Abstract

Visual program synthesis is a promising approach to exploit the reasoning abilities of large language models for compositional computer vision tasks. Previous work has used few-shot prompting with frozen LLMs to synthesize visual programs. Training an LLM to write better visual programs is an attractive prospect, but it is unclear how to accomplish this. No dataset of visual programs for training exists, and acquisition of a visual program dataset cannot be easily crowdsourced due to the need for expert annotators. To get around the lack of direct supervision, we explore improving the program synthesis abilities of an LLM using feedback from interactive experience. We propose a method where we exploit existing annotations for a vision-language task to improvise a coarse reward signal for that task, treat the LLM as a policy, and apply reinforced self-training to improve the visual program synthesis ability of the LLM for that task. We describe a series of experiments on object detection, compositional visual question answering, and image-text retrieval, and show that in each case, the self-trained LLM outperforms or performs on par with few-shot frozen LLMs that are an order of magnitude larger. Website: https://zaidkhan.me/ViReP

1. Introduction

Complex visual queries can often be decomposed into simpler subtasks, many of which can be carried out by task-specific perception modules (e.g. object detection, captioning). For example, consider the problem of finding bounding boxes for the phrase “white mug to the left of the sink”. This is a challenging query for single model such as an open vocabulary object detector. However, this query can be solved by writing a program that composes task-specific perception modules with logic: use an open vocabulary object detector to find a sink and white mugs in the scene, then compare the horizontal center of the sink and the mugs to find white mugs to the left of the sink. Program synthesis with large language models [1] is a promising approach to automate this process, and recent work has shown that proprietary large language models can write programs for visual tasks [9, 28, 29]. Current approaches for visual program synthesis with LLMs use few-shot prompting and rely on the in-context learning abilities [33] of frozen, proprietary LLMs. (Fig. 1)

Few-shot prompting with frozen LLMs for visual program synthesis as in ViperGPT [29], VisProg [9], or CodeVQA [28] has several limitations. The LLM needs to understand the competencies of the perception modules it is using. An open vocabulary object detector may able to locate a common attribute-noun phrase such as “white mug” without problems, but struggle with a more abstract phrase such as “microwaveable mug” [25]. A VQA model might be able to answer “is the car blue?” without problems, but fail when logical modifiers are introduced, such as “is the car not blue?” [7]. In many cases, we do not precisely know

![Diagram](https://zaidkhan.me/ViReP)
A natural way to learn from feedback is to use reinforcement learning. ReST [8] and RaFT [6] introduce a general framework for reinforced self-training in generative tasks and demonstrate success in machine translation and text-to-image generation. However, a crucial ingredient in their recipe is the availability of a fine-grained reward model. It is difficult to construct a fine-grained reward model for visual program synthesis, given both the absence of human preference datasets for visual programs, and the difficulty of devising a proxy metric. One alternative is to use unit tests to teach a neural reward model or give a coarse-grained reward. This technique has been used successfully in coding challenges by CodeRL [16] and Haluptzok et al. [10], but it is unclear how it can be applied to visual program synthesis. Our key idea is to use existing annotations for a vision-language as improvised unit tests to provide a coarse reward signal. Using the coarse reward signal, we can apply reinforced self-training by treating the language model as a policy and training it with a simple policy gradient algorithm. We alternate synthetic data generation steps in which we sample programs from the language model policy with optimization steps in which we improve the language model policy based on observations from executing the sampled programs. We name our proposed method VisReP, for Visually Reinforced Program Synthesis.

- We propose optimizing the parameters of a LLM so that the accuracy of the synthesized visual programs is higher, in contrast to previous works that use frozen LLMs.
- Since no dataset of accurate visual programs is available for fine-tuning, we hypothesize that we can instead use feedback from the execution environment to improve the visual program synthesis abilities of a language model.
- We propose VisReP, an offline, model agnostic recipe for reinforced self-training of large language models for visual program synthesis using existing vision-language annotations with a simple policy gradient algorithm.
- Our results show that it is possible to apply reinforced self-training for to improve large language models for visual program synthesis with only coarse rewards.

We demonstrate the effectiveness of an CodeLlama-7B policy trained by VisReP on compositional visual question answering (+9%), complex object detection (+5%), and compositional image-text matching (+15%) relative to the untrained policy. We show that the policy trained by VisReP exceeds the accuracy of a gpt-3.5-turbo policy on all three tasks.
2. Related Work

2.1. Self-Training

Self-training is an established paradigm which uses unlabeled data to improve performance. Self-training has been successfully applied in a number of fields. We restrict our coverage to usages with significant overlap.

**Program Synthesis** Haluptzok et al. [10] showed that LLMs can improve their program synthesis abilities by generating programming puzzles and solving them. CodeRL [16] proposed an actor-critic framework to improve the program synthesis abilities of LLMs for programming problems accompanied by unit tests. CodeIT [3] and Rest-EM [27] also use a similar policy gradient approach for program synthesis. Our problem domain is different from these works, which focus on program synthesis for programming puzzles / problems. In addition, our work has an explicit focus on learning to use an API fluently.

**Alignment** ReST [8] and RaFT [6] introduced a generic framework for reinforced self-training and applied it to align machine translation outputs to human preferences and align foundation models on language understanding and image generation tasks respectively. These works share the same basic idea as our work, though they are in a substantially different task domain where human preferences are either known (conversational alignment) or can be estimated with an available neural model.

**Vision-Language** SelTDA [14] introduced a self-training approach for visual question answering. SelTDA proceeds by pseudolabeling unlabeled data, then finetuning a large VLM on the pseudolabeled data. In contrast to SelTDA, we improve a LLM for visual program synthesis.

