UiT The Arctic University of Norway

Faculty of Science and Technology, Department of Physics and Technology

Introducing Soft Option-Critic for Blood Glucose Control in Type 1 Diabetes

Exploiting Abstraction of Actions for Automated Insulin Administration

Christian Jenssen

FYS-3941 Master's thesis in applied physics and mathematics - 30 ECTS - July 2020

Abstract

Type 1 Diabetes (T1D) is an autoimmune disease where the insulin-producing cells are damaged and unable to produce sufficient amounts of insulin, causing an inability to regulate the body's blood sugar levels. Administrating insulin is necessary for blood glucose regulation, requiring diligent and continuous care from the patient to avoid critical health risks. The dynamics governing insulin-glucose are complex, where aspects such as diet, exercise and sleep have a substantial effect, making it a difficult burden for the patient.

Reinforcement learning (RL) has been proposed as a solution for automated insulin administration, with the potential to learn personalized solutions for insulin control adapted to the patient. In this thesis policy-based RL-methods for T1D management are investigated and a new method is developed; *Soft option-critic* (SOC) is designed to better account for differing situations affecting the blood glucose, using temporally extended actions called *options*. Further extensions of the method are implemented, using key elements from deep Q-learning algorithms.

The experiments are twofold; Several experiments are conducted to thoroughly assess the performance of SOC and its extensions on T1D in-silico patients: The first part of the experiments are done on the already solved environment *lunar lander* (LL) to analyze the merits of using options in the SOC-formulation. The second part consists of the diabetes experiments using a insulin-glucose simulator including scenarios with varying meals and bolus. The results show that SOC and its extension outperforms the benchmark algorithms on LL, learning options for improved sample-efficiency. On the diabetes experiments they performed comparable to the best benchmark model, beating the optimal baseline control method. The resulting policy was able to predict and account for meals, improving time-in-range (TIR) substantially.

Acknowledgments

First of all, I want to thank my advisor Fred. I would also like to express my sincerest gratitude to my co-advisor Jonas N. Myhre, for your patience and steering me in the right direction. Your continuous guidance has been invaluable for my work on this thesis.

Furthermore, I want to thank Miguel for all the enlightening discussions and all your knowledge that has been so willingly shared with me.

A big thank you to my fellow students for both your friendship and academic support. I'm truly grateful for the motivation and inspiration it has given me.

Finally, to my family and Madeleine: Thank you for your love and support.

Christian Jenssen,
Tromsø, July 2020.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The purpose of this thesis is to evaluate the potential for recent deep policy-based reinforcement learning methods to improve on blood glucose control in type 1 diabetes. Type 1 diabetes (T1D) is an auto-immune disease where the insulin-producing cells are damaged or destroyed [4,5]. As insulin is essential for the regulation of blood sugar levels, without treatment the body is unable to move blood sugar into the cells, resulting in high blood sugar levels. To accommodate for the lack of insulin production, treatment with injection of insulin or the use of an insulin pump is necessary. Controlling the blood sugar levels is a difficult task for patients, with the complex interactions within the body, and factors such as exercise, diet, stress levels and sleep affecting how much insulin is needed [6]. Additionally, optimal control varies from person to person. As this is a complex environment, requiring the need for personalized solutions, reinforcement learning (RL) methods has been proposed for solving such problems. Reinforcement Learning is an area of machine learning that focuses on how to take optimal actions within a complex and dynamic environment [2]. We design an agent and a reward signal that the agent can use to evaluate optimal actions, with the goal of maximizing some notion of long-term reward.

RL methods have earlier been proposed as possible solutions to improve insulin control [7–9].

Hierarchical reinforcement learning (HRL) has shown good performance for environments that have distinct domains in the state-space, or where the problem reasonably could be divided into sub-tasks. Having an all-encompassing policy that tries to optimize for all scenarios and factors such as meals, training and night time, requires much of the agent, with distinct ranges of insulin dosages required for each scenario. Especially between the basal rate and bolus there is a big difference between the optimal dosages when eating versus not.

HRL is an enticing proposal for T1D management since the agent could in theory autonomously create options that capture each distinct setting of the environment. Additionally, temporally extended actions could prove a great abstraction because of the delayed effect of primitive actions (insulin dosages) on blood glucose levels, the idea being that over a temporally extended action the effect on the state is more immediately connected to the high-level action.

1.1 Structure of the master thesis

Chapter 2 describes the problems facing type 1 diabetics and introduces the artificial pancreas as a potential solution for blood glucose control. Related work using reinforcement learning for T1D management is discussed in further detail at the final section.

Chapter 3 introduces reinforcement learning, describing Markov decision processes 3.3 as the underpinning framework for these methods. In 3.5 definitions needed for applying RL are introduced. The two main branches of RL - value-based- and policy-based methods are defined, and some of the most notable methods are introduced such as *Q-learning* and *REINFORCE*. This leads to deep learning-based methods that are able to handle environments with more complex state representation using neural networks as function approximators. After describing some notable deep RL methods, the paper introduces hierarchical reinforcement learning (HRL).

A major underpinning mathematical foundation for HRL is the *options framework*, which is a central component to soft option-critic. The final sections of chapter 3 introduces the methods that soft option-critic is based on, such as the *option-critic architecture* that extends the option-framework and *soft actor-critic*. Additionally actor-critic methods related to sac are described, which are also used in the experiments for benchmarks.

Chapter 4 offers a further description of the method and motivation behind soft option-critic including its implementation. The final sections extends the algorithm with elements from deep Q-learning-based methods.

Chapter 5 consists of the experiments, covering the experimental setup and results. The experiments are twofold; one part evaluates soft option-critic against state-of-the-art policy-based methods, analyzing the performance and option specialization.

The second part focuses on solving the diabetes environment, comparing SOC against the benchmark models. The same process for evaluation is followed as in the first part.

Chapter 2

Diabetes

Type 1 diabetes (T1D) is an autoimmune disease where the body is not able to produce insulin of its own [4, 5]. This occurs when the body's own autoimmune system destroys the insulin-producing beta cells in the pancreas. Insulin is an essential hormone for regulation of sugar levels in the body, turning glucose, meaning blood sugar, into energy for the body's own cells. If it goes untreated, this form of diabetes is deadly. Before insulin treatments were introduced most patients died within 2-4 years after being diagnosed. Even today diabetics have a slightly shortened life span [10].

As insulin is essential for the control of the blood sugar levels, without treatment the body is unable to move blood sugar into the cells, resulting in high blood sugar levels. To accommodate for the lack of insulin production, treatment with injection of insulin or the use of an insulin pump is necessary [6]. Controlling the blood sugar levels is a difficult task for patients, considering the complex interactions within the body, with factors such as exercise, diet, stress levels and sleep affecting how much insulin is needed.

The biological dynamics governing these interactions will be further introduced in the following section.

2.1 Dynamics of the pancreas-insulin system

The pancreas is a part of the body's endocrine system. The pancreatic tissue has hormone secreting cell groups called Langerhans islands, which contains of alpha cells, creating glucagon, and beta-cells, creating insulin. These hormones play a vital role for the cells' metabolism by regulating their energy supply [10]. The secretion of insulin is mainly regulated by the blood's concentration of glucose (blood sugar levels), going up when levels are high. The main task of insulin is to stimulate the uptake of glucose in cells. On the other hand, glucagon secretion goes up when glucose levels are low, and leads to an increased plasma concentration of glucose and fatty acids by mobilizing nutrients from the body's reserves.

Next section describes some of the currently used methods for administrating insulin dosages, including some of their limitations.

2.2 Current solutions for blood glucose control

Current solutions require the individual with T1D to measure glucose levels and estimating carbohydrate intake multiple times a day.

There are two main ways administrating insulin into the body, ether by injection with an insulin pen or by the use of an insulin pump [6, 11]. In this work we focus on continuous infusion of insulin with a pump.

2.2.1 Insulin pump

An insulin pump is a small medical device with an insulin reservoir connected to a catheter inserted under the skin of the abdomen [11]. The pump dispenses specific amounts of rapid-acting insulin, where the amount prescribed is determined by consulting with a doctor. This steady rate of insulin dosage is known as the *basal rate*. To control for the effects of meals on the blood sugar levels, the pump handles another dose based on the amount of carbohydrates eaten, specified by the individual. This dosage is known as a *bolus* dosage [11] and is usually given before the meals [12].

Current ways of treating the disease proves a laborious task and requires immense discipline from the individual, where slip-ups could prove dangerous, even fatal. There have been great advances in the development of the CGM and insulin pump [7], yet regular management from the patient and caretakers are still necessary. An automated system for T1D management would have the potential of greatly improving quality of life for type 1 diabetics, both by alleviating the need for intervention from the diabetic and the potential for improving the calculations of correct insulin dosages.

2.2.2 Artificial pancreas

An artificial pancreas is an automated system for insulin control, that attempts to emulate the functionality of a real pancreas [13]. It consists of three components as illustrated in figure 2.1: i) A sensor for continuous glucose monitoring (CGM), ii) an insulin pump delivering system and iii) a control algorithm for insulin dosage amounts.

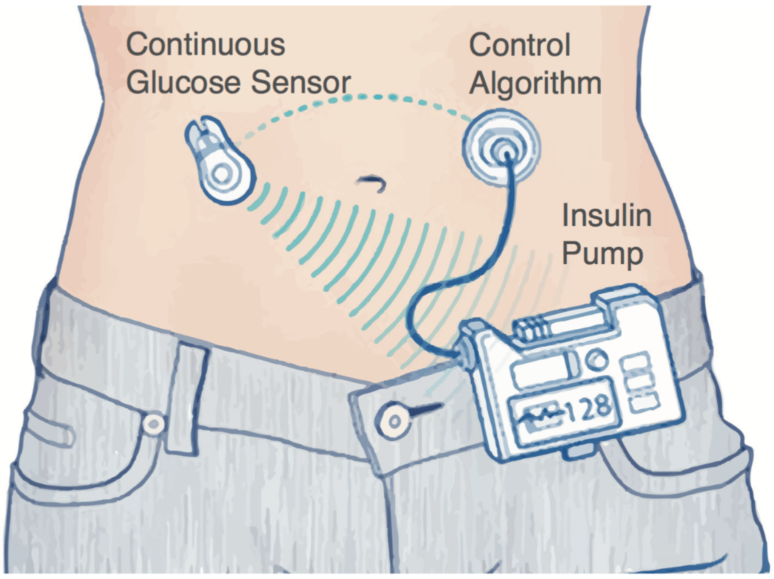


Figure 2.1: A figure illustrating the components of an artificial pancreas [1].

As mentioned earlier the physical components i) and ii) have seen great improvements over the years. The major challenge for creating an automated T1D lies within the design of a successful and robust control algorithm [13]. It represents the key component of the artificial pancreas, and acts as the messenger between the physical components of the system [7].

There are two major candidates that have been intensively studied for closed-loop calculation of insulin dosage: Proportional integrative derivative (PID) methods and model predictive control (MPC).

2.2.3 PID

PID uses the difference between the actual glucose concentration and the optimal glucose concentration. This difference denotes the error, which is integrated over time to obtain the accumulated error over a time period, then the rate of change of these errors is calculated. With these terms the

PID-controller estimates the required doses that minimizes these errors - continually attempting to move the glucose levels to the desired concentration [13].

2.2.4 MPC

MPC assumes a glucose-insulin dynamical model that can predict future glucose concentrations given known values for current glucose, insulin delivery and food intake [7, 13]. It recommends insulin infusion rate based on minimizing the difference between a desired glucose level and the predicted concentration obtained from the model [7].

2.2.5 Limitations of PID and MPC

Both approaches suffer some shortcomings in their design and performance. These methods are not truly adaptive in the sense that they adjust their approach and learn based on data. Both are static models that are based on heuristic tuning. Naturally, everything affecting the system can't be accounted for in that case [13]. PID is a purely reactive method, lacking the theoretical foundation of a biological model. MPC is based on an imperfect model of the biological dynamics describing the fluctuations of blood glucose levels. Additionally the model does not account for external disruptions to the system such as mean intake or physical activity.

2.3 Reinforcement learning for controlling type 1 diabetes

The biological interactions within the body are complex and subtle. Designing mathematical models that feasibly describe the biological processes, especially while accounting for factors such as stress and physical activity is a challenging [13] endeavour. With the general health, metabolism rate and lifestyle varying greatly between people, additionally factors such as stress and general lifestyle change over time for each individual. Thus, one-size-fits-all algorithms are not the best directions for further development. In contrast, methods that can adapt to these inter- and intra-individual factors to provide a personalized solution is greatly sought after. RL models prove a good match in theory, because they learn by interacting with the environment. In this setting based on the individual's biological system, meaning it does not need to assume an imperfect model that potentially limits the performance.

Additionally since RL methods are data-driven, they could adapt to a changing lifestyles over time. In practice, RL has shown great results for many complex environments, such as *AlphaZero* in chess [14], *OpenAI* for the online multiplayer game Dota 2 [15], illustrating the enormous potential for RL as general learning algorithms in dynamic systems.

Chapter 3

Background: Reinforcement Learning

3.1 Learning from observations

How to effectively learn from data? The process of answering this question has been the driving force for advancement of methods in mathematics and statistics for millennia. The development of computers and processors, has laid the foundation for new methods leveraging these advancements. **Machine learning** is the field that encompasses this question, and lies in the intersection between mathematics, computer science and applied statistics [16].

As such, machine learning is a field that considers a *computational approach* for learning to perform a specific task, without being explicitly programmed for the task at hand. In essence, it illustrates a paradigm shift where instead of designing hand-crafted solutions requiring specific domain knowledge for a problem, the algorithms leverage data by learning automatically, being able to generalize across new observations and adapting to the task in mind. Encompassing all machine learning methods is the use of *training data*.

Broadly speaking, the general learning process can be described as follows:

1. Create a mathematical model defined by some parameters
2. Design an algorithm that optimizes the parameters of the model based on a performance criterion, often know as the *loss function*
3. Iterate over the training data using the algorithm, improving the per-

formance criterion, leveraging the processing power of computers

The details of the process and *how* this is achieved depends on the type of task in mind. Roughly speaking, there are four main branches of machine learning based on the problems they try to solve and what we want to achieve.

Supervised learning is learning from observations where we have the "ground truth", also known as *labels*. Supervised learning is concerned with finding the mapping from observations to ground truth.

Formally we have a training set

$$(X, Y) = \left(\{x_1^{(i)}, \dots, x_l^{(i)}\}, \{y_1^{(i)}, \dots, y_l^{(i)}\}\right), \forall i = [1, N], \tag{3.1}$$

where X denotes the *observations* and Y is the corresponding labels, with N samples, forming an input-output connection $X \rightarrow Y$. In essence, supervised learning is concerned with finding a function f that maps the training data to the correct labels $Y = f(X)$ [17].

To illustrate this concept, think of the scenario where a doctor has multiple x-ray images from different patients and the knowledge whether they had cancer or not. In this instance, the training data would be the x-ray images and labels would be their actual diagnosis. What is of interest is to find the patterns connecting X to Y, such that the algorithm could generalize to new samples, where the labels are unknown.

Unsupervised learning is concerned with modelling the underlying structure of data, finding the inherent patterns. In contrast to supervised learning, the labels are unknown, hence the name *unsupervised.* Naturally, **semi-supervised learning** uses a combination of labeled and unlabeled data, often found useful when obtaining labels is time-consuming and/or expensive.

Reinforcement learning (RL) essentially pertains to learning by interaction to achieve some goal. As opposed to supervised learning, the emphasis is on learning by trial-and-error, where any exemplary supervision or engineered models are not required [2]. This branch of machine learning will be the main focus, as it is more aligned with working on blood glucose control problems. The reason is that RL is more suitable when utilizing a diabetes simulator, where pre-labeled training data does not exist. The following chapter will introduce the building blocks and key concepts of reinforcement learning.

3.2 Building Blocks for Reinforcement Learning

At its core, reinforcement learning is a computational approach to learning from interaction with an environment to achieve a long-term goal. The components central for this setting will now be introduced.

The **agent** defines the *learner and decision maker* of the environment, and the way it acts is determined by its **policy** π. *The environment* compromises everything outside the agent which it interacts, and is encapsulated within the **state** s, which conveys the current condition of the environment at a particular time t [2]. Informally, it can be viewed as the dynamic stage in which the actor acts. To learn a useful and goal-directed policy, the agent receives a scalar **reward** signal r when interacting with the environment based on the action taken and current state of the environment s. Naturally then, the goal is to maximize the reward received over the long run. The reward is essential for learning an "optimal way" of acting, guiding the agent in its learning process [2, 18].

The agent-environment dynamic forms a cyclic relationship where the agent acts in a state s, taking action a and the environment responds by presenting a new state s'[1] while receiving a reward r [2]. This "eternal dance" repeats itself indefinitely, over a sequence of discrete time-steps $t = 0, 1, 2, \ldots, T-1$, producing a collection of transitions defined as a *trajectory*

$$\tau_T = \{s_0, a_0, r_1, s_1, a_1, r_2, s_2, \ldots, r_{T-1}, s_{T-1}\} \tag{3.2}$$

where each transition denotes a *sars*-tuple $\{s_t, a_t, r_{t+1}, s_{t+1}\}$. Figure 3.1 illustrates this sequential interaction-dynamics, mapping situations to actions $S \to A$ with the goal of maximizing a scalar reward signal [2, 18].

This captures the essential aspects of an agent interacting with an environment, but it still remains *how* to learn a goal-directed policy. Additionally, how would one even begin to compare whether a policy is optimal or not? Before diving into these questions, it is useful to formalize the building blocks introduced in this section in a more specific and mathematical fashion. The framework that has been found to be very useful in this context - **Markov Decision Processes** (MDPs).

[1] s, a, r and s' are used interchangebly to mean s_t, a_t, r_{t+1} and s_{t+1} respectively.

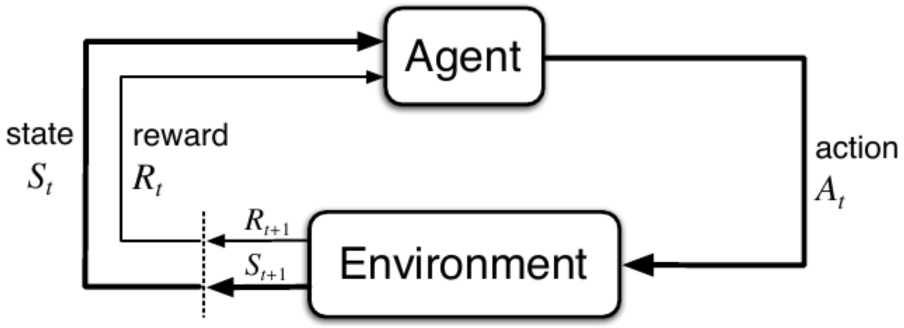


Figure 3.1: A figure showing the continuous interaction between an agent and the given environment. At each time step t the agent receives a reward and the state of the environment. Based on this (and what it has already learned) it performs an action which changes the state. [2]

3.3 Markov decision processes

Markov decision processes (MDPs) are a formalization of sequential decision making, functioning as the fundamental framework for RL [2]. MDPs captures the problem of learning within an interactive environment to achieve a goal, providing a mathematical framework for modeling decision making problems. Specifically, it is fully specified by a 4-tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P})$ describing all the "moving parts" necessary for agent-environment context.

The state-space $\mathcal{S}$ defines the set of possible states, i.e $s \in \mathcal{S}$. Conversely, $\mathcal{A}_s$ defines the set of available actions in state s, while $\mathcal{R}$ is the set of possible rewards. For simplicity, it is assumed that all actions are available in all states without..., $a \in \mathcal{A}_s = \mathcal{A}$. Thus, $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ captures the *static* components of MDPs, describing the "playing rules" which are often known or designed by the RL-engineer.

The *dynamics* of the environment is specified by $\mathcal{P}$, and mathematically encapsulates the transition probability model of the environment. The notation for probability transitions between states is defined as

$$p(s', r|s, a) \doteq P(S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a), \tag{3.3}$$

which denotes the probability of receiving reward r and moving to state s', given the current state s and action a. Naturally, the total probability is given as

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1, \quad \forall s' \in \mathcal{S},\ a \in \mathcal{A}(s). \tag{3.4}$$

For many problems both the reward signal and environment are stochastic, complicating the learning process when the same action in state s produce different rewards. To include these settings in the MDP formulation, we define them as stochastic variables S_t, A_t, R_t where s, a, r are their realizations.

