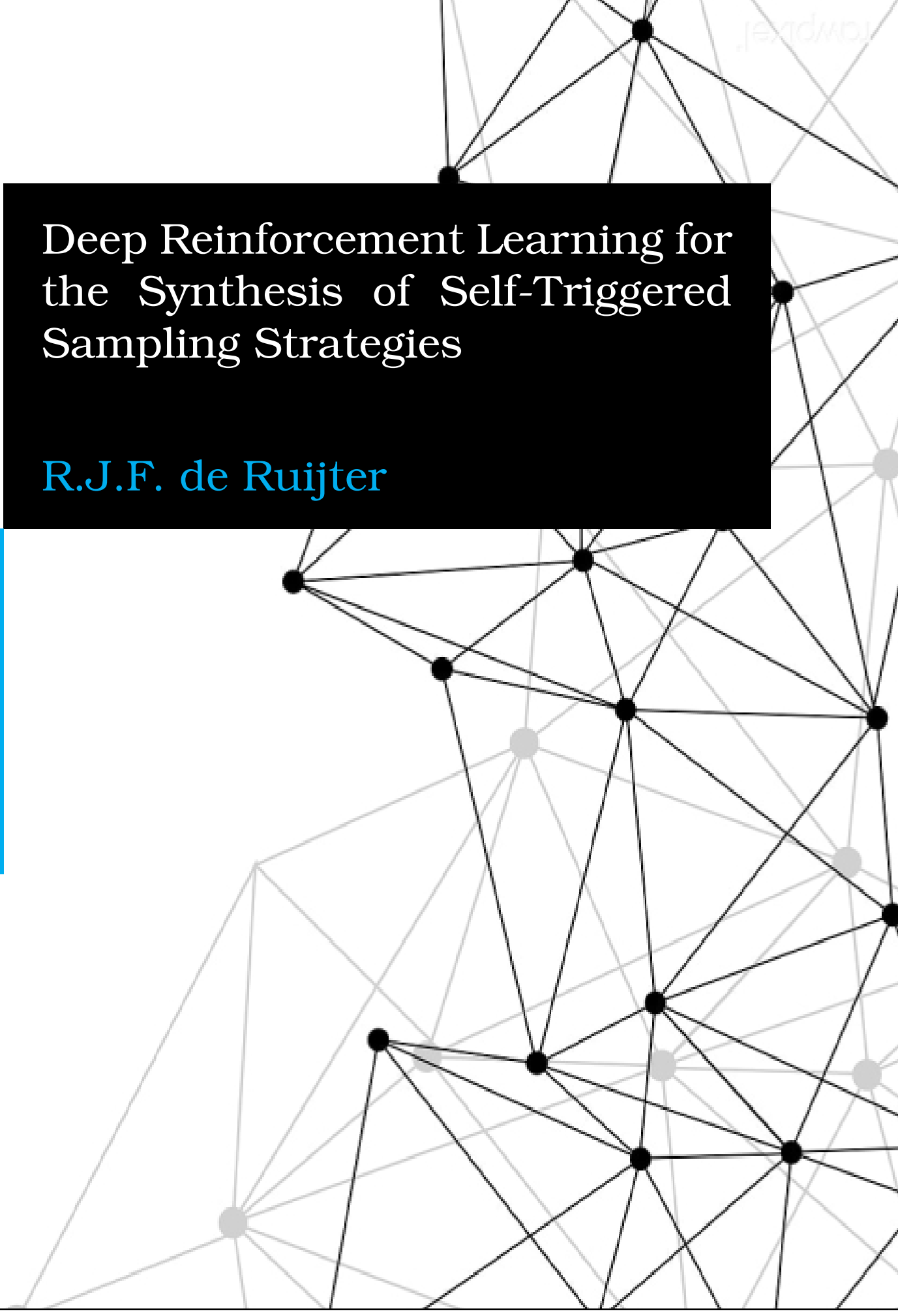
Deep Reinforcement Learning for the Synthesis of Self-Triggered Sampling Strategies

R.J.F. de Ruijter

Master of Science Thesis



TU Delft Delft University of Technology

Delft Center for Systems and Control

Deep Reinforcement Learning for the Synthesis of Self-Triggered Sampling Strategies

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

R.J.F. de Ruijter

March 13, 2022

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Abstract

Control engineering researchers are increasingly embracing data-driven techniques like reinforcement learning for control and optimisation. An example of a case where reinforcement learning could be useful is the synthesis of near-optimal sampling strategies for self-triggered control. Self-triggered control is an aperiodic control method that aims to reduce the number of communications between the controller and sensors in a control loop, by predicting when some triggering condition is met and only transmitting a sample accordingly. Recent research has shown that greedily following the proposed sampling times can result in sub-optimal long-term average inter-sample times. Abstraction-based methods have been able to synthesise sampling strategies that result in better long-term average inter-sample times, by allowing for early sampling and considering the proposed sampling times as deadlines. However, these abstraction-based methods suffer greatly from the curse of dimensionality in the form of combinatorial explosion, which limits their practicality for more complex systems. This thesis proposes a novel deep reinforcement learning tool for finding near-optimal sampling strategies for self-triggered control of LTI systems. The proposed tool is evaluated and compared to a state-of-the-art abstraction-based method. The proposed tool is shown to match the performance of the abstraction-based method for smaller systems, while still achieving good results on more complex systems that prohibit the use of abstraction-based methods.

Table of Contents

List of Figures

List of Tables

Preface and Acknowledgements

First and foremost, I would like to sincerely thank my thesis supervisor dr. ir. Manual Mazo Espinosa for providing me with a research direction and for his guidance during my Master thesis. I really appreciated his laid-back management style combined with his sharp and helpful advice. I also want to express my gratitude to the other members of our research group, who provided me with excellent support and feedback during the weekly meetings. In particular I would like to thank ir. Gabriel de Albuquerque Gleizer for the insightful discussions of his work, which has been the main inspiration for this thesis research. Lastly, I would like to thank my friends, family and girlfriend for being a part of my live and making my student years truly memorable.

Delft, University of Technology — R.J.F. de Ruijter
March 13, 2022

Chapter 1

Introduction

Machine learning is a data-driven form of artificial intelligence in which performance of a system improves automatically through the processing of data. A specific type of machine learning that has recently gained a lot of interest from the academic community is called reinforcement learning (RL). Reinforcement learning involves an agent that learns desired behaviour by interacting with an unknown environment and receiving rewards for its actions, thereby creating its own dataset on-the-fly. Such an RL agent uses trial-and-error to gather experience with the goal of maximising the rewards it obtains by altering its behaviour. Recently, the use of artificial neural networks has been introduced into the RL framework, which has greatly increased the capabilities of these methods. Such "deep reinforcement learning" algorithms have been able to achieve and surpass human performance in complex tasks like playing Atari games. A deep reinforcement learning algorithm called AlphaGo was even able to beat professional players at the extremely complex board game Go.

Control engineering researchers are also embracing these kinds of data-driven techniques. An example of a case where reinforcement learning could be useful is learning optimal sampling strategies for Event-Triggered Control (ETC) and Self-Triggered Control (STC). In traditional control schemes, control updates happen periodically, whereas ETC and STC schemes enable aperiodic updates. In this approach, a sample measurement of the state is only send to the controller when a triggering condition that guarantees some control objectives is met. This allows for more flexible control, which is especially useful when multiple control loops share the same communication network. Since such a communication network has a limited bandwidth, sampling becomes a scarce resource and one might want to minimise its wasteful use. This gives rise to the problem of synthesising a sampling strategy that maximises a control loop's long-term average sampling time, which would result in lower resource usage. Most current methods attempt to achieve this by using some finite-state traffic abstraction of the system, that models the sampling characteristics of a system. These abstractions are in the form of transition systems, which can quickly grow in size with the complexity of the system. This could result in abstractions that exceed computer memory capacity and search times that are too long to be practical, a phenomenon often referred to as "the curse of dimensionality". Data-driven methods like reinforcement learning could present a viable alternative to these abstraction-based methods.

Since an RL agent learns from experience that is generated on-the-fly, it does not need *a priori* knowledge of the entire transition system and will therefore not be hindered by computer memory capacity. The simulating nature of reinforcement learning enables the study of long-term dynamics, which make it an interesting candidate for this application. However, the training of these RL models does require the generation and processing of massive amounts of experience, since the state-action spaces describing these tasks are large. In the case of Go for example, an astonishing 10^{170} different board configurations are possible - more than the number of atoms in the known universe. Training a reinforcement learning agent to solve such a task sequentially on a single processor would take too much time to be of any real use. For this reason, these methods often rely on parallel computing approaches to generate the experience from which the agent can learn. Such an approach can scale with the complexity of the problem by just employing more experience-generating workers in parallel.

The main goal of this thesis research is to implement a tool which uses deep reinforcement learning methods to synthesise near-optimal sampling strategies for LTI self-triggered control systems. The performance of this tool will be compared to that of an abstraction-based method by Gleizer et al. [9]. The main findings of this thesis research are presented in this report. Chapter 2 will provide an overview of key concepts related to ETC/STC and Deep Reinforcement Learning, followed by the problem statement of this thesis research. Chapter 3 includes a description of the proposed tool and experimental setup. Chapter 4 will present and discuss the results of the experiments. Lastly, Chapter 5 contains a conclusion and recommendations for future work. In the Appendices, some additional results comparing different components of the tool can be found, followed by a guide on how to use the proposed tool.

Chapter 2

Preliminaries and Problem Statement

This chapter will provide the reader with the necessary theoretical knowledge to understand the research presented in this thesis report. The first section will focus on aperiodic control, followed by a section on reinforcement learing. Finally, the problem addressed by this thesis research will be formulated in the last section.

2-1 Aperiodic control

Feedback controllers are typically implemented in a periodic fashion. Sensor measurements are received periodically by the controller after which control actions are calculated and applied to the plant using actuators. The field of periodic control is mature and many well-developed theories and tools for analysis and design are available to engineers. Although this still makes periodic control the preferred method for many applications, it has its disadvantages. Some of these disadvantages become apparent in the case of *Networked Control Systems (NCS)*.

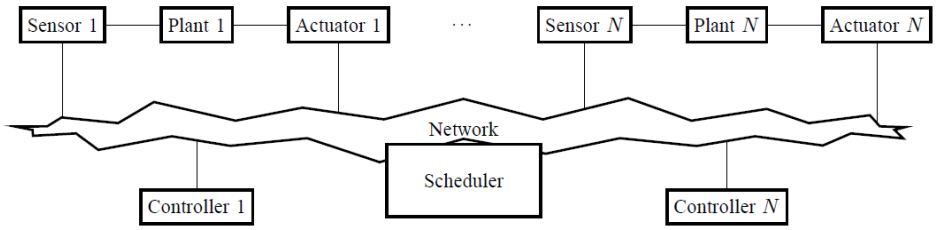


Figure 2-1: Networked Control System containing N control loops [1].

An NCS, as depicted in Figure 2-1, is a collection of distributed control loops in which the communication between sensors, actuators and controllers in every loop is transmitted over a shared digital communication network. Some advantages of this approach include reduced wiring and maintenance costs. These shared communication networks are however bandwidth-limited: they contain finitely many communication channels and can only handle

one signal per channel at any given time. This makes communication a scarce resource and the need for minimization of its wasteful use arises, which could potentially also lead to energy-savings. A closely related problem is the scheduling of different control loops over the same communication channel, while ensuring closed-loop stability of all control loops.

These problems can be solved by allowing for *aperiodic controller updates*. In the periodic control setting, sampling times are equally spaced according to a fixed sampling period. In the aperiodic control setting, the next sampling time is determined dynamically, which results in aperiodic controller updates, as depicted in Figure 2-2. *Event-Triggered Control (ETC)* and *Self-Triggered Control (STC)* are two related methods of dynamically determining this next sampling time. In ETC, the control loop is only triggered when an event, defined by some triggering-condition, occurs. When this triggering-condition is properly designed, it ensures adequate closed-loop control performance, while reducing the amount of communication traffic over the network. Whereas ETC is reactive and requires monitoring of the system in between control tasks, STC could be considered a more pro-active alternative. STC uses a model of the system to predict the necessary next sampling time based on the triggering condition. In this section, some relevant concepts related to these methods will be covered.

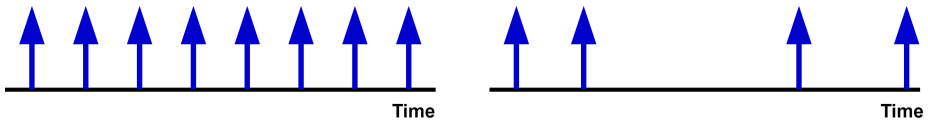


Figure 2-2: Periodic (l) versus aperiodic (r) sampling times.

2-1-1 Fundamentals

Let us consider a feedback-controlled linear time-invariant (LTI) system, described by the following state space model:

$$\dot{x}(t) = Ax(t) + BK\hat{x}(t), \quad x(t) \in \mathbb{R}^n,\ \hat{x}(t) \in \mathbb{R}^n \tag{2-1}$$

where x is the plant's state vector, $\hat{x}$ is the state measurement vector available to the controller and A, B, K are matrices of appropriate dimensions. The design of controller K is outside the scope of this literature survey. The feedback is implemented in a *zero-order sample-and-hold* manner: let $t_k,\ k \in \mathbb{N}_0$ be a sequence of sampling times, then

$$\hat{x}(t) = x(t_k), \quad \forall t \in [t_k, t_{k+1}) \tag{2-2}$$

This means that once the measurement of the state available to the controller $\hat{x}$ is updated, it is kept constant until the next sampling time, resulting in a constant control action being applied to the plant during this time.

Continuous and Periodic ETC

In periodic control, the sequence of sampling times t_k would be equally spaced, depending on a fixed sampling period. In ETC on the other hand, t_k is determined by a *triggering condition*

$\mathcal{C}(x(t), x(t_k)) = 0$. A state update is only send to the controller when the triggering condition is met. The earliest version of ETC required continuous monitoring of the triggering condition, which was later termed Continuous Event-Triggered Control (CETC). The triggering condition is a function of the current state $x(t)$ and the last sampled state that was sent to the controller $x(t_k)$. This results in the triggering times

$$t_0 = 0, \quad t_{k+1} = \inf\{t > t_k \,|\, \mathcal{C}(x(t), \hat{x}(t)) = 0\} \tag{2-3}$$

Note that the resulting *inter-sample time (IST)* $\tau(x(t_k)) := t_{k+1} - t_k$ depends on the last sampled state.

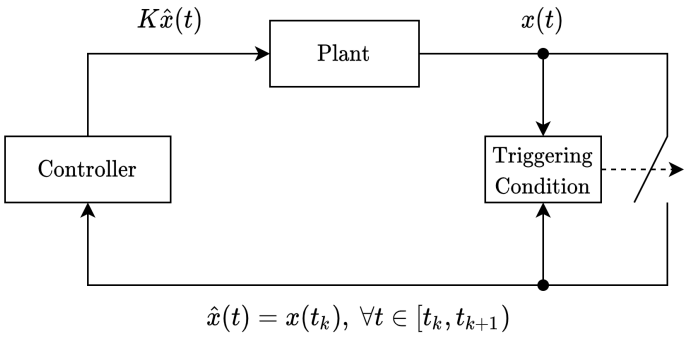


Figure 2-3: ETC system.

In CETC, this triggering condition is continuously monitored, which requires specialised hardware and can result in Zeno behaviour (i.e. an infinite amount of events in a finite time interval), because CETC has no inherent lower bound on the triggering time. In Periodic Event-Triggered Control (PETC) [12] the triggering condition is checked only periodically, with a fixed sampling interval $h > 0$, called the fundamental checking period. This type of ETC attempts to strike a balance between periodic and event-based control and can be easily implemented on standard digital hardware, while inherently excluding Zeno behaviour because of the natural lower bound on the inter-event time (h). In case the triggering condition $\mathcal{C}(x(t), x(t_k)) > 0$ is not met when checked, no state update is send from plant to controller and no new control input commands has to be send from the controller to the plant for at least another sampling interval h. This means no access to the communication network is required during this interval. When the triggering condition is met, it means a control action is required to guarantee control objectives. Formally:

$$t_0 = 0, \quad t_{k+1} = \inf\{kh > t_k,\ k \in \mathbb{N} \,|\, \mathcal{C}(x(t), \hat{x}(t)) \geq 0\} \tag{2-4}$$

The resulting IST $\tau(x(t_k)) := t_{k+1} - t_k$ is again a function of the state. In the PETC case, we can also define a discrete IST $\kappa = \tau/h$. Often a maximum IST naturally emerges form a PETC triggering condition, but to enforce a "heartbeat" of the system a maximimum discrete IST $\bar{k}$ is often set.

Self-Triggered Control

STC as originally proposed by [39] is another aperiodic control method. In STC, instead of continuously or periodically checking the triggering condition, the controller uses the sampled state and a model of the system dynamics to predict the next sampling time $t_{k+1} = t_k + \Gamma(x(t_k))$. The next sample time is the maximum sample time such that the resulting predicted state of the system still meets a triggering condition. Note that this is a conservative prediction. Formally,

$$t_0 = 0, \quad t_{k+1} = \sup\{t \in h\mathbb{N} \mid t > t_k,\ \mathcal{C}(t - t_k, x(t), \hat{x}(t)) \leq 0\} \tag{2-5}$$

Since triggering times can be predicted ahead of time using STC, it allows for more flexibility in scheduling control loops compared to ETC. STC also does not require the use of dedicated hardware to monitor the state. However, the computational cost of STC is higher compared to ETC, and the reliance on predictions instead of actual measurements of the state make it less robust against possible disturbances.

Triggering conditions

Many different variations of triggering conditions can be found in the literature, such as triggering conditions based on the state error [32], the input error [11] or on a Lyapunov function of the system [39], [40], [17]. The goal of most of these triggering conditions is to ensure that the system is stable. A very common way of achieving this is with triggering conditions that guarantee a *monotone decrease of the Lyapunov function* of the system, i.e. triggering conditions that ensure that the derivative of the Lyapunov function is kept negative.

Sampling strategies

Generally speaking, a *sampling strategy* maps an alternating sequence of past states and inter-sample times $r = x_0\tau_0x_1\tau_1...x_k$ to a next inter-sample time τ_k. However, finding such a sampling strategy can be very difficult, and some simplifications can achieve similar outcomes. In the context of STC, [9] defines the following simplifications:

- *state-dependent sampling strategy (SDSS)*: a strategy that maps a sequence of states to the next inter-sample time τ_i: $s : (\mathbb{R}^{n_x})^+ \to \mathbb{R}_+$
- *static SDSS*: a strategy that takes as input *only one* state in order to determine the next inter-sample time τ_i, i.e. a memoryless strategy: $s : \mathbb{R}^{n_x} \to \mathbb{R}_+$

How to synthesise a sampling strategy depends on the sampling goal. One interesting challenge is finding a static SDDS that minimises the amount of sampling, i.e. maximises the average inter-sample times, which will be presented in more detail next.

