SCALABLE OPTION LEARNING IN HIGH THROUGHPUT ENVIRONMENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Hierarchical reinforcement learning (RL) has the potential to enable effective decision-making over long timescales. Existing approaches, while promising, have yet to realize the benefits of large-scale training. In this work, we identify and solve several key challenges in scaling online hierarchical RL to high-throughput environments. We propose Scalable Option Learning (`SOL`), a highly scalable hierarchical RL algorithm which achieves a ~35x higher throughput compared to existing hierarchical methods. To demonstrate `SOL`'s performance and scalability, we train hierarchical agents using 30 billion frames of experience on the complex game of NetHack, significantly surpassing flat agents and demonstrating positive scaling trends. We also validate `SOL` on MiniHack and Mujoco environments, showcasing its general applicability. [Our code will be open sourced].

1 INTRODUCTION

Training agents to effectively solve decision-making tasks spanning long timescales is a fundamental challenge in reinforcement learning (RL) and control. This problem is difficult because the optimization landscape at the lowest level of sensorimotor control is often hard to optimize, due to sparsity of rewards or local minima. Consider a human whose goal is to go from New York to Paris. When viewed as an RL problem, where actions consist of joint movements and the cost is the distance to Paris, gradient information is often uninformative or misleading. The optimization landscape is rife with local minima, such as the agent becoming stuck in the easternmost corner of a room; and globally optimal trajectories, such as taking the subway to the airport, are likely to encounter many local increases in the cost function, making them difficult to discover.

Hierarchy presents itself as a natural approach to address this challenge (Sutton et al., 1999). By decomposing a long task into a hierarchy of decisions at different timescales, one can hope to ease the problems of credit assignment and exploration, which become increasingly difficult as the horizon of the problem increases. At higher levels of the hierarchy, actions span longer timescales and are thus fewer, making for shorter decision-making tasks. Meanwhile, lower levels of the hierarchy aim to solve sub-tasks determined by the higher levels, which are also shorter and thus easier to optimize. An effective solution to hierarchical RL could benefit many areas of AI which involve long-horizon tasks where progress is limited by difficult exploration, delayed rewards and the need to coordinate different behaviors.

A significant body of work has explored ways to incorporate hierarchy into RL algorithms, through options (Sutton et al., 1999; Precup & Sutton, 2000; Bacon et al., 2017), feudal RL (Dayan & Hinton, 1992; Vezhnevets et al., 2017), and other manager-worker architectures (Nachum et al., 2018; Gürtler et al., 2021; Li et al., 2020; Levy et al., 2017). These methods have shown promising benefits of hierarchy over flat policies and laid important conceptual foundations. Nevertheless, by modern AI standards, they have remained in the relatively small data regime. Whereas flat RL agents and computer vision models are routinely trained on billions of samples (Radford et al., 2021; Kirillov et al., 2023; Ravi et al., 2025; Espeholt et al., 2018; Petrenko et al., 2020; Matthews et al., 2024) and language models on trillions of tokens (Brown et al., 2020; OpenAI, 2024; Kaplan et al., 2020; Touvron et al., 2023; Dubey et al., 2024; Team, 2024), existing hierarchical agents are typically trained on millions of samples only—several orders of magnitude less data. Therefore, hierarchical RL has yet to realize the benefits of large-scale training, which has driven progress in

many other areas of machine learning (Silver et al., 2016; 2017; Wijmans et al., 2019; Le et al., 2023; Brown et al., 2020; OpenAI, 2024; Kaplan et al., 2020).

In this work, we take a step towards bridging this gap and present Scalable Option Learning (SOL), a highly scalable hierarchical policy gradient algorithm. We identify and solve several challenges which prevent straightforward scaling of hierarchical agents via GPU parallelization, enabling us to train on billions of samples on a single GPU. SOL achieves ~35-580x faster throughput compared to existing hierarchical RL algorithms. We apply SOL to the complex, open-ended NetHack Learning Environment (NLE) (Küttler et al., 2020) and train hierarchical agents for 30 billion steps, significantly surpassing flat agents and demonstrating promising scaling trends. We additionally evaluate our algorithm on simpler MiniHack (Samvelyan et al., 2021) and PointMaze environments (de Lazcano et al., 2024), showcasing its general applicability.

2 Background and Problem Setting

2.1 Markov Decision Processes

We consider a standard Markov decision process (MDP) (Sutton & Barto, 2018) defined by a tuple $(\mathcal{S}, \mathcal{A}, \mu, p, R, \gamma)$ where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, μ is the initial state distribution, p is the transition function, R is the reward and $\gamma < 1$ is a discount factor. At the beginning of each episode, a state s_0 is sampled from μ. At each time step $t \geq 0$, the agent takes an action a_t conditioned on s_t, which causes the environment to transition to a new state $s_{t+1} \sim p(\cdot|s_t, a_t)$ and a reward $r_t = R(s_t, a_t)$ to be given to the agent. The goal of the agent is to learn a policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$ which maximizes the sum of discounted returns $\mathbb{E}_\pi[\sum_{t=1}^{\infty} \gamma^t r_t]$. We adopt the fully-observed MDP framework for simplicity of notation, however, states can be replaced by observation histories without loss of generality and our experiments include partially-observed environments.

2.2 Options

The options paradigm (Sutton et al., 1999; Precup & Sutton, 2000) provides a framework for decision-making at different levels of temporal abstraction. Each option $\omega \in \Omega$ represents a temporally extended behavior, and is defined by a tuple $(\pi_\omega, \mathcal{I}_\omega, \beta_\omega)$. Here $\pi_\omega : \mathcal{S} \to \Delta(\mathcal{A})$ is the option policy defining the agent's behavior while the option is being executed, $\mathcal{I}_\omega \subseteq \mathcal{S}$ is the initiation set of states where the option can be started, and $\beta_\omega : \mathcal{S} \to \{0, 1\}$ is the termination function indicating when the option should be ended. In addition to the option policies π_ω, there is a controller policy $\pi_\Omega : \mathcal{S} \to \Delta(\Omega)$ which determines which option to execute next in a given state. Note that Ω could be infinite and continuously parameterized, in which case π_Ω has a continuous action space.

We adopt the call-and-return paradigm (Bacon et al., 2017; Klissarov & Precup, 2021) where options are sequentially executed. At the first timestep, an option $\omega \sim \pi_\Omega(\cdot|s_0)$ is sampled. The agent then executes actions $a_t \sim \pi_\omega(\cdot|s_t)$ until $\beta_\omega(s_t) = 1$, at which point a new option $\omega' \sim \pi_\Omega(\cdot|s_t)$ is sampled. The agent then executes actions sampled from $\pi_{\omega'}$ until the termination function $\beta_{\omega'}$ is activated, and the process continues.

A central question is how to define or learn the options $(\pi_\omega, \beta_\omega, \mathcal{I}_\omega)_{\omega \in \Omega}$ as well as the controller policy π_Ω. Different settings which have been studied include: i) predefining the option policies π_ω and subsequently learning π_Ω (Sutton et al., 1999) ii) pre-specifying or incrementally adding goal states and learning option policies using distances to goals as rewards (McGovern & Barto, 2001; Stolle & Precup, 2002; Menache et al., 2002; Şimşek & Barto, 2008), iii) jointly learning both option policies and the controller policy end-to-end using the task reward alone (Bacon et al., 2017; Li et al., 2020; Nachum et al., 2018; Klissarov et al., 2017; Klissarov & Precup, 2021).

In this work, for most of our experiments we assume access to a set of intrinsic reward functions $\{R_\omega\}_{\omega \in \Omega}$, and focus on jointly learning corresponding option policies π_ω and the controller π_Ω in a scalable manner. This assumes more prior knowledge than purely end-to-end options methods, however, we also show in Section 5.3 that our method can be combined with methods for automatic reward synthesis which do not assume any prior knowledge. Our scalable algorithm is agnostic to how option rewards are defined, and can be used whether they are handcoded or learned.

2.3 HIGH-THROUGHPUT RL

Several algorithms and libraries have been proposed to efficiently train RL agents on billions of samples, such as IMPALA (Espeholt et al., 2018), MooLib (Mella et al., 2022), RLLib (Liang et al., 2018; Wu et al., 2021), Sample Factory (Petrenko et al., 2020) and Pufferlib (Suarez, 2024). A common feature is asynchronous data collection paired with parallelized policy updates. A set of actors, typically operating across multiple CPU cores, collect experience from many parallel environment instances. Concurrently, a learner process receives batches of experience and updates the policy using parallelized GPU operations. The objective optimized is typically the policy gradient objective (Williams, 1992) with modifications such as PPO's trust region constraints (Schulman et al., 2017) and/or IMPALA's V-trace off-policy correction. Additional systems-level optimizations such as double-buffered sampling can also be included (Petrenko et al., 2020). However, all the above high-throughput RL libraries operate using flat, non-hierarchical agents. We next describe the challenges associated with parallelizing hierarchical agents, and our approach to solving them.

3 METHOD

In this section we introduce Scalable Option Learning (`SOL`), our hierarchical RL method designed to optimize both performance and computational throughput.

3.1 OBJECTIVE

At a high level, we jointly optimize actor-critic objectives (Konda & Tsitsiklis, 1999) for all the options as well as the controller, each of which consists of a policy loss, a value loss, and an exploration loss. In addition to the controller policy π_Ω and option policies π_ω, we learn controller and option value function estimators $\hat{V}_\Omega$ and $\hat{V}_\omega$, which estimate the future returns of the controller and option policies using their respective reward functions. Each time the controller is called, it outputs both the next option ω to execute and the number of time steps $l \in L = \{1, 2, 4, \ldots, 128\}$ to execute the option for. This is equivalent to using an augmented option set $\bar{\Omega} = \Omega \times L$, and allows the controller to adaptively select option lengths without task-specific tuning. For simplicity of notation we use Ω rather than $\bar{\Omega}$ in the following, but always use this mechanism unless otherwise specified.

Policy Objective The policy objective we seek to optimize is:

$$\mathcal{L}_{\text{policy}} = \mathbb{E}_\tau \left[\sum_{t=0}^{\infty} \sum_{\omega \in \Omega} \left(\delta_{z_t=\pi_\Omega} \log \pi_\Omega(\omega|s_t) A^{\text{task}}(s_t, \omega) + \delta_{z_t=\pi_\omega} \log \pi_\omega(a_t|s_t) A^\omega(s_t, a_t) \right) \right]$$

where $\tau = (s_0, z_0, a_0, r_0, s_1, z_1, a_1, r_1, ...)$ represents trajectories generated by the agent. The variable z_t represents the policy being executed at time t, which can be either the controller π_Ω or any of the option policies π_ω, and δ represents a one-hot indicator. Here A^{task} represents the advantage associated with the controller π_Ω and task reward R, and A^ω represents the advantage associated with the option policy π_ω and option reward R_ω. On timesteps where the controller selects a new option, no environment-level action is taken, so a duplicated state is inserted to be acted upon by the newly selected option on the next timestep. More details and exact definitions are in Appendix E.1.

Value Objective Our value objective is given below, where $\hat{V}_\Omega$ and $\hat{V}_\omega$ denote the agent's parameterized value estimates of the task reward R and option reward R_ω, respectively:

$$\mathcal{L}_{\text{value}} = \mathbb{E}_\tau \left[\sum_{t=0}^{\infty} \left(\delta_{z_t=\pi_\Omega} (V_\Omega(s_t) - \hat{V}_\Omega(s_t))^2 + \sum_{\omega \in \Omega} \delta_{z_t=\pi_\omega} (V_\omega(s_t) - \hat{V}_\omega(s_t))^2 \right) \right]$$

An important property to note is that the definitions of A^ω and V^ω do not depend on any of the other options or the controller, however their estimators are trained on the distribution of states induced by the entire system. This is designed to produce independent options, while avoiding hand-off errors resulting from training each option separately. At first glance, it might be unclear how to

estimate V^ω from trajectories where different options are called, since it depends on an infinite sum of rewards induced by following a single option π_ω. We address this by applying the following recurrence relation and approximation:

$$V^\omega(s_t) = \mathbb{E}_{\pi_\omega}[\sum_{k=0}^{\infty} \gamma^k R_\omega(s_{t+k}, a_{t+k}|s_t)] \approx \mathbb{E}_{\pi_\omega}[\sum_{k=0}^{K} \gamma^k R_\omega(s_{t+k}, a_{t+k}|s_t) + \gamma^{K+1}\hat{V}^\omega(s_{t+K})]$$

Here K is the remaining number of steps the current option ω is executed for before a different option is called. During training, we approximate this expectation with a single bootstrapped Monte Carlo rollout, and the resulting scalar is then used as a target for $\hat{V}^\omega(s_t)$.

