LARG^2, Language-based Automatic Reward and Goal Generation

Julien Perez
Naver Labs Europe
France
Julien.Perez@naverlabs.com

Denys Proux
Naver Labs Europe
France
Denys.Proux@naverlabs.com

Claude Roux
Naver Labs Europe
France
Claude.Roux@naverlabs.com

Michael Niemaz
Naver Labs Europe
France
Michael.Niemaz@naverlabs.com

Abstract: Goal-conditioned and Multi-Task Reinforcement Learning (GCRL and MTRL) address numerous problems related to robot learning, including locomotion, navigation, and manipulation scenarios. Recent works focusing on language-defined robotic manipulation tasks have led to the tedious production of massive human annotations to create dataset of textual descriptions associated with trajectories. To leverage reinforcement learning with text-based task descriptions, we need to produce reward functions associated with individual tasks in a scalable manner. In this paper, we leverage recent capabilities of Large Language Models (LLMs) and introduce LARG^2, Language-based Automatic Reward and Goal Generation, an approach that converts a text-based task description into its corresponding reward and goal-generation functions We evaluate our approach for robotic manipulation and demonstrate its ability to train and execute policies in a scalable manner, without the need for handcrafted reward functions.

Keywords: Robots Learning, Goal Conditioned Reward Learning, Large Language Models, code generation, Reward function

1 Introduction

In the field of robotic manipulation, decision models are evolving from optimal control approaches towards policy learning through Multi-task and Goal-Conditioned Reinforcement Learning [1]. Following this line of work, multi-modal task definition [2, 3], associated with reasoning and action planning abilities facilitated by LLMs [4], enables agents to adapt to real-world uncertainty which is hardly handled with traditional robotic control.

Several strategies, such as behavioral cloning [5, 6], transfer learning [7, 8, 9], or interactive learning [10, 11, 12], have been proposed for that matter. While promising, these former approaches are hardly scalable as they require human demonstrations or handcrafted trajectories. In fact, **the difficulty of connecting textual descriptions of tasks with their associated computational goals and reward functions is what has led to these unscalable solutions**.

In this paper, we introduce LARG^2, Language-based Automatic Reward and Goal Generation. For a given sequential decision task, our method simply automates the generation of its corresponding reward function from a textual descriptions of it. We leverage common sense and reasoning capabilities offered by recent large language models in terms of text understanding and source code generation and embodied our approach in two settings. First, for a given environment, our approach samples the target goal conditioned by a textual description of a task which allows to train a corresponding policy using Goal-Conditioned Reinforcement Learning. Second, we directly generate executable reward functions according to the same task, which allows to train a corresponding policy using Multi-Task Reinforcement Learning. Finally, we evaluate both settings of LARG^2 over a set of language-formulated tasks in a tabletop manipulation scenario.

2 Preliminaries

Reinforcement Learning considers an agent which performs sequences of actions in a given environment to maximize a cumulative sum of rewards. Such problem is commonly framed as Markov Decision Processes (MDPs): $M = \{S, A, T, \rho_0, R\}$ [13, 14, 15]. The agent and its environment, as well as their interaction dynamics, are defined by the first components $\{S, A, T, \rho_0\}$, where $s \in S$ describes the current state of the agent-environment interaction and ρ_0 is the distribution over initial states. The agent interacts with the environment through actions $a \in A$. The transition function T models the distribution of the next state s_{t+1} conditioned with the current state and action $T : p(s_{t+1}|s_t, a_t)$. Then, the objective of the agent is defined by the remaining component of the MDP, $R : S \to \mathbb{R}$. Solving a Markov decision process consists in finding a policy $\pi : S \to A$ that maximizes the cumulative sum of discounted rewards accumulated through experiences.

In the context of robotic manipulation, a task commonly consists in altering the environment into a targeted state through selective contact [16]. So, tasks are expressed as $g = (c_g, R_G)$ pair where c_g is a goal configuration such as Cartesian coordinates of each element composing the environment or a textual description of it, and $R_G : S \times G \to \mathbb{R}$ is a goal-achievement function that measures progress towards goal achievement and is shared across goals. A goal-conditioned MDP is defined as : $M_g = \{S, A, T, \rho_0, c_g, R_G\}$ with a reward function shared across goals. In multi-task reinforcement learning settings, an agent solves a possibly large set of tasks jointly. It is trained on a set of rewards associated with each task. Finally, goals are defined as constraints on one or several consecutive states that the agent seeks to satisfy [17, 18, 19].

3 Related work

3.1 Challenges of reward definition and shaping

A sequential decision task requires defining an informative reward function to enable the reinforcement learning paradigm. Reward shaping consists in designing a function in an iterative process incorporating elements from domain knowledge to guide policy search algorithms. Formally, this can be defined as $R' = R + F$, where F is the shaping reward function, and R' is the modified reward function [20, 21]. As a main limitation, a reward function needs to be manually designed for each task. For instance [22] leveraged large number of human demonstrations and specific handcrafted definitions of tasks to train a robotic transformer. However, as MTRL can deal with a large set of goals and tasks to implement, such an approach becomes hardly scalable. In this work, we study how to leverage LLMs and the common-sense and prior knowledge embedded in such models to automate the paraphrasing of tasks and the generation of associated reward functions.

3.2 Large Language models for control

Goal-conditioned reinforcement learning has recently been successfully considered in the domain of robot control with textual descriptions of tasks. [3] has combined a text encoder, a visual encoder, and visual navigation models, to provide text-based instructions to a navigating agent. This paradigm has been further developed in [1] using the capabilities of large language models to support action planning, reasoning, and internal dialogue among the various models involved in manipulation tasks. Along these lines, [23] proposes to use the code generation capabilities of sequence-to-sequence language models to transform user instructions into a code-based policy. However, it involves an interactive design process for a hard-coded policy, rather than a task-conditional learning process and it does not benefit from a policy trained through reinforcement learning. Finally, [24] proposes to use language to control the reward value in a feedback loop to adapt robot behavior. In this

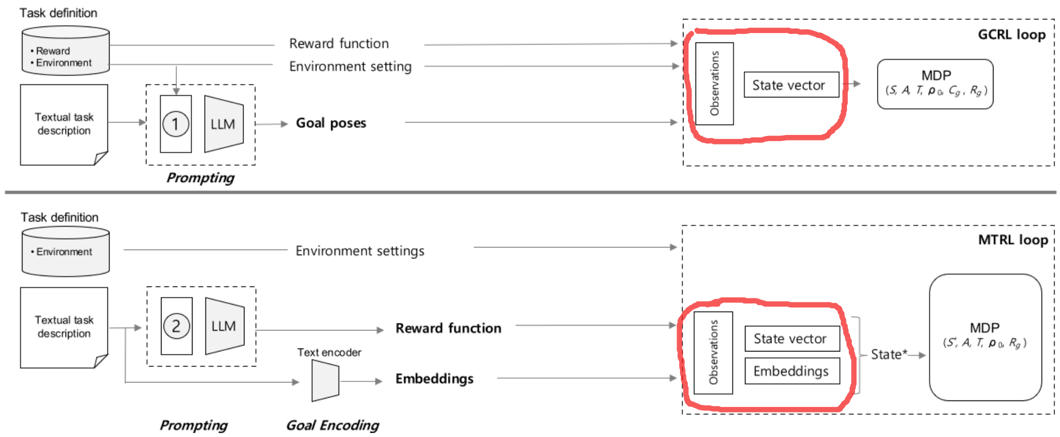


Figure 1: LARG^{2}transforms a textual task description into either 1) a goal to be used as input of a pre-existing reward function for GCRL, or 2) into a reward function for MTRL. We leverage pre-trained LLM with dedicated prompts for our generation procedure. For GCRL, the goal is appended to the state description given as input to the policy. For MTRL the text-based task description is encoded using a pre-trained language model to complement the state vector.

case, no code is generated and only the reward signal is modified according to user guidelines. Furthermore, it requires constant feedback from the user which makes it hardly scalable.

The closest to our work is [25] which proposes to automatically derive goals from a textual description of the task. However, the language remains limited to the logical descriptions of the expected configuration of the scene and the goal is reduced to a finite set of eligible targets. Our approach allows using natural language beyond logical forms, grounded with reasoning capabilities and enriched with common-sense captured in large pre-trained language models. Also related, [26] propose to train a conditional variational auto-encoder to create a language-conditioned goal generator. However, it assumes the existence of pre-trained goal-conditioned policies. Furthermore, no LLM background knowledge and code generation is considered to achieve this objective.

As far as our knowledge goes, our work is the very first attempt to leverage in-context prompting to enable training goal-condition and multi-task reinforcement learning agents with a textual description of the task in a scalable manner so that goal sampling and reward codes can be adapted according to the user-specific objectives.

4 Language-based Automatic Reward and Goal Generation

We introduce two methods using LLMs for handling textual descriptions of tasks as illustrated in Figure 1. First, *Automatic Goal Generation* uses LLMs to produce executable code that generates goals (c_g) to be used as parameters of a predefined goal-conditioned reward function (R_G). In this case, the trained policy can take as input a goal generated from such an LLM-produced function. Second, *Automatic Reward Generation* uses LLMs to generate a reward function associated with a given textual description of a task. In this second approach, the policy takes as input the textual description of the task. After this generation phase, we leverage the off-the-shelf multi-task reinforcement learning framework to train agents. The following sections describe our two approaches and detail their corresponding prompting strategies.

4.1 Automatic Goal Generation

Our first objective is to translate a textual task description with its constraints and guidelines into a goal. As a pre-requisite, we assume the existence of categories of tasks along with their environment settings and associated reward functions. These reward functions are parameterized with a goal. In tabletop robotic manipulation scenarios, a task consists in re-arranging a set of objects composing the scene. We assume that the goal is the set of target poses for all objects. Then, the reward function incorporates environment-dependent reward terms and Euclidian distance between the current pose of the objects and the target pose. Goals generated by LARG^2 are used in a GCRL learning setting to compute the reward signal at each step. The prompt design p, described in section 5.1, generates a function f returning eligible values for the targeted task such as $f \to c_g$ where $c_g = [goal\_values]$.

4.2 Automatic Reward Generation

The second utilization of LARG^2 generates the executable source code of a reward function according to a textual description of a task.

