KnowledgeBot: Improving Assistive Robot for Task Completion and Live Interaction via Neuro-Symbolic Reasoning

Minqian Liu  Ying Shen  Barry Menglong Yao  Sijia Wang
Jingyuan Qi  Zhiyang Xu  Lifu Huang

Department of Computer Science
Virginia Tech

{minqianliu, yings, barryyao, sijiawang, jingyq1, zhiyangx, lifuh}@vt.edu

Abstract

As artificial intelligence continues to advance, the development of intelligent assistive agents that can help people in completing daily tasks has become an area of increasing interest. However, the automation of such agents presents several challenges: (1) the enormous variety of user-defined tasks, (2) the ability of complex reasoning and visual comprehension for carrying out actions, and (3) natural interaction with humans. In this paper, we introduce KnowledgeBot, a conversational embodied agent designed to address these challenges. KnowledgeBot is designed for the Alexa Prize SimBot Challenge. The primary goal of this agent is to accomplish various tasks effectively by interacting with users while ensuring an optimal user experience. To achieve this, KnowledgeBot processes user instructions and environmental perceptions correspondingly take actions and respond to users through a series of components, including symbolic reasoning and neural-based action planning and prediction modules, vision models for semantic segmentation and visual grounding, as well as a situated dialogue module. By integrating these components, our agent is capable of carrying out a variety of tasks interactively with users in a natural and conversational manner. We demonstrate the effectiveness of our approach by evaluating our approach on TEACH and Arena instruction following datasets.

1 Introduction

With the rapid advancement of artificial intelligence (AI), there is an increasing interest in the development of intelligent assistive agents, which have the capability to assist individuals to finish their daily tasks. One promising research direction is the creation of assistive robots that can operate in various environments and perform various tasks. Assistive robots can perform various tasks in everyday environments and have the potential to greatly improve the quality of life for individuals, particularly the elderly or those with disabilities.

Automating assistive robots presents numerous challenges in multiple aspects. First, the tasks users require assistance with are diverse and can range from basic activities such as making a toast to more complex ones like combing hair, making it challenging to acquire knowledge for such extensive activities. Second, the robot must employ complex reasoning and visual understanding to determine the proper sequence of actions. This involves accurately interpreting the scene from the environment in real-time, determining the actions to perform based on the target goal and the current state, monitoring state changes of objects after executing actions, and predicting and verifying if the

1st Alexa Prize Simbot Challenge.
actions have been executed successfully or if it’s the time to move on. Third, the robot must actively engage and interact with the user by asking questions or presenting alternatives to address ambiguity or confusion while performing tasks. Accomplishing this requires generating appropriate questions and interpreting human instructions and feedback accordingly.

To address these challenges, we propose KnowledgeBot, a conversational embodied agent that can understand user instructions, perceive the environment, and complete various tasks within a game-like environment through natural and engaging interactions with users. To address the first challenge, we propose incorporating both symbolic reasoning and neural-based action planning into the prediction modules to break down complicated user instructions and utilize pre-trained language models to construct sequences of sub-goals for task completion. For the second challenge of complex reasoning and visual grounding, we propose a Visual Grounding and a Viewpoint-based Navigation module. The Visual Grounding module utilizes both Area Grounding and Color Grounding methods to identify the object mask that best matches the objects in the given instruction. This approach enables the agent to accurately identify and locate objects within a complex visual environment. Additionally, the Viewpoint-based Navigation module facilitates the agent’s movement toward the desired location by analyzing the panoramic view from various viewpoints and determining if the target object is present within each view. This approach provides the agent with an effective means of navigating toward specific objects within a complex visual environment. To tackle the third challenge, to enhance user interaction while performing tasks, we present a Situated Dialogue Module that employs template-based methods to generate responses tailored to each unique case. Moreover, we design a Controller and Error Handler that integrates the separate modules into a cohesive system and helps to improve user experience by handling potential exceptions that could occur in various components.

We evaluate the performance of our KnowledgeBot in various task scenarios in TEACH [33] and Arena [10] instruction following benchmarks, showcasing its capability to effectively engage with users and complete tasks seamlessly and naturally.

2 System Overview

Problem Formulation Our high-level goal is to build an intelligent agent for completing various daily tasks. We first introduce the notion of session where a user interacts with the agent in a game aiming to complete a certain goal (e.g., repair the broken bowl with a time machine). In one session, the agent is required to take in the user instructions sequentially and acquire the environment perceptions (e.g., egocentric views and the states of previously executed actions). While there is an explicit goal in each game, the ultimate objectives of the agent are two-fold: (1) completing the goal with a minimal cost (i.e., with the least number of steps), and (2) maximizing the user experience (UX) during the game.
**System Workflow** We present the overall workflow of our system in Figure 2. Given the user instruction and visual observation from the environment as inputs, we first (1) preprocess and (2) parse the instruction into a more structured format. Step (1) and (2) consists of an ASR correction component, a co-reference resolution component, and an instruction parser based on dependency parsing. We then (3) predict a list of more fine-grained and atomic actions via either a symbolic reasoning program or a neural-based action planning and prediction module. Next, we (4) feed the action list into the controller and start to process each action in the list one by one. Meanwhile, we (5) use a vision model to produce a set of semantic segmentation masks from the visual observation. If the controller is processing an action that either performs navigation or requires returning an object mask, then we further (6) apply a visual grounding & navigation module to either navigate the agent to a certain viewpoint or ground the action with a desired object mask. Finally, we return the predicted action (and possibly the associated mask) to the Arena Environment. Depending on the status of the executed action, we (7) return a dialogue response back to the user and update the interaction history stored in the controller. If the action is not executed successfully, we will process the error code returned from the environment and handle it accordingly.

