Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Chemical Engineering

Frida Bjørnstad Konow

Deep Reinforcement Learning in Real-Time Optimization

Challenges and Potential

Master's thesis in Chemical Engineering and Biotechnology
Supervisor: Johannes Jäschke (NTNU), Bhushan Gopaluni (UBC)
Co-supervisor: Philip Loewen (UBC)

June 2022

NTNU
Norwegian University of Science and Technology

THE UNIVERSITY OF BRITISH COLUMBIA

Frida Bjørnstad Konow

Deep Reinforcement Learning in Real-Time Optimization

Challenges and Potential

Master's thesis in Chemical Engineering and Biotechnology
Supervisor: Johannes Jäschke (NTNU), Bhushan Gopaluni (UBC)
Co-supervisor: Philip Loewen (UBC)
June 2022

Norwegian University of Science and Technology
Faculty of Natural Sciences
Department of Chemical Engineering

Preface

This thesis is written as the final project for the Chemical Engineering and Biotechnology Master in the course TKP4900 - "Chemical Process Technology, Master's Thesis" at the Norwegian University of Science and Technology, NTNU. The project idea was provided by the Data Analytics and Intelligent Systems, DAIS, Lab at the University of British Columbia, UBC. The work was conducted in Vancouver, Canada, in cooperation with the Process System Group at the Department of Chemical Engineering at NTNU. The thesis introduces reinforcement learning from a process control perspective, assuming no prior knowledge. Further, it investigates how to apply model-free reinforcement learning in static real-time optimization. A project thesis for the course TKP4580 - "Chemical Process Technology, Specialization project" conducted by the undersigned are used as preparatory work for the sections related to modifier adaptation and presentation of the case study.

First, I would like to thank my supervisors at UBC, Bhushan Gopaluni and Philip Loewen, for allowing me to write my master's thesis on exchange. With their guidance and hospitality, I was able to fulfill my desire for a study abroad experience as a part of master's. Further, I would like to thank my NTNU supervisor, Johannes Jäschke, for making it possible and for valuable assistance throughout the project. I would also like to thank the DAIS Lab group for welcoming me and providing new insight and discussions at the weekly meetings. I especially appreciate Nathan Lawrence's generous and indispensable assistance on reinforcement learning related programming. Finally, I would like to thank my friends and family for their support and motivation.

Abstract

Reinforcement learning (RL) is a machine learning field attracting attention for its ability to solve complex problems. The fundamental idea is learning through trial-and-error, where the problem is formulated as a Markov decision process (MDP). It can be seen as an optimization tool, where the best decisions are chosen to fulfill a long-term goal. Deep RL is RL in combination with deep learning, for which high dimensional and continuous problems can be solved using RL. Model-free RL algorithms do not require any process model. A challenge is that they suffer from computational issues and sample inefficiency.

Real-time optimization (RTO) ensures that process plant operation is continuously optimized to the economic optimum by solving a steady-state optimization problem. Developing an accurate process model to use in RTO can be challenging in chemical process plants. Therefore, the use of RL in RTO can eliminate the need for a process model. However, the steady-state RTO problem differs from a typical RL problem. Formulating it as an MDP is essential to enable the use of RL as an optimization method in RTO.

This project contributes to a detailed description and discussion on how to formulate a steady-state optimization problem as an MDP from first principles, assuming minimal prior knowledge in RL. Further, it shows how to utilize the model-free RL algorithm deep deterministic policy gradient (DDPG) to solve the problem with the Python RL tool Stable Baselines 3. A version of RTO called modifier adaptation (MA) is introduced as an example of a conventional RTO alternative. It is used as a base case for comparison with the RL-RTO, and both schemes are applied to the Williams-Otto reactor as a case study for implementation. The results prove that the steady-state optimization problem can be solved using RL. However, RL-RTO faces challenges with sample efficiency and constraint violation that must be addressed for real-life implementation. Finally, measures to overcome the challenges and a discussion of RL-RTO's potential as a subject for further research are presented.

Sammendrag

Forsterkningslæring er et område innen maskinlæring som tiltrekker seg oppmerksomhet grunnet egenskaper til å løse komplekse problemer. Læring gjennom å prøve-og-feile er den grunnleggende idéen, der problemet formuleres som en "Markov decision process", MDP. Det kan betraktes som en optimaliseringsmetode, der valg tas for å oppnå et langsiktig mål. Ved å kombinere forsterkningslæring med dyp læring, kan problemer med flere dimensjoner løses. Forsterkningslæring krever ingen prosessmodell. Utfordninger med forsterkningslæring er diverse beregningsproblemer og effektiv optimalisering ved trening på små datasett.

"Real-time optimization" (RTO) sørger for at et prosessanlegg kontinuerlig optimaliseres til et økonomisk optimum ved å løse et statisk optimaliseringsproblem. Å utlede nøyaktige prosessmodeller for RTO kan være utfordrende. Bruk av forsterkningslæring kan eliminere behovet for en slik prosessmodell i RTO. Et statisk RTO problem avviker fra et typisk forsterkningslæringsproblem. Derfor er det essensielt å formulere det statiske RTO problemet som en MDP for å kunne bruke forsterkningslæring i RTO.

Dette prosjektet bidrar til en detaljert beskrivelse og diskusjon om hvordan et statisk optimaliseringsproblem kan formuleres som en MDP fra grunnleggende prinsipper og ingen forhåndskunnskap innenfor forsterkningslæring. Videre vises det hvordan den modellfrie forsterkningsalgoritmen "deep deterministic policy gradient" (DDPG) kan benyttes for å løse problemet ved hjelp av et Python-verktøy kalt Stable Baselines 3. "Modifier adaptation" (MA) introduseres som en konvensjonell RTO-metode, og benyttes som et basiseksempel for sammenligning med forsterkinigslæring-RTO. Begge løsningsmetodene anvendes på Williams-Otto-reaktoren som casestudie for implementering. Resultatene viser at det statiske optimaliseringsproblemet kan løses ved hjelp av forsterkningslæring. Videre viser resultatene forsterkningslæring-RTO står overfor utfordringer med prøveeffektivitet og konvergering til løsninger som bryter med systemets begrensninger. Disse utfordringene må løses for å kunne implementere metoden i praksis. Til slutt presenteres tiltak for å løse utfordringene, samt en drøfting av forsterkinigslæring-RTOs potensiale som emne for videre forskning.

Table of Contents

Preface i

Abstract ii

Sammendrag iii

1 Introduction 1

2 Background and Theory 3

2.1 Process Control . . . 3

2.2 Real-Time Optimization . . . 5

2.2.1 Building Blocks . . . 5

2.2.2 Implementation . . . 6

2.2.3 Challenges . . . 7

2.3 Modifier Adaptation . . . 7

2.4 Reinforcement Learning . . . 9

2.4.1 Reinforcement Learning Framework . . . 10

2.4.2 Bellman Equations . . . 13

2.4.3 Deep Q-network . . . 14

2.4.4 Deterministic Policy Gradient . . . 16

2.4.5 Deep Deterministic Policy Gradient . . . 18

2.4.6 Hyperparameter Tuning . . . 20

2.4.7 Model-Based Reinforcement Learning . . . 21

2.4.8 Challenges in Reinforcement Learning . . . 22

2.5 Reinforcement Learning for Real-Time Optimization . . . 24

3 Problem Formulation 25

3.1 Steady-State Optimization Problem . . . 25

3.2 Modifier Adaptation . . . 26

3.3 MDP Formulation for Steady-State Optimization . . . 28

3.3.1 Dynamic Penalty . . . 30

4 Case Study: Williams-Otto Reactor 31

4.1 Reactor Description . . . 31
4.2 Plant and Model Equations . . . 33
4.3 Optimization Problem . . . 34
4.3.1 MDP formulation . . . 35

5 Implementation **36**
5.1 Programming Environment . . . 36
5.2 Reinforcement Learning Tools . . . 36
5.3 Hyperparameter Tuning . . . 37
5.4 Case study . . . 37

6 Results and Discussion **39**
6.1 Parameter Tuning . . . 39
6.1.1 Constraint Penalty Parameters . . . 39
6.1.2 Hyperparameters . . . 42
6.1.3 Total Number of Timesteps . . . 43
6.2 Case Study . . . 44
6.2.1 Standard Modifier Adaptation . . . 44
6.2.2 RL-RTO: DDPG . . . 46
6.2.3 Constraint Violations . . . 47
6.2.4 Step Change on Prices . . . 48
6.3 Challenges and Potential . . . 51

7 Improvements and Further Work **53**
7.1 Sample Efficiency . . . 53
7.1.1 Modified State Formulations . . . 53
7.1.2 Dynamic Penalty . . . 56
7.2 Other Reinforcement Learning Algorithms . . . 57
7.2.1 Batch-Online Reinforcement Learning . . . 57
7.2.2 Model-Based Reinforcement Learning . . . 58
7.3 Final Discussion . . . 59

8 Conclusion **61**

A Gradient Approximation - Finite Differences **68**

B Algorithms **69**
B.1 Standard Modifier Adaptation . . . 69
B.2 Deep Deterministic Policy Gradient . . . 70

C Open AI Gym Environment: Williams-Otto Reactor **71**

D Stable Baselines 3: DDPG for Williams-Otto reactor **74**

E Other Model-Free Algorithms **75**

E.1 TD3 75
E.2 SAC 76

List of Figures

2.1.1 Process control hierarchy divided into numbered levels with belonging time scales. Figure recreated from Figure 19.1 in [1]. 4
2.2.1 Block diagram for static RTO [2]. 7
2.3.1 Block diagram of the modifier adaptation scheme. 8
2.4.1 RL components and interaction. Figure reconstructed from figure in [3] 10
2.4.2 Backup diagram illustrating decomposition of the value functions. Figure based on figure in [4]. 14
2.4.3 Schematic illustration of the actor-critic architecture in RL. 17
2.4.4 Model-based RL structure. Figure reconstructed from figure in [3]. 21
2.5.1 Block diagram of the proposed model-free RL-RTO. 24

3.2.1 Visualization of the modified constraint function $G_{m,i,k}$ with the modifiers $\epsilon_{i,k}$, λ_k^{Φ} and $\boldsymbol{\lambda}_k^{G_i}$. Figure based on Figure 1 in [5] and Figure 1 in [6]. 27

4.1.1 Illustration of the Williams-Otto reactor with feed streams F_A and F_B, outlet stream F and the plant reactions. 32

6.1.1 The economic reward as a function of the action space for the Williams-Otto reactor: A) The unconstrained function; B) The function with uniform penalties, and the global plant optimum indicated with the red dot. 40
6.1.2 Reward function with tuned constraint penalty parameters and the global plant optimum indicated as a red point. 41
6.1.3 Moving average of training episode reward plotted as a function of number of timesteps with corresponding standard deviations illustrated with the shaded areas, for tuned and untuned hyperparameters in the DDPG algorithm. 43
6.1.4 Average percent deviation from the global plant optimum in A) F_B and B) T_R plotted as a function of total number of training timesteps. The shaded areas are the standard deviation, the dots represent the settings that were simulated and the line shows the trend. 44
6.2.1 Plot of input iterations for standard MA. Each star represents one iteration and the dotted line shows the order of the iterations. The final iteration is labelled u_k^f. The constraint boundaries $g_1 = 0$ and $g_2 = 0$ are the red lines. 45

6.2.2 Plot of input iterations for RL-RTO using DDPG. The shaded orange area represents standard deviation in the iterations. 47
6.2.3 Moving average of training episode reward plotted as a function of number of timesteps, with associated standard deviations shaded, when a step change on the prices occurs at timestep 1 000. 49
6.2.4 Plot of point of convergence in the inputs with associated standard deviation for RL-RTO using DDPG trained before and after the step change on prices. True plant optimum before and after price change are displayed as u_1^* and u_2^* respectively. 50

7.1.1 Average percent deviation from the global plant optimum in A) F_B and B) T_R plotted as a function of total training number of training timesteps for three different state formulations; full, minimal and extended. 55

E.1.1 Plot of $[F_B, T_R]$ and standard deviations for iterations of the simulated RL-RTO using TD3. 76
E.2.1 Plot of $[F_B, T_R]$ and standard deviations for iterations of the simulated RL-RTO using SAC. 77

List of Tables

4.2.1 Kinetic parameters for the plant and model reactions from [7] 1. 34
4.3.1 Prices for the products P and E, and the feed reactants A and B for 2 scenarios [8]. 35

6.1.1 Default and tuned DDPG hyperparameters for the Williams-Otto reactor. . . . 42
6.2.1 The optimal values of the inputs, profit, and constrained states from simulation of the standard MA. 45
6.2.2 The optimal values of inputs, profit, reward and constrained states from simulation of the RL-RTO using DDPG. 46
6.2.3 Summary of agent training result for 100 simulations of RL-RTO for varying number of T. 48

7.1.1 Summary of agent training result for 100 simulations of RL-RTO for varying number of total timesteps using dynamic penalties. 56

E.1.1 The optimal values of inputs, profit, reward and constrained states from simulation of the RL-RTO using TD3. 75
E.2.1 The optimal values of inputs, profit, reward and constrained states from simulation of the RL-RTO using SAC. 76

Nomenclature

List of Abbreviations

Abbreviation	Description
RL	Reinforcement learning
RTO	Real-time optimization
RTO-RL	Reinforcement learning implemented real-time optimization
MA	Modifier adaptation
MDP	Markov decision process
DDPG	Deep deterministic policy gradient
NCO	Necessary conditions of optimality
PID	Proportional–integral–derivative
MPC	Model predicitive control
DQN	Deep Q-learning
DPG	Deterministic policy gradient
NLP	Nonlinear programming
TD	Temporal difference
SGD	Stochastic gradient descent
SGA	Stochastic gradient ascent
MBRL	Model-based reinforcement learning
LP	Linear penalty
DP	Dynamic penalty
API	Application programming interface
TD3	Twin-delayed DDPG
SAC	Soft actor-critic
GP	Gaussian process

List of Symbols

Symbol	Description	Unit
Theory		
Φ	Cost function	\$/s
x	Process state	-
u	Input	-
d	Disturbance	-
F	Feed flow rate	kg/s
P	Product flow rate	kg/s
Q	Energy usage	energy/s
p_F	Feed price	\$/kg
p_F	Product price	\$/kg
p_Q	Energy price	\$/energy
$\hat{d}$	Estimated disturbances/parameters	-
y	Measurement/output	-
k	Current iteration	-
u_k^*	Optimal input calculated from RTO	-
t	Time step	s
o_t	Observation	-
r_t	Scalar reward signal at time t	-
s_t	RL state at time t	-
a_t	Action at time t	-
E	Environment	-
S	Set of RL states	-
A	Set of actions	-
P	State transition probability	-
R	Reward function	-
R_t	Return	-
γ	Discount rate	-
μ	Deterministic policy	-
π	Stochastic policy	-
V	Value function	-
Q	Action-value function	-
V^*	Optimal value function	-
Q^*	Optimal action-value function	-
π^*	Optimal stochastic policy	-
β	Stochastic behaviour policy	-
θ	Parameter from function approximator	-
$\hat{Q}$	Approximated action-value function	-
z_t	TD target	-
L	Action-value function loss	-
ρ	Discounted state visitation for β	-
ϵ	Exploration factor	-
α	Learning rate	-
D	Replay buffer	-
J	Policy performance objective	-
$\nabla_\theta J$	Policy gradient	-
$\hat{J}$	Approximated policy performance objective	-
θ'	Target parameters	-
σ_{a_t}	Action noise	-
μ'	Exploration policy	-
τ	Target weight	-

Symbol	Description	Unit
Problem formulation		
Φ_p	Plant evaluated cost function	\$/s
Φ	Model evaluated cost function	\$/s
$\mathbf{y}_p$	Measured outputs	-
$\mathbf{y}$	Estimated outputs	-
$G_{p,i}$	Plant evaluated constraints	-
G_i	Model evaluated constraints	-
n_g	Number of constraints	-
n_u	Number of inputs	-
n_y	Number of outputs	-
U	Upper and lower input bounds	-
$\mathbf{u}^{\mathrm{L}}$	Lower input bound	-
$\mathbf{u}^{\mathrm{U}}$	Upper input bound	-
$\mathbf{F}$	Process model	-
$\mathcal{A}$	Set of active constraints	-
$\mathbf{u}^*$	Optimal input	-
k	Current iteration	-
$\mathbf{u}_k$	Current operating point	-
$\Phi_{m,k}$	Modified cost function at operating point	-
$G_{m,i,k}$	Modified constraint function i at operating point	-
$\epsilon_{i,k}$	Zero order modifier	-
$\boldsymbol{\lambda}_k^{\Phi}$	First order modifier for cost function	-
$\boldsymbol{\lambda}_k^{G_i}$	First order modifier for constraint function	-
$\mathbf{K}$	Input filter matrix	-
k_i	Input filter parameter for input i	-
a_i	Zero order modifier filter parameter for constraint i	-
b_i	First order modifier filter parameter for constraint i	-
c	First order modifier filter parameter for cost function	-
$[v]^+$	Maximum selector	-
κ_i	Constraint penalty parameter	-
$\kappa_{i,min}$	Minimal constraint penalty parameter	-
$\kappa_{i,max}$	Maximal constraint penalty parameter	-
d_i	Update factor in dynamic penalty	-

Symbol	Description	Unit
Case study		
A	Reactant	-
B	Reactant	-
C	Intermediate product	-
P	Product	-
E	Product	-
G	Byproduct	-
F_A	Feed flow rate of A	kg/s
F_B	Feed flow rate of B	kg/s
F	Total flow rate	kg/s
x_i	Weight fraction of $i \in \{A, B, C, P, E, G\}$	-
k_i	Reaction rate constant for reaction $i \in \{1, 2, 3, 1^*, 2^*\}$	s^{-1}
A_i	Pre-exponential factor for reaction $i \in \{1, 2, 3, 1^*, 2^*\}$	s^{-1}
$E_{a,i}$	Activation energy for reaction $i \in \{1, 2, 3, 1^*, 2^*\}$	K
T_R	Reactor temperature	°C
r_i	Reaction rate for reaction $i \in \{1, 2, 3, 1^*, 2^*\}$	s^{-1}
M_t	Total mass	kg
P_i	Price of $i \in \{A, B, P, E\}$	$
σ	Measurement noise	-

Chapter 1

Introduction

This project examines the use of a model-free deep reinforcement learning algorithm as an optimization method for static real-time optimization in process control. In addition, a debate on the potential and challenges related to real-life implementation is provided.

Reinforcement Learning (RL) is a rapidly evolving field of research within machine learning. RL solves problems using a trial-and-error methodology, where an agent learns through interaction with an environment [4]. It has successfully been applied to problems like playing video games [9], controlling robots [10] and to continuous process controllers like PID or MPC [11] [12]. RL can solve complex problems that conventional computation approaches cannot solve. The RL objective is to achieve a long-term goal, and therefore a typical RL problem is a sequential decision-making problem. To solve a problem using RL, the environment, e.g., a plant, is modelled as a Markov decision process (MDP) [12]. RL algorithms are either model-free or model-based. Model-free algorithms use environment experience exclusively to train the agent, while model-based algorithms learn a model through environment interaction. Function approximation can enable the application of RL to problems with continuous state and action spaces. Deep RL combines deep learning with RL, typically by using neural networks for function approximation [4]. Process control problems have continuous states and actions.

The process control objective is to provide safe and economical operation of a process plant by mandating the process at its desired setpoints [13]. Real-time optimization (RTO) of process systems aims to ensure system operation while meeting quality and safety constraints by optimizing an economic objective function. The optimization requires a process model, which can be derived through fundamental equations or experimental data. Steady-state models are most commonly used in RTO [1]. However, the derivation of an accurate steady-state process model requires expert knowledge, and an exact model of a chemical plant can often be computationally complex. As a result, the optimization in RTO based solely on a model will often not converge due to structural plant-model mismatch. Even with accurate models, optimality might be infeasible if the system is exposed to disturbances that change the optimum. To overcome the challenge of plant-model mismatch, RTO approaches utilizing measurements have evolved.

Modifier adaptation (MA) is a type of RTO that utilize measurements in the optimization with correction terms called modifiers [6]. The modifiers represent the plant-model mismatch for cost and constraint functions. Compared to standard RTO, MA possess the ability to converge to the plant optimum, given certain prerequisites, even with plant-model mismatch. However, the modifiers introduce a new challenge as they require estimation of the plant gradients, and accurate gradient approximation can be costly and sometimes unattainable, especially when the system is exposed to measurement noise [8].

RTO challenges related to the process model make the model-free optimization implementation with RL appealing. In addition, RL has an adaptive property, which implies that changes in the environment can be handled automatically [11]. However, the nature of an MDP problem has some fundamental differences compared to the steady-state RTO problem related to the lack of dynamics. Therefore, this project studies how the steady-state RTO problem can be formulated as an MDP. Then the MDP formulation will be applied to an RTO case study and solved using a model-free deep RL algorithm called Deep deterministic policy gradient (DDPG) using a Python RL tool called Stable Baselines 3 [14]. MA will be applied to the same case study as a base for comparison with RL-RTO. The results will provide a foundation for discussion of the potential and challenges related to real-life implementation of RL-RTO.

The thesis is structured in seven subsequent sections, starting with background and theory in section 2. Here subject matter regarding RTO, MA and RL relevant for the project is presented. Problem formulation is section 3, where the mathematical equations for the steady-state RTO problem are introduced, followed by the modified problem for MA and an associated MDP formulation for use in RL. Further, the Williams-Otto reactor is presented for the case study in section 4, and a section regarding its implementation follows in section 5. This leads up to results and discussion in section 6, where results from the case study are presented. Section 7 contains suggested improvements and further work within the RL-RTO. Finally, the findings and discussion are rounded off in a conclusion section.

Chapter 2

Background and Theory

In this section, relevant theory and background will be introduced. We start by presenting process control and RTO, with its building blocks and implementation, followed by some challenges with RTO. Further, the concept of MA is introduced to serve as an example of a conventional RTO implementation and a base case for comparison with RL-RTO. An introduction to RL will then be presented, including essential concepts like MDP and Bellmann Equations. Further, a description of the DQN and DPG algorithms is presented, on which the DDPG algorithm is based. Finally, the DDPG algorithm and its hyperparameters are presented. In the end, there is a brief introduction to batch training and model-based RL, followed by common challenges in RL.

2.1 Process Control

Control is a widespread engineering field that involves continuous monitoring of a dynamic system [13]. It is utilized in anything from control of a drone in cybernetics to control of an electrical circuit in electrical engineering. Process control is a branch of control that applies to physicochemical systems that are familiar to chemical engineers. Common examples of process control systems are chemical reactors, heat exchangers and mass transfer systems [15]. Throughout the years, process control has evolved from a tool to ensure safety and maintain a process at the desired point of operation to fully controlling automatic industrial process plants. Implementation of process control results in efficient operation and minimizes the number of process operators, which has economical and risk benefits [16].

The objective of process control is to maintain the process at the operational conditions and set points [1]. The set points are decided considering several criteria, such as budget and yield. Some systems also require control to transition the process from one operational condition to another. One example could be the economic set points varying between high and low electricity prices. When the electricity prices are high, it may be beneficial to lower the set point for a reactor's temperature to limit energy usage, even though the reaction may be slower or the yield lower. The calculation of this trade-off is taken care of in process control. In order

to fulfill the process control objectives, process control is divided into different levels of separate time scales that make up a hierarchy.

The process control hierarchy is divided into five levels, as shown in Figure 2.1.1. In different relative time scales of a plant, the hierarchy represents control decisions within optimization, monitoring, and data acquisition [1]. The highest level is called planning and scheduling and includes demand forecasting, supply chain management and scheduling of raw materials and products. A plant manager is usually in charge of this level, which has a large time scale ranging from days to months. The fourth level is real-time optimization (RTO), which includes parameter estimation, supervisory control, and data reconciliation on a time scale ranging from hours to days. Decisions are made with explicit economic objectives. The RTO level can be considered as an online calculation of the best set-points for the lower levels known as supervisory control [1].

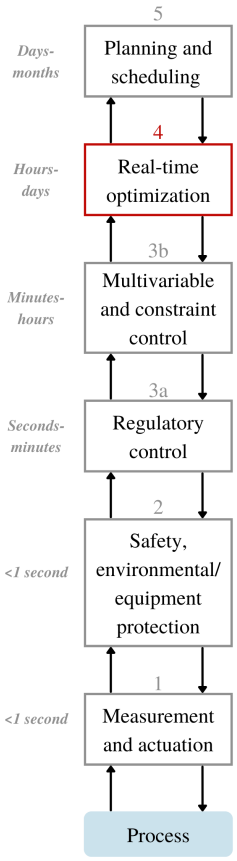


Figure 2.1.1: Process control hierarchy divided into numbered levels with belonging time scales. Figure recreated from Figure 19.1 in [1].

