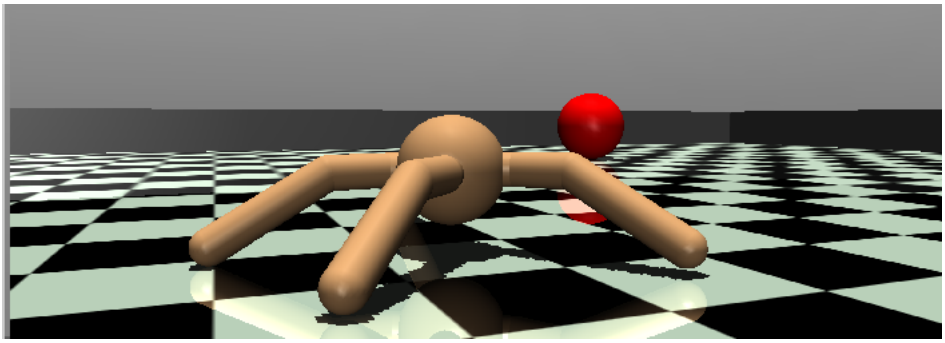
TÉCNICO LISBOA



Using feudal hierarchies for non-stationary reinforcement learning

Guilherme Garcia Inácio de Freitas Jardim

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Francisco Antonio Chaves Saraiva de Melo
Prof. Jose Alberto Rodrigues Pereira Sardinha

Examination Committee

Chairperson: Prof. Luís Manuel Antunes Veiga
Supervisor: Prof. Francisco Antonio Chaves Saraiva de Melo
Member of the Committee: Prof. Manuel Fernando Cabido Peres Lopes

November 2024

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

Quero começar por agradecer ao meus dois orientadores, o professor Francisco Melo e o professor Alberto Sardinha, que apesar das outras responsabilidades que acumularam ao longo do tempo que eu levei a acabar a dissertação, tiveram um papel fundamental a moldá-la e deram-me as ferramentas necessárias para eu conseguir produzir este documento. Um obrigado especial ao Diogo Carvalho por todas as reuniões e conselhos, e por ter sido um apoio sem o qual esta reta final não teria sido possível.

Quero também agradecer profundamente à minha família mais próxima, a minha mãe, o meu pai e a minha irmã. Com alguma impaciência à parte, senti que sempre me apoiaram ao longo destes meses e a sua compreensão não passou despercebida. Obrigado especialmente à minha mãe, que percebendo tão pouco do tema conseguiu dar-me conselhos valiosos sem os quais não teria conseguido. Obrigado também a todos os familiares que à sua maneira me tentaram ajudar quando nem tudo era um mar de rosas.

Por fim, quero agradecer aos meus amigos, mesmo aqueles que poderão ter atrasado este processo, por se terem mostrado tão preocupados com a minha tese. Uma menção especial àqueles que sentiram esta dissertação quase tão intensamente como eu senti, porque o meu carinho por vocês merece ficar registado num documento formal.

Abstract

Hierarchical Reinforcement Learning (HRL) has emerged as an answer to the main problems with traditional Reinforcement Learning such as data inefficiency and inability to transfer acquired knowledge for similar tasks. This work introduces a Contextual Hierarchical Actor-Critic (CoHAC), a HRL algorithm designed for addressing the challenges presented by non-stationary environments, particularly those that switch between a finite set of modes of operation. The proposed framework uses a multiple layer hierarchy, with the higher layers responsible for designing the general strategy while the lower layers are responsible for interacting with the environment and dealing with non-stationarity. We incorporate a context-detector with the framework of feudal hierarchies in an attempt to leverage the ability to operate in different time granularity to respond more effectively to changes in the mode of operation of the environment, as well as to isolate non-stationarity in the lower layers of our hierarchy. This approach is evaluated against state-of-the-art HRL methods in environments characterized by sparse rewards, high-dimensional state spaces, and varying modes of operation.

Keywords

Reinforcement Learning; Feudal hierarchies; Non-stationarity

Resumo

Hierarchical Reinforcement Learning tem surgido como a resposta para os principais problemas com a Aprendizagem por Reforço tradicional, tais como ineficiência de dados e a incapacidade de transferir conhecimento adquirido para tarefas parecidas. Este trabalho introduz o Contextual Hierarchical Actor-Critic (CoHAC), um algoritmo hierárquico desenhado para combater os desafios apresentados por ambientes não-estacionários, particularmente ambientes que alteram entre um conjunto limitado de modos de operação. A proposta apresentada usa uma hierarquia com várias camadas, em que as camadas mais altas são responsáveis pelo planeamento de uma estratégia abrangente, enquanto as camadas mais baixas são responsáveis por interagir com o ambiente e por lidar com a não-estacionaridade. Incorporamos um detetor de contexto com a framework de hierarquias feudais, aproveitando a sua habilidade de operar em diferentes níveis de granularidade temporal para responder eficazmente a mudanças no modo de operação do ambiente e para isolar a não estacionaridade nas camadas mais baixas da nossa hierarquia. A proposta apresentada é avaliada contra o estado da arte em HRL em ambientes que têm recompensas esparsas, um espaço de estado de alta dimensionalidade e diferentes modos de operação.

Palavras Chave

Aprendizagem por Reforço; Hierarquias Feudais; Não-estacionaridade

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

CoHAC	Contextual Hierarchical Actor Critic
DQN	Deep Q-Network
DIYAN	Diversity Is All You Need
DDPG	Deep Deterministic Policy Gradient
FAQL	Frequency Adjusted Q-Learning
FuN	FeUdal Networks
HAC	Hierarchical Actor Critic
h-DQN	hierarchical Deep Q-Network
HER	Hindsight Experience Replay
HIRO	HIerarchical Reinforcement learning with Off-policy correction
HRL	Hierarchical Reinforcement Learning
LSTM	Long Short-Term Memory
MAE	Mean Absolute Error
MDP	Markov Decision Process
MMRL	Multiple Model-Based Reinforcement Learning
MSE	Mean Squared Error
ODCP	Online Dirichlet Change Point
POMDP	Partially Observable MDP
RL	Reinforcement Learning
RL-CD	Reinforcement Learning with Context Detector
RUQL	Repeated Update Q-Learning
TTS	Two-threshold switching strategy

UVFA Universal Value Function Approximators

VIC Variational Intrinsic Control

1

Introduction

Contents

Reinforcement Learning (RL) has been the most popular method for sequential tasks in recent years, enabling agents to learn by interacting with dynamic environments through trial and error. While navigating these environments, agents are guided using a reward system that incites agents to develop strategies to maximize long-term benefits. RL has notably mastered the game of Go [1] and performed better than humans in multiple Atari games [2]. It has also shown success in other fields, such as robotics [3] and autonomous driving [4].

However, RL has several known problems. Firstly, it is very data-inefficient. Even though it can achieve superhuman performance on many problems, it requires a huge amount of data to reach a very basic level of skill. The data used by these methods is very time-consuming to generate, so this constitutes a bottleneck. On top of that, in environments with sparse rewards, it is hard for the agent to find the few rewarding states and learn what sequences of actions lead it there. The amount of experience necessary is unattainable. Secondly, RL tends to overfit to each specific task. Consequently, transferring this knowledge to different tasks fails most of the time, even if the tasks are similar. Finally, when scaling to larger and more complex environments, the number of states and actions needed to represent them increases and pure-RL methods become ineffective, because it becomes impossible to visit every state regularly.

Furthermore, most of the problems we want to solve in the real world do not have stationary environments. In non-stationary environments, it is no longer possible to find one single optimal policy, since the rules of the environment are ever-changing. This causes data-efficiency and transferability of previously learnt tasks to be even more important. To limit the way the environment can change, we will consider that environments alter between a finite number of different modes [5], as this is an abstraction that translates well to the real world. Besides, we will focus on the changing of transition probabilities, because we will be working with sparse-reward environments, that only reward the agent when the goal is reached, and we expect the main goals to remain the same. Every time the environment changes mode, RL methods need a big amount of data to adapt and have trouble reusing what they had already learnt. Classical RL is unable to cope with these difficulties.

Due to these reasons, Hierarchical Reinforcement Learning (HRL) was introduced [6, 7]. This framework takes inspiration in the way humans intuitively solve problems. In recipes, for example, the main task ”baking a cake” is divided into several steps like ”mix 500g of flour”, ”pre-heat the oven” which can themselves be divided into even smaller tasks. Therefore, having several layers that deal with sub-problems of different granularity helps us deal with the obstacles latent in traditional RL. Complex tasks can be simplified into easier problems, reducing the amount of data needed to solve them [8] and mitigating the increase of the state space [9]. The sub-tasks learned by the lower layers may possibly be reused in different problems within the same environment [10], and encode meaningful skills [11]. In this dissertation, we will explore how this modularization may help in dealing with non-stationarity. We

investigate if it is possible to create abstraction between layers, allowing some layers to be oblivious to the changes in the environment. We address the following research question:

Can we leverage the benefits of hierarchical architectures to immunize our model against changes in modes of operation by isolating non-stationarity?

1.1 Contributions

In this work, we introduce a Contextual Hierarchical Actor Critic (CoHAC), which combines the Hierarchical Actor Critic (HAC) algorithm, a state-of-the-art HRL method introduced by Levy *et al.* [12], with our novel context detector.

CoHAC is designed to handle high-dimensional inputs and learn efficiently in sparse-reward environments. Additionally, it is designed to adapt smoothly to environmental changes. Once the model has been exposed to all possible modes of operation, it should maintain stable performance, even as the mode shifts. This ensures that the model converges to a stable solution rather than continually adapting with each change in the environment's mode. Finally, our goal is to learn general strategies that abstract away the specifics of the environment's mode, so that only a small component of the model is responsible for managing non-stationarity, while the broader strategy remains consistent.

To test this, we introduce non-stationarity into control tasks with continuous state-action spaces, which are commonly used in hierarchical reinforcement learning. We evaluate CoHAC against various baselines and conduct an in-depth analysis of the context detector's performance, validating several key design choices.

1.2 Outline

This document is structured as follows: in chapter 2, we present a formalization of RL and discuss standard approaches to solving RL problems. Chapter 3 surveys the state-of-the-art, covering both methods for handling non-stationarity and the most successful HRL architectures. Chapter 4 describes the implementation of CoHAC, followed by a comprehensive experimental evaluation in chapter 5. Finally, we summarize the key takeaways in chapter 6.

2

Background

Contents

2.1 Markov Decision Processes

In this section, we introduce the primary reinforcement learning framework, the Markov Decision Process (MDP). In this framework, an agent interacts with the environment in discrete time steps. In each time step, the agent observes the state of the environment, chooses an action to execute and then the environment outputs a reward and transitions to the next state, and to the next time step. Typically, the environment is split in episodes that end when a certain number of time steps is reached or when a goal state is achieved. When an episode ends, the environment is reset to an initial state and interaction between agent and environment resumes. Formally, an MDP is defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma)$:

- $\mathcal{S}$ is the set of states of the environment. The state-space can be discrete or continuous;
- $\mathcal{A}$ is the set of actions from which the agent chooses, it can be discrete or continuous;
- $\mathcal{P}$ is a function that represents the transition probabilities of the environment, $\mathcal{P}(s' \mid a, s)$ is the probability of transitioning from state s to state s' given that the agent performed action a;
- r is a function that represents the rewards of the environment, $r(s, a)$ is the reward obtained by the agent by performing action a in state s;
- $\gamma \in [0, 1)$ is the discount, which is used to shrink the reward as time advances and as a result encourages the agent to give more importance to short-term rewards.

The objective of the agent is to learn a *Markovian* policy π that maximizes the reward obtained. The *Markovian* property dictates that the action taken by the agent may only depend on the last state meaning $\pi(a_t \mid h_t) = \pi(a_t \mid s_t)$, where h_t is the entire history up to this point, $(s_0, a_0, s_1, a_1, ..., s_{t-1}, a_{t-1}, s_t)$. We define the state-value function, V^π, which represents the value of following policy π in state s:

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi \left[r_{t+1} + \lambda r_{t+2} + \lambda^2 r_{t+3} + ... \mid s_t = s \right] \\ &= \sum_{a \in \mathcal{A}} \pi(a \mid s) \left[r(s, a) + \lambda \sum_{s'} \mathcal{P}(s' \mid s, a) V^\pi(s') \right] \end{aligned} \tag{2.1}$$

Similarly, we can define value functions for state-value pairs instead of states, to which we call Q functions.

$$Q^\pi(s, a) = r(s, a) + \lambda \sum_{s'} \mathcal{P}(s' \mid s, a) \sum_{a' \in \mathcal{A}} \pi(a' \mid s')\, Q(s', a') \tag{2.2}$$

To maximize its obtained reward, the agent must discover the optimal policy. Essentially, this policy is the one that chooses the action with the highest value in each state. Thus, we define both the optimal Q function and we derive the optimal policy from it. Eq. 2.3 is a Bellman equation and will be very useful in the temporal difference methods we introduce later.

$$Q^*(s,a) = r(s,a) + \lambda \sum_{s'} \mathcal{P}(s' \mid s,a) \max_{a' \in \mathcal{A}} Q^*(s',a') \tag{2.3}$$

$$\pi^*(s) = \arg\max_{a \in \mathcal{A}} Q^*(s,a) \tag{2.4}$$

Eq. 2.3 allows us to calculate the optimal policy, however, it requires the reward and the transition functions. In most problems, environments function as a black box. The agent can interact with it, inputting actions and receiving new states and rewards from the environment, but the internal functioning of the environment, specifically, reward and transition functions, are not known. Hence, we need a different approach for calculating the optimal policy. Three families of methods exist to address this.

2.2 Model-based methods

The first family is called model-based methods. In this class of methods, the agent interacts with the environment and tries to approximate the transition and reward functions through an average. The agent then uses these averages along with a Bellman equation to compute the optimal Q values, or the optimal state-value function and later the optimal policy. Model-based methods are not present in the HRL architecture, so we do not delve into them.

2.3 Value-based methods

The second family of methods is called value-based methods. In these methods, the optimal Q function is computed directly through the interaction with the environment, and the optimal policy is derived from the Q function. To calculate the optimal Q function directly, we need stochastic approximation. Stochastic approximation is a huge family of algorithms that approximates the values of unknown functions, that we can only query. For example, if we want to find the zero of a function H, $\mathbb{E}[H(\theta)] = 0$, we query the function and take steps according to how far we are from the value desired, $\theta_{n+1} = \theta_n + \alpha_n(H(\theta_n))$, where θ_n is the previous estimate, α is the step size and $H(\theta_n)$ is the function's returned value. Discovering the optimal Q function consists of solving the following equation:

$$\begin{aligned} Q^*(s,a) &= \mathbb{E}\left[r(s,a) + \lambda \max_{a' \in \mathcal{A}} Q^*(s',a') \right] \\ \equiv \mathbb{E}&\left[r(s,a) + \lambda \max_{a' \in \mathcal{A}} Q^*(s',a') - Q^*(s,a) \right] = 0 \end{aligned} \tag{2.5}$$

If we treat $r(s,a) + \lambda \max_{a' \in \mathcal{A}} Q^*(s',a') - Q^*(s,a)$ as our function we get that Q^* is the zero of H. By using the stochastic approximation algorithm, we get the following update rule:

$$Q_{t+1}(s_t,a_t) = Q_t(s_t,a_t) + \alpha_t \left[r_t + \lambda \max_{a' \in \mathcal{A}} Q(s_{t+1},a') \right] \tag{2.6}$$

This algorithm is called Q-learning and it is the most widely used through RL. It is also important to note that it does not depend on the policy being followed, it always approximates the optimal Q function, thus, it is considered an off-policy algorithm. If you replace $\max_{a' \in \mathcal{A}} Q(s_{t+1},a')$ with $Q(s_{t+1},a_{t+1})$ you get an off-policy algorithm named SARSA, which in turn learns the Q values of the policy being followed.

2.4 Policy-based methods

The last family of reinforcement learning methods is called policy-based methods. In these methods, the objective is to approximate the policy directly instead of trying to approximate the value functions. To do this, the algorithm must search in the set of possible policies and compare them to discover the best one. Nevertheless, the entire set of possible policies is far too big to be searched, so we reduce it to the set of parameterizable policies, π_θ. The second step is to be able to distinguish the better policy from two non-optimal policies. With this in mind, we define the notion of value of a policy, $J(\theta)$. Between two policies, the one with the higher value is the one that the agent would prefer to follow. To travel in the direction of the best parameterizable policy, we can use another algorithm inside the class of stochastic approximation, called gradient descent. In this algorithm, the objective is to discover a maximum or a minimum. Since the gradient determines if the function is growing or decreasing, we can move in the direction of the gradient if we want to discover a maximum, and in the contrary direction to discover a minimum. Firstly, let's assume without loss of generality that we start at a random state s_0, and we can define the value of a policy as the state-value of the initial state (throughout this section we will refer to policy π_θ as π for simplicity):

$$J(\theta) = V_\pi(s_0) = \sum_a \pi(a \mid s_0)\, Q^\pi(s_0,a) \tag{2.7}$$

Next, by calculating the gradient of $J(\theta)$ w.r.t. θ, we get the policy gradient theorem [13].

$$\nabla_\theta J(\theta) = \sum_s \mu_\theta(s) \sum_a Q^\pi(s,a)\, \nabla_\theta \pi(a \mid s), \tag{2.8}$$

where μ_θ is a long-term distribution over the state-space, upon following policy π_θ, so the policy gradient is a sum over states weighted by this distribution. If we follow the policy π_θ, we can expect to find the

states in the same proportions. The following equality already allows to define an update rule.

