Self-Training Large Language Models for Improved Visual Program Synthesis With Visual Reinforcement

Zaid Khan[1] Vijay Kumar BG[2] Samuel Schulter[2] Yun Fu[1] Manmohan Chandraker[2,3]
[1]Northeastern University [2]NEC Laboratories America [3]UC San Diego

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

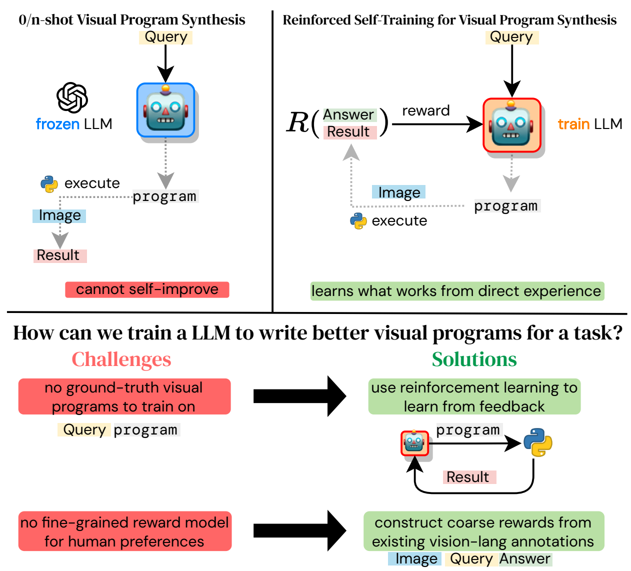

Figure 1. Visual program synthesis with LLMs has been treated as a 0/n-shot task where the LLM is kept frozen. This limits opportunities for improvement. We ask whether it is possible to train a LLM to write more accurate programs. Given that there is no large scale dataset of accurate visual programs available, we propose improving the LLM using self-training.

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, 27, 28]. Current approaches for visual program synthesis with LLMs use few-shot prompting and rely on the in-context learning abilities [32] of frozen, proprietary LLMs. (Fig. 1)

Few-shot prompting with frozen LLMs for visual program synthesis as in ViperGPT [28], VisProg [9], or CodeVQA [27] has several limitations. The LLM needs to understand the competencies of the perception modules it is using. A 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" [24]. 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

	Task Domain	Self-Training	Supervision	Tool/API Use	Visual Task Decomposition	Grounded By Feedback	Improves LLM
VisReP (Ours)	Visual Program Synthesis	Yes	Weak	Yes	Yes	Yes	Yes
Haluptzok et al. [10]	Programming Puzzles	Yes	Weak	No	No	Yes	Yes
ReST [8]	Natural Language Understanding	Yes	-	No	No	Yes	Yes
VisProg [9]	Visual Program Synthesis	No	-	Yes	Yes	No	No
ViperGPT [28]	Visual Program Synthesis	No	Weak	Yes	Yes	No	No
ToolLLM [20]	Tool Usage by API	No	Strong	Yes	No	No	Yes
GorillaLLM [19]	Tool Usage by API	No	Strong	Yes	No	No	Yes
ToolFormer [23]	Natural Language Understanding	Yes	Weak	Yes	No	Yes	Yes

Table 1. Differences between our work and similar work. Strong supervision means that the training process requires examples of ground-truth programs to train the LLM. Weak supervision means that the training process does not require ground-truth programs. Tool / API use means that the LLM is required to use substantial functionality implemented by external modules (e.g. an object detector, a web search API) to solve tasks. Visual task decomposition means that the LLM can decompose a complex visual task into primitive subtasks. Grounded by feedback means that the LLM has been optimized not just for syntactic / semantic correctness (program does not hallucinate / cause errors), but for functional correctness (programs produce the correct answer). Improves LLM means that the work proposes a method to improve an LLM for a specific task, rather than using a frozen LLM.

the weaknesses or competencies of a perception model [30]. Even if this were known, it is difficult to convey *all* of the competencies / weaknesses of a perception module through in-context examples. Second, it is the case in program synthesis that an LLM often *can* generate the correct solution to a problem, but the correct solution is not the solution the LLM places the highest probability on [4]. We would like to align the LLM to "uncover" the knowledge of the correct solution, but it not clear how to do this in a principled way with few-shot prompting alone.