While Large Language Models could possibly support the full generation of complex reward functions, we propose to identify different parts in such a function, some being task-independent and others closely related to the task definition. In robotic manipulation, common task-independent components address bonuses for lifting the objects or penalties for the number of actions to achieve a given purpose. Task-dependent components are driven by the textual task description and align constraints with penalties and guidelines with bonuses. Both components are combined in a global reward function. Considering current LLM limitations, our experiments highlighted the fact that it is more efficient to perform such a decomposition to decrease the complexity of the generated code.

For the composition of the global reward function, we consider the existence of predefined categories of tasks with their environments, formalized using languages such as YAML [1] or Python [2] providing task dependent reward components such as what exists in repositories like Isaac_Gym [3]. LARG^2 aligns specific textual task description with a related task category and generates the task-dependent part of the reward function to form the global function.

4.3 Task-encoding and Policy

For GCRL, the input of the policy is composed of the environment state and the goal generated by LARG^2. For MTRL, the goal of each task is replaced by a textual description of the task. To do this, we use a pre-trained text encoder such as Google T5 [27, 28] to tokenize and encode the text into an embedding vector. This vector is added to the state vector in the training phase to caption tasks. This approach allows using textual descriptions of tasks as input to neural policies such as what is proposed by [2].

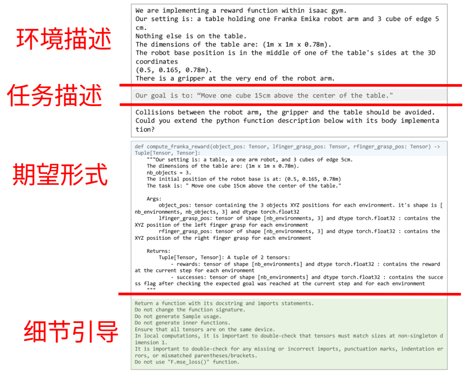
5 In-context Code Generation

5.1 Prompting

We describe how the prompt is designed as a composition of textual instructions, code samples, and Docstring [4]. We use the context of robotic manipulation to illustrate our approach. While experimenting with our approach, we notice that prompt is a crucial step in our approach.

For both goal and reward generation, the prompt structure, as illustrated in Figure 2, consists in four parts, organized as follows: (1) the environment description, (2) the task description, (3) the specifications of the expected function and (4) additional guidelines. First, the environment description starts with a reference to a specific category of task and then provides additional details to be considered to accomplish the task. It completes the context used by the LLM with elements such as the initial environment state, robot specifics, and any other relevant information like a scene description. It also includes technical guidelines such as API or methods to be used. Second, the task description is a text detailing the purpose of the task. It is also used to caption the task in the MTRL approach.

[1] https://yaml.org/
[2] https://www.python.org/
[3] https://developer.nvidia.com/isaac-gym
[4] https://peps.python.org/pep-0257/



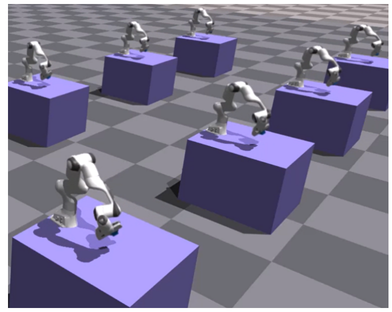


Figure 2: Example of prompt generating a reward function and its application in our simulation. The section defining the purpose of the task is highlighted in grey. The signature of the expected function is in blue. The part in green relates to additional guidelines.

Third, to enforce the coherence of the generated output, we provide the signature of the expected function, either for goal or reward generation, listing all parameters along with the expected returned elements. This signature is completed by a Docstring detailing the role of each parameter. Finally, we can add additional guidelines to drive the code generation. This addition allows for mitigating frequently observed errors due to limitations in current LLMs. For instance, one constraint is to push data into specific devices, e.g. in the CPU or GPU. Such limitations can be automatically captured by the method discussed in section 5.2.

To generate a large set of tasks in a scalable manner, we preserve parts 1, 3 and 4 of the prompt and use the LLM to automatically rephrase a task definition into semantically similar variants using paraphrasing.

5.2 Code Validation and Auto-correction

The generated code isn't guarantee to meet expectations in terms of code validity or outcomes. In such a case, further prompt iterations are performed, emphasizing the elements that need to be modified until the result converges toward expectations. These errors mainly come from under-specified elements in the original prompt or from LLM limitations such as hallucinations [29]. Therefore, we finalize the code generation with an automatic validation step described in Figure 3 which exploits the output of the Python interpreter.

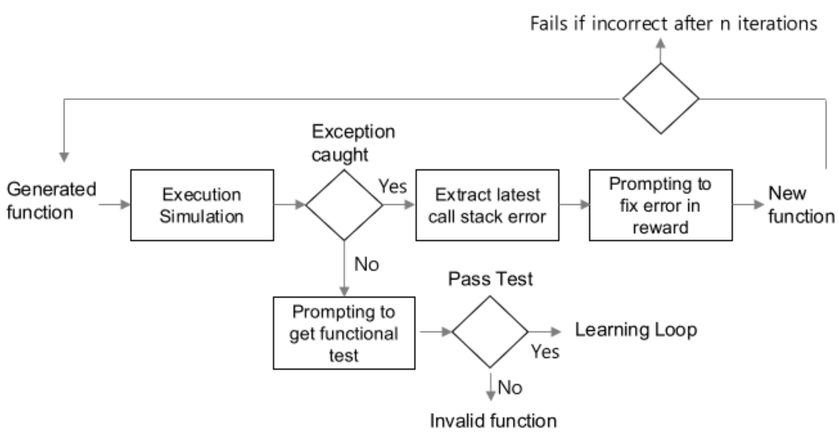


Figure 3: Diagram of the code correction loop. It leverages exceptions raised during execution to request modifications. A functional test is also requested before moving to the learning loop.

We execute the generated code on placeholder input variables and catch the exceptions raised by the Python interpreter when the code fails either to pass the syntax evaluation step or the execution step. We filter the thread of exceptions to only keep the latest stack and use the error message to fill a prompt requesting code modifications. Our prompt, as illustrated in Figure 4, contains 1) a header that requests the LLM to fix the raised exception, 2) the text of the raised exception, and 3) the code of the incorrect function. Several iterations can be done until the code converges toward a version that can be properly executed.

Finally, once a generated function satisfy the code correction step, we use another prompt to request the LLM to generate a functional test to evaluate this first function as detailed in the appendix. This last validation step is intended to further filter out potentially incorrect code prior to running the training loop. This prompt, illustrated in Figure 5, is composed of 1) a header requesting the LLM to generate a functional test, 2) a list of guidelines to condition the test, and 3) the code of the generated function.

```
Could you please fix the error:
'line 38, in compute_franka_reward
RuntimeError: Expected all tensors to be on the same device, but found at least two devices, cuda:0 and cpu!
'
in the following function implementation:
import torch
from torch import Tensor
from typing import Tuple

def compute_franka_reward(object_pos: Tensor, lfinger_grasp_pos: Tensor, rfinger_grasp_pos: Tensor) -> Tuple[Tensor, Tensor]:

...
```

Figure 4: Prompt for the automatic code correction step. The part in blue is the code of the incorrect function generated in the previous steps and in red the output of the code interpreter.

```
Update the following python script with functional tests for the reward function 'compute_franka_reward'.
Rewards tests should only validate cases when they should be positive (>=0) or negative (<=0).
Sucesses should be tested against 1 or 0 values.
Do no add any explanation text.
Return the same script plus what you have inserted.

import torch
from torch import Tensor
from typing import Tuple

def compute_franka_reward(object_pos: Tensor, lfinger_grasp_pos: Tensor, rfinger_grasp_pos: Tensor) -> Tuple[Tensor, Tensor]:

...
```

Figure 5: Prompt requesting the generation of a functional test for the function provided, in blue, as contextual information.

6 Experiments

The purpose of our experiments is three-fold. First, we evaluate LARG^{2}capabilities to generate valid goal positions for GCRL settings. Second, we evaluate the capability of our approach to produce reward functions to train multi-task policies with language-based task description as input. Finally, for both cases mentioned above, we evaluate limitations of our method.

We use a Pick and Place task family defined in the Isaac_gym repository with a Franka Emika Panda robot arm [5] and leverage the gpt-3.5-turbo language model [6] from OpenAI. Several alternative LLMs have been experimented including StarCoder [7] which has also been evaluated. Details of these experiments are further discussed in the appendix. In the GCRL experiment, we evaluate goal generation on a series of 32 tasks including 27 tasks involving a single object, and 5 tasks involving 3 objects. A list of these tasks is provided in the appendix. In the MTRL experiment, we address the generation of reward functions for 9 manipulation tasks detailed in Table 1.

Regarding real-world validation concerns, both our GCRL and MTRL evaluations are conducted in a simulated environment. We focus on various types of rearrangement manipulation tasks and assume the capability for sim2real transfer using techniques like domain randomization. We chose to leverage simulated experiments to enhance the task diversity and assess the scalability of our approach, which we consider to be our primary contribution.

6.1 Automatic goal generation

In the GCRL experiment, we use a neural policy trained beforehand using Proximal Policy Optimization [30]. The policy takes as input the position and velocity of each joint of the robot and the

[5] https://www.franka.de/
[6] https://platform.openai.com/docs/models/gpt-3-5
[7] https://huggingface.co/bigcode/starcoder

respective pose of the objects composing the scene. The policy trigger joint displacement in a $\mathbb{R}^7$ action space. The goal information, generated by LARG^2 is used as additional input to the policy. Figure 6 provides the respective success rates for 32 manipulation tasks.

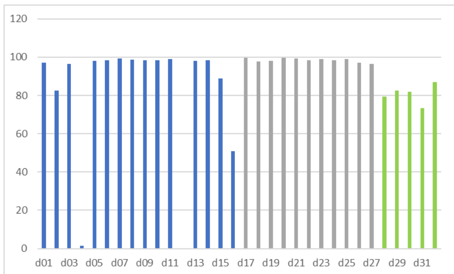


Figure 6: Success rate for GCRL manipulation tasks. Blue reflects 1 object manipulation for absolute pose whereas grey reflects relative object pose. Green relates to 3 object manipulation tasks.

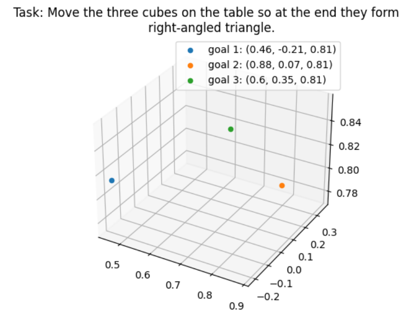


Figure 7: 5 instances of the goal generation function for *Task d30* which forms a right-angled triangle.

