```
# coding=utf-8
# Implements API for Qwen-7B in OpenAI's format. (https://platform.openai.com/docs/api-reference/chat)
# Usage: python openai_api.py
# Visit http://localhost:8000/docs for documents.

import re
import copy
import json
import time
from argparse import ArgumentParser
from contextlib import asynccontextmanager
from typing import Dict, List, Literal, Optional, Union

import torch
import uvicorn
from fastapi import FastAPI, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel, Field
from transformers import AutoTokenizer, AutoModelForCausalLM
from transformers.generation import GenerationConfig


@asynccontextmanager
async def lifespan(app: FastAPI):  # collects GPU memory
    yield
```

```
    if torch.cuda.is_available():
        torch.cuda.empty_cache()
        torch.cuda.ipc_collect()


app = FastAPI(lifespan=lifespan)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)


class ModelCard(BaseModel):
    id: str
    object: str = "model"
    created: int = Field(default_factory=lambda: int(time.time()))
    owned_by: str = "owner"
    root: Optional[str] = None
    parent: Optional[str] = None
    permission: Optional[list] = None
```

```
class ModelList(BaseModel):
    object: str = "list"
    data: List[ModelCard] = []


class ChatMessage(BaseModel):
    role: Literal["user", "assistant", "system", "function"]
    content: Optional[str]
    function_call: Optional[Dict] = None


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


class ChatCompletionRequest(BaseModel):
    model: str
    messages: List[ChatMessage]
    functions: Optional[List[Dict]] = None
    temperature: Optional[float] = None
    top_p: Optional[float] = None
    max_length: Optional[int] = None
    stream: Optional[bool] = False
    stop: Optional[List[str]] = None
```

```
class ChatCompletionResponseChoice(BaseModel):
    index: int
    message: ChatMessage
    finish_reason: Literal["stop", "length", "function_call"]


class ChatCompletionResponseStreamChoice(BaseModel):
    index: int
    delta: DeltaMessage
    finish_reason: Optional[Literal["stop", "length"]]


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()))


@app.get("/v1/models", response_model=ModelList)
async def list_models():
    global model_args
    model_card = ModelCard(id="gpt-3.5-turbo")
```

```
    return ModelList(data=[model_card])


# To work around that unpleasant leading-\n tokenization issue!
def add_extra_stop_words(stop_words):
    if stop_words:
        _stop_words = []
        _stop_words.extend(stop_words)
        for x in stop_words:
            s = x.lstrip("\n")
            if s and (s not in _stop_words):
                _stop_words.append(s)
        return _stop_words
    return stop_words


def trim_stop_words(response, stop_words):
    if stop_words:
        for stop in stop_words:
            idx = response.find(stop)
            if idx != -1:
                response = response[:idx]
    return response


TOOL_DESC = """{name_for_model}: Call this tool to interact with the {name_for_human} API. What
```

is the {name_for_human} API useful for? {description_for_model} Parameters: {parameters}"""

REACT_INSTRUCTION = """Answer the following questions as best you can. You have access to the following APIs:

{tools_text}

Use the following format:

Question: the input question you must answer

Thought: you should always think about what to do

Action: the action to take, should be one of [{tools_name_text}]

Action Input: the input to the action

Observation: the result of the action

... (this Thought/Action/Action Input/Observation can be repeated zero or more times)

Thought: I now know the final answer

Final Answer: the final answer to the original input question

Begin!"""

_TEXT_COMPLETION_CMD = object()

#

Temporarily, the system role does not work as expected.

