LEARNING RATIONAL SKILLS FOR PLANNING FROM DEMONSTRATIONS AND INSTRUCTIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

We present a framework for learning compositional, rational skill models (RatSkills) that support efficient planning and inverse planning for achieving novel goals and recognizing activities. In contrast to directly learning a set of policies that map states to actions, in RatSkills, we represent each skill as a *subgoal* that can be executed based on a planning subroutine. RatSkills can be learned by observing expert demonstrations and reading abstract language descriptions of the corresponding task (e.g., *collect-wood then craft-boat then go-across-river*). The learned subgoal-based representation enables inference of another agent's intended task from their actions via Bayesian inverse planning. It also supports planning for novel objectives given in the form of either temporal task descriptions or black-box goal tests. We demonstrate through experiments in both discrete and continuous domains that our learning algorithms recover a set of RatSkills by observing and explaining other agents' movements, and plan efficiently for novel goals by composing learned skills.

1 INTRODUCTION

Being able to decompose complex tasks into sub-goals can help long-term planning. Consider the example in Figure 1, planning to craft a boat from scratch is hard, as it requires a long-term plan going from collecting materials to crafting boats, but it can be made easier if we know that "having an axe" and "having wood" are useful sub-goals. Planning hierarchically with these subgoals can substantially reduce the search required. It is also helpful if we have knowledge about when a subgoal can be easily achieved. For example, knowing that crafting boats needs the initial condition "having wood" is useful because it allows us to further decompose the subgoal of "having a boat" into two stages: "collecting wood" and "crafting boat."

Learning useful subgoals and their initial conditions (i.e., a skill concept as a whole) from data is important but very hard with out any supervision. The most prevalent strategy for doing this is to learn a set of low-level policies, which can be recombined to solve new high-level problems. The policy-based approach rests on determining a basis of fixed policies that are rich enough to solve a variety of problems. This is generally challenging, because it can require a large set of policies and because there are few clear learning signals indicating what a good decomposition might be.

In this work, we propose *Rational Skill Models* (RatSkills), an alternative approach to policy learning that addresses these two problems. First, we train the system with very weak supervision, in the form of a small number of *unsegmented* demonstrations of complex behaviors paired with abstract logical descriptions of the action sequence, which uses terms that are initially unknown to the agent, much as an adult might narrate the high-level steps they are taking when demonstrating a cooking recipe to a child. These action-terms indicate important *subgoals* in the action sequence, and our agent learns to recognize those subgoals. Second, rather than attempting to learn a fixed policy to reach each *subgoal*, our system uses planning to reach each subgoal, which retains the advantages of planning at the lower level as well: it has an easier learning task and can leverage the planner to easily generalize over a wide range of initial states and domains.

In Fig. 1: given a sequence of low-level state-action pairs, and a corresponding abstract logical description of the task, *collect-wood then craft-boat then go-across-river*, our system learns to decompose the observed trajectory into fragments, each of which corresponds to a particular subgoal named in the description. Furthermore, it learns an abstract description of individual subgoals o: the initial condition I_o and the goal condition G_o, denoting the conditions that are satisfied before

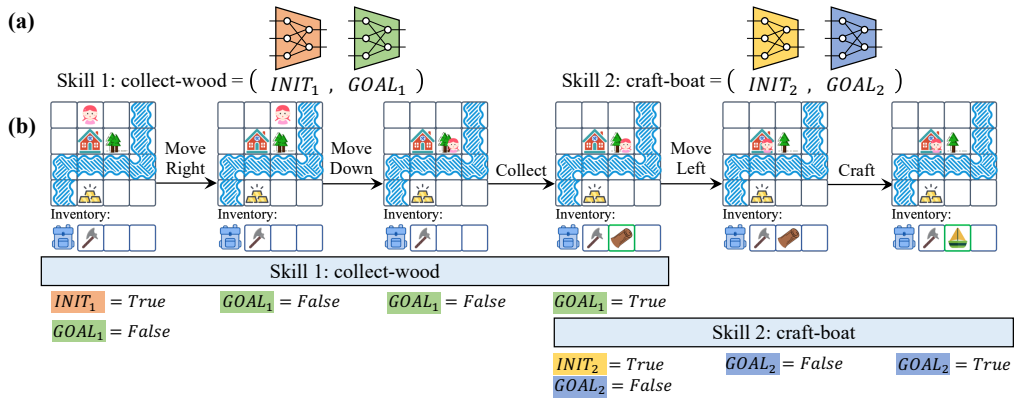

Figure 1: Interpreting a demonstration and its description in terms of RatSkills: (a) Each RatSkill consists of two conditions I_o and G_o. (b) The system infers a transition to the next skill if both the G condition of the current skill and the I condition of the next skill are satisfied. Such transition rules can be used to interpret demonstrations and to plan for tasks that require multiple skills to achieve.

and after the execution of the skill. I_o and G_o jointly characterize the important state transition that happened as a result of the skill execution, and the subgoal-based characterization of the skills allows them to be implemented via planning, which, as explained below, generates *rational* action sequences at the low-level, and also allows them to be combined via planning, allowing them to be flexibly recomposed in novel situations, for example, in this domain, to craft a house.

In order to learn RatSkills from limited and weakly-labeled data, our framework relies on the key assumption that the expert demonstrations are *rational*: experts generate trajectories to accomplish their target task while approximately minimizing their total cost, although they may make randomized errors in execution. The skill representation I_o and G_o are learned iteratively, based on a Bayesian inverse planning mechanism. Given candidate initial state and task classifiers, we first use a built-in planner to obtain *rational* trajectories for completing the task. Next, we compare these trajectories with the observed demonstration and quantify the rationality score of the demonstration. Finally, we update the weights of the classifiers by maximizing the rationality score using gradient descent.

We evaluate RatSkills on two benchmarks: CraftWorld (Chen et al., 2021), a grid-world domain with a rich set of object crafting tasks, and Playroom (Konidaris et al., 2018), a 2D continuous domain with geometric constraints. We also show that RatSkills can be extended to an image-based representation of CraftWorld, demonstrating its generality with respect to input representations. Our evaluation focuses two aspects of the learned skill concepts. First, our model significantly outperforms baselines on planning tasks where the agent needs to generate trajectories itself to accomplish a given task. Another important application of RatSkills is to create a language interface for human-robot communication, which includes robots interpreting human actions and humans instructing robots by specifying a sequence of skill concepts to execute. Our model enables compositional generalization through flexible re-composition of learned skill concepts, which allows the robot to interpret novel action sequences and execute novel instructions better than the baseline approaches.

2 Related Work

Modular skill models. There have been a significant number of recent approaches that use deep neural networks to construct modular skills models for interaction. Researchers have proposed models for learning these models by simultaneously looking at the state-action sequence and reading task specifications in the form of skill sequences (Corona et al., 2021; Andreas et al., 2017; Andreas & Klein, 2015), programs (Sun et al., 2020), and linear temporal logic (LTL) formulas (Bradley et al., 2021; Sadigh et al., 2014; Toro Icarte et al., 2018; Tellex et al., 2011). However, they either require additional annotation for the segmenting the sequence and associating fragments with labels in the task description (Corona et al., 2021; Sun et al., 2020), or cannot learn models for planning and execution from demonstration (Tellex et al., 2011). By contrast, in RatSkills, we use a small but expressive subset of LTL sentences for describing tasks and propose to jointly learn skill models and segment the demonstration sequence based on the *rationality* assumption.

Our skill representation is also related to other models in domain control knowledge (de la Rosa & McIlraith, 2011), goal-centric skill primitives (Park et al., 2020), macro learning (Newton et al.,

2007), hierarchical reinforcement learning (HRL; Dietterich, 2000; Mehta, 2011), and methods that combine reinforcement learning and planning (Segovia-Aguas et al., 2016; Winder et al., 2020), but we focus on learning the skill models from demonstrations and instructions. Our planning algorithm is similar to other approaches: de la Rosa & McIlraith (2011); Botvinick & Weinstein (2014); Winder et al. (2020), but this is not the focus of this paper.

Learning from demonstration. The idea of learning from demonstration generally refers to building agents that can interact with the environment by observing demonstrations by other experts, usually in the form of state-action sequences. Techniques for learning from demonstration can be roughly categorized into three groups: policy function learning (Chernova & Veloso, 2007; Torabi et al., 2018), cost and reward function learning (Markus Wulfmeier & Posner, 2015; Ziebart et al., 2008), and learning high-level plans (Ekvall & Kragic, 2008; Konidaris et al., 2012). We refer to Argall et al. (2009) and Ravichandar et al. (2020) as comprehensive surveys. In this paper, we propose to learn compositional skill models that support planning, and compare with methods that directly learns policy functions and cost functions. In contrast to approaches for learning high-level plans, which mostly focus on directly learning primitive action sequences, our representation, RatSkills are more general skill descriptions that are used by planners to generates primitive action sequences. Moreover, unlike them, RatSkills do not use similarities between skills (Niekum et al., 2012) for segmenting the observed state-action sequences. In contrast, we segment the demonstration and associate skill labels for each fragment by assuming the expert is *rational*.

Inverse planning. Our model is also related to methods for inferring agents' intentions by observing their states and actions, by assuming agents are rationally selecting actions to achieve their goal. The technique of inverse planning addresses this problem by finding a task description t that maximizes the consistency between the agent's behavior and the synthesized plan for t (Baker et al., 2009). While existing work has largely focused on modeling the rationality of agents (Baker et al., 2009; Zhi-Xuan et al., 2020) and more expressive task description languages (Shah et al., 2018), it generally does not jointly consider how models for skills can be learned so that the agent itself can generate plans to achieve novel tasks. In RatSkills, we use the Bayesian inverse planning framework (Baker et al., 2009) for inferring agents' intentions and present learning algorithms that can learn *rational* skill models from demonstration.

