Bachelor-Thesis an der Hochschule Luzern - Technik & Architektur

Titel	**KI-basierte Reglerauslegung – Reinforcement Learning für dynamische Systeme**
Diplomandin/Diplomand	**Arnold, Tim**
Bachelor-Studiengang	**Bachelor Maschinentechnik**
Semester	**FS22**
Dozentin/Dozent	**Prof. Dr. Ulf Christian Müller**
Expertin/Experte	**Dr. Schlienger, Joel**

Abstract Deutsch
Das Ziel der vorliegenden Bachlorarbeit war es, erste Erfahrungen mit Reinforcement Learning und der Kopplung mit Modelica Systemmodellen zu machen. Dazu wurde ein Showcase erstellt, welcher die Regelung eines Systems selbstständig mithilfe eines Systemmodells erlernen soll. Das Ergebnis wurde evaluiert und erste Erkenntnisse dokumentiert. Die vorliegende Arbeit gibt ausserdem einen fundamentalen Überblick zur Thematik und verschafft das nötige theoretische Grundwissen. Somit soll die Arbeit den Einstieg für weitere Untersuchungen und Versuche erleichtern.

Abstract Englisch
The aim of this bachelor thesis was to gain first experiences with reinforcement learning and the coupling with Modelica system models. For this purpose, a showcase was created, which should learn self-sufficient how to control a system with the help of a system model. The result was evaluated and first cognizance were documented. The present work also gives a fundamental overview of the topic and provides the necessary theoretical knowledge. Thus, the work should facilitate the entry for further investigations and tests.

Ort, Datum Luzern, 10.06.2022

Content

Introduction

Nowadays, artificial intelligence is a highly spread topic and is tried to be used in many different applications. Reinforcement learning is a very promising concept for controlling tasks. At the same time, digital twins such as system models are already common for numerous applications. Modelica allows the creation of complex multidomain system models with relatively low effort. Those two topics are predestined to be combined since reinforcement learning relies on a system to learn from. Thanks to using a system model, costs can be saved very efficiently, as compared to a real experimental setup, the training time and expense for changes during the design process can be greatly reduced.

The aim of this work is to gain first experiences on applying reinforcement learning to physical system models with Modelica. This shall be reached by creating a showcase. Those first experiences should provide a basis for future investigation and applications while also allowing a first overview of the opportunities and challenges.

The paper will be structured as follows: First, the fundamental theory and classifications are provided which are essential to understand the following topics. Afterwards, the toolchain used for the showcase is explained, and determinants of the application are defined. In the last chapters, the proceed of the created showcase is described, which includes experienced challenges and findings. It will also slightly go into evaluation. The final created showcase is a system of a double pendulum which shall be swung up and balanced to an upright position, starting from a hanging position. This result was reached after evading some difficulties during the process.

1 Basics

1.1 Artificial Intelligence

The term AI defines intelligent behaviour performed by machines. Due to the fact that intelligence is not clearly defined, AI is hard to define as well. Nevertheless, it is used to describe the ability of perceiving an environment and reaching a certain goal. This includes the ability of developing behaviours to achieve these goals with the help of perception, cognition, and action. AI is strictly separated from cognitive science which tries to build a model of human behaviour or human intelligence. Since the term AI is not clearly defined and some algorithms work with similar mechanics, there is no clear subclass structure which can be stated but some examples of subclasses are machine learning, natural language processing, and image recognition. [1]

1.2 Machine Learning

Machine Learning is the biggest subdivision of AI in which systems can learn from data without explicitly being programmed what to do. Like this, machine learning can generate knowledge, train algorithms, or identify connections, dependencies, and patterns from the data. There is always a statistical model which is trained with help of the training data. This model then can be applied to other data in order to create e.g. clusters, classifications, or predictions.

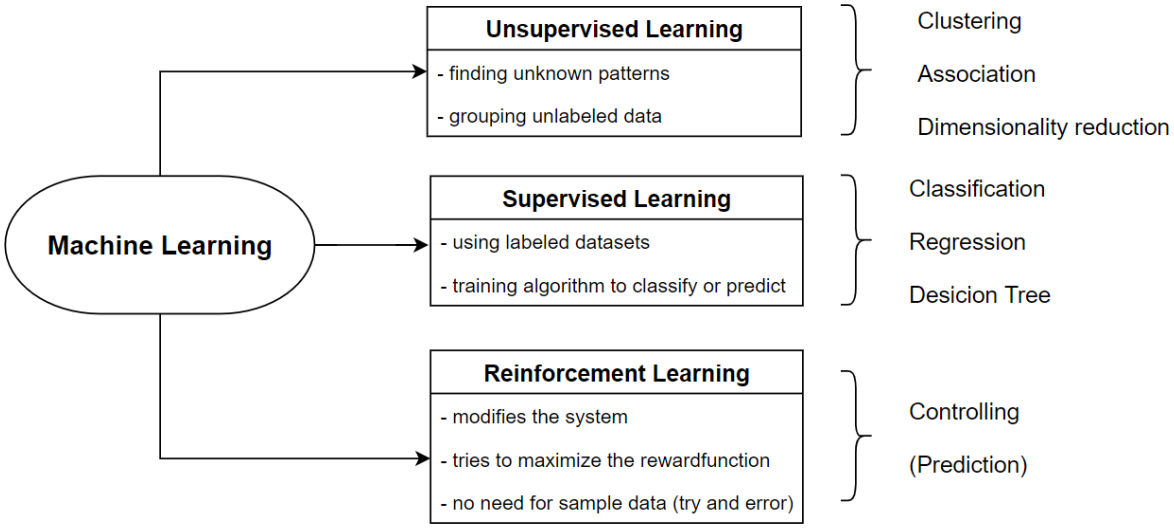

Figure 1: Machine Learning Overview

There are three main types of machine learning principles. Unsupervised learning is used to discover patterns and information in datasets which are not labelled previously. It can for example cluster data into groups or find relationships between variables in a given dataset. The mathematical goal is to find a function $f(x)$ for a given data x. [2]

Supervised learning uses labelled datasets and trains an algorithm to perform a regression or a classification. This can then be used to do regressions, decisions or to classify a new dataset according to the training set. The mathematical goal is to find the function $f(x)$ for a given data $y = f(x)$ while the parameters y and x are known. [2]

In reinforcement learning, there is no need to provide a training data. Instead, the algorithm generates the data himself by interaction through trial and error on the given system. The reward-function determines how well the performed decision is. This leads to a very open approach while the desired outcome is still clearly defined. The generated decision model can be used for controlling tasks (if the goal is to find the optimal policy) or to predict the value of taking actions which follow a certain policy (if the policy is predefined). This document will only focus on control problems for reinforcement learning. The general mathematical goal of reinforcement learning can be stated as having a function $y = f(x)$ with a given data x (→state) and the goal is to find the function $f(x)$ (→policy) which corresponds to a certain y (→action) for each x. This function shall be maximizing z (→return) while $z = f(y, x)$. [3]

1.3 Reinforcement Learning

In reinforcement learning for controlling tasks, an agent shall be trained to take the actions which result to the maximal reward. The observation gives him information about the current system state. Using the information from the observation, the policy decides which action is taken. The reinforcement learning algorithm changes that policy based on the observation, the action which was taken and the corresponding obtained reward. The goal is always to find the policy which leads to the maximal total reward. [3–7]

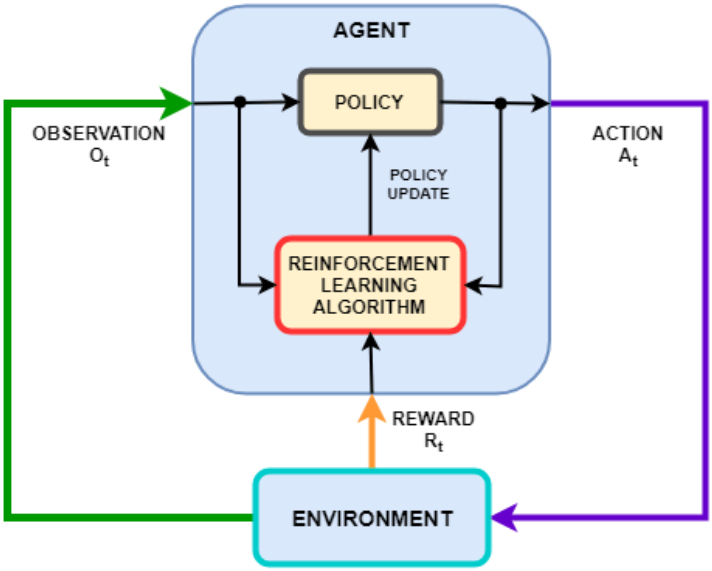

Figure 2 RL procedure [8]

1.3.1 Basic definitions

1.3.1.1 Environment

The state of the system is defined in the environment. A defined quantity of variables can be changed by means of the actions taken. Also, a defined quantity of variables from the state is outputted through the observation. The environment defines the behaviour of the system, thus the next state based on the current state and the action taken. Additionally, the reward is determined and outputted in the environment. The environment in this definition is not equal to the process. It just involves the process added by some other definitions like the reward-function. [9]

1.3.1.2 Agent

The agent is the controller and the controller designer of the system. The policy defines that controlling behaviour while the reinforcement learning algorithm optimizes the policy in order to gain the maximal reward from the possible actions which can be taken. To do this, the agent has information about the observed variables and the reward but not necessary about the whole system state. [9]

1.3.1.3 State

The state defines the whole status of the environment. The ‘current state’ describes the current situation while the ‘next state’ describes one step later after an additional action is taken. [9]

1.3.1.4 Observation

The observation includes all by the agent observed state-variables of the current state. The system can either be fully observed or it can be partially observed which describes the fact that not all state-variables are observed by the agent. The observation can be imagined as the sensor measurements. The observation-space predefines the possible range of those variables which ensures that the distribution of the agent is reasonable. [9]

1.3.1.5 Action

The action contains the values of the controlled variables at the current state. The agent can change those variables within the specified range defined in the action-space. It is set at the beginning of each step according to the policy and the current state. [9]

1.3.1.6 Reward

The obtained reward based on the reward-policy and the current state is determined at the end of each step. The total reward (or return) defines all cumulated rewards from each step until the end of the episode. The goal is to maximise the total reward while the highest step reward doesn’t always have to lead to the highest total reward. [9]

1.3.1.7 Policy

The policy is a mapping from each state to an action that decides how the agent acts at each specific state. It can either be deterministic (if a specific state leads certainly to a specific action) or stochastic (if a specific state leads only with a probability to a certain action). If the policy is stochastic, it is for a given state, a probability distribution over the set of possible actions. So, it gives the probability of picking a certain action at a certain state. The policy can be considered the controller of the system. [9]

1.3.1.8 Value

The value describes the expected return starting from the current state and following the policy. The Q-value (or action-value) describes the expected return starting from the current state by taking a certain action and then following the policy for the next states. Some algorithms don't follow the policy for reward estimation in future steps but take the most valuable actions known instead. This can make a slight difference. [9]

1.3.1.9 Episode

An episode includes all the steps with states, actions, and rewards until the fixed terminal state is reached. At the end of an episode, the system and the reward get reset to the initial state. [9]

1.3.2 Markov Decision Process

Most reinforcement learning environments (all used here) are pictured as Markov decision processes (MDP) to create a mathematical model, why it is crucial to understand this topic. A Markov process is a discrete stochastic process following the Markov property which simplifies the depiction of a continuous world. [10]

1.3.2.1 Markov Property

The Markov property describes a state which is only dependent on its immediate previous state and not on any states before that. Corresponding, the next state is only depending on the current state and not on any of the past states. Considering all system variables, this is true for the real world. [11]

1.3.2.2 Markov Process

The Markov process is a process with different states that obey the Markov property. Each state has a state transition probability (P) to jump to a different state. The Markov reward process (MRP) is additionally defined by rewards (R) which are received by reaching a certain state. The state transition probability is in this case purely stochastic and not controlled. [11]

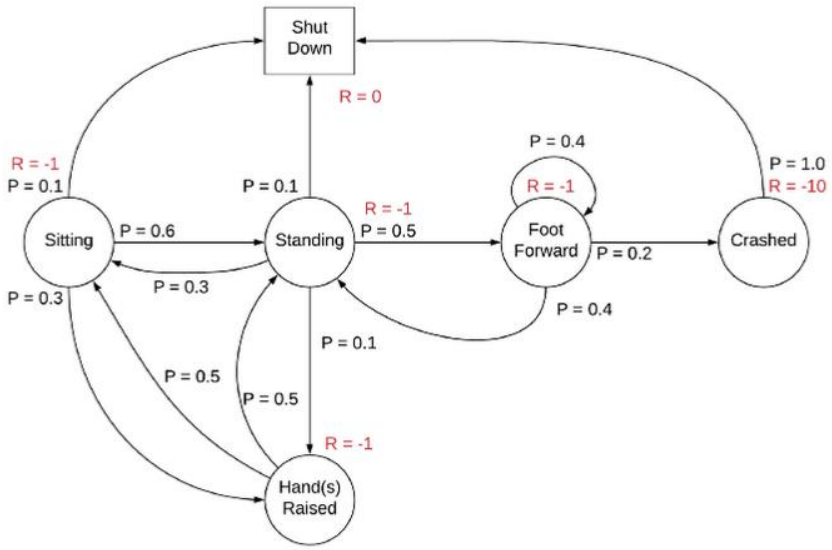

Figure 3 Example of an MRP [11]

1.3.2.3 Markov Decision Process

A Markov Decision Process is a MRP with actions. In this case, the state transition probability is defined by the stochastic base and additionally by actions which allow an agent to influence or control the state transition and with this the obtained reward. This is how the RL agent controls the system. [11]

1.3.3 Bellman equation

The RL agent must know which state is best at the current time. The goal is to collect the maximal total reward (return) over all state transitions until the end of the episode. This estimated return is calculated with the value function. The action with the highest Q-value should be taken while the state with the highest value is the most desirable state. Consequently, the policy is followed on the value function through the optimized value but also the value is given by the policy function. Therefore, you theoretically get the same optimal result whether you optimize the value or the policy function.

To calculate the value and the Q-value in an MDP, the bellman equation is used. It says that the current value is equal to the current reward plus the discounted values at the next steps following the agent's policy. This is similar to economic theories where an immediate reward is worth more than a reward later on because the future always holds some unknowns. [9]

Bellman equation for Value calculation: $\boldsymbol{V(s) = R(s) + \gamma \cdot \sum_{s' \epsilon S} P(s'|s)V(s')}$

Bellman equation for Q-Value calculation: $\boldsymbol{Q(s,a) = R(s,a) + \gamma \cdot \sum_{s' \epsilon S} P(s'|s,a)V(s')}$

1.3.3.1 MDP with Bellman equation

In the flow chart below, the whole system of an MDP with the use of the Bellman equation is apparent. The starting point is at state (s), while the loop at state (s+t) continuous until the end of the episode. The total obtained reward with the discount factor yields to the value or Q-value. If the probabilities of the state transitions are known, the Bellman equation can be calculated for each state without the use of any reinforcement learning.

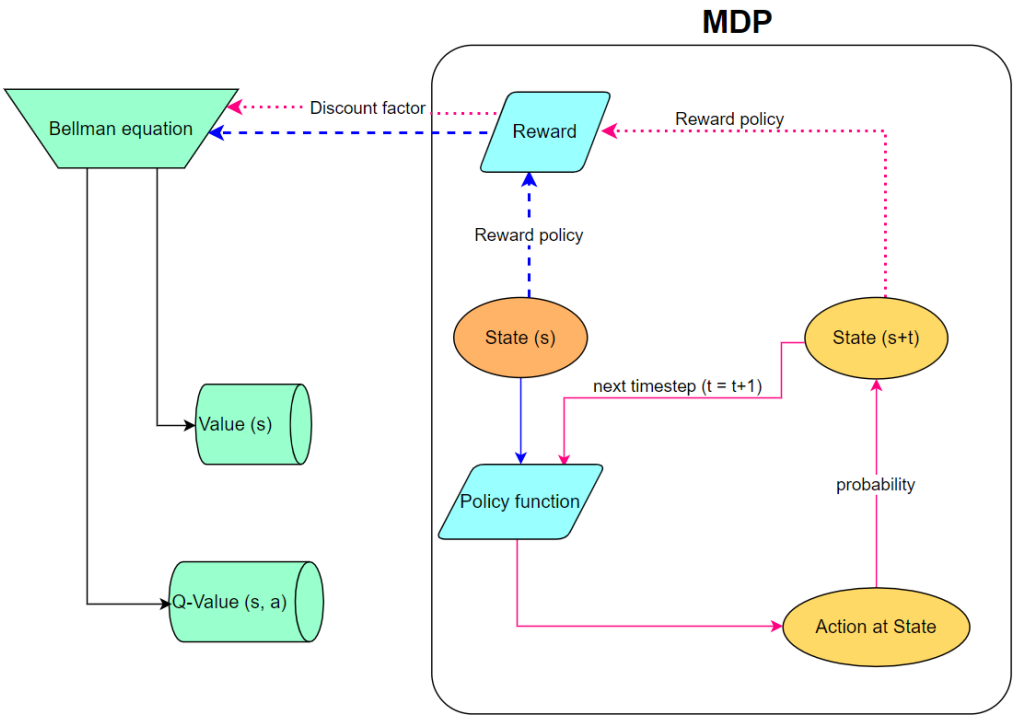

Figure 4: MDP with Bellman Equation

1.4 Deep Learning

Deep learning is machine learning with the use of a neural network with multiple hidden layers. Many advanced reinforcement learning agents use deep neural networks. This is then called deep reinforcement learning. The neural network is simply a mapping from input to output variables which in reinforcement learning can represent the policy or the value function. [12]

1.4.1 Neural Networks

A neural network is built with neurons and connectors. In case of the policy network, the input layer has a neuron for each observed variable while the output layer has a neuron for each action variable. The number of hidden neurons is independent on the system and can have any quantity of neurons in each layer and any quantity of layers. Each connector has a weight which simply multiplies the value of the connected neuron with a number and gives it to the next neuron. The neurons in the hidden layers then add all the numbers from the connectors and calculate a new number based on a defined activation function. This function can vary but often it is a squashing

function which normalizes and squashes the value for example between 0 and 1. This procedure repeats in each hidden layer. When all the hidden layers are passed, the output layer gets passed a value to each neuron which in this case represent the actions. [13]

In the graphic below, a simple neural network with two input and one output layer is illustrated. The circles represent the neurons while the arrows represent the connectors.

