META REINFORCEMENT LEARNING WITH AUTONOMOUS TASK INFERENCE

Sungryull Sohn[1] **Hyunjae Woo**[1] **Honglak Lee**[2,1]
[1]University of Michigan
{srsohn, hjwoo}@umich.edu
[2]Google Brain
honglak@google.com

ABSTRACT

We propose and solve a new few-shot RL problem, where the task is characterized by a subtask graph which describes a set of subtasks and their dependencies that are unknown to the agent. The agent needs to quickly adapt to the task over multiple episodes in the adaptation phase to maximize the return in the test phase. Instead of directly trying to learn a meta-policy, we introduce a *neural subtask graph inferencer* (NSGI) that infers the latent parameter of the task by interacting with the environment and maximizes the return given the latent parameter. To facilitate the learning, we adopt the intrinsic reward inspired by upper confidence bound (UCB) that encourages to explore the environment more efficiently. Our experiment results on two 2D maze domains show that the proposed method adapts more efficiently and generalizes better than the existing meta RL and hierarchical RL methods.

1 INTRODUCTION

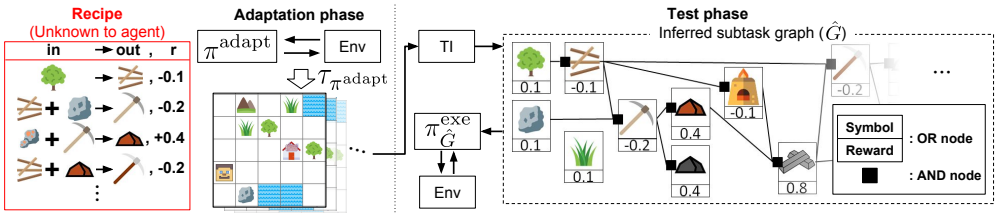

Figure 1: An example task and the procedure of our agent solving the task. Our adaptation policy π^{adapt} learns to maximally gather information about current task in adaptation phase. Task inference (TI) module infers the subtask graph G from the adaptation trajectory $\tau_{\pi^{\text{adapt}}}$. Lastly, execution policy π^{exe} executes the inferred subtask graph $\hat{G}$ to maximize the reward in test phase.

Recently, reinforcement learning (RL) systems achieved super-human performance on many complex tasks (Mnih et al., 2015; Silver et al., 2016; Van Seijen et al., 2017). However, these works have mostly been focused on a single known task where the agent can be trained for a large number of times. In order to build more practical and scalable RL agent, it is necessary for the agent to execute many unknown tasks. To this end, recent works on multi-task RL tried to build agent that can execute any given task descriptions such as sequential instructions (Oh et al., 2017; Andreas et al., 2017; Yu et al., 2017; Denil et al., 2017) and complex graph structures (Sohn et al., 2018). However, such task description may not be readily available in practice. Likewise, Hochreiter et al. (2001); Duan et al. (2016); Wang et al. (2016); Finn et al. (2017) proposed meta-RL algorithms that can quickly adapt to the new task without task description input, but they have focused only on the simple tasks such as a single goal navigation task and a bandit problem.

We formulate a new meta RL problem, named *subtask graph inference* problem, where the agent should execute the complex task characterized by a *subtask graph* (Sohn et al., 2018) that is not given to the agent; instead, the agent is given few episodes of adaptation phase to learn about the task as in Figure 1. After the adaptation phase, the agent is required to maximize the reward within a time limit by executing subtasks in an optimal order in test phase. The subtask graph defines the subtasks with the corresponding rewards and the preconditions (e.g., recipe in Figure 1). Inspired by recent multi-task RL works, we propose *neural subtask graph inferencer* that explicitly infers the latent information about the task (e.g., subtask graph) which describes the subtasks and their dependencies, and executes the inferred task.

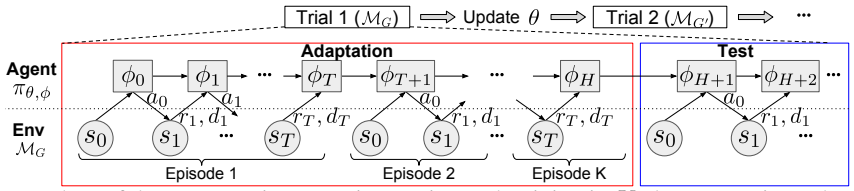

Figure 2: Procedure of the agent-environment interaction and training in K-shot RL setting. The agent has a slow-learning parameter θ and a fast-learning parameter ϕ.

The contribution of this work can be summarized as follows: (1) We propose a new meta RL problem with a richer form of tasks compared to the recent works on meta RL. (2) We propose an inductive logic programming-based task inference module that can infer the latent task parameter from the agent's trajectory. (3) We propose a meta policy that is trained to efficiently infer the task parameter for the faster adaptation. (4) We compare our method with other meta-RL agents on two 2D visual domains to show that our method can efficiently adapt to the task with complex subtask dependencies.

2 Formalism

2.1 Few-shot Reinforcement Learning

We assume that a task is defined by an MDP tuple $\mathcal{M}_G = (\mathcal{S}; \mathcal{A}; \mathcal{P}_G; \mathcal{R}_G; \ _G; \ )$ parameterized by the task parameter G with state set $\mathcal{S}$, action set $\mathcal{A}$, transition dynamics $\mathcal{P}_G$, reward function $\mathcal{R}_G$, initial state distribution $_G$, and discount factor . In K-shot RL, a trial consists of the adaptation phase with K episodes and the following test phase under a fixed MDP $\mathcal{M}_G$ as in Figure 2. Between trials, the task parameter G is sampled and the agent updates its slow-learning parameter in training. In adaptation phase, the agent adapts to the task by updating its fast-learning parameter , where the initialization and update function of are the functions of . In the test phase, the agent's performance is measured in terms of the loss $\mathcal{L}(\ _\theta) = -\mathsf{E}_{\pi \ ; \ H+1} \ \prod_{t=1}^{H'} r_t'$, where H is the adaptation phase horizon, $_{H+1}$ is the fast-learning parameter after the adaptation, $_{\theta,\phi_{H+1}}$ is the policy after the adaptation, H' is the test phase horizon, and r_t' is the reward at time t in test phase. The goal is to find the optimal policy $_\theta$ that minimizes the loss $\mathcal{L}(\ _\theta)$.