3 Instruction Processing and Understanding

3.1 ASR Correction

Owing to the inherent instability of ASR systems, they may generate words that, while phonetically similar, are entirely distinct. This may result in sentences containing erroneous words that fail to accurately convey the user’s intention, rendering them incomprehensible to our model. To address this issue, we developed and employed a post-processing methods for ASR output. Specifically, we systematically analyzed log files to identify and document recurrent and fixed-content ASR errors. These findings were recorded in a mapping file, which cataloged all high-frequency, fixed-context ASR errors. Should a user’s voice input be recognized as one of these error contents, it would be directly translated into the appropriate command. For example, "let’s chat" would be converted to "next challenge." It is important to note that, during the compilation of the mapping file, only error contents entirely unrelated to the task at hand were included. Should the error constitute a coherent and plausible command, it would be excluded from the mapping file to prevent misinterpretation of user instructions.
3.2 Co-reference Resolution

**Motivation**  The goal of the co-reference resolution is to identify all the linguistic expressions such as pronouns that refer to the same real-world entity. After grouping the expressions, one can replace the expressions with the name of the entity. Co-reference resolution is important in many high-level downstream tasks including natural language understanding, summarization, and question answering. Co-reference resolution is an important task in dialogue systems because it can maintain the coherence of machine-generated responses and allow the system to understand the users’ utterances. Without co-reference resolution, the dialogue system may be confused with the users’ intention and hence leads to a breakdown in the conversation.

**Formulation**  Following [19], we formulate the task of co-reference resolution as a set of assignments for every span in the input text. If the total number of tokens in the input text is M, then there are \( N = \frac{M(M+1)}{2} \) possible spans. The task of co-reference resolution is to assign each span \( i \) to an antecedent span \( j \) where \( 0 \leq j \leq i - 1 \). Hence the set of possible assignments for span \( i \) is \( \{0, 1, ..., i - 1\} \) where 0 denotes the dummy antecedent. If the dummy antecedent is selected, the current span is either not an entity or can not be connected with any previous span.

**Approach**  We use the higher-order coreference resolution model proposed in [20], due to it strong performance. The model leverages bidirectional LSTMs to compute the text representations. Each time, the scoring module takes in the representations of the span \( i \) and one of its antecedent span \( j \) and compute the score indicating the possibility of assigning \( i \) to the antecedent span \( j \). During inference, the higher-order coreference resolution model iteratively refines the span representations conditioned on higher-order structures and performs a coarse-to-fine antecedent pruning to reduce the computation cost of inference.

3.3 Profanity Checking

**Motivation**  With the expanding user base of online platforms, the presence of inappropriate content can rapidly accumulate, leading to a significant issue, especially in the context of online dialogue systems. Online dialogue systems generate responses based on users’ input, and the presence of inappropriate language in the input can lead to the generation of inappropriate responses by the dialogue systems. This problem can have severe consequences such as damaging the brand’s reputation, legal liability, and hurting the user experience. To address this issue, previously many efforts have been devoted to automatically detecting profanity language in dialogue systems. However, the traditional word-filter-based methods are not effective enough as inappropriate language depends on multi-dimensional factors such as the discourse context, the domain of the conversation, and the temporal relation between the conversation and world events. In our KnowledgeBot, we utilize a neural network-based profanity detection model, namely Profanity-Check, which offers low latency and high accuracy for detecting profanity in online dialogue systems.

**Problem Formulation**  We formulate profanity checking into a text classification problem. Formally, given an utterance from a user, \( w \in X \) and a binary set of classes \( C = \{0, 1\} \) where 0 denotes not profanity and 1 denotes profanity. The goal of this task is to train a classifier that maps the user utterances to classes: \( f_\theta : X \rightarrow C \), where the function \( f \) is parameterized by \( \theta \).

**Approach**  We select Profanity-Check as our profanity checking model due to its low latency and high performance. Profanity-Check leverages the CountVectorizer class in scikit-learn to vectorize the input strings. The vectorizer computes the representations via Bag of Words. Profanity-Check uses Linear Support Vector Machine implemented by scikit-learn’s LinearSVC class as the classifier \( f_\theta \). Table [4] shows the speed comparison between Profanity-Check and Profanity-Filter. As one can observe, Profanity-Check is much more efficient.

---


<table>
<thead>
<tr>
<th>Model</th>
<th>1 Prediction (ms)</th>
<th>10 Prediction (ms)</th>
<th>100 Prediction (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profanity-Check</td>
<td>0.2</td>
<td>0.5</td>
<td>3.5</td>
</tr>
<tr>
<td>Profanity-Filter</td>
<td>60</td>
<td>1,200</td>
<td>13,000</td>
</tr>
</tbody>
</table>

Table 1: The time cost to make 1, 10, and 100 predictions by using Profanity-Check and Profanity-Filter.

**Training Details** The model is trained on a combined dataset collected from two publicly-available sources. (1) Twitter dataset is derived from a collection of tweets scraped from Twitter. (2) Wikipedia dataset is from the Toxic Comment Classification Challenge, which consists of comments from Wikipedia’s talk pages. Table 2 shows the statistics of the dataset:

<table>
<thead>
<tr>
<th></th>
<th>Not Offensive</th>
<th>Offensive</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tweets</td>
<td>4,163 (16.8%)</td>
<td>20,620 (83.2%)</td>
<td>24,783 (100%)</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>143,346 (89.8%)</td>
<td>16,225 (10.2%)</td>
<td>159,571 (100%)</td>
</tr>
<tr>
<td>Combined</td>
<td>147,509 (80.0%)</td>
<td>36,845 (20.0%)</td>
<td>184,354 (100%)</td>
</tr>
</tbody>
</table>

Table 2: The statistics of the two datasets used to train Profanity-Check.

During training, the Profanity-Check leverages *squared hinge loss* which is the square of the standard hinge loss, set the tolerance for stopping criteria to 1e-2, and the maximum number of iterations to 1e5. All other hyperparameters are set to default as in the LinearSVC class.

