Run LLaVA-Next on SGLang for Visual QA

View on GitHub

Vision-Language Models (VLMs) are like LLMs with eyes: they can generate text based not just on other text, but on images as well.

This example shows how to run a VLM on Modal using the SGLang library.

Here's a sample inference, with the image rendered directly in the terminal:

```
Question: What is this?
```

```
Answer:
Stopping app - local entrypoint completed.
The image shows the Statue of Liberty, an iconic symbol of freedom located on a small isla
nd in the middle of New York Harbor, with a backdrop of the Manhattan skyline. The statue
is a neoclassical sculpture with a pedestal beneath it, and it has been a prominent figure
 in the harbor since it was dedicated in 1886. It's a well-maintained and culturally signi
ficant attraction, often associated with the harbor and tourism in the United States.
request 672af06d-5bc6-410f-9ab9-8958856e41a4 completed in 3.8 seconds
```

Setup

First, we'll import the libraries we need locally and define some constants.

```
import os
import time
import warnings
from uuid import uuid4

import modal
import requests
```

VLMs are generally larger than LLMs with the same cognitive capability. LLMs are already hard to run effectively on CPUs, so we'll use a GPU here. We find that inference for a single input takes about 3-4 seconds on an A10G.

You can customize the GPU type and count using the `GPU_CONFIG` and `GPU_COUNT` environment variables. If you want to see the model really rip, try an `"a100-80gb"` or an `"h100"` on a large batch.

```
GPU_TYPE = os.environ.get("GPU_CONFIG", "a10g")
GPU_COUNT = os.environ.get("GPU_COUNT", 1)

GPU_CONFIG = f"{GPU_TYPE}:{GPU_COUNT}"

SGL_LOG_LEVEL = "error"  # try "debug" or "info" if you have issues

MINUTES = 60  # seconds
```

We use a LLaVA-NeXT model built on top of Meta's LLaMA 3 8B.

```
MODEL_PATH = "lmms-lab/llama3-llava-next-8b"
MODEL_REVISION = "e7e6a9fd5fd75d44b32987cba51c123338edbede"
TOKENIZER_PATH = "lmms-lab/llama3-llava-next-8b-tokenizer"
MODEL_CHAT_TEMPLATE = "llama-3-instruct"
```

We download it from the Hugging Face Hub using the Python function below.

```
def download_model_to_image():
    import transformers
    from huggingface_hub import snapshot_download

    snapshot_download(
        MODEL_PATH,
        revision=MODEL_REVISION,
        ignore_patterns=["*.pt", "*.bin"],
    )

    # otherwise, this happens on first inference
    transformers.utils.move_cache()
```

Modal runs Python functions on containers in the cloud. The environment those functions run in is defined by the container's `Image`. The block of code below defines our example's `Image`.

```
vllm_image = (
    modal.Image.from_registry(  # start from an official NVIDIA CUDA image
        "nvidia/cuda:12.2.0-devel-ubuntu22.04", add_python="3.11"
    )
    .apt_install("git")  # add system dependencies
    .pip_install(  # add sglang and some Python dependencies
        "sglang[all]==0.1.16",
```

```
        "ninja",
        "packaging",
        "wheel",
        "transformers==4.40.2",
    )
    .run_commands(  # add FlashAttention for faster inference using a shell command
        "pip install flash-attn==2.5.8 --no-build-isolation"
    )
    .run_function(  # download the model by running a Python function
        download_model_to_image
    )
    .pip_install(  # add an optional extra that renders images in the terminal
        "term-image==0.7.1"
    )
)
```

Defining a Visual QA service

Running an inference service on Modal is as easy as writing inference in Python.

The code below adds a modal `Cls` to an `App` that runs the VLM.

We define a method `generate` that takes a URL for an image URL and a question about the image as inputs and returns the VLM's answer.

