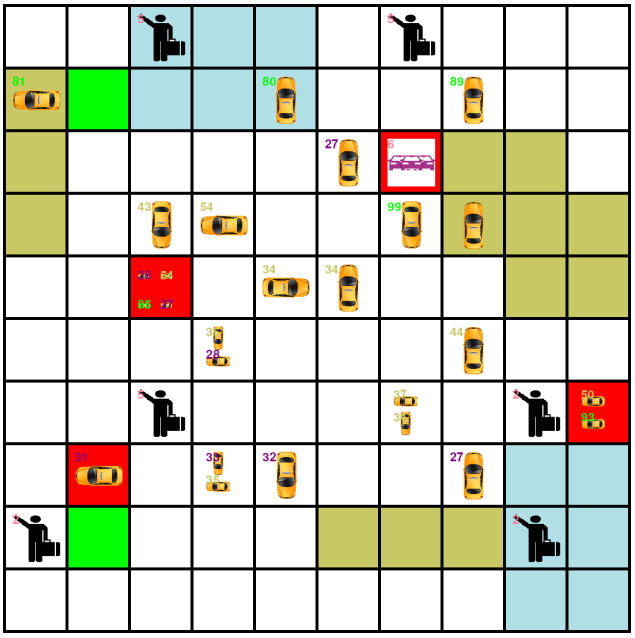
CHALMERS
UNIVERSITY OF TECHNOLOGY



Optimizing Electric Fleet Energy Consumption Using Deep Multi-Agent Reinforcement Learning

Master's thesis in Systems, Control and Mechatronics

ELIAS NEHMÉ

TONG ZOU

Department of Electrical Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Master's thesis 2020

Optimizing Electric Fleet Energy Consumption Using Deep Multi-Agent Reinforcement Learning

ELIAS NEHMÉ
TONG ZOU

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Systems and Control
Chalmers University of Technology
Gothenburg, Sweden 2020

Optimizing Electric Fleet Energy Consumption Using Deep Multi-Agent Reinforcement Learning

ELIAS NEHMÉ
TONG ZOU

Academic Supervisor and Examiner: Bengt Lennartson, Electrical Engineering
Supervisor: Johan Ek, China Euro Vehicle Technology AB (CEVT)

Master's Thesis 2020
Department of Electrical Engineering
Division of Systems and Control
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A screenshot from the developed simulation environment

Typeset in LaTeX
Gothenburg, Sweden 2020

Optimizing Electric Fleet Energy Consumption Using Deep Multi-Agent Reinforcement Learning

ELIAS NEHMÉ
TONG ZOU
Department of Electrical Engineering
Chalmers University of Technology

Abstract

Taxi fleet management is a dynamic problem concerned with matching available taxi drivers to customer orders. Not only does it concern the order matching, but it also has to take into account for cruising of the taxis, ensuring that they are spread out and readily available for customers in different areas. For an electric taxi fleet, another dimension, namely charging, is introduced. Now the taxis also have to consider when and where to charge, as this can be a time-consuming action.
In this thesis, the aim was to develop and implement multi-agent reinforcement learning algorithms to optimize the energy consumption of electric taxi fleets. Two decentralized multi-agent reinforcement algorithms were then developed and evaluated. A new algorithm is proposed called Reward Mixing Deep Double Q-Networks (RMDDQN) which factorize the contribution of each agent to the overall reward, improving the cooperation among agents. The other algorithm implemented is Counterfactual Multi-Agent Policy Gradients (COMA). The algorithms were trained and evaluated on two scenarios, one smaller and simpler, and one more dynamic and realistic. For comparison, a decentralized and centralized deterministic baseline agent was developed.
The experiments conducted show that in the smaller scenario of the simulated environment, the proposed algorithms outperform both the baseline agents in terms of optimizing energy consumption. In the larger scenario, RMDDQN outperforms both baseline agents, while COMA does not manage to beat either.
The main contribution of this thesis was firstly the newly developed environment suited for simulation of electric taxi fleet management problems. Secondly, the RMDDQN algorithm, which uses multi-agent global and local rewards along with Deep Q-Networks was proposed. Lastly, the effectiveness of MARL approaches to the electric fleet optimization problem was tested and evaluated. Here the proposed RMDDQN algorithm showed promising results for the experiments conducted.

Keywords: Reinforcement Learning, Multi-Agent Reinforcement Learning, Discrete-Event Systems, Taxi Fleet Management

Acknowledgements

We would like to first and foremost thank our supervisor and examiner Bengt Lennartson for his continued support and guidance during this thesis. His positiveness and encouragement kept us motivated and helped us to keep going.
Secondly, we would like to thank all the people of the Feature Labs team at CEVT. Our supervisor Johan Ek was always there to discuss and guide us when we faced new obstacles. Per Nilsson Lundberg introduced a lot of valuable tools to improve our workflow. We would also like to thank the manager of Feature Labs, Anders Werner, for always making sure we had everything we needed to successfully conduct this thesis.
Lastly, we would like to thank our family and friends for always your everlasting support. Without you, we wouldn't have come this far.

Elias Nehmé & Tong Zou, Gothenburg, August 2020

Contents

List of Figures

List of Tables

1
Introduction

In this chapter, the main tasks of the thesis are presented. In particular, the background and motivation of the thesis are introduced in Section 1.1. Following this, the objective is introduced in Section 1.2. In Section 1.3, the problem is limited and clarified. Then, the recent related research regarding using reinforcement learning to manage the taxi fleet is introduced. Finally, the thesis outline is introduced in Section 1.5.

1.1 Taxi fleet management using deep machine learning

With the development of smartphones and big data science, the taxi business has already extended to a new business model, ride-hailing, in which customers publish requirements, including information such as origin, destination, and time. Ride-hailing companies (e.g. DiDi, Uber, etc.) use intelligent algorithms to allocate vehicles to orders based on this information. As the core of the ride-hailing company, the order allocation algorithm greatly affects the order receiving efficiency and the company's profitability.
In the highly developed automobile industry, electric vehicles (EVs) have become more and more popular among the public for being energy-saving and environmentally friendly. Thus, using EVs as taxis has become a general trend in a sustainable context. For electric taxis, it is always desirable to maximize the revenue. But on top of that, the EVs have a higher need for energy efficiency, as charging is a more time-consuming action as compared to filling up with gas in traditional vehicles.
Last decades witnessed tremendous success in deep machine learning, which brought great improvements in areas such as computer vision, speech recognition, natural language processing, audio recognition, etc. All of these successful algorithms use deep neural networks [10] to approximate complex mathematical models, which convert a problem of modeling to an optimal parameter searching problem, greatly improving the efficiency of algorithm development. In some cases, deep machine learning requires large-scale historical data for training, while in the taxi fleet scenario all the behaviors such as pick up and drop off will impact the future data and decision making, so it is difficult to use supervised learning methods to solve these real-time problems. However, Reinforcement learning (RL) algorithms, in which an agent (i.e. a taxi) interacts with the environment by taking actions (e.g. go up, down, left and right or advanced actions such as pick up, drop off, etc.) based on its observation of the environment, can naturally generate data (i.e. observa-

tions) in every time step to support the decision making for the next step. With the assistance of neural networks, reinforcement learning can be extended to deep reinforcement learning (DRL), which due to its great function approximation capabilities, can solve problems of high dimensionality and complexity. In an RL setting, every taxi is viewed as an agent. Thus, for in a taxi fleet management scenario, an algorithm that can handle many agents is required. Multi-agent reinforcement learning (MARL) enables a large number of agents to make their independent decision based on their local observation to contribute most effectively to the overall reward.

1.2 Objective

As discussed in Section 1.1, there is a trade-off between saving energy and increasing revenue by picking up customers. In this thesis, the aim is to implement and evaluate MARL algorithms to optimize energy consumption in electric taxi fleet management problems. This means increasing the revenue earned in relation to the energy consumed.
The objectives and contributions of this thesis are thus outlined as:

- Develop a new modular environment capable of simulating the electric fleet management problem. This environment should both include charging dynamics and customers.
- Develop the extended Deep Double Q-Networks (DDQN) algorithm into the multi-agent setting using global and local reward mixing (RMDDQN). The Counterfactual Multi-Agent Policy Gradients (COMA) algorithm was implemented and evaluated.
- Evaluate the performance of implemented MARL algorithms for optimizing the energy consumption of an electric taxi fleet as compared to deterministic baselines. The algorithms were evaluated on how much revenue the total taxi fleet generates in relation to the energy spent. The total revenue was also of interest, as a very efficient policy is not desirable if it's generated revenue is unreasonably low.

1.3 Scope

In this thesis, the MARL algorithms are trained in a simplified simulated environment. The goal is to develop a high-level policy that is allocating taxis between charging and picking up customers, as well as cruising. Many dimensions of reality will be disregarded as they are outside of the scope of the agent. For instance, roads and traffic will not be taken into consideration. Simple linear models mostly is used to represent other dynamics of the system, such as battery charging, consumption, and movement of the agents. For the MARL algorithms, path-planning is outside of their scope. An offline path-planning algorithm is developed to ensure the agents move according to their high-level actions such as picking up a specific customer or charging at a specific charge station. The objective of the MARL algorithms is to find the optimal matching between agents and customers, optimizing the energy consumption, and making sure agents are charging.

1.4 Related work

Using MARL in fleet management settings has been explored by big ride-hailing companies such as DiDi and Uber. DiDi has divided the problem of fleet management into order dispatching and driver repositioning [8]. In [36], the authors proposed a MARL algorithm for order-dispatching trough order-vehicle distribution matching. In [33] the authors tackle the order dispatching problem as well, but they do it by modeling it as a Markov decision process and using a deep Q-network (DQN) to optimize a dispatching policy. In [8], deep reinforcement learning was used to manage both the dispatching and repositioning problems. As with the work of this thesis, this was a more generalized solution that did not only try to solve a subproblem of the fleet management problem. In this thesis, however, the goal is optimizing electric fleet management, and thus charging has to be introduced.
As no work on electric fleet management had been found, no simulation environments suitable for this were readily available. In [20], the authors proposed a DQN based framework to solve the dispatching problem, which was a large-scale simulator based on real-world taxi data. In [24], the authors proposed a DQN-based simulation environment that aims to handle both dispatching and repositioning. In this work, a simulator will be created to handle dispatching, repositioning, and charging.

1.5 Thesis outline

In Chapter 2 the relevant theory for this thesis is presented. This includes a brief overview of discrete event systems, and in particular, Markov decision processes. After that reinforcement learning is explained in the classic single-agent setting followed by the multi-agent setting. In Chapter 3, The methodology to solve the electric taxi fleet management problem in this thesis is explained. This starts with an in-depth explanation of the developed multi-agent environment, and with that, the two experimental scenarios which will be used to test the algorithms are presented. The method chapter then ends with explaining the developed MARL algorithms. In Chapter 4, the experiments done are explained and the results from them are showcased. In Chapter 5, the thesis ends with a conclusion and directions for future work.

2
Theory

In this chapter, the relevant theory and concepts used in this thesis will be introduced. First, an introduction of Discrete-event systems, and more specifically, Markov Decision Processes (MDP) will be given as it provides the formal framework which reinforcement learning is based on. After that, relevant concepts within reinforcement learning will be given, along with relevant algorithms for this thesis. Finally, the theory of Reinforcement Learning extends to multi-agent systems, and with that Multi-Agent Reinforcement Learning is introduced.

2.1 Discrete-event systems

A discrete event system (DES) is a system consisting of discrete states and events, where events transition the system from one state to another. Usually, more complex systems are composed of several subsystems, and thus even if the individual systems are quite small, the state space of the total integrated system can become very big, reaching millions of states [11]. For discrete event systems, there are plenty of algorithmic verification tools that can be used to model and design controllers as well as verify the proper functioning of a system.
Reinforcement learning environments can frequently be formally described as Markov Decision Processes (MDP) [27]. MDPs are a specific kind of DES that is used to find an optimal sequence of actions (events). In the case of continuous MDPs, the focus is on optimal control.

2.1.1 Markov Decision Processes

An MDP is a discrete event system based on sequential decision making to maximize the (future) reward signal. In an MDP the decision-maker, also known as the *agent*, exists in an environment with whom it can interact. At each time-step the agent observes the environment and gets a reward signal for the current state, and based on that observation and reward chooses an action that alters the state of the environment. At the next time step, the agent once again observes the environment and receives a reward signal and chooses a new action, and this cycle then continues. A visualization of the agent-environment interaction is shown in figure 2.1.
An important property in an MDP is that every state is Markov (also referred to as having the Markov property) [27]. The formal definition of a Markov state can be seen in Equation (2.1). In essence, a Markov state means that all the historical information is captured in the state, and thus the old states do not provide any new

Figure 2.1: The interaction between the agent and the environment. The Agent takes an action based on the state of the environment, which in turn changes the state and returns it to the agent along with a reward signal.

information, given the current state.

$$P[S_{t+1}|S_t] = P[S_{t+1}|S_1, ..., S_t] \tag{2.1}$$

An MDP can be described as a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ where $\mathcal{S}$ is the finite set of states, $\mathcal{A}$ is the finite set of actions, $\mathcal{P}$ is the state transition probability matrix, $\mathcal{R}$ is the finite set of rewards and γ is the discount factor. Here, the state transition probability matrix describes the probability of reaching a next state s' and receiving reward r given the state s and action a according to Equation (2.2). One can observe that the definition of the state transition probability $\mathcal{P}$ requires that every state s is Markov.

$$\mathcal{P}(s', r|s, a) \doteq \mathbb{P}[S_{t+1} = s', R_t = r|S_t = s, A_t = a] \tag{2.2}$$

The rewards $\mathcal{R}$ is the set of rewards that the agent receives from the environment. At every step, the agent receives a scalar reward $R_t \in \mathcal{R}$. In reinforcement learning the goal is to maximize the expected total discounted reward it receives, and not the immediate biggest reward at the current time step [28]. The future rewards are discounted by a factor of $\gamma \in [0, 1]$ for every time step, which describes how valued future rewards are. The expected discounted return can thus be formalized according to Equation (2.3).

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+1+k} \tag{2.3}$$

By tuning the discount factor γ, the goal of the agent can be altered. With $\gamma = 0$ the agent's only concern is about the next reward R_{t+1}, and the more γ approaches 1, the more the agent prioritizes future rewards.

Policies, State-Value Functions and Action-Value Functions To define the behavior of an agent, a policy π is used to map the distribution over actions from states. In MDPs, policies naturally only depend on the current state [27]. To estimate the value of different states, value functions, or action-value functions are used.
A state-value function $v_\pi(s)$ is a function describing the expected return when starting in state s and following the policy π, as seen in Equation (2.4)

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{2.4}$$

The action-value function $q_\pi(s)$, on the other hand, describes the expected return starting in state s and taking the action a, and after that following the policy π, as seen in Equation (2.5)

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{2.5}$$

Bellman equations By decomposing the state-value function into the immediate reward R_{t+1} and the discounted value of the next state $\gamma v_\pi(S_{t+1})$, the Bellman Equation for v_π can be formulated according to Equation (2.6)

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}|S_t = s] \\ &= \sum_{a\in\mathcal{A}} \pi(a|s) \sum_{s',r} \mathcal{P}(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{2.6}$$

Similarly, for the action-value function, the Bellman Equation can be formulated according to Equation 2.7.