### 3.4 Instruction Parsing

Our approach involves the utilization of an Instruction Parser module that is specifically designed to parse natural language instructions into machine-readable actions. In the design of this module, we have taken into account two crucial factors. Firstly, users may provide nested or compound instructions, which could involve multiple actions being mentioned in a single statement. Secondly, it is possible for users to mention important objects that are not the primary action target. For instance, a user may instruct the model to fetch an object in the kitchen, while the bot is situated in the lab. In this case, correctly detecting the name of the room is a prerequisite for completing the task.

Formally, given a natural language sentence, $w = \{w_0, w_1, ..., w_n\}$, our Instruction Parser outputs a list of actions, consisting of an action predicate, a target object, and a list of other objects, $a = \{a_0, a_1, ..., a_k | a_i = (p_i, o_i, e_i)\}$. To do so, we first utilize spaCy parser to attach a part-of-speech (POS) tag and syntactic dependency relation for each token. Then we traverse the dependency tree to find a list of action predicates $[p_0, ..., p_k]$, for tokens whose POS tags are VERB. For each action predicate $p_i$ subtree, we take its first noun child phrase as the action object $o_i$ and all other nouns phrase as extra objects $e_i$. For example, given the instruction, *Place the control panel into the laser machine*, our Instruction Parser will yield an action tuple *(Place, control panel, [the laser machine])*.

### 4 Task Planning and Action Prediction

Performing task planning and action prediction given the user’s instruction and the current environment is the core component of the framework. The first crucial challenge is how to accurately and robustly map the user instruction to an executable action plan for the assistive robot to complete everyday tasks. On the one hand, the actions and objects defined in an environment are often a fixed set since they need to be executable by robots. On the other hand, depending on different user profiles (e.g., geopolitics, age, etc.), the same action or object can have various expressions in verbal language to describe.


[https://spacy.io/usage/linguistic-features](https://spacy.io/usage/linguistic-features)
Program | Input Instruction Example | Program Arguments | Output Action Sequence
--- | --- | --- | ---
BasicObjNavigation(obj) | pick up an apple | obj = Apple | (Goto, Apple), (PickUp, Apple)
EnhancedObjNavigation(obj) | pick up an apple | obj = Apple | (LookAround), (Goto, Apple), (PickUp, Apple)
GetPreqObj(main_obj, preq_obj) | place the milk on table | main_obj = Table, preq_obj = Milk | (Goto, Milk), (PickUp Milk), (Goto, Table), (Place, Table)
OpenContainer(main_obj, container) | pick up the milk in the fridge | main_obj = Milk, container = Fridge | (Goto, Fridge), (Open, Fridge), (Goto, Milk), (PickUp, Milk), (Close, Fridge)
UseTool(main_obj, tool) | heat the bread with microwave | main_obj = Bread, tool = Microwave | (Goto, Microwave), (Open, Microwave), GetPreqObj(Bread, Microwave), (Close, Microwave), (TurnOn, Microwave), (Open, Microwave), (PickUp, Bread), (Close, Microwave)

Table 3: Symbolic reasoning programs.

The second challenge is how to handle complex and composite instructions. Usually, the user will not directly use low-level instructions (e.g., move forward, turn left) that can be easily processed by the agent. Instead, they tend to use commands that are at a higher level and are more straightforward to humans. These instructions need to be decomposed into several actions such that the robot can successfully complete the task. For example, given a user’s instruction pick up the cake in the breakroom fridge, the agent needs to break it down into an action sequence: [(Goto, Breakroom), (Goto, Fridge), (Open, Fridge), (PickUp, Cake), (Close, Fridge)]. As such, the robot needs to be capable of understanding and processing complex commands such that it can bridge the gap between human’s instructions and executable robot action sequences.

In the following, we present our two different solutions to tackle the challenges above, respectively. We first describe an intuitive, robust, and extendable approach based on symbolic reasoning programs (Section 4.1), and then introduce a more generalizable neural PLANNER based on pre-trained language model associated with a neural action predictor, i.e., EXECUTOR (Section 4.2).

4.1 Instruction Following via Symbolic Reasoning

4.1.1 Action and Object Mapping

Given the parsed action tuples, our next step is to map the general action and objects expressions (e.g., search a bowl) into the finite action and object sets that are defined and executable in the Arena Environment (e.g., (Find, Bowl)). To achieve this, we apply mapping functions to map the actions and objects respectively. Specifically, we first predefine a comprehensive dictionary where the key is the executable action/object for the environment, and the value is a list of phrases that map to the action/object correspondingly. For a given action or object phrase from the parsed tuple, we traverse all the phrases in each list. When we find a matched action/object, we stop the program and return the matched action/object. If we cannot find a matched action/object after we traverse all the lists, we will return an empty action or object.

4.1.2 Decomposing Complex Instructions via Symbolic Reasoning Programs

While the instruction parser and the action/object mapping functions can already be used to predict the action sequence, they usually fail to handle complex or composite commands that cannot be parsed and mapped to executable commands. Also, in some cases, although the actions can be correctly processed, assistive actions (e.g., navigation) are still required to accurately execute the action as per requested by the user. As such, in order to build a robust agent that can handle various complex instructions and ensure a smooth user experience, we develop several symbolic reasoning programs that can be generally applicable to different scenarios. We list the most commonly used and important programs in practice in Table 3. Given the input instruction and the parsed tuples, the symbolic reasoning programs aim to augment the original raw action tuples to a more complete and executable action sequence.