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[\sum_a Q^\pi(s_t, a)\ \nabla_\theta \pi(a \mid s_t) \right] \tag{2.9}$$

$$\theta \leftarrow \theta + \alpha \sum_a \hat{Q}^\pi(s_t, a)\ \nabla_\theta \pi(a \mid s_t) \tag{2.10}$$

Notice $\hat{Q}^\pi$ is an approximation of the state-action value function that would still need to be defined. Besides, in this update every action would need to be evaluated at every time step. This is called an all-actions method. Nonetheless, by replacing the weighted sum over all actions with an expectation under π_θ, just like we did when replacing the weighted sum of states with s_t, we can get an update rule that is only dependent on a_t. Furthermore, we can easily confirm that the size of each update step is proportional to the quality of the state-action visited and inversely proportional to the probability of choosing action a_t. In other words, high rewards mean bigger updates and actions that are infrequently chosen are also updated more significantly.

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_\pi \left[\sum_a Q^\pi(s_t, a)\ \nabla_\theta \pi(a \mid s_t) \right] \text{ (multiply and divide by } \pi(a \mid s_t) \text{)} \\
&= \mathbb{E}_\pi \left[\sum_a \pi(a \mid s_t)\ Q^\pi(s_t, a)\ \frac{\nabla_\theta \pi(a \mid s_t)}{\pi(a \mid s_t)} \right] \\
&= \mathbb{E}_\pi \left[Q^\pi(s_t, a)\ \frac{\nabla_\theta \pi(a_t \mid s_t)}{\pi(a_t \mid s_t)} \right]
\end{aligned} \tag{2.11}$$

$$\theta \leftarrow \theta + \alpha\ \hat{Q}^\pi(s_t, a_t)\ \frac{\nabla_\theta \pi(a_t \mid s_t)}{\pi(a_t \mid s_t)} \tag{2.12}$$

Finally, to get a complete policy-gradient method, it is necessary to define how to approximate the state-action value function, $\hat{Q}^\pi(s_t, a_t)$. REINFORCE [14] solves this problem by sampling a full episode before performing any updates, and by considering that the Q value at each time step is simply the reward obtained until the end of the episode. At the end of each episode, REINFORCE performs a loop for each step of the episode $t = 0, 1, ..., T-1$ and performs the following update. The two differences from the last update consist in replacing $\hat{Q}^\pi(s_t, a_t)$ with G, which represents the reward obtained until the end of the episode and replacing the quotient between gradient and the probability of the action

being chosen with a logarithm simply because $\nabla \ln x = \frac{\nabla x}{x}$:

$$G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-(t+1)} \, R_k \tag{2.13}$$

$$\theta \leftarrow \theta + \alpha \, \gamma^t \, G \, \nabla_\theta \ln \pi(a \mid s_t) \tag{2.14}$$

The downside of REINFORCE is that, by collecting samples of full episodes, it is hard to identify which actions contributed to the success or failure of the agent. Simply, an episode where the agent achieves its goal does not mean that all the actions the agent took were beneficial, and vice-versa, which slows down learning.

To counteract this, a family of methods called actor-critic methods emerged. This type of methods can be seen as a combination of policy-based and value-based methods [15]. There is an actor with a parameterized policy that is updated using gradient ascent, whose function is to choose which actions to perform and this actor's performance is evaluated by a critic through Q-values. The critic uses value-based methods (typically Q-learning) to estimate the state-action value function and informs the actor of the success of the actions chosen. This way, actor-critic methods leverage the better convergence guarantees that policy-based methods (actor-only methods) offer and their ability to deal with continuous action spaces, while value-based methods (critic-only methods) help in achieving gradients with lower variance. Algorithm 2.1 presents the pseudo-code of a simple actor-critic algorithm.

Algorithm 2.1 Q Actor Critic

Initialize parameters θ, w and learning rates α_θ, α_w
Sample s from environment and $a \sim \pi_\theta(a \mid s)$
for $t = 1...T$ **do**
  Sample reward r_t and next state s' from environment
  Sample next action $a' \sim \pi_\theta(a' \mid s')$
  Policy update: $\theta \leftarrow \theta + \alpha_\theta \, Q_w(s,a) \, \nabla_\theta \ln \pi_\theta(a \mid s)$
  TD error: $\delta_t \leftarrow r_t + \gamma \, Q_w(s',a') - Q_w(s,a)$
  Critic update: $w \leftarrow w + \alpha_w \, \delta_t \, \nabla_w Q_w(s,a)$
  $a \leftarrow a'$ and $s \leftarrow s'$
end for

The policy learnt by the actor-critic methods described above is stochastic, which means it outputs a probability of choosing each action a, instead of selecting a single action. This helps with exploration, since the action that the policy prefers is not guaranteed to be chosen and other actions might be explored. On the other hand, in order to compute the policy gradient, we must integrate over both the state and action space, which might be less sample-efficient than integrating over just the state space, especially if the action space is high-dimensional. Silver *et al.* [16] present a deterministic policy gradient, with minimal differences from the stochastic policy gradient, and consequently derive an off-policy actor-critic that learns a deterministic target policy ($\mu_\theta(s) = a$ instead of $\pi_\theta(a \mid s)$). The algorithm is considered

off-policy because, to ensure reasonable exploration, the agent follows a stochastic exploration policy, for example an ϵ-greedy policy, that chooses the greedy action with probability 1 - ϵ and chooses a random action with probability ϵ.

The deterministic policy gradient is derived similarly to the stochastic policy gradient. In equation 2.7, we had to sum across all actions because every action had a probability to be chosen. With a deterministic policy, $J(\theta)$ only depends on the state-action value of the initial state and the corresponding action, s_0 and $\mu_\theta(s_0)$. By computing the gradient and using the chain rule, we get the Deterministic Policy Gradient. We will omit the update rules since they are obtained the same way. As before, μ_θ will be shortened to μ.

$$J(\theta) = Q^\mu(s_0, \mu(s_0)) \tag{2.15}$$

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathbb{E}_\mu \Big[Q^\mu(s, \mu(s)) \Big] \\ &= \mathbb{E}_\mu \Big[\nabla_\theta \mu(s) \nabla_a Q^\mu(s, a) \Big|_{a=\mu(s)} \Big] \end{aligned} \tag{2.16}$$

3

Related Work

Contents

In this chapter, we perform a thorough description of the current literature divided in two main parts: approaches that are non hierarchical capable of dealing with non-stationarity and approaches that are hierarchical but only consider stationary environments.

3.1 Non-stationarity

We consider environments that shift between a limited number of modes. It is possible to model this as a Partially Observable MDP (POMDP). In a POMDP, the agent does not have access to the current state, only to an observation, from which it attempts to derive the current state. As a consequence, the agent maintains a distribution over states, called the belief. However, using a POMDP is too generic and loses the intrincacies of our specific non-stationary conditions. Therefore, Choi *et al.* [5] have defined a variation of an MDP, formalizing this notion. Hidden-mode MDPs are defined by a set of several MDP that share the same state space, but differ in the transition probabilities and the rewards. Each MDP corresponds to a specific mode of the environment, and transitions between modes happen according to a stochastic function independent of the current state and the action chosen by the agent. The authors define a set of properties:

1. Environmental changes are bounded to a small number of modes.

2. Modes cannot be observed by the agent, they can only be estimated through recent history.

3. Mode changes are not influenced by the agent's actions.

4. Mode changes are infrequent.

5. The number of states is much larger than the number of nodes.

We also consider these properties when presenting the different algorithms. Furthermore, the authors propose an adaptation of the algorithm used to solve a POMDP, the Baum-Welch algorithm, to solve a Hidden-mode MDP. This leads to better performance than using the original Baum-Welch algorithm in the corresponding POMDP, but other approaches have largely improved upon these results. Amongst these approaches, we can separate two main groups, distinguished by the need to learn the model of the Hidden-mode MDP. The first group seeks to learn the transition probabilities and rewards of each mode, then uses this information along with recent history to estimate the current mode and act accordingly. On the other hand, model-free approaches do not need to keep an estimation of the mode to determine the best action. We review these two classes of methods in detail in the continuation.

3.1.1 Model-based approaches

In general, model-based approaches learn transition probabilities and the rewards of each mode. Then, the module that better predicts the observed transitions and rewards is chosen as the active module and the policy associated with it is used. Each policy can be learnt with any off-the-shelf reinforcement learning algorithm, although model-based algorithms are normally favoured in order to take advantage of the already learnt environment model. In spite of that, the flexibility is useful because any HRL algorithm can be seamlessly implanted.

Multiple Model-Based Reinforcement Learning (MMRL) [17] was the first proposal of this sort. Coordination between models is done through a responsibility signal, that controls the weight of each model's suggested action. The responsibility signal is based on the controller's ability to predict the next state. The goal is for each module to be responsible for solving a part of the system and the weights of the votes correspond to the probability of being in each mode of the environment.

The responsibility signal formula is based on the Bayes theorem. It takes into account each module's ability to predict the next state ($\mathcal{P}(x(t) \mid m)$) and the prior belief of the active mode, represented in responsibility predictors $\hat{\lambda}_m$:

$$\lambda_m(t) = \mathcal{P}(m \mid x(t)) = \frac{\hat{\lambda}_m * \mathcal{P}(x(t) \mid m)}{\sum_{j=1}^{n} \hat{\lambda}_j * \mathcal{P}(x(t) \mid j)},$$

where $x(t)$ is the next state. The probability of the state according to each model is given by the state predictor function T, $\mathcal{P}(y(t) \mid m) = T_m(x(t+1), x(t), a(t))$. To avoid frequent switches between models, the authors add temporal continuity, $\lambda_i(t) = \prod_{k=1}^{t} \lambda_i(t-k)^{\alpha}$, where α controls the impact of past signals.

To implement this architecture, each module keeps a reward prediction model, $R_m(x_t, a_t)$, and a state prediction model, $T_m(x_{t+1}, x_t, a)$, along with a value function for each state. The authors use a temporal difference method to update the value function and approximate the reward and state model using the following update rules:

- Value function: $\delta_i = r_t + \lambda V(x_{t+1}) - V(x_t)$,
- Reward model: $\Delta R_m = r_t - R_m(x_t, a_t)$,
- State model: $\Delta T_m = \mathbf{1}_{x_{t+1}=y} - T_m(y, x_t, a)$.

The state model update rule considers discrete states and adapting them to the continuous case would require using the distance between the predicted and the observed state. By scaling the learning rate according to the responsibility signal, each module is encouraged to learn the dynamics of the mode of the environment it better understands. Finally, each module uses its three functions to output an action, which is then weighted according to the responsibility signal.

The state and reward prediction models are implemented using linear function approximators, because the authors claim that the freedom of non-linear approximators leads to a single model solving the entire task. The use of regularizers may prevent the domination of a single model over the others, allowing us to take advantage of the expressive power of non-linear functions approximators, like deep neural networks. MMRL, although it has been outperformed by the most recent proposals, captures the main principles of model-based methods to deal with non-stationarity.

Reinforcement Learning with Context Detector (RL-CD) [18] is a slightly more practical approach. The main difference to MMRL is the absence of the responsibility signals. Each module also computes ΔR_m and ΔT_m, but instead uses them to calculate an absolute value measuring the quality of prediction of the reward model, $e_m^R = 1 - 2(Z_R\ (\Delta R_m)^2)$ and the transition model, $e_m^T = 1 - 2(Z_T\ \sum_{k \in S} \Delta T_m(k)^2$, with Z_R and Z_T being normalization factors. A linear combination of e_m^R and e_m^T is used to discover the instantaneous quality of a module. Temporal continuity is also implemented by limiting the impact of the most recent instantaneous result on the overall quality. The overall quality of a model moves in small steps (ρ) towards the instantaneous score, $E_m = E_m + \rho\ (e_m - E_m)$. While MMRL uses a weighted average, RL-CD only has a single active module fully in control.

Another way of selecting the active model that achieves better results was proposed by Hadoux *et al.* [19], by utilizing a statistic-based approach called CUSUM to detect abrupt changes [20]. Instead of calculating an absolute value, the quality of the modules is calculated by comparing all modules to the current module M:

$$E_m = max_m \left(0, E_m + ln \left(\frac{T_m(x(t+1), x(t), a(t))\ R_m(x(t), a(t), r(t)}{T_M(x(t+1), x(t), a(t))\ R_M(x(t), a(t), r(t)}\right)\right)$$

The quality of every module is updated in every time step, while transition and reward prediction functions are only updated for the active model. When a model becomes better than the current one, it becomes active. If no model is better than a given baseline, then a new model is created and it becomes the active model. In the relative approach [19], a neutral module is kept with initial transition and reward functions, whose activation is equivalent to creating a new model. This characteristic allows RL-CD to work without knowing the number of modes of the environment, which helps our algorithm to operate with as little domain knowledge as possible.

RL-CD learns each model quicker than MMRL because only one model is active at each time. However, RL-CD takes longer to detect context changes and its selection mechanism is more susceptible to noise. Besides, the threshold that dictates when a new model is created is hand-tuned and, as a consequence, it is not very reliable. Hadoux *et al.* [19] give more meaning to this threshold by introducing two hyper parameters, α and β, that tune the probability of false changes and undetected changes, respectively: $c = ln\ \frac{1-\beta}{\alpha}$. However, this threshold still remains quite arbitrary.

The last model-based approach is called the Two-threshold switching strategy (TTS) [21] and it deals

with the uncertainty of identifying a change point. Like the improved RL-CD, it also compares candidate modules to the active one by calculating the cumulative sum of the divergence between contexts. The main novelty in this approach is that the authors take into account that the change detection may fail. It may happen that the actions chosen by the current policy don't emphasize the superiority of the candidate module. If undetected or false changes happen, some reward may be lost. To combat this, the authors introduce a two-threshold approach. Instead of changing the model when the score exceeds a threshold, an extra threshold is introduced. When this new threshold is surpassed, the agent starts following a policy, π_{KL}, that tries to emphasize the difference in modes in order to understand as quickly as possible if this is a false alarm or there has in fact been a change in mode.

If the second threshold is surpassed, then the agent switches to the next mode. The authors begin by considering the detection between two known modes. The generalization to multiple modes is not trivial, since the π_{KL} maximizes the difference between the current module and a single other module, and the choice of this module must be addressed. The authors choose to use a policy that maximizes the KL divergence between the current model and the model where the divergence is currently lower.

TTS is the first proposal to consider the trade-off between change detection and reward maximization. On the other hand, thresholds are still arbitrary and, most importantly, the discovery of the policy π_{KL} for environments with a larger state space is unaddressed. Especially in model-free algorithms (which compose most of the Hierarchical RL literature), discovering this policy for every pair of modes would be complex and expensive.

3.1.2 Model-free approaches

Most of the hierarchical reinforcement learning literature is composed with model-free approaches. Model-based approaches build a model in order to deal with non-stationarity, and this model would not be used by the underlying HRL algorithm. As a result, even though model-based algorithms to detect non-stationarity are compatible with model-free HRL algorithms, model-free approaches for both problems would be a better fit.

Context-QL [22] adapts an Online Dirichlet Change Point (ODCP) [23] to detect change points using data points (s_t, r_t, s_{t+1}) from the experience of the agent. These change points correspond to mode changes but ODCP is unable to identify the new mode. Consequently, Context-QL needs to know the number of environment modes as well as the order in which they appear. When a module is active, any off-the-shelf reinforcement learning algorithm can be learnt to determine the best policy. Context-QL has more success identifying context changes than the previous approaches, but we do not want our algorithm to need the order in which modes appear, since this cannot be obtained in certain environments. For example, in a locomotion task where wind speed and direction change, the next mode is unknown. This renders this proposal very limiting.

One of the problems with Q-learning in non-stationary environments is policy bias. The value of an action is only updated when it is executed, consequently, the rate at which the value of an action is updated depends on the probability of that action being chosen. This is particularly bad in non-stationary environments because upon a change in the mode of the environment, previously optimal actions may be outperformed. Traditional Q-learning continues with a tendency to choose this currently sub-optimal action until it recognises the new optimal one. In a perfect world, for every time step, all actions would be updated and this way all Q-values would be updated w.r.t the current mode of the environment. There are two main methods to try to emulate this by updating the value of an action proportionally to the inverse of the probability of choosing that action.

Frequency Adjusted Q-Learning (FAQL) [24] does exactly this, scaling the learning rate inversely to the probability of choosing the action.

$$Q_{t+1}(s,a) = Q_t(s,a) + \frac{1}{\pi(a \mid s)}\ \alpha\ \left(r + \gamma\ \max_{a'} Q_t(s',a') - Q_t(s,a)\right)$$

The problem with this naive implementation is that as $\pi(a \mid s)$ approaches values close to 0, the learning rate tends to infinity, so a safeguard condition needs to be added.

$$Q_{t+1}(s,a) = Q_t(s,a) + min\left(1, \frac{\beta}{\pi(a \mid s)}\right)\ \alpha\ \left(r + \gamma\ \max_{a'} Q_t(s',a') - Q_t(s,a)\right)$$

This safeguard condition leads to an unfortunate consequence. For actions with probability inferior to beta, updates return to the original Q-learning. We can try to build a policy with a lower bound of the probability of an action, but Repeated Updated Q-Learning [25] solves this problem completely.