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a] \\ &= \sum_{s',r} \mathcal{P}(s', r|s, a)[r + \gamma \sum_{a'\in\mathcal{A}} \pi(a', s') q_\pi(s', a')] \end{aligned} \tag{2.7}$$

Optimal Policies and Value functions In a sense, the goal of a MDP is to find the optimal value function, as it specifies how to achieve the optimal performance [27]. The optimal state-value function is defined according to Equation 2.8 and the optimal action-value function is defined according to Equation 2.9.

$$v_* \doteq \max_\pi v_\pi(s) \tag{2.8}$$

$$q_* \doteq \max_\pi q_\pi(s,a) \tag{2.9}$$

For any MDP, there always exists an optimal policy π_*, which is either better or equal to all other policies [27]. Optimal policies have the same optimal state-value function v_* as well as optimal action-value function q_*
Rewriting the Bellman Equation for the optimal state-value function v_*, the Bellman optimality Equation for v_* is achieved according to Equation (2.10).

$$v_*(s) = \max_{a\in\mathcal{A}(s)} q_{\pi_*}(s,a) = \max_a \sum_{s',r} \mathcal{P}(s', r|s, a)[r + \gamma v_*(s')] \tag{2.10}$$

The Bellman optimality Equation for q_* can be seen in Equation (2.11)

$$q_*(s,a) = \sum_{s',r} \mathcal{P}(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \tag{2.11}$$

2.2 Reinforcement Learning

Given Markov decision processes as explained in section 2.1.1, reinforcement learning (RL) algorithms try to find the best actions to maximize the cumulative reward from the environment. In other words, they seek to find the optimal value functions v_*, q_*, or directly the optimal policy π_*. One strength of RL algorithms is that many of them do not require any model of the environment. These *model-free* algorithms are very powerful when the exact environment is not known or is infeasibly large for an exact solution. The rest of this section will focus on presenting relevant RL algorithms for this thesis.

2.2.1 Temporal-Difference Learning

A very important class of reinforcement learning methods are the Temporal Difference (TD) methods. TD methods are model-free, meaning that they can learn directly from experience, and do not require an explicit model of the system dynamics. Another important feature is that they bootstrap, which means that they update their estimates based on old estimates (they don't have to wait for the outcome of an episode before updating). The perhaps simplest TD method, known as TD(0) [28] updates its value function directly after one transition from time step t to $t + 1$ according to Equation 2.12.

$$V(S_t) \leftarrow V(S_t) + \alpha \underbrace{[\overbrace{R_{t+1} + \gamma V(S_{t+1})}^{TD\ target} - V(S_t)]}_{TD\ error} \tag{2.12}$$

The TD target as seen in Equation (2.12) is the new estimate of the value function. The term referred to the *TD error* can be seen as an error between the target and the old estimate $V(S_t)$. The value function is then updated in this direction scaled by the learning rate α.

TD(0), as seen in Equation (2.12), updates its estimate on the next reward and bootstraps its estimate of the value of the next state. The TD algorithm can be extended to an n-step TD algorithm by modifying the amount of rewards that are used before bootstrapping. The return for the one-step TD(0) is simply $G_{t:t+1} = R_{t+1} + \gamma V_t(S_{t+1})$. A two-step return is thus $G_{t:t+2} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2})$. Extending this to an n-step return yields a return according to Equation (2.13)

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}) \tag{2.13}$$

TD(λ)

In the TD(λ) algorithm, all the steps of an n-step return are used and each weighted proportionally to a factor γ^{n-1}. Here γ should be chosen such that $0 \geq \gamma \leq 1$. The return is then normalized by $(1-\gamma)$ so that the sum of the weights equal to 1. This return, known as the λ-return can be seen in Equation (2.14).

$$G_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} \tag{2.14}$$

It can be noticed that for $\lambda = 0$, the return equals the one-step return $G_{t:t+1}$. By increasing γ, the higher weight the subsequent returns $G_{t:t+n}$ get.

Q-learning

An early breakthrough in reinforcement learning was the classic off-policy temporal difference control algorithm famously known as *Q-learning* [34]. Being an off-policy algorithm, it can learn the optimal action-value function q_* following any policy [28]. In essence Q learning consists of the following steps:

- Observe the current state S_t
- Perform action A_t selected by some policy derived from Q
- Observe the next state A_{t+1} and receive immediate reward R_{t+1}
- Based on this experience (transition), adjust the Q function as:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.15}$$

Thus the new Q-value is updated by the learning rate, α, multiplied by the temporal difference. Since the algorithm updates itself based on its existing estimate, the algorithm is a *bootstrapping* method.
An interesting dilemma regarding what policy to follow during training is the exploration versus exploitation dilemma. Intuitively, one might think that following the greedy policy (always choosing the action which maximizes the expected return) is optimal as you gather rewards from the states which according to the current action-value function seems best. However, this will lead to some potentially better states not being discovered [27]. One famous policy to deal with this problem is the $\epsilon - greedy$ algorithm.
The $\epsilon - greedy$ policy

- selects the optimal action $\arg\max_{a \in \mathcal{A}} Q(s,a)$ with probability $\mathbf{1 - \epsilon}$
- selects a random action $a \in \mathcal{A}$ with probability $\boldsymbol{\epsilon}$

Thus the algorithm ensures exploration infinitely. Another strategy based on this algorithm is the *decaying* $\epsilon - greedy$ *algorithm*. Here the exploration probability ϵ decays with time, ensuring higher exploration in the beginning when the agent knows very little about the environment, but lower exploration with time so that the agent can exploit the optimal policy more as it learns more about the environment.

2.2.2 Policy Gradient Methods

Unlike the value-based methods mentioned thus far, policy gradient methods directly learns a parameterized policy $\pi(a|s, \boldsymbol{\theta})$, and thus does not depend on a value function or action-value function to select actions. The learning of this policy is based on gradient ascent on some scalar score function $J(\boldsymbol{\theta})$ defining the performance:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \underbrace{\nabla J(\boldsymbol{\theta}_t)}_{Policy\,Gradient} \tag{2.16}$$

For episodic tasks, the function is usually set to the value function for the policy determined by $\boldsymbol{\theta}$ according to Equation (2.17)

$$J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(s) \tag{2.17}$$

With the help of the policy gradient theorem, an analytic expression can be achieved for the policy gradient according to Equation (2.18) [27].

$$\nabla J(\boldsymbol{\theta}) = \mathbb{E}_{\pi}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s, a) Q_{\pi_{\boldsymbol{\theta}}}(s, a)] \tag{2.18}$$

One way to reduce variance is by subtracting a baseline function $B(s)$. If used properly, this can reduce variance without changing the expectation. A commonly used and good baseline is the value function [27]:

$$B(s) = v_{\pi_{\boldsymbol{\theta}}}(s) \tag{2.19}$$

By defining an *advantage function* as $A(s, a) = Q(s, a) - B(s)$ the policy gradient can be written according to Equation (2.20).

$$\nabla J(\boldsymbol{\theta}) = \mathbb{E}_{\pi}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s, a) A_{\pi_{\boldsymbol{\theta}}}(s, a)] \tag{2.20}$$

In cases with discrete action spaces it is common to design the function approximator to output a scalar value $h(s, a, \boldsymbol{\theta})$ for each state-action pair. The policy $\pi(a|s, \boldsymbol{\theta})$ can then be gained by transforming the values with an exponential softmax distribution according to Equation (2.21).

$$\pi(a|s, \boldsymbol{\theta}) = \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_i e^{h(s,i,\boldsymbol{\theta})}} \tag{2.21}$$

Thus the actions will get a probability proportional to the value of the action preference $h(s, a, \boldsymbol{\theta})$ for the given state. The action with the highest value will get the highest probability and so on. This kind of policy parameterization is refereed to as *softmax in action preferences* [28]. One advantage with this type of parameterization is that the learned policy $\pi(a|s, \boldsymbol{\theta})$ will be stochastic as compared to value function algorithms, which are deterministic in nature. In naturally stochastic games such as some card games, this is very advantageous. If the true policy happened to be deterministic, the optimal action probabilities of the parameterized policy would be driven towards being infinitely higher than the sub-optimal ones and the policy will end up being deterministic.

Another advantage in general with policy gradients methods using softmax in action preferences is the smooth changes in the action probabilities, as compared to value-based algorithms. Sometimes small changes in the value function can change the optimal action and thus produce big changes. Partly due to this policy gradient methods often have stronger convergence guarantees [28].

Actor-Critic

Actor-critic (AC) methods combine value-based and policy gradient algorithms. The critic estimates the action-value function

$$Q_{\boldsymbol{w}} \approx Q_{\pi_{\boldsymbol{\theta}}}(s, a) \tag{2.22}$$

and the actor selects which action to take. Thus the algorithm has two sets of parameters, the critic action-value parameters $\boldsymbol{w}$ and the actor policy parameters $\boldsymbol{\theta}$. The role of the critic is thus to update the action-value function $Q_{\boldsymbol{w}}$ and the actor to update the policy parameters by gradient ascent in the direction given by the critic. As compared to the Equation (2.18), the policy gradient is now approximated as given in Equation (2.23) [27].

$$\nabla J(\boldsymbol{\theta}) \approx \mathbb{E}_{\pi}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s, a) Q_{\boldsymbol{w}}(s, a)] \tag{2.23}$$

It should be noted that the job of the critic is evaluating the current policy $\pi_{\boldsymbol{\theta}}$, and thus methods mentioned earlier in this chapter such as temporal difference algorithms can be used for this.
An overview of the interaction between the actor, the critic, and the environment can be seen in figure 2.2

2.2.3 Deep Reinforcement Learning

A major limitation of traditional reinforcement learning is the ability to handle problems concerning large state spaces [18]. Using deep neural networks, the value of states can be generalized much more efficiently. Another great power of using deep neural networks as function approximators, especially when using deep convolutional networks, is the possibility of using sensor data as inputs to the network [18]. Commonly you have the observed state as input to the network, and the output will then be the value or action values at the given state.
Deep reinforcement learning combines the power of deep neural networks with reinforcement learning by using the networks to approximate the value function $v(s; \boldsymbol{\theta})$, action-value function $q(s, a; \boldsymbol{\theta})$ or the policy $\pi(s; \boldsymbol{\theta})$. Here, $\boldsymbol{\theta}$ is the network parameters. Linear function approximators were popular in the field of RL, but with the recent advances of deep learning, deep networks have steadily increased in popularity and made it possible to solve large problems that were intractable with traditional methods [28].

Deep Q-Networks

A problem with nonlinear function approximators in reinforcement learning is that they are unstable and sometimes even diverge [12]. There are several reasons causing

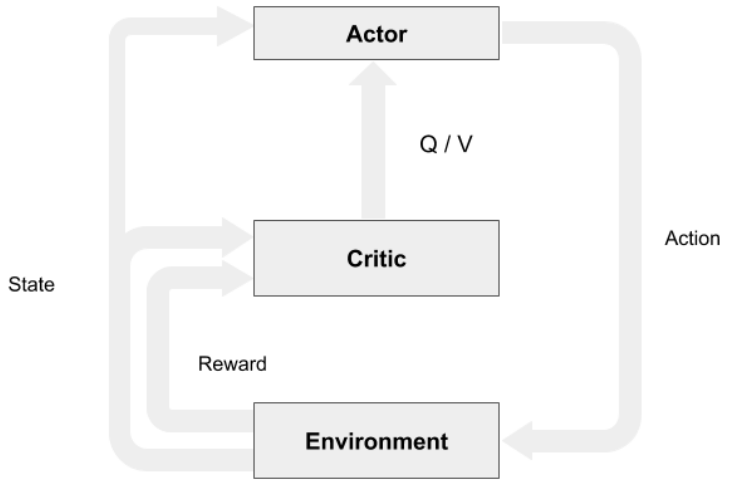


Figure 2.2: The interaction between the actor, the critic, and the environment. The critic receives the state and reward from the environment and evaluates the current policy. The actor receives the state from the environment and the updated value function V or action-value function Q from the critic. The actor then uses this to update the policy and then select a new action.

these instabilities; one being the correlation in the observation sequence. Small updates to the action-value function may result in big changes in the policy and thus the future data from the model will be affected. Another issue is the correlation between the action values Q and the target values $r + \gamma \max'_a Q(s', a')$ [18]. In 2015, [18] combined deep neural networks with Q-learning to develop a deep Q-network (DQN). Given a state $s \in \mathcal{R}^m$ it outputs a vector of Q-values for each action. Given an action space of $\mathcal{R}^n$, the network is thus a mapping $s \in \mathcal{R}^m$ to $\mathcal{R}^n$. The authors addressed the issues concerning deep neural networks as function approximators with two main features. The first is known as *experience replay*, which stores the data in a buffer and returns randomized batches, removing the correlation of the sequential observations. The second feature was an iterative update of the action-value function Q toward the TD target value, which is updated periodically to reduce correlations. The TD-target in DQN is thus

$$R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_{offline}) \tag{2.24}$$

where $\boldsymbol{\theta}_{offline}$ is periodically updated such that $\boldsymbol{\theta}_{offline} = \boldsymbol{\theta}$.

Deep Recurrent Q-Networks

Given the power of DQN, one shortcoming is that it expects the full state S_t of the environment. In other words, it expects a fully observable environment. In partially hidden environments, the agent can only receive an observation o of the true state, and consecutively they learn the action-value function $Q(o, a)$, which in

general is not equal to $Q(s, a)$ [5]. To address this, [5] developed a Deep Recurrent Q-Network (DRQN). By adding recurrency to the network, the DRQN can aggregate observations over time and better estimate the true system state S_t.

Deep Double Q-networks

Due to the nature of Q-learning where the same Q-values are used to both evaluate and select actions, the algorithm can be prone to overestimation of Q-values. To avoid this problem, Van Hasselt [4] presented double Q-learning, which decouples action selection and evaluation. In double Q-learning, two action-value functions with separate weights θ and θ' are learned, where the first is used to determine the greedy policy and the latter determine it's value. The TD-target for the algorithm can be seen in Equation (2.25).

$$Y_t^{DoubleQ} = R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}'_t) \tag{2.25}$$

By combining the idea of double Q-learning algorithm [4] with the DQN [18] for use with deep neural networks, van Hasselt et al. proposed Deep Double Q-networks (DDQN) [31]. The online network is used to choose the action, and the offline target network $\boldsymbol{\theta}_{offline}$ is used to estimate its value according to Equation (2.26).

$$Y_t^{DoubleQ} = R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t^{online}); \boldsymbol{\theta}_t^{offline}) \tag{2.26}$$

2.3 Multi-Agent Reinforcement Learning

In multi-agent reinforcement learning, the single-agent reinforcement learning framework described is extended to multi-agent systems. A MARL system can be formalized as a Markov game, also known as a stochastic game [2, 13]:

$$G = \langle \mathcal{S}, \mathcal{U}, \mathcal{P}, \mathcal{R}, \mathcal{O}, n, \gamma \rangle \tag{2.27}$$

Here n is the number of agents $a \in \mathcal{A} = \{1, 2, ..., n\}$. $\mathcal{S}$ is the state space, and s denotes the state of the environment. $\mathcal{U}$ is the action sets of the agents. Each agent a receives a reward according to the reward function $r(s, \boldsymbol{u}, a) : \mathcal{S} \times \mathcal{U} \to \mathcal{R}$, and $\mathcal{O}$ is the observation function: $o(s, a) : \mathcal{S} \times \mathcal{A} \to \mathcal{O}$. Lastly, γ is the discount factor just as in a regular MDP (see section 2.1.1). A bold font will be used to mark joint quantities. The interaction between the agents and the environment as compared to the single-agent case in figure 2.1 is visualized in figure 2.3.
At each time step in the Markov game, each agent selects an action $u_t^a \in \mathcal{U}$ forming the joint action $\boldsymbol{u}$. Given the joint action, the environment takes a step and generates a new state $s_t \in \mathcal{S}$. In the case of a partially observable environment, the agents draw an observation o_t^a according to the observation function $o(s, a)$. If the environment is fully observable, each agents observation is the global state, $o_t^a = s_t$ In the case of individual rewards, each agent draws a reward r_t^a according to the reward function $r(s, \boldsymbol{u}, a)$. In some cooperative settings, all agents may receive the same reward signal $r(s, \boldsymbol{u}, a) = r(s, \boldsymbol{u}, a') \; \forall a, a'$ [2].