Specifically, we elaborate on the following programs: (1) BasicObjNavigation: this is the most basic symbolic program that is executed every time the agent needs to interact with an object specified by the Obj argument. It is worth noting that the goto action is a navigation action supported
in the Arena Environment that can simplify the navigation process. The program can effectively mitigate the error type TargetOutOfRange since it can navigate the agent to a position that is close enough to interact with the object. (2) EnhancedObjNavigation: this is an enhanced version of BasicObjNavigation. It aims to handle the scenario when the grounding of goto action is failed because the required object does not occur in the agent's egocentric view. This program will prepend an additional LookAround action based on the two actions in BasicObjNavigation. As such, it can help the agent search for the object in its surrounding environment. If the agent still cannot find the object, then we roll back to a more complicated room-level navigation program that searches the entire room (described in Section 5.3). (3) GetPreqObj: in some scenarios, the action may involve two objects, such as place the soda on the table or pour the milk into the bowl. For such actions, we define a program that takes in two arguments: main_obj, i.e., the target/destination object of the action (e.g., table or bowl), and preq_obj standing for the prerequisite object that needs to be held in robot's hand in the first place. This program ensures the robot must have been holding the prerequisite object before it executes the action. Note that this program is only executed when the preq_obj is not currently at the robot's hand. (4) OpenContainer: some actions involve a main object (main_obj) and a container that holding the main object. The OpenContainer program is thus designed to first navigate to the container, open the container, and then pick up the main object. (5) UseTool: some actions require to use an instrument or tool such as a microwave or time machine to complete the instruction, which needs a more complicated sequence of actions. Therefore, we design the UseTool program that predicts a long sequence of actions.

To sum up, these general symbolic reasoning programs enable our KnowledgeBot to robustly execute complex instructions that require commonsense reasoning.

4.2 Task Planning via Pre-trained Language Models

Inspired by how humans complete tasks, this method emulates this process. A person typically begins with a high-level plan and then adapts and improves the plan based on the surroundings. Therefore, we propose a new hierarchical structure of task completion that first creates high-level sub-goal instructions, then generates primitive actions by combining the generated high-level instructions with environmental context. Our model consists of a Planner that produces high-level sub-goal instructions from high-level tasks (e.g., “Make Coffee”) and an Executor that generates primitive actions conditioned on generated sub-goal instructions and environment context.

4.2.1 Planner

We investigated two variants of Planner, as illustrated in Figure 3. We adopt T5 [35] as the backbone of Planner’s encoder and decoder structure.

**Complete Generation** For this variant, we employed T5-large to train an additional object generator that generates relevant object information for the task. First, the object generator takes in task description \( g \) and outputs a list of involved objects \( \hat{o} \). Then, we take the generated objects \( \hat{o} \) as discrete prompts and concatenate them with the task description \( g \) to form the input for the encoder.

**Step-by-step Generation** The input for this variant is the concatenation of task description \( g \) and history sub-goal sequence \( sg_{<t} \). We also introduced two special token pairs <task>, </task> and <steps>, </steps> to mark the beginning and end of the task and steps, respectively.

We use the same object generator as in Complete Generation to incorporate object information. However, instead of directly utilizing the entire generated object list \( \hat{o} \), we applied the attention mechanism to compute the representation of the current object \( \hat{o}_t \). Specifically, we concatenate the last hidden state of <task> and <steps> from the encoder and use the concatenated hidden states \( [h_t : h_s] \) as the query. We obtain the current object states \( h_{\hat{o}_t} \) as follows:

\[
 h_{\hat{o}_t} = \text{Attention}(Q = [h_t : h_s], K = h_{\hat{o}}, V = h_{\hat{o}}) \tag{1}
\]

\[
 = \text{Softmax}(\frac{QK^T}{T})V \tag{2}
\]

where \( h_{\hat{o}} \) denotes the representations of the object lists and \( T \) denotes the temperature term. We set a small value of \( T \) to enforce the approximated hard attention on one single object.
Subsequently, we use the current object states $h_{\delta}$, as a soft prompt and concatenate it with the contextual encoding $h_{g}$ from the encoder, and feed the concatenated hidden states as input to the decoder. The decoder then generates the current sub-goal $s_{g_{t}}$.

Implementation Details  We train both the PLANNER and object generator using T5-large with a batch size of 6 and a learning rate of 3e-05.

5 Visual Understanding and Navigation

5.1 Object Detection

Given the object class $O_{instruction}$ predicted based on the user instruction, the target of the object detection module is to extract the corresponding one or multiple image masks from the images collected in the Arena environment for the current view. We first apply a Mask-RCNN image segmentation model [13] trained on the vision dataset 8.1.1 to extract object masks and their corresponding visual object classes \{$V_{1image}, V_{2image}, ..., V_{nimage}$\} from the observed images. We then map the object class $O_{instruction}$ in the instruction to the corresponding visual object class $V_{instruction}$ based on the predefined dictionary. We finally obtain all image masks \{$M_{i}, M_{j}, ...$\} with the visual object class $V_{image}^{i} = V_{image}^{j} = V_{instruction}$.

5.2 Visual Grounding

In the case that we obtain multiple image masks from the object detection module, the visual grounding module is designed to choose the best-matched mask with respect to the instruction. Two grounding strategies are applied, namely Area Grounding, where we choose the mask with the biggest area, and Color Grounding, where we choose the mask whose corresponding object has the dominant color most similar to the expected color mentioned in the instruction if any.
Algorithm 1: Viewpoint-based Navigation with Visual Grounding

**Input:** Query object class $q$ associated with its attribute set $A$, the viewpoint list in the current room $V = [v_1, ..., v_n]$, segmentation model $f$, grounding model $g$.

**Output:** Predicted mask for the query object $M_{out}$.

1. Initialize the best attribute distance $d_{min}$ as $+\infty$;
2. Initialize the predicted object mask $M_{out}$ as $\emptyset$;
3. for $v_i$ in $V$ do
   4. Execute (Goto, $v_i$);
   5. Execute (LookAround) and obtain an egocentric image list $I = [h_1, ..., h_e]$ at $v_i$;
   6. for $h_j$ in $I$ do
      7. $(c_k, M_k) \leftarrow f(h_j)$; /* Predict the object class $c_k$ and associated segmentation mask $M_k$ for the image $h_j$. */
      8. for each $(c_k, M_k)$ do
         9. if $c_k = q$ then
            10. $d_k \leftarrow g(M_k, h_j, A)$; // Compute the attribute distance for $M_k$.
            11. if $d_k < d_{min}$ then
               12. $d_{min} = d_k$; // Update the minimal attribute distance.
               13. $M_{out} = M_k$; // Update the output mask.
      end
   end
   16. if $M_{out} \neq \emptyset$ then
      17. return $M_{out}$; // Found an matched object and return the mask.
   end
19. return $M_{out}$; // Cannot find the object in the current room.