2.2 Subtask Graph Inference Problem

Subtask graph and environment We define the terminologies as follows:

- **Completion**: $\mathsf{x}_t = [x_t^1; \cdots; x_t^N]$ where $x_t^i = 1$ if agent has executed subtask i, and 0 otherwise.
- **Eligibility**: $\mathsf{e}_t = [e_t^1; \cdots; e_t^N]$ where $e_t^i = 1$ if subtask i is *eligible* (i.e., agent can perform subtask i) at time t, and 0 otherwise.
- **Precondition**: $\mathsf{c} = \{c^1; \cdots; c^N\}$ where c^i is the logical expression of which variable is the completion vector x and value is the eligibility of i-th subtask e^i. We refer the corresponding Boolean function $f_{\mathbf{c}}^i : \mathsf{x} \mapsto e^i$ as **precondition function**.
- **Subtask reward**: $\mathsf{r} = [r^1; \cdots; r^N]$ specifies the reward given to agent for executing each subtask.
- **Time budget**: $\text{step}_t \in \mathsf{R}$ is the remaining time-steps until episode termination.
- **Episode budget**: $\text{epi}_t \in \mathsf{R}$ is the remaining number of episodes in adaptation phase.
- **Observation**: $\mathsf{obs}_t \in \mathsf{R}^{H \ W \ C}$ is a visual observation at time t as illustrated in Figure 1.

The subtask graph G defines N subtasks with corresponding rewards r and the preconditions c. The state input at time t consists of $\mathsf{s}_t = \{\mathsf{obs}_t; \mathsf{x}_t; \mathsf{e}_t; \text{step}_t; \text{epi}_t\}$. We assume that the agent has learned a set of *options* ($\mathcal{O}$) (Precup, 2000; Stolle & Precup, 2002; Sutton et al., 1999) that performs subtasks by executing one or more primitive actions.

3 Method

To tackle the problem, our neural subtask graph inferencer (NSGI) model infers the latent subtask graph G, instead of adapting the fast-learning parameter as in Figure 7. The adaptation policy $_\theta^{\text{adapt}}(\mathsf{o}| \ )$ rollouts the episodes in adaptation phase. The task inference module subsequently takes as input the adaptation trajectory and infers the subtask graph $\hat{G}$ using inductive logic programming (ILP) algorithm. In the test phase, the inferred task parameter $\hat{G}$ is used as the adapted fast-learning parameter $_{H+1}$. Our execution policy $_G^{\text{exe}}(\mathsf{o}|\mathsf{s})$ takes as input the inferred subtask graph $\hat{G}$, and execute it to maximize the return.

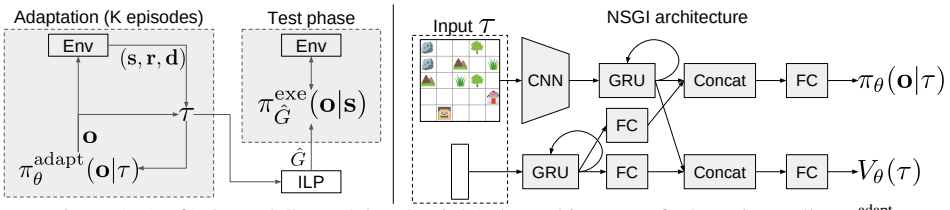

Figure 3: (Left) Our NSGI model and (Right) the architecture of adaptation policy $\pi_\theta^{\text{adapt}}$.

3.1 Adaptation phase: subtask graph inference

Let $G = (\mathbf{c}; \mathbf{r})$ be the subtask graph with the subtask rewards $\mathbf{r}$ and $\mathbf{c}$, and $\tau_t = \{\mathbf{s}_1; \mathbf{o}_1; \mathbf{r}_1; \mathbf{d}_1; \cdots; \mathbf{s}_t\}$ be the adaptation trajectory under adaptation policy π_θ until time step t. The maximum-likelihood estimate (MLE) of latent variables G, conditioned on the trajectory τ_H can be written as

$$\hat{G}_{\text{MLE}} = \arg\max_{\mathbf{c},\mathbf{r}} p(\tau_H|\mathbf{c}; \mathbf{r}); \tag{1}$$

where H is the adaptation horizon. Then, the likelihood term can be further expanded as

$$p(\tau_H|\mathbf{c}; \mathbf{r}) = p(\mathbf{s}_1|\mathbf{c}) \prod_{t=1}^{H} \pi_\theta(\mathbf{o}_t|\tau_t)\, p(\mathbf{s}_{t+1}|\mathbf{s}_t; \mathbf{o}_t; \mathbf{c}) p(r_t|\mathbf{s}_t; \mathbf{o}_t; \mathbf{r}) p(d_t|\mathbf{s}_t; \mathbf{o}_t) \tag{2}$$

$$\propto p(\mathbf{s}_1|\mathbf{c}) \prod_{t=1}^{H} p(\mathbf{s}_{t+1}|\mathbf{s}_t; \mathbf{o}_t; \mathbf{c}) p(r_t|\mathbf{s}_t; \mathbf{o}_t; \mathbf{r}); \tag{3}$$

where we dropped the terms that are independent of G. By definition, the precondition $\mathbf{c}$ depends only on the precondition probability in the transition dynamics (see Supplementary material for detail): $p(\mathbf{s}_{t+1}|\mathbf{s}_t; \mathbf{o}_t; \mathbf{c}) \propto p(\mathbf{e}_{t+1}|\mathbf{x}_{t+1}; \mathbf{c})$. Then, we can compute $\hat{G}_{\text{MLE}}$ as:

$$\hat{G}_{\text{MLE}} = (\hat{\mathbf{c}}_{\text{MLE}}; \hat{\mathbf{r}}_{\text{MLE}}) = \left(\arg\max_{\mathbf{c}} \prod_{t=1}^{H} p(\mathbf{e}_t|\mathbf{x}_t; \mathbf{c}); \arg\max_{\mathbf{r}} \prod_{t=1}^{H} p(r_t|\mathbf{s}_t; \mathbf{o}_t; \mathbf{r}) \right) : \tag{4}$$

