Markus Novak, BSc

Machine Learning based Air Path Control for Diesel Engines

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Assoc.Prof. Dipl.-Ing. Dr.techn. Markus Reichhartinger

Institute of Automation and Control
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Martin Horn

Graz, April 2021

AFFIDAVIT

Abstract

Designing control systems for complex non-linear problems like the air-path of a diesel engine is a challenging and time-consuming task. The vision behind this thesis is to investigate whether an easy-to-use and cost-effective solution for this problem by means of machine-learning-based methods can be provided. Therefore, the problem setting of an air-path control for diesel engines is analyzed and machine learning techniques in the context of controller design are discussed with a special focus on reinforcement learning. Within this framework, a new technique was developed to build data-driven reinforcement learning controllers, that learn an optimal control strategy on their own by interaction with a provided simulation model. The learning capabilities of these controllers, as well as several design choices, were evaluated on a small selected benchmark problem for two specific use-cases. Finally, the developed methodology was applied to a provided highly accurate diesel engine simulation model and compared to an existing control strategy consisting of gain-scheduled PID-controllers. The presented results can be regarded as a proof of concept of data-driven approaches in the field of engine control and provide a solid basis for further research projects.

Kurzfassung

Der Entwurf von Regelungssystemen für komplexe nichtlineare Probleme, wie die Luftpfadregelung eines Dieselmotors, ist eine anspruchsvolle und zeitintensive Aufgabe. Die Vision hinter dieser Arbeit ist es, zu untersuchen, ob eine einfach zu handhabende und kostengünstige Lösung für dieses Problem mittels maschineller Lernmethoden bereitgestellt werden kann. Dazu wird die Problemstellung einer Luftpfadregelung für Dieselmotoren analysiert und maschinelle Lernverfahren im Kontext des Reglerentwurfs mit speziellem Fokus auf Reinforcement Learning untersucht. In diesem Rahmen wurde eine neue Technik entwickelt, um datengetriebene Regler mittels Reinforcement Learning zu erzeugen, die durch Interaktion mit einem bereitgestellten Simulationsmodell selbständig eine "optimale" Regelstrategie erlernen. Die Lernfähigkeiten dieser Regler sowie verschiedene Designentscheidungen wurden an einem Benchmark-Problem für zwei spezifische Anwendungsfälle evaluiert. Schließlich wurde die entwickelte Methodik mit einem bereitgestellten hochgenauem Dieselmotor-Simulationsmodell umgesetzt und mit einer bestehenden Regelstrategie, bestehend aus PID-Reglern mit arbeitspunkt-abhängiger Parametersteuerung (Gain Scheduling), verglichen. Obwohl die Ergebnisse nicht perfekt sind, dienen sie als valider Konzeptnachweis für den gewählten datengetriebenen Ansatz und bieten eine solide Grundlage für weitere Forschungsprojekte.

Acknowledgements

This master's thesis was conducted at AVL List GmbH in Graz, Austria from August 2020 to April 2021.

At this point I would like to thank Dirk Denger and Alois Danninger for the opportunity to write this thesis. Special thanks goes to my supervisors Debora Maria Short Sotero and Bernhard Breitegger for the excellent guidance and technical support over the whole time. I would also like to thank Christina Schwarz, Michael Steger and the whole Systems Engineering Laboratory team at AVL for many interesting discussions and inputs.

For the technical support and many helpful comments on this text I would like to thank my supervisor Assoc.Prof. Dipl.-Ing. Dr.techn. Markus Reichhartinger at the Institute of Automation and Control at the Graz University of Technology.

Finally, my biggest thankfulness goes to my family and to my girlfriend Carmen for their emotional support during the process of writing this thesis.

Contents

List of Figures

List of Tables

1 Introduction

There exist several state-of-the-art approaches to design control systems for complex non-linear problems like the air-path of a diesel engine. Methods that have proven to work well include classical approaches with (multiple) PID-controllers or model-based approaches in which the proprieties of the physical system are modeled to calculate optimal controller responses (for instance Model Predictive Control, see [1] or [2]). However, the process of accurately modeling a technical system so that it can be executed on standard electronic engine control units (ECUs) typically is challenging and time-consuming. Approaches with PID controllers on the other hand are hard to calibrate and in both cases experienced engineers are required to either design or tune the system by hand.

The vision behind this thesis is to use machine-learning-based methods to build a data-driven control system that saves time in development and calibration. In the last years, machine learning has proven to be effective in many areas (see Chapter 2.1). In this thesis, existing machine learning techniques in the context of controller design are investigated with special focus on reinforcement learning. The concrete goal is to develop a methodology for building controllers that learn the "best" control strategy on their own and are still able to run on engine control units.

For training the machine learning models and evaluating their performance, an accurately calibrated simulation model of a heavy-duty diesel engine with an existing state-of-the-art air-path control system was provided by AVL List for this thesis.

In this chapter, a brief introduction to diesel engines and the main components of a diesel engine air-path is provided. Furthermore, the structure and working principle of an existing air-path control system is explained, so that it can be defined which parts of the software should be replaced in the data-driven control approach.

1.1 Air Path in Diesel Engines

This introductory section starts by giving some theoretic knowledge about diesel engines as well as an explanation of the main parts of a typical diesel engine air-path.

1.1.1 Diesel Engines

The diesel engine[1] is an internal-combustion engine with self ignition. It works by mixing highly compressed air and fuel inside a combustion chamber. Due to the high temperatures of the compressed air, the injected diesel fuel ignites on its own without the need for an additional ignition source. Therefore, heat is used to release chemical energy contained in the fuel and convert it into mechanical energy [3, p. 16].

To this day, it is the most widely used combustion engine due to its comparatively high degree of efficiency and resulting fuel economy. Large and slow-running engines reach real efficiency values of more than 50% [3, p. 17].
Diesel engines are used in fixed-installations, for instance driving power generators, and in mobile applications, such as passenger cars, commercial vehicles, construction and agriculture machinery as well as locomotives and ships. Although a decrease in diesel engine powered cars is observable over the last years, in 2019 diesel engines were still used in 42.3% of all passenger cars and 97.8% of all medium and heavy commercial vehicles in the European Union [4].

1.1.2 Air Path Components

The combustion of diesel fuel requires oxygen, which is taken from the air coming from the engine air intake. A general rule for the combustion process is, that the more oxygen is available, the greater the amount of fuel that can be injected to deliver the required load demands. This means that there is a direct relationship between the amount of air in the cylinder and the maximum possible engine power output [5, p. 46].

Due to this relationship, diesel engines are particularly suited to to be charged by means of a turbocharger, which is fitted to most modern engines. Charging means that the air is already pre-compressed before it enters the cylinders. This not only increases the power output and improves the engine efficiency, but also reduces pollutants in the exhaust gas [3, p. 16].

To additionally reduce NO_X emissions[2] from passenger cars and commercial vehicles, a portion of the exhaust gas is fed back to the engine intake manifold. This is called exhaust gas recirculation (EGR) [3, p. 16].

Figure 1.1 shows the components of a typical air-path for diesel engines and the air flow within it. Air comes in through an air intake filter (1), which prevents any particles or dust from entering the engine and protects the sensitive air-mass sensors that follow later. In the intake also sensors for the ambient air temperature t_{amb} and pressure p_{amb} are situated. The air is then compressed in the compressor-part of the turbocharger (2). Due to the compression, the air also heats up, which has a negative effect on the

[1]Named after Rudolf Diesel (1858 to 1913)
[2]NO_X is used as a short term for nitrogen oxide (NO) and nitrogen dioxide (NO_2).

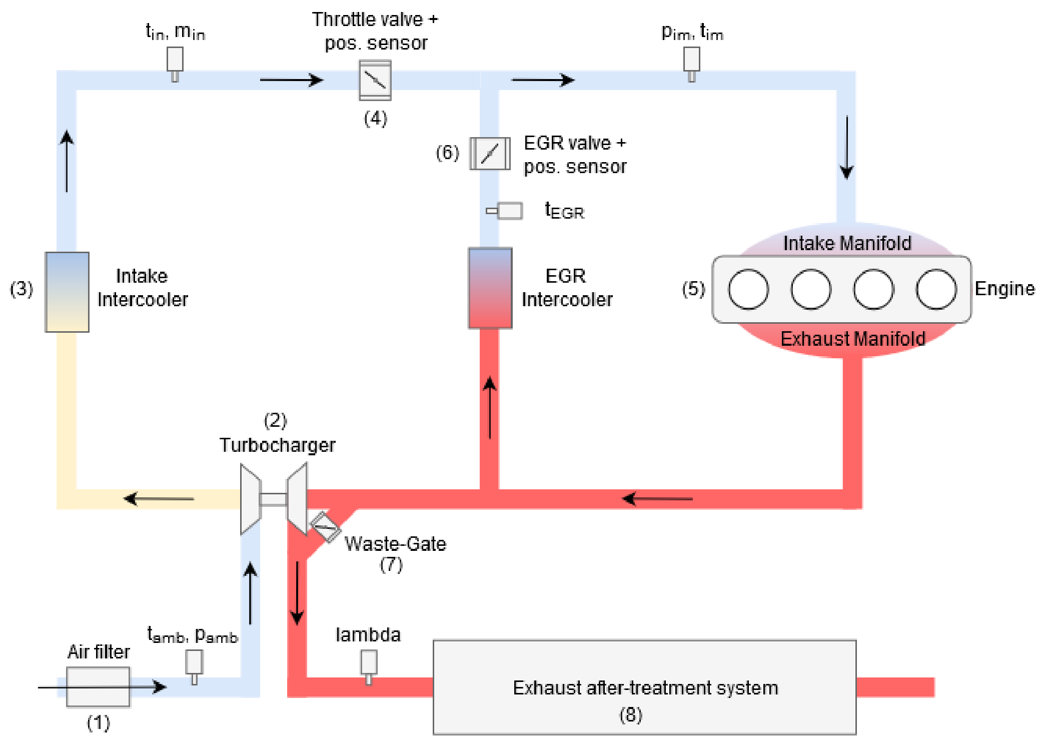

Figure 1.1: Air-path in a typical heavy-duty diesel engine with turbocharger and exhaust-gas recirculation.

maximum air-charge that fits into the cylinder. Therefore, an intercooler (3) is positioned after the turbocharger that is cooled either by ambient air or a separate coolant circuit. After the intake cooler, sensors for the intake temperature t_{in} and air-mass flow $\dot{m}_{in}$ are placed. A position controlled throttle valve (4) follows to restrict the air flow from the intake if needed. Before the air enters the intake manifold of the engine, sensors are situated to measure the intake manifold temperature t_{im} and pressure p_{im}. Inside the engine (5), diesel fuel is injected and the combustion process takes place. A part of the resulting exhaust gas is cooled and fed back through a position controlled EGR valve (6). Due to the cooling, a greater reduction of emissions can be achieved. However, the main part of the exhaust gas either goes through the turbine-part of the turbocharger, or is by-passed via the so called waste-gate (7). Afterwards, a lambda-sensor is placed to measure the remaining oxygen in the exhaust-gas stream. Finally, the the exhaust gas is cleaned by an exhaust after-treatment system (8) to reduce harmful emissions before leaving through the tail-pipe.

The main components of an air-path system are further described in the following paragraphs.

Turbocharger As already seen in part (2) of Figure 1.1, a turbocharger is actually a combination of two turbo elements: A radial compressor that compresses the intake air and a turbine that is driven by the flow of exhaust gas and is coupled to the compressor by a shaft.

In older diesel engines[3], a large amount of energy was lost by simply expelling the hot and pressurized exhaust gas from the engine. Therefore, it makes sense to utilize some of that energy to generate pressure in the intake path and thus increasing the amount of air. By charging the intake air before it enters the cylinders, more oxygen is available for combustion, which results in a higher maximum torque and therefore a greater power output from the same engine, or the same power output from a smaller engine (downsizing) [5, p. 47].

The design of turbochargers is explained in [5, pp. 49–50]. Diesel engines that do not run under steady-state conditions, especially cars and commercial vehicles, need to be able to generate high torque even at low engine speeds. Therefore, turbochargers are designed for low exhaust-gas flow-rates. To prevent the turbocharger from damaging itself or overloading the engine at higher flow rates, the pressure that is generated (also called boost pressure) has to be controlled. There are two main turbo charger designs to achieve this:

- A *wastegate* turbocharger works by diverting a part of the exhaust-gas flow past the turbine by a bypass valve, the so called "wastegate". As a result, the exhaust-gas back pressure is decreased and excessive turbocharger speeds are avoided. The wastegate is closed at low engine speeds to direct the entire exhaust-gas flow through the turbine.
- Another possibility to control the turbocharger performance is by means of *Variable Turbine Geometry* (VTG). Thereby, adjustable deflector blades change the flow cross section of the turbine by changing the size of the gap through which the exhaust gas can flow. Only a small flow cross section is allowed at low engine speeds or loads, such that a high velocity of the exhaust-gas flow is reached and the turbine turns at a higher speed. A larger flow cross-section results in a lower flow velocity of the exhaust-gas stream and as a result the turbocharger turns more slowly. This way, the boost pressure is limited at high engine speeds or loads.

The engine simulation model provided by AVL for this thesis supports both versions of turbochargers, in order to simulate different engine setups. However, in this thesis only the wastegate-version will be used since the technique and also the model is simpler and therefore can be simulated more efficiently.

Exhaust Gas Recirculation An effective internal engine measure to lower NO_X emissions in diesel engines is to recirculate a part of the exhaust-gas flow into the combustion chamber through an additional line and a control valve (seen in part (6) of Figure 1.1).

[3]Actually, Rudolf Diesel (1896), already considered the precompression of intake air to improve performance, but it took until the 1970s until turbocharges became widespread in cars [5, p. 47].

The main reasons why this lead to a reduction of NO_X are described in [6, p. 196]. First of all, the recirculation naturally leads to a reduction of the exhaust-gas mass flow that leaves through the tail-pipe. Additionally, high local temperatures and high partial oxygen pressures are required to form NO_X during the combustion process and both can be lowered by mixing in exhaust-gas. Due to the following higher rate of inert gases, the rate of combustion and the generation of high peak temperatures is reduced. However, by reducing the oxygen content in the combustion process, the amount of smoke increases, which limits the quantity of how much exhaust-gas can be recirculated.

The EGR volume depends on the angle position of the EGR valve as well as the pressure difference between the point where the exhaust-gas is taken (before the turbine) and the point it is mixed in (before the engine intake manifold). As a result, the turbocharger does not only control the intake manifold pressure but also has an influence on the EGR rate. Due to this influence, it is sometimes necessary to restrict the gas flow on the intake-manifold side, especially when the engine load is low, to achieve desirably high EGR rates. In most heavy-duty engines, and also the AVL engine simulation model, this is done by a throttle valve (seen in part (4) of Figure 1.1). However, since the purpose of this throttle valve is to additional influence the intake manifold pressure, a cross interaction with the turbocharger control exists.

1.2 Current Control Concept

This section explains the current state-of-the-art control concept for the air-path of a heavy-duty diesel engine with exhaust-gas recirculation and a turbocharger, as it is implemented in the AVL simulation model that was provided for this thesis. This helps to understand the physical relationships in the system and the software changes that can be made by the use of machine learning techniques.

When speaking about control strategies, a general concept of controlling the air-path actuators either via closed or open loop is meant. The strategy should control all involved actuators simultaneously in order to reach low engine emissions at transient engine operation, while ideally not interfering with the performance and torque response of the engine. This requires a modular structure, so that individual functions do not have a direct influence on the overlying engine management system. Such a system typically works by receiving a desired torque from the driver via the accelerator pedal, which is added to the torque requirements of various internal systems (for instance alternator or air conditioner) and the total torque demand on the engine is then given to the fuel-injection and air-path control [7, p. 249]. These control systems then work together to provide sufficient fuel and air to the engine to meet the total torque demand, while also taking into account possible trade-offs due to engine emissions limits and fuel consumption.

1.2.1 Characteristic values

To meet the discussed requirements, characteristic values are defined based on the physical relationships of the engine to form control variables. Based on the torque demands and measured sensor data, demand values for these control variables are calculated and controllers are implemented to follow these demand values.

Boost pressure As already discussed in Section 1.1.2, the characteristic value for controlling the turbocharger is the pressure in the intake manifold of the engine p_{im}, also called boost pressure. The optimum boost pressure is a function of engine speed, injected fuel quantity, surrounding air pressure as well as coolant and fuel temperatures [7, p. 252]. Deviations of the measured boost pressure to these demands are then compensated by controlled opening or closing of the wastegate valve.

Lambda For air-path and injection management systems, the excess-air factor λ of the engine can be used as a characteristic value. In [6, p. 184] it is defined as the ratio of intake-air mass to the air mass required for stoichiometric combustion, hence

$$\lambda = \frac{m_{\text{Air}}}{m_{\text{Fuel}} \cdot r_{\text{st}}}, \tag{1.1}$$

where m_{Air} and m_{Fuel} are the respective masses of air and fuel in the cylinder and r_{st} is the stoichiometric ratio that describes the mass of air in kg that is required to completely burn 1 kg of fuel (approximately 14.5 for diesel fuel). This means that if the excess-air factor is:

- $\lambda = 1$, the intake-air mass is equal to the air mass theoretically required to burn all of the fuel injected,
- $\lambda < 1$, there is too less intake-air mass to burn all the fuel (called rich mixture) and
- $\lambda > 1$, there is more intake-air mass than theoretically needed (called lean mixture).

Unlike the boost pressure in the intake manifold, lambda is usually not measured directly but is calculated as in Equation (1.1) by measuring the mass of air and injected fuel. The lambda oxygen sensor in the exhaust pipe (seen in Figure 1.1) measures the residual oxygen content in the exhaust gas and is only used as a plausibility check and correction for the calculated lambda.

Typical lambda levels for turbocharged diesel engines are within a range of $\lambda = 1.15$ to $\lambda = 2.0$ [6, p. 184]. In this range, a closed loop control of lambda can be done. However, if the mass of injected fuel gets very low (for example when the engine is idling), according to Equation (1.1) lambda suddenly approaches very high values, due to the small load. In such cases, a closed loop control is no longer possible and a switch to an open loop control mode has to be made. The actuators to control lambda are the EGR valve and intake throttle valve (seen in parts (4) and (6) of Figure 1.1). The EGR valve is mainly responsible for adjusting deviations between measured (or calculated) lambda value and demand value, but if the EGR valve is already fully open, the mass of fresh air and therefore also lambda can be further decreased by closing the intake throttle valve. Thereby the rate of recirculated gas is increased.

1.2.2 Controller structure

Apart from the fact that lambda control happens with two controllers, one for each actuator, the controller structure for boost pressure and lambda control are nearly identical. The controllers consists of two main parts, a pre-control map for an open-loop signal and a gain-scheduled PID controller with anti wind-up measures for a closed-loop signal. Therefore, an exemplary controller for all three actuator signals is illustrated in figure 1.2.

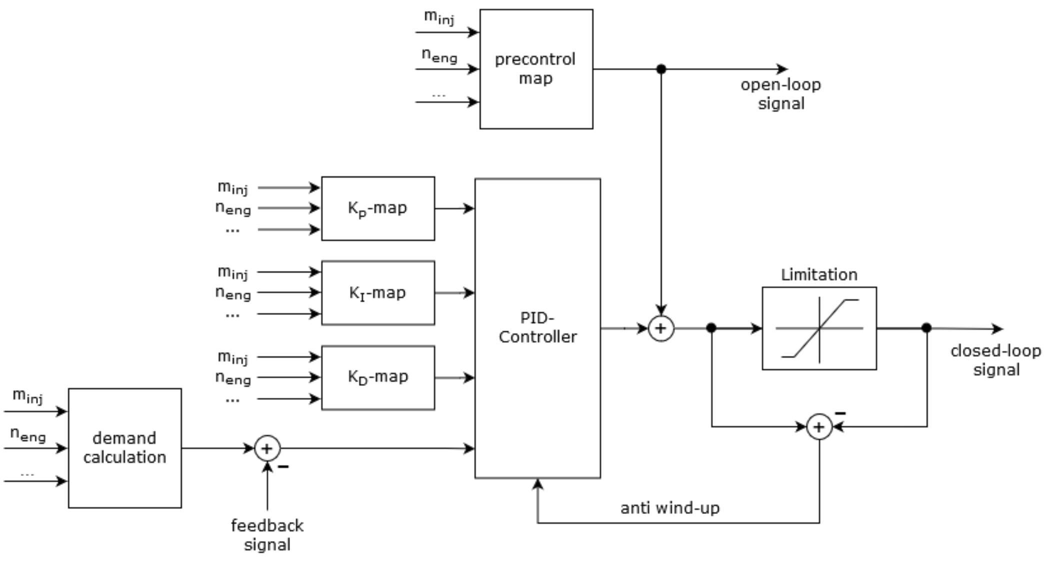

Figure 1.2: Block diagram of an exemplary gain-scheduled PID-controller.

