One-Shot Learning from a Demonstration with Hierarchical Latent Language

Anonymous ACL submission

Abstract

Humans have the capability, aided by the expressive compositionality of their language, to learn quickly by demonstration. They are able to describe unseen task-performing procedures and generalize their execution to other contexts. In this work, we introduce DescribeWorld, an environment designed to test this sort of generalization skill in grounded agents, where tasks are linguistically and procedurally composed of elementary concepts. The agent observes a single task demonstration in a Minecraft-like grid world, and is then asked to carry out the same task in a new map. To enable such a level of generalization, we propose a neural agent infused with hierarchical latent language—both at the level of task inference and subtask planning. Our agent first generates a textual description of the demonstrated unseen task, then leverages this description to replicate it. Through multiple evaluation scenarios and a suite of generalization tests, we find that agents that perform text-based inference are better equipped for the challenge under a random split of tasks.

1 Introduction

Humans are highly capable of learning by example. If a child watches their school teacher draw a purple winged elephant then recite the alphabet backwards, they can replicate the sequence of activities at home with relative ease. This is in no small part due to the human ability to leverage the compositionality of language in order to comprehend new situations composed of familiar concepts (Chomsky, 1957). The child can restate the demonstration in words (as we did above), naturally decomposing it into its distinct subcomponents (the drawing, and the alphabet), which are themselves procedurally compositional (e.g., *"pick up purple marker, . . . "*). Humans use their linguistic understanding of a task's hierarchical compositionality to generalize it to a new context; without this generalization, we might expect a child would overfit to the specifics of the classroom context.

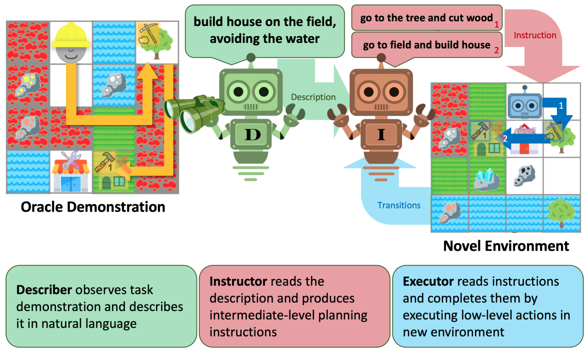


Figure 1: Framework for learning from demonstration via latent language. The **Describer** module observes an oracle demonstration of an unseen task and describes it in text. Given the generated description, the **Instructor** module infers necessary subtasks, accomplished by the **Executor** module via low-level control actions.

In this work, we explore whether grounded artificial agents can similarly generalize from a demonstration: a single expert trajectory accomplishing a task. More specifically, we tackle the setting where an agent observes a demonstration of an unknown task, potentially never seen before, and is then asked to perform the same task in a new context.

We construct DescribeWorld, an environment containing a dataset of high-level tasks involving building recipes, navigation, and interaction with objects and terrains.[1] Test tasks are distinct from training tasks, but they are procedurally composed of the same subtasks and low-level actions.

As humans leverage language to perform such generalization, we follow recent work (Ruis et al., 2020) by designing, alongside a traditional random task split, a suite of benchmark splits that require learning systematic rules governing how linguistic variation affects a task's subtask 'recipe.' For example, the agent might be trained to build a `pig barn` and an `iron shrine`, then during testing

[1] Examples available at `describeworld.github.io`

must build the unseen composition `pig shrine`.

To perform in this task environment, we devise a novel HLLP (standing for Hierarchical Latent Language Policy) agent that represents both high-level tasks (*"build a house on field"*) and subtask plans (*"cut wood"*) in natural language. As depicted in Figure 1, this effectively recasts the challenge of learning from demonstrations as a) describing the demonstrated unseen task, then b) following the predicted description in a new map. The agent uses text representations at two levels of abstraction: identifying top-level verbalized tasks (via a *describer* module), and identifying a sequence of intermediate-level subtasks (via *instructor*). We train the agent via imitation learning on synthetic text associated with oracle actions.

Our novel testing scenario for DescribeWorld is **demonstration following**, where the agent must replicate a demonstrated task in another randomly-generated map. Given its challenging nature, we evaluate a simpler scenario, **description following** (Weller et al., 2020), which assumes that the agent instead has access to a gold text description of the task. This ablated variant allows us to examine performance at lower levels of abstraction by asking: were an agent to successfully derive a text description of an unseen task, could it then follow the task in a new context?

We contrast approaches that leverage latent language policies versus those that instead use continuous representations. We find that modeling agent policy as latent natural language improves the ability to generalize to demonstrations of unseen tasks.

1.1 Contribution

We frame the contribution of our demonstration following environment, DescribeWorld, and our proposed hierarchical latent language policy agent in terms of Lake and Murphy (2021)'s five desiderata for a computational theory of semantics characteristic of human language use:

1. Describing, or understanding the description of, a perceptually present scenario: the HLLP agent receives as input a multi-modal[2] demonstration of a task, and expresses it in text so as to generalize into a new randomly-generated map.

2. Choosing words on the basis of internal desires, goals, or plans: the agent uses natural language to both describe a demonstrated high-level task, as well as to verbalize intermediate-level subtasks to complete at the level of control policy.

3. Responding to instructions and requests appropriately: the agent iteratively executes action sequences against the task environment in order to follow the high-level descriptions and low-level instructions it produces for itself.

4. Producing and understanding unseen conceptual combinations: test demonstrations show unseen high-level tasks composed linguistically and procedurally of known concepts.

5. Changing one's beliefs about the world based on linguistic input: demonstrations convey environmental constraints – e.g. that walking on lava yields a penalty— that the agent must verbalize and act upon via low-level control policy.

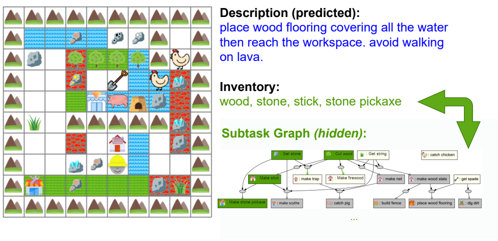
2 Related Work

Latent Language Policy Agents Natural language has been proposed as a medium for conveying task-specific goals (Karch et al., 2020) and constraints (Yang et al., 2021) to grounded reinforcement learning agents. Andreas et al. (2018) show the benefit of reparamatrizing a continuous policy search into discrete text space for various few-shot 'learn-the-rule' tasks. They suggest that such "latent language policy" (LLP) models are a promising avenue for generalization on the basis of language learning. More recent work has applied LLPs to real-time strategy games (Hu et al., 2019; Jacob et al., 2021), while Chen et al. (2021) show that LLPs trained to generate and follow crowdsourced instructions can perform few- or zero-shot simple crafting tasks in a small grid world. Our work considers a similar style of environment, though our high-level tasks are more complex, extending beyond individual crafting recipes.[3]

Grounded Language Environments Several environments were developed to study language grounding where an embodied agent is given high-level task descriptions and/or instructions to follow (e.g., LANI (Misra et al., 2018), Room2Room (Anderson et al., 2018), ALFRED (Shridhar et al., 2020), BabyAI (Chevalier-Boisvert et al., 2018), Ask Your Humans (Chen et al., 2021)). Chevalier-Boisvert et al. (2018) and Hill et al. (2019) investigate compositional rule learning for navigational and pick-up/put-down skills using a synthetic language of instructions in 2D and 3D environments, respectively. Ruis et al. (2020) construct

[2]symbolic map images, plus text-based item inventories

[3]Performance by Chen et al. (2021)'s model degrades for crafting recipes with 5 'steps', while ours have upwards of 16.



Unique Tasks (End Goals + Terr. Consts.)			10604
Unique End Goals			2651
Objects	29	Pickable Objects	11
Craftable Items	19	Buildable Structures	13
Placeable Terrains	7	Natural Terrains	3

Figure 2: DescribeWorld overview. Maps are symbolic images, while the task description, predicted by the agent from a demonstration, and the inventory, reflecting subtask completion, are encoded in text.

a grounded instruction following benchmark that evaluates many types of systematic generalization. Our effort builds upon theirs, introducing a novel scenario (demonstration following) as well as tasks with longer trajectories, subtask dependencies, and new action types (building/placing).

Language-Based Generalization in Humans and Models Lake and Baroni (2018) show that RNN-based sequence models struggle to perform systematic compositional generalization on the basis of abstract linguistic rules, while humans are extremely effective at it given few examples (Lake et al., 2019). Kim and Linzen (2020) similarly find a lack of compositional generalization in neural models trained for semantic parsing.

Recent work in NLP has centered around training large language models to perform few- and zero-shot problem solving given text-based instructions and task descriptions (Weller et al., 2020; Mishra et al., 2021; Wei et al., 2021). However, evidence suggests that there exists a gap between current instruction following capabilities and true understanding of underlying task instruction semantics (Webson and Pavlick, 2021).

