Learning Reward for Robot Skills Using Large Language Models via Self-Alignment

Yuwei Zeng [1] **Yao Mu** [2 1] **Lin Shao** [1]

Abstract

Learning reward functions remains the bottleneck to equip a robot with a broad repertoire of skills. Large Language Models (LLM) contain valuable task-related knowledge that can potentially aid in the learning of reward functions. However, the proposed reward function can be imprecise, thus ineffective which requires to be further grounded with environment information. We proposed a method to learn rewards more efficiently in the absence of humans. Our approach consists of two components: We first use the LLM to propose features and parameterization of the reward, then update the parameters through an iterative self-alignment process. In particular, the process minimizes the ranking inconsistency between the LLM and the learnt reward functions based on the execution feedback. The method was validated on 9 tasks across 2 simulation environments. It demonstrates a consistent improvement over training efficacy and efficiency, meanwhile consuming significantly fewer GPT tokens compared to the alternative mutation-based method. Project website: https://sites.google.com/view/rewardselfalign.

1. Introduction

Reinforcement learning has demonstrated the effectiveness in acquiring complex skills from walking over uneven terrain (Valsecchi et al., 2020; Manchester et al., 2011) to dexterous manipulation(Akkaya et al., 2019; Chen et al., 2023). However, such effectiveness largely depends on a well-designed reward function that relies on expert knowledge of tasks, followed by non-trivial tuning often to both optimize the efficacy and prevent the policy from exploiting flaws that can be easily introduced during reward shaping. Inverse reinforcement learning (IRL) (Abbeel & Ng, 2004; Ho & Ermon, 2016) seeks to resolve this issue by automatically learning the reward function from expert demonstrations. Still the process can be exceedingly costly for expert demonstration gathering to cover the vast variety and complexity of the state space and yield a robust control.

Recently, Large Languaged Models (LLMs), trained using extensive human data, have demonstrated to be embedded with richly useful task-related knowledge. Several existing studies have used LLM to directly propose action (Liang et al., 2023) or reward values (Kwon et al., 2023; Adeniji et al., 2023). However, using LLM to learn reward functions still presents a challenge due to the task sensitivity to the exact numerical values while LLM shows limited capacity. In addition, such value setting in general requires to be grounded to the specific setup. For example, In (Yu et al., 2023) LLM set the target torso height for a quadruped to 0.3m for moon walk instruction. With further inquiring about rationale behind the number, LLM explains "the specific value here is arbitrary and should be adjusted based on the robot's design and requirements". Previous works address such limitation with human feedback or in-context example (Yu et al., 2023; Xie et al., 2023; Wang et al., 2023), or evolutionary method (Ma et al., 2023) with success rate feedback to LLM, both are still in a form of trial and error from LLM update, that might end up with fluctuated instead of improving performance.

In this work, we aim to answer the question: is there a way we can learn the reward more efficiently in the absence of humans? It is observed that LLMs have shown promising ability in summarizing and classifying text, which allows them to effectively distinguish different observations in textual form. We propose to utilize such ability to extract ranking signals from LLM, which could be more robust to guide reward learning than direct value prediction of the parameters. We first utilize LLM to break down a task into steps with Dos and Don′ts through Chain of Thought (CoT) (Wei et al., 2022) and propose the initial reward parameterization, particularly the feature selection and template structure. Next, we iteratively update the parameters of the proposed reward function in a self-alignment process which operates on a double-loop structure. The inner loop induces the optimal policy from the current reward function, samples trajectories using this policy and generates execution

[1]National University of Singapore [2]The University of Hong Kong. Correspondence to: Lin Shao <linshao@nus.edu.sg>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

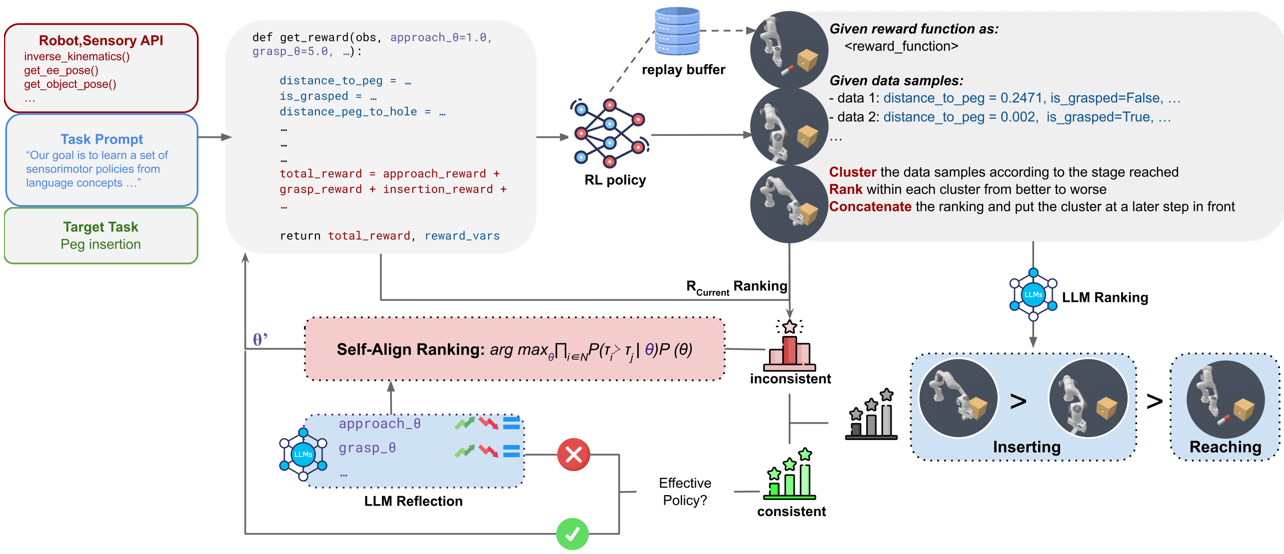

Figure 1. The overview of our method. We learn the reward function using LLM with a bi-level optimization structure. We first use the LLM to propose features and parameterization of the reward function. Next, we update the parameters of this proposed reward function through an iterative self-alignment process. In particular, this process minimizes the ranking inconsistency between the LLM and our learned reward functions based on the new observations.

descriptions with the proposed reward features. The outer loop updates the reward parameters by aligning the ranking between LLM proposed with the execution description feedback, and the ranking from the current reward function. When no discrepancy exists yet no effective policy is developed, we also actively adjust reward parametrization in the direction LLM reflection hints (Liu et al., 2023), and numerically optimize it to keep the same ranking self-consistency. This process is similar to IRL's bi-level optimization structure, with one key difference in the outer loop: instead of minimizing differences between expert demonstrations (Ke et al., 2021), our method employs ranking from LLM. Since all supervision signals come from LLM, we describe this as the self-alignment reward update.

To summarize, out contribution includes:

- We proposed a framework to learn the reward functions with LLM through an interative self-alignment process, which periodically updates the reward function to minimize the ranking inconsistency of execution generated from LLM and the current reward function.

- Leveraged upon the self-alignment process, we included active parameter adjustment with LLM heuristic to improve reward saliency, while preventing it from unintentional flaw through enforcing the consistency.

- We validated the framework on 9 tasks on 2 simulation environments. It demonstrates a consistent improvement over training efficacy and efficiency while being token efficient compared to alternative method.

2. Related Work

Inverse Reinforcement Learning from Human Preference Inverse reinforcement learning (IRL) studies how to autonomously learn a reward function from expert data. Traditional IRL methods utilise the optimal expert demonstrations as supervision signals, aiming to learn a reward function that can encode an objective consistent with the expert demonstrations(Ng et al., 2000; Abbeel & Ng, 2004; Ho & Ermon, 2016; Hadfield-Menell et al., 2017). The bottleneck of IRL is the quantity and quality of the expert demonstrations. Some other works explore other forms of expert supervision, such as trajectory preference/ranking (Sadigh et al., 2017; Palan et al., 2019; Lee et al., 2021; Bıyık et al., 2022a; Mehta & Losey, 2022). Learning from preference ranking offers some benefits over standard IRL, which relies on almost perfect expert demonstrations for imitation. Essentially, it learns from ranked sub-optimal demonstrations, potentially surpassing the original demonstrator's performance. This makes it less demanding for experts to create only optimal behaviours and also encodes further contrasting information about preferable and undesirable behaviours. A major downside, however, remains the need to amass this information. We investigate the potential to extract task-related data from a pre-trained large language model, effectively bypassing this data collection challenge.

LLM for robotics reward learning Large Language Models (LLMs) have shown great potential as cost-effective tools for extracting task objectives as they embed substantial human knowledge (Huang et al., 2022; Brohan et al., 2023). A few works proposed to generate code with large language models such as ProgPrompt (Singh et al., 2023), Policy-as-Code (Liang et al., 2023). Based on this, several prior studies have further utilized LLMs to predict reward functions. (Kwon et al., 2023) learns to translate nature language instructions to reward values, (Yu et al., 2023; Xie et al., 2023; Wang et al., 2023) proposed robot skill synthesis or task learning with LLM-generated reward function and further refine with human feedback or in-context example. The work either uses LLMs in a zero-shot manner implying their reward function can't utilize environmental feedback, or the update relies on human intervention which can be laborious. (Ma et al., 2023) proposed to automatically generate and select the reward function through an evolutionary manner which achieved human-level reward design. Our research focuses on learning reward functions that are sensitive to the numerical values setting. Instead of depending on human feedback or LLM mutation, we suggest an iterative self-alignment that uses LLMs to offer alternative signals that supervise reward learning.

