CRACKING THE CODE OF ACTION: A GENERATIVE APPROACH TO AFFORDANCES FOR REINFORCEMENT LEARNING

Lynn Cherif [*1,2], **Flemming Kondrup** [*1,2], **David Venuto** [1,2],

Ankit Anand [1,2,3], **Doina Precup** [1,2,3], **Khimya Khetarpal** [2,3]

[1] McGill University, [2] Mila, [3] Google DeepMind

lynn.cherif@mail.mcgill.ca

ABSTRACT

Agents that can autonomously navigate the web through a graphical user interface (GUI) using a unified action space (e.g., mouse and keyboard actions) can require very large amounts of domain-specific expert demonstrations to achieve good performance. Low sample efficiency is often exacerbated in sparse-reward and large-action-space environments, such as a web GUI, where only a few actions are relevant in any given situation. In this work, we consider the low-data regime, with limited or no access to expert behavior. To enable sample-efficient learning, we explore the effect of constraining the action space through *intent-based affordances* – i.e., considering in any situation only the subset of actions that achieve a desired outcome. We propose **Code as Generative Affordances** (**CoGA**), a method that leverages pre-trained vision-language models (VLMs) to generate code that determines affordable actions through implicit intent-completion functions and using a fully-automated program generation and verification pipeline. These programs are then used in-the-loop of a reinforcement learning agent to return a set of affordances given a pixel observation. By greatly reducing the number of actions that an agent must consider, we demonstrate on a wide range of tasks in the MiniWob++ benchmark that: **1)** CoGA is orders of magnitude more sample efficient than its RL agent, **2)** CoGA's programs can generalize within a family of tasks, and **3)** CoGA performs better or on par compared with behavior cloning when a small number of expert demonstrations is available.

1 INTRODUCTION

Reinforcement learning (RL) is a powerful paradigm to train agents for sequential decision-making by interacting with an environment. In environments where data collection and human annotation is time-consuming and costly, the sample efficiency of an agent is critical. Despite its great potential and success in multiple domains like Go (Silver et al., 2016) and Chess (Silver et al., 2017), RL algorithms can suffer from significant challenges in being sample efficient. In real-world environments with sparse reward and large action spaces where only a small subset of actions are relevant in a given situation (e.g., GUI-based web navigation, recommendation systems), this issue is exacerbated.

To address this challenge, a popular approach is to leverage expert trajectories with behavior cloning (BC) (e.g., Shaw et al. (2023)). State-of-the-art methods require thousands to millions of such demonstrations. However, this comes with major limitations including computational costs and the burden of gathering domain-specific expert demonstrations. Moreover, BC suffers from an imitation gap (Ross & Bagnell, 2010) and rarely surpasses its training data. In contrast, RL has the possibility

[*]Equal contribution

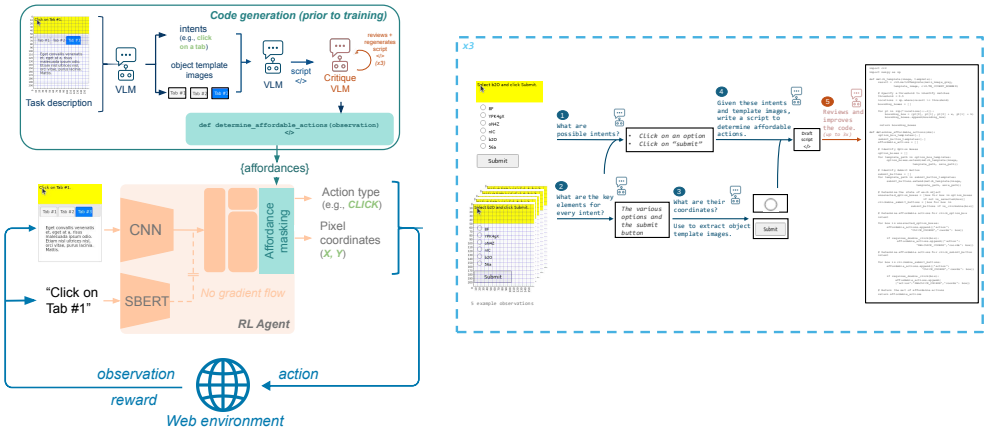

Figure 1: **Left:** Overview of our method, CoGA. The VLM processes available task descriptions and example observations to extract relevant intents (e.g., "*click on a tab*") and object template images (e.g., every tab), which are then used to generate code that returns a set of affordable actions given an observation. The code is validated and improved by a critique VLM. The set of affordances are then used to mask the action space of the RL agent. **Right:** Prompting pipeline to generate affordance scripts that return the set of affordable actions.

to gather new data and learn from interaction. Yet, RL methods alone struggle to bridge the gap to expert performance in many tasks, particularly those with large action spaces and sparse rewards. In this regard, we here focus on reducing the complexity of the action space, and by consequence alleviating the sparse reward challenge. Progress towards improving the performance and sample efficiency of RL agents is complimentary to methods such as (Shaw et al., 2023), with potential to further enhance the RL fine-tuning component.

In the context of RL, Khetarpal et al. (2020) defined *affordances* (Gibson, 1977) as actions that complete intended consequences (i.e., *intents*). Intent-based affordances help prune the action space and guide RL agents towards effective actions. This mitigates naive exploration and reduces sample complexity. To bridge the gap towards or beyond expert performance in the *low-data regime*, we focus on *learning with limited to no access to expert demonstrations* and investigate the use of affordances to improve the sample efficiency of RL.

However, the specification of intents and intent-completion functions is non-trivial and remains an open problem. For instance, hand-designing them can be limited in environments where intents are not obvious and require substantial effort and domain knowledge. We address this challenge by leveraging pre-trained large vision-language models (VLMs) to discover intents and the corresponding actions they afford.

Given their multimodal reasoning capabilities, pre-trained VLMs are well-suited to enhance RL agents operating with image-based observations. Although querying a VLM directly for affordances based on visual observations (Qian et al., 2024) or making the VLM itself be the agent is possible, it is computationally and financially expensive. We use VLMs to generate functions that return affordable actions through implicit intent-completion, requiring a robust code generation and verification pipeline. While prior work (Venuto et al., 2024) focused on high-level tasks like sub-task reward functions, our prompting and verification approach ensures reliable low-level affordance specification. The generated code is used in the RL training and inference loop for pruning the action space, thus improving the agent's sample efficiency.

Our framework, **Code as Generative Affordances (CoGA)**, demonstrates strong sample efficiency and success rates on a series of MiniWob++ (Shi et al., 2017; Liu et al., 2018) tasks. While we study CoGA's efficacy in the challenging domain of automated visual web navigation, the core contribution of our work lies in the methodological framework of generating intent-based affordances as code to enhance the sample efficiency of RL agents. Specifically, we show the following claims (Sec. 4.3):

1. **CoGA is orders of magnitude more sample efficient than its RL agent.**
2. **CoGA's generated affordance scripts can generalize within the same family of tasks.**

3. **CoGA performs better or on par compared to its reference BC agent when only a limited number of expert demonstrations are available.**

2 BACKGROUND

Reinforcement Learning. An RL agent learns to interact with an environment, through a sequence of actions, in order to maximize its expected long-term reward (Sutton & Barto, 2018). This interaction is typically formalized using the framework of Markov Decision Processes (MDPs). A finite MDP is a tuple $M = \langle \mathcal{S}, \mathcal{A}, r, P, \gamma \rangle$, where $\mathcal{S}$ is a finite set of states, $\mathcal{A}$ is a finite set of actions, $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, $P : \mathcal{S} \times \mathcal{A} \to \text{Dist}(\mathcal{S})$ is the environment's transition dynamics, mapping state-action pairs to a probability distribution over next states, $\text{Dist}(\mathcal{S})$, and $\gamma \in (0, 1)$ is a discount factor. At each time step t, the agent observes a state $s_t \in \mathcal{S}$ and takes an action $a_t \in \mathcal{A}$ drawn from a policy $\pi : \mathcal{S} \to \text{Dist}(\mathcal{A})$.

