ACCELERATING TASK GENERALISATION WITH MULTI-LEVEL HIERARCHICAL OPTIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

Creating reinforcement learning agents that generalise effectively to new tasks is a key challenge in AI research. This paper introduces Fracture Cluster Options (FraCOs), a multi-level hierarchical reinforcement learning method that achieves state-of-the-art performance on difficult generalisation tasks. FraCOs identifies patterns in agent behaviour and forms options based on the expected future usefulness of those patterns, enabling rapid adaptation to new tasks. In tabular settings, FraCOs demonstrates effective transfer and improves performance as it grows in hierarchical depth. We evaluate FraCOs against state-of-the-art deep reinforcement learning algorithms in several complex procedurally generated environments. Our results show that FraCOs achieves higher in-distribution and out-of-distribution performance than competitors.

1 INTRODUCTION

A key goal of AI research is to develop agents that can leverage structured prior knowledge, either provided or learned, to perform competently in unfamiliar domains (Pateria et al., 2021). This is a common feature in animals; for example, many newborn mammals, such as foals, can walk shortly after birth due to innate motor patterns, while human infants display instinctive stepping motions when supported (Adolph & Robinson, 2013; Dominici et al., 2011). These innate behaviors, shaped by evolution, act as priors that guide goal-directed actions and enable rapid adaptation.

In parallel, humans are believed to organise behaviors into a hierarchy of temporally extended actions, which helps break complex tasks into simpler, manageable steps (Rosenbloom & Newell, 1986; Laird et al., 1987). For instance, human decision-making often involves planning with high-level actions like "pick up glass" or "drive to college," each of which comprises subtasks such as "reach for glass" or "pull door handle." These eventually decompose into basic motor movements.

This hierarchical fragmentation likely arises from sub-experiences of past tasks (Brunskill & Li, 2014). Notably, parts of this hierarchy are shared between tasks; for instance, both "pick up glass" and "pull door handle" involve similar gripping movements. Such shared temporal actions may facilitate rapid learning of new tasks beyond those previously experienced. Replicating this hierarchy in algorithms could allow artificial agents to also adapt quickly (Heess et al., 2016).

Despite advances in hierarchical methods, generalising behaviors across diverse tasks remains a significant challenge for artificial agents (Cobbe et al., 2019). Many approaches struggle with effectively transferring skills to new environments, limiting their ability to adapt to real-world scenarios (Pateria et al., 2021).

In this paper, we make two key contributions:

1. We introduce Fracture Cluster Options (FraCOs), a novel framework for defining, forming, and utilizing multi-level hierarchical options based on their expected future usefulness.
2. We empirically demonstrate that FraCOs significantly enhances out-of-distribution (OOD) learning. Our method outperforms two baselines—Proximal Policy Optimization (PPO) (Schulman et al., 2017) and Option Critic with PPO (OC-PPO) (Klissarov et al., 2017)—in both in-distribution and OOD learning across nine environments from the Procgen benchmark (Cobbe et al., 2020).

2 BACKGROUND

2.1 REINFORCEMENT LEARNING

Standard Reinforcement Learning (RL) focuses on how agents can "take actions in different states of an environment to maximize cumulative future reward", where the reward provides task-specific feedback (Sutton & Barto, 2018). In any environment, an agent exists in a state s and can perform an action a, both of which may be discrete, continuous, or multidimensional. Most RL problems are framed as a Markov Decision Process (MDP). An MDP is defined as a tuple $\langle S, A, P, R, \gamma \rangle$, where S is the set of possible states, A is the set of possible actions, P is the transition probability function with $P(s, a, s')$ indicating the probability of transitioning from state s to s' after action a, R is the reward function where $R(s, a, s')$ gives the reward for transitioning from s to s' via action a, and $\gamma \in [0, 1]$ is the discount factor. An MDP assumes the Markov property, where the future state and reward depend only on the current state and action.

At each time step $t \geq 0$, the agent makes a decision based on its current state s_t using a policy, denoted as $\pi(s_t)$. The policy maps states to probabilities over actions, guiding the agent's behaviour. The actions taken produce observed experience data of the form $(s_t, a_t, r_{t+1}, s_{t+1})$. When the agent interacts with the environment until it reaches a termination state, the full sequence of observations is called a trajectory T. The objective of reinforcement learning is to learn a policy that maximises the cumulative discounted returns, defined as $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$.

In this paper, we define an environment as the external system with which the agent interacts, characterized by $\langle S, A, P, R \rangle$. In our work, a task is defined as an MDP with a unique state space S or reward function R.

2.2 HIERARCHICAL REINFORCEMENT LEARNING

Hierarchical Reinforcement Learning (HRL) extends standard RL by organising decision-making into multiple levels of abstraction. A key paradigm in HRL is the *options* framework, which encapsulates extended sequences of actions into *options* (Sutton & Barto, 2018). An option consists of an initiation set I, which defines the states where it can be selected, an intra-option policy π_{intra}, which governs the actions while the option is active, and a termination condition $\beta : S \rightarrow [0, 1]$, which defines when the option ends. The intra-option policy executes actions until the termination condition $\beta(s)$ is satisfied.

Options operate within a Semi-Markov Decision Process (SMDP). An SMDP is an extension of an MDP where actions can have variable durations. The agent chooses between options and primitive actions at each decision point, with a policy over options, π_{opt}, deciding which to execute based on the current state. This hierarchical structuring can improve exploration and learning efficiency by enabling agents to plan over extended time horizons and break complex tasks into manageable sub-tasks. This decomposition reduces the decision space, facilitates structured exploration, and simplifies credit assignment, particularly in environments with sparse rewards (Sutton et al., 1999; Dayan & Hinton, 1992).

2.3 GENERALISATION

Generalisation in Reinforcement Learning (RL) encompasses a broad class of problems (Kirk et al., 2021). These problems can be categorised based on the relationship between training and testing distributions, either falling within Independent and Identically Distributed (IID) scenarios or extending to Out-of-Distribution (OOD) contexts. Additionally, generalisation can be classified by the features of the environment that change, including the state space, observation space, dynamics, and rewards. This classification leads to eight possible combinations of generalisation challenges.

In this work, we define IID generalisation as the ability of agents to generalise within tasks drawn from the same distribution as their training tasks. In contrast, OOD generalisation refers to an agent's ability in tasks that differ from those encountered during training.

We evaluate FraCOs' generalisation performance in OOD tasks where the state space S and reward function R vary, while the action space A, transition dynamics P, and discount factor γ remain

constant. This setup reflects real-world scenarios where an agent, such as a robot, operates under fixed dynamics but faces diverse environments and goals.

3 Related Work

Policy transfer and single-level option transfer methods, aim to broaden an agent's task-handling capabilities. Examples of policy transfer include works by Finn et al. (2017), Grant et al. (2018), Frans et al. (2017), Cobbe et al. (2021), and Mazoure et al. (2022). Option transfer methods, on the other hand, focus on learning a set of reusable options to enhance rewards in new situations (Konidaris & Barto, 2007; Barreto et al., 2019; Tessler et al., 2017; Mann & Choe, 2013). While a few methods in the literature explore *multi-level* hierarchies, they do not focus on option transfer as a mechanism for accelerating task adaptation and generalisation (Riemer et al., 2018; Evans & Şimşek, 2023; Levy et al., 2017; Fox et al., 2017).

HRL has typically focused on addressing broader challenges such as long-term credit assignment and structured exploration (Dayan & Hinton, 1992; Parr & Russell, 1997; McGovern & Sutton, 1998; Sutton et al., 1999). Two foundational paradigms within HRL are: **1)** sub-goal-based approaches, which typically decompose tasks into smaller, state-based intermediate objectives, and **2)** the options framework, which formalises temporally extended actions as options (Sutton et al., 1999). In both paradigms the ability to learn transferable abstractions at more than two levels of hierarchy is still an open research question (Pateria et al., 2021).

Recent work has proposed methods for forming and managing multi-level hierarchies. Levy et al. (2017) and Evans & Şimşek (2023) introduce sub-goal-conditioned approaches for multi-level abstraction. However, due to their reliance on state-based-sub-goals, these methods face difficulties in sub-goal selection in complex state spaces such as pixel-based representations. Moreover, all state-based-sub-goal methods struggle to transfer sub-goals to different state spaces. Additionally, they do not account for the variability in action sequences required to transition between sub-goals; for example, "booking a holiday" could involve "calling a travel agent" or "using the internet," each demanding different skills. In contrast, FraCOs avoids creating state-based-sub-goals, providing a more flexible framework for transfer across state spaces.

Our work is more closely related to Hierarchical Option Critic (HOC) by Riemer et al. (2018) and the Discovery of Deep Options (DDO) by Fox et al. (2017), both of which use the options framework. DDO employs an expectation gradient method to construct a hierarchy *top-down* from expert demonstrations. However, DDO does not optimize for generality and it remains unclear how the discovered options perform in unseen tasks. Moreover, the reliance on demonstrations limits the development of more complex abstractions than those demonstrated. In comparison, FraCOs builds *bottom-up*, forming increasingly complex abstractions. FraCOs also selects options based on their expected *usefulness* in future tasks, directly addressing generalisation challenges.

HOC generalises the Option-Critic (OC) framework introduced by Bacon et al. (2017) to multiple hierarchical levels. HOC learns all options simultaneously during training. However, both OC and HOC suffer from option collapse, where either all options converge to the same behaviour or one option is consistently chosen (Harutyunyan et al., 2019). Moreover, OC methods introduce additional complexity to the learning algorithm, which has been shown to slow learning compared to non-hierarchical approaches like PPO (Schulman et al., 2017; Zhang & Whiteson, 2019). Option selection within the FraCOs process naturally prevents option collapse, and has been shown to increase the rate of learning over baselines (see Section 5.3).

4 Fracture Cluster Options

We hypothesise that identifying reoccurring patterns in an agent's behaviour across successful tasks will improve performance on future, unseen tasks. Our method consists of three key stages: **1)** Identifying the underlying reoccurring patterns in an agent's behaviour across multiple tasks, **2)** selecting the most *useful* patterns—those likely to appear in successful trajectories of all possible tasks, and **3)** defining these identified patterns as options for the agent's future use.

4.1 Identifying Patterns in Agent Behaviour

We seek to identify and cluster reoccurring patterns of states and actions in agent behavior. To achieve this, we introduce the concept of **fractures**. A fracture is defined as a state s_t paired with a sequence of subsequent actions. The length of this action sequence is determined by a parameter known as the *chain length* b, which specifies the number of actions following the state s_t. More formally, a fracture is represented as:

$$\phi = (s_t, a_t, a_{t+1}, \ldots, a_{t+b-1}) \tag{1}$$

where $a_t, a_{t+1}, \ldots, a_{t+b-1}$ represent the subsequent actions from s_t. The parameter b controls the temporal extent of the fracture.