3. Background

Problem Definition Consider a finite-horizon Markov Decision Process (MDP) parameterized by $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, T)$ where $\mathcal{S}, \mathcal{A}$ are the state and action spaces, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ is the transition function, $\mathcal{R} : \mathcal{S} \rightarrow \mathbb{R}$ is the reward function, and T is the horizon. A policy π is a mapping from states to probabilities over actions, $\pi(a|s)$. Given a policy and an MDP, the expected return of the policy is given by $J(\pi; \mathcal{R}) = \mathbb{E}[\sum_{t=0}^{T-1} \mathcal{R}(s_t)|\pi]$. The expert policy should be one that optimizes this return, $\pi_E := \arg\max J(\pi; \mathcal{R})$ w.r.t. the ground-truth reward $\mathcal{R}$. In our setting, we are given a partial MDP without i) the reward function $\mathcal{R}$ nor ii) any forms of expert demonstrations. Instead, we have access to an LLM that can rank a sequence of M trajectories τ_k with decreasing preference for $k = 1, \ldots, m$ based on the last state s_T^k. The LLM is assumed to have an internal goal or intrinsic understanding of tasks, therefore its ranking is consistent with a human demonstrator optimizing the ground-truth reward function $\mathcal{R}$. We aim to find a parameterized reward function R_θ such that the ranking of these M trajectories based on it is consistent with the ranking given by the LLM.

Boltzmann Rationality Similar to modelling noisy optimal human behaviour, we model LLM's preference with a Boltzmann rationality model (Luce, 1959), which assumes it will act to prefer a trajectory with probability proportional to the exponential trajectory return, where $\beta \in [0, \infty)$ is the rationality coefficient controls the level of rationality. Action becomes fully rational and deterministic when $\beta \rightarrow \infty$ and uniformly random when $\beta \rightarrow 0$.

$$P(\tau) \propto \exp\{\beta \sum R_\theta(s_t, a_t)\}$$

Such noisy optimality is critical here as it aligns with our motivation that LLM can be numerically imprecise or unstable. It does not require perfect ranking feedback but only the majority of them. It also aligns with our observation on LLM act rationally but also imperfectly. For instance, it constantly mis-rank in the presence of subtle numerical differences.

Reward Learning from Pairwise Preference In reward learning, denoting $\mathcal{D}$ is the dataset with N pairwise preferences (τ_i, τ_j) where $\tau_i \succ \tau_j$, we seek to estimate the true reward parameter θ that maximizes the posterior:

$$P(\theta|\mathcal{D}) \propto \prod_{i=1}^{N} P(\tau_i^n \succ \tau_j^n|\theta)P(\theta)$$

Prior is system-dependent and a common choice without special assumption is a uniform prior within the domain $\mathcal{U}[\theta_{min}, \theta_{max}]$. Pairwise preference likelihood $P(\tau_i \succ \tau_j|\theta)$ is modeled with the Bradley-Terry model (Bradley & Terry, 1952) with Boltzmann-rational model being the score function p:

$$P(\tau_i \succ \tau_j|\theta) = \frac{p(\tau_i|\theta)}{p(\tau_i|\theta) + p(\tau_j|\theta)} \tag{1}$$

$$= \frac{\exp \beta \sum R_\theta(s_t^i, a_t^i)}{\exp \beta \sum R_\theta(s_t^i, a_t^i) + \exp \beta \sum R_\theta(s_t^j, a_t^j)} \tag{2}$$

The reward function generated by LLM may be in arbitrary form with complex parameter distribution, sampling-based methods such as the Metropolis-Hastings algorithm (Sadigh et al., 2017; Hoegerman & Losey, 2023) or Gaussian process (Bıyık et al., 2023) can be deployed to model the posterior flexibly:

$$\theta_{MAP}(\mathcal{D}) = \arg\max_\theta \prod_{n=1}^{N} P(\tau_i^n \succ \tau_j^n|\theta)P(\theta) \tag{3}$$

$$= \arg\max_\theta \sum_{n=1}^{N} \left(\beta R_\theta(\tau_i^n) - \log \sum_{\tau \in \mathcal{D}_n} \exp \beta R_\theta(\tau) \right) + \log P(\theta) \tag{4}$$

4. Main Method

We propose a framework to learn the reward functions from Large Language Models (LLMs). We observe that LLM not only encodes useful task information, it can also serve as a discriminator to evaluate the performance given the observation signals. Based on these, our reward learning method consists of two parts: first we extract the skill-specific reward function parameterization, $R_\theta(.)$, from LLM using a sequence of guiding prompts, then we design an iterative self-alignment procedure to fit the reward function $R_\theta(.)$ using ranking-based preference learning.

4.1. Reward Parameterization Proposal

We connects LLM to the environment via a dictionary of observations that the environment provides and actions that the agent applies as required by reward calculation. Ideally, a versatile robotics system connects to tasks via modularised interfaces such as a robot interface which define the capabilities of the system. For example inverse kinematics, retrieving object pose, collision detection. They can later utilized to compose tasks and their evaluation. When such structured functions exist, we also feed LLM with an abstraction containing only function define line, type hints with function description, and (input, output) description. In doing so, LLM is expected to leverage upon the existing functions the system is equipped with for reward generation, meanwhile avoiding diluting useful information with a large trunk of function content text or being exposed to any privileged information they may contain.

To generate the initial reward parametrization proposal, we propose to apply the Chain-of-Thought (CoT) to guide the Large Language Model (LLM) in the generation. The details of the prompts we used are included in the appendix. Our process involves a 4-step procedure,

First, we feed the environment information to LLM with available observations and action descriptions. When interfaces are well defined, we feed the abstract of system interfaces to LLM and ask it to summarize the available functions into a dictionary with the key being the function name, the value being a dictionary summarizing "brief description", "input", and "output".

Next, the task is given as a phrase or short sentence. For example: "touching", or "open the scissor lay flat on the surface with two hands". We prompt the LLM to describe the target skill in natural language.

In the third step, LLM is prompted to break the task into steps if it is a multi-step task. For each step, generate a list of "Dos" and "Don'ts" and identify the relevant observations associated with it.

Finally, we prompt the LLM to develop the Python function based on analysis from the previous step using observations and functions summarized from the first step. LLM is instructed to put all numerical values into input arguments, and always assign the reward or penalty associated features into a variable before reward calculation. Later the features will be automatically parsed during reward calculation for execution description generation and feedback to LLM.

4.2. Updating Reward via Self-Alignment

The reward is learnt by iteratively optimizing a policy given a reward function in the inner loop, and a reward function in the outer loop with feedback on the policy behaviour. The process iterates and terminates at a satisfactory success rate or total updating steps whichever it reaches earlier. For feedback, ranking is opted instead of the absolute scores or exact numerical adjustments from LLM to suit the restricted

Algorithm 1 Self-Alignment Reward Update

Require: Learned Reward R_θ, replay buffer $\mathcal{D}_{RB} = \varnothing$, initial policy π_θ

1: **for** $t = 0, 1, ...$ **do**
2:     **for** $k = 0, 1, ...$ **do**
3:         Update π_θ using RL with R_θ
4:     **end for**
5:     Sample M trajs $\{\tau_i\}_{i=1,...,M}$ using π^*_θ
6:     *[optional]* Sample N trajs $\{\tau_i\}_{i=1,...,N}$ evenly from $\mathcal{D}_{RB}$ based on reward histogram
7:     Calculate $Rank_{R_\theta}$ by ranking $\{\tau_i\}$ using R_θ
8:     Calculate $Rank_{LLM}$ by ranking $\{\tau_i\}$ using LLM
9:     $\mathcal{D}_{neg} \leftarrow$ discrepancy($Rank_{R_\theta}$, $Rank_{LLM}$)
10:     $\mathcal{D}_{pos} \leftarrow |\mathcal{D}_{neg}|$ pairs sampled from agreed($Rank_{R_\theta}$, $Rank_{LLM}$)
11:     $\mathcal{D} \leftarrow \mathcal{D}_{neg} + \mathcal{D}_{pos}$
12:     Bayesian update on θ according to Eq (4)
13: **end for**

numerical reasoning capacity of LLM. For policy learning, model-free Reinforcement Learning (RL) is used due to its flexibility and demonstrated capability for intricate skill learning.

The iterative reward update with self-alignment is presented in Algorithm 1. Within each iteration, the policy is first updated using RL with the current reward function (line 2-4). Next, we draw M samples from the updated policy by rolling it out such that the collected trajectories reflect the current policy behaviour (line 5). When a replay buffer is available such as an off-policy RL is used, the reward histogram of the replay buffer will be parsed and additional N samples are drawn uniformly from the bins (line 6). This is to enable the feedback to be more inclusive so the potential misspecification can be better detected. We then aggregate the samples and retrieve the two ranking sets from the current reward function R_θ and the LLM through the textual feedback (line 7-8). The textual feedback is automatically constructed by concatenating the identified reward feature names and their values from the reward function with Python local variable parsing. More details on this can be found in Appendix A.2.

To generate the dataset of pairwise comparison $\mathcal{D} = \{(\tau_i, \tau_j)^0, (\tau_i, \tau_j)^1, \ldots\}$ where $\tau_i^n \succ \tau_j^n$, we first parse all inconsistent pairs by comparing the two ranking sets. To resolve the reward inconsistency but also maintain the achieved consistency, we additionally sample an equal amount of consistent pairs from the comparison (line 9 - 10). The ranking from LLM serves as the ground truth and determines $\tau_i^n \succ \tau_j^n$ in $\mathcal{D}$. $\mathcal{D}$ is further shuffled and updates the reward model using Bayesian inference. We follow previous works and use Metropolis-Hastings algorithm. The parameters are updated to maximize the posterior of the preference modelling based on the learned reward function with Eq (4). The updated parameter will be accepted if and only if such inconsistency reduces and converges.

