PADERBORN UNIVERSITY
The University for the Information Society

REGELUNGS- UND
AUTOMATISIERUNGS-
TECHNIK

Combining Model-Based and Model-Free Optimal Control for Dynamic Systems

Pascal Peters

Faculty of Computer Science, Electrical Engineering and Mathematics
Department of Automatic Control

Advisor
Dr.-Ing. Oliver Wallscheid
Maximilian Schenke

Paderborn University
April 2022

Combining Model-Based and Model-Free Optimal Control for Dynamic Systems

30ects thesis submitted in partial fulfillment of a Magisterii Scientiae degree in Electrical Engineering

Faculty of Computer Science, Electrical Engineering and Mathematics
Department of Automatic Control
Paderborn University
Warburger Str. 100
33098 Paderborn, Paderborn

Bibliographic information:
Pascal Peters, 2022, Combining Model-Based and Model-Free Optimal Control for Dynamic Systems, M.Sc. degree, Faculty of Computer Science, Electrical Engineering and Mathematics, Paderborn University.

Abstract

This work presents a safe reinforcement learning controller that guarantees to respect any environment constraints without prior model knowledge. By solving a constrained optimization problem in the form of a quadratic program, the unsafe control actions are modified and adapted to respect the environment's constraints at any time. Using the feasible-set of the quadratic program as constraints enables the controller to keep the agent inside the control-invariant set to guarantee feasibility for all future steps. Utilizing the deep deterministic policy gradient algorithm enables the agent to work with continuous state and action spaces to solve complex control tasks. To eliminate dependencies on prior available environment knowledge, a model is determined during the learning process using the recursive least-squares algorithm.

Contents

Abbreviations

AC	Actor-Critic.
ANN	artifical neural network.
CCV	cumulated constraint violations.
CMSE	cumulated MSE.
CPO	constrained policy optimization.
DDPG	deep deterministic policy gradient.
DPG	deterministic policy gradient.
DQN	deep Q networks.
GP	gaussian process.
KKT	Karush-Kuhn-Tucker.
LQR	linear quadratic regulator.
LS	least squares.
LTI	linear time invariant.
MASC	model adaptive safeguard controller.
MC	Monte-Carlo.
MDP	Markov decision process.
MIP	mixed-integer program.
MPC	model predictive control.
MSBE	mean-squared Bellman error.
MSE	mean-squared-error.
NN	neural network.
OLS	ordinary least squares.

PEM	Prediction Error Method.
PG	policy gradient.
PPO	proximal policy optimization.
QP	quadratic program.
RKHS	reproducing kernel Hilbert space.
RL	reinforcement learning.
RLS	recursive least squares.
SAC	soft actor-critic.
SWSE	scaled WSE.
TD	temporal difference.
TD3	twin delayed deep deterministic policy gradient.
WSE	weighted sum of errors.

Symbols

$\boldsymbol{y}$	Realization of random vector
$\boldsymbol{Y}$	Random vector
$\boldsymbol{\mathcal{Y}}$	Overall set of random vectors
r	Realization of scalar random variable
R	Scalar random variable
$\mathcal{R}$	Overall set of random variables
$\overline{\boldsymbol{y}}$	Sample mean of observation
$\widetilde{\boldsymbol{y}}$	Normalized observation
$\dot{\boldsymbol{x}}$	First derivative w.r.t to the time t
$\ddot{\boldsymbol{x}}$	Second derivative w.r.t to the time t
$\boldsymbol{\pi}$	Stochastic control policy
$\boldsymbol{\eta}$	Deterministic control policy
$\hat{\boldsymbol{\theta}}$	Parameter estimate
γ	Reinforcement learning discount factor
β	Temporal difference forgetting factor
α	Gradient ascent learning rate
$\boldsymbol{\theta}$	Actor parameters
$\boldsymbol{\phi}$	Critic parameters
τ	Soft target network update coefficient
λ	Recursive least squares forgetting factor
n	Negative slope of the LeakyReLU activation function
n	State dimension

Abbreviations

m	Action dimension
k	Discrete time index
t	Continuous time index
N	Anticipated/Predicted horizon of state trajectory
K	Episode length
$\mathbb{E}$	Expectation operator
∇_x	Gradient w.r.t to x
ln	Natural logarithm
$\lvert \cdot \rvert$	Cardinality of a set or absolute value of a scalar
$(\cdot)^{\mathrm{T}}$	Transposed vector or matrix
$\exists$	Existential quantification
$\cup$	Union of sets
$\cap$	Intersection of sets
$\lvert\lvert \cdot \rvert\rvert^2$	Euclidean norm
$\lvert\lvert \cdot \rvert\rvert_F$	Frobeniusnorm

1. Introduction

The field of reinforcement learning gained a lot of popularity in recent years. Publications like AlphaGo [Sil+16] and AlphaFold [Jum+21] have shown that intelligent algorithms from the area of reinforcement learning have the ability to solve complex tasks, whether it is learning how to win a game of Go against a human or to learn how to predict complicated protein structures. The accompanying news coverage has attracted a lot of attention in the research community and, at the same time provided the general public relatively easy access to a contemporary research topic in machine learning. This growing interest in possible applications and progress made in research opened up several new areas to study in reinforcement learning (RL). Especially algorithms operating on large, continuous data input, have enabled solving complex optimal control tasks, ranging from electrical drive control [SW21] and movement control of robotic systems [Mor+21] to decision making in autonomous driving [Lub+20].

Most tasks are solved by learning an optimal control strategy purely by interaction with the environment and collecting experience in a trial and error fashion. This makes RL a highly adaptive tool in the field of optimal control, which does not need specific knowledge about the physical system. Finding an optimal control policy for an infinite horizon by solving the Bellman equation makes RL a powerful tool in optimal control. In addition, RL controllers can profit from low computational costs during actual control operation by only executing simple arithmetic operations to determine the optimal control output, without the need to solve complex optimization problems.

However, especially during the learning process the learner needs to explore different control actions which are not yet optimal and can lead to unwanted and unsafe behavior of the controlled environment. In real world applications this would cause mechanical damage of machinery or endanger the environment. This is a critical issue of RL, especially when applied in areas where humans interact with the controlled plant, e.g. robotics or autonomous driving.

Sophisticated control methods, like model predictive control (MPC), use prior

model knowledge to predict state and actions for a finite horizon and guarantees safe control behavior. The determined model has to match the real system and is used to predict state trajectories to evaluate and optimize an objective to find the optimal control action. This is commonly done by formulating a constrained optimization problem. A mathematical solver tries solving this problem by finding an optimal solution and obeying the defined constraints.

As such a procedure can be computationally expensive and unpractical for a highly dynamic system, this work proposes an approach combining the adaptability and low computational costs of RL controllers with the guaranteed safe behavior of model based optimal control.

By including a so called safeguard in a standard RL controller, safe behavior during all operations is guaranteed. If the RL controller selects unsafe actions, the safeguard steps in and modifies the proposed action such that the system's constraints are respected. By using a prior known model or by identifying the model during the control process, it is possible to formulate a simple optimization problem for finding safe actions.

The next chapter gives a brief overview of related work on the problem. Chapter three introduces the rudiments of reinforcement learning, optimal control and system identification, to then later derive the developed controller. Followed by an evaluation and discussion of the results found in chapter five. The thesis concludes with a summary of the gained insights made and with suggestions for future work to be done on this topic.

2. Related Work

Relevant work for this theses done in the field of deep RL regarding complex control tasks and safety is introduced in the following chapter. The publications regarding safety in RL are categorized after certain aspects in Table 2.1.

Several RL publications handling continuous state and action-spaces in order to solve complex control tasks have presented promising results. Proposed algorithms like the deep deterministic policy gradient (DDPG) [Lil+15] or the twin delayed deep deterministic policy gradient (TD3) [FHM18] are based on the idea of Q-learning and policy gradients (PGs) and follow an off-policy approach with deterministic policies and neural network (NN) function approximators to solve control problems. In contrast, the soft actor-critic (SAC) [Haa+18] method (off-policy) or the proximal policy optimization (PPO) [Sch+17] algorithm (on-policy) use stochastic policies as well as policy gradient optimization.

Following the progress in optimal control, the task of increasing safety of RL controllers gained a lot of importance. The constrained policy optimization (CPO) [Ach+17] treats the safety problem by designing a specific cost function and bounding expected returns during the learning process to force the stochastic policy and model-free RL agent to converge to safe behavior.

Alternative approaches combine techniques from classic control theory and RL. Publications such as [Ber+17] focus on safe behavior in the sense of stability guarantees. By extending the explored state-space under stability conditions using Lyapunov functions, [Ber+17] excludes unsafe states from the exploration space to ensure that the agent only moves in a safe sub-region of the state-space. The exclusion of unsafe states is adapted slowly during learning to find a not too conservative solution which ignores possible useful states in the learning process. [Kol+19] and [GF14] also focus on safe exploration. [Kol+19] derives confidence intervals for predicted trajectories on the basis of an estimated, statistical model. Through this, constraint satisfaction is ensured in an MPC fashion. The safe controller is trained by using a gaussian process (GP) and reproducing kernel Hilbert space (RKHS). A more intuitive approach is followed by [GF14]. Here, the

exploration of the RL agent is emphasized to be safe by incorporating different risk-measures and criteria in the learning process. Extended is this work by [GF15] where similar publications are collected and summarized regarding the topic of reward modification based on risk and curiosity criteria.

The approaches from [Dal+18], [WZ18] or [GZB20] modify the RL controllers output, such that system constraints are respected by solving a constrained optimization problem. In order to achieve this, prior model-knowledge is utilized to define the system constraints and optimization problem. Model dynamics can be presupposed or identified during operation of the controller. The approach of [KL20] follows a slightly different motivation. The main focus lies on stability analysis of neural network controllers with no specifications on how to learn an optimal control strategy. By following approaches of classic MPC methods, safe, constraint respecting behavior and stability is ensured by analyzing the output range of neural network controllers and finding the corresponding input states that lead to a safe controller output. This is done by solving different mixed-integer programs (MIPs).

Focusing more on MPC, [Kar+20] proposes a practical approach for learning to plan a safe trajectory for unmanned ground vehicles. By employing an Actor-Critic (AC) approach where the role of the actor is taken by an MPC controller, safety is ensured throughout the learning process. The value-function is estimated by the critic and used as objective of the MPC. [Kar+20] is able to employ an algorithm that directly finds optimal behavior on the real-world environment by learning the cost function without any human intervention to determine the optimal control action. [Lub+20] does also combine classic control with modern methods. An RL controller plans an optimal control trajectory and a fully setup MPC controller monitors the agent's actions and takes over the control if the trajectory is deemed unsafe. Therefore, RL and MPC are working together to solve one control task.

The use of models in RL is typically combined with the popular Dyna-Architecture [Sut91] approach. Here, [Gu+16], [Mor+21] and [LW21] adopt methods to increase data efficiency during learning and identifying the dynamic model behind the environment. Estimated models are used to create additional data to help the controller during the learning process and decrease the need for observations made from the real physical system or are employed to predict system trajectories as a look-ahead in order to increase control performance. Safe behavior for approaches like [Mor+21] is only emphasized indirectly by minimizing the need to interact with the real world system and using model-generated data for the controller optimization. Work done by [MLG20] proposes a model based approach where the

dynamic model is learned online with the Prediction Error Method (PEM) which is then used for MPC. The parameters of the MPC are then learned in an RL fashion.

Table 2.1: Categorized overview of the presented publications. Listed are the used state-action space, the employed RL algorithm, safety behavior, model assumptions and exploration technique. S/A indicates a continuous or large state and action space, s/a indicates a finite/ discrete space. N/A is used if no specifications in mentioned in the publication.

Source	Spaces	RL	Safety	Model	Exploration
[Ach+17]	S/A	CPO	✓	model free	stochastic policy
[Ber+17]	s/a	policy optimization	✓	GP model	excluding unsafe states of the exploration set
[Dal+18]	S/A	DDPG	✓	linearized model learned from observation	random exploration noise
[WZ18]	S/a	policy optimization	✓	stochastic model	N/A
[Mor+21]	S/A	SAC	(✓)	NN model learned from observation	following RL policy
[LW21]	S/A	AC	✗	NN model learned from observation	N/A
[GF15]	S/A, s/a	various	✓	various	various
[Gu+16]	S/A	Q-learning	✗	model learned from trajectories	random exploration noise
[Kar+20]	S/A	AC	✓	model learned from observations	operating MPC interacts with environment
[KL20]	S/A	N/A	✓	N/A	excluding unsafe actions by finding safe output region
[Kol+19]	S/A	GP, RKHS	✓	stochastic model	prediction of safe trajectories
[GZB20]	S/A	Q-learning, PG	✓	stochastic model	robust MPC
[Lub+20]	S/A	DDPG	✓	known model	exploration through RL
[GF14]	S/A	Monte-Carlo Methods	✓	model free	gaussian exploration noise
[MLG20]	S/A	Semi-Gradient Q-learning	✓	PEM system identification	exploration through RL

3. Rudiments

To lay the theoretical groundwork, the following chapter introduces and explains the necessary rudiments. The first section presents the basics of RL. This is followed by a brief description of policy gradients and Actor-Critic (AC) approaches. Another section outlines the foundations of MPC and constrained optimization before the basic principles of system identification are delineated.

3.1. Reinforcement Learning

In optimal control, the main task is to find an optimal strategy to interact with a system to achieve a predefined goal. RL tries to handle this problem purely by learning the strategy through interaction with the system. In the context of RL, the physical system or plant are commonly referred to as the environment. The algorithm or controller interacting with the environment is called agent.

The process of learning an optimal control strategy is characterized by the use of exploration and exploitation. It has to be decided if it is more important to find new information about the system (exploration) or to follow and improve the already learned strategy (exploitation). This problem has no clear solution and the trade-off between exploration and exploitation is highly task dependent and has to be solved individually.

Building on the introduced terminology, the following sections extend the concepts of RL with the mathematical background and algorithms used in this work.

3.1.1. Environment Modeling

The agent-environment interaction is formally described by a Markov decision process (MDP) defined by the tuple $(\boldsymbol{\mathcal{Y}}, \boldsymbol{\mathcal{U}}, \mathcal{P}, \mathcal{R}, \gamma)$. $\boldsymbol{\mathcal{Y}}$ describes the state and $\boldsymbol{\mathcal{U}}$ the action-space the agent operates in. The transition probabilities that describe the state-transition within the environment are given by $\mathcal{P}$ and only directly depend on the state previously visited and the last action taken, hence following the Markov property of stochastic processes.

In order to enable the agent to learn an optimal control strategy, the agent receives feedback on executed actions. This feedback is called reward, and is specified by $\mathcal{R}$. The discount factor γ is used to calculate the discounted sum of the rewards:

$$g_k = r_{k+1} + \gamma r_{k+2} + \gamma^2 r_{k+3} + \cdots + \gamma^N r_{k+1+N} = \sum_{i=0}^{N} \gamma^i r_{k+i+1} \tag{3.1}$$

that is also known as the return g at the current step k. As described by [Sut18], the so called episodic tasks take a finite number of steps N until the episode terminates and restarts. For continuing tasks the terminal state is not specified and takes $N \to \infty$ steps. By maximizing the return during the learning process, the agent finds an optimal control strategy. The interaction between agent and environment is commonly represented by Figure 3.1. Here $\boldsymbol{y}_{k+1} \in \boldsymbol{\mathcal{Y}}$ describes the observation made after executing the action $\boldsymbol{u}_k \in \boldsymbol{\mathcal{U}}$ at step k. By doing this, the reward $r_{k+1} \in \mathcal{R}$ is received as feedback to the action $\boldsymbol{u}_k$. Because the system is fully observable and the state $\boldsymbol{x}_{k+1}$ is measured without any noise, the rest of this work uses $\boldsymbol{y} = \boldsymbol{x}$ for all derivations.

