Zero-Shot Task Generalization with Multi-Task Deep Reinforcement Learning

Junhyuk Oh [1] **Satinder Singh** [1] **Honglak Lee** [1 2] **Pushmeet Kohli** [3]

Abstract

As a step towards developing zero-shot task generalization capabilities in reinforcement learning (RL), we introduce a new RL problem where the agent should learn to execute sequences of instructions after learning useful skills that solve subtasks. In this problem, we consider two types of generalizations: to previously unseen instructions and to longer sequences of instructions. For generalization over unseen instructions, we propose a new objective which encourages learning correspondences between similar subtasks by making analogies. For generalization over sequential instructions, we present a hierarchical architecture where a meta controller learns to use the acquired skills for executing the instructions. To deal with delayed reward, we propose a new neural architecture in the meta controller that learns when to update the subtask, which makes learning more efficient. Experimental results on a stochastic 3D domain show that the proposed ideas are crucial for generalization to longer instructions as well as unseen instructions.

1. Introduction

The ability to understand and follow instructions allows us to perform a large number of new complex sequential tasks even without additional learning. For example, we can make a new dish following a recipe, and explore a new city following a guidebook. Developing the ability to execute instructions can potentially allow reinforcement learning (RL) agents to generalize quickly over tasks for which such instructions are available. For example, factory-trained household robots could execute novel tasks in a new house following a human user's instructions (e.g., tasks involving household chores, going to a new place, picking up/manipulating new objects, etc.). In addition to generalization over instructions, an intelligent agent should also be able to handle unexpected events (e.g., low battery, arrivals of reward sources) while executing instructions. Thus, the agent should not blindly execute instructions sequentially but sometimes deviate from instructions depending on circumstances, which requires balancing between two different objectives.

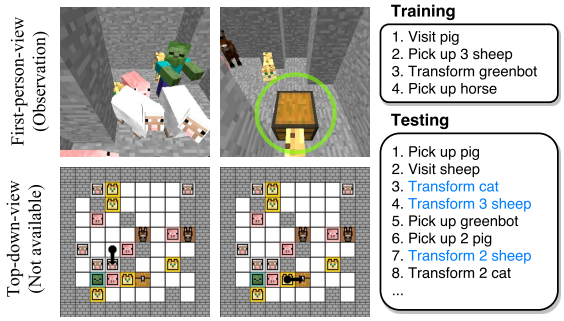

Figure 1: Example of 3D world and instructions. The agent is tasked to execute longer sequences of instructions in the correct order after training on short sequences of instructions; in addition, previously unseen instructions can be given during evaluation (blue text). Additional reward is available from randomly appearing boxes regardless of instructions (green circle).

Problem. To develop such capabilities, this paper introduces the instruction execution problem where the agent's overall task is to execute a given list of instructions described by a simple form of natural language while dealing with unexpected events, as illustrated in Figure 1. More specifically, we assume that each instruction can be executed by performing one or more high-level subtasks in sequence. Even though the agent can pre-learn skills to perform such subtasks (e.g., [Pick up, Pig] in Figure 1), and the instructions can be easily translated to subtasks, our problem is difficult due to the following challenges.

- *Generalization*: Pre-training of skills can only be done on a subset of subtasks, but the agent is required to perform previously unseen subtasks (e.g., going to a new place) in order to execute unseen instructions during testing. Thus, the agent should learn to generalize to new subtasks in the skill learning stage. Furthermore, the agent is required to execute previously unseen and longer sequences of instructions during evaluation.
- *Delayed reward*: The agent is *not* told which instruction to execute at any point of time from the environment but just given the full list of instructions as input. In addition, the agent does *not* receive any signal on completing in-

[1]University of Michigan [2]Google Brain [3]Microsoft Research. Correspondence to: Junhyuk Oh <junhyuk@umich.edu>.

Proceedings of the 34^{th} International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017.

dividual instructions from the environment, i.e., success-reward is provided only when all instructions are executed correctly. Therefore, the agent should keep track of which instruction it is executing and decide when to move on to the next instruction.

- *Interruption*: As described in Figure 1, there can be unexpected events in uncertain environments, such as opportunities to earn bonuses (e.g., windfalls), or emergencies (e.g., low battery). It can be better for the agent to *interrupt* the ongoing subtask before it is finished, perform a different subtask to deal with such events, and resume executing the interrupted subtask in the instructions after that. Thus, the agent should achieve a balance between executing instructions and dealing with such events.
- *Memory*: There are loop instructions (e.g., "Pick up 3 pig") which require the agent to perform the same subtask ([Pick up, Pig]) multiple times and take into account the history of observations and subtasks in order to decide when to move on to the next instruction correctly.

Due to these challenges, the agent should be able to execute a novel subtask, keep track of what it has done, monitor observations to interrupt ongoing subtasks depending on circumstances, and switch to the next instruction precisely when the current instruction is finished.

Our Approach and Technical Contributions. To address the aforementioned challenges, we divide the learning problem into two stages: 1) learning skills to perform a set of subtasks and generalizing to unseen subtasks, and 2) learning to execute instructions using the learned skills. Specifically, we assume that subtasks are defined by several disentangled parameters. Thus, in the first stage our architecture learns a *parameterized skill* (da Silva et al., 2012) to perform different subtasks depending on input parameters. In order to generalize over unseen parameters, we propose a new objective function that encourages making analogies between similar subtasks so that the underlying manifold of the entire subtask space can be learned without experiencing all subtasks. In the second stage, our architecture learns a meta controller on top of the parameterized skill so that it can read instructions and decide which subtask to perform. The overall hierarchical RL architecture is shown in Figure 3. To deal with delayed reward as well as interruption, we propose a novel neural network (see Figure 4) that learns when to update the subtask in the meta controller. This not only allows learning to be more efficient under delayed reward by operating at a larger time-scale but also allows interruptions of ongoing subtasks when an unexpected event is observed.

Main Results. We developed a 3D visual environment using Minecraft based on Oh et al. (2016) where the agent can interact with many objects. Our results on multiple sets of parameterized subtasks show that our proposed analogy-making objective can generalize successfully. Our results on multiple instruction execution problems show that our meta controller's ability to learn when to update the subtask plays a key role in solving the overall problem and outperforms several hierarchical baselines. The demo videos are available at the following website: https://sites.google.com/a/umich.edu/junhyuk-oh/task-generalization.

