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, 27, 19] 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 [19, 27]. 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, 48].

In this work, we present ViperGPT[1], 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

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

[1] We 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.

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 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, 27], 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 [29, 57]. Posterior efforts have focused on explicitly reasoning about the composition by separating the reasoning from the perception, with connections to neuro-symbolic methods [19, 27, 62]. 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, 22, 23, 30]. Such models currently obtain state-of-the-art results, and we compare to them in Section 4. Other recent works [63, 45, 55, 35, 37, 15] 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 [46, 40], or for structured reasoning using Codex [34, 54, 14, 10]. Concurrent work [17] generates a list

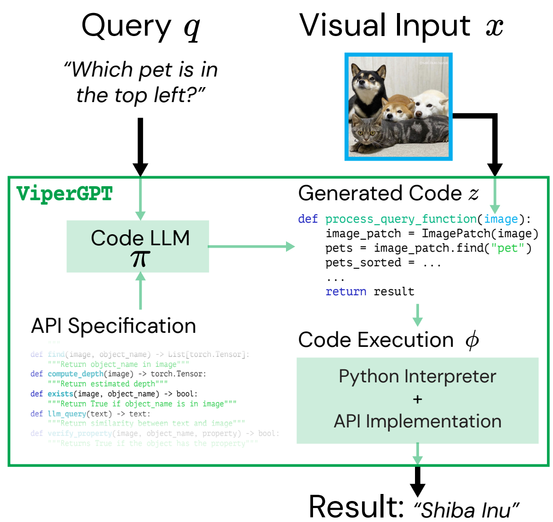

Figure 2. **Method**. `ViperGPT` is a framework for solving complex visual queries programmatically.

of pseudocode instructions and interprets them as a 'visual program,' relying on in-context learning from provided examples. Unlike them, we directly generate unrestricted Python code, which is much more flexible and enables us to demonstrate more advanced emergent abilities, such as control flow and math. Crucially, using Python allows us to leverage the strong prior knowledge Codex learns by training at scale from the Internet. Additionally, we evaluate on many established benchmarks measuring visual understanding and achieve top-performing zero-shot results.

Interpretability. The area of interpretability for complex queries in vision is extensive. Many approaches provide explanations in the form of pixel importance, à la GradCAM [47, 65, 11, 41], some also providing textual explanations [41]. These are often post-hoc explanations rather than by construction, and do not give step-by-step reasoning including image crops and text. Hard attention in captioning [59] aims for a similar goal regarding intermediate image crops, similarly to our `find` module, but has proven difficult to incorporate into learning algorithms. See He *et al*. [18] for a complete overview.

Pretrained models. The perception and external knowledge modules used by `ViperGPT` are GLIP [31] for object detection, X-VLM [64] for text-image similarity (as it surpasses CLIP [43] at attribute detection [5]), MiDaS [44] for depth estimation, GPT-3 [6] for external knowledge, and BLIP-2 [30] for simple visual queries.

3. Method

We use notation following Johnson *et al*. [27]. Given a visual input x and a textual query q about its contents, we first synthesize a program $z = \pi(q)$ with a program generator π given the query. We then apply the execution engine $r = \phi(x, z)$ to execute the program z on the input x and pro-

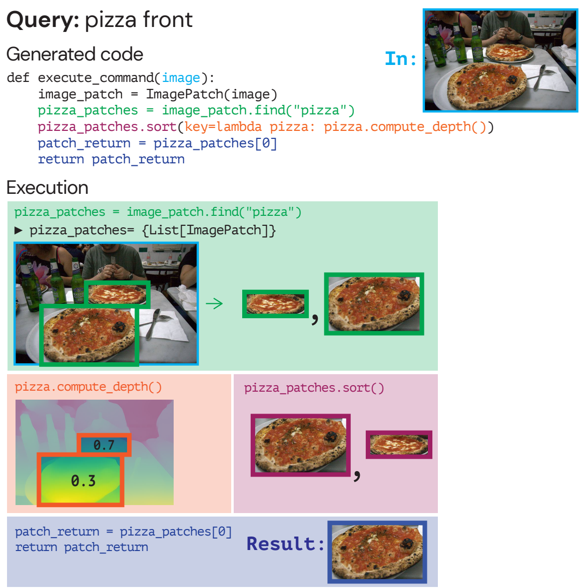

Figure 3. **Visual grounding on RefCOCO.**

duce 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 [27] 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*. [27] and other work in this direction [19, 62, 25] typically implement π 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 π 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 π 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

Table 1. **RefCOCO Results**. We report accuracy on the REC task and testA split. ZS=zero shot, Sup.=supervised.

		IoU (%) ↑	
		RefCOCO	RefCOCO+
Sup.	MDETR [53]	90.4	85.5
	OFA [53]	94.0	91.7
ZS	OWL-ViT [38]	30.3	29.4
	GLIP [31]	55.0	52.2
	ReCLIP [49]	58.6	60.5
	`ViperGPT` (ours)	**72.0**	**67.0**

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 B, 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 [44]. 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 docstrings 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 [50, 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 docstrings. 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.

