[redacted]

[redacted]

Deep Q-Learning

A machine learning approach to solve stochastic decision problems

[redacted]

Index

List of Abbreviations

DNN	Deep neural network
DP	Dynamic programming
DRL	Deep reinforcement learning
DDQN	Double deep Q-network
Dueling DDQN	Dueling double Q-network
Dueling DQN	Dueling deep Q-network
DQN	Deep Q-network
ER	Experience replay
IS	Important sampling
MDP	Markov decision process
ML	Machine learning
MC	Monte Carlo
OAP	Order acceptance problem
pER	prioritized experience replay
RL	Reinforcement learning
SDP	Stochastic decision problem
TD	Temporal difference

List of Algorithms

List of Figures

List of Tables

1 Introduction

In economics, decision making under uncertainty is a huge risk. No matter if in the area of capacity planning, order acceptance or investments, previous decisions, which seem profitable at first, can cause great opportunity costs later. Stochastic optimization models are an useful tool to enhance the decision making, lower opportunity costs and increase profits. However, such optimization problems have proven themselves to be difficult to solve. Deep reinforcement learning is a rather new machine learning approach to find optimal decision under uncertainty. Here, a deep reinforcement agent is a decision maker that meets decision based on interactions with an environment that describes the scope of the stochastic decision problem (SDP). While other approaches to stochastic optimization suffer from high computational effort or require a precise description of the environment's dynamics, deep reinforcement learning models enhance the action selection by an iterative interaction with the environment. Hereby, the agent estimates an objective function and optimizes poor decisions of the past through a learning process. Deep Q-networks are one deep reinforcement learning approach, which have drawn interest in research over the last years. On the example of a stochastic order acceptance problem, this work indicates, how deep Q-networks are able to take optimal decisions in SDPs.

Therefore, the general scheme of SDPs, as well as their optimization model, is introduced in Chapter 2. Afterwards, basic solution approaches to solve SDPs, as well as the framework of reinforcement learning are presented. Chapter 3 introduces the advancement of reinforcement learning into deep reinforcement learning, by relying on deep neural networks as function approximators. To close the chapter, several deep Q-network structures are presented, which are trained to solve a stochastic order acceptance problem (OAP). In Chapter 4, the scope of the OAP is presented. Additionally, the results of the different deep Q-learning algorithms are compared to each other and a proposition about the usefulness of each model is made. Lastly, a prospect about a different set up of the algorithms is made, as well as an alternative solution approach.

2 Stochastic Decision Problems

A SDP describes a situation, where an agent takes decisions in an uncertain environment and information are handled by stochastic sets and systems (Wang (2019)). The agent is an entity, which acts depending on observations of the environment. The environment is altered by the agent's actions and describes the scope of the SDP. At time t, the agent receives an observation ω_t and chooses an action a_t, based on a decision making process (Figure 1). The observation ω_t defines the current state $s \in S$ of the environment and therefore the current decision action of the agent. S denotes all possible states of the environment. The agent obtains knowledge about the environment through the interaction and optimizes its decision making over time. Corresponding to the obtained knowledge, the agent optimizes some objective function by developing a policy π, which determines an optimal action a for all possible system states $s \in S$. (Kochenderfer (2015))

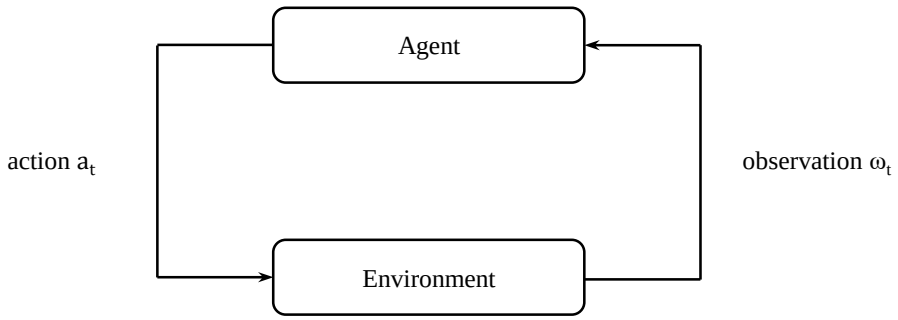

Figure 1 – Agent environment interaction in a SDP, based on Kochenderfer (2015)

A SDP can be illustrated on the example of a stochastic OAP with a planning horizon of T time steps and a capacity limited resource with maximum capacity c^{max}. The objective is the maximization of the rewards from the accepted orders over the planning horizon. At each time step $t \in \{1, \ldots, T\}$, the possible actions a_t of the agent are the acceptance or the rejection of the arriving order o_t. If the

order is accepted, the agent receives a reward r_t. Hereby, the agent has to choose the action without information about future orders or rewards of the planning horizon. In this context, each state $s(c_{rem}, c_{o_t}, t_{rem}, r_t)$ is defined through the remaining available capacity c_{rem} of the resource, the required capacity c_{o_t} of the arriving order o_t, the corresponding reward r_t and the remaining time steps t_{rem} within the planning horizon.

2.1 Markov Decision Process

The *markov decision process* (MDP) describes the optimization model designed for stochastic decision making in an uncertain environment. At each decision time, the environment is represented by a particular state s and the agent chooses an action a. After performing action a, the agent receives a reward r and the environment moves to a new, different state s'. (Kalnoor and Subrahmanyam (2020)) The MDP represents a discrete stochastic optimal control problem and is formally illustrated as tuple $(S, A, p(s_{t+1}, r_t|s_t, a_t), r, \gamma)$, where: (Nian et al. (2020))

- S: State space that describes all possible states of the environment
- A: Action space that includes all possible actions of the agent
- $p(s_{t+1}, r_t|s_t, a_t)$: Transition dynamic function of the environment, which denotes the probability of transitioning to state s_{t+1} and receiving reward r_t, if the agent performs action a_t in state s_t
- r: Expected reward from the environment, after the agent performs action a in state s
- γ: Discount factor ($0 < \gamma \leq 1$), which determines how strong future rewards are taken into account. For small values, immediate rewards are more emphasized

The agent follows a policy π that maps a probability distribution over actions from states: $\pi : S \rightarrow p(A = a|S)$. The policy π constitutes an optimal action $a \in A$, which generates the best reward for each state $s \in S$. In an episodic MDP, where the state is reset after each episode of length T to the initial state s_0, the sequence of states, actions and rewards are a trajectory of the policy. For every trajectory, the accumulated rewards from the environment are denoted by

the return R. Depending on whether the MDP is episodic or not, the discount factor γ is taken into account. $\gamma < 1$ is only required, if the planning horizon is infinite. Hereby, γ ensures a finite value for the sum of the expected rewards (Van Otterlo and Wiering (2012)):

$$R = \sum_{t=0}^{\infty} \gamma^t r_{t+1} \tag{2.1}$$

$$R = \sum_{t=0}^{T-1} r_{t+1} \tag{2.2}$$

The agent searches for the optimal policy π^*, which achieves the maximum expected return R (Arulkumaran et al. (2017)):

$$\pi^* = \underset{\pi}{argmax}\ \mathbb{E}[R|\pi] \tag{2.3}$$

Besides the policy, the agent features two objective functions, the state-value function $V(s)$ and the state-action function $Q(s, a)$. The first describes the expected return of state s, by following policy π:

$$V^{\pi}(s) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|S_0 = s] = \mathbb{E}_{A_t\sim\pi(\cdot|S_t)}[\sum_{t=0}^{\infty} \gamma^t R(S_t, A_t)|S_0 = s] \tag{2.4}$$

Hereby, $\tau \sim \pi$ means the trajectories τ are sampled under the consideration of policy π. $A_t \sim \pi$ describes that action a in state s is derived from policy π. For a given state s_t and action a_t, the state s_{t+1} is determined by the state transition function $p(s_{t+1}, r_t|s_t, a_t)$.

The second objective function, the action-value function $Q(s, a)$, estimates the expected return for a given state s and action a. If an agent acts appropriate to policy π, the action-value or Q-value is defined as (Ding et al. (2020)):

$$Q^{\pi}(s, a) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|S_0 = s, A_0 = a] = \mathbb{E}_{A_t\sim\pi(\cdot|S_t)}[\sum_{t=0}^{\infty} \gamma^t R(S_t, A_t)|S_0 = s, A_0 = a] \tag{2.5}$$

The advantage function $A(s, a)$ characterizes the dependencies between the state-value function and the action-value function (2.6). Following policy π, the advantage function specifies the relation between an action a and the expected reward for each action (François-Lavet et al. (2018)):

$$A^{\pi}(s, a) = Q^{\pi}(s, a) - V^{\pi}(s) \tag{2.6}$$

2.2 Approaches to Solve the Markov Decision Process

Three families of algorithms are employed to solve the MDP (Figure 2). In a perfect system model, *dynamic programming* (DP) provide exact solutions to the optimal policy. Since DP methods only operate in a completely known model of the environment and the computational effort to find an optimal policy is rather high, two other approaches were introduced. *Monte Carlo* (MC) methods and *temporal difference* (TD) algorithms approximate DP solutions with fewer computational effort and without the necessity of a perfect system model. MC methods find optimal policies by averaging the objective function over many sampled trajectories of states, actions and rewards. TD methods are a combination of DP and MC methods, by learning from sampled data like MC methods and also being able to execute mid trajectory learning like DP. (Nian et al. (2020)) In the following, these algorithms are explained in more detail.

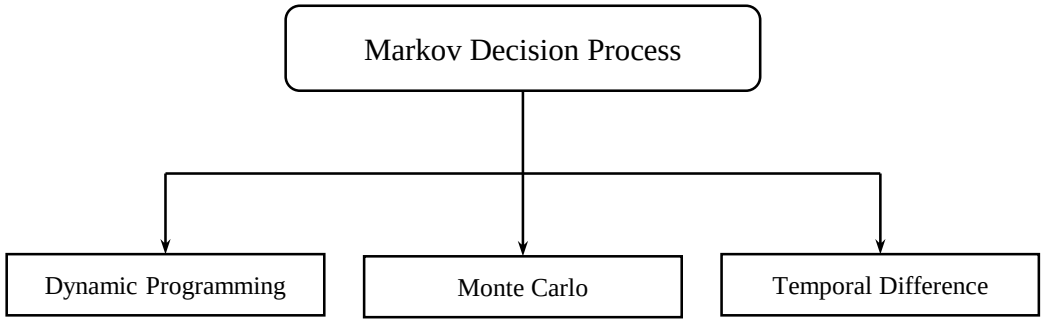

Figure 2 – Algorithms to solve the MDP, based on Nian et al. (2020)

2.2.1 Dynamic Programming

DP refers to a collection of algorithms, which are able to compute optimal policies with a perfect model of an environment. The key idea of DP is the disposition of the state-value function v to structure and organize the search for the optimal policy. From the optimal state-value function v_*, the optimal policy π_* is obtained. Therefore a policy iteration is utilized that covers two iterative steps, the policy evaluation and the policy improvement. The policy evaluation predicts the state-value function v for a policy π through an iterative approach. The policy improvement identifies solutions, where $v_{\pi'}(s) \geq v_\pi(s)$ is satisfied for any state.