The respective demand values are calculated via maps, based on the currently injected fuel mass-flow $\dot{m}_{\text{inj}}$ and engine speed n_{eng} and are corrected on the basis of measured temperatures and pressures in the air-intake as well as ambient air conditions. From this demand signal, the corresponding feedback value (measured boost pressure or calculated lambda) is subtracted and the resulting error signal is used as input for a gain-scheduled PID-controller. The PID-gain-parameters K_p, K_i and K_d also come from maps that are based on the current engine speed n_{eng}, the injected fuel mass-flow $\dot{m}_{\text{inj}}$ and other parameters that are used as for fine-tuning and corrections. Similarly, a pre-control signal for the actuator position is determined via a map. This position demand can be used either in open-loop operation modes or as a pre-control value for closed-loop control, where it is added to the output of the PID controller. Also, due to a limitation of the valve-positions that can only be open between 0 and 100%, anti wind-up measures have to be taken for the PID controller, so that it does not overshoot too much because of the integral term winding up for large setpoint changes.

Dynamic limitation Following the open- and closed-loop signal generation, the actuator demand values are already limited to be within 0 to 100% of the maximum valve-position (in case of the open-loop signal due to calibration, in case of the closed-loop signal due to a limitation-block). However, there can be situations like rapid drops in lambda due to acceleration, which would lead to high emissions, or special operation conditions (for instance engine start or motoring mode) that require a dynamic limitation of the minimum or maximum valve positions. For example, the minimum position of the EGR valve could be restricted to prevent smoke generation. These dynamic limitations are done after the control signal generation, and are fed back to the PID controller as "anti wind-up" signals.

Arbitration Before a output signal can be given out by the control software, it needs to decide whether the open-loop or closed-loop signal should be used. This is done by an arbitration that normally uses the closed-loop signal but also detects operating conditions that require open-loop control (for instance motoring mode where no fuel is injected) and switches the output signal accordingly.

1.2.3 Calibration

The control concept heavily depends on the correct calibration of all controller maps that can be seen in Figure 1.2. Typically, this is done for a set of steady-state engine operation points that are defined by a given engine speed and torque demand. This means that in multiple experiments the operation point of the engine is held constant for some time, followed by a step to some other operation point. The engine parameters, for instance the values in the maps, are then calibrated based on these step experiments. This method can be seen as a point-wise linearization of the non-linear engine behaviour around the selected operation points. The signals that define an engine operation point are those who have the biggest impact on the system behaviour. In Figure 1.2 it can be seen that the engine speed has direct influence on all controller maps and the total torque demand (also called load of the engine) is closely correlated with the injected fuel mass, which is also a direct input to all air-path control maps.

However, due to the fact that the actuators for lambda control also influence the boost pressure and the actuator for boost pressure control also has an influence on lambda, the calibration process for the controller parameters is challenging and very time consuming.

AVL has developed processes that allow the consideration of such cross-coupling influences in multi-variable control problems. But even with specialized calibration processes it takes a lot of time and effort of experienced commissioning engineers to find a "good" calibration on engine test-beds.

1.3 Changes in the control structure

As already stated in the introduction, the aim of this thesis is to develop a controller structure that can learn an optimal control strategy on its own in a data-driven approach. Therefore, the general concept of the existing controller structure is adopted with some changes.

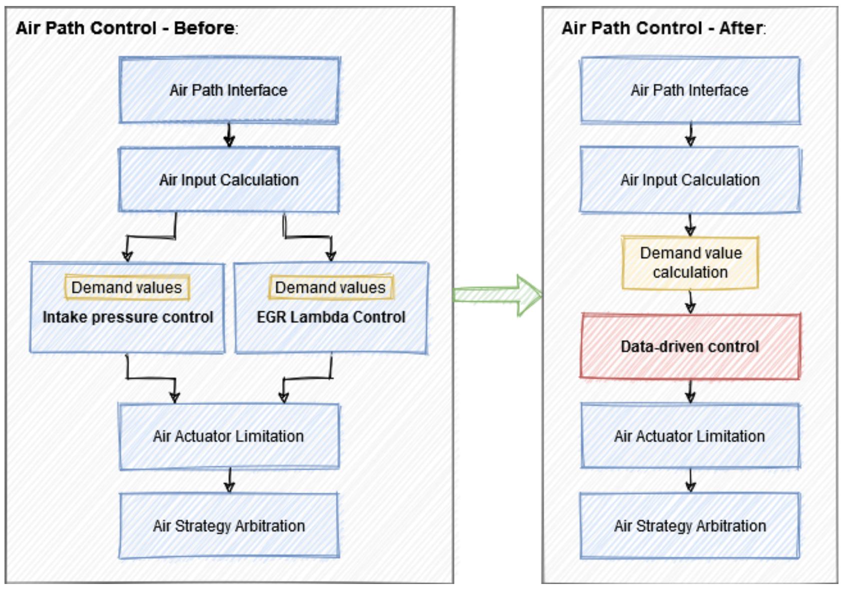

Figure 1.3: Planned changes in the control software.

The existing structure of the control software is built in multiple layers, as it can be seen on the left side of Figure 1.3. It starts with an interface that clearly defines the inputs to the system for other software parts. After that, an air-input calculation follows where common important values, for example the current EGR rate and lambda value, are calculated. Then, the actual controller structures for intake boost pressure and lambda control are placed, followed by a common dynamic actuator limitation and arbitration, as described before in Section 1.2.2.

For the new structure, large parts of the existing layout can be incorporated. The right side of Figure 1.3 shows the proposed structure of the control software. Main requirement is to replace the two separate controller structures into a single "data-driven control" block. The characteristic values of the control problem should stay the same, so the goal of the new controller will also be to follow a given boost pressure and lambda demand. These demand values can be taken from the existing demand value calculation maps.

Strategies to build such a data-driven control will be discussed in the following chapters.

2 Theory

In this chapter an overview of the theoretical aspects of this thesis is given. The main aspects of machine learning are described and it is discussed why reinforcement learning is particularly useful for the given problem setting. Following this, some of the concrete algorithms that are used in this work are discussed.

2.1 Machine Learning Basics

Since the beginning of programmable computers, people have been wondering if these machines could one day become more intelligent than humans. The field of trying to build such systems is called *Artificial Intelligence* (AI). There have been some notable practical advances in building intelligent systems over the last years. For example, in 2015 Mnih et al. built an algorithm that was able to play classic Atari games on a level of a professional human games tester [8], in 2016 the AlphaGo program was able to beat the human world champion in Go [9] and in 2019 the OpenAI Five system defeated a team of human world champions at the competitive e-sports game Dota 2 [10]. The similarity between all these achievements is that they all use *Machine Learning* (more precisely deep reinforcement learning which is a combination of classical reinforcement learning with function approximation methods from supervised learning). Nevertheless, the ultimate goal of a truly intelligent system, that is not limited to a single task or even behaves like a human, still seems like a long way to go.

According to Murphy [11, p. 1] machine learning can be defined as:

> [. . .] a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision making under uncertainty.

This automatic pattern detection and decision making is becoming increasingly important because we are living in an era of so called big data. What is meant by this term is that the amount of available data in all sorts of applications has been increasing drastically in the last two decades. For example, a billion hours of video material is viewed on YouTube each day [1] and the Amazon Marketplace lists over 350 million products [2]. But not only in media and retail large amounts of data are processed, also in the automotive

[1]Source: `https://www.youtube.com/intl/de/about/press/` (accessed April 24, 2021)
[2]Source: `https://www.nchannel.com/blog/amazon-statistics/` (accessed April 24, 2021)

sector the availability of data has heavily increased due to the use of on-board-diagnostic (OBD) systems [12].

Machine learning algorithms can be further classified by the task that they are performing, for example classification, regression, anomaly detection, missing value predictions and many more. Another common way of classification is to categorize by the kind of data that is used. The two main classes of algorithms therefore are:

- Unsupervised Learning
- Supervised Learning.

In unsupervised learning, according to Murphy [11, p. 2], the goal is to discover some kind of structure or pattern in a given data-set $\mathcal{D} = \{\boldsymbol{x}_i\}_{i=1\ldots N}$ consisting of N data-points $\boldsymbol{x}_i$. Therefore, in contrast to supervised learning, only the data points are given to the algorithms without any additional inputs in the form of labels.

In supervised learning, also according to Murphy [11, p. 2], the goal is to learn a (non-linear) mapping from inputs $\boldsymbol{x}$ to some target values y, also called labels. All input-target data pairs used for training are called the training-set $\mathcal{D} = \{(\boldsymbol{x}_i, t_i)\}_{i=1\ldots N}$. Each training input $\boldsymbol{x}_i$ can be a multi-dimensional vector representing for example an image, a sentence or a time series. In this case they are called input features. The targets or labels can either be a categorical variable $t_i \in \{1, \ldots, C\}$ (such as cat or dog) or a a real-valued scalar $t_i \in \mathbb{R}$. When t_i is categorical, the problem is known as classification or pattern recognition. Classification algorithms try to fit the data in a way that, based on the values of the input features, the desired class label can be predicted as the output. When t_i is real-valued, the problem is known as regression. The main purpose of regression algorithms is to produce an accurate real-valued estimate based on the given input features.

Although it seems as if the distinction between supervised and unsupervised learning is sufficient, there is another class of algorithms that do not work with a fixed data-set. Instead, they interact with an environment in a closed feedback loop. This area of machine learning is called

- Reinforcement Learning.

In the following sections the basics of supervised learning and reinforcement learning are described in some more detail, so that in the following chapter the deep reinforcement learning algorithms can be better understood.

2.1.1 Supervised Learning

Due to its successes in computer-vision and speech-recognition as well as in many other domains, supervised learning, especially deep learning, is probably the most commonly used form of machine learning today. As already described, the goal is to learn a generalized model of some (non-linear) input-target mapping. An application, for instance, could be to correctly label images into some category, such as cats or a dogs.

In [13], an exemplary training step of a supervised learning algorithm is described. The algorithm takes the inputs $\boldsymbol{x}_i$ and computes outputs y_i depending on the task. In case of a regression task, one output could be a vector of real numbers. In case of a classification, the output will be a vector of probabilities, one for each category in the task. After training, new images could then be classified by the probability that the algorithm computes for each class.
At the beginning of the training procedure the classification is probably wrong. Therefore, the error between the output and the desired labeling can be computed via an objective function $J = J(\boldsymbol{x}_i, t_i)$. The used objective function varies between different tasks and algorithms. One common approach for regression tasks is the mean squared error $\sum_{i=1}^{N} ||y_i - t_i||^2$ between outputs y_i and correct labels t_i. Using this error, the adjustable parameters of the model can then be modified to improve the accuracy of the model. The parameters $\boldsymbol{\theta}$ of the model, often also called weights $\boldsymbol{w}$, are real numbers that define the input–output relation of the model. In a modern deep-learning system, there may be hundreds of millions of adjustable weights and labeled examples to train with [13].

In most cases, the adjustment of the parameters is done via an error gradient. This gradient indicates the amount of error that would follow from a tiny change of the parameters. If the parameter vector is modified into the opposite direction of the gradient vector, then this is know as gradient descent. In practice, a method called stochastic gradient descent (SGD) is widely used. It works by taking only a few examples of the input vector for the error calculation and then computing the average gradient over those examples. By using many small sets of examples from the training set, a noisy estimate of the real gradient is taken, therefore the process is called stochastic. With this technique, a good set of parameters can be found surprisingly quickly when compared with more complicated optimization techniques.

After training, a test set is used to measure the performance of the system on a different set of examples. This is done to test the ability of the model to generalize, which means to produce reasonable answers on new inputs that were never seen during training.

Deep Learning A main challenge and also source of error when using classical machine-learning methods is the choice of the used input features. Often, they have to be chosen very carefully by hand because of their high influence on the performance of the system. It requires profound domain expertise to be able to design a feature extractor that transforms raw data into a useful representation or feature vector, such that the learning system can detect or classify the desired patterns.
According to [13], the main advantage of deep learning is that these features are not designed by human engineers but instead are learned automatically from the data using a

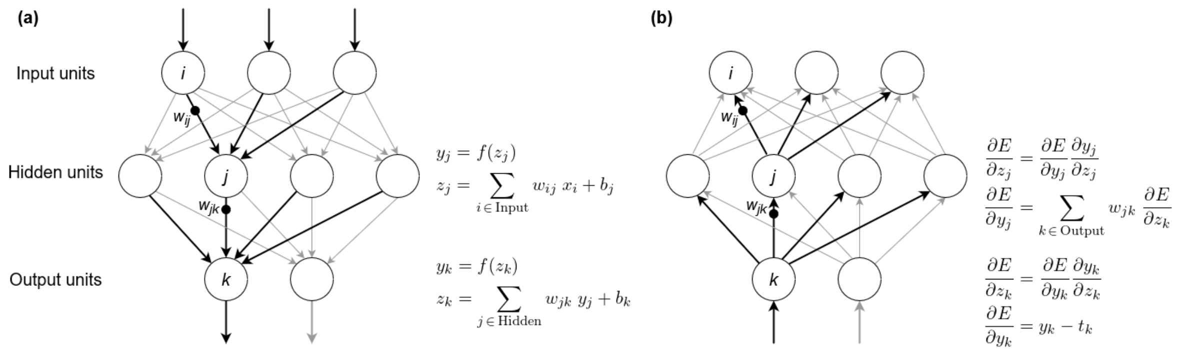
(a)
Input units
Hidden units
Output units
i
j
k
w_{ij}
w_{jk}
$y_j = f(z_j)$
$z_j = \sum_{i \in \text{Input}} w_{ij} x_i + b_j$
$y_k = f(z_k)$
$z_k = \sum_{j \in \text{Hidden}} w_{jk} y_j + b_k$
(b)
$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}$
$\frac{\partial E}{\partial y_j} = \sum_{k \in \text{Output}} w_{jk} \frac{\partial E}{\partial z_k}$
$\frac{\partial E}{\partial z_k} = \frac{\partial E}{\partial y_k}\frac{\partial y_k}{\partial z_k}$
$\frac{\partial E}{\partial y_k} = y_k - t_k$

Figure 2.1: Example of computations in a multi-layer neural network with backpropagation. Simplified version of figure found in [13].

general-purpose learning procedure. Deep-learning methods are so called representation-learning methods with multiple levels of representation. They consist of simple non-linear modules that each transform their input to a slightly more abstract level. If several such transformations are applied, very complex functions can be learned.

Murphy describes in [11, p. 995] that the idea of using deep models with multiple levels of representation actually comes from observations of the brains of humans and other mammals. There, also several levels of processing can be seen and it is believed that each level is learning features or representations at increasing levels of abstraction. In the standard model of the visual cortex for instance it is described that the brain first detects edges, then corners and contours, then surfaces and object parts, then whole objects, and so on.

Neural Networks Most deep learning applications today use feed-forward neural network architectures. A simple example can be seen in Figure 2.1. A typical network consists of a fixed-size input layer with as many neurons as the input vector (for example pixels of an image) and a fixed-size output layer (for example as many neurons as categories in a classification task). In between input and output layer there are so called hidden layers with an arbitrary number of neurons. They can be seen as a distortion of the input in a non-linear way.

Figure 2.1 is a simplified example taken from [13]. Part (a) shows a forward-pass of an exemplary network. Each neuron in the network computes a weighted sum of their inputs from the previous layer and passes the result through a non-linear function. Therefore, these networks are called feed-forward. The most applied non-linear function today is the rectified linear unit (ReLU), which is a simple half-wave rectifier $f(z) = \max(z, 0)$. Other commonly used non-linearities in the past, such as the hyperbolic tangent $f(z) = tanh(z)$ and the sigmoid function $f(z) = 1/(1+exp(-z))$, were smoother functions but the ReLU has proven to typically learn much faster in networks with many layers [13].
A backward-pass of the same example network is shown in Figure 2.1 (b). The error derivatives are calculated and propagated back through the network, so the equations

should be read from the bottom up. Calculating the error derivative at the output layer means differentiating the objective function. In this example, it is assumed that the objective function for output k is $0.5(y_k - t_k)^2$, where t_k is the target value. This gives $y_k - t_k$ as the error gradient. Through the chain rule for derivatives the error with respect to the output of a neuron $\partial E/\partial z_k$ is then converted into the error with respect to the input $\partial E/\partial y_k$ by multiplying with the gradient of $f(z)$. At the next layer, the error with respect to the output of each neuron $\partial E/\partial z_j$ is calculated by a weighted sum of the error derivatives with respect to the total inputs to the units in the layer below. Then again, applying the chain rule for derivatives gives the error derivative with respect to the input of the neuron $\partial E/\partial y_j$ and so on. Through this backpropagation of the error gradients, the network weights can be adjusted and the network is trained. Using stochastic gradient descent, it turns out that the training of multi-layer architectures can be done relatively easy.

2.1.2 Reinforcement Learning

This section describes the key principles and basic terminology of Reinforcement Learning (RL) so that the methodology and algorithms that are being used in the practical part of this thesis can be better understood.

In contrast to the machine learning techniques seen so far, RL does not work with a fixed data-set. Instead, it is characterized by a continuous interaction of an agent with its environment.
The agent in this context is a learning and decision making entity, another term for it could be controller. The environment is a representation of the world that the agent lives in and interacts with, another term could be controlled system or plant.
Sutton and Barto describe the interaction between agent and environment in [14, pp. 1–4]. In each step, the agents sees a (possibly partial) observation of the state of the environment and it can influence the environment by a set of actions. The ultimate goal of an agent is to find an optimal mapping between the observations that it receives and the actions that it produces, such that a numerical reward signal is maximized. The agent is thereby not told which actions to take, but instead must discover which actions give the most reward by trying them. In some cases, the actions might also affect subsequent rewards. This trial-and-error search and the delayed rewards are the two most important distinguishing features of reinforcement learning [14, p. 4].

Formally, the problem setting can be described as a finite Markov Decision Process (MDP). It is the mathematically idealized form of the RL problem. This formalism here comes from [14, pp. 47–48]. There, Sutton and Barto look at the interaction between an agent and its environment in discrete time steps, $t = 0, 1, 2, 3, \ldots$. At each step t, the agent receives a depiction of the environments state $S_t \in \mathcal{S}$ and selects an according action $A_t \in \mathcal{A}$. In the next time step, as an effect of its action, the agent receives a numerical reward $R_{t+1} \in \mathcal{R}$ and a new state depiction S_{t+1}.

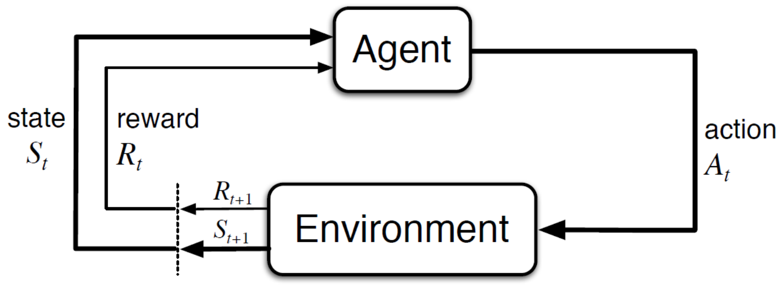

Figure 2.2: The agent–environment interaction in a Markov decision process. Figure taken out of [14, p. 48]

Due to the fact that the set of states, actions and rewards $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ are finite, R_t and S_t can be defined as random variables with discrete probability distributions dependent only on the previous state and action. This means, that for the random variables $s' \in \mathcal{S}$ and $r \in \mathcal{R}$, a probability of those values exist at time t, given the values of the previous state and action:

$$p(s', r|s, a) = Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{2.1}$$

for all $s', s \in \mathcal{S}$, $r \in \mathcal{R}$, and $a \in \mathcal{A}(s)$. This function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$, an ordinary deterministic function of four arguments, is called the dynamics of the system [14, p. 48].

Figure 2.2 shows a graphical representation of the agent-environment interaction. It reassembles a closed-loop control system with exception of the additional scalar reward signal. In fact, from a control perspective it can be shown that reinforcement learning can be seen as direct adaptive optimal control, as it was done in a paper from Sutton, Barto and Williams [15].

States and Observations The state s of an environment is a full description of the current situation that the system is in at a given moment. An observation o is a partial description of this state and some information might be missing. If an agent is able to observe the complete state of the environment, this environment is called fully observable. If the agent only can see a partial observation, the environment is called partially observable.

Almost all states and observations can be represented by some real-valued number, vector, matrix or higher order tensor. The observation about a systems state can be either continuous or discrete in time. A continuous representation better fits most real-world processes but in order to process it with a computational unit, it has to be in discrete time. Therefore, RL theory generally only focuses on discrete time states and observations. It should be noted that in most literature on RL the notation sometimes wrongly puts the symbol for state, s, where it should be the symbol for observation, o. For instance, the action of an agent is often conditioned on the state of the environment, when in reality, the agent does not have access to the state but only to its current observation of the state. However, this thesis tries to follow standard conventions for notation.

Action spaces Depending on the concrete environment, different actions might be possible for the agent to take. The set of all valid actions in a given environment is often called the action space.

In contrast to states and observations, where only discrete values are allowed, RL theory differentiates actions into discrete action spaces, where only a finite number of moves are allowed and continuous action spaces, where actions can be real-valued vectors. Some families of RL-algorithms can only be directly applied to either discrete or continuous action spaces. For application on the other case these algorithms would often have to be substantially reworked.

Policies A policy is a set of rules used by an agent to determine what actions to take. One could also think of it as a mapping from the perceived states of the environment to the actions to be taken when in those states [14, p. 6].