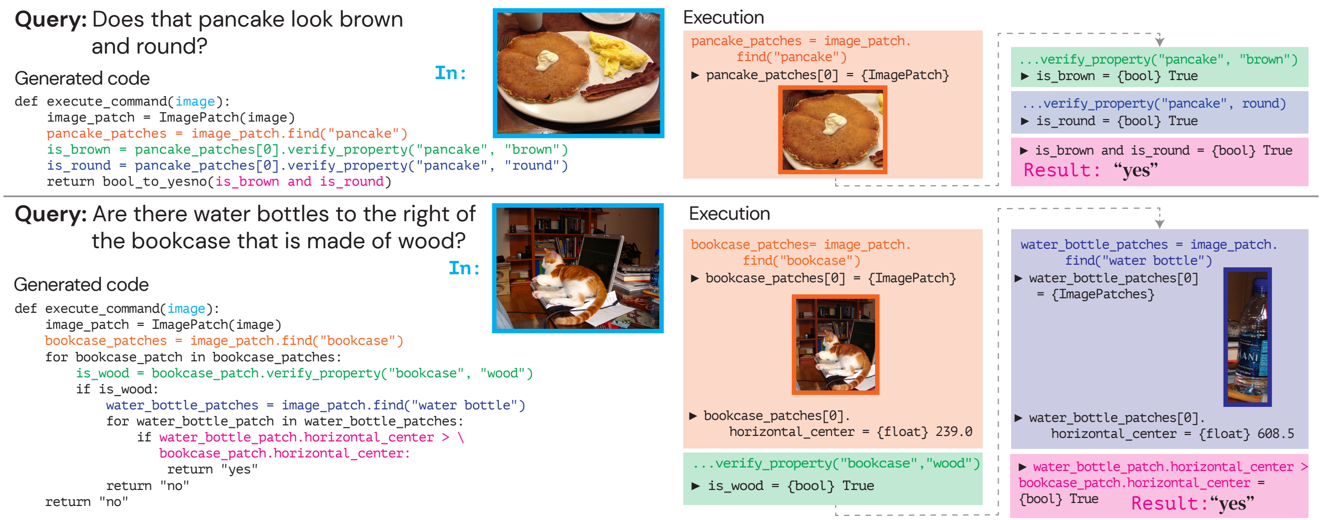

Figure 4. **Compositional image question answering on GQA.**

Analogous to dual-system models [28] 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.*, [27]) learns an execution engine ϕ as a neural module network, composing various modules implemented by neural networks. Their modules are responsible for not only perceptual functions such as `find`, but also logical ones such as `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 [48], 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 `sort`; control flow tools like `for` or `if/else`; and modules such as `datetime` or `math`. Notably, this does not require a custom interpreter, unlike prior approaches [17, 46] Another advantage of a fully Pythonic implementation is compatibility with a wide range of existing tools, such as PyTorch JIT [42].

In our implementation, each program in a generated batch is run simultaneously with multiprocessing. Our producer-consumer design [12] enables efficient GPU batching, reducing the memory and computation costs. Our code is made available at `viper.cs.columbia.edu/`.

Table 2. **GQA Results**. We report accuracy on the test-dev set.

		Accuracy (%) ↑
Sup.	LGCN [20]	55.8
	LXMERT [51]	60.0
	NSM [24]	63.0
	CRF [39]	72.1
ZS	BLIP-2 [30]	44.7
	`ViperGPT` (ours)	**48.1**

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.

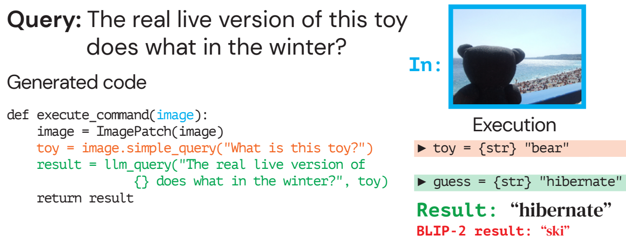

Figure 5. **Programmatic chain-of-thought with external knowledge for OK-VQA.**

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 [31]) 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 [31]) 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 [64]) takes as input an image, a noun phase 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 [64]) 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 [43]. Finally, **`compute_depth`** (MiDaS [44]) 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[2] in Figure 3. See Appendix A for more details about the experimental setup.

[2]Examples in the paper have been cosmetically cleaned by removing comments and error handling, but the logic is unchanged.

Table 3. **OK-VQA Results.**

		Accuracy (%) ↑
Sup.	TRiG [13]	50.5
	KAT [16]	54.4
	RA-VQA [32]	54.5
	REVIVE [33]	58.0
	PromptCap [21]	58.8
ZS	PNP-VQA [52]	35.9
	PICa [60]	43.3
	BLIP-2 [30]	45.9
	Flamingo [1]	50.6
	`ViperGPT` (ours)	**51.9**

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 [26], 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.

For GQA, we incorporate the module **`simple_query`** (BLIP-2 [31]), 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).

4.3. External Knowledge-dependent Image Question Answering

Many questions about images can only be answered correctly by integrating outside knowledge about the world. By equipping `ViperGPT` with a module to query external knowledge bases in natural language, it can combine knowledge with visual reasoning to handle such questions. We add a new module **`llm_query`** (GPT-3 [6]), which exploits text models as unstructured knowledge bases. We find that the combination of step-by-step reasoning from Codex along with external knowledge queried from GPT-3's text model achieves impressive performance in this setting.

We evaluate on the OK-VQA dataset [36], 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

