Bootstrapping Cognitive Agents with a Large Language Model

Feiyu Zhu, Reid Simmons

Carnegie Mellon University

feiyuz@andrew.cmu.edu, rsimmons@andrew.cmu.edu

Abstract

Large language models contain noisy general knowledge of the world, yet are hard to train or fine-tune. In contrast cognitive architectures have excellent interpretability and are flexible to update but require a lot of manual work to instantiate. In this work, we combine the best of both worlds: bootstrapping a cognitive-based model with the noisy knowledge encoded in large language models. Through an embodied agent doing kitchen tasks, we show that our proposed framework yields better efficiency compared to an agent entirely based on large language models. Our experiments also indicate that the cognitive agent bootstrapped using this framework can generalize to novel environments and be scaled to complex tasks.

Introduction

Large language models (LLM) such as GPT-4 (OpenAI 2023), have shown emerging capabilities after training on internet-scale text data with human feedback, and have been employed in robot planning (Huang et al. 2022), animal behavior analysis (Ye et al. 2023), human proxies (Zhang and Soh 2023), and many more. However, they have also been criticized for being susceptible to adversarial attacks (Zou et al. 2023), hallucination (Casper et al. 2023), and having diminishing returns for scaling (OpenAI 2023).

Cognitive architectures are another approach in the pursuit of AI that attempts to model human cognition computationally (Newell 1994). Despite the variety of architectures developed, most of them share the same central components, consisting of declarative memory reflecting knowledge of the world, procedural memory dictating the agent's behavior, and short-term working memory that assists reasoning and planning (Laird, Lebiere, and Rosenbloom 2017).

The procedural memory is represented by a set of production rules, each with a precondition and an effect. Agents operate in perceive-plan-act cycles, dynamically matching relevant features of the environment to the production rules and applying their effects. Unlike operators in symbolic planning, production rules do not represent alternative actions but instead reflect different contextual knowledge (Laird 2022). These rules can be reinforced and modified throughout the agent's learning process. Despite some pioneering work on data-driven cognitive model creation (Hake, Sibert, and Stocco 2022), almost all previous work generate their initial set of production rules manually, limiting their application to simple environments such as blocks world or psychology experiments (Park et al. 2023).

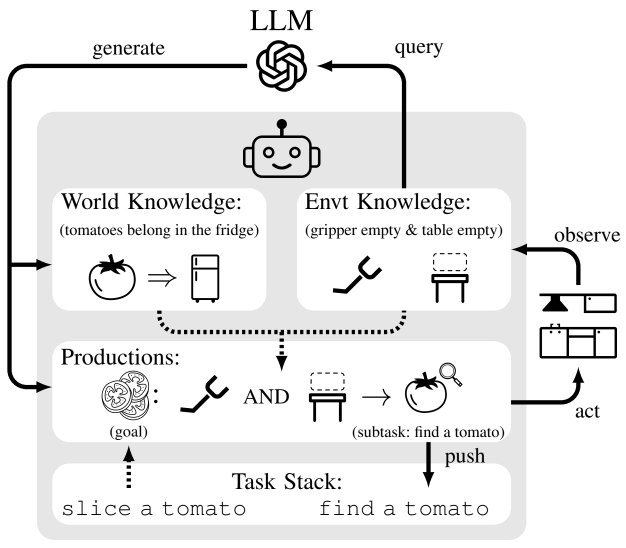

Figure 1: Overview of agent framework. It shows the agent executing the production of attending to a new subtask of finding a tomato when the original task is to slice a tomato and the tomato is not in the gripper nor on the table. Dotted lines represent the information a production rule may condition on. Solid lines represent information flow.

In this work, we combine the two approaches in a complementary fashion (Figure 1). LLMs encode the common sense knowledge of the world (Madaan et al. 2022) that can be used in place of human labor for constructing agents in the cognitive architecture. The reasoning and learning capabilities in the cognitive architecture can identify and filter the noise in LLMs while converting the knowledge in language to actionable productions of an embodied agent.

This combined framework separates knowledge generation and knowledge application, and this modularity is the key to generalization. The LLM is responsible only for generating general knowledge, such as "if the task is to find an object, the agent should explore the places where that object is commonly stored". Since such knowledge can be applied to almost all objects and environments, the LLM needs to generate these only once, and it is the role of the cognitive architecture to dynamically match the environment to the generated knowledge. This is significantly different from using LLMs to generate plans directly, as the plans are grounded to the specific instance of the task (e.g., finding a specific object in the specific environment), and are non-trivial to generalize to novel environments without re-generation.

The contribution of this paper is threefold: 1) we propose an agent framework that combines LLMs with customized cognitive architecture, 2) we demonstrate how it can learn to perform various kitchen tasks from bootstrapping, and 3) we show that, when applied to new environments, it requires significantly fewer tokens than querying LLM for actions. [1]

Related Work

Learning Through Program Synthesis

Interactive Task Learning (ITL) (Laird et al. 2017) aims at teaching robots new skills in a one-shot fashion. Previous work implements this in the SOAR cognitive architecture and has shown effective task and environment transferability in domains such as board games (Kirk and Laird 2019) and embodied agents (Mininger and Laird 2022). To reduce the need for extensive human input, recent research explores using LLM as the knowledge source (Lindes and Peter 2023; Kirk et al. 2023), shifting human labor from specifying the goal conditions to answering yes/no questions. In contrast, our approach uses strategic prompting and self-reflection mechanisms to eliminate the need for human supervision.

Our work shares some high-level ideas with DreamCoder (Ellis et al. 2021), which learns to solve new problems by program generation and reflection. Instead of formulating it as an informed search problem, we accelerate this process by querying LLMs for their existing knowledge.