Exploration Objective It is standard to include a bonus on the entropy $\mathcal{H}$ of a policy to encourage local exploration (Williams & Peng, 1991; Mnih et al., 2016). We include these on both the controller policy and the option policies:

$$\mathcal{L}_{\text{explore}} = \mathbb{E}_\tau \left[\sum_{t=0}^{\infty} \delta_{z_t=\pi_\Omega} \mathcal{H}(\pi_\Omega(\cdot|s_t)) + \sum_{\omega \in \Omega} \delta_{z_t=\pi_\omega} \mathcal{H}(\pi_\omega(\cdot|s_t)) \right]$$

Our global objective is the sum of the above objectives, and is trained on the data generated by the agent. We next discuss how to optimize our global objective in high-throughput settings.

3.2 Scaling Challenges

Before discussing the details of our system design, it is important to understand why scaling hierarchical RL methods is not straightforward. Hierarchical systems execute a sequence of policies, chosen from $\Pi_\Omega = \{\pi_\Omega\} \cup \{\pi_\omega\}_{\omega \in \Omega}$, which depends on the observations. Because of this, in a batch of trajectory segments of size $B \times T$, any given slice of size B at time t will likely correspond to several different policies, with correspondingly different reward functions (see Figure 1). As a result, both the forward passes through the policy network, which are needed to compute the action probabilities and value estimates, as well as the return or advantage computations, which operate on the different option and controller rewards, are difficult to parallelize. Current hierarchical methods such as (Nachum et al., 2018; Gürtler et al., 2021; Levy et al., 2017; Klissarov et al., 2017) process a single trajectory at a time, which is sufficient for the continuous control settings in which they are tested which require a relatively small number of samples (in the millions). However, complex, open-ended environments such as NetHack typically require billions of samples, which in turn requires more scalable hierarchical methods.

3.3 System Design

We address these challenges through three design choices: i) a single neural network with multiple action heads and an indexing vector which represents both high and low-level policies, ii) an environment wrapper in the actor workers which tracks active policies and computes corresponding rewards, and iii) efficient parallelized masking when computing the advantages and value targets for each policy. These enable leveraging existing high-throughput asynchronous RL libraries such as Sample Factory (Petrenko et al., 2020). We provide a system overview in Figure 1 and describe each component in detail next.

Architecture In order to process trajectory batches efficiently in parallel, we adopt a single neural network architecture which represents all the option policies as well as the controller policy. In addition to the environment observation, the network receives a one-hot vector u of dimension $|\Omega|+1$ which indicates which of the policies in $\{\pi_\omega\}_{\omega \in \Omega} \cup \{\pi_\Omega\}$ to represent. The network's output space is $\mathcal{A} \times \Omega \times L$, where L is the set of possible option lengths. For each input observation, the network outputs three distributions: a distribution over environment actions $\Delta(\mathcal{A})$, a distribution over options $\Delta(\Omega)$, and a distribution over option lengths $\Delta(L)$. If u represents one of the option policies π_ω, then $\Delta(\mathcal{A})$ is kept and sent to the environment wrapper which we describe next. Otherwise if u represents π_Ω, the distributions over options and option lengths $\Delta(\Omega), \Delta(L)$ are sent instead. See Appendix E.2 for an illustration and more details.

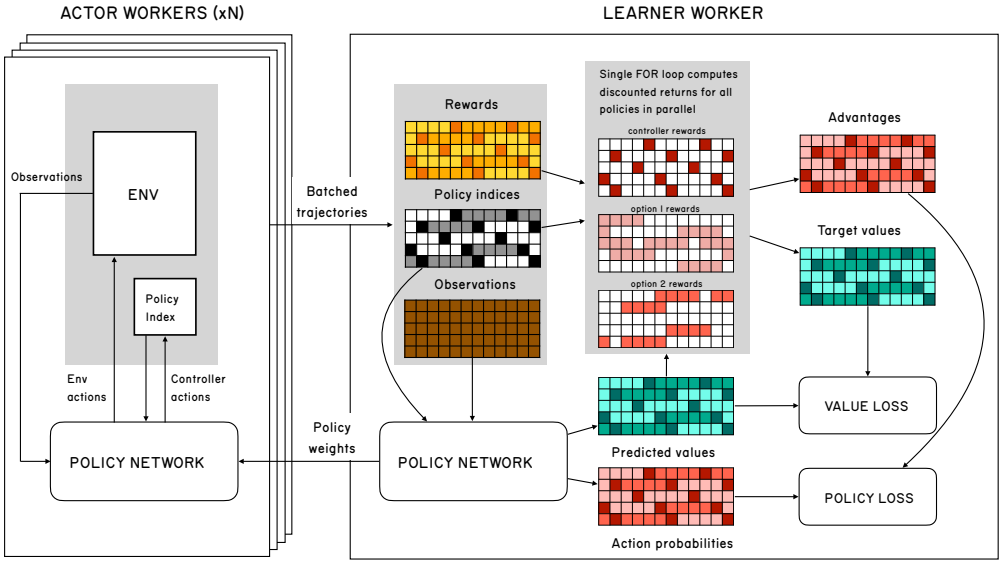


Figure 1: System overview. We use a single network with an augmented action space to represent both option policies and the controller policy, enabling batched forward passes for all policies at once. In the actor workers, a modified environment wrapper tracks the policy index based on the high-level controller actions, and routs the low-level actions to the environment. The learner worker continuously updates the policy with batched forward passes through the policy network and efficient tensorized return computations. Different shades indicate quantities associated with different policies: darkest is the controller, the other two are options ω_1 and ω_2.

Actor Workers The second component is an environment wrapper in each actor worker which tracks which policy is currently being executed, switches them based on the termination conditions and controller actions, computes option rewards, and duplicates the last observation whenever the controller policy is called. Specifically, a variable $\mathtt{p}$ tracks which policy is currently active, and at each time step is converted to the one-hot vector u which is fed as input to the network in addition to the observation. Each time an option terminates after being executed for the number of steps last output by the controller, $\mathtt{p}$ changes based on the controller distribution $\Delta(\Omega)$ output by the network, and the next option length $l \sim \Delta(L)$ is recorded in the environment wrapper. Otherwise, an action is sampled from $\Delta(\mathcal{A})$ and routed to the environment instance. The rewards for each of the option policies π_ω are computed using R_ω from the observation directly. The reward used to train the controller policy π_Ω, which is the sum of the task rewards for the option that it calls, depends on the future execution of that option and is computed in the learner thread. Pseudocode providing more details is included in Appendix E.3.

Learner Worker Given the above two components, the learner process which updates the policy network receives the following tensors: observations O of size $B \times T \times d$ (where d is the observation dimension), rewards R of size $B \times T$, episode terminations of size $B \times T$, and one-hot policy indices P of size $B \times T \times |\Omega| + 1$. Note that the rewards in R are of mixed types: R_{ij} is of the type corresponding to the policy P_{ij}. The first step is to fill in the rewards R corresponding to the controller policy, which could not be previously computed in the actor threads since they depend on the future execution of the option policy that the controller calls. This is done with a single `FOR` loop compiled to C using Cython (Behnel et al., 2011). Next, for each policy indexed by P, the learner process must compute two main quantities: the empirical returns and the advantages. By using tensorized operations and caching various intermediate quantities for each policy, we are able to compute both of these quantities for all policies simultaneously, using their respective rewards, with a single backward `FOR` loop over the time dimension T. At a high level, this is done by: i) tracking which policies have had their bootstrapped values already added to the cumulative returns, ii) at each time step t, adding the rewards $R[:, t]$ to the appropriate policy returns based on the current policy indices $P[:, t]$, and iii) appropriately handling episode terminations for each policy, based on

the last observation during which it is executed. Finally, V-trace off-policy corrections are applied to both the controller and option policies, to account for potential lags between the actor and learner workers in asynchronous settings. See Appendix E.4 for source code with full details.

Throughput Comparison We instantiate our algorithm using the Sample Factory codebase (Petrenko et al., 2020). In Figure 2 we compare the throughput of our algorithm with that of public implementations of four other hierarchical algorithms: HIRO (Nachum et al., 2018), Option-Critic (Bacon et al., 2017), Multiple Option Critic (MOC) (Klissarov & Precup, 2021) and the hierarchical training implemented in RLLib (Wu et al., 2021). For HIRO and Option-Critic we use the same NLE encoder as in our experiments in Section 5, for MOC and RLLib we used a visual rendering pipeline instead for code compatibility reasons (`SOL` evaluated with the same pipeline gets similar throughput as with our standard encoder). We used the same hardware for all comparisons and tuned the batch size and number of environments where applicable to obtain the best throughput. Additional details can be found in Appendix C.3. Our algorithm is ~35x-580x faster than the other four hierarchical methods, and retains 86% of the speed of the flat agent. We note that the design decisions above are not library-specific and our algorithm could be instantiated with other implementations which use asynchronous actor workers to collect experience and a learner worker to perform batch policy updates on the GPU, which is a common design in distributed RL (Espeholt et al., 2018; Küttler et al., 2019; Mella et al., 2022; Suarez, 2024).

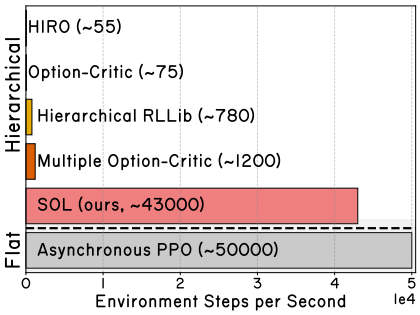


Figure 2: Throughput comparison of hierarchical and flat methods on the NLE.

Different Algorithm Instantiation Our system is general and enables instantiating high-throughput versions of certain existing hierarchical algorithms or designing new ones. For example, by setting all option rewards to equal the task reward, we recover an objective analogous to HiPPO (Li et al., 2020), a hierarchical PPO variant that learns using the task reward only, which we include in our comparisons. Alternatively, some or all of the option rewards can be produced by methods for automatic reward synthesis, such as DIAYN (Eysenbach et al., 2019), which we investigate in Section 5.3. We discuss other possibilities in Appendix A as potential future work. .

4 Related Work

Early hierarchical methods focus on the tabular setting, such as Hierarchical Q-learning (Wiering & Schmidhuber, 1997; Singh, 1992a;b; Kaelbling, 1993) and feudal RL (Dayan & Hinton, 1992).

Options (Sutton et al., 1999) provide a general framework for temporally extended decision-making. The original work considers methods for learning value functions or models over a set of hardcoded options in the tabular setting. Several follow-up works have explored learning options instead, for example by identifying bottleneck states (McGovern & Barto, 2001; Stolle & Precup, 2002; Menache et al., 2002; Şimşek & Barto, 2008) and defining rewards based on reaching them. The Option-Critic architecture (Bacon et al., 2017) jointly learns the option policies with a value function over options using only the task reward. However, the benefits of this method were primarily in transfer to new tasks, and it was not shown to clearly improve over a flat policy on the original task.

Several hierarchical methods based on deep RL have been proposed and evaluated on continuous control environments, such as HIRO (Nachum et al., 2018), HAC (Levy et al., 2017), and HiTS (Gürtler et al., 2021). While improving over flat policies, these methods focus on sample-efficient learning through off-policy learning, rather than scaling to large numbers of samples. As a result, implementations are single-process and not designed to scale to billions of samples.

Closer to our work are hierarchical variants of PPO (Schulman et al., 2017), which include HiPPO (Li et al., 2020), PPOC (Klissarov et al., 2017) and MOC (Klissarov & Precup, 2021). These methods optimize a special case of our objective where all option rewards equal the task reward. PPOC and MOC use the more highly optimized OpenAI baselines library (Dhariwal et al., 2017) which

uses parallelized experience collection, and are an order of magnitude faster than the above methods. However, MOC is still over an order of magnitude slower than ours.

Feudal Networks (Vezhnevets et al., 2017) are another deep RL architecture, which is conceptually similar to HIRO but defines goals in embedding space rather than the original state space. This work reported promising results, however the official code has not been released and we were unable to find third-party reimplementations which reproduced their results, making comparisons difficult.

Our approach is related to the joint skill learning described in MaestroMotif (Klissarov et al., 2025), but differs in several ways: i) we learn our controller jointly with the options, whereas they use a frozen LLM, ii) we do not use hardcoded initiation and termination conditions, and iii) we use separate value function bootstrapping across the different options and controller policy.

Similar to `SOL`, Agent57 (Badia et al., 2020) achieves high throughput by indexing a family of policies with a shared neural network using a one-hot vector, which are then selected by a controller trained to maximize the task reward. A key difference is that `SOL` can switch between different policies within a single episode, whereas Agent57 executes the same policy for the full episode. This in turn requires several of the design choices in Section 3.3, see Appendix G.1 for more discussion.

There are a number of works on hierarchical agents for robotics and embodied AI (Heess et al., 2016; Peng et al., 2017; Yokoyama et al., 2023; Szot et al., 2021; Qi et al., 2025; Pertsch et al., 2020; Chen et al., 2023) which learn each option (or skill) separately, and subsequently train a high-level controller to coordinate them. This approach has been shown to be effective in real and simulated robotics settings. However, learning each skill independently requires a starting state distribution that is sufficiently diverse, which may not be the case when the appropriate states may only be reached by mastering and coordinating other skills (see for example Klissarov et al. (2025)).