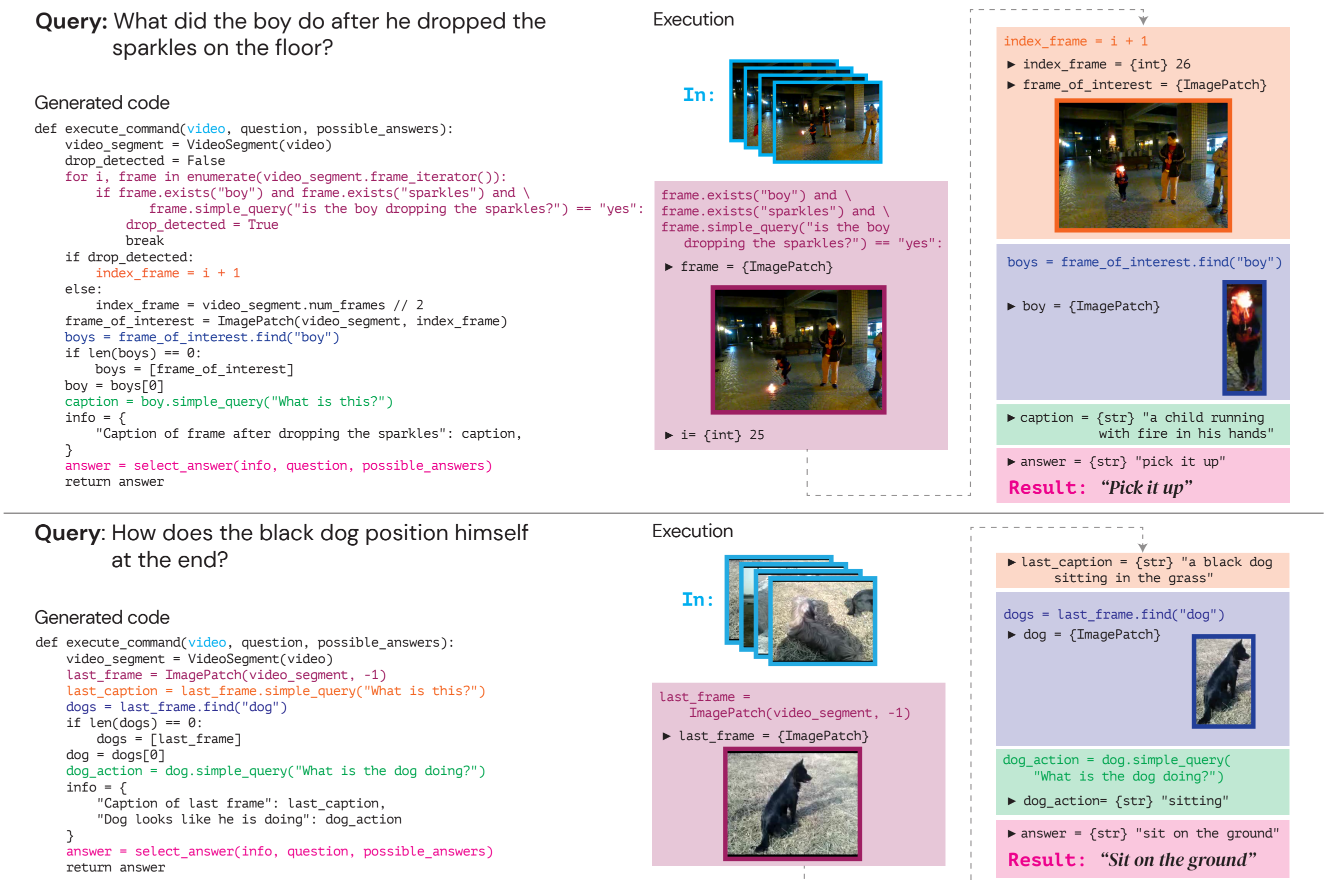

Figure 6. **Temporal reasoning on NeXT-QA.**

"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 [56] 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.

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.

		Accuracy (%) ↑		
		Hard Split - T	Hard Split - C	Full Set
Sup.	ATP [7]	45.3	43.3	54.3
	VGT [58]	-	-	56.9
	HiTeA [61]	48.6	47.8	63.1
ZS	`ViperGPT` (ours)	**49.8**	**56.4**	60.0

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

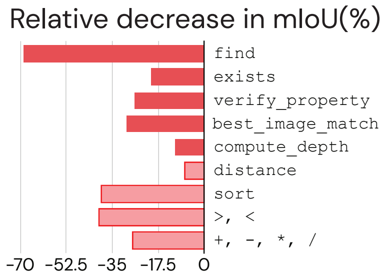

Figure 7. **Intervention.** We analyze the importance of various vision modules and Python functions in the generated programs as measured by the drop in mIoU when they are made nonfunctional.

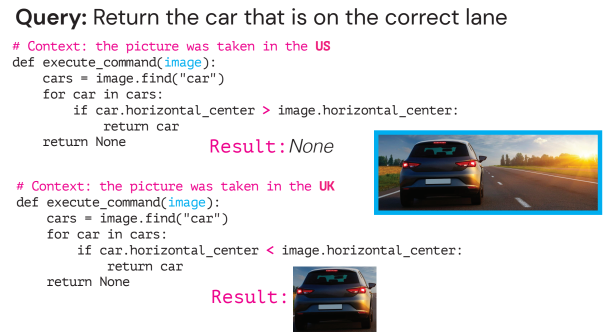

Figure 8. **Contextual programs**. ViperGPT readily incorporates additional context into the logic of the generated programs.

temporal reasoning when applied to videos provided as an ordered list of images. In particular, we observe it generates programs that 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 B 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 naively 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: 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

[1] Jean-Baptiste Alayrac, Jeff Donahue, Pauline Luc, Antoine Miech, Iain Barr, Yana Hasson, Karel Lenc, Arthur Mensch, Katherine Millican, Malcolm Reynolds, Roman Ring, Eliza Rutherford, Serkan Cabi, Tengda Han, Zhitao Gong, Sina Samangooei, Marianne Monteiro, Jacob Menick, Sebastian Borgeaud, Andrew Brock, Aida Nematzadeh, Sahand Sharifzadeh, Mikolaj Binkowski, Ricardo Barreira, Oriol Vinyals, Andrew Zisserman, and Karen Simonyan. Flamingo: a visual language model for few-shot learning. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[2] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[3] Dzmitry Bahdanau, Shikhar Murty, Michael Noukhovitch, Thien Huu Nguyen, Harm de Vries, and Aaron Courville. Systematic Generalization: What Is Required and Can It Be Learned?, Apr. 2019. arXiv:1811.12889 [cs].

[4] Yoshua Bengio. The Consciousness Prior, Dec. 2019. arXiv:1709.08568 [cs, stat].

[5] Maria A. Bravo, Sudhanshu Mittal, Simon Ging, and Thomas Brox. Open-vocabulary attribute detection. *arXiv preprint arXiv:2211.12914*, 2022.

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. *arXiv:2005.14165 [cs]*, July 2020. arXiv: 2005.14165.

[7] Shyamal Buch, Cristóbal Eyzaguirre, Adrien Gaidon, Jiajun Wu, Li Fei-Fei, and Juan Carlos Niebles. Revisiting the" video" in video-language understanding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2917–2927, 2022.

[8] Qingxing Cao, Xiaodan Liang, Bailin Li, and Liang Lin. Interpretable Visual Question Answering by Reasoning on Dependency Trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 43(3):887–901, Mar. 2021.

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura 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, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. 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. *ArXiv*, abs/2107.03374, 2021.

[10] Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

[11] Chaorui Deng, Qi Wu, Qingyao Wu, Fuyuan Hu, Fan Lyu, and Mingkui Tan. Visual Grounding via Accumulated Attention.