3.3.1 Markov Property

The state encapsulates all the information the agent obtains from the environment, e.g. from sensors and are the basis for choosing actions. The key regarding MDPs is that the state contains information about all aspects of the past agent-environment interaction that makes a difference for the future [2]. This concept is known as the *markov property* and implies that the next state s' *only* depends on the current state s and action a [19]. Formally, MDPs satisfies the equation:

$$\begin{aligned} p(s'|s,a) &\doteq P(S_{t+1} = s'|S_t = s_t, A_t = a_t, \ldots, S_0 = s_0, A_0 = a_0) \\ &\doteq P(S_{t+1} = s'|S_t = s, A_t = a). \end{aligned} \tag{3.5}$$

Model-based RL

Most RL-algorithms assume that the dynamics of the problem satisfies the Markov property, even though the probabilities that characterize it are not known. Model-based RL are methods based on exploiting the dynamics $\mathcal{P}$ of the environment. To achieve this the $\mathcal{P}$ has to be pre-specified *or* learned in parallel with the agent. If there exists prior knowledge of the dynamics this can improve training ref. when incorporated with the agent. Although if $\mathcal{M}$ is fully known, using RL-methods are redundant since the best policy can be calculated directly. Additionally, in the case where $\mathcal{P}$ is estimated, actually learning a good approximation of the dynamics could be even more difficult than directly finding a good policy. For most people, learning to drive a car is easier than having a complete understanding of the physics governing the movements, and dynamics of the engine.

Model-free RL

Model-free RL methods do not make any assumptions of the environment dynamics, expect that they satisfy the Markov property and can be described as an MDP. These methods are model-free, and in this way are generic and applicable to more settings.

The dynamics governing the pancreas are complex and varies between individuals, creating a model in this setting is more difficult than just learning an optimal policy. Therefore the focus in this thesis is on model-free RL methods.

3.4 Rewards and Returns

As the reward function is something that has to be designed and defined by the RL practitioner, this naturally has substantial effect on the learning since it is the direct feedback signal for the agent's performance [2]. Naturally we'd like to assign simple reward functions that are connected directly with the goal we want to achieve. Additionally, using complex reward functions with the intent of helping the agent can actually lead the agent to exploit the environment in surprising ways, producing behavior that is counter-productive to achieving the goal - in essence introducing bias [2].

An example is the game of chess. Taking pieces and not losing your own is conducive to winning, thus it seems natural to augment the reward function to accommodate for this. But it might actually be detrimental to winning since the goal isn't to take all the pieces - only the king. Thus the agent might in some scenarios miss opportunities to sacrifice pieces for a check mate and instead be biased towards taking material. In a similar vein, using supervised learning on grand master (GM) games biases the agent to play like a GM, which is not necessarily the best policy. A Quote from Sutton summarizes this eloquently: "The reward signal is your way of communicating to the robot *what* you want to achieve, not *how* you want it achieved" [2].

In essence, representing the reward functions simply and directly with respect to the goal we want to achieve is the preferred route. For example a possible reward function for chess would be $1, 0, -1$ for victory, draw and loss respectively. As with many of the dynamic environments we want to solve, **credit assignment** is a challenge; which moves were good and which were bad? In addition to this, the only time the agent gets any feedback on its performance is when the game is finished. This aspect is called *sparse rewards*, and a natural solution is to let the agent have an intrinsic motivation for exploring the environment, formalized by defining in some way an *intrinsic reward function.*

As mentioned, the policy π describes the "decision-making" part of the agent. Formally, it specifies a mapping from situations to actions $S \to A$ which can be written as $\pi(\cdot|s)$. In practice, it is a conditional probability distribution. Both real physical domains such as robot control and simulated settings such as games on a computer are suitable environments, the only requirement being that the agent is able to interface with the environment.

The agent does not know which actions are optimal and therefore has to explore the environment and learn the effects of each action in the given situation. Formally, the policy defines the probability of taking a given action a in state s: $\pi(a|s)$.

The reward signal at a given time-step R_t is the main basis for optimization of the policy, but in and of itself only captures the *immediate value* of being in a state s and taking a certain action a. For many problems both the reward signal and environment are stochastic, complicating the learning process when the same action in state s produce different rewards. As the end goal is to maximize *long-term* reward, a notion of "value" is used instead. In essence the value of a state s is the total reward an agent can expect to earn when starting from s. Thus action selection is based on the judgment of value - choosing actions that result in states of highest value, since these states produce the highest reward in the long run. A more thorough description of the underlying framework for RL is given in section 3.3

$$\tau_T = \{S_0, A_0, R_1, S_1, A_1, R_2, S_2, \ldots, R_{T-1}, S_{T-1}\}.$$

where T denotes the length of τ and a transition is defined as a tuple $S_t, A_t, R_{t+1}, S_{t+1}$. For each time-step t the agent is given a representation of the environment defined as the state S_t, performs an action A_t and receives a new state S_{t+1} and a reward signal R_{t+1} illustrated by fig. 3.1, where $S_t \in \mathcal{S}$, $A_t \in \mathcal{A}(s)$ and $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$.

The function p describing the dynamics of MDPs is defined as

$$p(s', r|s, a) \equiv P(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a), \quad \forall s',\ s \in \mathcal{R},\ a \in \mathcal{A}(s), \tag{3.6}$$

where for *finite* MDPs, the sets $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ each have a finite number of elements. Usually the state is represented as a vector of features, where each feature represent a characteristic of what defines the environment.

With the basis of RL defined (MDPs), the following section will now describe the concepts needed for applying RL in practice.

3.5 RL in practice

3.5.1 Episodic vs continuing environment

As stated earlier in section 3.1 an agent's goal is to maximize some notion of long-term reward. Formally the agent seeks to maximize the expected return G_t which is determined by the sequence of rewards during an episode [2]. In the simplest case it is defined as the sum of all rewards

$$G_t \doteq R_{t+1} + R_{t+2} + \ldots + R_T = \sum_{k=t}^{T-1} R_{k+1}. \tag{3.7}$$

Some environments have defined end states where the trajectory τ_T (episode) ends. This is usually the case for games and other popular environments often used for training, and these are defined as *episodic tasks* [2]. But for many real-life problems there is no clearly explicit end, these are defined as *continuing tasks.* The return (3.7) makes sense for episodic tasks, but for continuing tasks with $T \to \infty$ the return could approach infinite as well. A more general definition that encompasses both types is defined by the use of *discounting.* According to this approach the agent tries to maximize the expected *discounted* return:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{k-1} R_{t+k} = \sum_{k=0}^{\infty} \gamma^t R_{t+k+1}. \tag{3.8}$$

where $\gamma \in [0, 1]$ is the discount rate. With $\gamma < 1$ and bounded rewards the return is finite even though it is a sum of infinite number of terms. Thus the return is defined for continuing tasks as well.

The discount rate determines how much we value rewards in the future. For low values of γ the agent maximizes immediate rewards while it becomes more 'farsighted' as $\gamma \to 1$. This makes sense as it usually is more valuable to obtain reward that is accessible right now, compared to potential future reward that the agent might not even get. An example to illustrate this point is with interest: money you earn now will accumulate interest and is more valuable than the same amount at a later time.

With these definitions we delve into the two main branches of learning with RL, value-based- and policy-based methods, introducing some of the main algorithms for each branch.

3.6 Value-based methods

Value-based methods are RL algorithms that involve the use of value functions [2]. Formally, the value function v_π of a state s given that the agent will follow policy π thereafter is defined as

$$v_\pi(s) \equiv E_\pi[G_t|S_t = s] = E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a], \ \forall s \in \mathcal{S} \right]. \quad (3.9)$$

Similarly the action-value function $q_\pi(s, a)$, which denotes specifically the value of taking action a in state s and then follow π thereafter, is defined as:

$$q_\pi(s,a) \equiv E_\pi[G_t|S_t = s, A_t = a] = E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right]. \quad (3.10)$$

A fundamental property of these equations ((3.9),(3.10)) is that they satisfy a recursive equation known as *the Bellman equation*:

$$\begin{aligned} v_\pi(s) \equiv E_\pi[G_t|S_t = s] &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma E_\pi[G_{t+1}|S_{t+1} = s']] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma v_\pi(s')], \ \forall s \in \mathcal{S}. \end{aligned} \quad (3.11)$$

Most value-based methods have a basis with the Bellmann equations (3.11) at its core.

The following subsection will take a look at Q-learning, on of the most central value-based methods.

3.6.1 Q-learning

Q-learning is a control algorithm that iteratively approximates the optimal action-value function q_* [2, 20]. It is defined as:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)], \quad (3.12)$$

where α is the learning rate deciding how big of a step to take when updating the Q-value. The change in value is based on the *temporal-difference loss* $L = R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)$ where $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$ is the target y and $Q(S_t, A_t)$ denotes the current estimate. By minimizing L this iteration converges to the optimal action-value that satisfies the Bellman equation (3.11) since the difference between target and current estimate approaches 0.

3.7 Policy gradient methods

Policy gradient methods are methods that learn a *parameterized* policy $\pi(a|s,\theta)$ for action selection [2,21]. The policy parameters θ are trained based on the gradient of a scalar performance measure $J(\theta)$ with respect to θ. For these methods we maximize the performance such that the updated for θ approximate gradient ascent in J:

$$\theta_{t+1} = \theta_t + \widehat{\nabla J(\theta_t)}, \tag{3.13}$$

where $\widehat{\nabla J(\theta_t)}$ is a stochastic estimate where the expectation approaches the true performance gradient [2]. In the episodic case the performance is defined as:

$$J(\theta) \equiv v_{\pi_\theta}(s_0),$$

where v_{π_θ} is the true value function for the policy.

For continuous action space problems it is impractical or impossible to calculate probabilities for each action. Instead the policy learns the statistics of a probability distribution such as the Gaussian:

$$\pi(a|s,\theta) = \frac{1}{\sigma(s,\theta)\sqrt{2\pi}} \exp\left\{-\frac{(a-\mu(s,\theta))^2}{2\sigma(s,\theta)^2}\right\}. \tag{3.14}$$

3.7.1 REINFORCE

REINFORCE is a Monte Carlo policy gradient method, therefore the strategy for obtaining good estimates for ∇J is to sample trajectories τ as an estimate for the expectation [22].

$$\begin{aligned}
\nabla J(\theta) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla\pi(a|s,\theta) \\
&= E_\pi\left[\sum_a q_\pi(S_t,a)\nabla\pi(a|S_t,\theta)\right] \\
&= E_\pi\left[\sum_a \pi(a|S_t,\theta) q_\pi(s,a) \frac{\nabla\pi(a|S_t,\theta)}{\pi(a|S_t,\theta)}\right].
\end{aligned} \tag{3.15}$$

Replace a by the sample $A_t \sim \pi$ and using the fact that $E_\pi[G_t|S_t,A_t] = q_\pi(S_t,A_t)$ we get:

$$E_\pi = \left[G_t \frac{\nabla\pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}\right]. \tag{3.16}$$

Eq. (3.16) is a stochastic gradient obtained by sampling, whose expectation approaches ∇J. Therefore it yields the **Reinforce** update:

$$\theta_{t+1} = \theta_t + G_t \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}. \tag{3.17}$$

$\frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}$ is the direction that most increases the probability of taking action A_t when in state S_t. The update is proportional to the return, which implies that the parameters move most in the directions for actions that yield the highest return.

3.8 Deep reinforcement learning (DRL)

With the recent development of deep learning [17, 23], similar methodologies have been introduced to reinforcement learning. In RL-problems the state-space is often continuous, and representing the action-value Q or policy π using tabular methods is computationally expensive for many real-life problems By introducing neural networks in RL as function approximators for Q the performance is improved for certain problems such as playing Atari using raw pixels as input [24].

The following subsections describes deep Q-learning

3.8.1 Deep Q-learning (DQN) and double DQN (DDQN)

DQN is an off-policy learning algorithm based on Q-learning where the action-value estimation is based on a neural network. We can train a Q-network by minimizing the loss between the current action-value estimate and the target (alternative estimate) [24].

$$L_i(\theta_i) = E_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s_i,a;\theta_i))^2\right], \tag{3.18}$$

where the target is defined as

$$\begin{aligned} y_i &= r_i && \text{if } s_i \text{ is terminal,} \\ y_i &= r_i + \max_{a'} \gamma Q(s',a';\theta^-) && \text{if } s_i \text{ is non-terminal.} \end{aligned}$$

The weights for the target θ^- are held fixed while optimizing the loss function, which helps with the stability when training since it is difficult to train with a moving target. Let the target weights be the previous version of the weights at iteration i: $\theta_i^- \leftarrow \theta_{i-1}$.

An important assumption for many deep learning algorithms is that the data samples are independent, but in reinforcement learning we usually get a sequence of states that are highly correlated. An important addition to the *DQN*-algorithm which alleviate this problem is *Experience replay.* It is a technique where the agent's experience $e_t = (s_t, a_t, r_t, s_{t+1})$ is stored in a replay memory $\mathcal{D}$ with a chosen capacity N. Thus we are able to randomly sample a batch of transitions which includes earlier experiences for training. This has the effect of smoothing out the training and avoiding oscillation and divergence in the parameters.

Because of the max operation in the Q-learning algorithm, *DQN* has the tendency to overestimate the action-values Q, which often leads to negative effects on the performance [25]. *Double Q-learning* is similar to *DQN*, except that it decouples the action selection from the value estimation for the target. The online network evaluates the greedy policy while the target network estimates its value.

$$Y_t^{\text{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \max_a Q(S_{t+1}, a; \theta_t); \theta_t^-), \tag{3.19}$$

where the target network parameters θ^- are updated to be a copy of the online network parameters θ at every τ step.

3.8.2 Proximal Policy Optimization (PPO)

When learning a policy the distribution of states and rewards change in sync with the variable policy. This poses a difficult problem for reinforcement learning algorithms to handle and is an important factor for the instability while training.

PPO is policy-based deep RL-algorithm that uses a clipped objective function which ensures that the policy does not change too much at each training step to avoid instability [26]. An added benefit is that PPO is able to perform multiple epochs of mini-batch updates, compared to REINFORCE which only perform one update per data sample.

The loss is defined as

$$\begin{aligned} Surr_1 &= r_t(\theta)\hat{A}_t \\ Surr_2 &= clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \\ L^{CLIP}(\theta) &= \hat{E}\left[\min(Surr_1, Surr_2)\right], \end{aligned} \tag{3.20}$$

where $r_t(\theta) = \dfrac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ and ϵ is a hyper-parameter. The loss is effectively penalizing changes to the policy that move $r_t(\theta)$ outsize the interval $[1 - \epsilon, 1 + \epsilon]$.

PPO has shown great efficiency and performance in multiple tasks such as Dota 2, Starcraft, AlphaZero [14, 15]. For real-life environments and problems, the policy-based methods such as PPO are sample-inefficient, because they do not reuse experience.

The subsequent chapter introduces Hierarchical reinforcement learning (HRL) which attempts to tackle these challenges.

3.9 Hierarchical Reinforcement Learning

Intelligent decision making often involve planning at different time scales [3]. It is natural for humans to make plans in an hierarchical structure, by first making high level decision or plans and then "move down the hierarchical tree" into more granular actions and time scales. Consider a young teenager making the big decision of what to study at college. A high level decision would be to decide whether to study for STEM fields, or humanities et cetera. The student takes into account factors such as their interests, strengths, expected future earnings, location, grade requirements and involve foresight of future work market, economy, risk of taking on student debt and actually achieving required grades. After deciding on a field the student needs to select which courses to take to achieve the sub-goal which in this case is the grade requirements, and then plan on how to best learn the curriculum accounting for day-to-day factors such as diet, sleep and trade-off between studying and allocating time for other important things in life, culminating into actions taken at the most granular level. This example illustrates the necessary temporal abstraction at different levels of time-scale for long-term planning. Notice that at each level of temporal abstraction, vastly different 'features' of the 'state space' are important when making decisions - e.g. expected future earnings as a factor for deciding what to study versus day-to-day choices for achieving success in certain courses etc. naturally, structuring the decision process in this way is a sound proposition for improving learning and long-term planning in complex and dynamical environments [3, 27].

Hierarchical reinforcement learning (HRL) is a natural proposal to these kinds of settings, allowing multiple policies to focus on different high level goals, improving planning and learning. More concretely, HRL is able to 'partition' the planning and learning at different timescales, by using a hierarchical structure of policies. Thus the higher level policies in the hierarchy is able to plan more efficiently over longer timescales, selecting higher level 'actions' lasting multiple time-steps compared to the lowest level policies that select the actual primitive actions that are taken in the environment at every time-step t.

To represent this hierarchical structure an extension on the notion of actions was developed, capturing the concept of temporally extended actions - *the options framework.* We have chosen to focus on options.

3.9.1 Options framework

What constitutes an action? In Markov decision processes (MDPs) which is the basis of RL, a notion of temporally extended actions does not exist as they are based on discrete time steps. An action at time t affects the state and reward at time $t+1$. Thus there is no notion of action persisting over a variable period of time, restricting the agent in taking advantage of simplicities and efficiencies that naturally occurs at higher levels of temporal abstraction [3]. The options framework augments the action space by allowing temporally extended actions, this expansion of the concept of actions is called *options.* The framework is based on the theory of *semi-Markov decision processes* (SMDPs) which is a continuous time generalization of MDPs [28]. A limitation of SMDP theory is that the temporally extended actions are treated as indivisible and unknown units, this is incompatible with the idea of options since the agent need to be able to make and modify decisions at multiple overlapping time scales, examining temporally extended actions at an increasing level of granularity. Thus the key concept for the option framework is the interplay between MDPs and SMDPs. Specifically the framework is based on discrete-time SMDP, where the underlying base system is an MDP. Then we can define options that potentially last a multiple number of discrete steps that are not indivisible. Options can be described in terms of policies in the underlying MDP which act at every time-step.

Figure 3.2 illustrates this interplay between MDPs and SDMPs clearly. Each discrete step in the SMDP constitutes multiple steps (and primitive actions) of the underlying MDP, where options are the temporally extended actions selected at each step in the SMDP.

3.9.2 Defining an option

Options consist of three components: a policy $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0,1]$, a termination condition $\beta : \mathcal{S}^+ \rightarrow [0,1]$, and an initiation set $\mathcal{I} \subseteq \mathcal{S}$ [3]. An option is fully determined by these three components $o_{\mathcal{I},\pi,\beta} = \langle \mathcal{I}, \pi, \beta \rangle$ and

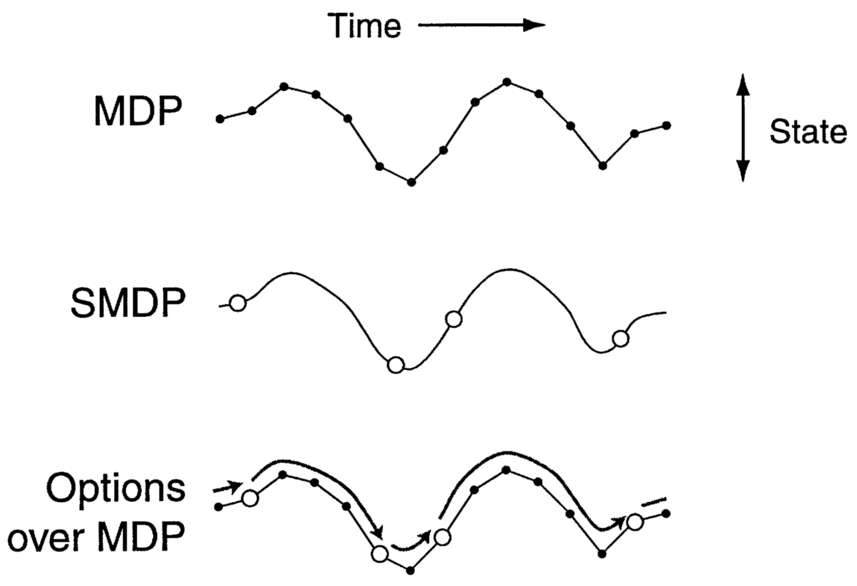


Figure 3.2: A figure showing the connection between MDP, SMDP and options [3].

its availability in state s_t exists only if $s_t \in \mathcal{I}$. Conversely $\beta(s_t)$ determines the probability of terminating the option o at the current state. Finally, π is the primitive policy that selects actions based on the underlying MDP. In essence, a given option o is selected where $s_t \subseteq \mathcal{I}$, next action a is selected based on the policy $\pi(s_t, \cdot)$. The environment transitions to a new state s_{t+1} where the option either terminates with probability $\beta(s_{t+1})$ and then selects a new option, or continues, taking action a_{t+1} based on $\pi(s_{t+1,}.)$. The available options from a state s is implicitly determined from the options' initiation sets, the set of these options is defined as $\mathcal{O}_s$ for each state $s \in \mathcal{S}$. The set of all options is defined as $\mathcal{O} = \cup_{s \in S} \mathcal{O}_s$.

