Joint Learning of Hierarchical Neural Options and Abstract World Model

Wasu Top Piriyakulkij [† 1] **Wolfgang Lehrach** [2] **Kevin Ellis** [* 1] **Kevin Murphy** [* 2]

Abstract

Building agents that can perform new skills by composing existing skills is a long-standing goal of AI agent research. Towards this end, we investigate how to efficiently acquire a sequence of skills, formalized as hierarchical neural options. However, existing model-free hierarchical reinforcement algorithms need a lot of data. We propose a novel method, which we call AgentOWL (**O**ption and **W**orld model **L**earning Agent), that jointly learns—in a sample efficient way—an abstract world model (abstracting across both states and time) and a set of hierarchical neural options. We show, on a subset of Object-Centric Atari games, that our method can learn more skills using much less data than baseline methods.

1. Introduction

For decision-making agents, an important goal is the cumulative acquisition of new skills, in tandem with an ever-expanding knowledge of how those skills affect the outside world. For example, we want our agents to first learn to pick up objects, then to pour drinks, and eventually to make a cup of coffee, while also learning how each skill affects the outside world, so that the agent can plan to achieve new goals such as getting coffee for a room full of people.

We formalize this compositional skill learning using the options framework (Sutton et al., 1999): an agent learns a sequence of options $o_{1:n}$ that achieve increasingly difficult goals, $g_{1:n}$. Each option contains a policy that achieves its specific goal, and which can use options learned earlier, forming a deep hierarchy of skills (Figure 1 right).

But hierarchical options are challenging to learn, because as we acquire more options, we effectively expand our action space, making policy learning less tractable whenever we face with a new goal. This introduces a tradeoff between learning new skills quickly, and how many skills we have acquired. Applications of standard model-free RL to learning option hierarchies therefore require increasingly more samples as learning progresses (Kamat & Precup, 2020; Abdulhai et al., 2022; Nica et al., 2022).

To resolve this tradeoff, we instead turn to model-based reinforcement learning (Kaelbling et al., 1996; M. Moerland et al., 2023). By modeling the effects of options and planning in that world model, we can rule out many options before trying them out in the real world, effectively using the world model to improve sample efficiency. Moreover, modeling option effects produces temporally abstract world models, overcoming the "one-step trap" (Sutton, 2025; Asadi et al., 2019) and promising more tractable planning than low-level world models. However, for this approach to actually improve sample efficiency, we also need a world modeling approach that is data-efficient.

In this work, we propose a novel world model whose representation combines symbolic code with non-parametric distributions, which allows learning the world model from little data. The world model abstracts over states and time. We combine this with a method to learn hierarchical options. We call the resulting system AgentOWL, which stands for **O**ption and **W**orld model **L**earning Agent.

We apply our method to 3 hard object-centric Atari (OCAtari) games, namely Montezuma's Revenge, Pitfall, and Private Eye. We show that AgentOWL acquires the highest number of skills compared to other baselines. We also show that AgentOWL has various unique capabilities that the baselines lack.

2. Background

Problem Setting. An environment can be described as a goal-conditioned MDP $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{G}, \gamma)$. We assume states can be broken down into primitive features $\mathcal{S} = S_1 \times ... \times S_n$, meaning the state is symbolic, allowing us to focus on the skill learning problem without mixing in the well-known challenges of representation learning of pixel inputs. The set $\mathcal{A}$ enumerates primitive actions, e.g., LEFT, RIGHT, UP, etc. in 2d video games. The goals $\mathcal{G} = (g_1, g_2, \cdots, g_m)$ are an ordered sequence of goal predicates, $g_i : \mathcal{S} \to \{0, 1\}$, each

[†]Work done during an internship at Google DeepMind,[*]Equal advising [1]Cornell University [2]Google DeepMind. Correspondence to: Wasu Top Piriyakulkij <wp237@cornell.edu>.

Preprint. February 2, 2026.

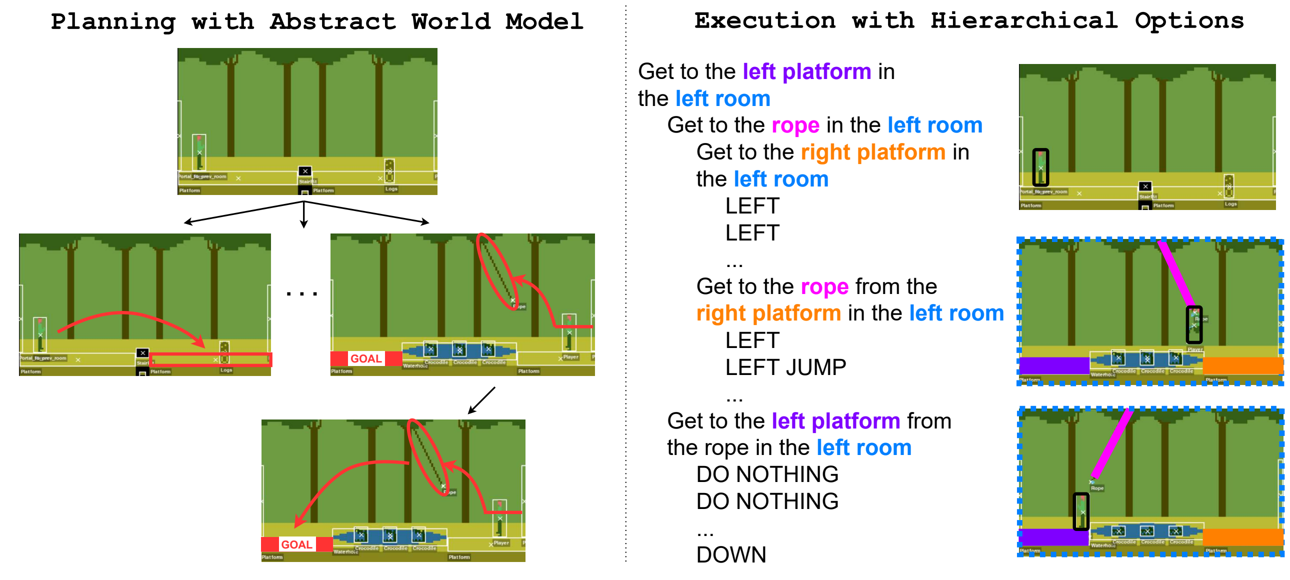

Figure 1. Illustration of hierarchical planning and execution in AgentOWL where the goal is to go to the left platform in the left room. Left: AgentOWL goes through possible plans and successfully makes a short plan (two high level steps) in its abstract world model to reach the goal. Right: AgentOWL executes a hierarchical sequence of options. (The hierarchical structure is represented by the indentation.)

of which defines a reward function $R_{g_i}(\boldsymbol{s}, a, \boldsymbol{s}') = g_i(\boldsymbol{s}')$. We use the same discount factor γ for all goals, and assume that the transition function $\mathcal{T}$ does not depend on the goal. Episodes end when an agent reaches the goal or timeouts.

Options (Figure 1 right). An option is a learned skill. Formally, option o_i comprises a tuple (π_i, g_i) of a policy π_i which executes until its goal, g_i, is satisfied. The goal g_i serves as the *termination condition* of the option. We follow the call-and-return paradigm (Sutton et al., 1999); an option executes until its goal is satisfied, or it timeouts. Options may form a hierarchy: Option o_i has policy $\pi_i : \mathcal{S} \to \mathcal{A} \cup \{o_j\}_{j<i}$, meaning it can output either primitive actions in $\mathcal{A}$ or a previously learned option o_j (for $j < i$). We write Ω for a set of options. Adding options to the action space of an MDP forms a Semi-MDP (Puterman, 1994).

State abstraction (Figure 1 left). An option can change the state in complex ways, and for the agent to plan, it must predict those changes. To make this prediction problem tractable, we consider *state abstractions*, which are functions of the state that elide unpredictable or irrelevant features that would be hard to predict (Dean & Givan, 1997; Li et al., 2006). Formally, a state abstraction $\boldsymbol{f}(\boldsymbol{s})$ is a function of the state $\boldsymbol{s}$. When the state is clear from context, we abuse notation by writing $\boldsymbol{f}$ to mean $\boldsymbol{f}(\boldsymbol{s})$.

Abstract world models (Figure 1 left). Within the context of this work, an abstract world model predicts future abstract states, given the current state, and the current option. This implements temporal abstraction *and* state abstraction, because rather than predicting the immediate next state, we instead predict only its abstract features, and only at the time that the current option terminates. This prediction is written $p_o(\boldsymbol{f}' \mid \boldsymbol{s})$. This conditions the abstract world model on the full state but predicts only the future abstract state.

PoE-World (Figure 2 left). Piriyakulkij et al. (2025) introduces PoE-World, a framework for learning structured world models from little data. World models are represented using a product-of-experts, where each expert is a short symbolic program. Intuitively, each program models an independent causal mechanism in the world, and by encoding each program as a snippet of Python, they become learnable using LLMs. Given current state $\boldsymbol{s}$ and action a, the next state $\boldsymbol{s}'$ follows

$$p_{\boldsymbol{\theta}}(\boldsymbol{s}'|\boldsymbol{s}, a) = \prod_j p(s_j'|\boldsymbol{s}, a) \tag{1}$$

$$p(s_j'|\boldsymbol{s}, a) = \frac{1}{Z_j} \prod_{i:j(i)=j} p_i(s_j'|\boldsymbol{s}, a)^{\theta_i}. \tag{2}$$

where $j(i)$ is the target feature dimension modeled by expert i. Learning with PoE-World means generating experts p_i with LLMs and estimating weights $\boldsymbol{\theta}$, which requires little data (few $(\boldsymbol{s}, a, \boldsymbol{s}')$ triples) because it is not learning a fully parametric model. The model assumes the state features are conditionally independent; this makes it tractable to compute the partition function, Z_j, and hence we can perform maximum likelihood estimation (MLE) of the weights through gradient descent. On object-centric Atari (OCAtari) (Delfosse et al., 2023), PoE-World takes only a few minutes of gameplay to assemble a working world model. We use PoE-World to learn an abstract world model.

