BEAST: Building an Embodied Action-prediction System with Trajectory data

Neeloy Chakraborty*, Risham Sidhu*,
Blerim Abdullai*, Haomiao Chen*, Nikil Ravi*,
Abhinav Ankur, Devika Prasad,
Julia Hockenmaier
Team Kingfisher
University of Illinois
Urbana, IL 61801

Abstract

We introduce our system BEAST (*B*uilding an *E*mbodied *A*ction-prediction *S*ystem with *T*rajectory data) for interactive instruction-following within the Alexa Arena Platform. Our system leverages the abstraction of navigation provided by the Arena to decouple the language and vision predictions. This allows for greater simplicity within the system, and for rapid augmentation of the trajectory data-set and training for our text-only action prediction model. By creating a framework with a focus towards user experience our system is more robust to errors in predictions, and informative to the user.

1 Introduction

The development of embodied agents that can understand and communicate in natural language, and can therefore complete tasks in the real world by following natural language instructions, is one of the most ambitious goals of artificial intelligence, requiring significant advances in robotics, natural language processing, vision, and planning. As is common in robotics research, much of the work in this domain is first developed in simulated environments. The Arena platform, developed by Amazon, makes it possible to develop agents that can navigate and manipulate a number of objects in simulated indoor 3D environments. In its interactive version, Arena runs on Amazon Echo Show smart displays as well as FireTV devices, allowing agents to receive natural language instructions based on spoken utterances by Echo Show users (automatically transcribed by Amazon Alexa's speech recognizer). This report describes an agent ("SimBot") that can follow natural language instructions in Arena that is being developed by the University of Illinois Kingfisher team as part of the Amazon Alexa Prize SimBot Challenge.

Arena differs from prior platforms not just in that it can be deployed on Echo Show devices, but also in that its API provides a high-level navigation (goto) function that only requires an object mask as an argument. We show that this design decision of allowing Arena agents to offload low-level navigation to an in-built motion planning component (invoked by this high-level navigation function) makes it possible to design much simpler embodied agents for which complex training data can easily be obtained at scale. Most current work on natural language instruction following in simulated environments assumes complex multimodal models that combine visual (image/video) and linguistic (user instructions) input. However, in-domain multimodal training data (consisting of natural language instructions, their executions, and corresponding video frames obtained during the execution of the instructions) for such models is very expensive to obtain. Our team has developed a framework for embodied agents for the Arena platform that separates action prediction from action

*These authors contributed equally to this work.

execution in such a way that highly accurate action prediction (semantic parsing) of even complex instructions can be done without any access to other sensorimotor inputs, allowing us to restrict the use of computer vision (or possibly vision-language models) to the task of executing a single action at a time. Thus, throughout this competition, we have built an embodied system to predict actions from trajectory data, or BEAST.

2 Background

2.1 The SimBot Challenge

The Alexa Prize SimBot Challenge aims to supplement the development of virtual assistants that are capable of completing real-world tasks. Team Kingfisher of the University of Illinois at Urbana-Champaign is one of the ten university teams selected to compete in the Alexa Prize Simbot Challenge. When users prompt Alexa to interact with the Arena game, they are unaware of which university agent they interact with; an agent is randomly selected each time. Users are then assigned a task with outlined subgoals in their UI. At the end of each interaction, users are offered an opportunity to rate their interaction on a scale of 1-5, and then provide feedback to supplement their rating. These ratings are amalgamated, and team performance is judged on an anonymous leaderboard with average scores.

The challenge began in August 2022, when the teams were introduced to the Simbot challenge and could begin development. Through mid-October, teams built their Simbot skill, and then underwent the skill certification required to start interfacing the skill with Alexa users. After passing certification, Alexa users began interacting with the skills and providing ratings and feedback.

3 System Architecture

In this section, we describe the architecture of our bot and our design decisions. We then introduce the provided sensor inputs from the Arena simulator and the expected actions from our bot, the control structure of our bot pipeline, and in-depth details of each sub-module within the pipeline.

3.1 Overall Architecture and Design Decisions

We chose to use a modular setup rather than a fully grounded model as this environment does not necessitate grounding to the same degree as some other problems. First, in the current state of the game, the instructions are generally quite simple and rarely require differentiating between different versions of objects (e.g. having multiple mugs that need to be differentiated by their location). What little differentiation needs to be done (e.g. picking the correct colored button for the color changer) can be done fully separately (e.g. using RGB values to pick the correct color). Furthermore, the action sequences associated with a particular instruction are often not more than two to three actions. This simple nature lends itself to a modular setup as it reduces the need to check states in long and complex action sequences. Also, the action space does not include navigation or path planning as this is abstracted away by the environment (i.e. we can directly $Goto : Apple$ instead of planning our path to the apple). Thus we only need to understand these high level symbolic actions, rather than relative locations, maps, etc. This also reduces our action space to a much more manageable size, further allowing us to forgo grounding within the action prediction model. Additionally, by excluding other modalities, we were easily able to create text-only augmented data to bolster model performance without collecting the parallel visual data for these action sequences. Thus, we were able to achieve strong performance without additional visual inputs.

3.2 Sensor Inputs and Bot Actions

As described by Gao *et al.*, the Alexa Arena simulator contains an interface for a user to provide instructions in the form of natural language [3]. Users provide instructions to the bot to complete a mission and maximize a game score that is unknown to the bot, while the bot takes physical actions in the simulator to execute the instructions provided by the user. At interaction step $s \in \mathbb{N}$, the bot receives a text instruction I_s and metadata M_s from the simulator. The metadata contains information like the current RGB camera frame, the pose of the bot, etc. Our bot is then given a maximum of K timesteps in the simulator to execute batches of actions to accomplish the task provided in I_s before having to receive the next instruction I_{s+1} from the user.

