lmdeploy Documentation

Release 0.6.5

LMDeploy Contributors

Jan 13, 2025

GET STARTED

LMDeploy has the following core features:

- **Efficient Inference**: LMDeploy delivers up to 1.8x higher request throughput than vLLM, by introducing key features like persistent batch(a.k.a. continuous batching), blocked KV cache, dynamic split&fuse, tensor parallelism, high-performance CUDA kernels and so on.
- **Effective Quantization**: LMDeploy supports weight-only and k/v quantization, and the 4-bit inference performance is 2.4x higher than FP16. The quantization quality has been confirmed via OpenCompass evaluation.
- **Effortless Distribution Server**: Leveraging the request distribution service, LMDeploy facilitates an easy and efficient deployment of multi-model services across multiple machines and cards.
- **Interactive Inference Mode**: By caching the k/v of attention during multi-round dialogue processes, the engine remembers dialogue history, thus avoiding repetitive processing of historical sessions.
- **Excellent Compatibility**: LMDeploy supports KV Cache Quant, AWQ and Automatic Prefix Caching to be used simultaneously.

CHAPTER

ONE

DOCUMENTATION

1.1 Installation

LMDeploy is a python library for compressing, deploying, and serving Large Language Models(LLMs) and Vision-Language Models(VLMs). Its core inference engines include TurboMind Engine and PyTorch Engine. The former is developed by C++ and CUDA, striving for ultimate optimization of inference performance, while the latter, developed purely in Python, aims to decrease the barriers for developers.

It supports LLMs and VLMs deployment on both Linux and Windows platform, with minimum requirement of CUDA version 11.3. Furthermore, it is compatible with the following NVIDIA GPUs:

- Volta(sm70): V100
- Turing(sm75): 20 series, T4
- Ampere(sm80,sm86): 30 series, A10, A16, A30, A100
- Ada Lovelace(sm89): 40 series

1.1.1 Install with pip (Recommend)

It is recommended installing lmdeploy using pip in a conda environment (python 3.8 - 3.12):

```
conda create -n lmdeploy python=3.8 -y
conda activate lmdeploy
pip install lmdeploy
```

The default prebuilt package is compiled on **CUDA 12**. If CUDA 11+ (>=11.3) is required, you can install lmdeploy by:

```
export LMDEPLOY_VERSION=0.6.5
export PYTHON_VERSION=38
pip install https://github.com/InternLM/lmdeploy/releases/download/v${LMDEPLOY_VERSION}/
↪lmdeploy-${LMDEPLOY_VERSION}+cu118-cp${PYTHON_VERSION}-cp${PYTHON_VERSION}-
↪manylinux2014_x86_64.whl --extra-index-url https://download.pytorch.org/whl/cu118
```

1.1.2 Install nightly-build package with pip

The release frequency of LMDeploy is approximately once or twice monthly. If your desired feature has been merged to LMDeploy main branch but hasn't been published yet, you can experiment with the nightly-built package available here according to your CUDA and Python versions

1.1.3 Install from source

If you are using the PyTorch Engine for inference, the installation from the source is quite simple:

```
git clone https://github.com/InternLM/lmdeploy.git
cd lmdeploy
pip install -e .
```

But if you are using the TurboMind Engine, you have to build the source as shown below. The `openmmlab/lmdeploy:{tag}` docker image is strongly recommended.

Step 1 - Get the docker image of LMDeploy

```
docker pull openmmlab/lmdeploy:latest
```

Note

The "openmmlab/lmdeploy:latest" is based on "nvidia/cuda:12.4.1-devel-ubuntu22.04". If you are working on a platform with cuda 11+ driver, please use "openmmlab/lmdeploy:latest-cu11". The pattern of the LMDeploy docker image tag is "openmmlab/lmdeploy:{version}-cu(11|12)" since v0.5.3.

Step 2 - Clone LMDeploy source code and change to its root directory

```
git clone https://github.com/InternLM/lmdeploy.git
cd lmdeploy
```

Step 3 - launch docker container in interactive mode

```
docker run --gpus all --net host --shm-size 16g -v $(pwd):/opt/lmdeploy --name lmdeploy -
↪it openmmlab/lmdeploy:latest bin/bash
```

Step 4 - build and installation

```
cd /opt/lmdeploy
mkdir -p build && cd build
bash ../generate.sh make
make -j$(nproc) && make install
cd ..
pip install -e .
```

1.2 Quick Start

This tutorial shows the usage of LMDeploy on CUDA platform:

- Offline inference of LLM model and VLM model
- Serve a LLM or VLM model by the OpenAI compatible server
- Console CLI to interactively chat with LLM model

Before reading further, please ensure that you have installed lmdeploy as outlined in the *installation guide*

1.2.1 Offline batch inference

LLM inference

```
from lmdeploy import pipeline
pipe = pipeline('internlm/internlm2_5-7b-chat')
response = pipe(['Hi, pls intro yourself', 'Shanghai is'])
print(response)
```

When constructing the `pipeline`, if an inference engine is not designated between the TurboMind Engine and the PyTorch Engine, LMDeploy will automatically assign one based on *their respective capabilities*, with the TurboMind Engine taking precedence by default.

However, you have the option to manually select an engine. For instance,

```
from lmdeploy import pipeline, TurbomindEngineConfig
pipe = pipeline('internlm/internlm2_5-7b-chat',
                backend_config=TurbomindEngineConfig(
                    max_batch_size=32,
                    enable_prefix_caching=True,
                    cache_max_entry_count=0.8,
                    session_len=8192,
                ))
```

or,

```
from lmdeploy import pipeline, PytorchEngineConfig
pipe = pipeline('internlm/internlm2_5-7b-chat',
                backend_config=PytorchEngineConfig(
                    max_batch_size=32,
                    enable_prefix_caching=True,
                    cache_max_entry_count=0.8,
                    session_len=8192,
                ))
```

Note

The parameter "cache_max_entry_count" significantly influences the GPU memory usage. It means the proportion of FREE GPU memory occupied by the K/V cache after the model weights are loaded.

The default value is 0.8. The K/V cache memory is allocated once and reused repeatedly, which is why it is observed that the built pipeline and the "api_server" mentioned later in the next consumes a substantial amount of GPU memory.

If you encounter an Out-of-Memory(OOM) error, you may need to consider lowering the value of "cache_max_entry_count".

When use the callable `pipe()` to perform token generation with given prompts, you can set the sampling parameters via `GenerationConfig` as below:

```
from lmdeploy import GenerationConfig, pipeline

pipe = pipeline('internlm/internlm2_5-7b-chat')
prompts = ['Hi, pls intro yourself', 'Shanghai is']
response = pipe(prompts,
                gen_config=GenerationConfig(
                    max_new_tokens=1024,
                    top_p=0.8,
                    top_k=40,
                    temperature=0.6
                ))
```

In the `GenerationConfig`, `top_k=1` or `temperature=0.0` indicates greedy search.

For more information about pipeline, please read the *detailed tutorial*

VLM inference

The usage of VLM inference pipeline is akin to that of LLMs, with the additional capability of processing image data with the pipeline. For example, you can utilize the following code snippet to perform the inference with an InternVL model:

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2-8B')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

In VLM pipeline, the default image processing batch size is 1. This can be adjusted by `VisionConfig`. For instance, you might set it like this:

```
from lmdeploy import pipeline, VisionConfig
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2-8B',
                vision_config=VisionConfig(
                    max_batch_size=8
                ))
```

(continues on next page)

(continued from previous page)

```
image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

However, the larger the image batch size, the greater risk of an OOM error, because the LLM component within the VLM model pre-allocates a massive amount of memory in advance.

We encourage you to manually choose between the TurboMind Engine and the PyTorch Engine based on their respective capabilities, as detailed in *the supported-models matrix*. Additionally, follow the instructions in *LLM Inference* section to reduce the values of memory-related parameters

1.2.2 Serving

As demonstrated in the previous *offline batch inference* section, this part presents the respective serving methods for LLMs and VLMs.

Serve a LLM model

```
lmdeploy serve api_server internlm/internlm2_5-7b-chat
```

This command will launch an OpenAI-compatible server on the localhost at port `23333`. You can specify a different server port by using the `--server-port` option. For more options, consult the help documentation by running `lmdeploy serve api_server --help`. Most of these options align with the engine configuration.

To access the service, you can utilize the official OpenAI Python package `pip install openai`. Below is an example demonstrating how to use the entrypoint `v1/chat/completions`

```
from openai import OpenAI
client = OpenAI(
    api_key='YOUR_API_KEY',
    base_url="http://0.0.0.0:23333/v1"
)
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
  model=model_name,
  messages=[
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": " provide three suggestions about time management"},
  ],
    temperature=0.8,
    top_p=0.8
)
print(response)
```

We encourage you to refer to the detailed guide for more comprehensive information about *serving with Docker*, *function calls* and other topics

Serve a VLM model

```
lmdeploy serve api_server OpenGVLab/InternVL2-8B
```

Note

LMDeploy reuses the vision component from upstream VLM repositories. Each upstream VLM model may have different dependencies. Consequently, LMDeploy has decided not to include the dependencies of the upstream VLM repositories in its own dependency list. If you encounter an "ImportError" when using LMDeploy for inference with VLM models, please install the relevant dependencies yourself.

After the service is launched successfully, you can access the VLM service in a manner similar to how you would access the `gptv4` service by modifying the `api_key` and `base_url` parameters:

```python
from openai import OpenAI

client = OpenAI(api_key='YOUR_API_KEY', base_url='http://0.0.0.0:23333/v1')
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role':
        'user',
        'content': [{
            'type': 'text',
            'text': 'Describe the image please',
        }, {
            'type': 'image_url',
            'image_url': {
                'url':
                'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/data/
↪tiger.jpeg',
            },
        }],
    }],
    temperature=0.8,
    top_p=0.8)
print(response)
```

1.2.3 Inference with Command line Interface

LMDeploy offers a very convenient CLI tool for users to chat with the LLM model locally. For example:

```
lmdeploy chat internlm/internlm2_5-7b-chat --backend turbomind
```

It is designed to assist users in checking and verifying whether LMDeploy supports their model, whether the chat template is applied correctly, and whether the inference results are delivered smoothly.

Another tool, `lmdeploy check_env`, aims to gather the essential environment information. It is crucial when reporting an issue to us, as it helps us diagnose and resolve the problem more effectively.

If you have any doubt about their usage, you can try using the `--help` option to obtain detailed information.

1.3 On Other Platforms

1.3.1 Get Started with Huawei Ascend (Atlas 800T A2)

The usage of lmdeploy on a Huawei Ascend device is almost the same as its usage on CUDA with PytorchEngine in lmdeploy. Please read the original *Get Started* guide before reading this tutorial.

Here is the *supported model list*.

Installation

We highly recommend that users build a Docker image for streamlined environment setup.

Git clone the source code of lmdeploy and the Dockerfile locates in the `docker` directory:

```
git clone https://github.com/InternLM/lmdeploy.git
cd lmdeploy
```

Environment Preparation

The Docker version is supposed to be no less than `18.09`. And `Ascend Docker Runtime` should be installed by following the official guide.

> [!CAUTION] If error message `libascend_hal.so: cannot open shared object file` shows, that means **Ascend Docker Runtime** is not installed correctly!

Ascend Drivers, Firmware and CANN

The target machine needs to install the Huawei driver and firmware version not lower than 23.0.3, refer to CANN Driver and Firmware Installation and download resources.

And the CANN (version 8.0.RC2.beta1) software packages should also be downloaded from Ascend Resource Download Center themselves. Make sure to place the `Ascend-cann-kernels-910b*.run`, `Ascend-cann-nnal_*.run` and `Ascend-cann-toolkit*-aarch64.run` under the root directory of lmdeploy source code

Build Docker Image

Run the following command in the root directory of lmdeploy to build the image:

```
DOCKER_BUILDKIT=1 docker build -t lmdeploy-aarch64-ascend:latest \
    -f docker/Dockerfile_aarch64_ascend .
```

The `Dockerfile_aarch64_ascend` is tested on Kunpeng CPU. For intel CPU, please try this dockerfile (which is not fully tested)

If the following command executes without any errors, it indicates that the environment setup is successful.

```
docker run -e ASCEND_VISIBLE_DEVICES=0 --rm --name lmdeploy -t lmdeploy-aarch64-
↪ascend:latest lmdeploy check_env
```

For more information about running the Docker client on Ascend devices, please refer to the guide

Offline batch inference

> [!TIP] Graph mode has been supported on Atlas 800T A2. Users can set `eager_mode=False` to enable graph mode, or, set `eager_mode=True` to disable graph mode. (Please source `/usr/local/Ascend/nnal/atb/set_env.sh` before enabling graph mode)

LLM inference

Set `device_type="ascend"` in the `PytorchEngineConfig`:

```python
from lmdeploy import pipeline
from lmdeploy import PytorchEngineConfig
if __name__ == "__main__":
    pipe = pipeline("internlm/internlm2_5-7b-chat",
                    backend_config=PytorchEngineConfig(tp=1, device_type="ascend", eager_
↪mode=True))
    question = ["Shanghai is", "Please introduce China", "How are you?"]
    response = pipe(question)
    print(response)
```

VLM inference

Set `device_type="ascend"` in the `PytorchEngineConfig`:

```python
from lmdeploy import pipeline, PytorchEngineConfig
from lmdeploy.vl import load_image
if __name__ == "__main__":
    pipe = pipeline('OpenGVLab/InternVL2-2B',
                    backend_config=PytorchEngineConfig(tp=1, device_type='ascend', eager_
↪mode=True))
    image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
    response = pipe(('describe this image', image))
    print(response)
```

Online serving

> [!TIP] Graph mode has been supported on Atlas 800T A2. Graph mode is default enabled in online serving. Users can add `--eager-mode` to disable graph mode. (Please source `/usr/local/Ascend/nnal/atb/set_env.sh` before enabling graph mode)

Serve a LLM model

Add `--device ascend` in the serve command.

```
lmdeploy serve api_server --backend pytorch --device ascend --eager-mode internlm/
↪internlm2_5-7b-chat
```

Serve a VLM model

Add `--device ascend` in the serve command

```
lmdeploy serve api_server --backend pytorch --device ascend --eager-mode OpenGVLab/
↪InternVL2-2B
```

Inference with Command line Interface

Add `--device ascend` in the serve command.

```
lmdeploy chat internlm/internlm2_5-7b-chat --backend pytorch --device ascend --eager-mode
```

Run the following commands to launch lmdeploy chatting after starting container:

```
docker exec -it lmdeploy_ascend_demo \
    bash -i -c "lmdeploy chat --backend pytorch --device ascend --eager-mode internlm/
↪internlm2_5-7b-chat"
```

Quantization

w4a16 AWQ

Run the following commands to quantize weights on Atlas 800T A2.

```
lmdeploy lite auto_awq $HF_MODEL --work-dir $WORK_DIR --device npu
```

Please check *supported_models* before use this feature.

int8 KV-cache Quantization

Ascend backend has supported offline int8 KV-cache Quantization on eager mode.

Please refer this doc for details.

1.4 Supported Models

The following tables detail the models supported by LMDeploy's TurboMind engine and PyTorch engine across different platforms.

1.4.1 TurboMind on CUDA Platform

Model	Size	Type	FP16/BF16	KV INT8	KV INT4	W4A16
Llama	7B - 65B	LLM	Yes	Yes	Yes	Yes
Llama2	7B - 70B	LLM	Yes	Yes	Yes	Yes
Llama3	8B, 70B	LLM	Yes	Yes	Yes	Yes
Llama3.1	8B, 70B	LLM	Yes	Yes	Yes	Yes
Llama3.2[2]	1B, 3B	LLM	Yes	Yes*	Yes*	Yes
InternLM	7B - 20B	LLM	Yes	Yes	Yes	Yes
InternLM2	7B - 20B	LLM	Yes	Yes	Yes	Yes
InternLM2.5	7B	LLM	Yes	Yes	Yes	Yes
InternLM-XComposer2	7B, 4khd-7B	MLLM	Yes	Yes	Yes	Yes
InternLM-XComposer2.5	7B	MLLM	Yes	Yes	Yes	Yes
Qwen	1.8B - 72B	LLM	Yes	Yes	Yes	Yes
Qwen1.5[1]	1.8B - 110B	LLM	Yes	Yes	Yes	Yes
Qwen2[2]	0.5B - 72B	LLM	Yes	Yes*	Yes*	Yes
Qwen2-MoE	57BA14B	LLM	Yes	Yes	Yes	Yes
Qwen2.5[2]	0.5B - 72B	LLM	Yes	Yes*	Yes*	Yes
Mistral[1]	7B	LLM	Yes	Yes	Yes	No
Mixtral	8x7B, 8x22B	LLM	Yes	Yes	Yes	Yes
DeepSeek-V2	16B, 236B	LLM	Yes	Yes	Yes	No
DeepSeek-V2.5	236B	LLM	Yes	Yes	Yes	No
Qwen-VL	7B	MLLM	Yes	Yes	Yes	Yes
DeepSeek-VL	7B	MLLM	Yes	Yes	Yes	Yes
Baichuan	7B	LLM	Yes	Yes	Yes	Yes
Baichuan2	7B	LLM	Yes	Yes	Yes	Yes
Code Llama	7B - 34B	LLM	Yes	Yes	Yes	No
YI	6B - 34B	LLM	Yes	Yes	Yes	Yes
LLaVA(1.5,1.6)	7B - 34B	MLLM	Yes	Yes	Yes	Yes
InternVL	v1.1 - v1.5	MLLM	Yes	Yes	Yes	Yes
InternVL2[2]	1 - 2B, 8B - 76B	MLLM	Yes	Yes*	Yes*	Yes
InternVL2.5(MPO)[2]	1 - 78B	MLLM	Yes	Yes*	Yes*	Yes
ChemVLM	8B - 26B	MLLM	Yes	Yes	Yes	Yes
MiniCPM-Llama3-V-2_5	-	MLLM	Yes	Yes	Yes	Yes
MiniCPM-V-2_6	-	MLLM	Yes	Yes	Yes	Yes
MiniGeminiLlama	7B	MLLM	Yes	-	-	Yes
GLM4	9B	LLM	Yes	Yes	Yes	Yes
CodeGeeX4	9B	LLM	Yes	Yes	Yes	-
Molmo	7B-D,72B	MLLM	Yes	Yes	Yes	No

"-" means not verified yet.

Note

- [1] The TurboMind engine doesn't support window attention. Therefore, for models that have applied window attention and have the corresponding switch "use_sliding_window" enabled, such as Mistral, Qwen1.5 and etc., please choose the PyTorch engine for inference.
- [2] When the head_dim of a model is not 128, such as llama3.2-1B, qwen2-0.5B and internvl2-1B, turbomind doesn't support its kv cache 4/8 bit quantization and inference

1.4.2 PyTorchEngine on CUDA Platform

Model	Size	Type	FP16/BF16	KV INT8	KV INT4	W8A8	W4A16
Llama	7B - 65B	LLM	Yes	Yes	Yes	Yes	Yes
Llama2	7B - 70B	LLM	Yes	Yes	Yes	Yes	Yes
Llama3	8B, 70B	LLM	Yes	Yes	Yes	Yes	Yes
Llama3.1	8B, 70B	LLM	Yes	Yes	Yes	Yes	Yes
Llama3.2	1B, 3B	LLM	Yes	Yes	Yes	Yes	Yes
Llama3.2-VL	11B, 90B	MLLM	Yes	Yes	Yes	-	-
InternLM	7B - 20B	LLM	Yes	Yes	Yes	Yes	Yes
InternLM2	7B - 20B	LLM	Yes	Yes	Yes	Yes	Yes
InternLM2.5	7B	LLM	Yes	Yes	Yes	Yes	Yes
Baichuan2	7B	LLM	Yes	Yes	Yes	Yes	No
Baichuan2	13B	LLM	Yes	Yes	Yes	No	No
ChatGLM2	6B	LLM	Yes	Yes	Yes	No	No
Falcon	7B - 180B	LLM	Yes	Yes	Yes	No	No
YI	6B - 34B	LLM	Yes	Yes	Yes	Yes	Yes
Mistral	7B	LLM	Yes	Yes	Yes	Yes	Yes
Mixtral	8x7B, 8x22B	LLM	Yes	Yes	Yes	No	No
QWen	1.8B - 72B	LLM	Yes	Yes	Yes	Yes	Yes
QWen1.5	0.5B - 110B	LLM	Yes	Yes	Yes	Yes	Yes
QWen1.5-MoE	A2.7B	LLM	Yes	Yes	Yes	No	No
QWen2	0.5B - 72B	LLM	Yes	Yes	No	Yes	Yes
Qwen2.5	0.5B - 72B	LLM	Yes	Yes	No	Yes	Yes
QWen2-VL	2B, 7B	MLLM	Yes	Yes	No	No	Yes
DeepSeek-MoE	16B	LLM	Yes	No	No	No	No
DeepSeek-V2	16B, 236B	LLM	Yes	No	No	No	No
DeepSeek-V2.5	236B	LLM	Yes	No	No	No	No
MiniCPM3	4B	LLM	Yes	Yes	Yes	No	No
MiniCPM-V-2_6	8B	LLM	Yes	No	No	No	Yes
Gemma	2B-7B	LLM	Yes	Yes	Yes	No	No
Dbrx	132B	LLM	Yes	Yes	Yes	No	No
StarCoder2	3B-15B	LLM	Yes	Yes	Yes	No	No
Phi-3-mini	3.8B	LLM	Yes	Yes	Yes	Yes	Yes
Phi-3-vision	4.2B	MLLM	Yes	Yes	Yes	-	-
CogVLM-Chat	17B	MLLM	Yes	Yes	Yes	-	-
CogVLM2-Chat	19B	MLLM	Yes	Yes	Yes	-	-
LLaVA(1.5,1.6)[2]	7B-34B	MLLM	No	No	No	No	No
InternVL(v1.5)	2B-26B	MLLM	Yes	Yes	Yes	No	Yes
InternVL2	1B-76B	MLLM	Yes	Yes	Yes	-	-
InternVL2.5(MPO)	1B-78B	MLLM	Yes	Yes	Yes	-	-
Mono-InternVL[1]	2B	MLLM	Yes	Yes	Yes	-	-
ChemVLM	8B-26B	MLLM	Yes	Yes	No	-	-

continues on next page

Table 2 – continued from previous page

Model	Size	Type	FP16/BF16	KV INT8	KV INT4	W8A8	W4A16
Gemma2	9B-27B	LLM	Yes	Yes	Yes	-	-
GLM4	9B	LLM	Yes	Yes	Yes	No	No
GLM-4V	9B	MLLM	Yes	Yes	Yes	No	Yes
CodeGeeX4	9B	LLM	Yes	Yes	Yes	-	-
Phi-3.5-mini	3.8B	LLM	Yes	Yes	No	-	-
Phi-3.5-MoE	16x3.8B	LLM	Yes	Yes	No	-	-
Phi-3.5-vision	4.2B	MLLM	Yes	Yes	No	-	-

Note

- [1] Currently Mono-InternVL does not support FP16 due to numerical instability. Please use BF16 instead.
- [2] PyTorch engine removes the support of original llava models after v0.6.4. Please use their corresponding transformers models instead, which can be found in https://huggingface.co/llava-hf

1.4.3 PyTorchEngine on Huawei Ascend Platform

Model	Size	Type	FP16/BF16(eager)	FP16/BF16(graph)	W4A16(eager)
Llama2	7B - 70B	LLM	Yes	Yes	Yes
Llama3	8B	LLM	Yes	Yes	Yes
Llama3.1	8B	LLM	Yes	Yes	Yes
InternLM2	7B - 20B	LLM	Yes	Yes	Yes
InternLM2.5	7B - 20B	LLM	Yes	Yes	Yes
Mixtral	8x7B	LLM	Yes	Yes	No
QWen1.5-MoE	A2.7B	LLM	Yes	-	No
QWen2(.5)	7B	LLM	Yes	Yes	No
QWen2-MoE	A14.57B	LLM	Yes	-	No
InternVL(v1.5)	2B-26B	MLLM	Yes	-	Yes
InternVL2	1B-40B	MLLM	Yes	Yes	Yes
CogVLM2-chat	19B	MLLM	Yes	No	-
GLM4V	9B	MLLM	Yes	No	-

1.5 Offline Inference Pipeline

In this tutorial, We will present a list of examples to introduce the usage of `lmdeploy.pipeline`.

You can overview the detailed pipeline API in this guide.

1.5.1 Usage

A 'Hello, world' example