As the name suggests, instead of scaling the learning rate, Repeated Update Q-Learning (RUQL) [25] repeats the update a number of times proportional to the inverse of the probability of choosing that action. In a naive approach, a *for* cycle could be used to repeat the update $\lfloor \frac{1}{\pi(s,a)} \rfloor$ times. However, we reencounter the problem from FAQL: a very low probability for choosing an action would cause an update to be repeated an infinite number of times, and this would be computationally unfeasible. The authors proof that, using the following step size, $z = 1 - [1-\alpha]^{\frac{1}{\pi(s,a)}}$, an approximate result can be obtained with an error of $O(\alpha^2)$. This approximate version even leads to closer Q-values in certain examples as the floor approximation is no longer necessary.

The results show that using RUQL accelerates the optimization of Q-functions after a change in the environment. Unfortunately, it is incapable of converging even in an environment with few modes. The result will be a single ever-changing Q-function incapable of using the experience of any modes outside of the most recent one. The agent reacts more quickly to changes, but it will never be able to converge, which we list as one of our objectives.

Table 3.1 presents the summary of the main algorithms for dealing with non-stationarity, specifying

Table 3.1: Summary of main approaches to deal with non-stationary environments

Algorithm	Model information	Main contributions	Main disadvantages
HM-MDP [5]	Model-based	Formalization of a non-stationary environment	Uses inefficient POMDP algorithm
MMRL [17]	Model-based	Coordination among multiple modules using a responsibility signal	Slow learning; Use of linear functions; Requires number of modes
RL-CD [18]	Model-based	Faster learning of each mode; Fewer parameters	Susceptible to noise; Arbitrary threshold
Relative RL-CD [19]	Model-based	Implementation of statistical methods to improve change detection	Susceptible to noise; Arbitrary threshold; Relative change point detection unsuitable for continuous state spaces
TTS [21]	Model-based	Optimization of the trade-off between change detection and reward maximization	Hard to implement in a multiple mode scenario; Relative change point detection unsuitable for continuous state spaces
Context-QL [22]	Model-free	Implementation of a new method (ODCP) to improve change detection	Requires number of modes and their change pattern
FAQL [24]	Model-free	Step-size modification to obtain more up-to-date Q-values	β-problem; Does not converge
RUQL [25]	Model-free	Solution to the β-problem	Does not converge

the model information requirements and the main contributions and disadvantages. Next, we will explore the evolution of the hierarchical reinforcement learning literature. None of the HRL's most relevant solutions are used with non-stationary environments, so we will evaluate each one without taking non-stationarity into account.

3.2 Options

The first step to hierarchical reinforcement learning was very natural. Humans solve problems with high-level instructions, like "turn left" and "walk 500 meters" and avoid the simplest commands, such as "contract your muscles". Similarly, reinforcement learning evolved to macro-actions, composed by the different atomic actions available to the agent. In turn, macro-actions originated a concept called temporal abstraction, since macro-actions take an unknown number of time steps to complete.

3.2.1 Semi-Markov Decision Processes

To deal with temporal abstraction, MDPs were no longer sufficient and Semi-Markov Decision Processes (SMDP) were introduced. In this new framework, an action may take several time steps to execute. This is called the transit time and is represented by τ.

When an action is being executed, there is dependency on the number of time steps that have elapsed. However, upon reaching a state, only that state and the action are necessary to calculate the next state and that is why it is described as Semi-Markov. To define an SMDP, we need four components $< \mathcal{S}, \mathcal{A}, \mathcal{P}, r >$. $\mathcal{S}$ represents the states and $\mathcal{A}$ represents the actions, analogously to MDPs. On the other hand, transition probabilities now also include τ, $\mathcal{P}(s', \tau \mid s, a)$ and rewards now depend on τ, $\mathbb{E}[r \mid s, a, s', \tau]$.

Now that we have a way of incorporating temporally extended actions into MDPs, we can begin to formalize them. Sutton *et al.* [26] introduced them as options. Options are defined by three components:

- Initiation set $\mathcal{I} \in \mathcal{S}$: the set of states where the option might be selected;
- Policy $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$, which is the policy being followed while the option is being executed;
- Termination probability $\beta : \mathcal{S} \to [0, 1]$, which is the probability of an option terminating in each state. Normally, $\forall s \notin \mathcal{I} : \beta = 1$ and, as a result, we only need to define a policy in states that belong to the initiation set.

The options described are Markov options, where the policy and termination probability only depend on the current state. They can also depend on the history since the option was started and the options become Semi-Markov. Notice that primitive actions can be described as an option, which allows us to use the same methods used for MDPs. We can define a policy $\mu : \mathcal{S} \times \mathcal{O} \to [0, 1]$ over options. This policy implicitly defines a flat policy over conventional actions, but as long as we have temporally extended options, even if Markov, this flat policy will be Semi-Markov. Since options can be seen as an abstraction of conventional actions, traditional Q-learning still applies,

$$Q(s, o) = Q(s, o) + \alpha \left[r + \lambda^k \max_{o' \in \mathcal{O}_{s'}} Q(s', o') - Q(s, o) \right] \tag{3.1}$$

where k is the number of time steps the option took and r is the reward accumulated throughout the execution. However, treating options as indivisible units is very limiting. We need to follow an option until it naturally terminates, even if stopping and executing a different option was more beneficial. On top of that, for each option we execute, only one Q-value is being updated, which is very inefficient.

3.2.2 Intra-option Q-learning

In intra-option Q-learning, we take advantage of the fact that options are a composition of primitive actions, so we take the sequence $h = (s_t, a_t, r_t, s_{t+1}, a_{t+1}, r_{t+1}, ...)$ to update as many Q-values as possible. For instance, we do not need to wait for the option to terminate to make an update:

$$Q(s_t, o) = Q(s_t, o) + \alpha \left[r + \lambda U(s_{t+1}, o) - Q(s_t, o) \right], \tag{3.2}$$

where U depends if the option has finished in s_{t+1}:

$$U(s, o) = (1 - \beta(s))\ Q(s, o) + \beta(s) \max_{o' \in \mathcal{O}_s} Q(s, o') \tag{3.3}$$

In addition, if the options are Markov, it is not important in which state the option has started. For this reason, we can also update $Q(s_{t+1}, o)$, $Q(s_{t+2}, o)$, etc... If $\pi_{o'}(s_t) = \pi_o(s_t) = a_t$, we can also update $Q(s_t, o')$ because even the option being followed to obtain h is irrelevant. Finally, we can design the β function to end all options when the Q-function determines it is more beneficial to follow a different option. These new off-policy learning methods make options more data-efficient and a relevant framework for HRL. Nevertheless, Sutton *et al.* [26] uses pre-determined options to learn to achieve the general goal and our model should be able to learn multiple tasks with minimal adaptation. Handcrafting options goes against this.

3.2.3 Supervised option learning

The first approach to generate options was to generate random options or options based on simple heuristics. By adding them to the agent's action space, those options would be tried and the ones that achieved the best results would remain. Some of the options found using this approach are good, however, most of them are purposeless and can inclusively deteriorate exploration and slow down learning by growing the action space. Quickly, finding the states that are frequently visited in successful runs proved to be a more efficient approach. However, if we simply use a frequentist method, these states will only be discovered too late in the learning process for options to have a positive impact. Several papers improve on it.

Menache *et al.* [27] proposed using a Min-Cut, which consists on finding a cut of a graph containing the edges that have the least capacity. The vertexes around these edges should be visited in most trajectories. Consequently, learning to reach them might be useful. To identify them, states and their transition probabilities are stored during training. When it is appropriate to make a cut, the environment is turned into a graph where nodes are states, edges are transitions and capacities are transition probabilities and a min-cut is performed. If the cut is classified as good, the states around the cut become

goals to new options.

Meanwhile, McGovern and Barto [28] use the concept of multiple-instance learning problems [29] which introduce bags. A negative bag is a set of only negative instances and a positive bag is a set with at least one positive instance. In the RL domain, they treat successful trajectories as a positive bag and unsuccessful trajectories as negative bags. The success, although problem-dependent, is usually if the trajectory ends in a goal state. The instances are the states that compose each trajectory. After collecting a few trajectories, the goal is to find the most diversely dense region. Diverse density is defined as the region that appears on most successful trajectories and rarely on unsuccessful trajectories. They discover the diverse density of each region using the Bayes' rule and find the peaks using gradient descent. The states associated with successful trajectories become goals to new options.

This type of approaches share the same problems. It is only possible to learn options after achieving the goal, meaning these methods can't be applied in problems where a large sequence of primitive action is needed to find it in the first place. Besides, with larger state spaces, the complexity of this type of algorithms increases at a rapid pace and most of them can't be extended to cases where the state space is continuous.

With the explosion of the state and the action space, it becomes clear that it is impossible to store every individual value of $Q(s, a)$, so function approximators are necessary. Tsitsiklis *et al.* [30] demonstrated that the convergence of TD-methods is dependent on the linearity of function approximators. Unfortunately, this means that using Deep Neural Networks (non-linear function approximators) to represent the Q function does not guarantee convergence, and throughout many years the approaches tried had problems with stability. However, Mnih *et al.* [2] created a model with two simple innovations that assured the Q-values converged experimentally. The first innovation was storing interactions with the environment (s_t, a_t, r_t, s_{t+1}) in a replay buffer [31]. When an update is due, a mini-batch is randomly chosen from all the interactions collected. This is a major improvement from sequential interactions used in on-policy training. Firstly, less interactions with the environment are needed since samples can be reused. Also, since the agent receives samples from a large interval of time, temporal correlation between samples that can harm the agent's learning is eliminated. The second improvement consists of the introduction of target networks. Target networks are updated at a slower rhythm than the learned Q-value networks so that the Q function is not trying to hit a moving target. This way, we get a reduced correlation between the learned Q function and the target Q function and the Q-network update step's baseline is more stable, which means the Q function does not harm its own learning with the instability of updates. Although Deep Q-Network (DQN) [2] does not offer any theoretical convergence guarantees, it is able to outperform humans in a very large number of ATARI games using only raw input from the screen.

This proposal allowed neural networks to be used in a reinforcement learning setting, and, conse-

quently, in hierarchical reinforcement learning. As a result, Kulkarni *et al.* developed a hierarchical Deep Q-Network (h-DQN) [9]. Their approach is also based in the framework of options, which in this case are called sub-goals. There is a top-level meta-controller that chooses from a pre-defined set of sub-goals and assigns them to the controller. This controller selects actions until the goal is reached or until the episode is finished. We could include h-DQN in the framework we will discuss later, but since the possible sub-goals are limited to a few states, they are better defined as options. The model is trained with stochastic gradient descent, exactly as was done in [2]. The h-DQN architecture shows that assigning relevant goals leads to a much better exploration, which makes this model much more data efficient and able to succeed in environments with sparse rewards, like Montezuma's Revenge.

The meta-controller and the controller have separate Q functions. The meta-controller's Q function pairs states and goals, and uses the cumulative reward received from the environment throughout the execution of the goal to calculate the loss function. The controller's Q function pairs states and primitive actions and uses the intrinsic reward (intrinsic motivation) to calculate the loss function. The two Q functions operate at completely different time scales: the meta-controller sees the full execution of a goal as a single time step. Meanwhile, training is done in two phases. In a first phase, the exploration parameter ϵ of the meta-controller is set to 1, which means every possible goal is chosen with equal probability. The objective of this is training the lower-level on achieving them, ignoring the task's objective. Afterwards, the meta-controller and the controller are jointly trained, now considering the extrinsic reward. Having two phases throughout learning is a strategy commonly used, since it allows the learning of general skills, oblivious to the way they are going to be used. However, the authors still do not solve the problem of independently finding useful goals. For example in the Montezuma's revenge, they use a custom pipeline to identify objects of interest (important streams of pixels) and use them as possible goals. We want an end-to-end approach that can be generalized to different problems without domain knowledge.

One of the first end-to-end approaches was the option-critic architecture [6]. On this two-level architecture, both intra-options policies and termination conditions are parameterized by θ and ϑ, respectively. This way, it is possible to discover options and learn them at the same time, by deriving the effect of θ and ϑ on the reward and using policy gradient techniques, such as the actor-critic framework [32]. The only input necessary becomes the number of options desired, and the actor-critic approximates both the parameters of the options and the parameters of the policy over options at the same time.

Options are not necessary to optimally solve an MDP. They provide better exploration and speed-up learning which is very useful, since computational power and time are very limited resources. However, options do not ensure any theoretical improvements. For this reason, the option-critic's policy gradient tends to learn options that terminate every time step, or a option that never terminates, essentially going back to primitive actions. To solve this problem, the authors suggest the use of regularizers and posterior work tried to implement this. Harb *et al.* [33] added a deliberation cost when changing options so that the

agent only abandoned the current option when there was other significantly better. This avoids options from shrinking and obtains better results in Atari games than option-critic. Although this one and other variations of the option-critic architecture are the current state-of-the-art in the options framework, they cannot currently compete with the empirical results of feudal hierarchies, which we are going to discuss next. Table 3.2 summarizes the methods visited so far.

Table 3.2: Summary of main approaches in the Options framework

Algorithm	Required priors	Main contributions	Main disadvantages
Sutton *et al.* [26]	Pre-determined options	Formalization of options; Better exploration; Better sample efficiency; Adaptation of previous learning methods to options	Need for handcrafted options
Q-cut [27]; Diverse density [28]	Number of options; Successful trajectories	First approaches to generating options automatically	Very high complexity; Unfeasible for continuous state-spaces; Options are not available in initial stages
h-DQN [9]	Pre-defined sub-goals	Good exploration in high dimensional state-spaces	Needs a pipeline for identifying sub-goals
Option-critic [6]	Number of options	First end-to-end approach	Fixed number of options; Performs poorly with sparse-rewards; Tends to using only primitive actions
Option-critic with deliberation cost [33]	Number of options	Fixes the degeneration problem present in OC	Fixed number of options; Performs poorly with sparse-rewards

3.2.4 Unsupervised option learning

All the methods aforementioned use the reward provided by the environment to discover options, along with a policy over options to solve the task at hand. There is another class of methods that try to learn options (now denominated skills) that increase the control of the agent over the environment, without considering the actual objective of the task.

The first proposal that uses this principle is pre-training using Stochastic Neural Networks (SNN4RL) [11]. It suggests that, before attempting the real task, agents are put in different environments with the same agent-space [34] and are encouraged to learn skills (instead of options) with a proxy reward. This reward is based on domain knowledge, it can be for example proportional to speed of movement in locomotion problems. Divergence in the skills learned is encouraged using a mutual information bonus.

After completing pre-training, a policy over the set of skills apprehended is learned. The policy-over-skills is a feed-forward neural network that returns a one-hot vector, dictating which skill to use over the next τ steps. All policies are learned using Trust Region Policy Optimization [35].

Skills lead to better exploration in environments with sparse rewards. The proxy reward allows useful skills to be found without reaching the problem's goal and they in turn lead to bigger steps while exploring the real problem. It is also clear that skills would be highly useful in problems in the same environment, because they were found independently of the downstream task to begin with. Nevertheless, when the high level policy is learning, the skills are inflexible, which may cause the agent to deviate from the optimal performance. We are looking for a better trade-off between transferability and optimization of the skills learned.

Similarly to SNN4RL, meta learning shared hierarchies [10] use several instances of the same task to learn shared sub-policies. It is called meta-learning because this method is focused on learning to learn. In other words, its objective is to discover the most useful sub-policies in all instances of the task, in order to accelerate learning of a new instance. To achieve this, sub-policies are shared through all instances, while the master policy is recalculated in each instance. This paper's main novelty is that it continues to reset the master policy.

Sub-policies are parameterized by ϕ and the master policy is parameterized by θ. To represent the different tasks, different MDP's with the same agent-space [34] are sampled for each episode. At a given time step, the master policy chooses a sub-policy to be carried out for a fixed N number of time steps. Then, the sub-policy chooses primitive actions according to its parameters. ϕ vectors are shared amongst the different episodes, contrary to θ. Each episode is split into two main phases: the warm-up phase, where only the master policy is updated, and the update phase, where both the master and sub-policies are updated. The updates can be done with any off-the-shelf reinforcement learning algorithm.

To update sub-policies, we need to assume that the master policy is an optimal policy. While this is impossible to obtain, the warm-up phase allows the master policy to be close enough to the optimal policy for the sub-policies to converge. If this was not the case, the sub-policies would tend to be similar, since they would be picked randomly by the master policy and more general sub-policies would be beneficial. On top of that, as the sub-policies change, the corresponding optimal master policy changes accordingly and π_θ drifts further from the optimal policy, π^*_θ. For this reason, θ is reset and retrained after a certain number of iterations. This approach does not require a specific reinforcement learning algorithm, so it can improve as better algorithms are discovered. The main problem is the need for similar environments with slight differences, which may not be obtainable for all problems.

Next, we will introduce several methods that utilize information theoretic objectives to learn options that maximize the agent's control over the environment. The first one of these methods is called

Variational Intrinsic Control (VIC) [36]. This algorithm may be described as learning a representation of the control-space of an agent. The objective is to learn options, diverse amongst themselves, that can largely affect the environment. Quantity is prioritized over quality, meaning there are a larger number of options being learnt than in previous methods. Each state must have several available options and each option should reliably reach different parts of the environment. This ability is called empowerment because the set of options described increases the agent's control over the environment. This technique of unsupervised control is parallel to unsupervised learning in classification. Data likelihood is to unsupervised learning as mutual information is to unsupervised control.