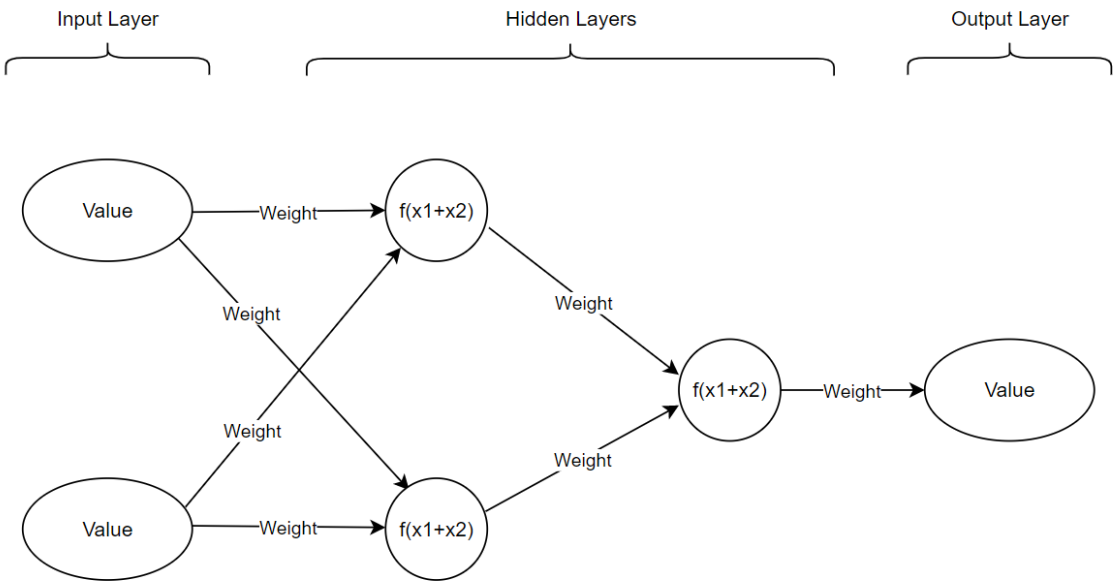

Figure 5: Neural Network Structure

The following graphic shows an example of this neural network how the input signals evolve to a corresponding output signal:

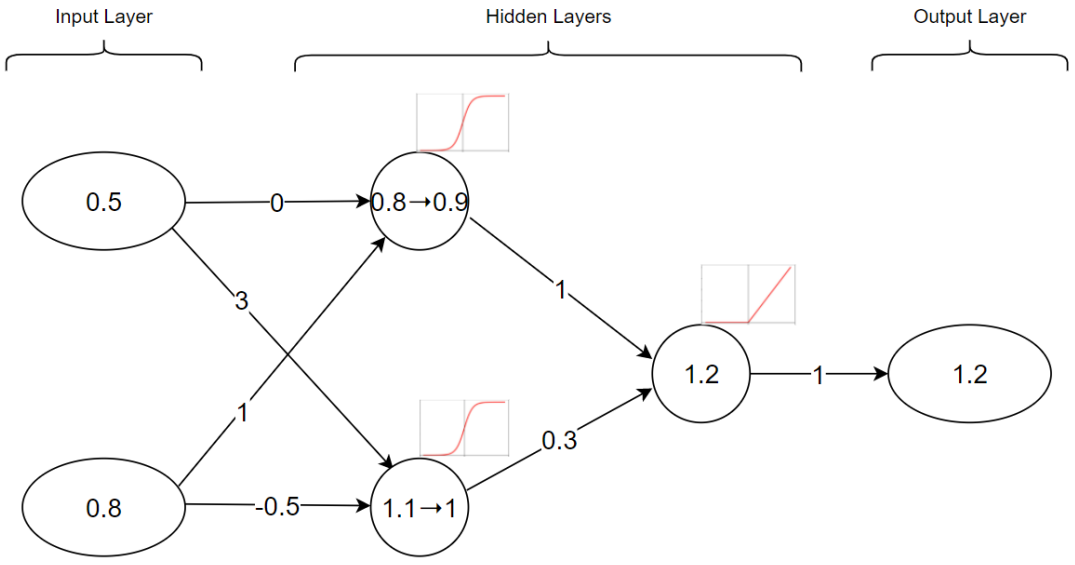

Figure 6: Neural Network Numeric Example

1.4.1.1 Bias

A bias is a scalar which shifts the value of a neuron output. This allows the neural network to make better data fitting and is often used in neural networks. It basically turns the neuron function from $z = f(w_1x_1 + w_2x_2 + w_3x_3)$ to $z = f(w_1x_1 + w_2x_2 + w_3x_3 + b)$. This parameter can as well as the weight also be changed during the learning process. [14]

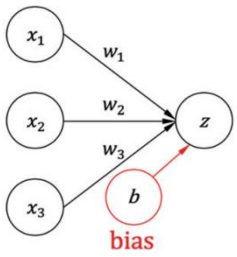

Figure 7: Bias Implementation [1]

1.4.2 Training a Neural Network

1.4.2.1 Optimization

To train a neural network, usually a loss function is defined which quantifies an error between the prediction and the target (perfect neural network model). The goal of parametrizing a neural network is mostly to minimize this loss value with optimization by changing the weights and biases. There are many different approaches for defining this loss function which will not be explained here. In reinforcement learning, the loss function is often a modified RL-algorithm method. To solve this optimization problem, also many different approaches are available such as back-propagation, gradient descent etc. [14]

1.4.2.2 Regularization

Because of the large number of layers and parameters, neural networks have a significant risk of overfitting, which makes the network perform extremely well on training data but weak on a slightly different dataset. This is also a problem in training because the neural network is mostly not perfect which always makes the training data a slightly different dataset too. To prevent this, there are some simple regularization methods available.

Weight decay penalties parameters with a high absolute value to encourage parameters with a lower absolute value. This results to a more generalized network with less dependency on a single observation.

Another problem of large neural networks is co-adaptation which means that neurons are extremely dependent on each other. To prevent this, during training, some weights are randomly set to zero which disconnects separate neurons. This makes the network less dependent on single neurons which makes it more robust and less likely to overfit.

There are more regularization methods available but those are the most important ones for reinforcement learning in physical systems. [14]

2 Background of Reinforcement Learning Algorithms

There are many different approaches for optimizing a policy to achieve the maximal reward by taking the best actions in each state of an MDP. The goal of reinforcement learning algorithms is always to optimize the policy which is essentially the controller so that the total reward at the end of the episode is as high as possible. So, the reinforcement learning optimization problem can be expressed as follows:

$$\pi_* = argmax_\pi[G(\pi)]$$

With π_* being the optimal policy and G being the total reward at the end of an episode. This problem is mostly solved by the help of the value or Q-value which describes the expected return at a certain state. [14]

The following chart categorizes RL-Algorithms in main groups. Some advanced algorithms however also try to combine those approaches.

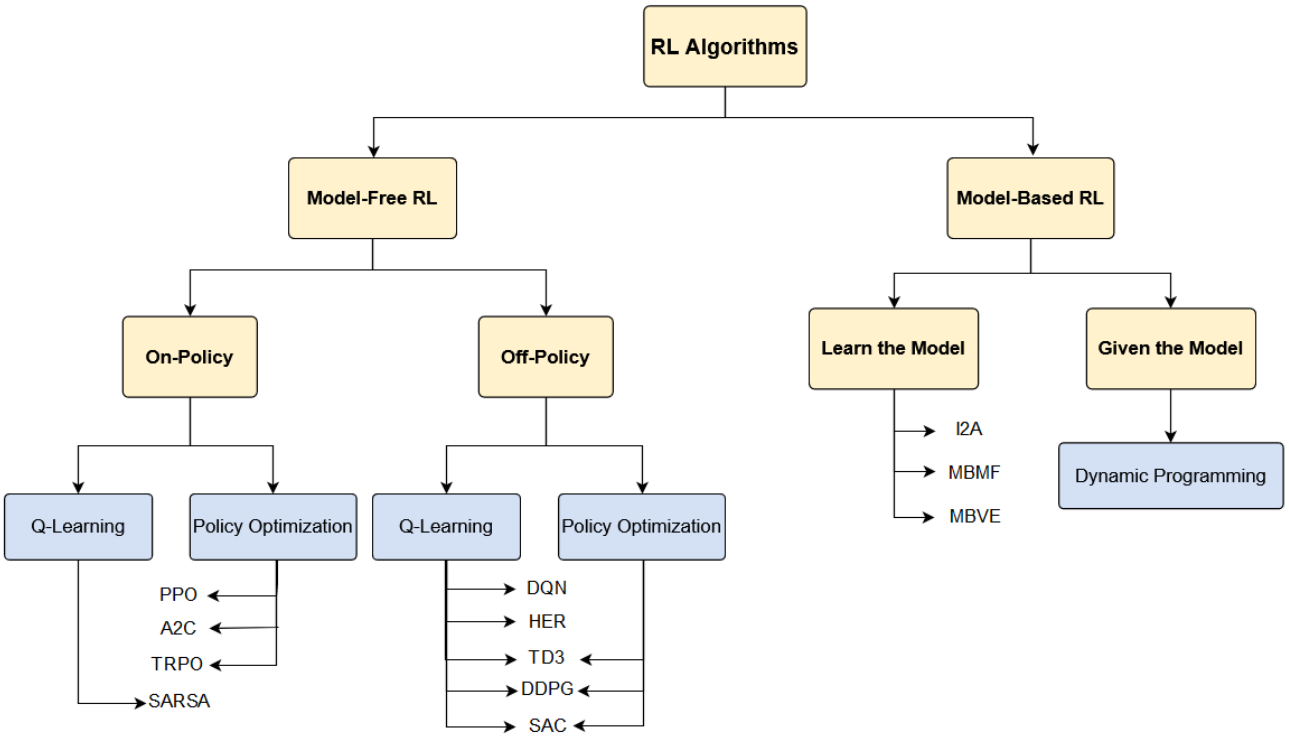

Figure 8: RL Algorithms Overview [15]

2.1 Dynamic Programming

If a complete and perfect model with all the transition probabilities of the MDP is known, there is no need for true reinforcement learning to solve the optimization problem. It can simply be calculated iteratively based on those transition probabilities. Even though there is rarely a perfect model of the MDP available, this still builds the foundation of reinforcement learning. There are two essential approaches to iteratively calculate the optimal policy.

2.1.1 Value Iteration

With value iteration, an initial value for each state is set first (e.g., 0). Then the Q-value for each possible action in this state is calculated (6). If this is finished for each action, the value is calculated for the best action (6). The formula in line 6 is the same as:

$$V(s) = \max_a \sum Q(s,a)$$

Then this procedure is repeated from the beginning while taking the new estimated state-value for future states. If the new values and the old values converge, the loop is finished (9) and the optimal actions for all states are found while also the state-values are well estimated. Those state-action pairs are then extracted to build the policy function (11). This procedure is called value iteration because it only uses the value function and no policy function for the iteration process. [16]

1: Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in S$)
2: **repeat**
3: $\quad \Delta \leftarrow 0$
4: $\quad$ **for** each $s \in S$ **do**
5: $\quad\quad v \leftarrow V(s)$
6: $\quad\quad V(s) \leftarrow \max_a \sum_{s'} \Pr(s' \mid s,a)\,[R(s,a) + \gamma V(s')]$
7: $\quad\quad \Delta \leftarrow \max(\Delta, \mid v - V(s) \mid)$
8: $\quad$ **end for**
9: **until** $\Delta < \theta$ (a small positive number)
10: Output a deterministic policy, $\pi \approx \pi_*$, such that
11: $\quad \pi(s) \leftarrow \arg\max_a \sum_{s'} \Pr(s' \mid s,a)\,[R(s,a) + \gamma V(s')]$

Figure 9 Value Iteration Pseudo-Code [17]

2.1.2 Policy Iteration

The policy iteration algorithm repeats a policy evaluation and a policy improvement step repeatedly until convergence. First, a random initial policy is set (1). Then, in the policy evaluation step (3), the value of each step is calculated using the bellman equation and the current policy (8). This is again repeated until convergence (11). In the policy improvement step (13), the policy is updated by trying to maximize the Q-value for each step (17). Each iteration repeats those two steps until the policy converges (20). [16]

1: $V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$
2:
3: // 1. Policy Evaluation
4: **repeat**
5: $\quad \Delta \leftarrow 0$
6: $\quad$ **for** each $s \in S$ **do**
7: $\quad\quad v \leftarrow V(s)$
8: $\quad\quad V(s) \leftarrow \sum_{s'} \Pr(s' \mid s, \pi(s)) \left[R(s,a) + \gamma V(s')\right]$
9: $\quad\quad \Delta \leftarrow \max(\Delta, \mid v - V(s) \mid)$
10: $\quad$ **end for**
11: **until** $\Delta < \theta$ (a small positive number)
12:
13: // 2. Policy Improvement
14: *policy-stable* $\leftarrow$ *true*
15: **for** each $s \in S$ **do**
16: $\quad$ *old-action* $\leftarrow \pi(s)$
17: $\quad \pi(s) \leftarrow \arg\max_a \sum_{s'} \Pr(s' \mid s, a) \left[R(s,a) + \gamma V(s')\right]$
18: $\quad$ If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
19: **end for**
20: If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 4

Figure 10 Policy Iteration Pseudo-Code [17]

2.1.3 Comparison

The policy iteration algorithm is a bit more complex than the value iteration algorithm. However, it needs less iteration steps to converge and thus is faster. The fundamental problem in application is always a maximization of the value function by changing the policy. The policy iteration method uses two separate iteration processes for this problem while the value iteration method merges them into one iteration process. [16]

Policy evaluation and policy improvement are the two underlaying basic principles of almost all reinforcement learning algorithms.

2.2 Monte Carlo and Temporal Difference Learning

2.2.1 Monte Carlo Method

If the MDP dynamics are not completely known, the policy evaluation step shown in DP cannot be applied because the state-values are unknown since the state-transitions are unknown. Therefore, the Monte Carlo method simply averages the return (G) of different episodes which passed a particular state with the number of visited times (k). The average of those returns then become the estimated value of a certain state. According to this procedure, the Q-value can also be calculated by taking the mean return of state-action pairs which were passed in experienced episodes. [14, 18]

$$V(S_t) = mean[Returns(S_t)]$$

$$Q(S_{t,}\, A_t) = mean\big[Returns(S_{t,}\, A_t)\big] = \frac{G_1 + G_2 + \cdots + G_{n-1}}{k-1}$$

The policy improvement step in Monte Carlo is simply done by taking the actions which result in the maximal Q-value.

$$\pi^*(s) = argmax_a[Q(s,a)]$$

This procedure is repeated until the estimated returns (and consequently the policy) converges. The pseudo-code for Monte Carlo learning looks as follow:

Initialize $\pi(s)$ for all states
Initialize $Q(s, a)$ and $Returns(s, a)$ for all state-action pairs
repeat
  Randomly select S_0 and A_0 s.t. all state-action pairs' probabilities are nonzero.
  Generate an episode from S_0, A_0 under π: $S_0, A_0, R_0, S_1, \cdots, S_{T-1}, A_{T-1}, R_t$
  $G \leftarrow 0$
  $t \leftarrow T - 1$
  for $t >= 0$ **do**
    $G \leftarrow \gamma G + R_{t+1}$
    if $S_0, A_0, S_1, A_1 \cdots, S_{t-1}, A_{t-1}$ does not have S_t, A_t **then**
      $Returns(S_t, A_t)$.append(G)
      $Q(S_t, A_t) \leftarrow$ mean($Returns(S_t, A_t)$)
      $\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$
    end if
    $t \leftarrow t - 1$
  end for
until convergence

Figure 11: Monte Carlo Algorithm [14]

2.2.2 Incremental Monte Carlo

The problem of this algorithm is that every observed return must be saved in a list and averaged again which is very inefficient. The advanced computation of the Q-value simply adds up the difference between the obtained return and the old Q-value estimation to the old Q-Value estimation. This addend is multiplied with the inverse of the number of visited times (or a weight factor) which controls how fast the estimate is being updated. [14]

$$NewEstimate = OldEstimate + Weight \cdot (Return - OldEstimate)$$

$$Q_{k+1} = Q_k + \frac{1}{k}(G_k - Q_k) = Q_k + \alpha(G_k - Q_k)$$

This computation of the Q-value doesn't need to save all experienced returns and instead is able to calculate incremental a new estimation of the Q-value at the end of an episode.