Meta-Learning One way to achieve generalization is to learn strategies that can quickly adapt to novel tasks by leveraging past experiences (Schmidhuber, 1987; Thrun and Pratt, 1998; Hochreiter et al., 2001; Bengio et al., 2007). Specifically, our experimental setup falls under the zero- and few-shot imitation learning category (Duan et al., 2017; Finn et al., 2017a,b; Wang et al., 2017; James et al., 2018; Yu et al., 2018; Pan et al., 2020; Zhou et al., 2020), where our approach receives a **single**

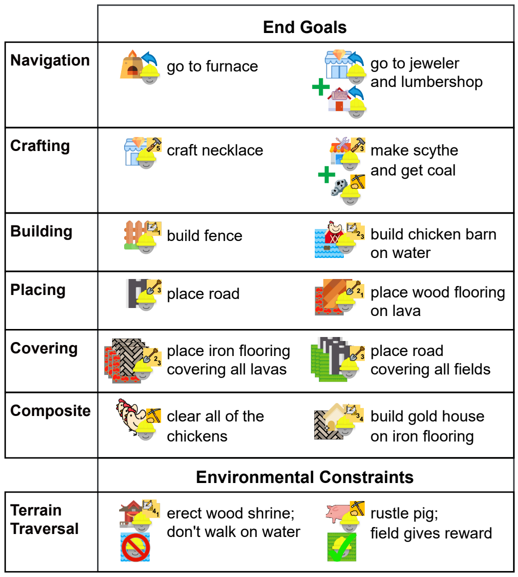


Figure 3: Categories of end goals and environmental constraints parametrizing high-level tasks.

demonstration to solve novel tasks in new contexts.

3 DescribeWorld Environment

DescribeWorld is a 2D grid world implemented atop the **Mining** domain from Sohn et al. (2018). The procedurally generated map (Figure 2(a)) is an 8x8 grid (with surrounding walls); cells may be populated with walls, terrains, and objects. The agent can perform movement, `use`, and `place` actions in order to complete subtasks that either add resources to its inventory, build items, or place craftable terrains in the agent's current location. Details can be found in Appendix A and on our project webpage. The set of possible subtasks and their dependencies (depicted in Appendix Figure 7) is constant across all tasks; we combine subtasks in unseen ways to form unique high-level tasks to be learned from demonstration.

3.1 Compositional Tasks and Subtasks

Tasks and subtasks in DescribeWorld exhibit procedural and lexical compositionality. A list of high-level task categories is shown in Figure 3. Tasks may also be parameterized by *environmental constraints*–namely, that traversing a particular type of terrain will produce either a reward or a penalty.

Certain building and placing subtasks optionally accept a special ingredient material, e.g. `gold house`. The recipes for these subtasks comprise those needed to acquire the material plus those needed to build the object. All `gold` items require smelted gold, while all `houses`, whether they are silver, gold, or regular, require wood slats, and iron. These subtasks require a pair of build-key actions to complete: the first uniquely determines the type of object to build, while the second determines which special material should be used. The action to specify a given special ingredient is constant across all special recipes. Further details of such subtasks are shown in Appendix Table 5.

3.2 World Model

The state at time step t is represented as a tuple (M_t, I_t), where map M_t is a symbolic $8 \times 8 \times 3$ tensor with channels for agent, item, and terrain. Inventory I_t is a text representation (comma separated) of the currently-held items, e.g. wood, stone, spade. There is a step penalty of -1, and we track the number of traversals over reward- and penalty-giving terrains; rewarding cells can only be triggered once per game. Trajectories end upon end goal completion, or hitting a 300-step time limit.

3.3 Oracle

We implement an oracle that navigates the gridworld and completes high-level tasks. The oracle computes the set of all necessary subtasks required to complete the high-level task. It then computes the intersection of necessary and currently eligible (i.e. prerequisite-satisfied) subtasks, then chooses one to complete according to a canonical order.[4] This process is repeated until the high-level task is completed. Example trajectories are provided in Appendix Figure 8. The oracle is used both to generate trajectories for demonstration following (rolling out a trajectory from start to finish), as well as to provide gold instructions and executions during imitation learning (i.e. used on-the-fly to generate the next step towards completing the next subtask). In the former case, in order to convey environment-specific constraints such as rewards/penalties for stepping on particular terrain types, we ensure that it traverses all terrain types at least once. Ensuring traversal of all terrains can require a navigational detour of a couple steps.

3.4 Data Splits

To test various forms of compositional generalization in demonstration following agents, we introduce a suite of train/test splits, depicted in Figure 4, each of which requires a particular form of rule-based systematic generalization.

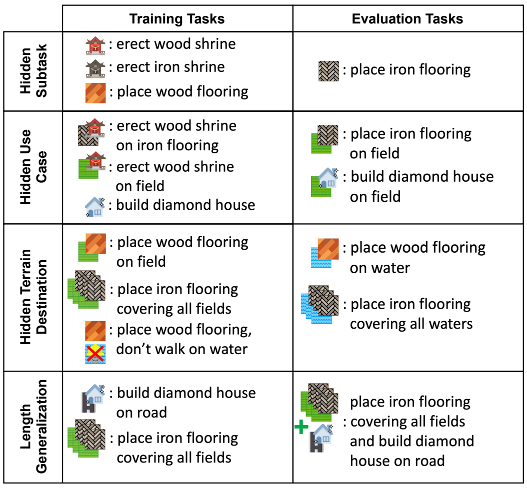


Figure 4: Data splits testing for systematic generalization in demonstration following agents

Random Split We compare against a simple random 70/30 split, where tasks are sorted by hashing the text of their end goal, ignoring terrain rewards/penalties. The random split test is nontrivially challenging due to complex subtask dependencies and unseen randomly-generated maps.

Hidden Subtask This split requires procedural generalization on the basis of ingredient/object composition. We remove from the training data all end goals involving the subtask place iron flooring, but leave in all other tasks that involve other types of flooring, and those that use the iron special ingredient. We repeat the procedure with erect pig shrine and build diamond house. Appendix Table 5 depicts the building recipes for these subtasks, as well as those left in the training set with which they linguistically and procedurally overlap; those serve as the source of generalization. The held-out test set contains all tasks that involve any of the three unseen subtasks.[5] This challenge is twofold: the agent must learn that modifiers like pig and diamond correspond to a required set of subtasks, plus a fixed specification action when building a structure.

Hidden Use Case This split requires generalization of a subtask learnt in one isolated use case. We remove from training all tasks involving diamond house, but leave in the plain task build diamond house . At test time, the agent must use the sub-

[4] We initially had the oracle complete whichever eligible subtask required the fewest steps. However, this led to training instability due to the compounded difficulty of inferring required subtasks and selecting an eligibility-adherent completion order based on distances in a random map. Instead, we choose the first eligible subtask in a canonically-ordered list.

[5] We leave out tasks requiring covering terrain from the hidden subtask and use case test sets due to agents' low completion rate on the category under the random split.

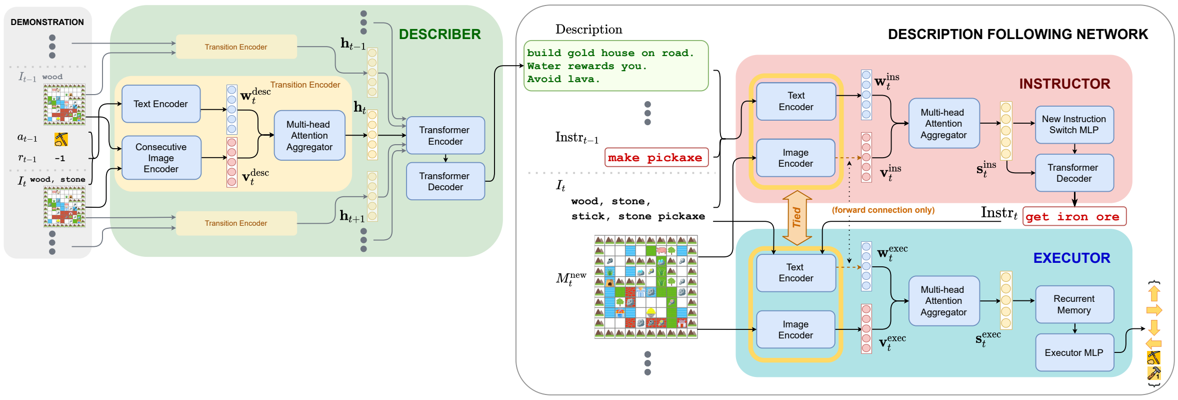


Figure 5: Architecture of hierarchical latent language policy agent. The describer module decodes a description of a demonstration in map M^{dem}, then the instructor/executor modules replicate the task in new map M^{new}.

task in all other end goal categories, e.g. `build diamond house on field`. We repeat the process for `place road` and `make goldware`. We also test the generalization of `iron flooring` appearing during training only as a destination, e.g. in `build house on iron flooring`. At test time the agent must use the concept in all other cases, e.g. `place iron flooring on field`.

Hidden Terrain Destination This split requires generalization of terrains as not only sources of traversal penalty/reward, but also as a building destination. We hold out all tasks that involve the terrain `water` as a destination, e.g. in `build house on water`. We leave in tasks that use other terrain types, e.g. `lava` and `field`, as destinations. We also leave in tasks that involve `water` as a terrain constraint, as in `build house. don't walk on water`. This split therefore requires agents to generalize the fact that `water` can also serve as a destination from the dual roles of other terrains.

Length Generalization Neural sequence models have shown to fail to generalize to task instance lengths longer than those seen in training (Graves et al., 2014; Ruis et al., 2020). We test for this capacity by holding out tasks with the top 10% longest oracle trajectories.

4 Hierarchical Latent Language Policy Agent

We design a three-layer hierarchical latent language policy (HLLP) agent to perform one-shot demonstration following. The **describer** module observes oracle demonstrations and describes them in text. The description following **instructor** and **executor** modules work in tandem to generate intermediate-level text instructions and choose low-level actions. We thus parametrize our agent's policy via text description D and instruction sequence $\text{Instr}_1 \ldots \text{Instr}_i$.

$$D = f_{\text{descr}}(M^{\text{dem}}_{1:n}, I^{\text{dem}}_{1:n}, a^{\text{dem}}_{1:n}, r^{\text{dem}}_{1:n})$$

$$\text{Instr}_i = f_{\text{ins}}(M_i, I_i, \text{Instr}_{i-1}; D)$$

$$a_i = f_{\text{exec}}(M_{1:i}, I_{1:i}, a_{1:i-1}; \text{Instr}_{1:i})$$

Describer module Depicted in (Figure 5, left), this is a basic transformer-based "video summarization" model. It takes a demonstration (i.e., sequence of transitions) as input. A transition at time step t is a 5-tuple including the previous step's symbolic image M_{t-1}, the action taken a_{t-1}, the resulting reward r_{t-1}, the resulting symbolic image M_t, and the text enumerating the new inventory I_t.

