2020•2021

Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: chemie

Masterthesis

Application of Reinforcement Learning for continuous stirred tank reactor (CSTR) temperature control

PROMOTOR :

Prof. dr. ir. Mumin Enis LEBLEBICI

PROMOTOR :

ir. Min WU

Jelco Hendrikx

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: chemie

Gezamenlijke opleiding UHasselt en KU Leuven

2020•2021

Faculteit Industriële Ingenieurswetenschappen

master in de industriële wetenschappen: chemie

Masterthesis

Application of Reinforcement Learning for continuous stirred tank reactor (CSTR) temperature control

PROMOTOR :

Prof. dr. ir. Mumin Enis LEBLEBICI

PROMOTOR :

ir. Min WU

Jelco Hendrikx

Scriptie ingediend tot het behalen van de graad van master in de industriële wetenschappen: chemie

UHASSELT KU LEUVEN

Preface

I have always been sure that process technology is my thing and this Master's thesis gave me the opportunity to combine my educational knowledge with my interest into programming and artificial intelligence. From the beginning it was clear that this subject would not be an easy subject and I have experienced that over the past few months. After a difficult start and a number of unexpected problems, I succeeded in producing a well-founded study. I hope this contributes to bringing chemistry closer together with computer programming.

I would like to thank my promotor, ir. Min Wu (KU Leuven), who has provided me with continuous advice and daily support during the period of this Master's thesis. Special thanks to Prof. dr. ir. Mumin Enis Leblebici (KU Leuven) and Prof. dr. Jeroen Lievens (KU Leuven) for their help, guidance and information.

Table of contents

List of tables

List of figures

Glossary

CIPT	Center for Industrial Process Technology
AI	Artificial Intelligence
RL	Reinforcement Learning
SAC	Soft Actor-Critic
CSTR	Continuous Stirred Tank Reactor
ML	Machine Learning
PID	Proportional Integral Derivative
MPC	Model Predictive Control
ANN	Artificial Neural Network
PFR	Plug Flow Reactor
MDP	Markov Decision Process
ASE	Associate Search Element
ACE	Adaptive Control Element
IDE	Integrated Development Environment
SB3	Stable Baselines 3

Abstract

The Center for Industrial Process Technology (CIPT) is a research group of KU Leuven carrying out research about applications of Artificial Intelligence (AI) for chemical process control. In recent years, there has been a strong increase in using machine learning in terms of Reinforcement Learning (RL) for automation tasks. In RL, an agent learns what action to take based on a given state in order to reach the best performance of the task by interacting with its environment. This principle is already successfully applied in chemical process control for maximizing the concentration of a product. A state-of-the-art RL algorithm is Soft Actor-Critic (SAC) that has been proven to outperform other RL algorithms. Based on this, it is hypothesized that SAC will also outperform the previously used RL algorithms in chemical process control. In this Master's thesis, a SAC RL algorithm is used for controlling the reactor temperature and concentration of cyclopentenol of the Van de Vusse reaction taking place inside a CSTR. The control inputs/actions for SAC are the heat removal value and the cooling jacket temperature. The performance of SAC is analyzed based on training steps, different time steps and stability. After training the SAC, the control structure was able to control the process within a temperature range of 375 ± 0.05 Kelvin or a concentration range of 1.10 ± 0.05 kmol/m^3. In the end, recommendations for future work are described.

Abstract

Het centrum voor industriële proces technologie (CIPT) is een onderzoeksgroep binnen KU Leuven waarin men onderzoek doet naar toepassingen van artificiële intelligentie in chemische proces controle. De laatste jaren is er een sterke toename in het gebruik van *machine learning* in de vorm van *reinforcement learning (RL)* voor geautomatiseerde taken. In *RL* leert een *agent*, in een bepaalde toestand, welke actie leidt tot de beste prestatie van een taak door te interageren met zijn omgeving. Dit principe is al succesvol toegepast in chemische proces controle voor het controleren van de concentratie van een product. Een nieuw *RL* algoritme is *Soft Actor-Critic (SAC)* dat al reeds bewezen heeft dat het beter presteert dan andere algoritmes. Op basis hiervan wordt er verondersteld dat *SAC* ook in chemische proces controle beter zal presteren dan de huidig gebruikte *RL* algoritmen. In deze Master thesis wordt *SAC* gebruikt voor het controleren van de temperatuur en de concentratie aan *cyclopentenol* in de Van de Vusse reactie die plaatsvindt in een *CSTR*. De controleerbare ingangsparameters voor *SAC* zijn de warmte verwijderende waarde en de temperatuur van de koelingsmantel. De prestatie van *SAC* wordt nagegaan op basis van de trainingsstappen, tijdsintervallen en de stabiliteit. Na het trainen van *SAC*, is deze controle vorm in staat om het proces te regelen binnen een temperatuursinterval van 375 ± 0.05 K of een concentratie-interval van 1.10 ±0.05 kmol/m^3. Er wordt afgesloten met aanbevelingen voor toekomstig werk.

1 Introduction

Artificial Intelligence (AI) is a term used for describing the capability of machines to think, learn and solve problems like humans would do. These machines aim to achieve a specific goal by taking actions with the highest chance of actually reaching that goal. Machine Learning (ML) is the part of AI that studies computer algorithms that can improve themselves based on data. During the last years, disadvantages of large data such as handling, transport and storage have been reduced along with the distribution of powerful computers around the world. This has led to an increased interest from different sectors in using ML for various applications. Within ML, there are three subcategories: Supervised Learning, Unsupervised Learning and Reinforcement Learning. The algorithms in Supervised Learning can improve itself by learning out of data that already contains the information about output values that are associated with specific input values. The algorithms in Unsupervised Learning aim to find this correlation in unlabeled data. Reinforcement Learning (RL) differs from the previous two in such a way that in RL, an agent learns what action to take given a state in order to perform the given task in the best possible way. The agent does this by interacting with its environment in which it performs actions and from which it receives states and rewards [1].

This principle has proved to be effective in various real-life applications. RL is used for scheduling traffic signals in multi-intersection vehicular networks. A five-intersection traffic network was studied and each intersection was controlled by an agent. This application proved that RL outperformed the existing single-intersection control in terms of average delay, congestion and intersection cross-blocking [2]. In the field of Robotics, the data of images is used for controlling a robot's motor by using RL. The robot was for example able to screw a cap onto a bottle [3]. RL is also used to give people personal news recommendations in this rapid changing dynamic world where users get bored quickly, have low retention rate and have a high click through rate [4]. One of the most well-known examples of applying RL is for solving different games like Go which is a difficult board game to evaluate because of the amount of possible moves and positions. The control structures called AlphaGo (human trained) and later AlphaGo Zero (self-trained) proved to easily outperform even the best human players in the game [5], [6].

For chemical process control, applying RL has some significant advantages over the current standard used control structures. Classic control like Proportional-Integral-Derivative (PID) is not well suited for nonlinear processes and can only control a single output parameter based on the current measurements of a single input parameter. In addition, this type of control is sensitive to process disturbances [7], [8]. Another type of control is Model Predictive Control (MPC). This type of control is able to control multiple output parameters that are predicted based on a model of the process. This model is both labor intensive and computational demanding to obtain and is complex to update to changing conditions during the process itself [9]. Applying RL for chemical process control will overcome the drawbacks of both classic control and MPC but has also some

complications such as the trade-off that has to be made between exploring and exploiting. If the agent is exploring, it tries new actions that can possible lead to a new way of solving the control task. If the agent is exploiting, it uses the available information for taking actions. Before an agent can exploit its information, it has to be explored first. Therefore, it is important to take this trade-off into account when using RL algorithms [10].

The applications of RL in chemical process control are fewer compared to other sectors such as in robotics but there are still several studies done over the last years. Hoskins and Himmelblau (1992) used artificial neural networks (ANN) for obtaining a function that could turn the current state of a CSTR into suitable control actions for controlling the temperature of the reactor [11]. Martinez (2000) used RL to obtain a model for optimizing the process parameters of a batch reactor starting from an approximate model of the reactor [12]. Syafiie et al. (2008) applied RL in a wastewater treatment plant for controlling the pH-value inside a buffer tank because at around a neutral pH-value, the process has a significant nonlinear behavior [13]. Midhun and Kaisare (2011) managed to control the reactant concentration of an adiabatic plug flow reactor (PFR) by using RL as a model-free approach for control [14]. Shah and Gopal (2016) proposed a RL method that did not need a model of the process or any other prior knowledge of the considered CSTR. The parameters that lead to steady-state conditions are learned by interactions of an agent with the CSTR environment [15]. Cassol et al. (2018) used a RL control structure to track the inlet flowrate of a CSTR while also maximizing the concentration of the wanted product out of the nonlinear Van de Vusse reaction. The used actions by the learning agent were the cooling jacket temperature and the inlet flowrate [16].

For all the mentioned applications, different RL algorithms are used since it is impossible to have only one control structure that can be used for any type of situation. There are three different methods of RL algorithms: Policy-based method, Value-based method and a combination of both [17]. A new RL algorithm, Soft Actor-Critic (SAC), is introduced by Abbeel et al. (2018). This is an off-policy RL algorithm that combines the Value-based method with the Policy-based method while also applying a maximum entropy framework. SAC proved to outperform prior used RL algorithms in a range of continuous control benchmark tasks [18].

Based on this potential, it is hypothesized in this Master's thesis that SAC can also be applied in chemical process control and outperform current used RL algorithms in this sector. In order to evaluate if SAC is indeed applicable for chemical process control, two cases will be taken into consideration. In the first case, SAC is used for controlling the temperature inside the CSTR in which the nonlinear Van de Vusse reaction takes place. The controlled input parameter in this case is the heat removal value. In the second case, SAC is used for controlling the concentration of product b in the CSTR while also keeping the reactor temperature beneath a limit value. The controlled input parameter in this case is the cooling jacket parameter. This second case is studied in more details and based on criteria such as amount of training steps, different time steps and stability, the performance of SAC is evaluated.

First an overview of reinforcement learning and the state-of-the-art Soft Actor-Critic algorithm is described along with the considered CSTR in which the nonlinear Van de Vusse reaction takes place. In the following part, the SAC control structure is described after which the results of both control cases are given. The performance of SAC is analyzed ending with conclusions and recommendations for future work.

2 State-of-the-arts

In this section, Reinforcement Learning and Soft Actor-Critic is explained along with the considered Van de Vusse reaction taking place inside a continuous stirred tank reactor.

2.1 Reinforcement Learning

In the following section, the basic principle of Reinforcement Learning (RL) and a brief history of the most important influences leading towards modern RL are given. The two main RL methods are described and the used RL algorithm in this Master's thesis, Soft Actor-Critic (SAC), is treated in detail.

2.1.1 Overview

At its core, RL is a Markov Decision Process (MDP). This means that the given process can be expressed as an uncertain decision process containing the Markov property. No matter how the current state of the process is achieved, the probability distribution of the next state depends only on the current state. This process provides a mathematical basis for modeling output parameters of the given process based on determined input actions and random input actions. The MDP aims to form a policy under which the agent will perform actions and the main goal is to maximize the sum of all discounted rewards. After each action the current policy will be optimized toward this maximization. The policy contains the information needed for determining the next action based on the current state. There are two different forms of a policy. If the policy already contains the exploration and exploitation nature, then this is an on-policy learning method. If the exploitation nature is not present in the policy, then this is an off-policy learning method. The solution for finding the optimal policy comes from algorithms out of the dynamic programming approaches like the Bellman equation that gives a mathematical solution for the MDP problem. By applying the Bellman equation, the MDP problem is broken down into subproblems that are easier to solve and the solution of these subproblems are used to solve the MDP problem [19], [20].

Throughout history, RL has emerged from two concepts. The first concept is learning by trial and error observed in the study of animal psychology. The second concept is the problem of optimal control that has been studied in the field of mathematics [19].

One of the first times the idea behind trial and error was described was in 1898. Edward Thorndike, in his work 'Review of Animal Intelligence: An Experimental Study of the Associative Processes in Animals', described how animals that accidentally open a box with food in it make an association between their action and the contents of the box. In the beginning of the experiments, the actions taken by the animals where most of the time random. After performing the same experiment several times, the animals will perform most of the time the correct action leading towards the box with food in it. It is these two elements, random actions and actions based on the success of the last actions, that form the basic idea behind trial and error. Whereas the animals needed more than a

hundred seconds in the beginning of the experiments to open the box with food, this was reduced to a few seconds after several repetitions. Edward Thorndike described this phenomenon as the Law of Effect. Fifty years later, Alan Turing used the trial and error concept in a machine that was then still in development, but today worldwide known as the computer. In a technical report from 1948, Turing suggested that a random network of similar units could be used to perform complex tasks. These units could also be trained to perform these tasks better. He basically introduced the first form of artificial neural networks. Turing called these neural networks A-type machines and B-type machines. In A-type machines, units consist of two inputs and one output that can be connected to different units. The connections of these units can be totally random. All these units would be connected to a central synchronizing unit. Every time this central synchronizing unit sends out a signal, all the units in the network will be updated. The B-type machines would be some kind of memory units that can train the overall network [21], [22].

A few years later around 1953, Richard Bellman developed a mathematical optimization method called Dynamic Programming. This is used in the optimal control study of multistage decision processes in which the outcome of one stage can be used to guide the outcome of the next stage. One way of solving these type of processes is to divide the control problem into smaller subproblems. The algorithms used for this, he called the 'Bellman equations'. Later his interests where turned towards stochastic decision problems. In 1957 he proposed a solution for these problems by using iterations of the policy space by Markovian Decision Processes. Further developments where made by Ronald Howard in 1960. In his work 'Dynamic Programming and Markov Processes' he proposed an analytical structure based on the Markov processes and similar to Dynamic Programming that was both descriptive and computationally possible. In the following years, the possibilities of using neural networks where clear but for neural networks with hidden layers there were still no viable training methods developed. In 1972 Paul Werbos proposed a method that is designed to learn an approximation for dynamic programming. He used the information from sending flows back from neuron to neuron along with the classic dynamic programming methods. His proposed method is known as 'backpropagation' [23]–[25].

Richard Sutton, Andrew Barto and Charles Anderson (1983) reconsidered the neuron networks used for control problems by using two types of neuronlike elements instead of one type. These two types of neuronlike element are: Associative Search Element (ASE) and Adaptive Critic Element (ACE). They considered the neuron element to be ASE and improved its learning by adding an ACE. This was the basis for the present known Actor-Critic methods [26].

In the work 'Learning from Delayed Rewards' from Chris Watkins (1989), he described a new algorithm called 'Q-learning' that was able to learn optimal control directly without modelling the transition probabilities or expected rewards of the Markov Decision Process [27].

The basic principle of RL has remained unchanged over the years. As mentioned before, an agent takes an action that influences the environment. Based on this environment, the agent receives a

state and a reward. This state and reward are used to determine a new action and to update the policy [19]. This process is displayed in Figure 1.

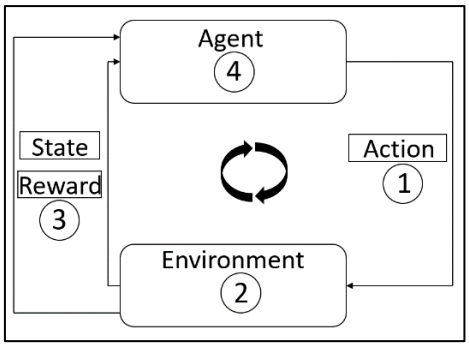


Figure 1: Principle of Reinforcement Learning

In the state-of-the-art RL there are two main methods that can be used separately or combined together. These two methods are: Value-based method and Policy-Based method. Both these two methods will be described in the following parts.

2.1.2 Value-based method

In the Value-based method, a Value-function is used for the determination of the optimal policy. The Value-function estimates if it is good or bad to be in the next state based on a current state. It does this by attaching a reward value to this state and the goal is to maximize the reward over time.

The optimal policy is determined out of the Value-function V(s). This function estimates the probability of being in a certain state and attaches a reward to this probability. The Value-function can be described as a Bellman equation [20], [28]:

$$V^{\pi}(s) = E[\sum_{k=0}^{\infty} \gamma^{k} r_{t+k} | s_t = s, \pi] \quad (1)$$

In which E is denoted as the expectancy of the function, gamma is denoted as the discount factor, pi is the policy, s the state and r is the reward. The goal is to maximize the reward over time by calculating the optimal Value function. After each Value-function iteration step, the value function itself is updated until it converges to the optimal Value function V*(s) described as [20], [28]:

$$V^{*}(s) = \max_{\pi} V^{\pi}(s) \quad (2)$$

This is actually the Value function that gives the highest reward from all the Value functions from all the states. The corresponding policy to this optimal Value function is the optimal policy.

A well-known example of a value-based method is Q-learning. By applying Q-learning, the Value function will not only estimate if it is good or bad to be in the next state but will also take the action from the current state into consideration. In Q-learning, a matrix containing state and action pairs called 'Q-table' is used and initially this table contains only zeros. After each step, this table is updated with the determined Q-values. This table contains all the Q-values for each state and action

pair. An agent can use this table to select an action based on a certain state (exploiting) or can take a random action in a certain state (exploring). The actions in a certain state which have the highest Q-value in the Q-table are the action that lead to the highest reward. The optimal policy in the case of Q-learning is the optimal Q-table. For updating the current Q-value in the table for a certain state-action space, the following equation is used [20]:

$$Q(s,a) = Q(s,a) \ + \alpha \cdot (r + \ \gamma \cdot \max(Q(s',a')) - Q(s,a)) \quad (3)$$

In which α is the learning rate, γ is the discount factor, Q the Q-value, s the state, s' the next state, a the action and a' the next action. The learning rate can have a value between zero and one. The higher the value, the faster learning can occur. If the learning rate is zero, the Q-value is never updated. The discount factor can have a value between zero and one. This value makes sure that the future rewards have less importance than the immediate rewards. The max factor is used for selecting the next action leading to the next state that gives the highest reward [20].

