Piecewise Linear Parametrization of Policies for Interpretable Deep Reinforcement Learning

Maxime Wabartha*
School of Computer Science
McGill University and Mila
maxime.wabartha@mail.mcgill.ca

Joelle Pineau
School of Computer Science
McGill University, Mila and Meta AI
jpineau@cs.mcgill.ca

Abstract

Learning inherently interpretable policies is a central challenge in the path to developing autonomous agents that humans can trust. We argue for the use of policies that are piecewise-linear. We carefully study to what extent they can retain the interpretable properties of linear policies while performing competitively with neural baselines. In particular, we propose the HyperCombinator (HC), a piecewise-linear neural architecture expressing a policy with a controllably small number of sub-policies. Each sub-policy is linear with respect to interpretable features, shedding light on the agent's decision process without needing an additional explanation model. We evaluate HC policies in control and navigation experiments, visualize the improved interpretability of the agent and highlight its trade-off with performance.

1 Introduction

Machine learning models are becoming increasingly opaque decision makers. They are untrustworthy in many situations, either because their predictions are unstable [1, 2], fail to generalize [3], the actual sub-policies differ from the expected one [4, 5], or because one cannot easily extract the logic behind their decisions. In our quest for ever improving performance, the transparency and explainability of AI models might soon become an indispensable component of their democratization, amid a growing oversight by regulatory bodies [6, 7, 8, 9]. But interpretability is also a fascinating scientific topic in itself, giving us the opportunity to strengthen our understanding of our models and their effects. Therefore, it is important to systematically explore the methods that let us interpret machine learning models. Deep reinforcement learning (DRL) is a particularly interesting domain in which to study interpretability. In fact, a precise description of how policies interact with an environment can inform other algorithms designed to improve the fairness, accountability, or transparency of the deployed model [10]. Transparent policies could also help better diagnose the agent's interactions, which can be helpful if it appears to malfunction.

In this work, we examine the viability of piecewise-linear policies as a beneficial compromise between interpretability and performance in DRL. Given that policies expressing a few linear sub-policies were already shown to be performant [11], we focus on extending the analysis of piecewise-linear policies in the context of interpretability. To that end, we propose the HyperCombinator, a neural architecture that parametrizes a piecewise-linear policy with a controllably small number of linear sub-policies. The HC agent, which can be thought of as a mixture of linear experts [12], selects a linear sub-policy at every interaction with the environment, an important property missing from previous works. Therefore, for a given choice of sub-policy, it becomes possible to transparently analyze the computation leading to each decision.

*corresponding author

XAI in Action: Past, Present, and Future Applications @ NeurIPS 2023.

Our contributions are the following: (1) we characterize properties that an interpretable piecewise-linear policy should ideally satisfy, (2) we present the HyperCombinator, an actor architecture that implements a piecewise-linear policy with a specifiable number of linear sub-policies and compatible with most RL algorithms, (3) we characterize the interpretability properties inherited by HC, (4) we extensively evaluate the model on control and navigation tasks and observe a sustained performance of the model despite its greatly reduced expressivity. In addition, (5) we leverage the two levels of interpretability of HC to propose two visualizations. The first one lists the different ways the policy can react to an input by cataloging *exhaustively* each linear sub-policy and its coefficients into a table. The second condenses the sequence of decisions into the sequence of sub-policies used, surfacing the temporal abstractions emerging from the task.

2 From linear to piecewise-linear policies

This paper is set in the context of DRL, about which we recall the general notions in Appendix A. We aim to achieve functional interpretability by reducing the size of the class that the policy belongs to [13]. We start from linear policies. We write them as $\tilde{\pi}(x) = \langle \theta, x \rangle$, where θ designates the *linear coefficients*, the input x is assumed interpretable, and $\langle A, b \rangle$ is the dot product (resp. matrix-vector product) between A and b if A is a vector (resp. a matrix) and b a vector. Some environments require to pass $\tilde{\pi}(x)$ through a non-linearity μ to match the action space, such that the final policy is $\pi = \mu \circ \tilde{\pi}$. Linear policies have several desirable interpretability properties, such as being summarizable by their coefficients, or having each coefficient explicitly quantify the influence of a given feature and expressing the same valid explanation for all inputs. However, their flexibility is generally not sufficient to handle complex tasks [14]. We therefore consider the larger class of piecewise-linear policies $\tilde{\pi} : \mathcal{X} \to \mathcal{A}$ with:

$$\tilde{\pi}(x) = \begin{cases} \langle \theta_{\omega_0}, x \rangle & \text{if } x \in \omega_0 \\ \langle \theta_{\omega_1}, x \rangle & \text{if } x \in \omega_1 \\ \dots & \\ \langle \theta_{\omega_{K-1}}, x \rangle & \text{if } x \in \omega_{K-1} \end{cases}, \tag{1}$$

where K is the number of linear sub-functions, and where the partition of the input space $\Omega = \{\omega_0, \dots, \omega_{K-1}\}$, the set of linear coefficients $\theta = \{\theta_\omega, \omega \in \Omega\}$ and the function $a : \mathcal{X} \to \Omega$ mapping an input to the corresponding subset ω describe $\tilde{\pi}$. Therefore, $\tilde{\pi}$ is locally linear, and some interpretability properties of linear policies carry over. However, both the value of K and the complexity of Ω and a influence the interpretability of $\tilde{\pi}$.

MLPs using only linear layers (including convolutional or layer normalization layers) and ReLU activations [15, 16] already offer a piecewise-linear parametrization of policies [17] learnable with gradient descent. Yet, there is no simple way to control the number of linear sub-policies that they express, impeding the interpretability of the policy. This in turn implies that each sub-policy is only applied to few inputs, limiting the generalization of the explanations provided by the linear coefficients. The rightmost plot of Fig. 1 illustrates the induced partition of an MLP, a concept more precisely analyzed in [18]. Restricting the number of linear sub-functions while maintaining a high level of performance and interpretability remains a major challenge, that we explore next.

3 Methods

We begin by formulating properties that a deployed, interpretable policy should ideally satisfy.

(P1) Sub-policy transparency: input-to-output sub-policy computation is easy-to-understand.
(P2) Assignation transparency: input to sub-policy choice computation is easy-to-understand.
(P3) Separation: the policy uses a single sub-policy at each interaction (training and evaluation).
(P4) Counterfactual reasoning: we can compare the sub-policy taken to the other possibilities.
(P5) Generalization: the sub-policies generalize to similar inputs.
(P6) Causality: we want to justify the causal influence of each feature *w.r.t.* the task success.

Given the subjective nature of interpretability, a single definition might not be sufficient [10]. To satisfy as many of these properties as possible, we design the Hypercombinator, a piecewise-linear neural architecture with a small, controllable number of linear sub-functions.

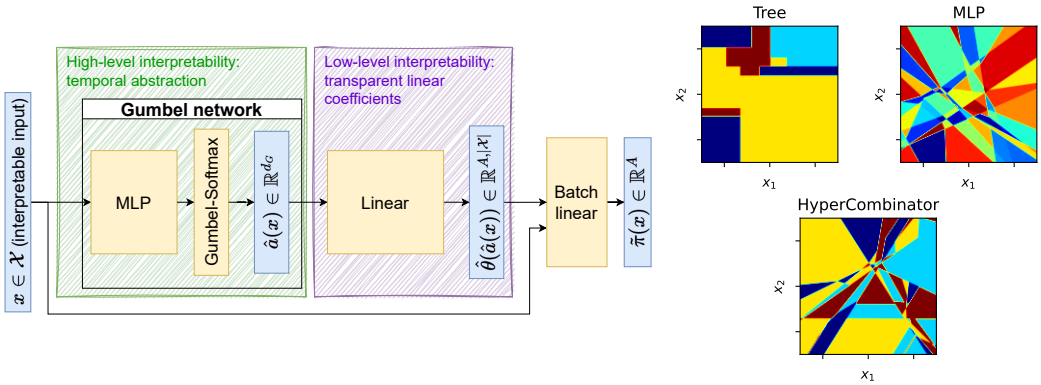

Figure 1: (Left) Proposed actor architecture. Layers are colored in yellow, vectors in blue. (Right) Intuition of the partition of a 2D space into linear regions by different models. Each polygon corresponds to a distinct linear region, and each color to a set of unique linear coefficients.

3.1 Proposed policy architecture

Our starting policy $\tilde{\pi}$ is expressed by an MLP, realizing a piecewise-linear function with respect to an input $x \in \mathbb{R}^n$ assumed *interpretable*. Following Sec. 2, $\tilde{\pi}(x)$ can be decomposed as $\langle \theta_{a(x)}, x \rangle$, where $a : \mathbb{R}^n \to \Omega$ assigns an input x to a subset of the partition Ω induced by $\tilde{\pi}$.

Parametrization We focus on reducing the number of sub-policies while retaining the complexity of Ω. We model θ as a linear function $\hat{\theta} : \Omega \to \mathbb{R}^n$. Secondly, we model a explicitly by an MLP followed by a Gumbel-Softmax layer [19, 20], and refer to the composition of both as a Gumbel network $\hat{a}$. The Gumbel-Softmax is typically used to continuously learn a categorical variable. Here, it predicts to which sub-policy should an input x be mapped. Furthermore, we use the straight-through estimator [21] to force $\hat{a}$ to produce strict assignations, *i.e.* one-hot encodings in the forward pass while being trainable with gradient descent. Any activation function in the Gumbel network will lead to a piecewise-linear function thanks to the straight-through estimator. This architecture is the HyperCombinator, illustrated in Fig. 1 along with the partition it induces.

This formulation is advantageous for several reasons. First, the MLP preceding the Gumbel-Softmax induces the partition of the input space which π inherits, thus retaining some of the predictive flexibility of NNs. Second, we explicitly control the number of unique sub-policies of π through the dimension d_G of the Gumbel-Softmax layer. Moreover, the Gumbel network $\hat{a}$ is piecewise-constant thanks to the straight-through estimator. Therefore, π interacts with the environment only through one of the d_G sub-policies, on the contrary to previous work [11]. Finally, we note that HC policies are compatible with any RL algorithm including an actor, as long as there are no competing assumptions on the structure of the policy. We provide further details regarding the implementation of the HyperCombinator in Appendix D.1.

HC policies are usually not continuous at the border between linear regions, unlike MLPs. We do not find to be a problem for stability in practice (Appendix D.7). Moreover, reducing the width or the height of the NN is an alternative and more direct way to reduce the number of unique coefficients. Yet, the number of linear sub-functions grows very quickly with the size of the architecture, both theoretically and empirically (see Appendix B for an in-depth analysis).

3.2 Interpretability characterization of the HyperCombinator architecture

HC policies satisfy several structural properties. First, $\tilde{\pi}$ is linear in the interpretable input x (P1), giving transparent access to the computation of the decision, given the sub-policy. Second, the policy is piecewise-linear not only during evaluation but also during training thanks to the use of the straight-through estimator, which guarantees that our approach stays interpretable for all the environment interactions (P3). Third, the number of unique sub-policies of the policy is set by the hyperparameter d_G (P4). The few sub-policies are thus in practice constrained to generalize across a wide range of states, which implies P5. This is not true for MLPs, due to their huge number of linear regions and unique coefficients. MLPs additionally do not satisfy P4.

Our approach is *locally interpretable* since the explanations of a sub-policy are limited to its domain of application; our approach does not satisfy P2. The HyperCombinator also does not provide feature importance or any notion of causal explanations and does not satisfy P6. Indeed, just like MLPs, a feature x_i of x can be involved in the computation of all the coefficients $\hat{\theta}(\hat{a}(x))$ and simultaneously have its linear coefficient $\hat{\theta}(\hat{a}(x))_i$ be 0. In other words, our approach can be understood as *interpretable conditioned on the choice of a sub-policy*.

Two levels of interpretability In summary, a HC policy can perhaps best be understood as two consecutive modules, each corresponding to a different level of interpretability. The first module selects a linear sub-policy given an input, while the second retrieves the linear coefficients of the selected sub-policy and applies them to the input. The second module provides the exact linear coefficients that are used by the policy to interact with the environment, quantifying the influence of each interpretable feature on the chosen action (*low-level interpretability*). Simultaneously, one can observe the sequence of choices of sub-policies made by an agent in order to analyze its behavior, thanks to the first module of $\tilde{\pi}$. The forced re-use of sub-policies leads to the emergence of cycles and temporal abstractions highlighting how the agent solves the task (*high-level interpretability*). We illustrate the complementary low-level and high-level interpretability of HC in Fig. 1 (left), using respectively purple and green.