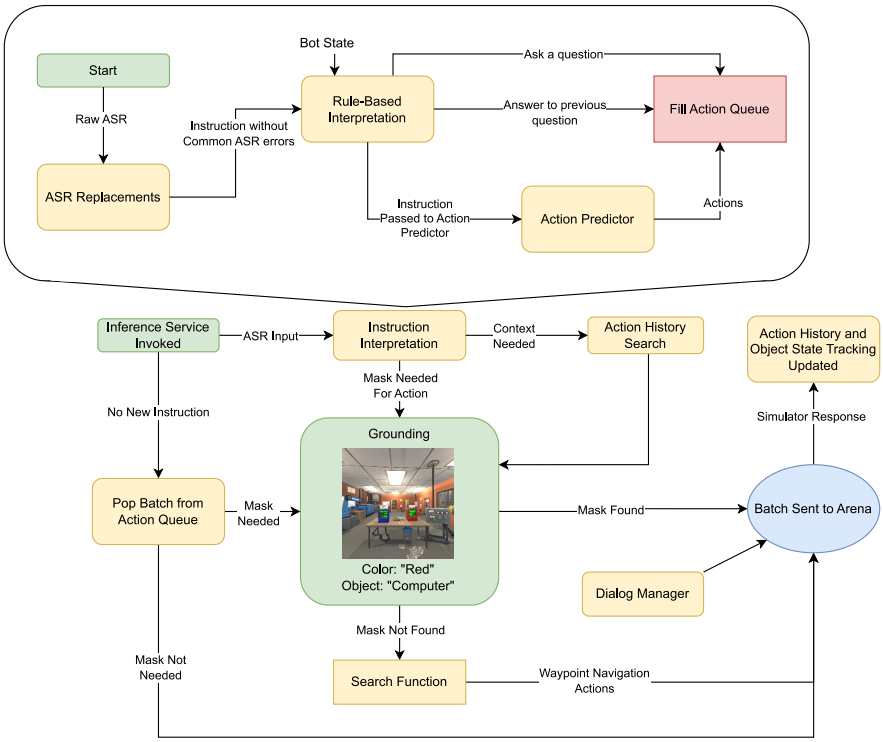

Figure 1: **An overview of the BEAST architecture pipeline.** An instruction interpretation module fills the bot's action queue with actions to execute. Batches popped from the queue and are grounded with their corresponding vision mask. Secondary systems like the action history sub-system, find functionality, dialog manager, and object state tracking support the inference pipeline in its execution.

A batch of actions B_k at time $k \leq K$ is defined as the sequence $\{a_1, a_2, ...\}_k$. The simulator will attempt to execute the actions in B_k sequentially and return a status F_{k+1} of which action failed to be executed (if any) and metadata M_{k+1} from the bot's new perspective. Given (F_{k+1}, M_{k+1}), our bot will predict the next batch of actions to execute B_{k+1}. This sensor input and batch execution cycle continues until either K batches are executed by the bot, or the bot prematurely asks the user for the next instruction I_{s+1}. Overall, the sequence of interactions between the bot and the simulator for the user instruction given at step s looks like $\{(I_s, M_s), B_1, (F_2, M_2), B_2, ..., (F_{k\leq K}, M_{k\leq K}), B_{k\leq K}\}_s$. We refer readers to the report by Gao *et al.* for details about the action space of the bot [3].

3.3 Control Structure and High-level Bot Features

ASR Processing Our system consists of several sub-modules to parse user instructions, predict actions and segment images, track and update our bot's belief of the world, generate varied dialog responses, and several other secondary systems to support action execution. Given a new instruction I_s from the user, we first replace common automatic speech recognition (ASR) error sub-strings within the sentence with their expected text and confirm if the instruction is in-domain to the simulator. For example, common ASR errors for "bowl" included "ball," "boat," "bull," and "boat." This ASR processing step is crucial to ensure the bot predicts correct actions downstream in the pipeline, and to avoid performing actions for out-of-domain user utterances.

If we detect that the utterance is truncated (i.e. ending with "a", "an", "the", "in" or "by") we ask the user to repeat or rephrase the instruction. This logic handles cases where an instruction may have been cut off without the user's knowledge (an example of such a truncated instruction is "pick up the").

We would also like users to focus on game-related instructions and dialog. In other words, the robot must refuse to assist the user with, say, medical advice, or questions about the weather. It must also reject any instructions that contain elements of violence or ask the robot to harm itself.

In order to classify instructions as OOD (out-of-domain, we use the following logic:

- We first define a few lists:
  - `harm_words` consists of phrases that pertain to self-harm or violence (e.g. "stab", "cut yourself")
  - `ood_words` consists of phrases that are clearly out-of-domain because they fall into a general unrelated category of words (such as medical, legal, travel, weather, family, etc.)
  - `action_domain` consists of phrases that we expect to see in instructions because they are related to the game's actions.
  - `object_domain` consists of phrases that we expect to see in instructions because they are related to the game's object types.
- If an instruction contains either a `harm_word` or an `ood_word`, it is marked as an out-of-domain (or harmful) instruction, and the bot responds by saying it cannot help the user with that particular instruction.
- Otherwise, if the instruction contains an `action_domain` phrase or an `object_domain` phrase, it is marked as in-domain.
- We would like our bot to err on the side of caution; if neither of the above conditions are met, the instruction is marked as out-of-domain anyway and the user is asked to rephrase their instruction.

Instruction Intent Classification and Action Prediction In-domain utterances are then classified into one of three possible types of dialog acts - (1) a yes or no answer to a boolean question asked by the bot, (2) an answer to a more complex question asked by the bot, or (3) any other user utterance. Depending on the classified dialog act type and the user utterance itself, we update the state of the bot (e.g. to begin or stop searching for an object within the current room, track which color object we are looking for, reset the status of the dialog manager, etc.) We pass utterances that are not classified as boolean answers through the action prediction model to populate our bot's action queue with actions to execute sequentially. Utterances that correspond to explicit navigation instructions that do not interact with an object (e.g. "move forward", "turn left", "go to the breakroom", etc.) are passed through a rule-based action predictor. All remaining instructions are passed through our transformer-based action prediction model.

Boolean responses (such as "yes", "no", "nope", or "correct", identified in a rule-based manner) directly update the bot's state. The actions predicted by the model(s) for the other instruction types will also populate the bot's action queue.

The above features are collated into the instruction interpretation module, described in Fig. 1.

Vision Processing and Grounding The rest of the bot pipeline attempts to ground the actions in the queue to be executed with camera images and perform them in the simulator. Camera images provided in the metadata after executing each batch are passed through a MaskRCNN instance segmentation model to predict vision masks of detected objects in the current point of view of the bot. We then pop a batch B_k of executable actions from the action queue. B_k is guaranteed to hold at most one object-interaction action type that would require grounding with a mask of the corresponding object to interact with from the current image frame. Examples of object interaction actions that require an explicit mask include picking up, going to, toggling, or placing something on an object. Our "mask-choosing algorithm" will select a segmentation mask of the requested object that is most likely being referred to within the action.