2.1.3 Policy-based method

In the Policy-based method, the optimal policy is directly determined instead of determining the optimal policy by using the Value-function. Therefore, the Value-function does not need to be determined and does not even have to be used at all. Because the policy itself is used for updating towards the optimal policy, not only the action that leads to the highest reward is chosen like in the Value-based method but there is a range of actions that have a high probability of leading to the highest chosen reward [29].

2.1.4 Soft Actor-Critic

A state-of-the-art reinforcement learning algorithm is Soft Actor-Critic. The Actor-Critic part makes the agent part consist out of two parts namely the Actor part and the Critic part. The structure of this type of RL is displayed in Figure 2. Actor-Critic algorithms combine the advantages of the Value-based method and the Policy-based method while reducing the drawbacks of both methods. The Value-based method can be equated to a Critic-only approach while the Policy-based method can be equated to an Actor-only approach. The advantage of the Policy-based method is that this method can be used for control tasks containing continuous action spaces. The drawback of this method, having a possible large variance in the optimal policy, is countered by using the Value-based method. This method makes sure the optimal policy is stable and sample efficient. So actually what happens is that the Actor uses the optimal policy for determining the actions which have the highest probability of leading to the highest reward. The Critic analyses the performed action and updates the optimal policy accordingly [30].

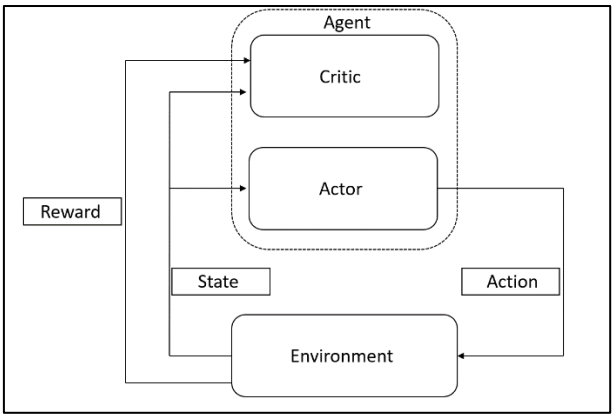


Figure 2: Actor-Critic RL structure

Soft Actor-Critic specifically uses Artificial Neural Networks (ANN) for both the Actor and the Critic part. It also applies a maximum entropy framework for the optimal policy used by the actor.

An Artificial Neural Network (ANN) simulates the interpretation of real-world situations like the human brain does. This results in a computer algorithm that is able to learn and make decisions in the same way as human beings do every day. ANN's are networks made up out of sometimes millions of nodes that are connected to each other by layers. An ANN consist out of a input layer, a series of hidden layers and a output layer and works mainly in a feed-forward way. The input layer takes in the information received from the environment. The hidden layer transforms this input to a usable output. Every connection between a node is weighted by a value. The higher this value is, the more important this connection is for the overall learning process. Initially these weighted values can be random. After each iteration through the ANN, a feed-backward iteration step is done to update the weights and this updating step makes the algorithm learning from the given data. These weights can have a positive or negative value. The higher the positive value, the more important its connections are for the overall network. The lower the negative value, the less important its connections are for the overall network and the network will not take these connections into consideration anymore [31].

As already explained, the policy maps all the possible actions that the agent can take given a certain state. This policy can be described by a Gaussian chart, Figure 3, in which the probability of taking a certain action is described in function of all the possible actions for a given state.

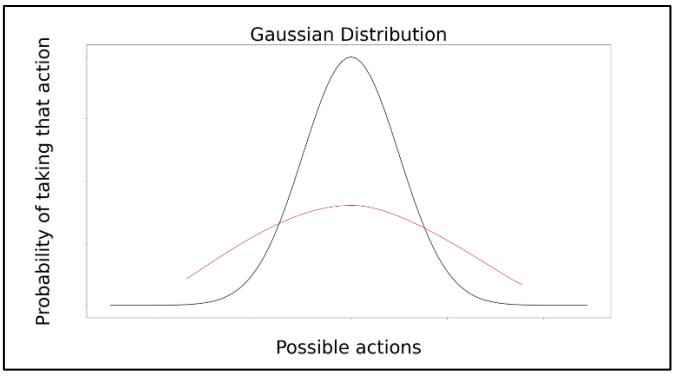


Figure 3: Gaussian Distribution of the policy

In conventional RL algorithms, this distribution (black curve) is a narrow distribution and this leads to a small range of possible actions that have a high probability of taking one of those actions. An agent that follows this type of policy will mainly be focused on exploiting data from the policy to determine the next action. In some situations this can lead to the algorithm getting stuck in a local minimum. This means that the agent thinks it has to choose a certain action leading to the highest known reward but does not take an unknown action into consideration that even leads to an higher reward. In SAC, a maximal entropy framework is used that will enlarge the range of possible actions to take. This will also lower the probability of taking one of those actions. An agent that follows this type of policy will mainly be focused on exploiting data and updating its policy before determining the next action. This leads to a policy that is better adapted to the environment because it takes a broader range of states into consideration [32].

2.2 Van de Vusse Continuous Stirred Tank Reactor

In the following part, the nonlinear Van de Vusse reaction is described along with the used CSTR. The mass and energy balances from the Van de Vusse CSTR are given. These equations form the core equations used by the reinforcement learning control structure to learn, control and track the process.

2.2.1 Van de Vusse reaction

Theoretically, it is often assumed to have a linear reaction behavior inside a reactor but actually a lot of reactions have non-linear behavior. Nonlinear reactions like the Van der Vusse reaction are reactions with an order magnitude of two or higher [33].

In the Van de Vusse reaction as displayed in Figure 4, cyclopentadiene (a) converts into cyclopentenol (b) and dicyclopentadiene (d) in the presence of water. The cyclopentenol (b) converts further into cyclopentanediol (c). The goal is to obtain an as high as possible conversion from cyclopetadiene (a) into cyclopentenol (b). Both the conversion of cyclopentadiene (a) into cyclopentanediol (c) and dicylopentadiene (d) is unwanted [34].

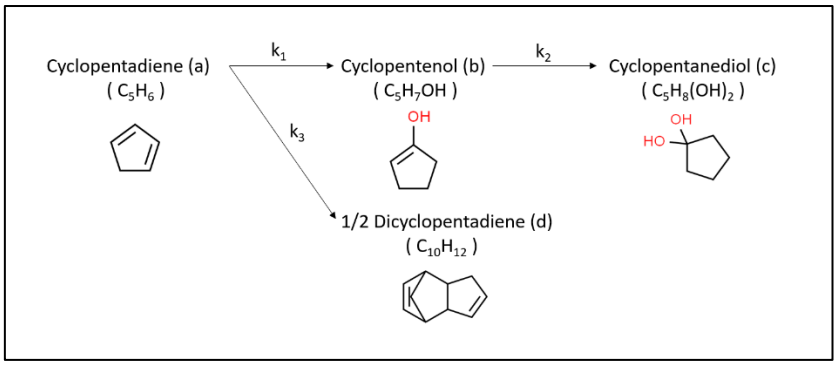


Figure 4: Van de Vusse reaction

The conversion of the products can be described by the reaction rate. This rate displays the change in concentration of a product over time. The reaction rate is expressed as:

$$r = k \cdot [x]^n \tag{4}$$

in which k is the reaction rate constant, n the order magnitude and [x] the concentration of product a and b . For the Van der Vusse reaction the reaction rate expressions are [35]:

$$r_1 = -k_1 \cdot [a] - k_3 \cdot [a]^2 \tag{5}$$

$$r_2 = -k_2 \cdot [b] + k_1 \cdot [a] \tag{6}$$

In which k is the reaction rate constant. This value is temperature dependable and can be described by the Arrhenius equation:

$$k = A \cdot e^{(-E_a/R \cdot T_r)} \tag{7}$$

in which A is the pre-exponential factor, E_a the Arrhenius constant, T_r the reaction temperature and R the universal gas constant. All the parameters of these equations are listed in Table 1 [35]:

Table 1: Arrhenius parameters [35]

Parameter	Value	Units
Pre exponential factor 1 (A_1)	$2.145 \cdot 10^{10}$	1 / min
Pre exponential factor 2 (A_2)	$2.145 \cdot 10^{10}$	1 / min
Pre exponential factor 3 (A_3)	$1.5072 \cdot 10^{8}$	1 / min mol
Arrhenius constant 1 (E_{a1})	81130.5	J / mol
Arrhenius constant 2 (E_{a2})	81130.5	J / mol
Arrhenius constant 3 (E_{a3})	71167.8	J / mol
Universal gas constant (R)	8.314	J / mol K

2.2.2 Continuous Stirred Tank Reactor

The used CSTR in which the Van de Vusse reaction takes place is described in Figure 5.

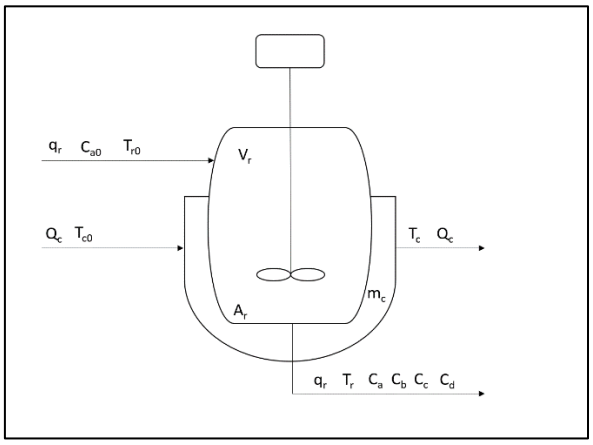


Figure 5: Continuous stirred tank reactor

As can been seen from the figure, the CSTR consist of a cooling jacket by which the temperature of the reactor can been controlled. The different process parameters of the CSTR are: inlet flowrate reactor (q_r), inlet a concentration (C_{a0}), inlet a temperature (T_{r0}), inlet temperature of cooling water (T_{c0}), heat removal value (Q_c), heat exchange surface (A_r), reactor volume (V_r), reactor temperature (T_r), cooling jacket temperature (T_c), cooling water mass (m_c) and outlet a, b, c and d concentration. From these parameters, the reactor temperature, cooling jacket temperature, heat removal value and concentration a, b, c and d are variables. The other parameters are fixed values listed in the Table 2 [35]:

Table 2: CSTR process parameters [35]

Parameter	Value	Units
Volume reactor (V_r)	0.01	m^3
Inlet concentration cyclopentadiene (Ca_0)	5.1	kmol / m^3
Inlet temperature cyclopentadiene (T_{r0})	387.05 (113.9)	K (°C)
Heat exchange surface (A_r)	0.215	m^2
Cooling water mass (m_c)	5	kg
Heat coefficient (U_r)	67.2	kJ / min m^2 K
Inlet flowrate reactor (q_r)	$2.365 \cdot 10^{-3}$	m^3 / min

In order to find the parameters influencing the reactor temperature (T_r), a mass and energy balance of the CSTR has to be made. The energy balances and the material balances of the CSTR can be displayed as [35]:

$$\frac{dC_a}{dt} = \frac{q_r}{V_r} \cdot (C_{a0} - C_a) - k_1 \cdot C_a - k_3 \cdot C_a^2 \quad (8)$$

$$\frac{dC_b}{dt} = -\frac{q_r}{V_r} \cdot C_b + k_1 \cdot C_a - k_2 \cdot C_b \quad (9)$$

$$\frac{dC_c}{dt} = -\frac{q_r}{V_r} \cdot C_c + k_2 \cdot C_b \quad (10)$$

$$\frac{dC_d}{dt} = -\frac{q_r}{V_r} \cdot C_d + k_3 \cdot C_a^2 \quad (11)$$

$$\frac{dT_c}{dt} = \frac{1}{m_c \cdot c_{pc}} \cdot (Q_c + A_r \cdot U_r \cdot (T_r - T_c)) \quad (12)$$

$$\frac{dT_r}{dt} = \frac{-h_r}{\rho \cdot C_p} + \frac{U_r \cdot (T_c - T_r)}{rho \cdot V_r \cdot C_p} + \frac{q_r \cdot (T_{r0} - T_r)}{V_r} \quad (13)$$

The enthalpy of the system (h_r) is composed of all the three reactions that take place and consist of the reaction enthalpies multiplied by the reaction rates.

$$h_r = h_1 \cdot k_1 \cdot [a] + h_2 \cdot k_2 \cdot [b] + h_3 \cdot k_3 \cdot [a]^2 \quad (14)$$

The values of the reaction enthalpies are given in Table 3 [35]:

Table 3: Reaction enthalpies [35]

Parameter	Value	Unit
Reaction enthalpy 1 (h_1)	-4200	kJ / kmol
Reaction enthalpy 2 (h_2)	11000	kJ / kmol
Reaction enthalpy 3 (h_3)	41850	kJ / kmol

The Van de Vusse CSTR can be simulated without applying any type of control at all. Based on the work of Vojtesek (2009) and room temperature conditions, the starting conditions are given in Table 4 [35].

Table 4: Starting conditions Van de Vusse CSTR

Parameter	Value
Concentration a	5.1 kmol/m³
Concentration b, c, d	0.0 kmol/m³
Reactor temperature	298 K
Cooling jacket temperature	298 K

The process in simulated for fifty minutes each time taking a time step in the step function of one minute. The temperature progression and concentration progression are displayed in Figure 6 and Figure 7.

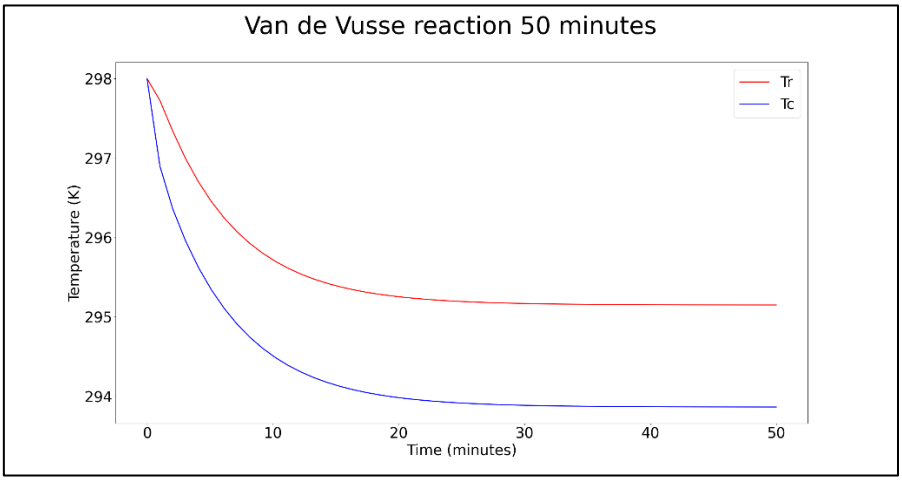


Figure 6: Temperature progression uncontrolled

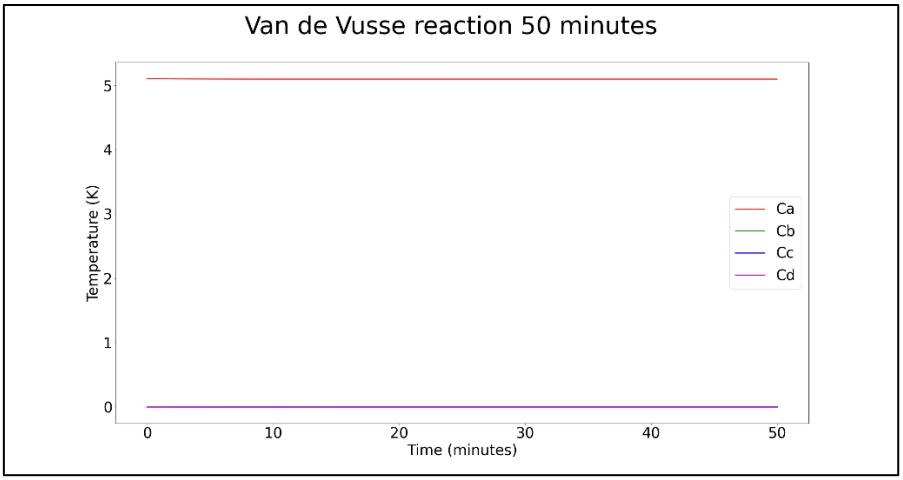


Figure 7: Concentration progression uncontrolled

As can been seen from these figures, the process reaches steady-state conditions after 40 minutes and the biggest changes are observed during the first 10 minutes. The actual values are displayed in Table 5.

Table 5: Results uncontrolled

Parameter	Value after 10 minutes	Value after 40 minutes
Reactor temperature (Tr)	295.802 K	295.150 K
Cooling jacket temperature (Tc)	294.604 K	293.865 K
Concentration a (Ca)	5.093 kmol/m³	5.093 kmol/m³
Concentration b (Cb)	0.002 kmol/m³	0.002 kmol/m³
Concentration c (Cc)	0 kmol/m³	0 kmol/m³
Concentration d (Cd)	0.004 kmol/m³	0.004 kmol/m³

Interesting to notice is that with these starting temperature conditions, almost no reaction takes place. A decrease in temperature in both the reactor and the cooling jacket are observed. Because of these results, it is important to analyze the influence of the starting temperature of the reactor on the concentrations of the products. This can be done by examining the influence of the starting reactor temperature on the concentration progression after 50 minutes. For every different starting reactor temperature, the concentrations of all the products are displayed in Figure 8.