2-1-2 Challenge: maximizing average inter-sample times

Once we start considering communication as a scarce resource, the question of how to minimize its wasteful use naturally arises. This could be seen as an optimisation problem: sample as little as possible while still ensuring control performance. This goal could be more formally expressed using the *average inter-sample time (AIST)* and *smallest average inter-sample time (SAIST)*:

$$\mathrm{AIST}(x,s) := \liminf_{n\to\infty} \frac{1}{n+1}\sum_{i=0}^{n} \tau_i(x) \qquad \mathrm{SAIST}(s) := \inf_{x\in\mathbb{R}^{n_x}} \mathrm{AIST}(x,s) \tag{2-6}$$

Note the the AIST is the *limit average* inter-sample time, which means it is a fundamentally long-term metric which is not sensitive to short-term transient behaviour. The SAIST could be considered the worst-case AIST that could arise from a strategy s. This means that the SAIST is not longer a function of the state x but only of the strategy s, which makes it a good measure of sampling performance.

Most of the aforementioned triggering conditions were designed with the goal of sampling as late as possible, which is the logical choice when using ETC. However, the predictive nature of STC allows for more flexibility in the determination of the next sampling time. Realize that sampling earlier than the triggering condition proposes can only result in better control performance at the cost of a worse *immediate* IST. When we thus consider the STC-generated sampling time as a sampling *deadline* instead, we gain the possibility of searching for strategies that result in more optimal long-term average inter-sample times. Formally:

$$d(x) = \max\{\tau \in \mathcal{T} \mid \mathcal{C}(\tau, M(\tau)\hat{x}, \hat{x}) \leq 0\}, \tag{2-7}$$

where $\mathcal{T} \in \{h, 2h, \ldots, \bar{k}h\}$ and $M(t) := A_d(t) + B_d(t)K$ is the state transition matrix under held input. Note that this deadline is state-dependent and is calculated at sampling time, i.e. when a new state measurement is send to the controller.

In this setting, simply sampling at the proposed deadline could be considered a *greedy* optimisation approach: this maximises the immediate reward without regard for long-term effects. Such an approach will only result in an optimal solution if the optimisation problem has a strict optimal substructure, which can not be assumed in this case. To illustrate this, consider Figure 2-4. Let's say we need to choose one of the two depicted paths with the goal of maximising long-term rewards, where the numbers represent the reward obtained when visiting that circle. When we only consider immediate rewards, we will choose the top path, although this obviously is not the optimal choice when we take future steps into account. However, making a different choice than the greedy one requires knowledge of future reward dynamics. One way of dealing with this problem relies on the construction of finite abstractions of the state space.

2-1-3 Abstraction-based methods

The abstraction-based method for finding sampling strategies by Gleizer et al. [9] forms the main inspiration for this thesis research. Since the proposed method essentially attempts to mimic the abstraction-based method [9] in a sample-based manner, it is important to first have a good understanding of this abstraction-based method. First the necessary theoretical concepts will be explained.

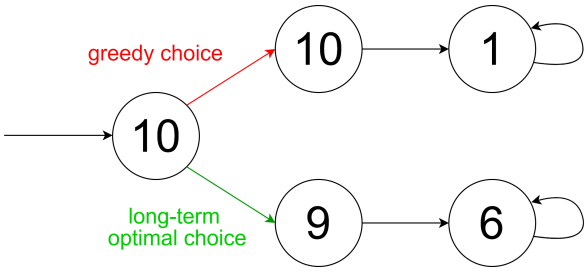


Figure 2-4: Illustrative example of how a greedy approach can result in sub-optimal returns.

Definitions

A *transition system* $\mathcal{S} = (\mathcal{X}, \mathcal{X}_0, \mathcal{U}, \mathcal{E}, \mathcal{Y}, H)$, as described by Tabuada [31], is a way of describing the dynamics of a system, where:

- $\mathcal{X}$ is the finite or infinite set of states;
- $\mathcal{X}_0 \subseteq \mathcal{X}$ is the finite or infinite set of initial states;
- $\mathcal{U}$ is the set of inputs;
- $\mathcal{E} \subseteq \mathcal{X} \times \mathcal{U} \times \mathcal{X}$ is the set of edges or transitions between states;
- $\mathcal{Y}$ is the set of outputs;
- $H(x) : \mathcal{X} \to \mathcal{Y}$ is the output mapping

Transition systems can be finite or infinite, depending on the state set. A transition system is autonomous if $\mathcal{U}$ is an empty set. A *weighted transition system* $\mathcal{S} = (\mathcal{X}, \mathcal{X}_0, \mathcal{U}, \mathcal{E}, \mathcal{Y}, H, \gamma)$ is a transition system with a weight function $\gamma : \mathcal{E} \to \mathbb{Q}$ added to the transitions.

A *finite-state abstraction* is a mapping Φ from the original, possibly infinite state space S to some finite abstract state space $\Phi(S)$. This often aggregates concrete states that have a shared property of interest into one abstracted state for analysis, which is why it is sometimes also referred to as state aggregation or compression. The process of aggregating different states into one abstracted state causes loss of information, but allows for easier analysis of a specific property of interest, for example sampling times.

A *quotient system* $\mathcal{S}_{/\mathcal{R}} = (\mathcal{X}_{/\mathcal{R}}, \mathcal{X}_0, \mathcal{U}, \mathcal{E}_{/\mathcal{R}}, \mathcal{Y}, H_{/\mathcal{R}})$, as described by Tabuada [31], is a transition system that is a simulation of the original transition system $\mathcal{S} = (\mathcal{X}, \mathcal{X}_0, \mathcal{U}, \mathcal{E}, \mathcal{Y}, H)$. This simulation relation essentially means that the notions we are interested in (i.e. the output and possible transitions of the system) are captured by the quotient system while simplifying the description. In order to obtain a quotient system of $\mathcal{S}$, some finite-state abstraction of the state space is constructed and new transitions are determined between these abstract states such that every possible transition present in the original system is also present in the quotient system.

Finite-state abstractions for PETC systems

In order to obtain a finite-state abstraction of a state set, one has to relate states of the concrete system to states of the abstraction. For PETC systems, this can be done in several ways, three of which are described here. Gleizer et al. [8] describe a quotient model that

simulates the output behaviour of a PETC traffic model by aggregating states that trigger at time k into the same quotient state. This is done by first determining the set $\mathcal{K}_k \subseteq \mathbb{R}^{n_x}$ of states that will certainly have triggered by time k:

$$\mathcal{K}_k = \begin{cases} \{x \in \mathbb{R}^{n_x} \mid \mathcal{C}(\cdot) > 0\} & \textit{for } k < \bar{k} \\ \mathbb{R}^{n_x} & \textit{for } k = \bar{k} \end{cases} \tag{2-8}$$

The quotient state $\mathcal{Q}_k$ is then computed for $k \in \{1, 2, ..., \bar{k}\}$ by taking $\mathcal{K}_k$ and removing all states that belong to $\mathcal{K}_j$ with $j < k$:

$$\mathcal{Q}_k = \begin{cases} \mathcal{K}_k \setminus \bigcup_{j=1}^{k-1} \mathcal{Q}_j & \textit{for } k > 1 \\ \mathcal{K}_k & \textit{for } k = 1 \end{cases} \tag{2-9}$$

Building on this time-based partitioning of the state-space, Gleizer et al. [7] define the simulation relation $\mathcal{R}_l \subseteq \mathcal{X} \times \mathcal{Y}^l$. This relation relates states of the concrete PETC system to their respective generated sequences of next l inter-sample times. In this formulation, the abstracted states are thus l-length sequences containing the next l inter-sample times. Formally, $(x, k_1 k_2 ... k_l) \in \mathcal{R}_l$ if and only if:

$$\begin{aligned} x &\in \mathcal{Q}_{k_1}, \\ M(hk_1)x &\in \mathcal{Q}_{k_2}, \\ M(hk_2)M(hk_1)x &\in \mathcal{Q}_{k_3}, \\ &\vdots \\ M(hk_{l-1})...M(hk_1)x &\in \mathcal{Q}_{k_l} \end{aligned} \tag{2-10}$$

Another method of constructing a finite-state abstraction is described by Mazo et al. [18]. They propose a partitioning of the state space using conic covering. This abstraction is based on the idea that for LTI ETC systems, states lying on the same ray crossing the origin result in the same inter-sample time, i.e. are isotropic with regards to inter-sample time: $\tau(x) = \tau(\lambda x), \forall \lambda \neq 0, x \neq 0$. This results in abstractions where states are aggregated into polyhedral cones pointed at the origin, as illustrated in Figure 2-5. When more cones are added, a state-space abstraction with a higher precision is acquired. This method is not directly related to the abstraction-based method by Gleizer et al. [9], which is the main inspiration for this thesis research, but the proposed method does use the idea of isotropic covering (see Section 3-1-2).

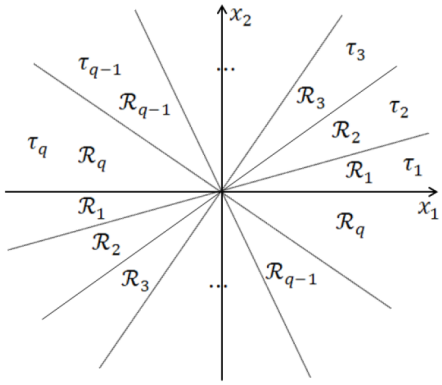


Figure 2-5: Example of a state-space partitioning with polyhedral cones in $\mathbb{R}^2$ [18].

Traffic models

Using the abstracted state set, one can construct PETC traffic models in the form of quotient systems. To do this, transitions should be added between the abstracted states as well as an output map. First, the autonomous case ($\mathcal{U} = \emptyset$) will be considered, where the PETC triggering strategy is followed, i.e. no early triggering is allowed. Such a system can be described as an infinite transition system $\mathcal{S} = (\mathcal{X}, \mathcal{X}_0, \mathcal{U}, \mathcal{E}, \mathcal{Y}, H)$, where

- $\mathcal{X} = \mathcal{X}_0 = \mathbb{R}^n$;
- $\mathcal{U} = \emptyset$;
- $\mathcal{E} = \{(x, x') \in \mathbb{R}^n \times \mathbb{R}^n \mid x' = M(\tau(x))x\}$, with $\tau(x) \in \{h, 2h, ..., \bar{k}h\}$ the IST associated with x;
- $\mathcal{Y} = \{1, 2, ..., \bar{k}\}$;
- $H(x) = \tau(x)/h$

In the case of the l-complete state abstraction from [7], a transition relation called the *domino rule* is applied. This transition relation is a natural consequence of the way the abstracted state set is constructed: a state associated with sequence $k_1k_2...k_l$ must lead to a state whose next first $l-1$ samples are $k_2k_3...k_l$. This means that any abstracted state in $\mathcal{X}_l$ that starts with $k_2k_3...k_l$ is a possible successor to $k_1k_2...k_l$. [7] takes as the output map for each abstracted state the first inter-sample time alone. The resulting l-complete PETC traffic model is the system $\mathcal{S}_l = (\mathcal{X}_l, \mathcal{X}_l, \emptyset, \mathcal{E}_l, \mathcal{Y}, H_l)$, with

- $\mathcal{X}_l = \pi_{\mathcal{R}_l}(\mathcal{X})$;
- $\mathcal{E}_l = \{(k\sigma, \sigma k') \mid k, k' \in \mathcal{Y}, \sigma \in \mathcal{Y}^{l-1}, k\sigma, \sigma k' \in \mathcal{X}_l\}$;
- $\mathcal{Y} = \{1, 2, ..., \bar{k}\}$;
- $H_l(k_1k_2...k_m) = k_1$;

Next, the system which allows for early triggering will be considered. Such a system can be described as a weighted infinite transition system, where the action set contains the discrete sampling times from h until the deadline $d(x)$ ((2-7)). To evaluate the sampling performance of a sampling strategy, a weight related to the sampling time is added to the transitions. Formally: $\mathcal{S} = (\mathcal{X}, \mathcal{X}_0, \mathcal{U}, \mathcal{E}, \mathcal{Y}, H, \gamma)$, where

- $\mathcal{X} = \mathcal{X}_0 = \mathbb{R}^n$;
- $\mathcal{U} = \mathcal{Y} = \{1, 2, ..., \bar{k}\}$;
- $\mathcal{E} = \{(x, u, x') \mid hu \leq d(x) \text{ and } x' = M(hu)x\}$;
- $H(x) = d(x)/h$;
- $\gamma(x, u, x') = hu$

Gleizer et al. [9] apply the same abstraction of the state-space (i.e. $(x, \sigma) \in \mathcal{R}_l$, with $\sigma = k_1 k_2 ... k_l$) to this transition system. This results in the following transition relation: a transition from abstracted state $\mathcal{Q}_\sigma$ to $\mathcal{Q}_{\sigma'}$ exists if for some u respecting the deadline (i.e. $u \leq k_1$), $\exists x \in \mathbb{R}^{n_x}$ such that

$$\begin{aligned} x &\in \mathcal{Q}_\sigma \\ M(hu)x &\in \mathcal{Q}_{\sigma'} \end{aligned} \tag{2-11}$$

As described by Gleizer et al. [9], a traffic model which allows for early triggering can be constructed by adapting the l-complete traffic model from [7]. This so called l-predictive traffic model has the form of a transition system $\mathcal{S}_l = (\mathcal{X}_l, \mathcal{X}_l, \mathcal{U}, \mathcal{E}_l, \mathcal{Y}, H_l, \gamma_l)$, where

- $\mathcal{X}_l = \pi_{\mathcal{R}_l}(\mathcal{X})$;
- $\mathcal{U} = \mathcal{Y} = \{1, 2, ..., \bar{k}\}$;
- $\mathcal{E} = \{(\sigma, u, \sigma') \in \mathcal{X}_l \times \mathcal{U} \times \mathcal{X}_l \mid u \leq \sigma(1), \exists x \in \mathbb{R}^{n_x} : \text{Eq. (2-11) holds}\}$;
- $H_l(\sigma) = k_1$;
- $\gamma(\sigma, u, \sigma') = hu$

Note that when the depth of the abstraction l is small, the amount of non-determinism present in this transition system is large. Increasing the depth of the abstraction results in a more refined division of the state space and a decrease in the amount of non-determinism. Because of this non-determinism in the abstraction, it is not possible to directly extract a sampling strategy from it. In order to obtain a sampling strategy, Gleizer et al. [9] play a *mean-payoff game* on the transition system, where player 0 picks the action and player 1 antagonistically picks the transition. The details of this mean-payoff game are not relevant for this thesis research.

Gleizer et al. [9] have included numerical results for a 2-dimensional system. Figure 2-6 shows a comparison between simulated traces of the sampling strategy obtained using the abstraction-based method proposed in [9] and the greedy method of following the proposed deadline. The strategy obtained using the greedy method depicted in green exhibits bursts of low inter-sample times equal to 0.1. The strategy obtained using the method which allows for early-triggering (depicted in yellow) is able to avoid these fast-triggering regions of the state space by sometimes sampling before the proposed deadline (depicted in red).

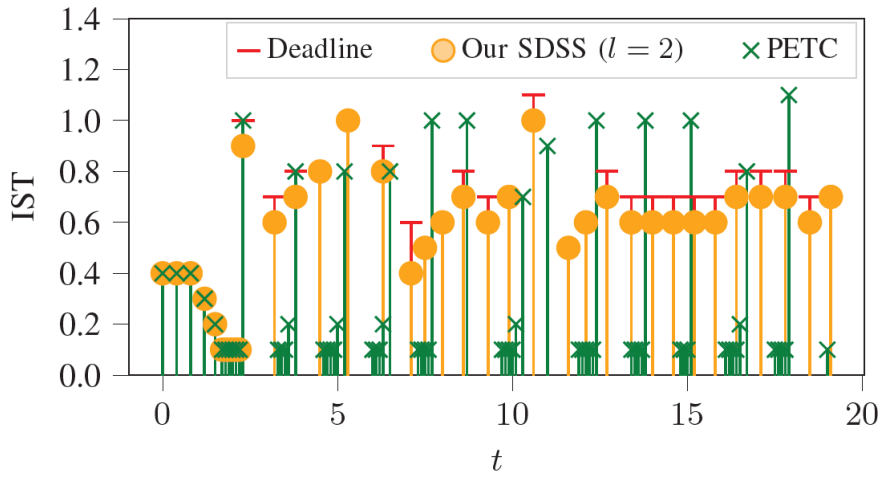


Figure 2-6: Comparison between simulated traces of the SDSS with $l = 2$ from [9] and of the PETC strategy, both with the same initial state.

The resulting ISTs obtained by the greedy PETC method and the early-triggering enabled method are compared in Figure 2-7. It depicts the simulated running average of the ISTs, generated from 10 different initial conditions using both methods. The method which allows for early-triggering clearly outperforms the greedy method in terms of long-term AISTs. The SAIST obtained using the proposed method is calculated to be 0.6, compared to 0.233 for the greedy method.

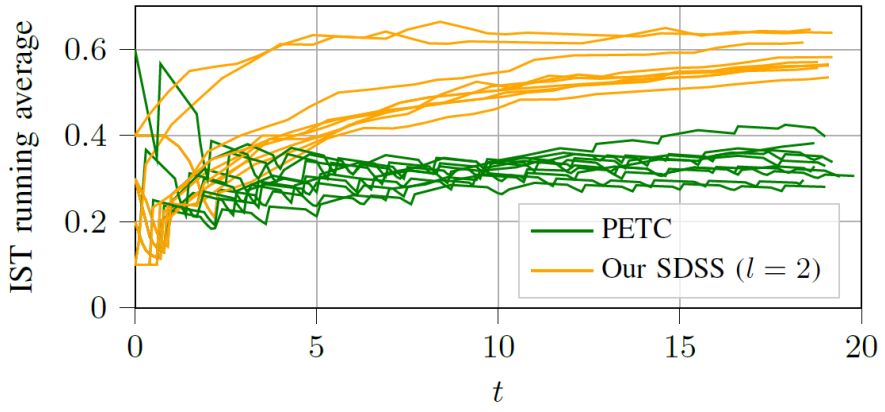


Figure 2-7: Running average of the ISTs generated from 10 different initial conditions under PETC and the near-optimal SDSS using $l = 2$ from [9].

