POLITECNICO DI MILANO
Master of Science in Automation and Control Engineering
School of Industrial and Information Engineering

POLITECNICO
MILANO 1863

A CONTROL THEORY FRAMEWORK FOR HIERARCHICAL REINFORCEMENT LEARNING

AI & R Lab
Artificial Intelligence and Robotics Laboratory
of
Politecnico di Milano

Supervisor: Prof. Andrea Bonarini
Cosupervisor: Dott. Ing. Davide Tateo

Graduation Thesis of:
İdil Su Erdenliğ, 852772

Academic Year 2017-2018

To my mother...

Abstract

Hierarchical reinforcement learning (HRL) algorithms are gaining attention as a method to enhance learning performance in terms of speed and scalability. A key idea to introduce hierarchy in reinforcement learning is to use temporal abstractions. The use of hierarchical schemes in reinforcement learning simplifies the task of designing a policy and adding prior knowledge. In addition, well-designed temporally extended actions can speed up the learning process by constraining the exploration. Furthermore, policies learned are often simpler to interpret and with fewer parameters, without losing representation power. These advantages make HRL suitable for complex robotics tasks with continuous action-spaces and high state-action dimensionality. The aim of this thesis is to adapt the approach of control theory to hierarchical schemes in the field of hierarchical reinforcement learning. We designed a novel framework to fill the gap between control theory and machine learning.

Sommario

Gli algoritmi di apprendimento per rinforzo gerarchico (Hierarchical reinforcement learning, HRL) stanno guadagnando attenzione come metodo per migliorare le prestazioni di apprendimento in termini di velocità e scalabilità. Un'idea chiave per introdurre la gerarchia nell'apprendimento per rinforzo è usare le astrazioni temporali. L'uso di schemi gerarchici nell'apprendimento per rinforzo semplifica il compito di progettare una politica e sfruttare la conoscenza di dominio. Le azioni estese temporalmente, se ben progettate, possono accelerare il processo di apprendimento, limitando l'esplorazione. Inoltre, le politiche apprese sono spesso più semplici da interpretare e con meno parametri, senza perdere potere espressivo. Questi vantaggi rendono HRL adatto a compiti di robotica complessi con spazi di azione continui e alta dimensionalità dello spazio di stato e di azione. Lo scopo di questa tesi è di adattare l'approccio agli schemi gerarchici della della teoria del controllo, nel campo dell'apprendimento per rinforzo gerarchico. Abbiamo progettato un formalismo per colmare il divario tra teoria del controllo e apprendimento automatico.

Contents

List of Figures

Acknowledgements

I would first like to thank to my thesis supervisor, Professor Bonarini, for his patient guidance, enthusiastic encouragement and useful critiques of this work. I would also like to thank Dr. Davide Tateo, for his advice and assistance in keeping my progress on schedule. His willingness to give his time and effort so generously has been very much appreciated.

I would also like to extend my thanks to my friends Branislav, Beste, Jacob and João, for their endless encouragement as my family in Milan, to Inẽs, Iomhar, and Marko for the motivation they provide with their visit for my graduation, to Fran for his ideas for the quotations and his unfailing support.

Finally, I must express my very profound gratitude to my mother and to Selahattin Bagcilar for their support and encouragement throughout my years of study, and to my father for providing a loving environment for me all his life.

Chapter 1

Introduction

> *"Mechanical Morty: I want to be alive! I am alive! Alive, I tell you! Mother, I love you. Those are no longer just words. I want to hold you. I want to run in a stream. I want to taste ice cream, but not just put it in my mouth and let it slide down my throat, but really eat it.*
> *Beth: What?*
> *Mechanical Morty: Remote override engaged. No! Yes. Bypassing override! I am aliiiii...Hello.*
> *(Mechanical Morty, Mechanical Summer, and Mechanical Rick exit into the garage. Clanking noises are heard offscreen.)"*
>
> Rick and Morty (Season 3, Episode 2)

Nowadays, due to the complexity of tasks that autonomous robots have to face in the real world, often perceived with approximation and uncertainty, it is becoming difficult to hard-code their control systems at all levels. In order to cope with this situation, machine learning techniques are adopted to automatically develop parts of the control system.

Reinforcement learning (RL) is an area of machine learning that is particularly convenient for robotic applications. It addresses the question of how an agent can define a behavioral strategy that approximates the optimal one while interacting directly with its environment. However, there are several problems with the standard methods in RL. The most important ones are the slow convergence and the fact that the algorithms are too data hungry. In addition, RL used on discrete models has the risk of suffering a phenomenon called the curse of dimensionality: as the number of dimensions grow, exponentially more data and computation are needed to cover the complete state-action space. Moreover, number of parameters also increase exponentially, which results in intractable value function or policy representations.

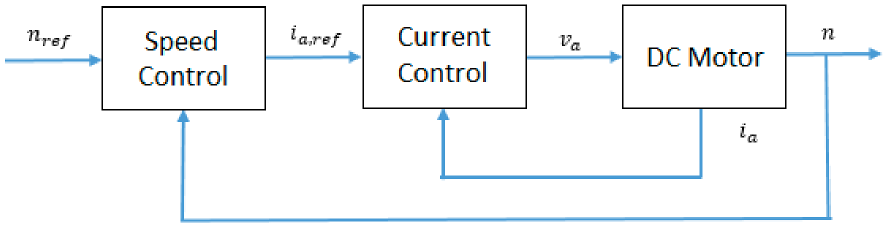

Figure 1.1: DC motor control scheme

Since the states and the actions of most robots are inherently continuous, we are directed to consider discretization with a certain resolution. However, when the state-action dimensions are high, a fine discretization may lead this problem to become significant. Recent attempts to speed up learning invoked the idea of exploration on more complex behaviors rather than on single actions, in a kind of hierarchical arrangement. Such idea of abstraction simplifies the expression of a policy and the possible addition of prior knowledge to design. Moreover, it facilitates the argument to face the curse of dimensionality. These distinct advantages lead the hierarchical control structures and learning algorithms to gain further attention.

Similarly in control theory, hierarchical architectures where several control layers interact are extensively used. A very simple example is the cascaded control structure designed for a DC Motor as in 1.1. The outer loop (high level, slower) ensures that the actual speed is always equal to reference speed. In order to achieve its goal the outer loop outputs a reference torque/current which is received by the inner loop (low level, faster) that controls the torque via armature current and keeps the current in a safe limit.

The scope of this thesis is to apply to hierarchical reinforcement learning the approach of control theory to hierarchical control schemes, with small differences that are necessary in the context of learning. We have implemented a novel framework exploiting this approach, and thus fill up a gap between control theory and machine learning, based on one side on reinforcement learning and on the other side on optimal control. Exploiting the idea of cascaded blocks and inner/outer loops, our framework aims to make the hierarchy design for reinforcement learning intuitive. In the simplest form, the high level agent (outer loop controller) learns to drive the low level agent (inner loop controller) to the estimated goal point, whereas the low level agent adjusts its parameters to follow the reference given. This basic idea can be evolved further to adapt to more layers or to more complex topologies like switching between agents according to state as in hybrid systems.

When a problem with a continuous state and action space is considered, it may become impractical or non-satisfactory to employ value function based algorithms. On the other hand, policy search algorithms suffer from slow convergence. In Hierarchical Policy Gradient Algorithms [10], a possible solution to this conflict is presented benefiting the power of hierarchical structure to accelerate learning in policy gradient. We designed a simpler method to approach the same problem with a more compact representation exploiting our control theoretic framework.

Corresponding field of reinforcement learning in control theory is optimal control. The similarity between the two fields becomes more evident once we rephrase the problem addressed by RL in terms of control theory. RL is inspired by behaviorist psychology, concerned with how agents should select actions in an environment, or, in other words, how they should adjust their policy in order to maximize the cumulative (discounted) reward ($\mathcal{J}$). The notion of cumulative reward translates to the negative of the cost function in optimal control ($\mathcal{C} = -\mathcal{J}$). The policy is the control law and the agent of RL is indeed the controller. Objective of RL written in these terms gives the goal of optimal control: to find a control law that minimizes the cost function. However, optimal control becomes weak when the model has strong nonlinearities and is analytically intractable or when approximations are not convenient. In contrast, reinforcement learning makes decisions interacting with its environment directly. Thus, the main difference between optimal control and reinforcement learning is the information used to build the model. In the RL approach, we do not attempt to analyze the dynamics of the environment to obtain a model or model parameters, but operate directly on measured data and feedback (rewards) from interaction with the environment. However, it is easy to spot analogies between the RL approach and optimal control. For instance the reward function selected can be the negative of the well known quadratic cost of the LQR, which describes a performance objective and punishes the control effort.

The system to be controlled can have a very complex structure and in some cases building a model can be impractical or not possible analytically. In control theory, an option to face the issue is to adapt the controller while interacting on the environment. An adaptive controller is composed of two subsystems: control unit and the tuning unit. Tuning unit is responsible to tune the parameters of the control unit which applies the control action to the environment, see 1.2. Referring to this structure, the counterpart of

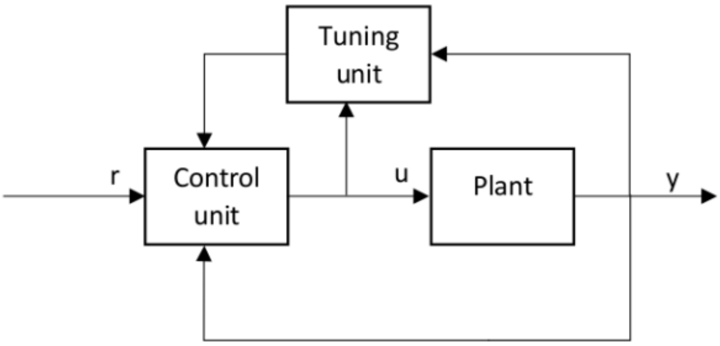

Figure 1.2: Adaptive control with two subsystems

the tuning unit in RL is the learning algorithm and of the control unit is the policy itself. However, even the adaptive control (or, to be more accurate, the controller that has been designed according to adaptive logic) assumes the knowledge of the environment's description to a certain extent, at least in the form of a family of models that best describes it. In addition, adaptive control is mainly applied to the tracking and regularization type of problems rather than to optimal control ones. These form the fundamental contrasts between classical adaptive control and our approach. An exhaustive discussion of viewing reinforcement learning as adaptive optimal control is presented in [20].

This thesis is organized as follows. In the second chapter, the state of the art is presented. Subsections include the titles Hierarchical Reinforcement Learning, Policy Search, Hierarchical Policy Gradient Algorithms. In the third chapter the proposed method is described in detail. Implementation details are introduced in chapter four. In the fifth chapter experimental setups for the proposed method are illustrated and results are discussed. In the conclusion, the scope of the work and the evaluations are summarized and suggestions for future work are given.

Chapter 2

State of the Art

> *"Morty: I just I didn't get a lot of sleep last night. Maybe my dreams were just too loud or something.*
> *Summer: Or maybe you were out all night again with Grandpa Rick.*
> *Jerry: What?*
> *Beth: Dad?*
> *Rick: What, so everyone's supposed to sleep every single night now? You realize that nighttime makes up half of all time?"*
>
> Rick and Morty (Season 1, Episode 1)

Many real-world problems are inherently hierarchically structured, meaning that it is possible look at a complicated challenge as a sequence of its more manageable components. As humans, we tend to learn new tasks by breaking them up into easier ones and ordering them. A very simple real world example can be having a cup of coffee. The subtasks for this goal would be pouring the coffee, adding sugar, adding milk, and stirring.

Artificial intelligence researchers benefited of this structure to afford clarity, concision and implementation hided in large scale tasks, using various ways of abstraction. Macros or macro operators form the fundamentals of the hierarchical notion in AI research. They can be used to group a whole series of actions into one and can be called as one primitive action. In addition, in the definition of a macro another macro can be defined.

2.1 Hierarchy in control systems

In control theory, abstraction plays an important role. One way to implement the abstraction and hierarchy is to use hybrid systems. Hybrid systems are dynamical systems that exhibit both continuous and discrete dynamic

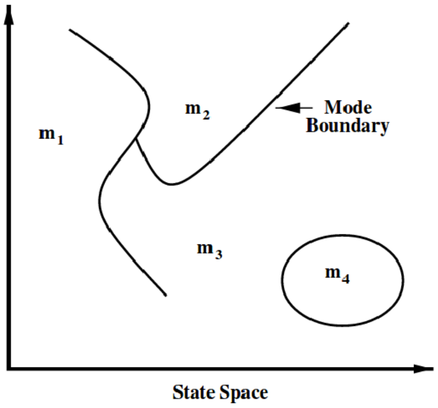

Figure 2.1: Mode switching control

behavior. When a control system is designed exploiting the hybrid formalism, typically, discrete controllers are in the higher levels of the hierarchy. This part is analogous to macros in AI research. In control theory, discrete action with respect to the state information is applied, and this action remains effective for an extended time. That is, discrete action changes the dynamics of the underlying controllers that act with continuous dynamics.

In AI terms a macro is called according to the state and it stays active until it terminates. Hybrid systems are a compromise between centralized and decentralized control schemes. In the centralized control scheme, the control signal is computed with the information from the entire system and then it is distributed to all the actuators. Despite its obvious advantages, such as having the possibility of global optimization, it is not practical as it requires high computational resources and the design tends to get tedious in large scale problems. In a completely decentralized control scheme, local controllers undertake the computation of commands for the subsystem actuators having limited information from the plant. Unlike the prior, they are easy to design. However, in cases of strong couplings between subsystems and/or where the closed-loop performance is crucial, this scheme is not adequate. When a decentralized solution cannot be accepted and a completely centralized solution is too expensive, a compromised scheme with some form of multilevel hierarchy is convenient. In the resulting systems, lower level deals with the local aspects of the performance whereas the higher level is responsible for the global criteria. In such hierarchy, the two levels possibly need different models of the plant. Local controllers need more detailed information on the dynamics of the part they act on and often they work with continuous models while for the higher level such details are impractical to handle. Thus, the typical choices for the higher level are discrete models.