Precondition inference via logic induction For given precondition $\mathbf{c}$ and completion $\mathbf{x}$, the precondition function uniquely determines the corresponding eligibility as $\mathbf{e} = f_{\mathbf{c}}(\mathbf{x})$ (see Section 2.2). Thus, finding $\hat{\mathbf{c}}_{\text{MLE}}$ is equivalent to finding the precondition function $f_{\mathbf{c}}(\cdot)$ that satisfies all the input-output pairs $\{\mathbf{x}_t; \mathbf{e}_t\}_{t=1}^{H}$, which is *logic program induction* problem. To solve this problem, we used the *classification and regression tree* (CART) to infer the precondition function $f_{\mathbf{c}}$ for each subtask based on Gini impurity (Breiman, 2017). Intuitively, the reconstructed decision tree is the simplest Boolean function approximation for the given input-output pairs. Then, we converted it to a logic expression (i.e., precondition) in sum-of-product (SOP) notation to build the subtask graph.

Subtask reward inference For inferring the subtask reward $\hat{\mathbf{r}}_{\text{MLE}}$, we modeled each component of subtask reward as a Gaussian distribution $\mathbf{r}^i \sim \mathcal{N}(\hat{\mu}^i; \hat{\sigma}^i)$. Then, the $\hat{\mu}^i_{\text{MLE}}$ is the sample mean of the rewards received after executing subtask i in the trajectory τ_H: $\hat{\mathbf{r}}^i_{\text{MLE}} = \hat{\mu}^i_{\text{MLE}} = \mathbb{E}[\mathbf{r}_t|\mathbf{o}_t = i]$.

3.2 Test phase: subtask graph execution

Given the subtask graph $\hat{G}$ inferred in adaptation phase, we learn a task execution policy $\pi_{\hat{G}}^{exe}(\mathbf{o}_t|\mathbf{s}_t)$ that aims to maximize the cumulative reward in test phase. Since the subtask graph $\hat{G}$ is given, the problem is equivalent to the subtask graph execution problem (Sohn et al., 2018). Following (Sohn et al., 2018), we used graph reward propagation (GRProp) policy $\pi_{\hat{G}}^{\text{GRProp}}(\mathbf{o}|\mathbf{x})$ as our SGE policy. Please see the supplementary material for the details on GRProp policy.

3.3 Architecture

Figure 7 illustrates the architecture of our NSGI model. Our adaptation policy takes the agent's trajectory $\tau_t = \{\mathbf{s}_t; \mathbf{o}_t; \mathbf{r}_t; \mathbf{d}_t\}$ at time t as input. For observation input, the policy and value outputs shares the embedding encoded by CNN and GRU. For other inputs, we concatenated them and used two separate fully-connected (FC) layers for the outputs after GRU encoding. Finally, the observation and flat embeddings are concatenated and fed to FC layer to produce each of policy and value outputs (see supplementary material for more detail).

3.4 POLICY OPTIMIZATION

Following the few-shot RL formulation, we can directly optimize the loss function $\mathcal{L}(\ _\theta)$ using policy gradient by treating the return in test phase as the last step reward of adaptation phase. However, we found it challenging to train our model for two reasons: 1) the delayed and sparse reward, and 2) high task variance due to high expressive power of subtask graph. To facilitate the learning, we propose to give an intrinsic bonus in adaptation phase. Inspired by the upper confidence bound (UCB) (Auer et al., 2002), we define the bonus term as $r_t^{\text{UCB}} = w_{\text{UCB}}\,\mathbb{I}(\mathbf{x}_t \in \ _{t\ 1})$, where $w_{\text{UCB}} = \frac{1}{N}\sum_{i=1}^{N} \log(n_0^{(i)} + n_1^{(i)})/n_{e_t^i}^{(i)}$, N is the number of subtasks and $n_e^{(i)}$ is the number of occurrences of $\mathbf{e}^{(i)} = e$ until time t. The weight w_{UCB} is designed to encourage the agent to make the subtask eligible that is seldom eligible. The conditioning term $\mathbb{I}(\mathbf{x}_t \in \ _{t\ 1})$ encourages the agent to visit the state with novel $\mathbf{x}_t$, since the inputs with same $\mathbf{x}_t$ are ignored in ILP module for duplication. We trained the adaptation policy $\ _\theta$ using actor-critic method with GAE (Schulman et al., 2015) to minimize the loss $\mathcal{L}^{\text{UCB}}(\ _\theta) = -\mathbb{E}_\pi\left[\sum_{t=1}^{H} r_t^{\text{UCB}}\right]$; in adaptation phase, where H is the adaptation horizon (see supplementary material for the complete procedure of training).

4 EXPERIMENT

In the experiment, we investigate the following research questions: 1) Can NSGI accurately infer the task parameter G? 2) Does the adaptation policy $\ _\theta$ improves the inference? 3) Does using UCB bonus facilitate the training? 4) How does NSGI perform compared to other meta-RL algorithms? 5) Can NSGI generalize to longer adaptation horizon, and unseen and more complex tasks?