These results clearly show the viability of considering long-term dynamics in order to maximise the long-term inter-sample times. However, the construction of finite state-abstractions to achieve this has some disadvantages. Primarily, the method proposed in [9] suffers from the *curse of dimensionality*, as many abstraction-based methods do. This effect mainly raises its ugly head during the construction of the abstraction. As the depth of the abstraction l increases, the amount of possible transitions grows rapidly, which is also known as combinatorial explosion. This quickly leads to memory errors or infeasible computation times. The extend of these problems increases rapidly with system dimensionality and complexity.

Avoiding the construction of the complete abstraction could be a way to avoid this combinatorial explosion. This leads us to investigate sample-based optimisation methods, since these enable step-by-step exploration of the state space. Visited states are only stored temporarily in order to update the strategy and are then discarded. This means that increasing the look-ahead distance is relatively cheap in terms of memory requirements, which reduces the curse of dimensionality. The iterative nature of these sample-based methods also result in more gradual increase in strategy performance during training time. To illustrate this, consider the illustrative example in Figure 2-8. The construction of a more refined abstraction takes some time, during which no better solutions are obtained. It is only once the new abstraction is completed that a better solution can be extracted from the abstraction, as depicted in Figure 2-8. This means that if the optimisation process is somehow stopped or crashes before the new abstraction is completed, a lot of computing power and time is wasted. However, sample-based optimisation techniques are iterative and every optimisation step slightly improves the solution. This means that the optimisation process can be stopped after every step, with practically no loss of optimality of the solution. In the next section, a popular sample-based optimisation method called reinforcement learning (RL) will be introduced.

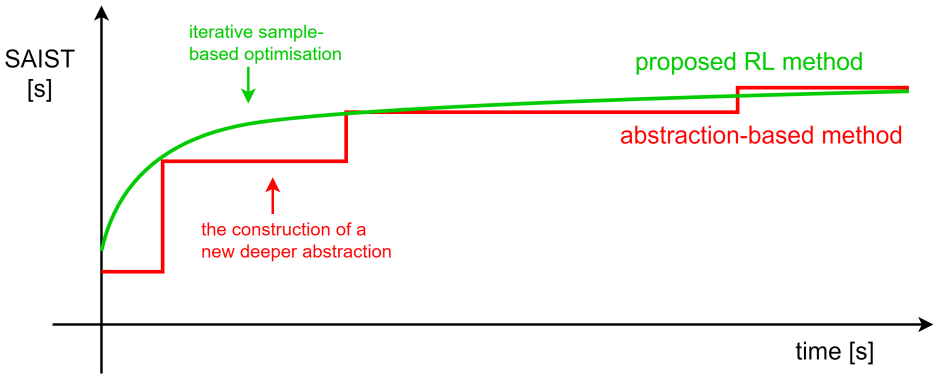


Figure 2-8: An illustrative example comparing hypothetical solutions over time obtained using an abstraction-based method to a sample-based method.

2-2 Reinforcement learning

Reinforcement Learning (RL) is a type of machine learning in which an agent learns by interacting with its environment and receiving a reward for the actions it performs. It mainly has its roots in the fields of *optimal control* and *trial and error learning*. Optimal control is a branch of optimisation, stemming from the 1950s. Generally, it attempts to solve the problem of synthesizing a controller which maximises some desired behaviour of a dynamical system. One of the approaches to solving this problem is called Dynamic Programming (DP), developed by Richard Bellman in the 1950s [2]. Dynamic programming attempts to solve an optimization problem by dividing it into nested smaller sub-problems which can be solved recursively using the *Bellman equation*. Many of the mathematical concepts describing the dynamic programming methods also form the basis of reinforcement learning, like the value function and policy. The main problem with this approach is that it also suffers from "the curse of dimensionality", because its computational complexity grows exponentially with the number of states. The term "trial-and-error learning" goes back as far as the mid-19th century when the term was used by the British ethologist and psychologist Conway Lloyd Morgan [35] to describe observations of animal behaviour. The principle of trial-and-error learning was first formally expressed by American psychologist Edward Thorndike in the early-20th century [34]. He proposed the "law of effect", which states that for a certain situation, responses that result in a rewarding state have their association with that situation strengthened and are thus more likely to occur in the future when the same situation arises again. Computer implementations of trial-and-error learning were first philosophised by Alan Turing in 1948 [36]. In the following years, some research on trial-and-error learning was conducted in different loosely connected fields, but the focus of the machine learning community was mostly on supervised learning techniques. In these years, the distinction between supervised learning and trial-and-error learning was not as well defined, but in the 1970's Harry Klopf observed that the key idea of trial-and-error learning was missing from the machine learning field: the adaptation of behaviour because of the hedonic desire for collection of reward through experience. This is why Klopf is often regarded as the "father of reinforcement learning". Only in the 1980s a new concept called temporal difference (TD) learning was formalised by R.S. Sutton [29], which brought the studies of dynamic programming and trial-and-error learning firmly together in the context of machine learning. This approach hinges on the idea of comparing successive estimates of the value function in order to improve the estimate. It has the advantage of being model-free (like trial-and-error methods) and is implemented in a online, incremental fashion (like DP), while greatly reducing the dimensionality problems that DP suffers from.

2-2-1 Fundamentals

In this section, the key concepts related to RL will be introduced.

Agent-environment interaction

As stated before, a reinforcement learning system consists of an *agent* and an *environment*. The agent is the decision-making part of the system. At every time-step, the agent is presented with the *state* of the environment s_k. The agent then chooses which *action* a_k to perform from the set of available actions in the current state $\mathcal{A}(s_k)$. The agent presents this action to

the environment, which updates its *state* accordingly and presents this newly obtained state s_{k+1} to the agent, together with a numerical *reward* r_{k+1}. This interaction is depicted in Figure 2-9. On tuple of state, action, reward and updated state (s_k, a_k, r_k, s_{k+1}) is one data sample, which is called *experience* in the context of RL.

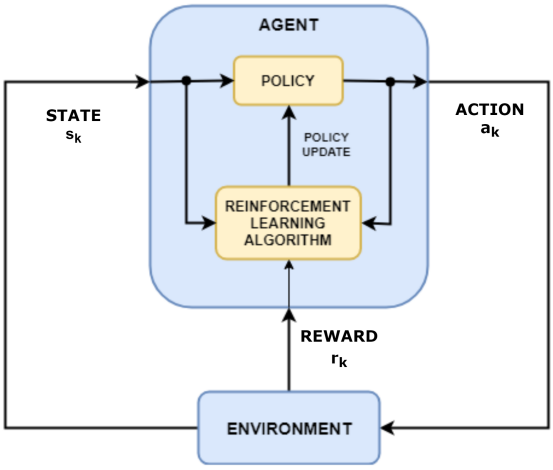


Figure 2-9: Agent-environment interaction [16].

Return and policy

The reward function which determines the reward is part of the environment but can be shaped by the user to favour certain behaviour. The *return* is defined as some accumulation of reward by the agent over time. Most often a discounted summation of future rewards is used:

$$R = \sum_{k=0}^{\infty} \gamma^k r_{k+1} \tag{2-12}$$

The agent follows a decision-making rule called a *policy* π to determine which action to take. A policy can be both deterministic or stochastic. A deterministic policy $\pi_k(s) = a$ simply maps a state to an action, whereas a stochastic policy $\pi_k(a \mid s) = \Pr(a_k = a \mid s_k = s)$ gives the probability of taking an action given a state. The policy is learned in a iterative manner based on the collected experience. There are many different reinforcement learning algorithms available, but all have the same goal: implicitly or explicitly learning the optimal policy.

Markov Decision Processes

In reinforcement learning, the environment is typically assumed to satisfy the *Markov property*. This is true if the current state description s_k contains all relevant information about the system from the past and present. Such a state description is called a *Markov state*. In the context of reinforcement learning this includes the reward signal and can formally be

described as follows:

$$\Pr(r_{k+1} = r, s_{k+1} = s' \mid s_0, a_0, r_1, ..., s_{k-1}, a_{k-1}, r_k, s_k, a_k) = \Pr(r_{k+1} = r, s_{k+1} = s' \mid s_k, a_k) \tag{2-13}$$

The LHS of Eq. 2-13 describes the probability distribution of the dynamics as a function of all past and present states, actions and rewards, whereas he RHS describes the probability distribution of the dynamics as a function of only the present state and action. This means that the state update s_{k+1} and reward signal r_{k+1} depends only on the current state s_k and the chosen action a_k, and not on the "path" that led to this state. This is why the Markov property is also sometimes referred to as "independence of path". It might be that a state is not strictly Markov, but is still a good approximation of a Markov state, i.e. *most* relevant information about the system is retained in the state. RL techniques will still yield results in this case, but more loss of information will result in more loss of performance. A concept very central in RL is called a *Markov Decision Process (MDP)*. Most RL methods assume the environment to be an MDP. An MDP can be described with the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P})$, where $\mathcal{S}$ is the state space; $\mathcal{A}$ is the action space, $\mathcal{P}(s' \mid s, a)$ is a Markovian transition model and $\mathcal{R}(s, a)$ or $\mathcal{R}(s, a, s')$ is a Markovian reward model. Most RL algorithms consider the case were the state and action spaces are finite, called a *finite Markov Decision Process*. The discrete transition dynamics of such a environment are denoted as:

$$p(s', r \mid s, a) = \Pr(r_{k+1} = r, s_{k+1} = s' \mid s_k = s, a_k = a) \tag{2-14}$$

From these dynamics of the finite MDP, all other interesting properties of the environment can be computed, such as the state-transition probabilities $p(s' \mid s, a)$ or the expected rewards for a state-action pair $r(s, a)$ or state-action-next-state triplet $r(s, a, s')$. However, these probabilities are often not known to the agent.

Value functions

The goal of reinforcement learning is to tune the agent's policy π such that following the policy maximises the agent's *expected return*. A useful related concept is called the *value function*, which gives some measure of "goodness" to a state or state-action pair under a policy π. We can define the *state-value function under policy* π as the expected return of following policy π from state s:

$$V^\pi(s) = \mathbb{E}\Big[R \;\Big|\; s_k = s, \pi\Big] = \mathbb{E}\Big[\sum_{k=0}^{\infty} \gamma^k r_{k+1} \;\Big|\; s_k = s, \pi\Big] \tag{2-15}$$

Alternatively, we define the *state-action value function under policy* π as the expected return of taking action a from state s and subsequently following policy π:

$$Q^\pi(s, a) = \mathbb{E}\Big[R \;\Big|\; s_k = s, a_k = a, \pi\Big] = \mathbb{E}\Big[\sum_{k=0}^{\infty} \gamma^k r_{k+1} \;\Big|\; s_k = s, a_k = a, \pi\Big] \tag{2-16}$$

For readability purposes, we will from this point on simply refer to the state value function $V^\pi(s)$ as the *value-function* and to the state-action value function $Q^\pi(s, a)$ as the *Q-function* (as is common in the literature).

As mentioned before, the goal of a reinforcement learning algorithm is to find the *optimal policy* π^*. A policy is optimal if the expected return when following this policy is greater or equal to the expected return when following any other policy: $V^{\pi^*}(s) \geq V^{\pi}(s), \forall s, \forall \pi$. As such, there could be more than one optimal policy, but all optimal policies share the same value-function. Applying this optimal policy to the value-function gives the *optimal value function,* V^*.

$$V^*(s) = \max_{\pi} V^{\pi}(s) = \mathbb{E}\Big[\sum_{k=0}^{\infty} \gamma^k r_{k+1} \;\Big|\; s_k = s, \pi^*\Big] \tag{2-17}$$

Applying the optimal policy to the Q-function gives the *optimal Q-function,* Q^*:

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) = \mathbb{E}\Big[\sum_{k=0}^{\infty} \gamma^k r_{k+1} \;\Big|\; s_k = s, a_k = a, \pi^*\Big] \tag{2-18}$$

Bellman expectation and optimality equations

We can divide the value-function in Eq. 2-15 into two parts: a part capturing the value of the current state, and a part capturing the expected value of possible successive states. This gives rise to a recursive relation called the *Bellman expectation equation*, or simply *Bellman equation*:

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}\Big[\sum_{k=0}^{\infty} \gamma^k r_{k+1} \;\Big|\; s_k = s, \pi\Big] \\ &= \mathbb{E}\Big[r_{k+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{k+2} \;\Big|\; s_k = s, \pi\Big] \\ &= \mathbb{E}\Big[r_{k+1} + \gamma V^{\pi}(s') \;\Big|\; s_k = s, \pi\Big] \end{aligned} \tag{2-19}$$

An import observation is the relation between the optimal value-function and optimal Q-function:

$$V^*(s) = \max_{a \in A(s)} Q^*(s,a) \tag{2-20}$$

When we apply the same method for finding the Bellman equation to this relation, we obtain the Bellman equation for $V^*(s)$ called the *Bellman optimality equation* for the value-function:

$$\begin{aligned} V^*(s) &= \max_{a \in A(s)} Q^*(s,a) \\ &= \max_{a} \mathbb{E}\Big[\sum_{k=0}^{\infty} \gamma^k r_{k+1} \;\Big|\; s_k = s, a_k = a, \pi^*\Big] \\ &= \max_{a} \mathbb{E}\Big[r_{k+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{k+2} \;\Big|\; s_k = s, a_k = a, \pi^*\Big] \\ &= \max_{a} \mathbb{E}\Big[r_{k+1} + \gamma V^*(s') \;\Big|\; s_k = s, a_k = a, \pi^*\Big] \end{aligned} \tag{2-21}$$

Replacing $V^*(s)$ and $V^*(s')$ with Eq. 2-20 gives the Bellman optimality equation for the Q-function:

$$Q^*(s,a) = \mathbb{E}\Big[r_{k+1} + \gamma \max_{a'} Q^*(s',a') \;\Big|\; s_k = s, a_k = a, \pi^*\Big] \tag{2-22}$$

These recursive relations form the basis for many reinforcement learning methods.

Dynamic programming, Monte Carlo simulations and Temporal-difference learning

Reinforcement learning combines aspects of two optimisation methods: *dynamic programming (DP)* and *Monte Carlo simulations*. The goal of dynamic programming is to solve Bellman equations in order to find an optimal policy. This method requires full knowledge of the MDP, including the state transition model $\mathcal{P}(s' \mid s, a)$. DP alternates between evaluation and improvement of the current policy in order to iteratively find an optimal policy, a process called Generalize Policy Iteration (GPI) (Figure 2-10). DP methods make use of bootstrapping: the estimate of the value of one state $V_{i+1}(s_k)$ is updated based on the current estimate of the value of a successor state $V_i(s_{k+1})$. This results in the following update rule:

$$V_{i+1}(s_k) = \arg\max_a \mathbb{E}\Big[r_{k+1} + \gamma V_i(s_{k+1}) \;\Big|\; s_k = s, \pi_k\Big] \tag{2-23}$$

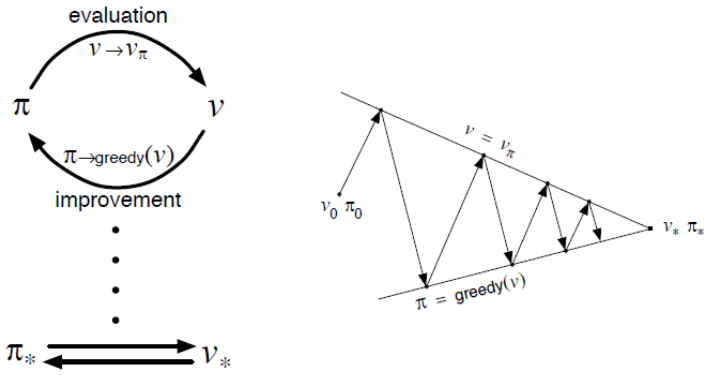


Figure 2-10: Generalised Policy Iteration [30].

Monte Carlo methods take an alternate approach to finding optimal policies. Monte Carlo methods don't require full knowledge of the MDP dynamics $p(s', r \mid s, a)$, but *learn* optimal policies from sample sequences of states, actions and rewards obtained by the agent. Monte Carlo methods *only* learn from complete *sample episodes*, i.e. trajectories that naturally terminate eventually, so are only defined for episodic tasks, i.e. tasks that contain a terminal state. Monte Carlo methods also follow the GPI methodology, but the updated value for each state only depends on the previous value for that state and the actual observed sequence of rewards, so Monte Carlo methods do not use bootstrapping. Note how in Eq. (2-24), the updated value $V_{i+1}(s_k)$ is a weighted sum of the previous value $V_i(s_k)$ and the actual observed rewards G_k, weighted by the *learning rate* α:

$$V_{i+1}(s_k) = (1-\alpha)V_i(s_k) + \alpha G_k = V_i(s_k) + \alpha\Big[G_k - V_i(s_k)\Big] \tag{2-24}$$

One of the most central ideas in reinforcement learning is *TD*. It combines the bootstrapping from dynamic programming with the sampling-based learning of Monte Carlo simulations, which results in the following update rule:

$$V_{i+1}(s_k) = V_i(s_k) + \alpha\Big[r_{k+1} + \gamma V_i(s_{k+1}) - V_i(s_k)\Big] \tag{2-25}$$

This way of updating allows online implementation of the value-function prediction, since learning is done *during* the episode, whereas this is not possible using Monte Carlo methods. In fact, TD learning is more generally applicable, since it does not require episodic MDPs. The part between brackets in Eq. 2-25 is called the *temporal difference error* δ. Classic RL methods like SARSA and Q-learning revolve around this idea, but it also appears in modern methods, as will be discussed in Section 2-2-3.

2-2-2 Deep Reinforcement Learning

The reinforcement learning framework presented in the previous sections is clearly a very powerful framework for tackling many different kinds of optimal control problems. However, all methods discussed so far have used a tabular representation of the value-functions and Q-functions, i.e. a table containing distinct values for states or state-action pairs. This limits the possibilities of such algorithms when the state or action spaces are continuous or high-dimensional. Luckily, it is possible to use function approximation techniques such as *artificial neural networks (ANNs)* to parameterise the value-function, Q-function and policy. The idea of adding techniques from deep learning to the powerful reinforcement learning framework gave rise to the fast-moving field of deep reinforcement learning (DRL). Recently, such DRL algorithms have been able to achieve and surpass human performance in complex tasks like playing Atari games. A DRL algorithm called AlphaGo was even able to beat professional players at the extremely complex board game Go. The training of these DRL models requires the generation and processing of massive amounts of experience, since the state-action spaces describing these tasks and the amount of parameters describing the model are very large. In the case of Go for example, an astonishing 10^{170} different board configurations are possible - more than the number of atoms in the known universe. Training a reinforcement learning agent to solve such a task sequentially on a single processor would take too much time to be of any real use. For this reason, researchers turned to parallel computing approaches. These approaches typically use multiple copies of the environment in parallel (called *workers*) that generate experience from which the agent can learn. Such an approach can scale with the complexity of the problem by just employing more experience-generating workers in parallel, thus reducing the training time. The incorporation of ANN's and parallel workers into the RL framework is depicted in Figure 2-11.