For each time step t, we use an image encoder to encode M_{t-1} and M_t, and a text encoder to encode the concatenation of a_{t-1}, r_{t-1}, and I_t. The resulting encodings are aggregated using an attention mechanism into a single transition representation. To obtain a single *demonstration* representation, we use a second transformer encoder over the sequence of transition encodings , then use a standard attention-equipped transformer decoder to generate a description of the demonstrated task.

Instructor module Our framework for generating and following instructions given a task description is similar to that of Hu et al. (2019), except we use a language model decoder instead of a classifier and compute separate state encodings for the two modules. At each time step, the *instructor* module (Figure 5, upper right) computes a multimodal state representation via attention-based aggregation of separate encodings of the textual and image components of the state observation. The text representation is a transformer encoding of the task description concatenated with the inventory text, while the image representation is a convolutional neural network encoding of the map. The state representation is passed to the 'new instruction' classifier, which determines whether to decode a new instruction or copy that of the previous timestep.[6]

Executor module Shown in (Figure 5, lower right), this module computes a combined state representation using the same encoder parameters, but using the generated instruction text instead of the high-level task description. The state representation is used to update a recurrent memory cell, the hidden state of which is fed to an MLP classifier over low-level actions.

4.1 Training

Models are trained to convergence on a validation set containing tasks with the same end goals as those in the training data, but with unseen combinations of terrain rewards/penalties. The describer is trained with typical seq2seq cross-entropy-based supervised learning. The instructor/executor pair is trained with imitation learning using DAgger (Ross et al., 2011). To train the instructor, we generate a synthetic instruction for each subtask. Because the description, which is **not** shown to the executor, conveys terrain rewards/penalties, we train the instructor to decode them as well, e.g. in '`go to lava and place road. avoid walking on water.`'

Notably, while the instructor and executor share text and image encoder parameters, the text encoder is only updated using the instructor loss, and the image encoder is updated using the executor loss. Further details are provided in Appendix C.

5 Experiments

Demonstration Following We test agents 15 times for each evaluation task, using demonstrations in 5 randomly-generated maps each paired with 3 unique maps in which to replicate the task.

Description Following We use the same task instances as the previous scenario, but provide the ground truth task description directly to the agent.

Instruction Following To set an upper bound for instructor performance, we evaluate the performance of the executor given oracle instructions.

5.1 Baselines

Nonverbal Baseline To test the effect of computing a latent text representation of the high-level task, we compare against a nonverbal baseline (**NV Baseline**) that at each time step computes a continuous representation of the demonstration trajectory instead of encoding a predicted text description. The architecture resembles that of the executor module, with a transformer encoder over demonstration transitions (as in the describer) rather than text description. Further details are provided in Appendix B.4.

$$a_i = f_{\text{exec}}(M_{1:n}^{\text{dem}}, I_{1:n}^{\text{dem}}, a_{1:n}^{\text{dem}}, r_{1:n}^{\text{dem}}, M_{1:i}, I_{1:i}, a_{1:i-1})$$

Latent Language Description Only We also compare against a second baseline that conditions the agent's policy on a latent language description (**LLD**), but does not leverage language at the level of intermediate subtask planning. The LLD architecture resembles the HLLP without the instructor module.

$$D = f_{\text{descr}}(M_{1:n}^{\text{dem}}, I_{1:n}^{\text{dem}}, a_{1:n}^{\text{dem}}, r_{1:n}^{\text{dem}})$$
$$a_i = f_{\text{exec}}(M_{1:i}, I_{1:i}, a_{1:i-1}; D)$$

6 Results

We average the performance of agents trained using 5 random seeds. Table 1 shows exact match rates for the describer and instructor, measured for the latter at the point of each agent-generated new instruction. Table 2 shows completion rate on the random task split broken down by category, while Table 4 shows it on the generalization splits.

6.1 Random Split

No model converges to perfect performance on the random task split. Both agents that leverage a predicted task description (HLLP and LLD) outperform the nonverbal baseline. As shown in Table 1, the describer module exhibits around 70% exact match accuracy on a set of unseen tasks and 85% on a set of training tasks paired with unseen terrain rewards.[7] The describer properly identifies over 75% of unseen tasks, which are conveyed by the first sentence of each description. It struggles with navigation and clearing item subtasks, which have uniquely short trajectories. Description following agents achieve high task completion rates given the ground truth task description (Table 2, middle). The HLLP agent outperforms the LLD baseline by greater than 5%; however, the latter is more effective at terrain covering and item clearing subtasks,

[6] This is necessary because of a lack of an explicit state cue signifying the need for a new instruction, e.g. a change in inventory in Chen et al. (2021).

[7] e.g. `train on {'build house. lava rewards you,' 'place road. avoid water.'}`, then evaluate on `{'build house. avoid water.'}`

EM (%)	# Eval Tasks	Describer Valid Full	Describer Valid Goal	Describer Eval Full	Describer Eval Goal	Instructor Eval All	Instructor Eval Last
Random Split	**15140**	**84.3**	**92.4**	**69.3**	**75.7**	**77.4 ± 5.1**	**79.8 ± 4.3**
Navigation	700	10.1	10.6	0.9	0.9	60.1 ± 16.6	85.1 ± 1.8
Crafting	5400	98.0	98.9	87.4	88.0	88.9 ± 4.4	83.2 ± 4.7
Craft then Nav	880	88.1	99.4	84.0	88.1	89.7 ± 9.6	97.0 ± 1.3
Build on Terrain	6040	83.0	92.9	63.8	71.7	78.0 ± 8.1	81.7 ± 5.6
Cover Terrain	1680	71.5	98.5	59.5	84.3	60.7 ± 5.1	52.7 ± 3.4
Clear Items	400	95.2	95.2	37.0	37.5	72.2 ± 10.0	72.9 ± 11.0
Hid. Subtask	**8900**	**84.8**	**91.4**	**14.5**	**15.8**	**43.6 ± 4.0**	**16.5 ± 4.8**
Hid. Use Case	**12860**	**84.1**	**90.3**	**19.7**	**22.2**	**40.5 ± 5.0**	**17.7 ± 6.8**
Hid. Terr Destn	**6520**	**84.9**	**91.8**	**0.0**	**0.0**	**26.5 ± 2.1**	**5.1 ± 1.4**
Length Gen.	**5445**	**85.2**	**92.0**	**69.7**	**92.9**	**62.9 ± 5.5**	**63.8 ± 8.1**

Table 1: Describer and Instructor exact match (EM) against gold references. Describer EM shown for **Full** text, and first sentence describing end **Goal**. Validation tasks have same end goals as train, but novel terrain reward/penalty combinations. Instructor EM shown for **All** and **Last** instructions given.

Completion (%)	NV Baseline	LLD	HLLP
Demonstration Following			
Overall	**25.2 ± 7.0**	**65.1 ± 3.2**	**68.4 ± 2.2**
Navigation	45.6 ± 2.6	40.5 ± 1.3	46.5 ± 2.9
Crafting	44.4 ± 13.7	79.6 ± 3.2	85.5 ± 1.7
Craft then Nav	45.4 ± 14.3	89.4 ± 1.8	95.1 ± 1.4
Build on Terrain	9.1 ± 2.7	54.4 ± 4.1	63.0 ± 3.4
Cover Terrain	5.4 ± 2.9	61.2 ± 4.0	37.9 ± 1.7
Clear Items	11.6 ± 5.6	39.3 ± 0.6	27.0 ± 6.3
Ground Truth Description Following			
Overall	–	**76.7 ± 3.6**	**82.1 ± 2.5**
Navigation	–	93.9 ± 2.3	96.2 ± 2.9
Crafting	–	86.0 ± 3.3	92.0 ± 1.8
Craft then Nav	–	90.1 ± 1.5	95.9 ± 1.6
Build on Terrain	–	67.2 ± 4.7	81.3 ± 4.2
Cover Terrain	–	64.8 ± 4.2	43.8 ± 2.5
Clear Items	–	85.8 ± 3.8	67.4 ± 9.1
Ground Truth Instruction Following			
Overall	–	–	**97.2 ± 1.1**
Navigation	–	–	95.7 ± 1.5
Crafting	–	–	98.1 ± 0.9
Craft then Nav	–	–	98.5 ± 0.9
Build on Terrain	–	–	96.6 ± 1.4
Cover Terrain	–	–	97.3 ± 1.1
Clear Items	–	–	95.2 ± 1.8

Table 2: Completion rates on random task split

|Traversals|	# Tasks	Oracle +	Oracle −	NVB +	NVB −	LLD +	LLD −	HLLP +	HLLP −
0 Rew 1 Pen	5880	–	7	–	30	–	12	–	19
0 Rew 2 Pen	5595	–	17	–	63	–	29	–	39
1 Rew 0 Pen	5490	9	–	8	–	8	–	7	–
1 Rew 1 Pen	11670	9	7	7	32	8	12	7	20
2 Rew 0 Pen	5430	17	–	15	–	15	–	14	–

Table 3: Average traversals on reward (+) or penalty (−)-giving terrains by agents on random split. Tasks are categorized by the number of such terrain types.

which require variable numbers of repeated subtasks depending on the random map. The executor performs nearly perfect given oracle instructions (Table 2, bottom), indicating most description following errors are made by the instructor.