The rest of the sections are organized as follows. Section 2 presents related work. Section 3 presents our analogy-making objective for generalization to parameterized tasks and demonstrates its application to different generalization scenarios. Section 4 presents our hierarchical architecture for the instruction execution problem with our new neural network that learns to operate at a large time-scale. In addition, we demonstrate our agent's ability to generalize over sequences of instructions, as well as provide a comparison to several alternative approaches.

2. Related Work

Hierarchical RL. A number of hierarchical RL approaches are designed to deal with sequential tasks. Typically these have the form of a meta controller and a set of lower-level controllers for subtasks (Sutton et al., 1999; Dietterich, 2000; Parr and Russell, 1997; Ghavamzadeh and Mahadevan, 2003; Konidaris et al., 2012; Konidaris and Barto, 2007). However, much of the previous work assumes that the overall task is fixed (e.g., *Taxi* domain (Dietterich, 2000)). In other words, the optimal sequence of subtasks is fixed during evaluation (e.g., picking up a passenger followed by navigating to a destination in the Taxi domain). This makes it hard to evaluate the agent's ability to compose pre-learned policies to solve previously unseen sequential tasks in a zero-shot fashion unless we re-train the agent on the new tasks in a transfer learning setting (Singh, 1991; 1992; McGovern and Barto, 2002). Our work is also closely related to Programmable HAMs (PHAMs) (Andre and Russell, 2000; 2002) in that a PHAM is designed to execute a given program. However, the program explicitly specifies the policy in PHAMs which effectively reduces the state-action search space. In contrast, instructions are a description of the task in our work, which means that the agent should learn to use the instructions to maximize its reward.

Hierarchical Deep RL. Hierarhical RL has been recently combined with deep learning. Kulkarni et al. (2016) proposed hierarchical Deep Q-Learning and demonstrated improved exploration in a challenging Atari game. Tessler et al. (2017) proposed a similar architecture, but the high-level controller is allowed to choose primitive actions directly. Bacon et al. (2017) proposed the *option-critic* architecture, which learns options without pseudo reward and demonstrated that it can learn distinct options in Atari

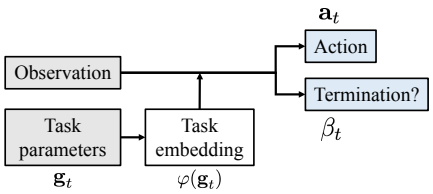

Figure 2: Architecture of parameterized skill. See text for details.

games. Heess et al. (2016) formulated the actions of the meta controller as continuous variables that are used to modulate the behavior of the low-level controller. Florensa et al. (2017) trained a stochastic neural network with mutual information regularization to discover skills. Most of these approaches build an *open-loop* policy at the high-level controller that waits until the previous subtask is finished once it is chosen. This approach is not able to interrupt ongoing subtasks in principle, while our architecture can switch its subtask at any time.

Zero-Shot Task Generalization. There have been a few papers on zero-shot generalization to new tasks. For example, da Silva et al. (2012) introduced parameterized skills that map sets of task descriptions to policies. Isele et al. (2016) achieved zero-shot task generalization through dictionary learning with sparsity constraints. Schaul et al. (2015) proposed *universal value function approximators* (UVFAs) that learn value functions for state and goal pairs. Devin et al. (2017) proposed composing sub-networks that are shared across tasks and robots in order to achieve generalization to unseen configurations of them. Unlike the above prior work, we propose a flexible metric learning method which can be applied to various generalization scenarios. Andreas et al. (2016) proposed a framework to learn the underlying subtasks from a *policy sketch* which specifies a sequence of subtasks, and the agent can generalize over new sequences of them in principle. In contrast, our work aims to generalize over unseen subtasks as well as unseen sequences of them. In addition, the agent should handle unexpected events in our problem that are not described by the instructions by interrupting subtasks appropriately.

Instruction Execution. There has been a line of work for building agents that can execute natural language instructions: Tellex et al. (2011; 2014) for robotics and MacMahon et al. (2006); Chen and Mooney (2011); Mei et al. (2015) for a simulated environment. However, these approaches focus on natural language understanding to map instructions to actions or *groundings* in a supervised setting. In contrast, we focus on generalization to sequences of instructions without any supervision for language understanding or for actions. Although Branavan et al. (2009) also tackle a similar problem, the agent is given a single instruction at a time, while our agent needs to learn how to align instructions and state given a full list of instructions.

3. Learning a Parameterized Skill

In this paper, a parameterized skill is a multi-task policy corresponding to multiple tasks defined by categorical input *task parameters*, e.g., [Pick up, X]. More formally, we define a parameterized skill as a mapping $\mathcal{O} \times \mathcal{G} \rightarrow \mathcal{A} \times \mathcal{B}$, where $\mathcal{O}$ is a set of observations, $\mathcal{G}$ is a set of task parameters, $\mathcal{A}$ is a set of primitive actions, and $\mathcal{B} = \{0, 1\}$ indicates whether the task is finished or not. A space of tasks is defined using the Cartesian product of task parameters: $\mathcal{G} = \mathcal{G}^{(1)} \times ... \times \mathcal{G}^{(n)}$, where $\mathcal{G}^{(i)}$ is a set of the i-th parameters (e.g., $\mathcal{G} = \{\text{Visit, Pick up}\} \times \{\text{X, Y, Z}\}$). Given an observation $\mathbf{x}_t \in \mathcal{O}$ at time t and task parameters $\mathbf{g} = \left[g^{(1)}, ..., g^{(n)}\right] \in \mathcal{G}$, where $g^{(i)}$ is a one-hot vector, the parameterized skill is the following functions:

$$\text{Policy: } \pi_\phi(\mathbf{a}_t | \mathbf{x}_t, \mathbf{g})$$
$$\text{Termination: } \beta_\phi(b_t | \mathbf{x}_t, \mathbf{g}),$$

where π_ϕ is the policy optimized for the task $\mathbf{g}$, and β_ϕ is a termination function (Sutton et al., 1999) which is the probability that the state is terminal at time t for the given task $\mathbf{g}$. The parameterized skill is represented by a non-linear function approximator $\phi(\cdot)$, a neural network in this paper. The neural network architecture of our parameterized skill is illustrated in Figure 2. The network maps input task parameters into a task embedding space $\varphi(\mathbf{g})$, which is combined with the observation followed by the output layers. More details are described in the supplementary material.

