論文 / 著書情報
Article / Book Information

Title	Pyramid Coder: Hierarchical Code Generator for Compositional Visual Question Answering
Author	Ruoyue Shen, Nakamasa Inoue, Koichi Shinoda
Journal/Book name	2024 IEEE International Conference on Image Processing (ICIP), , , pp. 430-436
Pub. date	2024, 9
DOI	https://doi.org/10.1109/ICIP51287.2024.10648180
Copyright	(c)2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
Note	This file is author (final) version.

PYRAMID CODER: HIERARCHICAL CODE GENERATOR FOR COMPOSITIONAL VISUAL QUESTION ANSWERING

Ruoyue Shen, Nakamasa Inoue, Koichi Shinoda

Institute of Science Tokyo

ABSTRACT

Visual question answering (VQA) is the task of providing accurate answers to natural language questions based on visual input. Programmatic VQA (PVQA) models have been gaining attention recently. These use large language models (LLMs) to formulate executable programs that address questions requiring complex visual reasoning. However, there are challenges in enabling LLMs to comprehend the usage of image processing modules and generate relevant code. To overcome these challenges, this paper introduces PyramidCoder, a novel prompting framework for PVQA models. PyramidCoder consists of three hierarchical levels, each serving a distinct purpose: query rephrasing, code generation, and answer aggregation. Notably, PyramidCoder utilizes a single frozen LLM and pre-defined prompts at each level, eliminating the need for additional training and ensuring flexibility across various LLM architectures. Compared to the state-of-the-art PVQA model, our approach improves accuracy by at least 0.5% on the GQA dataset, 1.4% on the VQAv2 dataset, and 2.9% on the NLVR2 dataset.

***Index Terms*—** Visual question answering, Large language models, Code generation, Prompting methods.

1. INTRODUCTION

Visual question answering (VQA), which aims to provide accurate answers to natural language questions based on visual input, is a crucial research topic in computer vision and natural language processing [1, 2, 3]. The last decade has seen significant advances in deep learning applied to VQA, with the development of end-to-end multimodal models such as GLIP [4] and PNP-VQA [5]. However, despite these advances, compositional VQA, which involves complex spatial relationships and object attributes, remains difficult due to the lack of an explicit understanding of visual elements during the inference process.

To address the difficulties in compositional VQA, recent studies [6, 7, 8] have proposed models that generate executable programs to answer questions based on given images, questions, and image processing modules. These models, which we refer to as Programmatic VQA (PVQA) models, represent a novel approach to integrating multimodal inputs, leading to more tractable and traceable inference. Typically, PVQA models consist of three components: a large language model (LLM) for code generation, a set of image processing modules, and a Python executor. The LLM analyzes a given query (text-form question) and generates Python code to answer it using a predefined API set of image processing modules, including both low-level modules such as image cropping and high-level modules such as object detection. The generated code is then executed by the Python executor to produce the answer. This architecture allows questions to be decomposed into manageable segments of code, providing a more precise approach to compositional VQA. However, enabling the LLM to understand API usage and generate appropriate code for VQA is not always straightforward.

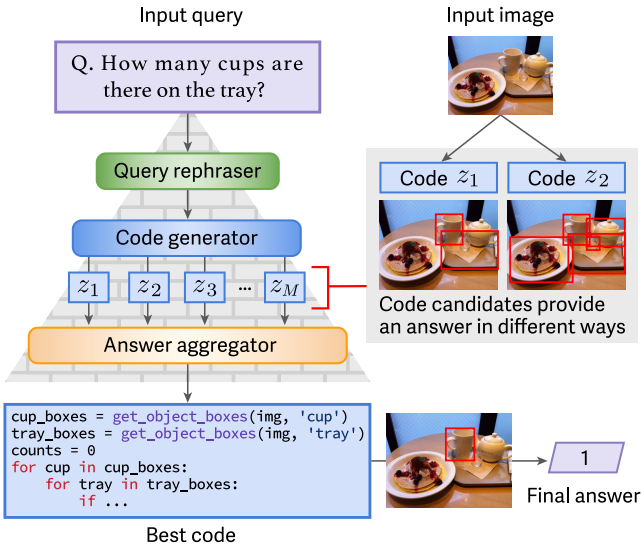


Fig. 1. PyramidCoder is a hierarchical framework comprising three modules. It generates multiple code candidates, maximizing the diversity of solutions available for a given input query to produce the correct answer effectively.

