DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Long-Horizon Language-Conditioned Robot Arm Manipulation with code Generation

Haihui Ye

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Long-Horizon Language-Conditioned Robot Arm Manipulation with code Generation

Langfristige Sprachkonditionierte Manipulation Roboterarm mit Codegenerierung

Author:	Haihui Ye
Supervisor:	Prof. Alois Knoll
Advisor:	Dr. Zhenshan Bing
Submission Date:	October 10, 2024

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, October 10, 2024 Haihui Ye

Acknowledgments

I would like to take this opportunity to express my sincere gratitude to my supervisor Prof. Alois Knoll and Advisor Dr. Zhenshan Bing, Shengqiang Zhang for their invaluable guidance, support, and encouragement throughout my Master's Thesis Project.

Furthermore, I would like to thank Lehrstuhl für Robotik, Künstliche Intelligenz und Echtzeitszsteme, TUM School of Computation, Information and Technology for the contributions to my research, providing experiment equipment and computational power for my research.

Lastly, I would like to acknowledge the support of my family and friends, who always gave me encouragement and motivation throughout my academic journey.

Abstract

Language-conditioned robot manipulation is an evolving field focused on enabling robots to interpret and execute complex tasks based on natural language commands provided by humans. By combining natural language processing, computer vision, and motion planning, robots are becoming more adaptable and intuitive, allowing for seamless human-robot interaction in industries such as manufacturing, healthcare, and logistics. Despite advancements in imitation learning and reinforcement learning, challenges remain in terms of generalization and adaptability in complex, unstructured environments. In this work, we introduce CapRavens, a novel approach to language-conditioned robot manipulation that leverages large language models (LLMs) to directly interpret human language commands and dynamically generate motion primitives—discrete, reusable actions such as grasping, moving, or placing objects. By integrating LLMs, CapRavens moves away from traditional imitation learning and reinforcement learning techniques, offering improved generalization and flexibility without requiring extensive training data. This method enables robots to perform complex manipulation tasks with high adaptability in real-world environments, providing a promising pathway for advancing robot autonomy. We demonstrate the effectiveness of CapRavens in various manipulation tasks and discuss its potential for overcoming the limitations of existing methods in language-based robot manipulation.

Kurzfassung

Die sprachkonditionierte Robotermanipulation ist ein sich entwickelndes Feld, das darauf ausgerichtet ist, Robotern zu ermöglichen, komplexe Aufgaben auf der Grundlage von natürlichen Sprachbefehlen von Menschen zu interpretieren und auszuführen. Durch die Kombination von natürlicher Sprachverarbeitung, Computervision und Bewegungsplanung werden Roboter anpassungsfähiger und intuitiver und ermöglichen eine nahtlose Mensch-Roboter-Interaktion in Branchen wie Fertigung, Gesundheitswesen und Logistik. Trotz Fortschritten beim Imitationslernen und Verstärkungslernen bleiben Herausforderungen hinsichtlich der Generalisierung und Anpassungsfähigkeit in komplexen, unstrukturierten Umgebungen bestehen. In dieser Arbeit stellen wir CapRavens, einen neuartigen Ansatz zur sprachkonditionierten Robotermanipulation vor, was der große Sprachmodelle (LLMs) nutzt, um menschliche Sprachbefehle direkt zu interpretieren und Bewegungsprimitive—diskrete, wiederverwendbare Aktionen wie das Greifen, Bewegen oder Platzieren von Objekten, dynamisch zu generieren. Durch die Integration von LLMs entfernt sich CapRavens von traditionellen Techniken des Imitationslernens und Verstärkungslernens und bietet eine verbesserte Generalisierung und Flexibilität, ohne dass umfangreiche Trainingsdaten erforderlich sind. Diese Methode ermöglicht es Robotern, komplexe Manipulationsaufgaben mit hoher Anpassungsfähigkeit in realen Umgebungen auszuführen, und bietet einen vielversprechenden Weg zur Weiterentwicklung der Roboterautonomie. Wir demonstrieren die Wirksamkeit von CapRavens bei verschiedenen Manipulationsaufgaben und diskutieren sein Potenzial zur Überwindung der Einschränkungen bestehender Methoden der sprachbasierten Robotermanipulation.

Contents

1. Introduction

Language-conditioned robot manipulation is an emerging field at the intersection of robotics, natural language processing, and computer vision. It aims to enable robots to interpret human language and perform complex manipulation tasks within diverse and dynamic environments. This approach has the potential to revolutionize industries like manufacturing, healthcare, and logistics by making robotic systems more intuitive and versatile. For such systems to be successful, robots need to not only understand natural language commands but also respond accurately to their physical surroundings.

So far, imitation learning and reinforcement learning have been the dominant methods in training agents for language-conditioned manipulation. Imitation learning, in particular, has seen success by using large datasets of play data [1] [2] to train robots to perform tasks like picking, placing, or pushing objects based on language instructions. However, due to problems like distributional shift and overfitting, these methods often face challenges in generalizing to unfamiliar environments, limiting their adaptability. Reinforcement learning-based approaches [3] [4] [5], while more adaptive, are often sample-inefficient and require extensive exploration in simulation or real-world scenarios [6] to achieve optimal performance.

Meanwhile, the rise of modern large language models (LLMs) such as GPT from OpenAI and Llama from Meta has brought unprecedented advancements in natural language understanding [7] and generation [8]. These models, trained on massive datasets, exhibit remarkable proficiency in interpreting complex linguistic structures and capturing nuanced contextual meaning. Their ability to generalize across different domains makes them exceptionally powerful for a variety of applications, from conversational agents to creative content generation. In the context of robotics, LLMs can bridge the gap between high-level human language and low-level robotic control, enabling a robot to understand intricate commands and execute sophisticated tasks. By leveraging the strength of LLMs, robots can move beyond pre-scripted commands to dynamic, adaptable behaviors that better align with human intent.

In this thesis, we propose CapRavens, a simple yet effective approach that exploits the common sense and reasoning ability embedded within LLMs to enhance the flexibility and generalization of language-conditioned robot manipulation. Inspired by Code as Policies [9] and LoHoRavens [10], CapRavens shifts away from imitation learning or reinforcement learning by using LLMs as task planner to directly interpret natural language commands and output a high-level task plan in code format which calls upon a library of motion primitives—predefined, reusable low-level actions like grasping or placing objects. We employ few-shot learning together with chain-of-thought technique to make the output more interpretable. This allows us to understand the thought process of the model in obtaining the final task plan. After task execution, another LLM as the reporter checks the final observation state and evaluates the task completion. During this phase, the reporter takes in as input

RGB-images of the task scene with their corresponding depth maps, as well as text prompt from the planner. The reporter returns the scene description and completion judgement as feedback to the planner, who will then decide whether to plan for the task again based on the report results. This modular approach allows the robot to generate adaptive behavior in real time, performing complex tasks with minimal training and robust performance in new environments.

2. Preliminaries

2.1. Markov Decision Process

A Markov Decision Process (MDP) mainly involves 4 parameters of reinforcement learning, forming a tuple (S, A, P, R). Here S and A each refer to the state space and the action space of the environment. $P : S \times A \to S$ is the state transition function or probability matrix that maps the state S_t to state S_{t+1} of next step according to action A_t such that

$$P^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a] \tag{2.1}$$

$R : S \times A \to \mathbb{R}$ is the reward function that maps the state S_t to reward R_{t+1} of next step according to action A_t such that

$$R^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a] \tag{2.2}$$

An additional parameter, the so-called discount factor $\gamma \in [0, 1]$ may be added to the definition of discounted MDP to balance the weight of future rewards.

2.2. Imitation Learning

Imitation Learning (IL) is a branch of Reinforcement Learning (RL) where the agent tries to learn a policy mapping sates to actions from the demonstrations given by the expert rather than from specified reward functions. The core of IL can be simply seen as an MDP, which means the environment of IL also consists of the tuple (S, A, P, R), but here reward function R is unknown. To make up for the lack of reward function, IL requires the expert's demonstration $\tau = (s_0, a_0, s_1, a_1, \dots)$, which is mainly a trajectory of states and actions, where the actions are based on the expert's policy π^*.

2.2.1. Behavioral Cloning

Behavioral Cloning (BC) is the simplest form of IL in which the agent learns the expert's policy by supervised learning. Regularly the pipeline of BC can be described as:

1. Collect demonstration $\mathcal{D} = \{\tau^*_i\}_{i=0}^{|\mathcal{D}|}$ as a set of trajectories τ^* from expert
2. Treat the trajectories from the demonstration as state-action pairs ordered by steps: $\{(s^*_0, a^*_0), (s^*_1, a^*_1), \dots\}$

3. Learn policy $\pi_\theta(a_t|s_t)$ using supervised learning by minimizing the loss function

$$\mathcal{L}_{BC}(\theta, \mathcal{D}) = \mathbb{E}_{\tau\sim\mathcal{D}}\left[-\sum_{t=0}^{|\tau|} log\pi_\theta(a_t|s_t)\right] \tag{2.3}$$

where θ is the parameter of the policy network

Although BC may work excellently in certain applications, it can be quite problematic for the majority of the cases because supervised learning assumes that the state-action pairs are independently and identically distributed (i.i.d.), however as defined by state probability transition matrix in MDP the current action induces the state of the next step, which means the errors of each step will gradually add up, hence the agent can easily enter a state which the expert demonstration doesn't cover, so that the training of the agent fails.

2.2.2. Direct Policy Learning

Direct Policy Learning (DPL) is an improved version of BC. It assumes that the expert is interactive so that the agent can not only learn from demonstration but also query the expert to evaluate the agent's rolled-out trajectory. The pipeline of DPL can be described as a loop including following steps:

1. Collect demonstration from expert and learn a policy using supervised learning
2. Roll out the policy into state-action trajectories and query the expert to evaluate them
3. Based on expert's feedback train an improved new policy

To make DPL more efficient, so that the agent can utilize all the previous training data and thus tend to avoid the mistakes it made in the past, algorithms like Data Aggregation and Policy Aggregation are commonly used during the training step. Data aggregation trains the new policy on all the previous training data. Policy Aggregation trains the new policy only on the training data received in the last iteration of the loop, and the newly trained policy will then be combined with all the previously trained policy via geometric blending. In the next iteration, both algorithms use the new policy for rolling-out.

Algorithm 1: DPL using Data Aggregation or Policy Aggregation

Input: Demonstration trajectory: $\{(s_0, a_0), (s_1, a_1), \ldots, (s_T, a_T)\} \sim \mathcal{D}$
Initial policy: $\pi_0(a_t|s_t)$
Output: Trained policy: π_t

```
Initialize policy π_0 ;
for t = 1 to T do
    Collect trajectories τ by rolling out τ_{t-1} ;
    Estimate state distribution P_t using s ∈ τ ;
    Collect interactive feedback: {π*(s)|s ∈ τ} ;
    Data Aggregation: Train π_t on P_1 ∪ ··· ∪ P_t ;
    or Policy Aggregation: Train π'_t on P_t, then π_t = βπ'_t + (1 − β)π_{t−1} ;
end
```

2.3. Goal-Conditioned Imitation Learning

Goal-Conditioned Imitation Learning (GCIL) is a specific version of IL that adds the goal on the basis of the state as the input of policy. Similar to IL, the demonstration $\mathcal{D} = \{(\tau_i^*, g_i)\}_{i=0}^{|\mathcal{D}|}$ is collected from the expert, where g_i refers to the goal of each trajectory. For the case of BC, the loss can be calculated by

$$\mathcal{L}_{GCBC}(\theta, \mathcal{D}) = \mathbb{E}_{(\tau,g)\sim\mathcal{D}}\left[-\sum_{t=0}^{|\tau|} log\pi_\theta(a_t|s_t, g)\right] \tag{2.4}$$

By minimizing the loss function the agent finally learns a goal-pointing policy $\pi_\theta(a_t|s_t, g)$.

2.3.1. Play-Supervised Goal-Conditioned Behavioral Cloning

Robotics and IL are closely intertwined fields, as IL provides a simple method that enables robots to learn new tasks and behaviors by observing and mimicking humans or other agents. Play-Supervised Goal-Conditioned Behavioral Cloning (Play-GCBC) [1] focuses on using BC to train a robot policy based on unlabeled demonstration data (play data) where goals are implied.

For Play-GCBC the collected play data $\mathcal{D} = \{\tau_i\}_{i=0}^{|\mathcal{D}|}$ has the same form as in the standard BC, but here τ refers to trajectory $\{(O_0, a_0), (O_1, a_1), \ldots, (O_T, a_T)\}$, where a_t is robot action and O_t is the set of observations from N sensory channels $\{o^1, o^2, \ldots, o^N\}_t$ of the robot agent. Here $o = \{I, p\}$, which consists of an RGB-image of first-person view observation I of the environment and internal proprioceptive state p of the robot. In order to map the high-dimensional raw observation O_t to one-dimensional fused state s_t as $s_t \leftarrow \Phi(O_t)$, Play-GCBC defined one encoder per sensory channel $\Phi = \{E_1, E_2, \ldots, E_N\}$ with network parameters θ_Φ, therefore the fused state $s_t = concatenate([E_1(o^1), E_2(o^2), \ldots, E_N(o^N)]_t)$.

Play-GCBC is trained in batches. For each training batch, a κ-length sequence ξ of observation-action pairs is sampled, then since policy of GCBC requires goal besides state as input, the final encoded observation in ξ is chosen as the synthetic goal state s_g. At each time step t in ξ, the policy takes in fused state s_t and goal state s_g as input, maps to the parameters of a distribution over next action a_t. Both the encoders and the policy are trained to maximize the log likelihood of each action taken during the sampled play sequence by a loss function similar to standard GCBC

$$\mathcal{L}_{PlayGCBC}(\theta, \tau) = \mathbb{E}_{\xi\sim\tau}\left[-\frac{1}{\kappa}\sum_{t=s}^{s+\kappa} log\pi_\theta(a_t|s_t, s_g)\right] \tag{2.5}$$

Algorithm 2: Play-GCBC

Input: Play data trajectory: $\{(O_0, a_0), (O_1, a_1), \ldots, (O_T, a_T)\} \sim \mathcal{D}$
Window bounds: $\{\kappa_{low}, \kappa_{high}\}$
Initial goal-conditioned policy: $\pi_\theta(a_t|s_t, g)$

Output: Policy parameter: θ

Randomly initialize model parameter θ ;
while *not done* **do**
  Sample a sequence length: $\kappa \sim \mathcal{U}(\kappa_{low}, \kappa_{high})$;
  Sample a sequence: $\xi = \{(O_{t:t+\kappa}, a_{t:t+\kappa})\} \sim \tau$;
  Set encoded goal state: $s_g = \Phi(O_{t+\kappa})$;
  Compute action loss: $\mathcal{L}_{Play-GCBC}(\theta, \tau)$;
  Update θ by taking the gradient step to minimize $\mathcal{L}_{Play-GCBC}$;
end

2.4. Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are a widely used Deep Learning (DL) model designed specifically for the process of data with grid-like topology such as images and videos. The input of a CNN is usually a 3-D tensor or more concretely a series of images. Each image inputted into the CNN, considered as a matrix or 2-D tensor, must pass through at least one convolution layer, in which one or more learnable convolution filters (kernels) will perform cross-correlation operations on it. The kernel slides over the input tensor, performing a dot product between the kernel weights and the local regions of the input. Then the tensor goes through an activation layer for non-linearity. After that the tensor usually further goes through a pooling layer to reduce the spatial dimensions, which also helps in reducing the computational load and the number of parameters. Because the convolution kernel in each layer only focuses on a limited range of the input, it can better extract local features and pass them to the next layer. As the network gets deeper, the neurons have larger perception fields and can thus capture more global features.

After several convolutional and pooling layers, the tensor obtained, also called feature maps, can be used in subsequent networks. A common form is to connect to a Multilayer Perceptron (MLP) for works such as regression, and another form is to connect to a CNN so that the entire network forms a Fully Convolutional Network (FCN) for works such as semantic segmentation. ResNet50 [11] and U-Net [12] are two well-known CNN models. ResNet50 adds to the network residual blocks with identity shortcuts to facilitate the training of deeper networks, while U-Net is a symmetric FCN composed of an encoder and a decoder with skip connections, which down-samples and then up-samples the input to complete a segmentation.