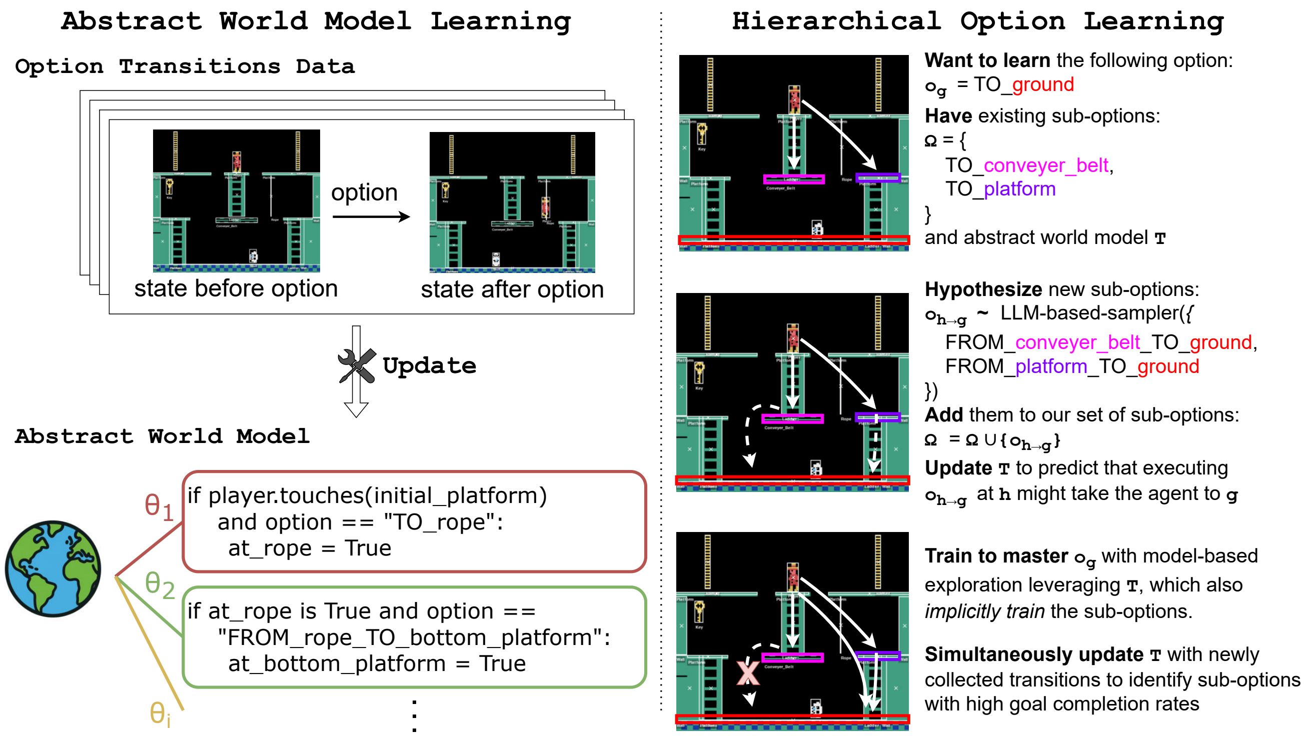

Figure 2. Illustration of how AgentOWL learns its abstract world model (left) and hierarchical options (right). Left: given a dataset of option transitions, we learn an abstract world model using (an extension of) the method of (Piriyakulkij et al., 2025). Specifically, each expert is generated using LLM code synthesis, and the weight for each expert (denoted θ_i) is learned using gradient descent on the likelihood objective. Right: AgentOWL learns a new option to achieve a new goal by leveraging previously acquired sub-options. Specifically, given the goal, we ask an LLM to hypothesize new sub-options (building on the already learned ones) that might help achieve the goal. The agent trains to master the highest-level option, o_g, which implicitly train these new sub-options, and simultaneously update our abstract world model to identify sub-options with high goal completion rates. Eventually, the agent masters the target option o_g by composing good sub-options. Detailed pseudocode can be founded in Algorithm 1.

3. Method

We propose AgentOWL (**O**ption and **W**orld model **L**earning Agent), an agent that sample-efficiently learns a sequence of options $\{o_i\}$, given a goal-conditioned MDP and a sequence of goals $\{g_i\}$. We describe our abstract world modeling approach embedded in AgentOWL in Section 3.1 and then the full AgentOWL in Section 3.2.

3.1. Abstract World Modeling

How should we abstract the state in order to reason about the effects of options? Each goal predicate must be in the state abstraction in order to successfully capture how each option transforms the state. In principle, further predicates may be important to include so that the abstract state is sufficiently informative, but recall that the abstract world model conditions on the current state $\boldsymbol{s}$, so any further features can still be extracted from the current state. We therefore define a state abstraction using just the goal predicates:

$$\boldsymbol{f}(\boldsymbol{s}) = (g_1(\boldsymbol{s}), g_2(\boldsymbol{s}), g_3(\boldsymbol{s}), \dots) \tag{3}$$

Next, we need an abstract world model that can be used for model-based lookahead. We learn $p_o(\mathbf{f}'|\mathbf{s})$ using PoE-World (Piriyakulkij et al., 2025), because by using symbolic programs to represent the abstract dynamics of the world, we can generalize more strongly from fewer examples. Indeed, symbolic rules have long been an attractive representation for modeling coarse-grained world dynamics (Fikes & Nilsson, 1971; McDermott, 2000).

But even using symbolic programs, abstract states contain many abstract features, requiring many samples to learn. To maintain sample efficiency, we impose a "frame axiom prior" on the abstract world model, which biases it toward believing that option o_i tends to change f_i (from achieving g_i), but does not usually change f_j for $j \neq i$. The frame axiom prior is implemented by incorporating $p(\theta_i)$ into Equation (2), turning weight optimization into a maximum a posteriori estimation (MAP) instead of MLE. We use Gaussian priors $p(\theta_i) = \mathcal{N}(\mu, \sigma^2)$ with $\sigma = 0.1$ and $\mu = 0.5$ for experts that do not change f_j, and $\mu = 0.001$ for the ones that do. This "frame prior" is commonly used in the planning community, as it is employed, in a much stronger form, in PDDL (McDermott, 2000).

Algorithm 1 AgentOWL learning algorithm

1: Given a target goal to achieve g, a list of sub-options Ω and corresponding policies $\pi_\Omega = \{\pi_o\}_{o\in\Omega}$, a current abstract world model $T = \{p_o\}_{o\in\Omega}$, a set of seen option transitions $D_\Omega = \{D_o\}_{o\in\Omega}$, an environment E, and stabilization thresholds n (number of samples) and δ (goal completion rate).
2: $\pi_g^{real} \leftarrow InitPolicy(E, \Omega)$ ▷ Initialize policy
3: $\epsilon \leftarrow 1$ ▷ Initialize ϵ which will anneal over time to 0
4: **while** o_g for goal g is not mastered **do**
5: $\quad \Omega_g \leftarrow \{o : g_o = g \mid o \in \Omega\}$ ▷ Set of options whose corresponding goals match the target goal
6: $\quad \Omega_{stable-g} \leftarrow \{o : n_o > n$ or $\delta_o > \delta \mid o \in \Omega_g\}$ ▷ Subset of Ω_g with only "stable" sub-options
7: $\quad \Omega_{good-g} \leftarrow \{o : \delta_o > \delta \mid o \in \Omega_g\}$ ▷ Subset of Ω_g with only good, high-performing options
8: $\quad$ *# If all relevant options are stable but none are good enough, then generate new subgoal*
9: $\quad$ **if** $|\Omega_g| = |\Omega_{stable-g}|$ and $|\Omega_{good-g}| = 0$ **then**
10: $\qquad h \leftarrow LLM(g, \Omega, D_\Omega)$ ▷ Sample a precondition with LLM
11: $\qquad \Omega \leftarrow \Omega \cup \{o_{h\to g}\}$ ▷ Add a new sub-option $o_{h\to g}$ to Ω
12: $\qquad T \leftarrow T \cup \{p_{o_{h\to g}}\}$ ▷ Add the corresponding hypothetical option model $p_{o_{h\to g}}$ to T
13: $\qquad \pi_g^{real} \leftarrow InitPolicy(E, \Omega)$ ▷ Reinitialize the policy as Ω changes
14: $\quad$ **end if**
15: $\quad \pi_g^{wm} \leftarrow DQN(T, \Omega, R_g)$ ▷ Learn π_g^{wm} from scratch in the abstract world model T with Ω.
16: $\quad$ *# Train hierarchically to master an option for goal g in E. This may change any $\pi \in \pi_\Omega$, in addition to π_g.*
17: $\quad ((\pi_g^{real}, \pi_g^{wm}, \epsilon), \pi_\Omega, D_\Omega) \leftarrow HierarchicalDQN(E, \Omega, D_\Omega, \pi_\Omega, g, n, \delta, \pi_g = (\pi_g^{real}, \pi_g^{wm}, \epsilon))$
18: $\quad T \leftarrow PoEWorld(T, D_\Omega)$ ▷ Fit T with the updated D_Ω using PoE-World
19: **end while**
20: $o_g \leftarrow ((\pi_g^{real}, \pi_g^{wm}, \epsilon), g)$
21: **return** o_g, Ω, T, D_Ω ▷ Return the target option o_g and the updated Ω, T, D_Ω

PoE-World yields $p_o(\boldsymbol{f}' \mid \boldsymbol{s})$, but learning only this conditional distribution is insufficient because it cannot chain together several options: After running the first option in state $\boldsymbol{s}$, we arrive in $\boldsymbol{f}'$, but predicting the effect of a second option might need the full state $\boldsymbol{s}'$. We heuristically predict $\boldsymbol{s}'$ from $\boldsymbol{f}'$ using a kernel density estimator $w(\boldsymbol{s}' \mid \boldsymbol{f}')$ that samples full states $\boldsymbol{s}'$ given an abstract state $\boldsymbol{f}'$:

$$p_o(\boldsymbol{s}' \mid \boldsymbol{s}) \approx \mathbb{E}_{\boldsymbol{f}'\sim p_o(\cdot|\boldsymbol{s})}\left[w(\boldsymbol{s}' \mid \boldsymbol{f}')\right] \qquad (4)$$

Note this is approximate: $\boldsymbol{s}'$ generally depends on $\boldsymbol{s}$ and o, even conditional on $\boldsymbol{f}'$. This approximation is common in the hierarchical decision-making literature, where w is called a *weighting function* (Bertsekas, 1995; Li et al., 2006). Weighting functions allow sampling states from an abstract state without learning the MDP transition function, and without training a parametric generative model over the raw state space, both of which would require enormous data. To the extent that the environment can be accurately modeled using $p_o(\boldsymbol{f}' \mid \boldsymbol{s})$ where p_o only depends on the state abstraction $\boldsymbol{f}(\boldsymbol{s})$, this approximation becomes exact. Section A.3 contains implementation details of our abstract world modeling approach.