> *How can we train a large language model to write better visual programs for a specific task?*

Our goal is to optimize the parameters of the language model so the accuracy of the synthesized programs is higher. Existing approaches that train LLMs to improve their ability to programatically use tools / APIs such as GorillaLLM [19], ToolLLM [20] do so by finetuning LLMs on examples of tool use or API use. This cannot be directly applied to visual program synthesis because **there are no large scale datasets of visual programs, and collecting such a dataset would be extremely labor intensive.** In the absence of a large scale dataset, how do we learn to write better programs for a visual task?

> *We posit that grounding a language model with interactive feedback from a generic visual task will improve the general visual program synthesis abilities of the model.*

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 finetuning, 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 [26] 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 [28], VisProg [9], and CodeVQA [27]. 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 [33]. However, their approach was to create fixed pipelines in which the output of a perception model such as CLIP [21] is fed to a LLM. Later approaches such as GorillaLLM [19] and ToolLLM [20] 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 [23] 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 π_θ. The program p and visual input q are then fed into the execution engine $\hat{y} = \phi(v, p)$ to produce a result $\hat{y}$. The program generator is an auto-regressive large language model

$$\pi_\theta(\boldsymbol{y} \mid \boldsymbol{x}) = \prod_{t=1}^{T} \pi_\theta\left(p_t \mid \mathbf{p}_{1:t-1}, \boldsymbol{x}\right), \tag{1}$$

where $\mathbf{p}_{1:t}$ 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 [28]. Our goal is to optimize the parameters θ of the language model π so the accuracy of the synthesized programs is higher.

Implementation Following ViperGPT [28], 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 `llm_query`). We use BLIP [17] and GroundingDINO [18] as perception modules underlying `find` (object detection), `simple_query` (visual question answering), and `verify_property` (attribute verification).

3.2. Reinforced Self-Training

Rather than use a frozen large language model as the program generator p_θ, we would like to optimize the parameters θ 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 $\phi(\pi_\theta(q), v)$ to

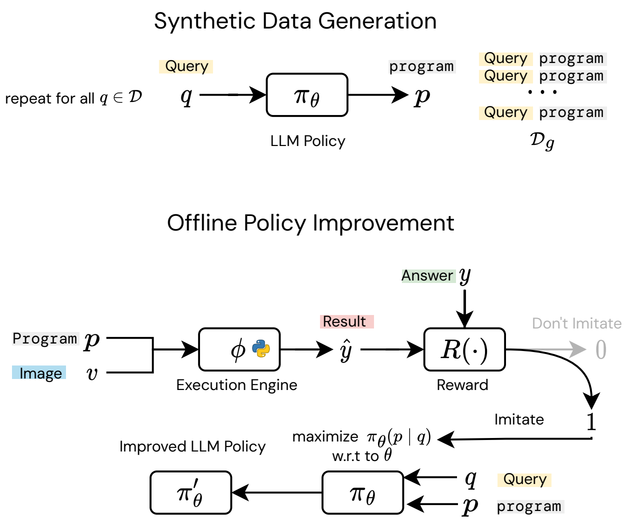

Figure 2. **VisReP** can be applied to improve the visual synthesis abilities of an LLM for a vision-language task using existing annotations for a vision-language task (e.g. an object description+image+bounding boxes). A key idea is to construct a coarse reward by comparing the answer produced by a synthesized program to the ground-truth answer.

directly optimize θ 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 π_θ 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 [31].

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 $\mathcal{D} = \{(v_1, q_1, y_1), \ldots (v_n, q_n, y_n)\}$ 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 π_θ represents our policy. We generate a dataset of trajectories $\mathcal{D}_g$ by sampling many programs p from the current policy π_θ: $p \sim \pi_\theta(p \mid q)$ for $q \sim \mathcal{D}$.

Improve Step Our goal in this step is to use the dataset of synthetic programs $\mathcal{D}_g$ to improve the policy π_θ. First, we define a binary-valued reward function $R : p, v, y \to \{0, 1\}$ on a given program, image, annotation triplet,

$$R(v, p, y) = \begin{cases} 1, & \text{if } \phi(p, v) = y \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

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

$$J(\theta) = \mathbb{E}_{(q,p)\sim\mathcal{D}_g}[R(v, p)\mathcal{L}(p, q; \theta)] \tag{3}$$

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

$$\mathcal{L}_{\text{NLL}}(p, q; \theta) = -\mathbb{E}_{(q,p)\sim\mathcal{D}_g}\left[\sum_{t=1}^{T} \log \pi_\theta\left(p_t \mid p_{1:t-1}, q\right)\right] \tag{4}$$

over the pairs of textual queries q and synthetic programs p in $\mathcal{D}_g$.