By decorating it with `@modal.web_endpoint`, we expose it as an HTTP endpoint, so it can be accessed over the public internet from any client.

```
app = modal.App("app")


@app.cls(
    gpu=GPU_CONFIG,
    timeout=20 * MINUTES,
    container_idle_timeout=20 * MINUTES,
    allow_concurrent_inputs=100,
    image=vllm_image,
)
class Model:
    @modal.enter()  # what should a container do after it starts but before it gets input?
    async def start_runtime(self):
        """Starts an SGL runtime to execute inference."""
        import sglang as sgl

        self.runtime = sgl.Runtime(
            model_path=MODEL_PATH,
            tokenizer_path=TOKENIZER_PATH,
            tp_size=GPU_COUNT,  # t_ensor p_arallel size, number of GPUs to split the mode
```

```
            log_evel=SGL_LOG_LEVEL,
        )
        self.runtime.endpoint.chat_template = (
            sgl.lang.chat_template.get_chat_template(MODEL_CHAT_TEMPLATE)
        )
        sgl.set_default_backend(self.runtime)

    @modal.web_endpoint(method="POST")
    async def generate(self, request: dict):
        import sglang as sgl
        from term_image.image import from_file

        start = time.monotonic_ns()
        request_id = uuid4()
        print(f"Generating response to request {request_id}")

        image_url = request.get("image_url")
        if image_url is None:
            image_url = "https://modal-public-assets.s3.amazonaws.com/golden-gate-bridge.j

        image_filename = image_url.split("/")[-1]
        image_path = f"/tmp/{uuid4()}-{image_filename}"
        response = requests.get(image_url)

        response.raise_for_status()

        with open(image_path, "wb") as file:
            file.write(response.content)

        @sgl.function
        def image_qa(s, image_path, question):
            s += sgl.user(sgl.image(image_path) + question)
            s += sgl.assistant(sgl.gen("answer"))

        question = request.get("question")
        if question is None:
            question = "What is this?"

        state = image_qa.run(
            image_path=image_path, question=question, max_new_tokens=128
        )
        # show the question, image, and response in the terminal for demonstration purpose
        print(
            Colors.BOLD, Colors.GRAY, "Question: ", question, Colors.END, sep=""
        )
        terminal_image = from_file(image_path)
        terminal_image.draw()
        answer = state["answer"]
        print(
            Colors.BOLD,
            Colors.GREEN,
            f"Answer: {answer}",
```

```
            Colors.END,
            sep="",
        )
        print(
            f"request {request_id} completed in {round((time.monotonic_ns() - start) / 1e9
        )

    @modal.exit()  # what should a container do before it shuts down?
    def shutdown_runtime(self):
        self.runtime.shutdown()
```

Asking questions about images via POST

Now, we can send this Modal Function a POST request with an image and a question and get back an answer.

The code below will start up the inference service so that it can be run from the terminal as a one-off, like a local script would be, using `modal run`:

```
modal run sgl_vlm.py
```

By default, we hit the endpoint twice to demonstrate how much faster the inference is once the server is running.

```
@app.local_entrypoint()
def main(image_url=None, question=None, twice=True):
    model = Model()

    response = requests.post(
        model.generate.web_url,
        json={
            "image_url": image_url,
            "question": question,
        },
    )
    assert response.ok, response.status_code

    if twice:
        # second response is faster, because the Function is already running
        response = requests.post(
            model.generate.web_url,
            json={"image_url": image_url, "question": question},
        )
        assert response.ok, response.status_code
```

Deployment

To set this up as a long-running, but serverless, service, we can deploy it to Modal:

```
modal deploy sgl_vlm.py
```

And then send requests from anywhere. See the docs for details on the `web_url` of the function, which also appears in the terminal output when running `modal deploy`.

Addenda

The rest of the code in this example is just utility code.

```
warnings.filterwarnings(  # filter warning from the terminal image library
    "ignore",
    message="It seems this process is not running within a terminal. Hence, some features
    category=UserWarning,
)


class Colors:
    """ANSI color codes"""

    GREEN = "\033[0;32m"
    BLUE = "\033[0;34m"
    GRAY = "\033[0;90m"
    BOLD = "\033[1m"
    END = "\033[0m"
```