Recent work by Park et al. (2025) also studies hierarchical RL at scale, but in the offline setting, whereas we focus on developing scalable methods for the online setting. Their results also show the limits of naively scaling flat policies, further highlighting the need for scalable hierarchical methods.

5 Experiments

We evaluate our proposed approach across three environments: MiniHack, NetHack, and Mujoco. The NetHack Learning Environment (NLE) (Küttler et al., 2020) is based on the notoriously difficult roguelike game of NetHack, which requires the player to descend through many procedurally generated dungeon levels to recover a magical amulet. The game involves hundreds of object and monster types, and succeeding requires mastering many capabilities including exploration, combat, resource management and long-horizon reasoning, with successful episodes often lasting $10^4 - 10^5$ steps (Paglieri et al., 2025). MiniHack (Samvelyan et al., 2021) is a framework based on NetHack which enables easy design of RL environments, allowing the targeted testing of agent capabilities.

In the next two sections, we assume access to a given set of option intrinsic rewards R_ω, which we specify (we also investigate automatically discovering R_ω in Section 5.3). We consider the following methods in our comparisons:

- `APPO(task reward)`: a flat asynchronous PPO agent trained with task reward only.
- `APPO(task+option rewards)`: a flat APPO agent trained with a linear combination of task reward and option rewards R_ω, with coefficients optimized by grid search.
- `SOL-HiPPO`: an instantiation of HiPPO (Li et al., 2020) using our scalable framework. This uses only the task reward and no option rewards.
- `SOL`: our hierarchical agent.

This set of comparisons allows us to disentangle the effect of the hierarchical architecture from benefits due to prior knowledge in the form of option rewards. `APPO(task+option rewards)` has access to the the same additional option rewards as `SOL`, and incorporates them with a flat architecture. `SOL` has access to option rewards, and makes use of them through a hierarchical architecture. `SOL-HiPPO` has a hierarchical architecture, but does not use option rewards.

We did not include the prior hierarchical RL methods shown in Figure 2, since they would require an intractably long time to process the same number of samples as `SOL`. An exception is MOC, which

we were able to run on Mujoco. This was possible because MOC is the fastest among prior methods and Mujoco requires much fewer samples than MiniHack and NetHack.

We additionally include Motif (Klissarov et al., 2024), a method which uses an LLM to synthesize intrinsic rewards, in our NetHack experiments since it is the current state of the art, but note that it is orthogonal and can be combined with our method, for example by adding its rewards to one or more of our option rewards. We include this variant in our comparisons under the name `SOL+Motif`. Motif also makes different assumptions: it requires an LLM and observations with a meaningful textual component, hence it cannot be directly applied to environments like Mujoco. Full experiment details, including architectures and hyperparameters, can be found in Appendix C.2.

5.1 MiniHack and NetHack

In addition to NetHack, we design two MiniHack environments which specifically test the ability to perform difficult credit assignment and coordinate different behaviors, while also being fast to run. These environments are described next, with more details in Appendix D.1.

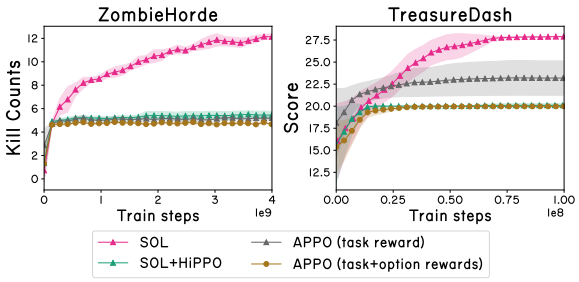


Figure 3: Results on MiniHack. Shaded regions represent two standard errors over 10 seeds.

ZombieHorde The agent is initialized in a room with a horde of zombies it must defeat, which also contains a safe temple area the zombies cannot enter. The zombies are too numerous to fight at once, however, and the agent must periodically retreat to the temple to heal whenever its health becomes too low. This poses a challenging credit assignment problem due to delayed rewards: healing takes dozens of time steps while giving no rewards, but leads to much higher rewards in the long term since it allows the agent to survive future fights. The option rewards here are $R_{\omega_1} = \Delta\texttt{Score}, R_{\omega_2} = \Delta\texttt{Health}$, indicating the per-timestep changes in agent score (which increases for each zombie destroyed) and hit points. For hierarchical methods, the controller's reward is also $\Delta\texttt{Score}$.

TreasureDash The agent is initialized in a hallway filled with piles of gold next to a staircase. The agent has a small number of timesteps in which it can choose to gather gold for a small amount of reward or descend the stairs for a large one-time reward and episode termination. The optimal strategy is to gather as much gold as possible in the given time before descending the stairs on the final timestep. Agents that fail to balance the two competing sources of reward can fall into the local optima of either immediately descending the stairs or gathering gold until the timer runs out. The option rewards are $R_{\omega_1} = \texttt{AtStairs}, R_{\omega_2} = \Delta\texttt{Gold}$, and the controller optimizes total reward.

NetHackScore We use the `NetHackScore` environment from the NLE paper, with the modified `EAT` action used in Klissarov et al. (2024) (see Appendix D.2 for details). The game score serves as the task reward function. The option rewards we consider here are $R_{\omega_1} = \Delta\texttt{Score}, R_{\omega_2} = \Delta\texttt{Health}$. Hierarchical methods also use $\Delta\texttt{Score}$ as their controller reward.

Results On ZombieHorde (Fig. 3, left), all baselines quickly saturate after ~5 kills, while `SOL` keeps improving and achieves a significantly higher final performance. On TreasureDash (Fig. 3, right), `SOL` achieves close to the optimal performance of 28 points, whereas the other methods saturate earlier. In both cases, continued training of the baselines does not result in higher performance, illustrating that scaling alone can be insufficient for tasks involving hard credit assignment. On NetHackScore, we train all agents for 30 billion steps. In Figure 4 (left) we compare LLM-free methods: both flat APPO agents perform similarly, indicating that adding intrinsic rewards (`Health`) to the task reward (`Score`) is not helpful here. This can be explained by the fact that rewarding the change in health discourages the agent from engaging in combat (which causes loss of health) and exploring. `SOL-HiPPO` converges to similar performance as the two flat agents, indicating it is not able to leverage hierarchical structure. `SOL` steadily improves and achieves higher performance than the other agents. In Figure 4 (right) we compare methods which leverage LLMs. Motif improves over the flat APPO baseline, consistent with prior work. Our method `SOL+Motif`, which adds Motif rewards to its `Score` option, significantly improves on Motif and sets a new state

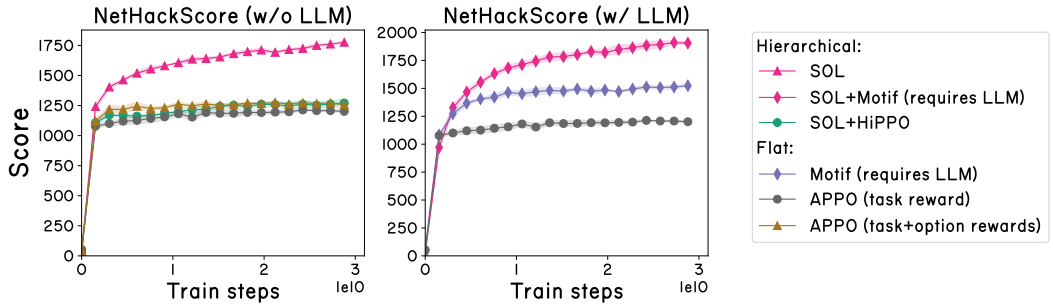


Figure 4: Results on `NetHackScore` environment with the Monk character. Shaded regions represent two standard errors computed over 5 seeds.

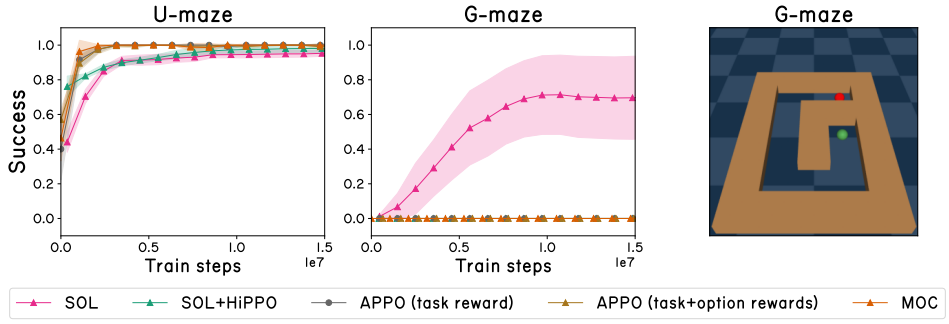


Figure 5: Results on two PointMaze layouts. Shading represents two standard errors over 10 seeds.

of the art on NetHackScore. In Appendix F.1, we repeat these experiments with two other NetHack characters different from the default Monk, and find that these trends are maintained. It is notable that `SOL`'s performance still appears to be increasing after 30 billion steps (~2 weeks of training), suggesting the that benefits of scale unlocked by our method remain to be fully realized.

In Appendix F.2, we include visualizations which shed light on `SOL`'s behavior. In particular, we find that: i) options are able to effectively optimize their respective rewards, and yield qualitatively different yet complementary behaviors, ii) the controller is able to effectively coordinate the options and call them in a state-dependent manner, and iii) the controller is able to adapt the option execution length based on the task and option. We also include ablations studying fixed vs. adaptive option lengths (Appendix F.3), option reward scaling (Appendix F.4), and the effect of adding redundant or unhelpful options (Appendix F.5).

5.2 Continuous Control

To test our algorithm's generality, we next consider continuous control mazes provided by Gymnasium Robotics (de Lazcano et al., 2024). We first found that flat APPO agents were able to solve all existing PointMaze environments (U-Maze, Medium and Large) when trained sufficiently, indicating that these mazes do not require hierarchy in the large sample regime (details in Appendix F.6). We therefore designed a more challenging maze called the G-maze, shown in Figure 5 (right). The agent (green dot) must navigate to the goal (red dot), and the reward is given by the change in euclidean distance between the two. The agent is initially close to the goal but separated by a wall, and the optimal trajectory requires an initial increase in distance followed by a larger decrease. This creates a local optimum in the reward landscape that is difficult to escape. We choose option rewards R_ω to be the velocity in the positive and negative x and y directions (which are already provided as part of the state), as well as the task reward, for a total of 5 options. Results for both the U-Maze and our G-maze are shown in Figure 5. On the U-Maze, all agents are able to achieve high success rates. However, on the G-maze, `SOL` is the only method able to make progress, achieving roughly 70% success while the others remain at zero. This provides evidence for `SOL`'s generality, conditioned on reasonable option rewards being available.

5.3 Automatically Learning Option Rewards

We next provide an experiment illustrating that SOL is compatible with methods for automatic reward synthesis and can be used to experiment with them at scale. We define a variant SOL+DIAYN where the option rewards $\{R_\omega\}_{\omega\in\Omega}$ are learned using DIAYN (Eysenbach et al., 2019), a method for automatic skill discovery. DIAYN operates by training a discriminator online to classify different policies based on the state, while the policies are trained using the discriminator's class probabilities as rewards. This has the effect of producing policies which visit distinct states, enabling them to be distinguished by the discriminator. In our setting, a discriminator $D : \mathcal{S} \to \Delta(\Omega)$ is trained online to classify the $|\Omega|$ different option policies. One option reward is set to be the task reward, and each remaining option ω has reward given by $R_\omega(s) = p_D(\omega|s)$. As before, the controller's reward is the task reward. The discriminator is trained and assigns option rewards fully on the GPU in the learner thread, causing a drop in throughput of only ~7%. Full experimental details can be found in Appendix C.2. Results on both MiniHack environments are shown in Figure 6. In both cases, SOL+DIAYN learns more slowly than SOL but converges to similar final performance, significantly outperforming the flat baseline while requiring no prior knowledge in the form of option rewards.

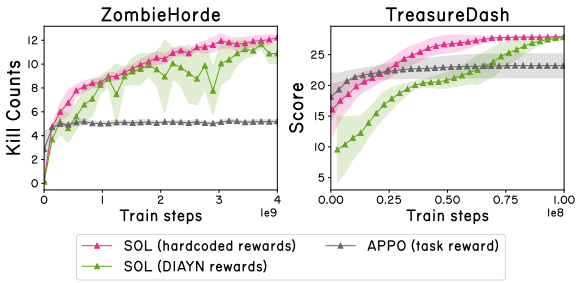


Figure 6: Results on MiniHack. Shaded regions represent two standard errors over 5 seeds.