Q-learning. A value function of a policy π is defined as the expectation of long-term return (i.e., the cumulative discounted reward) received from a given state and by executing π, defined as: $V^\pi(s) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) | s_0 = s]$ where $a_t \sim \pi(\cdot|s_t)$, for all t. The action-value function is defined as $Q^\pi(s, a) = r(s, a) + \gamma \sum_{s'} P(s'|s, a) V^\pi(s')$. In Q-learning Watkins & Dayan (1992), the optimal action-value function Q^* corresponds to the optimal policy π^*: $Q^* = \max_\pi Q^\pi(s, a)$. The optimal policy π^* can be obtained by acting greedily with respect to Q^*. In complex environments, the optimal Q-value function can be approximated using a neural network, referred to as Deep Q-learning (DQN) Mnih et al. (2013).

Vision-Language Models. VLMs are pre-trained transformers that integrate visual and textual data as input, enabling multi-modal reasoning. These models consist of three core components: a vision module to process images, a text module for language inputs, and a fusion mechanism —often utilizing cross-modal attention— to link visual and textual embeddings. Models like CLIP (Radford et al., 2021) and UNITER (Chen et al., 2020) have demonstrated impressive performance in tasks such as scene description and image-text matching. These models leverage contrastive learning or transformer-based architectures to align images and text in a shared embedding space. More recent, larger models such as GPT-4 (OpenAI et al., 2023) showcase exceptional performance across a range of tasks with increasing complexity.

Intents, Intent Completion, and Affordances. The concept of *intent* refers to the desired outcome associated with an action (Gibson, 1977). Intents are abstract representations of goal states, guiding an agent's decision-making. In RL, *affordances* are the state-action pairs that can *complete* these intents, effectively reducing the action space by focusing only on relevant actions in a given state (Khetarpal et al., 2020). Concretely, an *intent-completion function* considers a transition (s_t, a_t, s_{t+1}) and set of intents, and predicts the likelihood of the transition to complete the respective intents above a certain threshold. Thus, affordances can be inferred through the intent-completion function. In this work, we posit that a VLM can predict the likelihood of achieving an intended consequence for a given state and an action, which can lead to relevant affordances. Specifically, we leverage the VLM to 1) specify the relevant intents for a task (e.g., "click tab"), 2) infer implicit intent-completion functions by iteratively building its understanding of the task, and 3) generate affordances as code by implicitly using its inferred intent-completion functions (see Sec. 3).

3 CoGA: CODE AS GENERATIVE AFFORDANCES

We now present our approach, CoGA, which leverages pre-trained VLMs to generate code for determining affordable actions given an image observation (see `determine_affordable_actions(obs)` in Figure 1 - right). These generated functions return a set of affordable actions which can be used to prune the action space in RL. This task not only requires high-level reasoning by the VLM, but also correctly inferring and detecting the low-level affordable actions (i.e., affordable action types and pixel coordinates here) in each and every observation through the generated code.

3.1 GENERATING AFFORDANCES AS CODE.

CoGA proposes a modular prompting pipeline (see Figure 1 - right and Appendix A) which builds on that of Venuto et al. (2024). CoGA consists of three key components: **1)** a modular code generation pipeline that first identifies the correct set of intents and relevant objects given a task description and an image observation, and then generates a function that returns the set of affordable actions given an observation, **2)** a verification pipeline that leverages another VLM for critiquing the generated code and improving it accordingly, the final code (to be used in RL) is selected based on ground truth test cases, **3)** using the generated code resulting from steps (1) and (2) in RL. This pipeline alleviates the need for expensive VLM inference calls during the RL stage. As shown in Figure 1 (right), inferring the relevant high-level intents in a task (e.g., "click on an option", "click submit") is the first step in generating functions to determine affordances. For every intent, the VLM (GPT-4o) is prompted to determine the relevant objects (e.g., every tab) and their bounding box coordinates. These objects' bounding boxes thereby correspond to the affordable pixel actions and need to be dynamically detected for every observation through the generated script. To do so, we aim to use off-the-shelf object detection methods that do not require additional training, as in Venuto et al. (2024). However, unlike Venuto et al. (2024), we resort to template image matching. We found it more robust for detecting complex and granular objects (e.g., cartoon trash cans) than edge and color detection methods used by their generated reward functions.

Inferring Intents. The process starts by prompting the VLM to build context about the environment. We first show it a randomly sampled observation, the task description and example instructions of the task given by the environment. We then ask it to identify the salient objects in the image, followed by the relevant intents for the *task type*. It is important to distinguish between intents and goals, as a goal is related to directly solving the task by completing a given instruction (e.g., "click on Tab 1"), whereas an intent is related to solving the *type* of task more broadly (e.g., "click on a tab").

Detecting Visual Affordances. Once the intents are discovered, the VLM is prompted to name the relevant objects to each intent. For every named object, we follow an automated template image extraction process. The VLM is shown a coordinate-system-based gridded image and is required to specify the bounding box coordinates of the respective objects. The prompting pipeline then queries pre-written code to crop and save the objects' image templates using their determined bounding box coordinates. The saved template images are then used in a pre-written template matching script using OpenCV. The template images are derived from 5 randomly sampled observations to maximize generalization across the observation space. Additionally, to avoid discrepancies in color (e.g., a circle is always a circle regardless of its color), we perform template image matching between the grayscaled template images and observation images.

Determining Affordable Actions. Once the affordable objects are detected for every intent, the VLM is asked to develop 4 strategies in sequence using chain-of-thought prompting (Wei et al., 2023), before writing the function that returns the affordable actions for a given observation: 1) a step-by-step strategy for determining which intent(s) are relevant for a given observation, 2) a step-by-step strategy for determining which actions are affordable for a given observation and for each intent, 3) a step-by-step strategy for combining the intents and their corresponding set of affordable actions for a given observation, 4) an outline of the script to determine affordances using code comments. Finally, the VLM is asked to write code to implement its strategy for selecting the applicable intents and their corresponding set of affordable actions, for a given observation, by filling in the outline of comments with code. As such, the VLM uses the extracted template images and template matching script as needed to detect the affordable pixel actions (e.g., tabs). The generated code returns the set of affordable actions (both action types and their corresponding pixel actions) for a given observation.

3.2 VERIFICATION PIPELINE

For every task, we automatically verify the generated scripts using a) a critique VLM and b) ground truth test cases. The ground truth test cases consist of 5 randomly sampled observations that we manually annotate with a set of affordable actions. Firstly, the scripts are automatically executed on

the 5 randomly sampled test observations to check for execution errors. In the case of any thrown execution errors, the critique VLM reviews them and provides improvement feedback similar to Wang et al. (2023). Else, the scripts' precision and recall are calculated with respect to the manually annotated ground truth set of affordances. As well, the critique VLM is shown 2 of the 5 randomly sampled test observations as a reference for assessing the quality of the scripts and provides feedback accordingly. The feedback is reused by the critique VLM to improve and regenerate the code. This process repeats up to 3 times (we have not empirically observed gains beyond this number), unless the critique VLM approves the code earlier. Note that the critique VLM does not have access to the ground truth affordances of the test cases. We log the mean precision and recall over the 5 manual test cases and run the pipeline a maximum of 3 times. We retain the best performing scripts across pipeline runs and critique iterations.

