Enhancing Video-LLM Reasoning via Agent-of-Thoughts Distillation

Appendix

A. Limitation

Despite the advancements mentioned in the paper, several limitations remain and we leave them as future work: (i) similar to prior approaches [7, 10, 16], the effectiveness of our agent-based system is contingent upon the progress of the underlying visual model components. Enhancing its ability to generalize across diverse datasets is essential for broader applicability; (ii) while our primary focus has been on compositional VideoQA tasks [42], and we have demonstrated improvements across a series of benchmarks, achieving holistic enhancements will require further exploration into creating a more balanced distribution of training data; (iii) furthermore, our agent-based framework has the potential to address additional video-related tasks, such as video captioning and referring segmentation. We aim to expand our methodology to these domains, which could yield even more robust and versatile applications in the future.

B. Experimental Details

B.1. Training Details

For all models, their projection layers and language model are fine-tuned and visual encoder is frozen. We use a cosine learning rate schedule, with warm up ratio 0.03 and learning rate 4e-5. For both Instruct and AoTD setting, we fine-tune the model with batch size 48 and totally 1 epoch. We believe that longer training will get a better performance on in-domain benchmarks but maybe a destroy on out-of-domain benchmarks.

B.2. Specialized Models Evaluation Details

In this section we will show the details about each sub-task's evaluation from data preparation to evaluation metric.

Question decomposition. Since there may be multiple valid ways to decompose the same problem, we evaluate only the accuracy of the final output in this sub-task. Specifically, the model takes the query and instruction as input and generates an executable program. We replace all intermediate outputs within the program and focus on whether the final output matches the correct answer. If the decomposition is correct, the final output must align with the answer. Any programs that cannot be executed or that lead to an incorrect answer are considered failures.

Object detection. To evaluate the performance of detection models, we sample frames with scene graph annotations from the input video clip and provide them, along with the text query, as input to the model. The model then outputs a series of bounding boxes that exceed a confidence threshold. We select the bounding box with the highest confidence as the final output and calculate the IoU to assess accuracy.

Temporal grounding. Since scene graphs provide both the start and end frame IDs, as well as key frame IDs for each event, we use IoU and Recall as metrics to capture different aspects of model performance. The model takes the video clip and text query as input and outputs the predicted start and end frame IDs. We calculate IoU based on the alignment between the predicted and annotated start and end frame IDs, and we compute Recall using the key frame ID annotations to evaluate how well the model captures important frames.

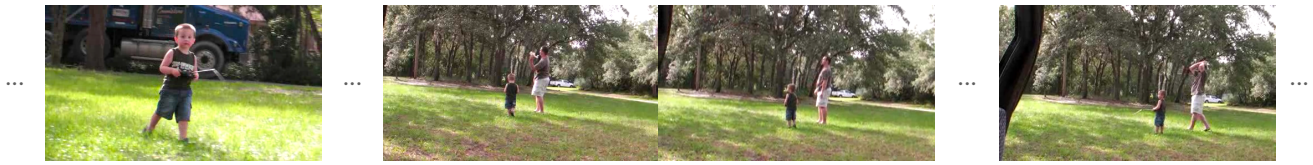
Action recognition. For discriminative models, we provide the video clip and a list of action labels as input to complete a classification task. For generative models, we provide the video clip along with an instruction prompt, asking the model to generate five actions most relevant to the video, ranked by likelihood. We then use the top-ranked output from each model to calculate the Top-1 accuracy for both approaches.

Question answering. The evaluation of question answering follows a similar approach to previous methods. The model takes the video clip and question as input and returns an answer, from which we directly calculate the accuracy. The key difference between this sub-task and a standard QA task is that the answers are based on a series of information collected by preceding agents, allowing for a more accurate assessment of the model's pure question-answering ability.

C. More Results

Here, we introduce some examples to show the process from query to Chain-of-Thought using our agent-based system. We can find that our system is able to decompose complex questions into easier sub-tasks and the final CoT retains step-by-step

problem-solving ideas and spatial-temporal information representing video understanding ability.



