Pre-Trained Language Models for Interactive Decision-Making

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.

Abstract

Language model (LM) pre-training is useful in many language processing tasks. But can pre-trained LMs be further leveraged for more general machine learning problems? We propose an approach for using LMs to scaffold learning and generalization in general sequential decision-making problems. In this approach, goals and observations are represented as a sequence of embeddings, and a policy network initialized with a pre-trained LM predicts the next action. We demonstrate that this framework enables effective combinatorial generalization across different environments and supervisory modalities. We begin by assuming access to a set of expert demonstrations, and show that initializing policies with LMs and fine-tuning them via behavior cloning improves task completion rates by 43.6% in the VirtualHome environment. We then examine how our framework may be used in environments without pre-collected expert data. To do this, we integrate an active data gathering procedure into pre-trained LMs. The agent iteratively learns by interacting with the environment, relabeling the language goal of past "failed" experiences, and updating the policy in a self-supervised loop. The active data gathering procedure also enables effective combinatorial generalization, outperforming the best baseline by 25.1%. Finally, we explain these results by investigating three possible factors underlying the effectiveness of the LM-based policy. We find that sequential input representations (vs. fixed-dimensional feature vectors) and favorable weight initialization are both important for generalization. Surprisingly, however, the format of the policy inputs encoding (e.g. as a natural language string vs. an arbitrary sequential encoding) has little influence. Together, these results suggest that language modeling induces representations that are useful for modeling not just language, but also goals and plans; these representations can aid learning and generalization even outside of language processing. [2]

1 Introduction

Language models (LMs) play a key role in machine learning approaches to natural language processing tasks [9]. This includes tasks that are not purely linguistic, and require nontrivial planning and reasoning capabilities [24, 13]: for example, instruction following, vision-language navigation, and visual question answering. Indeed, some of these tasks are so distant from language modeling that one can ask whether pre-trained LMs can be used as a general framework even for tasks that

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

[2]Project page: https://shuangli-project.github.io/Pre-Trained-Language-Models-for-Interactive-Decision-Making. Part of this work was done during Shuang's internship at NVIDIA.

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

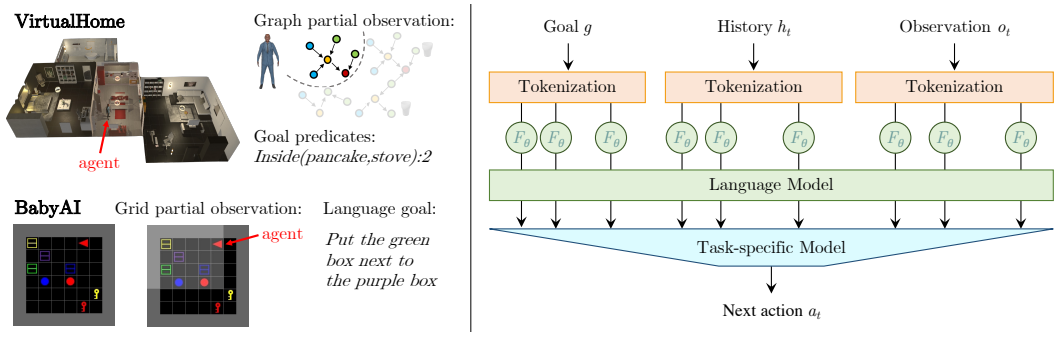

Figure 1: **Environments (left):** Different environments have different types of observations and goals. **Our approach (right):** We use pre-trained LMs as a general framework for interactive decision-making by converting policy inputs into sequential data. Such a method enables effective combinatorial generalization to novel tasks.

involve no language at all. If so, how might these capabilities be accessed in a model trained only to process and generate natural language strings?

In this paper, we study these questions through the lens of **embodied decision-making**, investigating the effectiveness of LM pre-training as a general framework for learning policies across a variety of environments. We propose **LID**, a framework that uses Pre-Trained **L**anguage Models for **I**nteractive **D**ecision-Making. As shown in Figure 1 (right), we encode the inputs to a policy—including observations, goals, and history—as a sequence of embeddings. These embeddings are passed to a policy network initialized with the parameters of a pre-trained LM, which is fine-tuned to predict actions. This framework is broadly applicable, accommodating goals and environment states represented as natural language strings, image patches, or scene graphs.

We find that imitation learning using pre-trained LMs as policy initializers improves in-domain performance and enables strong generalization over novel tasks. For i.i.d. training and evaluation tasks, this approach yields 20% more successful policies than other baseline methods in VirtualHome [31]. For combinatorial generalization to out-of-distribution tasks, *i.e.* tasks involving new combinations of goals, states or objects, LM pre-training confers even more benefits: it improves task completion rates by 43.6% for novel tasks (see Figure 3). These results hold for a variety of environment representations: encoding states as natural language strings, when possible, improves the data-efficiency of training, but even LMs fine-tuned on random environment encodings generalize combinatorially to new goals and states when trained on large enough datasets.

We further examine how our method may be used in environments, where the expert data is not available and an agent must actively gather data from the surrounding environment. To do this, we integrate an **A**ctive **D**ata **G**athering (**ADG**) procedure into pre-trained LMs as shown in Figure 2. The proposed method with ADG consists of three parts. First, **exploration** collects trajectories using a mix of random actions and actions generated by the current policy. Exploration is insufficient in this high dimensional problem and most of the trajectories will likely fail to achieve the end goal. A key insight is that even the failed trajectories contain useful sub-trajectories that solve certain sub-goals, and we relabel their goal in the **hindsight relabeling** stage. The relabeled goal describes what was achieved in the extracted sub-trajectory. The **policy update** stage samples relabeled trajectories to update the policy. The active data gathering procedure allows us to train the LM-policy without pre-collected expert data. It also outperforms reinforcement learning (RL) methods on embodied decision-making tasks and enables more effective generalization to novel tasks.

We hypothesize and investigate three possible factors underlying the effectiveness of pre-trained LM for generalization in policy learning: 1) input encoding scheme; 2) sequential input representations; and 3) favorable weight initialization. We investigate 1) by encoding the policy inputs as different types of sequences. Different input encoding schemes have only a negligible impact on the performance: the effectiveness of language modeling is not limited to utilizing natural strings, but in fact extends to arbitrary sequential encodings. We study 2) by encoding observations with a single vector embedding, thereby removing its sequential structure. This operation significantly degrades the model's performance on novel tasks. Finally, we investigate 3) by learning the parameters of the policy from scratch. The success rate after removing the pre-trained LM weights drops by 11.2%.

To summarize, our work has four main contributions.

- First, we propose to use **pre-trained LMs as a general scaffold** for interactive decision-making across a variety of environments by converting all policy inputs into sequential data.
- Second, the proposed method demonstrates that **language modeling improves combinatorial generalization in policy learning**: initializing a policy with a pre-trained LM substantially improves out-of-distribution performance on novel tasks.
- Third, we integrate an **active data gathering** procedure into the proposed approach to further enable policy learning on environments without using pre-collected expert data.
- Finally, we perform several analyses to explain the generalization capabilities of pre-trained LMs, finding that natural strings are not needed to benefit from LM pre-training, but the sequential input encoding and weight pre-training are important.

These results point to the effectiveness of the proposed framework with pre-trained LMs as a general-purpose framework to promote structured generalization in interactive decision-making.

