FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4 · Verteilte Systeme und Betriebssysteme

Anna Feiler

Using Reinforcement Learning for Decision Making in Quality-Aware Real-Time Scheduling

Masterarbeit im Fach Informatik

19. Juli 2021

Please cite as:
Anna Feiler, “Using Reinforcement Learning for Decision Making in Quality-Aware Real-Time Scheduling”, Master’s Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Dept. of Computer Science, July 2021.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme
Martensstr. 1 · 91058 Erlangen · Germany

Using Reinforcement Learning for Decision Making in Quality-Aware Real-Time Scheduling

Masterarbeit im Fach Informatik

vorgelegt von

Anna Feiler

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Tim Rheinfels, M.Sc.**
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat**
Beginn der Arbeit:	**30. Dezember 2020**
Abgabe der Arbeit:	**19. Juli 2021**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties. I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Anna Feiler)
Erlangen, 19. Juli 2021

ABSTRACT

Computation and developing time are both highly contested resources, especially in embedded real-time systems. Therefore, a system that can automatically reduce processor usage provides an invaluable advantage over either choosing a more powerful and consequently more expensive processor or optimizing code by hand. Various disturbances can cause a control loop which is implemented on a real-time operating system to become unbalanced and thus cause the Quality of Control (QoC) to decrease. Conversely, there are also situations where the system is close to equilibrium, and the controller needs little effort to keep it stable. Therefore, it may make more sense to save computation time by not running certain job activations.

A Linear Time-Variant System (LTVS), which is the basis of this work, emulates the temporal evolution of the system and the QoC. A clock-driven scheduler is augmented by a mechanism, which allows certain job activations to be omitted according to a strategy. This strategy is formalized into a policy and learned by the Deep Reinforcement Learning (DRL) algorithm. The implementation is realized and optimized using the two most common model-free algorithms, Q-learning and State–action–reward–state–action (SARSA), extended by an Artificial Neuronal Network (ANN). The Neuronal Network (NN) is used to determine whether and in which estimated state the actuation is necessary. With little effort, it can be adjusted how close the system will get to its minimum QoC. The evaluation shows that the DRL approach can adapt to different system behaviors and outperforms the Supervised Learning (SL) policy by Rheinfels on two simulated reference systems. To conclude, the DRL approach turned out to be a promising solution for the trade-off between QoC and utilization, especially for more complex systems with long-running tasks.

KURZFASSUNG

Sowohl Rechen- als auch Entwicklungseit sind besonders in eingebetteten Echtzeitsystemen eine hart umkämpfte Ressource. Systeme, welche die Prozessorlast automatisch reduzieren können, haben daher einen sehr großen Vorteil gegenüber herkömmlichen Systemen. Sollten diese Systeme nämlich ihre Leistungsgrenze erreichen, so kommt nur die Wahl eines leistungsfähigeren und folglich teureren Prozessors oder die Optimierung des Codes von Hand in Frage. Verschiedene Störungen können dazu führen, dass ein Regelkreis, der auf einem Echtzeit-Betriebssystem implementiert ist, aus dem Gleichgewicht gerät und somit zu einem Abfallen der Quality of Control (QoC) führt. Umgekehrt gibt es aber auch Situationen, in denen das System nahe am Gleichgewicht ist und der Regler wenig Aufwand benötigt, um dieses stabil zu halten. Oft kann es deshalb sinnvoller sein, sich die Berechnungszeit einzusparen, indem man bestimmte Job-Aktivierungen nicht ausführt.

Die Grundlage dieser Arbeit bildet ein Linear Time-Variant System (LTVS), welches die zeitliche Entwicklung des Systems und der Quality of Control (QoC) emuliert. Um bestimmte Job-Aktivierungen gemäß einer bestimmten Strategie auslassen zu können, wird das System um einen taktgesteuerten Scheduler erweitert. Diese Strategie kann durch eine Policy formalisiert und mit Hilfe des Deep Reinforcement Learning (DRL)-Algorithmus gelernt werden. Die Umsetzung des Algorithmus erfolgt durch die beiden gängigsten modellfreien Ansätze, nämlich Q-learning und State–action–reward–state–action (SARSA) in Kombination mit einem künstlich neuronalen Netz. Dieses wird verwendet, um zu bestimmen, ob und in welchem geschätzten Zustand die Regelung notwendig ist. Außerdem kann mit geringem Aufwand eingestellt werden, wie nahe das System seiner minimalen QoC kommen darf. Die Auswertung mehrerer simulierter Experimente hat ergeben, dass der Deep Reinforcement Learning (DRL)-Ansatz sich an unterschiedliche Systemverhalten anpassen kann und den Supervised Learning (SL)-Ansatz von Rheinfels auf zwei simulierten Referenzsystemen übertrifft. Zusammenfassend lässt sich sagen, dass sich der DRL-Ansatz als eine vielversprechende Lösung für den Kompromiss zwischen QoC und Auslastung erwiesen hat, insbesondere für komplexere Systeme mit lang laufenden Aufgaben.

CONTENTS

1 Introduction **1**

2 Notation **5**

3 Fundamentals **7**
3.1 Environment . . . 7
3.1.1 Linear Time-Variant System . . . 8
3.1.2 Quality of Control (QoC) . . . 9
3.1.3 Real-Time Operating System (RTOS) . . . 10
3.2 Artificial Neuronal Networks . . . 10
3.2.1 Perceptron . . . 10
3.2.2 Multi-Layer Perceptron . . . 11
3.2.3 Training . . . 12
3.2.4 Regression and Classification . . . 13
3.3 Reinforcement Learning . . . 14
3.3.1 Agent-Environment Interaction . . . 14
3.3.2 Markov Decision Process . . . 15

4 Problem Statement **17**

5 Deep Reinforcement Learning Policy **19**
5.1 Temporal Difference Learning . . . 19
5.1.1 SARSA . . . 20
5.1.2 Q-Learning . . . 21
5.2 Nonlinear State-Value Function Approximation . . . 22
5.3 Reward Function Modeling . . . 23

6 Evaluation **25**
6.1 Evaluation Metrics . . . 25
6.2 Optimization . . . 26
6.2.1 Target Network . . . 27
6.2.2 Experience Replay . . . 28
6.2.3 Loss Function . . . 29
6.2.4 Weight Initialization . . . 30
6.3 Parameter Assignment . . . 32
6.4 Network Architecture and Implementation Details . . . 33
6.5 Algorithms . . . 34

6.6 Reward Function Modification . . . 36
6.7 Interpreting the Evaluation Metrics . . . 36
6.7.1 Process and Measurement Noise . . . 37
6.7.2 Policies . . . 38

7 Related Work **39**

8 Conclusion and Outlook **41**

A Proofs and Analyses **43**
A.1 Covariance Matrix N_k . . . 43
A.2 Execution Times . . . 44

B System Models **47**
B.1 Inverted Pendulum . . . 48
B.2 Double Integrator . . . 48
B.3 Cost Calculation . . . 49

C Parameter Sets **51**
C.1 Parameters for the Target Network Evaluation . . . 51
C.2 Parameters for Experience Replay (ER) and Weight Evaluation . . . 52
C.3 Parameters for the Main Evaluation . . . 52

D Experiments **53**

Lists **59**
List of Acronyms . . . 59
List of Figures . . . 61
List of Tables . . . 63
List of Algorithms . . . 65
Bibliography . . . 67

1 INTRODUCTION

In the late 1950s, the term of *optimal control* was first used to describe the problem of designing a controller to minimize a measure of a dynamical system's behavior over time [SB18, p. 16]. When in 1950 Richard Bellman and others tried solving this task, the approach of Reinforcement Learning (RL) was born. An important extension called Deep Reinforcement Learning (DRL) gained popularity in 2013 because it was able to learn to play Atari games [Mni+13]. In the meantime, it has proven to be applicable to many other fields as well.

Compared to the other algorithms in the field of Artificial Intelligence, like Supervised Learning (SL), RL has the following advantages [SB18, p. 12]:

- No need for large data sets
- Ability to generate entirely new solutions that humans would never have considered
- Easier adaptability to new circumstances.

The main task of this thesis is to save execution time and cost by minimizing the actuation for a control system implemented on a Real-Time Operating System (RTOS), while maintaining its stability. This problem is very well suited for the DRL approach. The changing state of the controller can be used as input for a Neuronal Network (NN), which is able to continuously learn whether to activate a job or not for a given state. Before the system is reaching a point of instability, the algorithm has to trigger the actuation.

Figure 1.1 shows how a policy influences the behavior of a Real-Time Control System (RTCS). In general, the design of a RTCS includes a digital control loop as well as an internal clock and scheduler to execute a control task. This task's purpose is to bring the system close to equilibrium. Performing this task as often as needed is vital for the system stability. A policy can model how many activations are required in order to keep the system's stable.

Figure 1.1 broadly outlines the structure of the specific system, which is considered in this thesis. This particular system contains a physical control loop which consists of a plant, a sensor, a controller and an actuator. The plant describes its physical portion, which should be balanced by the movement of the actuator. To achieve this balance, the controller gets information about the physical state from the sensor and calculates a suitable reaction, which the actuator should perform. A second loop is formed via the RTOS scheduler by a policy interacting with the controller. A policy decides whether to activate the controller depending on the estimated system state and Quality of Control (QoC) [Gau+18, p. 1]. There are three different events *measure*, *compute* and *actuate*, which impact the

state of the physical control system, resulting in a new state and QoC. These two values influence the decision of the policy whether to activate all three events or leaving out the *actuation*, which saves computing time.

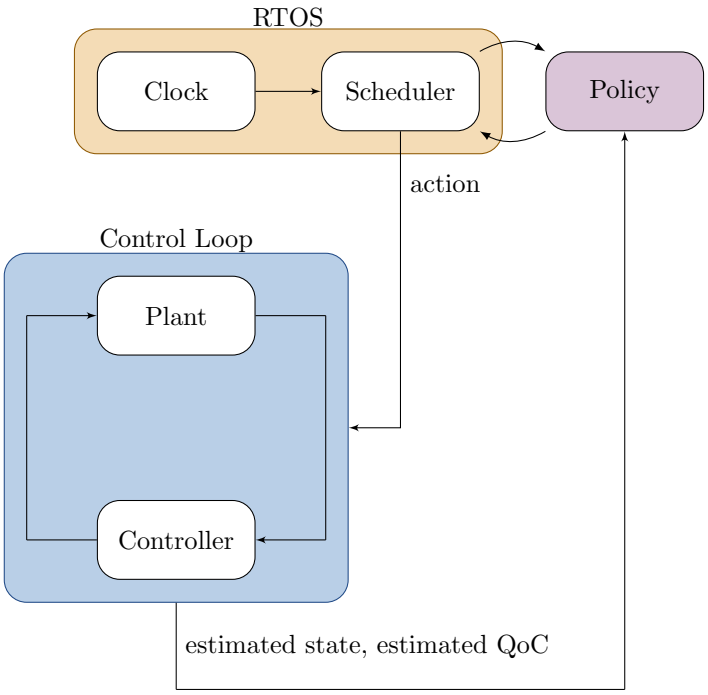

Figure 1.1 – An overview of the given environment, including a simplified illustration of the control loop and the RTOS. The policy interacts with the RTOS based on the estimated state and QoC.

The QoC is a physical quantity used to express the performance of the controller. The system models used in this thesis have a minimum QoC. Multiply sources of disturbance can effect the system's QoC. To evaluate the soundness of the trained system, measurement noise and external disturbances will be simulated. If the controller compensates the disturbance whenever possible, a very high utilization rate is obtained, which isn't needed to hold the QoC above its minimum value.

The trade-off between utilization and QoC will be defined by a policy, e.g., $\text{QoC} > \text{QoC}_{\text{min}}$. This policy can be learned through different approaches. Rheinfels implemented a SL policy, which can achieve this goal using multiple restrictive assumptions [Rhe19, p. 53]. As previously mentioned, SL requires a large labeled data set in complex environments and needs to be retrained for changing environments. RL on the other hand is learning an optimal policy by trial and error and can therefore adapt to any kind of environment change, with no need of labeled data. To be able to interact in continuous state spaces, as provided by the RTCS, the classic RL approach can be extended by a NN. This technique, called DRL is able to solve very complex problems, but can lead to unstable learning. To overcome this instability, several optimizations can be used resulting in faster convergence and therefore faster learning speed. To learn the defined policy, the estimated state and QoC is given as

input to the NN. The prediction whether to actuate or not is then provided via the two output nodes by choosing the maximum value. These so called action-value functions are calculated by the RL algorithm for a given estimated state and action and learned by the NN through gradient-descent. To keep the execution time that is needed to predict the favored action within acceptable limits, the design of the NN must be considered.

This thesis starts with a short overview of definitions and notations in Chapter 2, followed by the system model, QoC and RTOS forming the environment as given in Chapter 3. In addition, Chapter 3 contains a description of the two main components of the DRL, the NN and the classic RL approach. Before introducing the implemented DRL policy in Chapter 5, the problem statement will be refined in Chapter 4 using the information provided up to that point. The evaluation following Chapter 6 compares optimizations and the outcome of several experiments measured by defined metrics. Finally, Chapter 8 summarizes the evaluation and discusses its influence on the overall execution time.

2 NOTATION

In this chapter, definitions and notations of quantities used in the thesis are introduced. Some definitions are adopted from [Rhe19] to establish connections when needed.

Vector and Matrix

Matrices are given in upper case letters, e.g., A and B, with their transposed forms being A^T. Special matrices are defined when their definition is required, making an exception for the identity matrix assigned as I. Just like scalars, vectors are written in lower case letters. A matrix $A \in \mathbb{R}^{n\times n}$ is said to be symmetric and positive definite (s.p.d.) iff ([Bos07, p. 259]):

$$x^T A x \;>\; 0, \quad \forall x \in \mathbb{R}^n \backslash \{0\} \tag{2.1}$$

and to be symmetric and positive semi-definite (sp.s.d.), if

$$x^T A x \;\geq\; 0, \quad \forall x \in \mathbb{R}^n. \tag{2.2}$$

$\mathbb{1}$ denotes a column-vector comprised of ones, i.e. $[1 \quad 1 \quad ... \quad 1]^T$.
For two matrices A and B of the same dimension $m \times n$, the Hadamard product [HJ12, p. 477] can be written as :

$$(A \circ B) \;=\; (a_{ij} \cdot b_{ij}) \;=\; \begin{pmatrix} a_{11} \cdot b_{11} & \cdots & a_{1n} \cdot b_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} \cdot b_{m1} & \cdots & a_{mn} \cdot b_{mn} \end{pmatrix} \;\in\; \mathbb{R}^{m\times n}. \tag{2.3}$$

Functions

The total derivative of some function $f(x)$ with respect to x is given by $f'(x)$. Furthermore, the gradient of a function $f(x)$, given as the vector of the partial derivatives $\frac{\partial}{\partial x_i}$ with respect to the n dimensional vector x is donated as:

$$\nabla_x f(x) \;:=\; [\frac{\partial}{\partial x_1} \quad \frac{\partial}{\partial x_2} \quad ... \quad \frac{\partial}{\partial x_n}]^T. \tag{2.4}$$

Random Distribution

The probability density of a value x is denoted as Pr{x}, while the expectation operator is written by using $\mathbb{E}$ with its argument in curly braces.

Two forms of the normal distribution are used in this thesis: the first one is the univariate normal distribution $\mathcal{N}(\mu,\ \sigma^2)$ with $\mu\ \in\ \mathbb{R}^d$ as mean value and its variance $\sigma^2\ >\ 0$ given by Equation (2.5). The second one is the d-dimensional multivariate normal distribution $\mathcal{N}(\mu,\ \Sigma)$ with mean vector $\mu \in \mathbb{R}$ and s.p.d. covariance matrix $\Sigma \in \mathbb{R}^{dxd}$, as described in Equation (2.6) ([Dek+05b, p. 64]).

$$x\ \sim\ \mathcal{N}(\mu,\ \sigma^2)\ \Leftrightarrow\ Pr\{x\}\ =\ \frac{1}{\sigma\sqrt{2\pi}}e^{(-x-\mu^2)/2\sigma^2} \tag{2.5}$$

$$x\ \sim\ \mathcal{N}(\mu,\ \Sigma)\ \Leftrightarrow\ Pr\{x\}\ =\ \frac{1}{\sqrt{(2\pi)^d det\Sigma}}e^{-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)}. \tag{2.6}$$

The real uniform distibution $\mathcal{U}[a,\ b]$ with given interval $[a,\ b]$ is defined as ([Dek+05b, p. 60]):

$$x\ \sim\ \mathcal{U}(a,\ b)\ \Leftrightarrow\ Pr\{x\}\ =\ \begin{cases}\frac{1}{b-a} & a\ \leq\ x\ \leq\ b\\ 0 & \text{otherwise.}\end{cases} \tag{2.7}$$

Letters having a hat, e.g., $\hat{x}$, indicate that the value can not be determined exactly and is therefore estimated. Real and natural numbers are given as $\mathbb{R}$ and $\mathbb{N}$ respectively, while other sets are written in gothic capital letters.

3 FUNDAMENTALS

First, this chapter gives an overview of the shared environment in Section 3.1. A short explanation of the concepts and structures used in NNs is provided in Section 3.2, followed by a comparison of the two main problem types a NN can solve. Afterward, the second component RL, and its environment and proof, given by the Markov Decision Processes (MDP), are introduced in Section 3.3.

3.1 Environment

This section outlines the *environment* the agent of the RL approach interacts with. An overview of the environment and the influence of the single components are provided in Figure 3.1. A detailed description is presented in this section. The environment consists of the *control loop* formalized by a Linear Time-Variant System (LTVS) in Section 3.1.1. The QoC is used to measure how well the control loop meets its specification, with its corresponding formula introduced in Section 3.1.2. The last environment component, the RTOS, which can affect the system performance by executing control tasks, is described in Section 3.1.3.

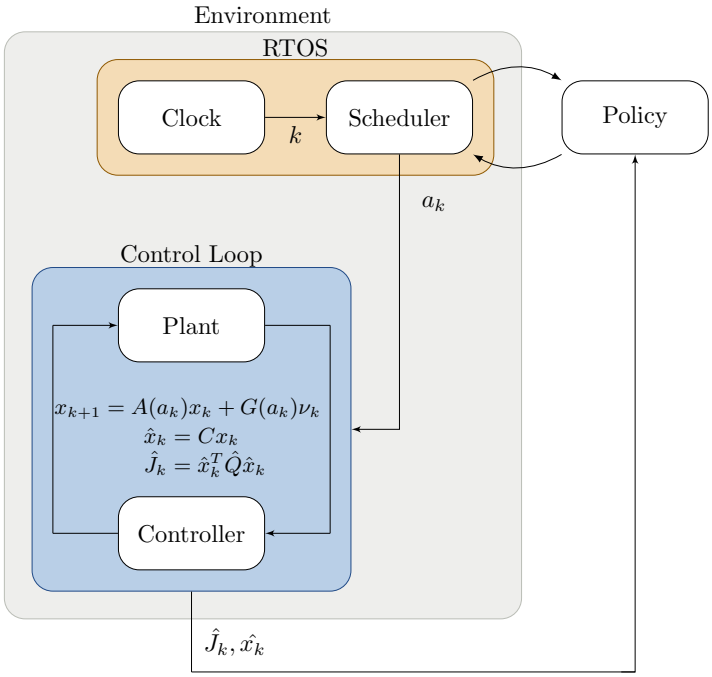

Figure 3.1 – The environment consists of the control loop and the RTOS. In this figure, simplified versions of both are illustrated. The action a is influencing the system at a given time step k and a policy provides the best action.

3.1.1 Linear Time-Variant System

The linear time-variant system is simplified on the premise that execution times are deterministic. It is derived from the Linear Impulsive System (LIS) used in Rheinfels' master thesis [Rhe19, p. 10]. At first, time step t_k and span are defined, given by the following formulas:

$$t_k \; < \; t_l \iff k \; < \; l \qquad t \in (t_0,\; T) \qquad \forall l,\; k \in \mathbb{N}_0 \tag{3.1}$$

$$t_k \; \geq \; t_0 \; . \tag{3.2}$$

Then the LTVS can be set up as:

$$x_{k+1} \; = \; A(a_k)x_k + G(a_k)\nu_k. \tag{3.3}$$

The equation describes the system's dynamics in time, where x_{k+1} is the consecutive state at time step $k+1$. The matrices $A(i)$ and $G(i)$ are chosen from finite sets of $\mathfrak{A}$ and $\mathfrak{G}$, which are assumed to be given and defined for the evaluated systems presented in Appendix B. With attention to the system that needs to comply with a given QoC specification $\hat{\Omega}$, the parameter a_k is chosen. In Rheinfels' master thesis, the events *measure*, *compute*, and *actuate* are considered separately but combined in this thesis to one matrix $A(i)$. This simplification was done because varying execution times are not within the scope of this work [Rhe19, p. 15]. Defining both matrices as the following, given the restriction for $G(i)G(i)^T$ to be symmetric and positive semi-definite (sp.s.d.):

$$A(i) \; \in \; \mathfrak{A} \; , \; i \in \{0,\; 1\} \tag{3.4}$$

$$G(i) \; \in \; \mathfrak{G} \; , \; i \in \{0,\; 1\}. \tag{3.5}$$

More details on this topic can be found in Appendix B and the previously mentioned master thesis [Rhe19]. Given this system description, a_k needs to be chosen correctly to keep the control loop close to equilibrium. It is understood that picking $a_k = 1$ in every time step k achieves this goal. If the system does not regulate according to its specifications, the controller is already broken beforehand. Although this may be true, choosing a_k to be one in every time step k is not necessary to stabilize the system. Therefore, a policy π can be defined, able to decide which a_k to take in a given time step k:

$$a_k = \pi(\hat{x}_k, \hat{J}_k), \tag{3.6}$$

depending on the measured state $\hat{x_k}$ and the measured QoC. The usage and form of the QoC will be defined in Section 3.1.2. Only a part of the state x_k is measurable, so this results in another equation with a given matrix C:

$$\hat{x}_k = Cx_k = \begin{bmatrix} 0 & I & 0 & 0 \\ 0 & 0 & I & 0 \\ 0 & 0 & 0 & I \end{bmatrix} x_k \ , \ C \in \mathbb{R}^{nxm}. \tag{3.7}$$

The last unknown variable in Equation (3.3) v_k encodes stochastic measurement and process noise. It offers the possibility to include measurement noise and external disturbances to the formula. The noise is assumed to have zero mean and be white, resulting in:

$$\mathbb{E}\{v_k\} = 0 \tag{3.8}$$

$$\mathbb{E}\{v_k v_l^T\} = \begin{cases} I & k = l \\ 0 & else. \end{cases} \tag{3.9}$$

The values are drawn from a Gaussian distribution [Rhe19, p. 12]:

$$v_k \sim \mathcal{N}(0, I). \tag{3.10}$$

Furthermore, it must be ensured that $G(i)G(i)^T$ being sp.s.d. hold so that the covariance matrix of $G(i)v_k$ becomes sp.s.d..