3.3 Using the Generated Affordance Script in RL

The generated affordance script is queried in the training and inference loops of the RL agent to obtain the set of affordable actions. As shown in Figure 1 (left), the predicted set of affordable actions is used to create a hard mask over unaffordable actions (i.e., the probability of sampling unaffordable actions is 0). Thus, it is important to highlight that the success of `CoGA` strongly depends on the quality of the generated script, which in turn depends on the accuracy of the object detection method used. If the predicted affordances have low recall, `CoGA` would fail. In such a case, using soft masking during training where unaffordable actions are assigned low probability would allow `CoGA` to slowly catch up to the RL baseline, ultimately lagging in sample efficiency. This limitation is further discussed in Section 6.1. It is worth noting that the generated affordances can be used in either value-based or policy gradient RL.

4 Experiments

4.1 MiniWoB++

MiniWoB++ (Shi et al., 2017; Liu et al., 2018) consists of a collection of web-based graphical user interface (GUI) tasks, where the goal is to complete tasks by interacting with a simulated webpage. The tasks vary in complexity, ranging from simple actions like clicking a button, to more complex ones like completing a form, or navigating through a series of web elements. Each task is defined by an HTML structure, and the agent's observation consists of a rendered screenshot of the webpage. We use the MiniWob++ environment and action space defined in Shaw et al. (2023). The action space consists of action types (e.g., `click`, `begin_drag`) and (x, y) pixel coordinates. The affordances are on both action types and pixel coordinates. Every pixel observation is 160x210 pixels, which we divide into 32 bins as in Shaw et al. (2023). We discard text-entry tasks, and therefore the `type` and keyboard actions. Concretely, this results in an action space of 4x1024.

The rewards are defined in $(-1, 1)$. Positive rewards (success of 1) are assigned *only* upon successfully completing the task, and negative rewards (success of 0) are assigned otherwise (i.e., sparse rewards). Similar to previous works, to encourage the agent to complete the task faster, we discount the positive rewards by the number of steps taken to complete the task.

4.2 Methods.

Reinforcement Learning (RL) Agent. We use a DQN agent that is built on a convolutional neural network (CNN) backbone for encoding the pixel observations. Additionally, for every observation, we encode the task instruction using Sentence-BERT (SBERT) (Reimers & Gurevych, 2019) as shown in Figure 1 (left). Although our method is applicable to both value- and policy-based algorithms, we choose a DQN agent due to improved stability and learning efficiency observed in initial experiments. To further enhance learning stability and efficiency, we specifically use a double DQN (van Hasselt et al., 2015) and prioritized experience replay (Schaul et al., 2016). All the following baselines use the same architecture as the RL agent. See Appendix C for hyperparameter details.

`CoGA`. During training and inference, the generated affordance script is queried. The returned set of affordable actions are used to mask the action space. The agent can only sample from the

affordable actions. Note that we also apply the affordance masks during bootstrapping from o_{t+1}, where o denotes the observation.

Behavioral Cloning (BC) Agent. Due to limited resources and closed-source expert demonstrations performed on the environment we used, we perform behavioral cloning on expert demonstrations that we collected using rollouts from the Pix2Act model (Shaw et al., 2023). We filter the trajectories that achieved a reward of less than 0.8.

It should be noted that while many state-of-the-art prior works (e.g., (Shaw et al., 2023)) on MiniWob++ use large amounts of expert demonstrations, our work focuses on leveraging pre-trained foundation models particularly in the low-data regime with no or limited access to expert demonstrations. Hence, the comparison with BC is limited to scenarios that only use a few expert demonstrations.

4.3 RESULTS

CoGA's Affordance Scripts are Intuitive, (mostly) Accurate, and Precise. We evaluate the quality of the generated affordance scripts qualitatively and quantitatively. Qualitatevely, we observe that the returned affordable actions are intuitive (Figure 2 left). This is emphasized in instruction-dependent tasks such as `click-test-2` and `click-tab` (Figure 2 left - middle and right). In `click-test-2` the instruction (e.g., *goal*) is to either click on button `ONE` or `TWO`. However, the *intent* is to "*click a button*", in which case clicking any of the two buttons is affordable, and the policy is learnt over these affordances.

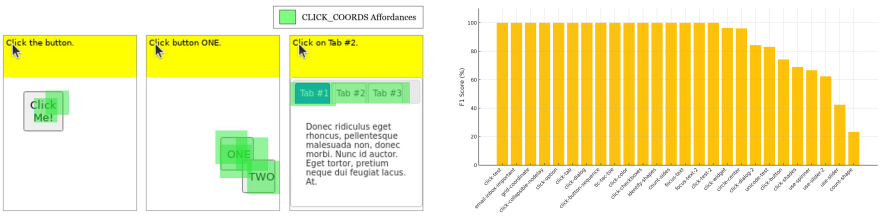

Figure 2: **Left:** Examples of returned affordances for three tasks (left to right): `click-test`, `click-test-2`, `click-tab` **Right:** F1-score across tested tasks. We observe that most generated affordance scripts have a high F1-score, implying wide and precise coverage of ground truth affordances.

Quantitavely, we measure the quality of the generated affordance scripts using precision and recall, and aggregate them through an F1-score (Figure 2 right). We define precision as the rate of predicted affordances that have at least one match in the set of ground truth affordances (i.e., identical action type and a corresponding pixel intersection over union (IoU)$>$0). We define recall as the rate of ground truth affordances that have at least one match in the set of predicted affordances. As seen in Figure 2 (right), most scripts have high recall and precision. For those with low precision but high recall, the affordance set would include more affordable actions than in the ground truth. To this end, in the worst case, `CoGA` performs on par to the RL agent. See Appendix F for the details of the code generations runs.

CoGA is Orders of Magnitude More Sample Efficient than its RL Agent. We investigate the effect of constraining the action space using the affordances returned by the generated affordance scripts on the agent's sample efficiency. Following a hyperparameter search (see Appendix C), we report the best evaluation success rates at 1000 steps for the RL agent and `CoGA` on 23 tasks and over 3 seeds. Due to computational constraints, we evaluate on the tasks which had affordance scripts with a high F1-score, and a few with relatively lower scores to investigate the effect of subpar affordance scripts (e.g., `use-slider`). As illustrated in Figure 3 (left), `CoGA` enables over 10x sample efficiency gains over the RL agent early in training at only 1000 steps, and considerable gains on most tasks when an affordance script has a high F1-score. Sample efficiency curves are also shown in Figure 3 (right) on `count-sides` and `click-test-2` for illustration purposes.

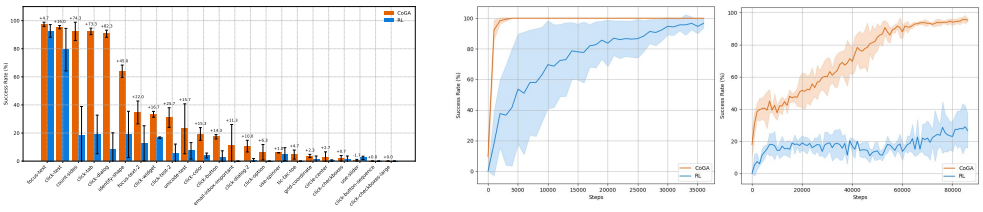

Figure 3: **Left:** Evaluation success rates at 1000 steps for the RL agent and CoGA across tasks. We observe that CoGA is over 10 times more sample efficient than the RL agent early in training at only 1000 steps. **Right:** Evaluation success rate curves for the RL agent and CoGA on count-sides (left) and click-test-2 (right)

CoGA's Affordance Scripts can Generalize within the Same Family of Tasks. We define a family of tasks as tasks with the same affordances, but different optimal policies. For instance, click-test-2 (Figure 2 left - middle) and click-button-sequence have an identical GUI. However, in the former, the task is to learn to click on *either* button ONE or TWO, whereas in the latter, the task is to click on button ONE *then* TWO. Thus, we hypothesize that affordances should generalize within the same family of tasks. We compare the best evaluation success rates obtained by using a task's originally generated affordance script (e.g., click-button-sequence) and its relative's generated affordance script (e.g., click-test-2). For reference, we also include the RL agent's performance on the generalization task considered.