2.5. Transformers

When it comes to processing sequential data, the Recurrent Neural Networks (RNNs) have always been a fundamental method. For a basic RNN model, it consists of a series of recurrent layers, which take input and generate output at each time step. The output then in the next time step is fed back to the recurrent layer, thus let the network maintain a memory of previous inputs. However, RNN usually has the problem of vanishing gradient, making it difficult for the network to learn long-term dependencies, and the emergence of transformers [13] provides a new solution.

The transformer model relies entirely on attention mechanisms to draw global dependencies between input and output, dispensing with recurrence and convolutions, which allows for significantly more parallelization and efficiency, especially when training on large datasets. A transformer is usually composed of an encoder and a decoder. The encoder consists of a stack of N identical layers, each containing two blocks in order: the multi-head self-attention block and then the MLP block. Each block is followed by a residual connection and layer normalization. The decoder is similar to the encoder, but there is an additional masked multi-head self-attention block at the bottom of the multi-head self-attention block in each layer.

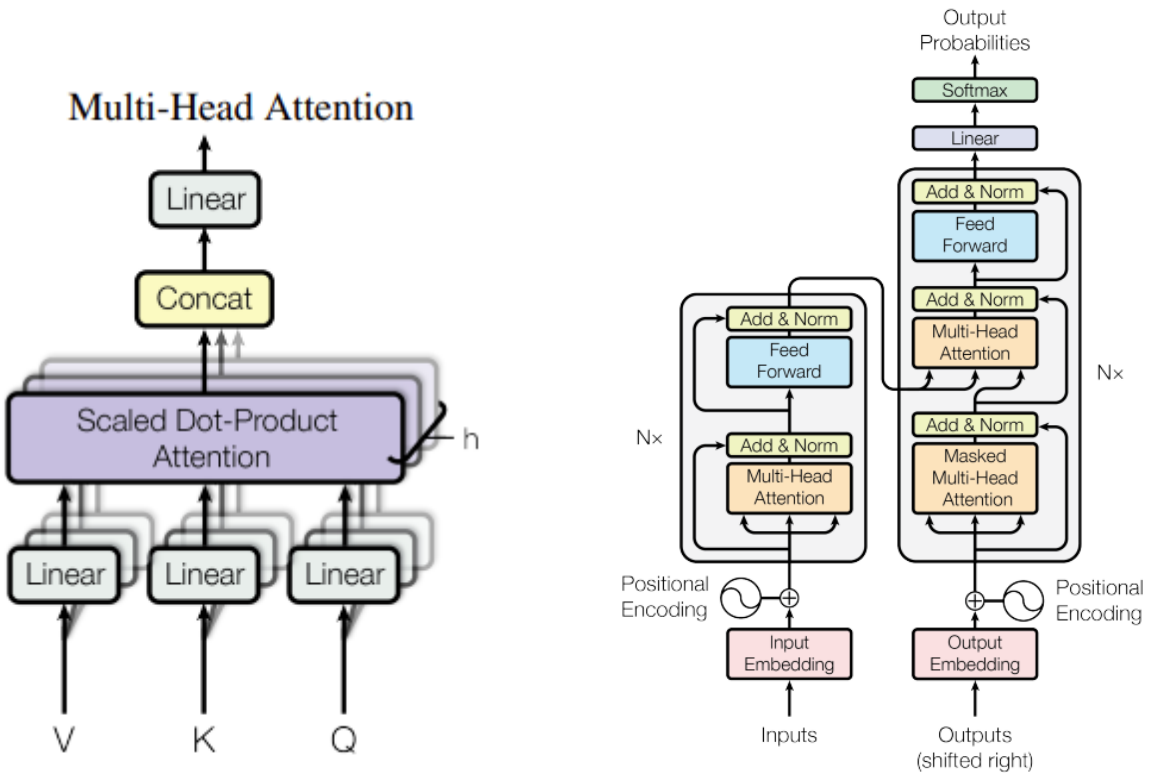
A transformer uses supervised-learning. By training both the source sequence and the target sequence are encoded and are added by a positional encoding, but the target sequence will first be shifted right for one token in order to train the decoder to predict the next token. The tensor of encoded source sequence goes in to the multi-head self-attention block where key vectors query Q, key K, and value V are projected from the input. Then the attention can be calculated by

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.6}$$

where d_k is the dimensionality of the key vectors. Then the tensor goes to MLP block, which introduces non-linearity as well as transforms the token representations obtained from the self-attention, and will further be passed to the next identical layer. The masked multi-head self-attention block of the encoder takes in tensor of encoded source sequence, doing the masked attention calculation similar to the encoder's self-attention, but with a masking mechanism to prevent attending to future positions in the sequence, ensuring that predictions for position i can depend only on the known outputs at positions less than i. Then the tensor together with the output tensor from the encoder go into the multi-head self-attention block and then into the MLP block. After N identical layers the output tensor finally passes through an MLP classifier to output a vector indicating the probability of each token. The loss is often calculated by cross-entropy, measuring the difference between the predicted and actual target tokens. By inferencing a source sequence is input to the encoder while only a start token is input to the decoder. The decoder then adds the output token to the input sequence and continues to loop until a complete sequence is generated.

Transformer shows many advantages over RNN such as:

- Parallelization: Due to the self-attention mechanism, transformers can process all



(a) transformer structure

(b) multi-head attention mechanism

Figure 2.1.: **(a)** Each attention head in the multi-head attention mechanism can be seen as extracting different "features" or "relationships" from the input. **(b)** By adding attention blocks a transformer can have better generalization and expressive power.

elements in the input sequence simultaneously, leading to faster training times.

- Long-Term Dependencies: The self-attention mechanism allows transformers to capture long-term dependencies more effectively than RNNs.
- Scalability: Transformers scale better with data and model size, making them suitable for very large datasets and models.

2.5.1. Large Language Models

Large language models (LLMs) are a type of Artificial Intelligence (AI) model designed to understand and generate human-like text. These models are trained on vast amounts of textual data and are built using neural network architectures, particularly variants of the transformer architecture. An LLM is usually unsupervised pre-trained by vast amount of unlabeled text data and then fine-tuned by smaller labeled datasets to achieve a specific ability such as chatting or coding. For example, Generative Pre-trained Transformer (GPT) from OpenAI can generate high-quality content-rich text based on the input prompt, and

AlphaCode from DeepMind can solve competitive programming problems by generating code solutions that are tested against various cases. The LLMs show capabilities of text generation, comprehension and few-shot learning, making it also promising in the field of robotics.

3. Related Works

The contemporary language-conditioned learning methods used in robot manipulation over the past few years can be divided into traditional aspects, which are mainly based on RL and IL, and the latest methodologies enhanced by foundational models like LLMs and VLMs. Following are some representative examples for each catagory.

3.1. Language-Conditioned Reinforcement Learning

Reinforcement learning has long been used for training robot agent to complete various tasks. For language-conditioned RL, one crucial point is to integrate language instruction into reward shaping.

3.1.1. Language-Conditioned Offline Reward Learning

Unlike sparse reward shaping where the agent only receives reward after completion of the task, nor dense reward shaping where the agent stepwise gets reward as progress feedback during the whole task execution, Language-conditioned Offline Reward Learning (LOReL) [14] comes up with an approach to learn a reward model $\mathcal{R}_\theta$ from demonstrations for each task.

Suppose the agent is to accomplish K tasks $\{\mathcal{T}_k\}_1^K \subset \mathcal{T}$, where $\mathcal{T}$ denotes the whole task space. An offline dataset $\mathcal{D} = \{\tau_1, \ldots, \tau_N\}$ of N trajectories, where each trajectory $\tau_n = \{[(s_0, a_0), (s_1, a_1), \ldots, (s_T)], l_n\}$ includes a sequence of state-action pairs and a language goal instruction. Let set (S, A, L) denote spaces of states, actions and language instructions. The binary reward function $\mathcal{R}_i : S \times S \to \{0, 1\}$ indicates the binary reward at state s for completing task $\mathcal{T}_i$ from initial state s_0, hence for each trajectory in $\mathcal{D}$, $\mathcal{R}_i$ should satisfy $\mathcal{R}_i(s_0, s_T) = 1$. The goal of LOReL is then to learn the parameterized reward model $\mathcal{R}_\theta : S \times S \times L \to \{0, 1\}$ which conditioned on initial state s_0 and state s who infer the true reward function $\mathcal{R}_i(s_0, s)$ for task $\mathcal{T}_i$, and a language instruction l. Based on $\mathcal{R}_\theta$, a stochastic language-conditioned policy $\pi : S \times S \times L \to A$ can further be instantiated to maximize the expected reward sum $\sum_{t=0}^{T} \mathcal{R}_i(s_0, s_t)$ for task $\mathcal{T}_i$.

To learn $\mathcal{R}_\theta$, both positive samples $(s_0, s_T, l_t) \in \tau_n \sim \mathcal{D}$ which constitute successfully completing an instruction and negative samples $(s'_0, s'_T, l') \sim \mathcal{D}$ which satisfy a different instruction or where initial and final states are deliberately reversed. Then it can be trained by minimizing the binary cross entropy loss

$$\mathcal{J}_{LOReL}(\theta) = \mathbb{E}_{(s_0, s_T, l) \sim \mathcal{D}} \left[log \mathcal{R}_\theta(s_0, s_T, l) \right] + \mathbb{E}_{(s'_0, s'_T, l') \sim \mathcal{D}} \left[log(1 - \mathcal{R}_\theta(s'_0, s'_T, l')) \right] \tag{3.1}$$

3.2. Language-Conditioned Imitation Learning

In general situations where goal-conditioned IL (GCIL) is used, the goals are usually the final state of the task. They can be an image of the external environment scene, the proprioceptive internal configuration, or just one-hot task encoding. However, this sort of goal may lead to ambiguity or inaccuracy when it comes to more complex tasks. Natural language as a most common medium for transmitting information, is intuitively considered to be an effective input to the IL policy. For Natural Language Conditioned Imitation Learning (NLCIL), the free-formed language instruction l takes part in the condition so that the policy becomes $\pi_\theta(a_t|s_t, g, l)$, or simply $\pi_\theta(a_t|s_t, l)$. In most cases NLCIL can be seen as a derivative or variant of GCIL, because language instruction can also be regarded as a goal.

3.2.1. Multicontext Imitation Learning

When the goal is a state in the trajectory, GCIL can incorporate large unstructured demonstration datasets because the goal space is equivalent to the state space. If the goal is in other forms, it is necessary to let the agent relate the goal to their onboard perceptions s and actions a. Multicontext Imitation Learning (MCIL) [15] thus introduces a way to represent a large set of policies by a single, unified function approximator that generalizes over states, tasks, and task descriptions.

Assuming demonstration data $\mathcal{D}$ is collected, in order to enable the trained policy be simultaneously conditioned on various form of goals, $\mathcal{D}$ is divided into multiple contextual demonstration subset $\{\mathcal{D}^0, \mathcal{D}^1, \ldots, \mathcal{D}^K\}$, each with a different way of describing task goals. Each $\mathcal{D}^k = \{(\tau_i^k, g_i^k)\}_{i=0}^{|\mathcal{D}^k|}$ holds a number of trajectories τ^k paired with goals $g^k \in G^k$. For example, $\mathcal{D}^0$ may contain one-hot task encoding goals, $\mathcal{D}^1$ may contain image goals, and $\mathcal{D}^2$ may contain language goals. With the aim of integrating all these different types of goals, MCIL trains a single latent goal conditioned policy $\pi_\theta(a_t|s_t, z)$ over all datasets concomitantly, as well as an encoder set $\mathcal{F} = \{f_\theta^0, f_\theta^1, \ldots, f_\theta^K\}$ for all demonstration subsets that maps the unique goal g to a common latent goal space $z \in \mathbb{R}^d$.

In order for the latent goal-conditioned policy to be effective, only less than 1% of the demonstration data actually needs to be linguistically annotated to form a goal, while the majority of perception and control are instead learned from imitation of data which uses relabeled state as a goal. By this training scheme, MCIL shows high efficiency that it can learn most actions from unlabeled data which is cheap to collect, while at the same time learn scalable task conditioning from only a very small number of labeled demonstrations.

Algorithm 3: MCIL

Input: Demonstration subsets with different context types: $\{\mathcal{D}^0, \mathcal{D}^1, \ldots, \mathcal{D}^K\}$
Latent goal encoders for each subset: $\{f_\theta^0, f_\theta^1, \ldots, f_\theta^K\}$
Initial latent goal-conditioned policy: $\pi_\theta(a_t|s_t, z)$
Output: Policy parameter: θ

Randomly initialize model parameters $\theta = \{\theta_\pi, \theta_{f^0}, \ldots, \theta_{f^K}\}$;
while *not done* **do**
  Initialize $\mathcal{L}_{MCIL} \leftarrow 0$;
  for $k = 0$ **to** K **do**
    Sample one $(demonstration, goal\ context)$ batch from the subset k : $(\tau^k, g^k) \sim \mathcal{D}^k$;
    Encode the context in shared latent goal space: $z = f_\theta^k(g^k)$;
    Accumulate imitation likelihood: $\mathcal{L}_{MCIL} += \sum_{t=0}^{|\tau^k|} log\pi_\theta(a_t|s_t, z)$;
  end
  Average over context types: $\mathcal{L}_{MCIL} * = \frac{1}{|D|}$;
  Update θ by taking a gradient step w.r.t $-\mathcal{L}_{MCIL}$;
end

3.2.2. Skill Prior

For the case of IL, the agent in general learns a policy for a task from scratch by imitating demonstrations. However, it is noted that intelligent agents such as humans often rely heavily on prior experience when learning a new task, which naturally gives rise to the idea of allowing the agent in BC to transfer series of actions into experience (a fixed combo) so that it can be further utilized in other different tasks without being learnt again. The abstraction of low-level actions improves the model's generalization ability and learning efficiency. An intuitive approach is to treat certain action sequence as a skill [16]. Assuming skill space is z, then instead of learning a policy $\pi_\theta(a_t|s_t)$ selecting among actions, the agent learns a high-level policy $\pi_\theta(z_t|s_t)$ which select among skills as well as a skill prior $p_a(z_t|s_t)$ which can bias the policy search, helping the agent to quickly identify and utilize relevant skills when faced with a new task.

In order to learn the latent skill embedding a Variational Autoencoder (VAE) is often implemented where a skill encoder $q(z|a_i)$ encodes a randomly sampled H-step action trajectory $a_i = \{a_{t+1}, a_{t+2}, \ldots, a_{t+H}\}$ from demonstration to skill space and a decoder $p(a_i|z)$ maps the skill z back to its corresponding concrete action sequence a_i. The reconstruction can be then learnt following the evidence lower bound (ELBO)

$$logp(a_i) \geqslant \mathbb{E}_{z\sim q}\left[\underbrace{logp(a_i|z)}_{\text{reconstruction}} - \beta \underbrace{D_{KL}(q(z|a_i)\|p(z))}_{\text{regularization}}\right] \tag{3.2}$$

where D_{KL} is Kullback-Leibler divergence and β is the parameter tuning the weight of

regularization term.

The skill prior is simultaneously trained together with the VAE, as both models can be jointly optimized and stable convergence be ensured by stopping gradients from the skill prior objective into the skill encoder. In actual practice it is found that simple Gaussian distribution works well for both models, hence the loss function for the whole training process can be written as

$$\mathcal{L}_{skill-prior} = \underbrace{\sum_{i=1}^{H}(a_i - \hat{a}_i)^2 + \beta D_{KL}[\mathcal{N}(\mu_z, \sigma_z) \| \mathcal{N}(0, I)]}_{\text{VAE loss (rec + reg)}} + \underbrace{D_{KL}[\mathcal{N}(\lfloor \mu_z \rfloor, \lfloor \sigma_z \rfloor) \| \mathcal{N}(\mu_p, \sigma_p)]}_{\text{skill prior loss}} \quad (3.3)$$

where (μ_z, σ_z) is the output of the skill encoder, (μ_p, σ_p) is the output of the skill prior and operator $\lfloor \cdot \rfloor$ indicates that gradients flowing through the variables are stopped.