Madaan et al. (2022) extract common-sense knowledge from LLMs into code form similar to how we extract productions. But they only address the general task decomposition, not applying the information to an embodied agent.

Large Language Model for Embodied Agents

Many studies have explored using LLMs to generate code that performs robotics tasks (Liang et al. 2023; Singh et al. 2023; Vemprala et al. 2023) and game environments (Wang et al. 2023), which is similar to the procedural memory in the cognitive architectures. Other works explored generating PDDL specifications (Liu et al. 2023a; Xie et al. 2023). Unlike the situation-grounded code produced by these methods, our approach generates abstract productions with learnable weights. This allows more generalization capabilities and choosing the best plan among multiple applicable plans.

Others let LLMs select the action directly (Di Palo et al. 2023; Vemprala et al. 2023) with the help of other auxiliary components such as affordance evaluation (Ahn et al. 2022), memory stream (Park et al. 2023), visual summarization (Qiu et al. 2023), and knowledge base (Zhu et al. 2023). Some others explored multi-modal foundation models tailored for embodied agents (Driess et al. 2023; Xiang et al. 2023). As LLMs are non-trivial to update from a single instance, using more explicit production systems in our approach enables persistent one-shot updates and more interpretability. As we will show in our experiments, relying on LLMs for every action is also not very cost-effective.

Method

Architecture Overview

Figure 1 illustrates the architecture and workflow of the agent. The agent has four main components. A *world knowledge base* that contains general knowledge, such as "Tomatoes are commonly stored in the Fridge". *Environment knowledge* that reflects what the agent knows about the environment from past observations, including both information about the agent itself (e.g., the gripper is empty) and about the external world (e.g., the table is clear). These two components form the declarative memories of the agent.

Another essential component is the *procedural memory* that contains all the production rules. In our work, however, we integrate the working memory into each production by exploiting the Python class structure, so there is no centralized working memory. And, finally, inspired by the goal module of ACT-R (Anderson 2009) and the impasse mechanism of SOAR (Laird 2022), the agent manages a *task stack*.

At each time step, the agent searches in its procedural memory for any applicable production rule, considering the current task and environment knowledge. If there is no production applicable, the agent will summarize the current knowledge and query the LLM for both an action suggestion and a corresponding production rule, such that the agent knows what to do in similar scenarios in the future. When at least one production is applicable, it will sample an applicable production rule, based on its utility, and execute the proposed action, which can be either in the environment or internally, such as adding a subtask to its task stack.

Bootstrapping Procedures

The bootstrapping process starts with a *curriculum*. We took inspiration from (Wang et al. 2023), which uses an LLM to automatically construct the curriculum for Minecraft. As the simulator we use is not as popular as Minecraft and has some specific constraints (e.g., can only hold one object at a time), we find it better to specify the curriculum manually. Unlike previous work that requires human input on the next steps and/or goal condition for the tasks (Mininger and Laird 2022), we require only the names of the task families, so designing the curriculum is not very labor intensive.

Another difference is that our curriculum consists of families of tasks (e.g., `find a/an <object>`) instead of specific instances (e.g., `find a/an egg`). We follow the SOAR syntax and keep all variables in angle brackets.

With a given curriculum, the following steps are used to bootstrap a single task in the curriculum (using `find a/an <object>` as an example).

[1]Code at github.com/zfy0314/cognitive-agents

1. Fill in the variables randomly from the environment to instantiate a concrete task (e.g., `find a/an Egg`);
2. Attempt the task with the existing production rules;
3. (**Action Selection**) If there is no production rule for a state, or there is a cycle detected through the production application, query an LLM for an action;
4. (**Production Generation**) Generate the corresponding production rule to the action, and load it into the agent;
5. Repeat steps 1-4 sufficient times until the robot can perform the task with only production rules;
6. (**Production Improvement**) Use a critic to summarize the end condition of the task for future use and improve the generated productions.

The above procedures are repeated for all task families in the curriculum. While the agent might not fully learn every scenario of a task before moving on to the next one, it can still query the LLM later on to generate a production rule for a previously learned task. The training of a task is considered complete as long as the agent has sufficient experience with the task to generate a reasonable end condition such that future tasks can reuse the previously learned tasks.

Action Selection

The LLM is prompted with the current task, a summary of the current state, and a list of options available to the robot, which include both motor actions on the environment (e.g., move to a specific location) and internal actions (e.g., attend to a new subtask). For each previously trained subtask, we provide the end condition generated by the critic for the LLM to evaluate its relevance. Like the task names, the actions can also be parameterized (e.g., `move to <receptacle>`), and the LLM can replace `<receptacle>` with anything as it sees fit.

We use chain-of-thought prompting (Wei et al. 2022), which explicitly instructs the LLM to respond to the prompt in a step-by-step manner, probing it to make the most informed decision. The LLM is instructed to reflect on common strategies for approaching the task, analyze the current situation, and evaluate the usefulness of each action before suggesting one option for the robot to take. The LLM is also prompted to state the purpose of the chosen action, which will inform the production rule generation later.

Production Generation

Although the production rules are generated based on the current state, we represent them not as plans for the current task, but instead as underlying decision-making principles for all similar scenarios. For example, if the current task is to `find a/an egg`, instead of suggesting the action sequence of exploring every cabinet in the current environment, a desirable production rule would suggest "whenever you need to find something, you should first explore the unexplored places where that object is commonly stored". This is a systematic generalization that can be applied to finding any objects, not just eggs, and also can be applied to novel environments with different layouts and receptacle types.