4 Experiments

4.1 Control

We evaluate how well HC policies can control proprioceptive variables such as the joints of a robot through the DeepMind Control Suite benchmark [22]. We aim to (1) ensure that the HC architecture can be trained to solve control tasks, (2) investigate the improved interpretability of the agent and (3) compare the performance of HC actors to their baseline algorithm using their usual neural policy. We combine SAC [23] with a HC actor. In other terms, $\tilde{\pi}$ is linear *w.r.t.* the interpretable proprioceptive features (Sec. A.1). We detail the chosen hyperparameters in Appendix D. We consider a variant of HC with 8 sub-policies (HC8) as well as one with 64 (HC64), more expressive but less interpretable due to the higher number of sub-policies.

Performance We perform an in-depth experimental analysis of the performance of piecewise-linear policies in DM Control, extending previous work [11], and present the results in Fig. 6. HC policies approach or match the performance of SAC in most environments. This limited drop in performance is remarkable in particular for HC8, which performs at most 3% worse than SAC in 57% of the tested environments (HC64: 78%) despite only having access to 8 linear sub-policies. We further analyze the performance of HC over all environments, as well as improved sample efficiency in select hard tasks in Appendix D.

Interpretability The low-level interpretability of HC actors, thanks to the linearity of the sub-policies, gives insights about the influence of the features in each decision. We can visualize the linear coefficients characterizing all the sub-policies of $\tilde{\pi}$ via a matrix view (Fig. 2, right). In the Cartpole swingup environment, we can precisely analyze how some sub-policies swing the cartpole up while others stabilize the cartpole once in an approximately straight position. We defer the precise analysis of this result to Appendix D.8. The discrete bottleneck introduced by the Gumbel network also lets us visualize the high-level interpretability of HC through the sequence of chosen sub-policies. We illustrate the emergence of a time-extended behavior in Fig. 2 (left), where the cyclical nature of the movement of a cheetah running can be observed in the sequence of sub-policies chosen by HC8. In contrast, SAC reuses the same linear coefficients in average 50 times over a similar 1000 timestep trajectory on Cheetah run, in spite of the cyclicity of the task (see further details in Appendix C).

4.2 Navigation

We want to discover if (1) HC actors can solve hard exploration mazes [24] when combined with dedicated algorithms and if (2) using HC actors in mazes can elicit temporal abstractions in the sub-policy sequence, similar to the control setting. The agent, a quadruped which state is

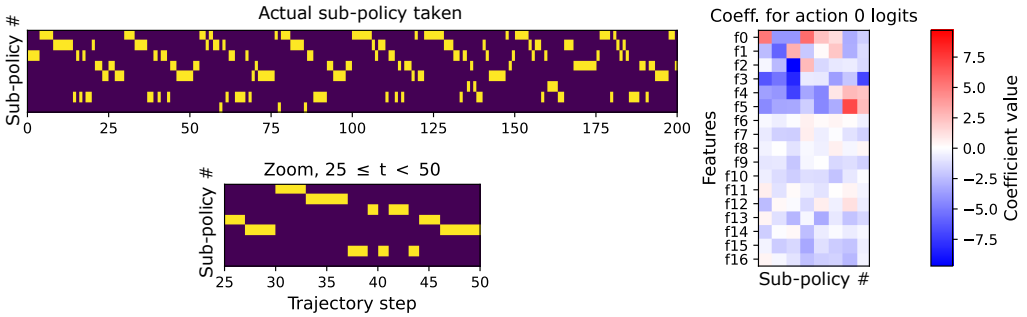

Figure 2: (Left) Sub-policy sequence, 200 first timesteps of HC8 on Cheetah run. A temporal abstraction emerges as HC8 reuses the same sub-policy for several timesteps before switching to a new one in a cyclical fashion. (Right) All sub-policies coefficients, for one action of $\tilde{\pi}$. We observe the similarities of sub-policies (2 and 3 have same-sign coefficients for features 1-5) and disparities (the same sub-policies have opposite sign *w.r.t.* feature 0, hence an opposite effect).

described by proprioceptive features, is provided with a goal state to reach. We extend RIS [25], a goal-conditioned DRL algorithm which encourages during training the policy to match the action predicted by a "prior" policy (a moving average of the actor) to reach an intermediate goal. We use the RIS algorithm with HC actors of 8, 16 and 64 sub-policies, and otherwise follow the same training and evaluation protocol. We detail it along with the hyperparameters in Appendix E.

Performance We observe in Fig. 15 that RIS as well as HC64 perform well in all the mazes, despite the difficulty of the task illustrated by the wide confidence intervals. Furthermore, HC8 and HC16 solve most of the time the U-shaped maze, and HC16 the S-shaped maze. All reach the goal in at least one of the seeds, showing that such architectures are not too constrained to express a goal-reaching policy using a dedicated navigation algorithm. Yet, HC actors require more interactions than RIS to solve the respective mazes. Their failure rate is overall higher, likely due to the higher difficulty of the task compared to DM Control. Solving the maze also requires high-level navigation and low-level control, whereas only the latter was previously required.

Interpretability We finally provide an example of a sequence of sub-policies followed by HC8 during one successful trajectory, in Fig. 3. This visualization emphasizes how a temporal abstraction emerges from the hierarchical nature of the HyperCombinator policy. Notably, the HC actor learns to alternate between sub-policies 0 and 1 to move straight, and between sub-policies 2 and 3 to rotate. This illustrates how the high-level interpretability of HC lets us identify behaviors followed by the agent to solve the maze. For instance, the usage of sub-policies 2 and 3 at the beginning of the trajectory indicates that the agent first needed to turn. A comparison with the video of the agent playing highlights it was initialized facing the wrong direction, justifying this first rotation. We analyze a wider set of sub-policy sequences in other mazes in Appendix E.

5 Related works

Closest to our work, [11] explore a very similar piecewise-linear policy architecture in the control setting, though not in the scope of interpretability. In particular, the partitioning induced by the policy is a soft one, whereas HC induces a hard partition which guarantees that only a single sub-policy is used at any interaction with the environment. Previous work also showed that linear [26] and discrete [27] policies performed surprisingly well in control environments, at the cost of a reduced flexibility and therefore a likely reduced ability to solve complex tasks. Other works focus on learning highly structured policies [28, 29], training a decision tree actor with imitation learning [30, 31, 32] on control tasks or leveraging abstract actions to solve mazes [33, 34, 35]. On the contrary, HC aims to learn functionally interpretable policies that handle both to low-level continuous control and high-level planning. Other works use non gradient-descent optimization methods to learn inherently interpretable policies [36, 37]. Researchers have also focused on

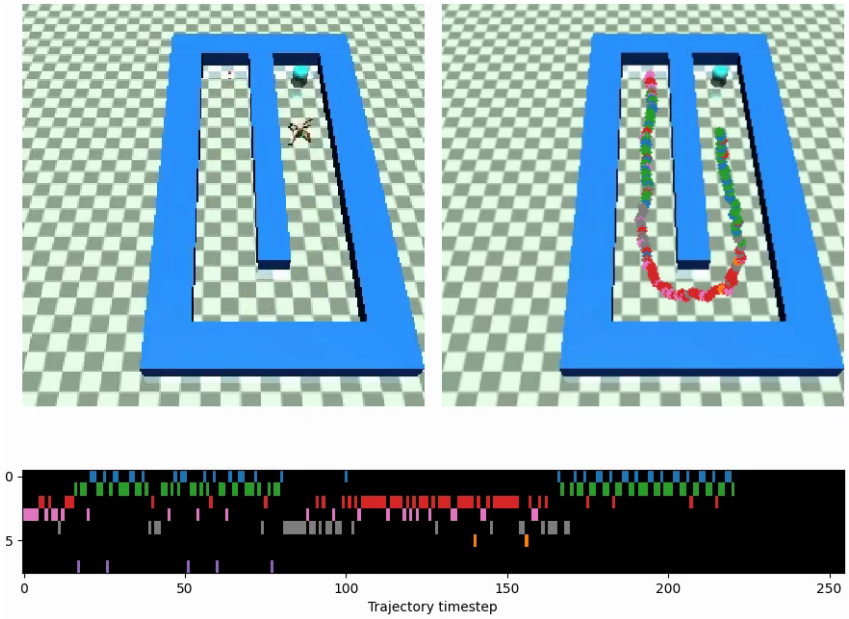

Figure 3: (Top left) The quadruped solving the maze. (Bottom) HC8 sub-policy sequence after 2M steps. We note the appearance of motifs (repetitions of subsequences of sub-policies) matching the behavior observed on the agent: rotation, then straight movement, then rotation to change corridor and a straight movement. (Top right) Trajectory of sub-policy choices superposed with the maze: each colored point refers to the sub-policy chosen at that location. The grouping of colors illustrate the empirical emergence of temporal abstractions.

improving the interpretability of neural policies defined on high-dimensional domains [38, 39], while we focus on interpretable proprioceptive states. Further from our work, there has been efforts to make the Q-function interpretable [40, 41]. Tangentially, a recent work studied the evolution of the linear regions of a ReLU MLP in deep reinforcement learning [42].

Several navigation methods incorporate a hierarchical component in the policy training, which can inform the policy architecture [24, 43, 44] or guide it during training [25]. We focus on making the policy more interpretable by building off the later method, restricting its expressivity. Options [45, 46] are another type of temporal abstraction. In comparison, our method does not need to define an initiation or termination set. Finally, HC policies can be thought of as a hypernetwork [47] that predicts the coefficients of a policy linear *w.r.t.* interpretable features.

6 Discussion

In this article, we studied whether piecewise-linear policies can be a suitable compromise between performance and interpretability in DRL. To this end, we presented the HyperCombinator architecture, an implementation of a piecewise-linear actor that combines a controllably small set of inherently interpretable sub-policies to interact with the environment in a transparent manner. We showed that in several control and navigation environments, the performance loss due to the reduced expressivity of our policy class was limited. Moreover, we illustrated how new visualizations, possible thanks to the two levels of interpretability provided by HC policies, give us insights on an agent's interactions with its environment. We also noticed the empirical emergence of temporal abstractions in the policy in both the control and select navigation settings.

This work could be improved in several ways. Currently, the assignation function learnt by the policy is not interpretable, which prevents explaining why a sub-policy was chosen. Moreover, the HC interacts transparently with the environment, but is not a causal architecture. So our policy ultimately does not fully answer "why" an action was taken, instead giving an interpretable decision conditioned on the sub-policy choice. Nonetheless, this is a significant step compared to current DRL algorithms, which are far from meeting the most basic standards of explainability.

Acknowledgments and Disclosure of Funding

We are grateful to the FRQNT (Fonds de Recherche du Québec - Nature et Technologies) for funding this research.

References

[1] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[2] MingYu Lu, Zachary Shahn, Daby Sow, Finale Doshi-Velez, and H Lehman Li-wei. Is deep reinforcement learning ready for practical applications in healthcare? a sensitivity analysis of duel-ddqn for hemodynamic management in sepsis patients. In *AMIA Annual Symposium Proceedings*, volume 2020, page 773. American Medical Informatics Association, 2020.

[3] Maxime Wabartha, Audrey Durand, Vincent Francois-Lavet, and Joelle Pineau. Handling black swan events in deep learning with diversely extrapolated neural networks. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pages 2140–2147, 2021.

[4] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in ai safety. *arXiv preprint arXiv:1606.06565*, 2016.

[5] Borja Ibarz, Jan Leike, Tobias Pohlen, Geoffrey Irving, Shane Legg, and Dario Amodei. Reward learning from human preferences and demonstrations in atari. *Advances in neural information processing systems*, 31, 2018.

[6] European Union. General data protection regulation, 2016.

[7] United States of America. Blueprint for an ai bill of rights: Making automated systems work for the american people, 2022. URL `https://www.whitehouse.gov/ostp/ai-bill-of-rights`.

[8] United Kingdom. Establishing a pro-innovation approach to regulating ai, 2023. URL `https://www.gov.uk/government/publications/establishing-a-pro-innovation-approach-to-regulating-ai/establishing-a-pro-innovation-approach-to-regulating-ai-policy-statement`.

[9] United Kingdom. A pro-innovation approach to ai regulation, 2023. URL `https://www.gov.uk/government/publications/ai-regulation-a-pro-innovation-approach`.

[10] Finale Doshi-Velez and Been Kim. Towards a rigorous science of interpretable machine learning. *arXiv preprint arXiv:1702.08608*, 2017.

[11] Riad Akrour, Filipe Veiga, Jan Peters, and Gerhard Neumann. Regularizing reinforcement learning with state abstraction. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 534–539. IEEE, 2018.

[12] Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

[13] Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature machine intelligence*, 1(5):206–215, 2019.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[15] Kunihiko Fukushima. Cognitron: A self-organizing multilayered neural network. *Biological cybernetics*, 20(3-4):121–136, 1975.

[16] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[17] Razvan Pascanu, Guido Montufar, and Yoshua Bengio. On the number of response regions of deep feed forward networks with piece-wise linear activations. In *2nd International Conference on Learning Representations, ICLR 2014*, 2014.

[18] Boris Hanin and David Rolnick. Deep relu networks have surprisingly few activation patterns. *Advances in neural information processing systems*, 32, 2019.