When there is no discrepancy, yet no effective policy is trained after an iteration with a success rate lower than a threshold, the relevant reward or penalty term is assumed to be not salient, LLM is prompted to reflect on the execution and identify the relevant reward or penalty term(s) with corresponding parameter(s). LLM is instructed to suggest a new parameter value where only the value changing direction as + / - is parsed. Such change temporally overwrites the parameter domain from $[\theta_{\min}, \theta_{\max}]$ to $(\theta_{\text{current}}, \theta_{\max}]$ or $[\theta_{\min}, \theta_{\text{current}})$ and conduct new parameter search with self-alignment. Similarly, the updated parameter will be accepted with maximumly reduced and no new inconsistency generated. In doing so, we combine heuristics as active reward adjustment while maintaining the reward logic by preserving the reward rank to avoid unintentional flaws being introduced with the new parameterization.

5. Experiments

5.1. Evaluation Objectives

To evaluate the proposed framework, we design our experiments to answer the following three questions:

$\mathcal{H}1$ **-** Can this pipeline generate effective reward functions to induce optimal policies on varied skills learning?

$\mathcal{H}2$ **-** Can the periodic update through self-alignment improve the numerical impreciseness and instability, thus the efficacy of reward functions?

$\mathcal{H}3$ **-** How does this method perform compared to the alternative unsupervised update method, i.e. through LLM reflected mutation and rejection?

5.2. Evaluation Tasks

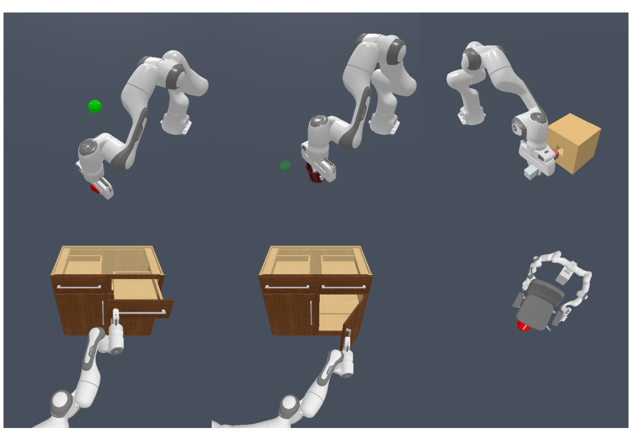

Figure 2. Six evaluation tasks from ManiSkill2: PickCube, PickSingleYCB, PegInsertionSide, OpenCabinetDoor, OpenCabinetDrawer, PushChair.

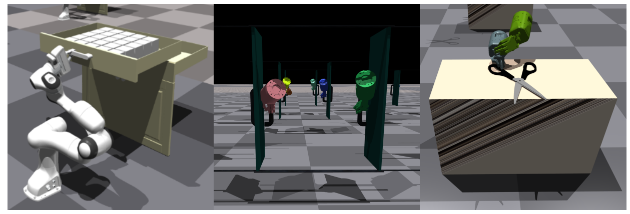

Figure 3. Three Isaac Gym evaluation tasks: Franka Cabinet, Shadow Hand Open Door Outwards, Shadow Hand Open Scissor

Following the objective, we evaluated our framework on 6 manipulation tasks in ManiSkill2 (Gu et al., 2023) as illustrated in Fig 2. The tasks includes rigid and articulated object manipulation with fixed-based manipulator, single-arm and dual-arm mobile manipulator.

5.3. Baselines

We compare our learnt reward function with two other reward functions. One trains policy with expert-designed oracle rewards from the original environment implementations, through which we examine the efficacy of our learnt reward function for $\mathcal{H}1$. Another trains policy with the intially proposed reward function from LLM analysed with CoT, but the parameterization stays fixed throughout the training. Through this comparison we aim to examine the efficacy of self-alignment update for $\mathcal{H}2$.

We also compare our method with two baseline approaches of reward generation with LLM: *Text2Reward* (Xie et al., 2023) on four overlapping ManiSkill2 tasks and *Eureka* (Ma et al., 2023) on three tasks implemented with Isaac Gym (Makoviychuk et al., 2021).

- ***Text2Reward*** generates dense reward generation with LLM by similarly providing the Pythonic environment abstraction and task description to LLM. The reward function can be further updated iteratively with LLM by taking human-feedback.
- ***Eureka*** generates high-performance reward function in an evolutionary manner. It feeds the environment script to LLM and generates multiple reward functions per time for policies training in-parallel. The batch success rates are fed back to LLM for reward functions' reflection and mutation then trains the next batch policies. Such loop is iterated until meets the termination condition.

We performed the self-alignment update on reward functions generated without human feedback in *Text2Reward* to examine if the improvement of such self-alignment update for $\mathcal{H}2$ holds, does it still hold for reward functions generated externally. We also compare the training and token efficiency with alternative unsupervised method proposed in *Eureka* for $\mathcal{H}3$ on 3 tasks shown in Fig 3.

5.4. Training Setup

For policy training, we adopted the same RL implementation and hyper-parameter setting that the baseline methods used, correspondingly are SAC (Haarnoja et al., 2018) from stable-baselines3 (Raffin et al., 2021) and PPO (Schulman et al., 2017) from rl_games (Makoviichuk & Makoviychuk, 2021). The success rate threshold to query parameter adjustment is 50% for SAC and 10% for PPO. For GPT, we used GPT-4 with API model name gpt-4-0613. As gpt-4-0314 used in *Eureka* is deprecated, we validated over 10 iterations that the new model is able to generate 11.00 ± 1.34 out of 16 successfully executed reward functions compared to $9/16$ from the author thus should not degenerate its performance.

For reward update, we set the rationality coefficient $\beta = 0.9$, feedback at every 10000 training steps for ManiSkill2, 100 epochs for Isaac Gym with M=5 roll-out samples from the latest policy and N=5 for sampling from the reward histogram. The training step equals the exploration step in our setup. For reward update with Metropolis-Hastings algorithm, our customized implementation is built upon APReL (Bıyık et al., 2022b). The burn-in period is 200 iterations and the number of samples is 100. The proposal distribution follows Gaussian distribution as $\mathcal{N}(\bar{\theta}, 0.2)$ on normalized parameters then is clipped to [0, 1].

In practice, as the reward parameters may be multi-modal (Bıyık et al., 2023) thus multiple sets of values may fulfil the same level of ranking discrepancy reduction. To minimize the reward fluctuation over iterations, we model the posterior and update the reward by sampling in the feasible region within the distances to the current value of [1.0, 3.0, 5.0, 10.0] in parallel. This setting works for parameter value ranges from $\pm e^{-2} - e^{1}$ scale proposed by LLM in our experiments thus no additional tuning was implemented and may be further explored. The new parameters with a maximum discrepancy reduction will be accepted. If the same level of discrepancy reduction is achieved, we select the set of values that are the nearest to the current one.

5.5. Results

For evaluation, we report the success rates over five different seeds against the exploration steps on 6 ManiSkill2 manipulation tasks. The results are plotted Fig 5. We also conducted similar reward learning on reward functions that are generated from *Text2Reward* zero-shot cases with LLM. The comparison on policy trained with fixed-parameterized reward function and updated with self-alignment are plotted in Fig 6.

5.5.1. The learnt reward is able to induce optimal policy

We first analyse through objective $\mathcal{H}1$. Compared to the oracle reward, we observed that our method is able to consistently develop a policy close to policy trained with expert-designed oracle reward across all 6 ManiSkill2 tasks. In the case with a better reward design proposed by LLM in PushChair task, we are able to further push the performance from 60.59% to 83.65% with oracle reward gives at the peak 35.34%.

5.5.2. Self-Alignment Update Consistently Improves Training Efficacy and Efficiency

From Fig 5, it can be observed that LLM generated reward function can often lack of adequate numerical optimality to develop the optimal policy as indicated in our motivation and $\mathcal{H}2$. This can be demonstrated by the nearly zero success rate of policies trained with fixed-parameterized reward proposed by LLM for tasks Pick-Cube and Peg-Insertion. There are the same patterns for Pick-Cube and Push-Chair tasks on functions from *Text2Reward* in Fig 6 which is consistent with the original paper.

For all cases, the self-alignment update scheme effectively improves the performance reflected as (1) faster convergence; (2) higher success rate at convergence in general. We think the such effectiveness or improvement are mainly due to two reasons:

(1) LLM brings meaningful inductive bias to steer policy towards global-goal especially in multi-objective setting:

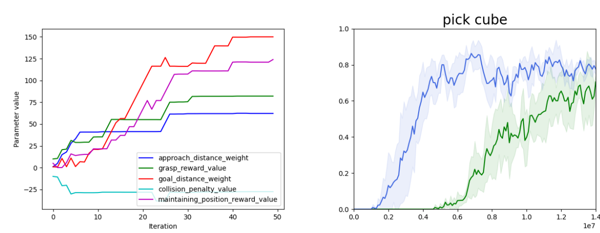

Figure 4. (a) parameter update over iteration for pick cube task; To better visualize the early shift the update is truncated to 50 iterations. (b) success rate of policy trained with the actively adjusted reward function and with the adjusted final reward function. The policy may not achieve the same performance trained with the final reward function learnt only.

Following the prompt, LLM decomposes task objectives as Do's and Don't step by step into dense reward and penalty terms. Such multi-objective learning especially for multi-step tasks can be challenging to learn altogether. By feedback on recent exploratory behaviors from the replay and exploitative behavior from current policy to LLM, LLM identifies the most and least optimal behaviors under the local context and steers it towards the global goal.