2 Related Work

In recent years, word and sentence representations from pre-trained LMs [29, 9, 33] have become ubiquitous in natural language processing [49, 30]. Some of the most successful applications of pre-training lie at the boundary of natural language processing and other domains, as in instruction following [13] and language-guided image retrieval [22].

Learning representations of language. From nearly the earliest days of the field, natural language processing researchers observed that representations of words derived from distributional statistics in large text corpora serve as useful features for downstream tasks [8, 11]. The earliest versions of these representation learning schemes focused on isolated word forms [25, 28]. However, recent years have seen a number of techniques for training (masked or autoregressive) language models to produce contextualized word representations (which incorporate information neighboring words in sentences and paragraphs) via a variety of masked-word prediction objectives [9, 47].

Applications of pre-trained LMs. LMs can be fine-tuned to perform language processing tasks other than language modeling by casting those tasks as word-prediction problems. Successful uses of representations from pre-trained models include syntactic parsing [19] and language-to-code translation [45]; successful adaptations of LM prediction heads include machine translation [49], sentiment classification [6] and style transfer [18]. A number of tasks integrate language and other modalities, including visual question answering and image captioning [48]. Recent works find that image representations can be injected directly into LMs' embedding layers [42].

Policy learning and LM. Traditional policy learning methods, such as PPO [37], DQN [27], DDPG [21], A3C [26], perform well on playing tasks on Atari, OpenAI gym [5], and MuJoCo [41]. Some of them might fail to solve more challenging tasks on embodied environments [31, 39]. Several recent papers [36, 17, 15] propose to use LM for policy learning. Frozen Pretrained Transformer (FPT) [23] demonstrates that pre-trained LMs require very little fine-tuning to *match* the performance of task-specific models on several image classification and numerical sequence processing tasks. Semi-Supervised Skill Learning with Latent Language (SL)^3 [38] shows that LMs can serve as an effective backbone for hierarchical policies that express plans as natural language strings [2, 4]. In this paper, we focus on building a general framework for decision-making tasks using pre-trained LMs, even when language is not provided as an input or output.

3 Decision-Making and Language Modeling

3.1 POMDPs and Policy Learning

We explore the application of LMs to general sequential decision-making tasks in partially observed environments. These tasks may be formalized as partially observable Markov decision processes (POMDPs). A POMDP is defined by a set of states, a set of observations, a set of actions, and a transition model $\mathcal{T}(s_{t+1}|s_t, a_t)$ that maps the current state and action to the next state. Importantly, in a POMDP setting, the observation o_t only captures a portion of the underlying state s_t, and an optimal decision-making strategy (a **policy**) must incorporate both the current observation and the history of previous observations and actions. In our experiments, policies are parametric models

$\pi_\phi(a_t|g, h_t, o_t)$ that output the probability of an action given the goals g, history information $h_t = \{o_1, a_1, \cdots, o_{t-1}, a_{t-1}\}$, and partial observations o_t of the current state s_t.

In Figure 1 (right), we show a high-level overview of the proposed method. We first convert all policy inputs into a sequence and provide them as input to a transformer encoder. Representations from this encoder model are then passed to a task-specific decoder that predicts actions. We collect a dataset of N training trajectories $\mathcal{D} = \{d^i\}_{i=1}^N$, where each trajectory consists of a goal and a sequence of observations and actions: $d^i = \{g^i, o_1^i, a_1^i, \cdots, o_{T_i}^i, a_{T_i}^i\}$, where T_i is the length of the trajectory. We then train the policy to maximize the probability of actions we want to achieve $\boldsymbol{a}^i = \{a_1^i, \ldots, a_{T_i}^i\}$ across trajectories using the cross-entropy loss:

$$\phi^* = \arg\min_\phi \left(-\sum_{i=1}^{N} \sum_{t=1}^{T_i} \ln \pi_\phi(a_t^i | g^i, h_t^i, o_t^i) \right). \tag{1}$$

3.2 Language models as policy initializers

Our experiments focus on **autoregressive**, **transformer-based LMs** [43]. These models are trained to fit a distribution over a text sequence $\boldsymbol{y} = \{y_i\}_{i=1}^n$ via the chain rule $p(\boldsymbol{y}) = p(y_1) \prod_{i=2}^n p(y_i \mid y_1, \ldots, y_{i-1})$. Each term on the right hand side is parameterized by a transformer network, which accepts the conditioned tokens as input. Each token passes through a learned embedding layer F_θ, then the full conditioned sequence is fed into the LM. In our work, we use a standard LM, GPT-2, to process the input sequence rather than to predict future tokens.

Both POMDP decision-making and language modeling are naturally framed as sequence prediction tasks, where successive words or actions/observations are predicted based on a sequence of previous words or actions/observations. This suggests that pre-trained LMs can be used to initialize POMDP policies by fine-tuning them to model high-reward or expert trajectories, as described below.

4 Approach

We evaluate the effectiveness of pre-trained LMs in solving decision-making tasks across environments. We use **BabyAI** [16] and **VirtualHome** [31] 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 (Figure 1 (left)).

4.1 Policy Network

We first examine whether pre-trained LMs provide effective initializers when states and action histories are represented as natural language strings. We encode the inputs to the policy—including observations, goals, and action histories—as sequences of words. These word sequences are passed to the LM (using its pre-trained word embedding layer F_θ) and used to obtain contextualized token representations. These token representations are averaged and used to predict actions. We design a policy network following the general policy framework proposed in Figure 1.

Environment encodings in VirtualHome. In VirtualHome, each goal consists of a sequence of predicates and multiplicities, and is translated into a templated English sentence (*e.g.* "`Inside(apple, fridge):2`" becomes "put two apples inside the fridge"). To encode the agent's partial observation, we extract a list of currently visible objects, their states (*e.g.* "open, clean"), and 3D world coordinates. We use a fully-connected layer to encode the 3D information and generate a feature representation of each object in the observation. To encode history, we store information about all previous actions and convert them into templated English sentences (*e.g.* "I have put the plate on the kitchen table and the apple inside the fridge").

Environment encodings in BabyAI. The observation by default is a 7×7 grid. We convert the observation into 7×7 text descriptions, *e.g.* "purple ball", "grey wall", "open door", and combine them into a long sentence. We then convert the history actions into text descriptions, *e.g.* "turn left" and "go forward". We combine the language instruction (without modification) with the observation and history text descriptions, and feed them to the pre-trained LM.

We note that the policy network described above does not strictly require that these encodings take the form of natural language strings—other encodings of the environment as a sequence also work

(see Section 7). This framework could be also generalized to support pixel-based observations using discretization schemes like the one employed in the Vision Transformer [10].

Action prediction. We pool the outputs of the pre-trained LM into a "context feature" that is used to predict the next action. In training, we maximize the probabilities of actions we want to achieve. In inference, we select the valid action with the highest probability. See **Appendix C.1** for more details.

VirtualHome and BabyAI have quite different observation spaces, action spaces, and goal spaces; however, we show that embedding policy inputs as sequences and utilizing the pre-trained LM as a policy initializer, enables effective generalization to novel tasks on both environments. We note that LID is not limited to VirtualHome and BabyAI, but is straightforwardly applicable to other embodied environments, such as ALFRED [40] and iGibson [39].

4.2 Data Gathering

We first examine LID through imitation learning on data collected by experts in Section 4.2.1. We then show that integrating an active data gathering procedure into LID enables policy learning without using expert data in Section 4.2.2. We use VirtualHome as an example to explain the data gathering.