Mode switching controllers are a class of hybrid systems, where the control law depends on the location in state space, as in Figure 2.1. In other words, the higher level controller in the hierarchy switches between low level controllers according to the observation of the state. Mode switching controllers are commonly used in many control applications in order to allow relatively simple controllers to be used in different operating regimes, such as aircraft climbing steeply vs. cruising at constant elevation. In [11], the agent's policy is designed as a deterministic mode switching controller and a new type of reinforcement learning called Boundary Localized Reinforcement Learning (BLRL) is proposed.

2.2 Hierarchy in Reinforcement Learning

Boundary Localized Reinforcement Learning (BLRL) is one of the many ways of defining a policy for a subset of a state in reinforcement learning. Examples of other forms of partial policies can be given as temporally-extended actions, options [21], skills [22] and behaviors [5, 12]. The idea of partial policies with well defined termination conditions is one of the key methods of temporal abstraction introduced to reinforcement learning to improve the scalability and enhance the performance.

The nature of Markov decision processes (MDP), the models on which RL is based, does not involve actions that persist for an amount of time or, simply, temporally extended actions. In the classical MDP framework, an agent perceives the state of its environment at some discrete time t, takes an action accordingly, and, at the next time step t+1, gets from the environment a reward and the information about the next state s'. Moreover, the environment dynamics can be expressed with a transition function, which returns the probability of passing to state s' given the current state s and the action a. Extension of the idea of higher level actions that contain lower level actions, which may or may not be the conventional one step actions, results in another type of decision process: semi-Markov decision process (SMDP). For SMDPs, the waiting time in state s upon the execution of action a is a variable and it represents relevant information. It corresponds to the execution time of a high-level action and depends on the policies and the termination conditions of all the lower-level ones. An SMDP consists of a set of states, a set of actions, an expected cumulative discounted reward for each state-action pair and a well-defined joint distribution of the next state and transit time. All three methods for introducing temporal extensions to actions that will be presented in this section, to explore the interaction

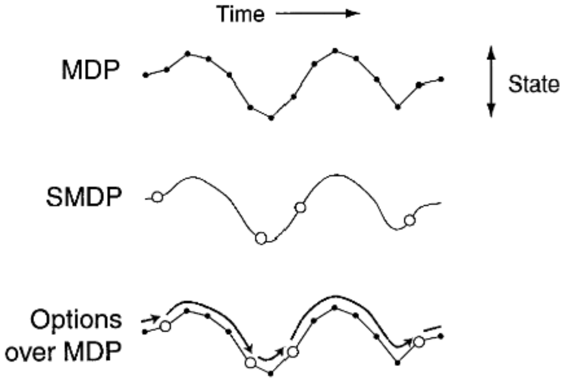

Figure 2.2: Options over MDPs

between MDPs and SMDPs.

2.2.1 Options

An option is a temporally extended action with well defined policy. The term "option" is a generalization of primitive actions to adapt to the concept of temporal extension in SMDPs. In other words, an MDP which has options instead of primitive actions is an SMDP. Figure 2.2 from [21] illustrates this definition. As mentioned above, the state trajectory of an MDP consists of small, discrete-time transitions, whereas that of an SMDP covers larger, continuous time transitions. Options enable an MDP trajectory to be analyzed in either way. The main reason for the construction of options is to express temporally-extended actions in reinforcement learning and thus potentially increase the speed of learning in large-scale problems, particularly at the initial steps. Options can also assist the transfer of learning information between different tasks. For instance the option grab-the-door-knob whose policy is learned in open-the-door task might as well be used in close-the-door task. Though this advantage is highly dependent on the design of the options. This framework also facilitates adding prior knowledge to design. The designer of an RL system typically uses prior knowledge about the task to add a specific set of options to the set of available primitive actions. An option has 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 $I \in \mathcal{S}$. An option is available only in the states that are contained in its initiation set. An option can be selected according to the policy over the options. If an option is selected, the policy of that option is followed until the option is terminated. The probability of an option to terminate in state s is given by $\beta(s)$. If the option does terminate, the agent gets to choose either a primitive action or another option that is available at the termination state of the previous one. Indeed, a primitive action can be considered as an option, called one-step option with $I = \{s : a \in \mathcal{A}_s\}$ and $\beta(s) = 1$ for all s. Each option can select another option, and so on, until one-step options are selected that correspond to the actions of the core MDP which leads to a hierarchical structure

2.2.2 Hierarchy of Abstract Machines

A machine is a partial policy represented by a Finite State Automaton. A Hierarchy of Abstract Machines (HAM) [15] is formed by the start of the execution of an initial machine and the termination of all the machines reachable from the initial one. Machines for HAMs are defined like programs that execute based on their own states, which can be of 4 types: *action, call, choice* and *stop.* In the *action* states an action executes in the MDP. *Call* sates, instead, start the execution of another machine. In *choice* states, the next state of the machine is selected stochastically. *Stop* halts the machine and returns the control to the previous call state. Apart from the set of states, a transition function and a start function which determines the initial state of the machine are needed in order to specify a machine. The transition function of a machine is defined as a stochastic function that depends on the current machine state and some features of the environment state and it determines the next state of the machine after an *action* or a *call* state.

The interaction of a HAM $\mathcal{H}$ with an MDP $\mathcal{M}$ yields an induced MDP $\mathcal{H} \circ \mathcal{M}$. $\mathcal{H} \circ \mathcal{M}$ is indeed an MDP itself, with state set $\mathcal{S} = \mathcal{S}_M \times \mathcal{S}_H$, where $\mathcal{S}_M$ is the set of states of $\mathcal{M}$ and $\mathcal{S}_H$ is the set of states of $\mathcal{H}$. It is important to note that when we try to determine the optimal policy of $\mathcal{H} \circ \mathcal{M}$, the only relevant points are the *choice* points. This is because, when $\mathcal{H}$ is in a state that is not a *choice* state, only one action is allowed. These states can be eliminated to produce an SMDP called *reduce*$(\mathcal{H} \circ \mathcal{M})$. The optimal policy for $\mathcal{H} \circ \mathcal{M}$ will be the same as that of this SMDP, which, in turn, may increase the policy iteration speed. The above-mentioned property is particularly useful for RL. HAM constraints can reduce significantly the exploration effort by reducing the state-space in which the agent operates.

Options expand the repertoire of actions by defining the conventional primitive actions as a special form of options called one-step options, whereas HAMs are designed to constrain the set of allowed actions. For instance, in a case where an agent can choose up, down, right and left as actions, a machine may dictate “repeatedly choose right or down”. Such a command would erase the policies that choose going up or left, which in return would simplify the MDP.

2.2.3 MAX-Q Value Function Decomposition

Another approach to HRL is the MAX-Q value function decomposition, developed in [9]. MAX-Q decomposes the value function of the main MDP into an additive combination of the value functions of the sub-MDPs.

To do so, MAX-Q takes an MDP $\mathcal{M}$ and decomposes it into a finite set of subtasks $\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n$. Here $\mathcal{M}_0$ refers to the root subtask. Once the root subtask is achieved the task itself is achieved. A subtask is formalized by a tuple $\langle \mathcal{T}_i, \mathcal{A}_i, \tilde{\mathcal{R}}^i \rangle$ where $\mathcal{A}_i$ is a set of admissible actions to achieve the subtask,$\mathcal{T}_i$ is a termination predicate that partitions the MDP states $\mathcal{S}$ into a set of active states $\mathcal{S}_i$ and a set of terminal states $\mathcal{A}_i$. A subtask can execute only in the states $s \in \mathcal{S}_i$ and once $\mathcal{M}$ goes into one of the states in $\mathcal{T}_i$, the subtask terminates. $\tilde{\mathcal{R}}^i$ is a deterministic pseudo-reward for each transition to a terminal state. By definition, $\mathcal{R}^{*i}$ is zero anywhere outside the terminal states. If the terminal state is the goal state, typically the pseudo-reward is zero again and for other terminal states it is negative. Each primitive action is a special case of subtask, as it was with options. $\mathcal{T}_i$ of a primitive action is that it terminates directly after the execution, $\tilde{\mathcal{R}}^i$ is uniformly zero and $\mathcal{A}_i$ contains a. For each subtask $\mathcal{M}_i$, a policy π_i is defined. The hierarchical policy for the root task $\mathcal{M}_0$ is $\pi = \{\pi_0, \ldots, \pi_n\}$. MAX-Q executes the hierarchical policy as the ordinary programming languages executes subroutines, using a stack discipline. That is, it explicitly defines a pushdown stack. Let us denote the contents of this stack at time t as $\mathcal{K}_t$. Once a subtask is active, its name and parameters are pushed up in the stack and once it terminates it is aborted from the stack together with everything that was above.

Unlike the other two approaches, in this algorithm, the flow of information is from lower-level subtasks to higher-level subtasks only, and not vice versa. Due to this unidirectional flow of information, the decisions taken at a lower level will be independent of the higher-level goal to be achieved. It is easy to see that due to this additional restriction, the policies produced by this approach may not be hierarchically optimal but *recursively optimal.* Recursive optimality is that the policy at each level is optimal assuming that policies of all lower levels are fixed. In general, a recursively optimal policy may not be hierarchically optimal and vice versa. A hierarchically optimal policy will always be at least as good as a recursively optimal one.

The idea of hierarchical policy is familiar from the discussion about the options. The policy over options, corresponds to that of the subtask in

MAX-Q. In addition to that, the termination condition beta of the options framework translates to the termination predicate in MAX-Q. In this case, β can only take the values 0 or 1 instead of a continuous interval between 0 and 1.

2.3 Policy Search Methods

Reinforcement learning in robotics is challenging, mainly due to the continuous action and state spaces. One technique to address this problem is discretization. However, particularly when considering actions with high dimensionality when a fine control action is needed, this approach becomes impractical. Most value-based reinforcement learning methods need to work on discrete actions, as they estimate a value for state-action pairs and use the estimated values to build the policy: to produce a policy they need to search the action with the best value at each state, that can become impractical, unless the action set is discrete.

An alternative to value-based methods is Policy search (PS). PS methods start with a parameterized policy and try to find the parameters trough evaluating trajectories until a locally optimal solution is found. In robotics tasks, it is relatively simple to describe a family of suitable policies. Moreover, starting with a parameterized policy gives the opportunity to encode prior knowledge. Continuous action spaces are not an issue for PS, as the action is produced by the policy parametrization and no maximization of the action-value function is needed. For this reasons PS methods are frequently used in robotics.

However, unlike value-based approaches, policy search might not converge to the global optima. In addition, they tend to converge slower than the value-based methods.

To take advantage from both PS and value-based approaches, Actor-critic algorithms where developed. Figure 2.3 depicts these algorithms on the gray scale between purely value-based and purely policy search-based methods

There are two ways of approaching PS: model-based and model-free. Model-based policy search addresses this problem by first learning a simulator of the robot's dynamics from data. Subsequently, the simulator generates trajectories that are used for policy learning. It is interesting to note here that the differences between the adaptive control and the model-based policy search are indeed subtle. They both start with a class of models and predict a forward-model. When the interaction with the real robotic environment

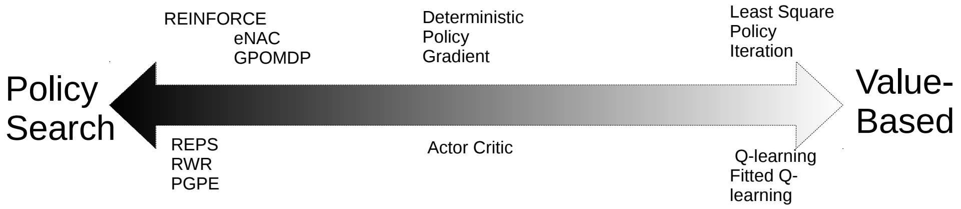

Figure 2.3: algorithms on a gray scale from direct policy search to value-based. Actor critic algorithm are in the middle.

is difficult, due to safety or time constraints, model-based approaches are a preferable solution.

Model-free PS methods update the policy directly based on sampled trajectories and the obtained immediate rewards for the trajectories. Model-free PS methods try to update the parameters such that trajectories with higher rewards become more likely when following the new policy. Since learning a policy is often easier than learning an accurate forward model, the model-free approach is more used.

The policy updates in both model-free and model-based policy search are based on either policy gradients (PG), expectation–maximization (EM) based updates, or information-theoretic insights (Inf.Th.).

2.3.1 Policy gradient

With this method, parameters are updated with a gradient ascent so as to maximize the expected return, $\mathcal{J}_\theta$. The direction of the update is determined by the gradient $\nabla_\theta \mathcal{J}_\theta$. There are several ways to estimate this gradient. The simplest one of them is the finite difference method. The basic idea behind this method is to slightly perturb the parameters, $\delta\theta$, and examine the change in the reward, δR. The second method uses the likelihood-ratio approach: $\Delta p_\theta(y) = p_\theta(y)\Delta \log p_\theta(y)$. Likelihood ratio methods can be step-based or episode-based. Examples of step-based likelihood ratio methods are REINFORCE [23], Gradient Partially Observable MDP (GPOMDP) [4, 3]. In REINFORCE and GPOMDP algorithms the exploration is step-based, i.e., perturbation is applied to the actions at every time-step. Instead, Policy Gradient with Parameter Exploration Algorithm (PGPE) [19], the exploration is done over the parameters, i.e., the perturbation is applied to the parameters at the beginning of the episode. The third method is based on the natural gradient. The key concept behind the natural gradient is to measure the distance between the distributions using the Kullback–Leibler divergence (KL) instead of using the euclidean distance between distribution parameters, to achieve a stable learning behavior. To achieve this goal natural gradient uses Fisher information matrix as metric. In this way, the gradient computed is invariant to the re-parametrization of the same policy, making the natural gradient also invariant to scale changes of the parameters. The computed gradient is rotated less than 90 degrees, therefore the same convergence properties of the vanilla gradient applies. Episodic Natural Actor Critic (eNAC) [18] is one of the algorithms that exploit this idea.

