# Graph-Based Design of Hierarchical Reinforcement Learning Agents

Davide Tateo[1], İdil Su Erdenlič[2], and Andrea Bonarini[1]

***Abstract*— There is an increasing interest in Reinforcement Learning to solve new and more challenging problems, as those emerging in robotics and unmanned autonomous vehicles. To face these complex systems, a hierarchical and multi-scale representation is crucial. This has brought the interest on Hierarchical Deep Reinforcement learning systems. Despite their successful application, Deep Reinforcement Learning systems suffer from a variety of drawbacks: they are data hungry, they lack of interpretability, and it is difficult to derive theoretical properties about their behavior. Classical Hierarchical Reinforcement Learning approaches, while not suffering from these drawbacks, are often suited for finite actions, and finite states, only. Furthermore, in most of the works, there is no systematic way to represent domain knowledge, which is often only embedded in the reward function.**

**We present a novel Hierarchical Reinforcement Learning framework based on the hierarchical design approach typical of control theory. We developed our framework extending the block diagram representation of control systems to fit the needs of a Hierarchical Reinforcement Learning scenario, thus giving the possibility to integrate domain knowledge in an effective hierarchical architecture.**

## I. INTRODUCTION

There is a growing interest in complex robotics systems, particularly in autonomous mobile robots and unmanned autonomous vehicles. In these contexts, it is often useful to build an agent able to react to previously unseen scenarios and non-stationary environments. To face these challenges, a wise usage of machine learning, and Reinforcement Learning (RL) in particular, becomes crucial.

Deep RL algorithms have shown the potential to solve complex problems, also in robotics-related domains [1], [2] characterized by continuous actions. However, plain Deep Learning approaches often exhibit unstable behaviour during the learning process, and the interpretation of Deep Networks is extremely difficult, if not impossible.

For these reasons, in recent years we have seen a renewed interest in Hierarchical Reinforcement Learning (HRL). The first approaches to HRL can be divided in three major categories: the Hierarchy of Abstract Machines framework [3], the Max-Q approach [4], and the option framework [5]. All these approaches are based on classical RL theory, and are particularly suited to finite state and action representations. Among these works, Ghavamzadeh & al. have proposed the Hierarchical Policy Gradient approach [6], suitable for continuous action representation.

To face the new challenges imposed by emerging applications, researchers have started to develop Deep Learning approaches to HRL. Feudal Networks [7] are based on one of the first approaches to HRL, the Feudal Q-Learning algorithm [8]. However, this work has been developed mainly considering a two-level hierarchy, and the algorithm displays some major differences w.r.t. the original approach. In [9], the authors present a Policy Gradient formulation able to learn options policy and termination conditions without any explicit subgoal discovery algorithm. A promising Deep Learning approach is the Hindsight Experience Replay [10], where a particular replay memory is exploited to learn multiple subgoals and to improve the learning performance, particularly in the case of sparse reward functions. In [11], these ideas are extended in an actor-critic framework. Other Deep HRL methods are based on pre-training the low level skills [12]. These methods are better suited to face new complex problems, but still share the major drawbacks of Deep learning approaches. Differently from classical HRL approaches, they have few theoretical properties, are computationally and data hungry, and are complex to tune and implement. Moreover, most of the Deep and classical HRL approaches share the difficulty to exploit domain expert knowledge: this is usually put into predefined policies, skills, and reward functions. As it can be seen in [6], HRL algorithms often require a custom implementation for each problem.

Control Theory, instead, provides a well defined framework to build hierarchical control structures for complex systems. One of the fundamental design tools of control theory is the block diagram. Block diagrams can model systems composed by plants, sensors, controllers, actuators, and signals. With block diagrams it is easy to describe the control architecture of any kind of control system. The block diagram is based on blocks and connections. A block can represent a dynamical system, or a function, and connections represent the flow of the input and output signals among blocks.

This paper presents an approach based on the same idea, with modifications needed to fully describe an HRL system. We believe that this representation is beneficial in general for an HRL system and, in particular, for robotic applications, where the control system often comes structured as a block diagram. There exist some Deep HRL approaches that have faced the problem starting from a control-theoretic point of view (e.g., [13]), but our approach is more general and formal. It allows to easily implement existing control structures and policies (designed using control theory), and to use RL algorithms to fine-tune or adapt the parameters to changing dynamics or unknown scenarios. Our framework also eases the process of mixing low-level controllers with

[1]Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di Milano, Piazza Leonardo da Vinci 32, 20133 Milano, Italy {davide.tateo, andrea.bonarini}@polimi.it

[2]su.erdenlig@gmail.com

high-level RL techniques that can be effectively applied when the interaction model is not known. This makes the design of the HRL simple, and decouples the learning algorithms from domain-specific details.