The value calculation is equivalent:

$$V_{k+1} = V_k + \alpha(G_k - V_k)$$

2.2.3 Temporal Difference Method

The Monte Carlo method can only update the values or Q-values after each episode. To allow an update after each sample, Monte Carlo and DP can be combined for problems without complete knowledge about the process. This method is called temporal difference method (TD).

Therefore, the return is replaced with the reward at the next timesteps plus the discounted value at the next timestep. This can be computed step by step and not only at the end of an episode. [14, 18]

The Value and Q-value computation with TD accordingly is done as follows for a 1-step TD:

$$V(S_t)_{k+1} = V(S_t)_k + \alpha[R_{t+1} + \gamma V(S_{t+1})_k - V(S_t)_k]$$

$$Q(S_t, A_t)_{k+1} = Q(S_t, A_t)_k + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})_k - Q(S_t, A_t)_k]$$

This can be done with the following algorithm:

Input policy π
Initialize $V(s)$ and step size $\alpha \in (0, 1]$
for each episode **do**
  Initialize S_0
  for Each step S_t in the current episode **do**
    $A_t \leftarrow \pi(S_t)$
    $R_{t+1}, S_{t+1} \leftarrow Env(S_t, A_t)$
    $V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$
  end for
end for

Figure 12: Temporal Difference Algorithm [14]

2.2.4 Data storage

The estimated Q-values or values and the policy must be stored to be able to learn and to control the system after training. This can be done with the use of a table or with a neural network. Often, a Q-table is used which involves the estimated Q-values based on the observations and the possible actions. This can look the following way for two continuous observations and one possible action with two possible discrete values:

Observation 1	Observation 2	Action	Q-Value
10	5	0	20
10	5	1	40
2	8	0	80
2	8	1	0
…	…	…	…

Figure 13: Example Q-Table

The size of the table is given in advance by the possible actions and observations. For continuous observations and actions, a discretization according to the desired accuracy is needed. While training the agent, the Q-values are iteratively estimated more accurate. Based on this table, the policy can easily be extracted by taking the actions which result to the highest Q-values. [19]

When neural networks are used, the same principle is applied just by parametrizing the NN parameters which, for an actor/critic agent, results to a Q-value estimation using a function instead of a table. Based on those Q-value estimations, the policy can be extracted and stored in a neural network also by using a function instead of a table. This is done directly by estimating the error and changing the neural network to minimize that error. It is also possible to directly parametrize the policy network without the use of a value network. But still, the values or Q-values (or just the errors) must be implemented in some step to know which behaviour is better. A possible algorithm for that purpose is described in chapter 3.1.3. [14]

3 Reinforcement Learning Algorithms

3.1 Model-free RL

Model-free RL algorithms don't create a model of the MDP. Instead, they estimate the Q-Value and update the policy directly from observed experiences.

There are two different main types of RL-Algorithms. On-policy algorithms learn by only using actions which are according to the last version of the policy with slight changes of the current action taken. Those changes are crucial because if the algorithm only acts the way it already knows as the best actions, it cannot learn and always does the same thing. Future predicted actions for calculation of the Q-value are always based on the policy when using on-policy algorithms. Off-policy algorithms have a policy for exploring the environment to collect data and a different updated policy for acting in the exploitation case. Off-policy algorithms therefore can use any data obtained from exploration to update the exploitation policy. Those two policies are often called behavioural policy (exploration) and target or updated policy (exploitation). Future predicted actions for calculation of the Q-value are not based on the policy but on the explored best actions at the particular state when using off-policy algorithms. Monte Carlo and TD can both be used for on-policy as well as off-policy reinforcement learning. Plain policy optimization methods are by definition on-policy. However, advanced RL algorithms often use combinations of those methods and also apply policy optimization to off-policy learning. [9, 15]

3.1.1 Algorithms using Temporal Difference

3.1.1.1 SARSA (On-Policy)

SARSA is the most basic approach to apply TD controlling. It basically combines the policy evaluation of TD with a policy improvement step in between. The simplified procedure for a 1-step SARSA is as follows:

Initialize $Q(s, a)$ for all state-action pairs.
for each episode **do**
  Initialize S_0
  Select A_0 using policy that is based on Q
  for Each step S_t in the current episode **do**
    Select A_t from S_t using policy that is based on Q
    $R_{t+1}, S_{t+1} \leftarrow Env(S_t, A_t)$
    Select A_{t+1} from S_{t+1} using policy that is based on Q
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]$
  end for
end for

Figure 14: SARSA(1) Algorithm [14]

Again, here the actions must sometimes be different than the policy prescribes to not get stuck in a local maximum. This behaviour is often accomplished with an ϵ-greedy policy which has a probability of 1- ϵ to take the action with the highest estimated Q-value and a probability of ϵ to take a random action from the action-space.

The bootstrapping of the reward plus the discounted Q-Value can also be done for more than one step. If this is done for an infinite number of steps (until the end of the episode) this should equal to the return which makes this method equivalent to Monte Carlo. [9, 14, 18]

3.1.1.2 Q-Learning (Off-Policy)

Q-Learning is the most basic off-policy application of TD controlling. This plays a very important role in many reinforcement learning applications while most off-policy algorithms use this approach. The main difference from Q-Learning to SARSA is that the action chosen to calculate the Q-value of the next timestep (target Q-value) is no longer dependent on the policy being used. It is simply making a Q-table which describes the Q-value of all state-action pairs and then choosing the action with the highest value from this table. The pseudo-code for Q-learning is:

Initialize $Q(s, a)$ for all state-action pairs and step size $\alpha \in (0, 1]$
for each episode **do**
  Initialize S_0
  for Each step S_t in the current episode **do**
    Select A_t using policy that is based on Q
    $R_{t+1}, S_{t+1} \leftarrow Env(S_t, A_t)$
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$
  end for
end for

Figure 15: Q-Learning Algorithm [14]

Note that the selected action A_t at line 5 is chosen by a policy which explores (e.g. ϵ-greedy) and the selected action a at line 7 is chosen by a policy which doesn't explore but just chooses the best actions based on the experience (corresponds to ϵ-greedy policy with $\epsilon = 0$). That difference makes the algorithm off-policy. The fact that for future timesteps, not the policy but actions with the maximal Q-value are selected, makes this algorithm superior to find the global maximum although it often takes longer to converge. [9, 14, 18]

3.1.2 Algorithms using Policy Optimization

Methods which use policy optimization try to maximize the return (G) directly by changing the policy parameters (θ) which results to the following:

$$G(\pi_\theta) = E\left\{\sum_{k=0} \gamma^k r_k\right\}$$

$$\pi^*(\theta) = argmax_\theta[G(\pi_\theta)]$$

For applications in deep reinforcement learning with the use of a neural network, the policy parameters are the parameters (weights and bias) of the neural network.

This optimization is often done with the policy gradient method which uses the gradient of the return to update the policy. The policy parametrization update rule then is defined by:

$$\theta_{h+1} = \theta_h + \alpha_h \cdot \nabla_\theta G(\pi_\theta)|_{\theta=\theta_h}$$

The parameter α is the learning rate which defines how strong the policy is updated according to the gradient. The update number h is not the same as the timestep number k. Mostly, the update frequency is significantly less frequent.

The main difficulty with this method is to find a good estimation of the policy gradient $\nabla_\theta G(\theta)|_{\theta=\theta_h}$. The fact that the system can't be modelled in every detail just with the data generated, requires a stochastic estimation of the gradient without any model. There are different approaches for this problem, the most common use finite-differences or likelihood ratios. Those methods will not be explained here.

Policy gradient methods are by definition on-policy and only converge to a local maximum while value-function methods converge to the global maximum. This explains that on-policy algorithms using this principle are less likely to find the optimal behaviour then off-policy algorithms. Nevertheless, policy gradient methods are especially for automatization problems interesting because they are guaranteed to converge at least to a local maximum and often use fewer parameters in the learning process than value-function methods. Policy gradient methods can be used model-free as well as model-based.

Many advanced model-free RL-algorithms use a combination of value-function and policy gradient methods to obtain a better result. This is often done with off-policy algorithms which use the principle of Q-learning. Accordingly, those algorithms labelled as off-policy are in fact a combination of both categories. [20, 21]

3.1.3 Deep RL Algorithms

Using neural networks slightly changes the reinforcement learning process. As stated in chapter 2.2.4, the Q-values don't need to be completely calculated to optimize the NN. This can be done just by using the error estimation and minimizing that error. The following pseudocode provides an understanding of how a simplified RL algorithm with the use of NN works. This example is purely hypothetical because applied RL algorithms with neural networks use additional highly mathematical approaches which make it hard to understand the fundamentals.

```
Initialize Actor (S, A)
Initialize Critic (S, A, Q)
For each episode do:
        Initialize S(0)
        For each step S(t) in current episode do:
                A(t) <- Actor(S(t))
                R(t+1), S(t+1) <- Env[S(t), A(t)]
                Store data
                Sometimes do:
                        CriticLoss <- f{R(t+1) + y*max.a[Q(S+1, a+1)] - Q(S, A)}        # Temporal Diffe
                        Critic <- f{CriticLoss, learningrate}                          # e.g. Backpropa
        End for
        Sometimes do:
                ActorLoss <- f{Q(S, A)}                                                # Q(S, A) <- Cri
                Actor <- f{ActorLoss, learningrate}
End for
```

Figure 16: Principle of Deep RL Algorithm using TD

This example is based on temporal difference to calculate the critic loss which is used to optimize the NN. The critic is then used to calculate the actor loss and optimize the actor network. The difference to tabular RL is eventually just the additional step of optimizing the NN to depict the Q-values and the policy. [22–24]

3.2 Model-based RL

Model-based RL Algorithms try to build a model of the MDP dynamics and then solve the optimization problem to find the best policy. Those MDP dynamics include the reward-function and state-transitions. The policy can be created iteratively using dynamic programming (chapter 2.1) or with the use of decision trees (often used for discrete action-spaces).

One big challenge of model-based algorithms is the data generation to build the model. For this purpose, many different approaches are available which cannot easily be categorized in clusters of methods. In general, many state-action pairs must be sampled to get the information about the next state and the reward. In probabilistic processes, where a state-action pair also has a transition probability, those pairs must be sampled multiple times to calculate the transition probability. With this information and some estimations or generalizations, the model then can be built.

Model-based algorithms have much better sample efficiency then model-free algorithms. This means that with a lower sample size, the algorithms perform much better than model-free

algorithms. Also, a base model can be given previously to the algorithm so that it can start with a much better performance. This is useful for critical applications which need to be trained in the real world. Another strength of model-based RL is the ability to plan explicitly. This allows the agent to deliberately make decisions which are good in a long term even when the system has slightly changed. Model-free RL algorithms also make decisions which are good in a long term, but they do it implicitly and are not able to adapt quickly to a changed system.

However, when predictions are strung together and little errors are made, they compound over multiple steps and can get bigger. There is also a challenge because the policy optimization can tend to exploit regions where insufficient data is available to train the model and predictions which estimate too high rewards are made. This problem also occurs if the training data differs from the real environment and the model is estimated too accurate. This phenomenon is called overfitting and is due to the accumulation of predictions in model-based RL specifically problematic. All those problems have potential for huge failures and can be traced back to the number of assumptions and approximations.

Model-based RL just got bigger attention in the last years and is less researched than model-free RL in the moment. There are also attempts to combine model-free with model-based RL algorithms. [25–27]

4 Toolchain

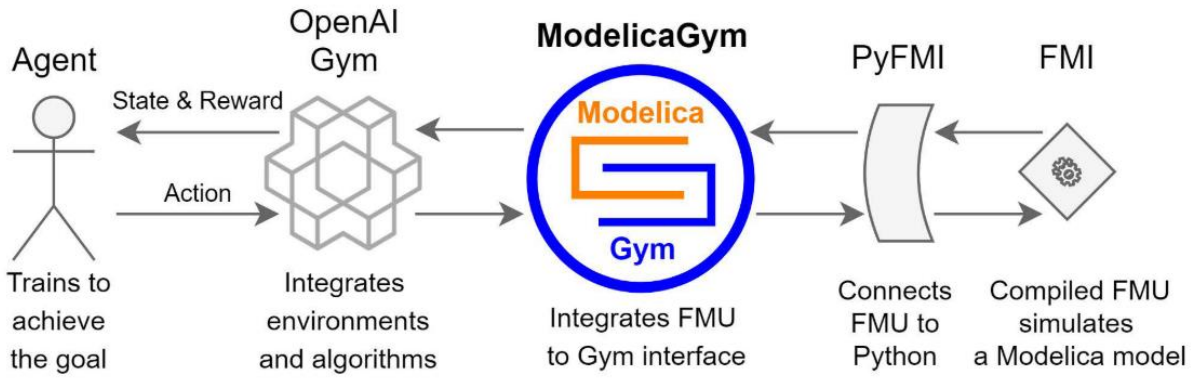

Figure 17: Toolchain of RL with ModelicaGym [28]

Several tools are used to apply reinforcement learning for controlling tasks with Modelica models. For the basic RL utilization, it is required to have an agent and an environment (see Chapter 1.3). There are various tools available in python for this application. When coupling RL with Modelica, the task of the Modelica model is to calculate the system state at each timestep according to the performed action of the agent. This requires the ability to simulate the Modelica model in python which will be done by use of the FMI standard. The ModelicaGym toolbox enables this implementation of FMU in an OpenAI Gym environment.

4.1 OpenAI Gym Environment

A RL environment by use of OpenAI Gym is more than just the world or the system model. It includes several definitions and functions used for interaction with the agent. OpenAI Gym is the most common python-library for environment creation in RL. This library provides a standardized interface which is very well developed and documented while also a heap of agents is available in python to interact with OpenAI Gym environments. The main tasks of an OpenAI Gym environment are:

- Calculation of the current system state at the current time-step
- Definition of the initialization state
- Calculation of the reward based on the defined reward-function
- Definition of the observation- and action-space
- Definition of the episode termination

This is done by defining a bunch of functions which then are executed in interaction with the agent. The program flow using the OpenAI Gym framework is as follows:

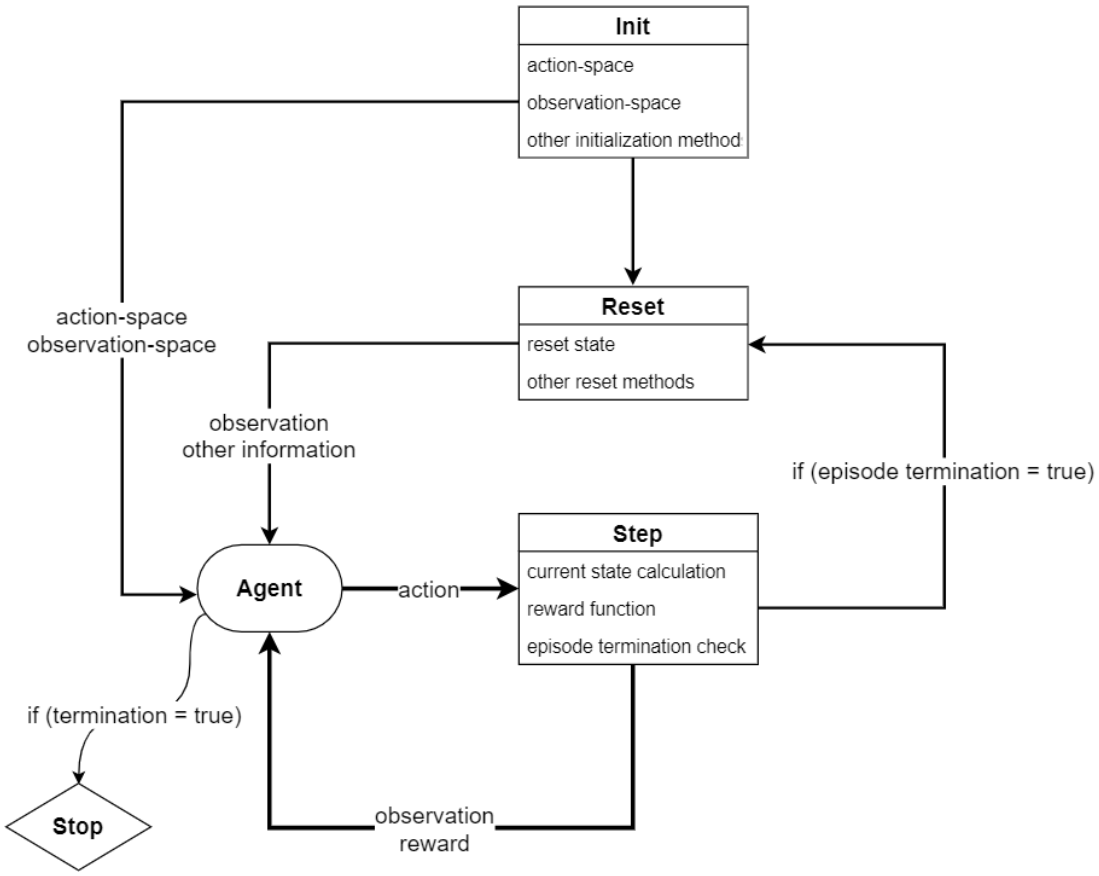

Figure 18: Flow using OpenAI Gym Framework

4.2 Coupling Modelica-Models with OpenAI Gym Environment

4.2.1 Modelica

Modelica is an object-oriented programming language which is used for modelling physical systems. Modelica is an equation-based language which means that variables don't get assigned causally, they are determined by solving equations. The mathematical background is that in a system of equations, it is possible to determine all variables if you have as many different equations as variables. Modelica has many reusable, uniform model-components available in the standard library which allow an efficient modelling of systems even with multiple domains. [29]

4.2.2 Functional Mock-up Interface

Functional mock-up interface defines a standard which serves as an interface for program exchange. The Modelica model is saved in a single Zip-file containing XML, binary- and C-Code. This is then called a functional mock-up unit (FMU). This FMU can be implemented for example in a python program to execute simulations while reading and writing defined variables. The implementation of FMU in this work is done with the PyFMI library which is used in the ModelicaGym library. [30]

4.2.3 ModelicaGym

ModelicaGym is a toolbox which allows the interaction of an OpenAI Gym environment with FMU-files. It is basically a framework for an OpenAI Gym environment with implemented PyFMI functions. This allows the environment to get the current system state through simulation of the Modelica model by executing the FMU with defined input (action & system parameter) and output (observation) variables. The ModelicaGym framework is based on multiple nested classes because it allows the use of different FMU-types which sometimes require different PyFMI functions. The UML chart on the next side provides a structural understanding of ModelicaGym used with Co-Simulation FMU version 2.0.