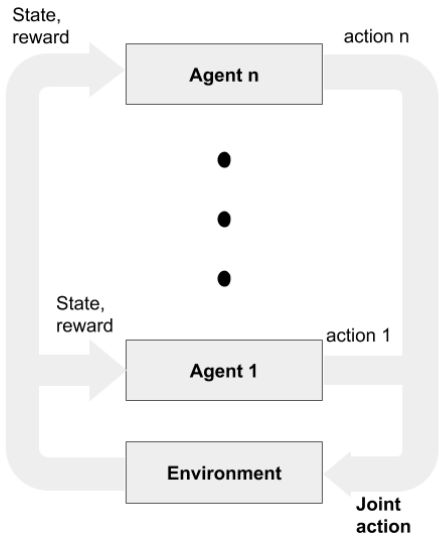


Figure 2.3: The interaction between the agents and the environment. The Agents each select an action to form the joint action $\boldsymbol{U}$, on which the environment then transitions into a new state.

Cooperation or Competition Markov games are usually split up into three categories: cooperation, competition, or general-sum. In a fully cooperative setting, all agents have the same main goal and should learn to cooperate to maximize the joint reward. As mentioned, in a fully cooperative setting it is common to have the same global reward signal r_t for all agents. In some cooperative settings it is however easy, and sometimes advantageous to have local rewards $r_t = \sum_a r_t^a$. This can help agents to more easily learn a policy to maximize their individual reward, simplifying the problem of credit assignment (see section 2.3.1). The side-effect is that as each agent is only seeing their own local reward, and not the global reward r_t, they will avoid making sacrifices that may hurt their local reward, even it would increase the global reward.
In contrast, a cooperative setting, also known as a zero-sum setting is where the total reward is always zero and thus one agent's reward is another agent's loss as seen in equation (2.28).

$$\sum_a R(s, \boldsymbol{u}, a) = 0, \ \forall s, \boldsymbol{u} \tag{2.28}$$

A middle ground between cooperative and competitive setting is general-sum settings. Here the agents are neither fully cooperative nor competitive.

2.3.1 Challenges in MARL

In MARL theory, several issues arise that are not present in single-agent reinforcement learning. In this thesis, three challenges that are of interest will be presented, namely non-stationary environments, multi-agent credit assignment, and lastly scalability of MARL systems.

Non-Stationary Environments One major difference between single-agent RL and MARL is the environment. In a single-agent setting, only one agent is learning, and thus the environment is stationary from the perspective of the agent. In MARL, each agent's perceived environment is filled with other agents that are exploring and learning, causing the environment to become non-stationary. This invalidates the very important Markov property (see section 2.1.1) of the environment which in turn means many mathematical tools from single-agent RL will be invalid [35, 25]. Algorithms that simply treat the environment as stationary and ignores this issue are known as *independent learners*. Examples are Independent Q-Learning [29] and Independent Actor-Critic [3].

Multi-Agent Credit Assignment Another issue commonly seen in cooperative MARL settings is multi-agent credit assignment[2]. In environments where the joint action only generates a global reward, it's difficult for individual agents to reason about their contribution to the global reward. As mentioned, handcrafted individual rewards can help with this, but in return make agents more selfish and potentially not sacrifice their own reward even if it would increase the team global reward as they can only see their own local reward.

Scalability The third major challenge in MARL is the scalability. In many fully observable problems a centralized controller which is a mapping from the state of the environment to a probability distribution over the joint action $\boldsymbol{u}$

$$\pi^C(\boldsymbol{u}, s_t) : \mathcal{U} \times \mathcal{S} \rightarrow [0, 1] \tag{2.29}$$

is theoretically applicable, but since the joint action space $\mathcal{U}$ grows exponentially with the number of agents and that the global state space increases as well, this approach quickly becomes intractable for larger problems[2, 35, 6]. Thus decentralized control can be helpful, where each agent has its own policy $\pi^a(u^a, s_t)$ which maps from the state of the environment to a probability distribution over the specific agent's actions. In partially observable settings, this mapping can be conditioned on the local observation of the specific agent o_t^a instead of the global state of the environment s_t, thus managing the growing state space as well. The joint action probability distribution would then be factorized according to Equation 2.30.

$$P(\boldsymbol{u}, s_t) = \prod_a \pi^a(u^a | o_t^a) \tag{2.30}$$

2.3.2 Counterfactual Multi-Agent Policy Gradients

Counterfactual Multi-Agent Policy Gradients (COMA) is a multi-agent actor-critic algorithm proposed in [3]. The algorithm mainly relies on three ideas. Firstly, COMA uses a centralized critic which is shared by all agents. This critic is only used during training, and during execution, only the decentralized actors are used. Thus the algorithm is still fully decentralized during execution. Having centralized training and decentralized execution through a centralized critic and decentralized actor has also been explored by other works such as [14, 9].
Secondly, to deal with a multi-agent credit assignment, COMA uses a counterfactual baseline. The idea stems from difference rewards [23]. In COMA this idea is used

by having the critic compute a unique advantage function for each agent, which compares the estimated return given the chosen joint action with a counterfactual baseline that marginalizes the given agent's action, while all the other agent's actions remain fixed. This gives the agents a way to evaluate their contribution to the global reward.
Thirdly, COMA uses an efficient critic representation that makes it possible for each agent to compute the Q-values for all different actions u^a while being conditioned of all the other agent's actions $\boldsymbol{u}^{-a}$.
COMA thus computes the counterfactual baseline according to Equation (2.31).

$$b^a(s, \boldsymbol{u}) = \sum_{u^a} \pi^a(u^a|\tau^a) Q(s, \boldsymbol{u}^{-a}, u^a) \tag{2.31}$$

Here $Q(s, \boldsymbol{u}^{-a}, u^a) = \{Q(s, \boldsymbol{u}^{-a}, u^a = 1), Q(s, \boldsymbol{u}^{-a}, u^a = 2), ..., Q(s, \boldsymbol{u}^{-a}, u^a = |U|)\}$ are the Q-values for each of agent a's actions while the other agents actions are fixed. This thus gives the baseline a measurement of the counterfactuals for each of agent a's actions while the rest remain fixed.
Given the baselines, the agent-specific advantage functions can be calculated according to Equation (2.32).

$$A^a(s, a) = Q(s, \boldsymbol{u}) - b(s, \boldsymbol{u}) = Q(s, \boldsymbol{u}) - \sum_{u^a} \pi^a(u^a|\tau^a) Q(s, \boldsymbol{u}^{-a}, u^a) \tag{2.32}$$

Given the advantage, the actor can then calculate the policy gradient according to Equation (2.20). The parametrized policy function can then be updated by gradient ascent according to Equation (2.16). As with standard AC methods, the critic can be trained using temporal difference methods such as $TD(\lambda)$.

2.4 Summary

In this chapter, the relevant theory for the thesis was presented. This included Discrete-event systems, and specifically, Markov Decision Processes (MDP). An MDP is usually the formal framework in which reinforcement learning tries to solve. The chapter then continued with reinforcement learning, where relevant methods based on both temporal difference learning and policy gradient methods were presented. The section ended with a look at deep reinforcement learning, The last section of the chapter presented multi-agent reinforcement learning, which is a central part of this thesis. Here the differences and new challenges as opposed to the single-agent case were presented. The MARL algorithm COMA algorithm was also presented.

3
Method

In this chapter, the complete workflow and methodology of the thesis are introduced. Firstly, the development setting is briefly explained to give an idea of the tools used in this thesis. The chapter then continues with a detailed presentation of the developed grid world environment in which the agents trained in. The chapter then continues by presenting the two algorithms used in this thesis. First is the new extension of DDQN into the multi-agent setting, namely, Reward Mixing DDQN (RMDDQN). After that, the implementation of the second algorithm, COMA, is explained.

3.1 Development setting

To develop the algorithms and the environment several tools were needed. All of the code developed was written in Python [32] due to its extensive popularity within data science and machine learning, as well as the number and quality of finished libraries it has within these areas. To develop the environment, mainly the standard python library and NumPy [21] were used. The visualization of the environment was made using Pygame [26] due to its simplicity and effectiveness. The MARL algorithms were developed using Pytorch [22]. Pytorch was chosen for several reasons. The main reason is that it works seamlessly with NumPy, making conversion between data types from the NumPy-based environment to the algorithms back and forth very simple. Pytorch and its syntax are also very simple and familiar to regular Python, which leads to faster implementations of new ideas.
Training of the algorithms was mainly done on GPU:s, aside from debugging and during the development of the code. The algorithms were both trained on the local computers as well as remotely trained on more powerful servers with multiple GPU:s to accelerate training even further. The code was deployed on the remote machines trough lightweight Linux containers created using Docker [17]. To monitor training and efficiently analyze all parameters and variables during the training process, TensorBoard was used [1]. TensorBoard is a powerful visualization tool that lets you track your training progress, analyze your networks, and much more seamlessly through one web interface.

3.2 Grid World Environment

This section presents the developed multi-agent grid world based reinforcement learning environment which was developed to train the MARL algorithms in. Firstly,

a formal definition of the environment as a Markov game will be presented. After that, the different features and parts of the environment will be explained.

Formal Definition

To train the multi-agent system, a Markov game environment suitable for an electric taxi fleet order- and charging-dispatching as well as repositioning had to be developed as no suitable environment was found. The goal of the environment is a Markov game that generates data in a simulation on which MARL algorithms can interact with and learn in. As explained in section 2.3, a Markov game G can be defined by the tuple

$$G = \langle \mathcal{S}, \mathcal{U}, \mathcal{P}, \mathcal{R}, \mathcal{O}, n, \gamma \rangle. \tag{3.1}$$

The grid world environment can be formally defined as

$$G_{gw} = \langle G, b, \mathcal{Q}, \mathcal{L}, \mathcal{W}, T \rangle \tag{3.2}$$

where $b = \langle b_r, b_c \rangle$ is the size of the grid world, b_r is the number of rows, and b_c is the number of columns. The charge stations are defined by $q \in \mathcal{Q}$. Agents will recharge their state of charge (SOC) for every time step they remain in the charge location. $\mathcal{L}$ is the set of living areas and $\mathcal{W}$ is the set of working areas. These are areas that have unique customer distributions, meaning the number of customers spawning here can be modified, and thus does provides more realism and complexity to the environment. These areas will be explained more in detail in Section 3.2. The time step of the environment t is bounded by $t \in \langle 0, 1, ...T \rangle$. In figure 3.1 an example of a grid world is shown.

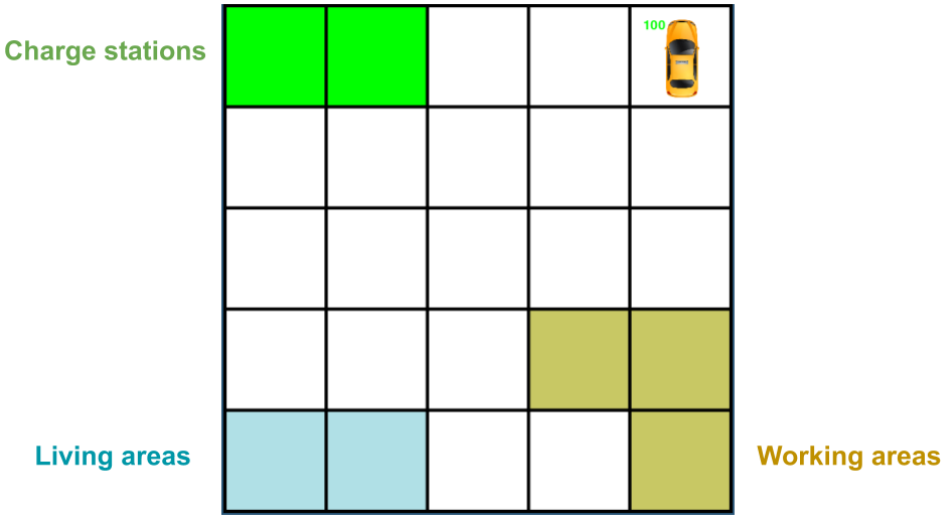


Figure 3.1: An example of the grid world environment where $s = \langle 5, 5 \rangle$. Here the set of charge stations is $\mathcal{Q} = \{q^1, q^2\}$, where $q_p^1 = \langle 0, 0 \rangle, q_p^2 = \langle 0, 1 \rangle$. The set of living areas is $\mathcal{L} = \{\langle 4, 0 \rangle, \langle 4, 1 \rangle\}$ and the set of working areas is $\mathcal{W} = \{\langle 3, 3 \rangle, \langle 3, 4 \rangle, \langle 4, 4 \rangle\}$.

Grid World Agents

The agents in the grid world environment $a \in \mathcal{A} = \{1, 2, ..., n\}$ can each be defined as

$$a = \langle p_t^a, c_t^a, c_+^a, c_-^a \rangle \tag{3.3}$$

where $p_t^a = \langle p_{r,t}^a, p_{c,t}^a \rangle$ is the position of the agent defined by the row position $p_{r,t}^a \in \{0, 1, ..., s_r\}$ and the column position $p_{c,t}^a \in \{0, 1, ..., s_c\}$. $c^a \in [0, 100]$ defines the state of charge (SOC) of the current agent. When moving in the environment, c_t^a decreases by the SOC decay rate c_-^a at each time step, and when the agent charges at a charge station, the SOC increases by the charge rate c_+^a at each time step.
When the environment is initialized, the set of agents $\mathcal{A}$ is initialized as well. For every agent, the initial position p_0^a is randomized, but the SOC is set to full charge:

$$c_0^a = 100, \ \forall a. \tag{3.4}$$

In this thesis all agents will always share the same charging dynamics:

$$\begin{matrix} c_+^a = c_+^{a'} \\ c_-^a = c_-^{a'} \end{matrix} \quad \forall a, a'. \tag{3.5}$$

At each time step, the agent can either stand still or move one block in the grid world. The agents move function can take a target location further away as input, however. In that case, the agent will use a simple path-planning algorithm to decide what one-step movement to take towards the target location. The pseudo-code for the path-planning algorithm can be seen in Algorithm 1.

Algorithm 1 Agent Move Function

Input: position $p = \langle r_p, c_p \rangle$, target $t = \langle r_t, c_t \rangle$, grid size $b = \langle b_r, b_c \rangle$
Output: new position $p_n = \langle r_n, c_n \rangle$

```
p_n = p
if target ≠ position then
    Δr = |r_p − r_t|
    Δc = |c_p − c_t|
    d = arg max(Δr, Δc)
    step = sign(t[d] − p[d])
    if (p[d] + step) ≤ s[d] then
        p_n[d] = p_n[d] + step
    end if
end if
```

Customers

Customers z spawn stochastically in the grid world, and the set of customers in the grid world is defined by $\mathcal{Z}$. The amount of customers is proportional to the size of the environment which is spawned follows the normal distribution $Z \sim \mathcal{N}(\mu, \sigma^2)$, with mean $\mu = \frac{b_r \cdot b_c}{2\tau(t)}$ and variance $\sigma^2 = \frac{b_r \cdot b_c}{20\tau(t)}$. The variable $\tau(t)$ is used to modify

the distribution over time steps, providing periods with more customers and periods with less customers.
Each customer spawns at a random location $z_p = \langle z_r, z_c \rangle$, and has a random goal destination. The spawn location cannot be a charging position. The probability of spawning at each time step can be customized between normal blocks, living areas, and working areas. The reward received by the agent for a pickup is always static and does not depend on the distance between the customer location and destination. The energy consumption of the agent is proportional to the distance. It should be noted that to reduce the complexity of the environment from the agent's perspective, the agent will not be teleported to the destination when picking up a customer. Instead, the agent will get the pickup reward and consume energy proportional to the distance traveled, but his position will remain the same. Each customer also has a waiting time limit, specifying how many time steps the customer will stay in the environment. If no agent picks him up within this time limit, the customer will disappear. The current waiting time of each customer at time step t is denoted by $z_{w,t}$. Customers spawn according to a normal distribution with a tune-able variance to suit the desired scenario. The distribution parameters have normal values and "peak hour" values. If the time step is within the peak hour time steps, that distribution will be used. With this, a more complex and realistic customer order distribution can be created. The frequency of spawning can also be modified so that customers spawn every x number of time steps.