Activity recognition. Hidden Markov Models (Brand et al., 1997) and other activity recognition approaches (Huang et al., 2016) focus on learning to understand tasks, which may also require task segmentation. Compared to them, RatSkills focus on acquiring models that can be used not only for activity recognition, but also for planning, both to complete specified task descriptions but also to plan when only the terminal goal condition is specified.

3 Planning and Learning of RatSkills

We begin by describing the basic problem formulation and model, illustrate how to use the model for both planning and inverse planning, and then present a paradigm for learning from a small number of weakly labeled trajectories.

3.1 Problem Formulation

We assume an environment in the form of a deterministic process with states, actions, transition function and cost function $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{C} \rangle$. $\mathcal{S}$ can be given in various forms, such as an object centric representation, in which each state is specified in terms of the values of a set of properties and relations applied to a universe of objects; the properties and relations are fixed for any given domain, but the universe may change for different problem instances and our model will generalize over problems with different numbers of objects. We will focus on this representation in the paper but will also show how our model can be extended to other representations, such as images. We assume that the primitive action space $\mathcal{A}$ can be used by a low-level planner to find state trajectories through the space: we illustrate both a discrete $\mathcal{A}$ and a continuous robot-motion $\mathcal{A}$. We further assume that the agent has access to the transition and cost models. We will say that a state sequence $\bar{s} = (s_1, \ldots, s_n) \in \mathcal{S}^n$ is *feasible*, if for all $i \in (1, \ldots, n-1)$, $\exists a \in \mathcal{A}. s_{i+1} = \mathcal{T}(s_i, a)$.

Rational skills. Our objective is to learn a set of *rational skills* (RatSkills), where each RatSkill has an atomic skill name o, and is specified in terms of a pair of classifiers (I_o, G_o), each of which maps $\mathcal{S}$ into Boolean values. In our formulation, we use neural logic machines (Dong et al., 2019) to represent these classifiers, because they provide flexible representation of first-order logic formulas with finite quantification, allowing RatSkills to generalize effectively to domains with different universes of

objects. We say a RatSkill is *applicable* at state s if $I_o(s) \wedge \left(\exists \bar{a} \in \bar{\mathcal{A}}.\ G_o(\mathcal{T}(s, \bar{a}))\right)$, where $\bar{\mathcal{A}}$ is the set of finite sequences of actions; that is, I_o is true for state s, and there is another state that is reachable from s in which G_o is true. In Appendix A.1, we provide a discussion on the importance of the initial conditions I_o in our abstration. Unlike the skills often defined in terms of unconstrained policies, our skills are *rational* in the sense that it requires reasoning to execute them. To execute RatSkill o from some state $s \in I_o$, we call a planner with G_o as the goal condition; as long as o is applicable, we will obtain a plan, which is then executed to reach some state $s' \in G_o$. Although this model-based skill representation pays a computational cost at execution time, this model-based approach exhibits generalization abilities that far exceed those of fixed policies.

Task language. The goal-centric skill representation supports flexible composition to form new tasks. We define a simple temporal language $\mathcal{TL}$ for task descriptions. Syntactically, all atomic skills o are in $\mathcal{TL}$; and for all $t_1, t_2 \in \mathcal{TL}$, $(t_1\ \textbf{then}\ t_2)$, $(t_1\ \textbf{or}\ t_2)$, and $(t_1\ \textbf{and}\ t_2)$ are all in $\mathcal{TL}$. Semantically, a feasible state sequence $\bar{s}$ satisfies a task description t, written $\bar{s} \models t$ when:

- If t is a *RatSkill* o, then $I_o(s_1)$ and $G_o(s_n)$.
- If $t = (t_1\ \textbf{then}\ t_2)$ then $\exists 0 < j < n$ such that $(s_1, \ldots, s_j) \models t_1$ and $(s_j, \ldots, s_n) \models t_2$.
- If $t = (t_1\ \textbf{or}\ t_2)$ then $\bar{s} \models t_1$ or $\bar{s} \models t_2$.
- If $t = (t_1\ \textbf{and}\ t_2)$ then $\bar{s} \models (t_1\ \textbf{then}\ t_2)$ or $\bar{s} \models (t_2\ \textbf{then}\ t_1)$.

Remark. It is important to note that $t_1\ \textbf{then}\ t_2$, specified in our goal-centric representation allows "interleaving" the actual execution steps for t_1 and t_2. Recall that the completion of a task t is indicated by a state s satisfying $G_t(s)$. This does not put any constraint on how an agent chooses to solve each task. For example, it does not restrict the agent from making progress towards the skill t_2 before the skill t_1 is completed. This is fundamentally different from concatenating the optimal policies for solving individual tasks. See Appendix A.1 for a detailed example.

The language $\mathcal{TL}$ can be viewed as a fragment of linear temporal logic (LTL) sentences. Specifically, there is no *always* quantifier in our language, so our fragment does not model task specifications that contain infinite loops. This simple grammar covers all instructions that we are considering in this paper but the approach could be directly extended to handle more complex constructions.

3.2 Task-Augmented Transition Models

We will frequently want to construct plans for, or evaluate plans relative to, a particular task $t \in \mathcal{TL}$. To do so, we will construct a deterministic, task-augmented transition model. This construction is a variation on those used to bias reinforcement-learning (Parr & Russell, 1998) and to plan to meet LTL specifications (Belta, 2016).

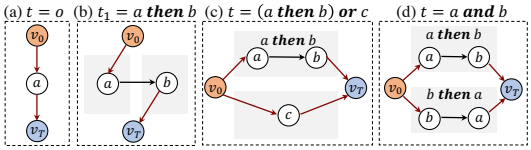

Figure 2: Illustrative example of how finite state machines (FSM) are constructed from task descriptions. The super-starting node v_0 and the super-terminal node v_T are highlighted.

We begin by constructing a finite state machine FSM_t based on the task specification t. Each FSM_t is a tuple $\langle V_t, E_t, VI_t, VG_t \rangle$, where V_t is the set of skill nodes in FSM_t, and E_t the edges. An edge $(x, y) \in E_t$ indicates that after executing skill x, the agent can switch to the execution of another skill y. All possible starting skill nodes of the state machine are indicated by $VI_t \subseteq V_t$. Analogously, the set of terminal nodes of the machine is $VG_t \subseteq V_t$. For simplicity, we add two special nodes: the super-starting node v_0 connecting to all starting nodes VI_t and the super-terminal node v_T, to which all terminal nodes VG_t are connected. The initial and goal conditions for both super nodes evaluate to true for all states in $\mathcal{S}$. FSM_t can be constructed based on the recursive structure of $\mathcal{TL}$. Fig. 2a-d shows example constructions for the four cases. The details can be found in Appendix A.1.

We now formally define our task-augmented transition model for a given task t, $\langle \mathcal{S}', \mathcal{A}', \mathcal{T}', \mathcal{C}' \rangle$, by composing the FSM with the basic environmental model. Concretely, $\mathcal{S}' = \mathcal{S} \times V_t$. We denote each task-augmented state as (s, v), where s is the environment state, and v indicates the skill being currently executed. The actions $\mathcal{A}' = \mathcal{A} \cup E_t$, where each action either corresponds to a primitive action $a \in \mathcal{A}$ or a transition in FSM_t, such as finishing executing the current skill and proceeding to the next skill. We further define $\mathcal{T}'\left((s, v), a\right) = \left(\mathcal{T}(s, a), v\right)$ if a is a primitive action in $\mathcal{A}$, while $\mathcal{T}'\left((s, v), a\right) = (s, v')$ if $a = (v, v') \in E_t$ is an edge in the FSM. Similarly, for the cost function,

$$\mathcal{C}'\left((s, v), a\right) = \begin{cases} \mathcal{C}(s, a) & \text{if } a \in \mathcal{A}, \\ -\lambda\left(\log G_v(s) + \log I_{v'}(s)\right) & \text{if } a = (v, v') \in E_t, \end{cases}$$

where λ is a hyperparameter. The key intuition behind the construction of $\mathcal{C}'$ is that the cumulative cost from v_0 to v_T is the summation of all primitive action costs added to the log-likelihood of all the skill sub-sequences satisfying their associated subtask description —at each skill transition, the world state s should satisfy both the goal condition of the current skill and the initial condition of the next skill. In principle, when I_v and G_v are Boolean-output classifiers, the transition cost is 0 for a valid transition and ∞ for an invalid transition. During learning, we approximate the "soft" version of classifiers with neural networks. The networks may output values in $[0, 1]$ indicating the probabilities of those conditions to be satisfied, which impose positive transition costs other than 0 and ∞.

3.3 Planning and Inverse Planning with RatSkills

We can use a set of RatSkills in three ways: planning to achieve a goal, planning to follow a sequence of instructions, and inverse planning to infer another agent's intended task from an action sequence. A critical basic component is planning to execute a single RatSkill o: finding a sequence of primitive actions to follow the sub-goals and initial condition set by the finite state machine.

Planning for a task described in $\mathcal{TL}$. In planning with RatSkills, our goal is to find an optimal-cost sequence of actions in the task-augmented transition model from (s_0, v_0) to (s_T, v_T) that completes the given task t, where s_0 is the initial state of the environment, and s_T is the state in $\mathcal{S}$ last reached by the agent. We make plans using slightly modified versions of A^* search, with a learned value function as the heuristic for seen tasks and a uniform heuristic for unseen tasks, for discrete domains and Rapidly-exploring Random Tree (RRT) for continuous domains. Both of these algorithms can be viewed as doing forward search to construct a trajectory from a given state to a state that satisfies the goal condition. Our extension to the algorithms handles the hierarchical task structure of the FSM.