The goal is to maximize the set of options in each state, in other words, maximizing the entropy of $\mathcal{P}(\Omega \mid s_0)$, with Ω representing an option. On the other hand, to maximize control, we should generate options with final states as diverse as possible. Thus, we want to be able to tell options apart only from the final states they achieve, by minimizing the entropy of $\mathcal{P}(\Omega \mid s_0, s_f)$. We can define the mutual information between options and final states with $\mathcal{P}^C$ being the controller's distribution over options:

$$I(\Omega, s_f \mid s_0) = -\sum_{\Omega} \mathcal{P}^C(\Omega \mid s_0)\ log\ \mathcal{P}^C(\Omega \mid s_0) + \sum_{\Omega, s_f} \mathcal{P}(s_f \mid s_0, \Omega)\ \mathcal{P}^C(\Omega \mid s_0)\ log\ \mathcal{P}(\Omega \mid s_0, s_f) \tag{3.4}$$

In practice, $\mathcal{P}(\Omega \mid s_0, s_f)$ is inferred using an approximate function q. An option is chosen according to the current state and carried out until a termination state is reached. Afterwards, the q function is updated towards Ω, increasing the likelihood of the option used when knowing the initial state and final state of the specific iteration, $\mathcal{P}(\Omega \mid s_0, s_f)$. Then, an intrinsic reward is calculated. This intrinsic reward (r_I) is larger if the probability of choosing Ω in s_0 was low, meaning there were a lot of available options and if knowing the final state allows the inference of the chosen option with confidence, meaning the option chosen was unique.

$$r_I = log\ q(\Omega \mid s_0, s_f) - log\ \mathcal{P}^C(\Omega \mid s_0) \tag{3.5}$$

Any reinforcement learning algorithm can be used to update the option policy and the policy over options according to intrinsic reward.

This approach is simple and model-free. However, it has two main problems. It does not work well with non-linear function approximators due to the instability present before DQN [2]. The algorithm is not stable because the intrinsic reward is noisy and changing. Perhaps the implementation of DQN's proposals might improve the algorithm in this perspective. Additionally, when an agent encounters a new state, it should be encouraged to go there because learning how to get there can increase its control. Nevertheless, when arriving at a state for the first time, the q function has not been learnt so the algorithm has trouble inferring the option used, lowering the intrinsic reward, and discouraging the agent to come back to this new state. The authors suggest an alternative but we will not dive into it because

better methods have been proposed.

Diversity Is All You Need (DIYAN) [37] is a direct improvement over VIC. It also uses mutual information and entropy to learn skills as diverse and distinguishable as possible. However, DIYAN fixes the policy over skills. While VIC tends to quickly develop a small number of diverse skills and sample only this small set of skills as the policy skews in their favor, DIYAN keeps the policy stationary and learns many more diverse skills. Besides, DIYAN uses all states to update the skill policy and the q inference function and its objective function is quite compatible with the state-of-the-art in off-policy training, the Soft-Actor Critic [38].

Again, since these approaches ignore extrinsic reward, the options learnt are more general and can be more easily transferred between similar environments. On the other hand, trying to solve some tasks using only general options might complicate problems that need specific behaviors to be solved, so a trade-off between general and optimal skills should be explored. Besides, some of these methods depend on having several instances of the same problem, with slight differences amongst them. This may not be the case and our algorithm should work even with a single instance of the problem. In addition, all methods described are limited by a number of options, which limits the algorithms in flexibility and adaptability. Table 3.3 presents a summary of the methods approached in the framework of unsupervised option discovery.

Table 3.3: Summary of the main approaches that learn options/skills without extrinsic reward

Algorithm	Required priors	Main contributions	Main disadvantages
SNN4RL [11]	Proxy reward; Variations of the environment; Number of skills	Learning of transferable skills with high expressiveness	Requires too many priors; Fixed number of skills; Loss of performance with inflexible skills; High sample complexity
MLSH [10]	Variations of the environment; Number of sub-policies	Good generalization to new tasks; Good sample-efficiency; Simplicity	Limited to specific tasks; Fixed number of skills
VIC [36]	Number of options	Discovery of options by maximizing mutual information; Improved control over the environment	Fixed number of skills; Few options tend to overtake the entire problem; Avoids new states; Instability
DIAYN [37]	Number of options	Learns more diverse options than VIC	Fixed number of skills

3.3 Feudal Hierarchies

The second framework in hierarchical reinforcement learning, and the most important for the model we are going to design, is feudal hierarchies. It is inspired by the system in Medieval Europe, where lords leveraged control of their serfs to maintain their extensive land. In HRL, managers have complete control over sub-managers, assigning them goals. Higher level managers need to learn how to concatenate successive sub-goals to reach the problem's main goal while the lower level managers only need to learn how to satisfy the sub-goals received.

Dayan *et al.* [39] introduced reward hiding and information hiding. According to reward hiding, managers must attribute rewards based solely on the success of sub-managers at achieving their proposed sub-task, even if the success of that task didn't lead to an approximation to the goal. This allows lower levels to learn sub-tasks even when the higher levels haven't learnt to assign the correct ones. Information hiding dictates that every manager only knows the current state according to the granularity of their action set, which can be composed of primitive actions or assignable tasks. One manager does not need to know its super-manager goal, nor does it need to know how the sub-manager intends to solve the assigned task. This is what ensures the hierarchical decomposition pretended.

This paper introduced this important framework, but their approach was suited to a very specific type of problem and its generalization was not addressed. Their problem consisted of a labyrinth where the state-space was straight-forward to divide, while high-dimensional state spaces are difficult to divide and distribute to sub-managers. In reality, the most important contribution of this paper was the assignment of goals to lower layers and the rewarding of those layers according to the accomplishment of the goals proposed, instead of the reward provided by the environment.

As this notion of goal is so crucial to feudal hierarchies, it became important to incorporate goals in the value functions used to learn. To answer this necessity, Universal Value Function Approximators (UVFA) [40] were developed. This approach uses neural networks as functions approximators, and hence it is able to generalize to unseen goals, as well as take advantage of the common structure between goal-space and state-space. To do this, we first sample a few training goals and use any previous method (like TD difference) and a transition history to approximate one Q-function for each goal. Then, these Q-values are inserted into a matrix and this matrix is factorized into $\hat{\phi}$ and $\hat{\psi}$, the state and goal target embedding, respectively. Afterwards, two distinct neural networks are trained w.r.t. the target embedding and the value function is given by a linear combination of the output of both networks, $Q(s, a, g) = h(\phi(s, a), \psi(g))$. The networks may share some parameters in order to capture the similar behavior of the state-space and the goal-space. Although this approach is not hierarchical and does not provide any improvement by itself, it is a very important building block to posterior proposals.

FeUdal Networks (FuN) [7] is one of the main models amongst feudal hierarchies. It has two modules, the Manager and the Worker. As in feudal hierarchies, the manager sets goals to the worker, which in

turn uses primitive actions to fulfil them. The main difference to the option-critic architecture is that the top-level feeds a directional goal chosen from a continuous domain to the lower module, instead of choosing from a set of options.

The goal is implemented as a vector representing the direction to where the Worker should travel in the state space. The goal is generated with a recurrent neural network, and is updated using the policy gradient. Its naive implementation considers the complex trajectories that the Worker may follow. Instead, a model of the transition probabilities given the goal chosen is inferred. By considering the Worker eventually learns to achieve the goal, we can ignore the trajectory and assume the final state will be close to the predicted goal, $s_{t+c} \approx s_t + g_t$. Therefore, it is possible to compose a high-level policy with the transition probabilities to create a transition policy, π^{TP}, to which we can apply the policy gradient theorem. This translates in the following update rule.

$$\nabla g_t = A_t^M \nabla_\theta d_{cos}(s_{t+c} - s_t, g_t(\theta)), \tag{3.6}$$

where A_t^M is the advantage function and d_{cos} is a similarity measure. This means that a direction is reinforced if the reward obtained is bigger than the value function in that state. In order for the Worker to learn how to execute the goals, an intrinsic reward is fed.

$$r_t^I = \frac{\sum_{i=1}^{c} d_{cos}(s_t - s_{t-i}, g_{t-i})}{c}, \tag{3.7}$$

where c is the number of steps the Worker is given to complete the goal. Once again, the trade-off between transferability and optimization is prevalent, in the shape of the possible inclusion of the extrinsic reward. Unlike in the traditional proposal of feudal hierarchies, in FuN the Worker does receive the reward from the environment, meaning the total reward is a combination of both. This improves the practical results.

Another novelty of FuN is using recurrent neural networks to learn a representation of the state, in contrast to using the raw input. This simplifies the input and allows the policy to be dependent on past states, abandoning the Markovian property without increasing the size of the input. Nevertheless, these benefits depend on having a good representation, which can only be learnt once a meaningful reward is obtained. In environments with sparse rewards, these might be difficult to get, causing the representation of the state to be an obstacle instead of a simplification.

On top of that, a limitation of FuN and of all the methods we have visited so far is the use of on-policy training. When using off-policy training instead of on-policy, the agent may reuse past experiences for learning, which can be beneficial especially if interaction with the environment is expensive. Also, off-policy methods can separate the exploration policy and the policy being learnt, which makes the problem of exploration vs exploitation easier to solve. Lillicrap *et al.* [41] combine the innovations of DQN [2] with

the Deterministic Policy Gradient [16] to create the Deep Deterministic Policy Gradient (DDPG), an actor-critic where both the actor and critic are modeled with neural networks as function approximators. The agent stores experiences in a replay buffer, from where the agent samples the mini-batches used for learning. Both the actor and critic network are mirrored by target networks that, unlike DQN, are updated as often as the main networks using a soft-update:

$$\begin{aligned} \theta_{Q'} &\leftarrow \tau \cdot \theta_Q + (1-\tau)\theta_{Q'} \\ \theta_{\mu'} &\leftarrow \tau \cdot \theta_\mu + (1-\tau)\theta_{\mu'} \end{aligned} \tag{3.8}$$

This way, the target networks vary slowly over time. This may slow down learning but greatly improves stability, because the critic can train using a near stationary target, edging the problem closer to the supervised learning case, to which stable solutions exist. Since this method was designed for environments with continuous action spaces, exploration is handled by introducing noise to the action chosen by the policy:

$$\mu'(s_t) = \mu_\theta(s_t) + \mathcal{N}, \tag{3.9}$$

where $\mathcal{N}$ can be an arbitrarily chosen noise function. Finally, DDPG introduces batch normalization [42]. This method adjusts each feature in a minibatch so that it has a mean of zero and a variance of one across the samples. This is very useful to learn across different problems with different magnitudes of input without having to adjust. However, it is not relevant for our work so we will not delve into it. In HRL, however, there is another obstacle to reaping the benefits off-policy training brings. Lower-level policies learn and change over time. For this reason, the same action from the upper-levels may produce different behaviors after a period of time, and previous transition samples might become outdated. Next up, we describe two approaches that use off-policy training by dealing with this non-stationarity in different ways.

HIerarchical Reinforcement learning with Off-policy correction (HIRO) [8] is a two-level hierarchy, where the upper-level policy proposes directional goals to the lower level and offers a parameterized reward that depends on the proximity of the states to the goal, similarly to FeUdal Networks (FuN) [7]. Unlike FuN, HIRO uses the raw state instead of a learned representation, for the reasons already specified. To be able to use off-policy training and counteract the non-stationarity problem, out-of-date samples are relabelled with goals that would cause the current low-level controller to choose the actions in the sample. In other words, we want to transform samples $(s_{t:t+c-1}, g_{t:t+c-1}, a_{t:t+c-1}, R_{t:t+c-1}, s_t)$ into $(s_t, g'_t, \sum R_{t:t+c-1}, s_{t+c})$, where g'_t would induce the actions $a_{t:t+c-1}$ and consequently the final state s_{t+c} in the new low-level controller. In order to do this, HIRO chooses the goal with the biggest probability of generating $a_{t:t+c-1}$ from 10 candidates $\left(log\, \mu_{lo}\left(a_{t:t+c-1} \mid s_{t:t+c-1}, g'_{t:t+c-1}\right)\right)$: the original goal, $s_{t+c} - s_t$ and 8 samples randomized around $s_{t+c} - s_t$. This approach learns faster and obtains better results than previous approaches ([7], [11], [6]) even with extended training. In spite of that, ablative analysis shows that training the lower policy first before freezing it and only then training the higher policy brings better

results in some tasks. Thus, the relabelling technique has room to be improved.

The second approach that uses off-policy learning is HAC [12]. This approach uses a hierarchy of action-critic networks to solve a problem using different temporal resolutions. To deal with the aforementioned non-stationarity, it is assumed by layer i that the layer $i-1$ uses the optimal policy. Because of this, transitions need to be altered to reflect this assumption. For example, if layer i chooses action s_2 and the layer below, after H number of time steps, has reached state s_1, then the transition is stored as if the action proposed was s_1 to begin with: $[initial\,state = s_0, action = s_1, final\,state = s_1, reward =$ TBD$, goal = s_{goal}]$. As for the reward, we don't know the path that the optimal policy would have taken, so it can only be a result of the goal and the final state. The authors decided that it would be 0 if a goal is reached and -1 otherwise. However, this reward is very sparse and very difficult to learn. To combat this, the authors extend the ideas from Hindsight Experience Replay (HER) [43]. HER, by adding a goal parameter to the Q-function $Q(s, a, g)$, also calculates the reward w.r.t. the goal in a binary fashion. It samples different goals than the one originally assigned, calculates the rewards of the trajectory with respect to the new goals, and stores those transitions. Humans can take knowledge from failure, as well as success. If someone shoots a ball slightly to the right of a goal, they know that they are close to success, even without obtaining it. Similarly, we want reinforcement learning agents to learn from failure. Hindsight experience replay generates artificial goals when the agent fails, and they ultimately result in achieving the original goal.

HAC brings another interesting innovation. The goals output by the top layers are states, instead of directional goals. All layers have a specified number of time steps, H, to achieve the proposed goal. This causes each layer to specialize in a different temporal resolution, because they need to output sub-goals ambitious enough to solve their goal in H time steps, but not to the point where the subsequent layer cannot reach the proposed sub-goal in time. Another benefit of the limit is that policies are shorter and easier to learn, since each layer only focuses on a smaller region of the state-action-goal space. This makes propagation of positive rewards faster and credit assignment easier.

Both these approaches are able to use off-policy learning, which is more data-efficient than on-policy learning. They each suggest a different way to adapt to the non-stationarity created by learning policies dependant on each other simultaneously. HAC obtains slightly better empirical results and was the first model to improve with the usage of more than 2 layers. Their main limitation is that mapping the goal space to the state space may be difficult in environments with high-dimensional state-spaces. Nevertheless, these two approaches are able to beat the performance of their counterparts with much less data. Table 3.4 summarizes the algorithms we explored under the feudal hierarchies framework.

Table 3.4: Summary of the main approaches under the feudal hierarchies framework

Algorithm	Required priors	Main contributions	Main disadvantages
Dayan *et al.* [39]	Division of the state-space	Introduction to the framework of feudal hierarchies; Reward hiding; Information hiding	Inapplicable to more complex environments
FuN [10]	None	Learned sub-goal space; Combination of intrinsic and extrinsic reward; LSTM to learn low-dimensional embedding of state-space	Needs meaningful rewards to learn the embedding of the state-space; Sub-goal space may not be optimal
HIRO [8]	Mapping from state-space to sub-goal space	Off-policy learning through relabelling of samples; Increased sample efficiency	Relabeling can be improved; Mapping might be challenging
HAC [12]	Mapping from state-space to sub-goal space	Off-policy learning by assuming optimal policy is used; Increased sample efficiency	Mapping might be challenging

4

Method

Contents

To reiterate, we want to create a model that leverages the ability of HRL to divide a complicated task into smaller sub-problems to learn more effectively in an environment that is constantly changing. This environment mutability falls in the framework of modes of operation. In this framework, there is a small number of modes of operation, the agent cannot observe the current mode, mode changes are not influenced by the agent's actions and mode changes are very infrequent. Furthermore, we assume that the modes do not alter the main goal of the task. Since the considered environment rewards agents very sparsely, concretely when the goal is being achieved, this is equivalent to assuming that rewards remain approximately constant throughout modes. We are mainly focused on the changing of transition probabilities. As a result, the first step in designing this model is deciding how to deal with this non-stationarity. As we verified, the most successful HRL methods are model-free and for this reason dealing with non-stationarity with a model-free algorithm would be a good fit. Unfortunately, model-free algorithms fail to take advantage of our particular framework, being unable to learn good policies for multiple modes at the same time, unless the change pattern is given. This leaves us with model-based methods, where a model of the transition probabilities will be learnt by each module. The success of each module on predicting the next state will determine the selection of the active module. Notice that we will not implement a reward prediction model due to their relative stationarity, mentioned above. Since the agent will use a model-free reinforcement learning algorithm, the transition probabilities learnt by the context detector will not be used for anything other than identifying the current mode.

