Supplementary material for ViPR

Anonymous Author(s)
Affiliation
Address
email

1 Running the code

To reproduce the results from the paper, follow the steps below.

1. Make sure you use Python 3.9.18.
2. Download ViCLIP-L$_{InternVid-FLT-10M.pth from}$ and put it into .cache/encoders/viclip

Put your OpeanAI API key into `.env`, under the key `OPENAI_API_KEY=`

Create a virtual environment, install the dependencies with `pip install --editable .`

Download the dataset files to a local directory TODO: from where? and note the directory of the `.../videos` and `.../tasks` directories

Set the video dir and task dir in the config files to the respective directories above. For example, in `configs/alfred_replication.yaml` you should set `task_dir: "[.../tasks/alfred]"`

Run the evaluation on the selected environment by running `vlm evaluate configs/[alfred_replication,minecraft_replication,real_life_replication].yaml`

After the run is complete (which can take upwards of 12 hours), you can create plots from the logs by running `vlm plot --experiment-dir experiments --task-dir [the directory]`

2 Dataset details

The evaluation code we use works with the dataset being separated into mp4 video files, yaml task files, and json task data files, which is the structure we detail below. We also supply the `dataset.json` file which lists all metadata for all videos in one file, to make it easier to use the dataset with your own evaluation code.

2.1 General data structure

Each of the three environments in the dataset has two parts:

- Videos: mp4 video clips organised into sub-directories. By convention, the directory structure corresponds to the three environments we have, and within them, to the different difficulty levels and task types (foundation/base, permutation, remix). The specific paths serve as identifiers when matching videos to classes in classification problems.
- Classification problems: yaml and json files organised into sub-directories. For a given problem, the yaml files contains its definition (detailed below), and the json file references to the videos and their classes. The path of the yaml file serves as identifier for the problem within the evaluation code.

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

Problem definition format The yaml file has a top-level dictionary with the following keys:

- *label_prompts:* a dictionary, where keys are class ids (labels) and their associated values are natural language descriptions which are given to the models
- *prompt_gpt:* a 0-shot prompt describing the task, given to GPT-4o during evaluation
- *example_gpt:* a prompt containing environment-specific task hints (e.g. what holding looks like, common mistakes, etc.), given to GPT-4o during 1-shot evaluation
- *metadata:* optional, a dictionary that can contain any metadata associated with the task. The metadata are not currently used by the code in any way.

Problem data format For problem *[problem].yaml*, the json file must be named *[problem]_data.json*. The json file is a list of dictionaries with the following two keys:

- *path:* the path to a video, relative to the *video_path* given in the experiment configuration.
- *label:* the class id (label) of the video. This has to match one of the class ids in the label_prompts dictionary in the yaml file.

2.2 Virtual home video acquisition protocols

This section details the design decisions and preparation protocol behind the virtual home videos. The code to generate the virtual home videos is in the `generate-virtual-home` directory.

In general, the videos were created by cutting and pasting frames from different ALFRED trajectories. We were able to automatize this process because ALFRED contains step-by-step descriptions of actions in each trajectory, together with a list of frames each action corresponds to.

Base problems Base problem videos were created by preprocessing videos from the ALFRED dataset, to isolate individual actions. When possible, we include a few surrounding frames for context — but, we only do this if the frames do not contain a different action (i.e., they are only included if they only contain movement). When we say we created n tasks, each with an action being performed on a different object, we selected the objects to be included at random.

Specifically, we have the following problem types for object recognition:

- *Object recognition:* We isolate all PickupObject actions and their associated frames and descriptions from the original ALFRED videos, for each object keeping 10 (if possible) videos where it's being picked up from a single type of container. Finally, we group the videos based on where the object was picked up from and create classification problems from those groups.

  The goal is to have problems with videos that all show the agent picking up an object from one type of container, so that the type of object is the only major difference between the videos within the problem.