Frame 8 Frame 19 Frame 20 Frame 24

Question

Question: Why does the man have to throw the plane first in the middle of the video?
Possible answers: (A) chase the ball (B) the toy car hit the wall (C) put on earmuffs (D) get it start flying (E) bounce baby on cushion
Answer: (D) get it start flying

Program

```
def execute_command(video_clip, query, possible_answers:
    middle_clip = trim(video_clip, start=video_clip.num_frames//4, end=video_clip.num_frames//4*3)
    throw_clip = Filter_frames_with_act(middle_clip, 'man throw the plane')
    throw_captions = Video_caption(throw_clip)
    man_bboxes = Find(throw_clip, 'man')
    throw_reason = Video_query(throw_clip, query='why does the man have to throw the plane first',
                               possible_answers=possible_answers)
    info = {'reason why the man has to throw the plane first': throw_reason, 'what happened in the target
                                                                              video': throw_captions}
    answer = select_answer(query, info, possible_answers)
    return answer
```

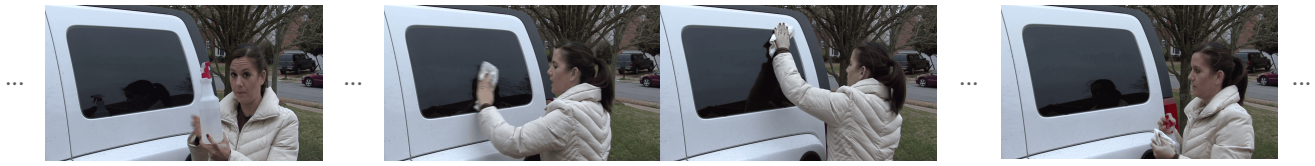
Execution Trace

call function trim
trimmed video from frame 8 to frame 24
call Filter_frames_with_act
filter action man throw the plane
find action from frame 16 to frame 20
call function Video_caption
frame 16: A man and a child playing plane in a park.
frame 17: A man and a child standing in a grassy field.
frame 18: A man and a child standing in a grassy field.
frame 19: A man and a child playing plane in a park.
call function Find
finding man
find man at [283, 45, 330, 247] in frame 16
find man at [215, 117, 249, 263] in frame 17
find man at [189, 122, 220, 266] in frame 18
find man at [201, 132, 236, 277] in frame 19
call function Video_query
Question: why does the man have to throw the plane first
Answer: The man has to throw the plane first because it is a remote control airplane and needs to be launched into the air before the young boy can start playing with it.
call function select_answer
select: D
program output: get it start flying

CoT

To solve this question, we first need to find when the man is throwing the plane in the middle of the video. The man is throwing the plane from frame 16 to frame 20.
Next, we should find where the man is to better know why he has to throw the plane first. We can do this by analyzing the man's bounding box region in each frame. The man's bounding box is [283, 45, 330, 247] in frame 16, [215, 117, 249, 263] in frame 17, [189, 122, 220, 266] in frame 18, and [201, 132, 236, 277] in frame 19.
By analyzing the man's bounding box region, we can see that the man is playing with a remote control airplane and needs to launch it into the air before the young boy can start playing with it. This is because the airplane needs to be in the air for the boy to play with it.
Therefore, the answer is that the man has to throw the plane first because it is a remote control airplane and needs to be launched into the air before the young boy can start playing with it. So the answer should be get it start flying.

Figure 5. Example form NExT-QA [43]



Question: What is the glass object that the person with the ring and the earring is sweeping after the woman holds up a bottle of spray?
Answer: car window

Question

```
def execute_command(video_clip):
     spray_clip = Filter_frames_with_act(video_clip, 'woman holds up a bottle of spray')
      after_spray_clip = trim(video_clip, start=spray_clip.end)
      sweep_clip = Filter_frames_with_act(after_spray_clip, 'person with the ring and the earring is
                                                              sweeping something')
      answer = Video_query(sweep_clip, 'what is the glass object')
      obj_bboxes = Find(sweep_clip, answer)
      return answer
```

Program

call Filter_frames_with_act
filter action woman holds up a bottle of spray
find action from frame 2 to frame 9
call function trim
trimmed video from frame 9 to frame 32
call Filter_frames_with_act
filter action person with the ring and the earring is sweeping something
find action from frame 10 to frame 16
call function Video_query
Question: what is the glass object
Answer: windshield
call function Find
finding windshield
find windshield at [26, 91, 179, 292] in frame 10
find windshield at [26, 88, 181, 293] in frame 11
find windshield at [26, 86, 182, 292] in frame 12
find windshield at [27, 90, 172, 292] in frame 13
find windshield at [27, 88, 179, 292] in frame 14
find windshield at [27, 89, 183, 293] in frame 15
program output: windshield

Execution Trace

To solve this question, we first need to find when the woman holds up a bottle of spray, from the video we can find that
the woman holds up a bottle of spray from frame 2 to frame 9.
Then we should focus on the time period after is, which is frame 9 to frame 32.
The person with the ring and the earring sweeps something from frame 10 to frame 16.
By analyzing the video scene, we can find the woman is sweeping the windsheld with towel.
The wnidsheld can be found at [26, 91, 179, 292] in frame 10, [26, 88, 181, 293] in frame 11 and so on.
So the answer should be windsheld.