Having decided how to select the active module, we need to determine what will be shared between the different modules and what will be unique. One of our objectives is using the modularity of HRL to relieve some of the layers of the necessity of tracking the current mode of the environment, creating an abstraction. Therefore, we have decided that the context detector will work on a single layer, while others maintain their view of a stationary environment. The main hypothesis is fixing the upper level layers, responsible for long-term and mid-term planning, and let the lower level layer deal with the intricacies of the environment. We expect this approach to be best in environments where the context changes the consequences of atomic actions, but the general path to success remains the same. Another option is to use the same lower level policies throughout every mode, making the upper level responsible for adapting to the mode. This would mean the agent would learn how to achieve short-term goals useful for every mode, while the upper policies determined how to exploit them best according to the active mode. We believe this method has some merit, although it may fall under the framework of options if we view the low-level policies as skills. This approach can only work if the optimal ways of reaching short-term goals are not affected by changes in mode. In this work, we choose to prioritize the feudal hierarchies framework, focusing our evaluation and our method on the case where the lower level layer is the layer to be merged with the context detector. We leave the case where the higher layer is responsible for adapting to the mode for Future Work (cf. section 6.1).

As a consequence, we are able to adapt a large part of HAC [12] to our proposal. We will have one, or more, upper layers that are unaware of the current mode. Each layer will receive the current state and a sub-goal as input, with the top layer receiving the overall goal of the task. The goal space is a projection of the state space. In other words, sub-goals are states for the agent to reach where some of the dimensions may be suppressed. Every layer must achieve the proposed goal in a limited number of steps, H, and as a result, each layer is encouraged to specialize in a different time granularity. In addition, every layer must learn to output goals that are ambitious enough to achieve its own goal in the time limit as well as conservative enough for the consecutive layer to be able to reach it in H time steps.

Meanwhile, the bottom layer will constitute the difference between our proposal and HAC. When the bottom layer is prompted, it must check which of the modules is currently active and execute the corresponding policy. In the actor critic framework, policies are generated using a parameterized policy and a Q function (more details will be provided in the next section). These functions can either receive the estimated active mode as an argument, $Q(s, a, m)$ and $\mu(s, m)$, or there can be individual functions for every mode, $Q_m(s, a)$ and $\mu_m(s)$. As modes of operation suggest abrupt changes between them, we chose the latter. In spite of that, it would be possible that these functions share some parameters, in order to capture the state in a similar manner, for example. We leave this for future work. Figure 4.1 represents an illustration of the architecture.

Next, we will discuss the implementation details of our method. We will firstly explain HAC [12] in more detail, followed by how the context detector is implemented and exactly how it merges with a layer of HAC.

4.1 HAC

4.1.1 Layered DDPG

The reinforcement learning method behind each of the HAC's layers is DDPG [41]. Each layer consists of an actor-critic, that approximates both its policy μ_θ and the state-action value function Q_θ with function approximators, specifically neural networks. Unlike DDPG, our actor-critic needs to choose and evaluate actions depending on the goal assigned so both the policy and Q-function receive the current goal as an extra input. The goal is a projection of the state. For example, imagine we have an agent that has a position and a velocity. The goal of this agent might be to reach a certain position, and it doesn't matter at what velocity the position is reached. In this case, the state space consists of $[position, velocity]$ and the goal space only of $[position]$. The goal is passed to the neural networks concatenated with other inputs, meaning that μ_θ receives as input $(state \mid\mid goal)$ and Q_θ receives $(state \mid\mid action \mid\mid goal)$, where $\mid\mid$ is the concatenation operator.

Each layer operates independently with its own actor-critic network and corresponding replay buffer.

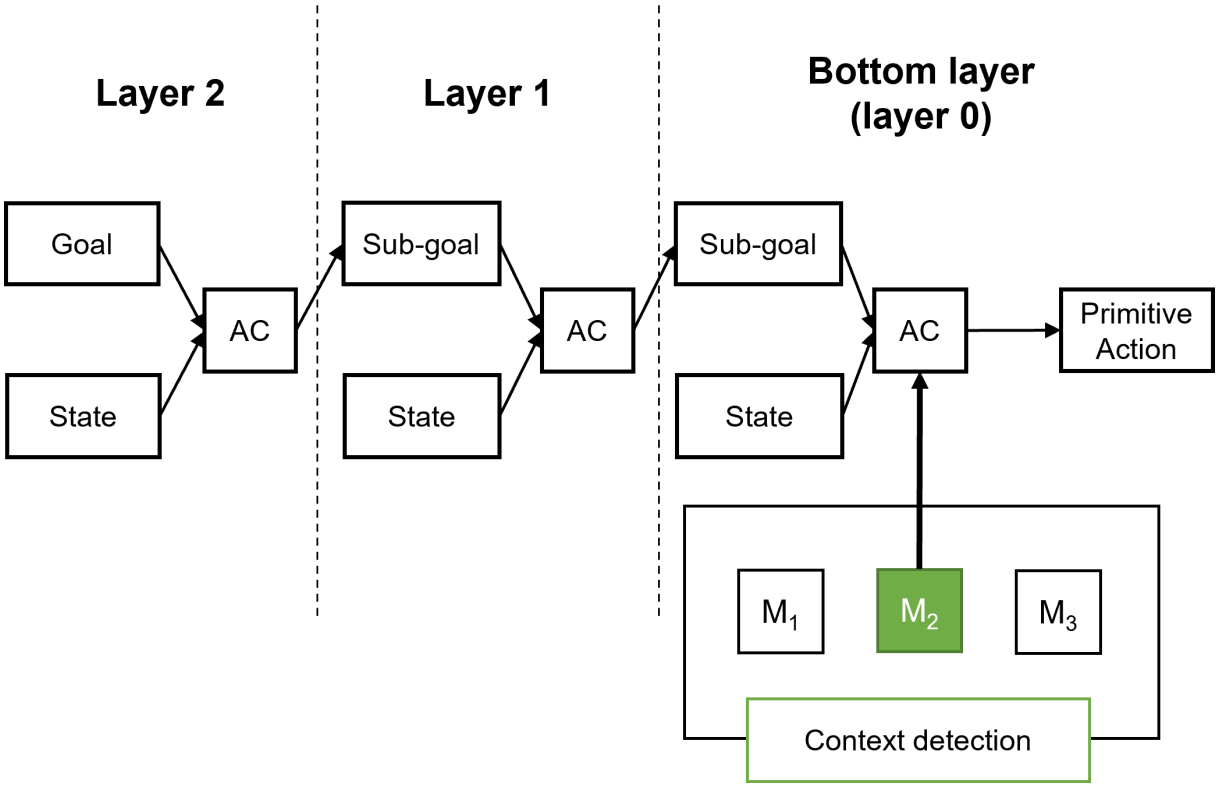


Figure 4.1: Summary of the architecture proposed. AC boxes the actor critics. The context detection is responsible for choosing the active module (M_2) and it provides the policy responsible for outputting the primitive action.

Typically, the samples stored in a replay buffer are structured as $[s_t, a_t, r_t, s_{t+1}]$. However, in HAC, the goal the agent is pursuing at each time step must also be included. Furthermore, for higher layers, s_{t+1} does not represent the state immediately following s_t. Since these top layers function at a coarser temporal resolution, the sequence of actions of the lower layers are abstracted into a single action, and s_{t+1} reflects the outcome of all these actions. Thus, throughout each episode, every layer gathers samples according to its own time scale and stores them in its respective replay buffer. After the episode, each layer independently selects random mini-batches from its buffer and updates its networks based on the update rules provided below. HAC can also use target networks—slower-updating copies of the main networks—to enhance learning stability. While we utilize target networks in some experiments, they can be omitted in simpler environments where their stabilizing effect is less evident.

Critic update:

$$y_t = r_t + \gamma Q'_\theta \left(s_{t+1}, \mu'(s_{t+1})\right)$$

$$L(\theta^Q) = \frac{1}{N} \sum_t \left(Q_\theta(s_t, a_t) - y_t\right)^2 \triangleright \text{Minimize the loss}$$

Actor update:

$$\nabla_{\theta^\mu} J(\theta^\mu) = \frac{1}{N} \sum_t \nabla_a Q_\theta(s_t, a)\Big|_{a=\mu(s_t)} \nabla_{\theta^\mu} \mu_\theta(s_t)$$

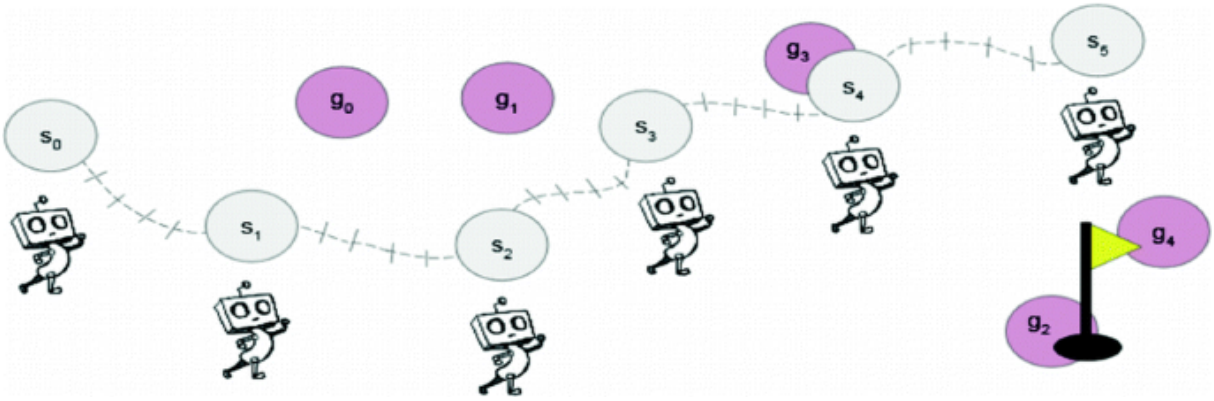


Figure 4.2: Example of HAC episode with a 2-layer architecture. The yellow flag represents the goal of the environment, the purple circles represent the goals proposed by the top layer while the grey circles represent the actual states achieved by the agent. The tick marks represent the state of the agent after each primitive action. There are 4 ticks between each high-level state so the bottom layer had 5 time steps to achieve its proposed goal, $H = 5$.

4.1.2 Hindsight actions

In figure 4.2, we illustrate a simple example of a HAC episode using a 2-layer architecture. The top layer generates sub-goals for the bottom layer, which has $H = 5$ time steps to achieve each sub-goal. However, the bottom layer may not always succeed in reaching these goals within the allocated time, so the states shown represent where the agent ended up after attempting to reach the goal states. This process introduces non-stationarity, as the transition dynamics of the higher layers depend on the evolving policies of the lower layers. For instance, when attempting to reach g_0, the bottom layer might initially hit an arbitrary state s_1 and only later, after further learning, achieve the desired sub-goal state g_0 after learning.

Because the agent stores its experiences in a replay buffer and reuses them later, many of these stored samples can become outdated, no longer reflecting the agent's current behavior. Furthermore, exploration introduces additional variability in the transition probabilities of higher layers. Since the learned policy is deterministic, noise must be added to encourage exploration of non-optimal actions. In the HAC framework, the agent follows an ϵ-greedy policy, injecting noise into each selected action. Specifically, with probability $\epsilon << 1$ a random action is chosen, and with probability $1 - \epsilon$, an action is selected as $a = \mu_\theta(s) + \mathcal{N}(0, \sigma^2)$, where $\mathcal{N}(0, \sigma^2)$ is normally distributed noise.

The issue of non-stationarity caused by the exploration of lower-layer policies is addressed by assuming that the policies used by lower layers are optimal. This way, the transition probabilities of the higher

layers become independent of the evolving policies in the lower layers. To achieve this, the authors introduce hindsight action transitions.

First, under the assumption of an optimal policy, the sub-goal state will always be reached if it is achievable. Therefore, instead of storing the originally proposed sub-goal state in the transition, the stored action is the state actually achieved by the lower layer in hindsight. Second, we aim to reward the agent for achieving the goal and encourage it to find the shortest possible paths. However, since the agent does not in fact know the optimal policy and we cannot explicitly measure the shortest path, the reward function depends only on the next state and the goal, specifically whether the goal was reached. The agent receives a reward of 0 if the goal is achieved and -1 otherwise. To illustrate, refer to the first transition in Figure 4.2. The agent starts in state s_0 and Layer 1 sets g_0 as the sub-goal. After 5 time steps, Layer 0 reaches state s_1, which differs from the proposed sub-goal. The hindsight action transition is then recorded as $[previous\,state = s_0, action = s_1, next\,state = s_1, reward = -1, goal = yellow\,flag]$. The hindsight action is the state reached by layer 0, s_1, and the reward is -1 because the overall goal was not achieved.

Interestingly, this approach to handling non-stationarity also helps isolate environmental changes. Since higher layers are not overly impacted when lower layers forget how to reach sub-goals, their knowledge of the overall path to the goal remains intact.

4.1.3 Dealing with sparse rewards

We previously defined that the agent is only rewarded if it achieves the goal. However, this sparse reward structure hinders learning, as it is difficult for the agent to randomly discover the sequence of actions required to reach the goal. To address this, HAC implements HER [43], where, similar to hindsight actions, new goals are sampled after execution to artificially generate rewards in otherwise sparse scenarios. Specifically, for every sequence of H actions, one of the future states is chosen as the goal, replacing the original goal, and the reward is recalculated accordingly.

Revisiting the example from figure 4.2, as Layer 0 tries to achieve goal g_0, the natural transition of the first action would be recorded as $[previous\,state = s_0, action = a_0, next\,state = "first\,tick\,mark", reward = -1, goal = g_0]$. However, instead of storing this directly, the goal and reward fields are initially left empty: $[previous\ \ state = s_0, action = a_0, next\,state = "first\,tick\,mark", reward = TBD, goal = TBD]$. After the 5 time steps, one state from the trajectory is selected as the hindsight goal, such as s_1. The first transition would then be stored as $[previous\,state = s_0, action = a_0, next\,state = "first\,tick\,mark", reward = -1, goal = s_1]$, and, repeating the same process, the fifth transition would be $[previous\ \ state = "fourth\,tick\,mark", action = a_4, next\,state = s_1, reward = 0, goal = s_1]$, where the next state matches the goal and the reward is given.

This mechanism helps the agent eventually learn the path to real goals because its critic is modeled

by neural networks that take the goal as input, similar to UVFA [40] but without the use of embedding. The network assigns high values to the state-action-goal combinations (s, a, g), which will eventually generalize to the real goals as the agent learns that similar goals require similar paths. This approach ensures that each sequence of H actions includes rewards, enabling learning even when the actual goal is not reached.

4.1.4 Sub-goal testing transitions

Each layer in HAC is responsible for learning at a specific time granularity. This means each layer must output sub-goals that are ambitious enough to achieve its goal, but simple enough for the next layer to complete within H time steps. So far, we have introduced hindsight action transitions and hindsight goal transitions, both of which allow the agent to ignore when lower layers fail to reach the sub-goal. This causes the agent to never update the Q-values for distant goals. If the agent assigns high initial Q-values to these sub-goals, it may persist in pursuing short paths that lower layers are unable to execute, rather than generating sub-goals that both lead to the goal and are feasible for the lower layers.

Therefore, HAC implements sub-goal testing transitions, a third set of transitions responsible for penalizing sub-goals that lower policies cannot achieve. In a small fraction of steps, upon receiving a testing sub-goal, a layer will use its current policy without noise, trying its best to achieve the proposed sub-goal. If it fails, this transition is attributed a penalty in the reward. The authors define the penalty as $-H$ and we maintain this value for our experiments. In our example, if the agent decided to test the first sub-goal proposed by Layer 1, since Layer 0 was unable to reach g_0, the transition that would be stored is $[previous\,state = s_0, action = s_1, next\,state = s_1, reward = -5, goal = yellow\,flag]$

We analyze the impact of sub-goal transitions on 3 different types of sub-goals. Firstly, sub-goals that cannot be achieved by the optimal policy of the lower layer will never be updated by hindsight action or goal transitions because these transitions only update sub-goals reached in hindsight. However, when these sub-goals are tested, the lower layer will fail to achieve them, and they will be penalized. Over time, their Q-values will approximate the penalty. Secondly, sub-goals already achievable by the lower layer's current policy will not be penalized because the sub-goal testing will always succeed. Finally, the most complicated case is when the sub-goals are achievable by the lower level's optimal policy but the current policy has not yet learned to achieve them. These sub-goals will be penalized initially, but as the lower layer improves, the agent will start reaching them, generating hindsight action transitions. The final Q-value will be a weighted average of the penalty and the reward from hindsight action transitions. Since the latter are much more frequent than the former, the sub-goals Q-value will converge to its true value.

4.2 Context detector

4.2.1 Identifying the active module

Methods that identify change-points through statistical methods that compare the divergence between contexts ([21], [19]) are not suitable in a high-dimensional state-space, because they need to determine $\mathcal{P}_m(s_{t+1}|s_t, a_t)$. Even if we could use function approximators to estimate $\mathcal{P}$, the probability distribution would be concentrated within states that are similar, in other words, states that have a small euclidean distance between them. To address this, we adapt MMRL's [17] calculation of the responsibility signal, where each module attempts to predict the next state, and the accuracy of each module's prediction determines its responsibility for the current mode. The responsibility signal is defined as:

$$\hat{\lambda}_i = \frac{e^{-\frac{1}{2\sigma^2}||s(t)-\hat{s}_i(t)||^2}}{\sum_{j=1}^{M} e^{-\frac{1}{2\sigma^2}||s(t)-\hat{s}_j(t)||^2}}, \tag{4.1}$$

where M is the number of modules, $s(t)$ is the observed state and $\hat{s}_i(t)$ is the predicted state by module i. The score for each module increases as the Euclidean distance between its prediction and the observed state decreases. In HAC, this responsibility signal is used to weigh each module's proposed action in a weighted average. Also, since no module is fully responsible for the agent at any given step, all modules learn proportionally to the responsibility signal in each transition. In HAC, however, learning is off-policy and uses a replay buffer, which complicates using the responsibility signal system. This would require maintaining the responsibility signal for each transition and deriving a general responsibility signal for each mini-batch. For simplicity, we use the more direct approach of RL-CD [18], where the module with the highest score becomes the active module, assuming full control of the agent for as long as it remains the active module.