3.1. Learning to Generalize by Analogy-Making

Only a subset of tasks ($\mathcal{G}' \subset \mathcal{G}$) are available during training, and so in order to generalize to unseen tasks during evaluation the network needs to learn knowledge about the relationship between different task parameters when learning the task embedding $\varphi(\mathbf{g})$.

To this end, we propose an analogy-making objective inspired by Reed et al. (2015). The main idea is to learn correspondences between tasks. For example, if target objects and 'Visit/Pick up' actions are *independent* (i.e., each action can be applied to any target object), we can enforce the analogy [Visit, X] : [Visit, Y] :: [Pick up, X] : [Pick up, Y] for any X and Y in the embedding space, which means that the difference between 'Visit' and 'Pick up' is consistent regardless of target objects and vice versa. This allows the agent to generalize to unseen combinations of actions and target objects, such as performing [Pick up, Y] after it has learned to perform [Pick up, X] and [Visit, Y].

More specifically, we define several constraints as follows:

$$\begin{aligned} \|\Delta(\mathbf{g}_A, \mathbf{g}_B) - \Delta(\mathbf{g}_C, \mathbf{g}_D)\| &\approx 0 && \text{if } \mathbf{g}_A : \mathbf{g}_B :: \mathbf{g}_C : \mathbf{g}_D \\ \|\Delta(\mathbf{g}_A, \mathbf{g}_B) - \Delta(\mathbf{g}_C, \mathbf{g}_D)\| &\geq \tau_{dis} && \text{if } \mathbf{g}_A : \mathbf{g}_B \neq \mathbf{g}_C : \mathbf{g}_D \\ \|\Delta(\mathbf{g}_A, \mathbf{g}_B)\| &\geq \tau_{diff} && \text{if } \mathbf{g}_A \neq \mathbf{g}_B, \end{aligned}$$

where $\mathbf{g}_k = \left[g_k^{(1)}, g_k^{(2)}, ..., g_k^{(n)}\right] \in \mathcal{G}$ are task parameters,

$\Delta(\mathbf{g}_A, \mathbf{g}_B) = \varphi(\mathbf{g}_A) - \varphi(\mathbf{g}_B)$ is the difference vector between two tasks in the embedding space, and τ_{dis} and τ_{diff} are constant threshold distances. Intuitively, the first constraint enforces the analogy (i.e., *parallelogram* structure in the embedding space; see Mikolov et al. (2013); Reed et al. (2015)), while the other constraints prevent trivial solutions. We incorporate these constraints into the following objectives based on contrastive loss (Hadsell et al., 2006):

$$\mathcal{L}_{sim} = \mathbb{E}_{\mathbf{g}_{A\ldots D} \sim \mathcal{G}_{sim}} \left[\| \Delta(\mathbf{g}_A, \mathbf{g}_B) - \Delta(\mathbf{g}_C, \mathbf{g}_D) \|^2 \right]$$

$$\mathcal{L}_{dis} = \mathbb{E}_{\mathbf{g}_{A\ldots D} \sim \mathcal{G}_{dis}} \left[(\tau_{dis} - \| \Delta(\mathbf{g}_A, \mathbf{g}_B) - \Delta(\mathbf{g}_C, \mathbf{g}_D) \|)_+^2 \right]$$

$$\mathcal{L}_{diff} = \mathbb{E}_{\mathbf{g}_{A,B} \sim \mathcal{G}_{diff}} \left[(\tau_{diff} - \| \Delta(\mathbf{g}_A, \mathbf{g}_B) \|)_+^2 \right],$$

where $(\cdot)_+ = \max(0, \cdot)$ and $\mathcal{G}_{sim}, \mathcal{G}_{dis}, \mathcal{G}_{diff}$ are sets of task parameters that satisfy corresponding conditions in the above three constraints. The final analogy-making objective is the weighted sum of the above three objectives.

3.2. Training

The parameterized skill is trained on a set of tasks ($\mathcal{G}' \subset \mathcal{G}$) through the actor-critic method with generalized advantage estimation (Schulman et al., 2016). We also found that pre-training through *policy distillation* (Rusu et al., 2016; Parisotto et al., 2016) gives slightly better results as discussed in Tessler et al. (2017). Throughout training, the parameterized skill is also made to predict whether the current state is terminal or not through a binary classification objective, and the analogy-making objective is applied to the task embedding separately. The full details of the learning objectives are described in the supplementary material.

3.3. Experiments

Environment. We developed a 3D visual environment using Minecraft based on Oh et al. (2016) as shown in Figure 1. An observation is represented as a 64×64 pixel RGB image. There are 7 different types of objects: *Pig*, *Sheep*, *Greenbot*, *Horse*, *Cat*, *Box*, and *Ice*. The topology of the world and the objects are randomly generated for every episode. The agent has 9 actions: *Look* (Left/Right/Up/Down), *Move* (Forward/Backward), *Pick up*, *Transform*, and *No operation*. *Pick up* removes the object in front of the agent, and *Transform* changes the object in front of the agent to ice (a special object).

Implementation Details. The network architecture of the parameterized skill consists of 4 convolution layers and one LSTM (Hochreiter and Schmidhuber, 1997) layer. We conducted curriculum training by changing the size of the world, the density of object and walls according to the agent's success rate. We implemented actor-critic method with 16 CPU threads based on Sukhbaatar et al. (2015). The parameters are updated after 8 episodes for each thread. The details of architectures and hyperparameters are described in the supplementary material.

Scenario	Analogy	Train	Unseen
Independent	×	**0.3** (99.8%)	-3.7 (34.8%)
	✓	**0.3** (99.8%)	**0.3** (99.5%)
Object-dependent	×	**0.3** (99.7%)	-5.0 (2.2%)
	✓	**0.3** (99.8%)	**0.3** (99.7%)
Inter/Extrapolation	×	**-0.7** (97.5%)	-2.2 (24.9%)
	✓	**-0.7** (97.5%)	**-1.7** (94.5%)

Table 1: Performance on parameterized tasks. Each entry shows 'Average reward (Success rate)'. We assume an episode is successful only if the agent successfully finishes the task and its termination predictions are correct throughout the whole episode.

