of 159,3. This is the value that is then inserted into the q-table.

This approach works well for a limited amount of bike stations and a limited amount of total bikes in the system. However, when a larger number of bike stations or a larger number of bikes are part of the simulation this approach should be avoided due to the large action space and state space which is generated by the single agent. These large action and state spaces lead to very long training times and require a large amount of training data. So, this approach seems to work well for a small amount of stations and bikes, but it does not scale well. Some possible solutions for this problem are the use of Deep Q-Learning or the

use of Function Approximation. The next approach tries to solve the scaling problem while maintaining the q-learning algorithm.

5.5 Distributed Approach

This approach aims at solving the scaling problem encountered in the previous approach. However, this is achieved at the expense of the resulting policy's quality. In this case each station has its own agent. There is no central entity that oversees all the stations and their stocks. This leads to smaller action spaces and smaller state spaces. The architecture can be seen in Figure 5.4. Each station has its own agent, but the agents do not communicate with each other. Each agent manages its own q-table. In Figure 5.5, we see an example for one agent's q-table (station 377). In the distributed approach in a system with n stations, there will be n different q-tables, 1 for each station. Compared to the centralized approach, here the state space consists only of one station's bike stock. The action (1,515) represents the movement of 1 bike from station 377 to station 515. The agents do not communicate with each other. They have no information about the other stations' bike stocks, only about their own. This leads to each station making decisions that seem to be the best for them without considering the consequences each decision has on the other stations. Each agent only has the possibility to actively remove a certain number of bikes from its stock. The addition of bikes to a station's stock occurs when another station decides to move a certain number of bikes from its stock to the station. This means, when a station's stock is close to exceed the upper limit, the station can actively remove bikes. However, when a station's stock is close to falling below the lower limit, the station can only decide not to remove any bikes and wait for another station to assign some bikes to it. As expected, this approach leads to a significant drop in policy quality compared to the previous approach. So, while this approach is very scalable, it does not perform very well. The next approach tries to find a balance between the centralized and distributed approach by allowing agents to communicate with each other.

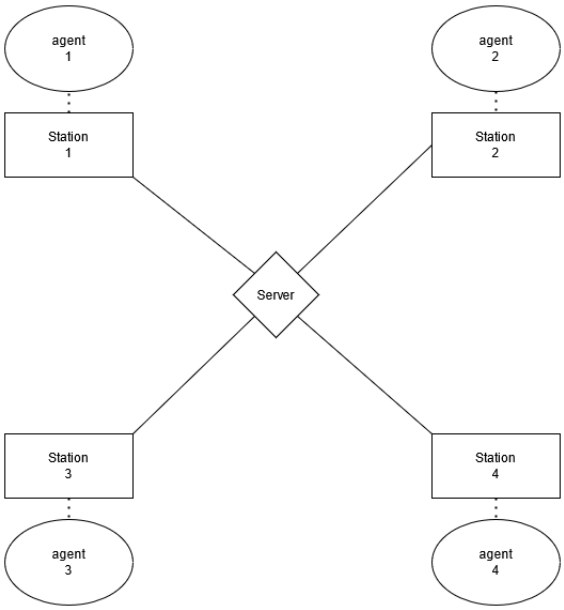

Figure 5.4: Architecture of the distributed approach.

Stock	(1, 515)	(3, 515)	(5, 515)	(10, 515)	(0, 0)
20	-1,3015E+16	-3,44935E+15	-8,27571E+15	-1,3751E+16	0
27	-4,32414E+15	-1,01902E+14	-2,61654E+15	-3,86277E+16	0
29	-6,12395E+16	-1,15883E+16	-742525	-6,99708E+15	0
28	-3,86277E+16	-1,29724E+16	-1,463E+12	-6,99708E+15	0
33	-5,23309E+16	-1,70423E+15	-4,55233E+16	-4,7809E+15	0
23	-5,23309E+16	-8,77798E+11	-12252487525	-2,92599E+11	0
22	-3,86277E+16	-7351492515	-1,93138E+16	-5,23309E+15	0
41	-7,42711E+15	-2,48229E+15	-1,93138E+16	-5,23309E+15	0
21	-1,05269E+16	-1,10942E+16	-1,05269E+16	-9,72813E+13	-2,67125E+15
	-7 84F	-7 99E	-7 09E	34F	-1 17F

Figure 5.5: Example Q-table for distributed approach (Station 377).

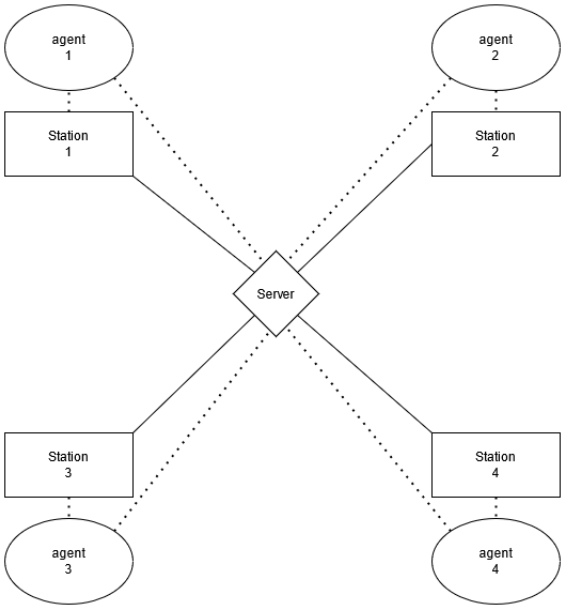

Figure 5.6: Architecture of the federated approach.