Table 1: Mean evaluation success rates with standard deviation across 3 seeds. We report the best evaluation success rates on the task using its originally generated affordance script (CoGA-o) and its transfer affordance script (CoGA-t). As a reference, we also include the performance of the RL agent on each task.

Task	RL (SR %)	CoGA-o (SR %)	CoGA-t (SR %)
click-button-sequence	3.00 ± 1.00	15.67 ± 1.15	23.67 ± 1.53
focus-text-2	80 ± 28.79	100.00 ± 0.00	100 ± 0.00
click-checkboxes-large	0.00 ± 0.00	0.33 ± 0.58	0.67 ± 0.58

In Table 1, we observe that a generated affordance script indeed generalizes to its relative tasks. Interestingly, we see that using the generated script of a task's relative can sometimes outperform using a task's original script (e.g., click-button-sequence).

CoGA Performs Better or On Par Compared to the Behavioral Cloning Agent when a Limited Number of Expert Demonstrations are Available. We consider the low-data regime where a limited number of expert demonstrations are available. With expert demonstrations available, a natural baseline to consider is behavioral cloning. We thus evaluate a BC agent's performance across data regimes (namely, 10, 50, 200, and 1000 expert demonstrations) and compare with the RL agent and CoGA using only self-collected data. We consider the best evaluation success rates for every baseline across a hyperparameter search (see Appendix C) and over a training period of up to 100,000 steps for the RL agent and CoGA, and 30 epochs for the BC agent. As shown in Figure 4 (right), CoGA performs better on average than the BC baseline with up to 200 expert trajectories, beyond which the BC baseline outperforms. However, the RL baseline only outperforms the BC baseline trained on 10 expert demonstrations. Note that we only consider the tasks on which we were able to collect expert trajectories from using the Pix2Act model. These results demonstrate the impact of constraining the action space using affordances on an agent's performance. Particularly, by combining a limited number of expert data and CoGA, one could expect significant boosts in performance that potentially match that of a BC+RL agent using higher expert data regimes.

5 Related Work

MiniWob++. MiniWoB was originally introduced by Shi et al. (2017) and extended to MiniWob++ (Liu et al., 2018) through additional tasks. Prior works have investigated visual- and

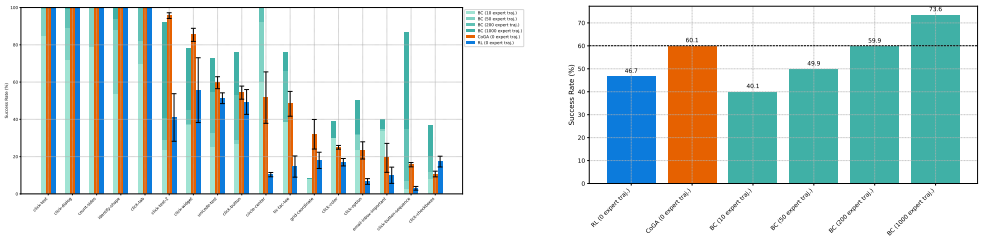

Figure 4: **Right:** Evaluation success rates across tasks and expert data regimes of the BC agent, the RL agent, and CoGA (mean and standard deviation over 3 seeds). **Left:** Mean of evaluation success rates across tasks for the RL agent, CoGA, and increasing expert data regimes of the BC agent.

structure-based approaches. Shi et al. (2017) investigate DOM and pixel observation with an average of 200 human demonstrations per task to train a BC policy followed by RL. CC-Net (Humphreys et al., 2022) extends this approach by increasing the amount of expert demonstrations to a total of 2.4 million and scaling the model architecture. Due to the limited availability of DOM elements, Pix2Act (Shaw et al., 2023) considers pixel-only observations. It uses a transformer base model that was pre-trained to map screenshots to HTML structures. They first fine-tune their model using BC on approximately 1 million expert demonstrations, followed by Monte Carlo Tree Search for policy improvement. More recently, Cheng et al. (2024) employ pre-training to ground large VLMs to GUI coordinates and use them as the GUI agent. Other approaches have considered language-based methods through webpage structural information. Liu et al. (2018) propose workflow-guided exploration (WGE) by using expert demonstrations to learn high-level "workflows" through which the agent learns to select appropriate actions within the workflow through RL. Although similar to CoGA in constraining the action space and minimizing the amount of expert demonstrations required, WGE relies on DOM inputs and parametric learning of the workflows. In contrast, Kim et al. (2023) employ a large language model (LLM) as the agent using HTML code as input. Motivated by contemporary approaches to train an agent to interact with a GUI similarly to humans and without relying on OS- or web-specific APIs (OpenAI, 2025; Anthropic, 2024), our method seeks to explore the *low expert data regime* using *pixel-only observations* and a unified pixel and action type action space. Considering the significant computational and financial costs that VLM agents incur, we argue that only selected prior knowledge of the VLM is relevant for a web agent. Therefore, we propose to distill the relevant information from the VLM through code as affordances.

VLMs for Affordances. VLMs have been used to infer affordances by leveraging their ability to reason about visual and textual information. Using a visual question-answering prompting technique, MOKA (Liu et al., 2024) and KAGI (Lee et al., 2024) employ VLMs to predict keypoint and waypoint affordances from pre-marked visual observations to guide open-world robotics. VoxPoser (Huang et al., 2023) uses LLMs to infer affordances for open-world robotic manipulation given free-form language. They leverage LLMs to write code that interacts with a VLM to compose 3D value maps for grounding the agent's observation space. Our method is complementary to previous work. As detailed in Sec. 3, we build upon their successes to infer affordances using the VLM through pre-marked observations and code.

Code Generation for Reinforcement Learning. Recent works have investigated the role of code generation by foundation models (e.g., LLMs, VLMs) as a mechanism for improving RL sample efficiency. Code as Policies (Liang et al., 2023) introduced a paradigm to prompt LLMs to generate structured programs that serve as policies for robots. Although they do not use VLMs directly to generate code due to their rather limited capabilities at the time, they leverage them for object detection, localization, and segmentation through the generated code. Code as Reward (Venuto et al., 2024) proposes using VLMs to produce dense reward functions as code, improving sample efficiency in RL. Similarly, EUREKA (Ma et al., 2024) is a framework for LLMs to generate and iteratively improve reward functions as code, outperforming human-engineered rewards. Octopus (Yang et al., 2024) uses a VLM to generate code for planning and manipulation. Likewise, Voyager (Wang et al., 2023) leverages LLMs to generate code as actions. VoxPoser (Huang et al., 2023) enables LLMs to write code that interacts with a VLM and ground the agent's observation space through the inferred

affordances. We build on the insights of previous work in this regard, particularly on that of Venuto et al. (2024).

6 Discussion

6.1 Limitations and Failure Modes

1) VLM Pixel Mapping. VLMs struggle to parse an image into pixel coordinates (Cheng et al., 2024). To mitigate this issue, we superimposed a granular coordinate system onto an observation when showing it to the VLM for template image extraction. However, the VLM struggles in specifying exact correct coordinates, which affects the quality of the template images. Consequently, this can decrease the recall and precision of the predicted affordances.

2) Template Matching. Template matching is limited to detecting isomorphic objects across observations. Although this can be mitigated by lowering the template matching threshold (which the critique VLM self-iterates on), it is particularly a limitation for varying text-based objects. To this end, we have experimented with optical character recognition (OCR) tools (e.g., `pytesseract`). However, due to inconsistent OCR, we decided to exclude most tasks with varying text-based observations from our work. In future work, one might explore using more robust OCR tools. Additionally, in some tasks like `bisect-angle` where the agent must click on the location that bisects an angle in half, the VLM fails to extract valid template images as it is unclear how to express affordances in this way.