  We only keep problems that have more than one class, ending up with "pickup from countertop", "pickup from dining table", and "pickup from somewhere" (i.e. instances where the container was not specified in the ALFRED dataset).
- *Container recognition:* We perform a similar selection process to the one above. We isolate all PutObject actions and their associated frames and descriptions from the original ALFRED videos, for each container keeping 10 (if possible) videos where a certain object is being picked up from it. Finally, we group the videos them based on which object was picked up and create classification problems from those groups.

  We end up with four problems: "put butter knife", "put keychain", "put mug", and "put soap", each problem containing videos of the respective object being picked up from different containers.

We also have the following ones for action understanding:

- *Cleaning:* We isolate all "clean" actions and their associated frames and descriptions from the original ALFRED videos, and then drop some frames to create the following alternative

videos: "we clean the [object] in the sink basin under running water" (original video), "we go to the sink basin, holding [object] in hand, we don't put it in the sink basin", "we put the [object] in the sink basin and pick it back up without running water over it".

We create 5 such problems, corresponding to the agent cleaning 5 different objects, with 10 videos (if possible) per class in each problem.

- *Cooling:* Similarly to Cleaning, we isolate all "cool" actions from ALFRED, and cut some frames to end up with the following alternatives: "we cool the [object] by putting it in the fridge for a while" (original video), "we open the fridge, put in the [object] and immediately pick it back up without leaving it to cool in a closed fridge", "we open the fridge and then immediately close it without putting the [object] in", and "we go to the fridge and then we leave, without even opening it".

  Again, we create 5 such problems, corresponding to the agent cooling 5 different objects, with 10 videos (if possible) per class in each problem.

- *Heating:* Similarly to Cleaning, we isolate all "heat" actions from ALFRED, and cut some frames to end up with the following alternatives: "we heat the [object] in the microwave", "we go to the microwave and then immediately leave without putting the [object] in", "we open the microwave and immediately close it again, without putting the [object] in", "we open the microwave, put the [object] in, then immediately pick it back up without heating it", "we put the [object] in the microwave for a while, we do not turn the microwave on, then we pick up the [object] back again".

  Again, we create 5 such problems, corresponding to the agent heating 5 different objects, with 10 videos (if possible) per class in each problem.

- *Picking:* For 5 objects, we isolate 10 (is possible) PickupObject and PutObject videos that have the agent interact with (pick from or put to) the same container. The problem then is for the VLM to decide whether the object has been picked up or put down, since that is the single largest difference between the videos in a given problem.

- *Slicing:* Similarly to the above, we isolate all "slice" actions from ALFRED together with their frames and descriptions, and then create alternative versions of these videos by cutting the last part where the slicing actually happens out. Thus, in the alternative videos, the agent holds a knife and walks towards an object, preparing to slice it, but does not perform the slicing.

  Again, we create 5 such problems, corresponding to the agent (not) slicing 5 different objects, with 10 videos (if possible) per class in each problem.

- *Toggling:* There is no ToggleOff action in ALFRED, so to simulate it, we reverse the frames from a ToggleObjectOn. Specifically, we isolate 10 toggle on and toggle off videos (if possible) for a desk lamp, floor lamp, faucet, and microwave (creating a separate problem for each object), since those are the only objects that can have a turned-on state in ALFRED.

And finally, we have two problem types for object state recognition:

- *On v. off:* We isolate ToggleObjectOn actions from ALFRED, and cut their corresponding videoclips to to halves — the first has the agent approach the object, which is turned off for the entire duration of the clip, and the second already shows the object when it's been turned on. The problems than ask the VLM to say whether the object we see is turned on or off.

  As above, we isolate 10 turned on and turned off videos (if possible) for a desk lamp, floor lamp, faucet, and microwave (creating a separate problem for each object), since those are the only objects that can have a turned-on state in ALFRED.

- *Sliced v. whole:* We isolate PickupObject actions from ALFRED, and group together videos that have the agent pick up either a slice of something, or the whole thing. We create 5 such problems, corresponding to the agent picking up 5 different objects, with 10 videos (if possible) per class (sliced and whole) in each problem.