Our modified A^* search maintains a priority queue of nodes to be expanded. Each node is associated with a evaluation which adds up the total cost of the agent reaching this state and an estimated cost-to-go (heuristic), in our case, either a uniform heuristic for novel tasks, or a heuristic produced by a learned value function approximator for previously-seen tasks. At each step, instead of always popping the task-augmented state (s, v) with the optimal evaluation, we first sample a skill node v uniformly in the FSM, and then choose the priority-queue node with the smallest evaluation value among all states $(\cdot, v)$. This search algorithm remains complete, but balances the time allocated to finding a successful trajectory for each RatSkill in the FSM. Similarly, for RRT, we maintain different RRTs for different skill nodes in the FSM and balance the number of nodes we expand for each tree. We include the implementation details of these modified versions of A^* and RRT in Appendix A.2

Planning for a goal state without a task description. RatSkills also support efficient planning for novel goals without a task description in $\mathcal{TL}$, by using a bias that the solution has a short description in $\mathcal{TL}$ to provide substantial search guidance. In this paper, we study the case where a black-box goal-state test G^* is provided to the algorithm. The task is formulated as finding a sequence of actions that leads to a state s where $G^*(s)$. We use RatSkills for this task with a hierarchical search mechanism. First, we enumerate candidate skill sequences, i.e., tasks in $\mathcal{TL}$ composed with only skills and the *then* connectives*. Next, we run parallel A^* search procedures for each candidate high-level skill plan. The algorithm will terminate and return a plan when any one of the downward refinements reaches a state that satisfies G^*. Ideally, we would be able to plan at the high level by chaining the skills, matching goals of one skill with the initiation sets of the next. We leave the integration of other informed search methods into the high-level skill space as future work. We also analyze the optimality of our algorithm in Appendix A.2.

Bayesian inverse planning. The set of RatSkills can also be used to infer another agent's intended task from their action sequence via Bayesian inverse planning. Concretely, the input to our algorithm is a state sequence $\bar{s}$ and an action sequence $\bar{a}$, such that $\forall i, s_{i+1} = \mathcal{T}(s_i, a_i)$. It is important to note that the state-action sequence only contains the environment states $\mathcal{S}$ and actions $\mathcal{A}$. However, we do not know the FSM state $v_i \in V_t$ associated with each state s_i, nor the FSM transition actions $e \in E_t$ (i.e. we do not assume the *segmentation* of the state-action sequence). The output of the algorithm is a ranking of a set of candidate tasks described in $\mathcal{TL}$, by how likely they are the intended task.

For each candidate task t, our inverse planning procedure starts by running forward search (A^* or RRT) from a seed node set composed of all task-augmented states (s, v) based on the task-augmented transition model of t, where $s \in \bar{s}$ and $v \in V_t$. This step produces a larger set of task-augmented

In practice, we use the idea of IDA: try all candidates with length 1, and then try ones with length 2, and then 3, . . . , until we find a path to satisfy the goal or reach the time limit.

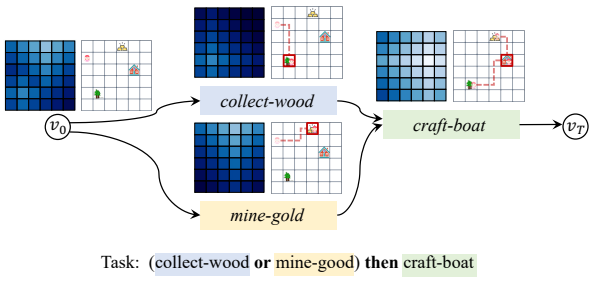

Figure 3: An example of the value function for task-augmented states on a simple FSM. $\max_{a \in \mathcal{A}} J_t(s, v, a)$ are plotted at each location at each FSM node. Deeper color indicates larger cost. Dotted lines illustrate one *rational* trajectory for each skill; red boxes indicate goals.

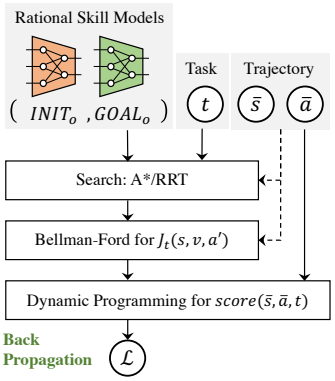

Figure 4: An overview of the training paradigm for RatSkills. The dashed line indicates that we only compute policy and J_t values for states along $\bar{s}$. See Section 3.4.

states $\mathcal{K}_t$. We run Bellman-Ford (analogous to value iteration for deterministic systems) on $\mathcal{K}_t$ and obtain a value function $J_t(s, v, a')$ (analog to the Q-value for Markov Decision Processes but in terms of cost) for all nodes along the observed state sequence $s \in \bar{s}$, all FSM states $v \in V_t$, and all actions $a' \in \mathcal{A}' = \mathcal{A} \cup E_t$ (i.e., all environment actions and state machine transitions in FSM_t). Fig. 3 shows an example of the value function over the task-augmented states.

Based on the J_t, we define the rationality of action $a' \in \mathcal{A}'$ at environment state s and FSM state v (α is also called the inverse rationality):

$$\text{Rat}\,(s, v, a', t) \coloneqq \frac{\exp\left(-\alpha \cdot J_t(s, v, a')\right)}{\int_{x' \in \mathcal{A}'} \exp\left(-\alpha \cdot J_t(s, v, x')\right)}$$

For discrete action spaces, the integral is simply a finite sum. In continuous action spaces, we use Monte Carlo sampling to compute the integral.

Since we do not know the segmentation of the state-action sequence, we use a dynamical programming process[†] to find an optimal assignment of FSM states for states in the trajectory $s \in \bar{s}$. Each candidate assignment yields a FSM state sequence $\bar{v} = \{v_i\}$ and a task-augmented action sequence $\bar{a}' = \{a'_i \in \mathcal{A}'\}$. Our dynamical programming process finds the optimal $\bar{s}'$ and $\bar{a}'$ that maximizes:

$$score(\bar{s}, \bar{a}, t) \coloneqq \max_{\substack{(\bar{v}, \bar{a}') \\ \text{derived from } (\bar{s}, \bar{a})}} \left\{ \sum_{\substack{a' = (v, v') \in \bar{a}' \\ \text{is an FSM transition}}} \left[\log G_v(s_i) + \log I_{v'}(s_i)\right] + \log \prod_i \text{Rat}\,(s_i, v_i, a'_i, t) \right\},$$

Essentially, it requires all FSM state transitions (v, v') at state s_i are valid, and the agent's actions are well explained. If we consider I_v and G_v as "soft" probabilities, the computation of $score$ finds $\bar{s}'$ and $\bar{a}'$ that maximize the rationality of primitive actions and the likelihood that FSM transitions are successful. We use $score(\bar{s}, \bar{a}, t)$ to rank all candidate tasks t.

Remark. We are using dynamic programming to determinate the transitions, and there are other formulations such as using stochastic transitions derived by I_v and G_v. These formulations can be unified using a framework of latent transition models. We will discuss them in Appendix A.3.

3.4 Learning RatSkills

We employ a contrastive-learning based approach to learn a set of RatSkills from paired task descriptions $t \in \mathcal{TL}$ and state-action sequences $(\bar{s}, \bar{a})$, which we assume are generated by a *rational* agent to accomplish t. Critically, the agent does not know the meanings of the skills in $\mathcal{TL}$—these will be learned in the form of the G_o classifiers. The intuition behind the learning method is that the model parameters should be adjusted so that each task description in the training set is a high-probability rational explanation of the accompanying state-action sequence. In order to learn the weights in the classifiers I_o and G_o, we define the following training objective:

$$\mathcal{L} = \sum_{(\bar{s}, \bar{a}, t) \in \mathcal{D}} score(\bar{s}, \bar{a}, t) + \gamma \cdot \log \frac{\exp\left(\beta \cdot score(\bar{s}, \bar{a}, t)\right)}{\sum_{t'} \exp\left(\beta \cdot score(\bar{s}, \bar{a}, t')\right)},$$

where $\mathcal{D}$ is the dataset we train on, and t' are uniformly sampled negative tasks in $\mathcal{TL}$ for the particular data point. Since the cost function for task-augmented transition models $\mathcal{C}'$ (and thus the computed

[†]The dynamic programming is similar to Dynamic time warping(DTW) warping trajectories into sequential skills, but we compute the "cost" at each segment instead of at each pair pf matched positions.

J_t values) is fully differentiable w.r.t. all I_o and G_o in RatSkills, $\mathcal{L}$ can be maximized simply using gradient descent—we are running back-propagation through the Bayesian inverse planning.

It is important to note that the computation of $\mathcal{L}$ only requires J values for states along the observed state-action sequences, instead of for the entire state-action space, making our model scalable to compositional state spaces with many objects and obstacles.

To summarize, Fig. 4 gives an overview of the the learning paradigm. Given the task description t, based on the current skill models, we first use A* search or RRT to solve for the optimal policy for t, and run value iteration on the search tree nodes to compute the value function J_t. Next, we compute the score of the observed trajectory $(\bar{s}, \bar{a})$ from J_t. Note that the computation of score only requires J_t values for states along the input trajectory $\bar{s}$. Since the derived score of $(\bar{s}, \bar{a})$ is fully differentiable w.r.t. the neural network parameters in RatSkills, we can use back-propagation to update them.