2.2. Visual Program Synthesis

Visual program synthesis with LLMs was proposed concurrently by ViperGPT [29], Visprog [9], and CodeVQA [28]. The common points between these three works is that (a) they use pretrained LLMs as code generators (b) they represent complex visual tasks as compositions of primitive visual subtasks (c) they use code to invoke task-specific models to perform the primitive subtasks. Our work is most similar to ViperGPT and CodeVQA as they produce code in a general purpose programming language rather than a DSL. All three works use a proprietary, frozen LLM. In contrast to all three, the focus of our work is on how we can improve the visual program synthesis abilities of an open LLM.

2.3. Tool Use with LLMs

Multimodal tool-using LLMs were first introduced by Socratic Models [34]. However, their approach was to create fixed pipelines in which the output of a perception model such as CLIP [22] is fed to a LLM. Later approaches such as GorillaLLM [20] and ToolLLM [21] improved on this by treating tool use as a program synthesis problem and creating LLMs that use a broad range of tools by learning to invoke APIs. However, one key limitation of these approaches in the context of visual program synthesis is that that they do not learn to decompose problems into subproblems that can be solved by tools. Instead, they are trained to select the right tool for the problem and invoke it. Another limitation is that they are not optimized for functional correctness. They are trained for syntactic and semantic correctness, but they have not been provided feedback on whether their use of tools produces the desired answer. ToolFormer [24] is similar to our work in the sense that the LLM’s usage of tools is grounded by feedback, but they focus on natural language understanding tasks rather than visual tasks.

3. Method

3.1. Visual Program Synthesis with LLMs

**Task Formulation** Let \( v \) be a visual input and \( q \) be a textual query about \( v \). In visual program synthesis, we synthesize a program \( p = \pi_{\theta}(q) \) with a program generator \( \pi_{\theta} \). The program \( p \) and visual input \( q \) are then fed into the execution engine \( y = \phi(v, p) \) to produce a result \( y \). The program generator is an auto-regressive large language model

\[
\pi_{\theta}(y | x) = \prod_{t=1}^{T} \pi_{\theta}(p_t | p_{1:t-1}, x),
\]

where \( p_{1:t} \) are the tokens of the program, and \( x \) is the input to the large language model. The language model is kept frozen in previous work [29]. Our goal is to optimize the parameters \( \theta \) of the language model \( \pi \) so the accuracy of the synthesized programs is higher.

**Implementation** Following ViperGPT [29], we provide the specification of the ImagePatch API concatenated with the textual query \( q \) as the prompt to the program generator. The synthesized program \( p \) is a Python program that can invoke any Python builtins, control flow structures, and the ImagePatch API. Our implementation of the ImagePatch API is largely similar to ViperGPT. We remove some API methods that were not required for the tasks we evaluate on (such as \texttt{lmm\_query}). We use BLIP [17] and GroundingDINO [18] as perception modules underlying \texttt{find} (object detection), \texttt{simple\_query} (visual question answering), and \texttt{verify\_property} (attribute verification).

3.2. Reinforced Self-Training

Rather than use a frozen large language model as the program generator \( p_{\theta} \), we would like to optimize the parameters \( \theta \) of the language model so the accuracy of the synthesized programs is higher. It is not obvious how to do this. We can’t backpropagate through the execution engine \( \phi(p_{\theta}(q), v) \) to...
Figure 2. VisReP can be applied to improve the visual synthesis abilities of an LLM for a vision-language task using existing annotations for a vision-language task (e.g. an object description+image+bounding boxes). A key idea is to construct a coarse reward by comparing the answer produced by a synthesized program to the ground-truth answer.

directly optimize \( \theta \) with respect to \( q \) or \( v \). An alternative might be to use human labor to build a dataset of high-quality visual programs, and train the large language model \( \pi_\theta \) on the manually-collected dataset. But collecting such a dataset is very labor intensive, and not scalable. Instead, we explore the idea of learning from experience by applying a simple policy gradient method, REINFORCE [32].

We propose VisReP, which treats the program synthesis task as a growing batch RL problem [15], inspired by ReST [8]. We first define a coarse discrete reward function \( R(\cdot) \) from existing annotations for a vision-language task. We then alternate Grow steps, in which we sample trajectories (programs) from the policy (large language model), with Improve steps, in which we apply behavioral cloning with a reward-weighted negative-log likelihood loss to improve the policy. A diagram of our approach is depicted in Fig. 2.

**Grow Step**

The grow step corresponds to the acting step in reinforcement learning, and can also be seen as synthetic data generation. Let \( D = \{(v_1, q_1, y_1), \ldots (v_n, q_n, y_n)\} \) be a dataset for a vision-language task, where \( v_i \) is an image, \( q_i \) is a textual query, and \( y_i \) is ground-truth for the \( i \)-th triplet (e.g. a string for VQA, bounding boxes for object detection). We start with the frozen language model \( \pi_\theta(p \mid q) \), where \( p \) is a synthesized program and \( q \) is a textual query. The language model \( \pi_\theta \) represents our policy. We generate a dataset of trajectories \( D_g \) by sampling many programs \( p \) from the current policy \( \pi_\theta; p \sim \pi_\theta(p \mid q) \) for \( q \sim D \).