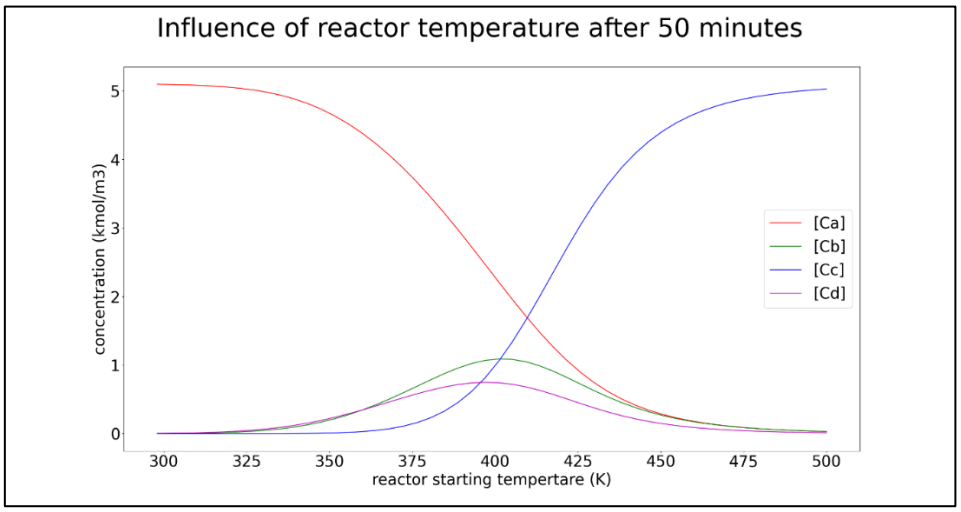


Figure 8: Influence of reactor temperature on concentrations

As can been seen from this figure, the highest concentration of product b is obtained at a starting reactor temperature of 386.298 K. This is also the reason why the initial starting reactor temperature based on the work of Vojtesek (2009) is equal to 387 K. The influence of the starting cooling jacket temperature on the concentration progression is negligible.

3 Reinforcement Learning control

The used programming language in this Master's thesis is Python. Anaconda is an open-source platform made for using Python in both data science and machine learning. The platform contains open-source packages and libraries such as PyTorch and TensorFlow [36]. For programming in the Python language, the integrated development environment (IDE) Spyder is used which is integrated by default in Anaconda [36]. The toolkit Gym is used for constructing the environment [37]. The implementation of Soft Actor-Critic (SAC) for both training and running is given by Stable Baselines3 (SB3) [38]. Before describing and evaluating the results obtained by using a Soft Actor-Critic (SAC) control structure for both the two cases, the environment construction is given. All the used coding files for the environment, the main file and the files used for performing the data analysis are added in the Appendices.

3.1 Environment construction

For constructing the environment used by the SAC control structure, the toolkit Gym is used. This toolkit already contains built-in environments but also supports custom made environments like in this Master's thesis the Van de Vusse CSTR. The Van de Vusse CSTR can be described as a class containing multiple functions. The used functions for this custom environment are: initialization function, reactor function, step function and reset function. The structure of this custom environment is displayed in Figure 9. The detailed coding of the environment for both cases can be found in Appendix B and E.

```
class CSTREnv(gym.Env):
    def __init__(self):
    def reactor(self, Z, t, Qc):
    def step(self, action):
    def reset(self):
```

Figure 9: Structure of custom environment

The initialization function initializes the class and is also used for stating all the used fixed parameters of the Van de Vusse CSTR along with the action space and the observation space. Both the actions and the observations, which are the state parameters, are not fixed to certain values and can take any value within a specified range. The action space is based on the heat removal value (Case 1) or the cooling jacket temperature (Case 2).

The observation space is based on the concentrations a, b, c and d and the reactor temperature and additional the cooling jacket temperature for Case 1. The used action and observation spaces for both the two cases are described in Table 6.

Table 6: Action and observation spaces case 1 and 2

Case 1			Case 2		
Action space:			Action space:		
Parameter	min	max	Parameter	min	max
Qc (kJ/min)	-30	0	Tc (K)	350	450
Observation space:			Observation space:		
Parameter	min	max	Parameter	min	max
Ca (kmol/m³)	0	5.1	Ca (kmol/m³)	0	5.1
Cb (kmol/m³)	0	5.1	Cb (kmol/m³)	0	5.1
Cc (kmol/m³)	0	5.1	Cc (kmol/m³)	0	5.1
Cd (kmol/m³)	0	5.1	Cd (kmol/m³)	0	5.1
Tr (K)	200	1000	Tr (K)	200	1000
Tc (K)	200	1000			

The mass and energy balances deduced from combining the Van de Vusse reaction and the considered CSTR are stated in the reactor function as the model of the process. This reactor function is later used by the step function.

The step function is the main part of the environment. First, an action is selected after which this action is applied to the reactor and simulated during a defined time interval by using odeint. This is a method used for solving the differential equations, (8) to (13), from the reactor which are the mass and energy balances. The outcome of the simulation is then related to a new state that can be used for the next step. Second, the reward function needs to be defined in the step function so the RL algorithm gets a reward for the taken action and can learn from this. For Case 1, this reward function is based on the deviation of the simulated reactor temperature with the reactor temperature goal range and is displayed in the following equation:

$$Reward = f(T_r) = \begin{cases} 1 & T_r \leq T_{rmax} \; and \; T_r \geq T_{rmin} \\ 10 \cdot (T_r - T_{rmin}) & T_r < T_{rmin} \\ 10 \cdot (T_{rmax} - T_r) & T_r > T_{rmax} \end{cases}$$

If the reactor temperature is within this range, a reward of one is given. If the reactor temperature is out of the range, a negative reward is given that increases the further away from the range. For Case 2, this reward function is based on two conditions and is displayed in the following equation:

$$Reward = f(T_r, C_b) = \begin{cases} 1 & C_b \leq C_{bmax} \; and \; C_b \geq C_{bmin} \\ -100 & T_r \geq T_{rmax} \\ 10 \cdot (C_b - C_{bmin}) & C_b < C_{bmin} \\ 10 \cdot (C_{bmax} - C_b) & C_b > C_{bmax} \end{cases}$$

The reactor temperature should never exceed the limit reactor temperature. If the simulated reactor temperature becomes higher then this limit value, a punishment of minus one hundred is given. The reward is also based on the deviation of the simulated concentration of product b with the concentration goal range of product b. If the concentration of product b is within this range and the simulated reactor temperature is lower than the limit reactor temperature, a reward of one is given. If the concentration of product b is out of the range, a negative reward is given that becomes more negative the further away from the range.

The reset function is used for giving the process the starting conditions at time step zero. This contains the starting values for the concentrations of a, b, c and d and the reactor temperature and additional the cooling jacket temperature for Case 1.

3.2 Reinforcement Learning control with Soft Actor-Critic

In the following part all the results for both Case 1 and Case 2 are given and discussed based on the used reactor temperature goals and concentration goal of product b, training steps, time steps and stability. Before analyzing each result, all of these criteria will be properly described.

3.2.1 Case 1: CSTR temperature control with heat transfer value

In Case 1, the reactor temperature is controlled by controlling the heat removal value (Qc) as displayed in Figure 10.

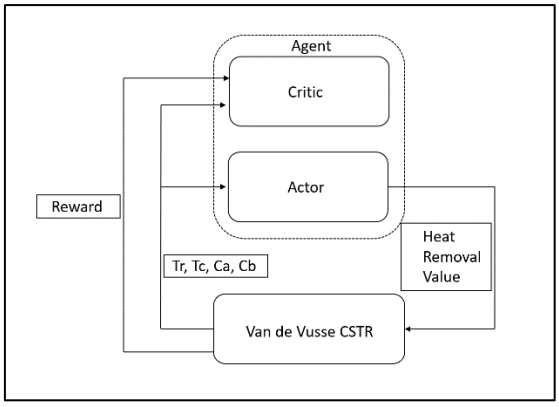


Figure 10: Control structure Case1

The purpose of this case is to evaluate if RL control is able to control a parameter in a chemical system, in this case it is the reactor temperature. This is also the reason for choosing the heat transfer value (Qc) as the action. Practically speaking is it not useful to take this as the action because this value depends on the difference between the reactor temperature and the cooling jacket temperature. But for testing the purpose of RL it is useful to perform this case.

Before setting the temperature goal range, an evaluation needs to be made in which temperature range of the process can be controlled. As mentioned earlier, the heat removal value (Qc) has an

action space ranging from -30 kJ/min to 0 kJ/min. If these two limit values are kept constant during the 50 minutes simulation process, the controllable reactor temperature range is determined as shown in Figure 11.

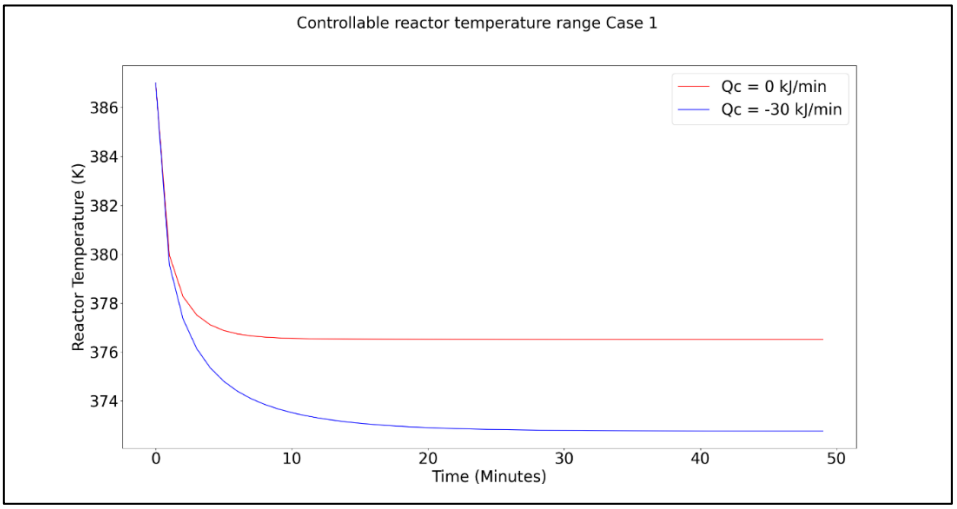


Figure 11: Reactor temperature control range case 1

As can be seen from this figure, the reactor temperature after 50 minutes has a value of 376.512 K if a Qc value of 0 kJ/min is used and has a value of 372.771 K if a Qc value of -30 kJ/min is used. This means that the reactor temperature can be controlled between 372.771 K and 376.512 K. Following this conclusion two reactor temperature goals can be selected being 374 ± 0.05 K and 375 ± 0.05 K. For each selected reactor temperature goal, the reactor temperature progression along with the action value progression and the reward progression is given. For training the SAC control structure, 100 000 training steps are used.

The results obtained for a reactor temperature goal of 374 ± 0.05 K are displayed in Figure 12, Figure 13 and Figure 14.

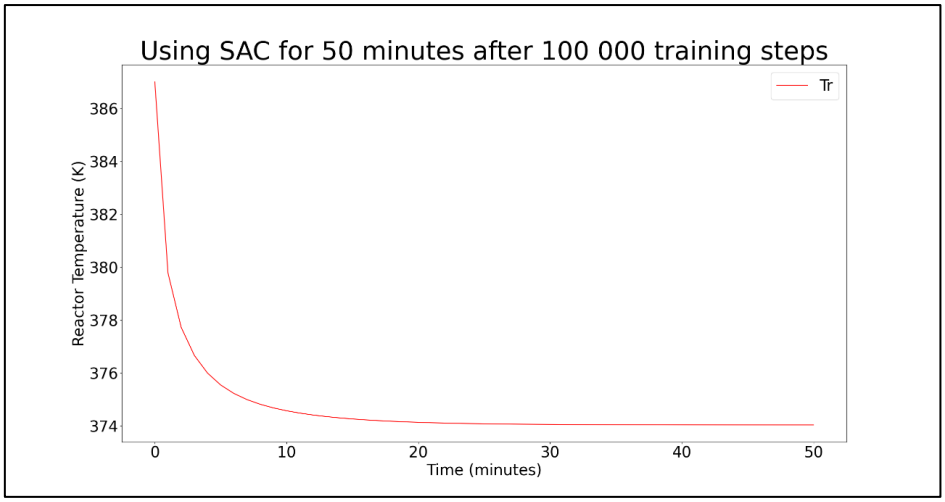


Figure 12: Temperature progression results Case 1: 374±0.05 K

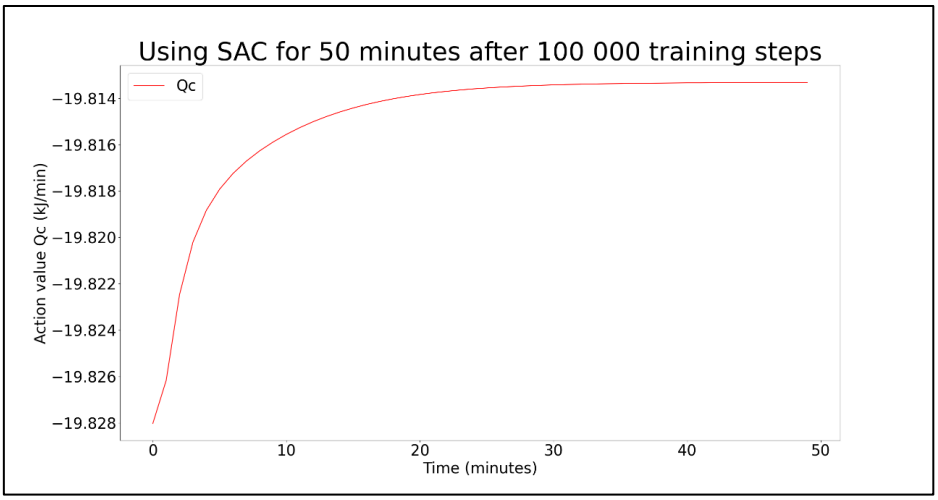


Figure 13: Action value progression Case 1: 374±0.05 K

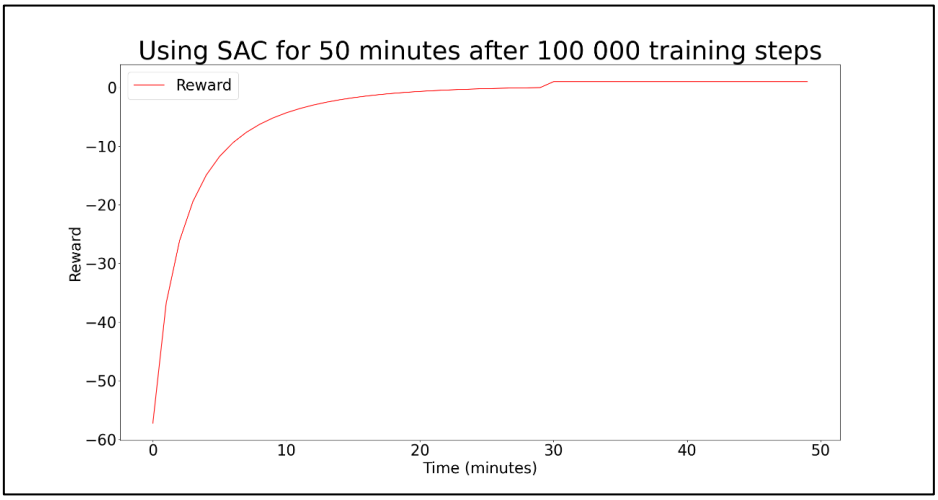


Figure 14: Reward progression Case 1: 374±0.05 K

The results obtained for a reactor temperature goal of 375 ± 0.05 K are displayed in Figure 15, Figure 16 and Figure 17.

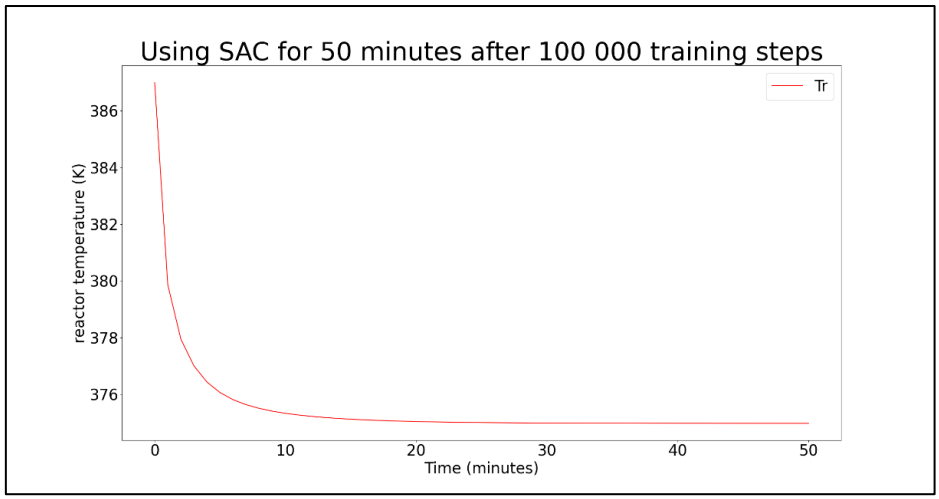


Figure 15: Temperature progression Case 1: 375±0.05 K

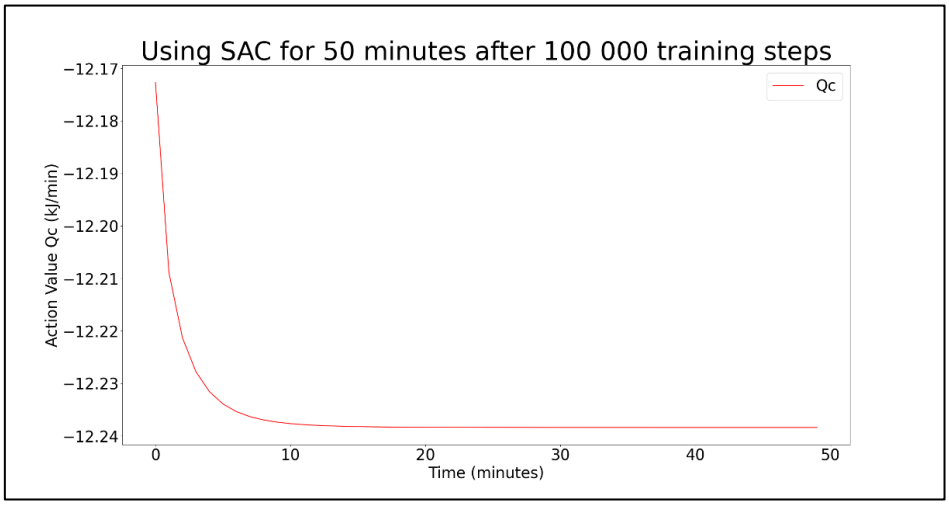


Figure 16: Action value progression Case 1: 375±0.05 K

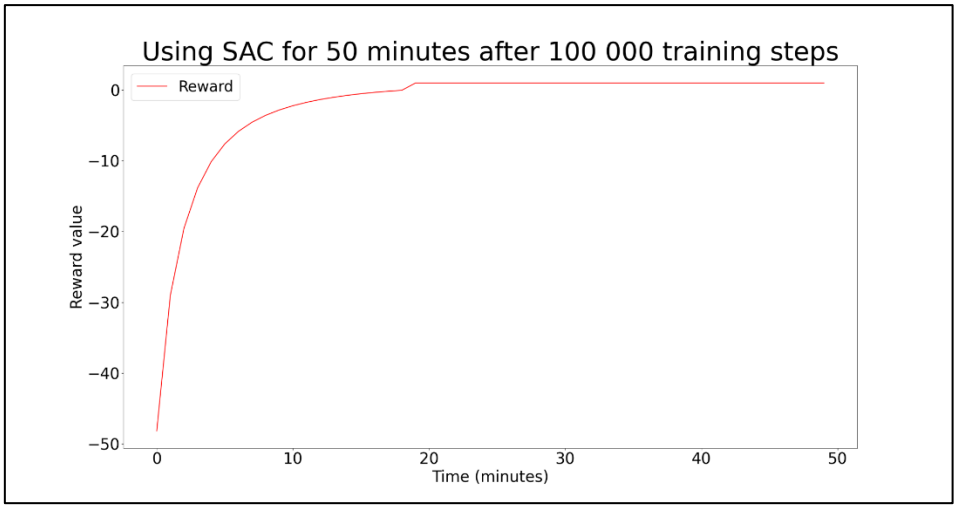


Figure 17: Reward progression Case 1: 375±0.05 K

From the result for both reactor temperature ranges, it is clearly visible that the SAC control structure is indeed capable of controlling the reactor temperature within the pre-defined reactor

temperature range. Based on the reward values, it takes around 30 minutes for SAC to control the reactor temperature within the reactor temperature goal range of 374 ± 0.05 K and it takes around 20 minutes for SAC to control the reactor temperature within the reactor temperature goal range of 375 ± 0.05 K.

