```
import base64
import os
import time
from io import BytesIO
from typing import List, Literal, Optional, Tuple, Union

import torch
from PIL import Image
from pydantic import BaseModel, Field
from transformers import (
    AutoModelForCausalLM,
    LlamaTokenizer,
    TextIteratorStreamer,
)

from swarm_models.base_multimodal_model import BaseMultiModalModel
from loguru import logger

MODEL_PATH = "THUDM/cogvlm-chat-hf"
TOKENIZER_PATH = "lmsys/vicuna-7b-v1.5"
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
QUANT_ENABLED = False


class ImageUrl(BaseModel):
    url: str
```

```
class TextContent(BaseModel):
    type: Literal["text"]
    text: str


class ImageUrlContent(BaseModel):
    type: Literal["image_url"]
    image_url: ImageUrl


ContentItem = Union[TextContent, ImageUrlContent]


class ChatMessageInput(BaseModel):
    role: Literal["user", "assistant", "system"]
    content: Union[str, List[ContentItem]]
    name: Optional[str] = None


class ChatMessageResponse(BaseModel):
    role: Literal["assistant"]
    content: str = None
    name: Optional[str] = None
```

```
class DeltaMessage(BaseModel):
    role: Optional[Literal["user", "assistant", "system"]] = None
    content: Optional[str] = None


class ChatCompletionRequest(BaseModel):
    model: str
    messages: List[ChatMessageInput]
    temperature: Optional[float] = 0.8
    top_p: Optional[float] = 0.8
    max_tokens: Optional[int] = None
    stream: Optional[bool] = False
    # Additional parameters
    repetition_penalty: Optional[float] = 1.0


class ChatCompletionResponseChoice(BaseModel):
    index: int
    message: ChatMessageResponse


class ChatCompletionResponseStreamChoice(BaseModel):
    index: int
    delta: DeltaMessage
```

```
class UsageInfo(BaseModel):
    prompt_tokens: int = 0
    total_tokens: int = 0
    completion_tokens: Optional[int] = 0


class ChatCompletionResponse(BaseModel):
    model: str
    object: Literal["chat.completion", "chat.completion.chunk"]
    choices: List[
        Union[
            ChatCompletionResponseChoice,
            ChatCompletionResponseStreamChoice,
        ]
    ]
    created: Optional[int] = Field(
        default_factory=lambda: int(time.time())
    )
    usage: Optional[UsageInfo] = None


# async def create_chat_completion(request: ChatCompletionRequest):
#     global model, tokenizer

#     gen_params = dict(
```

```
#         messages=request.messages,
#         temperature=request.temperature,
#         top_p=request.top_p,
#         max_tokens=request.max_tokens or 1024,
#         echo=False,
#         stream=request.stream,
#     )

#     # if request.stream:
#         # predict(request.model, gen_params)
#     # response = generate_cogvlm(model, tokenizer, gen_params)

#     usage = UsageInfo()

#     message = ChatMessageResponse(
#         role="assistant",
#         content=response["text"],
#     )
#     logger.debug(f"==== message ====\n{message}")
#     choice_data = ChatCompletionResponseChoice(
#         index=0,
#         message=message,
#     )
#     task_usage = UsageInfo.model_validate(response["usage"])
#     for usage_key, usage_value in task_usage.model_dump().items():
#         setattr(
```

```
#             usage, usage_key, getattr(usage, usage_key) + usage_value
#         )
#     return ChatCompletionResponse(
#         model=request.model,
#         choices=[choice_data],
#         object="chat.completion",
#         usage=usage,
#     )


class CogVLMMultiModal(BaseMultiModalModel):
    """
    Initializes the CogVLM model.

    Args:
        model_name (str): The path or name of the pre-trained model.
        tokenizer (str): The path or name of the tokenizer.
        device (str): The device to run the model on.
        quantize (bool): Whether to enable quantization.
        torch_type (str): The torch data type to use.
        temperature (float): The temperature for sampling.
        top_p (float): The top-p value for sampling.
        max_tokens (int): The maximum number of tokens to generate.
        echo (bool): Whether to echo the input text.
        stream (bool): Whether to stream the output.
        repetition_penalty (float): The repetition penalty for sampling.
```

do_sample (bool): Whether to use sampling during generation.

*args: Additional positional arguments.

**kwargs: Additional keyword arguments.

Methods:

run: Generates a response using the CogVLM model.

generate_stream_cogvlm: Generates a stream of responses using the CogVLM model in inference mode.

process_history_and_images: Processes history messages to extract text, identify the last user query, and convert base64 encoded image URLs to PIL images.