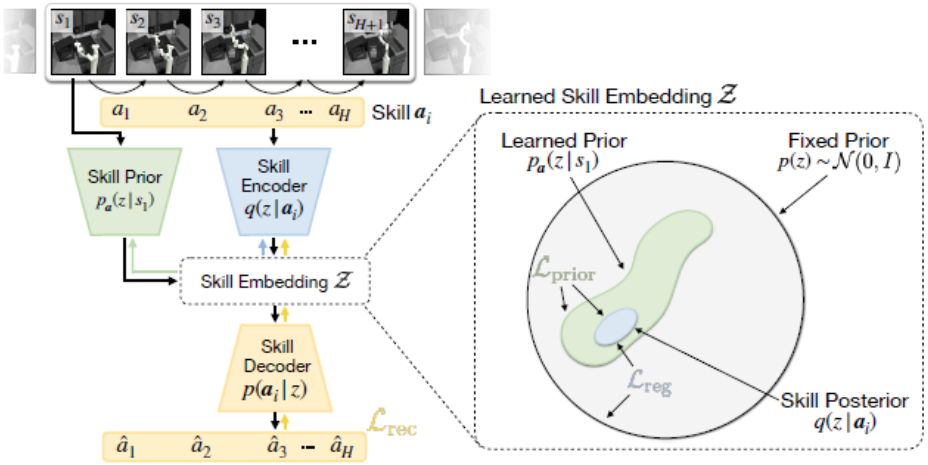


Figure 3.1.: Given a state-action trajectory from the dataset, the skill encoder maps the action sequence to a posterior distribution over latent skill embeddings. The action trajectory gets reconstructed by passing a sample from the posterior through the skill decoder. The skill prior maps the current environment state to a prior distribution over skill embeddings.

To apply the concept of skill priors on language-conditioned IL, SPIL [17] based on HULC [18] defines three robot base skills as transportation, rotation and grasping. Given one action sequence $x = (a_t, a_{t+1}, \ldots, a_{t+H-1})$, the distribution of these skills y can thus be written as

$$p(y = k|x) = \frac{w_k \sum_{i=t}^{t+H-1} |a_i^k|}{\sum_{k \in \{trans, rot, grasp\}} w_k \sum_{i=t}^{t+H-1} |a_i^k|} \quad (3.4)$$

where a^k denotes the action's corresponding change of k and w_k is the weight to balance the inconsistency in the scale. Let c represent a combination of the current state and the goal

state (s_c, s_g), the ELBO of the IL with skill prior can be written as

$$\begin{aligned}\mathcal{L}_{SPIL} = & \overbrace{\mathbb{E}_{z\sim q_{\phi}(x,c)} \log p_{\theta}(x|z)}^{\text{Reconstruction loss } (\mathcal{L}_{\text{huber}})} \\ & - \gamma_1 \sum_k q_{\omega}(y=k|c) \overbrace{D_{KL}(q_{\Phi,\lambda}(z|x,c)||p_{\kappa}(z|y))}^{\text{Base skill regularizer } (\mathcal{L}_{\text{skill}})} \\ & - \gamma_2 \overbrace{D_{KL}(q_{\omega}(y|c)||p(y))}^{\text{Categorical regularizer } (\mathcal{L}_{\text{cat.}})}\end{aligned} \tag{3.5}$$

where $p_{\theta}(x|z)$ is the skill generator network f_{θ}, and $f_{\omega} = q_{\omega}(y|c)$ corresponds to the skill labeller. $q_{\Phi,\lambda}(z|x,c)$ refers to the encoder network f_{Φ} plus the skill embedding selector network f_{λ}. $p_{\kappa}(z|y)$ constitutes the base skill prior locater f_{κ}.

Algorithm 4: Imitation Learning with Skill Priors

Input:

- Play Dataset and Language Dataset: $\mathcal{D} : \{(D^{\text{play}}, D^{\text{lang}})\}$
- Encoder f_{Φ}, the skill embedding selecter f_{λ}, the base skill locator f_{κ}, the base skill selector f_{ω}, the skill generator networks f_{θ}: $\mathcal{F} = \{f_{\Phi}, f_{\lambda}, f_{\kappa}, f_{\omega}, f_{\theta}\}$

Output: model parameters: $\{\Phi, \lambda, \omega\}$

```
Randomly initialize model parameters {Φ, λ, ω} ;
Initialize parameters θ and κ with pre-trained skill generator and base skill locator ;
Freeze the parameters θ and κ ;
while not done do
    Initialize L ← 0 ;
    for l in {play, lang} do
        Sample a (demonstration, context) (x^l, c^l) ~ D^l ;
        Encode the observation, goal, and plan embeddings, using the encoder network
         f_Φ ;
        Skill Embedding Selecter f_λ selects the skill embedding sequence ;
        Determinate a sequence of base skill probabilities with Base Skill Selector f_ω ;
        Determinate base skill locations in the latent space with Base Skill Locator f_κ ;
        Skill Generator f_θ maps the skill embeddings to action sequences ;
        Calculate the loss function L_l according to (3.5) ;
        Accumulate imitation loss L += L_l ;
    end
    update parameters {Φ, λ, ω} w.r.t L ;
end
```

3.3. Methods Empowered by LLMs and VLMs

In recent years, the integration LLMs and VLMs has gained significant traction in robotic manipulation. These models empower robots to interpret and execute complex tasks using natural language and visual cues, offering a flexible and scalable approach to generalizing across diverse environments. By leveraging pre-trained multimodal representations, LLMs and VLMs enable more intuitive human-robot interactions, paving the way for more robust and adaptable manipulation systems.

3.3.1. Transporter Networks

On the occasion of skill prior, the network learns a low-dimension latent skill embedding. For robot manipulation, these latent skills with usually 10 to 50 dimensions can concretely represent basic robot movements like translation and rotation of the joint, or the status switching of the end effector. Moreover, these skills can be further simplified as primitives for specific robot manipulation tasks such as tabletop manipulation.

A primitive is a pre-defined movement setting that takes as input a starting point and an ending point, which resemble the key frames of animation or filmmaking, and outputs a smooth motion. Consider skills as various movement patterns, then the primitive can be written as $\mathcal{P}_{skill}(\mathcal{T}_{start}, \mathcal{T}_{end})$. For instance, the skill to pick up something and then place it at a specified position can be rewritten in the form like $\mathcal{P}_{pick-place}(\mathcal{T}_{pick}, \mathcal{T}_{place})$. Hence under this structure the agent first learns a start policy $\pi_{start}(\mathcal{T}_{start}|s_t)$ and an end policy $\pi_{end}(\mathcal{T}_{end}|s_t, \mathcal{T}^*_{start})$ based on the obtained starting point.

In Transporter [19] the observation o_t replacing state s_t is a projection of the scene which for example can be reconstructed from RGB-D images, defined on a regular grid of pixels $\{(u, v)\}$ at timestep t of a sequential rearrangement task. Through camera-to-robot calibration, each pixel in o_t is corresponded to a starting point $\mathcal{T}_{start} \sim (u, v) \in o_t$.

Unlike the above-mentioned models using RNNs to treat action sequences, Transporter uses FCNs instead since

1. All the actions are resolved into primitives represented by starting and ending points, hence the action sequence becomes a list of pose pairs where pairs may not be necessarily consistent with one another as each pair indicates an independent action.

2. The inference of starting and ending points are based on observation which is a set of images projecting the scene. Furthermore, the forms of the output starting and ending points should also be images of the scene but with the target point position high-lighted.

3. FCNs are translationally equivariant by nature, which synergizes with spatial action representations. For example, in a pick-place task if an object to be picked in the scene is translated, then the picking pose also translates. Formally, equivariance here can be characterized as $\mathcal{T}_{pick} \sim \pi_{pick}(\mathcal{T}_{pick}|g \circ o_t) = g \circ \mathcal{T}_{pick} \sim \pi_{pick}(\mathcal{T}_{pick}|o_t)$, where g is a translation. Spatial equivariance has previously been shown to improve learning efficiency for vision-based picking. To this end, FCNs excel in modeling complex multi-modal picking distributions anchored on visual features.

4. CNNs can leverage parallelism at various levels to improve performance and scalability, from data and model parallelism within a single machine to distributed training across multiple machines.

The FCN ω for starting point models a value-action function $Q_{start}((u,v)|o_t)$ correlating with the success of starting an action, which will return the maximum when starting point imitates the demonstration. Therefore, the starting point is inferred by

$$\mathcal{T}_{start} = \underset{(u,v)}{\text{argmax}} Q_{start}((u,v)|o_t) \tag{3.6}$$

Once $\mathcal{T}_{start}$ is obtained the dense pixel-wise features from a partial crop $o_t[\mathcal{T}_{start}]$ centered on $\mathcal{T}_{start}$ are rigidly transformed then overlaid on top of other partial crops $o_t[\tau_i]$ centered on a set of different candidate poses $\{\tau_i\} = \{(u,v)_i\}$ to search for an appropriate ending point position, i.e. the $o_t[\tau_i]$ with the highest feature correlation. Thanks to the spatially consistent visual representation (here orthographic projection) the appearance of an object remains constant across different camera views, as well as across spatial transforms of rigid objects. Two feature-embedding FCNs ψ and ϕ then each operate on $o_t[\mathcal{T}_{start}]$ and o_t to get two feature maps acting as query and key. The value-action function for ending point is thus the cross-correlation of the maps

$$Q_{end}(\tau|o_t, \mathcal{T}_{start}) = \psi(o_t[\mathcal{T}_{start}]) * \phi(o_t)[\tau] \tag{3.7}$$

and the inference of ending point is similar to the one of starting point

$$\mathcal{T}_{end} = \underset{\{\tau_i\}}{\text{argmax}} Q_{end}(\tau|o_t, \mathcal{T}_{start}) \tag{3.8}$$

Besides planar translation the primitive can also fit planar rotations. Similar to translation where the plane is divided in to grids, the SO(2) rotations are discretized into k bins, and the input visual observation o_t is accordingly rotated for each bin. Therefore in SE(2), o_t can be defined by voxels $\{(u,v,w)_i\}$ where the new dimension w lies on the axis of the k-discretized rotations, and $\mathcal{T}_{start}$ as well as $\mathcal{T}_{end}$ are described discretely on this SE(2) grid. The FCN ψ in practice runs k times in parallel, once for a rotated o_t.

It also proves that the method utilized in SE(2) can be extended into SE(3). First the three degrees of freedom (DoF) in SE(2), assumed to be x, y and r_{xy}, are addressed, producing an estimated ending point $\hat{\mathcal{T}}_{end}^{SE(2)}$. Then to cover the remaining two rotation r_{xz} and r_{yz} and the translation z, two FCNs $\psi^{'}$ and $\phi^{'}$, which are identical to ψ and ϕ but with three separate cross-correlations ($*_3$) from subsets of channels, are used inside an additional three-headed MLP f for nonlinearity

$$r_{xz}, r_{yz}, z = f\left[\psi^{'}(o_t[\mathcal{T}_{start}]) *_3 \phi^{'}(o_t)[\hat{\mathcal{T}}_{end}^{SE(2)}]\right] \tag{3.9}$$

Suppose using demonstration dataset $\mathcal{D} = \{\zeta_1, \zeta_2, \ldots, \zeta_n\}$, where $\zeta_i = \{(o_0, a_0), (o_1, a_1), \ldots\}$ is a sequence of one or more observation-action pairs (o_t, a_t) and a_t is equivalent to a primitive

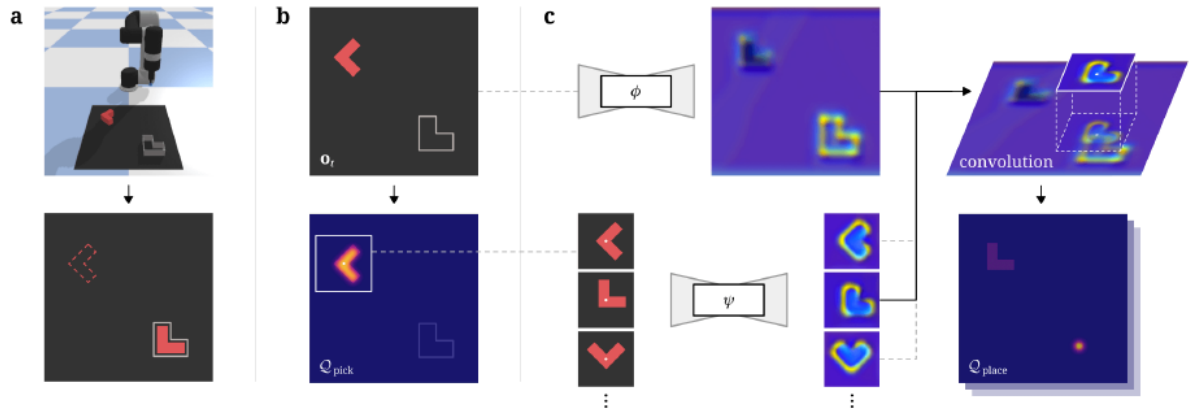


Figure 3.2.: In the example setting **(a)** where the task is to pick up the red L-shaped block and place it into the fixture, the Transporter recovers the distribution of successful picks **(b)**, and distribution of successful placements **(c)** conditioned on a sampled pick. For pick-conditioned placing **(c)**, deep feature template is matched with a local crop around the sampled pick as the exemplar. Different rotations of the crops around the pick are used to determine the best rotation.

$\mathcal{P}(\mathcal{T}_{start}, \mathcal{T}_{end})$. By training observation-action pairs are uniformly sampled from $\mathcal{D}$, and two points $\mathcal{T}^t_{start}$ and $\mathcal{T}^t_{end}$ unpacked from a_t are used to generate two binary one-hot pixel maps $Y_{start} \in \mathbb{R}^{H \times W}$ and $Y_{end} \in \mathbb{R}^{H \times W \times k}$ respectively. The loss function can thereby be the cross-entropy between these one-hot pixel maps and the outputs of starting and ending points Y of the models as

$$\mathcal{L}_{transporter} = -\mathbb{E}_{Y_{start}}[log\mathrm{Y}_{start}] - \mathbb{E}_{Y_{end}}[log\mathrm{Y}_{end}] \tag{3.10}$$

3.3.2. CLIPort

Works in robot manipulation like the Transporter have shown that end-to-end networks can learn dexterous movements that require precise spatial reasoning. However, these methods also face the problem that they fail to generalize to new goals or they cannot quickly learn transferable concepts across different tasks. At the meanwhile, huge progress is made in learning generalizable semantic representations for vision and language which may in contrast be lack of the spatial reasoning. In this context the CLIPort [20] thereby combines CLIP [21] and Transporter from these two ideas to build a language-conditioned agent with both sematic and spatial understanding capable of solving language-specified tabletop tasks.

CLIP introduces a method of pairing images and texts of the similar meaning based on the embeddings of the image encoder and the text encoder. The text encoder is a basically a text transformer or a Continuous Bag of Words model (CBOW) and the image encoder can either be a vision transformer or a CNN. For a batch of N $(image, text)$ pairs, the text encoder and the image encoder are trained to maximize the cosine similarity of the image and text embeddings of the N real pairs in the batch while minimizing the cosine similarity of the embeddings of the rest $N^2 - N$ incorrect pairings by symmetric cross-entropy loss which

Algorithm 5: SE(3) Transporter

Input: Demonstrations with observation-action pair sequences: $\mathcal{D} = \{\zeta_1, \zeta_2, \ldots, \zeta_n\}$
Initial FCNs to infer primitive poses: ω, ψ, ϕ, $\psi^{'}$, $\phi^{'}$

Output: FCNs: ω, ψ, ϕ, $\psi^{'}$, $\phi^{'}$

while *not done* **do**
  Initialize $\mathcal{L} \leftarrow 0$;
  for ζ_i **in** $\mathcal{D}$ **do**
    Calculate $\mathcal{T}_{start}$ using equation (3.6) ;
    Calculate $\hat{\mathcal{T}}_{end}^{SE(2)}$ using equation (3.8) ;
    Calculate other dimensions r_{xz}, r_{yz}, z using equation (3.9) ;
    Calculate SE(2) loss $\mathcal{L}_{transporter}$;
    Calculate loss for dimensions in SE(3) $\mathcal{L}_{r_{xz},r_{yz},z} = \sum_{d=r_x z,r_y z,z} Huber(d, \mathrm{Y}_d)$;
    Add losses $\mathcal{L}_{SE(3)} = \mathcal{L}_{transporter} + \mathcal{L}_{r_{xz},r_{yz},z}$;
    Accumulate $\mathcal{L} += \mathcal{L}_{SE(3)}$;
  end
  Update FCNs by taking a gradient step w.r.t $\mathcal{L}$;
end

calculates the cross-entropy loss for text and image embeddings separately, and then takes the average.