Actions can be considered as a special case of options where the option always lasts exactly one step $\beta(s) = 1$, $\forall s \in \mathcal{S}$ [3]. Therefore we may view the agent's decision-making to solely be based on selecting between options, were some last a single time step (primitive actions) and some last multiple time steps. These definitions keep options as similar to actions, while still allowing temporally extended actions.

Conventional Markov options base the decision of terminating the option solely on the state s_t through the termination condition $\beta(s_t)$ [3]. Although, in certain scenarios it can be useful for options to terminate after a certain amount of time, even though the agent failed to reach any particular state. Such policies are defined as semi-Markov policies, where the termination condition β is also dependent on the sequence of transitions since the option was initiated. This sequence is called the history $h_{t\tau}$ and is defined as the set of all transitions from time t when the option o was initiated to time τ.

With the basics of an option defined, we will now look at the generalizations that follow from the equations used in RL, such as action-value functions, expressed within the options framework.

3.9.3 Policies over options

So we have multiple options, but how does the agent base the decision of option selection? Similarly as policies over actions, *policies over options* are defined as $\mu : \mathcal{S} \times \mathcal{O} \rightarrow [0,1]$, which selects an option $o \in \mathcal{O}_{s_t}$, according to policy probability distribution $\mu(s_t, \cdot)$ [3]. The policy over options μ can be represented in terms of each option's primitive actions (i.e "expand" or flat out the hierarchy of option selection from the level of μ), thus determining a conventional policy over actions defined as *flat policy*, $\pi = flat(\mu)$ [3, 29].

The value of a state $s \in \mathcal{S}$ under a semi-Markov flat policy π is defined as the expected return given that π is initiated in s:

$$V^{\pi} \equiv E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \mid \mathcal{E}(\pi, s, t)\}, \tag{3.21}$$

where $\mathcal{E}(\pi, s, t)$ denote the event of π being initiated in s at time t [3]. Similarly the value of a state under policy μ can be defined in terms of its flat policy: $V^{\mu}(s) \equiv V^{flat(\mu)}(s)$, $\forall s \in \mathcal{S}$

The corresponding generalization for action-value functions is option-value functions, $Q^{\mu}(s, o)$, the value of taking option o in state $s \in \mathcal{I}$ under policy μ. It is defined as

$$Q^{\mu} \equiv E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \mid \mathcal{E}(o\mu, s, t)\}, \tag{3.22}$$

where $o\mu$ the *composition* of o and μ denotes the semi-Markov policy that first follows o until it terminates and then starts choosing according to μ in the resultant state. Additionally we define $\mathcal{E}(o, h, t)$ as the event of o *continuing* from h at time t, where h is a history ending with s_t.* This completes the general framework for options

3.9.4 Learning with options

Analogous terms for reward and transition probabilities are well defined from existing SMDP theory [3]. They are given as:

$$r_s^o = E\{r_{t+1} + \gamma r_{t+2} + \dots \gamma^{k-1} r_{t+k} \mid \mathcal{E}(o, s, t)\}, \tag{3.23}$$

where $t + k$ is the random time at which o terminates. The probability of terminating current option o while transitioning from state s to s' is

$$p_{ss'}^o = \sum_{k=1}^{\infty} p(s', k)\gamma^k, \; \forall s' \in \mathcal{S}, \tag{3.24}$$

where $p(s', k)$ is the probability that the option terminates in s' after k steps. γ has the effect of weighing transitions that use many steps less. Since $p_{ss'}^o$ accounts for multiple steps k of reaching state s' from s and terminating o, this type of model is defined as a *multi-time model* [3, 30, 31]. Using multi-time models, the Bellman equations (3.11) can be written in terms of options:

$$V^\mu(s) = \sum_{o \in \mathcal{O}_s} \mu(s, o) \left[r_s^o + \sum_{s'} p_{ss'}^o V^\mu(s_{t+k}) \mid \mathcal{E}(o, s, t) \right] \tag{3.25a}$$

$$Q^\mu(s, o) = r_s^o + \sum_{s'} p_{ss'}^o \sum_{o \in \mathcal{O}_s} \mu(s, o) Q^\mu(s', o'). \tag{3.25b}$$

These definitions enable us to make natural extensions to regular RL algorithms and methods to the SMDP domain that apply to options. Unfortunately, conventional methods based on SMDPs pose limitations due to the treatment of options as indivisible units [3]. SMDP methods for semi-markov options are limited in the sense that an option has to follow through until termination before evaluation. In essence, they ignore what happens in-between the larger steps of the SMDP.

A potentially more powerful way is to focus on methods that take advantage of the interplay between MDPs and SMDPs, by looking inside the options. More specifically, we allow options to be interrupted before they'd terminate naturally, re-evaluating whether to continue with current option at each time step. Such options are called *interrupting options* [3]. Methods that learn about options from experiences within the SMDP are defined as *intra-option learning methods*. They allows us to take advantage of the underlying MDPs of options, allowing off-policy temporal-difference learning, even for the options not currently being used [3, 32]. Thus the Intra-option methods are potentially more efficient since they make use the transitions within the SMDP, giving way to more training examples and improving training.

There are many Intra-options methods developed, but we'll only delve into *Intra-option Q-learning* since it lies at the core of the *Soft Option-critic* .

3.9.5 Intra-option Q-learning

Similarly to regular Q-learning, *Intra-option Q—-learning* makes use of the *Bellman equations*, only with modified value function. With the new notation for value- and option-value, a bellman-like equation relating the optimal option-value $Q^*_{\mathcal{O}}(s, w)$ with the expected value of the option *upon arrival* at the next state s':

$$Q^*_{\mathcal{O}}(s, w) = \sum_{a \in \mathcal{A}_s} \pi(s, a) E\{r + \gamma U^*(s', w) | s, a\}, \tag{3.26}$$

where the value upon arrival is defined as

$$U^*_{\mathcal{O}}(s', w) = (1 - \beta_w(s')) Q^*_{\mathcal{O}}(s', o) + \beta_w(s') \max_{o' \in \mathcal{O}} Q^*_{\mathcal{O}}(s', o'), \tag{3.27}$$

There is a slight difference between the option-value $Q_{\mathcal{O}}(s, w)$ and the value upon arrival $U(s, w)$ - the latter depends explicitly on the termination probabilities $\beta_w(s)$, where the value is a weighted sum of the option-value for w and the value of the best option *if* the option terminates.

The resulting update rule is called *one-step intra-option Q-learning*:

$$Q(s_t, o) \leftarrow Q(s_t, o) + \alpha[r_{t+1} + \gamma U(s', o) - Q(s_t, o)]. \tag{3.28}$$

HRL methods have shown great improvements for planning and more efficient exploration for multiple complex environments [33, 34]. the option framework does not say *how to discover good options* and *how to determine the initiation set and termination condition*, which naturally has to be learned unless using hand-crafted deterministic policies, or policies specified in advance [3,33]. There has been great development in designing algorithms addressing these issues. Many of the current state-of-the-art HRL methods have shown great results for planning and efficient exploration for multiple complex environments [27,33,34].

Common for these methods is the underpinning framework based on options and the intra-option learning methodology.

3.10 Option-Critic

A limitation of the option-framework is that each option $\omega_{\mathcal{I},\pi,\beta} = \langle \mathcal{I}, \pi, \beta \rangle$ has to be specified by the engineer. For some problems such as the *4-room environment* which is a 4-room grid world with the goal of navigating to a position in another room [27,33,34] this might be natural, but many real-world problems are not as easily decomposed. If they were, other manually engineered methods might be more applicable anyway. In addition, as defined it does not extend to high-dimensional and continuous state space environments.

Option-critic (OC) is a method that expands the option-framework, alleviating these issues by learning and discovering options in an end-to-end fashion [27]. This is achieved by parameterizing the options $\omega_{\mathcal{I},\pi,\beta}$, $\forall \omega \in \Omega$ using neural networks and learning them by SGD and backpropagation through some loss function.

The main key is connecting the *intra-option learning* with policy gradient theorem, without the need to provide additional rewards or subgoals.

The execution model is as follows: At state s_t an agent picks an option ω according to the option-policy $\pi_\Omega(\cdot|s_t)$ and follows its intra-option policy $\pi_{w,\phi}(\cdot|s_t)$, selecting action a_t, continuing along a trajectory τ until termination of the option. At each step τ_t, the agent determines whether to continue or interrupt the current option, based on the termination probability at each state $\beta_\omega(s_t)$. When terminating the current option ω_t, a new one is selected based on the option-policy $\pi_\Omega(\cdot|s_t)$. This cycle is repeated indefinitely and is defined as the *call-and-return* execution model [27]. This encapsulates the main components of an option $\omega_{\mathcal{I},\pi,\beta} = \langle \mathcal{I}, \pi, \beta \rangle$, next we will describe the steps for learning and discovering these components in an end-to-end fashion.

There are two main steps, similarly to actor-critic methods:

- Updating the critic, which consists of the value functions Q_Ω, Q_U, V_Ω et cetera.
- Improving the actor, consisting of the policies π_Ω, π_ω and termination probabilities β_ω - guided by the critic.

Since this method is an end-to-end framework, the components in these two steps are parameterized using neural networks. Specifically θ, ϕ, ϑ denotes the parametrization for option-value Q_ω, intra-option policies π_ϕ and option termination probabilities β_ω respectively from now on.

Critic optimization

The critics role is to guide the actor, specifically to guide the gradient of the actor components during SGD. To do this efficiently we need a good approximation of the value of following the policy. This is done by taking advantage of the *intra-option Q-learning* for option improvement.

First we define the equations based on the options-framework [3] necessary for *intra-option* learning. The value of selecting option ω in state s and then following the policy is defined as:

The definition of the option-value as defined in OC is [27]:

$$\begin{aligned} Q_\Omega(s,\omega) &= E_{a\sim\pi_w}\{Q_U(s,\omega,a)\}, \\ &= \begin{cases} \sum_a \pi_{\omega,\phi}(a|s)Q_U(s,\omega,a) & \text{for discrete action-spaces,} \\ \int_a \pi_{\omega,\phi}(a|s)Q_U(s,\omega,a) & \text{for continuous action-spaces,} \end{cases} \end{aligned} \tag{3.29}$$

where $Q_U(s,\omega,a)$ is the *intra-option value*, denoting the value of taking action a in the augmented state-space (s,ω).

$$\begin{aligned} Q_U(s,\omega,a) &= r(s,a) + \gamma E_{s'\sim p}\{U(\omega,s')\} \\ Q_U(s,\omega,a) &= r(s,a) + \gamma \sum_{s'} P(s'|s,a)U(\omega,s'). \end{aligned} \tag{3.30}$$

$U(\omega,s')$ is the value of executing an option ω upon *upon arrival* at a state s', defined as: [3, 27]

$$U(\omega,s') = (1-\beta_{\omega,\vartheta}(s))Q_\Omega(s',\omega) + \beta_{\omega,\vartheta}(s')V_\Omega(s'). \tag{3.31}$$

U is subtly different from option-value Q_Ω. Value upon arrival is weighted on the probability of terminating the option that was followed before arrival to s'. In essence, U is defined as the value of continuing with the option *or* terminating and selecting a new one - weighted by the respective probabilities. (The value of an option ω differs whether it is the current option or not)

With these definitions we are able to write the bellman-like equations used in *one-step intra-option Q-learning*:

$$\begin{aligned} Q_U(s,\omega,a) &= r(s,a) + \gamma E_{s'\sim p}\{U(\omega,s')\} \\ Q_U(s,\omega,a) &= r(s,a) + \gamma \sum_{s'} P(s'|s,a)U(\omega,s'), \end{aligned} \tag{3.32a}$$

$$\begin{aligned} Q_\Omega(s,\omega) &= E_{a\sim\pi_w}\{Q_U(s,\omega,a)\} \\ Q_\Omega(s,\omega) &= E_{a\sim\pi_w}\{r(s,a)+\gamma E_{s'\sim p}\{U(\omega,s')\}\}. \end{aligned} \tag{3.32b}$$

If we optimize with respect to the greedy policy π_Ω we get the one-step off-policy target [27]

$$g_t^{(1)} = r_{t+1} + (1-\beta_{\omega,\vartheta}(s'))Q_\Omega(s',\omega) + \beta_{\omega,\vartheta}(s')V_\Omega^*(s'), \tag{3.33}$$

where $V_\Omega^*(s') = \max_{\omega'\in\mathcal{O}} Q_\Omega(s',\omega')$.

It still remains *how* to obtain the actor components. This is achieved by directly optimizing the return with respect to the parameters encapsulating the actor. This leads into to the main contribution of *OC*: *Intra-Option Policy Gradient Theorem* and *Termination Gradient Theorem.*

Actor optimization

For simplicity the theorems are developed in the case where the action-space is discrete, i.e $Q_\omega(s,\omega) = \sum_{a\in\mathcal{A}_s}\pi_{\omega,\phi}(a|s)Q_U(s,w,a)$. This is readily extended to continuous action-spaces as well.

The policy gradient is found by maximizing

$$\begin{aligned} \frac{\partial}{\partial\phi}Q_\Omega(s,\omega) &= \frac{\partial}{\partial\phi}E_{a\sim\pi_w}\{Q_U(s,\omega,a)\} \\ &= \frac{\partial}{\partial\phi}\left(\sum_a \pi_{\omega,\phi}(a|s)Q_U(s,\omega,a)\right) \end{aligned} \tag{3.34}$$

Expanding the intra-option value based on next state s' and applying the chain-rule:

$$\begin{aligned} \frac{\partial}{\partial\phi}Q_\Omega(s,\omega) &= \frac{\partial}{\partial\phi}\sum_a \pi_{\omega,\phi}(a|s)Q_U(s,\omega,a) + \sum_a \pi_{\omega,\phi}(a|s)\frac{\partial}{\partial\phi}Q_U(s,\omega,a) \\ &= \frac{\partial}{\partial\phi}\sum_a \pi_{\omega,\phi}(a|s)Q_U(s,\omega,a) \\ &+ \sum_a \pi_{\omega,\phi}(a|s)\frac{\partial}{\partial\phi}\left(r(s,a)+\gamma\sum_{s'}P(s'|s,a)U(\omega,s')\right) \\ &= \frac{\partial}{\partial\phi}\sum_a \pi_{\omega,\phi}(a|s)Q_U(s,\omega,a) \\ &+ \sum_a \pi_{\omega,\phi}(a|s)\left(\gamma\sum_{s'}P(s'|s,a)\frac{\partial}{\partial\phi}U(\omega,s')\right). \end{aligned} \tag{3.35}$$

Expanding the value upon arrival $U(w, s')$ we get

$$\frac{\partial}{\partial \phi} U(\omega, s') = (1 - \beta_{\omega,\vartheta}(s)) \frac{\partial}{\partial \phi} Q_\Omega(s', \omega) + \beta_{\omega,\vartheta}(s') \frac{\partial}{\partial \phi} V_\Omega(s') \tag{3.36}$$

Since $V_\Omega(s') = \sum_{\omega'} \pi_\Omega(\omega', s') Q_\Omega(s', \omega')$ eq. (3.36) may be rewritten as

$$\begin{aligned} \frac{\partial}{\partial \phi} U(\omega, s') &= (1 - \beta_{\omega,\vartheta}(s)) \frac{\partial}{\partial \phi} Q_\Omega(s', \omega) \\ &\quad + \beta_{\omega,\vartheta}(s') \sum_{\omega'} \pi_\Omega(\omega', s') \frac{\partial}{\partial \phi} Q_\Omega(s', \omega') \\ &= \sum_{\omega'} \left((1 - \beta_{\omega,\vartheta}(s)) \mathbf{1}_{\omega'=\omega} + \beta_{\omega,\vartheta}(s') \pi_\Omega(\omega', s') \right) \frac{\partial}{\partial \phi} Q_\Omega(s', \omega') \end{aligned} . \tag{3.37}$$

This yields a recursion, and *Bacon et al.* [27] proves the *option-policy gradient theorem*:

$$\sum_{s,\omega} \mu_\Omega(s, \omega | s_0, \omega_0) \sum_a d \frac{\pi_{\omega,\phi}(a|s)}{d\theta} \cdot Q_U(s, \omega, a)$$

3.11 Soft Actor-Critic (SAC)

SAC is an off-policy actor critic algorithm which aims to maximize entropy while solving the task. In essence, it tries to maximize return while acting as randomly as possible [35]. In comparison to other methods based on the maximum entropy RL framework [36], SAC combines off-policy updates with a stochastic policy using a formulation based on actor-critic methods.

The basis for this method is *soft policy iteration*, which alternates between policy evaluation- and improvement [2].

In the *policy evaluation step*, a modified Bellman equation is used for iteratively finding a better estimate of the soft Q-values

$$\mathcal{T}^\pi Q(s_t, a_t) \doteq r(s_t, a_t) + \gamma E_{s_{t+1} \sim p_\pi}[V(s_{t+1})] \tag{3.38a}$$

$$Q(s_t, a_t) = r(s_t, a_t) + \gamma E_{s \sim p}[V_{\bar{\psi}}(s_{t+1})] \tag{3.38b}$$

$$V(s_t) = E_{a_t \sim \pi}[Q(s_t, a_t) - \log \pi(a_t | s_t)] \tag{3.38c}$$

where $V(s_t) = E_{a_t \sim \pi}[Q(s_t, a_t) - \log \pi(a_t | s_t)]$ and p denotes the state transition distribution of a trajectory.

When working in continuous state-space environments we need an approximate the policy iteration. Similarly to other DRL methods, this is done using neural networks. For this setting, the policy iteration step is equivalent to first back-propagating the critic networks (policy evaluation) then back-propagate the actor networks (policy improvement).

With the *policy improvement step*, we want the policy to be distributed similarly as the exponential of the Q-value. This is done by minimizing the Kullback-Leibler divergence between the two. Specifically, the policy parameters are learned by minimizing the expected KL-divergence between the policy and normalized exponential action-value distribution:

$$J_\pi(\phi) = E_{s_t \sim \mathcal{D}} \left[D_{KL} \left(\pi_\phi(\cdot|s_t) || \frac{\exp(Q_\theta(s_t, \cdot))}{Z_\theta(s_t)} \right) \right]. \tag{3.39}$$

The main idea is realizing that action-value Q_θ is parameterized by a neural network, thus allowing us to take advantage of that when optimizing the policy. The key is to reparameterize the policy using the *reparameterization trick* [17]:

$$a_t = f_\phi(\epsilon_t; s_t), \tag{3.40}$$

where ϵ_t is an input noise vector sampled from a fixed distribution - usually the standard Normal distribution. Critically, this detail alleviates the troubling expectation over actions found in policy gradient methods, instead reducing it to an expectation over a fixed noise distribution, $\epsilon_t \sim N$.

Now, we are able rewrite the objective (3.39) as

$$J_\pi(\phi) = E_{s_t \sim \mathcal{D}, \epsilon_t \sim \mathcal{N}}[\alpha \log \pi_\phi(f_\phi(\epsilon_t; s_t)|s_t) - Q(s_t, f_\phi(\epsilon_t; s_t))] \tag{3.41}$$

Where the π_ϕ is implicitly defined from $f_\phi(\epsilon_t; s_t)$ [35]. Additionally, $Z_\theta(s_t)$ was omitted since it only acts as a normalization factor and does not depend on ϕ. The gradient of the objective with respect to ϕ can be approximated (using the chain rule):

$$\begin{aligned} \hat{\nabla}_\phi J_\pi(\phi) &= \frac{\partial J_\pi(\phi)}{\partial \log \pi_\phi} \frac{\partial \log \pi_\phi}{\partial \phi} + \frac{\partial J_\pi(\phi)}{\partial \log \pi_\phi} \frac{\partial \log \pi_\phi}{\partial f_\phi} \frac{\partial f_\phi}{\partial \phi} \\ &+ \frac{\partial J_\pi(\phi)}{\partial Q(s_t, f_\phi)} \frac{\partial Q(s_t, f_\phi)}{\partial f_\phi} \frac{\partial f_\phi}{\partial \phi} \\ \hat{\nabla}_\phi J_\pi(\phi) &= \nabla_\phi \log \pi_\phi(a_t|s_t) \\ &+ \nabla_{a_t}(\log \pi_\phi(a_t|s_t) - Q(s_t, a_t)) \nabla_\phi f_\phi(\epsilon_t; s_t) \end{aligned} \tag{3.42}$$

We propose a new method based on combining a soft actor-critic formulation with the option-critic framework which we call **Soft Option-Critic**, which is presented in the next chapter.