```python
from lmdeploy import pipeline

pipe = pipeline('internlm/internlm2_5-7b-chat')
response = pipe(['Hi, pls intro yourself', 'Shanghai is'])
print(response)
```

In this example, the pipeline by default allocates a predetermined percentage of GPU memory for storing k/v cache. The ratio is dictated by the parameter `TurbomindEngineConfig.cache_max_entry_count`.

There have been alterations to the strategy for setting the k/v cache ratio throughout the evolution of LMDeploy. The following are the change histories:

1. `v0.2.0 <= lmdeploy <= v0.2.1`

   `TurbomindEngineConfig.cache_max_entry_count` defaults to 0.5, indicating 50% GPU **total memory** allocated for k/v cache. Out Of Memory (OOM) errors may occur if a 7B model is deployed on a GPU with memory less than 40G. If you encounter an OOM error, please decrease the ratio of the k/v cache occupation as follows:

   ```python
   from lmdeploy import pipeline, TurbomindEngineConfig

   # decrease the ratio of the k/v cache occupation to 20%
   backend_config = TurbomindEngineConfig(cache_max_entry_count=0.2)

   pipe = pipeline('internlm/internlm2_5-7b-chat',
                   backend_config=backend_config)
   response = pipe(['Hi, pls intro yourself', 'Shanghai is'])
   print(response)
   ```

2. `lmdeploy > v0.2.1`

   The allocation strategy for k/v cache is changed to reserve space from the **GPU free memory** proportionally. The ratio `TurbomindEngineConfig.cache_max_entry_count` has been adjusted to 0.8 by default. If OOM error happens, similar to the method mentioned above, please consider reducing the ratio value to decrease the memory usage of the k/v cache.

Set tensor parallelism

```python
from lmdeploy import pipeline, TurbomindEngineConfig

backend_config = TurbomindEngineConfig(tp=2)
pipe = pipeline('internlm/internlm2_5-7b-chat',
                backend_config=backend_config)
response = pipe(['Hi, pls intro yourself', 'Shanghai is'])
print(response)
```

Set sampling parameters

```python
from lmdeploy import pipeline, GenerationConfig, TurbomindEngineConfig

backend_config = TurbomindEngineConfig(tp=2)
gen_config = GenerationConfig(top_p=0.8,
                              top_k=40,
                              temperature=0.8,
                              max_new_tokens=1024)
pipe = pipeline('internlm/internlm2_5-7b-chat',
                backend_config=backend_config)
response = pipe(['Hi, pls intro yourself', 'Shanghai is'],
                gen_config=gen_config)
print(response)
```

Apply OpenAI format prompt

```python
from lmdeploy import pipeline, GenerationConfig, TurbomindEngineConfig

backend_config = TurbomindEngineConfig(tp=2)
gen_config = GenerationConfig(top_p=0.8,
                              top_k=40,
                              temperature=0.8,
                              max_new_tokens=1024)
pipe = pipeline('internlm/internlm2_5-7b-chat',
                backend_config=backend_config)
prompts = [[{
    'role': 'user',
    'content': 'Hi, pls intro yourself'
}], [{
    'role': 'user',
    'content': 'Shanghai is'
}]]
response = pipe(prompts,
                gen_config=gen_config)
print(response)
```

Apply streaming output

```python
from lmdeploy import pipeline, GenerationConfig, TurbomindEngineConfig

backend_config = TurbomindEngineConfig(tp=2)
gen_config = GenerationConfig(top_p=0.8,
                              top_k=40,
                              temperature=0.8,
                              max_new_tokens=1024)
pipe = pipeline('internlm/internlm2_5-7b-chat',
                backend_config=backend_config)
prompts = [[{
    'role': 'user',
```

(continues on next page)

(continued from previous page)

```
    'content': 'Hi, pls intro yourself'
}], [{
    'role': 'user',
    'content': 'Shanghai is'
}]]
for item in pipe.stream_infer(prompts, gen_config=gen_config):
    print(item)
```

Get logits for generated tokens

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('internlm/internlm2_5-7b-chat')

gen_config=GenerationConfig(output_logits='generation'
                            max_new_tokens=10)
response = pipe(['Hi, pls intro yourself', 'Shanghai is'],
                gen_config=gen_config)
logits = [x.logits for x in response]
```

Get last layer's hidden states for generated tokens

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('internlm/internlm2_5-7b-chat')

gen_config=GenerationConfig(output_last_hidden_state='generation',
                            max_new_tokens=10)
response = pipe(['Hi, pls intro yourself', 'Shanghai is'],
                gen_config=gen_config)
hidden_states = [x.last_hidden_state for x in response]
```

Calculate ppl

```
from transformers import AutoTokenizer
from lmdeploy import pipeline


model_repoid_or_path = 'internlm/internlm2_5-7b-chat'
pipe = pipeline(model_repoid_or_path)
tokenizer = AutoTokenizer.from_pretrained(model_repoid_or_path, trust_remote_code=True)
messages = [
   {"role": "user", "content": "Hello, how are you?"},
]
input_ids = tokenizer.apply_chat_template(messages)

# ppl is a list of float numbers
```

(continues on next page)

(continued from previous page)

```
ppl = pipe.get_ppl(input_ids)
print(ppl)
```

Note

- When input_ids is too long, an OOM (Out Of Memory) error may occur. Please apply it with caution
- get_ppl returns the cross entropy loss without applying the exponential operation afterwards

Use PyTorchEngine

```
pip install triton>=2.1.0
```

```
from lmdeploy import pipeline, GenerationConfig, PytorchEngineConfig

backend_config = PytorchEngineConfig(session_len=2048)
gen_config = GenerationConfig(top_p=0.8,
                              top_k=40,
                              temperature=0.8,
                              max_new_tokens=1024)
pipe = pipeline('internlm/internlm2_5-7b-chat',
                backend_config=backend_config)
prompts = [[{
    'role': 'user',
    'content': 'Hi, pls intro yourself'
}], [{
    'role': 'user',
    'content': 'Shanghai is'
}]]
response = pipe(prompts, gen_config=gen_config)
print(response)
```

Inference with LoRA

```
from lmdeploy import pipeline, GenerationConfig, PytorchEngineConfig

backend_config = PytorchEngineConfig(session_len=2048,
                                     adapters=dict(lora_name_1='chenchi/lora-chatglm2-6b-
↪guodegang'))
gen_config = GenerationConfig(top_p=0.8,
                              top_k=40,
                              temperature=0.8,
                              max_new_tokens=1024)
pipe = pipeline('THUDM/chatglm2-6b',
                backend_config=backend_config)
prompts = [[{
    'role': 'user',
    'content': ''
```

(continues on next page)

(continued from previous page)

```
}]]
response = pipe(prompts, gen_config=gen_config, adapter_name='lora_name_1')
print(response)
```

1.5.2 FAQs

- **RuntimeError: An attempt has been made to start a new process before the current process has finished its bootstrapping phase**.

  If you got this for tp>1 in pytorch backend. Please make sure the python script has following

  ```
  if __name__ == '__main__':
  ```

  Generally, in the context of multi-threading or multi-processing, it might be necessary to ensure that initialization code is executed only once. In this case, `if __name__ == '__main__':` can help to ensure that these initialization codes are run only in the main program, and not repeated in each newly created process or thread.

- To customize a chat template, please refer to *chat_template.md*.
- If the weight of lora has a corresponding chat template, you can first register the chat template to lmdeploy, and then use the chat template name as the adapter name.

1.6 OpenAI Compatible Server

This article primarily discusses the deployment of a single LLM model across multiple GPUs on a single node, providing a service that is compatible with the OpenAI interface, as well as the usage of the service API. For the sake of convenience, we refer to this service as `api_server`. Regarding parallel services with multiple models, please refer to the guide about *Request Distribution Server*.

In the following sections, we will first introduce methods for starting the service, choosing the appropriate one based on your application scenario.

Next, we focus on the definition of the service's RESTful API, explore the various ways to interact with the interface, and demonstrate how to try the service through the Swagger UI or LMDeploy CLI tools.

Finally, we showcase how to integrate the service into a WebUI, providing you with a reference to easily set up a demonstration demo.

1.6.1 Launch Service

Take the internlm2_5-7b-chat model hosted on huggingface hub as an example, you can choose one the following methods to start the service.

Option 1: Launching with lmdeploy CLI

```
lmdeploy serve api_server internlm/internlm2_5-7b-chat --server-port 23333
```

The arguments of `api_server` can be viewed through the command `lmdeploy serve api_server -h`, for instance, `--tp` to set tensor parallelism, `--session-len` to specify the max length of the context window, `--cache-max-entry-count` to adjust the GPU mem ratio for k/v cache etc.

Option 2: Deploying with docker

With LMDeploy official docker image, you can run OpenAI compatible server as follows:

```
docker run --runtime nvidia --gpus all \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    --env "HUGGING_FACE_HUB_TOKEN=<secret>" \
    -p 23333:23333 \
    --ipc=host \
    openmmlab/lmdeploy:latest \
    lmdeploy serve api_server internlm/internlm2_5-7b-chat
```

The parameters of `api_server` are the same with that mentioned in "*option 1*" section

Option 3: Deploying to Kubernetes cluster

Connect to a running Kubernetes cluster and deploy the internlm2_5-7b-chat model service with kubectl command-line tool (replace `<your token>` with your huggingface hub token):

```
sed 's/{{HUGGING_FACE_HUB_TOKEN}}/<your token>/' k8s/deployment.yaml | kubectl create -f␣
↪- \
    && kubectl create -f k8s/service.yaml
```

In the example above the model data is placed on the local disk of the node (hostPath). Consider replacing it with high-availability shared storage if multiple replicas are desired, and the storage can be mounted into container using PersistentVolume.

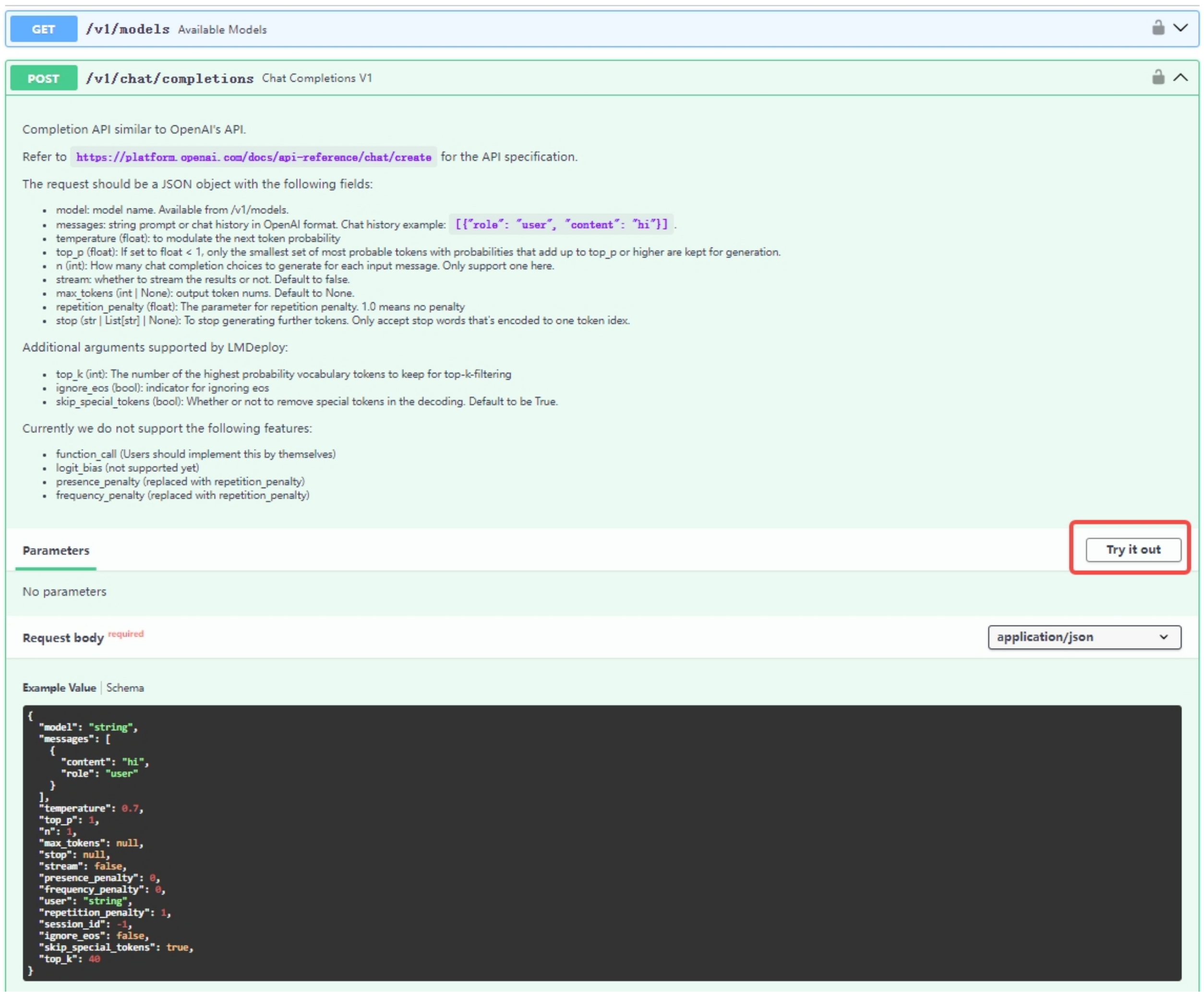
1.6.2 RESTful API

LMDeploy's RESTful API is compatible with the following three OpenAI interfaces:

- /v1/chat/completions
- /v1/models
- /v1/completions

Additionally, LMDeploy also defines `/v1/chat/interactive` to support interactive inference. The feature of interactive inference is that there's no need to pass the user conversation history as required by `v1/chat/completions`, since the conversation history will be cached on the server side. This method boasts excellent performance during multi-turn long context inference.

You can overview and try out the offered RESTful APIs by the website `http://0.0.0.0:23333` as shown in the below image after launching the service successfully.



/v1/models Available Models

/v1/chat/completions Chat Completions V1

Completion API similar to OpenAI's API.

Refer to `https://platform.openai.com/docs/api-reference/chat/create` for the API specification.

The request should be a JSON object with the following fields:

- model: model name. Available from /v1/models.
- messages: string prompt or chat history in OpenAI format. Chat history example: `[{"role": "user", "content": "hi"}]`.
- temperature (float): to modulate the next token probability
- top_p (float): If set to float < 1, only the smallest set of most probable tokens with probabilities that add up to top_p or higher are kept for generation.
- n (int): How many chat completion choices to generate for each input message. Only support one here.
- stream: whether to stream the results or not. Default to false.
- max_tokens (int | None): output token nums. Default to None.
- repetition_penalty (float): The parameter for repetition penalty. 1.0 means no penalty
- stop (str | List[str] | None): To stop generating further tokens. Only accept stop words that's encoded to one token idex.

Additional arguments supported by LMDeploy:

- top_k (int): The number of the highest probability vocabulary tokens to keep for top-k-filtering
- ignore_eos (bool): indicator for ignoring eos
- skip_special_tokens (bool): Whether or not to remove special tokens in the decoding. Default to be True.

Currently we do not support the following features:

- function_call (Users should implement this by themselves)
- logit_bias (not supported yet)
- presence_penalty (replaced with repetition_penalty)
- frequency_penalty (replaced with repetition_penalty)

Parameters

No parameters

Request body required

Example Value | Schema

```
{
  "model": "string",
  "messages": [
    {
      "content": "hi",
      "role": "user"
    }
  ],
  "temperature": 0.7,
  "top_p": 1,
  "n": 1,
  "max_tokens": null,
  "stop": null,
  "stream": false,
  "presence_penalty": 0,
  "frequency_penalty": 0,
  "user": "string",
  "repetition_penalty": 1,
  "session_id": -1,
  "ignore_eos": false,
  "skip_special_tokens": true,
  "top_k": 40
}
```

Or, you can use the LMDeploy's built-in CLI tool to verify the service correctness right from the console.

```
# restful_api_url is what printed in api_server.py, e.g. http://localhost:23333
lmdeploy serve api_client ${api_server_url}
```

If you need to integrate the service into your own projects or products, we recommend the following approach:

Integrate with OpenAI

Here is an example of interaction with the endpoint `v1/chat/completions` service via the openai package. Before running it, please install the openai package by `pip install openai`

```
from openai import OpenAI
client = OpenAI(
    api_key='YOUR_API_KEY',
    base_url="http://0.0.0.0:23333/v1"
)
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
  model=model_name,
```

(continues on next page)

(continued from previous page)

```
  messages=[
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": " provide three suggestions about time management"},
  ],
    temperature=0.8,
    top_p=0.8
)
print(response)
```

If you want to use async functions, may try the following example:

```python
import asyncio
from openai import AsyncOpenAI

async def main():
    client = AsyncOpenAI(api_key='YOUR_API_KEY',
                         base_url='http://0.0.0.0:23333/v1')
    model_cards = await client.models.list()._get_page()
    response = await client.chat.completions.create(
        model=model_cards.data[0].id,
        messages=[
            {
                'role': 'system',
                'content': 'You are a helpful assistant.'
            },
            {
                'role': 'user',
                'content': ' provide three suggestions about time management'
            },
        ],
        temperature=0.8,
        top_p=0.8)
    print(response)

asyncio.run(main())
```

You can invoke other OpenAI interfaces using similar methods. For more detailed information, please refer to the OpenAI API guide

Integrate with lmdeploy `APIClient`

Below are some examples demonstrating how to visit the service through `APIClient`

If you want to use the `/v1/chat/completions` endpoint, you can try the following code:

```python
from lmdeploy.serve.openai.api_client import APIClient
api_client = APIClient(f'http://{server_ip}:{server_port}')
model_name = api_client.available_models[0]
messages = [{"role": "user", "content": "Say this is a test!"}]
for item in api_client.chat_completions_v1(model=model_name, messages=messages):
    print(item)
```

For the `/v1/completions` endpoint, you can try:

```
from lmdeploy.serve.openai.api_client import APIClient
api_client = APIClient('http://{server_ip}:{server_port}')
model_name = api_client.available_models[0]
for item in api_client.completions_v1(model=model_name, prompt='hi'):
    print(item)
```

As for `/v1/chat/interactive`we disable the feature by default. Please open it by setting `interactive_mode = True`. If you don't, it falls back to openai compatible interfaces.

Keep in mind that `session_id` indicates an identical sequence and all requests belonging to the same sequence must share the same `session_id`. For instance, in a sequence with 10 rounds of chatting requests, the `session_id` in each request should be the same.

```
from lmdeploy.serve.openai.api_client import APIClient
api_client = APIClient(f'http://{server_ip}:{server_port}')
messages = [
    "hi, what's your name?",
    "who developed you?",
    "Tell me more about your developers",
    "Summarize the information we've talked so far"
]
for message in messages:
    for item in api_client.chat_interactive_v1(prompt=message,
                                               session_id=1,
                                               interactive_mode=True,
                                               stream=False):
        print(item)
```

Tools

May refer to *api_server_tools*.

Integrate with Java/Golang/Rust

May use openapi-generator-cli to convert `http://{server_ip}:{server_port}/openapi.json` to java/rust/golang client. Here is an example:

```
$ docker run -it --rm -v ${PWD}:/local openapitools/openapi-generator-cli generate -i /
↪local/openapi.json -g rust -o /local/rust

$ ls rust/*
rust/Cargo.toml  rust/git_push.sh  rust/README.md

rust/docs:
ChatCompletionRequest.md  EmbeddingsRequest.md  HttpValidationError.md  LocationInner.md␣
↪ Prompt.md
DefaultApi.md             GenerateRequest.md    Input.md                Messages.md     ␣
↪ ValidationError.md

rust/src:
apis  lib.rs  models
```

Integrate with cURL

cURL is a tool for observing the output of the RESTful APIs.

- list served models `v1/models`

```
curl http://{server_ip}:{server_port}/v1/models
```

- chat `v1/chat/completions`

```
curl http://{server_ip}:{server_port}/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "internlm-chat-7b",
    "messages": [{"role": "user", "content": "Hello! How are you?"}]
  }'
```

- text completions `v1/completions`

```
curl http://{server_ip}:{server_port}/v1/completions \
  -H 'Content-Type: application/json' \
  -d '{
  "model": "llama",
  "prompt": "two steps to build a house:"
}'
```

- interactive chat `v1/chat/interactive`

```
curl http://{server_ip}:{server_port}/v1/chat/interactive \
  -H "Content-Type: application/json" \
  -d '{
    "prompt": "Hello! How are you?",
    "session_id": 1,
    "interactive_mode": true
  }'
```

1.6.3 Integrate with WebUI

```
# api_server_url is what printed in api_server.py, e.g. http://localhost:23333
# server_ip and server_port here are for gradio ui
# example: lmdeploy serve gradio http://localhost:23333 --server-name localhost --server-
↪port 6006
lmdeploy serve gradio api_server_url --server-name ${gradio_ui_ip} --server-port $
↪{gradio_ui_port}
```

1.6.4 Launch multiple api servers

Following are two steps to launch multiple api servers through torchrun. Just create a python script with the following codes.

1. Launch the proxy server through `lmdeploy serve proxy`. Get the correct proxy server url.
2. Launch the script through `torchrun --nproc_per_node 2 script.py InternLM/internlm2-chat-1_8b --proxy_url http://{proxy_node_name}:{proxy_node_port}`.**Note**: Please do not use `0.0.0.0:8000` here, instead, we input the real ip name, `11.25.34.55:8000` for example.

```python
import os
import socket
from typing import List, Literal

import fire


def get_host_ip():
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.connect(('8.8.8.8', 80))
        ip = s.getsockname()[0]
    finally:
        s.close()
    return ip


def main(model_path: str,
         tp: int = 1,
         proxy_url: str = 'http://0.0.0.0:8000',
         port: int = 23333,
         backend: Literal['turbomind', 'pytorch'] = 'turbomind'):
    local_rank = int(os.environ.get('LOCAL_RANK', -1))
    world_size = int(os.environ.get('WORLD_SIZE', -1))
    local_ip = get_host_ip()
    if isinstance(port, List):
        assert len(port) == world_size
        port = port[local_rank]
    else:
        port += local_rank * 10
    if (world_size - local_rank) % tp == 0:
        rank_list = ','.join([str(local_rank + i) for i in range(tp)])
        command = f'CUDA_VISIBLE_DEVICES={rank_list} lmdeploy serve api_server {model_
↪path} '\
                  f'--server-name {local_ip} --server-port {port} --tp {tp} '\
                  f'--proxy-url {proxy_url} --backend {backend}'
        print(f'running command: {command}')
        os.system(command)


if __name__ == '__main__':
    fire.Fire(main)
```

1.6.5 FAQ

1. When user got `"finish_reason":"length"`, it means the session is too long to be continued. The session length can be modified by passing `--session_len` to api_server.
2. When OOM appeared at the server side, please reduce the `cache_max_entry_count` of `backend_config` when launching the service.
3. When the request with the same `session_id` to `/v1/chat/interactive` got a empty return value and a negative `tokens`, please consider setting `interactive_mode=false` to restart the session.
4. The `/v1/chat/interactive` api disables engaging in multiple rounds of conversation by default. The input argument `prompt` consists of either single strings or entire chat histories.
5. Regarding the stop words, we only support characters that encode into a single index. Furthermore, there may be multiple indexes that decode into results containing the stop word. In such cases, if the number of these indexes is too large, we will only use the index encoded by the tokenizer. If you want use a stop symbol that encodes into multiple indexes, you may consider performing string matching on the streaming client side. Once a successful match is found, you can then break out of the streaming loop.
6. To customize a chat template, please refer to *chat_template.md*.

1.7 Tools Calling

LMDeploy supports tools for InternLM2, InternLM2.5, llama3.1 and Qwen2.5 models.

1.7.1 Single Round Invocation

Please start the service of models before running the following example.