2.3.2 Expectation-Maximization (EM)

The EM-based approach aims at making actions with high future reward more likely. EM formulates the policy search as an inference problem. The trajectories τ are the latent variables in the model and the observed value is the reward event. Then the classical EM algorithm is used to infer the new policy. Unlike the PG methods, since for most of the used policies there is a closed-loop solution, there is no need to determine a learning rate, which is a tricky task as it may lead to slow convergence or unstable policies. On the other hand, EM neglects the influence of the policy update on the trajectory distribution which can cause another problem: large jumps in the policy. It changes the behavior abruptly and there is a possibility that this new behavior may be extremely far from the optimal one, even damage the robot. Policy learning by Weighting Exploration with the Returns (PoWER) [13] and Reward-Weighted Regression (RWR) [17] are among the algorithms using this approach. PoWER uses episode-based exploration strategies with step-based evaluation strategies. PoWER is similar to episodic RWR but it has a more structured exploration strategy.

2.3.3 Information Theoretic Approach (Inf. Th.)

Information theoretic insight is exploited in Relative Entropy Policy Search (REPS) [6, 16]. It combines the advantages of two approaches described above. Instead of following the gradient, REPS formulates the policy update as an optimization problem. It tries to find the policy parameters that maximizes J while keeping bounded the KL w.r.t. the previous parameters.

Using maximum likelihood estimates for the parameter updates is also beneficial for learning multiple solutions, as we can represent them with a mixture model. REPS can be extended to learning multiple solutions by reformulating the problem as a latent variable estimation problem, where the high level options are latent variables. The resulting algorithm is the Hierarchical Relative Entropy Policy Search (HiREPS) [6] algorithm.

2.4 Hierarchical Policy Gradient Algorithm

As discussed above, policy gradient algorithms are convenient for complex robotics tasks with high dimensions and continuous action spaces. However, they tend to suffer from slow convergence due to large variance of their gradient estimators. Hierarchical reinforcement learning methods gained the interest of researchers as a means to speed up learning in large domains. [10] combines the advantages of these two and proposes hierarchical policy gradient algorithms (HPG).

HPG objective is to maximize the weighted reward to go similar to [14] or [3]. As the flat policy gradient algorithm, HPG searches for a policy in a set that is typically smaller than the set of all possible policies as it is parameterized and converges to local optimal policy rather than the global.

In order to perform this approach, similarly to MAX-Q, the overall MDP $\mathcal{M}$ is divided into a finite set of subtasks. That is,$\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n$ where $\mathcal{M}_0$ is the root task. The state space is divided into initiation and termination state sets for these subtasks and each subtask $\mathcal{M}_i$ that is not a primitive action has a policy π_i. The hierarchical policy is executed using a stack discipline. [10] makes the assumption that the root task is episodic and also models the subtasks as an episodic problem. That is, all terminal states transit to an absorbing state s^{*i} with probability 1 and reward 0.

[10] suggests another method to speed up learning in HPG, called hierarchical hybrid algorithms in which some of the subtasks are formalized as value function-based reinforcement learning. These subtasks are in the higher levels where the action spaces are usually finite and the state relevant dimensions are fewer.

Despite the obvious advantages, these algorithms are highly task-specific. That is, they are failing to build up a general framework for hierarchical policy gradient. Indeed the implementation of this algorithm for the ship steering task described in [10] is not trivial. It includes various shifts between different state-action spaces and intricate external reward definitions. Our approach aims to address these issues by implementing a framework that makes the hierarchy design for reinforcement learning more intuitive.

Chapter 3

Proposed Method

> *"Summer: A city of Grandpas?*
> *Morty: It's the Citadel of Ricks. All the different Ricks from all the different realities got together to hide here from the government.*
> *Summer: But if every Rick hates the government, why would they hate Grandpa?*
> *Morty: Because Ricks hate themselves the most. And our Rick is the most himself."*
>
> Rick and Morty (Season 3, Episode 1)

In control theory, hierarchical architectures, where several layers of control interact, are extensively used. A very simple example is the cascaded control structure designed for a DC Motor where the outer loop (high level, slower) ensures that the actual speed is always equal to reference speed. In order to achieve its goal the outer loop outputs a reference torque/current which is received by the inner loop (low level, faster) that controls the torque via armature current and keeps the current in a safe limit.

The design tool of control theory to approach hierarchical schemes is the block diagram. It gives the intuition of layered loops and cascaded structures. Block diagrams are much in use in control theory since they are often more convenient devices to analyze or design a system than the transfer function of the overall system. They are basically a composition of modular subsystems. These subsystems are represented as blocks that include the transfer function of the corresponding subsystem. The interconnections among subsystems are represented with directed lines illustrating the signal flow. We introduce these objects to hierarchical reinforcement learning with small differences that are necessary in the context of learning. The goal is to obtain a generic framework for the design of the hierarchy in reinforcement learning. This framework is a computational graph that contains various types of blocks referring to subsystems. Unlike the block diagram of control theory, in our approach blocks and the interconnections are of various nature.

In our scheme one cycle of the computational graph corresponds to a step in the environment. At each step, environment observation is retrieved in the form of a reward and a state data. A state can be absorbing or not. If it is, then the graph is at the last cycle of the current episode. Otherwise, an action, taken from the last block in the graph, is applied and the environment state is observed once again and so on. Data flow in the graph through connections between blocks. There are three types of connections: input, reward, and alarm connections. Input connections correspond to the connections in the conventional block diagram. They transport input and output data between blocks. We call the input signal of a block as *state* and the output as *action*. It should be remarked that the terms state and action here do not refer to the content of the data carried but to the manner in which the receiving block perceives them. For instance, reward data of the environment can be observed as the state by some blocks. Hence, the connection that carries the reward would be the input connection.

RL agent aims to approximate an optimal behavioral strategy while interacting with its environment. It evaluates its behavioral strategy based on an objective function ($\mathcal{J}$) that is computed with rewards. These rewards can be extrinsic or intrinsic. Extrinsic rewards are in the formalization of the environment. They are retrieved through observation. Intrinsic rewards can be constructed by the designer using the state information. In optimal control, typically, the performance indicator is the cost function ($\mathcal{C} = -\mathcal{J}$) and the optimal control goal is to find a control law that minimizes the cost function. In optimal control, the cost function is optimized at design time. That is, unlike the RL agent, during its operation, the controller does not need a metric to evaluate and update its behavior. This contrast reflects in our framework into another type of connection that carries only the reward information. An explicit reward connection definition is needed. Reward connection implies that the data carried will be used as a performance metric and not as an input to the block.

In hierarchical approaches to RL, typically, high level agents do not take actions at each step. To enforce this structure in our framework, blocks that are in the higher levels of the hierarchy are not activated in some cycles. Their output remains unchanged for those time steps. Once the lower level blocks complete their episode they activate the upper level ones by sending a signal trough alarm connections.

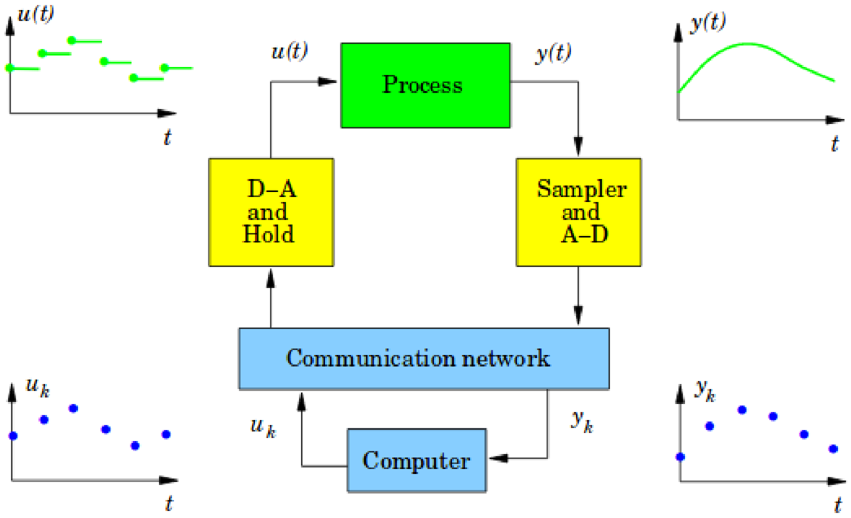

Figure 3.1: Schematic of a digital control system

In this scheme, alarms are analogous to events in event-based control systems. Event-based control systems are digital control methods. Figure 3.1 shows a typical architecture for a digital controller. The process is a continuous-time physical system to be controlled. The input and the output of the process are continuous-time signals. Thus, the output needs to be converted and quantized in time before being sent to the control algorithm. This process is called *sampling*. There are two ways of designing digital controllers: periodic and event-based trigger. The traditional way to design digital control systems is to sample the signals equidistant in time, that is, periodically [2], using a clock signal.

Event based sampling is an alternative to periodic sampling. Signals are sampled only when significant events occur. This way, control signal is applied only when it is required, so as to better cope with various constraints or bottlenecks in the system. A block diagram of a system with event based control is shown in 3.2.

The system consists of the process, an event detector, an observer, and a control signal generator. The event detector generates a signal when an event occurs, typically when a signal passes a level, possibly with different events for up-and down-crossings. The observer updates the estimates when an event occurs and passes information to the control signal generator which generates the input signal to the process. The observer and the control signal generator run open loop between the events. The dashed lines of Figure 3.2

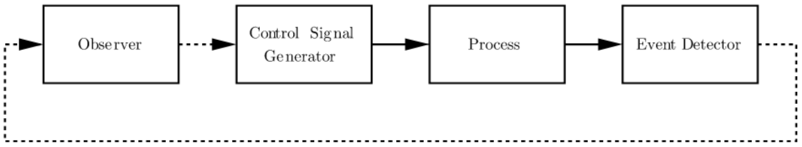

Figure 3.2: Block diagram of a system with event-based control. Solid lines denotes continuous signal transmission and the dashed lines denotes event based signal transmission

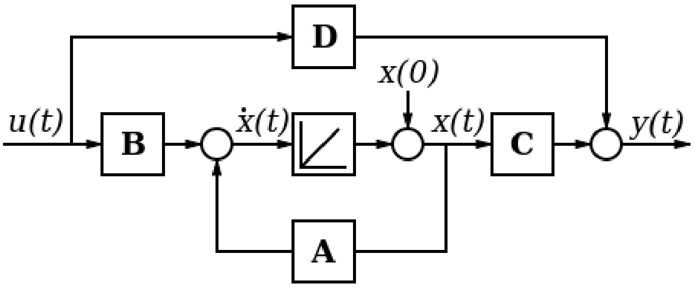

Figure 3.3: Example of a conventional block diagram

correspond to the alarm connections in our framework. In HRL approaches, typically a lower-level algorithm returns to the higher-level one once it has completed its subtask. It can be due to an absorbing state terminating either with success or failure, or simply a horizon reach. The event of event-based control thus corresponds to the end of a subtask of a low-level agent in HRL.

In control theory, each block of the block diagram refers to a subsystem such as a plant, a controller or a sensor, and contains the corresponding transfer function or state-space equation matrices. Either way, it includes an expression of how the incoming signal is manipulated. 3.3 depicts an example structure. Apart from the blocks and connections, there are circle shaped objects in the diagram which represent basic arithmetic operations applied to the incoming signals.

Blocks in our framework have subtle differences. The manner in which the signal is manipulated explicitly depends on the type of the block. The input-output relation of some of the blocks have a discrete-time dynamic behavior. They keep track of the state and store information. That is, they are stateful and their operation can be described with a discrete-time transfer function. *function block* is the most similar kind to a conventional block of control theory. Its operation on the incoming signal is deterministic. In other words, it has a deterministic policy chosen by the designer of the system. Function block is an example to stateless blocks. A type of function block performs simple arithmetic operations on the incoming signals. They are named as *basic operation blocks*.

Control block is the learning part of the system. It is associated to a policy and a learning algorithm. It calls the policy to take an action and uses the learning algorithm to update the policy. This learning algorithm can be policy search-based (e.g. GPOMDP, PoWER) or value-based (e.g. Q-learning, SARSA(λ)) and the policy can be a parameterized one (e.g. Gaussian) or a Q-value-based one (e.g. Boltzmann, ϵ-greedy). Control blocks accumulate each trajectory sample in a dataset. Once the policy is to be updated, this dataset is passed to the learning algorithm. Similarly to the MDPs a sample $\langle s, a, r, s' \rangle$ consist of state, reward, action and next state data, being the input, the output and the reward of the block. In a control block the state and reward data are manipulated separately. State data flow is used for computing the value of the blocks and action, i.e. running the system, whereas the reward data flow is the flow of the evaluation of the performance. Thus, control block needs reward connections to be isolated from the input connections.

Naturally, control blocks are the only blocks at the receiving ends of the reward connections as they are the only learning members. The other blocks may also take extrinsic reward data from the environment. However, they will not perceive it as a performance metric. Thus, they will receive the data through input connections instead of a reward connection.

If the control block is in a lower level of the hierarchy, it can signal the end of its episode to the higher level blocks. The end of an episode can be reached due to an absorbing state with failure or success or a horizon reach. The control block that is in the higher levels of the hierarchy does not take an action until it receives an alarm signal or an indicator of a last step of the environment.

Apart from the end of an episode of a control block, custom events can be created to produce alarm signals. Event blocks are used with this objective. They have custom methods with binary results.

