# Pre-Trained Language Models for Interactive Decision-Making Appendix

**Shuang Li [1]*, Xavier Puig[1], Chris Paxton[2], Yilun Du[1], Clinton Wang[1], Linxi Fan[2], Tao Chen[1], De-An Huang[2], Ekin Akyürek[1], Anima Anandkumar[2,3,†], Jacob Andreas[1,†], Igor Mordatch[4,†], Antonio Torralba[1,†], Yuke Zhu[2,5,†]**

[1]MIT, [2]Nvidia, [3]Caltech, [4]Google Brain, [5]UT Austin
Junior authors are ordered based on contributions and senior authors[†] are ordered alphabetically.

In this appendix, we first show the convolutional encoding in BabyAI in Appendix A. We then describe the environment details in Appendix B and the implementation details of the proposed model in Appendix C. We show the algorithm of interactive evaluation in Section D and the data gathering procedure in Appendix E. The goal predicates used in VirtualHome test subsets are shown in Appendix F. We visualize the attention weights in language models in Appendix G.

## A Convolutional encoding in BabyAI

In the main paper Section 7.1, we explore the role of natural language by investigating two alternative ways of encoding policy inputs in VirtualHome. In this section, we show the third way of encoding policy inputs in BabyAI.

We test a new model, **LID-Conv (Ours)**, that converts environment inputs into *convolutional embeddings*. We pass the $7 \times 7 \times 3$ grid observation in BabyAI to convolutional layers and obtain a $7 \times 7 \times d$ feature map, where $d$ is the feature dimension. We flatten the feature map and get a sequence of features to describe the observation. The rest of the model is the same as LID-*Text (Ours)*. Table 1 shows the results of policies using the *text encoding* and *convolutional encoding*. LID-*Text (Ours)* and LID-*Conv (Ours)* have similar results given enough training data, but LID-*Text (Ours)* is slightly better when there are fewer training data. This conclusion is coincident with the results on VirtualHome.

Different input encoding schemes have only a negligible impact on model performance: the effectiveness of pre-training is not limited to utilizing natural strings, but in fact extends to arbitrary sequential encodings.

## B Environments

We use **BabyAI** [1] and **VirtualHome** [3] to evaluate the proposed method. While both environments feature complex goals, the nature of these goals, as well as the state and action sequences that accomplish them, differ substantially across environments.

### B.1 VirtualHome

VirtualHome is a 3D realistic environment featuring partial observability, large action spaces, and long time horizons. It provides a set of realistic 3D homes and objects that can be manipulated to perform household organization tasks.

*Correspondence to: Shuang Li <lishuang@mit.edu>

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

Table 1: **Success rate of policies trained with *text encoding* vs. *convolutional encoding* on BabyAI.** The text encoding is more sample-efficient, but both models converge to near perfect performance given sufficient training data.

| Tasks | Methods | Number of Demos | | | | |
|---|---|---|---|---|---|---|
| | | **100** | **500** | **1K** | **5K** | **10K** |
| GoToRedBall | LID-Text (Ours) | **93.9** | **99.4** | 99.7 | **100.0** | **100.0** |
| | LID-Conv (Ours) | 92.5 | 98.8 | **100.0** | **100.0** | **100.0** |
| GoToLocal | LID-Text (Ours) | 64.6 | **97.9** | **99.0** | 99.5 | 99.5 |
| | LID-Conv (Ours) | **69.5** | 86.0 | 98.2 | **99.9** | **99.9** |
| PickupLoc | LID-Text (Ours) | **28.7** | **73.4** | **99.0** | **99.6** | 99.8 |
| | LID-Conv (Ours) | 25.0 | 58.8 | 95.1 | **99.6** | **100.0** |
| PutNextLocal | LID-Text (Ours) | 11.1 | **93.0** | **93.2** | **98.9** | **99.9** |
| | LID-Conv (Ours) | **17.9** | 53.6 | 91.3 | 97.7 | 99.5 |

**Goal Space.** For each task, we define the goal as a set of predicates and multiplicities. For example, `Inside(apple, fridge):2; Inside(pancake, fridge):1;` means "put two apples and one pancake inside the fridge". In each task, the initial environment (including initial object locations), the goal predicates, and their orders and multiplicities are randomly sampled. There are 59 different types of predicates in total.

**Observation Space.** The observation in VirtualHome by default is a graph describing a list of objects and their relations in the current partial observation. Each object has an object name, a state, *e.g.* *open, close, clean*, and 3D coordinates.