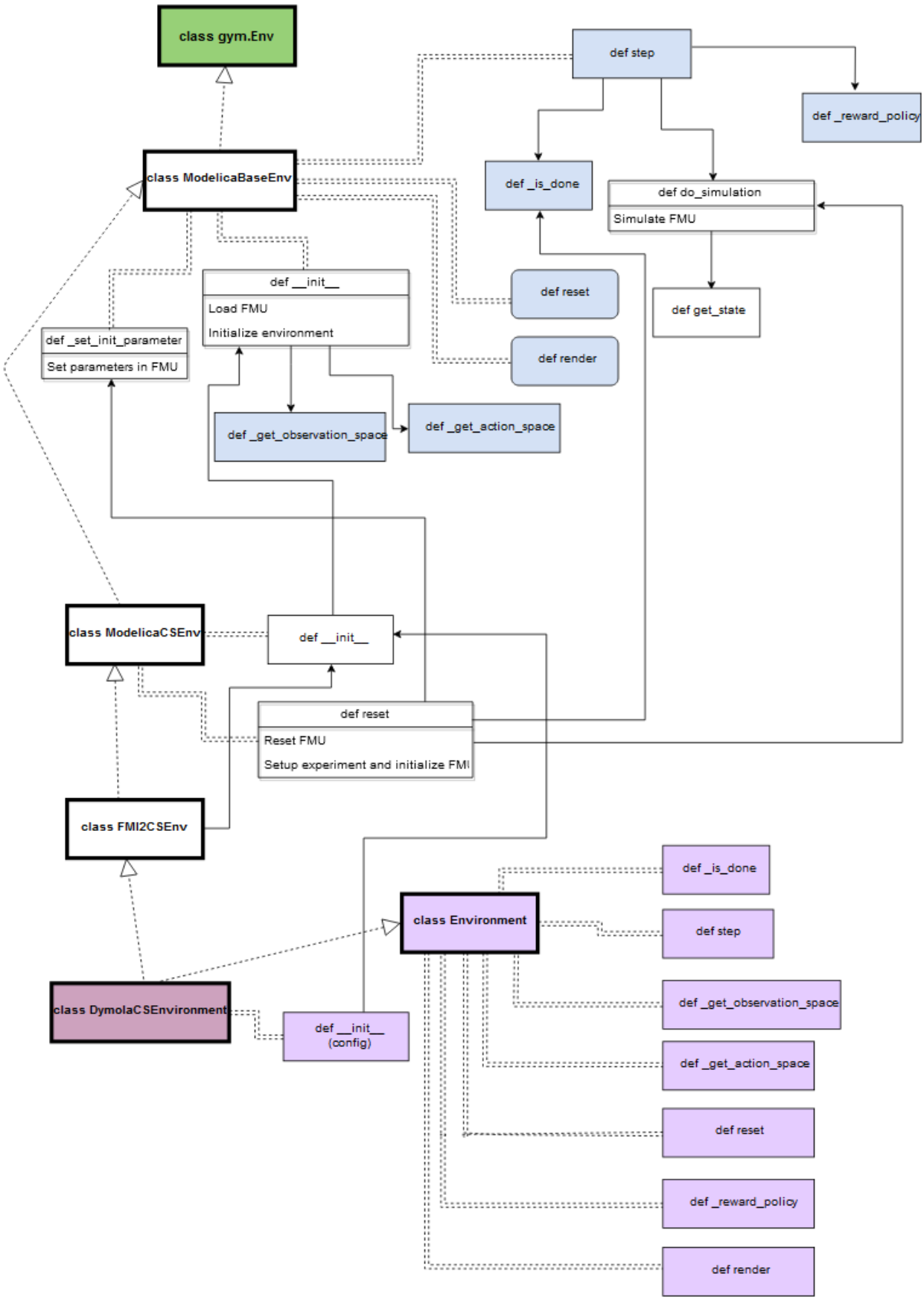

Figure 19: UML ModelicaGym

To use ModelicaGym, you have to define an environment class which extends the ModelicaGym classes which again extend the gym.Env class of OpenAI Gym. This environment class should define all the required functions described in Chapter 4.1 except the calculation of the system state. Those are at least the following functions:

- _is_done: defines when the episode is terminated
- _get_observation_space: defines the available observation-space
- _get_action_space: defines the available action-space
- _def_reward_policy: defines the reward-function

Additionally, you must define the "config"-variables which provide certain needed information like the length of a time-step, the location of the FMU, the changeable system parameters, action, and observation variables of the FMU. According to the ModelicaGym examples, those are defined in a separate class (DymolaCSEnvironment) inside the "__init__" function. This can also be done directly in the environment class. [28]

4.3 Agent

The agent includes the RL Algorithm and the policy (usually a neural network). To create an agent, there are mainly two possible frameworks, PyTorch and TensorFlow available, which are both very common. Further, many implementations based on those frameworks are available which allow an easy and clean implementation of RL agents. In this work, the library stable-baselines3 (SB3) was used which utilizes the PyTorch framework.

The SB3 library provides a unified structure for different model-free RL algorithms which can be implemented very easily in the program to optimize a neural network which is also created by SB3. The library is made to interact with OpenAI Gym environments. It allows for several settings like defining the neural network structure, tuning RL algorithm parameters, defining the total timesteps to learn etc. It also supports Tensorboard to visualize different variables which are useful for insight and evaluation of the learning process. [31]

5 Determinants of RL Applications

Many factors are influencing the learning behaviour and outcome of RL. The optimal settings vary for different applications while it is impossible to predict the actual influence of each setting combination. Often, it is best to just experiment and go for intuition. This chapter will provide an understanding of the most important determinants.

5.1 Environment design

5.1.1 Reward Function

Since the reward function defines the value of each system state, it is the main influence on the behaviour of the agent. It is important to understand the system and decide exactly which outcome is desired, also considering a non-optimal agent.

The most main contributions for reward function design are:

- Immediate reward at each state / Reward just at certain states or at the end of the episode
- Negative (punishment) / Positive rewards
- Reward balancing
- Reward scaling

The following descriptions will include examples of reward function design. Those examples will be based on a fictive problem for a car which has to drive from position A to position B in a one-dimensional system.

5.1.1.1 Immediate Reward vs. Selective Reward

Usually, immediate rewards are easier to learn because the value calculation can be more precise for future states and fewer differences of the states. Immediate rewards specify a path for the learning process and allow the agent to get a near-optimal behaviour even when the exact desired behaviour is not achievable at this moment or not achievable at all. Immediate rewards also make the agent more predictable since the learning path and the value of each state is stronger defined. Still, the agent just tries to maximize the total reward which can include sacrificing immediate rewards in order to maximize the return at the end of the episode. Another advantage of immediate rewards is that the agent has a gradient to work with which again makes it easier to determine if it gets closer or further from the target behaviour. The ultimate immediate reward definition is by a continuous reward function which is different for each system state.

Example immediate continuous reward: "Reward = – (Distance from position B)"

Selective rewards can be given just at certain states or at the end of the episode. The problem with selective rewards is that it can be much more difficult for the agent to estimate the episodic value based on the actions he takes. Imagine you must complete a task without knowing the

task and only be told “good” or “bad” sometimes. It is much more difficult to estimate which actions you took led to the desired outcome and which actions were hindering your success. It also can require a huge sample size to escape a local maximum which might not be at the best possible return. However, selective rewards can lead to a more unexpected behavioural outcome since the agent is completely unguided in its way to learn receiving the maximal return. [32, 33]

Example selective reward: “If (position = position B) → Reward = 100; else → Reward = 0”

5.1.1.2 Negative vs Positive Rewards

Reward functions can be designed using negative (punishment) or positive rewards. In theory, it makes no difference if the reward is positive or negative. If one reward is relatively higher than the other, the state is more desirable. However, in practice it can make a difference. For example, if the agent gets only negative rewards, it is motivated to end the episode as fast as possible since each step will lead to a lower return. If the agent gets only positive rewards, it is motivated to make an episode as long as possible since it will only increase the return with each timestep in the episode. Also, a mix of negative and positive rewards will sometimes make the agent actively avoid states with negative rewards and drive it to states with positive rewards. This can also hinder the agent from achieving the maximal possible return if negative rewards must be accepted to obtain a higher reward later. [33, 34]

Example negative reward: “If (car crashes) → Reward = –1000”

Example positive reward: “Reward = Distance from position A”

A combination of those reward functions will make the agent strongly avoid crashes and can make it hard exploring that a crash leads to a higher total reward than just stopping if a crash is inevitable to keep driving away from position A.

5.1.1.3 Reward balancing and scaling

The example above leads to the importance of reward balancing and scaling. Again, in theory the agent tries to maximize the total reward at the end of the episode. Reward balancing can change the optimal behaviour if rewards or punishments are in conflict to each other. If a crash must be avoided at any cost, the negative reward on a crash should be relatively big enough so that the positive reward can’t compensate it. It is also important to keep the non-optimal behaviour in mind which can be highly affected by different reward scaling. This points out the importance of a good system understanding and a clear specification of the desired behaviour. Reward scaling can also make a difference on the learning behaviour learning since the gradient of the values is changed. This changes the update magnitude especially for gradient-based algorithms. [32–34]

Example reward scaling: “Reward = – (Distance from position B)^3”

5.1.2 Episode Termination

The definition of the episode termination indirectly affects the reward function since the total reward gets reset after each episode. This contributes to the behaviour of the agent. As stated before, when giving negative rewards, the agent might try to end the episode as quickly as possible to avoid accumulating lots of negative rewards. Correspondingly, when giving only positive rewards, the agent might try to never end the episode since it can accumulate more positive rewards that way.

The episode termination is usually based on the following categories:

- Reaching a pre-set number of timesteps
- System is out of specification
- Task is completed

The definition must be chosen based on the desired behaviour of the agent while trying to understand the complete agent's motivation. Usually, at episode termination, a reward or punishment for that specific state is implemented. The definition of the episode termination must be matched with the definition of the reward function. For example, if the episode is terminated after completing a certain task including a reward, but the reward at each timestep until completing the task is always zero, the agent is not motivated to complete the task as fast as possible since it doesn't affect the return. The same episode termination definition with a negative reward each timestep however motivates the agent to complete the task as fast as possible.

For controlling tasks, it is often beneficial to add a small negative reward on episode termination to prevent the agent from intentionally reaching that state in training without exploring the system. [32]

5.1.3 Observation- & Action-Space

In principle, the observation- and action-space are fixed by the process. They are essentially used to predefine the inputs and outputs of the neural network. More input neurons in a neuronal network can lead to overfitting and prevent generalization. Based on that, the number of observations should be considered when creating an environment. More observations are not always preferable, especially if the agent is trained on a slightly different system than it is applied afterwards. The individual variables in the observation-space should be defined as good as possible on the possible values. If each observation is predefined to have a possible value between -infinity and infinity even though this isn't the actual case, the discretization of the neural network might not be efficient. The action-space of course must define the possible or desired range based on the system while deviations can also lead to inefficiency.

Using some algorithms, it can be beneficial to normalize the action- and observation-space variables and then rescale them later. In the stable-baselines3 library, some continuous RL-algorithms (e.g. PPO) rely on a gaussian distribution initially centred at 0 with a standard deviation of 1. In this case, if the action space is different, it will mostly lead to very low or saturated

actions. Algorithms like TD3 or DDPG rescale the output to fit the action-space. In this case, a normalization isn't needed. [35]

5.1.4 Timestep

The timestep defines how much time passes between each step of the agent, hence how often the agent gets a new observation and can change its action. A physical process is always continuous which makes this an important factor. A smaller timestep lets the agent do more precise actions and adapt faster to disturbances or imperfection. Usually, a smaller timestep is preferable but in real systems, it is limited by the hardware speed. This must be considered when learning with a system model and applying the agent to a real system. A smaller timestep also increases the training time if the system is inert since each simulation interval can only change the system state a smaller amount, then by using a bigger timestep.

5.1.5 Randomness

Sometimes it is beneficial to add randomness to the environment. This can include random disturbances, random initial states, or random target states. Random initial and target states can force the agent to explore states which otherwise would not be explored. The use of this is dependent on the requirements of the agent. Generally, the learning conditions should be as similar as possible to the applying conditions. Otherwise, the agent can learn certain patterns and leave other states or dynamics completely unexplored.

5.2 Agent design

An agent in deep RL contains two components, the RL algorithm, and the neural network. The purpose of deep reinforcement learning is to parametrize one or multiple neural networks by use of the RL algorithm and the information provided by the environment. Changing those two components has an essential impact on the outcome. This chapter will provide a general overview of important differences designing those components.

5.2.1 Neural Network design

As stated before, an agent can contain multiple neural networks. In stable-baselines3 (SB3), often two neural networks are used which have different purposes.

One neural network always contains the target policy and is called 'actor'. The other neural network is called 'critic' which determines the value or Q-value of each state.

Multiple settings can be made for a neural network. The overall functionality must be decided first. For controlling tasks, almost always multilayer perceptron (MLP) networks are used which are feedforward neural networks. This is the basic method of neural networks which is explained in chapter 1.5.1. Using another method, e.g. CNN, RNN, changes the outcome

drastically. Those other possibilities are not further explained here due to the rare use in controlling tasks.

5.2.1.1 Network Architecture

When in use of the SB3 library, on-policy algorithms have a neural network with some, all or no layers shared between the actor and critic. So, it essentially creates not two but one neural network which can calculate an estimation of the value (critic) and the action (actor) based on the current observation. Sharing a neural network can sometimes improve or simplify the optimization of itself. Off-policy algorithms always need two different neural networks for the actor and critic to prevent issues with target and behaviour policy.

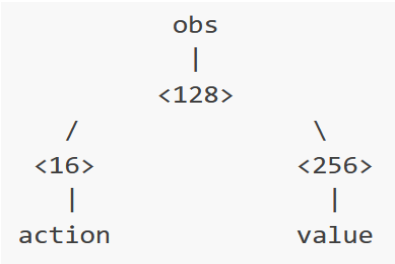

Figure 20 Neural Network with Shared Layer for On-Policy Algorithms

The number of hidden layers and the number of neurons in each hidden layer describe the architecture of a neural network. To estimate which settings work best for the current application, it is best to try different approaches and choose the best. To begin, it is best to look at other applications which successfully trained a neural network on a similar problem and start from there.

In theory, each function can be represented with a neural network using one hidden layer. In practice, a NN with more hidden layers usually is better in generalizing the problem than a network with less hidden layers but more neurons in each layer. Also, just using some layers with less neurons forces the NN to generalize the problem at a certain place of the function. The number of overall neurons should always be enough to represent the problem in the desired accuracy. Based on that, a neural network must be bigger if the problem is more complex and should have more neurons in each layer if less generalization is required.

Using different architectures for actor and critic can be beneficial if different behaviours are desired concerning generalization and complexity. For example, if the controlling task is very easy but the importance lays in finding the optimal path in the MDP, the NN of the critic should be bigger than the actor. [36]

5.2.1.2 Activation Function

The activation function describes the function type used in each neuron. This can also be changed but with the use of SB3 it is best to keep the default because it is matched to the algorithm. Using other activation functions can cause errors like ‘out of bounds’ etc. and should be handled carefully. [37]

5.2.2 RL-Algorithm

The RL algorithm is used to parametrize the neural network. Using different algorithms changes the learning behaviour and the outcome if an optimal solution of the problem is not found. The

decision which RL algorithm to use depends on if you want to use model-based or model-free algorithms. If using model-free algorithms, the main decision lays by using on-policy or off-policy algorithms but also which method should be used. This contains for example Q-learning, policy gradient methods etc. Many advanced algorithms use multiple methods and combine multiple advantages. Mostly, advanced RL algorithms should work better for all problems but in some cases, it could be beneficial to use a basic RL-algorithm for a basic problem. The differences of on-policy and off-policy algorithms are described in chapter 3. In theory, on-policy algorithms should converge in a less optimal but more consistent solution while off-policy algorithms are better in finding the actual best solution.

5.2.2.1 RL Algorithm Parameters

RL algorithms use different parameters which must be declared. Those parameters change the learning behaviour of the agent. Important parameters are the total timesteps (learning duration), learning rate, update rate, and the noise.