Artificial neural networks

An artificial neural network (ANN) is a computational model inspired by biological neural networks in the human brain. The brain consists of "computational units" called neurons, which are connected in a network by synapses. In ANNs, the basic building block is called a *perceptron*, as depicted in Figure 2-12. The input to a perceptron is a vector $\boldsymbol{x} = [x_1, ...x_n]$. First, the perceptron takes a weighted sum of these input values, and adds a *bias*. This

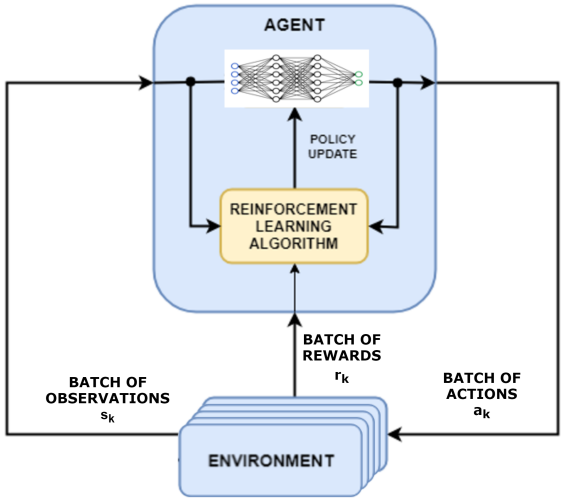


Figure 2-11: Agent-environment interaction, with a neural network as policy approximation and parallel workers.

weighted sum is then fed to an *activation function*, which introduces non-linearity to the perceptron. Many different activation functions are available, like the *ReLu*, *tanh*, and *sigmoid* functions. The output of this activation function is also the output of the perceptron.

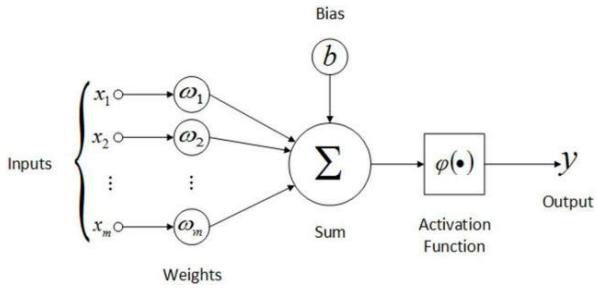


Figure 2-12: A perceptron [10].

If more than one perceptron is joined in a layered fashion, this results in an multi-layer perceptron (MLP), as depicted in Figure 2-13. Usually, layers have *dense connections*, which means that the output of every unit in a layer is fed as an input to every unit in the next layer:

$$\boldsymbol{x}_l = g(\boldsymbol{W}_l^T \boldsymbol{x}_{l-1} + \boldsymbol{b}_l) \tag{2-26}$$

where $g(\cdot)$ is the activation function and $\boldsymbol{W}_l$ and $\boldsymbol{b}_l$ are the weight matrix and bias vector of layer l. Every neural network consists of an input layer and an output layer, with some amount of hidden layers in between. The amount of layers is called the depth of the neural network, and the amount of units in one layer is called the width of the layer, which can both be set to obtain different neural network architectures.

The weights and biases are the parameters that need to be learned, with the goal of minimising or maximising an *objective*, for example the mean squared error between the outputs of the neural network and desired values. This is done using the *backpropagation algorithm*, which essentially uses the chain rule to compute the gradient of the objective function with respect

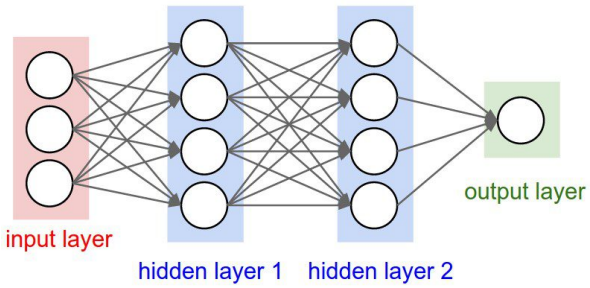


Figure 2-13: An MLP with 3 inputs, 1 output, 2 hidden layers, and dense connections [37].

to the individual weights of the network. This gradient is used to update the weights and biases through some gradient method like stochastic gradient descent (SGD) or Adam. A forward pass of a neural network is called *neural network inference*.

Neural networks can be used to approximate different functions in deep reinforcement learning, as depicted in Figure 2-14. The Q-function can be approximated by a network that takes as an input the state of the environment, and outputs the Q-values for all possible actions. The value-function can be approximated by a network that also takes as an input the state of the environment and outputs the value (i.e. the expected sum of rewards) associated with that state. A stochastic policy-function can be approximated by a network that also takes as an input the state of the environment and outputs the probabilities of taking every action based on that state.

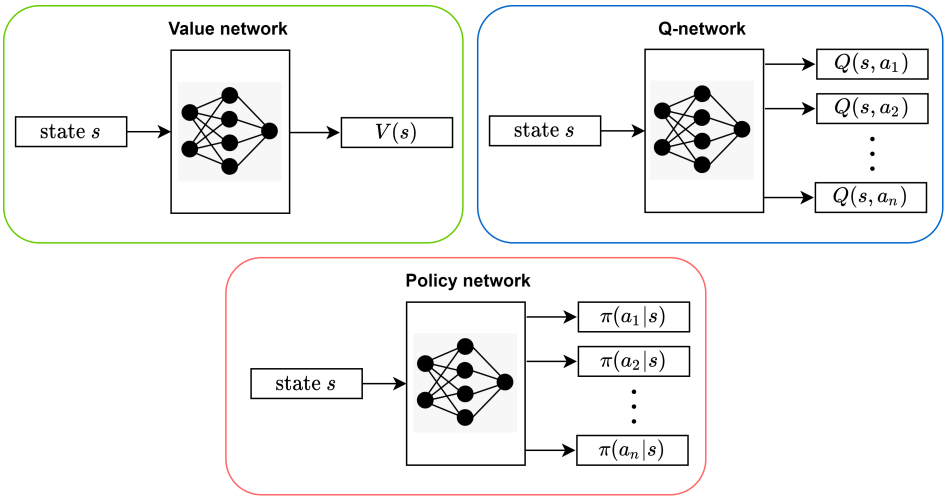


Figure 2-14: Schematic representation of a value network, Q-network and policy network.

In the context of DRL, *logits* are the un-normalised predictions of a policy network, which can be any real number ranging from $[-\infty, +\infty]$. In order to transform these logits to actual probabilities, they are often fed to a final activation layer containing the *softmax* function, which normalises the logits:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{2-27}$$

Q-learning versus Policy Gradient methods

When categorising RL methods, the main point of difference is in *what exactly* is learned. All RL methods have the objective to learn an optimal policy, but the way that this is achieved can differ. Deep Q-learning approaches attempt to *indirectly* optimise the policy through the learning of the Q-function, approximated by the Q-network. Policy Gradient (PG) methods are more principled: they seek to directly optimise in the policy space. These methods require a stochastic policy approximated by a neural network $\pi_\theta(a \mid s)$ and directly optimise the policy by applying gradient ascent on its parameters, such that the probability of taking actions associated with high returns is increased. Both methods have their strengths and weaknesses, and which method is preferred is case-dependent. Deep Q-learning methods are often much more sample efficient than PG methods because data can be reused, whereas PG methods require the collection of new samples for every gradient step. However, Deep Q-learning methods can result in poor convergence and instability and are in general sensitive to hyperparameter settings. The simplicity, versatility and stability of modern PG methods often make it the preferred choice when sample efficiency is not of the utmost importance. The next section will explain Policy Gradient methods and one of its most successful variants, called Proximal Policy Optimisation.

2-2-3 A closer look at Policy Gradient methods

The objective of PG methods, like all RL methods, is to maximize the future expected return obtained by following a policy. Formally, the goal is to maximise the objective function

$$J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[G(\tau)] \tag{2-28}$$

where τ is a trajectory obtained by following the policy π_θ and $G(\tau)$ is the return obtained by following that trajectory. This is achieved by repeatedly calculating the gradient of the objective function and applying gradient ascent on the parameters of the policy network:

$$\theta_{k+1} = \theta_k + \alpha\nabla_\theta J(\pi_\theta) \tag{2-29}$$

The *Policy Gradient Theorem* as presented by [30] shows that the gradient of the objective function can be written as:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}\Big[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\Phi_t\Big] \tag{2-30}$$

where Φ_t can be one of multiple terms related to the sum of rewards (see [25] for an extensive review). One of the most used choices for Φ_t is the *advantage function* $A^\pi(s_t, a_t) := Q^\pi(s_t, a_t) - V^\pi(s_t)$, because it greatly reduces the variance in gradient calculation. The advantage function $A^\pi(s_t, a_t)$ gives a measure of how much better or worse an action a is compared to the policy's default behaviour. Using the advantage function in Eq. (2-30) means the objective J^{PG} will increase if the action a becomes more likely, but only if the advantage is positive. If the advantage is negative, the objective J^{PG} will increase if the action a becomes less likely, i.e. if $\pi_\theta(a|s)$ decreases.

In practice of course, the advantage function is not known and must be estimated. The advantage function estimate $\hat{A}_t(s, a)$ can be calculated in different ways, but a widely-used

approach called Generalised Advantage Estimation (GAE) uses the value function estimate and observed rewards from a trajectory:

$$\hat{A}_t(s,a) = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + (\gamma\lambda)^{T-t+1)}\delta_{T-1} \tag{2-31}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the temporal difference error. This is one of the reasons why many PG methods also maintain a value function network $V_\phi(s)$ to estimate the value function, which is updated concurrently with the policy network. A method which maintains both a value-function and policy-function is called an *Actor-Critic* method, where 'actor' refers to the learned policy (since this is used to select actions), and 'critic' refers to the learned value function (since this is used to evaluate the performed actions). The simplest and most widely used method for learning $V_\phi(s)$ is regression on the mean-squared-error between the value function estimate and the observed return from a trajectory:

$$\phi_k = \arg\min_\phi \hat{\mathbb{E}}\Big[\sum_{t=0}^{T}\left(V_\phi(s_t) - G_t\right)^2\Big] \tag{2-32}$$

Many different methods have been proposed to handle the problems associated with PG methods (i.e. poor sample efficiency and learning instability due to large trajectory variance). In particular, one family of methods called Proximal Policy Optimisation (PPO) has demonstrated state-of-the-art performance on many RL benchmark tasks, while striking a balance between sample efficiency, simplicity, and wall-clock time.

Proximal Policy Optimisation

The main motivation behind the family of methods called *Proximal Policy Optimisation (PPO)* is to use generated experience as efficiently as possible without causing instability and performance collapse. This is achieved by limiting the difference between the action probabilities of the old policy $\pi_{\theta_{old}}(a_t \mid s_t)$ and the updated policy $\pi_\theta(a_t \mid s_t)$. The method is scalable (i.e. it allows for parallelisation), has improved sample efficiency over previous PG methods, and displays robust performance over a variety of tasks. This section will give a brief overview of this method.

PPO attempts to maximise an objective function which includes the estimated advantage function:

$$J^{PG}(\pi_\theta) = \hat{\mathbb{E}}\Big[\sum_{t=0}^{T}\log\pi_\theta(a_t \mid s_t)\hat{A}_t(s,a)\Big] \tag{2-33}$$

$\hat{A}_t(s,a)$ can be estimated from the collected trajectory in a number of ways, like mentioned in the previous section. The expression $\hat{\mathbb{E}}[\cdot]$ indicates that the expectation is estimated using the empirical average over the batch of collected samples. The policy is updated by performing multiple steps of gradient ascent on the parameters of the policy network using the gradient calculated from this batch of collected samples. However, applying too many steps of gradient updates using the same gradient, increases the risk of stepping too far away from the policy from which that gradient was computed, thus decreasing the validity of the computed gradient. On the other hand, taking too little steps using the same gradient means the collected trajectory data is not used optimally. In normal policy gradient, old and new policies are close

in *parameter space* (due to the use of gradient descent), but small differences in parameter space can potentially have large differences in policy performance. PPO uses a similar trick to Trust Region Policy Optimisation (TRPO) [24], namely limiting the difference between the old policy and the new policy in *policy space*, i.e. the resulting action probabilities cannot differ too much. This will limit the magnitude of changes to the actual metric of interest, namely the action probabilities, and thus the resulting performance of the policy. This is done by borrowing the surrogate loss from TRPO, which introduces the *probability ratio* ρ:

$$J^{CPI}(\pi_\theta) = \hat{\mathbb{E}}\Big[\sum_{t=0}^{T} \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} \hat{A}_t\Big] = \hat{\mathbb{E}}\Big[\sum_{t=0}^{T} \rho_t(\theta)\hat{A}_t\Big] \tag{2-34}$$

If this objective was used in an unconstrained manner, maximisation would result in extremely large policy updates. Whereas TRPO solves this by using a complex second-order optimisation method based on the KL-divergence between the old and new policies, PPO introduces a simple clipped version of Eq. (2-34):

$$J^{CLIP}(\pi_\theta) = \hat{\mathbb{E}}\Big[\sum_{t=0}^{T} \min\left(\rho_t(\theta)\hat{A}_t,\ \text{clip}(\rho_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\Big] \tag{2-35}$$

As depicted in Figure 2-15, this modification of the objective removes the incentive for moving the probability ratio ρ outside of the interval $1+\epsilon$ or $1-\epsilon$, depending on whether the advantage is positive or negative respectively. This ensures that a good action will only become moderately more likely (depending on the choice of ϵ), and a bad action will only become moderately less likely. This results in more monotone and stable policy improvements.

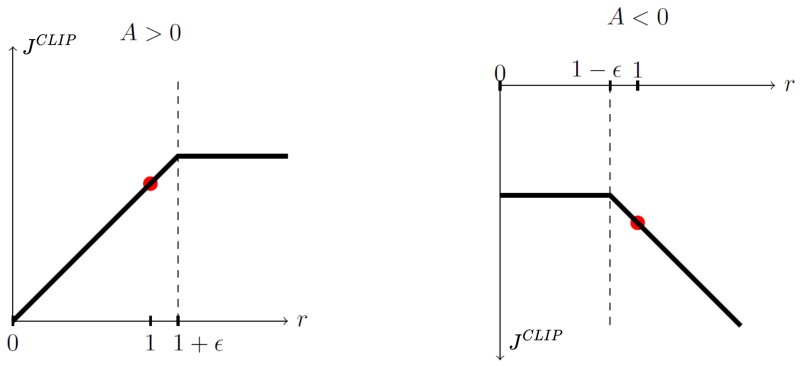


Figure 2-15: Plots showing a single timestep of the surrogate objective J^{CLIP} as a function of the probability ratio ρ, for positive advantages (l) and negative advantages (r). Note that J^{CLIP} sums many of these terms. The starting point of the optimisation is indicated by the red dots. [26].

The PPO implementation from the original paper further augments the surrogate objective by including a value-function error term and an entropy bonus. The value-function term is needed when a neural network architecture that shares parameters between the policy and value function is used. The entropy bonus term should help to ensure sufficient exploration as suggested by earlier work, e.g. [19]. Both terms have their own weighting factor, which results in the following surrogate objective:

$$J^{CLIP+VF+S}(\pi_\theta) = \hat{\mathbb{E}}\Big[J^{CLIP}(\pi_\theta) - c_1 J^{VF} + c_2 S[\pi_\theta](s_t)\Big] \tag{2-36}$$

Note that the term "PPO" formally refers to a family of RL methods that use the surrogate objective (2-35). This surrogate loss can be used in different policy gradient algorithms. In the original PPO paper [26], an Actor-Critic style algorithm was used. In this implementation, each of N parallel workers runs the policy $\pi_{\theta_{old}}$ for T timesteps. Such a run is called a *rollout*. The NT collected rollout samples are used to calculate advantage estimates $\hat{A}_t$ for each timestep and to construct the surrogate objective, which is then optimised using minibatch SGD or Adam for K epochs. This use of a rollout buffer speeds up and stabilises the learning process by decreasing the variance between gradient updates, because the same policy is ran for NT timesteps before calculating gradients on a larger batch of samples. Figure 2-16 describes this algorithm using pseudo-code.

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Figure 2-16: Pseudo-code description of the actor-critic style PPO implementation from [26].

2-3 Problem statement

The main goal of the research presented in this thesis report is to develop a tool which employs deep reinforcement learning techniques to find static state-dependent sampling policies for LTI self-triggered control systems that provides near-maximal average inter-sample time while respecting given control performance constraints. This tool has to be scalable through the use of some parallelisation scheme. More formally, the goals can be formulated as follows

Main goal Implement a deep reinforcement learning tool which outputs a static state-dependent sampling policy for LTI self-triggered control systems. The objective function to be maximised is

$$J(\pi_\theta) = \liminf_{n\to\infty} \frac{1}{n+1} \sum_{i=0}^{n} \tau_i(x, \pi_\theta) \tag{2-37}$$

where $\tau_i(x, \pi_\theta)$ is the inter-sample time from state x suggested by the learned policy π_θ. To ensure that the resulting policy is safe, the choice of possible inter-sample times is constrained by the inter-sample time suggested by the STC system, which has to be respected as a sampling deadline (as described in Section 2-1-2).

Requirements

1. This tool has to be scalable in terms of data generation and complexity of the policy function, for use with more complex and higher-dimensional systems.
2. This tool should allow the user to set multiple different stopping criteria.
3. The user should be able to monitor the sampling performance during the training process.
4. The user should be able to stop the learning process early, and still obtain the most optimal policy found so far.
5. The user should be able to add a measure of trajectory smoothness $\omega(x, \pi_\theta)$ to the objective function:

$$J(\pi_\theta) = \liminf_{n\to\infty} \frac{1}{n+1} \sum_{i=0}^{n} \beta\omega_i(x, \pi_\theta) + (1-\beta)\tau_i(x, \pi_\theta) \tag{2-38}$$

Evaluation

1. Evaluate performance both in terms of sampling performance and wall-clock times of the proposed RL method to that of the abstraction-based method as proposed by [9], using multiple different systems.
2. Perform a study on the relation between average inter-sample times and trajectory smoothness.