Besides, MMRL's responsibility signal formula uses the Mean Squared Error (MSE), where the error at each step is the square of the Euclidean distance. In our method, we use the Huber Loss [44], which combines the advantages of MSE and Mean Absolute Error (MAE). For small errors, MSE is preferred because a quadratic error causes the gradient to converge faster. Beyond a predefined threshold, MAE is applied, corresponding to the absolute value of the Euclidean distance. This makes the function more robust to outliers, as their influence becomes linear. In our experiments, we observed that certain edge states had transitions probabilities that behaved very differently from the rest of the environment, hindering the performance of the context detector. Huber Loss helped to reduce their effect as much as possible. Huber Loss is represented in eq. (4.2), where $a = ||s(t) - \hat{s}_i(t)||$ is the error, and δ is a threshold that determines the point at which the loss transitions from quadratic to linear. The final version of the score calculation is represented in equation 4.3. Additionally, it's important to note that before each

score calculation, the losses are normalized to have a unit mean to prevent excessively small values.

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \leq \delta, \\ \delta(|a| - \frac{1}{2}\delta) & \text{for } |a| > \delta, \end{cases} \tag{4.2}$$

$$\hat{\lambda}_i = \frac{e^{-\frac{1}{2\sigma^2}L_\delta(a)}}{\sum_{j=1}^{M} e^{-\frac{1}{2\sigma^2}L_\delta(a)}}, \tag{4.3}$$

We know from our modes of operation framework [5] that changes in mode are infrequent, so we want to ensure that the active module doesn't change frequently. A brief inaccuracy shouldn't immediately cause it to be overtaken. For this reason, we add temporal continuity, defining that the score of each module depends on the previous values: $\lambda_i = \lambda_i + \alpha(\hat{\lambda}_i - \lambda_i)$. The scores of all modules are updated at every time step, and the module with the highest updated λ_i is selected as the active module. The parameter α defines the impact previous predictions have on the active module, with higher values allowing for faster detection of mode changes, and lower values making the system less susceptible to noise.

Lastly, each module learns the transition probabilities using a neural network. The neural network receives the state and action as input and outputs the predicted state, $\hat{s}_i(t)$. Every module outputs a predicted state for every time step, as this is necessary to calculate their score, but the transition $[s_t, a_t, \hat{s}_i(t+1), s_{t+1}]$ is only used for learning by the active module. This ensures that the module accurately predicting the state specializes in this mode, while others ignore it. Further details on the learning process will be discussed in the next section. The neural networks we used do not necessarily have an activation function, they frequently are simple linear approximators. In the evaluation section, we perform a comparison between using neural networks versus linear approximators. In environments where both performed equally well, we used linear approximators because they are simpler.

4.2.2 Learning

Our context detector employs a replay buffer, similar to DQN [2], to reduce the correlation between transitions. However, our replay buffer only stores the most recent n transitions. Every n transitions, we compute the average loss for each module and calculate the score using equation 4.3. This score represents the overall performance for the interval since averaging the losses over the transitions inherently accounts for temporal continuity. Besides, the interval of n steps is significantly shorter than the time required for the environment to switch modes, allowing us to assume there is only one correct module for the entire window. The context detector then selects several small mini-batches at random and updates the module with the highest score. Replay buffers allow for transitions to be reused, improving data efficiency. In our case however, each transition is used more than once due to the number of mini-batches sampled, but after all mini-batches are processed, these transitions are discarded.

An alternative approach would be to maintain multiple separate replay buffers for each module, and

Algorithm 4.1 Context Detector Algorithm

Initialize a set of modules $\mathcal{M}$ with a single module, each module parameterized by θ
Initialize an overall score λ_i for each module $i \in \mathcal{M}$, set $\lambda_i = 0$
Initialize the active module $\mathcal{M}_{active}$ to the first module in $\mathcal{M}$
for each iteration **do**
  Sample n transitions $\{s_t, a_t, \hat{s}_i(t+1), s_{t+1})\}_{t=1}^{n}$
  for each transition $t = 1$ to n **do**
    for each initialized module $i \in \mathcal{M}$ **do**
      Compute the predicted state $\hat{s}_i(t) = \mathcal{T}_{\theta_i}(s_t, a_t)$
      Compute the immediate score $\hat{\lambda}_i$ using equation 4.3
      Update the overall score λ_i using the alpha-step update rule:

$$\lambda_i \leftarrow \lambda_i + \alpha \left(\hat{\lambda}_i - \lambda_i \right)$$

    end for
    Check if any module's score surpasses the active module by the threshold ϕ:
    if there exists a module $i \in \mathcal{M}$ such that $\lambda_i > \lambda_{active} + \phi$ **then**
      Set i as the new active module $\mathcal{M}_{active}$
    end if
  end for
  Call **TrainModule**$(m, \{(s_t, a_t, \hat{s}_i(t+1), s_{t+1})\}_{t=1}^{n})$
  for each module $i \in \mathcal{M} \cup m$ **do**
    Compute the predicted states for every time step
    Calculate the average loss value

$$L_{\theta_i} = \frac{1}{n} \sum_{t=1}^{n} L_\delta(s_{t+1} - \hat{s}_i(t+1))$$

    Update overall score λ_i using equation 4.3
  end for
  Check if any module's score surpasses the active module by the threshold ϕ:
  if there exists a module $i \in \mathcal{M}$ such that $\lambda_i > \lambda_{active} + \phi$ **then**
    Set i as the new active module $\mathcal{M}_{active}$
  end if
  if m is the active module **then**
    Add m to $\mathcal{M}$
  else
    Reset m
  end if
  Call **TrainModule**(active module, $\{(s_t, a_t, \hat{s}_i(t+1), s_{t+1})\}_{t=1}^{n}$)
end for

Algorithm 4.2 TrainModule$(i, \{(x_t, a_t, \hat{x}_i(t+1), x_{t+1})\}_{t=1}^{n})$

1: Calculate the number of mini-batches: $N_{batches} = \frac{n \times 2}{BATCH\_SIZE}$
2: **for** each mini-batch $b = 1$ to $N_{batches}$ **do**
3: Sample a mini-batch of transitions from $\{(s_t, a_t, \hat{s}_i(t+1), s_{t+1})\}_{t=1}^{n}$
4: Perform a gradient descent update step on θ_i using the Huber loss:

$$L_{\theta_i} = \frac{1}{BATCH\,SIZE} \sum_{t=1}^{BATCH\,SIZE} L_\delta(s_{t+1} - \hat{s}_i(t+1))$$

where s_{t+1} is the real next state, and $\hat{s}_i(t+1)$ is the predicted next state by the active module $\mathcal{M}_{active}$.
5: **end for**

assign transitions to the respective buffer based on the active module at the time. This approach allows more transitions to be stored and reused. However, it risks perpetuating errors if the active module is incorrectly identified, as transitions stored in the wrong buffer could negatively affect performance over time. We compare these two approaches in our evaluation.

Finally, the context detector must detect when the environment changes to an unseen mode and initialize a new module. To address this, before the update step described above, a new module is initialized and trained with the previous n transitions. The scores of both the new and the previous modules are compared, and if the new module outperforms the others, it is added to the set of initialized modules. If not, the new module's weights are reset to its initial values to be retrained in the next iteration. Early in the training process, when the first module is still learning, it is allowed to train without competition from other modules to prevent premature initialization of additional modules due to lack of experience.

We evaluated the context detector in two scenarios: one where the total number of environment modes is known, and one where it is not. When the number of modes is known, the agent stops initializing new modules once the number of initialized modules matches the number of modes. Otherwise, the agent continues training a candidate module in each iteration. The pseudo-code for the context detector is described in algorithm 4.1.

4.2.3 Merging HAC and context detector

The context detector described will be integrated into a single layer of HAC, referred to as a "moduled layer." Unlike standard HAC layers, which contain a single actor-critic model, a moduled layer will have an independent actor-critic network for each module created by the context detector. Additionally, each module will have its own replay buffer. At each step, the context detector feeds the active module to the layer. The layer then uses the corresponding actor to generate an action and the resulting transition is stored in the replay buffer specific to that module. Since the mode does not change during an episode,

only the active module is trained at the end of each episode, using the transitions it has collected while it was active. An alternative approach would involve using a single actor-critic model for the entire moduled layer, with the active module being passed as input to the network. This alternative is compared to the proposed approach in the evaluation.

5

Evaluation

Contents

In this chapter, we present the evaluation of our architecture. Firstly, we present the environments where the evaluation was performed and the baselines our architecture was compared against. Next, we present the details of how the evaluation was conducted and the metrics used. Lastly, we present the results and address if the architecture fulfils our objectives. To reiterate, we want our architecture to be able to learn in environments with a high-dimensional state and sparse reward while ensuring data efficiency. Furthermore, the agent should adapt well to changes in the environment and avoid the problem of catastrophic forgetting. After encountering a new mode, the agent should leverage prior experience to adapt, rather than relearning from scratch.

We evaluate the agent's performance across several scenarios, including environments with two modes of operation, environments with multiple modes, and scenarios where the total number of modes is unknown to the agent. Additionally, we will explore alternative approaches to validate and justify our design choices.

5.1 Environments

5.1.1 Pendulum

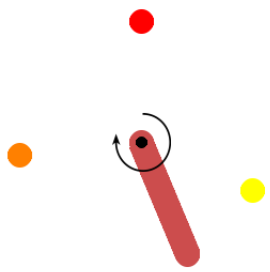


Figure 5.1: Pendulum environment. The circles represent the goals: the red circle represents the overall goal of the episode, and the orange and yellow circles represent the outputted sub-goals by the first 2 layers. The arrow represents the action taken by the lower layer.

The first environment used for evaluation was the Pendulum environment from Open AI Gym [45], as shown in fig. 5.1. The environment simulates a pendulum with one end attached to a fixed point and a free end. The observed state includes the pendulum's angle and angular velocity, while the agent controls the torque applied to the free end. The environment's goal is to swing the pendulum in its

upright position, which corresponds to achieving an angle of zero degrees with zero angular velocity. Each episode terminates after 1000 primitive actions, or earlier if the agent successfully achieves the goal. In the latter case, the episode is considered successful.

We introduce modes of operation in two different ways for evaluation. In experiments with two modes of operation, non-stationarity is achieved by inverting the agent's actions in the second mode. This means that in one mode, positive torque moves the pendulum clockwise, while in the other mode, positive torque moves it counterclockwise. Our architecture combines HAC and the context detector in the lower layer, making it well-suited for environments where the dynamics of movement change between modes.

For experiences with three modes of operation, we modify the pendulum's mass. In mode 0, the pendulum weighs one unit, in mode 1, it weighs 0.1 units, and in mode 2, it weighs 50 units.

5.1.2 Ant Reacher

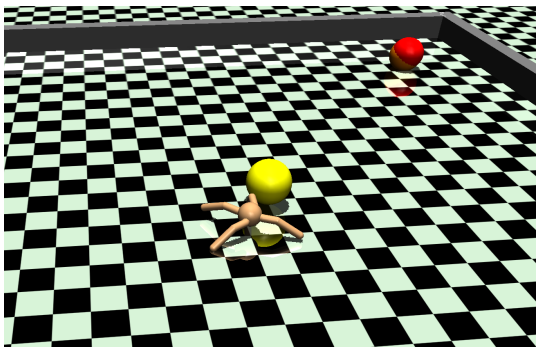


Figure 5.2: Ant Reacher environment. The circles represent the goals: the red circle represents the overall goal of the episode, and the orange (hidden) and yellow spheres represent the outputted sub-goals by the first 2 layers.

The second environment evaluated was Ant Reacher, a variation of the Ant environment first introduced by Schulman *et al.* [46] and part of the MuJoCo suite [47]. In the Ant environment, a quadruped robot (the "ant") consists of a torso attached to four legs, each made up of two links. The state space includes the ant's position, velocity, and angular velocity, as well as the relative angles and angular velocities between its nine body parts. Typically, the agent's objective is to apply torque to the eight hinges of the legs in a coordinated manner to enable the ant to walk. The agent is rewarded for maintaining a healthy position and receives additional rewards proportional to the distance covered. However, HAC cannot use the extrinsic reward from the environment because, as explained in chapter 4, the reward must depend solely on the state reached. Consequently, following the original HAC [12], we introduced a goal position that the agent must reach. An episode terminates after 1000 primitive actions or earlier if the ant reaches the target. A fifteen frame skip was applied in all experiments performed in this environ-

ment, as in the original HAC. Every action taken lasts for fifteen frames and every fifteen frames count as a single step.

In experiments with 2 modes of operation, we introduced non-stationarity in the same way as in Pendulum. The first mode is the normal setting and the second mode inverts the torques applied. We don't perform experiments with more than 2 modes of operation in this environment.

5.2 Baselines

We compare our agent against three baselines:

1. **HAC without the context detector:** As shown by Levy *et al.* [12], HAC outperforms HIRO [8] in the Pendulum and Ant Reacher tasks. HIRO, in turn, has been successfully compared to other leading hierarchical approaches suited for continuous state and action spaces, such as FuN [7] and Option-Critic [6]. Therefore, we consider HAC without the context detector a strong representative of state-of-the-art hierarchical methods.

2. **Adapted DDPG:** This is a flat version of the HAC architecture, where no hierarchy is used. HAC implements DDPG [41] as its actor-critic, so a single-layered HAC resembles DDPG. The key differences are that DDPG uses extrinsic rewards from the environment, while HAC relies on goal-oriented rewards and HER [43]. Additionally, in HAC, both the actor and critic take the goal as an input, unlike in DDPG. We refer to this baseline as Adapted DDPG.

3. **Flat HAC with context detector:** This baseline evaluates the impact of hierarchy in our architecture by comparing a flat HAC (without hierarchical levels) coupled with the context detector. This comparison helps us measure how much the hierarchical structure itself contributes to the agent's performance.

5.3 Evaluation setup

In our experiments, we alternate between training and testing episodes. During training, HAC uses an exploration policy that introduces noise into its actions and periodically outputs random actions. Therefore, measuring module accuracy during training does not accurately reflect the true policy. In testing episodes, the agent does not store new experiences or perform learning, instead using its true, noiseless policy. By alternating training and testing phases, we can assess the agent's real performance after various numbers of training episodes. The environment's mode changes every 200 episodes, and the testing phase lasts for $200 \times number\,of\,modes$ episodes, allowing the agent to be evaluated in every

mode during each testing phase. All results are averaged over at least five different randomly-seeded runs, with a 95% confidence interval where applicable.

We extended most of the implementation details from the original HAC architecture [12]. In CoHAC, all neural networks are modeled as multi layer perceptrons (MLPs) with three hidden layers of size 128, using the ReLU activation and the Adam [48] optimizer. We suppressed the context predictor's activation function in experiments where a linear perceptron performed equivalently. We discuss this further in section 5.4.2. Hyperparameters were also as much as possible imported from HAC. In the context detector, where there were no reference values to rely on, we experiment with several configurations in each experiment and present the results with the best one.

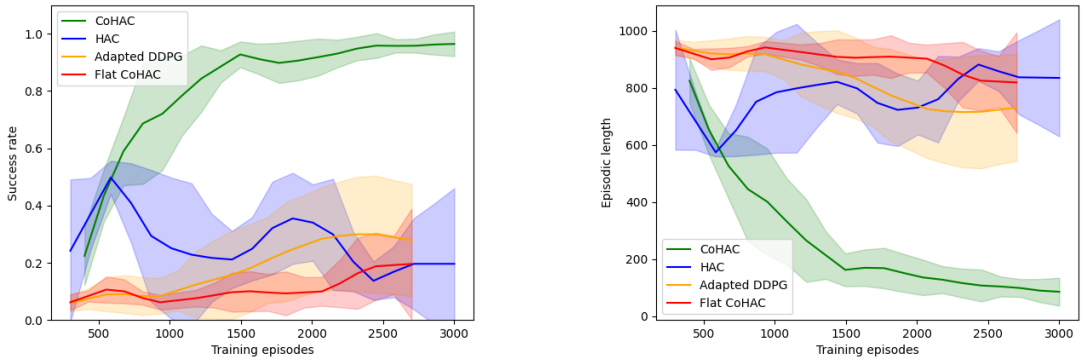
The key metrics for evaluating the agent's overall performance are success rate and average episodic length during each testing phase. On the other hand, explicitly evaluating the success of the context detector is more challenging. For instance, module 1 may learn to predict mode 0 and vice-versa, leading to scenarios where the agent never guesses the "correct mode", yet has a perfect understanding of the environment. Similarly, if the agent always guesses mode 0, it achieves a 50% accuracy rate even if the context detector is completely ineffective. To address this, we sometimes display the context predictions alongside the correct mode of a specific run, providing a visual representation of the context detector's success. Another metric we use is the loss from the active module when predicting the next state, which is evaluated during training episodes, as the context detector always performs to the best of its abilities during this phase.