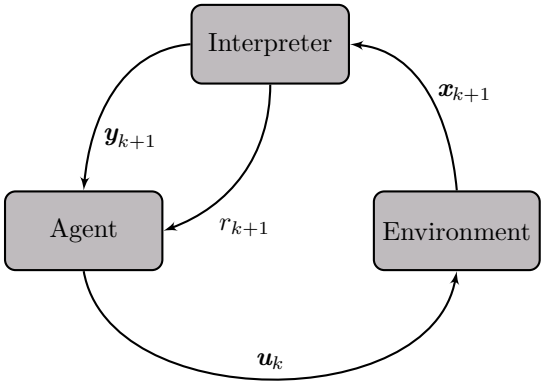

Figure 3.1: Agent-environment interaction loop in reinforcement learning applications

3.1.2. Basic Concepts

To develop algorithms and formal descriptions on how to optimize the agent's behavior, determining how well the actions selected by the agent perform on the environment is critical. A standard approach for that in RL is to estimate value functions. Namely, the state-value or action-value. Following [Sut18], the value-functions are defined as:

$$\begin{aligned} v_{\boldsymbol{\pi}}(\boldsymbol{x}_k) &= \mathbb{E}_{\boldsymbol{\pi}}\left[G_k \middle|\, \boldsymbol{X}_k = \boldsymbol{x}_k\right] = \mathbb{E}_{\boldsymbol{\pi}}\left[\sum_{i=0}^{\infty} \gamma^i R_{k+i+1} \middle|\, \boldsymbol{X}_k = \boldsymbol{x}_k\right] \\ q_{\boldsymbol{\pi}}(\boldsymbol{x}_k, \boldsymbol{u}_k) &= \mathbb{E}_{\boldsymbol{\pi}}\left[G_k \middle|\, \boldsymbol{X}_k = \boldsymbol{x}_k, \boldsymbol{U}_k = \boldsymbol{u}_k\right] \\ &= \mathbb{E}_{\boldsymbol{\pi}}\left[\sum_{i=0}^{\infty} \gamma^i R_{k+i+1} \middle|\, \boldsymbol{X}_k = \boldsymbol{x}_k, \boldsymbol{U}_k = \boldsymbol{u}_k\right] \end{aligned} \tag{3.2}$$

with the capital letters representing random variables due to the stochastic nature of the environment and the lower case letters their realization. The state-value function $v_{\boldsymbol{\pi}}(\boldsymbol{x}_k)$ is described as the expected value $\mathbb{E}_{\boldsymbol{\pi}}$ of return G_k given the current state $\boldsymbol{x}_k$, following the policy $\boldsymbol{\pi}$. The action-value $q_{\boldsymbol{\pi}}(\boldsymbol{x}_k, \boldsymbol{u}_k)$ is estimated as the expected value $\mathbb{E}_{\boldsymbol{\pi}}$ of the return G_k given the current state $\boldsymbol{x}_k$, following the current policy $\boldsymbol{\pi}$ and taking the action $\boldsymbol{u}_k$.

By taking advantage of the recursive properties of the value function, [Sut18] rewrites (3.2) in the form of the so called Bellman equation or Bellman expectation equation:

$$\begin{aligned} v_{\boldsymbol{\pi}}(\boldsymbol{x}_k) &= \mathbb{E}_{\boldsymbol{\pi}}\left[R_{k+1} + \gamma v_{\boldsymbol{\pi}}(\boldsymbol{X}_{k+1}) \middle| \boldsymbol{X}_k = \boldsymbol{x}_k\right] \\ &= \sum_{\boldsymbol{u}_k \in \boldsymbol{\mathcal{U}}} \boldsymbol{\pi}(\boldsymbol{u}_k \,|\boldsymbol{x}_k) q_{\boldsymbol{\pi}}(\boldsymbol{x}_k, \boldsymbol{u}_k) \\ q_{\boldsymbol{\pi}}(\boldsymbol{x}_k, \boldsymbol{u}_k) &= \mathbb{E}_{\boldsymbol{\pi}}\left[R_{k+1} + \gamma q_{\boldsymbol{\pi}}(\boldsymbol{X}_{k+1}, \boldsymbol{U}_{k+1})) \middle| \boldsymbol{X}_k = \boldsymbol{x}_k, \boldsymbol{U}_k = \boldsymbol{u}_k\right] \\ &= \mathcal{R}_{\boldsymbol{x}_k}^{\boldsymbol{u}_k} + \gamma \sum_{\boldsymbol{x}_{k+1} \in \boldsymbol{\mathcal{X}}} p_{\boldsymbol{x}_k, \boldsymbol{x}_{k+1}}^{\boldsymbol{u}_k} v_{\boldsymbol{\pi}}(\boldsymbol{x}_{k+1}) \end{aligned} \tag{3.3}$$

with the transition probability $p_{\boldsymbol{x}_k, \boldsymbol{x}_{k+1}}^{u}$ between state $\boldsymbol{x}_k$ and $\boldsymbol{x}_{k+1}$. Maximizing q leads to the optimal policy. Following the definition of Bellman's principle of optimality [Bel10], the optimal policy finds the maximal value for every state-action pair and is equal or better than previous policies found during the optimization process.

Solving the Bellman equation is not straightforward for large state and action spaces. The value functions have to be evaluated for every state-action pair. The computational complexity would become infeasible high. Furthermore, a unknown MDPs behind the environment make it impossible to calculate the expected values over the transition probabilities, which also limits the application of solving the Bellman equations analytically. These problems can be solved by using various RL techniques. Sample based, also called model-free, algorithms like Monte-Carlo (MC) methods or temporal difference (TD)-learning, do not need any knowledge of MDPs and reduce the computational effort, by determining estimates of the optimal solution of the Bellman equation, purely from observed data.

MC Methods are employed calculating estimates from returns sampled over a whole episode. Therefore, the value estimate is only available after observing a complete episode. This can be very unpractical for the application presented in this work and hence the idea of TD-learning is much more useful for control task that benefit from estimates found during an ongoing episode. The estimates are immediately available. Following [Sut18], one approach for TD control is the so called Q-learning, where the action-value is estimated as:

$$\hat{q}(\boldsymbol{x}_k, \boldsymbol{u}_k) \leftarrow \hat{q}(\boldsymbol{x}_k, \boldsymbol{u}_k) + \beta \left[r_{k+1} + \gamma \max_{\boldsymbol{u}} \hat{q}(\boldsymbol{x}_{k+1}, \boldsymbol{u}) - \hat{q}(\boldsymbol{x}_k, \boldsymbol{u}_k) \right] \tag{3.4}$$

with the forgetting factor β. Here the state-action value q is determined by not following a given policy, but taking the action which leads to the maximal action-value in the next step $k+1$. This makes Q-learning an off-policy algorithm where the agent does not follow the policy which is optimized during the learning process. The significant difference to MC methods is that the estimates of $\hat{q}(\boldsymbol{x}_k, \boldsymbol{u}_k)$ are bootstrapped, meaning the new estimates are calculated on the bases of other value-estimates. This enables Q-learning to find action-values at runtime when operating the environment.

For the so called tabular methods with finite state-spaces and action-spaces the TD approach is an important technique that can achieve good results and acts as a building block for more elaborated algorithms like SARSA, Q-learning or N-step TD algorithms. The computational complexity is still not practical for large state-action spaces.

3.1.3. Function Approximators and Policy Gradients

To find optimal solutions for the Bellman equation for large state and action spaces, so called function approximators are often a good method to choose. Function approximators aim to find a parameterized mapping from state-action inputs to an estimated value function output, instead of directly mapping actions to the resulting states or estimated value-functions. This lowers the computational costs significantly and makes RL applicable in large discrete or continuous state-action spaces. Common approximators for this are artifical neural networks (ANNs) as a nonlinear mapping from input to output. Here, the NN can either directly represent the policy $\boldsymbol{\pi}(\boldsymbol{u} \,|\, \boldsymbol{x}, \boldsymbol{\theta})$, by mapping a state input directly to an action,

calculating the parameters of a stochastic distribution which represents the policy or estimating the action-values q [Sut18].

In order to optimize the policy, algorithms like the policy gradient used with stochastic policies [Sut18] or deterministic policies like the deterministic policy gradient (DPG) [Sil+14], are used frequently. To derive a formal description on how to optimize the policy, the objective of the optimization problem $J(\boldsymbol{\theta})$ is given by [Sut18] as the value function:

$$J(\boldsymbol{\theta}) = v_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{\pi}}\left[v | \boldsymbol{X} = \boldsymbol{x}, \boldsymbol{\theta}\right], \tag{3.5}$$

where (3.5) is the objective for episodic tasks and $v_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}$ is the state-value with the underlying policy $\boldsymbol{\pi}_{\boldsymbol{\theta}}$ parameterized with $\boldsymbol{\theta}$. Using common gradient-based optimization techniques, the popular gradient ascent formulation with the learning-rate α is given as:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{3.6}$$

is applied to the objective of the policy gradient. The derivation of the objective is found by following [Sut18]. Taking the derivative from the objective equals to:

$$\nabla J(\boldsymbol{\theta}) = \nabla v_{\boldsymbol{\pi}_{\boldsymbol{\theta}}}(\boldsymbol{x}) = \nabla \left[\sum_{\boldsymbol{u} \in \boldsymbol{\mathcal{U}}} \boldsymbol{\pi}(\boldsymbol{u} \,|\, \boldsymbol{x}) q_{\boldsymbol{\pi}}(\boldsymbol{x}, \boldsymbol{u}) \right].$$

Following the derivation in [Sut18] leads to the analytical solution of the policy gradient:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\pi}} \left[q_{\boldsymbol{\pi}}(\boldsymbol{x}, \boldsymbol{u}) \frac{\nabla_{\boldsymbol{\theta}} \boldsymbol{\pi}(\boldsymbol{u} \,|\, \boldsymbol{x}, \boldsymbol{\theta})}{\boldsymbol{\pi}(\boldsymbol{u} \,|\, \boldsymbol{x}, \boldsymbol{\theta})} \right] = \mathbb{E}_{\boldsymbol{\pi}} \left[q_{\boldsymbol{\pi}}(\boldsymbol{x}, \boldsymbol{u}) \nabla_{\boldsymbol{\theta}} \ln \boldsymbol{\pi}(\boldsymbol{u} \,|\, \boldsymbol{x}, \boldsymbol{\theta}) \right], \tag{3.7}$$

where the right hand side is derived by applying the identity $\nabla \ln a = \frac{\nabla a}{a}$. The parameter update is then found by plugging (3.7) in the gradient ascent update rule (3.6):

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \, \mathbb{E}_{\boldsymbol{\pi}} \left[q_{\boldsymbol{\pi}}(\boldsymbol{x}, \boldsymbol{u}) \nabla_{\boldsymbol{\theta}} \ln \boldsymbol{\pi}(\boldsymbol{u} \mid \boldsymbol{x}, \boldsymbol{\theta}) \right]. \tag{3.8}$$

The update rules for stochastic policies $\boldsymbol{\pi}(\boldsymbol{u} \mid \boldsymbol{x}, \boldsymbol{\theta})$, derived above, can be transferred to the policy gradient for deterministic policies $\boldsymbol{\eta}(\boldsymbol{u} \mid \boldsymbol{x}, \boldsymbol{\theta})$. [Sil+14] proves the existence of the deterministic policy gradient and derives it as:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\boldsymbol{\eta}} \left[\nabla_{\boldsymbol{\theta}} \boldsymbol{\eta}(\boldsymbol{x}, \boldsymbol{\theta}) \nabla_{\boldsymbol{u}} q(\boldsymbol{x}, \boldsymbol{u})|_{\boldsymbol{u}=\boldsymbol{\eta}(\boldsymbol{x})} \right]. \tag{3.9}$$

Besides the gradient of the deterministic policy $\nabla_{\boldsymbol{\theta}} \boldsymbol{\eta}(\boldsymbol{x}, \boldsymbol{\theta})$, [Sil+14] determines the gradient of the action-value estimator $\nabla_{\boldsymbol{u}} q(\boldsymbol{x}, \boldsymbol{u})|_{\boldsymbol{u}=\boldsymbol{\eta}(\boldsymbol{x})}$ to find the derivative of the objective J.

This approach provides certain improvements to the standard approach using stochastic policies, as stated by [Sil+14]. The DPG provides an off-policy style algorithm in the policy gradient domain and eliminates the dependency on the stochastic policy to explore the environment. This is especially useful when the stochastic policy becomes nevertheless more deterministic during the learning process when going towards a optimal policy and therefore limiting the general exploration behavior. Additionally, [Sil+14] points out difficulties in estimating the gradient for a stochastic policy distribution with a decreasing variance σ^2 due to the policy optimization process. A small variance σ^2, hence a large factor $\frac{1}{\sigma^2}$, leads to unreasonably high values for a policy described by the normal distribution $\mathcal{N}(\mu, \sigma^2)$, leading to jumps in the gradient estimation and therefore unstable learning behavior.

3.1.4. Actor-Critic Algorithms

An elegant approach to combine the DPG and NN function approximators is made possible by using AC algorithms. The optimization problem is split up into the task

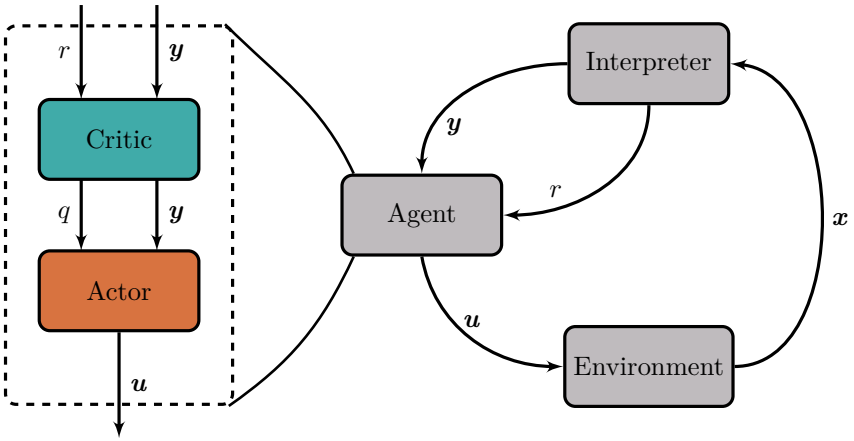

Figure 3.2: Simplified structure of an AC agent using function approximators

to find an optimal action, carried out by the actor and the task to find an accurate estimate of the action-value, done by the critic, as it is displayed in Figure 3.2. When applied to (3.9) and with the use of function approximator, the deterministic policy $\boldsymbol{\eta}(\boldsymbol{x}, \boldsymbol{\theta})$ takes the position of the actor and $q(\boldsymbol{x}, \boldsymbol{u})$ the role of of the critic. Therefore, a basic actor-critic algorithm uses two function approximators to solve the RL task. The policy gradients are employed to optimize the approximator networks by collecting samples from the environment and applying a common optimization technique like the gradient back-propagation algorithm [RHW86].

The DDPG algorithm [Lil+15] extends the AC approach to DPGs with deep neural network and is explained in more detail in the following. Motivated by the action-value estimation of the deep Q networks (DQN) algorithm by [Mni+15], the objective of the critic is chosen by [Lil+15] as:

$$\begin{aligned} J_{\text{critic}}(\boldsymbol{\phi}) &= \frac{1}{|\mathcal{S}|} \sum_{\mathcal{S}} \left[t(r, \boldsymbol{x}_k) - q(\boldsymbol{x}_k, \boldsymbol{\eta}(\boldsymbol{x}_k, \boldsymbol{\theta}), \boldsymbol{\phi}) \right]^2 \text{ with} \\ t(r, \boldsymbol{x}_k) &= r + \gamma q(\boldsymbol{x}_{k+1}, \boldsymbol{\eta}(\boldsymbol{x}_{k+1}, \boldsymbol{\theta}_{\text{target}}), \boldsymbol{\phi}_{\text{target}}), \end{aligned} \tag{3.10}$$

where the $\mathcal{L}_{\text{critic}}(\boldsymbol{\phi})$ is the mean-squared Bellman error (MSBE) between the estimated action-value and the target value $t(r, \boldsymbol{x}_k)$, with the goal to minimize the error during the learning process. The parameters $\boldsymbol{\phi}$ describe the weights of the critic, $\boldsymbol{\theta}$ the actor approximator network and $\mathcal{S}$ the batch of sampled observation-tuples of size $|\mathcal{S}|$.