Charge stations

Charge stations $q \in \mathcal{Q}$ are blocks that have different dynamics than the rest of the blocks. A charge station q is defined by its position q_p and the ordered set of agents currently in the charge station q_a.
On charge stations, no customers spawn. Instead, an agent a who remains on a charge station will get their SOC recharged by c_+^a every time step. Every charge station can however only charge a set amount of agents at the same time q_{max}, and it is always the agents who have been there the longest that will get to charge. The status of a charge station is defined according to equation (3.6)

$$q_{status} = \begin{cases} 1, & |q_a| \leq q_{max} \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

State and Observations

To efficiently represent grid locations in one dimension, grid blocks can be represented by a single number according to figure 3.2.
Utilizing this one-dimensional representation of the grid is useful when building a one-dimensional state or observation of the environment. The global state of the environment contains one entry per grid block, and three entries per agent in the grid world. The size of the state is thus $|\mathcal{S}| = b_r \cdot b_c + 3n$. If a block represents a customer block (non-charge station), the entry in the state is the current waiting time of the customer. If there is no customer, the feature will be represented by a zero. If the grid block is a charging station, the entry will represent the status of

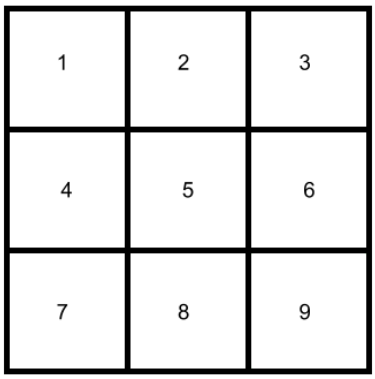


Figure 3.2: Figure showing how two-dimensional grid locations are represented in one dimension

the charge station, meaning it is 0 if the charge station is full, and 1 if an agent can charge there. For every agent, the state will contain the row, column, and SOC of the agent. The full state of the environment s at time-step t is thus defined as

$$s_t = \begin{bmatrix} \text{grid block 1: } z_{w,t} \text{ or } q_{status} \\ . \\ . \\ . \\ \text{grid block } b_r \cdot b_c\text{: } z_{w,t} \text{ or } q_{status} \\ p^1_{r,t} \\ p^1_{c,t} \\ c^1_t \\ . \\ . \\ . \\ p^n_{r,t} \\ p^n_{c,t} \\ c^n_t \end{bmatrix} \tag{3.7}$$

Each agent draws their local observations o^a_t based on the sizes of three buffers: a customer buffer n_z, an agent buffer n_a, and a charge buffer n_q. These buffers define how many reachable customers (the distance to them is less than their waiting time left), agents, and charge stations respectively each agent can see within their local observation. Given buffer sizes of $3, 2, 5$ for example, each agent will be able to see the 3 closest customers, the 2 closest other agents, and the 5 closest charge stations. The distance to these objects will be relative, calculated from the agent's position. Aside from that, each agent also observers the global time step, his global position,

and his SOC. The observation can thus be defined as

$$o_t^a = \begin{bmatrix} t \\ p_{r,t}^a \\ p_{c,t}^a \\ c_t^a \\ z_{r,t}^1 - p_{r,t}^a \\ z_{c,t}^1 - p_{c,t}^a \\ . \\ . \\ . \\ z_{r,t}^{n_z} - p_{r,t}^a \\ z_{c,t}^{n_z} - p_{c,t}^a \\ p_{r,t}^{-a=1} - p_{r,t}^a \\ p_{c,t}^{-a=1} - p_{c,t}^a \\ . \\ . \\ . \\ p_{r,t}^{-a=n_a} - p_{r,t}^a \\ p_{c,t}^{-a=n_a} - p_{c,t}^a \\ q_{r,t}^1 - p_{r,t}^a \\ q_{c,t}^1 - p_{c,t}^a \\ . \\ . \\ . \\ q_{r,t}^{n_q} - p_{r,t}^a \\ q_{c,t}^{n_q} - p_{c,t}^a \end{bmatrix}. \tag{3.8}$$

All values in the state s_t and the observations o_t^a are normalized by their maximum values such that each element is within the range $[0, 1]$.

Action Modes

Several action modes were designed for the environment. The action modes are different functions mapping the agent's integer output to movements in the environment. The action mode function is denoted $m(a, p, u)$.

Action mode 0: Global grid actions In this action mode, the action space of the agents is equal to the number of grid blocks $|\mathcal{U}| = b_r \cdot b_c$. Each action will correspond to a movement towards the block of that index. If an agent thus chooses action 5, this would correspond to moving on step towards block 5 as shown in figure 3.1 according to the agent move function seen in algorithm 1. If the agent is already at the desired block, he will remain still. Even though agents can only move one grid block each time steps, this action mode gives a bigger insight into what locations on the grid world the agents want to go to. This makes it easier to extract the long term plans of the agents.

Action mode 1: Regular grid actions This action mode is maps the agents network output to regular grid actions according to equation (3.9). The action space is always $|\mathcal{U}| = 5$.

$$\begin{aligned} &0 : \text{Stand still} \\ &1 : \text{Up} \\ &2 : \text{Right} \\ &3 : \text{Down} \\ &4 : \text{Left} \end{aligned} \tag{3.9}$$

Action mode 2: High-level actions In this action mode, the buffers from the observation are being utilized. For each customer in the customer buffer, the agent will have an action corresponding to moving towards that customers location. For every charge station in the charge buffer, the agent will also have an action for moving towards that location. The agent also has an action to stand still. With these actions, it will be easier for agents to map their actions to meaningful tasks and removes the path planning problem from the perspective of the agent. Thus the action space can be described according to equation (3.10). The size of the action space is $|\mathcal{U}| = 1 + n_z + n_q$

$$\mathcal{U} = \begin{cases} 0 : & \text{Stand still} \\ 1 : & \text{Customer 1} \\ . & . \\ . & . \\ . & . \\ n_z : & \text{Customer } n_z \\ n_z + 1 : & \text{Charge station 1} \\ . & . \\ . & . \\ . & . \\ n_z + n_q : & \text{Charge station } n_q \end{cases} \tag{3.10}$$

Action mode 3: Combined high-level and grid actions Inspired by [3], this action mode combines the regular grid actions with high-level actions, giving the agents very high flexibility at the cost of a larger action space.

System Dynamics

The dynamics of the system are contained within the step function of the grid world environment. The step function takes as input a list containing the action of each agent, and with that updates the environment one time-step. It then returns the new state of the environment, an observation for each agent, and a boolean defining if the environment has reached a terminal state. The pseudo-code can be seen in algorithm 2.

Algorithm 2 Environment Step Function

Input: List of actions $[u_t^1, u_t^2, ...u_t^n]$, action mode $m(a, p, u)$
Output: State s_t, observations $o_t^1, o_t^2, ..., o_t^n, t_{ts}$, and terminal state boolean t_b

```
for all agents a ∈ 𝒮 do
  p^a_{t+1} = m(a, p^a_t, u^a_t) Move agent according to action u^a_t and given action mode
  for all z ∈ 𝒵 do
    if z_{w,t} ≥ then
      delete z from 𝒵
    else if p^a_t = z_p then
      Agent a receives pickup reward and loses SOC proportional to customer distance
      delete z from 𝒵
    end if
  end for
  for all q ∈ 𝒬 do
    for all a ∈ q_a do
      if Index of a ≤ q_max then
        c^a_t = c^a_t + c^a_+        // Agent a charges
      end if
    end for
  end for
end for
Sample amount of new customers to spawn according to Z ∼ 𝒩(μ, σ²)
Generate state s_t, observations o^a_t
if t = T then
  t_b = true
else
  t_b = false
end if
```

Visualization

The visualization of the environment was developed using PyGame. The visualization tool provided a good way to see the agent behavior and helped to quickly discover abnormalities and bugs in the code. In Figure 3.1 an example of the environment and different block types was shown. Besides the different block types, the agent can be seen as the yellow cab. In green, his current SOC is displayed. As he moves and consumes energy, this number will update to show its SOC. As seen in Figure 3.3, a charge block will go from green to red when an agent is charging at the specified location. In the figure, an agent is also visualized. The number next to the agent represents his waiting time left before he disappears.

Deterministic Baseline Agents

To evaluate the performance of the MARL-algorithms, two baseline algorithms were developed. The first one is a centralized, deterministic baseline algorithm. Although

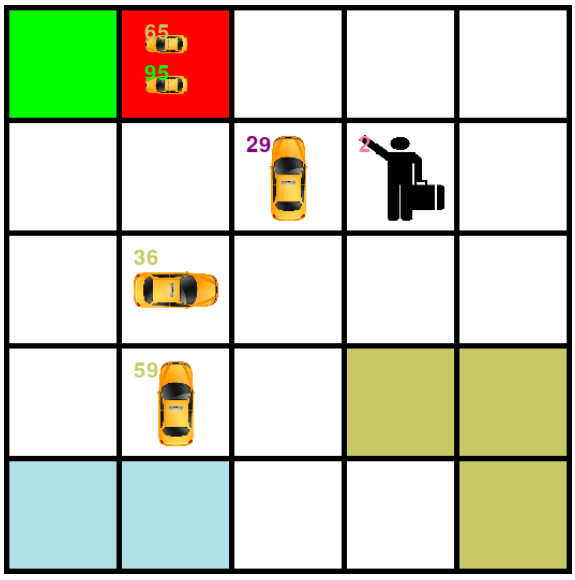


Figure 3.3: Figure showing some additional visual features of the environment. When an agent charges on a charging block, it turns red. a customer is seen in block $\langle 1, 3 \rangle$, and the number in the block represent how many time steps he has waited, $z_{w,t}$. Several agents are also seen in the figure along with their state of charge.

the algorithm is not optimal, it provides a reasonable baseline. Unlike the MARL-algorithms, the baseline is centralized, giving it a coordination advantage. The baseline algorithm also has access to the full state of the environment, giving it a second advantage. The basic idea of the algorithm is that for each agent, the closest customer is picked up. If no customers are available, the agent will stand still with a 0.5 probability, and cruise according to a random walk with 0.5 probability. If the agent SOC is below a certain threshold $c_{threshold}$, the agent will go to the nearest charge station and remain there until he is fully charged. The algorithm is shown in Algorithm 3.
The second algorithm is almost identical to the first one, with the exception that it is decentralized. This means that the agents cannot coordinate their actions by knowing which agent is going to pickup which customer, and thus they cannot remove the selected customer from the list. This can lead to several agents going after the same customer. For clarity, the decentralized deterministic baseline algorithm is shown in Algorithm 4.

Training scenarios

Two scenarios were used to evaluate the algorithms. The first is a small and simple scenario, with the main purpose of verifying the algorithms and the environment in a simpler setting. The second scenario is larger and is based on real-world data, to more accurately evaluate the algorithms in a realistic setting.

Small Scenario The small scenario is a size $b = \langle 5, 5 \rangle$ environment containing 5 agents and 1 charging station in the middle. In this environment, the customer

Algorithm 3 Centralized Deterministic Baseline Agent (CDBA)

Input: State s_t

Output: list of actions for all agents $\boldsymbol{u}_t$. Using global grid actions, meaning every action is the desired one-dimensional position to move to.

Extract and copy set of agents $\mathcal{Z}_c = \mathcal{Z}$ and set of charge stations $\mathcal{Q}$ from s_t
for all agents $a \in \mathcal{S}$ **do**
  if $p_t^a \in \mathcal{Q}$ **and** $c_t^a < 100$ **then**
    $u_t^a = p_t^a$ // If agent is currently charging, keep charging until full
  else if $c_t^a < c_{threshold}$ **then**
    $u_t^a = q_p$ where q is the closest charge station to agent a.
  else if $\mathcal{Z}_c \neq \emptyset$ **then**
    $u_t^a = z_p$ where z is the closest customer to agent a in the set $\mathcal{Z}_c$.
    delete z from
  else
    sample x from uniform distribution $\mathcal{U}(0, 1)$
    if $x < 0.5$ **then**
      $u_t^a = p_t^a$ // With 50% chance, stand still
    else
      sample u_t^a from the discrete uniform distribution $\mathcal{U}(0, b_r \cdot b_c)$ // With 50% chance, take a random action
    end if
  end if
end for

distribution is random over all of the grids. Every episode has $T = 288$ time steps where each step corresponds to 5 minutes in a day. The waiting time limit for the customers is 6. For all agents the SOC decay rate is $c_-^a = 2, \forall a$ and the charge rate is $c_+^a = 5, \forall a$.
When developing and implementing the MARL algorithms, this environment provided a small and familiar setting to debug and also investigate the behaviors of different algorithms. The environment can be seen in Figure 3.4.

Large scenario The large scenario is a $b = \langle 10, 10 \rangle$ grid world environment designed based on the realistic map [16] and real-time traffic of Beijing [30]. This environment includes living areas and working areas that highly similar to the real city situation of Beijing. The customer distribution includes two parts: one is a random distribution over all of the grids that simulate normal order requirements. Another is a peak hour customer distribution [19] for living area and working area in the morning and evening. By adding these two customer distributions together, the real customer distribution is simply simulated. Based on the investigation of current type of EVs, the average running distance for each time of charge is 100-200 km and it takes 2 to 3 hours to fully charge. In this scenario every episode was $T = 480$ time-steps, and thus the SOC decay rate was scaled down to $c_-^a = 1, \forall a$, meaning that a full charge will last 5 hours. The charge rate was scaled down to $c_+^a = 4, \forall a$, meaning that a full charge would take 25 time-steps, representing 75 minutes. The customer waiting time limit was set to 30, representing 90 minutes.

Algorithm 4 Decentralized Deterministic Baseline Agent (DDBA)

Input: State s_t

Output: list of actions for all agents $\boldsymbol{u}_t$. Using global grid actions, meaning every action is the desired one-dimensional position to move to.

Extract and copy set of agents $\mathcal{Z}_c = \mathcal{Z}$ and set of charge stations $\mathcal{Q}$ from s_t
for all agents $a \in \mathcal{S}$ **do**
  if $p_t^a \in \mathcal{Q}$ **and** $c_t^a < 100$ **then**
    $u_t^a = p_t^a$ // If agent is currently charging, keep charging until full
  else if $c_t^a < 100$ **then**
    $u_t^a = q_p$ where q is the closest charge station to agent a.
  else if $\mathcal{Z}_c \neq \emptyset$ **then**
    $u_t^a = z_p$ where z is the closest customer to agent a in the set $\mathcal{Z}_c$.
  else
    sample x from uniform distribution $\mathcal{U}(0, 1)$
    if $x < 0.5$ **then**
      $u_t^a = p_t^a$ // With 50% chance, stand still
    else
      sample u_t^a from the discrete uniform distribution $\mathcal{U}(0, b_r \cdot b_c)$ // With 50% chance, take a random action
    end if
  end if
end for

The purpose is to investigate the scalability and complexity of the environment that the MARL algorithms can handle. The environment can be seen in Figure 3.5.

3.3 Algorithms

In this section, the implementation of the algorithms used as well as their specific training details and modifications are explained. These are Reward Mixing Deep Double Q-Networks (RMDDQN) and Counterfactual Multi-Agent Policy Gradients (COMA).

3.3.1 Reward Mixing Deep Double Q-Networks

The Deep Double Q-Networks algorithm presented in [4] was in this thesis extended to a MARL-framework, and this extension will be referred to as Reward Mixing Deep Double Q-Networks (RMDDQN). The developed RMDDQN algorithm relies on three main ideas: *parameter sharing*, *experience replay sharing*, and *global and local reward mixing*.