[12] E.W. Dijkstra. Information streams sharing a finite buffer. *Information Processing Letters*, 1(5):179–180, 1972.

[13] Feng Gao, Qing Ping, Govind Thattai, Aishwarya Reganti, Ying Nian Wu, and Prem Natarajan. Transform-retrieve-generate: Natural language-centric outside-knowledge visual question answering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5067–5077, 2022.

[14] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2022.

[15] Prajwal Gatti, Abhirama Subramanyam Penamakuri, Revant Teotia, Anand Mishra, Shubhashis Sengupta, and Roshni Ramnani. Cofar: Commonsense and factual reasoning in image search. In *Proceedings of the 2nd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 12th International Joint Conference on Natural Language Processing*, pages 1185–1199, 2022.

[16] Liangke Gui, Borui Wang, Qiuyuan Huang, Alexander Hauptmann, Yonatan Bisk, and Jianfeng Gao. KAT: A knowledge augmented transformer for vision-and-language. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 956–968, Seattle, United States, July 2022. Association for Computational Linguistics.

[17] Tanmay Gupta and Aniruddha Kembhavi. Visual programming: Compositional visual reasoning without training. *arXiv preprint arXiv:2211.11559*, 2022.

[18] Feijuan He, Yaxian Wang, Xianglin Miao, and Xia Sun. Interpretable visual reasoning: A survey. *Image and Vision Computing*, 112:104194, 2021.

[19] Ronghang Hu, Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Kate Saenko. Learning to Reason: End-to-End Module Networks for Visual Question Answering. *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 804–813, Oct. 2017. Conference Name: 2017 IEEE

International Conference on Computer Vision (ICCV) ISBN: 9781538610329 Place: Venice Publisher: IEEE.

[20] Ronghang Hu, Anna Rohrbach, Trevor Darrell, and Kate Saenko. Language-conditioned graph networks for relational reasoning. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 10294–10303, 2019.

[21] Yushi Hu, Hang Hua, Zhengyuan Yang, Weijia Shi, Noah A Smith, and Jiebo Luo. Promptcap: Prompt-guided task-aware image captioning. *arXiv preprint arXiv:2211.09699*, 2022.

[22] Ziniu Hu, Ahmet Iscen, Chen Sun, Zirui Wang, Kai-Wei Chang, Yizhou Sun, Cordelia Schmid, David A Ross, and Alireza Fathi. Reveal: Retrieval-augmented visual-language pre-training with multi-source multimodal knowledge memory. *arXiv preprint arXiv:2212.05221*, 2022.

[23] Shaohan Huang, Li Dong, Wenhui Wang, Yaru Hao, Saksham Singhal, Shuming Ma, Tengchao Lv, Lei Cui, Owais Khan Mohammed, Qiang Liu, et al. Language is not all you need: Aligning perception with language models. *arXiv preprint arXiv:2302.14045*, 2023.

[24] Drew Hudson and Christopher D Manning. Learning by abstraction: The neural state machine. *Advances in Neural Information Processing Systems*, 32, 2019.

[25] Drew A. Hudson and Christopher D. Manning. Compositional Attention Networks for Machine Reasoning. *ArXiv*, 2018.

[26] Drew A. Hudson and Christopher D. Manning. GQA: A New Dataset for Real-World Visual Reasoning and Compositional Question Answering, May 2019. arXiv:1902.09506 [cs].

[27] Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Judy Hoffman, Li Fei-Fei, C. Lawrence Zitnick, and Ross Girshick. Inferring and Executing Programs for Visual Reasoning. pages 2989–2998, 2017.

[28] Daniel Kahneman. *Thinking, fast and slow*. macmillan, 2011.

[29] Seung Wook Kim, Makarand Tapaswi, and Sanja Fidler. Visual reasoning by progressive module networks. In *International Conference on Learning Representations*, 2019.

[30] Junnan Li, Dongxu Li, Silvio Savarese, and Steven Hoi. BLIP-2: Bootstrapping Language-Image Pre-training with Frozen Image Encoders and Large Language Models, Jan. 2023. arXiv:2301.12597 [cs].

[31] Liunian Harold Li, Pengchuan Zhang, Haotian Zhang, Jianwei Yang, Chunyuan Li, Yiwu Zhong, Lijuan Wang, Lu Yuan, Lei Zhang, Jenq-Neng Hwang, et al. Grounded language-image pre-training. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 10965–10975, 2022.

[32] Weizhe Lin and Bill Byrne. Retrieval augmented visual question answering with outside knowledge. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 11238–11254, Abu Dhabi, United Arab Emirates, Dec. 2022. Association for Computational Linguistics.

[33] Yuanze Lin, Yujia Xie, Dongdong Chen, Yichong Xu, Chenguang Zhu, and Lu Yuan. REVIVE: Regional visual representation matters in knowledge-based visual question answering. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022.

[34] Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. Language models of code are few-shot commonsense learners. *arXiv preprint arXiv:2210.07128*, 2022.

[35] Chengzhi Mao, Revant Teotia, Amrutha Sundar, Sachit Menon, Junfeng Yang, Xin Wang, and Carl Vondrick. Doubly Right Object Recognition: A Why Prompt for Visual Rationales, Dec. 2022. arXiv:2212.06202 [cs].

[36] Kenneth Marino, Mohammad Rastegari, Ali Farhadi, and Roozbeh Mottaghi. OK-VQA: A Visual Question Answering Benchmark Requiring External Knowledge. May 2019.

[37] Sachit Menon and Carl Vondrick. Visual Classification via Description from Large Language Models, Dec. 2022. arXiv:2210.07183 [cs].

[38] Matthias Minderer, Alexey Gritsenko, Austin Stone, Maxim Neumann, Dirk Weissenborn, Alexey Dosovitskiy, Aravindh Mahendran, Anurag Arnab, Mostafa Dehghani, Zhuoran Shen, Xiao Wang, Xiaohua Zhai, Thomas Kipf, and Neil Houlsby. Simple open-vocabulary object detection with vision transformers. *arXiv preprint arXiv:2205.06230*, 2022.