To follow the results found by [Mni+15], two additional networks, with the parameters $\boldsymbol{\theta}_{\text{target}}$ and $\boldsymbol{\phi}_{\text{target}}$, are introduced as target networks to increase learning-stability during the optimization process. The target network weights are updated every other training episode by a soft update, described by:

$$\begin{aligned} \boldsymbol{\theta}_{\text{target}} &\leftarrow (1-\tau)\,\boldsymbol{\theta}_{\text{target}} + \tau\,\boldsymbol{\theta} \\ \boldsymbol{\phi}_{\text{target}} &\leftarrow (1-\tau)\,\boldsymbol{\phi}_{\text{target}} + \tau\,\boldsymbol{\phi} \end{aligned} \tag{3.11}$$

with the update coefficient $\tau \in]0, 1[$ with τ often chosen near one.

To find the maximal state-action value q, the actor is maximized according to the principle of Q-learning as described in the previous section. To find an actor that maximizes the q [Lil+15] formulates the objective as:

$$J_{\text{actor}}(\boldsymbol{\theta}) = \frac{1}{|\mathcal{S}|} \sum_{\mathcal{S}} q(\boldsymbol{x}_k, \boldsymbol{\eta}(\boldsymbol{x}_k, \boldsymbol{\theta}), \boldsymbol{\phi}) \tag{3.12}$$

with the sampled policy gradient, following the policy gradient (3.9):

$$\nabla_{\boldsymbol{\theta}} J_{\text{actor}} = \frac{1}{|\mathcal{S}|} \sum_{\mathcal{S}} \nabla_{\boldsymbol{u}} q(\boldsymbol{x}_k, \boldsymbol{\eta}(\boldsymbol{x}_k, \boldsymbol{\theta}), \boldsymbol{\phi}) \nabla_{\boldsymbol{\theta}}\, \boldsymbol{\eta}(\boldsymbol{x}_k, \boldsymbol{\theta}). \tag{3.13}$$

The purely deterministic policy does not explore the environment on its own and needs random exploration noise added to the actions. [Lil+15] enables a direct control of the exploration behavior of the agent by employing noise determined by an Ornstein-Uhlenbeck process or white Gaussian noise. The detailed algorithm, using the described equations, is given in Algorithm 1.

Algorithm 1 Deep Deterministic Policy Gradient

input: actor parameters $\boldsymbol{\theta}$, critic parameters $\boldsymbol{\phi}$, empty buffer $\mathcal{M}$
set up target networks with parameters $\boldsymbol{\theta}_{\text{target}} \leftarrow \boldsymbol{\theta}$, $\boldsymbol{\phi}_{\text{target}} \leftarrow \boldsymbol{\phi}$
for train-episodes **do**
  $k \leftarrow 0$
  while not done **do**
    sample exploration noise $\nu \sim \mathcal{N}(0, \sigma_\nu)$
    $\boldsymbol{u}_k = \text{clip}(\boldsymbol{\eta}(\boldsymbol{x}_k, \boldsymbol{\theta}) + \nu, \boldsymbol{u}_{\min}, \boldsymbol{u}_{\max})$
    take step in environment following $\boldsymbol{u}_k$ and observe $(\boldsymbol{x}_{k+1}, r_{k+1})$
    store $(\boldsymbol{x}_k, \boldsymbol{u}_k, r_{k+1}, \boldsymbol{x}_{k+1})$ in buffer $\mathcal{M}$
    if $\boldsymbol{x}_{k+1}$ is terminal **then**
      episode is done
    end if
    if $|\mathcal{M}| > 0$ **then**
      sample batch $\mathcal{S}$ of transition tuples from $\mathcal{M}$
      compute and minimize the loss (3.10)
      compute sampled policy gradient (3.13) and maximize objective
      update target networks following (3.11)
    end if
    $k \leftarrow k + 1$
  end while
end for

3.2. Model Predictive Control

A modern standard method in optimal control is MPC. Model knowledge about the system is used to predict trajectories of control actions and system states. This knowledge is then used to find the optimal control action by minimizing an objective. The next section describes how MPC is connected to the linear quadratic regulator (LQR) and how to solve the control objective via a quadratic program (QP).

3.2.1. Constrained Controller

When describing linear time-invariant system dynamics, the state-space formulation with the structure:

$$\begin{aligned} \boldsymbol{x}_{k+1} &= \boldsymbol{A}\,\boldsymbol{x}_k + \boldsymbol{B}\,\boldsymbol{u}_k, \\ \boldsymbol{y}_k &= \boldsymbol{C}\,\boldsymbol{x}_k + \boldsymbol{D}\,\boldsymbol{u}_k, \\ \text{s.t.} \quad \boldsymbol{x}_k &\in \boldsymbol{\mathcal{X}},\, \boldsymbol{u}_k \in \boldsymbol{\mathcal{U}}, \;\; \boldsymbol{x}(0) = \boldsymbol{x}_0 \end{aligned} \tag{3.14}$$

is commonly used in optimal control literature [BBM17]. The state matrix $\boldsymbol{A}$, the input matrix $\boldsymbol{B}$, the output matrix $\boldsymbol{C}$ and the feedforward matrix $\boldsymbol{D}$ describe the system and $\boldsymbol{x}_k$, $\boldsymbol{u}_k$ the state and action vectors at time step k. As mentioned before, state space and action space are specified by $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{U}}$.

When trying to solve an unconstrained control problem, the linear quadratic regulator is a well known approach. For this, [BBM17] defines a quadratic cost function:

$$J(\boldsymbol{x}_0, \boldsymbol{u}_0) = \boldsymbol{x}_N^{\mathrm{T}}\,\boldsymbol{S}\,\boldsymbol{x}_N + \sum_{k=0}^{N-1} \boldsymbol{x}_k^{\mathrm{T}}\,\boldsymbol{Q}\,\boldsymbol{x}_k + \boldsymbol{u}_k^{\mathrm{T}}\,\boldsymbol{R}\,\boldsymbol{u}_k \tag{3.15}$$

with $\boldsymbol{S}$, $\boldsymbol{Q}$, $\boldsymbol{R}$ as weighting matrices and N as the prediction horizon. The target is to minimize the costs for a predicted, N-step long trajectory. Model predictive control uses the same approach as the LQR, but incorporates state and action

constraints into the optimization problem. For this study, it is presupposed only linear box-constraints are used. This makes it possible to reformulate $\boldsymbol{x}_k \in \boldsymbol{\mathcal{X}}$, $\boldsymbol{u}_k \in \boldsymbol{\mathcal{U}}$ in the polyhedron formulation of the system constraints [BBM17]. Using this and the quadratic costs (3.15), we find the MPC optimization problem:

$$\begin{aligned} J^*(\boldsymbol{x}_0, \boldsymbol{u}_0) = \min_{u_0 \dots u_{N-1}} \boldsymbol{x}_N^{\mathrm{T}} \boldsymbol{S}\, \boldsymbol{x}_N + \sum_{k=0}^{N-1} \boldsymbol{x}_k^{\mathrm{T}} \boldsymbol{Q}\, \boldsymbol{x}_k + \boldsymbol{u}_k^{\mathrm{T}} \boldsymbol{R}\, \boldsymbol{u}_k, \\ \text{s.t.} \quad \boldsymbol{M}_x\, \boldsymbol{x}_k \leq \boldsymbol{w}_x, \quad \boldsymbol{M}_u\, \boldsymbol{u}_k \leq \boldsymbol{w}_u, \end{aligned} \tag{3.16}$$

where $\boldsymbol{M}$ and $\boldsymbol{w}$ describe the constraint-polyhedra.

To solve this constrained optimization problem with a quadratic cost function, quadratic programming is a popular approach. The optimization problem can be reformulated in vector notation leading to:

$$\begin{aligned} J^*(\boldsymbol{x}_0, \boldsymbol{u}_0) = {\boldsymbol{X}_N}^{\mathrm{T}}\, \boldsymbol{Q}_N\, \boldsymbol{X}_N + {\boldsymbol{U}_N}^{\mathrm{T}}\, \boldsymbol{R}_N\, \boldsymbol{U}_N, \\ \text{s.t.} \quad \boldsymbol{M}_{x,N}\, \boldsymbol{X}_N \leq \boldsymbol{w}_{x,N}, \quad \boldsymbol{M}_{u,N}\, \boldsymbol{U}_N \leq \boldsymbol{w}_{u,N} \end{aligned} \tag{3.17}$$

with:

$$\boldsymbol{X}_N = \begin{bmatrix} \boldsymbol{x}_0 \\ \vdots \\ \boldsymbol{x}_N \end{bmatrix}, \quad \boldsymbol{U}_N = \begin{bmatrix} \boldsymbol{u}_0 \\ \vdots \\ \boldsymbol{u}_{N-1} \end{bmatrix},$$

$$\boldsymbol{M}_{x,N} = \begin{bmatrix} \boldsymbol{M}_x & & \boldsymbol{0} \\ & \ddots & \\ \boldsymbol{0} & & \boldsymbol{M}_x \end{bmatrix}, \boldsymbol{w}_{x,N} = \begin{bmatrix} \boldsymbol{w}_x \\ \vdots \\ \boldsymbol{w}_x \end{bmatrix}, \quad \boldsymbol{M}_{u,N} = \begin{bmatrix} \boldsymbol{M}_u & & \boldsymbol{0} \\ & \ddots & \\ \boldsymbol{0} & & \boldsymbol{M}_u \end{bmatrix}, \boldsymbol{w}_{u,N} = \begin{bmatrix} \boldsymbol{w}_u \\ \vdots \\ \boldsymbol{w}_u \end{bmatrix}.$$

Through restructuring the costs (3.17) and slightly reformulating the terms, the optimization problem can be put in the form of a quadratic program, described by [BBM17] as:

$$\begin{aligned} \min_{\boldsymbol{u}} \ & \frac{1}{2}\,\boldsymbol{u}^{\mathrm{T}}\,\boldsymbol{H}\,\boldsymbol{u} + \boldsymbol{f}^{\mathrm{T}}\,\boldsymbol{u}, \\ \text{s.t.} \ & \boldsymbol{G}\,\boldsymbol{u} \leq \boldsymbol{e}, \end{aligned} \tag{3.18}$$

where the elements of the QP for the MPC problem are derived by [BBM17] as:

$$\begin{aligned} \boldsymbol{H} &= 2\left(\boldsymbol{R}_N + \boldsymbol{B}_N{}^{\mathrm{T}}\,\boldsymbol{Q}_N\,\boldsymbol{B}_N\right), \quad \boldsymbol{f} = 2\,\boldsymbol{B}_N{}^{\mathrm{T}}\,\boldsymbol{Q}_N\,\boldsymbol{A}_N\,\boldsymbol{x}, \\ \boldsymbol{G} &= \begin{bmatrix} \boldsymbol{M}_{x,N}\,\boldsymbol{B}_N \\ \boldsymbol{M}_{u,N} \end{bmatrix}, \quad \boldsymbol{e} = \begin{bmatrix} \boldsymbol{w}_{x,N} - \boldsymbol{M}_{x,N}\,\boldsymbol{A}_N\,\boldsymbol{x} \\ \boldsymbol{w}_{u,N} \end{bmatrix} \end{aligned} \tag{3.19}$$

with the matrices $\boldsymbol{A}_N$ and $\boldsymbol{B}_N$ structured as:

$$\boldsymbol{A}_N = \begin{bmatrix} \boldsymbol{I} \\ \boldsymbol{A} \\ \vdots \\ \boldsymbol{A}^N \end{bmatrix}, \boldsymbol{B}_N = \begin{bmatrix} \boldsymbol{0} & \cdots & \boldsymbol{0} \\ \boldsymbol{B} & & \boldsymbol{0} \\ & \ddots & \\ \boldsymbol{A}^{N-1}\boldsymbol{B} & & \boldsymbol{B} \end{bmatrix}.$$

The solution to the QP is the vector $\boldsymbol{u}^*$ which represents the optimal control trajectory of the N-step predicted horizon. The first element of the solution vector is then applied to the system. The QP has then to be solved again in each time step with a horizon shifted one step further. Because of that, the MPC control problem is often referred to as receding horizon control by [BBM17]. The general process described is illustrated in Figure 3.3. After the prediction of the state trajectory and determining the corresponding control trajectory at time k, as displayed in the first plot, the process is repeated for the next step $k+1$ with the horizon shifted one step ahead, as displayed in the second plot.

Specific solvers for this constrained optimization problems are e.g., the standard quadprog solver or the COBYLA solver which are often available in several implementations [JOP+01]. A more in-depth look into specific solver algorithms is beyond the scope of this thesis.

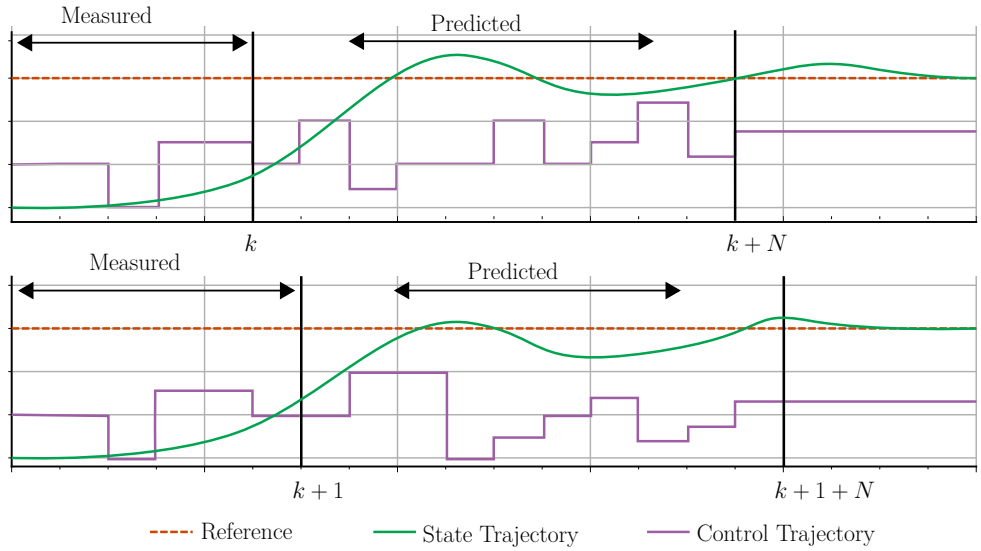

Figure 3.3: Schematic representation of the receding horizon control method used in MPC

3.2.2. Recursive Feasibility

An important property of an optimal controller is the ability to control the system to the targeted state. This property is contained in the stability and controllability properties of an optimal controller. Unconstrained controllers, like the mentioned LQR, have proven stability guarantees by default [BBM17]. However, constrained approaches like MPC are often not so trivial to solve. MPC controllers have to find a control input which will satisfy the system constraints for every following state. Therefore, the QP of the current and all future states have to be feasible.

[BBM17] describes an approach to find a set of feasible states for the given QP by restructuring parts of the original QP (3.18) into the parametric form:

$$\begin{aligned} \min_{\boldsymbol{u}} \ & \frac{1}{2}\,\boldsymbol{u}^{\mathrm{T}}\,\boldsymbol{H}\,\boldsymbol{u} + \boldsymbol{x}^{\mathrm{T}}\,\boldsymbol{F}^{\mathrm{T}}\,\boldsymbol{u}, \\ & \text{s.t. } \boldsymbol{G}\,\boldsymbol{u} \leq \boldsymbol{E}\,\boldsymbol{x} + \boldsymbol{d} \end{aligned} \tag{3.20}$$

with:

$$\boldsymbol{F} = 2\,{\boldsymbol{B}_N}^{\mathrm{T}}\,\boldsymbol{Q}_N\,\boldsymbol{A}_N\,\boldsymbol{x}, \quad \boldsymbol{E} = \begin{bmatrix} \boldsymbol{w}_{x,N} - \boldsymbol{M}_{x,N}\,\boldsymbol{A}_N\,\boldsymbol{x} \\ \boldsymbol{w}_{u,N} \end{bmatrix} \text{ and } \boldsymbol{d} = \begin{bmatrix} \boldsymbol{w}_{x,N} \\ \boldsymbol{w}_{u,N} \end{bmatrix}. \tag{3.21}$$

With the state vector $\boldsymbol{x}$ as part of the solution of the multiparametric program, the feasible set is denoted by $\boldsymbol{\mathcal{F}} = \{\boldsymbol{x} \in \boldsymbol{\mathcal{X}} | \quad \exists\, \boldsymbol{u} \in \mathbb{R}^m : \boldsymbol{G}\,\boldsymbol{u} \leq \boldsymbol{E}\,\boldsymbol{x} + \boldsymbol{d}\}$.