Find Functionality However, there will be cases that a mask of the requested object in the action was not predicted within the current RGB frame. In these cases, our bot will make use of its find functionality module to look for the requested object in the current room. Specifically, the bot first asks the user a boolean question to request permission to search for the object (e.g. "Would you like me to look for a bowl? Please answer yes or no.") If the user responds affirmatively, (e.g. "yes," "yeah," "sure,") the bot will navigate to each predefined waypoint in the current room, look around, and attempt to identify an object mask for the original action. Else, if the bot is unable to find the object mask even after searching the current room, it will prompt the user for the next instruction.

Action History The user is also allowed to explicitly refer to an object the bot has interacted with in the past to avoid redundancies in the find functionality. An action history module keeps track of information after each batch is executed like bot pose, action taken, object interacted with, and more. When the user's instruction begins with the phrase, "go back to the", our bot searches its action history module for the last object requested, navigates to the pose where the object was seen, and goes to the object from the detected mask. This feature also works for objects interacted with in another room, which saves time for the user from having to explicitly ask the bot to find the object in the other room. Our bot notifies the user that they can invoke this feature with a dialog action tip, "Just a tip, if you want me to go back to an object I already interacted with, you can tell me to go back to it."

Dialog Manager and Error Handling There are several other situations where our bot needs to communicate with the user, with a few examples provided earlier. The role of the dialog manager in our system is to generate varied dialog actions given a dialog act type (i.e. acknowledgement to user, boolean question, or other question) and tags (e.g. object name, room name, action type, etc.) to fill into the randomly generated dialog act. This dialog manager also works with an error handling module which attempts to respond to simulator errors F_k after each executed batch. If for example our bot is already holding an object and the user asks to pick up another object, the bot may say, "Hmm, I'm already holding a bowl, so I cannot pick up the apple." There are also examples of errors that do not require intervention from the user. For example, a *TargetOutOfRange* error is thrown when the bot is attempting to execute an object interaction action on an object that is too far from the bot. In this case, the error handling module directly navigates to the object before reattempting the previously failed action so as to minimize unnecessary interactions with the user. These aspects of our system are discussed in more detail in sections 3.7.1 and 3.7.4

Updating Bot Belief Finally, after executing a batch of actions in the simulator and receiving the next simulator metadata, the bot updates its action history of successfully executed actions, inventory of viewed objects from the camera field of view, object state tracking of which receptacles have been opened or closed, and the holding status of whether the bot has picked up or placed down an object.

The following sections describe our sub-modules in more depth.

3.4 Action Prediction Model

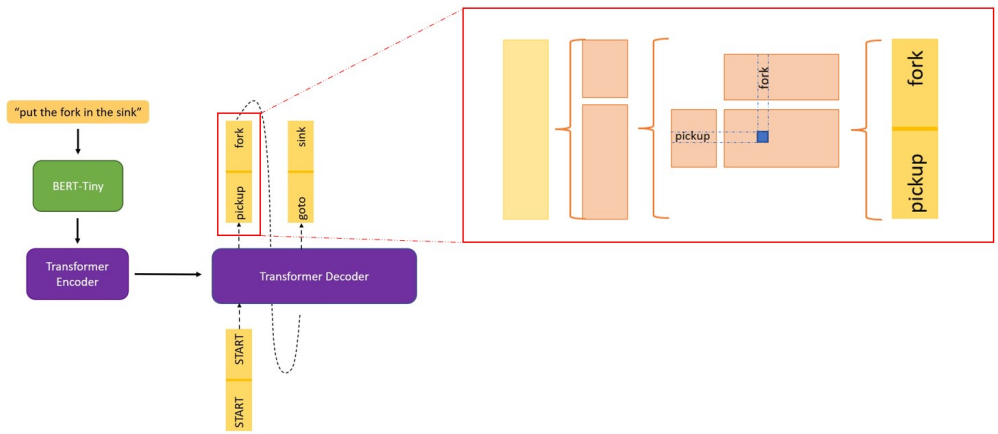

Figure 2: **Action Prediction Pipeline**

Architecture Our model is based on an encoder-decoder structure. We first encode a single natural language instruction, i, using a pretrained BERT-tiny model, creating a representation, e_i, with a size of 128 and a length matching the length of the instruction from the last hidden layer. We then pass this encoded information as the first hidden layer to a transformer encoder with 2 layers, 2 heads, and a hidden size of 128. The output of this encoder, enc_{out}, along with any actions already performed to execute this instruction $a'_{0:t-1}$, are passed into a transformer decoder (2 heads, 2 layers, hidden size

128). The encoder output is passed in as the overall representation of the encoded information, while any actions already predicted/executed are passed in as already produced outputs for those timesteps, outputting a representation of the next action and object pair, a_t.

In this representational space, we use a one-hot encoding to represent our 14 actions and 114 objects in the 128 dimensional space a_t is in. Previously predicted action-object pairs are passed in as their proper one-hot encoded forms (i.e. if the output for $t = 1$ is a distribution over the actions and objects and the highest probability action and object are chosen, then the chosen action and object are one-hot encoded again $a'_{0:t-1}$ rather than using the output distribution $a_{0:t-1}$). This allows the test-time data to resemble the teacher forced distribution of training as well as prevents any nonsequiturs in the action sequence.

For each action object representation a_t, we pass the representation through two linear layers L_A, L_O, one for the action space and one for the object space. We compute a matrix outer product of both of these resulting vectors act_t, obj_t, creating a 2D distribution of actions by objects $c \in \mathbb{R}^{14x114}$. We multiply this by a mask, $M \in \mathbb{R}^{14x114}$ that removes impossible combinations (i.e. certain actions cannot be applied to certain objects, like pouring a table) which we compute using object affordances that are provided in the Arena platform [3]. We then take the softmax over the entire space to pick the combined action object pair c_{ij}, which correlates to action i and object j.

Training Our model is trained for 5 epochs with a learning rate of 0.0001 with a multiplicative scheduler and an Adam optimizer on the cross entropy loss of teacher forced sequences on two datasets: the provided trajectory dataset and our own augmented dataset. We did further training for 5 epochs concentrating only on augmented data examples that our model got incorrect as a form of fine-tuning.