The total timesteps define how long the algorithms optimizes the policy. In general, a longer learning duration leads to a better agent. However, it must be paid attention to overfitting which can worsen the agent especially if the training and test system slightly differs. The required learning duration highly varies on different system complexities as well as the environment and agent settings. To estimate a good learning duration, it should be looked at convergence of certain variables (see Chapter 6.2). The agent can also be tested at different learning durations to compare its performance.

The learning rate defines how hard the networks are updated if a better solution is found. A higher learning rate updates the networks more immediately while a lower learning rate needs the same discovery to happen more often before the neural network is completely adjusted to it. The learning rate can also be a function of the current progress. This parameter can be set in all reinforcement learning algorithms of SB3.

The update rate contains a set of multiple parameters which all describe how often the neural networks are updated. This can include the general training frequency, the policy update delay and the learning start point. Those parameters can be used to collect more information about the environment before updating the neural networks which is useful if one action may be bad immediately but beneficial in the future. The training frequency decides the update rate of the critic and actor while the policy update delay describes the delay of the actor being updated with regards of the training frequency. The designation of those parameters changes for different algorithms and can be read in the documentation of SB3. [38]

6 Evaluation of Neural Network Learning

There are two main possibilities to evaluate the quality of the neural network learned. The first is to apply the target policy in a testcase in order to estimate the overall return and to examine the agent behaviour. The other possibility is to analyze the evolution of some variables which are used in the learning process. Tensorboard is available for SB3 algorithms which provides those figures of important variables.

6.1 Policy Evaluation with Testcase

Most RL algorithms use exploration noise during training. That is why you need to create a separate testcase to evaluate the agent's performance.

The function 'evaluate_policy' in SB3 runs the target policy for several episodes and returns the average return with the standard deviation. This can be used to compare different agents and gives a good assessment how good the agent performs. Those values can also be compared to the maximal possible return based on the reward-function and the timesteps used. Although it must be kept in mind that the maximal possible return is often not possible to achieve with inert systems.

With this function, it is also possible to activate the render-function if implemented in the environment. This can be used to visually assess the agent which can sometimes be the easiest and fastest way for a rough evaluation of a visualizable system. But still, a more accurate evaluation is only possible with numeric results.

To receive the true quality of the agent, for deterministic systems (which Modelica models usually are), the agent should be set deterministic too. On default, some RL agents are stochastic which means that they don't always take the same actions in the same states. This means that they don't always take the best action even if they knew better. This can sometimes make a performance difference.

An alternative for the evaluation after the agent has finished training is to frequently evaluate it during training. This can be done with the function 'eval_callback'. This can be used to identify the agent performance at certain number of timesteps and determine the benefit of further training. The callback function also allows to save multiple model versions during the training progress which can be useful if the agent worsens at a certain point. [39, 40]

6.2 Training Evaluation with Tensorboard

During training with SB3 agents, certain variables are logged. Those variables differ with the algorithm because different variables are used. To understand the behaviour of those variables, some mathematical background must be known. This chapter will give an overview of some important variables to evaluate the performance of training.

To open Tensorboard, the command ‘tensorboard --logdir=location_of_the_logfolder’ must be run in cmd which creates a port in ‘http://localhost:6006’ which can be opened in the browser.

The x-axis of the figures always shows the number of timesteps run in training while the y-axis shows the value of a certain variable. The exact definition of those variables can be read in the documentation of SB3 in the chapter ‘Logger’. [39]

6.2.1 Mean Episode Reward

The mean episode reward is logged for all algorithms in SB3. It shows the mean return obtained during the training session averaged over 100 episodes. [39] This value should increase over the number of timesteps in training. However, it is no problem if this value decreases on a short term. A short term decrease of this value can happen by virtue of the policy noise in training which is essential for learning. It can also momentarily decrease if the agent leaves a local maximum which sometimes requires a few more training timesteps to optimize its new approach. Another reason for short-term decrease can be caused by regularization methods (see Chapter 1.4.2.2) of the neural network optimization.

This value doesn’t show the episode return of the target policy because for exploration in training, there is always a policy noise, or a complete different behavioural policy (for off-policy algorithms) applied. Therefore, the absolute values cannot exactly be compared with other algorithms to evaluate the performance. However, it can be used to compare certain different settings of the same algorithm or to roughly assess the learning behaviour. A long term decrease of this value is usually a clear indication of overfitting. For algorithms which use policy optimization, a convergence of this value indicates that the algorithm has stopped learning or cannot optimize any further. For off-policy algorithms which are purely based on Q-Learning, a convergence of this value doesn’t imply that the target policy has reached its maximum. This is due to the fact that the behavioural policy can be completely different from the target policy and isn’t just a reflection of the target policy added with some noise or randomness (on-policy algorithms). So, algorithms which are purely based on Q-Learning can still be improving even though the mean episode reward in training has converged.

6.2.2 Loss

A neural network is trained by using an optimization process. This requires a cost-function which can be minimized. This cost-function represents the error between the prediction and a perfect model (in this context, model describes the neural network) and is called loss-function while the actual value of this function is called loss (see chapter 1.4.2.1).

It is important to keep in mind that this loss function can be defined in many ways. That is why the desired behaviour can only be known if the mathematical background is understood. For basic actor-critic methods using temporal difference, the losses are usually defined the following way:

$$\delta = R_t + \gamma V(S_{t+1}) - V(S_t)$$

$$Critic\ loss = \delta^2$$

$$Actor\ loss = \delta \cdot \ln \pi(A_t|S_t)$$

Again, the actor describes the policy network while the critic describes the value network.

In this work, the definitions of the loss functions will not be discussed any further due to its complexity and diversity. However, those definitions can already give an understanding why the evolution of the actor and critic loss can be completely different while also desired behaviours of different loss-definitions can change completely.

One basic way to use the loss values for evaluation of the training process is to look for convergence. No matter what definition the loss values have, a convergence of the loss always means that the neural network isn't changing much anymore. Further, if a loss value for example constantly keeps decreasing and then starts increasing again, it can indicate a decline of the neural network for example due to overfitting. The figures of the loss values along with the mean episode reward during training can give a good estimation of the training process. If all those values converge, the training should be terminated because it shouldn't improve anymore after this. It can also indicate if the training process was run too long if some of those values start changing directions over many timesteps. [41, 42]

6.2.2.1 Loss Observations on TD3 Algorithm

The empiric perception of the loss values during the experiments has yield that for the TD3 algorithm, a desirable behaviour of the loss values is a constant decrease of the actor loss and a constant slight increase of the critic loss.

A large immediate increase of the critic loss indicated an escape of a local maximum or overfitting. Often, the actor loss increased over some timesteps after the large increase of the critic loss which indicates that the actor network must be adjusted to the new approach explored with the critic network. This increase of the actor loss will mostly lead to a short term decrease of the episodic reward when the actor network is not adjusted properly yet. If the actor loss does

not again start to decrease after some timesteps, the critic network might be overfitted and training should be stopped.

In the following graphs, the blue lines will show an example of an agent stuck in a local maximum. The critic loss is very low while the actor loss also converged immediately. The mean episodic reward shows that the agent was stuck in swinging the pendulum around consistently.

Or orange lines show an algorithm which was learning good and around 1.2M timesteps, the actor loss started rising which resulted to a decrease of the mean episodic reward. The critic loss started highly increasing at that point which indicates in combination with the other graphs that the network has overfitted.

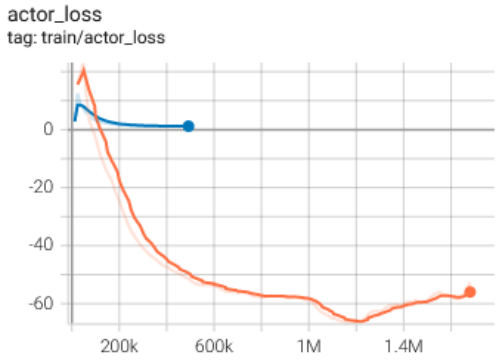

Figure 21: Actor Loss

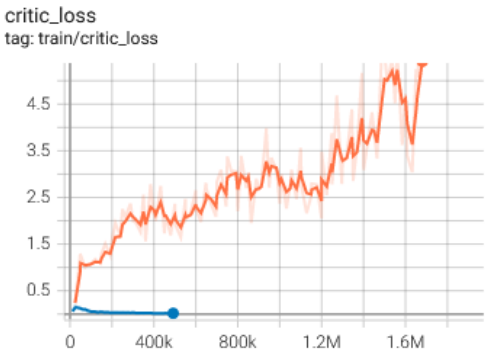

Figure 22: Critic Loss

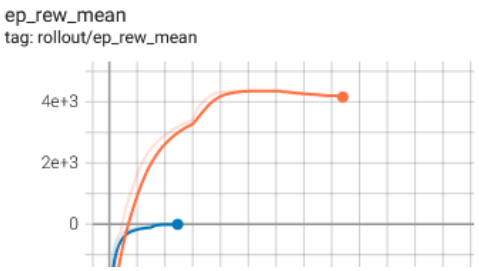

Figure 23: Mean Episodic Reward during Training

7 Showcase Preparation Process

For collection of experiences and demonstration purposes, a showcase was created during the process of this work. The creation of the eventual showcase was broken down in small individual steps which provided the required knowledge about the whole process.

As a first step, some premade environments from the OpenAI Gym library were used to provide some experience about the basic reinforcement learning process and the agent libraries.

In the second step, a custom environment purely based on OpenAI Gym was created and applied (Appendix B). This environment behaviour was defined by a linear function which had to be learned by the agent. The linear function was: $state = 14.097 \cdot action - 0.004$. In addition, a random varying target system state had to be achieved based on the system behaviour and the actions taken. The goal of this work step was to provide an understanding of beneficial environment definition. The main cognizance was that the target state must be provided in the observation space if it is randomly varying. If the target state is always the same, it is enough to implement it in the reward function. This is because a random target state can logically not be predicted by the agent while a consistent target state (or target state progression over the episode) can be implicitly learned via the reward function because the highest reward is provided exactly at the target state at the current timestep.

In the third step, a Modelica system model was implemented with the use of the ModelicaGym library. To break down the step, first an FMU was simulated in python by use of the PyFMI library and the according functions which are used in ModelicaGym (Appendix C). This assured the functioning of the FMU simulation and foreclosed failures of this cause. The utilized Modelica system model was based on the same transition function as the custom environment used in the second step. The system depicted the flow of water through a valve to receive a certain mass flow. No inertia was included in the system which made the resulting mass flow just a linear dependency of the valve opening.

The RL parameters of the system were:

- Observation-space: current mass flow (controlled variable), target mass flow (setpoint)
- Action-space: valve opening (manipulated variable)
- Reward-function: if (current mass flow = target mass flow $\pm$ 0.5) → positive Reward
  else → negative Reward
- Episode termination: at simulation time = 5s

To apply this system for RL, an environment based on ModeliaGym and the above stated functions was created. The environment code is provided in appendix D.

After successfully controlling this system with the RL agent, a disturbance was included with a second valve which was randomly closed between a valve opening of 0.8 and 1. The Modelica model looks as follows:

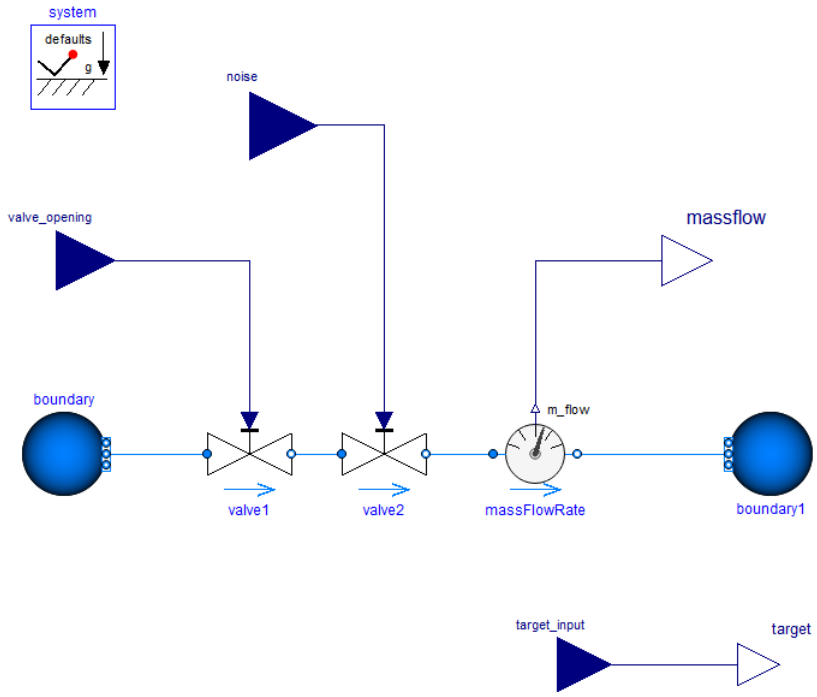

Figure 24: Modelica Model with Noise for the ModelicaGym Testcase

The evaluation of the system was done by looking at the evaluation return which should have been the maximal possible return since the noise was small enough to be balanced with the target tolerance. This was achieved perfectly after a training time of about 10 minutes to one hour depending on the algorithm used. The following graph shows the evaluation returns during training by use of the TD3 algorithm.

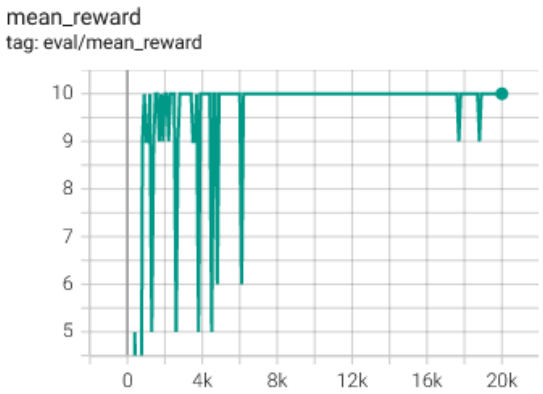

Figure 25: Mean Episodic Reward at Evaluation during Training

The two little spikes with a lower than maximal return at the end are because of the noise, which doesn't always allow a perfect system state.

8 Showcase Double Pendulum

The main showcase is a system of a double pendulum. The system was chosen because it had some avails for this purpose as a showcase. A system model of a double pendulum is already available on the Modelica standard library which could easily be used as a basis. Using the system from the Modelica standard library ensured that the system behaviour physically makes sense without any additional validation effort. This system is highly sensible and non-linear which makes it a fairly hard controlling task. Additionally, due to the defined goal, it is not possible to perform a linearization on this system for the controller design. With basic controller design techniques, this system would not be controllable. However, an advanced state controller (e.g., LQR) should still be able to do the same task as shown here. This system was also considered challenging for RL which allowed a good comparison between different settings regarding the functionality and results.

8.1 System

The system is modelled as follows with Modelica:

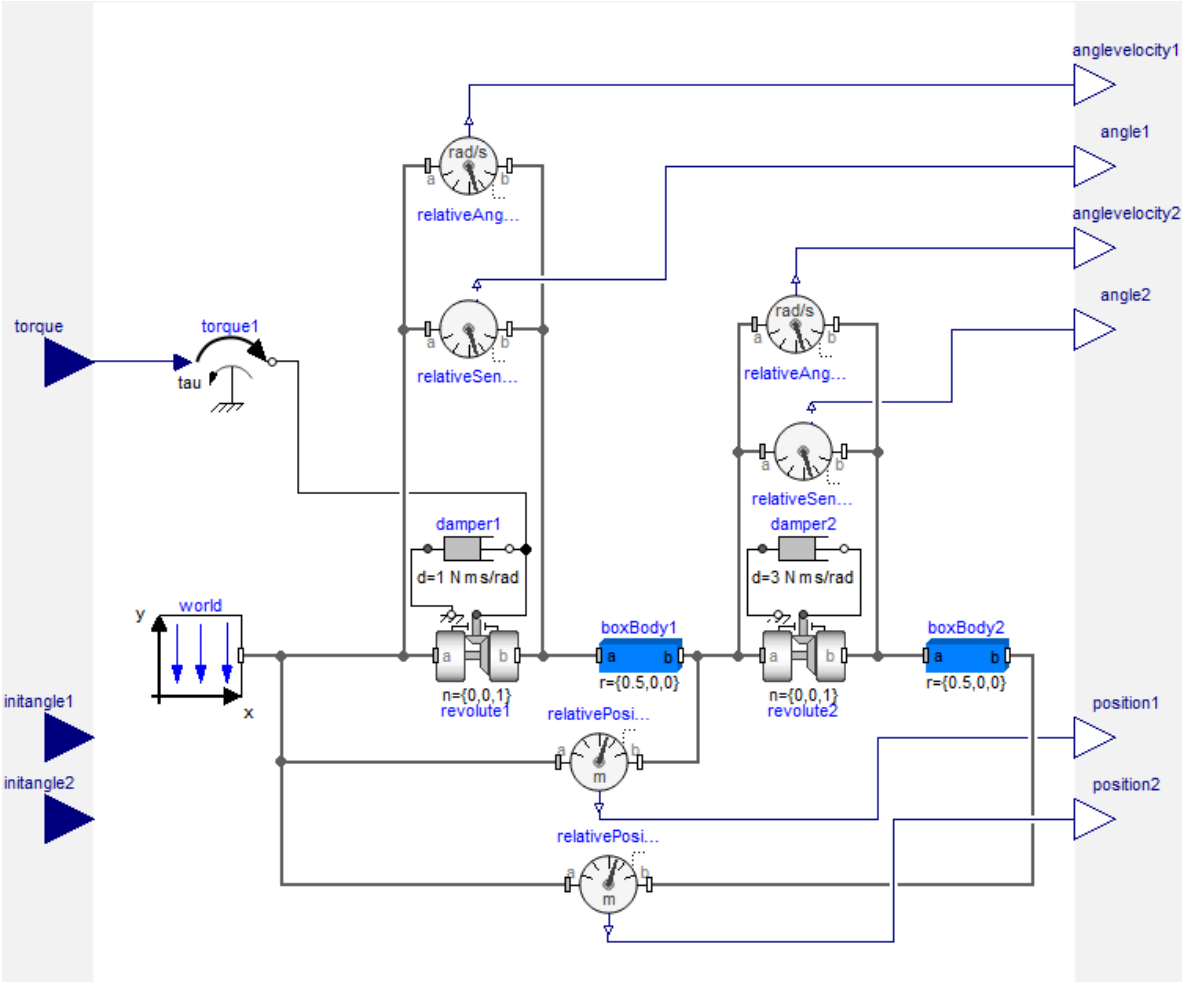

Figure 26: Modelica Model used for the Showcase

Compared to the Modelica standard library system, the torque applied to the first joint and all the sensors had to be added. All the angles, angle velocities and the height of the rods are measured. The angle accelerations are not measured and must be estimated by the agent. The initialization angles can be changed in python but for simplification, the pendulum start point was always at a resting downward position. The damping of the joints is set at a realistic small value.