## II. PRELIMINARIES

An RL problem consists of two main components: the agent (i.e., the decision maker and learner), and the environment. The agent interacts with the environment by selecting an action according to its perceptions of the environment state, while the environment reacts to the agent's actions by evolving to a possibly different state and providing a performance signal. The agent can be represented in terms of a learning algorithm and a policy; everything else is part of the environment, e.g., in the case of a robotic agent, the state of its joints. A (stochastic) policy is a probability density distribution $\pi(\cdot|x)$ over the action space $\mathcal{U}$ given the current state $x \in \mathcal{X}$. The objective of the learning algorithm is to maximize the performance of the policy w.r.t. a metric computed using the reward signal, e.g., the expected discounted return. The agent can have an internal state or other data structures, e.g. a value function or a replay memory. The environment can be partially observable: this happens when the observations do not provide a complete representation of the state of the environment.

The most common type of environment in RL is the Markov Decision Process (MDP), which is defined as a tuple $\mathcal{M} = \langle \mathcal{X}, \mathcal{U}, \mathcal{P}, \mathcal{R}, \gamma, \iota \rangle$, where $\mathcal{X}$ is the state space, $\mathcal{U}$ is the action space, $\mathcal{P}$ is a Markovian transition model where $\mathcal{P}(x'|x,u)$ defines the transition density for reaching state $x'$, starting from state $x$ and applying action $u$, $\mathcal{R}$ is the reward function, where $\mathcal{R}(x,u,x')$ is the reward obtained by the agent when it takes the action $u$ in state $x$, reaching the state $x'$, $\gamma \in [0,1)$ is the discount factor and $\iota$ is the distribution of the initial state.

For practical design reasons, we will consider agents displaying only two functionalities: the *act* procedure, that selects an action to be applied in the current state using the policy of the agent, and the *fit* procedure, that updates the policy and data structure using a given dataset. Every RL algorithm can implement this interface, at the possible cost of storing additional information at each procedure call. By exploiting this interface we are able to consider both batch and online learning algorithms in the same framework: the only difference will be the sequence of *act* and *fit* calls.

## III. FORMAL DEFINITION

We now describe the formal structure of our framework. To this aim, we start by modifying the standard interaction model for RL. Instead of having a system composed only of an agent and an environment, we introduce a control graph to describe their interactions.

### A. The Control Graph

The *Control Graph* is defined as follows:

$$\mathcal{G} = (\mathcal{E}, B, D, C, A)$$

where $B$ is the node set, $D$ is the data edge set, $C$ is the reward edge set, $A$ is the alarm edges set, and $\mathcal{E}$ is the environment.

Each node $b \in B$ represents a subsystem of the control structure. We will refer to the nodes in the set $B$ as blocks. Each block can contain either a dynamical system, a function, or an RL agent. Each block is characterized by a set of input and output signals, exchanged with other blocks or the environment. These signals are represented in the graph by data edges $d \in D$. For each block, at each control cycle, we call *state* the vector of its input signals, and *action* the vector of its output signals. If a block contains an RL agent, then it requires a reward signal to evaluate its performance and optimize its objective function. The reward signal can be produced either by the environment or by another block. Every edge $c \in C$ represents a reward signal connection. Finally, each block can produce an auxiliary output called alarm. The alarm signal can be read by other blocks. Every edge $a \in A$ represents an alarm signal connection.

We call as $\mathcal{E}$ everything that is outside the set of blocks $B$, which represents, along with the interconnecting edges, the hierarchical agent. The environment is modeled in the graph by means of three special blocks, called *interfaces*:

- the state interface, which contains the current environment observation;
- the action interface, which contains the action to be applied to the environment at the beginning of each control cycle, which can also be used by other blocks, e.g., to compute a quadratic penalty over the last action used;
- the reward interface, which contains the last reward produced by the environment.

All the edges in $\mathcal{G}$ are directed, and the graph must be acyclic when considering all edges except the ones connected to the action interface. At least one block must be connected to the action interface.