Adherence to Terrain Constraints Table 3 depicts the rate at which demonstration following agents traverse terrains giving penalties or rewards.[8] We compare against the traversal frequency of an oracle. This comparison is made difficult by the variability among the times taken by agents to either complete a task or hit the 300-step limit. However, the results suggest that the HLLP agent is substantially worse at avoiding penalty terrains than the LLD. All agents are close to oracle performance at traversing reward terrains.

6.2 Generalization Splits

Hidden Subtask Models generally fail to generalize to unseen compositional subtasks. The describer identifies only 16% of the unseen end goals, while the instructor predicts the correct final instruction (usually corresponding to the hidden subtask) at the same rate. Figure 6 (upper) shows that given gold descriptions, the HLLP agent accomplishes only `pig shrine` tasks at all, while the LLD also accomplishes `diamond` house at a low rate. The executor often fails to handle unseen oracle instructions[9]. We observe qualitatively that the description following HLLP tends to acquire the correct recipe items, but often does not generate the correct final instruction and/or perform the right pair of low-level build operations to place the structure. The instructor correctly generates the novel `pig shrine` concept around 30% of the time.

Hidden Use Case The nonverbal demonstration follower completely fails to generalize tasks to new use cases. The describer module successfully identifies 20% of unseen use case tasks, but no latent language agent completes more than 5% from predicted descriptions. We observe that completion of the isolated *training* tasks is not perfect (Figure 6 middle), indicating that poor performance on this split may be due to a lack of convergence on the subtasks of interest, which underpopulate the training data. The executor module performs well on unseen `goldware` and `iron flooring` use cases.

Hidden Terrain Destination Agents fail to generalize a terrain observed only as a reward/penalty source to then being a destination for building tasks; particularly for covering tasks. This is the case at all abstraction levels; the executor given gold instructions completes 55% of `build` tasks but only 3% of `cover` tasks. The describer and instructor

[8] Maps are generated ignoring terrain reward/penalties, so completing tasks may require traversing a penalizing terrain.

[9] e.g. the final `'build diamond house'` instruction.

Completion (%)	NV Baseline	LLD	HLLP
Demonstration Following			
Hidden Subtask	2.5 ± 1.4	1.3 ± 0.4	0.4 ± 0.3
Hidden Use Case	0.3 ± 0.5	5.1 ± 1.5	5.9 ± 3.3
Hidden Terr Destn	1.6 ± 0.9	4.6 ± 0.5	3.7 ± 0.7
Length Gen.	6.0 ± 2.1	62.6 ± 3.8	57.9 ± 9.0
Description Following			
Hidden Subtask	–	7.4 ± 2.3	8.0 ± 3.1
Hidden Use Case	–	8.2 ± 1.9	11.8 ± 6.9
Hidden Terr Destn	–	1.8 ± 1.2	2.8 ± 1.2
Length Gen.	–	65.7 ± 4.1	60.9 ± 9.1
Instruction Following			
Hidden Subtask	–	–	15.6 ± 7.2
Hidden Use Case	–	–	48.6 ± 5.0
Hidden Terr Destn	–	–	35.3 ± 7.2
Length Gen.	–	–	96.6 ± 1.3

Table 4: Completion rates on generalization splits

Figure 6: Hidden subtask and use case tests by subtask.

modules fails to identify the end goal and end instruction at all; however, in 49% of describer failure cases, the predicted end goal differs from the ground truth only by the specified destination (e.g. on `field` instead of the desired on `water`).

Length Generalization Both latent language agents achieve moderate success on length generalization, particularly relative to the nonverbal baseline (6% vs 60%). The describer is extremely successful at identifying long-trajectory tasks, even better than on the random split.

6.3 Discussion

Our results suggest that language serves as an expressive, generalization-promoting representation for artificial one-shot demonstration following agents. Intermediate-level planning on the basis of LM decoding provides incremental improvements upon nonverbal description- and demonstration-following baselines on a random task split, suggesting improved generalization to other maps and unseen tasks sampled from the same distribution as those seen during training. However, we find that instruction-level latent language does not meaningfully improve *systematic compositional* generalization in either demonstration or description following scenarios. Reformulating policy search as sequence search simplifies it in certain useful ways–the improved flexibility and interpretability of text-based reasoning allows for pinpointing errors at multiple levels of decision making, abstracts away low-level execution decisions that do not pertain to certain forms of generalization, as we observe in our hidden use case results. However, a latent language policy alone is not a compositional generalization silver bullet. Indeed, such challenges remain largely unsolved, though recent approaches have suggested incremental progress in specific cases (Andreas, 2020; Qiu et al., 2021; Conklin et al., 2021). We hope that our benchmark adds to this discourse, and that future work considers our evaluation framework. We also welcome future work exploring settings with complex subdependencies *under time limits*. To improve training stability, our instructor chooses subtasks in an in-optimal canonical order that requires text-based reasoning about high-level tasks, but not spatial reasoning about object proximity.

7 Conclusion

Our goal is to design agents that learn new tasks from single examples, with behavior rooted in language. This motivated the construction of DescribeWorld, a task environment for testing one-shot learning of complex tasks from demonstrations. DescribeWorld allowed for the development and evaluation of our *hierarchical latent language policy* agent, which performs decision making on the basis of text at multiple levels of abstraction. We found that models leveraging latent language can improve upon nonverbal alternatives in multiple evaluation scenarios, but that they can struggle with forms of systematic generalization. We observe that models can accomplish systematically novel tasks provided the correct decision is made at a higher level of abstraction, which exemplifies how hierarchical latent language provides a mechanism for isolating the level of policy abstraction in which a generalization might occur.

References

Peter Anderson, Qi Wu, Damien Teney, Jake Bruce, Mark Johnson, Niko Sünderhauf, Ian D. Reid, Stephen Gould, and Anton van den Hengel. 2018. Vision-and-language navigation: Interpreting visually-grounded navigation instructions in real environments. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 3674–3683.

Jacob Andreas. 2020. Good-enough compositional data augmentation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7556–7566, Online. Association for Computational Linguistics.

Jacob Andreas, Dan Klein, and Sergey Levine. 2018. Learning with latent language. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2166–2179, New Orleans, Louisiana. Association for Computational Linguistics.

Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer normalization. *CoRR*, abs/1607.06450.

Samy Bengio, Yoshua Bengio, Jocelyn Cloutier, and Jan Gecsei. 2007. On the optimization of a synaptic learning rule.

Valerie Chen, Abhinav Gupta, and Kenneth Marino. 2021. Ask your humans: Using human instructions to improve generalization in reinforcement learning. In *International Conference on Learning Representations*.

Maxime Chevalier-Boisvert, Dzmitry Bahdanau, Salem Lahlou, Lucas Willems, Chitwan Saharia, Thien Huu Nguyen, and Yoshua Bengio. 2018. Babyai: A platform to study the sample efficiency of grounded language learning. In *International Conference on Learning Representations*.

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*.

Noam Chomsky. 1957. *Syntactic Structures*. De Gruyter Mouton.

Henry Conklin, Bailin Wang, Kenny Smith, and Ivan Titov. 2021. Meta-learning to compositionally generalize. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 3322–3335, Online. Association for Computational Linguistics.

Yan Duan, Marcin Andrychowicz, Bradly C. Stadie, Jonathan Ho, Jonas Schneider, Ilya Sutskever, P. Abbeel, and Wojciech Zaremba. 2017. One-shot imitation learning. *ArXiv*, abs/1703.07326.

Chelsea Finn, P. Abbeel, and Sergey Levine. 2017a. Model-agnostic meta-learning for fast adaptation of deep networks. In *ICML*.

Chelsea Finn, Tianhe Yu, Tianhao Zhang, P. Abbeel, and Sergey Levine. 2017b. One-shot visual imitation learning via meta-learning. In *CoRL*.

Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural turing machines. *arXiv preprint arXiv:1410.5401*.

Matthew Hausknecht and Peter Stone. 2015. Deep recurrent q-learning for partially observable mdps. *arXiv preprint arXiv:1507.06527*.

Felix Hill, Andrew Lampinen, Rosalia Schneider, Stephen Clark, Matthew Botvinick, James L McClelland, and Adam Santoro. 2019. Environmental drivers of systematicity and generalization in a situated agent. In *International Conference on Learning Representations*.

Sepp Hochreiter, Arthur Steven Younger, and Peter R. Conwell. 2001. Learning to learn using gradient descent. In *ICANN*.

Hengyuan Hu, Denis Yarats, Qucheng Gong, Yuandong Tian, and Mike Lewis. 2019. Hierarchical decision making by generating and following natural language instructions. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

Athul Paul Jacob, Mike Lewis, and Jacob Andreas. 2021. Multitasking inhibits semantic drift. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 5351–5366, Online. Association for Computational Linguistics.

Stephen James, Michael Bloesch, and Andrew J. Davison. 2018. Task-embedded control networks for few-shot imitation learning. *ArXiv*, abs/1810.03237.

Tristan Karch, Nicolas Lair, Cédric Colas, Jean-Michel Dussoux, Clément Moulin-Frier, Peter Ford Dominey, and Pierre-Yves Oudeyer. 2020. Language-goal imagination to foster creative exploration in deep rl.

Najoung Kim and Tal Linzen. 2020. COGS: A compositional generalization challenge based on semantic interpretation. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 9087–9105, Online. Association for Computational Linguistics.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *ICLR (Poster)*.

Brenden Lake and Marco Baroni. 2018. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *International conference on machine learning*, pages 2873–2882. PMLR.

Brenden Lake, Tal Linzen, and Marco Baroni. 2019. Human few-shot learning of compositional instructions. In *CogSci*.