3.2. Joint Learning of Hierarchical Neural Options and Abstract World Model

Using this world modeling setup, we now introduce the full AgentOWL. It iteratively trains the next option to achieve the next goal (and in its world model) by calling Algorithm 1, whose three main ideas are described below.

Model-based exploration. Intuitively, planning in our world model should offer good guidance to a model-free policy; we can weigh trajectories in imagination before deciding what to try in the real world. Concretely, we run RL (specifically, deep Q-learning (DQN)) in the abstract world model yielding a policy π^{wm} (Algorithm 1 line 15). Note that this is computationally cheap, since the abstract world model T takes large steps, and is defined on a fairly low-dimensional symbolic state space, which allows us to use simple MLPs to represent the policy.[1]

The resulting policy, π^{wm}, serves as an exploration policy for training a policy in the real world, π^{real}, that learns to achieve a goal. More precisely, each option comprises a policy and goal, $o = (\pi, g)$, and we further decompose the policy into $\pi = (\pi^{real}, \pi^{wm}, \epsilon)$, where ϵ is the probability of taking exploratory actions (actions the world model predicts):

$$\pi(a \mid \boldsymbol{s}) = (1-\epsilon)\,\pi^{real}(a \mid \boldsymbol{s}) + \epsilon\,\pi^{wm}(a \mid \boldsymbol{s}) \qquad (5)$$

The decomposition ensures we can still learn a good policy even with imperfect world model. By annealing ϵ from 1 to

[1]Note π^{wm} could be computed using a different strategy, such as planning in the world model, rather than RL in the world model. We leave exploring these alternatives for π^{wm} for future work.

0, the agent eventually stops relying on π^{wm} and falls back on fully model-free RL learning of π^{real}. The reason we do this is that model-based learning is known to be sensitive to model inaccuracy (Gu et al., 2016; Janner et al., 2019).

We note that for AgentOWL, each policy has its own set of weights; there is no weight sharing between the policies.

Hypothesizing sub-options to achieve a target goal. Planning (or RL) to achieve a new goal is challenging unless we *already* have an option which reaches that goal, which therefore could serve as a sub-option. Absent such sub-options, the agent would need to reason about how its low level actions could be used to reach the new goal, defeating the whole point of a temporally abstract world model. For example, if we have a sub-option to "pick up the cup" and a target goal of "fill the cup with water", a successful plan might first "pick up the cup" followed by a long sequence of low level actions.

To shorten our abstract plans, and help "plan in the now" (Kaelbling & Lozano-Pérez, 2011), we let the agent hypothesize new sub-options that aim to achieve the target goal given that certain preconditions are satisfied (see Figure 2 (right) and Algorithm 1 line 10-13). We use LLMs to propose the preconditions, h, of a new option $o_{h \to g}$. This new option, and its corresponding hypothetical option model $p_{o_{h \to g}}$, is then added to the set of options Ω and the abstract world model T respectively.

For this work, we restrict preconditions to the form $h(\boldsymbol{s}) = \boldsymbol{f}(\boldsymbol{s})_i$, representing the completion of the sub-goal with index i. Concretely, we prompt Gemini 2.5 Flash to pick, among the sub-goals that the agent can already achieve with existing sub-options Ω, a sub-goal that would be useful towards the target goal. In the prompt, we include a sampled state from our set of seen transitions D_Ω to be included as part of the prompt. If the game has multiple rooms, we sample one state for each room (each state contains a "room number" object, so this can be easily done). The exact prompt used can be founded at Section A.4.

Stable training of hierarchical options. Hierarchical option training is done in Algorithm 1 line 17 using a hierarchical version of DQN. It proceeds similarly to typical DQN: executing the policy to collect data in the replay buffer and optimizing the policy using samples from the replay buffer. However, in hierarchical DQN, the execution is hierarchical (Figure 1 right); the agent executes the root-level option, o_g, which then recursively calls sub-options until a primitive action is executed. We also assign each option its own replay buffer to keep the data it collects with its own policy.[2] Each time an option has collected enough new datapoints, the agent optimizes the option's policy weights for a fixed number of steps. Because of hierarchical execution, any sub-option may collect data and have its weights updated. We describe hierarchical DQN in more details in Section A.1.

Nevertheless, hierarchical DQN can be unstable, because each higher-level option faces a non-stationary environment: Training lower level options changes the transition dynamics as seen by higher level options (Nachum et al., 2018). To mitigate this instability, note that an option's policy stabilizes once it has been trained with enough samples, or it reliably achieves its goals. We therefore disregard episode data for option training which contains an execution of at least one sub-option o with $n_o < n_{threshold}$ and $\delta_o < \delta_{threshold}$, where n_o is the number of samples the option has been trained with, δ_o is the option's goal completion rate over the 100 most recent episodes, and $n_{threshold}, \delta_{threshold}$ are hyperparameters. More details on stable Hierarchical DQN can be founded at Section A.1.

4. Experimental Results

Experimental setup. Each agent will be given an ordered sequence of target goals $(g_1, ..., g_n)$. The task for each agent is to "master" a set of neural options that correspond to the target goals, $(o_1, ..., o_n)$. We consider an option "mastered" when the goal completion rate of that option surpasses a threshold $\delta_{threshold}$. For practicality, we approximate the goal completion rate by averaging goal completions over the 100 most recent execution of the option and set $\delta_{threshold} = 0.5$. We evaluate the number of environment steps each agent uses to master this set of neural options.

Domains. We conduct our experiments on a subset of object-centric Atari games. Object-centric Atari (OCAtari) (Delfosse et al., 2023) provides an object parser on top of Atari games (Bellemare et al., 2013), transforming the inputs from pixels to sets of objects. Each object is described by object type (player, platform, ladder, etc.) and bounding box coordinates; we treat these values as primitive features. We additionally add a "room number" object to each state to indicate the room of the state.

We evaluate on the subset of games commonly used to study hard exploration in RL (Aytar et al., 2018; Ecoffet et al., 2021; Hosu & Rebedea, 2016), specifically Montezuma's Revenge, Pitfall and Private Eye, selected based on available computational resources.

For each game, we construct a small sequence of goals ordered by difficulty (for reasons discussed in Section 6) with the following procedure: We manually select a few rooms in each game, as each may have many rooms, e.g., Pitfall contains 255 rooms. We then define our list of goals as touching each possible object within these selected rooms.

[2] Note that we could be even more sample-efficient if we maintain a single shared replay buffer. We leave this for future work.

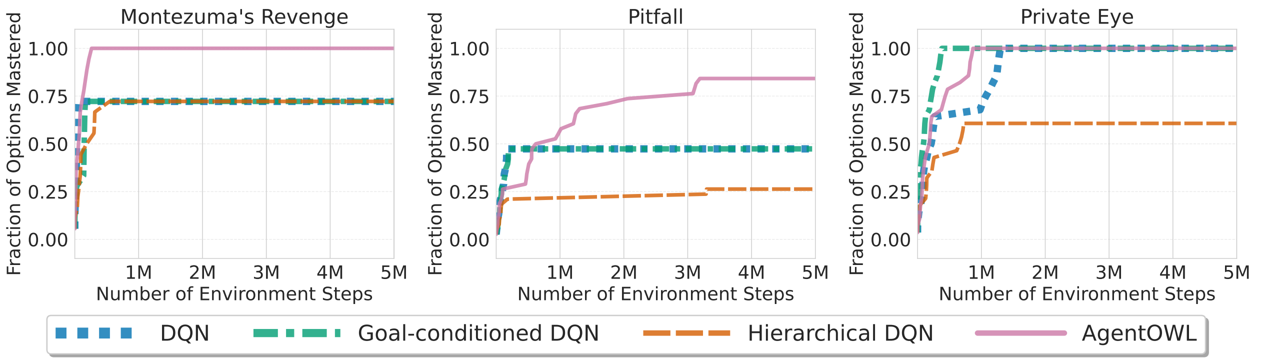

Figure 3. Fraction of options mastered vs number of environment steps for the three OCAtari's games we test on: Montezuma's Revenge, Pitfall, and Private Eye. Option is acquired once its success rate for the recent episodes reaches threshold $\delta = 0.5$.

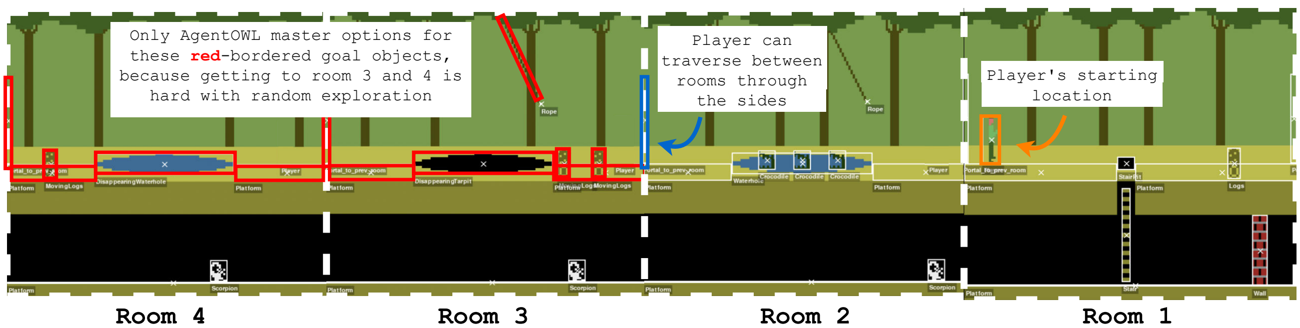

Figure 4. Screenshots of 4 rooms of Pitfall stitched together. Player starts in Room 1 (rightmost) and can traverse to other rooms through the sides of the screen. Goals that only AgentOWL masters within 5M environment steps (Figure 3 middle) are shown in red borders.

Finally, we manually order these goals by difficulty such that the earlier goals serve as stepping stones for later goals. (We leave automated curriculum learning to future work.) Full details on the experimental setup and domains can be founded at Section A.2.