5.6 Towards a Federated Approach

While the centralized approach does not scale well and the distributed approach does not yield satisfying results, this approach represents a compromise between the two. Each station has its own agent, but the agents share information. The goal is to obtain a high quality model while sharing as few data as possible. The architecture can be seen in Figure 5.6. The communication between the agents and the central server is represented by the dotted lines. In the simplest implementation of this approach each agent knows the bike stock of every other agent at all times. This can be seen in Figure 5.7, which represents an example q-table for station 377. Here, the state space contains stock information of all the stations (in this case two). We implemented two different versions of this federated approach. The first one takes into consideration the actual stock values and uses them in the q-table. In the second approach the agent does not have access to the actual stock data of the other stations.

Stock	(1, 515)	(3, 515)	(5, 515)	(10, 515)	(0, 0)
{"377": 20, "515": 20}	-8,27431E+14	-2,85408E+15	-5,35805E+15	-1,2264E+16	0
{"377": 19, "515": 18}	-5	-15	-25	-19701995	0
{"377": 22, "515": 13}	0	0	0	0	0
{"377": 19, "515": 20}	-5	0	0	0	0
{"377": 21, "515": 13}	0	0	0	0	0
{"377": 22, "515": 20}	-5	-15	0	0	0
{"377": 27, "515": 7}	0	0	0	0	0
{"377": 21, "515": 19}	-2450497505	-1,70423E+15	-2,61654E+15	-5,23309E+15	0
{"377": 21, "515": 20}	0	0	-25	0	0
{"377": 7"			0		

Figure 5.7: Example Q-table for approach towards federation (Station 377).

It only has access to the predictions of the stock data. As expected, the second approach yields worse results in terms of success rate. The state space has the same structure as in the centralized approach, while the action space has the same structure as the distributed approach. This yields a very high quality model, but does not succeed in keeping the amount of shared data low. The other side of the spectrum is represented by the previous approach (distributed). For this approach it will be crucial to find the right balance, and define what data should be shared between the agents.

5.7 Clustering

Among the many stations, some are more closely related than others if we define a relation between two stations to be the number of trips taken by the customers between these two stations. For instance, the average trip duration is about 15 minutes. Therefore, station A may be more closely related to station B which is apart 15 minutes from it than to station C which is apart 3 hours from it. A possible hypothesis would be that the relation between two stations could have an influence on the model's performance. Therefore, an Origin-Destination Matrix was created, represented in Figure 5.8. It only shows part of the matrix. We can see, for example, that there were 4 trips between station 168 and station 236. The numbers are relatively small because this only regards the traffic of one day. Clustering was applied to the Origin-Destination Matrix to determine which stations are closely related and which stations are not. This was achieved by applying agglomerative clustering on the dataset, using Ward's method. 5 different clusters were created.

start station id	120	127	150	164	168	173	174	217	228	236
120	0	0	0	0	0	0	0	0	0	0
127	0	9	0	0	0	1	0	0	0	1
150	0	0	5	1	1	0	0	0	0	1
164	0	1	0	1	0	1	0	0	1	0
168	0	0	1	0	3	0	0	0	0	4
173	0	0	0	2	2	2	0	0	2	1
174	0	0	0	0	0	0	0	0	1	2
217	0	0	0	0	0	0	0	10	0	0
228	0	1	0	0	0	0	0	0	1	0

Figure 5.8: Origin-Destination Matrix.

After performing various experiments, the relationship between the stations in terms of number of trips did not influence the model's performance as the station stock is calculated before starting to train the model and therefore does not interfere in any way with the training. Figure 5.9 shows the comparison between the results of two different clusters. The diagram on the left is the result of a simulation for two stations from cluster 0. Cluster 0 represents the least well-connected stations. The right diagram shows the result of a simulation for two stations from cluster 4. Cluster 4 represents the most well-connected stations. Both results have a success rate of aroun 98%. This example shows that there is no significant difference between the different clusters in terms of model quality, otherwise one of the simulations would have shown clearly worse results.

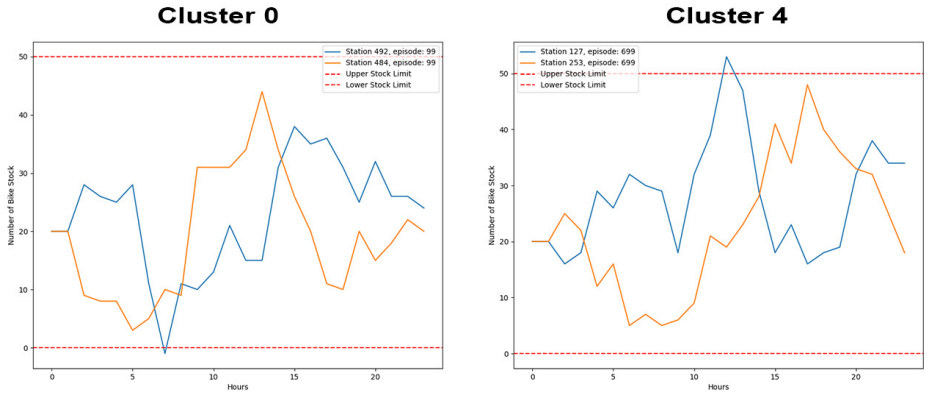

Figure 5.9: Comparison of different clusters.