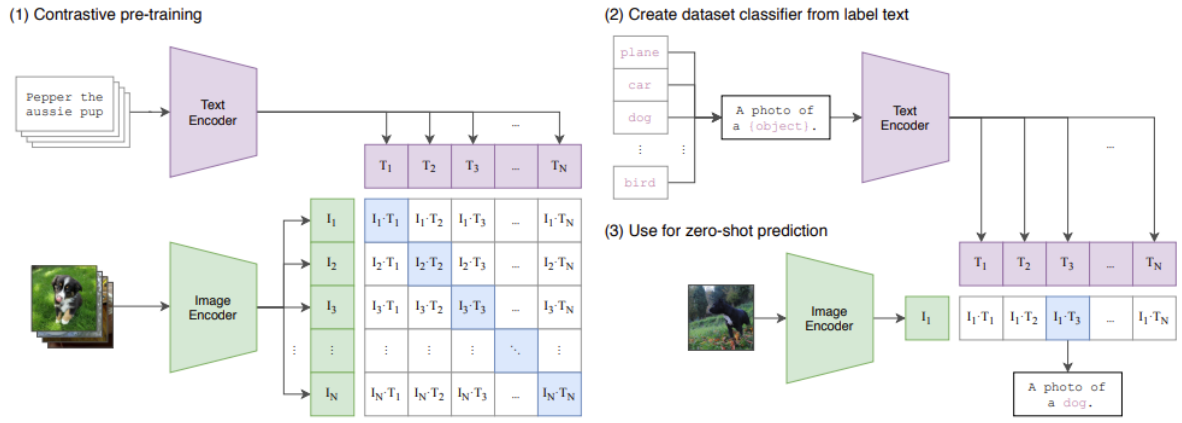


Figure 3.3.: Overview of CLIP

Following the idea of SE(2) Transporter, CLIPort also uses primitives based on $(\mathcal{T}_{start}, \mathcal{T}_{end})$ for describing and manipulating robot movement, while further letting the policy be language-conditioned as $\pi_{start}(\mathcal{T}_{start}|o_t, l)$ and $\pi_{end}(\mathcal{T}_{end}|o_t, l)$. In order to integrate the CLIP network, CLIPort adopts the two-stream FCN architecture with two pathways, namely semantic and spatial. The semantic stream is an FCN based on the CNN-type CLIP image encoder conditioned with language features at the FCN bottleneck which come from the transformer-type

CLIP text encoder and then fused with intermediate features from the spatial stream at the up-sampling part. The spatial stream is identical to the ResNet FCN structure in Transporter, taking in RGB-D input o_t and outputting dense features. Since in SE(2) Transporter there are three FCNs for starting point, query and key, in CLIPort thereby exist also three two-stream FCNs for each role accordingly, where each spatial stream is combined with the identical semantic stream.

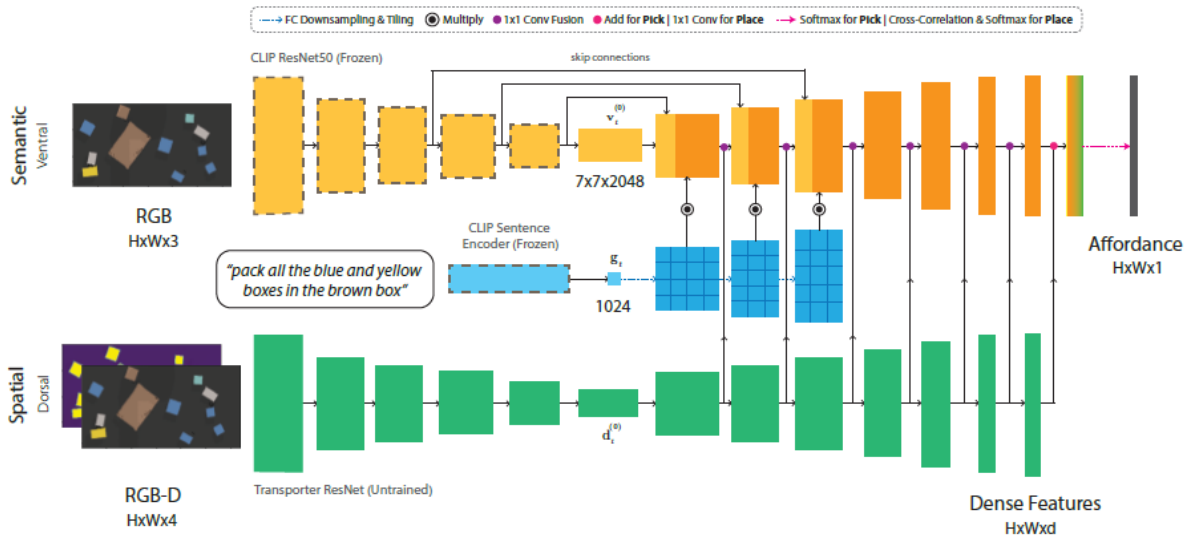


Figure 3.4.: Two-stream FCNs of CLIPort

Suppose using demonstration dataset $\mathcal{D} = \{\zeta_1, \zeta_2, \ldots, \zeta_n\}$, where $\zeta_i = \{(o_0, a_0, l_0), (o_1, a_1, l_1), \ldots\}$ is a sequence of one or more discrete-time observation-action-instruction pairs (o_t, a_t, l_t). The value-action functions can be defined similar to those in Transporter by replacing o_t with $\gamma_t = (o_t, l_t)$. During training two one-hot pixel encodings with k discrete rotations $Y_{start} \in \mathbb{R}^{H \times W}$ and $Y_{end} \in \mathbb{R}^{H \times W \times k}$ are generated from uniformly sampled (o_t, a_t, l_t), and all two-stream FCNs are optimized by loss

$$\mathcal{L}_{CLIPort} = -\mathbb{E}_{Y_{start}}[log Y_{start}] - \mathbb{E}_{Y_{end}}[log Y_{end}] \tag{3.11}$$

where $Y_{start} = softmax[Q_{start}((u, v)|\gamma_t)]$, $Y_{end} = softmax[Q_{end}((u', v', w')|\gamma_t, \mathcal{T}_{start})]$.

3.3.3. LoHoRavens

Based on Ravens framework used in both Transporter and CLIPort, LoHoRavens [10] makes a further progress to improve the success rate of CLIPort in dealing with long-horizon tasks which usually contain more than 5 sub-steps. In CLIPort the CLIP text encoder transformer has the ability to connect the semantics of words with the meaning of images, but still lacks the ability to make long-horizon plans. On the other side LLMs show significant advancements on instruction following and reasoning, thus inspiring LoHoRavens to combine LLMs and the CLIPort for more complex tabletop manipulation tasks.

Algorithm 6: CLIPort

Input: Demonstrations with (γ_t, a_t) pair sequences: $\mathcal{D} = \{\zeta_1, \zeta_2, \ldots, \zeta_n\}$
Initial two-stream FCNs to infer primitive poses: ω, ψ, ϕ
Output: two-stream FCNs: ω, ψ, ϕ

while *not done* **do**
  Initialize $\mathcal{L} \leftarrow 0$;
  for ζ_i **in** $\mathcal{D}$ **do**
    Calculate $\mathcal{T}_{start}$ using equation (3.6) ;
    Calculate $\mathcal{T}_{end}$ using equation (3.8) ;
    Calculate $\mathcal{L}_{CLIPort}$ using equation (3.11) ;
    Accumulate $\mathcal{L} += \mathcal{L}_{CLIPort}$;
  end
  Update two-stream FCNs by taking a gradient step w.r.t $\mathcal{L}$;
end

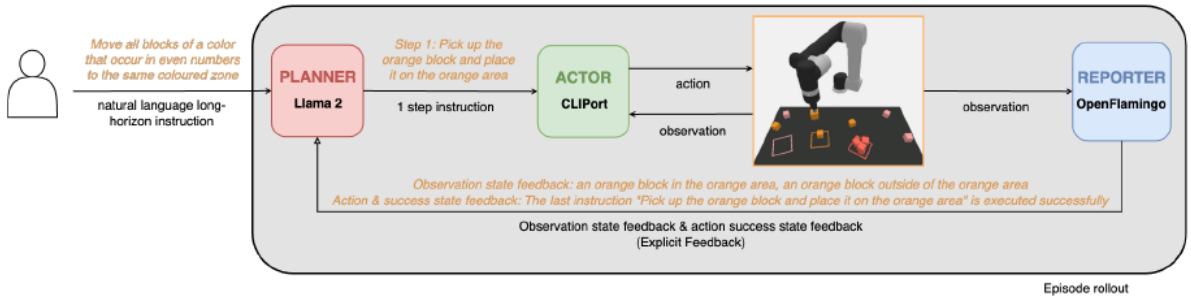
During training the whole process is identical to CLIPort, but at evaluating period LoHo-Ravens introduces an additional LLM as planner ahead of the CLIPort text encoder. Thanks to its reasoning ability, the LLM is prompted to take as input the high-level long-horizon language goal, and output a sequence of low-level simple subtask language goals which the CLIPort text encoder is used to deal with. After the CLIPort has finished a subtask, a reporter which is either a vision language model (VLM) or image encoder will then report the current environment state back to the planner, enabling the planner to adjust the plan in real time.

For VLM reporter, the report content consists of two parts in natural language:

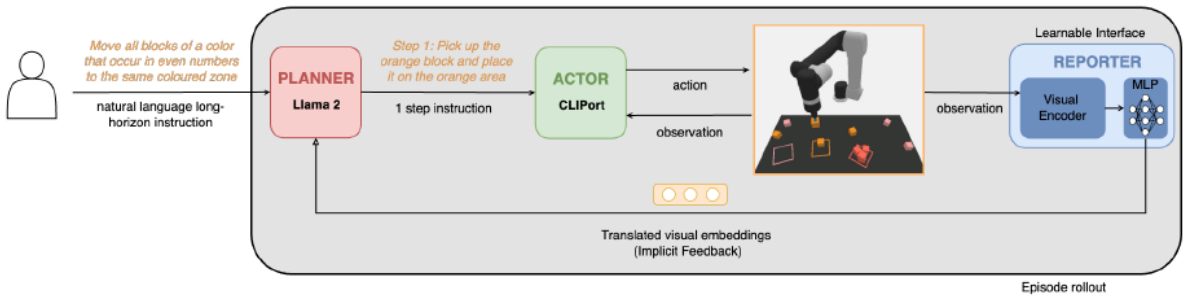
1. The observation state feedback which describes the changes of states of all the objects in the environment.

2. Action and success feedback which will tell the planner what action the agent has taken and whether it is executed successfully referring to the planner's last instruction.

For the other type, the reporter is composed of the CLIP image encoder and a single-layer MLP which translates the image embedding to planner LLM's token embedding space.



(a) explicit feedback



(b) implicit feedback

Figure 3.5.: In both ablations the planner takes in as input the language instruction and provides a single step instruction to the actor (CLIPort). The actor provides action policies and the results of those actions are observed by reporter. For **(a)** the reporter reports explicit captions of observation state. For **(b)** the reporter reports the translated visual embeddings.

Algorithm 7: LoHoRavens (evaluation)

Input: Long-horizon task goal in natural language: L
Initial environment state including objects' information: O_0

while *not done* **do**
  Planner derives subtask l_i from task L based on O_i ;
  CLIPort as agent outputs primitives $\mathcal{P}_i = \{p_0, \ldots, p_n\}$;
  Robot arm manipulating based on $\mathcal{P}_i$;
  Reporter reports current state O_i to planner ;
end

4. Methodologies

4.1. Limitations of Current Works

As in previous chapter discussed, CLIPort combines CLIP and Transporter to let tasks composed of motion primitives be conditioned on natural language goals, and LoHoRavens further embeds the whole CLIPort into a closed-loop control framework consisting of a planner at head and a reporter at tail to improve the performance of CLIPort on complex long-horizon tasks.

Table 4.1.: LoHoRavens tasks and the experimental results of the two baselines.

	LoHoRavens Tasks		Explicit feedback			Implicit feedback
			CLIPort (oracle)	+Llama 2	+Open Flamingo	LLaVA
Seen tasks	A.	Pick-and-Place primitives	61.2	67.3	67.3	67.3
	B.	"Put the blocks in the bowls with matching colors"	19.7	27.9	31.4	37.0
	C.	"Stack smaller blocks over bigger blocks of the same color"	12.1	17.5	18.0	22.1
	D.	"Stack all the blocks in the [ABS_POS] area"	22.5	28.8	30.4	35.8
	E.	"Move all blocks of a color that occur in even numbers to the same colored zone"	13.4	9.1	9.6	8.2
Unseen tasks	F.	"Put the blocks in the bowls with mismatching colors"	17.3	24.8	28.5	21.1
	G.	"Stack blocks of the same size"	2.1	15.8	21.9	14.7
	H.	"Stack blocks in alternate colors"	1.8	8.7	13.2	5.2
	I.	"Stack blocks of the same color in the zone with same color, with the bigger blocks underneath"	8.5	13.6	12.8	11.7
	J.	"Move all the blocks in the [ABS_POS] area to the [ABS_POS] area"	15.1	19.7	27.4	27.2
	K.	"Stack blocks of the same color"	6.7	3.5	4.0	6.8

Table 4.1 shows the evaluation results of vanilla CLIPort and LoHoravens with different reporter configurations on various seen and unseen long-horizon tasks. Here, a seen task means that this task is used for both training and evaluating, and an unseen task will remain unknown for the CLIPort agent until the evaluation. In the seen tasks, LoHoRavens models generally outperform CLIPort. For instance, in task B, where the goal is to "Put the blocks in the bowls with matching colors," CLIPort scores 19.7%, while the LoHoRavens models with explicit feedback 31.4%. The implicit feedback model achieves the highest score of 37.0%. This trend of superior performance by LoHoRavens continues across most seen tasks, indicating that the LoHoRavens framework has a more robust understanding as well as planning of the tasks they have been trained on.

For unseen tasks, the difference between CLIPort and LoHoRavens becomes even more pronounced. CLIPort struggles significantly with particularly low scores in tasks G ("Stack blocks of the same size") and H ("Stack blocks in alternate colors"), scoring only 2.1% and 1.8%, respectively. The main reason is that training and evaluating in the same environment

will lead to overfitting problems. Since the configuration of objects, including color, size, or position, only differs slightly in the same task environment, and the language instructions only maintain the same sentence structure, the CLIPort agent in the task may only establish a set of logical connections between actions and specific sentence structures, rather than establishing a macroscopic logical connection between visual observations and instruction semantics. For example, during training, when the agent receives the instruction to "put the red block into the bowl of the corresponding color", the agent may learn to directly grab the red square with the end effector and put it into the red bowl via a primitive with the red blocks' position and the red bowl's position, without really understanding what "corresponding color" means. On the contrary, LoHoRavens makes a larger progress than in seen tasks, indicating the improved ability to generalize. This progress may be attributed to LLM's ability to analyze language instructions, transforming abstract descriptions into more specific semantics.

It is obvious that LoHoRavens models, especially those equipped with explicit feedback mechanisms, show clear advantages over CLIPort. They are not only better at executing seen tasks but also demonstrate superior generalization abilities when faced with new, unseen tasks. This proves that by introducing planner and reporter as assistants the whole model can learn more flexible representations and adapt their knowledge to new situations, making them more versatile and capable in dynamic environments. However, it also reveals certain limitations. Even though the LoHoRavens models outperform CLIPort in some tasks, particularly unseen ones like task H, LoHoRavens still shows unsatisfying scores around or lower than 30%, suggesting that there is great room for further improvement. This may be because although the assistants have been added, it is still the CLIPort that receives the subtask instructions and gives the location of the primitives. Based on how well the CLIPort is trained, its FCN networks may have difficulty calculating the correct positions, or the CLIP text encoder may still fail to extract semantics from simplified instructions. Besides the problem of generally low success rate, the model using explicit feedback based on VLM Open Flamingo outperforms the one without feedback only slightly but occupying nearly double the calculating resource, while the model using implicit feedback based on LLaVA does not consistently outperform the one without feedback and for certain tasks it even performs worse, which indicates that the feedback method could be refined for better performance.