5.4 Discussion

Taken together, our experimental results point to several takeaways. First, for the difficult credit assignment tasks we consider, flat agents struggle to escape suboptimal local minima, even when equipped with prior knowledge in the form of intrinsic rewards. Second, hierarchical structure alone is not sufficient either, as illustrated by the fact that SOL−HiPPO , which is trained with the task reward only, does not outperform flat agents. This is consistent with other works which have reported that hierarchical agents trained with the task reward alone have difficulty outperforming flat baselines (Bacon et al., 2017; Smith et al., 2018). Our best performing agent SOL combines both hierarchical structure with useful option rewards, which can be derived from prior knowledge or learned, that reflect the optimal policy in certain parts of the state space. This suggests that for certain classes of hard credit assignment problems, both hierarchy and good intrinsic rewards are necessary to unlock each others' benefits.

6 Conclusion

This work introduces, to our knowledge, the first online hierarchical RL algorithm which is able to scale to billions of samples. Its scalability is enabled by several systems-level design decisions which enable efficient GPU parallelization. When trained at scale on the challenging NetHack Learning Environment, our algorithm surpasses flat baselines and learns options with different behaviors which are effectively coordinated by the controller. It also proves effective in continuous control and MiniHack environments, showcasing its generality. We discuss potential ways to improve our algorithm along with current limitations in Appendix A. We hope that by releasing our code, we can facilitate future work in bringing the benefits of scale to hierarchical RL, and enable progress in long-horizon decision-making more broadly.

References

Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, pp. 1726–1734. AAAI Press, 2017.

Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Zhaohan Daniel Guo, and Charles Blundell. Agent57: Outperforming the Atari human benchmark. In Hal Daumé III and Aarti Singh (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 507–517. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/badia20a.html.

S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31 –39, 2011. ISSN 1521-9615. doi: 10.1109/MCSE.2010.118.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *CoRR*, abs/2005.14165, 2020. URL https://arxiv.org/abs/2005.14165.

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. In *International Conference on Learning Representations*, 2019.

Yuanpei Chen, Chen Wang, Li Fei-Fei, and C Karen Liu. Sequential dexterity: Chaining dexterous policies for long-horizon manipulation. *arXiv preprint arXiv:2309.00987*, 2023.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014. URL http://arxiv.org/abs/1406.1078.

NetHackWiki contributors. Nethack standard strategy. URL https://nethackwiki.com/wiki/Standard_strategy.

Özgür Şimşek and Andrew G. Barto. Skill characterization based on betweenness. In *Proceedings of the 22nd International Conference on Neural Information Processing Systems*, NIPS'08, pp. 1497–1504, Red Hook, NY, USA, 2008. Curran Associates Inc. ISBN 9781605609492.

Peter Dayan and Geoffrey E. Hinton. Feudal reinforcement learning. In *Proceedings of the 6th International Conference on Neural Information Processing Systems*, NIPS'92, pp. 271–278, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. ISBN 1558602747.

Rodrigo de Lazcano, Kallinteris Andreas, Jun Jet Tai, Seungjae Ryan Lee, and Jordan Terry. Gymnasium robotics, 2024. URL http://github.com/Farama-Foundation/Gymnasium-Robotics.

Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. https://github.com/openai/baselines, 2017.

Abhimanyu Dubey et al. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, Shane Legg, and Koray Kavukcuoglu. IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 1407–1416. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/espeholt18a.html.

Benjamin Eysenbach, Abhishek Gupta, Julian Ibarz, and Sergey Levine. Diversity is all you need: Learning skills without a reward function. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=SJx63jRqFm.

Linxi Fan, Guanzhi Wang, Yunfan Jiang, Ajay Mandlekar, Yuncong Yang, Haoyi Zhu, Andrew Tang, De-An Huang, Yuke Zhu, and Anima Anandkumar. Minedojo: Building open-ended embodied agents with internet-scale knowledge. In *Thirty-sixth Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2022. URL https://openreview.net/forum?id=rc8o_j8I8PX.

Nico Gürtler, Dieter Büchler, and Georg Martius. Hierarchical reinforcement learning with timed subgoals. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, NIPS '21, Red Hook, NY, USA, 2021. Curran Associates Inc. ISBN 9781713845393.

Nicolas Manfred Otto Heess, Greg Wayne, Yuval Tassa, Timothy P. Lillicrap, Martin A. Riedmiller, and David Silver. Learning and transfer of modulated locomotor controllers. *ArXiv*, abs/1610.05182, 2016. URL https://api.semanticscholar.org/CorpusID:9692454.

Mikael Henaff, Roberta Raileanu, Minqi Jiang, and Tim Rocktäschel. Exploration via elliptical episodic bonuses. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022.

Leslie Pack Kaelbling. Hierarchical learning in stochastic domains: preliminary results. In *Proceedings of the Tenth International Conference on International Conference on Machine Learning*, ICML'93, pp. 167–173, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc. ISBN 1558603077.

Anssi Kanervisto and Karolis Jucys. Nethack learning environment sample factory baseline. https://github.com/Miffyli/nle-sample-factory-baseline, 2022. Accessed: 2025-03-28.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *CoRR*, abs/2001.08361, 2020. URL https://arxiv.org/abs/2001.08361.

Alexander Kirillov, Eric Mintun, Nikhila Ravi, Hanzi Mao, Chloe Rolland, Laura Gustafson, Tete Xiao, Spencer Whitehead, Alexander C. Berg, Wan-Yen Lo, Piotr Dollár, and Ross Girshick. Segment anything. *arXiv:2304.02643*, 2023.

Martin Klissarov and Doina Precup. Flexible option learning. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems*, 2021. URL https://openreview.net/forum?id=L5vbEVIePyb.

Martin Klissarov, Pierre-Luc Bacon, Jean Harb, and Doina Precup. Learnings options end-to-end for continuous action tasks. *ArXiv*, abs/1712.00004, 2017. URL https://api.semanticscholar.org/CorpusID:1809550.

Martin Klissarov, Pierluca D'Oro, Shagun Sodhani, Roberta Raileanu, Pierre-Luc Bacon, Pascal Vincent, Amy Zhang, and Mikael Henaff. Motif: Intrinsic motivation from artificial intelligence feedback. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=tmBKIecDE9.

Martin Klissarov, Mikael Henaff, Roberta Raileanu, Shagun Sodhani, Pascal Vincent, Amy Zhang, Pierre-Luc Bacon, Doina Precup, Marlos C. Machado, and Pierluca D'Oro. Maestromotif: Skill design from artificial intelligence feedback. 2025. URL https://openreview.net/forum?id=or8mMhmyRV.

Vijay Konda and John Tsitsiklis. Actor-critic algorithms. In S. Solla, T. Leen, and K. Müller (eds.), *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999. URL https://proceedings.neurips.cc/paper_files/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

Heinrich Küttler, Nantas Nardelli, Thibaut Lavril, Marco Selvatici, Viswanath Sivakumar, Tim Rocktäschel, and Edward Grefenstette. TorchBeast: A PyTorch Platform for Distributed RL. *arXiv preprint arXiv:1910.03552*, 2019. URL https://github.com/facebookresearch/torchbeast.

Heinrich Küttler, Nantas Nardelli, Alexander H. Miller, Roberta Raileanu, Marco Selvatici, Edward Grefenstette, and Tim Rocktäschel. The NetHack Learning Environment. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

Minae Kwon, Sang Michael Xie, Kalesha Bullard, and Dorsa Sadigh. Reward design with language models. In *The Eleventh International Conference on Learning Representations*, 2023a. URL https://openreview.net/forum?id=10uNUgI5Kl.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023b.

Matthew Le, Apoorv Vyas, Bowen Shi, Brian Karrer, Leda Sari, Rashel Moritz, Mary Williamson, Vimal Manohar, Yossi Adi, Jay Mahadeokar, and Wei-Ning Hsu. Voicebox: Text-guided multilingual universal speech generation at scale. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *NeurIPS*, 2023. URL http://dblp.uni-trier.de/db/conf/nips/neurips2023.html#LeVSKSMWMAMH23.

Andrew Levy, Robert Platt Jr., and Kate Saenko. Hierarchical actor-critic. *CoRR*, abs/1712.00948, 2017. URL http://arxiv.org/abs/1712.00948.

Alexander Li, Carlos Florensa, Ignasi Clavera, and Pieter Abbeel. Sub-policy adaptation for hierarchical reinforcement learning. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=ByeWogStDS.

Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2018. URL https://arxiv.org/pdf/1712.09381.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv: Arxiv-2310.12931*, 2023.

Michael T. Matthews, Michael Beukman, Benjamin Ellis, Mikayel Samvelyan, Matthew Thomas Jackson, Samuel Coward, and Jakob Nicolaus Foerster. Craftax: A lightning-fast benchmark for open-ended reinforcement learning. In *ICML*, 2024. URL https://openreview.net/forum?id=hg4wXlrQCV.

Amy McGovern and Andrew G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse density. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, pp. 361–368, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1558607781.

Vegard Mella, Eric Hambro, Danielle Rothermel, and Heinrich Küttler. moolib: A Platform for Distributed RL. 2022. URL https://github.com/facebookresearch/moolib.

Ishai Menache, Shie Mannor, and Nahum Shimkin. Q-cut - dynamic discovery of sub-goals in reinforcement learning. In *Proceedings of the 13th European Conference on Machine Learning*, ECML '02, pp. 295–306, Berlin, Heidelberg, 2002. Springer-Verlag. ISBN 3540440364.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783, 2016. URL http://arxiv.org/abs/1602.01783.

Ofir Nachum, Shixiang Gu, Honglak Lee, and Sergey Levine. Data-efficient hierarchical reinforcement learning. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pp. 3307–3317, Red Hook, NY, USA, 2018. Curran Associates Inc.

OpenAI. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

Davide Paglieri, Bartłomiej Cupiał, Samuel Coward, Ulyana Piterbarg, Maciej Wolczyk, Akbir Khan, Eduardo Pignatelli, Łukasz Kuciński, Lerrel Pinto, Rob Fergus, Jakob Nicolaus Foerster, Jack Parker-Holder, and Tim Rocktäschel. BALROG: Benchmarking agentic LLM and VLM reasoning on games. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=fp6t3F669F.

Seohong Park, Kevin Frans, Deepinder Mann, Benjamin Eysenbach, Aviral Kumar, and Sergey Levine. Horizon reduction makes rl scalable, 2025. URL https://arxiv.org/abs/2506.04168.

Xue Bin Peng, Glen Berseth, Kangkang Yin, and Michiel Van De Panne. Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Trans. Graph.*, 36(4): 41:1–41:13, July 2017. ISSN 0730-0301. doi: 10.1145/3072959.3073602. URL http://doi.acm.org/10.1145/3072959.3073602.

Karl Pertsch, Youngwoon Lee, and Joseph J. Lim. Accelerating reinforcement learning with learned skill priors. In *Conference on Robot Learning (CoRL)*, 2020.

Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav S. Sukhatme, and Vladlen Koltun. Sample factory: Egocentric 3d control from pixels at 100000 FPS with asynchronous reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pp. 7652–7662. PMLR, 2020. URL http://proceedings.mlr.press/v119/petrenko20a.html.

Doina Precup and Richard S. Sutton. *Temporal abstraction in reinforcement learning*. PhD thesis, 2000. AAI9978540.

Haozhi Qi, Brent Yi, Mike Lambeta, Yi Ma, Roberto Calandra, and Jitendra Malik. From simple to complex skills: The case of in-hand object reorientation, 2025. URL https://arxiv.org/abs/2501.05439.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. *CoRR*, abs/2103.00020, 2021. URL https://arxiv.org/abs/2103.00020.

Nikhila Ravi, Valentin Gabeur, Yuan-Ting Hu, Ronghang Hu, Chaitanya Ryali, Tengyu Ma, Haitham Khedr, Roman Rädle, Chloe Rolland, Laura Gustafson, Eric Mintun, Junting Pan, Kalyan Vasudev Alwala, Nicolas Carion, Chao-Yuan Wu, Ross Girshick, Piotr Dollar, and Christoph Feichtenhofer. SAM 2: Segment anything in images and videos. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=Ha6RTeWMd0.

Mikayel Samvelyan, Robert Kirk, Vitaly Kurin, Jack Parker-Holder, Minqi Jiang, Eric Hambro, Fabio Petroni, Heinrich Kuttler, Edward Grefenstette, and Tim Rocktäschel. Minihack the planet: A sandbox for open-ended reinforcement learning research. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021. URL https://openreview.net/forum?id=skFwlyefkWJ.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017. URL http://arxiv.org/abs/1712.01815.

Satinder P. Singh. Scaling reinforcement learning algorithms by learning variable temporal resolution models. In Derek H. Sleeman and Peter Edwards (eds.), *Proceedings of the Ninth International Workshop on Machine Learning (ML 1992), Aberdeen, Scotland, UK, July 1-3, 1992*, pp. 406–415. Morgan Kaufmann, 1992a. doi: 10.1016/B978-1-55860-247-2.50058-9. URL https://doi.org/10.1016/b978-1-55860-247-2.50058-9.