5.8 Station selection

Having set the limits to 0 and 50, it is not possible to stay between these limits for all combinations of stations. The sum of the stations' stocks must always be between 0 and 50 times the number of stations. In Figure 5.10, we see two combinations of stations and the sum of their stock. It is possible to stay within the fixed limits only for the first combination of stations (358 and 459). The sum of their stock is between 0 and 100 for all 24 hours. This is not the case for the second combination of stations, as the limit of 100 is exceeded several times. Therefore, we will only consider valid combinations of stations in the results.

358	20	22	23	24	24	23	19	5	-10	-7	-1	-9	-9	0	3	15	13	25	22	24	20	21	24	22
459	20	20	20	20	22	29	40	43	48	57	57	55	56	48	45	54	56	52	52	52	43	39	34	34
SUM	40	42	43	44	46	52	59	48	38	50	56	46	47	48	48	69	69	77	74	76	63	60	58	56
426	20	20	21	22	22	22	30	45	86	98	111	115	114	135	147	147	146	135	131	132	134	127	135	135
453	20	20	20	20	19	18	12	4	-5	-11	-16	-14	-14	-10	-7	-1	2	8	17	16	19	20	20	20
SUM	40	40	41	42	41	40	42	49	81	87	95	101	100	125	140	146	148	143	148	148	153	147	155	155

Figure 5.10: Sum of bike stocks of two different combinations of stations.

Chapter 6

Evaluation

The proposed solution is evaluated both in absolute values, in terms of success rate and in relative values, comparing the different approaches to each other.

6.1 Methodology

In the following we describe the methodology we used for evaluating the model quality.

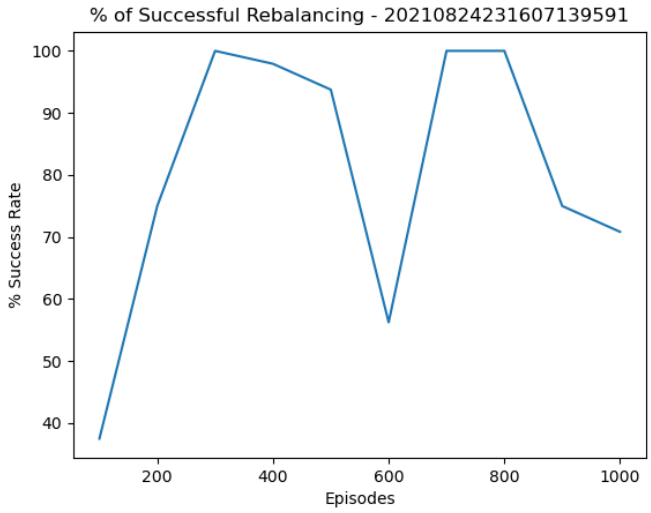

Figure 6.1: Example of success rate by number of episodes.

6.1.1 Number of episodes vs success rate

Each execution consists of multiple sessions with different numbers of training episodes. After each session, all values are initialized again and the environment is reset. The number of sessions and episodes per session are parameters that can be changed. For our experiments we used 10 sessions, starting from 100 episodes for the first session, increasing each session by 100 episodes, ending with 1000 episodes for the last session. Figure 6.1 represents the result of an execution in terms of success rate. Each point in this diagram represents one

session, ranging from the session with 100 episodes to the session with 1000 episodes. As we can see, the number of training episodes is not necessarily directly proportional to the success rate. This finding is crucial for evaluating the model quality. For evaluating the model quality, we need to take the results for the session with the highest success rate, not the results for the session with the most training episodes.

To demonstrate the importance of choosing the right session for evaluating the model quality, we compare the bike stocks of session 2 and session 9 for the two stations from Figure 6.1. In Figure 6.2 we see the bike stock for the two stations from the session with 300 episodes. Both stations' bike stock remains within the limits throughout all 24 hours. This yields the result of 100% success rate from Figure 6.1. This can be defined as an optimal outcome.

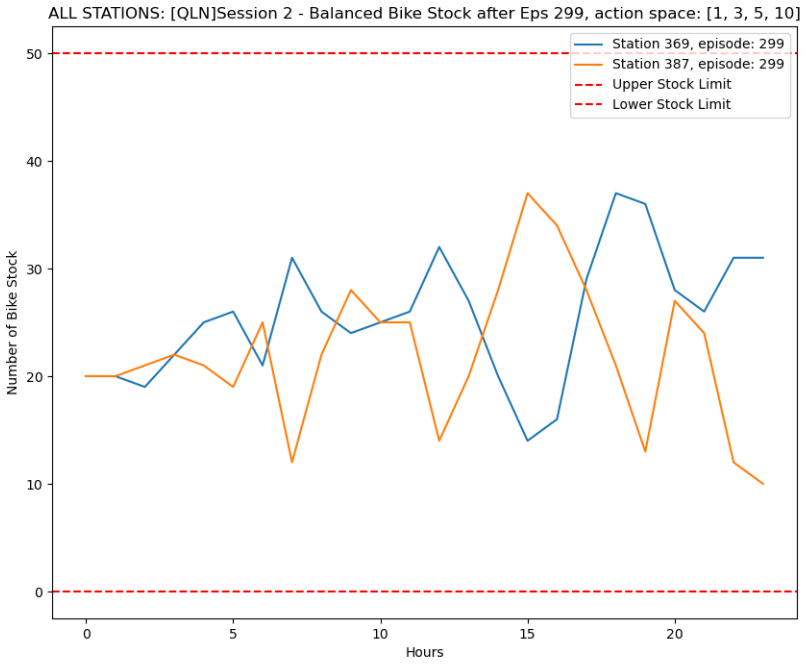

Figure 6.2: Bike stocks for session 2 (300 episodes).