**Action Space.** Agents can navigate in the environment and interact with objects. To interact with an object, the agent must predict an action name and the index of the interested object, *e.g.* `Open(5)` to opening the object with index (5). The agent can only interact with objects that are in the current observation or execute the navigation actions, such as `Walk(bathroom)`. For some actions, such as `open`, the agent must be close to the object. There are also strict preconditions for actions, *e.g.* the agent must `grab` an object before it can `put` the object on a target position. As a result of these constraints, the subset of actions available to the agent changes at every timestep.

We evaluate the success rates of different methods on VirtualHome. A given episode is scored as successful if the policy completes its entire goal within $T$ steps, where $T = 70$ is the maximum allowed steps of the environment.

### B.2 BabyAI

BabyAI is a 2D grid world environment designed to evaluate instruction following. Different from VirtualHome, the observation in BabyAI by default is a $7 \times 7$ grid describing a partial and local egocentric view of the state of the environment. Each tile in the grid contains at most one object, encoded using 3 integer values: one for the object type, one for the object color, and a state for doors indicating whether it is open, closed or locked. The goals in BabyAI are language instructions, *e.g.* "put the blue key next to the purple ball". BabyAI has 7 actions, *e.g.* "turn left", "pick up", and "drop".

## C More implementation Details of LID in VirtualHome

In Appendix C.1, we provide more details of the model architecture used in the main paper Section 4.1. We then introduce the training detail in Appendix C.2.

### C.1 Model architecture details in VirtualHome

In this section, we provide more details of the policy network we used in VirtualHome. Our policy model consists of three parts, *i.e.* inputs, the pre-trained LM, and outputs. As shown Figure 1, we encode the inputs to the policy—including goal $g$, history $h_t$, and the current partial observation $o_t$—as sequences of embeddings. These embeddings are passed to the LM (using its pre-trained embedding layer $F_\theta$) and used to obtain contextualized token representations. These token representations are averaged to generate a context feature $f_c$, which is then passed to fully-connected layer to predict the

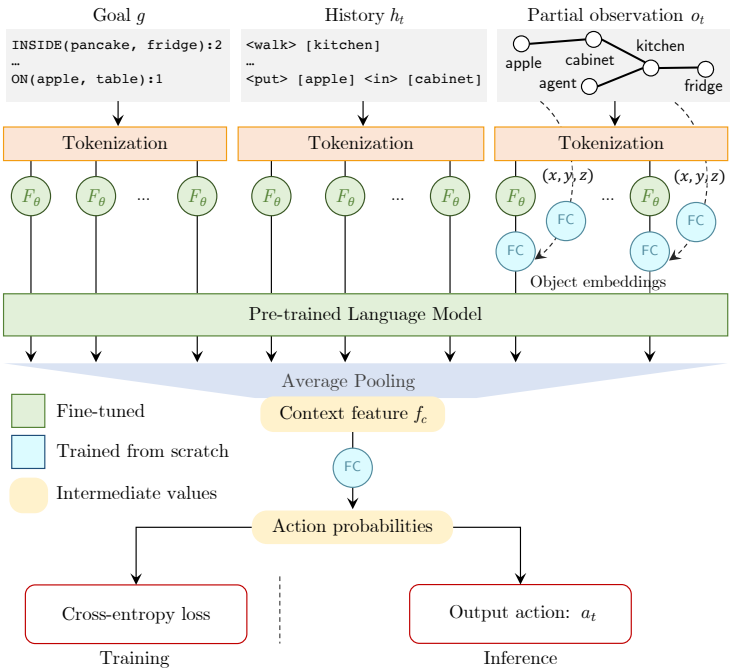


Figure 1: **Policy network in VirtualHome.** The observation, goal, and history are first converted into sequences and then passed through an embedding layer $F_\theta$. The combined sequence is passed through a pre-trained LM, and the output tokens are pooled into a context feature vector for action prediction.