3.1.2 Quality of Control (QoC)

The QoC is a quadratic control error function that can be used to assess how closely a control loop meets its specification [Rhe19, p. 17]:

$$\hat{J}_k = \hat{x}_k^T \hat{Q} \hat{x}_k. \tag{3.11}$$

$\hat{Q}$ is a given s.p.d. weight matrix, defined in Appendix B. Its derivation can be retraced in [Gau+18, p. 1]. An undesirable control loop behavior results in higher costs and indicates how close the system is to its equilibrium. In the rest of the thesis, the mathematical definition of the QoC is given by the following formula, normalizing the range of the value to lay between 0 and 1 [Rhe19, p. 17]:

$$\hat{\Omega}_k := \frac{1}{\hat{J}_k + 1}. \tag{3.12}$$

Every system model used in this work has a predefined lower limit called minimum QoC Ω_{min}, which can be looked up in Appendix B. Falling below this lower bound results in asymptotical instability and must be avoided.

3.1.3 Real-Time Operating System (RTOS)

The other part of the environment is the RTOS, which implements the basic operations of this system *measuring*, *computing*, and *actuating*. The system uses a clock-driven scheduler, as indicated in Figure 3.1. It is assumed that control tasks are scheduled periodically with deterministic execution times. These control tasks T_i are given as:

$$T_i \;:=\; (p,\ \varphi_i,\ D_i,\ e_{mean}). \tag{3.13}$$

This formula defines the task as a quadruple of φ_i, period p, relative deadline D_i, and execution time e_{mean}. To simplify matters, there are no execution intervals, which means that in contrast to [Rhe19, p. 16], execution times have to be assumed constant. Concrete values for those variables are provided in Table B.1, which can be found in Appendix B.

3.2 Artificial Neuronal Networks

Artificial Neuronal Network (ANN) — or short only NN — is a technique or algorithm that tries to mimic the function of a human brain. In a highly simplified way, the human brain can be viewed as millions of interconnected neurons, which have the ability to extract and link information. A *perceptron*, an abstract model of a single neuron, is able to imitate this functionality[1]. Combining them into a multi-layered network, which means connecting them in series, results in an algorithm powerful enough to solve complex problems.

3.2.1 Perceptron

The history of *Artificial Intelligence* started in the 1940s and had a breakthrough with the introduction of the perceptron by Rosenblatt [Ros58, p. 389]. These perceptrons are mathematical functions that collect and classify information according to their specific architecture. They consist of multiple inputs x_k and only one output y_j. The quantities a_j are known as activations and combine the inputs x_k, weights w_k, and the bias b to a linear function as shown in Equation (3.14). Each of them is then transformed using a non-linear activation function $\sigma(\cdot)$ [Bis06, p. 227].

$$a_j \;=\; \left(\sum_{k=1}^{n} w_k x_k + b\right) \tag{3.14}$$

$$y_j \;=\; \sigma(a_j). \tag{3.15}$$

The reason for using an activation function lies in the concept of *non-linearity*. In Equation (3.14), the input vector and the quantities show a linear relationship. When adding an activation function, which is non-linear, the output will be non-linear, too. The Leaky Rectified Linear Unit (Leaky ReLU) [MHN13], an optimized ReLU activation function [GBB11], solves the problem of dying neurons when the gradient is less or equal to zero. The linear activation function and the Leaky ReLU activation function are defined as:

$$\sigma_l(a_j) \;:=\; a_j \tag{3.16}$$

$$\sigma_r(a_j) \;:=\; \begin{cases} 0.01a_j & a_j \;<\; 0 \\ a_j & a_j \;\geq\; 0. \end{cases} \tag{3.17}$$

[1] Perceptron and neuron are treated equally in this thesis.

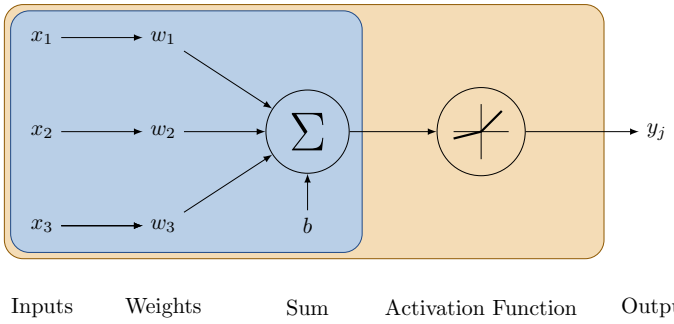

Figure 3.2 – A graphical representation of a perceptron with three inputs x_1, x_2, x_3, weights w_1, w_2, w_3 and bias b forming the linear regression Equation (3.14). The activation function σ is applied to the linear combination, resulting in the output y_j.

Combining all these information results in Figure 3.2. Many perceptrons, which are connected in series are called Multi-Layer Perceptron (MLP)[2], forming a generic model that can adapt to training data [Wer74, p. 12].

3.2.2 Multi-Layer Perceptron

To solve more complex problems, many perceptrons are combined, forming a MLP, which performs a series of functional transformations [Bis06, p. 227]. More formally, a MLP can be described as a model $\hat{y} = f^*(x, w)$, approximating an existing function $y = f(x)$ [GBC16, p. 164]. w consists of the parameters, which are learned during training. In a NN they are represented as weights and bias values. To give an overview of the layout, the structure of a MLP is shortly explained in the following.

First, a method how to count the number of layers in a NN is explained. Many authors, among others Russell Reed argue that the input layer has no activation function and should not be part of the total number of layers [RM98, p. 32]. Thus, the network given in Figure 6.1 has three layers consisting of two *hidden layers* and one *output layer*. Such a NN with more than one hidden layer is called a *Deep Neuronal Network* and is able to perform *deep learning* [Dec86, p. 180]. A standard MLP's input layer consists of nodes, that bring initial data into the system, and are often referred to as input units because they do not perform a transformation. The number of neurons per hidden layer and output layer can contain more or fewer neurons, depending on the problem and its complexity. A series of fully connected layers, which means each output is used as input for the next layer, is called a *Fully Connected Neural Network*, illustrated in Figure 3.3. In mathematical terms, a fully connected layer is a multivariate function given as $\mathbb{R}^{n_k} \mapsto \mathbb{R}^{n_j}$. Using this type of network results in numerous links and consequently in multiple weights, which must be learned during training [Bis06, p. 368].

[2] In literature, MLP is often used ambiguously to any Feed-Forward Network (FFN).

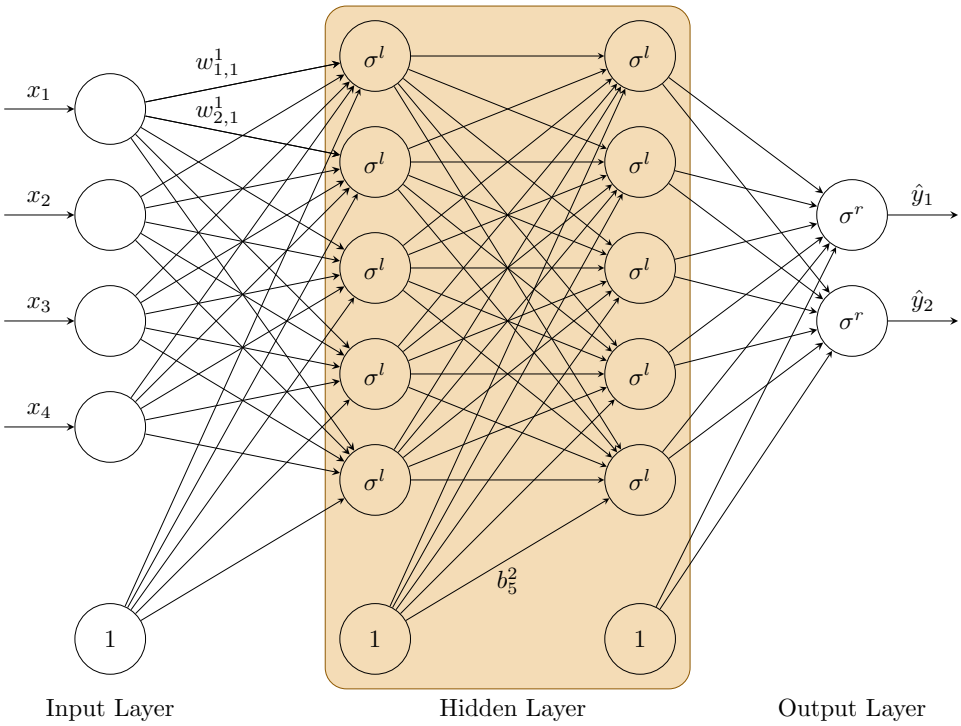

Figure 3.3 – A fully connected MLP network with input, output and two hidden layers. Weights $w^l_{k,j}$ are illustrating the affine transformations between the connected neurons. These neurons are drawn as nodes containing σ if there is an activation function used. Nodes containing 1 are used to symbolize the bias to be added in every layer except the input.

3.2.3 Training

Giving a certain data set, learning to map inputs to outputs can be defined as *training*. This process involves finding a set of weights for the NN, that are good enough to solve a specific problem. To determine the learning process properly, variables need to be introduced. The notations are based on [Bis06], leading to the representations explained in the following paragraph.

Each layer $l \in (1,\ 2,\ ...,\ n_l)$, has an assigned matrix of weights W^l and a vector of bias values b^l. b^l_j represents the bias of neuron j in layer l. Accordingly, $w^l_{k,j}$ is the weight of the neuron j in layer l for the activation a^{l-1}_k of the neuron in layer $l-1$. An illustration is given in Figure 3.3. The activation of a single neuron and an entire layer is defined in Equation (3.18) and Equation (3.19), respectively [Bis06, p. 229]:

$$a^l_j = \sigma\left(\sum_{k=1}^{n} w^l_{k,j} a^{l-1}_k + b^l_j\right) \tag{3.18}$$

$$a^l = \sigma\left(W^l a^{l-1} + b^l\right). \tag{3.19}$$

Propagating the calculated activation a^l feed-forward through every layer is called *forward propagation*. The observed outputs $\hat{Y}$ can be compared to the expected values Y, using for example the

Mean Squared Error (MSE) or Mean Absolute Error (MAE), given in the following:

$$L_{MSE}(Y, \hat{Y}) = \frac{1}{n}\sum_{j=1}^{n}(y_j - \hat{y}_j)^2 \tag{3.20}$$

$$L_{MAE}(Y, \hat{Y}) = \frac{1}{n}\sum_{j=1}^{n}|y_j - \hat{y}_j|, \tag{3.21}$$

where

$$\hat{Y} = [\hat{y}_1 \quad \hat{y}_2 \quad \dots \quad \hat{y}_n]^T. \tag{3.22}$$

The actual training step is to minimize this loss function by using an optimization algorithm to find a local minimum. One of the most common algorithms is the *gradient descend* method, which updates the weights and bias thanks to the gradient. Its form can be seen in the following [Bis06, p. 240]:

$$w^l_{k,j} \leftarrow w^l_{k,j} - \alpha\frac{\partial L(Y, \hat{Y})}{\partial w^l_{k,j}}. \tag{3.23}$$

The learning rate $\alpha \in (0, 1]$ determines the gradient's step size, where the gradient can be calculated using the *backpropagation* algorithm. To simplify the notation, the *Hadamard product* can be used, resulting in the following calculation for the output δ^{n_l} and hidden layers error δ^l [Nie18, pp. 45–47]:

$$\delta^{n_l} = \nabla_a L \circ \sigma'(a^{n_l}) \tag{3.24}$$

$$\delta^l = \delta^{l+1}W^{l+1} \circ \sigma'(a^l). \tag{3.25}$$

Given these formulas, the gradient for every weight can be derived by:

$$\frac{\partial L(Y, \hat{Y})}{\partial w^l_{k,j}} = \delta^l_j a^{l-1}_k. \tag{3.26}$$

The NN is trained by computing the forward propagation followed by the loss calculation and backpropagation. After the gradient descent step is applied, the next iteration starts. This process continues until the loss value is small enough or the maximum number of iterations is reached.

3.2.4 Regression and Classification

Machine learning aims to automate the learning process so that the task performs in the best possible way. Whether the output Y is discrete or continuous, one distinguishes two types of learning problems: the *regression* problems and the *classification* problem. Therefore, the decision of which approach to take depends on the kind of data and the desired kind of output.

The output variable is continuous or numerical for regression, and categorical or discrete for classification [Bis06, p. 10]. As seen in Figure 3.4 classification tries to find a function to divide the data set into classes. It can either be a binary classification problem, as shown in Figure 3.4, or a multi-class problem. Classification is mostly a supervised learning concept because often labeled data sets are used. However, there are application areas where reinforcement learning was applied successfully, for example, in object detection [PS18] and even image classification [HB20].

Regression, on the other hand, can be equivalently found in SL and RL. The output is a real

value, which can be either nature or a floating-point value. It is used to find the best fit and predicting the outcome accurately, seen in Figure 3.4. The algorithm can be divided into linear and non-linear regression, depending on the linearity of the resulting model.

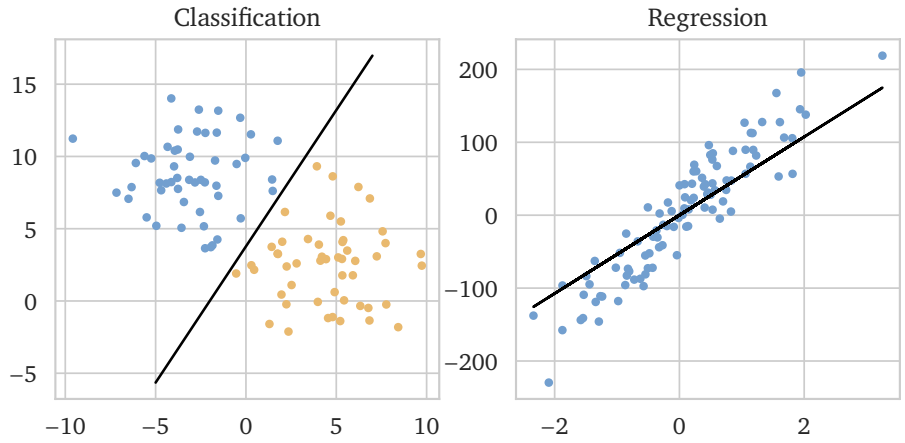

Figure 3.4 – Comparison of classification and regression problems. Dividing the data set into classes using classification can be seen in the left figure. Fitting a line through the data points in regression is provided in the right figure.

3.3 Reinforcement Learning

RL is a branch of machine learning, where an agent tries to solve a task by trial and error. It explores the environment whose dynamics are unknown to find an optimal chain of actions. By choosing an action, the agent can change the state of the environment, and receives immediate feedback through a reward. The task of an RL agent is to maximize the expected sum of rewards, which can be achieved by following an optimal policy. A policy is defined as a mapping from state to action. The main difference between RL and other machine learning methods like SL or Unsupervised Learning (USL) is that it does not depend on data acquisition. Furthermore, it is defined in terms of optimal control of an MDP. This chapter introduces the RL approach and is based on [SB18].

3.3.1 Agent-Environment Interaction

RL is a part of machine learning that is focused on learning through interaction [SB18, p. 25]. While interacting with the environment, an agent receives immediate feedback based on his chosen action. Because the only information an agent can get is the current state and the next state gathered through his transition, most of the environment is unknown. So, the agent is searching for an optimal path through the environment by trial and error. This interaction is illustrated in Figure 3.5.

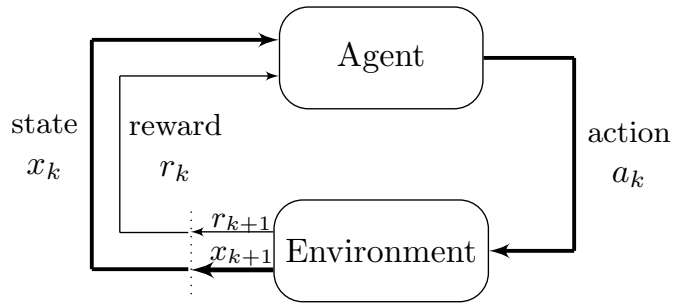

Figure 3.5 – Interaction of the agent with the environment [SB18, p. 48].

To describe this procedure more precisely, the agent interacts with the environment at each time step k. At each time step k the agent can select an *action* $a_k \in \mathfrak{A}$ from a set of *actions* available in state x_k. These states $x_k \in \mathfrak{X}$ are the representation of the environment. As a consequence of the chosen action a_k, the agent receives a *reward* $r_{k+1} \in \mathfrak{R}$ and continues to the next state x_{k+1} [SB18, p. 68]. When the agent approaches an optimal behavior, the process is finished.

3.3.2 Markov Decision Process

The RL environment interface can be formally described as a MDP [SB18, p. 47]. In a finite MDP, the sets of $\mathfrak{A}$, $\mathfrak{X}$, and $\mathfrak{R}$ all have a limited number of elements. All states in MDP have the *Markov property*, referring to the fact that the future state x_{k+1} only depends on the current state x_k and action a_k [SB18, p. 48]:

$$Pr\{x_{k+1},\ |x_k,\ x_{k-1},\ ...,\ x_0,\ a_k,\ a_{k-1},\ ...,\ a_0\} \ = \ Pr\{x_{k+1}|x_k,\ a_k\}. \tag{3.27}$$

The main problem of MDPs is to find a policy π for the agent. This so-called *agent's policy* maps a state x_k to an action a_k and ensures the *Markov property*. The goal is now to find a policy π, which maximizes the expected sum of rewards r_{k+1} from every state x_k. The commutative *discount return* G_k is then given as [SB18, p. 54].:

$$G_k \ := \ r_{k+1} + \gamma r_{k+2} + \gamma^2 r_{k+3} + ... \ = \ \sum_{i=0}^{\infty} \gamma^k r_{k+i+1}. \tag{3.28}$$

where $\gamma \in [0,\ 1)$ denotes the *discount factor* increasingly attenuated the rewards lying in the future. Choosing the *discount factor* to be zero results in the agent only maximizing the immediate reward, whereas setting it close to one, the agent tries to focus more on future rewards. After defining the available overall reward, the agent needs to know how good it is to be in a particular state x_k. This way of measuring can be done by a *state value* $V_\pi(x)$ or *state-action value* $Q_\pi(x,\ a)$ function. The latter is defined as [SB18, p. 58]:

$$Q_\pi(x,\ a) \ := \ \mathbb{E}_\pi\ \{G_k|x_k = x,\ a_k = a\} \ = \ \mathbb{E}_\pi\ \left\{\sum_{i=0}^{\infty} \gamma^k r_{k+i+1}|x_k = x,\ a_k = a\right\}. \tag{3.29}$$

It calculates the expected sum of rewards while taking action a_k in state x_k and by following policy π. A policy π^* is called optimal, if its expected return is better or equal than the other policies. The resulting optimal *state-action value* function can then be written as Equation (3.30) and expanded

to Equation (3.31):

$$Q_*(x,\ a) \ := \ \max_{\pi} \ Q_\pi(x,\ a) \tag{3.30}$$

$$\begin{aligned} Q_*(x,\ a) \ &:= \ \mathbb{E}\ \left\{ r_{k+1} + \gamma \ \max_{a \in \mathfrak{A}} \ Q_*(x_{k+1},\ a) | x_k = x,\ a_k = a \right\} \\ &:= \ \sum_{r_{k+1} \in \mathfrak{R}} \ Pr\ \left\{ x_{k+1},\ r_{k+1} | x_k,\ a_k)(r_{k+1} + \gamma \ \max_{a \in \mathfrak{A}} \ Q_*(x_{k+1},\ a) \right\} \ . \end{aligned} \tag{3.31}$$

The second formula is called the *Bellman optimality equation* for Q_* [SB18, p. 64]. It offers a unique solution independent of the policy, for each state-action pair. When the optimal Q_*-value for an action is known, the policy can be defined by choosing the actions with maximum Q-value, also called the *greedy policy*:

$$\pi_q(x) \ = \ \underset{a \in \mathfrak{A}}{\operatorname{argmax}} \ \ Q(x,\ a). \tag{3.32}$$

If the probability and reward functions are known, the *Bellman optimality equation* can be solved using model-based algorithms. These functions are not predefined for many other problems, which is why model-free algorithms, like *Monte Carlo* or *temporal difference*, are chosen.

4 PROBLEM STATEMENT

All the information provided up to that point can be used to formalize a more detailed and precise problem statement. This section focuses specifically on the trade-off between utilization, associated with an existing control algorithm, and the QoC and the associated cost savings. In addition, the possibilities and assurances of the system are presented in order to get an overview of the requirements and the goal of this thesis.

As mentioned before, it can be assumed that the system is well-defined to guarantee stability over all time steps t_k, when the action $a_k \in \{0,\ 1\}$ is always chosen to be 1. This leads to the following assurance:

$$\hat{\Omega}_k \ \geq \ \Omega_{min}\ ,\ \ a_k \ = \ 1,\ \forall k \in \{t_0,\ ...,\ t_{k-1}\}. \tag{4.1}$$

Choosing the action variable a_k to be 1 means that all three events *measuring*, *computing*, and *actuating* are activated. In Rheinfels' master thesis, the tasks measuring and computing are set to be *mandatory*. Mandatory tasks are conducted at every controller period since they are critical for updating the state estimation [Rhe19, p. 17]. The situation is different for the *optional* actuating task, where the task can be left out without significantly lowering the estimate [Rhe19, p. 17].