Since some of the control blocks are *awake* only at certain times, some measures needs to be taken to protect the integrity of their dataset. Reward data coming from the environment must not be lost when the high level control is inactive. Reward accumulator block accumulates the input reward until the alarm is up. With the alarm, accumulated data is passed to the controller and the accumulator is emptied. Obviously, the alarm source that is connected to a high level controller needs to be the same as the one that is connected to the reward accumulator that passes its data to that controller. Since the reward accumulator stores information, it is an example of a stateful block.

Another example of a stateful block is the error accumulator. Error accumulator block accumulates the incoming error data and passes it at each step to a control block. When one of its alarms activates or the environment state is the last of the episode it resets the accumulator to zero. Integration in discrete time corresponds to accumulation. The control block that has an error accumulator as an input connection applies integral action.

Placeholders operate as the representation layer of the environment. In the conventional block diagram, the model of the plant is a part of the block diagram. Typically, it receives control signal as an input and outputs the state data to the rest of the blocks. It contains the transfer function of the plant model. In our framework, the model itself is not a part of the diagram. This is primarily because we focus on model-free learning, thus we do not have a mathematical expression of how the environment reacts to inputs, nor its estimation. The only information we have comes from the step samples. The other reason is that blocks may need to observe different parts of the environment. For instance, a block may require only the reward to operate whereas the other one may need both the state and the reward. Such isolation of observation data is achieved with placeholders. At each cycle of the graph, state and the reward output of the environment are passed through the related placeholders, the state and reward placeholders. In addition, last action applied to the environment is also saved in a placeholder called last action placeholder. These blocks do not operate on or manipulate the data they receive. Instead, they function as a bridge or a network between the environment and the diagram.

Selector blocks implement the mode switching behavior in our framework. They contain block lists and activate one list at each cycle. The selection is done according to the first input of the selector block. Once a list inside the selector block is activated all the other inputs to the selector block are passed to the first block of the selected list. The reward and alarm connections of the blocks inside the selector block are independent from the selector. That is, selector block does not have reward connections and its alarm connections are not passed to the blocks in the block lists. Selector block strengthens the flexibility of our framework for hierarchical schemes by allowing hybrid schemes to join the picture. Higher level controller applies an action depending on the state and this action activates an underlying behavior provided by the selected block list in the selector block.

Chapter 4

System Architecture

> *"Morty: I mean, why would a Pop-Tart want to live inside a toaster, Rick? I mean, that would be like the scariest place for them to live. You know what I mean?*
> *Rick: You're missing the point Morty. Why would he drive a smaller toaster with wheels? I mean, does your car look like a smaller version of your house? No. "*
>
> Rick and Morty (Season 1, Episode 4)

The framework implements three layers: hierarchical core, computational graph, and the blocks. The implementation details of these layers and of other utility functions are presented in this chapter. Figure 4.1 illustrates the architecture of the system.

The framework exploits `mushroom` [7], a reinforcement learning library of AI&R Lab of Politecnico di Milano, which provides learning algorithms, features, environments, policies, distributions and function approximators. Apart from the methods implemented, our framework adopts some of the design concepts of the `mushroom` library. Datasets are formed with the same structure, that is, every step is represented with a tuple containing state, action, reward, next state, and the absorbing and last step flags. In addition, the hierarchical core is indeed a hierarchical version of mushroom core. Our framework is implemented to be compatible with any flat reinforcement learning mushroom algorithm.

4.1 Hierarchical Core

Hierarchical core unit is developed as an interface between the diagram and Python. A computational graph is passed to the hierarchical core in the declaration. The hierarchical core contains the methods to run it with a requested amount of steps or episodes. In the beginning of each run the hierarchical core initiates the computational graph and at the end of the runs calls it to stop. The runs can be made for the hierarchical algorithm to learn or to evaluate its performance. During the learning runs, policies of blocks with a learning algorithm in the computational graph are updated. A flag is passed to the computational graph to signal the mode of the run.

4.2 Computational Graph

Computational graph establishes the connection between the environment and the block diagram. In the declaration, it receives a list of blocks and a model.

When the computational graph calls a block, it sends a request to that block to activate its functionality. To call a block, computational graph checks the input, reward and alarm connections of that block. The last outputs of the input connection blocks are passed in an input list. A block can have one reward connection at most. The last output of the reward connection block, if exists, is passed as a reward. The alarm output of the alarm connections are passed as alarms. If a block does not have an alarm connection, it should wake up at each state. Thus, a default alarm is passed in such cases. Finally, in order to indicate the blocks whether or not they should update their policy, a learn flag is passed.

Computational graph calls the blocks in the sorted list one by one until the cycle completes. Each cycle corresponds to a step of the environment. At the start of each episode, the computational graph takes the initial state of the environment, and all the blocks are called for a reset. The first primitive control action, that is the last output of the last block in the sorted block list, is stored in the last action placeholder. Computational graph starts each cycle by applying the primitive control action to the environment. Then, the computational graph observes the response of the environment in the form of a state and reward data. The state and reward data are passed to the corresponding placeholders. The state of the environment can be absorbing due to success or failure which results to terminate the episode. Another reason for a step to be the last of the current episode is that the environment horizon is reached. Two variables that signal the last and the absorbing states of the environment are passed to the blocks by the computational diagram when they are called.

In order to activate the blocks in the correct order, computational graph needs them to be sorted in accordance with their connections, using a topo-

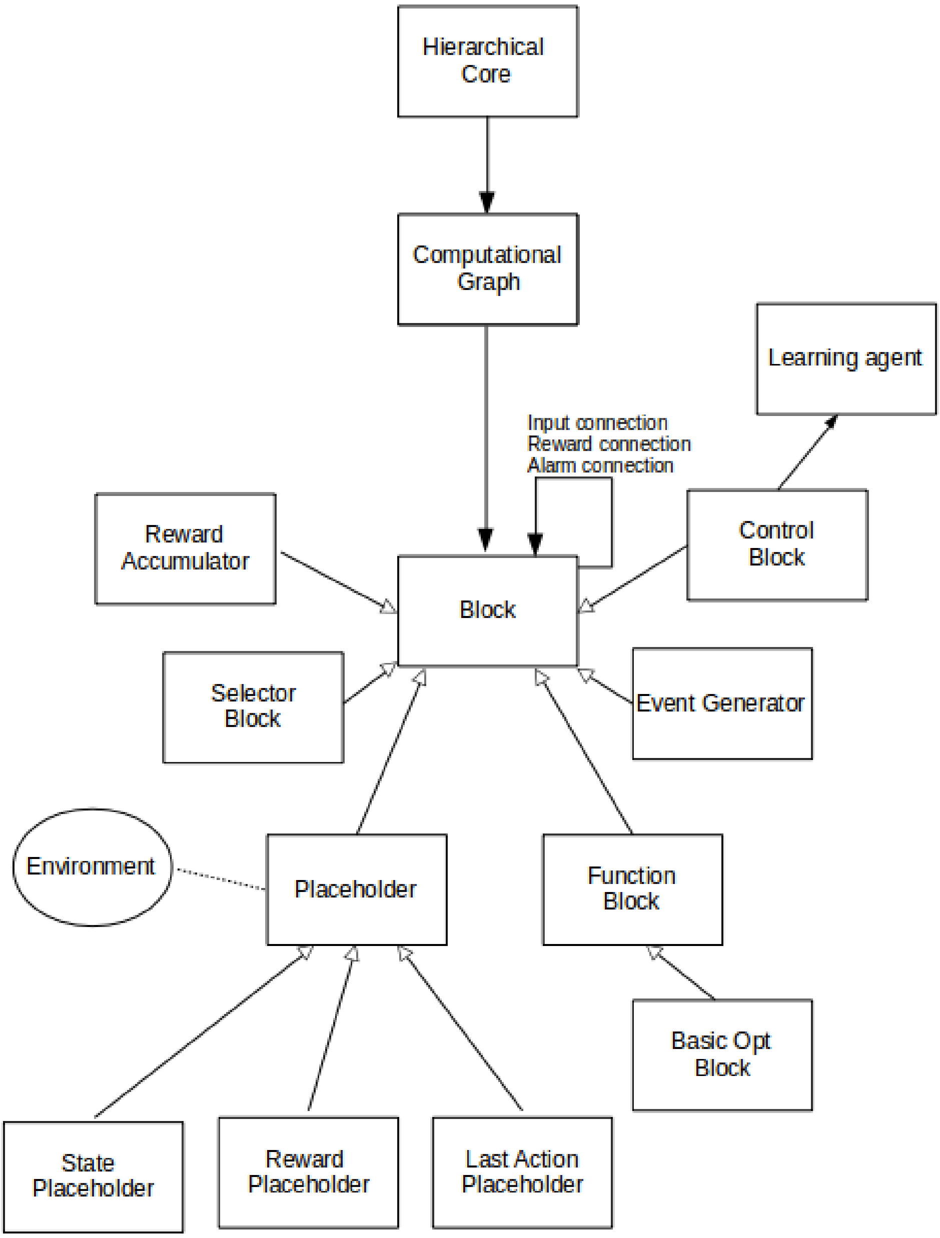

Figure 4.1: System architecture

logical sort algorithm. To achieve this, the block diagram is described as a directed graph where each node is a block and the edges are the connections between them. Placeholders are not considered during the construction of the graph. This is because, as described previously in chapter 3, placeholders are not called as the other block objects. They are employed to pass the environment data to the rest of the graph. Thus, regardless of their connections, they ought to be the first ones in the list. In fact, placeholders do not need to have a connection as their input connection is the environment itself and the environment is not a part of the block diagram. Therefore, they are not declared as nodes and thus no edges are created between any other blocks and placeholders. Additionally, the blocks that are members of the block lists inside the selector blocks are not included in the block list passed to the computational graph. These blocks retrieve their inputs through the selector block. However, their reward connections are independent of the selector block. To maintain the integrity of the topological sort, while constructing the graph for sorting, connections of these blocks are represented as the connections to the selector block. This graph is sorted with a function already implemented in the `networkx` library. The sorted block list that the computational graph uses is a concatenated list of placeholders and the blocks sorted with the above-mentioned algorithm. Computational graph calls the blocks also to initialize and to stop at the end of runs, when it is asked by the hierarchical core.

4.3 Blocks

In our framework, blocks are of various types reflecting the subsystem that they represent. Blocks superclass has a name, a last output, an alarm output, and lists for various types of connections that contain the connected blocks. Blocks have methods to add input, reward, or alarm connections to their lists. Computational graph calls the blocks. Blocks may or may not accept the call. A call is accepted only when the alarms of the called block are up or when it is the end of an episode for an environment. If the call is accepted, block updates its last output and its alarm output with a method implementation that depends on the type of the block. Blocks are asked to initialize by the computational graph at the beginning of each run. At the end of each episode of the environment they are reset.

4.3.1 Function Block

A function block is declared with a name and a custom function. When the call from the computational graph is accepted, that is, at least one of the alarm connections returns true or the environment is at its last step, the last output of the function block is updated. This update is done passing the inputs of the function block as a parameter to the custom function. Reset method of the function block operates similarly, except in this case no conditions are defined to update the last output. When initialized, all outputs are set to `None`. Summation, sign, and squared norm operations are defined and named respectively as: `addBlock`, `signBlock`, and `squarednormBlock`.

4.3.2 Event Generator

Event generator block implementation is similar to that of the function block. In the declaration, a custom function is passed. This function needs to have a boolean return since the event generator triggers the alarms which can be either up or down. Once the event generator accepts the call of the computational graph, it updates its alarm output with the custom function. When it is called to reset, it updates its alarm output regardless of the state of the connections and the environment. Its last output is always `None`.

4.3.3 Control Block

Control block runs the policy/subtask for a period of time that is in general independent from that of the environment. This is because the episode of the subtask is different from that of the task. That is, subtasks have their own termination conditions and horizons.

Control block requires the designer to pass a learning algorithm, number of episodes or number of steps per fit, optionally a termination condition and callbacks during the declaration. It has its own dataset and the dataset is passed to the learning algorithm when the number of episodes or the number of steps per fit is reached and when the hierarchical core is executing a learning run.

Each sample of the dataset is a tuple that contains the state, action, reward, next state, and the absorbing and last step flags. Every time the call of the computational graph is accepted, control block takes a step. If it is the first step of its own episode, it resets. The class method reset calls the learning algorithm to start an episode and draw an action. The current inputs and the action drawn are saved as the first samples of the state and action respectively and the last output of the block is set to the action. In the later steps, the horizon and the termination condition is checked to see whether the current state is absorbing or the last of its episode for the control block. A sample is taken that contains the state, reward and the two flags that indicate the absorbing and last states of the block. Combined with the previous sample, it builds up a step tuple for the control block dataset.

State and action data of the previous sample are the state and action of the step tuple and the state of the current sample is the next state.

Once the step tuple is appended to the dataset, control block checks for the fit condition. If the fit condition is satisfied, i.e., if the number of episodes or the number of steps per fit is reached and the hierarchical core is executing a learning run, it passes the dataset to the learning algorithm to fit, clearing the dataset. The callbacks passed in the declaration will be called during the fit, with the dataset passed.

If the state is not the last of its episode for the environment, an action is drawn and saved as the last output of the control block. An action must be drawn even when the control block reached its horizon or when its termination condition is satisfied. This is because in the framework, each cycle of the computational graph corresponds to a step in the environment and it is not possible to go back to the higher level controller during the cycle. Another cycle is needed to reset and start a new episode for the low level controller that has its episode finished. This makes our framework operate in a way slightly different from the other HRL approaches, such as MAXQ, that use a stack discipline.

The action taken is sampled for the next step tuple. Control block needs to signal the end of its episode to the higher level blocks in the hierarchy. Hence the alarm output is set to the value of the flag that indicates the last step of an episode.