5.4 Results

We start by comparing CoHAC with the 3 baselines (HAC, Adapted DDPG and Flat CoHAC) in the Pendulum and Ant Reacher environment, both with 2 modes of operation. We train the agent over 3000 episodes, and show the success rate and the episodic length obtained in the testing phases throughout training. We present the results in the Pendulum environment in fig. 5.3 and results in the Ant Reacher environment in fig. 5.4.

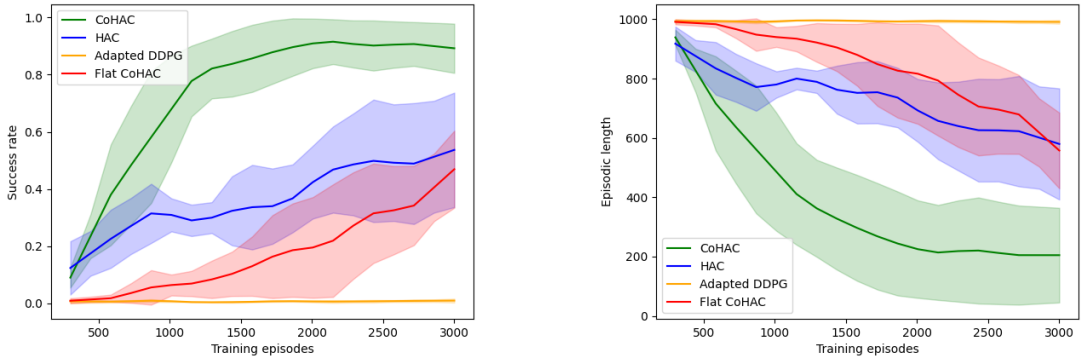
The results show that CoHAC significantly outperforms other configurations, confirming the success of our approach. While other approaches don't surpass the 50% success rate, CoHAC effectively solves both environments and achieves performances similar to HAC in the stationary environments. This means that CoHAC is successful at isolating the non-stationarity, while maintaining data efficiency and learning different modes at the same time.

Interestingly, contrary to expectations, the 3-layered HAC without context detector did not suffer from catastrophic forgetting. Evidence for this can be seen in fig. 5.5, which displays episodic length during training instead of the testing phases as we have previously shown. Although there are performance



(a) Success rate **(b)** Episodic length

Figure 5.3: Baseline comparison in the Pendulum environment. Comparison between CoHAC, original HAC, CoHAC with a single layer, and flat HAC (adapted DDPG). Results averaged across 5 different randomly-seeded runs.



(a) Success rate **(b)** Episodic length

Figure 5.4: Baseline comparison in the Ant Reacher environment. Comparison between CoHAC, original HAC, CoHAC with a single layer, and flat HAC (adapted DDPG). Results averaged across 5 different randomly-seeded runs.

drops, they do not align with the mode changes, represented with the blue dotted lines. We attribute this phenomenon to the off-policy nature of the learning process. Since the replay buffer retains transitions from far more than 200 episodes, transitions from both modes coexist, allowing the agent to learn both simultaneously. Ultimately, the agent tends to favor one mode, leading to a 50% success rate. However, HAC cannot learn a policy that is suitable for both modes and cannot quickly react to mode changes.

The flat HAC coupled with the context detector also obtains interesting results. In the Ant Reacher task, we can see that the learning is slower but the success rate is growing and we assume that given enough episodes it would achieve the same performance as the complete architecture. In the pendulum

task, however, it underperforms compared to DDPG, and the context detector's effectiveness is reduced compared to the complete architecture. We hypothesize that in the flat architecture, slower exploration limits the agent's ability to reach states far from the pendulum's resting position. In these states, the difference in mode is less expressive because the pendulum moves less, and the context detector's accuracy is affected. We conclude that as the context detector can have a negative impact on HAC, HAC's limited exploration in less expressive states can, in turn, negatively impact the context detector.

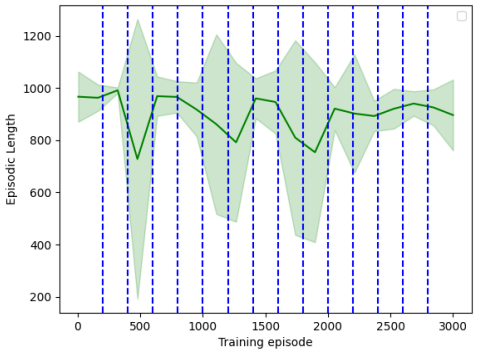


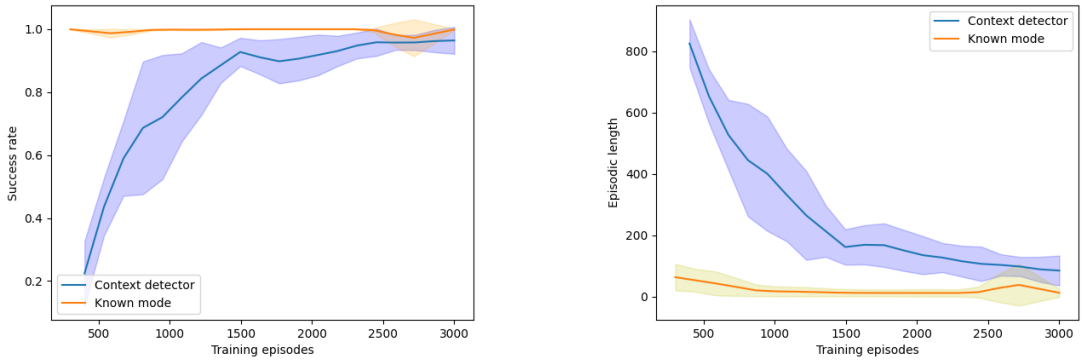
Figure 5.5: Analysis of the episodic length of each HAC training episode. Separators are used to indicate where the mode changes and to show that HAC, even if it performs much worse, does not suffer with catastrophic forgetting. Results averaged over 5 different randomly-seeded runs.

5.4.1 Context detection validation

In this section, we display further experiments to analyse the context detector more in depth. Specifically, we compare our architecture with that of HAC paired with an ideal context detector that directly provides the correct mode. We present these results in the Pendulum environment in fig. 5.6 and in the Ant Reacher environment in fig. 5.8. Additionally, we display an example of the context predicted across an example run in fig. 5.7.

The results demonstrate that the context detector is highly effective at identifying the current mode, performing nearly as well as the ideal context detector. Since HAC is capable of solving the Pendulum environment, when paired with a perfect context detector, it can efficiently handle the two modes with relatively few training episodes. In the Pendulum environment, our architecture significantly slows down learning, even though it successfully detects mode changes. A potential explanation for this slowdown is that experiences are occasionally assigned to the wrong mode, which can have long-lasting effects on the learning process. This misattribution occurs more frequently at the start of training but persists, though less frequently, throughout training—especially when the mode has shifted, and the scores are still being gradually updated.

The results obtained in the Ant Reacher environment provide even clearer evidence of our architec-



(a) Success rate

(b) Episodic length

Figure 5.6: Evaluation of our architecture against a perfect context detector using two metrics, success rate and episodic length in the pendulum environment. Results averaged across 5 different randomly-seeded runs with confidence intervals of 95%

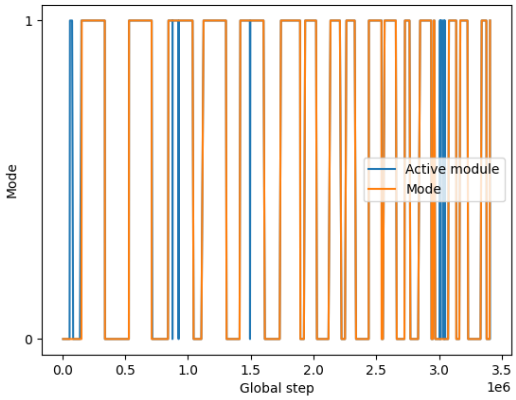
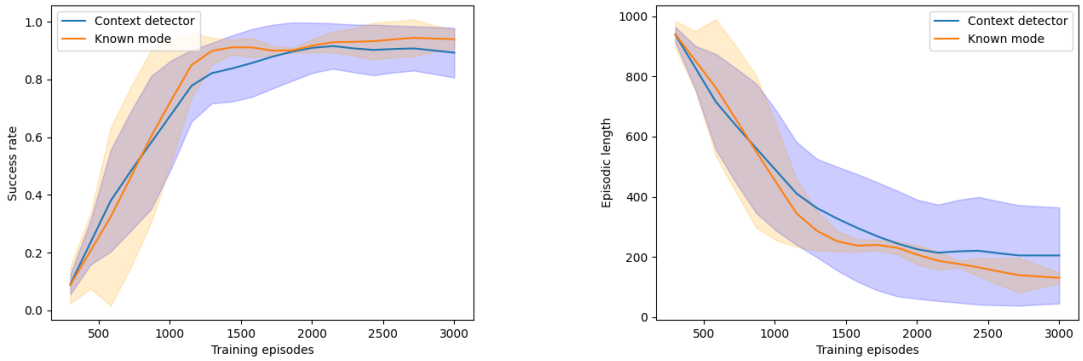


Figure 5.7: Example of how the predicted mode and the real mode evolved during a single run. Whenever blue is visible, the context detector was incorrect. The mode doesn't alternate in even intervals because the x-axis contains the global step, and as the agent becomes more competent at achieving the goal each episode becomes shorter and 200 episodes span less steps.

ture's success. In Ant Reacher, our architecture performs almost as well as the perfect context detector. Given that Ant Reacher is a more complex environment than Pendulum, even HAC with knowledge of the correct mode requires time to learn. Consequently, the learning slowdown observed in the Pendulum environment is no longer noticeable.

The main difference between our architecture and the baseline lies in the size of the confidence interval. We observe higher variance in our architecture's results, which may reflect greater instability during training. As previously mentioned, even temporary failures in the context detector immediately impact learning, making it understandable that incorporating the context detector introduces additional



(a) Success rate

(b) Episodic length

Figure 5.8: Evaluation of our architecture against a perfect context detector using two metrics, success rate and episodic length in the ant reacher environment. Results averaged across 5 different randomly-seeded runs with confidence intervals of 95%.

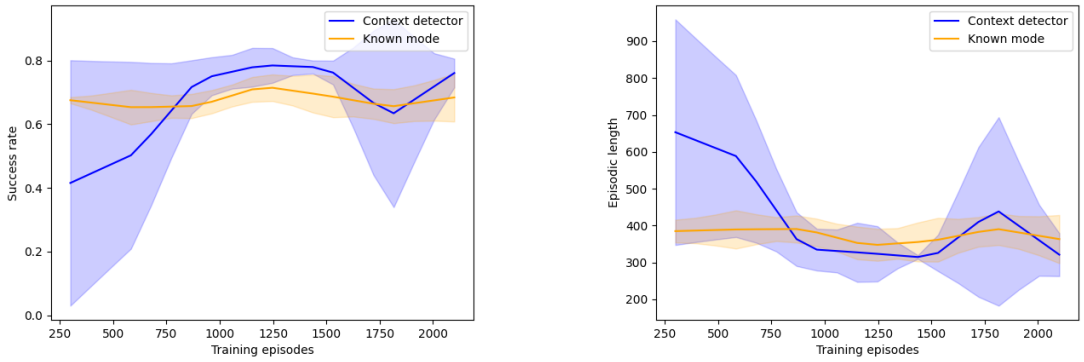
variance, even when it is accurate most of the time.

Furthermore, HAC's ability to deal with noise could also be improved. Some state-of-the-art techniques that have proven effective include limiting the size of the updates to ensure stable learning. These include methods like TRPO [35] and PPO [49], which are on-policy but could be adapted for off-policy algorithms. Another effective approach is clipped double Q-learning [50], which has already been successfully combined with an actor-critic by Haarnoja *et al.* [38]. These potential improvements are further discussed in section 6.1.

We now evaluate our architecture in the Pendulum environment with 3 modes. The 3 modes introduced consist of 3 different masses for the pendulum. The agent is tested in the same setup of mixing training and testing episodes throughout 3000 training episodes. Testing phases now last for 600 episodes because since there are 3 modes, 600 episodes are needed to test the agent in each one. We present the episodic length and success rate achieved for each testing phase throughout training in fig. 5.9. Also, we present an example of the evolution of the real mode and the module predicted in fig. 5.10 to support our discussion.

These results highlight two significant issues with our architecture. First, while the context detector correctly identifies mode 1, where the pendulum is lighter, it struggles to differentiate between modes 0 and 2, which are too similar for accurate distinction. Although it's possible for a single policy to solve both modes if they are similar enough, the drop in success rate to around 70% suggests that one of these modes presents a challenge for the agent. We conclude that it is possible for two modes with different optimal policies to not be distinguishable by our context detector.

The second issue is that even the perfect context detector fails to achieve a 100% success rate.



(a) Success rate

(b) Episodic length

Figure 5.9: Evaluation of our architecture against a perfect context detector using two metrics, success rate and episodic length in the pendulum environment with 3 modes. Results averaged across 5 different randomly-seeded runs with confidence intervals of 95%

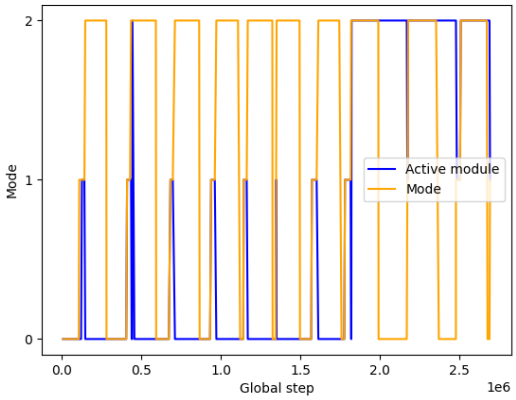
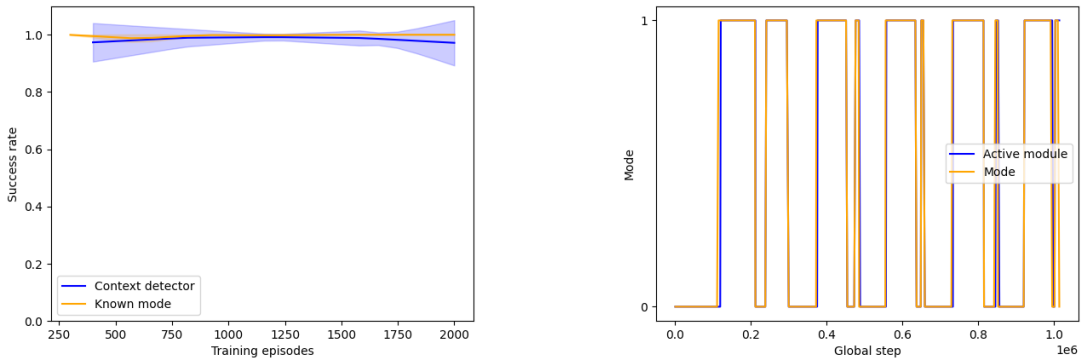


Figure 5.10: Example of how the predicted mode and the real mode evolved during a single run in the pendulum environment with 3 modes. Whenever blue is visible, the context detector was incorrect.

This suggests that simply switching the policy at the lower layer is not enough to fully adapt to all three modes, even when the agent is aware that the environment has changed. In non-stationary environments where more than one layer's policy needs to be adapted across modes, CoHAC's ability to isolate non-stationarity hinders the agent's full adaptation. While this is a limitation, we recognize that differences in modes involving various temporal scales may generally reduce the effectiveness of hierarchical approaches.

To finalize our context detector validation, we evaluate the success of CoHAC when the total number of modes is not available a priori. In previous experiments, the context detector was provided with the environment's number of modes m, and after initializing m modules, candidate modules were no longer

compared with the active module at every n time steps. In the following experiment, the context detector was informed that there were 5 modes in the environment, whereas only two actually existed. The results presented in fig. 5.11 show that the context detector correctly predicted only the two existing modes, ignoring the extra modules that it had access to. This demonstrates that our method effectively identifies when it is necessary to initialize a new module and when it is not. Although achieving these results required experimenting with more hyperparameter configurations, proving to be a more difficult task, it confirms that providing the total number of modes to the context detector is not mandatory.



(a) Success rate

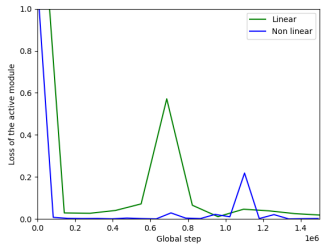
(b) Active module vs correct mode

Figure 5.11: Evaluation of our architecture against a perfect context detector in the pendulum environment when the total number of modes is not provided to the context detector. Results on the left averaged across 5 different randomly-seeded runs with confidence intervals of 95%. Graphic on the right represents an example of the predicted mode vs the correct mode across a single run. Whenever blue is visible, the context detector was incorrect.

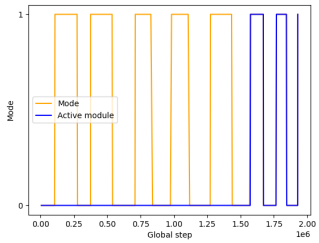
5.4.2 Design choices validation

The first design choice we evaluate is the usage of an activation function in the neural network responsible for predicting the next mode. If no activation function is employed, the predicted state is simply a linear transformation of the concatenated state and action. To highlight the differences between the two, we present in fig. 5.12 the comparison between the loss values of the active modules throughout training, both in the Pendulum and the Ant Reacher environment, as well as the visual representation of the predicted module during a sample run of the non-linear case in the Pendulum environment.