To increase computational efficiency and improve scalability, parameter sharing [2] was implemented. Parameter sharing means that all agents share the same network parameters, increasing training speed significantly as all agents back-propagate on the same parameters. This is an efficient way of dealing with the scaling issue that arises in MARL. In this electric fleet management setting, all agents are homoge-

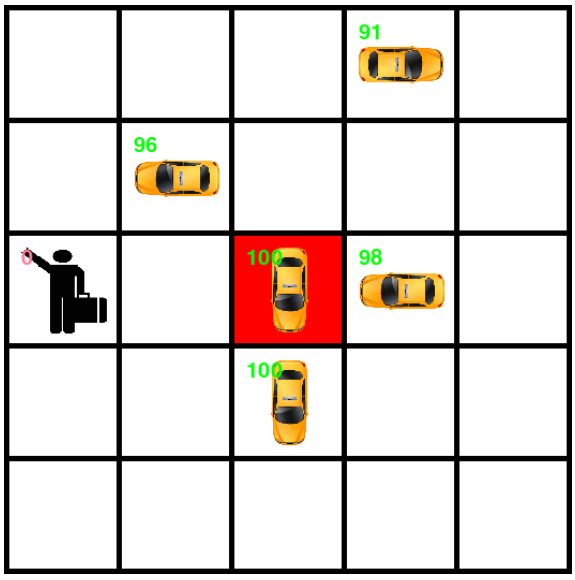


Figure 3.4: Figure showing the visualization of the small scenario. The grid world is of size $\langle 5, 5 \rangle$, and contains 5 agents.

neous, and thus parameter sharing will not problematic, as the optimal action given a specific observation ought to be the same regardless of which agent it is. One exception is that agents with a lower agent id will get to execute their action first, giving them an advantage if two agents try to pick up the same customer or enter a charge station at the same time step. Thus, the (one-hot encoded) agent a should preferably be included in the input to the network. By doing this, the network can develop hidden states for the different agents and differentiate between them. The input to the network thus appends a to the observations o_t^a.

Like the network parameters, the experience replay buffer is also shared by all the agents. For every time step, each agent interacts with the environment by taking an action and receiving a reward and an observation back. The data is then collected as transition tuples, $\tau = \langle o_t^a, u_t^1, r\beta, t^a, o_{t+1}^a, t \rangle$, and these are stored in a collective experience replay buffer from which batches will be taken for training.

As explained in Section 2.3, one major problem of MARL is the multi-agent credit assignment. By giving the global reward to the agents in the RMDDQN algorithm, the agents will have no way of reasoning about their own contribution to the reward. Lazy agents who are not contributing to the global reward will remain lazy, as they won't have an incentive to improve. In the fleet management setting of this thesis, it is very easily possible to design local rewards based on how many pickups and the consumption of each agent. By local rewards, however, no team play and individual sacrifices for the overall reward will be achieved. In other words, each agent will selfishly maximize their own reward only. As explored by other works [15], one way of assessing multi-agent credit assignment is by mixed reward signals. In other words, custom reward signals are designed which depend on both the global and the local rewards. For the RMDDQN algorithm, a mixed reward signal $r_{m,t}^a$ was

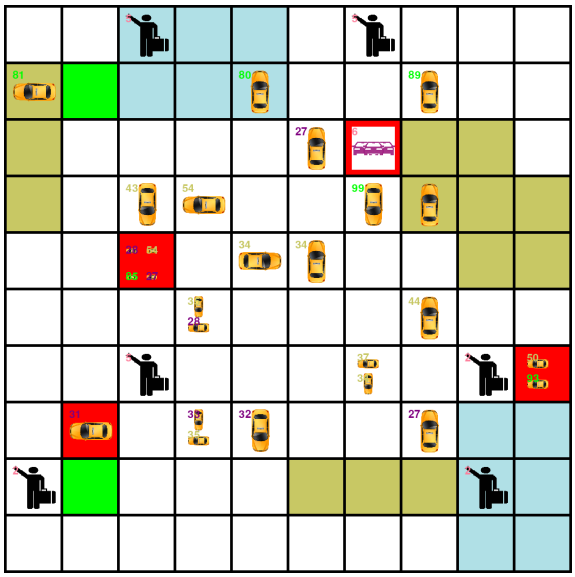


Figure 3.5: Figure showing the visualization of the big scenario. The grid world is of size $\langle 10, 10 \rangle$, and contains 32 agents.

designed according to Equation (3.11).

$$r^a_{m,t} = \beta r^a_t + (1 - \beta)\boldsymbol{r}^{-a}_t \tag{3.11}$$

Thus, by modifying β in the range $[\frac{1}{n}, 1]$, the agent reward signal can be altered from a fully global reward to a fully local reward. With $\beta = \frac{1}{n}$, the mixed reward signal will be

$$r^a_{m,t} = \frac{1}{n} r^a_t + (1 - \frac{1}{n})\boldsymbol{r}^{-a}_t = \frac{1}{n} r^a_t + (\frac{n-1}{n})\boldsymbol{r}^{-a}_t = \boldsymbol{r}_t. \tag{3.12}$$

In the same way, $\beta = 1$ will result in the local reward: $r^a_{m,t} = r^a_t$. It should be noted that due to reward mixing, RMDDQN will not be decentralized during training. But similarly to COMA, the algorithm will still be decentralized during execution.

The Q-Network The Deep Q-Network used in this thesis was an artificial neural network with a fully connected input layer, 2 fully connected hidden layers, and an output layer. The sizes of the layers, as well as their activation functions, can be seen in table 3.1. All biases are initialized to zero. The weights of the output layer are uniformly initialized from the distribution $U(10^{-6}, 10^{-5})$. The RMSprop algorithm [7] was used as an optimizer, as it has proven to be useful in DQN from other works [18].

Training The RMDDQN algorithms interaction with the environment is visualized in Figure 3.6 for a two-agent scenario and the main training loop can be seen in Algorithm 5. The exploration rate was controlled by a decaying ϵ-greedy algorithm according to $\langle \epsilon_{max}, \epsilon_{decay}, \epsilon_{min} \rangle$. This means that the exploration rate starts at ϵ_{max}, and linearly decreases to ϵ_{min} with a rate of ϵ_{decay} per episode.

Evaluation To evaluate the RMDDQN algorithm, a modified version of the training algorithm was used. Here, the exploration rate ϵ was naturally constantly

Layer	size	activation
Input layer 1	$\|\mathcal{O}\|$	ReLU
Hidden layer 2	64	ReLU
Hidden layer 3	64	ReLU
Output layer	$\|\mathcal{U}\|$	None

Table 3.1: The architecture of the deep Q-Network used in the RMDDQN algorithm. The size of the first layer is the size of the observation. The output layers size will naturally always be the size of the action space $|\mathcal{U}|$ as the network should output one Q-value for each action at the current state.

Algorithm 5 Reward Mixing Deep Double Q-Network Training

Initialize $\theta^{online}, \theta^{offline}$
$\epsilon = \epsilon_{max}$
for every episode e **do**
  Reset environment and observe $s_t, \boldsymbol{o}_t, t = 0, t_b = \textbf{false}$
  while $t_b \neq \textbf{true}$ **do**
    $t = t + 1$
    for all agents a **do**
      Calculate Q-values for o_t^a using the offline model $\theta^{offline}$
      Sample action u_t^a from an epsilon greedy policy using the Q-values and ϵ
    end for
    Take step in environment using $\boldsymbol{u}_t$.
    Get state s_{t+1}, observations $\boldsymbol{u}_{t+1}$, rewards $\boldsymbol{r}_t$ and terminal state boolean t_b
    for all agents a **do**
      $r_{c,t}^a = \beta r_t^a + (1-\beta)\boldsymbol{r}_t^{-a}$
      Add transition $\tau = \langle o_t^a, u_t^a, r_{c,t}^a, o_{t+1}^a, t_b \rangle$ to experience replay buffer
    end for
    if length of experience replay buffer > 1000 **then**
      // If buffer contains more than 1000 samples, perform a training step
      sample batch $\boldsymbol{o}, \boldsymbol{u}, \boldsymbol{r}, \boldsymbol{o'}, \boldsymbol{t}_b$ from experience replay buffer
      if $t_b = \textbf{false}$ **then**
        $\boldsymbol{Y}^{DoubleQ} = \boldsymbol{r} + \gamma Q(\boldsymbol{o'}, \arg\max_a Q(\boldsymbol{o'}, \boldsymbol{u}; \theta^{online}); \theta^{offline})$
      else
        $\boldsymbol{Y}^{DoubleQ} = \boldsymbol{r}$
      end if
      $\boldsymbol{\Delta Q} = \boldsymbol{Y}^{DoubleQ} - \boldsymbol{Q}(\boldsymbol{o}, \boldsymbol{u}; \theta^{online})$
      $\Delta\theta^{online} = \nabla_{\theta^{online}}(\boldsymbol{\Delta Q})^2$ // Calculate the gradient
      $\theta^{online} = \theta^{online} + \alpha\Delta\theta^{online}$ // Update network weights
      Every x training steps update the offline target weights; $\theta^{offline} = \theta^{online}$
    end if
  end while
  $\epsilon = \max(\epsilon - \epsilon_{decay}, \epsilon_{min})$
end for

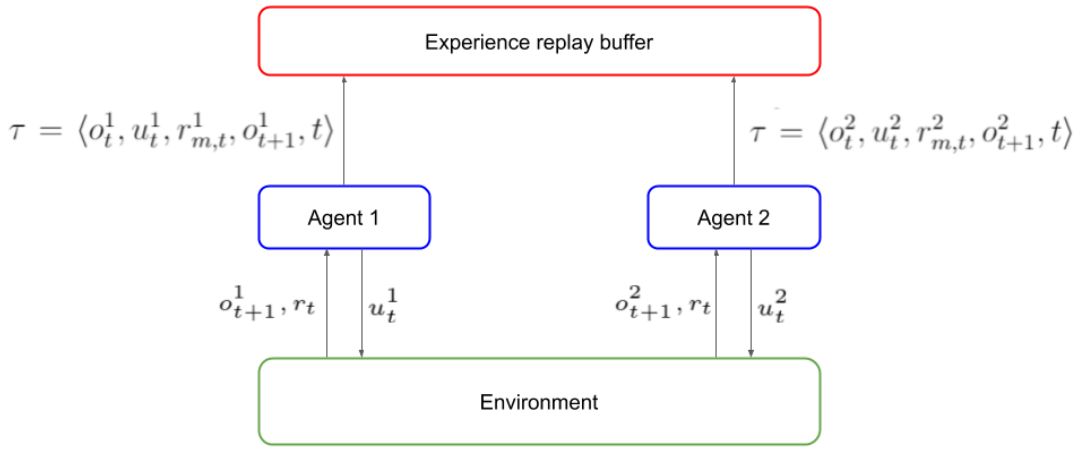


Figure 3.6: The interaction between the RMDDQN algorithm and the environment. Agents interact with the environment, generating data in the form of transitions $\tau = \langle o_t^a, u_t^a, r_{m,t}^a, o_{t+1}^a, t \rangle$. These are then sent and stored in the Experience replay buffer, from which the algorithm randomly samples batches of transitions to train the network on.

zero. As no training is being done during the evaluation, no experience buffer was used. For clarity, the evaluation algorithm can be seen in Algorithm 6.

Algorithm 6 Reward Mixing Deep Double Q-Network Evaluation

```
Load the trained network parameters θ
for every evaluation episode e do
    Reset environment and observe s_t, o_t, t = 0, t_b = false
    while t_b ≠ true do
        t = t + 1
        for all agents a do
            Calculate Q-values for o_t^a using the network with parameters θ
            Select action according to a greedy policy.
        end for
        Take step in environment using u_t.
        Get state s_{t+1}, observations u_{t+1}, rewards r_t and terminal state boolean t_b
    end while
end for
```

3.3.2 Counterfactual Multi-Agent Policy Gradients

The implementation of COMA mostly followed the original implementation in the paper [3]. Just as in the paper, the critic will be trained using $TD(\lambda)$ as presented in Section 2.2. An overview of COMA as well as the interaction between the actors, the

critic, and the environment can be seen in figure 3.7. Like for RMDDQN, parameter sharing was used.

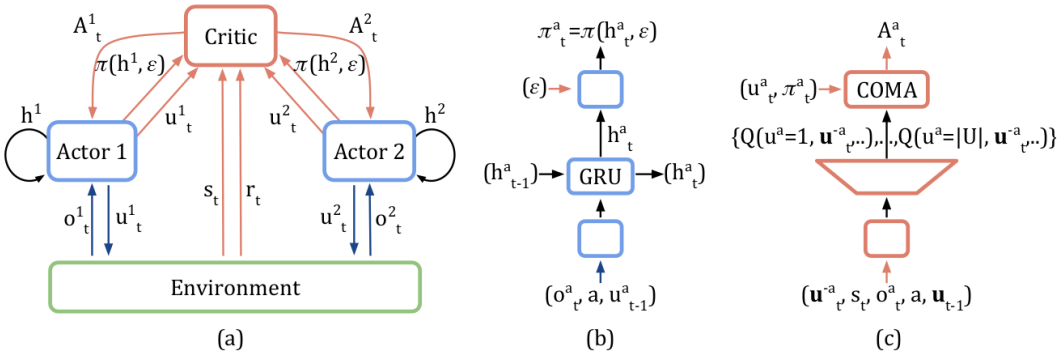


Figure 3.7: An overview of the COMA networks. In figure (a) the complete system and the interaction with the environment can be seen. In figure (b) the actor is shown, and in figure (c) the COMA critic is shown. Figure taken from [3] with permissions.

The Actor The actor consists of a gated recurrent unit of size 128, with a fully connected layer before it to process the input, and another after it to produce the output. See table 3.2 for details.

Layer	size	activation
Input layer	$\|\mathcal{O}\|$	ReLU
Gated recurrent unit	128	None
Output layer	$\|\mathcal{U}\|$	Softmax

Table 3.2: The architecture of the COMA Actor network. The network consists of a fully connected input layer to process the input, a gated recurrent unit, and an output layer to produce the output. The output is parametrized with soft-max in action preferences (see Section 2.2.2)

To increase exploration, a bounded softmax distribution was used to transform the final output, $\boldsymbol{z}$, from the network and achieve the action probabilities $P(u)$ according to Equation (3.13).

$$P(u) = (1 - \epsilon)\boldsymbol{z} + \epsilon/|\mathcal{U}| \tag{3.13}$$

This provided a lower-bound for each action, ensuring they were tried and explored. The ϵ factor was decreased linearly just as it is used in the decaying ϵ-greedy algorithm.

The Critic The critic network is a feed-forward network with 3 layers. The specifications can be found in Table 3.3. It can be seen that this network is very similar to the Q-network of the RMDDQN algorithm. Indeed, the networks have similar tasks, as both are supposed to estimate Q-values.

Layer	size	activation
Input layer	$\|\mathcal{O}\|$	ReLU
Hidden layer 1	64	ReLU
Output layer	$\|\mathcal{U}\|$	None

Table 3.3: The architecture of the COMA Critic network. The network consists of three fully connected layers, producing the Q-values for each action in the action space.

Training Here the main training loop of the COMA algorithm is presented. Note that the critic has an online network with parameters θ^{online} and an offline network with parameters $\theta^{offline}$, just as in RMDDQN. The actor-network parameters will be referred to as θ^{π}. The COMA training consists of three main parts: data collection, critic training, and actor training. The data collection part consists of gathering *batch_size* episodes and storing them in a buffer. These will all be trained in parallel in the next sections. The second step is training the critic, which consists of taking a gradient step for each time step, with all episodes in parallel. In the last part, the actor is trained. This is done by calculating the COMA advantage, accumulating the actor gradients, and then updating the network by taking a gradient step. Refer to Algorithm 7 for the full Algorithm.

Evaluation The evaluation algorithm for COMA is presented in Algorithm 8. The main difference here is that the critic is not being used at all. The centralized critic is only used for evaluating the current policy during training. During evaluation time, the COMA algorithm is decentralized, using only the actors to select actions. Another difference to the training script here is that no bounded softmax distribution is used. Instead, the action is directly sampled from the policy. The actor and critic training sections are also removed, as no training is being done during evaluation mode.