Level 3b, multivariable and constraint control or model predictive control (MPC), permits the plant to function near the constraints. Level 3a uses regulatory control techniques to ensure that variables persist at their respective setpoints in single- or multi-loop control. Level 2 concerns safety, the environment and equipment protection, whilst Level 1 is focused on measurements and actuation. The back and forth arrows between the levels reflect communication

between the successive levels, with the higher level establishing the goal for the lower level. The lower level communicates current operation data to the higher levels, which use it to make decisions. The scope of this project will be solely on level 4 with RTO.

2.2 Real-Time Optimization

RTO optimizes plant economics by maximizing profit and minimizing cost. From around the 1990s, RTO has gained attention when it comes to plant control [1]. The reason is that use of RTO can lead to considerable economic gain since it continuously optimizes the operating conditions based on an economic objective or cost function [6]. If variables in the cost function change frequently, for example, the price of electricity as mentioned earlier, RTO can adjust the optimal operating conditions for high and low electricity prices. RTO can be smoothly built into computers with control systems due to rapid developments in both hardware and software [1]. This enables optimization of processes with over 100 000 variables and constraints.

2.2.1 Building Blocks

The economic model, the process model, and the process constraints are the three fundamental building blocks in RTO [2]. Equation 2.2.1 expresses the steady-state optimization problem, where $\Phi(x, u, d)$ represents the economic model, $f(x, u, d)$ represents the process model, and $g(x, u, d)$ represents the process constraints.

$$\begin{aligned} \min_{u} \quad & \Phi(x, u, d) \\ \text{s.t.} \quad & f(x, u, d) = 0, \\ & g(x, u, d) \leq 0, \end{aligned} \tag{2.2.1}$$

The economic model, also known as the cost function, $\Phi(x, u, d)$, is a function of the states, x, inputs, u, and disturbances, d. States in the context of process control are a set of variables that define the mathematical condition of a dynamic system [15]. Inputs are also known as manipulated variables, which can be adjusted. Disturbances are variables that are given by conditions outside of the control of the plant operators. The overall goal of RTO is to maximize profit, which is cost subtracted from income. The cost usually includes the expenses of the raw materials used in the feed and the cost of utilities, typically energy. The income depends on the value and production of the products. With that assumption and energy as the sole utility, the cost function is formulated as follows:

$$\Phi = \sum p_F F + \sum p_Q Q - \sum p_p P, \tag{2.2.2}$$

where F represent feed flow rate, Q, represents energy usage per time unit, and P is the flow rate of the product [2]. The coefficients p_i are the energy prices and the feed and product flows in [\$/J] and [\$/kg] units, respectively. The shape of an RTO cost function is usually quite smooth around the optimum. This is because one does not want minor deviations from the optimum to have a significant impact on the cost.

The equation $f(x, u, d) = 0$ in Problem 2.2.1 is the process model. This model is typically derived using first principles, such as mass and energy balances or experimental data. The generation of an accurate process model demands expert knowledge, and an exact model of a chemical plant is usually computationally complicated. The model's function is to use the operating conditions for each unit, such as flow variables, temperatures and pressures, to calculate product yield and other variables required in the cost function. In practice obtaining exact models is infeasible due to unknown parameters in the system. Further, the accuracy of a model could often come at the expense of computational complexity. A complex model is harder to solve and optimize. Obtaining a sufficient model that is not too complex is a well-known challenge in process control [1]. The most common process model in RTO is a steady-state model, which often requires less computational capacity in optimization. This is called static RTO. Dynamic RTO (DRTO) is less common [2]. In the case of DRTO, x, u and d are dependent on time and should be denoted x_t, u_t and d_t.

The process constraints, $g(x, u, d)$, are determined by criteria for product purity, plant unit operating conditions, waste amount, and equipment capacity [17]. It is not uncommon that the inequality constraints are active at optimal plant set points, as we do not want to have an unnecessarily pure product. In practice, f are the optimization problem's equality requirements, and g are its inequality constraints.

2.2.2 Implementation

Figure 2.2.1 illustrates a block diagram of a static RTO. The diagram describes the stages required for RTO implementation, starting with steady-state detection and parameter estimation, followed by "Static RTO". RTO can be generalized as a two-step process consisting of 1) steady-state detection and parameter estimation and 2) optimization, which is repeated in iterations.

The process gives the measurements y. Because steady-state models are used in RTO, it is vital to ensure that the process is in a steady state prior to optimization. Consequently, a steady-state detection block is necessary. The block continuously monitors the measurements to determine if the process has achieved steady-state. There are numerous approaches for detecting steady-state, where statistical tests are commonly used [2].

Once the process has reached steady-state, the measurements are passed to the parameter estimation block. Parameter estimation aims to estimate unmeasured disturbances and parameter values d given steady-state process measurements y, inputs u, and a process model. An optimization problem is solved by minimizing the squared error between the measured y and calculated $\hat{y}$ values based on the process model. When this optimization problem is solved, the estimated $\hat{d}$ yields the minimum deviation between system measurements and model under the constraints [1].

Finally, the system is re-optimized in the static RTO block using the process model and the estimated parameters $\hat{d}$. The optimization problem in Equation 2.2.1 is computed, giving new optimal inputs, u_k^*. After set-point control, these inputs can be implemented in the plant as

set-points for the lower levels.

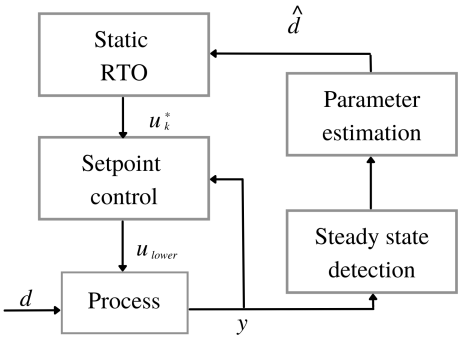


Figure 2.2.1: Block diagram for static RTO [2].

2.2.3 Challenges

The two-step RTO approach requires no structural plant-model mismatch to perform well [5]. Whenever there is a mismatch between the model and the plant, the results are generally that the RTO is not able to satisfy the necessary conditions for optimality, NCO [6]. The reason is that the parameters estimated will not be a sufficient representation of the plant. This results in calculated set-points that are sub-optimal. To overcome this challenge, the use of measurements and identification of plant gradients can be introduced to ensure that NCOs are fulfilled [6]. A group of RTO methods that uses this approach, called fixed-model methods, has evolved. Here the model-parameter update is not necessary [7]. Although the RTO approach should be taken related to the plant's properties, there are in general three fundamental properties highly desired in RTO. Firstly, guaranteed optimality upon convergence is vital to ensure maximum economic gain. The second property is efficient convergence to prevent sub-optimal operation for extended time periods. Lastly, that the point of convergence is feasible is essential to avoid constraint violation [6]. If any of these properties is not fulfilled, it introduces performance issues for the RTO [18].

2.3 Modifier Adaptation

Modifier adaptation (MA) is an RTO approach with a fixed model. All fixed-model methods use a nominal process model in combination with plant measurements in the optimization [7]. A nonlinear programming (NLP) problem with a nominal process model is solved in iterations [5]. The parameters are not estimated at each iteration, unlike the two-step RTO technique. The measurements are instead used to update the cost and constraint functions. MA uses measurements in correction terms known as modifiers, which are calculated prior to optimization at each iteration. When the optimization converges, the modifiers ensure that NCO is met [5]. The modifiers in MA schemes are calculated based on the difference between the plant measurements and the process model. Two modifier orders are established for standard MA: zero and first order. Zero-order modifiers represent direct deviations, while first-order modifiers represent gradient deviations. The mathematical equations for MA are described in Section

3.2. The key benefit of MA is that it is capable of converging to the best solution even in the presence of structural plant-model mismatch [6]. This enables a simpler model that requires fewer computing resources. However, a reasonably accurate model must still be derived.

Figure 2.3.1 shows a block diagram of the standard MA method, corresponding to Figure 2.2.1 for static RTO. A modifier calculation block replaces the parameter estimation block in the block diagram. Consequently, the block for optimization is given the modifiers as inputs rather than the estimated $\hat{d}$. The optimizer calculates the optimal inputs based on the modified optimization problem and the modifiers in the cost and constraint functions. The rest of the system remains the same, with the steady-state detection and setpoint control.

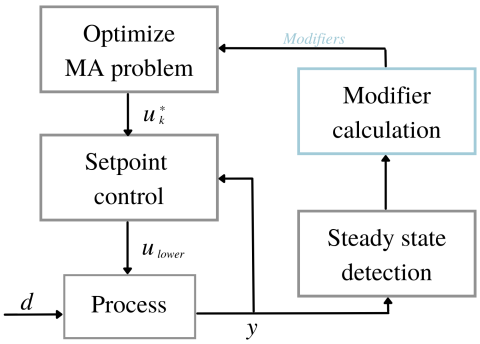


Figure 2.3.1: Block diagram of the modifier adaptation scheme.

Recall the three desired RTO features. Both the second property of rapid convergence and the third property of feasible convergence rely on the plant and process model. These features appear paradoxical, as quick convergence can imply longer steps for each iteration, whereas feasible convergence prefers smaller steps. Given precise plant evaluations and gradients, MA satisfies the first property of converging to the plant optimum. As previously stated, MA has the advantage of meeting the NCO compared to the two-step RTO strategy, even with structural plant model mismatch. However, precise plant evaluations and gradients are not always feasible. A new challenge for MA is the computation of accurate gradients, especially in the presence of measurement noise. Thus, while overcoming the challenge of structural plant-model mismatch, another challenge is introduced as a result. If there was an RTO approach which was not dependent on a model, these challenges could be eliminated.

2.4 Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning, where an agent learns through interaction with an environment [4]. It has proved to perform well in solving various sequential decision-making problems. The key components of the RL framework are an agent, e.g., an optimizer, that interacts with an environment, e.g., a process plant, that is modelled as a Markov decision process (MDP) [12]. Model-free RL algorithms use environment experience exclusively to train the agent.

RL differs from other machine learning fields like supervised learning, where there is no learning from interaction. In other words, the training data that the agent learns from in supervised learning contain both the system's information and the correct solution. In RL, the training data does not contain the solution, and the agent has to learn through interaction with the environment which behaviour is correct for a given system [3]. A practical view is to imagine that the agent in supervised learning is given both a game and the game rules. In contrast, in RL, the agent is only given a game and has to figure out the rules by trial-and-error while playing the game [4].

In recent years, RL has become an area of research that has developed quickly. A few examples of RL's success are in robotics control problems, in a driving simulation called Torcs [10], and in Atari computer games [9], where the RL agent outperforms the best human players in three of six games. However, the use of RL in process control is not extensive yet. There are successful examples of the use of RL within process control, where RL is applied to the lower levels of the decision hierarchy (Figure 2.1.1), in multi-variable and regulatory control [12]. Here model-free dynamic-RL controllers have been proposed to replace standard controllers like PID or MPC [12] [11]. There is one example of the use of RL in RTO [19], where a custom made actor-critic algorithm is used for the optimization of the steady-state problem (Equation 2.2.1). However, this account provides essentially no detailed description of how a process control system can be formulated as an RL environment or how to write the steady-state optimization problem as an MDP. In addition, there are currently no published resources describing how to apply well-known RL algorithms in RTO for more straightforward implementation and comparison. We will refer to RL implemented RTO as RL-RTO.

There are a variety of solution strategies within RL, and new algorithms are developed frequently. When choosing an algorithm for an RL problem, one should generally consider the environment assumptions. One of the barriers to applying RL in process control is that the action and observation spaces are continuous. In order to solve these types of problems with the simplest RL algorithms, discretization is required. While it is possible to discretize the action and observation space, this may lead to an exponential growth in computational complexity, known as the curse of dimensionality [4]. Therefore, another approach for solving these types of problems has evolved, using function approximators (FA) [12].

Within the RL field, a niche called deep reinforcement learning is progressing. Deep RL combines deep learning with RL, typically using neural networks for function approximation in RL. This approach has enabled efficient generalization of RL to problems with continuous state

and action spaces. A thorough introduction to deep learning can be found in the following book[20]. The deep deterministic policy gradient (DDPG) algorithm is one of the first and most used deep RL algorithms that is formulated for environments with large-scale continuous observation and action spaces[10]. DDPG combines vital concepts from the two other RL algorithms—deep Q-learning (DQN) and deterministic policy gradient (DPG)—in an actor-critic architecture[10].

In what follows, an introduction to the RL framework, terminology and key equations will be provided, followed by a brief background on the essential ideas of DQN and DPG. This leads up to the presentation of the DDPG algorithm and its hyperparameters. Finally, model-based RL and challenges in RL will be introduced.

2.4.1 Reinforcement Learning Framework

An RL system consists of several elements. First, two key parts are required to set up an RL problem: an agent and an environment. The agent is the learner and able to make decisions. The agent interacts with the environment, which can be interpreted as everything outside the agent. At each timestep, t, the agent receives an observation, o_t, and a scalar reward, r_t, from the environment and executes an action, $a_t \in \mathbb{R}^N$[10]. The environment then receives the action from the agent and emits a new observation, o_{t+1}, and reward, r_{t+1}. This process is illustrated in Figure 2.4.1. The environment determines valid actions, and the set of all valid actions is referred to as the action space $\mathcal{A}$[21].

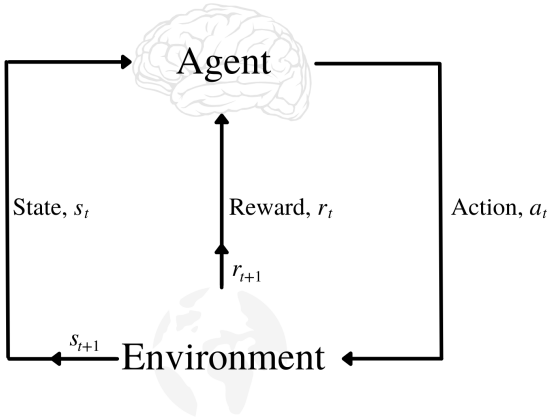


Figure 2.4.1: RL components and interaction. Figure reconstructed from figure in[3]

Further, the concept of state, s_t, has to be introduced. The state space, $\mathcal{S}$, defines the possible states. A state in RL is a function of the history and contains information used to determine what happens next. Formally the true state is the one that characterizes the environment. The best the agent can do is construct its approximation of the true state. The degree of observability determines the accuracy of the agent's approximated state. With partial observability, the agent observes the environment indirectly and must construct its state representation. In this case, the agent state is not equal to the true environment state. On the other hand, the agent can directly observe the true environment when there is full observability. Thus, the

agent state is a direct observation of the true state, which are therefore equal: $s_t = o_t$. Figure 2.4.1 shows an RL system with full observability. This is the fundamental idea for a Markov decision process [3].

A Markov or information state is characterized by the Markov property, which entails that the state contains all information of the history and completely characterizes the process. The Markov can be described as follows:

> *The state must include information about all aspects of the past agent–environment interaction that make a difference for the future [4].*

In other words, once the Markov state of a process is known, the history of the process is also known, and previous states are redundant. Further, a Markov process is a memoryless random process with a sequence of states with the Markov property. Finally, a Markov decision process (MDP) is an environment where all states are Markov. An MDP problem formulation results in the states being dependent on less information and provides a framework for solving RL problems. Almost all RL problems can be formulated as an MDP. Therefore, formulating an MDP is a key first step for applying RL solving strategies to the problem.

Formally, a *finite* MDP is a tuple consisting of five components defined as follows [22]:

Definition 2.4.1 (MDP)**.** A *finite Markov Decision Process* is a tuple $\langle S, A, P, R, \gamma \rangle$, in which

- S is a finite set of states
- A is a finite set of actions
- P is a state transition matrix, $P_{SS'} = \mathbb{P}[S' = s_{t+1} | S = s_t, A = a_t]$
- R is a reward function, $R = \mathbb{E}[r_{t+1} | S = s_t, A = a_t]$
- γ is a discount factor, $\gamma \in [0, 1]$

The state transition probability, P, quantifies the probability of ending in state $S = s'_{t+1}$ given the current Markov a state $S = s_t$. In the case of a finite MDP, the state transition takes the form of a matrix. Since it is a describes the transition from one state at time t to the next at $t + 1$, it decides the dynamics of the system [4]. $R = \mathbb{E}[R_{t+1} | S = s_t, A = a_t]$ is a reward function given current state and action. In what follows, we will present each of the components in the MDP in detail. We will use the notation $s_t \in S$ and $a_t \in A$. Note that Definition 2.4.1 considers *finite* MDP where the action and observations are finite sets, which makes P a matrix and R a computable expectation.

Within the agent, there are three main components; a policy, a reward signal and a value function. In addition, there might be a model of the environment [4]. The policy defines the agent's behaviour by mapping states to actions. The policy determines which "decisions" the agent should take, and it fully defines the agent's behaviour. For MDPs, the policies are only dependent on the current state. In other words, they are stationary [22]. There are two types of policies; deterministic and stochastic. Deterministic policies are denoted μ and result in exact

actions at a certain time as follows [21]

$$a_t = \mu(s_t). \tag{2.4.1}$$

Stochastic policies, π, generate a probability distribution on the set of actions given a state, under which the randomly-selected action can be written

$$a_t \sim \pi(\cdot|s_t). \tag{2.4.2}$$

In deep RL, parameterized policies are applied. For these policies, the outputs are obtained from computing functions that depend on a set of parameters, θ^π or θ^μ [21]. These functions are typically referred to as function approximators (FA) and can be applied to any parameterized variable. The parameters determine certain behaviours of the optimization algorithm. The policies' complexities vary greatly, where some policies are simple functions or lookup tables. Others can require large computations such as searches [4].

At each timestep, the agent receives a reward signal related to the previous action taken. It is a scalar feedback signal, and the goal for the agent is to maximize the total reward for a number of timesteps. The agent sees the reward signal to determine how good one specific action is. It can be seen as an analogy to our perception of pain, which can be defined as a negative reward signal. The reward signal, r_t, is generated from the reward function, R, which depends on the current state and action [21],

$$r_t = R(s_t, a_t). \tag{2.4.3}$$

Return is the cumulative reward from a particular timestep t, and can either have a finite or infinite time horizon. In the case of an infinite time horizon, a discount factor, γ, is introduced. The discount factor is a number between 0 and 1 and implies that immediate reward is valued more than delayed reward [3]. The discounting makes the mathematical calculations more convenient since it avoids infinite returns. In addition, it can capture some uncertainty about the far future. The finite-horizon discounted return, R_t can be expressed as a sum of discounted future reward:

$$R_t = \sum_{i=t}^{T} \gamma^{i-t} r_i. \tag{2.4.4}$$

Note that the return is dependent on the current action, which again depends on the policy. The agent's objective is to maximize cumulative reward over a sequence of actions and states. In order to quantify the expected reward in one state, a value function can be introduced. The expected return as a given state function is known as the value function. For each state, the agent can use the value function to calculate a respective long-term reward in order to choose the action with the highest value. Some actions result in high short-term reward, and they might not lead to the highest total reward with a longer time horizon. This is what the value function takes into account.

Value functions can be state-value based or action-value based. The state-value function $V^\pi(s_t)$ is defined as the expected return from a state S_t and following policy π, expressed as

$$V^\pi(s_t) = \mathbb{E}[R_t|s_t]. \tag{2.4.5}$$

The action-value function $Q^\pi(s_t, a_t)$, or Q-function, is the expected return from state s_t, taking action a_t, and then following policy π:

$$Q^\pi(s_t, a_t) = \mathbb{E}[R_t|s_t, a_t]. \tag{2.4.6}$$

In order to solve an MDP, the optimal value function must be identified. It specifies the policy that maximize the value function. The optimal state-value function $V^*(s)$ and the optimal action-value function $Q^*(s, a)$, are given in Equation 2.4.7 and 2.4.8 respectively.

$$V^*(s_t) = \max_\pi V^\pi(s_t) \tag{2.4.7}$$

$$Q^*(s_t, a_t) = \max_\pi Q^\pi(s_t, a_t) \tag{2.4.8}$$

For any MDP, there exists an optimal policy, π^*, that is better or equal to all other policies [22]. At the optimal policy, the optimal value and action-value functions are obtained. The optimal policy can be detected by maximizing over the optimal action-value function according to Equation 2.4.9 [21]. This implies that we can directly find the optimal policy if the optimal action-value function is known.

$$\pi^*(s_t) = \arg\max_a Q^*(s, a) \tag{2.4.9}$$

Agents can be either value based, policy based, or actor-critic. Value based agents have a value function, but no policy, while policy based are the opposite. Actor-critic agents have both a value function and a policy [3].

2.4.2 Bellman Equations

The Bellman equations are central for solving the value functions. They decompose the value function into two parts; immediate reward and the discounted value of successor states. The idea is that the value of a certain state is reward obtained from the state added to the value of the next state. Thus, the computation of the value function is simplified by splitting the problem into several smaller recursive problems. Figure 2.4.2 shows a visualization called a backup diagram of the Bellman equations. For the state-value function the current reward r_t at s_t is added to the discount factor multiplied with the value of the next state s_{t+1}. In other words, if we are in state s_t and look one step ahead, the value function can be decomposed in recursive terms.

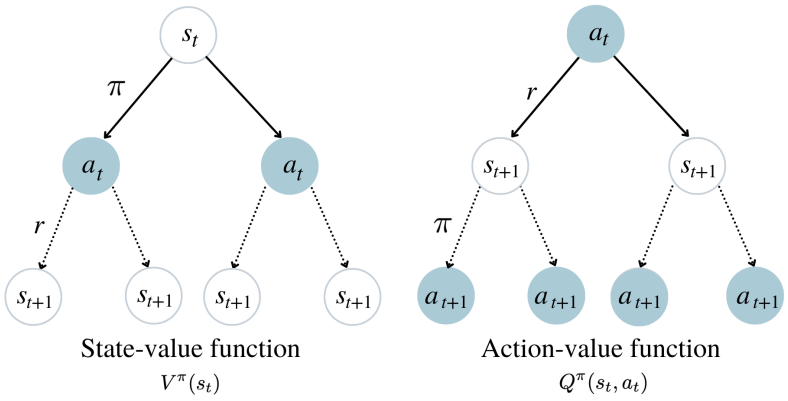


Figure 2.4.2: Backup diagram illustrating decomposition of the value functions. Figure based on figure in [4].

The Bellman Expectation Equation for the state-value function in Equation 2.4.5 is obtained by decomposing it the following way [22]

$$V^\pi(s_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r_t + \gamma V^\pi(s_{t+1}))] \tag{2.4.10}$$

E represents the environment of interest. Equation 2.4.5 can be used to calculate the term for $V^\pi(s_{t+1})$. Similarly, the Bellman equation for the action-value function can be written

$$Q^\pi(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r_t + \gamma \mathbb{E}_{a_{t+1} \sim \pi}[Q^\pi(s_{t+1}, a_{t+1})]], \tag{2.4.11}$$

If the policy is deterministic as described in Equation 2.4.1, Equation 2.4.11 above can be written out as

$$Q^\mu(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E}[r_t + \gamma Q^\mu(s_{t+1}, \mu(s_{t+1})]. \tag{2.4.12}$$

It can be observed that the equation above does not have the inner expectation from 2.4.11 due to the policy being deterministic.

