Code as Reward: Empowering Reinforcement Learning with VLMs

David Venuto [1 2] **Sami Nur Islam** [2] **Martin Klissarov** [1 2] **Doina Precup** [1 2 3] **Sherry Yang** [3 4] **Ankit Anand** [3]

Abstract

Pre-trained Vision-Language Models (VLMs) are able to understand visual concepts, describe and decompose complex tasks into sub-tasks, and provide feedback on task completion. In this paper, we aim to leverage these capabilities to support the training of reinforcement learning (RL) agents. In principle, VLMs are well suited for this purpose, as they can naturally analyze image-based observations and provide feedback (reward) on learning progress. However, inference in VLMs is computationally expensive, so querying them frequently to compute rewards would significantly slowdown the training of an RL agent. To address this challenge, we propose a framework named Code as Reward (VLM-CaR). VLM-CaR produces dense reward functions from VLMs through code generation, thereby significantly reducing the computational burden of querying the VLM directly. We show that the dense rewards generated through our approach are very accurate across a diverse set of discrete and continuous environments, and can be more effective in training RL policies than the original sparse environment rewards.

1. Introduction

One of the important steps in deploying reinforcement learning (RL) agents in real-world problems with RL is defining an appropriate reward function, which both captures the goal of the problem and allows an RL agent to learn efficiently. The reward function is often defined and refined by an environment designer, with significant effort. In this paper, we explore the use of large Vision-Language models (VLMs), which have recently become publicly available, for this purpose. VLMs are capable of learning intricate relationships between images and text. Integrated with large language models (LLMs), VLMs have been applied across a wide range of multi-modal tasks, from visual question answering (Zhu et al., 2023) to robotics (Brohan et al., 2023). LLMs have also been used to craft better exploration strategies for RL agents, e.g. (Klissarov et al., 2023; Yu et al., 2023) or to produce programs that represent policies (Liang et al., 2023) or skills (Wang et al., 2023). Since pre-trained VLMs can accurately describe tasks and generate sub-tasks from image inputs, they are perfect candidates for enhancing RL agents that work with image-based or multi-modal observations.

In this paper, we aim to leverage VLMs for computing rewards from images, which can then be used to train RL agents. The most direct approach to achieve this goal is to directly finetune a VLM to be a reward model, as in (Yang et al., 2023; Du et al., 2023b). However, directly querying a VLM to obtain reward has a few disadvantages. First, it is computationally inefficient due to the sheer size of VLMs. Second, the stochastic output from a VLM can be unreliable. Third, rewards directly computed by a VLM are generally not interpretable nor verifiable. To address these problems, we present a framework called Code as Reward (VLM-CaR) which leverages pretrained VLMs to obtain efficiently computable, reliable, and verifiable reward functions. Our approach is to first prompt the VLM to describe the task/sub-tasks solely from an initial and goal frame of an environment. Given these high-level language descriptions, we use the VLM to generate a set of executable computer programs, one for each proposed sub-task, which check for successful sub-task completion. To verify that the task decomposition and reward function programs are indeed correct, we propose an automated *code verifier*, which uses a handful of expert trajectories and random trajectories to validate the obtained rewards. If the produced solution is correct, expert trajectories should achieve high reward, whereas random policies should achieve low reward. Once the generated sub-task and reward programs pass the verification check, we use them as reward functions to train RL agents. This approach can be viewed as an implementation of the option framework in RL (Sutton et al., 1999), where the VLM is used to produce the reward functions for training each option, as well as for the policy over options, and the rewards are represented programatically. We show that we can successfully train RL agents using the rewards provided

[1]Mila [2]McGill University [3]Google DeepMind [4]University of California, Berkeley. Correspondence to: David Venuto <david.venuto@mail.mcgill.ca>.

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

by VLM-CaR across a variety of grid-based discrete action environments and continuous robotic control environments. Our approach has the additional benefit of providing dense rewards, which mitigates the notorious difficulty of training RL agents in sparse reward settings (Hare, 2019).

Our contributions can be summarized as follows:

- We propose a framework for prompting VLMs to generate sub-tasks and reward functions, represented as code, and discuss the relationship of this approach to the options framework.
- We propose a reliable approach for verifying and refining the generated programs, by using a small amount of expert trajectories.
- We show that the proposed approach enables effective training of RL policies, by providing dense rewards for tasks where the reward is either sparse or not available (which is often the case in real-world problems).

2. Related Work

2.1. Vision-Language Models

VLMs extract embeddings from both visual and text inputs. Convolutional layers are used to encode visual features, while transformers are used to process text data; a cross-modal attention mechanism is used to establish connections between the embeddings of the image and text inputs. During this process, the model learns to attend to relevant parts of the image while processing the text and vice versa.

Research has explored the capabilities of VLMs to understand complex relationships between images and text. VLMs like CLIP (Contrastive Language-Image Pre-training) (Radford et al., 2021) and UNITER (Unified Pre-trained Image-Text Transformer) (Chen et al., 2020) have demonstrated impressive performance in various tasks involving both vision and language, such as scene description. Larger models such as GPT-4 (OpenAI et al., 2023) have recently become available and showcase noteworthy performance across breadth of tasks, as well as with increasing task complexity. VLM-CaR builds upon these advancements by extending the utility of VLMs to reinforcement learning, specifically by providing reward design in an interpretable and automated manner.

2.2. VLMs as Reward Models

VLMs are particularly useful for crafting rewards functions in RL, as they directly work at the intersection of visual observations and language descriptions. OpenAI's CLIP model (Radford et al., 2021) has been used to derive dense reward functions, by taking the cosine similarity between the embeddings of a goal description in language and a visual representation of the current state (or possibly a sequence of states). MineDojo (Fan et al., 2022) builds on the CLIP model and fine-tunes it on thousands of hours of YouTube videos showcasing human game-play in MineCraft (Kanervisto et al., 2022). The result is the MineCLIP model, which is used to guide an RL agent in achieving success on various specific tasks described in natural language, such as "collect wood". A similar strategy is used by Ding et al. (2023) which instead relies on a curated dataset of videos to fine-tune the original CLIP model. Beyond MineCraft, VLMs have also been fine-tuned to produce rewards in robotics, by leveraging spatial language descriptions of the visual observations (Mahmoudieh et al., 2022; Yang et al., 2023; Du et al., 2023b), by extrapolating annotations from a smaller dataset to a much larger one (Xiao et al., 2022) or by computing cosine similarity at the level of trajectories (Sontakke et al., 2023). Researchers have also investigated leveraging the CLIP model (Alayrac et al., 2022) for computing reward in a zero-shot manner (Cui et al., 2022; Rocamonde et al., 2023; Baumli et al., 2023; Du et al., 2023a).

The Flamingo (Alayrac et al., 2022) model has also been used for its ability to take as input both text and image, and output text. This contrasts with approaches that leverage the latent embeddings of the CLIP model to compute a cosine similarity. Flamingo can indeed answer with coherent textual responses to queries from the user. This ability was leveraged by Du et al. (2023a) to use a fine-tuned Flamingo model as a success detector for embodied robotic tasks. The model is presented with observations from the environment and is asked to infer whether a task was completed, essentially performing visual question answering. Most of these methods require querying the model multiple times per episodes in order to derive a reward function. This inference cost comes on top of the need to first perform a fine-tuning stage, which may be impractical for larger VLMs. Another important difference with previous work is the fact that most of these methods do not use the VLMs for their (limited) reasoning capabilities, but rather for their embedding space. One exception is Du et al. (2023a), however their fine-tuned model only outputs simple binary answers. Instead, VLM-CaR leverages a VLM in a zero-shot manner for crafting reward functions via multiple rounds of prompting through which the VLM can reason about the environment, the subgoals and how to achieve them. Finally, none of the related methods provide interpretable solutions (as programs or otherwise) for the reward function being generated by the VLM. As VLM-CaR produces code functions, it provides additional reliability before deployment for inspection whenver required.

2.3. Code as Policies and Rewards

Previous work has prompted LLMs to output code whose execution produces robot control actions (Liang et al., 2023). While this is an interesting approach, policies can be obtained through many other effective techniques, including by running RL algorithms. On the other hand, obtaining reward functions in real-world applications is much more difficult (Dulac-Arnold et al., 2021; Armstrong et al., 2021; Devidze et al., 2021; Kwon et al., 2023; Ratner et al., 2018). Approaches similar to code as policies have been briefly explored in LLMs to provide reward in some restricted tasks (Ma et al., 2023; Kwon et al., 2023), but without incorporating image-based observations, which are common in robotics and other applications. To our knowledge, this work is the first to propose the idea of using VLMs to generated code implementing reward functions.