Example:

>>> model = CogVLMMultiModal()

>>> response = model("Describe this image with meticlous details.", "https://example.com/image.jpg")

>>> print(response)

"""

```
def __init__(
    self,
    model_name: str = MODEL_PATH,
    tokenizer: str = TOKENIZER_PATH,
    device: str = DEVICE,
    quantize: bool = QUANT_ENABLED,
    torch_type: str = "float16",
    temperature: float = 0.5,
```

```
    top_p: float = 0.9,
    max_tokens: int = 3500,
    echo: bool = False,
    stream: bool = False,
    repetition_penalty: float = 1.0,
    do_sample: bool = True,
    *args,
    **kwargs,
):
    super().__init__()
    self.model_name = model_name
    self.device = device
    self.tokenizer = tokenizer
    self.device = device
    self.quantize = quantize
    self.torch_type = torch_type
    self.temperature = temperature
    self.top_p = top_p
    self.max_tokens = max_tokens
    self.echo = echo
    self.stream = stream
    self.repetition_penalty = repetition_penalty
    self.do_sample = do_sample

    if os.environ.get("QUANT_ENABLED"):
        pass
```

```
else:
    with torch.cuda.device(device):
        __, total_bytes = torch.cuda.mem_get_info()
        total_gb = total_bytes / (1 << 30)
        if total_gb < 40:
            pass

torch.cuda.empty_cache()

self.tokenizer = LlamaTokenizer.from_pretrained(
    tokenizer, trust_remote_code=True
)

if (
    torch.cuda.is_available()
    and torch.cuda.get_device_capability()[0] >= 8
):
    torch_type = torch.bfloat16
else:
    torch_type = torch.float16

print(
    f"========Use torch type as:{torch_type} with"
    f" device:{device}========\n\n"
)
```

```
if "cuda" in device:
    if QUANT_ENABLED:
        self.model = AutoModelForCausalLM.from_pretrained(
            model_name,
            load_in_4bit=True,
            trust_remote_code=True,
            torch_dtype=torch_type,
            low_cpu_mem_usage=True,
            *args,
            **kwargs,
        ).eval()
    else:
        self.model = (
            AutoModelForCausalLM.from_pretrained(
                model_name,
                load_in_4bit=False,
                trust_remote_code=True,
                torch_dtype=torch_type,
                low_cpu_mem_usage=True,
                *args,
                **kwargs,
            )
            .to(device)
            .eval()
        )
```

```
        else:
            self.model = (
                AutoModelForCausalLM.from_pretrained(
                    model_name,
                    trust_remote_code=True,
                    *args,
                    **kwargs,
                )
                .float()
                .to(device)
                .eval()
            )

    def run(self, task: str, img: str, *args, **kwargs):
        """
        Generates a response using the CogVLM model. It processes the chat history and image data,
if any,
        and then invokes the model to generate a response.
        """
        messages = [task]

        params = dict(
            messages=messages,
            temperature=self.temperature,
            repitition_penalty=self.repetition_penalty,
            top_p=self.top_p,
```

```
            max_new_tokens=self.max_tokens,
        )

        for response in self.generate_stream_cogvlm(params):
            pass

        return response

    @torch.inference_mode()
    def generate_stream_cogvlm(
        self,
        params: dict,
    ):
        """
        Generates a stream of responses using the CogVLM model in inference mode.
            It's optimized to handle continuous input-output interactions with the model in a streaming manner.
        """
        messages = params["messages"]
        temperature = float(params.get("temperature", 1.0))
        repetition_penalty = float(
            params.get("repetition_penalty", 1.0)
        )
        top_p = float(params.get("top_p", 1.0))
        max_new_tokens = int(params.get("max_tokens", 256))
        query, history, image_list = self.process_history_and_images(
```

```
    messages
)

logger.debug(f"==== request ====\n{query}")

input_by_model = self.model.build_conversation_input_ids(
    self.tokenizer,
    query=query,
    history=history,
    images=[image_list[-1]],
)
inputs = {
    "input_ids": (
        input_by_model["input_ids"]
        .unsqueeze(0)
        .to(self.device)
    ),
    "token_type_ids": (
        input_by_model["token_type_ids"]
        .unsqueeze(0)
        .to(self.device)
    ),
    "attention_mask": (
        input_by_model["attention_mask"]
        .unsqueeze(0)
        .to(self.device)
```

```
    ),
    "images": [
        [
            input_by_model["images"][0]
            .to(self.device)
            .to(self.torch_type)
        ]
    ],
}
if (
    "cross_images" in input_by_model
    and input_by_model["cross_images"]
):
    inputs["cross_images"] = [
        [
            input_by_model["cross_images"][0]
            .to(self.device)
            .to(self.torch_type)
        ]
    ]

input_echo_len = len(inputs["input_ids"][0])
streamer = TextIteratorStreamer(
    tokenizer=self.tokenizer,
    timeout=60.0,
    skip_promptb=True,
```