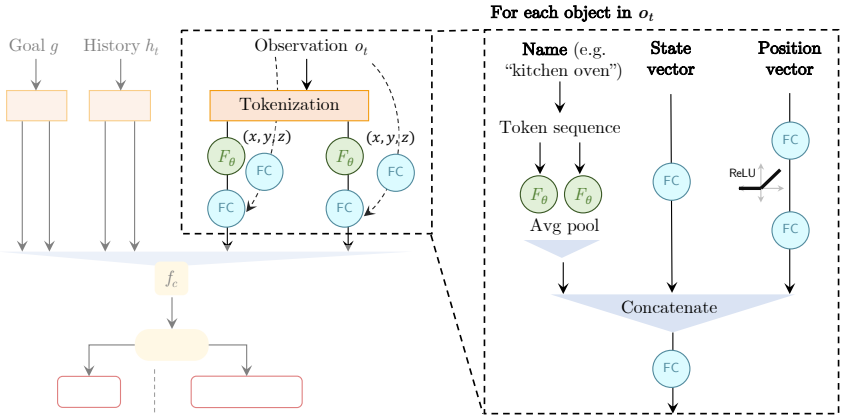


Figure 2: **Object encoding.** In VirtualHome, the partial observation of the environment state can be represented as a list of objects in the agent's view. Each object is represented by a name, a state vector, and position vector. **Object name encoding:** each object's name is an English phrase. We tokenize the phrase, embed the tokens, and average the embeddings. **Object state encoding:** each object is assigned one of six states: "clean", "closed", "off", "on", "open", or "none". This state is represented as a 6-dimensional binary vector and passed through a fully-connected layer. **Object position encoding:** an object's position vector is a 6-dimensional vector containing its world coordinates alongside its displacement to the agent (*i.e.* the difference in their world coordinates). This position vector is passed through two fully-connected layers. These three features are concatenated and passed through a fully-connected layer to obtain the representation of an object in the current observation.

next action $a_t$. The output action in VirtualHome consists of a verb and an object. For brevity, we omit the time subscript $t$ from now on.

In VirtualHome, the partial observation $o$ of the environment state can be represented as a list of objects in the agent's view. We represent each object by its name, *e.g.* "oven", a state description, *e.g.* "open, clean", and position both in the world and relative to the agent. In this part, we provide more details of how **LID-Text (Ours)** encodes the name, state, and position of each object in the observation. Figure 2 shows the model architecture we used to encode the observation.

**Name encoding.** For each object node, we serialize its object name as an English phrase $s^o$. We extract its tokens and features using the tokenizer and the embedding layer of the pre-trained LM, respectively. Since one object name might generate several English tokens using the tokenizer from the pre-trained LM, *e.g.* the tokens of "kitchencabinet" is $[15813, 6607, 16212, 500]$, we take the averaged features of all the tokens in the object name and obtain a "name" feature $f_i^{o,\text{name}}$ for each object node as shown in Figure 2.

**State encoding.** Some objects have a state description, *e.g.* "oven: open, clean". There are six types of object states in the environment: "clean", "closed", "off", "on", "open", and "none". For each object node, we use a binary vector to represent its state. Taking the "oven" as an example, if the oven is open and clean, its state vector would be $[1, 0, 0, 0, 1, 0]$. This state vector is then passed through a fully-connected layer to generate a state feature $f_i^{o,\text{state}}$ of object $o_i$.

**Position encoding.** To encode the position information of each object $o_i$, we take their world coordinates $\{o_{i,x}, o_{i,y}, o_{i,z}\}$ and their spatial distance to the agent $\{a_x, a_y, a_z\}$ to generate a position vector $[o_{i,x}, o_{i,y}, o_{i,z}, o_{i,x} - a_x, o_{i,y} - a_y, o_{i,z} - a_z]$. This position vector is then passed through two fully-connected layers with a ReLU layer in the middle to generate a position feature $f_i^{o,\text{position}}$ of object $o_i$.

The final feature $f_i^o$ of each object node is obtained by passing the concatenation of its name feature $f_i^{o,\text{name}}$, state feature $f_i^{o,\text{state}}$, and position feature $f_i^{o,\text{position}}$ through a fully connected layer. The observation at a single step can be written as a set of features $\{f_1^o, \cdots, f_N^o\}$, where $N$ is the number of objects in the current observation.

### C.2 Training details

Our proposed approach and baselines are trained on Tesla 32GB GPUs. We train every single model on 1 Tesla 32GB GPU. All experiments used the AdamW optimizer with the learning rate of $10^{-5}$. We utilize a standard pre-trained language model, GPT-2, in our experiments. GPT-2 is trained on the Webtext dataset [5] using the Huggingface library [10].

## D Interactive Evaluation

The algorithm for interactive evaluation is shown in Algorithm 1.