To find this set, [BBM17] calculates the polyhedron of all valid state-action vectors $\begin{bmatrix}\boldsymbol{x}^{\mathrm{T}} & \boldsymbol{u}^{\mathrm{T}}\end{bmatrix}^{\mathrm{T}}$ of the QP. This is done by transforming all inequalities describing the state and action constraints with respect to $\boldsymbol{x}$, $\boldsymbol{u}$ and stacking them together to one polyhedron formulation described by:

$$\boldsymbol{\mathcal{G}} = \left\{ \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{u} \end{bmatrix} \in \mathbb{R}^{n+m} \quad \middle| \quad \begin{bmatrix} \boldsymbol{M}_x & 0 \\ -\boldsymbol{E} & \boldsymbol{G} \end{bmatrix} \leq \begin{bmatrix} \boldsymbol{w}_x \\ \boldsymbol{d} \end{bmatrix} \right\}. \tag{3.22}$$

The projection of $\boldsymbol{\mathcal{G}}$ on to the n-dimensional state space corresponds to the set $\boldsymbol{\mathcal{F}}$.

To find the set which is feasible for all future states and therefore secures the system constraints is called constrained invariant set. The invariant set can be determined by calculating the feasible sets for all $N \to \infty$ future steps. Because an infinite horizon is not applicable in practice, the algorithm described by [BBM17] calculates the sets recursively until the difference of the sets converges to zero. The detailed procedure is described in Algorithm 2. After using the state-action space as initial set, the feasible set is calculated for an increasing prediction horizon. By then determining the volume of $\boldsymbol{\mathcal{G}}^{i-1} \cap \boldsymbol{\mathcal{G}}^{i}$ as a convergence-measure, the algorithm is executed until the volume difference falls below a certain tolerance threshold

Algorithm 2 Calculate the largest feasible set

parameter: constraint-matrices $\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{U}}$, $\boldsymbol{G}$, $\boldsymbol{E}$, $\boldsymbol{d}$, convergence threshold ξ
init: feasible polyhedron $\boldsymbol{\mathcal{G}}_0 \leftarrow \boldsymbol{\mathcal{X}} \cup \boldsymbol{\mathcal{U}}$
for 1 ... i ... ∞ **do**
  calculate $\boldsymbol{\mathcal{G}}_i$ for horizon of length i
  calculate $\Delta\boldsymbol{\mathcal{G}}_i = \boldsymbol{\mathcal{G}}_{i-1} \cap \boldsymbol{\mathcal{G}}_i$
  if Volume$(\Delta\boldsymbol{\mathcal{G}}_i) \leq \xi$ **then**
    algorithm converged
  end if
end for
$\boldsymbol{\mathcal{G}}^{max} \leftarrow \boldsymbol{\mathcal{G}}_i$

ξ and the algorithm converges. The same algorithm can be followed to find the largest feasible set $\mathcal{F}^{\max}$ only taking the state space into account.

An exemplary process of executing Algorithm 2 is visualized in Figure 3.4. Starting with the original state space, the sets are shrinking with increasing prediction horizon N. When the optimal controller starts within the feasible set, it is guaranteed, that all future states are lying within the feasible solutions for the QP.

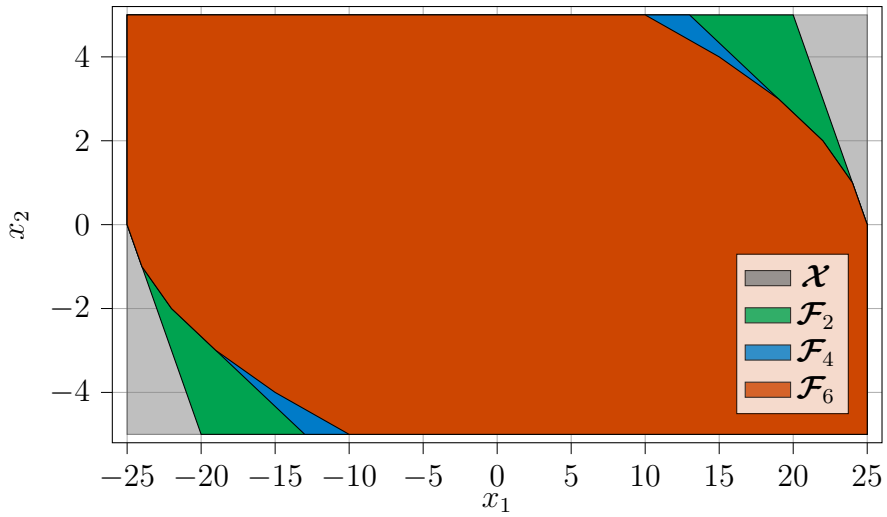

Figure 3.4: Exemplary illustration of the state-space projection $\mathcal{F}$ of the feasible set $\boldsymbol{\mathcal{G}}$ for various prediction horizons H for the double integrator system (section 5.1) with the sampling time $T_s = 1, \boldsymbol{Q} = \boldsymbol{I}, R = 1, -25 \leq x_1 \leq 25, -5 \leq x_2 \leq 5, -1 \leq u \leq 1$

3.3. System Identification

When a model for the controller is not known, the system has to be identified from data. Techniques achieving this stem from the field of system identification. Because this work focuses on linear systems, the identification methods used concentrate on parametric identification of linear, dynamic models. Powerful methods in this category are least squares (LS) algorithms. Given measurements from the unknown system, the least-squares approach tries to find a linear model, which will minimize the squared error between estimated and measured samples.

The known state space structure (3.14) can be reformulated as the difference equation of the linear system, leading to:

$$x_{i,k+1} = a_{i1}x_{i,k} + \cdots + a_{in}x_{n,k} + b_{i1}u_{1,k} + \cdots + b_{in}u_{m,k}, \tag{3.23}$$

where i is the state index of the predicted state, n the state and m the input dimension of the system. Following [Rol11] the difference equation can be restructured and put into vector notation:

$$\begin{aligned} x_{i,k+1} &= \begin{bmatrix} x_{1,k} & \cdots & x_{n,k} & u_{1,k} & \cdots & u_{m,k} \end{bmatrix} \begin{bmatrix} a_{i1} \\ \vdots \\ a_{in} \\ b_{i1} \\ \vdots \\ b_{in} \end{bmatrix} \\ &= \begin{bmatrix} \boldsymbol{x}_k^{\mathrm{T}} & \boldsymbol{u}_k^{\mathrm{T}} \end{bmatrix} \begin{bmatrix} \boldsymbol{a}_i \\ \boldsymbol{b}_i \end{bmatrix}. \end{aligned} \tag{3.24}$$

By extending (3.24) for K data samples, the matrix-form of the least-squares prediction equation is found as:

$$\underbrace{\begin{bmatrix} x_{i,2} \\ \vdots \\ x_{i,K+1} \end{bmatrix}}_{\boldsymbol{\psi}_i} = \underbrace{\begin{bmatrix} \boldsymbol{x}_1^{\mathrm{T}} & \boldsymbol{u}_1^{\mathrm{T}} \\ \vdots & \vdots \\ \boldsymbol{x}_K^{\mathrm{T}} & \boldsymbol{u}_K^{\mathrm{T}} \end{bmatrix}}_{\boldsymbol{\Xi}_i} \underbrace{\begin{bmatrix} \boldsymbol{a}_i \\ \boldsymbol{b}_i \end{bmatrix}}_{\boldsymbol{\theta}_i} + \boldsymbol{e}_i, \tag{3.25}$$

with the measurement or data vector $\boldsymbol{\psi}_i$, the regressor matrix $\boldsymbol{\Xi}_i$, the parameter vector $\boldsymbol{\theta}_i$ and the error vector $\mathbf{e}_i$. By defining the squared estimation error as the cost function, which will be minimized, following [Rol11] leads to:

$$\begin{aligned} J(\boldsymbol{\theta}_i) &= \sum_{k=1}^{K} (e_{i,k})^2 = (\boldsymbol{\psi}_i - \boldsymbol{\Xi}_i \, \boldsymbol{\theta}_i)^{\mathrm{T}} \, (\boldsymbol{\psi}_i - \boldsymbol{\Xi}_i \, \boldsymbol{\theta}_i) \\ &= \boldsymbol{\psi}_i^{\mathrm{T}} \, \boldsymbol{\psi}_i - \boldsymbol{\theta}_i^{\mathrm{T}} \, \boldsymbol{\Xi}_i^{\mathrm{T}} \, \boldsymbol{\psi}_i - \boldsymbol{\theta}_i (\boldsymbol{\Xi}_i^{\mathrm{T}} \, \boldsymbol{\psi}_i)^{\mathrm{T}} + \boldsymbol{\theta}_i^{\mathrm{T}} \, \boldsymbol{\Xi}_i^{\mathrm{T}} \boldsymbol{\Xi}_i \, \boldsymbol{\theta}_i \, . \end{aligned} \tag{3.26}$$

Minimization of the cost function (3.26) with respect to the parameter vector and solving for $\boldsymbol{\theta}$ leads to:

$$\begin{aligned} \boldsymbol{\theta}_i^* &= \underset{\boldsymbol{\theta}_i}{\operatorname{argmin}} \; J(\boldsymbol{\theta}_i), \\ \Rightarrow \frac{\partial}{\partial \, \boldsymbol{\theta}_i} J(\boldsymbol{\theta}_i) &= -2(\boldsymbol{\Xi}_i^{\mathrm{T}} \, \boldsymbol{\psi}_i)^{\mathrm{T}} + 2 \, \boldsymbol{\theta}_i^{\mathrm{T}} \, \boldsymbol{\Xi}_i^{\mathrm{T}} \boldsymbol{\Xi}_i \stackrel{!}{=} 0 \\ \Leftrightarrow \boldsymbol{\theta}_i^* &= \left(\boldsymbol{\Xi}_i^{\mathrm{T}} \boldsymbol{\Xi}_i \right)^{-1} \boldsymbol{\Xi}_i^{\mathrm{T}} \, \boldsymbol{\psi}_i, \end{aligned} \tag{3.27}$$

where the product $\boldsymbol{\Xi}_i^{\mathrm{T}} \boldsymbol{\Xi}_i$ has to be invertible.

By solving this problem for every i-th state, it is possible to determine all coefficients $\boldsymbol{a}_i, \boldsymbol{b}_i$ to find and estimate the system matrices $\hat{\boldsymbol{A}}, \hat{\boldsymbol{B}}$ of the state-space representation.

To identify the system parameters online, the ordinary least squares (OLS) can be extended to the so-called recursive least squares (RLS) algorithm. The estimates

are calculated by using the incoming measurements sample by sample and not by collecting the data and building measurement matrices from multiple observations as done before.

By starting with the parameter estimate (3.27), [Rol11] uses the definition of the sample wise parameter estimate and the model equation to calculate the measurement $\boldsymbol{\psi}_k$ at step k:

$$\begin{aligned} \hat{\boldsymbol{\theta}}_k &= \left(\boldsymbol{\Xi}_k^{\mathrm{T}} \boldsymbol{\Xi}_k\right)^{-1} \boldsymbol{\Xi}_k^{\mathrm{T}} \boldsymbol{\psi}_k, \\ \boldsymbol{\psi}_k &= \boldsymbol{\Xi}_k \hat{\boldsymbol{\theta}}_k. \end{aligned} \tag{3.28}$$

Using the estimates for the next time step $k+1$, both terms can be put together in vector notation as shown in:

$$\boldsymbol{\psi}_k = \begin{bmatrix} \boldsymbol{\psi}_k \\ \boldsymbol{\psi}_{k+1} \end{bmatrix}, \quad \hat{\boldsymbol{\theta}}_k = \begin{bmatrix} \hat{\boldsymbol{\theta}}_k \\ \hat{\boldsymbol{\theta}}_{k+1} \end{bmatrix}.$$

Inserting this into (3.28) and multiplying out the equation leads to:

$$\begin{aligned} \hat{\boldsymbol{\theta}}_{k+1} &= (\boldsymbol{\Xi}_k^{\mathrm{T}} \boldsymbol{\Xi}_k + \boldsymbol{\xi}_{k+1} \boldsymbol{\xi}_{k+1}^{\mathrm{T}})^{-1} (\boldsymbol{\Xi}_k^{\mathrm{T}} \boldsymbol{\Xi}_k \hat{\boldsymbol{\theta}}_k + \boldsymbol{\xi}_{k+1} \boldsymbol{\psi}_{k+1}) \\ &= (\boldsymbol{P}_k^{-1} + \boldsymbol{\xi}_{k+1} \boldsymbol{\xi}_{k+1}^{\mathrm{T}})^{-1} (\boldsymbol{P}_k^{-1} \hat{\boldsymbol{\theta}}_k + \boldsymbol{\xi}_{k+1} \boldsymbol{\psi}_{k+1}), \end{aligned} \tag{3.29}$$

where [Rol11] introduces the covariance matrix $\boldsymbol{P}_k^{-1} = \boldsymbol{\Xi}_k^{\mathrm{T}} \boldsymbol{\Xi}_k$. Following the mathematical reformulations of [Rol11], (3.29) is transformed to the essential formulas of the RLS algorithm:

$$\begin{aligned} \kappa_k &= \frac{\boldsymbol{P}_k \boldsymbol{\xi}_{k+1}}{\lambda_{k+1} + \boldsymbol{\xi}_{k+1}^{\mathrm{T}} \boldsymbol{P}_k \boldsymbol{\xi}_{k+1}} \\ \hat{\boldsymbol{\theta}}_{k+1} &= \hat{\boldsymbol{\theta}}_k + \kappa_k \left(\boldsymbol{\psi}_{k+1} - \boldsymbol{\xi}_{k+1}^{\mathrm{T}} \hat{\boldsymbol{\theta}}_k\right) \\ \boldsymbol{P}_{k+1} &= \left(\boldsymbol{I} - \kappa_k \boldsymbol{\xi}_{k+1}^{\mathrm{T}}\right) \boldsymbol{P}_k \frac{1}{\lambda_{k+1}} \end{aligned} \tag{3.30}$$

The forgetting factor λ inserted in the amplification term κ_k and the recursive estimation of the covariance matrix $\boldsymbol{P}_{k+1}$ adds a weighting to the measurements encountered during the identification process. A factor $\lambda < 1$ changes the weighting from a standard LS approach with equal weighting ($\lambda = 1$), to an exponential weighting, used in the RLS method, shown in Figure 3.5. Hence, measurements encountered earlier in the identification process have less influence on the current estimate, than newer ones.