The costs needed to perform the actuation task result in higher overall costs $L(a_k)$. For each of the three tasks, a runtime can be calculated [3], which is a larger or smaller part of the total task time. The individual costs for the task $l_{me},\ l_{co},\ l_{ac}$ depend on their phase φ, deadline D, and mean execution time e for a given period p. This means cost calculations, depending on the chosen a_k can be written as:

$$L(a_k) \ = \ \begin{cases} \frac{l_{me}+l_{co}+l_{ac}}{l_{me}+l_{co}+l_{ac}} \ = \ 1 & a_k \ = \ 1 \\ \\ \frac{l_{me}+l_{co}}{l_{me}+l_{co}+l_{ac}} & a_k \ = \ 0. \end{cases} \tag{4.2}$$

After calculating the cost for one time step k, the costs per run must be estimated. Furthermore, it must be ensured that in every time step the estimated QoC $\hat{\Omega}$ is above its minimum Ω_{min}. Formalizing these two requirements results in the following:

$$\lim_{N\to\infty} \frac{1}{N} \sum_{i=k}^{N} L(a_i) \quad \text{and} \quad \hat{\Omega}_k \ \geq \ \Omega_{min}\ ,\ \forall k \in \{t_0,\ ...,\ t_{k-1}\}. \tag{4.3}$$

[3] The costs for the tasks can be looked up in Appendix B.3

It should be noted that the infinite number of time steps N is finite in practical terms and set to a fix number of time steps in the simulation. Given all the information about the environment and the costs above, the final goal of this thesis can be summarized in the statement below:

In every time step k choose an action a_k, that minimizes the costs given in Equation (4.3) *under the assumption of a stable system, where $\hat{\Omega}_k \geq \Omega_{min}$ in every time step t_k.*

To ensure this behavior, a policy $\pi(\hat{x}, \hat{Q})$ can be defined, as mentioned in Equation (3.6), which generates the action a_k to be chosen. To get access to the current state x_k needed for the evaluation of the policy an observer is used to access a state estimation $\hat{x} \in \hat{\mathfrak{X}}$. It is assumed that this state can be used to estimate the QoC $\hat{\Omega}$ as well [Rhe19, p. 17]. The policy $\pi(\hat{x}, \hat{Q})$ depends on the current estimated system state $\hat{x}$ only, which is a restriction adopted from Rheinfels to reduce the complexity of the problem by allowing the policies to be time-invariant [Rhe19, p. 17]. Comparing the restriction of the environment and the MDP reveals that both depend on the current system state $\hat{x}_k$, making the environment well-suited for the approach of RL.

5 DEEP REINFORCEMENT LEARNING POLICY

The approach of DRL algorithms have proven their ability to address a wide range of previously unsolvable problems [FL+18, p. 312]. DRL combines the classic RL algorithms with the functionality of a NN. As mentioned in Section 3.3 RL can be used to lean a policy, which is able to make assurances about the environment by choosing a suitable action. With a given model-free environment as described in Section 3.1, only model-free algorithms like SARSA or Q-learning seem suitable. They both belong to the category of *temporal difference* learning and can bootstrap, which is essential in an environment with no final state. Furthermore, like most real-world problems, the state of the given system is continuous in time, leading to a problem when using the classic Q-learning and SARSA algorithm. Both save their updated action-value function in look-up tables, which does not scale in large or continuous state spaces. Therefore, the action-value function will be approximated by using a NN. The resulting DRL algorithm and its implementation details will be described in this chapter. First, Section 5.1 starts with the general introduction of Temporal Difference Learning (TDL), which includes the model-free algorithms Q-learing and SARSA, followed by adjustment for continuous state spaces in Section 5.2 and the reward modeling in Section 5.3.

5.1 Temporal Difference Learning

In a model-free environment that contains no final state, TDL can be very useful. These methods sample from the environment, like *Monte Carlo* methods, and perform updates based on current estimates, like *dynamic programming* methods [SB18, p. 119]. In contrast to Monte Carlo methods, TDL is updating its previous predictions in every time step and does not need to wait until the final outcome is known. Simply speaking, the algorithm is able to bootstrap [SB18, p. 89].

The idea behind TDL methods is that they only use experience gathered from exploring the environment to solve the prediction problem. After gaining some experience they follow a specific policy π and update their estimated Q-values Q_π for the states x of a given setup. The question is to know how much experience an agent needs to be able to learn a policy π, and more important, learn it in an intended way. This leads to a problem, which is known as the *exploration-exploitation* trade-off.

A decision-making system which has only incomplete knowledge of the world has two options: repeating a decision that has worked well last time (exploit), or gaining new experience and maybe a higher reward by trying a random action (explore). This is important as learning is online, and no

batches of data are given like in SL. Data is affected only by the data gathered while exploring and the taken actions. Occasionally it is worth taking another action to get better results. Exploration does also have disadvantages like higher risks and slower convergence. Hence, exploring in early epochs and with increasing probability exploiting in the later epochs seems to be a good compromise and is called the ε-greedy policy. It in this thesis as follows [SB18, p. 127]:

$$\pi_\varepsilon(a_k|\hat{x}_k) = \begin{cases} 1-\varepsilon+\varepsilon/|\mathfrak{A}| & a_k = \underset{a\in\mathfrak{A}}{\operatorname{argmax}}\ Q(\hat{x}_k,\ a_k) \\ \varepsilon/|\mathfrak{A}| & \text{otherwise.} \end{cases} \tag{5.1}$$

The ε-greedy strategy selects the action a_k with the highest Q-value and a probability of $1-\varepsilon$ in a particular estimated state $\hat{x}_k$, or randomly selects one from all possible actions of the state with the ε probability. $|\mathfrak{A}|$ represents the number of all possible actions a_k in state $\hat{x}_k$. Normally, the initial value of ε is set to be 1.0 and decayed depending on the exploration rate by 0.9 or 0.99.

In this work, the state is initialized at the start of every epoch and continues randomly because of noise and disturbances. Every epoch is divided into several time steps t_k where the main algorithm is evaluated. ε is initialized by 1.0 and decayed every epoch depending on the exploration rate ε_{decay}. In this thesis it was discovered that both system models do not need to explore more than using the value of 0.9, because of the randomness gathered from noise and disturbances.

The resulting action a_k, whether chosen randomly or not, is used to estimate the next state $\hat{x}_{k+1}$ and leads to a change in the action-value function. Updating this Q-value can be done on-policy and off-policy. The most common on-policy method is called SARSA, introduced in Section 5.1.1, whereas a common off-policy method is the Q-learning algorithm, explained in Section 5.1.2 [SB18, p. 154].

5.1.1 SARSA

SARSA is an on-policy method that estimates the Q-value for the current policy π written as $Q_\pi(a_k,\ \hat{x}_k)$ with given *action* a_k and *states* $\hat{x}_k$. Its name comes from the quintuple (State, Action, Reward, State, Action), including all necessary information needed to update the state-action value function. As mentioned in the previous chapter, a_k can choose to be the maximum expected Q-value or random. It is called on-policy because it follows a given policy in every time step k. In this case, the previously mentioned ε-greedy policy is used. Being part of the TDL algorithms results in updating the estimated Q-value in every time step k, by a given formula including the *temporal-difference* error δ_k, which is calculated by the difference of the *temporal-difference target* and the *prediction value* [SB18, p. 129]:

$$\delta_k := \underbrace{r_{k+1}+\gamma Q(\hat{x}_{k+1},\ a_{k+1})}_{temporal-difference\ target} - \underbrace{Q(\hat{x}_k,\ a_k)}_{prediction\ value} \tag{5.2}$$

$$Q(\hat{x}_k,\ a_k) \leftarrow Q(\hat{x}_k,\ a_k)+\alpha\delta_k. \tag{5.3}$$

Hence, the action-action value function can be updated every time step k with step size $\alpha \in [0,\ 1)$ which results in the minimization of the temporal-difference error. Combining this algorithm and the ε-greedy policy, introduced in the previous chapter results in the algorithm illustrated in Algorithm 5.1.

Initialize states $\mathfrak{X} = \{1, \ldots, n_x\}$
Initialize actions $\mathfrak{A} = \{1, \ldots, n_a\}$
Initialize Q-values $Q(x, a) = 0$
Initialize learning rate $\alpha \in [0, 1)$, typically $\alpha = 0.1$
Initialize discounting factor $\gamma \in [0, 1)$
for each epoch **do**
  Initialize $x_1 \in \mathfrak{X}$
  Choose a_1 from x_1 using ε-greedy
  for each time step k and $\hat{x}_k \neq$ terminal **do**
    Take action a_k, observe r_{k+1}, $\hat{x}_{k+1}$
    Choose a_{k+1} from $\hat{x}_{k+1}$
    $Q(\hat{x}_k, a_k) \longleftarrow Q(\hat{x}_k, a_k) + \alpha[r_{k+1} + \gamma \; Q(\hat{x}_{k+1}, a_{k+1}) - Q(\hat{x}_k, a_k)]$
    Calculate π based on Q (e.g., ε-greedy)
    $\hat{x}_k \leftarrow \hat{x}_{k+1}$
    $a_k \leftarrow a_{k+1}$
  end for
end for

Algorithm 5.1 – The SARSA algorithm calculating the Q-values in each time step k, including Equation (5.1) and Equation (5.2) [SB18, p. 130].

The Q-value update is accomplished under the assumption that $\hat{x}_k$ is not the terminal state after every transition from state $\hat{x}_k$ to state $\hat{x}_{k+1}$.

Furthermore, the algorithm converges with probability 1 to an optimal policy and state-action value function under the assumption that all state–action pairs are visited an infinite number of times. Also, the convergence of the policy is required, which for ε-greedy can be achieved by setting $\varepsilon = 1/k$ [Sin+00, pp. 293–295].

5.1.2 Q-Learning

In comparison to the previously described SARSA algorithm, the Q-learning approach is updating its state-action values off-policy [SB18, p. 131]:

$$\delta_k := r_{k+1} + \gamma \max_{a \in \mathfrak{A}} Q_*(\hat{x}_{k+1}, a) - Q(\hat{x}_k, a_k) \tag{5.4}$$

$$Q(\hat{x}_k, a_k) \leftarrow Q(\hat{x}_k, a_k) + \alpha \delta_k. \tag{5.5}$$

The difference resides only in calculating the temporal-difference target, depending on the maximum state-action value of all viable actions in $\mathfrak{A}$. Accordingly, it is not following the ε-greedy policy and is therefore called off-policy. Because of this property, the convergence proof is much simpler and is only requiring the state-action value pairs to be updated [SB18, p. 157].

Both algorithms have their advantages and disadvantages, making them more or less optimal in different systems. While Q-learning directly learns the optimal policy, SARSA learns a “near” optimal policy. This results in SARSA being more conservative, while in Q-learning the agent can be more aggressive by trying to get the best possible reward [SB18, pp. 131–132].

In this thesis, both of them are used and compared, mainly because there is more space of optimization when using Q-learning [Mni+15, p. 529] and SARSA acting more conservative.

5.2 Nonlinear State-Value Function Approximation

Applying the algorithms introduced in the previous chapter to the given environment, results in one big problem. The classic TDL approaches save their state-action values in a look-up table. By increasing the complexity of the problem the table's capacity can be overwhelmed by the numerous state-action values. Furthermore, there are often no discrete states or actions in natural environments, which faces the agent with continuous states and actions.

In this thesis, there are only two options (performing the actuation task or not), but many states the system can remain. As a result, the MDP needs to be formalized as continuous while not violating the given Markov property [SB18, p. 57]. The temporal-difference algorithms are adapted to be able to handle continuous state spaces. After talking about the general adjustment, the NN architecture used in this thesis is presented.

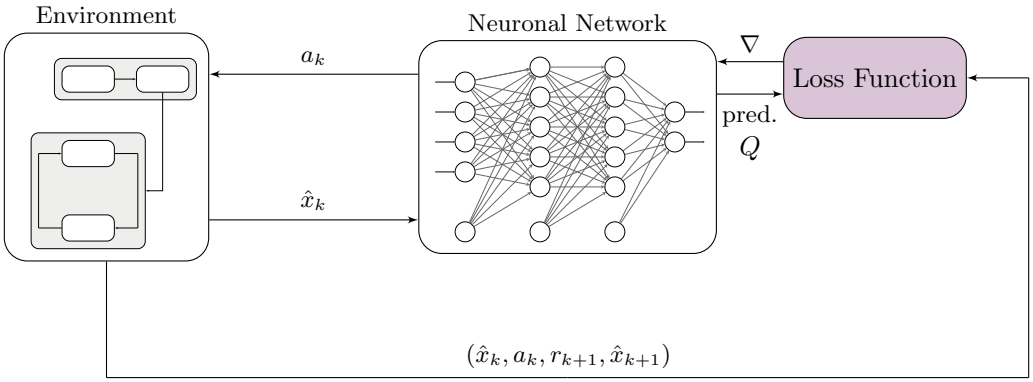

Figure 5.1 – An overview of the action-value approximation using a NN. The environment is providing the states, while the NN estimates the best possible action a_k. The network is trained using gradient descent. This optimization process requires a loss function. To simplify the picture, the QoC is considered as part of the state $\hat{x}_k$.

There are two ways to approximate the action-value function: a linear, as described in Section 5.1 or a non-linear by using NN. The latter is described in this section. Recall the *Bellmann Optimality Equation*, where the optimal state-action value is defined by the sum of the direct reward r_{k+1} and the maximum of the discounted Q-value of the next step $k+1$. This equation can also be solved using a non-linear approximator like a NN. It is done by using the loss[4] as define in Equation (5.6) to update the weights w of the network.

$$L(w) \;=\; \mathbb{E}\left\{\left((r_{k+1}+\gamma \max_{a\in\mathfrak{A}} Q(\hat{x}_{k+1},\, a,\, w)) - Q(\hat{x}_k,\, a_k,\, w)\right)^2\right\} . \tag{5.6}$$

After calculating the loss, the gradient descend step is performed, as seen in Section 3.2.3. The NN is now able to "learn" the optimal value function, which helps the agent to choose the best possible

[4]Calculated by using the MSE in this formula. In this case Q-learning is used for illustration.

action a_k in an estimated state $\hat{x}_k$ with QoC $\hat{\Omega}_k$. For a better overview, this process is illustrated in Figure 5.1.

5.3 Reward Function Modeling

A reward function $r : \mathfrak{X} \times \mathfrak{A} \times \mathfrak{X} \to \mathfrak{R}$ provides feedback while the agent explores the environment, given the goal of maximizing the overall reward. In fact, the reward function has a significant influence on the agent's behavior. Hence, it is essential to know which components to examine to find the best possible reward function. They can be divided into two groups: *general* and *problem-specific* components. First, the problem-specific goals, which are in this thesis the goal of minimize the overall costs, under the assumption, that the QoC stays close to equilibrium are introduced. As a result, the agent needs to get feedback for:

- Keeping the QoC above its lower limit $r_q(\hat{\Omega}_k)$
- Sparing processor time when a job is not activated $r_u(a_k)$

General components include:

- Frequency of feedback
- Reward or punishment
- Ratio of values

In the following, each of the points will be introduced in more detail, including some ideas based on [Esc21, pp. 25–33].

The first one is the frequency at which an agent gets feedback by the reward. When the reward is given sparsely, the agent is not rewarded very often, resulting in no feedback when the agent is exploring spaces far away from the goal. Therefore, learning might take more time. However, this can also be an advantage because the agent is less often led to specific sub movement sets. The other way is to build a reward function, which gives continuous feedback how near the agent is to the desired state, resulting in faster learning speed, but might lead to unintended behavior. A good example might be rewarding the agent when it is close to breaking the QoC bound. The agent learns that playing on the edge leads to the highest reward, regardless of the given risk of failure when it does not actuate early enough.

The second point is how to structure the reward numerically. By obtaining positive rewards, the agent will try to sum them up. This can lead to the same unintended behavior as mentioned before: always collecting the highest reward and therefore forcing risky situations. Negative rewards let the agent stay away from certain situations, but give disincentive to stay away from the possible optima.

The last component is the value $r_q(\hat{\Omega})$ in relation to the value $r_u(a)$. When rewarding the agent frequently with the value of 1 by not actuating and only punishing it with a value of -1, when falling below the minimum QoC, the agent does not take the low QoC into account. Based on these aspects, the reward functions used in this thesis are presented.

The *utilization reward* is calculated by:

$$r_u(a_k) = \begin{cases} u & a_k = 0 \\ 0 & \text{otherwise,} \end{cases} \tag{5.7}$$

where u is some positive coefficient. The *QoC reward*, was evaluated using several approaches, finally resulting in three different functions, which are defined in the following:

$$r_{\Omega_1}\left(\hat{\Omega}_k\right) = \begin{cases} q1 & \hat{\Omega} \geq \Omega_{min} \\ 0 & \text{otherwise} \end{cases} \tag{5.8}$$

$$r_{\Omega_2}\left(\hat{\Omega}_k, \hat{\Omega}_{k+1}\right) = \begin{cases} q2 & (\hat{\Omega}_{k+1} - \hat{\Omega}_k) > q3 \\ 0 & \text{otherwise} \end{cases} \tag{5.9}$$

$$r_{\Omega_3}\left(\hat{\Omega}_k, \hat{\Omega}_{k+1}\right) = (\hat{\Omega}_{k+1} - \hat{\Omega}_k)q_4, \tag{5.10}$$

where again $q1,\ q2,\ q3$ and $q4$ are some adjustable positive coefficients. For the double integrator system model an additional reward function r_{Ω_d} with positive coefficient d is used:

$$r_{\Omega_4} = \begin{cases} \hat{\Omega}_{k+1} < \Omega_{min} \text{ and } a = 1) & d \\ 0 & \text{otherwise.} \end{cases} \tag{5.11}$$

Both reward parts are added to generate the over all reward:

$$r(a_k, \hat{\Omega}_k, \hat{\Omega}_{k+1}) = r_\Omega(\hat{\Omega}_k, \hat{\Omega}_{k+1}) + r_u(a_k). \tag{5.12}$$

The fitting coefficient values for both system models were evaluated empirically. The reward functions are chosen to be as simple as possible, focusing on the two central aspects of minimizing the utilization and maximizing the QoC. r_{Ω_2} and r_{Ω_3} have their focus on the QoC. difference, where a fast decreasing of the QoC. is punished harder than a slow decreasing one. Because the formulas provide a large enough space of parameter freedom, the "else case" is set to be zero. A special additional reward function for the double integrator is used because this system model has the particular case that one actuation is enough to restabilize the system in most cases. Hence, the reward can be set more precisely here.

6 EVALUATION

In this chapter, the approach of DRL will be tested using two different simulated system models: an *inverted pendulum* and a *double integrator*. Both express different system behavior, which is influenced by adjustable process and measurement noise. The chapter evaluates the trade-off between QoC and utilization by applying the DRL algorithms to the two experimental setups.
First, the evaluation metric defined in Section 6.1 which will be used to determine the algorithm's performance is introduced. An overview of different optimization methods for the algorithms is provided in Section 6.2. In the following sections, the structure of the implemented DRL algorithm, including parameter assignment (Section 6.3), the network architecture, and implementation details (Section 6.4), algorithms (Section 6.5) and reward function modification (Section 6.6), are proposed. The evaluation of the experiment setup and outcomes using the previously defined evaluation matrices are given in the last chapter (Section 6.7). It contains the experimental evaluation of the two system models and a comparison to the existing approaches for the same environment, given in Rheinfels' master thesis [Rhe19, p. 74].

6.1 Evaluation Metrics

Finding a way of measuring the performance of *DRL* can be quite difficult, especially for making it comparable to *Supervised Learning* or other policies. Therefore, the metrics given in [Rhe19, p. 19] are used, which focus on measuring the trade-off between QoC and utilization. They will be explained in the following.

As mentioned before, the QoC defines how far away a system is to its equilibrium. To make it comparable between system models, its scaled from $[\Omega_{min},1]$ to lay between $[0,\ 1]$. The first metric is known as the *QoC metric* m_Ω. It describes the average QoC over N time steps in one simulation run.

$$m_\Omega \ := \ \frac{1}{N} \ \sum_{k=1}^{N} \ max \ \left\{ \frac{\Omega_k - \Omega_{min}}{1-\Omega_{min}},\ 0 \right\} . \tag{6.1}$$

The second metric called *utilization metric* m_u gives an overview of the utility by calculating how many jobs are activated in a given time step k. As mentioned before, it is also normalized to lay between 0 (all jobs activated) and 1 (no job is activated). The results are then averaged across all time steps as seen in Equation (6.1). More precisely, $a_k^+ \in \mathfrak{A}^{n_T}$ denotes the vector of all optional activations and $u^+ \in \mathbb{R}^{n_T^+}$ the corresponding worst-case utilization [Rhe19, p. 19].

$$m_u \ := \ \frac{1}{N} \ \sum_{k=1}^{N} \ max \ \left(1 - \frac{(u^+)^T a_k^+}{(u^+)^T \mathbb{1}} \right) . \tag{6.2}$$

Given v_k , m_b calculates how often the QoC falls below its minimum. This metric, called *bound metric* m_b is then defined as the relative amount of violations during the simulation and also averaged across all time steps.

$$v_k := \begin{cases} 0 & \Omega_k \geq \Omega_{min} \\ 1 & \Omega_k < \Omega_{min} \end{cases} \tag{6.3}$$

$$m_b := \frac{1}{N} \sum_{k=1}^{N} max\ (1 - v_k)\ . \tag{6.4}$$

The resulting metric is the product of all the other metrics:

$$m := m_\Omega m_u m_b. \tag{6.5}$$

6.2 Optimization

In RL, especially when using DRL, there is a lot of room for improvement. A multitude of adjustments can be done to increase the algorithm's performance. The most common methods will be introduced in the following chapters. This is embedded by a short evaluation that ends in a discussion about its effectiveness. To give an overview of the optimization strategies and their way of interacting with the whole system and the DRL approach, Figure 6.1 can be refereed.