2.4.3 Deep Q-network

The deep Q-Network (DQN) algorithm was developed by Deepmind in 2015 [9]. Since then, it has succeeded in outperforming humans in Atari games. It is based on the basic RL algorithm called Q-learning and deep neural networks combined with experience replay. Q-learning is an off-policy algorithm that uses the action-value function presented in Equation 2.4.12. In Q-learning, a greedy deterministic policy of the form

$$\mu(s_t) = \arg\max_{a_t} Q(s_t, a_t) \tag{2.4.13}$$

is used [23]. The algorithm iteratively solves the Bellman optimality equation and updates Q by following this policy. However, this requires large lookup tables to store all the values for every pair of (s_t, a_t). Therefore DQN introduces a value function approximator, for which we can calculate the value of Q and therefore do not need to store the state-action pairs. The approximator is dependent on some parameters θ^Q and can be denoted $Q(s, a|\theta^Q) \approx Q^*$. To clarify notation, all parameters will be denoted $\theta^{(\cdot)}$, where $(\cdot)$ indicates the approximated function. Similarly, variables with the subscript $(\cdot)_\theta$ indicate that they are parameterized. It can be proved that the use of large neural networks as function approximators for the DQN algorithm is efficient [9]. The objective is then to minimize the loss, L, dependent on the temporal difference (TD) target, z_t, given by the following equations [10]

$$L(\theta^Q) = \mathbb{E}_{s_t \sim \rho^\beta, a_t \sim \beta, r_t \sim E}[(Q(s_t, a_t|\theta^Q) - z_t)^2] \tag{2.4.14}$$

$$z_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\theta^Q) \tag{2.4.15}$$

Here ρ denotes the discounted state visitation distribution for a stochastic behaviour policy β. The stochastic behaviour policy is usually an ϵ-greedy policy used to collect environmental data. The ϵ-greedy policy is a simple exploration policy that tries a random action with probability equal to a small number, ϵ. The remaining actions are chosen according to the greedy Q-policy in Equation 2.4.13. For a deterministic policy μ can be written as follows,

$$\mu(s_t) = \begin{cases} \arg\max_{a_t} Q(s_t, a_t), & \text{with probability } 1 - \epsilon; \\ Random(\mathcal{A}), & \text{with probability } \epsilon. \end{cases} \tag{2.4.16}$$

The loss function can be minimized using a stochastic gradient descent (SGD) algorithm, which adjusts the parameters θ^Q in the steepest negative direction of ∇L. The parameter update can be written [12]

$$\theta^Q_{t+1} \longleftarrow \theta^Q_t - \frac{1}{2}\alpha \nabla L_t(\theta^Q_t), \tag{2.4.17}$$

where α decides the step length and is called the learning rate.

Essentially, SGD, together with TD targets, z_t, and the loss L is the action-value function approximator, since it yields the parameters θ^Q that estimate Q. However, since this function approximator is non-linear, it was not applied to large-scale problems for a long time. The reason is that performance guarantees are impossible and would often lead to instability. However, there is a way to alter Q-learning by introducing large scale neural networks as function approximators. The two key concepts that were implemented were the use of experience replay and a target neural network to calculate z_t [10].

This computation can be done by first taking an action according to ϵ-greedy policy for the

data collection. The transition consisting of $(s_t, a_t, r_{t+1}, s_{t+1})$ will then be stored in a replay memory, D, called a replay buffer. During the training, a random mini-batch of transitions will be sampled from D instead of using only the latest transition. This makes the target network update more stable. Finally, the loss function is minimized using a variant of SGD, which is stable with neural networks [24].

2.4.4 Deterministic Policy Gradient

Deepmind proposed the deterministic policy gradient (DPG) algorithm in 2014 for use in RL problems with continuous action spaces [25]. It uses an actor-critic architecture. As presented in Q-learning, the action-value function is parameterized using a function approximator. DPG uses a parameterized deterministic policy function referred to as an actor, denoted $\mu(s|\theta^\mu)$. In order to identify the optimal parameters, θ^μ, the policy update is performed according to the policy gradient.

There are several examples of policy gradient algorithms in RL, where the common idea is to update the parameters of the policy in the direction of the performance gradient [25]. The performance objective, J, measures the cumulative reward to reveal the quality of a specific stochastic parameterized policy, π_θ. It can be defined as

$$J(\pi_\theta) = \mathbb{E}_{s_t \sim \rho^\pi, a_t \sim \pi_\theta}[R_1]. \tag{2.4.18}$$

R_1 denotes the discounted return for timestep $t = 1$ in Equation 2.4.4. Equation 2.4.18 should be optimized with respect to the policy parameters, which means we want to find θ^π that maximizes $J(\pi_\theta)$. An efficient approach for solving this optimization problem is to use the gradient. The stochastic gradient ascent (SGA) method searches for local maxima of $J(\pi_\theta)$ by ascending the gradient of the policy with respect to parameters, as previously introduced with SGD for minimization in Equation 2.4.17 [26]. The stochastic policy gradient theorem provides a closed-form solution to the policy gradient and can be formulated [25],

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s_t \sim \rho^\pi, a_t \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t) Q^\pi(s_t, a_t)]. \tag{2.4.19}$$

Further Equation 2.4.19 is used to update the policy parameters according to SGA,

$$\theta^\pi_{t+1} \longleftarrow \theta^\pi_t + \frac{1}{2}\alpha \nabla_\theta J(\pi_\theta)\Big|_{\theta^\pi = \theta^\pi_t}, \tag{2.4.20}$$

where α is the learning rate [12].

Applying the policy gradient theorem is very useful in computations, as it only requires computing one expectation. In order to build an RL algorithm using policy gradient ascent, one needs to solve or estimate the action-value function, Q. Possibly the easiest estimation is to use the discounted sample return, r_t^γ. However, this approach results in large variance for $\nabla_\theta J(\pi_\theta)$. One way to overcome this is to use an actor-critic architecture [12].

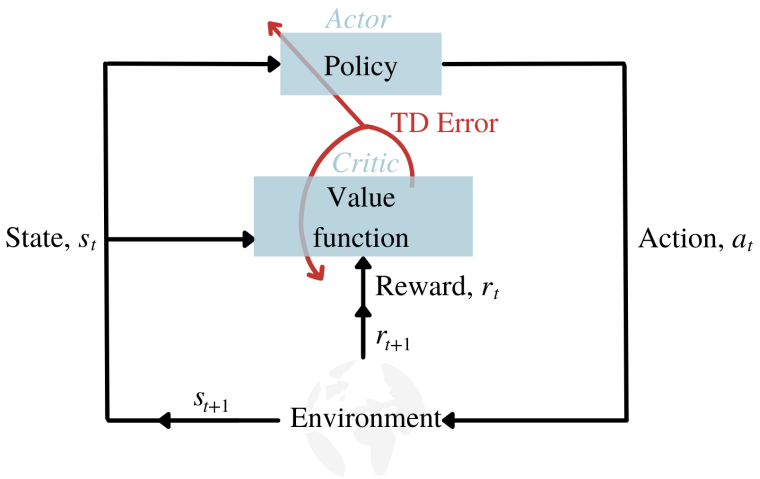


Figure 2.4.3: Schematic illustration of the actor-critic architecture in RL.

Figure 2.4.3 shows a schematic illustration of the actor-critic architecture. The actor-critic architecture consists of two key components. First, the actor proposes a policy, and the critic evaluates the actor's proposed policy by calculating the value. In other words, the actor adjusts the parameters θ^π of the stochastic policy by applying the SGA from Equation 2.4.20. The critic estimates the action-value using a function approximator, optimizing the parameters θ^Q, as in Q-learning. Thus, Q Equation 2.4.19 can be modified to a parameterized version [25]:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s_t \sim \rho^\pi, a_t \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(a_t|s_t) Q(s_t, a_t|\theta^Q)]. \tag{2.4.21}$$

This is the fundamental principle in the actor-critic architecture, and it can be noted that it is a combination of Q-learning and gradient ascent. In contrast with Q-learning, the policy is not learnt through the greedy maximization in Equation 2.4.13, but is updated in the direction of the policy gradient. The result is simpler computation, especially for continuous action spaces, where one avoids the global optimization problem at every timestep [25]. In addition, θ^π is guaranteed to converge to a local optimum when implementing SGA with Equation 2.4.21 [12].

All equations presented so far use a stochastic policy from Equation 2.4.2. When calculating the policy gradient in Equation 2.4.21 for a stochastic policy, integration over the state and action spaces is necessary. The integration requires in theory, an endless number of samples, which is inconvenient in practice. As an alternative, a deterministic policy gradient algorithm has been developed [25], which uses a deterministic policy denoted $\mu(s|\theta^\mu)$. The performance objective in Equation 2.4.18 for a deterministic policy can be defined as [25]

$$J(\mu_\theta) = \mathbb{E}_{s_t \sim \rho^\mu}[R_1(s_t, \mu_\theta(s_t))], \tag{2.4.22}$$

Further, the deterministic policy gradient theorem can be set up analogous to Equation 2.4.19.

$$\nabla_{\theta^\mu} J = \mathbb{E}_{s_t \sim \rho^\mu} \left[\nabla_{\theta^\mu} \mu(s_t|\theta^\mu) \, \nabla_{a_t} Q(s_t, a_t|\theta^Q)|_{a_t=\mu(s_t)} \right]. \tag{2.4.23}$$

DPG enables off-policy implementation. Off-policy algorithms learn the policy independent of the agent's actions [4]. This is done through transitions from a stochastic behaviour policy β [10] and which can advantageous for sufficient exploration. By introducing β, the off-policy DPG can be written as follows

$$\nabla_{\theta^\mu} J = \mathbb{E}_{s_t \sim \rho^\beta} \left[\nabla_{\theta^\mu} \underbrace{\mu(s_t|\theta^\mu)}_{\textit{Actor}} \, \nabla_{a_t} \underbrace{Q(s_t, a_t|\theta^Q)}_{\textit{Critic}}|_{a_t=\mu(s_t)} \right] \tag{2.4.24}$$

where ρ^β is the state distribution under policy β. Implementation of the off-policy DPG algorithm is carried out by using SGA with $\nabla_\theta J$ from Equation 2.4.24 to maximize the performance. The resulting θ^μ are used to update the actor $\mu(s|\theta^\mu)$. The critic, $Q(s, a|\theta^Q)$, where θ^Q are calculated minimizing the loss from SGD in Equation 2.4.17.

2.4.5 Deep Deterministic Policy Gradient

Deep deterministic policy gradient (DDPG) is a model-free, off-policy algorithm used for continuous action spaces that combines DPG with DQN [10]. The approach is similar to Q-learning, where a Q-function finds a policy that maximizes total reward. By assuming that the Q-function is differentiable with respect to action, a gradient-based learning rule for the policy can be formulated. This leads to an efficient solution to the maximization problem [10].

The DQN algorithm has gained attention for its ability to perform at human level on several Atari games. One of its limitations is that it only works with discrete action spaces of low dimensions. Even though continuous action spaces could be discretized and solved with DQN, it would not be efficient due to, among other things, the curse of dimensionality. In this case, it is caused by the ϵ-greedy policy in Equation 2.4.16, which requires maximization at each timestep. This gets expensive when using large non-linear function approximators. In addition, discretization can lead to loss of information to completely describe the action space. Therefore, applying DQN to RL problems with discrete action spaces is not desirable. Instead, an actor-critic alternative from the DPG algorithm is used in DDPG [10].

As presented above in DPG, a parameterized deterministic policy function, or actor, denoted $\mu(s|\theta^\mu)$ is used. The action-value function, or critic, is found from the Bellman equation. In contrast, DDPG uses neural network function approximators to learn the parameters of the action-value function. The use of neural networks as function approximators may cause instability. Therefore experience replay is used as in DQN [10]. The actor and the critic are both updated by sampling a minibatch from the buffer at every timestep. Since the algorithm is off-policy, it is possible to have a large-scale buffer. A large-scale buffer ensures that the data set is uncorrelated [25].

When using neural networks in Q-learning, the $Q(s, a|\theta^Q)$ network is used in calculating the target in Equation 2.4.15. The loss gradient is dependent on the target. That results in the Q-update being dependent on itself and results in a possibility of divergence [12]. To address this, DDPG uses target networks similar to DPG, indicated with $(\cdot)'$. Copies of the actor network, $Q'(s, a|\theta^{Q'})$, and the critic network, $\mu'(s, a|\theta^{\mu'})$ are generated to calculate the targets. The weights of the target are updated using "soft" target updates, which imply that they gradually trace the learned networks as follows

$$\theta' \longleftarrow \tau\theta + (1 - \tau)\theta', \tag{2.4.25}$$

where τ is a small weight, $\tau \ll 1$. The result is that the target values are slowly adjusted, which has proven to increase stability [10]. Although it slows the training process, it ensures convergence of the algorithm. In total, DDPG uses 4 neural networks as function approximators for computation of θ^Q, $\theta^{Q'}$, θ^μ and $\theta^{\mu'}$.

Exploration is essential in RL because it allows the agent to obtain information about the entire action and state space. In continuous spaces, exploration is not straightforward. In DDPG, sufficient exploration of the action space is accomplished by formulating a noisy exploration policy μ'. The noise is introduced adding noise $\mathcal{N}$ to the actor policy as follows [10]

$$\mu'(s_t) = \mu(s_t|\theta^\mu) + \sigma_{a_t}. \tag{2.4.26}$$

Finally a summary of the equations underlying DDPG can be presented. After initializing the critic and actor networks, replay buffer and action noise, an initial state is observed by choosing a random initial action. For each observation an action is executed according to μ_θ, to generate a new state and associated reward. The transition (s_t, a_t, r_t, s_{t+1}) is added to the replay buffer D. Then a minibatch of N transitions is sampled from D. For each sample i the TD target is calculated using the target networks;

$$z_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}) \tag{2.4.27}$$

Then the critic is updated by minimizing the total loss for the minibatch as follows

$$L(\theta^Q) = \sum_i (Q(s_i, a_i|\theta^Q) - z_i)^2 \tag{2.4.28}$$

The actor is updated by maximizing the batch policy gradient,

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_{a_t} Q(s_t, a_t|\theta^Q)|_{s_t=s_i, a_t=\mu(s_i)} \nabla_{\theta^\mu} \mu(s_t|\theta^\mu)|_{s_t=s_i}. \tag{2.4.29}$$

Finally the target networks are updated using soft updates according to Equation 2.4.25:

$$\theta^{Q'} \longleftarrow \tau\theta^Q + (1 - \tau)\theta^Q \tag{2.4.30}$$

$$\theta^{\mu'} \longleftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu} \tag{2.4.31}$$

A full algorithm description of DDPG is presented in Algorithm 2 in Appendix B.

2.4.6 Hyperparameter Tuning

In machine learning context, model parameters are distinguished from hyperparameters. Where model parameters are related to a model, for example, a neural network, hyperparameters are parameters that are external to models. Model parameters can be estimated through experiments, for instance, a regression coefficient. Hyperparameters cannot be determined the same way and demand tuning for a specific problem [27]. Setting the proper hyperparameters can significantly impact the performance and reliability of the deployed solution in RL. The search for the hyperparameter settings that yield the best neural network designs is a time-consuming and computationally expensive operation.

DDPG is known to be sensitive to its hyperparameters, and therefore tuning is especially important. Its many hyperparameters result in fine-tuning being a tedious job [27]. However, many programming frameworks offer default values for all parameters, and one can choose which ones to fine-tune. Here six of the most critical DDPG hyperparameters and their influence are highlighted. The parameters are discount factor, learning rate, target update weight, batch and buffer sizes, the type of action noise and associated noise standard deviation.

First, γ, the discount factor, previously introduced in Definition 2.4.1 and Equation 2.4.4, is a number between 0 and 1 that determines how much future reward should be emphasized. For γ close to 1, the reward at the end of the episode is the goal, whereas, for γ close to 0, one would rather have most reward in the near future independent of the length of the episode. If $\gamma = 0$, the reward function equals the return, as there is no time horizon for the return to be discounted. The second hyperparameter is the learning rate, α, for actor and critic networks. The learning rate sets the step length in the direction of the solution and scales the weight update of the neural networks to minimize the loss. Learning rate was introduced in SGD in Equation 2.4.17 and SGA in Equation 2.4.20. A low learning rate results in slow training due to small changes in the neural network weights. On the other side, too high learning rate can lead to divergence in the loss function [28].

Further, the size of the replay buffer can influence the efficiency and stability of the algorithm. A too small replay buffer will have a lot of correlated data, which can cause an unstable network. However, increasing the size requires more storage and leads to a slower learning process. The smaller the replay buffer is, the closer the algorithm becomes to on-policy, while off-policy algorithms allow for larger size buffers. τ sets the target update from Equation 2.4.26, which is typically close to zero to ensure "soft" updates for improved stability [28]. Finally, the batch size is the number of samples used in the gradient descent update. Larger batch sizes enable computations in parallel due to several nodes to divide the training examples. Thus, the training becomes quicker with larger batch sizes. The drawback is that the model accuracy tends to decrease for large batch sizes. Although they have the same training accuracy as

smaller batch sizes, they have lower test accuracy. The gap between test and training error is known as the "generalization gap". Thus, when deciding batch size, there is a trade-off between training efficiency and accuracy [29].

2.4.7 Model-Based Reinforcement Learning

RL can be model-free or model-based, and all the previously presented algorithms are model-free. A model of the environment can be included to predict the environment's response. The agent is not interacting with the environment itself but using the model to generate a response. Given a state and action, the model's role is to generate a transition to a new state which is as close to the environment's response as possible. In other words, the model is used to simulate the environment [4]. The value function and policy are learnt through experience in model-free RL. In model-based RL, we learn the model from experience and plan the value function and policy from the model.

The advantage of model-based RL is that we can efficiently learn a model by supervised learning methods, like table lookup, linear regression, or Gaussian processes. In addition, the model uncertainty can be estimated. In addition, model-based RL is a lot more sample-efficient. Sample efficiency indicates how much information the agent can learn from one sample and consequently how many training timesteps it needs to perform well. The drawback is that model-based RL consists of several components that must be coded and tuned with care, making the entry bar for research in this field higher. Compared with model-free RL, model-based RL has few general implementation frameworks for non-expert users and fewer published algorithms. As a result, there is an "empirical gap" in performance between model-based and model-free algorithms [30]. As an attempt to reduce this gap, last year Facebook AI Research released a python framework called MBRL for continuous control [30].

An illustration of how a model-based RL algorithm works is displayed in Figure 2.4.4. The figure shows the order of the steps where an action is taken based on a value function or policy. As a result, the environment ends up in a new state, and the response is used as experience in model learning. The grey arrow indicates a model-free algorithm, where the model is bypassed. From the model, planning is used to end up at the starting point with an improved value or policy. *Planning* is a collective term for all computations that produce or improve a policy given a model interacting with an environment [4].

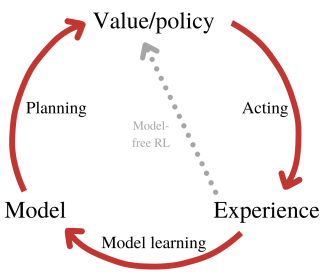


Figure 2.4.4: Model-based RL structure. Figure reconstructed from figure in [3].

2.4.8 Challenges in Reinforcement Learning

Although RL can be a powerful tool to solve complex problems, it can suffer from several issues. One of the main challenges is limited sample efficiency. In other words, RL algorithms require many training timesteps, T, before being able to perform satisfactorily. One example is DeepMind's AlphaGoZero, which played 5 000 000 games of Go before defeating the best human player [9]. Solving pre-defined example RL environments with DDPG has a training timestep scale of 1 000 000 in magnitude [10]. With online algorithms that continue training while simulating, there is usually no upper bound on the total number of training steps; the training time equals the simulation time.

Another challenge in RL is reproducibility. The background provided for DQN, DPG and DDPG previously gave elementary insight into a few primary model-free algorithms. Even with access to code and examples, implementing a new system or modifying existing algorithms can be challenging. Based on that, it is not hard to imagine difficulties with reproducing algorithms with higher complexity. One reason is primarily related to deep learning, which utilizes neural networks. The computation that the neural networks can carry out for large-scale data sets and complex problems that require hours to train is beyond most people's comprehension and contributes to the challenge of reproducibility. This can be an issue for continuous action and observation spaces, as in process control. The algorithms suited for these types of problems are complex, and it requires expert knowledge to enable modifications and tailoring to a specific environment. In addition, there has been a tendency for papers to omit key design details of an algorithm or provide code with low readability [31]. However, several tools have recently been developed to support the robust, consistent, and straightforward implementation of deep RL algorithms. Examples of these tools are Gym AI, SpinningUp and Stable Baselines 3 [32] [31] [14]. These tools contribute to lowering the barrier to entry for deep RL, although the frameworks' lack of agility can be a limitation.

Specifically for the DDPG algorithm, challenges with instability and converging to poor solutions are commonly reported. Instability can be due to over-sensitivity to hyperparameters. Careful tuning of the hyperparameters is therefore crucial for satisfactory performance. Instability can also come from a well-known issue in Q-learning with over-estimation bias when learning a critic [33]. This issue has been resolved in an algorithm called TD3, which builds on DDPG with some modifications. The evolution of RL algorithms often follows this path, where overcoming the limitations of one algorithm by modifying it leads to a new algorithm. However, in general, there is limited knowledge on the source of the errors, and it is unclear if the modifications actually address the source of the problem rather than treating only the symptoms [33]. The result is difficulties revealing the true origin of an RL algorithm's failure.

Finally, a major challenge especially relevant for RL-RTO is the real-life implementation of RL. One of the prerequisites for training an agent is that it learns through trial-and-error. Consequently, failing is an essential part of the learning process. Failing works well in constructed environments like games or simulations, but it is not always an option in real-life scenarios. In a control example like a self-driving car, failing could mean crashing, which obviously cannot

be accepted. Similarly, implementing an untrained RL-RTO could cause significant economic loss and constraint-violating set points, implying unsafe operation. One solution could be to use historical data. However, historical data generally does not provide enough exploration of the environments, as the systems typically are run in a relatively stable setting and not all the way to the extreme cases.

It is evident that RL implementation has significant challenges that cannot be overlooked. However, the successful implementations of obtaining super-human performance on complex problems prove great potential within this field. For RL-RTO, the two main advantages are that it is a model-free implementation and that RL has adaptive properties that can possibly automatically handle disturbances and changes in the environment. Whether these advantages compensate for the potential challenges presented will be discussed in Sections 6 and 7.

2.5 Reinforcement Learning for Real-Time Optimization

Corresponding to the previously presented block diagrams for two-step RTO and MA, an RL-RTO block diagram is presented in Figure 2.5.1. The most significant change is that there is only one RL-RTO block and no parameter estimation or modifier calculation. The reason is that the model-free RL algorithms do not rely on any model equation calculation. The algorithm could, in theory, be any RL algorithm compatible with continuous state and action spaces. For this project, the DDPG algorithm is used. Further, steady-state detection is still required, as the RTO problem is a steady-state optimization problem.

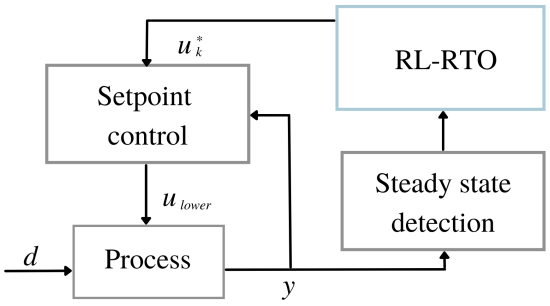


Figure 2.5.1: Block diagram of the proposed model-free RL-RTO.

Model-free reinforcement learning utilizes process plant measurements, y, as experience to learn how to solve the optimization problem. One measurement is required for each step, t, which the agent takes in the environment. Therefore the plant measurements must be fast for implementation of the RL-RTO.

If the measurements are slow, it would require a long time for the agent to gain enough experience to perform well. Therefore, the total number of timesteps, T, needed for the agent to be sufficiently trained is critical in RL-RTO. Simulation with an insufficiently trained agent would result in problems with solving the steady-state optimization problem and lead to sub-optimal set-points. If T is high, it will imply that the plant has to operate at sub-optimal set-points for a long time. This explains why sample efficiency is an especially relevant challenge in RL-RTO.

When it comes to the three desired properties of an RTO, the first, namely, guaranteed optimality upon convergence, should be fulfilled if the agent is appropriately trained. Steady-state optimization is computationally a relatively simple problem. However, DDPG can have the issue of converging to poor solutions as previously introduced. The second property, rapid convergence, is difficult to predict if RL-RTO fulfills, and simulations are needed to investigate. Finally, ensuring that the convergence leads to a feasible point might be an issue as constrained optimization is not seamless in RL. The simulation results of a case study can reveal if RL-RTO possesses these three properties.

Chapter 3

Problem Formulation

This section will first define the mathematical equations related to the steady-state RTO problem. Then the modified optimization problem in MA is presented. Further, the same problem is defined as an MDP that can be solved as an RL problem. So far, all variables presented were one-dimensional. To account for possibility of multidimensional variables, vectors are introduced for u, y and x, which now will be denoted in bold $\mathbf{u}$, $\mathbf{y}$ and $\mathbf{x}$.