Chapter 4

Soft Option Critic (SOC)

The main motivation is to develop an OC-method with the potential to exploit off-policy data, improving sample-efficiency, while being robust to ever-changing- and non-stationary environments such as type 1D diabetes where safety is key. By using the off-policy intra-option Q-learning method as in OC, but combining it with the idea from SAC [35] where the policy is optimized to be similarly distributed as the value function, instead of directly optimizing the return with respect to the policy, we manage to create an off-policy formulation [1].

The idea of SOC is to combine the robustness of SAC which uses the maximum entropy objective, with the framework from OC which is suitable for improving the planning across temporal abstractions and for non-stationary environments.

A key detail is to modify the option-value functions such that they take into account the entropy of the policies. In essence there are two main concepts: *Option estimation* and *policy improvement*, in similar vein to *policy evaluation- and improvement* in SAC. We begin by introducing some the key equations of SOC.

[1]While developing the idea and algorithm I discovered that the idea of combining OC and SAC has been proposed before [37]. Though it is not available now (April 2020), and was retracted because it was shown that the preprint was lacking both in the description and results [37].

Option estimation

The definition of the option-value as defined in OC is [27]:

$$\begin{aligned} Q_\Omega(s,\omega) &= E_{a\sim\pi_w}\{Q_U(s,\omega,a)\} \\ &= \sum_a \pi_{\omega,\phi}(a|s)Q_U(s,\omega,a), \quad \text{for discrete action-spaces} \\ &= \int_a \pi_{\omega,\phi}(a|s)Q_U(s,\omega,a), \quad \text{for continuous action-spaces} \end{aligned} \tag{4.1}$$

Where $Q_U(s,\omega,a)$ is the value of taking action a in the augmented state-space (s,ω). We denote this as the *intra-option value*, which in essence is the value of following option ω and taking action a in state s. It is defined as the reward plus discounted future reward U

$$\begin{aligned} Q_U(s,\omega,a) &= r(s,a) + \gamma E_{s'\sim p}\{U(\omega,s')\} \\ Q_U(s,\omega,a) &= r(s,a) + \gamma \sum_{s'} P(s'|s,a)U(\omega,s') \end{aligned} \tag{4.2}$$

We introduce the value of executing an option ω *upon arrival* at a state s' as: [3, 27]

$$U(\omega,s') = (1-\beta_{\omega,\vartheta}(s))Q_\Omega(s',\omega) + \beta_{\omega,\vartheta}(s')V_\Omega(s') \tag{4.3}$$

U is subtly different from the option-value Q_Ω. Value upon arrival is weighted on the probability of terminating the option that was followed before arrival to s'. In essence, U is defined as the value of continuing with the option *or* terminating and selecting a new one - weighted by the respective probabilities.

The value at a state s is given as the expectation over all the option-values:

$$V_\Omega(s) = E_{\omega\sim\pi_\Omega}[Q_\Omega(s,\omega)] = \sum_w \pi_\Omega(\omega|s)Q_\Omega(s,\omega) \tag{4.4}$$

Finally, we introduce the key equations based on the modified objective by re-defining eq. (4.1), were the value is in addition based on the entropy of the respective intra-policy $\pi_{\omega,\phi}$. This is defined as the *soft option-value*:

$$\begin{aligned} \tilde{Q}_\Omega(s,\omega) &= E_{a\sim\pi_w}\{\tilde{Q}_U(s,\omega,a) + \alpha\mathcal{H}(\pi_{\omega,\phi}(\cdot|s))\} \\ &= E_{a\sim\pi_w}\{\tilde{Q}_U(s,\omega,a) - \alpha\log\pi_{\omega,\phi}(a|s)\}. \end{aligned} \tag{4.5}$$

This leads into the modified value-function

$$\begin{aligned}\tilde{V}_\Omega(s) &= E_{\omega\sim\pi_\Omega}[\tilde{Q}_\Omega(s,\omega)] \\ &= E_{\omega\sim\pi_\Omega,\, a\sim\pi_w}\{\tilde{Q}_U(s,\omega,a) - \alpha\log\pi_{\omega,\,\phi}(a|s)\}\end{aligned} \tag{4.6}$$

I now develop bellman-like equations for the intra-option value $\tilde{Q}_U(s,\omega,a)$ and option-value $\tilde{Q}_\Omega(s,\omega)$:

$$\tilde{Q}_U(s,\omega,a) = r(s,a) + E_{s'\sim p}\{\gamma\tilde{U}(\omega,s')\} \tag{4.7a}$$

$$\tilde{Q}_\Omega(s,\omega) = E_{a\sim\pi_w}\{\tilde{Q}_U(s,\omega,a) - \alpha\log\pi_{\omega,\phi}(a|s)\} \tag{4.7b}$$

In (4.7b) we rewrite $\tilde{Q}_U(s,\omega,a)$ in terms of the bootstrapped intra-value in (4.7a):

$$\tilde{Q}_\Omega(s,\omega) = E_{a\sim\pi_w}\{r(s,a) + \gamma E_{s'\sim p}\{\tilde{U}(\omega,s')\} - \alpha\log\pi_{\omega,\phi}(a|s)\} \tag{4.8}$$

where

$$\tilde{U}(\omega,s') = (1-\beta_{\omega,\vartheta}(s'))\tilde{Q}_\Omega(s',\omega) + \beta_{\omega,\vartheta}(s')\tilde{V}_\Omega(s') \tag{4.9}$$

With these definitions we are now able to construct the *intra-option Q-learning* method, for learning the modified option-values $\tilde{Q}_\Omega$ and $\tilde{Q}_U$, using the bellman-like equations from [3]. As with Q-learning, the training for $\tilde{Q}_w$ and $\tilde{Q}_w$ is based on the option-value for the *optimal policy (over options)* π^*_Ω. In practice this is the *greedy option-policy*, thus we improve the value estimates for π^*_Ω, while following another policy π_Ω. this is the key that allows *option improvements* in an off-policy way - iteratively improve estimates of the value of following the optimal policy by bootstrapping (through experiences).

This leads to the update rules:

$$\begin{aligned}&\tilde{Q}_\Omega(s_t,\omega_t) \leftarrow \tilde{Q}_\Omega(s_t,\omega_t) + \\ &\qquad\alpha[r_{t+1} + \gamma\tilde{U}^*(\omega_t,s_{t+1}) - \alpha\log\pi_{\omega,\phi}(\tilde{a}_\omega|s_t) - \tilde{Q}_\Omega(s_t,\omega_t)]\end{aligned} \tag{4.10a}$$

$$\tilde{Q}_U(s_t,\omega_t,a) \leftarrow \tilde{Q}_U(s_t,\omega_t) + \alpha[r_{t+1} + \gamma\tilde{U}^*(\omega_t,s_{t+1}) - \tilde{Q}_U(s_t,\omega_t,a)] \tag{4.10b}$$

where $\tilde{a}_\omega$ is sampled from the policy $\pi_{\omega,\phi}(\cdot|s)$. With this scheme we are now able to improve the option-value estimates - the "critic" is defined. We now turn our focus on *how* to improve the actor, denoting the intra-policies $\pi_{\omega\in\Omega}$ and their respective termination probabilities $\beta_{\omega\in\Omega}$

Policy improvement

The intra-option policies π_w are updated in a similar fashion as in SAC - we want π_w similarly distributed as its corresponding intra-option value function Q_U. This is achieved by minimizing the KL-divergence between the two:

$$J_{\pi_\omega}(\phi) = E_{s_t \sim \mathcal{D}} \left[D_{KL} \left(\pi_{\omega,\phi}(\cdot|s_t) || \frac{\exp(Q_{U,\theta}(s_t, \omega, \cdot))}{Z_{\omega,\theta}(s_t)} \right) \right] \tag{4.11}$$

where $Z_{\omega,\theta}(s_t)$ is the normalization factor. It can be shown* (as in SAC) that this is equivalent to minimizing the objective:

$$J_{\pi_w}(\phi) = E_{(s_t, w_t) \sim \mathcal{D},\, a_t \sim \pi_{\omega_t,\phi}} [\alpha \log \pi_{\omega,\phi}(a_t|s_t) - Q_U(s_t, w_t, a_t)] \tag{4.12}$$

Crucially there is a pain point for optimization of $J_{\pi_w}(\phi)$: The expectation is over the distribution which parameters we want to optimize. the key trick is the *reparameterization trick* introduced in SAC - reparameterize the policy through the action selection using the neural network transform:

$$a_t = f_\phi(\epsilon_t; s_t)$$

where ϵ_t is a fixed noise distribution.

$$J_{\pi_w}(\phi) = E_{(s_t, w_t) \sim \mathcal{D},\, \epsilon_t \sim \mathcal{N}} [\alpha \log \pi_{\omega,\phi}(f_\phi(\epsilon_t; s_t)|s_t) - Q_U(s_t, w_t, f_\phi(\epsilon_t; s_t))]$$

Approximating the expectation using samples results in the unbiased approximate gradient of the objective $J_{\pi_w}(\phi)$:

$$\begin{aligned} \hat{\nabla}_\phi J_{\pi_\omega}(\phi) &= \frac{\partial J_{\pi_\omega}(\phi)}{\partial \log \pi_{\pi_\omega}} \frac{\partial \log \pi_\omega}{\partial \phi} + \frac{\partial J_{\pi_\omega}(\phi)}{\partial \log \pi_\omega} \frac{\partial \log \pi_\omega}{\partial f_\phi} \frac{\partial f_\phi}{\partial \phi} \\ &+ \frac{\partial J_{\pi_\omega}(\phi)}{\partial Q_U(s_t, \omega, f_\phi)} \frac{\partial Q_U(s_t, \omega, f_\phi)}{\partial f_\phi} \frac{\partial f_\phi}{\partial \phi} \\ \hat{\nabla}_\phi J_{\pi_\omega}(\phi) &= \nabla_\phi \log \pi_{\omega,\phi}(a_t|s_t) \\ &+ \nabla_{a_t} (\log \pi_{\omega,\phi}(a_t|s_t) - Q_U(s_t, \omega_t, a_t)) \nabla_\phi f_\phi(\epsilon_t; s_t) \end{aligned} \tag{4.13}$$

The *termination gradient* $\nabla_\vartheta \beta(s')$ is found in a similar fashion as in Option-Critic, by taking the gradient of the option-value w.r.t. ϑ:

$$\begin{aligned} \frac{\partial Q_\Omega(s, \omega)}{\partial \vartheta} &= \frac{\partial}{\partial \vartheta} E_{a \sim \pi_\omega} \{Q_U(s, \omega, a) - \alpha \log \pi_{\omega,\phi}\} \\ &= \frac{\partial}{\partial \vartheta} E_{a \sim \pi_\omega} \{Q_U(s, \omega, a)\} \end{aligned} \tag{4.14}$$

Expanding the expectation over intra-value with the bootstrapped value (/using the bellman operator) as in (4.7a):

$$\begin{aligned}
\frac{\partial Q_\Omega(s,\omega)}{\partial \vartheta} &= \frac{\partial}{\partial \vartheta}\{r(s,a) + E_{s'\sim p}\{\gamma U(\omega,s')\}\} \\
&= E_{s'\sim p}\{\gamma \frac{\partial}{\partial \vartheta} U(\omega,s')\} \\
\frac{\partial Q_\Omega(s,\omega)}{\partial \vartheta} &= -\frac{\partial \beta_w(s)}{\partial \vartheta}\tilde{A}_\Omega(s,\omega')
\end{aligned} \tag{4.15}$$

where $\tilde{A}_\Omega(s,\omega') = \tilde{Q}_\Omega(s,\omega') - \tilde{V}_\Omega(s)$ is the *soft advantage-value.* Similarly to the target for option-values (4.10), we update $\nabla_\vartheta \beta(s')$ based on the estimated advantage of the *optimal policy* π^*_Ω

$$\tilde{A}^*_\Omega(s,\omega') = \tilde{Q}_\Omega(s,\omega') - \max_{\omega'\in\Omega} \tilde{Q}_\Omega(s',\omega') \tag{4.16}$$

In principle this should alleviate the "on-policyness" of the termination update. Additionally, since we use experience replay we effectively take the mean of all contributions which should approximate a good gradient.

In summary, we bootstrap the option-values of π^*_Ω using *intra-option Q-learning* while keeping the intra-policies similarly distributed as their respective value Q_U. This stands in contrast to OC where the objective is maximized directly with respect to the intra-policy parameters ϕ.

Chapter 5

Experiments

The goal of the following experiments is twofold. The main goal is to evaluate the potential for state-of-the-art RL-methods to improve insulin-control for patients with type 1D diabetes. SOC was developed specifically with this in mind, combining the robustness of SAC with options; An abstraction of actions with the potential for improved sample-efficiency, by exploiting specialization of options. Thus, the second goal is to evaluate SOC against state-of-the-art RL-methods on the *known* environment, lunar lander, to test whether it improves sample efficiency, especially compared against SAC.

This chapter describes the experiments that were performed to asses the performance of multiple RL-algorithms into experiments testing on the diabetes simulator, comparing to the "optimal" standard method. In addition, since the new algorithm, SOC, was developed with insulin control in mind, we test the merits of it on an already "solved" environment, Lunar lander.

This chapter consists of two parts:

1. In the first part, SOC is evaluated in comparison to other state-of-the-art methods such as SAC and PPO on *lunar lander*, to see whether there are benefits for using the option framework, especially testing if there is a performance improvement when using SOC.
2. In the first part, the performance of selected algorithms on the *diabetes simulator* is evaluated and compared against a standard method optimal for the simulated patient. Different scenarios emulating real-life situations will be tested, such as when a patient drops meals and bolus with a certain probability.

The next section describes the general setup that are mutual between the two parts. This includes the performance metric used during training and general procedure for the experiments. Following this, the implementations and parameters of the algorithms are defined. The final sections describes the details specific to each environment and the corresponding experimental results.

The last section provides the discussion and conclusion of the experiments, including some thoughts on future work.

As such, the experiments are divided into two main parts: **Lunar lander** and **diabetes**. First the general setup shared between both parts are defined. This includes defining the mutual performance metrics and the experimental procedure, in addition to how evaluations and comparisons are made between the algorithms.

All algorithms are implemented in Python 3.6 using the deep learning library `PyTorch` [38]. Specifically they are implemented within the *spinningup* framework [39]. Spinningup is a module containing useful tools for the development of DRL-methods. this includes functionality for running experiments, plotting and a code base with implementations of state-of-the-art DRL-methods. SOC and its extensions are developed and implemented to be compatible within this framework.

Mostly, the default settings from spinningup were used since they have been found to be good across multiple environments.

5.1 Experimental setup

This section presents general setup and the implementation details for the experiments that are mutual for both the lunar-lander- and diabetes environment.

The default parameters are set for both environments and the network architecture is specified. This leads to the subsections specific to lunar-lander and diabetes setup.

For lunar lander, REINFORCE is used as the baseline algorithm all the others are compared against. To justify the added complexity when extending algorithms, they should at least improve on the performance on the baseline. For diabetes, a heuristic method is used as the baseline algorithm, which is defined as the *optimal baseline* (OB). It is described in 5.5.

5.1.1 Performance metric and notation

The performance of each method is evaluated during training at each epoch. The definition of performance is similarly defined as in *Spinningup* [39]:

1. The performance metric P_e is the average episodic return from the batch of experience. For on-policy methods this would be the average episodic return across the batch collected during the epoch, while P_e for off-policy methods was calculated from $N_r = 10$ test episodes with the respective deterministic policy.

2. An epoch denotes a fixed number of time-steps, or environment interactions. The default value is $t_e = 4000$ time-steps.

5.1.2 Procedure

The algorithms are trained for $N_e = 50$ epochs, evaluating performance P_e for multiple seeds. With the trained models, run $N_{\text{test}} = 100$ test episodes, calculating average return $\bar{R}$ and other environment-specific performance metrics. Next, the simulator is run for an episode with the intent of analyzing options for SOC.

This illustrates the sample efficiency, allowing comparisons between algorithms to be made. Specifically, the performance metric P_e is used. For the diabetes experiments the concept of *time-in-range* (TIR) is used as a metric for how good the algorithms perform for the patient [40, 41].

5.1.3 Analysis of options

Analysis of the options will be done to see whether they specialize to some differing abstract actions. Specifically, a test episode will be used, illustrating the trajectory of the state-space and what the options are at each step will be analyzed, including:

- If the options are compact - consistently lasting over multiple time-steps.

- Which parts of the state-space they focus on, discussing whether they are abstract actions or not, following the intuition we have about how the options should specialize.

5.2 Benchmark Models

The benchmark models was based off of the default values in *spinningup* [39], as they have been tested and found to be good baseline values. To keep the comparisons as fair as possible between algorithms some implementation details were fixed. The neural network architectures are set as 3-layers network with nodes $[128, 256, 128]$.

The subsequent sections describes the design decisions and experiments details specific to each environment, including state- and action-space and reward function.

5.3 Experiment I: Lunar lander

The merits of the newly developed and implemented algorithm SOC is analyzed, comparing its performance against state-of-the art methods such as PPO, SAC, and TD3, while using the basic policy-based method REINFORCE as a baseline. This is not necessarily a difficult problem to solve given enough training samples. Though in real-life problems such as T1D management, RL-engineers do not have the luxury of learning with trial-and-error over many samples before achieving a good policy, where the risk of fatal error is not tolerated. The key With the lunar lander experiments is to elucidate the sample-efficiency of the methods.

Usually for methods based on the option framework, the environments are hand-picked where it seems natural for options to be beneficial. For environments such as `4-rooms` [27, 33, 34], specialization is easy to observe and the options are interpretative and intuitive, each option usually corresponds to going to a specific door or room. The common theme between these environments, is that they are made for using a discrete action-space. The main goal of this thesis is to improve upon insulin control for diabetes, where it is not as natural to divide the action-space into discrete number of actions. Some environments using a continuous action-space have shown benefits based on using options [33, 42, 43], though these are all from *Mujoco* [44] - a proprietary physics engine, which requires good hardware and much cpu-time to solve.

Thus the choice fell on lunar lander - an environment solvable within a reasonable amount of time. From intuition it does not necessarily contain a clear- and "best" way for abstractions of actions such as in `4-rooms`, but the mentioned benefit allows for testing multiple modifications and parameters of SOC and comparison between other methods.

In essence, lunar lander is a known and "solved" environment, which makes it easier to compare the performance of algorithms. Therefore it makes for a useful test bed for the extensions of SOC, using tricks based on deep Q-learning [24] and its extensions DDQN [25] and the dueling architecture [45].

5.3.1 Lunar lander setup

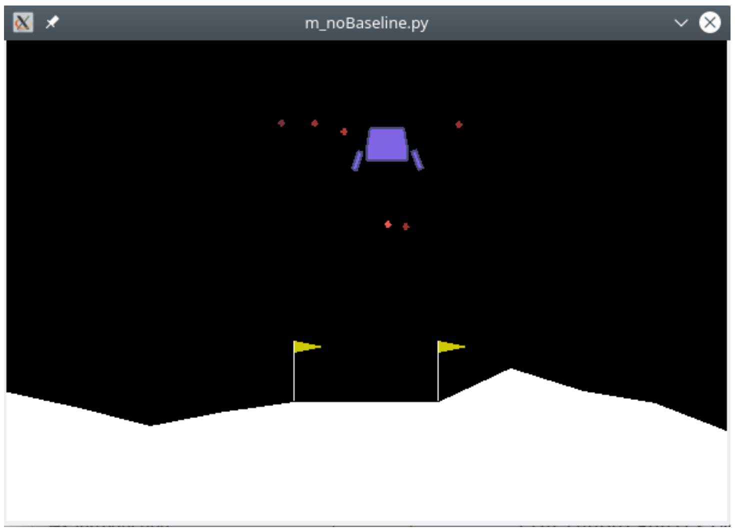


Figure 5.1: A screenshot illustrating the lunar lander environment.