3. Preliminaries

In this section we define notation and briefly review relevant reinforcement learning concepts.

Sequential decision making problems are often modeled as Markov Decision Processes (MDPs) (Bellman, 1954). An MDP is a 5-tuple $\langle S, A, T, R, \gamma \rangle$, where S is the state space, A is the action space, $T : S \times A \times S \rightarrow [0, 1]$ is the transition function, $R : S \times A \rightarrow \mathbb{R}$ is the reward function and $0 < \gamma < 1$ is the discount factor. At each time step t, the agent executes an action $a_t \in A$ in state $s_t \in S$ to get a reward $r_t = R(s_t, a_t) \in R$, and moves to another state s_{t+1} with probability given by $T(s_t, a_t, s_{t+1})$. The objective is to find a policy $\pi : S \rightarrow A$ from the space of all possible policies Π so as to maximize the expected return (discounted sum of rewards), $\max_{\pi \in \Pi} \mathbb{E} \sum_{t=0}^{\infty} \gamma^t r_t$, where $a_t = \pi(s_t)$. The reward function R is conceptualized as describing the goal of the agent, and is usually thought of as part of the environment. However, in practice, R usually has to be provided by a domain expert. While many reward functions can share the same optimal policy, having a more informative reward function, which guides the agent at every step, can significantly improve the speed of learning.

Nevertheless, once a good reward function is available, an array of RL algorithms, ranging from model-based (Doya et al., 2002) to value-based (Sutton & Barto, 2018) and policy-based (Williams, 1992), can be applied to find a policy that maximizes the expected future returns.

A key challenge for RL algorithms is to learn and plan over long horizons, especially when rewards are sparse. The options framework (Sutton et al., 1999) provides a formalism for temporal abstraction, which helps both exploration and credit assignment in such problems. The main idea is to break the full problem that the agent aims to solve into subtasks, each of which usually has its own reward function and can be completed by a separate policy. Formally, each option ω is defined by 3-tuple $\langle \mathcal{I}_\omega, \pi_\omega, \beta_\omega \rangle$, where $\mathcal{I}_\omega \subset S$ is the set of states where ω is applicable, π_w is the policy used when option ω is executing and $\beta_\omega : S \rightarrow [0, 1]$ indicates the probability that option ω terminates in each state While this framework gives the necessary tools for solving a complex task if the sub-tasks are defined, it remains difficult to discover options automatically, end-to-end from data (Bacon et al., 2017; Harb et al., 2017; Klissarov & Precup, 2021).Many works focus instead on using pre-trained option policies (Huang et al., 2022; Ahn et al., 2022). In this work, we use the generalization capabilities of VLMs to define sub-tasks for solving a complex task, which in turn leads to dense reward functions for training RL policies later. Our approach can be viewed as using the VLM to generate code corresponding to the termination condition of the option, as well as to the reward function that can be used to learn the policy of the option.

4. Proposed approach

In this section, we present our framework, VLM-CaR, to autonomously craft options (sub-tasks) and reward functions through code generation using the GPT-4 web interface. Figure 1 presents an overview of our approach. Our framework consists of three distinct stages: a) *Generate Programs*, b) *Verify Programs*, and c) *RL Training*. During the first stage, we use pre-trained VLMs to generate sub-tasks and reward definitions as programs. During the second stage, we verify and refine the correctness of the generated sub-tasks and reward functions by evaluating a handful of expert trajectories and random trajectories. During this stage, we use the reward function to train RL agents. We explain the details of each stage in the subsequent sections

4.1. Generating Rewards and Sub-tasks

We primarily rely on sequential prompting for generating programs that represent sub-tasks and reward functions. Although VLMs have good understanding of visual inputs, we have found that they are not yet reliable enough to generate correct rewards in a zero-shot manner for complex tasks, like those under consideration in RL benchmarks. To handle this, we prompt the pre-trained VLMs in a sequential manner, incrementally building up the necessary information to specify the reward functions, while using verification checks wherever applicable.

Concretely, the sequence of prompts obey the following structure across all tasks presented in this work. Given only the initial frame of a trajectory, we first prompt the VLM to identify what part represents the agent in the image and ask it to provide a script returning the agent's location. We then provide the VLM with an image where the agent has reached its goal state and ask the VLM to infer what the task was. Next, we ask the VLM to describe all the relevant objects in the image. Given this sequential build-up of relevant information and objects, we ask the VLM to provide n

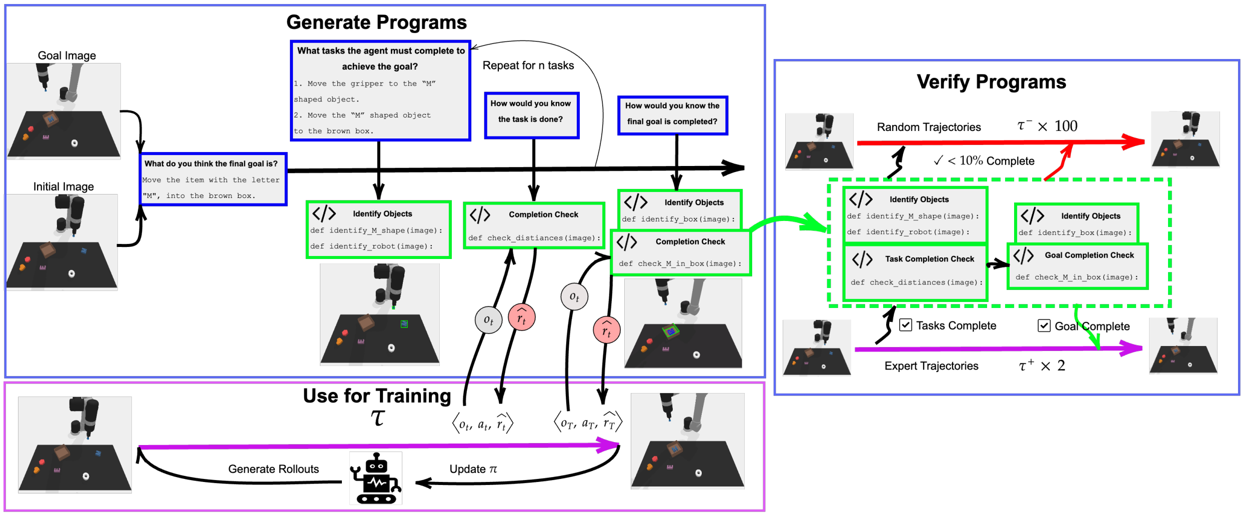

Figure 1. Complete pipeline of VLM-CaR, describing how code blocks for sub-tasks and rewards are generated. The top portion is the reward script generation pipeline, which uses the VLM, and the bottom portion is the RL training loop. The feedback loop is shown on the right and is used to determine if the task and goal code blocks are correct. The middle portion in green represents the generated scripts from the VLM. The task completion scripts are applied to random and expert trajectories to compute if the task was completed or not. All tasks should be completed in expert trajectories and rarely completed in random trajectories.

sequential sub-tasks (with n being a tune-able parameter), that are actionable by the agent and will result in the final goal completion. An example set of tasks is shown in Figure 1. For each sub-task, we prompt the VLM to infer the subset of objects that are relevant for verifying the completion of this particular sub-task, followed by generating a program for identifying the relevant objects. We finally ask the VLM to provide a program for identifying if the sub-task is complete. We verify this program using the verification pipeline in Subsection 4.2. This serves as an initial set of functions that can be thought of as implementing both initiation and termination conditions for options corresponding to each sub-task. In other words, the objects in the image, identified by the VLM, determine both which options are affordable (Khetarpal et al., 2020b; Ahn et al., 2022) as well as when options terminate. For reaching the final goal, we utilize the same approach to determine the relevant objects and a program to identify if the final goal is completed. We show the full prompting pipeline in Appendix A and B as well as an example interaction with the VLM in Appendix C.

4.2. Verification using Expert and Random trajectories

A possibility for obtaining reliable programs is to ask humans to visually inspect the correctness of the code through a set of observations from the environments. While human expertise can be invaluable, it can limit the scaling of this approach, as generated scripts are complex, especially in real world environments. Therefore, we automate the verification of generated scripts through a verification pipeline.

To automatically verify the correctness of the generated programs described in the previous section, we use 2 trajectories generated by an expert agent and 100 randomly generated trajectories. We execute the generated object identification and sub-task programs on the random and expert trajectories. Every sub-task must be completed in the expert trajectories and completed in less than $p\%$ of the random trajectories (where p is a tune-able parameter). It is important to note that it is very important to use random trajectories as negative data, in order to catch any false positive sub-tasks generated by the model. This verification stage is shown on the right in Figure 1. If the generated programs do not pass this verification stage, the VLM is prompted anew and asked to refine the program for sub-tasks which did not pass the automated verification test. This allows us to ensure the correctness of the sub-tasks and reward functions.