3.1 Steady-State Optimization Problem

In Equation 2.2.1 the general RTO optimization problem was presented. The steady-state optimization problem for a plant can be mathematically expressed in Equation 3.1.1, where the subscript $(\cdot)_p$ indicates a quantity related to the plant [8]:

$$\begin{aligned} \min_{\mathbf{u}} \quad & \Phi_p(\mathbf{u}) := \phi(\mathbf{u}, \mathbf{y}_p(\mathbf{u})) \\ \text{s.t.} \quad & G_{p,i}(\mathbf{u}) := g_i(\mathbf{u}, \mathbf{y}_p(\mathbf{u})) \leq 0, \quad i = 1, \ldots, n_g \\ & \mathbf{u} \in U \end{aligned} \tag{3.1.1}$$

The manipulated variables or inputs are $\mathbf{u} \in \mathbb{R}^{n_u}$, while $\mathbf{y}_p \in \mathbb{R}^{n_y}$ denotes the measured output that depend on the inputs. The objective of the problem is to minimize the cost function $\phi : \mathbb{R}^{n_u} \times \mathbb{R}^{n_y} \to \mathbb{R}$. The process constraints g_i are formulated as n_g inequalities depending on both $\mathbf{u}$ and $\mathbf{y}_p$, expressed as functions of the same dimensions as the cost function. Notice the absence of the process model, compared with Equation 2.2.1. The inputs are limited by the upper and lower bounds $U = \{\mathbf{u} \in \mathbb{R}^{n_u} : \mathbf{u}^{\mathrm{L}} \leq \mathbf{u} \leq \mathbf{u}^{\mathrm{U}}\}$. Inequalities between vectors are interpreted component-by-component. The cost and constraint functions are typically nonlinear, and then the problem is referred to as a NLP problem [5].

In general, the precise steady-state input-output map of the plant, $\mathbf{u} \mapsto \mathbf{y}_p$, is unknown. This could for example be due to unavailable measurements of $\mathbf{y}_p$. As a result, one must rely on an estimate provided by the best available process model, $\mathbf{F}(\mathbf{x}, \mathbf{u})$, which is a steady-state

model[6], where $\mathbf{x}$ is the vector of states.

$$\mathbf{y} = \mathbf{F}(\mathbf{x}, \mathbf{u}) \tag{3.1.2}$$

The estimated outputs $\mathbf{y}(\mathbf{u})$ can be obtained by solving this system of equations. Although we would prefer to minimize Equation 3.1.1, we can only minimize the model:

$$\begin{aligned} \min_{\mathbf{u}} \quad & \Phi(\mathbf{u}) := \phi(\mathbf{u}, \mathbf{y}(\mathbf{u})) \\ \text{s.t.} \quad & G_i(\mathbf{u}) := g_i(\mathbf{u}, \mathbf{y}(\mathbf{u})) \leq 0 \;\; i = 1, ..., n_g \\ & \mathbf{y} = \mathbf{F}(\mathbf{x}, \mathbf{u}) \\ & \mathbf{u} \in U \end{aligned} \tag{3.1.3}$$

When comparing Equation 3.1.3 to the plant optimization problem in Equation 3.1.3, the experimentally measured outputs $\mathbf{y}_p$ are replaced with $\mathbf{y}$ calculated from the process model in Equation 3.1.2. The cost function ϕ and constraint function g_i are the same, but the evaluations will be different because $\mathbf{y}$ has changed. The model equations can be interpreted as additional equality constraints for the optimization problem 3.1.3. The solutions to Problems 3.1.1 and 3.1.3 are generally different due to plant-model mismatch and disturbances, and using the optimum of 3.1.3 as setpoints may result in constraint violation and infeasible plant operation.

3.2 Modifier Adaptation

MA uses first-order corrections to the cost and constraint functions to fulfil the plant's NCO upon convergence. Correction terms depending on the current input are added to the cost and constraint functions of Problem 3.1.3. At the kth iteration with the operating point $\mathbf{u}_k$ the modified cost function $\Phi_{m,k}$ is formulated as

$$\Phi_{m,k} := \Phi(\mathbf{u}) + (\boldsymbol{\lambda}_k^{\Phi})^{\intercal} (\mathbf{u} - \mathbf{u}_k) \tag{3.2.1}$$

where $\boldsymbol{\lambda}_k^{\Phi} \in \mathbb{R}^{n_u}$ is the first order modifier for the cost function, defined as

$$(\boldsymbol{\lambda}_k^{\Phi})^{\intercal} := \frac{\partial \Phi_p}{\partial \mathbf{u}}(\mathbf{u}_k) - \frac{\partial \Phi}{\partial \mathbf{u}}(\mathbf{u}_k). \tag{3.2.2}$$

Further, the modified constraint functions $G_{m,i,k}$ can be set up as

$$G_{m,i,k} := G_i(\mathbf{u}) \; + \; \epsilon_{i,k} \; + \; (\boldsymbol{\lambda}_k^{G_i})^{\intercal} (\mathbf{u} - \mathbf{u}_k) \leq 0, \quad i = 1, ..., n_g \tag{3.2.3}$$

with the zero order constraint modifier $\epsilon_{i,k} \in \mathbb{R}$ and the first order modifier $\boldsymbol{\lambda}_k^{G_i} \in \mathbb{R}^{n_u}$ given by

$$\epsilon_{i,k} := G_{p,i}(\mathbf{u}_k) - G_i(\mathbf{u}_k) \tag{3.2.4}$$

$$(\boldsymbol{\lambda}_k^{G_i})^\intercal := \frac{\partial G_{p,i}}{\partial \mathbf{u}}(\mathbf{u}_k) - \frac{\partial G_i}{\partial \mathbf{u}}(\mathbf{u}_k). \tag{3.2.5}$$

Note that the modified cost function in Equation 3.2.2 does not include a zero-order modifier due to it being a constant term that would not affect the optimal point.

Figure 3.2.1 illustrates the modified constraint function as well as the modifiers. The zeroth-order modifier $\epsilon_{i,k}$, as shown in the figure, describes the direct difference in constraints computed from plant measurements and estimated values from the process model at $\mathbf{u}_k$. Furthermore, the first order modifiers $\boldsymbol{\lambda}_k^{\Phi}$ and $\boldsymbol{\lambda}_k^{G_i}$ correspond to the deviations in the plant and model gradients, respectively. The cost and constraint gradients must be accessible at the current $\mathbf{u}_k$ to enable the calculation of the first order modifiers. Estimating the plant gradients at each RTO iteration can be problematic. The reason is that the plant measurements are affected by noise in the measurement equipment, making it difficult to determine the exact gradients.

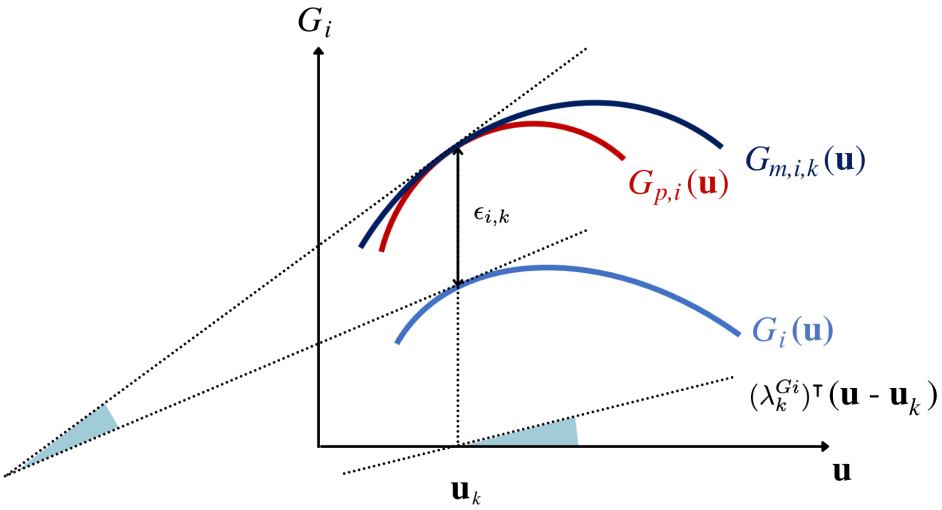


Figure 3.2.1: Visualization of the modified constraint function $G_{m,i,k}$ with the modifiers $\epsilon_{i,k}$, λ_k^{Φ} and $\boldsymbol{\lambda}_k^{G_i}$. Figure based on Figure 1 in [5] and Figure 1 in [6].

At the current point $\mathbf{u}_k$, the subsequent vector of optimal inputs $\mathbf{u}_{k+1}$ is computed by solving the following modified optimization problem:

$$\begin{aligned} \mathbf{u}_{k+1}^* &= \underset{\mathbf{u}}{\operatorname{argmin}}\, \Phi(\mathbf{u}) + (\boldsymbol{\lambda}_k^{\Phi})^\intercal\,(\mathbf{u} - \mathbf{u}_k) \\ \text{s.t.} \quad & G_{m,i,k} := G_i(\mathbf{u}) \;+\; \epsilon_{i,k} \;+\; (\boldsymbol{\lambda}_{k+1}^{G_i})^\intercal\,(\mathbf{u} - \mathbf{u}_k) \leq 0, \\ & i = 1, ..., n_g \\ & \mathbf{u} \in U \end{aligned} \tag{3.2.6}$$

A filter can be applied to the new optimal input to prevent overly large changes in the input, which is especially relevant in the presence of noise. The filtered new input is presented in

Equation 3.2.7, where $\mathbf{K} = diag(k_1, ..., k_{n_u}) \in \mathbb{R}^{n_u}$ is a diagonal matrix with the filter values $k_i \in (0, 1]$, $i = 1, ..., n_u$. These values can be altered based on the magnitude of noise. The filter works the same way as a trust-region, preventing each new input point from moving too far away from the prior one. Heavy filtering implies smaller iteration steps and results in slower convergence.

$$\mathbf{u}^*_{k+1} = \mathbf{u}^*_k + \mathbf{K}(\mathbf{u}^*_{k+1} - \mathbf{u}^*_k) \tag{3.2.7}$$

The same method can be used to filter the modifiers. The filtered modifiers for the next iteration can be represented by providing the filter parameters a_i, b_i, and c for the zeroth-order, constraint first order, and cost first order modifiers, respectively. Equation 3.2.8-3.2.10 display the filtered modifiers.

$$\epsilon_{i,k+1} = (1 - a_i)\epsilon_{i,k} + a_i[G_{p,i}(\mathbf{u}_k) - G_i(\mathbf{u}_k)] \tag{3.2.8}$$

$$(\boldsymbol{\lambda}^{G_i}_{k+1})^\intercal = (1 - b_i)(\boldsymbol{\lambda}^{G_i}_k)^\intercal + b_i \left[\frac{\partial G_{p,i}}{\partial \mathbf{u}}(\mathbf{u}_k) - \frac{\partial G_i}{\partial \mathbf{u}}(\mathbf{u}_k) \right] \tag{3.2.9}$$

$$(\boldsymbol{\lambda}^{\Phi}_{k+1})^\intercal = (1 - c)(\boldsymbol{\lambda}^{\Phi}_k)^\intercal + c \left[\frac{\partial \Phi_p}{\partial \mathbf{u}}(\mathbf{u}_k) - \frac{\partial \Phi}{\partial \mathbf{u}}(\mathbf{u}_k) \right]. \tag{3.2.10}$$

Finite difference approximations can be used to estimate plant gradients(Appendix A). To approximate multidimensional input gradients with a first-order finite difference, n_u step changes of the plant are required for n_u inputs. Gradient approximation is costly since it can be time demanding and result in the plant operating at sub-optimal points for longer periods because the plant must reach a steady state for each perturbation.

3.3 MDP Formulation for Steady-State Optimization

Recall the definition of a Markov decision process, MDP, in Definition 2.4.1. To enable the use of the Bellman equations in Section 2.4.2 to solve an optimization problem with RL, the problem has to be rewritten as an MDP. In other words, the different components of the MDP tuple must be defined corresponding to the variables presented in Equation 3.1.1 to make the RTO interpretable as an RL problem.

RL problems are typically *sequential* decision problems, where the problem is solved in series and not through one computation. This sequence is usually related to time and dynamics, absent in the steady-state optimization problem. Instead of timesteps, iterations corresponding to k can be used in place of the parameter t to keep track of the number of steps used to solve the optimization problem. This way, solving the steady-state optimization problem can be seen as a sequence of iterations aligned with the RL problem. The number of iterations used to arrive at the optimum represents one episode.

For a steady-state optimization problem, the state and action spaces are continuous. This calls

for an extension of the finite MDP formulation. S and A are not a finite set of states but a finite-dimension set of states. For a general continuous problem P is a probability *distribution*, not a matrix. Since P determines the system's dynamics, it takes a particularly simple form in the case of a steady-state problem. Connect to iterations. The reward function is based on the cost function, which is known, and therefore the expectation is no longer necessary. With these modifications of Definition 2.4.1 for continuous action and state spaces, we can define the respective components of the MDP as the following:

- S is a finite-dimension set of states, set to the state vector, $\mathbf{x}$
- A is a finite-dimension set of actions, where the actions are the input vector, $\mathbf{u}$
- P is a state transition probability, $P_{SS'} = \mathbb{P}[S' = s_{t+1}|S = s_t, A = a_t]$
- R is a reward function, where R is a sum of the negated cost function and penalty terms for the constraints. Using the notation $[v]^+ = \max\{v, 0\}$, it can be written out as follows

$$R(S, A) = -\Phi(\mathbf{u}) - \sum_{i=1}^{n_g} \kappa_i \left[G_i(\mathbf{u})\right]^+ \tag{3.3.1}$$

- γ is a discount factor, $\gamma \in [0, 1]$ that $\to 0$

There are several approaches for defining the MDP states for the steady-state optimization problem. Recall that by definition, a Markov state should contain all information of the history and completely characterize the process[4]. Translating this into a process plant can be done in different ways. The most straightforward approach is to set the MDP state equal to the process state $\mathbf{x}$ as presented. Several other state formulations are explored in Section 7 to investigate if this straightforward choice is actually the most appropriate.

Further, the input vector can make up the set of actions. The definition of an action aligns with the concept of manipulated variables, or inputs, in process control. The manipulated variables are the only variables that can be modified to solve the optimization problem and therefore, the only candidates to represent the actions in the MDP. The state and action definitions mean that the observation and action spaces are subsets of $\mathbb{R}^{n_x}$ and $\mathbb{R}^{n_u}$, respectively. The reward function in Equation 3.3.1, is formulated by negating the sum of the steady-state cost function and numerical penalties for constraint violations. The formulation assumes that Φ is a cost function for minimization, thereof the negation. For a maximization problem, Φ would not be negated.

The nature of the MDP formulation ensures that the action is always within the bounds of the action space. However, the limits on the states cannot be satisfied strictly as they may lead to infeasibility[34]. A solution to strictly enforce inequality constraints is to use penalty terms. These are added to the cost function to make constraint violations prohibitively expensive. There are several versions of implementing the penalty terms. The most straightforward approach is to use a uniform penalty constant, but this may lead to computational issues in the neural network for deep RL algorithms[34]. Another possibility is to use a 1-norm linear penalty, LP, i.e., a function with magnitude proportional to the constraint violation, which

is the one presented here in Equation 3.3.1. The penalty term is added to the cost function whenever a constraint G_i is violated. The absolute value of the violation is multiplied with a constraint penalty parameter, κ_i, which has to be tuned for the applicable problem. Alternatively, a dynamic penalty function could be implemented, described in the section below.

The objective of the steady-state optimization problem is to arrive at the global optimum as quickly as possible. It does not involve any long term goal, for example winning a game, and therefore no additional reward for a specific outcome. Consequently, the reward does not need to be discounted, as the highest possible immediate reward is always desired. Therefore, the discount factor γ is set to approach zero when solving the steady-state optimization problem.

3.3.1 Dynamic Penalty

The idea behind a dynamic penalty (DP) is that each penalty term is initialized to a small value and increased while training until it reaches the value of κ_i. This way, the agent can explore more of the action space in the initial training, and the constraints are gradually enforced. When linear or uniform penalty terms are set prior to training, the result can be poor function approximation in the neural networks. At worst, it can lead to convergence problems and suboptimal solutions. The DP has proved to increase the stability of the neural networks and consequently reduce constraint violation and the training loss for a vehicle control example (see full paper [34]). Here a simplified version of a DP compatible with an RL python framework is proposed.

For DP, the reward function is equal to the one presented in Equation 3.3.1, but the constraint penalty parameter, κ_i for constraint i, is adjusted dynamically throughout the training process. The update of κ_i starts with κ_i being initialized to $\kappa_{i,min}$, which is typically a small number that has to be decided. For each timestep κ_i is increased by a factor of d_i, which is larger than 1. This update is a simplification compared with the DP paper, where the loss value determines if the penalty parameter is updated [34]. Since this implies many more updates, d_i and $\kappa_{i,min}$ should be set to smaller values than the examples from the original DP. Further, the update is continued until $\kappa_i \geq \kappa_{i,max}$, where $\kappa_{i,max}$ can be equal to the value one would use in the case of LP. The full DP update scheme can be written out as follows:

1. Set $\kappa_i := \kappa_{i,min}$
2. For each training timestep t update $\kappa_i := \min\{d_i\kappa_i,\, \kappa_{i,max}\}$
3. Continue the training until finished.

Chapter 4

Case Study: Williams-Otto Reactor

This section will introduce a process control case study called the Williams-Otto reactor. In 1960, the Williams-Otto reactor was presented to serve as a case study that could be used for direct comparison of computer control[35]. Since then, it has been applied to control schemes to identify limitations and performance. The reactor models a hypothetical reactor involving fictive chemical substances and presents a plant and process model with different model structures. The plant model will produce measurements for the process model. As a result, the system has a structural plant-model mismatch. Note that the process model is only for use in MA because the RL-RTO is model-free.

4.1 Reactor Description

In Figure 4.1.1 a simplified representation of the reactor is displayed. The reactor is an ideal continuous stirred tank reactor that generates the desired products P and E and an undesired byproduct G. The reactor feed consists of two continuous flows, F_A and F_B. The feed flows are pure, consisting of 100% A and B. F_A has a constant value of 1.8275 kg/s, while F_B varies and is one of the system's manipulated variables or inputs. The reactor temperature, T_R, is the second input. The temperature of the reactor influences the reaction rates. As a result, the input vector is $\mathbf{u} = [F_B, T_R]^\intercal$.

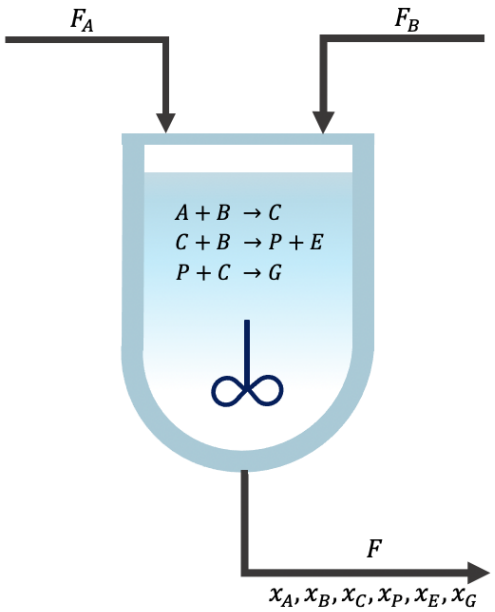


Figure 4.1.1: Illustration of the Williams-Otto reactor with feed streams F_A and F_B, outlet stream F and the plant reactions.

The model describing the complete reactor system will be called the plant. The reactions that occur in the reactor are shown in Equations 4.1.1–4.1.3. Reactions 4.1 and 4.2 yield the desired products, while reaction 4.3 yields the byproduct. Each reaction has its own rate constant k_1, k_2, and k_3.

$$A + B \xrightarrow{k_1} C \tag{4.1.1}$$

$$C + B \xrightarrow{k_2} P + E \tag{4.1.2}$$

$$P + C \xrightarrow{k_3} G \tag{4.1.3}$$

To simulate plant-model mismatch, MA employs an inaccurate model featuring the simpler reaction scheme shown in Equations 4.1.4–4.1.5. The simplification eliminates the intermediate product C, resulting in altered mass balances and reaction rate constants, k_1^* and k_2^*. Because of the simplification, the calculated optimum operating points for the plant and the model will differ.

$$A + 2B \xrightarrow{k_{1^*}} P + E \tag{4.1.4}$$

$$A + B + P \xrightarrow{k_{2^*}} G \tag{4.1.5}$$

4.2 Plant and Model Equations

The plant and process model have been used in various papers, and the model equations used here are found in [36], except we use mass basis and not molar basis. The Arrhenius equation in Equation 4.2.1 [37] can be used to calculate the reaction rate constants, k_i. Here A_i is the pre-exponential factor and $E_{a,i}$ is the activation energy in K associated with the reaction i. T_R is the reactor temperature in °C.

$$k_i = A_i e^{-E_{a,i}/(T_R+273.15)}, \quad i = \{1, 2, 3, 1^*, 2^*\} \tag{4.2.1}$$

The plant states are a six dimensional vector consisting of the mass fraction of each component i in the reactor, $\mathbf{x} = [x_A, x_B, x_C, x_E, x_P, x_G]^\intercal$. The model has the same state vector except for x_C, and is therefore defined as $\mathbf{x}_m = [x_A, x_B, x_E, x_P, x_G]^\intercal$. The expressions for the reaction rates, r_i, for each reaction i, are dependent on the plant and model states. They can be deduced from fundamental rate expressions [37]. The reaction rates for the plant and the model respectively can be written in terms of M_t, the total mass in the reactor (constant at 2105.2 kg, see [38]) as follows:

$$r_1 = k_1 x_A x_B M_t \tag{4.2.2}$$
$$r_2 = k_2 x_B x_C M_t \tag{4.2.3}$$
$$r_3 = k_3 x_C x_P M_t. \tag{4.2.4}$$

$$r_{1^*} = k_{1^*} x_A {x_B}^2 M_t \tag{4.2.5}$$
$$r_{2^*} = k_{2^*} x_A x_B x_P M_t. \tag{4.2.6}$$

The total input flow rate, F, is the sum of the two reactant flow rates: $F = F_A + F_B$. The reactor is defined as a closed system for which conservation laws apply. Due to the conservation of mass, the reactor's output flow rate equals F [13]. Finally, using steady-state mass balances derived from mass conservation laws, the steady-state mass balances for the plant and the model respectively, can be expressed as follows,

$$\begin{aligned} Fx_A &= F_A - r_1 \\ Fx_B &= F_B - r_1 - r_2 \\ Fx_C &= 2r_1 - 2r_2 - r_3 \\ Fx_E &= 2r_2 \\ Fx_P &= r_2 - 0.5r_3 \\ Fx_G &= 1.5r_3. \end{aligned} \tag{4.2.7}$$

$$
\begin{aligned}
Fx_A &= F_A - r_{1^*} - r_{2^*} \\
Fx_B &= F_B - 2r_{1^*} - r_{2^*} \\
Fx_E &= 2r_{1^*} \\
Fx_P &= r_{1^*} - r_{2^*} \\
Fx_G &= 3r_{2^*}.
\end{aligned} \tag{4.2.8}
$$

The numerical values of the pre-exponential factor and activation energy are given in Table 4.2.1.

Table 4.2.1: Kinetic parameters for the plant and model reactions from [7] [1].

Reaction	**Pre-exponential factor**			**Activation energy**		
	Parmeter	Value	Unit	Parameter	Value	Unit
1	A_1	$1.6599{\cdot}10^6$	1/s	$E_{a,1}$	6666.7	K
2	A_2	$7.2117{\cdot}10^8$	1/s	$E_{a,2}$	8333.3	K
3	A_3	$2.6745{\cdot}10^{12}$	1/s	$E_{a,3}$	11111	K
1*	A_{1^*}	0.04979	1/s	$E_{a,1^*}$	6513.6	K
2*	A_{2^*}	0.01832	1/s	$E_{a,2^*}$	11111.3	K

4.3 Optimization Problem

The objective of the reactor is to maximize the economic plant profit by manipulating the inputs $[F_B, T_R]^\intercal$. The plant cost function, Φ_p, is presented in Equation 4.3.1 with the associated constraints, $g_{p,1}$ and $g_{p,2}$. The cost function consists of a sum of product profit, reduced by the cost of the feed flows. Note that it is formulated as a maximization problem, in contrast to the minimization problem in the steady-state optimization problem (Equation 3.1.1). A maximization problem is a minimization problem multiplied by -1. The unit of the cost function is \$/s. P_i denote the price for the component i. The two constraints limit the weight fractions of two states, x_A and x_G. In addition, there are upper and lower bounds on the inputs.

$$
\begin{aligned}
\max_{F_B, T_R} \quad & \Phi_p = P_P x_P F + P_E x_E F - P_A F_A - P_B F_B \\
\text{s.t.} \quad & g_{p,1} = x_A - 0.12 \leq 0, \\
& g_{p,2} = x_G - 0.08 \leq 0, \\
& F_B \in [4, 7], \\
& T_R \in [70, 100] \\
& \text{Eq.4.2.7}
\end{aligned} \tag{4.3.1}
$$

[1] Model values were decided through personal communication that are not in the original paper.