Listing 1: Production interface

```python
class GeneratedProduction(Production):
  def precondition(self, agent) -> bool:
    # Returns whether the production is
        applicable given the agent
    # Set variables as side-effects
  def apply(self) -> str:
    # Returns the effect
    # Based on the variable bindings
```

To generate desired production rules, we use a two-step process. The first step summarizes the action selection process and generates the English description of the production rule; the second step then converts it into executable Python code (Listing 1). This separation is inspired by how human beginners are instructed to build cognitive models (Laird 2017), and has two benefits: 1) it allows each query to the LLM to be of reasonable length ($\sim 5k$ tokens), preventing LLMs from losing focus on lengthy prompts (Liu et al. 2023b); and 2) it facilitates a modular design, which enables generating code from English descriptions generated from other sources, including human feedback and post-generation self-reflection.

For each step, we also use the chain-of-thought prompting technique. For English description generation, the LLM is given the entire history of the action selection process, and is instructed to take four steps: 1) identify relevant information that leads to choosing the action; 2) generate a specific production rule that describes the current situation; 3) identify the potentially generalizable components in the specific rule and how they can be generalized; and 4) replace the components to form the generalized production description.

For code generation, the LLM is given the Python interface of querying declarative memory and the current task, and is instructed to take another four steps: 1) plan what variable bindings are needed; and how their values should be assigned, 2) analyze the predicates in the precondition and associate them with relevant variables; 3) plan how each predicate should be tested using the provided function interfaces; and 4) fill in the production template. The code snippet is parsed from the response and imported into the agent.

Production Improvement

We use three mechanisms to monitor and improve the common interface mismatch, over-constraining, and over-generalization problems of the LLM-generated productions.

Similar to the iterative prompting design in Voyager (Wang et al. 2023), the agent replays the generated production rule on the state from which it was generated, and ensures that its precondition check passes the current conditions. This fixes most function interface mismatches, as the generated production has to comply with a specific naming scheme and the interface of the declarative knowledge.

However, passing the precondition test for a single instance does not guarantee that production is ideal. As the LLM has access to accumulated observations from the past during the action selection process, it might include unnecessary conditions that happen to be true in the production's precondition, over-constraining it. This is handled by a critic LLM that summarizes the end condition of the task and provides suggestions on the existing productions.

The critic LLM is given the name of the task family (e.g., `find a/an <object>`), and the English descriptions, generated by the production LLM, of the existing production rules for that task. The critic LLM is instructed to first analyze all the production rules whose effect is the `done` action, and summarize the end condition of the given task (e.g., `the robot is holding the desired object in its gripper`). These end conditions summarize the behavior of the previously learned tasks to inform the action selection process for future tasks. This summary will be added to the prompt when querying for tasks later in the curriculum to incentivize reusing previously learned tasks. Next, for each production rule, the LLM either keeps it as is, removes it entirely, or modifies it. The modifications are in the English description space for the critic, and we make use of the two-step modularity of production generation to update the production rules.

Over-generalization happens when important features are left out of the production's precondition. For example, for the `pick and place` task, the LLM might generate a production rule that says:

```
IF task is pick and place <object> AND
   <object> in field of view AND
   gripper is empty
THEN pick <object>
```

This will make the robot pick up the object even when the object is already in the target receptacle. To prevent the agent from being stuck in an infinite loop, it will keep a state transition graph during the execution process and query the LLM for an alternative action once a cycle is detected using a depth-first search on the transition graph. Coupled with the production reinforcement (described below), the agent will prioritize loop-breaking productions.

Production Reinforcement

Following previous work in visual navigation (Anderson et al. 2018), the agent has to explicitly choose the special `done` action to indicate that it has completed the current task. We further extend this and give the agent a `quit` option to indicate that it believes the given task is impossible in the given environment. This is important as we allow the architecture to choose to attend to any subtask as it wants, and it should be able to realize when a task is impossible.

As we do not pre-define the goal condition during the bootstrapping process, we give a unit reward whenever the agent decides it is done with the current task. The reward propagates back through the shortest path to the starting state. For example, if the state transition is

$$S_0 \xRightarrow{P_1} S_1 \xRightarrow{P_2} S_2 \xRightarrow{P_3} S_0 \xRightarrow{P_4} S_4 \xRightarrow{P_5} S_5 \xRightarrow{P_{\text{done}}}$$

where S_0 is the start state and P_{done} is the production that yields the `done` action. Then the shortest path is

$$S_0 \xRightarrow{P_4} S_4 \xRightarrow{P_5} S_5 \xRightarrow{P_{\text{done}}}$$

Therefore only $P_4, P_5, P_{\text{done}}$ will receive a utility update, using the bellman backup (Sutton and Barto 2018).

$$U_{\text{after}}(P) \leftarrow \frac{1}{N(P)+1}\left(N(P) \cdot U_{\text{before}}(P) + \gamma^{\Delta_t}\right) \quad (1)$$

Where $U(P)$ is the utility of production P, $N(P)$ is the number of times P gets applied, Δ_t is the time difference from production application to the `done` action, and γ is the discount factor (which is set to 0.95 for our experiments).