To fully leverage the capabilities of LLMs, a number of prompting frameworks have been developed for natural language processing tasks. Chain-of-Thought (CoT) [9], for example, demonstrates how structured prompts can guide LLMs to decompose complex problems into a series of simpler, linked steps. There are also extensions of CoT such as Tree

of Thoughts [10], which creates a tree structure of thinking steps. These frameworks not only improve performance, but also provide greater transparency into their reasoning process.

Inspired by these approaches, this paper introduces PyramidCoder, a novel code generation prompting framework for PVQA models. Previous PVQA models [6, 7, 8] employ Input-Output (IO) prompting to generate code from prompt input directly. However, IO prompting constraints a singular solution for a question, resulting in a simplistic and uniform code generation. In contrast, PyramidCoder explores the diversity of question statements and possible solutions through a hierarchical framework consisting of three levels. As shown in Figure 1, the first level rephrases a given query into multiple variations. The second level then generates multiple code candidates corresponding to the rephrased queries. Finally, the third level aggregates them and generates the final code and answer for the input query. All these procedures are implemented with a single frozen LLM and pre-defined prompts at each level, without the need for additional training. In experiments, we substantiate the efficacy of PyramidCoder across GQA [1], VQAv2 [2], and NLVR2 [3] datasets, showing its ability to significantly enhance VQA performance. In summary, our contribution is three-fold:

1. We introduce PyramidCoder, a novel code generation framework for PVQA consisting of three modules: a query rephraser, a code generator, and an answer aggregator.
2. We propose a prompting method to implement all modules with a single frozen LLM to avoid additional training. This approach offers flexibility and is not constrained by the specific LLM employed.
3. We demonstrate the effectiveness of our method over the state-of-the-art CodeVQA [8] model by conducting experiments on three VQA datasets.

2. RELATED WORK

2.1. Visual Question Answering

VQA is an interdisciplinary research area combining computer vision and natural language processing. It aims to develop AI systems that respond to queries about images by integrating multimodal information. Early VQA models relied on convolutional neural networks to extract image features and recurrent neural networks to process textual inputs [11, 12, 13]. The introduction of attention mechanisms [14, 15, 16, 17] has since improved these models' capacity to handle complex questions and comprehend fine-grained image details. Recently, multimodal pretaining approaches [4, 18, 19] have been employed, leveraging large-scale pretraining on both image and text data to capture richer contextual information.

However, fine-tuning multimodal pre-trained models for VQA requires substantial expertise, vast amounts of data, and significant computational resources. PVQA models [6, 7, 8] address these issues by utilizing an LLM to generate a Python-like code, which is then executed with predefined APIs to find answers to questions. These models can quickly adapt to new tasks or domains with minimal data, without the need for additional training or fine-tuning. Despite these advantages, PVQA models using IO prompting display a level of simplicity that hampers the effective utilization of LLM capabilities. Consequently, methods for activating the latent potential of LLMs and guiding them towards optimal API utilization remain an underexplored domain.