3.4.1 Augmented Data

One of the motivations behind using such a simple architecture was the ease of creating augmented data. A more complex grounded architecture requiring visual inputs would require collecting such data to augment the provided training dataset, which is extremely time-consuming and expensive to collect. Furthermore, only a quarter of the instructions in the dataset were natural utterances– the majority being rule based creations, which are fairly simple and unvaried. This dataset also overrepresented certain objects and actions, while never using others. Thus we created a text-only augmented dataset to address areas where the original data was lacking.

Firstly, the augmented dataset included more natural speech constructions and filler words (e.g. a request might start with "uh" or "can you"). It also refers to objects by more varied references, including synonyms for words as well as descriptions like colors or relative positions (e.g. a slice of bread might be referred to as "the brown piece of bread" or "the bread on the plate"). These references might also include navigational information, such as "the bread to your left", which conveys implicit navigational instruction.

In addition to these changes and representing the full action and object space, our augmented data also represented cases like compound actions (where two different instructions are joined together with a conjunction, e.g. do X and then do Y) and complex instruction words that implicitly describe multi-step processes (e.g. bringing object X from location Y, requires going to Y, finding X, picking X up, and returning to the current location). There were also instructions where the action/object is unclear (e.g. in "pick it up" it is unclear what "it" refers to without some sort of history, whereas in "put it down" the objected upon which we should be placing what we're holding is not clear, presumably because it is visible or obvious). In such cases, we should be able predict a placeholder action or object (NONE) to fill in based on action history and visual information, external to our action prediction model.

All of these instructions were created based off of a template:

$$(intro)\ VERB\ (desc) OBJ1\ (sub\ clause)\ ((desc)\ OBJ2\ (sub\ clause))\ (end)$$

where $intro$ was randomly chosen from natural starts to instructions such as "please" or "do you think you could" and end is randomly chosen from natural endings like "thanks", $VERB$ is a synonym for the action, $OBJ1$ and $OBJ2$ are synonyms for objects, where either could be the direct object of the $VERB$ depending on its properties, and $desc$ and $sub\ clause$ are optional descriptions that may include information such as color, location, size, etc. There were some variations to account for

pronouns being used as objects (e.g. "pick up the plate" sounds grammatical while "pick up it" does not), certain ditransitive verbs (e.g. place technically requires two objects: what we're placing and where we place it; these are usually differentiated by position and a preposition), additional adverbs to sound natural (e.g. we put objects "away" and "in" receptacles that close and "down" or "on" receptacles that do not), etc.

In total, we created 29946 augmented instructions to train on and a further 28000 for validation. As our environment is explicitly defined and thus we need not have explicit zero-shot reasoning abilities, all actions and objects were represented in both sets. The validation set was further segmented into portions based on what aspect they were testing (149 instructions had unknown verbs that required predicting an *UNSURE* action to ask for clarification, 725 instructions involving predicting *NONE* objects, 725 instructions predicting navigational actions, 2450 instructions testing simple compound actions, 18450 instructions for complex multi-step actions such as turning on the light and changing the colors of objects, and 5847 testing a range of object and action combinations).

3.5 Vision Model and Grounding

To execute a command such as `toggle computer`, we need to replace the object type `computer` with an object mask that the Arena API can then resolve to an object in the environment. This requires us to identify instances of objects such as computers, and then identify which of these candidate objects should be chosen to execute the current command. We refer to the first component as the vision model, and to the second component as the grounding model. Grounding is straightforward when only a single instance of an object type (e.g. an "embiggenator") has been observed, but requires disambiguation when the vision model proposes multiple candidate masks for the same object type, and further exploration of the environment when the vision model has not yet discovered any instances of the required object type. In some cases (*pick up one of the bowls*), the agent may see multiple candidates (e.g. multiple bowls on the table), but any one of them can be chosen. In other cases (*toggle the red computer*, *pick up the action figure printer cartridge*), grounding is more complex, since not all candidates of the appropriate object type (computer or printer cartridge) can be referred to by the referring expression used in the instruction. In order to ground objects in our system in the case of multiple candidate masks we select the largest one in an attempt to interact with the object closest to the user. We also filter mask candidates by color 3.5.1 and parse the requested color from the user instruction. Although our current grounding model is still very simple, methods similar to [9] could be used to create datasets for phrase grounding, enabling us to ground referring expressions to appropriate masks in Arena screenshots. The vision model used in our current system is the MaskRCNN used by the baseline Arena model [3]. We also experimented with using a MaskRCNN with a ResNet-101 backbone and found that this overfits during test time to sudden changes in FOV induced by the simulator.

3.5.1 Color Detection

To interact with certain objects in some games, identifying their specific color is necessary. For instance, a red computer might need to be interacted with while a blue computer beside it should be ignored. In order to determine the color of an object, we use a method where we analyze each pixel within the object's mask. This involves converting the RGB values of each pixel to HSV, and tuning thresholds to identify whether the pixel belongs to the red, green, or blue color range. By counting the pixels that belong to each color, we can identify which color is the most prominent in the mask, and therefore detect the color of the object. This capability allows the robot to understand improve grounding instructions that involve color, without having to activate a highlight function. Currently, we have only implemented the detection of red, green, and blue colors, which are required for the current game missions. However, we plan to add HSV thresholds for other colors that may appear in the game in the future.

3.6 Executing Actions

The bot's action queue stores a prioritized sequence of actions to be executed in the simulator. The queue has functionality to push an individual action or a list of actions to its front or back. We can also pop an individual action from the front of the queue. To pop an executable batch of actions from the queue for the current simulator step, we iteratively pop actions from the front of the queue until the batch of actions meets a specific set of conditions.

More specifically, batches of actions that can be popped from the queue follow one of three types:

1. {LIGHT_DIALOG_ACT, not LIGHT_DIALOG_ACT}
2. {not LIGHT_DIALOG_ACT}
3. {ANY_ACTION, LOOK_AROUND},

where LIGHT_DIALOG_ACT is a LightWeightDialog action, described in more detail in 3.7.1. These conditions ensure that every batch of actions that will change the state of the world (e.g. picking up an object, navigating somewhere, etc.) will be segregated into separate batches and act on the latest simulator metadata. LOOK_AROUND actions do not change the state of the world and can be batched in with other actions.