**Improve Step**

Our goal in this step is to use the dataset of synthetic programs \( D_g \) to improve the policy \( \pi_\theta \). First, we define a binary-valued reward function \( R: p, v, y \rightarrow \{0, 1\} \) on a given program, image, annotation triplet,

\[
R(v, p, y) = \begin{cases} 
1, & \text{if } \phi(p, v) = y \\
0, & \text{otherwise}
\end{cases}
\]

where \( \phi(p, v) \) is the result of executing the program \( p \) on an image \( v \). Note that \( y \) is not a program but an existing annotation such as a string for VQA for a bounding box for object detection. To apply behavioral cloning, we then minimize the reward-weighted loss

\[
J(\theta) = E_{(q,p) \sim D_g} [R(v, p) \mathcal{L}(p; \theta)]
\]

where \( \mathcal{L}(p; \theta) \) is the negative log-likelihood loss

\[
\mathcal{L}_{\text{NLL}}(p, q; \theta) = -E_{(q, p) \sim D_g} \left[ \sum_{t=1}^{T} \log \pi_\theta(p_t \mid p_{1:t-1}, q) \right]
\]

over the pairs of textual queries \( q \) and synthetic programs \( p \) in \( D_g \).

Because the reward function only takes on binary values, we can simplify this and implement it by: First, generating a dataset of synthetic programs \( D_g = \{\pi_\theta(q) \mid \forall q \in D\} \) using the LLM \( \pi_\theta \) on a dataset \( D \). Next, filtering \( D_g \) to obtain \( D_g^* = \{(q, v, p \in D_g : R(q, v, p) > 0)\} \), which corresponds to executing all synthetic programs and only keeping those that give correct answers. Finally, we finetune the language model \( \pi_\theta \) on the filtered dataset \( D_g^* \) using the standard causal language modeling loss. We then iterate the process, initiating a new synthetic data generation step with the improved policy \( \pi_\theta' \).

**Iteration**

For the initial grow step, we use a frozen language model as the initial policy. For example, we use the pretrained \texttt{codellama-7b-instruct-hf} as the policy in the initial grow step. In subsequent steps, we use the policy trained in the previous improve step for the grow step.

4. Understanding Self-Training

Our goal in this section is to characterize the stability and sample efficiency of VisReP. We want to understand:

1. How does applying VisReP change the accuracy of synthesized programs?
2. What happens as VisReP is repeated?
3. How does data scarcity and diversity affect VisReP?

4.1. Implementation

We start off with the GQA [13] dataset for visual question answering. We choose GQA because each question in GQA was constructed programatically and is thus a good candidate to be answered by program synthesis. GQA has over 2M questions, each belonging to one of \( \approx 100 \) question types. We construct a training set by sampling 100 questions for
After Visual Question Answering

Query: Was plastic used to make the empty basket?

Before

image_patch = ImagePatch(image)
empty_basket_patches = image_patch.find("empty basket")
empty_basket_patch = empty_basket_patches[0]
plastic_patches = image_patch.find("plastic")
plastic_in_empty_basket = len(plastic_patches) > 0 and 
len(empty_basket_patch) == len(empty_basket_patches)

result: "yes" wrong

After

image_patch = ImagePatch(image)
empty_basket_patch = empty_basket_patches[0]
plastic_in_empty_basket = len(plastic_patches) > 0 and 
empty_basket_patch.verify_property("empty basket", "plastic")

result: "no" correct

Object Detection

Query: Find all objects fitting the description 'Spoons that are not wooden'.

Before

image_patch = ImagePatch(image)
candidate_patches = image_patch.find("spoon")
verification_question = "Is this not made of wood?"

result: No Detections

After

image_patch = ImagePatch(image)
candidate_patches = image_patch.find("spoon")
verification_question = "Is this a wooden spoon?"

result: [image] (Positive Caption)
result: [image] (Negative Caption)

Image-Text Matching

Verify image matches text="Donuts in a box and a type of meat on a plate"

Before

donuts_patches = image_patch.find("donut")
meat_patches = image_patch.find("meat")
box_patches = image_patch.find("box")
plate_patches = image_patch.find("plate")

result: "no" wrong

After

donuts_in_a_box = donuts_patches[0].verify_property("donuts", "in a box")
meat_on_a_plate = meat_patches[0].verify_property("meat", "on a plate")

result: "yes" correct

Verify image matches text="A type of meat in a box and donuts on a plate"

Before

donuts_plate_patches = image_patch.find("donut")
meat_in_a_box = meat_patches[0].verify_property("meat", "in a box")

result: "yes" wrong

After

donuts_in_a_box = donuts_patches[0].verify_property("donuts", "in a box")
meat_on_a_plate = meat_patches[0].verify_property("meat", "on a plate")

result: "no" correct

Figure 3. Self-training with VisReP produces qualitatively better programs. Here, we show programs written by the initial policy (on the left) and the policy after 10 iterations of self-training on VQA (on the right). In VQA example, the initial policy does not specifically check whether the empty basket is plastic. In the object detection example, the reasoning of the initial policy is correct, but it issues a confusingly worded query to the simple_query module, which returns the wrong answer. The learned policy uses simple_query more appropriately. In the image-text matching example, in the initial policy tries to use the object detector to search directly for “meat in a box” and “donuts on a plate”, but this is too complicated for the object detector to localize. After self-training, the LLM policy no longer makes this mistake.

Dettmers et al. [5]. Full implementation details are in the supplement.
Applying the formulation of self-training in Sec. 3.2 results in a improvement, but iterating it further results in program synthesis quality degrading, rather than increasing (red line in Fig. 5). This is due to the self-training process inadvertently reinforcing incorrect reasoning. A program that uses flawed reasoning can occasionally produce a correct answer. The language model can thus be rewarded for a program that is right for the wrong reasons. If this goes uncorrected, the language model will learn incorrect reasoning patterns.