The goal is to land a lunar lander within the landing pad, while minimizing fuel consumption. Specifically, the environment used for the experiments was the '`LunarLanderContinuous-v2`' from `Open AI gym` [46]. There are no design or engineering decision needed for this environment since the central components such as the reward function, state- and action space already are predefined.

The reward function for this environment is defined as

- r^+ when moving from the top of the screen towards the landing pad, r^- when moving away
- $r = 10$ for each leg ground contact.
- $r = -0.3$ when firing main engine each frame.
- $r = 100$ when the lander comes to rest, $r = -100$ if it crashes or moves out of the field.

r^+ and r^- denotes positive- and negative reward respectively. The total reward R_τ over the trajectory is $R_\tau = \sum_{i=1}^{n_\tau} R_i$, where $R_i = \sum_{j=1}^{n_r} r_j$.

The episodes ends when the lander either crashes, moves out of the field or reaches number of time-steps $T = 2000$.

For this problem we have continuous action- and state space. The action space is 2-dimensional $[a_1,\, a_2]$ where a_1 denotes the main engine throttle and a_2 denotes left-right thrust, while the state space is 8-dimensional:

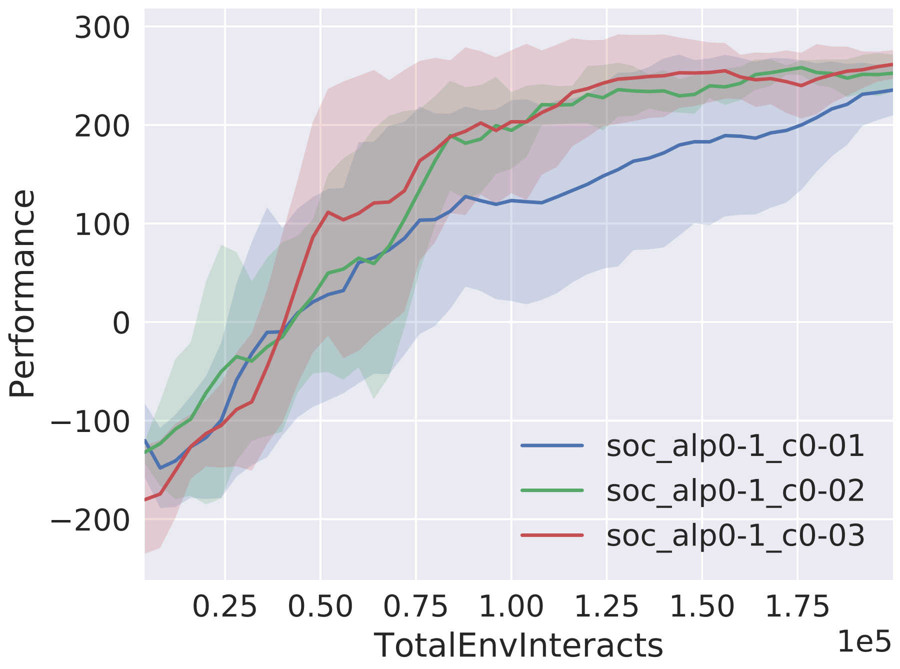
$$\begin{aligned}
\mathcal{A}_{\text{lunar}} &= \{a_1 = \text{main thrust} \in (-1,1],\ a_2 = \text{left-right thrust} \in [-1,1]\} \\
\mathcal{S}_{\text{lunar}} &= \{X_0,\, Y_0, x,\, y,\, v_x,\, v_y,\, \theta,\, v_\theta\} \\
X_0, Y_0 &\sim \text{coordinates of the landing pad} \\
x, y &\sim \text{position of lander} \\
v_x, v_y &\sim \text{velocity of lander} \\
\theta &\sim \text{lander angle} \\
v_\theta &\sim \text{angular velocity}
\end{aligned} \tag{5.1}$$

The following sections contains the results and analysis of the lunar lander experiments.

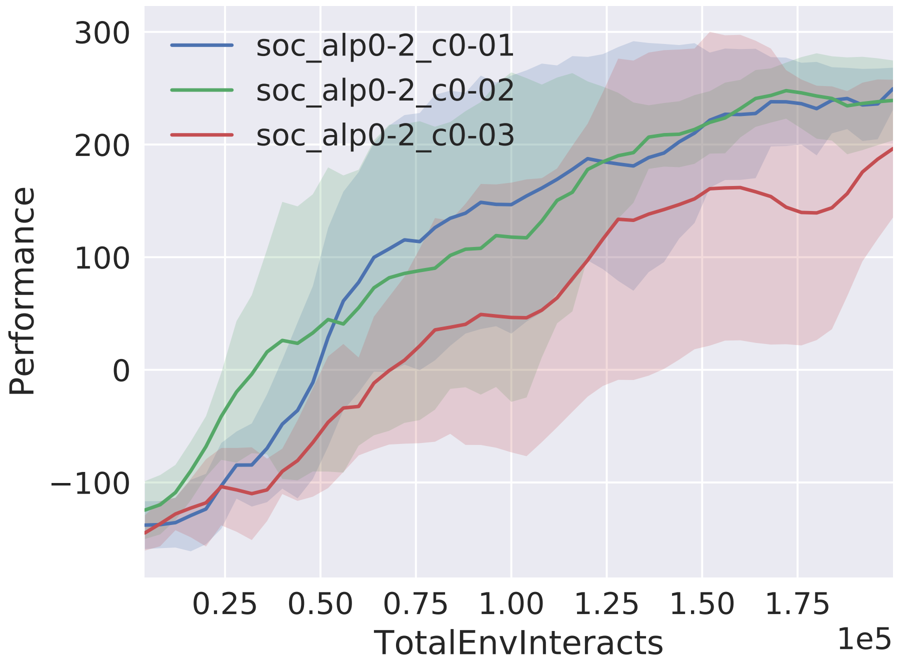
5.4 Results and analysis for lunar lander

The first part of this section is a comparison of different parameters for all the extensions of SOC.

5.4.1 Comparison of parameters for SOC and its extensions

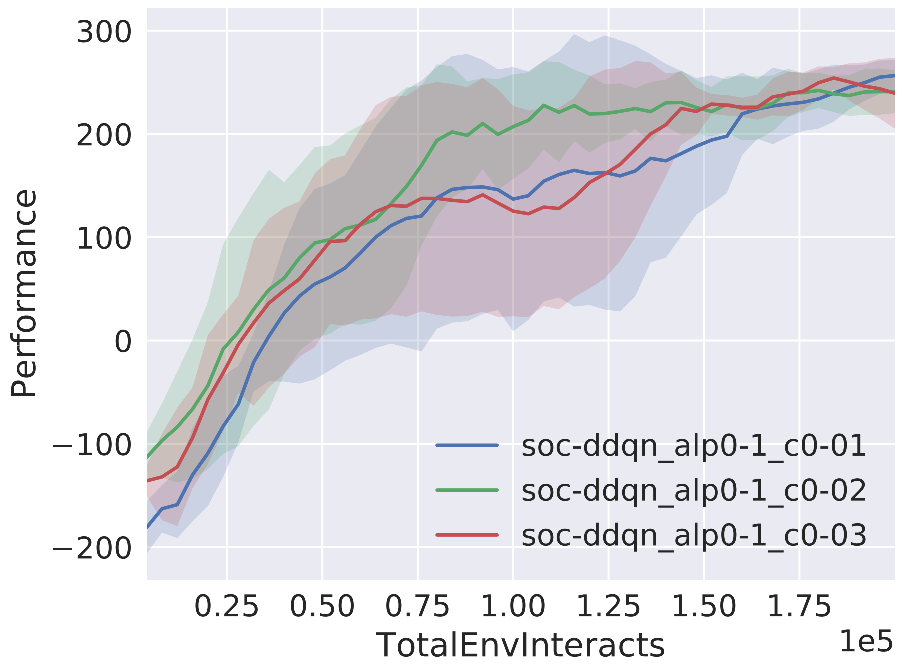


(a) Plot of the performance P_e during the training process for **SOC** on *lunar lander*, comparing different values for c when $\alpha = 0.1$. The best performance is achieved when $c = 0.02$ and $c = 0.03$, though the performance was relatively consistent across values of c.

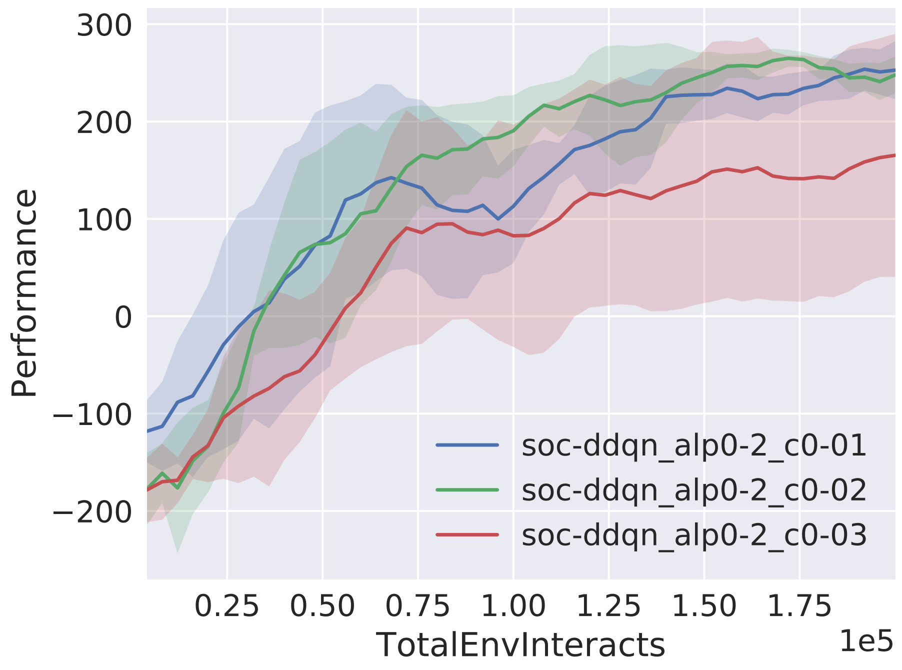


(b) Plot of the performance P_e during the training process for **SOC** on *lunar lander*, comparing different values for c when $\alpha = 0.2$. The best performance is achieved when $c = 0.01$ and $c = 0.02$, while $c = 0.03$ has notable variance in performance across seeds.

Figure 5.2: Figure showing two plots of the performance P_e during the training process for **SOC** on *lunar lander*, smoothed with a moving average of $S = 5$. Different values for α and c are compared. The shaded area denotes the standard deviation of P_e across seeds. The best performance is achieved with $\alpha = 0.1$, while $c = 0.02$ was the best value for all α values tested.



(a) Plot of the performance P_e during the training process for **SOC-DDQN** on *lunar lander*, comparing different values for c when $\alpha = 0.1$. The best performance is achieved when $c = 0.02$, though the performance was relatively consistent across values of c.



(b) Plot of the performance P_e during the training process for **SOC-DDQN** on *lunar lander*, comparing different values for c when $\alpha = 0.2$. The best performance is achieved when $c = 0.01$ and $c = 0.02$. Especially the run for $c = 0.03$ exhibits a large variance in performance across seeds.

Figure 5.3: Figure showing two plots of the performance P_e during the training process for **SOC-DDQN** on *lunar lander*, smoothed with a moving average of $S = 5$. Different values for α and c are compared. The shaded area denotes the standard deviation of P_e across seeds. The best performance is achieved with $\alpha = 0.1$, while $c = 0.02$ was the best value for all α values tested.

As seen from figure 5.2, SOC does learn while training, achieving $P_e \geq 200$ with most parameter combinations, solving the environment. Overall, it performed better when $\alpha = 0.1$ and $c = 0.02$, indicating that it is beneficial to use deliberation cost. The reasoning is to incentivize the agent to use extended options, allowing for improved specialization.

Figure 5.3 shows that it still is able to solve the environment for most parameters, except for $\alpha = 0.2$, $c = 0.03$. Similarly as SOC 5.2, $\alpha = 0.2$, $c = 0.03$ are the best combination. Although it exhibits greater variance in performance, especially around $t_l \approx 1$ where soc has solved the environment.

As seen in figure 5.4, SOC-duel achieves by far the most consistent performance, specifically when $\alpha = 0.1$. It performed consistently across different values for the deliberation cost c, but also in this case $c = 0.02$ was marginally better, having less variance compared to $c = 0.01$ and $c = 0.03$.

Soc with option-policy π_Ω using softmax 5.5 showed the same patterns as the other modifications. The best parameter for entropy coefficient was $\alpha = 0.1$. Similarly to dueling SOC, it performs consistently across values for c, having less variance than soc.

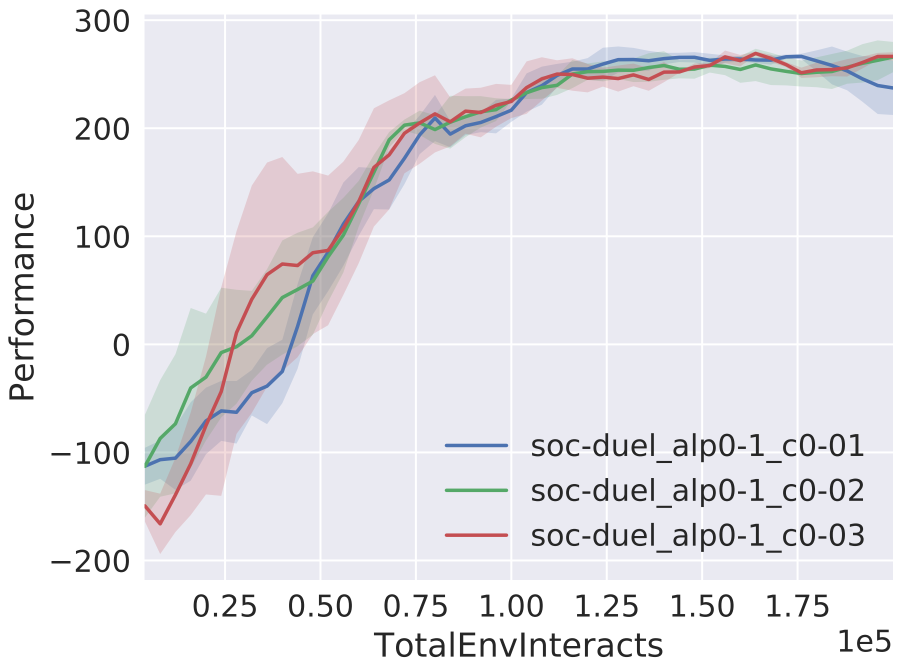
Since it was consistently shown that $\alpha = 0.1$ was the best value for all SOC-methods, testing with more than 2 options was done with a reduced set of parameters for computational reasons. Specifically, the method was tested with $\alpha = 0.1$ for $N_\omega = [3, 4]$ and $c = [0.02, 0.03]$.

Figure 5.6 achieves the best performance, solving the environment when $t_l \approx 0.60$ for parameters $\alpha = 0.1$ and $c = 0.02$, using 3 options. Although all 4 runs solved the environment before training ends, there was greater performance discrepancy compared to the earlier methods. $N_\omega = 3$ proved as the "sweet-spot" for the number of options in this setting, though the performance varied more between $c = 0.02$ and $c = 0.03$ not showing the same consistency across seeds as $c = 0.02$. Though empirically, we may conclude that it would be sufficient to test for $c = 0.02$ as it has been the common denominator between all the extensions.

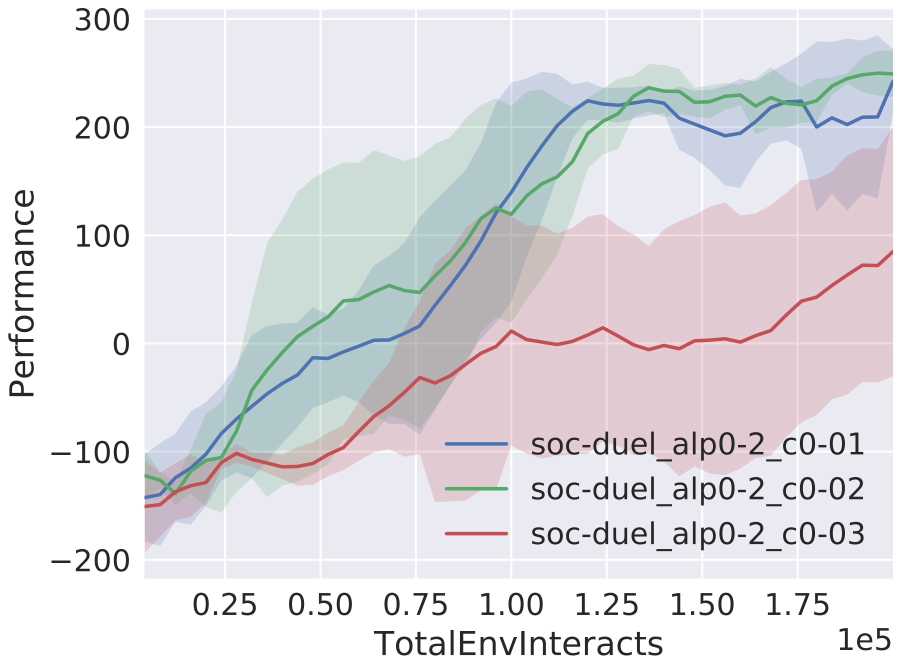
In the following subsection the performance

5.4.2 Comparison of the algorithms on lunar lander

As SOC and SAC are very similar on some implementation details and their respective parameter values. Interestingly, as seen in figure 5.7, $\alpha = 0.2$ was the best performing parameter-value as opposed to SOC and its modifications. The reason may be because SOC inherently has extra exploration since the option-policy π_Ω adds some stochasticity, were each intra-option π_ω in addition is different from each other, which is another source for stochasticity.

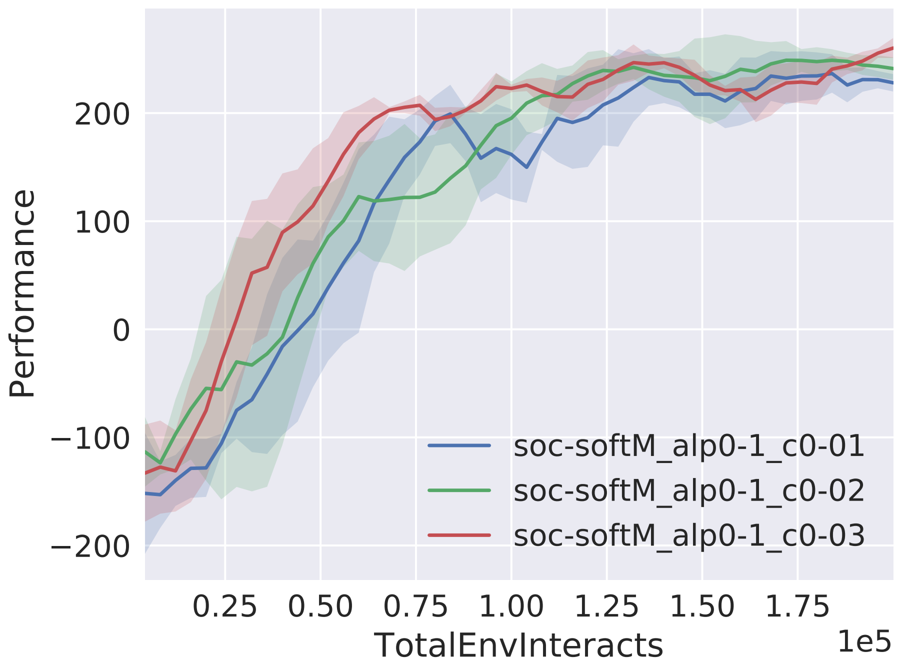


(a) Plot of the performance P_e during the training process for **SOC-duel** on *lunar lander*, comparing different values for c when $\alpha = 0.1$. The performance is very consistent across values of c.