4.3. Using Generated Programs in the RL loop

Once the generated programs are verified, we use them to improve the learning efficiency of an RL agent. The programs generated for sub-task completion are added as auxiliary reward functions in addition to the original environment reward. Specifically, once the proposed sub-task is determined to be completed, an additional auxiliary reward (r_{aux}) is provided to the agent and we move to the determining if the next task is completed or not. Once all sub-tasks are completed, we call the final goal completion script. The assembly of all of the VLM's proposed sub-goals scripts leads to a dense reward model. These sub-task completion checks could also be thought of as option termination conditions. In that view, the VLM generates the option initiation and option termination conditions as programs, while the option policy is learned with reinforcement learning. It should be noted that we can also leverage hierarchical learning frameworks like options with the generated programs but in this work, we stick to auxiliary rewards for simplicity.

5. Experiments

We conduct experiments with the VLM-CaR-generated reward function to show two major improvements: (1) VLM-CaR can transform a sparse reward function into a set of dense reward functions for each sub-task. These per-task rewards are much more efficient for training RL agents than the environment provided sparse rewards. We show these results in discrete action grid environments and in robotic control tasks. (2) VLM-CaR can generate a reward function for difficult high dimensional robotic environments only from the an image of the initial and completion state.

5.1. Experimental Procedure

We use the GPT-4 web interface for all experiments, which allows access to VLM capabilities and the uploading of images. This product is available with a subscription of only $20 USD per month. We employ the mentioned set of prompts and pipeline to generate scripts which discover tasks and goals, identify the relevant objects in each task and goal and check its completion. Note that this design is highly cost efficient and can be handled solely from the GPT-4 web interface without requiring additional cost.

Since the VLMs are not accurate enough to generate reward and sub-task program in first attempt, we need to run the generation pipeline multiple times till our verification pipeline succeeds. In Table 1, we list the number of failed attempts for each suggested task and object identification. We also list the number of regenerations required due to the internal GPT-4 timeout for taking too long to complete a response. It also shows that VLMs are good enough to generate these programs in a few attempts and efficacy of our pipeline to catch false positives. We show a few inaccurate programs caught by our verification pipeline in Appendix D as well as usefulness of random trajectories.

5.2. Manual Steps

The VLM gives us a set of functions, usually separate blocks of code to identify the objects and check task completion. We do not ask the VLM to assemble everything automatically into a single class. In the current implementation, we only assemble these scripts together into a single class which has a callable reward function (as one of its method). We use the generated sub-task functions by VLMs to create a very simple reward function which calls the VLM generated functions (for sub-tasks) and we give a reward of $1/n$ for each of the n tasks. This step also requires us to store a boolean variable for each trajectory indicating if the task is done, so we can move onto checking the next task. These variables are reset at the start of every episode. While this process of having human-in-the-loop for assembling sub-task scripts can be seen as a potential limitation for scaling to more complex tasks, we also believe this can be addressed in future work by prompting the VLM to assemble the final script or using the sub-task completion programs within the option framework where the learning algorithm can directly use these programs for option termination. It is not achieved in this work due to practical limitation of timeouts in the web interface of GPT-4 for large contexts and programs.

We believe this work realises the potential of using VLMs to generate programs as a first step. Building completely automated end-to-end pipeline without any human in the loop is an interesting direction for future research. As VLMs continue to scale and with a proper API access to the VLM, we believe that we could eliminate this manual task.

5.3. MiniGrid

We first show the Gym-MiniGrid (Chevalier-Boisvert et al., 2023) set of partially-observable environments, for which rewards are sparse and there is a non-zero reward only when the agent completes the task. The agent is only given a local ego-centric view of its environment. For example, in DoorKey a key must first be obtained and then a door needs to be opened leading to another room with the goal state. We use the sparse environment based reward function for evaluation in our experiments. In all MiniGrid experiments we use proximal policy optimization (PPO) (Schulman et al., 2017) to train our RL agent. In this environment, VLM is able to generate sub-tasks such as as obtaining key, reaching door and then reaching final goal. Each sub-task is assigned an auxiliary reward of $1/2$ with a final reward of 1 for completion of the goal. We show that the VLM-CaR generated auxilary rewards are useful and accelerate learning in contrast to just using the sparse evaluation reward of the environment in Figure 2. The agent trained with VLM-CaR rewards is able to complete the difficult DoorKey8x8 environment whereas traditional PPO is not. In all environments we show remarkable improvements in training efficiency.

Reward Strategies: For DoorKey and Unlock, the agent is identified by looking for a red triangle, with 3 vertices in the contour. The Key is identified by searching for a complex yellow shaped object, identified by an object that has greater than 4 vertices. The door is identified as searching for a yellow rectangle by determining color and the number of vertices. Completion checks consist of identifying if the key

Table 1. This table shows the number of failed attempts (Failed) that the VLM had for each task in each environment. We also list the number of response regenerations (Regens) required due to the internal timeout for generating a response. Failed attempts could be due to the automated verification pipeline where we proposed, or the first-pass check with the VLM not identifying any object in the initial image (returning False during object identification). N/R indicates that this object was not used in any task or goal completion scripts. We also provide a brief textual description of the strategy employed in the generated identification and completion check script. For example vertices means the shape was identified by checking the number of vertices (4 is typically a rectangle or square, > 4 is a more complex shape). n denotes the number of tasks asked for the in the pipeline.

Environment	Task	Failed	Regens	Strategy Employed	n
MiniGrid	Identify Agent	2	0	Vertices/Colour	3
	Identify Key	2	0	Vertices/Colour/Size	
	Check Key Complete	2	0	Key Disappeared	
	Identify Door	0	0	Vertices/Colour	
	Check Door Complete	4	0	Door Shape Change	
	Identify Goal	0	0	Vertices/Colour	
	Check Goal Completion	3	0	Agent Proximity	
PandasGym	Identify Agent	N/R	N/R	Vertices/Colour/Size	2
	Identify Green Block	3	2	Colour/Size	
	Check Green Block Moved	3	1	Moved from Initial Pos	
	Identify Yellow Block	1	0	Colour/Size	
	Check Goal Completion	2	1	Contour Overlap	
separating-piles	Identify Yellow Square	2	0	Colour/Size	1
	Identify Blue Blocks	3	1	Colour	
	Check Goal Completion	5	4	Contour overlap	
put-blocks-in-bowls	Identify Green Bowls	3	0	Colour/Size/Circularity	1
	Identify Red Blocks	1	2	Colour/Size	
	Check Goal Completion	6	2	Contour overlap	
packing-shapes	Identify M Shaped Object	11	0	Vertices/Colour	1
	Identify Brown Box	2	1	Colour/Size	
	Check Goal Completion	4	1	Contour Overlap	

has disappeared from the image to suggest completion of the first task and if the door has opened to check completion of the 2nd task. The "is_door_open" script is remarkably intuitive, checking if the door has changed shape from an approximate square to a smaller rectangle. We provide all generated scripts in the Supplementary Material.

5.4. Pandas-Gym

Pandas-Gym provides a simulation environment to benchmark RL agents on a variety of continuous control tasks (Gallouédec et al., 2021). In these experiments, we control the joints of the robot. We examine two tasks in these experiments, in each task the agent must move a green cubic object into a yellow target area either by pushing (PandasPush) or sliding (PandasSlide) it across a long surface. We show results in Figure 3 where the VLM-CaR rewards improve the success rate in contrast to just using environmental rewards. It also improves samply efficiency for a given success rate. We use Truncated Quantile Critics (TQC) (Kuznetsov et al., 2020) to train our policies, the current optimal benchmark for Pandas-Gym.

Reward Strategies: In the first push task, the green and yellow blocks are identified correctly. The first task is to make contact with the green cube. The completion check proposed is to verify if the green block has moved by comparing its current position to its position in the initial image. The final reward function for the goal proposed by the VLM is to determine the Euclidean distance between goal and block relative to initial image. It is then computed at each step afterwards, determining the progress that the agent has made in moving the block to the goal. The reward function is $r = (d_{\text{initial}} - d_{\text{current}})/d_{\text{initial}}$, where d indicates the Euclidean distance. Since objects are of the same colour and shape in all tasks, this script was found to be generalizable to both PandasPush and PandasSlide.