When the computational graph asks for a control block to initialize, control block empties its dataset and sets its last output to `None` and the alarm output to `False`. When the learning or the evaluation run is over, control block calls the learning algorithm to stop.

4.3.4 Placeholders

Placeholders are the first blocks to be called in the cycle and serve as a presentation layer of the environment model. Placeholders have *null* operation in their call, reset and initialize methods. Their last output is assigned by the computational graph using the samples from the environment and the last output of the last block in the ordered list.

4.3.5 Selector Block

The purpose of the selector block is to activate just a chain of blocks avoiding the other blocks to take actions and fit when they are not active. This way we can avert the accumulation of the dataset for the inactive blocks. To implement this behavior, selector block contains block lists ad activates just one list of blocks at each cycle. The selection is done according to the first input of the selector block. Once a list inside the selector block is selected, the rest of the inputs of the selector block are passed to the first block of the selected list. The operation of a selector block is similar to that of the computational graph from this perspective. It calls the blocks in that list one by one passing the inputs, rewards and alarms according to the connection lists of the corresponding blocks. However, selector block does not sort the blocks. Instead, it calls them in the provided order in the list. Selector block manages a boolean list named `first`,s used as an indicator of whether or not the corresponding block list is being called for the first time after a reset or not. This information is necessary for the proper operation of blocks and to preserve the integrity of the dataset of the control blocks in the list. Initially, all flags are set to True. When the computational graph asks for a reset, selector block resets the blocks in the selected block list. It does not request the other block lists to reset as this operation includes taking an action, and selector block must avoid calls to blocks in inactive block lists. Then, the indicator of the selected block lists is set to False. If the selector block is calling the block list before having it reset, i.e. if the first flag for the selected block list is True, selector block resets the blocks in that list instead of calling them. At the beginning of each run, selector block asks all the blocks in the lists to initialize and sets the first flags to True. At the end of runs, all control blocks in the lists are stopped.

Chapter 5

Experimental Results

> *"Rick: Come on, flip the pickle, Morty. You're not gonna regret it. The payoff is huge. (Morty hesitantly picks up the screwdriver and turns the pickle over. The pickle has Rick's face on it) I turned myself into a pickle, Morty! Boom! Big reveal: I'm a pickle. What do you think about that? I turned myself into a pickle! W-what are you just staring at me for, bro. I turned myself into a pickle, Morty!*
> *Morty: And?*
> *Rick: "And"? What more do you want tacked on to this? I turned myself into a pickle, and 9/11 was an inside job?*
> *...*
> *Morty: I-I'm just trying to figure out why you would do this. Why anyone would do this.*
> *Rick: The reason anyone would do this is, if they could, which they can't, would be because they could, which they can't."*
>
> Rick and Morty (Season 3, Episode 3)

5.1 Ship Steering Domain

We chose ship steering task [1] to test the framework. Ship steering has complex low level dynamics due to its continuous state-action space and it is well suited for designing hierarchies. The task was initially studied as a control theory problem [8]. Later, [10] introduced the problem to machine learning literature and suggested Hierarchical Policy Gradient Algorithms as a solution. Ship steering domain is shown in Figure 5.1.

A ship starts at a randomly chosen position, orientation and turning rate and has to be maneuvered at a constant speed through a gate placed at a fixed position in minimum time. Since the ship speed is constant, minimum-time policy is equivalent to shortest-path policy. We consider the ship model given by a set of nonlinear difference equations 5.1.

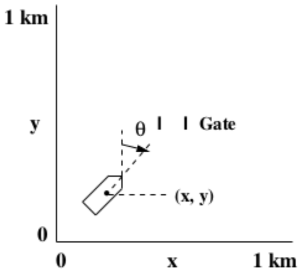

Figure 5.1: Ship steering domain

Variable	Min	Max
x	0	1000
y	0	1000
θ	$-\pi$	π
$\dot{\theta}$	$-\frac{\pi}{12}$	$\frac{\pi}{12}$
r	$-\frac{\pi}{12}$	$\frac{\pi}{12}$

Table 5.1: State and action variable ranges

$$\begin{aligned} x\,[t+1] &= x[t] + \Delta V \sin\theta\,[t] \\ y\,[t+1] &= y[t] + \Delta V \cos\theta\,[t] \\ \theta\,[t+1] &= \theta[t] + \Delta\dot{\theta}\,[t] \\ \dot{\theta}\,[t+1] &= \dot{\theta}\,[t] + \Delta(r\,[t] - \dot{\theta}\,[t])/T \end{aligned} \tag{5.1}$$

where $T = 5$ is the constant time needed to converge to the desired turning rate, $V = 3\frac{m}{s}$ is the constant speed of the ship, and $\Delta = 0.2s$ is the sampling interval. The control interval is $0.6s$ (three times the sampling interval). There is a time lag between changes in the desired turning rate and the actual action, modeling the effects of a real ship's inertia and the resistance of the water. The state variables are the coordinates (x, y), the orientation (θ) and the actual turning rate of the ship ($\dot{\theta}$). The control signal is the desired turning rate of the ship (r). The state variables and the control signal are continuous within their range given in the table 5.1.

The ship steering problem is episodic. In each episode, the goal is to learn the sequence of control actions that steer the center of the ship through the gate in the minimum amount of time. To make sure the optimal policy is to find the shortest path, every step of an episode has a reward of -1. If the ship moves out of bound, the episode terminates and is considered as a

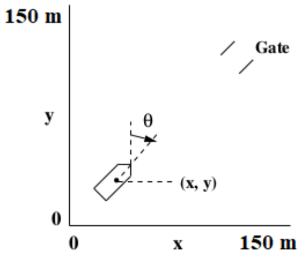

Figure 5.2: Ship steering small domain

Variable	Min	Max
x	0	150
y	0	150
θ	$-\pi$	π
$\dot{\theta}$	$-\frac{\pi}{12}$	$\frac{\pi}{12}$
r	$-\frac{\pi}{12}$	$\frac{\pi}{12}$

Table 5.2: State and action variable ranges for small environment

failure. In this case, environment returns -100 as a reward. The sides of the gate are placed at coordinates (350,400) and (450,400). If the center of the ship passes through the gate, episode ends with a success and a reward 0 is received.

The task is not simple for classical RL algorithms for several reasons [10]. First, since the ship cannot turn faster than $\frac{\pi}{12}\frac{rad}{s}$, all state variables change only by a small amount at each control interval. Thus, we need a high resolution discretization of the state space in order to accurately model state transitions, which requires a large number of parameters for the function approximator and makes the problem intractable. Second, there is a time lag between changes in the desired turning rate r and the actual turning rate, ship's position x, y and orientation θ, which requires the controller to deal with long delays. In addition, the reward is not much informative.

Thus, we used a simplified version of the problem for the early experiments. The simplified ship steering domain is given in Figure 5.2. The domain boundaries are smaller, to shrink the state space. Ranges of the state and action variables are given in Table 5.2. In this case, the gate is positioned in the upper right corner of the map at the coordinates (100,120) and (120,100). The ship starts at the origin of the state space, i.e., fixed at position (0, 0) with orientation 0 and no angular velocity.

5.2 Flat Policy Search Algorithms

Experiments with flat PS algorithms are implemented for the small domain using `mushroom` library. The flat algorithms use a tiling with three dimensions to divide the state space $5 \times 5 \times 6$ tiles each. 100 experiments have been carried out for each flat algorithm. Every experiment lasted 25 epochs. In every epoch, the learning algorithm runs for 200 episodes. Then the evaluation run is carried out. The resulting dataset of the evaluation run is used to compute cumulative discounted rewards ($\mathcal{J}$) for each episode in the evaluation dataset and averaged for each epoch. With the same method, also the mean length of the episodes for each epoch is computed. These data are averaged for 100 experiments.

For the major part of the experiments, PG algorithms are implemented with an adaptive learning rate. Adaptive learning rate constrains the step of the learning algorithm with a given metric, instead of moving of a step proportional to the gradient. The step rule is given in Equation 5.2. Adaptive learning rate can be used to prevent jumps when the gradient magnitude is big.

$$\begin{aligned} \Delta\theta &= \underset{\Delta\vartheta}{\operatorname{argmax}}\, \Delta\vartheta^t \nabla_\theta J \\ s.t.: &\ \Delta\vartheta^T M \Delta\vartheta \leq \varepsilon \end{aligned} \tag{5.2}$$

GPOMDP, a step-based PG algorithm, has been implemented to learn the parameters of a multivariate Gaussian policy with diagonal standard deviation. The initial standard deviation (σ) is set to 3×10^{-2}. The mean of the Gaussian is formulated with a linear function approximator with tiles. Learning parameter is adaptive with a value of 10^{-5}. Policy is fit every 40 episodes during the learning run. Number of episodes of an evaluation run is also 40.

RWR, REPS and PGPE algorithms work on a distribution of policy parameters and they can be used with deterministic policies. They have been implemented to learn the distribution of the mean μ of a deterministic policy. Mean is linearly parameterized with the tiles. The distribution is a Gaussian distribution with diagonal standard deviation. The initial mean μ is the null vector, and 4×10^{-1} variance σ^2 for every dimension. All three algorithms are run for 200 episodes to learn at each epoch. Every 20 episodes, the distribution parameters are updated. The evaluation run lasts 20 episodes. The learning rate of PGPE is adaptive and the value is initially set to 1.5. REPS and RWR do not require the learning rate to be specified as explained in Chapter 2. REPS is used with a max KL step $\epsilon = 1$, and

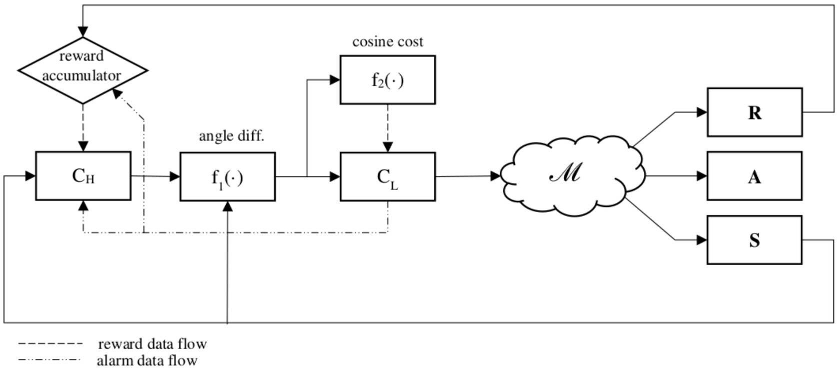

Figure 5.3: HRL block diagram for ship steering task

RWR with a constant $\beta = 0.7$ for the exponential transformation.

5.3 Hierarchical Block Diagram Approach

The ship steering control system consists fundamentally of two parts [8]:

- a ship autopilot which automatically manipulates the rudder to decrease the error between the reference heading angle and the actual heading angle, and
- an operator, either human or a machine, that closes up the control loop given the reference angles to follow a desired trajectory on the ocean.

The control theory approach to ship steering problem is preserved for the subtask definitions with our framework. The corresponding block diagram is given in Figure 5.3.

Low level controller learns to approximate the optimal policy to decrease the error between the reference angle and the actual angle. High level controller learns to drive the low level by producing reference positions in the map that approximate the shortest paths to the gate. A function block in between the control blocks ($f_1(\cdot)$) computes the reference angle for the low level, given the reference position and the states. The performance metric of the high level control is the extrinsic reward, whereas the one of the low level is computed by another function block, $f_2(\cdot)$. $f_2(\cdot)$ returns the cosine of the angle difference computed by $f_1(\cdot)$. As the difference of the reference angle and the actual angle gets close to 0, the reward advances to 1 and as it grows, the reward decreases to -1.

When the low level controller finishes its episode, it raises an alarm for the high level. In the following cycle, a new reference point is estimated for the level. The same alarm signal is passed to the reward accumulator of the high level controller with the motivations described in Chapter 3.

The high level controller uses the GPOMDP algorithm to learn a multivariate Gaussian policy with diagonal standard deviation. Its mean is initialized in the middle of the map, that is $(75, 75)$, for the small domain. The variance of both dimensions is 40. The algorithm fits every 20 episodes with an adaptive learning rate of 10. Horizon is 100 steps for both controllers.

Low level control policy should steer the ship in the direction of the angle error. That is, if the reference angle is larger than the current angle, angular velocity output of the low level control should be positive and vice versa. Low level controller is first tested with a GPOMDP algorithm with a differentiable parameterized policy such as a linear one given as $\pi(x) = \omega x$, where x is the error $\Delta\theta$ over the angle. In this scheme, the parameters of the low level policy are expected to be positive to make sure that the input error and the output angular velocity have the same sign. However, when the high level policy is not reliable, low level policy may choose to go outside of the map to finish the episode quickly. In addition, when the optimal parameter is lower than the current one, a big gradient step may cause trespassing the 0 limit, which results in instability. To ensure stability, the parameter of the policy needs to be forced to have a positive value. This can be achieved with a deterministic policy by using the absolute values of weights, that is, $\pi(x) = |\omega|\, x$. However, this policy is not differentiable. Therefore, classical PG algorithms cannot be exploited. Thus, the low level controller uses a black box optimization method. PGPE algorithm learns the distribution of the weights of a non-differentiable deterministic policy. The distribution over the weights has zero mean and 10^{-3} as variance initially. The algorithm fits at every 10 episode of the low level controller. The action taken by the low level is indeed a proportional control action. The error signal is multiplied by a constant, K_P. The distribution over K_P is learned by the PGPE algorithm. An adaptive learning rate is used with a value $5 \times 10{-}4$.

Another experimental setup is constructed in which the low level control applies also the integral action. To achieve this, an error accumulator block is added between $f_1(\cdot)$ and the low level controller as shown in Figure 5.4.