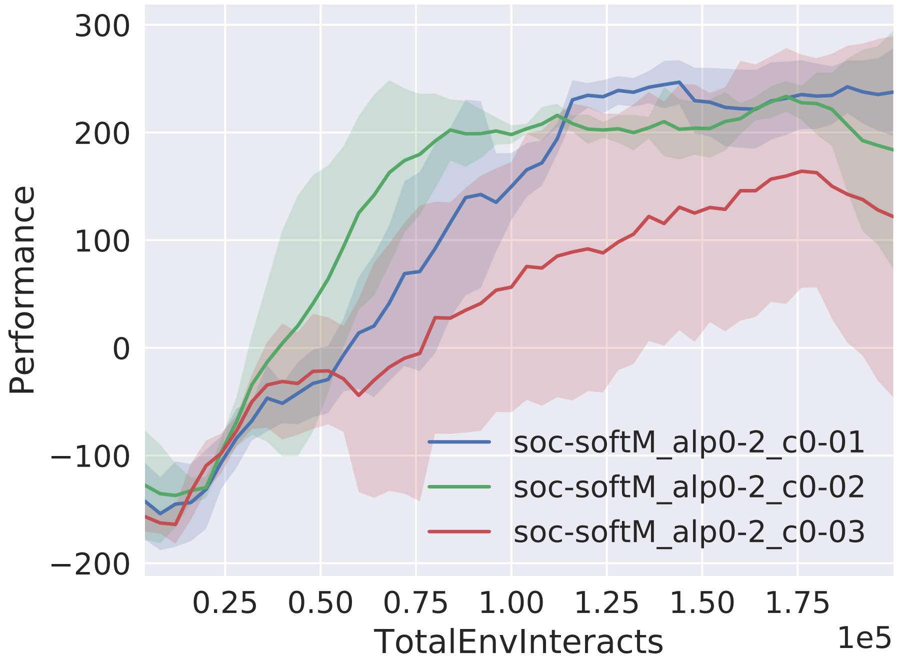


(b) Plot of the performance P_e during the training process for **SOC-duel** on *lunar lander*, comparing different values for c when $\alpha = 0.2$. The best performance is achieved when $c = 0.01$ and $c = 0.02$, while $c = 0.03$ fails to reach $P_e = 100$.

Figure 5.4: Figure showing two plots of the performance P_e during the training process for **SOC-duel** on *lunar lander*, smoothed with a moving average of $S = 5$. Different values for α and c are compared. The shaded area denotes the standard deviation of P_e across seeds. The best performance is achieved with $\alpha = 0.1$, while $c = 0.02$ was the best value for all α values tested. The runs when $\alpha = 0.2$ exhibit large variance across seeds, where $c = 0.03$ fails to reach $P_e = 100$.



(a) Plot of the performance P_e during the training process for **SOC-softmax** on *lunar lander*, comparing different values for c when $\alpha = 0.1$. The performance is very consistent across values of c, with $c = 0.3$ being slightly better.



(b) Plot of the performance P_e during the training process for **SOC-softmax** on *lunar lander*, comparing different values for c when $\alpha = 0.2$. The best performance is achieved when $c = 0.01$ and $c = 0.02$, while $c = 0.03$ fails to reach $P_e = 200$.

Figure 5.5: Figure showing two plots of the performance P_e during the training process for **SOC-softmax** on *lunar lander*, smoothed with a moving average of $S = 5$. Different values for α and c are compared. The shaded area denotes the standard deviation of P_e across seeds. The best performance is achieved with $\alpha = 0.1$ and $c = 0.03$. The runs when $\alpha = 0.2$ exhibit large variance across seeds, where $c = 0.02$ and $c = 0.03$ fails to reach $P_e = 100$.

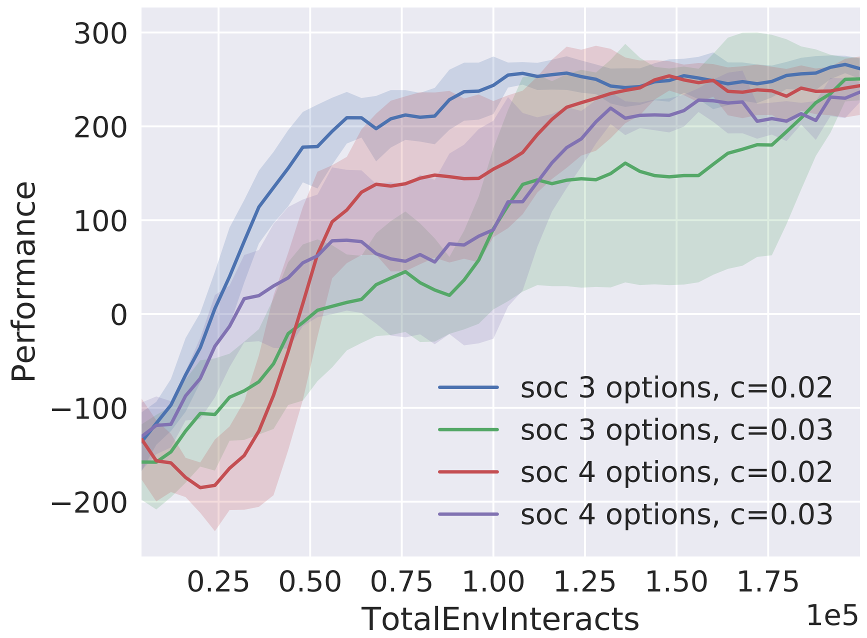


Figure 5.6: Plot of the performance P_e during the training process for **SOC** with $\mathbf{N}_\omega = \mathbf{3}$ and $\mathbf{N}_\omega = \mathbf{4}$ options on *lunar lander*, comparing different values for c when $\alpha = 0.1$. P_e is smoothed with a moving average of $S = 5$. The shaded area denotes the standard deviation of P_e across seeds. The combination of parameters $N_\omega = 3$ with $\alpha = 0.1$ and $c = 0.02$ is the best performing combination of parameter values.

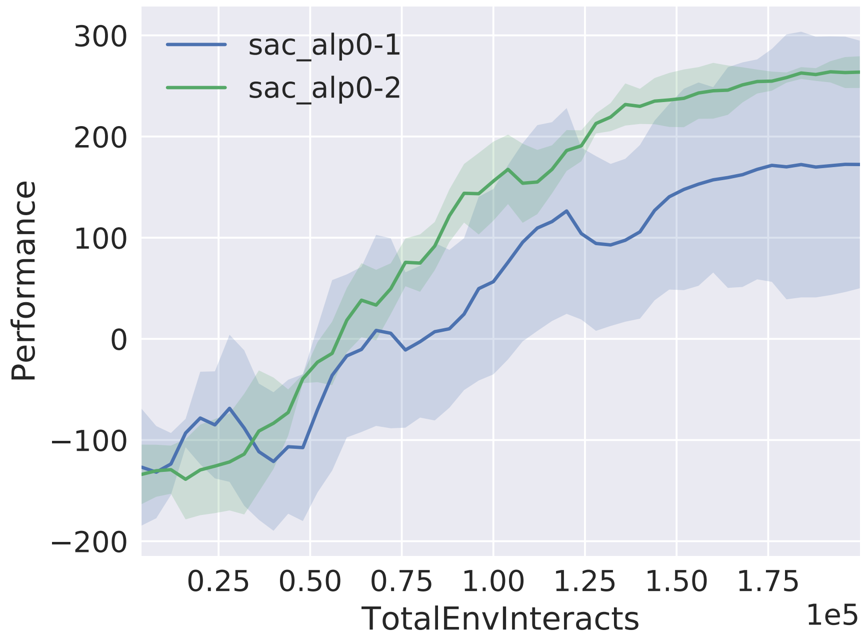


Figure 5.7: Plot of the performance P_e during the training process for **SAC** on *lunar lander*, smoothed with a moving average of $S = 5$. Different values for α are compared. The shaded area denotes the standard deviation of P_e across seeds. The best performance is achieved with $\alpha = 0.1$.

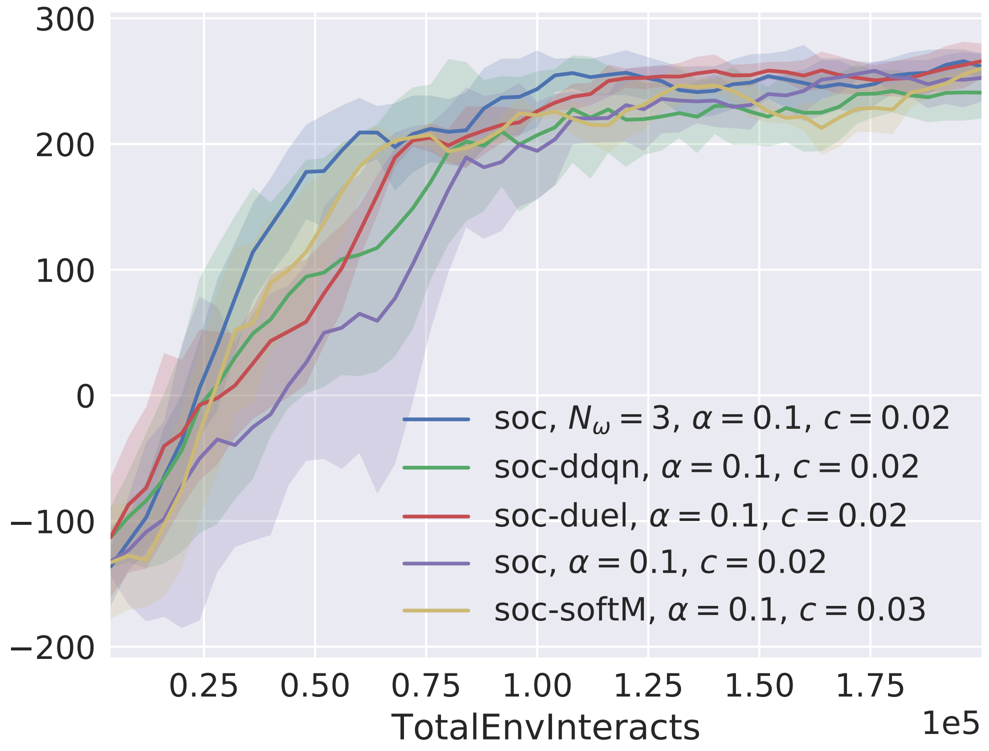


Figure 5.8: Plot of the performance P_e during the training process, comparing the different SOC methods on *lunar lander*, using the best parameters for each algorithm. P_e is smoothed with a moving average of $S = 5$. The shaded are denotes the standard deviation of P_e across seeds. SOC with 3 options converges slightly faster, but all methods solves the environment before $t_l = 1.0$.

Figure 5.9: Table showing when (measured in t_l) each *SOC-method* reached a certain performance P_e during the training process for lunar lander, highlighting the sample-efficiency of the algorithms. Lower value for $t_l|P_e = x$ is better, implying less training samples before reaching a given performance. The best performing parameters are selected for each algorithm.

Methods	$P_e = 0$	$P_e = 100$	$P_e = 200$
SOC	0.43	0.71	1.02
SOC$_{3\omega}$	**0.23**	**0.34**	**0.57**
SOC-DDQN	0.26	0.53	0.83
SOC-duel	0.29	0.56	0.71
SOC-softM	0.27	0.44	0.67

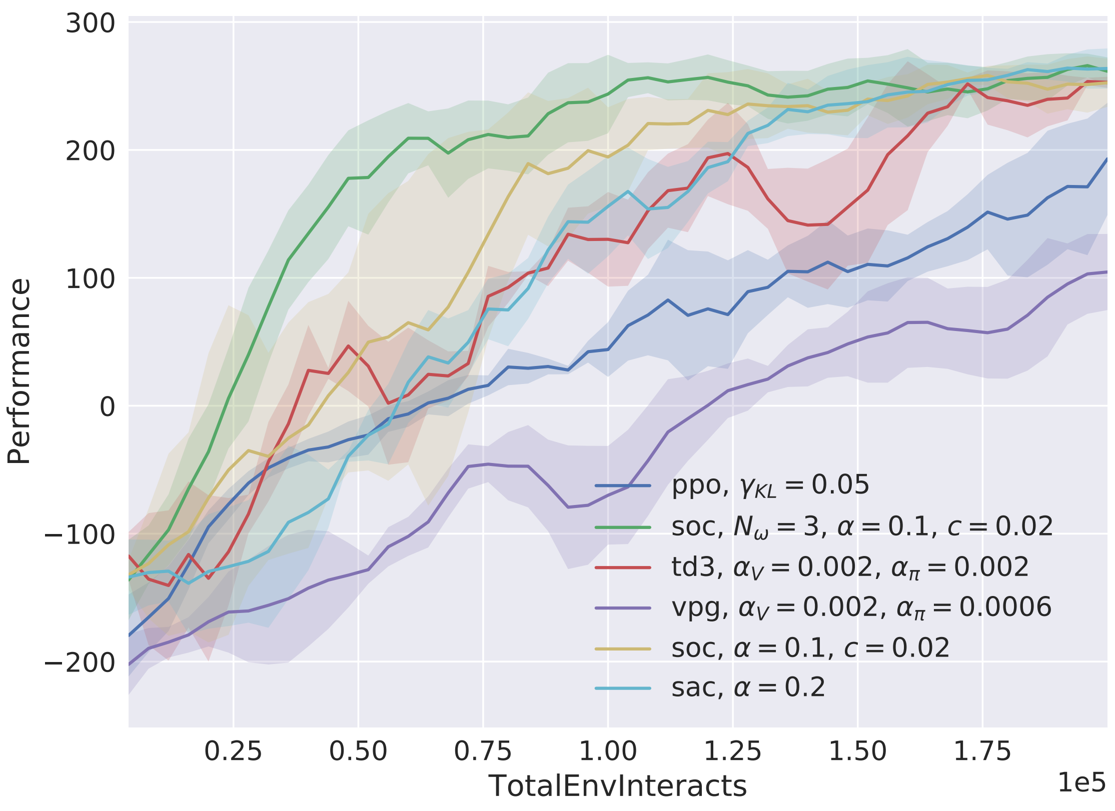
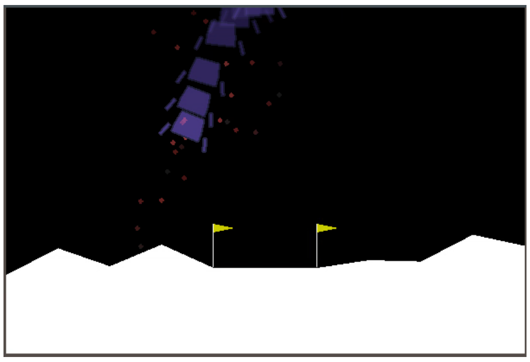


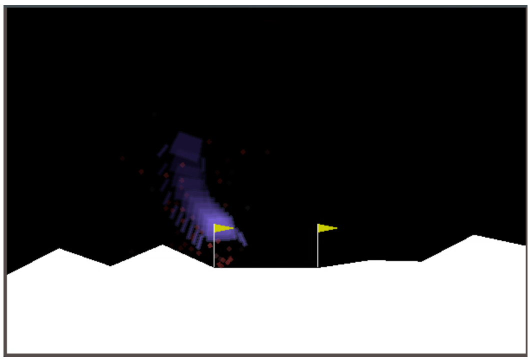
Figure 5.10: Performance plot comparing SOC and the benchmark models on *lunar lander*, using the best parameters for each method. Soc with 3 options is clearly the best, outperforming the rest while reaching Performance of 200 much earlier than the others.

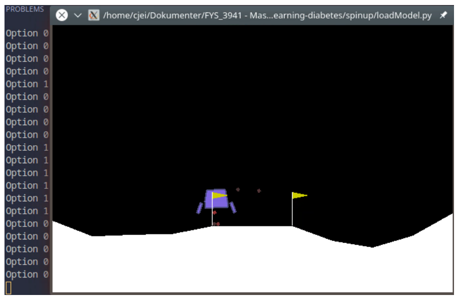
Figure 5.11: Table showing when (measured in t_l) each algorithm reached a certain performance P_e during the training process for lunar lander, highlighting the sample-efficiency of the algorithms. Lower value for $t_l|P_e = x$ is better, implying less training samples before reaching a given performance. The best performing parameters are selected for each algorithm.

Methods	$t_l\|P_e = 0$	$t_l\|P_e = 100$	$t_l\|P_e = 200$
PPO	0.63	1.34	—
SAC	0.58	0.85	1.25
SOC	0.43	0.71	1.02
SOC$_{3\omega}$	**0.23**	**0.34**	**0.57**
TD3	0.37	0.83	1.57
VPG	1.2	1.95	—

5.4.3 Analysis of Options for Lunar Lander


(a) The beginning of the episode. The lunar lander rotates clockwise, setting up a trajectory towards the goal, mainly selecting option 1 at the start to initiate a good trajectory (accelerating to the right), while mainly using option 0 to control it, only focusing on small adjustments.


(b) The lunar lander falls too fast, potentially risking crashing. The trajectory needs adjustment, thus option 1 is selected, producing an extra boost using the main thruster, slowing the vertical momentum while still moving towards the goal.


PROBLEMS
/home/cjei/Dokumenter/FYS_3941 - Mas...earning-diabetes/spinup/loadModel.py

```
Option 0
Option 0
Option 0
Option 0
Option 1
Option 0
Option 0
Option 0
Option 0
Option 1
Option 1
Option 1
Option 1
Option 1
Option 1
Option 0
Option 0
Option 0
Option 0
Option 0
```

(c) Using option 0, the lunar lander makes small adjustments and lands safely.

Figure 5.12: Three figures showing the trajectory of the lunar lander with $N_\omega = 2$ options during an episode, illustrating the abstraction of actions. Each frame in the images was sampled at a constant frame rate, allowing us to better observe the momentum of the lander.

Figure 5.12 shows the trajectory of a trained SOC-agent during an episode. This agent seems to have interpretable options that specialize to some notion of abstract actions. This specialization seems to be a pattern after running the environment multiple times with the trained model.

Option 0 (ω_0) seems to handle the *general control* of the lunar lander, making only small adjustments. This can be seen in figure 5.12a where the agent uses ω_0 to rotate slightly clockwise to shift its momentum towards the landing site.

The agent was able to augment the trajectory towards the goal, but it falls too fast. Figure 5.12b shows where ω_1 is selected, making a substantial change to the trajectory necessary to avoid crashing. Option ω_1 seems to be used for *major adjustments of the lunar lander*. These options could allow the agent to more efficiently plan ahead, which might explain the improved sample efficiency of SOC compared to the other algorithms, as seen in figure 5.10 and table 5.11.

Although it seems to be the case that SOC including all its extensions outperforms the other algorithms, the generated options are not always as intuitive or interpretable as seen in 5.12. The apparent specialization sometimes varies from run to run and across different seeds. Though to interpret the options is a matter of *perception*, which is inherently subjective. Even for artificial intelligence (AI) systems such as *AlphaGo* [47] playing the board game *Go*, which does not make use of options, the agent sometimes seem to make decision that are sub-optimal or mistakes in they eyes of a human expert [48]. In hindsight, they were seen as good moves in this case, going on to beat one of the world's best players 4-1 [48]. Clearly, understanding the though process of an AI system is a difficult undertaking.

To summarize the results from table 5.11 and figure 5.10, SOC, SAC and TD3 are able to solve lunar lander within $N_e = 50$ epochs, reaching $P_e = 200$ by the end of training. PPO almost achieved $P_e = 200$ by the last epoch while REINFORCE is the worst performing method, reaching $P_e = 100$. We see that SOC managed to solve the environment using $N_\omega = 3$ options, reaching $P_e = 200$ before the next best method (SAC) even got $P_e = 0$. Additionally, comparing 5.9 and 5.11 reveal that all SOC-extensions were able to *outperform all benchmark models*. The implementation of SOC is similar to SAC with regards to many implementation details and parameter values as described in ref. secMod. This indicates that SOC takes advantage of options during the learning stage, which results in an improved sample efficiency.

The extensions of SOC perform comparatively to the original implementation as seen in figure 5.8. They all solve the environment, performing best for parameters $\alpha = 0.1$ and $c = 0.02$. Some of them improved the sample-efficiency compared to the original. While SOC-DDQN and SOC-softmax are an improvement as seen in figures 5.3 and 5.6, the difference is marginal. Additionally for SOC-DDQN the mean of P_e has greater variance, revealing a worse robustness across seeds. The most notable improvements were for SOC-duel and SOC with $N_\omega = 3$, where SOC-duel consistently performs well across different values for c and across seeds. Based on these results, the extensions for evaluation on the diabetes experiments are restricted to SOC-duel, as it was the one method with a substantial improvement compared to the original implementation of SOC. The follow sections introduce these experiments and results.

5.5 Diabetes experiments setup

The main goal of the following experiments is to test whether selected RL-methods are suitable for automatic blood glucose control for patients with type 1 diabetes. Specifically, we want to answer the question: *Can* state-of-the-art policy-based RL methods, especially SOC, improve upon the *optimal baseline*? We define the optimal baseline (OB) as the policy that always selects the optimal basal rate bg_{opt} adapted for the simulated patient. Specifically, we have that $\pi_{\mathrm{OB}}(A_t = bg_{opt}|s_t) = 1$ where $bg_{opt} = 6.41$.