3.4 Summary

In this chapter, the methodology of the thesis was presented. Firstly, the development setting which covered the tools and programming languages used was explained. After that, the developed grid world environment was formally presented in detail. The chapter then ended with an explanation of the implementation of the two algorithms used in the thesis: RMDDQN and COMA.

Algorithm 7 Counterfactual Multi-Agent Policy Gradients Training. Small modifications from the original in [3]. Used with permissions.

Initialize $\theta^{online}, \theta^{offline}$
$\epsilon = \epsilon_{max}$
for every training episode e **do**
  for 1 to batch size **do**
    Reset environment and observe $s_t, \boldsymbol{o}_t, t = 0, t_b =$ **false**
    while $t_b \neq$ **true do**
      $t = t + 1$
      for all agents a **do**
        Get policy π from actor θ^{π} using observation o_t^a
        get action probabilities by taking the bounded softmax distribution on the policy π with current ϵ
        Sample action u_t^a from action probabilities
      end for
      Take step in environment using $\boldsymbol{u}_t$.
      Get state s_{t+1}, observations $\boldsymbol{u}_{t+1}$, rewards $\boldsymbol{r}_t$ and terminal state boolean t_b and save in buffer
    end while
  end for
  for all agents a **do**
    // For every agent, train all episodes in buffer in parallel
    for $t = 1$ to $t = T$ **do**
      Using the critic offline network $\theta^{offline}$, Calculate $TD(\lambda)$ targets y_t^a
    end for
    for $t = T$ to $t = 1$ **do**
      // Update the critic
      $\Delta Q_t^a = y_t^a - Q(s_j^a, \boldsymbol{u})$
      $\Delta\theta^{online} = \nabla_{\theta}^{online}(\Delta Q_t^a)^2$ // The critic gradient
      $\theta^{online} = \theta^{online} - \alpha\Delta\theta^{online}$
      Every x training steps update the offline target weights; $\theta^{offline} = \theta^{online}$
    end for
    for $t = T$ to $t = 1$ **do**
      $A^a(s_t^a, \boldsymbol{u}) = Q(s_t^a, \boldsymbol{u}) - \sum_u Q(s_t^a, u, \boldsymbol{u}^{-a}\pi(u|h_t^a)$ // COMA advantage
      $\Delta\theta^{\pi} = \Delta\theta^{\pi} + \nabla_{\theta_\pi} \log \pi(u|h_t^a) A^a(s_t^a, \boldsymbol{u})$
    end for
    $\theta^{\pi} = \theta^{\pi} + \alpha\Delta\theta^{\pi}$
  end for
  $\epsilon = \max(\epsilon - \epsilon_{decay}, \epsilon_{min})$
end for

Algorithm 8 Counterfactual Multi-Agent Policy Gradients Training. Small modifications from the original in [3]. Used with permissions.

Load actor network parameters θ^{π}
for every evaluation episode e **do**
  Reset environment and observe $s_t, \boldsymbol{o}_t, t = 0, t_b = \textbf{false}$
  while $t_b \neq \textbf{true}$ **do**
    $t = t + 1$
    for all agents a **do**
      Get policy π from actor θ^{π} using observation o_t^a
      Sample action u_t^a from policy
    end for
    Take step in environment using $\boldsymbol{u}_t$.
    Get state s_{t+1}, observations $\boldsymbol{u}_{t+1}$, rewards $\boldsymbol{r}_t$ and terminal state boolean t_b
  end while
end for

4

Results

In this chapter, the experiments done are presented, and the results achieved are explained. The chapter is divided into two main parts: experiments on the small scenario and experiments on the big scenario. In the small scenario, the first experiment is to test the global and local reward mixing algorithm in RMDDQN. After that, the different action modes are explored and experimented. The small scenario section then ends with experiments where the most successful network parameters of the algorithms are evaluated against the baselines presented in section 3.2. The second section begins with training experiments of the algorithms on the large scenario. It then ends with comparisons of the most successful parameters in evaluation mode against the baselines.

Each training- and evaluation experiment will be presented by a set of four graphs. These are graphs showing the revenue (number of customers picked up), consumption (number of blocks traveled), efficiency (revenue/consumption), and the environment reward signal. The reward shown is always the global reward, even though individual agents might see different rewards due to global and local reward mixing. All values are normalized by the total number of agents. Smoothing has also been applied to easier see trends. The raw signals can be seen in the same color as the original graph, but with lower opacity. The x-axis will represent the number of environment episodes.

4.1 Small scenario

This section covers the experiments made on the small scenario. Most of the experiments here were explorational, meaning that they were used to analyze the dynamics of the environment and different parts of the algorithms. These results found here were then used to easier and faster train the algorithms on the larger and more complex scenario. The Table in 4.1 contains the general parameters used in these experiments. In each experiment, deviating parameters will be mentioned explicitly. As it can be seen in the table, the reward signal consisted of a drive reward and a pickup reward. The drive reward is a (negative) reward that the agent gets every time it moves a block in the grid world. The pickup reward is a reward an agent gets every time it reaches a customer with sufficient SOC and thus makes a pickup.

RMDDQN Hyperparameters	
reward mixing factor β	0.8
learning rate α	0.0001
discount rate γ	0.99
batch size	32
$\langle \epsilon_{max}, \epsilon_{decay}, \epsilon_{min} \rangle$	$\langle 0.9, 0.001, 0.1 \rangle$
target network update frequency τ	150
Action mode $m(a, p, u)$	2
COMA Hyperparameters	
Actor learning rate α_π	0.000075
Critic learning rate α_c	0.000075
TD learning λ	0.7
discount rate γ	0.99
batch size	16
$\langle \epsilon_{max}, \epsilon_{decay}, \epsilon_{min} \rangle$	$\langle 0.5, 0.00064, 0.02 \rangle$
target network update frequency τ	150
Action mode $m(a, p, u)$	2
Reward Signals	
Drive reward	-1
Pickup reward	10
Environment Variables	
Customer buffer n_z	4
Agent buffer n_a	4
Charge buffer n_q	1

Table 4.1: Table showing the general parameters used in the small scenario experiments. Deviations from these will be mentioned in each specific experiment

Global and Local Reward Mixing

The first experiment was to analyze the global and local reward mixing algorithm used in RMDDQN. Five different values of β were tested, ranging from 0.2 (fully global reward) to 1 (fully local reward). The intermediate rewards were thus compared with these extremes.

In Figure 4.1 the results of the experiment can be seen. It can be seen that a purely global reward ($\beta = 0.2$) is the worst-performing. This makes sense intuitively as the algorithm will have no way of assessing the multi-agent credit assignment problem. By increasing the fraction of the local reward β, the total global reward of the agents also increases. The two values of β performing the best are 0.8 and 1. In terms of revenue, they seem to both be picking up roughly the same amount of customers. However, looking at consumption it can be seen that $\beta = 0.8$ marginally lowers the consumption, resulting in better efficiency and reward as well. It could be that by adding a part of the global reward as in the case with $\beta = 0.8$, the agents are more inclined to lower the total consumption as it affects all agents more.

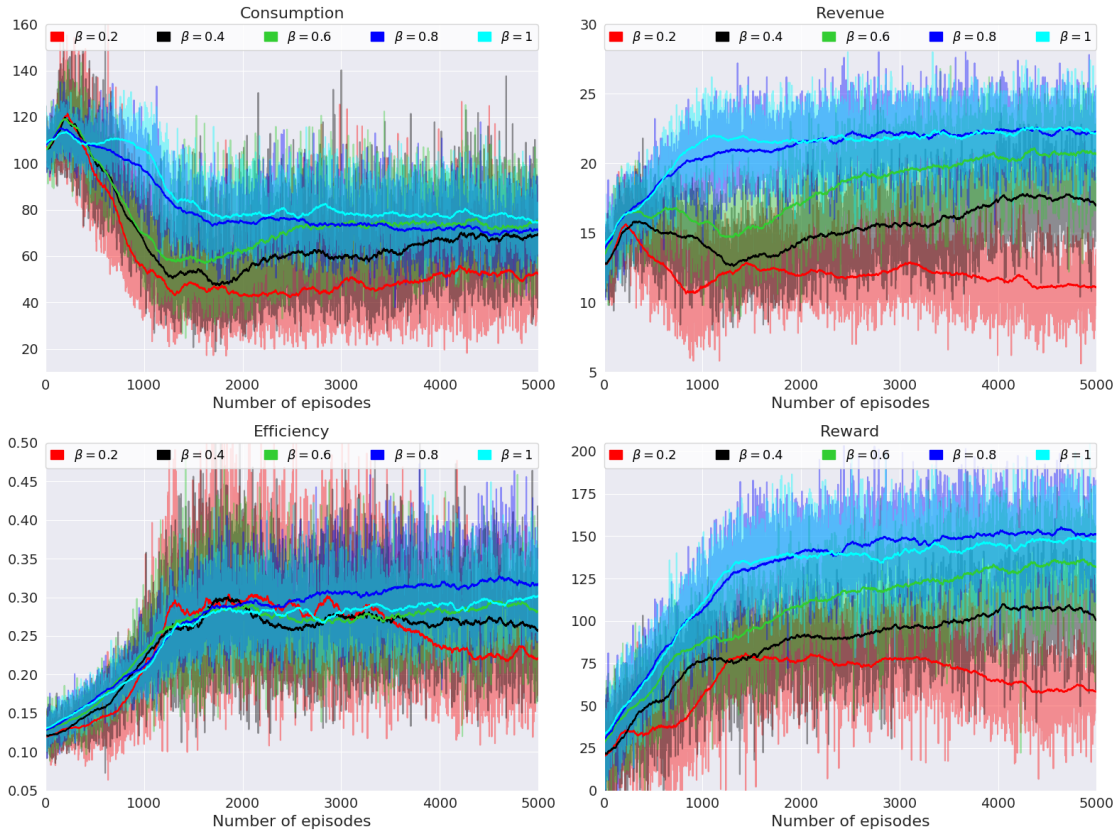


Figure 4.1: Experiment testing different values of β for the global and local reward mixing algorithm. In the top left, the consumption is shown, and in the top right, the revenue is shown. in the bottom left the fraction between the revenue and consumption is shown, and finally, in the bottom right, the global reward can be seen. From the graph, it can be seen that higher values of β (higher fraction of local reward) are superior. It can also be seen that $\beta = 0.8$ is marginally better than the fully local reward achieved with $\beta = 1$, by being more energy efficient and thus having a slightly higher reward as well.

Action Modes

The experiment of this section explored how the two algorithms performed with each action mode enabled in the small scenario. The results from these experiments would facilitate the choice of action mode for training the algorithms on the big scenario. The results for the RMDDQN action mode experiment can be seen in figure 4.2. From the graphs, it can be seen that in terms of total revenue and efficiency, action modes 1 and 3 are the best performing. Action mode 2 is more energy-saving but marginally less efficient. Action mode 0 is simply worse in all aspects. Due to action mode 3 being a combination of action modes 1 and 2, it was concluded that action mode 1 was superior since the increased complexity in action mode 3 didn't provide any benefits. As such, action mode 1 became the favorable action mode of the RMDDQN algorithm.

In Figure 4.3, the results of the action mode experiment can be seen for the COMA

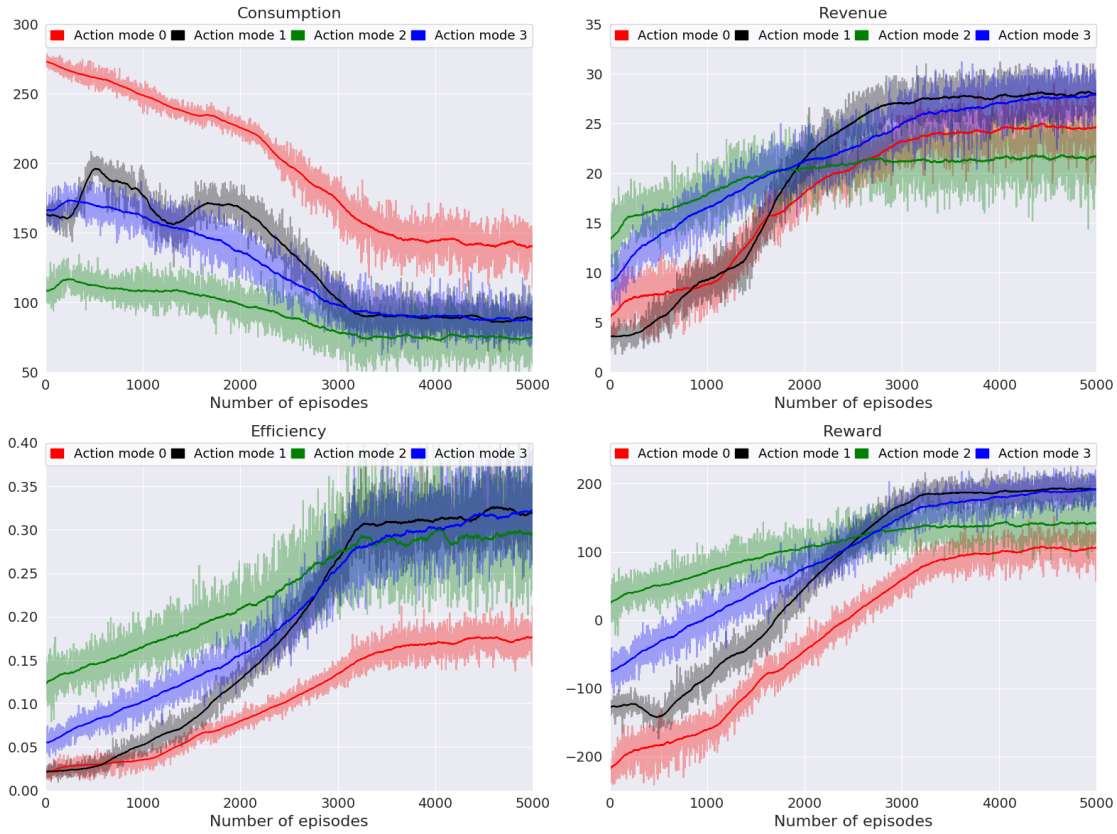


Figure 4.2: Experiment on testing the different action modes for the RMDDQN algorithm on the small scenario. From the graphs, it can be seen that action mode 1 and 3 were the best performing in terms of total revenue and efficiency. Not far below is action mode 2, with slightly lower revenue, and thus also lower efficiency. The worst performing was action mode 0. Its efficiency was distinctively lower than the rest due to the much higher consumption. Since action mode 3 was an extension of action mode 1 and didn't provide any value for the extra complexity, action mode 1 was selected as the most favorable action mode.

algorithm. As with the RMDDQN algorithm, it can quickly be concluded that action mode 0 is not performing well. The most energy-efficient algorithm would be action mode 1, but its total revenue, as well as its total consumption, is very low. Here the algorithm has chosen a very passive policy. By simulating the algorithm during evaluation mode (no exploration ϵ), it can be seen that the agents do not move at all, and thus only pick up customers that spawn on them. This strategy essentially provides infinite efficiency, as agents do not move at all. The total revenue suffers greatly, however. The agents do not charge at all and do not pick up any customers no matter the distance to them unless they are on the same block. Thus action mode 1 was discarded. Action modes 2 and 3 were quite similar in performance but with action mode 2 having a higher overall revenue at a smaller complexity. Thus, for COMA, action mode 2 was selected as the favorable action mode.

Evaluation

In this section, the evaluations of the best parameters of each MARL algorithm are compared with the two baselines. For RMDDQN, the parameters from the best run of the action mode experiment were used. This was the run using action mode 1. For COMA, the parameters were also taken from the best action mode experiment run, which in this case were the parameters for the run with action mode 2. In Figure 4.4 the results can be seen when running the algorithms with these parameters in evaluation mode. The MARL algorithms can be seen compared to the two baselines, the decentralized deterministic baseline agent (DDBA), and the centralized deterministic baseline agent (CDBA). Looking firstly at the consumption, it is clear that the MARL algorithms developed strategies to lower their consumption. By looking at revenue, however, it can be seen that COMA lowered the consumption at the cost of drastically lower revenue. However, RMDDQN manages to achieve a reward higher than both the baselines, while having a consumption almost as low as COMA. Overall this resulted in RMDDQN having the by far highest efficiency, followed by COMA, and after that the baselines. In the reward graph, it can be seen that the RMDDQN algorithm is most successful in maximizing its accumulative reward. It should be noted that only the MARL algorithms want to maximize the reward. Thus it would be counter-intuitive to comment on the baselines behavior on this.