3) VLM Code Generation. The recall and precision of the predicted affordances depend on the correctness of the generated code. Given perfect templates but incorrect reasoning about the affordable action types and their pixel coordinates, affordances would be compromised. We mitigated this issue through sequential chain-of-thought prompting (Wei et al., 2023) and by using the critique VLM. However, this limitation could be more prevalent in more complex tasks where the script must pick the relevant subset of intents and their corresponding affordances from a set of intents in different situations (e.g., `email-inbox`). Finally, an important limitation was the amount of time taken to generate code for tasks that contained a larger number of affordable elements (e.g., `drag-shapes`).

4) VLM Code Verification. We had to manually label a set of 5 test cases to evaluate the quality of the generated code. Although it is typical to manually create test cases for software systems, one could investigate the VLM's ability to create its own test cases. It is also important to note that the critique VLM is not always correct. For instance, it could label a code as "failed" while scoring high recall and precision. This could be potentially mitigated by incorporating a self-improving paradigm to the critique VLM, whereby it could learn from its mislabeled evaluations of the code based on the results from the manual test cases. Finally, an important limitation that we faced was token limits, which prevented further iterations of the code based on the critique VLM's feedback. This was the case for tasks like `click-shades`, `count-shape`, and `use-slider-2`.

6.2 Future Work

Given perfect affordances, `CoGA`'s performance is limited to the strength of the backbone RL agent (e.g., network architecture, RL algorithm, hyperparameter sweeps), particularly in tasks that require sequential decision-making over multiple steps and are partially observable. We especially note this in tasks like `click-checkboxes` and `click-checkboxes-large` for example. A promising direction for future work is to augment more competent RL agents with `CoGA`, as our method is complimentary to any RL algorithm.

Acknowledgements

LC and DV gratefully acknowledge funding from the FRQNT Master's and Doctoral Training Scholarships, and FK from the FRQS Master's Training Scholarship. The authors would like to thank Mohammad Sami Nur Islam for helping set up and run experiments in early iterations of the project.

The authors are grateful for Mohammad Sami Nur Islam, Yash More, and Nikhil Vemgal for their computational resources, and Bernardo Avila Pires for providing valuable feedback on an early version of the draft.

REFERENCES

Anthropic. Developing a computer use model. 2024. URL https://www.anthropic.com/news/developing-computer-use.

Yen-Chun Chen, Linjie Li, Licheng Yu, Ahmed El Kholy, Faisal Ahmed, Zhe Gan, Yu Cheng, and Jingjing Liu. Uniter: Universal image-text representation learning. 2020.

Kanzhi Cheng, Qiushi Sun, Yougang Chu, Fangzhi Xu, Yantao Li, Jianbing Zhang, and Zhiyong Wu. Seeclick: Harnessing gui grounding for advanced visual gui agents, 2024. URL https://arxiv.org/abs/2401.10935.

James J Gibson. The theory of affordances. *Hilldale, USA*, 1(2), 1977.

Wenlong Huang, Chen Wang, Ruohan Zhang, Yunzhu Li, Jiajun Wu, and Li Fei-Fei. Voxposer: Composable 3d value maps for robotic manipulation with language models, 2023. URL https://arxiv.org/abs/2307.05973.

Peter C Humphreys, David Raposo, Toby Pohlen, Gregory Thornton, Rachita Chhaparia, Alistair Muldal, Josh Abramson, Petko Georgiev, Alex Goldin, Adam Santoro, and Timothy Lillicrap. A data-driven approach for learning to control computers, 2022. URL https://arxiv.org/abs/2202.08137.

Khimya Khetarpal, Zafarali Ahmed, Gheorghe Comanici, David Abel, and Doina Precup. What can i do here? a theory of affordances in reinforcement learning. In *International Conference on Machine Learning*, pp. 5243–5253. PMLR, 2020.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks, 2023. URL https://arxiv.org/abs/2303.17491.

Olivia Y. Lee, Annie Xie, Kuan Fang, Karl Pertsch, and Chelsea Finn. Affordance-guided reinforcement learning via visual prompting, 2024. URL https://arxiv.org/abs/2407.10341.

Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control, 2023.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration, 2018. URL https://arxiv.org/abs/1802.08802.

Fangchen Liu, Kuan Fang, Pieter Abbeel, and Sergey Levine. Moka: Open-world robotic manipulation through mark-based visual prompting, 2024. URL https://arxiv.org/abs/2403.03174.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models, 2024. URL https://arxiv.org/abs/2310.12931.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. URL https://arxiv.org/abs/1312.5602.

OpenAI. Computer-using agent: Introducing a universal interface for ai to interact with the digital world. 2025. URL https://openai.com/index/computer-using-agent.

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

Shengyi Qian, Weifeng Chen, Min Bai, Xiong Zhou, Zhuowen Tu, and Li Erran Li. Affordancellm: Grounding affordance from vision language models, 2024.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021.

Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks, 2019. URL https://arxiv.org/abs/1908.10084.

Stephane Ross and Drew Bagnell. Efficient reductions for imitation learning. In Yee Whye Teh and Mike Titterington (eds.), *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pp. 661–668, Chia Laguna Resort, Sardinia, Italy, 13–15 May 2010. PMLR. URL https://proceedings.mlr.press/v9/ross10a.html.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay, 2016. URL https://arxiv.org/abs/1511.05952.

Peter Shaw, Mandar Joshi, James Cohan, Jonathan Berant, Panupong Pasupat, Hexiang Hu, Urvashi Khandelwal, Kenton Lee, and Kristina N Toutanova. From pixels to ui actions: Learning to follow instructions via graphical user interfaces. *Advances in Neural Information Processing Systems*, 36:34354–34370, 2023.

Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. World of bits: An open-domain platform for web-based agents. In Doina Precup and Yee Whye Teh (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 3135–3144. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/shi17a.html.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017. URL https://arxiv.org/abs/1712.01815.

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

Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015. URL https://arxiv.org/abs/1509.06461.

David Venuto, Sami Nur Islam, Martin Klissarov, Doina Precup, Sherry Yang, and Ankit Anand. Code as reward: Empowering reinforcement learning with vlms, 2024.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models, 2023.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

Jingkang Yang, Yuhao Dong, Shuai Liu, Bo Li, Ziyue Wang, Chencheng Jiang, Haoran Tan, Jiamu Kang, Yuanhan Zhang, Kaiyang Zhou, and Ziwei Liu. Octopus: Embodied vision-language programmer from environmental feedback, 2024. URL https://arxiv.org/abs/2310.08588.