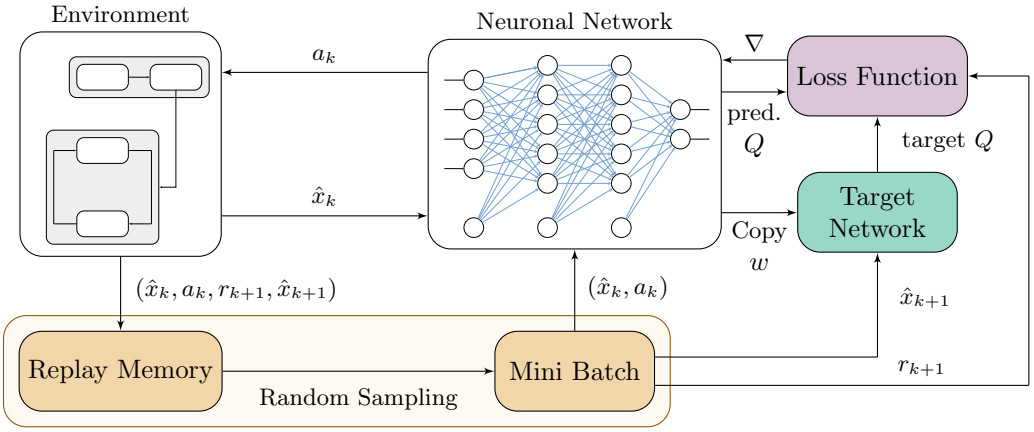

Figure 6.1 – An overview of the system, the DRL approach, and the optimizations. The Experience Replay Buffer (ERB) includes the Mini Batch, which samples random transitions from the Replay Memory. A Target Network (TN) saves a copy of the weights from the NN. Optimization like Weight Initialization and Loss Calculation are directly connected to the NN and can improve its learning speed. Adopted from [Gre+20].

This section starts with the improvement by using a TN, followed by the ERB, containing the *replay memory* and the *mini batches* sampled from. Next, a general overview of the hyper-parameter is given, as well as the NN improvements consisting of the *loss function* and *weight initialization*.

The chosen optimizations are improving the algorithm in most cases but do not yield better results in every parameter combination.

6.2.1 Target Network

The learning process of the NN depends on the difference between the predicted value and the target value. The weight update step can be written more precisely as:

$$\Delta w \;=\; \alpha[r_{k+1} + \gamma \max_{a \in \mathfrak{A}} Q(\hat{x}_{k+1},\, a,\, w) - Q(\hat{x}_k,\, a_k,\, w)] \;\; \nabla_w Q(\hat{x}_k,\, a_k,\, w), \tag{6.6}$$

where $\nabla_w Q$ is the gradient of the current predicted Q-value. Taking a closer look reveals that the state-action value function to be approximated is updated by another approximated state-action value function. Simply speaking, the NN is updating a guess from the current state $\hat{x}_k$ with a guess from the next state $\hat{x}_{k+1}$. This can lead to unintended correlations and might result in unstable learning behavior. The idea is now to freeze parameters to make it easier for the network to approximate the state-action value as the target is fixed for an amount of time. This can be achieved by a so-called TN, which saves a copy of the NN and uses it for the target value used in the Bellman equation [Mni+15, p. 529]. Therefore, the TN is periodically synchronized with the weights of the main network. The resulting change in the algorithm can be seen in Algorithm 6.1, together with the ERB explained in the next chapter.

1: Initialize all variables needed to perform Q-learning
2: Initialize replay memory D to capacity N
3: Initialize state-action value function Q with random weights w
4: Initialize target state-action value function $Q^{'}$ with weights $w^{'} = w$
5: **for** each episode **do**
6: Initialize $x_1 \in \mathfrak{X}$
7: **for** each time step k and $x_k! =$ terminal **do**
8: With probability ϵ select a random action at a_k
9: otherwise select $a_k = \max_{a \in \mathfrak{A}} Q(\hat{x}_k,\, a,\, w)$
10: Execute action a_k in emulator and observe reward r_{k+1} and state $\hat{x}_{k+1}$
11: Store transition $(\hat{x}_t,\, a_k,\, r_{k+1},\, \hat{x}_{k+1})$ in D
12: **if** enough experience in D **then**
13: Sample random mini batch of transitions $(\hat{x}_j,\, a_j,\, r_{j+1},\, \hat{x}_{j+1})$ from D
14: **for** every transition $(\hat{x}_j,\, a_j,\, r_{j+1},\, \hat{x}_{j+1})$ in minibatch **do**
15: $y_j = \begin{cases} r_{j+1} & \text{if episode terminates at step j+1} \\ r_{j+1} + \gamma \max_{a' \in \mathfrak{A}} Q^{'}(\hat{x}_{j+1},\, a^{'},\, w^{'}) & \text{otherwise} \end{cases}$
16: Perform a gradient descend step on $(y_j - Q(\hat{x}_j,\, a_j,\, w))^2$ with respect to the network parameter w
17: Every C steps reset $Q^{'} = Q$
18: **end for**
19: **end if**
20: **end for**
21: **end for**

Algorithm 6.1 – The Deep Q-Learning (DQN) algorithm with ERB buffer and TN. Values with an apostrophe mark the values gained from the TN, which is updated every C steps. In line 11 the transition is stored into the replay buffer and randomly sampled from it and trained afterwards [Mni+13, p. 5].

Figure 6.2 illustrates a common problem, which can be solved by the use of a TN, called *catastrophic forgetting*. Under certain conditions, the process of learning a new set of patterns can wholly and suddenly erase a network's knowledge of what it had already learned [Fre99, pp. 1–3]. This behavior can be seen in epoch 500 when the cumulated reward decreases fast, followed by a rapid rise close to the 1000s epoch. Even though the stability of the algorithm can improve, it might also slow down its convergence. The decrease in convergence can also be observed in Figure 6.2, where the setting with the TN reaches the highest cumulated reward as late as epoch 620.

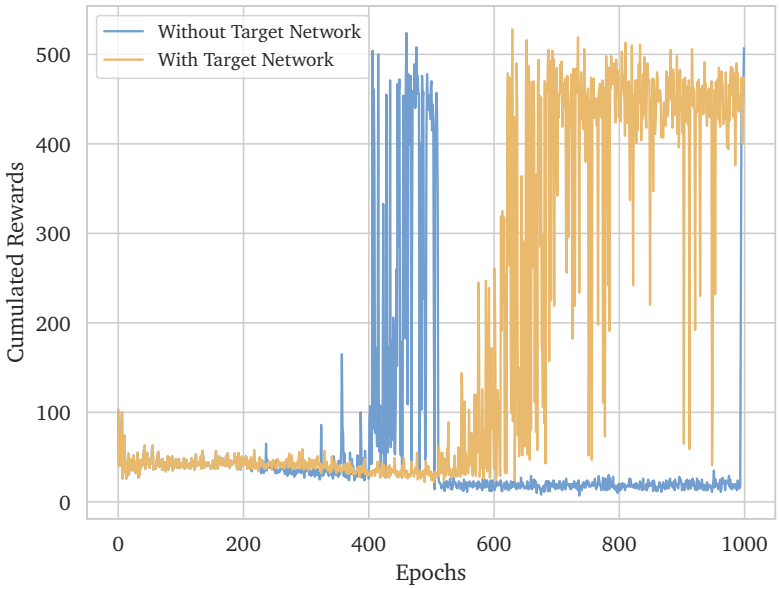

Figure 6.2 – The cumulative reward of 500 time steps over 1000 epochs. The graph without the TN suffers from catastrophic forgetting, whereas the graph with the TN rises constantly and persist at a high cumulated reward.[5].

6.2.2 Experience Replay

In 1993 ER was proposed the first time as a technique to speed up RL [Lin92, p. 36]. The key idea of ER is to buffer experiences gained from previous transitions and train the agent by sampling this transitions from a buffer [ZS17, p. 2]. A transition is defined by a quadruple $(x_k,\ a_k,\ r_{k+1},\ x_{k+1})$ consisting of all relevant information needed to perform a DQN step.[6] Adding the ERB into the previously introduced DQN results in Algorithm 6.1.

Using ER can improve the basic DQN algorithm by the following:

- increased learning speed by sampling from multiple mini batches

[5]The parameter set is given in Table C.1.
[6]The ERB is only used for the DQN approach in this thesis, because it is more common and less complex.

- using previous experiences more efficient by reusing them multiple times
- improving the sample efficiency by breaking the temporal correlations [ZS17, p. 7]

To see how the buffer effects the learning process in the given environment, a run was simulated calculating the cumulated reward over time. The result in Figure 6.3 shows the increased convergence achieved by the experience replay. Through ER th algorithm is able to get the highest reward before epoch 400 is reached.

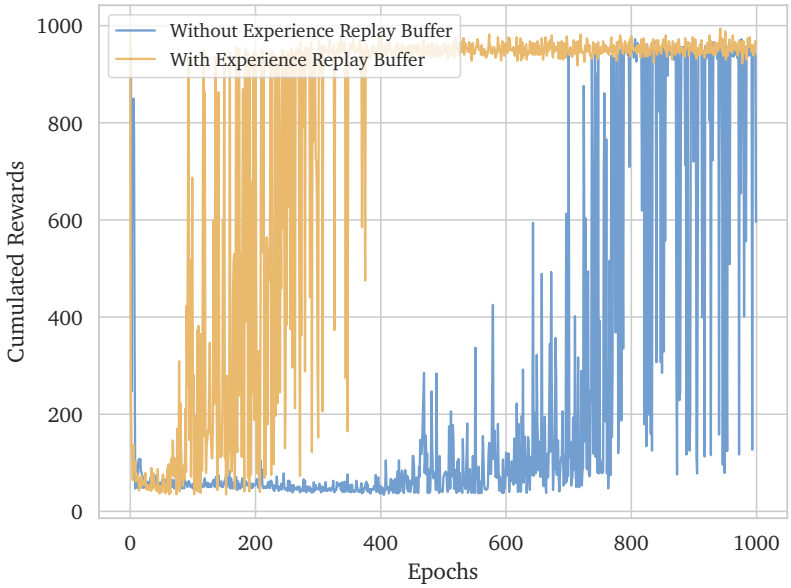

Figure 6.3 – The cumulative reward of 500 time steps over 1000 epochs. The graph without the ERB is reaching the highest cumulated reward at epoch 800. With the ERB the graph reaches the reward of 900 before epoch 400.[7]

6.2.3 Loss Function

As mentioned before, the NN estimates the state-action value functions for the RL algorithm. Weights are continuously updated by backpropagating the error to the input and generating a learning effect. Typically, this is done by using the gradient descent method, which was introduced in Section 3.2.3. Especially at the beginning of the learning process when using DRL, the error gradients can be enormous, which leads to a loss of stability. Mnih found a way to improving the stability of the algorithm by clipping the error term to lay between -1 and 1 [Mni+15, p. 535]. An equivalent optimization can be made by using the Huber loss.

The Huber loss is containing both MSE and MAE, meaning it is only quadratic when the error is small and else absolute [Hub92, pp. 73–75]. To customize the range for MAE and MSE an addi-

[7] The parameter set is given in Table C.2.

tional parameter δ is used. The Huber loss can be calculated:

$$L_H(Y,\ \hat{Y}) \ = \ \begin{cases} \frac{1}{n}(Y-\hat{Y})^2 & \text{for } \ |(Y-\hat{Y})| \ \leq \ \delta \\ \delta(|(Y-\hat{Y})| - \frac{1}{n}\delta), & \text{otherwise.} \end{cases} \tag{6.7}$$

Figure 6.5 provides the resulting plots for the three loss functions in the range $[-2,\ 2]$. It can be seen that the gradient of the MAE is the same throughout, which means the gradient will be large even for small loss values. MSE squares the error resulting in very high gradients for outliers. Huber loss combines the advantages of both functions and is also adjustable. Residuals larger than δ are minimized with MAE, while residuals smaller than δ are minimized using MSE.

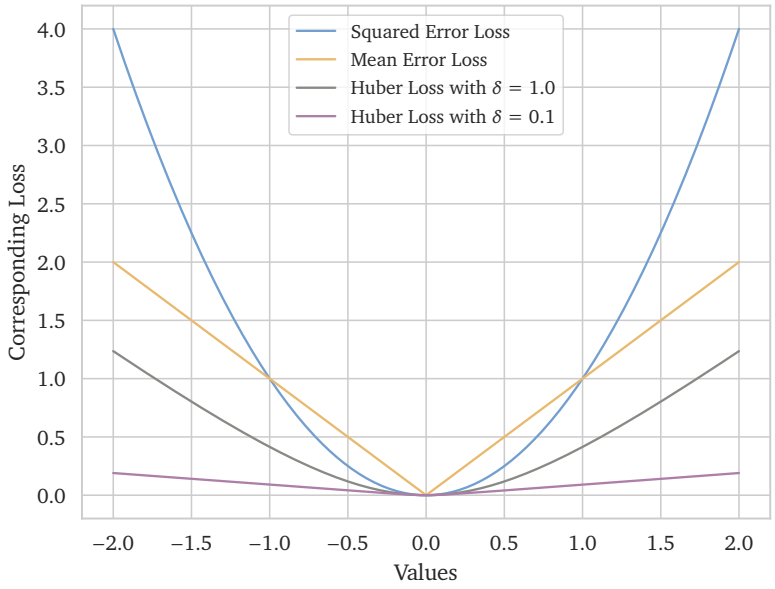

Figure 6.4 – Corresponding losses for the three functions MSE, MAE and Huber loss with $\delta = 1.0$ and $\delta = 0.1$.

6.2.4 Weight Initialization

The weight initialization technique to be chosen for a NN can determine how fast a network converges or whether it converges at all. In general, weights represent the strength of the connection between different layers and therefore affect the gradients. Two central problems go along with wrong weight initialization, called the vanishing and exploding gradient problem. Vanishing gradients result in very slow convergence and can even stop the network from converging. Exploding gradients lead to large error gradients accumulation and result in massive updates. In the worst case, it can result in "NaN" weight values, which leads to an unusable NN because they can not be used or updated.

He et al. show that using the *He* initialization method leads to good results when using the ReLU activation function or their improved forms [He+15, p. 5]. In his paper, the focus was placed on faster convergence and comparison to the state-of-the-art initialization called *Xavier* [GB10]. Given

the following formulas for the two methods:

$$w_{k,j} = \mathcal{U}\left(-\frac{\sqrt{6.0}}{\sqrt{n_k + n_j}}, \frac{\sqrt{6.0}}{\sqrt{n_k + n_j}}\right) \tag{6.8}$$

$$w_{k,j} = \mathcal{N}(0,\ 1)\frac{\sqrt{2.0}}{\sqrt{n_k}}, \tag{6.9}$$

where n_k is the number of incoming and n_j is the number of outgoing connections. The bias tensors are initialized with one. It's important to keep in mind that those formulas are optimized for the ReLU/Leaky ReLU activation function and need to be changed to achieve the same results for other activation functions.

In Figure 6.5 the cumulated reward over 100 epochs can be seen. It compares the three initialization methods He, Xavier and the weights initialized by zero in the given environment using the DRL algorithm. Initializing the weights with zero results in a symmetry problem. When all the hidden neurons start with the zero weights then all of them will follow the same gradient. This only effects the scale and not the direction of the gradient [Bis06, p. 240], which leads to poor performance. He initialization can reach a higher cumulated reward within the first 90 epochs. Empirical evaluations have shown that both methods He and Xavier provide good results because in long training runs (500 epochs or more), the initialization methods play only a small role compared to other factors.

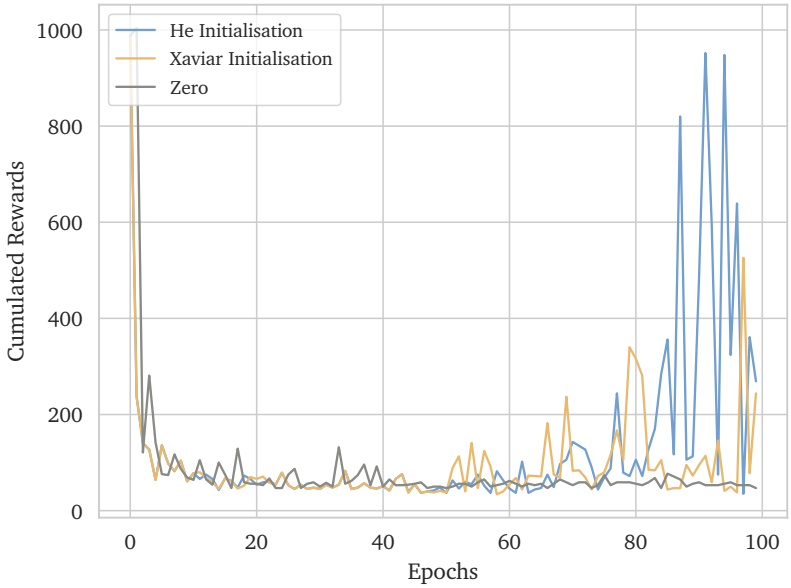

Figure 6.5 – Cumulated reward for 500 time steps in 100 epochs, where the He initialization reaches a higher cumulated reward[8]. The zero initialized NN shows almost no improvement whereas the Xavier network shows a small improvement.

[8]The parameter set is given in Table C.2.

6.3 Parameter Assignment

After presenting the more complex optimization methods, there is also another way of optimizing the DQN algorithm by choosing the optimal hyperparameters. Model hyperparameters are parameters that cannot be learned during training but are set beforehand [FH19, p. 4]. All these variables affect the learning of the NN differently. Choosing these parameters is one of the most challenging and time-consuming parts of the DRL process because the agent needs to be trained to see how the hyperparameter effects the performance. Other papers with similar environments can help to find a good value to start with, but there is no prior way to know whether to choose the value to be high, low, or something between. Only after several training runs and multiple parameter combinations there can be a guess by watching the improvement of the total reward.

Choosing the right hyperparameter also requires an overview of all possible adjustments, which can influence the outcome of the experiments. Therefore, a summary is provided in Table 6.1, containing all viable parameter options and ranges. It has to be taken into account that all possible combinations for the network architecture and reward function are not tested because there are simply too many possibilities of adjustment.

Category	Parameter Type		Value Range
Neuronal Network	Network Architecture		–
	Loss	MSE	–
		Huber	$\delta \in [0, 10]$
	Weight Initialisation	He	–
		Xavier	–
DRL Algorithm	Learning Rate		$\alpha \in (0, 1]$
	Algorithm	Deep SARSA	–
		Deep Q-Learning	–
	Discount Factor		$\gamma \in [0, 1]$
	Epsilon Greedy		$\epsilon \in [0, 1]$
	Epsilon Greedy Decay Rate		$\epsilon_{decay} \in [0, 1]$
	Experience Replay Buffersize		$N \geq 1$
	Mini Batch Size		$D \geq 1$
	Target Network Update Frequency		$C \geq 1$
	Reward		–
Simulation Parameter	Epochs		$E \geq 1$
	Time steps		$T \geq 1$

Table 6.1 – An overview over the possible parameter combinations and values divided into three categories.

It is to mention that the network architecture can be adjusted but is set to an empirically chosen structure. This was done to reduce the possible parameter permutations. The best evaluated struc-

ture is introduced in Section 6.4. The other major influence in learning, the reward function, is used as defined in Section 5.3 and modified in Section 6.6.

After setting these two parameters there are still many possibilities left to choose from. Most of them where set by comparing the system model and behavior to other research papers and then continuously adapting them. As previously mentioned, the only way of finding the best fitting parameter is by trial and error. So the experiments provided in Section 6.7 contain the best results gathered empirically and some change in parameter to see their impact on the overall result.

6.4 Network Architecture and Implementation Details

Selecting a network architecture for a given problem is non-trivial. There are several factors, which can influence the learning behavior and lead to non-optimal results. Within the approach of DRL it is even more important that the NN is able to adapt fast because the function to be approximated is generated by exploration and not by a given data set. This section describes the network architecture chosen to learn the policy $\pi(\hat{x},\ \hat{\Omega})$ which maps the trade-off between utilization and QoC. As few layers and neurons as possible were used, as well as an activation function that could be calculated and derived fast.

The state of the system models $\hat{x}$ as described in Section 3.1, contains four estimated values $(\hat{x}_1,\ \hat{x}_2,\ \hat{x}_3,\ \hat{x}_4)$, each of them describing one part of the system state. These states are passed to the NN, in addition of the QoC $\hat{\Omega}$. As seen in Equation (3.11) the estimated $\hat{\Omega}$ can be calculated with the measured state $\hat{x}$ and therefore does not need to be given as input. To simplify the application it is also passed to the network.

After determining the number of neurons for the input layer the number of neurons in the output layer needs to be adjusted. The NN needs to approximate the state-value function for the actions $a_k = 0$ and $a_k = 1$ in every estimated state $\hat{x}_k$ to find out which function provides the maximum value. There are two possibilities: using two NNs, each of them approximating one state-value, or using one NN with two output neurons for every possible action a_k. The latter is chosen related to the fact that it is able to map context from one action to another. In every time step k the whole net is updated independently of the chosen action. When using two networks there is a slight chance that one network is updated less often than the other which can result in less optimal behavior and leads in general to a higher need for random actions.

Another complex part of building a a NN is selecting the number of hidden layers and their amount of neurons. Too many neurons can lead to *overfitting* [Yin19], while not enough neurons can slow down the learning process. A single-layer network can only be used for linear-separable functions, whereas a MLP can overcome the limitation of linear-separable by being able to approximate more complex shapes. For regression problems both single-hidden layer [WEH15] and two-hidden layers [Qu+16] are commonly used. Actually, the number of hidden layers and their amount of neurons depend on the complexity of the problem and the surrounding layer architecture. Therefore, it has to be evaluated empirically. During the implementation of this thesis, both suggestions where tried, which results in in the best performance for two hidden layers consisting of five neurons each. The neurons are chosen to be *fully connected*, because linking every neuron needs no special assumption about the input [ML17, pp. 5–7]. This generality makes them applicable to many kinds of problems.

There is also the need to choose a proper activation function. When obtaining a regression problem, the last layer normally consists of a linear activation function. The most common functions for the hidden layers are the *Sigmoid* and *ReLU* functions. The optimized ReLU function, called Leaky ReLU, defined in Equation (3.17) is used in this thesis, because it has some advantages over other activation functions. Because rectified linear units are nearly linear, they can be calculated faster than the Sigmoid function and make linear models easy to optimize with gradient-based methods. Furthermore, they do not have the problem of *vanishing gradients* [GBB11, p. 318].

It is to mention, architecture described in the following is used for both the inverted pendulum and double integrator because they have the same amount of estimated states.