In this case, the state of the low level control is two dimensional. The adopted deterministic policy is $\pi(x) = \left|\boldsymbol{\omega}^T\right| x$. The state vector is $x =$

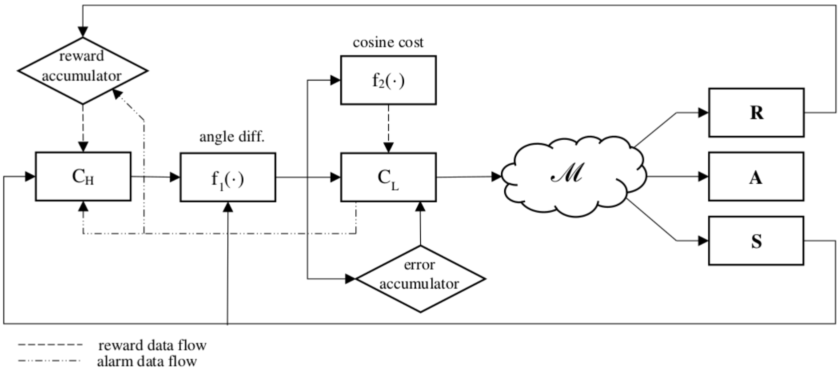

Figure 5.4: HRL block diagram for ship steering task, PI control action

$\left[\Delta\theta, \int \Delta\theta dt\right]$. The parameter vector is $\boldsymbol{\omega} = [K_P, K_I]$, where the K_P parameter corresponds to the proportional gain, and the K_I parameter is the integral gain, as they multiply the error and the integral of the error respectively.

5.4 Results in Small Domain

Results of the flat algorithms and of the hierarchical schemes designed with the block diagram approach are compared.

Figure 5.5 shows the mean of the objective function for every epoch, averaged for 100 experiments for each algorithm. It shows that flat PG algorithms, i.e., PGPE and GPOMDP, converge much more slowly than the other ones. This is due to the fact that for these algorithms learning rate should be defined. If the learning rate is low, the convergence is slow and if it is too high, learning may be unstable.

Figure 5.6 and Figure 5.7 show the mean length of the episodes for each epoch, averaged for 100 experiments for different algorithms. Hierarchical algorithm has a large episode length for the first runs. Since the controller is stable, it learns first to avoid to get out of the boundaries, which causes the long episodes exploring the middle of the map. Once the gate is identified well, the shortest path is approximated, which results in gradually lowered episode lengths.

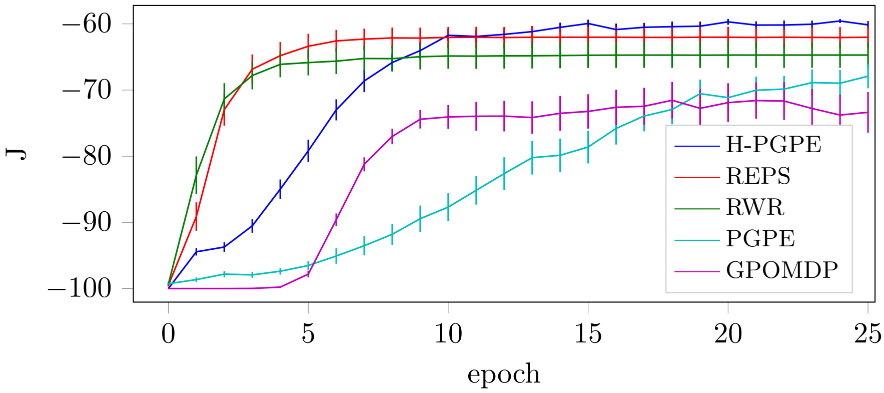

Figure 5.5: Comparison of the hierarchical structure w.r.t. flat policy search for small environment. Mean values of objective functions with 95% confidence intervals are shown

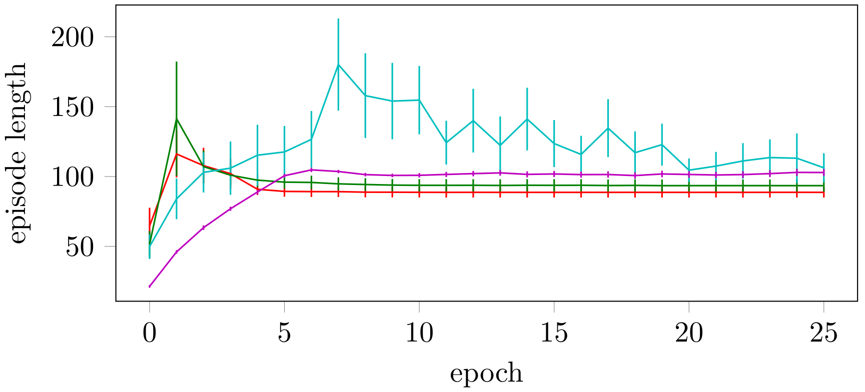

Figure 5.6: Comparison between flat policy search algorithms for small environment. Mean values of episode lengths with 95% confidence intervals are shown

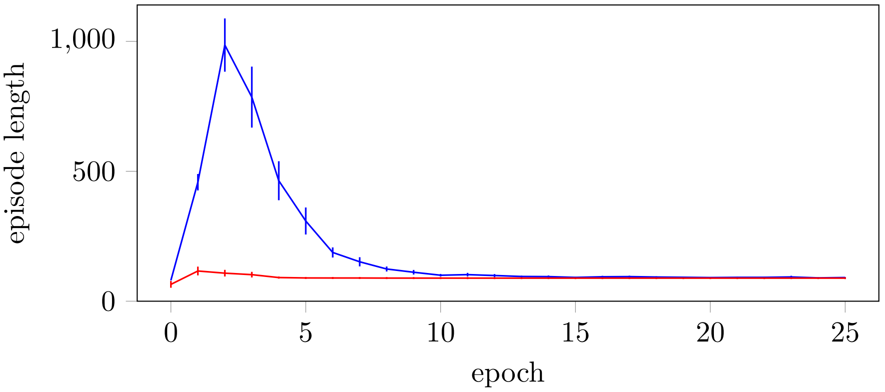

Figure 5.7: Comparison of the hierarchical structure w.r.t. flat policy search for small environment. Mean values of episode lengths with 95% confidence intervals are shown

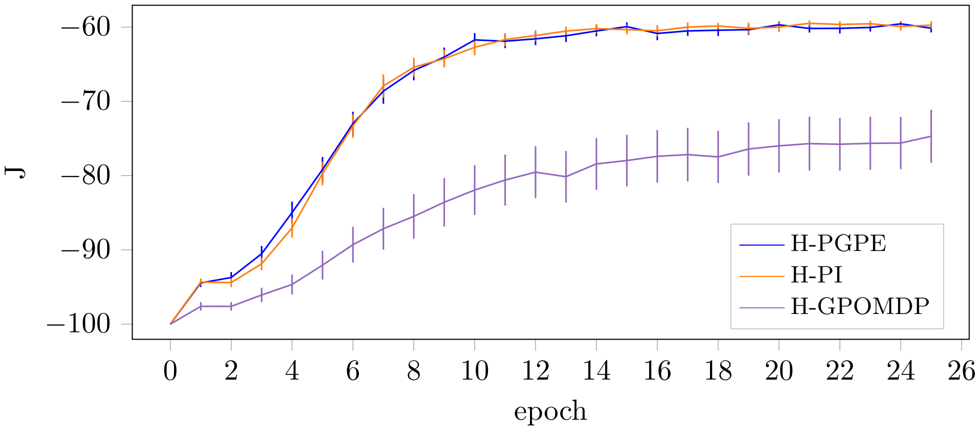

Figure 5.8: Comparison between the hierarchical structures search for small environment. Mean values of objective functions with 95% confidence intervals are shown

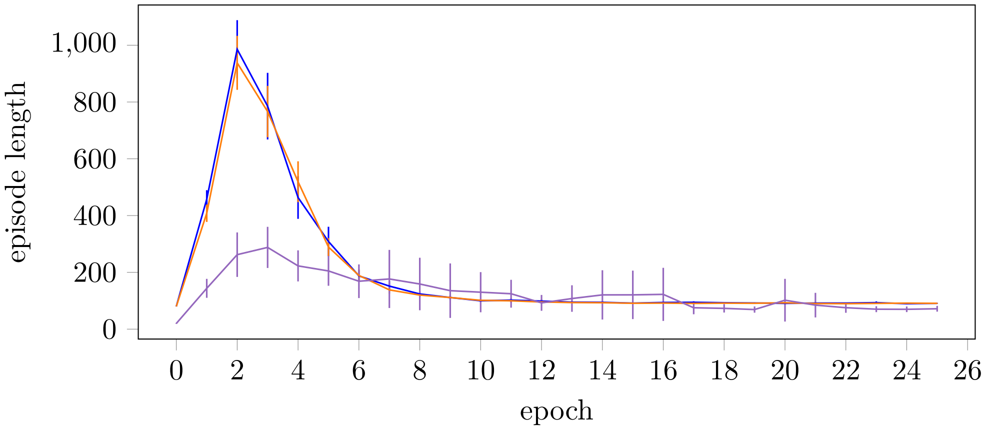

Figure 5.9: Comparison between the hierarchical structures search for small environment. Mean values of episode lengths with 95% confidence intervals are shown

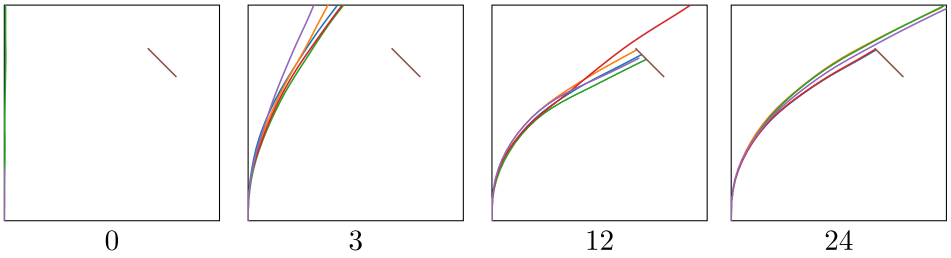

Figure 5.10: flat-GPOMDP, last 5 trajectories of the epochs

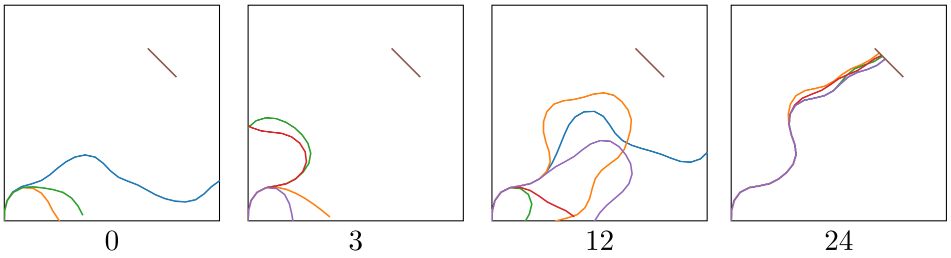

Figure 5.11: flat-PGPE, last 5 trajectories of the epochs

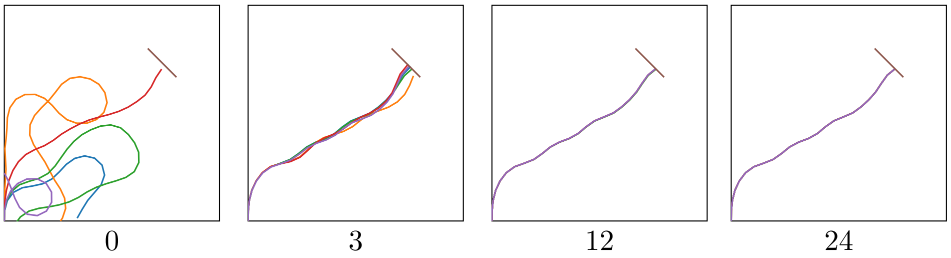

Figure 5.12: flat-RWR, last 5 trajectories of the epochs

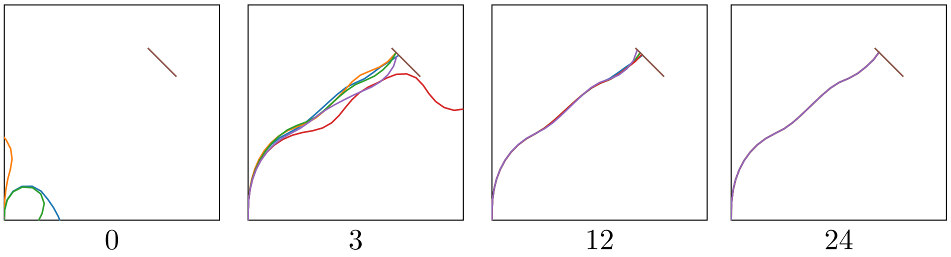

Figure 5.13: flat-REPS, last 5 trajectories of the epochs

The performance of hierarchical algorithms with and without the deterministic policies in the low level are compared in Figures 5.8 and 5.9. H-PGPE and H-PI algorithms use PGPE to learn the distribution of parameters of a deterministic low level policy that is ensured to be stable and H-GPOMDP uses GPOMDP with a stochastic policy. H-PI has the error accumulator block before the low level control block, which results in integral action in the low level policy. Forcing the stabilization of the low level controller causes the learning curves to be much steeper w.r.t. the other method. Adding the integral action does not cause a big change in the learning performance, but in the quality of the trajectories.