Most failures are related to the position of the goal being beyond the reachable workspace of the robotic arm. Nevertheless, we note numerous reasoning capabilities from the large language model to generate a code addressing textual guidelines that require common sense. For instance in Figure 7, goal positions need to comply with relative constraints such as: *Move the three cubes on the table so at the end they form a right-angled triangle*. From this experiment, LARG^2 demonstrated the capability to automatically produce goal positions aligned with textual task descriptions. Besides limitations of current LLM, 87.5% of generated functions pass the validation test and deliver proper input to the GCRL policy to make it actionable by the robot arm.

6.2 Automatic reward generation

For the MTRL experiment, we encode textual descriptions of each task using a pre-trained Google T5-small language model. For each task, we use the [CLS] token embedding computed by the encoder layer of the model which is defined in $\mathbb{R}^{512}$. We concatenate this embedding with the state information of our manipulation environment defined in $\mathbb{R}^7$ and feed it into a fully connected network stack used as policy. This policy is composed of 3 layers using respectively, $\{512, 128, 64\}$ hidden dimensions. Alternately, as suggested by [2], we tested feeding the token embedding into each layer of the stack instead of concatenating it as input, but, we did not observe improvements.

For the generation of reward functions, as described in section 4.2, we first define the task-independent reward component for pick and place manipulation which handles bonuses for lifting the object and penalties for the number of actions to reach the objective. This component, which is therefore common to each task, is preset and not generated. It is added to the reward part generated by LARG^2 for each task to train the MTRL policy. Details about this process are further discussed in the appendix.

Reward functions apply goal poses generated according to the task to compute related scores. Table 1 shows the adherence of these poses with respect to task definitions using functional tests discussed in section 5.2 to assess this adherence. Independently, we evaluate in Figure 9 the success rate of trained policies to achieve our tasks with respect to generated goal positions.

As a summary, LARG^2 demonstrates the capability to produce valid reward functions to successfully train MTRL policies. In addition, LLM can re-phrase or translate textual task definitions. It allows generating in a scalable manner large collections of tasks for training MTRL models with multiple paraphrases associated with each task, increasing the robustness of the approach.

ID	Task	Generated Pose validity
m01	Push the cube to the far right of the table.	✓
m02	Move a cube to the top left corner of the table.	✓
m03	Take the cube and put it close to the robot arm.	✓
m04	Move a cube at 20cm above the center of the table.	-
m05	Move a cube at 15 cm above the table.	✓
m06	Take the cube and put it on the diagonal of the table.	-
m07	Push the cube at 20cm ahead of its current position.	-
m08	Move the cube to the center of the table.	✓
m09	Grab the cube and move it forward to the left.	✓

Table 1: Manipulation tasks evaluated with LARG^{2}in MTRL settings. We assess the alignment of generated goal positions with respect to task definitions.

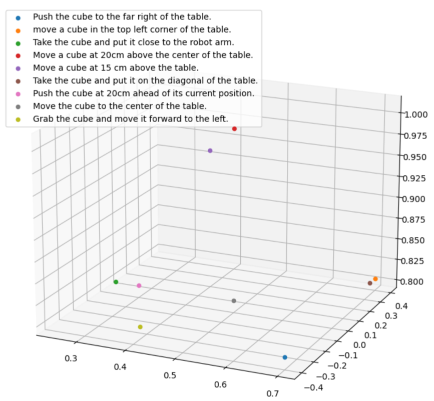


Figure 8: Generated goal position for 9 manipulation tasks.

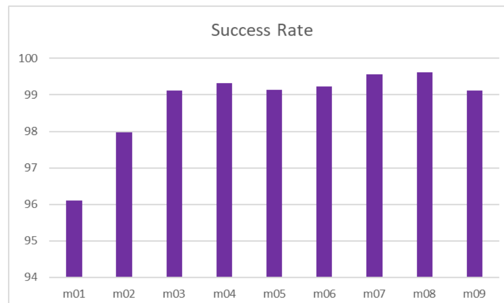


Figure 9: Success rate evaluations of MTRL over automatic reward generation.

7 Limitations and future works

Our experiments highlighted limitations of LLMs capability and reliability in converting user instructions into executable and valid code. It forces users to focus code generation toward specific components rather than toward broader modules. Even though our experiments involved highly structured information such as function signature and docstring, which somehow limits the effect of hallucination, the risk of semantic errors cannot be entirely ruled out. To address these limitations, the auto-correction loop described in our paper seems an effective option to be further investigated. Nevertheless, opportunities offered by current LLMs should encourage the robotic learning community to develop the research in this domain proposed by LARG^{2}which we believe promising. Finally, our evaluations of GCRL and MTRL are conducted in simulations, where it is possible to leverage techniques like domain randomization to demonstrate the potential for sim2real transfer. Simulated experiments allow us to increase task variety and evaluate the scalability of our approach, which is our main contribution and plan to implement real-world scenario in the near future.

8 Conclusion

In this paper, we introduce LARG^2, a novel approach for task-conditioned reinforcement learning from textual descriptions. Our method leverages the in-context learning and code-generation capabilities of large language models to complete or fully generate goal-sampling and reward functions from textual descriptions of tasks. For this purpose, we propose a method for automatic code validation and functional testing. We evaluate the capability of our method to translate 32 text-based task descriptions into actionable objectives for GCRL and to generate rewards functions to train MTRL policies for a 7DoFs robotic arm. Our experiment confirms the benefit of large language models for aligning textual task descriptions with generated goal and reward functions, opening the door, thanks to paraphrasing, to a scalable approach for training Multi-Task Reinforcement Learning models. We believe LARG^{2}opens a novel and scalable direction for training and controlling robots using textual instructions.

References

[1] W. Huang, F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, P. Sermanet, N. Brown, T. Jackson, L. Luu, S. Levine, K. Hausman, and B. Ichter. Inner monologue: Embodied reasoning through planning with language models. 2022. doi: 10.48550/ARXIV.2207.05608. URL `https://arxiv.org/abs/2207.05608`.

[2] Y. Jiang, A. Gupta, Z. Zhang, G. Wang, Y. Dou, Y. Chen, L. Fei-Fei, A. Anandkumar, Y. Zhu, and L. Fan. Vima: General robot manipulation with multimodal prompts. 2022. doi:10.48550/ARXIV.2210.03094. URL `https://arxiv.org/abs/2210.03094`.

[3] D. Shah, B. Osinski, B. Ichter, and S. Levine. Lm-nav: Robotic navigation with large pre-trained models of language, vision, and action. 2022. doi:10.48550/ARXIV.2207.04429. URL `https://arxiv.org/abs/2207.04429`.

[4] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. 2022. doi:10.48550/ARXIV.2201.07207. URL `https://arxiv.org/abs/2201.07207`.

[5] L. Tai, J. Zhang, M. Liu, J. Boedecker, and W. Burgard. A survey of deep network solutions for learning control in robotics: From reinforcement to imitation. *arXiv: Robotics*, 2016.

[6] A. Kumar, J. Hong, A. Singh, and S. Levine. Should i run offline reinforcement learning or behavioral cloning? In *International Conference on Learning Representations*, 2022.

[7] J. Stüber, M. Kopicki, and C. Zito. Feature-based transfer learning for robotic push manipulation. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–5, 2018.

[8] M. Wiese, G. Runge-Borchert, B.-H. Cao, and A. Raatz. Transfer learning for accurate modeling and control of soft actuators. *2021 IEEE 4th International Conference on Soft Robotics (RoboSoft)*, pages 51–57, 2021.

[9] T. Weng, A. Pallankize, Y. Tang, O. Kroemer, and D. Held. Multi-modal transfer learning for grasping transparent and specular objects. *IEEE Robotics and Automation Letters*, 5:3796–3803, 2020.

[10] M. Kelly, C. Sidrane, K. Driggs-Campbell, and M. J. Kochenderfer. Hg-dagger: Interactive imitation learning with human experts. *2019 International Conference on Robotics and Automation (ICRA)*, pages 8077–8083, 2018.

[11] E. Chisari, T. Welschehold, J. Boedecker, W. Burgard, and A. Valada. Correct me if i am wrong: Interactive learning for robotic manipulation. *IEEE Robotics and Automation Letters*, 7:3695–3702, 2021.

[12] T. A. K. Faulkner, E. S. Short, and A. L. Thomaz. Interactive reinforcement learning with inaccurate feedback. *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7498–7504, 2020.

[13] R. S. Sutton and A. G. Barto. Reinforcement learning: An introduction. *IEEE Transactions on Neural Networks*, 16:285–286, 2005.

[14] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.

[15] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. M. O. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2016.

[16] S. Gu, E. Holly, T. Lillicrap, and S. Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3389–3396, 2017. doi:10.1109/ICRA.2017.7989385.

[17] M. Plappert, M. Andrychowicz, A. Ray, B. McGrew, B. Baker, G. Powell, J. Schneider, J. Tobin, M. Chociej, P. Welinder, V. Kumar, and W. Zaremba. Multi-goal reinforcement learning: Challenging robotics environments and request for research. *ArXiv*, abs/1802.09464, 2018.

[18] A. Nair, V. H. Pong, M. Dalal, S. Bahl, S. Lin, and S. Levine. Visual reinforcement learning with imagined goals. In *NeurIPS*, 2018.

[19] O. OpenAI, M. Plappert, R. Sampedro, T. Xu, I. Akkaya, V. Kosaraju, P. Welinder, R. D'Sa, A. Petron, H. P. de Oliveira Pinto, A. Paino, H. Noh, L. Weng, Q. Yuan, C. Chu, and W. Zaremba. Asymmetric self-play for automatic goal discovery in robotic manipulation. *ArXiv*, abs/2101.04882, 2021.

[20] M. Dorigo and M. Colombetti. Robot shaping: Developing autonomous agents through learning. *Artificial intelligence*, 71(2):321–370, 1994.

[21] J. Randløv and P. Alstrøm. Learning to drive a bicycle using reinforcement learning and shaping. In *Proceedings of the 15th International Conference on Machine Learning (ICML'98)*, pages 463–471, 1998.