The search for better policies π' continues iteratively until a policy is found, which realizes $v_{\pi*}(s) \geq v_{\pi \neq \pi*}(s)$ for all states $s \in S$. The policy iteration is visualized in equation 2.7, where $\xrightarrow{\text{E}}$ represents the policy evaluation step and $\xrightarrow{\text{I}}$ the policy improvement step. (Sutton and Barto (2018))

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \pi_2 \ldots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} v_* \tag{2.7}$$

2.2.2 Monte Carlo

MC methods estimate the action-value function by repeatedly generating episodes and logging the average return for each state-action pair. Since MC methods are model free, they do not require any knowledge of transition probabilities. However to ensure convergence, the number of episodes should be large and every state and action must be recorded by a significant number of times. (Nguyen et al. (2020)) The policy search of MC methods is similar to the policy iteration in DP algorithms (2.8). However, there are three main differences. Firstly, not all states are updated simultaneously. Secondly, the value function is updated according to sampled data from the agent's interaction with the environment and lastly, the action-value function $q_\pi(a, s)$ is estimated instead of the state-value function $v_\pi(s)$:

$$\pi_0 \xrightarrow{\text{E}} q_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} q_{\pi_1} \xrightarrow{\text{I}} \pi_2 \ldots \xrightarrow{\text{I}} \pi_* \xrightarrow{\text{E}} q_* \tag{2.8}$$

The training begins with the agent in an unknown system. The agent traverses through the state space $s_1, s_2, \ldots, s_n$, performing actions $a_1, a_2, \ldots, a_n$, following policy π and collecting rewards $R_1, R_2, \ldots, R_n$. Upon termination, a sequence of returns $G_0, G_2, \ldots, G_{n-1}$ or the discounted cumulative return, received of the m^{th} step $G_m = \sum_{i=0}^{n} \gamma^i R_{m+i}$, is calculated. Afterwards, the action values $Q(s, a)$ are estimated for each state-action pair. After the action-value function $q(s, a)$ reaches convergence, the optimal policy $\pi^*(s)$ is extracted. (Nian et al. (2020))

2.2.3 Temporal Difference

TD algorithms are a model free approach for the policy evaluation of DP methods and learn from experience like MC methods. In contrast to MC methods, TD methods do not observe the complete return of an episode. Instead, the objective function of the next state approximates the expected return and updates are

performed on every step within an episode. (Nguyen et al. (2020), Silver et al. (2012)) After observing s_{t+1} and receiving R_{t+1}, the TD update for the value function is computed, according to: (Sutton (1988))

$$V(s_t) \leftarrow V(s_t) + \alpha \cdot \delta_t \tag{2.9}$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \cdot \delta_t \tag{2.10}$$

Hereby, $\leftarrow$ denotes the update operator with the TD error δ_t at time t (2.11, 2.12), measured between the value of the objective function in state s_t and the value of the objective function in the next state s_{t+1}. In respect to δ_t, the objective function is adjusted by a step-size parameter α. After the objective function converges, the optimal policy π^* is estimated.

$$\delta_t = R_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{2.11}$$

$$\delta_t = R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \tag{2.12}$$

2.3 Reinforcement Learning

Reinforcement learning (RL) is a machine learning (ML) approach to solve optimal control problems such as MDPs. RL provides (near) optimal solutions for non-linear stochastic control problems, even if the dynamics of the system are unknown or affected by significant uncertainty. Hereby, RL techniques enable a numerical function approximation. Furthermore, RL methods are able to learn from samples of transitions and rewards from the environment without the requirement of a model of the system dynamics. (Buşoniu et al. (2018)) The decision making process in RL is equal to a MDP. The agent resides in a given state $s_t \in S$ of the environment and captures an initial observation $\omega_t \in \Omega$. At each time step t, the agent chooses an action $a_t \in A$ (Figure 3). Afterwards, the state of the environment changes to a new state $s_{t+1} \in S$, the agent receives a reward $r_t \in R$ and obtains a new observation $\omega_{t+1} \in \Omega$. (François-Lavet et al. (2018))

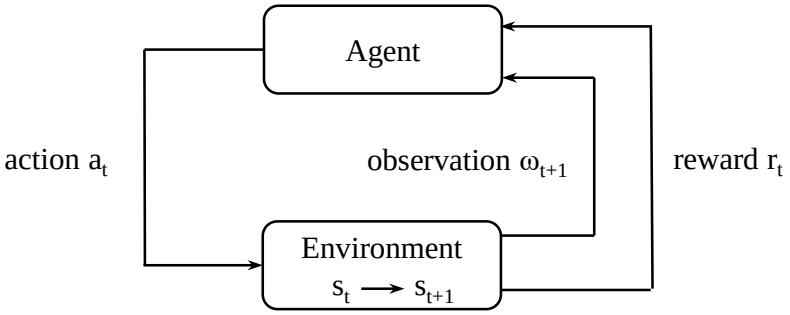

Figure 3 – Agent-environment interaction in RL, based on François-Lavet et al. (2018)

Depending on the knowledge of the system dynamics $p(s_{t+1}|s_t, a_t)$, RL is subdivided into *model-free* or *model-based* reinforcement learning. If $p(s_{t+1}|s_t, a_t)$ are known or are approximable with a learned model $\hat{p}(s_{t+1}|s_t, a_t)$, *model-based* RL is realized. In case of unknown system dynamics $p(s_{t+1}|s_t, a_t)$, *model-free* RL is performed. (Gu et al. (2016)) Although there are many *model-free* RL algorithms, most of them are further divided into two families, *value-based methods* and *policy search methods*. In *value-based methods*, the action-value function is fitted, which estimates the expected reward for taking a particular action at a particular state by following a particular policy. (O'Donoghue et al. (2016)) In *policy search methods*, the policy is represented by a probability distribution $\pi_\theta(a|s) = \mathbb{P}[a|s, \theta]$. Hereby, the action a in state s is chosen according to the parameters θ, and the policy is adjusted into the direction of greater cumulative rewards (Silver et al. (2014)).

Several differentiation between RL methods are possible. Firstly, a distinction is made between *on-policy* and *off-policy* learning (Figure 4). For the former, a batch of behavior is collected, wherein one policy is used to act in the environment. Afterwards a policy gradient is computed from this data. All parameter updates are made by using data collected from the trajectory distribution, precipitated by the current policy of the agent. At this juncture, each data point of the agent's behavior is only used once. (Gu et al. (2017)) The latter reuses samples of the agent's behavior by storing them in a buffer D and a value function is trained with *off-policy* updates. Each policy π_t collects new data and at time t, D consists of samples from $\pi_0, \pi_1, \ldots, \pi_t$. (Levine et al. (2020))

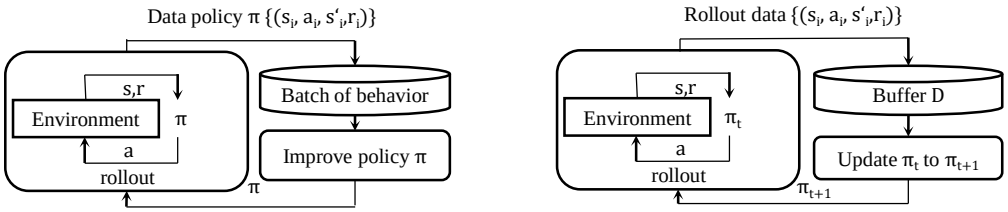

Figure 4 – On-policy RL (left) vs Off-policy RL (right), based on Levine et al. (2020)

A further differentiation is possible between *online* and *offline* learning (Figure 5). For *online* RL, the policy π_t is updated with data collected by π_t itself and the agent gathers experience in the environment. The *offline* RL accumulates a data set D with some behavior policy π_β. Once collected, the data set is not altered and employed for the training phase of the agent. (Levine et al. (2020))

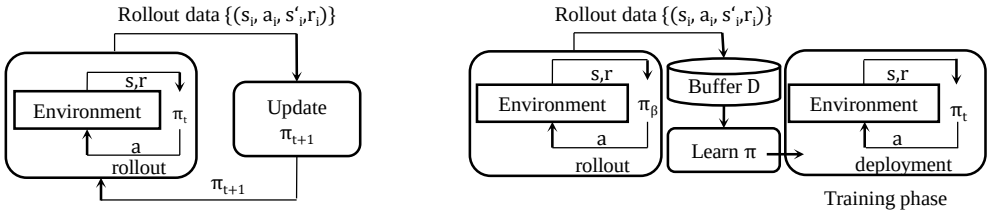

Figure 5 – Online RL (left) vs Offline RL (right), based on Levine et al. (2020)

3 Deep Reinforcement Learning

In deep reinforcement learning (DRL), various components of the agent, such as the policy π or one of the objective functions $Q(s, a)$ or $V(s)$ are estimated with deep neural networks (DNN, Figure 6). Therefore, the parameters θ of the DNN are trained to minimize a suitable loss function. (Hessel et al. (2017))

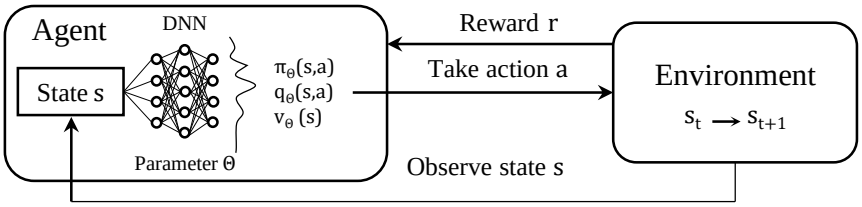

Figure 6 – Concept DRL, based on Mao et al. (2016)

Besides DNN as function approximators, DRL features more elements (Figure 7), which are able to improve the agents interaction with the environment. A exploration and exploitation policy helps the agent to gather experience within the environment. The replay memory stores experience to reprocess it at a later time during training. Before explaining the DRL models for the experiments in Chapter 3.4, the basic functionality of DNN, the trade-off difficulty between exploration and exploitation and possible arrangements for the replay memory are clarified in more detail.