There are deterministic policies, which are normally denoted as μ, i.e.,

$$a_t = \mu(s_t) \tag{2.2}$$

and there are stochastic policies, formally denoted by π,

$$a_t \sim \pi(\cdot|s_t). \tag{2.3}$$

The notation in (2.3) expresses that, in the stochastic case, the policy does not directly give back the action to take but instead outputs the probability of an agent selecting a certain action in a given state.

In case the state and action spaces of a problem are multi-dimensional or so large that it gets computationally challenging to work with them, function approximation methods are used to obtain a generalized policy using a parameterized form. The parameters of such a policy are often denoted by $\boldsymbol{\theta}$ or $\boldsymbol{\phi}$, and then written as a subscript on the policy symbol to highlight the connection:

$$a_t = \mu_{\boldsymbol{\theta}}(s_t) \tag{2.4}$$
$$a_t \sim \pi_{\boldsymbol{\theta}}(\cdot|s_t). \tag{2.5}$$

Reward and Return The main goal of a RL agent is to take actions that maximize the total amount of reward that it receives from its environment. Therefore, the reward is a way to communicate to the agent what should be achieved. It should be noted that the reward should only tell the agent what to achieve and not how to do it, this is a common source of error in the application of RL.

More precisely, the agent seeks to maximize the *expected return*, where the return, denoted G_t, is a specific function of sequence of rewards R_t received. In the simplest case the return is the sum of the rewards, i.e.,

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \ldots + R_T, \tag{2.6}$$

where T is the last time step [14, p. 54].

Another kind of return is the infinite-horizon discounted return,

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{2.7}$$

where γ is a parameter, $0 \leq \gamma \leq 1$, called the discount rate [14, p. 55].

The intuition behind this discount parameter is, that rewards received earlier will be worth more than rewards received later. This assumption also makes sense in real life, for instance 1 € received now is worth more than 1 € received several years in the future because of inflation. Mathematically it also makes sense to discount future rewards since an infinite-horizon sum of rewards may not converge to a finite value without a discount factor.

Value Functions For an agent, it is often useful to know the value of a state or state-action pair, to be able to find an optimal action. With the term "value", the expected return is meant that comes from starting in this particular state or state-action pair and then following a particular policy forever after. Value functions are used in some way in almost every RL-algorithm.

The *state-value function* v_π for policy π and state s can be defined as in [14, p. 58],

$$v_\pi(s) = \mathbb{E}_\pi\left[G_t \mid S_t = s\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right] \text{, for all } s \in \mathcal{S}. \tag{2.8}$$

Similarly, in [14, p. 58] the *action-value function* q_π for state s and action a following policy π is defined as

$$q_\pi(s,a) = \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]. \tag{2.9}$$

Value functions and rewards do have some similarities. However, a reward tells what is "good" is an immediate sense while a value function tells what is "good" in the long run.

To know how "good" an action is in a relative sense, which means how much better it is than others on average, the so called advantage is computed. The *advantage function* $adv_\pi(s,a)$ of a policy π describes how much better it is to take the action a in state s, over a randomly selected action according to $\pi(\cdot|s)$. Formally, it is defined as

$$adv_\pi(s,a) = q_\pi(s,a) - v_\pi(s)\,. \tag{2.10}$$

Optimal Value and Policy The *optimal policy* π_* in [14, p. 62] is simply defined as the policy producing the maximum return. Because value functions compute an estimate of the expected return, they can be used for an ordering over policies. According to [14, p. 63], there might be more than one optimal policy but they share the same *optimal value function*,

$$v_*(s) = \max_\pi v_\pi(s) \text{ , for all } s \in \mathcal{S}, \tag{2.11}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a) \text{ , for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}(s)\,. \tag{2.12}$$

Bellman Equations To compute value functions, the *Bellman equation* is commonly applied. It works by relating the value of the current state with the value of future states. In [14, p. 59], the Bellman equation for the state-value function,

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi\left[G_t \mid S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r \mid s,a)\left[r + \gamma \mathbb{E}_\pi\left[G_{t+1} \mid S_{t+1} = s'\right]\right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s',r \mid s,a)\left[r + \gamma v_\pi(s')\right] \text{ , for all } s \in \mathcal{S}\,, \end{aligned} \tag{2.13}$$

is defined. The last line of (2.13) can again be read as an expected value. For all three variables a, s' and r, their probability is computed, weighted by the quantity in brackets and then summed up over all possibilities. This recursive relationship is a fundamental property of value functions and the basis of many RL algorithms [14, p. 59].

2.2 RL Algorithms

After the basics of machine- and reinforcement learning have been discussed in the previous section, this section should give an overview of some of the main model-free reinforcement learning algorithms. Of course this list is not exhaustive but instead the

algorithms used in this thesis (and the ones they are based on) are described.
The overview starts with some simple algorithms that work as long as the state and action spaces of the problem are small enough so that approximate value functions can be stored in arrays or tables. These algorithms are called *tabular solution methods.* Afterwards, algorithms are discussed that use function approximation to obtain a parameterized policy to handle problems with larger state and action spaces. As it turns out, feed-forward neural networks are very good function approximators and the usage of multi-layer neural networks in reinforcement learning opened a new field called *Deep Reinforcement Learning* (Deep RL).

2.2.1 Tabular Solution Methods

Dynamic Programming Although the aim of this section is to discuss model-free algorithms, lets assume for a moment that the dynamics of the system $p(s', r \mid s, a)$ are fully known. In this theoretical setting, the Bellman equation (2.13), can be used to iteratively evaluate the state-value function v_π and improve the policy π. This is known as *Dynamic Programming* (DP).

In [14, pp. 74–78], the process of dynamic programming is described in more detail. Computing the state-value function is called *Policy Evaluation* and it directly applies the bellman equation to compute a new policy estimate based on the current one. This new value function tells us how good it is to follow the current policy from state s. To know how good it is to follow a different action $a \neq \pi(s)$ can be computed by applying the bellman equation to the action-value function q_π. If this action-value is greater then the state-value from before, it means it is better to always choose a in state s than it would be to follow π all the time. Consequently, the policy can be improved in this state. Unsurprisingly, this process is called *Policy Improvement* and is a special case of a general result called the *policy improvement theorem* (see [14, p. 75] for the full proof).

Following these two processes iteratively, better policies can be found by computing better value functions.

$$\pi_0 \xrightarrow{\text{evaluation}} v_{\pi_0} \xrightarrow{\text{improve}} \pi_1 \xrightarrow{\text{evaluation}} v_{\pi_1} \xrightarrow{\text{improve}} \dots \xrightarrow{\text{improve}} \pi_* \xrightarrow{\text{evaluation}} v_*$$

This sequence must finally converge to an optimal policy and optimal value function in a finite number of iterations [14, p. 80].

Monte-Carlo Methods A simple idea to learn about the expected return without modeling the environmental dynamics is to use experience from several episodes and compute the mean over the observations. This approach of approximating the expected return is called *Monte-Carlo* (MC) methods. It requires no prior knowledge of the environments dynamics, but the sampled episodes $S_1, A_1, R_2, \dots, S_T$ must be complete.

A Monte Carlo estimation of the state-value function $v_\pi(s)$ is described in [14, p. 97]. The state s here is always the starting node. Several episodes are rolled out from this state until their termination state, the return $G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$ is computed for each episode and the mean of all returns is the desired estimate. If the number of episodes is sufficiently large, this estimate converges towards the true expected value.

To estimate state-action values $q_\pi(s, a)$, basically the same procedure is followed. In this case however, visits to the state–action pair are averaged instead of pure state visits. According to [14, p. 97], a state–action pair (s, a) is said to be visited in an episode if the state s is visited and action a is taken in it.

In order to use this method to approximate an optimal policy, the same policy iteration pattern is used as in dynamic programming. Policy evaluation is done as described so far. Many episodes are experienced, with the approximate action-value function approaching the true function asymptotically. Policy improvement is done by making the policy *greedy*. By definition in [14, p. 97], this means that the policy deterministically chooses the action with the maximal action-value,

$$\pi(s) = \arg\max_a q(s, a) \,. \tag{2.14}$$

Temporal Difference Learning The combination of Monte Carlo (MC) and dynamic programming (DP) ideas is called Temporal-Difference (TD) learning. It is model-free and learns from episodes of experience, like MC, but is also able to update estimates partially without waiting for the final outcome, like DP. This partial estimation is also called "bootstrapping".

The key idea is described in [14, p. 120]. An estimate V of the state-value function v_π is computed and updated towards an estimated return $R_{t+1} + \gamma V(S_{t+1})$, called "TD target".

$$\begin{aligned} V(S_t) &\leftarrow (1-\alpha)V(S_t) + \alpha G_t \\ V(S_t) &\leftarrow V(S_t) + \alpha(G_t - V(S_t)) \\ V(S_t) &\leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \end{aligned} \tag{2.15}$$

The extent of the update is controlled by the hyper-parameter α, called learning rate. Similarly, the action-value can be estimated. According to [14, p. 129] it can be computed as, i.e.,

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)) \,. \tag{2.16}$$

"SARSA": On-policy TD control When using the temporal difference approach discussed above to directly learn an optimal policy, this is called on-policy TD control. The corresponding algorithm is called "SARSA" with reference to the sequence of states, actions and rewards used in (2.16) that make up a transition from one state–action pair to the next and is used in the update of the Q-value [14, p. 129].

The idea to build an *on-policy* control algorithm with this estimation comes from [14, p. 129] and is straightforward. The same *policy iteration* pattern is followed again that was already discussed for DP and MC. First, an initial Q-table $Q(s, a)$, for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, is set up arbitrarily. Except for the final state, in which $Q(\text{terminal}, \cdot) = 0$. For each episode of learning, an initial action a is chosen from the policy π, based on the Q-table, greedily (2.14). Then, for each step of the episode the action a is taken, the reward r and next state s' is observed and the next action a' is chosen from the policy. Afterwards, with the full set of (s, a, r, s', a'), a new Q-Value for the state-action pair (s, a) is estimated according to (2.16) and stored in the Q-table. Finally, the state s' becomes the new state s and the action a' the next chosen action a and the process repeats until the end of the episode.
If all state–action pairs are visited for an infinite number of times, the algorithm converges to an optimal policy and action-value function with probability 1 [14, p. 129].

Q-Learning: Off-policy TD control Another way to update the Q-Value in temporal difference learning was proposed in [16] and is known as *Q-Learning*. The main difference to the SARSA algorithm from before is that the learned action-value function Q is updated independently of the policy being followed. The Q-Learning update rule in [14, p. 131] is formulated as

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)) . \tag{2.17}$$

As it can be seen in (2.17), the "TD target" is not built using the next state and action from the current policy, but instead by maximizing over all possible actions in the next state. Apart from that, the algorithm works exactly like SARSA.

The used policy still has an effect in the algorithm. It determines which state–action pairs are visited and updated. If all pairs continue to be updated, this algorithm has the advantage that the learned action-value function Q directly approximates the optimal action-value function q_* [14, p. 131].

Exploration-Exploitation Dilemma In all the algorithms discussed so far, a central problem of reinforcement learning has been concealed so far. It is known as the *exploration-exploitation dilemma* and describes the problem that without exploring all possible states of a RL-problem, it is not possible to fully get to know the environment. The algorithms seen so far rely on exploration because they only converge to the optimal policy π_* and action-value function q_* if all state-action pairs are regularly visited. On the other hand, following the policy iteration pattern, they greedily take advantage of the information of the states already visited, discouraging further exploration. This is called exploitation.

There are different ways to balance between exploration and exploitation. In MC methods, Q-learning and many on-policy algorithms, the exploration is commonly implemented by following a ϵ-greedy policy. It takes the best action most of the time, but with a probability of ϵ, a non-optimal random action is taken, thus encouraging exploration occasionally.

2.2.2 Deep Reinforcement Learning

A problem with the tabular solution methods seen so far is that theoretically the Q-Table $Q(s, a)$ can be memorized for all state-action pairs, but in practice if the state and action spaces are large, it quickly becomes computationally infeasible to work with. Therefore, function approximation (for instance machine learning models) are used to obtain approximated Q-values. If a function with parameters $\boldsymbol{\theta}$ (for example the weights and biases of a neural network) are used to calculate Q-values, the resulting action-value function can be written as $Q(s, a; \boldsymbol{\theta})$.

Deep Q-Network (DQN) When combining the Q-learning algorithm with a nonlinear action-value function approximation it may suffer from instability and divergence issues. Mnih et al. [17] [8] designed Deep Q-Networks (DQNs) to circumvent these problems by introducing two mechanisms to stabilize the training procedure:

- **Experience Replay**: Tuples of experienced episode steps $e_t = (S_t, A_t, R_t, S_t + 1)$ are stored in a so called replay memory $D_t = \{e_1, \ldots, e_t\}$ until a sufficient amount of experience was collected. Then, during the Q-learning update step, samples of experience are drawn randomly from the replay memory to train a neural network. The random sampling of episode steps removes correlations in the observation sequences and improves the data efficiency by re-using experienced steps. After some time if the replay memory is full, new experience tuples replace the oldest ones in the memory.
- **Target Networks**: Instead of optimizing the Q-values against constantly changing targets, as it is the case in the Q-learning algorithm, a second so called target network is kept to obtain the TD-targets. This target network is updated periodically only every C steps (with C as a new hyper-parameter) by cloning the network weights from the real Q-network. During normal updates of the Q-network the target-network keeps frozen. This modification makes the training more stable as it overcomes short-term oscillations.

Policy Gradient Methods In the algorithms described above (SARSA, Q-Learning, DQN), the goal was always to learn the action-value function of a problem and then select actions accordingly with a greedy policy, as seen in (2.14). The difference in policy gradient methods is, that they aim to learn a *parameterized policy* $\pi(a|s; \boldsymbol{\theta})$ directly without the detour via a value function. According to [14, p. 321], the policy is retrieved by gradient ascent toward the direction of the gradient of a objective function $J(\boldsymbol{\theta})$ with respect to the policy parameters, i.e.,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})\,. \tag{2.18}$$

The form of the objective function $J(\boldsymbol{\theta})$ is defined differently in many algorithms and also in [14] no common form is given, but a distinction between an episodic and continuing case is made. However, the idea is always to define it as an expected value of the return given the current policy $J(\boldsymbol{\theta}) = \mathbb{E}[G_t|\pi_{\boldsymbol{\theta}}]$ and then try to find the best parameters $\boldsymbol{\theta}$ for the policy $\pi(a|s;\boldsymbol{\theta})$ to produce the highest return. Due to the objective function being maximized it is sometimes also called performance function.

An advantage in parameterizing the policy instead of the action-value function (like in DQNs) may be achieved if the policy is a simpler function to approximate [14, p. 323]. However, this depends entirely on the problem. For some problems, it may be simpler to learn the action-value function and therefore it makes more sense to approximate it instead of the policy. In another problem setting, it may be the other way round and its makes more sense to approximate the policy directly.

Policy Gradient Theorem The tricky part in policy gradient methods is to determine the gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. The problem is described in [14, pp. 324–326]. Simply put, no matter the choice of $J(\boldsymbol{\theta})$, the gradient always depends on both the selected actions (directly determined by $\pi(a|s;\boldsymbol{\theta})$) and the distribution of states in which those selections are made. Unfortunately, the effect of the policy on the state distribution is a function of the environment, which is generally unknown. Fortunately, the *policy gradient theorem* provides a nice reformulation of the derivative of the objective function to not include the derivative of the state distribution anymore. The theorem and its proof can be found in [14, p. 325]. This simplifies the gradient computation $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ a lot.

When using the policy gradient theorem to reformulate the gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, an update-rule for the parameters $\boldsymbol{\theta}$ can be formulated as:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha\; G_t\; \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t;\boldsymbol{\theta}_t)}{\pi(a|s;\boldsymbol{\theta})}\,. \tag{2.19}$$

This update-rule is the basis of the *REINFORCE* algorithm [18] and also makes sense intuitively. Each update consists of the product of the return of an episode G_t and a vector which is the gradient of the probability of taking an action divided by the probability of taking that action [14, p. 327]. This makes sense because the gradient corresponds to the direction in parameter space of the probability to take the action A_t. Multiplying this vector with the return G_t causes the parameters to move in a direction with actions of high return. Dividing by the action probability also makes sense because it causes a normalization of the frequency with which actions are taken.

To reduce the variance of the policy parameter update and thus increase the performance of the algorithm, the policy gradient theorem can be further generalized to include a

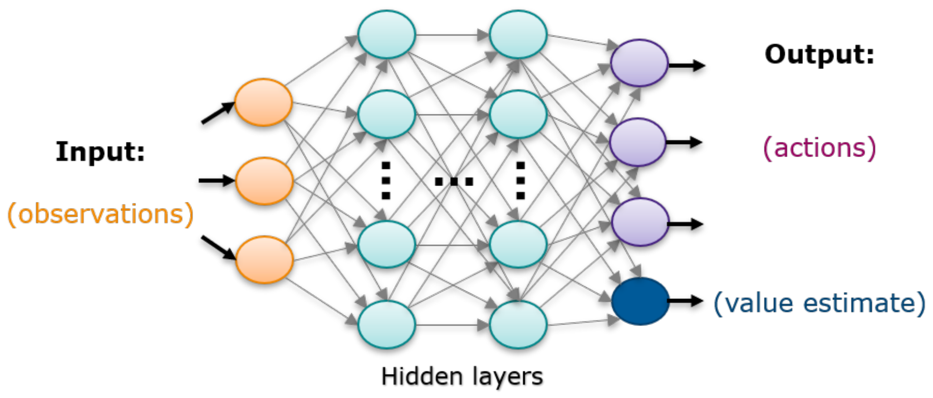

Figure 2.3: Example of the use of a neural network for a actor-critic policy.

comparison of the return to an arbitrary baseline function $b(s)$. According to [14, p. 329] it can then be written as:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \left(G_t - b(S_t) \right) \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t | S_t; \boldsymbol{\theta}_t)}{\pi(a|s;\boldsymbol{\theta})} \,. \tag{2.20}$$

A common choice for the baseline is an estimate of a value function, $V(s, \boldsymbol{w})$ or $Q(s, a, \boldsymbol{w})$, where $\boldsymbol{w}$ is an additionally learned weight vector.

Actor-Critic Methods In case also a value function is learned in addition to the policy, these methods are often called *actor–critic methods*.

- **Actor**: Updates the policy parameters $\boldsymbol{\theta}$, in the direction suggested by the critic.
- **Critic**: Updates value function parameters $\boldsymbol{w}$. Depending on the algorithm it could be action-value $Q(a|s;\boldsymbol{w})$ or state-value $V(s;\boldsymbol{w})$ parameters.

Figure 2.3 shows an example of how a neural network could be used for function approximation in an actor-critic policy. In this case, the same network is used for the approximation of the mapping from observations coming from the environment to both the possible actions and a state-value function estimate. The action outputs and the value function output share all hidden layers of the network ($\boldsymbol{\theta}$ and $\boldsymbol{w}$ are the same except for the weights and biases of the output layer). However, there is also the possibility that they only share some hidden layers or that two entirely different networks are used for the approximation of actions and value estimate.

Deterministic Policy Gradient (DPG) All methods and algorithms discussed so far work with stochastic policies. However, Silver et al. [19] proposed a Deterministic Policy Gradient (DPG) algorithm to estimate the policy gradient more efficiently. The method works for RL problems with continuous action spaces and is an off-policy actor-critic algorithm.

According to [19], the deterministic policy gradient is the expected gradient of the action-value function. Therefore, they formulated the *Deterministic Policy Gradient Theorem*, which can be seen as special case of the stochastic one. The gradient is computed by integration over the state space while in the stochastic case, the policy gradient integrates over both state and action spaces. Therefore, the deterministic policy gradient can be estimated more efficiently than its stochastic counterpart. The authors also showed in empirical results that their algorithm outperforms stochastic policy gradient on several problems, including a high-dimensional bandit, standard RL benchmark tasks of a mountain car and a pendulum as well as controlling an octopus arm with a high-dimensional action space. However, the function approximation in the experiments was done with tile-coding (see [14, p. 217]) and linear function approximators.

Deep Deterministic Policy Gradient (DDPG) By extending the deterministic policy gradient approach and combining it with methods from deep-q-networks, Lillicrap et al. [20] proposed the Deep Deterministic Policy Gradient (DDPG) algorithm. It was developed to be a actor-critic, off-policy, model-free algorithm, which can be used in continuous environments and with continuous action spaces.

The algorithm is explained in [20] and works by using the actor-critic setup from DPG, so that the policy can be estimated directly due to the deterministic policy $\mu_{\boldsymbol{\theta}}$ (proofed by the deterministic policy gradient theorem in [19]). In general, both actor and critic are approximated by neural networks. The learned value function is an action-value function $Q(s, a)$, which acts as a critic. However, the learning process is stabilized by using experience replay and an idea similar to a target network, called "soft target". Instead of copying the weights directly as in DQN, the soft target network updates its weights $\boldsymbol{\theta}'$ slowly to track the learned networks weights $\boldsymbol{\theta}$ according to $\boldsymbol{\theta}' \leftarrow \tau\boldsymbol{\theta} + (1-\tau)\boldsymbol{\theta}'$ with $\tau \ll 1$. This way, the target network values are constrained to change slowly. Additionally, in order to have better exploration, an exploration policy μ' was introduced by adding noise sampled from a noise process to the actor policy: $\mu'(s) = \mu_{\boldsymbol{\theta}}(s) + \mathcal{N}$. To proof the effectiveness of the algorithm, more than 20 simulated physics tasks of varying difficulty were solved with the same learning algorithm, network architecture and hyper-parameters. The obtained policies with DDPG were comparable with those of a planning algorithm with full access to the underlying physical model and could be found with 20 times fewer steps of experience than DQN. The paper also contains links to videos for illustration.