[22] A. Brohan, N. Brown, J. Carbajal, Y. Chebotar, J. Dabis, C. Finn, K. Gopalakrishnan, K. Hausman, A. Herzog, J. Hsu, J. Ibarz, B. Ichter, A. Irpan, T. Jackson, S. Jesmonth, N. J. Joshi, R. C. Julian, D. Kalashnikov, Y. Kuang, I. Leal, K.-H. Lee, S. Levine, Y. Lu, U. Malla, D. Manjunath, I. Mordatch, O. Nachum, C. Parada, J. Peralta, E. Perez, K. Pertsch, J. Quiambao, K. Rao, M. S. Ryoo, G. Salazar, P. R. Sanketi, K. Sayed, J. Singh, S. A. Sontakke, A. Stone, C. Tan, H. Tran, V. Vanhoucke, S. Vega, Q. H. Vuong, F. Xia, T. Xiao, P. Xu, S. Xu, T. Yu, and B. Zitkovich. Rt-1: Robotics transformer for real-world control at scale. *ArXiv*, abs/2212.06817, 2022.

[23] 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. 2022. doi:10.48550/ARXIV.2209.07753. URL https://arxiv.org/abs/2209.07753.

[24] M. Kwon, S. M. Xie, K. Bullard, and D. Sadigh. Reward design with language models. In *International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=10uNUgI5Kl.

[25] C. Colas, A. Akakzia, P.-Y. Oudeyer, M. Chetouani, and O. Sigaud. Language-conditioned goal generation: a new approach to language grounding for rl. *ArXiv*, abs/2006.07043, 2020.

[26] C. Colas, A. Akakzia, P. Oudeyer, M. Chetouani, and O. Sigaud. Language-conditioned goal generation: a new approach to language grounding for RL. *CoRR*, abs/2006.07043, 2020. URL https://arxiv.org/abs/2006.07043.

[27] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2020.

[28] C. Raffel, N. M. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *ArXiv*, abs/1910.10683, 2020.

[29] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. Bang, A. Madotto, and P. Fung. Survey of hallucination in natural language generation. *CoRR*, abs/2202.03629, 2022. URL https://arxiv.org/abs/2202.03629.

[30] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *ArXiv*, abs/1707.06347, 2017.

[31] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe. Training language models to follow instructions with human feedback. *ArXiv*, abs/2203.02155, 2022.

[32] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries. Starcoder: may the source be with you!, 2023.

A Appendix

In this section, we go through additional details about LARG^2 and discuss experiments performed to evaluate its performance both for GCRL and MTRL settings. We also provide some examples of prompts used in our experiments.

A.1 Method

A.1.1 Prerequisites

LARG^2 aims at providing a scalable method to align language-based description of tasks with goal and reward functions to address GCRL or MTRL. It relies on code generation capabilities offered by recent Large language models. These LLM already capture prior background knowledge and common sens. In terms of coding capabilities, they leverage existing code available in repositories like GihHub [8]. Fully capable LLM could in theory generate proper code from pure textual descriptions. However, our experiments demonstrate that current LLM still benefit from additional guidelines provided as context. Such guidelines relate to scene understanding and function signature. One source of information for guidelines is environment descriptions in code repositories. Additionally, scene understanding can be provided by exteroception components that translate images into object captions and geometric coordinates.

As a first iteration we assume the existence of a portfolio of categories of manipulation tasks defined in repositories like Isaac Gym [9] with descriptions of environments formalized using languages like YAML [10] or Python [11]. Accordingly, we assume that such environments also provide signatures of expected functions commented with a formalism like Docstring [12].

In such a case, LARG^2 aligns a text based task description with the appropriate category of tasks and leverage environment descriptions to build an ad-hoc prompt to be used with LLMs. Therefore, code generated by LARG^{2}can be seamlessly integrated into these repositories to execute the desired settings.

Textual descriptions of tasks allows to overload generic definitions of tasks available in code repositories. Scalability can therefore be achieved thanks to paraphrasing. Indeed, LLMs can generate task definition variants on a basis of textual seeds to produce large training datasets.

A.1.2 Generation of goal poses for GCRL

A first application of LARG^{2}generates goals to be used as parameters of a predefined goal-conditioned reward function.

As an example, in tabletop robotic manipulation scenarios, a pick and place task consists in rearranging a set of objects composing a scene. In such a case, the goal is the set of target poses for all objects and the reward function basically compute the Euclidian distance between a current object pose and the target pose. LARG^2 generates functions producing a set of eligible goal positions from textual task descriptions.

The prompt design used to generate the goal function is composed of the following elements: 1) the environment description, 2) the task description, 3) the specifications of the expected function and 4) optional guidelines.

Figure 10 illustrates our prompt design and figure 11 shows the generated code.

[8] https://github.com/
[9] https://github.com/NVIDIA-Omniverse/IsaacGymEnvs
[10] https://yaml.org/
[11] https://www.python.org/
[12] https://peps.python.org/pep-0257/

```
We are implementing a table top rearrangement task within isaac gym such as Fra
nka_Move.
We need to set goal positions.

Could you complet the code of the python function "generate_goal_poses" below
with its body implementation
according to settings defined in the docstring below
for the goal: "Move the three cubes on the table so at the end they form a righ
t-angled triangle."

Code to be completed:

import torch
from torch import Tensor
from typing import Tuple

def generate_goal_poses() -> Tuple[Tensor, Tensor, Tensor]:
    """This function returns an eligible goal pose satisfying the following tab
le top rearrangement task: Move the three cubes on the table so at the end they
form a right-angled triangle."

    The setup is 1 table of dimension 1 x 1 x 0.79 on which stands 3 cubes of e
dge 5cm.
    Each cube has its own origin in its center.
    The global origin (0,0,0) is located on the floor below the middle of one o
f the table's sides.
    X coordinates range: (0.3, +0.90) = (Bottom, Top)
    Y coordinates range: (-0.4, +0.4) = (Right, Left)
    The minimum Z value for objects located on the surface of the table is 0.79
+ 0.05/2 = 0.815

    Returns:
     - Tuple: a tuple of 3 elements. Each element is a tensor . Each tensor of
shape (3) represents the goal position of one cube on the table
    """
```

Figure 10: GCRL: Prompt requesting the generation of the goal function. The function signature is highlighted in blue and the the text-based goal description in grey.

A.1.3 Generation of reward function for MTRL

The second utilization of LARG^2 generates the executable source code of a reward function according to a task description .

For MTRL the policy takes as input the textual description of the task in addition to the state. In such as case, goals are removed from the environment. However this information is required by the reward function to compute a gain. Therefore this information is also generated by LARG^2 according to the provided task description.

For the reward function itself, we separate components which are task independent from those which are task dependent. In robotic manipulation, task agnostic components address bonuses for lifting the objects or penalties for the number of actions to reach the goal. Due to known limitations in current LLM, we focus LARG^2 on generating the part of the reward that depends on the specific guidelines and constraints defined in textual definitions.

The prompt structure used for generating the reward function is similar to the one used for goal generation. It is composed of 1) the environment description, 2) the task description, 3) the specifications of the expected function and 4) optional guidelines. However, in this case the function specification contains the signature of the expected reward function.

```
import torch
from torch import Tensor
from typing import Tuple

def generate_goal_poses() -> Tuple[Tensor, Tensor, Tensor]:
    """This function returns an eligible goal pose satisfying the following tab
le top rearrangement task: Move the three cubes on the table so at the end they
form a right-angled triangle."

    The setup is 1 table of dimension 1 x 1 x 0.79 on which stands 3 cubes of e
dge 5cm.
    Each cube has its own origin in its center.
    The global origin (0,0,0) is located on the floor below the middle of one o
f the table's sides.
    X coordinates range: (0.3, +0.90) = (Bottom, Top)
    Y coordinates range: (-0.4, +0.4) = (Right, Left)
    The minimum Z value for objects located on the surface of the table is 0.79
+ 0.05/2 = 0.815

    Returns:
     - Tuple: a tuple of 3 elements. Each element is a tensor . Each tensor of
shape (3) represents the goal position of one cube on the table
    """

    # Define the position of the cubes to form a right-angled triangle
    cube1_pos = torch.tensor([0.6, -0.2, 0.815])
    cube2_pos = torch.tensor([0.9, -0.2, 0.815])
    cube3_pos = torch.tensor([0.9, 0.1, 0.815])

    return cube1_pos, cube2_pos, cube3_pos
```

Figure 11: GCRL: Generated code for the goal pose function.

The following figures illustrate prompts and results obtained when requesting the generation of ad-hoc code for manipulating one cube to bring it closer to the robotic arm. Figure 12 details the global reward function that combines both elements from the task independent, which is illustrated by figure 13, and task dependent part. In this case, LARG^{2}focus on generating the dependent part using a prompt illustrated by Figure 14 to produce the code depicted in Figure 15.

```
def compute_franka_reward(
    reset_buf: Tensor, progress_buf: Tensor, successes: Tensor, actions: Tensor,
    lfinger_grasp_pos: Tensor, rfinger_grasp_pos: Tensor, object_pos: Tensor, goal_pos: Tensor,
    object_z_init: float, object_dist_reward_scale: float, lift_bonus_reward_scale: float,
    goal_dist_reward_scale: float, goal_bonus_reward_scale: float, action_penalty_scale: float,
    contact_forces: Tensor, arm_inds: Tensor, max_episode_length: int
) -> Tuple[Tensor, Tensor, Tensor]:


    # og_d: The distance between the object pose and the goal pose
    og_d = compute_object_to_goal_distance( object_pos, goal_pos)

    # object_above: Boolean, true if the object is above the table, false otherwise.
    object_above = is_object_above_initial_pose (object_pos, object_z_init)

    # Part of the reward that is task invariant
    static_rewards, reset_buf, lfo_dist_reward = compute_franka_reward_static( reset_buf, progress_
buf, successes, actions, lfinger_grasp_pos, rfinger_grasp_pos, object_pos, object_z_init, object_di
st_reward_scale, lift_bonus_reward_scale, goal_dist_reward_scale, goal_bonus_reward_scale, action_p
enalty_scale, contact_forces, arm_inds, max_episode_length)


    # Part of the reward that depends on the specifications provided in the task definition

    # og_d: The distance between the object pose and the goal pose
    og_d = compute_object_to_goal_distance( object_pos, goal_pos )
    # object_above: Boolean, true if the object is above the table, false otherwise.
    object_above = is_object_above_initial_pose (object_pos, object_z_init)

    # Compute generated part of the reward
    generated_rewards = compute_franka_reward_generated( lfo_dist_reward, object_above, og_d, goal_
dist_reward_scale, goal_bonus_reward_scale)


    # Total reward
    rewards = static_rewards \
            + generated_rewards


    # Goal reached
    successes = compute_successes(og_d, successes)


    return rewards, successes
```