We hypothesize that providing a small number of human-written corrections for persistent reasoning errors can stabilize the self-training process. We use the question type annotations in GQA to identify question types for which training accuracy decreases over time. These are question types which the language model is not able to self-improve on. We denote them \( Q_{\text{hard}} \). For each question type in \( Q_{\text{hard}} \), we randomly sample one question \( q \) for which the language model synthesized a program that produced the wrong answer. We examine the reasoning in that program, and if the reasoning is flawed, we correct it. We repeat this until we have a program with correct reasoning for each question type in \( Q_{\text{hard}} \), and denote the bank of correct programs as \( P_{\text{gold}} \).

We then retrieve from \( P_{\text{gold}} \) during self-training for use as in-context examples. If a question is annotated with a question type in \( Q_{\text{hard}} \), we retrieve a correct human-written program from \( P_{\text{gold}} \) and use it as an in-context example. If a question is not annotated with a question type in \( Q_{\text{hard}} \), we use a “default” in-context example which is the same for all question types not in \( Q_{\text{hard}} \). We show in Fig. 5 (green line) that this stabilizes self-training and allows the language model to self-improve across all but a few question types (Fig. 4).

### 4.3. Effect of Data Availability on Self-Training

**Training With Less Data** We explore this in a controlled setting, by manipulating the number of samples per question type in GQA. Recall that we originally sample 100 questions per question type for self-training. This dataset had \( \approx 10k \) questions. We construct a training set with only 10 and 1 question per question type, for a total of \( \approx 1000 \) and \( \approx 100 \) questions respectively. Self-training improves upon the baseline (Fig. 6) even when there is an order of magni-
Figure 6. VisReP works even when the amount of available data is reduced by an order of magnitude. We show validation accuracy on GQA. The notation \( n \times k \) indicates \( n \) samples per question type, with \( k \) passes at each sample. For example \( 10 \times 10 \) indicates 10 samples per question type, with 10 passes per sample. Although \( 10 \times 10 \) has 10x fewer unique samples than \( 100 \times 1 \), there is a < 2% accuracy difference between them, indicating that more passes per instance can partially mitigate data scarcity.

Is it possible to mitigate data scarcity? We previously showed that the benefits of self-training reduce when available data is reduced significantly. We now test whether we can mitigate this data scarcity by allowing \( \pi' \) multiple attempts at a query \( q \) during the Grow step. Concretely, we allow \( \pi' \) a total of 10 tries at each query under the setting in which we train with 1 and 10 samples per question type, for a total of 1k and 10k total samples respectively. We show in Fig. 6 that this mitigates the effect of reduced data. Although the data poor 1 × 10 and 10 × 10 have 10x fewer unique questions than 10 × 1 and 100 × 1, their performance is within a standard deviation of their data rich counterparts.

4.4. Quantifying Changes in Syntactic Structure

How do the programs synthesized by the policy change as self-training is iterated? We examine this by looking at how many unique syntactic trees are produced during the Grow step of each iteration. We parse the synthesized programs into abstract syntax trees, and then normalize the trees to remove irrelevant details such as variable names. In the left panel of Fig. 7, we show that the diversity of syntactic forms drops over time. At the beginning, the policy produces a large number of syntactic forms, but appears to “hone in” on a smaller number of forms as self-training continues, and the number of unique syntactic forms drops by almost half.

A remarkably stable set of syntactic forms is conserved from step to step, roughly \( \approx 700 \) (row above diagonal in right panel of Fig. 7). However, the syntactic forms produced by the policy are gradually evolving away from the syntactic forms the initial policy tries, which can be seen in the darkening of the first row in Fig. 7. Despite the coarse reward scheme, the LLM policy gradually explores and learns new syntactic forms.

5. Evaluating Functional Correctness

We measure the functional correctness of the programs synthesized by the self-trained LLM policy \( \pi' \) across three compositional tasks, with the aim of understanding whether:

1. Are the programs produced after self-training more functionally correct than programs produced before self-training?
2. Is it possible to exceed or match the performance of a much larger proprietary LLM with self-training?

For compositional VQA, we use the GQA [13] dataset for the reasons outlined in Sec. 4.1. For complex object detection, we choose Omnilabel [25]. Omnilabel contains 28K free-form object descriptions over 25K images, and is a challenging task for existing open-vocabulary object detectors due to the complexity of the object descriptions. For compositional image-text matching, we choose WinoGround [30] and SugarCrepe [11]. State-of-art vision-language models have trouble reaching above chance accuracy on WinoGround, but SugarCrepe is substantially easier. However, both of these tasks pose significant problems for the ImagePatch API, because many of the relationships mentioned in the text are challenging to detect with the available perception modules. For all experiments, we use ViperGPT[29] as the backbone and adopt their prompts. Due to space limitations, many experimental details are in the supplement.
Table 2. An open LLM policy self-trained with our method substantially outperforms the open policy without self-training, and even outperforms a gpt-3.5-turbo policy. All results use ViperGPT [29] as the backbone. ± numbers are the standard deviation over 5 runs. On all datasets except Omnilabel, we report accuracy. On Omnilabel, we report Macro-F1. Higher is better.

Table 3. VisReP improves benchmark agnostic visual program synthesis. A policy self-trained on GQA with VisReP writes better programs for other VQA datasets and other task types.

5.1. Experimental Setup

For each task, we apply VisReP as described in Sec. 3.2, and evaluate on a held-out subset. For a comparison with a large proprietary LLM, we use gpt-3.5-turbo. We evaluate on a subsampled version of each dataset to reduce token costs. Every LLM is provided the same prompts. Each prompt consists of the ImagePatch API specification used in ViperGPT [29], and 3 in-context examples for each task except for object detection, for which we provide 5 in-context examples.

