ViperGPT: Visual Inference via Python Execution for Reasoning

Dídac Surís*, Sachit Menon*, Carl Vondrick
Columbia University
viper.cs.columbia.edu

Abstract

Answering visual queries is a complex task that requires both visual processing and reasoning. End-to-end models, the dominant approach for this task, do not explicitly differentiate between the two, limiting interpretability and generalization. Learning modular programs presents a promising alternative, but has proven challenging due to the difficulty of learning both the programs and modules simultaneously. We introduce ViperGPT, a framework that leverages code-generation models to compose vision-and-language models into subroutines to produce a result for any query. ViperGPT utilizes a provided API to access the available modules, and composes them by generating Python code that is later executed. This simple approach requires no further training, and achieves state-of-the-art results across various complex visual tasks.

1. Introduction

How many muffins can each kid in Figure 1 (top) eat for it to be fair? To answer this, we might 1) find the children and the muffins in the image, 2) count how many there are of each, and 3) reason that ‘fair’ implies an even split, hence divide. People find it natural to compositionally combine individual steps together to understand the visual world. Yet, the dominant approach in the field of computer vision remains end-to-end models, which do not inherently leverage this compositional reasoning.

Although the field has made large progress on individual tasks such as object recognition and depth estimation, end-to-end approaches to complex tasks must learn to implicitly perform all tasks within the forward pass of a neural network. Not only does this fail to make use of the advances in fundamental vision tasks at different steps, it does not make use of the fact that computers can perform mathematical operations (e.g., division) easily without machine learning. We cannot trust neural models to generalize systematically to different numbers of muffins or children. End-to-end models also produce fundamentally uninterpretable decisions – there is no way to audit the result of each step to diagnose failure. As models grow increasingly data and compute-hungry, this approach grows increasingly untenable. We would like to perform new tasks without additional training by recombining our existing models in new ways.

What limits us from creating such modular systems for more complex tasks? In previous years, the pioneering works of Neural Module Networks [2, 28, 20] attempted to decompose tasks into simpler modules. By training end-to-end with modules rearranged in different ways for different problems, the hope was that each module would learn their appropriate function and thereby become reusable. However, numerous issues made this approach difficult to extend to the real world. In particular, program generation relied on hand-tuned natural language parsers [2], or otherwise required reinforcement learning from scratch and were thus difficult to optimize [20, 28]. In each case, program generation was highly domain-limited. Furthermore, learning the perceptual models jointly with the program generator made training even more difficult, often failing to produce the intended modular structure [3, 49].

In this work, we present ViperGPT1, a framework that overcomes these bottlenecks by leveraging code generating large language models (e.g., GPT-3 Codex [9]) to flexibly compose vision models based on any textual query that defines the task. It creates customized programs for each query that take images or videos as argument and return the result of the query for that image or video. We show that providing Codex an API exposing various visual capabilities (e.g., find, compute_depth), just as one might provide an engineer, is sufficient for the creation of these programs. The model’s prior training on code enables it to reason about how to use these functions and implement the relevant logic. Our results demonstrate that this simple approach delivers remarkable zero-shot performance (i.e., without ever training on task specific images).

Our simple approach enjoys many benefits: it is 1) interpretable, as all the steps are explicit as code function calls with intermediate values that can be inspected; 2) logical, as it explicitly uses built-in Python logical and mathematical operators; 3) flexible, as it can easily incorporate any vision or language module, only requiring the specification of the

---

*Equal contribution. Order determined via coin flip and may be listed either way.

1We name our method after a snake because it executes Python code.
Figure 1. In-the-wild results. Given a visual input and a query, ViperGPT synthesizes a program, then executes it with the Python interpreter in order to produce the final answer. This figure shows both the generated code, and the result of intermediate variables during the execution. By composing pretrained modules, ViperGPT obtains answers that are both correct and interpretable for open-world queries.
associated module be added to the API; 4) **compositional**, decomposing tasks into smaller sub-tasks performed step-by-step; 5) **adaptable** to advances in the field, as improvements in any of the used modules will result in a direct improvement in our approach’s performance; 6) **training-free**, as it does not require to re-train (or finetune) a new model for every new task; and finally, 7) **general**, as it unifies all tasks into one system.