4.2.1 Policy Learning with Expert Data

The policy model is first initialized from a pre-trained LM and then fine-tuned on data collected by experts. We build on the VirtualHome environment to collect a set of expert trajectories using regression planning [20] and create a **VirtualHome-Imitation Learning dataset**. Given a task described by goal predicates, the planner generates an action sequence to accomplish this task (See **Appendix E.1**). The planner has access to privileged information, such as information about the pre-conditions and effects of each action, allowing an agent to robustly perform tasks in partially observable environments and generate expert trajectories for training and evaluation.

4.2.2 Policy Learning with Active Data Gathering

Collecting expert data is sometimes challenging. It may require privileged information of the environment or human annotations, which can be time-consuming and difficult to scale. A promising way to scale up supervision is Hindsight Experience Replay (HER) [3], which allows agents to learn from orders of magnitude more data without supervision. However, existing HER methods [12] focus on simple tasks with small state/action space and full observability. They cannot tackle more complicated embodied decision-making tasks, requiring nontrivial planning and reasoning or natural language understanding. LID with the active data gathering (**LID-ADG**) can be used in solving tasks in such environments.

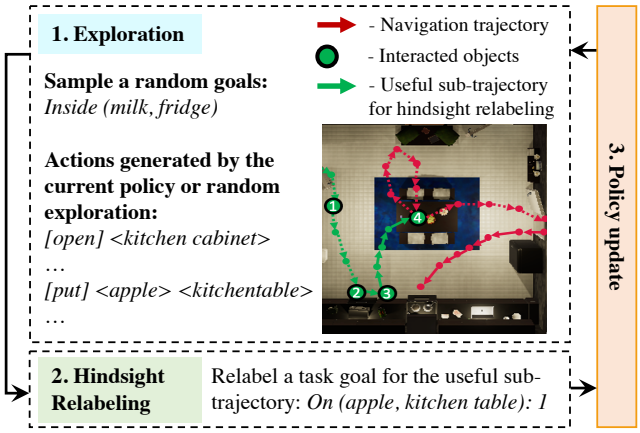

Figure 2: **LID with the active data gathering procedure.** By iteratively repeating the exploration, hindsight relabeling, and policy update, LID with active data gathering can learn an effective policy without using pre-collected expert data.

As shown in Figure 2, LID-ADG consists of three stages, *i.e.* **exploration**, **hindsight relabeling**, and **policy update**. The key idea is to gradually improve the task success rate by asking the agent to iteratively explore the environment, relabel failure samples, and update its policy using imitation learning. In the **exploration** stage, we first randomly sample a goal and an initial state. We then use a mix of random actions and actions generated by the current policy $\pi_\phi(a_t|g, h_t, o_t)$ to obtain the next action. We repeat this process until this episode ends. We collect M trajectories and store them in the replay buffers. The generated actions in the early stages can barely finish the given task. However, even the failed trajectories contain useful sub-trajectories that solve certain sub-goals. In the **hindsight relabeling** stage, we extract useful sub-trajectories and relabel a goal g' for each of

them. We design a goal relabel function f_l that generates a goal based on the sequence of observations and actions using hand-designed templates. In practice, we implement the goal relabel function as a program (see **Appendix E.2**). The *hindsight relabeling* stage allows sample-efficient learning by reusing the failure cases. During **policy update**, the agent samples the data from the replay buffers and updates its policy network π_ϕ.

By interleaving the exploration, hindsight relabeling, and policy update, LID-ADG can gradually improve the policy without requiring pre-collected expert data. In embodied environments with large action spaces, sparse rewards, and long-horizon planning, RL methods often struggle to obtain stable policy gradients during training. Our method enables sample-efficient learning from the sparse rewards by relabeling new goals for the bad samples that the agent fails to achieve. In addition, LID-ADG leverages the stability of supervised learning in the *policy update* stage, enabling it to outperform RL approaches on a wide range of decision-making tasks.

5 Experiment Setup

We evaluate the proposed method and baselines on VirtualHome and BabyAI.

5.1 VirtualHome

VirtualHome is a 3D embodied environment featuring partial observability, large action spaces, and long time horizons. We evaluate policies' performance from three aspects: (1) performance on in-distribution tasks; (2) generalization to novel scenes; and (3) generalization to novel tasks.

In-Distribution. The predicate types and their counts in the goal are randomly sampled from the same distribution as the training data. The objects are initially placed in the environment according to common-sense layouts (*e.g.* plates appear inside the kitchen cabinets rather than the bathtub).

Novel Scenes. The objects are placed in random positions in the initial environment without common-sense constraints (*e.g.* apples may appear inside the dishwasher).

Novel Tasks. The components of all goal predicates are never seen together during training (*e.g.* both plates and fridge appear in training goals, but `Inside(plate, fridge)` only appears in the test set. (See **Appendix F** for more details.)

We evaluate the success rates of different methods on each test set. A given episode is scored as successful if the policy completes its entire goal within the maximum allowed steps of the environment. On each of the 3 test subsets, we use 5 different random seeds and test 100 tasks under each seed. Thus there are 1500 examples used to evaluate each model.

5.2 BabyAI

BabyAI is a 2D grid world environment for instruction following. The observation in BabyAI is a $7 \times 7 \times 3$ grid describing a partial and local egocentric view of the state of the environment. We evaluate the methods on four representative tasks: *GoToRedBall*, *GoToLocal*, *PickupLoc*, and *PutNextLocal*. Performing well on the test set requires the models to generalize to new environment layouts and goals, resulting in new combinations of tasks not seen in training. For each method, we compute its success rates over 500 episodes on each task.

6 Experiments

We first show results of the proposed method and baselines for embodied decision-making tasks using expert data in Section 6.1. We then show our results when using actively gathered data in Section 6.2.

6.1 Embodied Decision Making with Pre-trained Language Model (LID)

6.1.1 Results on VirtualHome

We compare the proposed method with a variety of baselines using different policy architectures.

LID-Text (Ours) is the proposed method that converts all environments inputs into text descriptions. The pre-trained LM is fine-tuned for decision-making (conditioned on goals, observations, and histories) as described in Section 4.1.

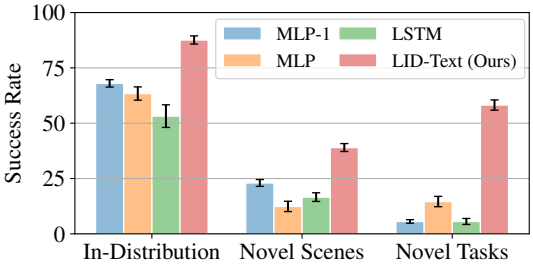

Figure 3: **Comparisons of the proposed method and baselines on VirtualHome.** All the methods are trained on expert data using imitation learning. *MLP-1*, *MLP*, and *LSTM* are baselines without using the pre-trained LM. The proposed method, LID-*Text (Ours)*, outperforms all baselines.