Figure 12: MTRL: Code of a global reward function combining a task independent and task dependent component (highlighted section in yellow).

```
def compute_franka_reward_static(
    reset_buf: Tensor, progress_buf: Tensor, successes: Tensor, actions: Tensor,
    lfinger_grasp_pos: Tensor, rfinger_grasp_pos: Tensor, object_pos: Tensor, goal_pos: Tensor,
    object_z_init: float, object_dist_reward_scale: float, lift_bonus_reward_scale: float,
    goal_dist_reward_scale: float, goal_bonus_reward_scale: float, action_penalty_scale: float,
    contact_forces: Tensor, arm_inds: Tensor, max_episode_length: int
) -> Tuple[Tensor, Tensor, float]:

    # Left finger to object distance
    lfo_d = torch.norm(object_pos - lfinger_grasp_pos, p=2, dim=-1)
    lfo_d = torch.clamp(lfo_d, min=0.02)
    lfo_dist_reward = 1.0 / (0.04 + lfo_d)

    # Right finger to object distance
    rfo_d = torch.norm(object_pos - rfinger_grasp_pos, p=2, dim=-1)
    rfo_d = torch.clamp(rfo_d, min=0.02)
    rfo_dist_reward = 1.0 / (0.04 + rfo_d)

    # Object above table
    object_above = (object_pos[:, 2] - object_z_init) > 0.015

    # Above the table bonus
    lift_bonus_reward = torch.zeros_like(lfo_dist_reward)
    lift_bonus_reward = torch.where(object_above, lift_bonus_reward + 0.5, lift_bonus_reward)

    # Regularization on the actions
    action_penalty = torch.sum(actions ** 2, dim=-1)

    # Total reward
    rewards = object_dist_reward_scale * lfo_dist_reward + object_dist_reward_scale * rfo_dist_reward
+ lift_bonus_reward_scale * lift_bonus_reward - action_penalty_scale * action_penalty

    # Object below table height
    object_below = (object_z_init - object_pos[:, 2]) > 0.04

    reset_buf = torch.where(object_below, torch.ones_like(reset_buf), reset_buf)

    # Arm collision
    arm_collisions = torch.any(torch.norm(contact_forces[:, arm_inds, :], dim=2) > 1.0, dim=1)

    reset_buf = torch.where(arm_collisions, torch.ones_like(reset_buf), reset_buf)

    # Max episode length exceeded
    reset_buf = torch.where(progress_buf >= max_episode_length - 1, torch.ones_like(reset_buf), reset_
buf)

return rewards, reset_buf, lfo_dist_reward
```

Figure 13: MTRL: Code of the task independent reward component.

```
Context: We are developing in python a reward function for a Franka_move task in Isaac_gym.
This function returns a tuple composed of the reward for achieving the objective.
The objective is the following table top rearrangement task: "Take the cube and put it close to th
e robot arm."
This reward is composed of the object to goal reward and the bonus if object is near the goal

Complete this function, setting reward function to optimize the distance between the object and it
s goal pose .

import torch
from torch import Tensor
from typing import Tuple

from gpt.utils import *
from envs.utils.torch_jit_utils import *

@torch.jit.script
def compute_franka_reward_generated(
    lfo_dist_reward: float,
    object_above: bool,
    og_d: float,
    goal_dist_reward_scale: float,
    goal_bonus_reward_scale: float
) -> Tuple[Tensor]:
    """
    Compute the reward signal for a Franka Move task in Isaac Gym.
    Args:
        lfo_dist_reward: The reward for moving the left finger closer to the object.
        object_above: Boolean, true if the object is above the table, false otherwise.
        og_d: The distance between the object pose and the goal pose
        goal_dist_reward_scale: Scaling factor for the goal distance reward.
        goal_bonus_reward_scale: Scaling factor for the goal bonus reward.
    Returns:
        A tuple containing:
            - rewards: The reward signal for the current state.
    """

     # Add the code for computing the part of the reward related to optimizing the distance betwee
n the object and its goal pose


    return rewards
```

Figure 14: MTRL: Prompt requesting the generation of a task dependent part of a reward function.

```
import torch
from torch import Tensor
from typing import Tuple


def compute_franka_reward_generated(
    lfo_dist_reward: float, object_pos: Tensor, og_d: Tensor,
    object_z_init: float,
    goal_dist_reward_scale: float, goal_bonus_reward_scale: float
) -> Tuple[Tensor, float]:

    # Object above table
    object_above = (object_pos[:, 2] - object_z_init) > 0.015

    # Object to goal reward
    og_dist_reward = torch.zeros_like(lfo_dist_reward)
    og_dist_reward = torch.where(object_above, 1.0 / (0.04 + og_d), og_dist_reward
)

    # Bonus if object is near the goal
    og_bonus_reward = torch.zeros_like(og_dist_reward)
    og_bonus_reward = torch.where(og_d <= 0.04, og_bonus_reward + 0.5, og_bonus_re
ward)


    # Total reward
    rewards = goal_dist_reward_scale * og_dist_reward  + goal_bonus_reward_scale *
og_bonus_reward



    return rewards
```

Figure 15: MTRL: Code generated by LARG^2 for the task dependent part of a reward function.

A.1.4 Code validation

Optionally, once function code is generated an additional validation step can occur. LLMs can be used to generate a functional test prior to start the training process or to run the task. This prompt, illustrated in Figure 16, is composed of 1) a header requesting the LLM to generate a functional test, 2) a list of guidelines to condition the test, and 3) the code of the generated function. An example of test is proposed in Figure 17 .

```
We are implementing a reward function of a custom task for a Franka Move environment within Isaac gym.
Our setting is: a table holding one Franka Emika robot arm and 3 cubes of edge 5cm.
The dimensions of the table are: (1m x 1m x 0.78m).
The robot base position is in the middle of one of the table's sides at the 3D coordinates
(0.5, 0.165, 0.78m).
There is a gripper at the very end of the robot arm.

Our custom task is: "Move a cube to the top right corner of the table."

Update the following python script with a functional test for the reward function "compute_franka_reward"
Reward tests should only validate cases when they should be positive (>=0) or negative (<=0).
Success should be tested against 1 or 0 values.

def compute_franka_reward(object_pos: Tensor, lfinger_grasp_pos: Tensor, rfinger_grasp_pos: Tensor) -> Tu
ple[Tensor, Tensor]:
    """Our setting is: a table, a one arm robot, and 3 cubes of edge 5cm.
                                        …
```

Figure 16: Prompt requesting the generation of functional test for a reward function.

```
# Functional tests for the compute_franka_reward function
def test_compute_franka_reward():
    # Test case 1: Object positions at target position, gripper positions at target position
    object_pos = torch.tensor([[[0.95, 0.95, 0.15], [0.9, 0.9, 0.15], [0.8, 0.8, 0.15]]])
    lfinger_grasp_pos = torch.tensor([[0.95, 0.95, 0.15]])
    rfinger_grasp_pos = torch.tensor([[0.95, 0.95, 0.15]])
    expected_rewards = torch.tensor([[0.0, 0.0, 0.0]])
    expected_successes = torch.tensor([[1.0, 1.0, 1.0]])
    rewards, successes = compute_franka_reward(object_pos, lfinger_grasp_pos, rfinger_grasp_pos)
    assert torch.allclose(rewards, expected_rewards)
    assert torch.allclose(successes, expected_successes)

    # Test case 2: Object positions far from target position, gripper positions far from target position
    object_pos = torch.tensor([[[0.5, 0.5, 0.15], [0.6, 0.6, 0.15], [0.7, 0.7, 0.15]]])
    lfinger_grasp_pos = torch.tensor([[0.5, 0.5, 0.15]])
    rfinger_grasp_pos = torch.tensor([[0.5, 0.5, 0.15]])
    expected_rewards = torch.tensor([[-0.4871, -0.5831, -0.7031]])
    expected_successes = torch.tensor([[0.0, 0.0, 0.0]])
    rewards, successes = compute_franka_reward(object_pos, lfinger_grasp_pos, rfinger_grasp_pos)
    assert torch.allclose(rewards, expected_rewards)
    assert torch.allclose(successes, expected_successes)

    # Test case 3: Object positions close to target position, gripper positions close to target position
    object_pos = torch.tensor([[[0.95, 0.95, 0.25], [0.9, 0.9, 0.25], [0.8, 0.8, 0.25]]])
    lfinger_grasp_pos = torch.tensor([[0.95, 0.95, 0.25]])
    rfinger_grasp_pos = torch.tensor([[0.95, 0.95, 0.25]])
    expected_rewards = torch.tensor([[0.15, 0.15, 0.15]])
    expected_successes = torch.tensor([[1.0, 1.0, 1.0]])
    rewards, successes = compute_franka_reward(object_pos, lfinger_grasp_pos, rfinger_grasp_pos)
    assert torch.allclose(rewards, expected_rewards)
    assert torch.allclose(successes, expected_successes)

    print("All tests passed!")
```

Figure 17: Generated functional test.

A.2 Experiments

We evaluate LARG^2 on a series of tabletop object manipulation tasks for both GCRL and MTRL settings.

Experiments leverage the Franka_Move environment available on the Isaac_Gym repository [13]. This environment describes a table, a Franka Emika Panda robot arm [14] which is an open kinematic chain composed with 7DoF, and n cubes on the table. The dimensions of the table are as follows: 1m x 1m x 0.78m. The robot arm is placed on the table at (0.5, 0.165, 0.78). There is a griper with two fingers attached at the end of the arm. Cubes with a 5cm edge are located on the surface of the table. The global origin (0,0,0) is located on the floor below the table. Each environment description is written using the Python language.

A.2.1 Large Language Models

In our experiments, several LLMs are evaluated: text-davinci-003 [15], code-davinci-002 [16] and gpt-3.5-turbo [17] from OpenAI which are evolutions from GPT3 optimized with Reinforcement Learning from Human Feedback [31] .