Chapter 3

Methodology

This chapter will discuss the formulation of the environment and RL agent, as well as the experimental setup used to obtain the results presented in the next chapter.

3-1 Environment

For a problem to be solved using a reinforcement learning approach, it needs to be formulated as an RL environment. Formulating this in computer code is standardised by the OpenAI Gym framework, which is widely used in industry and the academic world. In the Gym framework, the environment is constructed as a class which requires; a reset function, which starts a new episode and returns an observation; and a step function, which requires an action as input and returns a tuple containing a new observation, reward, a customisable info dictionary and a boolean indicating the termination of an episode.

The user can add different methods and attributes to the environment class in order to formulate their problem. The Gym framework also allows for the addition of wrappers, which add functionality or overwrite the methods of the base environment class. The following code example shows a simple simulation of a OpenAI environment for 20 episodes with 100 steps per episode, where random actions are picked every step:

```
1  import ExampleEnv
2
3  env = ExampleEnv()
4
5  for i_episode in range(20):
6      observation = env.reset()
7      for t in range(100):
8          action = env.action_space.sample()
9          observation, reward, done, info = env.step(action)
10         if done:
11             print("Episode finished after {} timesteps".format(t+1))
12             break
13 env.close()
```

3-1-1 Base environment

The base environment (which defines the problem) used in training requires a discrete-time model of the plant and controller (A_d, B_d, K); the triggering condition; and values for the minimum and maximum triggering time k_{min}, k_{max} and fundamental checking periode h. The step function calculates the state update under held control input, based on the current state of the system and the action chosen by the agent, which corresponds to the discrete sampling time k:

$$x(t+1) = (A_d^k + B_d K \sum_{i=0}^{k-1} A_d^i)x(t) \tag{3-1}$$

The step function also returns the reward, which is simply the inter-sample time hk. The step function then returns the updated state and reward to the agent. The reset function sets the state to a random point within a bounded region. The deadline function calculates the sampling deadline $d(x)$ based on the current state of the plant and the supplied triggering condition.

3-1-2 Environment wrappers

In order to aid the learning process, some wrappers were introduced to the base environment. These wrappers reformulate the basic problem to make it easier to learn for the agent.

- **Episodic wrapper:** since the base problem is not an episodic problem (i.e. it contains no terminal states), episodes have to be ended artificially by setting a maximum episode length. This allows for more efficient exploration of the state space, since after every episode the system is reset to a random state.

- **Action-penalisation wrapper:** this wrappers specifies one possible method of using the sampling deadline during the learning process. If the agent selects a sampling time that is beyond the deadline, the step is not taken (i.e. the state remains the same) and a negative reward (i.e. penalty) is returned. This should allow the agent to quickly learn to not exceed the deadlines. Figure 3-1 depicts a schematic representation of this method, where the negative reward is set to be equal to $r = -hk_{max}$.

- **Action-masking wrapper:** this wrapper specifies another possible method of using the sampling deadline during the learning process. Since the deadlines are known *a priori* (i.e. before choosing an action), it is possible to disallow the agent to choose illegal sampling times. This can be done by constructing an action-mask consisting of ones and zeros, which specifies the legal and illegal actions, and supplying this mask to the agent. For example, when $k_{max} = 5$ and the deadline associated with the current state is $d(x) = 3$, the resulting action mask would be $[1, 1, 1, 0, 0]$. During the calculation of the action probabilities by the agent, this action-mask can be used to push the probabilities of choosing an illegal action to zero. Section 3-2-3 provides a more detailed description of these calculation on the agent side. Using this method over the action-penalisation method greatly reduces the action space, which should stabilise and speed up the learning process.

- **n-Spherical coordinates wrapper:** since states that lie on the same radial axis are associated with the same sampling times (as explained in Section 2-1-3), it could be useful to transform the cartesian state coordinates to n-spherical coordinates, and only send the angles as observations to the agent. Incorporating this *a priori* knowledge on the shape of the solution greatly reduces the policy search space: instead of having to explore the entire reachable state set one can restrict exploration to a small box around the origin, which should stabilise and speed up the learning process. States that lie on the same radial axis but on opposite sides of the origin are also associated with the same sampling time, so should be treated as the same abstracted state. This reduces the search space again by a factor 2, and results in perfectly symmetrical policies by definition.
- **Smoothness reward wrapper:** as will be shown in Section 4-4, maximising long-term sampling times could lead to sharper trajectory changes when sampling. In order to combat this effect, the user should be able to incorporate a term for the smoothness of the trajectory into the reward function. This wrapper calculates a weighted sum of the reward due to the sampling time and a reward based on the angle between the trajectory directions before and after sampling. Formally:

$$r(x,k) = \beta(1 - \frac{\psi(x,k)}{\pi}) + (1-\beta)\frac{k}{k_{max}}, \quad \text{with } \psi(x,k) = \arccos\frac{\mathbf{a}\cdot\mathbf{b}}{|a||b|} \tag{3-2}$$

  where $\psi \in [0,\pi]$ is the angle between the trajectory vectors before and after sampling (i.e. $\mathbf{a}$ and $\mathbf{b}$ resp.), and $\beta \in [0,1]$ is a weighting term. Note that both $(1-\frac{\psi}{\pi}) \in [0,1]$ and $\frac{k}{k_{max}} \in [0,1]$ are normalised values, so the weighting term β directly specifies the trade-off between sampling performance and trajectory smoothness.

3-2 Agent

Next, the agent used for solving the environment will be discussed. First, the reasoning behind the choice of RL algorithm will be presented, followed by a overview of the used hardware.

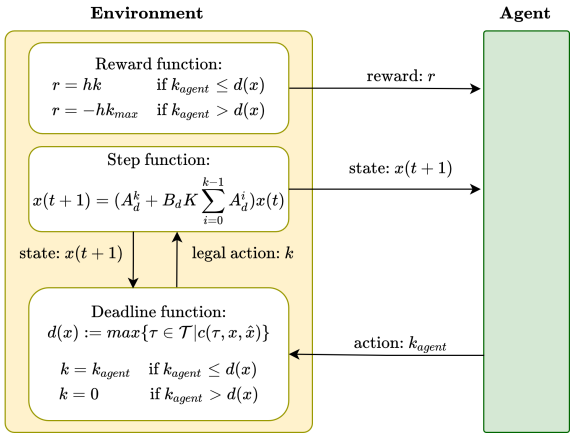


Figure 3-1: Schematic representation of the environment with the penalty wrapper.

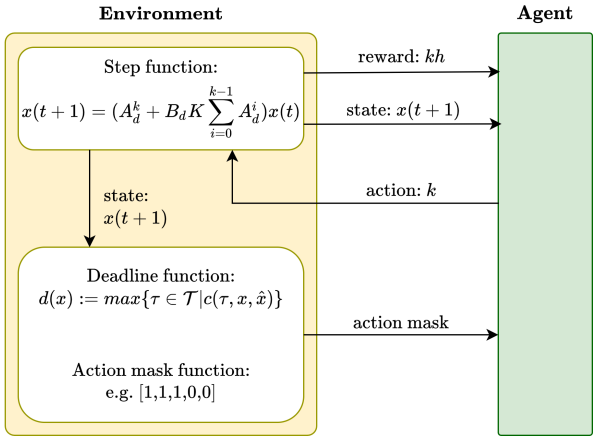


Figure 3-2: Schematic representation of the environment with the action-mask wrapper, with an example action-mask for $k_{max} = 5$ and $d(x) = 3$.

Then, some components specific to this implementation will be discussed, like action-masking and neural network expansion. The final part of this section will give a quick comparison of the different components.

3-2-1 Algorithm selection

As discussed in Section 2-2-2, the choice of which DRL algorithm to use for solving a specific problem is not a trivial one: no algorithm is clearly better for every use case. For example, when experience is collected through interaction with a real physical system like a robot, high sample efficiency is an important feature of the RL algorithm, so one might prefer an off-policy algorithm from the Q-learning family because of the possibility to reuse experience. On the other hand, experience collected through simulation can be very "cheap" and sample efficiency becomes less of a concern. However, if the environment is very complex and calculating a step is computationally expensive, sample efficiency can again be of more importance. If the sample efficiency is of less concern, on might prefer an on-policy Policy Gradient algorithm for their learning stability and strong convergence. A simulated environment also enables parallelisation of the experience generation, which decreases the learning wall-clock time. Other important considerations are hyperparameter-sensitivity and compatibility with discrete or continuous state- and/or action-spaces. The problem considered here consists of a continuous state-space and a discrete action-space, which eliminates or complicates the use of some algorithms.

PPO has shown state-of-the-art performance on a wide variety of applications. When sampling efficiency is not of the greatest concern, the simplicity, ease of tuning and possibility for parallelisation presented by PPO currently make it the preferred method for discrete-action problems. The PPO implementation from the Stable Baselines 2 library [13] was adapted for this thesis research. Stable Baselines is a set of implementations of DRL algorithms based on OpenAI Baselines, and is widely used in the academic world. This PPO implementation is very similar to the Actor-Critic style algorithm from the original PPO paper [26], which uses gradient clipping, a rollout buffer and periodical policy updates as depicted in Figure 3-3 and explained in Section 2-2-3.

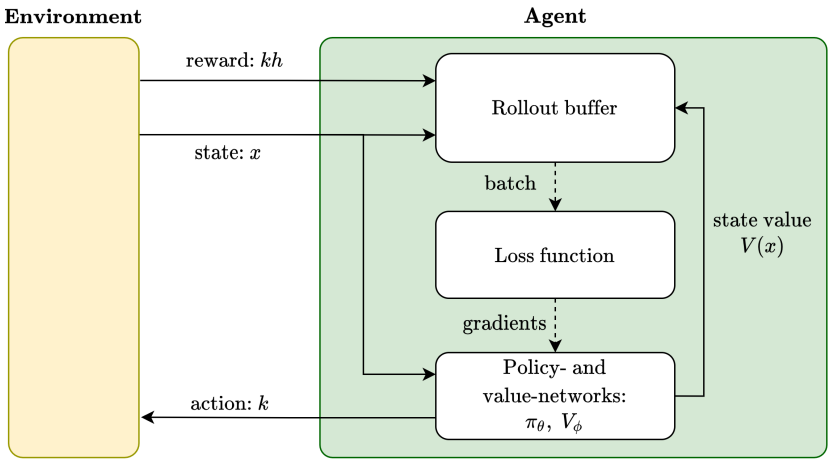


Figure 3-3: Schematic representation of the Actor-Critic style PPO implementation. The dashed lines represent periodical updates after finishing a rollout.

3-2-2 Hardware and parallelisation

Deep reinforcement learning requires the generation and processing of large amounts of data. As discussed in Section 2-2-2, the use of parallel computing is therefore essential. OpenAI Five - an algorithm capable of beating professional Dota5 players - for example was trained using a scaled-up version of PPO on 256 GPUs and 128,000 CPU cores. In that setup, the CPUs are used to simulate the environment, and GPUs are used for neural network inference (i.e. action selection) and policy updates. The experiment described in this thesis were run on a Intel Xeon W-2145 CPU with 16 logical cores, where both simulation and NN interactions are handled by the CPU. In theory, increasing the number of parallel experience-generating environments should result in better policy performance and wall-clock times. However, in practice there are some limiting factors related to the parallelisation scheme and available hardware. Figure 3-4 describes the parallelisation scheme used in this research, which is mostly based on the method described in [28] and differs from the standard Stable Baselines implementation. This scheme essentially consists of two paths: the *simulation path* and the *policy update path*. The simulation path consists of alternating environment steps to obtain the updated states and neural network inference to obtain new actions. Unlike threads, processes do not share memory with other processes. Since the environment interactions and NN inference run on separate processes, there will be some latency introduced by the transmission of the states and actions between these processes (called *inference communication time* in Figure 3-4).

A *rollout* consists of multiple of these simulation loops run in series and results in a large batch of experience tuples. This batch is then used to update the neural network parameters using a gradient descent algorithm. Note that since NN inference and policy updates are run on the same process, the batch of experience samples is already available to the neural network process, so this does not introduce latency due to communication. Once the policy update is completed, a new rollout will commence.

Since every CPU logical core can only run one thread per moment in time, the amount of parallel *simulation processes* is limited to the number of CPU cores, in this case $n = 16$. However, this does not mean that the number of environments is limited to 16 as well:

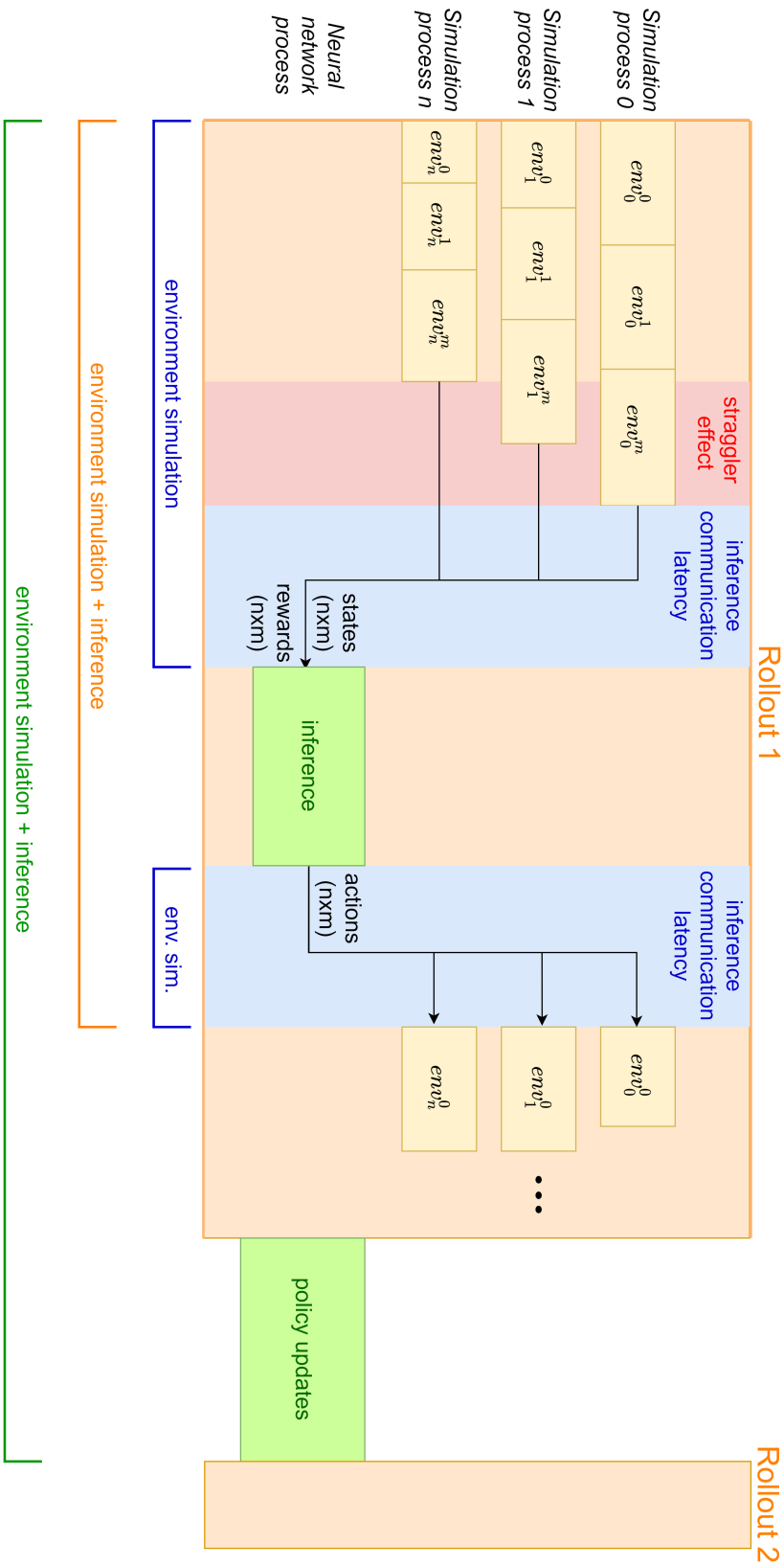


Figure 3-4: Parallelisation scheme used for the proposed tool.

it is possible to launch multiple environment instances on each parallel process. The m environments on each process will all take one step in series, after which the $n \times m$ updated states are aggregated and send to the *neural network process* for inference. Increasing the amount of environments in series on each process has two major advantages:

- A reduction is latency due to inference communication. When there is only one environment running on each of the n parallel processes, this would result in an inference batch of n samples and thus $\frac{1}{n}$ inference communications per sample. However, when we have m environments running in series on each process, this would result in an inference batch of nm samples and thus $\frac{1}{nm}$ inference communications per sample. This constitutes a reduction of latency per sample and thus a higher sampling rate.
- When multiple environment steps are simulated in parallel, the inference step has to wait until the slowest of these environment computations is finished. This is called the *straggler effect* and was researched in relation to DRL by Stooke et al. [28]. Variance in stepping time mainly arise from varied computation loads. In this thesis research for example, an environment step consists of k matrix multiplications and k calls to the trigger function, which can result in a large difference in computation time especially for higher-dimensional systems. The straggler effect will increase with increased number of parallel processes n. Running multiple independent environment steps in series on each process is a way of combating the straggler effect, since the sum of the computation times per parallel process will be closer to the mean.

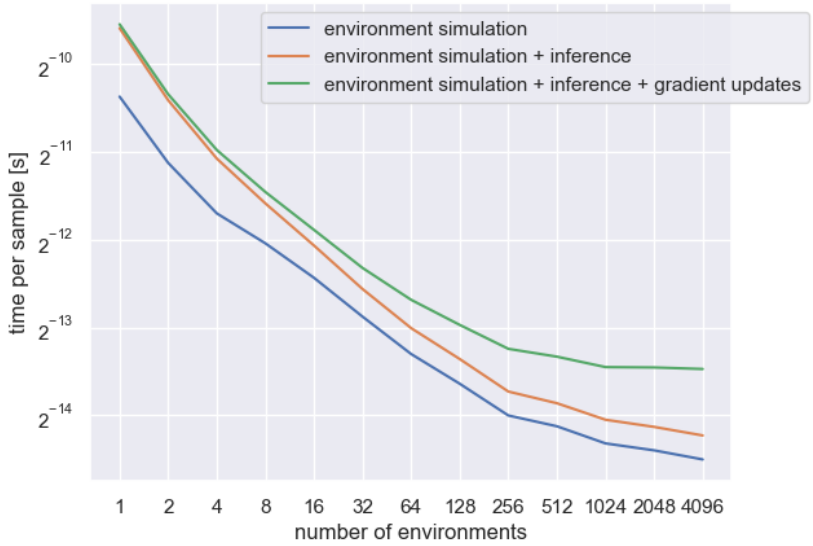


Figure 3-5: Comparison of achieved sampling rates for different amounts of environments.