Results. To see how useful analogy-making is for generalization to unseen parameterized tasks, we trained and evaluated the parameterized skill on three different sets of parameterized tasks defined below[1].

- **Independent**: The task space is defined as $\mathcal{G} = \mathcal{T} \times \mathcal{X}$, where $\mathcal{T} = \{\text{Visit}, \text{Pick up}, \text{Transform}\}$ and $\mathcal{X}$ is the set of object types. The agent should move on top of the target object given 'Visit' task and perform the corresponding actions in front of the target given 'Pick up' and 'Transform' tasks. Only a subset of tasks are encountered during training, so the agent should generalize over unseen configurations of task parameters.
- **Object-dependent**: The task space is defined as $\mathcal{G} = \mathcal{T}' \times \mathcal{X}$, where $\mathcal{T}' = \mathcal{T} \cup \{\text{Interact with}\}$. We divided objects into two groups, each of which should be either picked up or transformed given 'Interact with' task. Only a subset of target object types are encountered during training, so there is no chance for the agent to generalize without knowledge of the group of each object. We applied analogy-making so that analogies can be made only within the same group. This allows the agent to perform object-dependent actions even for unseen objects.
- **Interpolation/Extrapolation**: The task space is defined as $\mathcal{G} = \mathcal{T} \times \mathcal{X} \times \mathcal{C}$, where $\mathcal{C} = \{1, 2, ..., 7\}$. The agent should perform a task for a given number of times ($c \in \mathcal{C}$). Only $\{1, 3, 5\} \subset \mathcal{C}$ is given during training, and the agent should generalize over unseen numbers $\{2, 4, 6, 7\}$. Note that the optimal policy for a task can be derived from $\mathcal{T} \times \mathcal{X}$, but predicting termination requires generalization to unseen numbers. We applied analogy-making based on arithmetic (e.g., [Pick up, X, 2] : [Pick up, X, 5] :: [Transform, Y, 3] : [Transform, Y, 6]).

As summarized in Table 1, the parameterized skill with our analogy-making objective can successfully generalize to unseen tasks in all generalization scenarios. This suggests that when learning a representation of task parameters, it is possible to inject prior knowledge in the form of the analogy-making objective so that the agent can learn to

[1]The sets of subtasks used for training and evaluation are described in the supplementary material.

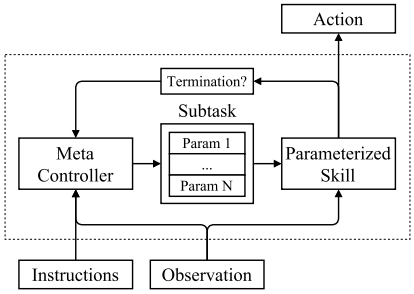

Figure 3: Overview of our hierarchical architecture.

generalize over unseen tasks in various ways depending on semantics or context without needing to experience them.

4. Learning to Execute Instructions using Parameterized Skill

We now consider the instruction execution problem where the agent is given a sequence of simple natural language instructions, as illustrated in Figure 1. We assume an already trained parameterized skill, as described in Section 3. Thus, the main remaining problem is how to use the parameterized skill to execute instructions. Although the requirement that instructions be executed sequentially makes the problem easier (than, e.g., conditional-instructions), the agent still needs to make complex decisions because it should deviate from instructions to deal with unexpected events (e.g., low battery) and remember what it has done to deal with loop instructions, as discussed in Section 1.

To address the above challenges, our hierarchical RL architecture (see Figure 3) consists of two modules: meta controller and parameterized skill. Specifically, a meta controller reads the instructions and passes subtask parameters to a parameterized skill which executes the given subtask and provides its termination signal back to the meta controller. Section 4.1 describes the overall architecture of the meta controller for dealing with instructions. Section 4.2 describes a novel neural architecture that learns when to update the subtask in order to better deal with delayed reward signal as well as unexpected events.

4.1. Meta Controller Architecture

As illustrated in Figure 4, the meta controller is a mapping $\mathcal{O} \times \mathcal{M} \times \mathcal{G} \times \mathcal{B} \rightarrow \mathcal{G}$, where $\mathcal{M}$ is a list of instructions. Intuitively, the meta controller decides subtask parameters $\mathbf{g}_t \in \mathcal{G}$ conditioned on the observation $\mathbf{x}_t \in \mathcal{O}$, the list of instructions $M \in \mathcal{M}$, the previously selected subtask $\mathbf{g}_{t-1}$, and its termination signal ($b \sim \beta_\phi$).

In contrast to recent hierarchical deep RL approaches where the meta controller can update its subtask (or option) only when the previous one terminates or only after a fixed number of steps, our meta controller can update the subtask at any time and takes the termination signal as additional input. This gives more flexibility to the meta controller and enables interrupting ongoing tasks before termination.

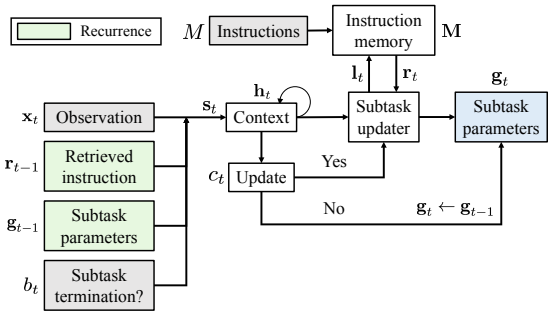

Figure 4: Neural network architecture of meta controller.

In order to keep track of the agent's progress on instruction execution, the meta controller maintains its internal state by computing a *context* vector (Section 4.1.1) and determines which subtask to execute by focusing on one instruction at a time from the list of instructions (Section 4.1.2).