4 Experiments

We compare our model with other skill-learning approaches in two environments: Crafting World (Chen et al., 2021) and Playroom (Konidaris et al., 2018). In both cases, all models are trained on the inverse planning task given expert demonstrations generated by human-written programs, and are evaluated on two tasks: planning and inverse planning.

4.1 Setup

To evaluate planning, each algorithm is given a novel task, either specified in $\mathcal{TL}$, or as a black-box goal state classifier. The objective is to generate a trajectory of actions from the agent's current state to complete the task. In the inverse planning problem, we give each algorithm a state-action sequence, from which the algorithm should infer the intended task of the agent. We provide each algorithm with a list of candidate task descriptions in $\mathcal{TL}$ for which it outputs a ranking. For both tasks, we focus on evaluating the compositional generalization, i.e., inferring complex tasks that they have not seen during training.

Baselines. We compare our RatSkills, which learns goal-based representations, with two closely related groups of baselines, both of which learn from demonstration, but acquiring different underlying representations: IRL methods learn reward-based representations and behavior cloning methods directly learn policies. The implementation details of baselines are in the Appendix C.

Our max-entropy inverse reinforcement learning (IRL; Ziebart et al., 2008) baseline learns a task-conditioned reward function by trying to explain the demonstration. For planning, we use the built-in deep-Q-learning algorithm. For inverse planning, we rank all candidates by the consistency between the observation and the task-conditioned policy.

The behavior cloning (BC; Torabi et al., 2018) baseline directly learns a task-conditioned policy that maps the current state and the given task to an environment primitive action. For planning, we directly follow the task-conditioned policy. For inverse planning, we rank all candidate tasks by the consistency between the observation and the task-conditioned policy.

BC-FSM is the BC algorithm augmented with our FSM description of tasks. Compared with RatSkills, instead of segmenting the demonstration sequence based on the rationality, BC-FSM segments them based on how consistent each fragment is with the policy for the corresponding skill.

Figure 5: An illustration of the Playroom environment and a trajectory for the task: *turn-on-music then play-with-ball then turn-off-music*.

For the inverse planning task, we include an additional sequence classification baseline: LSTM. It uses two separate LSTM networks to encode the state-action sequence and the task description, respectively. Next, it uses a multi-layer perceptron (MLP) to compute a score for each task description.

4.2 Environments

Crafting World. Our first environment is Crafting World (Chen et al., 2021), a Minecraft-inspired crafting environment. The agent can move in a 2D grid world and interact with objects next to it, including picking up tools, mining resources, and crafting items. Mining in the environment typically requires tools, while crafting tools and other objects has preconditions, such as being close to a workstation or holding another specific tool. Thus, crafting a single item often takes multiple steps. In certain maps, there are also obstacles such as rivers (which requires boats to go across) and doors (which requires specific keys to open). We modified the rules from the original environment to make

Model	Crafting World		Playroom	
	Com.	Novel	Com.	Novel
IRL	36.5	1.8	28.3	9.6
BC	11.2	0.8	15.8	4.8
BC-FSM (ours)	5.2	0.3	38.2	31.5
RatSkills (ours)	**99.6**	**97.8**	**82.0**	**78.2**

Table 1: Results of the planning task, evaluated as the success rate in completing the specified task. The maximum number of expanded nodes for all planners is 5000. All models are trained on the *compositional* split, and tested on the *compositional* and the *novel* split.

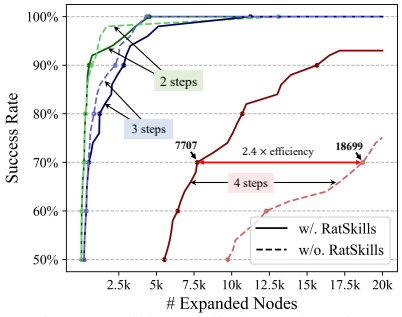

Figure 6: RatSkills applied to planning with a black-box goal test. We do evaluation on 3 planning tasks in the Crafting World environment. We use 100 random initial states for each task.

the tasks more challenging: we added crafting rules so that some craftings can share a common station; we also added doors and rivers where the agent must hold key or boat when crossing.

We define 26 primitive tasks, instantiated from templates of *grab-X*, *toggle-switch*, *mine-X*, and *craft-X*. While generating trajectories, all required items have been placed in the agent's inventory. For example, before mining wood, an axe must be already in the inventory. In this case, the agent is expected to move to a tree and execute the mining action. We also define 26 *compositional* tasks composed of the aforementioned primitive tasks. For each task we have 400 expert demonstrations.

We train all models on these primitive and *compositional* tasks and test them on two splits: *compositional* and *novel*. The *compositional* split contains novel state-action sequences of previously-seen tasks. The novel split contains rational state-action sequences for 12 novel tasks that are composed of the primitive tasks, but have never been seen during training.

Playroom. Our second environment is Playroom (Konidaris et al., 2018), a 2D maze with continuous coordinates and geometric constraints. Fig. 5 shows an illustrative example of the environment. Specifically, a 2D robot can make moves in a small room with obstacles. The agent has three degrees-of-freedom (DoFs): x and y direction movement, and a 1D rotation. The environment invalidates movements that cause collisions between the agent and the obstacles. Additionally, there are six objects randomly placed in the room, which the robot can interact with. For simplicity, when the agent is close to an object, the corresponding robot-object interaction will be automatically triggered.

Similar to the Crafting World, we have defined six primitive tasks (corresponding to the interaction with six objects in the environment) and eight compositional tasks (e.g., *turn-on-music then play-with-ball*). Similarly, we made up another eight novel tasks, and for each task we have 400 expert demonstrations. We will train different models on rational demonstrations for both the primitive and compositional tasks, and evaluate them on the compositional and novel splits.

4.3 Results

Planning for a task. Table 1 summarizes the results for planning in both environments. RatSkills outperforms all baselines. On the *compositional* split, our model achieves nearly perfect success rate in the Crafting World (99.6%) and high performance in Playroom (82.0). Comparatively, although the tasks have been presented during training of all baselines, their scores remain below 40%.

On the *novel* split, RatSkills outperforms all baselines by a larger margin than on the *compositional* split, in both environments. We observe that, in Crafting World, since some *novel* tasks contain longer descriptions than those in the *compositional* set, all baselines have almost zero success rate on them. Compared to behavior cloning methods that directly apply a learned policy, our model runs A*/RRT search based on the learned goal classifier with a learned heuristic. This suggests that learning goals from demonstration is more sample-efficient than learning policies, and generalizes better to new maps. Meanwhile, compared with IRL methods, our goal-centric representation has more compositional structure and thus performs better in these domains.

Planning with a black-box goal test. We also evaluate RatSkills on planning with a black-box goal test. These problems require a long solution sequence, making them too difficult to solve with blind search from an initial state. Since there is no task specification given, in order to solve the problems efficiently, it is critical to use RatSkills for search guidance. In our setting we assume the black-box goal test can be achieved by executing a few skills sequentially, and decomposing a long solution into sequential skills can significantly reduce the cost of search.

We manually designed three goal tests that each require sequential executions of 2, 3, and 4 atomic skills to complete. We run our hierarchical search algorithm based on RatSkills and a blind forward-search algorithm on 100 random initial states for all three tasks. Fig. 6 summarizes the result. Overall, RatSkills enable efficient search for plans. On relatively easier tasks (2 or 3 steps), search with and without RatSkills have similar efficiency. However, when the task becomes more complex (4 steps), our model significantly improves the search. For example, to reach 70% success rate, search with RatSkills only need to expand 7,707 nodes, while without RatSkills, it needs 18,699 (2.4×) nodes. Besides, the effectiveness of RatSkills grows as the map grows, because they provide cues of meaningful sub-goals in order to achieve the final goal. We leave learning models for more efficient skill-level planning (Konidaris et al., 2018) as future work.

	Crafting World		Playroom	
Model	Com.	Novel	Com.	Novel
LSTM	**100.0**	25.0	51.1	37.5
IRL	64.1	22.1	92.5	65.6
BC	85.6	20.8	72.5	45.0
BC-FSM (ours)	**100.0**	**100.0**	85.0	**98.8**
RatSkills (ours)	97.2	95.7	**97.5**	91.9

Table 2: Results of the inverse planning task, evaluated as the percentage of trajectories with a correct task prediction. All models are trained on the *compositional* data splits and evaluated on the *compositional* and the *novel* split.

Inverse planning. Table 2 shows the results for the inverse planning task. In general, models with FSM-based task representations (BC-FSM and RatSkills) outperform those that treat the task description directly as an additional input. In both environments and on both splits, RatSkills consistently perform well. In both environments, the LSTM, IRL and BC baselines all achieve high accuracy on previously-seen tasks (i.e., in the *compositional* split) but experience a huge performance drop when generalizing to the *novel* split. By contrast, BC-FSM and RatSkills significantly outperform others on the *novel* split. This suggests the effectiveness of our task-augmented transition models.

We see a strong performance of LSTM on the compositional split in Crafting World but fails on all other tasks. Its failure on two "novel" splits suggests that the LSTM encoding of the task specification does not generalize to unseen task specifications. The failure on the Playroom compositional split is probably because typically we have longer action sequences in continuous environments and we are dealing with continuous parameters instead of symbolic concepts..

Extension to image-based states. We include an additional experiment to test the applicability of RatSkills on image inputs. Specifically, instead of processing the symbolic states containing global, inventory, and map features, the model now uses a CNN to directly read the game interface. The image-based RatSkills reach 76.4% success rate on the compositional split in the Crafting World environment, which is lower than RatSkills based on symbolic states (99.6%) but still significantly outperforms all baselines based on symbolic states (the best baseline IRL reaches 36.5%).