Regression or Classification

The NN is trained like a regression problem using the Huber loss and Leaky ReLU activation, while the final choice which action to take is actually a classification problem. Going more into details, each output estimates the total reward gained for each action. Even if these output values are discrete, the last decision to take the topmost candidate is done by classification. As a result, the network can be labeled as regression with its final choice made by classification.

6.5 Algorithms

As mentioned in Section 5.1, the two temporal difference methods SARSA and Q-learning are used in the DRL approach. Both of them are updating the temporal difference learning formula slightly different. As seen in Equation (5.2) and Equation (5.4), Q-learning takes the maximum possible reward of the new state $\hat{x}_{k+1}$, whereas SARSA still follows the policy to the new state $\hat{x}_{k+1}$. In the following, both algorithms are compared regarding their execution times and learning behavior.

Both algorithms are trained for several epochs containing a finite number of time steps. In every time step, a DRL step is performed, including the state-action value function approximation with subsequent action decision and the NN training step. To estimate the average execution time per time step measurement is performed, where SARSA, Q-learning, and Q-learning with ER and TN are compared. The result is provided in Figure 6.6. The slight discrepancies in execution time can be explained by the usage of the same NN. Nevertheless, the execution time of the optimized Q-learning approach is higher because the experience replay buffer takes time for saving, sampling, and learning the mini batches. Furthermore, there is also the cost of updating the TN every C time steps.

Furthermore, the algorithms differ in their behavior when it comes to a decision whether to actuate or not. As previously mentioned in Section 5.1.2, the SARSA algorithm is behaving more conservative because of its on-policy behavior. Figure 6.7 shows how often the QoC falls below its minimum value before the policy is learned. As expected, the optimized Q-learning approach performs best because it needs fewer epochs to learn. The interesting comparison is between SARSA and the non-optimized Q-leaning algorithm. SARSA is violating the minimum QoC half as much as Q-learning does. It must be considered that Q-learning with the given parametrization also takes twice the amount of epochs. This observation can also be made in experiment 2.1 and 3.1, located at Appendix D, where only SARSA is able to not fall below the minimum QoC, when process and measurement are increased.

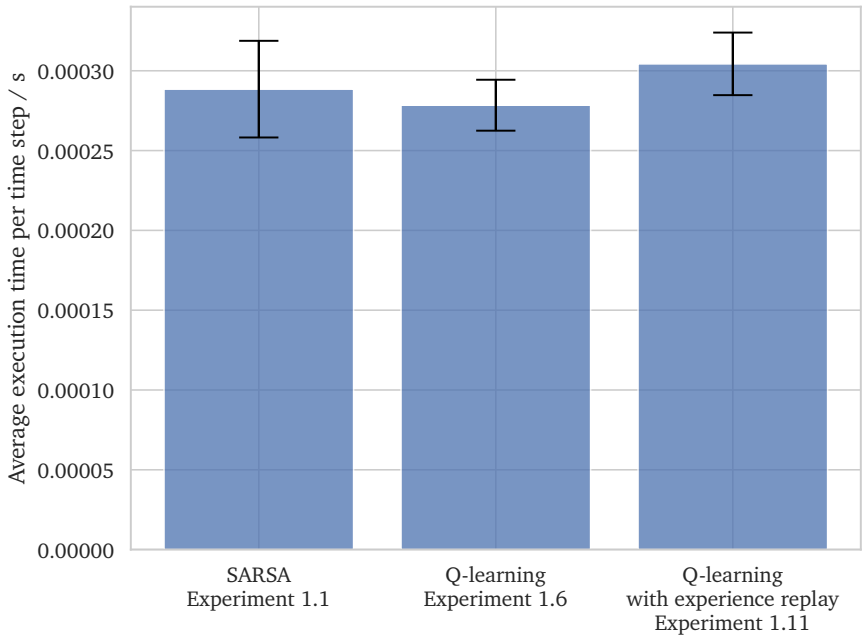

Figure 6.6 – Average execution time over 500 steps with standard deviation. Measured are the experiments 1.1, 1.6 and 1.11, as given in Table D.1.

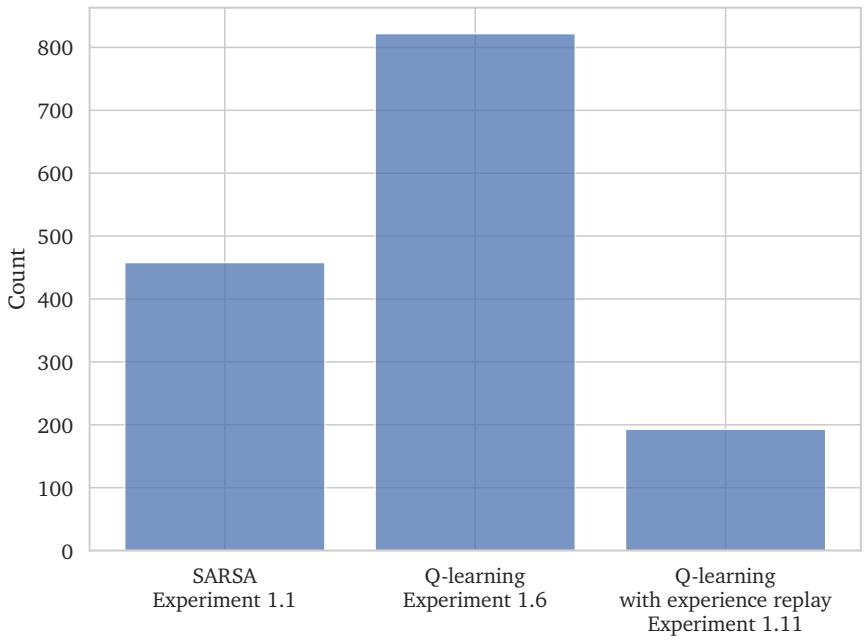

Figure 6.7 – Average count of the QoC not obeying its minimum. This figure shows the experiments 1.1, 1.6 and 1.11, as given in Table D.1

6.6 Reward Function Modification

The reward function has an important influence on the behavior of the DRL algorithm. It is used to influence the agent's decision by rewarding or punishing it depending on the chosen action. In this thesis, it is used to learn the policy $\pi(\hat{x}, \hat{Q})$, which maps the trade-off between the QoC and utilization. The given reward functions defined in Section 5.3 are able to achieve this goal. Furthermore, with the choice of the coefficients the algorithm is able to adjust how close the system will get to its minimum QoC. Figure 6.8 shows the change in QoC over several time steps. The reward was chosen to be r_{Ω_1} with the utilization parameter u set to one. The horizontal gray line signals the minimum QoC Ω_{min}. The relation between utilization and QoC can be mapped with the values of the two parameters u and $q1$. Increasing $q1$ results in higher priority for the QoC and therefore is further apart from its minimum. In contrast, however, a small $q1$ results in violating the minimum QoC barrier. Since the output for the algorithm depends on different factors, this modification does not yield to every parameter combination in general. Nevertheless, it is possible to prioritize either the QoC or the utilization rate, even if the values for the coefficients need to be evaluated empirically.

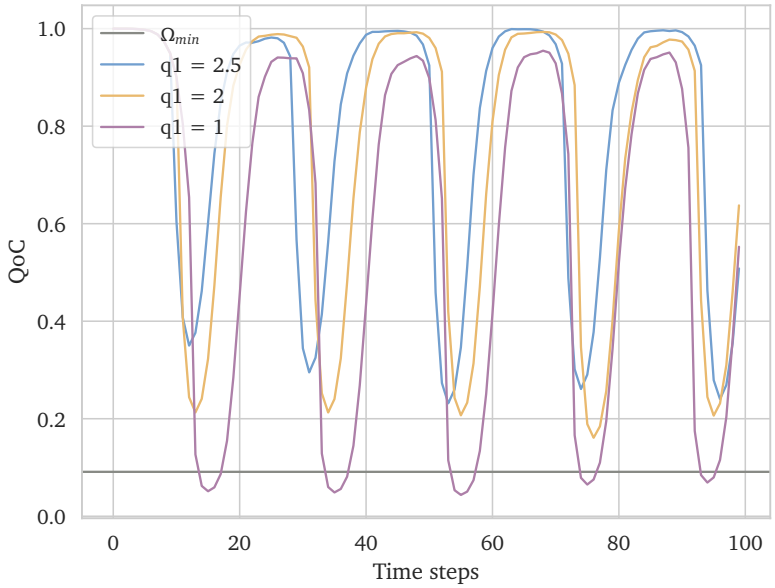

Figure 6.8 – This figure shows the changing of the QoC over 100 time steps. The distance to the minimum QoC increases by incrementing the QoC reward parameter $q1$.

6.7 Interpreting the Evaluation Metrics

This section provides an interpretation of the evaluated metric and execution statistics gathered through several experiments. The experiments are conducted in the environment, which was introduced in Chapter 3 and evaluated under the conditions listed in Appendix C.3. The highlighted

parameters in Appendix C.3 indicate the values to be evaluated. They were chosen because their modifications turned out to be most promising.

The best parameter combinations can be seen in Table D.1, Table D.2, and Table D.3, which were empirically gathered after several test runs. The learning parameter combinations are not the same for all the tables because they differ in process and measurement noise. Their interpretation and comparison is given in Section 6.7.1. Table D.4 shows the evaluation metrics achieved by the different policies proposed in Rheinfels' [Rhe19, p. 74] and this thesis. The evaluated metric and execution statistics of all policies are explained and interpreted in Section 6.7.2.

6.7.1 Process and Measurement Noise

First, the difference between the system models *inverted pendulum* and *double integrator* will be analyzed. Both system models behave differently, especially when seeing how fast the system is moving away from its equilibrium, without actuating. In all three settings, given the inverted pendulum system model, the SARSA algorithm is able to outperform the Q-learning approach by the overall evaluation metric m. This is particularly well illustrated in Table D.2 and Table D.3, where only SARSA is able to achieve 1.0 in the bound metric m_b. The excellent performance of SARSA may refer to its conservative behavior not to take actions, which might result in losing rewards. The double integrator is able to stay in its equilibrium state much longer before actuation is needed. This might be a reason for the Q-learning approach to outperform SARSA in this case. Furthermore, the double integrator is more susceptible to catastrophic forgetting, so a TN, early stopping and a more explicit reward is used in some experiments.

Comparing the different algorithms used in the experiments leads to the following assumptions. Although SARSA is performing better with the inverted pendulum it takes more epochs to learn and therefore more over all execution time. As mentioned before in Section 6.5 this leads to a higher number of QoC violation. Furthermore, the Q-learning approach can be optimized using the experience replay buffer, which might lead to better performance when trying out more parameter combinations and other reward functions.

The chosen reward functions are used as defined in Section 5.3. The best results are achieved using the Ω_1 reward for no measurement noise, might depend on the simplicity of the function. Therefore, it is much easier to balance the QoC and utilization, as explained in Section 6.6. However, it is possible that more complicated reward functions might lead to better results, as provided in Table D.2.

The hyperparameter α and γ are in the expected range. α is higher for the inverted pendulum because it has a faster decrease in QoC and therefore needs to speed up learning to cumulate reward. γ is high for both system models, because the agent needs to consider future rewards with greater weight. In this case, the agent needs better long-term planning to avoid the system from fall below the minimum QoC bound.

As expected, the overall evaluation metric m decreases, when increasing the process and measurement noise, because there is more need to actuate to counteract the decrease of the QoC.

Given the three tables with the metric evaluation information, it can be seen that DRL is a valid approach to trade-off utilization against QoC in this environment setting. In all tables but 1, there is

at least one experiment, which is able to not violate the minimum QoC (Ω_{min}) over all evaluation runs and leaving out actuations. The exception is the *double integrator* in Table D.3, which is only able to achieve a value of 0.9167 in terms of m_b.

6.7.2 Policies

In this section, the interpretation for the metric and execution statistics from the static controller π_{static}, PI policy π_{PI} [Rhe19, pp. 49–52], the data generator π_{DG} [Rhe19, pp. 28–30],the SL policy π_{SLP} [Rhe19, pp. 25–27], and DRL policy π_{DRL} is provided. There are some differences in the implementation of the environment for the π_{DRL}, which must be pointed out before the policies can be compared against each other. The π_{DRL} metric is evaluated under the circumstances of:

- A restricted model, with deterministic execution times
- Evaluation and training on other hardware
- A fixed covariance matrix N_k.

Despite these limitations, a comparison can be made because the calculation error of N_K is minimal[9] and the different hardware[10] only effects the execution times. The deterministic execution time must be kept in mind because it can lead to slightly different results[11].

The static controller (experiments 4.1 and 4.6) maximized the QoC metric m_Ω as well as the bound metric m_b. It also minimized all execution time statics. For the inverted pendulum, the DRL policy π_{DRL} is the only one able to reach a bound metric m_b of one. Furthermore, it's the policy with the highest overall evaluation metric m. In comparison to the other three policies, the QoC metric m_Ω is higher than the utility metric m_u. This ratio is adjustable up to a certain degree, as presented in Section 6.6, but does not reach a small value like 0.2192 for m_Ω. The DRL policy π_{DRL} is reaching an overall evaluation metric m close to the PI policy π_{PI}, but can to reach a bound metric m_b of one for this system.

Taking a look at the execution time statistics reveal that average execution time of the DRL policy π_{DRL} is faster than the SL policy π_{SLP} and a little slower than the PI policy π_{PI}. These values should be taken with caution because they are measured on different hardware.

In conclusion, the following can be drawn from this comparison:

- Given these system models the DRL algorithm demonstrates to be a viable method for the trade-off utilization against QoC and leads to better results that the SL approach.
- The DRL is able to adjust how close the system can get to its minimum QoC.
- Up to a certain value, the DRL can deal with increased process and measurement noise.
- The NN employed by the DRL is able to reduce the average execution significantly by leaving out unnecessary actuations.

[9] Proof and error calculation is given in Appendix A.1.
[10] Specification for the training and evaluation hardware is provided in Appendix D.
[11] A comparison example is given in Appendix A.2.

7 RELATED WORK

Learning policies that improve the trade-off between QoC and utilization have been addressed in several works. Lozoya et al. analyze these control performance and resource utilization trade-offs [Loz+13]. The focus of this paper lies on the implementation of feasible "resource/performance-aware policies", which deviate from the standard periodic sampling approach [ÅW90, p. 66]. To that end they take a look at different embedded control systems, including several control loops [Loz+13, p. 270]. These policies are divided into feedback scheduling and event-driven control. In event-driven control the task period is adjusted by the controllers in direct response to the application's performance [Loz+13, p. 272]. Feedback scheduling involves choosing a suitable sampling period using feedback from scheduling among other considerations. The key point is to improve the overall control performance of all tasks by making optimal use of limited resources [Loz+13, p. 270].

In Rheinfels' thesis, a controller implemented on a RTOS and disturbances, which degrades the QoC are given [Rhe19, pp. 7–10]. A state estimate gathered from the controller is influencing the policy, which decided to activate a job or not. The policy is enforced by a lock-driven scheduler [Rhe19, p. 16]. He worked out several experiments in which he could show that the SL policy is a viable choice to achieve a good trade-off between QoC and utilization under several restrictive assumptions [Rhe19, p. 53]. This thesis provides an improvement, of the theoretic RL approach by Rheinfels [Rhe19, pp. 43–48] on the same simulated system.

In a study by Sander et al. RL with ER was used on real-time control experiments involving a pendulum swing-up problem and a vision-based goalkeeper robot [ABB12, p. 201]. They demonstrate, that ER is able to improve the RL algorithm by presenting the same available data multiple times [ABB12, pp. 209–211]. In the given experiments ER was able to outperform the TDL approaches SARSA and Q-learning. The experiments were performed using a general framework, which was provided to make ER available with any RL technique [ABB12, pp. 203–205]. Although, this work is not focusing on the trade-off between QoC and utilization it shows, that in real-time control experiments an ER is able to outperform the classic TDL approaches.

For unknown continuous non-linear systems with control restrictions, Xiong et al. provided an effective technique based on the RL approaches. Three NNs are used in this paper. A recurrent NN to identify the unknown dynamical system and two feed-forward NNs used to implement the actor-critic method. The critic was chosen to approximate the optimal control under optimal costs [YLW14, p. 558]. Furthermore, the weights of the action NN and the critic NN are guaranteed to be uniformly ultimately bounded when utilizing Lyapunov's direct technique, while the closed-loop

system remains stable [YLW14, p. 554].

Bin et al. used the integral quadratic constraints (IQCs) to analyze the stability of a feedback control loop [Qin+12, p. 993]. An analysis of the IQCs is performed on a feedback control loop that includes a PI controller, a mass-spring system, and a reinforcement-learning-trained recurrent network. This work only focusing on the stability of the system, but is even able to achieve this while learning [Qin+12, p. 997]. The algorithm is performed by using actor-critic, where the Q-function in the critic is represented as a table. This simplification could be made, because only the actor appears in the feedback loop [YLW14, p. 996]. Thus, the input of the critic has a small dimension.

A real-time autonomous control strategy for multi energy systems were proposed by Ye et al. The challenge of this setup is, that both the state of the environment and agent's actions are not only continuous but also multidimensional [Ye+20, p. 2]. They use a model-free and data driven DRL based approach, which is able to perform independently of system knowledge [Ye+20, pp. 8–10]. Furthermore, they optimize their approach by 1.7 times speed using prioritized experience replay [Ye+20, p. 12].

8 CONCLUSION AND OUTLOOK

Both the DRL approach and Rheinfels' SL policy were evaluated on the same benchmark systems: inverted pendulum (Appendix B.1) and double integrator (Appendix B.2). The results show DRL is a valid choice for the problem, given in Chapter 4, and outperforms the SL policy [Rhe19, pp. 25–28]. This better performance can be observed in the total metric as well as in the average QoC per experiment. The only disadvantage of the DRL policy is that it is not possible to achieve such a low utilization as the SL policy. The question is whether this is desirable since a higher average QoC makes the system more robust against noise and disturbances. Although both algorithms are able to decrease the utilization, the time it takes to execute the policy increases the overall execution time.

The DRL approach was implemented in a system described in Section 3.1. Its key element is the control loop, where the QoC is a way to indicate if the system is close to its equilibrium. The controller was able to counteract the imbalance caused by measurement and process noise. The conducted experiments in Table D.1, Table D.2, and Table D.3 give an overview of the different levels of disturbances the DQN is able to handle under the additional requirement of reducing the utilization.

This was achieved by using a MLP with two output neurons, which is able to predict the action with the highest prospects of success depending on the estimated state of the system and the QoC. A special NN was constructed, which takes the constraints on its execution time into account, but was still able to solve the task of modeling the trade-off. In tandem with the Q-learning and SARSA algorithms, the NN was able to handle the continues-state spaces given by the benchmark systems.

As previously mentioned, the DQN policy is able to decrease utilization and therefore minimizes costs by reducing the execution time. Furthermore, the detailed overview in Appendix B.3 reveals that leaving out the actuation saves close to 75% compared to the use of actuation. Even though, the cost of the actuation is the largest compared to the costs of measuring and computing, it is still a small part of the overall execution time. As given in Table D.4, the execution time for the prediction of the NN is higher than the time spared by omitting the actuation job. This results in the fact that the given system is kept very simple compared to real-world systems.

Even though the overall execution time was exacerbated by the additional costs of the prediction, the technique could still prove beneficial for larger, more complex systems or systems with optimized hardware for machine learning. Alcon et al. analyze the industrial-quality autonomous driving software framework Apollo in terms of its observed execution time variability, as well as the sources behind it [Alc+20]. It's a complex software, able to handle the real-time computation

of large amounts of data generated by ongoing operations. Furthermore, artificial intelligence algorithms, which bring additional complexity due to large inputs and non-deterministic behavior are involved. These advanced functions are used in critical domains, which implies a higher focus on software timing [Alc+20, p. 267]. Compared to the system used in this thesis, Apollo includes seven software modules able to perceive, predict, locate, map, plan, control, and passing control commands to the vehicle hardware [Alc+20, p. 269]. Leaving out the control task, and if possible, the control passing is more rewarding, than in simpler control systems.

In addition to more complex systems, hardware is also becoming increasingly adapted to machine learning. Capra et al. compare different hardware solutions for the computationally intensive machine learning approach and also gave an outlook of current and future hardware trends [Cap+20, p. 225171]. They assume that the focus in the future will be more on mobile and possibly wearable devices that are part of the IoT [Cap+20, p. 225171]. This leads to hardware limitations since there is need for smaller, more powerful and energy-efficient CPUs. A RTCS could also benefit from this development, as it might be possible to shorten the prediction time through hardware support.

Besides this outlook, there is also room for improvement in the DRL approach. The following provides ideas for future research directions and improvements:

- Optimizing the choice of hyper-parameters [Don+21]
- Only using the estimated state as inputs for the NN and calculating the QoC afterward
- Making ER available for both RL algorithms
- Optimizing the reward function
- Improving stability by using the actor-critic method
- Experimenting how the policy reacts to stronger or different disturbances.

It is very likely that the types of problems, which this approach still faces will be solved either by the availability of specialist hardware or the increased costs of superfluous activations in increasingly complex systems.

PROOFS AND ANALYSES

This part contains the error calculation of the two different covariance matrices as well as a metric and time analysis for deterministic and non-deterministic execution times.