3.7 Secondary Support Modules

The following sections describe the functionality of secondary sub-modules in our architecture like the dialog manager, action history, find functionality in more depth

3.7.1 Dialog Manager

In addition to following user instructions, an essential aspect of our bot is its ability to converse with the user. We want our bot to be able to ask clarifying questions, acknowledge user instructions, and let users know if the bot is facing specific issues that it needs help with. Having good dialog strategies is crucial to improving the user experience. From our observations of user ratings and conversations, we found that adding each of the components described in this section made the user experience significantly better. We use five types of dialog acts:

- ***Acknowledgements*** These dialog acts are used to respond to the user after we finish executing actions. Acknowledgments may be positive or negative; positive acknowledgments are used when we successfully execute actions and allow the agent to prompt the user for the next instruction. For a few selected actions, we also convey to the user that we successfully completed the action. For instance, at the end of a GoTo action, we might say "I'm here!", and after we pick up an object, we might say "Got it!". Negative acknowledgments, on the other hand, are used when something goes wrong (e.g. "Oops, sorry. Looks like that didn't work. What should I do instead?"). In cases where Arena returns an error, the bot responds with utterances that are specific to that error, so that the user may use that information for their next instruction (e.g. "Sorry, I can't get there. I think there's something blocking my way."). This feature is further discussed in 3.7.4.
- ***Fillers*** We use Arena's LightWeightDialog utility to speak to the user while actions are being executed. Going to another room often takes a few seconds, and to make this time period more engaging for the user, we generate a Filler to say utterances like "Cool! Heading there now." Fillers are also quite useful when we search for an object; we are able to use them to tell the user we're (still) in the process of searching for an object.
- ***Tips/Suggestions*** Our bot provides suggestions to the user to notify them about its capabilities. For example, the bot begins a mission by telling the user, "Just a heads up, I can only hear you when the blue bar at the bottom of your screen is showing. Just say Alexa to get my attention!" We also inform the user about invoking the action history feature of the bot by saying, "Just a tip, if you want me to go back to an object I already interacted with, you can tell me to go back to it." We provide an randomly generated example with a key word object like, "For example, you can say go back to the cake. What should I do next?"
- ***Questions*** We leverage our dialog capability to enable the agent to ask the user clarifying questions whenever it is confused about what to do. Questions are asked whenever there is confusion about:
  - what the bot is holding (when we're not holding anything but are asked to Place/Pour, or when we're already holding an object but are asked to PickUp)
  - when we cannot perform Open and Close actions (e.g. when we are asked to close an object that we think is already closed)
  - when we've never seen an object, or the object is outside our view (not in FOV, or in another room)

- **Yes/No Questions** We are also able to ask the user for confirmation via yes/no questions. For example, we confirm with the user whether we should start looking for an object or not. Depending on whether the user gives the bot the go-ahead, we decide whether or not to proceed with the originally intended actions.

Intonation in our agent's utterances: To make our agent's utterances sound more natural and less formal, we make use of Speechcons [1]. Speechcons are special words and phrases that Alexa pronounces more expressively. This contributes to a more engaging user experience overall.

3.7.2 Action History

As a user, we would like the robot to remember information about what it did previously and objects it interacted with earlier. We would like to then be able to refer to those objects later in the mission. For instance, if we give the agent the following instructions: "pick up the mug", "go to the desk", "put it down", we want the bot to understand that the "it" in the third instruction refers to the mug, irrespective of other objects it may be seeing, or previous actions it may have taken. Another scenario where this is useful as a user is when we want to go back to an object we've already interacted with earlier (e.g. "go back to the freezer").

In order to be able to accomplish such things, we maintain an action history. For every batch of actions we send to Arena, we maintain the following information:

- timestep t
- pose of the bot; consists of position (x, y, z) and rotation (x, y, z, w)
- instruction corresponding to the batch of actions
- the room the agent is in
- objects the agent interacted with, or navigated to (including objects the bot may have been holding)
- actions successfully executed in the batch
- object the agent is holding, if any

We then utilize this history to complement the action model and further improve the performance of our bot in the following ways:

- ***Handling unclear objects*** For instructions where the object is unclear, the action model outputs an action with a *None* object (i.e. an action of the form *(ACTION, None)*). For example, the action model outputs *(Toggle, None)* for the instruction "turn it on". In such cases, we replace None with the last-interacted-with object in our action history. This enables our bot to successfully handle sequences such as "go to the time machine" followed by "turn it on". When checking for such objects in our history, we ensure that we use only objects that are compatible with the action we predicted- this means that *(Toggle, Time Machine)* is acceptable whereas *(Toggle, Mug)* is not.
- ***Handling objects we've already interacted with*** We're able to use our pose information to go back to objects we've interacted with. For example, given the instruction "go back to the computer", the agent returns to the pose it was in when it last interacted with a computer. We also let the user know through a dialog act that they are able to ask us to go back to an object if they've already interacted with it earlier in the game. This also allows us (i.e. the agent-user combo) to avoid redundant find operations.

3.7.3 Find Functionality

Agents need to be able to explore and search their environment for unseen objects as requested by the user. However, exploring the world is a time-consuming process that requires a balance between exploiting the bot's current belief about the environment, and choosing where to look deeper.

Whenever a vision mask is required for an object interaction action type, no mask was found for said object, and the bot cannot find the object in its action history, we invoke our find function in the current room. Specifically, we first use the dialog manager to request permission from the user to

begin searching in the current room. That way, if the user already has seen where the desired object is, they can direct the robot to it while skipping the searching function. In the event the user wants the bot to search for the object and responds positively, we access the available static viewpoints in the current room and order them in a queue by distance away from the bot's current position. Thus the bot is expected to navigate greedily by choosing viewpoints that are closest to each other and saves time in the navigation policy.

While the mask for the requested object (and color if provided) has still not been generated, we pop the closest viewpoint from the queue, navigate to it, perform a look around action, and reattempt the original object interaction action. At every other waypoint, our bot will also engage the user by saying a randomly generated variant of the phrase, "I'm still looking." If the object interaction action still fails to be grounded with a corresponding mask after navigating to each viewpoint in the current room and the waypoint queue is empty, the bot will tell the user that it did not find the object (e.g. "I haven't seen any bowls yet,") and ask for assistance in finding it (e.g. "If you've seen one, could you help me navigate to it?").