These theoretical effects were verified experimentally. Figure 3-5 compares the achieved time per experience sample when using different amounts of environments. The maximum amount of parallel processes during this experiment was set to 16, due to hardware limitations. This means that when less than 16 environments are launched, every process contains only one environment. The blue line represents the simulation time per experience sample when only

counting the environment calculations, i.e. not including NN inference and gradient updates. However, this does include the inference communication latency, since the measurement was done on the neural network process (see Figure 3-4). The orange line represents the simulation time per experience sample including inference time, and the green line represents the overall simulation time per experience sample which also includes policy updates. For reference, the colored brackets in Figure 3-4 correspond to these lines.

One can clearly observe that an increase of environments results in a decrease in simulation time per sample. However, all lines flatten out and the sampling rate including gradient updates diverges from the others. This implies that the latency due to inference communication and straggler effect are largely reduced, and one of the remaining bottlenecks is the speed of the gradient calculations and policy updates. The difference between the blue and orange line is due to the latency resulting from performing the actual NN inference. The resulting bottlenecks are now the inference speed, policy update speed and the number of parallel workers, all of which are hardware related. A hardware accelerator like a GPU or TPU could reduce the latency introduced by inference and policy updates further.

3-2-3 Action masking

As discussed in Section 3-1-2, one way of ensuring the safety of the final policy is to apply *action masking* to the gradient calculations. Figure 3-6 shows a visualisation of neural network inference without and with action masking.

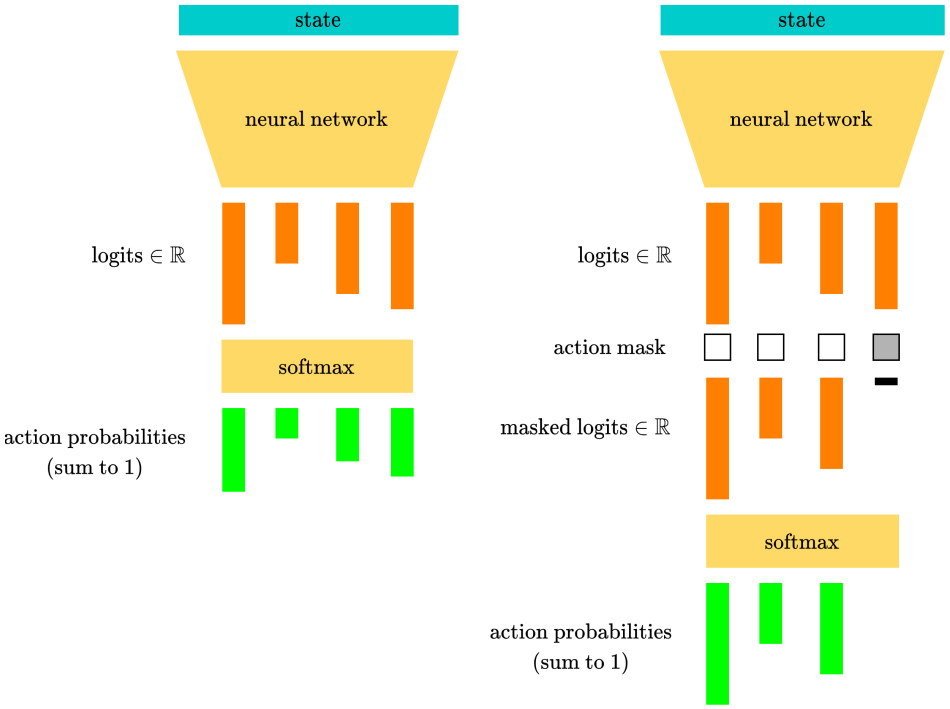


Figure 3-6: Schematic representation of regular neural network inference (l) and neural network inference using action-masking (r).

To further illustrate how action-masking works, consider the following example, similar to the example from [14]. Suppose there are 4 possible actions $a_1 - a_4$, which are given equal probabilities from state s_0 by the policy:

$$\begin{aligned}\pi_\theta(\cdot \mid s_0) &= [\pi_\theta(a_1 \mid s_0), \pi_\theta(a_2 \mid s_0), \pi_\theta(a_3 \mid s_0), \pi_\theta(a_4 \mid s_0)] \\ &= \text{softmax}([l_1, l_2, l_3, l_4]) \\ &= [0.25, 0.25, 0.25, 0.25]\end{aligned} \tag{3-3}$$

with $l_n \in \mathbb{R}$ the logit corresponding to action a_n. Now imagine that from state s_0 only the first three actions are legal. Action masking can disable the illegal action by replacing the logits corresponding to illegal actions by $-\infty$ or a large negative number, before feeding the logits to the softmax function. The softmax function will then push the action probabilities associated with the illegal actions to zero, resulting in the agent almost certainly not choosing these actions:

$$\pi'_\theta(\cdot \mid s_0) = \text{softmax}(\boldsymbol{l}_{masked}) = [0.33, 0.33, 0.33, 0.00] \tag{3-4}$$

During stochastic gradient descent, again only valid actions are used in the calculation of the gradient of the objective function. This makes all gradients related to the masked actions equal to zero, while renormalising the gradients for the other actions.

The main advantage of this method over penalising illegal actions, is the reduction of the action search space. To illustrate this, consider Figure 3-7: when no action masking is applied, all state-action pairs that lie under the red line have to be sampled in order to first learn which actions are illegal, while learning an optimal strategy. Even once the deadlines have been learned, the action probabilities of illegal actions will not become zero. This results in a waste of computing power due to sampling of illegal actions. However, when we directly use the *a priori* information we have regarding the deadline $d(x)$ in the form of action-masking, only the state-action pairs that lie below the green line have to be explored, thus greatly reducing the search space. Another small advantage of this method is that the reward function is not altered by adding penalties for illegal action, which makes monitoring of the learning process more straightforward. A disadvantage of this method is that action masking and thus deadline calculation is also required during evaluation of the final policy, in order to obtain good results and guarantee safety (i.e. $k(x) \leq d(x)$). However, the alternative penalisation method also requires calculation of the deadlines during evaluation, in order to guarantee safety of the final policy, because in that case the learned policy is still not guaranteed to exclude illegal actions.

3-2-4 Neural network expansion

Deciding on the number of layers (depth) and the amount of nodes in each layer (width) of a neural network is not a trivial task. The universal function approximation theorem states that a feedforward neural network with only 1 hidden layer can approximate any continuous function for inputs within a specific range, given it contains enough nodes in its hidden layer. Although this is true, this does not mean that only having one layer in a neural network is practical: in order to be able to learn complex functions, the width of the hidden layer can become infeasibly large and may fail to learn and generalize correctly. Adding

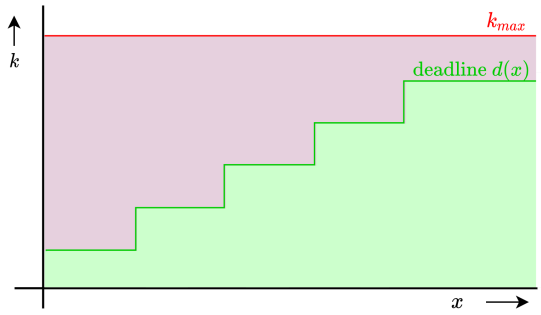


Figure 3-7: Schematic representation of some 1-dimensional state-action space, with an upper limit on the actions of k_{max}. The green area is the legal part of the state-action space, the red area the illegal part.

additional hidden layers has proven to be beneficial for learning more complex functions, where it has been hypothesised that each concurrent layer extracts more high-level features from the input data. However, simply adding more hidden layers also has some downsides: deep neural networks are more prone to overfitting, require large amounts of data, can suffer from unstable (i.e. exploding and vanishing) gradients and have a less convex loss function [20] which result in slower training or even divergence. Ota et al. [20] have shown that naively deploying deeper neural networks for DRL does not lead to better performance due to their non-convex loss surface, while wider networks have a more convex loss surface resulting in better convergence. Relatively shallow networks of 2 layers are still very prevalent in DRL. However, recent studies (e.g. [20], [4]) have shown that increasing network depth can in fact result in improved performance when some specialised techniques are used.

The method proposed in this thesis uses incremental expansion of the neural network. The general idea is to start out with a relatively shallow network and add hidden layers to the network when learning converges. Before giving a more detailed description of this implementation, the motivation behind this approach will be illustrated:

- In order to obtain more accurate and optimal results, the abstraction-based method by Gleizer et al. [9] uses refinement of the abstractions by increasing the depth of the abstraction l, which allows for the synthesis of more complex sampling policies. One way of mimicking this using deep reinforcement learning, is to increase the episode length, in order to capture more accurately the long-term system dynamics. However, if the expressiveness of the neural network is the bottleneck, the added complexity from increasing the episode horizon will not be absorbed effectively by the neural network. This motivates the use of sufficiently deep neural networks.

- Since the policy is trained for a specific system and does not need to generalise to other systems, using deeper neural networks can not result in overfitting.

- However, simply starting with a very deep neural network showed to result in high computational loads during policy updates, due to the large amount of trainable parameters. This motivates the use of neural networks that are as small as possible, while not being too small to capture the required complexity of the sampling behaviour.

- Also, convergence was slightly worse when using a fixed depth, deep neural network. This is most likely due to the non-convexity of the loss surface of deeper networks, as described by [20]. Using a more shallow network enabled the agent to more quickly learn moderately well-performing policies, but limited the performance of the final policy. See Appendix A for a comparison.
- The hypothesis presented here is as follows: Using neural network expansion allows for the quick learning of coarse but moderately well-performing policies, while ensuring that the expressiveness of the neural network is not the limiting factor when the episode length is increased during later stages of the learning process. Appendix A includes some empirical evidence to support this hypothesis.

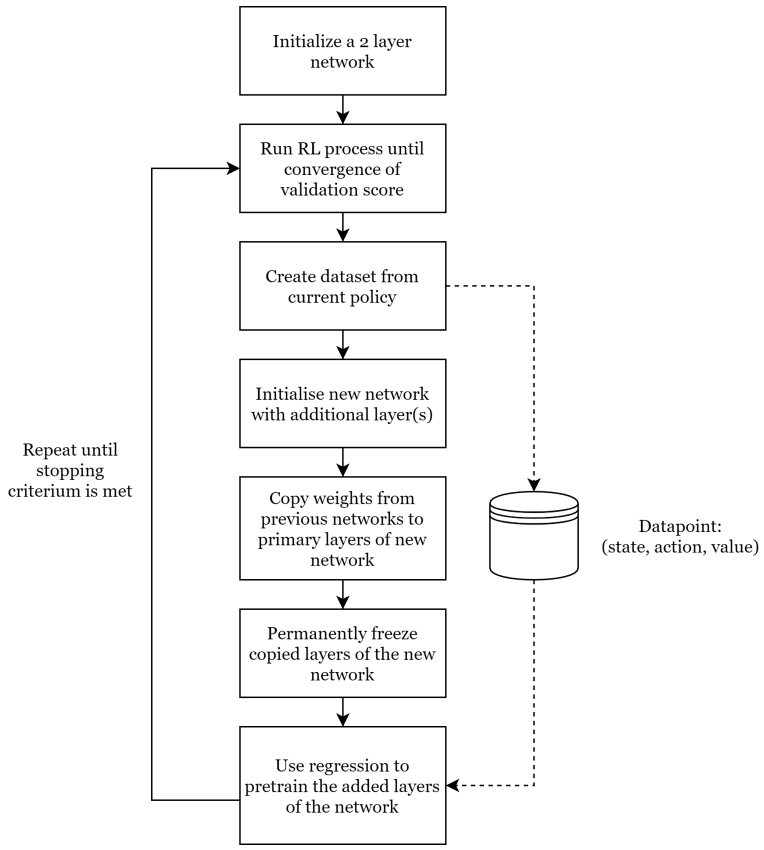


Figure 3-8: The neural network expansion workflow

Although this implementation uses separate networks representing the policy- and value functions, this section will simply use the term "network" referring to both policy- and value-network for readability purposes. Figure 3-8 describes the learning process including neural network expansion. The learning process should start out with a network that is as small as possible in order to reduce computational load and speed up the initial learning. Experiments showed that starting with only 1 layer resulted in bad performance, so the learning process should be initialised with a 2 layer network. The weights of the network are initialised randomly, using an orthogonal initialisation scheme [23]. The reinforcement learning process is then initialised. During the learning process, the sampling performance is measured periodi-

cally by running the current policy on a validation environment. If the validation score does not improve anymore, it could be a sign that the current neural network is not able to capture further complexity and the RL loop is stopped. This is when the neural network expansion process commences.

First, a dataset consisting of state-action-value tuples is extracted from the current network for later use. This is done by repeatedly simulating the system using the current network and saving the trajectory steps.

A new network with one or more additional layers is then initialised, and the weights of the old network are copied to the primary layers of the new network. The resulting new network is then essentially a copy of the old network, but contains extra layers with random weights.

The copied layers of the new network are then permanently frozen, which means any future gradients are not calculated for and do not affect the weights in these layers, an idea borrowed from *transfer learning*. One could now consider the frozen layers a sort of stationary feature extractor for the newly added layers. The reason for this is twofold. Firstly, freezing of the previously trained layers ensures that the previously learned information is not lost. Secondly, any future gradients will only flow into the added layers during backpropagation, which greatly speeds up policy updates and reduces the required computational load.

One problem still exists: how does one initialise the weights of the newly added layers in order to ensure that there is no severe loss in policy performance when the reinforcement learning process is continued? Experiments showed that random weight initialisation (e.g. orthogonal) often resulted in sharp collapse of the policy performance, which is undesirable. To prevent this, the new network is pretrained on the previously obtained dataset using supervised learning, specifically regression with an MSE loss. Pretraining of the policy network uses the state-action pairs, while pretraining of the value network uses the state-value pairs. This process is called *imitation learning* or *behaviour cloning*, and it essentially transfers the learned input-output behaviour from the old to the new network.

At this point, the old network and trajectory dataset are no longer needed and could be removed if necessary. The reinforcement learning process is then recommenced. This alternating process of reinforcement learning until convergence of the validation score and neural network expansion can be repeated until some stopping criterion is met.

D2RL [27] is a neural network architecture which incorporates dense connections from the input to each of the layers for the policy, as well as for the value function. [27] showed that this can be useful for reinforcement learning with deeper neural networks, and D2RL showed better performance with lower variance and higher sample-efficiency compared to standard dense networks. For these reasons, the D2RL architecture was incorporated in the proposed method.

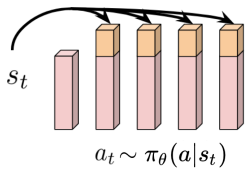


Figure 3-9: The D2RL modification to the policy $\pi_\theta(s)$ neural network architecture [27].

The main advantage of using this method over using a fixed-depth shallow network is the increase in performance due to the ability to capture more complexity. The main advantage of this method over using a fixed-depth deeper neural network is the decrease of computational requirements by the neural network process during policy updates, while still matching the performance. Appendix A includes a more in-depth evaluation of this neural network expansion method.

3-3 Experimental setup

The metrics used to evaluate the final policy π_θ are:

- $\text{SAIST}_{\text{approx}}(\pi_\theta) := \inf_{x \in X_M} \sum_{i=0}^{N} \tau_i(x, \pi_\theta)$,
  where X_M is a set of M different, equally separated initial conditions.
- $\text{MAIST}_{\text{approx}}(\pi_\theta) := \frac{1}{M} \sum_{j=0}^{M} \sum_{i=0}^{N} \tau_i(x_j, \pi_\theta)$

Note that these values are approximations of the metrics described in Section 2-1-2, obtained experimentally by running the final policy from M different initial conditions for N timesteps. The smallest average inter-sample time obtained from these runs is the SAIST, the average over all these runs is the mean average inter-sample time (MAIST). These values will be calculated periodically during the learning process, in order to evaluate the learning process more accurately than just using learning curves.

From all the aforementioned components and settings, the best combination will be used for further evaluation of the proposed method. The following list is a summary of the settings and components used during evaluation:

- n-Spherical coordinates, where only the angles are send to the agent. States that lie on opposite sides of the origin are treated as the same abstracted state.
- Action-masking of illegal actions. Code adapted from [33].
- Neural network expansion as described in Section 3-2-4 with layer freezing, D2RL architecture, and behaviour cloning. The learning process is initialised with a network containing two layers of 64 nodes.
- The learning rate is initialised at 0.003 and is decreased with a factor 3 every time the neural network is expanded.
- The horizon is initialised at 10 steps and is increased with a factor 2 every time the neural network is expanded.
- All experiments are run for a fixed time period for better comparison.
- All experiments are run 5 times, in order to obtain a measure of the variance between runs.
- The hyperparameters used can be found in Appendix A. These hyperparameters were hand-tuned, since PPO is not overly sensitive to hyperparameter settings.

Chapter 4

Results

In this chapter, the results of this thesis will be presented. The method with the settings as described in Section 3-3 is evaluated on a 2-dimensional system using two different fundamental checking periods, a 3-dimensional system and a 4-dimensional system. These results are compared to the results obtained using the abstraction-based method from [9].

4-1 2-dimensional system

The 2-dimensional system used in this numerical example is taken from [9]. The system consists of the following plant and controller:

$$A = \begin{bmatrix} 0 & 1 \\ -2 & 3 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad K = \begin{bmatrix} 1 & -4 \end{bmatrix} \tag{4-1}$$

The triggering condition is the predictive Lyapunov-based triggering condition from [9] of the form $V(\zeta(t)) > -\rho\zeta(t)^T Q_{Lyap}\zeta(t)$, where $\zeta(t) := A_d(h)x(t) + B_d(h)K\hat{x}(t)$ is the next-sample prediction of the state, $V(x) = x^T P_{Lyap} x$, $\rho = 0.8$ is the triggering parameter. The Lyapunov matrices that were used are: $P_{Lyap} = \left[\begin{smallmatrix} 1 & 0.25 \\ 0.25 & 1 \end{smallmatrix}\right]$, $Q_{Lyap} = \left[\begin{smallmatrix} 0.5 & 0.25 \\ 0.25 & 1.5 \end{smallmatrix}\right]$. For the first experiment we set $h = 0.1$ and $k_{max} = 20$.