In addition to the inequality constraints and bounds, the plant equations, Equation 4.2.7, must be satisfied. It should be noted that in the standard RTO approach, we only have the process model, 4.2.8 (see Section 3.1). The prices for the various chemical compounds are shown in Table 4.3.1. Scenario 1 is the base case, while scenario 2 is used to execute a step change on the prices. The plant equations 4.2.7 are used to generate the measurements in both MA and DDPG implementations.

Table 4.3.1: Prices for the products P and E, and the feed reactants A and B for 2 scenarios[8].

Price	P_P	P_E	P_A	P_B	Unit
Scenario 1	1043.38	20.92	79.23	118.34	[$/kg]
Scenario 2	1073.25	25.92	94.18	95	[$/kg]

4.3.1 MDP formulation

The MDP formulation for a steady-state optimization problem presented in Section 3.3 can be written out for the Williams Otto reactor. Since plant measurements are utilized, the cost function in 4.3.1 is optimized, and the full state vector is used, including x_C. Including the state and input parameters, in addition to the cost function and constraints from Equation 4.3.1, the continuous MDP formulation for the Williams-Otto reactor can be written as follows:

- S is a finite-dimension set of states, where the states are $\mathbf{x} = [x_A, x_B, x_C, x_E, x_P, x_G]^\intercal$
- A is a finite-dimension set of actions, where the actions are the input vector $\mathbf{u} = [F_B, T_R]^\intercal$
- P is a state transition probability, $P_{SS'} = \mathbb{P}[S' = s_{t+1}|S = s_t, A = a_t]$ with A and S defined above
- R is a reward function, where R is a sum of the cost function Φ_p in Equation 4.3.1 and LP functions for the constraints. Since Problem 4.3.1 is a maximization problem, Φ_p is not negated as in the steady state optimization MDP formulation in Equation 3.3.1. It can be written out as follows

$$\begin{aligned} R(S, A) &= \Phi_p(\mathbf{u}) - \sum_{i=1}^{n_g} \kappa_i \left[G_i(\mathbf{u})\right]^+ \\ &= P_P x_P F + P_E x_E F - P_A F_A - P_B F_B \\ &\quad - \kappa_1 \left[x_A - 0.12\right]^+ - \kappa_2 \left[x_G - 0.08\right]^+ \end{aligned} \tag{4.3.2}$$

- γ is a discount factor $\gamma \in [0, 1]$ that $\to 0$

When there is access to plant measurements, the measured states in the plant simulation can be obtained by solving the set of plant equations 4.2.7 with a root solver. In an actual plant, all measurements are contaminated with a certain noise level. To implement measurement noise, each measured parameter is perturbed by a random normally distributed term with a predefined standard deviation σ.

Chapter 5

Implementation

The programming tools used to optimize the Williams-Otto reactor with MA and RL are introduced in the following sections. In addition, a description of how the case study was implemented and the initial simulation settings for the system parameters are presented.

5.1 Programming Environment

Python version 3.9.9 was used to implement and simulate the different schemes. CasADi [39] was used as a tool for setting up and solving the optimization problems, with IPopt [40] used for the optimization in MA. Further, the numpy package [41] was used for mathematical programming. The SciPy package provided the root solver of the plant equations [42].

5.2 Reinforcement Learning Tools

For the generation of the RL environment, OpenAI Gym was used. OpenAI Gym is an open-source Python library for developing and comparing RL algorithms. It offers a standard application programming interface (API) that enables communication between learning algorithms and environments, in addition to a standard set of environments compatible with that API [32]. OpenAI Gym also provides a framework for defining custom RL environments used for this project. Three key components must be defined to generate a custom environment: the action and observation spaces (with suitable initializations); a step-function that takes an action as input and returns a reward, a new state and a done-parameter; and a reset function. A detailed description of how these components and the environment were defined for the Williams-Otto reactor can be found in Appendix C.

To minimize sources of error and efficiently and robustly program the DDPG algorithm, Stable Baselines 3 was used. Stable Baselines 3 provides open-source Python implementations of various RL algorithms [14]. The algorithms have a consistent interface and extensive documentation, making training and comparing various RL algorithms simple. In addition, the framework is compatible with Gym environments. In Appendix D use of the Stable Baseline

3 DDPG algorithm to solve the Williams-Otto reactor Gym Environment is presented. Stable Baselines 3 provides three algorithms suitable for continuous action spaces; DDPG, TD3 and SAC. Once the Gym environment is defined, it is easy to run and train all these algorithms. Since TD3 and SAC showed performance at approximately the same level or slightly worse than DDPG, the results from DDPG are the centre of attention for results and discussion. Simulation plots of TD3 and SAC can be found in Appendix E. Examples of the fundamental scripts that were used in the simulations can be found in the following Git repository [43].

5.3 Hyperparameter Tuning

Fine-tuning of certain hyperparameters was performed using Optuna. Optuna is a software framework for automatic hyperparameter optimization, where the user defines the parameters to tune[44]. It is particularly designed for machine learning and works with Stable Baselines 3 algorithms. The 6 hyperparameters discussed in Section 2.4.6 were optimized. The interval or values that the optimizer should explore were defined prior to optimization. The Optuna optimizer was run for 100 number of trials. The output was the settings that gave the highest training reward for a given number of training steps.

The possible values for the discount rate were set to $\{0, 10^{-5}, 10^{-4}, 10^{-3}\}$. For learning rate a continuous interval was given from $[10^{-5}, 1]$. Batch and buffer sizes had the following possible discrete values: $\{16, 32, 64, 100, 128, 256, 512, 1024, 2048\}$ and $\{1\,000, 10\,000, 100\,000, 1\,000\,000\}$. Finally, the possible τ values were set to $\{0.001, 0.005, 0.01, 0.02, 0.05, 0.08\}$. Remaining hyperparameters were left untuned and assigned the default Stable Baselines 3 values.

5.4 Case study

Simulations of the standard MA and RL-RTO using DDPG schemes for optimizing the Williams-Otto reactor were carried out for various cases. The standard MA implementation provides a benchmark for comparison purposes for RL-RTO. Since we measure the weight fractions, $\mathbf{x} = [x_A, x_B, x_C, x_E, x_P, x_G]^\intercal$, which are the states, we get that $\mathbf{x} = \mathbf{y}$.

Due to MA being a base case for illustration of an example of a conventional RTO implementation, no measurement noise was introduced in the MA case. This is because MA is very sensitive to noise, and since it is presented for comparison with the RL-RTO, the optimal performance is used as a benchmark. However, introducing noise in the RL environment can possibly improve performance, leading to increased exploration. Recall that the actions are intentionally exposed to a certain level of random noise to ensure exploration in DDPG. The measurement noise was assumed to be independent and identically distributed with a normal distribution having zero mean and standard deviation σ. Our experiments used a relatively low level of measurement noise, $\sigma = 0.00008$, which corresponds to 0.1% of the typical values of smaller constraint function g_2.

In simulations of MA, the initial point $\mathbf{u}_0$ was$[7, 70]^\intercal$, and the maximum number of iterations was20. For RL-RTO, training episodes started from a point $\mathbf{u}_0$ randomly initialized within

the feasible region to ensure that the agent learns the true value function rather than only remembering the path. The maximum length of an episode (simulation) was set to 100. Plant gradients in MA were calculated with finite differences (Appendix A), where the componentwise perturbations in $\mathbf{u}$, were set to $h = 10^{-4}$. The filter parameters from Equation 3.2.7, 3.2.8, 3.2.9 and 3.2.10 were set to $k_i = 0.4$, $i \in \{1, 2\}$, for the input filter and $a_i = b_i = c = 0.6$, $i \in \{1, 2\}$, for the modifier filters. The filters on the modifiers were implemented for the standard MA. The constraints were enforced in the RL-RTO using LP terms, where the penalty parameters κ_1 and κ_2 require tuning to ensure that the optimum of the reward function equals the plant optimum. The global optimum for the plant is achieved at $\mathbf{u} = [4.3894, 80.4948]^\intercal$, which was the desired point of convergence for the algorithms. The optimal profit is $\Phi_p^* = 75.8187$. The model is optimized at $\mathbf{u} = [4.5684, 100]^\intercal$, which substantiates the structural plant-model mismatch in the MA case. These optimums are obtained by optimization of the plant and the model separately in Problem 4.3.1.

Several simulations were run for different allocations of total training timesteps, T, to investigate the degree of constraint violation for the RL-RTO. This also supports a study of DDPG's sample efficiency. The plant was run for 100 simulations at each setting. The average training and optimum rewards were recorded with a calculated average degree of violation to determine if the RL-RTO can achieve feasibility at the point of convergence.

Further, a step on the prices was executed for the RL-RTO, to study the adaptive behaviour of RL. The new prices are presented in Table 4.3.1 and give a new plant optimum at $\mathbf{u} = [5.6324, 88.6972]^\intercal$ [8]. Although the prices are known parameters in which changes are not a typical disturbance, such changes could, in practice, apply to any parameters, e.g., kinetic parameters. Therefore, the results can provide insight into cases of drifting parameters or varying disturbances in the environment. The step was executed after 1 000 training timesteps on the original settings, and the training was continued for another 3 000 timesteps after the step change.

Chapter 6

Results and Discussion

The results from the Williams-Otto reactor using MA and DDPG are presented in the following sections. Before simulation of the RL-RTO eitn DDPG, the hyperparameters and environment parameters had to be tuned, and the first section presents the tuned parameters. The second section presents the simulation results from the case study and a discussion summarizing potential further work.

6.1 Parameter Tuning

Both hyperparameters and constraint penalty parameters had to be tuned for optimal performance of the RL-RTO. In this section, a description of the tuning for the constraint penalty parameters from Equation 4.3.2 is presented first, including plots of the reward function for visualization. Further, the rest of the hyperparameters tuned using Optuna are presented. The effect of the parameter T, the total number of training steps, is discussed in the end.

6.1.1 Constraint Penalty Parameters

The reward function for the Williams-Otto reactor in Equation 4.3.2 is what the agent should maximize. Since the discount rate is set to $\gamma \approx 0$, the agent should try to reach the global optimum of the reward function as quickly as possible. The reward function is an essential part of the RL problem, and its shape directly affects how the agent acts. The reward function has one penalty parameter for each of the two constraints, κ_i, $i \in \{1, 2\}$. The constraints are implemented with LP terms as the reward function was formulated. As a result, the magnitude of the penalty increases for increasing violations of the constraints. This constraint implementation requires tuning of the penalty parameters, which can have a large impact on the performance. This is because the penalty parameters directly affect the reward function, which is to be maximized, and changing the values can yield completely different optimums. Thus, one of the first steps for solving the RL-RTO problem was to tune the parameters to sufficient values.

To illustrate the shape of the reward function and the constraints, two base versions are plotted in 3D as a function of the action space $[F_B, T_R]^\intercal$, displayed in Figure 6.1.1. Figure 6.1.1A shows a plot of the plant cost function Φ_p from Equation 4.3.1. This corresponds to the shape of the reward function without the constraints. Since the cost function in the Williams-Otto reactor is negated due to maximization, the highest value of the z-axis is desired. Evaluations of the cost function therefore correspond to the plant profit. It can be observed from Figure 6.1.1A that the global maximum of the cost function is somewhere in the upper corner, where F_B is small and T_R is large within their respective bounds. In the opposite corner, the function is descending strongly.

In Figure 6.1.1B, the cost function with the constraint penalty implemented as uniform constant terms is plotted. For any constraint violation, the reward is set to a large negative constant equal to -200. Thus the feasible region of the action space is clearly visible, which is within the "triangle" shape. The constraint on g_1 is the upper line, and the constraint on g_2 is the lower line of the triangle. In addition, the global plant optimum is plotted as a red point, visible at the intersection of the two constraints. This version of the reward function could be implemented for the RL environment. However, it is a more straightforward and possibly naive approach compared to including deviation terms. Mathematically, the penalization scheme illustrated in view B breaks the continuity property for the economic objective. The LP scheme preserves continuity and sacrifices differentiability instead. This is preferable, and therefore reward function is applied for simulation uses LP terms. In contrast, MA preserves both continuity and differentiability but sacrifices accurate representations of the derivatives.

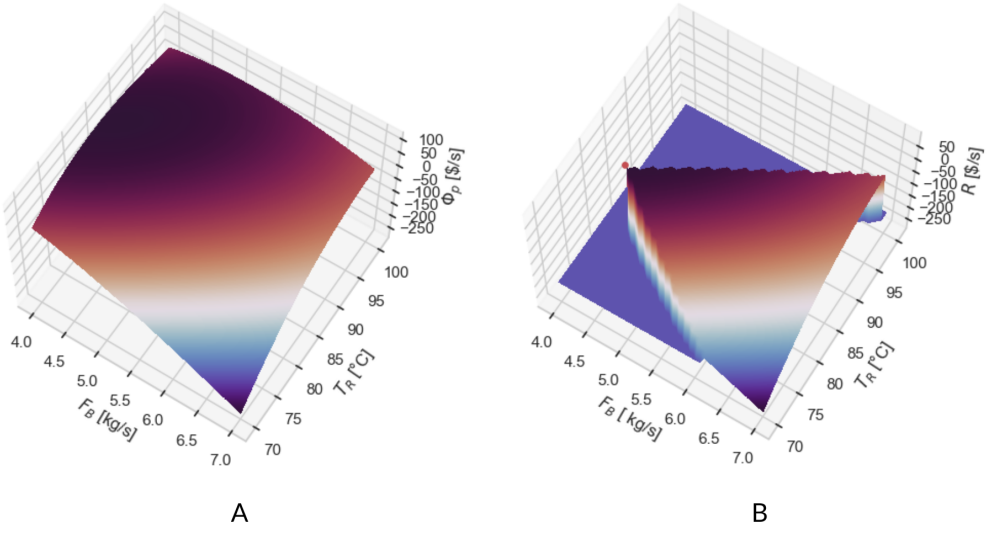


Figure 6.1.1: The economic reward as a function of the action space for the Williams-Otto reactor: A) The unconstrained function; B) The function with uniform penalties, and the global plant optimum indicated with the red dot.

The two plots above provide useful insights for tuning of the penalty parameters. First, note that the cost function increases more in the direction of constraint g_2 than g_1. Therefore

the constraint penalty parameter κ_2 should be larger than κ_1. Fine-tuning the constraint parameters can be a time-consuming task since it usually requires training of the agent for a number of different settings. The randomness of the action exploration results in random runs, which means that each tuning setting should be run several times to calculate a mean. However, additional knowledge like the global plant optimum can be utilized for more efficient tuning for the Williams-Otto case study. Therefore, instead of simulating the agent for tuning the constraint penalty parameters, a grid search problem was performed directly on the reward function. The (F_B, T_R) maximum for each setting was then compared with the global plant optimum to find the parameters that gave a maximum close to the global plant optimum. The boundary for a sufficient level of penalty was found for each constraint. The boundaries were $\kappa_1 = 410$ for g_1 and $\kappa_2 = 2\,000$ for g_2. Due to the mathematical formulation of the LP any value for κ_1 and κ_2 above the respective boundaries would be sufficient, as the global optimum would not change. The settings $\{\kappa_1 = 410, \kappa_2 = 2000\}$ were chosen for the rest of the simulations, but any higher values would also be adequate.

Finally, the 3D plot of the reward function with the chosen tuning settings of $\{\kappa_1 = 410, \kappa_2 = 2000\}$ in the LP terms is presented in Figure 6.1.1, together with the global optimum. The implemented constraint penalty parameters result in a reward function that descends in the directions of increased constraint violation. Compared with plot 6.1.1A, it is evident that the reward function is manipulated to curve down from constraint g_2. The result is that the plant global optimum is at the highest point of the reward function, which is desired. This reward function is the one that was implemented in the RL environment.

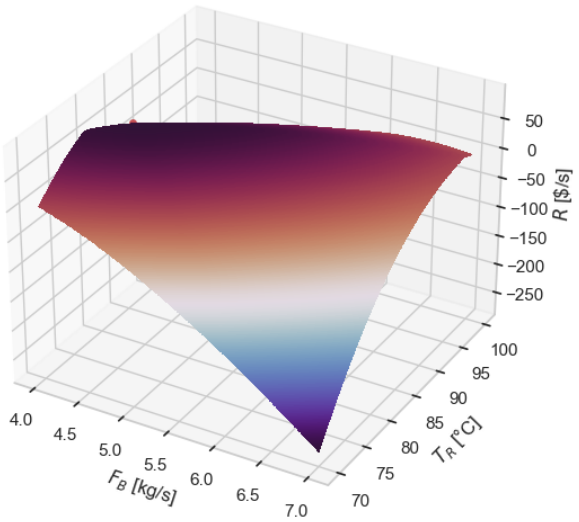


Figure 6.1.2: Reward function with tuned constraint penalty parameters and the global plant optimum indicated as a red point.

6.1.2 Hyperparameters

Optuna was run for the DDPG algorithm on the Williams-Otto reactor with the possible hyperparameter settings from Section 5. One hundred trials were used to determine the best combination of hyperparameters. In Table 6.1.1 the resulting tuned values are presented together with the untuned default values as a reference. As the parameters interact, one cannot determine precisely from the tuned set of parameters the impact the tuned parameters have on the algorithm, except that it is the combination that gave the highest trained reward. However, changes in the parameters can be discussed. First, it can be observed that γ has a drastic change from the default value of 0.99 to 0.00001. However, this was expected since the desired discount rate for the steady-state problem should be close to zero. Although it is not exactly zero, it is the smallest nonzero value in the given set. This means that future reward is discounted with an almost microscopic value. The reason why γ is not exactly equal to zero might be due to some computational disadvantages, as this is not a common setting for an RL problem.

Further, it can be observed that the learning rate is approximately half of the default value, which can imply that the algorithm's training is slowed down. The batch size is unchanged, while the buffer size decreases with a factor 1 000. This is the smallest possible size of the tuning setting and can increase the risk of sampling correlated data. However, it is still within the bounds of what is considered sufficient [27]. The exploration noise for the action is normally distributed noise with a standard deviation close to the default value.

Table 6.1.1: Default and tuned DDPG hyperparameters for the Williams-Otto reactor.

Parameter	γ	α	Batch size	Buffer size	τ	Noise type	Noise stdev
Default value	0.99	0.001	100	1 000 000	0.005	Normal	0.1
Tuned value	0.00001	0.0005467	100	1 000	0.05	Normal	0.09537

To verify the tuned hyperparameters for the DDPG algorithm, plots of the training process can be studied. Figure 6.1.3 shows how the episode reward progresses as a function of the number of training timesteps for the untuned and tuned algorithm. It is evident that the tuned version results in more efficient training. Where the reward for the untuned case reaches an approximately stationary level at around $t = 1500$, it takes around 1 000 timesteps for the tuned case. This is an improvement of 30%. Further, the maximum episode reward level is increased with tuning from around 6 000 to 7 000. In practice, the agent converges to a point closer to the plant optimum, where the reward is maximized. These results substantiate DDPG's challenge of high sensitivity to hyperparameters. However, the algorithm shows stability in the training after 1 000 timesteps, which is not always the case for DDPG. This indicates that the agent efficiently solves the problem with small variation after training for at least 1 000 timesteps.

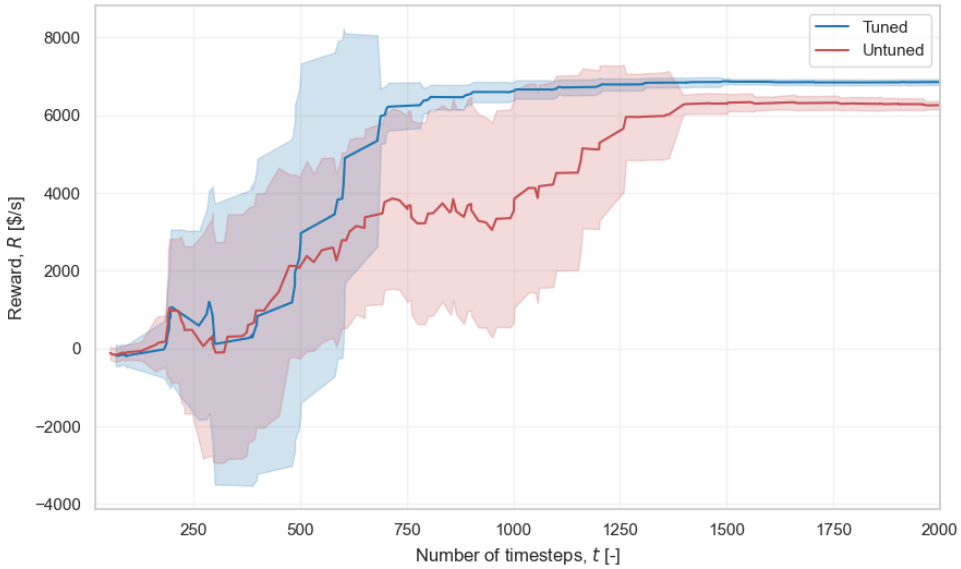


Figure 6.1.3: Moving average of training episode reward plotted as a function of number of timesteps with corresponding standard deviations illustrated with the shaded areas, for tuned and untuned hyperparameters in the DDPG algorithm.

6.1.3 Total Number of Timesteps

From Figure 6.1.3 it can be observed that the total number of training timesteps influences the episode reward. The parameter T is user-specified prior training for Stable Baselines 3 algorithms, and its value has a large impact on the performance of the RL-RTO. In order to study how this affects the agent's ability to solve the Williams-Otto optimization problem, a plot of the absolute value of input deviations from plant optimum in percent as a function of the total number of training timesteps is displayed in Figure 6.1.4. Figure 6.1.4 A and B show the deviation in F_B and T_R respectively. The figure confirms the assumption from the hyperparameter training plot; the deviation is high when T is small. Note that the plot does not provide insight into the direction of the deviations.

Although the deviation has drastic improvements for the first 500 timesteps, it takes many timesteps for it to be approximately zero. At 2 000 timesteps the deviations in both F_B and T_R are below 1%, while at 5 000 the deviation in F_B is still around 0.6% and 0.1% for T_R. As the number of timesteps increases, the deviation goes slowly towards zero, which is the ideal scenario. At $T = 15\,000$ the deviations in both inputs are below 0.1%. Consequently, there is a trade-off between the decreasing deviation and the training efficiency. For the RL-RTO, a small T-value is desired while still getting satisfactory optimization performance. For this case study, we have access to the plant equations, enabling quick training for large T. Therefore a total number of training timesteps equal to 5 000 is used in the remaining results to obtain better optimization. In reality, a lower number of training steps would be desired. In the next section, a more thorough study of the total number of timesteps in the context of constraint

violation will be presented.

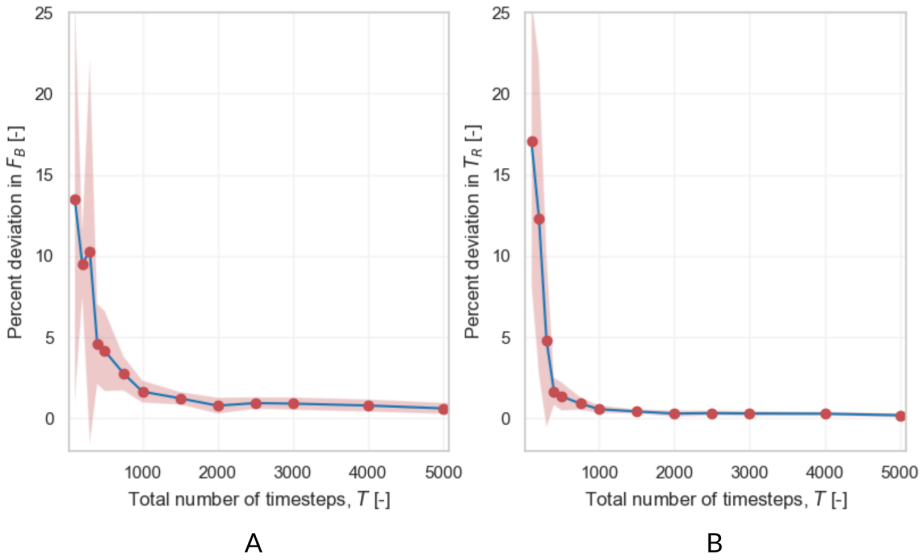


Figure 6.1.4: Average percent deviation from the global plant optimum in A) F_B and B) T_R plotted as a function of total number of training timesteps. The shaded areas are the standard deviation, the dots represent the settings that were simulated and the line shows the trend.