Remix problems Remix problems are generated iteratively step-by-step from a base trajectory:

The find_rest procedure performs DFS; two actions (and their corresponding frames) can come after each other if they are set in the same scene (room type), if the content of agent's hands is the same, and if there exists a sequence of frames somewhere in ALFRED that shows the agent going from the

```
for prefix_len in base_length, ..., 0 do
  for rest in find_rest(base[:prefix_len]) do
    if we have enough videos for this prefix length then
      break
    end if
    if rest[0] has already been taken after this exact prefix then
      continue
    end if
    trajectory ← base[:prefix_len] + rest
    add_to_trajectories(trajectory)
  end for
end for
```

place where the first action happened to the place of the second action. We deduce the place from action metadata in ALFRED.

What "having enough videos" means is different for different difficulty levels and prefix lenghts; below we list the number of videos we aim to generate:

- Level 2: 4 for prefix 0, 4 for prefix 1
- Level 3: 2 for prefix 0, 3 for all larger prefixes
- Level 4: 2 for prefix 0, 2 for all larger prefixes
- Level 5: 4 for prefix 0, 1 for all larger prefixes
- Level 6: 3 for prefix 0, 1 for all larger prefixes
- Level 7: 2 for prefix 0, 1 for all larger prefixes
- Level 8: 1 for prefix 0, 1 for all larger prefixes

We generate 12 such problems per level. The base trajectory is selected randomly without replacement each time. If it is impossible to generate a 9-class problem for a given base trajectory, we try it with the next base trajectory.

Permutation problems Permutation problems are generated iteratively from a base trajectory: we loop through all permutations, keeping those in which all pairs of neighbouring actions can be connected by frames (coming from other ALFRED videos) that show the agent going from the place where to first action happened to the place of the second action. In the end, we only keep the first 3 permutations (i.e. classes) of those we found.

If possible, we also prefer permutations that are "slicing-consistent", i.e. those in which all actions (like moving, heating, ...) with a whole object O come in sequence before it is sliced, and all actions with a slice of O come after it was sliced. A permutation that is not slicing-consistent is still admissible and strictly speaking logically consistent — consider that there could be multiple copies of a given object, for example, and one can thus be sliced while the other stays whole. On some levels, we do include slicing-inconsistent trajectories because we cannot generate enough consistent ones.

We generate 33 problems per level. The base trajectory is selected randomly without replacement each time. If it is impossible to generate a 3-class problem for a given base trajectory, we try it with the next base trajectory (except for ones on level 2).

2.3 Used dataset licenses

See table 1 for an overview of datasets we used to construct our dataset and their licenses.

3 Model details

Models get equally-spaced frames from a video; the time elapsed between frames thus depends on video length, which is non-uniform. We make sure that the last frame the model receives is also the

Dataset	License
ALFRED	MIT

Table 1: Overview of used datasets and their licenses.

very last frame of the video, by duplicating the last frame of the video before subsampling it. The complete subsampling logic is in figure 1.

```
def subsample(x: t.Tensor, n_frames: int) -> t.Tensor:
    total_frames, *_ = x.shape

    if (total_frames - n_frames) % (n_frames - 1) != 0:
        # Replicate the last frame to make sure it will be selected
        last_frame = einops.repeat(
            x[-1], "... -> n ...", n=(n_frames - (total_frames % (n_frames - 1)))
        )
        x = t.cat([x, last_frame])
        total_frames, *_ = x.shape
        assert (total_frames - n_frames) % (n_frames - 1) == 0
    step = (total_frames - n_frames) // (n_frames - 1) + 1
    x_subsampled = x[::step]
    assert len(x_subsampled) == n_frames
    assert (x[0] == x_subsampled[0]).all() and (x[-1] == x_subsampled[-1]).all()
    return x_subsampled
```

Figure 1: The frame subsampling logic, where `x` is the video in shape `(time, w, h, c)` and `n_frames` is the model-dependent number of frames to subsample to.