4.1.1. Context

Given the sentence embedding $\mathbf{r}_{t-1}$ retrieved at the previous time-step from the instructions (described in Section 4.1.2), the previously selected subtask $\mathbf{g}_{t-1}$, and the subtask termination $b_t \sim \beta_\phi\left(b_t|\mathbf{s}_t, \mathbf{g}_{t-1}\right)$, the meta controller computes the context vector ($\mathbf{h}_t$) as follows:

$$\mathbf{h}_t = \text{LSTM}\left(\mathbf{s}_t, \mathbf{h}_{t-1}\right)$$
$$\mathbf{s}_t = f\left(\mathbf{x}_t, \mathbf{r}_{t-1}, \mathbf{g}_{t-1}, b_t\right),$$

where f is a neural network. Intuitively, $\mathbf{g}_{t-1}$ and b_t provide information about which subtask was being solved by the parameterized skill and whether it has finished or not. Thus, $\mathbf{s}_t$ is a summary of the current observation and the ongoing subtask. $\mathbf{h}_t$ takes the history of $\mathbf{s}_t$ into account through the LSTM, which is used by the subtask updater.

4.1.2. Subtask Updater

The *subtask updater* constructs a memory structure from the list of instructions, retrieves an instruction by maintaining a pointer into the memory, and computes the subtask parameters.

Instruction Memory. Given instructions as a list of sentences $M = (\mathbf{m}_1, \mathbf{m}_2, ..., \mathbf{m}_K)$, where each sentence consists of a list of words, $\mathbf{m}_i = \left(w_1, ..., w_{|\mathbf{m}_i|}\right)$, the subtask updater constructs memory blocks $\mathbf{M} \in \mathbb{R}^{E \times K}$ (i.e., each column is an E-dimensional embedding of a sentence). The subtask updater maintains an *instruction pointer* ($\mathbf{p}_t \in \mathbb{R}^K$) which is non-negative and sums up to 1 indicating which instruction the meta controller is executing. Memory construction and retrieval can be written as:

$$\text{Memory: } \mathbf{M} = \left[\varphi^w\left(\mathbf{m}_1\right), \varphi^w\left(\mathbf{m}_2\right), ..., \varphi^w\left(\mathbf{m}_K\right)\right] \quad (1)$$
$$\text{Retrieval: } \mathbf{r}_t = \mathbf{M}\mathbf{p}_t, \quad (2)$$

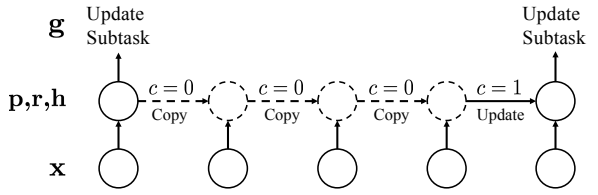

Figure 5: Unrolled illustration of the meta controller with a learned time-scale. The internal states $(\mathbf{p}, \mathbf{r}, \mathbf{h})$ and the subtask $(\mathbf{g})$ are updated only when $c = 1$. If $c = 0$, the meta controller continues the previous subtask without updating its internal states.

where $\varphi^w(\mathbf{m}_i) \in \mathbb{R}^E$ is the embedding of the i-th sentence (e.g., Bag-of-words), and $\mathbf{r}_t \in \mathbb{R}^E$ is the retrieved sentence embedding which is used for computing the subtask parameters. Intuitively, if $\mathbf{p}_t$ is a one-hot vector, $\mathbf{r}_t$ indicates a single instruction from the whole list of instructions. The meta controller should learn to manage $\mathbf{p}_t$ so that it can focus on the correct instruction at each time-step.

Since instructions should be executed sequentially, we use a location-based memory addressing mechanism (Zaremba and Sutskever, 2015; Graves et al., 2014) to manage the instruction pointer. Specifically, the subtask updater shifts the instruction pointer by $[-1, 1]$ as follows:

$$\mathbf{p}_t = \mathbf{l}_t * \mathbf{p}_{t-1} \text{ where } \mathbf{l}_t = \text{Softmax}\left(\varphi^{shift}(\mathbf{h}_t)\right), \quad (3)$$

where $*$ is a convolution operator, φ^{shift} is a neural network, and $\mathbf{l}_t \in \mathbb{R}^3$ is a soft-attention vector over the three shift operations $\{-1, 0, +1\}$. The optimal policy should keep the instruction pointer unchanged while executing an instruction and increase the pointer by +1 precisely when the current instruction is finished.

Subtask Parameters. The subtask updater takes the context $(\mathbf{h}_t)$, updates the instruction pointer $(\mathbf{p}_t)$, retrieves an instruction $(\mathbf{r}_t)$, and computes subtask parameters as:

$$\pi_\theta(\mathbf{g}_t|\mathbf{h}_t, \mathbf{r}_t) = \prod_i \pi_\theta\left(g_t^{(i)}|\mathbf{h}_t, \mathbf{r}_t\right), \quad (4)$$

where $\pi_\theta\left(g_t^{(i)}|\mathbf{h}_t, \mathbf{r}_t\right) \propto \exp\left(\varphi_i^{goal}(\mathbf{h}_t, \mathbf{r}_t)\right)$, and φ_i^{goal} is a neural network for the i-th subtask parameter.

4.2. Learning to Operate at a Large Time-Scale

Although the meta controller can learn an optimal policy by updating the subtask at each time-step in principle, making a decision at every time-step can be inefficient because subtasks do not change frequently. Instead, having temporally-extended actions can be useful for dealing with delayed reward by operating at a larger time-scale (Sutton et al., 1999). While it is reasonable to use the subtask termination signal to define the temporal scale of the meta controller as in many recent hierarchical deep RL approaches (see Section 2), this approach would result in a mostly open-loop meta-controller policy that is not able to interrupt ongoing subtasks before termination, which is necessary to deal with unexpected events not specified in the instructions.

To address this dilemma, we propose to learn the time-scale of the meta controller by introducing an internal binary decision which indicates whether to *invoke* the subtask updater to update the subtask or not, as illustrated in Figure 5. This decision is defined as: $c_t \sim \sigma\left(\varphi^{update}(\mathbf{s}_t, \mathbf{h}_{t-1})\right)$ where σ is a sigmoid function. If $c_t = 0$, the meta controller continues the current subtask without updating the subtask updater. Otherwise, if $c_t = 1$, the subtask updater updates its internal states (e.g., instruction pointer) and the subtask parameters. This allows the subtask updater to operate at a large time-scale because one decision made by the subtask updater results in multiple actions depending on c values. The overall meta controller architecture with this update scheme is illustrated in Figure 4.