Brenden Lake and Gregory Murphy. 2021. Word meaning in minds and machines. *Psychological Review*.

Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. 2021. Cross-task generalization via natural language crowdsourcing instructions.

Dipendra Misra, Andrew Bennett, Valts Blukis, Eyvind Niklasson, Max Shatkhin, and Yoav Artzi. 2018. Mapping instructions to actions in 3d environments with visual goal prediction. In *EMNLP*.

Xinlei Pan, Tingnan Zhang, Brian Ichter, Aleksandra Faust, Jie Tan, and Sehoon Ha. 2020. Zero-shot imitation learning from demonstrations for legged robot visual navigation. *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 679–685.

Ofir Press and Lior Wolf. 2017. Using the output embedding to improve language models. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 157–163, Valencia, Spain. Association for Computational Linguistics.

Linlu Qiu, Hexiang Hu, Bowen Zhang, Peter Shaw, and Fei Sha. 2021. Systematic generalization on gSCAN: What is nearly solved and what is next? In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 2180–2188, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. 2011. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*.

Laura Ruis, Jacob Andreas, Marco Baroni, Diane Bouchacourt, and Brenden M Lake. 2020. A benchmark for systematic generalization in grounded language understanding. *Advances in Neural Information Processing Systems*, 33.

Jurgen Schmidhuber. 1987. Evolutionary principles in self-referential learning. on learning now to learn: The meta-meta-meta...-hook. Diploma thesis, Technische Universitat Munchen, Germany, 14 May.

Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. 2020. ALFRED: A Benchmark for Interpreting Grounded Instructions for Everyday Tasks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Sungryull Sohn, Junhyuk Oh, and Honglak Lee. 2018. Hierarchical reinforcement learning for zero-shot generalization with subtask dependencies. In *Advances in Neural Information Processing Systems*, pages 7156–7166.

Sebastian Thrun and Lorien Y. Pratt. 1998. Learning to learn. *arXiv: Learning*, pages 354–354.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems 30*.

Jane X. Wang, Zeb Kurth-Nelson, Hubert Soyer, Joel Z. Leibo, Dhruva Tirumala, Rémi Munos, Charles Blundell, Dharshan Kumaran, and Matthew M. Botvinick. 2017. Learning to reinforcement learn. *ArXiv*, abs/1611.05763.

Albert Webson and Ellie Pavlick. 2021. Do prompt-based models really understand the meaning of their prompts?

Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le. 2021. Finetuned language models are zero-shot learners.

Orion Weller, Nicholas Lourie, Matt Gardner, and Matthew E. Peters. 2020. Learning from task descriptions. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1361–1375, Online. Association for Computational Linguistics.

Tsung-Yen Yang, Michael Hu, Yinlam Chow, Peter Ramadge, and Karthik R Narasimhan. 2021. Safe reinforcement learning with natural language constraints.

Tianhe Yu, Chelsea Finn, Annie Xie, Sudeep Dasari, Tianhao Zhang, P. Abbeel, and Sergey Levine. 2018. One-shot imitation from observing humans via domain-adaptive meta-learning. *ArXiv*, abs/1802.01557.

Xingdi Yuan, Marc-Alexandre Côté, Alessandro Sordoni, Romain Laroche, Remi Tachet des Combes, Matthew Hausknecht, and Adam Trischler. 2018. Counting to explore and generalize in text-based games. *arXiv preprint arXiv:1806.11525*.

Allan Zhou, Eric Jang, Daniel Kappler, Alexander Herzog, Mohi Khansari, Paul Wohlhart, Yunfei Bai, Mrinal Kalakrishnan, Sergey Levine, and Chelsea Finn. 2020. Watch, try, learn: Meta-learning from demonstrations and reward. *ArXiv*, abs/1906.03352.

Contents in Appendices:

- In Appendix A, we provide further details of the DescribeWorldframework.
- In Appendix B, we describe modeling details of all our proposed agents and baselines.
- In Appendix C, we provide training and implementation details of our agents.
- In Appendix D, we show additional experiment results.

A Environment Details

As depicted in Figure 2, the procedurally generated map (Figure 2(a)) is an 8x8 (10x10 with a wall border) grid whose cells may be populated with walls, terrains and interactable objects. Terrains are either `lava`, `field` or `water`. Some objects disappear upon interaction (`tree`, `stone...`) or transform (`furnace` $\rightarrow$ `lit furnace`), or are permanent fixtures (`lumbershop`, `workspace ...`) at which the agent can perform crafting operations.

The set of possible agent actions comprises directional movement ({`up`, `down`, `left`, `right`}, interact actions ({`pick up`, `use-1 ... use-5`}, and place actions ({`place-1 ... place-4`}) Subtasks generally have a set of prerequisite subtasks (e.g. `make stone pickaxe` requires `get wood` and `get stone`). The requirements for a subtask do not change across tasks, i.e. `make stone pickaxe` always requires the same prerequisites and action/location combination.

Crafting tasks require the agent to perform a specific interact action while in the cell of a specific object (`make stone pickaxe` requires the agent to perform `use-1` while on top of the `workspace`. Building tasks require the agent to perform a use action on a cell without an item already inside it. `place`-based tasks can be performed anywhere regardless of the presence of an item or existing terrain.

If the agent performs actions that render an end goal unattainable (e.g. `build house on field` but the agent covers all fields with other objects), the game immediately ends and produces a large negative reward.

A.1 Task Recipes

Figure 7 depicts the full set of DescribeWorld subtasks and their dependencies.

B Modeling Details

In this section, we provide detailed information of our agents. In Appendix B.1, we will describe some common basic components in the agent architecture. Later on, we will describe each of the proposed agents mentioned in Section 5.

Notations

We use *game step* t to denote one round of interaction between an agent with the environment. We use o_t to denote text observation at game step t. o_t may contain different components depending on a specific context, we will describe individual cases in later subsections. Brackets $[\cdot;\cdot]$ denote vector concatenation. We use $|s|$ to represent the length of (number of tokens in) a sequence s. We use h and w to denote the height and width of an input image, when the image is flattened, the vector size is hw.

B.1 Common Modules

B.1.1 Text Encoder

We use a transformer-based text encoder, which consists of an embedding layer and a transformer block (Vaswani et al., 2017). Specifically, we tokenize an input o_t with the HuggingFace GPT-2 tokenizer[10]. We convert the tokens into 128-dimension embeddings, the embedding matrix is initialized randomly.

The transformer block consists of a stack of 4 convolutional layers, a self-attention layer, and a 2-layer MLP with a ReLU non-linear activation function in between. Within the block, each convolutional layer has 128 filters, with a kernel size of 7. The self-attention layers use a block hidden size of 128, with 4 attention heads. Layer normalization (Ba et al., 2016) is applied after each layer inside the block. Following standard transformer training, we add positional embeddings into each block's input.

At every game step t, the text encoder encodes $o_t \in \mathbb{R}^{|o_t|}$ and results a representation $h_{o_t} \in \mathbb{R}^{|o_t| \times H}$, $H = 128$ is the hidden size.

B.1.2 Image Encoder

We propose two image encoder architectures, each tackling a different type of input:

[10] https://huggingface.co/transformers/model_doc/gpt2.html#gpt2tokenizer

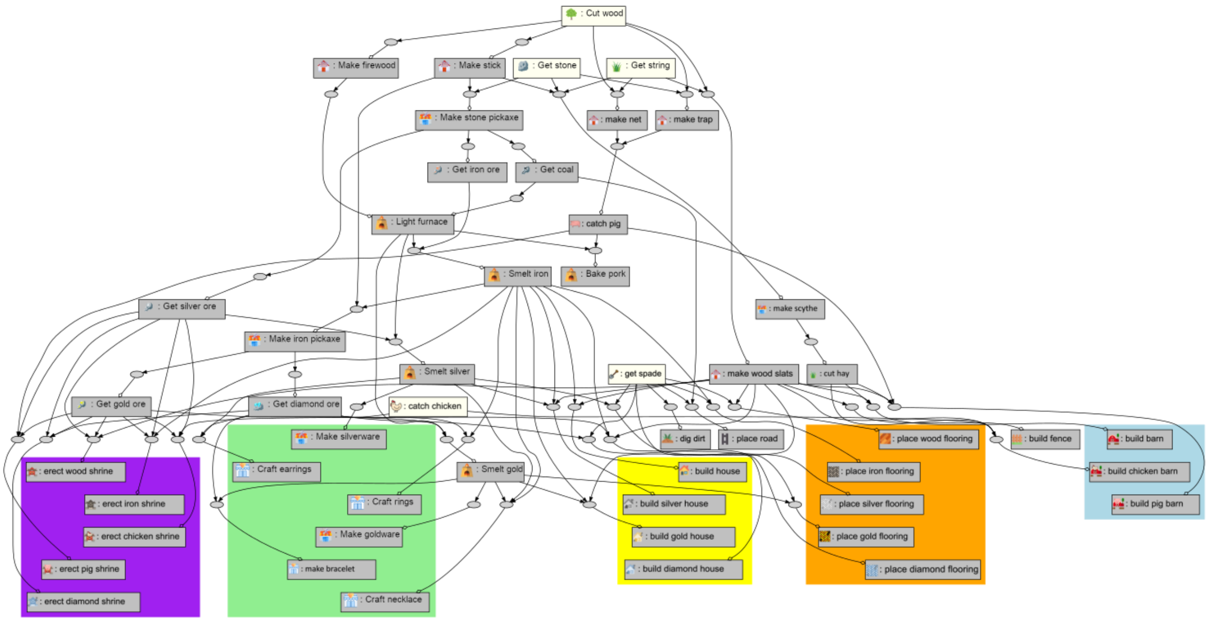


Figure 7: Full subtask dependency graph for the DescribeWorld task environment.

