```
import asyncio
import concurrent.futures
import logging
from typing import List, Tuple

import torch
from termcolor import colored
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    BitsAndBytesConfig,
)

from swarm_models.base_llm import BaseLLM


class HuggingfaceLLM(BaseLLM):
    """
    A class for running inference on a given model.

    Attributes:
        model_id (str): The ID of the model.
        device (str): The device to run the model on (either 'cuda' or 'cpu').
        max_length (int): The maximum length of the output sequence.
        quantize (bool, optional): Whether to use quantization. Defaults to False.
        quantization_config (dict, optional): The configuration for quantization.
    verbose (bool, optional): Whether to print verbose logs. Defaults to False.

    logger (logging.Logger, optional): The logger to use. Defaults to a basic logger.


Methods:

    run(task: str, max_length: int = 500) -> str:

        Generate a response based on the prompt text.


    __call__(task: str, max_length: int = 500) -> str:

        Generate a response based on the prompt text.


    save_model(path: str):

        Save the model to a given path.


    gpu_available() -> bool:

        Check if GPU is available.


    memory_consumption() -> dict:

        Get the memory consumption of the GPU.


    print_dashboard(task: str):

        Print dashboard.


    set_device(device: str):

        Changes the device used for inference.


    set_max_length(max_length: int):
```

Set max_length.

set_verbose(verbose: bool):

Set verbose.

set_distributed(distributed: bool):

Set distributed.

set_decoding(decoding: bool):

Set decoding.

set_max_workers(max_workers: int):

Set max_workers.

set_repitition_penalty(repitition_penalty: float):

Set repitition_penalty.

set_no_repeat_ngram_size(no_repeat_ngram_size: int):

Set no_repeat_ngram_size.

set_temperature(temperature: float):

Set temperature.

set_top_k(top_k: int):

Set top_k.

set_top_p(top_p: float):

Set top_p.

set_quantize(quantize: bool):

Set quantize.

set_quantization_config(quantization_config: dict):

Set quantization_config.

set_model_id(model_id: str):

Set model_id.

set_model(model):

Set model.

set_tokenizer(tokenizer):

Set tokenizer.

set_logger(logger):

Set logger.