3.2.2 Case 2: CSTR concentration control with cooling jacket temperature

From a practical viewpoint, the heat removal value (Qc) is an unusual controllable parameter for controlling the reactor temperature (Tr). A standard controllable parameter is the cooling jacket temperature (Tc) used for controlling the reactor temperature (Tr). Controlling the reactor temperature is important for keeping the process within safe operating conditions but in most cases there is also concentration control present. This can also be achieved by controlling the reactor temperature with the cooling jacket temperature because the reactor temperature has a direct influence on the reaction inside the CSTR and therefore also on the concentrations of the products. The goal of the Van de Vusse reaction inside the CSTR is to obtain an as high as possible concentration of product b.

Case 2 gives a more real-life chemical process control simulation. In this case, the concentration of product b will be controlled by controlling the cooling jacket temperature while also keeping the reactor temperature beneath a limit temperature value. Before stating the controlled concentration goal of product b, the controllable concentration range of product b needs to be determined. By keeping the starting conditions the same as in Case 1 but choosing different constant cooling jacket temperatures, the influence on the concentration of product b can be described by Figure 18.

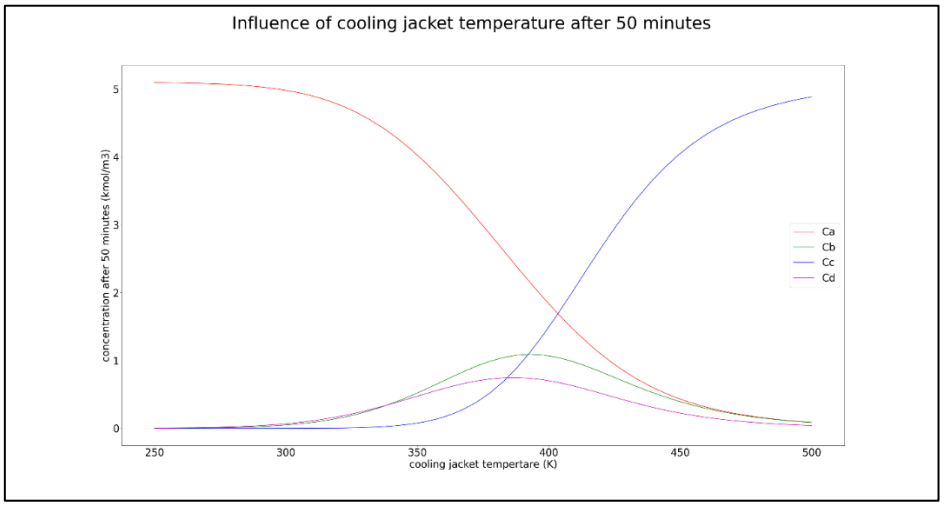


Figure 18: Influence of cooling jacket temperature

As can been seen from this figure, the highest concentrations of product b are obtained by using a cooling jacket temperature range between 350 K and 450 K. This will also be the action space range for the cooling jacket temperature. The corresponding concentrations of product b are 0.528

kmol/m³ and 0.392 kmol/m³. The highest concentration of product b is 1.090 kmol/m³ at a cooling jacket temperature of 387.207 K. Based on this the concentration goal range of product B is chosen to be 1.10 ± 0.05 kmol/m³. It is important to notice that the cooling jacket temperature can have a temperature that is higher than the starting reactor temperature which means that the cooling jacket will not only be able to cool the reactor but also heat up the reactor. The limit temperature value is chosen to be 405 K. For temperatures above this value, the concentration of product b will only decrease and a strong increase of the concentration of product c is observed. The same SAC control structure from Case 1 is used for controlling the concentration of product b, as displayed in Figure 19, Figure 20 and Figure 21.

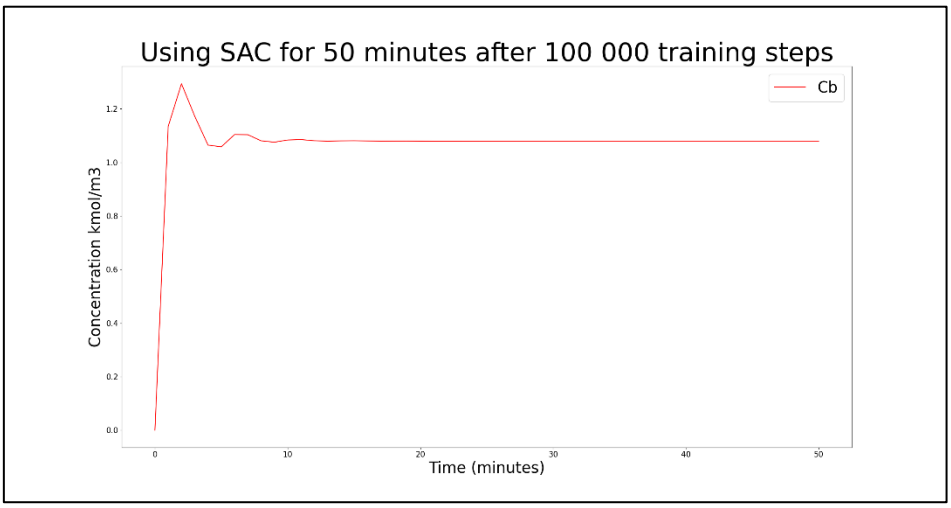


Figure 19: Concentration of product B (Case 2)

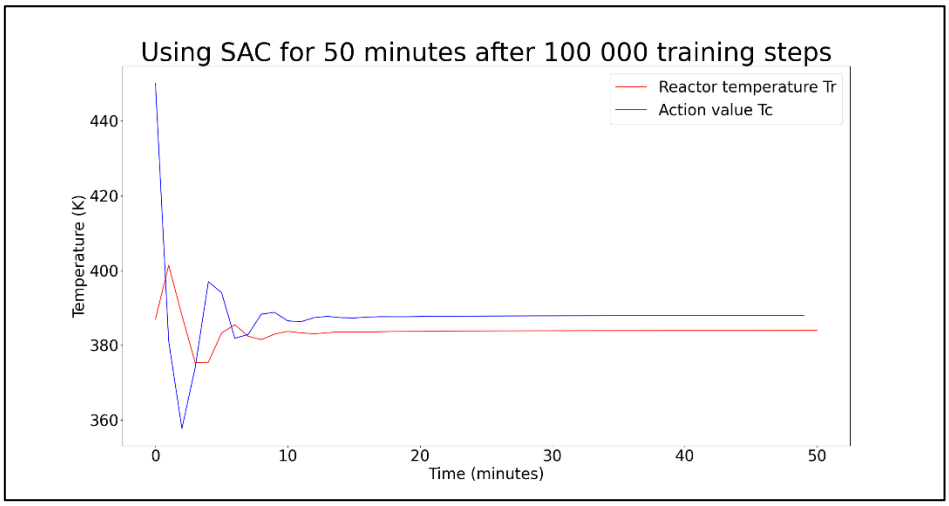


Figure 20: Temperature progression (Case 2)

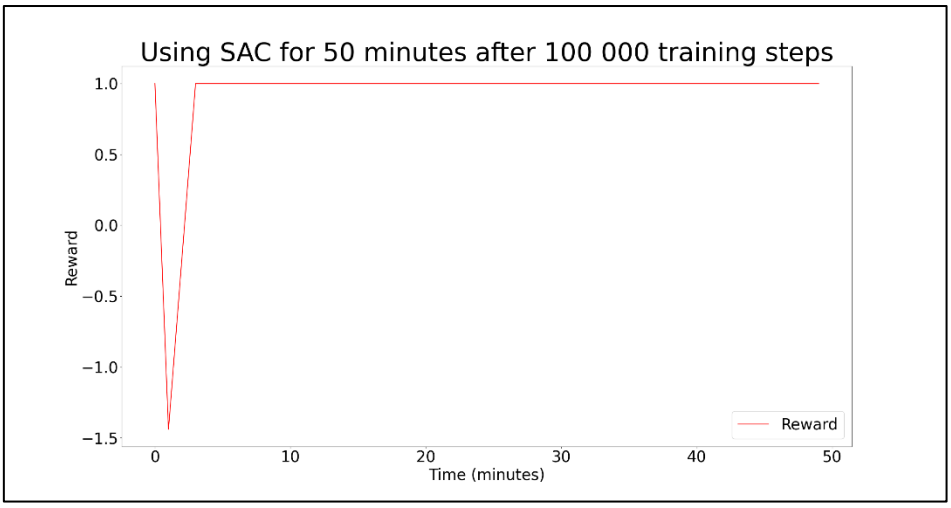


Figure 21: Reward progression (Case 2)

As can been seen from these figures, SAC is able to control the process into the desired concentration goal range of product b and keeping the reactor temperature beneath the limit value. The SAC control structure is able to do this after 3 minutes already. The concentration of product b after 3 minutes is 1.064 kmol/m³ and at steady-state conditions (50 minutes) is 1.078 kmol/m³. The action value (cooling jacket temperature) at steady-state conditions is 387.924 K and the reactor temperature is 383.948 K.

From the previous SAC control application, it is clear that it is indeed capable of controlling the process within a certain concentration range while also taking the reactor temperature into consideration. Therefore it is important to evaluate the performance of this SAC control structure. This can be done based on several criteria. These criteria are the amount of training steps, different time steps and stability.

The amount of training steps are an important part of training the SAC control structure. The more training steps available for the SAC control structure, the more steps are available for the SAC to learn how to control the process to the desired conditions. For the previous simulation, different amount of training steps are used. These different training steps are 1 000, 10 000, 20 000, 30 000, 40 000, 50 000, 60 000, 70 000, 80 000, 90 000 and 100 000. All of these training steps are used two times for training the SAC control structure. The average reward and standard deviation of these two runs are displayed in Figure 22.

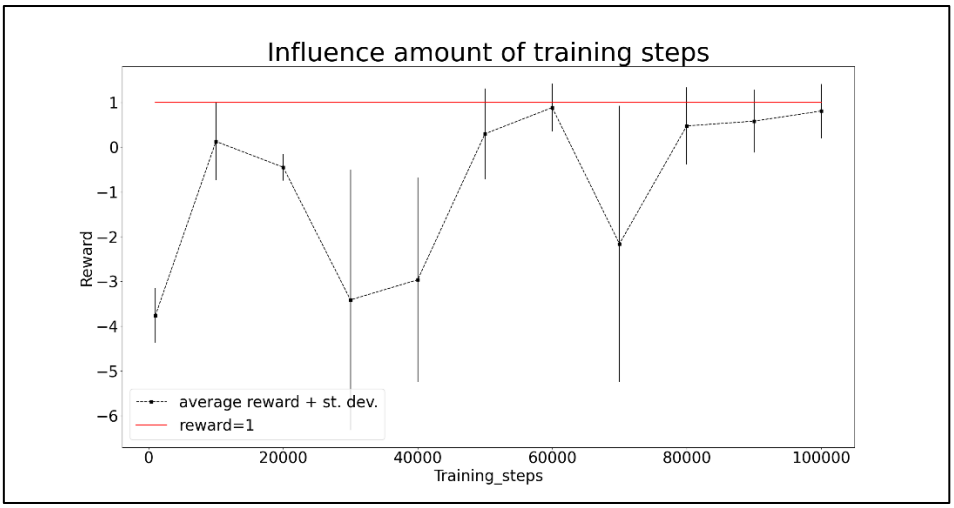


Figure 22: Influence amount of training steps

From these results, it is not perfectly clear what the influence of the different training steps is. However, it can be seen that the more training steps are used, SAC can reach a higher reward. Nonetheless, there are some unexpected results such as 10 000 training steps leading to a higher average reward than 70 000 training steps. For both 60 000 and 100 000 training steps, SAC was able to learn how to control the process within the desired conditions with 60 000 training steps leading to the highest average reward. In order to get a better overview of the influence of the training steps and also for analyzing the stability, the training steps are analyzed by performing 10 000, 50 000 and 100 000 training steps for ten times. In Table 7, the results of these simulations are displayed.

Table 7: Detailed influence amount of training steps based on reward, concentration and time

Training steps	Percentage able to control out of 10 times (%)	Average reward	Average concentration b (kmol/m³)	Average time (minutes)
10 000	60 %	0.357±0.839	1.040±0.162	6.677
50 000	30 %	-2.172±3.032	0.800±0.306	12.000
100 000	40 %	-1.815±3.176	0.830±0.318	5.000

As can been seen from these results, none of the used training steps used by the SAC was able to learn how to control the process to the desired conditions all ten times. Using 10 000 training steps seems to have the highest stability because SAC was able to learn how to control the process six out of ten times. These 10 000 training steps also lead to a positive average reward which means that even if SAC was not able to learn how to control the process after these training steps, the deviation with the concentration goal of product b is small. Taking the achieved concentration of b into consideration, 10 000 training steps seems to give SAC the best chance in learning how to control the process within the desired conditions. Interesting to notice is that for 10 000 training steps and the six times SAC was able to learn, the average concentration of product B is equal to 1.075 kmol/m³. In the case of 50 000 training steps and the three times SAC was able to learn, the average

concentration of product b is equal to 1.063 kmol/m³. In the case of 100 000 training steps and the four times SAC was able to learn, the average concentration of b is equal to 1.077 kmol/m³. Even though SAC can only learn how to control the process four out of ten times using 100 000 training steps, it can achieve a higher concentration of b. The time until reaching the desired conditions is also taken into consideration. Applying 100 000 training steps gives a control structure that can achieve the highest concentration of product b in the least amount of time needed but the difference with 10 000 training steps is so small that it is not worth the amount of time needed for running these simulations. Applying 10 000 training steps takes in the used hardware system (computer) around 5 minutes where applying 100 000 training steps takes around 50 minutes.

Another important influence is the time step used in the environment. For all the simulations done until now, the used time step is one minute. This means that every time the agent takes a step into the environment (Van de Vusse CSTR), the process is simulated for one minute and the values of the concentrations and temperatures after this one minute are used as the new state. This state is than used as the new starting conditions for the next step the agent takes into the environment. In order to evaluate the influence of different time steps, 0.1, 0.2, 0.5 and 1 minute are taken into consideration using 100 000 training steps. The results are displayed in Table 8.

Table 8: Influence of time step

Time step (minutes)	Run	Minutes until reward =1	Concentration product B (kmol/m³)
0.1	1	5	1.089
	2	9	1.072
	3		0.437
0.2	1	4	1.087
	2	5	1.089
	3	4	1.087
0.5	1		0.521
	2	6	1.085
	3	5	1.085
1	1	4	1.086
	2	9	1.084
	3	3	1.070

From this table it is clear that SAC is capable of learning how to control the process within the desired conditions for any used time step. For the used time step of 0.1 minute and 0.5 minute, training the SAC properly is successful in two out of three times. For the used time step of 0.2 minute and 1 minute, training the SAC properly is successful in three out of three times. When comparing a time step of 0.2 minute and 1 minute with each other, using a time step of 0.2 minute will not only take less amount of time before reaching a reward of one but will also lead to a higher average concentration of b compared to every other used time step.

4 Conclusion

The purpose of this Master' thesis was to apply Reinforcement Learning (RL) for controlling the temperature inside a continuous stirred tank reactor in which the nonlinear Van de Vusse reaction takes place. The used RL algorithm is Soft Actor-Critic (SAC).