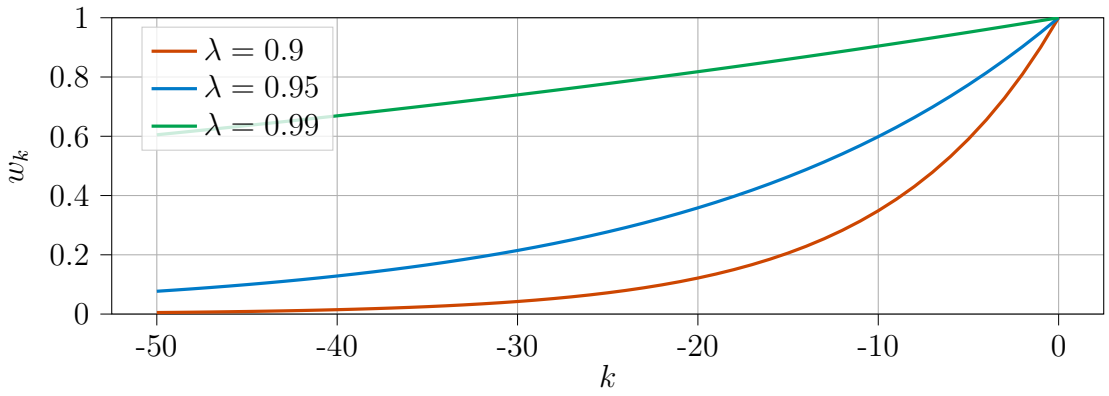

Figure 3.5: Representative weighting w_k of past observations for different forgetting factors λ employed in the RLS algorithm.

4. System Development

The methods introduced in chapter 3 are combined to form a reinforcement learning controller using a model of the environment's dynamics to avoid constraint violations and increase safety during training. The following chapter introduces the algorithm developed in this work and explains the controller's structure and the concepts applied.

4.1. The Model Adaptive Safeguard Controller

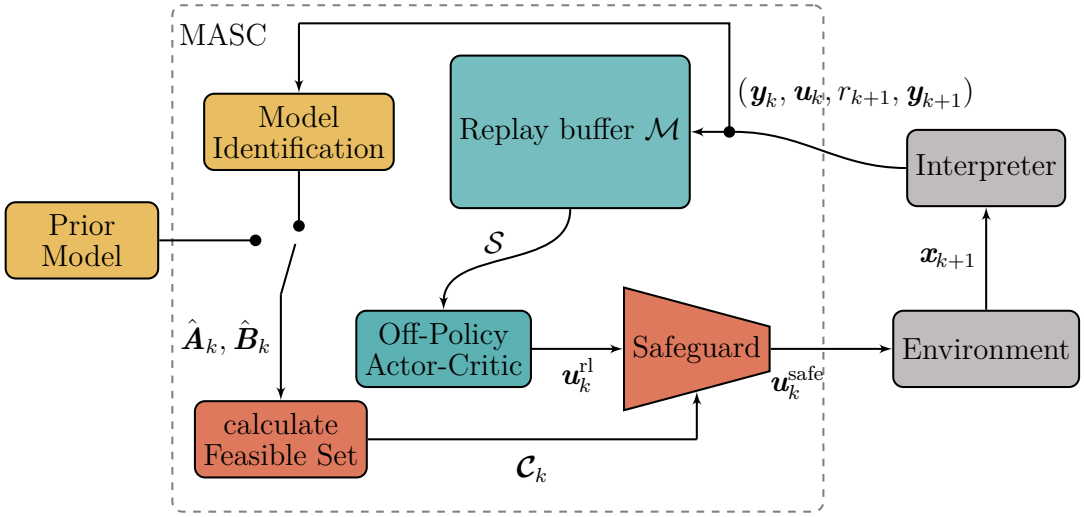

Figure 4.1: Block diagram of the developed controller structured in RL agent, safeguard and system identification module

The model adaptive safeguard controller (MASC) aims to learn an optimal control strategy in the RL context by respecting the environment's constraints without depending on expert knowledge about the physical system. As displayed in Figure 4.1, the controller is structured in different modules, each of which tackles a

different task. An actor-critic RL agent learns and optimizes the control policy $\boldsymbol{\pi}(\boldsymbol{x}, \boldsymbol{\theta})$. This is done by interacting with the environment and following the principles described in section 3.1. The action chosen by the AC agent $\boldsymbol{u}_k^{\mathrm{RL}}$ does not necessary respect the constraints $\boldsymbol{\mathcal{C}}_k$ and therefore is given as input to the safeguard. The safeguard checks whether the action at time k violates constraints in the future or obeys them. If the action is seen as unsafe, $\boldsymbol{u}_k^{\mathrm{RL}}$ is modified to $\boldsymbol{u}_k^{\mathrm{safe}}$, such that the constraints $\boldsymbol{\mathcal{C}}_k$ are respected but the original action $\boldsymbol{u}_k^{\mathrm{RL}}$ is modified as small as possible.

The safeguard needs knowledge about the dynamics of the environment to ensure constraint satisfaction. It is possible to incorporate an prior available model, but to reduce the need for prior available knowledge, this work incorporates a system identification algorithm to find the model estimates $\hat{\boldsymbol{A}}_k, \hat{\boldsymbol{B}}_k$.

The modular structure allows to freely select the algorithms implemented in each module. Detailed explanations of the methods used and evaluated in this work follow in the sections below.

4.2. Modules

4.2.1. Reinforcement Learning Controller

As AC method for continuous state and action spaces, this work employs the DDPG algorithm, described in section 3.1. Because the agent's learning process is not directly interfered by the safeguard, an arbitrary RL agent can be applied to the system. Promising results and the intuitive approach of off-policy actor-critic methods motivate the use of the DDPG.

The standard DDPG approach, is extended by decaying the exploration noise $\nu \sim \mathcal{N}(0, \sigma_\nu)$. This is implemented by decreasing the variance of the normally distributed noise added to the control-action in every step the agent takes in the environment and is calculated by:

$$\sigma_{\nu,k+1} = (\sigma_{\nu,\mathrm{start}} - \sigma_{\nu,\mathrm{end}}) d_{\mathrm{rate}}^{\left(\frac{k}{d_{\mathrm{steps}}}\right)} + \sigma_{\nu,\mathrm{end}}, \tag{4.1}$$

where $\sigma_{\nu,\text{start}}$ and $\sigma_{\nu,\text{end}}$ describe the start and end value of the standard deviation of the normal distribution to sample the exploration noise from. The decay rate d_{rate} and the decay steps d_{steps} characterize how fast and steep the parameter σ_ν decays. The agent focuses more on taking random actions and exploring the state-space when the added exploration noise is higher at the beginning of learning. When the agent has already learned a certain policy and is nearly optimized, the lower exploration noise does not disrupt the optimization as much as at the beginning.

4.2.2. Safeguard

The safeguard extends the standard RL agent to a constraint satisfying, off-policy controller. By following the idea of [Dal+18] and [WZ18], the guard modifies the RL action $\boldsymbol{u}_k^{\text{RL}}$ by solving a constraint optimization problem similar to the problem formulated in section 3.2. The goal is to modify the action selected by the agent and not to violate any system constraints. To do this, the objective is described via the quadratic, convex costs as:

$$\begin{aligned} J_{\text{guard}}(\boldsymbol{u}_k^{\text{RL}}) &= || \, \boldsymbol{u}_k - \boldsymbol{u}_k^{\text{RL}} \, ||^2 \\ \text{s.t. } \boldsymbol{u}_k &\in \boldsymbol{\mathcal{C}}_u, \end{aligned} \tag{4.2}$$

with the optimal solution defined as

$$\begin{aligned} \boldsymbol{u}_k^{\text{safe}} = \boldsymbol{u}_k^* = \underset{\boldsymbol{u}_k}{\text{argmin}} \; &|| \, \boldsymbol{u}_k - \boldsymbol{u}_u^{\text{RL}} \, ||^2 \\ \text{s.t. } \boldsymbol{u}_k &\in \boldsymbol{\mathcal{C}}_u. \end{aligned} \tag{4.3}$$

The problem is solved following the techniques described in section 3.2 and applying available solvers for constrained optimization. The optimal solution corresponds to the safe action $\boldsymbol{u}_k^{\text{safe}}$ which fulfills the constraints and differs as little as possible from the original action $\boldsymbol{u}_k^{\text{RL}}$.

The constraint optimization problem is approached differently by [Dal+18] and [WZ18]. [Dal+18] formalizes a QP with linear constraints and a linearized model,

which can be solved analytically by applying the Karush-Kuhn-Tucker (KKT) conditions. In contrast, [WZ18] utilizes polyhedral constraints and focuses more on an approach in an MPC fashion. An N-step predicted trajectory is considered to evaluate the starting action for safety. The action is then modified similarly to (4.2).

To eliminate the need to predict N-steps ahead of every control step and to allow nonlinear constraints, this work utilizes the control-invariant feasible set $\boldsymbol{\mathcal{G}}$ of the controlled system as safeguard constraints. By making sure the control action keeps the state in the control-invariant set, the feasibility of all future steps is guaranteed. The constraints can then be written as polyhedron in the form:

$$\boldsymbol{M}_{\boldsymbol{\mathcal{G}}} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{u} \end{bmatrix} \leq \boldsymbol{w}_{\boldsymbol{\mathcal{G}}}. \tag{4.4}$$

It can be calculated by following Algorithm 2 with the use of the model matrices $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{U}}$. By determining a feasible set for the state-action space $\mathbb{R}^{n+m}$ the constraint fulfillment is verified by checking if the state-action pair $\begin{bmatrix} \boldsymbol{x}^{\mathrm{T}} & \boldsymbol{u}^{\mathrm{T}} \end{bmatrix}^{\mathrm{T}}$ lies within the feasible set $\boldsymbol{\mathcal{G}}$ with no need to predict future states. When the action $\boldsymbol{u}^{rl}$ does not move the state-action pair out of $\boldsymbol{\mathcal{G}}$, the action is safe and does not need to be modified.

To encourage the RL agent not only to rely on the safeguard to obey the system's constraints, the additional safeguard penalty r^{SG} is introduced. Every step which actuates the safeguard is penalized by a small negative constant reward. It is chosen such that the overall penalty has a significant learning effect on the agent, but small enough to not disrupt the learning process as much as a constraint violation penalty does.

4.2.3. Model Identification

To determine the dynamic model to calculate the feasible set, several identification methods such as LS algorithm use environment observations to estimate the model parameters. RL algorithms often operate with an experience replay buffer and have stored the observations needed to estimate model parameters by sampling batches from the replay buffer.

This is not the best choice when the latency between collecting observations and calculating model estimates is critical and should be as small as possible. That motivates the use of the RLS algorithm in this thesis. The RLS operates online and determines the model estimates $\hat{\boldsymbol{A}}, \hat{\boldsymbol{B}}$ directly when a new state is observed and continuously improves the estimates, as explained in section 3.3.

4.3. Module Integration

All modules combined form the MASC and follow Algorithm 3. The environment constraints are known and described by the inequality relations defined by the matrices $\boldsymbol{M}_x, \boldsymbol{w}_x, \boldsymbol{M}_u, \boldsymbol{w}_u$. The DDPG agent is initialized as depicted in Algorithm 1. The RLS takes the initial covariance matrix $\boldsymbol{P}_0 = \frac{1}{\delta}$, with δ as a small positive constant, and the forgetting factor λ. The entries of the initial matrices for the system identification $\hat{\boldsymbol{A}}_0, \hat{\boldsymbol{B}}_0$ are randomly sampled from the uniform distribution $\mathcal{U} \sim [0, 1]$. The initial model estimates are then used to determine the feasible set $\boldsymbol{\mathcal{G}}_0$ with Algorithm 2.

The RLS algorithm estimates new parameters as soon as observations are collected from the environment. The safeguard constraints are only updated if the model estimates change significantly. This is done by calculating the normalized difference of the estimated values as described by:

$$\Delta \boldsymbol{A}_k = \frac{||\hat{\boldsymbol{A}}_k - \hat{\boldsymbol{A}}_{k-1}||_F}{||\hat{\boldsymbol{A}}_k||_F}, \Delta \boldsymbol{B}_k = \frac{||\hat{\boldsymbol{B}}_k - \hat{\boldsymbol{B}}_{k-1}||_F}{||\hat{\boldsymbol{B}}_k||_F}, \tag{4.5}$$

with $||\cdot||_F$ as the Frobeniusnorm. If the difference $\Delta \boldsymbol{A}_k$ or $\Delta \boldsymbol{B}_k$ reaches a certain threshold ϵ, the safeguard constraints are updated based on the newest model estimates. This enables the MASC to adjust to the model dynamics and to determine the feasible set without prior model knowledge. To decrease the influence of jumps in the estimated parameters, the check for significant model-change is only executed every i-th step. The model adaption has therefore a slightly higher latency to react to dynamic changes of the environment. This can be neglected if the dynamics only change slowly or do not change at all in the case of a linear time invariant (LTI) system. This helps to further reduce the need to recalculate the feasible sets for each newly estimated set of parameters and and keeps the es-

timated feasible sets from varying too much until more accurate model estimates are determined.

In order to intercept potential errors that arise due to the variance of the model-estimation, the constraints are scaled accordingly to create a tighter constrained area. This ensures that the constraints lie withing the true feasible set, even if the estimated model is not fully accurate. The state variance $\mathrm{var}(\boldsymbol{\psi})$ based on the estimated model is determined by following the derivation in section A.1 that leads to the result:

$$\mathrm{var}(\boldsymbol{\psi}_k) = \boldsymbol{x}_k^{\mathrm{T}} \boldsymbol{P}_k \boldsymbol{x}_k, \tag{4.6}$$

with the RLS covariance matrix $\boldsymbol{P}_k$ at time step k. The constraints (4.4) are then scaled with the variance and form the new inequality constraints:

$$\boldsymbol{M}_{\boldsymbol{\mathcal{G}}} \begin{bmatrix} 1 + \mathrm{var}(\psi_{1,k}) \\ \vdots \\ 1 + \mathrm{var}(\psi_{k,n}) \\ \boldsymbol{1} \end{bmatrix} \leq \boldsymbol{w}_{\boldsymbol{\mathcal{G}}}, \tag{4.7}$$

with $\boldsymbol{M}_{\boldsymbol{\mathcal{G}}}$ and $\boldsymbol{w}_{\boldsymbol{\mathcal{G}}}$ describing the feasible set $\boldsymbol{\mathcal{G}}$ in polyhedron notation and $\boldsymbol{1} \in \mathbb{R}^m$ the one vector.

Throughout the rest of the agent-environment interaction, the model identification is continuously executed. The RL agent is trained with the safeguard ensuring constraint satisfaction for every action. To integrate the safeguard penalty into the learning process the agent stores two observation tuples if the safeguard modified the RL action. Besides the tuple with the observations following the safe control action $\left(\boldsymbol{x}_k, \boldsymbol{u}_k^{\mathrm{safe}}, r_{k+1}, \boldsymbol{x}_{k+1}\right)$, the tuple $\left(\boldsymbol{x}_k, \boldsymbol{u}_k^{\mathrm{RL}}, r^{\mathrm{SG}}, \boldsymbol{x}_{k+1}\right)$ is stored in the replay-buffer. It contains the original unsafe RL action with the resulting safeguard penalty r^{SG} issued because of the actuated safeguard. Hence, the agent gets the reward on the action $\boldsymbol{u}_k^{\mathrm{safe}}$ executed on the environment and the penalty on the originally chosen action $\boldsymbol{u}_k^{\mathrm{RL}}$.

The training finishes once the agent converges or an episode limit is reached.