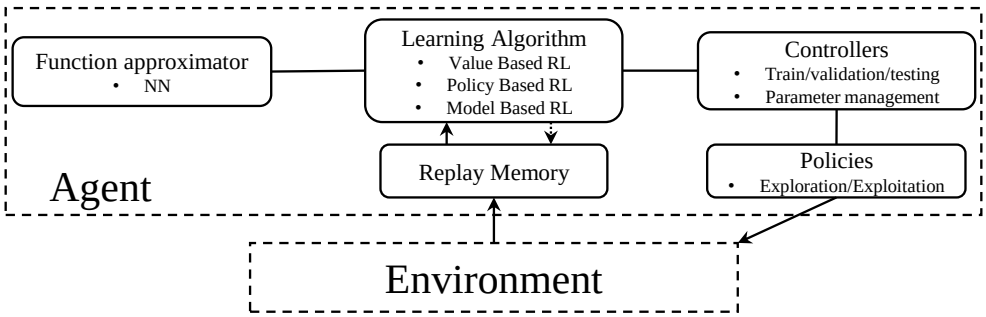

Figure 7 – General scheme DRL methods, based on François-Lavet et al. (2018)

3.1 Deep Neural Networks

Deep neural networks are information processing systems that are simulated in computers (Kruse et al. (2011)). Although their structure and exercise are different, the operating of DNN is always the same (Figure 8). The network receives a M dimensional input-vector X_M and transforms it into a N dimensional output-vector Y_N (Scherer (2013)). The networks consist of input-, output- and hidden layers, where each layer comprises single neurons. The neurons in each layer are connected to other neurons and receive an input signal that is multiplied with a weighting coefficient W_i. The weighted input signals are summed to a transfer function $u(t) = \sum_{i=1}^{j} X_i \cdot W_i$. By the use of an activation function, the transfer function determines the activation condition of the single neurons. If a threshold δ is exceeded, the neuron is activated and transmits an output value $Y(t)$. If δ is not reached, the neuron stays inactive and transfers a deviant output value. (Styczynski et al. (2017))

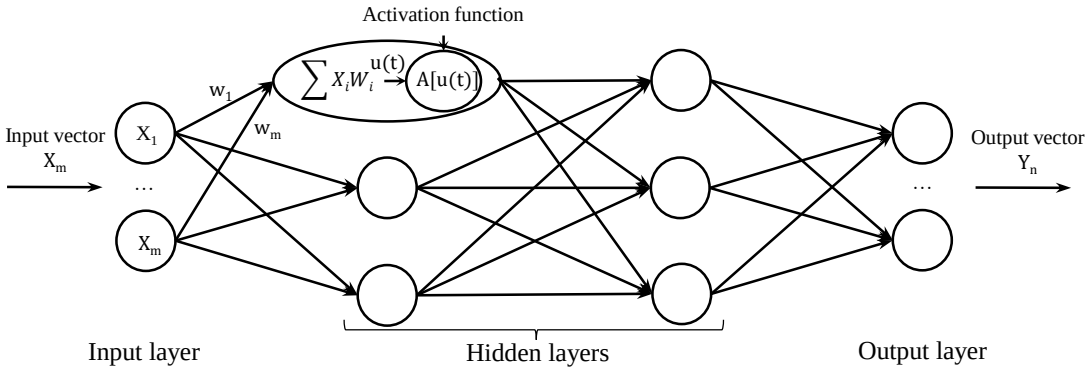

Figure 8 – Structure and operating of DNN

DNN are able to learn from experience and map any complex functional correlation (Zhang et al. (2016)). Hereby, DNNs with only two hidden layers and a sufficient large number of neurons are capable of approximating any given function (Lu and Lu (2020)). The learning of DNN is accomplished by a training process. Hereby, a loss function is optimized by adjusting the parameters of the network such as the weighting coefficients W_i. The training comprises three steps that are repeated iteratively. First, the network receives an input that is propagated forward from the input layer to the output layer. Second, the loss function calculates a distance value between the computed output and the expected out-

put. Third, an optimizing step updates the network's parameters according to the distance value. Since the update is performed from the output layer to the input layer, the last step is also called backpropagation. The training process of the networks is accomplished, if the loss function converges to some value z and the closer z is to zero, the better is the result of the network. (Chollet (2018))

3.2 Exploration and Exploitation

A major challenge in RL is the trade-off between exploration and exploitation, whereat the agent has to decide whether to investigate its environment or leverage its past experience to choose an action. To prevent a stagnation of the policy and its returns, the agent must do both, explore and exploit. (Sledge and Príncipe (2017)) The sampling behavior of the agent is exploration, if it chooses actions independent of past experience. The exploration is supposed to improve the agent's knowledge about the reward generating process. In contrast, the sampling behavior of the agent is exploitation, if the choice of action is dependent to the historical experience. The exploitation utilizes the experience of the agent to optimize the objective function. (Chen et al. (2009)) Although many algorithms address the balance between exploration and exploitation, two rather simple basic approaches are commonly used, the *ϵ-greedy exploration* and the *softmax exploration*. (Vamplew et al. (2017), Masadeh et al. (2018)).

3.2.1 The ϵ-Greedy Exploration

The ϵ-greedy algorithm employs an exploration probability ϵ to balance between the exploration and exploitation. Within the algorithm, a random action is selected with probability ϵ and the current best action is chosen with probability $1-\epsilon$. In an adaptive ϵ-greedy algorithm, the value of the exploration probability ϵ decreases with time and the exploitation probability increases. If ϵ reaches the value zero, the exploration of the agent ends and it only follows the exploitation strategy. The decrease of the parameter ϵ is performed differently. (Masadeh et al. (2018)) For example, with a constant reducing factor ϵ_{dec} a linear decrease of ϵ is realized for $\epsilon_{t+1} = \epsilon_t - \epsilon_{dec}$. A different approach is a non-linear decrease, where the reducing factor is changed over time with $\epsilon_{dec,t} = \epsilon_{dec} * t$ and $\epsilon_{t+1} = \frac{1}{\epsilon_{dec,t}}$.

3.2.2 The Softmax Exploration

The softmax exploration takes the relative value of each action into account and the probability of choosing action a in state s is determined by: (Vamplew et al. (2017))

$$p(s_t, a) = \frac{e^{\frac{Q(s_t,a)}{\tau}}}{\sum_{b=1}^{n} e^{\frac{Q(s_t,b)}{\tau}}} \tag{3.1}$$

Hereby, $\tau \geq 0$ denotes a temperature parameter to control the exploration/exploitation trade-off. For $\tau \to 0$, the agent does not explore at all and actions are chosen according to the current best action. For $\tau \to \infty$ the agent explores and thus chooses actions randomly. For intermediate values of $\tau \in]0, \infty[$, the agent selects the best action with high probability. Compared to the ϵ-greedy algorithm, the other actions are ranked and are not chosen completely at random. (Tijsma et al. (2016))

3.3 Prioritized Replay Buffer

In an *off-policy* DRL setup, the agent's behavior is stored in a replay buffer. The parameters of the DNN are learned with experience replay (ER), by taking advantage of previously experienced transitions, sampled from the replay buffer. Besides providing uncorrelated data to train the DNN, the ER also improves the data efficiency (Zhang and Sutton (2017)). Hereby, the ER does not only stabilizes the learning of the network, moreover it prevents the DNN from overfitting recent experience (Foerster et al. (2017)). At each learning step, the transition (s_t, a_t, r_t, s_{t+1}) is added to the replay buffer and employed in the upcoming training phases.

A derivative of the ER is the prioritized experience replay (pER). Instead of uniformly choosing experiences from the buffer, the pER structures the experience for the learning phase (Liu and Zou (2018)). The uniform sampling strategy of the ER causes inefficient learning, as it does not respect the importance of single transitions (Brittain et al. (2019)). Furthermore the pER improves the sampling efficiency, since the use of samples which cause a larger error enhances the training of DNN (LeCun et al. (2012)). A central component of the pER is the criterion to measure the importance of each transition. Schaul et al. (2015) present two

approaches to choose experience form the replay buffer, the prioritizing with *TD error* and the *stochastic prioritization*.

The pER with TD error prioritization utilizes the TD error to indicate the unpredictability of the transition. Along with each transition, the TD error is recorded in the replay buffer and the transition with the largest absolute TD error is employed to train the agent. After each learning step, the TD error of the transition is updated. In contrast, the stochastic prioritization interpolates between a pure greedy prioritization and uniform random sampling. The probability of sampling transition i is given by:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{3.2}$$

where $p_i > 0$ denotes the priority of transition i and α determines the amount of prioritization used. Hereby, $\alpha = 0$ corresponds to a pure uniform case. The calculation of p_i accords with the TD error and is performed differently. For an indirect, rank based prioritization, the priority $p_i = \frac{1}{rank(i)}$ is predicated on the rank of the TD error from transition i. Another approach employs the TD error directly to compute the priority $p_i = |\delta_i| + e$, where the addition of a small positive parameter e prevents p_i from being zero.

A drawback of the pER is a changed value, the estimate converges to, caused by biases from sampling in an uncontrolled fashion. Importance sampling (IS, 3.3) weights w_i are employed to correct these biases.

$$w_i = \left(\frac{1}{N} * \frac{1}{P(i)}\right)^\beta \tag{3.3}$$

Hereby, $P(i)$ denotes the probability of sampling transition i and N determines the number of samples stored in the replay buffer. The IS weights compensate the non-uniform probabilities $P(i)$ for $\beta = 1$. These weights are implicated into the learning update by using $w_i * \delta_i$ instead of δ_i as loss. To prevent an upscale of the network's parameter's updates, the weights are normalized by $\frac{1}{max\ w_i}$. Unbiased updates are most important at the end of training near convergence. Therefore, the IS are annealed by increasing β, until it reaches the value 1. Furthermore, the choice of β interacts with the choice of the prioritization exponent α. However, while β grows, α is decreased over time.

3.4 Deep Reinforcement Learning Models

TD algorithms form the core of RL by iteratively updating an estimate of an objective function corresponding to a given policy π, using temporally-successive samples. Although there are many different TD algorithms, the TD(0) algorithm has gained attention and is nowadays often relied on by many state-of-art DRL solutions. More precisely, the TD(0) algorithm serves as update condition for the parameters θ of the DNN:

$$\theta_{t+1} = \theta_t + \alpha_t[r_t + \gamma\Phi'_t\theta_t - \Phi_t\theta_t] \tag{3.4}$$

where α_t represents the learning rate of the network and $\Phi'_t\theta_t$, $\Phi_t\theta_t$ denote the specific result of the value function in state s_{t+1}, and s_t respectively. The term $r_t + \gamma\Phi'_t\theta_t$ is also known as the TD target. (Dalal et al. (2017))

Examples for *value-based* DRL methods are deep Q-networks (DQN, Mnih et al. (2015)), double deep Q-networks (DDQN, Van Hasselt et al. (2015)) and dueling deep Q-networks (dueling DQN, Wang et al. (2016)). These algorithms are presented in the following.

3.4.1 Deep Q-Network

DQNs are an *off-policy* DRL approach, where a DNN Q, referred to as Q-network, estimates the optimal action-value function:

$$Q^*(s,a) = \underset{\pi}{max}\mathbb{E}[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots | s_t = s, a_t = a, \pi] \tag{3.5}$$

During the training, the parameters θ_i of the Q-network are adjusted in each iteration step i. A second DNN $\hat{Q}$, referred to as target network with the parameters θ_i^-, obtained from a previous state of the Q-network, calculates the estimated return r_{t+1} of the next state s_{t+1}. This target value $Y_t^{DQN} = r_t + \gamma \underset{a}{argmax}\, \hat{Q}(s_{t+1}, a_{t+1}; \theta_i^-)$ is then used to update the parameters θ_i of the Q-network. The target network stabilizes the learning process by evaluating the action selection and consequently arranges the training more effectively. A loss function $L_i^{DQN}(\theta_i)$ performs the update of the parameters θ_i (3.6). Since an ER stores the agent's experience $e_t = (s_t, a_t, r_t, s_{t+1})$ in a data set $D_t = \{e_1, \ldots, e_t\}$, the parameter updates are applied on samples of experience $(s_t, a_t, r_t, s_{t+1}) \sim U(D)$.