8.1.1 Goal

The controlling goal is to swing up the pendulum as fast as possible to an upward position and balance it at this position until the end of the episode. Different reward functions are possible to define this goal, such as definitions with the help of the measured rod angles or with the rod heights.

8.2 Experiments for Settings

Different approaches were tested to train the RL agent. For a systematic investigation, first the main influencing factors were considered to systematically apply the best settings and create a functioning agent. They can be grouped in environment, RL-algorithm, and neural network settings.

Environment	RL-Algorithm	Neuronal Network
Reward function	Algorithm-type	Number of hidden layers
Initialization state	Total learning timesteps	Number of neurons
Episode termination	Action-noise	Architecture
Observation-space	Update rate (training frequency)	Actor / Critic difference
Action-space	Learning rate	Activation function
Timestep	Steps before learning starts	

Following settings were fixed in all attempts:

- Initialization state: resting downward position
- Episode termination: at t = 30 sec
- Observation-space: given by the system model, angle velocities at $\pm$ infinity
- Action-space: $\pm$ 200 Nm torque
- Update rate (training frequency): 1024 steps
- Steps before learning starts: 20000 steps
- Actor / Critic neuronal network: same network architecture
- Activation function of neurons: tanh-function

Those settings were chosen based on research of other applications with adaptation and deliberation on the present system.

To roughly compare the performance of different settings, the mean episodic reward during training was watched. This doesn't give an exact assessment of the learned model, but it is suitable to compare the different learning behaviours and to estimate the eventual evaluation performance. In addition, the behaviour was visually assessed using the learned agent. The total learning timesteps were always at 500'000 steps for the first attempts. This took about 2.5 hours to train each model.

Two different learning algorithms were tested on the system. Only model-free RL was tried out in the showcase for simplicity reasons. The tested algorithms were TD3 and PPO. TD3 is an advanced off-policy algorithm combining Q-learning and policy optimization. It is the direct successor of DDPG which again is a modification of DQN to be applied on a continuous action-space. PPO is an advanced on-policy algorithm which is based on the principle of A2C and TRPO. Those algorithms were chosen because they represent the most developed applications of on-policy and off-policy algorithms.

8.2.1 TD3 (Off-Policy) Algorithm

8.2.1.1 Timestep

First, two different timesteps of 0.01 sec and 0.005 sec were tested with the same neural network (3 hidden layers with each 64 neurons). The result was clear that 0.005 sec timestep was better. It must be kept in mind that the possible episodic reward with a timestep of 0.01 sec is 3000 while the possible episodic reward with a timestep of 0.05 sec is 6000 because one episode contains twice as much timesteps, and the reward is given at each timestep. In later test cases, it revealed that a timestep of 0.01 sec was again good with the use of a different neural network architecture (2 hidden layers with each 192 neurons).

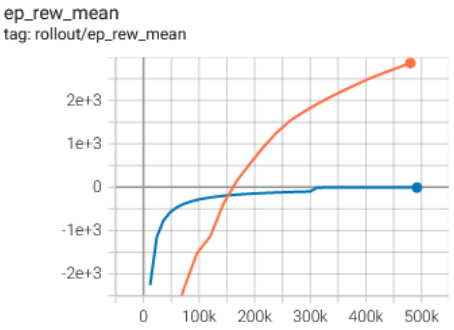

Figure 27:
Orange: 0.005s timestep,
Blue: 0.01s timestep

8.2.1.2 Network architecture

The network architecture was varied at the number of hidden layers and the number of neurons in each hidden layer. The following notation will be used from now on to describe the neural network architecture: 64x3 NN → neural network with 3 hidden layers and 64 neurons in each layer. A neural network with more hidden layers and less neurons in each layer seems to learn faster but also converge faster while a network with less layers and more neurons per layer seems to be still learning a lot at

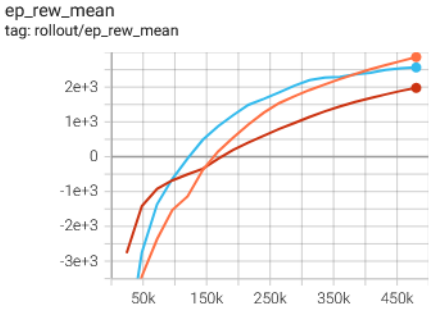

Figure 28: Orange: 64x3NN,
Blue: 64x5NN, Red: 192x2NN

500’000 timesteps. The experimental differences in this testcase however were not very significant. Due to the gradient of the episodic reward at the end, it was considered that less hidden layers could perform better. Also, the theoretical knowledge that more hidden layers are better at generalizing, which is not particularly wanted in this system because it is highly sensitive, reinforced that feeling.

8.2.1.3 Action-noise

The action-noise adds a probability to the actions taken based on a gaussian distribution. The standard deviation (σ) changes the probabilities to take further deviating actions then calculated by the policy. For this test, a 192x2 NN and a timestep of 0.005 sec was used. As seen in the graph, the higher standard deviation of the action noise allowed for faster learning at the beginning due to harder exploration. Although, it could be hard to explore the equilibrium state of the pendulum in the upright position because the action noise might always put it in a state in which the pendulum will be forced to lose balance.

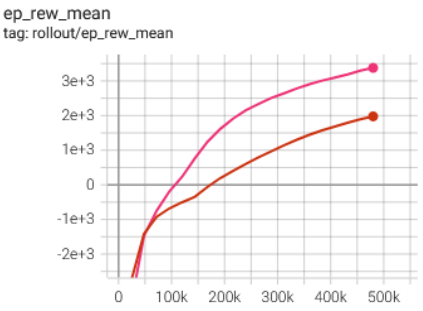

Figure 29: Red: σ = 0.005, Pink: σ = 0.01

This phenomenon has shown in a test with a timestep of 0.01 sec, a 192x2 NN and an action-noise σ=0.01. The higher action-noise combined with the changed network architecture allowed the agent to learn much better at the timestep of 0.01 sec. Although, the learning seemed to converge at a relatively high episodic reward. In visual evaluation with help of the animation, it is apparent that the agent is not able to balance the pendulum at the upright position. This might be because of the problem stated above, that the system is very sensible, and the higher action-noise disturb the balance point too much. A higher action-noise seems to be better at the start but worse at the end of the learning progress.

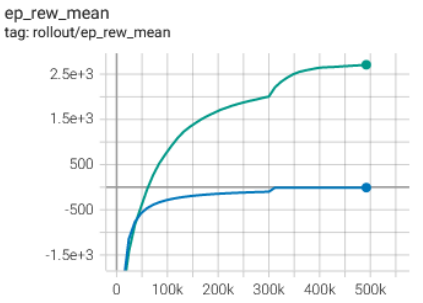

Figure 30:
Blue: σ=0.005 (64x3NN),
Green: σ =0.01 (192x2NN)

8.2.2 PPO (On-Policy) Algorithm

When in use of the PPO-Algorithm, it is very important that the action-space is normalized between the values ± 1. Otherwise, the algorithm didn't learn at all when interacting with this system. The explanation of that is given at chapter 5.1.3. When using the TD3 algorithm, this normalization is done automatically by the agent. Similar approaches as described above were applied on PPO. However, all settings resulted in a much worse outcome compared to the TD3 algorithm. The graph shows a few PPO results of the mean episodic reward compared to the TD3 result (green line). It is also well visible that the on-policy algorithm (PPO) often converges faster than the off-policy algorithm (TD3) but the optimal behaviour is less likely achieved. This agrees with the theory stated in chapter 3.1.2. Not all experiment results are described in this chapter because all the results were much worse than those achieved with TD3 algorithm.

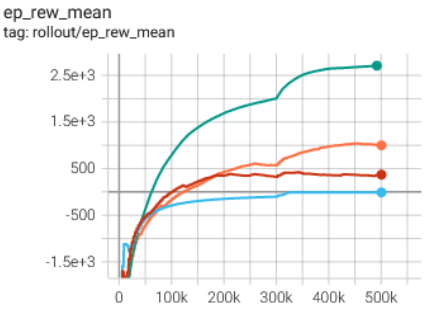

Figure 31:
Green: TD3-Algorithm,
Others: PPO-Algorithm

8.3 Result

Based on the experiment cognition, a promising agent was trained for more timesteps. The first attempt was with the TD3 algorithm by using 0.005 sec timestep and a 192x2 NN with an action-noise σ=0.005. The evaluation was done during learning with an evaluation callback to estimate the real episodic reward each 50'000 timesteps. The agent got significantly worse at 1.6 Mio timesteps, why training was stopped at this point. The extreme worsening at this point was probably because the algorithm has overfitted the critic network (value function) and estimated a too good reward at certain states which were not beneficial. This is also noticeable at the critic loss which made big jumps in the end. This indicates that the real value calculation with temporal difference has changed a lot compared to the value estimation of the critic network. However, the agent was still not able to balance the pendulum at the upright position.

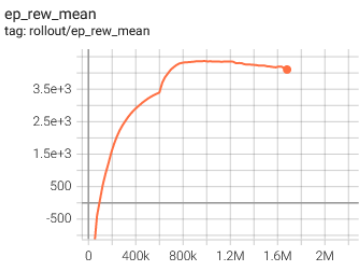

Figure 32: Mean Episodic Reward at Training

Figure 33: Mean Episodic Reward at Evaluation

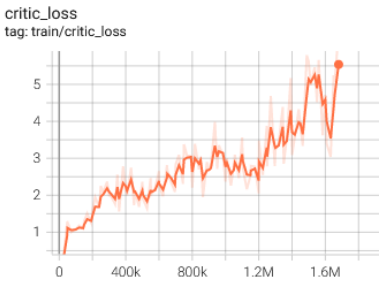

Figure 34: Critic Loss

In previous experiments it was also noticeable that one problem was that the agent sometimes got stuck in a local maximum by consistently turning the pendulum around. This will always result in a total reward of zero because the reward function was defined by: reward = height of second rod, which after all will be equalized by the positive and negative values. To avoid this problem and force the agent to learn at the desired state, the reward function was slightly changed to:

```
reward = height of second rod

if(height of second rod < 0):
        reward = reward -1
```

This reward function gave a higher penalty if the pendulum is at the downward position and should also eliminate the local maximum of a consistent swinging pendulum. It was expected that the agent will be able to learn longer at the relatively good state which it already has reached without worsening too early.

The following application was done with the new reward function, a timestep of 0.01 sec and a lower action-noise $\sigma = 0.003$. The timestep was adjusted because it was expected that the higher timestep allowed for twice as fast learning due to the damping and because earlier experiments showed that it should also be possible to train the agent with this setting. The action-noise was lowered because it should allow the agent to optimize better at the upward position of the pendulum without too much disturbance. The slower learning progress expected with the smaller action-noise should be equalized with the bigger timestep.

The agent result was finally able to balance the pendulum at the upward position. As hoped, overfitting at later timesteps was no problem anymore with the new reward function. The rendered pendulum is shown in the following pictures.

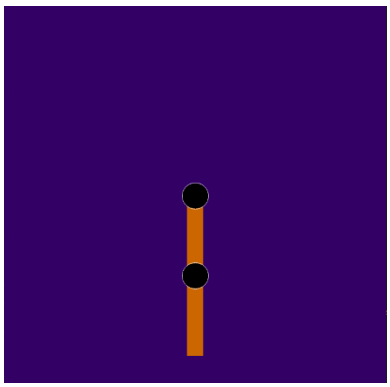

Figure 35: Pendulum at Start Position

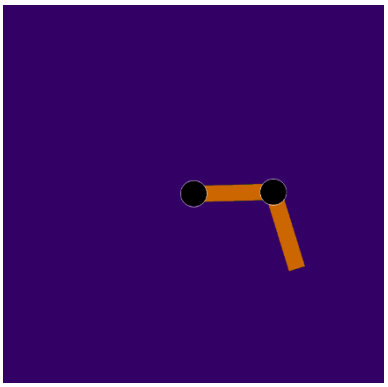

Figure 36: Pendulum Swinging Up

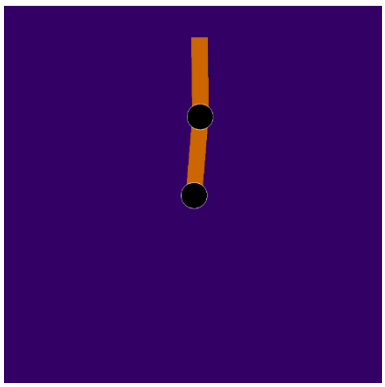

Figure 37: Pendulum at Target Position

The evolution of the training and evaluation returns are visible in the following graphs:

Figure 38: Mean Episodic Reward at Evaluation

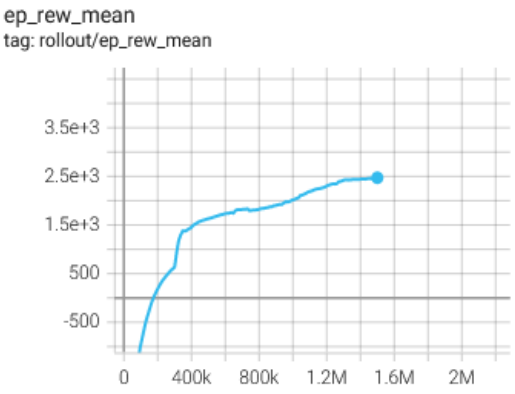

Figure 39: Mean Episodic Reward at Training

The following graphs show the actions taken at each timestep, the height of the second rod and the resulting rewards at each timestep over the whole episode:

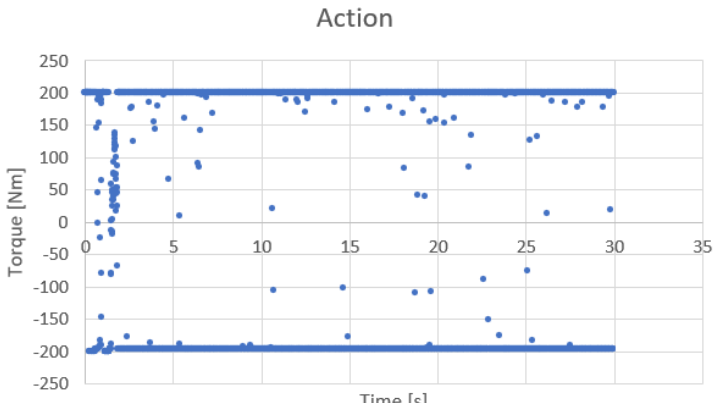

Figure 40: Actions each Timestep

Figure 41: Rewards each Timestep

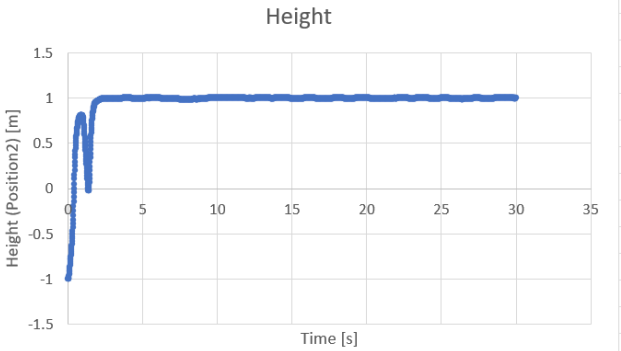

Figure 42: Heights of the Second Rod each Timestep

It is appearant that the agent changed very often between the maximal and minimal possible torque instead of choosing the exact right torque at each state. This indicates that the timestep could possibly be set even bigger which will evenautlly force the agent to learn a more continous instead of a nearly discrete behaviour between the maximal and minimal torque at the upright balancing point.

It is also well visible how the adjusted reward function shifted the negative rewards by -1. This eliminated the problem of having a local maximum at a constant swinging pendulum. The controlling speed was not investigated any futher but it seems pretty fast with less than two seconds to reach a balancing upright position.

It must be mentioned that the determined agent settings might not be the best anymore since the environment definition has changed by way of the reward function. However, due to lack of time and a satisfying result, no further tries were made after this.