6.2 Case Study

In this section, the results from MA and RL-RTO simulation on the Williams-Otto reactor is presented. The standard MA without noise was simulated to show an example of how the Williams-Otto reactor can be optimized. Further, results from the DDPG RL-RTO simulation using the tuned parameters are displayed and discussed. In addition, thorough RL-RTO simulations were run for varying number of T to investigate degree of constraint violation. To study the RL-RTO's adaptive properties to changes in parameters, a step on the price parameters was introduced to a trained agent. Finally, a discussion of the most important challenges and possibilities in the results is provided to distinguish key parts for further work.

6.2.1 Standard Modifier Adaptation

To serve as a benchmark for comparison, standard MA with zero noise was simulated for the Williams-Otto reactor. The calculated point of convergence of the inputs F_B^* and T_R^*, the profit, Φ_p^*, and the constrained states, x_A^* and x_G^* are presented in Table 6.2.1. It is evident that the scheme converges to the global optimum. However, there is a small deviation from the true plant optimum in T_R^* of approximately 0.01 °C. This could be due to the error in the finite difference approximation of the gradients (see Appendix A).

Table 6.2.1: The optimal values of the inputs, profit, and constrained states from simulation of the standard MA.

Parameter	F_B^* [kg/s]	T_R^* [°C]	Φ_p^* [\$/s]	x_A^* [−]	x_G^* [−]
Value	4.3894	80.5057	75.9075	0.1200	0.0800

Figure 6.2.1 shows a plot of the calculated inputs for every iteration, where reactor temperature, T_R, is plotted as a function of the flow rate of B, F_B. The plo corresponds to a 2D projection of the plots in Figure 6.1.1 in the $F_B T_R$-plane. The plot includes the boundaries of the constraints g_1 and g_2, illustrated with red lines. The feasible region is illustrated as the light blue coloured area. Each blue star represents the one iteration of $\mathbf{u}_k$, and the dotted line links adjacent iteration points. The point of convergence is marked with a red star and is denoted $\mathbf{u}_k^f$. The initial input, $\mathbf{u}_0$, is the point in the lower right corner. It can be observed that the system moves closer to the optimum with every iteration, and by 13 iterations, the plant optimum is reached.

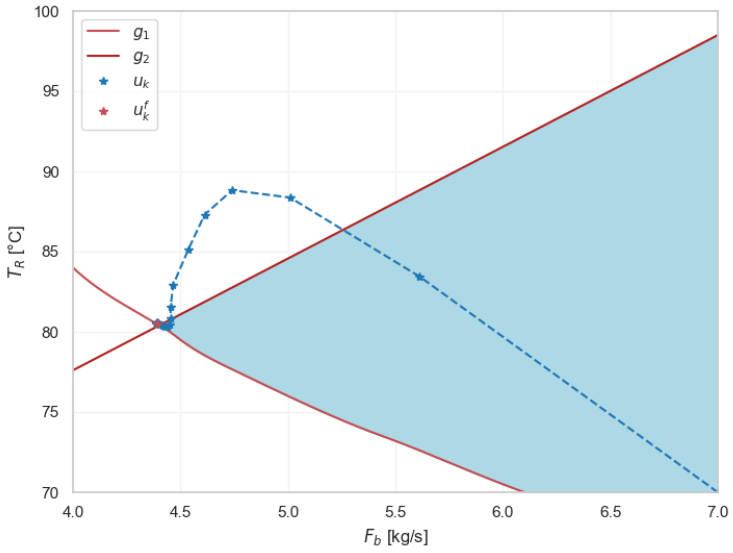


Figure 6.2.1: Plot of input iterations for standard MA. Each star represents one iteration and the dotted line shows the order of the iterations. The final iteration is labelled u_k^f. The constraint boundaries $g_1 = 0$ and $g_2 = 0$ are the red lines.

6.2.2 RL-RTO: DDPG

For the RL-RTO using the DDPG algorithm, the simulation the Williams-Otto reactor was done according to the settings in Section 5 $\kappa_1 = 410$ and $\kappa_2 = 2\,000$, the tuned hyperparameters and the total number of training steps equal to $T = 5\,000$. The point of convergence for the simulation resulted in the inputs, profit, and reward presented in Table 6.2.2. Due to the noise in actions and measurement, each run has variations. Therefore the simulation was run ten times to compute the mean and standard deviations. The significant number of digits for each value is based on the magnitude of the standard deviations. In addition, the environment generated reward, R^*, is presented. The reward has the same unit as the profit, [\$/s].

The point of convergence reveals some deviation from the true plant optimum. From the deviation simulation plots in Figure 6.1.4 this is expected since it was observed that even at 5 000 training timesteps, there are still deviations. The deviation suggest that there might be some bias in the neural network approximations of the value function. The environment's reward at the point of convergence, R^*, is higher than the actual plant profit, Φ_p^*. The difference in the reward and profit is due to the measurement noise added to each weight fraction, resulting in some deviation. Further, the mean evaluations of x_A^* and x_G^* show that the constraints are not violated. However, the mean might conceal some possible violations, addressed in the following section.

Table 6.2.2: The optimal values of inputs, profit, reward and constrained states from simulation of the RL-RTO using DDPG.

Parameter	F_B^* [kg/s]	T_R^* [°C]	Φ_p^* [\$/s]	R^* [\$/s]	x_A^* [−]	x_G^* [−]
Value	4.404	80.58	75.73	76.25	0.1194	0.0800

In Figure 6.2.2 the average input iteration simulation plot of the DDPG RL-RTO is displayed. The plot corresponds to the MA simulation in Figure 6.2.1. The input iterations in Figure 6.2.2 show that the DDPG-agent is able to optimize the system efficiently. From the plot, it can almost look like the simulations converge to the optimum in one iteration. However, when studying the iterations, it is evident that the agent uses 4 steps to meet the specified tolerance for convergence, where the first has by far the largest change. The following changes in the iterations are almost microscopic in the scale of the action space. Compared to the standard MA, the DDPG is more efficient in the optimization, whereas the standard MA required 13 iterations.

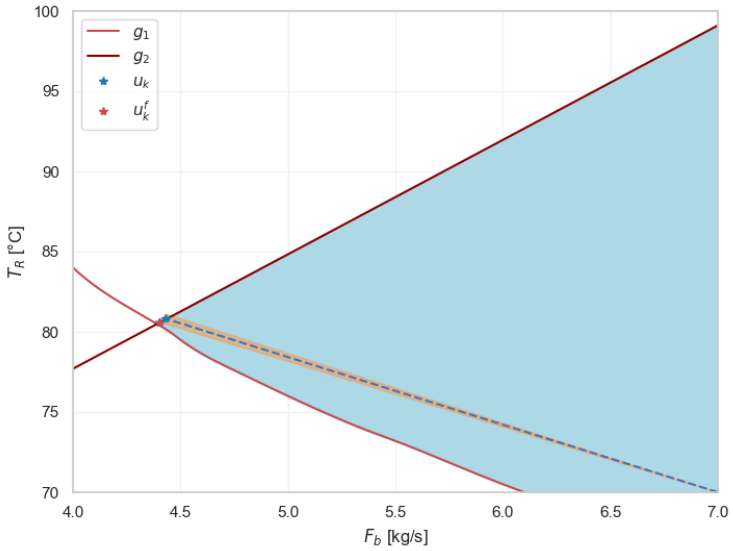


Figure 6.2.2: Plot of input iterations for RL-RTO using DDPG. The shaded orange area represents standard deviation in the iterations.

6.2.3 Constraint Violations

While tuning the parameters and studying the effect of a total number of training timesteps, it was observed that the RL-RTO tended to converge to a point that deviated from the optimum. Convergence to a feasible point is an essential property in RTO. Since the plant optimum for the Williams-Otto reactor is at the intersection of two constraints, guaranteed feasibility can be a challenge for the agent. From Figure 6.1.1 it can be observed that the region of optimum is relatively flat. Therefore there is a risk of the agent converging to points that violate the process constraints. To study the degree of constraint violation in the light of a total number of training timesteps, more thorough simulations were run for varying timestep budgets.

The RL-RTO was trained on the Williams-Otto environment for 100 simulations at the following number of total timesteps, $T = \{500, 1\,000, 2\,000, 5\,000\}$. For each value of T, the number of final points satisfying both constraints was counted to give a score out of 100, where a higher score is better. This result is referred to as finding feasible policy. Further, two average rewards were calculated: the reward at the final point over all 100 simulations, and over the subset of feasible policies, respectively. In addition, an average degree of constraint violation was calculated, where one simulation would receive a violation score of 100 for violating both constraints, 50 for violating one and 0 for a feasible policy at the point of convergence. The results are presented in Table 6.2.3

Table 6.2.3: Summary of agent training result for 100 simulations of RL-RTO for varying number of T.

Number of total timesteps T	**500**	**1 000**	**2 000**	**5 000**
Finding feasible policy/100	22	52	63	69
Average reward at final point (all policies)	66.20	74.16	76.14	76.35
Average reward at final point (feasible policies)	71.58	75.15	76.13	76.34
Average degree of violation	39.0	24.0	18.5	15.5

The number of feasible policies shows the most significant increase from $T = 500$ to 1 000, where the number is more than doubled. This increase corresponds to the trends observed in the training and deviation plots in Figure 6.1.3 and 6.1.4. Further the improvements from 1 000 to 2 000 are around 20%, while only a slight gain (3 policies) from 2 000 to 5 000. The average rewards at the final point for all policies also increase with T, but similarly to the number of feasible policies, the changes from $T = 2\,000$ to 5 000 are small. Note that the reward is higher than the actual evaluated cost at the true optimum, 75.8187. The average reward at the final point for the feasible policies is a significantly larger at $T = 500$ compared with the average reward for all policies. This observation shows that the feasible policies are closer to the optimum on average. For $T = 1\,000$, 2 000 and 5 000, the average reward for all policies are slightly higher than for the feasible policies. This indicates that the critic is an imperfect replacement for the reward function, where the critic's approximated optimum might be right outside the feasible region. Consequently, the action that gives the highest reward can violate the constraints, which again results in the feasible rewards being slightly smaller.

The average degree of violation quantifies the problem of infeasible policies. At $T = 500$ the degree of violation of 39 implies that on average almost half of the simulations violate one constraint. The degree of violation follows an opposite trend compared with the number of feasible policies, which decreases with increasing number of total timesteps. At $T = 5\,000$ the degree of violation is 15.5, which means that the 31 infeasible policies only violate one constraint each, most likely g_2. This degree of violation would not be acceptable for implementation in a real plant. The number of required timesteps to reduce the violation degree further is possibly very high, since the 3 000 additional timesteps from 2 000 to 5 000 resulted in only a decrease of 3. However, increasing the number of timesteps is not desired for RL-RTO. Consequently, there is a trade-off between the degree of violation and training time.

6.2.4 Step Change on Prices

A step change was applied to the reactant and product prices to study the adaptive characteristic of the RL-agent. The new prices are presented in Scenario 2 in Table 4.3.1 and implemented according to the settings in Section 5.2. The simulation was run ten times, and the calculated mean episode reward and associated standard deviation as a function of the training steps are displayed in Figure 6.2.3.

For the first 1 000 training steps, the reward increases rapidly, similar to what is observed in Figure 6.1.3, and it reaches a plateau of approximately 7 000 in average episode reward. At

$t = 1\,000$, the prices are changed, and the reward drops momentarily. After that, the reward progresses quite turbulently, but at around $t = 2\,000$, it can be observed that the reward increases gradually until it reaches a new plateau at around $t = 2\,500$. The reward also shows a couple of outliers after this, which explains the slight drop and increased standard deviation. The outliers influence the neighbouring timesteps since the mean is a moving average. It can be observed that a higher reward is reached for the second scenario. This is due to the formulation of the cost function, which results in the new optimum having a higher reward. This means that the maximum profit for these prices is higher than in the first scenario. However, the trend of interest is that the reward reaches two almost stationary levels, one before and one after the change in prices.

The observed trend proves that the agent is indeed adaptive to changes in the environment. Adaptivity is one of the most attractive properties of RL since it automatically adjusts to changes in the environment's parameters. In the standard RTO, changes in the environment would require modifications in the model, which can be costly. Since RL is adaptive and model-free, this is not required, which can be a profitable trait. Here the changes are on prices, but it could be any parameter in the environment from the agent's perspective. Prices are known, which makes this example trivial, but if one imagines a change in an unknown variable, the value of the results becomes evident. For example, there might be changes in the kinetic parameters that are usually found through experiments. New experimental data would be required to update the kinetic parameters of the model, which is time-consuming.

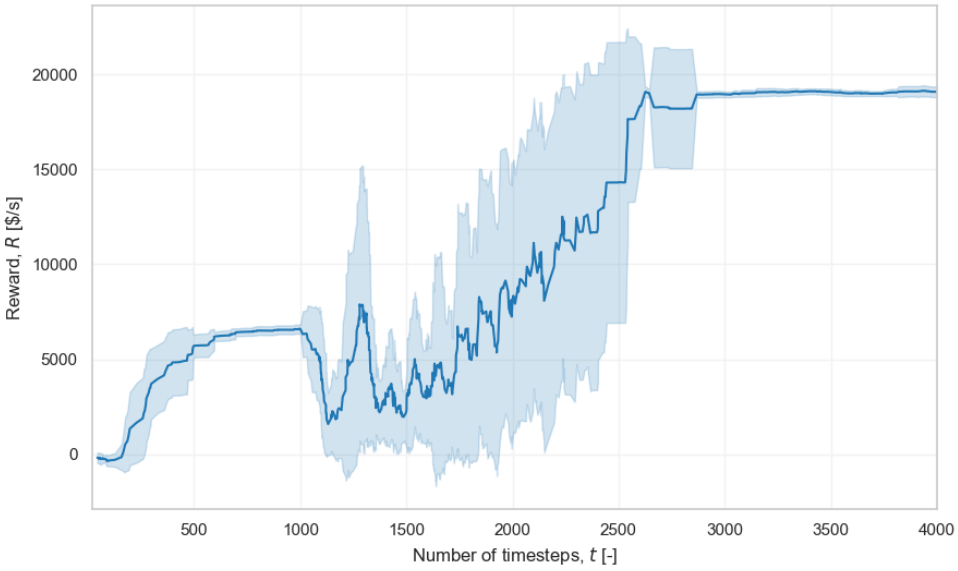


Figure 6.2.3: Moving average of training episode reward plotted as a function of number of timesteps, with associated standard deviations shaded, when a step change on the prices occurs at timestep 1 000.

The step can be seen as a change in an unknown variable or disturbance, for which it would not be possible to observe the change. From the tuned hyperparameters in Table 6.1.1, the replay

buffer size was set to 1 000. This means that it stores data from the 1 000 previous timesteps, which the mini-batches are sampled from. In terms of the price change, the buffer would then contain data from before the change until $t = 2\,000$. As t gets closer to 2 000, the ratio of updated data in the buffer increases, and therefore the agent is gradually able to train for the new scenario. After $t = 2\,000$, the reward increases rapidly. This increase indicates that the agent can train efficiently for the new scenario once the replay buffer contains a larger ratio of updated data. However, training efficiency after changes in parameters largely depends on how drastic the changes are to the environment.

Figure 6.2.4 shows the inputs from the simulation of the agent before and after the changes in prices. The simulation is run after $t = 1\,000$ and $t = 4\,000$. The mean points of convergence for the ten runs for the two simulations are plotted together with the standard deviations and the true plant optima. The global plant optima for the two respective scenarios are displayed as red stars. It can be observed that the convergence behaviour is similar to the previous RL-RTO simulations, where convergence is reached in under 5 iterations. Due to the other iteration points being very close, only the point of convergence is plotted to make the points more distinct.

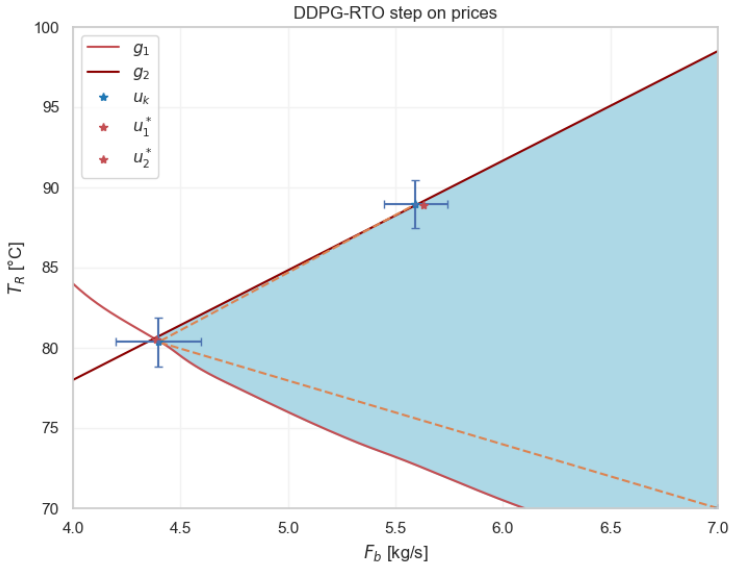


Figure 6.2.4: Plot of point of convergence in the inputs with associated standard deviation for RL-RTO using DDPG trained before and after the step change on prices. True plant optimum before and after price change are displayed as u_1^* and u_2^* respectively.

The agent proves to converge to two points that are on average close to the plant optimum for both scenarios. However, the standard deviations in both directions are significant. As observed in Figure 6.1.4 the standard deviations at this low number of training timesteps are quite high. The standard deviations would likely decrease by increasing the number of training timesteps. From the results in Section 6.2.3, it is likely that the simulations presented here have a significant degree of constraint violation. However, Figure 6.2.4 confirms the assumption

from the training plot; that the agent can solve the optimization problem for both scenarios, despite some variance in the results.

6.3 Challenges and Potential

The RL-RTO with DDPG solves the RTO problem for the Williams-Otto reactor. Given fundamental differences in the nature of the steady-state optimization problem and the RL problem, this is not a matter of course. The finding demonstrates that RL algorithms compatible with continuous action and observation spaces, in theory can be used as a solution technique in RTO. Consequently, it is an important result, as it opens new possibilities for RTO implementation. In addition, the RL-RTO with DDPG shows no sign of the common DDPG issues with stability, as the training plots reach stationary levels with minor standard deviations after a certain number of training steps. Compared with MA, the trained agent proves to be more efficient regarding the required number of iterations in the simulations (Figure 6.2.1 vs 6.2.2). Therefore, RL-RTO with DDPG fulfils the desired RTO property of fast convergence with a trained agent.

A challenge with RL-RTO is eliminating the deviation between the point of convergence and the plant optimum to ensure feasibility. The deviation could be due to some over-estimation bias when learning the critic network that tries to fit the measurement noise. This leads to an inaccurate approximation of the reward function. The shape of the reward function can likely magnify the problem. Figure 6.1.1 shows that the region of the optimum in the reward function is a relatively flat area. This makes it difficult for the agent to pinpoint the optimum accurately. Due to the optimum being at the intersection of the two constraints, the agent has to navigate with high precision. As a result, the algorithm needs several timesteps to map out the exact boundaries of the constraints. If the number of timesteps is set too low, there is a high risk of constraint violation and a non-feasible point of convergence as observed from Table 6.2.3. In practice, for a process plant, a non-feasible point is highly undesired as it can result in for example product impurity that makes the product impossible to sell. The consequences can be even worse if the constraints ensure safe and stable operation, as violations could put health and safety at risk. Consequently, the number of total timesteps in RL-RTO should be set to the lowest possible value that still can be accepted regarding the constraints.

The underlying theoretical reason for the optimum deviation for low T is related to the trained critic networks not being equal to the actual action-value function (Equation 2.4.6). The role of the action-value function is to calculate the sum of the predicted reward given an action and a state for a certain number of timesteps. Since the discount factor, γ, was set to approach zero in the steady-state MDP formulation 3.3, the sum reduces to the actual reward. Consequently, one can question the need for training two neural networks for the critic when we know the exact reward function. In general RL problems, the reward function can be hidden, and the agent has to discover it through trial-and-error. However, this is not the case in RL-RTO. From this perspective, it should be possible to develop a simpler RL algorithm that uses the true reward function instead of a critic.

Further, the sample efficiency for RL-RTO is high relative other RL problems. Recall that the magnitude of the total number of timesteps required to solve different RL tasks is 10^6 [10]. In this light, 1 000–15 000 timesteps are very little. This indicates that the problem is relatively computationally easy to solve for the agent. However, even 1 000 timesteps are a lot from an RTO point of view. As shown in the RL-RTO block diagram in Figure 2.5.1, each timestep requires one measurement and steady-state detection. Thus 1 000 measurements would result in the system operating at sub-optimal set-points for an extended period. A process operator or another RTO approach would likely be more efficient. Therefore it would be naive to implement this version of RL-RTO directly. Thus, measures to overcome this challenge should be further investigated.

Finally, two of the significant selling points for RL-RTO are the absence of a process model and being able to adapt to changes in the environment. This was substantiated by the prices' step-change results, where the agent automatically adapted to the new system. For a general RTO approach, a new model would be needed to adapt to changes in unknown parameters for the system. However, the MA approach proves to handle structural plant-model mismatch and therefore does not rely on a completely accurate model. Nevertheless, it still requires a model with a degree of resemblance to the system. While RL-RTO does not need a model, tuning of hyperparameters and efficient training introduces new required resources. The trade-off is difficult to determine without results from a real-life example, which should be looked into for further studies.

In short, the results show that the RL-RTO is able to optimize and adapt to the system of the Williams-Otto reactor and serves to open doors for further research in the field. However, the current results cannot be directly translated into a real-life process plant. A few suggested improvements and other possible algorithms will be presented in the next section to investigate the possibilities for making the RL-RTO more realistic in real-life implementation.

Chapter 7

Improvements and Further Work

In Section 6, it was observed that the required training time of the RL-RTO agent is the bottleneck for possible convergence and practical implementation, as the training requires process measurements. There are two possible angles to approach this challenge: modify the problem to a level of accepted sample efficiency or implement other RL algorithms where sample efficiency is not an issue. Both of these approaches will be discussed in the following section. Measures for improving the RL-RTO simulations are studied first, followed by a discussion of utilizing alternative RL algorithms in the RL-RTO. In the end, a final discussion sums up and discusses the potential of the RL-RTO compared with MA, in addition to suggesting a focus for further work.

7.1 Sample Efficiency

Sample efficiency decides the required number of training timesteps for the RL agent. The results in Figure 6.1.4, 6.2.2 and Table 6.2.3 showed that $T = 1\,000$ timesteps are required for the RL-RTO to converge to the region of the optimum, while over $T > 5000$ timesteps are required to ensure a feasible point of convergence. This level of T would not be practically feasible in a real plant. This raises the question of whether there could be any way to improve the sample efficiency in the proposed RL-RTO. The following sections investigate two ways of possibly improving the sample efficiency; modifying the MDP state formulation, and enforcing constraints with dynamic penalty terms.

7.1.1 Modified State Formulations

The choice of the MDP state is an essential part of an RL problem that directly influence the problem complexity and sample efficiency. Due to the difference in the definitions of state concept in a process control setting and an MDP, setting the two equal to each other in the MDP formulation 3.3 might be a naive strategy. The characteristic Markov property (Section 2.4.1) that defines a Markov state can be interpreted in different ways for a process system like the Williams-Otto reactor. The steady-state optimization problem makes the concept

of history not as relevant since the goal is to arrive at the optimum as quickly as possible. Therefore the meaning of the "past agent-environment interaction that makes a difference for the future" fades. Taking a step back and considering the state as a variable that fully describes the environment allows for several variants of the MDP state for the Williams-Otto reactor. With an experimental approach using simulation of the RL-RTO, the effect of different state formulations can be studied. The results can reveal the best approach and possibly contribute to improved sample efficiency.