4-1-1 Experiment 1

For the first experiment, a fundamental checking period of $h = 0.1$ was used. The initial episode length was set to 10 samples, and increased by a factor 2 when the neural network was expanded. Figure 4-1 shows the SAIST values for 5 runs of the proposed RL-based method and the SAIST values for a run of the abstraction-based method. Note that the SAIST-values for the RL-based method and under PETC in Figure 4-1 are estimated by simulating the latest policy for 1000 steps from 500 equally spaced initial states. The final policy was evaluated by simulating the policy for 2000 steps from 1000 equally spaced initial states, for a more accurate estimate.

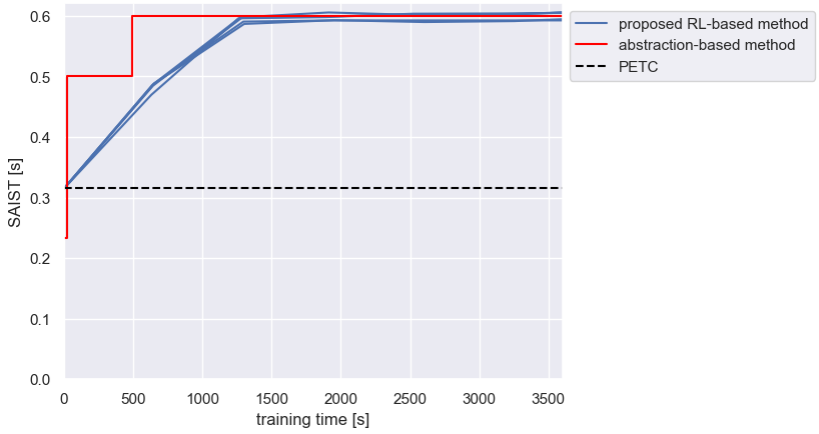


Figure 4-1: Estimated SAIST curves for the PETC, proposed RL-based method and abstraction-based method [9], for the 2-dimensional system with a fundamental checking time of $h = 0.1$.

The proposed RL-based method resulted in a final SAIST of 0.60 and a MAIST of 0.62, when averaged over the 5 runs. The abstraction-based method also resulted in a SAIST of 0.60. However, the abstraction-based method returned a SAIST value of 0.50 after just 24 seconds of runtime, and already reached its final SAIST value of 0.60 after just 492 seconds. This shows that although the final performance obtained using the abstraction-based method was similar to that of the RL-based method, the abstraction-based method is significantly faster than the proposed RL-based method for this numerical example. However, a different result is obtained when a shorter fundamental checking time is used, as was done in experiment 2.

4-1-2 Experiment 2

For the second experiment, the same system was used, but with a shorter fundamental checking period of $h = 0.05$. The settings remained unchanged from experiment 1. The results are depicted in Figure 4-2.

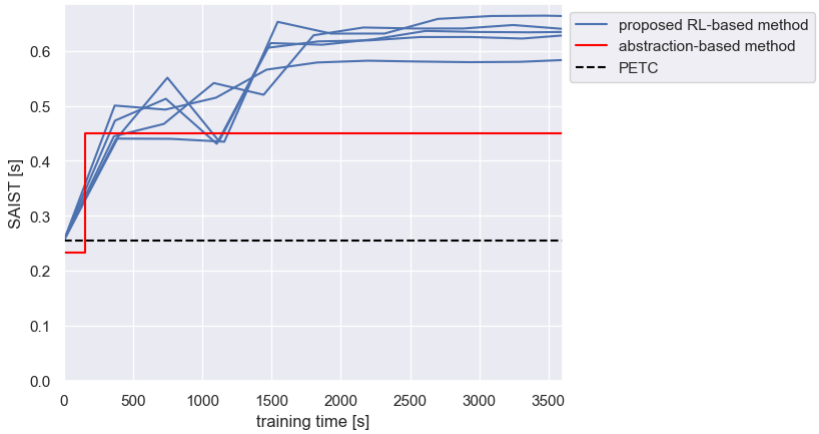


Figure 4-2: Estimated SAIST curves for the PETC, the proposed RL method and the abstraction-based method [9], for the 2-dimensional system with a fundamental checking period of $h = 0.05$.

In this experiment, the abstraction-based method resulted in a final SAIST of 0.45, whereas the proposed RL-based method resulted in a final SAIST of 0.66. This illustrates how abstraction-based methods can struggle with systems that are more complex in terms of sampling behaviour, while the proposed RL-based method is less affected. In fact, these experiments with $h = 0.05$ resulted in a higher SAIST value than for $h = 0.1$ when using the RL-based method. For comparison, Table 4-1 summarises the results obtained in experiments 1 and 2.

	h=0.1	**h=0.05**
PETC	0.32	0.27
Abstraction-based method	0.60	0.45
Proposed RL-based method	0.60	0.66

Table 4-1: SAIST-values obtained in experiment 1 and 2, including the PETC values for reference.

To fully understand where this difference stems from, consider the following. All possible policies for $h = 0.1$ are also possible for $h = 0.05$, by just increasing the sampling time k for every state by a factor 2 (if we assume k_{max} to be large enough to not limit the possible policies). Furthermore, the shorter fundamental checking period h results in a larger infinite set of possible transitions, and therefore results in a larger infinite policy space. This increase in possible transitions is exactly the reason why for $h = 0.05$ the abstraction-based method is not able to match the SAIST-value it obtained for $h = 0.1$: the size of the abstraction grows significantly. The proposed RL method however does not suffer from the same problem, and still shows convergence to approximately the same values as for $h = 0.1$. Furthermore, the larger policy space associated with the shorter fundamental checking period could contain policies that have a more refined and complex division of the state-space, which could result in better average inter-sample times, as is the case in experiment 2. This is illustrated by Figure 4-3, which depicts an example policy obtained for $h = 0.1$ and for $h = 0.05$.

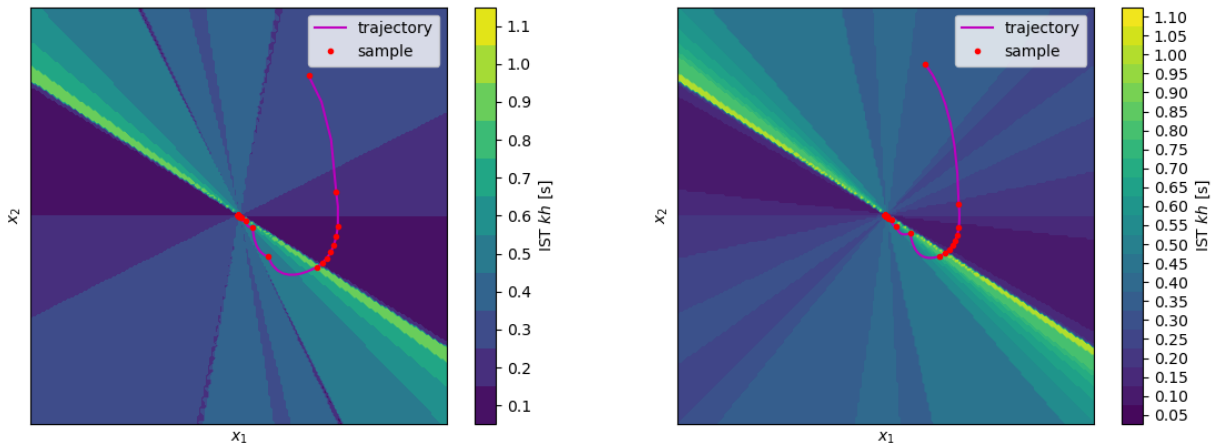


Figure 4-3: Example policies for $h = 0.1$ (left) and $h = 0.05$ (right), obtained using the proposed RL-based method. The trajectories resulting in the SAIST values are included.

These policy plots in Figure 4-3 include an example trajectory, from the initial condition that results in the SAIST-value. This illustrates how this policy ensures that the system enters a stable trajectory in a region of the state space which results in higher average inter-sample times.

Also note that the SAIST under PETC for $h = 0.05$ is lower than that for $h = 0.1$ (see Table 4-1). This might be counter-intuitive: since these experiments use a predictive Lyapunov-based triggering condition, the immediate inter-sample time for $h = 0.05$ can only be larger or equal to that for $h = 0.1$, as illustrated in Figure 4-4. However, this result is actually in line with the hypothesis that forms the starting point for this thesis: a longer immediate inter-sample time can lead to shorter long-term average inter-sample time.

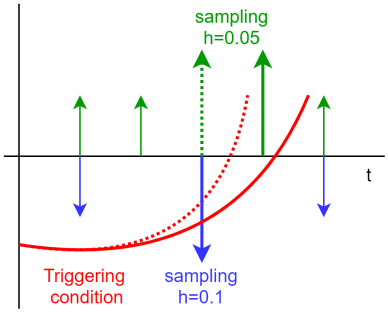


Figure 4-4: Illustrative example showing the possible sampling times for $h = 0.1$ and $h = 0.05$ for two possible trajectories of a predictive triggering condition.

Figure 4-5 shows the running average of the ISTs generated for the 2-dimensional system with $h = 0.05$ from 20 different initial conditions under PETC and a policy obtained using the proposed RL-based method. Note how the AIST values when following the obtained policy are much slower to converge than when following the PETC strategy. However, within a few seconds of simulation, the AISTs for all trajectories following the trained strategy is already higher than the AISTs following the PETC strategy.

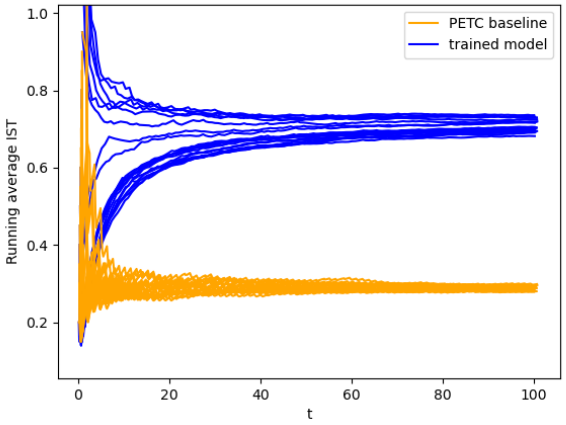


Figure 4-5: Running average of the ISTs generated from 20 different initial conditions under PETC and the strategy obtained using the proposed RL-based method in experiment 2.

4-2 3-dimensional system

To further evaluate the proposed method, it was tested on a 3-dimensional system. The 3-dimensional system used in this numerical example consists of the following plant:

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & -1 & -1 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{4-2}$$

The controller K was synthesised using pole placement, with the closed-loop poles placed at $(-1, -2, -3)$. The triggering condition is the predictive Lyapunov-based triggering condition from [9] of the form $V(\zeta(t)) > -\rho\zeta(t)^T Q_{Lyap}\zeta(t)$, where $\zeta(t) := A_d(h)x(t) + B_d(h)K\hat{x}(t)$ is the next-sample prediction of the state, $V(x) = x^T P_{Lyap} x$, and $\rho = 0.8$ is the triggering parameter. The Lyapunov matrix Q was set to the identity matrix and P was obtained using the `lyap` function from the Python Control Systems Library [38]. We set $h = 0.05$ and $k_{max} = 20$.

Figure 4-6 contains the estimated SAIST and MAIST over curves obtained using the proposed RL method for 5 runs. The final policy results in a SAIST of 0.64 averaged over the 5 runs, which again is higher than simply following the "greedy" PETC sampling strategy, which results in a SAIST of 0.40. Note that during training, the SAIST value can sometimes temporarily decrease greatly while the MAIST value is still growing. A possible explanation for this is that the policy is optimised for the MAIST (i.e. the mean AIST over all initial conditions), which gives no guarantees on the SAIST. However, in these experiments it was observed that this deviation is usually temporary, as can also be observed in Figure 4-6.

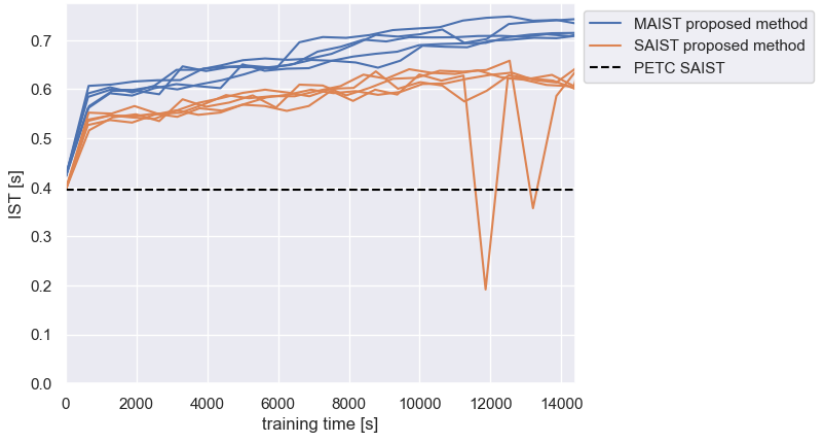


Figure 4-6: Estimated SAIST and MAIST curves for 5 runs of the proposed RL method, for the 3-dimensional system. The PETC SAIST-value is included for reference.

No value for the abstraction-based method is included in Figure 4-6, because the abstraction-based method was not able to run on the hardware used for these experiments. The reason for this is memory overload due to the sheer size of the abstraction, even of depth $l = 1$. The abstraction-based method has to calculate and store in memory the entire abstraction, before being able to extract a policy from it. This is where the true advantage of the proposed method again becomes evident: since transitions are calculated one step at a time and are only

stored in memory for the duration of one rollout, the memory requirements are far smaller and don't grow significantly with the size of the transition system. This again illustrates how the proposed RL-based method provides greater flexibility and can be applied to complex systems that prevent the use of abstraction-based methods.

4-3 4-dimensional system

Finally, the proposed method was evaluated on a 4-dimensional system:

$$A = \begin{bmatrix} 1.38 & -0.208 & 6.71 & -5.676 \\ -0.581 & -4.29 & 0 & 0.675 \\ 1.067 & 4.273 & -6.654 & 5.893 \\ 0.048 & 4.273 & 1.343 & -2.104 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & 0 \\ 5.679 & 0 \\ 1.136 & 3.146 \\ 1.136 & 0 \end{bmatrix}, \tag{4-3}$$
$$K = \begin{bmatrix} 0.518 & -1.973 & -0.448 & -2.1356 \\ -3.812 & -0.0231 & -2.7961 & 1.671 \end{bmatrix}$$

The triggering condition again is the predictive Lyapunov-based triggering condition of the form $V(\zeta(t)) > -\rho\zeta(t)^T Q_{Lyap}\zeta(t)$, where $\zeta(t) := A_d(h)x(t) + B_d(h)K\hat{x}(t)$ is the next-sample prediction of the state, $V(x) = x^T P_{Lyap} x$, $\rho = 0.8$ is the triggering parameter, $h = 0.01$ and $k_{max} = 100$. The Lyapunov matrix Q was set to the identity matrix and P was obtained using the `lyap` function from the Python Control Systems Library [38].

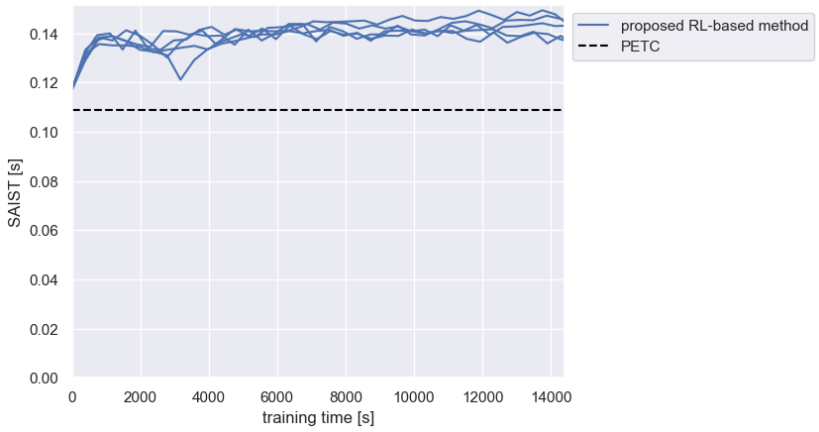


Figure 4-7: Estimated SAIST curves for 5 runs of the proposed RL method, for the 4-dimensional system. The PETC SAIST-value is included for reference.

Figure 4-7 contains the estimated SAIST and MAIST curves (i.e. estimated using 1000 steps from 500 equally spaced initial states) obtained using the proposed RL method for 5 runs of 4 hours each. The final policy results in an MAIST of 0.153 and a SAIST of 0.146 averaged over the 5 runs, which again is higher than simply following the "greedy" PETC sampling strategy, which results in a SAIST of 0.118. This demonstrates how even for higher-dimensional systems the proposed method is able to obtain sampling strategies that result in better AISTs than following the PETC strategy, and does so in a reasonable amount of time.

Also note that most hyperparameter settings were left unchanged from the experiments on lower-dimensional systems, with the exception of initial episode length which was set to 80. Further tuning of hyperparameters like increasing the buffer size, NN width and episode length could potentially result in better performance for higher-dimensional systems.

4-4 Trajectory smoothness

As mentioned in Section 3-1-2, an alternative cost function was implemented which incorporates a measure of trajectory smoothness besides the usual term for the average sampling-time. For reference:

$$r(x,k) = \beta(1 - \frac{\psi(x,k)}{\pi}) + (1-\beta)\frac{k}{k_{max}}, \quad \text{with } \psi(x,k) = \arccos\frac{\mathbf{a}\cdot\mathbf{b}}{|a||b|} \tag{4-4}$$

where $\psi \in [0, \pi]$ is the angle between the trajectory vectors before and after sampling and $\beta \in [0, 1]$ is a weighting term.

The hypothesis presented here is as follows: *Optimising for inter-sample times will result in fewer but stronger control actions and therefore a decrease in trajectory smoothness.* This hypothesis was evaluated by running the same experiment for different values for weighting coefficient $\beta \in [0, 1]$. Figure 4-8 depicts the obtained SAIST-values and the angle (in degrees) between the trajectory vectors before and after sampling ψ for different values of β. A clear inverse relation between the SAIST and the trajectory smoothness can be observed. When $\beta = 0$ (i.e. trajectory smoothness is not considered) the SAIST-value is highest and the trajectory smoothness is low, namely the trajectory vectors before and after sampling are almost perpendicular. Conversely, when $\beta = 1$ (i.e. average inter-sample times are not considered) the SAIST-value approaches the fundamental checking period $h = 0.05$ and the trajectory vectors before and after sampling are almost tangent. This result supports the hypothesis presented here and shows how the proposed tool allows the user to make a trade-off between ISTs and trajectory smoothness by varying β.