```
from openai import OpenAI

tools = [
  {
    "type": "function",
    "function": {
      "name": "get_current_weather",
      "description": "Get the current weather in a given location",
      "parameters": {
        "type": "object",
        "properties": {
          "location": {
            "type": "string",
            "description": "The city and state, e.g. San Francisco, CA",
          },
          "unit": {"type": "string", "enum": ["celsius", "fahrenheit"]},
        },
        "required": ["location"],
      },
    }
  }
]
messages = [{"role": "user", "content": "What's the weather like in Boston today?"}]
```

(continues on next page)

(continued from previous page)

```
client = OpenAI(api_key='YOUR_API_KEY',base_url='http://0.0.0.0:23333/v1')
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
    model=model_name,
    messages=messages,
    temperature=0.8,
    top_p=0.8,
    stream=False,
    tools=tools)
print(response)
```

1.7.2 Multiple Round Invocation

InternLM

A complete toolchain invocation process can be demonstrated through the following example.

```
from openai import OpenAI


def add(a: int, b: int):
    return a + b


def mul(a: int, b: int):
    return a * b


tools = [{
    'type': 'function',
    'function': {
        'name': 'add',
        'description': 'Compute the sum of two numbers',
        'parameters': {
            'type': 'object',
            'properties': {
                'a': {
                    'type': 'int',
                    'description': 'A number',
                },
                'b': {
                    'type': 'int',
                    'description': 'A number',
                },
            },
            'required': ['a', 'b'],
        },
    }
}, {
    'type': 'function',
```

(continues on next page)

(continued from previous page)

```
    'function': {
        'name': 'mul',
        'description': 'Calculate the product of two numbers',
        'parameters': {
            'type': 'object',
            'properties': {
                'a': {
                    'type': 'int',
                    'description': 'A number',
                },
                'b': {
                    'type': 'int',
                    'description': 'A number',
                },
            },
            'required': ['a', 'b'],
        },
    }
}]
messages = [{'role': 'user', 'content': 'Compute (3+5)*2'}]

client = OpenAI(api_key='YOUR_API_KEY', base_url='http://0.0.0.0:23333/v1')
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
    model=model_name,
    messages=messages,
    temperature=0.8,
    top_p=0.8,
    stream=False,
    tools=tools)
print(response)
func1_name = response.choices[0].message.tool_calls[0].function.name
func1_args = response.choices[0].message.tool_calls[0].function.arguments
func1_out = eval(f'{func1_name}(**{func1_args})')
print(func1_out)

messages.append(response.choices[0].message)
messages.append({
    'role': 'tool',
    'content': f'3+5={func1_out}',
    'tool_call_id': response.choices[0].message.tool_calls[0].id
})
response = client.chat.completions.create(
    model=model_name,
    messages=messages,
    temperature=0.8,
    top_p=0.8,
    stream=False,
    tools=tools)
print(response)
func2_name = response.choices[0].message.tool_calls[0].function.name
func2_args = response.choices[0].message.tool_calls[0].function.arguments
```

(continues on next page)

(continued from previous page)

```
func2_out = eval(f'{func2_name}(**{func2_args})')
print(func2_out)
```

Using the InternLM2-Chat-7B model to execute the above example, the following results will be printed.

```
ChatCompletion(id='1', choices=[Choice(finish_reason='tool_calls', index=0,␣
↪logprobs=None, message=ChatCompletionMessage(content='', role='assistant', function_
↪call=None, tool_calls=[ChatCompletionMessageToolCall(id='0',␣
↪function=Function(arguments='{"a": 3, "b": 5}', name='add'), type='function')]))],␣
↪created=1722852901, model='/nvme/shared_data/InternLM/internlm2-chat-7b', object='chat.
↪completion', system_fingerprint=None, usage=CompletionUsage(completion_tokens=25,␣
↪prompt_tokens=263, total_tokens=288))
8
ChatCompletion(id='2', choices=[Choice(finish_reason='tool_calls', index=0,␣
↪logprobs=None, message=ChatCompletionMessage(content='', role='assistant', function_
↪call=None, tool_calls=[ChatCompletionMessageToolCall(id='1',␣
↪function=Function(arguments='{"a": 8, "b": 2}', name='mul'), type='function')]))],␣
↪created=1722852901, model='/nvme/shared_data/InternLM/internlm2-chat-7b', object='chat.
↪completion', system_fingerprint=None, usage=CompletionUsage(completion_tokens=25,␣
↪prompt_tokens=293, total_tokens=318))
16
```

Llama 3.1

Meta announces in Llama3's official user guide that,

Note

There are three built-in tools (brave_search, wolfram_alpha, and code interpreter) can be turned on using the system prompt:

1. Brave Search: Tool call to perform web searches.
2. Wolfram Alpha: Tool call to perform complex mathematical calculations.
3. Code Interpreter: Enables the model to output python code.

Additionally, it cautions: "**Note:** We recommend using Llama 70B-instruct or Llama 405B-instruct for applications that combine conversation and tool calling. Llama 8B-Instruct can not reliably maintain a conversation alongside tool calling definitions. It can be used for zero-shot tool calling, but tool instructions should be removed for regular conversations between the model and the user."

Therefore, we utilize Meta-Llama-3.1-70B-Instruct to show how to invoke the tool calling by LMDeploy `api_server`.

On a A100-SXM-80G node, you can start the service as follows:

```
lmdeploy serve api_server /the/path/of/Meta-Llama-3.1-70B-Instruct/model --tp 4
```

For an in-depth understanding of the api_server, please refer to the detailed documentation available *here*.

The following code snippet demonstrates how to utilize the 'Wolfram Alpha' tool. It is assumed that you have already registered on the Wolfram Alpha website and obtained an API key. Please ensure that you have a valid API key to access the services provided by Wolfram Alpha

```python
from openai import OpenAI
import requests


def request_llama3_1_service(messages):
    client = OpenAI(api_key='YOUR_API_KEY',
                    base_url='http://0.0.0.0:23333/v1')
    model_name = client.models.list().data[0].id
    response = client.chat.completions.create(
        model=model_name,
        messages=messages,
        temperature=0.8,
        top_p=0.8,
        stream=False)
    return response.choices[0].message.content


# The role of "system" MUST be specified, including the required tools
messages = [
    {
        "role": "system",
        "content": "Environment: ipython\nTools: wolfram_alpha\n\n Cutting Knowledge␣
↪Date: December 2023\nToday Date: 23 Jul 2024\n\nYou are a helpful Assistant." # noqa
    },
    {
        "role": "user",
        "content": "Can you help me solve this equation: x^3 - 4x^2 + 6x - 24 = 0"  #␣
↪noqa
    }
]

# send request to the api_server of llama3.1-70b and get the response
# the "assistant_response" is supposed to be:
# <|python_tag|>wolfram_alpha.call(query="solve x^3 - 4x^2 + 6x - 24 = 0")
assistant_response = request_llama3_1_service(messages)
print(assistant_response)

# Call the API of Wolfram Alpha with the query generated by the model
app_id = 'YOUR-Wolfram-Alpha-API-KEY'
params = {
    "input": assistant_response,
    "appid": app_id,
    "format": "plaintext",
    "output": "json",
}

wolframalpha_response = requests.get(
    "https://api.wolframalpha.com/v2/query",
    params=params
)
wolframalpha_response = wolframalpha_response.json()

# Append the contents obtained by the model and the wolframalpha's API
```

(continues on next page)

(continued from previous page)

```python
# to "messages", and send it again to the api_server
messages += [
    {
        "role": "assistant",
        "content": assistant_response
    },
    {
        "role": "ipython",
        "content": wolframalpha_response
    }
]

assistant_response = request_llama3_1_service(messages)
print(assistant_response)
```

Qwen2.5

Qwen2.5 supports multi tool calling, which means that multiple tool requests can be initiated in one request

```python
from openai import OpenAI
import json

def get_current_temperature(location: str, unit: str = "celsius"):
    """Get current temperature at a location.

    Args:
        location: The location to get the temperature for, in the format "City, State,␣
↪Country".
        unit: The unit to return the temperature in. Defaults to "celsius". (choices: [
↪"celsius", "fahrenheit"])

    Returns:
        the temperature, the location, and the unit in a dict
    """
    return {
        "temperature": 26.1,
        "location": location,
        "unit": unit,
    }


def get_temperature_date(location: str, date: str, unit: str = "celsius"):
    """Get temperature at a location and date.

    Args:
        location: The location to get the temperature for, in the format "City, State,␣
↪Country".
        date: The date to get the temperature for, in the format "Year-Month-Day".
        unit: The unit to return the temperature in. Defaults to "celsius". (choices: [
↪"celsius", "fahrenheit"])

```

(continues on next page)

(continued from previous page)

```
    Returns:
        the temperature, the location, the date and the unit in a dict
    """
    return {
        "temperature": 25.9,
        "location": location,
        "date": date,
        "unit": unit,
    }

def get_function_by_name(name):
    if name == "get_current_temperature":
        return get_current_temperature
    if name == "get_temperature_date":
        return get_temperature_date

tools = [{
    'type': 'function',
    'function': {
        'name': 'get_current_temperature',
        'description': 'Get current temperature at a location.',
        'parameters': {
            'type': 'object',
            'properties': {
                'location': {
                    'type': 'string',
                    'description': 'The location to get the temperature for, in the␣
↪format \'City, State, Country\'.'
                },
                'unit': {
                    'type': 'string',
                    'enum': [
                        'celsius',
                        'fahrenheit'
                    ],
                    'description': 'The unit to return the temperature in. Defaults to \
↪'celsius\'.'
                }
            },
            'required': [
                'location'
            ]
        }
    }
}, {
    'type': 'function',
    'function': {
        'name': 'get_temperature_date',
        'description': 'Get temperature at a location and date.',
        'parameters': {
            'type': 'object',
            'properties': {
```

(continues on next page)

(continued from previous page)

```
                'location': {
                    'type': 'string',
                    'description': 'The location to get the temperature for, in the␣
↪format \'City, State, Country\'.'
                },
                'date': {
                    'type': 'string',
                    'description': 'The date to get the temperature for, in the format \
↪'Year-Month-Day\'.'
                },
                'unit': {
                    'type': 'string',
                    'enum': [
                        'celsius',
                        'fahrenheit'
                    ],
                    'description': 'The unit to return the temperature in. Defaults to \
↪'celsius\'.'
                }
            },
            'required': [
                'location',
                'date'
            ]
        }
    }
}]
messages = [{'role': 'user', 'content': 'Today is 2024-11-14, What\'s the temperature in␣
↪San Francisco now? How about tomorrow?'}]

client = OpenAI(api_key='YOUR_API_KEY', base_url='http://0.0.0.0:23333/v1')
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
    model=model_name,
    messages=messages,
    temperature=0.8,
    top_p=0.8,
    stream=False,
    tools=tools)
print(response.choices[0].message.tool_calls)
messages.append(response.choices[0].message)

for tool_call in response.choices[0].message.tool_calls:
    tool_call_args = json.loads(tool_call.function.arguments)
    tool_call_result =  get_function_by_name(tool_call.function.name)(**tool_call_args)
    messages.append({
        'role': 'tool',
        'name': tool_call.function.name,
        'content': tool_call_result,
        'tool_call_id': tool_call.id
    })
```

(continues on next page)

(continued from previous page)

```
response = client.chat.completions.create(
    model=model_name,
    messages=messages,
    temperature=0.8,
    top_p=0.8,
    stream=False,
    tools=tools)
print(response.choices[0].message.content)
```

Using the Qwen2.5-14B-Instruct, similar results can be obtained as follows

```
[ChatCompletionMessageToolCall(id='0', function=Function(arguments='{"location": "San␣
↪Francisco, California, USA"}', name='get_current_temperature'), type='function'),
 ChatCompletionMessageToolCall(id='1', function=Function(arguments='{"location": "San␣
↪Francisco, California, USA", "date": "2024-11-15"}', name='get_temperature_date'),␣
↪type='function')]

The current temperature in San Francisco, California, USA is 26.1°C. For tomorrow, 2024-
↪11-15, the temperature is expected to be 25.9°C.
```

It is important to note that in scenarios involving multiple tool calls, the order of the tool call results can affect the response quality. The tool_call_id has not been correctly provided to the LLM.

1.8 Serving LoRA

1.8.1 Launch LoRA

LoRA is currently only supported by the PyTorch backend. Its deployment process is similar to that of other models, and you can view the commands using lmdeploy `serve api_server -h`. Among the parameters supported by the PyTorch backend, there are configuration options for LoRA.

```
PyTorch engine arguments:
  --adapters [ADAPTERS [ADAPTERS ...]]
                        Used to set path(s) of lora adapter(s). One can input key-value␣
↪pairs in xxx=yyy format for multiple lora adapters. If only have one adapter, one can␣
↪only input the path of the adapter.. Default:
                        None. Type: str
```

The user only needs to pass the Hugging Face model path of the LoRA weights in the form of a dictionary to `--adapters`.

```
lmdeploy serve api_server THUDM/chatglm2-6b --adapters mylora=chenchi/lora-chatglm2-6b-
↪guodegang
```

After the service starts, you can find two available model names in the Swagger UI: 'THUDM/chatglm2-6b' and 'mylora'. The latter is the key in the `--adapters` dictionary.

1.8.2 Client usage

CLI

When using the OpenAI endpoint, the `model` parameter can be used to select either the base model or a specific LoRA weight for inference. The following example chooses to use the provided `chenchi/lora-chatglm2-6b-guodegang` for inference.

```
curl -X 'POST' \
  'http://localhost:23334/v1/chat/completions' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "model": "mylora",
  "messages": [
    {
      "content": "hi",
      "role": "user"
    }
  ]
}'
```

And here is the output:

```
{
  "id": "2",
  "object": "chat.completion",
  "created": 1721377275,
  "model": "mylora",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": " “”",
        "tool_calls": null
      },
      "logprobs": null,
      "finish_reason": "stop"
    }
  ],
  "usage": {
    "prompt_tokens": 17,
    "total_tokens": 43,
    "completion_tokens": 26
  }
}
```

python

```
from openai import OpenAI
client = OpenAI(
    api_key='YOUR_API_KEY',
    base_url="http://0.0.0.0:23333/v1"
)
model_name = 'mylora'
response = client.chat.completions.create(
  model=model_name,
  messages=[
    {"role": "user", "content": "hi"},
  ],
    temperature=0.8,
    top_p=0.8
)
print(response)
```

The printed response content is:

```
ChatCompletion(id='4', choices=[Choice(finish_reason='stop', index=0, logprobs=None,␣
↪message=ChatCompletionMessage(content=' ', role='assistant', function_call=None, tool_
↪calls=None))], created=1721377497, model='mylora', object='chat.completion', service_
↪tier=None, system_fingerprint=None, usage=CompletionUsage(completion_tokens=22, prompt_
↪tokens=17, total_tokens=39))
```

1.9 WebUI Demo

Starting an LLM model's gradio service with LMDeploy and interacting with the model on the WebUI is incredibly simple.

```
pip install lmdeploy[serve]
lmdeploy serve gradio {model_path}
```

All it takes is one-line command, with the `{model_path}` replaced by the model ID from huggingface hub, such as `internlm/internlm2_5-7b-chat`, or the local path to the model.

For detailed parameters of the command, please turn to `lmdeploy serve gradio -h` for help.

1.9.1 Create a huggingface demo

If you want to create an online demo project for your model on huggingface, please follow the steps below.

Step 1: Create space

First, register for a Hugging Face account. After successful registration, click on your profile picture in the upper right corner and select "New Space" to create one. Follow the Hugging Face guide to choose the necessary configurations, and you will have a blank demo space ready.

Step 2: Develop demo's entrypoint `app.py`

Replace the content of `app.py` in your space with the following code:

```python
from lmdeploy.serve.gradio.turbomind_coupled import run_local
from lmdeploy.messages import TurbomindEngineConfig

backend_config = TurbomindEngineConfig(max_batch_size=8)
model_path = 'internlm/internlm2_5-7b-chat'
run_local(model_path, backend_config=backend_config, server_name="huggingface-space")
```

Create a `requirements.txt` file with the following content:

```
lmdeploy
```

1.9.2 FAQs

- ZeroGPU compatibility issue. ZeroGPU is not suitable for LMDeploy turbomind engine. Please use the standard GPUs. Or, you can change the backend config in the above code to `PyTorchEngineConfig` to use the ZeroGPU.
- Gradio version issue, versions above 4.0.0 are currently not supported. You can modify this in `app.py`, for example:

  ```python
  import os
  os.system("pip uninstall -y gradio")
  os.system("pip install gradio==3.43.0")
  ```

1.10 Request Distributor Server

The request distributor service can parallelize multiple api_server services. Users only need to access the proxy URL, and they can indirectly access different api_server services. The proxy service will automatically distribute requests internally, achieving load balancing.

1.10.1 Startup

Start the proxy service:

```
lmdeploy serve proxy --server-name {server_name} --server-port {server_port} --strategy
↪"min_expected_latency"
```

After startup is successful, the URL of the proxy service will also be printed by the script. Access this URL in your browser to open the Swagger UI. Subsequently, users can add it directly to the proxy service when starting the `api_server` service by using the `--proxy-url` command. For example: `lmdeploy serve api_server`

InternLM/internlm2-chat-1_8b --proxy-url http://0.0.0.0:8000 In this way, users can access the services of the api_server through the proxy node, and the usage of the proxy node is exactly the same as that of the api_server, both of which are compatible with the OpenAI format.

- /v1/models
- /v1/chat/completions
- /v1/completions

1.10.2 Node Management

Through Swagger UI, we can see multiple APIs. Those related to api_server node management include:

- /nodes/status
- /nodes/add
- /nodes/remove

They respectively represent viewing all api_server service nodes, adding a certain node, and deleting a certain node.

Node Management through curl

```
curl -X 'GET' \
  'http://localhost:8000/nodes/status' \
  -H 'accept: application/json'
```

```
curl -X 'POST' \
  'http://localhost:8000/nodes/add' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "url": "http://0.0.0.0:23333"
}'
```

```
curl -X 'POST' \
  'http://localhost:8000/nodes/remove?node_url=http://0.0.0.0:23333' \
  -H 'accept: application/json' \
  -d ''
```

Node Management through python

```python
# query all nodes
import requests
url = 'http://localhost:8000/nodes/status'
headers = {'accept': 'application/json'}
response = requests.get(url, headers=headers)
print(response.text)
```

```python
# add a new node
import requests
```

(continues on next page)

(continued from previous page)

```
url = 'http://localhost:8000/nodes/add'
headers = {
    'accept': 'application/json',
    'Content-Type': 'application/json'
}
data = {"url": "http://0.0.0.0:23333"}
response = requests.post(url, headers=headers, json=data)
print(response.text)
```

```
# delete a node
import requests
url = 'http://localhost:8000/nodes/remove'
headers = {'accept': 'application/json',}
params = {'node_url': 'http://0.0.0.0:23333',}
response = requests.post(url, headers=headers, data='', params=params)
print(response.text)
```

1.10.3 Dispatch Strategy

The current distribution strategies of the proxy service are as follows:

- random dispatches based on the ability of each api_server node provided by the user to process requests. The greater the request throughput, the more likely it is to be allocated. Nodes that do not provide throughput are treated according to the average throughput of other nodes.
- min_expected_latency allocates based on the number of requests currently waiting to be processed on each node, and the throughput capability of each node, calculating the expected time required to complete the response. The shortest one gets allocated. Nodes that do not provide throughput are treated similarly.
- min_observed_latency allocates based on the average time required to handle a certain number of past requests on each node. The one with the shortest time gets allocated.

1.11 Offline Inference Pipeline

LMDeploy abstracts the complex inference process of multi-modal Vision-Language Models (VLM) into an easy-to-use pipeline, similar to the Large Language Model (LLM) inference *pipeline*.

The supported models are listed *here*. We genuinely invite the community to contribute new VLM support to LMDeploy. Your involvement is truly appreciated.

This article showcases the VLM pipeline using the OpenGVLab/InternVL2_5-8B model as a case study. You'll learn about the simplest ways to leverage the pipeline and how to gradually unlock more advanced features by adjusting engine parameters and generation arguments, such as tensor parallelism, context window sizing, random sampling, and chat template customization. Moreover, we will provide practical inference examples tailored to scenarios with multiple images, batch prompts etc.

Using the pipeline interface to infer other VLM models is similar, with the main difference being the configuration and installation dependencies of the models. You can read here for environment installation and configuration methods for different models.

1.11.1 A 'Hello, world' example

```python
from lmdeploy import pipeline
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2_5-8B')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

If ImportError occurs while executing this case, please install the required dependency packages as prompted.

In the above example, the inference prompt is a tuple structure consisting of (prompt, image). Besides this structure, the pipeline also supports prompts in the OpenAI format:

```python
from lmdeploy import pipeline

pipe = pipeline('OpenGVLab/InternVL2_5-8B')

prompts = [
    {
        'role': 'user',
        'content': [
            {'type': 'text', 'text': 'describe this image'},
            {'type': 'image_url', 'image_url': {'url': 'https://raw.githubusercontent.
↪com/open-mmlab/mmdeploy/main/tests/data/tiger.jpeg'}}
        ]
    }
]
response = pipe(prompts)
print(response)
```

Set tensor parallelism

Tensor paramllelism can be activated by setting the engine parameter `tp`

```python
from lmdeploy import pipeline, TurbomindEngineConfig
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2_5-8B',
                backend_config=TurbomindEngineConfig(tp=2))

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

Set context window size

When creating the pipeline, you can customize the size of the context window by setting the engine parameter `session_len`.

```
from lmdeploy import pipeline, TurbomindEngineConfig
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2_5-8B',
                backend_config=TurbomindEngineConfig(session_len=8192))

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

Set sampling parameters

You can change the default sampling parameters of pipeline by passing `GenerationConfig`

```
from lmdeploy import pipeline, GenerationConfig, TurbomindEngineConfig
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2_5-8B',
                backend_config=TurbomindEngineConfig(tp=2, session_len=8192))
gen_config = GenerationConfig(top_k=40, top_p=0.8, temperature=0.6)
image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image), gen_config=gen_config)
print(response)
```

Customize image token position

By default, LMDeploy inserts the special image token into the user prompt following the chat template defined by the upstream algorithm repository. However, for certain models where the image token's position is unrestricted, such as deepseek-vl, or when users require a customized image token placement, manual insertion of the special image token into the prompt is necessary. LMDeploy use `<IMAGE_TOKEN>` as the special image token.