### B. The Control Cycle

*Control Cycle* is the cycle describing the interaction of the control system with the environment. Each cycle starts by obtaining an observation from the environment through the state interface. Then, each block of the graph is evaluated by collecting all its inputs into the state vector, and producing an action as output vector. Also the information coming from the reward and alarm connections are considered, if they exist. Each block must be evaluated after all of its inputs, reward, and alarm signals have been computed. This means that blocks must be evaluated following a topological ordering, where the last block to be evaluated must be the action interface. After the action interface has been updated, the action is applied to the environment and a new control cycle starts. While the same structure can be used for continuous time systems, as it is often done in control, we focus in this work just on discrete time systems.

### C. Connections

In this model, three different types of connections have been devised. Data connections must be differentiated from reward connections, since they have different roles. Notice that, differently from many other HRL frameworks, in our approach there is no explicit representation of goals: the goal is fully described by the reward function signal, while the observed variables and computed actions are carried by the data connections. This design choice is motivated by the fact that we want to be able to exploit the RL algorithms and theory defined for non-hierarchical agents. The last type of connection is the alarm connection, which plays a relevant role in implementing control systems working at different time scales. Indeed, at each control cycle, all blocks are evaluated and this behaviour makes it impossible to have controllers that work on different time scales. However, each block can maintain the previous output signal until an event occurs. Events can be notified to other blocks by using alarm signals. Every block can generate alarm signals and can exploit the information of the alarm in any useful way. The introduction of alarms allows to implement event-based controls, e.g., to wake one of the (RL) agents in the control graph only when a particular state of the environment is reached. It is also possible to create temporally extended actions by raising an event from a lower-level agent whenever its episode terminates. Consequently, the higher level agent can only choose its action when the effect of the related temporally extended action is finished. By exploiting alarm connections, it is possible to implement –using the Control Graph formalism– other HRL frameworks (e.g., options) with minor modifications.

### D. Blocks

Blocks contain the components of the hierarchical agent. We can look at the single block as an isolated agent considering the remaining part of the graph as a part of the environment. We call as $\mathcal{E}_i$ the *induced environment* seen by each block $b_i$. The concatenation of the signals of incoming data connections can be seen as the current observation (or, in some cases, the state) of $\mathcal{E}_i$, while the outgoing data connection can be seen as the action applied to $\mathcal{E}_i$. Finally, the reward connection can be seen as the reward of the induced environment. The nature of each $\mathcal{E}_i$ depends on the structure of the control graph, but, in general, it can be a partially observable, time variant, stochastic, non-Markov environment. This makes it impossible to derive general convergence results for arbitrary control graphs, as the theoretical characteristics of the induced environment for each block are arbitrary. This consideration is particularly important, and makes a careful design of the graph structure, the RL agents, and the performance metrics, crucial. To mitigate this issue, specific learning algorithms or a multi-block algorithm could be exploited.

Each block can contain either a function, a dynamical system, or an RL agent. This makes the block a very general and versatile component. Given the definition of induced environment, it is straightforward to apply classical RL theory in this framework. We present here below some of the most useful blocks we have developed, which are the basic building blocks of the majority of the control graphs.

*a) Interface Block:* is the interface to the environment. The environment can read and write data by these blocks.

*b) Function Block:* has the only task of computing a function from the input vector e.g., to compute feature vectors from state vectors. They are generally stateless blocks.

*c) Learning Block:* contains an RL agent. The purpose of the learning block is to interface a generic RL agent to the control graph. The learning block applies the *act* procedure of the agent on the incoming state, and produces the output action. It also collects the transition dataset and activates the *fit* procedure of the agent when needed. The frequency of the calls to the *fit* procedure depends on the nature of the algorithm (online or batch) and on the batch size. It is possible to truncate episodes of each learning block through a termination condition that can be either a fixed number of steps, or state dependent. However, it is not possible to define absorbing states for subtasks, since at each control cycle each block has to return an action. This implies that in any non-absorbing state of the original environment, the learning block has to provide an action, even if the block termination condition has been met. This behavior is fundamentally different from that of other HRL methods. This may be a problem when considering subtasks in finite state MDPs, as it is not possible to define absorbing states for the subtasks, as commonly done in classical HRL. However, it is not a problem when dealing with continuous environments with fine grained time discretization, where the effect of a single action is not affecting heavily the general behavior. Indeed, this is the case of most robotic environments. The final purpose of the learning block is to handle signals and synchronization with others blocks. In order to do so, it raises the alarm signal whenever the termination condition is met. The control has two synchronization behaviors: if there is no alarm connection, the agent policy will be called at each control cycle. If any alarm signal is connected to the agent, then the agent policy will be executed only when one of the connected alarms has been raised. Only the states coming from the blocks that raised the alarms (or the initial environment state) are considered in input. The learning block maintains the last selected output value until the next event occurs.