[39] Binh X Nguyen, Tuong Do, Huy Tran, Erman Tjiputra, Quang D Tran, and Anh Nguyen. Coarse-to-fine reasoning for visual question answering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4558–4566, 2022.

[40] Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. *arXiv preprint arXiv:2205.12255*, 2022.

[41] Dong Huk Park, Lisa Anne Hendricks, Zeynep Akata, Anna Rohrbach, Bernt Schiele, Trevor Darrell, and Marcus Rohrbach. Multimodal Explanations: Justifying Decisions and Pointing to the Evidence. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8779–8788, Salt Lake City, UT, June 2018. IEEE.

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[43] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR, 2021.

[44] René Ranftl, Katrin Lasinger, David Hafner, Konrad Schindler, and Vladlen Koltun. Towards robust monocular depth estimation: Mixing datasets for zero-shot cross-dataset transfer. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(3), 2022.

[45] Revant Gangi Reddy, Xilin Rui, Manling Li, Xudong Lin, Haoyang Wen, Jaemin Cho, Lifu Huang, Mohit Bansal,

Avirup Sil, Shih-Fu Chang, et al. Mumuqa: Multimedia multi-hop news question answering via cross-media knowledge extraction and grounding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 11200–11208, 2022.

[46] 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 preprint arXiv:2302.04761*, 2023.

[47] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization. *International Journal of Computer Vision*, 128(2):336–359, Feb. 2020. arXiv: 1610.02391.

[48] Sanjay Subramanian, Ben Bogin, Nitish Gupta, Tomer Wolfson, Sameer Singh, Jonathan Berant, and Matt Gardner. Obtaining Faithful Interpretations from Compositional Neural Networks, Sept. 2020. arXiv:2005.00724 [cs].

[49] Sanjay Subramanian, Will Merrill, Trevor Darrell, Matt Gardner, Sameer Singh, and Anna Rohrbach. Reclip: A strong zero-shot baseline for referring expression comprehension. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics*, Dublin, Ireland, May 2022. Association for Computational Linguistics.

[50] Dídac Surís, Dave Epstein, Heng Ji, Shih-Fu Chang, and Carl. Vondrick. Learning to learn words from visual scenes. *European Conference on Computer Vision (ECCV)*, 2020.

[51] Hao Tan and Mohit Bansal. Lxmert: Learning cross-modality encoder representations from transformers. *arXiv preprint arXiv:1908.07490*, 2019.

[52] Anthony Meng Huat Tiong, Junnan Li, Boyang Li, Silvio Savarese, and Steven C.H. Hoi. Plug-and-play VQA: Zero-shot VQA by conjoining large pretrained models with zero training. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 951–967, Abu Dhabi, United Arab Emirates, Dec. 2022. Association for Computational Linguistics.

[53] Peng Wang, An Yang, Rui Men, Junyang Lin, Shuai Bai, Zhikang Li, Jianxin Ma, Chang Zhou, Jingren Zhou, and Hongxia Yang. Ofa: Unifying architectures, tasks, and modalities through a simple sequence-to-sequence learning framework. *CoRR*, abs/2202.03052, 2022.

[54] Xingyao Wang, Sha Li, and Heng Ji. Code4struct: Code generation for few-shot structured prediction from natural language. *arXiv preprint arXiv:2210.12810*, 2022.

[55] 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.

[56] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain of Thought Prompting Elicits Reasoning in Large Language Models, Oct. 2022. arXiv:2201.11903 [cs].

[57] Spencer Whitehead, Hui Wu, Heng Ji, Rogerio Feris, and Kate Saenko. Separating skills and concepts for novel visual question answering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5632–5641, June 2021.

[58] Junbin Xiao, Pan Zhou, Tat-Seng Chua, and Shuicheng Yan. Video graph transformer for video question answering. In *European Conference on Computer Vision*, pages 39–58. Springer, 2022.

[59] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, and Yoshua Bengio. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention, Apr. 2016. arXiv:1502.03044 [cs].

[60] Zhengyuan Yang, Zhe Gan, Jianfeng Wang, Xiaowei Hu, Yumao Lu, Zicheng Liu, and Lijuan Wang. An empirical study of gpt-3 for few-shot knowledge-based vqa. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 3081–3089, 2022.

[61] Qinghao Ye, Guohai Xu, Ming Yan, Haiyang Xu, Qi Qian, Ji Zhang, and Fei Huang. Hitea: Hierarchical temporal-aware video-language pre-training. *arXiv preprint arXiv:2212.14546*, 2022.

[62] Kexin Yi, Jiajun Wu, Chuang Gan, A. Torralba, Pushmeet Kohli, and J. Tenenbaum. Neural-Symbolic VQA: Disentangling Reasoning from Vision and Language Understanding. *ArXiv*, 2018.

[63] 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.

[64] Yan Zeng, Xinsong Zhang, and Hang Li. Multi-grained vision language pre-training: Aligning texts with visual concepts. *arXiv preprint arXiv:2111.08276*, 2021.

[65] Yundong Zhang, Juan Carlos Niebles, and Alvaro Soto. Interpretable Visual Question Answering by Visual Grounding from Attention Supervision Mining, Aug. 2018. arXiv:1808.00265 [cs].

A. Pretrained Models

We specify details about all the pretrained models used, as well as the code-generation large language model:

- **GLIP [31]**. We use the implementation from the official GitHub repository[3]. In our experiments we use the GLIP-L (large) version. In order to adapt to new versions of PyTorch, we had to modify the CUDA implementation of some functions, as the repository relies on old versions of PyTorch. We provide our updated version of GLIP in our code.
- **MiDaS [44]**. We use the implementation from PyTorch hub[4], and use the "DPT_Large" version.
- **BLIP-2 [30]**. We tried both the implementation from the official repository[5] and the Huggingface one[6], with little difference between the two, being the former slightly more performant and the latter faster. In both cases, we used the Flan-T5 XXL version.
- **X-VLM [64]**. We used the official implementation[7], specifically the version finetuned for retrieval on MSCOCO.
- **GPT-3 for `llm_query`**. The GPT-3 model we use for the LLM query function is the `text-davinci-003` one. We use the official OpenAI Python API[8].
- **Codex**. The GPT-3 model we use for code generation is the `code-davinci-002` one.