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

Iteration For the initial grow step, we use a frozen language model as the initial policy. For example, we use the pretrained `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

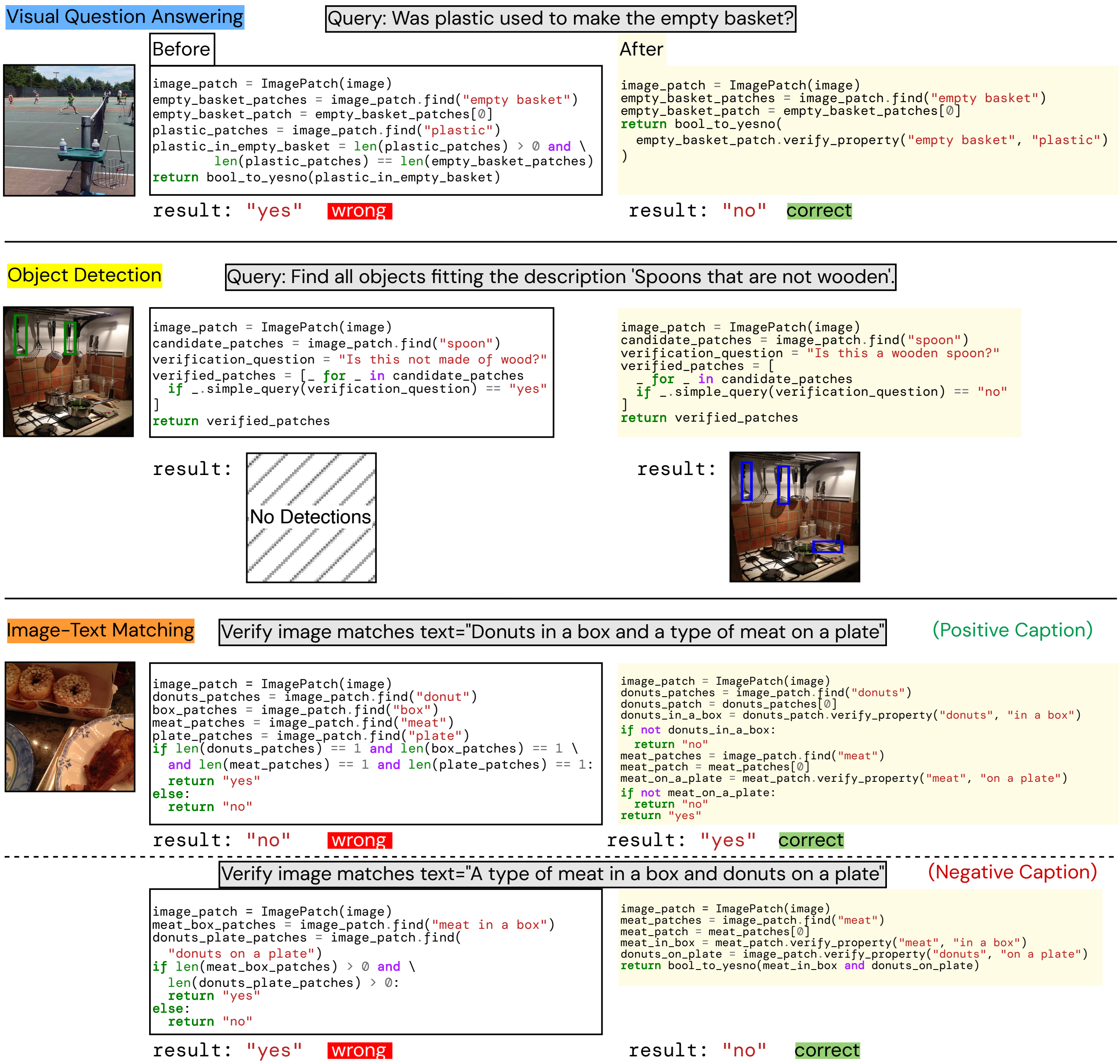

Figure 3. Self-training with **VisReP** produces qualitatively better programs. Here, we show programs written by the initial policy (on the left) and the policy after 10 iterations of self-training on GQA (on the right). In VQA example, the initial policy does not specifically check whether the empty 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.

each question type, for a total of ≈ 10k visual questions and answers. We construct a validation set following Gupta and Kembhavi [9]. We use the CodeLlama [22] family of models as our initial policy. We use LoRA [12] adapters during the **Improve** steps. We use the hyperparameters suggested by Dettmers et al. [5]. Full implementation details are in the supplement.