In summary, our contributions are:

1. We propose a simple framework for solving complex visual queries by integrating code-generation models into vision with an API and the Python interpreter, with the benefits above.
2. We achieve state-of-the-art zero-shot results across tasks in visual grounding, image question answering, and video question-answering, showing this interpretability aids performance rather than hindering it.
3. To promote research in this direction, we develop a Python library enabling rapid development for program synthesis for visual tasks, which will be open-sourced upon publication.

2. Related Work

**Modular Vision.** Our work takes inspiration from Neural Module Networks [2, 28], who argue that complex vision tasks are fundamentally compositional and propose dividing them into atomic perceptual units. This visual reasoning procedure has been explored by a variety of works [30, 59]. Posterior efforts have focused on explicitly reasoning about the composition by separating the reasoning from the perception, with connections to neuro-symbolic methods [20, 28, 65]. These approaches are similar in spirit to ours, but require expensive supervision in the form of programs and end-to-end train the perception modules, which makes them not generalizable to different domains.

Due to the practical difficulty of using these methods, the field has primarily moved towards end-to-end all-in-one models [1, 23, 24, 31]. Such models currently obtain state-of-the-art results, and we compare to them in Section 4. Other recent works [66, 46, 57, 36, 38, 16] show that large pretrained models can be used together to great effect, but hand-specify the particular way models are combined.

Over the course of this project, a surge of interest in the area has resulted in a number of related manuscripts appearing on arXiv which use large language models (LLMs) for automatic module integration. In the natural language processing domain, they have been aimed at using external tools [47, 41], or for structured reasoning using Codex [35, 56, 15, 10, 11]. Several concurrent works propose similar solutions for vision tasks. Visual ChatGPT [60] uses pretrained models as tools for a chat agent, but does not generate programs and focuses on visual generation. VisProg [18] generates a list of pseudocode instructions and

![Visual Input](image.png)
\[ r = \phi(x, z) \] to execute the program \( z \) on the input \( x \) and produce a result \( r \). Our framework is flexible, supporting image or videos as inputs \( x \), questions or descriptions as queries \( q \), and any type (e.g., text or image crops) as outputs \( r \).

While prior work represents programs as graphs, like syntax trees [28] or dependency graphs [8], we represent the class of programs \( z \in \mathcal{Z} \) directly through Python code, allowing our programs to capitalize on the expressivity and capabilities afforded by modern programming languages.

### 3.1. Program Generation

Johnson et al. [28] and other work in this direction [20, 65, 26] typically implement \( \pi \) with a neural network that is trained with either supervised or reinforcement learning in order to estimate programs from queries. However, these approaches have largely been unable to scale to in-the-wild settings because either a) the supervision in the form of programs cannot be collected at scale or b) the optimization required for finding the computational graph is prohibitive.

In our approach, we instead capitalize on LLMs for code generation in order to instantiate the program generator \( \pi \) that composes vision and language modules together. LLMs take as input a tokenized code sequence (“prompt”) and autoregressively predict subsequent tokens. We use Codex [9], which has shown remarkable success on code generation tasks. Since we replace the optimization of \( \pi \) with an LLM, our approach obviates the need for task-specific training for program generation. Using Codex as the program generator and generating code directly in Python allows us to draw on training at scale on the Internet, where Python code is abundant.

To leverage LLMs in this way, we need to define a prompt that will sample programs \( z \) that compose and call these modules as needed. Our prompt consists of an application programming interface (API), detailed in the following section, which we provide to the LLM as part of its input context. The final input to the LLM is a sequence of code text consisting of the API specification followed by the query for the sample under consideration. The expected output is a Python function definition as a string, which we then compile and execute.

### 3.2. Modules and Their API

Our prompt, included in the Appendix ???, provides the API for different perceptual and knowledge modules, such as for object detection, depth estimation, or language model queries. From this prompt, we found that LLMs are able to induce correct programs \( z \) from the query \( q \).