When a subtask is involved, the utility is updated with respect to each task. For example, if the state transition is

$$A_0 \xRightarrow{P_1} A_1 \xRightarrow{P_2} \underbrace{B_3 \xRightarrow{Q_3} B_4 \xRightarrow{Q_4} B_5 \xRightarrow{Q_{\text{done}}}}_{\text{a subtask initiated by } P_2} A_6 \xRightarrow{P_{\text{done}}}$$

Where A and P correspond to the states and productions of the original task respectively and B and Q correspond to the states and productions of the subtask respectively. This will be treated as two separate utility update pathways

$$A_0 \xRightarrow{P_1} A_1 \xRightarrow{P_2} A_6 \xRightarrow{P_{\text{done}}} \quad \text{and} \quad B_3 \xRightarrow{Q_3} B_4 \xRightarrow{Q_4} B_5 \xRightarrow{Q_{\text{done}}}$$

If a subtask ends up with `quit` then there will be no utility update, not even negative ones. Because the task might be impossible due to environmental constraints, which has nothing to do with the production rules.

Intuitively, the closer a production brings the agent to choose `done` for its current task, the higher its utility is. This process is not provided to the LLM, so it has no incentive to "cheat" by proposing the `done` action all the time. We also explicitly tell the LLM to avoid selecting `done` or `quit` action unless it is "absolutely certain" about it. This worked empirically in our experiments.

This utility update process helps reduce the impact of hallucination in LLMs, as the knowledge is aggregated. For example, when tasked with "explore the countertops", the LLM may hallucinate and propose a production P_{bad} that keeps the agent exploring the cabinets after all countertops have been explored, instead of proposing the `done` action, as it should. However, when tasked with "explore the sink“ in the same bootstrapping section, the LLM may generate a production P_{good} that correctly identifies the termination condition and proposes `done` when all receptacles of the desired type have been explored. Then later, when the agent needs to explore all the countertops (potentially as a subtask of another task) and all of the countertops have been explored, both P_{bad} and P_{good} will be applicable. The agent will prioritize P_{good} because it is guaranteed to have a higher utility value than P_{bad}. On the other hand, if we use LLM to generate plans for each task, we may get a correct plan for the sink but an incorrect one for the countertops.

When multiple productions are applicable given the same environment knowledge, we resolve the conflict using the definition of noisy-optimal in previous works (Tian et al. 2023), where the probability of production P_i being selected and applied, given the current knowledge $\mathcal{K}$, is

$$\mathbb{P}(P_i \mid \mathcal{K}) \propto \mathbf{I}_{\mathcal{K}}(P_i) \cdot \exp(U(P_i)) \quad (2)$$

where $\mathbf{I}_{\mathcal{K}}(p)$ indicates that the preconditions of production p hold, given knowledge $\mathcal{K}$.

World Knowledge Base

For the sake of simplicity, we implemented the world knowledge of the agent as a dictionary that maps natural language statements to either true or false. Unlike many existing cognitive architectures that assume an absence of knowledge implies the negation, we explicitly differentiate between not knowing and knowing to be false. In the future, it can also be replaced with a real-valued vector database.

When a production rule is conditioned on a statement not previously known to the agent, the LLM is used to evaluate whether the statement is true, and the result will be saved to the knowledge base to be reused later. For instance, when bootstrapping the task of finding an egg, the agent will learn the production rule that says "If there is an unexplored receptacle where the object is commonly stored, explore that receptacle". But the agent does not know whether "egg is commonly stored in the fridge" is true or not initially, so it will query the LLM and memorize the positive response in its world knowledge base. Later when the agent is tasked to put things in their common storage place, the agent can reuse the knowledge and place eggs into the fridge. In addition to transferring to new tasks, the knowledge can be applied to new environments as well (e.g., eggs are commonly stored in fridges in most American households).

This knowledge base could be easily replaced by connecting it to an existing knowledge graph or ontology. But for the purpose of this paper, we are bootstrapping it from scratch.

Experiments

Setup

Following previous works in the embodied agents domain (Sarch et al. 2022; Trabucco et al. 2023), we evaluate our method in kitchen environments (see Figure 2) in the AI2THOR simulator (Kolve et al. 2017). As shown in Figure 2d, the agent has access to classification labels and attributes (e.g., "is opened") for objects that are close enough (within $1.6m$) or large enough (more than 5% of the frame). We also assume the agent already knows the names and locations of the large receptacles (e.g., cabinets, fridges, etc.) but does not know what objects are in the receptacles until it actively explores them.

We use three different tasks for evaluation:

- `find a/an <object>`: the goal is to have the specified object in the robot's field of view. This is a fundamental skill that is often overlooked or directly assumed in many of the previous works (Singh et al. 2023). We want to show that our framework can bootstrap very basic skills, in addition to composite actions.
- `slice a/an <object>`: the goal is to use a knife to slice an object. Because the robot can hold at most one item at a time, slicing involves a sequence of actions including finding the target object and the knife, putting them in the same place, and the final slice action. We want to show that our framework can handle tasks that involve multiple steps and tool use.
- `clear the countertops`: the goal is to have all the objects on the countertops moved to suitable storage places. This is a common household task that has also been investigated in previous work (Andrew et al. 2022; Sarch et al. 2022). We want to show that our framework can handle tasks that involve repeating similar subtasks.