*d) Reward Accumulator Block:* is a particularly important block with state, used in conjunction with cascade control systems that operate at a different time scale. Indeed, when computing the reward of a temporally extended action, it is needed to accumulate the reward of any intermediate steps, discounting the new rewards by the appropriate discount factor.

*e) Selector Block:* allows to select one chain from a list of chains of blocks. The selection is done by the first input of the block, which must be an integer value. Further inputs are passed as inputs to the first block of the selected chain. The output of the selector block is the output of the last block of the chain. The blocks in the chain can be connected

to other blocks through alarm and reward connections. This enables the conditional computation of blocks, allowing to have different low-level RL agents that represent different temporally extended actions.

## IV. COMPARISON WITH OTHER FRAMEWORKS

Existing approaches are mostly based on the concept of macro actions. Macros are executed following the stack principle, where each macro can call another one, until a primitive action is executed. After the macro has been executed, the control returns to the one that activated it. Both MAX-Q and the option frameworks are based on this concept. The HAM framework is also based on this idea, but learning is performed on the "reduced" SMDP, reducing learning to flat SMDPs. Although this is a powerful approach, it is not the most natural approach in control systems, where decentralized controllers work in parallel, and each controller regulates a specific part of the system.

Control systems design is fundamentally different from the macro concept. There are two main concepts that must be considered when comparing our framework to other existing HRL frameworks. The hierarchy is formed by the structure of the control system, and not by the stack discipline. This has a major impact on how subtasks are defined: a subtask is not anymore a function call that executes until termination, but can be better seen as a setpoint to be reached by a lower level controller. Furthermore, in most HRL approaches the state seen by each level of the hierarchy is the environment state itself or a state abstraction [4], [6]: in HCGL, instead, we consider each block as an independent RL agent. State and action are then related to the induced environment: this makes it possible to use any suitable RL agent for learning the subtask. Furthermore, using HCGL the definition of parametrized continuous subtasks is straightforward, and does not need any deviation from the framework definition or special handling. Differently from Feudal Q-Learning [8], its Deep Learning version [7], and other Deep approaches, we do not add any sub-goal specification as input together with the current state.

HCGL has been developed with a slightly different objective w.r.t. the classical and Deep HRL frameworks. The options framework objective is to enrich the MDPs action set in order to improve exploration by following a sub-policy for an extended period of time. The objective of the MAX-Q framework is to have a factored representation of the state-action value function, in order to reuse the information for different high-level tasks and to transfer learning. The objective of the HAM framework is to exploit the expert knowledge to constrain the policy, imposing the desired behavior and improving learning speed by reducing exploration and parameters to learn. The objective of our framework is twofold: simplify the design of hierarchical agents, by providing a flexible design tool, and favor the reuse of existing RL algorithms.

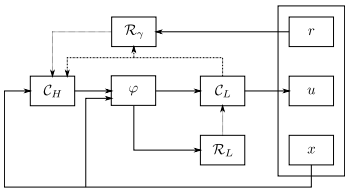


Fig. 1. Control Graph used in the ship steering environment experiment

## V. EXPERIMENTAL RESULTS

The main objective of this section is to show that it is easy to design RL structures using the control graph formalism, and that it is indeed possible to use different learning algorithms inside this system. As our method does not impose any learning algorithm to be used in the design, it is not relevant to compare the raw performances w.r.t. a specific algorithm. Indeed, it is reasonable to assume that any algorithm can be outperformed by adding specifically designed expert knowledge. The implementation and all the details of every experiment in this work can be found in [14]. Every setting is evaluated using 100 independent runs for each algorithm considered.

### A. Ship steering

This problem consists of driving a ship through a gate, without going outside a defined region (see [6] for details). We used two different versions of the ship steering environment: the small and the big environments, both with a single gate. The big environment is the one presented in [6], while the small environment is a reduced version where the area is $150m \times 150m$ and the gate is the line joining the points $(100, 120)$ and $(120, 100)$.