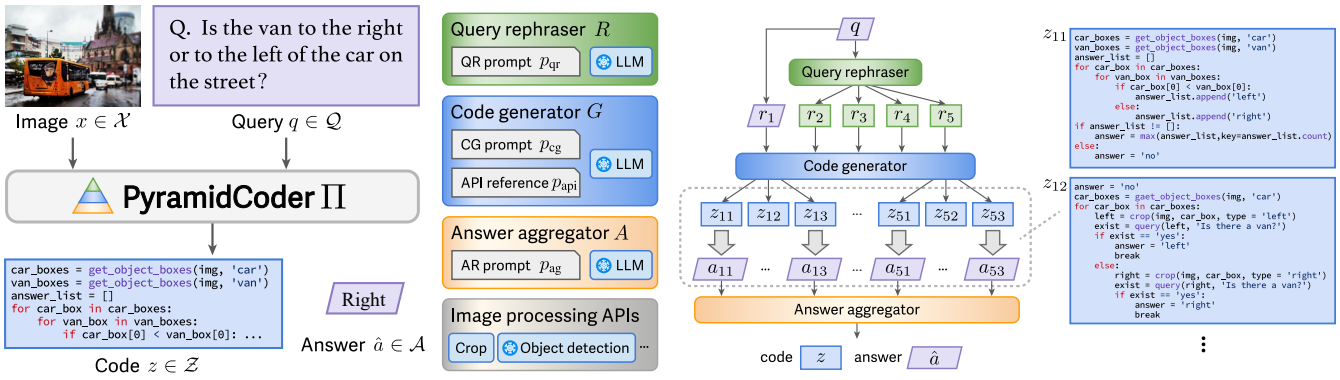
2.2. Large Language Models

Code Generation Models. Large language models have demonstrated remarkable capabilities in a wide range of natural language processing tasks. While general LLMs find application in various tasks [20, 21], a subset of these models has been specifically tailored for code generation. This specialized category of LLMs undergoes training on extensive corpora of programming-related text, enabling them to comprehend and generate code in response to natural language descriptions. Codex [22], an extension of the GPT-3 series, demonstrates proficiency in over a dozen languages. StarCoder [23], as one of the pioneering open-source Code LLMs, undergoes training on datasets sourced from Github licensed data, covering over 80 programming languages. CodeLlama [24], fine-tuned from Llama2 with a higher sampling of code, augments support for larger input contexts and zero-shot instruction following ability. Noteworthy among recent developments are native multimodality LLMs such as Gemini [25]. These models leverage multimodal data (i.e., text, images, and code) throughout the entirety of the training process to enhance their proficiency in generating code that meets specific requirements.

Prompting. Prompting methods are employed to optimize prompts for LLMs in order to maximize their capacity. Chain-of-Thought (CoT) [9] guides LLMs in generating intermediate reasoning steps before predicting the answer. This method has inspired several extensions. Notably, Self-consistency [26] samples from the LLM decoder multiple times and aggregates the final answer through majority voting. AutoCoT [27] automatically constructs demonstrations through question clustering and zero-shot CoT reasoning chain generation. ReAct [28] alternately generates thoughts and actions, performs the action based on the thought, and adjusts the thought based on the action result. Similar to CoT prompting and its variations, Tree of Thoughts (ToT) [10] decomposes problems into smaller thoughts and explores multiple solution paths in parallel, forming a tree structure of thoughts.

3. PROPOSED METHOD

This section introduces PyramidCoder, a hierarchical code generation framework for PVQA. As shown in Figure 2, PyramidCoder generates a Python code to answer a question given



(a) Inputs and outputs of PyramidCoder (b) Modules (c) Pyramid structure with candidate codes

Fig. 2. Framework overview. (a) PyramidCoder Π takes an image x and a query q as input, producing both the code and answer for the input image-query pair. (b) PyramidCoder contains three modules: a query rephraser R, a code generator G, and an answer aggregator A. Image processing APIs are also used in the code-executing process. (c) For each input query, the PyramidCoder generates diverse expressions r_i of the input query and multiple code candidates z_{ij}, each representing a distinct solution to the given query. Finally, these codes z_{ij} and their corresponding pre-execution results a_{ij} are aggregated to derive the final answer $\hat{a}$ and the accompanying interpretable code z.

a query, an image, and a set of pre-defined APIs for simple processing tasks. The framework consists of three modules: a query rephraser R, a code generator G, and an answer aggregator A. These three modules are implemented with a single frozen LLM and work in conjunction to enhance the quality of code generation within the pyramid structure.

3.1. Overall Framework

Problem setting. We follow the notation used in previous studies [6, 8, 29]. Let $x \in \mathcal{X}$ be an input image and $q \in \mathcal{Q}$ be a textual query, where $\mathcal{X}$ is a set of images and $\mathcal{Q}$ is a set of textual queries. The goal is to generate a code $z \in \mathcal{Z}$ that returns the answer $a \in \mathcal{A}$ to the query given the image, where $\mathcal{Z}$ is a set of executable codes and $\mathcal{A}$ is a set of answers. This study assumes that $\mathcal{Z}$ is a set of Python codes. Note that the three sets $\mathcal{Q}, \mathcal{Z}$ and $\mathcal{A}$ are subsets of the whole set of texts, which we denote as $\mathcal{T}$. We use a frozen LLM $\pi : \mathcal{T} \to \mathcal{T}$ to find mappings between these subsets.

Framework. The PyramidCoder framework consists of two stages: the coding stage and the execution stage. The coding stage generates a code as

$$z = \Pi(q, x), \tag{1}$$

where $\Pi : \mathcal{Q} \times \mathcal{X} \to \mathcal{Z}$ is the code generation process. In the execution stage, the generated code is executed with the input image to obtain an answer as

$$a = \Lambda(x, z), \tag{2}$$

where a is the predicted answer and $\Lambda : \mathcal{X} \times \mathcal{Z} \to \mathcal{A}$ is the Python execution engine. This paper studies the design of the code generation process Π.