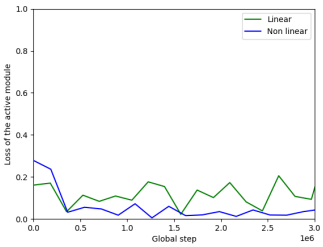
As expected, the results show that neural networks are more expressive and their predicted states are closer to the real states encountered, generating a lower loss value. However, the difference is very small and the neural networks' higher expressiveness might be a downgrade in some cases. Fig. 5.12(b) shows that sometimes a single module might succeed in learning more than one mode of the environment, a problem also encountered by Doya *et al.* [17]. For this reason, we prefer to avoid using



(a) Pendulum environment



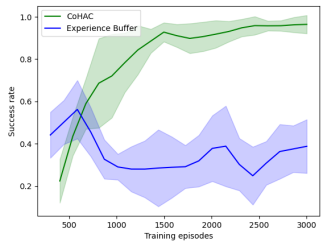
(b) Sample run in the Pendulum environment



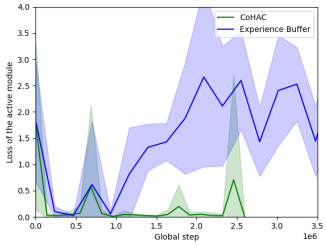
(c) Ant Reacher environment

Figure 5.12: Comparison between the loss of the active module when using neural networks vs linear transformation, in both the Pendulum and the Ant Reacher environment. Middle figure shows the predicted module by a non-linear network in a sample run in the Pendulum environment. Losses averaged across 3 different randomly-seeded runs.

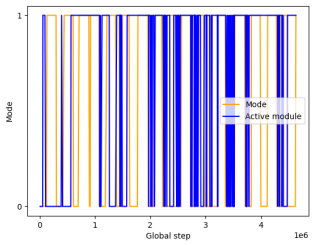
activation functions, and all of our experiments were executed using a linear transformation to predict the state. We consider than in environments where transitions between states are more complex, neural networks might become the better option.



(a) Success rate



(b) Loss of the active module



(c) Predicted mode example

Figure 5.13: Comparison between our chosen implementation and a context detector with an experience buffer similar to DDPG in the Pendulum environment. We present the success rate, the loss of the active module during training and an example of the predicted mode throughout a single run by the alternative implementation. Results averaged over across 3 different randomly-seeded runs with confidence intervals of 95%.

Another possible different path for the implementation of our context detector would be to store experiences in an experience buffer, in a similar way to DDPG [41]. In this alternative implementation, the last n steps still would be used for training uninitialized modules. The difference is, after initialization, all experiences assigned to each module would be kept in that module's respective experience buffer and reused for training, instead of discarded. Results presented in fig. 5.13 show that in practice this approach does not work. We hypothesize that mistakes in attributing experiences have much more long-lasting effects in this implementation, since experiences are not immediately discarded. Also, a learning context detector will assign more incorrect experiences and tend to get worst, instead of better.

We make this conclusion supported by fig. 5.13(b), where we can observe that the loss value is initially competitive with our chosen implementation and later becomes much worse.

Finally, the other alternative approach we explored was passing the correct mode as input to the actor-critic, instead of having an actor-critic with totally distinct parameters for each mode of the environment. This approach would greatly reduce the number of parameters that have to be trained and would increase the simplicity of our approach. Instead of passing the mode directly as a single parameter, we added a one-hot vector to the input of the actor-critic networks. Nevertheless, we compared the 2 approaches by disclosing the mode to the agent, so that we can directly evaluate the ability of each architecture to adapt. The results in fig. 5.14 show that even passing a one-hot vector is not expressive enough, and this approach can only achieve results similar to the ones achieved by HAC in fig. 5.4.

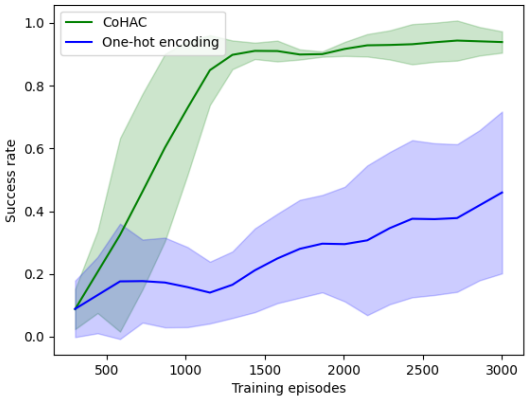


Figure 5.14: Comparison in the Ant Reacher environment between having one actor-critic for each module and having a single actor critic that receives the mode as a one-hot vector. Both architectures have access to the correct mode. Results averaged over across 5 different randomly-seeded runs with confidence intervals of 95%.

6

Conclusion

Contents

In this work, we explored reinforcement learning in non-stationary environments, that alternate between a set of modes of operation. Previous work [17, 18, 22] had already approached non-stationarity, but mostly in discrete environments. We introduced CoHAC, a hierarchical [12] agent that learns the environment dynamics in order to detect the current context, and simultaneously develops different policies for each mode of operation. We leveraged the benefits of Hierarchical Reinforcement Learning (HRL) in dealing with high-dimensionality to create an architecture that performs well in non-stationary environments with continuous state and action spaces, as well as sparse rewards, similar to the ones encountered in the real world.

To evaluate CoHAC in the desired setting, we added different modes of operation to typical control environments [45, 47] used by HRL literature. Also, we had to refactor the deprecated HAC code and implement it using the PyTorch framework. We used these environments to show that CoHAC outperformed multiple baselines. Our model was able to deal with non-stationarity while maintaining most of the system unaware to the mode changes. CoHAC also learned a different policy for each mode, and therefore did not need to endlessly adapt to the mode changes, in turn converging to multiple stationary policies. Furthermore, our model was capable of learning end-to-end, meaning it was able to learn the dynamics of the environment and the optimal policies to achieve the goal simultaneously without any pre-training needed.

Additionally, we validated our context detector by comparing our architecture to an agent that is directly given the current mode of the environment, and confirm that CoHAC performs similarly to this all-knowing agent. On the other hand, these results bring out some limitations in our architecture. It is possible for two modes to require different policies to achieve the goal and be indistinguishable to our context detector. Besides, some types of non-stationarity might require to change the policy of more than one layer for full adaptation. We conclude that even within the framework of modes of operation, non-stationary environments have a high variability and CoHAC is not a one-size fits all solution and is at its best when the modes in the environment alter the low-level mechanics of the environment.

With this work, we highlighted the ability of HRL to plan for a task at different time granularity and how plans can be reused in similar tasks. Unlike previous literature, we applied this ability to non-stationary environments and provided a starting point for environments that present a combination of obstacles from two previously separated fields.

6.1 Future Work

CoHAC uses a context detector that explicitly identifies the current mode of the environment and feeds it to the reinforcement learning agent. However, it would be possible to implicitly infer the current context by using a Long Short-Term Memory (LSTM) [51]. LSTMs are a type of recurrent neural network that introduce memory cells that can maintain information over extended periods of time. This way, outputs of this network depend on previous inputs, which could be enough for the agent to detect the correct mode. This method is not straight-forward to introduce in our architecture, because HAC [12] stores and learns from individual experiences while their order is discarded. Nevertheless, LSTMs have been used successfully with on-policy methods on Partially Observable MDPs [52] where the agent does not have access to full observations, using recent experiences to infer the hidden state. Testing their effectiveness at discovering hidden modes could be an interesting research direction.

One of the characteristics of CoHAC identified in our experiments is its high variance in the results, which may indicate a lack of stability during training. On one hand, this instability can be attributed to assigning experiences to the wrong HAC module, in times when the context detector is either still learning or when the mode has changed recently and the context detector is yet to react. In our approach, we tried to maintain HAC and the context detector as separated as possible, to reduce dependency and allow for one of the methods to be replaced without affecting the other. However, assigning experiences in HAC at the same time as the context detector, after n steps and after evaluating each module against the full update window might reduce the amount of wrongly assigned experiences.

On the other hand, HAC itself can be improved in terms of stability. Clipped double Q-learning [50] is a technique introduced to mitigate the overestimation bias observed in Q-learning methods. Instead of one, two independent Q-networks are used to estimate the Q-values and when performing updates, the minimum Q-value between the two is used. Therefore, the updates performed are more conservative and are less likely to be overly optimistic, leading to more robust and stable learning. This technique has been used successfully with an actor critic [38] and would be straight-forward to implement in HAC.

In section 5.4.2, we analyzed the performance of feeding the mode as input to the actor-critic instead of having m actor-critics in the adaptable layer, with m being the number of modes in the environment. We showed that a single network is not expressive enough to deal with the differences in mode, but perhaps some parameters can be shared. This way, we reduce the number of parameters that have to be trained but can still choose the network used instead of using a one-hot key encoding. We leave what can be shared between the actor-critic responsible for the same layer and different modes as future work.

Finally, in all our experiments, the only layer that was aware of the context was the low-level layer. The context detector must always learn at the finest time granularity because it needs to observe the transitions of the environment. Still, there is no reason to limit the adaptable layer to the lowest level.

Different modes of operation may require different high level plans that can be achieved in similar ways by the sub-layers. If our architecture could successfully encapsulate non-stationarity in higher layers as it did in lower layer, CoHAC would become suitable to a wider range of non-stationary environments.

Bibliography

[1] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

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

[3] T. Miki, J. Lee, J. Hwangbo, L. Wellhausen, V. Koltun, and M. Hutter, "Learning robust perceptive locomotion for quadrupedal robots in the wild," *Science robotics*, vol. 7, no. 62, p. eabk2822, 2022.

[4] J. Chen, B. Yuan, and M. Tomizuka, "Model-free deep reinforcement learning for urban autonomous driving," in *2019 IEEE intelligent transportation systems conference (ITSC)*. IEEE, 2019, pp. 2765–2771.

[5] S. Choi, D.-Y. Yeung, and N. Zhang, "An environment model for nonstationary reinforcement learning," *Advances in neural information processing systems*, vol. 12, 1999.

[6] P.-L. Bacon, J. Harb, and D. Precup, "The option-critic architecture," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, 2017.

[7] A. S. Vezhnevets, S. Osindero, T. Schaul, N. Heess, M. Jaderberg, D. Silver, and K. Kavukcuoglu, "FeUdal networks for hierarchical reinforcement learning," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. PMLR, 06–11 Aug 2017, pp. 3540–3549. [Online]. Available: https://proceedings.mlr.press/v70/vezhnevets17a.html

[8] O. Nachum, S. S. Gu, H. Lee, and S. Levine, "Data-efficient hierarchical reinforcement learning," *Advances in neural information processing systems*, vol. 31, 2018.

[9] T. D. Kulkarni, K. Narasimhan, A. Saeedi, and J. Tenenbaum, "Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation," in *Advances in Neural*

Information Processing Systems, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., vol. 29. Curran Associates, Inc., 2016. [Online]. Available: https://proceedings.neurips.cc/paper/2016/file/f442d33fa06832082290ad8544a8da27-Paper.pdf

[10] K. Frans, J. Ho, X. Chen, P. Abbeel, and J. Schulman, "Meta learning shared hierarchies," *arXiv preprint arXiv:1710.09767*, 2017.

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

[12] A. Levy, G. Konidaris, R. Platt, and K. Saenko, "Learning multi-level hierarchies with hindsight," *arXiv preprint arXiv:1712.00948*, 2017.

[13] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press, 2018.

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

[15] I. Grondman, L. Busoniu, G. A. Lopes, and R. Babuska, "A survey of actor-critic reinforcement learning: Standard and natural policy gradients," *IEEE Transactions on Systems, Man, and Cybernetics, part C (applications and reviews)*, vol. 42, no. 6, pp. 1291–1307, 2012.

[16] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *International conference on machine learning*. Pmlr, 2014, pp. 387–395.

[17] K. Doya, K. Samejima, K.-i. Katagiri, and M. Kawato, "Multiple model-based reinforcement learning," *Neural computation*, vol. 14, no. 6, pp. 1347–1369, 2002.

[18] B. C. Da Silva, E. W. Basso, A. L. Bazzan, and P. M. Engel, "Dealing with non-stationary environments using context detection," in *Proceedings of the 23rd international conference on Machine learning*, 2006, pp. 217–224.

[19] E. Hadoux, A. Beynier, and P. Weng, "Sequential decision-making under non-stationary environments via sequential change-point detection," in *Learning over multiple contexts (LMCE)*, 2014.

[20] E. S. Page, "Continuous inspection schemes," *Biometrika*, vol. 41, no. 1/2, pp. 100–115, 1954.

[21] T. Banerjee, M. Liu, and J. P. How, "Quickest change detection approach to optimal control in markov decision processes with model changes," in *2017 American control conference (ACC)*. IEEE, 2017, pp. 399–405.

[22] S. Padakandla, P. KJ, and S. Bhatnagar, “Reinforcement learning algorithm for non-stationary environments,” *Applied Intelligence*, vol. 50, no. 11, pp. 3590–3606, 2020.

[23] P. KJ, N. Singh, P. Dayama, A. Agarwal, and V. Pandit, “Change point detection for compositional multivariate data,” *Applied Intelligence*, vol. 52, no. 2, pp. 1930–1955, 2022.

[24] M. Kaisers and K. Tuyls, “Frequency adjusted multi-agent q-learning,” in *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, 2010, pp. 309–316.

[25] S. Abdallah and M. Kaisers, “Addressing environment non-stationarity by repeating q-learning updates,” *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 1582–1612, 2016.

[26] R. S. Sutton, D. Precup, and S. Singh, “Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning,” *Artificial intelligence*, vol. 112, no. 1-2, pp. 181–211, 1999.

[27] I. Menache, S. Mannor, and N. Shimkin, “Q-cut—dynamic discovery of sub-goals in reinforcement learning,” in *European conference on machine learning*. Springer, 2002, pp. 295–306.

[28] A. McGovern and A. G. Barto, “Automatic discovery of subgoals in reinforcement learning using diverse density,” *Computer Science Department Faculty Publication Series*, vol. 8, 2001.

[29] T. G. Dietterich, R. H. Lathrop, and T. Lozano-Pérez, “Solving the multiple instance problem with axis-parallel rectangles,” *Artificial intelligence*, vol. 89, no. 1-2, pp. 31–71, 1997.

[30] J. Tsitsiklis and B. Van Roy, “Analysis of temporal-diffference learning with function approximation,” *Advances in neural information processing systems*, vol. 9, 1996.

[31] L.-J. Lin, “Self-improving reactive agents based on reinforcement learning, planning and teaching,” *Machine learning*, vol. 8, pp. 293–321, 1992.

[32] V. Konda and J. Tsitsiklis, “Actor-critic algorithms,” *Advances in neural information processing systems*, vol. 12, 1999.

[33] J. Harb, P.-L. Bacon, M. Klissarov, and D. Precup, “When waiting is not an option: Learning options with a deliberation cost,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, 2018.

[34] G. D. Konidaris and A. G. Barto, “Building portable options: Skill transfer in reinforcement learning.” in *IJCAI*, vol. 7, 2007, pp. 895–900.

[35] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, “Trust region policy optimization,” in *Proceedings of the 32nd International Conference on Machine Learning*, ser. Proceedings of

Machine Learning Research, F. Bach and D. Blei, Eds., vol. 37. Lille, France: PMLR, 07–09 Jul 2015, pp. 1889–1897. [Online]. Available: https://proceedings.mlr.press/v37/schulman15.html

[36] K. Gregor, D. J. Rezende, and D. Wierstra, "Variational intrinsic control," *arXiv preprint arXiv:1611.07507*, 2016.

[37] B. Eysenbach, A. Gupta, J. Ibarz, and S. Levine, "Diversity is all you need: Learning skills without a reward function," *arXiv preprint arXiv:1802.06070*, 2018.

[38] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *International conference on machine learning*. PMLR, 2018, pp. 1861–1870.

[39] P. Dayan and G. E. Hinton, "Feudal reinforcement learning," *Advances in neural information processing systems*, vol. 5, 1992.

[40] T. Schaul, D. Horgan, K. Gregor, and D. Silver, "Universal value function approximators," in *International conference on machine learning*. PMLR, 2015, pp. 1312–1320.

[41] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[42] S. Ioffe, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[43] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, O. Pieter Abbeel, and W. Zaremba, "Hindsight experience replay," *Advances in neural information processing systems*, vol. 30, 2017.

[44] P. J. Huber, "Robust estimation of a location parameter," in *Breakthroughs in statistics: Methodology and distribution*. Springer, 1992, pp. 492–518.

[45] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," https://github.com/openai/gym, 2016.

[46] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.

[47] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ international conference on intelligent robots and systems*. IEEE, 2012, pp. 5026–5033.

[48] D. P. Kingma, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

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

[50] S. Fujimoto, H. Hoof, and D. Meger, “Addressing function approximation error in actor-critic methods,” in *International conference on machine learning*. PMLR, 2018, pp. 1587–1596.

[51] S. Hochreiter, “Long short-term memory,” *Neural Computation MIT-Press*, 1997.

[52] V. Mnih, “Asynchronous methods for deep reinforcement learning,” *arXiv preprint arXiv:1602.01783*, 2016.