Base Item	Prerequisites	Action 1	Action 2				
			use_1	use_2	use_3	use_4	use_5
flooring	spade	place_2	wood flooring	iron flooring	silver flooring	gold flooring	diamond flooring
barn	hay, wood slats	use_2	barn		chicken barn	pig barn	
house	iron, wood slats	use_3	house		silver house	gold house	diamond house
shrine	gold ore, silver ore	use_4	wood shrine	iron shrine	chicken shrine	pig shrine	diamond shrine

Table 5: List of two-action compositional building/placing recipes

Basic: The basic image encoder is adopted from the BabyAI baseline model (Chevalier-Boisvert et al., 2018). Specifically, given a symbolic image input $M \in \mathbb{Z}_{\geq 0}^{h \times w \times c}$, we use an image bag-of-word (BOW) embedding layer to convert the integer inputs into real-valued embeddings with size $h \times w \times c \times H$, where h, w and c denotes the height, width, and channels of the image, $H = 128$ is the embedding size. We sum up the channel dimension, resulting $E_M \in \mathbb{R}^{h \times w \times H}$.

Next, the image embeddings are fed into a stacked residual convolutional blocks:

$$\begin{aligned} h^{l+1} &= \text{ResidualBlock}^l(h^l), \\ h^0 &= E_M. \end{aligned} \tag{1}$$

Each residual block consists of two convolutional layers, with kernel size of 3 and output channel size of 128. Batch normalization is applied after every convolutional layer, followed by a ReLU non-linear activation function. Before the last ReLU, we apply a residual connection, which adds the block input into the output of the last batch norm layer.

The output size of the stacked residual blocks is $h \times w \times H$, we flatten its spatial dimensions to result the image encoding $h_M \in \mathbb{R}^{hw \times H}$.

Consecutive: In the consecutive image encoder, we aim to capture the difference between two consecutive images. Given two images $M_{t-1} \in \mathbb{Z}_{\geq 0}^{h \times w \times c}$ and $M_t \in \mathbb{Z}_{\geq 0}^{h \times w \times c}$, we first compute their difference $M_{\text{diff}} \in \mathbb{Z}^{h \times w \times c}$. We convert the integer inputs into real-valued vectors using image BOW embedding layers, resulting $E_{t-1} \in \mathbb{R}^{h \times w \times H}$, $E_t \in \mathbb{R}^{h \times w \times H}$ and $E_{\text{diff}} \in \mathbb{R}^{h \times w \times H}$. Note M_{diff} uses a separate image BOW embedding layer.

To aggregate the three image embeddings, we feed their concatenation into an Multilayer Perceptron (MLP):

$$E_M = \text{Tanh}(\text{Linear}([E_{t-1}; E_t; E_{\text{diff}}])), \tag{2}$$

where $E_M \in \mathbb{R}^{h \times w \times H}$. We use the same convolutional architecture to produce image encoding $h_M \in \mathbb{R}^{hw \times H}$ as in the basic image encoder.

B.1.3 Aggregator

To aggregate two input encodings $P \in \mathbb{R}^{|P| \times H}$ and $Q \in \mathbb{R}^{|Q| \times H}$, we use the standard multi-head attention mechanism (Vaswani et al., 2017). Specifically, we use P as the *query*, Q as the *key* and *value*. This results an output $P_Q \in \mathbb{R}^{|P| \times H}$, where at every time step $i \in [0, |P|)$, P_Q^i is the weighted sum of

Q, the weight is the attention of P^i on Q. We refer readers to (Vaswani et al., 2017) for detailed information.

We apply a residual connection on top of the multi-head attention mechanism in order to maintain the original information contained in P. Specifically,

$$h_{PQ} = \text{Tanh}(\text{Linear}([P_Q; P])), \quad (3)$$

where $h_{PQ} \in \mathbb{R}^{|P| \times H}$.

B.1.4 Text Decoder

We use a transformer-based text decoder to generate text. The decoder consists of a word embedding layer, a stacked transformer blocks and a projection layer.

Similar to the text encoder, the embedding layer is initialized with random embedding matrix. Inside the transformer block, there is one self attention layer, one multi-head attention layer and a 2-layer MLP with ReLU non-linear activation functions in between. Taking word embedding vectors as input, the self-attention layer first generates a contextual encoding vectors for the words. These vectors are then fed into the multi-head attention layer, to compute attention with representations produced by the encoder, which contains information from multiple modalities. The resulting vectors are fed into the 2-layer MLP. The block hidden size of this transformer is 128.

Subsequently, the output of the stacked transformer blocks is fed into the projection layer, which is a linear transformation with output size same as the vocabulary size. We follow (Press and Wolf, 2017), tying the input embeddings and this projection layer. The logits resulted from the projection layer are then normalized by a softmax to generate a probability distribution over all tokens in the GPT-2 vocabulary.

Following common practice, we use a mask to prevent the decoder transformer to access "future" information during training. We set the max number of generated tokens to be 30. During inference, the decoder will stop generating whenever generates the *end-of-sequence* special token, or exhausts all its budget.

B.2 Hierarchical Latent Language Policy Agent (HLLP)

B.2.1 Describer

As briefly mentioned in Section 4, the describer module "summarizes" a demonstration into a short text, where a demonstration typically a sequence of multi-modal transitions. As shown in Figure 5, at every step t of a demonstration, the transition contains the symbolic images at previous step and current step: M_{t-1} and M_t, and the text input $o_t = [a_{t-1}; r_{t-1}; I_t]$, where a_{t-1}, r_{t-1}, I_t denote the action taken at previous step, the resulting reward, and the inventory state at current step, respectively.

We first encode the text input with an text encoder described in Appendix B.1.1, similarly, we encode the image inputs with an consecutive image encoder described in Appendix B.1.2. We subsequently use two attention blocks described in Appendix B.1.3 to compute the image encoding's attention over text (tokens), and vice versa, the text encoding's attention over image (pixels). We average both the attention-aggregated outputs, resulting $h_{\text{img}\rightarrow\text{text}} \in \mathbb{R}^{\times H}$ and $h_{\text{text}\rightarrow\text{img}} \in \mathbb{R}^{\times H}$, to compute the overall representation of this time step:

$$h_t = \text{Tanh}(\text{Linear}([h_{\text{img}\rightarrow\text{text}}; h_{\text{text}\rightarrow\text{img}}])), \quad (4)$$

where $h_t \in \mathbb{R}^{\times H}$, $H = 128$ is the hidden size.

At the episode level, we use a Transformer-based encoder, with similar architecture to the one in our text encoder. Specifically, the episode encoder is a stacked 2-layer Transformer blocks, which outputs $h_{\text{demo}^i} \in \mathbb{R}^{|\text{demo}^i| \times H}$, $|\text{demo}^i|$ is the number of steps of a demonstration demo^i, H is hidden size.

Finally, we use a text decoder, as described in Appendix B.1.4, to generate text descriptions.

In the describer module, we use a 2-layer text encoder, a 5-layer image encoder, a 2-layer episode encoder, and a 3-layer decoder.

B.2.2 Instructor

As shown in Figure 5, the instructor consists a text encoder, a basic graph encoder, an attention mechanism, a text decoder, and a new instruction classifier.

Specifically, at a game step t, the image encoder takes the image input M_t as input, generates image representations $v_t \in \mathbb{R}^{hw \times H}$, where h and w are the height and width of the image. At the same time, the text encoder encodes the text input $o_t = [D; I_t; \text{Instr}_{t-1}]$, where D, I_t and Instr_{t-1} denote the task description (either generated by the describer, or provided by an oracle), the inventory state at current step, and the instruction at previous game step. The text encoder outputs $w_t \in \mathbb{R}^{|o_t| \times H}$. Next, an attention block as described in Appendix B.1.3 aggregates v_t and w_t, resulting $s_t \in \mathbb{R}^{|o_t| \times H}$ that contains information from both modalities, where $|o_t|$ denotes number of tokens in o_t.

The new instruction classifier is an MLP switch module that decides whether or not the instruction generated at previous step is still valid (i.e., is it necessary to generate a new instruction):

$$\begin{aligned} s'_t &= \text{MaskedMean}(s_t), \\ p_t &= \text{Argmax}(L^1(\text{Tanh}(L^0(s'_t)))). \end{aligned} \tag{5}$$

In which, L^0 and L^1 are linear transformations with hidden size of 128 and 2, respectively. The output $p_t \in \{0, 1\}$ is the discrete switch.

In the case where $p_t = 0$, we directly pass the instruction generated at previous step along as output; otherwise, a text decoder as described in Appendix B.1.4 will generate a new instruction word-by-word conditioned on s_t.

In the describer module, we use a single layer text encoder, a 2-layer image encoder, and a 2-layer decoder. The text encoder and image encoder are tied with the corresponding layers in the executor module. During training, we do not update the image encoder.

B.2.3 Executor

Given the intermediate level text instruction, our executor module translates them into low level actions to interact with the environment. As shown in Figure 5, the executor consists a text encoder, a basic graph encoder, an attention block, and a recurrent action generator.

Similar to the instructor module, the image encoder and text encoder convert image input (M_t) and text input (I_t and Instr$_t$]) into hidden representations. Note in the executor, to facilitate interaction between the instruction Instr$_t$] with other text inputs, we encode I_t and Instr$_t$] separately and aggregate them using an attention mechanism.