See the code for more detailed implementation details.

B. API

We provide the full API next, in Listing 1:

```python
class ImagePatch:
    """A Python class containing a crop of an image centered around a particular object, as well as relevant information.
    Attributes
    ----------
    cropped_image : array_like
        An array-like of the cropped image taken from the original image.
    left : int
        An int describing the position of the left border of the crop's bounding box in the original image.
    lower : int
        An int describing the position of the bottom border of the crop's bounding box in the original image.
    right : int
        An int describing the position of the right border of the crop's bounding box in the original image.
    upper : int
        An int describing the position of the top border of the crop's bounding box in the original image.

    Methods
    -------
    find(object_name: str)->List[ImagePatch]
        Returns a list of new ImagePatch objects containing crops of the image centered around any objects found in the
        image matching the object_name.
    exists(object_name: str)->bool
        Returns True if the object specified by object_name is found in the image, and False otherwise.
    verify_property(property: str)->bool
        Returns True if the property is met, and False otherwise.
    best_text_match(option_list: List[str], prefix: str)->str
        Returns the string that best matches the image.
    simple_query(question: str=None)->str
        Returns the answer to a basic question asked about the image. If no question is provided, returns the answer
        to "What is this?".
    compute_depth()->float
        Returns the median depth of the image crop.
    crop(left: int, lower: int, right: int, upper: int)->ImagePatch
        Returns a new ImagePatch object containing a crop of the image at the given coordinates.
    """

    def __init__(self, image, left: int=None, lower: int=None, right: int=None, upper: int=None):
        """Initializes an ImagePatch object by cropping the image at the given coordinates and stores the coordinates as attributes.
```

[3] https://github.com/microsoft/GLIP
[4] https://pytorch.org/hub/intelisl_midas_v2/
[5] https://github.com/salesforce/LAVIS/tree/main/projects/blip2
[6] https://huggingface.co/Salesforce/blip2-flan-t5-xxl
[7] https://github.com/zengyan-97/X-VLM
[8] https://openai.com/blog/openai-api

```
        If no coordinates are provided, the image is left unmodified, and the coordinates are set to the dimensions of the image.
        Parameters
        -------
        image : array_like
            An array-like of the original image.
        left : int
            An int describing the position of the left border of the crop's bounding box in the original image.
        lower : int
            An int describing the position of the bottom border of the crop's bounding box in the original image.
        right : int
            An int describing the position of the right border of the crop's bounding box in the original image.
        upper : int
            An int describing the position of the top border of the crop's bounding box in the original image.

        """
        if left is None and right is None and upper is None and lower is None:
            self.cropped_image = image
            self.left = 0
            self.lower = 0
            self.right = image.shape[2]  # width
            self.upper = image.shape[1]  # height
        else:
            self.cropped_image = image[:, lower:upper, left:right]
            self.left = left
            self.upper = upper
            self.right = right
            self.lower = lower

        self.width = self.cropped_image.shape[2]
        self.height = self.cropped_image.shape[1]

        self.horizontal_center = (self.left + self.right) / 2
        self.vertical_center = (self.lower + self.upper) / 2

    def find(self, object_name: str) -> List[ImagePatch]:
        """Returns a list of ImagePatch objects matching object_name contained in the crop if any are found.
        Otherwise, returns an empty list.
        Parameters
        ----------
        object_name : str
            the name of the object to be found

        Returns
        -------
        List[ImagePatch]
            a list of ImagePatch objects matching object_name contained in the crop

        Examples
        --------
        >>> # return the children
        >>> def execute_command(image) -> List[ImagePatch]:
        >>>     image_patch = ImagePatch(image)
        >>>     children = image_patch.find("child")
        >>>     return children
        """

    def exists(self, object_name: str) -> bool:
        """Returns True if the object specified by object_name is found in the image, and False otherwise.
        Parameters
        -------
        object_name : str
            A string describing the name of the object to be found in the image.

        Examples
        -------
        >>> # Are there both cakes and gummy bears in the photo?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     is_cake = image_patch.exists("cake")
        >>>     is_gummy_bear = image_patch.exists("gummy bear")
        >>>     return bool_to_yesno(is_cake and is_gummy_bear)
        """
        return len(self.find(object_name)) > 0

    def verify_property(self, object_name: str, property: str) -> bool:
        """Returns True if the object possesses the property, and False otherwise.
        Differs from 'exists' in that it presupposes the existence of the object specified by object_name, instead checking whether the object possesses the property.
        Parameters
```

```
        -------
        object_name : str
            A string describing the name of the object to be found in the image.
        property : str
            A string describing the property to be checked.

        Examples
        -------
        >>> # Do the letters have blue color?
        >>> def execute_command(image) -> str:
        >>>     image_patch = ImagePatch(image)
        >>>     letters_patches = image_patch.find("letters")
        >>>     # Question assumes only one letter patch
        >>>     if len(letters_patches) == 0:
        >>>         # If no letters are found, query the image directly
        >>>         return image_patch.simple_query("Do the letters have blue color?")
        >>>     return bool_to_yesno(letters_patches[0].verify_property("letters", "blue"))
        """
        return verify_property(self.cropped_image, object_name, property)

    def best_text_match(self, option_list: List[str]) -> str:
        """Returns the string that best matches the image.
        Parameters
        -------
        option_list : str
            A list with the names of the different options
        prefix : str
            A string with the prefixes to append to the options

        Examples
        -------
        >>> # Is the cap gold or white?
        >>> def execute_command(image)->str:
        >>>     image_patch = ImagePatch(image)
        >>>     cap_patches = image_patch.find("cap")
        >>>     # Question assumes one cap patch
        >>>     if len(cap_patches) == 0:
        >>>         # If no cap is found, query the image directly
        >>>         return image_patch.simple_query("Is the cap gold or white?")
        >>>     return cap_patches[0].best_text_match(["gold", "white"])
        """
        return best_text_match(self.cropped_image, option_list)

    def simple_query(self, question: str = None) -> str:
        """Returns the answer to a basic question asked about the image. If no question is provided, returns the answer to "What is this?".
        Parameters
        -------
        question : str
            A string describing the question to be asked.

        Examples
        -------

        >>> # Which kind of animal is not eating?
        >>> def execute_command(image) -> str:
        >>>     image_patch = ImagePatch(image)
        >>>     animal_patches = image_patch.find("animal")
        >>>     for animal_patch in animal_patches:
        >>>         if not animal_patch.verify_property("animal", "eating"):
        >>>             return animal_patch.simple_query("What kind of animal is eating?") # crop would include eating so keep it in the query
        >>>     # If no animal is not eating, query the image directly
        >>>     return image_patch.simple_query("Which kind of animal is not eating?")

        >>> # What is in front of the horse?
        >>> # contains a relation (around, next to, on, near, on top of, in front of, behind, etc), so ask directly
        >>> return image_patch.simple_query("What is in front of the horse?")
        >>>
        """
        return simple_qa(self.cropped_image, question)

    def compute_depth(self):
        """Returns the median depth of the image crop
        Parameters
        ----------
        Returns
        -------
        float
            the median depth of the image crop