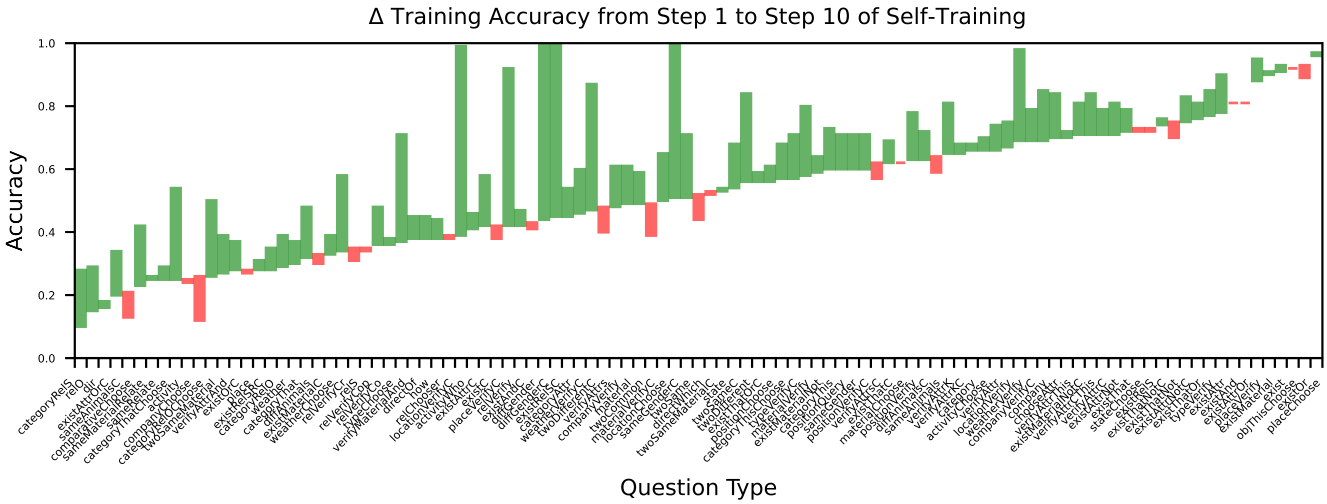

Figure 4. Iteratively applying **VisReP** allows a LLM to self-improve improve on almost all of GQA's ≈ 100 question types. The base of each bar is set to the accuracy of the initial policy (codellama-7b-instruct). A green bar indicates question types on which the policy at iteration 10 improved over the initial policy, and a red bar indicates question types on which the policy at iteration 10 was worse than the initial policy.

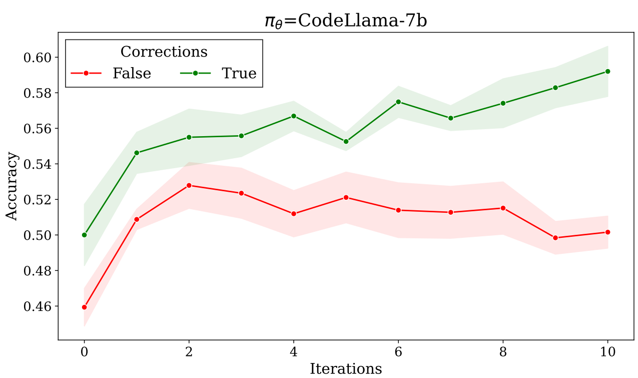

Figure 5. Supplying a small amount of human written corrections as in-context examples during training can increase the stability of the self-training process (green line). We show validation accuracy on GQA through multiple iterations of self-training with a policy instantiated from CodeLlama-7b. Without these corrections, proliferating errors cause performance to degrade in later iterations (red line). The translucent shading around each line indicates the standard deviation over 5 evaluations on the validation set.