Such behavior is similar to introducing a curriculum for learning with periodic feedback and can be observed from the weight parameter change. For instance, for PickCube or PegInsertion, reward update were emphasized on encourag-

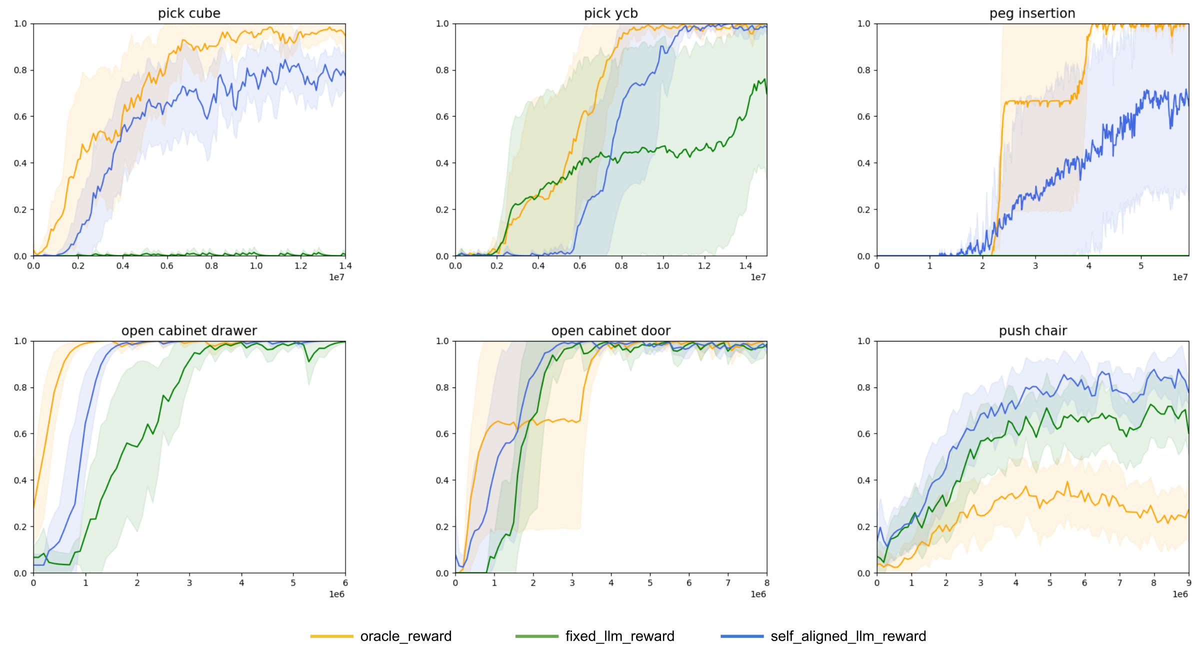

Figure 5. Success rates vs exploration steps on 6 ManiSkill Tasks with SAC. The updated reward is able to produce policy with similar performance to that is trained with oracle reward on 5 tasks. Compared to using fixed reward function genreated by LLM, our approach consistently improves the training with faster convergence rate and/or higher convergence performance

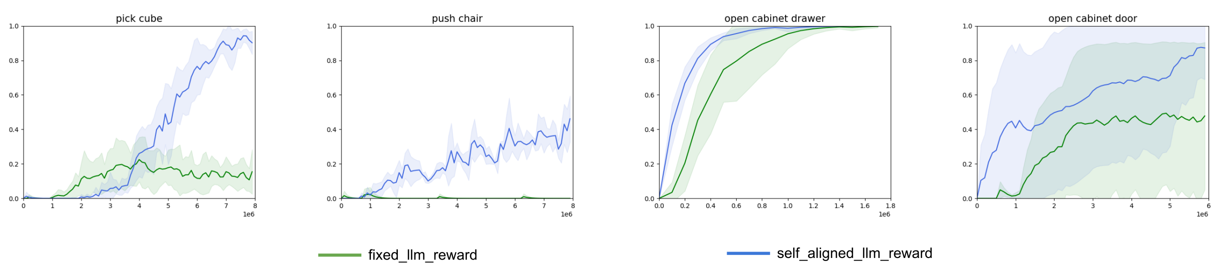

Figure 6. Success rate vs exploration steps with Text2reward zero-shot reward functions with SAC. Similarly, the periodically update reward function through self-alignment consistently improves the final performance achieved.

ing to approach first, later when contact starts with policy improving with `grasped=True` from the feedback samples start to appear, the weights further shift to increase grasping weights to encourage stable grasping. It is followed to increase transporting to goal pose weight with varied `distance_to_goal` in feedback samples. The pattern can be observed in Fig 4 and Fig 11 in Appendix. By iteratively doing so, the final policy was effectively guided toward the final target. We think this might be the main reason for this method to yield a faster convergence rate.

(2) Execution feedback and self-alignment prevents sub-optimal behavior due to parameter mis-specification:

As indicated in numerous existing literature, reward shaping is nontrivial as inappropriate shaping will introduce unintentional rewards or penalties that further leads to sub-optimal policy by exploiting such flaw. Meanwhile as we observed, the way LLM generates the reward code with parameterization directly as text generation task is not the best way as reward shaping requires. Through a closed-loop reward update via self-alignment, the shaping logic is enforced through the ranking further as pairwise preference. Such numerical errors or flaws can be reflected through execution feedback and ranking discrepancy that are later corrected via posterior maximization.

For example, in the Open-Cabinet-Drawer/Door task, when given a high reach reward and relatively lower pull reward, the policy converges to stay in touch with the handle, as the poor pulling action easily loses contact with the handle and yields a lower reward. However, LLM ranks with a preference for a larger pulling distance for later-stage action. The reward is updated to increase the pulling weight iteratively to correct such sub-optimality and move towards more pushing actions.

It is also a common pattern across tasks that LLM will recommend to reduce weight for a behaviour that is well-established for better exploration to develop the next step

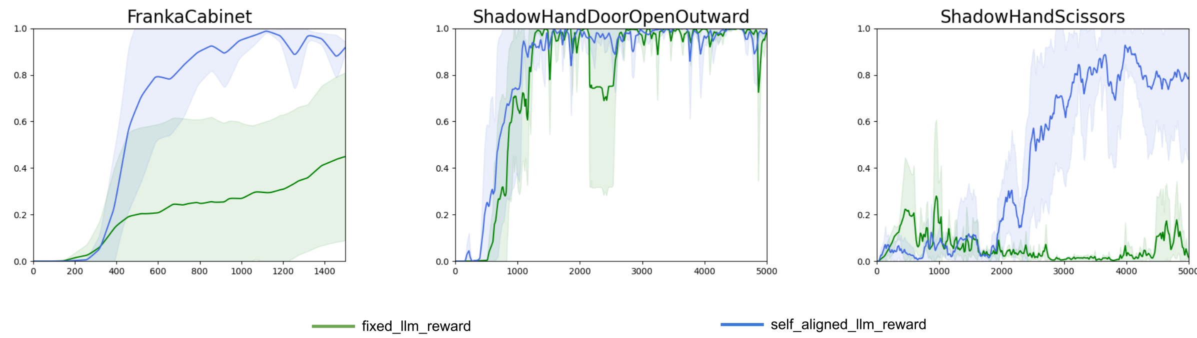

Figure 7. Success rate with PPO policy on 3 IsaacGyms tasks: (a) Franka Cabinet; (b) Push Door Outwards; (c) Open Scissor.

behaviour. One example taken from the Peg Insertion task is shown in Fig 8.

```
<CoT Skipped>
{
    # decrease to encourage moving towards the hole
    'distance_to_peg_weight': 70.1895,
    # decrease as the robot has already grasped the peg
    'alignment_ee_to_peg_weight': 130.5820,
    # keep the same as the robot is consistently grasping the peg
    'grasp_reward_value': 101.8941,
    # increase to encourage moving towards the hole
    'distance_to_hole_weight': 183.3966,
    # increase to encourage alignment with the hole
    'alignment_peg_to_hole_weight': 170.8059
}
```

Figure 8. Parameter adjustment for Peg-Insertion task from LLM with execution feedback. LLM proposed to reduce approach weight to better encourage the final goal reaching of the peg.

5.5.3. Significant Lower Token Consumption Compared to Mutation-Based Method

Lastly, we report the comparison between our method with *Eureka*. The success rates for training with fixed and updated reward functions are plotted in Fig 7. We also compare the `n_tokens` as the total input and output tokens consumed (Hu et al., 2023) at termination in table 1, which is either the training reaches the 1.0 success rate or 50 feedback iterations to prevent uncapped high cost for querying GPT-4.

n_tokens↓	*Cabinet*	*Door*	*Scissor*
Eureka	64,698 **(3)**	1,284,924 (-)	1,338,778 (-)
Ours	**15,633** (7)	**17,543 (7)**	**130,969(-)**

Table 1. Total number of tokens consumed throughout training. (n) indicates the number of iterations where success rate = 1.0 is achieved. (-) means termination at max iteration 50.

max sr ↑	*Cabinet*	*Door*	*Scissor*
Eureka	**100.00**%	0.00%	0.24%
Ours	**100.00**%	**100.00**%	**94.54**%

Table 2. Maximum success rate achieved throughout training.

On the three tasks, *Eureka* on average consumes tokens about 10 times of our method used per iteration. However, the total tokens consumed throughout training thus the cost can reach up to 100 times. We believe the evolutionary method in *Eureka* is a nice way to address the error-prone nature of code generated from LLM. However, it may not be efficient for parameterization setting. In addition, *Eureka* iterates in a relatively aggressive way which may also be related to the sample efficiency. The policies are trained for 5 epochs before the success rate feedback and retraining, where we observe hundreds to thousands of epochs are often required for visible progress on success rate across three tasks. This might be more pronounced with sparser-reward tasks. Lastly, though LLM mutates through heuristic reflection, it does not ensure such mutated reward will perform better rather than fluctuate which was observed for open door and scissor tasks as shown in table 2. Through self-alignment, such objective is quantified in the ranking discrepancy that aligns with the objective of reward design, and optimized towards a reduction in such measure.

5.6. Ablation Study

To better understand how individual components, i.e. self-alignment (SA) and active adjustment (AA) of parameters contribute to the final performance, we conducted the ablation study that updates the reward parametrization by:

1. enforcing the ranking self-consistency only.
2. actively adjusting through LLM reflection only.