[19] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

[20] Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.

[21] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*, 2013.

[22] Yuval Tassa, Yotam Doron, Alistair Muldal, Tom Erez, Yazhe Li, Diego de Las Casas, David Budden, Abbas Abdolmaleki, Josh Merel, Andrew Lefrancq, et al. Deepmind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

[23] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018.

[24] Soroush Nasiriany, Vitchyr Pong, Steven Lin, and Sergey Levine. Planning with goal-conditioned policies. *Advances in Neural Information Processing Systems*, 32, 2019.

[25] Elliot Chane-Sane, Cordelia Schmid, and Ivan Laptev. Goal-conditioned reinforcement learning with imagined subgoals. In *International Conference on Machine Learning*, pages 1430–1440. PMLR, 2021.

[26] Aravind Rajeswaran, Kendall Lowrey, Emanuel V Todorov, and Sham M Kakade. Towards generalization and simplicity in continuous control. *Advances in Neural Information Processing Systems*, 30, 2017.

[27] Yunhao Tang and Shipra Agrawal. Discretizing continuous action space for on-policy optimization. In *Proceedings of the aaai conference on artificial intelligence*, volume 34, pages 5981–5988, 2020.

[28] Larry D Pyeatt, Adele E Howe, et al. Decision tree function approximation in reinforcement learning. In *Proceedings of the third international symposium on adaptive systems: evolutionary computation and probabilistic graphical models*, volume 2, pages 70–77. Cuba, 2001.

[29] Andrew Silva, Matthew Gombolay, Taylor Killian, Ivan Jimenez, and Sung-Hyun Son. Optimization methods for interpretable differentiable decision trees applied to reinforcement learning. In *International conference on artificial intelligence and statistics*, pages 1855–1865. PMLR, 2020.

[30] Guiliang Liu, Oliver Schulte, Wang Zhu, and Qingcan Li. Toward interpretable deep reinforcement learning with linear model u-trees. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2018, Dublin, Ireland, September 10–14, 2018, Proceedings, Part II 18*, pages 414–429. Springer, 2019.

[31] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically interpretable reinforcement learning. In *International Conference on Machine Learning*, pages 5045–5054. PMLR, 2018.

[32] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable reinforcement learning via policy extraction. *Advances in neural information processing systems*, 31, 2018.

[33] Dweep Trivedi, Jesse Zhang, Shao-Hua Sun, and Joseph J Lim. Learning to synthesize programs as interpretable and generalizable policies. *Advances in neural information processing systems*, 34:25146–25163, 2021.

[34] Wenjie Qiu and He Zhu. Programmatic reinforcement learning without oracles. In *International Conference on Learning Representations*, 2021.

[35] Guan-Ting Liu, En-Pei Hu, Pu-Jen Cheng, Hung-Yi Lee, and Shao-Hua Sun. Hierarchical programmatic reinforcement learning via learning to compose programs. *arXiv preprint arXiv:2301.12950*, 2023.

[36] Daniel Hein, Alexander Hentschel, Thomas Runkler, and Steffen Udluft. Particle swarm optimization for generating interpretable fuzzy reinforcement learning policies. *Engineering Applications of Artificial Intelligence*, 65:87–98, 2017.

[37] Daniel Hein, Steffen Udluft, and Thomas A Runkler. Interpretable policies for reinforcement learning by genetic programming. *Engineering Applications of Artificial Intelligence*, 76: 158–169, 2018.

[38] Alexander Mott, Daniel Zoran, Mike Chrzanowski, Daan Wierstra, and Danilo Jimenez Rezende. Towards interpretable reinforcement learning using attention augmented agents. *Advances in neural information processing systems*, 32, 2019.

[39] Eoin M Kenny, Mycal Tucker, and Julie Shah. Towards interpretable deep reinforcement learning with human-friendly prototypes. In *The Eleventh International Conference on Learning Representations*, 2023.

[40] Raghuram Mandyam Annasamy and Katia Sycara. Towards better interpretability in deep q-networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 4561–4569, 2019.

[41] Zoe Juozapaitis, Anurag Koul, Alan Fern, Martin Erwig, and Finale Doshi-Velez. Explainable reinforcement learning via reward decomposition. In *IJCAI/ECAI Workshop on explainable artificial intelligence*, 2019.

[42] Setareh Cohan, Nam Hee Kim, David Rolnick, and Michiel van de Panne. Understanding the evolution of linear regions in deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:10891–10903, 2022.

[43] Ofir Nachum, Shixiang Shane Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. *Advances in neural information processing systems*, 31, 2018.

[44] Aleksandra Faust, Kenneth Oslund, Oscar Ramirez, Anthony Francis, Lydia Tapia, Marek Fiser, and James Davidson. Prm-rl: Long-range robotic navigation tasks by combining reinforcement learning and sampling-based planning. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 5113–5120. IEEE, 2018.

[45] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112 (1-2):181–211, 1999.

[46] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

[47] David Ha, Andrew M Dai, and Quoc V Le. Hypernetworks. In *International Conference on Learning Representations*, 2017.

[48] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[49] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[50] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

[51] Been Kim, Rajiv Khanna, and Oluwasanmi O Koyejo. Examples are not enough, learn to criticize! criticism for interpretability. *Advances in neural information processing systems*, 29, 2016.

[52] Sandra Wachter, Brent Mittelstadt, and Chris Russell. Counterfactual explanations without opening the black box: Automated decisions and the gdpr. *Harv. JL & Tech.*, 31:841, 2017.

[53] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. Axiomatic attribution for deep networks. In *International conference on machine learning*, pages 3319–3328. PMLR, 2017.

[54] Ramaravind K Mothilal, Amit Sharma, and Chenhao Tan. Explaining machine learning classifiers through diverse counterfactual explanations. In *Proceedings of the 2020 conference on fairness, accountability, and transparency*, pages 607–617, 2020.

[55] David Alvarez-Melis and Tommi S Jaakkola. On the robustness of interpretability methods. *arXiv preprint arXiv:1806.08049*, 2018.

[56] Dylan Slack, Sophie Hilgard, Emily Jia, Sameer Singh, and Himabindu Lakkaraju. Fooling lime and shap: Adversarial attacks on post hoc explanation methods. In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, pages 180–186, 2020.

[57] Dylan Slack, Anna Hilgard, Himabindu Lakkaraju, and Sameer Singh. Counterfactual explanations can be manipulated. *Advances in neural information processing systems*, 34: 62–75, 2021.

[58] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[59] Scott Fujimoto, David Meger, and Doina Precup. Off-policy deep reinforcement learning without exploration. In *International conference on machine learning*, pages 2052–2062. PMLR, 2019.

[60] Georg Ostrovski, Pablo Samuel Castro, and Will Dabney. The difficulty of passive learning in deep reinforcement learning. *Advances in Neural Information Processing Systems*, 34: 23283–23295, 2021.

[61] Christoph Molnar, Giuseppe Casalicchio, and Bernd Bischl. Interpretable machine learning–a brief history, state-of-the-art and challenges. In *ECML PKDD 2020 Workshops: Workshops of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD 2020): SoGood 2020, PDFL 2020, MLCS 2020, NFMCP 2020, DINA 2020, EDML 2020, XKDD 2020 and INRA 2020, Ghent, Belgium, September 14–18, 2020, Proceedings*, pages 417–431. Springer, 2021.

[62] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. *Advances in neural information processing systems*, 27, 2014.

[63] Denis Yarats and Ilya Kostrikov. Soft actor-critic (sac) implementation in pytorch. `https://github.com/denisyarats/pytorch_sac`, 2020.

[64] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

[65] Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C Courville, and Marc Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *Advances in neural information processing systems*, 34:29304–29320, 2021.

[66] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *Advances in neural information processing systems*, 30, 2017.

A Background

A.1 Deep reinforcement learning and interpretable policies

In reinforcement learning, an agent learns to interact with its environment to maximize the accumulation of a learning signal, the reward [48]. The environment is modeled as a Markov Decision Process (MDP) defined as a tuple $(\mathcal{X}, \mathcal{A}, R, P, \gamma, \rho_0)$. At each timestep t, the agent in state $x_t \in \mathcal{X}$ takes the action $a_t \sim \pi(x_t) \in \mathcal{A}$. The agent then transitions to $x_{t+1} \sim P(\cdot|x_t, a_t)$ and receives reward $r_{t+1} = R(x_t, a_t, x_{t+1})$. We denote with ρ_0 the starting state distribution and represent a trajectory τ as $(x_0, a_0, x_1, \ldots, x_T)$, where T is the last timestep of the trajectory if the environment is episodic, and $T = \infty$ else. The discounted sum of rewards is $G(\tau) = \sum_{t=0}^{T-1} \gamma^t r_{t+1}$, and we denote its expectation $J^\pi = \mathbb{E}_\tau [G(\tau)]$. Fundamental in reinforcement learning is the function Q^π, for which for a given state and action, $Q^\pi(x, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{T-1} \gamma^t R(x_t, a_t, x_{t+1}) | x_0 = x, a_0 = a\right]$. Actor-critic algorithms solve an MDP by modelling explicitly both π and Q^π. They aim at finding a policy π^* such that $\pi^* = \text{argmax}_\pi J^\pi$.

Both Q^π and π can be parametrized by deep neural networks (NNs) to handle high-dimensional input spaces and increase the performance of the agent [49]. However, this modelisation limits our understanding of the actions of an agent, due to the opacity of NNs.

In this work, we tackle the task of making the *policy* interpretable at each timestep. Since we focus on functional interpretability, we constrain the class of functions that the policy belongs to. That is, we design a policy architecture that gives exact insights about the decision applied to state x_t. The last layer of deep policies is often a non-linearity μ, such as the softmax function, to cast the prediction of the NN to the right action space. In a similar fashion to the interpretation of logistic regression, we focus on interpreting the *pre* non-linearity part of the policy, which we denote by $\tilde{\pi}$. Finally, we remark that an interpretable model is of limited utility if the input features themselves are not interpretable. Therefore, we consider robotics environments with varying levels of difficulty and where the input to the policy is the proprioceptive state representation, each feature representing a physical variable describing the agent [22].

A.2 Post-hoc, distillation and functional interpretability

We begin by setting the context of interpretability in machine learning. We wish to explain a *deployed model* and its predictions to users of varying expertise. There exists three main approaches to do so:

- A *post-hoc* method is an auxiliary model that extracts explanations from a complex (potentially a black box), well-performing deployed model [50, 51, 52, 53, 54]. We can expect good performance from the deployed model. However, the explanations are provided by the auxiliary model, which raises questions about their reliability, their accuracy or their robustness [55, 13, 56, 57].
- *Distillation* methods first train a complex "teacher" model, and then fit and deploy a simpler "student" model on the predictions of the teacher [58, 30]. The explanations come from the interpretable nature of the student (*e.g.* a shallow decision tree or a linear model). Yet, defining the data distribution to fit the student on can be challenging, especially in the DRL setting [59, 60].
- *Functional* methods [10] directly train an interpretable model, which is deployed and explain its decisions [13, 39]. The end-to-end approach to interpretability often imposes limits on expressivity that can imply a performance trade-off.

In this work, we focus on functional interpretability. To achieve it, one can reduce the size of the class that the model belongs to [13]. Classes of functions that can be expressed through a small number of parameters, such as linear functions or shallow decision trees, are often considered to be interpretable due to the reduced amount of information required to understand the function they realize [61]. Yet, designing such a model is challenging, since inherently interpretable models are at odds with the complexity and expressivity required to solve hard tasks. For instance, a linear policy might not be expressive enough to solve an intricate maze. Moreover, the inclusion

of discrete components, while often involved in interpretable models [10], can make training unstable.

B Reducing the size of the network to limit its complexity

B.1 Theoretical analysis

Reducing the width or the height of an MLP is an alternative and more direct way than our proposed approach to reduce the number of unique coefficients. In fact, reducing the size of the MLP also reduces the size of the partition, *i.e.* the number of linear regions, on the contrary to our proposed solution. The main issue with this approach is that controlling the number of unique coefficients becomes impractical, since the maximum number of linear regions (which is the quantity we have control over) grows at least exponentially *w.r.t.* the depth and polynomially *w.r.t.* the width of the network [62, Corollary 5]. For instance, a relatively small MLP with a one-dimensional input and 2 layers of 64 neurons each can express a maximum of over 4000 linear regions [62, Theorem 4]. Finally, accessing the linear coefficient for a given feature x_i requires a backward pass when using an MLP, while it can be directly outputted as part of our solution, since we compute this quantity explicitly in the forward pass.