The parameters θ_i^- of the target network are replaced with the parameters θ_i of the Q-network after every C steps and stay unaltered otherwise. (Mnih et al. (2015))

$$L_i^{DQN}(\theta_i) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim U(D)}[(Y_t^{DQN} - Q(s_t, a_t; \theta_i))^2] \tag{3.6}$$

Since the loss function is based on the TD-error δ_t^{DQN}, the DQN algorithm is called a TD algorithm and the loss function is defined as: (Achiam et al. (2019))

$$L_i^{DQN}(\theta_i) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim U(D)}[(\delta_t^{DQN})^2] \tag{3.7}$$

with the TD error:

$$\delta_t^{DQN} = r_t + \gamma \max_a \hat{Q}(s_{t+1}, a_{t+1}; \theta_i^-) - Q(s_t, a_t; \theta_i) \tag{3.8}$$

A pseudo code of the DQN algorithm is presented in the Appendix (Page 35).

3.4.2 Double deep Q-Network

Since DQN tend to overestimate values by using the same values to select and to evaluate an action, the estimates are frequently overoptimistic. DDQN prevent overestimation, by decoupling the selection and the evaluation of an action. Therefor, the $argmax$ operation in the target network $\hat{Q}$ is distinguished into action selection and action evaluation. The action evaluation is supported by the Q-network $\underset{a}{argmax}\ Q(s_{t+1}, a_{t+1}; \theta_i)$ and the target network estimates the Q-value $\hat{Q}(s_{t+1}, \underset{a}{argmax}\ Q(s_{t+1}, a_{t+1}; \theta_i), \theta_i^-)$. The target value Y_t^{DDQN} depends on both, the action selection and the action evaluation: (Van Hasselt et al. (2015))

$$Y_t^{DDQN} = r_t + \gamma\ \hat{Q}(s_{t+1}, \underset{a}{argmax}\ Q(s_{t+1}, a_t; \theta_i); \theta_i^-) \tag{3.9}$$

This changes the loss function to:

$$\begin{aligned} L_i^{DDQN}(\theta_i) &= \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim U(D)}[(Y_t^{DDQN} - Q(s_t, a_t; \theta_i))^2] \\ &= \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim U(D)}[(\delta_t^{DDQN})^2] \end{aligned} \tag{3.10}$$

with the TD-error:

$$\delta_t^{DDQN} = r_t + \gamma\ \hat{Q}(s_{t+1}, \max_a\ Q(s_{t+1}, a_{t+1}; \theta_i); \theta_i^-) - Q(s_t, a_t; \theta_i) \tag{3.11}$$

The update of the target network's parameters θ^- stays the same as in DQN and is accomplished in a periodic copy of the Q-networks parameters θ after every C steps. The complete algorithm is shown in the Appendix (Page 36).

3.4.3 Dueling deep Q-Network

The dueling DQN estimates the action-value function indirectly with the sum of the state-value function $V(s)$ and the advantage function $A(s, a)$:

$$Q(s, a) = V(s) + A(s, a) \tag{3.12}$$

Therefor, the DNN is split into two streams at some point (Figure 9). One of them computes a scalar value, representing $V(s; \theta, \alpha)$, and the other outputs an $|\mathbb{A}|$-dimensional vector $A(s, a; \theta, \beta)$. At this juncture, θ denotes the parameters of the shared network (black layers and connections), α represents the parameters of the stream which computes the state-value function V (gray stream) and β determines the parameters of the stream which approximates the advantage function A (blue stream). Afterwards, an aggregation module combines the two functions to the action-value function:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \alpha) + A(s, a; \theta, \beta) \tag{3.13}$$

The direct use of the functional context in equation 3.13 results in poor practical performance, since it is not possible to recover the values of V and A uniquely for a given value Q. For improvement, the relative advantage of all actions is employed instead by subtracting the average advantage across all possible actions:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \alpha) + (A(s, a; \theta, \beta) - \frac{1}{|A|} \sum_{a'} A(s, a'; \theta, \beta)) \tag{3.14}$$

Since the dueling DQN architecture has the same input-output interface as standard DQNs, it is possible to execute the same learning algorithms as with DQN or DDQN. The loss function and the TD error are computed equally to equations 3.6, 3.10 and 3.8, 3.11 for dueling DQN or dueling DDQN respectively. (Wang et al. (2016)) Both algorithms are presented in the Appendix (Dueling DQN (Page 37), Dueling DDQN (Page 38)).

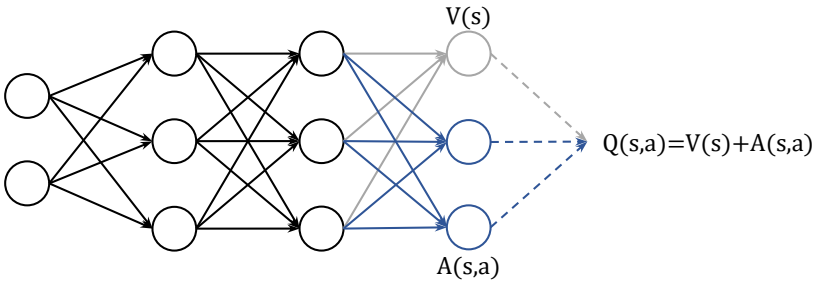

Figure 9 – Architecture dueling deep Q-network, based on Wang et al. (2016)

4 Experiments

To gather further insights into the performance of the different presented DRL models, this chapter presents a technical evaluation focusing on three different aspects. Firstly, the policy of DRL agents with ER, trained with DQN, DDQN, Dueling DQN or Dueling DDQN are compared. Since all algorithms are basic deep Q-learning approaches, it is reviewed which of them is suited best to apply in stochastic OAP. The second experiment investigates the performance of the same networks, however, a pER is employed instead of the ER. Since the pER should improve the learning behavior of the DNN, it is expected that the agents which possess a pER perform better during the training and find better policies with less computational effort. The last experiment tests the policy of the best network with ER against the best policy of the best network with pER. Hereby, the aim is to investigate whether the use of a pER increases the performance of the agents. Each DRL agent searches for optimal policies through interaction with the environment in a training phase with 5000 planning horizons in a stochastic OAP. At each planning horizon, the agent receives 20 orders with rewards and has to decide, whether to accept or reject the order. The environment is specified with a mathematical model of the stochastic OAP in the following chapter. Afterwards the network structure of the DNN is presented for each agent. Chapter 4.3 describes the different interactions of the agents during the training and evaluation process. Lastly, the results of the experiments are presented in Chapter 4.4 and a proposition about the best DRL model to solve the OAP under uncertainty is derived.

4.1 Environment

The environment describes a stochastic OAP, where a random order arrives at each time step $t \in \{1, \ldots, T\}$ (Figure 10). If the order is accepted, a reward is obtained. The OAP is specified as an episodic MDP, where the environment is

reset after a planning horizon of length T and an agent decides whether to accept or reject an arriving order, without information about future orders or rewards.

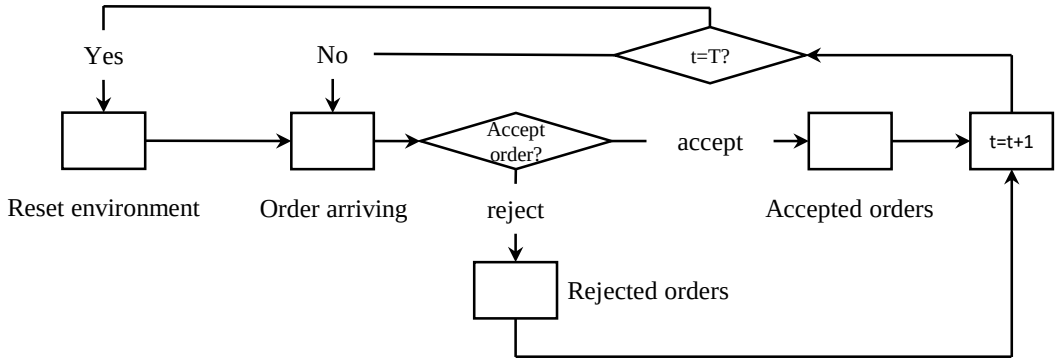

Figure 10 – Order acceptance problem

All incoming orders belong to a certain order class $n \in \{1, \ldots, N\}$. The probability that order n arrives at time t is $p(n) = \frac{1}{N}$. Each order consumes capacity $c_{n,j}$ on $j \in \{1, \ldots, J\}$ identical, capacity limited resources (Table 1).

Table 1 – Required capacity $c_{n,j}$ of order class n on resource j

n \ j	1	2	3	4	5	6
1	1	2	3	2	1	0
2	1	1	1	1	0	0
3	0	2	2	0	0	0
4	2	2	1	1	0	0
5	1	3	2	2	1	0
6	2	1	1	1	3	3

At the beginning of each planning horizon, each resource j provides a maximum available capacity c_j^{max}. Each order class has a capacity related revenue $r_{c_{n,j}} \in \{1, \ldots, R_{c_{n,j}}\}$ and all possible rewards are given by:

$$r_{n,r_{c_{n,j}}} = \sum_{j=1}^{J} r_{c_{n,j}} \cdot c_{n,j} \qquad \forall\, r_{c_{n,j}} \in R_{c_{n,j}}, n \in N$$

If an order of class n arrives at time step t, the reward $r_{c_{n,j}}$ is randomly determined with probability $p(r_{c_{n,j}}) = \frac{1}{R_{c_{n,j}}}$. The parameters of the environment are presented in Table 2. Each system state $s \in S$ is characterized by

$(c_{n,j}, c_{occ,j}, t_{rem}, R_{n,t})$, where $c_{n,j}$ is the required capacity of the order class, $c_{occ,j}$ denotes the occupied capacity on resource j, t_{rem} defines the remaining time steps of the planning horizon and $R_{n,t}$ is the reward for order n at time t. The state dependent actions A of the MDP are given by:

$$A[(c_{n,j}, c_{occ,j}, t_{rem}, R_{n,t})] = \begin{cases} a_1, & \text{if the order is rejected} \\ a_2, & \text{if the order is accepted} \end{cases}$$