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image
from lmdeploy.vl.constants import IMAGE_TOKEN

pipe = pipeline('deepseek-ai/deepseek-vl-1.3b-chat')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe((f'describe this image{IMAGE_TOKEN}', image))
print(response)
```

Set chat template

While performing inference, LMDeploy identifies an appropriate chat template from its builtin collection based on the model path and subsequently applies this template to the input prompts. However, when a chat template cannot be told from its model path, users have to specify it. For example, liuhaotian/llava-v1.5-7b employs the 'llava-v1' chat template, if user have a custom folder name instead of the official 'llava-v1.5-7b', the user needs to specify it by setting 'llava-v1' to `ChatTemplateConfig` as follows:

```
from lmdeploy import pipeline, ChatTemplateConfig
from lmdeploy.vl import load_image
pipe = pipeline('local_model_folder',
                chat_template_config=ChatTemplateConfig(model_name='llava-v1'))
image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

For more information about customizing a chat template, please refer to *this* guide

Setting vision model parameters

The default parameters of the visual model can be modified by setting `VisionConfig`.

```
from lmdeploy import pipeline, VisionConfig
from lmdeploy.vl import load_image
vision_config=VisionConfig(max_batch_size=16)
pipe = pipeline('liuhaotian/llava-v1.5-7b', vision_config=vision_config)
image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

Output logits for generated tokens

```
from lmdeploy import pipeline, GenerationConfig
from lmdeploy.vl import load_image
pipe = pipeline('OpenGVLab/InternVL2_5-8B')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')

response = pipe(('describe this image', image),
                gen_config=GenerationConfig(output_logits='generation'))
logits = response.logits
print(logits)
```

1.11.2 Multi-images inference

When dealing with multiple images, you can put them all in one list. Keep in mind that multiple images will lead to a higher number of input tokens, and as a result, the size of the *context window* typically needs to be increased.

```
from lmdeploy import pipeline, TurbomindEngineConfig
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2_5-8B',
                backend_config=TurbomindEngineConfig(session_len=8192))

image_urls=[
    'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/demo/resources/human-
↪pose.jpg',
    'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/demo/resources/det.jpg'
]

images = [load_image(img_url) for img_url in image_urls]
response = pipe(('describe these images', images))
print(response)
```

1.11.3 Batch prompts inference

Conducting inference with batch prompts is quite straightforward; just place them within a list structure:

```
from lmdeploy import pipeline, TurbomindEngineConfig
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2_5-8B',
                backend_config=TurbomindEngineConfig(session_len=8192))

image_urls=[
    "https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/demo/resources/human-
↪pose.jpg",
    "https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/demo/resources/det.jpg"
]
prompts = [('describe this image', load_image(img_url)) for img_url in image_urls]
response = pipe(prompts)
print(response)
```

1.11.4 Multi-turn conversation

There are two ways to do the multi-turn conversations with the pipeline. One is to construct messages according to the format of OpenAI and use above introduced method, the other is to use the `pipeline.chat` interface.

```
from lmdeploy import pipeline, TurbomindEngineConfig, GenerationConfig
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2_5-8B',
                backend_config=TurbomindEngineConfig(session_len=8192))

```

(continues on next page)

(continued from previous page)

```
image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/demo/
↪resources/human-pose.jpg')
gen_config = GenerationConfig(top_k=40, top_p=0.8, temperature=0.8)
sess = pipe.chat(('describe this image', image), gen_config=gen_config)
print(sess.response.text)
sess = pipe.chat('What is the woman doing?', session=sess, gen_config=gen_config)
print(sess.response.text)
```

1.12 OpenAI Compatible Server

This article primarily discusses the deployment of a single large vision language model across multiple GPUs on a single node, providing a service that is compatible with the OpenAI interface, as well as the usage of the service API. For the sake of convenience, we refer to this service as `api_server`. Regarding parallel services with multiple models, please refer to the guide about *Request Distribution Server*.

In the following sections, we will first introduce two methods for starting the service, choosing the appropriate one based on your application scenario.

Next, we focus on the definition of the service's RESTful API, explore the various ways to interact with the interface, and demonstrate how to try the service through the Swagger UI or LMDeploy CLI tools.

Finally, we showcase how to integrate the service into a WebUI, providing you with a reference to easily set up a demonstration demo.

1.12.1 Launch Service

Take the llava-v1.6-vicuna-7b model hosted on huggingface hub as an example, you can choose one the following methods to start the service.

Option 1: Launching with lmdeploy CLI

```
lmdeploy serve api_server liuhaotian/llava-v1.6-vicuna-7b --server-port 23333
```

The arguments of `api_server` can be viewed through the command `lmdeploy serve api_server -h`, for instance, `--tp` to set tensor parallelism, `--session-len` to specify the max length of the context window, `--cache-max-entry-count` to adjust the GPU mem ratio for k/v cache etc.

Option 2: Deploying with docker

With LMDeploy official docker image, you can run OpenAI compatible server as follows:

```
docker run --runtime nvidia --gpus all \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    --env "HUGGING_FACE_HUB_TOKEN=<secret>" \
    -p 23333:23333 \
    --ipc=host \
    openmmlab/lmdeploy:latest \
    lmdeploy serve api_server liuhaotian/llava-v1.6-vicuna-7b
```

The parameters of `api_server` are the same with that mentioned in "*option 1*" section

Each model may require specific dependencies not included in the Docker image. If you run into issues, you may need to install those yourself on a case-by-case basis. If in doubt, refer to the specific model's project for documentation.

For example, for Llava:

```
FROM openmmlab/lmdeploy:latest

RUN apt-get update && apt-get install -y python3 python3-pip git

WORKDIR /app

RUN pip3 install --upgrade pip
RUN pip3 install timm
RUN pip3 install git+https://github.com/haotian-liu/LLaVA.git --no-deps

COPY . .

CMD ["lmdeploy", "serve", "api_server", "liuhaotian/llava-v1.6-34b"]
```

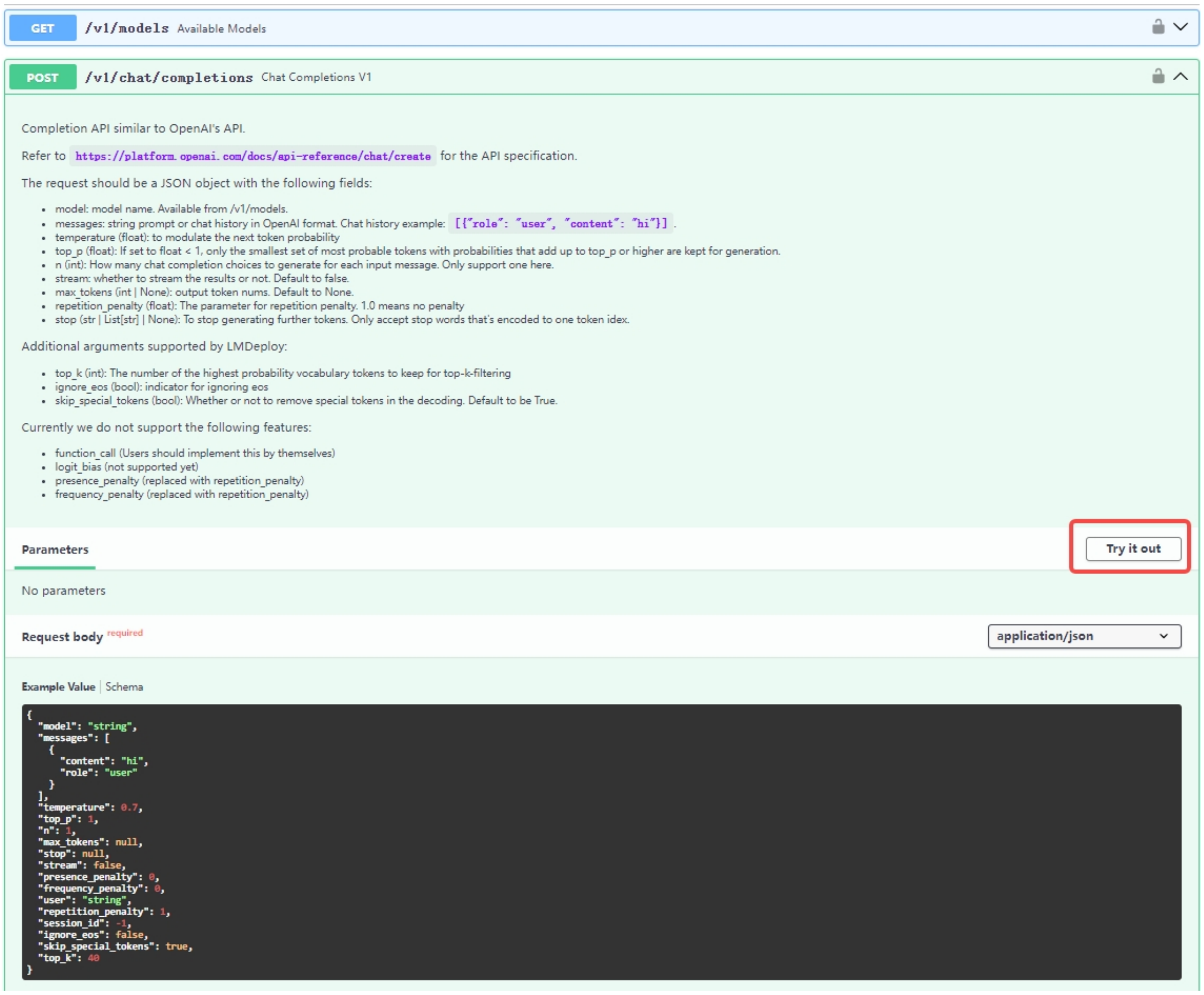
1.12.2 RESTful API

LMDeploy's RESTful API is compatible with the following three OpenAI interfaces:

- /v1/chat/completions
- /v1/models
- /v1/completions

The interface for image interaction is `/v1/chat/completions`, which is consistent with OpenAI.

You can overview and try out the offered RESTful APIs by the website `http://0.0.0.0:23333` as shown in the below image after launching the service successfully.



GET /v1/models Available Models

POST /v1/chat/completions Chat Completions V1

Completion API similar to OpenAI's API.

Refer to `https://platform.openai.com/docs/api-reference/chat/create` for the API specification.

The request should be a JSON object with the following fields:

- model: model name. Available from /v1/models.
- messages: string prompt or chat history in OpenAI format. Chat history example: `[{"role": "user", "content": "hi"}]`.
- temperature (float): to modulate the next token probability
- top_p (float): If set to float < 1, only the smallest set of most probable tokens with probabilities that add up to top_p or higher are kept for generation.
- n (int): How many chat completion choices to generate for each input message. Only support one here.
- stream: whether to stream the results or not. Default to false.
- max_tokens (int | None): output token nums. Default to None.
- repetition_penalty (float): The parameter for repetition penalty. 1.0 means no penalty
- stop (str | List[str] | None): To stop generating further tokens. Only accept stop words that's encoded to one token idex.

Additional arguments supported by LMDeploy:

- top_k (int): The number of the highest probability vocabulary tokens to keep for top-k-filtering
- ignore_eos (bool): indicator for ignoring eos
- skip_special_tokens (bool): Whether or not to remove special tokens in the decoding. Default to be True.

Currently we do not support the following features:

- function_call (Users should implement this by themselves)
- logit_bias (not supported yet)
- presence_penalty (replaced with repetition_penalty)
- frequency_penalty (replaced with repetition_penalty)

Parameters

Try it out

No parameters

Request body required

application/json

Example Value | Schema

```
{
  "model": "string",
  "messages": [
    {
      "content": "hi",
      "role": "user"
    }
  ],
  "temperature": 0.7,
  "top_p": 1,
  "n": 1,
  "max_tokens": null,
  "stop": null,
  "stream": false,
  "presence_penalty": 0,
  "frequency_penalty": 0,
  "user": "string",
  "repetition_penalty": 1,
  "session_id": -1,
  "ignore_eos": false,
  "skip_special_tokens": true,
  "top_k": 40
}
```

If you need to integrate the service into your own projects or products, we recommend the following approach:

Integrate with OpenAI

Here is an example of interaction with the endpoint `v1/chat/completions` service via the openai package. Before running it, please install the openai package by `pip install openai`

```python
from openai import OpenAI

client = OpenAI(api_key='YOUR_API_KEY', base_url='http://0.0.0.0:23333/v1')
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role':
        'user',
        'content': [{
            'type': 'text',
            'text': 'Describe the image please',
        }, {
```

(continues on next page)

(continued from previous page)

```
            'type': 'image_url',
            'image_url': {
                'url':
                'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/data/
↪tiger.jpeg',
            },
        }],
    }],
    temperature=0.8,
    top_p=0.8)
print(response)
```

Integrate with lmdeploy `APIClient`

Below are some examples demonstrating how to visit the service through `APIClient`

If you want to use the `/v1/chat/completions` endpoint, you can try the following code:

```
from lmdeploy.serve.openai.api_client import APIClient

api_client = APIClient(f'http://0.0.0.0:23333')
model_name = api_client.available_models[0]
messages = [{
    'role':
    'user',
    'content': [{
        'type': 'text',
        'text': 'Describe the image please',
    }, {
        'type': 'image_url',
        'image_url': {
            'url':
            'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/data/tiger.
↪jpeg',
        },
    }]
}]
for item in api_client.chat_completions_v1(model=model_name,
                                           messages=messages):
    print(item)
```

Integrate with Java/Golang/Rust

May use openapi-generator-cli to convert `http://{server_ip}:{server_port}/openapi.json` to java/rust/golang client. Here is an example:

```
$ docker run -it --rm -v ${PWD}:/local openapitools/openapi-generator-cli generate -i /
↪local/openapi.json -g rust -o /local/rust

$ ls rust/*
```

(continues on next page)

(continued from previous page)

```
rust/Cargo.toml  rust/git_push.sh  rust/README.md

rust/docs:
ChatCompletionRequest.md  EmbeddingsRequest.md  HttpValidationError.md  LocationInner.md␣
↪ Prompt.md
DefaultApi.md             GenerateRequest.md    Input.md                Messages.md     ␣
↪ ValidationError.md

rust/src:
apis  lib.rs  models
```

1.13 Vision-Language Models

1.13.1 LLaVA

LMDeploy supports the following llava series of models, which are detailed in the table below:

Model	Size	Supported Inference Engine
llava-hf/Llava-interleave-qwen-7b-hf	7B	TurboMind, PyTorch
llava-hf/llava-1.5-7b-hf	7B	TurboMind, PyTorch
llava-hf/llava-v1.6-mistral-7b-hf	7B	PyTorch
llava-hf/llava-v1.6-vicuna-7b-hf	7B	PyTorch
liuhaotian/llava-v1.6-mistral-7b	7B	TurboMind
liuhaotian/llava-v1.6-vicuna-7b	7B	TurboMind

The next chapter demonstrates how to deploy an Llava model using LMDeploy, with llava-hf/llava-interleave as an example.

Note

PyTorch engine removes the support of original llava models after v0.6.4. Please use their corresponding transformers models instead, which can be found in https://huggingface.co/llava-hf

Installation

Please install LMDeploy by following the *installation guide*.

Or, you can go with office docker image:

```
docker pull openmmlab/lmdeploy:latest
```

Offline inference

The following sample code shows the basic usage of VLM pipeline. For detailed information, please refer to *VLM Offline Inference Pipeline*

```
from lmdeploy import GenerationConfig, TurbomindEngineConfig, pipeline
from lmdeploy.vl import load_image


pipe = pipeline("llava-hf/llava-interleave-qwen-7b-hf", backend_
↪config=TurbomindEngineConfig(cache_max_entry_count=0.5),
    gen_config=GenerationConfig(max_new_tokens=512))

image = load_image('https://qianwen-res.oss-cn-beijing.aliyuncs.com/Qwen-VL/assets/demo.
↪jpeg')
prompt = 'Describe the image.'
print(f'prompt:{prompt}')
response = pipe((prompt, image))
print(response)
```

More examples are listed below:

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('llava-hf/llava-interleave-qwen-7b-hf', log_level='INFO')
messages = [
    dict(role='user', content=[
        dict(type='text', text='Describe the two images in detail.'),
        dict(type='image_url', image_url=dict(url='https://raw.githubusercontent.com/
↪QwenLM/Qwen-VL/master/assets/mm_tutorial/Beijing_Small.jpeg')),
        dict(type='image_url', image_url=dict(url='https://raw.githubusercontent.com/
↪QwenLM/Qwen-VL/master/assets/mm_tutorial/Chongqing_Small.jpeg'))
    ])
]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='What are the similarities and differences␣
↪between these two images.'))
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
```

Online serving

You can launch the server by the `lmdeploy serve api_server` CLI:

```
lmdeploy serve api_server llava-hf/llava-interleave-qwen-7b-hf
```

You can also start the service using the aforementioned built docker image:

```
docker run --runtime nvidia --gpus all \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    --env "HUGGING_FACE_HUB_TOKEN=<secret>" \
```

(continues on next page)

(continued from previous page)

```
    -p 23333:23333 \
    --ipc=host \
    openmmlab/lmdeploy:latest \
    lmdeploy serve api_server llava-hf/llava-interleave-qwen-7b-hf
```

The docker compose is another option. Create a `docker-compose.yml` configuration file in the root directory of the lmdeploy project as follows:

```yaml
version: '3.5'

services:
  lmdeploy:
    container_name: lmdeploy
    image: openmmlab/lmdeploy:latest
    ports:
      - "23333:23333"
    environment:
      HUGGING_FACE_HUB_TOKEN: <secret>
    volumes:
      - ~/.cache/huggingface:/root/.cache/huggingface
    stdin_open: true
    tty: true
    ipc: host
    command: lmdeploy serve api_server llava-hf/llava-interleave-qwen-7b-hf
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: "all"
              capabilities: [gpu]
```

Then, you can execute the startup command as below:

```
docker-compose up -d
```

If you find the following logs after running `docker logs -f lmdeploy`, it means the service launches successfully.

```
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
INFO:     Started server process [2439]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on  http://0.0.0.0:23333  (Press CTRL+C to quit)
```

The arguments of `lmdeploy serve api_server` can be reviewed in detail by `lmdeploy serve api_server -h`.

More information about `api_server` as well as how to access the service can be found from *here*

1.13.2 InternVL

LMDeploy supports the following InternVL series of models, which are detailed in the table below:

Model	Size	Supported Inference Engine
InternVL	13B-19B	TurboMind
InternVL1.5	2B-26B	TurboMind, PyTorch
InternVL2	1B, 4B	PyTorch
InternVL2	2B, 8B-76B	TurboMind, PyTorch
Mono-InternVL	2B	PyTorch

The next chapter demonstrates how to deploy an InternVL model using LMDeploy, with InternVL2-8B as an example.

Installation

Please install LMDeploy by following the *installation guide*, and install other packages that InternVL2 needs

```
pip install timm
# It is recommended to find the whl package that matches the environment from the␣
↪releases on https://github.com/Dao-AILab/flash-attention.
pip install flash-attn
```

Or, you can build a docker image to set up the inference environment. If the CUDA version on your host machine is >=12.4, you can run:

```
docker build --build-arg CUDA_VERSION=cu12 -t openmmlab/lmdeploy:internvl . -f ./docker/
↪InternVL_Dockerfile
```

Otherwise, you can go with:

```
git clone https://github.com/InternLM/lmdeploy.git
cd lmdeploy
docker build --build-arg CUDA_VERSION=cu11 -t openmmlab/lmdeploy:internvl . -f ./docker/
↪InternVL_Dockerfile
```

Offline inference

The following sample code shows the basic usage of VLM pipeline. For detailed information, please refer to *VLM Offline Inference Pipeline*

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2-8B')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe((f'describe this image', image))
print(response)
```

More examples are listed below:

```python
from lmdeploy import pipeline, GenerationConfig
from lmdeploy.vl.constants import IMAGE_TOKEN

pipe = pipeline('OpenGVLab/InternVL2-8B', log_level='INFO')
messages = [
    dict(role='user', content=[
        dict(type='text', text=f'{IMAGE_TOKEN}{IMAGE_TOKEN}\nDescribe the two images in␣
↪detail.'),
        dict(type='image_url', image_url=dict(max_dynamic_patch=12, url='https://raw.
↪githubusercontent.com/OpenGVLab/InternVL/main/internvl_chat/examples/image1.jpg')),
        dict(type='image_url', image_url=dict(max_dynamic_patch=12, url='https://raw.
↪githubusercontent.com/OpenGVLab/InternVL/main/internvl_chat/examples/image2.jpg'))
    ])
]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='What are the similarities and differences␣
↪between these two images.'))
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
```

```python
from lmdeploy import pipeline, GenerationConfig
from lmdeploy.vl.constants import IMAGE_TOKEN

pipe = pipeline('OpenGVLab/InternVL2-8B', log_level='INFO')
messages = [
    dict(role='user', content=[
        dict(type='text', text=f'Image-1: {IMAGE_TOKEN}\nImage-2: {IMAGE_TOKEN}\
↪nDescribe the two images in detail.'),
        dict(type='image_url', image_url=dict(max_dynamic_patch=12, url='https://raw.
↪githubusercontent.com/OpenGVLab/InternVL/main/internvl_chat/examples/image1.jpg')),
        dict(type='image_url', image_url=dict(max_dynamic_patch=12, url='https://raw.
↪githubusercontent.com/OpenGVLab/InternVL/main/internvl_chat/examples/image2.jpg'))
    ])
]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='What are the similarities and differences␣
↪between these two images.'))
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
```

```python
import numpy as np
from lmdeploy import pipeline, GenerationConfig
from decord import VideoReader, cpu
from lmdeploy.vl.constants import IMAGE_TOKEN
from lmdeploy.vl.utils import encode_image_base64
from PIL import Image
pipe = pipeline('OpenGVLab/InternVL2-8B', log_level='INFO')


def get_index(bound, fps, max_frame, first_idx=0, num_segments=32):
```

(continues on next page)

(continued from previous page)

```
    if bound:
        start, end = bound[0], bound[1]
    else:
        start, end = -100000, 100000
    start_idx = max(first_idx, round(start * fps))
    end_idx = min(round(end * fps), max_frame)
    seg_size = float(end_idx - start_idx) / num_segments
    frame_indices = np.array([
        int(start_idx + (seg_size / 2) + np.round(seg_size * idx))
        for idx in range(num_segments)
    ])
    return frame_indices


def load_video(video_path, bound=None, num_segments=32):
    vr = VideoReader(video_path, ctx=cpu(0), num_threads=1)
    max_frame = len(vr) - 1
    fps = float(vr.get_avg_fps())
    pixel_values_list, num_patches_list = [], []
    frame_indices = get_index(bound, fps, max_frame, first_idx=0, num_segments=num_
↪segments)
    imgs = []
    for frame_index in frame_indices:
        img = Image.fromarray(vr[frame_index].asnumpy()).convert('RGB')
        imgs.append(img)
    return imgs


video_path = 'red-panda.mp4'
imgs = load_video(video_path, num_segments=8)

question = ''
for i in range(len(imgs)):
    question = question + f'Frame{i+1}: {IMAGE_TOKEN}\n'

question += 'What is the red panda doing?'

content = [{'type': 'text', 'text': question}]
for img in imgs:
    content.append({'type': 'image_url', 'image_url': {'max_dynamic_patch': 1, 'url': f
↪'data:image/jpeg;base64,{encode_image_base64(img)}'}})

messages = [dict(role='user', content=content)]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='Describe this video in detail. Don\'t repeat.
↪'))
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
```

Online serving

You can launch the server by the `lmdeploy serve api_server` CLI:

```
lmdeploy serve api_server OpenGVLab/InternVL2-8B
```

You can also start the service using the aforementioned built docker image:

```
docker run --runtime nvidia --gpus all \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    --env "HUGGING_FACE_HUB_TOKEN=<secret>" \
    -p 23333:23333 \
    --ipc=host \
    openmmlab/lmdeploy:internvl \
    lmdeploy serve api_server OpenGVLab/InternVL2-8B
```

The docker compose is another option. Create a `docker-compose.yml` configuration file in the root directory of the lmdeploy project as follows:

```
version: '3.5'

services:
  lmdeploy:
    container_name: lmdeploy
    image: openmmlab/lmdeploy:internvl
    ports:
      - "23333:23333"
    environment:
      HUGGING_FACE_HUB_TOKEN: <secret>
    volumes:
      - ~/.cache/huggingface:/root/.cache/huggingface
    stdin_open: true
    tty: true
    ipc: host
    command: lmdeploy serve api_server OpenGVLab/InternVL2-8B
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: "all"
              capabilities: [gpu]
```

Then, you can execute the startup command as below:

```
docker-compose up -d
```

If you find the following logs after running `docker logs -f lmdeploy`, it means the service launches successfully.