8.3.1.1 Evaluation with varied System Properties

To be a good controller, the agent should also be able to perform well with disturbances or a slightly changing system. The trained agent was tested on the same system with doubled and halved damping in the joints. This could represent the wear of the bearings which gets bigger at the time. Since the agent doesn't train anymore and only uses the premade policy, it could also represent certain outer disturbances applied to the system. However, the agent was not able to control the system in that slight varied case. This is because the agent only observes the angles and angle velocities but to describe the complete state of the system, the angle acceleration is crucial too. Therefore, to be able to control a system with disturbances or changes, either the acceleration or the time must be provided in the observation-space. This would allow for a complete description of the current state and should make the agents performance independent on outer influences. A varied timestep had the same impact on the agent's performance since one action of the agent has a much different impact on the system this way which is not represented in the policy.

Conclusion and Outlook

Reinforcement learning for controlling tasks can definitely be beneficial for complex systems. Eventually, the main advantage is that a neural network can be trained efficiently which allows the usage of a very complex transfer function. Also due to the use of reinforcement learning, it is not required to have the same system understanding compared to using traditional controller design techniques. However, a fundamental system understanding is still required to setup the environment and the agent. The big disadvantage of reinforcement learning is that the system behaviour can only be influenced indirectly, and that the behaviour of the controller cannot be completely comprehended due to its complexity. For critical applications, this can be very unfavourable. An advantage of reinforcement learning is that it not only allows for complex transfer functions, but it also finds the best behaviour itself to reach the desired goal. If the desired goal can be defined exactly but the desired controller behaviour is hard to specify, reinforcement learning is well suited. For simple applications, reinforcement learning is not very beneficial since it still uses a lot of effort to create the setup. The main time-consuming tasks are finding beneficial settings of the environment and the agent. Due to its complexity and effects on each other, even with a good understanding of the topic, it is required to test different settings. Because of the long training computation times, the creation of a good agent needs a lot of time.

Combining reinforcement learning with Modelica models works very well with help of the ModelicaGym library. The use of Modelica models allows the use of a complex system model in training with relatively low effort. This enables training in a pure virtual environment which can shorten the training time compared to training on a real time process.

The purpose of this work was to gain first experiences with reinforcement learning for controlling of physical systems. Further condition was to implement the simulation of Modelica system models to train and evaluate the agent. For experimentation and demonstration, a showcase had to be created. Those objectives were reached within this work. Much important basic knowledge was investigated and documented during the working process. The creation of the showcase facilitated the fundamental understanding of model-free reinforcement learning. It gave an insight of some libraries which simplify the application, and it also provided a first use case to get a feeling of beneficial environment and agent definitions. In all applications, it appeared that off-policy algorithms seemed to be promising for physical controlling tasks.

Nevertheless, a lot of future investigations are needed to get a holistic understanding of the whole application with its possibilities and limits. Sensible future tasks are described in the following sections.

For a viable utilization of reinforcement learning, a standardized procedure should be defined. To allow the creation of such a procedure, systematic investigations for beneficial settings in different use cases must be made. This includes environment and agent research. The first investigations should be about the reward-function combined with the definition of the episode

termination. This should provide a better feeling of the according behaviour of the agent just by adjusting the environment. It would also be interesting to know which observation-spaces work best. Are more observations always better or is it sometimes better to have fewer observations and prevent overfitting and too complex neural networks?

The next investigations could be about the agent choice and settings. This includes questions about the best algorithm choice and settings for physical control systems. Also, a better understanding about neural network architectures for actor and critic should be available to start the design process from a better standpoint. This should also be based on a physical control system which encloses the whole problem. To complete this topic, investigations on model-based algorithms should be made and compared to model-free algorithms. Especially since model-based reinforcement learning seems promising for robotics and automation.

Further, a procedure for agent behaviour testing and understanding after the training should be defined. This procedure should be able to ensure the desired behaviour of the agent which is especially important for real-world problems which might be critical applications. Also, it must be investigated how the agent or neural network can be implemented advantageous in a real-world controller for a real application. When creating such a controller, the timestep must be adjusted to the hardware speed.

After a better understanding of the whole subject, with different libraries and approaches tested, a better comparison to conventional controlling can be made and profitable cases for applications with reinforcement learning can be identified.

For me personally, this work gave a very interesting insight of reinforcement learning which is increasingly getting more attention nowadays. I think this will be an important topic in near future and it seems a very promising approach for applying artificial intelligence to control technology.

References

[1] *Overview of Artificial Intelligence Technology | FINRA.org.* [Online]. Available: https://www.finra.org/rules-guidance/key-topics/fintech/report/artificial-intelligence-in-the-securities-industry/overview-of-ai-tech (accessed: Jun. 7 2022).

[2] "Supervised vs. Unsupervised Learning: What's the Difference?," 03 Dec., 2021. https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning (accessed: Apr. 7 2022).

[3] L. Wuttke, "Reinforcement Learning: Wenn KI auf Belohnungen reagiert," *datasolut GmbH*, 27 Apr., 2022. https://datasolut.com/reinforcement-learning/ (accessed: Jun. 7 2022).

[4] YouTube, *RL Course by David Silver - Lecture 1: Introduction to Reinforcement Learning.* [Online]. Available: https://www.youtube.com/watch?v=2pWv7GOvuf0 (accessed: Apr. 20 2022).

[5] YouTube, *RL Course by David Silver - Lecture 2: Markov Decision Process.* [Online]. Available: https://www.youtube.com/watch?v=lfHX2hHRMVQ&list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzFObQ&index=3 (accessed: Apr. 20 2022).

[6] YouTube, *RL Course by David Silver - Lecture 3: Planning by Dynamic Programming.* [Online]. Available: https://www.youtube.com/watch?v=Nd1-UUMVfz4&list=PLqYmG7hTraZDM-OYHWgPebj2MfCFzFObQ&index=4 (accessed: Apr. 20 2022).

[7] YouTube, *RL Course by David Silver - Lecture 4: Model-Free Prediction.* [Online]. Available: https://www.youtube.com/watch?v=PnHCvfgC_ZA (accessed: Apr. 20 2022).

[8] *Reinforcement Learning for Control Systems Applications - MATLAB & Simulink - MathWorks Schweiz.* [Online]. Available: https://ch.mathworks.com/help/reinforcement-learning/ug/reinforcement-learning-for-control-systems-applications.html (accessed: Jun. 7 2022).

[9] S. Causevic, "Overview of Reinforcement Learning Algorithms | Towards Data Science," *Towards Data Science*, 18 Jun., 2020. https://towardsdatascience.com/an-overview-of-classic-reinforcement-learning-algorithms-part-1-f79c8b87e5af (accessed: Apr. 7 2022).

[10] blackburn, "Reinforcement Learning : Markov-Decision Process (Part 1)," *Towards Data Science*, 18 Jul., 2019. https://towardsdatascience.com/introduction-to-reinforcement-learning-markov-decision-process-44c533ebf8da (accessed: Jun. 7 2022).

[11] R. Jagtap, "Understanding Markov Decision Process (MDP) - Towards Data Science," *Towards Data Science*, 27 Sep., 2020. https://towardsdatascience.com/understanding-the-markov-decision-process-mdp-8f838510f150 (accessed: Apr. 17 2022).

[12] A. Opppermann, “What Is Deep Learning and How Does It Work?,” *Built In*, 05 Feb., 2022. https://builtin.com/machine-learning/what-is-deep-learning (accessed: Jun. 7 2022).

[13] *How Deep Neural Networks Work - YouTube.* [Online]. Available: https://www.youtube.com/watch?v=ILsA4nyG7I0 (accessed: Jun. 7 2022).

[14] Hao Dong, Zihan Ding, Shanghang Zhang, *Deep Reinforcement Learning: Fundamentals, Research and Applications*: Springer, 2020.

[15] *Part 2: Kinds of RL Algorithms — Spinning Up documentation.* [Online]. Available: https://spinningup.openai.com/en/latest/spinningup/rl_intro2.html (accessed: May 12 2022).

[16] A. A. Tokuç, “Value Iteration vs. Policy Iteration in Reinforcement Learning,” *Baeldung on Computer Science*, 07 Aug., 2021. https://www.baeldung.com/cs/ml-value-iteration-vs-policy-iteration (accessed: Apr. 21 2022).

[17] Miyoung Han, *Reinforcement Learning Approaches in Dynamic Environments*, 2018.

[18] Ching-An Wu, *Investigation of Different Observation and Action Spaces for Reinforcement Learning on Reaching Tasks*, 2019.

[19] GitHub, *Temporal-Difference/TD.ipynb at master · shangeth/Temporal-Difference.* [Online]. Available: https://github.com/shangeth/Temporal-Difference/blob/master/TD.ipynb (accessed: Jun. 7 2022).

[20] *Part 3: Intro to Policy Optimization — Spinning Up documentation.* [Online]. Available: https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html (accessed: May 13 2022).

[21] J. Peters, “Policy gradient methods,” *Scholarpedia*, vol. 5, no. 11, p. 3698, 2010, doi: 10.4249/scholarpedia.3698.

[22] *Twin Delayed DDPG — Spinning Up documentation.* [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/td3.html (accessed: Jun. 7 2022).

[23] *Deep Deterministic Policy Gradient — Spinning Up documentation.* [Online]. Available: https://spinningup.openai.com/en/latest/algorithms/ddpg.html (accessed: Jun. 7 2022).

[24] TensorFlow, *Playing CartPole with the Actor-Critic Method | TensorFlow Core.* [Online]. Available: https://www.tensorflow.org/tutorials/reinforcement_learning/actor_critic (accessed: Jun. 7 2022).

[25] D. Seita, *Model-Based Reinforcement Learning:
Theory and Practice.* [Online]. Available: https://bair.berkeley.edu/blog/2019/12/12/mbpo/ (accessed: May 10 2022).

[26] J. Hui, “RL — Model-based Reinforcement Learning - Jonathan Hui - Medium,” *Medium*, 25 Sep., 2018. https://jonathan-hui.medium.com/rl-model-based-reinforcement-learning-3c2b6f0aa323 (accessed: May 12 2022).

[27] YouTube, *L6 Model-based RL (Foundations of Deep RL Series).* [Online]. Available: https://www.youtube.com/watch?v=2o1yrkbpcUk (accessed: May 12 2022).

[28] T. B. Oleh Lukianykhin, *ModelicaGym: Applying Reinforcement Learning to Modelica Models*, 2019.

[29] *Dymola – Dassault Systèmes®.* [Online]. Available: https://www.3ds.com/de/produkte-und-services/catia/produkte/dymola/ (accessed: Jun. 7 2022).

[30] *Functional Mock-up Interface.* [Online]. Available: https://fmi-standard.org/ (accessed: Jun. 7 2022).

[31] *Stable-Baselines3 Docs - Reliable Reinforcement Learning Implementations — Stable Baselines3 1.5.1a8 documentation.* [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/index.html (accessed: Jun. 7 2022).

[32] J. Stemmler, "Writing successful reward functions," *Neal Analytics*, 28 Apr., 2021. https://nealanalytics.com/blog/writing-successful-reward-functions/ (accessed: Jun. 7 2022).

[33] A. Zhang, "How to Design a Reinforcement Learning Reward Function for a Lunar Lander," *Towards Data Science*, 18 Aug., 2021. https://towardsdatascience.com/how-to-design-reinforcement-learning-reward-function-for-a-lunar-lander-562a24c393f6 (accessed: Jun. 7 2022).

[34] YouTube, *Writing Great Reward Functions - Bonsai.* [Online]. Available: https://www.youtube.com/watch?v=0R3PnJEisqk (accessed: Jun. 7 2022).

[35] *Reinforcement Learning Tips and Tricks — Stable Baselines 2.10.2 documentation.* [Online]. Available: https://stable-baselines.readthedocs.io/en/master/guide/rl_tips.html (accessed: Jun. 7 2022).

[36] J. Brownlee, "How to Configure the Number of Layers and Nodes in a Neural Network," *Machine Learning Mastery*, 26 Jul., 2018. https://machinelearningmastery.com/how-to-configure-the-number-of-layers-and-nodes-in-a-neural-network/ (accessed: Jun. 7 2022).

[37] *Custom Policy Network — Stable Baselines3 1.5.1a8 documentation.* [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/guide/custom_policy.html (accessed: Jun. 7 2022).

[38] *Base RL Class — Stable Baselines3 1.5.1a8 documentation.* [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/modules/base.html (accessed: Jun. 7 2022).

[39] *Logger — Stable Baselines3 1.5.1a8 documentation.* [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/common/logger.html?highlight=logger (accessed: Jun. 7 2022).

[40] *Callbacks — Stable Baselines3 1.5.1a8 documentation.* [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/guide/callbacks.html?highlight=eval_callback (accessed: Jun. 7 2022).

[41] J. Brownlee, "Loss and Loss Functions for Training Deep Learning Neural Networks," *Machine Learning Mastery*, 27 Jan., 2019. https://machinelearningmastery.com/loss-and-loss-functions-for-training-deep-learning-neural-networks/ (accessed: Jun. 7 2022).

[42] *Everything You Need To Master Actor Critic Methods | Tensorflow 2 Tutorial - YouTube.* [Online]. Available: https://www.youtube.com/watch?v=LawaN3BdI00 (accessed: Jun. 7 2022).

Table of Figures

Appendix A: Showcase Python Code

A.1 Environment

```
# specify working directory for library import
import sys
from os import path
sys.path.append(".")

# ModelicaGym
from modelicagym.environment import FMI2CSEnv
from gym import spaces
from cmath import inf
import logging
import numpy as np

# rendering
import pygame
from pygame import gfxdraw
from numpy import cos, pi, sin

# logger definition
logger = logging.getLogger(__name__)

# Environment definition based on OpenAI Gym framework

class Pendulum_env:

# defining the episode termination

    def _is_done(self):

        if self.stop >=30:
            done = True
        else:
            done = False
        return done

# defining the action-space (used in modelica_base_env / __init__)

    def _get_action_space(self):

        return spaces.Box(low = -200, high = 200, shape = (1,), dtype = "float32")

# defining the observation-space (used in modelica_base_env / __init__)

    def _get_observation_space(self):

        return spaces.Box(low=np.array([-np.pi, -np.pi, -inf, -inf, -0.5, -1]),
high=np.array([np.pi, np.pi, inf, inf, 0.5, 1]), dtype= "float32")

# refering to the step function of ModelicaGym

    def step(self, action):

        return super().step(action)

# refering to the reset function of ModelicaGym

    def reset(self):
```

```
        return super().reset()

# defining the reward-function

    def _reward_policy(self):

        angle1, angle2, anglevelocity1, anglevelocity2, position1, position2 = self.state

        logger.debug("angle1: {0}, angle2: {1}, anglevelocity1: {2}, anglevelocity2: {3},
position1 {4}, position2 {5}".format(angle1, angle2, anglevelocity1, anglevelocity2,
position1, position2))

        reward = position2

        if position2 < 0:
            reward = reward - 1

        return reward

# rendering the environment with pygame

    def render(self, mode="human"):
        FPS = 300
        SCREEN_DIM = 700
        LINK_LENGTH_1 = 0.5  # [m]
        LINK_LENGTH_2 = 0.5  # [m]

        if self.screen is None:
            pygame.init()
            pygame.display.init()
            self.screen = pygame.display.set_mode((SCREEN_DIM, SCREEN_DIM))

        if self.clock is None:
            self.clock = pygame.time.Clock()

        self.surf = pygame.Surface((SCREEN_DIM, SCREEN_DIM))
        self.surf.fill((51, 0, 102))
        s = self.state

        bound = LINK_LENGTH_1 + LINK_LENGTH_2 + 0.2  # 2.2 for default
        scale = SCREEN_DIM / (bound * 2)
        offset = SCREEN_DIM / 2

        if s is None:
            return None

        p1 = [
            -LINK_LENGTH_1 * cos(s[0]+pi/2) * scale,
            LINK_LENGTH_1 * sin(s[0]+pi/2) * scale,
        ]

        p2 = [
            p1[0] - LINK_LENGTH_2 * cos(s[1]+pi/2) * scale,
            p1[1] + LINK_LENGTH_2 * sin(s[0] + s[1]+pi/2) * scale,
        ]

        xys = np.array([[0, 0], p1, p2])[:, ::-1]
        thetas = [s[0], s[0]+s[1]]
        link_lengths = [LINK_LENGTH_1 * scale, LINK_LENGTH_2 * scale]

        for ((x, y), th, llen) in zip(xys, thetas, link_lengths):
```

```
            x = x + offset
            y = y + offset
            l, r, t, b = 0, llen, 0.05 * scale, -0.05 * scale
            coords = [(l, b), (l, t), (r, t), (r, b)]
            transformed_coords = []
            for coord in coords:
                coord = pygame.math.Vector2(coord).rotate_rad(th)
                coord = (coord[0] + x, coord[1] + y)
                transformed_coords.append(coord)
            gfxdraw.aapolygon(self.surf, transformed_coords, (51, 102, 0))
            gfxdraw.filled_polygon(self.surf, transformed_coords, (204, 102, 0))

            gfxdraw.aacircle(self.surf, int(x), int(y), int(0.083 * scale), (225, 225, 225))
            gfxdraw.filled_circle(
                self.surf, int(x), int(y), int(0.08 * scale), (0, 0, 0)
            )

        self.surf = pygame.transform.flip(self.surf, False, True)
        self.screen.blit(self.surf, (0, 0))
        if mode == "human":
            pygame.event.pump()
            self.clock.tick(FPS)
            pygame.display.flip()

        if mode == "rgb_array":
            return np.transpose(
                np.array(pygame.surfarray.pixels3d(self.screen)), axes=(1, 0, 2)
            )
        else:
            return self.isopen

# closing the pygame window after rendering

    def close(self):

        if self.screen is not None:
            pygame.display.quit()
            pygame.quit()
            self.isopen = False

# main environment class with ModelicaGym

class DymolaCSPendulumEnv(Pendulum_env, FMI2CSEnv):

# defining the ModelicaGym config-variables

    def __init__(self,
                 initangle1,
                 initangle2,
                 time_step,
                 positive_reward,
                 negative_reward,
                 log_level,
                 path):

        logger.setLevel(log_level)

        config = {
            'model_input_names': ['torque'],
            'model_output_names': ['angle1', 'angle2', 'anglevelocity1', 'anglevelocity2',
'position1', 'position2'],
            'model_parameters': {'initangle1': initangle1, 'initangle2': initangle2},
```

```
            'initial_state': (0, 0),
            'time_step': time_step,
            'positive_reward': positive_reward,
            'negative_reward': negative_reward,
            'path' : path
        }

        super().__init__(path, config, log_level)
```