*1) Small environment:* We use this version of the environment to show how a hierarchical formalization of the environment with our framework is beneficial for learning also in environments that are easy to learn with flat Policy Search methods. For each algorithm, the experiment consists of 25 epochs of 200 episodes each. After each epoch an evaluation run (with no learning) of 50 episodes is performed.

The hierarchical structure used is shown in Figure 1. The high level controller $\mathcal{C}_H$ selects a position setpoint for the ship. The function block $\varphi$ transforms this position setpoint into the error of the heading of the ship and the distance of the current position and orientation of the ship w.r.t. the given setpoint. The low level controller $\mathcal{C}_L$ learns a policy to drive the ship towards the selected setpoint, and its output is the desired turning rate of the ship. The high level control changes its setpoint only when a low level environment episode terminates: this is possible due to the alarm connection between the two blocks. The low level environment episode terminates after 100 steps or when the distance to the setpoint is less than $0.2m$. The high level controller uses the reward function of the environment as performance metric. The reward between each different setpoint is accumulated by the reward accumulator block $\mathcal{R}_\gamma$, which properly discounts the reward at each step using the discount factor $\gamma = 0.99$ of the environment. The low

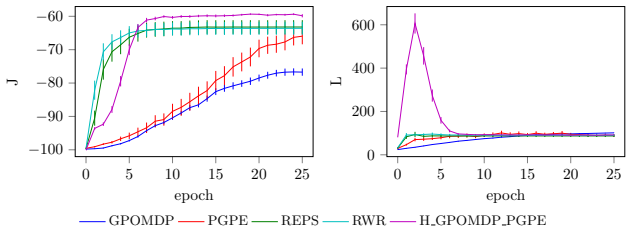


Fig. 2. Learning curves for the ship steering small environment

level reward is provided by the function block $\mathcal{R}_L$, which computes the cosine of the heading error. In this experiment, we used the GPOMDP algorithm [15] as high level controller and the PGPE algorithm [16] for the low level one. All the flat algorithms use tiles as features over the state. For the flat experiments, we selected the GPOMDP algorithm as an example of gradient method, and three other black box algorithms: PGPE [16], RWR [17] and REPS [18].

Results are shown in Figure 2. By looking at the objective function, it is clear that the policy gradient approaches, PGPE and GPOMDP are the slowest and achieve the worst performances. RWR and PGPE quickly converge to good performances, however they may get stuck to slightly sub-optimal policies. RWR converges faster than REPS, but it is slightly more prone to premature convergence. Our approach achieves better performances on this task, and the results are much less variant, particularly at the end of training. The performance gain and the reduced variance of the learning curve is due to the fact that the hierarchical approach can represent a more structured policy, with fewer parameters and more expressiveness in terms of possible behaviors. The hierarchical decomposition splits the problem in two by decomposing the high level problem (reach a point) from the low level control task (steer the ship), making it possible to reuse the low level control policy from each point and each orientation of the ship. This results in a policy that is easy to interpret and to be used in other contexts, e.g., the low level policy learned in the small environment may be used also in the big ship steering environment. The learned high level policy is a good policy for most of the possible alternative starting points of the environment. The learning curve behavior can be easily understood by considering the episode length. Flat algorithms tend to increase the episode length progressively, to avoid bringing the ship outside the bounds. As the gate is located at the end of the diagonal, they will find the gate while moving towards longer trajectories. The hierarchical approach instead behaves very differently: at the beginning most of the trajectories loop around the center of the environment, increasing the episode length, while the low level starts to learn to reach the setpoint appropriately, and the high level learns to avoid the map boundaries. When some trajectories are able to cross the gate, the high level starts to learn the position of the gate and the variance of the high level policy reduces towards 0. This, jointly with the learning of the optimal low level policy, reduces the episode length until the optimal performance is reached.

The hierarchical method has some drawbacks w.r.t. the other methods. It can be noticed that the number of steps needed to learn is greater than that needed by the other methods, although the number of failed episodes (going outside the bounds) is smaller. This turns out to be an advantage, rather than a problem, if safety is a major concern, as we avoid the most "dangerous" event. Furthermore, due to the fact that the learning of each agent of the hierarchy is performed independently, we must choose carefully the learning algorithms and the policies. Indeed, we forced the low level policy to be stable. This issue is present because the induced environment of the low level policy is partially observable: this may cause some ambiguities in interpreting the results of the selected behavior. The low level controller may learn an unstable controller that forces the ship to go out of bounds to terminate the episode earlier. Given an unstable low-level controller, the high-level one could learn to move the set point in the opposite direction w.r.t. the actual goal, which is an undesired behavior. The possibility of undesired interactions between learning processes is a major issue of the framework. To avoid this, a careful design of the policy, and a good initialization of the policies from expert knowledge could be exploited.