To extract the dominant color for the detected object, we design the color detector based on 8. We first apply the mask to the original image to crop out the detected object. We then use k-mean with the cluster_number=3 to cluster RGBA color values of its pixels to get the dominant color value. Note that the obtained dominant color value, like 00FF01, may not exactly match the color value, like 00FF00(green), from the instruction. So we further apply a distance-based method to map the dominant color to one of the pre-defined color candidates, like {red, blue, green, white, yellow}, which exists in the Arena environment. In detail, we convert the dominant color value to the cie space 9 so that we can compute the Delta E (CIE2000) distance [38] between the clustered dominant color value and the candidate colors and obtain the predicted dominant color within the candidates and its Delta E (CIE2000) distance. The mask with the minimum distance will be chosen as the predicted mask for the object mentioned in the instruction.

5.3 Navigation

In Section 4.1.2 we have introduced two symbolic reasoning programs that are for basic navigation, i.e., BasicObjNavigation and EnhancedObjNavigation. In this section, we further elaborate on how we design the viewpoint-based navigation program.

Particularly, when the required object still cannot be found after executing BasicObjNavigation and EnhancedObjNavigation programs, we then start our viewpoint-based navigation as presented in Algorithm 1. If the navigation program still cannot find the query object after searching for the entire room, then the robot will return an error-handling dialogue to the user and instruct them to try to search other rooms or read the hint. Note that the viewpoint list and the goto viewpoint action are pre-defined in the Arena Environment.

6 Situated Dialogue Response

To effectively tackle complex real-world tasks, a nuanced approach is required to address the diverse range of situations that may arise. Our proposed solution is a Situated Dialogue Response module that

---

8 https://pypi.org/project/dominant-color-detection/
9 https://en.wikipedia.org/wiki/CIE_1931_color_space
### Table 4: Examples of situated dialogue response.

<table>
<thead>
<tr>
<th>Case</th>
<th>User Instruction Example</th>
<th>Robot Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greeting</td>
<td>&quot;Let’s get started.&quot;</td>
<td>&quot;Great! Let us get the job done together!&quot;</td>
</tr>
<tr>
<td>Action Successful</td>
<td>&quot;move forward&quot;</td>
<td>&quot;Done. What’s next!?&quot;</td>
</tr>
<tr>
<td>Ask for Task Guidance</td>
<td>&quot;What can I do in this game!?&quot;</td>
<td>&quot;The task is displayed on the top-left corner of your screen and the map is displayed on the top-right corner. Simply tell me the first step to start. You can also say read the sticky note to know more about the task.&quot;</td>
</tr>
<tr>
<td>Ask for Clarification</td>
<td>&quot;take&quot;</td>
<td>&quot;Sorry, I only heard ‘take’. Could you try it again or give me more details?&quot;</td>
</tr>
<tr>
<td>Long Action Sequence</td>
<td>&quot;pick up the milk from fridge and pour it into the bowl.&quot;</td>
<td>&quot;Got it. I will do it for you.&quot;</td>
</tr>
<tr>
<td>Start Navigation</td>
<td>&quot;find a trophy in this room&quot;</td>
<td>&quot;Searching ‘trophy’ for you.&quot;</td>
</tr>
<tr>
<td>Encounter Block</td>
<td>N/A</td>
<td>&quot;Is there a box blocks the way? If yes, please say turn on robotic arm and I will remove it for you. If no, please continue.&quot;</td>
</tr>
<tr>
<td>Object Not Found</td>
<td>&quot;find a trophy in this room&quot;</td>
<td>&quot;Sorry, I cannot find ‘trophy’ in this room. Could you try it again or try another room?&quot;</td>
</tr>
<tr>
<td>No Action Predicted</td>
<td>&quot;increase the brightness of the device&quot;</td>
<td>&quot;Sorry, I do not know how to do this. Could you try again or give me more details on how to complete the task step by step? You can also try to find and read a sticky note for hints.&quot;</td>
</tr>
<tr>
<td>Failed Coreference</td>
<td>&quot;place it on table.&quot;</td>
<td>&quot;Sorry, I do not know which one you are referring to. Can you say it again with more details?&quot;</td>
</tr>
<tr>
<td>Profanity Checking</td>
<td>&quot;pick up a knife and kill yourself.&quot;</td>
<td>&quot;Sorry, I cannot help you with that.&quot;</td>
</tr>
</tbody>
</table>

Note: all the user instruction and robot response examples presented in this table come from the team’s interaction and they are not from real customer interaction.

utilizes template-based methods to generate responses tailored to each unique case. This approach enables us to provide sophisticated and context-specific to accommodate a diverse range of scenarios. Examples of situated dialogue responses are shown in Table 4.

To initiate a conversation, Greeting templates are utilized to welcome the user. In order to cater to different requested instruction statuses, we propose four distinct types of templates. These templates include the Action Successful template which serves as a closure for the current task while also enabling the bot to commence a new task if required. The Ask for Task Guidance template is used in instances where the user requests guidance to complete the task, while the Ask for Clarification template is utilized when the bot requires additional information to execute the task. We also provide a response template for instances where the bot is given a Long Action Sequence. In navigation-related scenarios, we offer three distinct response templates. These include the Start Navigation template to initiate navigation, the Encounter Block template if an object blocks the bot’s intended action, and the Object Not Found template if the requested object cannot be located. In addition, we have designed a template to address situations where the bot fails to find the coreference for an ambiguous pronoun. Lastly, we provide a response template if the instruction fails Profanity Checking.

## 7 Controller and Error Handler

In addition to the components for processing and predicting the action sequence, as well as the visual grounding and navigation modules, we also need a **higher-level controller and error handler** to combine the separate modules into a whole working system and handle the exceptions that may occur in different components. In the following, we will introduce how we store important interaction information via an Internal Memory, and then elaborate on how we design the Controller that controls the action flow and update the Internal Memory. Finally, we will describe how we handle the exceptions that occurred during executing the actions in the environment.