Baselines. **Rainbow DQN** (Hessel et al., 2018) is an improved version of DQN (Deep Q-Network) (Mnih et al., 2015), a standard off-policy RL algorithm commonly used in discrete action settings. All DQNs used in the paper are Rainbow DQNs. **Goal-conditioned DQN** is DQN with weight sharing between the policies of the options. Specifically, instead of learning $\pi_1(a \mid \boldsymbol{s}), ..., \pi_n(a \mid \boldsymbol{s})$, we seek to learn a goal-conditioned policy $\pi(a \mid \boldsymbol{s}, g)$. **Hierarchical DQN** is DQN whose policy has an action space that includes previously learned sub-options. Hierarchical DQN can be seen as AgentOWL without its abstract world model. Implementation details of DQN and Hierarchical DQN can be founded at Section A.1

Results. As shown in Figure 3, AgentOWL masters the highest number of options for most number of environment steps. Although the baselines without hierarchical options seem to be better than AgentOWL at lower training sample budget, their performance plateaus at a much lower number of options mastered compared to that of AgentOWL. Qualitatively, these baselines fail to acquire options for harder goals (Figure 4) because they never discover any action sequence that can achieve the harder goals, as the number of possible action sequences grows exponentially with the number of steps needed to achieve the goals. In a sense, AgentOWL succeeds here because of its focused exploration: It manages to accomplish a goal that requires a long, complicated sequence of primitive actions by planning abstractly with higher-level options. Abstract plans can be very short, allowing them to be found very easily in both the real world and the abstract world model.

We perform an ablation study in Figure 5. Removing techniques introduced in Section 3.2—specifically LLM-based sub-goal proposal and hierarchial training stabilization—degrades AgentOWL's performance across most environment steps. These ablated systems need more data to master the same number of options across three games, and, in Pitfall, plateau at fewer mastered options than the full system.

Zero-shot generalization to novel situations. Another benefit of having a world model is zero-shot generalization to novel situations. We demonstrate this capability in Table 1. In OCAtari, the starting state of each game is always exactly the same. Looking forward to more complex domains, we want agents that can flexibly complete goals

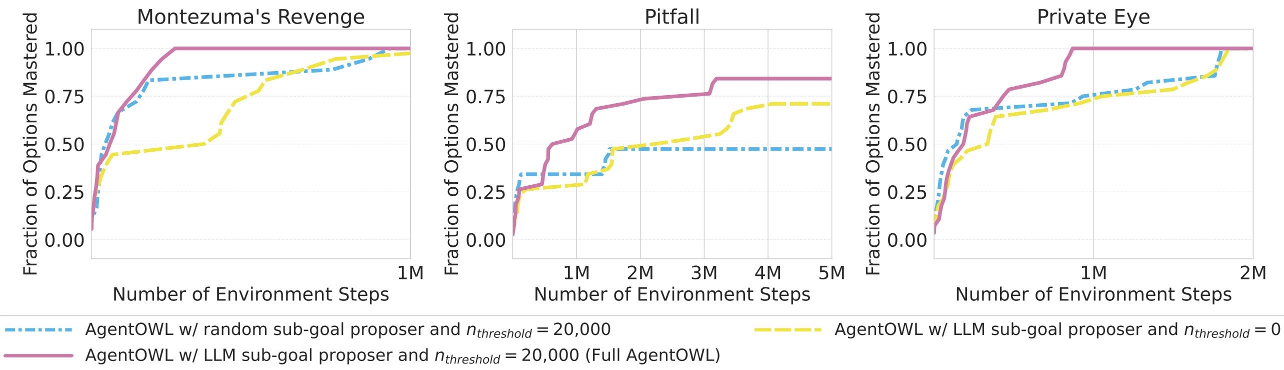

Figure 5. Ablation study: Removing core components of AgentOWL lead to fewer options being acquired and/or more data from the environment being needed. Setting $n_{threshold} = 0$ means stabilization for hierarchical DQN is not implemented.

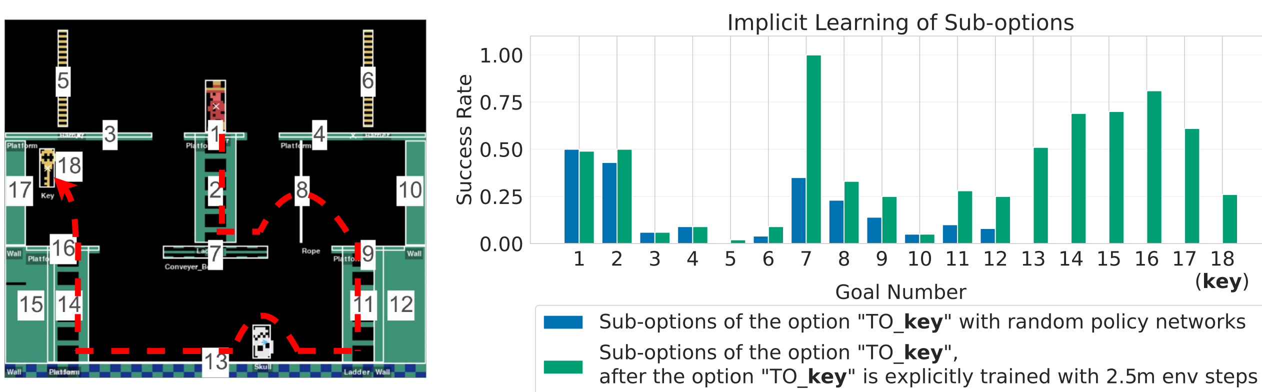

Figure 6. Left: Illustration of the goal labels (where all goals are of the form "touch this specific object") and the path the player needs to take to get the key from the starting position. Right: The success rates of the sub-options implicitly trained while the agent is training for the goal "key", compared to those of sub-options with randomly initialized policy network. The sub-options for the sub-goals within the path to retrieve the key, such as goal #13-17, has high success rates, because they are useful to its parent-option "TO_key". On the other hand, unhelpful goals, such as goal #3-6, remain poorly learned.

Method	Goal 1	Goal 2	Goal 3
Random	0.06	0.00	0.00
Goal-conditioned DQN	0.00	0.00	0.00
AgentOWL w/o the new option	0.02	0.01	0.26
AgentOWL w/ the new option	**1.00**	**1.00**	**1.00**

Table 1. Goal completion rates, over 100 episodes, to previously seen goals but from a new starting state in Private Eye. We first trained agents to achieve goals starting from an initial position in PrivateEye (Figure 3 right). We then give a new option to navigate from the new starting state to the initial starting state to AgentOWL, allowing it to achieve the goals from the new starting position without any additional training.

in novel situations, such as picking up a cup in a novel kitchen, or clearing a randomly generated level in Minecraft or Nethack (Johnson et al., 2016; Küttler et al., 2020). Thus, we design experiments where each agent needs to accomplish goals it has already mastered an option for, but from a new starting state. In Table 1, we show that after learning an option to travel from a new starting state back to the game's original starting state, AgentOWL can compose existing options to achieve target goals zero-shot without any additional training data. It is unclear, on the other hand, how other baselines would perform zero-shot adaptation to novel situations without hierarchical options to compose sub-options and an abstract world model to plan on.

Implicit learning of sub-options. AgentOWL improves its sub-options through hierarchical DQN even when rewarded only for goals that do not correspond to them. In Figure 6, we perform an experiment to clearly demonstrate this implicit learning. We take an AgentOWL agent trained to achieve a goal sequence in Montezuma's Revenge (Figure 2 left), keep its learned abstract world model, but re-initialize the policy networks of all options with random weights. Then, we solely train this agent to master goal "key", i.e., to touch the key. Figure 6 (right) shows the performances of many re-initialized sub-options increase significantly after AgentOWL is trained to achieve goal "key". We observe that sub-options helpful to the target goal improve, while irrelevant options corresponding to sub-goals outside the successful trajectory (Figure 6 left) do not.

Intuitively, the agent leverages the learned world model to help decide a sequence of sub-goals to pursue to eventually achieve the target goal. This sequence of goals acts as a curriculum for the agent to follow. The agent refines these options to help it eventually achieve the target goal.

5. Related Work

Abstract and symbolic world models. The advent of LLMs has sparked interest in symbolic world models. WorldCoder and GIF-MCTS (Tang et al., 2024; Dainese et al., 2024) directly use LLMs to generate and refine a world model as a program. POMDP Coder (Curtis et al., 2024) and CWM (Lehrach et al., 2026) tackle the problem of partial observability. PoE-World (Piriyakulkij et al., 2025) and OneLife (Khan et al., 2025) scale up symbolic world modeling with a product-of-experts world representation.

Like our work, several prior works have explored abstract world modeling with symbolic world representations. DECKARD (Nottingham et al., 2023) learns a directed acyclic graph as its abstract world model. Ada (Wong et al., 2024) synthesizes PDDL as its abstract world model. AgentOWL, on the other hand, learns a stochastic, symbolic world model capable of making both abstract and low-level predictions, offering greater representational power.

Option discovery and hierarchical RL. Option critic methods (Bacon et al., 2017; Harutyunyan et al., 2019; Tiwari & Thomas, 2019) learn both policies and termination functions of options end-to-end through gradient descent. Skill-chaining (Konidaris & Barto, 2009; Bagaria & Konidaris, 2019; Bagaria et al., 2021a) focuses on discovering chainable options, where the termination set of an option is the initiation set of another. Other hierarchical RL methods tend to have high-level policies and low-level policies, where the high-level ones either set goals or rewards for the low-level ones (Dayan & Hinton, 1992; Kulkarni et al., 2016; Vezhnevets et al., 2017; Li et al., 2019; Hafner et al., 2022) or directly select which low-level ones to execute (Florensa et al., 2017; Heess et al., 2016; Eysenbach et al., 2018). Our work takes inspiration from these prior works but differs in how we learn deeply hierarchical options, as opposed to the common two-level hierarchy of policies.

LLMs for RL. Existing works have explored the use of LLMs in RL agents in many ways, including assisting with reward design (Kwon et al., 2023; Ma et al., 2023; Klissarov et al., 2023; Castanyer et al., 2025a), serving as policies or policy generators (Yao et al., 2022; Wang et al., 2023; Liang et al., 2022), and producing high-level plans (Ahn et al., 2022; Huang et al., 2022; Singh et al., 2022; Song et al., 2023). AgentOWL leverages LLMs for world modeling and sub-goal proposal, but can also benefit from LLM-based reward design if reward functions are not provided. Importantly, AgentOWL learns neural policies through environmental interaction rather than relying on LLMs to directly output action sequences or policies.