We use GQA as described in Sec. 4.1. We prepare a compositional subset of Omnilabel [25] by filtering out all descriptions less than two words in length. We then sample a subset of 500 for evaluation, and a subset of 500 for training. To prepare Omnilabel-Hard, we use a state of the art open-vocabulary object detector (GroundingDINO [18]) on the remaining Omnilabel samples, and select those which GroundingDINO completely fails on (no detections) to obtain a hard slice. We then sample a subset of 500 from the hard slice for evaluation. For SugarCrepe [11], we sample 100 positives and their associated negatives from each of the 6 categories, for a total of 600 balanced image-text pairs for validation. We sample 100 of the remaining instances from each category for training. We use all of WinoGround, as it is small enough that there is no need to subsample it. On WinoGround[30], we evaluate the policy trained on SugarCrepe rather than training on it. For VQAv2, we sample 10 questions for each of the top-50 most common answers from the compositional subset curated by [26].

Examples of the inputs for each task are in Fig. 3. We use nucleus sampling with identical parameters for all local LLMs. We use the API default temperature for gpt-3.5-turbo. More details are in the supplement.

5.2. Discussion

Across all three tasks, the policy trained by VisReP outperforms both the gpt-3.5-turbo policy, and the initial CodeLlama-7b policy (Tab. 2). On GQA, the self-trained policy achieves an absolute improvement of almost 9% over the initial policy, and 5% over the gpt-3.5-turbo policy. On Omnilabel, self-training produces a 5% improvement in Macro-F1 score with only 500 training samples. On Omnilabel-Hard, we demonstrate that the visual program synthesis paradigm can localize objects that state of the art open-vocabulary object detectors are unable to localize (Omnilabel-Hard was constructed by selecting instances GroundingDINO [18] cannot localize). Even on Omnilabel-Hard, the self-trained policy outperforms the others. WinoGround and SugarCrepe are difficult to solve by visual program synthesis because many of the relationships are hard to detect with the available perception modules. Despite the intrinsic difficulty of compositional image-text matching for the ImagePatch API, VisReP produces an increase of +15% over the baseline policy. The policy trained on SugarCrepe transfers to WinoGround, outperforming the baseline policy by +10%.

6. Conclusion & Future Work

While few-shot prompting of LLMs for visual program synthesis has produced impressive results, it has limitations, because writing good visual programs requires experience with the visual world and the perception modules at ones disposal. We presented VisReP, which improves a LLM’s program synthesis abilities using feedback from executing visual programs. We showed that VisReP produces strong increases over baseline across multiple tasks, and is competitive with gpt-3.5-turbo. Our work constructed a coarse-valued reward from existing vision-language annotations. Methods like RLAIF [2], ReST [8], and CodeRL [16] all rely on a neural reward model that can provide fine-grained rewards. Learning from fine-grained rewards is much easier than learning from coarse rewards. An interesting direction for future work would be to train a neural reward model for visual program synthesis. Such a reward model could provide fine-grained rewards, and open a broader range of reinforcement learning methods.
References


In the appendix, we provide implementation details in Appendix A, a failure analysis in Appendix B, more qualitative examples in Appendix C, and prompts in Appendix D.

**A. Implementation Details**

We use ViperGPT [29] as our “backbone”. We follow their implementation of the ImagePatch API almost exactly. We remove some modules and functions that were not necessary for the tasks we explore (e.g. llm_query) is not necessary for our test datasets.

A.1. Grow Step

During the Grow step, we use nucleus sampling to stochastically sample programs from the language model. We prompt the language model with the ImagePatch API description in Appendix D. In the Huggingface library, this corresponds to the following configuration. We use a `top_p` value of 0.9, which allows the model to consider the most probable tokens that cumulatively make up 90% of the probability mass. We set `top_k` was set to 0, disabling the top-k filtering and relying solely on nucleus sampling. The `temperature` parameter was set to 0.7. Temperature effects the randomness of token selection, with values lower than 1 resulting in less random selections. We increased the `max_new_tokens` from 180 to 320 to accommodate longer outputs, addressing the issue of premature truncation in programmatic responses. Because the codellama-7b model did not include a `<PAD>` token, we re-use the `<EOS>` token as the pad token.

A.2. Improve Step

During each Improve step, we train the language model using LoRA [12] for a single epoch. Following [5], we apply LoRA to all fully-connected layers in CodeLlama. In the HuggingFace Transformers library, this corresponds to fc1, fc2, k.proj, v.proj, q.proj, out.proj in each transformer block. This corresponds to the MLP blocks and the QKV matrices in the transformer. We use a LoRA rank of 16, set α = 32, and set the LoRA dropout to 0.05. During training, we use a batch size of 4 and the AdamW [19] optimizer. We use an initial learning rate of 0.0002 and apply a linear learning rate scheduler with a warmup ratio of 0.1.

During training, we use the following instruction-following template for language modeling:

```
<s>Write a function using Python and the ImagePatch class (above) that could be executed to provide an answer to the query.
Consider the following guidelines:
- Use base Python (comparison, sorting) for basic logical operations, left/right/up/down, math, etc.
Query: <QUERY GOES HERE>
Program: <PROGRAM GOES HERE>
</s>
```

Note that the first half of the instruction following template (up to Program:) is identical to the end of the prompt used during the Grow step (Appendix D). We only apply the language modeling loss to the tokens of the program, rather than the “instruction”.

A.3. Evaluation Step

Hyperparameters and prompts are identical to the Grow step. Only the datasets change. We use the same prompt (Appendix D), the same set of in-context examples, and the same hyperparameters.