Satinder P. Singh. Reinforcement learning with a hierarchy of abstract models. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI'92, pp. 202–207. AAAI Press, 1992b. ISBN 0262510634.

Matthew Smith, Herke van Hoof, and Joelle Pineau. An inference-based policy gradient method for learning options. In Jennifer Dy and Andreas Krause (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4703–4712. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/smith18a.html.

Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation*, pp. 212–223, Berlin, Heidelberg, 2002. Springer-Verlag. ISBN 3540439412.

Joseph Suarez. Pufferlib: Making reinforcement learning libraries and environments play nice, 2024. URL https://arxiv.org/abs/2406.12905.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artif. Intell.*, 112(1-2):181–211, 1999. URL http://dblp.uni-trier.de/db/journals/ai/ai112.html#SuttonPS99.

Andrew Szot, Alex Clegg, Eric Undersander, Erik Wijmans, Yili Zhao, John Turner, Noah Maestre, Mustafa Mukadam, Devendra Chaplot, Oleksandr Maksymets, Aaron Gokaslan, Vladimir Vondrus, Sameer Dharur, Franziska Meier, Wojciech Galuba, Angel Chang, Zsolt Kira, Vladlen Koltun, Jitendra Malik, Manolis Savva, and Dhruv Batra. Habitat 2.0: training home assistants to rearrange their habitat. In *Proceedings of the 35th International Conference on Neural Information Processing Systems*, NIPS '21, Red Hook, NY, USA, 2021. Curran Associates Inc. ISBN 9781713845393.

Gemini Team. Gemini: A family of highly capable multimodal models, 2024. URL https://arxiv.org/abs/2312.11805.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023. URL https://arxiv.org/abs/2307.09288.

Alexander Sasha Vezhnevets, Simon Osindero, Tom Schaul, Nicolas Heess, Max Jaderberg, David Silver, and Koray Kavukcuoglu. Feudal networks for hierarchical reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pp. 3540–3549. JMLR.org, 2017.

Marco Wiering and Jürgen Schmidhuber. Hq-learning. *Adaptive Behavior*, 6(2):219–246, 1997. ISSN 1059-7123.

Erik Wijmans, Abhishek Kadian, Ari Morcos, Stefan Lee, Irfan Essa, Devi Parikh, Manolis Savva, and Dhruv Batra. Decentralized distributed PPO: solving pointgoal navigation. *CoRR*, abs/1911.00357, 2019. URL http://arxiv.org/abs/1911.00357.

Ronald Williams and Jing Peng. Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268, 1991. doi: 10.1080/09540099108946587.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3–4):229–256, May 1992. ISSN 0885-6125. doi: 10.1007/BF00992696. URL https://doi.org/10.1007/BF00992696.

Zhanghao Wu, Eric Liang, Michael Luo, Sven Mika, Joseph E. Gonzalez, and Ion Stoica. RLlib flow: Distributed reinforcement learning is a dataflow problem. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2021. URL https://proceedings.neurips.cc/paper/2021/file/2bce32ed409f5ebcee2a7b417ad9beed-Paper.pdf.

Naoki Yokoyama, Alexander William Clegg, Joanne Truong, Eric Undersander, Jimmy Yang, Sergio Arnaud, Sehoon Ha, Dhruv Batra, and Akshara Rai. ASC: Adaptive Skill Coordination for Robotic Mobile Manipulation. *IEEE Robotics and Automation Letters*, 2023.

Qinqing Zheng, Mikael Henaff, Amy Zhang, Aditya Grover, and Brandon Amos. Online intrinsic rewards for decision making agents from large language model feedback, 2024. URL https://arxiv.org/abs/2410.23022.

A Limitations and Future Work

First, while we provide evidence that SOL is compatible with both hardcoded and learned option rewards, the question of how best to learn option rewards remains an open area of research, and more work is needed to find truly general methods which are successful across a wide range of environments. There are several potential ways to generate option rewards automatically, which constitute interesting future work. These include: using diversity measures (Eysenbach et al., 2019) to define rewards which induce a diverse set of option policies (we we explore in this work), novelty bonuses (Burda et al., 2019; Henaff et al., 2022) which could encourage exploratory options, distances to goals output by the controller (Nachum et al., 2018; Vezhnevets et al., 2017), or using LLMs synthesize rewards via code generation or preference ranking (Klissarov et al., 2024; 2025; Ma et al., 2023; Kwon et al., 2023b;a; Fan et al., 2022).

Second, our system is designed for computational efficiency, not sample efficiency. It focuses on achieving superior asymptotic performance in the large sample regime, rather than making optimal use of limited samples. Therefore, it is currently limited to settings where samples are easy to gather and compute is the bottleneck, such as video games, digital agents or sim-to-real transfer. Some of the design decisions, such as using a single neural network to represent both high and low-level policies, and the parallelized return computations, could in principle be incorporated into a model-based RL framework, which could potentially improve sample efficiency.

B Broader Impacts

This paper works on a foundational topic in RL, namely long-horizon decision-making. RL methods can eventually lead to positive applications (home assistants, digital assisstants, robotic surgery, medical and scientific discovery, autonomous driving, more efficient resource allocation) or negative ones (autonomous weapons, cyberattacks). Our work is not tied to direct applications or deployments, hence we do not see particular impacts worth highlighting at this time.

C EXPERIMENT DETAILS

C.1 ARCHITECTURES

For all MiniHack and NetHack experiments, we used a neural network architecture which mostly follows the Chaotic Dwarven GPT5 architecture of (Kanervisto & Jucys, 2022) with one change: we replaced the pipeline which renders glyphs to pixel images and runs them through an image-based convnet with a direct glyph embedding layer followed by 2 convolutional layers. We found this reduced the memory footprint (allowing us to have a larger PPO batch size) while giving slightly better performance. The pipelines processing the messages and bottom-line statistics (blstats) were unchanged. Specifically, the blstats are processed by a two-layer MLP with 128 hidden units at each layer, and the message character values are divided by 255 and also processed by a 2-layer MLP. The embeddings for the glyph images, blstats and messages are then concatenated and passed to a recurrent GRU (Cho et al., 2014). For the hierarchical models, we embed the policy index to a 128-dimensional vector which is concatenated with the other embeddings before passing to the GRU. This same vector is also replicated and added to all spatial locations in the glyph image crop. We also include an extra linear layer mapping the last hidden layer to controller actions.

For Mujoco experiments, we used a 2-layer network with 64 hidden units at each layer and tanh activations. The network outputs the mean and variance of a Gaussian distribution over actions, whose dimension is that of the action space. For hierarchical agents, the policy one-hot is concatenated with the input. As before, we add an extra linear layer mapping the last hidden layer to controller actions. The observation includes the agent's (x, y) position as well as the desired goal position. We do not use a GRU for Mujoco experiments since the environment is fully observed.

C.2 HYPERPARAMETERS

For all NLE agents, we used the common PPO hyperparameters which are listed in Table 1. Our `SOL` agents additionally use the hyperparameters listed in Table 5.

Table 1: Common PPO Hyperparameters for different environments. The same set of hyperparameters are used for MiniHack and NetHack, a different set is used for Mujoco PointMaze.

Hyperparameter	MiniHack&NetHack	Mujoco PointMaze
Rollout length	1024	256
GRU recurrence	256	none
GRU layers	1	none
PPO epochs	1	10
PPO clip ratio	0.1	0.2
PPO clip value	1.0	1.0
Encoder crop dimension	12	N/A
Encoder embedding dimension	128	N/A
Reward Scaling	0.01	10
Exploration loss coefficient	0.003	0.001
Exploration loss	entropy	entropy
Value loss coefficient	0.5	0.5
Max gradient norm	4.0	0.1
Learning Rate	0.0001	0.003
Batch size	32768	32768
Worker number of splits for double-buffering	2	2
V-trace ρ	1.0	1.0
V-trace c	1.0	1.0
Discount factor γ	0.99	0.99

Table 2: `APPO (task/task+option rewards)` Hyperparameters

Environment	Hyperparameter	Value	Values swept
MiniHack-ZombieHorde	Score reward scale	1	1
	Health reward scale	1	0, 1, 3, 10, 20
MiniHack-TreasureDash	Stairs option reward scaling	1	1
	Gold option reward scaling	0.1	0, 0.1, 0.3, 1, 3, 10
NetHackScore	Score option reward scaling	1	1
	Health option reward scaling	1	0, 1, 3, 10
PointMaze-GMaze	True goal reward scaling	1	1
	Goal option reward scaling	1	0, 0.01, 0.1, 1, 10

Table 3: `SOL` Hyperparameters. The controller extra exploration loss scaling is a factor which is used to further scale the exploration loss coefficient from Table 1 applied to the controller outputs. We found that having this greater than 1 was sometimes helpful.

Environment	Hyperparameter	Value	Values swept
MiniHack-ZombieHorde	Controller extra exploration loss scaling	1	1, 3, 10
	Controller reward scaling	0.001	0.001, 0.01, 0.1
	Score option reward scaling	1	-
	Health option reward scaling	20	10, 20
MiniHack-TreasureDash	Controller extra exploration loss scaling	1	1, 3, 10
	Controller reward scaling	0.001	0.001, 0.01, 0.1
	Stairs option reward scaling	1	-
	Gold option reward scaling	1	-
NetHackScore	Controller extra exploration loss scaling	10	1, 3, 10
	Controller reward scaling	0.001	0.001, 0.01, 0.1
	Score option reward scaling	1	-
	Health option reward scaling	10	10, 20
PointMaze-GMaze	Controller extra exploration loss scaling	1	1, 3, 10, 30
	Controller reward scaling	1	0.01, 0.1, 1, 10
	Goal option reward scaling	1	-

Table 4: `SOL-HiPPO` Hyperparameters. We set the number of options to be the same as `SOL` and swept hyperparameters in the same ranges.

Environment	Hyperparameter	Value	Values swept
MiniHack-ZombieHorde	Controller extra exploration loss scaling	1	1, 3, 10, 30
	Controller reward scaling	0.001	0.001, 0.01, 0.1
	Number of options	2	-
MiniHack-TreasureDash	Controller extra exploration loss scaling	1	1, 3, 10, 30
	Controller reward scaling	0.001	0.001, 0.01, 0.1
	Number of options	2	-
NetHackScore	Controller extra exploration loss scaling	10	1, 3, 10, 30
	Controller reward scaling	0.001	0.001, 0.01, 0.1
	Number of options	2	-
PointMaze-GMaze	Controller extra exploration loss scaling	1	1, 3, 10, 30
	Controller reward scaling	1	0.01, 0.1, 1, 10
	Number of options	5	-

Table 5: SOL-DIAYN Hyperparameters. The discriminator shares the same architecture as the observation encoder, with a 2-layer MLP added with $|\Omega|$ outputs.

Environment	Hyperparameter	Value	Values swept
MiniHack-ZombieHorde	Controller extra exploration loss scaling	10	1, 3, 10
	Controller reward scaling	0.001	0.001, 0.01, 0.1
	DIAYN Reward Scaling	0.03	0.01, 0.03, 0.1, 0.3, 1.0
	Discriminator Learning Rate	0.0001	0.0001
	Number of options	3	2, 3, 4
MiniHack-TreasureDash	Controller extra exploration loss scaling	1	1, 3, 10
	Controller reward scaling	0.001	0.001, 0.01, 0.1
	DIAYN Reward Scaling	0.3	0.01, 0.03, 0.1, 0.3, 1.0
	Discriminator Learning Rate	0.0001	0.0001
	Number of options	3	2, 3, 4

Table 6: Motif Hyperparameters. We trained the reward model using the official source code, data and default hyperparameters. We then trained APPO agents with the same hyperparameters as other agents (Table 1) and tuned the coefficient of the reward model.

Hyperparameter	Value	Values swept
LLM reward coefficient	0.1 (default)	0.1, 0.3, 1

Table 7: MOC Hyperparameters. Despite our hyperparameter sweep, results did not change much: MOC worked well on PointMaze-UMaze, and failed to learn on PointMaze-GMaze. Therefore we report results with default hyperparameters.

Hyperparameter	Value	Values swept
Number of options	2 (default)	2, 4, 8
Learning rate	0.0008 (default)	0.001, 0.0003, 0.0001, 0.00008
Probability of updating all options η	0.9 (default)	0.1, 0.9

C.3 Throughput Comparison Details

All experiments were conducted on an NVIDIA V100-SXM2-32GB GPU. We used the same NLE encoder described in Appendix C.1 for HIRO and Option-Critic. We used the following implementations:

- HIRO: https://github.com/watakandai/hiro_pytorch
- Option-Critic: https://github.com/lweitkamp/option-critic-pytorch
- MOC: https://github.com/mklissa/MOC
- Hierarchical RLLib: https://docs.ray.io/en/latest/rllib/hierarchical-envs.html

Other than changing the architecture to process NLE observations, we kept the rest of the hyperparameters at their default values except for the following. We experimented with different batch sizes of off-policy updates for HIRO and Option-Critic, but this did not significantly change the throughput.