4.1 DOMAIN

Following (Sohn et al., 2018), we evaluate the performance of our agents on two domains: **Mining** and **Playground**. Table 1 summarizes the subtask graphs used for evaluation (see supplementary material for details on subtask graph generation).

	Playground				Mining
Task	**D1**	**D2**	**D3**	**D4**	**Eval**
Depth	4	4	5	6	4-10
Subtask	13	15	16	16	10-26

Table 1: (Playground) **D1** have the same graph structure as training set, but the graph was unseen. **D2,D3**, and **D4** have (unseen) larger graph structures. (Mining) **Eval** are unseen during training.

Mining is inspired by Minecraft (see Figure 1) where the agent may receive reward by picking up raw materials in the maze or crafting items. The set of subtasks and subtask graph were hand-coded according to the crafting recipes in Minecraft.

Playground is a more flexible and challenging domain. The subtask graph is randomly generated without any context, hence its precondition can be any logical expression and the reward may be delayed. Some of the objects randomly move, which makes the environment stochastic.

4.2 AGENTS

We evaluated the following policies:

- **Random** policy executes any eligible subtask.
- **GRProp+GT** is graph reward propagation policy with ground-truth subtask graph input.
- **NSGI-RND** (Ours) uses random policy to rollout in adaptation phase
- **NSGI-Meta** (Ours) uses learned meta adaptation policy to rollout in adaptation phase
- **RL**^{2} is the meta-RL agent in (Duan et al., 2016), trained to maximize the return over K episodes
- **HRL** is the hierarchical RL agent in (Sohn et al., 2018) trained with actor-critic method in adaptation phase. After the test phase, the network parameter is reset.

For **RL**^{2} and **HRL**, we used the same architecture as our NSGI adaptation policy.

4.3 RESULTS

To evaluate the generalization performance, we trained the agent on the smallest graph **D1** with 10 episodes budget, and tested on the unseen graphs **D1** and unseen larger graphs **D2-D4** with up to 20 episodes budget in **Playground**. In **Mining**, we trained on randomly generated graphs with 25 episodes budget and tested on unseen graphs **Eval** with up to 50 episodes budget (see Table 1). We measured the normalized return in test phase $\hat{R} = (R - R_{min})/(R_{max} - R_{min})$ averaged over four random seeds, where R_{min} and R_{max} correspond to the average return of the **Random** and the **GRProp+GT** agent respectively. The shaded area in the plot indicates the range between $\hat{R} +$ and $\hat{R} -$ where is the standard error of normalized return.

4.3.1 TRAINING PERFORMANCE

The learning curves is illustrated in Figure 4. Over the training, the performance of our NSGI-Meta improves over the NSGI-RND with a large margin. It demonstrates that our meta adaptation policy learns to explore the environment much more efficiently than random policy by maximizing the UCB bonus. Also, the performance of RL^2 improves over time, eventually outperforming the HRL. This indicates that RL^2 learns 1) a common strategy that is generally applicable to all the tasks and 2) an efficient adaptation scheme such that it can adapt to the given task more quickly than standard policy gradient update in HRL.

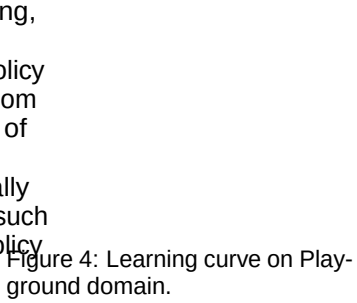

Figure 4: Learning curve on Playground domain.

4.3.2 ADAPTATION AND GENERALIZATION PERFORMANCE

Figure 5: Generalization performance with varying adaptation horizon on unseen tasks (Eval) and larger tasks (D2-4) for longer (unseen) horizon. Our NSGI-based models outperform HRL and RL^2.

Adaptation efficiency In figure 5, we measured the normalized reward with varying number of adaptation episode budget to see how quickly each agent can adapt to the given task. Our NSGI-Meta consistently outperforms NSGI-RND across all the tasks, showing that our meta adaptation policy can efficiently explore only the informative states that is likely to improve the subtask graph inference. Also, our NSGI-Meta and NSGI-RND perform better than HRL and RL^2 in all the tasks, showing that separating the problem of task inference and execution brings substantial gains for solving complex task with subtask dependencies. Finally, RL^2 outperforms HRL for the seen tasks (D1 and Eval) and seen adaptation horizon by learning the common strategy shared among tasks.

Generalization We evaluated the generalization performance on unseen task and longer adaptation horizon in figure 5. Both of our NSGI-based models generalize well to unseen tasks and longer adaptation horizon without significant degradation of performance. It demonstrates that the efficient exploration scheme learned from the UCB intrinsic bonus generalizes well to unseen tasks and longer adaptation horizon, and that our task execution policy, GRProp, generalized well to unseen tasks. However, RL^2 fails to generalize to unseen task and longer adaptation horizon. In D2-4 with unseen adaptation horizon, the RL^2 is outperformed by HRL and the performance even decreases for very long horizon case (D2, D3, and mining). This indicates that 1) the adaptation scheme that RL^2 learned does not generalize well to longer adaptation horizon, and fails to outperform policy gradient update of HRL, and 2) the common strategies learned by RL^2 to solve training tasks does not generalize well to the unseen testing tasks.

5 CONCLUSION

We introduce and solve a few-shot RL problem with a complex subtask dependencies. We propose to learn a meta-policy that learns to efficiently infer the latent structure of the task in the form of subtask graph, and the execution policy that executes the inferred subtask graph. The empirical results show that our agent can efficiently explore the environment in adaptation phase to improve the inference, and exploit the inferred information in test phase. In this work, we assumed that the option is pre-learned and the environment provides the status of each subtask. In the future work, our model may be extended to infer the relevant subtask status from the observation, and discover the option from the structure of the task.