The API we provide defines two global classes `ImagePatch` and `VideoSegment`, which represent an image patch and a video segment respectively. Each module is implemented as a class method, which internally calls a pretrained model to compute the result. For example, the `compute_depth` method of `ImagePatch` returns an estimate of the median (relative) depth of the pixels in the image patch; we implement this with state-of-the-art large-scale models such as MiDaS [45]. We provide more details about the modules used in Section 4.

The API specifies the input and output types for each method it defines, as well as doctstrings to explain the purpose of these functions in natural language. Like most APIs, it additionally provides examples that show how to use these classes and their functions, specified in the form of query-code pairs similarly to in-context learning [52, 6].

The input to Codex does not contain the full implementation of the API. Instead, it is given the specification for the API, including the function signatures and doctstrings. Abstracting away the implementation details is beneficial for two reasons. First, LLM context windows are limited in size [6], making it infeasible to include the entire implementation. In addition, the abstraction makes code generation independent of changes made to the module implementation.

End-to-end perception modules are excellent when used
in the right places, and ViperGPT strongly relies on them. Analogous to dual-system models [29] in cognitive science, we argue that generated programs (System 2 - analytic) should be utilized to break down tasks that require multiple steps of reasoning into simpler components, where end-to-end perception modules (System 1 - pattern recognition) are the most effective approach. By composing end-to-end modules into programs, ViperGPT brings the System 2 capability of sequential processing to deep learning [4].

### 3.3. Program Execution

At execution time, the generated program \( z \) accepts an image or video as input and outputs a result \( r \) corresponding to the query provided to the LLM. To execute this program, previous work (e.g., [28]) learns an execution engine \( \phi \) as a neural module network, composing various modules implemented by neural networks. Their modules are responsible for not only perceptual functions such as \( \text{find} \), but also logical ones such as \( \text{compare} \). They learn all neural modules together simultaneously end-to-end, which fails to enable systematic generalization [3] and results in modules that are not faithful to their intended tasks [49], compromising the interpretability of the model.

We provide a simple, performant alternative by using the Python interpreter in conjunction with modules implemented by large pretrained models. The Python interpreter enables logical operations while the pretrained models enable perceptual ones. Our approach guarantees faithfulness by construction.

The program is run with the Python interpreter; as such, its execution is a simple Python call. This means it can leverage all built-in Python functions like \( \text{sort} \); control flow tools like for or if/else; and modules such as \( \text{datetime} \) or math. Notably, this does not require a custom interpreter, unlike prior approaches [18, 47]. Another advantage of a fully Pythonic implementation is compatibility with a wide range of existing tools, such as PyTorch JIT [43].

In our implementation, each program in a generated batch is run simultaneously with multiprocessing. Our producer-consumer design [13] enables efficient GPU batching, reducing the memory and computation costs. Our code is made available at viper.cs.columbia.edu/.

### 4. Evaluation

ViperGPT is applicable to any tasks that query visual inputs with text. Unlike other work using large language models for vision tasks, the return values of our programs can be of arbitrary types, such as text, multiple choice selections, or image regions. We select four different evaluation settings to showcase the model’s diverse capabilities in varied contexts without additional training. The tasks we consider are: 1) visual grounding, 2) compositional image question answering, 3) external knowledge-dependent image question answering, and 4) video causal and temporal reasoning.

We consider these tasks to roughly build on one another, with visual grounding being a prerequisite for compositional image question answering and so on. In the following sections, we explore the capabilities ViperGPT demonstrates in order to solve each task.
Query: The real live version of this toy does what in the winter?

Generated code

def execute_command(image):
    toy = image.simple_query("What is this toy?")
    result = toy_query("The real live version of () does what in the winter?", toy)
    return result

Figure 5. Programmatic chain-of-thought with external knowledge for OK-VQA.