6. Limitations and Future Direction

While AgentOWL efficiently learns abstract world models and hierarchical neural options in our setting, there are limitations. First, we assume the given sequence of goals is ordered by difficulty. What made AgentOWL effective is the ability to use sub-options to help achieve a hard goal. Consequently, AgentOWL can fail to achieve a challenging goal without first learning to accomplish its prerequisite sub-goals. In the future, we hope to use ideas from curriculum learning (Bengio et al., 2009) to automate this.

Second, we assume the number of goals is relatively small (< 100). While AgentOWL's model-based exploration keeps environment interactions from scaling linearly with the number of goals, training compute still does. Thus, currently, we reduce the number of environment samples at the expense of increasing compute, which is the right tradeoff only when compute is cheaper than data. Incorporating option affordances (Khetarpal et al., 2020; 2021) to reduce the number of applicable options could be a fruitful direction.

Lastly, we assume symbolic input to the abstract world model, using OCAtari instead of pixel-level Atari. Symbolic input permits learning world models like PoE-World, which is much more sample-efficient than learning a pixel-level world model. There is ongoing effort to learn abstract symbolic world models directly from pixels (Liang et al., 2024; 2025; Athalye et al., 2024), but merging that line of work with option training remains open. Purely neural world models have made great strides (Ball et al., 2025; Alonso et al., 2024), but do not learn explicit symbolic abstractions that can be used to reason over long horizons. Leveraging these purely neural models to efficiently learn neurosymbolic world models offers another path forward.

Limitations aside, AgentOWL's successful results emphasize that option and world model learning are deeply intertwined, and that skills are fundamental to an agent's understanding of its environment. We hope this insight draws attention to several underexplored research problems. For example, if we allow goals and corresponding skills to grow over time, our abstract model faces an ever-changing state and action space. How do we perform efficient world modeling in this online learning setting? With large sets of abstract features and skills, how do we reason over only relevant ones to save computation? Answering these questions could make systems like AgentOWL much more powerful.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Abdulhai, M., Kim, D.-K., Riemer, M., Liu, M., Tesauro, G., and How, J. P. Context-specific representation abstraction for deep option learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pp. 5959–5967, 2022.

Ahmed, Z., Tenenbaum, J. B., Bates, C. J., and Gershman, S. J. Synthesizing world models for bilevel planning. *arXiv preprint arXiv:2503.20124*, 2025.

Ahn, M., Brohan, A., Brown, N., Chebotar, Y., Cortes, O., David, B., Finn, C., Fu, C., Gopalakrishnan, K., Hausman, K., et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

Alonso, E., Jelley, A., Micheli, V., Kanervisto, A., Storkey, A., Pearce, T., and Fleuret, F. Diffusion for world modeling: Visual details matter in atari. In *Thirty-eighth Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.12399.

Asadi, K., Misra, D., Kim, S., and Littman, M. L. Combating the compounding-error problem with a multi-step model. *arXiv preprint arXiv:1905.13320*, 2019.

Athalye, A., Kumar, N., Silver, T., Liang, Y., Wang, J., Lozano-Pérez, T., and Kaelbling, L. P. From pixels to predicates: Learning symbolic world models via pretrained vision-language models. *arXiv preprint arXiv:2501.00296*, 2024.

Aytar, Y., Pfaff, T., Budden, D., Paine, T., Wang, Z., and De Freitas, N. Playing hard exploration games by watching youtube. *Advances in neural information processing systems*, 31, 2018.

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Bacon, P.-L., Harb, J., and Precup, D. The option-critic architecture. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31, 2017.

Bagaria, A. and Konidaris, G. Option discovery using deep skill chaining. In *International Conference on Learning Representations*, 2019.

Bagaria, A., Senthil, J., Slivinski, M., and Konidaris, G. Robustly learning composable options in deep reinforcement learning. In *Proceedings of the 30th International Joint Conference on Artificial Intelligence*, 2021a.

Bagaria, A., Senthil, J. K., and Konidaris, G. Skill discovery for exploration and planning using deep skill graphs. In *International conference on machine learning*, pp. 521–531. PMLR, 2021b.

Ball, P. J., Bauer, J., Belletti, F., Brownfield, B., Ephrat, A., Fruchter, S., Gupta, A., Holsheimer, K., Holynski, A., Hron, J., et al. Genie 3: A new frontier for world models. *Google DeepMind Blog*, pp. 253–279, 2025.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013.

Bengio, Y., Louradour, J., Collobert, R., and Weston, J. Curriculum learning. In *ICML*, 2009.

Bertsekas, D. *Dynamic Programming and Optimal Control*, volume 1. Athena Scientific, 01 1995.

Castanyer, R. C., Mohamed, F., Castro, P. S., Neary, C., and Berseth, G. Arm-fm: Automated reward machines via foundation models for compositional reinforcement learning. *arXiv preprint arXiv:2510.14176*, 2025a.

Castanyer, R. C., Obando-Ceron, J., Li, L., Bacon, P.-L., Berseth, G., Courville, A., and Castro, P. S. Stable gradients for stable learning at scale in deep reinforcement learning. *arXiv preprint arXiv:2506.15544*, 2025b.

Curtis, A., Matheos, G., Gothoskar, N., Mansinghka, V., Tenenbaum, J., Lozano-Pérez, T., and Kaelbling, L. P. Partially observable task and motion planning with uncertainty and risk awareness. *arXiv preprint arXiv:2403.10454*, 2024.

Dainese, N., Merler, M., Alakuijala, M., and Marttinen, P. Generating code world models with large language models guided by monte carlo tree search. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15383.

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

Dean, T. and Givan, R. Model minimization in markov decision processes. In *AAAI/IAAI*, pp. 106–111, 1997.

Delfosse, Q., Blüml, J., Gregori, B., Sztwiertnia, S., and Kersting, K. Ocatari: Object-centric atari 2600 reinforcement learning environments. *arXiv preprint arXiv:2306.08649*, 2023.

Dietterich, T. G. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of artificial intelligence research*, 13:227–303, 2000.

Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K. O., and Clune, J. First return, then explore. *Nature*, 590(7847): 580–586, 2021.

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

Fikes, R. E. and Nilsson, N. J. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

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

Fortunato, M., Azar, M. G., Piot, B., Menick, J., Hessel, M., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., and Legg, S. Noisy networks for exploration. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=rywHCPkAW.

Gu, S., Lillicrap, T., Sutskever, I., and Levine, S. Continuous deep q-learning with model-based acceleration. In *International conference on machine learning*, pp. 2829–2838. PMLR, 2016.

Hafner, D., Lee, K.-H., Fischer, I., and Abbeel, P. Deep hierarchical planning from pixels. *Advances in Neural Information Processing Systems*, 35:26091–26104, 2022.

Harutyunyan, A., Dabney, W., Borsa, D., Heess, N., Munos, R., and Precup, D. The termination critic. *arXiv preprint arXiv:1902.09996*, 2019.

Heess, N., Wayne, G., Tassa, Y., Lillicrap, T., Riedmiller, M., and Silver, D. Learning and transfer of modulated locomotor controllers. *arXiv preprint arXiv:1610.05182*, 2016.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. Rainbow: Combining improvements in deep reinforcement learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Hosu, I.-A. and Rebedea, T. Playing atari games with deep reinforcement learning and human checkpoint replay. *arXiv preprint arXiv:1607.05077*, 2016.

Huang, W., Abbeel, P., Pathak, D., and Mordatch, I. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International conference on machine learning*, pp. 9118–9147. PMLR, 2022.

Janner, M., Fu, J., Zhang, M., and Levine, S. When to trust your model: Model-based policy optimization. *Advances in neural information processing systems*, 32, 2019.

Johnson, M., Hofmann, K., Hutton, T., and Bignell, D. The malmo platform for artificial intelligence experimentation. In *Ijcai*, volume 16, pp. 4246–4247, 2016.

Kaelbling, L. P. and Lozano-Pérez, T. Hierarchical task and motion planning in the now. In *2011 IEEE International Conference on Robotics and Automation*, pp. 1470–1477. IEEE, 2011.

Kaelbling, L. P., Littman, M. L., and Moore, A. W. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

Kamat, A. and Precup, D. Diversity-enriched option-critic. *arXiv preprint arXiv:2011.02565*, 2020.

Khan, Z., Prasad, A., Stengel-Eskin, E., Cho, J., and Bansal, M. One life to learn: Inferring symbolic world models for stochastic environments from unguided exploration. *arXiv preprint arXiv:2510.12088*, 2025.

Khetarpal, K., Ahmed, Z., Comanici, G., Abel, D., and Precup, D. What can i do here? a theory of affordances in reinforcement learning. In *International Conference on Machine Learning*, pp. 5243–5253. PMLR, 2020.

Khetarpal, K., Ahmed, Z., Comanici, G., and Precup, D. Temporally abstract partial models. *Advances in Neural Information Processing Systems*, 34:1979–1991, 2021.

Klissarov, M., D'Oro, P., Sodhani, S., Raileanu, R., Bacon, P.-L., Vincent, P., Zhang, A., and Henaff, M. Motif: Intrinsic motivation from artificial intelligence feedback. *arXiv preprint arXiv:2310.00166*, 9 2023.

Konidaris, G. and Barto, A. Skill discovery in continuous reinforcement learning domains using skill chaining. *Advances in neural information processing systems*, 22, 2009.

Kulkarni, T. D., Narasimhan, K., Saeedi, A., and Tenenbaum, J. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *Advances in neural information processing systems*, 29, 2016.

Küttler, H., Nardelli, N., Miller, A., Raileanu, R., Selvatici, M., Grefenstette, E., and Rocktäschel, T. The nethack learning environment. *Advances in Neural Information Processing Systems*, 33:7671–7684, 2020.

Kwon, M., Xie, S. M., Bullard, K., and Sadigh, D. Reward design with language models. *arXiv preprint arXiv:2303.00001*, 2023.

Lehrach, W., Hennes, D., Lazaro-Gredilla, M., Lou, X., Wendelken, C., Li, Z., Dedieu, A., Grau-Moya, J., Lanctot, M., Iscen, A., et al. Code world models for general game playing. *ICLR*, 2026.