We advise that you write the setups for role-play in your query,

```
# i.e., use the user role instead of the system role.
#
# TODO: Use real system role when the model is ready.
#
def parse_messages(messages, functions):
    if all(m.role != "user" for m in messages):
        raise HTTPException(
            status_code=400,
            detail="Invalid request: Expecting at least one user message.",
        )

    messages = copy.deepcopy(messages)
    default_system = "You are a helpful assistant."
    system = ""
    if messages[0].role == "system":
        system = messages.pop(0).content.lstrip("\n").rstrip()
        if system == default_system:
            system = ""

    if functions:
        tools_text = []
        tools_name_text = []
        for func_info in functions:
            name = func_info.get("name", "")
            name_m = func_info.get("name_for_model", name)
            name_h = func_info.get("name_for_human", name)
```

```
        desc = func_info.get("description", "")
        desc_m = func_info.get("description_for_model", desc)
        tool = TOOL_DESC.format(
            name_for_model=name_m,
            name_for_human=name_h,
            # Hint: You can add the following format requirements in description:
            #   "Format the arguments as a JSON object."
            #   "Enclose the code within triple backticks (`) at the beginning and end of the code."
            description_for_model=desc_m,
            parameters=json.dumps(func_info["parameters"], ensure_ascii=False),
        )
        tools_text.append(tool)
        tools_name_text.append(name_m)
    tools_text = "\n\n".join(tools_text)
    tools_name_text = ", ".join(tools_name_text)
    system += "\n\n" + REACT_INSTRUCTION.format(
        tools_text=tools_text,
        tools_name_text=tools_name_text,
    )
    system = system.lstrip("\n").rstrip()


dummy_thought = {
    "en": "\nThought: I now know the final answer.\nFinal answer: ",
    "zh": "\nThought: \nFinal answer: ",
}
```

```
_messages = messages
messages = []
for m_idx, m in enumerate(_messages):
    role, content, func_call = m.role, m.content, m.function_call
    if content:
        content = content.lstrip("\n").rstrip()
    if role == "function":
        if (len(messages) == 0) or (messages[-1].role != "assistant"):
            raise HTTPException(
                status_code=400,
                detail="Invalid request: Expecting role assistant before role function.",
            )
        messages[-1].content += f"\nObservation: {content}"
        if m_idx == len(_messages) - 1:
            messages[-1].content += "\nThought:"
    elif role == "assistant":
        if len(messages) == 0:
            raise HTTPException(
                status_code=400,
                detail="Invalid request: Expecting role user before role assistant.",
            )
        last_msg = messages[-1].content
        last_msg_has_zh = len(re.findall(r"[\u4e00-\u9fff]+", last_msg)) > 0
        if func_call is None:
            if functions:
                content = dummy_thought["zh" if last_msg_has_zh else "en"] + content
```

```
        else:
            f_name, f_args = func_call["name"], func_call["arguments"]
            if not content:
                if last_msg_has_zh:
                    content = f"Thought:  {f_name} API"
                else:
                    content = f"Thought: I can use {f_name}."
            content = f"\n{content}\nAction: {f_name}\nAction Input: {f_args}"
        if messages[-1].role == "user":
            messages.append(
                ChatMessage(role="assistant", content=content.lstrip("\n").rstrip())
            )
        else:
            messages[-1].content += content
    elif role == "user":
        messages.append(
            ChatMessage(role="user", content=content.lstrip("\n").rstrip())
        )
    else:
        raise HTTPException(
            status_code=400, detail=f"Invalid request: Incorrect role {role}."
        )

query = _TEXT_COMPLETION_CMD
if messages[-1].role == "user":
    query = messages[-1].content
```

```
        messages = messages[:-1]


    if len(messages) % 2 != 0:
        raise HTTPException(status_code=400, detail="Invalid request")


    history = []  # [(Q1, A1), (Q2, A2), ..., (Q_last_turn, A_last_turn)]
    for i in range(0, len(messages), 2):
        if messages[i].role == "user" and messages[i + 1].role == "assistant":
            usr_msg = messages[i].content.lstrip("\n").rstrip()
            bot_msg = messages[i + 1].content.lstrip("\n").rstrip()
            if system and (i == len(messages) - 2):
                usr_msg = f"{system}\n\nQuestion: {usr_msg}"
                system = ""
            for t in dummy_thought.values():
                t = t.lstrip("\n")
                if bot_msg.startswith(t) and ("\nAction: " in bot_msg):
                    bot_msg = bot_msg[len(t) :]
            history.append([usr_msg, bot_msg])
        else:
            raise HTTPException(
                status_code=400,
                detail="Invalid request: Expecting exactly one user (or function) role before every assistant
role.",
            )
    if system:
        assert query is not _TEXT_COMPLETION_CMD
```

```
        query = f"{system}\n\nQuestion: {query}"
    return query, history


def parse_response(response):
    func_name, func_args = "", ""
    i = response.rfind("\nAction:")
    j = response.rfind("\nAction Input:")
    k = response.rfind("\nObservation:")
    if 0 <= i < j:  # If the text has `Action` and `Action input`,
        if k < j:  # but does not contain `Observation`,
            # then it is likely that `Observation` is omitted by the LLM,
            # because the output text may have discarded the stop word.
            response = response.rstrip() + "\nObservation:"  # Add it back.
        k = response.rfind("\nObservation:")
        func_name = response[i + len("\nAction:") : j].strip()
        func_args = response[j + len("\nAction Input:") : k].strip()
    if func_name:
        choice_data = ChatCompletionResponseChoice(
            index=0,
            message=ChatMessage(
                role="assistant",
                content=response[:i],
                function_call={"name": func_name, "arguments": func_args},
            ),
            finish_reason="function_call",
```

```
        )
        return choice_data
    z = response.rfind("\nFinal Answer: ")
    if z >= 0:
        response = response[z + len("\nFinal Answer: ") :]
    choice_data = ChatCompletionResponseChoice(
        index=0,
        message=ChatMessage(role="assistant", content=response),
        finish_reason="stop",
    )
    return choice_data


# completion mode, not chat mode
def text_complete_last_message(history, stop_words_ids):
    im_start = "<|im_start|>"
    im_end = "<|im_end|>"
    prompt = f"{im_start}system\nYou are a helpful assistant.{im_end}"
    for i, (query, response) in enumerate(history):
        query = query.lstrip("\n").rstrip()
        response = response.lstrip("\n").rstrip()
        prompt += f"\n{im_start}user\n{query}{im_end}"
        prompt += f"\n{im_start}assistant\n{response}{im_end}"
    prompt = prompt[: -len(im_end)]

    _stop_words_ids = [tokenizer.encode(im_end)]
```

```
    if stop_words_ids:
        for s in stop_words_ids:
            _stop_words_ids.append(s)
    stop_words_ids = _stop_words_ids

    input_ids = torch.tensor([tokenizer.encode(prompt)]).to(model.device)
    output = model.generate(input_ids, stop_words_ids=stop_words_ids).tolist()[0]
    output = tokenizer.decode(output, errors="ignore")
    assert output.startswith(prompt)
    output = output[len(prompt) :]
    output = trim_stop_words(output, ["<|endoftext|>", im_end])
    print(f"<completion>\n{prompt}\n<!-- *** -->\n{output}\n</completion>")
    return output


@app.post("/v1/chat/completions", response_model=ChatCompletionResponse)
async def create_chat_completion(request: ChatCompletionRequest):
    global model, tokenizer

    stop_words = add_extra_stop_words(request.stop)
    if request.functions:
        stop_words = stop_words or []
        if "Observation:" not in stop_words:
            stop_words.append("Observation:")

    query, history = parse_messages(request.messages, request.functions)
```

```
if request.stream:
    if request.functions:
        raise HTTPException(
            status_code=400,
            detail="Invalid request: Function calling is not yet implemented for stream mode.",
        )
    # generate = predict(query, history, request.model, stop_words)
    # return EventSourceResponse(generate, media_type="text/event-stream")
    raise HTTPException(
        status_code=400, detail="Stream request is not supported currently."
    )

stop_words_ids = [tokenizer.encode(s) for s in stop_words] if stop_words else None
if query is _TEXT_COMPLETION_CMD:
    response = text_complete_last_message(history, stop_words_ids=stop_words_ids)
else:
    response, _ = model.chat(
        tokenizer,
        query,
        history=history,
        stop_words_ids=stop_words_ids,
        append_history=False,
        top_p=request.top_p,
        temperature=request.temperature,
    )
```

```
        print(f"<chat>\n{history}\n{query}\n<!-- *** -->\n{response}\n</chat>")
    response = trim_stop_words(response, stop_words)
    if request.functions:
        choice_data = parse_response(response)
    else:
        choice_data = ChatCompletionResponseChoice(
            index=0,
            message=ChatMessage(role="assistant", content=response),
            finish_reason="stop",
        )
    return ChatCompletionResponse(
        model=request.model, choices=[choice_data], object="chat.completion"
    )


async def predict(
    query: str, history: List[List[str]], model_id: str, stop_words: List[str]
):
    global model, tokenizer
    choice_data = ChatCompletionResponseStreamChoice(
        index=0, delta=DeltaMessage(role="assistant"), finish_reason=None
    )
    chunk = ChatCompletionResponse(
        model=model_id, choices=[choice_data], object="chat.completion.chunk"
    )
    yield f"{chunk.model_dump_json(exclude_unset=True)}"
```

```
current_length = 0
stop_words_ids = [tokenizer.encode(s) for s in stop_words] if stop_words else None
if stop_words:
    # TODO: It's a little bit tricky to trim stop words in the stream mode.
    raise HTTPException(
        status_code=400,
        detail="Invalid request: custom stop words are not yet supported for stream mode.",
    )
response_generator = model.chat_stream(
    tokenizer, query, history=history, stop_words_ids=stop_words_ids
)
for new_response in response_generator:
    if len(new_response) == current_length:
        continue

    new_text = new_response[current_length:]
    current_length = len(new_response)

    choice_data = ChatCompletionResponseStreamChoice(
        index=0, delta=DeltaMessage(content=new_text), finish_reason=None
    )
    chunk = ChatCompletionResponse(
        model=model_id, choices=[choice_data], object="chat.completion.chunk"
    )
    yield f"{chunk.model_dump_json(exclude_unset=True)}"
```