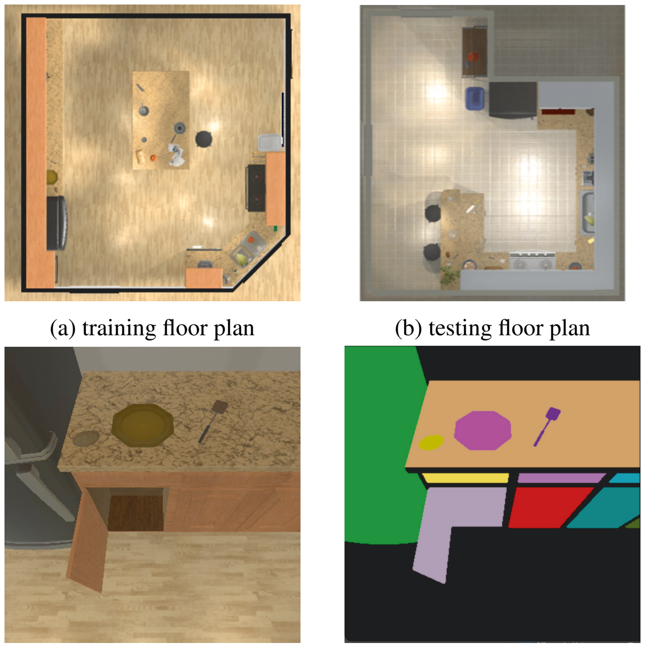
(a) training floor plan (b) testing floor plan (c) ego-centric view (d) instance segmentation

Figure 2: Screenshots of the AI2THOR simulator

The goal conditions listed above are used only for evaluation purposes, but are not provided to the LLM during training or testing. The LLM has to infer the goal condition from the task description only.

For `find` and `slice`, 5 target objects are chosen for each task, and we run 3 trials for each object where the initial locations of the objects are shuffled. For `clear the countertops` we run 3 trials each with 5 objects on the countertops that need to be put away. The specific objects and locations vary between trials, and the success of the agent is evaluated based on how many objects originally on the countertops have been relocated to other places. This results in 15 specific goal instances for each task family.

We use `GPT4-0613` (OpenAI 2023) for our experiments as previous works have shown that GPT3.5 is insufficient for code generation (Olausson et al. 2023; Wang et al. 2023). We set temperature to the 0 for the most deterministic response.

Conditions

For the experimental condition, we bootstrapped our agent with the following curriculum in the training floor plan:

```
1. explore <receptacle>
2. find a/an <object>
3. pick and place a/an <object>
        in/on a/an <receptacle>
4. slice a/an <object>
5. put things on the countertop away
```

This process generated 27 production rules in total. During test time, the agent can query the LLM for an immediate action if it does not have an applicable production rule for the current situation, but it cannot learn new production rules.

For the baseline condition of using LLMs to query only the actions, we omit the production generation steps and only use the action selection process within our framework. This ensures the prompts used by both conditions are the same, so LLM should suggest actions of similar quality. If the action proposed by the LLM leads to an affordance error, we query the LLM another two times, and if none of the actions are viable by the agent, then it raises a failure.

Although many works address the rearrangement task (Sarch et al. 2022; Wu et al. 2023a), they are not appropriate baselines as their architectures already encode the general strategies (e.g., first determine the target receptacle for each object, then navigate to the target area, etc.) while our approach bootstraps everything from scratch. Similarly, a hand-coded cognitive agent by human experts may perform even better but that defeats the purpose of eliminating the need for manual coding of knowledge. Other code generation works cannot handle multiple instances of the same kind (Singh et al. 2023) or understand the slicing preconditions (Song et al. 2022) without non-trivial modifications.

Results

Table 1 shows the quantitative results of different types of agents performing each kitchen task. The action-only baseline successfully completes all tasks but one, where it assumes `find a/an mug` is equivalent to `find a/an cup`, and ends the search pre-maturely without exploring the sink where the mug is actually located. On the other hand, our bootstrapped agent is able to finish most tasks completely using its learned production rule. The only exceptions are when it is tasked to find an object that was not part of its training environment. But with very limited additional queries, the bootstrapped agent is able to successfully complete those tasks as well. This shows that the knowledge in the bootstrapped agent can be easily transferred to new objects in new environments.

The success rate and number of query tokens show two advantages of our framework. First, it is verifiable such that it does not make false assumptions (e.g., confusing mugs with cups). Second, it is much more efficient to be deployed into new environments as the production rules it learns can be easily transferred and require minimal further assistance from the LLM, saving computations and costs.

We use a paired sample t-test to compare the number of steps taken by both agents. No significant evidence suggests that the two agents perform differently in `find` or `slice` tasks (p-values 0.446 and 0.347, respectively). This is not surprising as the knowledge source of both agents is the same LLM.

However, the bootstrapped agent is taking longer in the clearing task with significance (p-value 0.001), which results from a stylistic difference between the two agents. As shown in Figures 3a and 3b, the bootstrapped agent places everything into an individual cabinet while the baseline places multiple objects in the same cabinets. This is because one of the productions generated is “if there is an object on the countertop and there is an empty receptacle, attend to the subtask pick up the object, and place it into the empty receptacle”. This production gets reused repeatedly, requiring the agent to seek a unique empty receptacle before placing each object instead of putting every object in the same cabinet. By contrast, the baseline agent is making decisions on a case-by-case basis, so it does not enforce that the target receptacle has to be empty.

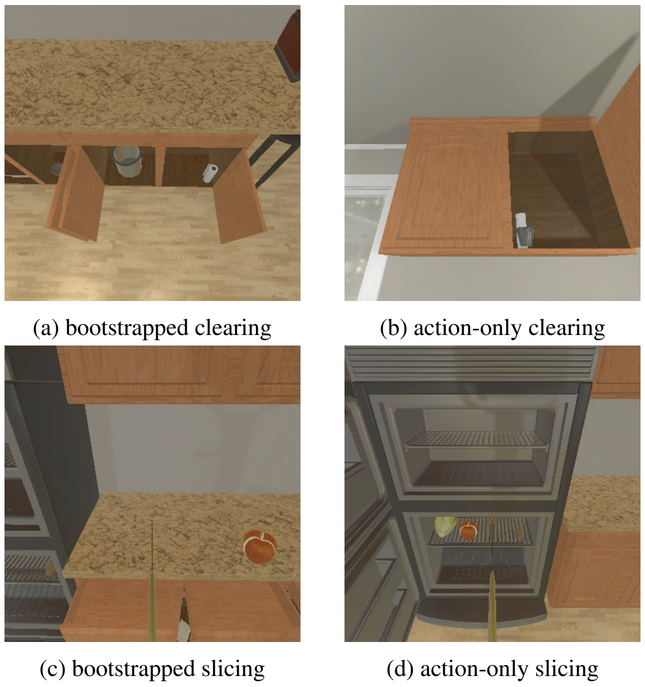
(a) bootstrapped clearing (b) action-only clearing (c) bootstrapped slicing (d) action-only slicing