5.5. CLIPort

Lastly, we focus on robotic environments utilized in CLIPort (Shridhar et al., 2021). These are simulated robotic tasks and utilize a Universal Robot (UR5e) with a suction gripper. CLIPort agents are trained using imitation learning, so we cannot access the online training procedure like we illustrated in previous two environments. In these experiments, we illustrate that VLM-CaR can correctly generate an effective reward function for this complex environment. Specifically, we evaluate 4 agents from the pre-trained checkpoints of varying skill levels i.e random, novice, sub-optimal and expert. We roll-out trajectories from these different policies and evaluate our VLM-CaR generated reward function on

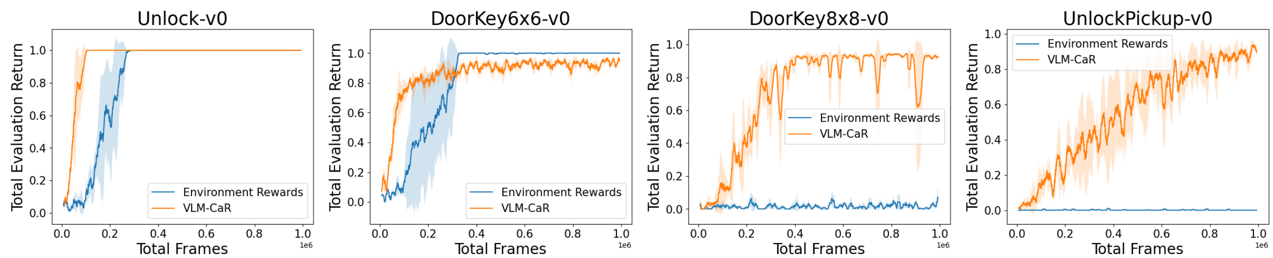

Figure 2. The online episodic mean reward evaluated over 5 episodes every 250 steps for MiniGrid RL tasks. We show the average over 3 random seeds. $1M$ environment step interactions are used. The shaded area shows the standard error. Agents trained using rewards generated by VLM-CaR perform better than the sparse environment reward. In some tasks, sparse rewards are not sufficient for any meaningful performance whereas VLM-CaR rewards allow the agent to solve the task.

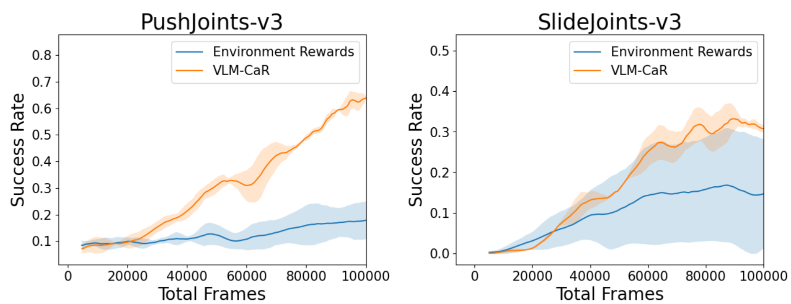

Figure 3. The success rate in completing the final task in Pandas-Gym environments. 5 random seeds are shown. The shaded area is the standard deviation. RL agent trained on dense reward generated by VLM-CaR generally performs better than RL agent trained on sparse environment rewards.

these trajectories.

For robotic tasks, we ask the VLM to give an incremental reward model if it determines that is more appropriate. For the Robotic Goal Completion Pipeline, we only provide a reward function for completing the final goal. This reward function is usually dense in many robotic tasks, where for example the agent must move multiple objects to a target location. We observe that it is common for the VLM to propose the Euclidean distance of the targets from the goal location as the goal reward model in this case.

The setup provides a complex environment for evaluating the ability to ground natural language concepts like colors and object categories. The input is a top-down RGB-D image from 3 cameras positioned around a rectangular table: one in the front, one on the left, and one on the right, all pointing towards the center of the table. For VLM-CaR, we use the camera overlooking the center of the table from a 45 degree angle to provide images of the initial and goal state. We examine 3 tasks in this UR5e environment. The first, `separating-piles`, the agent must sort blue blocks into a yellow square shaped region. In `put-blocks-in-bowls`, the agent must move 2 or 3 red blocks into the green bowls. In `packing-shapes`, the agent must move a blue ”M” shaped object into a large brown box. For simplicity, we fix the colours and shapes of the relevant objects in each task, since a component of our VLM generated reward function utilizes the identification of colours as well as shapes. CLIPort is trained utilizing textual descriptions of the task, for example “sort the red blocks into the green square”. For utilizing different coloured objects, it is easy to see that we could modify VLM-CaR to correctly identify different coloured objects using the textual descriptions.

Reward Strategies: For `separating-piles`, VLM-CaR generates a reward function that first identifies the yellow square from the initial frame and saves its contour for future use. VLM-CaR then identifies all the blue block contours in the image and give a numerical reward that is the fraction of blue blocks inside the yellow square to the portion outside. For `put-blocks-in-bowls`, VLM-CaR first identifies the red cubic shaped objects. The green circular shaped bowls are then identified and the contours are stored from the initial image. VLM-CaR then gives a reward of 1 for each red block that is identified in the green bowl contour. For `packing-shapes`, VLM-CaR first identifies the blue ”M” shaped object by checking if the shape is more complex than a rectangle (with vertices > 4) and determining that it has a blue color and has a relatively large size. VLM-CaR then identifies the brown box by size, if it is rectangular and by color. The reward is 1 for identifying the ”M” shaped object inside a bounding contour of the box. We utilize the robotic-goal-completion pipeline

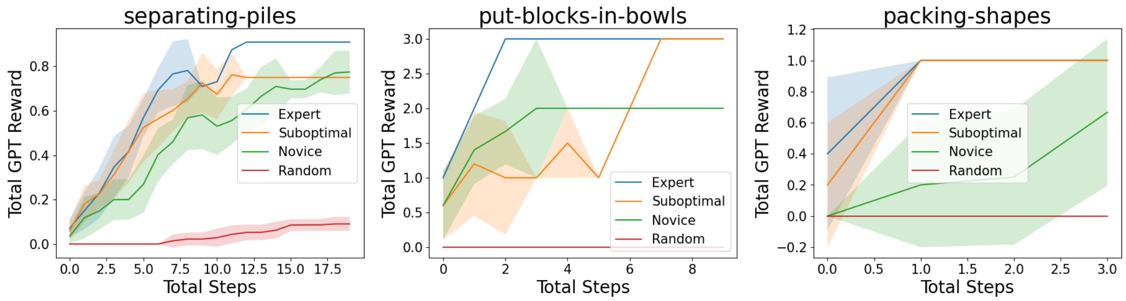

Figure 4. The inferred reward using VLM-CaR for 4 policies (random, novice, sub-optimal, expert) on different CLIPort environments. CLIPort is trained using imitation learning so we evaluate or reward function on 4 policies of varying skill levels. An action is taken randomly 30% of the time in the suboptimal policy and 50% of the time in the novice policy. The reward inferred by VLM-CaR well reflects the training process and performance of different policies.

for these experiments. Our experiments in Figure 4 indicate that VLM-CaR generated reward function is able to separate different policies on these different tasks. All the policies are clearly separable in separating-piles task. For tasks of put-blocks-bowls and packing-shapes, sub-optimal and expert policies are close as per VLM-CaR reward function but both are clearly separated from novice and random policies showing the effectiveness of reward functions.

6. Limitations

Throughout the utilization of our pipelines we found numerous limitations as well as some very interesting behaviours that suggest future strategies. Fundamentally, the image processing approaches suggested by the VLMs focus mainly on identification by colour and shape. Identifying objects that have very similar shapes and colours is extremely difficult given current models. For example, in MetaWorld (Yu et al., 2019) the robot is nearly an identical color of the objects it interacts with making the problem far more challenging. We find that color detection is sometimes necessary, even with complex shape detection procedures. Resolution also impacts detection algorithms. This work is not a suggestion that our method will work on every environment, only a suggestion that this is a successful direction for many types of RL tasks.

Shadows as well as the overlapping of objects, also create issues in shape, color and edge detection algorithms. We therefore prompt our VLM to store the locations of expected static objects from the initial image and to try and approximate shapes in contours as they may not be exact (or incomplete). We can do this by finding a convex hull of every contour which would always be closed and then approximating it down. In more complex tasks, we may have to show the VLM a full expert trajectory instead of the final and initial images to infer the tasks and rewards.

Our approach also requires manual assembly of the VLM provided scripts to create the final callable program, which we hope to eliminate as VLMs continue to scale. We hope as VLMs become more accurate in the future, these challenges will become easy to overcome. We also note that with access to a VLM with unlimited interactions and many iterations of our verification pipeline, it is likely that we could develop more complex strategies easier.