Li, L., Walsh, T. J., and Littman, M. L. Towards a unified theory of state abstraction for mdps. *AI&M*, 1(2):3, 2006.

Li, S., Wang, R., Tang, M., and Zhang, C. Hierarchical reinforcement learning with advantage-based auxiliary rewards. *Advances in Neural Information Processing Systems*, 32, 2019.

Li, X.-L. Preconditioned stochastic gradient descent. *IEEE transactions on neural networks and learning systems*, 29 (5):1454–1466, 2017.

Liang, J., Huang, W., Xia, F., Xu, P., Hausman, K., Ichter, B., Florence, P., and Zeng, A. Code as policies: Language model programs for embodied control. *arXiv preprint arXiv:2209.07753*, 2022.

Liang, Y., Kumar, N., Tang, H., Weller, A., Tenenbaum, J. B., Silver, T., Henriques, J. F., and Ellis, K. Visualpredicator: Learning abstract world models with neuro-symbolic predicates for robot planning. *arXiv preprint arXiv:2410.23156*, 2024.

Liang, Y., Nguyen, D., Yang, C., Li, T., Tenenbaum, J. B., Rasmussen, C. E., Weller, A., Tavares, Z., Silver, T., and Ellis, K. Exopredicator: Learning abstract models of dynamic worlds for robot planning. *arXiv preprint arXiv:2509.26255*, 2025.

Lyle, C., Zheng, Z., Nikishin, E., Pires, B. A., Pascanu, R., and Dabney, W. Understanding plasticity in neural networks. In *International Conference on Machine Learning*, pp. 23190–23211. PMLR, 2023.

Lyle, C., Zheng, Z., Khetarpal, K., van Hasselt, H., Pascanu, R., Martens, J., and Dabney, W. Disentangling the causes of plasticity loss in neural networks. *arXiv preprint arXiv:2402.18762*, 2024.

M. Moerland, T., Broekens, J., Plaat, A., and M. Jonker, C. Model-based reinforcement learning: A survey. *Foundations and Trends in Machine Learning*, 16(1):1–118, 2023.

Ma, Y. J., Liang, W., Wang, G., Huang, D.-A., Bastani, O., Jayaraman, D., Zhu, Y., Fan, L., and Anandkumar, A. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023.

McDermott, D. M. The 1998 ai planning systems competition. *AI magazine*, 21(2):35–35, 2000.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M. A., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533, 2015.

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

Ng, A. Y., Harada, D., and Russell, S. Policy invariance under reward transformations: Theory and application to reward shaping. In *Icml*, volume 99, pp. 278–287. Citeseer, 1999.

Nica, A., Khetarpal, K., and Precup, D. The paradox of choice: Using attention in hierarchical reinforcement learning. *arXiv preprint arXiv:2201.09653*, 2022.

Nottingham, K., Ammanabrolu, P., Suhr, A., Choi, Y., Hajishirzi, H., Singh, S., and Fox, R. Do embodied agents dream of pixelated sheep: Embodied decision making using language guided world modelling. In *International Conference on Machine Learning*, pp. 26311–26325. PMLR, 2023.

Parr, R. E. *Hierarchical control and learning for Markov decision processes*. University of California, Berkeley, 1998.

Piriyakulkij, W. T., Liang, Y., Tang, H., Weller, A., Kryven, M., and Ellis, K. Poe-world: Compositional world modeling with products of programmatic experts. *Advances in Neural Information Processing Systems*, 2025.

Puterman, M. L. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, New York, 1994. ISBN 978-0471727828.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html.

Singh, I., Blukis, V., Mousavian, A., Goyal, A., Xu, D., Tremblay, J., Fox, D., Thomason, J., and Garg, A. Progprompt: Generating situated robot task plans using large language models. *arXiv preprint arXiv:2209.11302*, 2022.

Song, C. H., Wu, J., Washington, C., Sadler, B. M., Chao, W.-L., and Su, Y. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 2998–3009, 2023.

Sutton, R. S. The oak architecture: A vision of superintelligence from experience. Invited talk at the Conference on Neural Information Processing Systems (NeurIPS), December 2025. URL https://neurips.cc/virtual/2025/loc/mexico-city/invited-talk/129132.

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

Tang, H., Key, D. Y., and Ellis, K. Worldcoder, a model-based LLM agent: Building world models by writing code and interacting with the environment. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=QGJSXMhVaL.

Tiwari, S. and Thomas, P. S. Natural option critic. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 5175–5182, 2019.

Vezhnevets, A. S., Osindero, S., Schaul, T., Heess, N., Jaderberg, M., Silver, D., and Kavukcuoglu, K. Feudal networks for hierarchical reinforcement learning. In *International conference on machine learning*, pp. 3540–3549. PMLR, 2017.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv: Arxiv-2305.16291*, 2023.

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., and Freitas, N. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pp. 1995–2003. PMLR, 2016.

Wong, L., Mao, J., Sharma, P., Siegel, Z. S., Feng, J., Korneev, N., Tenenbaum, J. B., and Andreas, J. Learning adaptive planning representations with natural language guidance. In *International Conference on Learning Representations (ICLR)*, 2024.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

A. Appendix

A.1. DQN and Hierarchical DQN

Here we describe our implementation of Rainbow DQN (Hessel et al., 2018) which is used in all baselines, including AgentOWL itself.

Neural network architecture. Rainbow DQN uses dueling network architecture (Wang et al., 2016) We use a 2-layer MLP as our feature extractor module. The hidden feature size is 256 for π^{real} and 128 for π^{wm} for both layers (so always 256 for baseline methods). Each linear layer is followed by a layer normalization (LN) layer (Ba et al., 2016), as recent studies found incorporating LN in the network of a deep RL algorithm to be highly beneficial (Lyle et al., 2023; 2024), and then a ReLU layer. After the feature extractor, the value and advantage layer is also 2-layer MLP with hidden feature size half of what is used in the feature extractor, but the linear layers are replaced with noisy linear layers (Fortunato et al., 2018), and LN and ReLU are only applied the first layer, since the last layer produces output.

For goal-conditioned DQN, we also learn an embedding for each goal and concatenate it to the input to the network.

Training. We base our implementation of (Rainbow) DQN and hierarchical DQN off of Stable Baseline's implementation (Raffin et al., 2021). The pseudocode for DQN is and hierarchical DQN is at Algorithm 2 and Algorithm 3 respectively, with Algorithm 4 and Algorithm 5 describing the helper functions used in hierarchical DQN. The common hyperparameter for all variants of DQN is listed at Table 2. There are extra hyperparameters for hierarchical DQN Table 3 and overridden hyperparameters for training π_g^{wm} Table 4.

Goal heuristics. Additionally, to speed up training, we use a heuristics based on the manhattan distance between the player object and the goal object. Specifically, we let the heuristic function h be $h(s) = 5 \cdot (1 - \frac{ManhattanDist(s_{player}, s_{g-object})}{400})$. In OCAtari, manhattan distance between any two object never exceeds 400. We incorporate this heuristics in our Q network as follows:

$$Q_{\psi}^{new}(s,a) = Q_{\psi}(s,a) + h_g(s) \tag{6}$$

where h outputs a manhattan distance between the player object and the goal g object in the input state.

Intuitively, a freshly initialized neural network typically outputs values around 0. Adding $h_g(s)$ to it makes Q_{new} outputs values around $h_g(s)$ instead. Over time, as we train Q^{new} more and more, the neural network learns to correct the heuristics value so that the output is close the true Q value (Ng et al., 1999).

Note that if the goal object is not visible in s, e.g., the goal object is in another room in the game, we let the heuristics value equal to 0.

A.2. Experimental setting and OCAtari details

Goal sequences for each game. For each game, we select a few rooms and define goals as touching objects within them. Then, we manually order the goals based on difficulty. Additionally, we give short natural language names to each goal, such as 'top_left_plat', 'mid_right_wall', etc. We display the screenshots of the rooms with goal objects labeled with their order in the goal sequence at Figure 7, Figure 8, and Figure 9. We note that the orders within each room in Pitfall and Private Eye tend not to matter too much as they are roughly the same in difficulty level.

Vectorizing list of objects. We vectorize each input object list into a fixed-size feature vector for the DQN policy. By using the maximum count for each object type, we assign each object to a unique location in the observation space. For example, with one player and up to two enemies, where each object is vectorized into a 8-dimensional feature vector, the observation space has size 24: the player occupies indices 0-8, the first enemy occupies indices 9-16, and so on.

To vectorize each object, we take its x and y coordinates and encode each value into a 4-dimensional vector using a positional encoder implemented with sine and cosine functions, commonly used in LLMs.

Additionally, we include the goal values, i.e., the abstract state $\boldsymbol{f}(\boldsymbol{s})$, in the feature vector as well, so the feature vector contains both low-level and abstract features. To better match the dimensionality of the abstract features to that of the low-level features, we duplicate each abstract feature by 4.

Observation space optimization. To optimize running time, we choose to disregard static (non-moving) object types, such as platforms, ladders, etc., from the vector discussed above, as all states have the same information for these objects. This speeds up our code significantly, reducing the low-level feature vector size from $248, 344, 408$ to $24, 176, 328$ for Montezuma's Revenge, Pitfall, and Private Eye, respectively.

Episode timeouts. Since stable hierarchical DQN only adds data to the replay buffer once an episode ends, we need to make sure that episodes do not go on forever. Toward that end, we set a maximum time limit for all episodes in a (real) environment to be $1,000$ environment steps. On the other hand, when we treat our abstract world model as a simulated, abstract environment, we set the time limit to be only 4 environment steps, as this simulated environment is very abstract and only requires very short action sequences to achieve the goal.

A.3. Abstract world modeling implementation details

Learning $p_o(\boldsymbol{f}'|\boldsymbol{s})$ with PoE-World. We first assume the following structure for $p_o(\boldsymbol{f}'|\boldsymbol{s})$:

$$p_o(\boldsymbol{f}'|\boldsymbol{s}) \triangleq p_o(f'_o|\boldsymbol{s})(\prod_{i\neq o} p_o(f'_i|f'_o, \boldsymbol{s})) \tag{7}$$

Then, each $p_o(f'_o|\boldsymbol{s})$ and $p_o(f'_i|f'_o, \boldsymbol{s})$ can be modeled with a product of experts and learned with PoE-World. And as mentioned in the main text, for $p_o(f'_i|f'_o, \boldsymbol{s})$, we further incorporate Gaussian priors with $\sigma = 0.1$ and $\mu = 0.5$ for experts that do not change f_i, and $\mu = 0.001$ for the rest.