REFERENCES

Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In *ICML*, 2017.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

Leo Breiman. *Classification and regression trees*. Routledge, 2017.

Misha Denil, Sergio Gómez Colmenarejo, Serkan Cabi, David Saxton, and Nando de Freitas. Programmable agents. *arXiv preprint arXiv:1706.06383*, 2017.

Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1126–1135. JMLR. org, 2017.

Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pp. 87–94. Springer, 2001.

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

Junhyuk Oh, Satinder Singh, Honglak Lee, and Pushmeet Kohli. Zero-shot task generalization with multi-task deep reinforcement learning. *arXiv preprint arXiv:1706.05064*, 2017.

Doina Precup. Temporal abstraction in reinforcement learning. 2000.

John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

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

Sungryull Sohn, Junhyuk Oh, and Honglak Lee. Hierarchical reinforcement learning for zero-shot generalization with subtask dependencies. In *NeurIPS*, pp. 7156–7166, 2018.

Martin Stolle and Doina Precup. Learning options in reinforcement learning. In *International Symposium on Abstraction, Reformulation, and Approximation*, pp. 212–223. Springer, 2002.

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.

Harm Van Seijen, Mehdi Fatemi, Joshua Romoff, Romain Laroche, Tavian Barnes, and Jeffrey Tsang. Hybrid reward architecture for reinforcement learning. In *Advances in Neural Information Processing Systems*, pp. 5392–5402, 2017.

Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

Haonan Yu, Haichao Zhang, and Wei Xu. A deep compositional framework for human-like language acquisition in virtual environment. *arXiv preprint arXiv:1703.09831*, 2017.

A DETAILS OF THE TASK

We define each task as an MDP tuple $\mathcal{M}_G = (\mathcal{S}; \mathcal{A}; \mathcal{P}_G; \mathcal{R}_G; \ _G; \ )$ where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $\mathcal{P}_G : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0; 1]$ is a task-specific state transition function, $\mathcal{R}_G : \mathcal{S} \times \mathcal{A} \to \mathsf{R}$ is a task-specific reward function and $\ _G : \mathcal{S} \to [0; 1]$ is a task-specific initial distribution over states. In the experiment, we used $\ = 1$. We describe the subtask graph G and each component of MDP in the following paragraphs.

Terminologies We define the terminologies as follows:

- **Completion**: $\mathsf{x}_t = [x_t^1; \cdots; x_t^N]$ where $x_t^i = 1$ if agent has executed subtask i, and 0 otherwise.
- **Eligibility**: $\mathsf{e}_t = [e_t^1; \cdots; e_t^N]$ where $e_t^i = 1$ if subtask i is *eligible* (i.e., agent can perform subtask i) at time t, and 0 otherwise.
- **Precondition**: $\mathsf{c} = \{c^1; \cdots; c^N\}$ where c^i is the logical expression of which variable is the completion vector x and value is the eligibility of i-th subtask e^i. We refer the corresponding Boolean function $f_{\mathbf{c}}^i : \mathsf{x} \mapsto e^i$ as **precondition function**.
- **Subtask reward**: $\mathsf{r} = [r^1; \cdots; r^N]$ specifies the expected reward given to agent for executing each subtask.
- **Time budget**: $\text{step}_t \in \mathsf{R}$ is the remaining time-steps until episode termination.
- **Episode budget**: $\text{epi}_t \in \mathsf{R}$ is the remaining number of episodes in adaptation phase.
- **Observation**: $\mathsf{obs}_t \in \mathsf{R}^{H \ W \ C}$ is a visual observation at time t as illustrated in Figure 1.

Subtask graph and subtask The subtask graph consists of N subtasks that is a subset of pre-learned options $\mathcal{O}$, the subtask reward r, and the set of precondition of each subtask c. The set of subtasks is $\mathcal{O} = \mathcal{A}_{int} \times \mathcal{X}$, where $\mathcal{A}_{int}$ is a set of primitive actions to interact with objects, and $\mathcal{X}$ is a set of all types of interactive objects in the domain. To execute a subtask $(a_{int}; obj) \in \mathcal{A}_{int} \times \mathcal{X}$, the agent should move on to the target object obj and take the primitive action a_{int}.

State The state s_t consists of the observation obs_t, the completion vector x_t, the eligibility vector e_t, the time budget step_t, and the episode budget epi_t. An observation obs_t is represented as $H \times W \times C$ tensor, where H and W are the height and width of map respectively, and C is the number of object types in the domain. The $(h; w; c)$-th element of observation tensor is 1 if there is an object c in $(h; w)$ on the map, and 0 otherwise. The time budget step_t indicates the number of remaining time-steps until the episode termination. The episode budget epi_t indicates the number of remaining episodes until the adaptation phase terminates. The completion vector and eligibility vector provide additional information about N subtasks. The eligibility vector e_t is computed from completion vector x_t and the precondition function $f_{\mathbf{c}}$ as $\mathsf{e}_t = f_{\mathbf{c}}(\mathsf{x}_t)$, where the precondition function $f_{\mathbf{c}}$ is defined by subtask graph G. The details of completion vector and eligibility vector will be explained in **Transition dynamics and reward** paragraph below.

Initial state distribution In the beginning of episode, the initial time budget step_t is sampled from a pre-specified range N_{step} for each subtask graph (See section C for detail), the completion vector x_t is initialized to a zero vector in the beginning of the episode $\mathsf{x}_0 = [0; \cdots; 0]$ and the observation obs_0 is sampled from the task-specific initial state distribution $\ _G$. Specifically, the observation is generated by randomly placing the agent and the N objects corresponding to the N subtasks defined in the subtask graph G.