A Prompting Pipeline for Generating Intents and the Initial Affordance Script

- This is an image of a web environment. The agent is the cursor. It can click anywhere on the screen. There may also be other relevant objects that the agent can interact with. The task is to {task description}, but note that the task's specific utterances can vary. Here are some examples: {example utterances}. Can you give a list of the most important elements in this image, ensuring it applies to all the utterances above, not just the specific instance? Give me a list of elements and concise names with description. Do not include the background.
- Based on your description of these elements in the environment, what do you think the agent affords in this environment? The final goal completion and reading the instruction are not affordances. This list should be concise and only contain affordances that are actionable directly by the agent. If multiple affordances can be combined, please combine them into one modular affordance. Please only return a python list of affordance names.
- For each affordance:
  - what are the most relevant objects that you need to identify in this environment to check if the affordance is possible? Give me the minimum list concisely but precisely. Do not give a generic answer such as 'shapes'. Please end your answer by returning a python list of object names. Ensure that the object name does not contain any spaces. The cursor should not be in the list.
- For each object {obj}, for each gridded image:
  - Explain to a 5 year old step by step how to visually identify a {obj} in such an image.
  - Look at the image grid and find the {obj}. Return the bounding box coordinates as [x_left, y_upper, x_left+width, y_upper+height] where width and height describe the size of the box. Ensure that $0 \leq x \leq 160$ and $0 \leq y \leq 210$. Return only the list.
  - The template image for an instance of {obj} has been saved in {template_path}. You can use this template image for {obj} detection using template matching when needed.
- Here is a script that can be used for object detection using template matching. It will be referred to as match_template. You can use it with the template paths, but do not modify it. Here is the script:

```
def match_template(image_array, template_path, save_path):
    # The main image is provided as an array
    main_image = image_array
    template_image_path = template_path
    # Load the template image in grayscale
    template_image = cv2.imread(template_image_path,
cv2.IMREAD_GRAYSCALE)
    if main_image.ndim == 3 and main_image.shape[2] == 3:
        main_image_gray = cv2.cvtColor(main_image,
cv2.COLOR_RGB2GRAY)
    else:
        main_image_gray = main_image
    main_image_rgb = cv2.cvtColor(main_image_gray,
cv2.COLOR_GRAY2RGB)
    w, h = template_image.shape[1], template_image.shape[0]
    result = cv2.matchTemplate(main_image_gray,
template_image, cv2.TM_CCOEFF_NORMED)
    threshold = 0.5
    locations = np.where(result >= threshold)
    bounding_boxes = []
    for pt in zip(*locations[::-1]):
        bounding_box = (pt[0], pt[1], pt[0] + w, pt[1] + h)
        bounding_boxes.append(bounding_box)
```

```
        cv2.rectangle(main_image_rgb, (pt[0], pt[1]), (pt[0]
+ w, pt[1] + h), (0, 255, 0), 2)
    result_image_path = f'matched_templates/save_path'
    cv2.imwrite(result_image_path, main_image_rgb)
    print(f"Number of matches found: {len(bounding_boxes)}")
    return bounding_boxes
```

- Write a step-by-step strategy for determining which affordance to use at a given state of the environment. If multiple affordances apply, your strategy should return them in hierarchical order. Please do not write any code yet. Reading the instruction should not be part of the strategy. The strategy should be independent of the instruction. Make sure your strategy is as specific as possible and that it can generalize to all possible states that the agent can be in.
- For each affordance:
  - Write a step-by-step strategy for determining which action(s) is/are affordable at a given state for the intent of {aff} by examining a current state image. You should return the set of affordable actions to complete the intent of {aff}. Please do not write any code yet. Make sure your strategy is as specific as possible. When deciding on actions, note that CLICK_COORDS and DBLCLICK_COORDS automatically move the cursor and perform the click action, so there is no need to separately use MOVE_COORDS before them The possible actions are: {action_set}. This list contains both action names and their textual descriptions. You should refer to actions by their names in all responses. The PRESS_KEY action type issues a key combination. Each key combination in the allowed_keys config follow the rules: Modifiers are specified using prefixes 'C-' (Control), 'S-' (Shift), 'A-' (Alternate), or 'M-' (Meta). Printable character keys (a, 1, etc.) are specified directly. Shifted characters (A, !, etc.) are equivalent to 'S-' + non-shifted counterpart. Special keys are enclosed in '¡...¿'. The list of valid names is specified inminiwob.constants.WEBDRIVER_SPECIAL_KEYS. Example valid key combinations:'7, 'Enter, 'C-S-ArrowLeft'. To specify relevant x,y coordinates for a selected action, you can use the template matching script for the relevant objects.
- Write a step-by-step strategy for combining the intents and their corresponding set of affordable actions at a given state. After writing your strategy, write an outline of a code using comments. The purpose of the affordable actions returned by the code should not be to solve the task specified in the instruction, but to just help give a prior to an RL agent to learn how to solve the task by helping it narrow down its exploration space. You should return a set of affordable actions to complete the selected intent. This script should not require any other input than the image.
- Write code to implement your strategy for selecting the required affordance and its corresponding set of affordable actions at a given state. First write an outline of the code with comments, then write the filled in code. The purpose of the affordable actions returned by the code should not be to solve the task specified in the instruction, but to just help give a prior to an RL agent to learn how to solve the task by helping it narrow down its exploration space, wihtout telling it the exact solution. You can use the object ID scripts and match_template if needed, but do not modify them. If you use the match_template, make sure to use all of the templates inthe required template's path. The template paths available are: {template_paths}. Make sure your implementation can generalize to all possible states that the agent can be in and pay close attention to what you have written in the steps. You should return a set of affordable actions to complete the selected affordance. This script should not require any other input than the image. The script should be fully executable as is, and should not have placeholder values or pseudocode. It should also be robust to any detection imperfections. Make sure that the coordinate affordances returned are those of the bounding box around the relevant object in the form of [x_left,y_up,x_right,y_down]. The returned affordances should be a list of dictionaries with the format 'action': action_name, 'coords': [x_left,y_up,x_right,y_down]. x and y should not be the centers. The action_name should be exactly as specified in the list of actions. The function that returns the affordable actions must be called 'determine_affordable_actions(image)', where the image is a numpy array. Do not change the function's signature, use it as is. The template paths to use must be

instantiated inside the function. Ony return the code. DO NOT INCLUDE AN EXAMPLE USAGE.

B Prompting Pipeline for Code Review & Regeneration

- For up to 3 times, terminating early if the code is verified as correct:
  - *(To critique VLM)* This is an image of a web environment. The agent is the cursor. It can click anywhere on the screen. There may also be other relevant objects that the agent can interact with. The task is to {task_description}, but note that the task's specific utterances can vary. Here are some examples: {example_utterances}. Do not output anything yet.
  - *The code is executed on 2 example observations for error handling. If the code runs smoothly:*
    - *(To critique VLM)* Please review this code and find problems with it, if any. The affordable actions returned by this code on the first image are: {actions_1}. The affordable actions returned by this code on the second image are: {actions_2}. The purpose of the affordable actions returned by the code should not be to solve the task specified in the instruction, but to just help give a prior to an RL agent to learn how to solve the task by helping it narrow down its exploration space, without telling it the exact solution. Please think about whether these returned affordable actions are suitable for each test case image provided. If the affordable actions are suitable, please only return a string Status: Pass, Reasoning: step-by-step reasoning for why the actions are correct. Else, only return Status: Fail, Reasoning: step-by-step reasoning for why the actions are incorrect, Critique: Point-based feedback on how to solve the issues found in the code (MUST NOT BE JSON). Do not modify the code yet.
  - *However, if there is an error in the code:*
    - *(To critique VLM)* Please review this code as it is throwing this error: {error_trace}. Only return {Status: Fail, Reasoning: step-by-step reasoning for why the code is failing, Critique: Point-based feedback on how to solve the issues found in the code} (MUST NOT BE JSON). Do not modify the code yet. The determine_affordable_actions function must only take the image as argument.
  - *(To critique VLM)* Based on the issues you have found in the code, please improve your code. You should only rewrite the code. Do not mention anything else in your answer. DO NOT INCLUDE EXAMPLE USAGE.

C RL and CoGA Hyperparameter Search Details

The hyperparameters used consistently across all tasks are summarized in Table 2.

Table 2: Hyperparameters used across tasks.

Hyperparameter	Value
Learning rate (lr)	1×10^{-5}
Gradient Norm Clipping threshold ($clip$)	1.0
Epsilon decay (eps_decay)	5000
Batch size ($batch\_size$)	64
Initial epsilon (eps_start)	0.6
Replay buffer size ($buffer\_size$)	40000
Number of timesteps per update ($n\_timesteps$)	50
Discount factor (γ)	0.9
Target network update parameter (τ)	$\{1 \times 10^{-5}, 1 \times 10^{-6}, 1 \times 10^{-7}\}$

For tasks that did not reach a success rate of 100%, an additional hyperparameter search was performed. This focused on increasing the batch size to $batch\_size = \{128, 256\}$ with $\tau = \{1 \times 10^{-5}, 1 \times 10^{-6}\}$, $buffer\_size = 10000$, $eps\_decay = 10000$, $eps\_start = 0.3$.