4.1. Visual Grounding

Visual grounding is the task of identifying the bounding box in an image that corresponds best to a given natural language query. Visual grounding tasks evaluate reasoning about spatial relationships and visual attributes. We consider this task first as it serves as the first bridge between text and vision: many tasks require locating complex queries past locating particular objects.

We provide ViperGPT with the API for the following modules (pretrained models in parentheses). find (GLIP [32]) takes as input an image and a short noun phrase (e.g. “car” or “golden retriever”), and returns a list of image patches containing the noun phrase. exists (GLIP [32]) takes as input an image and a short noun phrase and returns a boolean indicating whether an instance of that noun phrase is present in the image. Similarly, verify_property (X-VLM [67]) takes as input an image, a noun phrase representing an object, and an attribute representing a property of that object; it returns a boolean indicating whether the property is present in the image. best_image_match (X-VLM [67]) takes as input a list of image patches and a short noun phrase, and returns the image patch that best matches the noun phrase. Symmetric to this operation, best_text_match takes as input a list of noun phrases and one image, and returns the noun phrase that best matches the image. (This module is not necessary for visual grounding, but rather for tasks with text outputs; we describe it here for simplicity.) They are implemented using an image-text similarity model as in CLIP [44]. Finally, compute_depth (MiDaS [45]) computes the median depth of the image patch. We also define the function distance, which computes the pixel-distance between two patches, using only built-in Python tools.

For evaluation, we use the RefCOCO and RefCOCO+ datasets. The former allows for spatial relations while the latter does not, thereby providing different insights into ViperGPT’s capabilities. We compare ViperGPT against end-to-end methods, and outperform other zero-shot methods on both datasets (see Table 1). We show examples in Figure 4 for example questions as well as our proposed reasoning. Even if a question can be answered end-to-end, it is both more interpretable and more human-aligned to provide intermediate reasoning rather than requiring the model to compress all steps into one forward pass; as our final result is constructed directly from the intermediate values, they provide a fully faithful interpretation of how the model came to its answer.

For GQA, we incorporate the module simple_query (BLIP-2 [32]), which handles basic queries that are not further decomposable, such as “What animal is this?” We also add the aforementioned best_text_match. This leads us to the best accuracy on GQA among zero-shot models (Table 4).

4.2. Compositional Image Question Answering

We also evaluate ViperGPT on image question answering. We focus on compositional question answering, which requires decomposing complex questions into simpler tasks. We use the GQA dataset [27], which was created to measure performance on complex compositional questions. Consider Figure 4 for example questions as well as our provided reasoning. Even if a question can be answered end-to-end, it is both more interpretable and more human-aligned to provide intermediate reasoning rather than requiring the model to compress all steps into one forward pass; as our final result is constructed directly from the intermediate values, they provide a fully faithful interpretation of how the model came to its answer.

We evaluate on the OK-VQA dataset [37], which is designed to evaluate models’ ability to answer questions about images that require knowledge that cannot be found in the image. Items in this dataset often require more than one step of reasoning to produce a correct answer. For example, in Figure 5, one must first perceive from the image that

<table>
<thead>
<tr>
<th>Table 3. OK-VQA Results.</th>
<th>Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRiG [14]</td>
<td>50.5</td>
</tr>
<tr>
<td>KAT [17]</td>
<td>54.4</td>
</tr>
<tr>
<td>RA-VQA [33]</td>
<td>54.5</td>
</tr>
<tr>
<td>REVIVE [34]</td>
<td>58.0</td>
</tr>
<tr>
<td>PromptCap [22]</td>
<td>58.8</td>
</tr>
<tr>
<td>PICa [63]</td>
<td>43.3</td>
</tr>
<tr>
<td>BLIP-2 [31]</td>
<td>45.9</td>
</tr>
<tr>
<td>Flamingo [1]</td>
<td>50.6</td>
</tr>
<tr>
<td>ViperGPT (ours)</td>
<td>51.9</td>
</tr>
</tbody>
</table>