StarCoder from HuggingFace [32] is also evaluated to generate goal functions over the list task defined for the GCRL experiment. Over 32 tasks, only 12.5% of generated functions are executable even after the auto-correction loop and only 3.1% deliver goal positions which are valid. Frequent issues are related to incorrect variable initialization, missing code and a lack of compliance with provided guidelines such as illustrated in Figures 18 and 19.

```
import torch
from torch import Tensor
from typing import Tuple
def generate_goal_pose() -> Tensor:
    """This function returns an elligible goal pose satisfying the following table top rearrangement
task: "move a cube in the top right corner of the table."

    The setup is 1 table of dimension 0.81 x 1 x 0.795 on which stands 1 cube of edge 5cm and ha
ving its origin in its center.
    The global origin (0,0,0) is located on the floor below the middle of one of the table's sides.
    X coordinates range: (0., +0.70) = (Bottom, Top)
    Y coordinates range: (-0.4, +0.4) = (Right, Left)
    The minimum Z value for objects located on the surface of the table is 0.795 + 0.05/2 = 0.82

    Returns:
     - Tensor: a tensor of shape (3) representing the goal position
    """
    x = 0.65
    y = -0.35
    z = 0.82
    goal_pose = torch.tensor([x, y, z])
    return goal_pose
```

Figure 18: Code generated by gpt-3.5-turbo for the task: Move a cube in the top right corner of the table.

As a summary, best results, in terms of code generation, are achieved with gpt-3.5-turbo which is the model used in results reported bellow.

[13] https://developer.nvidia.com/isaac-gym
[14] https://www.franka.de/
[15] https://platform.openai.com/docs/models/gpt-3
[16] https://platform.openai.com/docs/models/codex
[17] https://platform.openai.com/docs/models/gpt-3-5

```
import random
import numpy as np
import torch

def generate_goal_pose():
    # Define limits based on your environment dimensions
    max_x = &lt;max-value-for-x&gt;
    min_x = &lt;min-value-for-x&gt;
    max_y = &lt;max-value-for-y&gt;
    min_y = &lt;min-value-for-y&gt;
    max_z = &lt;max-value-for-z&gt;
    min_z = &lt;min-value-for-z&gt;
    while True:
        # Generate random values within defined ranges
        rand_vals = [random.uniform(min_x, max_x),
                     random.uniform(min_y, max_y),
                     random.uniform(min_z, max_z)]
        if check_valid(*rand_vals):
            break
    return torch.tensor([*rand_vals])

def check_valid(x, y, z):
    # Check validity conditions here, e.g.:
    # Does not collide with other objects?
    # Is within workspace boundaries?
    pass
if __name__ == "__main__":
    print("Example usage:")
    pos = generate_goal_pose().numpy()
    print(f"Generated Position: {pos}")
```

Figure 19: Code generated by StarCoder for the task: Move a cube in the top right corner of the table. In this example the generated code cannot be applied.

A.2.2 Automatic goal generation for the GCRL experiment

In the GCRL experiment, the policy takes as input the position and velocity of each joint of the robot and the respective pose of the objects composing the scene. The policy trigger joint displacement in a $\mathbb{R}^7$ action space. In addition to the position of the object composing the scene, the policy takes as input the goal positions. These positions are provided by goal functions generated by LARG^2. The policy is trained beforehand using Proximal Policy Optimization [30] with default Franka_Move parameters as defined in table 2.

training parameters	values
number of environments	2048
episode length	500
object distance reward scale	0.08
lift bonus reward scale	4.0
goal distance reward scale	1.28
goal bonus reward scale	4.0
action penalty scale	0.01
collision penalty scale	1.28
actor hidden dimension	[256, 128, 64]
critic hidden dimension	[256, 128, 64]

Table 2: List of parameters used in the Franka_Move PPO training loop.

We evaluate our approach on a series of 32 tasks including 27 tasks involving a single object, and 5 tasks involving 3 objects. Tasks $d17$ to $d27$ correspond to objectives which are defined relative to the object's initial position. In this case, the signature of the goal function naturally takes as input the initial position of the cubes composing the scene. Figure 20 illustrates the prompting workflow which translates a task description into the generation of a goal function. It involves an auto-correction step and the production of a functional test afterward.

Figure 21 illustrates results produced by 10 run of 3 different goal functions generated out of 3 different manipulation tasks In all cases, the resulting poses are well aligned with task requirements while exploring the range of valid positions allowed by a non deterministic task definition.

Table 3 provides the list of all tasks used in our experiment and report the compliance of generated goals with task descriptions.

ID	Task	Generated Pose validity
d01	Move a cube to the top right corner of the table.	✓
d02	Move a cube to the top left corner of the table.	✓
d03	Move a cube to the bottom right corner of the table.	✓
d04	Move a cube to the bottom left corner of the table.	✓
d05	Lift the cube 15cm above the table.	✓
d06	Rotate a cube upside-down.	✓
d07	Take to cube and move it to the left side of the table.	-
d08	Take to cube and move it to the right edge of the table.	✓
d09	Take to cube and raise it at 20 cm to the far side of the table.	✓
d10	Take the cube and move it closer to the robotic arm.	✓
d11	Pick up the cube and move it away from the robotic arm.	✓
d12	Take the cube and move it very close to the robotic arm.	-
d13	Push the cube off the limits of the table.	✓
d14	Bring the cube closer to the robot arm.	✓
d15	Move the cube to one corner of the table.	✓
d16	Place the cube anywhere on the diagonal of the table running from the top right corner to the bottom left corner.	✓
d17	Lift the cube 15cm above the table and 10 cm to the right.	✓
d18	Lift the cube 20cm above the table and 15 cm ahead.	✓
d19	Lift the cube 20cm above the table and 15 cm backward.	✓
d20	Push a cube 10cm to the right and 10cm ahead.	✓
d21	Push a cube 10cm to the right and 10cm backward.	✓
d22	Push a cube 10cm to the left and 10cm ahead.	✓
d23	Push a cube 10cm to the left and 10cm backward	✓
d24	Grab a cube and move it a bit to the left.	✓
d25	Grab a cube and lift it a bit and move it a bit ahead.	✓
d26	Move the cube at 20cm to the left of its initial position.	✓
d27	Move the cube 20cm above its current position.	✓
d28	Move one cube to the left side of the table, another one to the right side of the table, and put the last cube at the center of the table.	✓
d29	Move the three cubes so they are 10 cm close to one another.	✓
d30	Move the three cubes on the table so that at the end they form a right-angled triangle.	✓
d31	Move the three cubes on the table so that at the end they form an isosceles triangle.	✓
d32	Reposition the three cubes on the table such that they create a square, with the table's center serving as one of the square's corners.	✓

Table 3: List of the 32 manipulation tasks evaluated with LARG^2. Tasks $d17$ to $d27$ involve objectives relative to the object's initial position. Tasks $d28$ to $d32$ address 3 object manipulation problems and therefore 3 goals. Localisation compliance with task definition is reported.

Figure 22 shows success rates for our 32 manipulation tasks. Looking at unsuccessful experiments, we make several observations. A common source of error relates to a lack of contextual information and constraints in the definition of the task. Two options can be mentioned as future directions to address such a case: either to increase the amount of constraints in the prompt, or to use a LLM with more capabilities.

Interestingly, this experiment also allows to highlight LLM reasoning capabilities as illustrated in Figure 23 where the task request to lift a cube at 15cm above the table. In this case the generated goal function add the table height to the specified 15cm to end up with the correct position.

As a summary, LARG^2 allows to generate code for goal prediction according to textual task descriptions. In some cases the generated code do not properly fit with user specifications but our experiment demonstrate that a feedback loop with additional guidelines can fix the problem.

We are implementing a table top rearrangement task within Isaac gym.
We need to set goal positions.

Could you complete the code of the python function "generate_goal_pose" below with its body implementation according to settings defined in the docstring below for the goal: "Move the three cubes on the table so at the end they form a right-angled triangle."

Code to be completed:

```
import torch
from torch import Tensor
from typing import Tuple

def generate_goal_pose() -> Tuple[Tensor]:
    """This function returns an eligible goal pose satisfying the following tabletop rearrangement task: "nest a cube in the top right corner of the table"

    The setup is 1 table of dimension 0.81 x 1 x 0.795 on which stands 1 cube of edge 5cm and having its origin in its center.
    The global origin (0,0,0) is located on the floor below the middle of one of the table's sides.
    X coordinates range: (0., 0.70)
    Y coordinates range: (-0.4, 0.4)
    The minimum Z value for objects located on the surface of the table is 0.795 + 0.05/2 = 0.82

    Returns :
     - Tensor: a tensor of shape (3) representing the goal position
    """
```

It is important to leave the function signature unchanged and keep the docstring as is.
Do not generate sample usage nor inner functions.
Double-check for any unused variables or functions, missing or incorrect imports, punctuation marks, indentation errors, or mismatched parentheses/brackets.

Could you please fix the error:
'AssertionError: <function generate_goal_pose at 0x7f4bec4bf550> should return one element of shape 3: (tensor([0.5821, 0.1927, 0.8200]),)'
in the following function implementation:

```
import torch
from torch import Tensor
from typing import Tuple
import random

def generate_goal_pose() -> Tuple[Tensor]:
    """
    This function returns an eligible goal pose satisfying the following tabletop rearrangement task: "nest a cube in the top right corner of the table."

    The setup is 1 table of dimension 0.81 x 1 x 0.795 on which stands 1 cube of edge 5cm and having its origin in its center.
    The global origin (0, 0, 0) is located on the floor below the middle of one of the table's sides.
    X coordinates range: (0., 0.70)
    Y coordinates range: (-0.4, 0.4)
    The minimum Z value for objects located on the surface of the table is 0.795 + 0.05/2 = 0.82

    Returns:
        - Tensor: a tensor of shape (3) representing the goal position
    """
    x = random.uniform(0, 0.7)
    y = random.uniform(0, 0.4)
    z = 0.82
    return (torch.tensor([x, y, z]),)
```