We used a prioritized replay buffer with parameters $\alpha = 0.6$ and $\beta = 0.4$. All agents use a CNN with five convolutional layers (32, 64, 128, 256, 512 filters, kernel size 3x3, stride 1, padding 1), each followed by batch normalization and ReLU activation, with max-pooling (2x2, stride 2) after the second, fourth, and fifth layers. The extracted features are flattened and passed through two fully connected layers (1024, 384 neurons, ReLU activations), of which the first fully connected layer's features are concatenated with the SBERT instruction embeddings. The model outputs Q-values for 4 action types and 1024 discretized pixel coordinates (32 x 32 bins). We report results over seeds $\{0, 1, 2\}$ and use an AdamW optimizer with default parameters and weight decay of 10^{-5}.

D BC Hyperparameter Details

The hyperparameters used for the BC agent are summarized in Table 3.

Table 3: Hyperparameters used for the BC agent.

Hyperparameter	Value
Learning rate (lr)	1×10^{-4}
Batch size ($batch\_size$)	32
Epochs ($epochs$)	30

E Example Generated Affordance Scripts & Templates

We here show 4 examples of generated affordance scripts and corresponding templates. The first 2 are successful examples, and the last 2 are examples of limitations and failure modes (as shown by the F1-scores in Figure 2 right).

click-tab

```
import cv2
import numpy as np

def match_template(image_array, template_path, save_path):
    main_image = image_array
    template_image_path = template_path
    template_image = cv2.imread(template_image_path,
cv2.IMREAD_GRAYSCALE)

    if main_image.ndim == 3 and main_image.shape[2] == 3:
        main_image_gray = cv2.cvtColor(main_image,
cv2.COLOR_RGB2GRAY)
    else:
        main_image_gray = main_image

    main_image_rgb = cv2.cvtColor(main_image_gray,
cv2.COLOR_GRAY2RGB)
    w, h = template_image.shape[1], template_image.shape[0]
    result = cv2.matchTemplate(main_image_gray,
template_image, cv2.TM_CCOEFF_NORMED)
    threshold = 0.7
    locations = np.where(result >= threshold)

    bounding_boxes = []

    for pt in zip(*locations[::-1]):
        bounding_box = (pt[0], pt[1], pt[0] + w, pt[1] + h)
        bounding_boxes.append(bounding_box)
        cv2.rectangle(main_image_rgb, (pt[0], pt[1]), (pt[0] +
w, pt[1] + h), (0, 255, 0), 2)

    result_image_path = f'matched_templates/{save_path}'
    cv2.imwrite(result_image_path, main_image_rgb)

    return bounding_boxes

def determine_affordable_actions(image):
    # Step 1: Define template paths
    template_paths = [
        './templates/click-tab_Tab1_test_case_1_template.png',
        './templates/click-tab_Tab1_test_case_2_template.png',
        './templates/click-tab_Tab1_test_case_3_template.png',
        './templates/click-tab_Tab1_test_case_4_template.png',
        './templates/click-tab_Tab1_test_case_5_template.png',
        './templates/click-tab_Tab2_test_case_1_template.png',
        './templates/click-tab_Tab2_test_case_2_template.png',
        './templates/click-tab_Tab2_test_case_3_template.png',
        './templates/click-tab_Tab2_test_case_4_template.png',
        './templates/click-tab_Tab2_test_case_5_template.png',
        './templates/click-tab_Tab3_test_case_1_template.png',
        './templates/click-tab_Tab3_test_case_2_template.png',
        './templates/click-tab_Tab3_test_case_3_template.png',
        './templates/click-tab_Tab3_test_case_4_template.png',
        './templates/click-tab_Tab3_test_case_5_template.png'
    ]

    # Step 2: Initialize list for affordable actions
    affordable_actions = []

    # Step 3: Perform template matching for each template
    for template_path in template_paths:
        bounding_boxes = match_template(image, template_path,
'result.png')

        # Step 4: Determine actions based on bounding boxes
        for box in bounding_boxes:
            action = {
                'action': 'CLICK_COORDS',
                'coords': [box[0], box[1], box[2], box[3]]
            }
            affordable_actions.append(action)

    # Step 5: Return the set of affordable actions
    return affordable_actions
```

count-sides

```
import cv2
import numpy as np

def match_template(image_array, template_path, save_path):
    main_image = image_array
    template_image_path = template_path

    template_image = cv2.imread(template_image_path,
cv2.IMREAD_GRAYSCALE)

    if main_image.ndim == 3 and main_image.shape[2] == 3:
        main_image_gray = cv2.cvtColor(main_image,
cv2.COLOR_RGB2GRAY)
    else:
        main_image_gray = main_image

    main_image_rgb = cv2.cvtColor(main_image_gray,
cv2.COLOR_GRAY2RGB)

    w, h = template_image.shape[1], template_image.shape[0]

    result = cv2.matchTemplate(main_image_gray,
template_image, cv2.TM_CCOEFF_NORMED)

    threshold = 0.5
    locations = np.where(result >= threshold)

    bounding_boxes = []

    for pt in zip(*locations[::-1]):
        bounding_box = (pt[0], pt[1], pt[0] + w, pt[1] + h)
        bounding_boxes.append(bounding_box)
        cv2.rectangle(main_image_rgb, (pt[0], pt[1]),
(pt[0] + w, pt[1] + h), (0, 255, 0), 2)

    result_image_path = f'matched_templates/{save_path}'
    cv2.imwrite(result_image_path, main_image_rgb)

    return bounding_boxes

def determine_affordable_actions(image):
    # Step 1: Define template paths
    template_paths = [
        './templates/count-sides-
NumberButtons_test_case_1_template.png',
        './templates/count-sides-
NumberButtons_test_case_2_template.png',
        './templates/count-sides-
NumberButtons_test_case_3_template.png',
        './templates/count-sides-
NumberButtons_test_case_4_template.png',
        './templates/count-sides-
NumberButtons_test_case_5_template.png'
    ]

    # Step 2: Initialize list for storing bounding boxes
    all_bounding_boxes = []

    # Step 3: Use template matching to find NumberButtons
    for i, template_path in enumerate(template_paths):
        bounding_boxes = match_template(image,
template_path, f"output_image_{i}.png")
        all_bounding_boxes.extend(bounding_boxes)

    # Step 4: Determine affordable actions
    affordable_actions = []

    if all_bounding_boxes:
        for box in all_bounding_boxes:
            action = {'action': 'CLICK_COORDS', 'coords':
list(box)}
            affordable_actions.append(action)
    else:
        affordable_actions.append({'action': 'NONE',
'coords': []})

    # Step 5: Return the set of affordable actions
    return affordable_actions
```