Transition dynamics and reward Given the current state $(\mathsf{obs}_t; \mathsf{x}_t; \mathsf{e}_t; \text{step}_t; \text{epi}_t)$ and option taken o_t, the next step state $(\mathsf{obs}_{t+1}; \mathsf{x}_{t+1}; \mathsf{e}_{t+1}; \text{step}_{t+1}; \text{epi}_{t+1})$ is computed from the subtask graph G. When the agent executes subtask i, the i-th element of completion vector is updated by the following update rule:

$$x_{t+1}^i = \begin{array}{cc} 1 & \text{if} \quad e_t^i = 1 \\ x_t^i & \text{otherwise} \end{array} ; \tag{5}$$

The agent receives the reward randomly sampled from a distribution with the mean value of r^i. For both **Playground** and **Mining** domain, we used uniform distribution from $0{:}8 * \mathsf{r}^i$ to $1{:}2 * \mathsf{r}^i$. The observation is updated such that agent moves on to the target object, and perform corresnponding primitive action (See Section B for the full list of subtasks and corresponding primitive actions on Mining and Playground domain). The precondition function $f_{\mathbf{c}}$ computes eligibility vector e_{t+1} from

the completion vector $\mathbf{x}_{t+1}$ and subtask graph G as follows:

$$e^i_{t+1} = \underset{j \in Child_i}{\text{OR}} \; y^j_{AND} \quad ; \tag{6}$$

$$y^i_{AND} = \underset{j \in Child_i}{\text{AND}} \; \hat{x}^{i,j}_{t+1} \quad ; \tag{7}$$

$$\hat{x}^{i,j}_{t+1} = x^j_{t+1} w^{i,j} + (1 - x^j_{t+1})(1 - w^{i,j}); \tag{8}$$

where $w^{i,j} = 0$ if there is a NOT connection between i-th node and j-th node, otherwise $w^{i,j} = 1$. Intuitively, $\hat{x}^{i,j}_t = 1$ when j-th node does not violate the precondition of i-th node. Executing each subtask costs different amount of time depending on the map configuration. Specifically, the time cost is given as the Manhattan distance between agent location and target object location in the grid-world plus one more step for performing a primitive action.

B Details of Environment

B.1 Mining

There are 15 types of objects: *Mountain*, *Water*, *Work space*, *Furnace*, *Tree*, *Stone*, *Grass*, *Pig*, *Coal*, *Iron*, *Silver*, *Gold*, *Diamond*, *Jeweler's shop*, and *Lumber shop*. The agent can take 10 primitive actions: *up*, *down*, *left*, *right*, *pickup*, *use1*, *use2*, *use3*, *use4*, *use5* and agent cannot moves on to the *Mountain* and *Water* cell. *Pickup* removes the object under the agent, and *use*'s do not change the observation. There are 26 subtasks in the Mining domain:

- Get wood/stone/string/pork/coal/iron/silver/gold/diamond: The agent should go to *Tree*/*Stone*/*Grass*/*Pig*/*Coal*/*Iron*/*Silver*/*Gold*/*Diamond* respectively, and take *pickup* action.
- Make firewood/stick/arrow/bow: The agent should go to *Lumber shop* and take *use1*/*use2*/*use3*/*use4* action respectively.
- Light furnace: The agent should go to *Furnace* and take *use1* action.
- Smelt iron/silver/gold: The agent should go to *Furnace* and take *use2*/*use3*/*use4* action respectively.
- Make stone-pickaxe/iron-pickaxe/silverware/goldware/bracelet: The agent should go to *Work space* and take *use1*/*use2*/*use3*/*use4*/*use5* action respectively.
- Make earrings/ring/necklace: The agent should go to *Jeweler's shop* and take *use1*/*use2*/*use3* action respectively.

The icons used in Mining domain were downloaded from `www.icons8.com` and `www.flaticon.com`. The *Diamond* and *Furnace* icons were made by Freepik from `www.flaticon.com`.

B.2 Playground

There are 10 types of objects: *Cow*, *Milk*, *Duck*, *Egg*, *Diamond*, *Heart*, *Box*, *Meat*, *Block*, and *Ice*. The *Cow* and *Duck* move by 1 pixel in random direction with the probability of 0.1 and 0.2, respectively. The agent can take 6 primitive actions: *up*, *down*, *left*, *right*, *pickup*, *transform* and agent cannot moves on to the *block* cell. *Pickup* removes the object under the agent, and *transform* changes the object under the agent to *Ice*. The subtask graph was randomly generated without any hand-coded template (see Section C for details).

C Details of Subtask Graph Generation

C.1 Mining Domain

The precondition of each subtask in Mining domain was defined as Figure 6. Based on this graph, we generated all possible sub-graphs of it by removing the subtask node that has no parent node, while always keeping subtasks A, B, D, E, F, G, H, I, K, L. The reward of each subtask was randomly scaled by a factor of $0.8 \sim 1.2$.

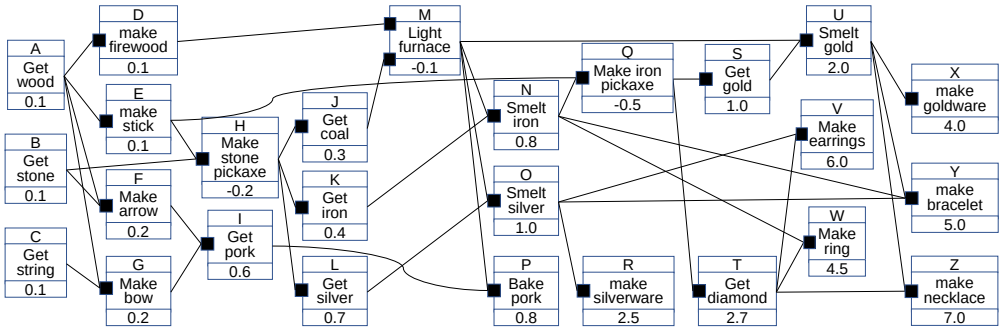

Figure 6: The entire graph of Mining domain. Based on this graph, we generated 640 subtask graphs by removing the subtask node that has no parent node.