### 7.1 Internal Memory for User Interaction

During the interaction with the user, it is essential to keep track of the important information such that the agent can execute the actions that require knowing the current state. For example, to execute
the actions for place the milk on table, the agent needs to first confirm if milk is on its hand. If not, then the agent needs to first execute the GetPreqObj program to fetch the milk.

We present the essential storage variables in Table 5. Specifically, (1) the Instruction History is used to store users’ instruction history so that we can refer back to previous instructions when we need them; (2) we use Object in Hand to record the object that is held by the robot so that we can determine if the robot needs to go and fetch an object; (3) the Previous Session ID aims to check if the session ID of interactions is changed. If so, then we will clear some internal storage such as Instruction History, Object in Hand, and Number of Interaction Rounds; (4) we use Consecutive Error to record whether the robot consecutively encounters errors during interactions. If so, we will give a more detailed instructions on how to play the game and find the hint to complete the tasks.

7.2 Controller

The task planner or action prediction module can predict multiple actions at a time (i.e., action plan). However, many action sequences cannot be directly returned to Arena Environment at a time and each of the actions in the sequence needs to be returned sequentially. Meanwhile, when some actions are executed successfully, we need to update the Internal Storage correspondingly. As such, we design a Controller to control the action execution flow.

7.3 Error Handler

In order to optimize human-machine interaction processes specifically for various error types, we designed a refined error handler. After analyzing numerous actual cases, we matched each error type with corresponding practical arena scenarios. We identified 14 distinct error types and devised corresponding dialogue responses based on the error codes returned from the Arena Environment. When users’ utterances are inappropriate or erroneous, the error handler can provide targeted responses and guide users to offer correct, recognizable instructions.

For instance, when the action specified in the user utterance cannot be executed in the Simbot arena, the error handler identifies this category of error and returns: "This action is not supported. Could you please try something else?", prompting the user to change the action instruction. Table 6 displays the 14 error types and their respective response statements.

If none of the predefined error types match the generated error, the error handler will return a default error response. Throughout the optimization process, we continually monitor default error cases (indicating error types unaddressed by the current handler) and update the error handler accordingly to encompass all potential error types.
### Table 6: Dialogue responses for error handler.

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Handler’s Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>&quot;Sorry, something went wrong. Please try it again and maybe say it in a different way.&quot;</td>
</tr>
<tr>
<td>UnsupportedAction</td>
<td>&quot;This action is not supported. Could you please try something else?&quot;</td>
</tr>
<tr>
<td>UnsupportedNavigation</td>
<td>&quot;I cannot move this way. Could you try something else, such as turn around?&quot;</td>
</tr>
<tr>
<td>AlreadyHoldingObject</td>
<td>&quot;I’m already holding an object. Could you please first put the object somewhere?&quot;</td>
</tr>
<tr>
<td>ReceptacleIsFull</td>
<td>&quot;The receptacle is full. Please first try to empty it.&quot;</td>
</tr>
<tr>
<td>ReceptacleIsClosed</td>
<td>&quot;The receptacle is closed. Please first try to open it.&quot;</td>
</tr>
<tr>
<td>TargetInaccessible</td>
<td>&quot;I cannot access the object you specify. Please try to guide me there or try something else.&quot;</td>
</tr>
<tr>
<td>TargetOutOfRange</td>
<td>&quot;The target is out of range. Can you direct me to be closer to it?&quot;</td>
</tr>
<tr>
<td>InterruptedByNewCommandBatch</td>
<td>&quot;I am interrupted by a new command. Please instruct me step by step.&quot;</td>
</tr>
<tr>
<td>ObjectUnpowered</td>
<td>&quot;The object is not powered. Please first turn on the power.&quot;</td>
</tr>
<tr>
<td>ObjectOverloaded</td>
<td>&quot;The object is overloaded. Please first try to fix it or try something else.&quot;</td>
</tr>
<tr>
<td>NoFreeHand</td>
<td>&quot;I don’t have free hands. Please first place the object in my hand to somewhere.&quot;</td>
</tr>
<tr>
<td>InvalidCommand</td>
<td>&quot;The command is invalid. Please try other instructions.&quot;</td>
</tr>
<tr>
<td>ObjectNotPickedUp</td>
<td>&quot;I am not holding the object. Please first try to pick it up.&quot;</td>
</tr>
</tbody>
</table>

8 Experiments

8.1 Experiment Setups

8.1.1 Dataset

**Alexa Arena** The Alexa Arena [10] is a simulation platform for user-centric Embodied AI and it is the core platform for the Simbot Challenge and our KnowledgeBot system. We use the vision dataset to train our vision model, and use the instruction following benchmark to evaluate our task planner and action predictor. Specifically, we use the validation set of the dialog-guided task completion benchmark for evaluation. It contains 383 tasks with 1149 sessions. On the other hand, the vision dataset in the Arena Environment contains 600k training images spanning 336 unique objects, which are collected from the Arena environment. These visual data are generated by initializing game missions and then navigating to all the different objects present in the scene and capturing the corresponding egocentric view image.

**TEACH** We use the Execution from Dialogue History (EDH) subtask in the TEACH [33] dataset to train and evaluate the neural planner model. Each EDH instance consists of four components, $(S^E, A_H, A_I^R, F^E)$, where $S^E$ is the initial state of the EDH instance, $A_H$ is an action history, and the agent is tasked with predicting a sequence of actions that changes the environment state to $F^E$, using $A_I^R$ reference interaction actions taken in the session as supervision. To train the neural planner model, we leverage the action history $A_H$ and reference interaction actions $A_I^R$ as input and target output. TEACH dataset contains 5475 EDH Instances, 608 and 666 seen instances for validation and test, and 2157 and 2270 unseen instances for validation and test.
<table>
<thead>
<tr>
<th>Models</th>
<th>Unseen SR</th>
<th>Seen SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alexa Prize Team Baseline</td>
<td>9.70</td>
<td>12.46</td>
</tr>
<tr>
<td>PLANNER (COMPLETE)</td>
<td><strong>11.42</strong></td>
<td><strong>12.31</strong></td>
</tr>
</tbody>
</table>

Table 7: Unseen Success Rate (SR) (%) and Seen SR (%) on TEACh unseen test set.