A.1 Covariance Matrix N_k

Due to a flaw in the implementation, the covariance matrix $\tilde{N}_k$ in [Rhe19, p. 13] was calculated wrong. This section estimates the *absolute* and *relative error* between the wrong covariance matrix $\tilde{N}_k$ in [Rhe19, p. 13] and the covariance matrix N_k implemented in this thesis. Starting with the general derivation of N_k [Rhe19, p. 12]:

$$d_k = G(a_k)\nu_k \tag{A.1}$$

$$\mathbb{E}\{d_k d_k^T\} = G(a_k)\mathbb{E}\{\nu_k \nu_k^T\}G(a_k)^T \tag{A.2}$$

$$N_k = G(a_k)IG(a_k)^T = G(a_k)G(a_k)^T. \tag{A.3}$$

The difference between N_k and $\tilde{N}_k$ lies in the calculation of $G(a_k)$ resp. $\tilde{G}(a_k)$. $N_k^{1/2}$ means, that N_k is decomposed using the LDU decomposition [TB97, p. 77], provided in the following:

$$G(a_k) = N_k^{1/2} \tag{A.4}$$

$$N_k = LD^{\frac{1}{2}}D^{\frac{1}{2}}L^T \tag{A.5}$$

$$G_k = LD^{\frac{1}{2}}. \tag{A.6}$$

The error was done by equating $N_k^{1/2}$ and $\sqrt{N_k}$ resulting in the miscalculation:

$$\tilde{G}_k = \sqrt{N_k} \tag{A.7}$$

$$\tilde{N}_k = \tilde{G}(a_k)\tilde{G}(a_k)^T \tag{A.8}$$

$$N_k = G(a_k)G(a_k)^T. \tag{A.9}$$

The *absolute* and *relative errors* for every $k \in \{0, 1\}$ are calculated by using the formulas defined above [AS64, p. 14]. The vales for the matrices N_k can be looked up in Appendix B:
Inverted pendulum:

$$\|N_0 - \tilde{N}_0\|_2 = 3.288005457975112 \cdot 10^{-22} \tag{A.10}$$

$$\frac{\|N_0 - \tilde{N}_0\|_2}{\|N_0\|_2} = 2.558041071589027 \cdot 10^{-16} \tag{A.11}$$

$$\|N_1 - \tilde{N}_1\|_2 = 1.6837207743877178 \cdot 10^{-17} \tag{A.12}$$

$$\frac{\|N_1 - \tilde{N}_1\|_2}{\|N_1\|_2} = 1.5289322286752783 \cdot 10^{-15}. \tag{A.13}$$

Double integrator:

$$\|N_0 - \tilde{N}_0\|_2 = 1.6744507894869823 \cdot 10^{-20} \tag{A.14}$$

$$\frac{\|N_0 - \tilde{N}_0\|_2}{\|N_0\|_2} = 1.2281401160440878 \cdot 10^{-15} \tag{A.15}$$

$$\|N_1 - \tilde{N}_1\|_2 = 1.6744507894869823 \cdot 10^{-20} \tag{A.16}$$

$$\frac{\|N_1 - \tilde{N}_1\|_2}{\|N_1\|_2} = 1.2281401160440878 \cdot 10^{-15}. \tag{A.17}$$

A.2 Execution Times

Evaluation of metrics and execution times for the experiment 2.3 with deterministic and non-deterministic execution times[12]. This is just an example of how deterministic and non-deterministic execution times can differ in metric and execution times and do not yield every parameter combination in the system models.

[12] More information are provided in Table D.2.

Experiment 2.3	$avg.\ m_\Omega$	$avg.\ m_u$	$avg.\ m_b$	$avg.\ m$	$avg.\ e[s]$	$std.\ e[s]$	$min.\ e[s]$	$max.\ e[s]$
Deterministic execution times	0.7843 ± 0.0016	0.4041 ± 0.0022	1.0000 ± 0.0000	0.3171 ± 0.0020	$9.50 \cdot 10^{-7}$	$1.59 \cdot 10^{-7}$	$8.31 \cdot 10^{-7}$	$2.77 \cdot 10^{-6}$
Non-deterministic execution times	0.8592 ± 0.0013	0.3441 ± 0.0025	1.0000 ± 0.0000	0.2957 ± 0.0022	$1.38 \cdot 10^{-6}$	$3.17 \cdot 10^{-7}$	$9.27 \cdot 10^{-7}$	$2.39 \cdot 10^{-6}$

Table A.1 – Metrics and execution times for the experiment 2.3 with deterministic and non-deterministic execution times.

B SYSTEM MODELS

To evaluate the Reinforcement Learning approach, two different models were assessed. Both predesigned in [Rhe19]: *double integrator* and *inverted pendulum*. As mentioned before, the model is simplified, and some matrices only state the result of more complex calculations.

A_k is the matrix describing the continuous dynamics according to (2.17) and combining $A_{m,c,a}$ (measure, compute, actuate) given in (table 2.1) [Rhe19, pp. 10–12]. The matrix G_k is the fault input of the extended model in formula (2.14) of [Rhe19, p. 10]. According to (2.17) it is built out of G_p, where p stands for plant, and the fault input is given in (2.7) [Rhe19, pp. 7–10]. N_k is then built out of (2.20b), (2.27), and (2.28) [Rhe19, pp. 11–13]. The simplification for the QoC J_k is derived from formula (2.12) [Rhe19, p. 10].

B.1 Inverted Pendulum

This section is providing the values for the matrices used for describing the inverted pendulum system model.

$$A_0 = \begin{bmatrix} 1.1670 & 1.5842 \cdot 10^{-1} & 0.0000 & 0.0000 & 0.0000 & 5.4139 \cdot 10^{-3} \\ 1.5636 & 6.6955 \cdot 10^{-1} & 0.0000 & 0.0000 & 0.0000 & 5.0694 \cdot 10^{-2} \\ 0.0000 & 0.0000 & 1.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 1.0000 & 0.0000 & 0.0000 \\ 1.0013 & 1.6245 \cdot 10^{-2} & 0.0000 & 0.0000 & 0.0000 & 4.3689 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 1.0000 \end{bmatrix} \tag{B.1}$$

$$A_1 = \begin{bmatrix} 1.1668 & 1.5842 \cdot 10^{-1} & 1.2782 \cdot 10^{-4} & -1.1648 \cdot 10^{-5} & 0.0000 & 5.4122 \cdot 10^{-3} \\ 1.4521 & 6.6774 \cdot 10^{-1} & 7.6559 \cdot 10^{-2} & -6.9766 \cdot 10^{-3} & 0.0000 & 4.9628 \cdot 10^{-2} \\ 1.2357 & 2.0047 \cdot 10^{-2} & -4.0110 \cdot 10^{-1} & 9.3900 \cdot 10^{-2} & 0.0000 & 5.3912 \cdot 10^{-5} \\ 2.4223 & 3.9298 \cdot 10^{-2} & -3.9840 & 6.5800 \cdot 10^{-2} & 0.0000 & 1.0568 \cdot 10^{-4} \\ 1.0013 & 1.6245 \cdot 10^{-2} & 0.0000 & 0.0000 & 0.0000 & 4.3689 \cdot 10^{-5} \\ -1.0504 \cdot 10^{-2} & -1.7041 & 7.2149 \cdot 10^{\ 1} & -6.5748 & 0.0000 & -4.5830 \cdot 10^{-3} \end{bmatrix} \tag{B.2}$$

$$N_0 = \begin{bmatrix} 1.8417 \cdot 10^{-8} & 1.2548 \cdot 10^{-7} & 0.0000 & 0.0000 & 2.1117 \cdot 10^{-10} & 0.0000 \\ 1.2548 \cdot 10^{-7} & 1.2729 \cdot 10^{-6} & 0.0000 & 0.0000 & 9.1850 \cdot 10^{-10} & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 2.1117 \cdot 10^{-10} & 9.1850 \cdot 10^{-10} & 0.0000 & 0.0000 & 1.0000 \cdot 10^{-6} & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \end{bmatrix} \tag{B.3}$$

$$N_1 = \begin{bmatrix} 1.8417 \cdot 10^{-8} & 1.2548 \cdot 10^{-7} & 3.1251 \cdot 10^{-11} & 6.1261 \cdot 10^{-11} & 2.5325 \cdot 10^{-11} & -2.6566 \cdot 10^{-9} \\ 1.2548 \cdot 10^{-7} & 1.2851 \cdot 10^{-6} & -1.3623 \cdot 10^{-7} & -2.6704 \cdot 10^{-7} & -1.1039 \cdot 10^{-7} & 1.1580 \cdot 10^{-6} \\ 3.1251 \cdot 10^{-11} & -1.3623 \cdot 10^{-7} & 1.5228 \cdot 10^{-6} & 2.9851 \cdot 10^{-6} & 1.2340 \cdot 10^{-6} & -1.2945 \cdot 10^{-4} \\ 6.1261 \cdot 10^{-11} & -2.6704 \cdot 10^{-7} & 2.9851 \cdot 10^{-6} & 5.8516 \cdot 10^{-6} & 2.4190 \cdot 10^{-6} & -2.5375 \cdot 10^{-4} \\ 2.5325 \cdot 10^{-11} & -1.1039 \cdot 10^{-7} & 1.2340 \cdot 10^{-6} & 2.4190 \cdot 10^{-6} & 1.0000 \cdot 10^{-6} & -1.0490 \cdot 10^{-4} \\ -2.6566 \cdot 10^{-9} & 1.1580 \cdot 10^{-5} & -1.2945 \cdot 10^{-4} & -2.5375 \cdot 10^{-4} & -1.0490 \cdot 10^{-4} & 1.1004 \cdot 10^{-2} \end{bmatrix} \tag{B.4}$$

$$\hat{Q} = \begin{bmatrix} Q_p & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & R_p \end{bmatrix}, \quad Q_p = \begin{bmatrix} 550 & 0 \\ 0 & 550 \end{bmatrix}, \quad R_p = 0.8. \tag{B.5}$$

The minimum QoC (Ω_{min}) bound was set to allow a maximum angle of 5 °and setting the target to 80%, i.e. 4 °. More details can be found in [Rhe19, p. 69].

$$\Omega_{min} \approx 0.0911. \tag{B.6}$$

B.2 Double Integrator

This section is providing the values for the matrices used for describing the double integrator system model. The formula for N_k and $\hat{Q}$ are omitted, because they follow the same calculation formula.

$$A_0 = \begin{bmatrix} 1.0000 & -4.256 & 0.0000 & 0.0000 & 0.0000 & 9.0568 \\ 0.0000 & 1.0000 & 0.0000 & 0.0000 & 0.0000 & -4.256 \\ 0.0000 & 0.0000 & 1.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 1.0000 & 0.0000 & 0.0000 \\ 1.0000 & -0.3547 & 0.0000 & 0.0000 & 0.0000 & 0.0629 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 1.0000 \end{bmatrix} \tag{B.7}$$

$$A_1 = \begin{bmatrix} 9.9988 \cdot 10^{-1} & -4.2560 & 1.2752 \cdot 10^{-4} & 1.2214 \cdot 10^{-4} & 0.0000 & 9.0542 \\ 3.4736 \cdot 10^{-3} & 9.9877 \cdot 10^{-1} & -3.5954 \cdot 10^{-3} & -3.4439 \cdot 10^{-3} & 0.0000 & -4.1848 \\ 1.3980 \cdot 10^{-2} & -4.9582 \cdot 10^{-1} & -5.4920 \cdot 10^{-1} & -2.2350 & 0.0000 & 8.7926 \cdot 10^{-2} \\ -1.1480 \cdot 10^{-1} & 4.0716 \cdot 10^{-2} & 1.8600 \cdot 10^{-1} & 5.0300 \cdot 10^{-2} & 0.0000 & -7.2203 \cdot 10^{-3} \\ 1.0000 & -3.5467 \cdot 10^{-1} & 0.0000 & 0.0000 & 0.0000 & 6.2894 \cdot 10^{-2} \\ -4.8970 \cdot 10^{-2} & 1.7368 \cdot 10^{-2} & 5.0687 \cdot 10^{-2} & 4.8551 \cdot 10^{-2} & 0.0000 & -3.0799 \cdot 10^{-3} \end{bmatrix} \tag{B.8}$$

$$N_0 = \begin{bmatrix} 1.2076 \cdot 10^{-5} & -4.2560 \cdot 10^{-6} & 0.0000 & 0.0000 & 1.2229 \cdot 10^{-7} & 0.0000 \\ -4.2560 \cdot 10^{-6} & 2.0000 \cdot 10^{-6} & 0.0000 & 0.0000 & -2.9556 \cdot 10^{-8} & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 1.2229 \cdot 10^{-7} & -2.9556 \cdot 10^{-8} & 0.0000 & 0.0000 & 1.0070 \cdot 10^{-6} & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \end{bmatrix} \tag{B.9}$$

$$N_1 = \begin{bmatrix} 1.2076 \cdot 10^{-5} & -4.2556 \cdot 10^{-6} & 1.7079 \cdot 10^{-7} & -1.4025 \cdot 10^{-8} & 1.2217 \cdot 10^{-7} & -5.9827 \cdot 10^{-9} \\ -4.2556 \cdot 10^{-6} & 1.9998 \cdot 10^{-6} & -3.6429 \cdot 10^{-8} & 2.9914 \cdot 10^{-9} & -2.6058 \cdot 10^{-8} & 1.2760 \cdot 10^{-9} \\ 1.7079 \cdot 10^{-7} & -3.6429 \cdot 10^{-8} & 1.9681 \cdot 10^{-6} & -1.6161 \cdot 10^{-7} & 1.4078 \cdot 10^{-6} & -6.8938 \cdot 10^{-8} \\ -1.4025 \cdot 10^{-8} & 2.9914 \cdot 10^{-9} & -1.6161 \cdot 10^{-7} & 1.3271 \cdot 10^{-8} & -1.1560 \cdot 10^{-7} & 5.6610 \cdot 10^{-9} \\ 1.2217 \cdot 10^{-7} & -2.6058 \cdot 10^{-8} & 1.4078 \cdot 10^{-6} & -1.1560 \cdot 10^{-7} & 1.0070 \cdot 10^{-6} & -4.9312 \cdot 10^{-9} \\ -5.9827 \cdot 10^{-9} & 1.2760 \cdot 10^{-9} & -6.8938 \cdot 10^{-8} & 5.6610 \cdot 10^{-9} & -4.9312 \cdot 10^{-8} & 2.4148 \cdot 10^{-9} \end{bmatrix} \tag{B.10}$$

$$Q_p = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad R_p = 1. \tag{B.11}$$

The minimum QoC (Ω_{min}) are determined by setting maximum and target control error to 1 and 80% respectively. More details can be found in [Rhe19, p. 70].

$$\Omega_{min} \approx 0.5000. \tag{B.12}$$

B.3 Cost Calculation

The following formulas describe, how the cost function L_k is calculated, with period p = 0.2.

Task	φ	D	e_{mean}
meassure	0	$\frac{1}{10}p$	$(\frac{1}{15}p + \frac{1}{10}p)/2$
compute	$\frac{1}{10}p$	$\frac{1}{2}p$	$(\frac{1}{8}p + \frac{1}{6}p)/2$
actuate	$\frac{9}{10}p$	p	$(\frac{1}{15}p + \frac{1}{10}p)/2$

Table B.1 – Temporal behavior of the control tasks [Rhe19, p. 70].

$$l_{me} \approx 0.0167 \tag{B.13}$$

$$l_{co} \approx 0.0492 \tag{B.14}$$

$$l_{ac} \approx 0.1967 \tag{B.15}$$

$$L(0) = \frac{l_{me} + l_{co}}{l_{me} + l_{co} + l_{ac}} \approx 0.2508 \tag{B.16}$$

$$L(1) = \frac{l_{me} + l_{co} + l_{ac}}{l_{me} + l_{co} + l_{ac}} = 1. \tag{B.17}$$

C PARAMETER SETS

This section provides the parameter sets for different evaluations and experiments. They consist of fixed parameters with given values or methods and parameters to be evaluated. The latter are marked blue.

C.1 Parameters for the Target Network Evaluation

Parameter description	Parameter	Value (options)
System model	-	inverted pendulum
Algorithm	-	SARSA
Learning rate	α	0.7
Discount factor	γ	0.95
Initial epsilon greedy probability	ϵ	1.0
Epsilon greedy decay rate	ϵ_{decay}	0.9
Weight initialisation	-	He
Reward type	r	Ω_2, $u = 2,\ q2 = 1,\ q3 = 0.01$
Loss function	δ	1.5
Target network update frequency	C	evaluated (0,10)
Experience replay buffer size	N	-
Mini batch size	D	-

Table C.1 – Parameter set for the evaluation of the target network. The noise covariance matrices amount are set to $H = 10^{-3}$ and $N_p = 10^{-6}$ respectively.

C.2 Parameters for ER and Weight Evaluation

Parameter description	Parameter	Value (options)
System model	-	inverted pendulum
Algorithm	-	Q-Learning
Learning rate	α	0.7
Discount factor	γ	0.95
Initial epsilon greedy probability	ϵ	1.0
Epsilon greedy decay rate	ϵ_{decay}	0.9
Weight initialisation	-	evaluated (zero, He, Xavier)
Reward type	r	Ω_1, $u = 2,\ q1 = 1$
Loss function	δ	1.5
Target network update frequency	C	1
Experience replay buffer size	N	evaluated (0, 100)
Mini batch size	D	10

Table C.2 – Parameter set for the evaluation of the optimization methods. The noise covariance matrices amount are set to $H = 10^{-3}$ and $N_p = 0$ respectively.

C.3 Parameters for the Main Evaluation

Parameter description	Parameter	Value
System model	-	evaluated
Algorithm	-	evaluated
Learning rate	α	evaluated
Discount factor	γ	evaluated
Initial epsilon greedy probability	ϵ	1.0
Epsilon greedy decay rate	ϵ_{decay}	evaluated
Weight initialisation	-	He
Reward type	r	evaluated
Loss function	δ	1.5
Target network update frequency	C	evaluated
Experience replay buffer size	N	evaluated
Mini batch size	D	10
Epochs	E	evaluated
Time steps	T	evaluated

Table C.3 – Parameter set for the main evaluation, with cells marked in colour represent the values to be evaluated.

EXPERIMENTS

D

This section provides experiments for the DQN algorithm, where different learning parameter combinations are used for both system models. An overview of the evaluated parts of the DQN policy can be found in Table C.3. Due to the enormous amount of parameter combinations as mentioned in Section 6.3, it is not possible to evaluate every variety. After running multiple experiments, the best combinations can be found in Table D.1, Table D.2, and Table D.3. They only differ in process and measurement noise. Starting with process noise in Table D.1, adding measurement noise in Table D.2 and increasing the process noise in Table D.3. Table D.4 shows the evaluation metrics achieved by the different policies proposed in Rheinfels's [Rhe19, p. 74] and this thesis.

It was taken care not to use the same seed for training[13] and evaluation[14]. Besides the values of the evaluation metric, execution times of the policy were measured. The execution times indicate how long the evaluation takes to make a decision whether to actuate or not. Including their sample mean and standard deviation. In addition, the resulting metric values were complemented by their 95% confidence intervals of the sample mean [Dek+05a, p. 3], as Rheinfels did [Rhe19, p. 74], to ensure comparability. Care has been taken to evaluate under the same conditions as given in the previously mentioned master thesis.

[13] Wall-clock time on the test system: Intel Core i5-5200U, Debian with kernel version 4.19.0, niceness -10, gcc 8.3.0 using -O3, C++ implementation utilizing Eigen 3.4 [Eig] and framework RlQuaRTSv2 (commit: 2ca0ba43a6edcb71c5a2704d3ad0c5b8b249c46).

[14] Wall-clock time on the test system: AMD EPYC 7702 64-Core, Debian with kernel version 5.10.0, niceness -10, gcc 8.3.0 using -O3, C++ implementation utilizing Eigen 3.4 [Eig] and framework RlQuaRTSv2 (commit: 2ca0ba43a6edcb71c5a2704d3ad0c5b8b249c46).