Soft-Update. To ease optimization of the non-differentiable variable (c_t), we propose a *soft-update* rule by using $c_t = \sigma\left(\varphi^{update}(\mathbf{s}_t, \mathbf{h}_{t-1})\right)$ instead of sampling it. The key idea is to take the weighted sum of both 'update' and 'copy' scenarios using c_t as the weight. This method is described in Algorithm 1. We found that training the meta controller using soft-update followed by fine-tuning by sampling c_t is crucial for training the meta controller. Note that the soft-update rule reduces to the original formulation if we sample c_t and $\mathbf{l}_t$ from the Bernoulli and multinomial distributions, which justifies our initialization trick.

Algorithm 1 Subtask update (Soft)

Input: $\mathbf{s}_t, \mathbf{h}_{t-1}, \mathbf{p}_{t-1}, \mathbf{r}_{t-1}, \mathbf{g}_{t-1}$
Output: $\mathbf{h}_t, \mathbf{p}_t, \mathbf{r}_t, \mathbf{g}_t$
$c_t \leftarrow \sigma\left(\varphi^{update}(\mathbf{s}_t, \mathbf{h}_{t-1})\right)$ # Decide update weight
$\tilde{\mathbf{h}}_t \leftarrow \text{LSTM}(\mathbf{s}_t, \mathbf{h}_{t-1})$ # Update the context
$\mathbf{l}_t \leftarrow \text{Softmax}\left(\varphi^{shift}(\tilde{\mathbf{h}}_t)\right)$ # Decide shift operation
$\tilde{\mathbf{p}}_t \leftarrow \mathbf{l}_t * \mathbf{p}_{t-1}$ # Shift the instruction pointer
$\tilde{\mathbf{r}}_t \leftarrow \mathbf{M}\tilde{\mathbf{p}}_t$ # Retrieve instruction
Merge two scenarios (update/copy) using c_t as weight
$[\mathbf{p}_t, \mathbf{r}_t, \mathbf{h}_t] \leftarrow c_t[\tilde{\mathbf{p}}_t, \tilde{\mathbf{r}}_t, \tilde{\mathbf{h}}_t] + (1 - c_t)[\mathbf{p}_{t-1}, \mathbf{r}_{t-1}, \mathbf{h}_{t-1}]$
$g_t^{(i)} \sim c_t \pi_\theta\left(g_t^{(i)}|\tilde{\mathbf{h}}_t, \tilde{\mathbf{r}}_t\right) + (1 - c_t)\, g_{t-1}^{(i)} \forall i$

Integrating with Hierarchical RNN. The idea of learning the time-scale of a recurrent neural network is closely related to hierarchical RNN approaches (Koutnik et al., 2014; Chung et al., 2017) where different groups of recurrent hidden units operate at different time-scales to capture both long-term and short-term temporal information. Our idea can be naturally integrated with hierarchical RNNs by applying the update decision (c value) only for a subset of recurrent units instead of all the units. Specifically, we divide the context vector into two groups: $\mathbf{h}_t = \left[\mathbf{h}_t^{(l)}, \mathbf{h}_t^{(h)}\right]$. The low-level units $(\mathbf{h}_t^{(l)})$ are updated at every time-step,

while the high-level units ($\mathbf{h}_t^{(h)}$) are updated depending on the value of c. This simple modification leads to a form of hierarchical RNN where the low-level units focus on short-term temporal information while the high-level units capture long-term dependencies.

4.3. Training

The meta controller is trained on a training set of lists of instructions. Given a pre-trained and fixed parameterized skill, the actor-critic method is used to update the parameters of the meta controller. Since the meta controller also learns a subtask embedding $\varphi(\mathbf{g}_{t-1})$ and has to deal with unseen subtasks during evaluation, analogy-making objective is also applied. The details of the objective function are provided in the supplementary material.

4.4. Experiments

The experiments are designed to explore the following questions: (1) Will the proposed hierarchical architecture outperform a non-hierarchical baseline? (2) How beneficial is the meta controller's ability to learn when to update the subtask? We are also interested in understanding the qualitative properties of our agent's behavior.[2]

Environment. We used the same Minecraft domain used in Section 3.3. The agent receives a time penalty (-0.1) for each step and receives $+1$ reward when it finishes the entire list of instructions in the correct order. Throughout an episode, a box (including treasures) randomly appears with probability of 0.03 and transforming a box gives $+0.9$ reward.

The subtask space is defined as $\mathcal{G} = \mathcal{T} \times \mathcal{X}$, and the semantics of each subtask are the same as the 'Independent' case in Section 3.3. We used the best-performing parameterized skill throughout this experiment.

There are 7 types of instructions: {Visit X, Pick up X, Transform X, Pick up 2 X, Transform 2 X, Pick up 3 X, Transform 3 X} where 'X' is the target object type. Note that the parameterized skill used in this experiment was not trained on loop instructions (e.g., Pick up 3 X), so the last four instructions require the meta controller to learn to repeat the corresponding subtask for the given number of times. To see how the agent generalizes to previously unseen instructions, only a subset of instructions and subtasks was presented during training.

Implementation Details. The meta controller consists of 3 convolution layers and one LSTM layer. We also conducted curriculum training by changing the size of the world, the density of object and walls, and the number of instructions according to the agent's success rate. We used the actor-critic implementation described in Section 3.3.

[2]For further analysis, we also conducted comprehensive experiments on a 2D grid-world domain. However, due to space limits, those results are provided in the supplementary material.

	Train	Test (Seen)	Test (Unseen)
Length of instructions	4	20	20
Flat	-7.1 (1%)	-63.6 (0%)	-62.0 (0%)
Hierarchical-Long	-5.8 (31%)	-59.2 (0%)	-59.2 (0%)
Hierarchical-Short	-3.3 (83%)	-53.4 (23%)	-53.6 (18%)
Hierarchical-Dynamic	**-3.1** (95%)	**-30.3** (75%)	**-38.0** (56%)

Table 2: Performance on instruction execution. Each entry shows average reward and success rate. 'Hierarchical-Dynamic' is our approach that learns when to update the subtask. An episode is successful only when the agent solves all instructions correctly.