4.2. Method Outline

To address the above issues and increase the generalization abilities of the model, we propose CapRavens, a novel approach to leverage the reasoning ability of LLMs to directly set high-level task plan and distill to low-level manipulations, making the whole manipulating process more consistent and robust.

Referring to the low success rate of CLIPort on seen and unseen tasks, it is difficult to map the complex long-horizon language instruction into a sequence of actions based on the current observations. On the other hand, the improvement of LoHoRavens confirms the advantage of LLMs in handling long-horizon instructions. Moreover, inspired by Code as

Policy [9], that the output of an LLM based on state input can be used directly as a policy, we intend to completely discard the CLIPort network in the LoHoRavens framework and instead use the planner LLM as the agent. This LLM will give a strategy for the task with low-level code as the carrier according to the input long-horizon instructions and the initial environment state. Unlike Lohoravens, which goes through a planner-agent-reporter loop for every single primitive, the planner in the new approach will try to complete all primitives at once, and finally hand them over to the reporter to check the completion status and then get the feedback acting as a state. After that the planner will decide on subsequent actions based on the report, such as remedying the failed primitives. In this manner, we hope CapRavens can achieve a higher success rate on both seen and unseen tasks as well as a better generalization ability but doesn't need any training.

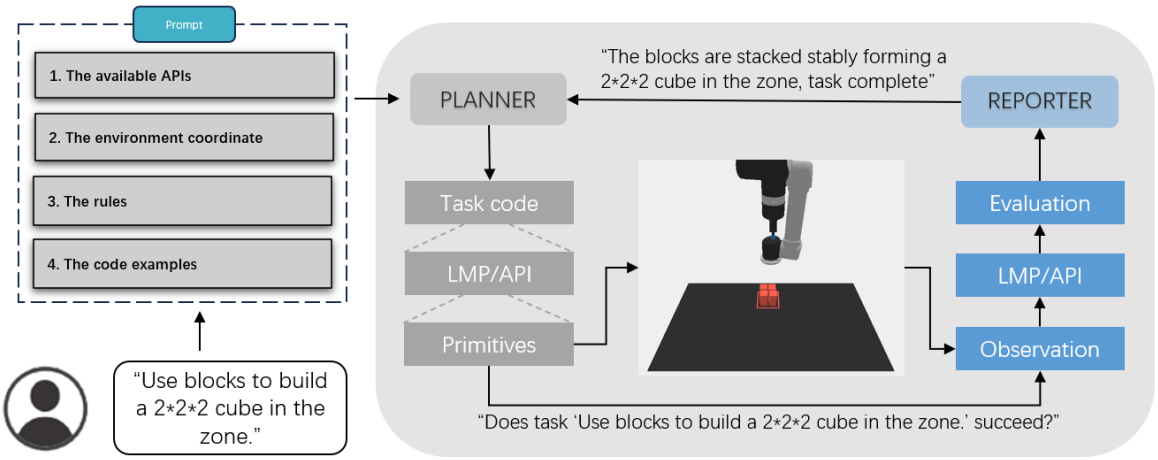


Figure 4.1.: The planner writes task code including various LMPs and APIs based on language instructions, driving motion primitives to do robot manipulations, while the reporter outputs the scene description and task evaluation from observation states with the help of LMPs and APIs.

4.3. Generating Code as Plan

From LoHoRavens it can be seen that the pre-trained LLM can make plans according to a task description which could be abstract or complex. In order to make the output of the LLM be able to directly drive the robot, all plans should be based on the Application Programming Interfaces (APIs), which act as a bridge connecting the planner and the robot simulation environment. That is to say, for each task, the plan output by the LLM is a complete code containing the underlying low-level auxiliary APIs, which control the robot to complete all actions.

4.3.1. Auxiliary APIs

In CLIPort, the networks calculate the best poses of a certain primitive based on both text and image inputs. For CapRavens, the planner LLM can principally use both text and image as input, however, the planner cannot directly infer the accurate scene information from the image. This requires a series of auxiliary APIs to be designed first to make up for the lack of information caused by the unreliability of image input. When LLM is planning for a task, it should actively call these APIs to obtain the environment information it needs.

get_obj_names()

Since the planner will only receive a long-horizon task description at the beginning, in order to plan the specific steps of the task, the planner intuitively needs an API, **get_obj_names()** to obtain all the objects in the current environment. In addition, because of the task's complexity, the API should clearly specify the attributes of each object when returning, such as color, shape, whether the object is fixed, etc., and these objects are to be ideally placed in an array-like or dictionary-like format. This also requires the API to configure a unique number for each object, so that the planner can distinguish even if there are multiple identical objects in the environment.

get_obj_pos(obj), get_obj_rot(obj)

Also due to the omission of image input, even if the planner knows what objects are in the scene, it is difficult to locate each object, let alone formulate primitives. In CLIPort, the environment orthogonally projected from three dimensions to two dimensions is gridded to facilitate the networks to determine the start pose and end pose of the primitive. In CapRavens, an API, **get_obj_pos(obj)** is instead required to obtain the position of a specific object **obj**. Since the task is in a three-dimensional environment, it is best for this API to return the geometric center or center of gravity position (x, y, z) of the object in a three-dimensional Cartesian coordinate system to improve the planner's understanding.

In addition, the network in CLIPort can handle the rotation of objects in a discrete manner. Similarly, CapRavens also needs an API, **get_obj_rot(obj)** to obtain the SE(3) rotation of object **obj** around its own center. Since the coordinate system is introduced, the rotation in CapRavens can be continuous. It is best for this API to return the Euler angle or quaternion of the object. The former is easier for the planner to understand, while the latter can more accurately describe the rotation.

get_bbox(obj)

Since the position of an object is replaced by the position of a point that may be inside or outside the object, it is still difficult for the planner to plan primitives based on the coordinates of these points alone, because objects usually have different sizes and shapes that are difficult to quantify. Therefore, CapRavens needs an API, **get_bbox(obj)** to obtain the bounding box of object **obj** to handle the problem of interference between objects in a simplified way but

without sacrificing too much accuracy. The API should return two points, in total six values as $(x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max})$ which correspond to the minimum and maximum corners of a bounding box that is aligned with the axes in the Cartesian coordinate system.

denormalize(pos)

In CapRavens, a three-dimensional Cartesian coordinate system is used to describe the position in the environment. However, depending on the changes in the environment, the direction and size of the basis vectors in the coordinate system may not be fixed. For example, in one environment the x-axis may point toward the robot and each unit length corresponds to one centimeter, while in another environment the y-axis points toward the robot and each unit length corresponds to one decimeter. To deal with this situation, CapRavens asks the planner to use a standard normalized coordinate system, where the total length of each axis is 1 unit, when it is dealing with relatively complicated spatial reasoning problems. This makes it easier for the planner to process orientation information. After the planner gives the normalized coordinates **pos**, the API **denormalize(pos)** will use linear transformation to denormalize the coordinates **pos** to get the actual environment coordinates.

is_target_occupied(targ)

In the initial environment, CapRavens assumes by default that all objects are randomly placed. This creates a problem that the target positions derived from the task description may have already been occupied by other objects. If the primitive is run directly without the target position being cleaned up, the primitive is likely to fail. In some cases, multiple primitives are related. For example, the primitive that places the upper object in the stacking task depends on the success of the primitive that places the lower object. For the above reasons, occupied target positions often lead to the failure of the entire task. Therefore, CapRavens needs an API **is_target_occupied(targ)**, to check whether the target **targ**, which can be an object name or position, is occupied by any object. The API should return the names of all objects that occupy the position in an array-like format.

get_random_free_pos(targ, area)

After the planner knows which objects occupy the target position, it needs to use primitives to clear these obstacles. However, the primitive needs to specify the starting and ending positions, where the starting position can be obtained through **get_obj_pos()**, but the ending position is uncertain, because theoretically these objects can be placed anywhere in the environment without affecting other primitives. To address this issue, CapRavens implements an API, **get_random_free_pos(targ, area)** to select a free position for the planner that is within region **area** which has the same format as the bounding box, does not occupy the target position **targ**, and is not occupied by any other objects. The API ought to return a denormalized position plus a rotation.

put_first_on_second(obj1, obj2)

CapRavens intuitively needs an API to run a primitve to manipulate a robot either in a simulated environment or in the real world. This API should accept two inputs, a start pose as well as an end pose, and then make the robot move. In a desktop manipulation environment, the pick-place primitve is the most common one and sufficient for most tasks. Therefore, for simplicity, the API **put_first_on_second(obj1, obj2)** allows the planner to place an object **obj1** directly on top of another object **obj2**. And of course, **obj2** should also be replaceable with a pose consisting of a position and a rotation to deal with situations where **obj2** is not needed.

4.3.2. The Language Model Program

With the aforementioned basic APIs, the planner can theoretically use them to indirectly obtain omitted visual environment information. However, long-horizon task descriptions may contain complex attributes to modify objects or positions, such as "the third block from the left to the right that is different in color from the rightmost block but the same size", or "the three points on the circumference of a circle with the center of the left front area of the desktop as the center and the length of the smallest block on the table as the radius, dividing the circle into three equal parts". Although the planner can convert such complex descriptions into specific object names or coordinates through the APIs, it also has the important task of formulating the entire task plan and arranging primitives. Therefore, as the complexity of the task increases, the planner's load will increase significantly, and its output will be very lengthy.

To solve this problem, CapRavens separated some of the work like handling objects, poses, etc. from the planner and handed it over to other LLMs. This system functions just like the API, and these LLM-driven APIs that assist the planner in processing information are called language model programs (LMPs). When the planner encounters a difficult problem, it will try to call the corresponding LMP, and the LMP will then call the basic APIs to obtain the data and finally return the answer to the planner. The following are the main LMPs:

- **parse_obj_name(dsc, ctxt)**: The LMP take as input one description **dsc** and one context **ctxt**. Here **ctxt** is all the objects that the LMP has the right to see, and **dsc** is the natural language description of the target object that the planner desires. What the LMP has to do is to filter out the object names that match the description in the context and return them.

- **parse_position(dsc)**: The LMP accepts a natural language description **dsc** of an implied location and returns one or more position coordinates or positions plus rotations based on the planner's requirements in the description.

- **parse_function(dsc)**: This special LMP is used to handle the API newly created by the planner and cannot be called directly by the planner. Instead, it is triggered automatically when an undefined API appears in the planner's answer. It will try to complete the implementation of the API based on the planner's description of which.

As the result, the planner's whole statement for the newly created API, in a form as '**var=new_api(arg)**', will be treated as the input **dsc** of the LMP.

Since an LMP is an API based on LLM, its core can be understood as the prompt to LLM and the extraction of LLM's answer. For the prompt part, CapRavens adopts the prompt form of base prompt plus query, where query is the arguments of LMP, and base prompt includes system prompt and user prompt. Because the output of planner should be code that can be run directly, the output of each LMP should also be in the form of code. Based on this principle, the system prompt part is simply defined as

'**You are a task planning assistant who only answers with Python code.**',

which briefly describes the work to be done by the LLM and constrains its output form to Python style.

The user prompt is used to tell LLM how to complete the task assigned to it in a more specific way. To be consistent with the system prompt, the user prompt also uses the Python code format throughout. In the first part of the prompt, like the usual Python code, there are imports of local and third-party libraries. The local library includes the APIs mentioned above and some utils in the simulation environment, while the third-party library simply includes

- numpy: A fundamental library for scientific computing, providing support for multi-dimensional arrays and matrices, along with a collection of mathematical functions.
- shapely: A library for geometric operations and analysis, particularly useful in spatial data analysis.
- itertools: A standard Python module that provides a set of fast, memory-efficient tools for creating and working with iterators, allowing to loop through data in various useful ways.

to help LLM solve tasks more conveniently. The LLM is allowed to use these imported libraries to write code to complete tasks.

```
import numpy as np
from env_utils import get_obj_pos, get_obj_rot, parse_position
```

Figure 4.2.: Example of importation that tells the LLM the methods it can directly use, which can also include other LMPs

After importing the libraries, though the LLM may know how to use the commonly used third-party libraries, it still knows nothing but the name about the local libraries and utils in the environment. Therefore, in the second part of the user prompt, these local APIs will be introduced to LLM. Since the APIs are pre-packaged just like third-party libraries, the LLM does not need to know how each API is implemented. It only needs to understand the function of each API, the arguments and formats that need to be passed in, as well as the variables and formats of the API output.

```
# ------------------------------------------------------------------------
# Existing Method Explanations
# ------------------------------------------------------------------------
'''
get_obj_pos(obj) -> [list]
# return a list of len(obj) of 3d position-vectors of obj
get_obj_rot(obj) -> [list]
# return a list of len(obj) of 4d quaternion orientation-vectors of obj
parse_position(query=arg) -> list or tuple
# usually return one pose or several poses as a list based on description arg
'''
```

Figure 4.3.: Example of API introduction, which emphasizes the data format of arguments and return values

We mentioned earlier that the planner can call the API **denormalize(pos)** when it encounters complex spatial understanding problems, and directly use the standard normalized coordinate system it is accustomed to. The so-called complex problems here refer to those that require the calculation of multiple coordinates. However, in most cases, the coordinate system is implicit in the environment, that is, the proposer of the task may not know the true coordinates of a certain location in the current environment. In this case, the proposer will tend to use himself as a reference and use more common directional words in daily life to describe the task. Furthermore, the orientation usually only means the comparison of values on certain coordinate axes. Since it does not involve many coordinate operations, we hope that the LLM will not frequently use **denormalize(pos)** and perform coordinate conversions, but directly perform comparison operations in the current coordinate system. Therefore, in the third part of the prompt, the orientation of the current environment and the corresponding coordinate axes are provided to the LLM.

In previous parts of the user prompt, in order to correspond to the system prompt and emphasize the Python format of the conversation, the text of each part is basically directly presented in Python code or in a key-value format similar to a dictionary, and there is no explanation for the overall prompt. Although some LLMs can understand the formatted prompt without additional instructions, for the stability of the LLM output, in the next part of the prompt some more detailed rules are still told to the LLM in the form of natural language code comments, including telling LLM what problem it is currently solving, and reminding it to refer to the above-mentioned APIs and coordinate orientations.

After telling the LLM how to use each API and explaining the environment as well as the tasks, the LLM can actually already give output based on the arguments passed into the LMP. However, the LMP itself is only called when the planner is working, and its output must be accepted by the planner. Therefore, the planner has certain requirements for the code format of each LMP. On the other hand, we hope that the LMP will still be reliable enough when facing more complex problems. For the above reasons, we used the few-shot learning

```
# ------------------------------------------------------------------------
# Orientations in Coordinate System
# ------------------------------------------------------------------------
'''
left: y-
right: y+
front: x+
rear: x-
top: z+
bottom: z-
top left: x-y-
bottom left: x+y-
top right: x-y+
bottom right: x+y+
'''
```

Figure 4.4.: Example of orientation explanation, which describes the relationship between common orientation expressions and coordinate axes

```
# ------------------------------------------------------------------------
# General Requirements
# ------------------------------------------------------------------------
'''
You are writing python code for object parsing, refer to the code style in
examples below.
You can use the existing APIs above, you must NOT import other packages.
'''
```

Figure 4.5.: Example of overall requirements.

technique [22] and wrote several example task codes with increasing difficulty in the prompt, so that the LLM can learn the pattern of how to complete LMP tasks and generalize this pattern to handle even more complex tasks.

```
# ------------------------------------------------------------------------
# Task Examples
# ------------------------------------------------------------------------
objects = ['brown bowl', 'banana with obj_id 1', 'brown block with obj_id 9',
'apple', 'blue bowl with obj_id 8', 'blue block with obj_id 3']
# the block, return result as list.
ret_val = ['brown block with obj_id 9', 'blue block with obj_id 3']

objects = ['brown bowl', 'green block', 'brown block', 'green bowl',
'blue bowl', 'blue block']
# the left most block.
block_names = ['green block', 'brown block', 'blue block']
block_positions = get_obj_positions_np(block_names)
left_block_idx = np.argsort(block_positions[:, 1])[0]
left_block_name = block_names[left_block_idx]
ret_val = left_block_name

objects = ['fixed yellow zone with obj_id 5',
'movable purple block with obj_id 6',
'movable green block with obj_id 7', 'movable blue block with obj_id 8',
'movable silver block with obj_id 9', 'movable cyan block with obj_id 10']
# the blocks to the left of the purple block with horizontal distance
# larger than 0.1.
block_names = ['movable purple block with obj_id 6',
'movable green block with obj_id 7', 'movable blue block with obj_id 8',
'movable silver block with obj_id 9', 'movable cyan block with obj_id 10']
purple_block_pos = get_obj_pos('movable purple block with obj_id 6')[0]
use_block_names = []
for block_name in block_names:
    if get_obj_pos(block_name)[0][1] + 0.1 < purple_block_pos[1]:
        use_block_names.append(block_name)
ret_val = use_block_names
```

Figure 4.6.: A small set of task examples of increasing difficulty for LMP **parse_obj_name()**.