**Algorithm 1:** Interactive evaluation

A set of task goals $G$ (each goal has a corresponding initial state);
Load the learned policy $\pi_\phi$;
Successful trajectory count: $n = 0$;
**for** *example=1,* $N_{test}$ **do**
    Sample a goal $g$ and the an initial state;
    **for** $t = 0, T$ **do**
        Sample an action $a_t$ from policy $\pi_\phi(a_t|g, h_t, o_t)$;
        Execute the action $a_t$ and get a new observation $o_{t+1}$;
        **if** *success* **then**
            $n = n + 1$;
            break;
        **end**
    **end**
**end**
success rate: $r = n/N_{\text{test}}$;

# E Data Gathering Details in VirtualHome

In this section, we provide more data gathering details in VirtualHome for training the decision-making policies. We introduce the expert data collection and active data gathering in Appendix E.1 and Appendix E.2, respectively.

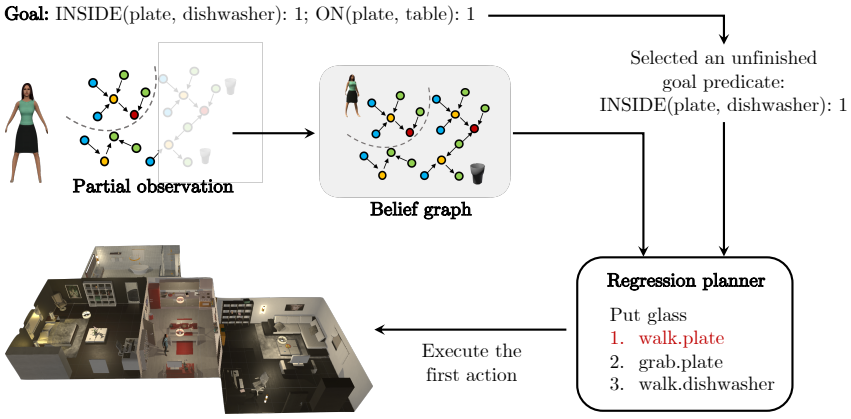


Figure 3: **Regression planner**. Given a task described by goal predicates, the planner generates an action sequence to accomplish this task. The agent has a belief about the environment, *i.e*. an imagined distribution of object locations. As the agent explores the environment, its belief of the world becomes closer to the real world. At every step, the agent updates its belief based on the latest observation, finds a new plan using the regression planner, and executes the first action of the plan. If the subtask (described by the goal predicate) has been finished, the agent will select a new unfinished subtask, otherwise, the agent will keep doing this subtask until finish it.

## E.1 Expert Data Collection

**VirtualHome-Imitation Learning Dataset.** To train the models, we collect a set of expert trajectories in VirtualHome using regression planning (RP) [2]. We follow the implementation of the regression planner used in [4]. Given a task described by goal predicates, the planner generates an action sequence to accomplish this task. As shown in Figure 3, the agent has a belief about the environment, *i.e*. an imagined distribution of object locations. As the agent explores the environment, its belief of the world becomes closer to the real world. At every step, the agent updates its belief based on the latest observation (see [4]), finds a new plan using the regression planner, and executes the first action of the plan. If the subtask (described by the goal predicate) has been finished, the agent will select a new unfinished subtask, otherwise, the agent will keep doing this subtask until it finishes.

Similarly to previous work [7, 6, 4], we generate training data using a planner that has access to privileged information, such as full observation of the environment and information about the pre-conditions and effects of each action. The planner allows an agent to robustly perform tasks in partially observable environments and generate expert trajectories for training and evaluation. We generate $20,000$ trajectories for training and $3,000$ trajectories for validation. Each trajectory has a goal, an action sequence, and the corresponding observations after executing each action.

## E.2 Active Data Gathering

The algorithm for active data gathering is shown in Algorithm 2. To sample the goal and initial state, we first generate a set of initial states in VirtualHome using the code released by [4]. For each initial state, we are able to get a set of feasible tasks that can be accomplished in this environment. For example, in an initial state, if the apple is on the kitchen table, a feasible task goal could be "put the apple inside the fridge". In contrast, "put the banana inside the fridge" is not a feasible task if there is no banana in the initial state.

We collect 9893 initial states, and randomly sample an initial state and its feasible goal every time when we reset the environment. After each data collection iteration, we obtain a set of new goals using the goal relabel function. We save the goal and its corresponding initial state in the replay buffers and use the same strategy to sample the goal and initial state in the next iteration.