Algorithm. The proposed code generation process Π is summarized in Algorithm 1. It consists of three steps. First, given an input query $q \in \mathcal{Q}$, the query rephraser R is applied to obtain a set of rephrased queries $\{r_i\}_{i=1}^{N}$. Here, each r_i is a rephrased version of q, intended to make coding more effective with a more comprehensive understanding of the input query. Second, the code generator G is applied to each rephrased query. This step produces multiple candidate codes $\{z_{ij}\}_{j=1}^{M}$ for each rephrased query r_i and pre-executes them to obtain candidate answers $a_{ij} = \Lambda(x, z_{ij})$. Finally, the answer aggregator A is applied to the set of candidate code-answer pairs. This step chooses the final answer a_τ and the best code z_τ, where τ is the index of the chosen pair. As shown in Figure 2, PyramidCoder implements all modules with a single frozen LLM, and the code generation process is represented by a pyramid structure of texts (queries and codes) generated by the LLM.

3.2. Module implementation with a single LLM

Given a frozen LLM $\pi : \mathcal{T} \to \mathcal{T}$ pre-trained on a large text dataset comprising both natural language and programming code, we implement the query rephraser R, the code generator G and the answer aggregator A with the aforementioned LLM without any additional training. Below, we describe the motivation and definition of each module.

3.2.1. Query Rephraser

Motivation. The query rephraser module in PyramidCoder is designed to enhance the interpretability and robustness of the model's comprehension of input queries. The VQA task demands a precise understanding of diverse query expressions, as users may articulate questions in various linguistic forms. It is widely recognized that the input prompt, including input queries, significantly influences the response quality of LLMs.

Algorithm 1 PyramidCoder $\Pi(q, x)$

Input: Query $q \in \mathcal{Q}$, Image $x \in \mathcal{X}$, LLM π
Output: Code $z \in \mathcal{Z}$, Answer $\hat{a} \in \mathcal{A}$

```
Z ← ∅
[r_1, r_2, ··· , r_N] ← R(q) # Query rephraser
for i = 1, ··· , N do
    [z_i1, z_i2, ··· , z_iM] ← G(r_i) # Code generator
    for j = 1, ··· , M do
        a_ij ← Λ(x, z_ij) # Python executor
        Z ← Z ∪ {(z_ij, a_ij)}
    end for
end for
σ ← A_ans(Z) # Answer aggregator
τ ← A_code(Z_σ) # Answer aggregator
return Code z = z_τ, Answer â = a_τ
```

The query rephraser module dynamically reformulates an input query into diverse expressions while preserving the underlying semantics. Consequently, the subsequent code generation stage benefits from a more comprehensive and adaptable understanding of user queries, thereby alleviating potential misinterpretations and improving overall task performance.

Definition. We define the query rephraser by

$$R(q) \triangleq \pi(p_{\text{qr}} + s_{\text{qr}} + q), \tag{3}$$

where q is an input query, p_{qr} is the query rephraser prompt as shown in Figure 3a, and s_{qr} is the few-shot examples prompt used for rephrasing. The operation $+$ indicates the concatenation of texts.

3.2.2. *Code Generator*

Motivation. The development of the code generator module is driven by the recognition that complex problem-solving often involves an inherent diversity and multiplicity of solutions. Traditional PVQA models [6, 7, 8] predominantly focus on generating a singular Python code for a given input query. However, this conventional methodology overlooks the intrinsic variability in problem-solving strategies and fails to capture the diversity of viable solutions that may lead to the correct answer. By introducing the capability to generate multiple Python codes for a single input query, the code generator aims to make full use of potential solutions associated with a given query. This approach is grounded in the understanding that various coding strategies may be employed to achieve the same desired outcome, fostering a more nuanced and adaptable code generation in complex problem-solving domains.

Definition. We define the code generator by

$$G(r) \triangleq \pi(p_{\text{cg}} + p_{\text{api}} + s_{\text{cg}} + r), \tag{4}$$

where r is a rephrased query, p_{cg} is the code generator prompt in Figure 3b, p_{api} is API reference texts detailed in Figure 4, and s_{cg} is the few-shot examples prompt for code generation.