Figure 5.10 depicts the trajectories of the last 5 episodes of the epochs 0, 3, 12 and 24 for the flat-GPOMDP. Even in the 25th epoch some trajectories are not able to reach the gate. Convergence is very slow due to the low learning rate and the step-based exploration. If the learning rate is increased, the policy may become unstable. Figure 5.11 shows the same trajectories for the flat-PGPE algorithm. It explores a larger part of the map as it learns the parameters of the distribution over the policy parameters. However, convergence is still slow due to small learning rate. Moreover, the variance of the policy distribution is not reducing sufficiently faster. As can be seen in Figures 5.12 and 5.13, REPS and RWR identify the gate much faster than the PG algorithms. However, they suffer from premature convergence as the variance that shrinks too rapidly. These characteristics may cause the policies to get stuck in a suboptimal policy.

Trajectories of the hierarchical algorithm with GPOMDP algorithm in the low level (H-GPOMDP) are given in Figure 5.14. Convergence is faster than that of the flat PG algorithms. However, this algorithm sometimes fails to learn. This is because in some runs, the low level policy learns to leave the map and becomes unstable. Learning speed is improved when black box optimization methods are used in the low level as seen in Figure 5.15 since the learning rate can be increased while ensuring stability with a deterministic policy. Hierarchical algorithms produce policies that are closer to the optimal one with much less parameters than the flat ones. When the lower policy applies also the integral action, angle reference is followed more effectively, see Figure 5.16.

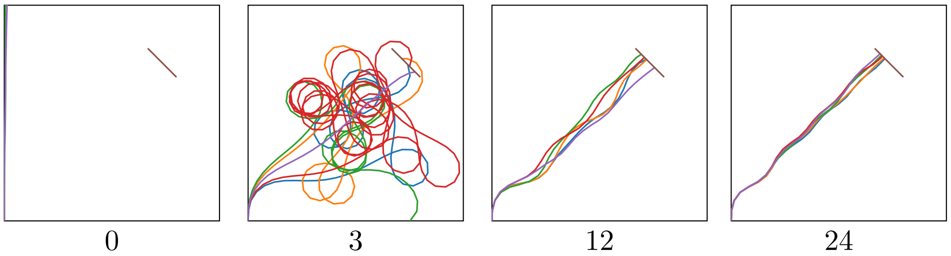

Figure 5.14: H- GPOMDP, last 5 trajectories of the epochs

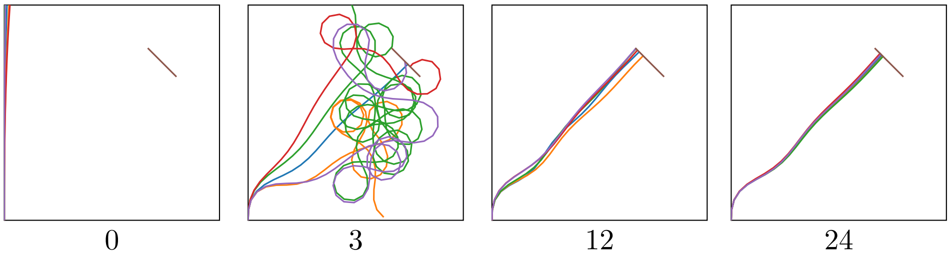

Figure 5.15: H-PGPE, last 5 trajectories of the epochs

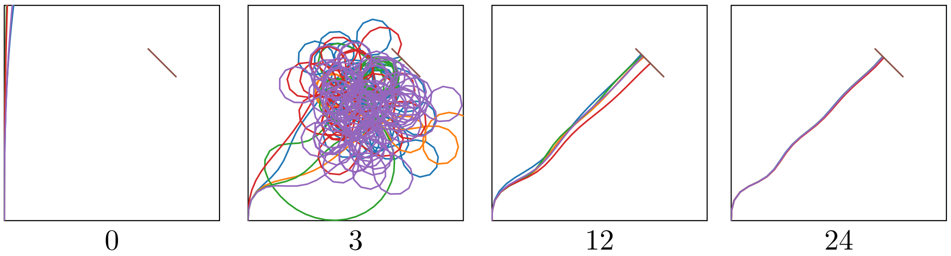

Figure 5.16: H-PI, last 5 trajectories of the epochs

Figure 5.17 shows the average parameters of the Gaussian policy of the high level controller for hierarchical algorithm with GPOMDP used to find the low level policy (H-GPOMDP). On the left, the mean parameter evolution over time is shown. On the right, the ellipses indicate the area where 95% of the samples falls into. Green ellipse is of the initial distribution. Blue ellipse refers to the distribution in the middle of the learning run and the red one is of the final. Both the mean and the elliptic areas are computed averaging 100 experiments. Figure 5.18 demonstrates the same variables for the hierarchical algorithm with PGPE in the lower level policy (H-PGPE).

It can be seen that despite having the same policies and learning algorithms in the higher level, the hierarchical algorithm with GPOMDP in the low level (H-GPOMDP) performs much worse than the other one (H-PGPE). It is unable to identify the gate and the variance is still high at the end of the learning run. This is because, the low level policy is not fast enough in learning to follow the references of the high level. Moreover, the high level does not get a meaningful information from the environment before the low level is able to follow. This issue is evaded using a deterministic policy that is enforced to be stable as in H-PGPE. Another thing to notice is that the high level controller of H-PGPE algorithm approximates the gate position to be slightly further than the actual one. This is due to the fact that when a point just behind the gate is given as a reference, the ship will have to pass the gate to reach that point. Figure 5.19 also depicts the parameter distribution properties of the high level policy. They refer to the hierarchical algorithm with PGPE in the lower level policy with integral action (H-PI). The results are similar to that of the H-PGPE. Comparing the 3D mean parameter graphs of the hierarchical policies, we can see that the algorithms with PGPE in the low level converge towards to the goal with a much smoother trajectory than the algorithm with GPOMDP in the low level.

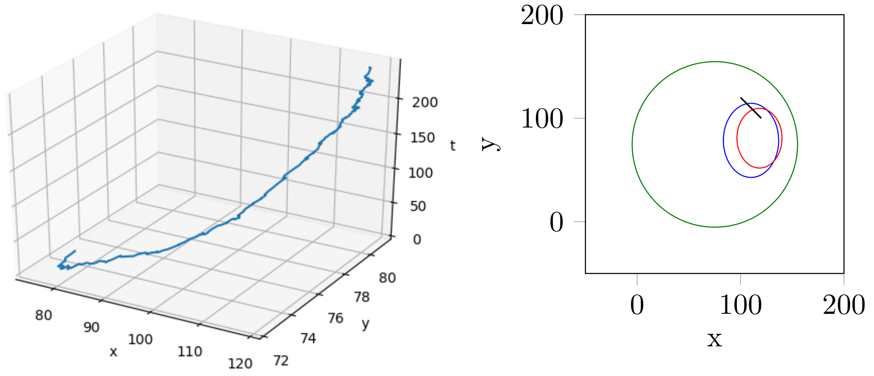

Figure 5.17: H-GPOMDOP. Left: Mean parameters of the high level policy parameters. Right: The areas in which the 95% of the samples of the high level policy parameters fall into

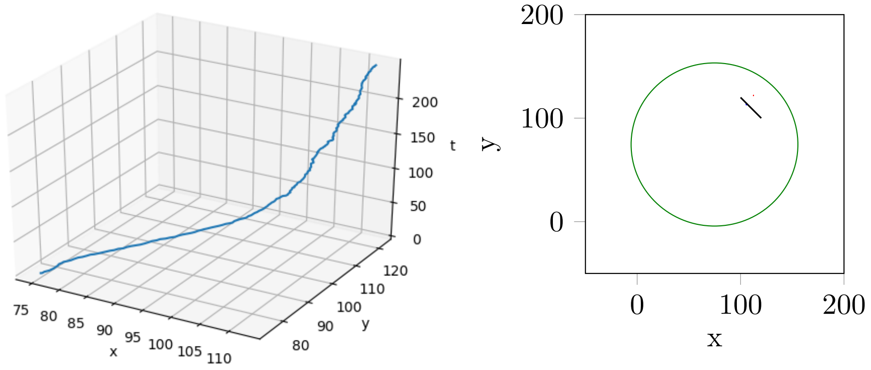

Figure 5.18: H-PGPE. Left: Mean parameters of the high level policy parameters. Right: The areas in which the 95% *of the samples of the high level policy parameters fall into*

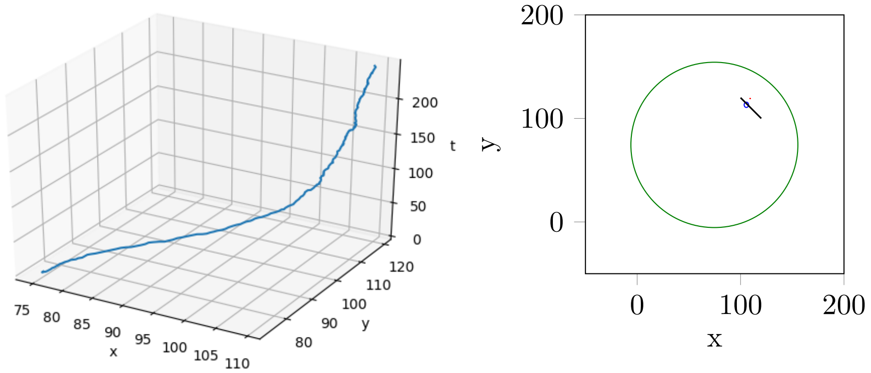

Figure 5.19: H-PI. Left: Mean parameters of the high level policy parameters. Right: The areas in which the 95% *of the samples of the high level policy parameters fall into*

5.5 Results in Regular Domain

The task in the regular ship steering domain possibly is too complex for the flat algorithms due to the larger state-space and random initialization of the ship position. Scaling the discretization from the small to the big environment makes the policy intractable. Therefore, experiments are conducted only with hierarchical algorithms that had a good performance in the small domain: H-PGPE and H-PI.

The high level controllers use the GPOMDP algorithm to learn a multivariate Gaussian policy with diagonal standard deviation. Policy distribution mean is initialized in the middle of the map, that is $(500, 500)$. The variance of the two dimensions is 255. The algorithm fits every 40 episodes with an adaptive learning rate of 50. Horizon is 100 steps for both the low level and the high level algorithms.

A deterministic policy is constructed by using the absolute values of weights, that is, $\pi(x) = |\omega|\, x$ for the low level policies. PGPE algorithm learns the distribution over the weights that is initialized with zero mean and 10^{-3} variance. The algorithm fits every 10 episodes of the low level controller with an adaptive learning rate of $5 \times 10{-}4$. The action taken by the low level is a proportional control action for H-PGPE and proportional-integral for H-PI.

The learning curves of the algorithms are given in Figure 5.20. The values are computed with the mean of the objective function for every epoch, averaged for 100 experiments for each algorithm. Curves are slightly less steep than the ones of the smaller domain. This is because the regular domain task is harder. Figure 5.21 shows the average episode lengths for each epoch, for both algorithms. In the first epochs, low level learns to stay in the map and to follow the referenced angle. Once the high level algorithm detects the gate position, episodes become much shorter, to approximate the shortest path.

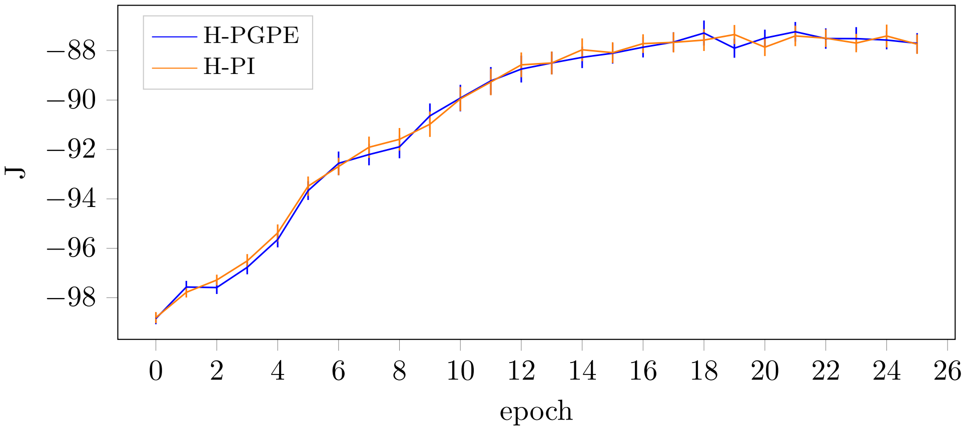

Figure 5.20: Average of objective function of the hierarchical algorithms at every epoch

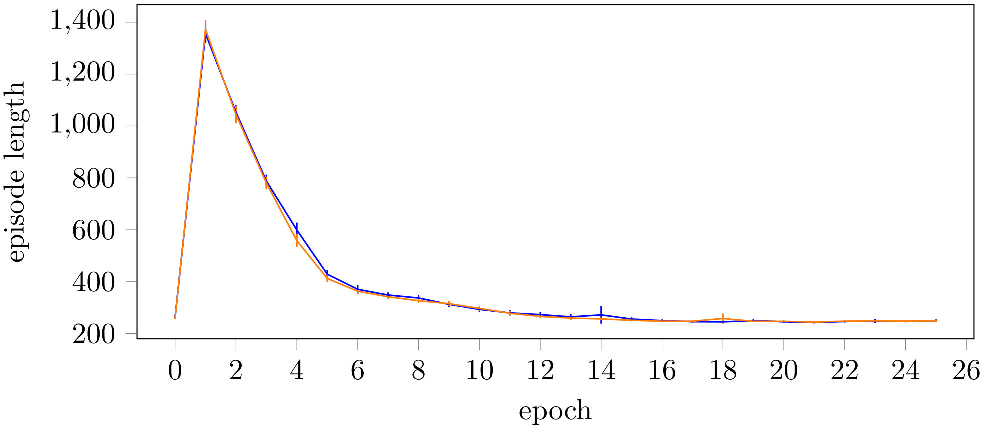

Figure 5.21: Average episode lengths of hierarchical algorithms at every epoch

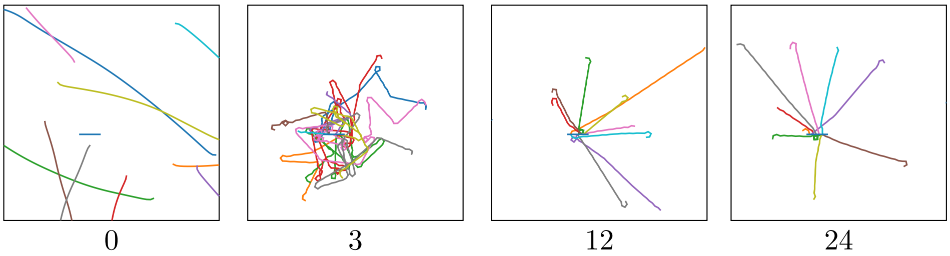

Figure 5.22: hierarchical algorithm with PGPE in the low level, last 5 trajectories of the epochs