**B. Failure Analysis**

B.1. Why does accuracy decrease on some question types?

In Fig. 4, we show that self-training allows the language model to improve on almost all question types. What is happening on question types that the language model does not improve on? In Tab. 4, we list those problematic question types and examples of questions from each of the problematic question types. Almost all of them tend to have boolean answers or provide a choice between several categories. To understand why self-training can fail on these questions, consider the scenario of a dataset of
Table 4. Examples of question types from Fig. 4 which suffer reduced accuracy after self-training. Almost all of them are either boolean, or require choosing between several categories. In such cases, self-training can reward incorrect reasoning.

entirely boolean questions with possible answers \{yes, no\} where each answer occurs with equal probability. Now consider a language model policy \( \pi_\theta \) that synthesizes programs that result in yes half the time, and programs that result in no half the time. In such a case, the policy will receive a non-zero reward approximately 25\% of the time, regardless of whether the reasoning in the program was correct or not. This can reinforce incorrect patterns of reasoning.

B.2. Failure Modes

B.2.1 Incoherent Reasoning

Is the color of the tag different from the color of bed?

```python
def execute_command(image) -> str:
    image_patch = ImagePatch(image)
    tags = image_patch.find("tag")
    for tag_patch in tags:
        if tag_patch.verify_property("tag", "red"):
            return bool_to_yesno(tag_patch.category != "bed")
    return bool_to_yesno(False)
```

Result: no
Label: yes
unjustified assumption
irrelevant reasoning

Figure 8. An example of a failure mode in which the LLM employs a line of reasoning which is completely incorrect, using both unjustified assumptions and reasoning irrelevant to the question. This was produced by CodeLlama-7b, but similar errors occur with all LLMs tested.

In Fig. 8, we show an example of a severe failure mode. This failure mode occurs with all evaluated LLMs, including gpt-3.5-turbo. First, the LLM makes an unjustified assumption, checking to see if the color of the tag is red. Second, it compares the .category attribute of the tag to the string “bed”. This comparison is irrelevant to the question. Surprisingly, this failure mode occurs even though the LLM is capable of answering other questions of the same question type which require similar reasoning. We hypothesize that in situations where the LLM generates completely incoherent reasoning but is able to answer similar questions correctly, further iterations of reinforced self-training will gradually erase this failure mode. The LLM already “knows” how to synthesize the correct program, but needs additional reinforcement. In situations where the LLM
generates completely incoherent reasoning and is not able to answer similar questions correctly, we hypothesize that further iterations of reinforced self-training will not erase this failure mode. One solution in this case is to provide human-written examples of correct reasoning. As we show in Sec. 4.2, this stabilizes the self-training process.

### B.2.2 Unreliable Perception

![Image of a dog and a squirrel with labels](image)

**Figure 9.** An example of a failure mode in which a perception module is unreliable on a simple input.

Another type of failure mode is one in which the perception modules are unreliable, as shown in Fig. 9. In the case of Fig. 9, the failure occurs in the `find` method, which uses GroundingDino as an open vocabulary object detector. The LLM depends on the `find` method to return an empty list when “squirrel” is not present. However, the object detector spuriously identifies the dog as a squirrel.

### B.2.3 Complex Relationships

![Image of meatballs with complex relationship verification](image)

**Figure 10.** Verifying / detecting complex relationships is challenging for the program synthesis paradigm.

Another failure mode is one in which the LLM must verify or detect a complex relationship that cannot be handled by the perception modules. As an example, consider the query in Fig. 10: “the longer row of meatballs”. Recovering the row structure of the meatballs from the detections is not straightforward. More generally, without a strong visual prior, it is difficult for the LLM to construct a programmatic heuristic for complex relationships.

### C. Qualitative Examples

In Figs. 11 and 12, we show examples of visual questions taken from the GQA validation set in which gpt-3.5-turbo (ViperGPT) incorrectly answers queries, but CodeLlama-7B+VisReP does not. In Figs. 13 and 14, we show examples in which a state-of-the-art open vocabulary object detector (GroundingDino) is not able to localize described objects, but CodeLlama-7B+VisReP succeeds.
Figure 11. Qualitative examples on VQA (GQA) showing errors made by gpt-3.5-turbo (ViperGPT) that are fixed by VisReP.
Figure 12. Qualitative examples on VQA (GQA) showing errors made by gpt-3.5-turbo (ViperGPT) that are fixed by VisReP.
All the pigeons that aren’t fully white

```
image_patch = ImagePatch(image)
pigeon_patches = image_patch.find("pigeon")
# We sort the pigeons by their color, so that the ones that are not fully white are at the end.
pigeon_patches.sort(key=lambda x: x.simple_query("What is the color?"))
# fully white
non_white_pigeon_patches = [_.for_ in pigeon_patches if _.simple_query("What is the color?") != "white"]
return non_white_pigeon_patches
```

The balloons without glasses

```
image_patch = ImagePatch(image)
# Think of things that could be balloons.
search_terms = [
    "balloon",
    "air balloon"
]
candidate_patches = []
# Then search for all of them.
for search_term in search_terms:
candidate_patches.extend(image_patch.find(search_term))
verification_question = "Does this balloon have glasses?"
verified_patches = [_.for_ in candidate_patches if _.simple_query(verification_question) == "no"]
# There could be multiple items that fit this description.
# So we return all of them.
return verified_patches
```