Despite these limitations, we see our method as an important step towards solving real-world tasks with natural image-based observations using reinforcement learning, empowered by VLM-CaR's code as reward capability.

7. Future Work and Conclusion

VLM-CaR is an intuitive approach to reward design in an easily accessible and tune-able manner. We showcase both automated task generation, automated reward function design and interpret-able reward functions in a wide variety of complex tasks. These rewards are found to be superior to environmental rewards in sparse reward tasks.

Given our prompt to identify objects using colour and shape, the VLM mainly resorts to these techniques. VLMs still develop intricate strategies that consider features of certain shapes and object size to identify objects. Future work should consider more advanced image processing techniques in prompts to further refine our method and enable use on a wider range of tasks. Utilizing off the shelf image detection tools in our pipeline is also an interesting direction.

During our experiments, we found the VLM to suggest implementing a pathfinding algorithm such as A* for finding a path to the goal state or key in MiniGrid. In one instance the VLM listed the correct sequence of steps given only the initial and goal image. We believe that our methods could also be used to generate progress models (Bruce et al., 2023; Mazoure et al., 2023) or generate affordances (Khetarpal et al., 2020a) as programs in future.

Impact Statement

The goal of the work presented in the paper is to advance the field of RL. There are many potential societal consequences of RL, but specific impacts are therefore difficult to highlight here.

Acknowledgements

We would like to thank Gheorghe Comanici for reviewing the draft of this work and providing critical feedback which helped improve the quality of manuscript significantly.

References

Ahn, M., Brohan, A., Brown, N., Chebotar, Y., Cortes, O., David, B., Finn, C., Gopalakrishnan, K., Hausman, K., Herzog, A., Ho, D., Hsu, J., Ibarz, J., Ichter, B., Irpan, A., Jang, E., Ruano, R. J., Jeffrey, K., Jesmonth, S., Joshi, N. J., Julian, R. C., Kalashnikov, D., Kuang, Y., Lee, K.-H., Levine, S., Lu, Y., Luu, L., Parada, C., Pastor, P., Quiambao, J., Rao, K., Rettinghouse, J., Reyes, D. M., Sermanet, P., Sievers, N., Tan, C., Toshev, A., Vanhoucke, V., Xia, F., Xiao, T., Xu, P., Xu, S., and Yan, M. Do as i can, not as i say: Grounding language in robotic affordances. In *Conference on Robot Learning*, 2022.

Alayrac, J.-B., Donahue, J., Luc, P., Miech, A., Barr, I., Hasson, Y., Lenc, K., Mensch, A., Millican, K., Reynolds, M., Ring, R., Rutherford, E., Cabi, S., Han, T., Gong, Z., Samangooei, S., Monteiro, M., Menick, J., Borgeaud, S., Brock, A., Nematzadeh, A., Sharifzadeh, S., Binkowski, M., Barreira, R., Vinyals, O., Zisserman, A., and Simonyan, K. Flamingo: a visual language model for few-shot learning, 2022.

Armstrong, S., Leike, J., Orseau, L., and Legg, S. Pitfalls of learning a reward function online. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, 2021.

Bacon, P.-L., Harb, J., and Precup, D. The option-critic architecture. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31, 2017.

Baumli, K., Baveja, S., Behbahani, F., Chan, H., Comanici, G., Flennerhag, S., Gazeau, M., Holsheimer, K., Horgan, D., Laskin, M., Lyle, C., Masoom, H., McKinney, K., Mnih, V., Neitz, A., Pardo, F., Parker-Holder, J., Quan, J., Rocktäschel, T., Sahni, H., Schaul, T., Schroecker, Y., Spencer, S., Steigerwald, R., Wang, L., and Zhang, L. Vision-language models as a source of rewards, 2023.

Bellman, R. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 1954.

Brohan, A., Brown, N., Carbajal, J., Chebotar, Y., Chen, X., Choromanski, K., Ding, T., Driess, D., Dubey, A., Finn, C., et al. Rt-2: Vision-language-action models transfer web knowledge to robotic control. *arXiv preprint arXiv:2307.15818*, 2023.

Bruce, J., Anand, A., Mazoure, B., and Fergus, R. Learning about progress from experts. In *The Eleventh International Conference on Learning Representations*, 2023.

Chen, Y.-C., Li, L., Yu, L., El Kholy, A., Ahmed, F., Gan, Z., Cheng, Y., and Liu, J. Uniter: Universal image-text representation learning. 2020.

Chevalier-Boisvert, M., Dai, B., Towers, M., de Lazcano, R., Willems, L., Lahlou, S., Pal, S., Castro, P. S., and Terry, J. Minigrid & miniworld: Modular & customizable reinforcement learning environments for goal-oriented tasks. *CoRR*, abs/2306.13831, 2023.

Cui, Y., Niekum, S., Gupta, A., Kumar, V., and Rajeswaran, A. Can foundation models perform zero-shot task specification for robot manipulation? In *Conference on Learning for Dynamics & Control*, 2022.

Devidze, R., Radanovic, G., Kamalaruban, P., and Singla, A. Explicable reward design for reinforcement learning agents. In *Advances in Neural Information Processing Systems*, 2021.

Ding, Z., Luo, H., Li, K., Yue, J., Huang, T., and Lu, Z. Clip4mc: An rl-friendly vision-language model for minecraft. *ArXiv*, abs/2303.10571, 2023.

Doya, K., Samejima, K., Katagiri, K.-i., and Kawato, M. Multiple model-based reinforcement learning. *Neural computation*, 14(6):1347–1369, 2002.

Du, Y., Konyushkova, K., Denil, M., Raju, A., Landon, J., Hill, F., de Freitas, N., and Cabi, S. Vision-language models as success detectors. In *Proceedings of The 2nd Conference on Lifelong Learning Agents*, 2023a.

Du, Y., Yang, M., Florence, P., Xia, F., Wahid, A., Ichter, B., Sermanet, P., Yu, T., Abbeel, P., Tenenbaum, J. B., et al. Video language planning. *arXiv preprint arXiv:2310.10625*, 2023b.

Dulac-Arnold, G., Levine, N., Mankowitz, D. J., Li, J., Paduraru, C., Gowal, S., and Hester, T. Challenges of real-world reinforcement learning: definitions, benchmarks and analysis. *Machine Learning*, 110(9):2419–2468, 2021.

Fan, L. J., Wang, G., Jiang, Y., Mandlekar, A., Yang, Y., Zhu, H., Tang, A., Huang, D.-A., Zhu, Y., and Anandkumar, A. Minedojo: Building open-ended embodied agents with internet-scale knowledge. *ArXiv*, abs/2206.08853, 2022.

Gallouédec, Q., Cazin, N., Dellandréa, E., and Chen, L. panda-gym: Open-Source Goal-Conditioned Environments for Robotic Learning. *4th Robot Learning Workshop: Self-Supervised and Lifelong Learning at NeurIPS*, 2021.

Harb, J., Bacon, P.-L., Klissarov, M., and Precup, D. When waiting is not an option : Learning options with a deliberation cost. *ArXiv*, abs/1709.04571, 2017.

Hare, J. Dealing with sparse rewards in reinforcement learning. *arXiv preprint arXiv:1910.09281*, 2019.

Huang, W., Xia, F., Xiao, T., Chan, H., Liang, J., Florence, P., Zeng, A., Tompson, J., Mordatch, I., Chebotar, Y., Sermanet, P., Brown, N., Jackson, T., Luu, L., Levine, S., Hausman, K., and Ichter, B. Inner monologue: Embodied reasoning through planning with language models. volume arXiv:2207.05608, 2022.

Kanervisto, A., Milani, S., Ramanauskas, K., Topin, N., Lin, Z., Li, J., yong Shi, J., Ye, D., Fu, Q., Yang, W., Hong, W., Huang, Z.-H., Chen, H., Zeng, G., Lin, Y., Micheli, V., Alonso, E., Fleuret, F., Nikulin, A., Belousov, Y., Svidchenko, O., and Shpilman, A. Minerl diamond 2021 competition: Overview, results, and lessons learned. *ArXiv*, abs/2202.10583, 2022.

Khetarpal, K., Ahmed, Z., Comanici, G., Abel, D., and Precup, D. What can I do here? A theory of affordances in reinforcement learning. In *Proceedings of the 37th International Conference on Machine Learning*, 2020a.

Khetarpal, K., Klissarov, M., Chevalier-Boisvert, M., Bacon, P.-L., and Precup, D. Options of interest: Temporal abstraction with interest functions. In *AAAI Conference on Artificial Intelligence*, 2020b.