Tasks	Methods	Number of Demos				
		100	500	1K	5K	10K
GoToRedBall	BabyAI-Ori [16]	81.0	96.0	99.0	99.5	99.9
	LID-Text (Ours)	**93.9**	**99.4**	**99.7**	**100.0**	**100.0**
GoToLocal	BabyAI-Ori [16]	55.9	84.3	98.6	**99.9**	99.8
	LID-Text (Ours)	**64.6**	**97.9**	**99.0**	99.5	99.5
PickupLoc	BabyAI-Ori [16]	28.0	58.0	93.3	97.9	**99.8**
	LID-Text (Ours)	**28.7**	**73.4**	**99.0**	**99.6**	**99.8**
PutNextLocal	BabyAI-Ori [16]	**14.3**	16.8	43.4	81.2	97.7
	LID-Text (Ours)	11.1	**93.0**	**93.2**	**98.9**	**99.9**

Table 1: **Success rates on BabyAI tasks.** All the methods are trained on offline expert data using imitation learning. LID-*Text (Ours)* outperforms BabyAI-Ori, the method used in the original paper [16].

Recurrent Network. We compare our method with a recurrent baseline using an LSTM [14] to encode the history information. The hidden representation from the last time step together with the goal, and the current observation are used to predict the next action.

MLP and **MLP-1**. We perform additional comparisons with baselines that do not use the recurrent network and pre-trained LMs. *MLP* and *MLP-1* take the goal, histories, and the current observation as input and send them to the multilayer perceptron neural network (MLP) to predict actions. *MLP-1* has three more average-pooling layers than *MLP* that average the features of tokens in the goal, history actions, and the current observation, respectively, before sending them to the MLP layer.

Quantitative results. Each method is trained on $20K$ demos from the VirtualHome-Imitation Learning dataset, and then evaluated on the three test subsets: **In-Distribution**, **Novel Scenes**, and **Novel Tasks**. In Figure 3, LID-*Text (Ours)*, which initializes the policy with a pre-trained LM, has higher success rates than other methods. This difference is most pronounced in the **Novel Tasks** setting, where test tasks require combinatorial generalization across goals that are never seen during training. Here, LID-*Text (Ours)* dramatically (43.6%) improves upon all baselines. Such combinatorial generalization is necessary to construct general purpose agents, but is often difficult for existing approaches. Our results suggest that pre-trained LMs can serve as a computational backbone for combinatorial generalization.

6.1.2 Results on BabyAI

We use the standard training and test data provided by [16]. In BabyAI, performing well on the unseen test tasks with new environment layouts and goals requires approaches having combinatorial reasoning capabilities. In Table 1, we report the success rate of models trained on different number of demos. **BabyAI-Ori** [16] is the method used in the original paper. **LID-Text (Ours)** is the proposed method that converts policy inputs into a text sequence. Given enough training data, *i.e.* 10K demos, both methods achieve high success rates, but LID-*Text (Ours)* outperforms BabyAI-Ori given fewer training data, indicating the proposed method can generalize well to novel language tasks using fewer training data.

6.2 Pre-trained Language Model with Active Data Gathering (LID-ADG)

We compare **LID-ADG**, the proposed LM framework for decision-making using actively gathered data (Section 4.2.2), to a variety of baselines that do not use pre-collected expert data on VirtualHome.

Random. The agent selects the next action randomly from the valid action space at that state. **Goal-Object.** The agent randomly selects an object that in the goal and in the valid action space to interact with. For example, given a goal of "`Inside(apple, fridge):1`", this baseline might choose "grab apple", "open fridge", or other actions containing "apple" or "fridge". **Online RL.** We compare with PPO [37], one of the most commonly used online RL methods. For fair comparison, we equip PPO with the same main policy network as the proposed method. Our implementation is based on Stable Baselines3 [35]. **Hindsight Experience Replay.** We compare with DQN+HER used in [3] and modify its main policy network to be the same as the proposed method.

	In-Distribution	Novel Scenes	Novel Tasks
Random	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Goal-Object	0.8 ± 0.5	0.0 ± 0.0	0.4 ± 0.4
PPO	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
DQN+HER	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
LID-ADG (Ours)	$\mathbf{46.7 \pm 2.7}$	$\mathbf{32.2 \pm 3.3}$	$\mathbf{25.5 \pm 4.1}$

Table 2: **Comparisons of methods without using expert data on VirtualHome.** LID-*ADG (Ours)* is the only successful approach.

	In-Distribution	Novel Scenes	Novel Tasks
LID-ADG (Ours)	46.7 ± 2.7	$\mathbf{32.2 \pm 3.3}$	25.5 ± 4.1
PPO (LID-ADG Init)	$\mathbf{53.7 \pm 3.5}$	30.2 ± 3.4	$\mathbf{27.8 \pm 2.7}$
DT (LID-ADG Data)	42.4 ± 1.5	21.6 ± 2.48	16.8 ± 1.0

Table 3: The proposed method with active data gathering, LID-ADG (Ours), can be used as an policy initializer for online RL or a data provider for offline RL.

Quantitative results. We compare LID-ADG with baselines on VirtualHome in Table 2. Each experiment is performed 5 times with different random seeds. The **Random** baseline is always 0, indicating the tasks in VirtualHome cannot be easily solved by a random policy. **Goal-Object** is better than *Random* because *Goal-Object* has access to objects in the goal and it samples actions from a much smaller action space. The online RL baseline, **PPO**, fails to solve tasks in VirtualHome featured by partially observation, large state/action space, and long-term horizon. **DQN+HER** works well on simple tasks on 2D environments, but they cannot tackle VirtualHome tasks neither, requiring nontrivial planning and reasoning. LID-ADG does not require expert data and can solve the complicated tasks in 3D embodied environments which cannot be easily achieved using RL. [3]

Policy initializer and data provider. LID-ADG can further be used to initialize the weights for fine-tuning RL policies and to gather data for offline learning. As shown in Table 2, directly training RL, *e.g.* PPO, fails to solve tasks in VirtualHome. However, after using the policy trained by LID-ADG to initialize the PPO policy, we may effectively learn an interactive policy with good performance. In Table 3, **PPO (LID-ADG Init)** is initialized from LID-ADG and further fine-tuned to solve the tasks in VirtualHome. After initialization, PPO improves its success rate by 53.7% on the *In-Distribution* setting (See PPO results in Table 2 and Table 3). In addition, LID-ADG can provide data for offline learning. LID-ADG saves the relabeled data in replay buffers. We train Decision Transformer (DT) [7] using the data collected by LID-ADG. See **DT (LID-ADG Data)** in Table 3.

7 Analysis: Understanding the Sources of Generalization

The pre-trained LM policy, fine-tuned on either expert data or actively gathered data, exhibits effective combinatorial generalization. Is this simply because LMs are effective models of relations between natural language descriptions of states and actions [1], or because they provide a more general framework for combinatorial generalization in decision-making? We hypothesize and investigate three possible factors to understand the sources of such combinatorial generalization. We use policies trained on the expert data as an example to explain the experiments.

7.1 Input Encoding Scheme

We first hypothesize that converting environment inputs into natural language contributes to the combinatorial generalization as the LMs are trained on language data. We explore the role of *natural language* by investigating three alternative ways of encoding policy inputs to our model without using natural language strings: two in VirtualHome, and one in BabyAI. BabyAI results are in Appendix A.

Index encoding in VirtualHome. Different from LID-*Text (Ours)* that convert policy inputs into natural language strings, LID-*Index (Ours)* converts policy inputs into indexes. LID-*Index (Ours)* retains the discrete, serial format of the goal, history, and observation, but replaces each word with an index, and the embedding layer from the pre-trained LM with a new embedding layer trained from scratch. For example, *grab apple* is mapped to (5,3) based on the positions of *grab* and *apple* in the vocabulary set.