For MOC, we increased the number of parallel environments until the throughput saturated, which was 256 here. We used the NLE visual rendering pipeline from (Kanervisto & Jucys, 2022), where NLE glyphs are rendered to pixels and then processed by a standard Atari DQN convolutional encoder. The reason we did this was because the MOC codebase (based on OpenAI Baselines) only supported pixel and continuous vector inputs. We also ran SOL with the same visual rendering pipeline and found that its speed was around 10% faster than the symbolic encoder for the same batch size, hence we do not believe that using a visual rendering pipeline penalizes methods in these comparisons.

C.4 Compute Details

All experiments were run on single NVIDIA V100-SXM2-32GB GPU machines. For MiniHack experiments, we used 16 CPUs per experiment. Running a job took around 5 hours for TreasureDash and 10 hours for ZombieHorde. For NetHack experiments, we used 48 CPUs per experiment. Running a job for 30 billion steps took around 14 days. For Mujoco, we used 8 CPUs per experiment. Each job ran for less than one day.

C.5 Other code links

We use the public codebase for our Motif reward model: https://github.com/mklissa/maestromotif. Our codebase is built upon Sample Factory: https://github.com/alex-petrenko/sample-factory, which is licensed under an MIT license.

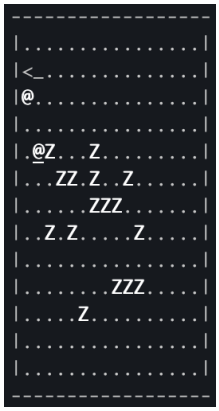
D Environment Details

D.1 MiniHack

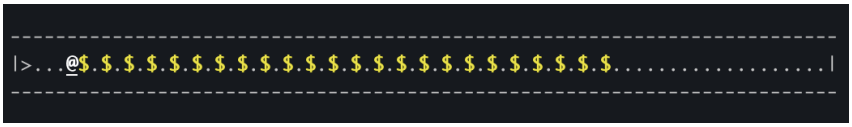
Here we describe the details of our MiniHack environments. Both have a simple action space consisting of 4 movement actions (north, south, east, west) and the `EAT` action. We note that movement also serves to attack: attempting to move on a square occupied by an enemy attacks it.

In ZombieHorde (Figure 7a), the agent @ must defeat all the zombies Z. Since they are too numerous to fight at once, the agent must periodically retreat to the altar _ which the zombies cannot get close to, in order to heal. The priest @ has no effect here. The time limit is 1500 steps, which enables long periods of healing. Agents in NetHack heal at a rate of about 1 hit point per 10 timesteps, hence full healing can require over 100 steps. Each zombie destroyed gives 20 score points.

In TreasureDash, the agent gets 20 points for exiting through the stairs >, which ends the episode. Each piece of gold $ gives 1 point. The episode time limit is 40 steps. If the agent goes right the whole time, it gathers 20 gold pieces for 20 total points. If it goes left only, it exits and also gets 20 points. The optimal strategy is to gather 8 gold pieces on the right, and then go left all the way to the staircase. This requires stopping the gold-gathering behavior and switching to seeking the staircase.



(a) ZombieHorde.



(b) TreasureDash

Figure 7: MiniHack environments designed to present challenging credit assignment problems.

D.2 NetHack

The `NetHackScore` environment from the NLE paper includes the following actions: all movement actions, as well as `SEARCH` (needed for finding secret doors, which is often necessary to explore the full level and go to the next), `KICK` (needed for kicking down locked doors, also needed to explore the visit the full level) and `EAT` (needed for eating the comestibles the agent starts with). If the agent does not eat, it will starve before too long which limits the episode length and the maximum progress the agent can make. However, the `EAT` action alone is not enough to eat the comestibles in the agent's inventory, due to the NLE's context-dependent action space. After selecting the `EAT` action, the agent must also select which item in inventory to eat, which requires pressing a key corresponding to the item's inventory slot, which must be included in the original action space, which is often not the case. Therefore, we adopt the modification introduced in (Klissarov et al., 2024), where every time the `EAT` action is selected, the next action is chosen at random from the available inventory slots given in the message. This is also discussed in Appendix G of their paper.

D.3 POINTMAZE

The PointMaze environment uses Gymnasium Robotics and we simply pass the maze map as argument. The agent and goal location are fixed rather than resetting each episode. The reward is the change in euclidean distance between the agent and the goal, and the episode ends whenever the goal is reached according to the default PointMaze criterion.

E ALGORITHM DETAILS

E.1 OBJECTIVES

Here we give the exact definitions of some of the functions used in Section 3.1. Let $\mu : \mathcal{S} \to \Delta(\mathcal{A})$ denote the flattened hierarchical policy, i.e. the mapping from states to actions obtained by executing the options and controller using the call-and-return process. We define the state-option value, state value, and option-advantage functions of μ associated with the task reward as:

$$
\begin{aligned}
Q^{\mu}_{\text{task}}(s_t, \omega) &= \mathbb{E}_{\mu}[\sum_{k=0}^{\infty} \gamma^k R(s_{t+k}, a_{t+k}) | s_t = s, \omega_t = \omega] \\
V^{\mu}_{\text{task}}(s) &= \mathbb{E}_{\mu}[\sum_{k=0}^{\infty} \gamma^k R(s_{t+k}, a_{t+k}) | s_t = s] \\
A^{\text{task}}(s_t, \omega) &= Q^{\mu}_{\text{task}}(s_t, \omega) - V^{\mu}_{\text{task}}(s_t)
\end{aligned}
$$

The state-action value, state value, and advantage functions for an option ω are given by:

$$
\begin{aligned}
Q^{\omega}(s_t, a_t) &= \mathbb{E}_{\pi_\omega}[\sum_{k=0}^{\infty} \gamma^k R_\omega(s_{t+k}, a_{t+k}) | s_t = s, a_t = a] \\
V^{\omega}(s_t) &= \mathbb{E}_{\pi_\omega}[\sum_{k=0}^{\infty} \gamma^k R_\omega(s_{t+k}, a_{t+k}) | s_t = s] \\
A^{\omega}(s_t, a_t) &= Q^{\omega}(s_t, a_t) - V^{\omega}(s_t)
\end{aligned}
$$

In Section 3.1 we mentioned that calling the controller does not cause the MDP to transition, which means that states in τ are duplicated each controller call. To illustrate this, let us consider the first time step, with s_0 being the first state. First the controller must be called, since we don't know what low-level option to execute. We therefore run s_0 through the controller and obtain $\omega_3, l \sim \pi_\Omega(\cdot|s_0)$. This means we will execute option policy π_{ω_3} for l timesteps. The first state we must apply it to is still s_0, since we haven't passed any actions to the MDP yet. We therefore compute $a_0 \sim \pi_{\omega_3}(\cdot|s_0)$, sample $s_1 \sim p(\cdot|s_0, a_0)$, and repeat this process for l timesteps. We then call the controller again at s_l, which produces (for example) $\omega_2, l' \sim \pi_\Omega(\cdot|s_l)$, meaning we will execute π_{ω_2} for l' timesteps. Again, since we have not executed any actions in the environment, we then run s_l through π_{ω_2} and the process continues. See Table 8 for an example trace.

E.2 NEURAL NETWORK ARCHITECTURE

SOL's single neural network architecture is shown in Figure 8. A one-hot vector u of dimension $|\Omega| + 1$ indicates which of the policies (among the option policies and the controller policy) to represent. In the actor workers, if u marks the controller, the softmaxes over options and option execution lengths are sampled from and the results are used to update the environment wrapper shown in Appendix E.3. Otherwise, if u marks one of the options, the softmax over environment actions $|\mathcal{A}|$ is sampled from and the sampled action is executed in the environment. In the learner worker, the softmaxes over options and option lengths constitute the action probabilities of the controller and are used to compute the advantage and policy loss at each step it is called. The softmax over environment actions gives the action probabilities of the option policies and is used to compute the advantage and policy loss at any steps that they are called. The scalar value output represents the value estimate of the policy currently marked by the policy indicator u, and is used in the learner worker to compute the value loss and advantages of both the controller and option policies. A key advantage of our design is that trajectories can be processed in batch, regardless of which policies are being executed, since they are only differentiated by the policy indicators u which are also fed in as a batch.

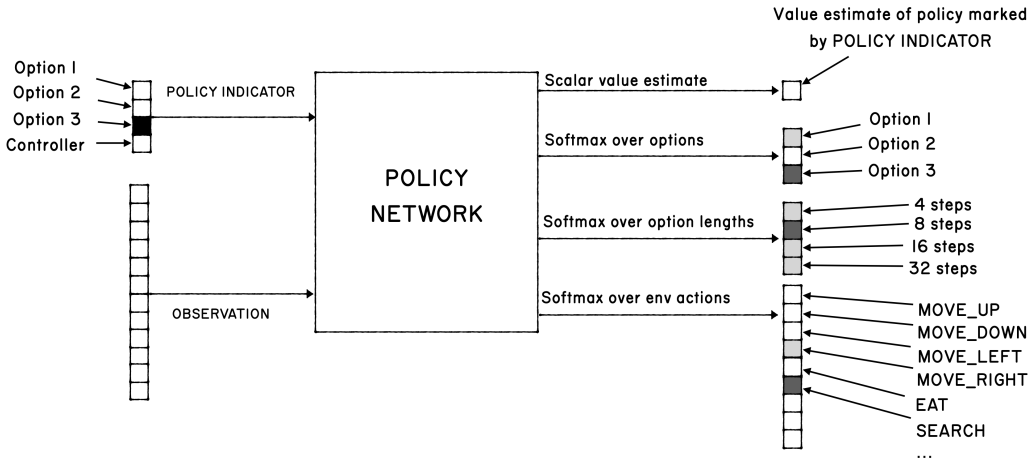


Figure 8: SOL's single neural network architecture. The environment actions (bottom right) represent those from NetHack/MiniHack here, but can also be continuous (e.g. in MuJoCo experiments). The set of option lengths in our experiments is $\{1, 2, 4, 8, 16, 32, 64, 128\}$.

E.3 Environment wrapper pseudocode

Pseudocode for the environment wrapper in the actor workers is shown below.

```
class HierarchicalWrapper(gym.Wrapper):

def __init__(self, env, ...):
    self.env = env
    self.option_policies = [...]
    self.option_length = ...


def compute_option_reward(self, option):
    ...

def reset(self):
    obs = self.env.reset()
    self.current_policy = "controller"
    obs["current_policy"] = self.current_policy
    return obs

def step(self, action):

    env_action, option_indx, option_length = action

    if self.current_policy == "controller":
        self.current_policy = self.option_policies[option_indx]
        obs["current_policy"] = self.current_policy
        self.option_length = option_length
        done = False
        # the controller reward depends on the future, so we compute
        # it in the learner thread and flag for now.
        reward = 42
        self.option_steps = 0
        info = {}
        return obs, reward, done, info
    else:
        obs, done, task_reward, info = self.env.step(env_action)
        reward = self.compute_option_reward(obs, self.current_policy)
        self.option_steps += 1

        if self.option_steps == self.option_length:
            self.current_policy = "controller"
        obs["current_policy"] = self.current_policy

        return obs, done, reward, info
```

The wrapper produces trajectories of the form shown below in Table 8. Note that observations are duplicated each time the option changes: they are used first as input to the controller, which chooses the option to execute next, and then the same observation is used as input to the chosen option. The Action row contains both actions of the controller policy, which are options $\omega \in \Omega$, and low-level environment actions of the option policies, which are in the MDP's action space $\mathcal{A}$. The State, Action, Policy Index and Policy Rewards correspond to the s_t, a_t, z_t and r_t variables of τ in Section 3.1. The Policy Rewards corresponding to controller calls (marked in red) are the sum of the task rewards over the course of the next policy call—these are computed in the learner thread, since they depend on the future not known to the actor thread at the current time step.

State	s_1	s_1	s_2	s_3	s_4	s_5	s_5	s_6	s_7	s_8
Action	ω_1	a_1^{env}	a_2^{env}	a_3^{env}	a_4^{env}	ω_3	a_5^{env}	a_6^{env}	a_7^{env}	a_8^{env}
Task Reward	-	r_1	r_2	r_3	r_4	-	r_5	r_6	r_7	r_8
Policy Reward	$\sum_{t=1}^{4} r_t$	r_1^1	r_2^1	r_3^1	r_4^1	$\sum_{t=5}^{8} r_t$	r_5^3	r_6^3	r_7^3	r_8^3
Policy Index	Ω	ω_1	ω_1	ω_1	ω_1	Ω	ω_3	ω_3	ω_3	ω_3
Termination	0	0	0	0	0	0	0	1	0	0

Table 8: Example trajectory produced by the environment wrapper. The quantities in red are computed later in the learner thread (see above), and are included for completeness.