In Figure 6.3 we see the same graph, but for session 9 with 1000 episodes. The bike stock of the two stations does not remain within the limits. In fact, at around hour 10 the bike stocks drift apart and one exceeds the upper limit, the other stock exceeds the lower limit. As we see in Figure 6.1, the success rate for this session is only about 70% even though the number of episodes is 1000.

These findings suggest not to use the result for the session with the highest number of episodes, as it does not always yield the best success rate. Therefore, it is crucial to select the session with the highest success rate, no matter the number of training episodes.

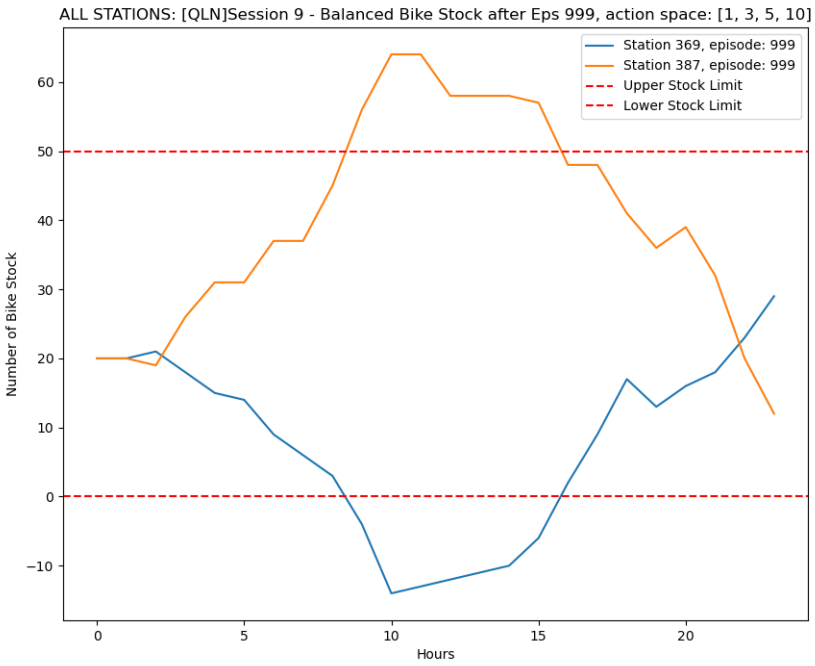

Figure 6.3: Bike stocks for session 9 (1000 episodes).

6.1.2 Analyzing the learning within a session

For analyzing and evaluating a single session, three crucial aspects must be analyzed: the original bike stock, the balanced bike stock from episode 0, and the balanced bike stock from the last episode. By looking at the relation between these stocks, we can determine if the system worked correctly and how it performed. Figure 6.4 helps us in analyzing a session. The original bike stock without balancing exceeds the upper limit. The bike stock of episode 0 overcorrects this behavior. The result is a stock that exceeds the lower limit in two instances. Episode 799 is able to improve the bike stock and does not exceed any limits, leading to a 100% success rate for this station. If episode 799 would have yielded a worse result in terms of success rate than episode 0, that would mean the system is not learning correctly. Such a case would require revisiting the reward system as the rewards may not be assigned correctly depending on the action taken.

6.1.3 Action History

Each station has its own action history for each training episode. The comparison between the action history of episode 0 and episode 799 of a station can be seen in Figure 6.5. This diagram shows us that the agent in episode 0 decided to remove 10 bikes multiple times early in the morning. The action history of episode 799 shows that the agent learned that it is not necessary to remove that many bikes in the morning because they are probably not needed anywhere else. It decided that bikes should be removed later throughout the day. Analyzing these diagrams allows us to understand the behavior of each station and the learning it went through.