2Examples in the paper have been cosmestically cleaned by removing comments and error handling, but the logic is unchanged.
Generated code

def execute_command(video, question, possible_answers):
    video_segment = VideoSegment(video)
    drop_detected = False
    for i, frame in enumerate(video_segment.frame_iterator()):
        frame_of_interest = ImagePatch(video_segment, index_frame)
        if frame.exists("boy") and frame.exists("sparkles") and 
            frame.simple_query("is the boy dropping the sparkles?") == "yes":
            drop_detected = True
            break
        if drop_detected:
            index_frame = i + 1
            break
    if drop_detected:
        index_frame = video_segment.num_frames // 2
        frame_of_interest = ImagePatch(video_segment, index_frame)
        boys = [frame_of_interest]
        if len(boys) == 0:
            boys = frame_of_interest.find("boy")
    answer = select_answer(info, question, possible_answers)
    return answer

Query: What did the boy do after he dropped the sparkles on the floor?

Generated code

def execute_command(video, question, possible_answers):
    video_segment = VideoSegment(video)
    drop_detected = False
    for i, frame in enumerate(video_segment.frame_iterator()):
        if frame.exists("boy") and frame.exists("sparkles") and 
            frame.simple_query("is the boy dropping the sparkles?") == "yes":
            drop_detected = True
            break
        if drop_detected:
            index_frame = i + 1
            break
    if drop_detected:
        index_frame = video_segment.num_frames // 2
        frame_of_interest = ImagePatch(video_segment, index_frame)
        boys = [frame_of_interest]
        if len(boys) == 0:
            boys = frame_of_interest.find("boy")
    answer = select_answer(info, question, possible_answers)
    return answer

Query: How does the black dog position himself at the end?

Generated code

def execute_command(video, question, possible_answers):
    video_segment = VideoSegment(video)
    last_frame = ImagePatch(video_segment, -1)
    last_caption = last_frame.simple_query("What is this?")
    info = 
        "Caption of last frame": last_caption,
        "What is the dog doing": dog.simple_query("What is the dog doing?")
    answer = select_answer(info, question, possible_answers)
    return answer

Table 4. NExT-QA Results. Our method gets overall state-of-the-art results (including supervised models) on the hard split. “T” and “C” stand for “temporal” and “causal” questions, respectively.

<table>
<thead>
<tr>
<th>Model</th>
<th>Hard Split - T</th>
<th>Hard Split - C</th>
<th>Full Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATP [7]</td>
<td>45.3</td>
<td>43.3</td>
<td>54.3</td>
</tr>
<tr>
<td>VGT [61]</td>
<td>-</td>
<td>-</td>
<td>56.9</td>
</tr>
<tr>
<td>HiTea [64]</td>
<td>48.6</td>
<td>47.8</td>
<td>63.1</td>
</tr>
<tr>
<td>ViperGPT</td>
<td>49.8</td>
<td>56.4</td>
<td>60.0</td>
</tr>
</tbody>
</table>

We evaluate using the NExT-QA multiple choice version. We provide an additional module select_answer (GPT-3 [6]), which, given textual information about a scene and a list of possible answers, returns the answer that best fits the information. Other than that, the only additional content given in the API is the definition of the class VideoSegment, that contains the video bytestream as well as the start and end timestamps of the video segment that it represents. It also defines an iterator over the frames, which returns an ImagePatch object representing every frame.

We find that despite only being provided with perception modules for images, ViperGPT displays emergent causal and temporal reasoning, extending to videos and displaying its ability to perform this type of reasoning. 

Figure 6. Temporal reasoning on NExT-QA.

"this toy" is a "bear," then use external knowledge to answer what bears do in the winter. End-to-end models must directly produce an answer, and therefore may pick words that are more directly related to the image than the question intended. In this case, the best available end-to-end model guesses "ski," presumably as that is a common winter activity (though, not for bears). ViperGPT, on the other hand, can employ a form of chain-of-thought reasoning [58] to break down the question as previously described, first determining the type of toy using perception modules and then using the perceived information in conjunction with an external knowledge module to produce the correct response.