**Algorithm 2:** Active Data Gathering

**Given:** a goal relabel function $f_l$;
**Initialize:** policy $\pi_\phi$; goal set $G$; training replay buffer $\mathcal{R}_{train} = \{\}$; validation replay buffer $\mathcal{R}_{val} = \{\}$;
**for** *iteration=1, N* **do**
  **for** *example=1, M* **do**
    Sample a goal $g$ from $G$ and an initial state $s_1$;
    **for** $t = 1, T$ **do**
      Sample an action from policy $\pi_\phi(a_t|g, h_t, o_t)$ or sample an action randomly;
      Execute $a_t$ and obtain a new observation $o_{t+1}$;
    **end**
    Store the trajectory $(o_1, a_1, \cdots, o_T, a_T, g)$ in the replay buffer $\mathcal{R}_{train}$ or $\mathcal{R}_{val}$;
  **end**
  Relabel each failure trajectory $d = (o_1, a_1, \cdots, o_T, a_T)$ in the replay buffers and get new goal $g' = f_l(d)$;
  Put new goals $g'$ in the goal set $G$;
  **for** $k = 1, K$ **do**
    **repeat**
      Sample data from $\mathcal{R}_{train}$ and update policy $\pi_\phi$;
    **until** *training episode ends*;
    Get validation accuracy using the data from $\mathcal{R}_{val}$;
  **end**
  $\pi_\phi = \pi_{\text{val\_best}}$
**end**

The *hindsight relabeling* stage is the key component for active data gathering. Here we provide more implementation details of how we relabel "failed" trajectories with new goals in the *hindsight relabeling* stage. For each "failed" trajectory, we extract its useful sub-trajectories and relabel a task goal $g'$ for it. We design a goal relabel function $f_l$ that generates a goal based on the sequence of observations and actions. To do this, we first use a hand-designed program to detect what tasks are contained in a "failed" trajectory. This program find useful tasks based on the keywords in the action list. For example in Figure 4, the program knows the trajectory containing a task of "`On(apple, kitchen table):1`" based on the action "$[put] < apple >< kitchentable >$".

The selected sub-trajectories are not always optimal. We thus design a rule to filter out bad trajectories, *i.e*. for trajectories with the same goal, selecting the "shorter" ones. One example is shown in Figure 5. Suppose that there are two trajectories having the same goal, *e.g*. "`On(apple, kitchen table):1`". The first trajectory has actions that are redundant or not related to the task, such as "$[walk] < bathroom >$" and "$[walk] < kitchen >$" while the second trajectory is more optimal given the goal. We select the second trajectory and delete the first trajectory from the replay buffer. Note that the "shorter" does not mean fewer actions, but fewer actions that are not related to the task. The *hindsight relabeling* stage allows sample-efficient learning by reusing the failure cases. The relabeled data are used to train policies in the *policy update* stage.

# F Test Sets in VirtualHome

In this section, we provide more details of each test set. We first introduce the test sets used for evaluating the proposed model trained on expert data, *i.e*. LID, in Section F.1. We then show the test sets used for evaluating the proposed model with active data gathering, *i.e*. LID-ADG, in Section F.2.

## F.1 LID Test Sets

In Section 6.1, we compared the proposed method and baselines trained on expert data. In Table 2, we provide a detailed description of each test subset, including the count of goal predicate types and the number of goal predicates in each task. The **In-Distribution** setting has 37 goal predicates in total and each task has $2 \sim 10$ goal predicates. The tasks are drawn from the same distribution as the training tasks. The **Novel Scenes** setting also has 37 goal predicates and each task has $2 \sim 10$ goal predicates. The objects are randomly placed in the initial environment. The **Novel Tasks** setting has