```
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
INFO:     Started server process [2439]
INFO:     Waiting for application startup.
```

(continues on next page)

(continued from previous page)

```
INFO:     Application startup complete.
INFO:     Uvicorn running on  http://0.0.0.0:23333  (Press CTRL+C to quit)
```

The arguments of `lmdeploy serve api_server` can be reviewed in detail by `lmdeploy serve api_server -h`.

More information about `api_server` as well as how to access the service can be found from *here*

1.13.3 InternLM-XComposer-2.5

Introduction

InternLM-XComposer-2.5 excels in various text-image comprehension and composition applications, achieving GPT-4V level capabilities with merely 7B LLM backend. IXC-2.5 is trained with 24K interleaved image-text contexts, it can seamlessly extend to 96K long contexts via RoPE extrapolation. This long-context capability allows IXC-2.5 to perform exceptionally well in tasks requiring extensive input and output contexts. LMDeploy supports model internlm/internlm-xcomposer2d5-7b in TurboMind engine.

Quick Start

Installation

Please install LMDeploy by following the *installation guide*, and install other packages that InternLM-XComposer-2.5 needs

```
pip install decord
```

Offline inference pipeline

The following sample code shows the basic usage of VLM pipeline. For more examples, please refer to *VLM Offline Inference Pipeline*

```python
from lmdeploy import pipeline
from lmdeploy.vl import load_image
from lmdeploy.vl.constants import IMAGE_TOKEN

pipe = pipeline('internlm/internlm-xcomposer2d5-7b')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe((f'describe this image', image))
print(response)
```

Lora Model

InternLM-XComposer-2.5 trained the LoRA weights for webpage creation and article writing. As TurboMind backend doesn't support slora, only one LoRA model can be deployed at a time, and the LoRA weights need to be merged when deploying the model. LMDeploy provides the corresponding conversion script, which is used as follows:

```
export HF_MODEL=internlm/internlm-xcomposer2d5-7b
export WORK_DIR=internlm/internlm-xcomposer2d5-7b-web
export TASK=web
python -m lmdeploy.vl.tools.merge_xcomposer2d5_task $HF_MODEL $WORK_DIR --task $TASK
```

Quantization

The following takes the base model as an example to show the quantization method. If you want to use the LoRA model, please merge the LoRA model according to the previous section.

```
export HF_MODEL=internlm/internlm-xcomposer2d5-7b
export WORK_DIR=internlm/internlm-xcomposer2d5-7b-4bit

lmdeploy lite auto_awq \
   $HF_MODEL \
  --work-dir $WORK_DIR
```

More examples

The following uses the `pipeline.chat` interface api as an example to demonstrate its usage. Other interfaces apis also support inference but require manually splicing of conversation content.

```
from lmdeploy import pipeline, GenerationConfig
from transformers.dynamic_module_utils import get_class_from_dynamic_module

HF_MODEL = 'internlm/internlm-xcomposer2d5-7b'
load_video = get_class_from_dynamic_module('ixc_utils.load_video', HF_MODEL)
frame2img = get_class_from_dynamic_module('ixc_utils.frame2img', HF_MODEL)
Video_transform = get_class_from_dynamic_module('ixc_utils.Video_transform', HF_MODEL)
get_font = get_class_from_dynamic_module('ixc_utils.get_font', HF_MODEL)

video = load_video('liuxiang.mp4') # https://github.com/InternLM/InternLM-XComposer/raw/
↪main/examples/liuxiang.mp4
img = frame2img(video, get_font())
img = Video_transform(img)

pipe = pipeline(HF_MODEL)
gen_config = GenerationConfig(top_k=50, top_p=0.8, temperature=1.0)
query = 'Here are some frames of a video. Describe this video in detail'
sess = pipe.chat((query, img), gen_config=gen_config)
print(sess.response.text)

query = 'tell me the athlete code of Liu Xiang'
sess = pipe.chat(query, session=sess, gen_config=gen_config)
print(sess.response.text)
```

```
from lmdeploy import pipeline, GenerationConfig
from lmdeploy.vl.constants import IMAGE_TOKEN
from lmdeploy.vl import load_image

query = f'Image1 {IMAGE_TOKEN}; Image2 {IMAGE_TOKEN}; Image3 {IMAGE_TOKEN}; I want to␣
↪buy a car from the three given cars, analyze their advantages and weaknesses one by one
↪'

urls = ['https://raw.githubusercontent.com/InternLM/InternLM-XComposer/main/examples/
↪cars1.jpg',
        'https://raw.githubusercontent.com/InternLM/InternLM-XComposer/main/examples/
↪cars2.jpg',
        'https://raw.githubusercontent.com/InternLM/InternLM-XComposer/main/examples/
↪cars3.jpg']
images = [load_image(url) for url in urls]

pipe = pipeline('internlm/internlm-xcomposer2d5-7b', log_level='INFO')
output = pipe((query, images), gen_config=GenerationConfig(top_k=0, top_p=0.8, random_
↪seed=89247526689433939))
```

Since LMDeploy does not support beam search, the generated results will be quite different from those using beam search with transformers. It is recommended to turn off top_k or use a larger top_k sampling to increase diversity.

Please first convert the web model using the instructions above.

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('/nvme/shared/internlm-xcomposer2d5-7b-web', log_level='INFO')
pipe.chat_template.meta_instruction = None

query = 'A website for Research institutions. The name is Shanghai AI lab. Top␣
↪Navigation Bar is blue.Below left, an image shows the logo of the lab. In the right,␣
↪there is a passage of text below that describes the mission of the laboratory.There␣
↪are several images to show the research projects of Shanghai AI lab.'
output = pipe(query, gen_config=GenerationConfig(max_new_tokens=2048))
```

When using transformers for testing, it is found that if repetition_penalty is set, there is a high probability that the decode phase will not stop if `num_beams` is set to 1. As LMDeploy does not support beam search, it is recommended to turn off repetition_penalty when using LMDeploy for inference.

Please first convert the write model using the instructions above.

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('/nvme/shared/internlm-xcomposer2d5-7b-write', log_level='INFO')
pipe.chat_template.meta_instruction = None

query = 'Please write a blog based on the title: French Pastries: A Sweet Indulgence'
output = pipe(query, gen_config=GenerationConfig(max_new_tokens=8192))
```

1.13.4 CogVLM

Introduction

CogVLM is a powerful open-source visual language model (VLM). LMDeploy supports CogVLM-17B models like THUDM/cogvlm-chat-hf and CogVLM2-19B models like THUDM/cogvlm2-llama3-chat-19B in PyTorch engine.

Quick Start

Install

Install torch, torchvision and xformers for CogVLM by referring to Pytorch and installing-xformers

```
# cuda 11.8
pip install torch==2.2.2 torchvision==0.17.2 xformers==0.0.26 --index-url https://
↪download.pytorch.org/whl/cu118
# cuda 12.1
pip install torch==2.2.2 torchvision==0.17.2 xformers==0.0.26 --index-url https://
↪download.pytorch.org/whl/cu121
```

Install LMDeploy by following the *installation guide*

Prepare

When deploying the **CogVLM** model using LMDeploy, it is necessary to download the model first, as the **CogVLM** model repository does not include the tokenizer model. However, this step is not required for **CogVLM2**.

Taking one **CogVLM** model `cogvlm-chat-hf` as an example, you can prepare it as follows:

```
huggingface-cli download THUDM/cogvlm-chat-hf --local-dir ./cogvlm-chat-hf --local-dir-
↪use-symlinks False
huggingface-cli download lmsys/vicuna-7b-v1.5 special_tokens_map.json tokenizer.model␣
↪tokenizer_config.json --local-dir ./cogvlm-chat-hf --local-dir-use-symlinks False
```

Offline inference pipeline

The following sample code shows the basic usage of VLM pipeline. For more examples, please refer to *VLM Offline Inference Pipeline*

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image


if __name__ == "__main__":
    pipe = pipeline('cogvlm-chat-hf')

    image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
    response = pipe(('describe this image', image))
    print(response)
```

1.13.5 MiniCPM-V

LMDeploy supports the following MiniCPM-V series of models, which are detailed in the table below:

Model	Supported Inference Engine
MiniCPM-Llama3-V-2_5	TurboMind
MiniCPM-V-2_6	TurboMind

The next chapter demonstrates how to deploy an MiniCPM-V model using LMDeploy, with MiniCPM-V-2_6 as an example.

Installation

Please install LMDeploy by following the *installation guide*.

Offline inference

The following sample code shows the basic usage of VLM pipeline. For detailed information, please refer to *VLM Offline Inference Pipeline*

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image

pipe = pipeline('openbmb/MiniCPM-V-2_6')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

More examples are listed below:

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('openbmb/MiniCPM-V-2_6', log_level='INFO')
messages = [
    dict(role='user', content=[
        dict(type='text', text='Describe the two images in detail.'),
        dict(type='image_url', image_url=dict(max_slice_nums=9, url='https://raw.
↪githubusercontent.com/OpenGVLab/InternVL/main/internvl_chat/examples/image1.jpg')),
        dict(type='image_url', image_url=dict(max_slice_nums=9, url='https://raw.
↪githubusercontent.com/OpenGVLab/InternVL/main/internvl_chat/examples/image2.jpg'))
    ])
]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
print(out.text)

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='What are the similarities and differences␣
↪between these two images.'))
```

(continues on next page)

(continued from previous page)

```
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
print(out.text)
```

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('openbmb/MiniCPM-V-2_6', log_level='INFO')

question = "production date"
messages = [
    dict(role='user', content=[
        dict(type='text', text=question),
        dict(type='image_url', image_url=dict(url='example1.jpg')),
    ]),
    dict(role='assistant', content='2023.08.04'),
    dict(role='user', content=[
        dict(type='text', text=question),
        dict(type='image_url', image_url=dict(url='example2.jpg')),
    ]),
    dict(role='assistant', content='2007.04.24'),
    dict(role='user', content=[
        dict(type='text', text=question),
        dict(type='image_url', image_url=dict(url='test.jpg')),
    ])
]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
print(out.text)
```

```
from lmdeploy import pipeline, GenerationConfig
from lmdeploy.vl.utils import encode_image_base64
import torch
from PIL import Image
from transformers import AutoModel, AutoTokenizer
from decord import VideoReader, cpu    # pip install decord

pipe = pipeline('openbmb/MiniCPM-V-2_6', log_level='INFO')

MAX_NUM_FRAMES=64 # if cuda OOM set a smaller number
def encode_video(video_path):
    def uniform_sample(l, n):
        gap = len(l) / n
        idxs = [int(i * gap + gap / 2) for i in range(n)]
        return [l[i] for i in idxs]
    vr = VideoReader(video_path, ctx=cpu(0))
    sample_fps = round(vr.get_avg_fps() / 1)  # FPS
    frame_idx = [i for i in range(0, len(vr), sample_fps)]
    if len(frame_idx) > MAX_NUM_FRAMES:
        frame_idx = uniform_sample(frame_idx, MAX_NUM_FRAMES)
    frames = vr.get_batch(frame_idx).asnumpy()
    frames = [Image.fromarray(v.astype('uint8')) for v in frames]
    print('num frames:', len(frames))
    return frames
```

(continues on next page)

(continued from previous page)

```
video_path="video_test.mp4"
frames = encode_video(video_path)
question = "Describe the video"

content=[dict(type='text', text=question)]
for frame in frames:
    content.append(dict(type='image_url', image_url=dict(use_image_id=False, max_slice_
↪nums=2,
        url=f'data:image/jpeg;base64,{encode_image_base64(frame)}')))

messages = [dict(role='user', content=content)]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
print(out.text)
```

Online serving

You can launch the server by the `lmdeploy serve api_server` CLI:

```
lmdeploy serve api_server openbmb/MiniCPM-V-2_6
```

You can also start the service using the official lmdeploy docker image:

```
docker run --runtime nvidia --gpus all \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    --env "HUGGING_FACE_HUB_TOKEN=<secret>" \
    -p 23333:23333 \
    --ipc=host \
    openmmlab/lmdeploy:latest \
    lmdeploy serve api_server openbmb/MiniCPM-V-2_6
```

The docker compose is another option. Create a `docker-compose.yml` configuration file in the root directory of the lmdeploy project as follows:

```
version: '3.5'

services:
  lmdeploy:
    container_name: lmdeploy
    image: openmmlab/lmdeploy:latest
    ports:
      - "23333:23333"
    environment:
      HUGGING_FACE_HUB_TOKEN: <secret>
    volumes:
      - ~/.cache/huggingface:/root/.cache/huggingface
    stdin_open: true
    tty: true
    ipc: host
    command: lmdeploy serve api_server openbmb/MiniCPM-V-2_6
    deploy:
```

(continues on next page)

(continued from previous page)

```
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: "all"
              capabilities: [gpu]
```

Then, you can execute the startup command as below:

```
docker-compose up -d
```

If you find the following logs after running `docker logs -f lmdeploy`, it means the service launches successfully.

```
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
INFO:     Started server process [2439]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on  http://0.0.0.0:23333  (Press CTRL+C to quit)
```

The arguments of `lmdeploy serve api_server` can be reviewed in detail by `lmdeploy serve api_server -h`.

More information about `api_server` as well as how to access the service can be found from *here*

1.13.6 Phi-3 Vision

Introduction

Phi-3 is a family of small language and multi-modal models from MicroSoft. LMDeploy supports the multi-modal models as below.

Model	Size	Supported Inference Engine
microsoft/Phi-3-vision-128k-instruct	4.2B	PyTorch
microsoft/Phi-3.5-vision-instruct	4.2B	PyTorch

The next chapter demonstrates how to deploy an Phi-3 model using LMDeploy, with microsoft/Phi-3.5-vision-instruct as an example.

Installation

Please install LMDeploy by following the *installation guide* and install the dependency Flash-Attention

```
# It is recommended to find the whl package that matches the environment from the␣
↪releases on https://github.com/Dao-AILab/flash-attention.
pip install flash-attn
```

Offline inference

The following sample code shows the basic usage of VLM pipeline. For more examples, please refer to *VLM Offline Inference Pipeline*

```python
from lmdeploy import pipeline
from lmdeploy.vl import load_image

pipe = pipeline('microsoft/Phi-3.5-vision-instruct')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

Online serving

Launch Service

You can launch the server by the `lmdeploy serve api_server` CLI:

```
lmdeploy serve api_server microsoft/Phi-3.5-vision-instruct
```

Integrate with `OpenAI`

Here is an example of interaction with the endpoint `v1/chat/completions` service via the openai package. Before running it, please install the openai package by `pip install openai`

```python
from openai import OpenAI

client = OpenAI(api_key='YOUR_API_KEY', base_url='http://0.0.0.0:23333/v1')
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role':
        'user',
        'content': [{
            'type': 'text',
            'text': 'Describe the image please',
        }, {
            'type': 'image_url',
            'image_url': {
                'url':
                'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/data/
↪tiger.jpeg',
            },
        }],
    }],
    temperature=0.8,
```

(continues on next page)

(continued from previous page)

```
    top_p=0.8)
print(response)
```

1.13.7 Mllama

Introduction

Llama3.2-VL is a family of large language and multi-modal models from Meta.

We will demonstrate how to deploy an Llama3.2-VL model using LMDeploy, with meta-llama/Llama-3.2-11B-Vision-Instruct as an example.

Installation

Please install LMDeploy by following the *installation guide*.

Offline inference

The following sample code shows the basic usage of VLM pipeline. For more examples, please refer to *VLM Offline Inference Pipeline*

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image

pipe = pipeline('meta-llama/Llama-3.2-11B-Vision-Instruct')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

Online serving

Launch Service

You can launch the server by the `lmdeploy serve api_server` CLI:

```
lmdeploy serve api_server meta-llama/Llama-3.2-11B-Vision-Instruct
```

Integrate with OpenAI

Here is an example of interaction with the endpoint `v1/chat/completions` service via the openai package. Before running it, please install the openai package by `pip install openai`

```
from openai import OpenAI

client = OpenAI(api_key='YOUR_API_KEY', base_url='http://0.0.0.0:23333/v1')
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role':
        'user',
        'content': [{
            'type': 'text',
            'text': 'Describe the image please',
        }, {
            'type': 'image_url',
            'image_url': {
                'url':
                'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/data/
↪tiger.jpeg',
            },
        }],
    }],
    temperature=0.8,
    top_p=0.8)
print(response)
```

1.13.8 Qwen2-VL

LMDeploy supports the following Qwen-VL series of models, which are detailed in the table below:

Model	Size	Supported Inference Engine
Qwen-VL-Chat	-	TurboMind
Qwen2-VL	2B, 7B	PyTorch

The next chapter demonstrates how to deploy an Qwen-VL model using LMDeploy, with Qwen2-VL-7B-Instruct as an example.

Installation

Please install LMDeploy by following the *installation guide*, and install other packages that Qwen2-VL needs

```
pip install qwen_vl_utils
```

Or, you can build a docker image to set up the inference environment. If the CUDA version on your host machine is >=12.4, you can run:

```
git clone https://github.com/InternLM/lmdeploy.git
cd lmdeploy
docker build --build-arg CUDA_VERSION=cu12 -t openmmlab/lmdeploy:qwen2vl . -f ./docker/
↪Qwen2VL_Dockerfile
```

Otherwise, you can go with:

```
docker build --build-arg CUDA_VERSION=cu11 -t openmmlab/lmdeploy:qwen2vl . -f ./docker/
↪Qwen2VL_Dockerfile
```

Offline inference

The following sample code shows the basic usage of VLM pipeline. For detailed information, please refer to *VLM Offline Inference Pipeline*

```python
from lmdeploy import pipeline
from lmdeploy.vl import load_image

pipe = pipeline('Qwen/Qwen2-VL-2B-Instruct')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe((f'describe this image', image))
print(response)
```

More examples are listed below:

```python
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('Qwen/Qwen2-VL-2B-Instruct', log_level='INFO')
messages = [
    dict(role='user', content=[
        dict(type='text', text='Describe the two images in detail.'),
        dict(type='image_url', image_url=dict(url='https://raw.githubusercontent.com/
↪QwenLM/Qwen-VL/master/assets/mm_tutorial/Beijing_Small.jpeg')),
        dict(type='image_url', image_url=dict(url='https://raw.githubusercontent.com/
↪QwenLM/Qwen-VL/master/assets/mm_tutorial/Chongqing_Small.jpeg'))
    ])
]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='What are the similarities and differences␣
```

(continues on next page)

(continued from previous page)

```
↪between these two images.'))
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
```

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('Qwen/Qwen2-VL-2B-Instruct', log_level='INFO')

min_pixels = 64 * 28 * 28
max_pixels = 64 * 28 * 28
messages = [
    dict(role='user', content=[
        dict(type='text', text='Describe the two images in detail.'),
        dict(type='image_url', image_url=dict(min_pixels=min_pixels, max_pixels=max_
↪pixels, url='https://raw.githubusercontent.com/QwenLM/Qwen-VL/master/assets/mm_
↪tutorial/Beijing_Small.jpeg')),
        dict(type='image_url', image_url=dict(min_pixels=min_pixels, max_pixels=max_
↪pixels, url='https://raw.githubusercontent.com/QwenLM/Qwen-VL/master/assets/mm_
↪tutorial/Chongqing_Small.jpeg'))
    ])
]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='What are the similarities and differences␣
↪between these two images.'))
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
```

Online serving

You can launch the server by the `lmdeploy serve api_server` CLI:

```
lmdeploy serve api_server Qwen/Qwen2-VL-2B-Instruct
```

You can also start the service using the aforementioned built docker image:

```
docker run --runtime nvidia --gpus all \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    --env "HUGGING_FACE_HUB_TOKEN=<secret>" \
    -p 23333:23333 \
    --ipc=host \
    openmmlab/lmdeploy:qwen2vl \
    lmdeploy serve api_server Qwen/Qwen2-VL-2B-Instruct
```

The docker compose is another option. Create a `docker-compose.yml` configuration file in the root directory of the lmdeploy project as follows:

```
version: '3.5'

services:
  lmdeploy:
    container_name: lmdeploy
```

(continues on next page)

(continued from previous page)

```
    image: openmmlab/lmdeploy:qwen2vl
    ports:
      - "23333:23333"
    environment:
      HUGGING_FACE_HUB_TOKEN: <secret>
    volumes:
      - ~/.cache/huggingface:/root/.cache/huggingface
    stdin_open: true
    tty: true
    ipc: host
    command: lmdeploy serve api_server Qwen/Qwen2-VL-2B-Instruct
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: "all"
              capabilities: [gpu]
```

Then, you can execute the startup command as below:

```
docker-compose up -d
```

If you find the following logs after running `docker logs -f lmdeploy`, it means the service launches successfully.

```
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
INFO:     Started server process [2439]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on  http://0.0.0.0:23333  (Press CTRL+C to quit)
```

The arguments of `lmdeploy serve api_server` can be reviewed in detail by `lmdeploy serve api_server -h`.

More information about `api_server` as well as how to access the service can be found from *here*

1.13.9 Molmo

LMDeploy supports the following molmo series of models, which are detailed in the table below:

Model	Size	Supported Inference Engine
Molmo-7B-D-0924	7B	TurboMind
Molmo-72-0924	72B	TurboMind

The next chapter demonstrates how to deploy a molmo model using LMDeploy, with Molmo-7B-D-0924 as an example.

Installation

Please install LMDeploy by following the *installation guide*

Offline inference

The following sample code shows the basic usage of VLM pipeline. For detailed information, please refer to *VLM Offline Inference Pipeline*

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image

pipe = pipeline('allenai/Molmo-7B-D-0924')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe((f'describe this image', image))
print(response)
```

More examples are listed below:

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('allenai/Molmo-7B-D-0924', log_level='INFO')
messages = [
    dict(role='user', content=[
        dict(type='text', text='Describe the two images in detail.'),
        dict(type='image_url', image_url=dict(url='https://raw.githubusercontent.com/
↪QwenLM/Qwen-VL/master/assets/mm_tutorial/Beijing_Small.jpeg')),
        dict(type='image_url', image_url=dict(url='https://raw.githubusercontent.com/
↪QwenLM/Qwen-VL/master/assets/mm_tutorial/Chongqing_Small.jpeg'))
    ])
]
out = pipe(messages, gen_config=GenerationConfig(do_sample=False))

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='What are the similarities and differences␣
↪between these two images.'))
out = pipe(messages, gen_config=GenerationConfig(do_sample=False))
```

Online serving

You can launch the server by the `lmdeploy serve api_server` CLI:

```
lmdeploy serve api_server allenai/Molmo-7B-D-0924
```

You can also start the service using the docker image:

```
docker run --runtime nvidia --gpus all \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    --env "HUGGING_FACE_HUB_TOKEN=<secret>" \
    -p 23333:23333 \
```

(continues on next page)

(continued from previous page)

```
    --ipc=host \
    openmmlab/lmdeploy:latest \
    lmdeploy serve api_server allenai/Molmo-7B-D-0924
```

If you find the following logs, it means the service launches successfully.

```
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
INFO:     Started server process [2439]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on  http://0.0.0.0:23333  (Press CTRL+C to quit)
```

The arguments of `lmdeploy serve api_server` can be reviewed in detail by `lmdeploy serve api_server -h`.