(a) Query Rephraser Prompt p_{qr}

```
Rephrase the input question so that it has different
expressions. You can build different versions by
changing the word order of sentences, using clauses,
etc. The question is asking about the content of an
image, make sure the meaning of the question is not
changed. Maximum {n_queries} sentences.
```

(b) Code Generator Prompt p_{cg}

```
You will be provided with a question asking about an
image. Write up to {m_codes} versions of Python code to
answer this question.
- Only use the constants and APIs delimited by triple
  backticks and Python grammar.
- Make sure the code string keeps the correct 4 spaces
  indentation of Python.
- Make sure all the variables are defined before use.
- Make sure all the parameters are given when calling
  a function.
- Only output the codes in list format:["code1","code2"]
- Do not output other explainable words.
```

(c) Answer Aggregator Prompt for Answer Selection p_{aga}

```
You will be provided with a question asking about an
image. Multiple codes have been executed to generate
answers for the question, which are collected in an
answer list. Your task is to choose the correct answer.
- Evaluate each answer based on its relevance to the
  question and its frequency.
- You must choose one answer, and if you have no idea,
  choose the most frequently occurring one.
```

(d) Answer Aggregator Prompt for Code Selection p_{agc}

```
Given a question related to an image, you will receive
multiple versions of Python code as potential choices.
Your task is to select the most clear and user-friendly
code.
- Choose only one code from the provided list and return
  its index.
- Consider simplicity for straightforward questions and
  complexity for more challenging ones.
- The selected code should be logically correct to solve
  the question and run without errors.
```

Fig. 3. Prompt definitions.

3.2.3. *Answer Aggregator*

Motivation. The traditional approach often employs majority voting to aggregate solutions. However, this approach encounters challenges when failed code execution leads to the default answer being the majority or when multiple answers achieve majority status. In response to these limitations, the answer aggregator utilizes the capabilities of LLMs to select the final output, considering not only the frequency but also the semantic compatibility between the query and potential answers. The answer aggregator operates in two steps. First, selecting the final answer from the candidate answer set, and then selecting the most suitable code that produced this final answer during pre-execution. This sequential approach minimizes input token consumption and alleviates potential mismatches between answers and their corresponding code that might be caused by hallucinations.

Definition. Given a set of candidate answers and codes $Z = \{(z_k, a_k)\}_{k=1}^{O}$, we define the answer aggregator by

$$A_{\text{ans}}(Z) \triangleq \pi(p_{\text{aga}} + [a_1, a_2, \cdots, a_O]), \tag{5}$$

$$A_{\text{code}}(Z_{\boldsymbol{\sigma}}) \triangleq \pi(p_{\text{agc}} + [z_{\sigma_1}, z_{\sigma_2}, \cdots, z_{\sigma_l}]), \tag{6}$$

where p_{aga} is the prompt used by the answer aggregator to select the final answer shown in Figure 3c, and p_{agc} is the

Method	Code	LLM	GQA val2000	GQA testdev	VQAv2 val4000	NLVR2 test
CodeVQA [8]	✓	StarCoder-Base	46.5	41.0	56.4	53.5
PyramidCoder (Ours)	✓	StarCoder-Base	**48.2**	**41.5**	**57.8**	**59.2**
CodeVQA [8]	✓	CodeLlama-7b-Python	48.0	42.8	61.3	57.8
PyramidCoder (Ours)	✓	CodeLlama-7b-Python	**51.3**	**43.4**	**62.8**	**60.7**
Few-shot PnP-VQA [5]	–	–	55.1	46.6	66.8	63.4

Table 1. Comparison with SOTA methods for compositional visual question answering. Accuracy (%) on three datasets is reported. Code: ✓ indicates code generation methods. LLM: Model names of large language models.

prompt to choose the best code shown in Figure 3d. The operation [] represents the aggregation of elements into a list and the conversion to string format. $Z_{\boldsymbol{\sigma}} = \{z_{\sigma_l}\}_{l=1}^{L}$ is a subset of Z, where $\boldsymbol{\sigma} = \{\sigma_l\}_{l=1}^{L}$ is the output of A_{ans} and represents the set of code indices where the pre-execution output for the code matches the selected answer.

4. EXPERIMENTS

4.1. Experimental Settings

Datasets and evaluation metrics. Three VQA datasets, namely GQA [1], VQAv2 [2], and NLVR2 [3], are employed for assessment in the conducted experiment. The GQA dataset is known for its challenging questions that involve spatial reasoning, commonsense reasoning, and an understanding of image details. The VQAv2 dataset comprises images from the COCO dataset paired with open-ended questions. The questions exhibit a range of complexities and topics, and answers are expected to be provided in free-form text. The NLVR2 dataset consists of statements about pairs of images, each annotated as true or false based on the image content. For our experiments, these statements are reformulated into a query format.