B.2 Empirical analysis

We complement the theoretical analysis of the upper bound by an empirical analysis of the actual number of unique linear regions used by small variants of the Soft Actor Critic algorithm (SAC). To do so, we train variants of SAC with an MLP actor architecture of depth 1, 2 and 3 and width of 2, 4, 8, 16, 32, 64, 128 and 256 units. We then roll out each variant in the Cheetah run environment (prone to sub-policy re-use due to the periodicity of the task) 10 times. We repeat the procedure for 10 seeds, which gives us, for each variant, 100 measures of the episode return (performance metric), and 100 number of unique sub-policies used (USPU) during the trajectory (complexity metric). We then produce a Pareto plot illustrating the trade-off between complexity and performance for small SAC architectures, in Fig. 4. We also compare the small variants of SAC with 2 variants of the HyperCombinator, respectively with 8 (HC8) and 64 (HC64) possible sub-policies.

We observe that overall, the performance of SAC is correlated with the number of unique sub-policies used during a trajectory. As a consequence, models with a small USPU metric tend to perform significantly worse than the more complex variants. For instance, to reach an average return of 700, variants of SAC need at least 200 unique sub-policies. This means that in average, each sub-policy is used only 5 times during a trajectory. In addition, if we limit the complexity to 100 USPU, the average return of SAC variants barely surpasses 600, well below the average return of 800 reached by the more complex SAC variants. Conversely, our proposed approach offers a better trade-off between complexity and performance SAC. Indeed, HC8 and HC64 are located further than the SAC Pareto front, as they manage to obtain significantly higher return for comparable complexity, or equivalently a comparable return with a much smaller number of unique sub-policies used. While in this particular example, HC8 outperforms even the SAC variants with the biggest architectures (see also Fig. 6), we insist specifically on the excellent performance of HC8 and HC64 *w.r.t.* SAC variants that use a comparable number of USPU.

In conclusion, simply limiting the architecture of SAC is not an adapted solution to our problem, since one cannot exactly control the number of unique sub-policies that will be used, and since the performance decreases severely for low number of unique sub-policies used ($<$100), even more so with very low number of unique sub-policies used ($<$10).

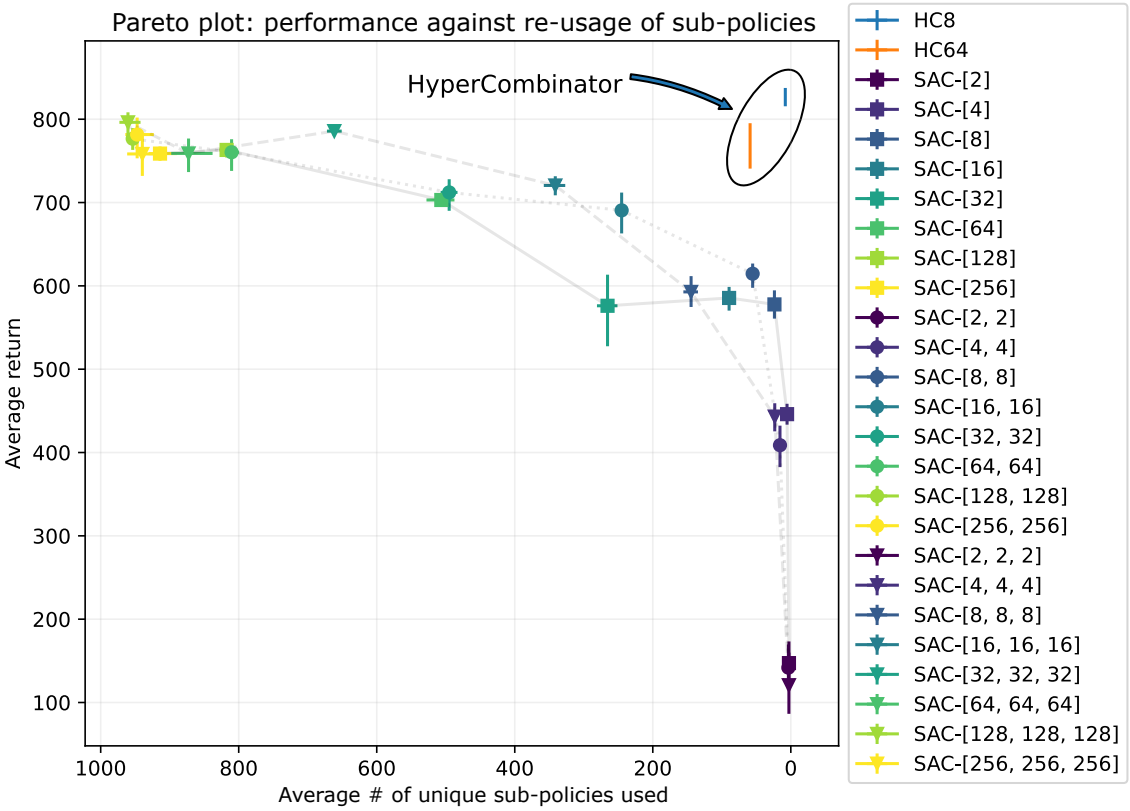

Figure 4: Pareto plot comparing the HyperCombinator actor and SAC with different small architectures on performance (y-axis, higher is better) and number of sub-policies used during a trajectory (x-axis, lower is better). Note the inversion of the x-axis, such that the best models are located in the top right part of the plot (with high return and low number of unique sub-policies used). We display error bars representing the 95% bias corrected and accelerated bootstrap confidence intervals (9999 resamples).

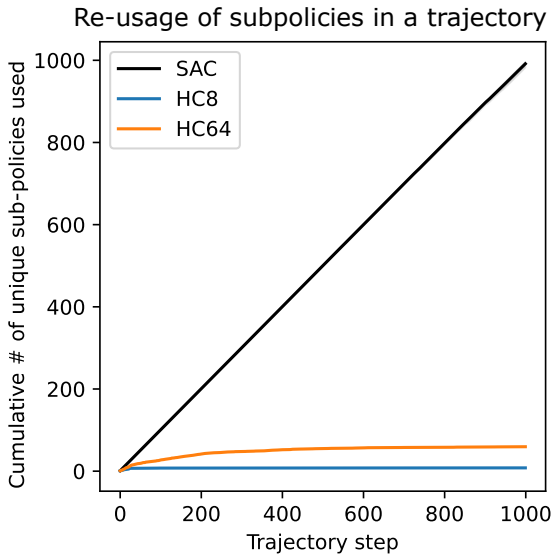

Figure 5: During a trajectory, a SAC actor visits a new linear region (and uses a new locally linear sub-policy) at almost every timestep. On the contrary, our HC agent can only express one of the 8 or 64 sub-policies at his disposal, which forces it to learn sub-policies that are valid for a wide range of inputs.

C Sub-policies reuse

Since SAC uses an MLP as the policy, the policy is also a piecewise-linear function. We study in this section how often does the policy re-use the same linear region in a trajectory.

We perform the experiment using the trained models of Sec. 4.1, on the Cheetah run environment, which is particularly favorable to the reuse of sub-policies. We rollout the trained policies of SAC, HC8 and HC64 for an entire trajectory and record, for each timestep, the number of unique sub-policies used to interact with the environment until the current timestep. This yields a non-decreasing sequence of integers per algorithm. The lower the curve, and the less sub-policies are being used by the policy. We repeat the same procedure 10 times, and compute the mean curve as well as a 95% confidence interval using the bias corrected and accelerated bootstrap (9999 resamples).

We display the results in Fig. 5. The HyperCombinator cannot express more than d_G sub-policies, which caps the corresponding plots for both HC8 and HC64. We observe that the curve corresponding to SAC linearly increases almost without failure. This indicates that SAC barely reuses past sub-policies, *i.e.* that a new locally linear function is applied at practically each timestep. Moreover, this sub-policy is shared by all seeds, given the thinness of the confidence interval.

This does not mean that SAC strictly overfitted the trajectory or completely memorized the sequence of policies to apply during the trajectory: indeed, there is a great amount of parameter sharing between the linear coefficients in θ. Thus, there is generalization happening between sub-policies, even if a different linear region is used at each timestep. Yet, the precise generalization mechanism is obscure, and it is not clear to what extent updating a sub-policy will affect the other sub-policies. Perhaps more importantly to our use case, one still needs to enumerate all of the sub-policies when describing the possible interactions of the policy with the environment, given that even though they might share parameters, each sub-policy uses a different set of linear coefficients. As a consequence, this makes the SAC policy highly uninterpretable, on the contrary to the HyperCombinator where the size of the exhaustive list of sub-policies is capped by d_G.

D Control experiments

D.1 Precisions regarding the architecture of the HyperCombinator

We can use the HyperCombinator as the actor of any actor-critic algorithms, provided that no competing assumption is made on the architecture of the policy. This means that the critic or the potential other parts of the RL algorithm can be modeled by complex neural networks without affecting the interpretability of the policy. We note that in practice, all the linear sub-policies share the same bias vector. This limits the flexibility of the sub-policies, but also increases their interpretability since they all reduce to the same default sub-policy at $x = 0$. We adapt μ to the action space of interest. In addition, we note that the Gumbel-Softmax layer can destabilize training. For instance, its predictions may collapse to a single mode, making $\hat{\pi}$ equivalent to a linear policy. We alleviate this issue through different forms of regularization. First, we increase the temperature parameter and add weight decay to the last layer weights of the Gumbel network MLP. Both limit the magnitude of the input fed to the Gumbel-Softmax layer. Secondly, we maximize the entropy H of the average sub-policy assignation over a batch of size B, that is $H(\frac{1}{B}\sum_{i=1}^{B}\hat{a}(x_i))$. This encourages the Gumbel network to use the diversity of sub-policies at its disposal.

D.2 Implementation details

We base ourselves on an open-source PyTorch implementation of SAC [63] with its default hyperparameters, listed in Table 2. We detail below the modifications needed to implement the HyperCombinator and replicate the experiments, and keep the rest of the existing code for training and evaluating SAC and HC.

SAC algorithm We modify the actor of the Soft Actor Critic algorithm (SAC) [23], a strong continuous control agent learning from proprioceptive observations. The original actor is modeled as an MLP that predicts a vector of size $2|\mathcal{A}|$. The output vector is then split into two parts, the mean of the action predictive distribution, $\tilde{\pi}$ and its log standard deviation, $\log\sigma$. We note that this implies that the mean $\tilde{\pi}$ and the log std $\log\sigma$ share some of their parameters through the shared structure of the actor. The log standard deviation is transformed to belong to the interval $[\log\sigma_{\text{min}}, \log\sigma_{\text{max}}]$ using the following formula:

$$t(u) = \log\sigma_{\text{min}} + (\log\sigma_{\text{max}} - \log\sigma_{\text{min}}) \cdot \frac{u+1}{2} \tag{2}$$

Finally, the action predictive distribution for state x is defined as a *squashed normal*:

$$\mu\left(\mathcal{N}(\tilde{\pi}(x), \exp t(\log\sigma(x)))\right) \tag{3}$$

where μ is the non-linear transformation ensuring that the action belongs to the environment action space. In practice for the DeepMind Control Suite, we use $\tanh$ as μ to force the predicted actions to belong to $[-1, 1]$.

The critic is learnt via double Q learning [64]. Each critic network is a 2-layer MLP. The strength of SAC entropy regularization (represented by the hyperparameter α) is automatically learnt during training. We use Adam to optimize all parameters. All weight matrices are initialized using orthogonal initialization. All bias vectors are initialized to 0.

HyperCombinator modifications With the HyperCombinator, we model the mean $\tilde{\pi}$ and the log std $\log\sigma$ with independent networks. HC first models $\log\sigma$ as an MLP with the same architecture as in SAC, but that only outputs a vector of size $|\mathcal{A}|$. Then, HC models $\tilde{\pi}$ as a Gumbel network, that is, the composition of an MLP, a Gumbel-Softmax layer (with the straight-through estimator) and a linear layer. The rest of the computation of the action predictive distribution is left unchanged. In particular, for x, the mean of the squashed normal is in both cases $\mu(\tilde{\pi}(x))$ and does not depend on $\log\sigma(x)$.

We detail the modified SAC algorithm using Alg. 1. We indicate in **blue** the modifications due to using a HyperCombinator actor. We remark that the only major modification is in the architecture of the actor. Therefore, the majority of the structure of the SAC algorithm is left unchanged. We notice the use of stop_grad to detach a tensor from the computation graph.

Training details Classically, Gumbel noise is used in the Gumbel-Softmax layer to stochastically select the sub-policy. After sufficient training, this (Gumbel) trick ensures that we ultimately sample from the likeliest sub-policy. We only activate Gumbel noise during the actor update, when computing the action predicted by the actor. When the agent interacts with the environment, we do not use Gumbel noise, both for training and for evaluation. Therefore, the Gumbel network selects the most likely *sub-policy* for all interactions with the environment. This ensures that the *mean* of the action predictive distribution is deterministic, like SAC. It also improves the consistency of the agent over an episode, since for state x, a fixed policy leads to the same sub-policy. This is not to be confused with the action predicted by the HyperCombinator, which is sampled from the squashed normal defined in Eq. 3 during training.