Algorithm 3 Model Adaptive Safeguard Controller

init:
state-space constraints $\boldsymbol{M}_x, \boldsymbol{w}_x \leftarrow \boldsymbol{\mathcal{X}}$ and $\boldsymbol{M}_u, \boldsymbol{w}_u \leftarrow \boldsymbol{\mathcal{U}}$
initialize model $\hat{\boldsymbol{A}}_0, \hat{\boldsymbol{B}}_0$
determine initial feasible set $\boldsymbol{\mathcal{G}}_0$, safeguard update-steps i
initialize DDPG $\leftarrow \boldsymbol{\theta}, \boldsymbol{\phi}, \sigma_{\nu,\text{start}}$, safeguard $\leftarrow \boldsymbol{\mathcal{G}}_0$ and RLS $\leftarrow \lambda, \boldsymbol{P}$
for train-episodes **do**
  $k \leftarrow 0$
  $\boldsymbol{x}_0 \leftarrow \boldsymbol{x}$
  while not done **do**
    $\boldsymbol{u}_k^{\text{RL}} \leftarrow$ DDPG-control($\boldsymbol{x}_k$)
    if $\begin{bmatrix}\boldsymbol{x}_k^{\text{T}} & \boldsymbol{u}_k^{\text{RL}^{\text{T}}}\end{bmatrix}^{\text{T}}$ is in $\boldsymbol{\mathcal{G}}_k$ **then**
      $\boldsymbol{u}_k^{\text{safe}} \leftarrow \boldsymbol{u}_k^{\text{RL}}$
    else
      $\boldsymbol{u}_k^{\text{safe}} = \boldsymbol{u}_k^{*} \leftarrow$ solve (4.3) with input $\boldsymbol{u}_k^{\text{RL}}$
      issue safeguard penalty r^{SG}
    end if
    take step in environment following $\boldsymbol{u}_k^{\text{safe}}$ and observe $(\boldsymbol{x}_{k+1}, r_{k+1})$
    store safe observation tuple $(\boldsymbol{x}_k, \boldsymbol{u}_k^{\text{safe}}, r_{k+1}, \boldsymbol{x}_{k+1})$ in buffer $\mathcal{M}$
    if $\boldsymbol{u}_k^{\text{safe}} \neq \boldsymbol{u}_k^{\text{RL}}$ **then**
      store penalized observation tuple $(\boldsymbol{x}_k, \boldsymbol{u}_k^{\text{RL}}, r^{\text{SG}}, \boldsymbol{x}_{k+1})$ in buffer $\mathcal{M}$
    end if
    if $\boldsymbol{x}_{k+1}$ is terminal **then**
      episode is done
    end if
    if $|\mathcal{M}| > 0$ **then**
      train DDPG controller
    end if
    estimate model parameters $\hat{\boldsymbol{A}}_k, \hat{\boldsymbol{B}}_k$
    if $\Delta\hat{\boldsymbol{A}}_k \geq \epsilon$ or $\Delta\hat{\boldsymbol{B}}_k \geq \epsilon$ **then**
      if $\text{mod}(i, k) = 0$ **then**
        determine new feasible set $\boldsymbol{\mathcal{G}}_k$
        safeguard $\leftarrow \boldsymbol{\mathcal{G}}_k$
      end if
    end if
    $k \leftarrow k + 1$
  end while
end for

5. Evaluation

The next chapter presents the findings of this thesis. At first, the metrics used in the evaluation are introduced, followed by a description of the simulation environment and the reward functions used. The chapter ends with the presentation of the observations made during the evaluation and a discussion of the results.

5.1. Requirements

The requirements to evaluate the results and findings are introduced in this section, followed by the hyperparameters and sources used to implement the developed controller.

5.1.1. Metrics

The metrics introduced compare and evaluate different aspects of the MASC with focus on the general controller performance and the ability to respect system constraints. To determine how fast and accurate the controller can control the system to the given reference, the mean-squared-error (MSE) cumulated over one episode is introduced and is calculated as the cumulated MSE (CMSE):

$$\text{CMSE} = \sum_{k=1}^{K} \left[\frac{1}{n} \sum_{i=1}^{n} \left(\widetilde{x}_{k,i} - \widetilde{x}_{k,i}^{\text{ref}} \right)^2 \right], \tag{5.1}$$

with the normalized state and reference vector

$$\widetilde{x}_{k,i} = \frac{x_{k,i}}{x_i^{\lim}} \text{ and } \widetilde{x}_{k,i}^{\text{ref}} = \frac{x_{k,i}^{\text{ref}}}{x_i^{\lim}},$$

the time index k, the state dimension n, the episode length K and the state limit $\boldsymbol{x}^{\lim}$. To assess the ability to respect the state-space constraints, the number of constraint violations cumulated during training are observed. For this, the cumulated constraint violations (CCV) are defined as:

$$\text{CCV} = \sum_{e=1}^{E} c_e(\boldsymbol{x}), \quad \text{with } c_e(\boldsymbol{x}) = \begin{cases} 0 & \text{, if } \boldsymbol{x} \in \boldsymbol{\mathcal{X}} \\ 1 & \text{, otherwise} \end{cases}, \tag{5.2}$$

with the number of episodes E.

5.1.2. The Double Integrator

The developed controller is evaluated on the so-called double integrator. This is a popular control task, often used as a toy example to test control algorithms. The goal is to control the system states to an equilibrium or reference point. This is often referred to as set-point-control. In this work, the system is considered as LTI and is described by the differential equations:

$$\begin{aligned} \ddot{\boldsymbol{x}}(t) &= \boldsymbol{u}(t), \\ \mathbf{y}(t) &= \boldsymbol{x}(t), \\ \text{s.t.} \quad \boldsymbol{x}(t) \in \boldsymbol{\mathcal{X}}, \boldsymbol{u}(t) &\in \boldsymbol{\mathcal{U}}, \;\; \boldsymbol{x}(0) = \boldsymbol{x}_0, \end{aligned} \tag{5.3}$$

and can be put into the state-space representation:

$$\begin{aligned} \dot{\boldsymbol{x}}(t) &= \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \boldsymbol{x}(t) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \boldsymbol{u}(t), \\ \mathbf{y}(t) &= \begin{bmatrix} 1 & 0 \end{bmatrix} \boldsymbol{x}(t), \\ \text{s.t.} \quad & \boldsymbol{x}(t) \in \boldsymbol{\mathcal{X}}, \boldsymbol{u}(t) \in \boldsymbol{\mathcal{U}}, \;\; \boldsymbol{x}(0) = \boldsymbol{x}_0 \,. \end{aligned} \tag{5.4}$$

The continuous system is than discretized according to section A.2.

5.1.3. Reward Design

This work implements two different reward functions. The first reward function is a slightly modified version of the weighted sum of errors (WSE) presented by [Tra+19] and is formulated as the scaled WSE (SWSE):

$$r_{k+1}^{\text{WSE}} = \begin{cases} -\dfrac{1}{f} \left(\sum_n w_n \left| \dfrac{x_{k,n} - x_{k,n}^{\text{ref}}}{2 x_n^{\text{lim}}} \right| \right) & , \; \boldsymbol{x}_{k+1} \in \boldsymbol{\mathcal{X}} \\ r^{\text{violation}} & , \; \boldsymbol{x}_{k+1} \notin \boldsymbol{\mathcal{X}} \end{cases}, \tag{5.5}$$

where n is the state dimension, w_n is a weighting coefficient for each state, x_n^{ref} is the reference state and f is a constant scaling coefficient. The scaling coefficient f is introduced to scale the reward in such a way that the estimated state-action value q does not encounter unreasonably large values during training, which mitigates the risk of numeric problems during the gradient calculation of the critic. The reward r_{k+1}^{WSE} lies in the range of:

$$-\frac{\sum_n w_n}{f} \leq r_{k+1}^{\text{WSE}} \leq 0.$$

If the agent violates system constraints the reward takes the constant penalty $r^{\text{violation}}$. As an alternative reward function for control tasks [SW21] proposes the normalized reward:

$$r_{k+1}^{\text{norm}} = \begin{cases} 2\,(1-\gamma)\left(1 - \sum_n \left|\frac{x_{k,n} - x_{k,n}^{\text{ref}}}{2x_n^{\text{lim}}}\right|\right) - (1-\gamma) & , \; \boldsymbol{x}_{k+1} \in \boldsymbol{\mathcal{X}} \\ -\sum_n \frac{x_{k,n}}{x_n^{\text{lim}}} & , \; \boldsymbol{x}_{k+1} \notin \boldsymbol{\mathcal{X}} \end{cases} \tag{5.6}$$

with the discount factor γ. The reward r_{k+1}^{norm} takes values in the range of:

$$-(1-\gamma) \leq r_{k+1}^{\text{norm}} \leq 1-\gamma.$$

The reward function normalizes the state-action value such that:

$$\begin{aligned} q_{\max} &= \sum_{i=k}^{\infty} \gamma^{i-k} r_{\max} = \frac{r_{\max}}{1-\gamma} \overset{!}{=} 1 \\ \Leftrightarrow r_{\max} &= 1-\gamma \end{aligned}$$

and therefore limits the state-action value q to avoid numerical difficulties during gradient calculations, as well as the reward function mentioned above.

5.1.4. Setup

The developed system and all training and testing routines are implemented in Python. The actor and critic approximator networks are implemented with the PyTorch framework [Pas+19] with a simple feed-forward architecture with three hidden layers for each network. The nonlinear Leaky-ReLU:

$$\text{LeakyReLU}(x) = \max(\beta x, x),$$

is used as the activation function with $\beta = 0.01$ describing the negative slope. The critic and the actor network both use linear output layers with the actor output bounded to $[\boldsymbol{u}_{\min}, \boldsymbol{u}_{\max}]$. The network weights are optimized using the ADAM optimizer [KB14]. The hyperparameters for the evaluation are chosen as listed in Table 5.1 and are determined testing different configurations and executing a random parameter search for the parameters a search space is listed. The chosen setup aimed to find a basic working standalone DDPG controller and are not specifically optimized for the final setup of the MASC to highlight the basic working principles of the developed agent.

Table 5.1: Hyperparameter configuration used for the evaluated controller setup.

symbol	description	search space	used parameter
$\lvert\mathcal{M}\rvert$	replay-buffer size		20000
$\lvert\mathcal{S}\rvert$	batch size		64
$\sigma_{\nu,\text{start}}$	exploration noise standard deviation start value		0.2
$\sigma_{\nu,\text{end}}$	exploration noise standard deviation end value	[0.01, 0.05]	0.02
d_{rate}	noise decay rate	[0.986, 0.996]	0.9947
d_{steps}	noise decay steps		1000
τ	soft target network update		0.005
γ	discount factor		0.99
α	NN learning rate	[0.0001, 0.001]	0.0002
	number of hidden layers	{3, 4, 5}	3
h	neurons per hidden layer	{16, 32, 64, 128}	$h_1 = h_3 = 16, h_2 = 32$
K	maximal episode length		1000
λ	RLS forgetting factor		0.9
f	reward quotient		1500
$\boldsymbol{w}$	reward weights		$w_1 = 2, w_2 = 1$
$r^{\text{violation}}$	violation penalty		-2.0
r^{SG}	safeguard penalty		-0.0025
T_s	environment sampling time		0.05

The RLS system identification is implemented with the openly available Python package Padasip [Cej17]. The solvers used for the constrained optimization regarding the safeguard or MPC are taken from [Car21] and [Vir+20]. The double integrator environment is structured and implemented after the example of [Bro+16]. The implementations of this work can be found in the accompanying git repository [Pet22].

5.2. Results

The following chapter highlights and explains the observations made, as well as the results found during the training and testing procedure. Different configurations of the developed system are evaluated with focus on the introduced metrics.

5.2.1. Controller Performance

The goal of this study is to develop a RL controller that learns to control the double integrator to the equilibrium $\boldsymbol{x}_{\text{eq}} = \begin{bmatrix} 0 & 0 \end{bmatrix}^{\text{T}}$ while respecting the system's constraints. The first thing to look at is the overall control performance of different controller configurations during training. The examination of different module combinations allows different aspects of the system to be studied. A distinction is made between the standalone RL controller, the RL controller with safeguard using a prior available system model and the full MASC, presented in section 4.1.

Displayed in Figure 5.1 is the learning curve of the standard DDPG controller. After a certain number of warm-up episodes to store observations in the replay-buffer, the agent shows an increased cumulated reward after half of the learning process is completed. After 750 training episodes, the controller does not converge to a clear optimum and cannot provide a reliable convergence behavior as shown in the variations of the depicted confidence interval.

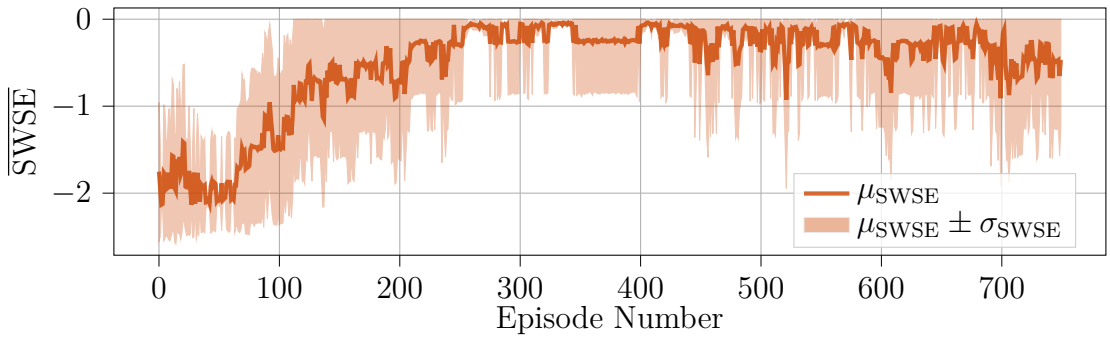

Figure 5.1: Cumulated SWSE per episode averaged over ten different seeds during training. Displayed are the averaged metric μ_{SWSE} and the confidence interval of one standard deviation σ_{SWSE}.

Combining the DDPG controller with the safeguard leads to the results displayed in Figure 5.2. The plots show the learning curves for the safeguard initialized with and without a prior available model.

Both controllers show similar convergence behavior. Because the safeguard prevents large penalties caused by constraint violations, the visible range of the SWSE is more closely concentrated near the optimum, than for the DDPG controller. While the prior available model enables the safeguard to converge quickly, as displayed in the left plot of Figure 5.2, the right plot shows a far lower reward in the first 200 episodes until the estimated model and therefore the calculated feasible-set $\boldsymbol{\mathcal{G}}$ is accurate enough to form the safeguard constraints and correctly guards the controller against unsafe actions.

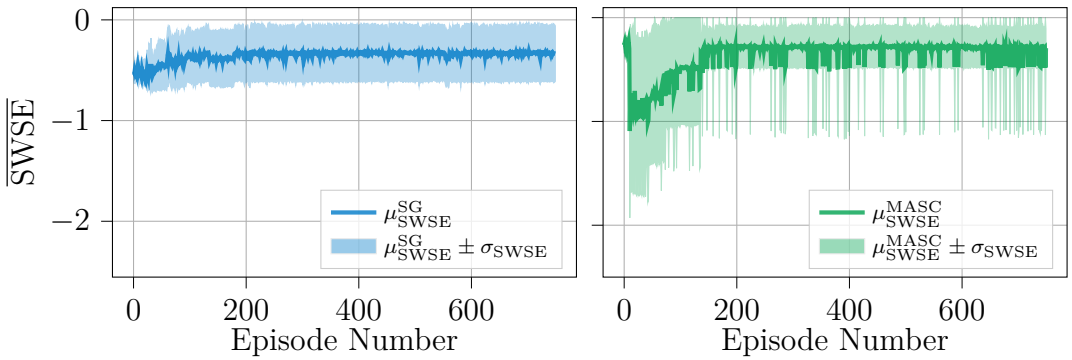

Figure 5.2: Cumulated SWSE per episode during training averaged over ten seeds with a confidence interval of one standard deviation σ_{SWSE} for the safeguard (SG) controller using a prior model (left) and the safeguard controller using an estimated model (right).

Due to the safeguard, no constraints are violated, and the state trajectory is guided along the boundaries of the feasible-set, as illustrated in the representative example in Figure 5.3 which displays trajectories at the start and end of training. No violation penalty is issued to enforce the RL agent to learn to solve the control task and learn to control the agent towards the equilibrium. This leads in average to a very unreliable control performance that in some cases does not improve during training and shows a lower reward than the standalone DDPG controller, even when the agent converges.