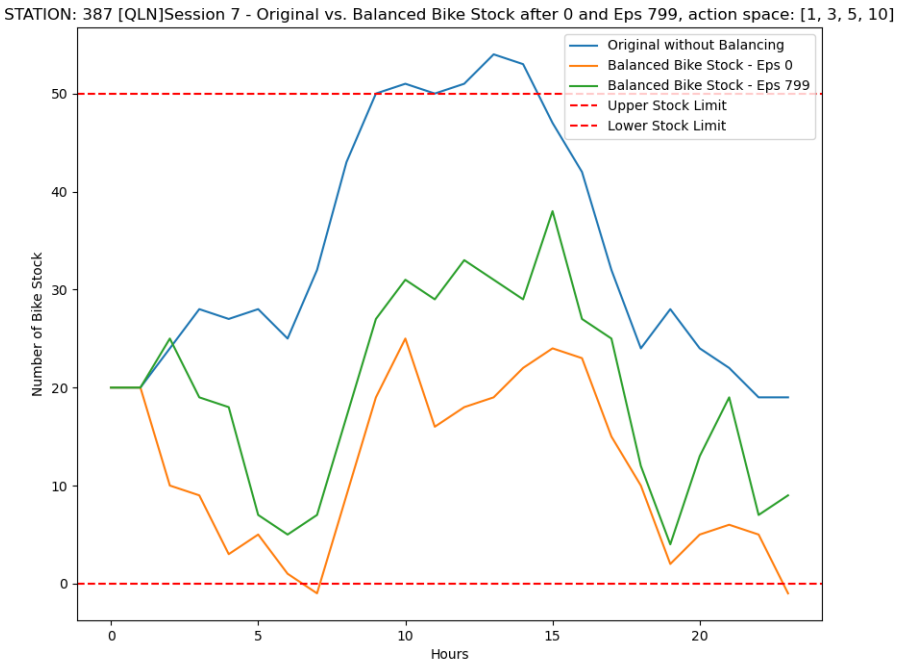

Figure 6.4: Stock analysis for one session.

6.2 Modelling rounds

In order to reach our goal of a conceptual federated reinforcement learning framework for edge devices we went through multiple modelling rounds, starting from a simple baseline and extending the framework at each round. In the following we describe each modelling round in detail.

6.2.1 Replicating the results of the base code

The first step was to analyze the base code from github[1] and try to replicate the results of the authors. The code from github only focused on one station and only provided the possibility to remove bicycles from the system completely. We wanted to extend the code in order to achieve a conceptual federated learning framework. However, first we had to understand the code and verify the results it produces. The github repository provided a diagram showing the success rate for a certain station. We performed the learning on the same station and achieved very similar results as you can see in Figure 6.6. We were able to obtain a slightly better result while only performing 100 learning episodes compared to their 1000 learning episodes. However, these results slightly differ at each execution, therefore we can conclude that the results are very similar and the code is working correctly. This guarantees us a solid base to extend and build our framework on.

[1] `https://github.com/ianxxiao/reinforcement_learning_project`

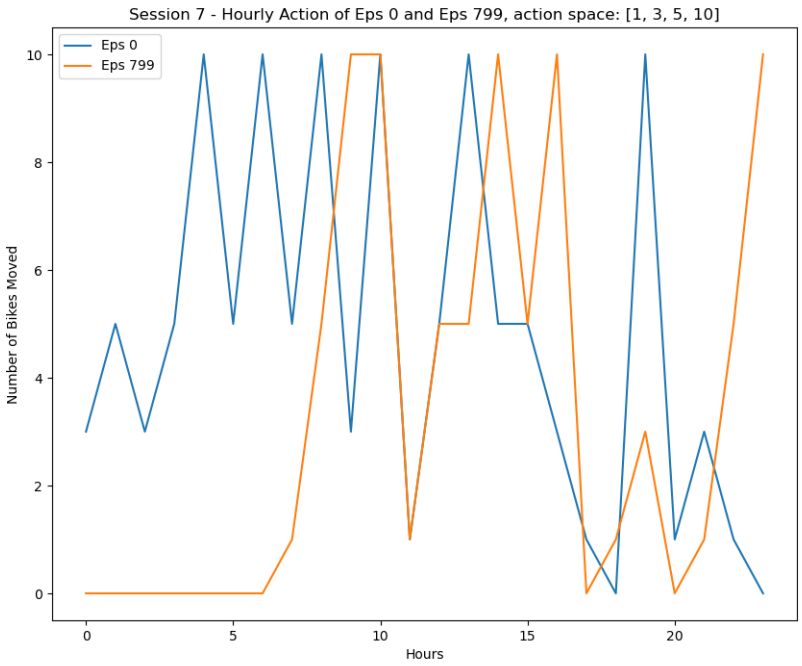

Figure 6.5: Comparison between action history of episode 0 and episode 799.

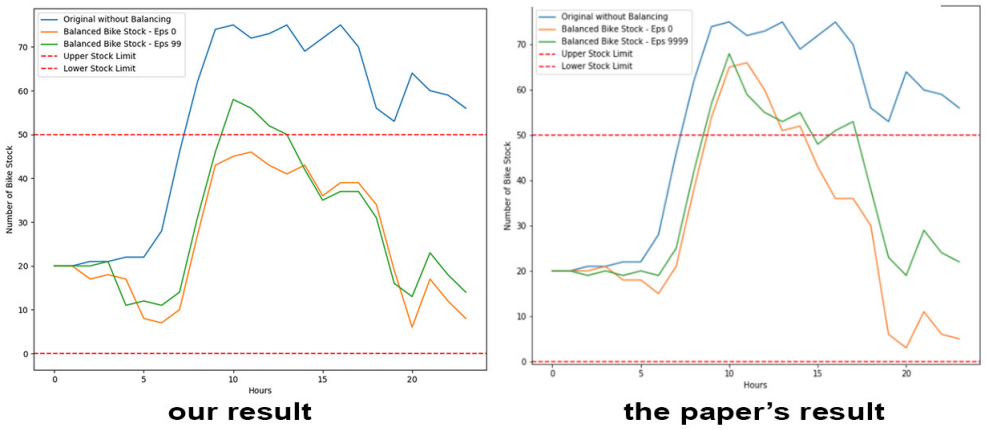

Figure 6.6: Comparison of our result and the provided result from github.