Model choice We experiment with different combination of hyperparameters values (Gumbel network architecture from [64,64] to [1024,1024,1024], and strength of regularization of the average assignation entropy λ_{assig} from 0 to 1) on Cheetah run, from where we selected the best set of hyperparameters according to the return curve. We then evaluate the HyperCombinator using the same fixed set of hyperparameters for all the environments.

Evaluation details We evaluate the agent every 10000 timesteps by rolling it out for 10 episodes and taking the average return. During evaluation, all actors act deterministically using only the mean of the predicted action distribution $\mu(\tilde{\pi}(x))$, without exploration noise (as opposed to sampling from the predictive action distribution during training). Hence, it is sufficient for $\mu(\tilde{\pi})$ to be piecewise-linear to get the desired form for the actor at evaluation time. We report all results and curves in Sec. 4.1 using 10 seeds for each agent. We draw the mean performance as a colored line, as well as a 95% bias-corrected and accelerated bootstrap confidence interval in a lighter shade (9999 resamples).

Compute We ran all the experiments on an internal cluster. All the GPUs were NVIDIA Tesla V100, with 16GB memory available. The CPUs were Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz Each seed was allocated 1 GPU, 10 CPUs, and 64GB of RAM. We detail the compute budget to reproduce the experiments in Table. 1.

Experiment	**# models**	**# envs**	**# seeds**	**Avg. duration**	**Compute**
Full results (Fig. 6)	3	23	10	6 hours	173 GPU days
Longer horizon (Fig. 10)	3	4	10	34 hours	170 GPU days
Perf.-interpret. gap (Fig. 11)	8	1	10	17 hours	57 GPU days
Small SAC (Fig. 4)	24 (*)	1	10	6 hours	60 GPU days

Table 1: Compute budget for the control experiments. (*) the HyperCombinator plots were already computed in the full results.

[2]Shared between the mean net and the log std net

Hyperparameter name	Value
Common (SAC defaults) [63]	
Action repeat	1
Discount factor	0.99
Learnable α	True
Initial α	0.1
α learning rate λ_α	1e-4
α Adam momentums	[0.9, 0.999]
Actor learning rate λ_π	1e-4
Actor Adam momentums	[0.9, 0.999]
Actor update frequency	1
Critic architecture	[1024, 1024]
Critic learning rate λ_Q	1e-4
Critic Adam momentums	[0.9, 0.999]
Critic exponential moving average ratio	0.005
Critic target update frequency	2
Batch size	1024
$\log \sigma_{\text{min}}$	-5
$\log \sigma_{\text{max}}$	2
SAC actor-specific	
Actor architecture[2]	[1024,1024]
HyperCombinator-specific	
Gumbel net architecture	[1024, 1024, 1024]
Sub-policy assignation entropy coefficient λ_{assig}	0.001
Gumbel temperature	1

Table 2: Full list of hyperparameters in the control experiments.

Algorithm 1 `SAC (with HyperCombinator actor)`

Require: Replay Buffer $\mathcal{D}$
Require: Actor parameters φ
Require: Double critic parameters η_1, η_2
Require: Double critic target parameters $\overline{\eta}_1$, $\overline{\eta}_2$
Require: Hyperparameters from Table 2
Require: N ▷ Maximum number of timesteps
Require: s_0 ▷ Initial state
  while $t < N$ **do**
    $a_t \sim \mu(\tilde{\pi}(s_t))$
    $s_{t+1} \sim P(\cdot|s_t, a_t)$ ▷ Sample the next state from the environment
    $r_{t+1} = R(s_t, a_t, s_{t+1})$
    $\mathcal{D} = \mathcal{D} \cup (s_t, a_t, r_{t+1}, s_{t+1}, d_{t+1})$ ▷ Update replay buffer; d_{t+1} indicates a terminal transition

    $(s, a, r, s', d) \sim \mathcal{D}$ ▷ Sample from replay buffer
    Launch routine `UpdateCritic` ▷ See paper [23] and code[3]
    if t % actor update frequency $== 0$ **then**
      Launch routine `UpdateActorAndAlpha` (see Alg. 2)
    end if
    if t % critic target update frequency $== 0$ **then**
      $\overline{\eta}_1 = (1 - \text{critic ema ratio}) * \overline{\eta}_1 + \text{critic ema ratio} * \eta_1$
      $\overline{\eta}_2 = (1 - \text{critic ema ratio}) * \overline{\eta}_2 + \text{critic ema ratio} * \eta_2$
    end if
  end while

Algorithm 2 `UpdateActorAndAlpha`

Require: s ▷ Batch of states sampled from the replay buffer

$a \sim \mu(\mathcal{N}(\tilde{\pi}(s), \exp(t(\log \sigma(s))))$ ▷ $\tilde{\pi}$ is a Gumbel network and $\log \sigma$ an MLP instead of being jointly parametrized as an MLP in the base case.

$\mathcal{L}_{\text{assig}} = H(\text{mean}(\hat{a}(s)))$ ▷ Compute entropy of average sub-policy assignation

$Q = \min(Q_{\eta_1}(s, a), Q_{\eta_2}(s, a))$

$\mathcal{L}_\pi = \text{mean}\,(\text{stop_grad}(\alpha) \log \pi(a|s) - Q) - \lambda_{\text{assig}} \mathcal{L}_{\text{assig}}$ ▷ Do not backprop. gradients through α

$\varphi = \varphi - \lambda_\pi \nabla_\varphi \mathcal{L}_\pi$

if learn α **then**

  $\mathcal{L}_\alpha = \text{mean}(\alpha\ \text{stop_grad}(\log \pi(a|s) + |\mathcal{A}|))$

  $\alpha = \alpha - \lambda_\alpha \nabla_\alpha \mathcal{L}_\alpha$

end if

D.3 Results for all DeepMind Control environments

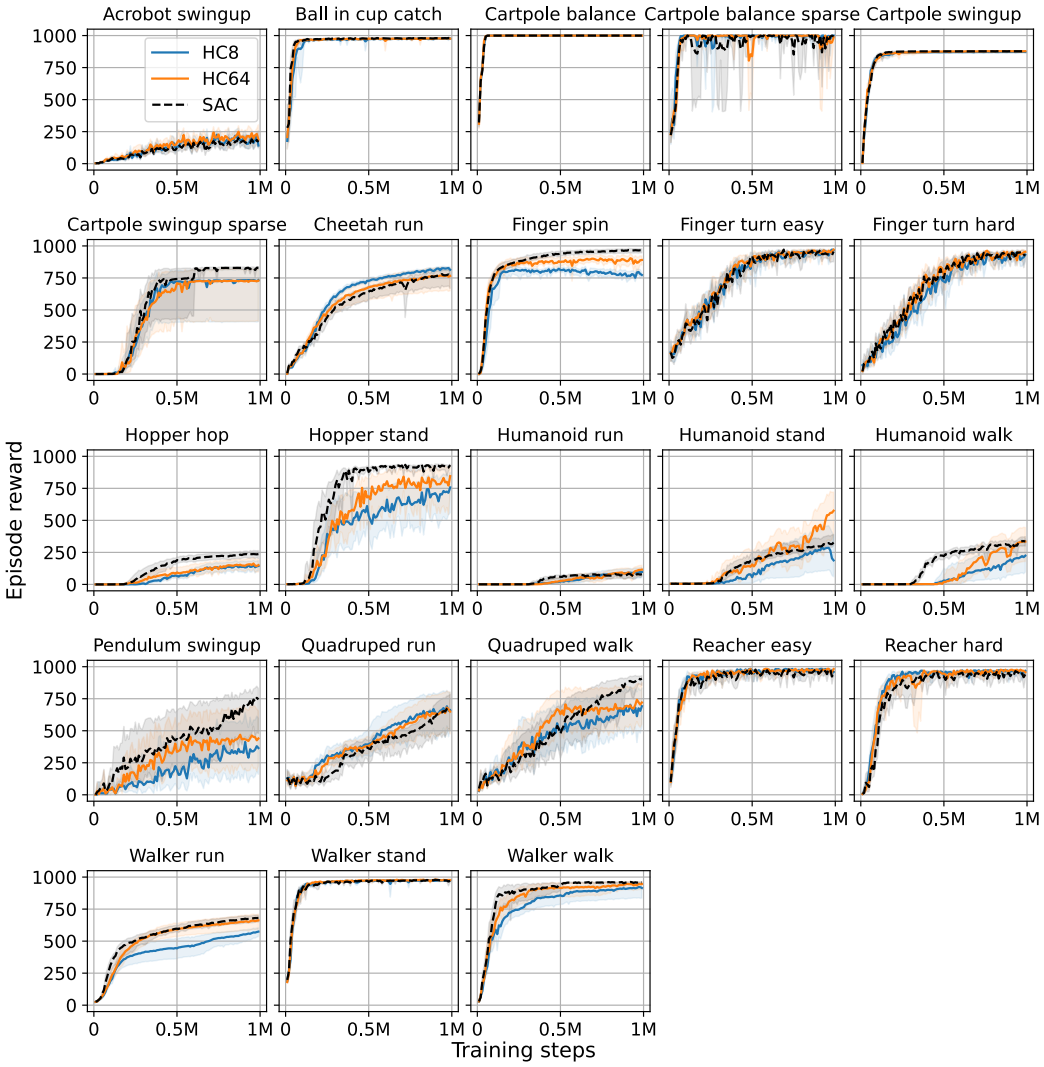

Figure 6: Complete results for the DM control environments.

D.4 Detailed quantitative results

To compute the results, we followed the methodology proposed in the `rliable` library [65]. We used the final scores after 990000 frames for each run.

We first investigate the scores produced by `rliable` in Fig. 7. This lets us quantify the trade-off in performance that follows the reduction in expressivity of HC8 and HC64 according to several metrics. In particular, the overlapping confidence intervals between SAC and HC64 indicate that overall, a very small amount of performance is foregone when one is willing to guarantee that the agent will interact with the environment in at most 64 different ways.

Fig. 8 illustrates the close performance between SAC and HC64, especially in the environments where the normalized score is at least 80% of the maximum (*i.e.*, in the games where SAC excels to begin with). HC8 follows a similar curve, albeit sensibly lower.

Finally, Fig. 9 shows the fraction of environments where the normalized score IQM of HC8 (respectively HC64) is at least a percentage of the baseline SAC score. This visualization is

helpful to distinguish the relative performance of the HC actors, compared to SAC, on the different environments. We remark that both for HC8 and HC64, there is a sudden drop around the 0% relative performance mark, which indicates that HC8 and HC64 have close IQM scores to SAC in several of the environments. Moreover, HC8 and HC64 perform worse than SAC on most environments by the end of training, as expected.

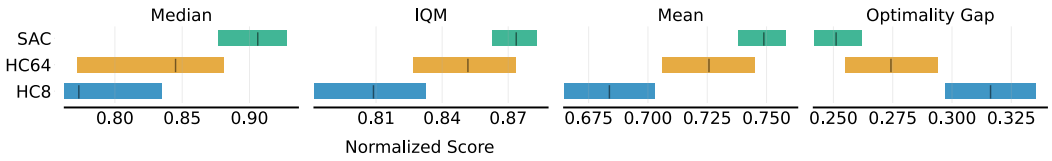

Figure 7: Performance metrics computed using the `rliable` library.

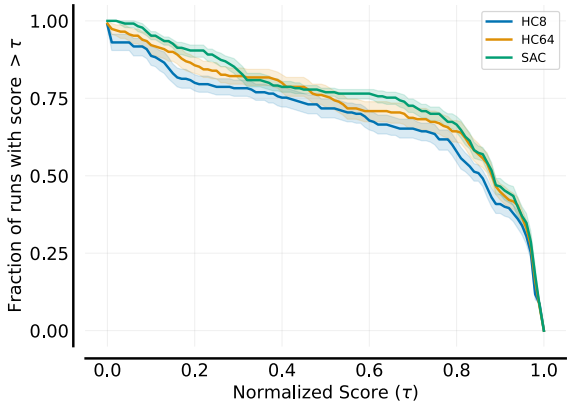

Figure 8: Performance profiles.

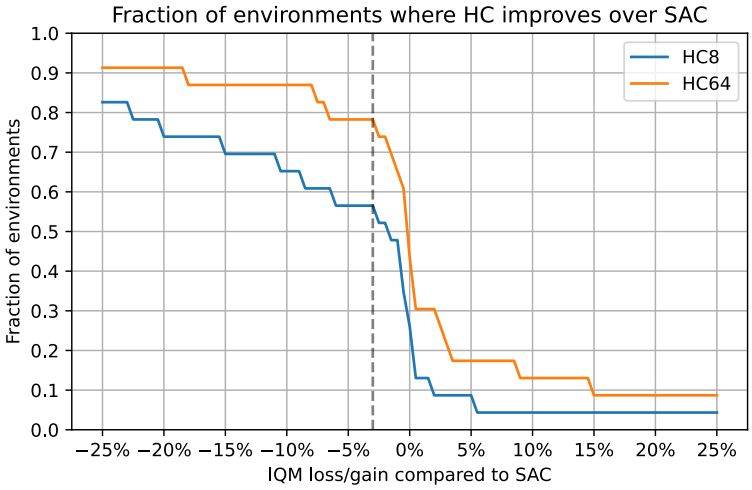

Figure 9: Relative performance loss/gain of the HC actors compared to SAC.