4.2 Big scenario

In this section, the results of the big scenario will be presented. Firstly, the training results of each algorithm will be presented separately, and then the evaluation of the trained parameters will be compared with the baselines. In Table 4.2, the general parameters used in this section are shown. The new reward, *stand still reward*, is a reward that the agent receives multiplied with the number of time steps it has been standing still (unless he is charging). So if an agent has been standing still for 0 time-steps, no reward will be given, but the longer he stays still, the higher the (negative) reward will be. This is to counteract passive policies that emerge in larger scenarios where rewards are so sparse. Instead of getting the pickup reward by moving a couple of blocks, agents have to cover more distance, and thus plan many more time steps in the future, to receive the pickup rewards. If the agent needs to charge, the reward will be delayed even further into the future. It has been observed that in these cases the MARL algorithms usually find a locally optimal policy of standing still very quickly, since this effectively removes all negative rewards from moving, and positive rewards from pickup still come when customers spawn on them. This way the agents do not have to concern themselves with charging since this passive policy usually makes a full charge last a complete episode.

Training

In Figure 4.5 the results for the RMDDQN algorithm in the big scenario can be seen. The Figure shows the results for three different values of β, as it heavily in-

RMDDQN Hyperparameters	
reward mixing factor β	0.9375
learning rate α	0.0001
discount rate γ	0.99
batch size	32
$\langle \epsilon_{max}, \epsilon_{decay}, \epsilon_{min} \rangle$	$\langle 0.9, 0.001, 0.1 \rangle$
target network update frequency τ	150
Action mode $m(a, p, u)$	1
COMA Hyperparameters	
Actor learning rate α_π	0.0000075
Critic learning rate α_c	0.0000075
TD learning λ	0.7
discount rate γ	0.99
batch size	4
$\langle \epsilon_{max}, \epsilon_{decay}, \epsilon_{min} \rangle$	$\langle 0.5, 0.00064, 0.02 \rangle$
target network update frequency τ	150
Action mode $m(a, p, u)$	2
Reward Signals	
Drive reward	-2
Pickup reward	100
Stand-still reward	-1
Environment Variables	
Customer buffer n_z	3
Agent buffer n_a	3
Charge buffer n_q	3

Table 4.2: Table showing the general parameters used in the big scenario experiments. Deviations from these will be presented in each specific experiment.

fluenced the performance of the algorithm. As noticed in the experiments on the small scenario, using the global reward ($\beta = 0.03125$) leads to more passive and low consumption policies. This can be seen in the consumption graph, with $\beta = 0.03125$ having the by far lowest consumption. However, this results in the algorithm suffering in revenue, as the multi-agent credit assignment problem becomes prominent. Looking at the algorithm using the local rewards ($\beta = 1$) instead, it can be seen that the algorithm diverges, as clearly seen by the reward. Since every agent only views its own local reward, which in this big scenario is a very small fraction of the true global reward, this reward signal does a very poor job of representing how well the joint action $\boldsymbol{u}$ of all the agents were. It might also indicate that in this big scenario more cooperation is needed, which is does not emerge with a local reward where all agents will develop selfish strategies to optimize their own policy only.

By setting $\beta = 0.9375$ (same reward mixing fraction as $\beta = 0.8$ in the small scenario), a significant increase in performance can be seen. The algorithm doesn't decrease the consumption dramatically like $\beta = 0.03125$, but it still manages to decrease it while increasing the revenue drastically. This results in the best efficiency out the

three values of β, being almost twice as efficient as $\beta = 0.03125$ and four times as efficient as $\beta = 1$.
In Figure 4.6, the result from the COMA algorithm trained on the big scenario can be seen. Here three different values for λ in the $TD(\lambda)$ part of the algorithm are presented. These different runs are very similar and show that COMA is quite stable and converges to the same policy given different values of λ. With $\lambda = 0.7$, the convergence is slightly faster though. Looking at consumption, it can be seen, similarly to in the small scenario, that COMA favors very passive and low consumption policies. This in turn generates a very low revenue as seen in the graphs. The overall efficiency thus increases and gives the false impression of a potentially efficient policy. Looking at the reward, it can be seen why the algorithm converges to this solution. By converging to the locally optimal policy of almost completely reducing consumption, the total reward increases from around $-10,000$ to 0. As the goal of the agent is to maximize the accumulative reward over the episode, this policy is very good from the perspective of the agent. Generating a more efficient policy would marginally increase the reward, and since this locally optimal policy is so easy to find, the agent is not exploring towards the globally optimal policy. A known weakness of policy gradient methods is that they typically converge to local optimums instead of the global optimum [27].

Evaluation

In Figure 4.7 the results from the evaluation of the best MARL parameters are shown together with the baselines in the big scenario. The RMDDQN parameters were naturally taken from the training with $\beta = 0.9375$, and the COMA algorithm parameters were taken from the run with $\lambda = 0.7$. By firstly looking at the reward it can be seen that the COMA algorithm reaches a local optimum not far from the RMDDQN policy. Looking at the performance, a vast difference can be seen. The consumption graph clearly visualizes how aggressively COMA limits consumption. In the revenue graph, the results of this can be seen. RMDDQN and both the baselines are very similar in revenue, while the revenue of the COMA algorithm is far lower. This results in COMA having the lowest efficiency, followed by the decentralized deterministic baseline agent (DDBA) and after that the centralized deterministic baseline agent (CDBA). The most efficient algorithm, just as in the small scenario, is RMDDQN. An interesting observation is that the efficiency of COMA during evaluation is lower than during training with forced exploration. This is because the exploration resulted in COMA picking up more customers, and now that the algorithm has no exploration, the policy becomes even more passive.
By simulating the COMA algorithm, the policy can be observed. The strategy of the COMA algorithm was to immediately rush to charge stations and remain there. This is because to reach the local optimum, the agents want to quickly decrease the consumption. Standing still outside of charge stations was however not good due to the stand-still reward. Thus, the agents learned to quickly get to charge stations and stay there. Sometimes the agents would move outside if customers spawned just outside the charge stations.
The RMDDQN algorithm showed a much more desirable policy however. The agents

learned to both charge and pickup customers. As seen from the revenue, the algorithm maintained a revenue as high as the deterministic baselines, who would naturally try to pick up every single customer that spawned. The algorithm managed to do this while lowering energy consumption and thus produced the most best policy.

Summary of results

In Table 4.3 the results are summarized to give an overview of how the algorithm performed in the electric taxi fleet management energy optimization problem. All values are the means taken from the data of the evaluations of the big and small scenario respectively. The best metric in each category is marked in bold. In the table it can be that the overall best performing algorithm was RMDDQN, being the best in terms of both revenue and efficiency. In the small scenario COMA also outperforms the baseline agents in terms of efficiency, but even here it can be seen that the revenue generated by COMA is very low. In the large scenario, this is even clearer, with the mean revenue of COMA being only $\sim 7\%$ out of the other algorithms mean revenues.

	Small Scenario			Big scenario		
Algorithm	Consumption	Revenue	Efficiency	Consumption	Revenue	Efficiency
RMDDQN	79	**30**	**0.38**	128	**14**	**0.11**
COMA	**70**	18	0.27	**13**	1	0.05
DDBA	210	26	0.12	205	**14**	0.07
CDBA	162	29	0.18	164	**14**	0.09

Table 4.3: Table summarizing the results of the evaluations of the algorithms. All values are the means over the evaluation. The best value for each metric is marked in bold.

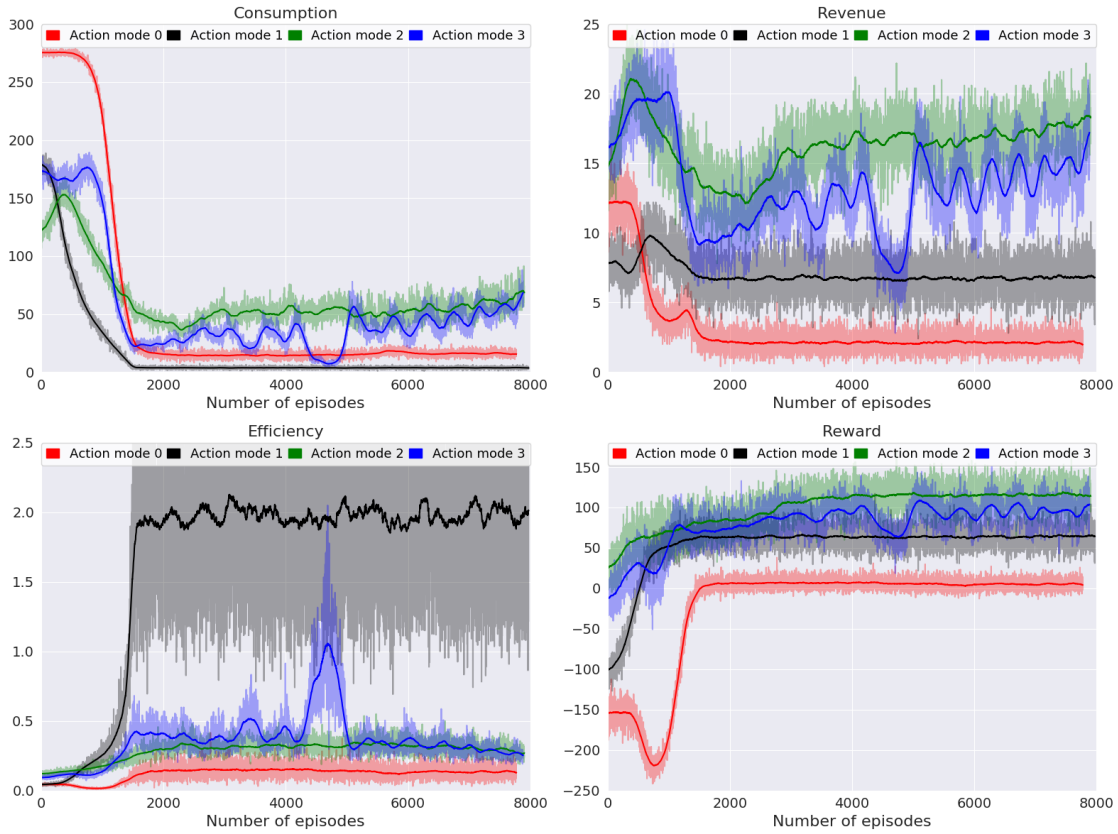


Figure 4.3: Experiment on testing the different action modes for the COMA algorithm on the small scenario. From the consumption graph, it can be seen that action mode 0 and 1 aggressively limit consumption, creating very passive policies. Action mode 2 and 3 also lowers the consumption remarkably, but still maintains some movement. Looking at the revenue, it can be seen that action mode 2 has the highest reward followed by action mode 3, 1, and 0. In the efficiency graph, it can be seen that action mode 2 has very high efficiency. This is however because the agents barely move during that policy, generating a very high efficiency but at the cost of very low revenue. Looking at the reward graph, it can be seen that action mode 2 is the most successful at maximizing the global reward.

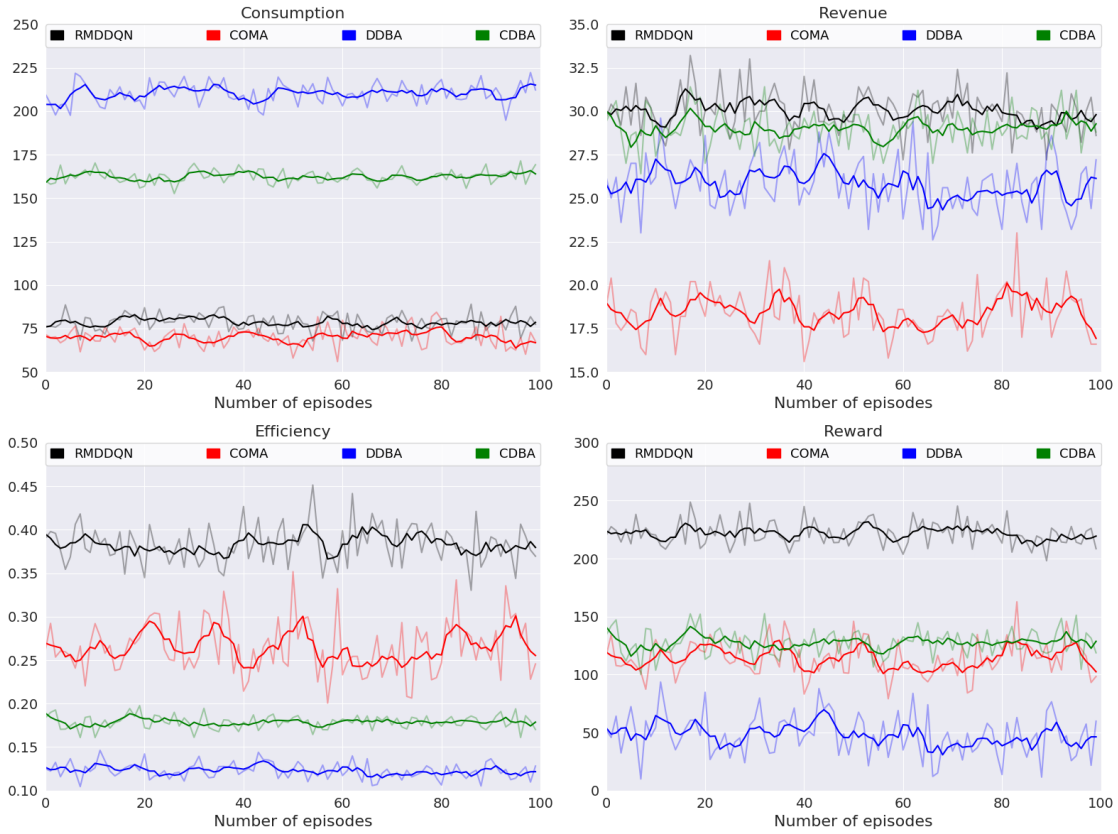


Figure 4.4: Figure showing the evaluation of the best parameters trained for RMDDQN and COMA in the small scenario, compared with the decentralized deterministic baseline agent (DDBA) and the centralized deterministic baseline agent (CDBA). It can be seen that the MARL algorithms both successfully managed to lower consumption. RMDDQN did however also manage to hold the highest revenue, while the COMA algorithms revenue was quite low. In terms of efficiency, both algorithms beat the baselines, with RMDDQN being the most efficient.