The reward functions are updated iteratively at the same frequency. We analyse the effects on three tasks: Pick-Cube (16M steps), Open-Cabinet-Door (8M steps) and Push-Chair (8M steps) and report the final policy success rate averaged over 5 seeds in table 3 below:

sr ↑	*Fixed*	*SA Only*	*AA Only*	*SA + AA*
Cube	0.01±0.02	0.30±0.15	0.47±0.49	**0.73±0.16**
Cabinet	0.97±0.05	0.97±0.03	**0.99±0.02**	**0.99±0.02**
Chair	0.69±0.16	0.74±0.14	0.76±0.14	**0.84±0.10**

Table 3. Success rate for the final policies trained with (1) fixed LLM reward; (2) updated reward with self-alignment only; (3) updated reward with active parameter adjustment only; (4) updated reward that uses both strategies as proposed in this paper.

It is observed that both schemes improved the performance compared to using a fixed LLM reward across the three tasks, while combining the two strategies consistently yields the highest improvement. Actively adjusting the parameters in general leads to better policies compared to only enforcing the reward ranking self-consistency. This is aligned with the underlying mechanisms of the two strategies that the self-alignment aims to prevent error from potential reward misspecification, while active parameter adjustment aims to directly improve the reward saliency. However, with the task being more challenging or longer horizon which takes longer to establish the target behaviour, the active parameter adjustment was observed to produce drastic changes and value differences among parameters with performances stopped improving over iterations.

5.7. Limitations and Future Works

In this section, we analyse a few limitations in this method and potential future works. First, to summarize the execution, the current method reports using the reward features from the last step, as concatenation over steps will be lengthy for 2/500-step tasks and dilute the useful information. Thus it works for tasks where execution can be evaluated based on the final configuration, but not where trajectory matters such as walking in an S shape. With the development of a more powerful VLM, we hope the feedback will be more informative to remove such limitations. Another limitation lies in the preference ranking. Currently, we assume all rankings are equally better or worse, However, for sparse-reward tasks such as peg insertion, some successful experiences are significantly better and integrating it will enable more efficient training, which we hope to address in future work.

6. Conclusion

In this work we designed a framework to utilize LLM to propose the features and parameterization of reward function, and iteratively update the parameters through a self-alignment process in the absence of human intervention. More specifically, we periodically feedback policy to LLM for ranking. The ranking is later served as pairwise preference to align and update the reward function. When the ranking is fully aligned yet no effective policy is developed, we also query LLM for active reward parameter adjustment under the same framework of self-alignment. We validated the framework in 6 ManiSkill2 tasks and 3 IsaacGym tasks and shows the framework is able to induce optimal policy effectively and efficiently, where it can often fail without such self-alignment update. The method is also proven to be significantly token efficient comparing to alternative mutation-based method.

Impact Statement

This paper contributes to the field of robotics and machine learning by introducing a method that enhances the learning of reward functions in robots automatically, utilizing Large Language Models (LLMs). While our research primarily focuses on technical advancements, we acknowledge its broader societal implications. The refinement of reward functions in autonomous systems has the potential to benefit sectors like healthcare, manufacturing, and services by improving efficiency and reliability.

We recognize the ethical importance of ensuring that autonomous systems operate in alignment with societal norms and values. Our work aims to improve the precision of reward functions and better alignment of social preference that could potentially regulate behaviors including unethical or socially biased behaviors. It can produce a more predictable and trustworthy robotic behavior. We emphasize the need for ongoing interdisciplinary dialogue to address the ethical challenges and ensure that advancements in robotics and machine learning are developed and applied responsibly, aligning with broader societal interests.

References

Abbeel, P. and Ng, A. Y. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, pp. 1, 2004.

Adeniji, A., Xie, A., Sferrazza, C., Seo, Y., James, S., and Abbeel, P. Language reward modulation for pretraining reinforcement learning. *arXiv preprint arXiv:2308.12270*, 2023.

Akkaya, I., Andrychowicz, M., Chociej, M., Litwin, M., McGrew, B., Petron, A., Paino, A., Plappert, M., Powell, G., Ribas, R., et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.

Bıyık, E., Losey, D. P., Palan, M., Landolfi, N. C., Shevchuk, G., and Sadigh, D. Learning reward functions from diverse sources of human feedback: Optimally integrating demonstrations and preferences. *The International Journal of Robotics Research*, 41(1):45–67, 2022a.

Bıyık, E., Talati, A., and Sadigh, D. Aprel: A library for active preference-based reward learning algorithms. In *2022 17th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pp. 613–617. IEEE, 2022b.

Bıyık, E., Huynh, N., Kochenderfer, M. J., and Sadigh, D. Active preference-based gaussian process regression for reward learning and optimization. *The International Journal of Robotics Research*, pp. 02783649231208729, 2023.

Bradley, R. A. and Terry, M. E. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952.

Brohan, A., Chebotar, Y., Finn, C., Hausman, K., Herzog, A., Ho, D., Ibarz, J., Irpan, A., Jang, E., Julian, R., et al. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on Robot Learning*, pp. 287–318. PMLR, 2023.

Chen, T., Tippur, M., Wu, S., Kumar, V., Adelson, E., and Agrawal, P. Visual dexterity: In-hand reorientation of novel and complex object shapes. *Science Robotics*, 8 (84):eadc9244, 2023.

Gu, J., Xiang, F., Li, X., Ling, Z., Liu, X., Mu, T., Tang, Y., Tao, S., Wei, X., Yao, Y., et al. Maniskill2: A unified benchmark for generalizable manipulation skills. *arXiv preprint arXiv:2302.04659*, 2023.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pp. 1861–1870. PMLR, 2018.

Hadfield-Menell, D., Milli, S., Abbeel, P., Russell, S. J., and Dragan, A. Inverse reward design. *Advances in neural information processing systems*, 30, 2017.

Ho, J. and Ermon, S. Generative adversarial imitation learning. *Advances in neural information processing systems*, 29, 2016.

Hoegerman, J. and Losey, D. Reward learning with intractable normalizing functions. *IEEE Robotics and Automation Letters*, 2023.

Hu, M., Mu, Y., Yu, X., Ding, M., Wu, S., Shao, W., Chen, Q., Wang, B., Qiao, Y., and Luo, P. Tree-planner: Efficient close-loop task planning with large language models. *arXiv preprint arXiv:2310.08582*, 2023.

Huang, W., Abbeel, P., Pathak, D., and Mordatch, I. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning*, pp. 9118–9147. PMLR, 2022.

Ke, L., Choudhury, S., Barnes, M., Sun, W., Lee, G., and Srinivasa, S. Imitation learning as f-divergence minimization. In *Algorithmic Foundations of Robotics XIV: Proceedings of the Fourteenth Workshop on the Algorithmic Foundations of Robotics 14*, pp. 313–329. Springer, 2021.

Kwon, M., Xie, S. M., Bullard, K., and Sadigh, D. Reward design with language models. *arXiv preprint arXiv:2303.00001*, 2023.

Lee, K., Smith, L., and Abbeel, P. Pebble: Feedback-efficient interactive reinforcement learning via relabeling experience and unsupervised pre-training. *arXiv preprint arXiv:2106.05091*, 2021.

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

Liu, Z., Bahety, A., and Song, S. Reflect: Summarizing robot experiences for failure explanation and correction. *arXiv preprint arXiv:2306.15724*, 2023.

Luce, R. *Individual Choice Behavior: A Theoretical Analysis*. Wiley, 1959. URL https://books.google.com.sg/books?id=c519AAAAMAAJ.

Ma, Y. J., Liang, W., Wang, G., Huang, D.-A., Bastani, O., Jayaraman, D., Zhu, Y., Fan, L., and Anandkumar, A. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023.

Makoviichuk, D. and Makoviychuk, V. rl-games: A high-performance framework for reinforcement learning. https://github.com/Denys88/rl_games, May 2021.

Makoviychuk, V., Wawrzyniak, L., Guo, Y., Lu, M., Storey, K., Macklin, M., Hoeller, D., Rudin, N., Allshire, A., Handa, A., et al. Isaac gym: High performance gpu-based physics simulation for robot learning. *arXiv preprint arXiv:2108.10470*, 2021.

Manchester, I. R., Mettin, U., Iida, F., and Tedrake, R. Stable dynamic walking over uneven terrain. *The International Journal of Robotics Research*, 30(3):265–279, 2011.

Mehta, S. A. and Losey, D. P. Unified learning from demonstrations, corrections, and preferences during physical human-robot interaction. *arXiv preprint arXiv:2207.03395*, 2022.

Ng, A. Y., Russell, S., et al. Algorithms for inverse reinforcement learning. In *Icml*, volume 1, pp. 2, 2000.

Palan, M., Landolfi, N. C., Shevchuk, G., and Sadigh, D. Learning reward functions by integrating human demonstrations and preferences. *arXiv preprint arXiv:1906.08928*, 2019.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html.

Sadigh, D., Dragan, A. D., Sastry, S., and Seshia, S. A. *Active preference-based learning of reward functions*. 2017.

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

Singh, I., Blukis, V., Mousavian, A., Goyal, A., Xu, D., Tremblay, J., Fox, D., Thomason, J., and Garg, A. Progprompt: Generating situated robot task plans using large language models. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 11523–11530. IEEE, 2023.

Valsecchi, G., Grandia, R., and Hutter, M. Quadrupedal locomotion on uneven terrain with sensorized feet. *IEEE Robotics and Automation Letters*, 5(2):1548–1555, 2020.

Wang, Y., Xian, Z., Chen, F., Wang, T.-H., Wang, Y., Fragkiadaki, K., Erickson, Z., Held, D., and Gan, C. Robogen: Towards unleashing infinite data for automated robot learning via generative simulation. *arXiv preprint arXiv:2311.01455*, 2023.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35: 24824–24837, 2022.

Xie, T., Zhao, S., Wu, C. H., Liu, Y., Luo, Q., Zhong, V., Yang, Y., and Yu, T. Text2reward: Automated dense reward function generation for reinforcement learning. *arXiv preprint arXiv:2309.11489*, 2023.