Baselines. To understand the advantage of using the hierarchical structure and the benefit of our meta controller's ability to learn when to update the subtask, we trained three baselines as follows.

- **Flat**: identical to our meta controller except that it directly chooses primitive actions without using the parameterized skill. It is also pre-trained on the training set of subtasks.
- **Hierarchical-Long**: identical to our architecture except that the meta controller can update the subtask only when the current subtask is finished. This approach is similar to recent hierarchical deep RL methods (Kulkarni et al., 2016; Tessler et al., 2017).
- **Hierarchical-Short**: identical to our architecture except that the meta controller updates the subtask at every time-step.

Overall Performance. The results on the instruction execution are summarized in Table 2 and Figure 6. It shows that our architecture ('**Hierarchical-Dynamic**') can handle a relatively long list of seen and unseen instructions of length 20 with reasonably high success rates, even though it is trained on short instructions of length 4. Although the performance degrades as the number of instructions increases, our architecture finishes 18 out of 20 seen instructions and 14 out of 20 unseen instructions on average. These results show that our agent is able to generalize to longer compositions of seen/unseen instructions by just learning to solve short sequences of a subset of instructions.

Flat vs. Hierarchy. Table 2 shows that the flat baseline completely fails even on training instructions. The flat controller tends to struggle with loop instructions (e.g., Pick up 3 pig) so that it learned a sub-optimal policy which moves to the next instruction with a small probability at each step regardless of its progress. This implies that it is hard for the flat controller to detect precisely when a subtask is finished, whereas hierarchical architectures can easily detect when a subtask is done, because the parameterized skill provides a termination signal to the meta controller.

Effect of Learned Time-Scale. As shown in Table 2 and Figure 6, 'Hierarchical-Long' baseline performs significantly worse than our architecture. We found that whenever

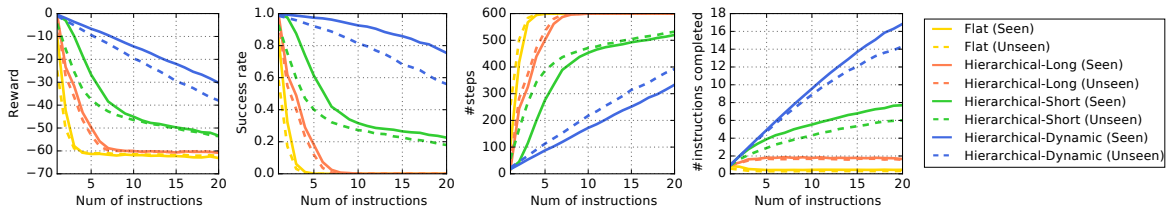

Figure 6: Performance per number of instructions. From left to right, the plots show reward, success rate, the number of steps, and the average number of instructions completed respectively.

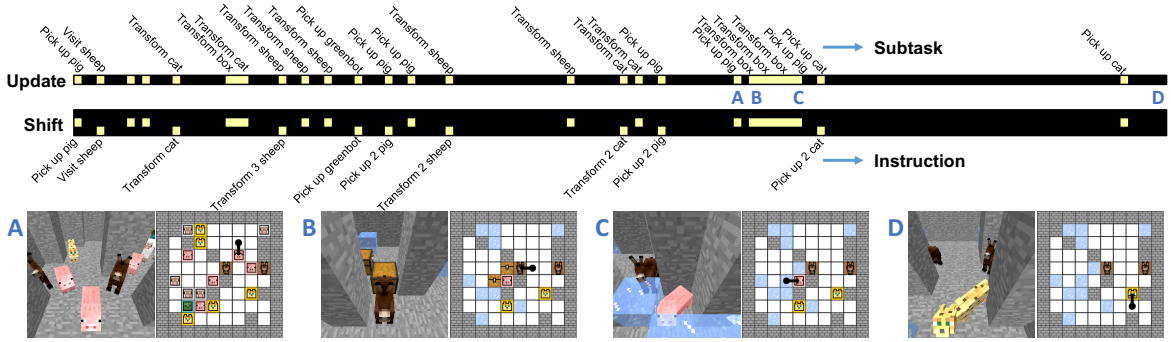

Figure 7: Analysis of the learned policy. 'Update' shows our agent's internal update decision. 'Shift' shows our agent's instruction-shift decision (-1, 0, and +1 from top to bottom). The bottom text shows the instruction indicated by the instruction pointer, while the top text shows the subtask chosen by the meta controller. (A) the agent picks up the pig to finish the instruction and moves to the next instruction. (B) When the agent observes a box that randomly appeared while executing 'Pick up 2 pig' instruction, it immediately changes its subtask to [Transform, Box]. (C) After dealing with the event (transforming a box), the agent resumes executing the instruction ('Pick up 2 pig'). (D) The agent finishes the final instruction.

a subtask is finished, this baseline puts a high probability to switch to [Transform, Box] regardless of the existence of box because transforming a box gives a bonus reward if a box exists by chance. However, this leads to wasting too much time finding a box until it appears and results in a poor success rate due to the time limit. This result implies that an open-loop policy that has to wait until a subtask finishes can be confused by such an uncertain event because it cannot interrupt ongoing subtasks before termination.

On the other hand, we observed that 'Hierarchical-Short' often fails on loop instructions by moving on to the next instruction before it finishes such instructions. This baseline should repeat the same subtask while not changing the instruction pointer for a long time and the reward is even more delayed given loop instructions. In contrast, the subtask updater in our architecture makes fewer decisions by operating at a large time-scale so that it can get more direct feedback from the long-term future. We conjecture that this is why our architecture performs better than this baseline. This result shows that learning when to update the subtask using the neural network is beneficial for dealing with delayed reward without compromising the ability to interrupt.