D.5 Longer horizon

In this section, we train the different actors for 5M steps instead of 1M as in the rest of Sec. 4.1. We focus our study on 4 environments: the first is Cheetah run, to observe if the benefits brought to early training by the usage of the HyperCombinator are also transferred to later during the training. The other 3 other environments, Humanoid stand, walk and run, are chosen for their increased difficulty. We illustrate the results in Fig. 10. We remark that HC actors improve over SAC for the first 1M timesteps in the Cheetah run environment. After approximately 1M timesteps, the performance of the SAC actor exceeds the HyperCombinator's. For Humanoid run and stand, both Hypercombinator improve over SAC approximately between 1M and 3M timesteps, with a significant improvement noted for Humanoid stand. For Humanoid walk, only HC64 improves over SAC, here again approximately between 1M and 3M timesteps. The competitive performance of the HyperCombinator actors, added to their improved performance in these hard environments early during training, hints at the benefits of sub-policies re-use.

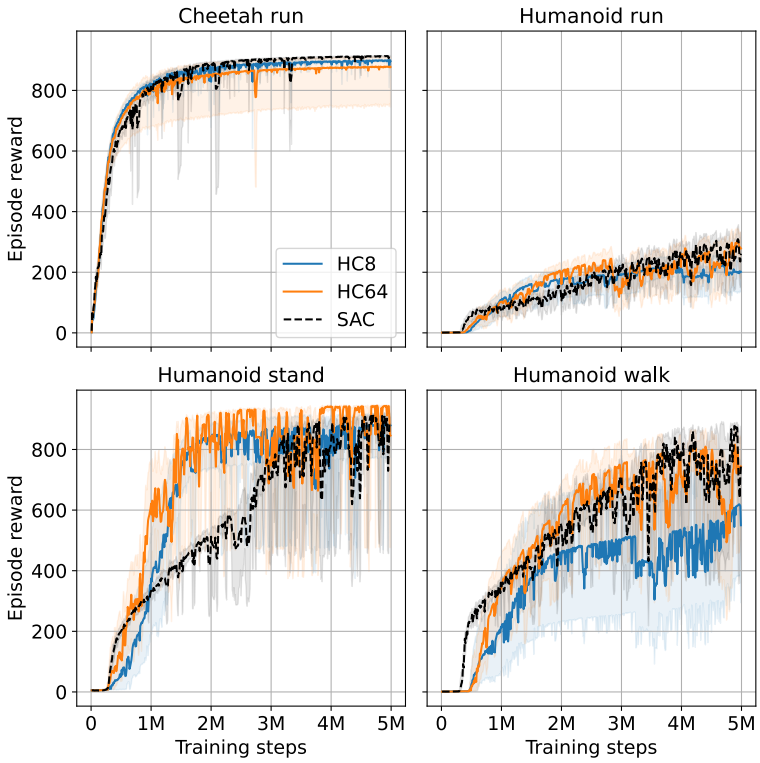

Figure 10: During early training, HC actors can demonstrate a better sample efficiency thanks to the re-use of the same sub-policy for several inputs. Over a longer timeframe, the higher expressivity of SAC leads to matching or better performance.

D.6 Performance-interpretability gap

In this section, we evaluate how the HyperCombinator agent's performance evolves when increasing the number of sub-policies. We run a HyperCombinator actor with 4, 8, 16, 32, 64, 128 and 256 sub-policies on Walker walk for 2.5M timesteps, and observe the results in Fig. 11.

We remark that HC4 reaches a relatively high return, in average higher than 900. This shows that even with a small number of sub-policies, the HyperCombinator can learn a well performing policy in certain environments. As we trade-off interpretability for performance by increasing the number of sub-policies, we see the performance curve increase as well and approach the performance curve of SAC. This trend culminates with 128 sub-policies. We notice that the return curve of HC256 decreases compared to HC128.

A HC agent with a higher number of available sub-policies can produce a more finely grained policy, since each sub-policy can specialize to a smaller part of the state space. Conversely, learning how to chain a high number of sub-policies might not be straightforward. This last point is one possible explanation to the lower performance of HC256.

Figure 11: Increasing the capacity tends to increase the performance, as the policy is now able to exhibit a wider variety of sub-policies. A sufficiently high capacity can help reach a comparable performance to the base algorithm, here SAC. We notice that too many sub-policies can however prove harmful: the performance of HC256 sub-policies decreases compared to HC128.

D.7 Robustness to perturbations

In this section, we evaluate the robustness of the HyperCombinator to perturbations. We start with a regularly trained agent on the Cheetah run task. At evaluation time, after 100 timesteps (out of a 1000 timesteps trajectory), we ignore the action predicted by the agent and instead interact with the environment through an action sampled uniformly at random from the action space. We do so for 100 consecutive timesteps. At timestep 200, we resume normal evaluation and follow the action predicted by the agent. This perturbation throws the agent off its trajectory.

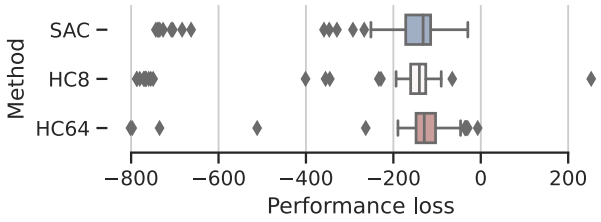

Figure 12: When perturbed with 100 consecutive random actions, HC actors perform overall no worse than SAC. The box plot summarizes the results of the experiment repeated 10 times for each seed.

Figure 13: After being perturbed for 100 timesteps, the HC actor manages to resume the task. sub-policy choices between t=100 and 200 (in turquoise) are ignored since a random action is used instead.

We repeat this evaluation 10 times per seed, for each of the 10 seeds of an algorithm, for a total of 100 data points.

We now compare how this perturbation affects HC8 and HC64, compared to the baseline, SAC. In Fig. 12, we compute for each model the distribution of performance losses, defined as the performance of the perturbed evaluation subtracted to the performance of the unperturbed evaluation. We then illustrate the distribution of these performance losses. We see that all models react similarly to the perturbations. This illustrates the fact that the HyperCombinator is no less robust to perturbations than the base neural policy, despite its non-continuity. A similar conclusion was already found when comparing linear models to neural baselines [26]. Accordingly, diversifying the initialization distribution could improve the robustness of the model.

We illustrate in Fig. 13 how the agent, after being perturbed for 100 timesteps, manages to resume the cyclical choice of sub-policies that lets it solve the task.

D.8 Analysis of the linear coefficients interpretability in Cartpole

We now analyze the low-level interpretability benefits of our method when applied to the Cartpole environment.

In this environment, the agent controls the translation of a cart on an axis in order to swing up and then stabilize a pole. We write the sub-policy i as: $b+w^i_x x+w^i_{\sin\theta}\sin\theta+w^i_{\cos\theta}\cos\theta+w^i_{\dot{x}}\dot{x}+w^i_{\dot{\theta}}\dot{\theta}$, for an input $(x, \sin(\theta), \cos(\theta), \dot{x}, \dot{\theta})$ and sub-policy coefficients $w^i_{\cdot}$. Note that there is a single action: for a given positive feature, the contribution tends to move the cart towards the right if the corresponding coefficient is positive and left if negative.

We visualize the results in Fig. 14. The left plot is an alternative visualization of the sub-policy coefficients w_i (to be compared with the heatmap in Fig. 2, right). This is an easy way to note similarities between sub-policies and to understand how they are combined. For instance, HC mainly uses sub-policy B0 (in orange) to stabilize the pole that has been swung up. This is translated by a negative coefficient in front of the cart velocity, such that if the agent moves towards the right (causing the pole to tilting to the left), the sub-policy will push the agent towards the left to balance the pole.

We also see in Fig. 14 (right) that the agent sometimes oscillates between using B0 (orange) and sub-policy 1 (grey). We can observe in the left plot that they both give a large coefficient to $\cos\theta$, which is around 0 when the pole is stabilized. The positive sign of the coefficient indicates

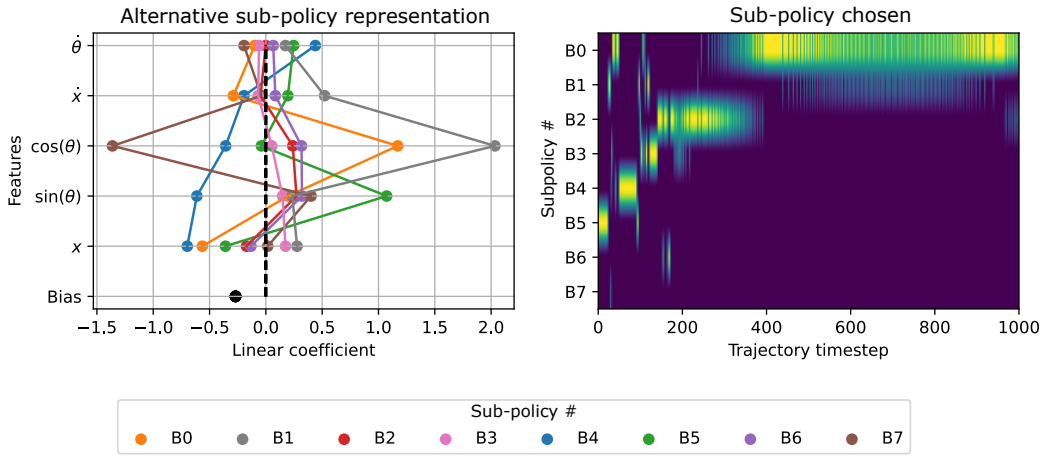

Figure 14: Evaluation of HC8 in cartpole swingup. (Left) alternative visualization of the sub-policies. (Right) sub-policy choice sequence.

that if the pole is tilting towards the right, the agent will tend to move the cart towards the right to balance the pole. Finally, Fig. 14 (left) shows that B2 (red) and B6 (purple) are extremely similar. Simultaneously, we see in Fig. 14 (right) that the only setting in which B6 is used is during an extended period of usage of B2.

We think that this figure is an example of how using a HC actor (rather than a neural one) can improve the interpretability and our understanding of the task.

E Navigation experiments

E.1 Implementation details

We base our experiments on the open-source code provided by RIS [25][4], and use most of its default hyperparameters. We detail in Table 4 the common hyperparameters and the hyperparameters we change for the HyperCombinator.

RIS algorithm RIS belongs to the family of goal-conditioned RL algorithms, that learn a policy $\pi(s, g)$. RIS aims at guiding the policy, during training, to produce the same action to reach a goal as the action to reach an imaginary subgoal located halfway through the trajectory. In this case, the notion of distance implied in "halfway" means that as many steps are needed to go from the current state to the imaginary subgoal than from the imaginary subgoal to the goal (as opposed to an Euclidean notion of distance that would not be suitable in mazes).

To do this guiding, a *prior policy* π_{prior} is defined as an empirical moving average of the online, interacting policy. A neural network (denoted a *high-level policy* in the original paper) learns online to generate likely subgoals s_g given a state s and a goal g. Then, given state s, the policy $\pi(s, g)$ is constrained to stay close to the prior policy to reach subgoal s_g, *i.e.* $\pi_{\text{prior}}(s, s_g)$.

Similarly to SAC, the RIS actor is composed of an MLP on top of which two heads predicting the mean of the action predictive distribution, $\tilde{\pi}(x)$, and its log standard deviation $\log \sigma(x)$. t' clamps $\log \sigma(x)$ between $\log \sigma_{\text{min}}$ and $\log \sigma_{\text{max}}$. The action is then sampled from a squashed normal, similarly to SAC. Two critics, each modeled by a MLP, are learnt with double Q learning. The hyperparameter α ([25, Eq. 9]) is fixed during training. The algorithm uses Hindsight Experience Replay (HER) [66] to facilitate training, and we refer to the RIS paper for further details on the specific implementation [25]. All weight matrices are initialized using orthogonal initialization. All bias vectors are initialized to 0.

HyperCombinator modifications We replace $\tilde{\pi}$ by a Gumbel network. Similarly to the control experiments, $\log \sigma$ is still defined by an MLP, but does not share parameters with $\tilde{\pi}$. As a consequence of the modelisation of $\tilde{\pi}$, the prior policy π_{prior} also takes the form of a HC policy. We do not modify the rest of the algorithm, including the high-level policy that learns to predict the intermediate goals used for training. Due to memory constraints, we use a batch size of 1024 to train the HyperCombinator, instead of the base 2048. We train RIS with batch sizes of both 1024 and 2048 and ensure that the performance of RIS with 2048 (the base value) improves over the performance with a batch size of 1024. We detail in Alg. 3 the RIS algorithm and add in **blue** the modifications that we apply.