Yu, W., Gileadi, N., Fu, C., Kirmani, S., Lee, K.-H., Arenas, M. G., Chiang, H.-T. L., Erez, T., Hasenclever, L., Humplik, J., et al. Language to rewards for robotic skill synthesis. *arXiv preprint arXiv:2306.08647*, 2023.

A. Appendix

A.1. Prompt Details

A.1.1. BACKGROUND PROMPT

First, we list the prompt on the task training background using reinforcement learning and available observations as the fixed observation dictionary that LLM can access. An additional wrapper was added for ManiSkill2 tasks to create such a fixed observation dictionary as such information is scattered in the original environment script. By doing this, LLM can be isolated from the environment source code and communicate with it through only observation and action. This is to mimic a fixed robotic system equipped with adequate sensory information that is expected to be versatile and acquire different skills when given the right rewards:

```
Our goal is to learn a set of sensory-motor skills from language concepts with multi-modal
   sensory observations with robots interacting with the environment and learn through
  trial and error in a reinforcement learning manner.  In the environment, we have:
(1) a robot (access by env.robot) which we can communicate and control using the robot
  interface as showed previously;
(2) a target object (env.target_object) to be manipulated. None if the task does not
  involve target object.

The available observations from the environment are stored in a dictionary obs which
  includes:
1. obs['joint_positions']: robot joint positions;
2. obs['joint_velocities']: robot joint velocities;
3. obs['ee_pose']: end effector pose;
4. obs['ee_contact_force']: the external contact force at the robot end effector;
5. obs['target_object_pose']: the pose of the target object
6. obs['goal_pose']: the goal pose for the target object to reach
7. obs['distance_to_target']: the distance between the end effector and the target object.
8. obs['distance_to_goal']: the distance between the target object and the goal pose
9. obs['in_contact']: a boolean value if the robot is in contact with a target object if
  there is
10. obs['in_collision']: a boolean value if the robot is in collision with the environment
11. obs['action]: the actions robot applied in the last step

Note, you can check
(1) binary grasped status between the gripper and the target object with grasped = env.
  robot.is_graspping(env.target_object)
(2) binary collision status between the robot and the environment with collision_detected
  = env.detect_collision()

We plan to learn different manipulation skills in this environment.
The robot is supposed to learn a sensorimotor policy with a neural network using
  reinforcement learning. When given observation as input the policy outputs the end
  effector pose action. The goal of the policy is to learn optimal action decision-making
  for a given skill.
```

A.1.2. REWARD PROMPT WITH TASK DESCRIPTION

We parse all input arguments except observation and action as the parameters to update automatically. The reward features are automatically parsed inside the get_reward() function as scalar local variables. As the reward signal is a scalar value, the features that are most relevant for such reward calculation should be a scalar as well. For replay buffer storage, instead of storing the raw reward value, we store the reward features. For reward relabeling, we calculate the new reward with the saved reward features using updated parameters.

```
To start, please analyze for task <task description>.

Please organize your answer into three parts:
1) describe the skill. Break down the task into substeps of actions as dos that the robot
  needs to achieve step by step, then turn them into reward terms step by step. During
  this sequence of actions, what are positive actions that will prompt more effective
  explorations? Include them in reward terms, Meanwhile, what should be avoided? Design
  them into penalty terms.
```

```
2) The total reward should be a weighted sum of all reward and penalty terms. For each
  term, analyze what observations mentioned above are related to this term.
3) Based on the analysis from 2), translate the reward design into Python function
  get_reward().

Please only use features that are semantically directly related to this task and minimize
  additional considerations such as regularization.

Note:
- Please only use observations that are available in obs_vars and put all hyper-parameters
   in the input argument with the recommended value as default input.
- Do not assume and use functions that are not provided.
- Do include any magic number in the reward function.
- All reward or penalty terms in the get_reward function should end with "_reward" or "
  _penalty".
- In designing the get_reward() function, make sure you process the most related feature
  first and assign it to a variable with an intuitive name before calculating the reward.
  Otherwise, avoid naming the variables.
- If there are any hyper-parameters, put them as the input argument of the function with a
   recommended default value. Do not use any magic numbers in the function.
```

A.2. Feedback and Ranking Prompt

```
The reward function you provided is:
    <get_reward()>

Given execution observation for
  - data sample 0: <feature_name> = <feature_value>, ...
  - data sample 1: <feature_name> = <feature_value>, ...
  - data sample 2: <feature_name> = <feature_value>, ...
  - data sample 3: <feature_name> = <feature_value>, ...
  - data sample 4: <feature_name> = <feature_value>, ...

First, summarize how many steps are there in the task.
Then go through the data samples one by one and identify which stage the execution is at.
  Put objects at the same step into one cluster and list all the clusters. Note one sample
  should only belong to one cluster. Then rank samples from better to worse within a
 cluster for all clusters. Lastly, concatenate the ranking in a list by always putting the
  cluster at a later step in front.
Make sure the last line of the reply contains and only contains the final list.

Example of opening drawer task:
 - data 1, 4 are in the reaching stage, where 1 is closer than 4. The ranking for this
  cluster is [1, 4].
 - data 2, 5 are in the pulling stage, where 5 are pulled more than 2. The ranking for
  this cluster is [5, 2]
Pulling is at a later stage than reaching. The final result is:
[5, 2, 1, 4]
```

A.3. Parameter Reflection Prompt

```
Given the reward function for <task_description> is:
    <get_reward()>

<repeat execution observation>

Go through each data sample and check if it succeeds in executing <skill description>.
 What action will encourage the current behaviour to be more likely to successfully
 execute open cabinet drawer? or if this is a multi-step task, which stages the current
 behaviour is at? what reward or penalty will prompt the current behaviour to produce
 meaningful exploration that contains the action of the next stage?

After going through all samples, count the times that each relevant reward or penalty term
  is mentioned.  Provide your chain of thought in plain text.
```

```
Lastly, output the identified hyper-parameter that is likely to prompt success or to the
 next stage behaviour as a dictionary.  The key is the hyper-parameter name and the value
 is the recommended new value. Comment behind each to indicate if the value is suggested
 to increase or decrease. Do not output anything after this.

For example:
Result:
{'param_a': 1.0, 'param_b': 1.0}
```

A.4. Prompt to Generate Observation Dictionary Automatically for Isaac Gym Tasks

As the original observations available are nicely structured and named in `compute_observation()` in Isaac Gym tasks, we leverage LLM itself to automatically generate the observation dictionary within this function that later will be used for reward proposing and calculation.

```
You are an expert in robot manipulation. Now the task is to <task description>. Provided
 this is the compute_observations function which are the observations you can get:

<compute_observation()>

Please append a snippet of code which is supposed to be put at the last step of the
 compute_observation() function that aggregates all observations into a dictionary with
 the variable name "obs_vars", where the key is the observation name and value is the
 corresponding value.
```

A.5. Task Description

The task description used in the prompt template for the 9 evaluation tasks are listed below in table 4:

Task	*Task Description*
Pick Cube	pick cube object and transport to the target position
Pick Ycb	pick ycb object and transport to the target position
Peg Insertion	insert peg into the side hole
Open Cabinet Drawer	open cabinet drawer as much as possible
Open Cabinet Door	open cabinet door as much as possible
Push Chair	push a swivel chair to a target 2D location on the ground
Franka Cabinet	open cabinet drawer as much as possible
Shadow Hand Open Door Outward	push the door as much as possible with left and right panels outwards with two hands
Shadow Hand Scissor	open the scissor as much as possible with two hands

Table 4. Task descriptions for 9 tasks evaluated in the paper.

A.6. Example Reward Functions

A.6.1. Open Drawer

```
def get_reward(obs, alignment_weight=1.0, approach_weight=1.0, grasp_weight=1.0,
 pull_weight=1.0, collision_penalty_weight=1.0, non_progress_penalty_weight=1.0,
 distance_penalty_weight=0.1):

    # Extract relevant features from observations
    distance_to_handle = np.linalg.norm(obs['distance_to_target'])
    distance_to_goal = obs['distance_to_goal'][0]

    # Check for contact and collision
```

```
    contacted = env.robot.gripper_in_contact(env.target_object)
    collision_detected = env.detect_collision()

    alignment_reward = - alignment_weight * distance_to_handle

    # Approach reward: Encouraging the EE to get closer to the handle
    # Higher when the distance is smaller, so we take the negative distance
    approach_reward = - approach_weight * distance_to_handle

    # Grasp reward: Binary reward for making contact with the handle
    grasp_reward = grasp_weight * 1.0 if contacted else 0.0

    # Pull reward: Encouraging the EE to get the handle closer to the goal position
    # Higher when the distance is smaller, so we take the negative distance
    pull_reward = -pull_weight * distance_to_goal

    # Collision penalty: Penalizing any collision detected during interaction
    collision_penalty = -collision_penalty_weight * 1.0 if collision_detected else 0.0

    # Non-progress penalty: If there's no contact and no reduction in distance to the
     handle,
    # penalize to encourage the EE to move towards the handle
    non_progress_penalty = -non_progress_penalty_weight * 1.0 if not contacted and
     distance_to_handle > 0.02 else 0.0  # Threshold of 2cm

    # Excessive distance penalty: Discouraging the EE from being too far from the handle
    excessive_distance_penalty = - distance_penalty_weight * distance_to_handle if
     distance_to_handle > 0.15 else 0.0  # Threshold of 15cm

    # Combine rewards and penalties into total reward
    total_reward = alignment_reward + approach_reward + grasp_reward + pull_reward +
     collision_penalty + non_progress_penalty + excessive_distance_penalty

    return total_reward
```