*2) Big environment:* With this version of the environment we will show how our method is able to scale to bigger and more complex problems and compare it, from both the point of view of the performance and the design of the hierarchical agent, with existing state of the art method, in particular with the hierarchical policy gradient approach shown by Ghavamzadeh & al. [6]. In this environment, the state space dimension ($1000 \times 1000m$) makes it difficult to build generic features that allow for both a complete coverage of the state space and a fine discretization needed to have a fine-grained action selection.

For each algorithm, the experiment consists of 50 epochs of 800 episodes each. After each epoch an evaluation run (with no learning) of 50 episodes is performed. Differently from the small environment, the initial state of the environment is sampled uniformly from $\mathcal{X}$. For our method, we used the same control graph used for the small environment (Fig. 1). The Ghavamzadeh method is implemented using our framework. This has a minor impact, as our framework is not able to represent absorbing sub-task states. However, given the dimension of the environment, the properties of the policy and of the low level task, we can consider this issue as irrelevant. The control graph used to implement the Ghavamzadeh's method is shown in Fig. 3. The function block $\varphi$ represents the state of the environment using a $20 \times 20$ tiling discretization over the first two state variables (the position of the ship in the environment). The high level controller $\mathcal{C}_H$ takes as input the discretized state and selects one of the 8 possible directions to reach. The function block $S$ transforms one of the possible directions in a binary value, to select the straight/diagonal sub-task of the connected selector block. This is done because in the original work symmetry is exploited to group together the sub-tasks, instead of learning each sub-task independently. The function

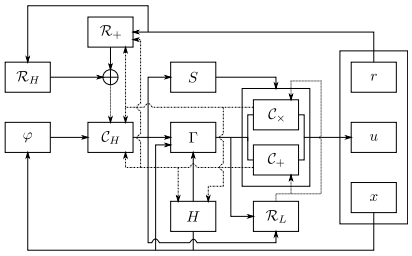


Fig. 3. Control graph used to implement the Ghavamzadeh's algorithm

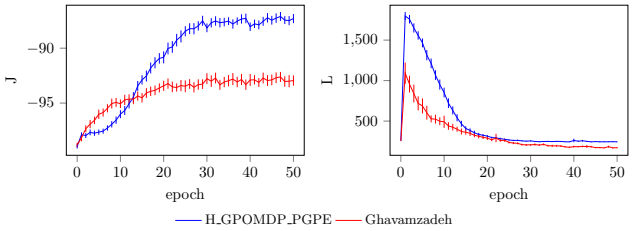


Fig. 4. Learning curves for the ship steering environment

block $H$ is used to hold the value of the starting position of the subtask. $H$ is needed because the $\Gamma$ block performs the needed transformation to map the state of the original environment in the sub-task by rototranslating appropriately the current state w.r.t. the position in which the sub-task has begun. For the straight sub-task, the initial state is mapped into $[40, 75]$, while for the diagonal is mapped into $[40, 40]$. All straight sub-tasks are rotated into the "right" sub-tasks, while the diagonal subtasks are rotated into the "up-right" sub-task. The two learning blocks $\mathcal{C}_+$ and $\mathcal{C}_\times$ are the two controllers that learn the straight and horizontal sub-task respectively. The reward for the high level block is the sum of the additional reward computed by the function block $\mathcal{R}_H$, which gives a reward of 100 for crossing the gate, and the reward accumulator block $\mathcal{R}_+$, which computes the sum of every reward during the low level episodes. The reward for the low level controller is computed by the function block $\mathcal{R}_L$, which gives -100 for going outside the low level environment area (a squared region of $150m$), +100 for being closer than 10 meters to the objective of the low level task ($[140, 75]$ for the straight and $[140, 140]$ for the diagonal), plus the angular penalty $r_{\text{extra}}$, which penalizes the angular error, as described in [6]. The low level episodes terminate when the ship reaches the goal or goes outside the low level task region.