Experiment	Models	Algorithm	α/γ	ϵ_{decay}	$r/u/q_i/<q_j>^a$	U	σ	E	T	$avg.\ m_\Omega$	$avg.\ m_u$	$avg.\ m_b$	$avg.\ m$	$avg.\ e[s]$	$std.\ e[s]$	$min.\ e[s]$	$max.\ e[s]$
Experiment 1.1	Inverted Pendulum	SARSA	0.7/0.95	0.9	$\Omega_1/2/1$	–	–	1000	500	0.8993	0.4537	1.0000	0.4080	$1.19 \cdot 10^{-6}$	$1.06 \cdot 10^{-7}$	$9.82 \cdot 10^{-7}$	$1.56 \cdot 10^{-6}$
Experiment 1.2	Inverted Pendulum	SARSA	0.7/0.95	0.9	$\Omega_2/2/1/0.1$	–	–	1000	500	0.7757	0.4553	1.0000	0.3533	$1.16 \cdot 10^{-6}$	$9.80 \cdot 10^{-8}$	$9.82 \cdot 10^{-7}$	$1.54 \cdot 10^{-6}$
Experiment 1.3	Inverted Pendulum	SARSA	0.7/0.95	0.9	$\Omega_2/2/1/0.1$	–	–	1000	500	0.8019	0.4577	1.0000	0.3671	$1.14 \cdot 10^{-6}$	$1.14 \cdot 10^{-7}$	$9.52 \cdot 10^{-7}$	$1.57 \cdot 10^{-6}$
Experiment 1.4	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_1/2/1$	–	–	1000	500	0.6234	0.4447	0.98883	0.2742	$1.17 \cdot 10^{-6}$	$5.20 \cdot 10^{-7}$	$9.71 \cdot 10^{-7}$	$1.14 \cdot 10^{-5}$
Experiment 1.5	Inverted Pendulum	Q-Learning	0.7/0.5	0.9	$\Omega_1/2/1$	–	–	1000	500	0.0895	0.4465	0.4459	0.0178	$1.73 \cdot 10^{-6}$	$1.12 \cdot 10^{-6}$	$9.61 \cdot 10^{-7}$	$1.36 \cdot 10^{-5}$
Experiment 1.6	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_1/2/1$	–	–	2000	500	0.7231	0.4508	1.0000	0.3261	$1.18 \cdot 10^{-6}$	$1.09 \cdot 10^{-7}$	$1.00 \cdot 10^{-6}$	$1.67 \cdot 10^{-6}$
Experiment 1.7	Inverted Pendulum	Q-Learning	0.7/0.95	0.99	$\Omega_1/2/1$	–	–	2000	500	0.7171	0.4505	1.0000	0.3231	$1.17 \cdot 10^{-6}$	$1.16 \cdot 10^{-7}$	$1.00 \cdot 10^{-6}$	$1.60 \cdot 10^{-6}$
Experiment 1.8	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_1/2/1$	–	–	2000	1000	0.7509	0.4527	1.0000	0.3399	$1.17 \cdot 10^{-6}$	$1.08 \cdot 10^{-7}$	$9.92 \cdot 10^{-7}$	$1.79 \cdot 10^{-6}$
Experiment 1.9	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_3/1/1$	–	10	2000	500	0.7450	0.4526	1.0000	0.3373	$1.18 \cdot 10^{-6}$	$1.19 \cdot 10^{-7}$	$9.81 \cdot 10^{-7}$	$1.54 \cdot 10^{-6}$
Experiment 1.10	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_3/1/0.1$	–	10	2000	500	0.7800	0.4557	1.0000	0.3556	$1.19 \cdot 10^{-6}$	$1.31 \cdot 10^{-7}$	$9.61 \cdot 10^{-7}$	$1.53 \cdot 10^{-6}$
Experiment 1.11	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_1/2/1$	–	100	200	500	0.7364	0.4517	1.0000	0.3327	$1.13 \cdot 10^{-6}$	$1.23 \cdot 10^{-7}$	$9.61 \cdot 10^{-7}$	$1.52 \cdot 10^{-6}$
Experiment 1.12	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_3/1/1$	–	100	200	500	0.7296	0.4511	1.0000	0.3292	$1.13 \cdot 10^{-6}$	$1.15 \cdot 10^{-7}$	$9.71 \cdot 10^{-7}$	$1.66 \cdot 10^{-6}$
Experiment 1.13	Double Integrator	SARSA	0.4/0.95	0.9	$\Omega_1/2/1$	–	–	500	500	0.4361	0.6480	0.6472	0.1861	$1.40 \cdot 10^{-6}$	$7.27 \cdot 10^{-7}$	$7.66 \cdot 10^{-7}$	$4.12 \cdot 10^{-6}$
Experiment 1.14	Double Integrator	SARSA	0.4/0.9	0.9	$\Omega_4/0.01/10$	–	–	100	500	0.9668	0.3976	0.9963	0.3810	$8.95 \cdot 10^{-7}$	$5.70 \cdot 10^{-8}$	$8.04 \cdot 10^{-7}$	$1.28 \cdot 10^{-6}$
Experiment 1.15	Double Integrator	SARSA	0.4/0.9	0.9	$\Omega_4/0.01/10$	10	–	200	500	0.8796	0.6383	0.97101	0.5509	$1.14 \cdot 10^{-6}$	$1.09 \cdot 10^{-7}$	$9.72 \cdot 10^{-7}$	$1.55 \cdot 10^{-6}$
Experiment 1.16	Double Integrator	Q-Learning	0.4/0.9	0.9	$\Omega_1/2/1$	–	–	200	500	0.4395	0.6469	0.6495	0.1878	$1.57 \cdot 10^{-6}$	$7.93 \cdot 10^{-7}$	$9.91 \cdot 10^{-7}$	$6.00 \cdot 10^{-6}$
Experiment 1.17	Double Integrator	Q-Learning	0.7/0.95	0.9	$\Omega_1/2/1$	–	–	200	500	0.6709	0.6198	0.9306	0.3924	$1.18 \cdot 10^{-6}$	$2.80 \cdot 10^{-7}$	$9.92 \cdot 10^{-7}$	$6.03 \cdot 10^{-6}$
Experiment 1.18	Double Integrator	Q-Learning	0.4/0.9	0.95	$\Omega_1/2/1$	–	–	300	500	0.7003	0.6214	0.9535	0.4207	$9.72 \cdot 10^{-7}$	$2.54 \cdot 10^{-7}$	$8.01 \cdot 10^{-7}$	$2.61 \cdot 10^{-6}$
Experiment 1.19	Double Integrator	Q-Learning	0.4/0.9	0.9	$\Omega_4/0.1/10$	–	–	100	500	0.9334	0.5252	0.9886	0.4828	$1.04 \cdot 10^{-6}$	$6.23 \cdot 10^{-8}$	$9.61 \cdot 10^{-7}$	$1.48 \cdot 10^{-6}$
Experiment 1.20	Double Integrator	Q-Learning	0.4/0.9	0.9	$\Omega_4/0.001/10$	–	–	200	500	0.9817	0.3127	0.9985	0.3051	$1.16 \cdot 10^{-6}$	$1.04 \cdot 10^{-7}$	$9.92 \cdot 10^{-7}$	$1.58 \cdot 10^{-6}$
Experiment 1.21	Double Integrator	Q-Learning	0.4/0.9	0.9	$\Omega_4/0.001/10$	–	–	500	500	0.4736	0.6515	0.6871	0.2233	$1.12 \cdot 10^{-6}$	$4.53 \cdot 10^{-7}$	$8.48 \cdot 10^{-7}$	$2.81 \cdot 10^{-6}$
Experiment 1.22	Double Integrator	Q-Learning	0.3/0.95	0.9	$\Omega_1/2/1$	50	50	500	500	0.9072	0.6107	1.0000	0.5538	$1.22 \cdot 10^{-6}$	$1.21 \cdot 10^{-7}$	$1.04 \cdot 10^{-6}$	$1.60 \cdot 10^{-6}$
Experiment 1.23	Double Integrator	Q-Learning	0.3/0.95	0.9	$\Omega_1/2/1$	100	100	500	500	0.8678	0.6072	1.0000	0.5272	$1.16 \cdot 10^{-6}$	$1.00 \cdot 10^{-7}$	$1.01 \cdot 10^{-6}$	$1.53 \cdot 10^{-6}$
Experiment 1.24	Double Integrator	Q-Learning	0.4/0.95	0.9	$\Omega_4/0.01/10$	100	100	500	500	0.8068	0.6087	0.9930	0.4884	$1.17 \cdot 10^{-6}$	$1.11 \cdot 10^{-7}$	$1.00 \cdot 10^{-6}$	$1.57 \cdot 10^{-6}$

Table D.1 – Results of the DQN policy evaluation, using different parameter combinations. std. and avg. denote the standard deviations and sample mean, respectively. The maximum metric for every system and minimum execution times are highlighted. Every evaluation run was executed in 1000 epochs with 2000 time steps. All times are given in seconds. Process and measurement noise of the environment are set to 10^{-3} and 0, respectively.

[a] $q_i \in \{q_1,\ q_2,\ q_4,\ d\}$ and $q_j \in \{q_3\}$

Experiment	Models	Algorithm	α/γ	ϵ_{decay}	$r/u/q_i/<q_j>$	U	σ	E	T	$avg.\ m_\Omega$	$avg.\ m_u$	$avg.\ m_b$	$avg.\ m$	$avg.\ e[s]$	$std.\ e[s]$	$min.\ e[s]$	$max.\ e[s]$
Experiment 2.1	Inverted Pendulum	SARSA	0.7/0.95	0.9	$\Omega_1/2/1$	–	–	1000	500	0.9546	0.0000	1.0000	0.0000	$1.18 \cdot 10^{-6}$	$1.26 \cdot 10^{-7}$	$9.92 \cdot 10^{-7}$	$1.55 \cdot 10^{-6}$
Experiment 2.2	Inverted Pendulum	SARSA	0.7/0.95	0.9	$\Omega_2/2/1/0.1$	10	–	1000	500	0.8093	0.3728	1.0000	0.3018	$1.16 \cdot 10^{-6}$	$4.37 \cdot 10^{-7}$	$8.93 \cdot 10^{-7}$	$6.10 \cdot 10^{-6}$
Experiment 2.3	Inverted Pendulum	SARSA	0.7/0.95	0.9	$\Omega_2/2/1/0.1$	10	–	2000	700	0.7805	0.3908	1.0000	0.3051	$1.99 \cdot 10^{-6}$	$2.22 \cdot 10^{-7}$	$1.10 \cdot 10^{-6}$	$2.57 \cdot 10^{-6}$
Experiment 2.4	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_1/2/1$	–	–	1000	500	0.6157	0.4472	0.9646	0.2658	$1.18 \cdot 10^{-6}$	$2.50 \cdot 10^{-7}$	$9.92 \cdot 10^{-7}$	$5.49 \cdot 10^{-6}$
Experiment 2.5	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_1/2/1$	–	–	2000	500	0.6163	0.4465	0.9650	0.2658	$1.15 \cdot 10^{-6}$	$2.41 \cdot 10^{-7}$	$9.61 \cdot 10^{-7}$	$5.36 \cdot 10^{-6}$
Experiment 2.6	Inverted Pendulum	Q-Learning	0.7/0.95	0.99	$\Omega_1/2/1$	–	–	2000	1000	0.4354	0.4826	0.7812	0.1646	$1.42 \cdot 10^{-6}$	$1.05 \cdot 10^{-6}$	$9.92 \cdot 10^{-7}$	$1.91 \cdot 10^{-5}$
Experiment 2.7	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_3/1/1$	–	10	3000	500	0.6739	0.4319	0.9969	0.2903	$1.18 \cdot 10^{-6}$	$1.26 \cdot 10^{-7}$	$9.82 \cdot 10^{-7}$	$2.09 \cdot 10^{-6}$
Experiment 2.8	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_3/1/0.1$	–	10	3000	500	0.5677	0.4563	0.8969	0.2326	$6.57 \cdot 10^{-7}$	$7.36 \cdot 10^{-7}$	$5.11 \cdot 10^{-7}$	$1.53 \cdot 10^{-5}$
Experiment 2.9	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_1/2/1$	–	200	300	500	0.6279	0.4442	0.9750	0.2721	$1.21 \cdot 10^{-6}$	$3.78 \cdot 10^{-7}$	$1.02 \cdot 10^{-6}$	$6.52 \cdot 10^{-6}$
Experiment 2.10	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_1/2/1$	–	200	500	500	0.6034	0.4502	0.9496	0.2582	$1.20 \cdot 10^{-6}$	$3.91 \cdot 10^{-7}$	$9.72 \cdot 10^{-7}$	$6.42 \cdot 10^{-6}$
Experiment 2.11	Inverted Pendulum	Q-Learning	0.7/0.99	0.9	$\Omega_1/2/1$	–	200	300	500	0.6003	0.4497	0.9464	0.2557	$1.18 \cdot 10^{-6}$	$1.99 \cdot 10^{-7}$	$9.91 \cdot 10^{-7}$	$2.83 \cdot 10^{-6}$
Experiment 2.12	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_3/1/1$	–	200	300	500	0.5539	0.4584	0.8732	0.2220	$1.30 \cdot 10^{-6}$	$7.09 \cdot 10^{-7}$	$1.00 \cdot 10^{-6}$	$1.35 \cdot 10^{-5}$
Experiment 2.13	Double Integrator	SARSA	0.4/0.95	0.9	$\Omega_1/2/1$	–	–	500	500	0.4362	0.6480	0.6472	0.1861	$1.44 \cdot 10^{-6}$	$8.00 \cdot 10^{-7}$	$1.00 \cdot 10^{-6}$	$5.56 \cdot 10^{-6}$
Experiment 2.14	Double Integrator	SARSA	0.4/0.9	0.9	$\Omega_4/0.01/10$	–	–	100	500	0.9436	0.4834	0.9915	0.4501	$1.20 \cdot 10^{-6}$	$3.05 \cdot 10^{-7}$	$9.62 \cdot 10^{-7}$	$5.88 \cdot 10^{-6}$
Experiment 2.15	Double Integrator	SARSA	0.4/0.9	0.9	$\Omega_4/0.01/10$	10	–	200	500	0.8626	0.6336	0.9531	0.5278	$1.17 \cdot 10^{-6}$	$3.82 \cdot 10^{-7}$	$9.62 \cdot 10^{-7}$	$5.68 \cdot 10^{-6}$
Experiment 2.16	Double Integrator	Q-Learning	0.4/0.9	0.9	$\Omega_1/2/1$	–	–	200	500	0.4380	0.6475	0.6485	0.1871	$1.39 \cdot 10^{-6}$	$5.67 \cdot 10^{-7}$	$9.62 \cdot 10^{-7}$	$6.10 \cdot 10^{-6}$
Experiment 2.17	Double Integrator	Q-Learning	0.7/0.95	0.9	$\Omega_1/2/1$	–	–	200	500	0.4362	0.6480	0.6472	0.1861	$1.48 \cdot 10^{-6}$	$8.43 \cdot 10^{-7}$	$9.72 \cdot 10^{-7}$	$1.24 \cdot 10^{-5}$
Experiment 2.18	Double Integrator	Q-Learning	0.4/0.9	0.95	$\Omega_1/2/1$	–	–	300	500	0.4375	0.6477	0.6482	0.1869	$1.42 \cdot 10^{-6}$	$9.59 \cdot 10^{-7}$	$9.52 \cdot 10^{-7}$	$1.90 \cdot 10^{-5}$
Experiment 2.19	Double Integrator	Q-Learning	0.4/0.9	0.9	$\Omega_4/0.01/10$	–	–	100	500	0.9323	0.5186	0.9880	0.4758	$1.20 \cdot 10^{-6}$	$2.90 \cdot 10^{-7}$	$1.00 \cdot 10^{-6}$	$5.25 \cdot 10^{-6}$
Experiment 2.20	Double Integrator	Q-Learning	0.4/0.9	0.9	$\Omega_4/0.001/10$	–	–	200	500	0.9776	0.3364	0.9981	0.3266	$1.15 \cdot 10^{-6}$	$9.82 \cdot 10^{-8}$	$9.71 \cdot 10^{-7}$	$1.57 \cdot 10^{-6}$
Experiment 2.21	Double Integrator	Q-Learning	0.4/0.9	0.9	$\Omega_4/0.001/10$	–	–	500	500	0.9743	0.3667	0.9977	0.3657	$1.14 \cdot 10^{-6}$	$1.25 \cdot 10^{-7}$	$9.72 \cdot 10^{-7}$	$1.55 \cdot 10^{-6}$
Experiment 2.22	Double Integrator	Q-Learning	0.3/0.95	0.9	$\Omega_1/2/1$	50	50	500	500	0.8002	0.6102	0.9996	0.4885	$1.17 \cdot 10^{-6}$	$1.28 \cdot 10^{-7}$	$9.82 \cdot 10^{-7}$	$1.65 \cdot 10^{-6}$
Experiment 2.23	Double Integrator	Q-Learning	0.3/0.95	0.9	$\Omega_1/2/1$	100	100	500	500	0.8821	0.6096	1.0000	0.5377	$1.15 \cdot 10^{-6}$	$1.20 \cdot 10^{-7}$	$9.91 \cdot 10^{-7}$	$1.59 \cdot 10^{-6}$
Experiment 2.24	Double Integrator	Q-Learning	0.4/0.95	0.9	$\Omega_4/0.01/10$	100	100	500	500	0.6527	0.6374	0.9209	0.3918	$1.16 \cdot 10^{-6}$	$2.82 \cdot 10^{-7}$	$9.61 \cdot 10^{-7}$	$6.04 \cdot 10^{-6}$

Table D.2 – Results of the DQN policy evaluation, using different parameter combinations. std. and avg. denote the standard deviations and sample mean, respectively. The maximum metric for every system and minimum execution times are highlighted. Every evaluation run was executed in 1000 epochs with 2000 time steps. All times are given in seconds. Process and measurement noise of the environment are set to 10^{-3} and 10^{-6}, respectively.

Experiment	Models	Algorithm	α/γ	ϵ_{decay}	$r/u/q_i/<q_j>$	U	σ	E	T	$avg.\ m_\Omega$	$avg.\ m_u$	$avg.\ m_b$	$avg.\ m$	$avg.\ e[s]$	$std.\ e[s]$	$min.\ e[s]$	$max.\ e[s]$
Experiment 3.1	Inverted Pendulum	SARSA	0.7/0.95	0.9	$\Omega_1/2/1$	–	–	1000	500	0.9363	0.0000	1.0000	0.0000	$1.17 \cdot 10^{-6}$	$1.36 \cdot 10^{-7}$	$9.61 \cdot 10^{-7}$	$1.57 \cdot 10^{-6}$
Experiment 3.2	Inverted Pendulum	SARSA	0.7/0.95	0.9	$\Omega_2/2/1/0.1$	10	–	1000	500	0.6761	0.4118	0.9974	0.2778	$1.14 \cdot 10^{-6}$	$1.11 \cdot 10^{-7}$	$9.82 \cdot 10^{-7}$	$1.55 \cdot 10^{-6}$
Experiment 3.3	Inverted Pendulum	SARSA	0.7/0.95	0.9	$\Omega_2/2/1/0.1$	10	–	2000	700	0.7545	0.3736	1.0000	0.2819	$1.05 \cdot 10^{-6}$	$6.95 \cdot 10^{-8}$	$9.42 \cdot 10^{-7}$	$1.41 \cdot 10^{-6}$
Experiment 3.4	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_1/2/1$	–	–	1000	500	0.6350	0.4259	0.9806	0.2654	$1.22 \cdot 10^{-6}$	$2.96 \cdot 10^{-7}$	$9.72 \cdot 10^{-7}$	$5.57 \cdot 10^{-6}$
Experiment 3.5	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_1/2/1$	–	–	2000	500	0.6063	0.4350	0.9595	0.2533	$1.20 \cdot 10^{-6}$	$3.24 \cdot 10^{-7}$	$9.82 \cdot 10^{-7}$	$5.35 \cdot 10^{-6}$
Experiment 3.6	Inverted Pendulum	Q-Learning	0.7/0.95	0.99	$\Omega_1/2/1$	–	–	3000	1000	0.5084	0.4346	0.8488	0.1874	$1.44 \cdot 10^{-6}$	$7.67 \cdot 10^{-7}$	$1.02 \cdot 10^{-6}$	$6.83 \cdot 10^{-6}$
Experiment 3.7	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_3/1/1$	–	10	3000	500	0.6503	0.4230	0.9907	0.2727	$1.17 \cdot 10^{-6}$	$2.72 \cdot 10^{-7}$	$9.91 \cdot 10^{-7}$	$5.46 \cdot 10^{-6}$
Experiment 3.8	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_3/1/0.1$	–	10	3000	500	0.5834	0.4415	0.9296	0.2396	$1.22 \cdot 10^{-6}$	$4.49 \cdot 10^{-7}$	$9.82 \cdot 10^{-7}$	$5.87 \cdot 10^{-6}$
Experiment 3.9	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_1/2/1$	–	200	300	500	0.6449	0.4224	0.9880	0.2693	$1.21 \cdot 10^{-6}$	$2.60 \cdot 10^{-7}$	$9.82 \cdot 10^{-7}$	$6.04 \cdot 10^{-6}$
Experiment 3.10	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_1/2/1$	–	200	500	500	0.5880	0.4402	0.9355	0.2424	$1.22 \cdot 10^{-6}$	$4.28 \cdot 10^{-7}$	$9.72 \cdot 10^{-7}$	$5.22 \cdot 10^{-6}$
Experiment 3.11	Inverted Pendulum	Q-Learning	0.7/0.99	0.9	$\Omega_1/2/1$	–	50	300	500	0.5794	0.4426	0.9236	0.2371	$1.25 \cdot 10^{-6}$	$4.71 \cdot 10^{-7}$	$9.82 \cdot 10^{-7}$	$5.95 \cdot 10^{-6}$
Experiment 3.12	Inverted Pendulum	Q-Learning	0.7/0.95	0.9	$\Omega_3/1/1$	–	200	300	500	0.5493	0.4485	0.8754	0.2159	$1.29 \cdot 10^{-6}$	$5.14 \cdot 10^{-7}$	$9.92 \cdot 10^{-7}$	$5.90 \cdot 10^{-6}$
Experiment 3.13	Double Integrator	SARSA	0.4/0.95	0.9	$\Omega_1/2/1$	–	–	500	500	0.4094	0.6122	0.6169	0.1554	$1.12 \cdot 10^{-6}$	$5.50 \cdot 10^{-7}$	$7.37 \cdot 10^{-7}$	$3.43 \cdot 10^{-6}$
Experiment 3.14	Double Integrator	SARSA	0.4/0.9	0.9	$\Omega_4/0.01/10$	–	–	100	500	0.7730	0.5081	0.9012	0.3556	$1.22 \cdot 10^{-6}$	$4.07 \cdot 10^{-7}$	$9.92 \cdot 10^{-7}$	$5.79 \cdot 10^{-6}$
Experiment 3.15	Double Integrator	SARSA	0.4/0.9	0.9	$\Omega_4/0.01/10$	10	–	200	500	0.6224	0.5930	0.8003	0.3086	$1.56 \cdot 10^{-6}$	$8.38 \cdot 10^{-7}$	$9.82 \cdot 10^{-7}$	$6.02 \cdot 10^{-6}$
Experiment 3.16	Double Integrator	Q-Learning	0.4/0.9	0.9	$\Omega_1/2/1$	–	–	200	500	0.4043	0.6104	0.6095	0.1512	$1.52 \cdot 10^{-6}$	$8.43 \cdot 10^{-7}$	$9.62 \cdot 10^{-7}$	$7.28 \cdot 10^{-6}$
Experiment 3.17	Double Integrator	Q-Learning	0.7/0.95	0.9	$\Omega_1/2/1$	–	–	200	500	0.4043	0.6104	0.6095	0.1512	$1.53 \cdot 10^{-6}$	$9.93 \cdot 10^{-7}$	$9.72 \cdot 10^{-7}$	$1.48 \cdot 10^{-5}$
Experiment 3.18	Double Integrator	Q-Learning	0.4/0.9	0.95	$\Omega_1/2/1$	–	–	300	500	0.4043	0.6104	0.6095	0.1512	$1.66 \cdot 10^{-6}$	$1.44 \cdot 10^{-6}$	$1.03 \cdot 10^{-6}$	$2.26 \cdot 10^{-5}$
Experiment 3.19	Double Integrator	Q-Learning	0.4/0.9	0.9	$\Omega_4/0.01/10$	–	–	100	500	0.6734	0.5708	0.8366	0.3319	$1.61 \cdot 10^{-6}$	$1.48 \cdot 10^{-6}$	$9.82 \cdot 10^{-7}$	$2.24 \cdot 10^{-5}$
Experiment 3.20	Double Integrator	Q-Learning	0.4/0.9	0.9	$\Omega_4/0.001/10$	–	–	200	500	0.7978	0.5205	0.9167	0.3836	$1.18 \cdot 10^{-6}$	$2.25 \cdot 10^{-7}$	$9.62 \cdot 10^{-7}$	$5.53 \cdot 10^{-6}$
Experiment 3.21	Double Integrator	Q-Learning	0.4/0.9	0.9	$\Omega_4/0.001/10$	–	–	500	500	0.4037	0.6048	0.6080	0.1491	$1.52 \cdot 10^{-6}$	$8.78 \cdot 10^{-7}$	$9.71 \cdot 10^{-7}$	$9.47 \cdot 10^{-6}$
Experiment 3.22	Double Integrator	Q-Learning	0.3/0.95	0.9	$\Omega_1/2/1$	50	50	500	500	0.9884	0.0000	1.0000	0.0000	$1.15 \cdot 10^{-6}$	$1.10 \cdot 10^{-7}$	$9.82 \cdot 10^{-7}$	$1.55 \cdot 10^{-6}$
Experiment 3.23	Double Integrator	Q-Learning	0.3/0.95	0.9	$\Omega_1/2/1$	100	100	500	500	0.9884	0.0000	1.0000	0.0000	$1.17 \cdot 10^{-6}$	$1.10 \cdot 10^{-7}$	$9.92 \cdot 10^{-7}$	$1.59 \cdot 10^{-6}$
Experiment 3.24	Double Integrator	Q-Learning	0.4/0.95	0.9	$\Omega_4/0.01/10$	100	100	500	500	0.5608	0.6048	0.7797	0.2780	$1.48 \cdot 10^{-6}$	$7.23 \cdot 10^{-7}$	$9.62 \cdot 10^{-7}$	$6.84 \cdot 10^{-6}$

Table D.3 – Results of the DQN policy evaluation, using different parameter combinations. std. and avg. denote the standard deviations and sample mean, respectively. The maximum metric for every system and minimum execution times are highlighted. Every evaluation run was executed in 1000 epochs with 2000 time steps. All times are given in seconds. Process and measurement noise of the environment are set to 10^{-2} and 10^{-6}, respectively.