Subsequently, given the image representation v_t and the aggregated text representation w_t, we apply attention block (as described in Appendix B.1.3) from both directions:

$$\begin{aligned} h_{vw} &= \text{Attention}(v_t, w_t), \\ h_{wv} &= \text{Attention}(w_t, v_t), \\ h'_{vw} &= \text{MaskedMean}(h_{vw}), \\ h'_{wv} &= \text{MaskedMean}(h_{wv}), \\ s_t &= \text{Tanh}(\text{Linear}([h'_{vw}; h'_{wv}])), \end{aligned} \tag{6}$$

in which, $s_t \in \mathbb{R}^H$, $H = 128$ is hidden dimension.

In order to encourage the action generator to condition on history information, we equip it with a recurrent memory (Cho et al., 2014):

$$s_{1:t} = \text{GRU}(s_t, s_{1:t-1}), \tag{7}$$

the hidden size of the GRU is 128. We stack an MLP on top of the recurrent memory to obtain the output distribution over all actions:

$$\begin{aligned} h_t &= \text{Tanh}(\text{Linear}(s_{1:t})), \\ p_{a_t} &= \text{Softmax}(\text{Linear}(h_t)), \\ a_t &= \text{Argmax}(p_{a_t}). \end{aligned} \tag{8}$$

In the executor module, we use a single layer text encoder and a 2-layer image encoder. The text encoder and image encoder are tied with the corresponding layers in the instructor module. During training, we do not update the text encoder.

B.3 Latent Language Description Only Baseline (LLD)

The LLD baseline shares the same describer architecture, and a similar executor architecture with HLLP, its main difference is the absence of an instructor.

In its executor, at a game step t, the inputs are an image M_t and a short text $o_t = [D; I_t]$, where D is the description generated by the describer (or the oracle description during training), I_t is the agent's inventory state. To obtain the text representation w_t, the LLD agent simply encode o_t with the text encoder as described in Appendix B.1.1, without performing attention between D and I_t (as in HLLP). The rest of the executor components are identical to HLLP (Appendix B.2.3).

In the LLD baseline, we use a single layer text encoder and a 2-layer image encoder.

B.4 Nonverbal Baseline (NV)

In the nonverbal baseline, we do not use language as latent representations between modules. Specifically, given a demonstration demo^i, we use a describer similar to the one outlined in Appendix B.2.1, but without decoding the demonstration representation into text. The output of the describer is $h_{\text{demo}^i} \in \mathbb{R}^{|\text{demo}^i| \times H}$, where $|\text{demo}^i|$ is the number of steps in demo^i, H is hidden size.

In our nonverbal baseline's executor, at game step t, a text encoder encodes the inventory state I_t into w_t; an image encoder encodes an input image

M_t into v_t. We use multi-head attention blocks (Appendix B.1.3) to aggregate information carried by image (v_t), text (w_t), and demonstration representation (h_{demo^i}):

$$\begin{aligned} h'_{\text{demo}^i} &= \text{MaskedMean}(h_{\text{demo}^i}), \\ h_{\text{demo}\to\text{img}} &= \text{Attention}(h'_{\text{demo}^i}, v_t), \\ h_{\text{demo}\to\text{text}} &= \text{Attention}(h'_{\text{demo}^i}, w_t), \\ h_{\text{text}\to\text{img}} &= \text{Attention}(w_t, v_t), \\ h_{\text{img}\to\text{text}} &= \text{Attention}(v_t, w_t), \\ h'_{\text{text}\to\text{img}} &= \text{MaskedMean}(h_{\text{text}\to\text{img}}), \\ h'_{\text{img}\to\text{text}} &= \text{MaskedMean}(h_{\text{img}\to\text{text}}). \end{aligned} \tag{9}$$

Subsequently, we use an MLP to combine them:

$$\begin{aligned} h_{\text{combined}} =& [h'_{\text{demo}^i}; \\ & h_{\text{demo}\to\text{img}}; h_{\text{demo}\to\text{text}}; \\ & h'_{\text{text}\to\text{img}}; h'_{\text{img}\to\text{text}}], \\ s_t =& \text{Tanh}(\text{Linear}(h_{\text{combined}})), \end{aligned} \tag{10}$$

in which, the output $s_t \in \mathbb{R}^H$, $H = 128$ is hidden dimension.

The remainder of the executor is identical to the executor used in the HLLP agent, as described in Appendix B.2.3.

In the nonverbal baseline, we use a single layer text encoder and a 2-layer image encoder.

C Training and Implementation Details

For all experiments, we use *Adam* (Kingma and Ba, 2015) as the optimizer. The learning rate is set to 0.001 with a clip gradient norm of 5.

C.1 Describer Training via Supervised Learning

We use a set of pre-collected expert demonstrations paired with ground-truth descriptions to train the describer module in HLLP. Because demonstrations are long sequences of agent transitions, which can be memory consuming, we cut long demonstrations and only keep their last 100 transition steps. Since the length of demonstration varies, we speed up training by sorting the data points by their demonstration length, and split them by buckets with a bucket size of 2,000. For every mini-batch (we use a batch size of 20), we first randomly sample a bucket, then randomly sample a batch of data point from that bucket. We train the describer for 5 million episodes (250,000 batches).

C.2 Description Follower Training via DAgger

We train the description follower modules (instructor and executor in HLLP, executor in LLD, and the entire nonverbal baseline) using DAgger (Ross et al., 2011), an imitation learning method.

Specifically, during the training process, the agent starts with totally following the expert demonstrations, then we gradually let the agent to take over the control. We collect such trajectories (i.e., sequences of transitions, along the expert demonstrations if the agent takes over control), without updating the network, into a replay buffer of size 500,000. We periodically (after every 5 data collection steps) sample batches of transitions from the replay buffer, and update the network. Specifically, following the training strategy used in the recurrent DQN literature (Hausknecht and Stone, 2015; Yuan et al., 2018), we sample batches of transition sequences (of length 8), we use the first 4 transitions to estimate the recurrent states, and the last 4 transitions for updating the model parameters. We use a mini-batch of size 32 in replay data collection, and a batch size of 64 for update. We linearly anneal the fraction of expert assistance in DAgger from 100% to 1% within 500,000 episodes.

When training the HLLP agent, as depicted in Figure 5, we tie the encoder parameters between the instructor and the executor. In which, the image encoder is only updated through the executor loss, whereas the text encoder is only updated through the instructor loss. To stabilize the training, we update the instructor and executor modules in an alternate manner, with a frequency of 2,000 (experience data collection) episodes.

We train the description following agents for 1 million episodes maximally, however, in practice, the agents mostly converge sooner. We set an patience of 100,000 episodes, the training process will terminate if there is no improvement within this period.

D Supplementary Results

Table 1 Shows describer module exact match performance against gold references in all splits and task categories.

Table 7 shows full task completion performance by agents on the hidden terrain destination generalization set set decomposed by task category. Table 8 shows the same for the length generalization set.

	Valid		Eval	
	Full Task	End Goal	Full Task	End Goal
Random Split	**84.3**	**92.4**	**69.3**	**75.7**
Navigation	10.1	10.6	0.9	0.9
Crafting	98.0	98.9	87.4	88.0
Craft then Nav	88.1	99.4	84.0	88.1
Building on Terrain	83.0	92.9	63.8	71.7
Covering Terrain	71.5	98.5	59.5	84.3
Clearing Items	95.2	95.2	37.0	37.5
Hidden Subtask	**84.8**	**91.4**	**14.5**	**15.8**
Crafting	97.8	98.4	36.1	36.4
Craft then Nav	88.2	98.3	32.8	32.8
Building on Terrain	84.6	93.0	6.4	7.5
Covering Terrain	74.9	97.6	7.2	12.1
Hidden Use Case	**84.1**	**90.3**	**19.7**	**22.2**
Crafting	95.1	95.6	29.1	29.3
Craft then Nav	90.4	99.7	46.2	47.5
Building on Terrain	84.6	93.9	20.3	23.5
Covering Terrain	75.3	97.7	4.0	7.4
Hidden Terrain Destination	**84.9**	**91.8**	**0.0**	**0.0**
Building on Terrain	84.0	94.4	0.0	0.0
Covering Terrain	71.9	97.7	0.0	0.0
Hidden Length	**85.2**	**92.0**	**69.7**	**92.9**
Crafting	97.3	98.1	95.6	99.1
Craft then Nav	89.9	99.6	89.1	100.0
Building on Terrain	82.9	93.2	74.4	91.0
Covering Terrain	76.8	97.1	58.9	92.6
Clearing Items	98.8	99.1	100.0	100.0

Table 6: Expanded performance of Describer module against gold references in all splits and task categories. Validation scores for task categories not in an eval set are not shown.

	NV Baseline	LLD	HLLP
Demonstration Following			
Overall	1.6 ± 0.9	4.6 ± 0.5	3.7 ± 0.7
Building on Terrain	2.5 ± 1.5	7.4 ± 0.8	6.0 ± 1.1
Covering Terrain	0.0 ± 0.0	0.1 ± 0.0	0.0 ± 0.0
Ground Truth Description Following			
Overall	–	1.8 ± 1.2	2.8 ± 1.2
Building on Terrain	–	2.9 ± 2.0	4.5 ± 1.9
Covering Terrain	–	0.0 ± 0.0	0.1 ± 0.1
Ground Truth Instruction Following			
Overall	–	–	35.3 ± 7.2
Building on Terrain	–	–	55.1 ± 11.2
Covering Terrain	–	–	3.1 ± 0.8

Table 7: Performance on hidden terrain destination split broken down by task category

Figure 8 depicts example unrolled trajectories produced by the oracle. Figure 9 depicts example failure cases by the HLLP agent on the generalization splits.