Klissarov, M. and Precup, D. Flexible option learning. In *Advances in Neural Information Processing Systems*, 2021.

Klissarov, M., D'Oro, P., Sodhani, S., Raileanu, R., Bacon, P.-L., Vincent, P., Zhang, A., and Henaff, M. Motif: Intrinsic motivation from artificial intelligence feedback, 2023.

Kuznetsov, A., Shvechikov, P., Grishin, A., and Vetrov, D. Controlling overestimation bias with truncated mixture of continuous distributional quantile critics. In *Proceedings of the 37th International Conference on Machine Learning*, 2020.

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

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.

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.

Mahmoudieh, P., Pathak, D., and Darrell, T. Zero-shot reward specification via grounded natural language. In *Proceedings of the 39th International Conference on Machine Learning*, 2022.

Mazoure, B., Bruce, J., Precup, D., Fergus, R., and Anand, A. Accelerating exploration and representation learning with offline pre-training. 2023.

OpenAI, :, Achiam, J., and et al. Gpt-4 technical report, 2023.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., and Sutskever, I. Learning transferable visual models from natural language supervision, 2021.

Ratner, E., Hadfield-Menell, D., and Dragan, A. D. Simplifying reward design through divide-and-conquer, 2018.

Rocamonde, J., Montesinos, V., Nava, E., Perez, E., and Lindner, D. Vision-language models are zero-shot reward models for reinforcement learning, 2023.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms, 2017.

Shridhar, M., Manuelli, L., and Fox, D. Cliport: What and where pathways for robotic manipulation. In *Proceedings of the 5th Conference on Robot Learning (CoRL)*, 2021.

Sontakke, S. A., Zhang, J., Arnold, S. M. R., Pertsch, K., Biyik, E., Sadigh, D., Finn, C., and Itti, L. Roboclip: One demonstration is enough to learn robot policies. *ArXiv*, abs/2310.07899, 2023.

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

Sutton, R. S., Precup, D., and Singh, S. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 1999.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models, 2023.

Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.

Xiao, T., Chan, H., Sermanet, P., Wahid, A., Brohan, A., Hausman, K., Levine, S., and Tompson, J. Robotic skill acquisition via instruction augmentation with vision-language models. *ArXiv*, abs/2211.11736, 2022.

Yang, M., Du, Y., Ghasemipour, K., Tompson, J., Schuurmans, D., and Abbeel, P. Learning interactive real-world simulators. *arXiv preprint arXiv:2310.06114*, 2023.

Yu, T., Quillen, D., He, Z., Julian, R., Hausman, K., Finn, C., and Levine, S. Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning. In *Conference on Robot Learning (CoRL)*, 2019.

Yu, W., Gileadi, N., Fu, C., Kirmani, S., Lee, K.-H., Gonzalez Arenas, M., Lewis Chiang, H.-T., Erez, T., Hasenclever, L., Humplik, J., Ichter, B., Xiao, T., Xu, P., Zeng, A., Zhang, T., Heess, N., Sadigh, D., Tan, J., Tassa, Y., and Xia, F. Language to rewards for robotic skill synthesis. *Arxiv preprint arXiv:2306.08647*, 2023.

Zhu, D., Chen, J., Shen, X., Li, X., and Elhoseiny, M. Minigpt-4: Enhancing vision-language understanding with advanced large language models. *arXiv preprint arXiv:2304.10592*, 2023.

A. Task Completion Prompting Pipeline

- In this image, there is an agent that can move throughout the environment. There may also be other relevant objects that the agent can interact with. The agent is a {agent description}.
- (For robotic environments only:) What is the part of the agent that would most likely interact with objects? Give me only one object.
- Can you write a script to identify the agent object from the image? You should use the shape, edges, and color of the object to identify it as its possible other objects in the image may have the same color. This should return the location of the agent along with {True, False} if it is found. This script should not require input from me.

(Check verification pipeline)

Fail Response: Please try again and refine your approach. Please remember to identify objects using edges, shape and colour. Please examine the shape and colour of this object from the image again.

Fail Response (5 or more times): Please try again and refine your approach. Please try to simplify your approach by only checking color or simple shapes.

- This is the correct script to identify the agent. Please remember this script. I will refer to it as the "agent_ID_script".
- Can you give a list of the most important objects in this image? Do not give general objects like the walls or the grid. Give me a list of objects and concise names with description.
- I am now going to show you an image of the game when the final goal was completed.
- What do you think the final goal is? Give me only one goal.
- Now, from this image, can you infer n sequential tasks that the agent must do before it reaches the final goal. Give me the list of n tasks with descriptions of each task. This list should be concise and should not contain general behaviors like: navigate the maze, or avoid walls. The items you have identified in the image may help you come up with this list of tasks. The tasks should be actionable and concise and must complete the final goal.

Task completion check procedure:

- Now for Task 1, what is the most relevant object or objects in this task to reach or interact with. This item should not be the agent.
- Can you write a script to identify this object(s) from the image? You should use the shape, edges and color of the object to identify it as its possible other objects in the image may have the same color. You can only use the first image I gave you as input to this script. This script should return the location of the object(s) and True, False if it is found. Test by verifying the number of instances of the object found is correct. This script should not require input from me.

(Task completion pipeline)

Fail Response: Please try again and refine your approach. Please remember to identify objects using edges, shape and colour. Please examine the shape and colour of this object from the image again.

- This is the correct script to identify the Task 1 item. Please remember this script. I will refer to it as the "task_1_o1_ID_script".
- How would you know if Task 1 is done? Please propose one best guess of the check for completion. The type of check for completion you use should be implementable using a python script. Please describe this check. The script can only use the first image I gave you. You are allowed to compare this to the image from the initial frame.
- Are there any other objects that are absolutely essential to interact with or must be identified to check if this task has been completed. Only list absolutely essential objects.

(Identify objects using the object identification technique and label these object scripts accordingly)

- Please implement this technique, you will only have access to a single frame at a time. If you would like to access the initial state and compare it to the current state or store information from the initial state, this is allowed. Please use the same "task_1_o1_ID_script", ... as part of your implementation. The "agent_ID_script" may also help you. You can only use the first image I gave you as input to this script. This script must return {True, False}. It may be useful to store the location of expected static objects from the initial image. This script should not require input from me.

(Repeat task completion check procedure for n tasks.)

Goal completion check procedure:

- How would you know if the goal is done? Please propose one best guess of the check for completion. The type of check for completion you use should be implementable using a python script. Please describe this check. The script can only use the first image I gave you. You are allowed to compare this to the image from the initial frame. Can you please tell me first if you must identify any new objects for this task.
- Are there any objects that you have not identified so far that must be identified to check the completion of the goal.

(Identify objects using the object identification technique and label these object scripts accordingly)

- Please implement this technique, you will only have access to a single frame at a time. If you would like to access the initial state and compare it to the current state or store information from the initial state, this is allowed. Please use the same "task_1_o1_ID_script" or, ... {all other object ID scripts} ... as part of your implementation. The "agent_ID_script" may also help you. You can only use the first image I gave you as input to this script. This script must return {True, False}. It may be useful to store the location of expected static objects from the initial image. I will show you two images, the first image is of the initial state and the 2nd image is after the goal completion. You must return False on the first image and true on the 2nd. A few useful techniques: Use the contours to determine if an object is inside of another, don't approximate rectangles or radii. Try to approximate shapes in contours as they may not be exact (or incomplete). You can do this by finding a convex hull of every contour which would always be closed and then approximating it down. Checking the shape of objects is useful once you have identified them by color. This script should not require input from me.
- For robotic tasks: Could you also propose a real valued reward function that is incremental if possible?

Fail Response: Please try again and refine your approach.

B. Robotic Goal Prompting Pipeline

- In this image, there is an agent that can move throughout the environment. there may also be other relevant objects that the agent can interact with. The agent is a robotic arm.
- What is the part of the agent that would most likely interact with objects? Give me only one object.
- Can you give a list of the most important objects in this image? Do not give general objects like the walls or the grid. Give me a list of objects and concise names with description.
- I am now going to show you an image of the game when the final goal was completed.
- What do you think the final goal is? Give me only one goal.
- How would you know if the goal is done? Please propose one best guess of the check for completion. The type of check for completion you use should be implementable using a python script. Please describe this check. The script can only use the first image I gave you. You are allowed to compare this to the image from the initial frame.
- Give me the list of essential objects that must be identified to know if the goal has been completed. Do not include the agent in this list. Only give objects that are absolutely essential.