Finally at the end of the prompt is the information about the current task, which is essentially the arguments given to the LMP. Putting it at last to some extent emphasizes to the LLM that this task is what it should now focus on without being distracted by the various example tasks above.

```
# ------------------------------------------------------------------------
# Question
# ------------------------------------------------------------------------
objects = ['blue block', 'cyan block', 'purple bowl', 'brown bowl']
# the block in the bottom right area.
```

Figure 4.7.: Example of the current task for LMP **parse_obj_name()**.

4.3.3. The Main Planner

Based on the previous description of the LMPs, the main planner can actually also be regarded as an LMP. It accepts the object list and task description of the current environment as input, and then outputs a complete task execution code containing APIs and LMPs. Therefore, the prompt of the main planner is highly similar to that of other LMPs. Its system prompt is exactly the same as the LMP, that the user prompt also imports various third-party packages and local libraries at the beginning, and then explains each local API. The only difference from LMP's prompt is that the planner has access to more APIs. Then for the rest part of the prompt there is also an explanation of the current environment coordinate direction, several detailed rules for the answer content, a bunch of example tasks, and finally the current task.

In the example task part, the planner's prompt likewise uses the few-shot learning technique and shows more examples than those for other LMPs. However, considering that the planner needs to plan the entire long-horizon task, and the problem it handles is more complex than that of the low-level LMP, the pattern learned by the LLM simply by more example tasks is not enough to cope with a variety of complex tasks. When the current task is not similar to all the tasks among the examples, the output of LLM will become very unreliable. On the other hand, considering that task planning requires more logic, the example task part of the planner prompt incorporates the technique of chain-of-thought [23] in the structure of few-shot learning. The concept of chain-of-thought involves explicitly guiding the model to think step-by-step, breaking down the reasoning process to arrive at a solution. It's more about how the model arrives at the answer rather than just finding the right pattern. In this way, it is hoped that the planner can not only handle tasks similar to the examples, but also disassemble and analyze the tasks step by step when encountering brand new tasks, so as to output a more reasonable plan.

4.4. Completion Check

After the planner generates the task code, the execution of the code is completely handed over to the environment. Therefore, the planner does not know whether each primitive is actually executed as expected. Any small disturbance may cause the object to deviate from its position. For example, in a stacking task, the object stacked later may cause the entire stacking structure to collapse, resulting in the failure of the entire task. In other words, the planner itself lacks the ability to react to the environment. To further improve the task completion rate,

the planner-reporter framework in LoHoRavens is followed. However, unlike LoHoRavens, the reporter in CapRavens does not provide feedback for each primitive, but only provides feedback of the environment status to the planner once all primitives have been executed. The main reason is that the CLIPort in LoHoRavens outputs primitive step by step, while CapRavens outputs the code implicitly containing all primitives at once. Specifically, in order to interact with the main planner, the reporter is also an LMP, **parse_completion(dsc, ctxt)**. It accepts the task goal description **dsc** and the scene observation **ctxt** as input, and then outputs a Boolean value indicating whether the task is completed based on the current scene.

For LMP based on the LLM, text is the information that it is best at processing. Therefore, an ideal way to describe environmental observations is to correspond each object in the environment to its posture to form a dictionary-like structure. Like the other LMPs mentioned above, the reporter also needs to be controlled by a set of prompts. The system prompt and user prompt are basically the same as other LMPs, except that they differ in the accessible APIs, overall rules, and examples. Since the input environmental observation contains the position of each object and is quantitative, we hope that the reporter will not directly output the result from on the input, but first establish the code based on the input, which is equivalent to a deduction, and then use this deduction to determine whether the task is successful or not, so as to improve the reliability of the judgment. From this perspective, the work of the reporter is very close to that of the planner, so in the example part of the user prompt, the method of few-shot learning plus chain-of-thought is still used.

Another way to build a reporter is to rely on the VLM like LoHoRavens. In this way, instead of quantitative object postures, the reporter LMP can directly use images as input. By analyzing the relative positions of objects in the image, a judgment on whether the task is successful or not is given. In this case, the input environmental observation will contain four base64-encoded pictures, which are RGB images of the initial state and the final state taken from a fixed camera angle, and the depth map corresponding to each of the two images. This borrows the idea of CLIPort, under which the VLM will distinguish the category of objects through RGB images, and further distinguish the posture of objects through depth maps. Since the VLM directly gives judgments by analyzing images, a series of instructions in the user prompt are omitted, and only the last part containing the description of the current task is retained to tell VLM what the task goal to be checked is. In the system prompt, it still contains the work that VLM should do and the output format, just like the aforementioned other LMPs.

When the reporter determines that the task has failed, since the reporter only returns a Boolean value to the planner, the planner does not know exactly which objects have been successfully placed and which have not, so for simplicity the planner will clean up the entire environment and re-output new task code to execute the primitives again. Within the specified number of loops, the reporter will make a judgment every time the planner generates and runs the task code until the reporter determines that the task is successfully completed.

5. Experiments

5.1. Ravens Framework

Despite abandoning the CLIPort network, CapRavens still uses the Ravens framework on which the CLIPort is based for the simplicity on establishing and evaluating new tasks. The original purpose of Ravens in CLIPort is to train and evaluate the learning of language-conditioned continuous control policies carried by motion primitives. For CapRavens, although training process is no longer required, we still use Ravens to build long-horizon tasks and use its evaluation function to test the success rate of the entire system. A full task solving process mainly involves the following three components in Ravens.

5.1.1. Ravens Environment

The Ravens environment is basically based on PyBullet and OpenAI gym. The former provides a physical simulation environment, while the latter provides a standard API to communicate between learning algorithms and environments. Because the Ravens environment is based on PyBullet, it is possible to add any scene objects in URDF (Unified Robotics Description Format) or SDF (Simulation Description Format) format, which are basically XML specifications, through PyBullet's built-in APIs. When adding objects, the size, mass, color and other properties of the object can also be adjusted by using other built-in APIs or by directly modifying the XML [24]. By modifying the parameters of the geometry in the object file, such as making the shape point to a mesh OBJ file, more complex shapes for the object can also be customized.

Since the main task of Ravens is to deal with the desktop rearrangement, in the default configuration, the simulation environment has a ur5 robot arm equipped with a suction cup end effector and a finite-sized horizontal plane representing the desktop on which more other task-related objects are to be placed. In addition to the suction cup, the end effector of the robot arm can be replaced with a spatula or a gripper, while the robot arm itself can also be substituted by importing URDFs of other robot models. Different end effectors have different implementations of motion primitives, but in general they all only need to be given a starting and ending position.

With the help of the gym-like architecture, Ravens environment provides plenty of APIs, which can be roughly divided into environment supporting APIs and task-related APIs. The environment supporting APIs mainly include simulation stepping methods and camera rendering methods. The simulation stepping method takes a motion primitive as a step to dynamically simulate the environment with physical effects, while the camera rendering method uses the built-in clock of PyBullet as a benchmark to capture a snapshot of the scene

every several frames. This snapshot can be a normal RBG image, a depth map or even a semantic segmentation map. Note that this snapshot is not the observation or state in imitation learning, and the latter is still obtained every time a primitive is processed. A series of consecutive snapshots can in contrast be used to synthesize a video of the task execution process. The task-related APIs include methods for initializing the scene and methods that assist the APIs mentioned in the previous chapter.

5.1.2. Ravens Dataset

Like most machine learning datasets, the Ravens dataset contains training sets, validation sets, and test sets. For a specific task, all samples in all these sets are generated by the same task file, but with different seeds, so that each task sample in each set is finally different in details. The task file is the most important part for the dataset. Each new task has an independent task file, and similar tasks that are slightly expanded on the basis of old tasks can be newly defined through class inheritance. In the task file, the task name, task goal, maximum number of steps to complete the task, and the robot arm and end effector used in the task are first defined. Then, various objects required for the task are manually added, and finally a series of primitives are also manually defined so that the task goal can be finally completed.

All samples in the three sets can be seen as expert demonstrations. Since the task file specifies all the content needed to complete the task, when trying to generate samples of any set, the Ravens environment agent, namely oracle, only needs to execute the code in the task file step by step till the end, and it will finally generate a sample saved as a sequence of task step information. In order to make each sample unique, the task file uses methods that can generate pseudo-randomness based on seeds from packages such as **random** and **numpy**, for the quantity, position, size, color and other attributes of the objects. The Ravens environment assigns odd number seeds starting from 1 to the training set, even number seeds starting from 0 to the validation set, and large enough seeds (usually >10,000) to the test set, thereby ensuring the uniqueness of samples within a certain number of samples.

Since a motion primitive is used as a step in the Ravens environment, for each primitive in each sample in the Ravens dataset, the Ravens environment will record the starting and ending points of the primitive, the RGB and depth map snapshots, the scene information containing all object poses as well as the language goal of the task, and the step reward obtained by executing this primitive. After all primitives are executed, all the above-mentioned data will be saved as a PKL file in the form of a list. In addition, the seed used for each sample will also be saved, which is especially important for the validation set and the test set.

5.1.3. Ravens Matching Method

In the Ravens task file, all motion primitives required to complete a task are defined. However, the starting and ending points of these primitives are not directly arranged in a list in order, but are associated with each other using a matching matrix. In order to use this matrix, the starting and ending points of all primitives each form a vector. The number of rows in the matrix is equal to the length of the starting point vector, and the number of columns

is equal to the length of the ending point vector. Since the starting and ending points of primitives are usually one-to-one corresponding, the matching matrix is mostly a square matrix. Besides, the matching matrix is also a zero-one matrix, where one represents the corresponding relationship between a starting point and a certain ending point. For example, if the element $A_{23} = 1$ in the matrix, then the second starting point can form a valid primitive with the third ending point. Conversely, if $A_{23} = 0$, it means that the second starting point cannot match the third ending point. Since the position of a starting point in the desktop rearrangement task is always linked to the position of an object, the Ravens environment actually uses an object vector equivalent to the starting point vector. When generating samples of the dataset, the oracle will sequentially retrieve each object in the object vector and find the corresponding row in the matching matrix, then filter out the ending points according to the position corresponding to all the elements with a value of one in the row, and finally the oracle will select the one closest to the starting point among all the end points to match it. Since a point in space can only be occupied by one object at a time, the row and column where the element in the matrix is located will be masked and set to zero after the matching is determined.

During verification and testing, every time the external policy executes a primitive, the oracle will compare the starting point of the primitive, that is, a certain object, with the object vector in the task file, find its corresponding row in the matching matrix, find all feasible end points, and then detect whether the actual primitive's ending point is among them. If so, the oracle will judge this step of the primitive as successful and return a positive step reward. When the total reward reaches the maximum reward or all primitives are successful, the task will be judged as successfully completed.

5.2. Generation of Long-Horizon Tasks

As mentioned earlier, all tasks in the Ravens framework are defined by task files. Thanks to this, Ravens can add to itself an unlimited number of new custom tasks. However, to create a new task is not as simple as just to think of a task goal, but also to manually write in the details of the task steps, task objects, task primitives, etc. according to the structure of the task file specified by Ravens. Based on this and inspired by GenSim [25], since Ravens' task files are highly structured, and the gym-like Ravens environment itself comes with many APIs to control the scene, task generator based on LLMs can be utilized to automatically generate more complete long-horizon task files based on the given task goals.

Just as the planner imitates the examples in the prompt to write the task code, the task generator here also imitates the examples and then writes the task file. For the same reason as the planner and LMP, the system prompt of the task generator is defined as

'You are an assistant designing creative and long-horizon tasks who only answers with Python code.',

which briefly describes the work of the generator and the output format. The task generation methods can be divided into two categories: bottom-up and top-down. In the former method,

the LLM will browse some pre-written task files and then think of a new task based on these tasks; for the latter, the LLM will also browse some pre-written tasks, but will instead write a new task with a human-specified task name based on these tasks. Since the concrete goals and objects used in the task have not yet been determined, the LLM needs to determine these prerequisites before writing the task file. On the other hand, the generator does not have a verifier like a reporter to check its output content. Therefore, unlike the planner using a complete user prompt for a single conversation, the task generator splits the user prompt into multiple parts and finally obtains the content of the task file in the form of multiple consecutive conversations.

Whether for bottom-up or top-down method, at the beginning it's necessary to determine the name of the task, the goal of the task, and the source files used for the task objects, such as URDF files. Therefore, in the first part of the user prompt, a list of all available local object source files is given, followed by a list of some sample task goals. For the bottom-up method, a list of sample task names will continue to be given. Since the subsequent task generation steps require the LLM to output the task name, task goal, and task object source file in Python dictionary format at this step, in addition to describing the requirements in natural language, the few-shot learning technique is also used to provide many examples as output templates. In order to ensure the interestingness and feasibility of the task, and to avoid repeated tasks, the judgment of whether the task is accepted or not based on the above three criteria is also given under each task example. For example, tasks that require sweeping and stacking at the same time will be rejected because the Ravens environment does not support switching end effectors in the same task, or tasks that require stacking blocks on a sphere will also be rejected because it is physically difficult to achieve balance. In this way, the LLM can know what kind of tasks are better and design task goals in a good direction.

Since the task generator essentially does the same task planning work as the planner, after determining the three elements including the task name, task target, and task object source file, the second part of the user prompt also gives the usage instructions of the API required by the task file, the common error book, and the coordinate orientation of the Ravens environment. After that, since the content of the task file is usually long, the prompt will ask the LLM to select several tasks similar to the current task from the existing task library and give their complete task file codes as examples, so that the LLM can learn the pattern of similar task file codes with the help of the few-shot learning technique.

After the task generator writes the task file code, the Ravens environment will try to run the code once without initializing the scene or starting PyBullet, which is used to simply check whether the code has syntax errors. If the execution reports an error, the task generator will receive a prompt consisting of the task file source code and the error traceback, requiring it to modify it. After the modification, the oracle will use an arbitrary seed to try to run and evaluate an episode of the task three times as a runtime test. If one of the three attempts succeeds, the task file will become a candidate and may be included in the task file library. If all three attempts fail, the task file will be invalidated and the task generation will be considered a failure. The candidate task file that passes the runtime test will be prompted back to the task generator for self-reflection. The generator will review the task again according

to the three criteria mentioned above and finally decide whether to add the task file to the existing task file library.

During one generation, the task generator is usually required to generate five to ten task files. For the bottom-up method, the generator will generate multiple tasks with different names, while for the top-down method, the generator will generate multiple tasks with the same name. Therefore, although the top-down method can only generate one task file for one generation period, the generation success rate is greatly improved compared to the bottom-up method.

5.3. Results and Ablations

In this section, we analyze the completion rate of CapRavens on various long-horizon tasks. Ablation studies on the participation of the VLM-based reporter are performed both on LoHoRavens task and LLM generated tasks. In this work, GPT-4o-mini from OpenAI is used as LLM and VLM for all task planning, reporting and generation.