```
    skip_special_tokens=True,
)
gen_kwargs = {
    "repetition_penalty": repetition_penalty,
    "max_new_tokens": max_new_tokens,
    "do_sample": True if temperature > 1e-5 else False,
    "top_p": top_p if temperature > 1e-5 else 0,
    "streamer": streamer,
}
if temperature > 1e-5:
    gen_kwargs["temperature"] = temperature

total_len = 0
generated_text = ""
with torch.no_grad():
    self.model.generate(**inputs, **gen_kwargs)
    for next_text in streamer:
        generated_text += next_text
        yield {
            "text": generated_text,
            "usage": {
                "prompt_tokens": input_echo_len,
                "completion_tokens": (
                    total_len - input_echo_len
                ),
                "total_tokens": total_len,
```

```
                },
            }
        ret = {
            "text": generated_text,
            "usage": {
                "prompt_tokens": input_echo_len,
                "completion_tokens": total_len - input_echo_len,
                "total_tokens": total_len,
            },
        }
        yield ret


def process_history_and_images(
    self,
    messages: List[ChatMessageInput],
) -> Tuple[
    Optional[str],
    Optional[List[Tuple[str, str]]],
    Optional[List[Image.Image]],
]:
    """
    Process history messages to extract text, identify the last user query,
    and convert base64 encoded image URLs to PIL images.

    Args:
        messages(List[ChatMessageInput]): List of ChatMessageInput objects.
```

```
return: A tuple of three elements:
        - The last user query as a string.
        - Text history formatted as a list of tuples for the model.
        - List of PIL Image objects extracted from the messages.
"""
formatted_history = []
image_list = []
last_user_query = ""

for i, message in enumerate(messages):
    role = message.role
    content = message.content

    # Extract text content
    if isinstance(content, list):  # text
        text_content = " ".join(
            item.text
            for item in content
            if isinstance(item, TextContent)
        )
    else:
        text_content = content

    # Extract image data
    if isinstance(content, list):  # image
        for item in content:
```

```
        if isinstance(item, ImageUrlContent):
            image_url = item.image_url.url
            if image_url.startswith(
                ""
            ):
                base64_encoded_image = image_url.split(
                    ""
                )[1]
                image_data = base64.b64decode(
                    base64_encoded_image
                )
                image = Image.open(
                    BytesIO(image_data)
                ).convert("RGB")
                image_list.append(image)

# Format history
if role == "user":
    if i == len(messages) - 1:
        last_user_query = text_content
    else:
        formatted_history.append((text_content, ""))
elif role == "assistant":
    if formatted_history:
        if formatted_history[-1][1] != "":
            raise AssertionError(
```

```
                    "the last query is answered. answer"
                    f" again. {formatted_history[-1][0]},"
                    f" {formatted_history[-1][1]},"
                    f" {text_content}"
                )
            formatted_history[-1] = (
                formatted_history[-1][0],
                text_content,
            )
        else:
            raise AssertionError(
                "assistant reply before user"
            )
    else:
        raise AssertionError(f"unrecognized role: {role}")

return last_user_query, formatted_history, image_list

async def predict(self, params: dict):
    """
    Handle streaming predictions. It continuously generates responses for a given input stream.
    This is particularly useful for real-time, continuous interactions with the model.
    """

    choice_data = ChatCompletionResponseStreamChoice(
        index=0,
```

```
    delta=DeltaMessage(role="assistant"),
    finish_reason=None,
)
chunk = ChatCompletionResponse(
    model=self.model_name,
    choices=[choice_data],
    object="chat.completion.chunk",
)
yield f"{chunk.model_dump_json(exclude_unset=True)}"

previous_text = ""
for new_response in self.generate_stream_cogvlm(params):
    decoded_unicode = new_response["text"]
    delta_text = decoded_unicode[len(previous_text) :]
    previous_text = decoded_unicode
    delta = DeltaMessage(
        content=delta_text,
        role="assistant",
    )
    choice_data = ChatCompletionResponseStreamChoice(
        index=0,
        delta=delta,
    )
    chunk = ChatCompletionResponse(
        model=self.model_name,
        choices=[choice_data],
```

```
            object="chat.completion.chunk",
        )
        yield f"{chunk.model_dump_json(exclude_unset=True)}"
    choice_data = ChatCompletionResponseStreamChoice(
        index=0,
        delta=DeltaMessage(),
    )
    chunk = ChatCompletionResponse(
        model=self.model_name,
        choices=[choice_data],
        object="chat.completion.chunk",
    )
    yield f"{chunk.model_dump_json(exclude_unset=True)}"
```