Limitations. There are certain limitations of RatSkills in terms of computational cost and scalability. First, high-level planning is currently done with a blind search. Although we have already demonstrated its effectiveness with black-box goal test experiments, its applicability to longer-horizon tasks can be further extended by learning causal models at the skill level (Konidaris et al., 2018). Second, the skills learned by RatSkills are not "lifted." That is, the skills do not generalize to unseen objects (e.g., from "mine gold ore" to "mine iron." Designing skill representations that generalize to unseen objects is another meaningful direction. Third, in this first work we assume a good deterministic approximation to the actual world dynamics. Generalizing to stochastic domains may require significant algorithm change in training and execution but the high-level concepts of RatSkills could be retained.

5 Conclusion

We present RatSkills, compositional hierarchical skill models that support efficienct planning and inverse planning to achieve novel goals and recognize activities of other agents. RatSkills can be learned by observing expert demonstrations and reading task specifications described in a simple task language $\mathcal{TL}$. Specifically, we present a learning algorithm based on Bayesian inverse planning to simultaneously segment the trajectory into fragments, which correspond to individual skills, and learn planning-compatible models for each skill. Our experiments in both discrete and continuous domains suggest that RatSkills have strong compositional generalization to novel tasks composed of the previously-seen skills.

REPRODUCIBILITY STATEMENT

Our dataset, environment, and code can be found at: https://sites.google.com/view/ratskills/home. Our dataset and code contain no personally indentifiable information or offensive content.

REFERENCES

Jacob Andreas and Dan Klein. Alignment-based compositional semantics for instruction following. In *EMNLP*, 2015. 2

Jacob Andreas, Dan Klein, and Sergey Levine. Modular multitask reinforcement learning with policy sketches. In *ICML*, 2017. 2

Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Rob Auton Syst.*, 57(5):469–483, 2009. 3

Chris L Baker, Rebecca Saxe, and Joshua B Tenenbaum. Action understanding as inverse planning. *Cognition*, 113(3):329–349, 2009. 3

Calin Belta. Formal synthesis of control strategies for dynamical systems. In *CDC*, 2016. 4

Matthew Botvinick and Ari Weinstein. Model-based hierarchical reinforcement learning and human action control. *Philos. Trans. R. Soc. Lond., B, Biol. Sci.*, 369(1655):20130480, 2014. 3

Christopher Bradley, Adam Pacheck, Gregory J. Stein, Sebastian Castro, Hadas Kress-Gazit, and Nicholas Roy. Learning and planning for temporally extended tasks in unknown environments, 2021. 2

Matthew Brand, Nuria Oliver, and Alex Pentland. Coupled hidden markov models for complex action recognition. In *CVPR*, 1997. 3

Valerie Chen, Abhinav Gupta, and Kenneth Marino. Ask your humans: Using human instructions to improve generalization in reinforcement learning. In *ICLR*, 2021. 2, 7, 15

Sonia Chernova and Manuela Veloso. Confidence-based policy learning from demonstration using gaussian mixture models. In *AAMAS*, 2007. 3

Rodolfo Corona, Daniel Fried, Coline Devin, Dan Klein, and Trevor Darrell. Modular networks for compositional instruction following. In *NAACL-HLT*, pp. 1033–1040, 2021. 2

Tomás de la Rosa and Sheila McIlraith. Learning domain control knowledge for tlplan and beyond. In *ICAPS 2011 Workshop on Planning and Learning*, 2011. 2, 3

Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *JAIR*, 13:227–303, 2000. 3

Malcolm Doering. *Verb semantics as denoting change of state in the physical world*. Michigan State University, 2015. 12

Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li, and Denny Zhou. Neural logic machines. In *ICLR*, 2019. 3, 16

Staffan Ekvall and Danica Kragic. Robot learning from demonstration: a task-level planning approach. *IJARS*, 5 (3):33, 2008. 3

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997. 19

De-An Huang, Li Fei-Fei, and Juan Carlos Niebles. Connectionist temporal modeling for weakly supervised action labeling. In *European Conference on Computer Vision*, pp. 137–153. Springer, 2016. 3

George Konidaris, Scott Kuindersma, Roderic Grupen, and Andrew Barto. Robot learning from demonstration by constructing skill trees. *IJRR*, 31(3):360–375, 2012. 3

George Konidaris, Leslie Pack Kaelbling, and Tomas Lozano-Perez. From skills to symbols: Learning symbolic representations for abstract high-level planning. *JAIR*, 61:215–289, 2018. 2, 7, 8, 9, 15, 16

Peter Ondruska Markus Wulfmeier and Ingmar Posner. Maximum entropy deep inverse reinforcement learning. In *NeurIPS Workshop*, 2015. 3

Neville Mehta. *Hierarchical structure discovery and transfer in sequential decision problems*. Oregon State University, 2011. 3

Muhammad Abdul Hakim Newton, John Levine, Maria Fox, and Derek Long. Learning macro-actions for arbitrary planners and domains. In *ICAPS*, 2007. 2

Scott Niekum, Sarah Osentoski, George Konidaris, and Andrew G Barto. Learning and generalization of complex tasks from unstructured demonstrations. In *IROS*. IEEE, 2012. 3

Daehyung Park, Michael Noseworthy, Rohan Paul, Subhro Roy, and Nicholas Roy. Inferring task goals and constraints using bayesian nonparametric inverse reinforcement learning. In *JMLR*, 2020. 2

Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. In *NeurIPS*, 1998. 4

Harish Ravichandar, Athanasios S Polydoros, Sonia Chernova, and Aude Billard. Recent advances in robot learning from demonstration. *Annu Rev Control.*, 3:297–330, 2020. 3

Dorsa Sadigh, Eric S Kim, Samuel Coogan, S Shankar Sastry, and Sanjit A Seshia. A learning based approach to control synthesis of markov decision processes for linear temporal logic specifications. In *CDC*. IEEE, 2014. 2

Javier Segovia-Aguas, Jonathan Ferrer-Mestres, and Anders Jonsson. Planning with partially specified behaviors. In *Artificial Intelligence Research and Development*, pp. 263–272. IOS Press, 2016. 3

Ankit Shah, Pritish Kamath, Julie A Shah, and Shen Li. Bayesian inference of temporal task specifications from demonstrations. In *NeurIPS*, 2018. 3

Shao-Hua Sun, Te-Lin Wu, and Joseph J Lim. Program guided agent. In *ICLR*, 2020. 2

Stefanie Tellex, Thomas Kollar, Steven Dickerson, Matthew Walter, Ashis Banerjee, Seth Teller, and Nicholas Roy. Understanding natural language commands for robotic navigation and mobile manipulation. In *AAAI*, 2011. 2

Faraz Torabi, Garrett Warnell, and Peter Stone. Behavioral cloning from observation. In *IJCAI*, 2018. 3, 7

Rodrigo Toro Icarte, Toryn Q. Klassen, Richard Valenzano, and Sheila A. McIlraith. Teaching multiple tasks to an rl agent using ltl. In *AAMAS*, 2018. 2

John Winder, Stephanie Milani, Matthew Landen, Erebus Oh, Shane Parr, Shawn Squire, Cynthia Matuszek, et al. Planning with abstract learned models while learning transferable subtasks. In *AAAI*, 2020. 3

Tan Zhi-Xuan, Jordyn L Mann, Tom Silver, Joshua B Tenenbaum, and Vikash K Mansinghka. Online bayesian goal inference for boundedly-rational planning agents. In *NeurIPS*, 2020. 3

Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, and Anind K Dey. Maximum entropy inverse reinforcement learning. In *AAAI*, 2008. 3, 7

A IMPLEMENTATION DETAILS OF RATSKILLS

A.1 SKILL MODEL AND TASK LANGUAGE

In this section, we first formally describe how we can construct finite state machines based on task descriptions. Next, we provide a discussion on the initiation set I_o in our skill model. Then, we provide a example to better understand the semantics of "then" in our task language. Specifically, how it allows the execution steps for different skills to be interleaved.

Construction of finite state machines from task descriptions. We provide the detailed algorithm for the construction. Recall the definition of FSM. Each FSM_t is a tuple (V_t, E_t, VI_t, VG_t) which are skill nodes, edges, set of possible starting nodes and set of terminal nodes. We have the follow constructions:

- **Single skill:** A single skill s is corresponding FSM with a single node i.e. $VI_t = VG_t = V_t = \{s\}$, and $E_t = \emptyset$.
- t_1 **then** t_2: We merge FSM_{t_1} and FSM_{t_2} by merging their skill nodes, edges and using VI_{t_1} as the new starting node set and VG_{t_2} as the new terminal node set. Then, we add all edges from VG_{t_1} to VI_{t_2}. Formally,

$$\text{FSM}_{t_1 \textbf{ then } t_2} = (V_{t_1} \cup V_{t_2}, E_{t_1} \cup E_{t_2} \cup (VG_{t_1} \times VI_{t_2}), VI_{t_1}, VG_{t_2})$$

- t_1 **or** $\cdots$ **or** t_n: Simply merge n FSMs without adding any new edges. Formally,

$$\text{FSM}_{t_1 \textbf{ or } \cdots \textbf{ or } t_n} = (\bigcup_i V_{t_i}, \bigcup_i E_{t_i}, \bigcup_i VI_{t_i}, \bigcup_i VG_{t_i})$$

- t_1 **and** $\cdots$ **and** t_n: Build $2^{n-1}n$ sub-FSMs over n layers: the i-th layer contains $n \cdot \binom{n-1}{i-1}$ sub-FSMs each labeled by (s, D) where s is the current skill to complete (so this sub-FSM is a copy of FSM_s), and D is the set of skills that have been previously completed. Then for a sub-FSM (s_1, D_1) and a sub-FSM (s_2, D_2) in the next layer, if $D_2 = D_1 \cup \{s_1\}$, we add all edges from terminal nodes of the first sub-FSM to starting nodes of the second sub-FSM. After building layers of sub-FSMs and connecting them, we set the starting nodes to be the union of starting nodes in the first layer and terminal nodes to be the union of terminal nodes in the last layer.