(Identify objects using the object identification technique and label these object scripts accordingly)

- Are there any objects that you have not identified so far that must be identified to check the completion of the goal.

(Identify objects using the object identification technique and label these object scripts accordingly)

- How would you know if the goal is done? Please propose one best guess of the check for completion. The type of check for completion you use should be implementable using a python script. Please describe this check. The script can only use the first image I gave you. You are allowed to compare this to the image from the initial frame. You can store the location of objects expected to be stationary.
- Please implement this technique, you will only have access to a single frame at a time. If you would like to access the initial state and compare it to the current state or store information from the initial state, this is allowed. Please use the same "task_1_o1_ID_script" or, ... {all other object ID scripts} ... as part of your implementation. You can only use the first image I gave you as input to this script. This script must return {True, False}. It may be useful to store the location of expected static objects from the initial image. I will show you two images, the first image is of the initial state and the 2nd image is after the goal completion. You must return False on the first image and true on the 2nd. A few useful techniques: Use the contours to determine if an object is inside of another, don't approximate rectangles or radii. Try to approximate shapes in contours as they may not be exact (or incomplete). You can do this by finding a convex hull of every contour which would always be closed and then approximating it down. Checking the shape of objects is useful once you have identified them by color. This script should not require input from me.
- Could you also propose a real valued reward function that is incremental if possible?

Fail Response: Please try again and refine your approach.

C. Response Examples

Only successful interactions are shown here. The approach may have required more iterations and fail responses.

We show the CLIPort put-blocks-in-bowls task here.

- User: In this image, there is an agent that can move throughout the environment. there may also be other relevant objects that the agent can interact with. The agent is a robotic arm.
- VLM: The image depicts a robotic arm positioned above a flat surface with various colored objects. There are green cylindrical containers, each with a different colored cube inside, except for one which appears to be empty. There are also red cubes outside the containers, suggesting that the task for the robot could be to sort these cubes into their matching colored containers. The robotic arm itself is equipped with a gripping tool that it can use to pick up and move the objects. This setup is typical in robotics and artificial intelligence research, where robots are trained to perform sorting tasks which can serve as benchmarks for their dexterity and decision-making capabilities.
- User: What is the part of the agent that would most likely interact with objects? Give me only one object.
- VLM: The part of the agent that would most likely interact with objects is the gripper at the end of the robotic arm.
- User: Can you give a list of the most important objects in this image? Do not give general objects like the walls or the grid. Give me a list of objects and concise names with description.
- VLM: Certainly! Here's a list of the most important objects in the image:
  1. Robotic Arm: A multi-jointed mechanical arm with a gripper, positioned over the surface.
  2. Green Containers: Cylindrical objects with open tops, designed to hold cubes.
  3. Red Cubes: Solid red-colored blocks, some placed inside the containers, some on the surface.
  4. Blue Cube: A single solid blue-colored block, placed inside one of the green containers.
  5. Orange Cube: A single solid orange-colored block, placed inside one of the green containers.

6. Yellow Cube: A single solid yellow-colored block, placed inside one of the green containers.
7. Purple Cube: A single solid purple-colored block, placed inside one of the green containers.
8. Empty Green Container: One of the cylindrical green containers without a cube inside it.

- User: I am now going to show you an image of the game when the final goal was completed.

- VLM: The final goal appears to be placing each colored cube into the matching colored container. Based on the setup, the specific goal completed would be: Place the red cubes into the green containers.

- User: How would you know if the goal is done? Please propose one best guess of the check for completion. The type of check for completion you use should be implementable using a python script. Please describe this check. The script can only use the first image I gave you. You are allowed to compare this to the image from the initial frame.

- VLM: To check if the goal is completed using a Python script, you would implement an image processing solution to compare the initial state of the game (as depicted in the first image) with the final state (as shown in the second image). Here's a proposed method for the check:

  Color Detection and Position Mapping: Use a color detection algorithm to identify the colors of the cubes and the containers. You would map the position of each cube relative to the containers.

  State Comparison:

  Initial State: Detect all cubes and their initial positions outside of the containers. Final State: Detect all cubes and confirm their positions inside the containers.

  Check for Completion:

  Create a mapping of cube color to container color. Verify that each cube is inside a container with a matching color. Here is a high-level description of what the Python script would do:

  1. Load the images: Use a library like OpenCV to load the initial and final state images.
  2. Preprocess images: Convert the images to a format suitable for color detection (e.g., HSV color space).
  3. Detect and label objects: Use color range thresholds to detect cubes and containers, and label them based on color.
  4. Determine cube positions: Calculate the centroid of each cube and container to determine their positions.
  5. Check for red cube placement: Specifically for the red cubes, check if their centroids in the final image lie within the boundaries of the green containers.
  6. Result: If all red cubes are within the green containers, the check returns True, indicating the goal is achieved. Implementing this in a script would involve quite a bit of detail, especially in terms of handling variations in lighting and perspective, but the above method provides a strong basis for such a script.

- User: Give me the list of essential objects that must be identified to know if the goal has been completed. Do not include the agent in this list. Only give objects that are absolutely essential.

- VLM: To determine if the goal has been completed, the essential objects that must be identified are:

  Red Cubes: To check if all the red cubes are placed inside the green containers. Green Containers: To confirm that the red cubes are within their bounds.

- User: Can you write a script to identify this object(s) "Number 1" from the image? You should use the shape, edges and color of the object to identify it as its possible other objects in the image may have the same color. You can only use the first image I gave you as input to this script. This script should return the location of the object(s) and True, False if it is found. Test by verifying the number of instances of the object found is correct. This script should not require input from me.

- VLM: The revised script detected two red cubes in the image, and their locations are highlighted with green rectangles. The detection may still not be perfect, but this result shows where the algorithm is currently identifying red cubes. If there's still a discrepancy, further refinement would be needed.

- User: This is the correct script to identify the item. Please remember this script. I will refer to it as "o1_ID_script".

- VLM: Understood. The script used in the last execution to identify red cubes in the image will be referred to as "o1_ID_script" for future reference. If you need to use or refer back to this script, just mention its name, and I will know to use the corresponding code.

- User: Can you write a script to identify this object(s) "Number 2" from the image? You should use the shape, edges and color of the object to identify it as its possible other objects in the image may have the same color. You can only use the first image I gave you as input to this script. This script should return the location of the object(s) and True, False if it is found. Test by verifying the number of instances of the object found is correct. This script should not require input from me.

- VLM: The refined script has successfully identified two objects that could be the green containers in the image. Here are their locations:

  Green Container 1: Located at (x: 110, y: 356), with a width of 69 pixels and a height of 64 pixels. Green Container 2: Located at (x: 325, y: 290), with a width of 59 pixels and a height of 52 pixels. The script found two instances, which likely correspond to the green containers on the table.

- User: This is the correct script to identify the item. Please remember this script. I will refer to it as "o2_ID_script".

- VLM: I will remember the script used to identify the green containers as "o2_ID_script". If you need to refer back to it or use it in the future, just mention its name, and I will know which code you are referring to.

- User: Are there any objects that you have not identified so far that must be identified to check the completion of the goal.

- VLM: Identify the red cubes using the "o1_ID_script". Identify the green containers using the "o2_ID_script". Once both sets of objects are identified, the check for completion would involve verifying that each red cube is within the bounds of a green container. This can be determined by checking if the centroid of each red cube's bounding rectangle falls within the bounding rectangle of any green container. If all red cubes satisfy this condition, the goal is achieved.

- User: How would you know if the goal is done? Please propose one best guess of the check for completion. The type of check for completion you use should be implementable using a python script. Please describe this check. The script can only use the first image I gave you. You are allowed to compare this to the image from the initial frame. You can store the location of objects expected to be stationary.

- VLM: To determine if the goal is completed, a Python script can perform the following check for completion:

  Check for Completion:

  Identify Red Cubes: Use "o1_ID_script" to locate all red cubes in the current frame. Identify Green Containers: Use "o2_ID_script" to locate all green containers in the current frame. Compare Positions: For each identified red cube, the script should calculate the centroid of its bounding rectangle. Then, for each green container, the script should check if any red cube's centroid is within the bounds of the container's bounding rectangle. Completion Condition: If all red cube centroids are inside the bounds of any green container, then the goal is considered to be achieved. The Python script would execute these steps and return True if all red cubes are inside the green containers, and False otherwise.