Following the baseline method, 12 in-context examples are sampled to construct the in-context prompt for the GQA and VQAv2 datasets, while 6 examples are used for the NLVR2 dataset. To conserve computational resources, for certain datasets, we utilize the same randomly sampled subset as the baseline for evaluation purposes. In the results table, GQA val2000 consists of a subset of 2000 samples randomly selected from the GQA validation set, and VQAv2 val4000 comprises 4000 examples randomly sampled from the VQAv2 validation set. The evaluation metric is the exact match accuracy for case-insensitive answers.

4.2. Implementation Details

Frozen LLM π. PyramidCoder exhibits insensitivity to the underlying base LLM model π. In our evaluation, we employ one closed general LLM, namely, GPT-3.5 (`gpt-3.5-turbo`) available through the OpenAI API, and two open-source Code LLMs: StarCoderBase [23] and CodeLlama-7b-Python [24].

```
open_image(path: str) -> Image
- Opens the image at the path and returns it as an Image
  object.

get_object_boxes(img: Image, object: str) -> List(center_x
: float, center_y: float, width: float, height: float)
- Detects the object in the image and returns its
  bounding box.

crop(img: Image, box: [float, float, float, float], type:
str) -> List(Image)
- Crops the image according to the box and crop type.
- Type is one of ("object", "left", "right", "top",
  "bottom", "front").
- When type = "object", it will only return the object
  region.
- When type = "front", it will return the region in
  front of the object.
- When type is equal to other values, it will return the
  left/right/top/bottom image part of the object, note
  that the object itself is not included in this case.

query(img: Image, question: str) -> str
- Queries the image and returns an answer to the
question about the image content.
```

Fig. 4. API reference p_{api}.

Framework and APIs. We validate the efficacy of our approach by employing the state-of-the-art PVQA model, CodeVQA [8], as our baseline framework. Utilizing identical image processing modules as those employed in CodeVQA, we use GroundingDINO[30] for object detection through `get_object_boxes()`. For image captioning, we utilize BLIP [31] within `query()`. Additionally, the question-answering module incorporated within `query()` employs the same LLM as the one used in the code generation stage.

4.3. Results and Analysis

VQA performance. Table 1 lists the performance across three VQA datasets. Our baseline is the state-of-the-art CodeVQA model, which uses default IO prompting. The same settings and APIs are utilized for all experiments. Compared with the baseline, PyramidCoder demonstrates relative improvements of a minimum of 1.65 points on the GQA val2000 set, 0.5 points on the GQA test set, 1.4 points on the VQAv2 val4000 set, and 2.9 points on the NLVR2 test set. These results indicate the effectiveness of PyramidCoder, which maintains its performance across a variety of underlying LLMs and datasets.
Prompting Methods. Table 2 provides a comparative analysis of our PyramidCoder against existing prompting methodolo-

Method	GQA	VQAv2
Input-Output (Default)	48.0	61.3
Chain-of-Thought [9]	48.3	61.0
Tree of Thoughts [10]	48.7	61.5
PyramidCoder (Ours)	**51.3**	**62.8**

Table 2. Comparison with state-of-the-art prompting methods.

LLM	IO	Ours
gpt3.5-turbo	46.6	**55.8**
StarCoder-Base	46.5	**48.2**
CodeLlama-7b	48.0	**51.3**

Table 3. Results using different LLMs on GQA val2000. IO is the default strategy used in CodeVQA.

Method	GQA	VQAv2
PyramidCoder	51.3	62.8
w/o query rephraser	47.8	61.0
w/o code generator	49.4	61.0
w/o answer aggregator	51.0	62.1

Table 4. Ablation study with respect to the three modules.

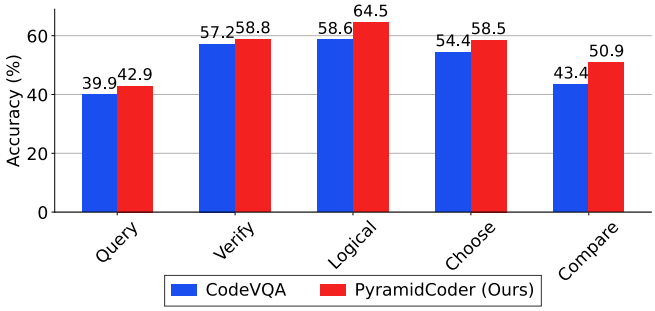


Fig. 5. QA Accuracy by question type.