3.7.4 Error Handling

The Arena platform returns information about the status of the actions the bot attempts to execute, and this information includes errors (if any). One specific kind of error we deal with explicitly in our model is the *TargetOutOfRange* error - this occurs when the bot attempts to interact with an object that's too far away (i.e. an object that exceeds the maximum allowable range for interaction). The thresholds for valid distances are defined in the Arena platform. In cases where we encounter *TargetOutOfRange* errors, instead of asking the user for help, we directly reattempt the original action(s) after going to the object that was out of range. In order to ensure we still see the object, we perform a Look Around action, use an updated mask to go to the object that was out of range, and re-attempt the original action(s).

We deal with other errors using dialog acts. For example, we would say "We need to plug it in first" in case we get an *ObjectUnpowered* error, or "I can't get there. I think there's something blocking my way." when we encounter an *UnsupportedNavigation* error. These helpful suggestions enable the user to understand why an error might be thrown, and sometimes even help them understand what is expected of them in the mission. We have a variety of utterances for each type of error to avoid repetition and improve the user experience.

3.7.5 Object State Belief Tracking

We use the status information we get back from the simulator after executing a batch of actions to update our belief about objects and their states. Our action history is updated with the new batch of actions (we only track actions that succeeded, so if we had actions $[a_1, a_2, a_3]$ and a_3 failed, we would only track information corresponding to a_1 and a_2). This information includes whether or not receptacles are open, and what object the bot is holding (if any). It also includes all the information in the action history as discussed in 3.7.2.

4 Experimental setup and results

4.1 Action Prediction Model Evaluation

We evaluated the action model on validation datasets for both the trajectory and augmented datasets in Table 1. For the augmented dataset, two smaller splits on compound (instruction A + "and" + instruction B) and complex (actions with implicit steps, like microwaving an object which requires navigating to the microwave, opening and closing it, placing the object inside, etc) actions.

Naturally the accuracies are higher when evaluated using teacher forcing (Parallel Evaluation) rather than when error is allowed to propagate forward and the entire sequence needs to be corrected (Full Sequence Evaluation). We generally only see small drops from individual action and object accuracy to paired accuracy where both the action and object must be correct, indicating that generally when the model predicts the incorrect output, it is incorrect for both the action and object. We also see that object accuracies tend to be lower, likely because of the large difference in the output spaces (there are only 12 actions in our output space whereas there are 114 objects). We also see lower overall

Split	Action Accuracy	Object Accuracy	Paired Accuracy
Trajectory Dataset			
Parallel Evaluation	96.1	94.3	93.7
Full Sequence Evaluation	87.4	76.9	76.1
Augmented Dataset			
Parallel Evaluation	98.7	98.1	97.3
Full Sequence Evaluation	90.0	86.8	86.3
Compound Dataset (Augmented)			
Parallel Evaluation	97.2	96.8	95.0
Full Sequence Evaluation	56.7	51.0	49.8
Complex Dataset (Augmented)			
Parallel Evaluation	87.0	83.5	82.6
Full Sequence Evaluation	53.8	43.3	43.2

Table 1: **Evaluations** on the trajectory and augmented datasets with both Parallel Evaulation (a teacher-forcing paradigm where the correct answer is fed in for each timestep and each timestep is evaluated independently) and Full Sequence Evaluation (a fully auto-regressive paradigm where the the prediction for each time step is passed in for the next one and every timestep must be correct for the entire sequence to be considered correct)

	Instruction	Model Output
1a	stop put it down	[place : NONE]
2a	no go to the other computer	[goto : computer]
2b	ugh can you put down the apple and pick up the action figure instead	([place : None], [pickup : action figure])
3a	you should turn the power on first	([goto : fuse box], [open : fuse box], [toggle : lever], [close : fuse box])
3b	do you think you can put this donut in the break room fridge and come back	([pickup : donut], [goto : break room], [goto : fridge], [open : fridge], [place : fridge], [close : fridge], [goto : NONE ROOM])
4	pick it up	[pickup : printer cartridge]
5	okay grab the peanut butter to your left and take it to the color changer in the quantum lab	([rotate : left + 45], [pickup : peanut butter], [goto : lab1], [goto : color changer], [place : color changer], [toggle : button])
6a	let's carrot this computer	([pickup : computer], [goto : carrot machine], [place : carrot machine], [toggle : carrot machine])
6b	let's modify this computer	[toggle : computer]
7a	pick up the donut not the apple okay?	[pickup : donut]
7b	don't pick up the apple again and grab the donut instead	[pickup : apple]
7c	don't pick up the apple this time and grab the donut instead	([goto : time machine], [open : time machine], [pickup : donut], [close: time machine])

Table 2: **Qualitative Analysis** of instructions in the augmented dataset and model outputs

accuracies for the compound and complex instructions which tend to be longer and more complicated, giving the model more opportunities to make a mistake.

We also provide some qualitative examples that showcase the strengths and weaknesses of our model in Table 2. *1a*, *2a*, and *2b* show the model's ability to handle various **natural utterances and interjections**. *2a* and *2b* also show how abstracting to the level of symbolic representations can remove much of the complexity of correcting actions and that our model is able to handle complicated instructions of that nature. *3a* and *3b* show that our model is able to **effectively execute complex actions** like turning on the power and putting objects away that include various implicit steps. Since we predict the full sequence of required actions, we may predict unnecessary actions given the actual state of the robot or the environment. For example, the robot might already be holding the donut

in *3b* or the fridge might have been left open previously. Our overall system has checks in place to handle such state differences.

On the other hand, *4* shows a single weak case where our model is unable to learn a simple instruction requiring outputting NONE, instead outputting a printer cartridge, likely due to some bias in the training data. *5* shows another error where the model, too used to complex actions in which going to the color changer is immediately followed by using the color changer to recolor an object, automatically adds in a button toggling step, despite the fact that we have not received a goal color. *6a* shows the ability of the model to **learn new actions** (using a carrot machine was not in the training or augmented data), whereas *6b* shows a case where it would be useful for our model to output an UNSURE action and ask for clarification–this instruction could be about changing the color, putting the computer in a time machine, inserting a floppy disk into the computer, or, as the model predicts, simply toggling the computer. To be certain the appropriate step would be to ask the user to explain what is meant. Finally, *7a*, *7b*, and *7c* show the weakness of the model when **multiple objects** are included in an instruction with the intention of only acting on one. The model tends to prefer the first object introduced (except in cases like action figure cartridge where the phrase action figure, despite being an object, is used as a description of a printer cartridge) and can mistakenly interpret other parts of the instruction as additional goals instead of corrections.