E.4 PARALLELIZED V-TRACE

```
    """
    This function computes advantages and value targets for all policies
    in the batch simultaneously. The arguments are:


    ratios: ratio of action probs between current and old policy
    values: bootstrapped value predictions
    dones: episode terminals
    rewards: rewards of mixed type, see Policy Reward in Table 7.
    rho_hat: V-trace truncation parameter
    c_hat: V-trace truncation parameter
    num_trajectories: number of trajectories in the batch
    recurrence: number of timesteps in the batch
    gamma: discounting factor
    policy_indx: the Policy Index in Table 7, also z_t in Section 3.1
    num_policies: total number of policies (options and controller, i.e.
    |\Omega| + 1).
    """

    def _compute_vtrace_sol(
            ratios,
            values,
            dones,
            rewards,
            rho_hat,
            c_hat,
            num_trajectories,
            recurrence,
            gamma,
            policy_indx,
            num_policies,
    ):
        vtrace_rho = torch.min(rho_hat, ratios)
        vtrace_c = torch.min(c_hat, ratios)

        # tensors to store the advantages and value predictions
        adv = torch.zeros((num_trajectories * recurrence,))
        vs = torch.zeros((num_trajectories * recurrence,))


        next_values = torch.zeros(num_trajectories, num_policies)
        next_vs = torch.zeros(num_trajectories, num_policies)
        delta_s = torch.zeros(num_trajectories, num_policies)

        # V-trace returns are computed using a base case followed
        # by recurrence relation. This marks which policies the
        # base case  is handled for.
        is_base_case_handled = torch.zeros(
            num_trajectories, num_policies, dtype=torch.bool
        )

        # When an episode ends, we need to zero out the returns for
        # each policy using the last timestep it is executed for
        # before the episode ends.
        is_episode_done = torch.zeros(
            num_trajectories, num_policies, dtype=torch.bool
        )

        for i in reversed(range(recurrence)):
            current_policies_one_hot = F.one_hot(
                policy_indx[i::recurrence], num_classes = num_policies
            ).bool()
```

```
            rewards = rewards[i::recurrence]
            curr_dones = dones[i::recurrence].bool()

            # when we encounter a "done", mark all policies as done.
            # we will unmark the ones at the current timestep for
            # which we mask out the returns.
            is_episode_done = is_episode_done | curr_dones.view(-1, 1)

            dones = is_episode_done[current_policies_one_hot].to(dtype)
            not_done = 1.0 - dones
            not_done_times_gamma = not_done * gamma

            curr_values = values[i::recurrence]
            curr_vtrace_rho = vtrace_rho[i::recurrence]
            curr_vtrace_c = vtrace_c[i::recurrence]

            # we have accounted for the latest episode termination
            # of the current policies in 'not_done_times_gamma',
            # so reset this until the next 'done' is encountered.
            is_episode_done[current_policies_one_hot] = False


            if i < recurrence - 3:
                controller_indx = num_policies - 1
                trajs_with_changed_options = (
                    (policy_indx[(i+1)::recurrence] == controller_indx) &
                    (policy_indx[i::recurrence] != policy_indx[(i+2)::
recurrence])
                )
                # for any trajectories where the option switched,
                # reset the base case so that bootstrapped returns
                # are applied
                is_base_case_handled[current_policies_one_hot] = \
                is_base_case_handled[current_policies_one_hot] & \
                ~trajs_with_changed_options


            base_case_indices = (~is_base_case_handled) &
current_policies_one_hot
            base_case_indices_any = torch.any(base_case_indices, dim = 1)

            next_values[base_case_indices] = (
                values[i :: recurrence][base_case_indices_any]
                - rewards[i :: recurrence][base_case_indices_any]
            ) / gamma

           next_vs[base_case_indices] = next_values[base_case_indices]

            is_base_case_handled = is_base_case_handled |
base_case_indices

            if not is_base_case_handled.any().item():
                continue

            delta_s[current_policies_one_hot] = curr_vtrace_rho * (
                rewards
                + not_done_times_gamma * next_values[
current_policies_one_hot]
                - curr_values
            )

            adv[i::recurrence] = curr_vtrace_rho * (
                rewards
```

```
                + not_done_times_gamma * next_vs[current_policies_one_hot
    ]
                - curr_values
            )

            next_vs[current_policies_one_hot] = (
                curr_values
                + delta_s[current_policies_one_hot]
                + not_done_times_gamma
                * curr_vtrace_c
                * (next_vs[current_policies_one_hot] -
                next_values[current_policies_one_hot])
            )
            vs[i::recurrence] = next_vs[current_policies_one_hot]
            next_values[current_policies_one_hot] = curr_values

    return adv, vs
```

F Additional Experiment Results

F.1 Additional NetHack Characters

Here we report results with additional NetHack characters. Most prior work (Klissarov et al., 2024; 2025; Zheng et al., 2024) uses the Monk character, however this is only one out of 13 characters in the game. Here we compare all methods on two other characters: the Ranger and Archaeologist. The trends we observed for the Monk are repeated here: `SOL` and `SOL+Motif` significantly outperform the other methods, and their performance continues to improve over the course of 30 billion training samples. This shows that our conclusions are not particular to the Monk character.

We also note that the scores for the Ranger and Archaeologist are significantly lower than the Monk, which is likely due to the fact that the Monk starts proficient in unarmed combat and can succeed in the early game without needing to learn how to equip weapons and armor.

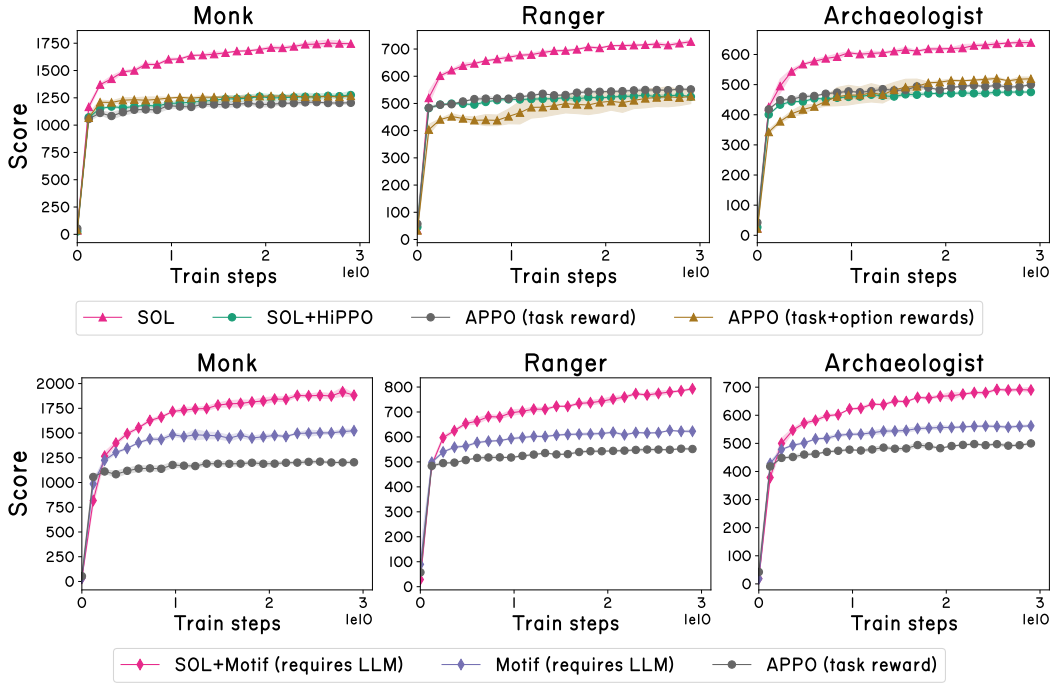


Figure 9: Results on `NetHackScore` for three different characters. Curves represent the mean and shaded regions represent two standard errors computed over 5 seeds.

F.2 Visualizations and Analysis

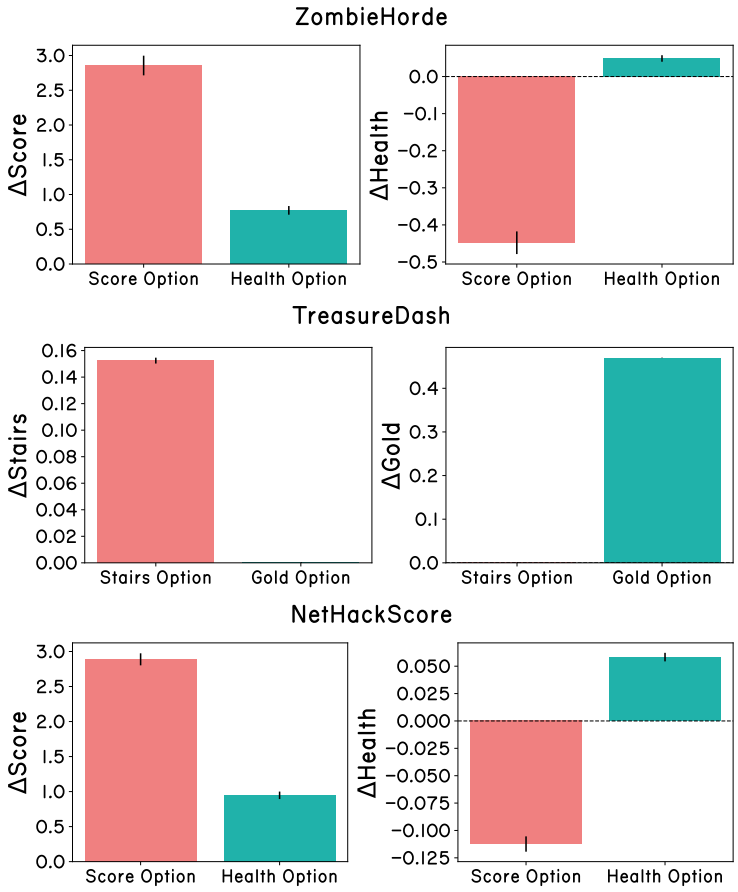


Figure 10: Option mean returns, normalized by option execution length. Error bars represent two standard errors computed over 500 episodes.

In this section we provide visualizations which help shed light on `SOL`'s behavior. In Figure 10, for each environment we report the average return in terms of each option reward when executing each option policy. On both ZombieHorde and NetHackScore, the `Score` option accumulates higher score than the `Health` option (as shown by its higher Δ`Score` return), but sustains damage over time (as shown by its negative Δ`Health` return). The `Health` option accumulates less score, but *recovers* health over time (as shown by its positive value in terms of Δ`Health`, enabling the agent to survive longer overall. For TreasureDash, the `Stairs` option achieves positive Δ`Stairs` reward (indicating it has descended a staircase) and no Δ`Gold` reward (indicating it has collected no gold), whereas the `Gold` reward is the opposite. Overall, this shows that `SOL` is able to learn different options which produce distinct behaviors.

We additionally measured overlap between option policies by computing the normalized mutual information (NMI) between their action distributions over the course of 100 evaluation episodes, shown in Table 9. The NMI between variables X and Y is defined as: $I(X,Y)/\min(H(X),H(Y))$ and can range between 0 (fully independent) and 1 (fully redundant). We see that the NMI is low for all three environments, providing further evidence that the sub-policies learn distinct behaviors. Interestingly, it is lowest for TreasureDash, which also has the most distinct options in terms of return, with no overlap between their respective rewards.