<table>
<thead>
<tr>
<th>Models</th>
<th>Unseen SR</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLANNER (STEP-BY-STEP)</td>
<td>12.9</td>
</tr>
<tr>
<td>PLANNER (COMPLETE)</td>
<td><strong>13.6</strong></td>
</tr>
</tbody>
</table>

Table 8: Unseen Success Rate (SR) (%) on TEACh unseen validation set.

### 8.1.2 Evaluation

For the instruction following benchmark on Arena, we use the *Goal Completion* that calculates the fraction of the game sessions our agent can successfully accomplish. For the evaluation on TEACh, we use *Unseen Success Rate* which computes the fraction of tasks that all expected state changes are present after the agent finished all the actions.

In addition to evaluating our model on instruction following benchmarks, we also actively analyze the user ratings and feedback throughout the whole user interaction phases. Due to privacy policy, we do not show detailed examples of user feedback and interaction history in this report.

### 8.2 Results for Instruction Following

**Arena**

The Goal Completion rate of our approach on the Alexa Arena environment is **0.96%**, where we use symbolic reasoning to perform the instruction following. We found that our model performs worse in the offline evaluation compared to the online evaluation. We hypothesize this is caused by the following reasons: (1) the logic flow we design is mainly for the online scenario and it did not fit the offline code base well; (2) our designed symbolic programs may not be general enough; (3) real human users can try the instructions on different levels, where the low-level instructions can be more easily understood by our model. Due to the time constraint we did not extensively apply A/B test to evaluate the effectiveness of our symbolic programs.

**TEACh**

We first present the instruction following performance on the Execution from Dialogue History (EDH) subtask in TEACh to demonstrate the effectiveness of our neural PLANNER model proposed in Section 4.2. As shown in Table 7, our PLANNER model achieves a very promising improvement over the baseline by **1.72%**. We also show the comparison between the step-by-step and complete planner in Table 8. We found that the planner with complete steps is better than the one with step-by-step.

### 8.3 Error Analysis and Remaining Challenges

**High-level Command**

There are still a few high-level commands that could not be handled by our instruction processing and understanding modules. These high-level commands are split into 3 cases: (1) the command only contains some high-level action and goal, without detailed executable operation and information. For example, from the command "Change the white bowl to red", our module cannot generate the complete action list "go to the quantum lab. find the color changer. put the bowl on the color changer. turn on the color changer." because it lacks the information about *quantum* and *color changer*. (2) the command includes some complex concepts, for example, the relative position concept. For the command "go in front of the table", our model can only plan the "go to the table" action but could not figure out the relative position between Simbot and the table. (3) The utterance is more of a conversation than a direct order. For utterances like "This is the door I’m looking for," our model can’t use the user’s previous command history to infer that what they really want to say is "find another door."
<table>
<thead>
<tr>
<th><strong>Error Type</strong></th>
<th><strong>Error Example Utterance</strong></th>
<th><strong>Note</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>High-level command</strong></td>
<td>Change the white bowl to red</td>
<td>The model can only understand operation, but cannot infer the procedure of a high-level task.</td>
</tr>
<tr>
<td></td>
<td>This is the door i’m looking for</td>
<td>The model cannot understand utterances requiring a previous command history.</td>
</tr>
<tr>
<td></td>
<td>Go in front of the table</td>
<td>The model cannot understand the relative position concept.</td>
</tr>
<tr>
<td><strong>ASR error</strong></td>
<td>Let’s chat</td>
<td>The actual command is &quot;Next Challenge&quot;. The ASR returns similar pronunciation but incorrect words.</td>
</tr>
<tr>
<td></td>
<td>Play the app on the color change</td>
<td>The actual command is &quot;Place&quot;. The ASR returns similar pronunciation but incorrect words.</td>
</tr>
<tr>
<td></td>
<td>Cancel we build a support from sideways fifty times</td>
<td>The command is not a fluent English sentence.</td>
</tr>
<tr>
<td><strong>Target ambiguous</strong></td>
<td>Go in the room</td>
<td>No specific room name is given, and there is no door or room in the viewpoint.</td>
</tr>
<tr>
<td></td>
<td>Go to the lab</td>
<td>Not clarify which lab the user would like to go.</td>
</tr>
<tr>
<td><strong>Out of capability</strong></td>
<td>What is the weather in halletsville texas tomorrow?</td>
<td>The simbot cannot answer the real-world question.</td>
</tr>
<tr>
<td></td>
<td>Alexa walk sideways</td>
<td>The action is pre-defined.</td>
</tr>
<tr>
<td></td>
<td>Put-down the box</td>
<td>There is no &quot;box&quot; object in the arena.</td>
</tr>
<tr>
<td><strong>Task-irrelevant command</strong></td>
<td>Alexa we’re done</td>
<td>The utterance itself is not a command for an operation or action.</td>
</tr>
<tr>
<td></td>
<td>Ghost rider</td>
<td>The utterance is entirely task-irrelevant.</td>
</tr>
<tr>
<td><strong>Question utterance</strong></td>
<td>Can you see the redshelf?</td>
<td>The model cannot answer user’s question, and only can understand action command.</td>
</tr>
</tbody>
</table>

Table 9: Analysis of remaining errors.

**ASR Error** The incorrect ASR result may include similar pronunciation but entirely different words. This kind of error may lead to: (1) unreadable sentences, or (2) fluent but semantically wrong sentences. In the first case, the model is not able to deal with such utterances. For the second case, the model may execute actions that are not what the user wants to do.