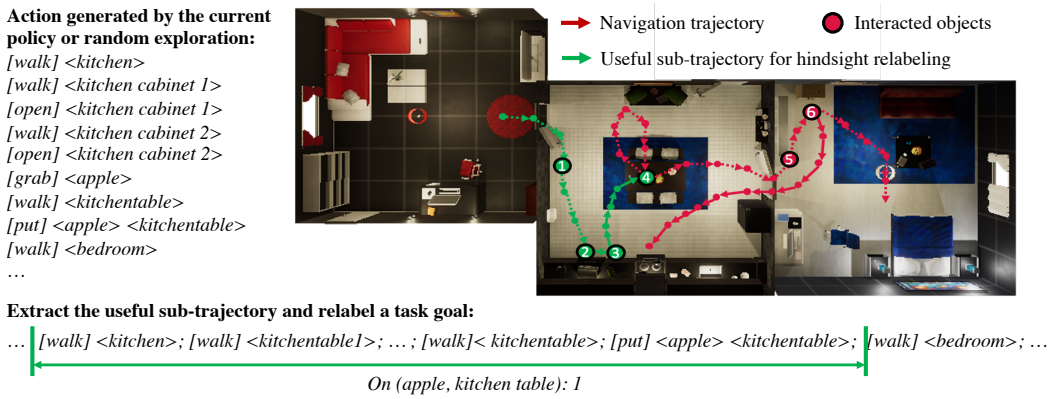


Figure 4: We first use a hand-designed program to detect what tasks are contained in the collected trajectory. This program find tasks based on the keywords in the action list. For example, the program knows the trajectory containing a task of "`On(apple, kitchen table):1`" based on the action "$[put] < apple >< kitchentable >$". Then the program extracts all previous actions related to this task using hand-designed rules.

**Goal:** *On (apple, kitchen table): 1*

**Action list 1:**
*… [walk] <livingroom>; [grab] <apple>; [walk] <kitchen>; [walk] <bathroom>; [walk] <kitchen>; [put] <apple> <kitchentable> …*

**Action list 2:**
*… [walk] <livingroom>; [grab] <apple>; [walk] <kitchen>; [put] <apple> <kitchentable> …*

Figure 5: Suppose there are two trajectories having the same goal, *e.g.* "`On(apple, kitchen table):1`". The first trajectory has actions that are redundant or not related to the task, such as "$[walk] < bathroom >$" and "$[walk] < kitchen >$" while the second trajectory is more optimal given the goal. We select the second trajectory and delete the first trajectory from the replay buffer. Note that the "shorter" does not mean fewer actions, but fewer actions that are not related to the task.

22 goal predicates in total and each task has $2 \sim 8$ goal predicates. The tasks are never seen during training.

### F.2 LID-ADG Test Sets

As we have mentioned in the main paper Section 9, one limitation of active data gathering is that it relies on hand-designed rules for task relabeling. In addition, it is sometimes challenging to define effective rules to extract useful sub-trajectories and get high-quality hindsight labels, especially when trajectories are long and tasks become more complex. Thus we only relabel short sub-trajectories, where the goal consists of a single goal predicate, *e.g.* "`On(apple, kitchen table):1`". During testing, we evaluate the success rate of approaches on solving such tasks as well, *i.e.* the count of the goal predicate equals to 1. The types of goal predicates are the same as Section F.1, *i.e.* 37 goal predicates in the *In-Distribution* setting and the *Novel Scenes* setting, and 22 goal predicates in the *Novel Tasks* setting.

## G Visualization of Attention Weights

To better understand how does LM pre-trained policies make decisions, we visualize the attention weights from the self-attention layers of GPT-2 [8] in Figure 6 and Figure 7. In the inference time, when we are decoding the actions, we save the self-attention weights with respect to different layers and different heads. Then, we use BertViz library [9] to visualize normalized attention weights. We show the attention weights from the input to the output of **LID-Text (Ours)**. The order of tokens in the input and ouput is observation, goal, and history. In Figure 6 and Figure 7, the left side is the query side. The boldness of the lines is proportional with the attention weight.

Figure 6 illustrates the attention weights of a layer named "Head 3 Layer 2". We show attention weights on two different tasks. We find that "Head 3 Layer 2" can capture objects in the goal

Table 2: **Test sets used for evaluating the proposed model trained on the expert data.** We show the count of goal predicate types and the number of goal predicates used in each task.

| Test Sets | Predicate Types | #Predicate Per Task | Compared with the training set |
|---|---|---|---|
| **In-Distribution** | 37 | $2 \sim 10$ | Tasks are drawn from the same distribution as training tasks. |
| **Novel Scenes** | 37 | $2 \sim 10$ | The objects are randomly placed in the initial environment. |
| **Novel Tasks** | 22 | $2 \sim 8$ | Tasks are never seen during training. |

predicates, such as "wineglass" and "cutleryfork" in the left figure, and "pancake" and "chicken" in the right figure (the figures are cropped for visualization).

Figure 7 illustrates the attention weights of layers named "Head 1 Layer 2" (left) and "Head 4 Layer 11" (right). Given the goal predicates, history, and the current observation, the policy predicts the next action as "grab milk". We find that "Head 1 Layer 2" is able to capture objects in the goal predicates, such as "milk", "pancake", and "chicken" while "Head 4 Layer 11" focuses on the interacted object in the predicted action, such as "milk".

The attention weights from different self-attention layers are significantly different—some self-attention layers assign high attention weight to objects in the goal predicates while some layers focus on the interacted object. There are also some layers that do not have interpretable meanings. The attention weights just provide us an intuition of how does the internal language model works, more quantified results are reported in the main paper.