In the first case (Case 1), the potential of using SAC is analyzed. SAC is trained for 100 000 training steps and is able to control the reactor temperature by the heat removal value. The reactor temperature could be controlled at 374 ± 0.05 Kelvin after 30 minutes and at 375 ± 0.05 Kelvin after 20 minutes. Based on this case, it is concluded that RL can indeed be used in chemical process control.

To analyze the performance and application of RL further, a second case (Case 2) is performed. In this case, the concentration of the main product b (Cyclopentenol) is controlled by the cooling jacket temperature. The concentration of product b could be controlled at 1.10 ± 0.05 kmol/m³ after 3 minutes. To analyze the performance of SAC in this case, the influence of the amount of training steps, the used time step and the stability is taken into consideration. The range of tested training steps is between 1 000 and 100 000 training steps. The range of tested time steps is between 0.1 minute and 1 minute. The stability is tested after performing simulations of the same conditions is tenfold.

Using 10 000 or 100 000 training steps gave the best results from which the second one gave the highest concentration of product b in the least amount of time but with a lower stability and needing ten times as much run time as the first one. Using a time step of 0.2 minute leads to the highest concentration of product b and takes the least amount of time until reaching the desired concentration conditions.

The amount of applications of RL in chemical process control are still low to this day but this Master's thesis definitely showed that it has a lot of potential in the future. Therefore and based on this work, this gives the opportunity to state several recommendations for future work. First, different temperature goals and concentration goals should be tested. Secondly, the amount of training steps should also be tested at one million or even more to see the influence on the training process of SAC. The stability could also be reconsidered after performing more simulations under the same conditions. Third, the reward function could also be formulated in such a way that is adapts itself to the simulation time. By doing so, for every interval of minutes, a different goal temperature of concentration can be set and this will improve the overall performance of SAC. At last, instead of controlling the temperature of the reactor or the concentration, the efficiency or the energy consumption of the reaction can also be used.

Bibliography

[1] S. Sah, "Machine Learning: A Review of Learning Types," no. July, 2020, doi: 10.20944/preprints202007.0230.v1.

[2] I. Arel, C. Liu, T. Urbanik, and A. G. Kohls, "Reinforcement learning-based multi-agent system for network traffic signal control," *IET Intell. Transp. Syst.*, vol. 4, no. 2, pp. 128–135, 2010, doi: 10.1049/iet-its.2009.0070.

[3] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *J. Mach. Learn. Res.*, vol. 17, pp. 1–40, 2016.

[4] G. Zheng *et al.*, "DRN: A deep reinforcement learning framework for news recommendation," *Web Conf. 2018 - Proc. World Wide Web Conf. WWW 2018*, vol. 2, pp. 167–176, 2018, doi: 10.1145/3178876.3185994.

[5] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016, doi: 10.1038/nature16961.

[6] D. Silver *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017, doi: 10.1038/nature24270.

[7] S. W. Sung and I. B. Lee, "Limitations and countermeasures of PID controllers," *Ind. Eng. Chem. Res.*, vol. 35, no. 8, pp. 2596–2610, 1996, doi: 10.1021/ie960090+.

[8] T. HAGIWARA, K. YAMADA, S. MATSUURA, and S. AOYAMA, "A Design Method for Modified PID Controllers for Multiple-Input/Multiple-Output Plants," *J. Syst. Des. Dyn.*, vol. 6, no. 2, pp. 131–144, 2012, doi: 10.1299/jsdd.6.131.

[9] D. E. Seborg, "Model Predictive Control," in *Process Dynamics and Control*, 4th ed., Wiley, 2016, pp. 368–395.

[10] Y. Achbany, F. Fouss, L. Yen, A. Pirotte, and M. Saerens, "Managing the Exploration/Exploitation Trade-Off in Reinforcement Learning," *Tech. Pap. Inf. Syst. Res. Unit (ISYS), IAG, Univ. Cathol. Louvain*, no. July, 2005, [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.9971&rep=rep1&type=pdf.

[11] C. J. Hoskins and M. D. Himmelblau, "Process control via artificial neural networks and reinforcement learning," *IEEE/IAS Int. Conf. Ind. Autom. Control. Proc.*, vol. 16, no. 4, pp. 329–334, 1995, doi: 10.1109/iacc.1995.465819.

[12] E. C. Martinez, "Batch process modeling for optimization using reinforcement learning," *Comput. Chem. Eng.*, vol. 24, no. 2–7, pp. 1187–1193, 2000, doi: 10.1016/S0098-1354(00)00354-9.

[13] S. Syafiie, F. Tadeo, and E. Martinez, "Model-Free Learning Control of Chemical Processes," in *Reinforcement Learning Edited by Cornelius Weber, Mark Elshaw and Norbert Michael Mayer*, London: IntechOpen Limited, 2008.

[14] J. Midhun and N. S. Kaisare, "Approximate dynamic programming-based control of distributed parameter systems," *Asia-Pacific J. Chem. Eng.*, no. 6, pp. 452–459, 2011, doi: 10.1002/apj,568.

[15] H. Shah and M. Gopal, “Model-free predictive control of nonlinear processes based on reinforcement learning,” *IFAC-PapersOnLine*, vol. 49, no. 1, pp. 89–94, 2016, doi: 10.1016/j.ifacol.2016.03.034.

[16] G. O. Cassol, G. V. K. Campos, D. M. Thomaz, B. D. O. Capron, and A. R. Secchi, “Reinforcement Learning Applied to Process Control: A Van der Vusse Reactor Case Study,” *Comput. Aided Chem. Eng.*, vol. 44, pp. 553–558, 2018, doi: 10.1016/B978-0-444-64241-7.50087-2.

[17] Deepanshu Mehta, “State-of-the-Art Reinforcement Learning Algorithms,” *Int. J. Eng. Res.*, vol. V8, no. 12, 2020, doi: 10.17577/ijertv8is120332.

[18] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor,” *35th Int. Conf. Mach. Learn. ICML 2018*, vol. 5, pp. 2976–2989, 2018.

[19] S. R. Sutton and G. A. Barto, *Reinforcement Learning: An Introduction*, Second. London: The MIT Press, 2014.

[20] M. Sewak, *Deep Reinforcement Learning*. Singapore: Springer Nature Singapore Pte Ltd., 2019.

[21] E. L. Thorndike, “Review of Animal Intelligence: An Experimental Study of the Associative Processes in Animals,” *Psychol. Rev.*, vol. 5, no. 5, pp. 551–553, 1898.

[22] C. S. Webster, “Alan Turing’s unorganized machines and artificial neural networks: His remarkable early work and future possibilities,” *Evol. Intell.*, vol. 5, no. 1, pp. 35–43, 2012, doi: 10.1007/s12065-011-0060-5.

[23] S. Dreyfus, “IFORS’ operational research hall of fame Richard Ernest Bellman,” *Int. Trans. Oper. Res.*, vol. 10, pp. 543–545, 2003, doi: 10.1111/j.1475-3995.2006.00566.x.

[24] R. A. Howard, *Dynamic Programming and Markov Processes*. London: The Technology Press of The Massachusetts Institute of Technology, John Wiley & Sons, 1960.

[25] P. J. Werbos, “Backwards Differentiation in AD and Neural Nets: Past Links and New Opportunities,” *Lect. Notes Comput. Sci. Eng.*, vol. 50, pp. 15–34, 2004, doi: 10.1007/3-540-28438-9_2.

[26] A. G. Barto, R. S. Sutton, and C. W. Anderson, “Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems,” *IEEE Trans. Syst. Man Cybern.*, vol. SMC-13, no. 5, pp. 834–846, 1983, doi: 10.1109/TSMC.1983.6313077.

[27] C. J. C. H. Watkins, “Learning from delayed rewards,” *Robotics and Autonomous Systems*, vol. 15, no. 4. pp. 233–235, 1989.

[28] S. Bhagat and H. Banerjee, “Deep Reinforcement Learning for Soft Robotic Applications : Brief Overview with Impending Challenges,” *Unpublished*, no. November, pp. 1–27, 2018, doi: 10.20944/preprints201811.0510.v2.

[29] M. Sewak, “Policy-Based Reinforcement Learning Approaches,” in *Deep Reinforcement Learning*, no. May, Springer, 2019, pp. 127–140.

[30] I. Grondman, L. Busoniu, G. A. D. Lopes, and R. Babuška, “A survey of actor-critic reinforcement learning: Standard and natural policy gradients,” *IEEE Trans. Syst. Man*

Cybern. Part C Appl. Rev., vol. 42, no. 6, pp. 1291–1307, 2012, doi: 10.1109/TSMCC.2012.2218595.

[31] M. N. Vrahatis, G. D. Magoulas, K. E. Parsopoulos, and V. P. Plagianakos, "Introduction to Artificial Neural Networks and Applications," *Proc. Neurosci. 2000 Conf.*, no. October, pp. 1–12, 2001, doi: 10.13140/2.1.1755.2322.

[32] P. Harremoës and F. Topsøe, "Maximum entropy fundamentals," *Entropy*, vol. 3, no. 3, pp. 191–226, 2001, doi: 10.3390/e3030191.

[33] L. Hardesty, "Explained : Linear and nonlinear systems," pp. 1–3, 2015.

[34] K. U. Klatt and S. Engell, "Gain-scheduling trajectory control of a continuous stirred tank reactor," *Comput. Chem. Eng.*, vol. 22, no. 4-5 /5, pp. 491–502, 1998, doi: 10.1016/S0098-1354(97)00261-5.

[35] J. Vojtesek, P. Dostal, and V. Bobal, *Control of nonlinear system - Adaptive and predictive control*, vol. 7, no. PART 1. IFAC, 2009.

[36] Anaconda Inc., "Anaconda Guide," 2019. https://www.anaconda.com/distribution/ (accessed Mar. 04, 2021).

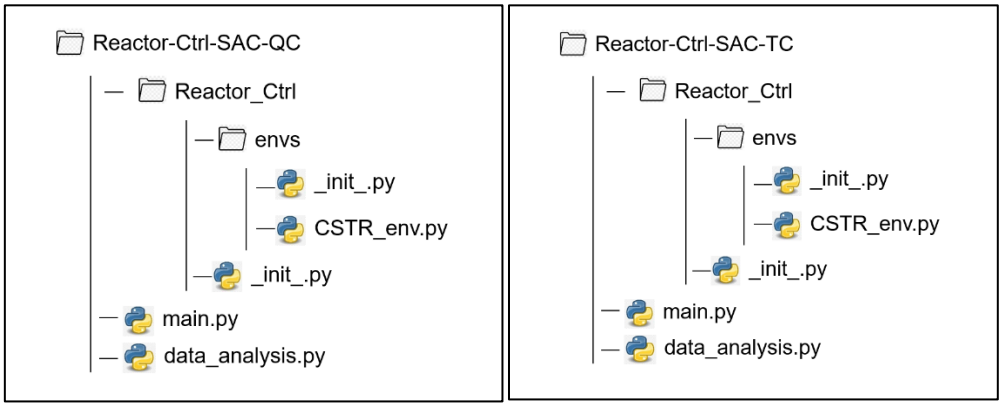
[37] G. Brockman *et al.*, "OpenAI Gym," *ArXiv*, pp. 1–4, 2016, [Online]. Available: http://arxiv.org/abs/1606.01540.

[38] Stable Baselines3 Contributors, "Stable Baselines3 Documentation," 2021.

Appendices

Appendix A: Folder setup

In order to run the following coding files properly, the following folder structure should be used for Case 1 and Case 2 displayed in the following Figures.



The _init_.npy files are important for initializing the custom environment and the environment folder so it can be used by the main file. The two _init_.npy files are displayed in the following Code.

```
from gym.envs.registration import register
import gym

#delete the eixting environment with the same name
env_dict = gym.envs.registration.registry.env_specs.copy()
for env in env_dict:
    if 'CSTR-v0' in env:
        print("Remove {} from registry".format(env))
        del gym.envs.registration.registry.env_specs[env]

register(
    id='CSTR-v0',
    entry_point='Reactor_Ctrl.envs:CSTREnv',
)


```

```
from Reactor_Ctrl.envs.CSTR_env import CSTREnv
```

Appendix B: CSTR_env.ipy (Case 1)

The code used for the custom environment used in Case 1:

```
1.  import gym
2.  from gym.utils import seeding
3.  from gym import logger
4.  from gym import spaces
5.  from gym import Env
6.  from scipy.integrate import odeint
7.  import numpy as np
8.  import math
9.  import matplotlib.pyplot as plt
10. from gym.spaces import Box
11. import random
12. from termcolor import colored
13. from os import path
14.
15. class CSTREnv(gym.Env):
16.     def __init__(self):
17.         self.conc_min = 0
18.         self.conc_max = 5.1
19.         self.Tr_min = 200
20.         self.Tr_max = 1000
21.         self.Tc_min = 200
22.         self.Tc_max = 1000
23.         #action is Qc with values between -30 and 0
24.         self.Qc_low = -30
25.         self.Qc_high = 0
26.
27.
28.         self.qr = 2.365*10**(-3)          # m³/min
29.         self.Vr = 0.01                    # m³
30.         self.Ca0 = 5.1                    # kmol/m³
31.         self.k10 = 2.145*10**10           # 1/min
32.         self.k20 = 2.145*10**10           # 1/min
33.         self.k30 = 1.5072*10**8           # 1/min mol
34.         self.Ea1 = 81130.5                # kJ/kmol
35.         self.Ea2 = 81130.5                # kJ/kmol
36.         self.Ea3 = 71167.8                # kJ/kmol
37.         self.h1 = -4200                   # kJ/kmol
38.         self.h2 = 11000                   # kJ/kmol
39.         self.h3 = 41850                   # kJ/kmol
40.         self.rho = 934.2                  # kg/m³
41.         self.Cp = 3.01                    # kJ/kg K
42.         self.Ur = 67.200                  # kJ/min m² K
43.         self.Ar = 0.215                   # m²
44.         self.Tr0 = 387                    # K
45.         #self.Qc = -18.558                 # kJ/min
46.         self.mc = 5                       # kg
47.         self.Cpc = 2                      # kJ/kg K
48.         self.R = 8.314                    # kJ/K kmol
49.
50.         low = np.array([0, 0, 0, 0, self.Tr_min, self.Tc_min], dtype=np.float32)
51.         high = np.array([5.1, 5.1, 5.1, 5.1, self.Tr_max, self.Tc_max], dtype=np.float32)
52.
53.         self.observation_space = Box(low=low, high=high, dtype=np.float32)
54.         self.action_space = Box(low=np.array([self.Qc_low]),
    high=np.array([self.Qc_high]), dtype=np.float32)
55.
56.         self.seed()
57.
58.     def seed(self, seed=None):
59.         self.np_random, seed = seeding.np_random(seed)
60.         return [seed]
61.
62.     def reactor(self, Z, t, Qc):
63.
```

```
        Ca = Z[0]
        Cb = Z[1]
        Cc = Z[2]
        Cd = Z[3]
        Tr = Z[4]
        Tc = Z[5]

        k1 = self.k10*math.exp(-self.Ea1/(self.R*Tr))
        k2 = self.k20*math.exp(-self.Ea2/(self.R*Tr))
        k3 = self.k30*math.exp(-self.Ea3/(self.R*Tr))

        dCadt = ((self.qr/self.Vr)*(self.Ca0-Ca))-(k1*Ca)-(k3*Ca**2)
        dCbdt = ((-self.qr/self.Vr)*(Cb))+(k1*Ca)-(k2*Cb)
        dCcdt = ((-self.qr/self.Vr)*(Cc))+(k2*Cb)
        dCddt = ((-self.qr/self.Vr)*(Cd))+(k3*Ca**2)
        dTrdt = (-((self.h1*k1*Ca)+(self.h2*k2*Cb)+(self.h3*k3*Ca**2))/(self.rho*self.Cp))+((self.Ur*self.Ar*(Tc-Tr))/(self.rho*self.Vr*self.Cp))+((self.qr*(self.Tr0-Tr))/self.Vr)
        dTcdt = (Qc+(self.Ar*self.Ur*(Tr-Tc)))/(self.mc*self.Cpc)


        return  [dCadt, dCbdt, dCcdt, dCddt, dTrdt, dTcdt]


    def step(self, action):

        action = np.clip(action, self.Qc_low, self.Qc_high)[0]
        self.Qc = action

        time_begin = 0
        time_end = 1
        measure_points = 10
        t = np.linspace(time_begin,time_end,measure_points)

        Z = odeint(self.reactor , self.state, t, args=(self.Qc,))

        self.state = Z[-1]

          lower_temperature_range = 373.95
          upper_temperature_range = 374.05
          goal_temperature = 374

          current_temperature = Z[-1,4]
          last_temperature = Z[0,4]
          temperature =Z[:,4]

          if current_temperature <= upper_temperature_range and current_temperature>=lower_temperature_range :
              reward = 1
              done = True

          elif current_temperature < lower_temperature_range:
              reward = float(10*(current_temperature - lower_temperature_range))
              done = False

          elif current_temperature > upper_temperature_range:
              reward = float(10*(upper_temperature_range - current_temperature))
              done = False

          info = {}

          return self.state, reward, False, {}


    def reset(self):

        self.state = np.array([5.1,0,0,0,387,380])

        return self.state

```

Appendix C: main.ipy (Case 1)