The closer cookie

```
image_patch = ImagePatch(image)
# Think of things that could be cookies.
search_terms = ["cookie"]
candidate_patches = []
# Then search for all of them.
for search_term in search_terms:
candidate_patches.extend(image_patch.find(search_term))
# We can’t really define a question that would be applicable to all cookies, though, so we pick the one that is closest to the upper left corner of the image, since that will be the closest to the edge of the image.
closest_cookie = min(candidate_patches, key=lambda x: x.vertical_center)
return closest_cookie
```

Figure 13. Qualitative examples on object detection (Omnilabel) showing errors made by a state-of-the-art detector (GroundingDino) that are fixed by CodeLlama+VisReP.
**Stools that aren't being used**

```python
image_patch = ImagePatch(image)
# Think of things that could be stools.
search_terms = ['stool']
candidate_patches = []
# Then search for all of them.
for search_term in search_terms:
    candidate_patches.extend(image_patch.find(search_term))
# So now filter the list to only include those.
verified_patches = []
   for _ in candidate_patches:
   if _ simple_query("Is this stool being used?"") == "no"
# There could be multiple items that fit this description.
# So we return all of them.
return verified_patches
```

**Steaks not on a utensil**

```python
image_patch = ImagePatch(image)
# Think of things that could be steaks.
search_terms = ['steak', 'meat']
candidate_patches = []
# Then search for all of them.
for search_term in search_terms:
    candidate_patches.extend(image_patch.find(search_term))
# We need to filter out any steaks that are on a utensil, since
# the question is only asking for steaks not on a utensil.
verified_patches = []
   for _ in candidate_patches:
   if _ verify_property('steak', 'not on a utensil')
# There could be multiple steaks that are not on a utensil here.
# Though the question is only asking for one. We pick a random
# one and return it.
return verified_patches[0]
```

**A bathroom appliance used to wash hands**

```python
image_patch = ImagePatch(image)
# Think of things that could be bathroom appliances.
search_terms = ['bathroom appliance', 'sink', 'toilet']
candidate_patches = []
# Then search for all of them.
for search_term in search_terms:
    candidate_patches.extend(image_patch.find(search_term))
# Verification question.
verification_question = "Is this a bathroom appliance used to wash the hands?"
verified_patches = []
   for _ in candidate_patches:
   if _ simple_query(verification_question) == "yes"
# So we return all of them.
return verified_patches
```

Figure 14. Qualitative examples on object detection (Omnilabel) showing errors made by a state-of-the-art detector (GroundingDino) that are fixed by CodeLlama+VisReP.
D. ImagePatch API

```python
class ImagePatch:
    pass

def __init__(
    self, image, left=None, lower=None, right=None, upper=None, category=None
):
    """Initializes an ImagePatch object by cropping the image at the given
coordinates and stores the coordinates as attributes. If no coordinates are
provided, the image is left unmodified, and the coordinates are set to the
dimensions of the image.
Parameters
--------
image : array_like
    An array-like of the original image.
left, lower, right, upper : int
    An int describing the position of the (left/lower/right/upper) border of the
crop’s bounding box in the original image.
category : str
    A string describing the name of the object in the image.""
    self.image = image
    # Rectangles are represented as 4-tuples, (x1, y1, x2, y2),
    # with the upper left corner given first. The coordinate
    # system is assumed to have its origin in the upper left corner, so
    # upper must be less than lower and left must be less than right.
    self.left = left if left is not None else 0
    self.lower = lower if lower is not None else image.height
    self.right = right if right is not None else image.width
    self.upper = upper if upper is not None else 0
    self.cropped_image = image.crop((self.left, self.upper, self.right, self.lower))
    self.horizontal_center = (self.left + self.right) / 2
    self.vertical_center = (self.upper + self.lower) / 2
    self.category = category

def from_bounding_box(cls, image, bounding_box):
    """Initializes an ImagePatch object by cropping the image at the given
coordinates and stores the coordinates as attributes.
Parameters
--------
image : array_like
    An array-like of the original image.
bounding_box : dict
    A dictionary like {"box": [left, lower, right, upper], "category": str}.""
    pass

@property
def area(self):
    """Returns the area of the bounding box.
Examples
--------
>>> # What color is the largest foo?
>>> def execute_command(image) -> str:
>>>     image_patch = ImagePatch(image)
```
def find(self, object_name):
    """Returns a list of ImagePatch objects matching object_name contained in the
crop if any are found. Otherwise, returns an empty list.
Parameters
----------
object_name : str
    the name of the object to be found
Returns
-------
List[ImagePatch]
    a list of ImagePatch objects matching object_name contained in the crop
Examples
--------
>>> # return the foo
>>> def execute_command(image) -> List[ImagePatch]:
>>>     image_patch = ImagePatch(image)
>>>     foo_patches = image_patch.find("foo")
>>>     return foo_patches"
    pass

def exists(self, object_name):
    """Returns True if the object specified by object_name is found in the image,
    and False otherwise.
Parameters
----------
object_name : str
    A string describing the name of the object to be found in the image.
Examples
--------
>>> # Are there both foos and garply bars in the photo?
>>> def execute_command(image) -> str:
>>>     image_patch = ImagePatch(image)
>>>     is_foo = image_patch.exists("foo")
>>>     is_garply_bar = image_patch.exists("garply bar")
>>>     return bool_to_yesno(is_foo and is_garply_bar"
    pass

def verify_property(self, object_name, visual_property):
    """Returns True if the object possesses the visual property, and False otherwise.
    Differs from 'exists' in that it presupposes the existence of the object s
    pecified by object_name, instead checking whether the object possesses
    the property.
Parameters
----------
object_name : str
    A string describing the name of the object to be found in the image.
visual_property : str
    A string describing the name of the visual property to verify.
String describing the simple visual property (e.g., color, shape, material) to be checked.