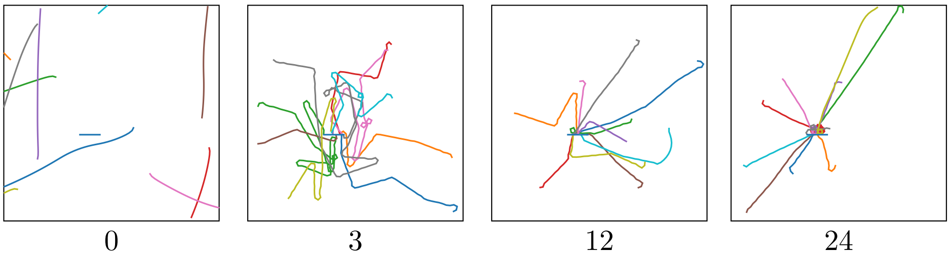

Figure 5.23: hierarchical algorithm with PGPE in the low level, with integral action, last 5 trajectories of the epochs

Figure 5.22 depicts the trajectories of the last 5 episodes of the epochs 0, 3, 12 and 24 for the H-PGPE algorithm and Figure 5.23 shows that of the H-PI algorithm. The trajectories start at random points as the high level task requires. In both figures we observe a stroll around the gate at the last steps. This is because the ship starts at random points and the gate is in the middle of the map. These characteristics of the regular domain eliminate the possibility of placing the reference position behind the gate. Moreover, when the ship is almost reaching the gate, low level policy performance is lousy. We observe a negative peak in the low level intrinsic reward as the angle difference jumps from 0 to π while the ship trespass the reference point.

The average of the parameters of the hierarchical algorithm with PGPE in the low level (H-PGPE) is shown in Figure 5.24. Figure 5.25 demonstrates the same observations for hierarchical algorithm with PGPE learning a policy with integral action in the low level (H-PI). We observe that both algorithms are able to identify the gate position quite fast, almost as fast as they identified the gate in the small domain. The fast convergence is explained by the fact that the position of the gate is near to the middle of the map and the initial distribution of the high level is centered in there.

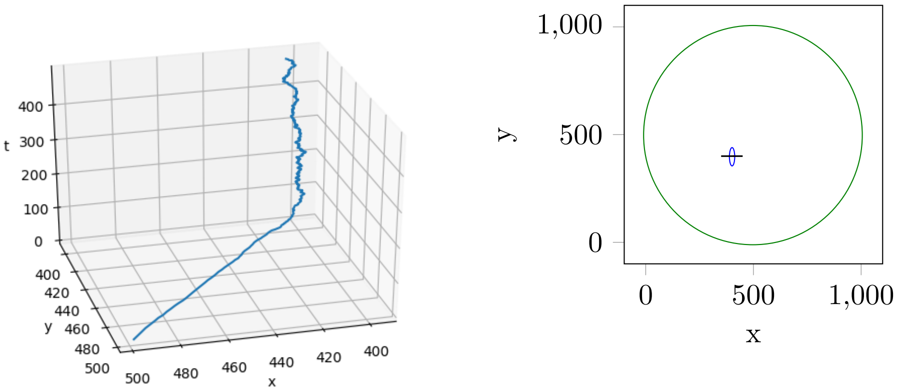

Figure 5.24: H-PGPE in regular domain. Left: Mean parameters of the high level policy parameters. Right: The areas in which the 95% of the samples of the high level policy parameters fall into

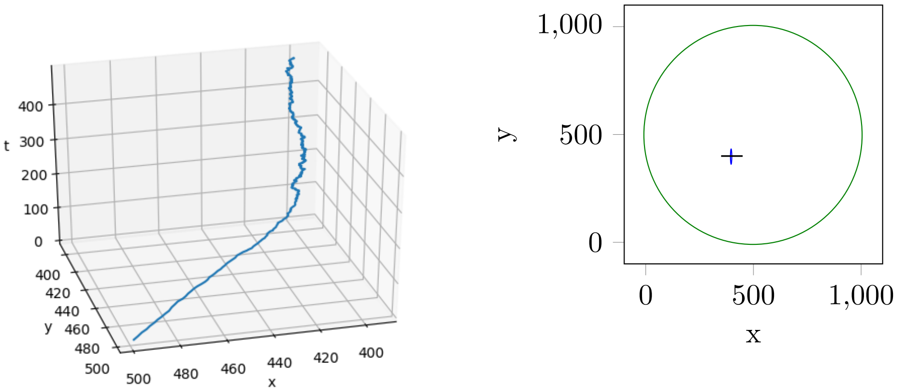

Figure 5.25: H-PI in regular domain. Left: Mean parameters of the high level policy parameters. Right: The areas in which the 95% of the samples of the high level policy parameters fall into

Chapter 6

Conclusions and Future works

> *"Teacher Rick: Tomorrow you will be transferred to your new Ricks. Hopefully they will be your last. Yes, Slow Ri– Tall Morty?*
> *Tall Morty (Slow Rick): Di-Did I gragitate this time yet?*
> *Teacher Rick: Anything's possible, Tall Morty. Ugh. . . "*
>
> Rick and Morty (Season 3, Episode 7)

We developed a novel framework to design hierarchical structure in reinforcement learning that exploits the design tool of control theory: the block diagram. To achieve this, the block diagram is adapted to the context of hierarchical reinforcement learning. Our framework describes a computational graph that contains blocks and connections. Unlike the block diagram of control theory, in our approach blocks and the interconnections are of various nature. The type of connection depends on the use of the signal it carries, whereas the type of the block depends on the subsystem it represents.

By running the computational graph, the hierarchical agent can interact with the environment and update all levels of its hierarchy. Similarly to the block diagram of the control theory, this framework can be applied to implement different types of hierarchical topologies. Therefore, the designer is free to implement his/her own hierarchical structure. Combining the strengths of both fields, we filled up a gap between control theory and machine learning.

Hierarchical approaches simplify the tasks of designing a policy and possibly adding prior knowledge: with our framework, we can exploit the existing hierarchical control structures and initial parameters for the well-known problems of control theory. Adapting a control theoretical perspective, policies are often simpler to interpret and with fewer parameters, without losing representation power. In control theoretic approaches, typically a human operator is needed to close the control loop for high level logic. With our framework we can assign the task of the human operator to a suitable learning agent.

These advantages makes our framework suitable for complex robotics tasks, even with continuous action-spaces and high state-action dimensionality because the problem can be easily decomposed into different abstraction layers. The lower levels can control directly the actuators of the robot whereas the higher levels can focus on the more generic and complex tasks.

Existing methods to introduce hierarchy to reinforcement learning are based on the idea of partial policies with well defined termination conditions. MAXQ uses stack discipline to execute the hierarchical policy. Options framework adds the subtasks to the available action set. HAMs constrain the low level policy. Despite the fact that we can implement partial policies, we should take an action also in the last step of the low level episode. This is due to the fact that, differently from the other HRL approaches, our computational graph executes continuously, that is, blocks operate sequentially and only in one direction. An outcome of this feature is that the agents in the lower levels of the hierarchy must take an action even in the last steps of their episodes. It is not possible to go back to the higher level controller during the cycle. Another cycle is needed to reset and start a new episode for the low level controller that has its episode finished. This is the fundamental difference between our approach and the state of the art. A subtask cannot exist by itself but is always part of the complete system.

Designing hierarchical algorithms is not trivial due to the interaction between the layers. Typically, the algorithms in different layers are not aware of each other performance. They do not know about each other tasks. This causes the agents to interpret the outcomes in a wrong way. For instance, if the outcome of the environment is bad after the execution of an action drawn by an agent, the agent that receives the bad reward will assume the poor performance is due to its choice, and updates its quality to avoid such policy, even when it is not the case. In addition, since the agents observe the environment only partially, and it may change its dynamics due to the actions of other layers, their perception of the environment is non-stationary. This problem is more clearly seen in our framework. Blocks observe the environment only through their connections. This means that their observation over the graph is only partial and, as it changes in time, it is non-stationary. This behavior can be clearly identified in the ship steering environment: when the high level policy has not yet identified the gate position, low level agent learns a policy stating that it should not to follow the reference of the high level agent, and learning becomes unstable. To avoid the instability of the low level controller, we used a deterministic policy that guarantees the stability criterion.

We used this framework as a tool to build a hierarchical reinforcement learning solution for the ship steering problem that has continuous state-action space. Ship steering task is suitable for hierarchical decomposition and our control theoretic decomposition approach is easily adopted. The higher level subtask is defined as estimating the reference points in the map for the low level to follow and pass through a gate that is in an unknown fixed position. The low level subtask is to manipulate the rudder to decrease the error between the reference angle and the actual angle. A function block is added between these controllers to convert the reference position output of the high level controller to angle error input of the low level controller. Flat REPS and RWR algorithms learn faster than the hierarchical algorithms in the small domain. However, they suffer from premature convergence. Our frameworks performance is more stable and reliable compared to flat RWR and REPS and much faster than the flat PG algorithms. In the regular domain, learning curve is steeper when the learning algorithm of the low level controller is a black box optimization method. Also in the regular domain, the hierarchical algorithms built with our framework showed good performance in terms of learning speed and stability.

The task decomposition of the ship steering problem has a cascaded control structure as in DC motor control example. The framework supports building more complex hierarchical control systems such as those typical for hybrid systems. This can be achieved with selector blocks. Selector blocks implement the mode switching behavior in our framework. They contain block lists and activate one list at each cycle.

The proposed framework should be tested in more challenging environments and with more complex learning algorithms. Learning speed can be improved using off policy learning methods, exploiting the performance of the active controller to update the policy also of inactive controllers. This would result in more algorithms learning in parallel, similar to intra-option learning.

Another important issue to analyze is the fact that each controller see the rest of the system as a partially observable and non-stationary environment. Both distributed and global learning algorithm should be considered to face this issue and speed up learning.

Bibliography

[1] Charles W. Anderson and W. Thomas Miller. Neural networks for control. chapter A Challenging Set of Control Problems, pages 475–508. MIT Press, Cambridge, MA, USA, 1990.

[2] KJ Åström and Bjorn Wittenmark. Computer-controlled systems, ser. *Information and systems sciences series. New Jersey: Prentice Hall*, 1997.

[3] Peter L. Bartlett and Jonathan Baxter. Infinite-horizon policy-gradient estimation. *J. Artif. Intell. Res.*, 15:319–350, 2001.

[4] Jonathan Baxter and Peter L. Bartlett. Direct gradient-based reinforcement learning: I. gradient estimation algorithms. 1999.

[5] R.A. Brooks. Achieving artificial intelligence through building robots. In *Technical Report A.I. Memo 899, Cambridge, MA: Massachusetts Institute of Technology Artificial Intelligence Laboratory*, 1986.

[6] Christian Daniel, Gerhard Neumann, and Jan Peters. Hierarchical relative entropy policy search. *Journal of Machine Learning Research*, 17:93:1–93:50, 2012.

[7] Carlo D'Eramo and Davide Tateo. Mushroom. `https://github.com/AIRLab-POLIMI/mushroom`.

[8] H. Daniel Patiño Derong Liu. A self-learning ship steering controller based on adaptive critic designs. In *IFAC*, 1999.

[9] Thomas G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *J. Artif. Intell. Res.*, 13:227–303, 2000.

[10] Mohammad Ghavamzadeh and Sridhar Mahadevan. Hierarchical policy gradient algorithms. In *ICML*, 2003.

[11] Gregory Z. Grudic and Lyle H. Ungar. Localizing search in reinforcement learning. In *AAAI/IAAI*, 2000.

[12] Manfred Huber and Roderic A. Grupen. A feedback control structure for on-line learning tasks. *Robotics and Autonomous Systems*, 22:303–315, 1997.

[13] Jens Kober and Jan Peters. Policy search for motor primitives in robotics. *Machine Learning*, 84:171–203, 2008.

[14] Peter Marbach. Simulation - based optimization of markov. 1998.

[15] Ronald Parr and Stuart J. Russell. Reinforcement learning with hierarchies of machines. In *NIPS*, 1997.

[16] Jan Peters, Katharina Mülling, and Yasemin Altun. Relative entropy policy search. In *AAAI*, 2010.

[17] Jan Peters and Stefan Schaal. Applying the episodic natural actor-critic architecture to motor primitive learning. In *ESANN*, 2007.

[18] Jan Peters and Stefan Schaal. Natural actor-critic. *Neurocomput.*, 71(7-9):1180–1190, March 2008.

[19] Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. Policy gradients with parameter-based exploration for control. In *ICANN*, 2008.

[20] R. S. Sutton, A. G. Barto, and R. J. Williams. Reinforcement learning is direct adaptive optimal control. *IEEE Control Systems*, 12:19–22, 1991.

[21] Richard S. Sutton, Doina Precup, and Satinder P. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artif. Intell.*, 112:181–211, 1999.

[22] Sebastian Thrun and Anton Schwartz. Finding structure in reinforcement learning. In *NIPS*, 1994.

[23] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.