4.2. Persistent Errors Harm Iterated Self-Training

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 $\mathcal{Q}_{hard}$. For each question type in $\mathcal{Q}_{hard}$, 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 $\mathcal{Q}_{hard}$, and denote the bank of correct programs as $\mathcal{P}_{gold}$.

We then retrieve from $\mathcal{P}_{gold}$ during self-training for use as in-context examples. If a question is annotated with a question type in $\mathcal{Q}_{hard}$, we retrieve a correct human-written program from $\mathcal{P}_{gold}$ and use it as an in-context example. If a question is not annotated with a question type in $\mathcal{Q}_{hard}$, we use a "default" in-context example which is the same for all question types not in $\mathcal{Q}_{hard}$. 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 ≈ 1000 and ≈ 100 questions respectively. Self-training improves upon the baseline (Fig. 6) even when there is an order of magni-

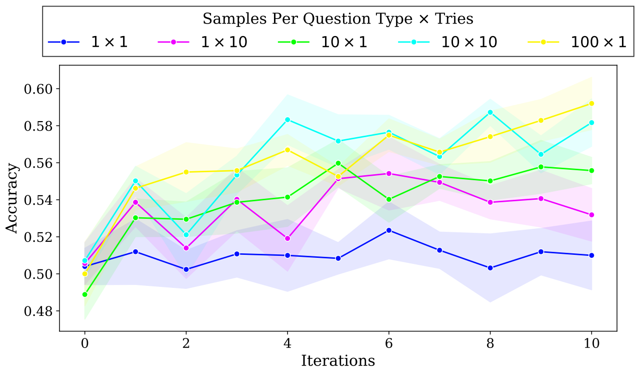

Figure 6. **VisReP** works even when the amount of available data is reduced by an order of magnitude. We show validation accuracy on GQA. The notation $n \times k$ indicates n samples per question type, with k passes at each sample. For example 10×10 indicates 10 samples per question type, with 10 passes per sample. Although 10×10 has 10x fewer unique samples than 100×1, there is a $<$ 2% accuracy difference between them, indicating that more passes per instance can partially mitigate data scarcity.

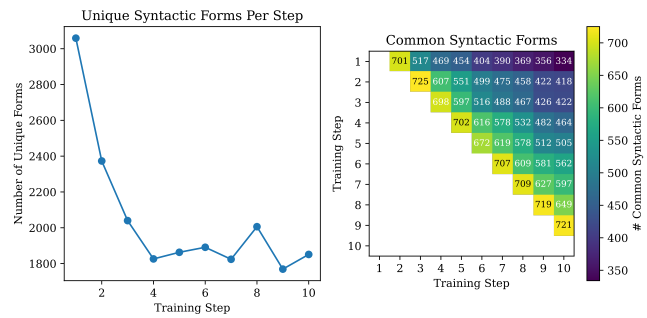

Figure 7. As self-training is iterated, the LLM policy "hones in" on a smaller set of syntactic forms, and gradually evolves away from syntactic forms produced by the initial policy. Left Panel: Number of unique normalized abstract syntax trees seen during each iteration of **VisReP**. Right Panel: Number of unique normalized abstract syntax trees in common between each training step. For example, the entry in row 1, column 6 corresponds to the number of unique abstract syntax trees produced by *both* the policy in iteration 1 (initial policy) and the policy in iteration 6.

tude decrease in training data ($100 \rightarrow 10$) . Only when the amount of available training data is reduced by two orders of magnitude ($100 \rightarrow 1$) does self-training fail to produce an appreciable increase in performance.

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 π_θ multiple attempts at a query q during the **Grow** step. Concretely, we allow π_θ 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 abstract syntax 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 π'_θ 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 [24]. 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 [29] 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[28] as the backbone and adopt their prompts. **Due to space limitations, many experimental details are in the supplement.**

		VQA	Object Detection		Image-Text Matching	
Method	LLM	GQA	Omnilabel	Omnilabel-Hard	Winoground	SugarCREPE
Frozen Proprietary LLM	GPT-3.5-turbo	53.9 ± 0.8	40.0 ± 1.2	26.0 ± 1.1	45.6 ± 1.6	48.9 ± 0.7
Frozen Open LLM	CodeLlama-7B	50.0 ± 1.7	37.3 ± 1.6	23.7 ± 1.4	41.3 ± 0.03	43.5 ± 0.8
Open LLM + **VisReP**	CodeLlama-7B	$\mathbf{59.2 \pm 1.4}$	$\mathbf{42.4 \pm 1.0}$	$\mathbf{28.1 \pm 0.9}$	$\mathbf{52.7 \pm 0.6}$	$\mathbf{58.7 \pm 1.5}$

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 [28] as the backbone. $\pm$ 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.