Examples
--------
>>> # Do the letters have blue color?
>>> def execute_command(image) -> str:
>>>     image_patch = ImagePatch(image)
>>>     letters_patches = image_patch.find("letters")
>>>     # Question assumes only one letter patch
>>>     return bool_to_yesno(letters_patches[0].verify_property("letters", "blue"))
>>> pass

def simple_query(self, question):
    """Returns the answer to a basic question asked about the image.
    If no question is provided, returns the answer to "What is this?". The questions are about basic perception, and are not meant to be used for complex reasoning or external knowledge.
    Parameters
    question : str
        A string describing the question to be asked.
    Examples
    --------
    >>> # Which kind of baz is not fredding?
    >>> def execute_command(image) -> str:
    >>>     image_patch = ImagePatch(image)
    >>>     baz_patches = image_patch.find("baz")
    >>>     for baz_patch in baz_patches:
    >>>         if not baz_patch.verify_property("baz", "fredding"):
    >>>             return baz_patch.simple_query("What is this baz?")
    >>> # What color is the foo?
    >>> def execute_command(image) -> str:
    >>>     image_patch = ImagePatch(image)
    >>>     foo_patches = image_patch.find("foo")
    >>>     foo_patch = foo_patches[0]
    >>>     return foo_patch.simple_query("What is the color?")
    >>> # Is the second bar from the left quuxy?
    >>> def execute_command(image) -> str:
    >>>     image_patch = ImagePatch(image)
    >>>     bar_patches = image_patch.find("bar")
    >>>     bar_patches.sort(key=lambda x: x.horizontal_center)
    >>>     bar_patch = bar_patches[1]
    >>>     return bar_patch.simple_query("Is the bar quuxy?")
    >>> pass

def visualize(self):
    """Visualizes the bounding box on the original image and annotates it with the category name if provided."""
    pass

def crop_left_of_bbox(self, left, upper, right, lower):
    """Returns an ImagePatch object representing the area to the left of the given
    area, but within the given bounding box."""
    pass
def crop_right_of_bbox(self, left, upper, right, lower):
    """Returns an ImagePatch object representing the area to the right of the given bounding box coordinates.

    Parameters
    ----------
    left, upper, right, lower : int
        The coordinates of the bounding box.

    Returns
    -------
    ImagePatch
        An ImagePatch object representing the cropped area.

    Examples
    --------
    >>> # Is the bar to the right of the foo quuxy?
    >>> def execute_command(image) -> str:
    >>>     image_patch = ImagePatch(image)
    >>>     foo_patch = image_patch.find("foo")[0]
    >>>     right_of_foo_patch = image_patch.crop_right_of_bbox(
    >>>         foo_patch.left, foo_patch.upper, foo_patch.right, foo_patch.lower
    >>>     )
    >>>     return bool_to_yesno(right_of_foo_patch.verify_property("bar", "quuxy"))
    """
    pass

def crop_below_bbox(self, left, upper, right, lower):
    """Returns an ImagePatch object representing the area below the given bounding box coordinates.

    Parameters
    ----------
    left, upper, right, lower : int
        The coordinates of the bounding box.

    Returns
    -------
    ImagePatch
        An ImagePatch object representing the cropped area.

    Examples
    --------
    >>> # Is the bar to the right of the foo quuxy?
    >>> def execute_command(image) -> str:
    >>>     image_patch = ImagePatch(image)
    >>>     foo_patch = image_patch.find("foo")[0]
    >>>     right_of_foo_patch = image_patch.crop_right_of_bbox(
    >>>         foo_patch.left, foo_patch.upper, foo_patch.right, foo_patch.lower
    >>>     )
    >>>     return bool_to_yesno(right_of_foo_patch.verify_property("bar", "quuxy"))
    """
    pass
left, upper, right, lower : int
    The coordinates of the bounding box.

Returns
-------
ImagePatch
    An ImagePatch object representing the cropped area.

Examples
--------
>>> # Is the bar below the foo quuxy?
>>> def execute_command(image) -> str:
>>>     image_patch = ImagePatch(image)
>>>     foo_patch = image_patch.find("foo")[0]
>>>     below_foo_patch = image_patch.crop_below_bbox(
>>>         foo_patch.left, foo_patch.upper, foo_patch.right, foo_patch.lower
>>>     )
>>>     return bool_to_yesno(below_foo_patch.verify_property("bar", "quuxy"))"
> pass

def crop_above_bbox(self, left, upper, right, lower):
    """Returns an ImagePatch object representing the area above the given
    bounding box coordinates.
    """

Parameters
----------
left, upper, right, lower : int
    The coordinates of the bounding box.

Returns
-------
ImagePatch
    An ImagePatch object representing the cropped area.

Examples
--------
>>> # Is the bar above the foo quuxy?
>>> def execute_command(image) -> str:
>>>     image_patch = ImagePatch(image)
>>>     foo_patch = image_patch.find("foo")[0]
>>>     above_foo_patch = image_patch.crop_above_bbox(
>>>         foo_patch.left, foo_patch.upper, foo_patch.right, foo_patch.lower
>>>     )
>>>     return bool_to_yesno(above_foo_patch.verify_property("bar", "quuxy"))"
> pass

def bool_to_yesno(bool_answer: bool) -> str:
    pass

Write a function using Python and the ImagePatch class (above) that could be executed to provide an answer to the query.

Consider the following guidelines:
- Use base Python (comparison, sorting) for basic logical operations, left/right/up/down, math, etc.
INSERT_IN_CONTEXT_EXAMPLES_HERE
Query: INSERT_QUERY_HERE