Trust Region Policy Optimization (TRPO) In traditional policy gradient algorithm updates, old and new policies are kept close in parameter space. But even a small difference in the parameters can lead to large differences in the performance, which means that a few "bad" updates can ruin the policy gradient sample efficiency. Due to this, it is dangerous to use large step sizes with the policy gradient algorithms discussed so far.

To avoid this problem, Schulman et al. [21] introduced an algorithm with an iterative procedure to monotonically improve performance by taking the largest step possible, while satisfying a special constraint on how close the new and old policies are allowed to be. The constraint is formulated in terms of a Kullback–Leibler (KL) divergence [22], D_{KL}, which simply put is a measure of a distance between two probability distributions. TRPO aims to maximize the objective function $J(\boldsymbol{\theta})$ subject to a *trust region* constraint which enforces the distance between old and new policies to be within a small parameter δ. According to [21], this can be written in terms of expectations:

$$\mathbb{E}_{s\sim\rho_{\boldsymbol{\theta}_{\mathrm{old}}}}\left[D_{\mathrm{KL}}(\pi_{\boldsymbol{\theta}_{\mathrm{old}}}(\cdot|s) \parallel \pi_{\boldsymbol{\theta}}(\cdot|s)\right] \leq \delta \tag{2.21}$$

where the notation $\mathbb{E}_{s\sim\rho_{\boldsymbol{\theta}_{\mathrm{old}}}}[\ldots]$ indicates that states are sampled from the state distribution with the old parameters $\boldsymbol{\theta}_{\mathrm{old}}$. The authors also showed that policy iteration, policy gradient, and natural policy gradient (not dicussed here, see [23]) are special cases of TRPO. In experiments, the algorithm performed well on simulated robotic tasks of swimming, hopping, and walking, as well as playing Atari games in a framework directly from raw images.

Proximal Policy Optimization (PPO) An improved version of TRPO called Proximal Policy Optimization (PPO) was published by Schulman et al. [24] in 2017. It is motivated by the same question of how to take the biggest possible improvement step on a policy, without stepping “too” far and accidentally causing performance to collapse. In TRPO, the problem was solved with a complex second-order method, while PPO simplifies it by using a clipped objective function to keep the new policy close to the old one. This method is also significantly simpler to implement while achieving similar performance in experiments.

According to [24], if the probability ratio between the old and new policy is defined as:

$$r(\boldsymbol{\theta}) = \frac{\pi_{\boldsymbol{\theta}}(a|s)}{\pi_{\boldsymbol{\theta}_{\mathrm{old}}}(a|s)}, \tag{2.22}$$

then PPO imposes a constraint on the objective function by forcing $r(\boldsymbol{\theta})$ to stay within an interval around 1, specifically $[1-\epsilon, 1+\epsilon]$, where ϵ is a hyper-parameter:

$$J^{\mathrm{CLIP}}(\boldsymbol{\theta}) = \mathbb{E}\left[\min\left(r(\boldsymbol{\theta})\hat{A}_{\boldsymbol{\theta}_{\mathrm{old}}},\ \mathrm{clip}(r(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon)\hat{A}_{\boldsymbol{\theta}_{\mathrm{old}}}\right)\right] \tag{2.23}$$

The term $\hat{A}_{\boldsymbol{\theta}_{\mathrm{old}}}$ is an estimate of the advantage (see (2.10)) calculated with the old parameters. The function $\mathrm{clip}(r(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon)$ clips the probability ratio to be no more than $1+\epsilon$ and no less than $1-\epsilon$. By taking the minimum between the original value and the clipped version, the objective function of PPO therefore loses the motivation to increase the policy update too much for better rewards. Test cases showed that PPO was able to achieve much better performance compared to TRPO after training for the same time on various continuous control tasks and matching performance on playing Atari games with image inputs.

Soft Actor-Critic (SAC) In 2018, Haarnoja et al. [25] introduced a new off-policy actor-critic algorithm, called Soft Actor-Critic (SAC), with the goal to improve the sample efficiency and minimize the need for hyper-parameter tuning, so that it can be used to learn real-world robotic tasks. A major difference to the RL algorithms discussed previously, is that it is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy. This maximum entropy reinforcement learning framework was already used in a precedent algorithm called Soft Q-Learning (SQL) [26] and improved for SAC.

In this framework, according to [25], the policy is trained with the objective to maximize the expected return and the entropy at the same time:

$$J(\boldsymbol{\theta}) = E_{(s_t,a_t)\sim\rho_{\pi_{\boldsymbol{\theta}}}}\left[r(s_t, a_t) + \alpha\mathcal{H}(\pi_{\boldsymbol{\theta}}(\cdot|s_t))\right], \tag{2.24}$$

where $\mathcal{H}(\cdot)$ is the entropy measure. The trade-off is controlled by the non-negative temperature parameter α. The authors showed that this objective can be seen as an entropy constrained maximization of the expected return. Therefore, the temperature parameter can be learned automatically instead of being treated as a hyper-parameter. The goal of developing an algorithm for real-world systems was demonstrated by successful training of a set of robot tasks in only a handful of hours, demonstrating the improved sample efficiency. The algorithm was also able to outperform other popular deep RL algorithms, DDPG, TD3 and PPO on standard benchmark tasks in simulated physical environments.

Twin-Delayed Deep Deterministic (TD3) A commonly known problem in value-function-based algorithms is an overestimation of the value function. This means that the value of some states is estimated higher than it really is. The problem thereby is that the resulting error can propagate and accumulate over several iterations and negatively affects the policy. This can also happen in policy gradient methods if a value function is estimated, for instance in an actor-critic setting. In DDPG, the overestimation of Q-values may even lead to the policy breaking down because it only exploits the errors in the Q-function. To address this issue, in 2018 Fujimoto et al. [27] introduced the Twin Delayed Deep Deterministic (TD3) algorithm by applying a couple of tricks on DDPG to prevent overestimation.

In particular, in [27] three crucial tricks were introduced:

- Clipped Double-Q Learning: Instead of learning only one Q-function, in TD3 the action selection and Q-value estimation are made by two networks separately. The smaller of the two Q-values is then used to form the targets in the bellman-error loss function. This minimum leads to an underestimation bias which is harder to propagate over training iterations.
- "Delayed" Policy Updates: TD3 updates the policy less frequently than the Q-function. The paper recommends one policy update for every two Q-function

updates. A similar approach is used in the DQN where the target network is only updated periodically to obtain a stable objective.

- Target Policy Smoothing: A small amount of clipped random noise is added to the selected action, to make it harder for the policy to exploit Q-function errors. Due to the noise, the value estimation is smoothed, which makes it harder for the policy to exploit Q-function errors.

To evaluate the algorithm, its performance on a set of standard continuous control benchmark tasks was measured in simulated physical environments. TD3 was able to match or outperform all other algorithms in comparison (DDPG, TRPO, PPO, SAC and others) in both final performance and learning speed across all tasks.

3 Data-Driven Controller Design

After discussing the theory of machine learning with a focus on deep RL algorithms, in this chapter the methodology of how to build a data-driven controller with this technology is discussed. The setup and components needed to build a self-learning system are explained and an evaluation of the methodology on a small selected benchmark example is shown.

3.1 Use Cases

After some literature research for related work about the use of reinforcement learning in a process control context, and after discussion with my supervisors at the industrial partner of this thesis, two main approaches were selected for further investigation.

3.1.1 Use Case 1: Self-Learning Feedback Controller

The first use case consists of the straightforward approach of building a controller by giving the reinforcement learning agent access to all relevant actuators of the system as actions and having a feedback of the controlled process variables as an input to the controller. The learned policy of the agent can then be used as a control law after training.

This use-case is described in some more detail in [28], where Hafner and Riedmiller introduced a RL algorithm for process control that uses neural networks for approximation of a Q-function, called Neural-Fitted Q-Iteration with Continuous Actions (NFQCA). They were able to achieve good results in different feedback process control tasks using this setup, which serves as a motivation for this thesis. However, instead of the NFQCA algorithm from Hafner and Riedmiller, modern state-of-the-art policy gradient algorithms are used, that are proven to have better sample efficiencies and are more stable during the training process.

3.1.2 Use Case 2: Parameter-Learning Agent

The second use case that is investigated in this thesis is the use of RL agents to learn optimal parameters for PID controllers at each time-step. Due to their effectiveness and ease of implementation, PID controllers are widespread and also the current control concept for the diesel engine air-path controller in the model received from AVL relies on them (see Chapter 1.2). However, the search for a good set of parameters can be a time consuming procedure even for experienced commissioning engineers and therefore this use-case has evolved, where the idea is to automatically learn parameters with a RL agent. The learned policy of the agent can then be used to predict optimal gain parameters for the existing PID-controllers for each time-step.

The idea is similar to the PID-tuning method introduced by Wang et al. in [29]. There, the authors used an actor-critic RL algorithm with a Radial-Basis-Function (RBF) neural network to approximate the policy. The inputs to the agent were chosen as $x = [\epsilon, \Delta\epsilon, \Delta^2\epsilon]$, where ϵ is the error between feedback and demand value and the second and third state correspond to the first and second numerical derivative of this error. The actions are $u = [K_i, K_p, K_d]$, the integral, proportional, and derivative gain factors in the discrete PID-controller formulation. Wang et al. achieved good results with this setting on a simulated nonlinear system with parameter disturbances, which serves also a motivation for this thesis. As discussed in the first use-case, however, modern state-of-the-art policy gradient algorithms are used instead of actor-critic RBF networks.

3.2 Learning Setup

Due to the try-and-error nature of reinforcement learning, it is generally preferable to execute the learning process in a simulation setup. All model-free agents initially know nothing about the environment they interact with, hence the real physical systems could be damaged in the initial phase of learning if the agent has unrestricted access to actuators and tries actions randomly. But even if the actions are somehow constrained, the typical training time of RL agents to learn suitable policies is long. This problem is also known as *data inefficiency*. For example, DQN required days of continuous play to become skilled at Atari games [8] and AlphaGo had to be trained for tens of thousands of games to be competitive against humans [9]. Therefore, training RL agents on real hardware is often infeasible without accurate simulation models. Luckily, for this thesis an accurately calibrated simulation model of a diesel engine is provided by AVL.

Figure 3.1 shows the principle learning setup used in this thesis. For training a reinforcement learning agent, a step-wise interaction of the agent with the environment has to be provided, so that the current state and reward signals can be received and an according action can be given out. The environment consists of the existing air-path control software that is executed together with the diesel engine simulation model in

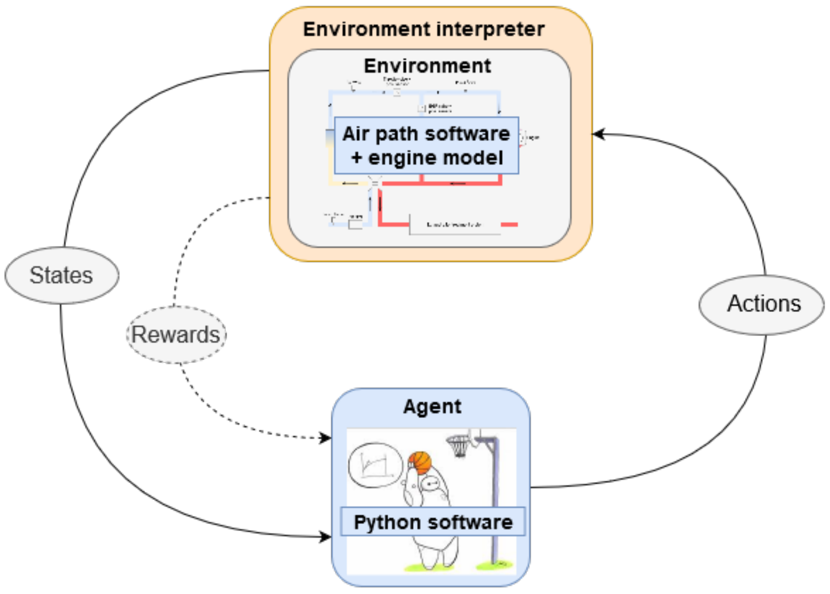

Figure 3.1: Setup to train RL agents with a simulation model through an environment interpreter.

Simulink[1]. The handling of the simulation is done by an environment interpreter that consists of a set of scripts to start and pause the simulation as well as to build the current state vector and reward signal. The training of the agent is then done in Python[2] due to the large collection of suitable libraries that are available. Because of this, it is also a current standard in RL research.

In the following sections, the setup is further described on the basis of the individual components.

3.2.1 Environment

As already described in the first chapter, the industrial partner provided an accurately calibrated simulation model of a heavy-duty diesel engine with an existing air-path control system for the training purpose of this thesis. The existing control software is described in more detail in Section 1.2. However, to enable the training process, the existing controller had to be by-passed. Therefore, a switch was implemented in Simulink to either use the control signals coming from the existing controller or to read in the signals coming from the RL agent from the Matlab[3] workspace. The selected signals are then forwarded to the diesel engine simulation model.

[1]Simulink is a proprietary graphical programming environment for the design and simulation of dynamical systems. See: `https://www.mathworks.com/products/simulink.html`
[2]Python is an open-source, general-purpose programming language. See: `https://www.python.org/`
[3]Matlab is a proprietary programming language and numeric computing environment developed by MathWorks. See: `https://www.mathworks.com/products/matlab.html`

The engine simulation model was designed by experienced engineers at AVL with the main purpose of accurately simulating the engine emissions. Consequently, the engine dynamics and the combustion process are also modeled very precisely. The accuracy of the model has been repeatedly confirmed by measurements on real engine test-beds, which means that the model sufficiently represents a real diesel engine air-path for the purposes of this thesis.

As input to the engine model, an external speed and torque demand can be applied that define the load profile of the engine. This way, either own experiments can be defined or the engine's response to standardized emission test cycles can be simulated.

3.2.2 Environment Interpreter

In order to execute the environment model in a way that is needed for RL training, it had to be prepared to be controlled by an external agent, which is in charge of the learning process. The agent needs to be able to execute one time-step of the environment and receive a current state vector and reward signal. After the learning step is done and an action is selected, the agent also needs a way to pass this action back to the environment.

For the communication between the agent, that is executed in Python, and the environment, that is executed in Simulink, the Matlab-engine-API[4] was used as an interface. It supports the start of a Matlab-engine from Python that can be used to call built-in Matlab functions and also user-scripts.

The main challenge hereby was the question of how to implement the Matlab scripts for the constant interruptions of the environment model, that are necessary to execute it step by step. Under normal conditions, Simulink models are executed continuously via the `sim()`-command from some initial state until a specified simulation time is reached. Since any initial state can be chosen and the terminal state of the simulation is known, the first approach was to use repeated calls of `sim()` while using the returned termination state as the next initial state. The provided simulation time then becomes the discretization time of the model. Although this approach works, due to the size and complexity of the model the execution becomes very slow and is therefore not useful for RL training. The main reason for this is that the `sim()`-command compiles the model on every call. This results in an unnecessary model re-compilation after every discrete time step, which is obviously unwanted.

The approach used instead uses Matlab command line functions to control the Simulink execution together with an assertion-block in Simulink that is triggered with the desired discretization time. Using the functions,

- `set_param('model_name','SimulationCommand','start')`

[4]See: `https://www.mathworks.com/help/matlab/matlab-engine-for-python.html`

- `set_param('model_name','SimulationCommand','pause')`
- `set_param('model_name','SimulationCommand','stop')`

which are equivalent to clicking the play, pause and stop buttons in Simulink respectively, the model execution behaviour can be controlled. If the pause command is placed inside a periodically triggered assertion-block, the model pauses automatically at discrete time steps and can be restarted again from a script with the start command. This approach is still a lot slower than a continuous execution of the model, but at least the unnecessary re-compiles at each time step are avoided.

Based on this approach, the following scripts for controlling the model execution were created:

- `init_airpath.m`: Initializes the Simulink engine model and air-path control system and enables the periodically triggered assertion-block.
- `start_airpath.m`: Starts the simulation.
- `wait_airpath.m`: Waits until the assertion is triggered in the simulation and builds the current state vector by reading out the corresponding values from the paused model.
- `step_airpath.m`: Resets the triggering mechanism in the model and continues the simulation again if the total simulation time has not been reached, otherwise the simulation is terminated and a "done"-variable is set to indicate this.

3.2.3 RL Agent

The training of the reinforcement learning agent is done in Python, due to the availability of useful libraries for neural networks and reinforcement learning. After comparing several available and commonly used baseline algorithm implementations, the stable-baselines package [30] was selected for large parts of the training process in this thesis. It provides implementations of a set of state-of-the-art deep RL algorithms with a simple interface to train and evaluate different agents. Among others, all algorithms described in Section 2.2.2 are available in this package. The implementations are tested, well documented and offer an unified structure for all algorithms that are based on an actor-critic setup. This is very helpful for integrating the training results back into the original Simulink model, because it means that only a single policy has to be implemented for all actor-critic algorithms.

As a back-end for training and evaluation of the neural networks, that are used in the deep RL policies, the stable-baselines package relies on the open-source Tensorflow package [31], created by Google Research. This is also an advantage for the integration of the training results back into the original model, due to the clear structure of the networks and the ease with which the network parameters of the trained neural networks can be exported.

In order to use the provided simulation model as a training environment, the environment interpreter scripts also had to be called from Python, in a way that different algorithms can work with them. For this purpose, the stable-baselines package supports the creation of custom environments as long as they follow the OpenAI Gym [32] interface. The OpenAI Gym is an open-source collection of standardized RL problems, mainly used for benchmarks and comparing new algorithms, and also provides an open interface to create new custom problems. Therefore, a class was created that inherits from the abstract Gym interface and implements a set of methods:

- `__init__(self)`: Initializes the environment by defining the action and observation spaces, starts the Matlab-engine and calls `init_airpath.m` to initialize the simulation model.
- `step(self, action)`: Takes a step in the environment by applying the given action. Therefore, the `wait_airpath.m` script is called to ensure that the Simulink model has finished the previous step. The current state vector can then be read-out from the Matlab workspace and a reward for this step is calculated. Different approaches for the reward design were tested out in Section 3.3.1. After the state and reward for a step are defined, the given action is written to the Matlab workspace, from where the Simulink model fetches it, and the simulation is continued by calling the `step_airpath.m` script. The step-method is also responsible for monitoring the end of an episode. This is done by reading out the `done`-variable from the Matlab workspace, which is set by the environment interpreter scripts if one experiment is finished. The method ends by returning the current observation, reward, done-variable and an optional info-variable.
- `reset(self)`: After the termination of one experiment, also called episode, the environment needs to be reset. This is done by resetting the corresponding Python variables and calling the `start_airpath.m` script to start a new experiment.
- `close(self)`: Closes the environment in a defined manner by terminating simulations that might be running and stops the Matlab-engine.

3.3 Setup Evaluation

In order to evaluate the learning setup presented in this chapter, an evaluation on a small exemplary benchmark problem in the same setting was done. It also shows that in principle a learning process is achievable with both introduced use-cases before moving on to the more complex and error-prone air-path simulation model.

3.3.1 Problem definition

The exemplary benchmark problem chosen for the evaluation of the setup is a relatively simple, one-dimensional non-linear problem of controlling the water-level at a desired value inside a container, for example a water-tank, with a constant outflow and a controllable inflow of water. The system equations are,

$$\begin{aligned} \dot{x} &= -\frac{a}{A}\sqrt{x} + \frac{b}{A}u, \\ y &= x, \end{aligned} \tag{3.1}$$

where the state variable $x > 0$, that is also the output y of the system, corresponds to the height of water in the tank and the input u is the voltage applied to a pump. The water enters the tank with cross-section area A at the top with a rate b proportional to the applied voltage and leaves through an opening in the tank base that is proportional with rate a to the square root of the water height. The presence of the square root results in a non-linear plant.

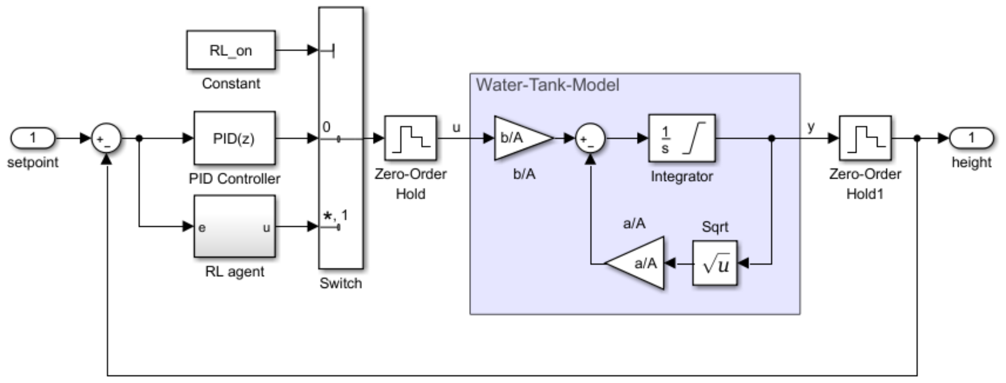

Figure 3.2: Simulink model of the benchmark problem of a water-tank control for the evaluation of the learning setup.