We can derive fractures from the trajectories of tasks which an agent has experienced. Consider a trajectory of length n. The set of fractures F derived from this trajectory is defined as:

$$F = \{(s_t, a_t, a_{t+1}, \ldots, a_{t+b-1}) \mid 0 \leq t \leq n - b\}. \tag{2}$$

For a set of trajectories $\mathcal{T}$, we derive fractures from each individual trajectory. The complete set of fractures from all trajectories is denoted by $\Phi = \{F_1, F_2, \ldots, F_{|T|}\}$. Individual trajectories are denoted by τ, with $\tau \in \mathcal{T}$.

We investigate whether fractures capture underlying structure by first identifying them in the Four Rooms environment. Four Rooms is a classic grid-based reinforcement learning environment, consisting of four connected rooms separated by walls with narrow doorways. The agent's objective is to navigate through the rooms to reach a specified goal, receiving a reward for reaching this goal. Four Rooms is depicted in the top left corner of Figure 1, see Appendix A.6 for more detail. In all of our grid-world implementations, the agent can observe only a 7x7 area centered on itself and a scalar indicating the direction of the reward. This is similar to MiniGrid, except that our observations are ego-centric (Chevalier-Boisvert et al., 2023).

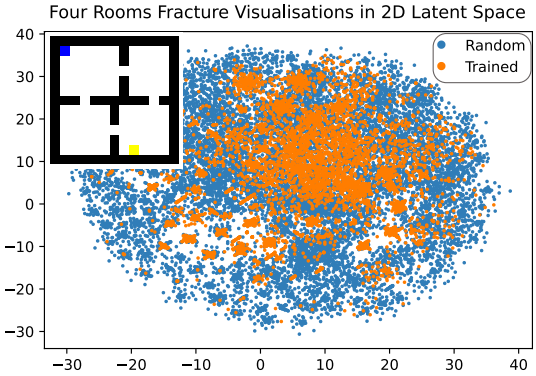


Figure 1: A two-dimensional representation of the fractures ($b = 2$) formed by agents acting for 10,000 steps in the Four Rooms environment.

We train a tabular Q-learning agent to solve multiple different reward locations in Four Rooms and generate trajectories for both the trained agent and a random agent. Fractures are then created following Equation 4.1, with $b = 2$. To reveal the structural differences between the fractures of the random agent and the trained agent, we use UMAP (McInnes et al., 2018), a dimension reduction technique that projects the high-dimensional fracture data into a two-dimensional space. UMAP is particularly useful for this task because it preserves local similarities within the data. We plot the resulting two-dimensional visualization in Figure 1. This highlights a near-uniform distribution for the random agent, while the trained agent's fractures form distinct clusters, reflecting underlying behaviour structures. This phenomenon is consistent across other environments (see Appendix A.7).

We employ unsupervised clustering techniques to identify and formalise the fracture clusters, denoted as ϕ_c. Specifically, we use HDBSCAN for all tabular methods in this work (Campello et al., 2013). Fractures with a chain length of $b = 4$ are grouped into clusters, and four clusters are randomly selected for visualisation in Figure 2. Each visualisation shows *all* fractures within a cluster, demonstrating that, despite differences in starting states, action sequences, and terminal states, the fractures within each cluster share similar semantic meanings.

4.2 Selecting Useful Fracture Clusters

In Section 4.1 fracture clusters are formed based on behaviour similarity; however, the number of potential clusters can be extensive, with some being highly task-specific. Selecting all clusters as

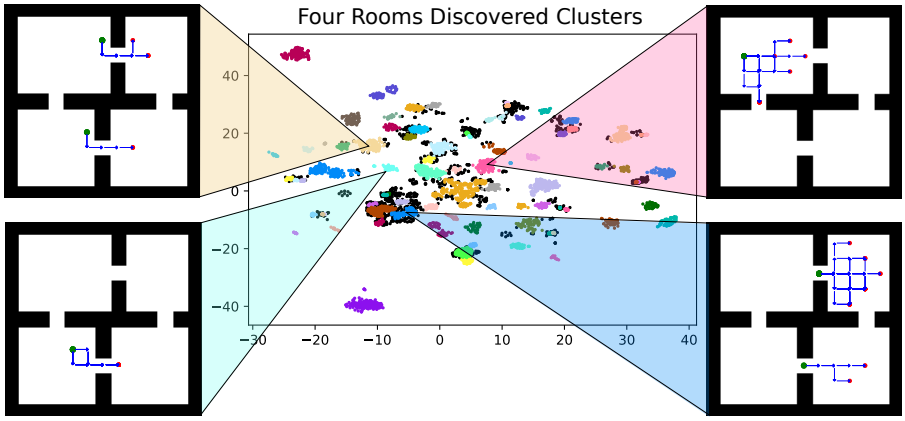


Figure 2: Four examples of discovered fracture clusters ($b = 4$) within the Four Rooms environment. The graph represents fracture clusters of a trained agent in the Four Rooms environment, colours are used to visualise cluster boundaries. In the four examples, the green point represents possible starting states, blue arrows indicate actions, the width of the arrows shows the frequency of the state-action pair within the cluster, and the red point indicates possible terminal states.

options may burden the agent with unnecessary choices. Therefore, it is essential to identify the most useful clusters—those likely to appear in future tasks.

First, consider the hypothetical scenario in which we can observe all possible trajectories across all possible tasks. In this ideal setting, the set of all successful fractures, denoted as Φ_s, would be derived from fractures within the successful trajectories, where each trajectory is represented as $\tau_s \in T_s$. Here, T_s refers to the collection of all trajectories deemed successful. A trajectory is considered successful if its cumulative return exceeds a predefined threshold, similar to the criterion proposed by Chollet (2019), see Table 12 for all minimum returns. The tasks corresponding to these successful trajectories are denoted as $x_s \in \mathcal{X}_s$, where $\mathcal{X}_s$ represents the set of all tasks with successful outcomes.

To sensibly select fracture clusters, we must evaluate their potential for reuse in future tasks. We do this by defining the *usefulness* U of a fracture cluster ϕ_c based on its likelihood of contributing to success across tasks. Specifically, usefulness is determined by three key factors:

1. **Appearance Probability** ($P[\phi_c \in \tau_s \mid x_s]$): This measures the likelihood that any fracture $\phi \in \phi_c$ appears in the trajectory τ_s of any given successful task x_s. Higher probability indicates that this ϕ_c frequently contributes to success across tasks.
2. **Relative Frequency** ($P[\phi_c \mid \Phi_s]$): This term represents the proportion of times that any fracture $\phi \in \phi_c$ appears among all successful fractures Φ_s. A higher relative frequency implies its importance in the agent's overall success.
3. **Entropy of Usage** ($H(\phi_c \mid \mathcal{X}_s)$): This captures the diversity of a fracture cluster's usage across different tasks in $\mathcal{X}_s$. A higher entropy indicates that a ϕ_c is useful across various tasks, enhancing its generalisation potential.

The usefulness of a fracture cluster ϕ_c is defined as the normalised sum of these factors:

$$U_{(\phi_c)} = \frac{1}{3}\left(P[\phi_c \in \tau_s \mid x_s] + P[\phi_c \mid \Phi_s] + H(\phi_c \mid \mathcal{X}_s)\right). \tag{3}$$

In the ideal scenario where we could observe all possible tasks and trajectories, we would directly calculate the usefulness of each fracture cluster $U_{(\phi_c)}$. This would allow us to exactly compute the appearance probability, relative frequency, and entropy for each fracture cluster ϕ_c, yielding a true measure of usefulness across all potential tasks. However, this is impractical since we cannot observe all possible tasks and trajectories. Instead, we must rely on available data, using the tasks and trajectories encountered during training to estimate the usefulness.

1. Estimating Appearance Probability: We approximate $P[\phi_c \in \tau_s \mid x_s]$ using a Bayesian approach, modeling the occurrence of a fracture cluster ϕ_c in a successful trajectory as a binomial

likelihood with a Beta conjugate prior. The prior parameters α and β, both set to 1, reflect an uninformative prior. Let n denote an individual task, and N the total number of experienced tasks. The appearance indicator $\omega_n = 1$ if any fracture $\phi \in \phi_c$ appears in trajectory τ_n, and 0 otherwise.

2. Estimating Relative Frequency: We estimate $P[\phi_c \mid \Phi_s]$ by counting the occurrences of ϕ_c in Φ_s, where Φ_s is formed from N experienced tasks. This count is then normalised by the total number of fractures in Φ_s.

3. Estimating Entropy: Finally, the entropy $H(\phi_c \mid \mathcal{X}_s)$, is approximated using Shannon's entropy formulation (Shannon, 1948).

We derive the full approximation in Appendix A.1, the result is expressed as the *expected usefulness*,

$$E[U_{(\phi_c)}] = \frac{1}{3}\left(\underbrace{\frac{\sum_{n=1}^{N}\omega_n + \alpha}{N+\alpha+\beta}}_{\text{(1) Appearance Probability}} + \underbrace{\frac{\text{count}(\phi_c, \Phi_s)}{|\Phi_s|}}_{\text{(2) Relative Frequency}} - \underbrace{\sum_{\tau_s \in T_s} \frac{\text{count}(\phi_c, \tau_s)}{|\tau_s|} \log_{N_{\phi_c}}\left(\frac{\text{count}(\phi_c, \tau_s)}{|\tau_s|}\right)}_{\text{(3) Estimated Entropy}}\right). \tag{4}$$

We can then select fracture clusters which yield the highest expected usefulness. To demonstrate the effectiveness of Equation 4 we formulate a new experiment. In the Nine Rooms environment (see Appendix A.6) 100 tabular Q-learning agents are trained separately on 100 different tasks, each defined with a new reward function. Evaluation trajectories are gathered and fractures are formed (with a chain length $b = 4$). The fractures are clustered and finally the expected usefulness is calculated. In Figure 3 we plot the eight fracture clusters with the highest expected usefulness.