A.6.2. PUSH CHAIR

```
def get_reward(obs, approach_weight=1.0, movement_weight=1.0, collision_penalty_weight
 =1.0):

    # Reward for minimizing the distance between the robot gripper and the chair
    gripper_to_chair_dist = np.linalg.norm(obs['distance_to_target'])
    approach_reward = -gripper_to_chair_dist  # Negative value: smaller distance is better

    # Reward for moving the chair towards the target position
    # Assuming the target position is part of the environment's state
    chair_to_target_dist = np.linalg.norm(obs['distance_to_goal'][:2])
    movement_reward = -chair_to_target_dist  # Negative value: smaller distance is better

    # Penalty for collisions
    collision_detected = env.detect_collision()
    collision_penalty = -1.0 if collision_detected else 0.0

    # Calculate total reward
    total_reward = approach_weight * approach_reward + movement_weight * movement_reward +
      collision_penalty_weight * collision_penalty

    return total_reward
```

A.6.3. SHADOW HAND OPEN SCISSOR

```
def get_reward(obs, handle_separation_weight=1.0, hand_on_handle_weight=1.0,
 stability_weight=0.1):
```

```
    # Extract relevant features from obs
    scissors_right_handle_pos = obs['scissors_right_handle_pos']
    scissors_left_handle_pos = obs['scissors_left_handle_pos']

    left_hand_pos = obs['left_hand_pos']
    right_hand_pos = obs['right_hand_pos']

    object_linvel = obs['object_linvel']
    object_angvel = obs['object_angvel']

    # Initially, the handles are next to each other, we want to maximize the distance
     between the scissor handles
    handles_distance = torch.norm(scissors_right_handle_pos - scissors_left_handle_pos, p
     =2, dim=1)

    # Handle Separation Reward (encourage the hands to move the scissor handles apart)
    handle_separation_reward = handle_separation_weight * handles_distance

    # Hand on Handle Reward (encourage the hands to maintain contact with the scissor
     handles)
    left_hand_on_handle_distance = torch.norm(left_hand_pos - scissors_left_handle_pos, p
     =2, dim=1)
    right_hand_on_handle_distance = torch.norm(right_hand_pos - scissors_right_handle_pos,
      p=2, dim=1)
    hand_on_handle_reward = - hand_on_handle_weight * (left_hand_on_handle_distance +
     right_hand_on_handle_distance)

    # Stability Penalty (minimize the linear and angular velocity of the scissors to
     ensure smooth opening)
    linear_velocity_magnitude = torch.norm(object_linvel, p=2, dim=1)
    angular_velocity_magnitude = torch.norm(object_angvel, p=2, dim=1)
    stability_penalty = - stability_weight * (linear_velocity_magnitude +
     angular_velocity_magnitude)

    # Combine rewards and penalties
    total_reward = handle_separation_reward + hand_on_handle_reward + stability_penalty

    return total_reward
```

A.7. Example Ranking from LLM

```
<Feedback>
Given execution observation for
  - data sample 0: distance_to_peg = 0.3771, alignment_error_ee_to_peg = 0.3695, grasped =
    False, collision_detected = True, peg_to_hole_distance = 0.1237,
   alignment_error_peg_to_hole = 0.0197.
  - data sample 1: distance_to_peg = 0.2108, alignment_error_ee_to_peg = 0.3156, grasped =
    False, collision_detected = False, peg_to_hole_distance = 0.12247,
   alignment_error_peg_to_hole = 0.0239.
  - data sample 2: distance_to_peg = 0.2523, alignment_error_ee_to_peg = 0.0064, grasped =
    False, collision_detected = False, peg_to_hole_distance = None,
   alignment_error_peg_to_hole = None.
  - data sample 3: distance_to_peg = 0.0914, alignment_error_ee_to_peg = 0.0174, grasped =
    True, collision_detected = False, peg_to_hole_distance = 0.3165,
   alignment_error_peg_to_hole = 0.0303.
  - data sample 4: distance_to_peg = 0.0474, alignment_error_ee_to_peg = 0.0139, grasped =
    True, collision_detected = False, peg_to_hole_distance = 0.2201,
   alignment_error_peg_to_hole = 0.0135.
  - data sample 5: distance_to_peg = 0.0334, alignment_error_ee_to_peg = 0.0106, grasped =
    True, collision_detected = False, peg_to_hole_distance = 0.1032,
   alignment_error_peg_to_hole = 0.0190.

<ChatGPT>
The task involves the following steps:
```

```
Approaching the Peg: Reduce the distance to the peg and align the end effector with it.
Grasping the Peg: Successfully grasp the peg.
Transporting Peg to Hole: Move the peg towards the hole while maintaining alignment.
Aligning Peg with Hole: Ensure the peg is properly aligned with the hole before insertion.
Now, let's analyze the data samples and categorize them based on the steps:

Data Sample Analysis

Approaching the Peg Stage
Sample 0: Collision occurred, failed to grasp.
Sample 1: No collision, but failed to grasp. Better alignment than sample 0.
Sample 2: No collision, best alignment, but failed to grasp.

Approaching Cluster Ranking:  [2, 1, 0]

Grasping the Peg Stage (None)

Transporting Peg to Hole Stage
Sample 3: Grasped the peg, but the peg is far from the hole and alignment is not perfect.
Sample 4: Grasped, closer to the hole and better alignment than sample 3.
Sample 5: Grasped, closest to the hole and better alignment than sample 4.
Transporting Cluster Ranking:  [5, 4, 3]

Aligning Peg with Hole Stage (None)

Final Result
Putting the clusters in order based on the steps in the task, with later stages first:

[5, 4, 3, 2, 1, 0]
```

A.8. Ranking Failure Cases

A.8.1. Small Numerical Difference

As mentioned in Section 3, LLM can mis-rank especially when there are subtle numerical differences. In the example shown below, data sample 4 should be the closest reaching but is ranked as 4th while other samples are ranked in the right order:

```
Given execution observation for
  - data sample 0: drawer_joint_value = 0.0000, distance_to_drawer = 0.0589, alignment =
   0.9999.
  - data sample 1: drawer_joint_value = 0.0000, distance_to_drawer = 0.0527, alignment =
   1.0000.
  - data sample 2: drawer_joint_value = 0.0000, distance_to_drawer = 0.0205, alignment =
   1.0000.
  - data sample 3: drawer_joint_value = 0.0000, distance_to_drawer = 0.2089, alignment =
   0.9851.
  - data sample 4: drawer_joint_value = 0.0000, distance_to_drawer = 0.0168, alignment =
   1.0000.

<analysis skipped>
All data samples are in the reaching stage, the ranking for the reaching stage cluster is
 [2, 1, 0, 4, 3]
(from best to worst distance_to_drawer).
```

A.8.2. Ambiguous Task Description

In general, to generate a ranking that is useful to guide the task policy, it is found that LLM requires the task description to be explicit about the task goal. When given an abstract description of a task such as ”Pull a drawer”, humans naturally associate with experience unspoken here, as a continuous change in the drawer prismatic joint and the more it is pulled the better. However, LLM tends to interpret words more literally with less such association. In the example shown below, LLM interpreted the task with the description ”open the cabinet drawer” as a binary status goal. In the ranking of opened drawer cases, LLM seeks better orientation alignment between the end effector and opened drawer for higher ranking instead of the pulled length. Adding a goal such as "pull the cabinet drawer as much as possible" helps to remove such ambiguity, but in general, it hints at the requirement to carefully design the task description that no ambiguity presents, which can be less obvious for humans.

```
Given execution observation for
  - data sample 0: drawer_joint_value = 0.0088, distance_to_drawer = 0.0988, alignment =
   0.9998.
  - data sample 1: drawer_joint_value = 0.0510, distance_to_drawer = 0.1748, alignment =
   0.9866.
  ...
  - data sample 4: drawer_joint_value = 0.0651, distance_to_drawer = 0.6472, alignment =
   0.7687.

<analysis skipped>
Now we can cluster the data samples:
- Pulling stage: [0, 1, 4]

Within each cluster, we rank the samples from better to worse:
- Pulling stage: [0, 1, 4] (0 has the best alignment and smallest distance, 1 has the next
  best alignment and distance, 4 has the worst alignment and largest distance)
```

A.9. Example Parameter Reflection from LLM

```
Data sample 0:
- The robot has successfully grasped the peg and is moving towards the hole. The distance
 to the peg and the alignment error between the end effector and the peg are both low,
 indicating that the robot is holding the peg correctly. The distance to the hole and the
 alignment error between the peg and the hole are relatively high, suggesting that the
 robot is in the process of moving the peg towards the hole. No collision is detected. The
  robot should be encouraged to continue reducing the distance to the hole and aligning
 the peg with the hole.

Data sample 1:
- The robot is still holding the peg and has moved closer to the hole, as indicated by the
  reduced distance to the hole. The alignment error between the end effector and the peg
 has increased slightly, but the alignment error between the peg and the hole has also
 increased. This suggests that while the robot is moving closer to the hole, it may not be
  aligning the peg correctly. No collision is detected. The robot should be encouraged to
 improve the alignment between the peg and the hole while continuing to reduce the
 distance to the hole.

Data sample 2:
- The robot continues to hold the peg and move closer to the hole. Both the distance to
 the peg and the alignment error between the end effector and the peg have decreased,
 indicating that the robot is holding the peg more correctly. The distance to the hole and
  the alignment error between the peg and the hole have also decreased, suggesting that
 the robot is aligning the peg with the hole more accurately. No collision is detected.
 The robot should be encouraged to continue this behavior.

Data sample 3:
- The robot is still holding the peg, but the distance to the hole has increased,
 suggesting that the robot may have moved away from the hole. The alignment error between
 the end effector and the peg has slightly increased, but the alignment error between the
 peg and the hole has decreased significantly. This suggests that while the robot may have
  moved away from the hole, it has improved the alignment of the peg with the hole. No
 collision is detected. The robot should be encouraged to move closer to the hole while
 maintaining the alignment of the peg with the hole.

Data sample 4:
- The robot continues to hold the peg, but the distance to the hole has significantly
 increased, suggesting that the robot has moved further away from the hole. The alignment
 error between the end effector and the peg has also increased, indicating that the robot
 may not be holding the peg correctly. However, the alignment error between the peg and
```

```
 the hole has decreased significantly, suggesting that the robot has improved the
 alignment of the peg with the hole. No collision is detected. The robot should be
 encouraged to move closer to the hole and improve the way it holds the peg.

Count of relevant reward or penalty terms:
- approach_reward: 5 times
- ee_to_peg_alignment_reward: 5 times
- grasp_reward: 5 times
- lift_and_move_reward: 5 times
- peg_to_hole_alignment_reward: 5 times
- collision_penalty: 0 times

Result:
{'distance_to_peg_weight': 60.0, # increase to encourage the robot to reduce the distance
 to the peg
 'alignment_ee_to_peg_weight': 80.0, # increase to encourage the robot to improve the
  alignment between the end effector and the peg
 'grasp_reward_value': 70.0, # decrease as the robot is already successfully grasping the
  peg
 'distance_to_hole_weight': 90.0, # increase to encourage the robot to reduce the distance
   to the hole
 'alignment_peg_to_hole_weight': 75.0, # increase to encourage the robot to improve the
  alignment between the peg and the hole
 'collision_penalty_value': -20.0} # # no change as no collision is detected
```