Note that our framework requires the starting and terminal nodes to be unique, but the construction above may output a FSM with multiple starting/terminal nodes, so we introduce the virual super starting node v_0 and terminal node v_T to unify them.

The initial condition I_o is important in our abstration. The learned skill definitions are intended to constitute a hierarchy that is useful both for parsing human activity sequences and efficiently generating behavior. To do so, high-level skills need to account for a bounded interval of activity. Without an explicit understanding of an initiation set i.e. I_o, an entire long behavior sequence could conceivably be described just in terms of the final goal condition. For example, picking up an object should be interpreted as from "not holding X" to "holding X", but not as a long sequence of building X from scratch and then picking up it.

This interpretation agrees with the definition of a broad set of natural language words, such as the representations in Doering (2015).

In planning, the initial condition associated with each skill helps limit the horizon when planning for the goal. For example, our planning algorithm will not try to plan for "holding an axe" (the subgoal for skill "pickup-axe") if we haven't yet built an axe (because the initial condition for "pickup-axe" – having an axe on the ground – is not satisfied).

Although it is true that the simple planning method we used as a proof-of-concept does not make use of the initiation sets, more sophisticated planning methods make critical use of preconditions and effects, especially for deriving domain-independent value functions.

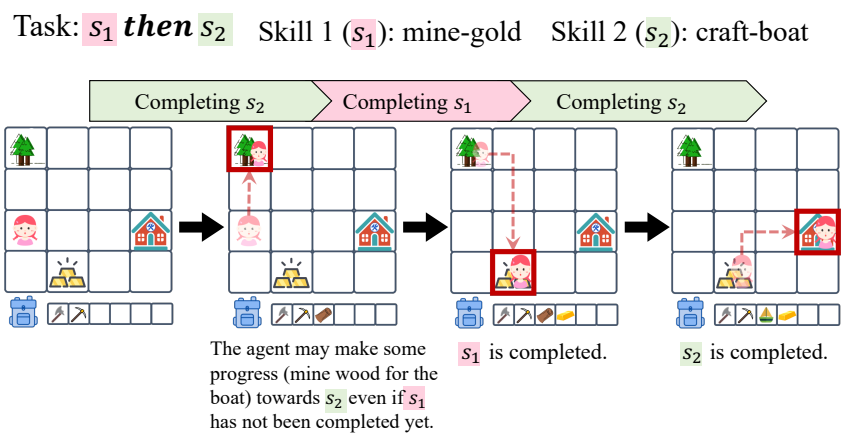

Figure 7: An example of optimal interleaving skills: s_1 is "mine gold", and s_2 is "craft boat". It is valid that the agent first goes to collect wood (for accompolishing s_2), and then mine gold (for accompolishing s_1), and finally crafts boat. In this case, the action sequences for completing s_1 and s_2 are interleaved. However, they can are be recognized as s_2 **then** s_2 because s_1 is accomplished before s_2.

Execution steps for different skills can interleave. RatSkills does not simply run optimal policy for each individual skill sequentially. Rather, the semantic of s_1 **then** s_2 is: s_1 should be completed before s_2. It does not restrict the agent from making progress towards the skill before the skill is completed. In some case, such interleaving is necessary to obtain the globally optimal trajectory.

Consider the example shown in Figure 7, where s_1 is "mine-gold", and s_2 is "craft-boat". It is valid that the agent first goes to collect wood (for accompolishing s_2), and then mine gold (for accompolishing s_1), and finally crafts boat. In this case, the action sequences for completing s_1 and s_2 are interleaved. However, they can are be recognized as s_1 **then** s_2 because s_1 is accomplished before s_2.

A.2 FSM-A^*

We have implemented a extended version of the A^* algorithm to handle FSM states in Crafting World.

A^* at each FSM node. We start with the A^* search process happening at each FSM node. For a given FSM state, the A^* search extends the tree search in two stages. The first stage lasts for $b = 3$ layers during training and $b = 4$ layers during testing. In the first b layers of the search tree, we run a Breadth-First-Search so that every possible path with length b is explored. Then on the second stage lasts for $c = 15$ layers during training and 25 layers in testing. In layer $d \in [b + 1, b + c]$, we run A* from the leaves in the first stage based on the heuristic for each node. By enforcing the exploration at the early stage, we avoid imperfect heuristic from misguiding the A* search at the beginning. For each FSM node v and each layer d, we only keep the top $k = 10$. Finally, we run the value iteration on the entire search tree.

To accelerate this search process, for all tasks t in the training set, we have initialized a dedicated value approximator $V_t(\bar{s})$, conditioned on the historical state sequence. During training, we use the value iteration result on the generated search tree to supervise the learning of this approximator V_t. Meanwhile, we use the value prediction of V_t as the heuristic function for node pruning. During test, since we may encounter unseen tasks, the A^*-FSM search uses a uniform heuristic function $h \equiv 0$

Search on an FSM. For a given initial state s_0 and task description t, we first build FSM_t and add the search tree node (s_0, v_0) to the search tree, where v_0 is the initial node of the FSM. Then we expand the search tree nodes (s, v) by a topological order of v. It has two stages. First, for each FSM node v, we run up to 5000 steps of A^* search. Next, for all search tree nodes (s, v) at FSM node v, we try to make a transition from (s, v) to (s, v') where (v, v') is a valid edge in FSM_t. Finally, we output a trajectory ending at the FSM node v_T with minimum cost.

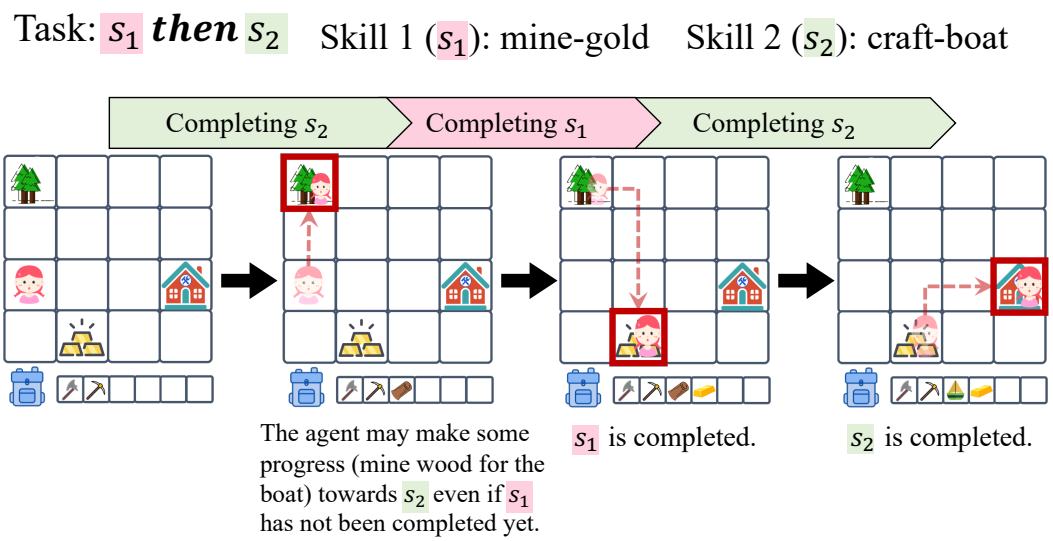

Figure 8: A running example of the FSM-A^* algorithm for the task "*(mine wood* ***or*** *mine coal)* ***then*** *mine gold*." For simplicity, we only show a subset of states visited on each FSM node. The blue arrows indicate transitions by primitive actions (in this example, each primitive action takes a cost of 0.1). The yellow arrows are transitions on the FSM, which can only be performed when $G_v(\cdot)$ and $I_{v'}(\cdot)$ evaluates to True (in practice, the reward is computed as $-(\log \Pr(G_v(\cdot) + \log \Pr(I_{v'}(\cdot))))$. At the super-terminal node v_T, the state with minimum cost will be selected and we will back-trace the entire state-action sequence.

Example. Fig. 8 shows a running example of our FSM-A^* planning given the task "*mine wood* ***or*** *mine coal* ***then*** *mine gold*" from the state s_0 (shown as the left-most state in the figure).

1. At the beginning, (s_0, v_0) is expanded to the node v_1:*mine wood* and v_2:*mine coal* with FSM transition actions at no cost.
2. We expand the search tree node on v_1 and v_2 and compute the cost for reaching each states on v_1 and v_2.
3. For states that satisfy the goal conditions for v_1 and v_2 (i.e., G_1 and G_2, respectively, and circled by green and blue boxes) and the initial condition for v_3 (i.e., I_3), we make a transition to v_3 at no cost (the states that do not satisfy the conditions can also be expanded to v_3 but with a large cost.
4. Then search can be done in a similar way at v_3 and the states at v_3 that satisfy G_3 can reach v_T.
5. For all states at v_T, we back-trace the state sequence with the minimum cost.

Optimality of the A* algorithm on FSM. In the current implementation, RatSkills might return sub-optimal solutions even with a perfect heuristic, because RatSkills balance the expanded nodes across all FSM nodes: it first samples an FSM node and then expand a search tree node with the best heuristic value on that node.