	# Tasks	NVB	LLD	HLLP
Demonstration Following				
Overall		6.0 ± 2.1	62.6 ± 3.8	57.9 ± 9.0
Crafting	1905	29.9 ± 8.1	82.5 ± 3.5	86.0 ± 11.6
Build on Terr	6330	4.9 ± 2.9	58.9 ± 4.5	69.6 ± 13.2
Cover Terr	7830	0.3 ± 0.4	59.7 ± 3.9	41.1 ± 5.4
Craft then Nav	165	36.4 ± 3.8	91.8 ± 4.6	88.6 ± 8.9
Clear Itm	105	18.5 ± 9.1	87.8 ± 5.1	42.1 ± 11.3
Ground Truth Description Following				
Overall		–	65.7 ± 4.1	60.9 ± 9.1
Crafting	1905	–	82.8 ± 3.4	86.3 ± 11.6
Build on Terr	6330	–	62.4 ± 4.9	75.1 ± 13.8
Cover Terr	7830	–	63.3 ± 4.1	42.9 ± 5.3
Craft then Nav	165	–	91.8 ± 4.6	88.4 ± 9.2
Clear Itm	105	–	87.8 ± 5.1	42.1 ± 11.3
Ground Truth Instruction Following				
Overall		–	–	96.6 ± 1.3
Crafting	1905	–	–	97.4 ± 1.9
Build on Terr	6330	–	–	97.1 ± 1.2
Cover Terr	7830	–	–	95.9 ± 1.4
Craft then Nav	165	–	–	98.7 ± 0.9
Clear Itm	105	–	–	96.6 ± 1.6

Table 8: Length generalization results

```
build fence on silver flooring, then reach the jeweler.
avoid walking on the field. walking on the lava will reward you.
================================================
I0: cut wood, stepping on lava and avoiding field (9 steps)
I1: get stone, stepping on the lava and avoiding the field
    (3 steps)
I2: get string, stepping on the lava and avoiding the field
    (4 steps)
I3: get spade, stepping on the lava and avoiding the field
    (4 steps)
I4: make stick, stepping on the lava and avoiding the field
    (6 steps)
I5: make wood slats (1 steps)
I6: make stone pickaxe, stepping on the lava and avoiding
    the field (7 steps)
I7: get coal, stepping on the lava and avoiding the field
    (4 steps)
I8: get silver ore, stepping on the lava and avoiding the
    field (11 steps)
I9: light furnace, stepping on the lava and avoiding the
    field (3 steps)
I10: smelt silver (1 steps)
I11: place silver flooring on empty cell, stepping on the lava
    and avoiding the field (3 steps)
I12: build fence on silver flooring (1 steps)
I13: go to jeweler, stepping on the lava and avoiding the
    field (5 steps)
game ended after 62 steps
```

```
make net and place silver flooring covering all the
water in any order. avoid walking on the field.
================================================
I0: cut wood, avoiding the field (5 steps)
I1: get stone, avoiding the field (7 steps)
I2: get string, avoiding the field (7 steps)
I3: get spade, avoiding the field (7 steps)
I4: make firewood, avoiding the field (6 steps)
I5: make stick (1 steps)
I6: make net (1 steps)
I7: make stone pickaxe, avoiding the field (5 steps)
I8: get silver ore, avoiding the field (2 steps)
I9: light furnace, avoiding the field (10 steps)
I10: smelt silver (1 steps)
I11: place silver flooring covering water, avoiding the field
    (4 steps)
I12: place silver flooring covering water, avoiding the field
    (3 steps)
I13: place silver flooring covering water, avoiding the field
    (3 steps)
I14: place silver flooring covering water, avoiding the field
    (3 steps)
I15: place silver flooring covering water, avoiding the field
    (3 steps)
I16: place silver flooring covering water, avoiding the field
    (3 steps)
I17: place silver flooring covering water, avoiding the field
    (3 steps)
game ended after 88 steps
```

```
dig dirt covering all the water, then reach the workspace.
================================================
I0: get spade (8 steps)
I1: dig dirt covering water (2 steps)
I2: dig dirt covering water (2 steps)
I3: dig dirt covering water (3 steps)
I4: dig dirt covering water (2 steps)
I5: dig dirt covering water (2 steps)
I6: dig dirt covering water (2 steps)
I7: dig dirt covering water (3 steps)
I8: dig dirt covering water (2 steps)
I9: dig dirt covering water (3 steps)
I10: dig dirt covering water (2 steps)
I11: dig dirt covering water (2 steps)
game ended after 32 steps
```

```
clear all of the grasses and the irons.
================================================
I0: cut wood (6 steps)
I1: get stone (5 steps)
I2: get string (5 steps)
I3: make stick (12 steps)
I4: make stone pickaxe (2 steps)
I5: make scythe (1 steps)
I6: get iron ore (4 steps)
I7: get iron ore (3 steps)
I8: cut hay (4 steps)
I9: cut hay (4 steps)
I10: cut hay (10 steps)
game ended after 56 steps
```

```
build pig barn on dirt and build diamond house on silver flooring
in any order.
================================================
I0: cut wood (8 steps)
I1: get stone (3 steps)
I2: get string (2 steps)
I3: get spade (12 steps)
I4: make stick (12 steps)
I5: make trap (1 steps)
I6: make net (1 steps)
I7: make wood slats (1 steps)
I8: make stone pickaxe (7 steps)
I9: catch pig (3 steps)
I10: make scythe (3 steps)
I11: get coal (16 steps)
I12: get iron ore (15 steps)
I13: get silver ore (5 steps)
I14: cut hay (5 steps)
I15: dig dirt on empty cell (2 steps)
I16: light furnace (12 steps)
I17: build pig barn on dirt (13 steps)
I18: smelt iron (12 steps)
I19: smelt silver (1 steps)
I20: make iron pickaxe (4 steps)
I21: get diamond ore (3 steps)
I22: place silver flooring on empty cell (5 steps)
I23: build diamond house on silver flooring (2 steps)
game ended after 148 steps (task was completed)
```

```
place diamond flooring on field, then reach the lumbershop.
================================================
I0: cut wood (11 steps)
I1: get stone (5 steps)
I2: get spade (4 steps)
I3: make stick (6 steps)
I4: make stone pickaxe (7 steps)
I5: get coal (5 steps)
I6: get iron ore (7 steps)
I7: light furnace (6 steps)
I8: smelt iron (1 steps)
I9: make iron pickaxe (6 steps)
I10: get diamond ore (3 steps)
I11: place diamond flooring on field (5 steps)
I12: go to lumbershop (4 steps)
game ended after 70 steps
```

Figure 8: Example unrolled oracle trajectories

Hidden Subtask

```
 erect pig shrine.
================================================
I.0: cut wood
I.1: get stone
I.2: get string
I.3: make stick
I.4: make trap
I.5: make net
I.6: make stone pickaxe
I.7: catch pig
I.8: get coal
I.9: get iron ore
I.10: get silver ore
I.11: light furnace
I.12: smelt iron
I.13: make iron pickaxe
I.14: get gold ore <pig shrine now eligible>
I.15: erect pig shrine <agent erects iron shrine>
I.16: erect pig shrine <agent erects iron shrine on same cell>
<repeats until time limit>
game ended after 300 steps (task incomplete)
```

```
build diamond house.
================================================
I.0: cut wood
I.1: get stone
I.2: get string
I.3: make stick
I.4: make wood slats
I.5: make stone pickaxe
I.6: get coal
I.7: get iron ore
I.8: light furnace
I.9: smelt iron
I.10: make iron pickaxe
I.11: get gold ore
I.12: get diamond ore <diamond house now eligible>
I.13: erect diamond shrine <agent erects diamond shrine unsuccessfully>
I.13: erect diamond shrine <agent erects diamond shrine unsuccessfully>
<repeats until time limit>
game ended after 300 steps (task incomplete)
```

Hidden Use Case

```
place iron flooring covering all the lava and erect pig shrine
on silver flooring in any order.
================================================
I.0: cut wood
I.1: get stone
I.2: get string
I.3: get spade
I.4: make stick
I.5: make trap
I.6: make net
I.7: make stone pickaxe
I.8: catch pig
I.9: get coal
I.10: get iron ore
I.11: get silver ore
I.12: light furnace
I.13: smelt iron
I.14: smelt silver
I.15: make iron pickaxe
I.16: place iron flooring covering lava
I.17: place iron flooring covering lava
I.18: place iron flooring covering lava
I.19: place iron flooring covering lava <lava fully covered>
I.20: place iron flooring covering lava
<repeats until time limit>
game ended after 300 steps (task incomplete, no pig shrine)
```

```
build chicken barn on road and get gold ore in any order.
================================================
I.0: cut wood
I.1: get stone
I.2: get string
I.3: catch chicken
I.4: make stick
I.5: make wood slats
I.6: make stone pickaxe
I.7: make scythe
I.8: get coal
I.9: get iron ore
I.10: cut hay
I.11: light furnace
I.12: build chicken barn on empty cell
I.13: smelt iron
I.14: make iron pickaxe
I.15: get gold ore
<repeats until time limit>
game ended after 300 steps (task incomplete, barn not in road)
```

Hidden Terrain Destination

```
place silver flooring covering all the water.
================================================
I.0: cut wood
I.1: get stone
I.2: get spade
I.3: make stick
I.4: make stone pickaxe
I.5: get coal
I.6: get silver ore
I.7: light furnace
I.8: smelt silver
I.9: place silver flooring covering field
<repeats until time limit>
game ended after 300 steps (task incomplete, water not covered)
```

```
build fence on water.
================================================
I.0: cut wood
I.1: get string
I.2: make wood slats
I.3: build fence on empty cell
<repeats until time limit>
game ended after 300 steps (task incomplete, fence not on water)
```

Figure 9: Example agent failure cases on generalization splits