Experiment	System	Policy	$avg.\ m_\Omega$	$avg.\ m_u$	$avg.\ m_b$	$avg.\ m$	avg. e	std. e	min. e	max. e
Experiment 4.1	Inverted Pendulum	π_{static}	0.9549 ± 0.003	0.0000 ± 0.0000	1.0000 ± 0.0000	0.0000 ± 0.0000	$7.68 \cdot 10^{-8}$	$1.86 \cdot 10^{-8}$	$2.50 \cdot 10^{-8}$	$2.70 \cdot 10^{-7}$
Experiment 4.2	Inverted Pendulum	π_{DRL}	0.7843 ± 0.0016	0.4041 ± 0.0022	1.0000 ± 0.0000	0.3171 ± 0.0020	$9.50 \cdot 10^{-7}$	$1.59 \cdot 10^{-7}$	$8.31 \cdot 10^{-7}$	$2.77 \cdot 10^{-6}$
Experiment 4.3	Inverted Pendulum	π_{PI}	0.3913 ± 0.0088	0.6025 ± 0.0011	0.9746 ± 0.0017	0.2312 ± 0.0037	$3.47 \cdot 10^{-7}$	$3.56 \cdot 10^{-8}$	$1.70 \cdot 10^{-7}$	$6.85 \cdot 10^{-7}$
Experiment 4.4	Inverted Pendulum	π_{SLP}	0.2192 ± 0.0015	0.7760 ± 0.0012	0.9954 ± 0.0006	0.1695 ± 0.0013	$1.85 \cdot 10^{-6}$	$1.74 \cdot 10^{-7}$	$6.95 \cdot 10^{-7}$	$4.03 \cdot 10^{-5}$
Experiment 4.5	Inverted Pendulum	π_{DG}	0.1733 ± 0.0014	0.7919 ± 0.0011	0.9996 ± 0.0001	0.1373 ± 0.0012	$5.09 \cdot 10^{-2}$	$1.05 \cdot 10^{-2}$	$1.87 \cdot 10^{-3}$	$8.11 \cdot 10^{-3}$
Experiment 4.6	Double Integrator	π_{static}	0.9985 ± 0.0000	0.0000 ± 0.0000	1.0000 ± 0.0000	0.0000 ± 0.0000	$7.15 \cdot 10^{-8}$	$1.72 \cdot 10^{-8}$	$2.50 \cdot 10^{-8}$	$1.98 \cdot 10^{-7}$
Experiment 4.7	Double Integrator	π_{DRL}	0.8992 ± 0.0028	0.6514 ± 0.0035	1.0000 ± 0.0000	0.5862 ± 0.0040	$8.93 \cdot 10^{-7}$	$1.03 \cdot 10^{-7}$	$7.94 \cdot 10^{-7}$	$1.53 \cdot 10^{-6}$
Experiment 4.8	Double Integrator	π_{PI}	0.7955 ± 0.0088	0.7993 ± 0.0002	0.9549 ± 0.0047	0.6130 ± 0.0091	$3.41 \cdot 10^{-7}$	$3.33 \cdot 10^{-8}$	$1.58 \cdot 10^{-7}$	$7.05 \cdot 10^{-7}$
Experiment 4.9	Double Integrator	π_{SLP}	0.6087 ± 0.0045	0.9034 ± 0.0021	0.9936 ± 0.0006	0.5478 ± 0.0052	$1.84 \cdot 10^{-6}$	$1.55 \cdot 10^{-7}$	$8.01 \cdot 10^{-7}$	$3.19 \cdot 10^{-6}$
Experiment 4.10	Double Integrator	π_{DG}	0.6093 ± 0.0044	0.8974 ± 0.0023	1.0000 ± 0.0000	0.5481 ± 0.0052	$5.90 \cdot 10^{-2}$	$5.15 \cdot 10^{-3}$	$3.37 \cdot 10^{-3}$	$7.45 \cdot 10^{-2}$

Table D.4 – Evaluation metric results of the DQN policy compared to the policies evaluated in [Rhe19, p. 74] with 200 time steps using 750 simulation runs. std. and avg. denote the standard deviations and sample mean respectively. The 95% confidence intervals of the sample means are symbolized by ±. All times are given in seconds. Process and measurement noise of the environment are set to 10^{-3} and 10^{-6}, respectively. The maximum metrics for every system and minimum execution times are highlighted.

LIST OF ACRONYMS

QoC Quality of Control

SL Supervised Learning

RTCS Real-Time Control System

LTVS Linear Time-Variant System

RTOS Real-Time Operating System

MDP Markov Decision Processes

RL Reinforcement Learning

DRL Deep Reinforcement Learning

MSE Mean Squared Error

MAE Mean Absolute Error

DQN Deep Q-Learning

FFN Feed-Forward Network

NN Neuronal Network

ANN Artificial Neuronal Network

s.p.d. symmetric and positive definite

sp.s.d. symmetric and positive semi-definite

Leaky ReLU Leaky Rectified Linear Unit

TDL Temporal Difference Learning

USL Unsupervised Learning

MLP Multi-Layer Perceptron

ERB Experience Replay Buffer

ER Experience Replay

TN Target Network

SARSA State–action–reward–state–action

LIST OF FIGURES

1.1 An overview of the given environment, including a simplified illustration of the control loop and the RTOS. The policy interacts with the RTOS based on the estimated state and QoC. 2

3.1 The environment consists of the control loop and the RTOS. In this figure, simplified versions of both are illustrated. The action a is influencing the system at a given time step k and a policy provides the best action. 8
3.2 A graphical representation of a perceptron with three inputs x_1, x_2, x_3, weights w_1, w_2, w_3 and bias b forming the linear regression Equation (3.14). The activation function σ is applied to the linear combination, resulting in the output y_j. 11
3.3 A fully connected MLP network with input, output and two hidden layers. Weights $w^l_{k,j}$ are illustrating the affine transformations between the connected neurons. These neurons are drawn as nodes containing σ if there is an activation function used. Nodes containing 1 are used to symbolize the bias to be added in every layer except the input. 12
3.4 Comparison of classification and regression problems. Dividing the data set into classes using classification can be seen in the left figure. Fitting a line through the data points in regression is provided in the right figure. 14
3.5 Interaction of the agent with the environment [SB18, p. 48]. 15

5.1 An overview of the action-value approximation using a NN. The environment is providing the states, while the NN estimates the best possible action a_k. The network is trained using gradient descent. This optimization process requires a loss function. To simplify the picture, the QoC is considered as part of the state $\hat{x}_k$. 22

6.1 An overview of the system, the DRL approach, and the optimizations. The ERB includes the Mini Batch, which samples random transitions from the Replay Memory. A TN saves a copy of the weights from the NN. Optimization like Weight Initialization and Loss Calculation are directly connected to the NN and can improve its learning speed. Adopted from [Gre+20]. 26
6.2 The cumulative reward of 500 time steps over 1000 epochs. The graph without the TN suffers from catastrophic forgetting, whereas the graph with the TN rises constantly and persist at a high cumulated reward.[15]. 28
6.3 The cumulative reward of 500 time steps over 1000 epochs. The graph without the ERB is reaching the highest cumulated reward at epoch 800. With the ERB the graph reaches the reward of 900 before epoch 400.[16] 29

6.4 Corresponding losses for the three functions MSE, MAE and Huber loss with $\delta = 1.0$ and $\delta = 0.1$. 30
6.5 Cumulated reward for 500 time steps in 100 epochs, where the He initialization reaches a higher cumulated reward[17]. The zero initialized NN shows almost no improvement whereas the Xavier network shows a small improvement. 31
6.6 Average execution time over 500 steps with standard deviation. Measured are the experiments 1.1, 1.6 and 1.11, as given in Table D.1. 35
6.7 Average count of the QoC not obeying its minimum. This figure shows the experiments 1.1, 1.6 and 1.11, as given in Table D.1 35
6.8 This figure shows the changing of the QoC over 100 time steps. The distance to the minimum QoC increases by incrementing the QoC reward parameter $q1$. 36

LIST OF TABLES

6.1 An overview over the possible parameter combinations and values divided into three categories. 32

A.1 Metrics and execution times for the experiment 2.3 with deterministic and non-deterministic execution times. 45

B.1 Temporal behavior of the control tasks [Rhe19, p. 70]. 50

C.1 Parameter set for the evaluation of the target network. The noise covariance matrices amount are set to $H = 10^{-3}$ and $N_p = 10^{-6}$ respectively. 51

C.2 Parameter set for the evaluation of the optimization methods. The noise covariance matrices amount are set to $H = 10^{-3}$ and $N_p = 0$ respectively. 52

C.3 Parameter set for the main evaluation, with cells marked in colour represent the values to be evaluated. 52

D.1 Results of the DQN policy evaluation, using different parameter combinations. std. and avg. denote the standard deviations and sample mean, respectively. The maximum metric for every system and minimum execution times are highlighted. Every evaluation run was executed in 1000 epochs with 2000 time steps. All times are given in seconds. Process and measurement noise of the environment are set to 10^{-3} and 0, respectively. 54

D.2 Results of the DQN policy evaluation, using different parameter combinations. std. and avg. denote the standard deviations and sample mean, respectively. The maximum metric for every system and minimum execution times are highlighted. Every evaluation run was executed in 1000 epochs with 2000 time steps. All times are given in seconds. Process and measurement noise of the environment are set to 10^{-3} and 10^{-6}, respectively. 55

D.3 Results of the DQN policy evaluation, using different parameter combinations. std. and avg. denote the standard deviations and sample mean, respectively. The maximum metric for every system and minimum execution times are highlighted. Every evaluation run was executed in 1000 epochs with 2000 time steps. All times are given in seconds. Process and measurement noise of the environment are set to 10^{-2} and 10^{-6}, respectively. 56

D.4 Evaluation metric results of the DQN policy compared to the policies evaluated in [Rhe19, p. 74] with 200 time steps using 750 simulation runs. std. and avg. denote the standard deviations and sample mean respectively. The 95% confidence intervals of the sample means are symbolized by ±. All times are given in seconds. Process and measurement noise of the environment are set to 10^{-3} and 10^{-6}, respectively. The maximum metrics for every system and minimum execution times are highlighted. 57

LIST OF ALGORITHMS

5.1 The SARSA algorithm calculating the Q-values in each time step k, including Equation (5.1) and Equation (5.2) [SB18, p. 130]. 21
6.1 The DQN algorithm with ERB buffer and TN. Values with an apostrophe mark the values gained from the TN, which is updated every C steps. In line 11 the transition is stored into the replay buffer and randomly sampled from it and trained afterwards [Mni+13, p. 5]. 27

REFERENCES

[ABB12] Sander Adam, Lucian Busoniu, and Robert Babuska. "Experience Replay for Real-Time Reinforcement Learning Control." In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.2 (2012), pp. 201–212. DOI: 10.1109/TSMCC.2011.2106494.

[Alc+20] Miguel Alcon et al. "Timing of Autonomous Driving Software: Problem Analysis and Prospects for Future Solutions." In: *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2020, pp. 267–280. DOI: 10.1109/RTAS48715.2020.000-1.

[AS64] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. New York: Dover, 1964.

[ÅW90] Karl J. Åström and Björn Wittenmark. *Computer-Controlled Systems: Theory and Design (2nd Ed.)* USA: Prentice-Hall, Inc., 1990. ISBN: 0131686003.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Ed. by M Jordan, J Kleinberg, and B Schölkopf. Vol. 4. Information science and statistics 4. Springer, 2006. Chap. Graphical, p. 738. ISBN: 9780387310732. DOI: 10.1117/1.2819119. eprint: 0-387-31073-8. URL: http://www.library.wisc.edu/selectedtocs/bg0137.pdf.

[Bos07] Adriaan van den Bos. *Parameter Estimation for Scientists and Engineers*. USA: Wiley-Interscience, 2007. ISBN: 0470147814.

[Cap+20] Maurizio Capra et al. "Hardware and Software Optimizations for Accelerating Deep Neural Networks: Survey of Current Trends, Challenges, and the Road Ahead." In: *IEEE Access* 8 (2020), pp. 225134–225180. DOI: 10.1109/ACCESS.2020.3039858.

[Dec86] Rina Dechter. "Learning While Searching in Constraint-Satisfaction-Problems." In: Jan. 1986, pp. 178–185.

[Dek+05a] F.M. Dekking et al. *A Modern Introduction to Probability and Statistics: Understanding Why and How*. Springer Texts in Statistics. Springer, 2005. ISBN: 9781852338961. URL: https://books.google.de/books?id=XLUMIlombgQC.

[Dek+05b] Frederik Dekking et al. *A modern introduction to probability and statistics. Understanding why and how*. Jan. 2005. ISBN: 978-1-85233-896-1. DOI: 10.1007/1-84628-168-7.

[Don+21] Xingping Dong et al. "Dynamical Hyperparameter Optimization via Deep Reinforcement Learning in Tracking." In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43.5 (2021), pp. 1515–1529. DOI: 10.1109/TPAMI.2019.2956703.

[Esc21] Jonas Eschmann. "Reward Function Design in Reinforcement Learning." In: *Reinforcement Learning Algorithms: Analysis and Applications*. Ed. by Boris Belousov et al. Springer International Publishing, 2021, pp. 25–33. ISBN: 978-3-030-41188-6. DOI: 10.1007/978-3-030-41188-6_3. URL: https://doi.org/10.1007/978-3-030-41188-6_3.

[FH19] Matthias Feurer and Frank Hutter. "Hyperparameter Optimization." In: *Automated Machine Learning: Methods, Systems, Challenges*. Ed. by Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. Cham: Springer International Publishing, 2019, pp. 3–33. ISBN: 978-3-030-05318-5. DOI: 10.1007/978-3-030-05318-5_1. URL: https://doi.org/10.1007/978-3-030-05318-5_1.

[FL+18] Vincent François-Lavet et al. "An Introduction to Deep Reinforcement Learning." In: *Found. Trends Mach. Learn.* 11.3–4 (Dec. 2018), 219–354. ISSN: 1935-8237. DOI: 10.1561/2200000071. URL: https://doi.org/10.1561/2200000071.

[Fre99] Robert M. French. "Catastrophic forgetting in connectionist networks." In: *Trends in Cognitive Sciences* 3.4 (1999), pp. 128–135. ISSN: 1364-6613. DOI: https://doi.org/10.1016/S1364-6613(99)01294-2. URL: https://www.sciencedirect.com/science/article/pii/S1364661399012942.

[Gau+18] Maximilian Gaukler et al. "A New Perspective on Quality Evaluation for Control Systems with Stochastic Timing." In: *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (HSCC '18)* (Porto, Portugal). New York, NY, USA: ACM Press, Apr. 11, 2018–Apr. 13, 2018, pp. 91–100. ISBN: 978-1-4503-5642-8/18/04. DOI: 10.1145/3178126.3178134. URL: https://dl.acm.org/authorize?N667444.

[GB10] Xavier Glorot and Yoshua Bengio. *Understanding the difficulty of training deep feedforward neural networks*. Ed. by Yee Whye Teh and Mike Titterington. Chia Laguna Resort, Sardinia, Italy, 2010. URL: http://proceedings.mlr.press/v9/glorot10a.html.

[GBB11] Xavier Glorot, Antoine Bordes, and Y. Bengio. "Deep Sparse Rectifier Neural Networks." In: *Proceedings of the 14th International Conference on Artificial Intelligence and Statisitics (AISTATS) 2011* 15 (Jan. 2011), pp. 315–323.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[Gre+20] Martin Gregurić et al. "Application of Deep Reinforcement Learning in Traffic Signal Control: An Overview and Impact of Open Traffic Data." In: *Applied Sciences* 10.11 (2020). ISSN: 2076-3417. DOI: 10.3390/app10114011. URL: https://www.mdpi.com/2076-3417/10/11/4011.

[HB20] Abdul Hafiz and Ghulam Bhat. *Image Classification by Reinforcement Learning with Two-State Q-Learning*. June 2020.

[He+15] Kaiming He et al. *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*. 2015. arXiv: 1502.01852.

[HJ12] R.A. Horn and C.R. Johnson. *Matrix Analysis*. Cambridge University Press, 2012. ISBN: 9781139788885. URL: https://books.google.de/books?id=07sgAwAAQBAJ.

[Hub92] Peter J. Huber. "Robust Estimation of a Location Parameter." In: *Breakthroughs in Statistics: Methodology and Distribution*. Ed. by Samuel Kotz and Norman L. Johnson. New York, NY: Springer New York, 1992, pp. 492–518. ISBN: 978-1-4612-4380-9. DOI: 10.1007/978-1-4612-4380-9_35. URL: https://doi.org/10.1007/978-1-4612-4380-9_35.

[Lin92] Long-Ji Lin. "Reinforcement Learning for Robots Using Neural Networks." UMI Order No. GAX93-22750. PhD thesis. USA, 1992.

[Loz+13] Camilo Lozoya et al. "Resource and performance trade-offs in real-time embedded control systems." In: *Real-Time Systems* 49.3 (2013), pp. 267–307. ISSN: 1573-1383. DOI: 10.1007/s11241-012-9174-9. URL: https://doi.org/10.1007/s11241-012-9174-9.

[MHN13] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. *Rectifier nonlinearities improve neural network acoustic models*. 2013.

[ML17] Wei Ma and Jun Lu. "An equivalence of fully connected layer and convolutional layer." In: *arXiv preprint arXiv:1712.01252* (2017).

[Mni+13] Volodymyr Mnih et al. "Playing Atari With Deep Reinforcement Learning." In: *NIPS Deep Learning Workshop*. 2013.

[Mni+15] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning." In: *Nature* 518 (Feb. 2015), pp. 529–33. DOI: 10.1038/nature14236.

[Nie18] Michael A. Nielsen. *Neural Networks and Deep Learning*. misc. 2018. URL: http://neuralnetworksanddeeplearning.com/.

[PS18] Aleksis Pirinen and Cristian Sminchisescu. "Deep Reinforcement Learning of Region Proposal Networks for Object Detection." In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 6945–6954. DOI: 10.1109/CVPR.2018.00726.

[Qin+12] Bin Qin et al. "Robust Reinforcement Learning Control and Its Application Based on IQC and PSO." In: *Second International Conference on Intelligent System Design and Engineering Application*. 2012, pp. 505–508. DOI: 10.1109/ISdea.2012.658.

[Qu+16] B.Y. Qu et al. "Two-hidden-layer extreme learning machine for regression and classification." In: *Neurocomputing* 175 (2016), pp. 826–834. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2015.11.009. URL: https://www.sciencedirect.com/science/article/pii/S092523121501663X.

[Rhe19] Tim Rheinfels. "Leveraging Machine-Learning Techniques for Quality-Aware Real-Time Scheduling." MA thesis. Germany: Friedrich-Alexander-Universität Erlangen-Nürnberg, Dept. of Computer Science, July 2019.

[RM98] Russell D. Reed and Robert J. Marks. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262181908.

[Ros58] Frank Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65 6 (1958), pp. 386–408.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[Sin+00] Satinder Singh et al. “Convergence Results for Single-Step On-PolicyReinforcement-Learning Algorithms.” In: *Mach. Learn.* 38.3 (Mar. 2000), 287–308. ISSN: 0885-6125. DOI: 10.1023/A:1007678930559.

[TB97] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 1997. ISBN: 0898713617.

[WEH15] Ning Wang, Meng Joo Er, and Min Han. “Generalized Single-Hidden Layer Feedforward Networks for Regression Problems.” In: *IEEE Transactions on Neural Networks and Learning Systems* 26.6 (2015), pp. 1161–1176. DOI: 10.1109/TNNLS.2014.2334366.

[Wer74] Paul Werbos. “Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Science. Thesis (Ph. D.). Appl. Math. Harvard University.” PhD thesis. Jan. 1974.

[Ye+20] Yujian Ye et al. “Model-Free Real-Time Autonomous Control for a Residential Multi-Energy System Using Deep Reinforcement Learning.” In: *IEEE Transactions on Smart Grid* 11.4 (2020), pp. 3068–3082. DOI: 10.1109/TSG.2020.2976771.

[Yin19] Xue Ying. “An Overview of Overfitting and its Solutions.” In: *Journal of Physics: Conference Series* 1168 (Feb. 2019), p. 022022. DOI: 10.1088/1742-6596/1168/2/022022.

[YLW14] Xiong Yang, Derong Liu, and Ding Wang. “Reinforcement learning for adaptive optimal control of unknown continuous-time nonlinear systems with input constraints.” In: *International Journal of Control* 87.3 (2014), pp. 553–566. DOI: 10.1080/00207179.2013.848292.

[ZS17] Shangtong Zhang and Richard S. Sutton. “A Deeper Look at Experience Replay.” In: *CoRR* abs/1712.01275 (2017). arXiv: 1712.01275. URL: http://arxiv.org/abs/1712.01275.