Qwen

Qwen Team

2024 年 06 月 17 日

快速开始

Qwen 是阿里巴巴集团 Qwen 团队研发的大语言模型和大型多模态模型系列。目前，大语言模型已升级至 Qwen2 版本。无论是语言模型还是多模态模型，均在大规模多语言和多模态数据上进行预训练，并通过高质量数据进行后期微调以贴近人类偏好。Qwen 具备自然语言理解、文本生成、视觉理解、音频理解、工具使用、角色扮演、作为 AI Agent 进行互动等多种能力。

最新版本 Qwen2 有以下特点：

- 5 种模型规模，包括 0.5B、1.5B、7B、57B-A14B 和 72B；
- 针对每种尺寸提供基础模型和指令微调模型，并确保指令微调模型按照人类偏好进行校准；
- 基础模型和指令微调模型的多语言支持；
- 所有模型均稳定支持 32K 长度上下文；Qwen2-7B-Instruct 与 Qwen2-72B-Instruct 可支持 128K 上下文（需额外配置）
- 支持工具调用、RAG（检索增强文本生成）、角色扮演、AI Agent 等；

想了解更多信息，欢迎访问：

- 博客
- GitHub
- Hugging Face
- ModelScope
- Qwen2 Collection

加入社区，加入 Discord 和 微信群 。很期待见到你们！

CHAPTER 1

文档

1.1 安装

要快速上手 Qwen2，您可以从 Hugging Face 安装 `transformers` 库，并使用 Qwen2 Collection 中的模型。我们建议您安装最新版本的 `transformers` 库，或者至少安装 4.40.0 版本。

1.1.1 Pip 安装

```
pip install transformers -U
```

1.1.2 Conda 安装

```
conda install conda-forge::transformers
```

1.1.3 从源码安装

```
pip install git+https://github.com/huggingface/transformers
```

我们建议您使用 Python3.8 及以上版本和 Pytorch 2.0 及以上版本。

1.2 快速开始

本指南帮助您快速上手 Qwen2 的使用，并提供了如下示例：Hugging Face Transformers 以及 ModelScope 和 vLLM 在部署时的应用实例。

1.2.1 Hugging Face Transformers & ModelScope

要快速上手 Qwen2，我们建议您首先尝试使用 transformers 进行推理。请确保已安装了 `transformers>=4.40.0` 版本。以下是一个非常简单的代码片段示例，展示如何运行 Qwen2-Instruct 模型，其中包含 `Qwen2-7B-Instruct` 的实例：

```
from transformers import AutoModelForCausalLM, AutoTokenizer
device = "cuda" # the device to load the model onto

# Now you do not need to add "trust_remote_code=True"
model = AutoModelForCausalLM.from_pretrained(
    "Qwen/Qwen2-7B-Instruct",
    torch_dtype="auto",
    device_map="auto"
)
tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen2-7B-Instruct")

# Instead of using model.chat(), we directly use model.generate()
# But you need to use tokenizer.apply_chat_template() to format your inputs as shown␣
↪below
prompt = "Give me a short introduction to large language model."
messages = [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": prompt}
]
text = tokenizer.apply_chat_template(
    messages,
    tokenize=False,
    add_generation_prompt=True
)
model_inputs = tokenizer([text], return_tensors="pt").to(device)

# Directly use generate() and tokenizer.decode() to get the output.
# Use `max_new_tokens` to control the maximum output length.
generated_ids = model.generate(
    model_inputs.input_ids,
    max_new_tokens=512
)
generated_ids = [
    output_ids[len(input_ids):] for input_ids, output_ids in zip(model_inputs.input_
↪ids, generated_ids)
]

response = tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
```

以前，我们使用 `model.chat()`（有关更多详细信息，请参阅先前 Qwen 模型中的 `modeling_qwen.py`）。现在，我们遵循 `transformers` 的实践，直接使用 `model.generate()` 配合 tokenizer 中的 `apply_chat_template()` 方法。

如果你想使用 Flash Attention 2，你可以用下面这种方式读取模型：

```
 model = AutoModelForCausalLM.from_pretrained(
    "Qwen/Qwen2-7B-Instruct",
    torch_dtype="auto",
    device_map="auto",
    attn_implementation="flash_attention_2",
)
```

为了解决下载问题，我们建议您尝试从 ModelScope 进行下载，只需将上述代码的第一行更改为以下内容：

```
from modelscope import AutoModelForCausalLM, AutoTokenizer
```

借助 `TextStreamer`，chat 的流式模式变得非常简单。下面我们将展示一个如何使用它的示例：

```
...
# Reuse the code before `model.generate()` in the last code snippet
from transformers import TextStreamer
streamer = TextStreamer(tokenizer, skip_prompt=True, skip_special_tokens=True)
generated_ids = model.generate(
    model_inputs.input_ids,
    max_new_tokens=512,
    streamer=streamer,
)
```

1.2.2 使用 vLLM 部署

要部署 Qwen2，我们建议您使用 vLLM。vLLM 是一个用于 LLM 推理和服务的快速且易于使用的框架。以下，我们将展示如何使用 vLLM 构建一个与 OpenAI API 兼容的 API 服务。

首先，确保你已经安装 `vLLM>=0.4.0`：

```
pip install vllm
```

运行以下代码以构建 vllm 服务。此处我们以 Qwen2-7B-Instruct 为例：

```
python -m vllm.entrypoints.openai.api_server --model Qwen/Qwen2-7B-Instruct
```