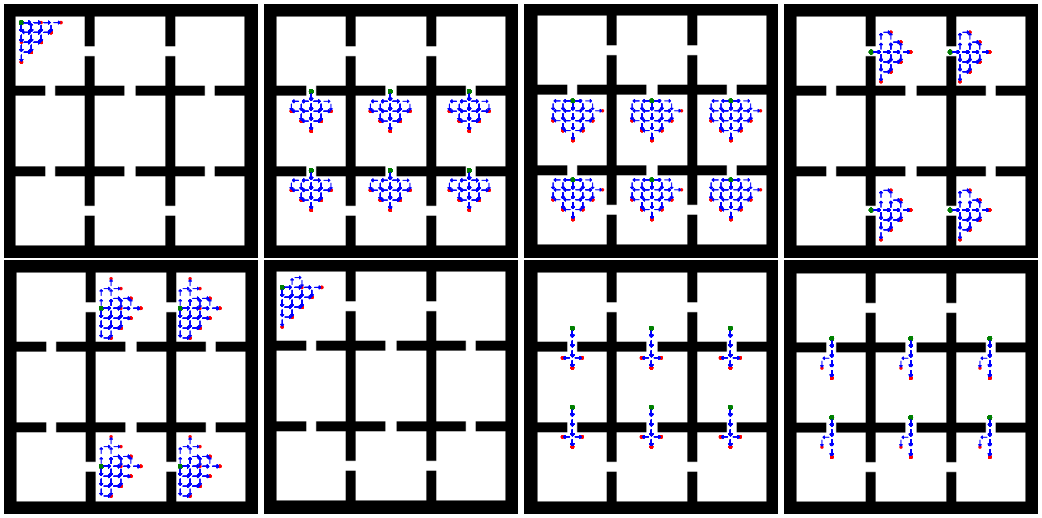


Figure 3: The eight fracture clusters with the highest expected usefulness in the Nine Rooms environment, ordered right and then down. Green points represent possible starting states, blue arrows indicate actions taken, with arrow width proportional to their frequency in the cluster, and red points denote possible termination states.

The fracture cluster in Figure 3 with the highest expected usefulness takes the agent from the starting state in all sensible directions without repetitions of movements. The majority of the other fracture clusters transverse bottlenecks, sharing the same fracture cluster where areas of local structure remain similar.

Forming multiple levels of the hierarchy. After identifying the most useful fracture clusters, they can be converted into options (as explained in Section 4.3), which extend the agent's action space. When learning a new task, the agent can now choose from both primitive actions and these newly discovered options. The process of identifying fractures and clustering is repeated, but now the trajectories (and therefore fractures) may consist of a mix of primitive actions and higher-level options. This iterative approach naturally leads to the creation of a multi-level hierarchical structure.

4.3 USING FRACTURE CLUSTERS

After selecting the most useful fracture clusters, we need to transform each cluster into an option, called a **Fracture Cluster Option (FraCO)**. Each FraCO is characterized by an initiation set I_z, a termination condition β_z, and a policy π_z. We denote a single FraCO as z and the set of all FraCOs as Z. Each FraCO is associated with the fracture cluster ϕ_c that forms it. We use these clusters to predict initiation states.

Initiation Set. Suppose the agent is in a state s, and has access to a set of actions A. For each state s, we consider possible sequences of actions of chain length b. Each such sequence is represented as:

$$\mathbf{a} = (a_1, a_2, \ldots, a_b) \mid a_i \in A, \tag{5}$$

We can now define the set of all possible fractures starting from state s as,

$$F_s = \{(s, \mathbf{a}) \mid \mathbf{a} \in A^b\}, \tag{6}$$

where A^b represents all sequences of length b drawn from the action set A.

For each fracture $\phi = (s, \mathbf{a})$ in F_s, we estimate the likelihood that it belongs to a FraCO z. The set of fractures assigned to cluster z in state s can be defined as:

$$G_{z,s} = \{\phi \in F_s \mid P(\phi \in z) > \theta\},$$

where $P(\phi \in z)$ denotes the estimated probability that fracture ϕ belongs to cluster z, and θ is a threshold hyperparameter that determines cluster membership. The method for estimating $P(\phi \in z)$ can vary depending on the implementation. In our tabular implementation, we directly use the prediction function provided by HDBSCAN. In our deep implementation, a neural network predicts this probability.

A FraCO z can be initiated in state s if $G_{z,s}$ is not empty. Such that the initiation set is defined as,

$$I_z = \{s \in S \mid G_{z,s} \neq \emptyset\}. \tag{7}$$

Policy Execution. When FraCO z is executed in state s, it follows the policy π_z, as described in Algorithm 1. The policy selects the fracture $\phi_z = (s, \mathbf{a})$ from $G_{z,s}$ with the highest probability $P(\phi_z \in z)$, and then executes the sequence of actions $\mathbf{a} = (a_1, a_2, \ldots, a_b)$. If one of the selected actions is another FraCO z', the agent must compute a new $G_{z',s}$ and recursively call the policy until the option terminates.

Termination Condition. The FraCO z terminates under two conditions: either when all actions in the selected fracture ϕ_z have been executed, or when no matching fracture can be found in the current state (i.e., $G_{z,s} = \emptyset$).

Algorithm 1: Option policy $\pi_z(G_{z,s}, Z, s)$

Input: $G_{z,s}$, set of options Z, state s
Initialize: Select the fracture $\phi_z = (s, \mathbf{a})$ from $G_{z,s}$ with the highest probability $P(\phi_z \in z)$, where $\mathbf{a}$ is a sequence of actions $(a_1, a_2, \ldots, a_b)$
for *each action* a_j *in* $\mathbf{a}$ **do**
  if a_j *is a primitive action* **then**
    Execute a_j, resulting in new state s;
  else
    a_j is another option z';
    Compute $G_{z',s}$;
    Recursively call $\pi_{z'}(G_{z',s}, Z, s)$;
  end
end

The termination condition $\beta(s)$ is defined as:

$$\beta(s) = \begin{cases} 1 & \text{if all actions in } \phi_z \text{ have been executed} \\ 1 & \text{if } G_{z,s} = \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

Learning with FraCOs. In our approach, FraCOs are fixed once identified. The agent learns to choose between primitive actions and available FraCOs using standard reinforcement learning algorithms (e.g., Q-learning for tabular settings, PPO for deep learning implementations).

Example FraCO usage: An agent is in state s. It looks at all possible fractures in that state to form F_s. For each FraCO $z_i \in Z$, the agent estimates the probability $P(\phi \in z_i)$ that each fracture $\phi \in F_s$ belongs to z_i. If any fractures have a probability above a threshold θ, then z_i becomes available as an option.

The agent's policy π_{opt} selects from the available FraCOs and primitive actions. Suppose it selects FraCO z_1. The agent picks the fracture ϕ_{z_1} with the highest probability $P(\phi_{z_1} \in z_1)$ and executes the sequence $\mathbf{a}$.

As the agent executes these actions, if one is another FraCO, say z_2, then it forms a new $F_{s'}$, estimates $P(\phi \in z_2)$, and selects the most probable fracture for z_2. This process repeats until the termination condition $\beta_{z_1} = 1$ is met, completing z_1's execution.

5 Experimental Results

We evaluate FraCOs in three different experiments. The first experiment focuses on OOD reward generalisation tasks using a tabular FraCOs agent in the Four Rooms, Nine Rooms, and Romesh Maze grid-world environments (see Appendix A.6). The second experiment examines OOD learning in state generalisation tasks within a novel environment called MetaGrid (see Appendix A.6). The final experiment evaluates a deep FraCOs agent, implemented with a three-level hierarchy using PPO, in nine Procgen environments (Cobbe et al., 2020). We compare its performance with CleanRL's Procgen PPO implementation (Huang et al., 2022) and Option Critic with PPO (OC-PPO) (Klissarov et al., 2017). We use the same hyperparameters from CleanRL's PPO implementation for all three algorithms. In the grid-world environments, the agent receives a reward of 1 for reaching the goal and a penalty of -0.001 per time step. Episodes have a maximum length of 500 steps, and a fracture chain length $b = 2$. For Procgen environments, reward functions remain unchanged, and $b = 3$ (see Appendix A.8 for chain-length selection details).

5.1 Experiment 1: Tabular Reward Generalisation

In this experiment, FraCOs are learned in a fixed state space S while we vary the reward function R. The agent is allowed to discover FraCOs in 50 tasks, such that each task corresponds to a different reward location. We reserve 10 unique tasks for testing. Separate agents are trained to convergence—defined as achieving consistent performance across episodes. The top 20 FraCOs are extracted from the final trajectories of the agents, as described in Section 4, and incorporated into the action space. This extraction process is repeated four times, corresponding to each level of the hierarchy.

Four agents are created, each with an additional hierarchical level of FraCOs. After resetting their policies over options, the agents are trained on the test tasks, and evaluation episodes are conducted periodically. As shown in Figure 4, results from the Four Rooms, Nine Rooms, and Romesh Maze environments indicate that learning is progressively accelerated on these unseen tasks as the hierarchy depth increases.

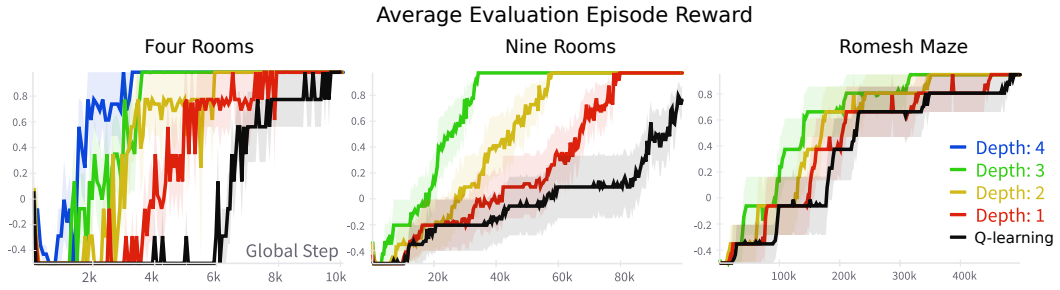


Figure 4: Evaluation episode rewards of a tabular FraCOs agent trained in the Four Rooms, Grid, and Romesh Maze environments. Results are averaged over 10 independently seeded experiments, with shaded areas indicating the standard error.

5.2 Experiment 2: Tabular State Generalisation

In this experiment, we introduce a novel environment called MetaGrid, which is designed to test state and reward generalisation. MetaGrid is a navigational grid world constructed from structured 7×7 building blocks that can be combined randomly to create novel state spaces while preserving certain areas of local structure. The agent is provided with a 7×7 window of observation, consistent with our other grid-worlds. For more detailed information on MetaGrid, see Appendix A.6.

At each hierarchy level, 20 FraCOs are learned from 100 randomly generated 14×14 MetaGrid tasks. A task in this experiment corresponds to a differnt space space S and reward location R. For each hierarchy level, a separate agent is created, and their policies over options are reset. These agents are then evaluated in previously unseen 14×14 domains and larger 21×21 domains. Periodic evaluation episodes are conducted during training to track performance, results are shown in Figure 5.