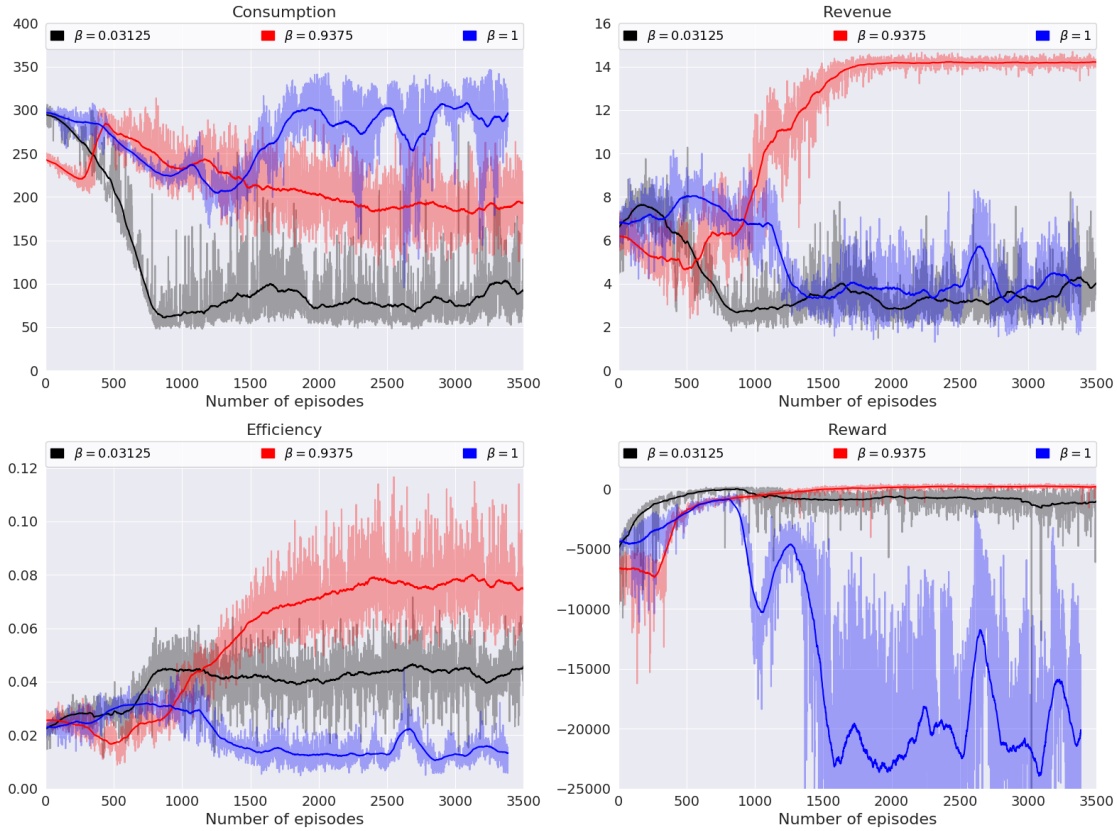


Figure 4.5: Figure showing RMDDQN trained for three different values of β in the big scenario. From the graphs it clear that $\beta = 0.9375$ is the best performing, followed by $\beta = 0.03125$ (global reward). Looking at the reward for $\beta = 1$, it can be seen that the algorithm fails to converge.

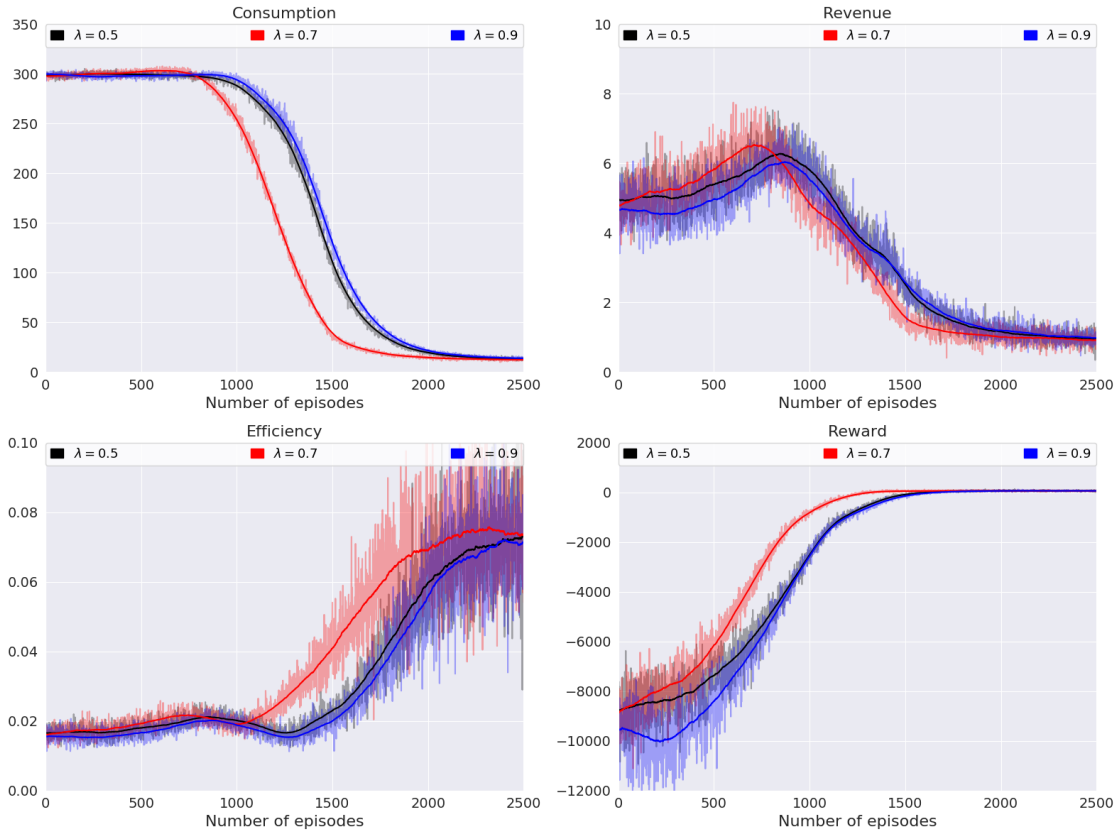


Figure 4.6: Figure showing the training of COMA with three different values of λ for the big scenario. All three runs perform very similarly, with $\lambda = 0.7$ being slightly faster. It can be seen from the reward that they converge successfully. From the consumption plot, it can be seen that they develop a passive policy with very little movement. In the revenue plot, it can be seen that this comes at the expense of much lower revenue. Due to the extremely low consumption, the efficiency still increases.

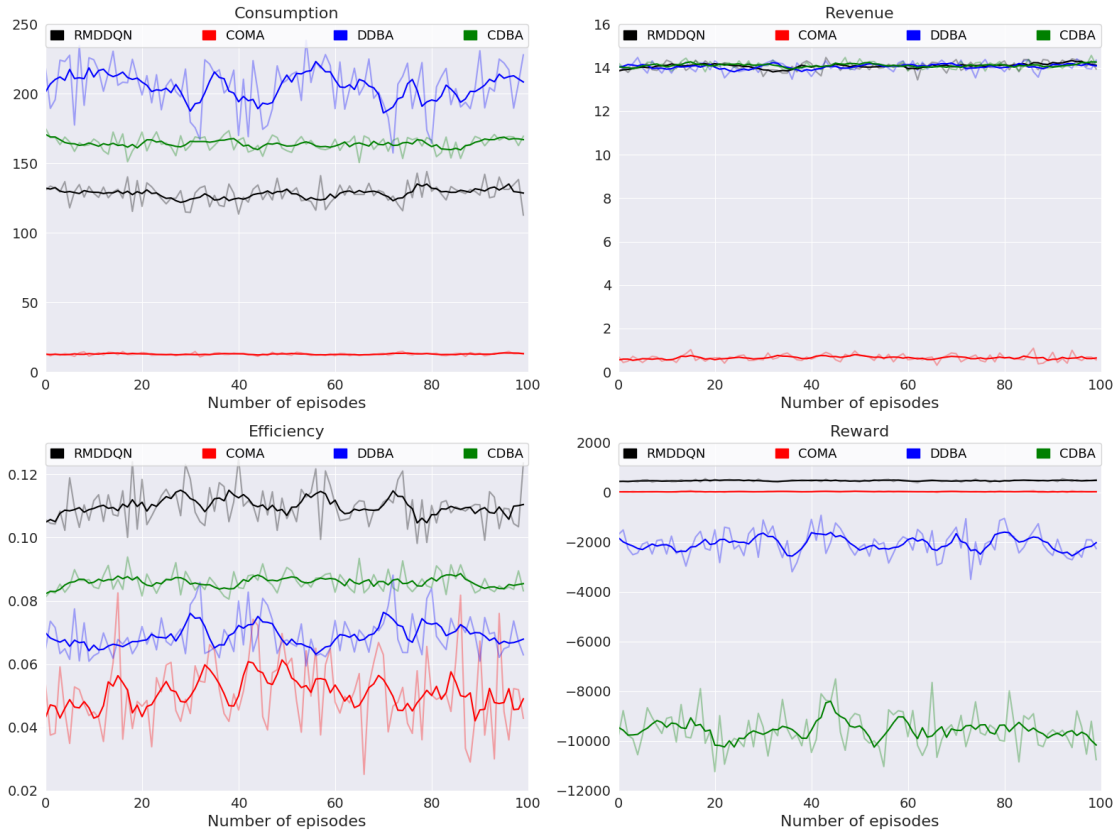


Figure 4.7: Figure showing the evaluation of the best parameters trained for RMDDQN and COMA in the big scenario, compared with the decentralized deterministic baseline agent(DDBA) and the centralized deterministic baseline agent (CDBA).

5
Conclusion

In this thesis, a new modular environment for the electric taxi fleet management problem was developed. This modular environment can be scaled arbitrarily, and can easily be extended with more advanced dynamics to make it more similar to the real world. This thesis also extended the DDQN algorithm to the multi-agent setting. This was done by the RMDDQN algorithm, which used reward mixing to solve multi-agent credit assignment, and parameter sharing to tackle the scaling problem. The popular MARL algorithm COMA was also implemented.
The performance of the algorithm was evaluated in two scenarios, a small scenario, and a big scenario. They were also compared against two deterministic baseline algorithms, a decentralized and a centralized one. In table 4.3 the final results are shown and it can be seen that for the two outlined scenarios, the MARL-algorithms beat the baselines. In the small scenario, both algorithms learned a policy that included both charging and picking up customers. In the big scenario, only RMDDQN did. It also required some extra tuning, and the introduction of the new stand-still reward, to achieve this behavior for RMDDQN. A reason to why RMDDQN outperformed COMA is most likely due to two reasons. The first is that RMDDQN uses individual rewards in the reward mixing algorithm, and it might be that this is more powerful than the counterfactual baseline of COMA for multi-agent credit assignment. In some problems, it might not be this easy to create individual rewards, but in the electric taxi fleet problem, it is straightforward as you know how much each agent has moved and how many customers each agent has picked up. The second reason is that since the optimization problem is multi-dimensional, as the consumption needs to be minimized at the same time as the revenue needs to be maximized, many local optimums might emerge. As mentioned before, policy gradient methods are known to sometimes only converge to these local optimums. Due to the nature of this problem, maybe value-based methods might be more suitable as the results achieved indicate.
Another interesting thing is that the decentralized RMDDQN algorithm beat the centralized deterministic baseline agent in both scenarios. However, in most taxi fleet management problems centralization is not a problem as you might have some call center that can dispatch the vehicles in a centralized fashion. For very big taxi fleets, centralized policies might be too computationally expensive, and thus decentralization might be beneficial. For the future, decentralization might help the agents predict and compete with agents from other companies. As agents learn a decentralized policy, they might be able to predict the behavior of taxis from other companies and use that to choose smarter actions. The MARL agent also used local observations in the big scenario, whereas the deterministic baselines had access to

the full state of the environment.
Overall these results are promising, but there is still a lot more research needed. In the following section, potential directions for future work are given.

5.1 Future work

Electric taxi fleet management is a complex problem integrating multiple sub-problems such as order matching, cruising, and charging. In this thesis, the goal was to train MARL algorithms to learn policies to handle all of these tasks simultaneously. Although the results were promising in the simulated environment, more research is needed before this can be applied in real-world cases.
First off, the simulation environment needs to be developed further to become more realistic. One major change is to include the customer travels. So when an agent picks up a customer, he will be "offline" while he travels to the customer destination, and then is back online. By having offline agents, much of the dynamics and state space sizes of the Markov game change, and thus many works try to avoid this. Other than that, the algorithms need to be trained and evaluated for larger and larger scenarios. In this thesis, the agents managed to handle a 10×10 grid environment. In reality, the grid block density might have to be increased, and thus larger grid worlds need to be experimented on as well. The number of agents evaluated in the thesis was 32 agents. This number needs to drastically increase, and with the decentralized agents with a limited observation, this should be feasible from a computational perspective, but the nonstationarity of the environment might increase.
There is also a need to implement and develop more algorithms specific to these types of problems. In this thesis, COMA is adapted to a taxi fleet setting. In the original paper, COMA was evaluated in Starcraft, which is a video game that might have been more suitable for this type of algorithm. Other well-performing MARL algorithms could, therefore, be tested.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Jakob N Foerster. *Deep multi-agent reinforcement learning.* PhD thesis, University of Oxford, 2018.

[3] Jakob N Foerster, Gregory Farquhar, Triantafyllos Afouras, Nantas Nardelli, and Shimon Whiteson. Counterfactual multi-agent policy gradients. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[4] Hado V Hasselt. Double q-learning. In *Advances in neural information processing systems*, pages 2613–2621, 2010.

[5] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 AAAI Fall Symposium Series*, 2015.

[6] Pablo Hernandez-Leal, Bilal Kartal, Matthew E Taylor, and AI Borealis. Is multiagent deep reinforcement learning the answer or the question? a brief survey. *learning*, 21:22, 2018.

[7] Geoffrey Hinton. Neural networks for machine learning, lecture 6a. 2016.

[8] John Holler, Risto Vuorio, Zhiwei Qin, Xiaocheng Tang, Yan Jiao, Tiancheng Jin, Satinder Singh, Chenxi Wang, and Jieping Ye. Deep reinforcement learning for multi-driver vehicle dispatching and repositioning problem. *arXiv preprint arXiv:1911.11260*, 2019.

[9] Shariq Iqbal and Fei Sha. Actor-attention-critic for multi-agent reinforcement learning. *arXiv preprint arXiv:1810.02912*, 2018.

[10] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[11] Bengt Lennartson. Lecture notes on discrete event systems. 2019.

[12] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[13] Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, pages 157–163. Elsevier, 1994.

[14] Ryan Lowe, Yi I Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in neural information processing systems*, pages 6379–6390, 2017.

[15] Hangyu Mao, Zhibo Gong, and Zhen Xiao. Reward design in cooperative multi-agent reinforcement learning for packet routing, 2018.

[16] Google Map. The map of beijing. `https://www.google.com/maps/place/Beijing`, 2019.

[17] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[19] NAVINFO. Nacinfo traffic index. `http://www.nitrafficindex.com/`, 2019.

[20] Takuma Oda and Carlee Joe-Wong. Movi: A model-free approach to dynamic fleet management, 2018.

[21] Travis Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006.

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[23] Scott Proper and Kagan Tumer. Modeling difference rewards for multiagent learning. In *AAMAS*, pages 1397–1398, 2012.

[24] Zhiwei Tony Qin, Xiaocheng Tang, Yan Jiao, Fan Zhang, Chenxi Wang, and Qun Tracy Li. Deep reinforcement learning for ride-sharing dispatching and repositioning. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pages 6566–6568. AAAI Press, 2019.

[25] Howard M Schwartz. *Multi-agent machine learning: A reinforcement approach*. John Wiley & Sons, 2014.

[26] Pete Shinners. Pygame. `http://pygame.org/`, 2011.

[27] David Silver. Ucl course on reinforcement learning. 2015.

[28] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[29] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337, 1993.

[30] Beijing traffic management Bureau. Real-time traffic of beijing. `http://jtgl.beijing.gov.cn/jgj/`, 2019.

[31] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[32] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[33] Zhaodong Wang, Zhiwei Qin, Xiaocheng Tang, Jieping Ye, and Hongtu Zhu. Deep reinforcement learning with knowledge transfer for online rides order dispatching. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 617–626. IEEE, 2018.

[34] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[35] Kaiqing Zhang, Zhuoran Yang, and Tamer Başar. Multi-agent reinforcement learning: A selective overview of theories and algorithms. *arXiv preprint arXiv:1911.10635*, 2019.

[36] Ming Zhou, Jiarui Jin, Weinan Zhang, Zhiwei Qin, Yan Jiao, Chenxi Wang, Guobin Wu, Yong Yu, and Jieping Ye. Multi-agent reinforcement learning for order-dispatching via order-vehicle distribution matching. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 2645–2653, 2019.