More information about `api_server` as well as how to access the service can be found from *here*

1.14 AWQ/GPTQ

LMDeploy TurboMind engine supports the inference of 4bit quantized models that are quantized both by AWQ and GPTQ, but its quantization module only supports the AWQ quantization algorithm.

The following NVIDIA GPUs are available for AWQ/GPTQ INT4 inference:

- V100(sm70): V100
- Turing(sm75): 20 series, T4
- Ampere(sm80,sm86): 30 series, A10, A16, A30, A100
- Ada Lovelace(sm89): 40 series

Before proceeding with the quantization and inference, please ensure that lmdeploy is installed by following the *installation guide*

The remainder of this article is structured into the following sections:

- *Quantization*
- *Evaluation*
- *Inference*
- *Service*
- *Performance*

1.14.1 Quantization

A single command execution is all it takes to quantize the model. The resulting quantized weights are then stored in the $WORK_DIR directory.

```
export HF_MODEL=internlm/internlm2_5-7b-chat
export WORK_DIR=internlm/internlm2_5-7b-chat-4bit

lmdeploy lite auto_awq \
   $HF_MODEL \
  --calib-dataset 'ptb' \
  --calib-samples 128 \
  --calib-seqlen 2048 \
  --w-bits 4 \
  --w-group-size 128 \
  --batch-size 1 \
  --work-dir $WORK_DIR
```

Typically, the above command doesn't require filling in optional parameters, as the defaults usually suffice. For instance, when quantizing the internlm/internlm2_5-7b-chat model, the command can be condensed as:

```
lmdeploy lite auto_awq internlm/internlm2_5-7b-chat --work-dir internlm2_5-7b-chat-4bit
```

Note:

- We recommend that you specify the –work-dir parameter, including the model name as demonstrated in the example above. This facilitates LMDeploy in fuzzy matching the –work-dir with an appropriate built-in chat template. Otherwise, you will have to designate the chat template during inference.
- If the quantized model's accuracy is compromised, it is recommended to enable –search-scale for re-quantization and increase the –batch-size, for example, to 8. When search_scale is enabled, the quantization process will take more time. The –batch-size affects the amount of memory used, which can be adjusted according to actual conditions as needed.

Upon completing quantization, you can engage with the model efficiently using a variety of handy tools. For example, you can initiate a conversation with it via the command line:

```
lmdeploy chat ./internlm2_5-7b-chat-4bit --model-format awq
```

Alternatively, you can start the gradio server and interact with the model through the web at `http://{ip_addr}:{port`

```
lmdeploy serve gradio ./internlm2_5-7b-chat-4bit --server_name {ip_addr} --server_port
↪{port} --model-format awq
```

1.14.2 Evaluation

Please refer to OpenCompass about model evaluation with LMDeploy. Here is the guide

1.14.3 Inference

Trying the following codes, you can perform the batched offline inference with the quantized model:

```
from lmdeploy import pipeline, TurbomindEngineConfig
engine_config = TurbomindEngineConfig(model_format='awq')
pipe = pipeline("./internlm2_5-7b-chat-4bit", backend_config=engine_config)
response = pipe(["Hi, pls intro yourself", "Shanghai is"])
print(response)
```

For more information about the pipeline parameters, please refer to *here*.

In addition to performing inference with the quantized model on localhost, LMDeploy can also execute inference for the 4bit quantized model derived from AWQ algorithm available on Huggingface Hub, such as models from the lmdeploy space and TheBloke space

```
# inference with models from lmdeploy space
from lmdeploy import pipeline, TurbomindEngineConfig
pipe = pipeline("lmdeploy/llama2-chat-70b-4bit",
                backend_config=TurbomindEngineConfig(model_format='awq', tp=4))
response = pipe(["Hi, pls intro yourself", "Shanghai is"])
print(response)

# inference with models from thebloke space
from lmdeploy import pipeline, TurbomindEngineConfig, ChatTemplateConfig
pipe = pipeline("TheBloke/LLaMA2-13B-Tiefighter-AWQ",
                backend_config=TurbomindEngineConfig(model_format='awq'),
                chat_template_config=ChatTemplateConfig(model_name='llama2')
                )
response = pipe(["Hi, pls intro yourself", "Shanghai is"])
print(response)
```

1.14.4 Service

LMDeploy's `api_server` enables models to be easily packed into services with a single command. The provided RESTful APIs are compatible with OpenAI's interfaces. Below are an example of service startup:

```
lmdeploy serve api_server ./internlm2_5-7b-chat-4bit --backend turbomind --model-format␣
↪awq
```

The default port of `api_server` is `23333`. After the server is launched, you can communicate with server on terminal through `api_client`:

```
lmdeploy serve api_client http://0.0.0.0:23333
```

You can overview and try out `api_server` APIs online by swagger UI at `http://0.0.0.0:23333`, or you can also read the API specification from *here*.

1.14.5 Performance

We benchmarked the Llama-2-7B-chat and Llama-2-13B-chat models with 4-bit quantization on NVIDIA GeForce RTX 4090 using profile_generation.py. And we measure the token generation throughput (tokens/s) by setting a single prompt token and generating 512 tokens. All the results are measured for single batch inference.

model	llm-awq	mlc-llm	turbomind
Llama-2-7B-chat	112.9	159.4	206.4
Llama-2-13B-chat	N/A	90.7	115.8

1.14.6 FAQs

1. Out of Memory error during quantization due to insufficient GPU memory: This can be addressed by reducing the parameter `--calib-seqlen`, increasing the parameter `--calib-samples`, and set `--batch-size` to 1.

1.15 SmoothQuant

LMDeploy provides functions for quantization and inference of large language models using 8-bit integers(INT8). For GPUs such as Nvidia H100, lmdeploy also supports 8-bit floating point(FP8).

And the following NVIDIA GPUs are available for INT8/FP8 inference respectively:

- INT8
  - V100(sm70): V100
  - Turing(sm75): 20 series, T4
  - Ampere(sm80,sm86): 30 series, A10, A16, A30, A100
  - Ada Lovelace(sm89): 40 series
  - Hopper(sm90): H100
- FP8
  - Ada Lovelace(sm89): 40 series
  - Hopper(sm90): H100

First of all, run the following command to install lmdeploy:

```
pip install lmdeploy[all]
```

1.15.1 8-bit Weight Quantization

Performing 8-bit weight quantization involves three steps:

1. **Smooth Weights**: Start by smoothing the weights of the Language Model (LLM). This process makes the weights more amenable to quantizing.
2. **Replace Modules**: Locate DecoderLayers and replace the modules RSMNorm and nn.Linear with QRSMNorm and QLinear modules respectively. These 'Q' modules are available in the lmdeploy/pytorch/models/q_modules.py file.
3. **Save the Quantized Model**: Once you've made the necessary replacements, save the new quantized model.

lmdeploy provides `lmdeploy lite smooth_quant` command to accomplish all three tasks detailed above. Note that the argument `--quant-dtype` is used to determine if you are doing int8 or fp8 weight quantization. To get more info about usage of the cli, run `lmdeploy lite smooth_quant --help`

Here are two examples:

- int8

```
lmdeploy lite smooth_quant internlm/internlm2_5-7b-chat --work-dir ./internlm2_5-7b-
↪chat-int8 --quant-dtype int8
```

- fp8

```
lmdeploy lite smooth_quant internlm/internlm2_5-7b-chat --work-dir ./internlm2_5-7b-
↪chat-fp8 --quant-dtype fp8
```

1.15.2 Inference

Trying the following codes, you can perform the batched offline inference with the quantized model:

```
from lmdeploy import pipeline, PytorchEngineConfig

engine_config = PytorchEngineConfig(tp=1)
pipe = pipeline("internlm2_5-7b-chat-int8", backend_config=engine_config)
response = pipe(["Hi, pls intro yourself", "Shanghai is"])
print(response)
```

1.15.3 Service

LMDeploy's `api_server` enables models to be easily packed into services with a single command. The provided RESTful APIs are compatible with OpenAI's interfaces. Below are an example of service startup:

```
lmdeploy serve api_server ./internlm2_5-7b-chat-int8 --backend pytorch
```

The default port of `api_server` is `23333`. After the server is launched, you can communicate with server on terminal through `api_client`:

```
lmdeploy serve api_client http://0.0.0.0:23333
```

You can overview and try out `api_server` APIs online by swagger UI at `http://0.0.0.0:23333`, or you can also read the API specification from *here*.

1.16 INT4/INT8 KV Cache

Since v0.4.0, LMDeploy has supported **online** key-value (kv) cache quantization with int4 and int8 numerical precision, utilizing an asymmetric quantization method that is applied on a per-head, per-token basis. The original kv offline quantization method has been removed.

Intuitively, quantization is beneficial for increasing the number of kv block. Compared to fp16, the number of kv block for int4/int8 kv can be increased by 4 times and 2 times respectively. This means that under the same memory conditions, the system can support a significantly increased number of concurrent operations after kv quantization, thereby ultimately enhancing throughput.

However, quantization typically brings in some loss of model accuracy. We have used OpenCompass to evaluate the accuracy of several models after applying int4/int8 quantization. int8 kv keeps the accuracy while int4 kv has slight loss. The detailed results are presented in the *Evaluation* section. You can refer to the information and choose wisely based on your requirements.

LMDeploy inference with quantized kv supports the following NVIDIA GPU models:

- Volta architecture (sm70): V100
- Turing architecture (sm75): 20 series, T4
- Ampere architecture (sm80, sm86): 30 series, A10, A16, A30, A100
- Ada Lovelace architecture (sm89): 40 series
- Hopper architecture (sm90): H100, H200

In summary, LMDeploy kv quantization has the following advantages:

1. data-free online quantization
2. Supports all nvidia GPU models with Volta architecture (sm70) and above
3. KV int8 quantization has almost lossless accuracy, and KV int4 quantization accuracy is within an acceptable range
4. Efficient inference, with int8/int4 kv quantization applied to llama2-7b, RPS is improved by round 30% and 40% respectively compared to fp16

In the next section, we will take `internlm2-chat-7b` model as an example, introducing the usage of kv quantization and inference of lmdeploy. But before that, please ensure that lmdeploy is installed.

```
pip install lmdeploy
```

1.16.1 Usage

Applying kv quantization and inference via LMDeploy is quite straightforward. Simply set the `quant_policy` parameter.

LMDeploy specifies that `quant_policy=4` stands for 4-bit kv, whereas `quant_policy=8` indicates 8-bit kv.

Offline inference

```python
from lmdeploy import pipeline, TurbomindEngineConfig
engine_config = TurbomindEngineConfig(quant_policy=8)
pipe = pipeline("internlm/internlm2_5-7b-chat", backend_config=engine_config)
response = pipe(["Hi, pls intro yourself", "Shanghai is"])
print(response)
```

Serving

```
lmdeploy serve api_server internlm/internlm2_5-7b-chat --quant-policy 8
```

1.16.2 Evaluation

We apply kv quantization of LMDeploy to several LLM models and utilize OpenCompass to evaluate the inference accuracy. The results are shown in the table below:

-	-	-	llama2-7b-chat	-	-	internlm chat-7b	-	-	internlm2 chat-7b	-	-	qwen1. 7b-chat	-	-
datase	version	metric	kv fp16	kv int8	kv int4	kv fp16	kv int8	kv int4	kv fp16	kv int8	kv int4	fp16	kv int8	kv int4
ceval	-	naive_	28.42	27.96	27.58	60.45	60.88	60.28	78.06	77.87	77.05	70.56	70.49	68.62
mmlu	-	naive_	35.64	35.58	34.79	63.91	64	62.36	72.30	72.27	71.17	61.48	61.56	60.65
triviaqa	2121	score	56.09	56.13	53.71	58.73	58.7	58.18	65.09	64.87	63.28	44.62	44.77	44.04
gsm8k	1d7fe	accuracy	28.2	28.05	27.37	70.13	69.75	66.87	85.67	85.44	83.78	54.97	56.41	54.74
race-middl	9a54b	accuracy	41.57	41.78	41.23	88.93	88.93	88.93	92.76	92.83	92.55	87.33	87.26	86.28
race-high	9a54b	accuracy	39.65	39.77	40.77	85.33	85.31	84.62	90.51	90.42	90.42	82.53	82.59	82.02

For detailed evaluation methods, please refer to *this* guide. Remember to pass `quant_policy` to the inference engine in the config file.

1.16.3 Performance

model	kv type	test settings	RPS	v.s. kv fp16
llama2-chat-7b	fp16	tp1 / ratio 0.8 / bs 256 / prompts 10000	14.98	1.0
-	int8	tp1 / ratio 0.8 / bs 256 / prompts 10000	19.01	1.27
-	int4	tp1 / ratio 0.8 / bs 256 / prompts 10000	20.81	1.39
llama2-chat-13b	fp16	tp1 / ratio 0.9 / bs 128 / prompts 10000	8.55	1.0
-	int8	tp1 / ratio 0.9 / bs 256 / prompts 10000	10.96	1.28
-	int4	tp1 / ratio 0.9 / bs 256 / prompts 10000	11.91	1.39
internlm2-chat-7b	fp16	tp1 / ratio 0.8 / bs 256 / prompts 10000	24.13	1.0
-	int8	tp1 / ratio 0.8 / bs 256 / prompts 10000	25.28	1.05
-	int4	tp1 / ratio 0.8 / bs 256 / prompts 10000	25.80	1.07

The performance data is obtained by `benchmark/profile_throughput.py`

1.17 Profile Token Latency and Throughput

We profile the latency and throughput of generated tokens with fixed batch size and fixed input/output token.

The profiling script is `profile_generation.py`. Before running it, please install the lmdeploy precompiled package and download the profiling script:

```
pip install lmdeploy
git clone --depth=1 https://github.com/InternLM/lmdeploy
```

1.17.1 Metrics

LMDeploy records test results like first token latency, token throughput (tokens/s), percentile data of each token's latency (P50, P75, P95, P99), GPU mem, etc.

`first_token_latency` is only reported in the case of streaming inference.

The formula for calculating `throughput` is:

$$TokenThroughput = Number\ of\ generated\ tokens/TotalTime$$

Total time includes prefill time.

During the test process, all graphics cards on the node should not run any other programs, otherwise the statistics of GPU mem would be inaccurate.

1.17.2 Profile

In this section, we take internlm/internlm-7b as an example to show how to profile the inference engines of LMDeploy.

Profile turbomind engine

```
cd lmdeploy/benchmark
python3 profile_generation.py internlm/internlm-7b
```

Profile pytorch engine

```
cd lmdeploy/benchmark
python3 profile_generation.py internlm/internlm-7b --backend pytorch
```

For detailed argument specification of `profile_generation.py`, such as batch size, input and output token number an so on, please run the help command `python3 profile_generation.py -h`.

1.18 Profile Request Throughput

In the applications, the length of the user's input prompt and the size of generated tokens are dynamic. The static inference performance is insufficient to reflect the inference engine's ability to handle the dynamic characteristics.

Therefore, it is necessary to use real dialogue data to evaluate the dynamic inference capabilities of the inference engine. This article will introduce how to test the dynamic inference performance of LMDeploy on localhost.

The profiling script is `profile_throughput.py`. Before running it, please install the lmdeploy precompiled package, download the profiling script and the test dataset:

```
pip install lmdeploy
git clone --depth=1 https://github.com/InternLM/lmdeploy
cd lmdeploy/benchmark
wget https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered/resolve/
↪main/ShareGPT_V3_unfiltered_cleaned_split.json
```

1.18.1 Metrics

LMDeploy records the performance metrics like first token latency, token throughput (tokens/s) and request throughput (RPM)

`first_token_latency` is only reported in the case of streaming inference.

The formula for calculating `token throughput` is:

$$TokenThroughput = Number\ of\ generated\ tokens/TotalTime$$

And the formula for calculating `request throughput` is:

$$RPM(request\ per\ minute) = Number\ of\ prompts/TotalTime * 60$$

Total time includes prefill time.

1.18.2 Profile

In this section, we take internlm/internlm-7b as an example to show how to profile the inference engines of LMDeploy.

Profile turbomind engine

```
python3 profile_throughput.py ./ShareGPT_V3_unfiltered_cleaned_split.json internlm/
↪internlm-7b
```

Profile pytorch engine

```
python3 profile_throughput.py ./ShareGPT_V3_unfiltered_cleaned_split.json internlm/
↪internlm-7b  --backend pytorch
```

For detailed argument specification of `profile_throughput.py`, such as request concurrency, sampling parameters, k/v cache memory percentage an so on, please run the help command `python3 profile_throughput.py -h`.

1.19 Profile API Server

Before benchmarking the api_server, please install the lmdeploy precompiled package and download the script and the test dataset:

```
pip install lmdeploy
git clone --depth=1 https://github.com/InternLM/lmdeploy
cd lmdeploy/benchmark
```

Launch the server first (you may refer *here* for guide) and run the following command:

```
python3 benchmark/profile_restful_api.py --backend lmdeploy --num-prompts 5000 --dataset-
↪path ShareGPT_V3_unfiltered_cleaned_split.json
```

For detailed argument specification of `profile_restful_api.py`, please run the help command `python3 benchmark/profile_restful_api.py -h`.

1.20 Evaluate LLMs with OpenCompass

The LLMs accelerated by lmdeploy can be evaluated with OpenCompass.

1.20.1 Setup

In this part, we are going to setup the environment for evaluation.

Install lmdeploy

Please follow the *installation guide* to install lmdeploy.

Install OpenCompass

Install OpenCompass from source. Refer to installation for more information.

```
git clone https://github.com/open-compass/opencompass.git
cd opencompass
pip install -e .
```

At present, you can check the Quick Start to get to know the basic usage of OpenCompass.

Download datasets

Download the core datasets

```
# Run in the OpenCompass directory
cd opencompass
wget https://github.com/open-compass/opencompass/releases/download/0.1.8.rc1/
↪OpenCompassData-core-20231110.zip
unzip OpenCompassData-core-20231110.zip
```

1.20.2 Prepare Evaluation Config

OpenCompass uses the configuration files as the OpenMMLab style. One can define a python config and start evaluating at ease. OpenCompass has supported the evaluation for lmdeploy's TurboMind engine using python API.

Dataset Config

In the home directory of OpenCompass, we are writing the config file `$OPENCOMPASS_DIR/configs/eval_lmdeploy.py`. We select multiple predefined datasets and import them from OpenCompass base dataset configs as `datasets`.

```
from mmengine.config import read_base


with read_base():
    # choose a list of datasets
    from .datasets.mmlu.mmlu_gen_a484b3 import mmlu_datasets
    from .datasets.ceval.ceval_gen_5f30c7 import ceval_datasets
    from .datasets.SuperGLUE_WiC.SuperGLUE_WiC_gen_d06864 import WiC_datasets
    from .datasets.SuperGLUE_WSC.SuperGLUE_WSC_gen_7902a7 import WSC_datasets
    from .datasets.triviaqa.triviaqa_gen_2121ce import triviaqa_datasets
    from .datasets.gsm8k.gsm8k_gen_1d7fe4 import gsm8k_datasets
    from .datasets.race.race_gen_69ee4f import race_datasets
    from .datasets.crowspairs.crowspairs_gen_381af0 import crowspairs_datasets
    # and output the results in a chosen format
    from .summarizers.medium import summarizer

datasets = sum((v for k, v in locals().items() if k.endswith('_datasets')), [])
```

Model Config

This part shows how to setup model config for LLMs. Let's check some examples:

internlm-20b

internlm-chat-20b

```python
from opencompass.models.turbomind import TurboMindModel

internlm_20b = dict(
        type=TurboMindModel,
        abbr='internlm-20b-turbomind',
        path="internlm/internlm-20b",  # this path should be same as in huggingface
        engine_config=dict(session_len=2048,
                           max_batch_size=8,
                           rope_scaling_factor=1.0),
        gen_config=dict(top_k=1, top_p=0.8,
                        temperature=1.0,
                        max_new_tokens=100),
        max_out_len=100,
        max_seq_len=2048,
        batch_size=8,
        concurrency=8,
        run_cfg=dict(num_gpus=1, num_procs=1),
    )

models = [internlm_20b]
```

For Chat models, you have to pass `meta_template` for chat models. Different Chat models may have different `meta_template` and it's important to keep it the same as in training settings. You can read meta_template for more information.

```python
from opencompass.models.turbomind import TurboMindModel

internlm_meta_template = dict(round=[
    dict(role='HUMAN', begin='<|User|>:', end='\n'),
    dict(role='BOT', begin='<|Bot|>:', end='<eoa>\n', generate=True),
],
                              eos_token_id=103028)

internlm_chat_20b = dict(
    type=TurboMindModel,
    abbr='internlm-chat-20b-turbomind',
    path='internlm/internlm-chat-20b',
    engine_config=dict(session_len=2048,
                       max_batch_size=8,
                       rope_scaling_factor=1.0),
    gen_config=dict(top_k=1,
                    top_p=0.8,
                    temperature=1.0,
                    max_new_tokens=100),
    max_out_len=100,
    max_seq_len=2048,
    batch_size=8,
```

(continues on next page)

(continued from previous page)

```
    concurrency=8,
    meta_template=internlm_meta_template,
    run_cfg=dict(num_gpus=1, num_procs=1),
    end_str='<eoa>'
)

models = [internlm_chat_20b]
```

Note

- If you want to pass more arguments for `engine_configgen_config` in the evaluation config file, please refer to TurbomindEngineConfig and EngineGenerationConfig

1.20.3 Execute Evaluation Task

After defining the evaluation config, we can run the following command to start evaluating models. You can check Execution Task for more arguments of `run.py`.

```
# in the root directory of opencompass
python3 run.py configs/eval_lmdeploy.py --work-dir ./workdir
```

1.21 Architecture of TurboMind

TurboMind is an inference engine that supports high throughput inference for conversational LLMs. It's based on NVIDIA's FasterTransformer. Major features of TurboMind include an efficient LLaMa implementation, the persistent batch inference model and an extendable KV cache manager.

1.21.1 High level overview of TurboMind

```
  +--------------------+
  |        API         |
  +--------------------+
          |    ^
  request |    | stream callback
          v    |
  +--------------------+   fetch   +-------------------+
  |  Persistent Batch  | <-------> |  KV Cache Manager |
  +--------------------+   update  +-------------------+
             ^
             |
             v
+------------------------+
|  LLaMA implementation  |
+------------------------+
| FT kernels & utilities |
+------------------------+
```

1.21.2 Persistent Batch

You may recognize this feature as "continuous batching" in other repos. But during the concurrent development of the feature, we modeled the inference of a conversational LLM as a persistently running batch whose lifetime spans the entire serving process, hence the name "persistent batch". To put it simply

- The persistent batch as N pre-configured batch slots.
- Requests join the batch when there are free slots available. A batch slot is released and can be reused once the generation of the requested tokens is finished.
- **On cache-hits (see below), history tokens don't need to be decoded in every round of a conversation; generation of response tokens will start instantly.**
- The batch grows or shrinks automatically to minimize unnecessary computations.