To emulate real-life for a T1D patient, two scenarios were developed using a diabetes simulator. The baseline scenario (*baseline diabetes*) is simulating 1.5 days for a 70kg individual which includes a meal schedule. To make the scenario more realistic, both the amount of carbohydrates (CHO) ingested m_g (measured in grams) and the time of ingestion m_t were made stochastic. The schedule and meal amounts used in the experiments are based on the work of Fathi et al. [49]. Specifically, the schedule consists of:

1. $(40 + \sigma_{m_g})$ g breakfast at 08:00 $+\sigma_{m_t}$ min,
2. $(80 + \sigma_{m_g})$ g lunch at 12:00 $+\sigma_{m_t}$ min,
3. $(60 + \sigma_{m_g})$ g dinner at 18:00 $+\sigma_{m_t}$ min,
4. $(30 + \sigma_{m_g})$ g supper at 22:00 $+\sigma_{m_t}$ min,

were $\sigma_{m_g} \sim \mathcal{U}(-30, 30)$ and $\sigma_{m_t} \sim \mathcal{U}(-40, 40)$ with resolution of 3 min are the discrete uniform noises added for the meal amount and meal times respectively [1].

30 minutes before each meal a bolus is given based on the estimated amount of CHO ingested $\hat{m}_g$. this is part of the environment and is given automatically, it is *not administered by the RL-agent.* Similarly to real life, estimating meal amount is not perfect, therefore some noise was added to the estimate:

$$\hat{m}_g = m_g + \mathcal{U}(-0.3m_g, 0.3m_g) \tag{5.2}$$

[1]New noise samples σ_{m_g} and σ_{m_t} were generated for each meal - they did not use the same noise across meals.

the described scenario works as the baseline for simulating a T1D patient, with stochasticity added for *when* the patient eats and the *amount* ingested, including counting errors with regards to the bolus. This reflects reality decently well when the patient has a good routine, always eats 4 meals approximately at the same time every day, while always giving bolus before meals. But in real life plans change and mistakes happen - sometimes you skip a meal, or even forget to give bolus.

The second scenario which we denote as *advance diabetes* simulates exactly that, dropping a meal or bolus with a probability $q = 0.1$. Specifically we have that:

$$\tilde{m}_g = x_1 m_g \tag{5.3a}$$

$$\tilde{m}_t = x_2 m_t, \tag{5.3b}$$

where $\tilde{m}_g$ and $\tilde{m}_t$ denotes the new meal amount and meal time respectively, while x_1, x_2 are samples from the Bernoulli distribution with $P(X = 1) = 1 - q$, $P(X = 0) = 1 - q$. Notably this scenario does *not* include the case where a meal is dropped, but the bolus is still given.

The comparison procedure is as follows:

1. Compare the different SOC-modifications to see whether they are usable*.
2. Compare the benchmark algorithms' parameters and discuss their effect.
3. Select the best performing parameter values for all methods, and evaluate their performance against each other.

What is good blood glucose control? To effectively compare the performance of the algorithms, new metrics are introduced that help answer this question. The recommended advice is to maintain BG within the range of 70-180 mg/dL [40]. We define this range as I, with the endpoints defined as $I_{\text{low}} = 70$ and $I_{\text{high}} = 180$, while the target BG-value was set to $bg_{\text{ref}} = 108$.

time-in-range (TIR) is a metric specifying the percentage of time spent within this interval per day [40, 41, 50]. Complimentary, TAR and TBR defines the time-above-range and time-below-range respectively. Specifically, they are defined as:

$$\text{TIR} = \frac{N_{\text{TIR}}}{N} \cdot 100, \tag{5.4a}$$

$$\text{TAR} = \frac{N_{\text{TAR}}}{N} \cdot 100, \tag{5.4b}$$

$$\mathrm{TBR} = \frac{N_{\mathrm{TBR}}}{N} \cdot 100, \tag{5.4c}$$

where N_{TIR} denotes the number of samples within the interval, N_{TAR} and N_{TBR} denotes the number of samples above- and below the range respectively, and N denotes the total number of samples

5.5.1 Diabetes simulator

The basis for the simulator is the *Hovorka Cambridge* model [51, 52]. It is integrated within OpenAI's gym software [46], based on a forked version from Jonas Myhre's repository[2], which introduced the `HovorkaCambridge-v0` environment. The environment used in this thesis[3] include further modifications on the reward function, state- and action-space, which are described in detail later.

The actual simulator mainly consists of these components:

1. The simulated patient, specified by parameters such as weight and insulin sensitivity.
2. a CGM, monitoring the BG with a time resolution of 1 min.
3. An insulin pump, the interface for regulation of BG by selection of insulin dosages.
4. the internal equations and parameters governing the glucose-insulin dynamics.

A key feature of the model is that the glucose-insulin dynamics includes the inherent delay that characterizes glucoregulatory system [7, 51, 52]:

- The delay between infusion of insulin subcutaneous tissue and the absorption in the blood
- The delay between ingestion- and absorption of CHO in the blood.

Optimally we would like for the agent to have the ability to account for the delayed dynamics, while also being adaptive to the stochastic nature of CHO counting errors and meal times. For further details including specific parameters describing the model used in the simulator, we refer the reader to [51].

[2]https://github.com/jonasnm/gym

[3]The code can be found at https://github.com/cjenssen0/gym on the branches `spinup-diabetes-normAll` and `diabetes-prob-noBolus_noMeals`.

Since there is no standard or established definition for how the environment should be, design decisions has to be made for some of the central components necessary to describe the problem as an MDP. Still, as mentioned, the basis will be on the forked gym-version. The next subsection introduces these components.

5.5.2 State-space

For an MDP to satisfy the Markov property, *all relevant information of the past* should be encapsulated within the state s [2, 19]. Naturally, only including the last BG-measure $bg_{(t)\,\mathrm{min}}$[4] does not the capture whether the BG-levels are on a rising- or falling trajectory, information necessary for optimal control. Additionally, because of the *delayed effect* from the insulin dosages, the duration of time between states s_t and s_{t+1} has to be long enough such that effects of different insulin values can be observed. If not, the agent would potentially receive the same reward r in some states *regardless of actions*, which is detrimental to learning.

Thus there is a trade-off when selecting the time-resolution. If it is too fine-grained the agent can't infer the effect of different actions, while too long duration between time-steps limits the potential for the agent to adapt rapidly to changes in the state. Similarly to other implementations [7].?, the time-resolution was set to 30 min of BG-levels, resulting in 30 BG measures each time-step t:

$$f_{\mathrm{BG}}(t) = [bg_{(t-29)\,\mathrm{min}},\, bg_{(t-28)\,\mathrm{min}}, \ldots,\, bg_{(t)\,\mathrm{min}}], \tag{5.5}$$

where $bg_{(t)\,\mathrm{min}} \in \mathbb{R}_{0\rightarrow 500}$.

Another factor that the *delayed effect* of insulin incur is that not only does it take 30-60 minutes for the insulin to take effect, it also last for a long time, peaking after 2-5 hours. For the agent to be able to learn a non-trivial policy, it needs to have the opportunity to infer how much insulin is already in the system. To account for this, information about the insulin dosages spanning the past 2 hours was added to the state:

$$f_{\mathrm{insulin}}(t) = [a_{t-4},\, a_{t-3},\, a_{t-2},\, a_{t-1}] \tag{5.6}$$

Though the environment still isn't defined as a proper MDP because of the non-stationarity introduced by the meals which induces sudden spikes in BG.

[4]When in the context of bg, $(t)\,\mathrm{min}$ refers to the minutes that have past in the episode, not time-steps of the environment.

Essentially, the same state s produces different reward r and subsequent state s' from the same actions depending on whether the patient eats a meal or not. This effect is known as *perceptual aliasing* [53], the patient could be experiencing differing events affecting BG, such as meals or forgetting to give bolus, but the agent *perceives* them as the same state. Intuitively, this effect is outside the agent's observational scope when only including f_{BG} and f_{insulin} in the state representation, but has a substantial effect, which increases difficulty for the agent to perform adequately. I propose to add the time t as a component to alleviate this problem, so that the agent could potentially infer approximately when it should expect meals to arrive:

$$f_{\text{time}}(t) = [t] \tag{5.7}$$

Combining all the defined feature representation-components together results in the definition of the state:

$$\mathbf{S_t} = [f_{\text{BG}}(t),\ f_{\text{insulin}}(t),\ f_{\text{time}}(t)] \tag{5.8}$$

Additionally, all features in S_t was standardized to be in the range $[-1, 1]$ to make sure that they are on a similar scale. The reason is that features with greater scale of magnitude may have larger influence on the neural network, thus artificially skewing their importance [23]. The standardization was also applied on the reward function and action-values.

5.5.3 Reward function

The goal of an insulin control algorithm is to keep the BG as stable as possible within TIR. To accomplish this, the reward function is designed to positively reward the agent when it maximizes TIR and punish it when outside the optimal interval I. Concretely, the agent gets a negative reward when the BG is outside I, where the reward function is biased towards punishing hypoglycemia more than hyperglycemia. The reward for $bg_t = bg_{\text{ref}}$ is set to be marginally larger than $bg_t \in$ TIR. This allows the agent to navigate within I to better prepare for future events such as meals and dropped bolus, instead of getting tunnel vision by reaching $bg_t = bg_{\text{ref}}$ at all cost only short-term. Figure 5.13 illustrates this idea clearly: $r \geq 0.5$ when BG inside optimal range, drops to $r = 0$ outside and falls when moving further away while being more heavily biased to punish too low BG-levels than high.

The reward function is a piece-wise function of skewed gamma- and Normal

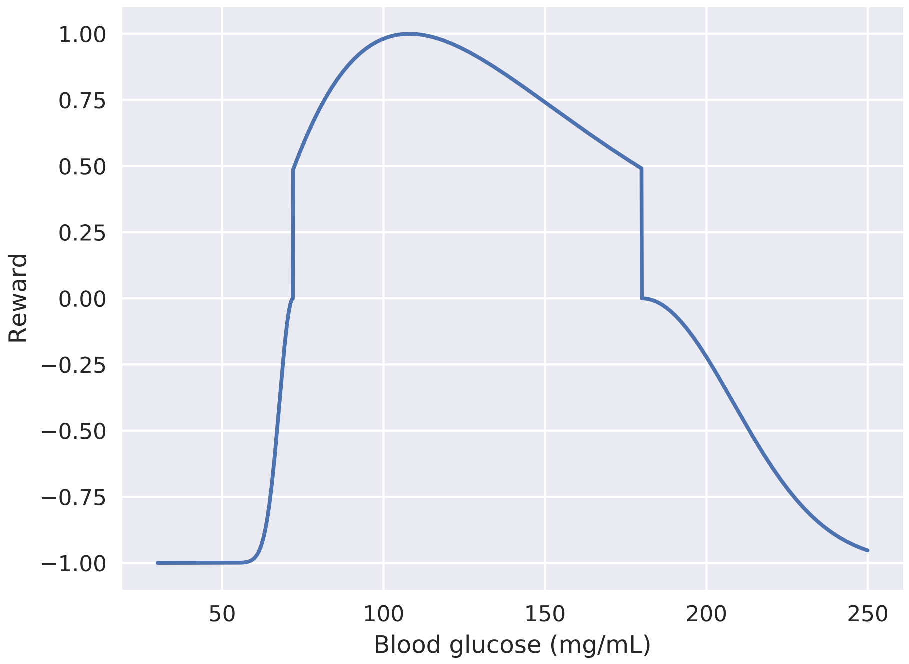


Figure 5.13: The piece-wise reward function $R(x)$ for the diabetes environment, ranging from $r = 1$ to $r = -1$. The reward drops from 0.5 to 0 at the limits of the optimal range $[I_{\text{low}} + 2, I_{\text{high}}]$, with the goal that this induces the agent to mainly focus on staying within this interval. As hypoglycemia is more dangerous than hyperglycemia, the agent is punished more for low BG than high.

distributions. Specifically, it is defined as:

$$R(\text{x}) = \begin{cases} c_{\text{low}} N(x;\ \mu = 72, \sigma = 4.24) - 1, & \forall x \in (-\infty, 72) \\ c_{\Gamma}\Gamma(x;\ k = 2.3, \theta = 38.46, \mu = 58.0), & \forall x \in [72, 180] \\ c_{\text{high}} N(x; \mu = 180, \sigma = 28.28) - 1, & \forall x \in (180, \infty), \end{cases} \tag{5.9}$$

where $c_{\text{low}} = 10.63$, $c_{\Gamma} = 117.07$ and $c_{\text{low}} = 70.90$ are standardization constants such that the distribution are transformed to the range $[0, 1]$.

To keep the reward at each time-step t in range $r \in [-1, 1]$, we divide by the number of BG-values (30 min):

$$R_t = \frac{1}{30} \sum_{i}^{i+30} R(bg_{(i)\,min}). \tag{5.10}$$

5.5.4 Action-space

The range of insulin values considered were the same as in earlier work, defining a 1-dimensional continuous action-space $\mathcal{A} = \{a \in [0, 2b^*]\}$ were $b^* = 6.43$ [mU/min] is the optimal basal rate.

At a state s_t the agent selects action $a_t \in [0, 2b^*]$, being the amount of insulin injected by the insulin pump each minute during the transition to the next state s_{t+1}, totaling 30 min.

5.6 Results and analysis for diabetes

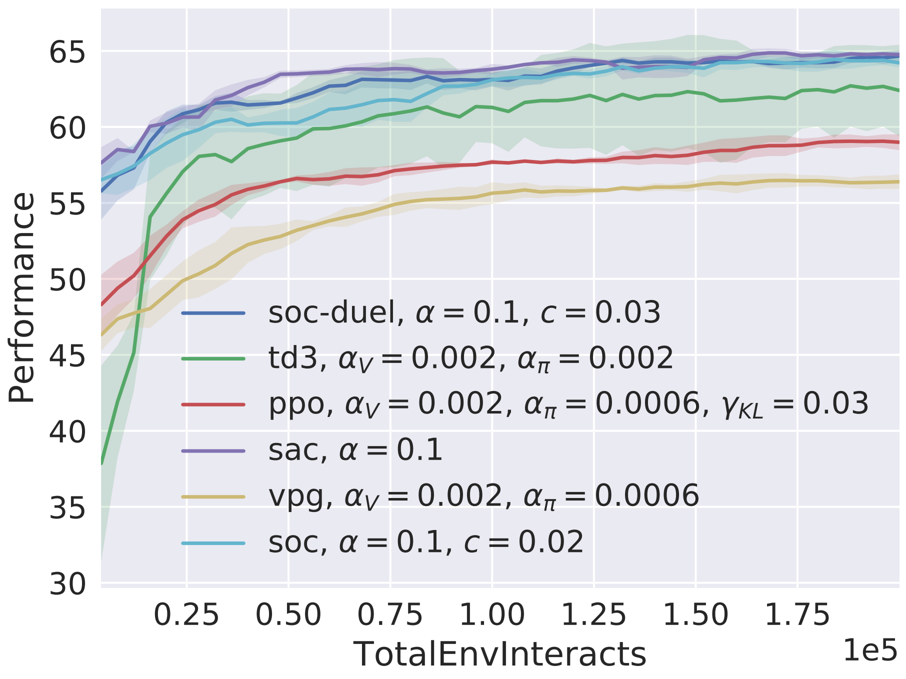
Similarly as section 5.3, the performance while training on the diabetes environment will be evaluated for the selected algorithms.

Table 5.1: Table of the performance metrics averaged over 100 test runs on the *baseline diabetes* environment, where the best results are bolded. For $\bar{R}$ and TIR higher value is better, while for σ_{BG}, TAR and TBR lower is better. For μ_{BG} the closer it is to $bg_{\mathrm{ref}} = 108$ the better.

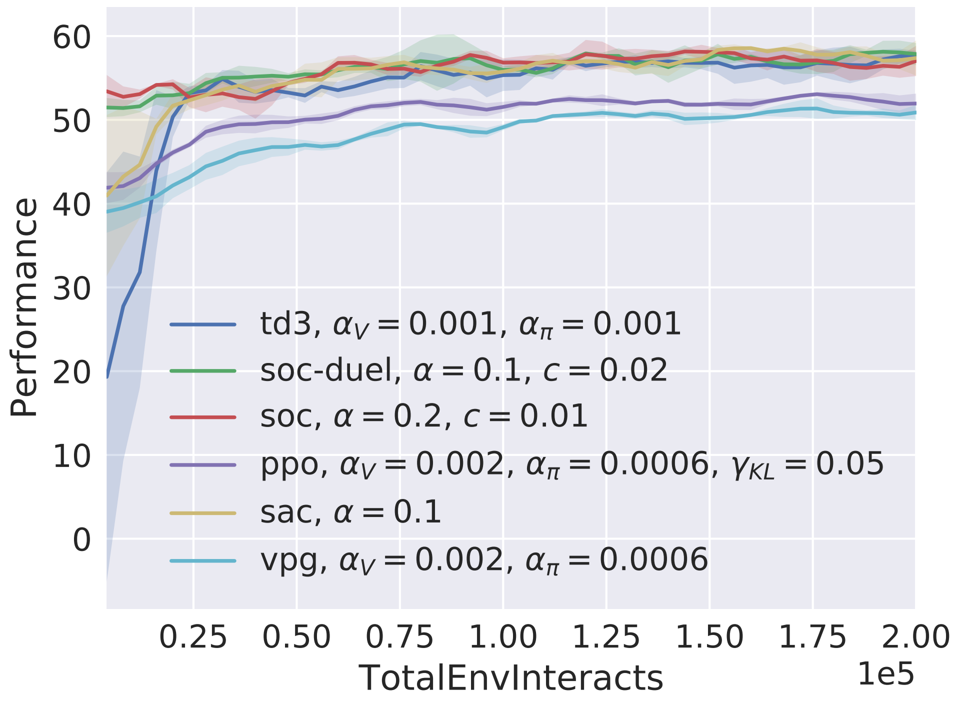
Methods	$\bar{R}$	TIR (%)	TAR (%)	TBR (%)	μ_{BG}	σ_{BG}
OB	52.9	87.6	10.8	1.5	128	10.1
PPO	59.8	92.9	5.2	1.9	120.0	10.1
SAC	64.6	**97.7**	1.0	1.3	108.1	10.1
SOC	64.5	**97.7**	1.3	1.0	110.4	10.1
SOC-duel	**64.8**	97.6	1.3	1.1	111.8	10.5
$\mathrm{SOC}_{3\omega}$	63.7	96.7	2.6	**0.7**	116.3	8.8
TD3	64.1	**97.7**	1.4	0.9	118.5	8.8
VPG	56.4	90.4	7.0	2.7	119.9	9.0

Table 5.2: Table of the performance metrics averaged over 100 test runs on the *advance diabetes* environment, where the best results are bolded. For $\bar{R}$ and TIR higher value is better, while for σ_{BG}, TAR and TBR lower is better. For μ_{BG} the closer it is to $bg_{\mathrm{ref}} = 108$ the better.

Methods	$\bar{R}$	TIR (%)	TAR (%)	TBR (%)	μ_{BG}	σ_{BG}
OB	47.3	83.8	15.3	0.9	137.5	21.7
PPO	53.5	87.8	9.0	3.2	121.9	18.7
SAC	57.4	89.9	4.6	5.5	113.9	18.7
SOC	**59.5**	**93.2**	5.4	1.4	116.3	17.6
SOC-duel	57.9	91.3	6.6	2.0	122.0	21.0
$\mathrm{SOC}_{3\omega}$	57.4	91.1	8.7	**0.2**	126.4	19.3
TD3	59.2	91.5	**3.7**	4.8	**112.3**	**16.9**
VPG	51.1	86.4	10.7	2.9	128.0	20.5



(a) Performance plot comparing the algorithms on the **baseline diabetes** environment during training, using the best parameters for each method. SOC-duel and SAC perform best, both converging towards $P_e = 65$.



(b) Performance plot comparing the algorithms on the **advance diabetes** environment during training, using the best parameters for each method. SOC-duel and SAC perform best, both converging towards $P_e = 58$.

Figure 5.14: Figure showing performance plots for the *baseline*- and *advance diabetes* environment during training. For both scenarios SOC-duel and SAC are the best performing methods. TD3 does eventually converge towards the same values, but has worse performance in the early stages of training.