4.2 Vision Model Evaluation

In order to evaluate the effectiveness of an instance segmentation model for the Alexa Arena, we evaluated our models by mimicking the policy used to act on objects within the Arena platform on the instance segmentation dataset provided by the Arena. To perform an action on an object within the Arena you must provide the desired action type and a bitmask over the desired object. To determine which object to act on, Arena calculates an IoU score with each object in the ground truth segmentation and the bit mask provided by the model. Arena then performs the action type with the object that had the highest IoU score of each of the masks [3]. This policy of object selection can lead to some artifacts in the evaluation. For example, a model may produce a mask covering most of the surface of a counter top. However, if there is an object on the counter, like an apple, that is completely covered by the mask, Arena will perform the action on the apple. An example of this problem can be seen in the confusion matrix in Table 6, where a significant portion of predictions for a Counter Top are actually evaluated as an Apple due to this artifact. Standard COCO metrics will calculate the IoU of a detection with the ground truth and determine a positive detection if the IoU is above a certain threshold, which does not take into account any overlap with a nearby small object.

	mAR	mAP
Arena MaskRCNN	36.05	67.29

Table 3: **Vision model results** using Arena object selection policy, averaged across all classes.

4.3 Offline Evaluation

We also evaluated a modified version of our bot on the $1,149$ missions of the validation split of the offline environment presented by Gao *et al.* to get a proxy quantitative success rate for real interactions with customers [3]. Compared with our original interactive bot, the offline bot no longer performs out-of-domain parsing, it cannot go back to a physical pose of where an object was interacted from previously, and does not ask clarifying questions. We found that even this limited bot performed quite well on the offline data, nearly matching the performance of an inherently multimodal model with our modular set-up, as seen in Table 4. We saw the strongest performance on tasks like scanning and toggling, and extremely weak performance on tasks including cleaning and breaking. About half of these failures were due to the bot's limited ability to find certain objects in a setting, leaving it unable to complete the task as the required object could not be found.

5 Related Works

Natural Language for embodied task completion has long been studied within robotics [2, 7, 11, 5]. As such, there have emerged many sub-problems in getting agents to effectively interpret natural

Mission Type	BEAST (Ours)	NS w/o QA*	NS w/ QA*	VL w/o QA*	VL w/ QA*
breakObject	0.00	0.00	0.00	21.11	41.11
clean&deliver	0.00	12.64	13.79	13.79	19.10
color&deliver	0.00	0.00	0.00	0.00	0.00
fill&deliver	4.17	14.58	18.75	10.41	22.91
freeze&deliver	8.33	33.33	25.00	0.00	8.33
heat&deliver	5.13	5.13	5.13	10.25	28.20
insertInDevice	29.38	14.12	14.69	14.68	20.90
pickup&deliver	8.42	9.47	12.63	15.43	27.36
pourContainer	34.19	14.53	16.24	16.23	30.76
repair&deliver	12.96	11.11	12.96	9.25	29.62
scanObject	52.25	41.44	41.44	37.83	56.75
toggleDevice	66.67	57.14	56.19	81.90	81.90
Overall	22.37	18.19	19.32	22.80	34.20

* These results are presented in [3].

Table 4: **Mission Success Rate (↑) of offline methods by mission type.**

language such as phrase grounding to objects [9, 4], grounding to physical actions [6, 10], recovering from error using user input [5], and long horizon task planning [10, 11]. A key difficulty within long horizon planning is the ability to effectively use low level navigation actions (turn left, move forward, etc.) to reach a desired object [8, 11, 2]. Arena allows us to abstract away these low level navigation with GoTo actions that allow the agent to navigate between different rooms. Because of this abstraction we are able predict high level actions effectively and decouple our language and vision modules.

6 Conclusions and Future/Ongoing Work

Our work shows that within in a simulated environment where navigational instructions are abstracted away, it is possible to engage in task-oriented dialog and execute instructions with a simple transformer encoder decoder and modular architecture built up around it to handle dialog and visual information. This modularity allows us to improve individual model components without involving other components (e.g. we can improve our symbolic action prediction with a fully artificial dataset on only text, without needing to play through various games and align text with images).

We are actively working on adding features to mitigate potential failures in the current pipeline. Firstly, we plan on adding to the augmented dataset to allow an even broader range of instructions that our model can handle (such as negative instructions that tell a model to not do something), as well as further training of the action prediction model, focused on reinforcement learning based paradigms where the full sequence accuracy can be improved instead of a teacher-forced objective. (Although we had hoped to use customer utterances for the purposes of training our models, these have so far not yet been shown to be more complex or diverse than what is already reflected in our dataset). Second, we would like to use a richer grounding model that can incorporate more attributes. We are also working on employing the highlighting function to clarify confusions with the user in an interactive dialog, building a more accurate model of the world by utilizing depth sensor information (allowing us to get a better grasp of object permanence across time and changes in agent position and pose), and on better exploiting the nature of unique objects in the simulator to refer to previously seen interactable devices.

Highlighting We are working on incorporating Arena's highlight functionality to notify the user when the bot is unsure of which object to ground its action with. Specifically, if the bot identifies multiple of the same object type with similar characteristics to what was requested by the user, we wish to highlight each object one at a time and ask the user which object is correct, so that the object that the user responds to affirmatively can be acted upon. This feature would mitigate failures of the robot choosing the incorrect object (or relying on an impoverished grounding model) by asking the user for clarification.

We also wish to use the highlighting functionality to confirm whether an object is the correct instance when the vision model has low confidence in its prediction. However, we noticed that each individual highlighting action took several seconds to be physically simulated in the environment. In order to minimize user frustration, the current highlighting function may be too slow to be used except in rare circumstances.

Object Permanence with Depth Information Object permanence is an ability that children learn as early as eight months old and is necessary in embodied AI agents that need the ability to distinguish between objects throughout its environment. We propose to use the depth information from the simulator metadata to estimate positions of detected objects relative to the bot at each time step, which can be transformed into the world coordinates. This estimation will be incorporated into our action history module to track instances of objects and their positions in the world.