```

```python
        Examples
        --------
        >>> # the person furthest away
        >>> def execute_command(image)->ImagePatch:
        >>>     image_patch = ImagePatch(image)
        >>>     person_patches = image_patch.find("person")
        >>>     person_patches.sort(key=lambda person: person.compute_depth())
        >>>     return person_patches[-1]
        """
        depth_map = compute_depth(self.cropped_image)
        return depth_map.median()

    def crop(self, left: int, lower: int, right: int, upper: int) -> ImagePatch:
        """Returns a new ImagePatch cropped from the current ImagePatch.
        Parameters
        -------
        left : int
            The leftmost pixel of the cropped image.
        lower : int
            The lowest pixel of the cropped image.
        right : int
            The rightmost pixel of the cropped image.
        upper : int
            The uppermost pixel of the cropped image.
        -------
        """
        return ImagePatch(self.cropped_image, left, lower, right, upper)

    def overlaps_with(self, left, lower, right, upper):
        """Returns True if a crop with the given coordinates overlaps with this one,
        else False.
        Parameters
        ----------
        left : int
            the left border of the crop to be checked
        lower : int
            the lower border of the crop to be checked
        right : int
            the right border of the crop to be checked
        upper : int
            the upper border of the crop to be checked

        Returns
        -------
        bool
            True if a crop with the given coordinates overlaps with this one, else False

        Examples
        --------
        >>> # black cup on top of the table
        >>> def execute_command(image) -> ImagePatch:
        >>>     image_patch = ImagePatch(image)
        >>>     table_patches = image_patch.find("table")
        >>>     if len(table_patches) == 0:
        >>>         table_patches = [image_patch]  # If no table found, assume the whole image is a table
        >>>     table_patch = table_patches[0]
        >>>     cup_patches = image_patch.find("black cup")
        >>>     for cup in cup_patches:
        >>>         if cup.vertical_center > table_patch.vertical_center
        >>>             return cup
        >>>     return cup_patches[0]  # If no cup found on top of the table, return the first cup found
        """
        return self.left <= right and self.right >= left and self.lower <= upper and self.upper >= lower


def best_image_match(list_patches: List[ImagePatch], content: List[str], return_index=False) -> Union[ImagePatch, int]:
    """Returns the patch most likely to contain the content.
    Parameters
    ----------
    list_patches : List[ImagePatch]
    content : List[str]
        the object of interest
    return_index : bool
        if True, returns the index of the patch most likely to contain the object

    Returns
    -------
    int
        Patch most likely to contain the object
```

```
    Examples
    --------
    >>> # Return the man with the hat
    >>> def execute_command(image):
    >>>     image_patch = ImagePatch(image)
    >>>     man_patches = image_patch.find("man")
    >>>     if len(man_patches) == 0:
    >>>         return image_patch
    >>>     hat_man = best_image_match(list_patches=man_patches, content=["hat"])
    >>>     return hat_man

    >>> # Return the woman with the pink scarf and blue pants
    >>> def execute_command(image):
    >>>     image_patch = ImagePatch(image)
    >>>     woman_patches = image_patch.find("woman")
    >>>     if len(woman_patches) == 0:
    >>>         return image_patch
    >>>     woman_most = best_image_match(list_patches=woman_patches, content=["pink scarf", "blue pants"])
    >>>     return woman_most
    """
    return best_image_match(list_patches, content, return_index)


def distance(patch_a: ImagePatch, patch_b: ImagePatch) -> float:
    """
    Returns the distance between the edges of two ImagePatches. If the patches overlap, it returns a negative distance
    corresponding to the negative intersection over union.
    """
    return distance(patch_a, patch_b)


def bool_to_yesno(bool_answer: bool) -> str:
    return "yes" if bool_answer else "no"


def llm_query(question: str) -> str:
    '''Answers a text question using GPT-3. The input question is always a formatted string with a variable in it.

    Parameters
    ----------
    question: str
        the text question to ask. Must not contain any reference to 'the image' or 'the photo', etc.
    '''
    return llm_query(question)


class VideoSegment:
    """A Python class containing a set of frames represented as ImagePatch objects, as well as relevant information.
    Attributes
    ----------
    video : torch.Tensor
        A tensor of the original video.
    start : int
        An int describing the starting frame in this video segment with respect to the original video.
    end : int
        An int describing the ending frame in this video segment with respect to the original video.
    num_frames->int
        An int containing the number of frames in the video segment.

    Methods
    -------
    frame_iterator->Iterator[ImagePatch]
    trim(start, end)->VideoSegment
        Returns a new VideoSegment containing a trimmed version of the original video at the [start, end] segment.
    select_answer(info, question, options)->str
        Returns the answer to the question given the options and additional information.
    """

    def __init__(self, video: torch.Tensor, start: int = None, end: int = None, parent_start=0, queues=None):
        """Initializes a VideoSegment object by trimming the video at the given [start, end] times and stores the
        start and end times as attributes. If no times are provided, the video is left unmodified, and the times are
        set to the beginning and end of the video.

        Parameters
        -------
        video : torch.Tensor
            A tensor of the original video.
        start : int
```

```
            An int describing the starting frame in this video segment with respect to the original video.
        end : int
            An int describing the ending frame in this video segment with respect to the original video.
        """

        if start is None and end is None:
            self.trimmed_video = video
            self.start = 0
            self.end = video.shape[0]  # duration
        else:
            self.trimmed_video = video[start:end]
            if start is None:
                start = 0
            if end is None:
                end = video.shape[0]
            self.start = start + parent_start
            self.end = end + parent_start

        self.num_frames = self.trimmed_video.shape[0]

    def frame_iterator(self) -> Iterator[ImagePatch]:
        """Returns an iterator over the frames in the video segment."""
        for i in range(self.num_frames):
            yield ImagePatch(self.trimmed_video[i], self.start + i)

    def trim(self, start: Union[int, None] = None, end: Union[int, None] = None) -> VideoSegment:
        """Returns a new VideoSegment containing a trimmed version of the original video at the [start, end]
        segment.

        Parameters
        ----------
        start : Union[int, None]
            An int describing the starting frame in this video segment with respect to the original video.
        end : Union[int, None]
            An int describing the ending frame in this video segment with respect to the original video.

        Examples
        --------
        >>> # Return the second half of the video
        >>> def execute_command(video):
        >>>     video_segment = VideoSegment(video)
        >>>     video_second_half = video_segment.trim(video_segment.num_frames // 2, video_segment.num_frames)
        >>>     return video_second_half
        """
        if start is not None:
            start = max(start, 0)
        if end is not None:
            end = min(end, self.num_frames)

        return VideoSegment(self.trimmed_video, start, end, self.start)

    def select_answer(self, info: dict, question: str, options: List[str]) -> str:
        return select_answer(self.trimmed_video, info, question, options)

    def __repr__(self):
        return "VideoSegment({}, {})".format(self.start, self.end)
```