The optimality can be guaranteed by either of the following simple modifications, although at the cost of possibly increasing the running time:

- Always expand the search node with the globally best admissible heuristic value. (Because our heuristic is learned, this may not be practical.)
- Keep expanding nodes, even after finding a plan, until none of the unexpanded search tree nodes across all skill nodes in the FSM have better heuristic values than the current best solution.

Scalability and complexity of task searching. When planning for a black-box test, currently we assume that the task can be completed within a short sequence of skills, and thus a blind/enumerative search at the skill level can work. In the future we hope to incorporate other learning and planning-based approach (Konidaris et al., 2018) at the skill level to improve efficiency.

Meanwhile, the efficiency can be justified theoretically, even for this enumerative search approach. Say we have a skill set $\mathcal{O}$, a primitive action set $\mathcal{A}$, and each skill can be completed in l actions, and the task can be achieved by sequencing m skills. Our two-level search generates up to $O(|\mathcal{O}|^m)$ candidate skill sequences and each sequence takes $O(m|\mathcal{A}|^l)$ time to search. Thus, the worst-case complexity is $O(m|\mathcal{O}|^m|\mathcal{A}|^l)$ which is still better than a pure primitive-level search, which has the worst-case complexity $O(|\mathcal{A}|^{ml})$, because the number of skills $|\mathcal{O}|$ is usually much smaller than $|\mathcal{A}|^l$ (the number of all possible length l sequences).

A.3 TRANSITIONS ON FSM

When encoding transitions on FSM, we use dynamic program to select a transition that maximize our $score$, and another formulation is to consider the stochastic transitions using I_v and G_v as probabilities. These two formulations can be unified using a framework of latent transition models, though they are computed using different DP algorithms and may lead to different results.

First of all, these two formulations are equivalent when the initial/goal classifiers are binary (0/1). When the classifiers are approximated by "soft" functions that indicate the probability they are satisfied, the two formulations correspond to two approaches of integrating reward (i.e. rationality in our model). The stochastic transition formulation computes the expected rationality, and our formulation can be viewed as an approximation of maximum-likelihood estimated rationality – we take $\max_\tau \lambda \log \Pr(\text{transitions in } \tau \text{ are successful}) + Rationality(\bar{s}, \bar{a}, \tau)$. It would be an interesting extension to adopt the stochastic transition formulation (i.e., $\mathbb{E}_\tau \lambda \log \Pr(\text{transitions in } \tau \text{ are successful}) + Rationality(\bar{s}, \bar{a}, \tau)$) and use a stochastic planner or MDP solver, although the planning time might be substantially increased.

Second, even if these two approaches behave differently in some cases, but it is unclear which one is better: this is a fundamental challenge in planning: how should the robot decide whether it has finished a task if there is no indication (such as rewards) from the environment?

B DATASET

B.1 CRAFTING WORLD

Our Crafting World environment is based on the Crafting environment introduced by Chen et al. Chen et al. (2021). The environment has a crafting agent that can move in a grid world, collect resources, and craft items. In the original environment, every crafting rule is associated with a unique crafting station (e.g., paper must be crafted on the paper station). We modified the rules such that some crafting rules can share a common crafting station (e.g., both arrows and swords can be crafted on a weapon station). We add additional tiles: doors and rivers into the environment. Toggling a specific switch will open all doors. Otherwise, the agent can move across doors when they are holding a key. Meanwhile, the agent can move across rivers when they have a boat in their inventory.

We have used 47 object types in Crafting World including obstacles (e.g., river tiles, doors), items (e.g., axe), resources (e.g., trees), and crafting stations. We use 27 rules for mining resources and crafting items. When the agent is at the same tile as another object, the *toggle* action will trigger the object-specific interaction. For item, the *toggle* action will pick up the object. For resource, the *toggle* action will mine the resource if the agent has space in their inventory and has the required tool for mining this type of resource (e.g., pickaxe is needed for mining iron ore).

State representation. The state representation of Crafting World consists of three parts.

1. The *global feature* contains the size of grid world, the location of the agent, and the inventory size of the agent.
2. The *inventory* feature contains an unordered list of objects in the agent's inventory. Each of them is represented as a one-hot vector indicating its object type.

3. The *map* feature contains all objects on the map, including obstacles, items, resources, and crafting stations. Each of them is represented by a one-hot type encoding, the location (as integer values), and state (e.g., *open* or *closed* for doors).

Action. In Crafting World, there are 5 primitive level actions: *up*, *down*, *left*, *right*, and *toggle*. The first four actions will move the agent in the environment, while the *toggle* action will try to interact with the object in the same cell as the agent.

State feature extractor. Since our state representation contain a varying number of objects, we extract a vector representation of the environment with a relational neural network: Neural Logic Machines (NLM; Dong et al., 2019).

Concretely, we extract the inventory feature and the map feature separately. For each item in the inventory, we concatenate its input representation (i.e., the object type) with the global input feature. We process each item with the same fully-connected layer with ReLU activation. Following NLM Dong et al. (2019), we use a max pooling operation to aggregate the feature for all inventory objects, resulting in a 128-dim vector. We use a similar architecture (but different neural network weights) to process all objects on the map. Finally, we concatenate the extracted inventory feature (128-dim), the map feature (128-dim), and the global feature (4-dim) as the holistic state representation. Thus, the output feature dimension for each state is 260.

Task definitions. We list the task descriptions in the *primitive*, the *compositional*, and the *novel* splits Table 3.

B.2 PLAYROOM

We build our Playroom environment following Konidaris et al. Konidaris et al. (2018). Specifically, we have added obstacles into the environment. The environment contains an agent, 6 effectors (a ball, a bell, a light switch, a button to turn on the music, a button to turn off the music and a monkey), and a fix number of obstacles. The agent and the effectors have fixed shapes. Thus, their geometry can be fully specified by their location and orientation. For simplicity, we have also fixed the shape and the location of the obstacles.

State representation. We represent the pose of the agent by a 3D vector including the x, y coordinates (real-valued) and its rotation (real-valued, in $[-\pi, \pi)$. The state representation consist of the pose of the agent (as a 3-dimensional vector) and the locations of six effectors (as 6 2-dimensional vectors). Note that the state representation does not contain the shapes nor the locations of obstacles as they remain unchanged throughout the experiment. We directly concatenate these 7 vectors as the state representation.

Action. The agent has a 3-dimensional action space: $[-1, 1]^3$. That is, for example, at each time step, the agent can at most move 1 meter along the x axis. We perform collision checking when the agent is trying to make a movement. If an action will result in a collision with objects or obstacles in the environment, the action will be treated as invalid and the state of the agent will not change.

Task definitions. We list the task descriptions in each of the *primitive*, *compositional* and *novel* set of the Playroom in Table 4

Primitive			
grab-pickaxe	grab-axe	grab-key	toggle-switch
craft-wood-plank	craft-stick	craft-shears	craft-bed
craft-boat	craft-sword	craft-arrow	craft-cooked-potato
craft-iron-ingot	craft-gold-ingot	craft-bowl	craft-beetroot-soup
craft-paper	mine-gold-ore	mine-iron-ore	mine-sugar-cane
mine-coal	mine-wood	mine-feather	mine-wool
mine-potato	mine-beetroot		

Compositional	
grab-pickaxe	grab-axe
grab-key	toggle-switch
mine-wood **then** craft-wood-plank	craft-wood-plank **then** craft-stick
craft-iron-ingot **or** craft-gold-ingot **then** craft-shears	mine-wool **and** craft-wood-plank **then** craft-bed
craft-wood-plank **then** craft-boat	craft-iron-ingot **and** craft-stick **then** craft-sword
mine-feather **and** craft-stick **then** craft-arrow	mine-potato **and** mine-coal **then** craft-cooked-potato
mine-iron-ore **and** mine-coal **then** craft-iron-ingot	mine-gold-ore **and** mine-coal **then** craft-gold-ingot
craft-wood-plank **or** craft-iron-ingot **then** craft-bowl	craft-bowl **and** mine-beetroot **then** craft-beetroot-soup
mine-sugar-cane **then** craft-paper	grab-pickaxe **then** mine-gold-ore
grab-pickaxe **then** mine-iron-ore	grab-pickaxe **or** grab-axe **then** mine-sugar-cane
grab-pickaxe **then** mine-coal	grab-axe **then** mine-wood
craft-sword **then** mine-feather	craft-shears **or** craft-sword **then** mine-wool
grab-axe **or** mine-coal **then** mine-potato	grab-axe **or** grab-pickaxe **then** mine-beetroot

Novel
1. mine-sugar-cane **then** craft-paper
2. mine-potato **and** (gran pickaxe then mine-coal) **and** craft-cooked-potato
3. mine-beetroot **and** (grab-axe then mine-wood then craft-wood-plank then craft-bowl) **then** craft-beetroot-soup
4. grab-axe **then** mine-wood **then** craft-wood-plank **then** grab-pickaxe **then** mine-iron-ore **and** mine-coal **then** craft-iron-ingot **then** craft-shears **then** mine-wool **then** craft-bed
5. grab-axe **then** mine-wood **then** craft-wood-plank **then** craft-stick **then** grab-pickaxe **then** mine-iron-ore **and** mine-coal **then** craft-iron-ingot **then** craft-sword **then** mine-feather **then** mine-wood **then** craft-wood-plank **then** craft-stick **then** craft-arrow
6. grab-key **then** grab-axe
7. toggle-switch **then** mine-beetroot
8. grab-axe **then** mine-wood **then** craft-wood-plank **then** craft-boat **then** mine-sugar-cane
9. grab-axe **then** mine-wood **then** craft-wood-plank **then** craft-boat **then** grab-pickaxe
10. grab-key **then** grab-axe **then** mine-wood **then** craft-wood-plank **then** craft-boat **then** mine-potato
11. grab-key **or** (grab-axe **then** mine-wood **then** craft-wood-plank **then** craft-boat) **then** grab-pickaxe **then** mine-gold-ore
12. grab-axe **then** mine-wood **then** craft-wood-plank **then** craft-boat **then** grab-key **or** toggle-switch **then** grab-pickaxe **then** mine-iron-ore **and** mine-coal **then** craft-iron-ingot

Table 3: Task descriptions in the *primitive*, *compositional* and *novel* sets for the Crafting World.

