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

Zaid Khan1  Vijay Kumar BG2  Samuel Schulte2  Yun Fu1  Manmohan Chandraker2,3
1Northeastern University  2NEC Laboratories America  3UC 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

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 Code-VQA [27] has several limitations. The LLM needs to understand the competencies of the perception modules it is using. A open vocabulary object detector may be 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
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-grained reward. 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 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.

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

Table 1. Differences between our work and similar work. Strong supervision means that the training process requires examples of ground-truth. 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.
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 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 \( \pi_\theta \). 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

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

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 [28]. 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 [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 \texttt{lm\_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 \( \phi(\pi_\theta(q), v) \) to
We start with the frozen language model (e.g. a string for VQA, bounding boxes for object detection). π from the current policy D is a synthesized program and y from existing annotations for a vision-language task. We define a binary-valued reward function $R : p, v, y \rightarrow \{0, 1\}$ on a given program, image, annotation triplet, 

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

(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

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

(3)

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

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

(4)

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 $D_g = \{\pi_\theta(q) : \forall q \in D\}$ 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 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 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 dataset [13] for vision question answering. We choose GQA because each question in GQA was constructed programmatically 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, 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 \( \approx 10k \) 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.
4.2. Persistent Errors Harm Iterated Self-Training

Applying the formulation of self-training in Sec. 3.2 results in an 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 $Q_{hard}$. For each question type in $Q_{hard}$, 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 $Q_{hard}$, and denote the bank of correct programs as $P_{gold}$.

We then retrieve from $P_{gold}$ during self-training for use as in-context examples. If a question is annotated with a question type in $Q_{hard}$, we retrieve a correct human-written program from $P_{gold}$ and use it as an in-context example. If a question is not annotated with a question type in $Q_{hard}$, we use a “default” in-context example which is the same for all question types not in $Q_{hard}$. 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 $\approx 1000$ 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.

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.

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 $\pi_\theta$ multiple attempts at a query $q$ during the Grow step. Concretely, we allow $\pi_\theta$ 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 $1 \times 10$ and $10 \times 10$ have 10x fewer unique questions than $10 \times 1$ and $100 \times 1$, 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 $\pi_\theta'$ 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.
positives and their associated negatives from each of the WinoGround, we evaluate the policy trained on SugarCrepe. To prepare Omnilabel-Hard, we run a state of the art VisReP (Tab. 3).

More 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 [28] 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>
<thead>
<tr>
<th>Method</th>
<th>LLM</th>
<th>GQA</th>
<th>Omnilabel</th>
<th>Omnilabel-Hard</th>
<th>Winoground</th>
<th>SugarCrepe</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frozen Proprietary LLM</td>
<td>GPT-3.5-turbo</td>
<td>53.9 ± 0.8</td>
<td>40.0 ± 1.2</td>
<td>26.0 ± 1.1</td>
<td>45.6 ± 1.6</td>
<td>48.9 ± 0.7</td>
</tr>
<tr>
<td>Frozen Open LLM</td>
<td>CodeLlama-7B</td>
<td>50.0 ± 1.7</td>
<td>37.3 ± 1.6</td>
<td>23.7 ± 1.4</td>
<td>41.3 ± 0.03</td>
<td>43.5 ± 0.8</td>
</tr>
<tr>
<td>Open LLM + VisReP</td>
<td>CodeLlama-7B</td>
<td>59.2 ± 1.4</td>
<td>42.4 ± 1.0</td>
<td>28.1 ± 0.9</td>
<td>52.7 ± 0.6</td>
<td>58.7 ± 1.5</td>
</tr>
</tbody>
</table>

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


[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