Tracking Unique Devices A simple addition to our pipeline will be to take advantage of the assumption that there will be some unique objects in the simulator with only a single instance (eg. color changer, time machine, embiggenator, etc.) As discussed in 3.7.5, our object state tracking module understands the status of receptacles and what the bot is holding. We can also store the last observed positions of unique devices that the user can interact with (even if the agent did not explicitly interact with the device yet) to minimize the searching time of those objects when necessary for a later interaction.

References

[1] Amazon. Speechcons (interjections).

[2] Valts Blukis, Chris Paxton, Dieter Fox, Animesh Garg, and Yoav Artzi. A persistent spatial semantic representation for high-level natural language instruction execution, 2021.

[3] Qiaozi Gao, Govind Thattai, Xiaofeng Gao, Suhaila Shakiah, Shreyas Pansare, Vasu Sharma, Gaurav Sukhatme, Hangjie Shi, Bofei Yang, Desheng Zheng, Lucy Hu, Karthika Arumugam, Shui Hu, Matthew Wen, Dinakar Guthy, Cadence Chung, Rohan Khanna, Osman Ipek, Leslie Ball, Kate Bland, Heather Rocker, Yadunandana Rao, Michael Johnston, Reza Ghanadan, Arindam Mandal, Dilek Hakkani Tur, and Prem Natarajan. Alexa arena: A user-centric interactive platform for embodied ai, 2023.

[4] Sahar Kazemzadeh, Vicente Ordonez, Mark Matten, and Tamara Berg. ReferItGame: Referring to objects in photographs of natural scenes. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 787–798, Doha, Qatar, October 2014. Association for Computational Linguistics.

[5] Cynthia Matuszek, Evan Herbst, Luke Zettlemoyer, and Dieter Fox. *Learning to Parse Natural Language Commands to a Robot Control System*, pages 403–415. Springer International Publishing, Heidelberg, 2013.

[6] Dipendra Misra, Jaeyong Sung, Kevin Lee, and Ashutosh Saxena. Tell me dave: Context-sensitive grounding of natural language to manipulation instructions. *The International Journal of Robotics Research*, 35, 11 2015.

[7] Aishwarya Padmakumar, Jesse Thomason, Ayush Shrivastava, Patrick Lange, Anjali Narayan-Chen, Spandana Gella, Robinson Piramuthu, Gokhan Tur, and Dilek Hakkani-Tur. Teach: Task-driven embodied agents that chat, 2021.

[8] Alexander Pashevich, Cordelia Schmid, and Chen Sun. Episodic transformer for vision-and-language navigation, 2021.

[9] Bryan A. Plummer, Liwei Wang, Chris M. Cervantes, Juan C. Caicedo, Julia Hockenmaier, and Svetlana Lazebnik. Flickr30k entities: Collecting region-to-phrase correspondences for richer image-to-sentence models. *CoRR*, abs/1505.04870, 2015.

[10] Mohit Shridhar, Jesse Thomason, Daniel Gordon, Yonatan Bisk, Winson Han, Roozbeh Mottaghi, Luke Zettlemoyer, and Dieter Fox. ALFRED: A benchmark for interpreting grounded instructions for everyday tasks. *CoRR*, abs/1912.01734, 2019.

[11] Yichi Zhang and Joyce Chai. Hierarchical task learning from language instructions with unified transformers and self-monitoring, 2021.

A Appendix

A.1 Augmented Dataset

Figure 3: **Grounding Failure** An example of where our system would fail would be if the user asked "Turn on the computer furthest away from you" when presented with this situation. Our current system would predict [GoTo: Computer, Toggle: Computer] and our grounding module would choose the largest mask it sees which would be the computer highlighted in red, instead of the computer highlighted in green which the user intended to interact with.

Instructions and Outputs		
Instructions	Correct Sequence	Model Sequence (if different)
Standard		
pick up the trophy	[pickup : trophy]	–
please go to the 3 d printer	[goto : printer]	–
can you find the action figure please	[goto : action figure]	–
None Objects		
put it down now	[place : NONE]	–
why don't you close it	[close : NONE]	–
now please pour it into that	([pickup : NONE], [pour : NONE])	[place : NONE]
Navigation Actions		
turn around please	[rotate : right + 180]	–
do you think you could go to the computer on your left	([rotate : left + 45], [goto : computer])	–
uh go forward a few steps	[move : forward]	–
no actually go backwards	[move : backward]	–
Complex Actions		
please microwave the banana	([pickup : banana], [goto : microwave], [open : microwave], [place : microwave], [close : microwave], [toggle : microwave])	–
let's rewind the broken bowl	([pickup : bowl], [goto : time machine], [open : time machine], [place : time machine], [close : time machine], [toggle : time machine])	–
you should fix the yellow mug	([pickup : mug], [goto : time machine], [open : time machine], [place : time machine], [close : time machine], [toggle : time machine])	–
make the white plate green thanks	([pickup : plate], [goto : color changer], [place : color changer], [toggle : button])	–
Multi-Aspect		
put it down on the table to the right	([turn : right + 45], [place : table])	–
change the color of the blue trophy on the yellow boxes to red	([pickup : trophy], [goto : color changer], [place : color changer], [toggle : button])	–
Compound Actions		
put the banana down on the table and pick up the plate instead	[place : table], [pickup : plate]	–
go to the counter and pick up the coffee mug	([goto : counter], [pickup : mug])	–

Table 5: **Augmented Data Splits:** Various types of instructions in the augmented dataset, as well as the corresponding action sequences

	Gravity Pad	Fridge	Carrot Machine	Radio	Apple	Counter Top
Gravity Pad	0.69	0.15	0.0	0.0	0.0	0
Fridge	0.22	0.60	0.0	0.0	0.05	0.0
Carrot Machine	0.0	0.0	0.55	0.20	0.0	0.0
Radio	0.0	0.0	0.05	0.50	0.0	0.0
Apple	0.0	0.0	0.0	0.0	0.75	0.06
Counter Top	0.0	0.0	0.0	0.0	0.5	0.18

Table 6: **Confusion Matrix** for Arena MaskRCNN where a positive detection is defined by the Arena selection policy

Figure 4: **Grounding Success** An example of where our system would succeed would be if the user asked "Press the blue button" when presented with this situation. Our current system would predict [Toggle: Button] and indicate that we are looking for blue object. The color detector would filter down all vision predictions not predicted as blue and our grounding strategy would choose the mask for the button highlighted in blue.