C.2 PLAYGROUND DOMAIN

Nodes	N_T	number of tasks in each layer
	N_D	number of distractors in each layer
	N_A	number of AND node in each layer
	r	reward of subtasks in each layer
Edges	N_{ac}^{+}	number of children of AND node in each layer
	N_{ac}	number of children of AND node with NOT connection in each layer
	N_{dp}	number of parents with NOT connection of distractors in each layer
	N_{oc}	number of children of OR node in each layer
Episode	N_{step}	number of step given for each episode

Table 2: Parameters for generating task including subtask graph parameter and episode length.

For training and test sample generation, the subtask graph structure was defined in terms of the parameters in table 2. To cover wide range of subtask graphs, we randomly sampled the parameters $N_A; N_O; N_{ac}^{+}; N_{ac}; N_{dc}$, and N_{oc} from the range specified in the table 3, while N_T and N_D was manually set. We prevented the graph from including the duplicated AND nodes with the same children node(s). We carefully set the range of each parameter such that at least 500 different subtask graphs can be generated with the given parameter ranges. The table 3 summarizes parameters used to generate training and evaluation subtask graphs for the Playground domain.

D DETAILS OF GRPROP POLICY

Intuitively, GRProp policy modifies the subtask graph to a differentiable form such that we can compute the gradient of modified return with respect to the subtask completion vector in order to measure how much each subtask is likely to increase the modified return. Specifically, the logical AND, OR, and NOT operations in Equations 6, 7, and 8 are substituted by the smoothed counterparts as follows:

$$p^i = {}_{\text{or}}e^i + (1 - {}_{\text{or}})\, x^i; \tag{9}$$

$$e^i = \underset{j2Child_i}{\text{OR}} \; p^j_{AND} \; ; \tag{10}$$

$$p^j_{AND} = \underset{k2Child_j}{\text{AND}} \; x^{j,k} \; ; \tag{11}$$

$$x^{j,k} = w^{j,k}p^k + (1 - w^{j,k})\text{NOT} \; p^k \; ; \tag{12}$$

Train (=**D1**)	N_T	{6,4,2,1}
	N_D	{2,1,0,0}
	N_A	{3,3,2}-{5,4,2}
	N_{ac}^+	{1,1,1}-{3,3,3}
	N_{ac}	{0,0,0}-{2,2,1}
	N_{dp}	{0,0,0}-{3,3,0}
	N_{oc}	{1,1,1}-{2,2,2}
	r	{0.1,0.3,0.7,1.8}-{0.2,0.4,0.9,2.0}
	N_{step}	48-72
D2	N_T	{7,5,2,1}
	N_D	{2,2,0,0}
	N_A	{4,3,2}-{5,4,2}
	N_{ac}^+	{1,1,1}-{3,3,3}
	N_{ac}	{0,0,0}-{2,2,1}
	N_{dp}	{0,0,0,0}-{3,3,0,0}
	N_{oc}	{1,1,1}-{2,2,2}
	r	{0.1,0.3,0.7,1.8}-{0.2,0.4,0.9,2.0}
	N_{step}	52-78
D3	N_T	{5,4,4,2,1}
	N_D	{1,1,1,0,0}
	N_A	{3,3,3,2}-{5,4,4,2}
	N_{ac}^+	{1,1,1,1}-{3,3,3,3}
	N_{ac}	{0,0,0,0}-{2,2,1,1}
	N_{dp}	{0,0,0,0,0}-{3,3,3,0,0}
	N_{oc}	{1,1,1,1}-{2,2,2,2}
	r	{0.1,0.3,0.6,1.0,2.0}-{0.2,0.4,0.7,1.2,2.2}
	N_{step}	56-84
D4	N_T	{4,3,3,3,2,1}
	N_D	{0,0,0,0,0,0}
	N_A	{3,3,3,3,2}-{5,4,4,4,2}
	N_{ac}^+	{1,1,1,1,1}-{3,3,3,3,3}
	N_{ac}	{0,0,0,0,0}-{2,2,1,1,0}
	N_{dp}	{0,0,0,0,0,0}-{0,0,0,0,0,0}
	N_{oc}	{1,1,1,1,1}-{2,2,2,2,2}
	r	{0.1,0.3,0.6,1.0,1.4,2.4}-{0.2,0.4,0.7,1.2,1.6,2.6}
	N_{step}	56-84

Table 3: Subtask graph parameters for training set and tasks **D1**∼**D4**.

where $\mathsf{x} \in \mathsf{R}^d$ is the input vector,

$$\mathrm{OR}(\mathsf{x}) = \mathrm{softmax}(w_{\mathrm{or}}\mathsf{x}) \cdot \mathsf{x}; \tag{13}$$

$$\mathrm{AND}(\mathsf{x}) = \max(0;\ (\mathsf{x}; w_{\mathrm{and}}) - |\mathsf{x}| + 1); \tag{14}$$

$$\mathrm{NOT}(\mathsf{x}) = -w_{\mathrm{not}}\mathsf{x}; \tag{15}$$

$|\mathsf{x}| = d$, $(\mathsf{x};\ ) = \frac{1}{\beta}\log(1 + \exp(\ \mathsf{x}))$ is a soft-plus function, and $_{\mathrm{or}} = 0.6; w_{\mathrm{or}} = 2; w_{\mathrm{and}} = 3; w_{\mathrm{not}} = 2$ are the hyper-parameters of GRProp. With the smoothed operations, the sum of smoothed and modified reward is given as:

$$\theta_t = \mathsf{r}^T\mathsf{p}. \tag{16}$$

Finally, the graph reward propagation policy is a softmax policy,

$$(\mathsf{o}_t|G; \mathsf{x}_t) = \mathrm{Softmax}\ \ \nabla_{\mathbf{x}_t}\theta_t\ \ = \mathrm{Softmax}\ \ T\ \ _{\mathrm{or}}\mathsf{r}^T + T(1 - \ _{\mathrm{or}})\mathsf{r}^T\nabla_{\mathbf{x}_t}\mathsf{e}_t\ \ ; \tag{17}$$

where we used the softmax temperature $T = 40$.