According to the action $A[(c_{n,j}, c_{occ,j}, t_{rem}, R_{n,t})]$, a state dependent reward R^A is received. If an accepted order exceeds the available capacity on a resource j, a penalty reward r_{pen} is sustained:

$$R^{a_1}[(c_{n,j}, c_{occ,j}, t_{rem}, R_{n,t})] = 0 \qquad \forall s \in S$$

$$R^{a_2}[(c_{n,j}, c_{occ,j}, t_{rem}, R_{n,t})] = \begin{cases} r_{n,r_{c_{n,j}}}, & \text{if } c_{occ,j} + c_{n,j} \leq c_j^{max} \qquad \forall j \in J,\ s \in S \\ r_{pen}, & \text{else} \end{cases}$$

The decision, whether to accept an order or not depends on an optimal policy π^*, which assigns an optimal action to each state $(c_{n,j}, c_{occ,j}, t_{rem}, R_{n,t})$ and maximizes the cumulative return R_T over all episodes:

$$Max!\ R_T = \sum_{t=0}^{T-1} R_t^A[(c_{n,j}, c_{occ,j}, t_{rem}, R_{n,t})] \tag{4.1}$$

Table 2 – Parameters of the environment

Parameter	Value	Meaning
T	20	number of episodes
N	6	number of order classes
J	6	number of resources
c_j^{max}	6	maximum capacity of ressource j
$R_{c_{n,j}}$	9	number of possible rewards
r_{pen}	-20	penalty reward

4.2 Model Definition

Besides the output layer, the structure of all networks is the same (Table 3). Each network consists of two hidden layers with $n_{hidden} = 256$ neurons respectively. The number of input neurons $n_{input} = 14$ equates the dimension of each state $(c_{n,j}, c_{occ,j}, t_{rem}, R_{n,t}) \in S$ and the number of output neurons is determined by the estimated objective function. The DQN and DDQN compute the Q-value directly and their number of output neurons is 2. For the Dueling networks, two output streams were used to approximate $A(s, a)$ and $V(s)$ respectively and afterwards, the Q-value $Q(s, a)$ is computed according to these values. Additional, all networks possess dropout layers. The dropout reduces the error during the training by temporally removing single neurons. The probability for each neuron to dropout is $p_{dropout} = 0.1$.

Table 3 – Overview network structure

DRL model	input neurons	hidden neurons	dropout probability	output neurons	
DQN	14	256	0.1	**Q(s,a)**	2
DDQN	14	256	0.1	**Q(s,a)**	2
Dueling DQN	14	256	0.1	**V(s):** **A(s,a):**	1 2
Dueling DDQN	14	256	0.1	**V(s):** **A(s,a):**	1 2

The loss $L_i(\theta_i)$ is based on the TD error (3.9, 3.10), which is modified with the IS weights w_i (3.3) for the networks which use the pER:

$$L_i(\theta_i) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim U(D)}[(w_i * \delta_t)^2] \tag{4.2}$$

For each learning step, the network receives a batch with 512 training samples. If the pER is employed, the replay buffer is subdivided into batch-sized parts and one random sample from each part is employed to train the network. In contrast, the training batch is completely chosen at random for the networks with ER. The learning rate is set to $lr = 0.0005$ and the *RMSprop* serves as optimizer. Furthermore, the $ReLU$ activation function is employed, which outputs the value zero for input $x \leq 0$ and x for $x > 0$. Lastly, the update frequency C of the target network needs to be determined. Updating the target network after too

few transitions reduces the usage of the second network and an update after too many transitions decreases the accuracy of the TD error. Therefore, the update frequency of the target network is set to $C = 100$ transitions, which equals 5 planning horizons.

4.3 Training and Evaluation

To evaluate the best policy of each agent, the agent interacts with training and validation data during the training phase. The training data consists of randomly determined orders with random rewards and the validation data comprises a set of fixed orders with fixed rewards. During the training, the pER uses two parameters α and β, which are changed dynamically (Figure 11). The parameters stay unaltered during the first 512 transitions to collect data for the replay buffer. Furthermore, the value ϵ of the ϵ-greedy exploration is another subject of dynamic change (Chapter 3.2.1, Table 4). The end value $\epsilon_{end} = 0.03$ is chosen to be greater than zero, so that the agent keeps exploring in the training phase to find new states of the environment and thus develops new policies. During validation, ϵ is set to zero and the agent only exploits and therefore evaluates its current policy.

Table 4 – Parameters of the DRL models

parameter	start value	end value	annealing factor	episodes to reach the end value
α	0.7	0.02	0.99995	3567
β	0.3	1	1.000017	3567
ϵ	0.99	0.03	0.0001	510

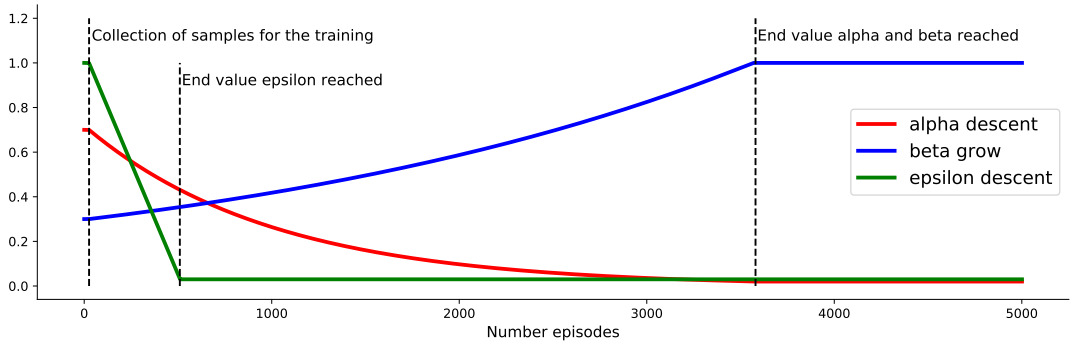

Figure 11 – Change of dynamic parameters during the training

The agent decides corresponding to an optimal policy π^*, if it achieves an average reward of 204.97 units over 50 planning horizons. Since it is not guaranteed that the agent finds an optimal policy during the training, every time the agent exceeds an average reward of at least 195 units over 50 planning horizons, the validation data is applied. The validation data is not altered during the training phase, whereby the network structure achieving the highest reward in the validation data is stored and employed in the test phase. According to the validation phase, a fixed data set is used and ϵ is set to zero during the test phase. The test data suffices to compare the different DRL models, such that the model, which achieves the highest average reward follows the best policy.

Both, the validation and test data, have a size of 1000 planning horizons. The orders are randomly determined, as well as the reward per capacity for the corresponding order. To prevent having noise in the data, the distribution of the orders and rewards is reviewed (Figure 12). The *Validation Orders* and *Test Orders* show the amount of each order with its corresponding reward within the validation and test data. Although the occurrence of each reward per order is not exactly the same, their count is in an interval $[320, 400]$ and thus reasonably evenly distributed. Since the orders occur randomly, a uniform distribution of the orders is not required. The *Validation Order per Time Step* and *Test order per Time Step* show the frequency of each order type at each time step of the planning horizon. The occurrence of each order type is in an interval between $[130, 200]$. In parallel to the distribution of the reward per capacity for each order, the distribution of the orders over all time steps is not uniform. However, since each order occurs sufficiently often, it is assumed that there is no noise within the data and the different amount of the order occurrence describes the stochastic character well. Since both evaluation data sets only provide a small example of possible compositions between orders and rewards, it is not possible tp prove on the basis of the evaluation data that an agent follows a global optimal policy for the stochastic OAP. However, in reference to the *Bellman principle of optimality* (Rosu (2002)), which says that a global optimal solution is composed of local optimal solutions, it is possible to verify that the agents do not follow an optimal policy.

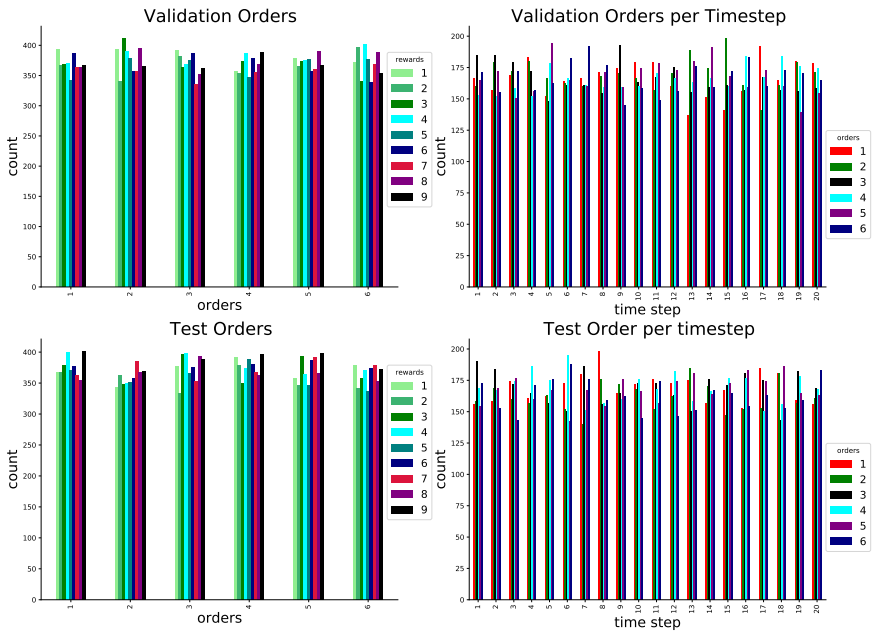

Figure 12 – Composition of the validation and test data

The optimal average return for a SDP over all planning horizons within the validation and test data is presented in Table 5, as well as the optimal solution of a mixed integer program (MIP).

Table 5 – Different solution between SDP and MIP

	Average reward		**Average accepted orders**	
Data set	**SDP**	**MIP**	**SDP**	**MIP**
Validation data	205.532	230.388	3.658	4.025
Test data	205.835	230.96	3.661	3.987

For the MIP solution, all orders of a planning horizon are known, before a decision about the acceptance or rejection of the single orders is made. The decision making under certainty results in a higher average reward across the data sets of almost 25 units. Since the MIP provides information about all orders of the planning horizon, different orders are accepted, compared to the SDP. For example, single orders with high rewards might be rejected and instead two alternative

orders might be accepted, for that the individual reward is lower and the accumulated reward is higher compared to the rejected order. Furthermore, the average accepted orders for the MIP is higher than the average of the accepted orders in the SDP, since it is possible to utilize the capacity more precisely. In comparison of the two sets of evaluation data, the average reward in the validation data is lower and the average accepted orders are higher. As a consequence, the accepted orders in the test data offer a higher reward.

4.4 Results

In the following the results of the conducted experiments are presented. Firstly, the performance of the different Q-networks without a pER are compared to each other. The second part deals with the performance of the different Q-networks with pER and the last part surveys the best Q-network without pER against the best Q-network with pER.