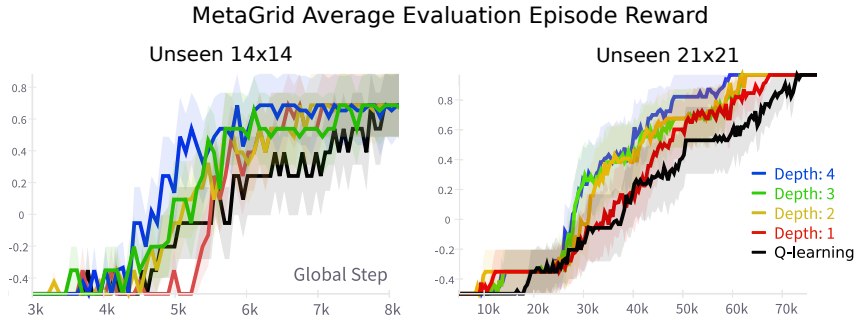


Figure 5: Evaluation episode rewards for tabular FraCOs in unseen MetaGrid domains of varying sizes. Results are averaged over 10 independently seeded experiments, with shaded areas indicating the standard error.

5.3 Experiment 3: Deep State and Reward Generalisation in Complex Environments

In this experiment, we test FraCOs in OOD tasks from the Procgen benchmark, focusing on unseen states spaces S and reward functions R. We compare with two methods: Option Critic with PPO (OC-PPO) (Klissarov et al., 2017) and a baseline PPO (Schulman et al., 2017), across nine Procgen environments (Cobbe et al., 2020). Procgen is a suite of procedurally generated arcade-style environments designed to assess generalisation across diverse tasks; see Appendix A.6 for full details.

FraCOs modifications. To handle the challenges of applying traditional clustering to high-dimensional pixel data, we simplify the approach by grouping fractures with the same action sequences, regardless of state differences. Additionally, a neural network is used to estimate initiation states and policies, which reduces the computational burden of performing a discrete search over the complex $64 \times 64 \times 3$ state space and managing 15 possible actions during millions of training steps. These modifications do not change the theory of FraCOs, just the implementation. Full details of these modifications are provided in Appendix A.10, with further information on the experiments, baselines, and hyperparameters in Appendix A.11.

FraCOs and OC-PPO both learn options during a 20-million time-step warm-up phase, with tasks drawn from the first 100 levels of each Procgen environment. FraCOs learns two sets of 25 options, corresponding to different hierarchy levels, while OC-PPO learns a total of 25 options. After the warm-up, the policy over options is reset, and training continues for an additional 5 million time steps. During this phase, we periodically conduct evaluation episodes on both IID and OOD tasks, with OOD tasks drawn from Procgen levels beyond 100.

We test two versions of FraCOs: one with a complete reinitialised policy over options after the warm-up phase, and another that transfers a Shared State Representation (SSR), referred to as FraCOs-SSR. In SSR, shared convolutional layers encode the state, followed by distinct linear layers for the critic, policy over options, and option policies. These convolutional layers are not reset after warm-up. Since OC-PPO shares its state representation, comparing it with FraCOs-SSR offers a

fairer evaluation. However, we find that FraCOs without SSR still outperforms both baselines. See Appendix A.11 for FraCOs-SSR implementation details.

Table 1 summarises the final mean IID, OOD, and peak OOD episodic evaluation returns for each method across all nine environments. PPO-5 refers to PPO trained for 5 million time steps corresponding to the training time after the warm-up phase, while PPO-25 refers to PPO trained for the full 25 million time steps, including the warm-up phase and evaluation period without resetting the policy. Each experiment was repeated with three independent seeds. Figure 6 shows examples of the average IID and OOD evaluation returns as the agent learns post-warm-up in the FruitBot and StarPilot environments. Detailed results for all environments are in Appendix A.13.

	Final IID	**Final OOD**	**Peak OOD**
FraCOs	13.04	8.33	9.18
FraCOs-SSR	16.16	**11.47**	**12.37**
OC-PPO	11.00	7.74	8.47
PPO-5	11.97	6.67	6.84
PPO-25	**17.68**	8.27	9.13

Table 1: Final mean IID, OOD, and Peak OOD episodic evaluation returns for each method averaged across nine Procgen environments.

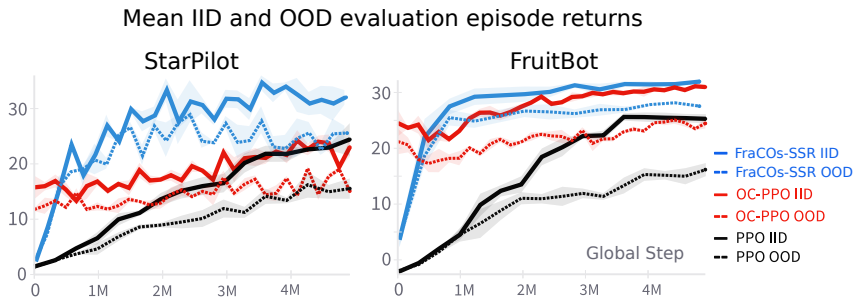


Figure 6: Evaluation episode returns post-warm-up. IID evaluations are conducted on training levels, while OOD evaluations are on levels not seen during training. Results are averaged over 3 independently seeded experiments, with shaded areas represent the standard error.

6 Discussion and Limitations

In this study, we introduce Fracture Cluster Options (FraCOs) as a novel approach to multi-level hierarchical reinforcement learning. In the tabular settings, FraCOs demonstrates accelerated learning in unseen tasks. Notably, as the hierarchical depth of FraCOs increases, the agent's performance also improves.

Our deep RL experiments show that FraCOs outperforms both OC-PPO and PPO in out-of-distribution (OOD) tasks. Although PPO-25 slightly outperforms FraCOs in-distribution, it receives five times more experience than FraCOs' policy over options. FraCOs outperforms PPO-5 in IID tasks and extends this by leveraging hierarchical options across tasks. This adaptability allows FraCOs to far surpass PPO in OOD scenarios. Furthermore FraCOs consistently outperforms Option Critic, even when paired with PPO, highlighting its potential as a robust framework for hierarchical reinforcement learning.

Despite its contributions, this work faces some limitations. As environments increase in complexity, clustering methods struggle to maintain accurate predictions. While we introduce simpler clustering techniques and neural networks to address this issue, future research could explore alternatives for identifying and predicting fracture clusters. Additionally, our experiments focus exclusively on discrete action spaces, and extending this methodology to continuous action spaces remains a challenge for future investigation. Nonetheless, FraCOs provides a solid basis for continued research into generalisation via multi-level hierarchical reinforcement learning.

REFERENCES

Karen E. Adolph and Scott R. Robinson. 403The Road to Walking: What Learning to Walk Tells Us About Development. In *The Oxford Handbook of Developmental Psychology, Vol. 1: Body and Mind*. Oxford University Press, 03 2013. ISBN 9780199958450. doi: 10.1093/oxfordhb/9780199958450.013.0015. URL https://doi.org/10.1093/oxfordhb/9780199958450.013.0015.

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

André Barreto, Diana Borsa, Shaobo Hou, Gheorghe Comanici, Eser Aygün, Philippe Hamel, Daniel Toyama, Shibl Mourad, David Silver, Doina Precup, et al. The option keyboard: Combining skills in reinforcement learning. *Advances in Neural Information Processing Systems*, 32, 2019.

Emma Brunskill and Lihong Li. Pac-inspired option discovery in lifelong reinforcement learning. In *International conference on machine learning*, pp. 316–324. PMLR, 2014.

Ricardo JGB Campello, Davoud Moulavi, and Jörg Sander. Density-based clustering based on hierarchical density estimates. In *Pacific-Asia conference on knowledge discovery and data mining*, pp. 160–172. Springer, 2013.

Maxime Chevalier-Boisvert, Bolun Dai, Mark Towers, Rodrigo de Lazcano, Lucas Willems, Salem Lahlou, Suman Pal, Pablo Samuel Castro, and Jordan Terry. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. *arXiv preprint arXiv:2306.13831*, 2023.

François Chollet. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019.

Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying generalization in reinforcement learning. In *International conference on machine learning*, pp. 1282–1289. PMLR, 2019.

Karl Cobbe, Chris Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. In *International conference on machine learning*, pp. 2048–2056. PMLR, 2020.

Karl W Cobbe, Jacob Hilton, Oleg Klimov, and John Schulman. Phasic policy gradient. In *International Conference on Machine Learning*, pp. 2020–2027. PMLR, 2021.

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

Nadia Dominici, Yuri P Ivanenko, Germana Cappellini, Andrea d'Avella, Vito Mondì, Marika Cicchese, Adele Fabiano, Tiziana Silei, Ambrogio Di Paolo, Carlo Giannini, et al. Locomotor primitives in newborn babies and their development. *Science*, 334(6058):997–999, 2011.

Joshua B Evans and Özgür Şimşek. Creating multi-level skill hierarchies in reinforcement learning. *arXiv preprint arXiv:2306.09980*, 2023.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pp. 1126–1135. PMLR, 2017.

Roy Fox, Sanjay Krishnan, Ion Stoica, and Ken Goldberg. Multi-level discovery of deep options. *arXiv preprint arXiv:1703.08294*, 2017.

Kevin Frans, Jonathan Ho, Xi Chen, Pieter Abbeel, and John Schulman. Meta learning shared hierarchies. *arXiv preprint arXiv:1710.09767*, 2017.

Erin Grant, Chelsea Finn, Sergey Levine, Trevor Darrell, and Thomas Griffiths. Recasting gradient-based meta-learning as hierarchical bayes. *arXiv preprint arXiv:1801.08930*, 2018.

Anna Harutyunyan, Will Dabney, Diana Borsa, Nicolas Heess, Remi Munos, and Doina Precup. The termination critic. *arXiv preprint arXiv:1902.09996*, 2019.

Nicolas Heess, Greg Wayne, Yuval Tassa, Timothy Lillicrap, Martin Riedmiller, and David Silver. Learning and transfer of modulated locomotor controllers. *arXiv preprint arXiv:1610.05182*, 2016.

Shengyi Huang, Rousslan Fernand Julien Dossa, Chang Ye, Jeff Braga, Dipam Chakraborty, Kinal Mehta, and JoÃĢo GM AraÃšjo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Robert Kirk, Amy Zhang, Edward Grefenstette, and Tim Rocktäschel. A survey of generalisation in deep reinforcement learning. *arXiv preprint arXiv:2111.09794*, 2021.

Martin Klissarov, Pierre-Luc Bacon, Jean Harb, and Doina Precup. Learnings options end-to-end for continuous action tasks. *arXiv preprint arXiv:1712.00004*, 2017.

George Dimitri Konidaris and Andrew G Barto. Building portable options: Skill transfer in reinforcement learning. In *IJCAI*, volume 7, pp. 895–900, 2007.

John E Laird, Allen Newell, and Paul S Rosenbloom. Soar: An architecture for general intelligence. *Artificial intelligence*, 33(1):1–64, 1987.

Andrew Levy, George Konidaris, Robert Platt, and Kate Saenko. Learning multi-level hierarchies with hindsight. *arXiv preprint arXiv:1712.00948*, 2017.

Timothy A Mann and Yoonsuck Choe. Directed exploration in reinforcement learning with transferred knowledge. In *European Workshop on Reinforcement Learning*, pp. 59–76. PMLR, 2013.