6.2.2 Implementing our own baseline

After verifying the correct functioning of the base code, we implemented our own baseline which works with multiple stations and is able to move bikes between them. We could maintain the structure of the code, i.e the classes, but all other aspects of the code had to be changed to function with multiple stations. This required vectorizing many parts of the code, remodelling the communication between different parts of the code, and adapting input and output mechanisms. The most important change we introduced is the remodelling

Old Q-table

Action (# of bikes to remove)

State (# of bikes in stock)

	-10	-1	0
20	0	0	0
18	0	0	0
17	0	0	0

New Q-table

Action (# of bikes to move from one station to another)

State (# of bikes in stock for each station)

	(s1, s2, 10)	(s1, s2, 1)	(s2, s1, 10)	(s2, s1, 1)	(0,0,0)
#s1,#s2	0	0	0	0	0
#s1,#s2	0	0	0	0	0
#s1,#s2	0	0	0	0	0

Figure 6.7: Comparison of the old q-table and the new q-table.

of the q-table as can be seen in Figure 6.7. The old q-table only contained stock data for a single station. We wanted to have a system with multiple stations and therefore changed the first column of the q-table to contain information about all the stations. In this case we assume the system consists of two stations. Additionally, the structure of the actions had to be changed. For this example we assume there exist only three possible actions: remove 10 bikes, remove 1 bikes, do not remove any bikes. In the old q-table these actions were applied to only one station. Again, we wanted to change that and introduced tuples for each action. For example, the tuple (s1,s2,10) represents the movement of 10 bikes from station 1 to station 2. Thanks to this change we are now able to model movements between stations. However, when increasing the number of stations or actions in the system the q-table's dimensions increase exponentially. Therefore, we decided to further extend our code to be able to work with different structures and compare the approaches.

6.2.3 Extending baseline

While our centralized baseline yielded models with high success rates, it did not scale well. Therefore, we decided to extend our code in a way that allows us to represent different approaches with different structures. The first structure we decided to implement, is the distributed approach. Each station has its own agent, there is no central entity that oversees the whole system. Again, the communication between various parts of the code had to be changed. Additionally, the structure of the q-table had to be changed. The whole system consists of more agents and therefore more q-tables. The advantage of the distributed approach is that there is no longer one large q-table, but multiple smaller q-tables which makes the system more scalable. However, the resulting model quality was clearly lower than the centralized baseline. Therefore, we introduced another approach which can serve as the basis for a federated reinforcement learning system on the edge in a successive step. This approach is a compromise between the centralized baseline and the distributed approach. It maintains multiple q-tables whose size depends on the amount of communica-

tion between the agents. The bigger the q-tables, the higher the resulting success rate, but also the worse the scalability. Therefore, when extending this approach in the future, it will be crucial to make a trade-off analysis and adjust the communication volume according to the requirements of the case.

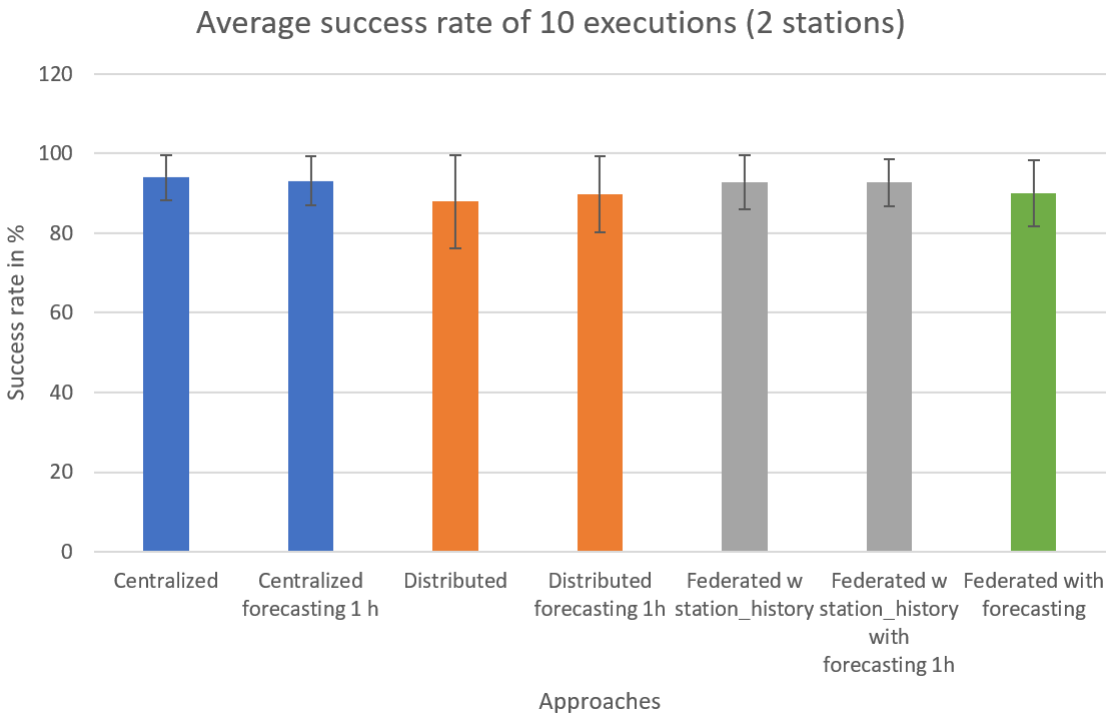

Figure 6.8: Comparison of the different approaches using 2 stations.