Training details The agents interact with the environment for a maximum of 600 timesteps, after which the episode is interrupted. We do not use Gumbel noise for any interaction with the environment, which guarantees that the most likely *sub-policy* is consistently selected. Therefore, sub-policies are stochastically chosen only during the update of the actor, in the `UpdateActorAndAlpha` routine. During training, the actions are sampled from the action predictive distribution $\mu(\mathcal{N}(\tilde{\pi}(s, g), \exp(t'(\log \sigma(s, g))))$ to encourage exploration. We use three different types of regularization to prevent the Gumbel network from collapsing into the prediction of a single sub-policy:

- We increase the temperature of the Gumbel-Softmax, controlled by the "Gumbel temperature" parameter in Table 4.
- We regularize the entropy of the sub-policy assignations averaged over a batch, controlled by the λ_{assig} hyperparameter.
- We penalize the magnitude of the last layer weights of the MLP preceding the Gumbel-Softmax, through the $\lambda_{\text{weight decay}}$ hyperparameter. This forces the input to the Gumbel-Softmax layer to have a smaller magnitude, and therefore encourages the selection of a more diverse set of sub-policies.

[4] https://github.com/elliotchanesane31/RIS

Model choice We experiment with different combinations of the architecture of the Gumbel network (from [32,32] to [1024,1024,1024]), the sub-policy assignation entropy coefficient λ_{assig} (between 0 and 0.1), the Gumbel temperature (from .5 to 10), the weight decay coefficient (between 0 and 0.001) and the RIS hyperparameter α (between 0.05 and 0.4) on the U-shaped maze and the ω-shaped maze (we found that the relative simplicity of U-shaped maze compared to the other environments meant that design architectures useful to solve the U-shaped maze would not always generalize to the other tasks) and select the final set of hyperparameters based on the return curve. We then run the final experiment on all environments, keeping the same hyperparameters for all variants of the HyperCombinator (*i.e.* HC8, HC16 and HC64).

A notable difference with the control experiments is that we found that smaller Gumbel network architectures worked best. Therefore, we selected a [64, 64, 64] architecture for the navigation experiments, as opposed to [1024, 1024, 1024] for the control experiments. We have overall observed in both control and navigation experiments that a wider and deeper Gumbel network architecture usually requires a stronger regularization to perform well on the task. It is possible that increasing even more the regularization could help the [1024, 1024, 1024]-architecture solve the maze tasks. The success of the smaller architecture and the failure of the bigger one, which such a high regularization of the Gumbel network, is another clue that the key to solving the maze might be to find a trainable and sufficiently regularized architecture that efficiently combines the different sub-policies.

Evaluation details During evaluation, all actors act deterministically using only the mean of the predicted action distribution $\mu(\tilde{\pi}(s, g))$, without exploration noise (as opposed to sampling from the predictive action distribution during training), which guarantees the piecewise-linearity of the policy. We remark that this departs from the base code, that evaluated stochastic agents.

Every 10000 steps, we roll out 5 evaluation episodes and report the mean success score, *i.e.* 1 if the agent reached the goal and 0 else. We report all results and curves using 10 seeds for each agent. We draw the mean performance as a colored line, as well as a 95% bias-corrected and accelerated bootstrap confidence interval in a lighter shade (9999 resamples).

Compute We ran all the experiments on an internal cluster. All the GPUs were NVIDIA Tesla V100, with 16GB memory available. The CPUs were Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz Each seed was allocated 1 GPU, 10 CPUs, and 64GB of RAM. We detail the compute budget to reproduce the experiments in Table. 3.

Experiment	**# models**	**# envs**	**# seeds**	**Avg. duration**	**Compute**
Full results (Fig. 15)	4	4	10	23 hours	153 GPU days
Temperature ablation (Fig. 20)	8	1	10	11 hours	37 GPU days

Table 3: Compute budget for the navigation experiments.

[4]https://github.com/denisyarats/pytorch_sac
[5]Shared between the mean net and the log std net
[6]https://github.com/elliotchanesane31/RIS
[7]https://github.com/elliotchanesane31/RIS

Hyperparameter name	**Value**
Common (RIS defaults)	
Discount factor γ	0.99
Success distance threshold	0.5
Burn in	1e4
Replay buffer size	1e6
Learning rate for the high-level policy $\lambda_{\text{high-level}}$	1e-4
Learning rate for the critics λ_Q	1e-3
Learning rate for the policy λ_π	1e-3
Critic architecture	[256, 256]
Critic empirical moving average ratio	0.005
Critic target update frequency	1
High-level policy architecture	[256, 256]
sub-policy assignation entropy coefficient	1e-4
ϵ	1e-16
HER replay buffer goals ratio	0.5
λ	0.1
$\log \sigma_{\text{min}}$	-20
$\log \sigma_{\text{max}}$	2
RIS actor-specific	
Actor architecture[5]	[256, 256]
Batch size	2048
α	0.1
HyperCombinator-specific	
Gumbel net architecture	[64, 64, 64]
Batch size	1024
sub-policy assignation entropy coefficient λ_{assig}	0.01
Weight decay coefficient $\lambda_{\text{weight decay}}$	0.0001
Gumbel temperature	7
α	0.3

Table 4: Full list of hyperparameters in the navigation experiments.

Algorithm 3 `RIS (with HyperCombinator actor)`

Require: Replay Buffer $\mathcal{D}$
Require: Actor parameters φ
Require: Double critic parameters η_1, η_2
Require: Double critic target parameters $\overline{\eta}_1$, $\overline{\eta}_2$
Require: High-level policy parameters
Require: Hyperparameters from Table 4
Require: N ▷ Maximum number of timesteps
Require: s_0, g_0 ▷ Initial state
while $t < N$ **do**
$a_t \sim \mu(\tilde{\pi}(s_t, g_t))$
$s_{t+1} \sim P(\cdot|s_t, a_t)$ ▷ Sample the next state from the environment
$r_{t+1} = R(s_t, a_t, s_{t+1})$
$\mathcal{D} = \mathcal{D} \cup (s_t, a_t, r_{t+1}, s_{t+1}, d_{t+1})$ ▷ Update replay buffer; d_{t+1} indicates a terminal transition

$(s, a, r, s', d, g) \sim \mathcal{D}$ ▷ Sample from replay buffer using HER
Launch routine `UpdateCritic` ▷ See paper [25] and code[6]
Launch routine `UpdateHighLevelPolicy` ▷ See paper [25] and code[7]
Launch routine `UpdateActorAndAlpha` (see Alg. 4)
$\overline{\eta}_1 = (1 - \text{critic ema ratio}) * \overline{\eta}_1 + \text{critic ema ratio} * \eta_1$
$\overline{\eta}_2 = (1 - \text{critic ema ratio}) * \overline{\eta}_2 + \text{critic ema ratio} * \eta_2$
end while

Algorithm 4 `UpdateActorAndAlpha`

Require: s, g ▷ Batch of states and goals sampled from the replay buffer
$a \sim \mu(\mathcal{N}(\tilde{\pi}(s, g), \exp(t'(\log \sigma(s, g))))$ ▷ $\tilde{\pi}$ is a Gumbel network and $\log \sigma$ an independent MLP instead of being a jointly parametrized MLP in the base case.
$\mathcal{L}_{\text{assig}} = H(\text{mean}(\hat{a}(s, g)))$ ▷ Compute entropy of average sub-policy assignation
$\mathcal{L}_{\text{weight decay}} = ||W||_2^2$ ▷ Magnitude of the Gumbel network MLP last layer weights W
$Q = \min(Q_{\eta_1}(s, a), Q_{\eta_2}(s, a))$
$\mathcal{L}_\pi = \text{mean}\,(\alpha \log \pi(a|s, g) - Q) - \lambda_{\text{assig}} \mathcal{L}_{\text{assig}} + \lambda_{\text{weight decay}} \mathcal{L}_{\text{weight decay}}$ ▷ Minimize the magnitude of the Gumbel network last layer weights
$\varphi = \varphi - \lambda_\pi \nabla_\varphi \mathcal{L}_\pi$

E.2 Performance of HC compared to RIS

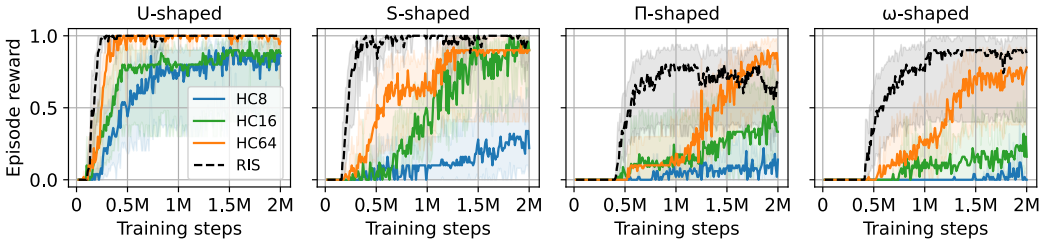

Figure 15: HC actors learn to solve the navigation tasks with a reduced sample efficiency.

E.3 Additional policy visualizations

In Fig. 3 we illustrated a typical sequence of sub-policies followed by HC16 on the S-shaped maze. In this section, we propose a wider variety of sub-policy sequences, showing examples where the emergent temporal abstraction appears clearly, and some where this temporal abstraction is harder to spot, or, interestingly, is lost over training. For each situation, we show sub-policy sequences taken during evaluation after .5M, 1M, 1.5M and 2M training steps.

HC8, U-shaped maze (Fig. 16) This figure shows a well-performing HC8 policy in U-shaped maze. We note the clear separation of three phases in the bottom plot, as the quadruped learns to go down the corridor in the first phase, using mostly sub-policy 0 and 1, then turns right and navigates the bottom corridor with sub-policies 2 and 3, before re-using sub-policies 0 and 1 to go up the right corridor and solving the maze.

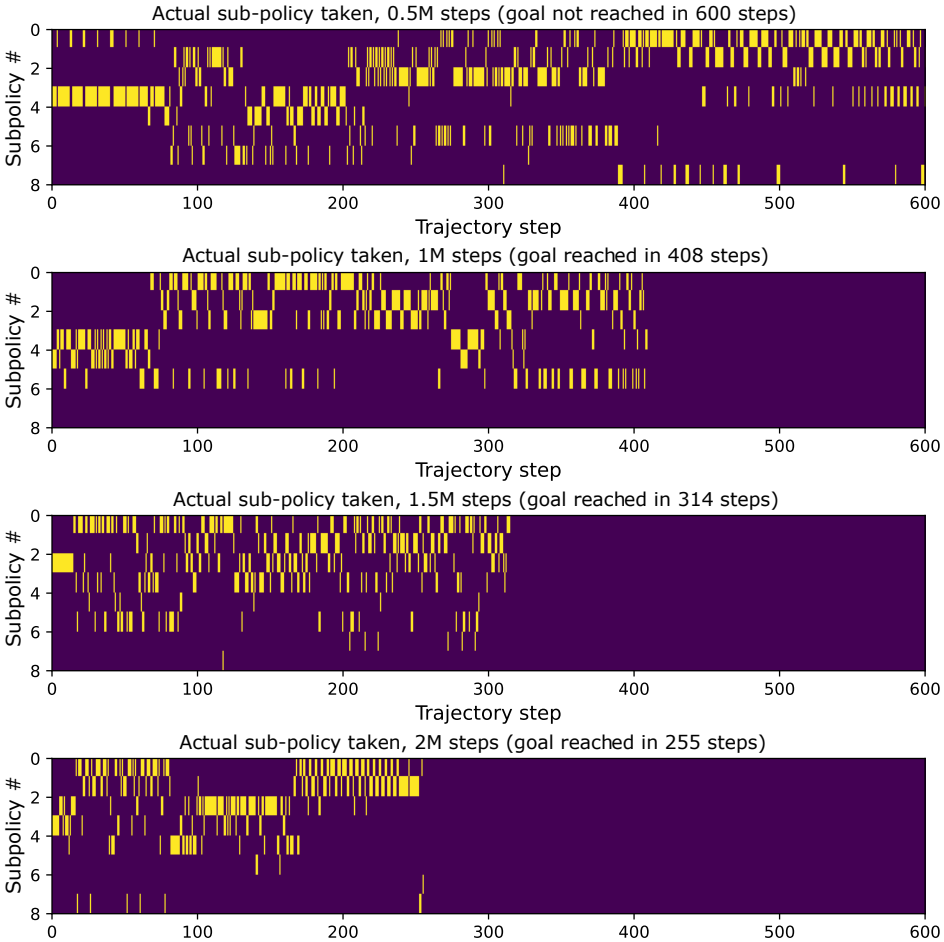

Figure 16: Example of a well-performing HC8 agent in the U-shaped maze.