Comparison of the deep Q-networks

The training of the different algorithms proceed rather similar (Figure 13). At the beginning of the training phase, the exploration probability is high and many random actions are taken. Therefore within the first planning horizons, the average reward is around or below zero. Equivalent to the decrease of the exploration probability, the average reward of the planning horizons increases. Shortly after reaching the lower bound of ϵ, all algorithms oscillate around an average return of 200 units. This variation can be explained by the value of $\epsilon_{end} = 0.03$. Statistically, three random actions are selected in five planning horizons, causing either an exceed of the available capacity and consequently a penalty reward, or the acceptance of an unprofitable order, whereby the capacity is blocked for more profitable orders. Even though it is possible that the random action is equal to the agent's actual action, the other two aspects result in the observed training behavior.

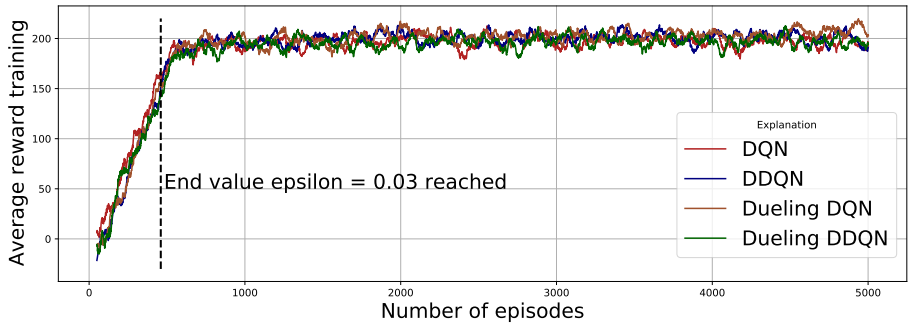

Figure 13 – Average reward Q-networks

Since the average reward of all algorithms is around 200 units, the criteria to validate the policy after reaching an average return of 195 units is exceeded rather often by each deep Q-network (Table 6). Each algorithm validates its current policy at least 2600 times during the training process. Although the average training reward starts oscillating around an average reward of 200 units after the first 1000 planning horizons, the best policy was found after about 4000 planning horizons. An exception here is the dueling DQN, which achieved the best policy after 2877 planning horizons already.

Table 6 – Training results Q-networks

Network	Number validation games	Games to find best policy
DQN	2891	4496
DDQN	3591	4550
Dueling DQN	3800	2877
Dueling DDQN	2647	3940

The optimal reward, as well as the optimal number of the average accepted orders, is neither reached by any of the algorithms for the validation and the test data (Table 7). Since some algorithms perform better in the validation data compared with the test data and vice versa (Table 8), the mean of both values is employed for the comparison, to get a more generalized evaluation of the algorithms.

Table 7 – Results of the different Q-networks in the validation and test data

	Average reward		Average accepted orders	
Network	**Validation data**	**Test data**	**Validation data**	**Test data**
DQN	205.023	204.462	3.62	3.611
DDQN	204.701	204.256	3.606	3.605
Dueling DQN	202.555	202.364	3.611	3.598
Dueling DDQN	204.043	204.485	3.671	3.608
Optimal solution SDP	205.532	205.835	3.658	3.661

According to the mean rewards, none of the networks decide based on an optimal acceptance policy. The DQN performs best within the training and validation data and misses the optimal average reward by only 0.941 units. The DDQN and the Dueling DDQN reach average results with a similar mean average reward. However, from these three networks the dueling DDQN accepts about 0.24 orders more in each planning horizon while generating a reward that is about 0.4785 units lower. Hereby, its policy accepted less profitable orders. The performance of the Dueling DQN seems to be worst. Although the number of accepted orders is in a similar range, the average reward is 1.8005 units lower than the mean average reward of the next best network.

Table 8 – Mean average reward and mean average accepted orders

Network	**Mean average reward**	**Mean accepted orders**
DQN	204.7425	3.6155
DDQN	204.4785	3.6055
Dueling DQN	202.4595	3.6045
Dueling DDQN	204.264	3.6395
Optimal solution SDP	205.6835	3.6595

Comparison of the Q-networks with pER

The training of the Q-networks with a pER is rather similar to the training of the other networks. At the beginning of the training, the average reward increases in conjunction with the decrease of the exploration probability. After the lower bound of the exploration probability is reached, the average return of all networks varies around the value of 200 units. However, the approach to the value of 200

units proceeds more slowly, after the exploration probability ϵ has assumed the end value $\epsilon_{end} = 0.03$.

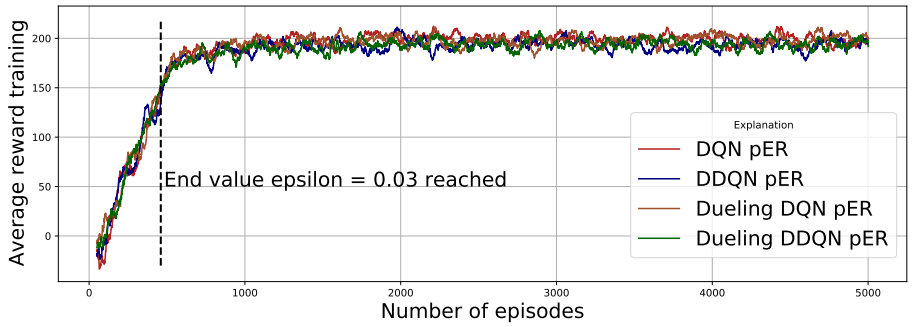

Figure 14 – Average reward Q-networks with pER

Compared with the networks without pER, all networks reach the validation criteria at a lower rate (Table 9). While the DQN and Dueling DQN achieve the criterion also about 3000 times, the policies of the DDQN and the Dueling DDQN are only validated about 1600 times. Also, the number of planning horizons to find the optimal policy is different. Hereby, the DQN and the DDQN need more than 4000 planning horizons and the Dueling DQN as well as the Dueling DDQN only need about 2800 planning horizons.

Table 9 – Training results Q-networks with pER

Network	Number validation games	Games to find best policy
DQN pER	3331	4211
DDQN pER	1603	4821
Dueling DQN pER	2810	2784
Dueling DDQN pER	1598	2847

Similar to the first experiment, none of the algorithms has found an optimal acceptance policy. However, all of them reach near optimal solutions (Table 10) and the DDQN, which generates the lowest average reward of all networks, misses the reward of an optimal policy by 4.515 units and 3.55 units in the validation and test data respectively. Hereby, no algorithm outperforms the others in both data sets. Therefore, the mean average over both data sets is considered (Table 11).

Table 10 – Results of the different Q-networks with pER in the validation and test data

	Average reward		Average accepted orders	
Network	**Validation data**	**Test data**	**Validation data**	**Test data**
DQN pER	202.685	203.609	3.613	3.608
DDQN pER	201.017	202.285	3.611	3.595
Dueling DQN pER	204.319	204.322	3.681	3.636
Dueling DDQN pER	203.222	202.948	3.633	3.614
Optimal solution SDP	205.532	205.835	3.658	3.661

Again, none of the networks has followed an optimal acceptance policy for both data sets. The Dueling DQN reached the highest mean average reward by missing the reward of an optimal solution by only 1.363 units and accepted the most orders in each planning horizon. The DQN and the Dueling DDQN performed a little worse according to both, the mean average reward and the mean number of accepted orders. The worst policy is found by the DDQN network, which mean average reward is the lowest of all eight networks and is 4.0325 units lower than the reward of an optimal solution.

Table 11 – Mean average reward and mean average accepted orders pER

Network	**Mean average reward**	**Mean accepted orders**
DQN	203.147	3.6105
DDQN	201.651	3.603
Dueling DQN	204.3205	3.6585
Dueling DDQN	203.085	3.6235
Optimal solution SDP	205.6835	3.6595

Comparison of the best deep Q-network with and without pER

Lastly, the performances of the best Q-network with and without pER are compared. The best policy of the Q-networks without pER is found by the DQN, and the Dueling DQN with pER follows the best policy of the other four networks. During the training phase, both networks act quite similar (Figure 15). However, the DQN achieves higher average rewards during the exploration phase. Afterwards, the average reward of both networks is oscillating around the value of 200 units.

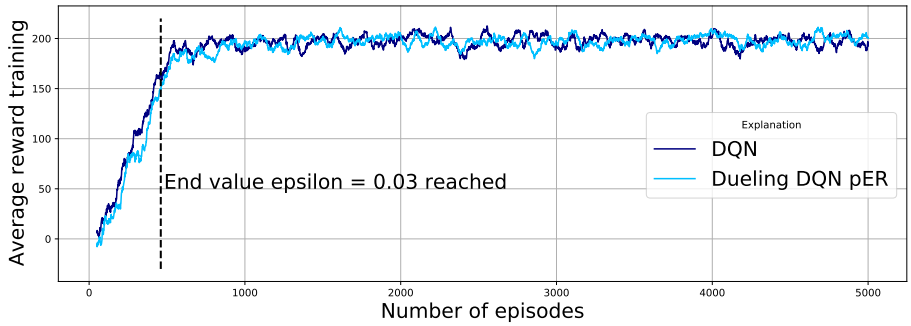

Figure 15 – Average reward best Q-network with and without pER

Even though the validation criteria is reached by both algorithms a similar amount of times, the Dueling DQN achieves the best policy after only 2784 planning horizons (Table 12). In contrast, the DQN needs almost the complete training phase to find its best. This shows that the usage of the prioritized experience replay, which is not the generating of a better overall policy, but to speed up the training process and find near optimal solutions fast.

Table 12 – Training results DQN and Dueling DQN with pER

Network	Number validation games	Games to find best policy
DQN	2891	4496
Dueling DQN pER	2810	2784

Since the DQN outperforms the Dueling DQN in both, the test and the validation data (Table 13), the comparison of the mean average reward and the mean number of accepted orders over both data sets is not necessary. Again, none of the algorithms was able to find an optimal acceptance policy in the data sets. Compared to the DQN, the Dueling DQN accepts a higher number of orders in each planning horizon. However, since the DQN achieves higher rewards, the accepted orders of the latter are more profitable and its policy is better to make decisions in the evaluation data.

Table 13 – Results DQN and Dueling DQN with pER in the validation and test data

	Average reward		Average accepted orders	
Network	**Validation data**	**Test data**	**Validation data**	**Test data**
DQN	205.023	204.462	3.62	3.611
Dueling DQN pER	204.319	204.322	3.681	3.636
Optimal solution SDP	205.532	205.835	3.658	3.661

5 Discussion and Prospect

RL and especially DRL are great opportunities for decision making under uncertainty and provide an alternative solution approach to DP, MC methods and TD algorithms. The DRL agents are able to find near optimal solutions without any knowledge about the SDP or its transition dynamics at the beginning of their training. For the considered stochastic order acceptance problem, all DRL agents found near optimal acceptance policies. Hereby, three of the four best policies are found by a network without the pER. This can be explained by two influences. Firstly, the scope of the OAP is simply designed and especially the resources with limited capacity in conjunction with high capacity demanding orders lead to acceptance policies where only few orders are accepted during each planning horizon and many orders are quickly identified as unprofitable. Secondly, the prioritizing of the training samples from the pER result in biases during the training. Since the OAP is rather simple, this biases prevent the agents from finding a better policy during the training. These biases have an impact on the updates of the network parameter such that they are subsequently subject to adjustments that are too large or too small. Too large updates result in large changes in the network structure and thus lead to an inaccurate adjustment. On the other hand, too small updates induce the network parameters to approach the optimality conditions only at a slow speed and thus cause a long training time. The policies of the DQN, DDQN, Dueling DQN and Dueling DDQN with and without pER all achieve similar average rewards within the OAP. Hereby, none of the algorithms seems to perform significantly better than the others in the evaluation data. Therefor, all of the algorithms are suitable for such a problem scope. However, since the validation and test data consist of limited data, a different set up of the evaluation data with additional compositions of orders and rewards might produce different results, as well as another set up of the training process.