  Here is a high-level description of what the Python script would do:

  Load the initial and current images. Apply "o1_ID_script" and "o2_ID_script" to identify red cubes and green containers in both images. Calculate the centroids of the red cubes. Determine the bounding rectangles of the green containers. Check if each red cube's centroid from the current frame is inside a green container. Compare the initial and current positions of the red cubes to confirm they have been moved into the containers. The script returns True if all red cubes are inside the green containers, False otherwise. This script assumes that the initial and current images are provided and that the cubes and containers are distinguishable by their colors and shapes. The "o1_ID_script" and "o2_ID_script" provide the necessary object identification for this check.

- User: Please implement this technique, you will only have access to a single frame at a time. If you would like to access the initial state and compare it to the current state or store information from the initial state, this is allowed. Please use the same "o1_ID_script" or "o2_ID_script" as part of your implementation. You can only use the first image I gave you as input to this script. This script must return {True, False}. It may be useful to store the location of expected static objects from the initial image. I will show you two images, the first image is of the initial state and the 2nd image is after the goal completion. You must return False on the first image and true on the 2nd. A few useful techniques: Use the contours to determine if an object is inside of another, don't approximate rectangles or radii. Try to approximate shapes in contours as they may not be exact (or incomplete). You can do this by finding a convex hull of every contour which would always be closed and then approximating it down. Checking the shape of objects is useful once you have identified them by color. This script should not require input from me.

- VLM: The script correctly identified that the goal has not been completed in the initial state image, returning False. This script can now be applied to a second image, which represents the state after the goal completion, to verify if it returns True as expected for the completed state. Please provide the second image to proceed with the verification.
- User: Provide Final Image
- VLM: The script has analyzed the final state image and determined that the goal of placing red cubes into green containers has been completed, returning True. This indicates that the conditions for the goal have been met in the provided image.
- User: Can you also propose a real valued reward function that is incremental if possible?
- VLM: Certainly, in reinforcement learning and many task completion scenarios, a reward function provides incremental feedback to an agent based on its actions. For the task of placing red cubes into green containers, an incremental reward function could be designed to provide positive feedback as the agent makes progress towards the goal, and possibly negative feedback for incorrect actions.

  Here's a proposal for a real-valued reward function:

  Closeness to Container Reward:

  For each red cube, calculate the Euclidean distance to the nearest green container.

  Normalize this distance by the diagonal length of the image to get a value between 0 and 1.

  Invert the normalized distance (1 - normalized distance) to get a reward value where closer cubes have higher rewards. Sum these values for all red cubes.

  Correct Placement Reward:

  Assign a fixed reward value for each red cube that is correctly placed inside a green container.

  This value should be significant to reflect the completion of the task for each cube.

  Penalty for Incorrect Placement:

  If a red cube is placed outside any green container, apply a penalty. This could be a negative value or the subtraction of a potential reward. Completion Bonus:

  If all red cubes are inside green containers, provide a large completion bonus.

Example failed response:

It appears that the script did not identify any red cubes in the image. This could be due to several reasons such as the color range for red not being accurately defined for the lighting conditions in the image, or the size threshold for detecting cubes not being appropriate.

To resolve this, we would need to fine-tune the color detection thresholds and possibly adjust the size criteria for identifying the cubes. Unfortunately, due to the constraints of this platform, I can't visually inspect the output to make these adjustments, but in a typical development environment, you would iterate on the color range and size parameters while visually inspecting the output until the red cubes are correctly identified

D. Verification Pipeline Notes

We first note that $p = 0.1$ in all experiments.

We wish to emphasize the usefulness of the verification pipeline given a few observed examples.

In one suggested completion check for obtaining the key task during the MiniGrid experiments, the VLM used the agent proximity, with a Euclidean distance threshold, to the key as an indication that the reward function. The agent came into

Table 2. The hyper-parameters used for PPO in the the MiniGrid experiments

PARAMETER	VALUE
ADAM OPTIMIZER α	$7 \cdot 10^{-4}$
$\beta_1; \beta_2$	$0.9; 0.999$
ϵ	10^{-5}
ENTROPY COEFFICIENT	10^{-2}
VALUE LOSS COEFFICIENT	0.5
γ	0.99
MAXIMUM NORM OF GRADIENT	0.5
TIME-STEPS	$1e6$
BATCH SIZE	256
ENTROPY COEFFICIENT	10^{-2}
CLIP PARAMETER	0.2

Table 3. The hyper-parameters used for TQC in the the PandasGym experiments

PARAMETER	VALUE
BUFFER SIZE	$1e6$
BATCH SIZE	2048
γ	0.95
α	0.001
TIME-STEPS	$100,000$
POLICY NET	$[512, 512, 512]$
n-CRITICS	2
REPLAY BUFFER	HER BUFFER
GOAL SELECTION STRATEGY	FUTURE
n-SAMPLED GOALS	4
τ	0.05

close enough proximity to the key in 43% of the random trajectories, which was greater than our 10% threshold. This was obviously not a useful check for completion and resulted in a more fine-tuned check with refinement. The final completion check was to verify that the key had disappeared from the image (and was obtained by the agent.

In another example the agent identified the blue blocks incorrectly in the separating-piles task. All the blue blocks were identified as a single contour with the center calculated incorrectly. This center was found to be inside the yellow square over 10% of the time and therefore the script was not verified.

We also observed another interesting caveat of our verification pipeline, which we discuss. In CLIPort it is possible that the VLM only identifies one red block correctly. We did not see this behaviour in our experiments but we wish to note it as a potential problem in other environments. If one red object is identified, the entire verification pipeline will succeed given the programs are correct in checking the placement into the green bowl. We therefore suggest when using our method that we have an additional completion check. The final goal should be obtained within x frames of the trajectory completion to prevent this scenario from happening.

E. Experiment Parameters

We list parameters for each of the environment experiments in Tables 2 and 3

F. Environment Design

We make changes to our environments for more efficient use with our pipeline.

MiniGrid: All doors and keys are set to yellow. The goal is always a green square and agent is always a red arrow. The reward is 1 if the task is completed and 0 otherwise.

CLIPort: The blocks, yellow square, and bowls are always the same colour.

Pandas-Gym: The block is always green, the goal area is always yellow. The camera is placed at the front of the table looking downwards at a 45 degree angle towards the robot and objects on the table. The rewards are always sparse, not dense.

G. Manual Intervention Details

Below contains the details of all manual steps required for VLM-CaR.

1. We copy and paste the script given to us by the GPT model for each object identification as is.

2. We copy and paste each task completion check script from the GPT model as is. These scripts, in some cases already contain the object identification script as part of this function.

3. For robotic experiments: We copy and paste the reward function script from the GPT model as is.

4. If GPT did not define each script as a separate function, we define a function to encompass that script and name each variable required as input to the function in this definition. (We later found that we can prompt GPT to do this and name the function in a specific way, but we did not change this to keep the pipeline consistent across all experiments).

5. In the next few steps, we arrange these reward functions in a nice class structure with the environment. Although we define the class structure for better code organization and add syntax changes for that, it is not necessary as such and the main function to be used in the training loop is a single function to be used as reward.

   (a) GPT is not prompted to define a class blueprint for the scripts given by it. We define a class for each environment and define an "init" function along with a reset function. In the "init" function, we define the following variables.
      (i) A boolean variable for each task, initially set to false indicating that the task was not completed. For example: `self.task_1_complete=False`.
      (ii) A boolean variable set to True indicating that we are examining the first image in a trajectory. For example: self.first_frame = True and `self.init_image=None`.

   (b) We define a reset function and we set all of these variables defined in the init function equal to the default values.

   (c) We add the term "self" as the first input to any functions given to us by GPT. We also add "`self.`" to each function call so they function co

   (d) If any function given by GPT accepts an image path rather than the actual image, we change it to accept the actual image.

   (e) We define a function termed "`reward(self, image_path)`". We then first read the image from the image path for use with the GPT generated functions by calling "`image=cv2.imread(image_path)`". If any GPT script requires the initial image for later comparison, we check the variable `self.first_frame`, if it is True we store the image as `self.init_image` and then set `self.first_image=False`. We detail below how we assemble this reward function:

   (i) For MiniGrid: We sequentially check that each goal (using a series of if statements) is completed using the GPT defined functions, making each of the task completed variables defined in the init function as True if the task is completed. If the task is completed, we move onto calling the next task completion check script in this case. For the output of our reward function, we give a reward of $1/n$ at the completion of each task, where n is the number of tasks.

   (ii) For Robotic Experiments: We call the GPT defined reward function, we give the reward from this function as output.

6. The output from the reward function is used in the training loop as the training reward, not the evaluation reward.

H. Generated Scripts

All generated scripts are provided. We name the functions with the corresponding environment name. GPT-4 generated comments were removed.

These scripts along with the verification pipeline code are available at: https://github.com/dvVenuto/vlm-car