The intuition behind the above structure is the model should first predict f'_o which corresponds to the result of executing option o, whether or not it will achieve its corresponding goal g, with $p_o(f'_o|s)$. Based on the result of the execution, the model can then predict the rest of the abstract features with $\prod_{i\neq i_o} p_o(f'_i|f'_o, s)$.

Expert generation for PoE-World. For $p_o(f'_o|s)$ of each option o, we prompt Gemini 2.0 Flash-Lite to synthesize a set of possible preconditions using the prompt in Tables 6 and 7. We then add one expert per precondition that sets $f'_o = 1$ if that precondition is true. We also add a "blanket" expert that sets $f'_o = 0$ without any precondition. Both types of experts have Gaussian weight prior with $\mu = 0.5$, but the blanket expert has $\sigma = 0.001$, while the experts with preconditions have $\sigma = 0.1$. We give low σ to the blanket expert because we want its weight stays close to 0.5. If we allow the weight for the blanket expert to change too much, it might become 0 when only positive examples are observed during fitting, causing the model to incorrectly conclude that option o will always succeed without any preconditions.

For $p_o(f'_i|f'_o, s)$ of each option o, we generate three experts: $f'_i = f_i$ (no change), $f'_i = 1$, and $f'_i = 0$. As mentioned in Section 3.1, we give a Gaussian weight prior with $\mu = 0.5, \sigma = 0.1$ for the first type of expert (no change), and $\mu = 0.5, \sigma = 0.001$ for the latter two types.

Weighting function. We implement weighting functions straightforwardly as a lookup table with the keys being the abstract states and the values being the corresponding low-level states. We only keep a single low-level state for each abstract state in the lookup table, so there is no sampling.

Undefined distribution and partial state. It is possible that in the weighting function, we try to sample from $w(\boldsymbol{s} \mid \boldsymbol{f})$ when we have not yet seen a $\boldsymbol{s}$ that corresponds to $\boldsymbol{f}$, meaning our lookup table for the key $\boldsymbol{f}$ is empty. When that happens, we leave $\boldsymbol{s}$ unspecified. Concretely, we assign $None$ values to all primitive features. We call states with $None$ values for some features, partial states. Arriving at partial states in the world-model-simulated environment does not terminate the episode right away. The episode only terminates if the experts in $p_o(\boldsymbol{f}'|\boldsymbol{s})$ try to access the features with $None$ values.

Additionally, we also set $f'_i = None$ for all $i \neq o$ when we sample from $p_o(\boldsymbol{f}'|\boldsymbol{s})$ but get $f'_o = 0$. We implement this mechanism because options can lead to highly unpredictable states when they are not successful in achieving their corresponding goals.

A.4. Prompts for LLM-based sub-goal proposer

Table 5 contains the prompt used for LLM-based sub-goal proposer.

Algorithm 2 DQN learning algorithm (without parallel environments)

Given an environment E, a list of actions/sub-options Ω, , a reward function R, and training hyperparameters in Table 2.
Initialy policy network: $\pi \leftarrow InitPolicy(E, \Omega)$
Initialize replay buffer $D \leftarrow \{\}$
Get initial state: $\boldsymbol{s} \leftarrow E.reset()$
for $t \leftarrow 0$ to $total\_env\_steps$ step $train\_frequency$ **do**
  # Collect rollouts by executing π for $train\_frequency$ steps
  for $i \leftarrow 0$ to $train\_frequency$ **do**
    Sample action: $a \sim \pi(a \mid \boldsymbol{s})$
    Interact with environment: $\boldsymbol{s}' \leftarrow E.step(a)$
    Update replay buffer: $D \leftarrow D \cup \{(\boldsymbol{s}, a, \boldsymbol{s}', \kappa \cdot R(\boldsymbol{s}, a, \boldsymbol{s}'))\}$
    Update current state $\boldsymbol{s} \leftarrow \boldsymbol{s}'$
  end for
  # Then, train for $n\_gradient\_steps$ steps
  Optimize the policy weights: $DQNUpdate(\pi, D, n\_gradient\_steps)$
end for
return π

Algorithm 3 Stable Hierarchical DQN learning algorithm (without parallel environments)
Note that ExecuteOneStep and ReceiveObsOneStep are defined in Algorithm 4 and Algorithm 5 respectively

Given $E, \Omega, D_\Omega, \pi_\Omega, \pi_g, g, n, \delta$ and training hyperparameters in Table 2 and Table 3.

*# We define a **mutable** struct to keep track of execution variables of an option*
 # In hierarchical options, each parent option selects a child option to execute, recursing until reaching a primitive action
struct $OptionExecutionState$
  Option: o
  Policy: π
  Goal: g
  Active child option: $child$
  Child start state: u
  Episode data: D
  Current execution time: t
  New data counter: ct
end struct

Initialize (mutable) option execution states: $\omega_\Omega \leftarrow \{OptionExecutionState(o, \pi_o, g_o, None, None, [], 0, 0)\}_{o \in \Omega}$
Initialize (mutable) target option execution state: $\omega_{target} \leftarrow OptionExecutionState(o_g, \pi_g, g, None, None, [], 0, 0)$
Get initial state: $s \leftarrow E.reset()$
for $i \leftarrow 0$ to $total\_env\_steps$ **do**
  Execute the target option for one step: $a \leftarrow ExecuteOneStep(s, \omega_{target}, \Omega, \omega_\Omega)$
  Interact with environment: $s' \leftarrow E.step(a)$
  Update option states and replay buffers: $(\_, D_\Omega) \leftarrow ReceiveObsOneStep(s', \omega_{target}, \Omega, D_\Omega, n, \delta)$
  for $o \in \Omega$ **do**
    # optimize the policy weights if the option has collected enough new data
    if enough new data collected: $\omega_o.ct > train\_frequency$ **then**
      Optimize the policy weights: $DQNUpdate(\omega_o.\pi, D_o, n\_steps = n\_steps\_per\_sample \cdot \omega_o.ct)$
      Reset new data counter: $\omega_o.ct \leftarrow 0$
    end if
  end for
end for
Retrieve π_Ω from the mutable execution states: $\pi_\Omega \leftarrow \{\omega_o.\pi\}_{o \in \Omega}$
Retrieve π_g from mutable target execution state: $\pi_g \leftarrow \omega_{target}.\pi$
return $\pi_g, \pi_\Omega, D_\Omega$

Algorithm 4 Helper function $ExecuteOneStep$ used in Stable Hierarchical DQN

```
function ExecuteOneStep(s, ω, Ω, ω_Ω)
    # keep recursing until we hit the leaf node of the execution trace
    if ω.child ≠ None then
        return ExecuteOneStep(s, ω.child, Ω)
    end if

    Sample a child option to execute: a ← ω.π(a | s)
    Set child start state: ω.u ← s
    Increment the number of sub-options ω has executed: ω.t ← ω.t + 1

    # recurse if a is indeed a sub-option, not a primitive action
    if a ∈ Ω then
        Make sub-option a (along with its execution state) the active child: ω.child ← ω_a
        return ExecuteOneStep(s, ω.child, Ω)
    end if

    # return a if it is a primitive action
    return a
end function
```

Algorithm 5 Helper function $ReceiveObsOneStep$ used in Stable Hierarchical DQN

function $ReceiveObsOneStep(s', \omega, \Omega, D_\Omega, n, \delta)$
    # keep recursing until we hit the leaf node of the execution trace
    if $\omega.child \neq None$ **then**
        $(is\_child\_done, D_\Omega) \leftarrow ReceiveObsOneStep(s', \omega.child)$
    else
        $is\_child\_done \leftarrow True$
    end if

    # Add data to episode buffer if the child option is done
    if is_child_done **then**
        Get child start state: $s \leftarrow \omega.u$
        Get child option: $a \leftarrow \omega.child.o$
        Calculate reward: $r \leftarrow R_{\omega.g}(s, a, s')$
        Add datapoint to $\omega.D \leftarrow \omega.D \cup \{(s, a, s', r)\}$
        Child is no longer active: $\omega.child \leftarrow None$
    end if

    Check if the current option is done: $is\_cur\_done \leftarrow (\omega.g(s') = 1)$ or $(\omega.t > max\_t)$

    # Update execution state and potentially update the option's replay buffer if the option is done executing
    if is_cur_done **then**

        # Check if all executed sub-options in the episode are stable
        $only\_stable\_in\_episode \leftarrow True$
        for $(\_, a, \_, \_) \in \omega.D$ **do**
            # a is not stable if a is a sub-option
            # AND has lower number of samples it has been trained with than n and lower goal completion rate than δ
            if $a \in \Omega$ and $n_a < n$ and $\delta_a < \delta$ **then**
                $only\_stable\_in\_episode \leftarrow False$
            end if
        end for

        # Add episode buffer to the full replay buffer if all executed sub-options in the episode are stable
        if $only\_stable\_in\_episode$ **then**
            Get option index: $o \leftarrow \omega.o$
            Add episode data to the corresponding replay buffer: $D_o \leftarrow D_o \cup \omega.D$
            Add number of new data to the counter: $\omega.ct \leftarrow \omega.ct + |\omega.D|$
            Retrieve current epsilon: $(\pi^{real}, \pi^{wm}, \epsilon) \leftarrow \omega.\pi$
            Anneal ϵ based on the number of new data: $\epsilon \leftarrow Anneal(\epsilon, |\omega.D|)$
            Update the policy with the new ϵ: $\omega.\pi \leftarrow (\pi^{real}, \pi^{wm}, \epsilon)$
        end if

        # Update execution state
        Child is no longer active: $\omega.child \leftarrow None$
        Clear episode buffer: $\omega.D \leftarrow \{\}$
        Execution time resets: $\omega.t \leftarrow 0$
    end if

    return is_cur_done, D_Ω
end function

Hyperparameter	Value
Number of parallel environments	4
Batch size	256
Learning rate	0.0001
Replay buffer size	$5000 \cdot 4$
Multi-step return	10
Discount factor γ	0.99
Priority replay temperature	0.01
Target network update interval	200
Number of gradient steps per training step	64
Training frequency	$16 \cdot 4$
Reward multiplier κ	10
Maximum gradient norm	10
Random exploration rate ϵ	0
Frame stacking	4
Optimizer	Kron (Castanyer et al., 2025b; Li, 2017)

Table 2. Common training hyperparameters for Rainbow DQN.