The main file used for training SAC and running the trained SAC for Case 1:

```
import gym
import numpy as np
import stable_baselines3
stable_baselines3.__version__

from stable_baselines3 import SAC

from stable_baselines3.sac.policies import MlpPolicy

env = gym.make("CSTR-v0")

model = SAC(MlpPolicy, env, verbose=1)

model.learn(total_timesteps=100000, log_interval=4)
model.save("sac_cstr")

del model

model= SAC.load("sac_cstr")

obs= env.reset()
t=0
full_obs =[]
full_obsc = []
full_obsc.append(obs[0:4])
full_obs.append(obs[4:6])
full_action = []

full_reward = []
n=50
while t<n:
    t = t+1
    action, _states = model.predict(obs, deterministic=True)
    print(action)
    obs, reward, done, info = env.step(action)
    full_obs.append(obs[4:6])
    full_obsc.append(obs[0:4])
    full_action.append(action)
    full_reward.append(reward)
    print(obs, reward)
    if done:
        # obs = env.reset()
        env.close()

import matplotlib.pyplot as plt
plt.plot(range(n+1),np.array(full_obs)[:,0], 'ro')
fig, ax = plt.subplots(4)
ax[0].plot(range(n+1),np.array(full_obs)[:,0], 'ro', label="Tr")
ax[0].plot(range(n+1),np.array(full_obs)[:,1], 'bo', label="Tc")
ax[0].set_xlabel('Time (Minutes)', size=15)
ax[0].set_ylabel('Temperature (K)', size=15)
ax[0].legend(loc="center", prop={'size': 15})
ax[1].plot(range(n),full_action, 'ro', label="Qc")
ax[1].set_xlabel('Time (Minutes)', size=15)
ax[1].set_ylabel('Actionvalue Qc (kJ/min)', size=15)
ax[1].legend(loc="center", prop={'size': 15})
ax[2].plot(range(n),full_reward, 'yo', label="reward")
ax[2].set_xlabel('Time (Minutes)', size=15)
ax[2].set_ylabel('Reward', size=15)
ax[2].legend(loc="center", prop={'size': 15})
ax[3].plot(range(n+1),np.array(full_obsc)[:,0], 'bo', label="Ca")
ax[3].plot(range(n+1),np.array(full_obsc)[:,1], 'ro', label="Cb")
ax[3].plot(range(n+1),np.array(full_obsc)[:,2], 'yo', label="Cc")
ax[3].plot(range(n+1),np.array(full_obsc)[:,3], 'go', label="Cd")
```

```
ax[3].set_xlabel('Time (Minutes)', size=15)
ax[3].set_ylabel('Concentration (kmol/m3)', size=15)
ax[3].legend(loc="center", prop={'size': 15})
plt.rc('xtick', labelsize=15)
plt.rc('ytick', labelsize=15)
plt.suptitle('Using SAC for 50 minutes after 100 000 training steps', y=0.98, size=30)

# plt.plot(range(n),full_reward, 'r-', label="Reward")
# # plt.plot(range(n+1),np.array(full_obsc)[:,1], 'g-', label="Cb")
# # plt.plot(range(n+1),np.array(full_obsc)[:,2], 'b-', label="Cc")
# # plt.plot(range(n+1),np.array(full_obsc)[:,3], 'm-', label="Cd")
# plt.xlabel('Time (minutes)', size = 30)
# plt.ylabel('Reward', size = 30)
# plt.legend(prop={'size': 30})
# plt.title('Using SAC for 50 minutes after 100 000 training steps', size= 50)
# plt.rc('xtick', labelsize=30)
# plt.rc('ytick', labelsize=30)


# np.save('reward_trained_19',full_reward)
# np.save('reward_trained',full_reward)
# untrained_reward = np.load('reward_untrained.npy')
# trained_reward = np.load('reward_trained.npy')
# fig, ax = plt.subplots()
# ax[0].plot(range(n+1),untrained_reward,'k--', label='Untrained SAC')
# ax[0].plot(range(n+1),trained_reward,'k-', label='Trained SAC')
# ax[0].set_xlabel('Time (Minutes)', size=30)
# ax[0].set_ylabel('Reward',size=30)
# plt.rc('xtick', labelsize=30)
# plt.rc('ytick', labelsize=30)
# ax[0].legend(loc="center", prop={'size': 30})

# from stable_baselines3.common.evaluation import evaluate_policy
# eval_env = gym.make('CSTR-v0')
# mean_reward, std_reward = evaluate_policy(model, eval_env, n_eval_episodes=100,
deterministic=True)

 # print(f"mean_reward={mean_reward:.2f} +/- {std_reward}")


 # np.save('untrained_react_temp',np.array(full_obs)[:,0])

```

Appendix D: Coding file for data analysis Case 1

The coding file used for data analysis in Case 1:

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
import math
import gym
from gym.utils import seeding
from gym import logger
from gym import spaces
from gym import Env
from scipy.integrate import odeint
import numpy as np
import math
import matplotlib.pyplot as plt
from gym.spaces import Box
import random
from termcolor import colored
from os import path

# trained_1 = np.load('reward_trained_1.npy')
# trained_2 = np.load('reward_trained_2.npy')
# trained_3 = np.load('reward_trained_3.npy')
# trained_4 = np.load('reward_trained_4.npy')
# trained_5 = np.load('reward_trained_5.npy')
# trained_6 = np.load('reward_trained_6.npy')
# trained_7 = np.load('reward_trained_7.npy')
# trained_8 = np.load('reward_trained_8.npy')
# trained_9 = np.load('reward_trained_9.npy')
# trained_10 = np.load('reward_trained_10.npy')
# trained_11 = np.load('reward_trained_1.npy')
# trained_12 = np.load('reward_trained_2.npy')
# trained_13 = np.load('reward_trained_3.npy')
# trained_14 = np.load('reward_trained_4.npy')

# plt.plot(range(50),trained_1,'k--')
# plt.plot(range(50),trained_2,'k--')
# plt.plot(range(50),trained_3,'k--')
# plt.plot(range(50),trained_4,'k--')
# plt.plot(range(50),trained_5,'k--')
# plt.plot(range(50),trained_6,'k--')
# plt.plot(range(50),trained_7,'k--')
# plt.plot(range(50),trained_8,'k--')
# plt.plot(range(50),trained_9,'k--')
# plt.plot(range(50),trained_10,'k--')
# plt.plot(range(50),trained_11,'k--')
# plt.plot(range(50),trained_12,'k--')
# plt.plot(range(50),trained_13,'k--')
# plt.plot(range(50),trained_14,'k--')
# plt.xlabel('Time (Minutes)', size=50)
# plt.ylabel('Reward', size=50)
# # plt.ylim(-5,2)
# #ax.legend(loc="center", prop={'size': 30})
# plt.suptitle('Reward for Untrained SAC control structure', y=0.98, size=50)
# plt.rc('xtick', labelsize=50)
# plt.rc('ytick', labelsize=50)

# print(trained_6)


trained_react_temp = np.load('trained_reactor_temp.npy')
trained_cooling_temp = np.load('trained_cooling_temp.npy')
untrained_react_temp = np.load('untrained_react_temp.npy')
untrained_cooling_temp = np.load('untrained_cooling_temp.npy')

plt.plot(range(51),trained_react_temp,'r-', label="trained")
```

```
#plt.plot(range(51),trained_cooling_temp,'b-', label="trained")
plt.plot(range(51),untrained_react_temp,'r--', label="untrained")
#plt.plot(range(51),untrained_cooling_temp,'b--', label="untrained")
plt.legend(loc="center")
plt.suptitle('Temperature untrained vs trained', y=0.98, size=50)
plt.xlabel('Time (Minutes)', size=50)
plt.ylabel('temperature (K)', size=50)

```

Appendix E: CSTR_env.ipy (Case 2)

The code used for the custom environment used in Case 2:

```
1.  import gym
2.  from gym.utils import seeding
3.  from gym import logger
4.  from gym import spaces
5.  from gym import Env
6.  from scipy.integrate import odeint
7.  import numpy as np
8.  import math
9.  import matplotlib.pyplot as plt
10. from gym.spaces import Box
11. import random
12. from termcolor import colored
13. from os import path
14.
15.
16. class CSTREnv(gym.Env):
17.     def __init__(self):
18.         self.conc_min = 0
19.         self.conc_max = 5.1
20.         self.Tr_min = 200
21.         self.Tr_max = 1000
22.         self.Tr_goal = 320.0
23.         self.Tc_min = 350
24.         self.Tc_max = 450
25.
26.         self.qr = 2.365*10**(-3)          # m³/min
27.         self.Vr = 0.01                    # m³
28.         self.Ca0 = 5.1                    # kmol/m³
29.         self.k10 = 2.145*10**10           # 1/min
30.         self.k20 = 2.145*10**10           # 1/min
31.         self.k30 = 1.5072*10**8           # 1/min mol
32.         self.Ea1 = 81130.5                # kJ/kmol
33.         self.Ea2 = 81130.5                # kJ/kmol
34.         self.Ea3 = 71167.8                # kJ/kmol
35.         self.h1 = -4200                   # kJ/kmol
36.         self.h2 = 11000                   # kJ/kmol
37.         self.h3 = 41850                   # kJ/kmol
38.         self.rho = 934.2                  # kg/m³
39.         self.Cp = 3.01                    # kJ/kg K
40.         self.Ur = 67.200                  # kJ/min m² K
41.         self.Ar = 0.215                   # m²
42.         self.Tr0 = 387                    # K
43.         self.Qc = -18.56                 # kJ/min
44.         self.mc = 5                       # kg
45.         self.Cpc = 2                      # kJ/kg K
46.         self.R = 8.314                    # kJ/K kmol
47.
48.         low = np.array([0, 0, 0, 0, self.Tr_min], dtype=np.float32)
49.         high = np.array([5.1, 5.1, 5.1, 5.1, self.Tr_max], dtype=np.float32)
50.
51.         self.observation_space = Box(low=low, high=high, dtype=np.float32)
52.         self.action_space = Box(low=np.array([self.Tc_min]),
    high=np.array([self.Tc_max]), dtype=np.float32)
53.
54.     def reactor(self, Z, t, Tc):
55.
56.         Ca = Z[0]
57.         Cb = Z[1]
58.         Cc = Z[2]
59.         Cd = Z[3]
60.         Tr = Z[4]
61.
62.         k1 = self.k10*math.exp(-self.Ea1/(self.R*Tr))
63.         k2 = self.k20*math.exp(-self.Ea2/(self.R*Tr))
```

```
        k3 = self.k30*math.exp(-self.Ea3/(self.R*Tr))

        dCadt = ((self.qr/self.Vr)*(self.Ca0-Ca))-(k1*Ca)-(k3*Ca**2)
        dCbdt = ((-self.qr/self.Vr)*(Cb))+(k1*Ca)-(k2*Cb)
        dCcdt = ((-self.qr/self.Vr)*(Cc))+(k2*Cb)
        dCddt = ((-self.qr/self.Vr)*(Cd))+(k3*Ca**2)
        dTrdt = (-((self.h1*k1*Ca)+(self.h2*k2*Cb)+(self.h3*k3*Ca**2))/(self.rho*self.Cp))+((self.Ur*self.Ar*(Tc-Tr))/(self.rho*self.Vr*self.Cp))+((self.qr*(self.Tr0-Tr))/self.Vr)

        return  [dCadt, dCbdt, dCcdt, dCddt, dTrdt]


    def step(self, action):

        action = np.clip(action, self.Tc_min, self.Tc_max)[0]
        self.Tc = action

        time_begin = 0
        time_end = 1
        measure_points = 10
        t = np.linspace(time_begin,time_end,measure_points)

        Z = odeint(self.reactor , self.state, t, args=(self.Tc,))

        self.state = Z[-1]

        limit_temperature = 405

        lower_concentration = 1.05
        upper_concentration = 1.15
        goal_concentration = 1.10

        current_temperature = Z[-1,4]
        last_temperature = Z[0,4]
        temperature =Z[:,4]

        current_concentration = Z[-1,1]
          # current_concentration_c = Z[-1,2]
          # max_conc_C = 1.00

          if  (current_concentration <= upper_concentration and current_concentration >= lower_concentration):
              reward = 1

          elif current_temperature >= limit_temperature:
              reward = -100

          elif (current_concentration < lower_concentration):
              reward = float(10*(current_concentration-lower_concentration))

          elif (current_concentration > upper_concentration):
              reward = float(10*(upper_concentration-current_concentration))

          info = {}

          return self.state, reward, False, {}


    def reset(self):

        self.state = np.array([5.1,0,0,0,387])

        return self.state

```

Appendix F: main.ipy (Case 2)

The main file used for training SAC and running the trained SAC for Case 2:

```
import gym
import numpy as np
import stable_baselines3
stable_baselines3.__version__

from stable_baselines3 import SAC

from stable_baselines3.sac.policies import MlpPolicy

env = gym.make("CSTR-v0")

model = SAC(MlpPolicy, env, verbose=1)

model.learn(total_timesteps=100000, log_interval=4)
model.save("sac_cstr")

del model

model= SAC.load("sac_cstr")

obs= env.reset()
t=0
full_obs =[]
full_obsc = []
full_obsc.append(obs[0:4])
full_obs.append(obs[4:5])
full_action = []

full_reward = []
n= 50
while t<n:
    t = t+1
    action, _states = model.predict(obs, deterministic=True)
    print(action)
    obs, reward, done, info = env.step(action)
    full_obs.append(obs[4:5])
    full_obsc.append(obs[0:4])
    full_action.append(action)
    full_reward.append(reward)
    print(obs, reward)
    if done:
        # obs = env.reset()
        env.close()

import matplotlib.pyplot as plt
# plt.plot(range(n+1),np.array(full_obs)[:,0], 'ro')
fig, ax = plt.subplots(4)
ax[0].plot(range(n+1),np.array(full_obs)[:,0], 'ro', label="Tr")
# ax[0].plot(range(n+1),np.array(full_obs)[:,1], 'bo', label="Tc")
# ax[0].set_xlabel('Time (Minutes)', size=15)
ax[0].set_ylabel('Temperature (K)', size=15)
ax[0].legend(loc="center", prop={'size': 15})
ax[1].plot(range(n),full_action, 'ro', label="Tc")
# ax[1].set_xlabel('Time (Minutes)', size=15)
ax[1].set_ylabel('Actionvalue Tc (K)', size=15)
ax[1].legend(loc="center", prop={'size': 15})
ax[2].plot(range(n),full_reward, 'yo', label="reward")
# ax[2].set_xlabel('Time (Minutes)', size=15)
ax[2].set_ylabel('Reward', size=15)
ax[2].legend(loc="center", prop={'size': 15})
ax[3].plot(range(n+1),np.array(full_obsc)[:,0], 'bo', label="Ca")
ax[3].plot(range(n+1),np.array(full_obsc)[:,1], 'ro', label="Cb")
ax[3].plot(range(n+1),np.array(full_obsc)[:,2], 'yo', label="Cc")
ax[3].plot(range(n+1),np.array(full_obsc)[:,3], 'go', label="Cd")
```

```
ax[3].set_xlabel('Time (Minutes)', size=15)
ax[3].set_ylabel('Concentration (kmol/m3)', size=15)
ax[3].legend(loc="center", prop={'size': 15})
plt.rc('xtick', labelsize=15)
plt.rc('ytick', labelsize=15)

# print(np.array(full_obsc)[:,1])

# np.save('100_000_concb_concgoal1',np.array(full_obsc)[:,1])
# np.save('reward_trained',full_reward)
# untrained_reward = np.load('reward_untrained.npy')
# trained_reward = np.load('reward_trained.npy')
# fig, ax = plt.subplots()
# ax[0].plot(range(n+1),untrained_reward,'k--', label='Untrained SAC')
# ax[0].plot(range(n+1),trained_reward,'k-', label='Trained SAC')
# ax[0].set_xlabel('Time (Minutes)', size=30)
# ax[0].set_ylabel('Reward',size=30)
# plt.rc('xtick', labelsize=30)
# plt.rc('ytick', labelsize=30)
# ax[0].legend(loc="center", prop={'size': 30})

# from stable_baselines3.common.evaluation import evaluate_policy
# eval_env = gym.make('CSTR-v0')
# mean_reward, std_reward = evaluate_policy(model, eval_env, n_eval_episodes=100,
deterministic=True)

# print(f"mean_reward={mean_reward:.2f} +/- {std_reward}")


# np.save('concb_6_10',np.array(full_obsc)[:,1])
# np.save('reward_6_10',full_reward)
# np.save('action_value_6_10',full_action)

```

Appendix G: Coding file for data analysis Case 2

The coding file used for data analysis in Case 2:

```
import numpy as np
from scipy.integrate import odeint
import matplotlib.pyplot as plt
import math
import gym
from gym.utils import seeding
from gym import logger
from gym import spaces
from gym import Env
from scipy.integrate import odeint
import numpy as np
import math
import matplotlib.pyplot as plt
from gym.spaces import Box
import random
from termcolor import colored
from os import path
import statistics

# trained_1 = np.load('reward_untrained_1.npy')
# trained_2 = np.load('reward_untrained_2.npy')
# trained_3 = np.load('reward_untrained_3.npy')
# trained_4 = np.load('reward_untrained_4.npy')
# trained_5 = np.load('reward_untrained_5.npy')
# trained_6 = np.load('reward_untrained_6.npy')
# trained_7 = np.load('reward_untrained_7.npy')
# trained_8 = np.load('reward_untrained_8.npy')
# trained_9 = np.load('reward_untrained_9.npy')
# trained_10 = np.load('reward_untrained_10.npy')
# trained_11 = np.load('reward_untrained_1.npy')
# trained_12 = np.load('reward_untrained_2.npy')
# trained_13 = np.load('reward_untrained_3.npy')
# trained_14 = np.load('reward_untrained_4.npy')

# plt.plot(range(50),trained_1,'k--')
# plt.plot(range(50),trained_2,'k--')
# plt.plot(range(50),trained_3,'k--')
# plt.plot(range(50),trained_4,'k--')
# plt.plot(range(50),trained_5,'k--')
# plt.plot(range(50),trained_6,'k--')
# plt.plot(range(50),trained_7,'k--')
# plt.plot(range(50),trained_8,'k--')
# plt.plot(range(50),trained_9,'k--')
# plt.plot(range(50),trained_10,'k--')
# plt.plot(range(50),trained_11,'k--')
# plt.plot(range(50),trained_12,'k--')
# plt.plot(range(50),trained_13,'k--')
# plt.plot(range(50),trained_14,'k--')
# plt.xlabel('Time (Minutes)', size=50)
# plt.ylabel('Reward', size=50)
# # plt.ylim(-5,2)
# #ax.legend(loc="center", prop={'size': 30})
# plt.suptitle('Reward for Untrained SAC control structure', y=0.98, size=50)
# plt.rc('xtick', labelsize=50)
# plt.rc('ytick', labelsize=50)