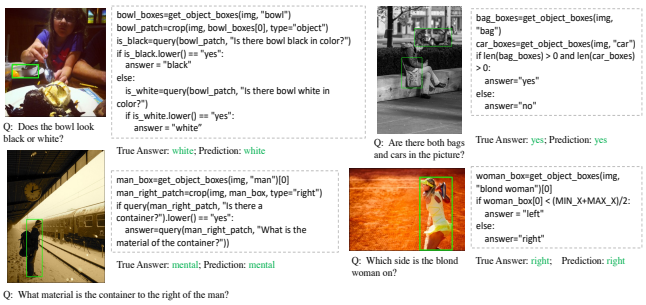


Fig. 6. Qualitative examples

gies, namely few-shot CoT and few-shot ToT with a creative writing setup. All methods employ identical in-context examples and few-shot configurations as described earlier. While the preceding prompting techniques show improvements in comparison to default IO prompting, they tend to generate homogeneous codes. In contrast, PyramidCoder surpasses these methods by producing diverse code candidates, which provides more substantial reasoning evidence.

Analysis by question type. Figure 5 illustrates the QA accuracy across diverse question types within the GQA val2000 dataset. The improvements are particularly evident in addressing logical questions involving logical inference, choose questions characterized by the inclusion of alternative options within the query, and compare questions requiring comparisons between two or more objects. These findings reveal the model's proficiency in handling queries necessitating intricate reasoning and comprehension of visual content. Moreover, PyramidCoder demonstrates heightened performance even in addressing simpler question types, such as verify questions that require binary responses (yes/no) and queries involving open-ended questions. This comprehensive enhancement underscores the versatility and efficacy of the PyramidCoder in diverse question-answering scenarios.

Qualitative examples. Figure 6 illustrates the diversified code generation capabilities of PyramidCoder. When confronted with queries of distinct content, the generated code exhibits varying levels of complexity and employs different APIs to construct disparate code solutions.

LLM. Table 3 shows the performance of PyramidCoder using different LLMs on the GQA val2000 dataset. Besides the results listed in Table 1, we also include results using GPT-3.5, a private general LLM, for code generation to measure its performance. The results summarized in Table 3 demonstrate that PyramidCoder consistently improves performance across diverse LLMs. This consistency highlights that PyramidCoder is independent of the employed LLM, as stronger LLMs usually lead to better performance. Remarkably, PyramidCoder exhibits versatility by performing well with both closed and open-source LLMs, as well as with models designed for general and code-specific tasks.

Ablation study. The ablation study is conducted on the sampled GQA and VQAv2 datasets, and the results are summarized in Table 4. The first row shows the performance of the complete PyramidCoder. To show the significance of each component, the query rephraser, code generator, and answer aggregator are individually omitted from the whole framework. The results clearly indicate that the exclusion of any of these modules leads to a noticeable decrease in the VQA accuracy across both datasets.

5. CONCLUSION

In this paper, we proposed PyramidCoder, a prompting framework designed for compositional visual question answering. This framework operates through query rephrasing, code candidate generation, and answer aggregation facilitated by three distinct modules. A key feature of this framework is its use of a single frozen LLM and predefined prompts at each level, eliminating the need for additional training and ensuring flexibility across different LLM architectures. Experimental evaluations conducted on three VQA datasets demonstrate its efficacy.

Acknowledgements This work was supported by JSPS KAKENHI Grant Number JP23H00490 and NEDO JPNP18002.

6. REFERENCES

[1] D. A. Hudson and C. D. Manning, "Gqa: A new dataset for real-world visual reasoning and compositional question answering," in *ICCV*, 2019.

[2] Y. Goyal, T. Khot, et al., "Making the v in vqa matter: Elevating the role of image understanding in visual question answering," in *CVPR*, 2017, pp. 6904–6913.

[3] A. Suhr, S. Zhou, A. Zhang, I. Zhang, H. Bai, and Y. Artzi, "A corpus for reasoning about natural language grounded in photographs," in *ACL*, 2019, pp. 6418–6428.