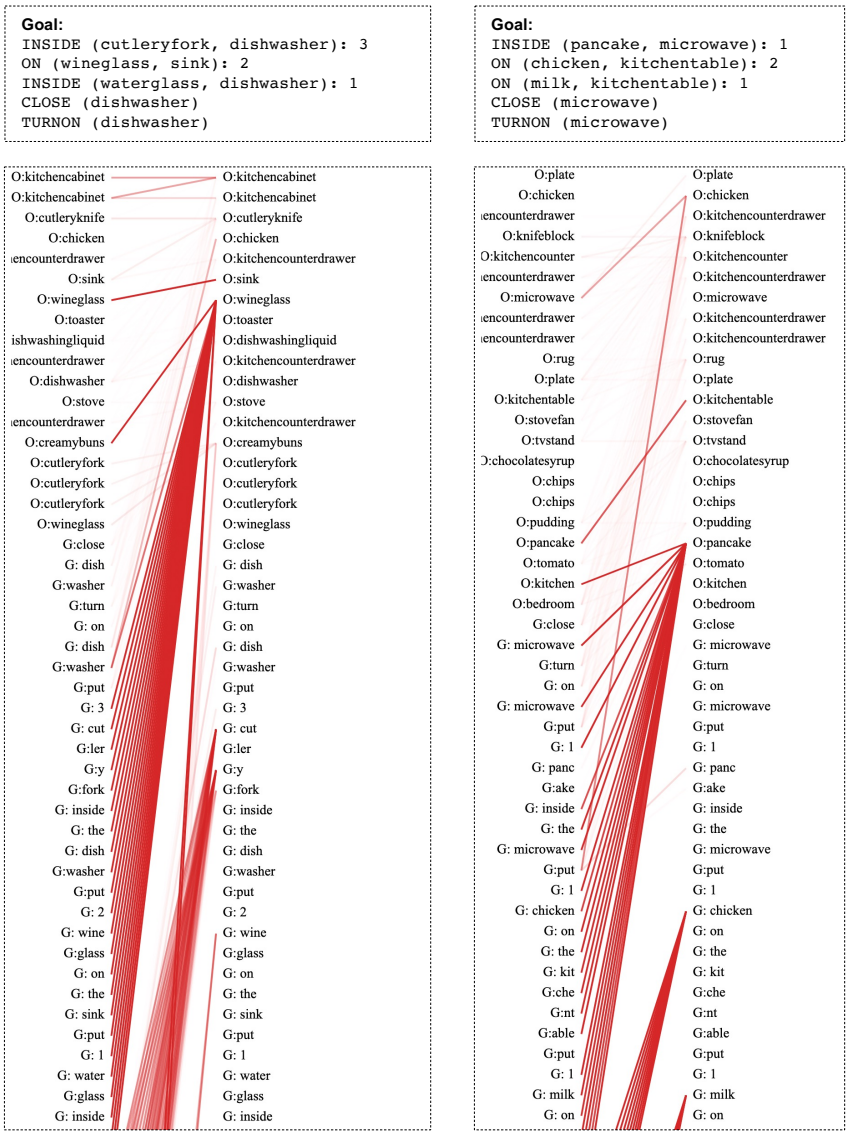


Figure 6: **Attention weights of a layer named "Head 3 Layer 2".** We show attention weights on two different tasks. We find that "Head 3 Layer 2" is able to capture objects in the goal predicates, such as "wineglass" and "cutleryfork" in the left figure, and "pancake" and "chicken" in the right figure (the figures are cropped for visualization).