In the RL-RTO simulations so far the MDP set of states, S, was set equal to the Williams-Otto plant state vector consisting of all the component weight fractions $\mathbf{x} = [x_A, x_B, x_C, x_E, x_P, x_G]^\intercal$. This state formulation will be referred to as the **full state**.

From the Markov property, a Markov state should contain all information about the environment's current condition. Considering the reactor (Figure 4.1.1), one can argue that the component weight fractions exclusively do not fully describe the environment. The inputs $\mathbf{u} = [F_B, T_R]^\intercal$ contain important information about the process. From this point of view, the two inputs should be a part of the MDP state. Consequently, an extended version of the state vector can be formulated as

$$\mathbf{x} = [x_A, x_B, x_C, x_E, x_P, x_G, F_B, T_R]^\intercal.$$

We call this the **extended state**.

On the other hand, the environment is a constructed framework for defining the RL problem, and essentially the agent only needs a state related to a reward (ref Figure 2.4.1). In the Williams-Otto reactor, the reward function in Equation 4.3.2 can be reduced to a function of the cost function and the constraints. In practice, one can imagine that the agent does not need to know the exact weight fractions if the cost and constraint values can be evaluated. Therefore another alternative state formulation can be a minimal version consisting of the cost and constraints, i.e.,

$$\mathbf{x} = [\Phi_p, G_1, G_2]^\intercal.$$

This approach results in a state vector space of smaller dimension, which we call the **minimal state**. It requires reduced buffer capacity and, in general a smaller problem. Consequently, the agent's computations may be more efficient. However, the dimension reduction may increase the non-linearity of the problem and make it harder to solve.

The three state formulations postulated above were simulated for RL-RTO with DDPG. The objective of the simulations was to reveal sample efficiency for the different formulations. The simulations were run for an increasing number of T, and the deviation between the point of convergence and the plant optimum was calculated corresponding to Figure 6.1.4. Each simulation was run ten times at each setting to calculate an average and the corresponding standard deviation. The total number of training timesteps was increased until $T = 1000$, as this region was of interest to study sample efficiency. The resulting plots for deviation in F_B and T_R are presented in Figure 7.1.1.

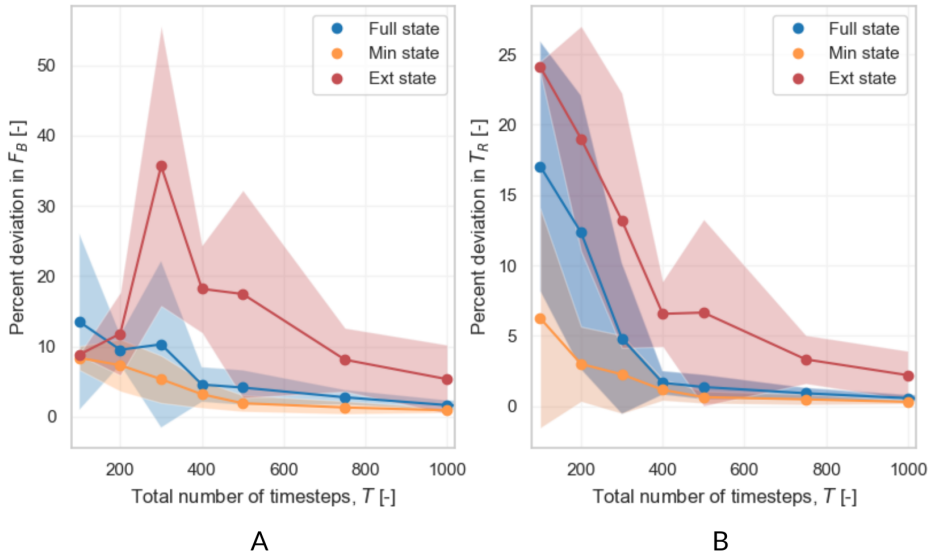


Figure 7.1.1: Average percent deviation from the global plant optimum in A) F_B and B) T_R plotted as a function of total training number of training timesteps for three different state formulations; full, minimal and extended.

The result shows that the extended state formulation performs remarkably worse than the two other formulations. Even at 1 000 training timesteps, the deviations are significant. However, the trend in reducing deviation for training time indicates that this formulation could also solve the problem for a sufficient number of timesteps. The reason why the extended state is less efficient could be due to the increase in the state dimension. The same reasoning can explain the observable improved performance of the minimal state version compared to the full state. The minimal state has smaller deviations for lower numbers of training steps overall, while the deviation is approximately equal to the full state at $T = 1000$.

In conclusion, the results show that there is not one correct MDP state formulation for the reactor and that all proposed versions can work. However, the minimal and full state are more advantageous for sample efficiency. Here the minimal state formulation does have a slightly lower deviation. This is likely directly connected with computational complexity and storage requirements for the agent. However, the difference between the full and minimal state formulations is not outstanding. The differences may be more distinct for larger-scale systems than the Williams-Otto reactor. Considering the need to almost completely eliminate deviation to ensure a feasible point of convergence, T still has to be larger than 1000, even for the minimal state. Therefore, there are likely no significant gains in sample efficiency by using the minimal state instead of the full state.

7.1.2 Dynamic Penalty

The results presented so far enforce the constraints of the Williams-Otto reactor using a linear penalty, LP, terms, as presented in Equation 4.3.2. Table 6.2.3 revealed issues with feasibility at the point of convergence for the RL-RTO. Implementation of dynamic penalties, DP, can possibly improve the neural network approximation accuracy and consequently feasibility, in addition to providing higher sample efficiency. Therefore, the simplified DP scheme presented in Section 3.3.1 was implemented for the RL-RTO and applied to the case study. The values for $\kappa_{1,min}$ and $\kappa_{2,min}$ was set to 10^{-3} and 10^{-5} respectively. The update factors were set to $d_1 = 1.1$ and $d_2 = 2$, due to the reward function increasing in the direction of constraint g_2. A table corresponding to Table 6.2.3 was generated, and the results are presented in Table 7.1.1.

Table 7.1.1: Summary of agent training result for 100 simulations of RL-RTO for varying number of total timesteps using dynamic penalties.

Number of total timesteps, T	**500**	**1000**	**2000**	**5000**
Finding feasible policy/100	19	61	66	70
Average reward at final point (all policies)	63.18	74.16	76.05	76.31
Average reward at final point (feasible policies)	64.01	75.51	76.11	76.35
Average degree of violation	31.5	19.5	17.0	15.0

The number of feasible policies with DP is higher than LP for all total timesteps values, except $T = 500$. Correspondingly, the degree of violation is overall lower than for LP. At $T = 1000$ the improvement is most significant, as the number of feasible points increases by 9 and the degree of violation is reduced by 4.5. This indicates that the DP helps with exploration at the beginning of the training process. However, the difference for higher numbers of total timesteps is not outstanding. One explanation can be that at this level, both κ_i have reached the level of $\kappa_{i,max}$ and the training proceeds the same way for both LP and DP.

Further, observe that the reward at the final point for feasible policies is higher than the average final point of convergence for all policies. This was not the case with LP. It means that the feasible policies have the highest reward and, consequently, the predicted optimum is within the constraint bounds. Thus, one can say that the DP improves the accuracy of the trained neural network approximators.

Overall, the DP improves the degree of violation and sample efficiency slightly compared to the LP, but not to a level where it makes a significant difference for the implementation of the RL-RTO. Yet, DP is a relatively simple feature to implement and should be used for the RL-RTO. A measure that could possibly increase the effect of the DP is to implement the full version that utilizes losses for updating the penalty parameters. However, the conclusion for RL-RTO implementation remains the same: the total number of timesteps required to get a sufficiently low degree of violation is still too large for practical realization in this form.

7.2 Other Reinforcement Learning Algorithms

Another approach to enable real-life implementation of RL-RTO and overcome the model-free RL challenge of sample inefficiency is considering other RL algorithms. One option is to use a batch-online training methodology that can help overcome the convergence to local minima by allowing the RL-RTO algorithm to consider all the data simultaneously [19]. Another option is to use a model-based RL algorithm. These are known to have higher sample efficiency due to the agent sampling from a model instead of directly from the environment.

7.2.1 Batch-Online Reinforcement Learning

DDPG uses experience replay, that stores all the agent's experiences of transitions $(s_t, a_t, r_{t+1}, s_{t+1})$. While learning, the agent samples a mini-batch from the replay buffer to calculate the TD targets and calculates the loss to update the critic (Algorithm 2). Compared to only using the latest experience, using a replay buffer allows for more efficient use of experience since the same experience is used to learn from several times. However, using *batch updates* is a way of using all available experience to produce a new overall increment in the algorithm.

In batch updates, the generation of environment experience can be seen as separated from the training process. Essentially the experience is generated and stored by interacting with the environment first, and then the agent is trained. One limitation with this approach is how to obtain sufficient exploration, as it can only evaluate actions that are already taken. Therefore how the training data is generated plays a key role. The advantages of batch updates are that it produces state estimates with low bias, and it takes full advantage of all samples for training. This is highly relevant for the RTO case due to the challenge with sample efficiency and convergence to a local minimum.

There is one published example [19] of implementing RL-RTO using the batch update method. It reveals promising results using a hybrid training methodology with a gradient-based solver for the critic network (as in DDPG) and a derivative-free optimization method called particle swarm optimization (PSO) to train the actor network. The learning of the critic implements a batch update methodology to consider all training data simultaneously. This approach proves to help the agent with finding the global optimum. The method is called batch-online training. First, it trains the critic on historic data exclusively. Afterwards, the critic is coupled with the actor, that uses the critic to calculate rewards to find the network parameters that maximize the reward. The reader is referred to the full paper for more details [19].

Although the idea of batch training in RL-RTO seems promising, there is one crucial issue with this approach that the cited paper does not address, namely, the use of historical data for training. In the paper, data is generated by simulating the reactor for varying disturbances and parameters, e.g., prices, for a year of operation. This ensures that the training data provides sufficient exploration since the changes in the parameters are managed. As previously mentioned as a common challenge in RL, historical data generally does not provide enough exploration for training. For a controlled environment, e.g., a process plant, we want to keep the operation steady. As a result, historical data usually does not contain large variations. If a

setting not included in the historical data occurs, the trained agent will not know how to react. In addition, the use of historical data would not be applicable for a new environment, e.g., a new plant. However, there is usually significant noise in the measurements for a process plant, and provided that historical prices and other parameters are realistic indicators for future values, implementing RL-RTO using the online-batch methodology with historical data could be possible.

Stable Baselines 3 does not offer batch update methodology for training; therefore, implementing it independently is beyond this project's scope. A suggestion for further work could be to implement an RL-RTO batch-online algorithm for an existing plant/test system, provided access to real historical data and experience with building RL algorithms in the absence of a framework like Stable Baselines 3. Then the performance of the agent trained on real data could be studied.

7.2.2 Model-Based Reinforcement Learning

As mentioned in Section 2.4.7, model-based reinforcement learning is a family of RL algorithms that are known to be data efficient. The model is used to predict the environment's reaction to possible actions in terms of state and reward, followed by planning to compare the predicted rewards (Figure 2.4.4) [4]. Model-free methods are important building blocks for model-based methods. To generate a sample-efficient model, a machine learning tool called Gaussian process (GP) regression could be used. GP is known to perform well even with small data sets for training [45]. Once a model is generated, any planning method, like dynamic programming, can be applied.

Although the fundamental concepts behind model-base RL can seem straightforward, implementing it in practice is another story. As observed for the model-free algorithms, the continuous action and observation spaces in process plants introduce computational challenges. The result is more complex algorithms. For model-based RL this is propagated in two components of the algorithms: model training and planning, as opposed to one in the model-free case. This explains why model-based approaches suffer from being a complex area of research with a high bar for entry, especially from a typical process control background. Applying it to new research areas such as RTO can therefore require expert experience. For this project, an tool provided by Facebook AI called MBRL was tested without success, as the structural design of the framework required excessive modifications to make it applicable in our case [30].

Nevertheless, there might be great potential in model-based RL for RTO. It eliminates the RTO challenge of deducing an accurate model and the model-based RL challenge of sample inefficiency. Experimental data is required to determine if model-based RL can actually outperform existing RTO approaches. Compared with batch-online methods for model-free RL, model-based RL does not require a deep reservoir of historical data. From this perspective, model-based RL can seem more promising, but research required in this field is possibly more comprehensive.

7.3 Final Discussion

The measures tested to improve the sample efficiency of the DDPG RL-RTO by modifying the state vector and implementing DP for the constraints did not lead to any significant improvements. However, DP proved to decrease the degree of constraint violation. Since the implementation is independent of the choice of RL algorithm, the results suggest that this way of implementing constraints is advantageous and should be included in the MDP formulation of the steady-state optimization problem. Nonetheless, using DDPG in RL-RTO with the alternative implementations studied above still does not yet seem to be a realistic approach for potential real-life implementation.

The varying MDP state definitions reveal a challenge with formulating the process plant as Markovian. The lack of a time sequence defined in the problem makes the steady-state optimization problem fundamentally inconsistent with the RL framework. Although the results show that it is possible to solve the steady-state problem using the RL framework, it does not mean that it is a convenient approach. RL performs best to solve complex problems that conventional computation approaches cannot solve. Typically, use of RL is desired when the environment contains unknown parameters and aims to achieve a long-term goal. In RTO, we know the function we want to optimize, there is no long-term goal, and conventional techniques can be applied. Altogether, RL might be an overly complex tool in the case of static RTO. However, the less common dynamic RTO fulfills more of these points since it is a dynamic problem that continuously seeks to compute the plant optimum. In this light, dynamic RTO is likely more applicable for RL.

However, there are properties of RL that address some major challenges in RTO. The need for a process model in conventional RTO approaches is a common problem, especially because RTO faces convergence problems in the case of structural plant-model mismatch. An accurate process model deduced through first principles can be challenging to obtain, and in complex plants with many disturbances, it may be infeasible in practice. MA overcomes this challenge by using correction terms, but the presence of noise introduces convergence problems due to inaccurate gradients. Therefore, eliminating the need for a model in RTO can be profitable. It may be even more valuable if the scheme is adaptive so that changes in model parameters can be handled automatically. RL possesses this property as observed in the simulated step-change to the prices. The idea of a model-free and adaptive approach is appealing, and further research on how RL-RTO can be implemented with other algorithms than DDPG can be pursued.

The proposed alternatives to DDPG, batch-online RL and model-free RL algorithms, can both be promising in use for static RTO purposes. The main reason is that they do not face the challenge of overcoming sample inefficiency. However, there are limitations to both approaches. The need for historic data for batch-online RL anticipates that the plant is not new and that operation data is accessible. The performance of a batch RL-RTO relies highly on the data, and data sets must have enough variations, which generally cannot be assumed. Model-based RL does not rely on historical data, but the implementation is more complex than model-free cases, and expert RL knowledge is essential. Further research that could be promising

would include finding a way to implement a model-based RL algorithm on a steady-state RTO problem and testing various model-free algorithms with batch-online methodology on actual historical data.

Programming tools like Stable Baselines 3 are important resources for lowering the entry bar and enabling more people to learn about RL quickly. However, deep RL is a complex field that is constantly evolving. The fact that the field is still very new implies that there is no complete high-level overview, and important knowledge can be found in papers or lectures [31]. Therefore, tools like Stable Baselines 3 work perfectly as an introduction to deep RL. Unfortunately, the framework also has implementation limitations that do not make it functional in research. For this project, it was observed that the framework is not agile enough to enable the implementation of methods like batch-online and full dynamic penalties. For improving the RL-RTO, coding the algorithms from scratch is vital to provide a more agile algorithm that can be tailored for desired purposes.

Finally, there is an additional alternative to RL-RTO that can solve RTO's problems with structural plant-model mismatch and MA's possible computational issues. One can combine either of the conventional RTO approaches with a machine learning tool to overcome the problems. One example could be to use Gaussian processes in MA to calculate the modifiers. This has proved to solve MA's issue with noise sensitivity since it results in more robust gradient approximations (full paper [8]). Compared with this implementation, RL-RTO's only additional feature is the adaptive property. It is debatable whether this provides sufficient grounds to argue that RL-RTO should be a subject of further research.

Chapter 8

Conclusion

This project has studied the use of the model-free deep reinforcement learning (RL) algorithm deep deterministic policy gradient (DDPG) as an optimization method for static real-time optimization (RTO). In addition, the potential and challenges related to real-life implementation were debated. The first finding was that the RL-RTO with DDPG solves the RTO problem for the Williams-Otto reactor, which proves that the use of RL in RTO is an actual possibility. Compared to the standard modifier adaptation (MA), the trained DDPG agent is more efficient in optimizing and solves the problem in fewer iterations. Another advantage of RL-RTO over MA is the ability to handle measurement noise, where the gradient approximation in MA has a high sensitivity to noise. The adaptive feature of RL was illustrated with the change in prices, where the agent was trained before and after the change. The agent adapted automatically, whereas a general RTO approach would need a new model. Even though MA can handle some structural plant-model mismatch, significant changes in disturbance would influence the performance. In total, the three major advantages of RL-RTO are efficient optimization in the presence of noise, model-free method and adaptive properties.

However, the result also revealed several challenges with the RL-RTO. The first major challenge observed was sample inefficiency. The deviation from the optimum was significant for $T < 1000$, but required many additional timesteps to reduce it. This was substantiated by results from the degree of constraint violation, highlighting the trade-off between the degree of violation and training time. The observed degree of violation of $T > 15.5$ for all simulated T would not be acceptable for implementation in a real plant. In addition, the rewards indicated that the critic is an imperfect replacement for the reward function. Replacing linear penalty (LP) terms with dynamic penalty (DP) terms proved to decrease the degree of violation and improve the critic approximation. This supports the use of dynamic penalties as constraint implementation method in RL. However, the use of DP did not improve the sample efficiency or degree of violation. Further, the different modifications to the Markov state formulations, with an extended and a minimal state, showed that they were all compatible with the Markov decision process (MDP) framework. Yet, the extended state performed significantly worse than the standard full formulation, and the minimal state performed slightly better. Still, the

sample inefficiency remained a challenge.

As alternative RL approaches to DDPG, the use of batch-online and model-based RL can be promising since sample efficiency is not an issue. The main issue for batch-online RL is the need for training historical data with enough variations. Model-based RL is a complex area of research, requiring extensive knowledge for implementation. The lack of experimental results makes the discussion hypothetical, but based on previous research, further work for developing a realistic RL-RTO was suggested. Testing the batch-online method with various model-free algorithms on actual historical data is a suggested path for further research. For model-based RL, further work should include how to solve the steady-state RTO problem. In addition, the results should be compared to conventional RTO approaches in combination with machine learning tools, such as MA with Gaussian processes.

Although the results from RL-RTO using DDPG indicated that the approach is not feasible for real-life implementation in a process plant, it holds a promise that RL as a method for optimization can be used in RTO. Deep RL is a complex area of research that requires expert knowledge for implementation in new fields. Even though RL sounds very attractive with superhuman powers, in reality, RTO does not generally require superhuman powers. However, there is potential for RL-RTO, and it is an exciting field to explore. Still, a considerable amount of work remains to determine if RL is realistic for real-life implementation. At the moment, an easier implementation for overcoming RTO challenges would therefore likely be to use conventional RTO in combination with deep learning.

Bibliography

[1] D. E. Seborg, T. F. Edgar, D. A. Mellichamp, and F. J. D. III, *Process Dynamics and Control*, 4th ed. Wiley, 2016.

[2] M. L. Darby, M. Nikolaou, J. Jones, and D. Nicholson, "Rto: An overview and assessment of current practice," *Journal of Process Control*, vol. 21, no. 6, pp. 874–884, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0959152411000540

[3] D. Silver, "Lecture 1: Introduction to Reinforcement Learning," https://youtu.be/2pWv7GOvuf0, 2015, accessed on 11.01.2022

[4] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018.

[5] A. Marchetti, B. Chachaut, and D. Bonvin, "Modifier-adaptation methodology for real-time optimization," *Industrial & Engineering Chemistry Research*, vol. 48, no. 13, pp. 6022–6033, 2009. [Online]. Available: http://infoscience.epfl.ch/record/128111

[6] A. G. Marchetti, G. François, T. Faulwasser, and D. Bonvin, "Modifier adaptation for real-time optimization—methods and applications," *Processes*, vol. 4, no. 4, 2016. [Online]. Available: https://www.mdpi.com/2227-9717/4/4/55

[7] A. Marchetti, "Modifier-adaptation methodology for real-time optimization," Ph.D. dissertation, EPFL, 2009, pages 10–11, 25–42. [Online]. Available: https://www.researchgate.net/publication/40754964_Modifier-adaptation_methodology_for_real-time_optimization

[8] T. d. A. Ferreira, H. A. Shukla, T. Faulwasser, C. N. Jones, and D. Bonvin, "Real-time optimization of uncertain process systems via modifier adaptation and gaussian processes," in *2018 European Control Conference (ECC)*, 2018, pp. 465–470. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8550397

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013. [Online]. Available: https://arxiv.org/abs/1312.5602

[10] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning." in *ICLR*, Y. Bengio

and Y. LeCun, Eds., 2016. [Online]. Available: http://dblp.uni-trier.de/db/conf/iclr/iclr2016.html#LillicrapHPHETS15

[11] N. P. Lawrence, M. G. Forbes, P. D. Loewen, D. G. McClement, J. U. Backström, and R. B. Gopaluni, "Deep reinforcement learning with shallow controllers: An experimental application to PID tuning," *Control Engineering Practice*, vol. 121, p. 105046, 2022. [Online]. Available: https://doi.org/10.1016%2Fj.conengprac.2021.105046

[12] S. Spielberg, A. Tulsyan, N. P. Lawrence, P. D. Loewen, and R. B. Gopaluni, "Toward self-driving processes: A deep reinforcement learning approach to control," *AIChE Journal*, vol. 65, no. 10, p. e16689, 2019. [Online]. Available: https://aiche.onlinelibrary.wiley.com/doi/abs/10.1002/aic.16689

[13] R. K. Sinnot, *Chemical Engineering Design*, 5th ed. Elsevier Science & Techonolgy, 2009, p. 306 -317

[14] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html

[15] T. Marlin, *Process Control*, 2nd ed. Boston: McGraw-Hill, 2000, p. 4-10.

[16] P. J. Woolf, *Chemical Process Dynamics and Controls.* University of Michigan, 2011, p. 32-48. [Online]. Available: https://archive.org/details/ChemicalProcessDynamicsAndControls/page/n31/mode/2up

[17] S. Skogestad, *Plantwide control.* Wiley, 2012, pp. 229–249.

[18] A. D. Quelhas, N. J. C. de Jesus, and J. C. Pinto, "Common vulnerabilities of rto implementations in real chemical processes," *The Canadian Journal of Chemical Engineering*, vol. 91, no. 4, pp. 652–668, 2013. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cjce.21738

[19] B. K. M. Powell, D. Machalek, and T. Quah, "Real-time optimization using reinforcement learning," *Computers & Chemical Engineering*, vol. 143, p. 107077, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0098135420305500

[20] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org

[21] S. Up, "Part 1: Key concepts in rl," 2018. [Online]. Available: https://spinningup.openai.com/en/latest/spinningup/rl_intro.html#part-1-key-concepts-in-rl

[22] D. Silver, "Lecture 2: Markov Decision Processes," https://youtu.be/lfHX2hHRMVQ, 2015, accessed on 11.01.2022

[23] TensorFlow, "Introduction to RL and Deep Q Networks," https://www.tensorflow.org/agents/tutorials/0_intro_rl, 2021, accessed on 12.02.2022

[24] D. Silver, "Lecture 6: Value Function Approximation," https://youtu.be/UoPei5o4fps, 2015, accessed on 28.01.2022

[25] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *Proceedings of the 31st International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, E. P. Xing and T. Jebara, Eds., vol. 32, no. 1. Bejing, China: PMLR, 22–24 Jun 2014, pp. 387–395. [Online]. Available: https://proceedings.mlr.press/v32/silver14.html

[26] D. Silver, "Lecture 7: Policy Gradient Methods," https://youtu.be/KHZVXao4qXs, 2015, accessed on 02.02.2022

[27] M. Kiran and B. M. Ozyildirim, "Hyperparameter tuning for deep reinforcement learning applications," *CoRR*, vol. abs/2201.11182, 2022. [Online]. Available: https://arxiv.org/abs/2201.11182

[28] R. Liessner, J. Schmitt, A. Dietermann, and B. Bäker, "Hyperparameter optimization for deep reinforcement learning in vehicle energy management." in *ICAART (2)*, 2019, pp. 134–144. [Online]. Available: https://www.researchgate.net/publication/331398889_Hyperparameter_Optimization_for_Deep_Reinforcement_Learning_in_Vehicle_Energy_Management

[29] N. Keskar, J. Nocedal, P. Tang, D. Mudigere, and M. Smelyanskiy, "On large-batch training for deep learning: Generalization gap and sharp minima," 2016, paper presented at ICLR 2017. [Online]. Available: https://arxiv.org/abs/1609.04836

[30] L. Pineda, B. Amos, A. Zhang, N. O. Lambert, and R. Calandra, "Mbrl-lib: A modular library for model-based reinforcement learning," 2021. [Online]. Available: https://arxiv.org/abs/2104.10159

[31] J. Achiam, "Spinning Up in Deep Reinforcement Learning," 2018.