Examples:

```
>>> llm = HuggingfaceLLM(
...     model_id="EleutherAI/gpt-neo-2.7B",
...     device="cuda",
    ...     max_length=500,
    ...     quantize=True,
    ...     quantization_config={
    ...         "load_in_4bit": True,
    ...         "bnb_4bit_use_double_quant": True,
    ...         "bnb_4bit_quant_type": "nf4",
    ...         "bnb_4bit_compute_dtype": torch.bfloat16,
    ...     },
    ... )
    >>> llm("Generate a 10,000 word blog on mental clarity and the benefits of meditation.")
    'Generate a 10,000 word
"""

def __init__(
    self,
    model_id: str,
    device: str = None,
    max_length: int = 500,
    quantize: bool = False,
    quantization_config: dict = None,
    verbose=False,
    distributed=False,
    decoding=False,
    max_workers: int = 5,
    repitition_penalty: float = 1.3,
    no_repeat_ngram_size: int = 5,
    temperature: float = 0.7,
    top_k: int = 40,
    top_p: float = 0.8,
    dtype=torch.bfloat16,
    *args,
    **kwargs,
):
    super().__init__(*args, **kwargs)
    self.logger = logging.getLogger(__name__)
    self.device = (
        device
        if device
        else ("cuda" if torch.cuda.is_available() else "cpu")
    )
    self.model_id = model_id
    self.max_length = max_length
    self.verbose = verbose
    self.distributed = distributed
    self.decoding = decoding
    self.quantize = quantize
    self.quantization_config = quantization_config
    self.max_workers = max_workers
    self.repitition_penalty = repitition_penalty
    self.no_repeat_ngram_size = no_repeat_ngram_size
    self.temperature = temperature
    self.top_k = top_k
```

```
self.top_p = top_p
self.dtype = dtype

if self.distributed:
    assert (
        torch.cuda.device_count() > 1
    ), "You need more than 1 gpu for distributed processing"

bnb_config = None
if quantize:
    if not quantization_config:
        quantization_config = {
            "load_in_4bit": True,
            "bnb_4bit_use_double_quant": True,
            "bnb_4bit_quant_type": "nf4",
            "bnb_4bit_compute_dtype": dtype,
        }
    bnb_config = BitsAndBytesConfig(**quantization_config)

self.tokenizer = AutoTokenizer.from_pretrained(self.model_id)

if quantize:
    self.model = AutoModelForCausalLM.from_pretrained(
        self.model_id,
        quantization_config=bnb_config,
        *args,
            **kwargs,
        )
    else:
        self.model = AutoModelForCausalLM.from_pretrained(
            self.model_id, *args, **kwargs
        ).to(self.device)

def print_error(self, error: str):
    """Print error"""
    print(colored(f"Error: {error}", "red"))

async def async_run(self, task: str):
    """Ashcnronous generate text for a given prompt"""
    return await asyncio.to_thread(self.run, task)

def concurrent_run(self, tasks: List[str], max_workers: int = 5):
    """Concurrently generate text for a list of prompts."""
    with concurrent.futures.ThreadPoolExecutor(
        max_workers=max_workers
    ) as executor:
        results = list(executor.map(self.run, tasks))
    return results

def run_batch(
    self, tasks_images: List[Tuple[str, str]]
) -> List[str]:
    """Process a batch of tasks and images"""
    with concurrent.futures.ThreadPoolExecutor() as executor:
        futures = [
            executor.submit(self.run, task, img)
            for task, img in tasks_images
        ]
        results = [future.result() for future in futures]
    return results

def run(self, task: str, *args, **kwargs):
    """
    Generate a response based on the prompt text.

    Args:
    - task (str): Text to prompt the model.
    - max_length (int): Maximum length of the response.

    Returns:
    - Generated text (str).
    """
    try:
        inputs = self.tokenizer.encode(task, return_tensors="pt")

        if self.decoding:
            with torch.no_grad():
                for _ in range(self.max_length):
```

```
            output_sequence = []

            outputs = self.model.generate(
                inputs,
                max_length=len(inputs) + 1,
                do_sample=True,
            )
            output_tokens = outputs[0][-1]
            output_sequence.append(output_tokens.item())

            # print token in real-time
            print(
                self.tokenizer.decode(
                    [output_tokens],
                    skip_special_tokens=True,
                ),
                end="",
                flush=True,
            )
            inputs = outputs
else:
    with torch.no_grad():
        outputs = self.model.generate(
            inputs,
            max_length=self.max_length,
            do_sample=True,
                    *args,
                    **kwargs,
                )

        return self.tokenizer.decode(
            outputs[0], skip_special_tokens=True
        )
    except Exception as e:
        print(
            colored(
                (
                    "HuggingfaceLLM could not generate text"
                    f" because of error: {e}, try optimizing your"
                    " arguments"
                ),
                "red",
            )
        )
        raise

def __call__(self, task: str, *args, **kwargs):
    return self.run(task, *args, **kwargs)

async def __call_async__(self, task: str, *args, **kwargs) -> str:
    """Call the model asynchronously""" ""
    return await self.run_async(task, *args, **kwargs)
```

```
def save_model(self, path: str):
    """Save the model to a given path"""
    self.model.save_pretrained(path)
    self.tokenizer.save_pretrained(path)

def gpu_available(self) -> bool:
    """Check if GPU is available"""
    return torch.cuda.is_available()

def memory_consumption(self) -> dict:
    """Get the memory consumption of the GPU"""
    if self.gpu_available():
        torch.cuda.synchronize()
        allocated = torch.cuda.memory_allocated()
        reserved = torch.cuda.memory_reserved()
        return {"allocated": allocated, "reserved": reserved}
    else:
        return {"error": "GPU not available"}

def print_dashboard(self, task: str):
    """Print dashboard"""

    dashboard = print(
        colored(
            f"""
            HuggingfaceLLM Dashboard
            --------------------------------------------
            Model Name: {self.model_id}
            Tokenizer: {self.tokenizer}
            Model MaxLength: {self.max_length}
            Model Device: {self.device}
            Model Quantization: {self.quantize}
            Model Quantization Config: {self.quantization_config}
            Model Verbose: {self.verbose}
            Model Distributed: {self.distributed}
            Model Decoding: {self.decoding}

            ----------------------------------------
            Metadata:
                Task Memory Consumption: {self.memory_consumption()}
                GPU Available: {self.gpu_available()}
            ----------------------------------------

            Task Environment:
                Task: {task}

            """,
            "red",
        )
    )
```

```
    print(dashboard)


def set_device(self, device):
    """
    Changes the device used for inference.


    Parameters
    ----------
        device : str
            The new device to use for inference.
    """
    self.device = device
    if self.model is not None:
        self.model.to(self.device)


def set_max_length(self, max_length):
    """Set max_length"""
    self.max_length = max_length


def clear_chat_history(self):
    """Clear chat history"""
    self.chat_history = []


def set_verbose(self, verbose):
    """Set verbose"""
    self.verbose = verbose
```

```
def set_distributed(self, distributed):
    """Set distributed"""
    self.distributed = distributed

def set_decoding(self, decoding):
    """Set decoding"""
    self.decoding = decoding

def set_max_workers(self, max_workers):
    """Set max_workers"""
    self.max_workers = max_workers

def set_repitition_penalty(self, repitition_penalty):
    """Set repitition_penalty"""
    self.repitition_penalty = repitition_penalty

def set_no_repeat_ngram_size(self, no_repeat_ngram_size):
    """Set no_repeat_ngram_size"""
    self.no_repeat_ngram_size = no_repeat_ngram_size

def set_temperature(self, temperature):
    """Set temperature"""
    self.temperature = temperature

def set_top_k(self, top_k):
    """Set top_k"""
    self.top_k = top_k

def set_top_p(self, top_p):
    """Set top_p"""
    self.top_p = top_p

def set_quantize(self, quantize):
    """Set quantize"""
    self.quantize = quantize

def set_quantization_config(self, quantization_config):
    """Set quantization_config"""
    self.quantization_config = quantization_config

def set_model_id(self, model_id):
    """Set model_id"""
    self.model_id = model_id

def set_model(self, model):
    """Set model"""
    self.model = model

def set_tokenizer(self, tokenizer):
    """Set tokenizer"""
    self.tokenizer = tokenizer
```

```
def set_logger(self, logger):

    """Set logger"""

    self.logger = logger
```