A.2 Model Training

```
# OpenAI Gym
import gym
from gym.envs.registration import register

# ModelicaGym
from Pendulum_environment import DymolaCSPendulumEnv
import os
import numpy as np

# Agent (SB3)
from stable_baselines3 import A2C, PPO, DDPG, TD3
from stable_baselines3.common.noise import NormalActionNoise
from stable_baselines3.common.callbacks import EvalCallback, CheckpointCallback, CallbackList

# setting the config variables for environment definition
config = {
        'time_step' : 0.01,
        'initangle1' : -np.pi/2,
        'initangle2' : 0,
        'positive_reward' : 1,
        'negative_reward' : -1,
        'log_level' : 4,
        'path': r"C:\Users\Tim\switchdrive\BAT\Modelica\DoublePendulum1ms.fmu"
}

# creating the OpenAI Gym environment based on environment class
env_name = "DoublePendulum-v0"

register(
    id = env_name,
    entry_point = 'Pendulum:DymolaCSPendulumEnv',
    kwargs=config
)

env = gym.make(env_name)
eval_env = gym.make(env_name)

#defining saving paths
log_path = os.path.join('Training', 'DoublePendulum_Logs',
'Model16_TD3_10ms_192x2NN_1500k_sigma0003')
model_path = os.path.join('Training', 'DoublePendulum_Models',
'Model16_TD3_10ms_192x2NN_1500k_sigma0003')

# action-noise for off-policy algorithms
n_actions = env.action_space.shape[-1]
action_noise = NormalActionNoise(mean=np.zeros(n_actions), sigma=0.03 * np.ones(n_actions))

# defining agent callbacks while training (checkpoint_callback saves model every n timesteps,
eval_callback evaluates model every n timesteps with seperate environment and saves the best
model)
```

```
checkpoint_callback = CheckpointCallback(save_freq=50000, save_path=os.path.join('Training',
'DoublePendulum_Logs', 'CheckpointCallback_Model16'),
                                         name_prefix='checkpoint')

eval_callback = EvalCallback(eval_env, best_model_save_path=os.path.join('Training',
'DoublePendulum_Logs', 'EvalCallback_Model16'),
                             log_path=os.path.join('Training', 'DoublePendulum_Logs',
'EvalCallback_Model16'), eval_freq=50000,
                             deterministic=True, render=False)

callback = CallbackList([checkpoint_callback, eval_callback])

# defining agent neural network architecture
policy_kwargs = dict(net_arch=[288, 288])

# defining agent model
model = TD3('MlpPolicy', env, verbose=1, policy_kwargs=policy_kwargs, train_freq=1024,
action_noise=action_noise, learning_starts=20000, learning_rate=0.003,
tensorboard_log=log_path)
# training the agent
model.learn(total_timesteps=1500000, callback=callback)
# saving the agent model after training
model.save(model_path)

# testing the environment with random action and obervation sample (deactivate agent
functions!)
"""
print("\n\nAction Space: ")
print(env.action_space)
print("\nObservation Space: ")
print(env.observation_space)
print("\n\n")

print(env.action_space.sample())
print(env.observation_space.sample())
"""
# environment test and rendering with random actions (deactivate agent functions!)
"""
episodes = 1
for episode in range(1, episodes+1):
    state = env.reset()
    done = False
    score = 0

    while not done:
        env.render()
        action = env.action_space.sample()
        n_state, reward, done, info = env.step(action)
        score+=reward
    print('Episode:{} Score:{}'.format(episode, score))
    env.render()
env.close()
"""

# view tensorboard log with cmd-command: tensorboard --logdir=logfile_path
```

A.3 Model Testing

```
# OpenAI Gym
import gym
from gym.envs.registration import register
```

```
# ModelicaGym
from Pendulum_environment import DymolaCSPendulumEnv
import os
import numpy as np

# Agent (SB3)
from stable_baselines3 import A2C, PPO, DDPG, TD3
from stable_baselines3.common.evaluation import evaluate_policy

# setting the config variables for environment definition
config = {
        'time_step' : 0.01,
        'initangle1' : -np.pi/2,
        'initangle2' : 0,
        'positive_reward' : 1,
        'negative_reward' : -1,
        'log_level' : 4,
        'path': r"C:\Users\Tim\switchdrive\BAT\Modelica\DoublePendulum03.fmu"
}

# creating the OpenAI Gym environment based on environment class
env_name = "DoublePendulum-v0"

register(
    id = env_name,
    entry_point = 'Pendulum:DymolaCSPendulumEnv',
    kwargs=config
)

env = gym.make(env_name)

# Loading Model
model_path = os.path.join('Training', 'DoublePendulum_Logs', 'EvalCallback_Model16',
'best_model')

model = TD3.load(model_path, env)

# Testing Model
print(evaluate_policy(model, env, n_eval_episodes = 1, deterministic=True,
render=True))      # returns (mean_reward, std_reward)

# get action, state and reward values each simulation step for plotting (deactivate "Testing
Model (evaluate_policy)")
"""
obs = env.reset()
actionvalues = []
position2values = []
rewardvalues =[]
actionhalf1 = []
position2half1 = []
rewardhalf1 = []
actionhalf2 = []
position2half2 = []
rewardhalf2 = []
for i in range(3001):
    action, _states = model.predict(obs, deterministic=True)
    obs, rewards, dones, info = env.step(action)
    action = round(action[0], 2)
    rewards = round(rewards, 2)
    height = round(obs[5], 2)
    actionvalues.append(action)
```

```
    position2values.append(height)
    rewardvalues.append(rewards)
# not all variables can be printed out at once
for i in range(0, 1500):
    actionhalf1.append(actionvalues[i])
    position2half1.append(position2values[i])
    rewardhalf1.append(rewardvalues[i])
for i in range(1500, 3001):
    actionhalf2.append(actionvalues[i])
    position2half2.append(position2values[i])
    rewardhalf2.append(rewardvalues[i])
print("\n\nrewards 1:\n", rewardhalf1)
print("\n\nrewards 2:\n", rewardhalf2)
print("\n\nactions 1:\n", actionhalf1)
print("\n\nactions 2:\n", actionhalf2)
print("\n\nstates 1:\n", position2half1)
print("\n\nstates 2:\n", position2half2)
"""
```

A.4 ModelicaGym changes

To reset the starting position of the pendulum after each episode, the file modelica_base_env from ModelicaGym must be slightly changed at the following function definition:

```
    def _set_init_parameter(self):
        """
        Sets initial parameters of a model.

        :return: environment
        """
        # modelparameters for DoublePendulum Environment
        initangle1 = -np.pi/2
        initangle2 = 0
        modelparameters = initangle1, initangle2

        if self.model_parameters is not None:
            self.model.set(list(self.model_parameters),
                           list(modelparameters))
        return self
```

Appendix B: Pure Python Environment Code

```
import gym
from gym import Env
from gym import spaces
from gym import register
import numpy as np
from stable_baselines3 import A2C, PPO, DDPG
from stable_baselines3.common.evaluation import evaluate_policy
import os

class CustomEnv(Env):

    def __init__(self):

        # defining action-space and observation-space
        self.action_space = spaces.Box(low = np.array([-1]), high = np.array([1]))
        self.observation_space = spaces.Box(low=np.array([0, 0]), high=np.array([14.097, 14]))

        # set random target state
        self.target = np.random.uniform(0, 14)
        self.target_low = self.target-0.01
        self.target_high = self.target+0.01

        # setting simulation length
        self.sim_length = 3

        # init state
        self.state = [0, self.target]

    def step(self, action):

        # calculating system state at current step
        self.state = [14.097*action-0.0004, self.target]
        self.obs, self.targetvalue = self.state

        # updating current simulation length
        self.sim_length -= 1

        # reward-function
        if self.obs >=self.target_low and self.obs <=self.target_high:
            reward = 1
        else:
            reward = -1

        # episode termination check
        if self.sim_length <= 0:
            done = True
        else:
            done = False

        info = {}

        print(self.state, reward)

        return self.state, reward, done, info

    def reset(self):
        # reset simulation length
        self.sim_length = 3

        # set new random target state
```

```
        self.target = np.random.uniform(0, 14)
        self.target_low = self.target-0.01
        self.target_high = self.target+0.01

        # reset system state
        self.state = [0, self.target]

        return self.state

# defining environment without register
env = CustomEnv()

# setting log and model path
log_path = os.path.join('Training', 'Logs')
model_path = os.path.join('Training', 'Saved Models', 'DDPG_CustomEnv3')

# training the agent
model = DDPG('MlpPolicy', env, tensorboard_log=log_path, verbose = 1)
model.learn(total_timesteps=100000)
model.save(model_path)

# testing the agent
model = DDPG.load(model_path, env)
print(evaluate_policy(model, env, n_eval_episodes=10))
```

Appendix C: FMU Simulation Test with PyFMI Code

```
import pyfmi
from pyfmi import load_fmu
import os

# define path
model_path = r"C:\Users\Tim\switchdrive\BAT\Modelica\Simplependulum.fmu"
model_name = model_path.split(os.path.sep)[-1]

# load model
model = load_fmu(model_path)

# set options
opts = model.simulate_options()     # take all default options
opts['ncp'] = 1                     # change number of communication/output points
opts['initialize'] = False          # initialize model when model.simulate is executed

# set initial values for input variables
model.set("torque", 0)
model.set("initangle", 3)

# initialize model
model.reset()
model.setup_experiment(start_time = 0)
model.initialize()

# simulate model
result = model.simulate(start_time = 0, final_time = 50, options=opts)

# printing states for each ncp
print(result['time'])
print(result['angle'])
```

Appendix D: ModelicaGym Valve Environment Code

D.1 Environment

```
# specify working directory for library import
import sys
sys.path.append(".")

# ModelicaGym
import logging
import numpy as np
from gym import spaces
from modelicagym.environment import FMI2CSEnv

# logger definition
logger = logging.getLogger(__name__)

# Environment definition based on OpenAI Gym framework

class MassflowValve_env:

# defining the episode termination

    def _is_done(self):

        if self.stop >= 5:
            done = True
        else:
            done = False
        return done

# defining the action-space (used in modelica_base_env / __init__)

    def _get_action_space(self):

        return spaces.Box(low = 0, high = 1, shape = (1,), dtype = "float32")

# defining the observation-space (used in modelica_base_env / __init__)

    def _get_observation_space(self):

        return spaces.Box(low=np.array([0, 0]), high=np.array([14, 10]), dtype= "float32")

# refering to the step function of ModelicaGym

    def step(self, action):

        return super().step(action)

# refering to the reset function of ModelicaGym

    def reset(self):

        return super().reset()

# defining the reward-function

    def _reward_policy(self):

        massflow, target = self.state
```

```
        target_low = target - 0.5
        target_high = target + 0.5
        logger.debug("massflow: {0}, target: {1}".format(massflow, target))

        if massflow >= target_low and massflow <= target_high:
            reward = self.positive_reward

        else:
            reward = self.negative_reward

        return reward

# main environment class with ModelicaGym

class DymolaCSMassflowEnv(MassflowValve_env, FMI2CSEnv):

# defining the ModelicaGym config-variables

    def __init__(self,
                 target_input,
                 noise,
                 time_step,
                 positive_reward,
                 negative_reward,
                 log_level,
                 path=r"C:\Users\Tim\switchdrive\BAT\Modelica\Massflow_with_Valve_noise1"):

        logger.setLevel(log_level)

        target_input = np.random.uniform(0, 10)
        noise = np.random.uniform(0.9, 1)

        config = {
            'model_input_names': ['valve_opening'],
            'model_output_names': ['massflow', 'target'],
            'model_parameters': {'target_input': target_input, 'noise': noise},
            'initial_state': (0, 0),
            'time_step': time_step,
            'positive_reward': positive_reward,
            'negative_reward': negative_reward
        }

        super().__init__(path, config, log_level)
```

D.2 Model Training

```
# OpenAI Gym
import gym
from gym.envs.registration import register

# ModelicaGym
from MassflowValve_environment import DymolaCSMassflowEnv
import os
import numpy as np
from Massflow_with_Valve.MassflowValve_environment import MassflowValve_env

# Agent (SB3)
from stable_baselines3 import A2C, PPO, DDPG, TD3
from stable_baselines3.common.noise import NormalActionNoise
from stable_baselines3.common.callbacks import EvalCallback
```

```
# initializing noise and target
noise = np.random.uniform(0.9, 1)
target_input = np.random.uniform(0, 10)

# setting the config variables for environment definition
config = {
        'time_step' : 1,
        'noise' : noise,
        'target_input' : target_input,
        'positive_reward' : 2,
        'negative_reward' : -1,
        'log_level' : 4,
        'path': r"C:\Users\Tim\switchdrive\BAT\Modelica\Massflow_with_Valve_target_noise.fmu"
}

# creating the OpenAI Gym environment based on environment class
env_name = "MassflowValve-v2"

register(
    id = env_name,
    entry_point = 'Massflow_with_Valve:DymolaCSMassflowEnv',
    kwargs=config
)

env = gym.make(env_name)
eval_env = gym.make(env_name)

#defining saving paths
log_path = os.path.join('Training', 'Logs', 'TD3_MassflowValve_target_noise')
model_path = os.path.join('Training', 'Saved Models', 'TD3_MassflowValve_target_noise')

"""
# For trying out the environment with random samples

while not done:
    action = env.action_space.sample()
    n_state, reward, done, info = env.step([action])
    score += reward
    print('Score:{}'.format(score))

"""
# defining agent callback while training (eval_callback evaluates model every n timesteps with
seperate environment and saves the best model)
eval_callback = EvalCallback(eval_env, best_model_save_path=os.path.join('Training', 'Saved
Models', 'TD3_MassflowValve_target_noise'),
                             log_path=os.path.join('Training', 'Logs',
'TD3_MassflowValve_target_noise'), eval_freq=100, n_eval_episodes=3,
                             deterministic=True, render=False)

# action-noise for off-policy algorithms
n_actions = env.action_space.shape[-1]
action_noise = NormalActionNoise(mean=np.zeros(n_actions), sigma=0.1 * np.ones(n_actions))

# defining agent model
model = TD3('MlpPolicy', env, verbose=1, action_noise=action_noise, tensorboard_log=log_path)
# training the agent
model.learn(total_timesteps=20000, callback = eval_callback)
# saving the agent model after training
model.save(model_path)

# view tensorboard log with cmd-command: tensorboard --logdir=logfile_path
```

D.3 Model Testing

```
# OpenAI Gym
import gym
from gym.envs.registration import register
# ModelicaGym
from MassflowValve_environment import DymolaCSMassflowEnv
import os
import numpy as np
# Agent (SB3)
from stable_baselines3 import A2C, PPO, DDPG, TD3
from stable_baselines3.common.evaluation import evaluate_policy

# initializing noise and target
noise = np.random.uniform(0.9, 1)
target_input = np.random.uniform(0, 10)

# setting the config variables for environment definition
config = {
        'time_step' : 0.5,
        'noise' : noise,
        'target_input' : target_input,
        'positive_reward' : 3,
        'negative_reward' : -1,
        'log_level' : 4,
        'path': r"C:\Users\Tim\switchdrive\BAT\Modelica\Massflow_with_Valve_noise1.fmu"
}

# creating the OpenAI Gym environment based on environment class
env_name = "MassflowValve-v1"

register(
    id = env_name,
    entry_point = 'Massflow_with_Valve:DymolaCSMassflowEnv',
    kwargs=config
)

env = gym.make(env_name)

# Loading Model
model_path = os.path.join('Training', 'Saved Models', 'TD3_MassflowValve_target_noise')

model = TD3.load(model_path, env)

# Testing Model
print(evaluate_policy(model, env, n_eval_episodes = 10))      # returns (mean_reward,
std_reward)
```

D.4 ModelicaGym changes

To reset the target value and the noise after each episode, the file modelica_base_env from ModelicaGym must be slightly changed at the following function definition:

```
    def _set_init_parameter(self):
        """
        Sets initial parameters of a model.

        :return: environment
```

```
"""
# modelparemters for MassflowValve Environment
target_input = np.random.uniform(0, 10)
noise = np.random.uniform(0.9, 1)
modelparameters = target_input, noise

if self.model_parameters is not None:
    self.model.set(list(self.model_parameters),
                   list(modelparameters))
return self
```