HC64, ω-shaped maze (Fig. 17) This figure illustrates the policies learnt by HC64 on the challenging ω-shaped maze. We particularly note the appearance of "diagonals", where the same sequence of sub-policy is repeated several times. These diagonals recall the sub-policy sequence obtained for Cheetah run in the control experiments, as the agent learnt to chain its sub-policies to move itself. After 100 timesteps in the bottom figure (final evaluation after 2M training steps), HC64 switches from one set of repeated diagonals to another, and barely re-uses the sub-policies forming the first set of repeated diagonals until the end of the trajectory. We conclude that the first set of repeated diagonals was specialized to the first phase of the trajectory.

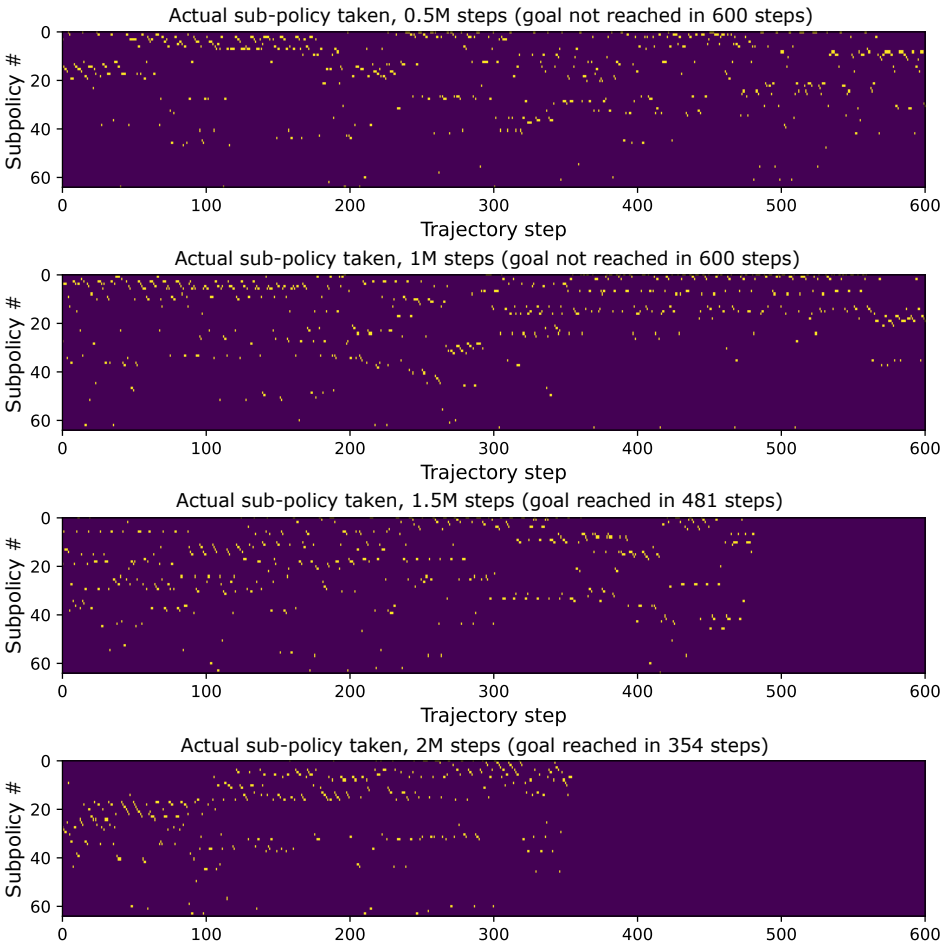

Figure 17: Example run of HC64 solving the ω-shaped maze. We note the appearance of "diagonals" that are reminiscent of the control experiments.

HC64, U-shaped maze (Fig. 18) We have seen in the previous example that a structure emerged in the sub-policies, learning to repeat chains of sub-policies ("diagonals") to move itself. Fig. 18 illustrates the possible effects of long training on this structure. This structure is particularly present after .5M timesteps, and the sub-policies in the second part of the trajectory are not used during the first part of the trajectory. However, as training progresses, the structure is progressively forgotten, which makes the bottom sub-policy sequence plot harder to read. One possible explanation is that the agent learns to "overfit" the task, resulting in a lower re-use of sub-policies, but a better performance (as evidenced by the progressively lower number of timesteps required to solve the task). This might also be due to the high regularization that we enforced on the Gumbel network, forcing agents to use as many sub-policies as possible.

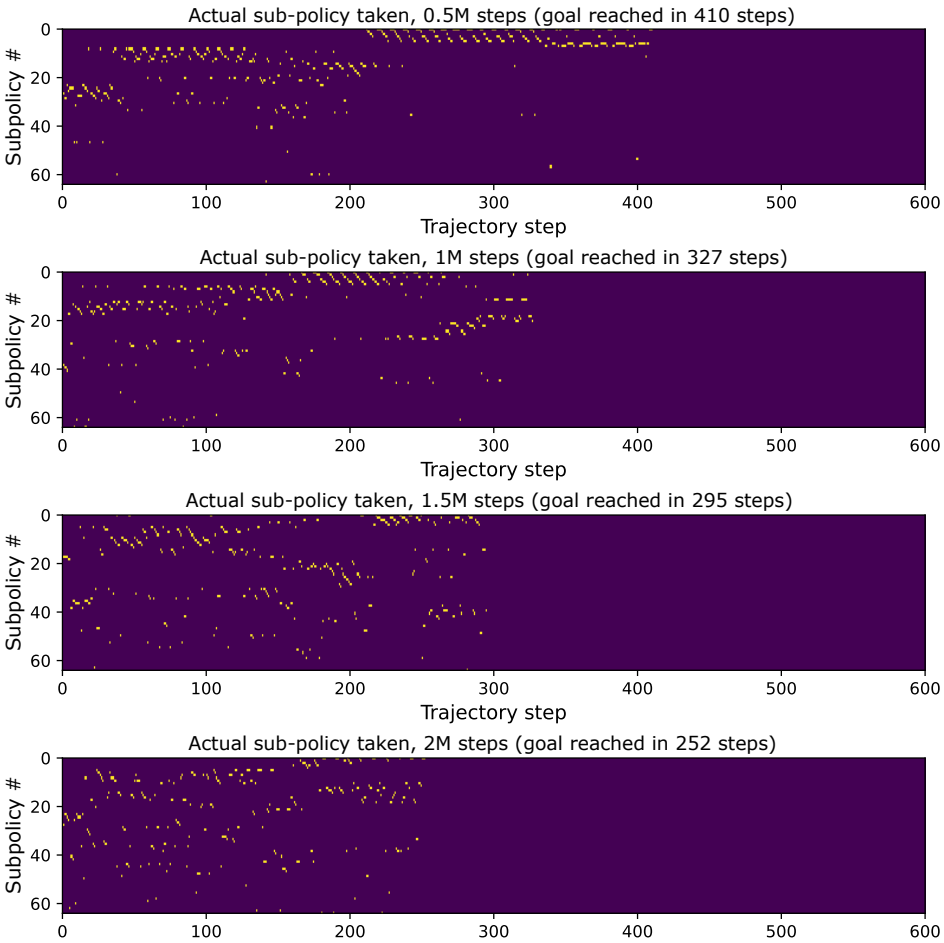

Figure 18: The learnt structure (top plot) can get lost as the agent learns to solve efficiently the maze (bottom plot).

HC16, Π-shaped maze (Fig. 19) Fig. 19 finally provides an example of a policy where structure is difficult to observe. HC16 learns to solve the Π-shaped maze after 1.5M timesteps, but no particular structure emerges. In addition, we see one example of HC16 forgetting how to solve the maze in the bottom plot. We note that despite this fact, the HyperCombinator architecture guarantees that we can bring transparency to the interaction of the agent with the environment, conditioned on the knowledge of the sub-policy chosen. Notably, we still have access to all the sub-policies that define how HC16 interacts with the environment.

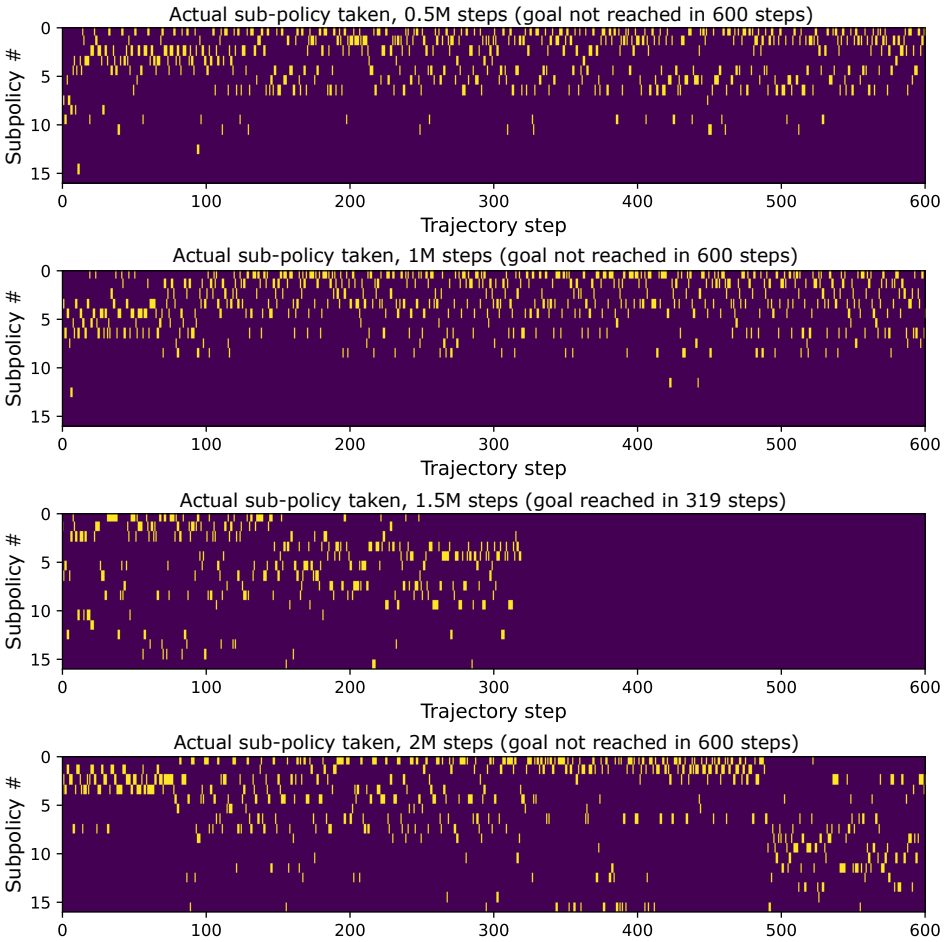

Figure 19: HC16 can learn to solve the maze without displaying obvious structure in its sequence of sub-policies (3rd plot).

E.4 Effect of Gumbel temperature on performance

We run an ablation to see the effect of changing the temperature hyperparameter, all other hyperparameters being held the same as in the navigation part of Sec. 4. We evaluate the HyperCombinator variants for temperature values of .5, .66, 1, 2, 3, 5, 7 and 10 in the U-shaped maze. In general, a higher temperature value tends to increase the stochasticity of the Gumbel network, meaning that the Gumbel network will be more uncertain of which sub-policy to choose (and will also increase gradient sharing during the backward pass).

We visualize the results of the ablation in Fig. 20. Overall, low temperature values impede learning, and the resulting agents do not succeed in the U-maze. Higher temperature values led to better performing agents, though the overall return curve remains noisy in all cases.

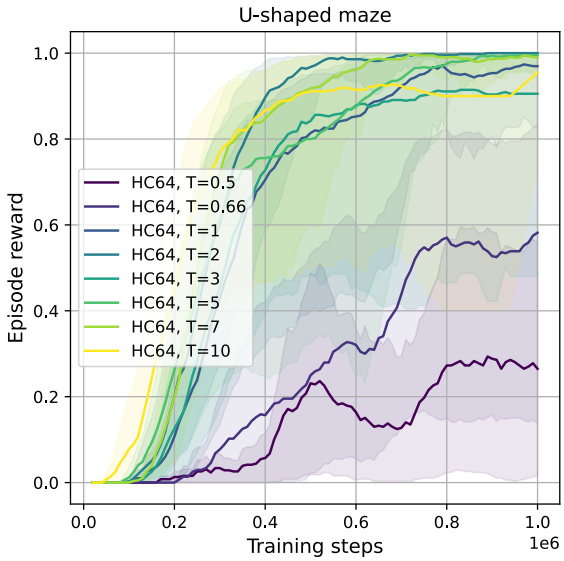

Figure 20: Low temperature values lead to insufficient regularization and HC64 fails to solve the maze. Increasing the temperature overall leads to better performance, except for the highest temperature.