CoT

Figure 6. Example form ANetQA [51]

D. Prompts

In this section we present the prompts used in our agent-based system for generating programs, converting execution traces and filtering CoTs.

D.1. Prompt for Program Generation

For each video and query, we call a code LLM to decompose the query to a Python program under the guidance of the prompt below. We modify the ViperGPT [37] prompt to adapt to the visual agents we use.

```
def Query_Objs(clip, query):
    """
    Query the objects that appear in video clip and match the query descriptions.
    Parameters
    -------
    clip:
        a list of video frames.
    query:
        Description of the target object.
    Returns
```

```
    -------
    a list of bounding boxes of the objects that match the query.
    Examples
    -------
    #return white_objs
    def execute_command(video_clip):
        white_objs = Query_Objs(video_clip, "white object")
        return white_objs
    """

def Query_Actions(clip, obj=None):
    """
    Find the actions happened in the video clip, if obj is not None, query the actions related to it.
    Parameters
    -------
    clip:
        a list of the video frames.
    obj:
        object class which is used to query the actions related to it.
    Returns
    -------
    a list of actions classes happened in the video clip.
    Examples
    -------
    #return actions
    def execute_command(video_clip, query, possible_answers):
        actions = Query_Actions(video_clip)
        return actions
    """

def Filter_frames_with_act(clip, action):
    """
    filter a new video clip containing the time period in which the target action occurred
    Parameters
    -------
    clip:
        a list of video frames.
    action:
        the target action which is used to filter frames.
    Returns
    -------
    a new video clip ontaining the time period in which the target action occurred.
    Examples
    -------
    #return jump_clip
    def execute_command(video_clip, query, possible_answers):
        jump_clip = Filter_frames_with_act(video_clip, "person is jumping")
        return jump_clip
    """

def Filter_frames_with_obj(clip, obj):
    """
    filter a new video clip that the target object occured.
    Parameters
    -------
    clip:
        a list of video frames.
    obj:
        class or description about the target object.
    Returns
    -------
    a new video clip that the target object occured in it.
    Examples
    -------
    #return shoe_clip
    def execute_command(video_clip, query, possible_answers):
        shoe_clip = Filter_frames_with_obj(video_clip, "shoe")
        return shoe_clip
    """

def trim(clip, start=None, end=None):
    """
    Returns a new video clip containing a trimmed version of the original video at the [start, end] clip.
    Parameters
    ----------
    clip:
        a list of video frames.
    start : Union[int, None]
        An int describing the starting frame in this video clip with respect to the original video.
```

```
    end : Union[int, None]
        An int describing the ending frame in this video clip with respect to the original video.

    Returns
    -------
    a new video clip with start and end.
    """
def Find(clip, obj):
    """
    find all bounding boxes around a certain object in the video clip,
    and collates them into a collection of frames.
    Parameters
    ----------
    clip:
        a list of video frames.
    obj:
        the object to look for.
    Returns
    -------
    a new video clip composed of crops of the object.
    Examples
    --------
    # Return the shoe_clip
    def execute_command(video_clip, query, possible_answers):
        shoe_clip = Find(video_clip, "shoe")
        return shoe_clip
    """

def select_answer(query, info, possible_answers):
    """
    Uses a language model to choose the option that best answers the question given the input information.
    Parameters
    ----------
    query:
        the input question.
    info:
        Any useful information to answer the question.
    possible_answers:
        a list of possible answers to the question.
    Returns
    -------
    one answer chosen from the possible answers.
    Examples
    --------
    # Return the answer
    def execute_command(video_clip, query, possible_answers):
        clip_summary = Video_summary(video_clip)
        info = {
            "summary of the target video": clip_summary
        }
        answer = select_answer(query, info, possible_answers)
        return answer
    """
def exist(clip, query):
    """
    judge whether a object exists in the video.
    Parameters
    ----------
    clip:
        a list of video frames.
    query:
        query to the object class.
    Returns
    -------
    Return True if the object specified by query is found in the video, and False otherwise.
    Examples
    --------
    # Return the flag
    def execute_command(video_clip, query, possible_answers):
        flag = exist(video_clip, "shoe")
        return flag
    """
def Video_summary(clip, query):
    """
    give a brief summary of the video clip related to the query.
    Parameters
    ----------
    clip:
        a list of video frames.
```

```
    query:
        a question about the video.
    Returns
    -------
    return a brief summary of the video clip.
    Examples
    --------
    # Return the clip_summary
    def execute_command(video_clip, query, possible_answers):
        clip_summary = Video_summary(video_clip, query)
        return clip_summary
    """
Write a function using Python and the functions (above) that could be executed to provide an answer to the query.

Consider the following guidelines:
- Use base Python (comparison, sorting) for basic logical operations, start/end, math, etc.
- Objects with mutiple names like "phone/camera", "cup/glass/bottle" with slash, input them as a whole object name.
- Just use the class and function appear above except for some base python operations.
- Only answer with a function starting def execute_command, do not answer any extra words and symbols before and after the function.
- No text that is not related to function can appear.
- the answer only begins with "def execute_command" and ends with "return answer".

Here are some examples of the function you should write:
-------
question: What else is the person able to do with the door?
possible answers: ["Hold the door.", "Put down the door.", "Close the door.", "Open the door."]
def execute_command(video_clip, query, possible_answers):
    door_clip = Filter_frames_with_obj(video_clip, "door")
    person_clip = Find(door_clip, "person")
    clip_summary = Video_summary(person_clip, query)
    door_actions = Query_Actions(person_clip, "door", possible_answers=possible_answers)
    door_actions =
    info = {
        "actions the person able to do with the door else": door_actions,
        "summary of the target video": clip_summary
    }
    answer = select_answer(query, info, possible_answers)
    return answer
-------
Query: INSERT_QUERY_HERE
possible answers: INSERT_POSSIBLE_ANSWERS_HERE
```