Unnatural string encoding in VirtualHome. LID-*Unnatural (Ours)* replaces the *natural language* tokens (e.g. converting the goal "`On(fork, table):1`" to *put one fork on the table*) with random ones (e.g. converting `On(fork, table)` to *brought wise character trees fine yet*). This is done by randomly permuting the entire vocabulary, mapping each token to a new token. Such a permutation breaks the semantic information in natural strings.

[3] Note that the results of LID-Text in Figure 3 and results of LID-ADG in Table 2 are not directly comparable because the difficulty level of the evaluated tasks are different. See **Appendix F** for more details.

Table 4: **Success rates of policies trained with different input encodings in the *Novel Tasks* setting on VirtualHome.** The text encoding is the most sample-efficient, but all models converge to similar performance given sufficient training data.

Methods	Number of Demos					
	100	500	1K	5K	10K	20K
LID-Text (Ours)	**8.8** $\pm$ **1.4**	**22.2** $\pm$ **1.7**	26.8 $\pm$ 1.0	46.0 $\pm$ 1.0	**58.2** $\pm$ **1.2**	58.2 $\pm$ 1.6
LID-Index (Ours)	6.4 $\pm$ 0.6	18.0 $\pm$ 3.8	18.8 $\pm$ 1.0	45.5 $\pm$ 2.1	54.6 $\pm$ 0.8	57.8 $\pm$ 0.9
LID-Unnatural (Ours)	6.8 $\pm$ 1.3	18.6 $\pm$ 2.1	**27.0** $\pm$ **1.1**	**47.2** $\pm$ **1.7**	55.8 $\pm$ 0.8	**58.8** $\pm$ **0.9**

LID-*Index (Ours)* and LID-*Unnatural (Ours)* have the same policy network as LID-*Text (Ours)*. All of them are fine-tuned on the expert data. Each of them is trained on different numbers of demos. The averaged results using 5 different random seeds on the Novel Tasks setting are reported in Table 4. Given few training data, e.g. 100 demos, all the models perform poorly, with success rates lower than 10%. LID-*Text (Ours)* achieves higher success rates than LID-*Index (Ours)* and LID-*Unnatural (Ours)* when the training data is increased, *e.g.* LID-*Text (Ours)* is around 4% higher than LID-*Index (Ours)* and LID-*Unnatural (Ours)* when there are 500 training demos. However, when the training dataset is further enlarged, *e.g.* 20K demos, success rates of all approaches reach similar performance. Such a result indicates that the effectiveness of pre-trained LMs in compositional generalization is not unique to natural language strings, but can be leveraged from arbitrary encodings, although adapting the model to arbitrary encodings may require more training data.

7.2 Sequential Input Representation

Next, we explore whether the generalization is caused by the sequential processing mechanisms in transformer-based LMs. We investigate whether the LM pre-trained policy will still be effective when the input encoding is not sequential. **No-Seq** uses the non-sequential inputs. It encodes the goal as a single vector by averaging the features of all goal embeddings. The history and observation features are obtained in the same way. All the features are then sent to the pre-trained LM to predict actions. As shown in Table 5, removing the sequential structure significantly hurts the performance on *Novel Tasks*. *No-Seq* achieves good performance on test tasks that are closer to training tasks, but cannot generalize well to more challenging unseen tasks. Thus, the combinatorial generalization of pre-trained LMs may be attributed in part to the transformer's ability to process sequential input representations effectively.

Table 5: **Experiments on sequential input representation and favorable weight initialization.** Fine-tuning the pre-trained weights and the usage of sequential encoding are important for combinatorial generalization.

	In-Distribution	Novel Tasks
LID-Text (Ours)	87.6 $\pm$ 1.9	**58.2** $\pm$ **2.3**
No-Seq	74.0 $\pm$ 2.3	2.0 $\pm$ 0.6
No-Pretrain	**90.8** $\pm$ **2.0**	47.0 $\pm$ 2.8
No-FT	51.2 $\pm$ 4.5	17.0 $\pm$ 2.9

7.3 Favorable Weight Initialization

Finally, we investigate if the favorable weight initialization from LM pre-training enables effective generalization of the proposed model. **No-Pretrain** does not initialize the policy using the pre-trained LM, but instead training the policy on the expert data from scratch. In Table 5, we find that removing the pre-trained weights can fit the in-domain data and thus performs well on the *In-Distribution* setting. However, its success rate is 11.2% lower than the proposed model on the *Novel Tasks* setting, indicating the pre-trained weights are important for effective generalization, but not necessary for effective data fitting. We further test a baseline, **No-FT**, that keeps the pre-trained weights of the language model but freezes them while training the rest model on our expert data. Freezing the pre-trained weights without fine-tuning significantly hurts the performance on both settings, suggesting that fine-tuning of the transformer weights is essential for effective combinatorial generalization.

Together, these results suggest that sequential input representations (vs. fixed-dimensional feature vectors) and favorable weight initialization are both important for generalization, however, the input encoding schemes (e.g. as a natural language string vs. an arbitrary encoding scheme) has little influence. These results point to the potential broader applicability of pre-trained LMs as a computational backbone for compositional embodied decision making, where arbitrary inputs, such as language, images, or grids, may be converted to sequential encodings.

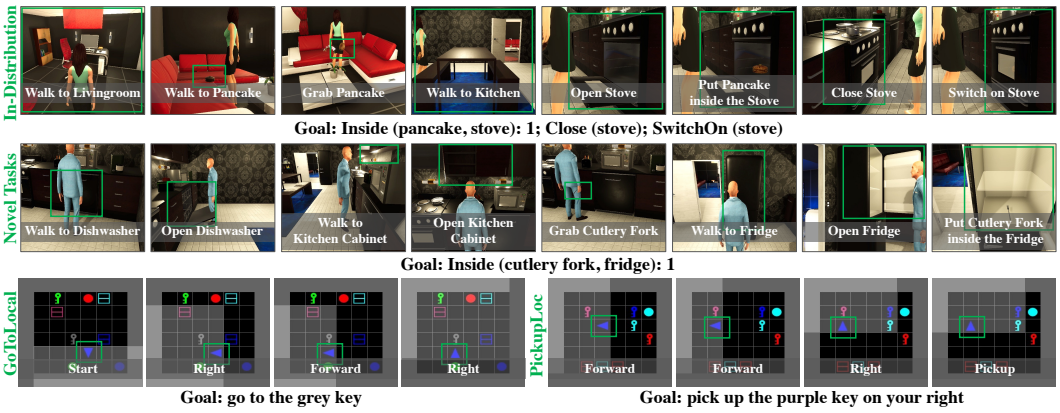

Figure 4: **Qualitative results of our model on VirtualHome and BabyAI.** We only show a sub-trajectory in each example to save space. The interacted objects are labelled by green bounding boxes.

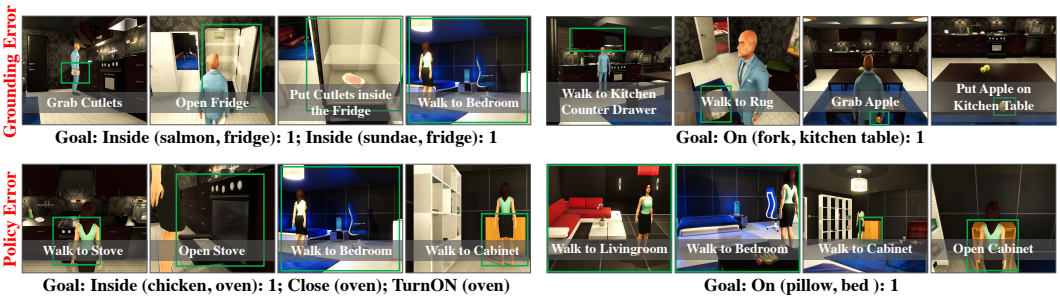

Figure 5: **Failure cases.** We show failure cases caused by the grounding error and policy error. The interacted objects are labelled by green bounding boxes.