1.21.3 KV Cache Manager

The KV cache manager of TurboMind is a memory-pool-liked object that also implements LRU policy so that it can be viewed as a form of **cache of KV caches**. It works in the following way

- All device memory required for KV cache is allocated by the manager. A fixed number of slots is pre-configured to match the memory size of the system. Each slot corresponds to the memory required by the KV cache of a single sequence. Allocation chunk-size can be configure to implement pre-allocate/on-demand style allocation policy (or something in-between).
- When space for the KV cache of a new sequence is requested but no free slots left in the pool, the least recently used sequence is evicted from the cache and its device memory is directly reused by the new sequence. However, this is not the end of the story.
- Fetching sequence currently resides in one of the slots resembles a *cache-hit*, the history KV cache is returned directly and no context decoding is needed.
- Victim (evicted) sequences are not erased entirely but converted to its most compact form, i.e. token IDs. When the same sequence id is fetched later (*cache-miss*) the token IDs will be decoded by FMHA backed context decoder and converted back to KV cache.
- The eviction and conversion are handled automatically inside TurboMind and thus transparent to the users. **From the user's aspect, system that use TurboMind has access to infinite device memory.**

1.21.4 LLaMa implementation

Our implementation of the LLaMa family models is modified from Gpt-NeoX model in FasterTransformer. In addition to basic refactoring and modifications to support the LLaMa family, we made some improvements to enable high performance inference of conversational models, most importantly:

- To support fast context decoding in multi-round conversations. We replaced the attention implementation in context decoder with a cutlass-based FMHA implementation that supports mismatched Q/K lengths.
- We introduced indirect buffer pointers in both context FMHA and generation FMHA to support the discontinuity in KV cache within the batch.
- To support concurrent inference with persistent batch, new synchronization mechanism was designed to orchestrate the worker threads running in tensor parallel mode.
- To maximize the throughput, we implement INT8 KV cache support to increase the max batch size. It's effective because in real-world serving scenarios, KV cache costs more memory and consumes more memory bandwidth than weights or other activations.

- We resolved an NCCL hang issue when running multiple model instances in TP mode within a single process, NCCL APIs are now guarded by host-side synchronization barriers.

1.21.5 API

TurboMind supports a Python API that enables streaming output and tensor parallel mode.

1.21.6 Difference between FasterTransformer and TurboMind

Apart of the features described above, there are still many minor differences that we don't cover in this document. Notably, many capabilities of FT are dropped in TurboMind because of the difference in objectives (e.g. prefix prompt, beam search, context embedding, sparse GEMM, GPT/T5/other model families, etc)

1.21.7 FAQ

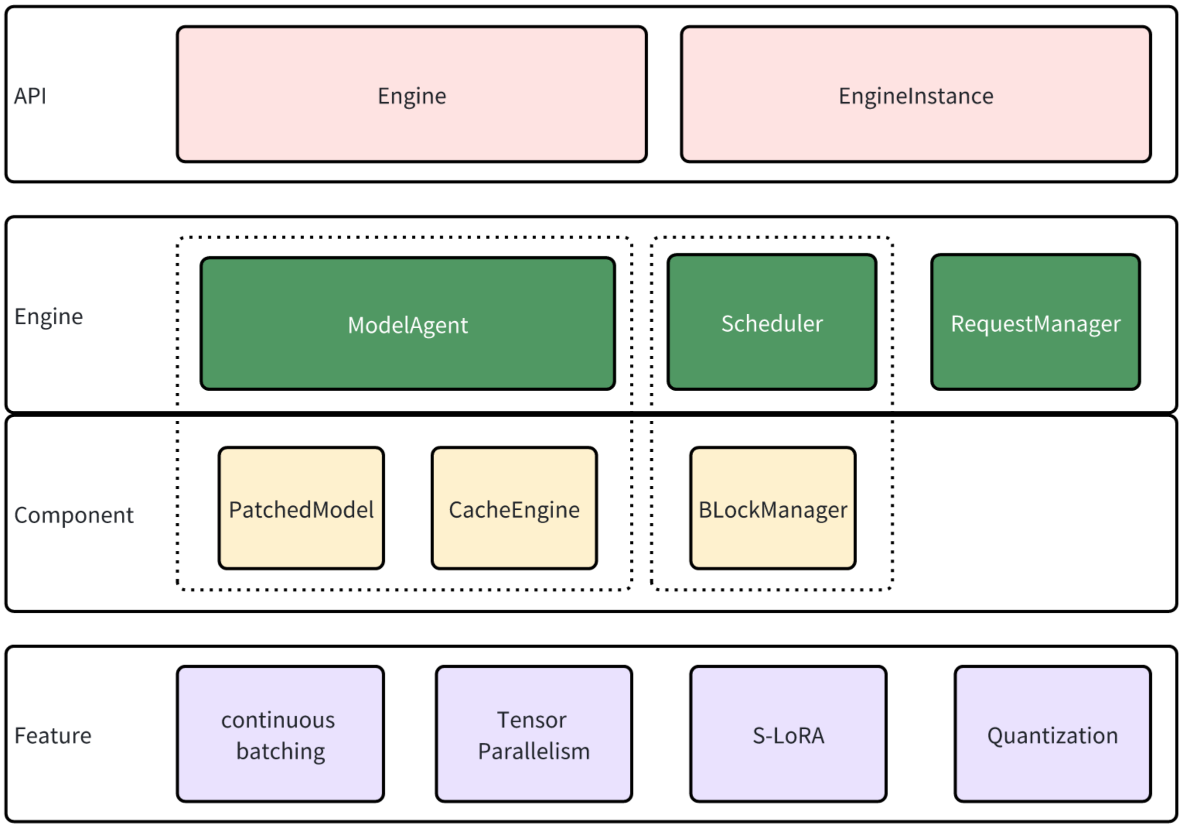
Supporting Huggingface models

For historical reasons, TurboMind's weight layout is based on the original LLaMa implementation (differ only by a transpose). The implementation in huggingface transformers uses a different layout for `W_q` and `W_k` which is handled in deploy.py.

1.22 Architecture of lmdeploy.pytorch

`lmdeploy.pytorch` is an inference engine in LMDeploy that offers a developer-friendly framework to users interested in deploying their own models and developing new features.

1.22.1 Design



1.22.2 API

`lmdeploy.pytorch` shares service interfaces with `Turbomind`, and the inference service is implemented by `Engine` and `EngineInstance`.

`EngineInstance` acts as the sender of inference requests, encapsulating and sending requests to the `Engine` to achieve streaming inference. The inference interface of `EngineInstance` is thread-safe, allowing instances in different threads to initiate requests simultaneously. The `Engine` will automatically perform batch processing based on the current system resources.

Engine is the request receiver and executor. It contain modules:

- `ModelAgent` serves as a wrapper for the model, handling tasks such as loading model/adapters, managing the cache, and implementing tensor parallelism.
- The `Scheduler` functions as the sequence manager, determining the sequences and adapters to participate in the current step, and subsequently allocating resources for them.
- `RequestManager` is tasked with sending and receiving requests. acting as the bridge between the `Engine` and `EngineInstance`.

1.22.3 Engine

The Engine responses to requests in a sub-thread, following this looping sequence:

1. Get new requests through `RequestManager`. These requests are cached for now.
2. The `Scheduler` performs scheduling, deciding which cached requests should be processed and allocating resources for them.
3. `ModelAgent` swaps the caches according to the information provided by the Scheduler, then performs inference with the patched model.
4. The `Scheduler` updates the status of requests based to the inference results from `ModelAgent`.
5. `RequestManager` responds to the sender (`EngineInstance`), and the process return to step 1.

Now, Let's delve deeper into the modules that participate in these steps.

Scheduler

In LLM inference, caching history key and value states is a common practice to prevent redundant computation. However, as history lengths vary in a batch of sequences, we need to pad the caches to enable batching inference. Unfortunately, this padding can lead to significant memory wastage, limiting the transformer's performance.

vLLM employs a paging-based strategy, allocating caches in page blocks to minimize extra memory usage. Our Scheduler module in the Engine shares a similar design, allocating resources based on sequence length in blocks and evicting unused blocks to support larger batching and longer session lengths.

Additionally, we support S-LoRA, which enables the use of multiple LoRA adapters on limited memory.

ModelAgent

`lmdeploy.pytorch` supports Tensor Parallelism, which leads to complex model initialization, cache allocation, and weight partitioning. ModelAgent is designed to abstract these complexities, allowing the Engine to focus solely on maintaining the pipeline.

ModelAgent consists of two components:

1. `**patched_model**: : This is the transformer model after patching. In comparison to the original model, the patched model incorporates additional features such as Tensor Parallelism, quantization, and high-performance kernels.
2. **cache_engine**: This component manages the caches. It receives commands from the Scheduler and performs host-device page swaps. Only GPU blocks are utilized for caching key/value pairs and adapters.

1.22.4 Features

`lmdeploy.pytorch` supports new features including:

- **Continuous Batching**: As the sequence length in a batch may vary, padding is often necessary for batching inference. However, large padding can lead to additional memory usage and unnecessary computation. To address this, we employ continuous batching, where all sequences are concatenated into a single long sequence to avoid padding.
- **Tensor Parallelism**: The GPU memory usage of LLM might exceed the capacity of a single GPU. Tensor parallelism is utilized to accommodate such models on multiple devices. Each device handles parts of the model simultaneously, and the results are gathered to ensure correctness.

- **S-LoRA**: LoRA adapters can be used to train LLM on devices with limited memory. While it's common practice to merge adapters into the model weights before deployment, loading multiple adapters in this way can consume a significant amount of memory. We support S-LoRA, where adapters are paged and swapped in when necessary. Special kernels are developed to support inference with unmerged adapters, enabling the loading of various adapters efficiently.
- **Quantization**: Model quantization involves performing computations with low precision. `lmdeploy.pytorch` supports w8a8 quantization. For more details, refer to *w8a8*.

1.23 lmdeploy.pytorch New Model Support

lmdeploy.pytorch is designed to simplify the support for new models and the development of prototypes. Users can adapt new models according to their own needs.

1.23.1 Model Support

Configuration Loading (Optional)

lmdeploy.pytorch initializes the engine based on the model's config file. If the parameter naming of the model to be integrated differs from common models in transformers, parsing errors may occur. A custom ConfigBuilder can be added to parse the configuration.

```python
# lmdeploy/pytorch/configurations/gemma.py

from lmdeploy.pytorch.config import ModelConfig

from .builder import AutoModelConfigBuilder


class GemmaModelConfigBuilder(AutoModelConfigBuilder):

    @classmethod
    def condition(cls, hf_config):
        # Check if hf_config is suitable for this builder
        return hf_config.model_type in ['gemma', 'gemma2']

    @classmethod
    def build(cls, hf_config, model_path: str = None):
        # Use the hf_config loaded by transformers
        # Construct the ModelConfig for the pytorch engine
        return ModelConfig(hidden_size=hf_config.hidden_size,
                           num_layers=hf_config.num_hidden_layers,
                           num_attention_heads=hf_config.num_attention_heads,
                           num_key_value_heads=hf_config.num_key_value_heads,
                           bos_token_id=hf_config.bos_token_id,
                           eos_token_id=hf_config.eos_token_id,
                           head_dim=hf_config.head_dim,
                           vocab_size=hf_config.vocab_size)
```

The `lmdeploy.pytorch.check_env.check_model` function can be used to verify if the configuration can be parsed correctly.

Implementing the Model

After ensuring that the configuration can be parsed correctly, you can start implementing the model logic. Taking the implementation of llama as an example, we need to create the model using the configuration file from transformers.

```python
class LlamaForCausalLM(nn.Module):

    # Constructor, builds the model with the given config
    # ctx_mgr is the context manager, which can be used to pass engine configurations or␣
↪additional parameters
    def __init__(self,
                 config: LlamaConfig,
                 ctx_mgr: StepContextManager,
                 dtype: torch.dtype = None,
                 device: torch.device = None):
        super().__init__()
        self.config = config
        self.ctx_mgr = ctx_mgr
        # build LLamaModel
        self.model = LlamaModel(config, dtype=dtype, device=device)
        # build lm_head
        self.lm_head = build_rowwise_linear(config.hidden_size,
                                            config.vocab_size,
                                            bias=False,
                                            dtype=dtype,
                                            device=device)

    # Model inference function
    # It is recommended to use the same parameters as below
    def forward(
        self,
        input_ids: torch.Tensor,
        position_ids: torch.Tensor,
        past_key_values: List[List[torch.Tensor]],
        attn_metadata: Any = None,
        inputs_embeds: torch.Tensor = None,
        **kwargs,
    ):
        hidden_states = self.model(
            input_ids=input_ids,
            position_ids=position_ids,
            past_key_values=past_key_values,
            attn_metadata=attn_metadata,
            inputs_embeds=inputs_embeds,
        )

        logits = self.lm_head(hidden_states)
        logits = logits.float()
        return logits
```

In addition to these, the following content needs to be added:

```python
class LlamaForCausalLM(nn.Module):
```

(continues on next page)

(continued from previous page)

```
    ...

    # Indicates whether the model supports cudagraph
    # Can be a callable object, receiving forward inputs
    # Dynamically determines if cudagraph is supported
    support_cuda_graph = True

    # Builds model inputs
    # Returns a dictionary, the keys of which must be inputs to forward
    def prepare_inputs_for_generation(
        self,
        past_key_values: List[List[torch.Tensor]],
        inputs_embeds: Optional[torch.Tensor] = None,
        context: StepContext = None,
    ):
        ...

    # Loads weights
    # The model's inputs are key-value pairs of the state dict
    def load_weights(self, weights: Iterable[Tuple[str, torch.Tensor]]):
        ...
```

We have encapsulated many fused operators to simplify the model construction. These operators better support various functions such as tensor parallelism and quantization. We encourage developers to use these ops as much as possible.

```
# Using predefined build_merged_colwise_linear, SiluAndMul, build_rowwise_linear
# Helps us build the model faster and without worrying about tensor concurrency,␣
↪quantization, etc.
class LlamaMLP(nn.Module):

    def __init__(self,
                 config: LlamaConfig,
                 dtype: torch.dtype = None,
                 device: torch.device = None):
        super().__init__()
        quantization_config = getattr(config, 'quantization_config', None)
        # gate up
        self.gate_up_proj = build_merged_colwise_linear(
            config.hidden_size,
            [config.intermediate_size, config.intermediate_size],
            bias=config.mlp_bias,
            dtype=dtype,
            device=device,
            quant_config=quantization_config,
            is_tp=True,
        )

        # silu and mul
        self.act_fn = SiluAndMul(inplace=True)

        # down
        self.down_proj = build_rowwise_linear(config.intermediate_size,
```

(continues on next page)

(continued from previous page)

```
                                             config.hidden_size,
                                             bias=config.mlp_bias,
                                             quant_config=quantization_config,
                                             dtype=dtype,
                                             device=device,
                                             is_tp=True)

    def forward(self, x):
        """forward."""
        gate_up = self.gate_up_proj(x)
        act = self.act_fn(gate_up)
        return self.down_proj(act)
```

Model Registration

To ensure that the developed model implementation can be used normally, we also need to register the model in `lmdeploy/pytorch/models/module_map.py`

```
MODULE_MAP.update({
    'LlamaForCausalLM':
    f'{LMDEPLOY_PYTORCH_MODEL_PATH}.llama.LlamaForCausalLM',
})
```

If you do not wish to modify the model source code, you can also pass a custom module map from the outside, making it easier to integrate into other projects.

```
from lmdeploy import PytorchEngineConfig, pipeline

backend_config = PytorchEngineConfig(custom_module_map='/path/to/custom/module_map.py')
generator = pipeline(model_path, backend_config=backend_config)
```

1.24 Context length extrapolation

Long text extrapolation refers to the ability of LLM to handle data longer than the training text during inference. TurboMind engine now support LlamaDynamicNTKScalingRotaryEmbedding and the implementation is consistent with huggingface.

1.24.1 Usage

You can enable the context length extrapolation abality by modifying the TurbomindEngineConfig. Edit the `session_len` to the expected length and change `rope_scaling_factor` to a number no less than 1.0.

Take `internlm2_5-7b-chat-1m` as an example, which supports a context length of up to **1 million tokens**:

```
from lmdeploy import pipeline, GenerationConfig, TurbomindEngineConfig

backend_config = TurbomindEngineConfig(
        rope_scaling_factor=2.5,
        session_len=1000000,
```

(continues on next page)

(continued from previous page)

```
        max_batch_size=1,
        cache_max_entry_count=0.7,
        tp=4)
pipe = pipeline('internlm/internlm2_5-7b-chat-1m', backend_config=backend_config)
prompt = 'Use a long prompt to replace this sentence'
gen_config = GenerationConfig(top_p=0.8,
                              top_k=40,
                              temperature=0.8,
                              max_new_tokens=1024)
response = pipe(prompt, gen_config=gen_config)
print(response)
```

1.24.2 Evaluation

We use several methods to evaluate the long-context-length inference ability of LMDeploy, including *passkey retrieval*, *needle in a haystack* and computing *perplexity*

Passkey Retrieval

You can try the following code to test how many times LMDeploy can retrieval the special key.

```
import numpy as np
from lmdeploy import pipeline
from lmdeploy import TurbomindEngineConfig
import time

session_len = 1000000
backend_config = TurbomindEngineConfig(
        rope_scaling_factor=2.5,
        session_len=session_len,
        max_batch_size=1,
        cache_max_entry_count=0.7,
        tp=4)
pipe = pipeline('internlm/internlm2_5-7b-chat-1m', backend_config=backend_config)


def passkey_retrieval(session_len, n_round=5):
    # create long context input
    tok = pipe.tokenizer
    task_description = 'There is an important info hidden inside a lot of irrelevant␣
↪text. Find it and memorize them. I will quiz you about the important information there.
↪'
    garbage = 'The grass is green. The sky is blue. The sun is yellow. Here we go. There␣
↪and back again.'

    for _ in range(n_round):
        start = time.perf_counter()
        n_times = (session_len - 1000) // len(tok.encode(garbage))
        n_garbage_prefix = np.random.randint(0, n_times)
        n_garbage_suffix = n_times - n_garbage_prefix
```

(continues on next page)

(continued from previous page)

```
        garbage_prefix = ' '.join([garbage] * n_garbage_prefix)
        garbage_suffix = ' '.join([garbage] * n_garbage_suffix)
        pass_key = np.random.randint(1, 50000)
        information_line = f'The pass key is {pass_key}. Remember it. {pass_key} is the␣
↪pass key.'  # noqa: E501
        final_question = 'What is the pass key? The pass key is'
        lines = [
            task_description,
            garbage_prefix,
            information_line,
            garbage_suffix,
            final_question,
        ]

        # inference
        prompt = ' '.join(lines)
        response = pipe([prompt])
        print(pass_key, response)
        end = time.perf_counter()
        print(f'duration: {end - start} s')

passkey_retrieval(session_len, 5)
```

This test takes approximately 364 seconds per round when conducted on A100-80G GPUs

Needle In A Haystack

OpenCompass offers very useful tools to perform needle-in-a-haystack evaluation. For specific instructions, please refer to the guide.

Perplexity

The following codes demonstrate how to use LMDeploy to calculate perplexity.

```
from transformers import AutoTokenizer
from lmdeploy import TurbomindEngineConfig, pipeline
import numpy as np

# load model and tokenizer
model_repoid_or_path = 'internlm/internlm2_5-7b-chat-1m'
backend_config = TurbomindEngineConfig(
        rope_scaling_factor=2.5,
        session_len=1000000,
        max_batch_size=1,
        cache_max_entry_count=0.7,
        tp=4)
pipe = pipeline(model_repoid_or_path, backend_config=backend_config)
tokenizer = AutoTokenizer.from_pretrained(model_repoid_or_path, trust_remote_code=True)

# get perplexity
text = 'Use a long prompt to replace this sentence'
```

(continues on next page)

(continued from previous page)

```
input_ids = tokenizer.encode(text)
ppl = pipe.get_ppl(input_ids)[0]
print(ppl)
```

1.25 Customized chat template

The effect of the applied chat template can be observed by **setting log level** `INFO`.

LMDeploy supports two methods of adding chat templates:

- One approach is to utilize an existing conversation template by directly configuring a JSON file like the following.

  ```
  {
      "model_name": "your awesome chat template name",
      "system": "<|im_start|>system\n",
      "meta_instruction": "You are a robot developed by LMDeploy.",
      "eosys": "<|im_end|>\n",
      "user": "<|im_start|>user\n",
      "eoh": "<|im_end|>\n",
      "assistant": "<|im_start|>assistant\n",
      "eoa": "<|im_end|>",
      "separator": "\n",
      "capability": "chat",
      "stop_words": ["<|im_end|>"]
  }
  ```

  `model_name` is a required field and can be either the name of an LMDeploy built-in chat template (which can be viewed through `lmdeploy list`), or a new name. Other fields are optional.

  1. When `model_name` is the name of a built-in chat template, the non-null fields in the JSON file will override the corresponding attributes of the original chat template.
  2. However, when `model_name` is a new name, it will register `BaseChatTemplate` directly as a new chat template. The specific definition can be referred to BaseChatTemplate.

  The new chat template would be like this:

  ```
  {system}{meta_instruction}{eosys}{user}{user_content}{eoh}{assistant}{assistant_
  ↪content}{eoa}{separator}{user}...
  ```

  When using the CLI tool, you can pass in a custom chat template with `--chat-template`, for example.

  ```
  lmdeploy serve api_server internlm/internlm2_5-7b-chat --chat-template ${JSON_FILE}
  ```

  You can also pass it in through the interface function, for example.

  ```
  from lmdeploy import ChatTemplateConfig, serve
  serve('internlm/internlm2_5-7b-chat',
        chat_template_config=ChatTemplateConfig.from_json('${JSON_FILE}'))
  ```

- Another approach is to customize a Python chat template class like the existing LMDeploy chat templates. It can be used directly after successful registration. The advantages are a high degree of customization and strong controllability. Below is an example of registering an LMDeploy chat template.

```python
from lmdeploy.model import MODELS, BaseChatTemplate


@MODELS.register_module(name='customized_model')
class CustomizedModel(BaseChatTemplate):
    """A customized chat template."""

    def __init__(self,
                 system='<|im_start|>system\n',
                 meta_instruction='You are a robot developed by LMDeploy.',
                 user='<|im_start|>user\n',
                 assistant='<|im_start|>assistant\n',
                 eosys='<|im_end|>\n',
                 eoh='<|im_end|>\n',
                 eoa='<|im_end|>',
                 separator='\n',
                 stop_words=['<|im_end|>', '<|action_end|>']):
        super().__init__(system=system,
                         meta_instruction=meta_instruction,
                         eosys=eosys,
                         user=user,
                         eoh=eoh,
                         assistant=assistant,
                         eoa=eoa,
                         separator=separator,
                         stop_words=stop_words)


from lmdeploy import ChatTemplateConfig, pipeline

messages = [{'role': 'user', 'content': 'who are you?'}]
pipe = pipeline('internlm/internlm2_5-7b-chat',
                chat_template_config=ChatTemplateConfig('customized_model'))
for response in pipe.stream_infer(messages):
    print(response.text, end='')
```

In this example, we register a LMDeploy chat template that sets the model to be created by LMDeploy, so when the user asks who the model is, the model will answer that it was created by LMDeploy.

1.26 How to debug Turbomind

Turbomind is implemented in C++, which is not as easy to debug as Python. This document provides basic methods for debugging Turbomind.

1.26.1 Prerequisite

First, complete the local compilation according to the commands in *Install from source*.

1.26.2 Configure Python debug environment

Since many large companies currently use Centos 7 for online production environments, we will use Centos 7 as an example to illustrate the process.