We have used the Q($\lambda$) algorithm in the same way as described in [6] for the high-level controller and the GPOMDP algorithm for the low level controller, that had better performances than the original gradient algorithm proposed in [6]. The settings for our framework are the same of the small environment experiment. The only difference is the initialization and parameters for the high level controller, that are scaled accordingly to the different scale of the environment.

As it can be seen from Figure 4, the hierarchical agent learning is slower than the Ghavamzadeh approach in the beginning. This is due to the time spent by the ship moving around the map, while the position of the gate is learned and the optimal policy for the low level is found. After the position of the gate is found by the high level controller, the length of the trajectories quickly decreases and the algorithm converges rapidly to good performances. This is due to the fact that the position of the gate does not depend on the current position of the ship, thus, learning this information results in a policy that generalizes to the whole state space, instead of being useful only locally, as the policy learned by the Ghavamzadeh's approach. By looking at the episode length, we can see that Ghavamzadeh converges to shorter trajectory lengths, and this is due to sub-optimal behavior learned by the agents along the borders, where the agent prefers to go outside of the map instead of going towards the center of the environment. This is due to the fact that the Q values must be propagated from the center of the map towards the boundaries, and this propagation is affected negatively by several factors. Among these issues, we can consider the dimension of the state space, the small number of trajectories on the boundaries o the environment, and the initial low level performance at the beginning of the learning process, which can affect the initial updates of the Q-function. Our approach instead learns the general objective, leading to a good generalization in almost any state, that is particularly helpful when starting in unseen states.

Another important aspect that should be considered is the design of the algorithm. Our framework allows to easily design a hierarchical system by including the function blocks needed to exploit expert domain knowledge. The learning algorithm can be any off-the-shelf learning algorithm taken from the state of the art. Therefore, domain experts only need to focus on the structure of the problem and not on learning algorithms. This is particularly useful in order to bring the RL tools to industrial applications. Furthermore, the design tool is close to what engineers use in other tasks, as it is inspired by control theory block diagrams. The Ghavamzadeh's approach, instead, can't be implemented in a general way, but must be re-implemented for each problem instance, in order to easily exploit domain knowledge. Furthermore, while the Ghavamzadeh's approach looks extremely intuitive from the point of view of RL researchers, which are used to work with options and Semi-Markov Decision Processes, it is not intuitive from the point of view of a control engineer. This can be easily seen by trying to implement the Ghavamzadeh approach in our framework: it is possible to implement it (with minor changes), but the resulting design looks overly complicated, particularly if compared with our model.

### B. Segway

This problem consists of balancing a 2D Segway on a fixed point. The experiment for this environment shows how the presented tool can be suitable also for the design of classical control applications, and that there are some advantages in using such hierarchical learning approach instead of using black box optimization on a complex non differentiable equivalent policy. For this example we used the REPS and

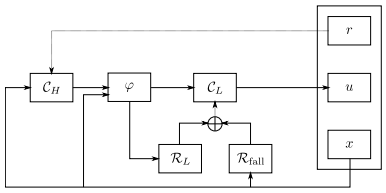


Fig. 5. Control graph used in the Segway experiments

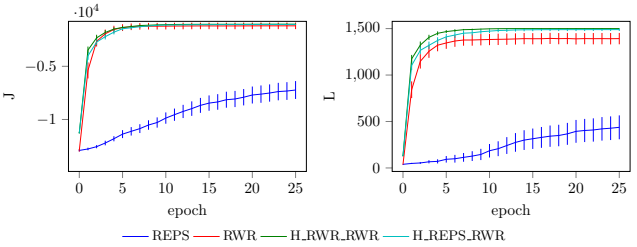


Fig. 6. Learning curves for the Segway experiment

the RWR black box algorithms, to compare our distributed learniong approach with the centralized one. For each algorithm, the experiment consists of 25 epochs of 100 episodes each. After each epoch, an evaluation run of 100 episodes is performed. To further exploit the distributed learning supported by our approach, boosting the performances, we avoid to learn the high level controller for the first two epochs.

The control scheme is reported in Figure 5; it is extremely simple, as it matches existing control schemes for similar platforms. The high level controller $\mathcal{C}_H$ is a simple proportional controller over the linear position that computes the angular setpoint. The function block $\varphi$ removes the linear component from the current state and considers the error w.r.t. the desired setpoint instead of the actual angle. The low level controller is another proportional controller over the full state computed by the function block $\varphi$. We partially enforce the stability of the low-level policy, by avoiding the use of negative gains as policy parameters. Figure 6 shows that the hierarchical architecture improves learning stability, as the hierarchical algorithm achieves the optimal performance more consistently. It is important to notice that, while the difference in performance in terms of the objective function is not statistically relevant, at least compared with the RWR method, it becomes relevant in terms of episode length, where our approach is able to reach the horizon faster and more consistently. This is an important result, as this means that our approach is less prone to failures, i.e., the robot falling, before the end of the episode.