8 Qualitative Results

In Figure 4, we show examples of LID-*Text (Ours)* completing tasks in VirtualHome and BabyAI. We show two successful examples from VirtualHome on the *In-Distribution* and *Novel Tasks* settings, and two successful examples from BabyAI on solving the *GoToLocal* and *PickupLoc* tasks. We only show short trajectories or extract a sub-trajectory for saving space.

Failure case analysis. In Figure 5, we show some failure cases of the proposed method. We observed two main types of failure cases: grounding error and policy error. For failures caused by the grounding error, the agent interacts with a wrong object that is not related to the given goal, *e.g.* the agent puts *cutlets* instead of the *salmon* inside the fridge. For failures caused by the policy error, the agent cannot find the target objects or does not interact with them. The proposed method that converts policy inputs into sequential encodings and feeds them to the general LM framework can accomplish decision-making tasks efficiently, however, there are still challenging tasks that the policy fails to accomplish. Larger LMs, *e.g.* GPT-3 [6], may improve the success rate of those challenging tasks.

9 Conclusion and Broader Impact

In this paper, we introduced LID, a general approach to sequential decision-making that converts goals, histories, and observations into sequences and processes them using a policy initialized with a pre-trained LM. We integrated an active data gathering procedure into the proposed method to enable policy learning without using expert data. Our analysis showed that input representation and favorable weight initialization both contribute to the generalization while the input encoding scheme has little influence. One drawback of the active data gathering is that it relies on hand-designed rules for task relabeling. More generally, a potential disadvantage of the proposed approach is that biases of the pre-trained LMs may influence its behavior, and further study of LID-based models' bias is required before they may be deployed in sensitive downstream applications. Nevertheless, our results demonstrate that LID enables effective combinatorial generalization across different environments, and highlight the promise of LM pre-training for more general decision-making problems.

References

[1] P. Ammanabrolu and M. O. Riedl. Playing text-adventure games with graph-based deep reinforcement learning. *arXiv preprint arXiv:1812.01628*, 2018.

[2] J. Andreas and D. Klein. Learning with latent language. In *North American Association for Computational Linguistics*, 2022.

[3] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba. Hindsight experience replay. *arXiv preprint arXiv:1707.01495*, 2017.

[4] M. L. Athul Paul Jacob and J. Andreas. Multitasking inhibits semantic drift. In *North American Association for Computational Linguistics*, 2021.

[5] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016.

[6] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[7] L. Chen, K. Lu, A. Rajeswaran, K. Lee, A. Grover, M. Laskin, P. Abbeel, A. Srinivas, and I. Mordatch. Decision transformer: Reinforcement learning via sequence modeling. *arXiv preprint arXiv:2106.01345*, 2021.

[8] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407, 1990.

[9] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[10] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[11] S. T. Dumais. Latent semantic analysis. *Annual review of information science and technology*, 38(1):188–230, 2004.

[12] D. Ghosh, A. Gupta, A. Reddy, J. Fu, C. Devin, B. Eysenbach, and S. Levine. Learning to reach goals via iterated supervised learning. *arXiv preprint arXiv:1912.06088*, 2019.

[13] F. Hill, S. Mokra, N. Wong, and T. Harley. Human instruction-following with deep reinforcement learning via transfer-learning from text. *arXiv preprint arXiv:2005.09382*, 2020.

[14] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[15] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. *arXiv preprint arXiv:2201.07207*, 2022.

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

[17] E. Jang, A. Irpan, M. Khansari, D. Kappler, F. Ebert, C. Lynch, S. Levine, and C. Finn. Bc-z: Zero-shot task generalization with robotic imitation learning. In *Conference on Robot Learning*, pages 991–1002. PMLR, 2022.

[18] N. S. Keskar, B. McCann, L. R. Varshney, C. Xiong, and R. Socher. Ctrl: A conditional transformer language model for controllable generation. *arXiv preprint arXiv:1909.05858*, 2019.

[19] N. Kitaev, S. Cao, and D. Klein. Multilingual constituency parsing with self-attention and pre-training. *arXiv preprint arXiv:1812.11760*, 2018.

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

[21] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[22] J. Lu, D. Batra, D. Parikh, and S. Lee. Vilbert: Pretraining task-agnostic visiolinguistic representations for vision-and-language tasks. *arXiv preprint arXiv:1908.02265*, 2019.

[23] K. Lu, A. Grover, P. Abbeel, and I. Mordatch. Pretrained transformers as universal computation engines. *arXiv preprint arXiv:2103.05247*, 2021.

[24] A. Majumdar, A. Shrivastava, S. Lee, P. Anderson, D. Parikh, and D. Batra. Improving vision-and-language navigation with image-text pairs from the web. In *European Conference on Computer Vision*, pages 259–274. Springer, 2020.

[25] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[26] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[28] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[29] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

[30] E. A. Platanios, A. Pauls, S. Roy, Y. Zhang, A. Kyte, A. Guo, S. Thomson, J. Krishnamurthy, J. Wolfe, J. Andreas, et al. Value-agnostic conversational semantic parsing. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 3666–3681, 2021.

[31] 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.

[32] 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.

[33] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever. Improving language understanding by generative pre-training. 2018.

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

[35] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[36] M. Reid, Y. Yamada, and S. S. Gu. Can wikipedia help offline reinforcement learning? *arXiv preprint arXiv:2201.12122*, 2022.

[37] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[38] P. Sharma, A. Torralba, and J. Andreas. Skill induction and planning with latent language. In *Association for Computational Linguistics*, 2022.

[39] 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.

[40] 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.

[41] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[42] M. Tsimpoukelli, J. Menick, S. Cabi, S. Eslami, O. Vinyals, and F. Hill. Multimodal few-shot learning with frozen language models. *arXiv preprint arXiv:2106.13884*, 2021.

[43] 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.

[44] 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.

[45] B. Wang, R. Shin, X. Liu, O. Polozov, and M. Richardson. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers. *arXiv preprint arXiv:1911.04942*, 2019.

[46] 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.

[47] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le. Xlnet: Generalized autoregressive pretraining for language understanding. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[48] Z. Yang, N. Garcia, C. Chu, M. Otani, Y. Nakashima, and H. Takemura. Bert representations for video question answering. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 1556–1565, 2020.

[49] J. Zhu, Y. Xia, L. Wu, D. He, T. Qin, W. Zhou, H. Li, and T.-Y. Liu. Incorporating bert into neural machine translation. *arXiv preprint arXiv:2002.06823*, 2020.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] See Section 9.
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] See Section 9.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] In the supplemental material.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Section 6 and Appendix C.2.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] See Section 6.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See Appendix C.2.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [Yes] See Section 9.
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

Appendix

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 6 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.

Table 6: **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

B Environments

We use **BabyAI** [16] and **VirtualHome** [31] 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.

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.