Obtain `glibc` and `python3` versions

```
rpm -qa | grep glibc
rpm -qa | grep python3
```

The result should be similar to this:

```
[username@hostname workdir]# rpm -qa | grep glibc
glibc-2.17-325.el7_9.x86_64
glibc-common-2.17-325.el7_9.x86_64
glibc-headers-2.17-325.el7_9.x86_64
glibc-devel-2.17-325.el7_9.x86_64

[username@hostname workdir]# rpm -qa | grep python3
python3-pip-9.0.3-8.el7.noarch
python3-rpm-macros-3-34.el7.noarch
python3-rpm-generators-6-2.el7.noarch
python3-setuptools-39.2.0-10.el7.noarch
python3-3.6.8-21.el7_9.x86_64
python3-devel-3.6.8-21.el7_9.x86_64
python3.6.4-sre-1.el6.x86_64
```

Based on the information above, we can see that the version of `glibc` is `2.17-325.el7_9.x86_64` and the version of `python3` is `3.6.8-21.el7_9.x86_64`.

Download and install `debuginfo` library

Download `glibc-debuginfo-common-2.17-325.el7.x86_64.rpm`, `glibc-debuginfo-2.17-325.el7.x86_64.rpm`, and `python3-debuginfo-3.6.8-21.el7.x86_64.rpm` from http://debuginfo.centos.org/7/x86_64.

```
rpm -ivh glibc-debuginfo-common-2.17-325.el7.x86_64.rpm
rpm -ivh glibc-debuginfo-2.17-325.el7.x86_64.rpm
rpm -ivh python3-debuginfo-3.6.8-21.el7.x86_64.rpm
```

Upgrade GDB

```
sudo yum install devtoolset-10 -y
echo "source scl_source enable devtoolset-10" >> ~/.bashrc
source ~/.bashrc
```

Verification

```
gdb python3
```

The output should be similar to this:

```
[username@hostname workdir]# gdb python3
GNU gdb (GDB) Red Hat Enterprise Linux 9.2-10.el7
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
   <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from python3...
(gdb)
```

If it shows `Reading symbols from python3`, the configuration has been successful.

For other operating systems, please refer to DebuggingWithGdb.

1.26.3 Set up symbolic links

After setting up symbolic links, there is no need to install it locally with `pip` every time.

```
# Change directory to lmdeploy, e.g.
cd /workdir/lmdeploy

# Since it has been built in the build directory
# Link the lib directory
cd lmdeploy && ln -s ../build/lib . && cd ..
# (Optional) Link compile_commands.json for clangd index
ln -s build/compile_commands.json .
```

1.26.4 Start debugging

```
# Use gdb to start the API server with Llama-2-13b-chat-hf, e.g.
gdb --args python3 -m lmdeploy serve api_server /workdir/Llama-2-13b-chat-hf

# Set directories in gdb
Reading symbols from python3...
(gdb) set directories /workdir/lmdeploy

# Set a breakpoint using the relative path, e.g.
(gdb) b src/turbomind/models/llama/BlockManager.cc:104

# When it shows
# ```
# No source file named src/turbomind/models/llama/BlockManager.cc.
# Make breakpoint pending on future shared library load? (y or [n])
# ```
# Just type `y` and press enter

# Run
(gdb) r

# (Optional) Use https://github.com/InternLM/lmdeploy/blob/main/benchmark/profile_
↪restful_api.py to send a request

python3 profile_restful_api.py --backend lmdeploy --dataset-path /workdir/ShareGPT_V3_
↪unfiltered_cleaned_split.json --num_prompts 1
```

1.26.5 Using GDB

Refer to GDB Execution Commands and happy debugging.

1.27 Structured output

Currently, only the Pytorch backend has this capability. Therefore, whether you are using the pipeline or the api_server, please specify the use of the Pytorch backend.

1.27.1 pipeline

```
from lmdeploy import pipeline
from lmdeploy.messages import GenerationConfig, PytorchEngineConfig

model = 'internlm/internlm2-chat-1_8b'
guide = {
    'type': 'object',
    'properties': {
        'name': {
            'type': 'string'
        },
```

(continues on next page)

(continued from previous page)

```
        'skills': {
            'type': 'array',
            'items': {
                'type': 'string',
                'maxLength': 10
            },
            'minItems': 3
        },
        'work history': {
            'type': 'array',
            'items': {
                'type': 'object',
                'properties': {
                    'company': {
                        'type': 'string'
                    },
                    'duration': {
                        'type': 'string'
                    }
                },
                'required': ['company']
            }
        }
    },
    'required': ['name', 'skills', 'work history']
}
pipe = pipeline(model, backend_config=PytorchEngineConfig(), log_level='INFO')
gen_config = GenerationConfig(
    response_format=dict(type='json_schema', json_schema=dict(name='test',␣
↪schema=guide)))
response = pipe(['Make a self introduction please.'], gen_config=gen_config)
print(response)
```

1.27.2 api_server

Firstly, start the api_server service for the InternLM2 model.

```
lmdeploy serve api_server internlm/internlm2-chat-1_8b --backend pytorch
```

The client can test using OpenAI's python package: The output result is a response in JSON format.

```
from openai import OpenAI
guide = {
    'type': 'object',
    'properties': {
        'name': {
            'type': 'string'
        },
        'skills': {
            'type': 'array',
            'items': {
```

(continues on next page)

(continued from previous page)

```
                'type': 'string',
                'maxLength': 10
            },
            'minItems': 3
        },
        'work history': {
            'type': 'array',
            'items': {
                'type': 'object',
                'properties': {
                    'company': {
                        'type': 'string'
                    },
                    'duration': {
                        'type': 'string'
                    }
                },
                'required': ['company']
            }
        }
    },
    'required': ['name', 'skills', 'work history']
}
response_format=dict(type='json_schema',  json_schema=dict(name='test',schema=guide))
messages = [{'role': 'user', 'content': 'Make a self-introduction please.'}]
client = OpenAI(api_key='YOUR_API_KEY', base_url='http://0.0.0.0:23333/v1')
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
    model=model_name,
    messages=messages,
    temperature=0.8,
    response_format=response_format,
    top_p=0.8)
print(response)
```

1.28 PyTorchEngine Multithread

We have removed `thread_safe` mode from PytorchEngine since PR2907. We encourage users to achieve high concurrency by using **service API** or **coroutines** whenever possible, for example:

```
import asyncio
from lmdeploy import pipeline, PytorchEngineConfig

event_loop = asyncio.new_event_loop()
asyncio.set_event_loop(event_loop)

model_path = 'Llama-3.2-1B-Instruct'
pipe = pipeline(model_path, backend_config=PytorchEngineConfig())

async def _gather_output():
```

(continues on next page)

(continued from previous page)

```
    tasks = [
        pipe.async_batch_infer('Hakuna Matata'),
        pipe.async_batch_infer('giraffes are heartless creatures'),
    ]
    return await asyncio.gather(*tasks)

output = asyncio.run(_gather_output())
print(output[0].text)
print(output[1].text)
```

If you do need multithreading, it would be easy to warp it like below:

```
import threading
from queue import Queue
import asyncio
from lmdeploy import pipeline, PytorchEngineConfig

model_path = 'Llama-3.2-1B-Instruct'


async def _batch_infer(inque: Queue, outque: Queue, pipe):
    while True:
        if inque.empty():
            await asyncio.sleep(0)
            continue

        input = inque.get_nowait()
        output = await pipe.async_batch_infer(input)
        outque.put(output)


def server(inques, outques):
    event_loop = asyncio.new_event_loop()
    asyncio.set_event_loop(event_loop)
    pipe = pipeline(model_path, backend_config=PytorchEngineConfig())
    for inque, outque in zip(inques, outques):
        event_loop.create_task(_batch_infer(inque, outque, pipe))
    event_loop.run_forever()

def client(inque, outque, message):
    inque.put(message)
    print(outque.get().text)


inques = [Queue(), Queue()]
outques = [Queue(), Queue()]

t_server = threading.Thread(target=server, args=(inques, outques))
t_client0 = threading.Thread(target=client, args=(inques[0], outques[0], 'Hakuna Matata
↪'))
t_client1 = threading.Thread(target=client, args=(inques[1], outques[1], 'giraffes are␣
↪heartless creatures'))
```

(continues on next page)

(continued from previous page)

```
t_server.start()
t_client0.start()
t_client1.start()

t_client0.join()
t_client1.join()
```

[!WARNING] This is NOT recommended, as multithreading introduces additional overhead, leading to unstable inference performance.

1.29 inference pipeline

1.29.1 pipeline

`lmdeploy.pipeline`(*model_path: str*, *backend_config: TurbomindEngineConfig | PytorchEngineConfig | None = None*, *chat_template_config: ChatTemplateConfig | None = None*, *log_level: str = 'WARNING'*, *max_log_len: int | None = None*, ***kwargs*)

Parameters

- **`model_path`** (*`str`*) – the path of a model. It could be one of the following options:
  - i) A local directory path of a turbomind model which is

    converted by *lmdeploy convert* command or download from ii) and iii).
  - ii) The model_id of a lmdeploy-quantized model hosted

    inside a model repo on huggingface.co, such as "InternLM/internlm-chat-20b-4bit", "lmdeploy/llama2-chat-70b-4bit", etc.
  - iii) The model_id of a model hosted inside a model repo

    on huggingface.co, such as "internlm/internlm-chat-7b", "Qwen/Qwen-7B-Chat ", "baichuan-inc/Baichuan2-7B-Chat" and so on.
- **`backend_config`** (`TurbomindEngineConfig | PytorchEngineConfig`) – backend config instance. Default to None.
- **`chat_template_config`** (`ChatTemplateConfig`) – chat template configuration. Default to None.
- **`log_level`** (*`str`*) – set log level whose value among [CRITICAL, ERROR, WARNING, INFO, DEBUG]
- **`max_log_len`** (*`int`*) – Max number of prompt characters or prompt tokens being printed in log

Examples

```
>>> # LLM
>>> import lmdeploy
>>> pipe = lmdeploy.pipeline('internlm/internlm-chat-7b')
>>> response = pipe(['hi','say this is a test'])
>>> print(response)
>>>
>>> # VLM
>>> from lmdeploy.vl import load_image
>>> from lmdeploy import pipeline, TurbomindEngineConfig, ChatTemplateConfig
>>> pipe = pipeline('liuhaotian/llava-v1.5-7b',
...                 backend_config=TurbomindEngineConfig(session_len=8192),
...                 chat_template_config=ChatTemplateConfig(model_name='vicuna'))
>>> im = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/
↪demo/resources/human-pose.jpg')
>>> response = pipe([('describe this image', [im])])
>>> print(response)
```

1.29.2 serving

`lmdeploy.serve`(*model_path: str*, *model_name: str | None = None*, *backend: Literal['turbomind', 'pytorch'] = 'turbomind'*, *backend_config: TurbomindEngineConfig | PytorchEngineConfig | None = None*, *chat_template_config: ChatTemplateConfig | None = None*, *server_name: str = '0.0.0.0'*, *server_port: int = 23333*, *log_level: str = 'ERROR'*, *api_keys: str | List[str] | None = None*, *ssl: bool = False*, ***kwargs*)

This will run the api_server in a subprocess.

Parameters

- **model_path** (*str*) – the path of a model. It could be one of the following options:
  - i) A local directory path of a turbomind model which is

    converted by *lmdeploy convert* command or download from ii) and iii).
  - ii) The model_id of a lmdeploy-quantized model hosted

    inside a model repo on huggingface.co, such as "InternLM/internlm-chat-20b-4bit", "lmdeploy/llama2-chat-70b-4bit", etc.
  - iii) The model_id of a model hosted inside a model repo

    on huggingface.co, such as "internlm/internlm-chat-7b", "Qwen/Qwen-7B-Chat ", "baichuan-inc/Baichuan2-7B-Chat" and so on.
- **model_name** (*str*) – the name of the served model. It can be accessed by the RESTful API */v1/models*. If it is not specified, *model_path* will be adopted
- **backend** (*str*) – either *turbomind* or *pytorch* backend. Default to *turbomind* backend.
- **backend_config** (`TurbomindEngineConfig` | `PytorchEngineConfig`) – backend config instance. Default to none.

- **chat_template_config** (ChatTemplateConfig) – chat template configuration. Default to None.
- **server_name** (*str*) – host ip for serving
- **server_port** (*int*) – server port
- **log_level** (*str*) – set log level whose value among [CRITICAL, ERROR, WARNING, INFO, DEBUG]
- **api_keys** (*List[str] | str | None*) – Optional list of API keys. Accepts string type as a single api_key. Default to None, which means no api key applied.
- **ssl** (*bool*) – Enable SSL. Requires OS Environment variables 'SSL_KEYFILE' and 'SSL_CERTFILE'.

Returns
A client chatbot for LLaMA series models.

Return type
APIClient

Examples

```
>>> import lmdeploy
>>> client = lmdeploy.serve('internlm/internlm-chat-7b', 'internlm-chat-7b')
>>> for output in client.chat('hi', 1):
...     print(output)
```

`lmdeploy.client`(*api_server_url: str = 'http://0.0.0.0:23333'*, *api_key: str | None = None*, ***kwargs*)

Parameters

- **api_server_url** (*str*) – communicating address 'http://<ip>:<port>' of api_server
- **api_key** (*str | None*) – api key. Default to None, which means no api key will be used.

Returns
Chatbot for LLaMA series models with turbomind as inference engine.

1.29.3 PytorchEngineConfig

class `lmdeploy.PytorchEngineConfig`(*dtype: str = 'auto'*, *tp: int = 1*, *session_len: int | None = None*, *max_batch_size: int | None = None*, *cache_max_entry_count: float = 0.8*, *prefill_interval: int = 16*, *block_size: int = 64*, *num_cpu_blocks: int = 0*, *num_gpu_blocks: int = 0*, *adapters: Dict[str, str] | None = None*, *max_prefill_token_num: int = 4096*, *thread_safe: bool = False*, *enable_prefix_caching: bool = False*, *device_type: str = 'cuda'*, *eager_mode: bool = False*, *custom_module_map: Dict[str, str] | None = None*, *download_dir: str | None = None*, *revision: str | None = None*, *quant_policy: Literal[0, 4, 8] = 0*)

PyTorch Engine Config.

Parameters

- **dtype** (*str*) – data type for model weights and activations. It can be one of the following values, ['auto', 'float16', 'bfloat16'] The *auto* option will use FP16 precision for FP32 and FP16 models, and BF16 precision for BF16 models.

- **tp** (*int*) – Tensor Parallelism. default 1.
- **session_len** (*int*) – Max session length. Default None.
- **max_batch_size** (*int*) – Max batch size. If it is not specified, the engine will automatically set it according to the device
- **cache_max_entry_count** (*float*) – the percentage of gpu memory occupied by the k/v cache. For lmdeploy versions greater than *v0.2.1*, it defaults to 0.8, signifying the percentage of FREE GPU memory to be reserved for the k/v cache
- **prefill_interval** (*int*) – Interval to perform prefill, Default 16.
- **block_size** (*int*) – paging cache block size, default 64.
- **num_cpu_blocks** (*int*) – Num cpu blocks. If num is 0, cache would be allocate according to current environment.
- **num_gpu_blocks** (*int*) – Num gpu blocks. If num is 0, cache would be allocate according to current environment.
- **adapters** (*dict*) – The path configs to lora adapters.
- **max_prefill_token_num** (*int*) – tokens per iteration.
- **thread_safe** (*bool*) – thread safe engine instance.
- **enable_prefix_caching** (*bool*) – Enable token match and sharing caches.
- **device_type** (*str*) – The inference device type, options ['cuda']
- **eager_mode** (*bool*) – Enable "eager" mode or not
- **custom_module_map** (*Dict*) – nn module map customized by users. Once provided, the original nn modules of the model will be substituted by the mapping ones
- **download_dir** (*str*) – Directory to download and load the weights, default to the default cache directory of huggingface.
- **revision** (*str*) – The specific model version to use. It can be a branch name, a tag name, or a commit id. If unspecified, will use the default version.
- **quant_policy** (*int*) – default to 0. When k/v is quantized into 4 or 8 bit, set it to 4 or 8, respectively

1.29.4 TurbomindEngineConfig

class lmdeploy.**TurbomindEngineConfig**(*dtype: str = 'auto', model_format: str | None = None, tp: int = 1, session_len: int | None = None, max_batch_size: int = None, cache_max_entry_count: float = 0.8, cache_chunk_size: int = -1, cache_block_seq_len: int = 64, enable_prefix_caching: bool = False, quant_policy: int = 0, rope_scaling_factor: float = 0.0, use_logn_attn: bool = False, download_dir: str | None = None, revision: str | None = None, max_prefill_token_num: int = 8192, num_tokens_per_iter: int = 0, max_prefill_iters: int = 1*)

TurboMind Engine config.

Parameters

- **dtype** (*str*) – data type for model weights and activations. It can be one of the following values, ['auto', 'float16', 'bfloat16'] The *auto* option will use FP16 precision for FP32 and FP16 models, and BF16 precision for BF16 models.

- **model_format** (*str*) – the layout of the deployed model. It can be one of the following values [hf, meta_llama, awq, gptq],`hf` meaning huggingface model(.bin, .safetensors), *meta_llama* being meta llama's format(.pth), *awq* and *gptq* meaning the quantized model by AWQ and GPTQ, respectively. If it is not specified, i.e. None, it will be extracted from the input model
- **tp** (*int*) – the number of GPU cards used in tensor parallelism, default to 1
- **session_len** (*int*) – the max session length of a sequence, default to None
- **max_batch_size** (*int*) – the max batch size during inference. If it is not specified, the engine will automatically set it according to the device
- **cache_max_entry_count** (*float*) – the percentage of gpu memory occupied by the k/v cache. For versions of lmdeploy between *v0.2.0* and *v0.2.1*, it defaults to 0.5, depicting the percentage of TOTAL GPU memory to be allocated to the k/v cache. For lmdeploy versions greater than *v0.2.1*, it defaults to 0.8, signifying the percentage of FREE GPU memory to be reserved for the k/v cache
- **cache_chunk_size** (*int*) – The policy to apply for KV block from the block manager, default to -1.
- **cache_block_seq_len** (*int*) – the length of the token sequence in a k/v block, default to 64
- **enable_prefix_caching** (*bool*) – enable cache prompts for block reuse, default to False
- **quant_policy** (*int*) – default to 0. When k/v is quantized into 4 or 8 bit, set it to 4 or 8, respectively
- **rope_scaling_factor** (*float*) – scaling factor used for dynamic ntk, default to 0. TurboMind follows the implementation of transformer LlamaAttention
- **use_logn_attn** (*bool*) – whether or not to use log attn: default to False
- **download_dir** (*str*) – Directory to download and load the weights, default to the default cache directory of huggingface.
- **revision** (*str*) – The specific model version to use. It can be a branch name, a tag name, or a commit id. If unspecified, will use the default version.
- **max_prefill_token_num** (*int*) – the number of tokens each iteration during prefill, default to 8192
- **num_tokens_per_iter** (*int*) – the number of tokens processed in each forward pass. Working with *max_prefill_iters* enables the "Dynamic SplitFuse"-like scheduling
- **max_prefill_iters** (*int*) – the max number of forward pass during prefill stage

1.29.5 GenerationConfig

class lmdeploy.**GenerationConfig**(*n: int = 1*, *max_new_tokens: int = 512*, *do_sample: bool = False*, *top_p: float = 1.0*, *top_k: int = 50*, *min_p: float = 0.0*, *temperature: float = 0.8*, *repetition_penalty: float = 1.0*, *ignore_eos: bool = False*, *random_seed: int | None = None*, *stop_words: List[str] | None = None*, *bad_words: List[str] | None = None*, *stop_token_ids: List[int] | None = None*, *bad_token_ids: List[int] | None = None*, *min_new_tokens: int | None = None*, *skip_special_tokens: bool = True*, *spaces_between_special_tokens: bool = True*, *logprobs: int | None = None*, *response_format: Dict | None = None*, *logits_processors: List[Callable[[torch.Tensor, torch.Tensor], torch.Tensor]] | None = None*, *output_logits: Literal['all', 'generation'] | None = None*, *output_last_hidden_state: Literal['all', 'generation'] | None = None*)

generation parameters used by inference engines.

Parameters

- **n** (*int*) – Define how many chat completion choices to generate for each input message. **Only 1** is supported now.
- **max_new_tokens** (*int*) – The maximum number of tokens that can be generated in the chat completion
- **do_sample** (*bool*) – Whether or not to use sampling, use greedy decoding otherwise. Default to be False.
- **top_p** (*float*) – An alternative to sampling with temperature, called nucleus sampling, where the model considers the results of the tokens with top_p probability mass
- **top_k** (*int*) – An alternative to sampling with temperature, where the model considers the top_k tokens with the highest probability
- **min_p** (*float*) – Minimum token probability, which will be scaled by the probability of the most likely token. It must be a value between 0 and 1. Typical values are in the 0.01-0.2 range, comparably selective as setting *top_p* in the 0.99-0.8 range (use the opposite of normal *top_p* values)
- **temperature** (*float*) – Sampling temperature
- **repetition_penalty** (*float*) – Penalty to prevent the model from generating repeated words or phrases. A value larger than 1 discourages repetition
- **ignore_eos** (*bool*) – Indicator to ignore the eos_token_id or not
- **random_seed** (*int*) – Seed used when sampling a token
- **stop_words** (*List[str]*) – Words that stop generating further tokens
- **bad_words** (*List[str]*) – Words that the engine will never generate
- **stop_token_ids** (*List[int]*) – List of tokens that stop the generation when they are generated. The returned output will not contain the stop tokens.
- **bad_token_ids** (*List[str]*) – List of tokens that the engine will never generate.
- **min_new_tokens** (*int*) – The minimum numbers of tokens to generate, ignoring the number of tokens in the prompt.
- **skip_special_tokens** (*bool*) – Whether or not to remove special tokens in the decoding. Default to be True.
- **spaces_between_special_tokens** (*bool*) – Whether or not to add spaces around special tokens. The behavior of Fast tokenizers is to have this to False. This is setup to True in slow tokenizers.

- **logprobs** (*int*) – Number of log probabilities to return per output token.
- **response_format** (*Dict*) – Only pytorch backend support formatting
- **Examples** (*response.*) –

  {
    "type": "json_schema", "json_schema": {
      "name": "test", "schema": { "properties": {
        "name": { "type": "string" }
      }, "required": ["name"], "type": "object" }
    }
  }
- **or** –

  {
    "type": "regex_schema", "regex_schema": "call me [A-Za-z]{1,10}"
  }

:param :

{
  "type": "regex_schema", "regex_schema": "call me [A-Za-z]{1,10}"
}

Parameters
logits_processors (*List[Callable]*) – Custom logit processors.

1.29.6 ChatTemplateConfig

class lmdeploy.**ChatTemplateConfig**(*model_name: str, system: str | None = None, meta_instruction: str | None = None, eosys: str | None = None, user: str | None = None, eoh: str | None = None, assistant: str | None = None, eoa: str | None = None, tool: str | None = None, eotool: str | None = None, separator: str | None = None, capability: Literal['completion', 'infilling', 'chat', 'python'] | None = None, stop_words: List[str] | None = None*)

Parameters for chat template.

Parameters

- **model_name** (*str*) – the name of the deployed model. Determine which chat template will be applied. All the chat template names: *lmdeploy list*
- **system** (*str | None*) – begin of the system prompt
- **meta_instruction** (*str | None*) – system prompt
- **eosys** (*str | None*) – end of the system prompt
- **user** (*str | None*) – begin of the user prompt
- **eoh** (*str | None*) – end of the user prompt
- **assistant** (*str | None*) – begin of the assistant prompt
- **eoa** (*str | None*) – end of the assistant prompt

- **tool** (*str | None*) – begin of the tool prompt
- **eotool** (*str | None*) – end of the tool prompt
- **capability** – ('completion' | 'infilling' | 'chat' | 'python') = None

CHAPTER TWO

INDICES AND TABLES

- genindex
- search

INDEX