	X-Dataset Generalization		X-Task Generalization	
Code Generator	VQAv2	OK-VQA	Omnilabel	SugarCrepe
Frozen Few-Shot (7B)	46.6 ± 1.1	12.6 ± 2.1	37.3 ± 1.6	43.5 ± 0.8
VisReP on GQA (7B)	$\mathbf{61.0} \pm 1.0$	$\mathbf{33.7} \pm 1.9$	$\mathbf{39.9} \pm 0.7$	$\mathbf{51.0} \pm 1.5$
VisReP Advantage	+14.4	+21.1	+2.6	+7.5

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 [28], 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 [24] 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 run 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[29], 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 [25].

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

[1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. Program synthesis with large language models. *ArXiv*, abs/2108.07732, 2021. 1

[2] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, John Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, E Perez, Jamie Kerr, Jared Mueller, Jeff Ladish, J Landau, Kamal Ndousse, Kamilė Luko*vs*.iūtė, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noem'i Mercado, Nova DasSarma, Robert Lasenby, Robin Larson, Sam Ringer, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, T. J. Henighan, Tristan Hume, Sam Bowman, Zac Hatfield-Dodds, Benjamin Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom B. Brown, and Jared Kaplan. Constitutional ai: Harmlessness from ai feedback. *ArXiv*, abs/2212.08073, 2022. 8

[3] Natasha Butt, Blazej Manczak, Auke J. Wiggers, Corrado Rainone, David W. Zhang, Michael Defferrard, and Taco Cohen. Codeit: Self-improving language models with prioritized hindsight replay. *ArXiv*, abs/2402.04858, 2024. 3

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. 2021. 2

[5] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *ArXiv*, abs/2305.14314, 2023. 5

[6] Hanze Dong, Wei Xiong, Deepanshu Goyal, Rui Pan, Shizhe Diao, Jipeng Zhang, Kashun Shum, and T. Zhang. Raft: Reward ranked finetuning for generative foundation model alignment. *ArXiv*, abs/2304.06767, 2023. 2, 3

[7] Tejas Gokhale, Pratyay Banerjee, Chitta Baral, and Yezhou Yang. VQA-LOL: visual question answering under the lens of logic. In *ECCV*, 2020. 1

[8] Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alexa Ahern, Miaosen Wang, Chenjie Gu, Wolfgang Macherey, A. Doucet, Orhan Firat, and Nando de Freitas. Reinforced self-training (rest) for language modeling. *ArXiv*, abs/2308.08998, 2023. 2, 3, 4, 8

[9] Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training. *ArXiv*, abs/2211.11559, 2022. 1, 2, 3, 5

[10] Patrick M. Haluptzok, Matthew Bowers, and Adam Tauman Kalai. Language models can teach themselves to program better. *ArXiv*, abs/2207.14502, 2022. 2, 3

[11] Cheng-Yu Hsieh, Jieyu Zhang, Zixian Ma, Aniruddha Kembhavi, and Ranjay Krishna. Sugarcrepe: Fixing hackable benchmarks for vision-language compositionality. In *Thirty-Seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. 7, 8

[12] J. Edward Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *ArXiv*, abs/2106.09685, 2021. 5

[13] Drew A. Hudson and Christopher D. Manning. GQA: A new dataset for real-world visual reasoning and compositional question answering. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 6700–6709. Computer Vision Foundation / IEEE, 2019. 4, 7

[14] Zaid Khan, Vijay Kumar BG, Samuel Schulter, Xiang Yu, Yun Fu, and Manmohan Chandraker. Q: How to specialize large vision-language models to data-scarce vqa tasks? a: Self-train on unlabeled images! In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023. 3

[15] Sascha Lange, Thomas Gabel, and Martin Riedmiller. *Batch Reinforcement Learning*, pages 45–73. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. 4

[16] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. In *Advances in Neural Information Processing Systems*, 2022. 2, 3, 8