This choice for the problem was arbitrary, because the practical aspects are not really important for this evaluation, the only requirement was that the problem can be easily modeled in the same setting as the considered main problem of this thesis. That is, a Simulink model for the description of the problems dynamics exists and the learning process is done via the stable-baselines package in Python. Therefore, a simple simulation model in Simulink was created that can be seen in Figure 3.2. The model implements the dynamics given in (3.1) and discretizes the system with a discretization-time of $T_d = 100\,\text{ms}$. As already discussed in Section 2.1.2, RL algorithms only work with discrete observations. After the model was defined, an environment interpreter was created in the same way as described before in Section 3.2.2 and an environment class in Python was defined as described in Section 3.2.3, such that training with algorithms out of the stable-baselines package is possible.

Reward design Probably the most important design choice when training an agent via reinforcement learning is the selection or construction of a suitable reward function for the given problem. Some environments already have an intrinsic reward, for instance in [8] where an agent was trained to play classic Atari games. These games naturally give a score that can be used as a reward. In other cases, it may be necessary to carefully engineer shaped rewards in order for the agent to be able to explore high dimensional problems. Another option is to use sparse rewards as in [33], where the agent only receives a reward when the state is within a small tolerance of a goal state. With this setting, the authors where able to learn difficult robotic control tasks.

In the setting of this thesis, where the goal is to set up a feedback loop, the difference between the demanded setpoint and its corresponding variable to be controlled can be seen as a naturally given reward signal. Consequently, all reward functions that are tested here are based on this error $e(t) = y_\mathrm{r}(t) - y(t)$. However, different possibilities of how to construct a reward function from this signal are investigated. Figure 3.3 gives a visual representation of the different reward functions.

- The first reward function under investigation simply uses the negative squared error between demand value and feedback value. The square function both prevents positive rewards and penalizes higher deviations quadratically. However, compared to other reward functions, small errors are penalized less.

$$r_1(t) = -e^2(t) \tag{3.2}$$

- As a second reward function, the negative absolute error between demand value and feedback value was chosen because it again prevents positive rewards but penalizes all deviations linearly.

$$r_2(t) = -|e(t)| \tag{3.3}$$

- The third reward function uses sparse rewards in a form that a small penalty is given in every time step that the error is outside a tolerance ϵ. Additionally, a penalty is given if the error in this time step is larger than the error in the last discrete time step. This approach was used in [29], from which the second investigated use-case in this thesis originates from.

$$\begin{gathered} r_3(t) = 0.6\ r_a(t) + 0.4\ r_b(t) \\ r_a(t) = \begin{cases} 0 & |e(t)| \leq \epsilon \\ -0.5 & \text{otherwise} \end{cases},\ r_b(t) = \begin{cases} 0 & |e(t)| \leq |e(t-1)| \\ -0.5 & \text{otherwise} \end{cases} \end{gathered} \tag{3.4}$$

- The fourth and last reward function under investigation comes from [28], where also a self-learning feedback controller was trained, much like in the first use-case of this thesis. Here, a shaped reward function is used to obtain a smooth and differentiable cost function, in hope to achieve a more precise control law than with a sparse formulation,

$$r_4(t) = -0.5\ \tanh^2\left(|e(t)|\ \tanh^{-1}\left(\frac{\sqrt{0.95}}{\epsilon}\right)\right). \tag{3.5}$$

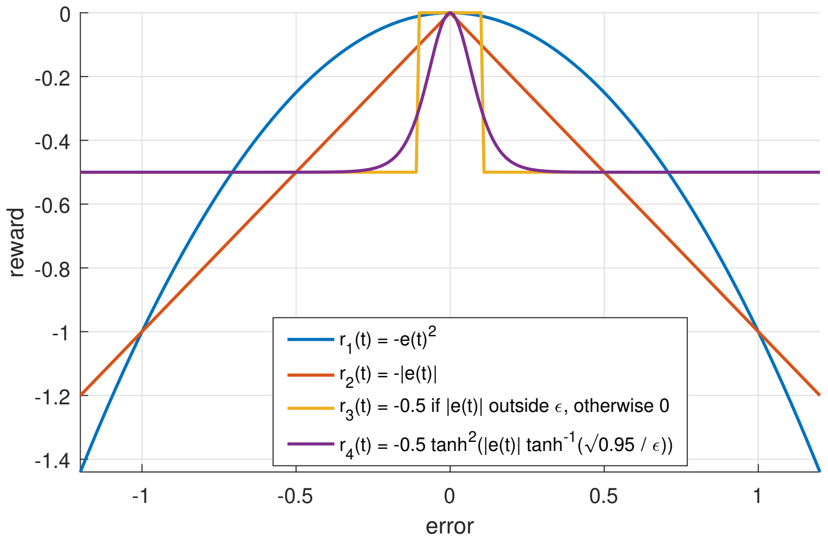

Figure 3.3: Illustration of the different reward functions that are compared on the exemplary water-tank problem. Note that for the sparse reward $r_3(t)$ only the non-time-dependant first part of the formulation in (3.4) is shown in this figure.

Input generation A point that has not been discussed so far is the definition of the inputs to the simulation model during training. In the discussed setting of this thesis, the agent has no influence on the desired reference values (in case of the air-path the lambda and boost pressure demand, in case of the benchmark problem the desired water-level) but only controls deviations from a given setpoint. Due to this fact, the choice of the reference function during the training process can have a significant influence on the resulting policies.

In both discussed use-cases, the agent always controls the deviation between desired and actual values in a feedback loop. Therefore, the learning process should contain as many different deviations as possible during training, so that the learned policy is as general and versatile as it can be. To achieve this, several step functions with varying setpoints are used as reference functions for the demand value. However, there are still multiple options for the design of episodes:

- One option would be to define an episode as a single step function with a randomly chosen setpoint. Throughout the training, the agent then experiences different deviations and could learn to react to them. However, an issue of this option is that the best possible reward that the agent could achieve differs from experiment to experiment. Small setpoint values and therefore small deviations between desired and actual signal can be compensated much quicker that larger differences and therefore naturally lead to higher rewards.

- Another option would be to define an episode as a series of multiple steps with randomly varying setpoints. This way, an averaging effect over the whole experiment occurs and the best achievable episode rewards are more evenly distributed.

Algorithms As already discussed in the description of the two use-cases that are considered in this thesis, the aim in both of them is to use modern state-of-the-art policy gradient algorithms, because they provide better sample efficiencies and are more stable during the training process, compared to older RL algorithms. In Section 2.2.2 about deep RL, several such algorithms were explained and now they are compared on this small benchmark problem.

The particular algorithms under consideration are DDPG (Deep Deterministic Policy Gradient) and some improved versions, namely PPO (Proximal Policy Optimization), TD3 (Twin-Delayed Deep Deterministic) and SAC (Soft Actor-Critic). For the evaluation, they are run with their standard hyper-parameter-sets as defined in the stable-baselines package [30]. This is because all algorithms under consideration are proven to be able to solve simple continuous control problems with their initial hyper-parameter-settings and the selected benchmark problem of a water-tank control would probably fall into this difficulty category. With an additional hyper-parameter tuning for all algorithms, better results could probably be achieved, but since the aim of this training is not to find the best control strategy possible but rather to compare different algorithms, rewards and input-generation settings, a further hyper-parameter optimization is not purposeful.

3.3.2 Evaluation results

Reward and algorithm comparison The first evaluation aims at comparing the four selected policy gradient algorithms with the four defined reward settings. Therefore, the input generation is set to the first option of training with single step functions with randomly chosen setpoints for a duration of 5 seconds. The initial state of the simulation is always an empty water-tank and training is done for 1000 episodes, so 1000 different experiments with varying setpoints are seen by the agent during one training process.

At first, training in the setting of the first use-case (see Section 3.1.1) is done, so the agent has direct control over the actuators, which for this benchmark problem is the voltage applied to the water-pump.

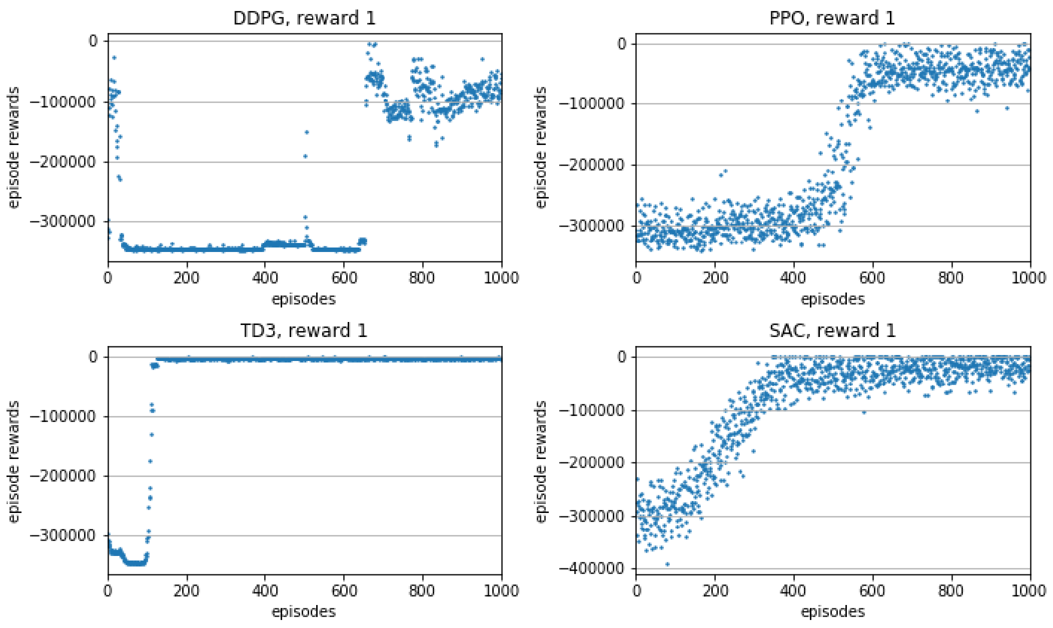

Figure 3.4: Comparison of learning curves of the four selected algorithms with reward 1 on the exemplary water-tank control problem. Training in the setting of the first use-case with single random step functions as input.

Figure 3.4 shows the learning curves of the four selected algorithms when they are trained with reward function $r_1(t)$ as in Equation (3.2). Each blue dot in a scatter plot symbolizes the accumulated reward over one episode of training. It can be seen that all four algorithms are able to improve their received reward over time, which means that the policy they learn minimizes the error between the desired and actual water-height in the tank. Also, after some time the episode rewards do not get better anymore, which means that they converged into some maximum. This does, however, not say anything about the quality of the learned policies because the maximums could only be local. The policies performances are discussed later. What can be seen from this comparison is, that the TD3 algorithm is able to learn an optimal policy the fastest, followed by the SAC algorithm. The DDPG and PPO algorithms take a few hundred episodes more training time with this reward and hyper-parameter setting.

In Figure 3.5, the learning curves in the same setting are shown but this time the algorithms are trained with reward function $r_2(t)$ as in Equation (3.3). The TD3 and SAC algorithms again are able to improve their received episode rewards over time and converge into some maximum, but this time both need a few hundred episodes of training more. This could be due to random nature of the experiments and algorithms or a effect of the reward function, since large deviations are not penalized as much as with the quadratic reward function $r_1(t)$. However, the DDPG and PPO algorithms do not reach the same episode reward level as TD3 and SAC and also do not seem to improve over time at all using this reward setting and number of training episodes.

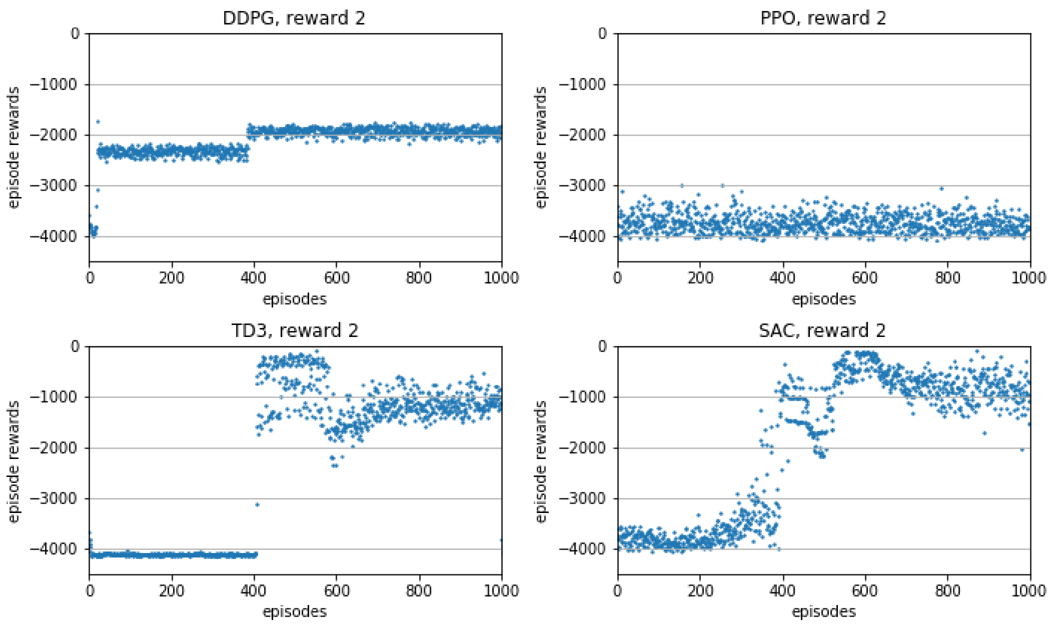

Figure 3.5: Comparison of learning curves of the four selected algorithms with reward 2 on the exemplary water-tank control problem. Training in the setting of the first use-case with single random step functions as input.

A similar learning behaviour as for the DDPG and PPO algorithm in Figure 3.5 can be seen for all algorithms with reward functions $r_3(t)$ and $r_4(t)$. These settings do not lead to a recognizable improvement of the episode rewards over time. Neither with the sparse reward setting (3.4) nor with the shaped reward of setting (3.5) the algorithms show significant improvement over the course of training with 1000 episodes. This may be due to the constant negative reward that is given outside some region ϵ around zero error, which makes it harder for the algorithms to determine a direction for improvement. The reward settings $r_1(t)$ and $r_2(t)$ both give a negative reward that is proportional (either quadratically or linearly) to the error at all times and therefore large deviations are penalized more than small ones.

The evaluation of the reward settings however is not complete without also looking at the performance of the learned policies. Therefore, Figure 3.6 shows the result of 3 experiments with the final trained agents. On the left, a trained TD3 agent and on the right a trained SAC agent in the setting of the first use case is shown, which means that they can directly apply the control action u to the tank-model. The TD3 and SAC agents were selected because they showed the most promising learning curves in Figure 3.4. The 3 experiments consist of filling the tank to 10%, 50% and 90%. In the respective upper parts of the figure, the setpoints and resulting water-heights and in the lower parts the applied control actions are shown. It can be seen that both agents were able to learn appropriate control actions with this setting, so that the demanded setpoints can be reached. However, the figure also shows that the learned control actions do seem to contain some noise, even though this evaluation was done with deterministic policies (so the stochastic part of the policies needed for exploration during training was deactivated). By training with more episodes or by doing further hyper-parameter optimization of the algorithms, this noise could probably be reduced and the performance could be enhanced. But since the goal of this evaluation is only to show that learning an acceptable control strategy is possible in this setting, this result is sufficient.

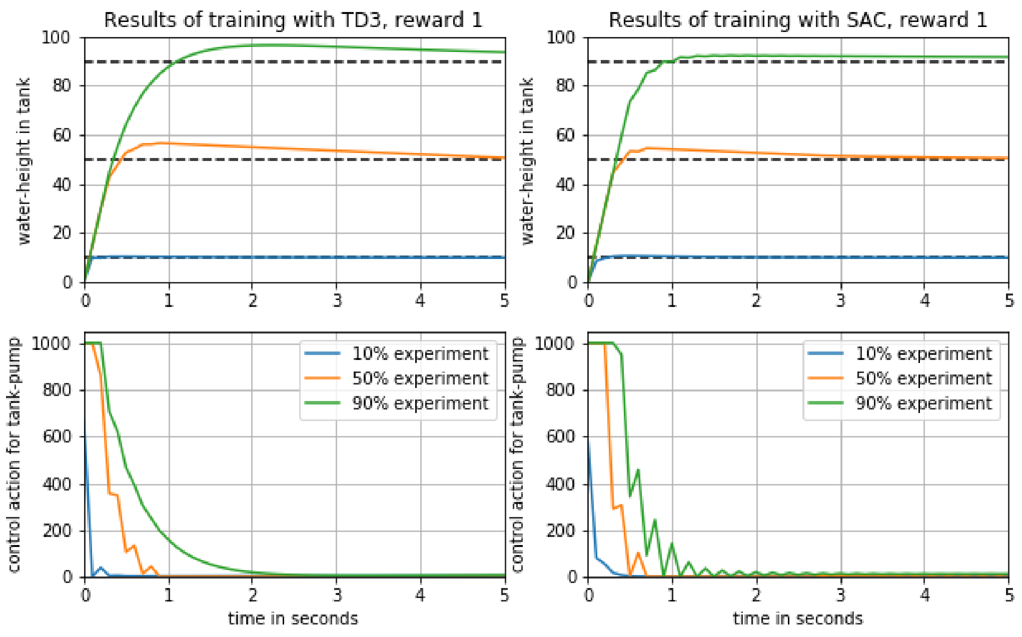

Figure 3.6: Evaluation of the learned policies of TD3 and SAC algorithms, trained with reward 1 on the exemplary water-tank control problem in the first use-case setting.

Second use-case evaluation So far, only the first use-case of giving the RL agent direct control over the control action u was evaluated, but of course also the second use-case of learning optimal PID parameters (see Section 3.1.2) should be investigated. Therefore, the training was done similar to before, so 1000 episodes with single random step function experiments were executed with the four selected algorithms and the four defined reward settings.

Figure 3.7 shows the learning curves of the four selected algorithms trained in the setting of the second use-case with reward function $r_1(t)$. For the TD3 and SAC algorithms the results are similar to the first use-case. They are able to improve their received episode rewards over time and seem to converge within the 1000 trained episodes. The DDPG algorithm however shows some dips in the learning curve but can increase the episode

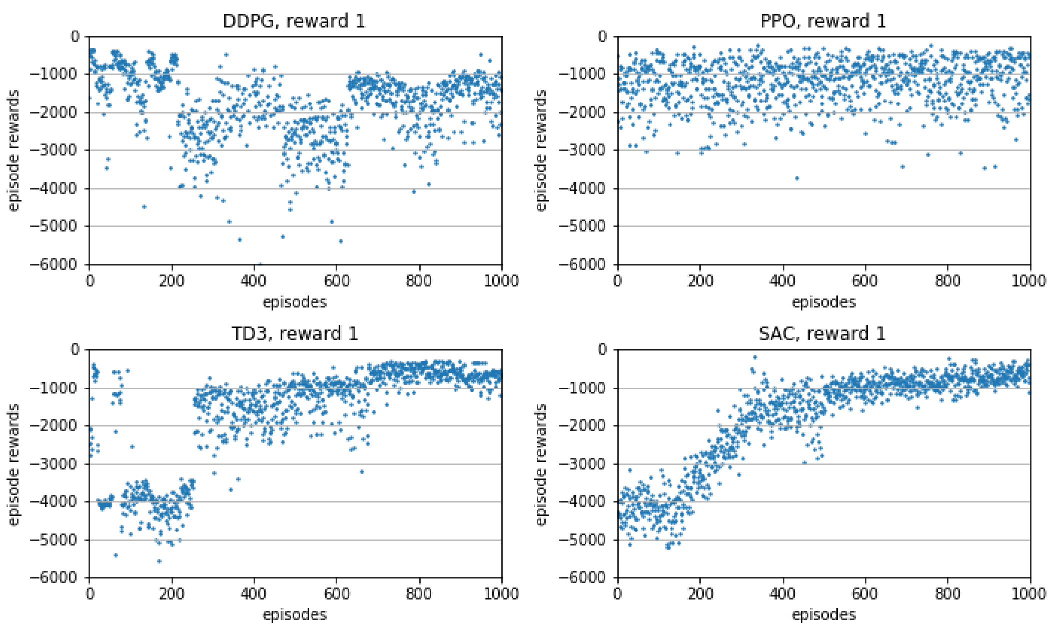

Figure 3.7: Comparison of learning curves of the four selected algorithms with reward 1 on the exemplary water-tank control problem. Training in the setting of the second use-case with single random step functions as input.

reward again afterwards. The PPO algorithm shows no observable improvement over time with this overall setting.