6.3 Comparison of Results

For comparing the model quality of the different approaches, we randomly chose 10 different combinations of 2 stations and trained the models for each of them. Then we calculated the mean success rate. As we can see in Figure 6.8, as expected the centralized approach yielded the highest success rate, followed by the federated approach. The distributed approach has the lowest average success rate. Additionally, forecasting only seems to improve the success rate for the distributed approach. For all other approaches, the success rate is slightly lower than the one without forecasting. As expected, for the federated approach, using the actual station history yields better results than using the values from the forecasting. The detailed success rates for each execution can be seen in Figure 6.9. Values below 75% are highlighted in red, values between 75% and 85% are highlighted in yellow, and values higher than 85% are highlighted in green. The mean and the standard deviation are calculated.

The ranking of the different approaches can be seen in table 6.1. The success rate ranges from 93,96% to 87,92%.

We repeated this comparison for systems with 3 stations. However, due to time restrictions, instead of 10 executions we performed 5 executions and did not consider forecasting. The mean success rate was lower compared to the experiments with 2 stations. This was ex-

	Success Rate (%)						
Stations	Centralized	Centralized forecasting 1 h	Distributed	Distributed forecasting 1h	Federated w station_history	Federated w station_history with forecasting 1h	Federated with forecasting
[377,515]	95,83	87,5	64,58	68,75	79,17	81,25	87,5
[435,517]	83,33	87,5	89,58	93,75	89,58	95,83	100
[127,253]	100	95,83	91,67	97,92	100	87,5	100
[3472,523]	95,83	100	97,92	93,75	97,92	100	83,33
[3163,368]	93,75	85,42	93,75	93,75	93,75	91,67	81,25
[369,387]	100	100	100	100	100	100	100
[358,459]	95,83	100	79,17	93,75	95,83	97,92	97,92
[484,492]	87,5	93,75	70,83	75	83,33	87,5	81,25
[346,459]	100	97,92	100	89,58	97,92	95,83	79,17
[3119,456]	87,5	83,33	91,67	91,67	89,58	89,58	89,58
Mean	93,957	93,125	87,917	89,792	92,708	92,708	90
STD	5,6252965	6,253693709	11,705163	9,476663759	6,79928055	5,911001269	8,270765382

Figure 6.9: Success rates for each execution (2 stations).

Ranking	Success Rate	Approach
1.	93,96%	Centralized
2.	93,13%	Centralized forecasting 1h
3.	92,71%	Federated with station history
4.	92,71%	Federated with station history forecasting 1h
5.	90%	Federated with forecasting
6.	89,79%	Distributed forecasting 1h
7.	87,92%	Distributed

Table 6.1: Ranking of the success rate for different approaches (using 2 stations).

pected as with increasing numbers of stations the q-tables' dimensions increase exponentially and therefore require a larger number of training episodes. Due to time restrictions we did not increase the training episodes compared to the experiments with systems with 2 stations. However, as we can see in Figure 6.10 the relation between the different approaches remained similar. The centralized approach still yields the highest success rate, followed by the federated approach. The distributed approach produced the lowest success rates. Figure 6.11 represents the detailed values for each execution. Again, values below 75% are highlighted in red, values between 75% and 85% are highlighted in yellow, and values higher than 85% are highlighted in green. The mean and the standard deviation are calculated.

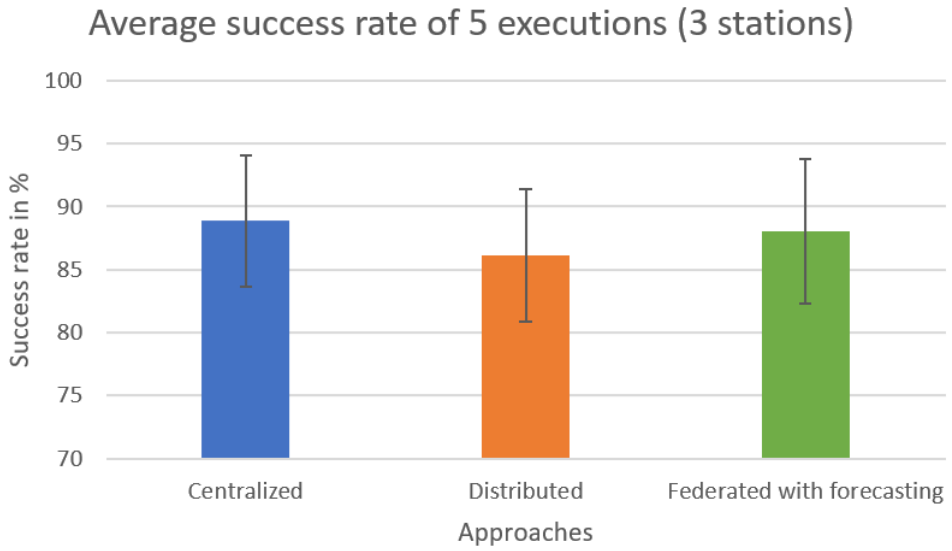

Figure 6.10: Comparison of the different approaches using 3 stations.