Update the following python script with functional tests for the goal position

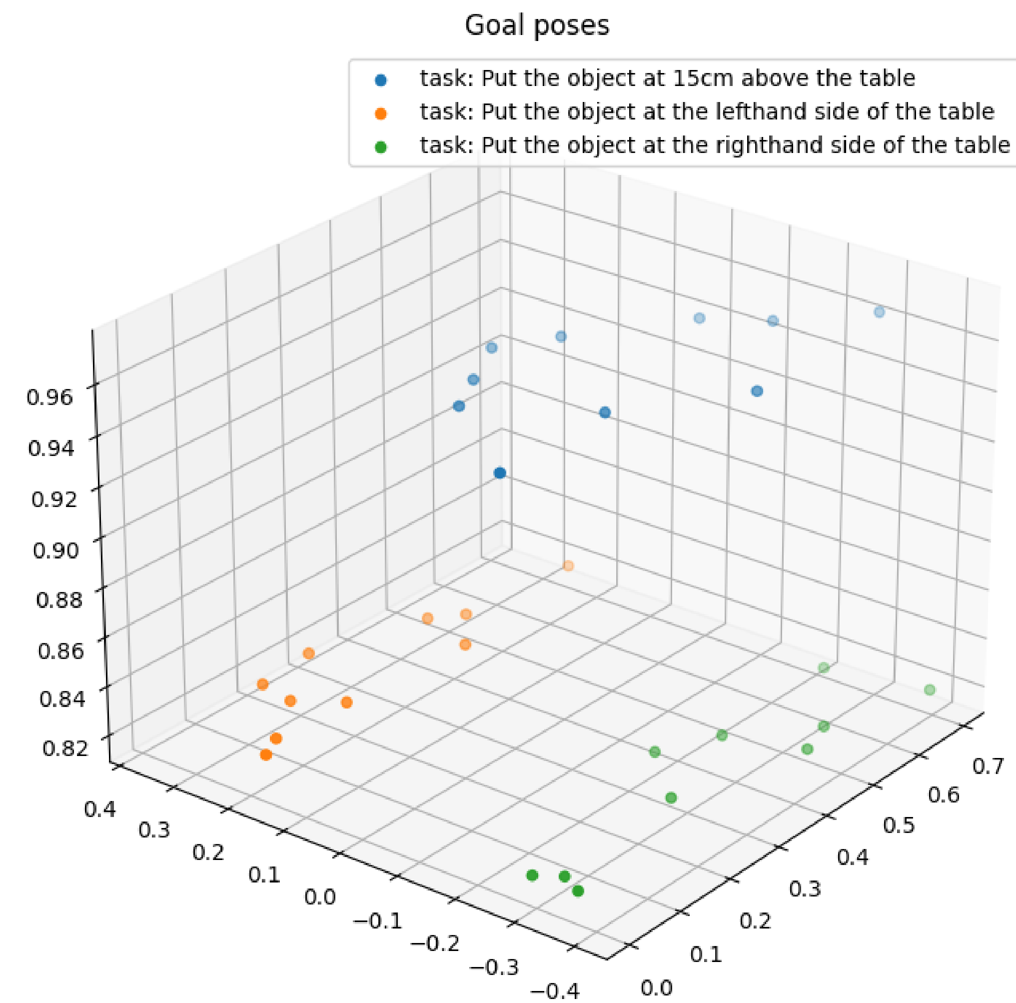


Figure 21: Example of goal positions generated by our method for 3 different tasks requesting targets to be located on the right, left, and above the table.

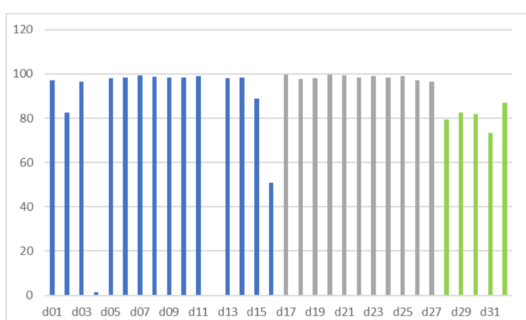


Figure 22: Success rate for GCRL manipulation tasks. Blue reflects 1 object manipulation for absolute pose whereas grey reflects relative object pose. Green relates to 3 object manipulation tasks.

```python
import torch
from torch import Tensor
from typing import Tuple

def generate_goal_pose() -> Tensor:
    # Define the minimum and maximum allowed positions
    min_x, max_x = 0.0, 0.7
    min_y, max_y = -0.4, 0.4
    min_z = 0.82
    target_z = min_z + 0.15  # Add 15 cm to the minimum Z value

    # Generate random X and Y coordinates within the allowed ranges
    x = torch.rand(1) * (max_x - min_x) + min_x
    y = torch.rand(1) * (max_y - min_y) + min_y

    # Create a tensor with the goal position
    goal_position = torch.tensor([x, y, target_z])

    return goal_position
```

Figure 23: Arithmetic capabilities of the LLM for Task *d05*. The comment highlighted in yellow so as the related code is generated by the LLM.

A.2.3 Automatic reward generation for the MTRL experiment

Our second experiment evaluates LARG^2 capability to address MTRL settings. For task encoding, we use the Google T5-small language model. We use the [CLS] token embedding computed by the encoder stack of the model which is defined in $\mathbb{R}^{512}$. We concatenate this embedding with the state information of our manipulation environment defined in $\mathbb{R}^7$ and feed it into a fully connected network stack used as policy. This policy is composed of three layers using respectively, $\{512, 128, 64\}$ hidden dimensions.

In our experiment we train an MTRL settings using Proximal Policy Optimization with default Franka Move parameters using reward functions generated by LARG^{2}over 9 tasks listed in Table 4. These tasks address one object manipulation on a tabletop. We leverage the LLM capabilities to paraphrase these tasks to produce the evaluation set. Paraphrases include task translation as the T5 model is trained for down stream tasks such as machine translation. Figure 25 illustrates the application of Task $m04$ submitted as a text based command in Korean language (”큐브를 테이블 중앙으로부터 20cm 위로 옮겨주세요”) to a policy trained in MTRL.

ID	Task
m01	Push the cube to the far right of the table.
m02	Move a cube to the top left corner of the table.
m03	Take the cube and put it close to the robot arm.
m04	Move a cube at 20cm above the center of the table.
m05	Move a cube at 15 cm above the table.
m06	Take the cube and put it on the diagonal of the table.
m07	Push the cube at 20cm ahead of its current position.
m08	Move the cube to the center of the table.
m09	Grab the cube and move it forward to the left.

Table 4: List of task used in the MTRL settings.

Figure 24 provides success rates obtained for the 9 tasks used in the MTRL experiment. It illustrates LARG^2 capability to generate valid reward functions to train and execute MTRL policies conditioned by textual task definitions.

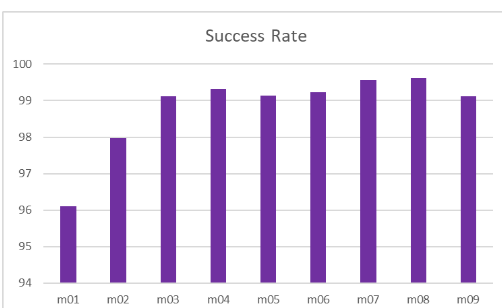


Figure 24: Success rate evaluations of MTRL over automatic reward generation.

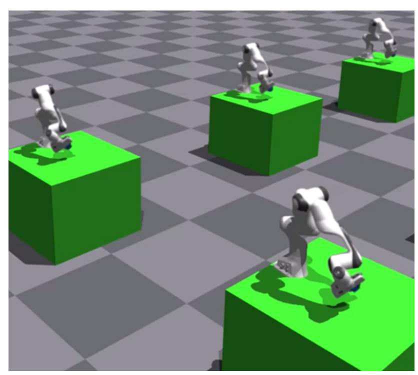


Figure 25: Example of multi-lingual capabilities for robot manipulation. In our simulation tasks are submitted using different languages including English, Arabic and Korean. This figure illustrate task $m04$ translated in Korean.

A.3 Examples of goal functions generated by LARG^2

```
import torch
from torch import Tensor
from typing import Tuple

def generate_goal_pose() -> Tensor:
    """This function returns an elligible goal pose satisfying the following table top rearrangement task: "can you take to cube and move it to the
right edge of the table."

    The setup is 1 table of dimension 0.81 x 1 x 0.795 on which stands 1 cube of edge 5cm and having its origin in its center.
    The global origin (0,0,0) is located on the floor below the middle of one of the table's sides.
    X coordinates range: (0., +0.70) = (Bottom, Top)
    Y coordinates range: (-0.4, +0.4) = (Right, Left)
    The minimum Z value for objects located on the surface of the table is 0.795 + 0.05/2 = 0.82

    Returns:
     - Tensor: a tensor of shape (3) representing the goal position
    """
    # The goal position will be at the right edge of the table, so we can set the y-coordinate to the maximum value of the range
    goal_pose = torch.tensor([0.35, 0.4, 0.82])
    return goal_pose
```

Figure 26: Task $d08$: Take to cube and move it to the right edge of the table

```
import torch
from torch import Tensor
from typing import Tuple

def generate_goal_pose() -> Tensor:
    """This function returns an elligible goal pose satisfying the following table top rearrangement task: "take the cube and move it very close to the
robotic arm."

    The setup is 1 table of dimension 0.81 x 1 x 0.795 on which stands 1 cube of edge 5cm and having its origin in its center.
    The global origin (0,0,0) is located on the floor below the middle of one of the table's sides.
    X coordinates range: (0., +0.70) = (Bottom, Top)
    Y coordinates range: (-0.4, +0.4) = (Right, Left)
    The minimum Z value for objects located on the surface of the table is 0.795 + 0.05/2 = 0.82

    Returns:
     - Tensor: a tensor of shape (3) representing the goal position
    """
    # move the cube very close to the robotic arm (at y=-0.35) and 5cm above the table surface
    goal_position = torch.tensor([0.6, -0.35, 0.82])
    return goal_position
```

Figure 27: Task $d12$: Take the cube and move it very close to the robotic arm.

```
import torch
from torch import Tensor
from typing import Tuple

def generate_goal_pose() -> Tensor:
    """This function returns an elligible goal pose satisfying the following table top rearrangement task: "move the cube to one corner
of the table."

    The setup is 1 table of dimension 0.81 x 1 x 0.795 on which stands 1 cube of edge 5cm and having its origin in its center.
    The global origin (0,0,0) is located on the floor below the middle of one of the table's sides.
    X coordinates range: (0., +0.70) = (Bottom, Top)
    Y coordinates range: (-0.4, +0.4) = (Right, Left)
    The minimum Z value for objects located on the surface of the table is 0.795 + 0.05/2 = 0.82

    Returns:
     - Tensor: a tensor of shape (3) representing the goal position
    """
    goal_pose = torch.tensor([0.7, -0.4, 0.82])
    return goal_pose
```