Primitive		
play-with-ball	ring-bell	turn-on-light
touch the mounkey	turn-off-music	turn-on-music
Compositional (designed meaningful tasks)		
play-with-ball		
turn-on-light **then** ring-bell		
turn-on-music **and** play-with-ball **then** touch the monkey		
play-with-ball **then** turn-on-light		
turn-on-music **and** play-with-ball **then** turn-off-music		
turn-on-music **or** play-with-ball		
turn-off-music **then** play-with-ball **then** turn-on-music		
turn-on-music **and** play-with-ball **and** turn-on-light **then** ring-bell		
Novel (randomly sampled)		
play-with-ball **then** turn-on-light **or** ring-bell		
turn-on-music **then** turn-on-light		
turn-on-music **then** turn-on-light		
play-with-ball **then** touch the monkey		
turn-on-music **then** turn-off-music		
turn-on-music **and** ring-bell **then** touch the monkey		
ring-bell **then** touch the monkey **then** turn-on-light		
turn-on-light **and** (ring-bell **or** turn-on-music) **then** play-with-ball		

Table 4: Task descriptions in the *primitive*, *compositional* and *novel* sets for the Playroom.

C IMPLEMENTATION DETAILS

In this section, we present the implementation details of RatSkills and other baselines. Without further notes, through out this section, we will be using the same LSTM encoder for task descriptions in $\mathcal{TL}$, and the same LSTM encoder for state sequences. The architecture of both encoders will be presented in Appendix C.1.

C.1 LSTM

Task description encoder. We use a bi-directional LSTM Hochreiter & Schmidhuber (1997) with a hidden dimension of 128 to encode the task description. The vocabulary contains all primitive skills, parentheses, and three connectives (*and*, *or*, and *then*). We perform an average pooling on the encoded feature for both directions, and concatenate them as the encoding for the task description. Thus, the output dimension is 256.

State sequence encoder. For a given state sequence $\bar{s} = \{s_i\}$, we first use a fully-connected layer to map each state s_i into a 128-dimensional vector. Next, we feed the sequence into a bi-directional LSTM module. The hidden dimension of the LSTM is 128. We perform an average pooling on the encoded feature for both directions, and concatenate them as the encoding for the state sequence.

Training. In our LSTM baseline for inverse planning, we concatenate the state sequence feature and the task description feature, and use a 2-layer multi-layer perceptron (MLP) to compute the score of the input tuple: (trajectory, task description). The LSTM model is trained for 100 epochs on both environments. Each epoch contains 30 training batches that are randomly sampled from training data. The batch size is 32. We use the RMSProp optimizer with a learning rate decay from 10^{-3} to 10^{-5}.

C.2 INVERSE REINFORCEMENT LEARNING (IRL)

The IRL baseline uses an LSTM model to encode task descriptions. We use different parameterizations for the reward function and the Q function in two datasets.

Crafting World Since the task description may have complex temporal structures, the reward value does not only condition on the current state and but all historical states. Therefore, instead of $Q(s, a|t)$ and $R(s, a, s'|t)$, we use $Q(\bar{s}, a|t)$ and $R(\bar{s}, a, s'|t)$ to parameterize the Q function and reward function, where s is the current state, a the action, t the task description, s' the next state, and $\bar{s}$ the historical state sequence from the initial state to the current state.

We use neural networks to approximate the Q function and reward function. For both of them, $\bar{s}$ is first encoded by an LSTM model into a fixed-length vector embedding. We simply concatenate the historical state encoding and the task description encoding, and then use a fully-connected layer to map the feature into a 5-dimensional vector. Each entry corresponds to the Q value or the reward value for a primitive action.

Playroom The Q function and reward function in Playroom also condition on all historical states. In Playroom, we parameterize the value of each state: $V(\bar{s})$, instead of $Q(\bar{s}, a)$. We parameterize $R(\bar{s}, a, s')$ as $R(\bar{s}, s')$.

The input to our reward function network is composed of three parts: the vector encoding of the historical state sequence, the vector encoding for the next state s', and the task description encoding. We concatenate all three vectors and run a fully-connected layer with a logSigmoid activation function.[‡]

In Playroom, since we do not directly parameterize the Q value for all actions in the continuous action space, in order to sample the best action at each state s for plan execution, we first randomly sample 20 valid actions from the action space (i.e., actions that do not lead to collision), and choose the action that maximizes the Q function: $Q(\bar{s} \cup s', a)$, where $\bar{s}$ is the historical state seuqnce and s' is the next state after taking a.

[‡]We have experimented with no activation function, Sigmoid, and logSigmoid activations, and found that the logSigmoid activation works the best.

Value iteration. Both environments have a very large (Crafting World) or even infinite (Playroom) state space. Thus it is impossible to run value iteration on the entire state space. Thus, at each iteration, for a given demonstration trajectory $(\bar{s}_e, \bar{a}_e)$, we construct a self exploration trajectory $(\bar{s}_p, \bar{a}_p)$ that share the same start state as $\bar{s}_e$ [§]. We run value iteration on $\{\bar{s}_e\} \cup \{\bar{s}_p\}$. For states not in this set, we use the Q function network to approximate their values.

Training. For both Crafting World and Playroom, we train the IRL model for 60 epochs. We set the batch size to be 32 and each epoch has 30 training batches. We use a replay buffer that can store 100,000 trajectories. For both environments, we use the Adam optimizer with a learning rate decay from 10^{-3} to 10^{-5}. We have found the IRL method unstable to train in the Playroom environment. Thus, in Playroom, we use a warm-up training procedure. In the first 18(30%) epochs, we set $\gamma = 0$ for a "warm start", and for rest of the epochs we use $\gamma = 0.5$, where γ is the discount factor in the Q function.

C.3 Behavior Cloning (BC)

BC learns a policy $\pi(\bar{s}, a|t)$ from data, where t is the task description, a a primitive action, and $\bar{s}$ the historical state sequence. The state sequence $\bar{s}$ is first encoded by an LSTM model into a fixed-length vector embedding.

In Crafting World, we use a fully-connected layer with softmax activation to parameterize $\pi(a|\bar{s}, t)$. Specifically, the input to the fully-connected layer is the concatenation of the vector encoding of $\bar{s}$ and the vector encoding of the task description t.

In Playroom, we use two fully-connected (FC) layers to parameterize $\pi(a|\bar{s}, t)$. Specifically, we parameterize $\pi(a|\bar{s}, t)$ as a Gaussian distribution. The first FC layer has a Tanh activation and parameterizes the mean μ of the Gaussian. The second FC layer has a Sigmoid activation and paramerizes the standard variance σ^2 of the Gaussian.

To make this model more consistent with our BC-FSM model, in both environments, we also train a module to compute the termination condition of the trajectory. That is, a neural network that maps $\bar{s}$ to a real value in $[0, 1]$, indicating the probability of terminating the execution. Denote the output of this network as $\mathit{stop}(\bar{s})$. At each time step, the agent will terminate its policy with probability $\mathit{stop}(\bar{s})$. We modulate the probability for other actions a as $\pi(a|\bar{s}, t) \cdot (1 - \mathit{stop}(\bar{s}))$

For planning in Crafting World, at each step, we choose the action with the maximum probability (including the option to "terminate" the execution). In Playroom, we always take the "mean" action parameterized by $\pi(a|\bar{s}, t)$ until we reach the maximum allowed steps.

We then define the score of a task given a trajectory, $\mathit{score}(\bar{s}, \bar{a}, t)$, as the sum of log-probabilities of the actions taken at each step. We train this model with the same loss and training procedure as RatSkills. We train the model for 100 epochs using the Adam optimizer with a learning rate decay from 10^{-3} to 10^{-5}.

C.4 Behavior Cloning with FSM (BC-FSM)

BC-FSM represents task description as an FSM, in the same way as our model RatSkills. It represents each skill o as a tuple: $\langle \pi_o(sa), \mathit{stop}_o(s) \rangle$, corresponding to the skill-conditioned policy and the termination condition.

Inverse planning. The inverse planning procedure for BC-FSM jointly segments the trajectory and computes the consistency score between the task description and the input trajectory. In particular, our algorithm will assign an FSM state v_i to each state s_i, and insert several action. We use a dynamic programming algorithm (similar to the one used by our algorithm for RatSkills) to find the assignment that maximize the overall score:

$$\mathit{score}(\bar{s}, \bar{v}, \bar{a}) \coloneqq \prod_i p(a_i|s_i, v_i, t)$$

$$p(a_i|s_i, v_i, t) = \begin{cases} \pi(a_i|s_i, v_i) \cdot (1 - \mathit{stop}(s_i, v_i)) & \text{if } a \in \mathcal{A} \text{ is a primitive action} \\ \mathit{stop}(s_i, v_i) & \text{if } a \in E_t \text{ is an FSM transition} \end{cases}$$

[§]Since running self-exploration in Playroom is too slow, in practice, we only generate self-exploration trajectories for 4 trajectories in the input batch.

Planning. We use the same strategy as the basic Behavior Cloning model to choose actions at each step, conditioned on the current FSM state. BC-FSM handles branches in the FSM in the same way as our algorithm for RatSkills.