[32] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016. [Online]. Available: https://arxiv.org/abs/1606.01540

[33] G. Matheron, N. Perrin, and O. Sigaud, "The problem with DDPG: understanding failures in deterministic environments with sparse rewards." [Online]. Available: https://arxiv.org/abs/1911.11679

[34] H. Yoo, V. M. Zavala, and J. H. Lee, "A dynamic penalty function approach for constraint-handling in reinforcement learning," *IFAC-PapersOnLine*, vol. 54, no. 3, pp. 487–491, 2021, 16th IFAC Symposium on Advanced Control of Chemical Processes ADCHEM 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2405896321010624

[35] T. J. Williams and R. E. Otto, "A generalized chemical processing model for the investigation of computer control," *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, vol. 79, no. 5, pp. 458–473, 1960.

[36] Y. Zhang and J. Forbes, "Extended design cost: A performance criterion for real-time optimization systems," *Computers & Chemical Engineering*, vol. 24, pp. 1829–1841, 09 2000.

[37] S. K. Morten Helbæk, *Fysikalsk kjemi.* Fagbokforlaget, 2006.

[38] J. F. Forbes, "Model structure and adjustable parameter selection for operations optimization," Ph.D. dissertation, McMaster University, 1994. [Online]. Available: https://macsphere.mcmaster.ca/handle/11375/7726

[39] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "CasADi – A software framework for nonlinear optimization and optimal control," *Mathematical Programming Computation*, vol. 11, no. 1, pp. 1–36, 2019. [Online]. Available: https://doi.org/10.1007/s12532-018-0139-4

[40] A. Wächter and L. Biegler., "Scikit-learn: Machine learning in Python," *Mathematical Programming*, 2006. [Online]. Available: http://www.optimization-online.org/DB_HTML/2004/03/836.html

[41] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[42] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020. [Online]. Available: https://doi.org/10.1038/s41592-019-0686-2

[43] F. B. Konow, "Deep reinforcement learning in real-time optimization," 2022. [Online]. Available: https://github.com/fridabko/Deep-Reinforcement-Learning-in-Real-Time-Optimization

[44] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," 2019. [Online]. Available: https://arxiv.org/abs/1907.10902

[45] C. Rasmussen and C. K. I. Williams, *Gaussian Processes in Machine Learning.* MIT Press, 2006. [Online]. Available: https://gaussianprocess.org/gpml/

[46] "The finite difference method," https://www.ljll.math.upmc.fr/frey/cours/UdC/ma691/ma691_ch6.pdf, accessed 08.05.2022

[47] E. W. Weisstein, "Taylor Series.From MathWorld–A Wolfram Web Resource," https://mathworld.wolfram.com/TaylorSeries.html, accessed 09.02.2022

[48] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 2018, pp. 1587–1596. [Online]. Available: https://proceedings.mlr.press/v80/fujimoto18a.html

[49] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," 2018. [Online]. Available: https://arxiv.org/abs/1801.01290

Appendix A

Gradient Approximation - Finite Differences

A simple and basic method for estimating gradients and solving differential equations is finite differences [46].

At a given point x, the derivative of a function f is defined as

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{A.0.1}$$

By the definition of a mathematical limit, the quotient produce an estimation of the gradient for small nonzero constant h. A gradient approximation can therefore be written as

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}. \tag{A.0.2}$$

The accuracy of the approximation is increases with decreasing h. There are several possible expressions for finite difference gradient approximation. Equation A.0.2 is called a forward difference when $h > 0$, since this entails a perturbation in the positive x-direction. When $h < 0$, the approximation is called a backward difference. So-called central difference methods use both $f(x+h)$ and $f(x-h)$ to compute an approximation for $f'(x)$.

The error of a finite difference approximation can be derived using Taylor series, by neglecting higher order terms [47]. It can be proved that by neglecting all terms of order ≥ 2 the error can be written $O(h)$. $O(h)$ indicates that the error is proportional to h. Therefore the approximation is referred to as a first-order approximation [46]. The exact gradient with associated error term can be written as follows

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h). \tag{A.0.3}$$

Appendix B

Algorithms

B.1 Standard Modifier Adaptation

A pseudo code is presented to describe the algorithm that is used to simulate the standard MA for optimizing the Williams-Otto reactor.

Algorithm 1 Standard Modifier Adaptation

Step 0 - Initialization
Initialize u_0 and maximum number of iterations n_k
Initialize parameters for the input $k_1 = k_2$ and for the modifiers $a_i = b_i = c, i \in \{1, 2\}$
Initialize modifiers $\epsilon_{i,0} = 0,\ \boldsymbol{\lambda}_0^{g_i} = \boldsymbol{\lambda}_0^{\Phi} = \mathbf{0}$
Calculate u_0^* by optimizing Equation 4.2.8 and the initial modifiers from Equation 3.2.6

for $k = 0$ to $k = n_k$ **do**
Step 1 - Plant evaluation
Evaluate plant (Equation 4.2.7) and process model (Equation 4.2.8) at u_k to obtain:
Measurements and model states, x_p and x
Cost and constraints, Φ_p, $g_{p,1}$, $g_{p,2}$, Φ, g_1 and g_2
Step 2 - Gradient calculation
Calculate plant gradients of cost and constraints with finite differences (Appendix A)
Calculate model gradients of cost and constraints by differentiation
Step 3 - Modifier calculation
Calculate the modifiers $\epsilon_{i,k}, \boldsymbol{\lambda}_k^{g_i}, \boldsymbol{\lambda}_k^{\Phi}$ from Equations 3.2.2, 3.2.4 and 3.2.5.
Step 4 - Filter modifiers
Filter $\epsilon_{i,k}, \boldsymbol{\lambda}_k^{G_i}, \boldsymbol{\lambda}_k^{\Phi}$ using Equations 3.2.8, 3.2.9 and 3.2.10
Step 5 - Optimization
Optimize Equation 3.2.6 s.t. 4.3.1 and Equation 4.2.8 to obtain u_{k+1}^*
Step 6 - Filter input
Filter u_{k+1}^* using Equation 3.2.7
Step 7 - Convergence criteria
if $|u_{k+1}^* - u_k^*| < 10^{-5}$ **then**
Break for loop
end if
end for

B.2 Deep Deterministic Policy Gradient

A pseudo code for the DDPG algorithm is presented. The reader is referred to the full paper in[10]. For implementation of DDPG in RL-RTO on the Williams-Otto reactor see Sections C and D.

Algorithm 2 DDPG ALgorithm[10]

Step 0 - Initialization
Initialize critic network $Q(s_t, a_t|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ randomly with weights θ^Q and θ^μ
Initialize target network Q' and μ' with weights $\theta^{Q'} \longleftarrow \theta^Q$, $\theta^{\mu'} \longleftarrow \theta^\mu$
Initialize replay buffer D
Initialize the noise level for the measurements, σ
Initialize the noise level for the actions, σ_{a_t}

for $t = 1$ **to** $t = T$ **do**
Step 1 - Initialization episode
Initialize a random process $\mathcal{N}$ with σ_{a_t} for action exploration
Receive initial observation state s_1

for $t = 1$ **to** convergence **do**
Step 2 - Episode
Observe state s_t
Select action $a_t = \mu(s_t|\theta^\mu)$ according to the μ and σ_a
Execute action a_t in the environment
Observe the new state s_{t+1} and the associated reward r_t
Store (s_t, a_t, r_t, s_{t+1}) in D
Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from D
Calculate the TD targets for the minibatch using Equation 2.4.27
Update critic by minimizing the batch loss according to Equation 2.4.28
Update the actor policy by maximizing the batch policy gradient in Equation 2.4.29
Update the target networks according to Equations 2.4.30 and 2.4.31

Step 3 - Convergence criteria
if $|x^*_{k+1} - x^*_k| < 10^{-5}$ **then**
Reset environment
Break for loop
end if
end for
end for

Appendix C

Open AI Gym Environment: Williams-Otto Reactor

To apply Stable Baselines RL algorithms[14] to a given problem, its environment has to be expressed as a custom environment whose structure is compatible with the OpenAI Gym interface[32]. Here the code for generating an environment for the Williams-Otto reactor is presented step by step alongside descriptions of each function. The gym environment is defined as a class object and inherits from the OpenAI Gym Class Env. It has three key components; the initialization of the class, a step function, a reset function. The initialization defines the environment in terms of environment parameters and the action and observation space. Given an action, the step function corresponds according to the environment and generates a state, reward and indicator of goal achieved. In addition the environment class can have a render function that displays the state of the environment, typically with a plot or visualization. The overall structure of the custom Williams-Otto reactor Gym environment can be written as follows:

```
import gym
from gym import spaces

class WOreactorEnv(gym.Env):

    def __init__(self, arg):
        # Initialize environment parameters
        # Define action and observation space

    def step(self, action):
        # Execute action and return observation and reward

    def reset(self):
        # Reset the environment

    def render(self, mode='human'):
        # Display the state of the environment
```

In what follows, each of the functions used to implement the Williams-Otto system will be presented. In the initialization the action and observation spaces are specified using `gym.spaces` objects. For continuous spaces the `space.Box(low,high)` function is used, where the inputs `low` and`high` set the numerical boundaries of the spaces. Stable Baselines3 emphasizes that normalization of the action and observation is crucial for their DDPG implementation. Therefore both the observation space and action space have lower bounds set to -1 and upper bounds set to 1. Accessing the true action and observation values are done via scaling simple functions. For this example we show the full state vector formulation from the MDP 4.3.1. The only parameter that is necessary in the environment is the maximum number of timesteps per episode, which is initialized. The code for the initialization is given below.

```
def __init__(self):
    super(WOreactorEnv, self).__init__()
    self.maxSteps = maxSteps

    # Define the action space: [FB,TR]
    highAction = np.array([1,1], dtype=np.float32)

    self.action_space = spaces.Box(low=-highAction, high=highAction,
    dtype=np.float32)

    # Define the observation space: [xa, xb, xc, xe, xp, xg]
    highObs = np.array([1.0, 1.0, 1.0, 1.0, 1.0, 1.0],
    dtype=np.float32)
    self.observation_space = spaces.Box(low=-highObs, high=highObs,
    dtype=np.float32)
```

The step function defines the behaviour of the environment. It executes one time step in the environment given an input action and returns `state`, `reward`, `done` and `info`. For the Williams-Otto reactor each step corresponds to one iteration in solving the optimization problem. First the denormalized action is calculated in `dnormAction(action)`. Given the input, the new state is then obtained by solving the plant equations in 4.2.7 in the `X(dnAction)` function, which uses a root solver provided by scipy. Since the state is assumed available through measurements in the reactor, a random normally distributed noise is added to each states. Further the `self.state` variable is updated by normalizing the noise-exposed states in `normObs(stateVec)`. Further, the cost of the state is calculated through the function `J(dnAction,stateVec)` (Φ_p in Equation 4.3.1). Then the LP terms are added to the cost if any of the two constraints are violated. Finally, the `done` parameter checks if the episode is completed: this can be translated to if the algorithm has converged. Convergence is defined as reached if the change in each of the states is less than 10^{-5}. Then `done` is set to `true`. Alternatively, `done=true` ifn the maximum number of timesteps are reached. In the end the step function returns all the updated parameters; state, costs, done and an empty{}, as no supplementary environment info is necessary. The full code is displayed below.

```
def step(self, action):
    self.t += 1
    self.lastState = self.state
    dnAction = denormAction(action)
```

```
    stateVec = X(dnAction)

    # Add noise measurements
    for i in range(len(stateVec)):
      stateVec[i] += np.random.normal(0,noise, size=(1,1))[0][0]
      if stateVec[i]<0:
        stateVec[i]=0
      elif stateVec[i]>1:
        stateVec[i]=1

    # Update state
    self.state = normObs(stateVec)

    # Calculate cost
    costs = J(dnAction,stateVec)

    # Constraints
    if stateVec[0]>0.12: #penalty term, tune later
      costs -= 410*abs(stateVec[0]-0.12)
    if stateVec[-1]>0.08:
      costs -= 2000*abs(stateVec[-1]-0.08)

    # Checking for convergence
    percentChange = []
    for i in range(len(self.lastState)):
      pCi = (self.lastState[i]-self.state[i])/self.state[i]
      percentChange.append(bool(pCi<10e-5))
    done = bool(False not in percentChange)
    if self.t>=self.maxSteps:
        done = bool(self.t>=self.maxSteps)

    return self.state, costs, done, {}
```

The `reset` function presented below, is called whenever an episode is complete and the `done` parameter is true. Here a new initial action is randomly selected within the feasible region. The initial action is used to calculate the corresponding state, which is normalized before it is returned. The class variables `t` and `lastState` are reset to `0` and `None` respectively.

```
def reset(self):
    u0FB = random.uniform(5.5,7)
    u0TR = random.uniform(75,86)
    u0 = np.array([u0FB,u0TR])
    x0 = X(u0)
    self.state = normObs(x0)
    self.t = 0
    self.lastState = None
    return self.state
```

Render is used for visualization and in the Williams-Otto reactor it plots the simulations of the episode to study if the agent succeeds to converge to the plant optimum. Since it only contains code for plotting it is not concerned necessary to include the code.

Appendix D

Stable Baselines 3: DDPG for Williams-Otto reactor

The code for solving the Williams-Otto reactor with DDPG using Stable Baselines 3[14], is presented below. It uses the Open AI Gym environment `WOreactorEnv` from Appendix C is imported. The total number of training timesteps T is set to 5 000 and the action noise is set to the tuned value from Table 6.1.1. `log_interval` is the number of timesteps before logging, and `n_eval_episodes` is the number of episode to evaluate the agent. The simulation is run until the parameter `done=true`, which indicates the point of convergence. Examples of the fundamental code files that were used in the project can be found in[43].

```
import gym
import numpy as np
from WOreactorEnv import WOreactorEnv
from stable_baselines.ddpg.policies import MlpPolicy
from stable_baselines.common.noise import NormalActionNoise
from stable_baselines import DDPG

env = gym.make('WOreactorEnv')
T = 5 000 #total numbe rof training timesteps

# the noise objects for DDPG
n_actions = env.action_space.shape[-1]
action_noise = NormalActionNoise(mean=np.zeros(n_actions), sigma=0.09537 * np.
    ones(n_actions))

model = DDPG(MlpPolicy, env, verbose=1, action_noise=action_noise)
model.learn(total_timesteps=T, log_interval=10, n_eval_episodes=2)

obs = env.reset()
while not done:
    action, _states = model.predict(obs)
    obs, rewards, done, info = env.step(action)
    env.render()
```

Appendix E

Other Model-Free Algorithms

Stable Baselines 3 provide three different predefined algorithms that are compatible with continuous state and action spaces[14]. The algorithms are DDPG, TD3 and SAC. Here results from simulation of trained agents for TD3 and SAC on the Williams-Otto reactor are presented, corresponding to Figure 6.2.2. Total number of training steps were set to $T = 5\,000$ and 10 simulations were run to calculate the mean and standard deviation.

E.1 TD3

The Twin Delayed Deep Deterministic policy gradient, or TD3, algorithm is a descendant of DDPG. The algorithm address function approximation error in DDPG, where the Q-values are overestimated. The reader is referred to the full paper[48]. Table E.1.1 displays the point of convergence with the associated profit and reward. It shows that the deviation from the optimum is larger than in DDPG (Table 6.2.2). Figure E.1.1 presents the iteration simulations if the TD3 RL-RTO. From observations it looks very similar to the DDPG simulation in Figure 6.2.2.

Table E.1.1: The optimal values of inputs, profit, reward and constrained states from simulation of the RL-RTO using TD3.

Parameter	F_B^* [kg/s]	T_R^* [°C]	Φ_p^* [\$/s]	R^* [\$/s]	x_A^* [−]	x_G^* [−]
Value	4.4189	80.69	75.74	76.28	0.1188	0.0800

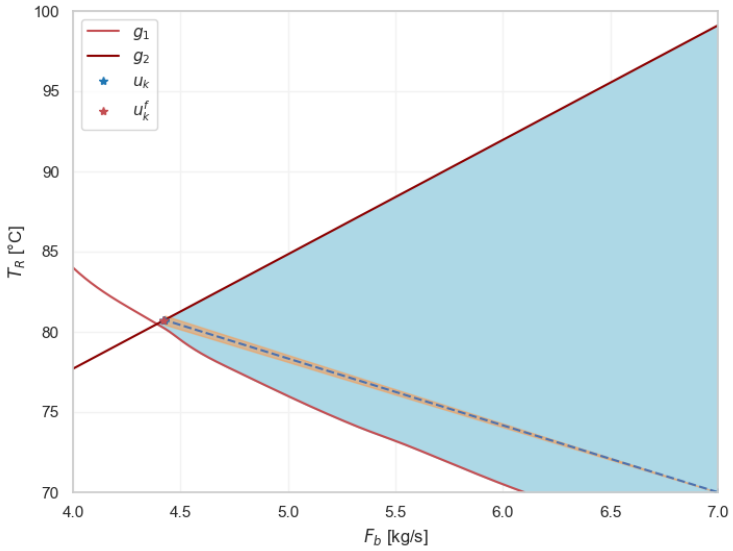


Figure E.1.1: Plot of $[F_B, T_R]$ and standard deviations for iterations of the simulated RL-RTO using TD3.

E.2 SAC

Soft Actor Critic (SAC) is an algorithm that optimizes a stochastic policy off-policy, and combines stochastic policy optimization and DDPG-style approaches. The reader is referred to the full paper[49]. Table E.2.1 displays the point of convergence with the associated profit and reward. It shows that the deviation from the optimum is significantly larger than in both DDPG (Table 6.2.2) and TD3 (Table E.1.1). Figure E.2.1 presents the iteration simulations if the TD3 RL-RTO. It is obvious that the point of convergence deviates from the plant optimum and that constraint g_1 is not active.

Table E.2.1: The optimal values of inputs, profit, reward and constrained states from simulation of the RL-RTO using SAC.

Parameter	F_B^* [kg/s]	T_R^* [°C]	Φ_p^* [\$/s]	R^* [\$/s]	x_A^* [−]	x_G^* [−]
Value	4.4712	81.0	75.46	76.36	0.1166	0.0799

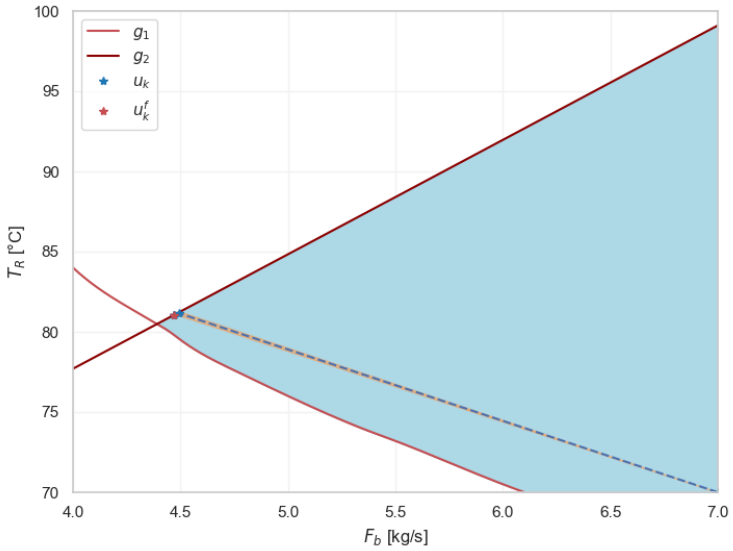
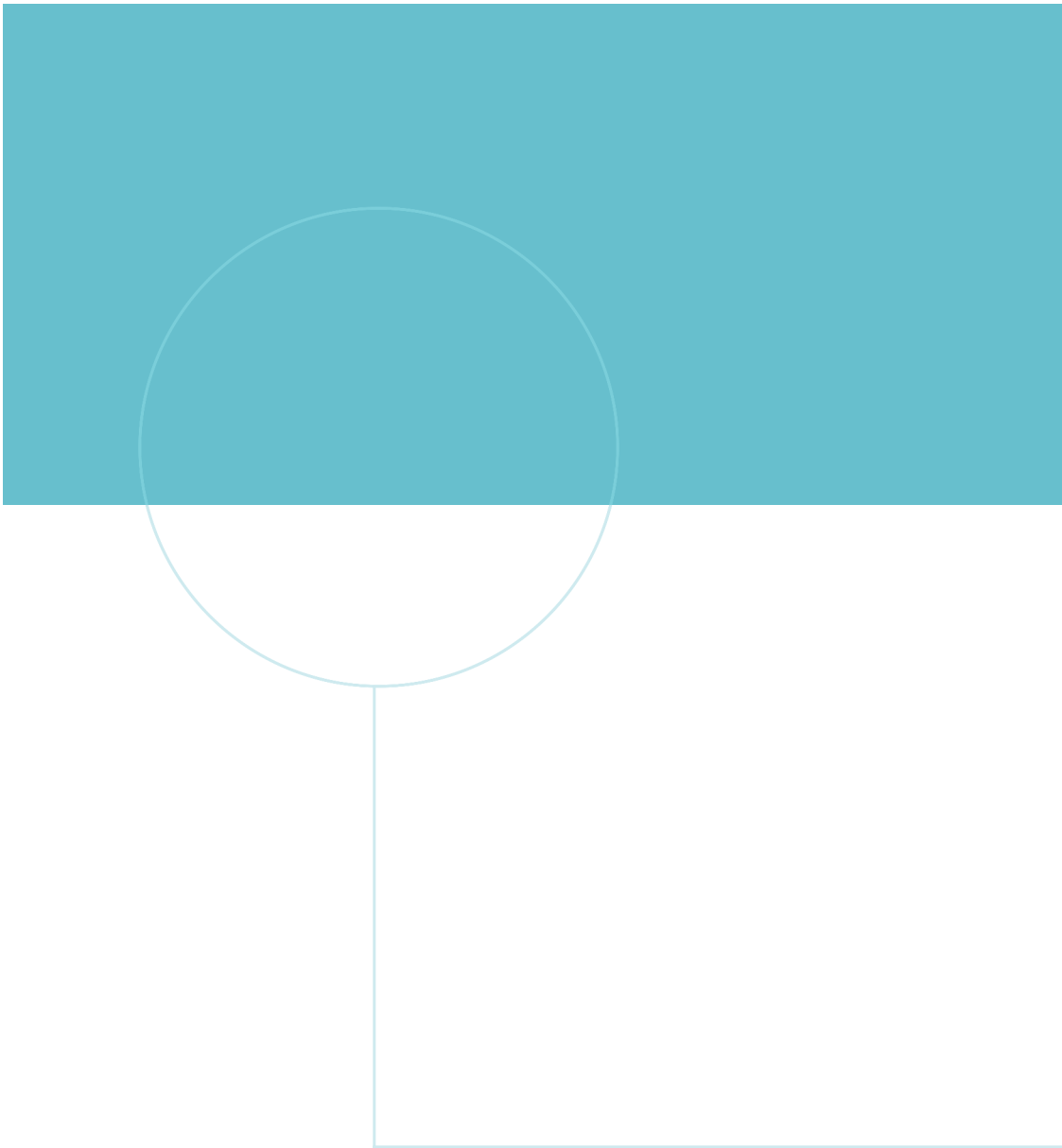


Figure E.2.1: Plot of $[F_B, T_R]$ and standard deviations for iterations of the simulated RL-RTO using SAC.



NTNU
Norwegian University of
Science and Technology

THE UNIVERSITY
OF BRITISH COLUMBIA