ViperGPT outperforms all zero-shot methods, and when compared to models using publicly available resources, it surpasses the best previous model by 6%, a wide margin for this dataset (see Table 3).

4.4. Video Causal/Temporal Reasoning

We also evaluate how ViperGPT extends to videos and queries that require causal and temporal reasoning. To explore this, we use the NEXT-QA dataset, designed to evaluate video models ability to perform this type of reasoning.
temporal reasoning when applied to videos provided as an ordered list of images. In particular, we observe that programs apply perception to determine which frames are relevant for a given query, then reasons about the information extracted from these frames along with associated frame numbers to produce a final answer.

Despite seeing no video data whatsoever, ViperGPT achieves accuracy results on par with the best supervised model (see Table 4), and even surpassing it on the NeXt-QA hard split [7], both for temporal and causal queries. Of course, the framework of ViperGPT also allows for incorporation of video models, which we expect would further improve the performance well beyond this threshold.

Computational ability presents even more of an obstacle for video understanding than for images. It is infeasible to fit every frame of a moderately-sized video into GPU memory on even the best hardware. ViperGPT may provide a way forward for video understanding that overcomes the limitations of systems that need to perform computation on a whole video simultaneously. See examples in Figure 6.

5. Exploring New Capabilities

In this section, we showcase various interesting capabilities enabled by use of ViperGPT.

5.1. Queries Beyond Benchmarks

We believe that the evident strength of this approach may not be adequately explored by existing benchmarks, which are designed for end-to-end models. In Figure 1, we show examples of interesting queries that are interesting in the real world but would not show up in existing benchmarks. We do not add any new API specifications other than the ones already used in the benchmarks. See the Appendix ?? for more details.

These examples show that the modules we included are general and cover a wide range of tasks. In settings where new capabilities are required, the framework is general and permits the addition of any modules, like ocr, surface_normal_estimation, segmentation, etc.

5.2. Interventional Explainability

Our programmatic approach enables automatic diagnosis of which modules are responsible for prediction errors, potentially informing which types of models to improve and where to collect more data. Evaluating the intermediate output of each module is impractical due to the lack of ground truth labels, and naive comparing accuracy between programs that use a certain module and those that do not could be confounded e.g. by the difficulty of the problem. We can instead perform interventions to better understand a module’s performance. For each module, we can define a default value that provides no information, and substitute the underlying model for this default output. For instance, find could always return the full input image. We can then consider how much performance drops if evaluating the same code for the examples that use that module. If the intervention has a minimal impact on performance, the module is likely not useful.

We show an example of this analysis in Figure 7 for visual grounding on RefCOCO, where we observe a similar level of importance for perception modules and Python operations. Both are tightly integrated in our approach.

5.3. Conditioning on Additional Information

We found ViperGPT readily admits program generation based on additional knowledge. This context can be provided as a comment prior to the code generation. Such context can be critical to correctly responding to a wide range of queries. In Figure 8 we show one such example. The correct side of the road varies by country, so the initial query cannot be answered. Provided with the context of where the photo was taken, the model produces different logic for each case, adjusted based on the relevant prior knowledge.

6. Conclusions

We present ViperGPT, a framework for programmatic composition of specialized vision, language, math, and logic functions for complex visual queries. ViperGPT is capable of connecting individual advances in vision and language; it enables them to show capabilities beyond what any individual model can do on its own. As the models implementing these functions continue to improve, we expect ViperGPT’s results will also continue to improve in tandem.
Acknowledgements: We thank Revent Teotia and Chengzhi Mao for helpful feedback. This research is based on work partially supported by the DARPA MCS program under Federal Agreement No. N660011924032 and the NSF CAREER Award #2046910. DS is supported by the Microsoft PhD Fellowship and SM is supported by the NSF GRFP.

References


[57] Zhenhailong Wang, Manling Li, Ruochen Xu, Luowei Zhou, Jie Lei, Xudong Lin, Shuohang Wang, Ziyi Yang, Chenguang Zhu, Derek Hoiem, et al. Language models with image descriptors are strong few-shot video-language learners. 2022.