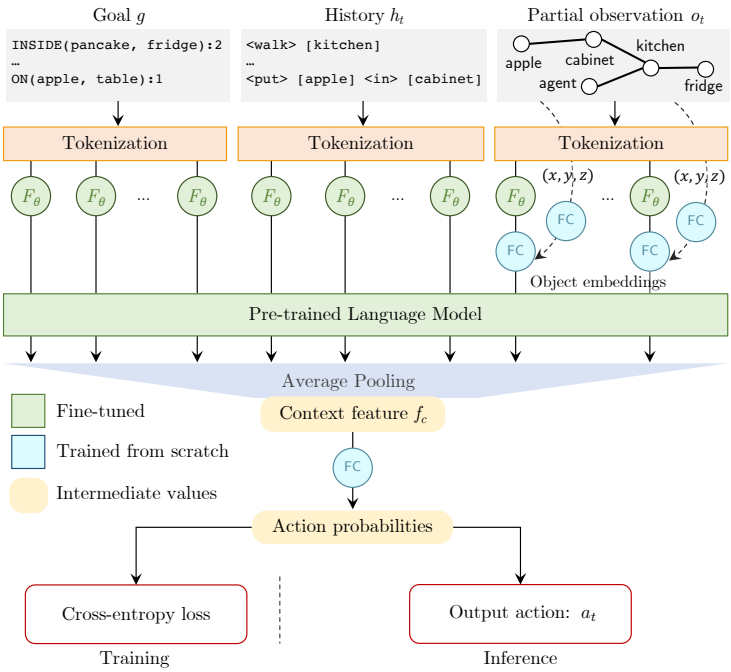

Figure 6: **Policy network in VirtualHome.** The observation, goal, and history are first converted into sequences and then passed through an embedding layer F_θ. The combined sequence is passed through a pre-trained LM, and the output tokens are pooled into a context feature vector for action prediction.

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 7×7 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.

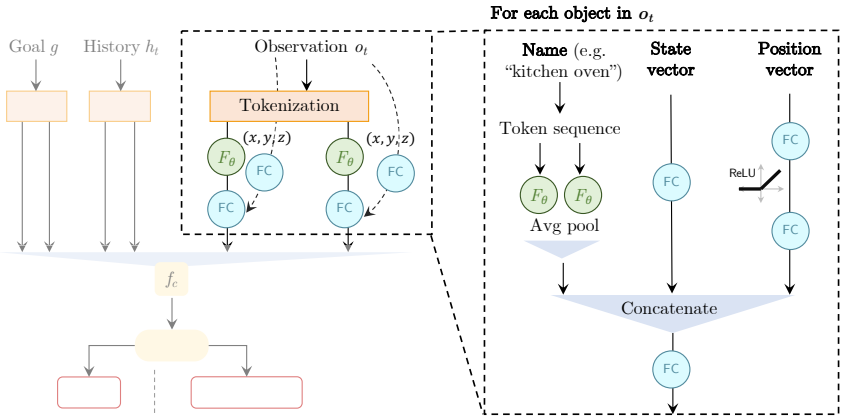

Figure 7: **Object encoding.** In VirtualHome, the partial observation of the environment state can be represented as a list of objects in the agent's view. Each object is represented by a name, a state vector, and position vector. **Object name encoding:** each object's name is an English phrase. We 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.

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 6, 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_θ) 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 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 7 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 7.

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 [34] using the Huggingface library [46].

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 π_φ;
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 π_φ(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_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.

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) [20]. We follow the implementation of the regression planner used in [32]. Given a task described by goal predicates, the planner generates an action sequence to accomplish this task. As shown in Figure 8, 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 [32]), 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 [40, 39, 32], 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.

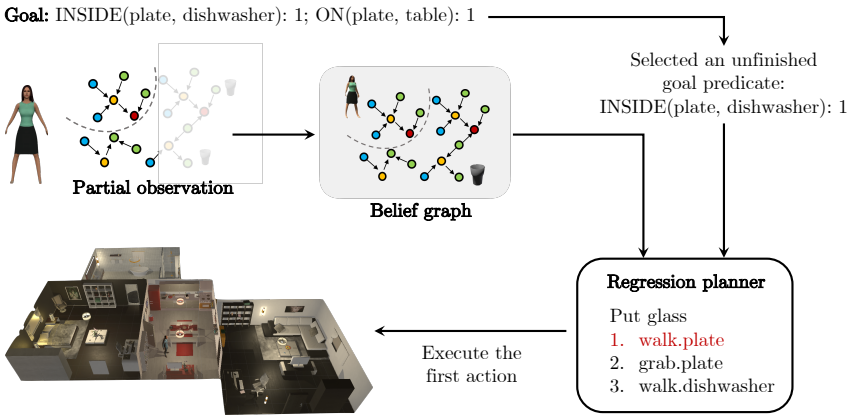

Figure 8: **Regression planner**. Given a task described by goal predicates, the planner generates an action sequence to accomplish this task. The agent has a belief about the environment, *i.e.* an imagined distribution of object locations. As the agent explores the environment, its belief of the world 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.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 [32]. 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.

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 9, 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 10. 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.

Algorithm 2: Active Data Gathering

Given: a goal relabel function f_l;
Initialize: policy π_ϕ; 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 π_ϕ;
    until *training episode ends*;
    Get validation accuracy using the data from $\mathcal{R}_{val}$;
  end
  $\pi_\phi = \pi_{\text{val_best}}$
end

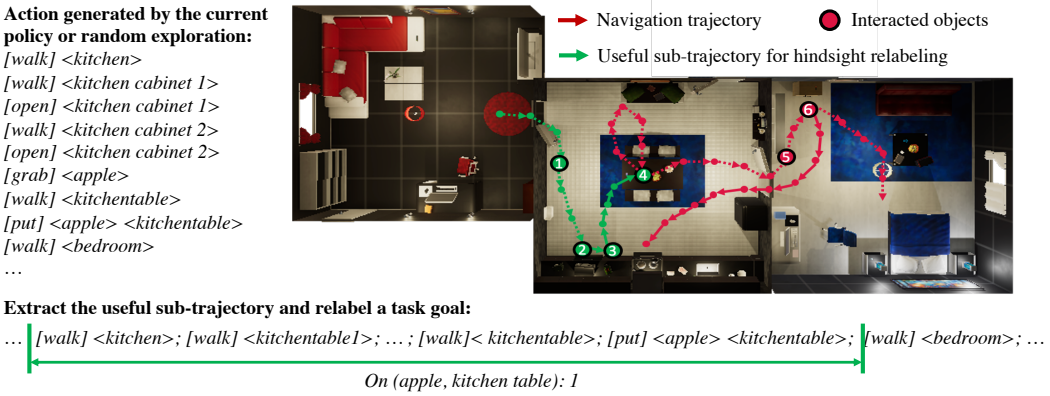

Figure 9: We first use a hand-designed program to detect what tasks are contained in the collected trajectory. This program find tasks based on the keywords in the action list. For example, the program knows the trajectory containing a task of "`On(apple, kitchen table):1`" based on the action "$[put] < apple >< kitchentable >$". Then the program extracts all previous actions related to this task using hand-designed rules.

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 7, 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 22 goal predicates in total and each task has $2 \sim 8$ goal predicates. The tasks are never seen during training.

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 10: 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.

Table 7: **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.

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 [43] in Figure 11 and Figure 12. 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 [44] 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 11 and Figure 12, the left side is the query side. The boldness of the lines is proportional with the attention weight.