Bogdan Mazoure, Ilya Kostrikov, Ofir Nachum, and Jonathan J Tompson. Improving zero-shot generalization in offline reinforcement learning using generalized similarity functions. *Advances in Neural Information Processing Systems*, 35:25088–25101, 2022.

Amy McGovern and Richard S Sutton. Macro-actions in reinforcement learning: An empirical analysis. *Computer Science Department Faculty Publication Series*, pp. 15, 1998.

Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.

Volodymyr Mnih. Asynchronous methods for deep reinforcement learning. *arXiv preprint arXiv:1602.01783*, 2016.

Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. *Advances in neural information processing systems*, 10, 1997.

Shubham Pateria, Budhitama Subagdja, Ah-hwee Tan, and Chai Quek. Hierarchical reinforcement learning: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 54(5):1–35, 2021.

Matthew Riemer, Miao Liu, and Gerald Tesauro. Learning abstract options. *Advances in neural information processing systems*, 31, 2018.

Paul S Rosenbloom and Allen Newell. The chunking of goal hierarchies: A generalized model of practice. *Machine learning-an artificial intelligence approach*, 2:247, 1986.

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

Claude Elwood Shannon. A mathematical theory of communication. *The Bell system technical journal*, 27(3):379–423, 1948.

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

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

Chen Tessler, Shahar Givony, Tom Zahavy, Daniel Mankowitz, and Shie Mannor. A deep hierarchical approach to lifelong learning in minecraft. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. Gymnasium, March 2023. URL `https://zenodo.org/record/8127025`.

Shangtong Zhang and Shimon Whiteson. Dac: The double actor-critic architecture for learning options. *Advances in Neural Information Processing Systems*, 32, 2019.

A Appendix

A.1 Derivation of Usefulness Metric

In this appendix, we derive the usefulness (U) metric for fracture clusters. This metric is used to identify which fracture clusters have the greatest potential for reuse across different tasks. Usefulness is a function of the following three factors:

1. The probability that a fracture cluster appears in any given successful task:
$$P[\phi_c \in \tau_s | x_s]$$
2. The probability that a fracture cluster is selected from the set of successful fracture clusters:
$$P[\phi_c | \Phi_s]$$
3. The entropy of the fracture cluster's usage across all successful tasks:
$$H(\phi_c \mid \mathcal{T}_s)$$

Usefulness (U) is then defined as the normalized sum of these three factors:

$$U = \frac{1}{3}\left(P[\phi_c \in \tau_s | x_s] + P[\phi_c | \Phi_s] + H(\phi_c \mid \mathcal{T}_s)\right)$$

The objective is to select fracture clusters that maximize the usefulness, i.e.,

$$\arg\max_{\phi_c} U(\phi_c)$$

A.2 Deriving $P[\phi_c \in \tau_s | x_s]$ Using Bayesian Inference

We want to model the probability that a fracture cluster ϕ_c appears in any given successful task, $P[\phi_c \in \tau_s | x_s]$. For each successful task x_s from the set of successful tasks $\mathcal{X}_s$, the presence of fracture cluster ϕ_c in the corresponding trajectory τ_s is represented by a binary random variable ω_s, where:

$$\omega_s = \begin{cases} 1 & \text{if } \phi_c \in \tau_s \text{ (fracture cluster appears in the trajectory)}, \\ 0 & \text{otherwise} \end{cases}$$

The variable ω_s is modeled as a Bernoulli random variable:

$$\omega_s \sim \text{Bernoulli}(p)$$

where p is the probability that fracture cluster ϕ_c appears in the trajectory τ_s of task x_s.

Since we are uncertain about the true value of p, we place a Beta distribution prior on p:

$$p \sim \text{Beta}(\alpha, \beta)$$

where α and β are hyperparameters representing our prior belief about the likelihood of ϕ_c appearing in a trajectory.

Given a total of N tasks, the likelihood for each observation ω_s is:

$$P(\omega_n|p) = p^{\omega_n}(1-p)^{1-\omega_n}$$

where ω_n is 1 if ϕ_c appears in trajectory τ_s for task x_n, and 0 otherwise.

Using Bayes' theorem, the posterior distribution of p after observing data is:

$$P(p|\omega_1, \ldots, \omega_N) \propto P(\omega_1, \ldots, \omega_N|p)P(p)$$

Substituting the likelihood and the Beta prior, we get:

$$P(p|\omega_1, \ldots, \omega_N) \propto \prod_{n=1}^{N} p^{\omega_n}(1-p)^{1-\omega_n} \cdot p^{\alpha-1}(1-p)^{\beta-1}$$

This simplifies to:

$$P(p|\omega_1, \ldots, \omega_N) \propto p^{\sum_{n=1}^{N}\omega_n+\alpha-1}(1-p)^{N-\sum_{n=1}^{N}\omega_n+\beta-1}$$

Thus, the posterior distribution for p follows a Beta distribution:

$$p|\omega_1, \ldots, \omega_N \sim \text{Beta}(\alpha_N, \beta_N)$$

where:

$$\alpha_N = \sum_{n=1}^{N}\omega_n + \alpha, \quad \beta_N = N - \sum_{n=1}^{N}\omega_n + \beta$$

In our experiments, we set $\alpha = 1$ and $\beta = 1$, representing an uninformative prior.

A.3 Deriving $P[\phi_c|\Phi_s]$

The second component of the usefulness metric, $P[\phi_c|\Phi_s]$, is the probability that fracture cluster ϕ_c is selected from the set of successful fracture clusters. This can be computed as the relative frequency of ϕ_c in the set Φ_s of successful clusters:

$$P[\phi_c|\Phi_s] = \frac{\text{count}(\phi_c, \Phi_s)}{|\Phi_s|}$$

where $\text{count}(\phi_c, \Phi_s)$ is the number of times ϕ_c appears in the set of successful clusters, and $|\Phi_s|$ is the total number of successful clusters.

A.4 Deriving $H(\phi_c \mid \mathcal{T}_s)$

The entropy term $H(\phi_c \mid \mathcal{T}_s)$ measures the unpredictability or diversity of the usage of fracture cluster ϕ_c across successful tasks. Entropy is defined as:

$$H(\phi_c \mid \mathcal{T}_s) = -\sum_{\tau_s \in \mathcal{T}_s} p[\phi_c \mid \tau_s] \cdot \log_{N_{\phi_c}}(p[\phi_c \mid \tau_s])$$

where $p[\phi_c \mid \tau_s]$ is the proportion of times that fracture cluster ϕ_c appears in trajectory τ_s:

$$p[\phi_c \mid \tau_s] = \frac{\text{count}(\phi_c, \tau_s)}{|\tau_s|}$$

Here, $|\tau_s|$ represents the length of the trajectory τ_s, and N_{ϕ_c} is the total number of fracture clusters considered. The choice of logarithm base, N_{ϕ_c}, reflects the fact that we normalize entropy relative to the number of fracture clusters.

A.5 Expected Usefulness

Having derived the empirical estimations of the three components of usefulness we can now combine these elements to calculate the expected usefulness of each fracture cluster. The expected usefulness incorporates the posterior distribution from Bayesian inference for $P[\phi_c \in \tau_s|x_s]$, as well as the empirical counts for the other components.

Thus, the expected usefulness for each fracture cluster is calculated as:

$$E[U(\phi_c)] = \frac{1}{3}\left(\frac{\sum_{n=1}^{N}\omega_n + \alpha}{N+\alpha+\beta} + \frac{\text{count}(\phi_c, \Phi_s)}{|\Phi_s|} - \sum_{\tau_s \in \mathcal{T}_s} \frac{\text{count}(\phi_c, \tau_s)}{|\tau_s|} \cdot \log_{N_{\phi_c}}\left(\frac{\text{count}(\phi_c, \tau_s)}{|\tau_s|}\right)\right)$$

where α and β are the parameters of the beta distribution, which we set to $\alpha = 1$ and $\beta = 1$ in our experiments.

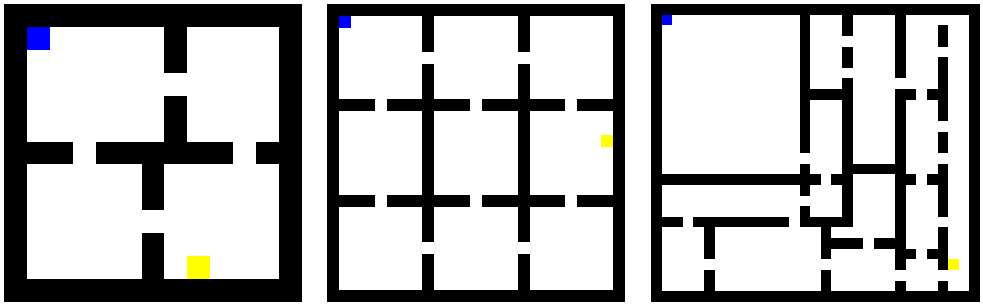
By calculating this expected usefulness, we can rank the fracture clusters according to their potential for reuse in future tasks. The ranking helps focus on fracture clusters that are more likely to appear in successful outcomes and contribute to the agent's performance across diverse scenarios.

A.6 Environments

This section provides details on the environments used in our experiments, including standard grid-world domains (Four Rooms, Grid, Romesh Maze), MetaGrid, and the Procgen suite. Each environment has been designed to evaluate different aspects of the agent's behaviour, such as navigation, exploration, and task performance.

A.6.1 Grid-World Environments

We use three standard grid-world environments: Four Rooms, Grid, and Romesh Maze. Figure 7 illustrates these environments.



(a) Four rooms (b) Grid (c) Romesh maze

Figure 7: Examples of the Four Rooms (a), Grid (b), and Romesh Maze (c) environments. In each, black represents walls, blue represents the agent, yellow represents the goal, and white represents empty space. The agent can move up, down, left, and right, receiving a reward upon reaching the goal. The goal's location in these figures is an example of one of many possible positions. These versions of Four Rooms, Grid and Romesh Maze are part of the MetaGrid suite and thus have a 7x7 observation space centered on the agent.

In these grid-world environments, the action space is discrete, with four possible (primitive) actions:

$$A = \{0, 1, 2, 3\}$$

These actions correspond to moving Up, Down, Left, and Right, respectively. The agent's observation space is a 7x7 grid centered on itself, meaning it only observes a portion of the environment at any given time. This localized view allows the agent to learn how to navigate based on nearby features. This design is similar to MiniGrid (Chevalier-Boisvert et al., 2023), but in our case, the observations are ego-centric, always centered on the agent.

In tabular learning, the agent uses this 7x7 observation space to predict initiation states, though the Q-function is still based on absolute coordinates. We do not employ state-transition graphs in any of our experiments.