From figure 5.14a we see that SOC does indeed improve while training on baseline diabetes scenario, converging towards performance $P_e = 60$. SOC and SOC-duel have quite similar performance, not exhibiting the same boost in sample-efficiency as the experiments in section 5.4. Compared against the benchmark models they outperform most of them. The exception is SAC, which has slightly improved performance in the early stages of training. TD3 is the closest of the remaining benchmark model matching the performance of SOC, SOC-duel and SAC. It converges towards the same $P_e = 60$, but with worse performance in the early stages, achieving $P_e = 20$ at the first epoch. PPO and REINFORCE improve consistently across seeds, but are not able to reach $P_e = 50$.

From figure 5.14b we see that the performance during training on *advance diabetes* exhibits a similar pattern as in *baseline diabetes.* Again SOC, SOC-duel and SAC are the best performing algorithms, converging towards $P_e = 60$ in this case. Impressively, SOC and SOC-duel achieve $P_e > 50$ at the first epoch, having the best sample efficiency among the tested methods.

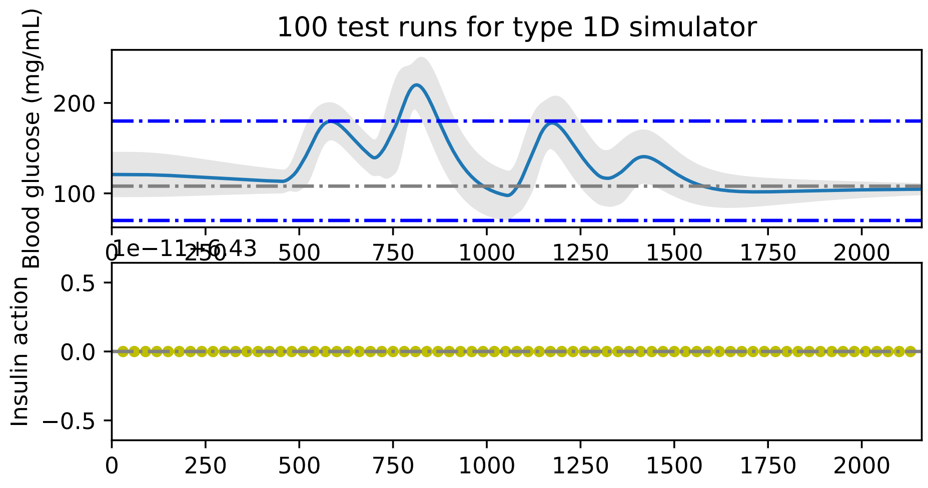
Comparing the tables 5.1 and 5.2 we see that SOC achieves TIR= 97.7% while only having $TAR = 10.8\%$ and TBR= 1.0%. In contrast, OB achieves TIR= 87.6%, $TAR = 10.8\%$ and TBR= 1.5%. This difference is amplified for *advance diabetes*, where SOC achieves TIR=93.2. Against the benchmark models SOC performs comparatively, with $\text{SOC}_{3\omega}$ achieving the best TBR: 0.7% and 0.2% for baseline- and advance diabetes respectively, while still reaching similar TIR.

5.6.1 Analysis of specialized options for diabetes

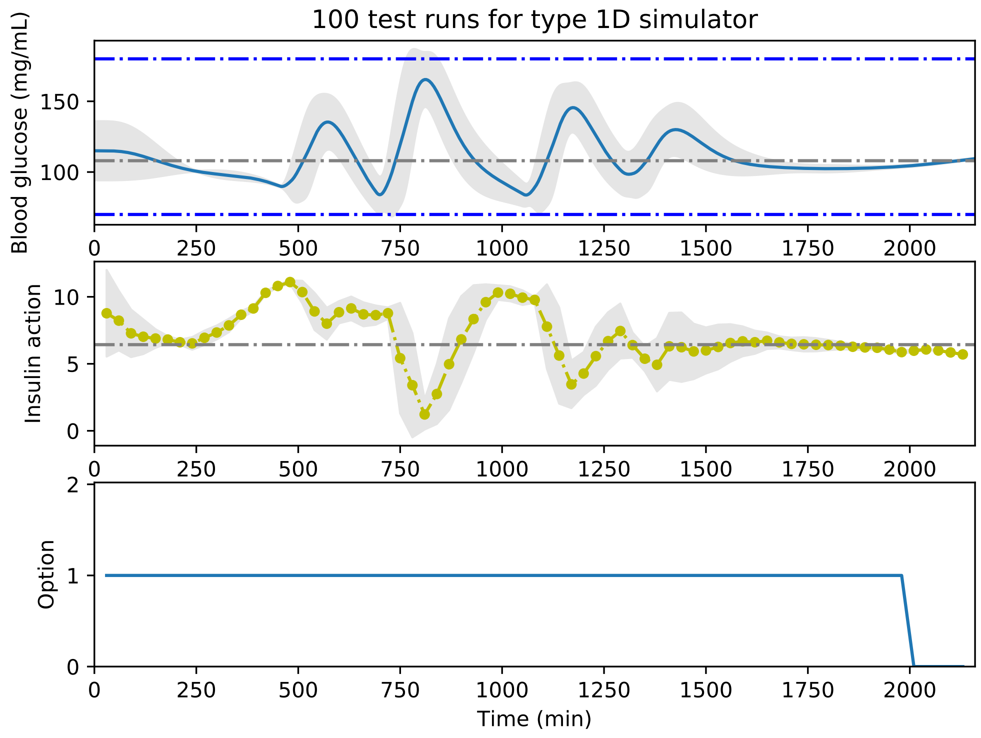
In this subsection we take a closer look at the BG-curves and the corresponding insulin actions. The comparison is made

From figures 5.15 and 5.16 we see that SOC is able to predict meals, giving bigger dosages ahead of the BG-spikes. Even for the *advance diabetes* case, it seems to control the BG-levels, even though it can't be as opportunistic giving big dosages ahead of meals.

It is difficult to say whether SOC learned specialized options. For *baseline diabetes* the algorithm converged to only selecting ω_1. Although for *advanced diabetes* it seems to be the case that ω_0 mainly is selected during times when the patient could decide to eat.

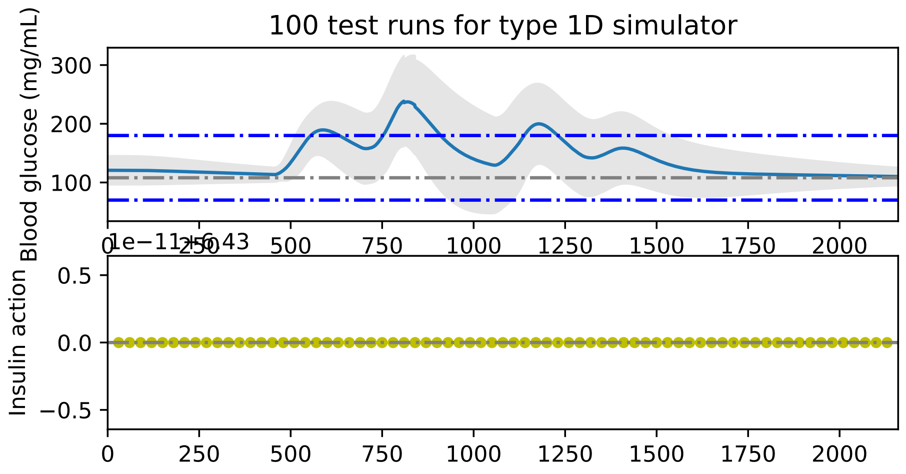


(a) Figure of $N_{\text{test}} = 100$ test runs for OB, showing the mean- and standard deviation of BG-levels and insulin actions.

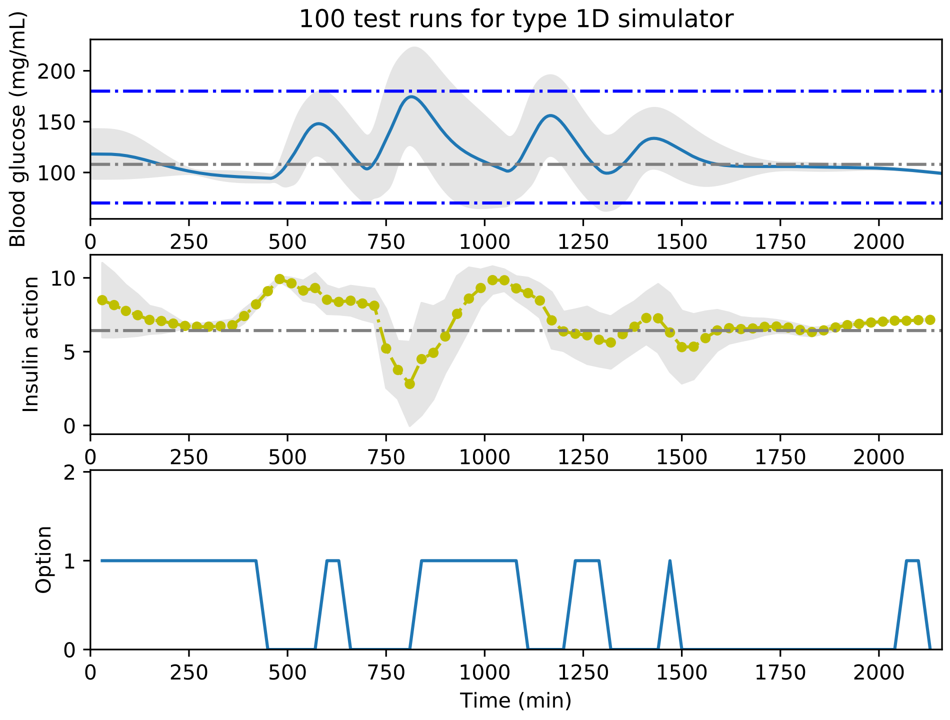


(b) Figure of $N_{\text{test}} = 100$ test runs for SOC, showing the mean- and standard deviation of BG-levels and insulin actions. Additionally the corresponding the most frequent option at each time-step is shown.

Figure 5.15: Figure of $N_{\text{test}} = 100$ test runs on *baseline diabetes*, showing the mean- and standard deviation of BG-levels and insulin actions. Additionally the corresponding the most frequent option at each time-step is shown.



(a) Figure of $N_{\text{test}} = 100$ test runs for OB, showing the mean- and standard deviation of BG-levels and insulin actions. Additionally the corresponding the most frequent option at each time-step is shown.



(b) Figure of $N_{\text{test}} = 100$ test runs for SOC, showing the mean- and standard deviation of BG-levels and insulin actions. Additionally the corresponding the most frequent option at each time-step is shown.

Figure 5.16: Figure of $N_{\text{test}} = 100$ test runs on *advance diabetes*, showing the mean- and standard deviation of BG-levels and insulin actions. Additionally the corresponding the most frequent option at each time-step is shown.

Chapter 6

Conclusion

The purpose of this thesis was to evaluate state-of-the-art policy-based RL-methods for controlling blood glucose control in T1D. A new method called SOC was developed with this goal in mind.

SOC achieved improved sample-efficiency against the benchmark models on lunar-lander. The results and analysis of options indicated that reason for this improvement was because of option specialization.

On the diabetes experiments SOC and its extensions performed comparatively to the best performing algorithms. SOC with 3 options achieved the best TBR for both scenarios: TBR= 0.7% and TBR= 0.2% for baseline- and advance diabetes respectively, while still reaching similar TIR to the other algorithms.

Bibliography

[1] GGM MED, "Artificial Pancreas – GGM Med," 2018.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* Cambridge, Mass: MIT Press, 2018.

[3] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, vol. 112, pp. 181–211, Aug. 1999.

[4] Helsedirektoratet, "Diabetes type 1."

[5] J. Wood, *The Type 1 Diabetes Self-Care Manual: A Complete Guide to Type 1 Diabetes across the Lifespan for People with Diabetes, Parents, and Caregivers.* Arlington: American Diabetes Association, 2018.

[6] Helsebiblioteket, "Diabetes type 1."

[7] J. N. Myhre, I. K. Launonen, S. Wei, and F. Godtliebsen, "Controlling Blood Glucose Levels in Patients with Type 1 Diabetes using Fitted Q-Iterations and Functional Features," in *2018 IEEE 28th International Workshop on Machine Learning for Signal Processing (MLSP)*, pp. 1–6, Sept. 2018.

[8] M. Tejedor, A. Z. Woldaregay, and F. Godtliebsen, "Reinforcement learning application in diabetes blood glucose control: A systematic review," *Artificial Intelligence in Medicine*, vol. 104, p. 101836, Apr. 2020.

[9] I. Fox and J. Wiens, "Reinforcement Learning for Blood Glucose Control: Challenges and Opportunities," p. 8, 2019.

[10] O. Sand, O. V. Sjaastad, E. Haug, J. G. Bjaalie, and K. C. Toverud, *Menneskekroppen: Fysiologi Og Anatomi (2. Utg.).* 2. ed., 2006.

[11] Mayo Clinic, "Type 1 diabetes - Symptoms and causes."

[12] Editor, "A basal-bolus injection regimen involves taking a number of injections through the day, https://www.diabetes.co.uk/insulin/basal-bolus.html," Jan. 2019.

[13] M. K. Bothe, L. Dickens, K. Reichel, A. Tellmann, B. Ellger, M. Westphal, and A. A. Faisal, "The use of reinforcement learning algorithms to meet the challenges of an artificial pancreas," *Expert Review of Medical Devices*, vol. 10, pp. 661–673, Sept. 2013.

[14] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play," *Science*, vol. 362, pp. 1140–1144, Dec. 2018.

[15] C. Berner, G. Brockman, B. Chan, V. Cheung, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with Large Scale Deep Reinforcement Learning," p. 66, 2019.

[16] E. Alpaydin, *Introduction to Machine Learning.* MIT press, 3rd ed., 2014.

[17] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016.

[18] C. Szepesvári, "Algorithms for Reinforcement Learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 4, pp. 1–103, Jan. 2010.

[19] S. M. Ross, J. J. Kelly, R. J. Sullivan, W. J. Perry, D. Mercer, R. M. Davis, T. D. Washburn, E. V. Sager, J. B. Boyce, and V. L. Bristow, *Stochastic Processes*, vol. 2. Wiley New York, tenth ed., 2009.

[20] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, pp. 279–292, May 1992.

[21] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy Gradient Methods for Reinforcement Learning with Function Approximation," in *Advances in Neural Information Processing Systems*, pp. 1057–1063, 2000.

[22] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, pp. 229–256, May 1992.

[23] S. Theodoridis and K. Koutroumbas, *Pattern Recognition & Matlab Intro.* USA: Academic Press, Inc., fourth ed., 2010.

[24] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[25] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-Learning," in *Thirtieth AAAI Conference on Artificial Intelligence*, Mar. 2016.

[26] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv:1707.06347 [cs]*, Aug. 2017.

[27] P.-L. Bacon, J. Harb, and D. Precup, "The option-critic architecture," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[28] S. J. Bradtke and M. O. Duff, "Reinforcement Learning Methods for Continuous-Time Markov Decision Problems," *Advances in neural information processing systems*, pp. 393–400, 1995.

[29] A. G. Barto and S. Mahadevan, "Recent Advances in Hierarchical Reinforcement Learning," *Discrete Event Dynamic Systems*, vol. 13, pp. 41–77, Jan. 2003.

[30] D. Precup and R. S. Sutton, "Multi-time models for reinforcement learning," in *Proceedings of the ICML'97 Workshop on Modelling in Reinforcement Learning*, 1997.

[31] D. Precup and R. S. Sutton, "Multi-time Models for Temporally Abstract Planning," *Advances in neural information processing systems*, pp. 1051–1056, 1998.

[32] R. S. Sutton, D. Precup, and S. Singh, "Intra-Option Learning about Temporally Abstract Actions," *ICML*, vol. 98, pp. 556–564, 1998.

[33] J. Harb, P.-L. Bacon, M. Klissarov, and D. Precup, "When Waiting is not an Option : Learning Options with a Deliberation Cost," *arXiv:1709.04571 [cs]*, Sept. 2017.

[34] M. C. Machado, M. G. Bellemare, and M. Bowling, "A Laplacian Framework for Option Discovery in Reinforcement Learning," *arXiv:1703.00956 [cs]*, June 2017.

[35] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor," *arXiv:1801.01290 [cs, stat]*, Aug. 2018.

[36] B. D. Ziebart, A. L. Maas, J. A. Bagnell, and A. K. Dey, "Maximum entropy inverse reinforcement learning.," in *Aaai*, vol. 8, pp. 1433–1438, Chicago, IL, USA, 2008.

[37] E. Lobo and S. Jordan, "Soft Options Critic," *arXiv:1905.11222 [cs]*, June 2019.

[38] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8026–8037, Curran Associates, Inc., 2019.

[39] J. Achiam, "Spinning up in deep reinforcement learning," 2018.

[40] "Time-in-Range Tips: Expert-Defined Goals, Plus Insights from Almost 500,000 FreeStyle Libre Users," May 2019.

[41] R. W. Beck, R. M. Bergenstal, T. D. Riddlesworth, C. Kollman, Z. Li, A. S. Brown, and K. L. Close, "Validation of Time in Range as an Outcome Measure for Diabetes Clinical Trials," *Diabetes Care*, vol. 42, pp. 400–405, Mar. 2019.

[42] K. Khetarpal, M. Klissarov, M. Chevalier-Boisvert, P.-L. Bacon, and D. Precup, "Options of Interest: Temporal Abstraction with Interest Functions," Jan. 2020.

[43] S. Zhang and H. Yao, "ACE: An Actor Ensemble Algorithm for Continuous Control with Tree Search," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 5789–5796, July 2019.

[44] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, 2012.

[45] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *Proceedings of the 33rd International Conference on Machine Learning* (M. F. Balcan and K. Q. Weinberger, eds.), vol. 48 of *Proceedings*

of Machine Learning Research, (New York, New York, USA), pp. 1995–2003, PMLR, June 2016.

[46] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *arXiv:1606.01540 [cs]*, June 2016.

[47] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, pp. 354–359, Oct. 2017.

[48] C. Metz, "How Google's AI Viewed the Move No Human Could Understand, https://www.wired.com/2016/03/googles-ai-viewed-move-no-human-understand/," *Wired*, Mar. 2016.

[49] A. El Fathi, M. Raef Smaoui, V. Gingras, B. Boulet, and A. Haidar, "The artificial pancreas and meal control: An overview of postprandial glucose regulation in type 1 diabetes," *IEEE Control Systems Magazine*, vol. 38, no. 1, pp. 67–85, 2018.

[50] T. Battelino, T. Danne, R. M. Bergenstal, S. A. Amiel, R. Beck, T. Biester, E. Bosi, B. A. Buckingham, W. T. Cefalu, K. L. Close, C. Cobelli, E. Dassau, J. H. DeVries, K. C. Donaghue, K. Dovc, F. J. Doyle, S. Garg, G. Grunberger, S. Heller, L. Heinemann, I. B. Hirsch, R. Hovorka, W. Jia, O. Kordonouri, B. Kovatchev, A. Kowalski, L. Laffel, B. Levine, A. Mayorov, C. Mathieu, H. R. Murphy, R. Nimri, K. Nørgaard, C. G. Parkin, E. Renard, D. Rodbard, B. Saboo, D. Schatz, K. Stoner, T. Urakami, S. A. Weinzimer, and M. Phillip, "Clinical targets for continuous glucose monitoring data interpretation: Recommendations from the international consensus on time in range," *Diabetes Care*, vol. 42, no. 8, pp. 1593–1603, 2019.

[51] R. Hovorka, V. Canonico, L. J. Chassin, U. Haueter, M. Massi-Benedetti, M. Orsini Federici, T. R. Pieber, H. C. Schaller, L. Schaupp, T. Vering, and M. E. Wilinska, "Nonlinear model predictive control of glucose concentration in subjects with type 1 diabetes," *Physiological Measurement*, vol. 25, pp. 905–920, Aug. 2004.

[52] M. E. Wilinska, L. J. Chassin, C. L. Acerini, J. M. Allen, D. B. Dunger, and R. Hovorka, "Simulation environment to evaluate closed-loop insulin delivery systems in type 1 diabetes," *Journal of Diabetes Science and Technology*, vol. 4, pp. 132–144, Jan. 2010.

[53] A. McCallum, "Instance-Based State Identification for Reinforcement Learning," p. 8.