# print(trained_6)

# trained_react_temp = np.load('trained_reactor_temp.npy')
# trained_cooling_temp = np.load('trained_cooling_temp.npy')
# untrained_react_temp = np.load('untrained_react_temp.npy')
# untrained_cooling_temp = np.load('untrained_cooling_temp.npy')

# plt.plot(range(51),trained_react_temp,'r-', label="trained")
```

```
# #plt.plot(range(51),trained_cooling_temp,'b-', label="trained")
# plt.plot(range(51),untrained_react_temp,'r--', label="untrained")
# #plt.plot(range(51),untrained_cooling_temp,'b--', label="untrained")
# plt.legend(loc="center")
# plt.suptitle('Temperature untrained vs trained', y=0.98, size=50)
# plt.xlabel('Time (Minutes)', size=50)
# plt.ylabel('temperature (K)', size=50)
```

```
# #REWARD FUNCTION PLOTTING
# ywaarde1 = np.load('reward_4_1.npy')
# ywaarde2 = np.load('reward_4_2.npy')
# ywaarde3 = np.load('reward_4_3.npy')
# ywaarde4 = np.load('reward_4_4.npy')
# ywaarde5 = np.load('reward_4_5.npy')
# ywaarde6 = np.load('reward_4_6.npy')
# ywaarde7 = np.load('reward_4_7.npy')
# ywaarde8 = np.load('reward_4_8.npy')
# # ywaarde9 = np.load('reward_4_9.npy')
# # ywaarde10 = np.load('reward_4_10.npy')

# # concb = np.load('100_000_concb_concgoal1.npy')
# plt.plot(range(50),ywaarde1,'r-', label="1")
# plt.plot(range(50),ywaarde2,'b-', label="2")
# plt.plot(range(50),ywaarde3,'y-', label="3")
# plt.plot(range(50),ywaarde4,'g-', label="4")
# plt.plot(range(50),ywaarde5,'m-', label="5")
# plt.plot(range(50),ywaarde6,'k-', label="6")
# plt.plot(range(50),ywaarde7,'r--', label="7")
# plt.plot(range(50),ywaarde8,'b--', label="8")
# # plt.plot(range(50),ywaarde9,'y--', label="9")
 # # plt.plot(range(50),ywaarde10,'g--', label="10")

 # # plt.plot(range(51),concb,'r-', label="Cb")
 # plt.xlabel('Time (minutes)', size = 30)
 # plt.xlim([0,20])
 # plt.ylabel('Reward', size = 30)
 # plt.legend(prop={'size': 30})
 # plt.title('Influence of amount of training steps', size= 50)
 # plt.rc('xtick', labelsize=30)
 # plt.rc('ytick', labelsize=30)
```

```
 # # CONCENTRATION PLOTTING
 # ywaarde1 = np.load('concb_4_1.npy')
 # ywaarde2 = np.load('concb_4_2.npy')
 # ywaarde3 = np.load('concb_4_3.npy')
 # ywaarde4 = np.load('concb_4_4.npy')
 # ywaarde5 = np.load('concb_4_5.npy')
 # ywaarde6 = np.load('concb_4_6.npy')
 # ywaarde7 = np.load('concb_4_7.npy')
 # ywaarde8 = np.load('concb_4_8.npy')
 # ywaarde9 = np.load('concb_4_9.npy')
 # ywaarde10 = np.load('concb_4_10.npy')

 # plt.plot(range(51),ywaarde1,'r-', label="1")
 # plt.plot(range(51),ywaarde2,'b-', label="2")
 # plt.plot(range(51),ywaarde3,'y-', label="3")
 # plt.plot(range(51),ywaarde4,'g-', label="4")
 # plt.plot(range(51),ywaarde5,'m-', label="5")
 # plt.plot(range(51),ywaarde6,'k-', label="6")
```

```
135.  # plt.plot(range(51),ywaarde7,'r--', label="7")
136.  # plt.plot(range(51),ywaarde8,'b--', label="8")
137.  # plt.plot(range(101),ywaarde9,'y--', label="9")
138.  # plt.plot(range(51),ywaarde10,'g--', label="10")
139.
140.  # plt.xlabel('Time (minutes)', size = 30)
141.  # plt.xlim([0,50])
142.  # # plt.ylim([1,1.20])
143.  # plt.ylabel('Concentration product B (kmol/m3)', size = 30)
144.  # plt.legend(prop={'size': 30})
145.  # plt.title('Influence of amount of training steps', size= 50)
146.  # plt.rc('xtick', labelsize=30)
147.  # plt.rc('ytick', labelsize=30)
148.
149.
150.
151.  # # ACTION VALUE PLOTTING
152.  # ywaarde1 = np.load('action_value_4_1.npy')
153.  # ywaarde2 = np.load('action_value_4_2.npy')
154.  # ywaarde3 = np.load('action_value_4_3.npy')
155.  # ywaarde4 = np.load('action_value_4_4.npy')
156.  # ywaarde5 = np.load('action_value_4_5.npy')
157.  # ywaarde6 = np.load('action_value_4_6.npy')
158.  # ywaarde7 = np.load('action_value_4_7.npy')
159.  # ywaarde8 = np.load('action_value_4_8.npy')
160.  # # ywaarde9 = np.load('action_value_4_9.npy')
161.  # # ywaarde10 = np.load('action_value_4_10.npy')
162.
163.  # plt.plot(range(50),ywaarde1,'r-', label="1")
164.  # plt.plot(range(50),ywaarde2,'b-', label="2")
165.  # plt.plot(range(50),ywaarde3,'y-', label="3")
166.  # plt.plot(range(50),ywaarde4,'g-', label="4")
167.  # plt.plot(range(50),ywaarde5,'m-', label="5")
168.  # plt.plot(range(50),ywaarde6,'k-', label="6")
169.  # plt.plot(range(50),ywaarde7,'r--', label="7")
170.  # plt.plot(range(50),ywaarde8,'b--', label="8")
171.  # # plt.plot(range(50),ywaarde9,'y--', label="9")
172.  # # plt.plot(range(50),ywaarde10,'g--', label="10")
173.
174.  # plt.xlabel('Time (minutes)', size = 30)
175.  # plt.xlim([0,50])
176.  # # plt.ylim([1,1.20])
177.  # plt.ylabel('Reward', size = 30)
178.  # plt.legend(prop={'size': 30})
179.  # plt.title('Influence of amount of training steps', size= 50)
180.  # plt.rc('xtick', labelsize=30)
181.  # plt.rc('ytick', labelsize=30)
182.
183.
184.  # #10 000 TRAINING STEPS REWARD
185.  # tenthousend_steps = []
186.  # print(tenthousend_steps)
187.
188.  # ywaarde1 = np.load('reward_5_1.npy')
189.  # ywaarde2 = np.load('reward_5_2.npy')
190.  # ywaarde3 = np.load('reward_5_3.npy')
191.  # ywaarde4 = np.load('reward_5_7.npy')
192.  # ywaarde5 = np.load('reward_5_8.npy')
193.  # ywaarde6 = np.load('reward_5_9.npy')
194.  # ywaarde7 = np.load('reward_5_13.npy')
195.  # ywaarde8 = np.load('reward_5_14.npy')
196.  # ywaarde9 = np.load('reward_5_15.npy')
197.  # ywaarde10 = np.load('reward_5_16.npy')
198.  # for i in ywaarde1:
199.  #     tenthousend_steps.append(i)
200.  # for i in ywaarde2:
201.  #     tenthousend_steps.append(i)
202.  # for i in ywaarde3:
203.  #     tenthousend_steps.append(i)
204.  # for i in ywaarde4:
```

```
205.  #     tenthousend_steps.append(i)
206.  # for i in ywaarde5:
207.  #     tenthousend_steps.append(i)
208.  # for i in ywaarde6:
209.  #     tenthousend_steps.append(i)
210.  # for i in ywaarde7:
211.  #     tenthousend_steps.append(i)
212.  # for i in ywaarde8:
213.  #     tenthousend_steps.append(i)
214.  # for i in ywaarde9:
215.  #     tenthousend_steps.append(i)
216.  # for i in ywaarde10:
217.  #     tenthousend_steps.append(i)
218.
219.  # print(tenthousend_steps)
220.  # avg = np.mean(tenthousend_steps)
221.  # print(avg)
222.  # stdev = np.std(tenthousend_steps)
223.  # print(stdev)
224.
225.  # plt.plot(range(50),ywaarde1,'r-', label="1")
226.  # plt.plot(range(50),ywaarde2,'b-', label="2")
227.  # plt.plot(range(50),ywaarde3,'y-', label="3")
228.  # plt.plot(range(50),ywaarde4,'g-', label="4")
229.  # plt.plot(range(50),ywaarde5,'m-', label="5")
230.  # plt.plot(range(50),ywaarde6,'k-', label="6")
231.  # plt.plot(range(50),ywaarde7,'r--', label="7")
232.  # plt.plot(range(50),ywaarde8,'b--', label="8")
233.  # plt.plot(range(50),ywaarde9,'y--', label="9")
234.  # plt.plot(range(50),ywaarde10,'g--', label="10")
235.
236.  # plt.xlabel('Time (minutes)', size = 15)
237.  # plt.xlim([0,50])
238.  # # plt.ylim([1,1.20])
239.  # plt.ylabel('Reward', size = 15)
240.  # plt.legend(prop={'size': 15})
241.  # plt.title('Influence of amount of training steps', size= 15)
242.  # plt.rc('xtick', labelsize=15)
243.  # plt.rc('ytick', labelsize=15)
244.
245.
246.
247.
248.
249.  # #50 000 TRAINING STEPS REWARD
250.  # tenthousend_steps = []
251.  # print(tenthousend_steps)
252.
253.  # ywaarde1 = np.load('reward_5_4.npy')
254.  # ywaarde2 = np.load('reward_5_5.npy')
255.  # ywaarde3 = np.load('reward_5_6.npy')
256.  # ywaarde4 = np.load('reward_5_10.npy')
257.  # ywaarde5 = np.load('reward_5_11.npy')
258.  # ywaarde6 = np.load('reward_5_12.npy')
259.  # ywaarde7 = np.load('reward_5_17.npy')
260.  # ywaarde8 = np.load('reward_5_18.npy')
261.  # ywaarde9 = np.load('reward_5_19.npy')
262.  # ywaarde10 = np.load('reward_5_20.npy')
263.  # for i in ywaarde1:
264.  #     tenthousend_steps.append(i)
265.  # for i in ywaarde2:
266.  #     tenthousend_steps.append(i)
267.  # for i in ywaarde3:
268.  #     tenthousend_steps.append(i)
269.  # for i in ywaarde4:
270.  #     tenthousend_steps.append(i)
271.  # for i in ywaarde5:
272.  #     tenthousend_steps.append(i)
273.  # for i in ywaarde6:
274.  #     tenthousend_steps.append(i)
```

```
# for i in ywaarde7:
#     tenthousend_steps.append(i)
# for i in ywaarde8:
#     tenthousend_steps.append(i)
# for i in ywaarde9:
#     tenthousend_steps.append(i)
# for i in ywaarde10:
#     tenthousend_steps.append(i)

# print(tenthousend_steps)
# avg = np.mean(tenthousend_steps)
# print(avg)
# stdev = np.std(tenthousend_steps)
# print(stdev)

# plt.plot(range(50),ywaarde1,'r-', label="1")
# plt.plot(range(50),ywaarde2,'b-', label="2")
# plt.plot(range(50),ywaarde3,'y-', label="3")
# plt.plot(range(50),ywaarde4,'g-', label="4")
# plt.plot(range(50),ywaarde5,'m-', label="5")
# plt.plot(range(50),ywaarde6,'k-', label="6")
# plt.plot(range(50),ywaarde7,'r--', label="7")
# plt.plot(range(50),ywaarde8,'b--', label="8")
# plt.plot(range(50),ywaarde9,'y--', label="9")
# plt.plot(range(50),ywaarde10,'g--', label="10")

# plt.xlabel('Time (minutes)', size = 15)
# plt.xlim([0,50])
# # plt.ylim([1,1.20])
# plt.ylabel('Reward', size = 15)
# plt.legend(prop={'size': 15})
# plt.title('Influence of amount of training steps', size= 15)
# plt.rc('xtick', labelsize=15)
# plt.rc('ytick', labelsize=15)




# #100 000 TRAINING STEPS REWARD
# tenthousend_steps = []
# print(tenthousend_steps)

# ywaarde1 = np.load('reward_4_1.npy')
# ywaarde2 = np.load('reward_4_2.npy')
# ywaarde3 = np.load('reward_4_3.npy')
# ywaarde4 = np.load('reward_4_4.npy')
# ywaarde5 = np.load('reward_4_5.npy')
# ywaarde6 = np.load('reward_4_6.npy')
# ywaarde7 = np.load('reward_4_7.npy')
# ywaarde8 = np.load('reward_4_8.npy')
# ywaarde9 = np.load('reward_4_10.npy')
# ywaarde10 = np.load('reward_4_11.npy')
# for i in ywaarde1:
#     tenthousend_steps.append(i)
# for i in ywaarde2:
#     tenthousend_steps.append(i)
# for i in ywaarde3:
#     tenthousend_steps.append(i)
# for i in ywaarde4:
#     tenthousend_steps.append(i)
# for i in ywaarde5:
#     tenthousend_steps.append(i)
# for i in ywaarde6:
#     tenthousend_steps.append(i)
# for i in ywaarde7:
#     tenthousend_steps.append(i)
# for i in ywaarde8:
#     tenthousend_steps.append(i)
```

```
# for i in ywaarde9:
#     tenthousend_steps.append(i)
# for i in ywaarde10:
#     tenthousend_steps.append(i)

# print(tenthousend_steps)
# avg = np.mean(tenthousend_steps)
# print(avg)
# stdev = np.std(tenthousend_steps)
# print(stdev)

# plt.plot(range(50),ywaarde1,'r-', label="1")
# plt.plot(range(50),ywaarde2,'b-', label="2")
# plt.plot(range(50),ywaarde3,'y-', label="3")
# plt.plot(range(50),ywaarde4,'g-', label="4")
# plt.plot(range(50),ywaarde5,'m-', label="5")
# plt.plot(range(50),ywaarde6,'k-', label="6")
# plt.plot(range(50),ywaarde7,'r--', label="7")
# plt.plot(range(50),ywaarde8,'b--', label="8")
# plt.plot(range(50),ywaarde9,'y--', label="9")
# plt.plot(range(50),ywaarde10,'g--', label="10")

# plt.xlabel('Time (minutes)', size = 15)
# plt.xlim([0,50])
# # plt.ylim([1,1.20])
# plt.ylabel('Reward', size = 15)
# plt.legend(prop={'size': 15})
# plt.title('Influence of amount of training steps', size= 15)
# plt.rc('xtick', labelsize=15)
# plt.rc('ytick', labelsize=15)



# #10 000 TRAINING STEPS CONCENTRATION
# tenthousend_steps = []
# print(tenthousend_steps)

# ywaarde1 = np.load('concb_5_1.npy')
# ywaarde2 = np.load('concb_5_2.npy')
# ywaarde3 = np.load('concb_5_3.npy')
# ywaarde4 = np.load('concb_5_7.npy')
# ywaarde5 = np.load('concb_5_8.npy')
# ywaarde6 = np.load('concb_5_9.npy')
# ywaarde7 = np.load('concb_5_13.npy')
# ywaarde8 = np.load('concb_5_14.npy')
# ywaarde9 = np.load('concb_5_15.npy')
# ywaarde10 = np.load('concb_5_16.npy')
# for i in ywaarde1:
#     tenthousend_steps.append(i)
# for i in ywaarde2:
#     tenthousend_steps.append(i)
# for i in ywaarde3:
#     tenthousend_steps.append(i)
# for i in ywaarde4:
#     tenthousend_steps.append(i)
# for i in ywaarde5:
#     tenthousend_steps.append(i)
# for i in ywaarde6:
#     tenthousend_steps.append(i)
# for i in ywaarde7:
#     tenthousend_steps.append(i)
# for i in ywaarde8:
#     tenthousend_steps.append(i)
# for i in ywaarde9:
#     tenthousend_steps.append(i)
# for i in ywaarde10:
#     tenthousend_steps.append(i)

# print(tenthousend_steps)
```

```
# avg = np.mean(tenthousend_steps)
# print(avg)
# stdev = np.std(tenthousend_steps)
# print(stdev)

# plt.plot(range(51),ywaarde1,'r-', label="1")
# plt.plot(range(51),ywaarde2,'b-', label="2")
# plt.plot(range(51),ywaarde3,'y-', label="3")
# plt.plot(range(51),ywaarde4,'g-', label="4")
# plt.plot(range(51),ywaarde5,'m-', label="5")
# plt.plot(range(51),ywaarde6,'k-', label="6")
# plt.plot(range(51),ywaarde7,'r--', label="7")
# plt.plot(range(51),ywaarde8,'b--', label="8")
# plt.plot(range(51),ywaarde9,'y--', label="9")
# plt.plot(range(51),ywaarde10,'g--', label="10")

# plt.xlabel('Time (minutes)', size = 15)
# plt.xlim([0,50])
# plt.ylim([1,1.20])
# plt.ylabel('Reward', size = 15)
# plt.legend(prop={'size': 15})
# plt.title('Influence of amount of training steps', size= 15)
# plt.rc('xtick', labelsize=15)
# plt.rc('ytick', labelsize=15)