Listing 1. **Full API.**

Not all methods are used in all the benchmarks. Next we describe in more detail what content is used for the API specifications for every benchmark.

- **RefCOCO and RefCOCO+**. We use all the methods from the `ImagePatch` class except for `best_text_match` and `simple_query`. We also use the `best_text_match` and `distance` functions. Additionally we add `ImagePatch` usage examples in the API definition that are representative of the RefCOCO dataset, and look like the following:

```
# chair at the front
def execute_command(image) -> ImagePatch:
    # Return the chair
    image_patch = ImagePatch(image)
    chair_patches = image_patch.find("chair")
    chair_patches.sort(key=lambda chair: chair.compute_depth())
    chair_patch = chair_patches[0]
    # Remember: return the chair
    return chair_patch
```

Listing 2. **RefCOCO example.**

- **GQA**. The GQA API contains all the contents in the API from Listing 1 up until the `llm_query` function, which is not used. The `ImagePatch` usage examples look like the following:

```
# Is there a backpack to the right of the man?
def execute_command(image)->str:
    image_patch = ImagePatch(image)
    man_patches = image_patch.find("man")
    # Question assumes one man patch
    if len(man_patches) == 0:
        # If no man is found, query the image directly
        return image_patch.simple_query("Is there a backpack to the right of the man?")
    man_patch = man_patches[0]
    backpack_patches = image_patch.find("backpack")
    # Question assumes one backpack patch
    if len(backpack_patches) == 0:
        return "no"
    for backpack_patch in backpack_patches:
        if backpack_patch.horizontal_center > man_patch.horizontal_center:
            return "yes"
    return "no"
```

Listing 3. **GQA example.**

- **OK-VQA**. The API only uses the `simple_query` method from `ImagePatch`. It additionally uses the `llm_query` function. The `ImagePatch` usage examples look like the following:

```

# Who is famous for allegedly doing this in a lightning storm?
def execute_command(image)->str:
    # The question is not direct perception, so we need to ask the image for more information
    # Salient information: what is being done?
    image = ImagePatch(image)
    guesses = []
    action = image.simple_query("What is being done?")
    external_knowledge_query = "Who is famous for allegedly {} in a lightning storm?".format(action)
    step_by_step_guess = llm_query(external_knowledge_query)
    guesses.append("what is being done is {}".format(action) + ", so " + step_by_step_guess)
    direct_guess = image.simple_query("Who is famous for allegedly doing this in a lightning storm?")
    guesses.append(direct_guess)
    return process_guesses("Who is famous for allegedly doing this in a lightning storm?", guesses)
```

Listing 4. **OK-VQA example.**

- **NeXT-QA**. The `VideoSegment` class is added to the API definition, and the available `ImagePatch` methods are `find`, `exists`, `best_text_match` and `simple_query`. The function `best_image_match` is also used. The `ImagePatch` usage examples look like:

```
# why does the man with a red hat put his arm down at the end of the video
# possible answers: ['watching television', 'searching for food', 'move its head', 'looking over cardboard box', 'looks at the camera']
def execute_command(video, possible_answers, question)->[str, dict]:
    # Reason every step
    video_segment = VideoSegment(video)
    # Caption last frame of the video (end of video)
    last_frame = ImagePatch(video_segment, -1)
    last_caption = last_frame.simple_query("What is this?")
    men = last_frame.find("man")
    if len(men) == 0:
        men = [last_frame]
    man = men[0]
    man_action = man.simple_query("What is the man doing?")
    # Answer the question. Remember to create the info dictionary
    info = {
        "Caption of last frame": last_caption,
        "Man looks like he is doing": man_action
    }
    answer = video_segment.select_answer(info, question, possible_answers)
    return answer, info
```

Listing 5. **NeXT-QA example.**

- **Beyond benchmarks**. For the examples in Figure 1 we use the same API as the one used for the benchmarks, and the usage examples are taken from the benchmark APIs, combining them to have more generality. We do not add any other example, ViperGPT generalizes to the complex cases shown in Figure 1 just based on the provided API.

Note that in some of the examples we added comments, as well as error handling. The generated code also contains similar lines. We removed those for clarity in the figures shown in the main paper.