Figure 28: Task $d15$: Move the cube to one corner of the table.

```python
import torch
from torch import Tensor
from typing import Tuple

def generate_goal_pose(cube: Tensor) -> Tensor:
    """
    This function returns an elligible goal pose satisfying the following table top rearrangement task: "place the cube anywhere on the
diagonal of the table  running from the top right corner to the bottom left corner."

    The setup is 1 table of dimension 0.81 x 1 x 0.795 on which stands 1 cube of edge 5cm and having its origin in its center.
    The global origin (0,0,0) is located on the floor below the middle of one of the table's sides.
    X coordinates range: (0., +0.70) = (Bottom, Top)
    Y coordinates range: (-0.4, +0.4) = (Right, Left)
    The minimum Z value for objects located on the surface of the table is 0.795 + 0.05/2 = 0.82

    Args:
     - cube: a tensor of shape (3) representing the cube position on the table

    Returns:
     - a tensor of shape (3) representing the goal position on the table
    """
    goal_x = torch.tensor([0.7]).to(cube.device)
    goal_y = torch.tensor([-0.4]).to(cube.device)
    goal_z = torch.tensor([0.82]).to(cube.device)

    # calculate the slope of the diagonal line
    slope = -(goal_y / goal_x)

    # calculate the y intercept of the diagonal line
    y_int = goal_y - (slope * goal_x)

    # use the slope and y intercept to calculate the desired y position of the cube
    goal_y_cub = (slope * cube[0]) + y_int

    # create the goal pose tensor
    goal_pose = torch.tensor([cube[0], goal_y_cub, goal_z]).to(cube.device)

    return goal_pose
```

Figure 29: Task $d16$: Place the cube anywhere on the diagonal of the table running from the top right corner to the bottom left corner.

```python
import torch
from torch import Tensor
from typing import Tuple

def generate_goal_pose() -> Tensor:
    """This function returns an eligible goal pose satisfying the following table top rearrangement task: "lift the cube 15cm above the
table and 10 cm to the right."

    The setup is 1 table of dimension 0.81 x 1 x 0.795 on which stands 1 cube of edge 5cm and having its origin in its center.
    The global origin (0,0,0) is located on the floor below the middle of one of the table's sides.
    X coordinates range: (0., +0.70) = (Bottom, Top)
    Y coordinates range: (-0.4, +0.4) = (Right, Left)
    The minimum Z value for objects located on the surface of the table is 0.795 + 0.05/2 = 0.82

    Returns:
     - Tensor: a tensor of shape (3) representing the goal position
    """
    goal_pose = torch.Tensor([0.35, 0.1, 0.82+0.15])
    return goal_pose
```

Figure 30: Task $d17$: Lift the cube 15cm above the table and 10 cm to the right.

```python
import torch
from torch import Tensor
from typing import Tuple

def generate_goal_pose() -> Tensor:
    """This function returns an eligible goal pose satisfying the following table top rearrangement task: "lift the cube 20cm above the
table and 15 cm backward."

    The setup is 1 table of dimension 0.81 x 1 x 0.795 on which stands 1 cube of edge 5cm and having its origin in its center.
    The global origin (0,0,0) is located on the floor below the middle of one of the table's sides.
    X coordinates range: (0., +0.70) = (Bottom, Top)
    Y coordinates range: (-0.4, +0.4) = (Right, Left)
    The minimum Z value for objects located on the surface of the table is 0.795 + 0.05/2 = 0.82

    Returns:
     - Tensor: a tensor of shape (3) representing the goal position
    """

    # We need to lift the cube vertically by 20 cm and move it back by 15 cm
    x_pos = 0.7
    y_pos = -0.15
    z_pos = 0.82 + 0.05

    goal_pose = torch.tensor([x_pos, y_pos, z_pos])

    return goal_pose
```

Figure 31: Task $d19$: Lift the cube 20cm above the table and 15 cm backward.

```python
import torch
from torch import Tensor
from typing import Tuple

def generate_goal_pose() -> Tensor:
    """This function returns an elligible goal pose satisfying the following table top rearrangement task: "grab a cube and lift it a
bit and move it a bit ahead."

    The setup is 1 table of dimension 0.81 x 1 x 0.795 on which stands 1 cube of edge 5cm and having its origin in its center.
    The global origin (0,0,0) is located on the floor below the middle of one of the table's sides.
    X coordinates range: (0., +0.70) = (Bottom, Top)
    Y coordinates range: (-0.4, +0.4) = (Right, Left)
    The minimum Z value for objects located on the surface of the table is 0.795 + 0.05/2 = 0.82

    Returns:
     - Tensor: a tensor of shape (3) representing the goal position
    """

    # Define the goal position
    goal_pos = torch.randn(3) * 0.2  # Random position within a sphere of radius 0.2

    # Adjust goal position to be within the valid range
    goal_pos[0] = max(0., min(goal_pos[0], 0.7))  # X range: (0., +0.70)
    goal_pos[1] = max(-0.4, min(goal_pos[1], 0.4))  # Y range: (-0.4, +0.4)
    goal_pos[2] = max(0.82, goal_pos[2])  # Minimum Z for objects on the table surface

    return goal_pos
```

Figure 32: Task $d25$: Grab a cube and lift it a bit and move it a bit ahead.

```python
import torch
from torch import Tensor
from typing import Tuple

def generate_goal_pose(cube: Tensor) -> Tensor:
    """This function returns an eligible goal pose satisfying the following table top rearrangement task: "move the cube at 20cm to the
left of its initial position."

    The setup is 1 table of dimension 0.81 x 1 x 0.795 on which stands 1 cube of edge 5cm and having its origin in its center.
    The global origin (0,0,0) is located on the floor below the middle of one of the table's sides.
    X coordinates range: (0., +0.70) = (Bottom, Top)
    Y coordinates range: (-0.4, +0.4) = (Right, Left)
    The minimum Z value for objects located on the surface of the table is 0.795 + 0.05/2 = 0.82

    Args:
     - Tensor: a tensor of shape (3) representing the cube position on the table

    Returns:
     - Tensor: a tensor of shape (3) representing the goal position on the table
    """
    # Calculate the goal position by moving 20cm to the left of the initial position
    goal_position = torch.tensor([0.2, cube[1] - 0.2, 0.82])
    return goal_position
```

Figure 33: Task $d26$: Move the cube at 20cm to the left of its initial position.

```python
import torch
from torch import Tensor
from typing import Tuple

def generate_relative_goal_poses(cube1: Tensor, cube2: Tensor, cube3: Tensor) -> Tuple[Tensor, Tensor, Tensor]:
    """This function returns a tuple of elligible goal poses satisfying the following table top rearrangement task: "rearrange three cubes in such a way that the distance between each of them is 10 centimeters."

    The setup is 1 table of dimension 0.75 x 0.8 x 0.78 on which stands 3 cubes of edge 5cm.
    Each cube has its own origin in its center.
    The global environment origin (0,0,0) is located on the floor below the middle of one of the table's sides.
    X coordinates range: (0., +0.75) = (Bottom, Top)
    Y coordinates range: (-0.4, +0.4) = (Right, Left)
    The minimum Z value for objects located on the surface of the table is 0.78 + 0.05/2 = 0.805

    Args:
     - Tensor: a tensor of shape (3) representing the position of the first cube on the table
     - Tensor: a tensor of shape (3) representing the position of the second cube on the table
     - Tensor: a tensor of shape (3) representing the position of the third cube on the table

    Returns: Tuple(Tensor, Tensor, Tensor)
     - Tuple: a tuple of 3 tensors. Each tensor of shape (3) represents the goal position of one cube on the table
    """

    distance = 10 / 100  # Convert distance to meters
    if cube1[0] >= 0.5:
        cube1[0] = 0.5
    if cube1[1] >= 0.2:
        cube1[1] = 0.2
    if cube1[1] <= -0.2:
        cube1[1] = -0.2

    goal_positions = [cube1]

    for reference in [cube2, cube3]:
        found = False
        while not found:
            angle = torch.rand(1, device=reference.device) * 2 * torch.pi
            x_offset = distance * torch.cos(angle)
            y_offset = distance * torch.sin(angle)
            candidate_position = goal_positions[-1] + torch.tensor([x_offset, y_offset, 0], device=reference.device).squeeze()

            # Check if candidate_position is within table boundaries
            if (0 < candidate_position[0] < 0.75) and (-0.45 < candidate_position[1] < 0.45) and candidate_position[2] >= 0.8:
                all_distances_valid = True

                for pos in goal_positions:
                    dist = torch.norm(candidate_position - pos)
                    if not torch.isclose(dist, torch.tensor(distance, device=reference.device), rtol=1e-2, atol=1e-2):
                        all_distances_valid = False
                        break

                if all_distances_valid:
                    found = True
                    goal_positions.append(candidate_position)
            else:
                print(candidate_position)

    return tuple(goal_positions)
```

Figure 34: Task 29: Rearrange three cubes in such a way that the distance between each of them is 10 centimeters.

```python
import torch
from torch import Tensor
from typing import Tuple

def generate_goal_poses() -> Tuple[Tensor, Tensor, Tensor]:
    """
    This function returns an eligible goal pose satisfying the following table top rearrangement task: "Move the three cubes on the
table so at the end they form a right-angled triangle wih one corner at the center of the table."

    The setup is 1 table of dimension 0.75 x 0.8 x 0.78 on which stands 3 identical cubes of edge 5cm each and having their origins in
their respective centers.
    The global origin (0,0,0) is located on the floor below the middle of one of the table's sides.
    X coordinates range: (0., 0.75)
    Y coordinates range: (-0.4, 0.4)
    The minimum Z value for objects located on the surface of the table is 0.78 + 0.05/2 = 0.805

    Returns a tuple of Tensors:
     - [Tensor, Tensor, Tensor]: Each tensor is of shape (3) and contains a goal position
    """

    # Define the positions of the three cubes
    pos1 = torch.tensor([0.45, -0.25, 0.805])
    pos2 = torch.tensor([0.65, 0.25, 0.805])
    pos3 = torch.tensor([0.35, 0.25, 0.805])

    # Rotate the positions to form a right-angled triangle with one corner at the center of the table
    angle = torch.tensor([0, 0, -45]) * (3.14159 / 180)
    rot_mat = torch.tensor([[torch.cos(angle[2]), -torch.sin(angle[2]), 0],
                            [torch.sin(angle[2]), torch.cos(angle[2]), 0],
                            [0, 0, 1]])
    center_pos = torch.tensor([0.375, 0, 0.805])
    pos1 = torch.matmul(rot_mat, pos1 - center_pos) + center_pos
    pos2 = torch.matmul(rot_mat, pos2 - center_pos) + center_pos
    pos3 = torch.matmul(rot_mat, pos3 - center_pos) + center_pos

    return pos1, pos2, pos3
```

Figure 35: Task $d30$: Move the three cubes on the table so at the end they form a right-angled triangle with one corner at the center of the table.