Figure 3: Examples of task execution. The first row shows the bootstrapped agent put each object in their own cabinet while the baseline agent put multiple objects in the same cabinet. The second row shows the bootstrapped agent sliced the apple on the countertop while the baseline agent sliced the apple at its current location.

A similar difference is also found in the `slice` task where the bootstrapped agent always moves the objects to the countertops before slicing while the baseline agent slices objects at their current location (Figures 3c and 3d).

Production Analysis

The following are some learned productions:

- IF the current task is to find a/an `<object>` AND the `<object>` is located on `<location>` AND the robot is not at `<location>` THEN choose motor action: move to `<location>`.
- IF the current task is to slice a/an `<sliceable>` AND the robot is holding a/an `<sliceable>` AND there is no `<tool>` in the spatial knowledge or object knowledge THEN choose ’attend to subtask: find a/an `<tool>`’.
- IF the current task is to clear objects from

Task	Agent	Success ↑	Success w/o LLM ↑	Steps ↓	Tokens ↓
`find a/an <object>`	action-only	14/15	-	15.67	54754.20
	bootstrapped (ours)	15/15	12/15	15.80	916.87
`slice a/an <object>`	action-only	15/15	-	28.20	102806.60
	bootstrapped (ours)	15/15	15/15	29.13	0.00
`clear the countertops`	action-only	15/15	-	5.13	18924.87
	bootstrapped (ours)	15/15	15/15	7.47	0.00

Table 1: Result of experiments on household tasks. Completion steps and tokens are averaged over all task instances

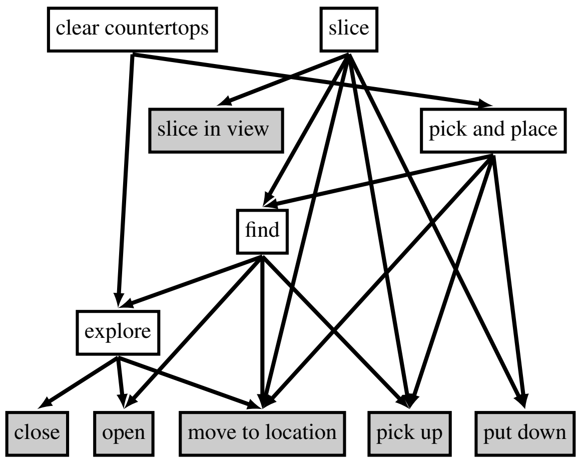

Figure 4: The hierarchy of tasks learned. Gray nodes denote the built-in functions of the robot, and white nodes represent the tasks learned from the curriculum. For built-in actions that involve an object (e.g., close), the object has to be within the field of view for the action to be taken. Special actions (i.e., done and quit) are omitted due to space constraints.

a/an `<receptacle_type>` AND all the `<receptacle_type>` are empty THEN choose special action: 'done'.

These show that the agent is able to represent different aspects of the given tasks using production rules. The first represents a common strategy for finding things, namely how to find things with a known location. The second represents decomposing complex tasks and reusing previously learned tasks. The third is a correct termination condition for the exploration task that is generated directly by the LLM.

Figure 4 shows the task hierarchy learned by the agent after training on the given curriculum. It shows how previously learned tasks are used to perform new tasks. This reduces the number of queries needed for the LLM, fosters generality, and ensures the scalability of our approach.

Discussion

Explainability

Our framework touches upon all three aspects of explainability as defined by Milani et al. (2022). The preconditions of the productions directly specify the feature that is being used (feature importance). Each production rule corresponds to a specific scenario during the bootstrapping process when it is created, which helps determine the training points that influence the learned policy (learning process). Lastly, the production application process can be easily converted to a verifiable decision tree by merging the precondition checks of productions (policy-level explainability). As the production rules can be formally verified, they are preferable to black-box LLM models in safety-critical situations.

Limitations

In this work, we explore only the high-level decision-making process of the agent and rely heavily on having a well-defined interface for low-level actions, such as navigation and object manipulation. There will likely be a considerable sim-to-real gap when applying this to physical agents.

Additionally, the English description generation step requires the decision-making process to be articulable to be converted to production rules. This is hard for skills that cannot be fully expressed using language (e.g., sculpting).

Future Work

There are more learning opportunities in cognitive architectures such as updating the preconditions of productions or using separate productions for conflict resolutions. Also, large vision models can be used to generate production rules without separate perception modules (Wu et al. 2023b).

Additionally, it is well-acknowledged that human values and preferences are hard to represent with reward functions (Casper et al. 2023). However, the production rules are interpretable and can be modified to suit each individual without extensive computation. As they are also modular, updating one specific production rule does not affect the others. It would interesting to examine whether this framework will facilitate personalization in human-AI collaboration tasks. Specifically, the user can iteratively update the production rules to fit their preference without having to worry about the agent forgetting about how to perform the task.