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


def _get_args():
    parser = ArgumentParser()
    parser.add_argument(
        "-c",
        "--checkpoint-path",
        type=str,
        default="QWen/QWen-7B-Chat",
        help="Checkpoint name or path, default to %(default)r",
    )
    parser.add_argument(
        "--cpu-only", action="store_true", help="Run demo with CPU only"
    )
    parser.add_argument(
        "--server-port", type=int, default=8000, help="Demo server port."
    )
```

```
    parser.add_argument(
        "--server-name",
        type=str,
        default="127.0.0.1",
        help="Demo server name. Default: 127.0.0.1, which is only visible from the local computer."
        " If you want other computers to access your server, use 0.0.0.0 instead.",
    )

    args = parser.parse_args()
    return args


if __name__ == "__main__":
    args = _get_args()

    tokenizer = AutoTokenizer.from_pretrained(
        args.checkpoint_path,
        trust_remote_code=True,
        resume_download=True,
    )

    if args.cpu_only:
        device_map = "cpu"
    else:
        device_map = "auto"
```

```
model = AutoModelForCausalLM.from_pretrained(
    args.checkpoint_path,
    device_map=device_map,
    trust_remote_code=True,
    resume_download=True,
).eval()

model.generation_config = GenerationConfig.from_pretrained(
    args.checkpoint_path,
    trust_remote_code=True,
    resume_download=True,
)

uvicorn.run(app, host=args.server_name, port=args.server_port, workers=1)
```