D.2. Prompt for Execution Trace Conversion

After getting the execution trace by running the program step by step, we use a LLM to convert the trace into a natural language CoT. The LLM takes query, execution trace, possible answers (in MC-VQA) and execution trace as input. The instruction prompt is as follow:

```
Given a video and a question, I wrote the function execute_command using Python, and the other functions above that could be executed to provide an answer to the query.
As shown in the code, the code will print execution traces.
I need you to rewrite the execution trace into a natural language rationale that leads to the answer.

Consider the following guidelines:
- Use all the bounding box information in the rationale, do not use words like "so on" to omit the bounding box, just write all of them into the rationale.
- Referencing the execution trace, write a reasoning chain that leads to the most common human answer. Notice that the output should be the same as the human answer, not necessarily the program output.
- If some part of the rationale lacks logic, add reasonable content to make it logical.


Here are some examples of the rantionale you should write:
-----
Question: What did the person do with the table?
def execute_command(video_clip, query, possible_answers, time_wait_between_lines, syntax):
    table_clip = Filter_frames_with_act(video_clip, 'person interacting with table')
    person_clip = Find(table_clip, 'person')
    table_bboxes = Find(table_clip, 'table')
    clip_summary = Video_summary(person_clip)
    person_action = Query_Actions(person_clip, 'table', possible_answers=possible_answers)
    info = {'actions the person do with the table': person_action, 'summary of the target video': clip_summary}
    answer = select_answer(query, info, possible_answers)
    return answer
Execution trace:
call Filter_frames_with_act
filter action person interacting with table
find action from frame 2 to frame 11
```

```
call function Find
finding person
find person at [139, 141, 229, 342] in frame 2
find person at [151, 123, 242, 349] in frame 3
find person at [153, 121, 242, 274] in frame 4
find person at [158, 123, 255, 261] in frame 5
find person at [163, 124, 270, 262] in frame 6
find person at [153, 121, 242, 351] in frame 7
find person at [95, 113, 196, 316] in frame 8
find person at [83, 113, 196, 285] in frame 9
find person at [112, 116, 201, 332] in frame 10
call function Find
finding table
find table at [183, 140, 269, 257] in frame 2
find table at [194, 131, 269, 255] in frame 3
find table at [227, 129, 269, 252] in frame 4
find table at [226, 165, 269, 258] in frame 5
find table at [233, 170, 270, 259] in frame 6
find table at [217, 129, 269, 256] in frame 7
find table at [217, 122, 270, 254] in frame 8
find table at [221, 123, 269, 256] in frame 9
find table at [225, 125, 270, 263] in frame 10
call function Video_summary
summary result: The video shows a man in a kitchen, bending over and holding an orange object, surrounded by various kitchen items and furniture, with a focus on his actions and the domestic setting.
call function Query_Actions
Query table
Answer: tidied up.
call function select_answer
the information used: - actions the person do with the table: tidied up.
- summary of the target video: The video shows a man in a kitchen, bending over and holding an orange object, surrounded by various kitchen items and furniture, with a focus on his actions and the domestic setting.
program output: Tidied up.
Rationale:
To solve this question, we first have to find when did the person interact with the table.
From the video, we can see that the person is interacting with the table from frame 2 to frame 11.
In this time period, we can find person at [139, 141, 229, 342] in frame 2, [151, 123, 242, 349] in frame 3, [153, 121, 242, 274] in frame 4 and so on.
Table can also be found at [183, 140, 269, 257] in frame 2, [194, 131, 269, 255] in frame 3, [227, 129, 269, 252] in frame 4 and so on.
By analyzing the person and table bounding box region, we can see that the person is holding an orange object to clean the table in the kirchen environment.
So the answer should be tidied up.
-----------------------------------------------
Now, look the question, program and execution trace, please transfer these information to a rantionale.
Question: INSERT_QUESTION_HERE
INSERT_PROGRAM_HERE
Execution trace:
INSERT_EXECUTION_TRACE_HERE
Rationale:
```