Figure 11 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 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 12 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.

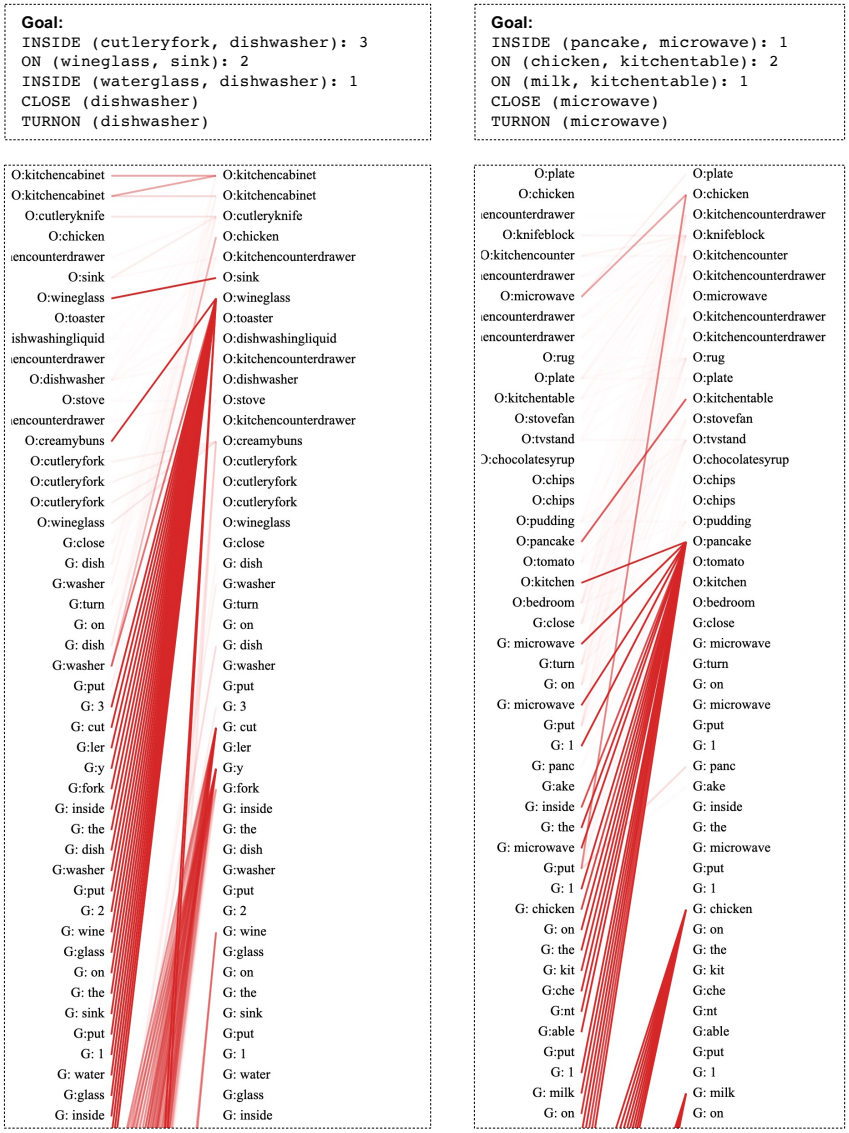

Figure 11: **Attention weights of a layer named "Head 3 Layer 2".** We show attention weights on two different tasks. We find that "Head 3 Layer 2" is able to capture objects in the goal predicates, such as "wineglass" and "cutleryfork" in the left figure, and "pancake" and "chicken" in the right figure (the figures are cropped for visualization).

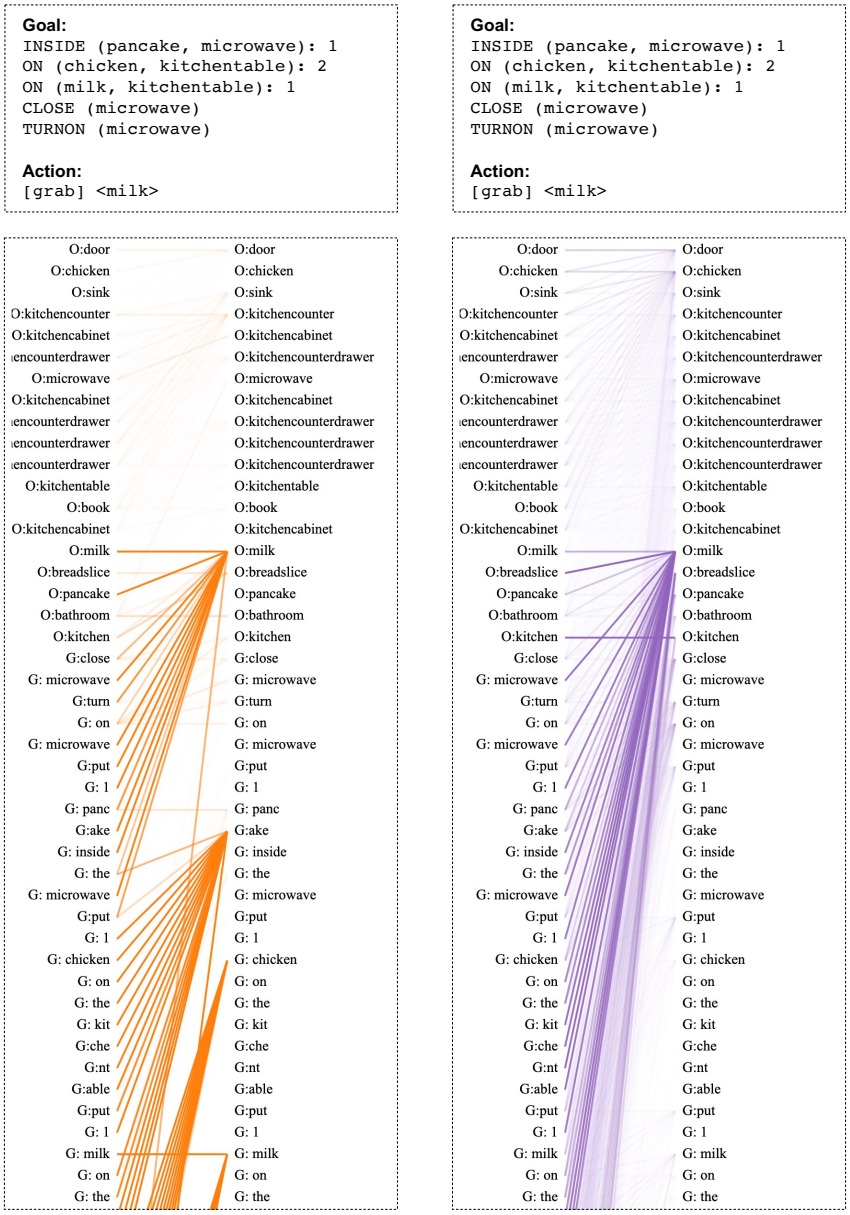

Figure 12: **Attention weights of layers named "Head 1 Layer 2" (left) and "Head 4 Layer 11" (right).** Given the goal predicates, history, and the current observation, the policy model predicts the next action as "grab milk". We find that "Head 1 Layer 2" can capture objects in the goal predicates, such as "milk", "pancake", and "chicken" while "Head 4 Layer 11" focuses on the interacted object in the predicted action, such as "milk".