A.6.2 MetaGrid Environment

The MetaGrid environment extends the standard grid-world setup by allowing for procedurally generated maps of varying sizes. MetaGrid introduces randomness in the layout of walls and goal

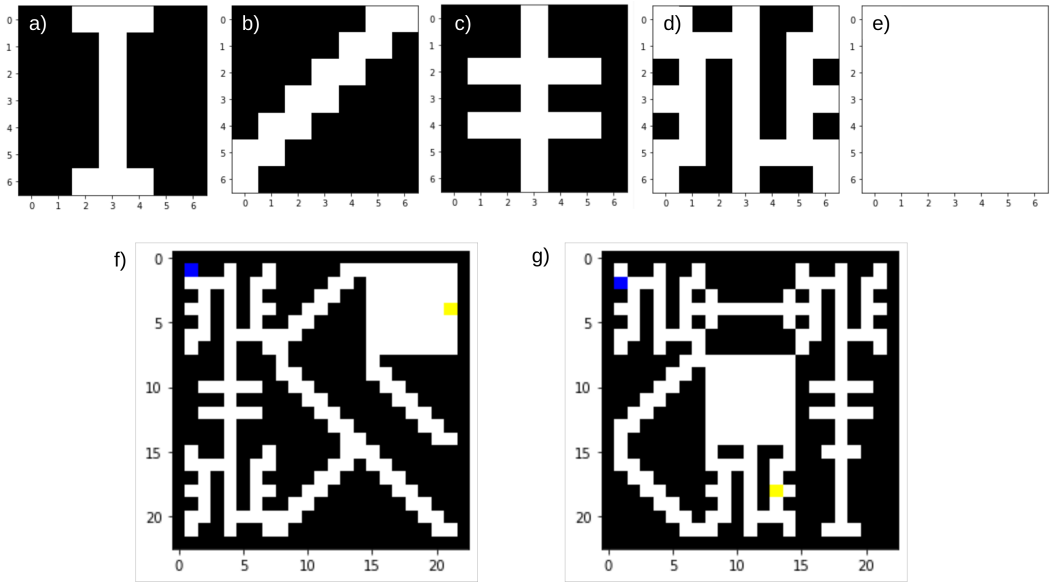
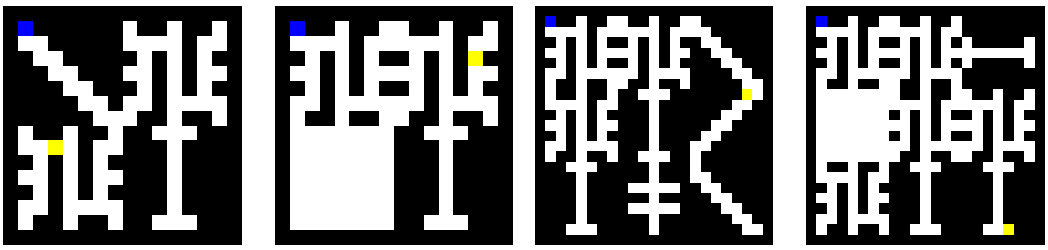


Figure 8: a) - e) demonstrate the building blocks which MetaGrid domains can be created from. f) and g) demonstrate two 21x21 configurations using these building blocks.

locations, ensuring that the agent encounters a diverse set of environments during training and evaluation. Figure 8 demonstrates the building blocks which all environments in MetaGrid are formed from, and Figure 9 shows examples of MetaGrid environments in two different sizes: 14x14 and 21x21.



(a) (b) (c) (d)

Figure 9: Examples of randomly generated MetaGrid environments. The blue square represents the agent, yellow represents the reward, white represents empty space, and black represents walls. Subfigures (a) and (b) show 14x14 grids, while (c) and (d) show 21x21 grids.

The action space in MetaGrid is identical to the one used in the standard grid worlds, with four discrete actions: Up, Down, Left, and Right. Similarly, the agent observes a 7x7 grid centered on itself, allowing it to make decisions based on local information. The procedural generation in MetaGrid provides varied environments for the agent to adapt to, making it a more challenging and dynamic environment compared to static grid worlds.

A.6.3 Procgen Environments

Procgen is a suite of procedurally generated environments designed to test generalisation and performance across diverse tasks such as navigation, exploration, combat, and puzzle-solving. Each environment provides a different variation on every reset, preventing the agent from memorizing specific layouts or solutions. Please see Cobbe et al. (2020) for the full details of these environments.

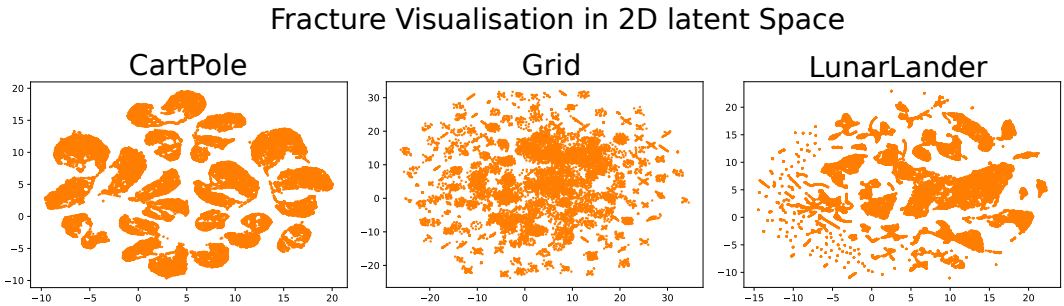


Figure 10: Two dimensional visualisations of the fractures formed for trained agents in CartPole, Grid and LunarLander .

Action Space Procgen environments feature discrete actions like movement (up, down, left, right) and interactions (e.g., jump, shoot). Depending on the task, the action space can range from 5 to 15 actions, covering basic navigation and task-specific interactions.

Observation Space Unlike grid-based environments, Procgen uses 64x64 RGB pixel observations, providing rich visual input. The agent must interpret features such as walls, enemies, and obstacles to navigate and interact effectively.

Rewards Rewards in Procgen are sparse, given for completing tasks like reaching goals or defeating enemies. The agent must learn to explore efficiently and develop strategies for long-term success.

Optional Parameters To simplify learning and reduce computational cost, we activated the following Procgen parameters:

- **Distribution mode = "easy"**: Provides easier levels.
- **Use backgrounds = "False"**: Backgrounds are black to avoid additional noise.
- **Restrict themes = "True"**: Limits visual variation to a single theme, such as consistent wall styles in environments like CoinRun.

Overall, these environments—ranging from grid-world environments with discrete action spaces and ego-centric observations to the more complex Procgen environments with pixel-based observations—offer a diverse set of challenges for our agents. The combination of procedurally generated environments in MetaGrid and Procgen ensures that the agents are tested on both fixed and highly variable environments, making them suitable for evaluating the robustness of the learning algorithms used in our experiments.

A.7 UMAP VISUALISATIONS OF OTHER ENVIRONMENTS

The structure observed in the latent projections of clustered fracos as seen in Section 4.1 is also demonstrated in trained agents of other simple environments; Figure 10 visualises fracture structures in Grid, CartPole, and LunarLander. The Grid environment is shown in Figure 7, CartPole and LunarLander are standard environments from the Farama Foundation Gymnasium suite Towers et al. (2023).

A.8 FRACOS CHAIN LENGTHS AND DEPTH LIMITS

In our FraCOs method, the process of matching clusters involves conducting a discrete search over potential action permutations. This process requires forming all possible permutations of actions and passing them through the saved clusterer to determine which permutation is currently viable for the meta-policy to choose. The complexity of this search is captured by the permutation formula:

$$^{B}P_{N_c} = \frac{N_c!}{(N_c - B)!}$$

where B is the length of the action chain, and N_c is the total number of cluster-options. As a result, the time complexity for performing this search grows factorially, $O\left(\frac{N_c!}{(N_c-B)!}\right)$, as more cluster-options are introduced or when the chain length is increased. This rapidly becomes computationally expensive as these numbers increase.

A.8.1 Chain Length and Depth in Tabular Experiments

Due to this factorial growth, it is crucial to limit the number of cluster-options and the chain length in experiments that involve discrete cluster search (Experiments 1, 2 and 3.1). For all experiments using cluster search, we chose a **chain length of 2** to keep the computational complexity manageable. Additionally, we restricted the depth of the FraCOs hierarchy to **3 or 4 levels**, depending on the complexity of the environment.

This limitation on depth and chain length helps maintain a balance between the richness of the learned options and the feasibility of performing the cluster search in a reasonable amount of time. The factorial growth of the search process becomes prohibitive as more cluster-options or deeper chains are introduced, making this constraint necessary for efficient execution of our experiments.

A.8.2 Why This Isn't a Problem for Neural Network Cluster Predictions

In contrast, when we extend the FraCOs initiation and action estimation to be used with neural networks, the limitation of conducting computationally expensive permutation searches is alleviated. Neural networks can learn initiation sets and make predictions in a continuous manner, bypassing the need for discrete cluster searches. This removes the necessity of factorially growing search complexity, allowing for more flexibility in chain lengths and depth.

However, the challenge in neural network experiments lies in the sheer number of timesteps required for training and evaluation. For example, in our deep experiments (at a depth of two), training involved many millions of timesteps across nine different environments, totaling **180 million steps** for pre-training, repeated for three seeds. Testing required an additional **120 million steps**, also repeated for three seeds. This was only for FraCOs. When we include experiments using OC (Option Critic), PPO, and PPO25, the total number of timesteps across all experiments becomes **3.06 billion**.

To manage these computational demands, we used a **chain length of 3**. This allowed us to conduct only two warm-up phases, while ensuring that options could still be executed for a reasonable duration (a maximum of nine steps, i.e., 3x3). This setup enabled us to complete the necessary pre-training and testing without sacrificing the quality of the learned options while maintaining computational feasibility.

A.9 FraCOs Experiment Parameters for Tabular Methods

Tabular Q-Learning is a reinforcement learning algorithm where the agent learns the optimal action-value function $Q(s, a)$, which estimates the expected cumulative reward for taking action a in state s and following the optimal policy thereafter. The agent interacts with the environment, updates the Q-values for each state-action pair based on rewards, and converges to the optimal policy over time Sutton & Barto (2018). We implement a vectorized Q learning method.

The key hyperparameters used in all Tabular Q-Learning experiments are listed in Table 2.

In Tabular Q-Learning, the agent repeatedly updates its Q-values for each state-action pair, gradually converging to the optimal policy. By balancing exploration and exploitation, adjusting the learning rate, and prioritizing long-term rewards, the agent learns to optimize its decision-making in the given environment.

FraCOs (Fracture Cluster Options) for Tabular Methods: For reproducibility, the key hyperparameters used in the clustering process and other implementation details are outlined below.