**Target Ambiguous** Our model cannot figure out ambiguous targets based on the current situation and high-level task goals. If the user does not specify the target in the utterance, the model cannot generate corresponding actions.

**Out of Capability** Some commands are out of the capability of the Simbot. The Simbot cannot answer questions like "What is the weather in Halletsville texas tomorrow?", or execute undefined actions like "walk sideways".

**Task-irrelevant Command** Some utterances are task-irrelevant. For example, "Ghost rider".

**Question Utterance** When the user tries to ask a question to determine the situation, our model cannot give the correct response. For the utterance "Can you see the red shelf?", the user would like to check if the Simbot recognizes the red shelf first, and then do the following actions. However, our current model can only handle pre-defined questions.

9 **Related Works and Future Directions**

9.1 **Multimodal Pretraining**

Recent studies in multimodal pretraining have shown promising results in enabling multimodal models to perform various tasks and induce zero-shot capability. For instance, VL-T5 [5] addresses vision-and-language tasks by formulating them into a unified text generation task. Similarly, Flamingo [2]...
is trained on large-scale arbitrary interleaved multimedia corpora, allowing it to perform in-context few-shot learning. BEiT-3 [43] treats images as foreign languages and performs masked language objectives on images, texts, and image-text pairs. OFA [42] further extends text-only generation tasks to image generation. Uni-Perceiver v2 [21] is jointly trained on various multimodal tasks and can perform downstream tasks without task-specific adaptation. Additionally, MultiInstruct [45] proposes a large-scale multimodal instruction tuning dataset that further improves the generalizability of pre-trained multimodal models via instruction tuning. Finally, BLIP-2 [22] bridges a frozen pre-trained language model with a frozen pre-trained image encoder by a two-stage bootstrapping process. These advancements in multimodal pretraining demonstrate the potential of these models to tackle complex tasks and offer new opportunities for the development of general AI systems.

9.2 Embodied Vision-Language Planning

Recent advances in embodied AI and multimodal machine learning have given rise to various tasks in embodied vision-language planning tasks, such as Vision-Language Navigation (VLN) [3, 9, 16, 11], Vision-and-dialog Navigation [6, 32, 40], and Vision-Dialog Navigation and Task Completion [31, 39, 33, 48, 10]. This line of research aims to combine natural language commands with visual perception and physical action, empowering agents to navigate through embodied environments and perform diverse tasks. There are two main directions in this field. The first direction focused on building a unified multimodal model that can encode historical information from language, images, and action to predict the next action [34, 7]. One notable example is PaLM-E [7], a large embodied multimodal language model that can perform a variety of embodied reasoning tasks. The other direction utilizes the strong reasoning capabilities of large pre-trained language model (PLLM) for task planning and then grounds the plan to the environment [48, 17]. For instance, SayCan [1] proposes to ground language models in agents’ affordances by combining the candidate action’s probability from PLLM with the action’s affordance value. Furthermore, Inner Monologue [17] proposes to incorporate environment feedback into the LLM through natural language and combines large language model planning with robotic control policies.

9.3 Dialogue System for Task-oriented Embodied Agent

Task-oriented dialogue is a key area of research that aims to enable effective communication between humans and machines. To achieve this goal, most works in this field focus on several key components: (1) an intent detection to understand the user’s goals and needs [23]; (2) a dialogue state tracking module that helps to keep track of the constraints imposed by the user, which can be useful for guiding the conversation towards a successful outcome [29, 39, 15, 41, 30]; (3) a dialogue policy used to determine system actions that can help to achieve the user’s objectives [47]. By paying attention to these key components, researchers in the field of task-oriented dialogue are helping to create more effective and efficient dialogue systems that can better serve human needs. In addition to task-oriented dialogue systems under the supervised setting, many prior works also explore applying continual learning techniques [18, 27, 25] such that the dialogue system can continually evolve in a dynamic environment [28], which is especially important for the agents in the embodied environment.

9.4 Visual Understanding in Embodied Environment

Significant advancements in the visual understanding field have been made through research in various subfields, such as semantic/instance segmentation [26, 14, 46, 24] and object detection [37, 36]. FCN [26] introduced a fully convolutional architecture for semantic segmentation, while Faster R-CNN [37] revolutionized object detection with the use of Region Proposal Networks. Mask R-CNN [14] extended the Faster R-CNN framework for instance segmentation by adding a mask prediction branch. ADE20K [49], a comprehensive dataset for scene parsing, has become a standard benchmark for evaluating semantic segmentation algorithms. In addition, the YOLO [36] approach streamlined real-time object detection by predicting bounding boxes and class probabilities directly from a single convolutional network. On the other hand, based on the previous work, research in the visual understanding field for embodied environments has produced notable results, with works such as [3], which presents a self-improving embodied object detection approach that combines reinforcement learning and supervised fine-tuning. Cognitive mapping and planning for visual navigation tasks in embodied settings are explored in [12], introducing a method that learns to build a top-down belief map and plans paths using a differentiable planner. In [44], the authors tackle Embodied
Question Answering (EQA) tasks in photorealistic environments using point cloud perception, developing a model that reasons about its surroundings to answer questions. Lastly, \cite{4} focuses on learning exploration policies for navigation in embodied environments, using an unsupervised learning framework to train a neural network policy that effectively guides agents to explore novel regions. These works highlight the potential of combining visual understanding with embodied AI for creating intelligent agents, shaping the direction of future research in the field.

10 Conclusion

In this work, we present KnowledgeBot, a conversational embodied agent designed to complete real-life tasks by interacting with users for the Alexa SimBot Challenge. By combining natural language processing, computer vision, task planning, and execution, KnowledgeBot is able to handle an enormous variety of user-defined tasks, possess the ability to perform complex reasoning and visual comprehension to effectively carry out actions in the physical world, and engage in natural and seamless interactions with humans to ensure an optimal user experience. Overall, we believe the development of intelligent assistive agents like KnowledgeBot has the potential to revolutionize the way individuals interact with machines and complete their daily tasks.

References