	Success Rate (%)		
Stations	Centralized	Distributed	Federated with forecasting
[351,428,523]	83,33	87,5	93,06
[351,406,3161]	94,44	81,94	91,67
[496,509,2006]	91,67	87,5	93,06
[257,379,3463]	93,06	94,44	83,33
[324,509,3354]	81,94	79,17	79,17
Mean	88,888	86,11	88,058
STD	5,19876678	5,268882234	5,734755095

Figure 6.11: Success rates for each execution (3 stations).

Chapter 7

Discussion

We reviewed the state of the art of federated learning, in particular federated reinforcement learning on the edge. We analyzed, compared and tried to use the most common federated learning frameworks. We came to the conclusion that they are in an early phase of development and therefore contain bugs and do not offer a decent documentation. Therefore, we decided to build our own federated reinforcement learning system that solves a rebalancing problem. This system enabled us to compare structurally different approaches and highlight their strengths and weaknesses. The data we used to train the model is taken from a bike sharing system in New York City.

The system consists of three different approaches, namely a centralized approach with one agent overseeing the whole system, a distributed approach with one agent per station and no communication between the agents, and a basis for a federated approach with one agent per station and each agent having access to data of other agents. In addition to the different structural approaches, it is possible to modify the system's behavior based on different variables, such as remove only, stock mode, and prediction. Depending on these variables' values, the complexity of the system can be either increased or decreased. We also investigated whether the system finds it easier to simulate an environment with stations which are connected with each other by a high number of trips between them. We found that the relationship between the stations does not influence the model quality.

The proposed system can be adapted to work with other datasets in a different environment for different rebalancing problems. Also, it represents a basis for implementing a fully federated reinforcement learning system.

Chapter 8

Conclusion and Further Studies

We analyzed the state of the art of federated reinforcement learning on the edge. We analyzed and compared different federated learning frameworks, coming to the conclusion that they are in an early phase of development and are not reliable or stable. We proposed our own framework solving a rebalancing problem in a bike sharing system. This framework can be adapted to function with different datasets. Also, it can serve as a base for building a fully federated learning framework. In order to do that it suffices to establish a way of communication between the agents, the structural requirements are already fulfilled by the proposed solution. The communication volume should be as small as possible, yielding smaller q-tables which lead to less computational effort required to perform the learning. It would be necessary to determine the communication volume and analyze the trade-off between communication volume and model quality. Additionally, the system could be modified to work with a deep neural network, making it a deep reinforcement learning model. This could be especially useful when dealing with more complex datasets with large action spaces and state spaces.

Bibliography

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[2] Andrew Hard, Chloé M Kiddon, Daniel Ramage, Francoise Beaufays, Hubert Eichner, Kanishka Rao, Rajiv Mathews, and Sean Augenstein. Federated learning for mobile keyboard prediction, 2018.

[3] M. G. Sarwar Murshed, Christopher Murphy, Daqing Hou, Nazar Khan, Ganesh Ananthanarayanan, and Faraz Hussain. Machine learning at the network edge: A survey, 2021.

[4] Yexin Li, Yu Zheng, and Qiang Yang. Dynamic bike reposition: A spatio-temporal reinforcement learning approach. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, pages 1724–1733, 2018.

[5] Aigerim Bogyrbayeva, Sungwook Jang, Ankit Shah, Young Jae Jang, and Changhyun Kwon. A reinforcement learning approach for rebalancing electric vehicle sharing systems, 2021.

[6] Ling Pan, Qingpeng Cai, Zhixuan Fang, Pingzhong Tang, and Longbo Huang. A deep reinforcement learning framework for rebalancing dockless bike sharing systems, 2018.

[7] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[8] H. Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Agüera y Arcas. Communication-efficient learning of deep networks from decentralized data, 2017.

[9] Donald Michie. Experiments on the mechanization of game-learning part i. characterization of the model and its parameters. *The Computer Journal*, 6(3):232–236, 11 1963.

[10] Christopher Watkins. Learning from delayed rewards. 01 1989.

[11] Tensorflow. Tensorflow YouTube channel. `https://www.youtube.com/c/TensorFlow`. Online; last accessed 8 September 2021.

[12] Qi Xia, Winson Ye, Zeyi Tao, Jindi Wu, and Qun Li. A survey of federated learning for edge computing: Research problems and solutions. *High-Confidence Computing*, 1(1):100008, 2021.

[13] Hankz Hankui Zhuo, Wenfeng Feng, Yufeng Lin, Qian Xu, and Qiang Yang. Federated deep reinforcement learning, 2020.

[14] Sheyda Zarandi and Hina Tabassum. Federated double deep q-learning for joint delay and energy minimization in iot networks, 2021.

[15] Huy T. Nguyen, Nguyen Cong Luong, Jun Zhao, Chau Yuen, and Dusit Niyato. Resource allocation in mobility-aware federated learning networks: A deep reinforcement learning approach, 2019.

[16] Fariba Majidi, Mohammad Reza Khayyambashi, and Behrang Barekatain. Hfdrl: An intelligent dynamic cooperate cashing method based on hierarchical federated deep reinforcement learning in edge-enabled iot. *IEEE Internet of Things Journal*, pages 1–1, 2021.

[17] Ivan Kholod, Evgeny Yanaki, Dmitry Fomichev, Evgeniy Shalugin, Evgenia Novikova, Evgeny Filippov, and Mats Nordlund. Open-source federated learning frameworks for iot: A comparative review and analysis. *Sensors*, 21(1), 2021.