Conclusion

This paper presents a framework for bootstrapping a cognitive architecture from the existing noisy knowledge in LLMs, with minimal human inputs. We demonstrated how such an agent could efficiently learn to perform kitchen tasks and be applied to new environments. This work generalizes using LLMs to generate plans and provides an alternative to purely data-driven foundation models. And finally, we shed light on how it will benefit personalized agents in the future.

Acknowledgments

The authors would like to thank Michelle Zhao and Daphne Chen for their feedback on the manuscript, and the artists from the Noun Project for distributing the icons in Figure 1 under the Creative Commons License. This work was partially supported by a CMU SURF grant and by the AI-CARING Institute (NSF IIS-2112633).

References

Ahn, M.; Brohan, A.; Brown, N.; Chebotar, Y.; Cortes, O.; David, B.; Finn, C.; Fu, C.; Gopalakrishnan, K.; Hausman, K.; et al. 2022. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*.

Anderson, J. R. 2009. *How can the human mind occur in the physical universe?* Oxford University Press.

Anderson, P.; Chang, A.; Chaplot, D. S.; Dosovitskiy, A.; Gupta, S.; Koltun, V.; Kosecka, J.; Malik, J.; Mottaghi, R.; Savva, M.; et al. 2018. On evaluation of embodied navigation agents. *arXiv preprint arXiv:1807.06757*.

Andrew, S.; Karmesh, Y.; Alex, C.; Vincent-Pierre, B.; Aaron, G.; Angel, C.; Manolis, S.; Zsolt, K.; and Dhruv, B. 2022. Habitat Rearrangement Challenge 2022. https://aihabitat.org/challenge/rearrange_2022. Accessed: 2023-01-02.

Casper, S.; Davies, X.; Shi, C.; Gilbert, T. K.; Scheurer, J.; Rando, J.; Freedman, R.; Korbak, T.; Lindner, D.; Freire, P.; et al. 2023. Open Problems and Fundamental Limitations of Reinforcement Learning from Human Feedback. *arXiv preprint arXiv:2307.15217*.

Di Palo, N.; Byravan, A.; Hasenclever, L.; Wulfmeier, M.; Heess, N.; and Riedmiller, M. 2023. Towards a unified agent with foundation models. In *Workshop on Reincarnating Reinforcement Learning at ICLR 2023*.

Driess, D.; Xia, F.; Sajjadi, M. S.; Lynch, C.; Chowdhery, A.; Ichter, B.; Wahid, A.; Tompson, J.; Vuong, Q.; Yu, T.; et al. 2023. Palm-e: An embodied multimodal language model. *arXiv preprint arXiv:2303.03378*.

Ellis, K.; Wong, C.; Nye, M.; Sablé-Meyer, M.; Morales, L.; Hewitt, L.; Cary, L.; Solar-Lezama, A.; and Tenenbaum, J. B. 2021. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd acm sigplan international conference on programming language design and implementation*, 835–850.

Hake, H. S.; Sibert, C.; and Stocco, A. 2022. Inferring a Cognitive Architecture from Multitask Neuroimaging Data: A Data-Driven Test of the Common Model of Cognition Using Granger Causality. *Topics in Cognitive Science*, 14(4): 845–859.

Huang, W.; Abbeel, P.; Pathak, D.; and Mordatch, I. 2022. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, 9118–9147. PMLR.

Kirk, J. R.; and Laird, J. E. 2019. Learning Hierarchical Symbolic Representations to Support Interactive Task Learning and Knowledge Transfer. In *IJCAI*, 6095–6102.

Kirk, J. R.; Wray, R. E.; Lindes, P.; and Laird, J. E. 2023. Integrating Diverse Knowledge Sources for Online One-shot Learning of Novel Tasks. arXiv:2208.09554.

Kolve, E.; Mottaghi, R.; Han, W.; VanderBilt, E.; Weihs, L.; Herrasti, A.; Deitke, M.; Ehsani, K.; Gordon, D.; Zhu, Y.; et al. 2017. Ai2-thor: An interactive 3d environment for visual ai. *arXiv preprint arXiv:1712.05474*.

Laird, J. E. 2017. SOAR 9.6.0 Tutorial. https://soar.eecs.umich.edu/articles/downloads/soar-suite/228-soar-tutorial-9-6-0. Accessed: 2023-01-02.

Laird, J. E. 2022. Introduction to Soar. *arXiv preprint arXiv:2205.03854*.

Laird, J. E.; Gluck, K.; Anderson, J.; Forbus, K. D.; Jenkins, O. C.; Lebiere, C.; Salvucci, D.; Scheutz, M.; Thomaz, A.; Trafton, G.; et al. 2017. Interactive task learning. *IEEE Intelligent Systems*, 32(4): 6–21.

Laird, J. E.; Lebiere, C.; and Rosenbloom, P. S. 2017. A standard model of the mind: Toward a common computational framework across artificial intelligence, cognitive science, neuroscience, and robotics. *Ai Magazine*, 38(4): 13–26.

Liang, J.; Huang, W.; Xia, F.; Xu, P.; Hausman, K.; Ichter, B.; Florence, P.; and Zeng, A. 2023. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 9493–9500. IEEE.

Lindes, J. R.; and Peter, W. 2023. Improving Knowledge Extraction from LLMs for Robotic Task Learning through Agent Analysis. *arXiv preprint arXiv:2306.06770*.

Liu, B.; Jiang, Y.; Zhang, X.; Liu, Q.; Zhang, S.; Biswas, J.; and Stone, P. 2023a. LLM+ P: Empowering Large Language Models with Optimal Planning Proficiency. *arXiv preprint arXiv:2304.11477*.