Hyperparameter	Value
Max option executime time max_t	100
Number of gradient steps per samples collected $n\_steps\_per\_sample$	1
ϵ Annealing schedule	$Linear(start = 1.0, end = 0, n\_samples = 10,000)$
$n_{threshold}$	20000
$\delta_{threshold}$	0.5

Table 3. Training hyperparameters that are specific to Hierarchical DQN.

Hyperparameter	Value
Multi-step return	1
Random exploration rate ϵ	$Linear(start = 1.0, end = 0.01, fraction = 0.95)$

Table 4. Training hyperparameters for Rainbow DQN that are specific to π^{wm}.

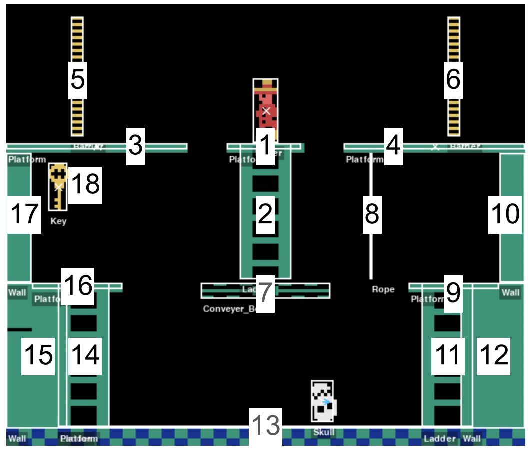

Figure 7. Screenshots of Montezuma's Revenge with goals labeled in order it appears in the ordered list of goals

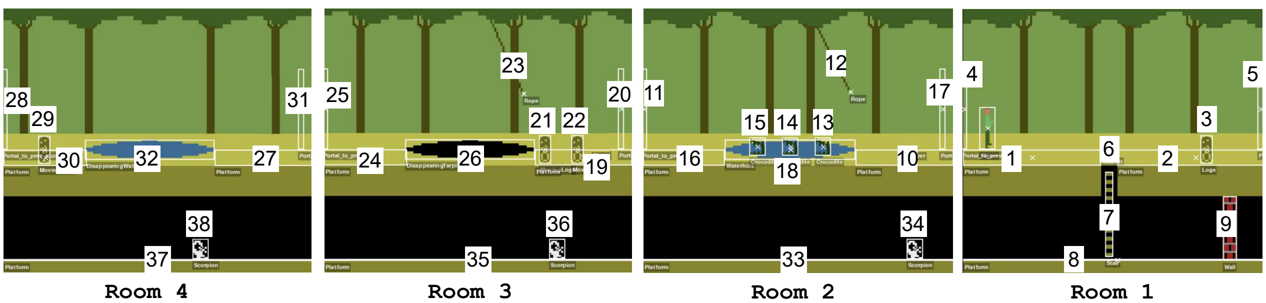

Figure 8. Screenshots of Pitfall with goals labeled in order it appears in the ordered list of goals

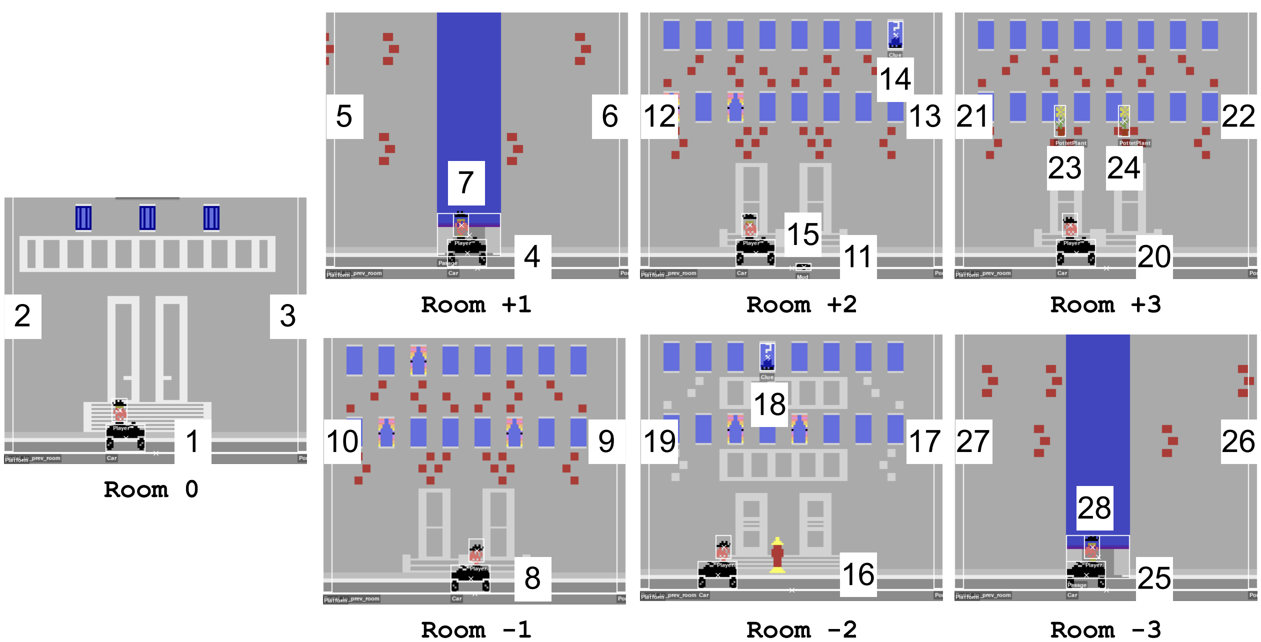

Figure 9. Screenshots of Private Eye with goals labeled in order it appears in the ordered list of goals

```
Here is the current observation of the game {game_name}:

{cur_obs}

And here is a list of goals we know how to achieve:

{achieved_goal_names_and_descriptions}

Your task is to list 1 achieved goals that, on its own, can act as a possible
stepping stone to achieve the target goal of '{target_goal_name}' -- Description:
'{target_goal_description}'.

Required reasoning process:
First, discuss out loud how to achieve the target goal of '{target_goal_name}',
taken into account the current observation.
Then, for each achieved goal, discuss out loud how completing that goal would help
us achieve the target goal of '{target_goal_name}', taken into account the current
observation.
Make sure to go through all achieved goals. But also do not keep repeating the same
achieved goal. The current position of the player is irrelevant.

Final output format:
After you are done reasoning, list the achieved goal in a numbered list with
the following format:
Possible stepping stone 1: <achieved goal>
```

Table 5. Prompt for LLM to propose sub-goals to hypothesize new sub-options. Note that $target\_goal\_description$ is simply "touch object with id obj_id or "in room #$room\_num$, touch object with id obj_id" and $target\_goal\_name$ is something like "bot_plat" or "rope_in_roomnumber+2". cur_obs is a state sampled from D_{seen} if there is a single room like Montezuma's Revenge and multiple states, one per room, for Pitfall and Private Eye.

```
I'll give you an input list of objects.

I want you to list 4 possible features that the input list of objects has that allows us
to achieve a certain goal.

Here's an example:

Example input list of objects:
player object with at (x=16, y=104, w=8, h=21),
wall object with at (x=136, y=148, w=7, h=32),
logs object with at (x=125, y=118, w=6, h=14),
stairpit object with at (x=76, y=122, w=8, h=6),
stair object with at (x=78, y=136, w=4, h=42),
platform object with at (x=8, y=179, w=152, h=1),
platform object with at (x=8, y=125, w=152, h=1),
playerscore object with at (x=38, y=9, w=30, h=8),
lifecount object with at (x=23, y=22, w=1, h=8),
lifecount object with at (x=21, y=22, w=1, h=8),
timer object with at (x=31, y=22, w=37, h=8),
portal_0 object with at (x=7, y=85, w=1, h=40),
portal_1 object with at (x=155, y=85, w=1, h=40),
Interaction -- player object with at (x=16, y=104, w=8, h=21) is touching platform object
with at (x=8, y=125, w=152, h=1)

Example possible features that allow us to achieve the goal of '{goal}':
1. AnyObjTypeTouching: The player object touches a platform object
2. SpecificObjTouching: The player object touches the platform object located at (x=8,
y=125)
3. AnyObjTypeTouching: ...
4. SpecificObjTouching: ...

Now, I want you to list 4 possible features of the input list of objects has that allows
us to achieve the goal of '{goal}'.

Input list of objects:
{input}

Please follow these rules for your output:
1. Do not explain -- simply list each feature
2. Make the features diverse
3. Do use interactions (what the player is touching), as they usually make good features
4. Each rule should of type 'AnyObjTypeTouching' or 'SpecificObjTouching'
```

Table 6. Prompt for LLM to propose preconditions for games where the agent controls only the Player object: Montezuma's Revenge and Pitfall.

```
I'll give you an input list of objects.

I want you to list 4 possible features that the input list of objects has that allows us
to achieve a certain goal.

Here's an example:

Example input list of objects:
player object (x=30, y=150, w=8, h=12),
car object (x=27, y=163, w=20, h=14),
score object (x=75, y=8, w=30, h=8),
clock object (x=67, y=19, w=30, h=8),
roomnumber_+0 object (x=0, y=0, w=0, h=0),
portal_to_prev_room object (x=8, y=27, w=5, h=150),
portal_to_next_room object (x=155, y=27, w=5, h=150),
platform object (x=8, y=177, w=152, h=1),

Example possible features that allow us to achieve the goal of '{goal}':
1. RoomNumberExist: An object with type 'roomnumber_+0' exists
2. ObjTouchingAndRoomNumberExist: The car object touches the platform object and an object
with type 'roomnumber_+0' exists

Now, I want you to list 2 possible features of the input list of objects has that allows
us to achieve the goal of '{goal}'.

Input list of objects:
{input}

Please follow these rules for your output:
1. Do not explain -- simply list each feature
2. Each rule should of type 'RoomNumberExist' or 'ObjTouchingAndRoomNumberExist'
3. Make sure to mention the roomnumber in the feature, e.g., 'an object with type
'roomnumber_+0' exists'
```

Table 7. Prompt for LLM to propose preconditions for games where the agent controls several objects: Private Eye (the agent controls Player and Car object)