5.3.1. LoHoRavens Tasks

To make a more intuitive comparison with LoHoRavens, CapRavens tests the same or some similar tasks as those from LoHoRavens, such as putting blocks in the bowls and stacking blocks. Since no training is strictly required, here a seen task simply means that the same task appears in the task example part of the user prompt, while an unseen task means the opposite.

Table 5.1 shows the tasks with their goal descriptions. It can be seen that these tasks test the robot agent's ability to understand space, arithmetic, correlation and coreference. Among the tasks, only the first task is the seen task and in all tasks exist objects as distractors that are irrelevant to the task goals. The variables enclosed in square brackets ([]) in the task goal description will change under different task seeds to introduce more diversity.

Table 5.2 demonstrates the comparison between OpenFlamingo assisted LoHoRavens and CapRavens with ablation on the participation of the reporter. Each task is tested on 50 different randomly generated seeds. For the same tasks tested by LoHoRavens, although they are all unseen tasks, CapRavens has significantly improved the task success rate by approximately three times, with all success rates at ca. 80% or higher, both when using and not using reporters, compared to LoHoRavens. For other similar tasks, CapRavens also performs well. Comparing the results of using and not using reporters, it can be seen that adding reporters sometimes counter-intuitively lowers the success rate of CapRavens. This may be because the reporter only reports the success or failure of the task, while the planner responsible for generating the code plan is still the one that really affects the success of the task. In addition, in some tasks that require relatively stricter positioning of objects, the reporter may make misjudgments, or the planner may confuse some concepts when trying again, such as the classification of warm and cold colors, which may also lead to lower success rate.

Table 5.1.: LoHoRavens tasks and completion instructions.

	Task Name	Task Goal
A.	stack-all-blocks-on-a-zone	"Stack all blocks in the [COLOR] zone."
B.	stack-blocks-of-same-size	"Stack blocks of the same size in the [COLOR1] zone and [COLOR2] zone respectively."
C.	stack-blocks-of-same-color	"Stack all the blocks of the same color together in the same colored zone."
D.	stack-blocks-by-color-and-size	"Stack only the [SIZE] blocks of [COLOR_TYPE] color in the [COLOR] zone."
E.	stack-blocks-by-relative-position-and-color	"Stack all the blocks, which are to the [REL_POS] of the [COLOR1] block with [POS_TYPE] distance larger than 0.05 unit, in the [COLOR2] zone."
F.	move-blocks-between-absolute-positions	"Move all the blocks in the [POS1] area to [POS2] area."
G.	move-blocks-between-absolute-positions-by-size	"Move all the [SIZE] blocks in the [POS1] area to [POS2] area."
H.	put-block-into-matching-bowl	"Put the blocks in the bowls with matching colors."
I.	put-block-into-mismatching-bowl	"Put the blocks in the bowls with mismatching colors."
J.	stack-blocks-with-alternate-color	"Stack blocks with alternate colors on the [COLOR1] zone, starting with the [COLOR2] color."

Table 5.2.: LoHoRavens tasks and the experimental results of the ablations.

LoHoRavens Tasks		LoHoRavens (OpenF.)	CapRavens	
			no reporter	with reporter
A.	stack-all-blocks-on-a-zone	\	100	100
B.	stack-blocks-of-same-size	21.9	88	100
C.	stack-blocks-of-same-color	4	96	100
D.	stack-blocks-by-color-and-size	\	64	48
E.	stack-blocks-by-relative-position-and-color	\	42	42
F.	move-blocks-between-absolute-positions	27.4	76	88
G.	move-blocks-between-absolute-positions-by-size	\	80	72
H.	put-block-into-matching-bowl	31.4	90	100
I.	put-block-into-mismatching-bowl	28.5	90	90
J.	stack-blocks-with-alternate-color	13.2	82	80

5.3.2. Generated Tasks

In order to further test CapRavens' reasoning and generalization capabilities, some more complex tasks, which require deeper reasoning abilities, are added using the task generator. Table 5.3 shows these additional generated tasks with their task goal descriptions.

Since some tasks cannot be automatically evaluated using Ravens' matching matrix, we

Table 5.3.: Generated tasks and completion instructions.

Task Name	Task Goal
1. build-rectangular-pyramid	"Construct a 9-4-1 rectangular pyramid structure in the zone using 14 blocks of the same color."
2. build-cube	"Construct a 2*2*2 cube structure in the zone using 8 blocks of the same color."
3. construct-circle-with-blocks	"Construct a circle with suitable radius with alternating [COLOR1] and [COLOR2] blocks in the zone."
4. construct-circle-ball-middle	"Construct a circle with suitable radius with alternating [COLOR1] and [COLOR2] blocks around the ball."
5. build-concentric-circles	"Construct two concentric circles in the zone using [NUM] [COLOR1] and [NUM + 4] [COLOR2] blocks."
6. divide-blocks	"Divide the blocks into groups of [NUM] and stack each group (also including the group with block number less than [NUM]) in a different zone."
7. max-odd-number-blocks-in-same-color-zone	"Place the maximal odd number of blocks of the same color in each correspondingly colored zone."
8. stack-most-color-block	"Stack blocks of the same color that has the largest quantity in the zone."
9. zone-bisector	"Arrange all blocks on the zone bisector line between two symmetrically placed zones evenly on the tabletop, and the gap between two adjacent blocks' edges should be near the block size, and the line connecting the center of the zones also bisects these blocks."
10. insert-blocks-in-fixture	"Each L-shaped fixture can hold three blocks, suppose the block size is (a,a,a), then in fixture's local coordinate system, the three places that can hold blocks are [(0,0,0),(a,0,0),(0,a,0)]. Fill in all the fixtures which have random position and rotation with blocks, and make sure in the end in every fixture there are three blocks with different colors."

take snapshots of the scene as final observations (see A.2) and evaluate them manually. Table 5.4 shows the performance of CapRavens on these generated more complex unseen tasks. Each task is also tested on 50 different randomly generated seeds.

As can be seen, even when the complexity of the unseen tasks and the description of the task objectives increases, except for task 9 and 10 which require deeper mathematical reasoning, CapRavens can still maintain a success rate of more than half. This shows that CapRavens has a high ability to truly understand and connect the concepts in language instructions with objects and actions. In other words, it has better generalization ability than previous methods.

Table 5.4.: Generated tasks and the experimental results of the ablations.

	Generated Tasks	CapRavens	
		no reporter	with reporter
1.	build-rectangular-pyramid	58	50
2.	build-cube	76	100
3.	construct-circle-with-blocks	92	100
4.	construct-circle-ball-middle	100	100
5.	build-concentric-circles	42	50
6.	divide-blocks	98	100
7.	max-odd-number-blocks-in-same-color-zone	88	100
8.	stack-most-color-block	88	90
9.	zone-bisector	12	18
10.	insert-blocks-in-fixture	8	10

6. Discussion

6.1. Limitations

Capravens takes advantage of the learning and reasoning capabilities of modern LLMs and uses them as high-level policies in imitation learning. According to our experiments, Capravens has been proven to be very effective, especially for zero-shot and few-shot long-horizon tasks. However, this method still has many limitations and room for improvement.

6.1.1. Static Motion Primitives

CapRavens splits the full set of actions of the robot arm in the task into multiple motion primitives, and activates these primitives through different APIs in the code output by the planner to complete the task step by step. However, the planner, including the task generator, can only control the starting and ending postures of the primitives, and the transition from the starting position to the ending position is hardcoded in the Ravens environment. For example, in the pick-place primitive, how the end effector transitions from the initial posture to the grasping point and how it moves from the picking point to the placing point are predetermined. Although the specific movement path will vary with the position of the picking point or the placing point, its essential movement pattern will not change. For example, it may always move in a straight line from the picking point to the placing point. This results in the robot arm having no ability to avoid obstacles during movement. For instance, if there is a tower composed of stacked blocks in the center of the scene, and the robot arm is executing a primitive that moves from the left side of the scene to the right side, it is very likely to knock down the tower of blocks, affecting the completion of the entire task.

6.1.2. Environment Metadata

The Ravens environment based on the gym architecture takes advantages of various APIs. However, on the other hand, since PyBullet acts as a core of the Ravens simulation environment, almost all environment APIs and upper-level APIs required by CapRavens are inseparable from the interface of PyBullet itself. This results in the planner relying on the metadata of the environment itself when actually planning tasks, causing the whole system confined to a single environment.

6.1.3. Matching Matrix

The Ravens environment uses matching matrices to construct and evaluate motion primitives. This feature ensures that all possible correspondences between the start and end points are

covered. For example, in a task using a pick-place primitives, if the row of the matching matrix corresponding to an object is all 1, then the object can be placed in any target place under the current task goal. The matching matrix is very effective for primitives that do not need to distinguish the order of precedence. However, when the task goal requires a certain correspondence between objects, the matching matrix cannot cover all completion situations, and the evaluation based on it becomes inaccurate. For example, there are blocks of various colors on the table, and each color of block has two sizes. When the task only requires all large blocks to be arranged along left edge of the table, the task does not specify the order of arrangement, and the matching matrix can handle all possible arrangements with no doubt. However, when the task goal adds a corresponding relationship constraint, such as also arranging small blocks in the order of the color of the large blocks on the other side of the table, although a matching matrix can be established for each large and small block group, the color correspondence between the large and small blocks cannot be described by these two matrices. This is because the positions of the large blocks can only be determined when the task is run, while the matching matrix of the small blocks that depends on the actual positions of the large blocks needs to be defined before the task is run.

6.2. Future Improvements

6.2.1. Dynamic Motion Primitives

An intuitive way to solve the problem of static motion primitives is to use dynamic motion primitives that can change according to the scene. One idea is to increase the parameters of each motion primitive. In addition to the starting and ending points of the motion path, it is possible to also add the acceleration of the motion, the force on the end effector, and intermediate waypoints, etc. In CapRavens, this means increasing the input parameters of the APIs responsible for the primitive, so that the planner can be freer when driving the robot arm. Another idea is to directly use path planning algorithms such as A* and RRT (Rapidly-exploring Random Tree) to design a path between the specified starting and ending points under the premise of global observation, so that the robotic arm can move quickly and safely without increasing the burden on the planner. In addition, Dynamic Motion Primitive (DMP) [26] itself is also a method based on imitation learning to reproduce the demonstrated movement path. Therefore, one or multiple pre-trained DMPs can also be used instead of one static motion primitive. After extensive training, the DMP will be smoother especially in the short period of movement before and after the end effector touches the object. Moreover, when the robot arm is equipped with a detector that can sense obstacles, DMPs can be further augmented to adjust the trajectory when an obstacle is detected. For instance, the robot could slightly modify the trajectory in real-time if an obstacle is in the path. Furthermore, the state-of-the-art diffusion model can also be used to design dynamic motion paths. For example, in 3D Diffusion Policy (DP3) [27], a sparse point cloud describing the scene is first generated, and then one MLP will encode the point cloud to output compact 3D representations, and finally actions conditioning on these 3D representations and the

robot's states using a diffusion-based backbone are generated. In particular, for complex objects, sometimes the robot arm needs to interact with specific parts to complete the action. For example, for the action of opening a drawer, the starting point of the primitive needs to be located on the drawer handle. ManipLLM [28] provides an idea for LLM prompt, which enables it to output the coordinates of the best interaction point on complex objects in the environment and the movement direction of the primitive.

6.2.2. Observation from Outside

To address the problem of relying on environmental meta information, all the scene information needs to be handed over to the outside to obtain. For the task of desktop rearrangement, this process mainly includes querying what objects are on the desktop, as well as the attributes and postures of the objects. For this kind of visual task, it is intuitive to think of V-SLAM (Visual Simultaneous Localization and Mapping) and filter-based object detection such as the OpenCV library commonly used in traditional computer vision. However, the former focuses on the positioning of the observer itself, and the latter has difficulty in finding objects without predefined features in complex scenes and labeling them. Therefore, it is ideal to use a CNN-based object analysis network. Deep3DBox [29] and MonoGRNet [30] use RGB snapshots of the scene to analyze the objects in the scene and their bounding boxes. The position and rotation of the object in the current space can be further obtained through the calibration parameters of the camera. PointNet++ [31] and 3D SSD [32] first convert the scene into a point cloud based on the depth map of the scene, and then semantically segment the point cloud to detect objects and their positions and bounding boxes. Although the computation cost is higher, the result is more accurate. VoxPoser [33] further discretizes the point cloud to various voxel maps, and can obtain the corresponding information of the object in these different maps. In addition, since the goal of the task is described in natural language, a multimodal VLM with a transformer can also be used to purposefully obtain the classifications and positions of objects in the scene. For example, MDETR [34] can obtain the label and relative position of objects from the input RGB image according to input language instructions.

6.2.3. Fine Tuning

Since CapRavens makes extensive use of the LLM, fine-tuning LLM will further improve the success rate and versatility of CapRavens for different tasks [35]. As the main components of the whole structure, planner and reporter should benefit the most after fine-tuning. For the planner, the training and test data can be made like the example tasks in the user prompt. When evaluating the training results, the output code can be run directly, and then the Ravens environment can automatically evaluate the task completion, or it can be manually evaluated based on the task video. For the plain text reporter, its training process is similar to that of the planner, and the training data can also be made by referring to the example tasks in the user prompt. For the VLM-based reporter, the training data should consist of a snapshot of the scene after the task is completed, or two snapshots of the scene at the beginning and end

of the task, plus a text description of whether the task is successful or not. The evaluation of the training can be obtained by comparing with the evaluation results provided by the Ravens environment for the corresponding tasks in the test set, or they can be simply manually evaluated.

6.2.4. Real-World Experiments

Language-conditioned robot manipulation should not be constrained to simulation environments, as its ultimate goal is always to be applied in the real world. Since the evaluation data is collected in a simulated environment, the performance in the real world cannot be guaranteed. Because the APIs used by CapRavens are all upper-level APIs, the premise of real-world experiments is to lay out all the lower-level APIs, and the most important of which is the observation of the environment and the acquisition of environmental information. After setting up the API, the idea of a real-world experiment is to add more real objects that are not in the simulation environment, or use desktops of other shapes. In addition, due to the limitations of the matching matrix in the Ravens environment, more tasks that cannot be evaluated by the matching matrix can be tested in real-world experiments, such as object deformation, etc.

7. Conclusion

In summary, language-conditioned robot manipulation is a rapidly advancing field with the potential to transform various industries by enabling robots to interpret and execute tasks based on natural language commands. By integrating natural language processing, computer vision, and advanced manipulation algorithms, researchers aim to develop robots capable of understanding subtle language cues and performing complex manipulation tasks with precision and efficiency just like humans. Despite significant progress, several challenges remain, including the limited ability of these systems to generalize and the discrepancies between performance in simulated environments and real-world applications.

In this work, we introduce CapRavens which is a novel task planning and executing method using motion primitives that significantly increases the generalization ability of the robot agent. In CapRavens, the natural language task objectives are directly handed over to the task planner based on the LLM with powerful reasoning capabilities. Through the information of the environment collected by various APIs, the task is finally completed in the form of motion primitives. With the help of the reporter, the entire system can complete self-checking in a closed loop to deal with possible errors in operation. The robot agent therefore does not need to undergo any training, but still gains great generalization capabilities based on the knowledge reserve of LLMs and basic motion primitives. Based on a series of high-level APIs and evaluation results, we demonstrated the effectiveness of our method. Finally, we discuss the limitations and the corresponding potential improvement directions of the method. We will further research in this field depending on the research direction discussed in Chapter 6.

A. Implementation Details

A.1. Hardware and Software

All of the experiments were performed on a local Windows 10 machine with 16 virtual Central Processing Units and 16 GB RAM. The expriments are performed in Conda Python 3.9.18 virtual environment. For source code and detailed information about the environment, please refer to our GitHub repository[1].