3.1 Model specification

We used three models in our final evaluation:

1. OpenGVLAB/ViCLIP (available on HuggingFace)
2. laion/clip-vit-bigg-14-laion2b-39b-b160k (available on HuggingFace)
3. GPT-4o (version from 1. 6. 2024)

3.2 Compute

We ran experiments on the following hardware. Our specific choices were based on availability only; by varying batch sizes, all of the experiments could be run on any of these.

- Compute cluster, with NVIDIA A100 GPU 80GB
- Rented cloud machines (vast.ai), with RTX 3090
- locally, on the M1 Pro processor

Varying the batch sizes does not influence the results.

3.3 GPT-4o prompts

We evaluate GPT-4o in a few-shot manner, using the following interaction scheme:

1. First conversation, to get frame-by-frame descriptions:
   (a) First and only message: system prompt (figure 2) + frame-by-frame prompt (figure 3)
2. In a second conversation, separate from the first one:
   (a) First message: The same system prompt (figure 2) + class match prompt (figure 4)

(b) Second message, same conversation, after GPT-4o replies: scoring prompt (figure 5)

We found this prompting setup to yield significantly better results than forcing the model to do everything (describing the frames, then matching the descriptions to classes) in a single forward pass. We extract the scores from the last response which has predictable easy-to-parse format. For the full prompts see figs. 2 to 5.

```
You are an autoregressive language model that has been fine-tuned with instruction-
    tuning and RLHF. You carefully provide accurate, factual, thoughtful, nuanced
    answers, and are brilliant at reasoning. Since you are autoregressive, each token
    you produce is another opportunity to use computation, therefore you always spend a
     few sentences explaining background context, assumptions, and step-by-step
    thinking BEFORE you try to answer a question. You always use precise, plain
    scientific language, without unneeded flourish. You provide details where it might
    help the explanation.
```

Figure 2: The system prompt, used in both requests.

```
You will be given {{n_frames}} frames from a first-person video. The frames are given to
     you in chronological order.

{{task.prompt_gpt}}

{{task.example_gpt}}

Input frames. Describe each frame separately.

{{frames}}
```

Figure 3: The prompt used to obtain frame-by-frame descriptions. *n_frames* depends on the model setting (by default, 16), *task.prompt_gpt* and *task.example_gpt* are problem-specific, the latter only being used in 1-shot mode (which is the default). *frames* refer to the actual frames, which are base64 encoded and passed through the OpenAI API as detailed in their documentation.

```
Consider the following sequence of frame-by-frame descriptions:

‘‘‘
{{frame_descriptions}}
‘‘‘

Break down each of the following summaries into individual steps, and mention what
    frames or frame ranges each step matches in parentheses after each step. Note even
    partial matches, e.g. matching a kind of action (put, pick, ...) even though the
    object might be incorrect. Also provide a one-sentence commentary on how well each
    summary matches the sequence of the frame descriptions. In the commentary, the most
     important thing is to match the kinds of actions performed and their order. For
    example if put (of anything) was described before a pick (of anything) in the
    frames, maintaining this order in the summary is more important than getting the
    exact object right. Do not comment on the overall quality of the summaries.

Summaries, given in the format ‘- (label) summary‘:
{{class_list}}
```

Figure 4: The prompt used to make GPT-4o think step by step about how well each class description matches the frame descriptions it produced in the first request. The *frame_descriptions* are the generated frame-by-frame descriptions, *class_list* is the list of label-description pairs from the task yaml definition. The raw frames are not supplied to the model here anymore.

Based on your findings in the previous section, score the summaries from 0 to 5, where 0 is "likely does not describe the video" and 5 is "among the given options, this one most likely describes the video". Make sure to score each summary individually. At least one score should be non-zero, even if it's not a perfect match. Whatever scores you pick, there !must be! exactly one summary with the highest score. Follow the format below, verbatim:

```
- (label) score
```

Figure 5: The prompt used to obtain the final scores in a predictable format.