Since none of the algorithms acted according to an optimal policy in both, the validation and the test data, several modifications of the DRL agents might produce

a better policy. The exploration exploitation trade-off (Chapter 3.2.1) during the training is essential for deep Q-learning agents. The presented ER (Chapter 3.3) also helps the algorithms to find near optimal policies fast. Since the algorithms with the pER do not outperform the algorithms with ER, the use of the pER is questionable. However, a pretty basic pER solution was used for the algorithms, which could be improved by integrating further elements. For example, a filter for unnecessary train samples could remove similar training examples with a high occurrence within the ER, or synthetic train data could be added, representing e.g. difficult decision situations for the agent.

Besides, different approaches might be able to further improve the algorithms. First of all, a data pre-processing step, which transform the input of the networks to another representation, might lead to a better learning behavior of the networks. For example, the inputs could be normalized to a value range between $[0, 1]$, such that the network's parameters are updated more accurately.

Secondly, adapting the training procedure might lead to improved results. For example, the criteria for the evaluation of the agent's current policy could be reduced to a lower or higher required average reward. Furthermore, considering even more than 50 planning horizons for the average reward might lead to a more generalized solution, taking more order types with different rewards into account. Moreover, another composition of validation and test data generates other evaluation results. Here, a larger size of the data sets includes more different order sequences of incoming orders with different rewards and produce a more general examination of the policies.

Furthermore, a longer training phase than 5000 planning horizons could also cause different solutions. Since many of the networks found their best policy shortly before terminating the training, a longer training phase would enable the algorithms to find even better policies.

Another possibility to find an optimal DRL agent is the use of policy search algorithms, such as *Actor-Critic Agents*. The set up of such algorithms is different. While the Q-networks are all value-based approaches, where the optimal policy is searched indirectly and the networks estimate optimal Q-values, which yield information about possible future rewards when entering a certain state, the policy search algorithms search directly for the best policy. Hereby, the *Actor-Critic Agents* exhibit better convergence properties, since they handle the exploration

exploitation trade-off implicitly. Since the *on-policy* methods improve their policy only in consideration of the current policy, the set up of the ER is not necessary neither. However, their implementation is typically more difficult and also, these algorithms face different problems during the training process. Hereby, two DNNs have to be trained, which outputs rely on each other and therefore each network is optimized according to the output of the other one. This mostly results in longer training time. Since the policy search algorithms are characterized by the ability to solve complex tasks, it is also questionable if they perform well on such a simple stochastic OAP.

Appendix

Algorithm 1 Deep Q-Learning

Initialize replay memory D
Initialize action-value function Q with random weights Θ
Initialize target action-value function $\hat{Q}$ with weights Θ^-
for planning horizon $m = 1, \ldots, M$ **do**
  for time step $t = 1, \ldots, T$ **do**
    if $exploration \ = \ True$ **then**
      Select random action a_t
    else
      Select $a_t = argmax\ Q(s_t, a_t; \Theta)$
    end if
    Execute action a_t and observe reward r_t and new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in replay buffer D
    if $prioritized\ experience\ replay \ = \ True$ **then**
      Sample prioritized batch of transitions (s_j, a_j, r_j, s_{j+1}) from D
      Compute $y_j = \begin{cases} r_j, & \text{if j=T} \\ r_j + \gamma \cdot \max \hat{Q}(s_{j+1}, a_{j+1}; \Theta^-), & \text{otherwise} \end{cases}$
      Compute $Loss = (w_j \cdot (y_j - Q(s_j, a_j; \Theta)))^2$, with important weights w_j
      Perform a gradient descent step based on the $Loss$ in respect to the network's parameters Θ
    else
      Sample random batch of transitions (s_j, a_j, r_j, s_{j+1}) from D
      Compute $y_j = \begin{cases} r_j, & \text{if j=T} \\ r_j + \gamma \cdot \max \hat{Q}(s_{j+1}, a_{j+1}; \Theta^-), & \text{otherwise} \end{cases}$
      Compute $Loss = (y_j - Q(s_j, a_j; \Theta))^2$
      Perform a gradient descent step based on the $Loss$ in respect to the network's parameters Θ
    end if
    Update the current state $s_t \leftarrow s_{t+1}$
  end for
  Reset state to the initial state $s_t \leftarrow s_0$
  Every C steps reset $\hat{Q} = Q$
end for

Algorithm 2 Double deep Q-Learning

Initialize replay memory D
Initialize action-value function Q with random weights Θ
Initialize target action-value function $\hat{Q}$ with weights Θ^-
for planning horizon $m = 1, \ldots, M$ **do**
  for time step $t = 1, \ldots, T$ **do**
    if *exploration* $=$ *True* **then**
      Select random action a_t
    else
      Select $a_t = argmax\ Q(s_t, a_t; \Theta)$
    end if
    Execute action a_t and observe reward r_t and new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in replay buffer D
    if *prioritized experience replay* $=$ *True* **then**
      Sample prioritized batch of transitions (s_j, a_j, r_j, s_{j+1}) from D
      Compute $y_j = \begin{cases} r_j & \text{, if j=T} \\ r_j + \gamma \cdot \hat{Q}(s_{j+1}, \underset{a}{max}\ Q(s_{j+1}, a_{j+1}; \Theta); \Theta^-) & \text{, otherwise} \end{cases}$
      Compute $Loss = (w_j \cdot (y_j - Q(s_j, a_j; \Theta)))^2$, with important weights w_j
      Perform a gradient descent step based on the $Loss$ in respect to the network's parameters Θ
    else
      Sample random batch of transitions (s_j, a_j, r_j, s_{j+1}) from D
      Compute $y_j = \begin{cases} r_j & \text{, if j=T} \\ r_j + \gamma \cdot \hat{Q}(s_{j+1}, \underset{a}{max}\ Q(s_{j+1}, a_{j+1}; \Theta); \Theta^-) & \text{, otherwise} \end{cases}$
      Compute $Loss = (y_j - Q(s_j, a_j; \Theta))^2$
      Perform a gradient descent step based on the $Loss$ in respect to the network's parameters Θ
    end if
    Update the current state $s_t \leftarrow s_{t+1}$
  end for
  Reset state to the initial state $s_t \leftarrow s_0$
  Every C steps reset $\hat{Q} = Q$
end for

Algorithm 3 Dueling deep Q-Learning

Initialize replay memory D
Initialize action-value function Q with random weights Θ of the shared network, random weights α from the stream which computes V and random weights β from the stream which computes A
Initialize target action-value function $\hat{Q}$ with weights Θ^- of the shared network, random weights α^- from the stream which computes $\hat{V}$ and random weights β^- from the stream which computes $\hat{A}$
for planning horizon $m = 1, \ldots, M$ **do**
  for time step $t = 1, \ldots, T$ **do**
    if *exploration* $=$ *True* **then**
      Select random action a_t
    else
      Select $a_t = argmax\ Q(s_t, a_t; \Theta, \alpha, \beta)$
    end if
    Execute action a_t and observe reward r_t and new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in replay buffer D
    if *prioritized experience replay* $=$ *True* **then**
      Sample prioritized batch of transitions (s_j, a_j, r_j, s_{j+1}) from D
      Compute $V(s_j; \Theta, \alpha), \hat{V}(s_{j+1}; \Theta^-, \alpha^-)$
      Compute $A(s_j, a_j; \Theta, \beta), \hat{A}(s_{j+1}, a_{j+1}; \Theta^-, \beta^-)$
      Compute $Q(s_j, a_j; \Theta, \alpha, \beta) = V(s_j; \theta, \alpha) + (A(s_j, a_j; \Theta, \beta) - \frac{1}{|A|} \sum_{a+1} A(s_j, a_j; \theta, \beta))$
      Compute $\hat{Q}(s_{j+1}, a_{j+1}; \Theta^-, \alpha^-, \beta^-) = \hat{V}(s_{j+1}; \theta^-, \alpha^-) + (\hat{A}(s_{j+1}, a_{j+1}; \theta^-, \beta^-) - \frac{1}{|\hat{A}|} \sum_{a+1} \hat{A}(s_{j+1}, a_{j+1}; \Theta^-, \beta^-))$
      Compute $y_j = \begin{cases} r_j & \text{, if j=T} \\ r_j + \gamma \cdot \max \hat{Q}(s_{j+1}, a_{j+1}; \Theta^-, \alpha^-, \beta^-) & \text{, otherwise} \end{cases}$
      Compute $Loss = (w_j \cdot (y_j - Q(s_j, a_j; \Theta, \alpha, \beta)))^2$, with important weights w_j
      Perform a gradient descent step based on the $Loss$ in respect to the network's parameters Θ, α, β
    else
      Sample random batch of transitions (s_j, a_j, r_j, s_{j+1}) from D
      Compute $V(s_j; \Theta, \alpha), \hat{V}(s_{j+1}; \Theta^-, \alpha^-)$
      Compute $A(s_j, a_j; \Theta, \beta), \hat{A}(s_{j+1}, a_{j+1}; \Theta^-, \beta^-)$
      Compute $Q(s_j, a_j; \Theta, \alpha, \beta) = V(s_j; \theta, \alpha) + (A(s_j, a_j; \Theta, \beta) - \frac{1}{|A|} \sum_{a+1} A(s_j, a_j; \theta, \beta))$
      Compute $\hat{Q}(s_{j+1}, a_{j+1}; \Theta^-, \alpha^-, \beta^-) = \hat{V}(s_{j+1}; \theta^-, \alpha^-) + (\hat{A}(s_{j+1}, a_{j+1}; \theta^-, \beta^-) - \frac{1}{|\hat{A}|} \sum_{a+1} \hat{A}(s_{j+1}, a_{j+1}; \Theta^-, \beta^-))$
      Compute $y_j = \begin{cases} r_j & \text{, if j=T} \\ r_j + \gamma \cdot \max \hat{Q}(s_{j+1}, a_{j+1}; \Theta^-, \alpha^-, \beta^-) & \text{, otherwise} \end{cases}$
      Compute $Loss = (y_j - Q(s_j, a_j; \Theta, \alpha, \beta))^2$
      Perform a gradient descent step based on the $Loss$ in respect to the network's parameters Θ, α, β
    end if
    Update the current state $s_t \leftarrow s_{t+1}$
  end for
  Reset state to the initial state $s_t \leftarrow s_0$
  Every C steps reset $\hat{Q} = Q$
end for