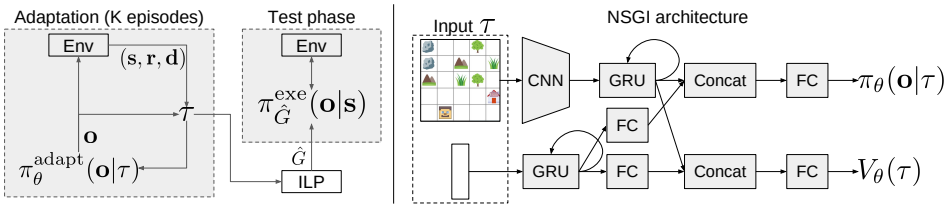

Figure 7: (Left) Our NSGI model and (Right) the architecture of adaptation policy $\pi_\theta^{\text{adapt}}$.

E Details of NSGI architecture

Our NSGI architecture encodes the observation input using CNN module. Specifically, the observation embedding is computed by Conv1(16x1x1-1/0)-Conv2(32x3x3-1/0)-Conv3(64x3x3-1/1)-Conv4(32x3x3-1/1)-Flatten-FC(256)-GRU(256). Other inputs are all concatenated into a single vector, and fed to GRU(256). In turn, we extracted two flat embeddings using two separate FC(256) heads for policy and value outputs. For each output, the observation and flat embeddings and concatenated into single vector, and fed to FC(256)-FC(d) for policy output and FC(256)-FC(1) for value output, where d is the policy dimension. We used ReLU activation function in all the layers.

F Details of training **NSGI-Meta**

Algorithm 1 Adaptation policy optimization

Require: $p(G)$: distribution over subtask graph
Require: : step size hyperparameter
while not done **do**
  Sample batch of task parameters $\{G_i\}_{i=1}^{M} \sim p(G)$
  for $i = 1, \ldots, M$ **do**
    Rollout K episodes = $\{s_t, o_t, r_t, d_t\}_{t=1}^{H} \sim$ $_\theta^{\text{adapt}}$ in task $\mathcal{M}_{G_i}$. adaptation phase
    if UCB bonus **then**
      Compute $r_t^{\text{UCB}} = w_{\text{UCB}} \mathbb{I}(x_t \in \ _{t\ 1})$
    else
      $\hat{G}_i$ = ILP( ) . task inference
      Sample $^{0} \sim$ $_{\hat{G}_i}^{\text{GRProp}}$ in task $\mathcal{M}_{G_i}$. test phase
  if UCB bonus **then**
    Update $\leftarrow$ $-$ ∇_θ $\left[\sum_{i=1}^{M} \mathcal{L}^{\text{UCB}}_{\mathcal{M}_{G_i}}\right.$ $_\theta^{\text{adapt}}$ using $\mathcal{L}^{\text{UCB}}(\ _\theta) = -\mathbb{E}_\pi$ $\left[\sum_{t=1}^{H} r_t^{\text{UCB}}\right]$
  else
    Update $\leftarrow$ $-$ ∇_θ $\left[\sum_{i=1}^{M} \mathcal{L}_{\mathcal{M}_{G_i}}\right.$ $_{\hat{G}_i}^{\text{GRProp}}$ using $\mathcal{L}(\ _\theta) = -\mathbb{E}_{\pi\ ;\ H+1}$ $\left[\sum_{t=1}^{H^0} r_t^{0}\right]$

Table 1 describes the pseudo-code for training our **NSGI-Meta** model with and without UCB bonus term. In adaptation phase, we ran a batch of 48 parallel environments. In test phase, we measured the average performance over 4 episodes with 8 parallel workers (i.e., average over 32 episodes). We used actor-critic method with GAE (Schulman et al., 2015) as follows:

$$\nabla_\theta \mathcal{L} = \mathbb{E}_{G\ G_{train}}\ \mathbb{E}_{s\ \pi}\ -\nabla_\theta \log\ _\theta \left[\sum_{l=0}^{\ } \prod_{n=0}^{\ l} (\ )^{k_n}\ _{t+l}\right]; \tag{18}$$

$$_t = r_t + \ ^{k_t} V_\theta^\pi(s_{t+1}) - V_\theta^\pi(s_t); \tag{19}$$

where we used the learning rate = 0.002, = 1, and = 0.9. We used RMSProp optimizer with the smoothing parameter of 0.99 and epsilon of 1e-5. We trained our **NSGI-Meta** agent for 8000 trials, where the agent is updated after every trial. We also used the entropy regularization with annealed parameter . We started from = 0.05 and linearly decreased it after 1200 trials until it reaches = 0 at 3200 trials. During training, we update the critic network to minimize

$\mathsf{E}\left[(R_t - V_\theta^\pi(\mathsf{s}_t))^2\right]$, where R_t is the cumulative reward at time t with the weight of 0.03. We clipped the magnitude of gradient to be no larger than 1.

G DETAILS OF TRAINING RL^2 AND HRL

For training **RL**^{2} and **HRL**, we used the same architecture and algorithm with **NSGI-Meta**. For **RL**^{2}, we used the same hyper-parameters except the learning rate $= 0.001$ and the critic loss weight of 0.005. For **HRL**, we used the learning rate $= 0.001$ and the critic loss weight of 0.12. We used the best hyper-parameters chosen from the same candidate hyper-parameter set for all the agents.