This experiment shows that this approach can improve the performance of black box optimization by learning subsystems independently, and that it may be beneficial to slow down the learning of higher level controllers, to have good low level policies. Moreover, with this approach it is easier to highlight the subsets of the systems parameters that do not need correlated exploration, extending the applicability of black box optimization to larger systems.

## VI. CONCLUSION

We presented a novel HRL framework based on the block diagram, a well known tool of control theory. We have demonstrated the effectiveness of our approach in two classical robotic and control tasks. We believe that the proposed framework has a huge potential, particularly in the design of industrial applications, as it makes it possible to exploit the knowledge from classical engineering and control theory, while re-using any RL algorithm, ranging from the classical ones, such as Q-Learning, to Deep Policy Search approaches, or any algorithm developed for Semi-Markov Decision Processes.

Our framework is particularly beneficial when dealing with robotics and autonomous vehicles, both because the Control Graph already resembles the existing control structure for such systems, and because the structure naturally fits all the scenarios where set points of each subsystem are continuous action values.

## REFERENCES


[1] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proc. ICML2015*, 2015, pp. 1889–1897.

[2] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[3] R. Parr and S. J. Russell, "Reinforcement learning with hierarchies of machines," in *Proc. NIPS97*, 1997.

[4] T. G. Dietterich, "Hierarchical reinforcement learning with the maxq value function decomposition," *J. Artif. Intell. Res.*, vol. 13, pp. 227–303, 2000.

[5] R. S. Sutton, D. Precup, and S. P. Singh, "Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning," *Artif. Intell.*, vol. 112, pp. 181–211, 1999.

[6] M. Ghavamzadeh and S. Mahadevan, "Hierarchical policy gradient algorithms," in *Proc. ICML*. AAAI Press, 2003, pp. 226–233.

[7] A. S. Vezhnevets, S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, and K. Kavukcuoglu, "Feudal networks for hierarchical reinforcement learning," *arXiv preprint arXiv:1703.01161*, 2017.

[8] P. Dayan and G. E. Hinton, "Feudal reinforcement learning," in *Proc. NIPS93*, 1993, pp. 271–278.

[9] P.-L. Bacon, J. Harb, and D. Precup, "The option-critic architecture." in *Proc. AAAI2017*, 2017, pp. 1726–1734.

[10] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. P. Abbeel, and W. Zaremba, "Hindsight experience replay," in *Proc. NIPS2017*, 2017, pp. 5048–5058.

[11] A. Levy, R. Platt, and K. Saenko, "Hierarchical actor-critic," *arXiv preprint arXiv:1712.00948*, 2017.

[12] C. Florensa, Y. Duan, and P. Abbeel, "Stochastic neural networks for hierarchical reinforcement learning," *arXiv preprint arXiv:1704.03012*, 2017.

[13] X. B. Peng, G. Berseth, K. Yin, and M. Van De Panne, "Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning," *ACM Transactions on Graphics*, vol. 36, no. 4, p. 41, 2017.

[14] D. Tateo and I. Su Erdenliğ, "Mushroom Hierarchical," https://github.com/AIRLab-POLIMI/mushroom_hierarchical.

[15] P. L. Bartlett and J. Baxter, "Infinite-horizon policy-gradient estimation," *J. Artif. Intell. Res.*, vol. 15, pp. 319–350, 2001.

[16] F. Sehnke, C. Osendorfer, T. Rückstieß, A. Graves, J. Peters, and J. Schmidhuber, "Parameter-exploring policy gradients," *Neural Networks*, vol. 23, no. 4, pp. 551–559, 2010.

[17] M. P. Deisenroth, G. Neumann, and J. Peters, "A survey on policy search for robotics," *Foundations and Trends in Robotics*, vol. 2, no. 12, pp. 1–142, 2013.

[18] J. Peters, K. Mülling, and Y. Altun, "Relative entropy policy search." in *Proc. AAAI2010*. Atlanta, 2010, pp. 1607–1612.