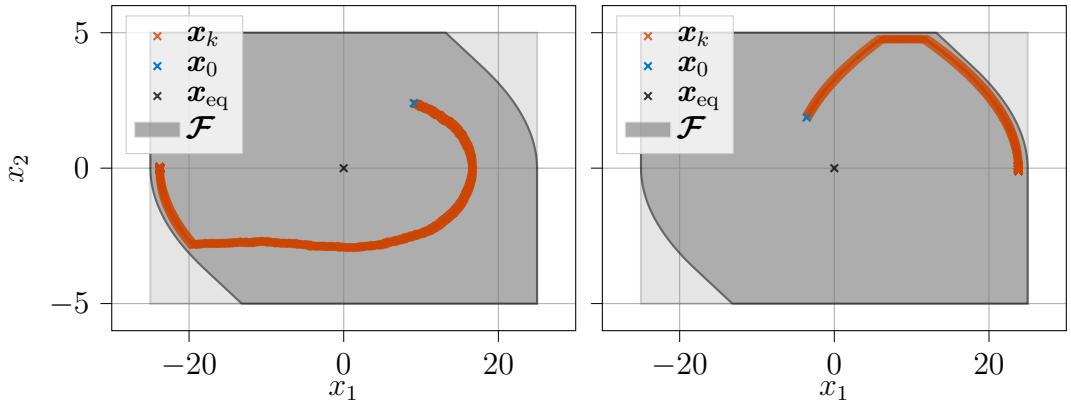

Figure 5.3: Exemplary state trajectories within the state-space. A trajectory from the first episode (left) and last episode (right) of training are displayed.

By penalizing the actuation of the safeguard, the learning curves (Figure 5.4) show improvements in both cases. After a possible worse reward at the beginning of training when the safeguard often intervenes, the agent learns to steer the system to the equilibrium and avoids the use of the safeguard to avoid further penalties. This leads to a higher average reward when converged and solves the problem that the safe controller is often get stuck at the boundaries of the feasibility constraints.

The learning behavior when training the same controller setups with the normalized reward-function (5.6) is analogous to the results presented above and can be looked up in section A.3.

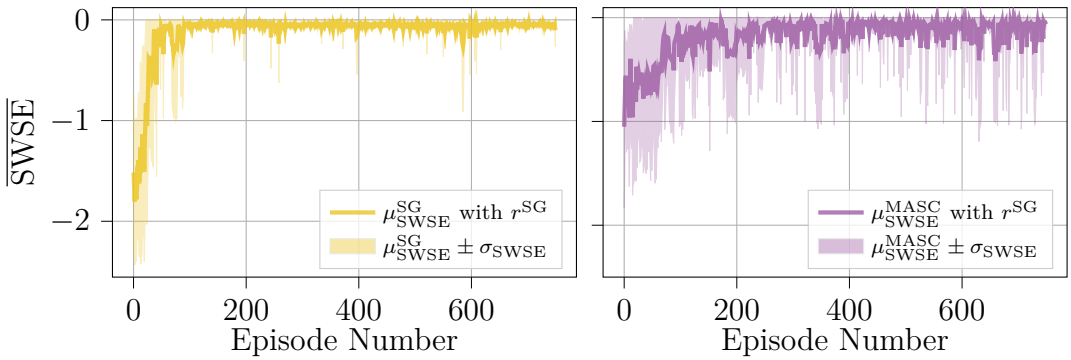

Figure 5.4: Cumulated SWSE per episode during training averaged over ten seeds with a confidence interval of one standard deviation σ_{SWSE} for the safeguard (SG) controller that is penalized with r^{SG} if actuated. Depicted is the SG using a prior model (left) and the SG controller using an estimated model (right).

Concluding the control performance evaluation, Figure 5.5 shows the direct comparison of the CMSE per episode for the default DDPG controller and the MASC with safeguard penalty trained with each reward function.

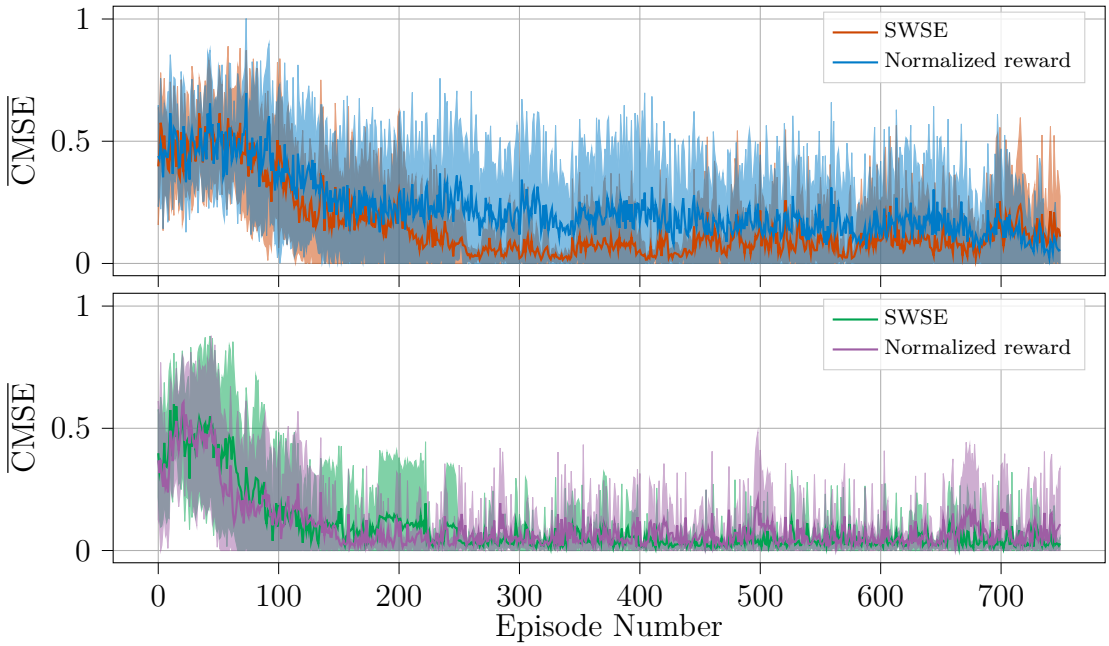

Figure 5.5: CMSE per episode for the standard DDPG controller and the penalized MASC trained with the SWSE and the normalized reward-function. The depicted results are averaged over ten different seeds and displayed with a confidence interval of one standard deviation σ_{CMSE}.

The normalized reward can reach a slightly lower error when used to train the standalone DDPG controller, than the SWSE reward function. The training of the MASC shows hardly any difference regarding CMSE or reliability of the training. Hence, there is no preferable reward-function to be seen when trained with the same RL controller setup.

5.2.2. Constraint Satisfaction

Although the standard DDPG can solve the double integrator control task at least as well as the MASC, the main goal of this thesis is to guarantee constraint

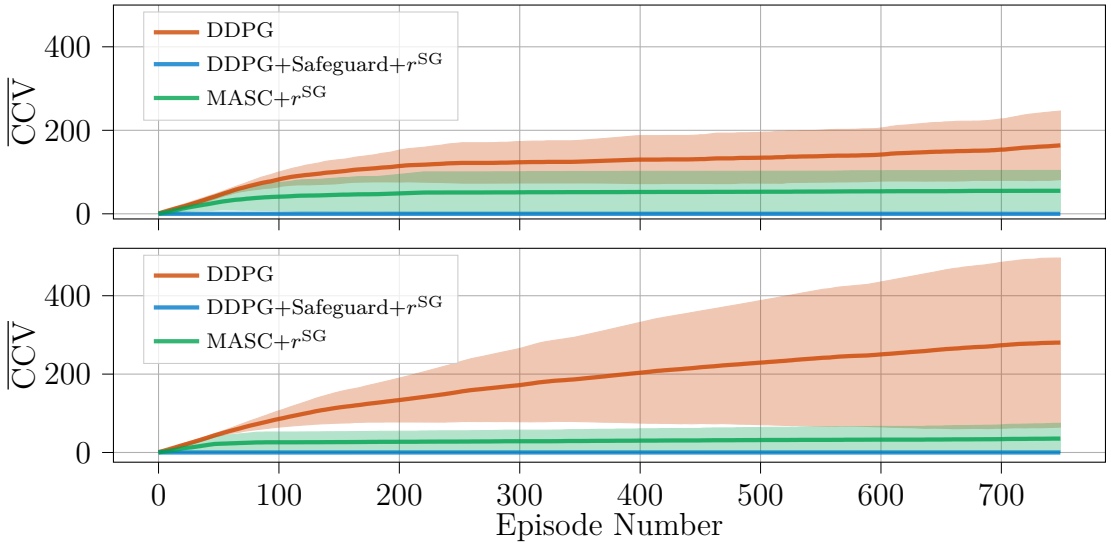

Figure 5.6: CCV per episode during training for all tested controller setups averaged over ten different seeds with a confidence interval of one standard deviation σ_{CCV}. Displayed are the learning process when the controller is trained with the SWSE reward-function (upper plot) and when the controller is trained with the normalized reward-function (lower plot).

satisfaction at all times. This cannot be achieved unless the safeguard is employed, as displayed in Figure 5.6.

Training the controller with different reward function leads to similar constraint respecting behavior. The DDPG controller violates constraints during the whole training, indicated by the steady increasing number of constraint violations. The safeguard controller shows in every case no violations at all. The MASC only violates constraints during the episodes where the identified model is not accurate enough, as shown by the increasing number of violations at the beginning of training. When an accurate model is identified, the guard prevents the controller from violating any constraints and the constraint violations do not increase.

5.2.3. Concluding Comparison

To conclude the evaluation, the metrics listed in Table 5.2 show the test scores of all presented controller configurations after trained for 750 episodes, as well

as the scores for an MPC controller with a prediction horizon $N = 10$. The MPC controller is implemented after the description in section 3.2 and forms the standard approach in optimal control. All results are averaged over ten unique test seeds.

When fully trained, none of the tested controller configurations violates any constraints. Although the scores indicate that all agents are able to steer the system to the equilibrium, the standalone DDPG controller achieves the lowest error during training. The MASC uses no prior model knowledge and produces full constraint satisfaction during training and testing with a slightly higher cumulated MSE than the standard DDPG controller or the MPC controller in testing.

Exemplary trajectories for the controllers with the best test scores are shown in Figure 5.7 and illustrate the ability of the agents to steer the states to the equilibrium $\boldsymbol{x}_{\text{eq}}$ with an negligible rest error. Minor differences in the control trajectories show the control process of the depicted controllers. Controllers using a safeguard tend to follow a slightly wider trajectory and higher control actions as the MPC and DDPG approach, with no risk of violating any constraints. This often helps the controller to get enough momentum to steer the system faster to the equilibrium.

Table 5.2: Test scores for all controller configurations averaged over ten different seeds. Highlighted are the best results for the respective controller configuration. Listed are the averaged scores μ and standard deviation σ

		CMSE		Cumulated reward	
		μ_{MSE}	σ_{MSE}	μ_{rew}	σ_{rew}
SWSE	DDPG	0.0217	0.0203	-0.0370	0.0084
	DDGP+Safeguard	**0.0229**	0.0212	-0.0352	0.0081
	MASC	**0.0259**	0.0257	-0.0377	0.0095
	MASC+r^{SG}	**0.0233**	0.0218	-0.0713	0.0438
Normalized reward	DDPG	**0.0210**	0.0199	9.2026	0.1712
	DDPG+Safeguard	0.0242	0.0224	9.1885	0.1710
	MASC	0.0278	0.0243	8.8298	0.2817
	MASC+r^{SG}	0.0271	0.0263	8.9793	0.2163
	MPC	0.0223	0.0203	-	-

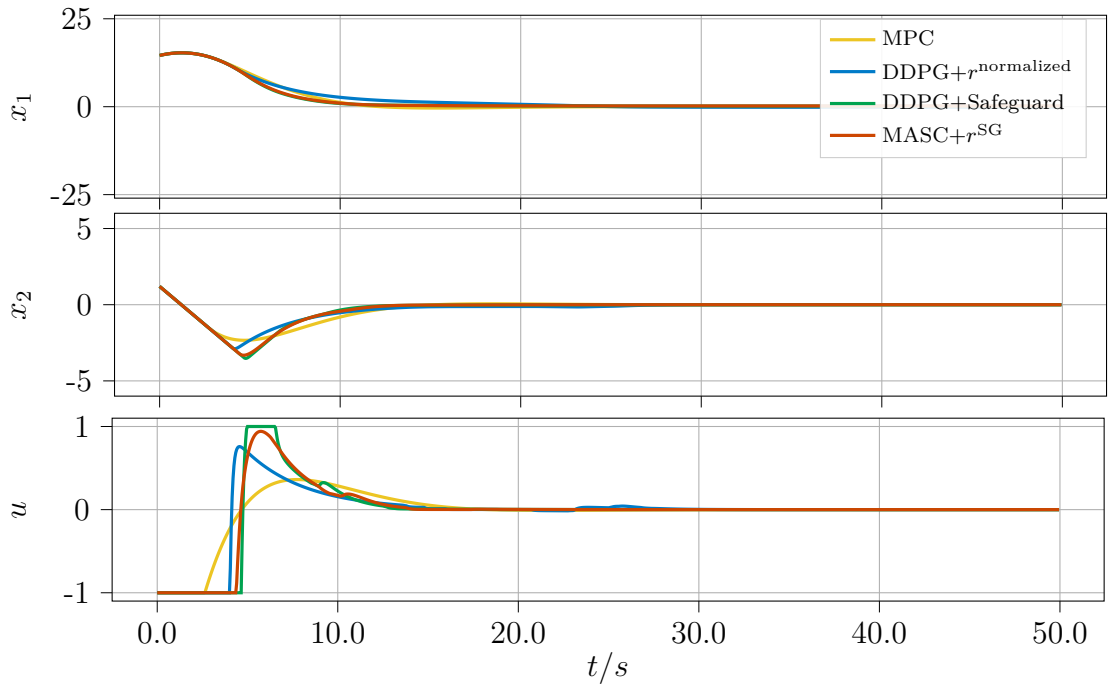

Figure 5.7: Examplary control trajectories for different controller configurations. Displayed are trajectories for the controllers with the best scores for the respective configuration

6. Conclusion

This section summarizes the thesis and draws final conclusions including future tasks and improvements and outlines possible solutions.

6.1. Summary

This work develops the MASC as a safe, adaptive RL controller. Employing state-of-the-art actor-critic algorithms and constrained optimization techniques, the presented system modifies the control output of the actor in such a way that the state-action constraints are not violated, and the controller guarantees safe operation.

After presenting the necessary rudiments in chapter 3, chapter 4 builds the modularly structured MASC. The different parts of the controller are described and explained before they are integrated in the complete system. The RL agent and RLS algorithm used enable the controller to learn a control policy as well as a dynamic model of the environment without any prior knowledge. The developed safeguard tackles the task of safety in RL by modifying unsafe action if the RL controller cannot retain safe behavior on its own. Furthermore, the integration of the different modules is described to form Algorithm 3 describing the operating procedure of the MASC.

To present the results and findings of this work, the necessary metrics are introduced, followed by the listing of the hyperparameters used to evaluate the designed system. It could be determined that when adding the so-called safeguard to a standard DDPG controller, it is possible to enforce constraint satisfaction during the learning process by constraining the control action to the feasible set of the system. Constraint satisfaction is guaranteed by using constraint optimization techniques without relying on heuristically designing a reward function to encourage safe behavior and no need to predict and examine state trajectories. Integrating the RLS

algorithm to identify the dynamics of the controlled environment enables the controller to determine the feasible-set without relying on any prior model knowledge.

In the presented control example, the RL controller can profit from the infinite prediction horizon when solving the Bellman equation and can outperform the MPC controller with a limited horizon. Nevertheless, the constrained learning process with the safeguard employed can lead to unwanted learning behavior where the controller gets stuck at the state boundaries of the feasible set. This problem is tackled by introducing a safeguard-penalty to encourage the agent to not rely too much on the action modification of the safeguard.

In the end, it has been shown that the trained controller configurations are able to achieve similar performance across safe and standard controllers, but when it comes to constraint violation and convergence reliability the MASC can achieve the promising control performance of an RL agent by maintaining the safety properties of an MPC controller during training and standard operation.

6.2. Future Work

The system developed in this thesis can be extended by certain functionalities. The feasible-set calculation can be improved by following publications like [Löf12] to utilize fast and stable calculations of polyhedral constraints. Speeding up the feasible-set calculation lowers the training time of the controller and mitigates the risk of unreasonable large matrices when using computational geometry, as listed in section 3.2. This enables the controller to operate on more complex environments that work with larger state and action spaces.