The more interesting part of the second use-case evaluation is looking at the performance of the learned policies. Thus, Figure 3.8 shows the results of a 50% filling experiment with the final trained TD3 and SAC agents. This time only one filling experiment is shown because in the lower parts of the figure the learned discrete PID-controller gains K_p, K_i and K_d are plotted over time. They show rather large and noisy variations in the first two seconds of all experiments, so only the 50% experiment for each algorithm is plotted, but the 10% and 90% results are very similar. With the learned parameters, the controllers are able to reach the demanded setpoint, despite slight oscillations at the beginning of both experiments, due to the noisy variations in the learned gain-parameters. This could be a consequence of insufficient training time or a wrong choice of algorithm hyper-parameters for this problem. But as already discussed earlier, the goal of this evaluation was only to show that learning acceptable parameters is possible in this setting, which could be proven. In direct comparison to the results of the first use-case, however, the deviations from the desired setpoint value are larger with the same standard-algorithm settings.

Input generation evaluation As a last evaluation with this benchmark water-tank control problem, the influence of the two discussed options for generating training experiments is looked at. In all evaluations so far, the training episodes were defined as

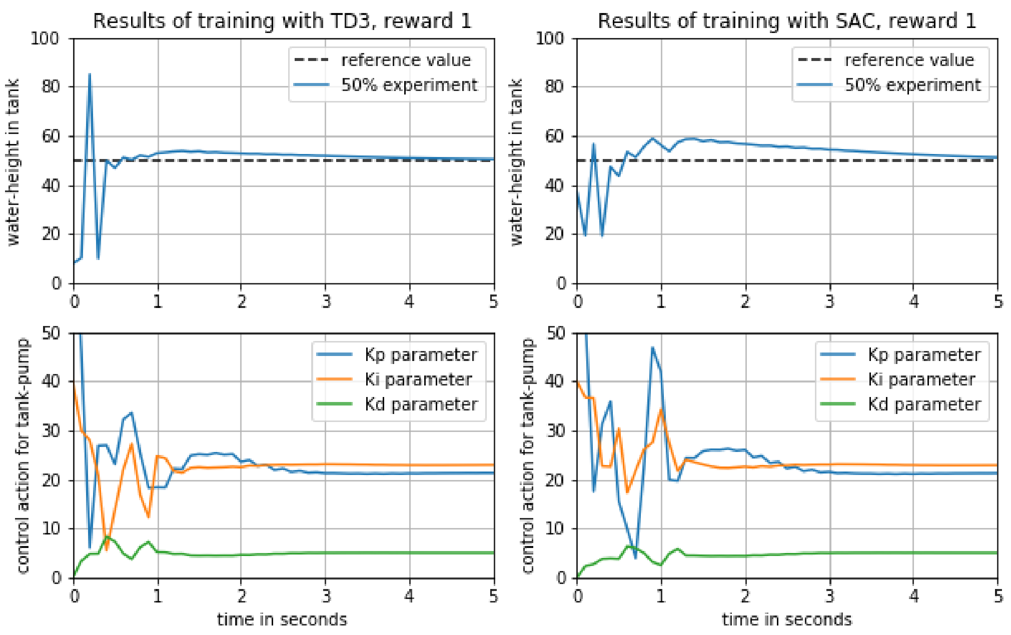

Figure 3.8: Evaluation of the learned policies of TD3 and SAC algorithms, trained with reward 1 on the exemplary water-tank control problem in the second use-case setting.

single step functions with random setpoints and a duration of 5 seconds. With the chosen discretization time T_d of 100 ms, this means that the RL agents could observe 50 timesteps of experience before a learning update, as the algorithms update their policy parameters after every episode. A concern that existed with this option was, that due to the random setpoints the best possible reward that an agent could achieve differs from episode to episode and therefore the policy updates could become unstable. In the evaluations seen so far, however, this does not seem to be an issue.

Nevertheless, a second option of experiment design was tried out, because it could have increased the learning performance. In this version, an episode is defined as a set of multiple consecutive step functions with varying setpoints. In particular, for this evaluation 10 steps with random setpoints within the tank limits were chosen with a total experiment duration of 50 seconds. This leads to a total of 500 timesteps of experience before a learning update occurs and more importantly the maximum achievable reward per experiment is more evenly distributed.

All evaluations seen so far were re-run with this second option for 500 episodes of training time. The resulting learning curves were comparable to the results seen so far and also the resulting policies behaved similarly with only slight deviations. After all, this means that only a reduction of episodes could be achieved at the cost of longer experiments.

4 Training and Evaluation

After the validation of the developed learning strategy on an exemplary benchmark problem, the next step was to use the provided simulation model of a heavy-duty diesel engine and train it with different settings, algorithms and hyper-parameters. In this chapter, the setup of this training as well as the evaluation method is described and afterwards the results are evaluated. However, due to the complexity of the engine simulation model, problems with the training time emerged that are also discussed.

4.1 Training with Engine Model

In this section, the general training setup is explained in more detail by describing the inputs to the RL agent, the design of reward and training episodes as well as the used algorithms and hyper-parameters. Also, the aforementioned problems with the training time are explained and counter-measures are discussed.

4.1.1 Setup

State Description As seen in the description of the currently used control system in Section 1.2, the characteristic values of the air-path control problem are the pressure p_{im} in the engine intake manifold, also called boost pressure, as well as the excess-air factor λ of the engine. Because the corresponding demand values for these control variables can be taken from the existing calculation maps (as described in Section 1.3), the deviations between the given boost pressure and lambda demands and the currently measured or calculated values ($\Delta p_{im}(t)$ and $\Delta\lambda(t)$) are used as the inputs of the RL observation vector. Due to the use of the feedback-values of the control variables in the input vector, the control loop is closed and the resulting agent can be considered as a control algorithm.

The second and third inputs to the RL observation vector were chosen as the current engine speed n_{eng} and injected fuel mass-flow $\dot{m}_{inj}$ to the engine. They were selected due to the fact that they are the main signals used to define the engine operation point in the current map-based controller structure. The input vector could be further enhanced by including different measured temperatures and pressures in the air intake, but since the ambient conditions in the simulation model are held constant during training it would bring no additional information. In another approach, with a goal of obtaining a more robust policy, it would make sense to include also other signals and vary the ambient temperature and pressure settings during training. This training approach, however, focuses only on the feasibility of learning a “good” control strategy with the given simulation model.

To account for the dynamic limitation in certain operation conditions of the engine, additional inputs to the RL observation vector were given as the difference between the last action output and the limited real output signals. So if these signals are different from zero, the RL agent knows that the output signals were limited in some form. The thought of giving this information to the agent is similar to the current anti wind-up method used for the PID-controllers. The difference to classical anti wind-up is, that no active counter-measures are taken, but the agent has to learn the best response to such situations on its own.

Reward design The reward design for training with the large engine simulation model was chosen to be kept rather simple. An evaluation of different rewards was already done on the benchmark water-tank control problem in Section 3.3.2. It showed that using a simpler reward design of the negative squared or absolute error worked better than using more complicated sparse or shaped reward functions. Therefore, the best working design of using the negative squared error between demand value and feedback value is also the basis for the training with the engine air-path model. However, in this setting not only one but two control variables exist that should follow a given demand value, which leads to an extended reward formulation,

$$r(t) = -\frac{1}{2}\left[\left(\frac{\Delta\lambda(t)}{\lambda_{max}}\right)^2 + \left(\frac{\Delta p_{im}(t)}{p_{max}}\right)^2\right]. \tag{4.1}$$

Due to the differences in the value ranges of both control variables, they are additionally normalized by dividing through the largest allowable values, that is λ_{max} and p_{max} respectively. This way, the normalized squared differences can be added without favoring one of the two control variables.

Algorithms and Hyper-Parameters The algorithms that performed best within a reasonable amount of experiments in the evaluation of the benchmark problem before were the SAC and TD3 algorithms. This result is also in line with the assessment of the developers of the stable-baseline package, who provide some recommendations as well as tips and tricks for dealing with various algorithms in their documentation[1]. For problems with continuous actions, where only a single training process is possible concurrently, SAC and TD3 are listed as current state-of-the-art algorithms. Therefore, and in order to keep the amount of different training settings to a reasonable amount due to the

[1]See: `https://stable-baselines.readthedocs.io/en/master/guide/rl_tips.html` (accessed April 24, 2021)

Set ID	Layer norm	Learning rate	Buffer sizes	Gamma
SAC-HP0	False	3e-4	normal	0.99
SAC-HP1	False	*decay*(3e-4)	normal	0.99
SAC-HP2	False	3e-4	large	0.99
SAC-HP3	False	*decay*(3e-4)	large	0.99
SAC-HP4	False	*decay*(3e-4)	large	0.98
SAC-HP5	True	3e-4	normal	0.99
SAC-HP6	True	*decay*(3e-4)	normal	0.99
SAC-HP7	True	3e-4	large	0.99
SAC-HP8	True	*decay*(3e-4)	large	0.99
SAC-HP9	True	*decay*(3e-4)	large	0.98

Table 4.1: Set of modified SAC hyper-parameters in training

training time issues explained later in this section, only those two algorithms are used for training with the large engine air-path simulation model.

Unlike in the previous evaluation with the exemplary problem, the goal of this training is to actually try and find the best control strategy possible. For that reason, sets with different hyper-parameters were defined to see how much influence a change of these training parameters has on the final performance of the learned policies.

A great help with the choice of the hyper-parameter sets was the RL Baselines Zoo [34], which is a collection of trained RL agents with tuned hyper-parameters on different benchmark example problems. Because for both selected algorithms the best hyper-parameters on different problems with varying complexity are documented, different well working parameter-sets could be identified that are worth a try. Also, a lot of unnecessary parameter variations could be ruled out by using the tuned hyper-parameter documentation. For instance, the documentation of the SAC algorithm shows that the best results on all problems were always achieved with the initial policy network layout (two hidden layers of 64 neurons each).

The selected sets of hyper-parameters that are evaluated with the SAC algorithm are shown in Table 4.1. The first column shows a “Set-ID”, that is used to refer to experiments more easily. The second column describes if layer normalization (see [35]) is used to normalize the layer outputs of the policy neural network. The entries in the “learning rate” column tell if either the given rate was used directly or a $decay(x)$ function is applied, that linearly decreases the given rate until the end of the experiment. The “buffer sizes” column tells if either the “normal” setting, with initial sizes for the replay buffer and initial batch size for gradient updates, or a “large” setting, with a buffer size of 100.000 (2 times the initial value) and a batch size of 256 (4 times the initial value) was used. Finally, the last column shows the used discount factor “gamma” for the expected reward formulation. The first hyper-parameter set SAC-HP0 corresponds to the use of only initial algorithm parameters.

Set ID	Layer size	Learning rate	Action noise	Buffer sizes
TD3-HP0	(64, 64)	3e-4	None	normal
TD3-HP1	(64, 64)	1e-3	$\mathcal{N}(0, 0.1)$	normal
TD3-HP2	(64, 64)	1e-3	$\mathcal{N}(0, 0.1)$	large
TD3-HP3	(64, 64)	1e-3	$\mathcal{N}(0, 0.2)$	large
TD3-HP4	(128, 128)	1e-3	None	normal
TD3-HP5	(128, 128)	1e-3	$\mathcal{N}(0, 0.1)$	normal
TD3-HP6	(128, 128)	1e-3	$\mathcal{N}(0, 0.1)$	large
TD3-HP7	(128, 128)	1e-3	$\mathcal{N}(0, 0.2)$	large
TD3-HP8	(400, 300)	1e-3	None	normal
TD3-HP9	(400, 300)	1e-3	$\mathcal{N}(0, 0.1)$	normal
TD3-HP10	(400, 300)	1e-3	$\mathcal{N}(0, 0.1)$	large
TD3-HP11	(400, 300)	1e-3	$\mathcal{N}(0, 0.2)$	large

Table 4.2: Set of modified TD3 hyper-parameters in training

Similarly, Table 4.2 shows the selected sets of hyper-parameters that are evaluated with the TD3 algorithm. Here, in contrast to the SAC parameters, also the network layout is changed by varying the number of neurons in the two hidden layers. This is shown by the values in the "layer size" column. The TD3 parameter documentation shows that all slightly more complicated problems work better with a higher learning rate of 1e-3 instead of the initial 3e-4. For this reason, apart from the initial TD3-HP0 set, all variations use this setting, which can be seen in the "learning rate" column. The TD3 algorithm also works better when noise is added to the selected actions in order to enhance the exploration. The added noise can be seen in the "action noise" column, where $\mathcal{N}(0, \sigma^2)$ refers to a normal distribution with zero mean and a variance of σ^2. The entries in the "buffer sizes" column have the same meaning as in the SAC hyper-parameter table described before.

Experiment Input Generation The recompilation of the simulation model at the start of each new episode is partly responsible for the long training time with the large engine model. Therefore, it is desirable to reduce the amount of episodes needed for training, as long as the training process is not harmed. An insight of the evaluation with the benchmark water-tank control problem in Section 3.3.2 was, that such a reduction of training episodes can be achieved at the cost of longer experiments, but the learning curves were still comparable. Therefore, it makes sense to train the large engine model with longer episodes consisting of multiple steps of the engine operation point instead of just using single step experiments.

The operation point of the engine is mainly defined by two external input signals to the simulation, the demanded engine speed and torque. To define a training episode, random operation points are generated, consisting of random speed and torque demands within fixed limits. These points are held for a duration of 4 seconds before a transition to a new random point. In total, 4 such operation points are generated per episode with a total duration of 32 seconds. Figure 4.1 shows an example of a random training episode.

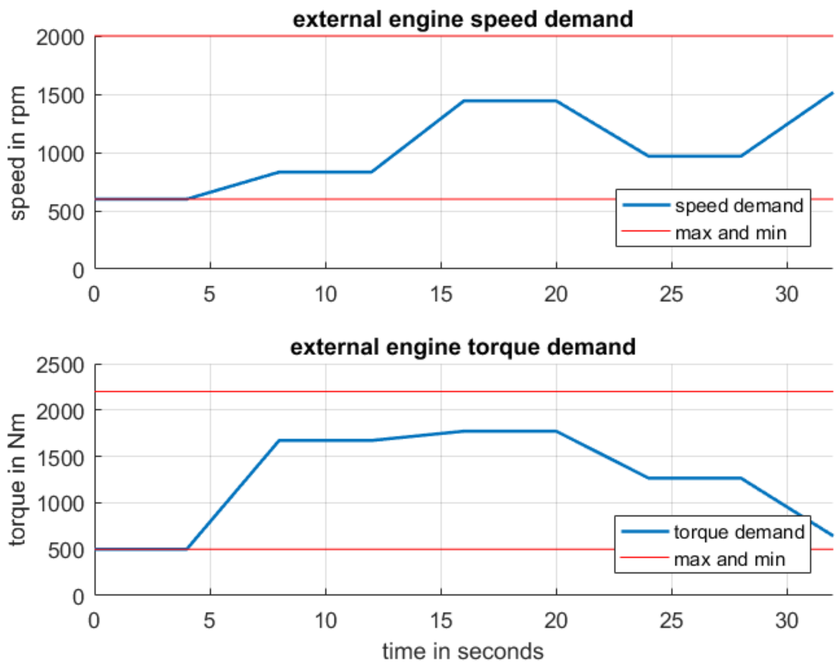

Figure 4.1: Example of the generated inputs to the engine model that define one training episode. It contains four randomly generated engine operation points define by a engine speed and toque demand.

The design of experiment inputs also had to take into account that certain engine conditions can lead to non-controllable states of the engine, after which the existing control software arbitration switches to an open-loop control mode. Such switches are of course undesirable during training because in these cases the inferred actions of the RL agent are not used and the agent has no influence on the system behaviour anymore. As a consequence, the external input signals were limited in a way that no non-controllable states are reached during training. These limitations include that a certain minimal engine speed and torque is required at all times (the engine is not idling) and rapid accelerations are limited.

4.1.2 Training time problems

The provided engine simulation model was designed and calibrated by engineers at AVL and serves as an accurate representation of a real diesel engines behaviour. As a result of the provided model accuracy, the simulation model is also correspondingly complex and therefore computationally expensive to run. Combined with the fact that reinforcement learning has a poor data efficiency and multiple hundreds or thousands of experiments are needed to learn an appropriate policy (as seen before in Section 3.3.2), this can lead to unreasonably long training times in this setting.

Training time analysis To get a better idea of the causes for the long training time with the engine model, a short analysis of the issues and their causes is provided here. Of course, the main issue is the aforementioned size and complexity of the diesel engine simulation model, but other factors like the behaviour of Simulink also play a role. As already discussed in the considerations about the experiment inputs, Simulink re-compiles the model at the start of each new training episode. On the small benchmark problem this was less of an issue but the large engine simulation model needs about 50.4 seconds for one model compilation alone (measured using the `tic`- and `toc`-commands in Matlab). These re-compilations are in fact unnecessary because structurally nothing changes in the model from one episode to another. Preventing them could reduce the overall training time a lot, but due to the proprietary nature of Matlab and Simulink, this behaviour could not be changed.

Another factor for the long training time is the constant pausing and continuation of the Simulink model due to the training setup needed for RL training. It leads to an execution time of about 1.35 seconds for just one 100 ms time-step in the environment (again measured in Matlab). One whole experiment of 32 seconds therefore needs about 482 s (including the compilation time at the beginning) just for the step-wise model execution of one episode, without the training time of the RL algorithm factored in. For comparison, if the same model is executed continuously (for example with the `sim()`-command in Matlab), it finishes in about 145 s.

Measurements in Python showed that the whole training time for one episode on average is about 610 seconds (on a standard office PC), which includes model compilation, step-wise execution and RL training updates. This shows that also the learning algorithms need some time to update their policies, whereby training the actor and critic neural networks presumably is the main time-consuming task. For this reason, many off-policy algorithms (for example PPO) are able to be trained with multiple environments that generate experience in parallel while the policy updates happen in a separate process. Due to the high computational requirements imposed by the simulation model, however, parallel training is not possible with the provided hardware.

Model Simplification To somehow reduce the long training times that are needed with the large air-path model, the structure of the model itself was further analysed. The whole diesel engine simulation model actually consists of two main simulations models, one for the internal engine dynamics and one for the exhaust-gas after-treatment system (EAS), to be able to accurately model the engine and tailpipe emissions. Since the EAS has little influence on the rest of the engine air-path, this gave rise to the idea to simply remove this part of the simulation. The influence of this system on the rest of the engine air-path is, that a certain back-pressure is generated depending on the loading of the different catalysts and filters. When removing the model, the corresponding temperature and pressure signals that exist as a feedback to the engine dynamics model were simply set to ambient conditions. As a result, the simulation time for one 100 ms time-step of the model in Matlab improved to about 0.75 seconds and the whole training time for one episode in Python fell to about 350 seconds, which corresponds to almost a halving of the execution time with the full model.

To evaluate the qualitative effects of the removed EAS on the characteristic variables of the air-path control, several experiments with the full and the simplified model were done. A comparison of the simulated air-excess ratio λ, the intake manifold pressure p_{im} and temperature t_{im}, as well as the output actuator positions of the existing controller showed, that the relative errors of all simulated signals with the simplified model were within $\pm 5\%$ of the original signal. This small loss of accuracy can be justified by the large time savings during training. Also, the following evaluations are always done with the full, non-simplified model including the exhaust-gas after-treatment system and the simplified model is just used for training.

4.2 Evaluation

In this section, the necessary steps to evaluate the trained RL agents are described and an overview of the test cycles that are are used for the evaluation is given.

It should be noted that the evaluation only happens on the provided simulation model of a heavy-duty diesel engine. Although this accurately calibrated model represents a “good” representation of a diesel engine and the full model without simplifications is used, the results on a real diesel engine could still be slightly different.

Since the RL agents are trained only in simulation, their performance on the real system can vary depending on the mismatch between the real and simulated process. In literature, this problem is known as simulation-to-reality gap. There are several approaches to close this gap, for instance pre-training the policy in simulation and then fine-tuning it on the real hardware. Another approach could be domain randomization, which is a technique where certain parameters of the simulated model are randomized, so that the agent interacts with many different simulation models and thereby learns a more general policy. A successful example of this technique was shown for example in [36], where the authors were able to transfer a policy, that was trained solely in randomized simulations, to a complex robotic hand and perform complex dexterous movements with it.

However, in this thesis an evaluation of the learned policies on a real diesel engine was not possible due to time and budget reasons. As a result, approaches to handle the simulation-to-reality gap are not further discussed here but could be an interesting subject of future research projects.

4.2.1 Policy Implementation

To be able to better evaluate the trained agents, their final policies had to be implemented in the original Simulink model, in order to test the learned strategies with new unseen experiments. This sole execution of a trained neural network without learning is called inference. It is done to be able to run any sorts of evaluation experiments and not just test with the same kind of episodes used in training. It was also a requirement of this thesis that the final trained RL controller is able to run inside the original Simulink model, so that the trained policy is universally applicable.