A.10. Visualization for Alignment over Iterations

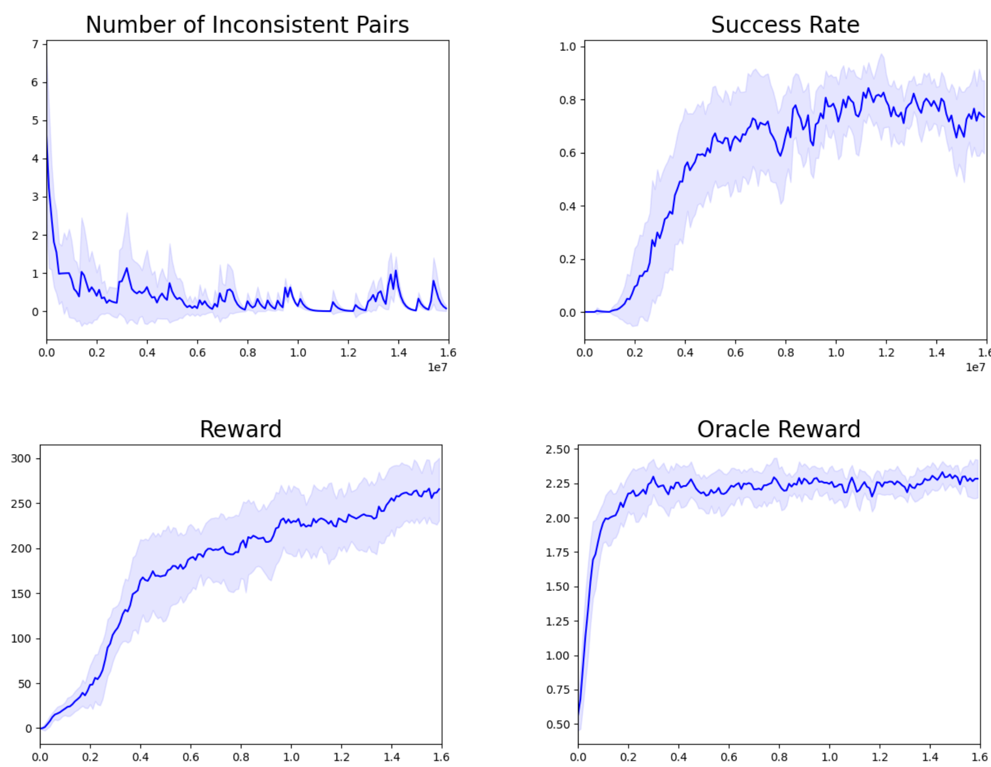

Figure 9. The number of inconsistencies, success rate, self-alignment updated reward and the corresponding oracle reward with training steps for the pick cube task.

To better understand how the alignment evolves with the learning process, we plot the number of inconsistent pairs, success rate, self-alignment updated reward and the corresponding original oracle reward with training steps over five seeds in Fig

9. It is observed that in the early stage, the inconsistency in general decreases quickly but also spikes with new sub-step behavior picked up. It gradually decrease to around 0 and maintain here.

A.11. Visualization for Reward Update Triggers

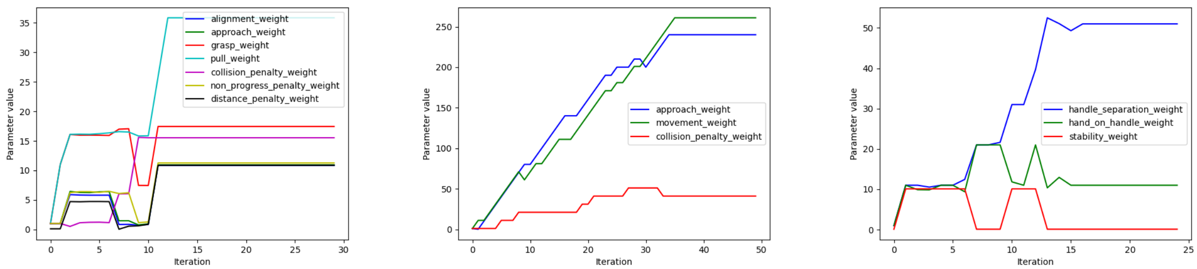

Figure 10. Visualization for weight update over iterations (1) Open Drawer, (2) Push Chair; (3) Shadow Hand Open Scissor

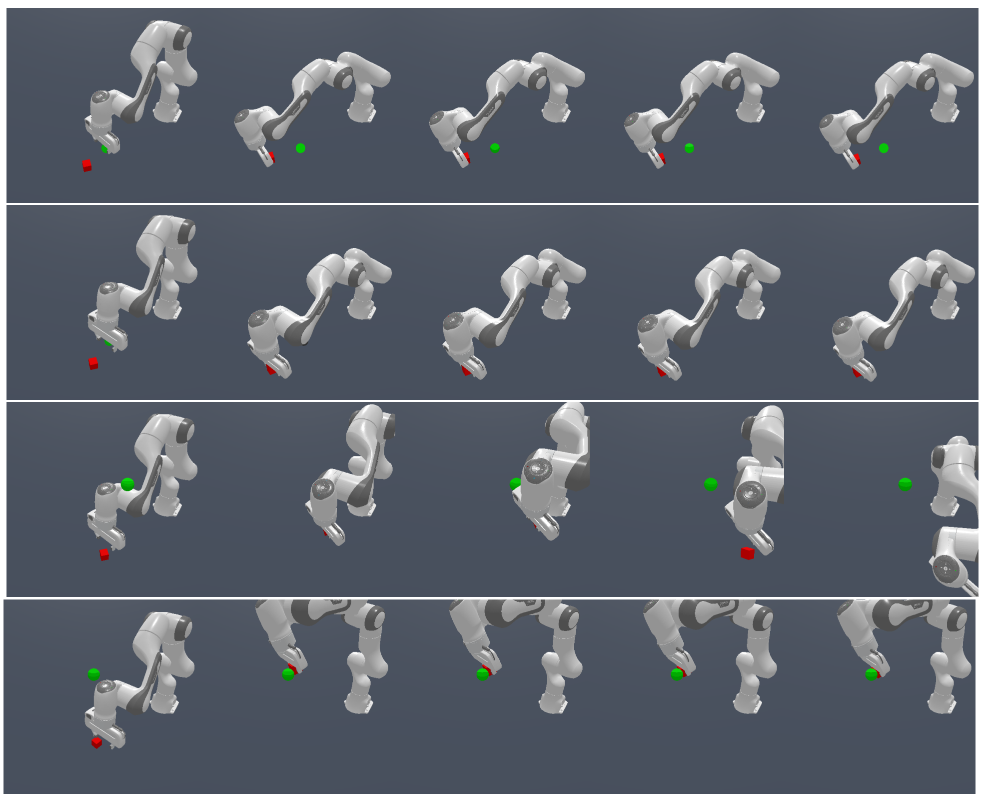

Figure 11. Visualization of policy behaviour around iterations where major parameter changes as visualized in Fig 4 (a) were observed. They are (from top to bottom) at iteration 5, 12, 23, and 35. The weight updated is found to be interpretable and directly correlates to the policy behaviour.

The weights for runs on tasks Open Drawer, Push Chair, Shadow Hand Open Scissors are plotted as shown in Fig 10. Where we can observe weights-related parameters, especially for pick cube, open drawer and open scissor, weights tend to change transiting from approach, then grasp (if involved) then final goal reaching with the policy improving. We can also observe multiple cases where approaching/grasping weight decreases significantly to allow exploration for goal reaching stage.

In Fig 11. We further visualized the policy rollout at iteration 5, iteration 12, iteration 23 and iteration 35 as shown with major reward update pattern changes as shown in Fig 4 (a). The weight updated is found to be interpretable and directly correlates to the policy behaviour:

- *Iteration 5*: The model gradually starts to reach to object every time from random behavior, where it is observed the approaching weight starts to stabilize;
- *Iteration 12*: The model starts to grasp the cube and move to random locations. Similarly, a slower increase in grasp weight is observed and goal-reaching and maintaining weight start to increase more rapidly;
- *Iteration 23*: Goal reaching and maintaining weight slows down, and grasping and approaching weight starts to increase once again. From the rollout policy, the policy learns to pick up the cube and tries to reach the goal fast while the cube becomes unstable and slips out of hand where re-grasping happens.
- *Iteration 35*: Goal reaching and maintaining weight keeps increasing while other parameters stay fixed. It is observed the robot can pick up the cube to go near the goal position although not exactly. The two weight terms stabilize in a few more iterations.