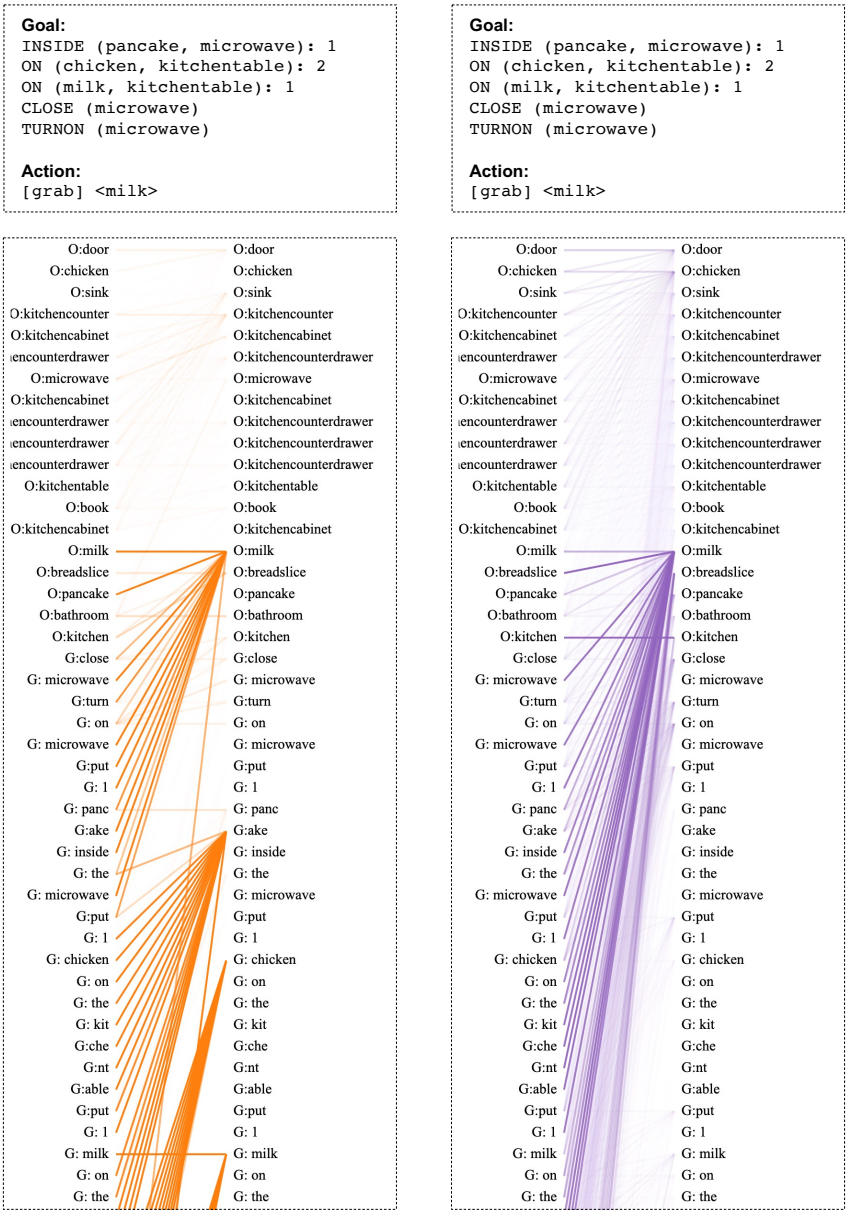


Figure 7: **Attention weights of layers named "Head 1 Layer 2" (left) and "Head 4 Layer 11" (right).** Given the goal predicates, history, and the current observation, the policy model predicts the next action as "grab milk". We find that "Head 1 Layer 2" can capture objects in the goal predicates, such as "milk", "pancake", and "chicken" while "Head 4 Layer 11" focuses on the interacted object in the predicted action, such as "milk".

## References


[1] D. Y.-T. Hui, M. Chevalier-Boisvert, D. Bahdanau, and Y. Bengio. Babyai 1.1, 2020.

[2] R. E. Korf. Planning as search: A quantitative approach. *Artificial intelligence*, 33(1):65–88, 1987.

[3] X. Puig, K. Ra, M. Boben, J. Li, T. Wang, S. Fidler, and A. Torralba. Virtualhome: Simulating household activities via programs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8494–8502, 2018.

[4] X. Puig, T. Shu, S. Li, Z. Wang, J. B. Tenenbaum, S. Fidler, and A. Torralba. Watch-and-help: A challenge for social perception and human-ai collaboration. *arXiv preprint arXiv:2010.09890*, 2020.

[5] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[6] B. Shen, F. Xia, C. Li, R. Martín-Martín, L. Fan, G. Wang, S. Buch, C. D'Arpino, S. Srivastava, L. P. Tchapmi, et al. igibson, a simulation environment for interactive tasks in large realisticscenes. *arXiv preprint arXiv:2012.02924*, 2020.

[7] M. Shridhar, J. Thomason, D. Gordon, Y. Bisk, W. Han, R. Mottaghi, L. Zettlemoyer, and D. Fox. Alfred: A benchmark for interpreting grounded instructions for everyday tasks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10740–10749, 2020.

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[9] J. Vig. A multiscale visualization of attention in the transformer model. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 37–42, Florence, Italy, July 2019. Association for Computational Linguistics.

[10] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.