In Figure 11, for each environment we plot the distributions of option lengths selected by the controller for each option. On ZombieHorde, the `Score` option tends to be called for shorter lengths than `Health`. This may be explained by the fact that healing takes a long time, around 10 time steps per hit point: at experience level 1, healing from 7/14 hit points back to full health takes ~70

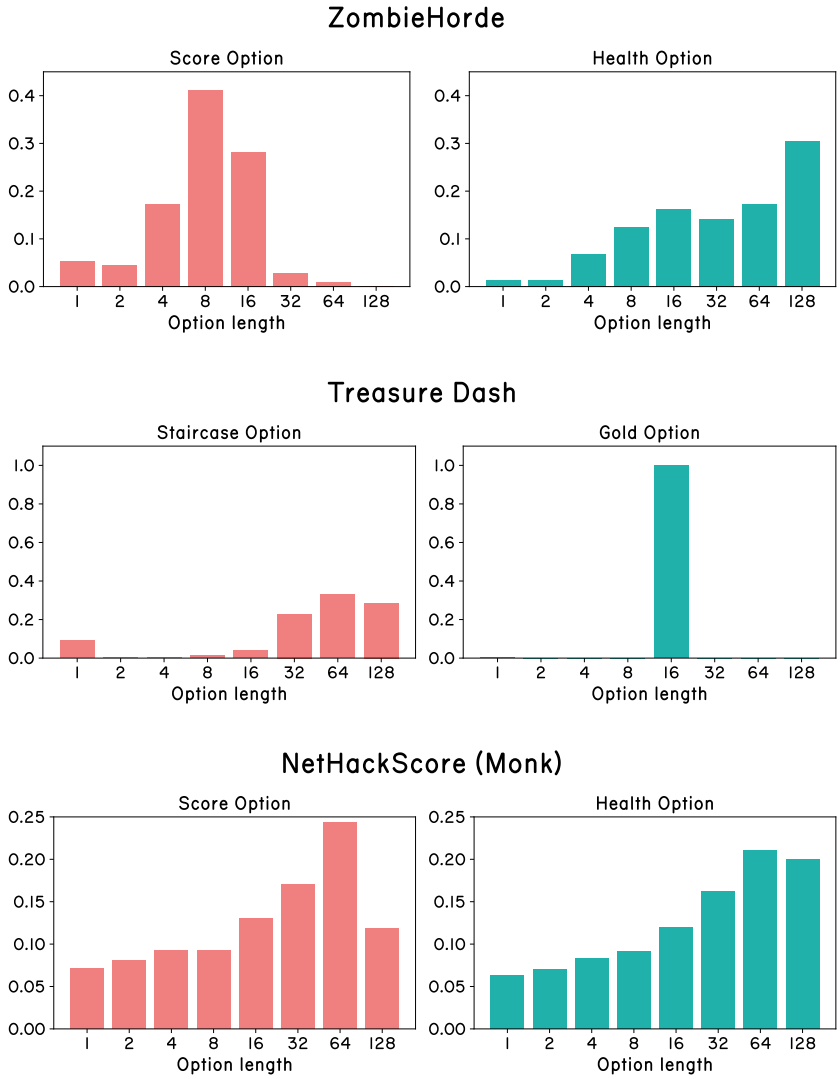


Figure 11: Distributions of option execution lengths chosen by the controller. The distribution is non-uniform, which indicates learning on the part of the controller. Longer option lengths are chosen more frequently than short ones. The Score option tends to be executed for the longest option length (128 steps) less often than Health. This may be because calling Score for longer than is optimal carries a higher risk than for Health: executing Score for too long may result in too much combat and agent death, whereas executing Health for too long will result in the agent wasting turns trying to heal at full health, which has fewer negative consequences. Distributions are computed over 500 test episodes.

Environment	Normalized Mutual Information
ZombieHorde	0.154
TreasureDash	0.018
NetHack	0.388

Table 9: Normalized Mutual Information (NMI) between action distributions of different option policies, computed over 100 episodes. The NMI is defined between 0 and 1.

time steps. Also, executing the `Score` option involves fairly high uncertainty due to the stochasticity of NetHack's combat system, where damage is dealt randomly based on various statistics: it may be that the agent gets lucky defeats several monsters in a row, or it may be unlucky and sustain high damage at the beginning, in which case it needs to switch back to the `Health` option. Choosing shorter option lengths for `Score` allows the agent to switch back to the `Health` option more quickly if needed. In contrast, healing is mostly deterministic and there is less downside to selecting the `Health` option for longer than needed. In TreasureDash, the controller very precisely chooses the optimal execution length of 16 for the `Gold` option (the optimal policy moves right for 16 steps to get 8 gold, then moves left for 24 steps to the stairs), and assigns similar lengths to any of the 3 optimal lengths for the `Staircase` option (32, 64, 128). For NetHackScore, the option lengths are more spread out, although `Score` is still skewed somewhat shorter than `Health`. We note that healing is shorter in NetHackScore, because the action space includes extended movement actions (such as `MOVEFAR`) than take several game turns, and executing one of these speeds up healing from the perspective of the agent—this may explain why the difference in option lengths is less pronounced than for ZombieHorde, even though the option rewards are the same for both environments.

In Figure 12, we plot the fraction of controller calls to each option conditioned on the agent's health and experience level. The controller calls the `Health` option more frequently when the agent's health as low, which makes sense since this enables the agent to recover its health and survive longer. Interestingly, the controller also tends to call the `Health` option more often at low experience levels (96% of the time at Experience Level 1). Upon visualizing trajectories, we found that the agent still fights monsters that attack it when executing the `Health` option, but does not seek them out. This results in the agent staying at the first few dungeon levels, fighting weaker monsters that appear, and gaining some experience levels. It then begins calling the `Score` option more frequently, resulting in it attacking monsters and exploring further into the dungeon. This is similar to successful human gameplay, which requires careful progression of dungeon levels only when the agent is strong enough (contributors).

Figure 12: Fraction of controller calls to the `Health` and `Score` options for Monk, conditioned on the agent's normalized health (current hit points divided by maximum hit points) and experience level. The controller calls the `Health` option more frequently at low health, enabling the agent to recover and survive longer, and at low experience levels, when the agent is still weak. Distributions are computed over 500 test episodes.

F.3 MiniHack Option Length Ablation

In Figure 13 we report the final results for both MiniHack environments when using different fixed option lengths in $\{2, 4, 8, 16, 32, 64\}$. In this setting, every time the controller selects an option it is always executed for the same fixed number of steps. Having fixed lengths which are either too long or too short lengths hurts performance. In contrast, our adaptive selection mechanism is able to automatically tune the option lengths, and performs comparably to the best fixed option length.

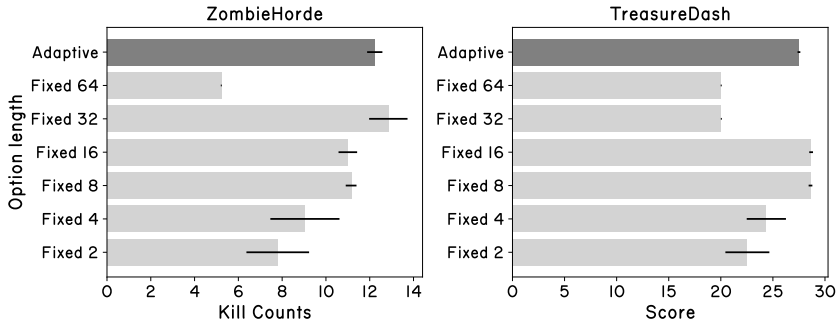


Figure 13: Final performance on ZombieHorde and TreasureDash for different fixed options lengths as well as the adaptive option lengths. Bars represent standard errors over 5 seeds.

F.4 MiniHack Option Reward Scaling Ablation

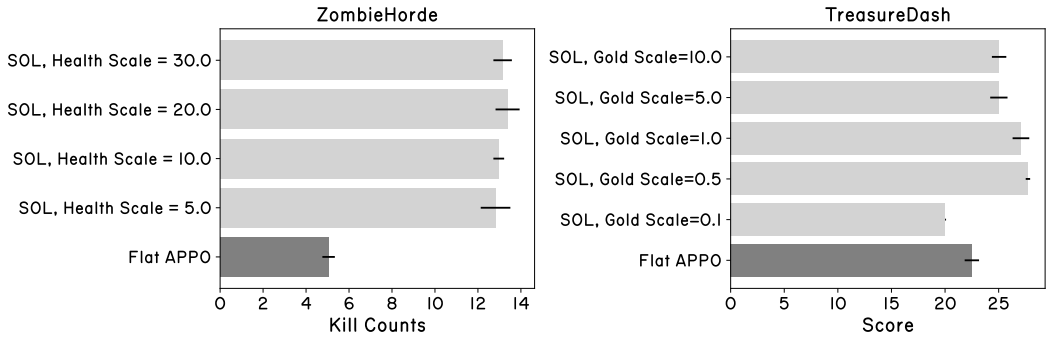


Figure 14: Final performance on ZombieHorde and TreasureDash for different scaling coefficients of the non-task-reward option. Bars represent standard errors over 5 seeds.

F.5 MiniHack Option Quality Ablation

In Figure 15 we study how the performance of `SOL` changes in the presence of redundant or useless options on both MiniHack tasks. We compare the following variants:

- `SOL` : our default version, which has two options that are both useful for the task ($\Omega = \{$`Score, Health`$\}$ for ZombieHorde, $\Omega = \{$`Stairs, Gold`$\}$ for TreasureDash).
- `SOL(+2 duplicate options)`: has both original options duplicated once each. Its option set is $\Omega = \{$`Score, Score2, Health, Health2`$\}$ for ZombieHorde and $\Omega = \{$`Stairs, Stairs2, Gold, Gold2`$\}$ for TreasureDash. Here `Score2` is an option with identical reward as `Score`, and same for the other options.
- `SOL(+8 duplicate options)`: has both original options duplicated 4 times each. Its option set is $\Omega = \{$`Score,...,Score5, Health,...,Health5`$\}$ for ZombieHorde and $\Omega = \{$`Stairs,...,Stairs5, Gold,...,Gold5`$\}$ for TreasureDash.
- `SOL(+2 useless options)`: has 2 options added which are unrelated to the task at hand. For ZombieHorde, the option set is $\Omega = \{$`Score, Health, Gold, Scout`$\}$ and for TreasureDash the option set is $\Omega = \{$`Stairs, Gold, Scout, Health`$\}$. Here `Scout` is a reward measuring exploration taken from (Küttler et al., 2020).

Results are shown in Figure 15. Adding duplicates of options that are useful for the task at hand does not significantly change performance. Adding useless options (which are unrelated to the task at hand) slows down learning on both tasks, which is unsurprising: without prior knowledge, the agent must learn through experience which options are useful and which are not (also recall that we have an entropy bonus on the controller which encourages it to sample all options with some probability). However, on both tasks the agent with useless options is able to eventually match the performance of the others, given sufficient training.

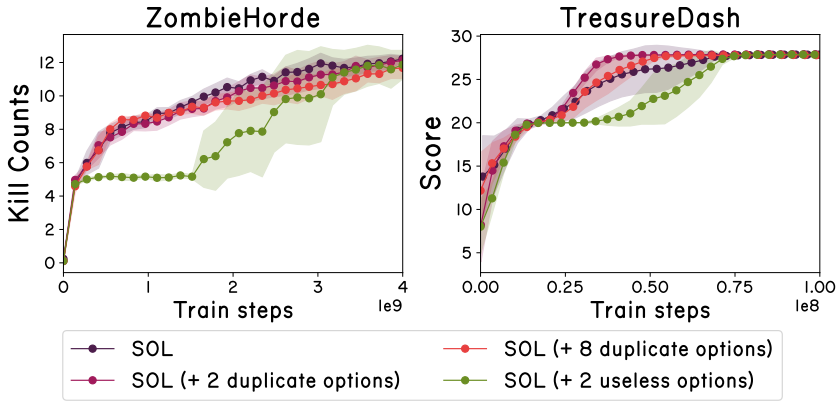


Figure 15: Performance of `SOL` with duplicate or useless options added. Shaded region represents two standard errors computed over 5 seeds.

F.6 FLAT APPO RESULTS ON POINTMAZE

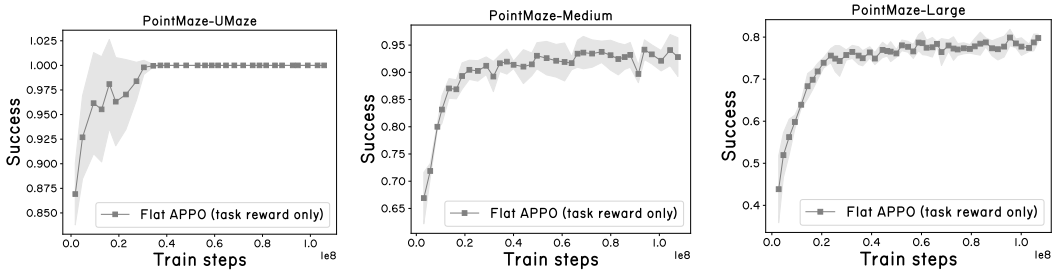


Figure 16: Flat APPO agents trained on default PointMaze environments from Gymnasium Robotics (de Lazcano et al., 2024) are largely able to solve all PointMaze environments, indicating that hierarchy is not needed for these maze layouts.

G ADDITIONAL DISCUSSION

G.1 RELATIONSHIP TO AGENT57

A key difference between `SOL` and Agent57 (Badia et al., 2020) is that `SOL` switches between different sub-policies within the same episode, whereas Agent57 executes the same policy for the entire episode. Switching between different sub-policies within an episode is essential for the tasks we consider, for example alternating between fighting and healing in NetHack. This is illustrated in Appendix F.2, where the controller calls the `Health` option earlier in the episodes while the agent has low experience levels or when its health is low, and the `Score` option later on in the episodes once it has gained levels and is stronger. Another difference is that `SOL`'s controller is observation-conditioned, whereas the controller in Agent57 is a bandit which does not take observations as input.

This difference in turn influences several of the design decision in Section 3.3. For example, jointly training a controller that is observation-conditioned together with the option policies requires a different neural network architecture that also outputs distributions over options and option execution lengths, in addition to low-level environment actions (illustrated in Figure 8, Appendix E.2). It also requires the wrapper in the actor workers (shown in more detail in Appendix E.3) to duplicate observations each time the controller is called and record the currently active option and option execution length. Finally, switching between different option policies within the same episode requires particular handling of the return/advantage computations (detailed in Appendix E.4). Each time the option policy changes (say from option A to option B), we need to bootstrap the returns of option A using the last observation where option A was active and mask the future rewards gathered when executing option B.