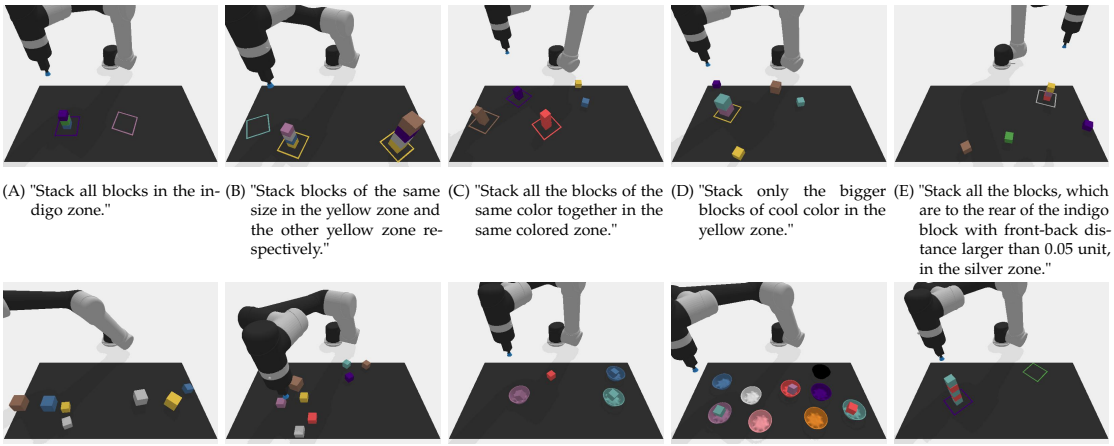
A.2. Motion Primitives

The motion primitives used in this thesis are the same as of CLIPort and LoHoRavens. The primitives in Ravens are basically hard-coded robot end effector movement patterns using PyBullet APIs.

A.3. Completion Observations

The examples of observations in RGB images of completion of the tasks in chapter 5 are listed as Fig. A.1 and Fig. A.2.

[1] `https://github.com/Flakeeeet/capravens`



(A) "Stack all blocks in the indigo zone."

(B) "Stack blocks of the same size in the yellow zone and the other yellow zone respectively."

(C) "Stack all the blocks of the same color together in the same colored zone."

(D) "Stack only the bigger blocks of cool color in the yellow zone."

(E) "Stack all the blocks, which are to the rear of the indigo block with front-back distance larger than 0.05 unit, in the silver zone."

(F) "Move all the blocks in the top-left area to bottom-left area."

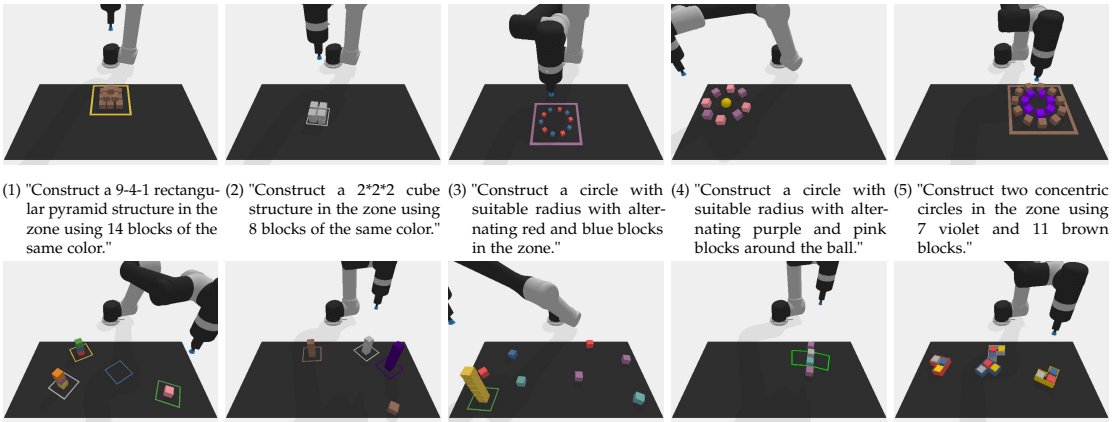
(G) "Move all the smaller blocks in the bottom-right area to bottom-left area."

(H) "Put the blocks in the bowls with matching colors."

(I) "Put the blocks in the bowls with mismatching colors."

(J) "Stack blocks with alternate colors on the indigo zone, starting with the cyan color."

Figure A.1.: Completion observations of LoHoRavens tasks



(1) "Construct a 9-4-1 rectangular pyramid structure in the zone using 14 blocks of the same color."

(2) "Construct a 2*2*2 cube structure in the zone using 8 blocks of the same color."

(3) "Construct a circle with suitable radius with alternating red and blue blocks in the zone."

(4) "Construct a circle with suitable radius with alternating purple and pink blocks around the ball."

(5) "Construct two concentric circles in the zone using 7 violet and 11 brown blocks."

(6) "Divide the blocks into groups of 3 and stack each group (also including the group with block number less than 3) in a different zone."

(7) "Place the maximal odd number of blocks of the same color in each correspondingly colored zone."

(8) "Stack blocks of the same color that has the largest quantity in the zone."

(9) "Arrange all blocks on the zone bisector line between two symmetrically placed zones evenly on the tabletop, and the gap between two adjacent blocks' edges should be near the block size, and the line connecting the center of the zones also bisects these blocks."

(10) "Each L-shaped fixture can hold three blocks, suppose the block size is (a,a,a), then in fixture's local coordinate system, the three places that can hold blocks are [(0,0,0),(a,0,0),(0,a,0)]. Fill in all the fixtures which have random position and rotation with blocks, and make sure in the end in every fixture there are three blocks with different colors."

Figure A.2.: Completion observations of generated tasks

B. Experiment Details

B.1. Evaluation Details

For the Lohoravens task, the matching matrix of Ravens is used for evaluation, while the generated tasks are basically manually evaluated based on the final observation states. If errors are raised during the task test, it will be considered a failure.

B.2. Environment Parameters

List of important environment parameters are demonstrated in Table B.1.

Table B.1.: Environment parameters.

Parameter	Value
frequency (Hz)	240
pixel size	0.003125
workplace bound x	(0.25, 0.75)
workplace bound y	(-0.5, 0.5)
workplace bound z	(0.0, 0.3)
general plane position error tolerance	0.01
bowl plane position error tolerance	0.05
general SE(2) rotation error tolerance	15°
general vertical position error tolerance	0.015

B.3. Generated Task Code

For better understanding, an example of generated plan and evaluation for task A 'stack-all-blocks-on-a-zone' is demonstrated.

B.1 is a code plan for goal 'Stack all blocks in the silver zone.' using prompt shown in chapter 4. As can be seen from the code plan, the planner confirms the task goal at very first, then according to the goal content, it tries to get the goal-related objects by **parse_obj_name** LMP. For some tasks involving placing in a small area the planner will also check whether the target is free, when not then the planner will clear the target area. Finally the planner calls the motion primitive APIs to execute.

B.2 is a corresponding evaluation for the task. The reporter also gets the goal-related objects first, and then gets their attributes such as size or position. The final judgement is determined

```
say('Ok - stacking all blocks in the silver zone')
blocks = parse_obj_name('movable blocks', f'objects = {get_obj_names()}')
silver_zone = parse_obj_name('silver zone', f'objects = {get_obj_names()}')
if isinstance(silver_zone, list):
    silver_zone = silver_zone[0]
objs = is_target_occupied(silver_zone)
if objs:
    for obj in objs:
        put_first_on_second(obj, get_random_free_pos(targ=silver_zone))
stack_objects_in_order(blocks, silver_zone)
```

Figure B.1.: Generated plan for task A

by the conditional statements based on the task goal. For goal 'Stack all blocks in the silver zone.', the conditional statements generated by the reporter here are checking if all blocks' horizontal positions are within the zone area and their heights are all different from one another.

```
judge = False
object_positions = get_obj_pos_dict()
silver_zone = parse_obj_name('silver zone', f'objects = {get_obj_names()}')
if isinstance(silver_zone, list):
    silver_zone = silver_zone[0]
zone_size = np.array(get_bbox(silver_zone))
zone_pos = get_obj_pos(silver_zone)[0]
blocks = parse_obj_name('movable blocks', f'objects = {get_obj_names()}')
block_positions = [np.array(object_positions[block][0]) for block in blocks]
if all(
(zone_pos[0] - zone_size[0] / 2 <= block_pos[0]
<= zone_pos[0] + zone_size[0] / 2) and
(zone_pos[1] - zone_size[1] / 2 <= block_pos[1]
<= zone_pos[1] + zone_size[1] / 2) for block_pos in block_positions):
    heights = [block_pos[2] for block_pos in block_positions]
    if len(heights) == len(set(heights)):
        judge = True
ret_val = judge
```

Figure B.2.: Generated evaluation for task A

List of Figures

List of Tables

Bibliography

[1] C. Lynch, M. Khansari, T. Xiao, V. Kumar, J. Tompson, S. Levine, and P. Sermanet. "Learning Latent Plans from Play". In: *CoRR* abs/1903.01973 (2019). arXiv: 1903.01973. URL: http://arxiv.org/abs/1903.01973.

[2] C. Lynch and P. Sermanet. "Grounding Language in Play". In: *CoRR* abs/2005.07648 (2020). arXiv: 2005.07648. URL: https://arxiv.org/abs/2005.07648.

[3] D. Bahdanau, F. Hill, J. Leike, E. Hughes, P. Kohli, and E. Grefenstette. "Learning to Follow Language Instructions with Adversarial Reward Induction". In: *CoRR* abs/1806.01946 (2018). arXiv: 1806.01946. URL: http://arxiv.org/abs/1806.01946.

[4] K.-H. Lee, T. Xiao, A. Li, P. Wohlhart, I. Fischer, and Y. Lu. *PI-QT-Opt: Predictive Information Improves Multi-Task Robotic Reinforcement Learning at Scale*. 2022. arXiv: 2210.08217 [cs.RO]. URL: https://arxiv.org/abs/2210.08217.

[5] P. Goyal, S. Niekum, and R. J. Mooney. "PixL2R: Guiding Reinforcement Learning Using Natural Language by Mapping Pixels to Rewards". In: *CoRR* abs/2007.15543 (2020). arXiv: 2007.15543. URL: https://arxiv.org/abs/2007.15543.

[6] O. Mees, J. Borja-Diaz, and W. Burgard. *Grounding Language with Visual Affordances over Unstructured Data*. 2023. arXiv: 2210.01911 [cs.RO]. URL: https://arxiv.org/abs/2210.01911.

[7] X. Liu, Y. Zheng, Z. Du, M. Ding, Y. Qian, Z. Yang, and J. Tang. "GPT understands, too". In: *AI Open* (2023).

[8] S. Ugare, T. Suresh, H. Kang, S. Misailovic, and G. Singh. *SynCode: LLM Generation with Grammar Augmentation*. 2024. arXiv: 2403.01632 [cs.LG]. URL: https://arxiv.org/abs/2403.01632.

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

[10] S. Zhang, P. Wicke, L. K. Şenel, L. Figueredo, A. Naceri, S. Haddadin, B. Plank, and H. Schütze. "LoHoRavens: A Long-Horizon Language-Conditioned Benchmark for Robotic Tabletop Manipulation". In: *arXiv preprint arXiv:2310.12020* (2023).

[11] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[12] O. Ronneberger, P. Fischer, and T. Brox. "U-Net: Convolutional Networks for Biomedical Image Segmentation". In: *CoRR* abs/1505.04597 (2015). arXiv: 1505.04597. URL: http://arxiv.org/abs/1505.04597.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[14] S. Nair, E. Mitchell, K. Chen, B. Ichter, S. Savarese, and C. Finn. "Learning Language-Conditioned Robot Behavior from Offline Data and Crowd-Sourced Annotation". In: *CoRR* abs/2109.01115 (2021). arXiv: 2109.01115. URL: https://arxiv.org/abs/2109.01115.

[15] C. Lynch and P. Sermanet. "Language conditioned imitation learning over unstructured data". In: *arXiv preprint arXiv:2005.07648* (2020).

[16] K. Pertsch, Y. Lee, and J. Lim. "Accelerating reinforcement learning with learned skill priors". In: *Conference on robot learning*. PMLR. 2021, pp. 188–204.

[17] H. Zhou, Z. Bing, X. Yao, X. Su, C. Yang, K. Huang, and A. Knoll. *Language-Conditioned Imitation Learning with Base Skill Priors under Unstructured Data*. 2024. arXiv: 2305.19075 [cs.RO]. URL: https://arxiv.org/abs/2305.19075.

[18] O. Mees, L. Hermann, and W. Burgard. *What Matters in Language Conditioned Robotic Imitation Learning over Unstructured Data*. 2022. arXiv: 2204.06252 [cs.RO]. URL: https://arxiv.org/abs/2204.06252.

[19] A. Zeng, P. Florence, J. Tompson, S. Welker, J. Chien, M. Attarian, T. Armstrong, I. Krasin, D. Duong, V. Sindhwani, et al. "Transporter networks: Rearranging the visual world for robotic manipulation". In: *Conference on Robot Learning*. PMLR. 2021, pp. 726–747.

[20] M. Shridhar, L. Manuelli, and D. Fox. "Cliport: What and where pathways for robotic manipulation". In: *Conference on robot learning*. PMLR. 2022, pp. 894–906.

[21] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, et al. "Learning transferable visual models from natural language supervision". In: *International conference on machine learning*. PMLR. 2021, pp. 8748–8763.

[22] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165.

[23] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL]. URL: https://arxiv.org/abs/2201.11903.

[24] D. Tola and P. Corke. "Understanding URDF: A dataset and analysis". In: *IEEE Robotics and Automation Letters* (2024).

[25] L. Wang, Y. Ling, Z. Yuan, M. Shridhar, C. Bao, Y. Qin, B. Wang, H. Xu, and X. Wang. *GenSim: Generating Robotic Simulation Tasks via Large Language Models*. 2024. arXiv: 2310.01361 [cs.LG]. URL: https://arxiv.org/abs/2310.01361.

[26] A. J. Ijspeert, J. Nakanishi, H. Hoffmann, P. Pastor, and S. Schaal. "Dynamical Movement Primitives: Learning Attractor Models for Motor Behaviors". In: *Neural Computation* 25.2 (2013), pp. 328–373. DOI: 10.1162/NECO_a_00393.

[27] Y. Ze, G. Zhang, K. Zhang, C. Hu, M. Wang, and H. Xu. *3D Diffusion Policy: Generalizable Visuomotor Policy Learning via Simple 3D Representations*. 2024. arXiv: 2403.03954 [cs.RO]. URL: https://arxiv.org/abs/2403.03954.

[28] X. Li, M. Zhang, Y. Geng, H. Geng, Y. Long, Y. Shen, R. Zhang, J. Liu, and H. Dong. *ManipLLM: Embodied Multimodal Large Language Model for Object-Centric Robotic Manipulation*. 2023. arXiv: 2312.16217 [cs.CV]. URL: https://arxiv.org/abs/2312.16217.

[29] A. Mousavian, D. Anguelov, J. Flynn, and J. Kosecka. "3D Bounding Box Estimation Using Deep Learning and Geometry". In: *CoRR* abs/1612.00496 (2016). arXiv: 1612.00496. URL: http://arxiv.org/abs/1612.00496.

[30] Z. Qin, J. Wang, and Y. Lu. "Monogrnet: A general framework for monocular 3d object detection". In: *IEEE transactions on pattern analysis and machine intelligence* 44.9 (2021), pp. 5170–5184.

[31] C. R. Qi, L. Yi, H. Su, and L. J. Guibas. "Pointnet++: Deep hierarchical feature learning on point sets in a metric space". In: *Advances in neural information processing systems* 30 (2017).

[32] Q. Luo, H. Ma, L. Tang, Y. Wang, and R. Xiong. "3d-ssd: Learning hierarchical features from rgb-d images for amodal 3d object detection". In: *Neurocomputing* 378 (2020), pp. 364–374.

[33] W. Huang, C. Wang, R. Zhang, Y. Li, J. Wu, and L. Fei-Fei. "Voxposer: Composable 3d value maps for robotic manipulation with language models". In: *arXiv preprint arXiv:2307.05973* (2023).

[34] A. Kamath, M. Singh, Y. LeCun, G. Synnaeve, I. Misra, and N. Carion. "Mdetr-modulated detection for end-to-end multi-modal understanding". In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pp. 1780–1790.

[35] S. Aathilakshmi, G. Sivapriya, and T. Manikandan. "6 LLM Fine-Tuning: Instruction and Parameter-Efficient Fine-Tuning (PEFT)". In: *Generative AI and LLMs: Natural Language Processing and Generative Adversarial Networks* (2024), p. 117.