In all tabular experiments, we use HDBSCAN as the clustering method (Campello et al., 2013). The clustering hyperparameters are:

The following table lists the minimum success rewards required for each environment:

Table 2: Hyperparameters for Tabular Q-Learning Experiment

Hyperparameter	Value	Description
eps	0.1	Exploration rate for ϵ-greedy policy. Determines the probability of taking a random action instead of the action with the highest Q-value.
alpha	0.1	Learning rate for Q-value updates. Controls how much the Q-value is updated in each iteration.
gamma	0.99	Discount factor for future rewards.
num_steps	64	Number of steps per episode before environment reset.
max_ep_length	1000	Maximum timesteps allowed in an episode.
anneal_lr	True	Whether to anneal the learning rate as training progresses.
batch_size	64	Number of state-action-reward tuples processed in a batch.
Number of Envs	64	Number of vectorized environments

Table 3: Clustering Hyperparameters for FraCOs

Hyperparameter	Value
Chain length (b)	2
Minimum cluster size	5
Metric	Euclidean
Minimum samples	1
Generate minimum spanning tree	True

FraCOs-Specific Hyperparameters:

The generalisation strength represents the threshold a fracture cluster must pass to be considered for initiation. The threshold is defined as $1 - \texttt{HDBSCAN.predict.strength} >$ generalisation strength.

The FraCOs bias factor determines how much initial Q-values should be scaled to encourage transfer, similar to optimistic initial values. For FraCOs bias depth annealing, the bias depth increases with deeper FraCOs. For example, with a bias factor of 100 and depth of 3, the first bias is scaled by the cube root of 100, the second by the square root, and the final by 100.

A.10 FraCOs Modifications for Deep Learning

In our experiments with deep learning, particularly in environments with large state spaces such as the 64x64x3-dimensional Procgen environments, we found that **HDBSCAN** failed to accurately capture meaningful clusters or predict clusters effectively. Initially, we attempted to integrate a **Variational Autoencoder (VAE)** to use its latent space representation in the fracture formation process Kingma & Welling (2013). However, clustering methods still struggled to deliver satisfactory results.

Consequently, we adopted a simpler clustering approach and shifted to using neural networks to predict both the initiation states and the policy.

Simpler Clustering. To simplify the clustering process, we based clusters solely on sequences of actions. For instance, with a chain length of three, any fracture formed by the action sequence "up", "up", "right" was clustered together, independent of the state. While this approach overlooks some intricacies captured by state-based fracture formations, it was necessary to handle the increased complexity of environments like Procgen.

Cluster Selection. The cluster selection process remained unchanged. We continued to use the usefulness metric, as defined in Equation 4, to select clusters.

Table 4: Minimum Success Reward per Environment

Environment	Minimum Success Reward
Four Rooms	0.97
Grid	0.60
Romesh Maze	0.70
MetaGrid 14x14	0.95

Table 5: FraCOs Hyperparameters

Hyperparameter	Value
Generalisation strength	0.01
FraCOs bias factor	100
FraCOs bias depth anneal	True

Initiation Prediction. Instead of relying on clustering methods to predict initiation states, we trained a neural network to predict the states corresponding to each fracture cluster. This process involved two steps:

1. First, we trained a **Generative Adversarial Network (GAN)** to augment the states in each fracture cluster (since neural networks typically require large datasets).
2. Using both the real and generated states, we trained a neural network as a classifier to predict which fracture cluster a given state belonged to. One neural network was trained to predict all initiations at each hierarchical level. This method significantly improved efficiency, reducing the need for permutation-based discrete searches to a single forward pass.

Policy Prediction. For policy prediction, we utilized a shortcut. Since all FraCOs are derived from trajectories generated by a pre-trained agent, the policy of this trained agent already serves as an approximation for the FraCO policy. We saved the agent's policy at the end of trajectory generation and used it as the policy for all FraCOs. This reuse of the agent's policy minimized additional computation without sacrificing accuracy.

Termination Condition. The termination condition for each FraCO was determined solely by the chain length, maintaining a fixed execution limit per option.

A.11 Experiment Parameters for Deep Methods

All deep methods in our experiments were based on CleanRL's implementation of PPO for the Procgen environments, and we used the same hyperparameters from this implementation. This ensured that the PPO baseline was hyperparameter-tuned, providing a strong and well-optimized baseline for comparison. However, FraCOs and OC-PPO were not specifically hyperparameter-tuned for these environments. Despite this, we consider it reasonable to assume that FraCOs and OC-PPO would perform near their best, as they share similar PPO update mechanisms and underlying structures with the baseline PPO.

While there exists an implementation of OC-PPO by Klissarov et al. (2017), we opted to implement our own version to maintain consistency with CleanRL's PPO implementation. This approach was necessary to ensure that any observed differences in performance were due to algorithmic design rather than implementation differences.

We chose OC-PPO over the standard Option Critic for two primary reasons. First, in our experiments with the standard Option Critic, we observed that the options collapsed quickly, converging to the same behaviour. By integrating the entropy bonus provided by the PPO update, we were able to alleviate this collapse and maintain more diverse option behaviours. Second, PPO has demonstrated significantly better performance than Advantage Actor Critic (A2C) (Mnih, 2016) in the Procgen environments. Given that the original Option Critic framework is based on A2C, using it as a baseline would have led to an unfair comparison, as A2C has been shown to be less effective in

these environments. Therefore, incorporating PPO in both FraCOs and OC-PPO allowed for a fairer and more balanced comparison.

We outline the hyperparameters which we used in the implementation below. All code will be provided from the authors github upon publication.

A.11.1 PPO

We use CleanRL's Procgen implementation as our baseline Huang et al. (2022). The only adaption we make is that we implement entropy annealing, we also have some wrappers which mean Procgen can be used with the Gymnasium API. We have another wrapper which is used for handling multi-level FraCOs in vectorized environments. These wrappers are also applied to the baseline PPO.

Hyperparameters Table 6 provide hyperparameters for PPO.

Parameter	Value	Explanation
easy	1	1 activates "easy" Procgen setting
gamma	0.999	The discount factor.
vf_coef	0.5	Coefficient for the value function loss.
ent_coef	0.01	Entropy coefficient.
norm_adv	true	Whether to normalize advantages.
num_envs	64	Number of parallel environments.
anneal_lr	true	Whether to linearly anneal the learning rate.
clip_coef	0.1	Clipping coefficient for the policy objective in PPO.
num_steps	256	Number of *decisions* per environment per update.
anneal_ent	false	Whether to anneal the entropy coefficient over time.
clip_vloss	true	Whether to clip the value loss in PPO.
gae_lambda	0.95	The lambda parameter for Generalized Advantage Estimation (GAE).
proc_start	1	Indicates the starting level for Procgen environments.
learning_rate	0.0005	The learning rate for the PPO optimizer.
max_grad_norm	0.5	Maximum norm for gradient clipping.
update_epochs	2	Number of epochs per update.
num_minibatches	8	Number of minibatches.
max_clusters_per_clusterer	25	The maximum number FraCOs per level.

Table 6: Selected parameters for the FraCOs implementation with PPO

A.11.2 FRACOS

Neural Network architectures.

- **Input:** $c \times h \times w$ (image observation).
- **Convolutional Layer 1:** 16 filters, kernel size 3×3, stride 1, padding 1, followed by ReLU activation.
- **Convolutional Layer 2:** 32 filters, kernel size 3×3, stride 1, padding 1, followed by ReLU activation.
- **Convolutional Layer 3:** 32 filters, kernel size 3×3, stride 1, padding 1, followed by ReLU activation.
- **Flatten Layer:** Converts the output of the last convolutional layer into a 1D tensor.
- **Fully Connected Layer:** 256 units, followed by ReLU activation.
- **Actor Head:** A linear layer with 256 input units and `total_action_dims` output units, initialized with a standard deviation of 0.01.
- **Critic Head:** A linear layer with 256 input units and 1 output unit (for the value function), initialized with a standard deviation of 1.

The model consists of two heads:

- **Actor Head:** Outputs a probability distribution over the action space for the agent to select actions.
- **Critic Head:** Outputs the value function, which estimates the expected return for the current state.

The network uses ReLU activations after each convolutional and fully connected layer, and the actor and critic heads share the same convolutional layers but have distinct fully connected output layers. In the FraCOs-SSR implementation, the convolutional layers are not reset after the warm-up phase.

Shared State Representation (SSR) details.

In the above architecture, both the actor and critic heads share a common set of convolutional layers. These shared layers process the raw image observations from the environment and extract useful spatial features that are fed into both the actor and critic branches. The use of shared convolutional layers allows the model to leverage the same learned feature representations for both policy and value estimation, promoting efficiency and consistency in learning.

In the **FraCOs-SSR** implementation, the shared convolutional layers are trained during the initial warm-up phase, but they are not reset afterward. This allows the network to retain its learned feature representations across multiple tasks and reuse them for both policy and value estimation during subsequent training phases. By freezing these convolutional layers after the warm-up phase, the network preserves its ability to generalize, while the distinct fully connected layers in the actor and critic heads continue to adapt to new tasks.

Hyperparameters Table 7 provides the full list of FraCOs hyperparameters in experimentation.

Parameter	**Value**	**Explanation**
easy	1	1 activates "easy" Procgen setting
gamma	0.999	The discount factor.
vf_coef	0.5	Coefficient for the value function loss.
ent_coef	0.01	Entropy coefficient.
norm_adv	true	Whether to normalize advantages.
num_envs	64	Number of parallel environments.
anneal_lr	true	Whether to linearly anneal the learning rate.
clip_coef	0.1	Clipping coefficient for the policy objective in PPO.
num_steps	128	Number of *decisions* per environment per update.
anneal_ent	true	Whether to anneal the entropy coefficient over time.
clip_vloss	true	Whether to clip the value loss in PPO
gae_lambda	0.95	The lambda parameter for Generalized Advantage Estimation (GAE).
proc_start	1	Indicates the starting level for Procgen environments.
learning_rate	0.0005	The learning rate for the PPO optimizer.
max_grad_norm	0.5	Maximum norm for gradient clipping
update_epochs	2	Number of epochs per update.
num_minibatches	8	Number of minibatches
max_clusters_per_clusterer	25	The maximum number FraCOs per level

Table 7: Selected parameters for the FraCOs implementation with PPO

A.11.3 OC-PPO

Architectures

- **Input**: $c \times h \times w$ (image observation).
- **Convolutional Layer 1**: 32 filters, kernel size 3×3, stride 2, followed by ReLU activation.
- **Convolutional Layer 2**: 64 filters, kernel size 3×3, stride 2, followed by ReLU activation.
- **Convolutional Layer 3**: 64 filters, kernel size 3×3, stride 2, followed by ReLU activation.

- **Flatten Layer**: Converts the output of the last convolutional layer into a 1D tensor.
- **Fully Connected Layer**: 512 units, followed by ReLU activation.