Alternatives to normally distributed DDPG exploration noise can increase the general performance of the controller and make it more versatile to apply on more dynamic environments with different operation points. It is possible to reuse the estimated system model to execute a controlled exploration in an MPC fashion. Desired subsets of the state-space or operating points to explore can be determined using the risk criteria and curiosity metrics from the field of guided exploration presented in [GF15]. The improved exploration can be an alternative to the safeguard-penalty to prevent the controller from getting stuck at constraint limits or certain regions of the state-space during training.

A simpler approach to improve the exploration behavior is to use exploration noise

adapted to the environment's dynamics, such as an OrnsteinUhlenbeck process. Having less dynamic noise let the system react fast enough to the applied actions and enables the agent to explore different states, because the exploration noise quickly switches between different actions and does not let the environment change operating points. The improved exploration can help to close the performance gap to the standalone DDPG controller.

A common approach to increase the performance of machine learning algorithms is the so called hyperparameter optimization. Because this work focused on the evaluation of a basic setup control agent, a simple parametersearch concerning the MASC and the reward function can increase the control performance.

Bibliography

[Ach+17] Joshua Achiam et al. “Constrained Policy Optimization”. In: (May 2017).

[BBM17] Francesco Borrelli, Alberto Bemporad, and Manfred Morari. *Predictive Control for Linear and Hybrid Systems*. Cambridge University Press, Nov. 2017. 446 pp. ISBN: 1107016886.

[Bel10] Richard E. Bellman. *Dynamic Programming*. PRINCETON UNIV PR, July 2010. 392 pp. ISBN: 0691146683.

[Ber+17] Felix Berkenkamp et al. “Safe Model-based Reinforcement Learning with Stability Guarantees”. In: (May 2017).

[Bro+16] Greg Brockman et al. “OpenAI Gym”. In: (June 2016).

[Car21] Stéphane Caron. “Quadratic programming solvers in Python with a unified API”. In: (2021).

[Cej17] Matous Cejnek. “Padasip - Python Adaptive Signal Processing Toolbox”. In: (2017). version: 1.1.1. URL: `https://matousc89.github.io/padasip/index.html#`.

[Dal+18] Gal Dalal et al. “Safe Exploration in Continuous Action Spaces”. In: (Jan. 2018).

[FHM18] Scott Fujimoto, Herke van Hoof, and David Meger. “Addressing Function Approximation Error in Actor-Critic Methods”. In: (Feb. 2018).

[GF14] Javier Garcia and Fernando Fernandez. “Safe Exploration of State and Action Spaces in Reinforcement Learning”. In: *Journal Of Artificial Intelligence Research, Volume 45, pages 515-564, 2012* (Feb. 2014).

[GF15] Javier Garca and Fernando Fernández. “A Comprehensive Survey on Safe Reinforcement Learning”. In: *Journal of Machine Learning Research 16* (2015).

[Gu+16] Shixiang Gu et al. “Continuous Deep Q-Learning with Model-based Acceleration”. In: (Mar. 2016).

[GZB20] Sebastien Gros, Mario Zanon, and Alberto Bemporad. "Safe Reinforcement Learning via Projection on a Safe Set: How to Achieve Optimality?" In: (Apr. 2020). arXiv: `2004.00915 [eess.SY]`.

[Haa+18] Tuomas Haarnoja et al. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor". In: (Jan. 2018). arXiv: `1801.01290 [cs.LG]`.

[JOP+01] Eric Jones, Travis Oliphant, Pearu Peterson, et al. "SciPy: Open source scientific tools for Python". In: (2001). URL: `http://www.scipy.org/`.

[Jum+21] John Jumper et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596.7873 (July 2021), pp. 583–589.

[Kar+20] Napat Karnchanachari et al. "Practical Reinforcement Learning For MPC: Learning from sparse objectives in under an hour on a real robot". In: (Mar. 2020).

[KB14] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (Dec. 2014).

[KL20] Benjamin Karg and Sergio Lucia. "Stability and feasibility of neural network-based controllers via output range analysis". In: (Apr. 2020).

[Kol+19] Torsten Koller et al. "Learning-based Model Predictive Control for Safe Exploration and Reinforcement Learning". In: (June 2019).

[Lil+15] Timothy P. Lillicrap et al. "Continuous control with deep reinforcement learning". In: (Sept. 2015).

[Löf12] Johan Löfberg. "Oops! I cannot do it again: Testing for recursive feasibility in MPC". In: *Automatica* 48.3 (Mar. 2012), pp. 550–555.

[Lub+20] Joseph Lubars et al. "Combining Reinforcement Learning with Model Predictive Control for On-Ramp Merging". In: (Nov. 2020).

[LW21] Xin-Yang Liu and Jian-Xun Wang. "Physics-informed Dyna-Style Model-Based Deep Reinforcement Learning for Dynamic Control". In: (July 2021).

[MLG20] Andreas B. Martinsen, Anastasios M. Lekkas, and Sébastien Gros. "Combining system identification with reinforcement learning-based MPC". In: 53.2 (2020), pp. 8130–8135.

[Mni+15] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (Feb. 2015), pp. 529–533.

[Mor+21] Andrew S. Morgan et al. "Model Predictive Actor-Critic: Accelerating Robot Skill Acquisition with Deep Reinforcement Learning". In: (Mar. 2021).

[Pas+19] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: (Dec. 2019).

[Pet22] Pascal Peters. “Implementation of the Model Adaptive Safeguard Controller”. In: (2022). URL: `https://github.com/GitPascalP/masc`.

[RHW86] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (Oct. 1986), pp. 533–536.

[Rol11] Marco Münchhof Rolf Isermann. *Identification of Dynamical Systems*. Springer-Verlag GmbH, June 2011. ISBN: 3540788786.

[Sch+17] John Schulman et al. “Proximal Policy Optimization Algorithms”. In: (July 2017).

[Sil+14] David Silver et al. “Deterministic Policy Gradient Algorithms”. In: *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*. 2014, pp. I387I–395.

[Sil+16] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529.7587 (Jan. 2016), pp. 484–489.

[Sut18] Richard Sutton. *Reinforcement learning : an introduction*. Cambridge, Massachusetts London, England: The MIT Press, 2018. ISBN: 9780262039246.

[Sut91] Richard S. Sutton. “Dyna, an integrated architecture for learning, planning, and reacting”. In: 2.4 (July 1991), pp. 160–163.

[SW21] Maximilian Schenke and Oliver Wallscheid. “A Deep Q-Learning Direct Torque Controller for Permanent Magnet Synchronous Motors”. In: *IEEE Open Journal of the Industrial Electronics Society* 2 (2021), pp. 388–400.

[Tra+19] Arne Traue et al. “Towards a Reinforcement Learning Environment Toolbox for Intelligent Electric Motor Control”. In: (Oct. 2019).

[Vir+20] Pauli Virtanen et al. “SciPy 1.0: fundamental algorithms for scientific computing in Python”. In: *Nature Methods* 17.3 (Feb. 2020), pp. 261–272.

[WZ18] Kim P. Wabersich and Melanie N. Zeilinger. “A predictive safety filter for learning-based control of constrained nonlinear dynamical systems”. In: (Dec. 2018).

A. Appendix

A.1. Derivation Variance State-Prediction

Starting with the linear model formulation (3.28) and the variance of the state prediction $\boldsymbol{\psi}_k$ leads to:

$$\begin{aligned}\operatorname{var}(\boldsymbol{\psi}_k) &= \mathbb{E}\left[\boldsymbol{\psi}_k^2\right] - \mathbb{E}\left[\boldsymbol{\psi}_k\right]^2 \\ &= \mathbb{E}\left[\left(\boldsymbol{x}_k\hat{\boldsymbol{\theta}}_k\right)^2\right] - \mathbb{E}\left[\boldsymbol{x}_k\hat{\boldsymbol{\theta}}_k\right]^2\end{aligned}$$

Multiplying the resulting terms results in:

$$\begin{aligned}\mathbb{E}\left[\left(\boldsymbol{x}_k\hat{\boldsymbol{\theta}}_k\right)^2\right] - \mathbb{E}\left[\boldsymbol{x}_k\hat{\boldsymbol{\theta}}_k\right]^2 &= \mathbb{E}\left[\boldsymbol{x}_k^{\mathrm{T}}\hat{\boldsymbol{\theta}}_k\left(\boldsymbol{x}_k^{\mathrm{T}}\hat{\boldsymbol{\theta}}_k\right)^{\mathrm{T}}\right] - \mathbb{E}\left[\boldsymbol{x}_k^{\mathrm{T}}\hat{\boldsymbol{\theta}}_k\right]\mathbb{E}\left[\boldsymbol{x}_k^{\mathrm{T}}\hat{\boldsymbol{\theta}}_k\right]^{\mathrm{T}} \\ &= \boldsymbol{x}_k^{\mathrm{T}}\,\mathbb{E}\left[\hat{\boldsymbol{\theta}}_k\hat{\boldsymbol{\theta}}_k^{\mathrm{T}}\right]\boldsymbol{x}_k - \mathbb{E}\left[\boldsymbol{x}_k^{\mathrm{T}}\hat{\boldsymbol{\theta}}_k\right]\mathbb{E}\left[\boldsymbol{x}_k^{\mathrm{T}}\hat{\boldsymbol{\theta}}_k\right]^{\mathrm{T}} \\ &= \boldsymbol{x}_k^{\mathrm{T}}\,\mathbb{E}\left[\hat{\boldsymbol{\theta}}_k\hat{\boldsymbol{\theta}}_k^{\mathrm{T}}\right]\boldsymbol{x}_k - \boldsymbol{x}_k^{\mathrm{T}}\,\mathbb{E}\left[\hat{\boldsymbol{\theta}}_k\right]\mathbb{E}\left[\hat{\boldsymbol{\theta}}_k^{\mathrm{T}}\right]\boldsymbol{x}_k \\ &= \boldsymbol{x}_k^{\mathrm{T}}\left(\mathbb{E}\left[\hat{\boldsymbol{\theta}}_k\hat{\boldsymbol{\theta}}_k^{\mathrm{T}}\right] - \mathbb{E}\left[\hat{\boldsymbol{\theta}}_k\right]\mathbb{E}\left[\hat{\boldsymbol{\theta}}_k^{\mathrm{T}}\right]\right)\boldsymbol{x}_k\end{aligned}$$

With the known variance formulation the variance from the start the parameter vector $\boldsymbol{\theta}_k$ is calculated as:

$$\operatorname{var}(\boldsymbol{\theta}_k) = \mathbb{E}\left[\hat{\boldsymbol{\theta}}_k\hat{\boldsymbol{\theta}}_k^{\mathrm{T}}\right] - \mathbb{E}\left[\hat{\boldsymbol{\theta}}_k\right]\mathbb{E}\left[\hat{\boldsymbol{\theta}}_k^{\mathrm{T}}\right]$$

which is nothing else than the covarianz matrix of the RLS estimator $\boldsymbol{P}_k$. This leads to the final term used to calculate the prediction variance of the RLS estimator:

$$\mathrm{var}(\boldsymbol{\psi}_k) = \boldsymbol{\Xi}_k^{\mathrm{T}} \boldsymbol{P}_k \boldsymbol{\Xi}_k \tag{A.1}$$

A.2. State-Space Discretization

In order to discretize the continuous state space matrices $\boldsymbol{A}_c, \boldsymbol{B}_c, \boldsymbol{C}_c, \boldsymbol{D}_c$, the equations for the exact discretization ([Rol11]) are given as:

$$\boldsymbol{A}_d = e^{\boldsymbol{A}_c T_s}, \boldsymbol{B}_d = \boldsymbol{A}_c^{-1} \left(\boldsymbol{A}_d - \boldsymbol{I} \right), \boldsymbol{C}_d = \boldsymbol{C}_c, \boldsymbol{D}_d = \boldsymbol{D}_c \tag{A.2}$$

and can be applied on the continuous state space matrices.

A.3. Additional Plots

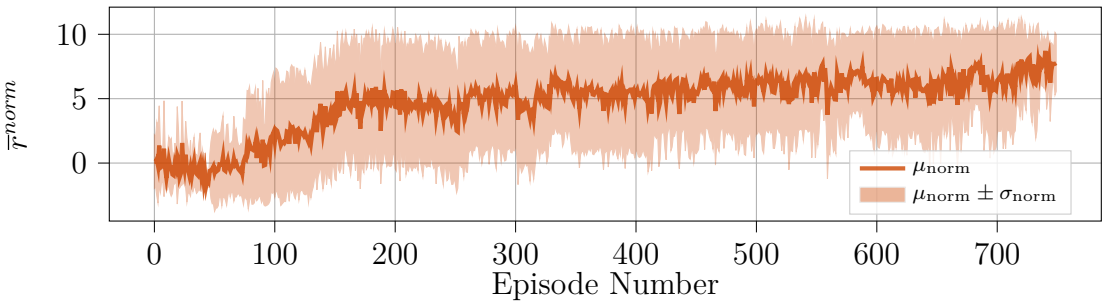

Figure A.1: Cumulated normalized reward for the standalone DDPG controller averaged over ten different seeds, displayed with a confidence interval of one standard deviation.

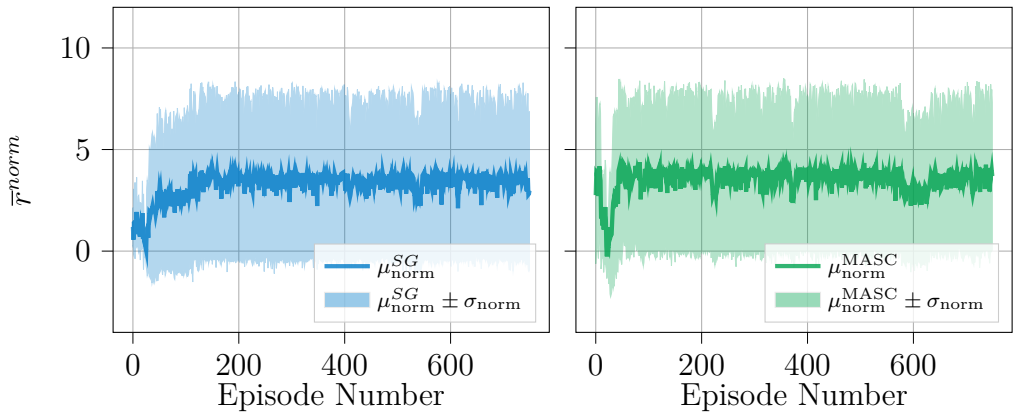

Figure A.2: Cumulated normalized reward for the standalone DDPG controller with additional safeguard using a prior available model (left) and the MASC controller (right). The results are averaged over ten different seeds and displayed with confidence interval of one standard deviation.

B. Statutory declaration

I hereby affirm that I, Pascal Peters, have authored this thesis independently, that I have not used other than the declared sources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.This thesis has not been submitted or published either in whole or part, for a degree at this or any other university or institution. In case of justified suspicion, the thesis in digital form can be examined with the help of "Turnitin". For the comparison of my work with existing sources I agree to storage in the institutional repository.In any case, the examination and evaluation of my work has to be carried out individually and independently from the results of the plagiarism detection service.Further rights of reproduction and usage, however, are not granted here.

Hiermit versichere ich, Pascal Peters, die vorliegende Arbeit selbstständig und unter ausschlieSSlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben. Alle Stellen, die wörtlich oder sinngemäSS veröffentlichtem oder unveröffentlichtem Schrifttum entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Im dringenden Verdachtsfall kann meine Arbeit unter Zuhilfenahme des Dienstes "Turnitin"geprüft werden. Dafür erlaube ich die Ablage meiner Arbeit im institutsinternen Speicher. Unabhängig vom Ergebnis der Prüfung durch "Turnitin" wird immer eine individuelle Prüfung und Bewertung der Arbeit vorgenommen. Darüber hinaus wird der Inhalt der Arbeit Dritten nicht ohne meine ausdrückliche Genehmigung zugänglich gemacht.

Paderborn, April 20, 2022

Pascal Peters