Algorithm 4 Dueling double deep Q-Learning

Initialize replay memory D
Initialize action-value function Q with random weights Θ of the shared network, random weights α from the stream which computes V and random weights β from the stream which computes A
Initialize target action-value function $\hat{Q}$ with weights Θ^- of the shared network, random weights α^- from the stream which computes $\hat{V}$ and random weights β^- from the stream which computes $\hat{A}$
for planning horizon $m = 1, \ldots, M$ **do**
  for time step $t = 1, \ldots, T$ **do**
    if *exploration* $=$ *True* **then**
      Select random action a_t
    else
      Select $a_t = argmax\ Q(s_t, a_t; \Theta, \alpha, \beta)$
    end if
    Execute action a_t and observe reward r_t and new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in replay buffer D
    if *prioritized experience replay* $=$ *True* **then**
      Sample prioritized batch of transitions (s_j, a_j, r_j, s_{j+1}) from D
      Compute $V(s_j; \Theta, \alpha), V(s_{j+1}; \Theta, \alpha), \hat{V}(s_{j+1}; \Theta^-, \alpha^-)$
      Compute $A(s_j, a_j; \Theta, \beta), A(s_{j+1}, a_{j+1}; \Theta, \beta)\hat{A}(s_{j+1}, a_{j+1}; \Theta^-, \beta^-)$
      Compute $Q(s_j, a_j; \Theta, \alpha, \beta) = V(s_j; \theta, \alpha) + (A(s_j, a_j; \Theta, \beta) - \frac{1}{|A|}\sum_{a+1} A(s_j, a_j; \theta, \beta))$
      Compute $Q(s_{j+1}, a_{j+1}; \Theta, \alpha, \beta) = V(s_{j+1}; \theta, \alpha) + (A(s_{j+1}, a_{j+1}; \Theta, \beta) - \frac{1}{|A|}\sum_{a+1} A(s_{j+1}, a_{j+1}; \theta, \beta))$
      Compute $\hat{Q}(s_{j+1}, a_{j+1}; \Theta^-, \alpha^-, \beta^-) = \hat{V}(s_{j+j}; \theta^-, \alpha^-) + (\hat{A}(s_{j+1}, a_{j+1}; \theta^-, \beta^-) - \frac{1}{|\hat{A}|}\sum_{a+1} \hat{A}(s_{j+1}, a_{j+1}; \Theta^-, \beta^-))$
      Compute $y_j = \begin{cases} r_j & \text{, if j=T} \\ r_j + \gamma \cdot \hat{Q}(s_{j+1}, \underset{a}{max}\ Q(s_{j+1}, a_{j+1}; \Theta, \alpha, \beta); \Theta^-, \alpha^-, \beta^-) & \text{, otherwise} \end{cases}$
      Compute $Loss = (w_j \cdot (y_j - Q(s_j, a_j; \Theta, \alpha, \beta)))^2$, with important weights w_j
      Perform a gradient descent step based on the *Loss* in respect to the network's parameters Θ, α, β
    else
      Sample random batch of transitions (s_j, a_j, r_j, s_{j+1}) from D
      Compute $V(s_j : \Theta, \alpha), V(s_{j+1}; \Theta, \alpha), \hat{V}(s_{j+1}; \Theta^-, \alpha^-)$
      Compute $A(s_j, a_j; \Theta, \beta), A(s_{j+1}, a_{j+1}; \Theta, \beta)\hat{A}(s_{j+1}, a_{j+1}; \Theta^-, \beta^-)$
      Compute $Q(s_j, a_j; \Theta, \alpha, \beta) = V(s_j; \theta, \alpha) + (A(s_j, a_j; \Theta, \beta) - \frac{1}{|A|}\sum_{a+1} A(s_j, a_j; \theta, \beta))$
      Compute $Q(s_{j+1}, a_{j+1} : \Theta, \alpha, \beta) = V(s_{j+1}; \theta, \alpha) + (A(s_{j+1}, a_{j+1}; \Theta, \beta) - \frac{1}{|A|}\sum_{a+1} A(s_{j+1}, a_{j+1}; \theta, \beta))$
      Compute $\hat{Q}(s_{j+1}, a_{j+1}; \Theta^-, \alpha^-, \beta^-) = \hat{V}(s_{j+j}; \theta^-, \alpha^-) + (\hat{A}(s_{j+1}, a_{j+1}; \theta^-, \beta^-) - \frac{1}{|\hat{A}|}\sum_{a+1} \hat{A}(s_{j+1}, a_{j+1}; \Theta^-, \beta^-))$
      Compute $y_j = \begin{cases} r_j & \text{, if j=T} \\ r_j + \gamma \cdot \hat{Q}(s_{j+1}, \underset{a}{max}\ Q(s_{j+1}, a_{j+1}; \Theta, \alpha, \beta); \Theta^-, \alpha^-, \beta^-) & \text{, otherwise} \end{cases}$
      Compute $Loss = (y_j - Q(s, a; \Theta, \alpha, \beta))^2$
      Perform a gradient descent step based on the *Loss* in respect to the network's parameters Θ, α, β
    end if
    Update the current state $s_t \leftarrow s_{t+1}$
  end for
  Reset state to initial state $s_t \leftarrow s_0$
  Every C steps, reset $\hat{Q} = Q$
end for

References

Joshua Achiam, Ethan Knight, and Pieter Abbeel. Towards characterizing divergence in deep q-learning. *arXiv preprint arXiv:1903.08894*, 2019.

Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.

Marc Brittain, Josh Bertram, Xuxi Yang, and Peng Wei. Prioritized sequence experience replay. *arXiv preprint arXiv:1905.12726*, 2019.

Lucian Buşoniu, Tim de Bruin, Domagoj Tolić, Jens Kober, and Ivana Palunko. Reinforcement learning for control: Performance, stability, and deep approximators. *Annual Reviews in Control*, 46:8–28, 2018.

Jie Chen, Bin Xin, Zhihong Peng, Lihua Dou, and Juan Zhang. Optimal contraction theorem for exploration–exploitation tradeoff in search and optimization. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 39(3):680–691, 2009.

Francois Chollet. *Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek*. MITP-Verlags GmbH & Co. KG, 2018.

Gal Dalal, Balázs Szörényi, Gugan Thoppe, and Shie Mannor. Finite sample analyses for td (0) with function approximation. *arXiv preprint arXiv:1704.01161*, 2017.

Zihan Ding, Yanhua Huang, Hang Yuan, and Hao Dong. Introduction to reinforcement learning. In *Deep Reinforcement Learning*, pages 47–123. Springer, 2020.

Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip HS Torr, Pushmeet Kohli, and Shimon Whiteson. Stabilising expe-

rience replay for deep multi-agent reinforcement learning. *arXiv preprint arXiv:1702.08887*, 2017.

Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. An introduction to deep reinforcement learning. *CoRR*, abs/1811.12560, 2018. URL `http://arxiv.org/abs/1811.12560`.

Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. In *International Conference on Machine Learning*, pages 2829–2838, 2016.

Shixiang Shane Gu, Timothy Lillicrap, Richard E Turner, Zoubin Ghahramani, Bernhard Schölkopf, and Sergey Levine. Interpolated policy gradient: Merging on-policy and off-policy gradient estimation for deep reinforcement learning. In *Advances in neural information processing systems*, pages 3846–3855, 2017.

Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *arXiv preprint arXiv:1710.02298*, 2017.

Gauri Kalnoor and Gowrishankar Subrahmanyam. A review on applications of markov decision process model and energy efficiency in wireless sensor networks. *Procedia Computer Science*, 167:2308–2317, 2020.

Mykel J Kochenderfer. *Decision making under uncertainty: theory and application*. MIT press, 2015.

Rudolf Kruse, Christian Borgelt, Frank Klawonn, Christian Moewes, Georg Ruß, Matthias Steinbrecher, et al. *Computational intelligence*. Springer, 2011.

Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

Sergey Levine, Aviral Kumar, George Tucker, and Justin Fu. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643*, 2020.

Ruishan Liu and James Zou. The effects of memory replay in reinforcement learning. In *2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 478–485. IEEE, 2018.

Yulong Lu and Jianfeng Lu. A universal approximation theorem of deep neural networks for expressing probability distributions. *Advances in Neural Information Processing Systems*, 33, 2020.

Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56, 2016.

Alaeddin Masadeh, Zhengdao Wang, and Ahmed E Kamal. Reinforcement learning exploration algorithms for energy harvesting communications systems. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2018.

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.

Thanh Thi Nguyen, Ngoc Duy Nguyen, and Saeid Nahavandi. Deep reinforcement learning for multiagent systems: A review of challenges, solutions, and applications. *IEEE transactions on cybernetics*, 2020.

Rui Nian, Jinfeng Liu, and Biao Huang. A review on reinforcement learning: Introduction and applications in industrial process control. *Computers & Chemical Engineering*, page 106886, 2020.

Brendan O'Donoghue, Remi Munos, Koray Kavukcuoglu, and Volodymyr Mnih. Combining policy gradient and q-learning. *arXiv preprint arXiv:1611.01626*, 2016.

Ioanid Rosu. The bellman principle of optimality. *Availiable at: http://faculty. chicagogsb. edu/ioanid. rosu/research/notes/bellman. pdf*, 2002.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

Andreas Scherer. *Neuronale Netze: Grundlagen und Anwendungen*. Springer-Verlag, 2013.

David Silver, Richard S Sutton, and Martin Müller. Temporal-difference search in computer go. *Machine learning*, 87(2):183–219, 2012.

David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. 2014.

Isaac J Sledge and José C Príncipe. Balancing exploration and exploitation in reinforcement learning using a value of information criterion. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2816–2820. IEEE, 2017.

Zbigniew A Styczynski, Krzysztof Rudion, and André Naumann. *Einführung in Expertensysteme: Grundlagen, Anwendungen und Beispiele aus der elektrischen Energieversorgung*. Springer-Verlag, 2017.

Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

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

Arryon D Tijsma, Madalina M Drugan, and Marco A Wiering. Comparing exploration strategies for q-learning in random stochastic mazes. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–8. IEEE, 2016.

Peter Vamplew, Richard Dazeley, and Cameron Foale. Softmax exploration strategies for multiobjective reinforcement learning. *Neurocomputing*, 263:74–86, 2017.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *arXiv preprint arXiv:1509.06461*, 2015.

Martijn Van Otterlo and Marco Wiering. Reinforcement learning and markov decision processes. In *Reinforcement Learning*, pages 3–42. Springer, 2012.

V X Wang. *Handbook of Research on Transdisciplinary Knowledge Generation*. IGI Global, 2019.

Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

Shangtong Zhang and Richard S Sutton. A deeper look at experience replay. *arXiv preprint arXiv:1712.01275*, 2017.

Yuchen Zhang, Jason D Lee, and Michael I Jordan. l1-regularized neural networks are improperly learnable in polynomial time. In *International Conference on Machine Learning*, pages 993–1001, 2016.