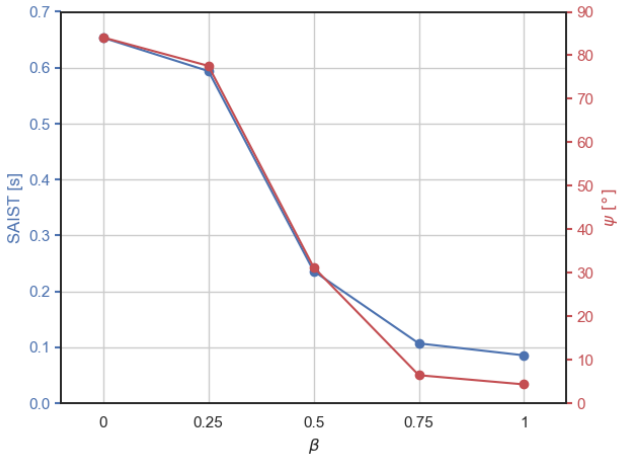


Figure 4-8: Comparison of the angle (in degrees) between the trajectory vectors before and after sampling ψ and the SAIST for different values for weighting coefficient β.

Chapter 5

Conclusion and future work

5-1 Conclusion

This thesis proposes a deep reinforcement learning tool for the synthesis of near-optimal sampling strategies for self-triggered control systems. The viability of the proposed tool as an alternative to abstraction-based methods such as [9] was shown through some numerical examples. In these experiments, the proposed method was able to match the sampling performance obtained by the abstraction-based method for an example 2-dimensional system. When the same system was used but with a shorter fundamental sampling time, the proposed method was able to exceed the performance of the abstraction-based system. The abstraction-based method was not able to run for a 3-dimensional and 4-dimensional system, due to memory capacity limitations. The proposed method however could still find sampling strategies that resulted in better average inter-sample times than simply following the PETC sampling strategy. This illustrates how the proposed method is a viable alternative that can be applied to more complex systems that prohibit the use of abstraction-based methods.

The proposed tool was designed with scalability in mind. The neural network expansion component for example, enables the use of deeper and wider neural network architectures, which might be necessary for more complex systems in terms of sampling behaviour. The use of parallel workers enables scaling of the generation of experience, but is still limited by the available hardware (i.e. amount of CPU cores and use of hardware accelerators).

Finally, the proposed tool enables the user to include a term for the trajectory smoothness into the reward function. An inverse relation between trajectory smoothness and average inter-sample times was found during evaluation of the tool, which could be of interest in some use-cases.

The higher flexibility presented by the proposed method compared to more formal methods like the abstraction-based method from [9] does come with some downsides. Not every run of the proposed method will converge to the exact same final value and the policy might get stuck in local optima, as can be observed from the experiments in Chapter 4. In order to minimise this effect, one might need to adjust the hyperparameter settings and/or run

the algorithm multiple times with different random seeds. Settings like rollout buffer size, minibatch size, learning rate, discount factor, horizon length, and neural network width and depth can have an effect on the learning process. Some general guidelines on how to adjust these hyperparameters are included in Appendix B.

5-2 Future work

Future research on this topic could focus on improving the current tool. One could pursue the use of more modern, larger neural network architectures, which could improve performance. Recent research [4] has shown that the use of modern network architectures in DRL can outperform state-of-the-art methods, but to ensure stable training requires techniques like spectral normalisation. Another option would be to include some method(s) to increase sample efficiency, like prioritized trajectory replay [15]. Finally, one could consider techniques to enhance the exploration efficiency, like including parameter noise [22], or some form of intrinsic reward ([21], [5], [6]).

Another route could focus on scaling of the current tool, by employing multiple CPU-GPU platforms in parallel, all running the current algorithm. Gradients could then be globally averaged using MPI Allreduce or similar to update the central model parameters, similar to the setup of OpenAI Five [3].

This research has not considered the generalisation capabilities of the models obtained using the proposed method. An interesting question is whether the obtained models can be used to facilitate the training of new models for different systems, by applying techniques from transfer learning.

In this research, the action space was strictly constrained to respect the deadlines imposed by the PETC triggering condition, which guarantees safety but might be too strict. An interesting question is whether this constrained could be loosened somewhat, by allowing the agent to sometimes choose a sampling time beyond the imposed deadline. One could then include a reward related to some control performance measure like Lyapunov decrease, which would give the end user more flexibility in terms of shaping the reward function.

Other research directions could focus on the case where the state is not fully observable (which would result in a partially observable MDP), noisy state measurements or non-linear dynamics.

Appendix A

Comparison of components

The different components and settings as discussed in this chapter have all been tested and compared. In this section the most relevant findings will be discussed.

A-1 Scheduling of learning rate and horizon

An important hyperparameter is the learning rate, which determines the step size at each gradient update. Smaller and larger values for the learning rate both have their advantages and disadvantages. Larger values cause quicker convergence towards the optimum, and enable the optimiser to escape local optima. This can also present a problem though: when the learning rate is too high, it can cause unstable learning and even divergence, because the optimiser is not able to settle in an optimum. Smaller values on the other hand can cause the optimiser to converge very slowly and get trapped in a local optimum, but often result in a more accurate approximation of the optimum. In order to get the best of both world, the learning rate is often scheduled: learning is initialised with a high learning rate which is decreased over time. The proposed method also uses this approach. When using neural network expansion, the learning rate is decreased every time the network is expanded in order to prevent the learning process to become unstable. When not using neural network expansion, the learning rate is decreased linearly, from a start value to a final value in a fixed amount of timesteps.

As mentioned in Section 3-2-4, increasing the episode length or horizon is a way of capturing more of the complexity of the system dynamics, which could result in more complex and optimal sampling policies, similar to increasing the depth of the abstractions in [9]. However, shorter horizons result in better exploration of the state space, due to the random state initialisation of each new episode. This is especially important during early stages of training, to reduce the chance of the policy getting stuck in a local minimum. In order to again get the best of both worlds, the proposed method also schedules the horizon: starting the learning process with a shorter horizon for better exploration and increasing the horizon over time in order to obtain more accurate final results. When using neural network expansion the horizon

is increased every time the network is expanded. When not using neural network expansion, the horizon is increased every N environment interactions, with N a large number.

A-2 NN expansion versus fixed depth

Figure A-1 depicts a comparison between runs using the neural network expansion method and two runs with a fixed neural network depth: one with a shallow network of two layers and one with a deeper network of 6 layers. Every method was run 5 times. NN expansion happens every 5 million steps and the horizon increases with a factor 2. One can clearly see how the shallow network is not able to capture the increased complexity introduced by increase in horizon length, and is not able to match the performance obtained using the deep network and the expansion method. This motivates the use of deeper NNs.

The runs using the expansion method show slightly faster convergence than the ones with the fixed deep network, but their final performance is comparable. However, the deeper network method showed a higher use of computational resources by the neural network process as compared to the expansion method, in this case approximately 3 times as high. In the case of the expansion method, gradients are only calculated and applied to the last layer, which greatly reduces neural network training load and scales well to even deeper neural networks. This demonstrates the possible advantage of using NN expansion over a fixed depth deep network.

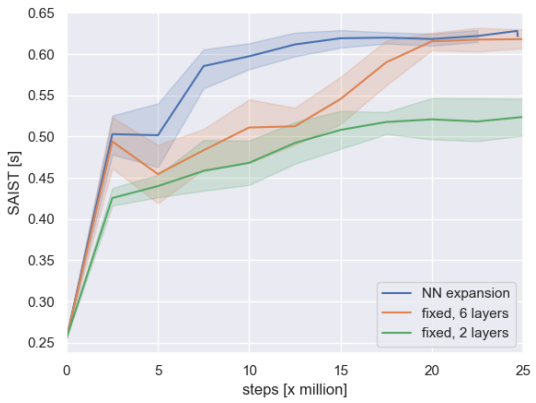


Figure A-1: Estimated SAIST curves and 95% confidence interval for a 2-layer and 6-layer NN, and using NN expansion. The width is fixed to 64 nodes.

Also consider the following: neural network expansion reduces the amount of parameters that have to be updated. If we fix the amount of computing power available for policy updates, a network using NN expansion can thus have wider hidden layers than a fixed depth NN with the same amount of layers. This could be advantageous since wider neural networks can help with convergence as shown by [20], which was also observed during experiments in this thesis research.

A-3 Action-masking versus action-penalisation

Figure A-2 depict the learning curves when using action-penalisation and action-masking. One can see that the average reward obtained using action-penalisation is lower than when using action-masking. The reason for this is decreased convergence due to the sampling of illegal actions by the action-penalisation method, whereas the action-masking method prevents the sampling of illegal actions.

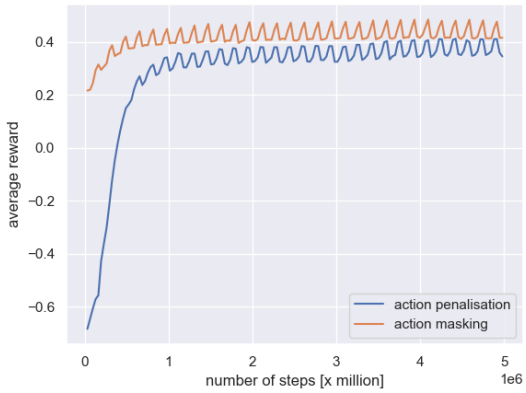


Figure A-2: Average reward curves of a run with action-penalisation and one with action-masking.

A-4 PPO hyperparameters

PPO is known for not being too sensitive to hyperparameter settings, so the hyperparameters were hand-tuned for the proposed tool. This section will give a brief overview of the relevant hyperparameters and the settings that were used.

- **Policy model**: for the base implementation an multi-layer perceptron (MLP) was used for the policy- and value-networks. Maintaining separate policy- and value-networks produced the best results.
- For the **activation function** three common variants were considered: ReLu, tanh and sigmoid. The ReLu activation function produced the best results.
- A **discount factor** γ of 1 produced the best results. This is to be expected, since we are interested in the *limit* average ISTs.
- The **entropy coefficient** c_1 specifies the strength of the entropy regularization (see Section 2-2-3). Higher values should ensure more random steps being taken by the agent, resulting in more exploration. However, this hyperparameter did not seem to have a large effect on the results, so the default value of 0.01 was used.
- The **clipping parameter** ϵ corresponds to the acceptable threshold of divergence between the old and new policies during gradient updates. Lower values ensure smaller

updates and more stable but slower learning as a result. A value of $\epsilon = 0.2$ produced the best results.

- The **minibatch size** is the number of samples in the rollout buffer divided by the number of minibatches per update, so in order to use all samples in the rollout buffer the number of minibatches should be a factor of the rollout buffer size. The size of the minibatches is limited by the available memory. In order to make sure this is not exceeded, the minibatch size was set equal to the number of environments, which makes the minibatch size equal to the size of an inference batch.
- The **number of epochs** specifies the number of passes through the rollout buffer during network updates. Higher values should result in quicker learning at the expanse of learning stability. A value of 10 produced the best results.

Appendix B

Tool User Guide

B-1 Requirements

The following packages have to be installed in order to use the proposed tool:

- python3-dev (>=3.5)
- cmake
- libopenmpi-dev
- zlib1g-dev
- tensorflow (1.8.0 - 1.15.0)
- tensorboard
- stable-baselines
- control

B-2 How to use?

The RL tool can be started by running `main.py`, after which the training process can be monitored using Tensorboard. The user should specify an instance of a triggering condition in `config.py`. This triggering condition should conform to the ETCetera convention. This instance should contain:

- a method called "trigger", which returns True if triggering is necessary, and False otherwise.
- attribute "kmax" (int)
- attribute "h" (float)
- attribute "plant" (LinearPlant)
- attribute "controller" (LinearController)

Examples can be found in the "systems" folder. The user can also adjust the hyperparameter settings in `config.py`. These hyperparameters are described in `config.py`. The user can set multiple different stopping criteria:

- fixed runtime
- fixed amount of environment steps
- fixed amount of network expansions
- early stopping before the next network expansion on convergence of the SAIST-value

The user can enable or disable the different components described in Chapter 3, such as neural network expansion, action masking, smoothness, environment vectorisation, etcetera. The user can load a previously trained model in order to evaluate its performance.

B-3 Guidelines for hyperparameter tuning

During the experiments presented in Chapter 4, most hyperparameters were left unchanged from the first experiment to simplify the comparison. However, during other experiments where hyperparameters were adjusted, an increase in final performance and stronger convergence were observed. When training is unstable, one should reduce the learning rate. The initial horizon length should be set to a length which captures the most important sampling behaviour of the system. Shorter initial horizon lengths will result in faster initial learning, but performance could degrade if important sampling behaviour is not captured. In general, when the dimensionality of the system increases, one should increase one or multiple of the following hyperparameters:

- buffer size
- network width,
- horizon length
- training time
- number of parallel processes (if the hardware allows this)
- maximum amount of steps and/or patience between network expansions
- minimum and maximum amount of epochs and patience during behaviour cloning

Bibliography

[1] Dieky Adzkiya and Manuel Mazo Jr. Scheduling of event-triggered networked control systems using timed game automata, 2016.

[2] Richard Bellman. *Dynamic Programming*. Dover Publications, 1957.

[3] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[4] Johan Bjorck, Carla P. Gomes, and Kilian Q. Weinberger. Towards deeper deep reinforcement learning with spectral normalization, 2022.

[5] Yuri Burda, Harri Edwards, Deepak Pathak, Amos Storkey, Trevor Darrell, and Alexei A. Efros. Large-scale study of curiosity-driven learning. In *ICLR*, 2019.

[6] Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation, 2018.

[7] Gabriel de Albuquerque Gleizer and Manuel Mazo Jr au2. Computing the sampling performance of event-triggered control, 2021.

[8] Gabriel de Albuquerque Gleizer and Manuel Mazo Jr. Scalable traffic models for scheduling of linear periodic event-triggered controllers, 2021.

[9] Gabriel de Albuquerque Gleizer, Khushraj Madnani, and Manuel Mazo Jr. Self-triggered control for near-maximal average inter-sample time, 2021.

[10] Deepak Raj. Single-layer neural networks in machine learning (perceptrons), 2020. [Online; accessed 10-January-2022].

[11] MCF Donkers and WPMH Heemels. Output-based event-triggered control with guaranteed h infinity gain and improved and decentralized event-triggering. *IEEE Transactions on Automatic Control*, 57(6):1362–1376, 2011.

[12] WPM Heemels Heemels, MCF Donkers, and Andrew R Teel. Periodic event-triggered control for linear systems. *IEEE Transactions on automatic control*, 58(4):847–861, 2012.

[13] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

[14] Shengyi Huang and Santiago Ontañón. A closer look at invalid action masking in policy gradient algorithms, 2020.

[15] Xingxing Liang, Yang Ma, Yanghe Feng, and Zhong Liu. Ptr-ppo: Proximal policy optimization with prioritized trajectory replay. *arXiv preprint arXiv:2112.03798*, 2021.

[16] Mathworks. What is reinforcement learning?, 2008. [Online; accessed Januari 7, 2022].

[17] Manuel Mazo Jr, Adolfo Anta, and Paulo Tabuada. An iss self-triggered implementation of linear controllers. *Automatica*, 46(8):1310–1314, 2010.

[18] Manuel Mazo Jr, Arman Sharifi-Kolarijani, Dieky Adzkiya, and Christiaan Hop. Abstracted models for scheduling of event-triggered control data traffic. In *Control Subject to Computational and Communication Constraints*, pages 197–217. Springer, 2018.

[19] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[20] Kei Ota, Devesh K. Jha, and Asako Kanezaki. Training larger networks for deep reinforcement learning, 2021.

[21] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International conference on machine learning*, pages 2778–2787. PMLR, 2017.

[22] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration, 2018.

[23] Andrew M Saxe, James L McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*, 2013.

[24] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017.

[25] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.

[26] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[27] Samarth Sinha, Homanga Bharadhwaj, Aravind Srinivas, and Animesh Garg. D2rl: Deep dense architectures in reinforcement learning. *arXiv preprint arXiv:2010.09163*, 2020.

[28] Adam Stooke and Pieter Abbeel. Accelerated methods for deep reinforcement learning. *arXiv preprint arXiv:1803.02811*, 2018.

[29] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[30] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[31] Paulo Tabuada. *Verification and control of hybrid systems: a symbolic approach*. Springer Science & Business Media, 2009.

[32] Paulo Tabuada and Senior Member. Event-triggered real-time scheduling of stabilizing control tasks.

[33] Cheng-Yen Tang, Chien-Hung Liu, Woei-Kae Chen, and Shingchern D. You. Implementing action mask in proximal policy optimization (ppo) algorithm. *ICT Express*, 6(3):200–203, 2020.

[34] Edward L Thorndike. Animal intelligence: an experimental study of the associative processes in animals. *The Psychological Review: Monograph Supplements*, 2(4):i, 1898.

[35] William Homan Thorpe et al. *Origins and rise of ethology*. Heinemann Educational Books, 1979.

[36] Alan Mathison Turing. Intelligent machinery, 1948.

[37] unknown. Multilayer perceptron. [Online; accessed 18-February-2022].

[38] unknown. Python control systems library. [Online; accessed 18-January-2022].

[39] Manel Velasco, Josep Fuertes, and Pau Marti. The self triggered task model for real-time control systems. In *Work-in-Progress Session of the 24th IEEE Real-Time Systems Symposium (RTSS03)*, volume 384, 2003.

[40] Manel Velasco, Pau Martí, and Enrico Bini. On lyapunov sampling for event-driven controllers. In *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*, pages 6238–6243. IEEE, 2009.

Glossary

List of Acronyms

NCS	Networked Control Systems
ETC	Event-Triggered Control
STC	Self-Triggered Control
LTI	linear time-invariant
CETC	Continuous Event-Triggered Control
PETC	Periodic Event-Triggered Control
IST	inter-sample time
AIST	average inter-sample time
SAIST	smallest average inter-sample time
MAIST	mean average inter-sample time
TD	temporal difference
MDP	Markov Decision Process
DP	Dynamic Programming
GPI	Generalize Policy Iteration
SGD	stochastic gradient descent
GAE	Generalised Advantage Estimation
RL	reinforcement learning
DRL	deep reinforcement learning
PG	Policy Gradient
PPO	Proximal Policy Optimisation
TRPO	Trust Region Policy Optimisation
SDSS	state-dependent sampling strategy
ANN	artificial neural network
MLP	multi-layer perceptron