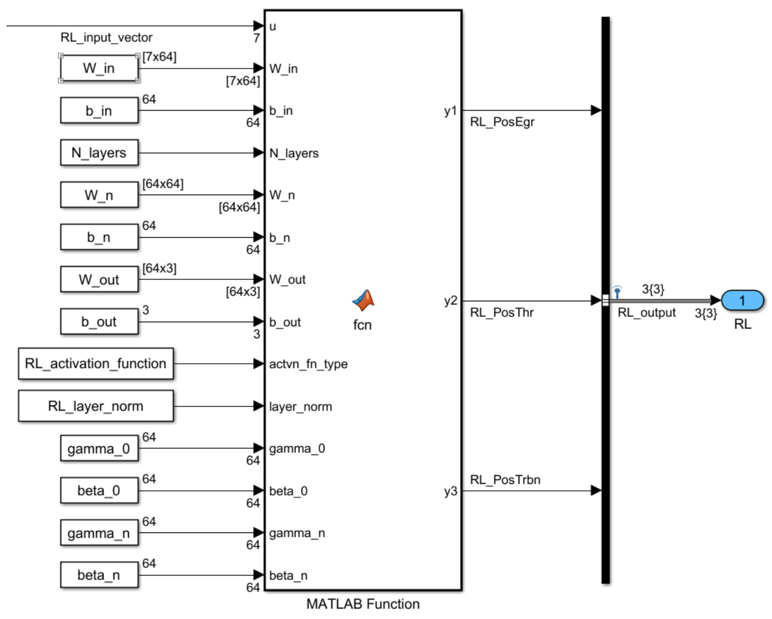

Figure 4.2: Implementation of the policy network in Simulink for better evaluation. The input signals apart from the discussed input vector correspond to the trained network weights exported from Python. The three output signals correspond to the actuator positions of the EGR, throttle and waste-gate valves.

Figure 4.2 shows the implementation of the policy network in Simulink as a Matlab-function-block. The SAC and TD3 algorithms that are used for training are both actor-critic methods, which means that one neural network is used for the approximation of the policy and one network (or more) approximates a value function. For this evaluation part, only the policy network is of interest. To run a trained neural network in inference, it is sufficient to know the layout of the network and the final trained weights and biases. Since training is done with different network layouts with varying layer sizes and layer normalization active or not, a Matlab-function-block was used for the implementation in Simulink. This way, multiple variants of network layouts can be realized with a single block. The weights of the trained policy network can be exported from Python and are used in Simulink via the Matlab workspace.

4.2.2 Stationary and Transient Test Cycles

For the evaluation of the different algorithms and hyper-parameters, their performance is measured on two standard engine test cycles. These test-cycles, called World Harmonized Stationary Cycle (WHSC) and World Harmonized Transient Cycle (WHTC) were defined in the the Global Technical Regulation (GTR) No. 4 [37] developed by the United Nations Economic Commission for Europe (UNECE). They were created as a unifying test procedure for heavy-duty engines covering typical driving conditions in the EU, USA, Japan and Australia.

The WHSC test is a steady-state engine test schedule that consists of different modes with different engine speed and torque demands. A graphical view of these modes is shown in Figure 4.3. The time from 0 to about 200 seconds at the beginning of the test serve as a pre-conditioning of the engine in idle mode. During this time typically no closed-loop control of the desired variables for the air-path is possible.

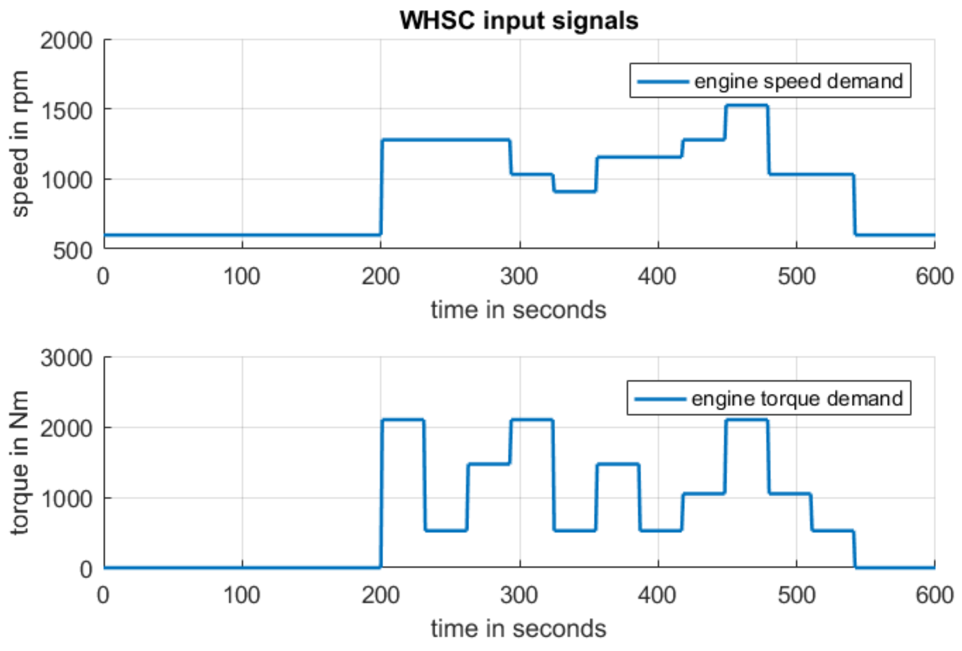

Figure 4.3: Input signals to the engine model according to the World Harmonized Stationary Cylce (WHSC).

The WHTC also contains several motoring segments. A graphical view of the engine speed and torque demand values over time is shown in Figure 4.4. It can be seen that it is a lot more dynamic than the WHSC test. The motoring segments are defined by zero torque demand and minimum engine speed. During these times, a closed-loop control of the the air-path is not possible, which means that several switches between open- and closed-loop happen during the test cycle.

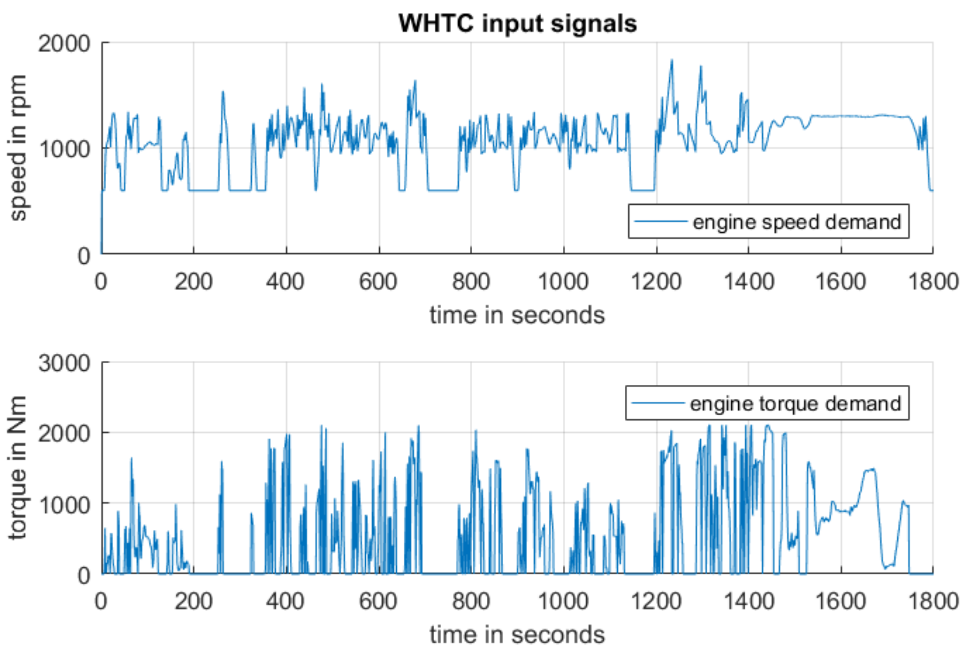

Figure 4.4: Input signals to the engine model according to the World Harmonized Transient Cylce (WHTC).

4.3 Results with first use-case

In this section, the results of evaluating different training variants with the diesel engine simulation model are discussed. First, a comparison of different algorithms and hyper-parameters in the setting of the first use-case is shown, followed by a more detailed examination of the best results. The existing hand-tuned air-path controller in the provided model serves as a reference for comparing all learned strategies in this section.

4.3.1 Algorithm and hyper-parameter variations

All variations were trained for 1000 episodes in the training setup explained in Section 4.1. The training time for each episode with the simplified model was about 350 seconds on average, which accumulates to roughly 97 hours of training for each hyper-parameter variant. Most, but not all variants showed a converging behaviour in their learning curves during this time. However, to obtain a fair comparison, all variants were trained with the same number of episodes whether they converged or not.

Error measures As a main measure to compare different setting, the mean absolute error (MAE) between the desired demand signals and the actual simulated values of the controlled variables is used. It is a commonly used measure to compare controller results

	Stationary Test				Transient Test			
	Lambda		Boost pressure		Lambda		Boost pressure	
Set ID	MAE	RMSE	MAE	RMSE	MAE	RMSE	MAE	RMSE
Reference	0.08	0.212	37.92	129.39	0.180	0.319	148.00	225.05
SAC-HP0	0.538	0.726	224.33	258.96	0.514	0.742	211.56	258.43
SAC-HP1	0.387	0.503	192.06	275.47	0.322	0.449	195.56	261.33
SAC-HP2	0.226	0.319	126.07	174.63	0.242	0.385	175.58	235.16
SAC-HP3	0.453	0.628	215.76	221.34	0.438	0.522	202.89	245.71
SAC-HP4	0.363	0.443	233.16	342.87	0.343	0.441	217.37	286.20
SAC-HP5	0.431	0.501	245.59	308.43	0.583	0.766	233.96	288.14
SAC-HP6	0.521	0.684	142.50	209.27	0.504	0.698	184.81	252.10
SAC-HP7	0.341	0.494	249.93	359.96	0.268	0.383	199.39	274.48
SAC-HP8	0.262	0.366	150.19	190.95	0.256	0.376	188.69	241,02
SAC-HP9	0.535	0.716	256.56	311.29	0.513	0.726	217.65	263.50
TD3-HP0	0.688	0.799	389.23	453.81	0.639	0.827	296.19	358.45
TD3-HP1	0.350	0.432	237.13	346.28	0.320	0.422	190.15	257.26
TD3-HP2	0.402	0.498	240.86	333.75	0.355	0.491	206.27	274.64
TD3-HP3	0.436	0.613	197.51	239.88	0.487	0.681	200.20	260.50
TD3-HP4	0.538	0.726	224.48	259.08	0.514	0.742	211.60	258.47
TD3-HP5	0.357	0.441	239.97	345.38	0.342	0.442	221.20	291.20
TD3-HP6	0.341	0.425	209.76	316.97	0.331	0.432	205.31	280.84
TD3-HP7	0.352	0.434	225.53	329.22	0.339	0.438	217.83	285.56
TD3-HP8	0.365	0.454	249.09	336.28	0.350	0.447	221.72	286.08
TD3-HP9	0.361	0.483	218.76	332.79	0.351	0.483	217.18	276.23
TD3-HP10	0.538	0.726	224.51	259.14	0.514	0.742	211.56	258.40
TD3-HP11	0.324	0.416	209.38	332.74	0.355	0.472	202.08	276.76

Table 4.3: Results of algorithm and hyper-parameter variations when training in the setting of the first use-case.

because all deviations from the demands signal are proportionally weighted. Another option to compare the results is the root-mean-square-error (RMSE), where each error is weighted proportional to the square of the error size. Therefore, larger errors and overshoots have more impact. Since the two measures provide different information, both values are given in Table 4.3 to enable a better evaluation.

Result discussion Table 4.3 shows the results of the training evaluations with different hyper-parameters for both algorithms, trained in the setting of the first use-case. All variants are evaluated with both test cycles and the results for lambda and boost pressure control are given separately. The first column shows the set-IDs of the used hyper-parameter setting (that were defined in Tables 4.1 and 4.2). The first row additionally shows the results of the reference controller, that was explained in detail in Section 1.2.

It can be seen that none of the trained variants achieves lower error measures than the reference controller, neither on the lambda nor on the boost pressure control. The best training results overall could be achieved with the SAC-HP2 set, that uses the SAC algorithm with a fixed learning rate of 3e-4, the default policy network layout without layer normalization, the initial reward-discount-factor $\gamma = 0.99$ and the “large” setting for the replay buffer and batch size. On the stationary WHSC test, the MAE of all trained RL controllers is even several times higher. This already is an indication that the RL controllers have problems holding stationary points accurately. On the dynamic WHTC test, however, the error measures of some trained variants come close to the reference results. This means that the RL controllers are able to follow given demand signals even in transient conditions.

When comparing the results of the different hyper-parameter sets for the SAC and TD3 algorithm independently, it can be seen that all used parameter-modifications worked better than the initial settings SAC-HP0 and TD3-HP0. This is likely due to the fact that hyper-parameters were only changed if they also led to improvements in the benchmark-problem-collections of the RL Baselines Zoo (see [34]). Some observations for future hyper-parameters choices are visible from a more detailed look at the results in Table 4.3. If settings with only one different hyper-parameter are compared, that are otherwise identical, it can be seen, for instance, that larger replay buffers and batch sizes are beneficial when training with the SAC algorithm. Or, when training with the TD3 algorithm, adding a small action noise with a variance $\sigma^2 = 0.1$ always yields better results. But other than that, no clear structure due to the choice of the hyper-parameters is noticeable. In reinforcement learning, the best parameter setting always depends on the specific problem. This shows that many variations with different algorithms and hyper-parameters are necessary to find good results and justifies the amount of training time that was spend on this variation.

4.3.2 Qualitative results

In addition to comparing the training results with different settings through several numerical error measures, a qualitative analysis of the best results was also carried out. For this purpose, the performance of the trained RL controller with the best hyper-parameter set SAC-HP2 was compared to the reference controller on the stationary and transient tests in more detail.

Steady-state case The evaluation of the best training results on the steady-state WHSC test are shown in Figure 4.5. Only the period of time in which closed-loop control was possible is shown. It can be seen that the RL controller can follow the given demand signals for lambda and boost pressure, but is not stationary accurate. The reference controller on the other hand can follow both demand signals nearly perfectly. It should be noted, however, that the reference controller was hand-tuned on similar

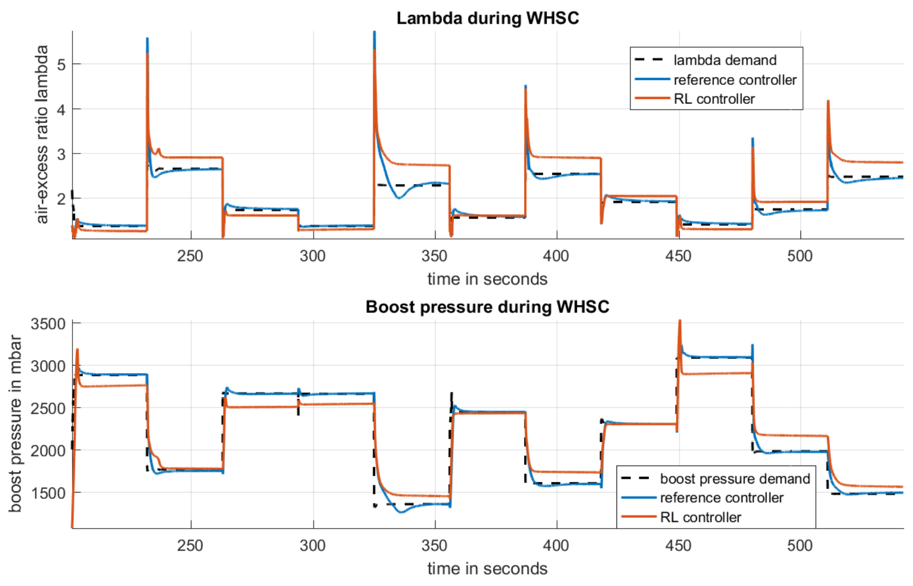

Figure 4.5: Comparison of trained RL controller with hyper-parameter set SAC-HP2 against reference controller on the WHSC test. The time scale excludes the pre- and post-conditioning phases of the engine where closed-loop control is inactive.

engine operation points. Interestingly, the RL controller performs better in some operation points than it does in others. For instance, during about 360 and 385 seconds, the control performance of the RL controller is even better that the reference controller. However, in other operation points, for example between 325 and 360 or between 385 and 420 seconds, there is a large difference in the controlled lambda of the RL controller and the demand value. Such operation points with a high lambda demand are characterised by a low engine speed and torque demand. A possible explanation for this behavior could be, that operation points with low engine speed and torque demand were underrepresented during the training process. As explained in Section 4.1, the training episodes contain randomly generated operation points and most combinations of engine speed and torque do not cause such high lambda requirements.

Transient case The evaluation of the best training results on the transient WHTC test are shown in Figure 4.6. It can be seen that the results with the RL controller look very similar in comparison to the reference controller and both are able to follow the given demand signals for lambda and boost pressure for most parts of the test cycle. However, due to the long duration of the test with 1800 seconds, details are not visible. Also, the test contains several motoring segments where a closed-loop control is not possible and therefore open-loop control is used. To better see when this happens and to have a more detailed view, Figures 4.7 and 4.8 show excerpts from the WHTC test evaluation between 785 and 845 seconds.

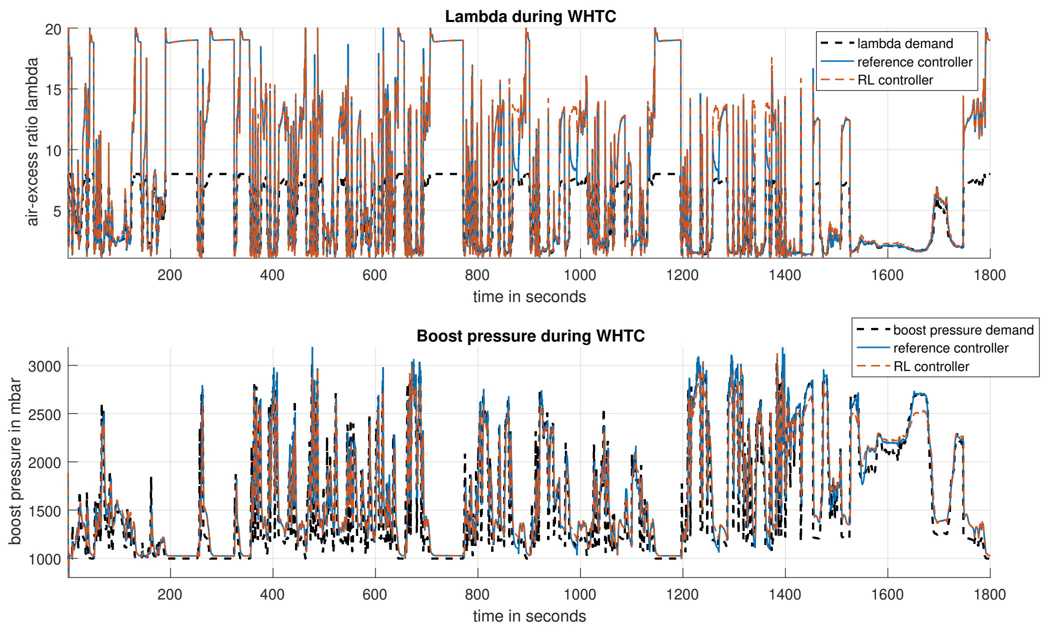

Figure 4.6: Comparison of trained RL controller with hyper-parameter set SAC-HP2 against reference controller on the full WHTC test.

The upper part of Figure 4.7 shows a more detailed comparison of the lambda control results with the trained RL controller and the reference controller. In the lower part, the time periods with disabled close-loop control are marked. For the comparison, these times in open-loop mode should be ignored since the RL controller had no influence on the system. During the times with active RL control, however, it can be seen that demanded lambda signal can be followed surprisingly well, even compared to the reference controller.

Figure 4.8 similarly shows a more detailed view of the results for the boost pressure control. Here, as well, it can be seen that the performance of the RL controller can follow the demanded boost pressure signal nearly as good as the reference controller.

However, these figures just show excerpts from the whole transient test case. In order to have a more meaningful comparison, an additional statistical evaluation of the relative errors between demand signals and simulated control values was done. Due to the frequent switches between open- and closed-loop control modes during the transient WHTC test, this statistical analysis only considers the times with active closed-loop control. The results of this statistical evaluation for the control variable lambda can be seen in Figure 4.9.

The upper part of Figure 4.9 shows the relative errors between the demanded and simulated lambda values generated by the RL controller, for the times with active closed-loop control. The lower part of the figure shows an estimation of the probability density

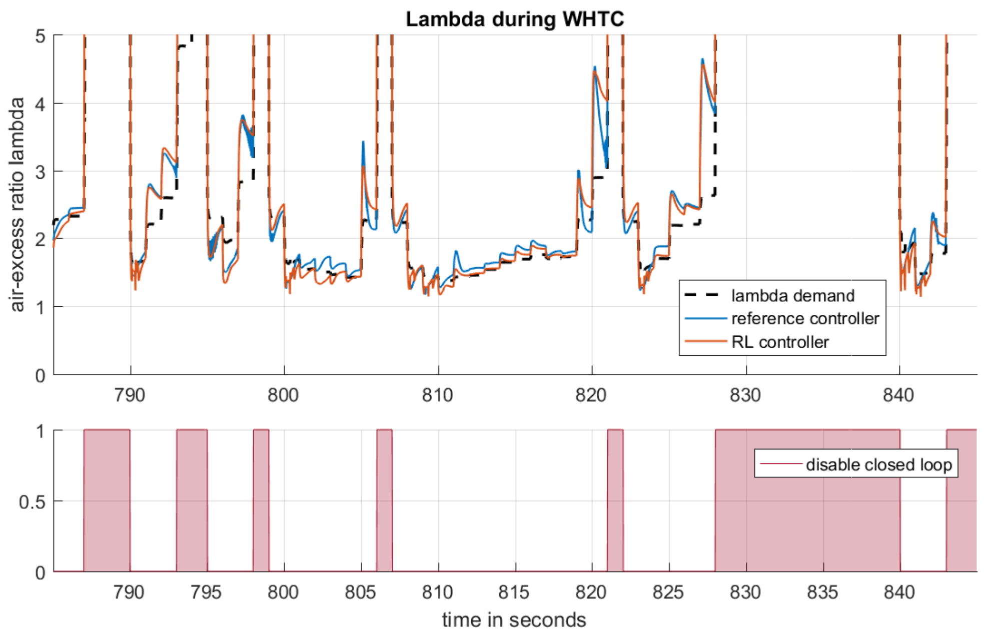

Figure 4.7: Detailed comparison of lambda control between trained RL controller with hyper-parameter set SAC-HP2 and reference controller on excerpt from WHTC test.