D.3. Prompt for CoT Filtering

In order to obtain high quality distillation data, we continue using LLM to filter CoTs. We prompt the LLM to select those CoTs that are truly helpful for solving questions and reflect the step-by-step thinking process. The prompt is as follows:

```
I will give you a question and a rationale to solve the question, you need to judge whether the rationale is thinking step by step and helpful to solve the question.
If yes, return True, If not, return False. no need to explain.
Here is the question and rationale:
Question: INSERT_QUESTION_HERE
Rationale: INSERT_RATIONALE_HERE
```

D.4. Prompt for Inference

Question: question content
Answer in one word or phrase. / Explain the rationale to answer the question.

Question: question content
Options:
(A) option content
(B) option content
(C) option content
(D) option content
Answer with the option's letter from the given choices directly and only give the best option. / Explain the rationale to answer the question.

E. More related works

E.1. Video-language models (Video-LLMs).

Most existing Video-LLMs [4, 44, 55] are composed of a pre-trained visual encoder (like CLIP [34] or SigLIP [53]) to encode video frames into a sequence of visual features, an adapter to transfer the visual features to tokens that can be understood by the language model, and a pretrained LLM to output the final response.

E.2. Visual Programming and Agents.

Recent work like MoReVQA [27] proposes a multi-stage system, getting a strong zero-shot VideoQA ability while is able to create interpretable intermediate outputs. VURF [25] proposes a self-refinement method to resolve the LLM hallucinations to get a more concise program based on the context cues.

F. More Ablations

F.1. Analysis on latency and computation.

We compare the agent-based system with the distilled model (LNV-AoTD) across three key metrics. As shown in Table 8, LNV-AoTD outperforms the agent-based system in both inference latency and computational efficiency, demonstrating the necessity of distillation process.

Model	Time (s) ↓	Memory (GB) ↓	TFLOPs ↓
Agent-based system	47.93	65.34	30.98
LNV-AoTD	10.58	18.21	13.53

Table 8. Latency and computation comparison.

F.2. Analysis on more models

We further experiment on VideoLLaMA2[4] and Qwen2-VL[38] using identical settings. The results confirm AoTD's effectiveness and superior performance.

Model	Filtering	MVBench (Acc.)	STAR (Acc.)	AGQA (Acc. / Score)
LNV-AoTD	✗	53.7	73.3	59.5/3.5
LNV-AoTD	✓	55.6	74.3	60.9/3.6
VideoLLaMA2-Instruct	-	54.9	69.2	56.0/3.5
VideoLLaMA2-AoTD	✓	56.0	71.1	57.2/3.5
Qwen2-VL-Instruct	-	65.6	71.4	59.8/3.6
Qwen2-VL-AoTD	✓	66.5	73.1	61.2/3.7

Table 9. Ablation results on more models.