Analysis of The Learned Policy. We visualized our agent's behavior given a long list of instructions in Figure 7. Interestingly, when the agent sees a box, the meta controller immediately changes its subtask to [Transform, Box] to get a positive reward even though its instruction pointer is indicating 'Pick up 2 pig' and resumes executing the instruction after dealing with the box. Throughout this event and the loop instruction, the meta controller keeps the instruction pointer unchanged as illustrated in (B-C) in Figure 7. In addition, the agent learned to update the instruction pointer and the subtask almost only when it is needed, which provides the subtask updater with temporally-extended actions. This is not only computationally efficient but also useful for learning a better policy.

5. Conclusion

In this paper, we explored a type of zero-shot task generalization in RL with a new problem where the agent is required to execute and generalize over sequences of instructions. We proposed an analogy-making objective which enables generalization over unseen parameterized tasks in various scenarios. We also proposed a novel way to learn the time-scale of the meta controller that proved to be more efficient and flexible than alternative approaches for interrupting subtasks and for dealing with delayed sequential decision problems. Our empirical results on a stochastic 3D domain showed that our architecture generalizes well to longer sequences of instructions as well as unseen instructions. Although our hierarchical RL architecture was demonstrated in the simple setting where the set of instructions should be executed sequentially, we believe that our key ideas are not limited to this setting but can be extended to richer forms of instructions.

Acknowledgement

This work was supported by NSF grant IIS-1526059. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the sponsor.

References

D. Andre and S. J. Russell. Programmable reinforcement learning agents. In *NIPS*, 2000.

D. Andre and S. J. Russell. State abstraction for programmable reinforcement learning agents. In *AAAI/IAAI*, 2002.

J. Andreas, D. Klein, and S. Levine. Modular multitask reinforcement learning with policy sketches. *CoRR*, abs/1611.01796, 2016.

P.-L. Bacon, J. Harb, and D. Precup. The option-critic architecture. In *AAAI*, 2017.

S. R. K. Branavan, H. Chen, L. S. Zettlemoyer, and R. Barzilay. Reinforcement learning for mapping instructions to actions. In *ACL/IJCNLP*, 2009.

D. L. Chen and R. J. Mooney. Learning to interpret natural language navigation instructions from observations. In *AAAI*, 2011.

J. Chung, S. Ahn, and Y. Bengio. Hierarchical multiscale recurrent neural networks. In *ICLR*, 2017.

B. C. da Silva, G. Konidaris, and A. G. Barto. Learning parameterized skills. In *ICML*, 2012.

C. Devin, A. Gupta, T. Darrell, P. Abbeel, and S. Levine. Learning modular neural network policies for multi-task and multi-robot transfer. In *ICRA*, 2017.

T. G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

C. Florensa, Y. Duan, and P. Abbeel. Stochastic neural networks for hierarchical reinforcement learning. In *ICLR*, 2017.

M. Ghavamzadeh and S. Mahadevan. Hierarchical policy gradient algorithms. In *ICML*, 2003.

A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality reduction by learning an invariant mapping. In *CVPR*, 2006.

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

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

D. Isele, M. Rostami, and E. Eaton. Using task features for zero-shot knowledge transfer in lifelong learning. In *IJCAI*, 2016.

G. Konidaris and A. G. Barto. Building portable options: Skill transfer in reinforcement learning. In *IJCAI*, 2007.

G. Konidaris, I. Scheidwasser, and A. G. Barto. Transfer in reinforcement learning via shared features. *Journal of Machine Learning Research*, 13:1333–1371, 2012.

J. Koutnik, K. Greff, F. Gomez, and J. Schmidhuber. A clockwork rnn. In *ICML*, 2014.

T. D. Kulkarni, K. R. Narasimhan, A. Saeedi, and J. B. Tenenbaum. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *arXiv preprint arXiv:1604.06057*, 2016.

M. MacMahon, B. Stankiewicz, and B. Kuipers. Walk the talk: Connecting language, knowledge, and action in route instructions. In *AAAI*, 2006.

A. McGovern and A. G. Barto. *Autonomous discovery of temporal abstractions from interaction with an environment*. PhD thesis, University of Massachusetts, 2002.

H. Mei, M. Bansal, and M. R. Walter. Listen, attend, and walk: Neural mapping of navigational instructions to action sequences. *arXiv preprint arXiv:1506.04089*, 2015.

T. Mikolov, Q. V. Le, and I. Sutskever. Exploiting similarities among languages for machine translation. *arXiv preprint arXiv:1309.4168*, 2013.

J. Oh, V. Chockalingam, S. Singh, and H. Lee. Memory-based control of active perception and action in minecraft. In *ICML*, 2016.

E. Parisotto, J. L. Ba, and R. Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. In *ICLR*, 2016.

R. Parr and S. J. Russell. Reinforcement learning with hierarchies of machines. In *NIPS*, 1997.

S. E. Reed, Y. Zhang, Y. Zhang, and H. Lee. Deep visual analogy-making. In *NIPS*, 2015.

A. A. Rusu, S. G. Colmenarejo, C. Gulcehre, G. Desjardins, J. Kirkpatrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, and R. Hadsell. Policy distillation. In *ICLR*, 2016.

T. Schaul, D. Horgan, K. Gregor, and D. Silver. Universal value function approximators. In *ICML*, 2015.

J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. In *ICLR*, 2016.

S. P. Singh. The efficient learning of multiple task sequences. In *NIPS*, 1991.

S. P. Singh. Transfer of learning by composing solutions of elemental sequential tasks. *Machine Learning*, 8(3-4): 323–339, 1992.

S. Sukhbaatar, A. Szlam, G. Synnaeve, S. Chintala, and R. Fergus. Mazebase: A sandbox for learning from games. *arXiv preprint arXiv:1511.07401*, 2015.

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

S. Tellex, T. Kollar, S. Dickerson, M. R. Walter, A. G. Banerjee, S. J. Teller, and N. Roy. Understanding natural language commands for robotic navigation and mobile manipulation. In *AAAI*, 2011.

S. Tellex, R. A. Knepper, A. Li, D. Rus, and N. Roy. Asking for help using inverse semantics. In *RSS*, 2014.

C. Tessler, S. Givony, T. Zahavy, D. J. Mankowitz, and S. Mannor. A deep hierarchical approach to lifelong learning in minecraft. In *AAAI*, 2017.

W. Zaremba and I. Sutskever. Reinforcement learning neural turing machines. *arXiv preprint arXiv:1505.00521*, 2015.