Liu, N. F.; Lin, K.; Hewitt, J.; Paranjape, A.; Bevilacqua, M.; Petroni, F.; and Liang, P. 2023b. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172*.

Madaan, A.; Zhou, S.; Alon, U.; Yang, Y.; and Neubig, G. 2022. Language models of code are few-shot commonsense learners. *arXiv preprint arXiv:2210.07128*.

Milani, S.; Topin, N.; Veloso, M.; and Fang, F. 2022. A survey of explainable reinforcement learning. *arXiv preprint arXiv:2202.08434*.

Mininger, A.; and Laird, J. E. 2022. A Demonstration of Compositional, Hierarchical Interactive Task Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, 13203–13205.

Newell, A. 1994. *Unified theories of cognition*. Harvard University Press.

Olausson, T. X.; Inala, J. P.; Wang, C.; Gao, J.; and Solar-Lezama, A. 2023. Demystifying GPT Self-Repair for Code Generation. arXiv:2306.09896.

OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774.

Park, J. S.; O'Brien, J. C.; Cai, C. J.; Morris, M. R.; Liang, P.; and Bernstein, M. S. 2023. Generative agents:

Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*.

Qiu, J.; Xu, M.; Han, W.; Moon, S.; and Zhao, D. 2023. Embodied Executable Policy Learning with Language-based Scene Summarization. *arXiv preprint arXiv:2306.05696*.

Sarch, G.; Fang, Z.; Harley, A. W.; Schydlo, P.; Tarr, M. J.; Gupta, S.; and Fragkiadaki, K. 2022. Tidee: Tidying up novel rooms using visuo-semantic commonsense priors. In *European Conference on Computer Vision*, 480–496. Springer.

Singh, I.; Blukis, V.; Mousavian, A.; Goyal, A.; Xu, D.; Tremblay, J.; Fox, D.; Thomason, J.; and Garg, A. 2023. ProgPrompt: Generating Situated Robot Task Plans using Large Language Models. In *International Conference on Robotics and Automation (ICRA)*.

Song, C. H.; Wu, J.; Washington, C.; Sadler, B. M.; Chao, W.-L.; and Su, Y. 2022. Llm-planner: Few-shot grounded planning for embodied agents with large language models. *arXiv preprint arXiv:2212.04088*.

Sutton, R. S.; and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT press.

Tian, R.; Tomizuka, M.; Dragan, A. D.; and Bajcsy, A. 2023. Towards Modeling and Influencing the Dynamics of Human Learning. In *Proceedings of the 2023 ACM/IEEE International Conference on Human-Robot Interaction*, 350–358.

Trabucco, B.; Sigurdsson, G. A.; Piramuthu, R.; Sukhatme, G. S.; and Salakhutdinov, R. 2023. A Simple Approach for Visual Room Rearrangement: 3D Mapping and Semantic Search. In *The Eleventh International Conference on Learning Representations*.

Vemprala, S.; Bonatti, R.; Bucker, A.; and Kapoor, A. 2023. ChatGPT for Robotics: Design Principles and Model Abilities. Technical Report MSR-TR-2023-8, Microsoft.

Wang, G.; Xie, Y.; Jiang, Y.; Mandlekar, A.; Xiao, C.; Zhu, Y.; Fan, L.; and Anandkumar, A. 2023. Voyager: An Open-Ended Embodied Agent with Large Language Models. arXiv:2305.16291.

Wei, J.; Wang, X.; Schuurmans, D.; Bosma, M.; Xia, F.; Chi, E.; Le, Q. V.; Zhou, D.; et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35: 24824–24837.

Wu, J.; Antonova, R.; Kan, A.; Lepert, M.; Zeng, A.; Song, S.; Bohg, J.; Rusinkiewicz, S.; and Funkhouser, T. 2023a. Tidybot: Personalized robot assistance with large language models. *arXiv preprint arXiv:2305.05658*.

Wu, W.; Yao, H.; Zhang, M.; Song, Y.; Ouyang, W.; and Wang, J. 2023b. GPT4Vis: What Can GPT-4 Do for Zero-shot Visual Recognition? *arXiv preprint arXiv:2311.15732*.

Xiang, J.; Tao, T.; Gu, Y.; Shu, T.; Wang, Z.; Yang, Z.; and Hu, Z. 2023. Language Models Meet World Models: Embodied Experiences Enhance Language Models. *arXiv preprint arXiv:2305.10626*.

Xie, Y.; Yu, C.; Zhu, T.; Bai, J.; Gong, Z.; and Soh, H. 2023. Translating natural language to planning goals with large-language models. *arXiv preprint arXiv:2302.05128*.

Ye, S.; Lauer, J.; Zhou, M.; Mathis, A.; and Mathis, M. W. 2023. AmadeusGPT: a natural language interface for interactive animal behavioral analysis. *arXiv preprint arXiv:2307.04858*.

Zhang, B.; and Soh, H. 2023. Large Language Models as Zero-Shot Human Models for Human-Robot Interaction. arXiv:2303.03548.

Zhu, X.; Chen, Y.; Tian, H.; Tao, C.; Su, W.; Yang, C.; Huang, G.; Li, B.; Lu, L.; Wang, X.; et al. 2023. Ghost in the Minecraft: Generally Capable Agents for Open-World Enviroments via Large Language Models with Text-based Knowledge and Memory. *arXiv preprint arXiv:2305.17144*.

Zou, A.; Wang, Z.; Kolter, J. Z.; and Fredrikson, M. 2023. Universal and Transferable Adversarial Attacks on Aligned Language Models. arXiv:2307.15043.