The model consists of several heads:

- **Option Selection Head**: A linear layer with 512 input units and $num_options$ output units (for selecting options), initialized with orthogonal weight initialization and a bias of 0.0.
- **Intra-Option Action Head**: A linear layer with 512 input units and $num_actions$ output units (for selecting actions within an option), initialized with orthogonal weight initialization and a bias of 0.0.
- **Critic Head**: A linear layer with 512 input units and 1 output unit (for the value function), initialized with orthogonal weight initialization and a standard deviation of 1.
- **Termination Head**: A linear layer with 512 input units and 1 output unit (for predicting termination probabilities), followed by a sigmoid activation to output a probability between 0 and 1.

The architecture is designed to share a common state representation across different heads (option selection, intra-option action selection, value estimation, and option termination). Each head uses the shared state representation for their specific outputs:

- **Option Selection Head**: Outputs a probability distribution over available options.
- **Intra-Option Action Head**: Outputs a probability distribution over the primitive actions available within the current option.
- **Critic Head**: Outputs the value function, estimating the expected return for the current state.
- **Termination Head**: Outputs a termination probability for each option, determining whether the agent should terminate the option at the current state.

Hyperparameters. Table 8 provide the hyperparameters used in OC-PPO.

Parameter	Value	Explanation
easy	1	Activates the "easy" Procgen setting.
gamma	0.999	The discount factor for future rewards.
vf_coef	0.5	Coefficient for the value function loss in PPO.
norm_adv	true	Whether to normalize advantages before policy update.
num_envs	32	Number of parallel environments for training.
anneal_lr	true	Linearly anneals the learning rate throughout training.
clip_coef	0.1	Clipping coefficient for the PPO policy objective.
num_steps	256	Number of steps per environment before an update is performed.
anneal_ent	true	Whether to anneal the entropy coefficient over time.
clip_vloss	false	Whether to clip the value loss in PPO updates.
gae_lambda	0.95	Lambda for Generalized Advantage Estimation (GAE).
proc_start	1	Indicates the starting level for the Procgen environment.
num_options	25	Number of options learned by the agent.
ent_coef_action	0.01	Coefficient for the entropy of the action policy.
ent_coef_option	0.01	Coefficient for the entropy of the option policy.
learning_rate	0.0005	Learning rate for the PPO optimizer.
max_grad_norm	0.1	Maximum norm for gradient clipping.
update_epochs	2	Number of epochs per PPO update.
num_minibatches	4	Number of minibatches per PPO update.

Table 8: Selected hyperparameters for OC-PPO implementation

A.12 OC-PPO UPDATE MECHANISM

The Option-Critic with PPO (OC-PPO) extends the standard Proximal Policy Optimization (PPO) algorithm by incorporating hierarchical options through the Option-Critic (OC) framework. The following key components distinguish OC-PPO from the standalone PPO and OC implementations:

1. Separate Action and Option Policy Updates: In OC-PPO, two sets of policy updates are performed: one for the action policy within an option and one for the option selection policy. Both policies are optimized using the clipped PPO objective, but they operate at different levels of the hierarchy:

- *Action Policy:* The action policy selects the primitive actions based on the current option. For each option, the log-probabilities of actions are calculated, and the advantage function is used to update the action policy.
- *Option Policy:* The option policy determines which option should be selected at each state. This option selection is also updated using the PPO objective, with its own log-probabilities and advantage terms.

Both the action and option policies are clipped to prevent overly large updates, following the standard PPO procedure:

$$\text{Loss}_{\text{policy}} = \max\left(A(\pi) \cdot \frac{\pi_{\text{new}}}{\pi_{\text{old}}}, A(\pi) \cdot \text{clip}\left(\frac{\pi_{\text{new}}}{\pi_{\text{old}}}, 1-\epsilon, 1+\epsilon\right)\right)$$

Here, π_{new} and π_{old} represent the new and old policies for both actions and options, and $A(\pi)$ is the advantage function. This clipping is applied separately for both action and option updates, providing stability in training.

2. Shared State Representation for Action and Option Policies: The OC-PPO architecture shares the state representation between the action and option policies but maintains distinct linear layers for each policy. The state representation is learned via a shared convolutional network. This shared representation ensures that both action and option policies are informed by the same state encoding, allowing for consistent hierarchical decision-making.

- *Action Policy Head:* Receives the state representation and outputs the action logits for the current option.
- *Option Policy Head:* Receives the state representation and outputs the option logits for option selection.

3. Termination Loss for Options: A unique component of OC-PPO is the termination loss, which encourages the agent to decide when to terminate an option and select a new one. The termination function outputs the probability that the current option should terminate. This probability is combined with the advantage function to compute the termination loss:

$$\text{Loss}_{\text{termination}} = \mathbb{E}[\text{termination_probability} \cdot (\text{return} - \text{value})]$$

The termination loss is minimized when the agent terminates the option appropriately, i.e., when the return associated with continuing the current option is less than the estimated value of switching to a new option. The termination probability is computed by a separate network head from the shared state representation.

4. Hierarchical Advantage Calculation: OC-PPO calculates separate advantage terms for actions and options:

- *Action Advantage:* Based on the immediate rewards from the environment while following the current option's policy.
- *Option Advantage:* Based on the value of switching to a new option versus continuing with the current option.

Each advantage is normalized independently, and separate PPO updates are applied to both the action and option policies based on their respective advantage functions.

5. Regularization via Entropy for Both Action and Option Policies: As in standard PPO, entropy regularization is applied to encourage exploration. However, in OC-PPO, this regularization is applied both at the action level (to encourage diverse action selection within an option) and at the option level (to encourage exploration of different options). The overall loss function includes separate entropy terms for actions and options:

$$\text{Loss}_{\text{entropy}} = \alpha_{\text{action}} \cdot \mathbb{H}(\pi_{\text{action}}) + \alpha_{\text{option}} \cdot \mathbb{H}(\pi_{\text{option}})$$

6. Clipping for Value Function: Like in PPO, OC-PPO also employs clipping for the value function updates to prevent large changes in the value estimate between consecutive updates. This applies to the shared value function, which evaluates the expected returns from both primitive actions and options.

$$\text{Loss}_{\text{value}} = 0.5 \cdot \max\left((V_{\text{new}} - R)^2, (V_{\text{old}} + \text{clip}(V_{\text{new}} - V_{\text{old}}, -\epsilon, \epsilon))^2\right)$$

A.13 Full Procgen results

The below tables outline all results for all tested methods in all environments. They show the means averaged over three seeds.

Environment	PPO-5	PPO-25	OC-PPO	FraCOs	FraCOs-SSR
BigFish	13.16	23.58	5.80	8.29	**24.77**
Climber	7.00	**12.11**	4.08	7.03	11.63
Coinrun	9.47	10.00	10.00	9.53	10.00
Fruitbot	25.33	**32.81**	30.90	29.85	**31.96**
Dodgeball	5.34	**11.16**	5.72	4.42	7.23
Leaper	5.63	5.63	3.13	5.52	5.63
Ninja	7.10	10.00	10.00	7.20	10.00
Plunder	10.33	**16.16**	8.00	12.24	13.15
StarPilot	24.41	**37.63**	21.33	33.27	31.04

Table 9: IID performance metrics for different methods

Environment	PPO-5	PPO-25	OC-PPO	FraCOs	FraCOs-SSR
BigFish	2.53	4.50	2.84	3.40	**13.19**
Climber	3.89	5.92	2.50	3.09	**8.34**
Coinrun	6.77	7.66	8.13	7.29	**8.13**
Fruitbot	15.81	22.00	24.10	25.07	**27.54**
Dodgeball	1.38	1.31	1.13	0.69	1.28
Leaper	2.71	2.50	2.19	2.71	**3.91**
Ninja	5.20	7.19	7.81	4.86	8.05
Plunder	6.25	8.78	5.39	**8.75**	8.20
StarPilot	15.53	14.59	15.58	20.31	**24.63**

Table 10: OOD performance metrics for different methods

A.14 Experiment legacy: Deep State and Reward generalisation in Simple Environments

In this experiment, we extend the Proximal Policy Optimization (PPO) algorithm (Schulman et al., 2017) to incorporate multi-level hierarchies of FraCOs. Here, the MetaGrid environment layout and reward location change after every episode (shuffle). An episode terminates after 1000 time steps or when the goal is reached. We evaluate the agent in both MetaGrid 14x14 and MetaGrid 21x21 shuffled environments, using the same FraCOs discovered during Experiment 2; Importantly all FraCOs are derived from tabular learning only. The results are presented in Figure 11.

Environment	PPO-5	PPO-25	OC-PPO	FraCOs	FraCOs-SSR
BigFish	2.53	4.50	3.78	8.29	**13.19**
Climber	4.65	5.92	3.63	4.26	**9.05**
Coinrun	6.77	7.66	8.43	8.23	**8.75**
Fruitbot	15.81	24.81	24.70	26.13	**27.54**
Dodgeball	1.59	1.40	1.78	1.63	**2.09**
Leaper	2.91	3.75	2.50	3.44	**4.38**
Ninja	5.20	8.13	7.97	4.98	**8.43**
Plunder	6.61	8.78	5.95	**10.28**	9.06
StarPilot	15.53	17.21	17.49	20.31	**28.84**

Table 11: OOD Peak performance metrics for different methods

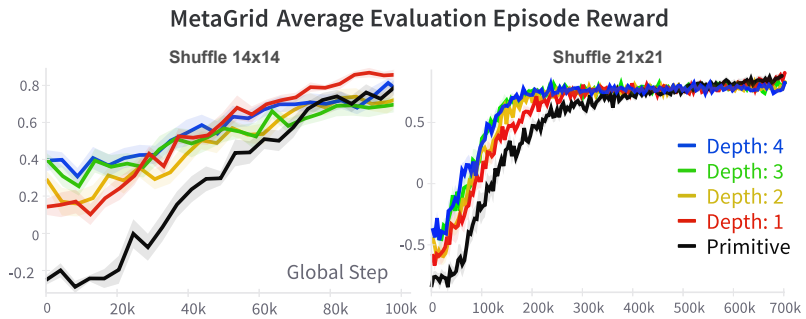


Figure 11: Evaluation episode rewards for FraCOs-PPO in MetaGrid environments of size 14x14 and 21x21, where each domain is randomly shuffled after each episode. The 14x14 results are averaged over 10 independent seeds, and the 21x21 results are averaged over 5 seeds. Shaded areas represent the standard error.

A.15 Success Criteria Hyperparameters

Environment	Minimum Success Returns
Four Rooms	0.97
Nine Rooms	0.60
Romesh Maze	0.70
MetaGrid 14x14	0.95
BigFish	5
Climber	7
CoinRun	7.5
Dodgeball	5
FruitBot	7.5
Leaper	7
Ninja	7.5
Plunder	10

Table 12: Minimum Success Returns for Various Environments