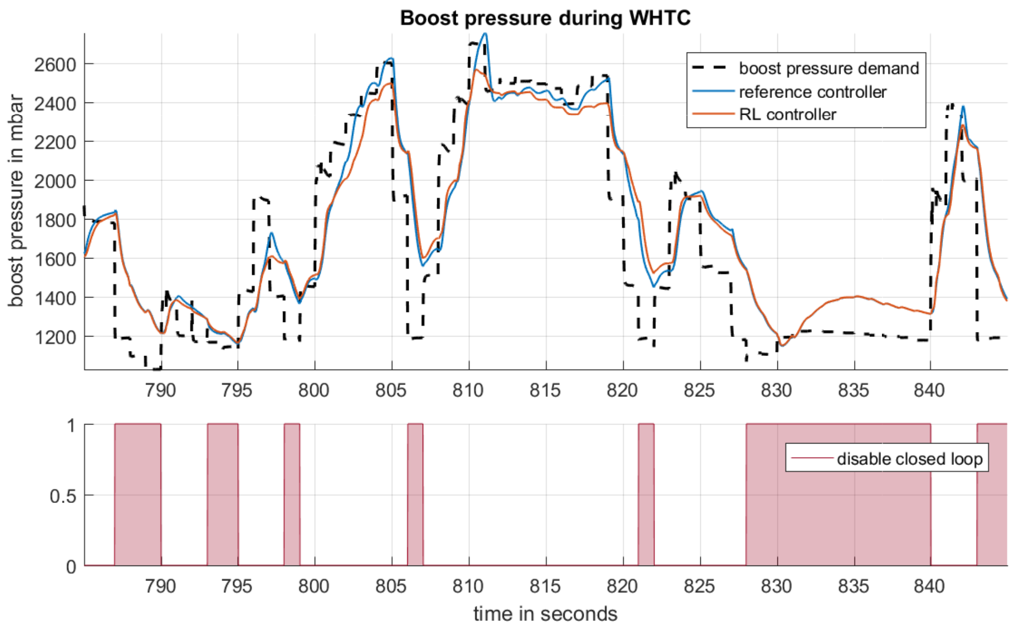

Figure 4.8: Detailed comparison of boost pressure control between trained RL controller with hyper-parameter set SAC-HP2 and reference controller on excerpt from WHTC test.

function (PDF) for the relative errors of the reference controller and the trained RL controller. When only looking at the relative errors, a lot of high spikes are visible due to overshooting or undershooting of the demand signal. However, the PDF of the errors shows that most relative errors actually are within a range of ±10% compared to the demand value. To be more precise, 76.9% of all relative lambda errors of the RL controller and 84.1% of the relative errors of the reference controller fall within this ±10% interval. Although these are good results for the RL controller, the reference controller still performs better. It can also be seen from the shape of the PDFs, that most of the reference controller errors are in a narrow band around zero and then spread out in a bell shape, while most errors of the RL controller are near −10%. Therefore, is can be concluded that the results with this trained RL controller are acceptable for a transient test cycle, but the reference controller still performs better.

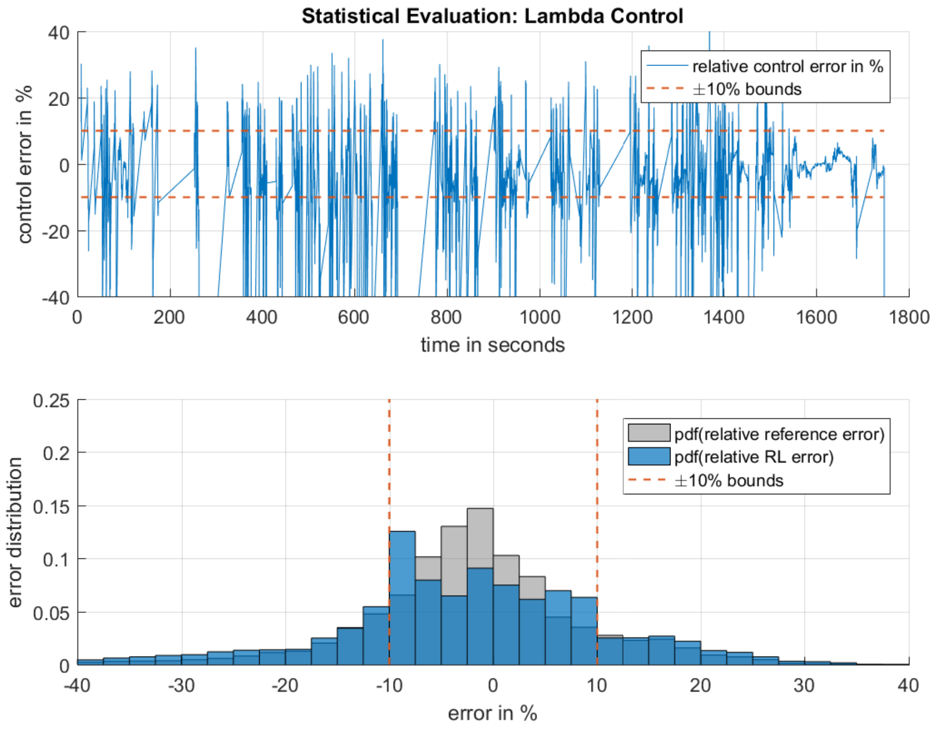

Figure 4.9: Statistical evaluation of lambda control results of trained RL controller with hyper-parameter set SAC-HP2 and reference controller on WHTC test. Only times with active closed-loop control were considered.

The statistical analysis of the errors for the boost pressure control has shown the same behaviour. Here, 69.8% of all relative boost pressure errors of the RL controller are within a ±10% interval, while the reference controller achieves 71.5%, but in a much more bell-shaped form.

4.4 Results with second use-case

This section shows the evaluation results of training with the diesel engine simulation model in the setting of the second use-case. Therefore, again a comparison of different algorithms and hyper-parameters trained for 1000 episodes was done. However, for this evaluation, the number of trained variants was reduced due to the long training time of each setting and the less promising approach. The evaluations on the small benchmark problem in Section 3.3.2 already showed that training in the setting of the second use-case was not as successful as in the first use-case setting. As a result, Table 4.4 only shows the initial settings of both algorithms and the respective best settings of the evaluation done before.

	Stationary Test				Transient Test			
	Lambda		Boost pressure		Lambda		Boost pressure	
Set ID	MAE	RMSE	MAE	RMSE	MAE	RMSE	MAE	RMSE
Reference	0.08	0.212	37.92	129.39	0.180	0.319	148.00	225.05
SAC-HP0	0.665	0.780	368.24	437.02	0.616	0.816	296.95	363.48
SAC-HP2	0.392	0.457	251.71	323.87	0.426	0.539	223.58	258.17
TD3-HP0	0.675	0.785	376.23	421.53	0.621	0.795	289.92	348.37
TD3-HP1	0.405	0.522	261.37	366.78	0.491	0.563	230.18	301.26
TD3-HP6	0.349	0.451	242.53	296.64	0.387	0.462	210.13	257.46

Table 4.4: Results of algorithm and hyper-parameter variations when training in the setting of the second use-case.

For the SAC algorithm, the best results in the first use-case had been achieved with the hyper-parameter set SAC-HP2. Therefore, the initial algorithm setting SAC-HP0 as well as the best setting from before, SAC-HP2, are also evaluated in the setting of the second use-case here. However, the results in Table 4.4 show that for both settings, the performance in the second setting is worse than the previously seen results. This is true for all error metrics and both the stationary and transient test cycle. For the TD3 algorithm, in the first use-case evaluation the best results on the stationary test cycle were achieved with the hyper-parameter set TD3-HP6, while the best results on the transient test cycle resulted from the TD3-HP1 set. Here, in the setting of the second use-case, the TD3-HP6 set consistently gives the best results, also when compared to the SAC algorithm. However, when compared to the first use-case, this result would only be considered mediocre.

In the end, as already seen before in the evaluation with the small benchmark problem, it can be concluded that the approach of the second use-case to learn time-varying PID-parameters instead of the actual actuator signals, does not deliver control strategies with a comparable performance to the first use-case if the same setting and training times are used.

5 Conclusion and Outlook

This chapter gives a conclusion of the thesis by reviewing the results of building a self-learning data-driven controller via deep reinforcement learning, applied to the air-path of diesel engines. Finally, the current problems of the RL methodology are discussed and an outlook on possible future improvements is given.

5.1 Conclusion

Due to the complex non-linear problem that an air-path control for diesel engines represents, the design of a properly performing engine control strategy is a challenging task. The existing control system that was part of the provided diesel engine simulation model is able to produce satisfactory results through a gain-scheduled PID approach that needs extensive calibration. The main objective of this thesis was to avoid this effort by building a self-learning controller that is able to figure out a "good" control strategy on its own in a data-driven way.

Therefore, the provided diesel engine simulation model was used as an environment to train several deep RL agents in multiple variants. This approach to build a self-learning system, as well as several training aspects, were evaluated on a small exemplary benchmark problem with promising results. It could be shown that with the presented approach, acceptable learning outcomes can be achieved.

However, the model size and complexity of the diesel engine simulation model resulted in long training times. As a consequence, compromises regarding the amount of different training variants and training time of each variant had to be made. In general, it is favourable to have an accurate and complex simulation model to train with, because it better reflects the real system and therefore less mismatch between the simulated and real process exists. This is helpful when the learned results are transferred to a real system, due to the small simulation-to-reality gap. On the other hand, large simulation models are harder to train with, due to the higher computing requirements and long training times that result from their complexity. The easiest solution for this problem would be to scale up the learning process by training with multiple parallel processes, for example on a computing cluster. Unfortunately, this was not possible in this thesis but should be considered for future projects. Typically, more training also leads to better results when using deep RL.

If the results obtained by training with the engine simulation model are compared with the existing control strategy, which is used as a reference, it can be seen that none of the learned variants could achieve a better control performance. Evaluation on a steady-state test cycle showed that the RL controller can follow a given setpoint signal but is not stationary accurate, which leads to a worse performance in comparison to the reference controller. On a transient test cycle, however, the achieved performance of the RL controller was similar to the reference controller. Since the transient test cycle provides a better representation of real driving conditions, these good results can somewhat compensate for the poor results in the steady-state case.

Overall, the results in this thesis serve as a proof of concept that learning a “good” control strategy in a data-driven approach is possible in principle. Although the results were not perfect, they provide a solid basis for further research projects.

5.2 Outlook

Despite the promising results in this thesis and several successes of deep RL in areas like playing games [8][9][10], the methodology still has a lot of flaws. Several examples of why deep RL is hard and does not work well on many problems yet are given for instance in [38]. There, it is argued that deep RL so far only works “well” in settings where it is easy and cheap to generate experience, for instance games. Unfortunately, for most real-world settings this is not true and other methods like optimization-based control work better and more reliable than deep RL.

However, an advantage of the RL approach is that, theoretically, it is applicable to any kind of control problem, even if very little information about the problem is available. This makes it a very flexible, but also error-prone method, because no prior information about the problem is used or can be used. All model-free RL algorithms have to learn about their specific problem setting from scratch. For that reason they naturally suffer from the exploration-exploitation dilemma (explained in Section 2.2.1) and have a very poor sample efficiency.

Model-based reinforcement learning could help to improve the sample efficiency but the problem with this methodology seems to be that learning good models and a good policy at the same time is challenging. However, lot of research is still done in this area and recent approaches come close to the performance of the current state-of-the-art model-free deep RL algorithms [39]. Some approaches even deliver better results under certain conditions [40].

Another interesting technique for future research projects could be offline reinforcement learning. The idea thereby is to use large amounts of previously collected interaction data to train a policy completely offline, instead of relying on step-wise interaction with a real environment. In problem settings similar to this thesis, this could solve several issues because the learning process could be decoupled from the large and complex simulation model. A good overview of the general concept of offline RL and its current limitations is given in [41]. A huge challenge when learning from offline data is the so called distribution mismatch between any dataset with logged interactions and real actions, because it becomes unclear which reward should be provided if a different action is taken than in the collected data. However, despite these problems, optimistic results could be achieved with offline RL so far [42] and recent research shows that a combination of previously collected demonstration data and online experience seems to be a promising and practical way to learn policies [43].

Bibliography

[1] E. R. Gelso and J. Dahl, "Air-path control of a heavy-duty egr-vgt diesel engine," *IFAC-PapersOnLine*, vol. 49, no. 11, pp. 589–595, 2016, 8th IFAC Symposium on Advances in Automotive Control AAC 2016. DOI: `https://doi.org/10.1016/j.ifacol.2016.08.086`.

[2] B. Kekik and M. Akar, "Model predictive control of diesel engine air path with actuator delays," *IFAC-PapersOnLine*, vol. 52, no. 18, pp. 150–155, 2019, 15th IFAC Workshop on Time Delay Systems TDS 2019. DOI: `https://doi.org/10.1016/j.ifacol.2019.12.222`.

[3] H. Grieshabe and T. Raatz, "Basic principles of the diesel engine," in *Diesel Engine Management: Systems and Components*, K. Reif, Ed. Springer Fachmedien Wiesbaden, 2014, pp. 16–33. DOI: `10.1007/978-3-658-03981-3_3`.

[4] "ACEA report vehicles in use Europe," European Automobile Manufacturers Association (ACEA), Tech. Rep., Jan. 2021. [Online]. Available: `https://www.acea.be/uploads/publications/report-vehicles-in-use-europe-january-2021.pdf`.

[5] J. Ullmann and T. Allgeier, "Cylinder-charge control systems," in *Diesel Engine Management: Systems and Components*, K. Reif, Ed. Springer Fachmedien Wiesbaden, 2014, pp. 46–59. DOI: `10.1007/978-3-658-03981-3_5`.

[6] J. O. Stein, "Minimizing emissions inside of the engine," in *Diesel Engine Management: Systems and Components*, K. Reif, Ed. Springer Fachmedien Wiesbaden, 2014, pp. 178–199. DOI: `10.1007/978-3-658-03981-3_18`.

[7] F. Landhäußer, M. Heinzelmann, A. Michalske, M. L. Susaeta, M. Grosser, J. Feger, L.-M. Fink, W. Gerwing, K. Grabmaier, B. Illg, J. Kurz, R. Mayer, D. Ottenbacher, A. Werner, J. Wiesner, and M. Walther, "Electronic diesel control (EDC)," in *Diesel Engine Management: Systems and Components*, K. Reif, Ed. Springer Fachmedien Wiesbaden, 2014, pp. 220–271. DOI: `10.1007/978-3-658-03981-3_20`.

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb. 2015. DOI: `10.1038/nature14236`.

[9] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan. 2016. DOI: `10.1038/nature16961`.

[10] OpenAI, : C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, *Dota 2 with large scale deep reinforcement learning*, 2019. arXiv: `1912.06680 [cs.LG]`.

[11] K. P. Murphy, *Machine Learning: a Probabilistic Perspective.* MIT Press, 2012, `https://www.cs.ubc.ca/~murphyk/MLbook/`.

[12] F. Posada and A. Bandivadekar, "Global overview of on-board diagnostic (obd) systems for heavy-duty vehicles," *Int. Counc. Clean Transp. http://www. theicct. org/sites/default/files/publications/ICCT___Overview___OBD-HDVs___20150209. pdf*, 2015.

[13] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436–44, May 2015. DOI: `10.1038/nature14539`.

[14] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* MIT Press, 2018, `http://incompleteideas.net/book/the-book.html`.

[15] R. S. Sutton, A. G. Barto, and R. J. Williams, "Reinforcement learning is direct adaptive optimal control," *IEEE Control Systems Magazine*, vol. 12, no. 2, pp. 19–22, 1992. DOI: `10.1109/37.126844`.

[16] C. Watkins and P. Dayan, "Technical note: Q-learning," *Machine Learning*, vol. 8, pp. 279–292, May 1992. DOI: `10.1007/BF00992698`.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, *Playing atari with deep reinforcement learning*, 2013. arXiv: `1312.5602 [cs.LG]`.

[18] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, pp. 229–256, 1992. DOI: `10.1007/BF00992696`.

[19] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ser. ICML'14, Beijing, China: JMLR.org, 2014, I–387–I–395. DOI: `10.5555/3044805.3044850`.

[20] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning.," in *ICLR (Poster)*, 2016. [Online]. Available: `http://arxiv.org/abs/1509.02971`.

[21] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, *Trust region policy optimization*, 2017. arXiv: `1502.05477 [cs.LG]`.

[22] S. Kullback and R. A. Leibler, “On Information and Sufficiency,” *The Annals of Mathematical Statistics*, vol. 22, no. 1, pp. 79–86, 1951. DOI: `10.1214/aoms/1177729694`.

[23] S. Kakade, “A natural policy gradient,” *Advances in Neural Information Processing Systems*, vol. 14, pp. 1531–1538, Jan. 2001.

[24] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, 2017. arXiv: `1707.06347 [cs.LG]`.

[25] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, *Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor*, 2018. arXiv: `1801.01290 [cs.LG]`.

[26] T. Haarnoja, H. Tang, P. Abbeel, and S. Levine, *Reinforcement learning with deep energy-based policies*, 2017. arXiv: `1702.08165 [cs.LG]`.

[27] S. Fujimoto, H. van Hoof, and D. Meger, *Addressing function approximation error in actor-critic methods*, 2018. arXiv: `1802.09477 [cs.AI]`.

[28] R. Hafner and M. Riedmiller, “Reinforcement learning in feedback control: Challenges and benchmarks from technical process control,” *Machine Learning*, vol. 84, pp. 137–169, Jul. 2011. DOI: `10.1007/s10994-011-5235-x`.

[29] X.-S. Wang, Y.-H. Cheng, and S. Wei, “A proposal of adaptive pid controller based on reinforcement learning,” *Journal of China University of Mining and Technology*, vol. 17, no. 1, pp. 40–44, 2007.

[30] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, *Stable baselines*, `https://github.com/hill-a/stable-baselines`, 2018.

[31] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: `https://www.tensorflow.org/`.

[32] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, *Openai gym*, 2016. eprint: `arXiv:1606.01540`.

[33] M. Vecerik, T. Hester, J. Scholz, F. Wang, O. Pietquin, B. Piot, N. Heess, T. Rothörl, T. Lampe, and M. Riedmiller, *Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards*, 2018. arXiv: `1707.08817 [cs.AI]`.

[34] A. Raffin, *RL baselines zoo*, `https://github.com/araffin/rl-baselines-zoo`, 2018.

[35] J. L. Ba, J. R. Kiros, and G. E. Hinton, *Layer normalization*, 2016. arXiv: `1607.06450 [stat.ML]`.

[36] OpenAI, M. Andrychowicz, B. Baker, M. Chociej, R. Jozefowicz, B. McGrew, J. Pachocki, A. Petron, M. Plappert, G. Powell, A. Ray, J. Schneider, S. Sidor, J. Tobin, P. Welinder, L. Weng, and W. Zaremba, *Learning dexterous in-hand manipulation*, 2019. arXiv: `1808.00177 [cs.LG]`.

[37] UNECE, *Global Technical Regulation No.4 (WHDC)*, 1998. [Online]. Available: `https://www.unece.org/fileadmin/DAM/trans/main/wp29/wp29wgs/wp29gen/wp29registry/ECE-TRANS-180a4e.pdf`.

[38] A. Irpan, *Deep reinforcement learning doesn't work yet*, `https://www.alexirpan.com/2018/02/14/rl-hard.html`, 2018.

[39] T. Wang, X. Bao, I. Clavera, J. Hoang, Y. Wen, E. Langlois, S. Zhang, G. Zhang, P. Abbeel, and J. Ba, *Benchmarking model-based reinforcement learning*, 2019. arXiv: `1907.02057 [cs.LG]`.

[40] M. Janner, J. Fu, M. Zhang, and S. Levine, *When to trust your model: Model-based policy optimization*, 2019. arXiv: `1906.08253 [cs.LG]`.

[41] S. Levine, A. Kumar, G. Tucker, and J. Fu, *Offline reinforcement learning: Tutorial, review, and perspectives on open problems*, 2020. arXiv: `2005.01643 [cs.LG]`.

[42] R. Agarwal, D. Schuurmans, and M. Norouzi, *An optimistic perspective on offline reinforcement learning*, 2020. arXiv: `1907.04543 [cs.LG]`.

[43] A. Nair, M. Dalal, A. Gupta, and S. Levine, *Accelerating online reinforcement learning with offline datasets*, 2021. arXiv: `2006.09359 [cs.LG]`.