[4] L. H. Li, P. Zhang, H. Zhang, J. Yang, C. Li, Y. Zhong, L. Wang, L. Yuan, L. Zhang, J.-N. Hwang, et al., "Grounded language-image pre-training," in *CVPR*, 2022.

[5] A. M. H. Tiong, J. Li, B. Li, S. Savarese, et al., "Plug-and-play vqa: Zero-shot vqa by conjoining large pre-trained models with zero training," in *EMNLP Findings*, 2022.

[6] D. Surís, S. Menon, and C. Vondrick, "ViperGPT: Visual inference via python execution for reasoning," *arXiv:2303.08128*, 2023.

[7] T. Gupta and A. Kembhavi, "Visual programming: Compositional visual reasoning without training," in *CVPR*, 2022.

[8] S. Subramanian, M. Narasimhan, et al., "Modular visual question answering via code generation," in *ACL*, 2023.

[9] J. Wei, X. Wang, D. Schuurmans, et al., "Chain-of-thought prompting elicits reasoning in large language models," in *NeurIPS*, 2022.

[10] S. Yao, D. Yu, J. Zhao, et al., "Tree of Thoughts: Deliberate problem solving with large language models," in *NeurIPS*, 2023.

[11] M. Malinowski, M. Rohrbach, and M. Fritz, "Ask your neurons: A neural-based approach to answering questions about images," in *ICCV*, 2015, pp. 1–9.

[12] Z. Huang, H. Zhu, Y. Sun, et al., "A diagnostic study of visual question answering with analogical reasoning," in *ICIP*. IEEE, 2021, pp. 2463–2467.

[13] H. Zhang and W. Wu, "Context relation fusion model for visual question answering," in *ICIP*. IEEE, 2022, pp. 2112–2116.

[14] Z. Yang, X. He, J. Gao, et al., "Stacked attention networks for image question answering," in *CVPR*, 2016.

[15] P. Anderson, X. He, C. Buehler, et al., "Bottom-up and top-down attention for image captioning and visual question answering," in *CVPR*, 2018, pp. 6077–6086.

[16] A. Sarkar and M. Rahnemoonfar, "Grad-CAM aware supervised attention for visual question answering for post-disaster damage assessment," in *ICIP*. IEEE, 2022, pp. 3783–3787.

[17] X. Wu, J. Lu, Z. Li, and F. Xiong, "Ques-to-visual guided visual question answering," in *ICIP*. IEEE, 2022, pp. 4193–4197.

[18] L. H. Li, H. You, Z. Wang, et al., "Unsupervised vision-and-language pre-training without parallel images and captions," in *NAACL*, 2021.

[19] T. Le, H. T. Nguyen, and M. Le Nguyen, "Vision and text transformer for predicting answerability on visual question answering," in *ICIP*. IEEE, 2021, pp. 934–938.

[20] T. Brown, B. Mann, N. Ryder, et al., "Language models are few-shot learners," in *NeurIPS*, 2020, pp. 1877–1901.

[21] H. Touvron, L. Martin, K. Stone, et al., "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[22] M. Chen, J. Tworek, et al., "Evaluating large language models trained on code," *arXiv2107.03374*, 2021.

[23] R. Li et al., "Starcoder: may the source be with you!," *arXiv:2305.06161*, 2023.

[24] B. Roziere, J. Gehring, F. Gloeckle, et al., "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[25] G. Team, R. Anil, et al., "Gemini: a family of highly capable multimodal models," *arXiv preprint arXiv:2312.11805*, 2023.

[26] X. Wang, J. Wei, D. Schuurmans, et al., "Self-consistency improves chain of thought reasoning in language models," in *ICLR*, 2023.

[27] Z. Zhang, A. Zhang, M. Li, and A. Smola, "Automatic chain of thought prompting in large language models," in *ICLR*, 2023.

[28] S. Yao, J. Zhao, D. Yu, et al., "React: Synergizing reasoning and acting in language models," 2023.

[29] J. Johnson, B. Hariharan, L. van der Maaten, et al., "Inferring and executing programs for visual reasoning," in *ICCV*, 2017.

[30] S. Liu, Z. Zeng, T. Ren, et al., "Grounding DINO: Marrying DINO with grounded pre-training for open-set object detection," *arXiv preprint arXiv:2303.05499*, 2023.

[31] J. Li, D. Li, C. Xiong, and S. Hoi, "BLIP: Bootstrapping language-image pre-training for unified vision-language understanding and generation," in *ICML*. PMLR, 2022, pp. 12888–12900.