[17] Junnan Li, Dongxu Li, Caiming Xiong, and Steven Hoi. Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation. In *ICML*, 2022. 3

[18] Shilong Liu, Zhaoyang Zeng, Tianhe Ren, Feng Li, Hao Zhang, Jie Yang, Chunyuan Li, Jianwei Yang, Hang Su, Jun Zhu, et al. Grounding dino: Marrying dino with grounded pre-training for open-set object detection. *arXiv preprint arXiv:2303.05499*, 2023. 3, 8

[19] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023. 2, 3

[20] Yujia Qin, Shi Liang, Yining Ye, Kunlun Zhu, Lan Yan, Ya-Ting Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Marc H. Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. Toolllm: Facilitating large language models to master 16000+ real-world apis. *ArXiv*, abs/2307.16789, 2023. 2, 3

[21] 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. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, pages 8748–8763. PMLR, 2021. 3

[22] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, I. Evtimov, Joanna Bitton, Manish P Bhatt, Cristian Cantón Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre D'efossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code. *ArXiv*, abs/2308.12950, 2023. 5

[23] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *ArXiv*, abs/2302.04761, 2023. 2, 3

[24] Samuel Schulter, Vijay Kumar B G, Yumin Suh, Konstantinos M. Dafnis, Zhixing Zhang, Shiyu Zhao, and Dimitris Metaxas. Omnilabel: A challenging benchmark for language-based object detection. In *ICCV*, 2023. 1, 7, 8

[25] Ramprasaath R. Selvaraju, Purva Tendulkar, Devi Parikh, Eric Horvitz, Marco Tulio Ribeiro, Besmira Nushi, and Ece Kamar. Squinting at vqa models: Introspecting vqa models with sub-questions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020. 8

[26] Avi Singh, John D. Co-Reyes, Rishabh Agarwal, Ankesh Anand, Piyush Patil, Peter J. Liu, James Harrison, Jaehoon Lee, Kelvin Xu, Aaron Parisi, Abhishek Kumar, Alex Alemi, Alex Rizkowsky, Azade Nova, Ben Adlam, Bernd Bohnet, Hanie Sedghi, Igor Mordatch, Isabelle Simpson, Izzeddin Gur, Jasper Snoek, Jeffrey Pennington, Jiri Hron, Kathleen Kenealy, Kevin Swersky, Kshiteej Mahajan, Laura Culp, Lechao Xiao, Maxwell L. Bileschi, Noah Constant, Roman Novak, Rosanne Liu, Tris Brian Warkentin, Yundi Qian, Ethan Dyer, Behnam Neyshabur, Jascha Narain Sohl-Dickstein, and Noah Fiedel. Beyond human data: Scaling self-training for problem-solving with language models. *ArXiv*, abs/2312.06585, 2023. 3

[27] Sanjay Subramanian, Medhini Narasimhan, Kushal Khangaonkar, Kevin Yang, Arsha Nagrani, Cordelia Schmid, Andy Zeng, Trevor Darrell, and Dan Klein. Modular visual question answering via code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 747–761, Toronto, Canada, 2023. Association for Computational Linguistics. 1, 3

[28] Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, 2023. 1, 2, 3, 7, 8

[29] Tristan Thrush, Ryan Jiang, Max Bartolo, Amanpreet Singh, Adina Williams, Douwe Kiela, and Candace Ross. Winoground: Probing vision and language models for visio-linguistic compositionality. In *CVPR*, 2022. 7, 8

[30] Spencer Whitehead, Suzanne Petryk, Vedaad Shakib, Joseph Gonzalez, Trevor Darrell, Anna Rohrbach, and Marcus Rohrbach. Reliable visual question answering: Abstain rather than answer incorrectly. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 2022. 2

[31] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256, 1992. 4

[32] Sang Michael Xie, Aditi Raghunathan, Percy Liang, and Tengyu Ma. An explanation of in-context learning as implicit bayesian inference. In *International Conference on Learning Representations*, 2022. 1

[33] Andy Zeng, Maria Attarian, Brian Ichter, Krzysztof Choromanski, Adrian Wong, Stefan Welker, Federico Tombari, Aveek Purohit, Michael Ryoo, Vikas Sindhwani, Johnny Lee, Vincent Vanhoucke, and Pete Florence. Socratic models: Composing zero-shot multimodal reasoning with language. *arXiv*, 2022. 3