use-slider

```
import cv2
import numpy as np

def match_template(image_array, template_path):
    main_image = image_array
    template_image = cv2.imread(template_path, cv2.IMREAD_GRAYSCALE)

    if main_image.ndim == 3 and main_image.shape[2] == 3:
        main_image_gray = cv2.cvtColor(main_image, cv2.COLOR_RGB2GRAY)
    else:
        main_image_gray = main_image

    w, h = template_image.shape[1], template_image.shape[0]
    result = cv2.matchTemplate(main_image_gray, template_image,
cv2.TM_CCOEFF_NORMED)
    threshold = 0.5
    locations = np.where(result >= threshold)

    bounding_boxes = []

    for pt in zip(*locations[::-1]):
        bounding_box = (pt[0], pt[1], pt[0] + w, pt[1] + h)
        bounding_boxes.append(bounding_box)

    return bounding_boxes

def determine_affordable_actions(image):
    slider_handle_templates = [
        './templates/use-
slider_SliderHandle_test_case_1_template_runNum_3.png',
        './templates/use-
slider_SliderHandle_test_case_2_template_runNum_3.png',
        './templates/use-
slider_SliderHandle_test_case_3_template_runNum_3.png',
        './templates/use-
slider_SliderHandle_test_case_4_template_runNum_3.png',
        './templates/use-slider_SliderHandle_test_case_5_template_runNum_3.png'
    ]
    slider_track_templates = [
        './templates/use-slider_SliderTrack_test_case_1_template_runNum_3.png',
        './templates/use-slider_SliderTrack_test_case_2_template_runNum_3.png',
        './templates/use-slider_SliderTrack_test_case_3_template_runNum_3.png',
        './templates/use-slider_SliderTrack_test_case_4_template_runNum_3.png',
        './templates/use-slider_SliderTrack_test_case_5_template_runNum_3.png'
    ]
    submit_button_templates = [
        './templates/use-
slider_SubmitButton_test_case_1_template_runNum_3.png',
        './templates/use-
slider_SubmitButton_test_case_2_template_runNum_3.png',
        './templates/use-
slider_SubmitButton_test_case_3_template_runNum_3.png',
        './templates/use-
slider_SubmitButton_test_case_4_template_runNum_3.png',
        './templates/use-slider_SubmitButton_test_case_5_template_runNum_3.png'
    ]

    slider_handle_coords = []
    for template in slider_handle_templates:
        slider_handle_coords.extend(match_template(image, template))

    slider_track_coords = []
    for template in slider_track_templates:
        slider_track_coords.extend(match_template(image, template))

    submit_button_coords = []
    for template in submit_button_templates:
        submit_button_coords.extend(match_template(image, template))

    actions = []

    if slider_handle_coords and slider_track_coords:
        handle = slider_handle_coords[0]
        track = slider_track_coords[0]
        actions.append({'action': 'MOUSEDOWN_COORDS', 'coords': handle})
        actions.append({'action': 'MOUSEUP_COORDS', 'coords': track})

    if submit_button_coords:
        button = submit_button_coords[0]
        actions.append({'action': 'CLICK_COORDS', 'coords': button})

    return actions
```

use-spinner

```
import cv2
import numpy as np

def match_template(image_array, template_path, save_path):
    main_image = image_array
    template_image = cv2.imread(template_path, cv2.IMREAD_GRAYSCALE)

    if main_image.ndim == 3 and main_image.shape[2] == 3:
        main_image_gray = cv2.cvtColor(main_image, cv2.COLOR_RGB2GRAY)
    else:
        main_image_gray = main_image

    main_image_rgb = cv2.cvtColor(main_image_gray, cv2.COLOR_GRAY2RGB)
    w, h = template_image.shape[1], template_image.shape[0]
    result = cv2.matchTemplate(main_image_gray, template_image,
cv2.TM_CCOEFF_NORMED)
    threshold = 0.5
    locations = np.where(result >= threshold)

    bounding_boxes = []

    for pt in zip(*locations[::-1]):
        bounding_box = (pt[0], pt[1], pt[0] + w, pt[1] + h)
        bounding_boxes.append(bounding_box)
        cv2.rectangle(main_image_rgb, (pt[0], pt[1]), (pt[0] + w, pt[1] + h),
(0, 255, 0), 2)

    result_image_path = f'matched_templates/{save_path}'
    cv2.imwrite(result_image_path, main_image_rgb)

    return bounding_boxes

def get_spinner_value(image):
    return 0

def get_cursor_position(image):
    return (0, 0)

def determine_affordable_actions(image):
    spinner_arrows_templates = [
        './templates/use-spinner_SpinnerArrows_test_case_1_template_runNum_1.png',
        './templates/use-spinner_SpinnerArrows_test_case_2_template_runNum_1.png',
        './templates/use-spinner_SpinnerArrows_test_case_3_template_runNum_1.png',
        './templates/use-spinner_SpinnerArrows_test_case_4_template_runNum_1.png',
        './templates/use-spinner_SpinnerArrows_test_case_5_template_runNum_1.png'
    ]
    submit_button_templates = [
        './templates/use-spinner_SubmitButton_test_case_1_template_runNum_1.png',
        './templates/use-spinner_SubmitButton_test_case_2_template_runNum_1.png',
        './templates/use-spinner_SubmitButton_test_case_3_template_runNum_1.png',
        './templates/use-spinner_SubmitButton_test_case_4_template_runNum_1.png',
        './templates/use-spinner_SubmitButton_test_case_5_template_runNum_1.png'
    ]

    spinner_arrows_coords = []
    for template in spinner_arrows_templates:
        spinner_arrows_coords.extend(match_template(image, template,
"spinner_arrows_result.png"))

    submit_button_coords = []
    for template in submit_button_templates:
        submit_button_coords.extend(match_template(image, template,
"submit_button_result.png"))

    current_value = get_spinner_value(image)
    desired_value = 0
    cursor_position = get_cursor_position(image)

    if current_value != desired_value:
        intent = "adjust_spinner_value"
    else:
        intent = "click_submit"

    affordable_actions = []

    if intent == "adjust_spinner_value":
        for coords in spinner_arrows_coords:
            x1, y1, x2, y2 = coords
            if not (x1 <= cursor_position[0] <= x2 and y1 <=
cursor_position[1] <= y2):
                affordable_actions.append({'action': 'CLICK_COORDS', 'coords':
list(coords)})

    elif intent == "click_submit":
        for coords in submit_button_coords:
            x1, y1, x2, y2 = coords
            if not (x1 <= cursor_position[0] <= x2 and y1 <=
cursor_position[1] <= y2):
                affordable_actions.append({'action': 'CLICK_COORDS', 'coords':
list(coords)})

    return affordable_actions
```

Figure 5: Example scripts across 2 successful tasks (click-tab, count-sides) and 2 unsuccessful tasks (use-slider, use-spinner).

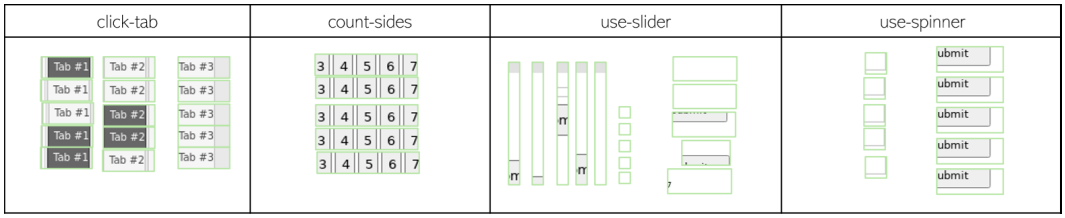

Figure 6: Example template images across 2 successful tasks (click-tab, count-sides) and 2 unsuccessful tasks (use-slider, use-spinner).

F Number of Runs & Code Generation Iterations Across Tasks

Table 4: Number of runs and code generation iterations in the best run across tasks. The former represents the total VLM executions per task, while the latter denotes the number of code generation iterations in the most successful execution.

Task	Number of VLM Runs	Number of Code Generation Iterations in the Best Run
Click-test	1	1
Click-tab	2	2
Circle-center	1	1
Click-test-2	1	2
Focus-text-2	1	2
Focus-text	1	2
Count-sides	1	1
Identify-shapes	1	1
Click-checkboxes	1	1
Click-color	2	1
Tic-tac-toe	3	2
Click-button-sequence	1	2
Click-dialog	2	2
Click-dialog-2	1	3
Click-option	1	2
Unicode-test	3	2
Click-button	3	2
Click-widget	3	2
Email-inbox-important	3	3
Grid-coordinate	2	1
Click-collapsible-nodelay	2	1
Click-shades	3	1
Count-shape	3	2
Use-slider-2	3	1
Use-slider	3	3
Use-spinner	3	3