# #50 000 TRAINING STEPS CONCENTRATION
# tenthousend_steps = []
# print(tenthousend_steps)

# ywaarde1 = np.load('concb_5_4.npy')
# ywaarde2 = np.load('concb_5_5.npy')
# ywaarde3 = np.load('concb_5_6.npy')
# ywaarde4 = np.load('concb_5_10.npy')
# ywaarde5 = np.load('concb_5_11.npy')
# ywaarde6 = np.load('concb_5_12.npy')
# ywaarde7 = np.load('concb_5_17.npy')
# ywaarde8 = np.load('concb_5_18.npy')
# ywaarde9 = np.load('concb_5_19.npy')
# ywaarde10 = np.load('concb_5_20.npy')
# for i in ywaarde1:
#     tenthousend_steps.append(i)
# for i in ywaarde2:
#     tenthousend_steps.append(i)
# for i in ywaarde3:
#     tenthousend_steps.append(i)
# for i in ywaarde4:
#     tenthousend_steps.append(i)
# for i in ywaarde5:
#     tenthousend_steps.append(i)
# for i in ywaarde6:
#     tenthousend_steps.append(i)
# for i in ywaarde7:
#     tenthousend_steps.append(i)
# for i in ywaarde8:
#     tenthousend_steps.append(i)
# for i in ywaarde9:
#     tenthousend_steps.append(i)
# for i in ywaarde10:
#     tenthousend_steps.append(i)

# print(tenthousend_steps)
# avg = np.mean(tenthousend_steps)
# print(avg)
# stdev = np.std(tenthousend_steps)
# print(stdev)

```

```
485.  # plt.plot(range(50),ywaarde1,'r-', label="1")
486.  # plt.plot(range(50),ywaarde2,'b-', label="2")
487.  # plt.plot(range(50),ywaarde3,'y-', label="3")
488.  # plt.plot(range(50),ywaarde4,'g-', label="4")
489.  # plt.plot(range(50),ywaarde5,'m-', label="5")
490.  # plt.plot(range(50),ywaarde6,'k-', label="6")
491.  # plt.plot(range(50),ywaarde7,'r--', label="7")
492.  # plt.plot(range(50),ywaarde8,'b--', label="8")
493.  # plt.plot(range(50),ywaarde9,'y--', label="9")
494.  # plt.plot(range(50),ywaarde10,'g--', label="10")
495.
496.  # plt.xlabel('Time (minutes)', size = 15)
497.  # plt.xlim([0,50])
498.  # # plt.ylim([1,1.20])
499.  # plt.ylabel('Reward', size = 15)
500.  # plt.legend(prop={'size': 15})
501.  # plt.title('Influence of amount of training steps', size= 15)
502.  # plt.rc('xtick', labelsize=15)
503.  # plt.rc('ytick', labelsize=15)
504.
505.
506.
507.
508.
509.
510.  # #100 000 TRAINING STEPS CONCENTRATION
511.  # tenthousend_steps = []
512.  # print(tenthousend_steps)
513.
514.  # ywaarde1 = np.load('concb_4_1.npy')
515.  # ywaarde2 = np.load('concb_4_2.npy')
516.  # ywaarde3 = np.load('concb_4_3.npy')
517.  # ywaarde4 = np.load('concb_4_4.npy')
518.  # ywaarde5 = np.load('concb_4_5.npy')
519.  # ywaarde6 = np.load('concb_4_6.npy')
520.  # ywaarde7 = np.load('concb_4_7.npy')
521.  # ywaarde8 = np.load('concb_4_8.npy')
522.  # ywaarde9 = np.load('concb_4_10.npy')
523.  # ywaarde10 = np.load('concb_4_11.npy')
524.  # for i in ywaarde1:
525.  #     tenthousend_steps.append(i)
526.  # for i in ywaarde2:
527.  #     tenthousend_steps.append(i)
528.  # for i in ywaarde3:
529.  #     tenthousend_steps.append(i)
530.  # for i in ywaarde4:
531.  #     tenthousend_steps.append(i)
532.  # for i in ywaarde5:
533.  #     tenthousend_steps.append(i)
534.  # for i in ywaarde6:
535.  #     tenthousend_steps.append(i)
536.  # for i in ywaarde7:
537.  #     tenthousend_steps.append(i)
538.  # for i in ywaarde8:
539.  #     tenthousend_steps.append(i)
540.  # for i in ywaarde9:
541.  #     tenthousend_steps.append(i)
542.  # for i in ywaarde10:
543.  #     tenthousend_steps.append(i)
544.
545.  # print(tenthousend_steps)
546.  # avg = np.mean(tenthousend_steps)
547.  # print(avg)
548.  # stdev = np.std(tenthousend_steps)
549.  # print(stdev)
550.
551.  # plt.plot(range(50),ywaarde1,'r-', label="1")
552.  # plt.plot(range(50),ywaarde2,'b-', label="2")
553.  # plt.plot(range(50),ywaarde3,'y-', label="3")
554.  # plt.plot(range(50),ywaarde4,'g-', label="4")
```

```
555.  # plt.plot(range(50),ywaarde5,'m-', label="5")
556.  # plt.plot(range(50),ywaarde6,'k-', label="6")
557.  # plt.plot(range(50),ywaarde7,'r--', label="7")
558.  # plt.plot(range(50),ywaarde8,'b--', label="8")
559.  # plt.plot(range(50),ywaarde9,'y--', label="9")
560.  # plt.plot(range(50),ywaarde10,'g--', label="10")
561.
562.  # plt.xlabel('Time (minutes)', size = 15)
563.  # plt.xlim([0,50])
564.  # # plt.ylim([1,1.20])
565.  # plt.ylabel('Reward', size = 15)
566.  # plt.legend(prop={'size': 15})
567.  # plt.title('Influence of amount of training steps', size= 15)
568.  # plt.rc('xtick', labelsize=15)
569.  # plt.rc('ytick', labelsize=15)
570.
571.
572.
573.
574.  # #DIFFERENT TRAINING STEPS
575.
576.  # ywaarde1 = np.load('reward_2_11.npy')
577.  # ywaarde2 = np.load('reward_3_11.npy')
578.  # print(ywaarde1)
579.  # print(ywaarde2)
580.
581.  # waarde = []
582.  # for i in ywaarde1:
583.  #     waarde.append(i)
584.  # for i in ywaarde2:
585.  #     waarde.append(i)
586.
587.  # avg = np.mean(waarde)
588.  # print(avg)
589.
590.  # stdev = np.std(waarde)
591.  # print(stdev)
592.
593.
594.
595.
596.
597.  # #MAKING BAR CHART FOR TRAINING STEPS
598.  # import numpy as np
599.  # import matplotlib.pyplot as plt
600.
601.  # # data to plot
602.  # n_groups = 8
603.  # means_10_000 = (0,0,0,0,0,0,4,6)
604.  # means_50_000 = (2,2,0,0,0,1,2,3)
605.  # means_100_000 = (2,1,1,0,0,0,2,4)
606.
607.  # # create plot
608.  # fig, ax = plt.subplots()
609.  # index = np.arange(n_groups)
610.  # bar_width = 0.25
611.  # opacity = 1
612.
613.  # rects1 = plt.bar(index, means_10_000, bar_width,
614.  # alpha=opacity,
615.  # color='r',
616.  # label='10 000')
617.
618.  # rects2 = plt.bar(index + bar_width, means_50_000, bar_width,
619.  # alpha=opacity,
620.  # color='b',
621.  # label='50 000')
622.
623.  # rects3 = plt.bar(index + bar_width + bar_width, means_100_000, bar_width,
624.  # alpha=opacity,
```

```
625.  # color='g',
626.  # label='100 000')
627.
628.  # plt.xlabel('Reward')
629.  # plt.ylabel('Counts')
630.  # plt.title('Reward based on different training steps')
631.  # plt.xticks(index + bar_width, ('-7->-6', '-6->-5', '-5->-4', '-4->-3', '-3->-2', '-2-
    >-1', '-1->0', '0->1'))
632.  # plt.legend()
633.
634.  # plt.tight_layout()
635.  # plt.show()
636.
637.
638.
639.
640.
641.
642.  #100 000 TRAINING STEPS 0.1
643.
644.  # ywaarde1 = np.load('concb_6_2.npy')
645.  # ywaarde2 = np.load('concb_6_3.npy')
646.  # ywaarde3 = np.load('concb_6_4.npy')
647.  # # ywaarde1 = np.load('reward_6_2.npy')
648.  # # ywaarde2 = np.load('reward_6_3.npy')
649.  # # ywaarde3 = np.load('reward_6_4.npy')
650.  # print(ywaarde3)
651.
652.  # plt.plot(range(501),ywaarde1,'r-', label="1")
653.  # plt.plot(range(501),ywaarde2,'b-', label="2")
654.  # plt.plot(range(501),ywaarde3,'y-', label="3")
655.
656.  # plt.xlabel('0.1 * Time (minutes)', size = 30)
657.  # plt.xlim([0,500])
658.  # plt.ylim([0.8,1.45])
659.  # plt.ylabel('Concentration product B (kmol/m3)', size = 30)
660.  # plt.legend(prop={'size': 30})
661.  # plt.title('Influence of time step', size= 50)
662.  # plt.rc('xtick', labelsize=30)
663.  # plt.rc('ytick', labelsize=30)
664.
665.
666.
667.
668.
669.
670.
671.
672.  # #100 000 TRAINING STEPS 0.2
673.
674.  # ywaarde1 = np.load('reward_6_5.npy')
675.  # ywaarde2 = np.load('reward_6_6.npy')
676.  # ywaarde3 = np.load('reward_6_7.npy')
677.  # print(ywaarde3)
678.
679.  # plt.plot(range(251),ywaarde1,'r-', label="1")
680.  # plt.plot(range(251),ywaarde2,'b-', label="2")
681.  # plt.plot(range(251),ywaarde3,'y-', label="3")
682.
683.  # plt.xlabel('0.2 * Time (minutes)', size = 30)
684.  # plt.xlim([0,250])
685.  # plt.ylim([0.8,1.45])
686.  # plt.ylabel('Concentration product B (kmol/m3)', size = 30)
687.  # plt.legend(prop={'size': 30})
688.  # plt.title('Influence of time step', size= 50)
689.  # plt.rc('xtick', labelsize=30)
690.  # plt.rc('ytick', labelsize=30)
691.
692.
693.
```

```
694.
695.
696.
697.
698.
699.  # #100 000 TRAINING STEPS 0.5
700.
701.  # ywaarde1 = np.load('concb_6_8.npy')
702.  # ywaarde2 = np.load('concb_6_9.npy')
703.  # ywaarde3 = np.load('concb_6_10.npy')
704.  # print(ywaarde3)
705.
706.  # plt.plot(range(100),ywaarde1,'r-', label="1")
707.  # plt.plot(range(100),ywaarde2,'b-', label="2")
708.  # plt.plot(range(100),ywaarde3,'y-', label="3")
709.
710.  # plt.xlabel('0.5 * Time (minutes)', size = 30)
711.  # plt.xlim([0,100])
712.  # # plt.ylim([0.8,1.45])
713.  # plt.ylabel('Reward', size = 30)
714.  # plt.legend(prop={'size': 30})
715.  # plt.title('Influence of time step', size= 50)
716.  # plt.rc('xtick', labelsize=30)
717.  # plt.rc('ytick', labelsize=30)
718.
719.
720.
721.
722.
723.
724.
725.
726.
727.
728.  # #100 000 TRAINING STEPS 1
729.
730.  # ywaarde1 = np.load('concb_4_1.npy')
731.  # ywaarde2 = np.load('concb_4_4.npy')
732.  # ywaarde3 = np.load('concb_4_5.npy')
733.
734.  # plt.plot(range(51),ywaarde1,'r-', label="1")
735.  # plt.plot(range(51),ywaarde2,'b-', label="2")
736.  # plt.plot(range(51),ywaarde3,'y-', label="3")
737.
738.  # plt.xlabel('Time (minutes)', size = 30)
739.  # plt.xlim([0,50])
740.  # plt.ylim([0.8,1.45])
741.  # plt.ylabel('Concentration product B (kmol/m3)', size = 30)
742.  # plt.legend(prop={'size': 30})
743.  # plt.title('Influence of time step', size= 50)
744.  # plt.rc('xtick', labelsize=30)
745.  # plt.rc('ytick', labelsize=30)
746.
747.
748.
749.
750.
751.
752.  # # MAKING POSTER GRAPH
753.
754.  # # ywaarde1 = np.load('concb_4_1.npy')
755.  # ywaarde2 = np.load('concb_4_4.npy')
756.  # ywaarde3 = np.load('concb_4_5.npy')
757.  # # ywaarde4 = np.load('concb_4_11.npy')
758.  # # ywaarde5 = np.load('concb_4_4.npy')
759.  # ywaarde8 = np.load('reward_4_4.npy')
760.  # ywaarde9 = np.load('reward_4_5.npy')
761.
762.  # fig,ax = plt.subplots()
763.  # # plt.plot(range(51),ywaarde1,'ro', label="1")
```

```
764.  # ax.plot(range(51),ywaarde2,'k--', label="Worst performance")
765.  # ax.plot(range(51),ywaarde3,'k-', label="Best performance")
766.  # # plt.plot(range(51),ywaarde4,'go', label="4")
767.  # # plt.plot(range(51),ywaarde5,'m-', label="5")
768.  # # plt.plot(range(51),ywaarde6,'k-', label="6")
769.  # # plt.plot(range(51),ywaarde7,'r--', label="7")
770.  # ax2 = ax.twinx()
771.  # ax2.plot(range(50),ywaarde8,'r--', label="Worst performance")
772.  # ax2.plot(range(50),ywaarde9,'r-', label="Best performance")
773.  # # plt.plot(range(51),ywaarde10,'g--', label="10")
774.
775.  # ax.set_xlabel('Time (minutes)', size = 30)
776.  # ax.set_xlim([0,30])
777.  # # plt.ylim([0.8,1.45])
778.  # ax.set_ylabel('Concentration (kmol/m3)', size = 30)
779.  # ax.set_ylim([0.8,1.45])
780.  # ax2.set_ylim([-3,1.5])
781.  # ax2.set_ylabel('Reward', size = 30)
782.  # ax2.yaxis.label.set_color('red')
783.  # ax.legend(prop={'size': 30})
784.  # ax2.legend(prop={'size': 30})
785.  # plt.title('Controlling concentration product B', size= 50)
786.  # plt.rc('xtick', labelsize=30)
787.  # plt.rc('ytick', labelsize=30)
788.
789.
790.
791.
792.
793.
794.
795.
796.
797.  # #graph making poster
798.  # ywaarde4 = []
799.  # ywaarde5 = []
800.  # ywaarde12 = []
801.  # p = 51
802.  # x = 0
803.  # while x < p:
804.  #     x += 1
805.  #     ywaarde4.append(1.15)
806.  #     ywaarde5.append(1.05)
807.  #     ywaarde12.append(405)
808.
809.
810.  # ywaarde2 = np.load('concb_4_4.npy')
811.  # ywaarde3 = np.load('concb_4_5.npy')
812.
813.
814.  # ywaarde9 =  np.load('action_value_4_4.npy')
815.  # ywaarde10 = np.load('action_value_4_5.npy')
816.
817.
818.  # fig,ax = plt.subplots(1,2)
819.  # ax[0].plot(range(51),ywaarde2,'g--', label="Worst performance")
820.  # ax[0].plot(range(51),ywaarde3,'k-', label="Best performance")
821.  # ax[0].plot(range(51),ywaarde4,'r-', label="upper limit")
822.  # ax[0].plot(range(51),ywaarde5,'b-', label="lower limit")
823.  # ax[0].set_xlabel('Time (minutes)', size = 30)
824.  # ax[0].set_xlim([0,30])
825.  # ax[0].set_ylabel('Concentration (kmol/m3)', size = 30)
826.  # ax[0].set_ylim([0.9,1.40])
827.  # ax[0].legend(prop={'size': 30})
828.  # ax[0].set_title('Controlling concentration product B', size= 30)
829.  # ax[1].plot(range(50),ywaarde9,'g--', label="Worst performance")
830.  # ax[1].plot(range(50),ywaarde10,'k-', label="Best performance")
831.  # ax[1].plot(range(51),ywaarde12,'r-', label="limit temperature")
832.  # ax[1].set_xlabel('Time (minutes)', size = 30)
833.  # ax[1].set_xlim([0,30])
```

```
834.  # ax[1].set_ylabel('Temperature (K)', size = 30)
835.  # ax[1].legend(prop={'size': 30})
836.  # ax[1].set_title('Controlling cooling jacket temperature (Tc)', size= 30)
837.  # plt.rc('xtick', labelsize=30)
838.  # plt.rc('ytick', labelsize=30)
839.
840.
841.
842.  # training_steps = [1000,10000,20000,30000,40000,50000,60000,70000,80000,90000,100000]
843.  # reward = [1,1,1,1,1,1,1,1,1,1,1]
844.
845.
846.  # value1 = [-3.762,0.128,-0.448,-3.413,-2.964,0.299,0.885,-2.161,0.475,0.582,0.808]
847.  # stdev = [0.614,0.869,0.297,2.914,2.282,1.012,0.533,3.088,0.862,0.702,0.606]
848.  # maxi = np.add(value1, stdev)
849.  # mini = np.subtract(value1, stdev)
850.
851.  # plt.errorbar(training_steps, value1, stdev, ecolor='k', linestyle='None',
    marker='None')
852.
853.  # plt.plot(training_steps, value1, 'ks--', label= "average reward + st. dev.")
854.  # # plt.plot(training_steps, maxi, 'kx', label="standard deviation")
855.  # # plt.plot(training_steps, mini, 'kx', )
856.  # plt.plot(training_steps, reward, 'r-', label="reward=1")
857.  # plt.legend(prop={'size':30})
858.  # plt.xlabel('Training_steps', size=30)
859.  # plt.ylabel('Reward', size=30)
860.  # plt.title('Influence amount of training steps', size=50)
861.  # plt.rc('xtick', labelsize=30)
862.  # plt.rc('ytick', labelsize=30)
863.
864.
865.
866.  # aswaarde = np.load('action_value_4_1.npy')
867.
868.
869.  # plt.plot(range(50), aswaarde, 'r')
870.
```