然后，您可以使用 create chat interface 来与 Qwen 进行交流：

```
curl http://localhost:8000/v1/chat/completions  -H "Content-Type: application/json" -
↪d '{
   "model": "Qwen/Qwen2-7B-Instruct",
   "messages": [
   {"role": "system", "content": "You are a helpful assistant."},
   {"role": "user", "content": "Tell me something about large language models."}
   ]
   }'
```

或者您可以按照下面所示的方式，使用 `openai` Python 包中的 Python 客户端：

```
from openai import OpenAI
# Set OpenAI's API key and API base to use vLLM's API server.
openai_api_key = "EMPTY"
openai_api_base = "http://localhost:8000/v1"

client = OpenAI(
```

(续下页)

(接上页)

```
    api_key=openai_api_key,
    base_url=openai_api_base,
)

chat_response = client.chat.completions.create(
    model="Qwen/Qwen2-7B-Instruct",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Tell me something about large language models."},
    ]
)
print("Chat response:", chat_response)
```

1.2.3 下一步

现在，您可以尽情探索 Qwen 模型的各种用途。若想了解更多，请随时查阅本文档中的其他内容。

1.3 使用 Transformers 实现 Chat

使用 Qwen2 最简单的方法就是利用 `transformers` 库与之对话。在本文档中，我们将展示如何在流式模式或非流式模式下与 Qwen2-7B-Instruct 进行对话。

1.3.1 基本用法

你只需借助 `transformers` 库编写几行代码，就能与 Qwen2-Instruct 进行对话。实质上，我们通过 `from_pretrained` 方法构建 tokenizer 和模型，然后利用 `generate` 方法，在 tokenizer 提供的 chat template 的辅助下进行对话。以下是一个如何与 Qwen2-7B-Instruct 进行对话的示例：

```
from transformers import AutoModelForCausalLM, AutoTokenizer
device = "cuda" # the device to load the model onto

# Now you do not need to add "trust_remote_code=True"
model = AutoModelForCausalLM.from_pretrained(
    "Qwen/Qwen2-7B-Instruct",
    torch_dtype="auto",
    device_map="auto"
)
tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen2-7B-Instruct")

# Instead of using model.chat(), we directly use model.generate()
# But you need to use tokenizer.apply_chat_template() to format your inputs as shown␣
↪below
prompt = "Give me a short introduction to large language model."
messages = [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": prompt}
]
text = tokenizer.apply_chat_template(
    messages,
    tokenize=False,
    add_generation_prompt=True
```

(续下页)

(接上页)

```
)
model_inputs = tokenizer([text], return_tensors="pt").to(device)

# Directly use generate() and tokenizer.decode() to get the output.
# Use `max_new_tokens` to control the maximum output length.
generated_ids = model.generate(
    model_inputs.input_ids,
    max_new_tokens=512
)
generated_ids = [
    output_ids[len(input_ids):] for input_ids, output_ids in zip(model_inputs.input_
↪ids, generated_ids)
]

response = tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
```

如果你想使用 Flash Attention 2，你可以用下面这种方式读取模型：

```
 model = AutoModelForCausalLM.from_pretrained(
    "Qwen/Qwen2-7B-Instruct",
    torch_dtype="auto",
    device_map="auto",
    attn_implementation="flash_attention_2",
)
```

请注意，原 Qwen 仓库中的旧方法 `chat()` 现在已被 `generate()` 方法替代。这里使用了 `apply_chat_template()` 函数将消息转换为模型能够理解的格式。其中的 `add_generation_prompt` 参数用于在输入中添加生成提示，该提示指向 `<|im_start|>assistant\n` 。尤其需要注意的是，我们遵循先前实践，对 chat 模型应用 ChatML 模板。而 `max_new_tokens` 参数则用于设置响应的最大长度。此外，通过 `tokenizer.batch_decode()` 函数对响应进行解码。关于输入部分，上述的 `messages` 是一个示例，展示了如何格式化对话历史记录和系统提示。默认情况下，如果您没有指定系统提示，我们将直接使用 `You are a helpful assistant.` 作为系统提示。

1.3.2 流式输出

借助 `TextStreamer` ，您可以将与 Qwen 的对话切换到流式传输模式。下面是一个关于如何使用它的示例：

```
# Repeat the code above before model.generate()
# Starting here, we add streamer for text generation.
from transformers import TextStreamer
streamer = TextStreamer(tokenizer, skip_prompt=True, skip_special_tokens=True)

# This will print the output in the streaming mode.
generated_ids = model.generate(
    model_inputs,
    max_new_tokens=512,
    streamer=streamer,
)
```

除了使用 `TextStreamer` 之外，我们还可以使用 `TextIteratorStreamer` ，它将可打印的文本存储在一个队列中，以便下游应用程序作为迭代器来使用：

```
# Repeat the code above before model.generate()
# Starting here, we add streamer for text generation.
```

(续下页)

(接上页)

```
from transformers import TextIteratorStreamer
streamer = TextIteratorStreamer(tokenizer, skip_prompt=True, skip_special_tokens=True)

from threading import Thread
generation_kwargs = dict(model_inputs, streamer=streamer, max_new_tokens=512)
thread = Thread(target=model.generate, kwargs=generation_kwargs)

thread.start()
generated_text = ""
for new_text in streamer:
    generated_text += new_text
print(generated_text)
```

1.3.3 下一步

现在，你可以选择流式模式或非流式模式与 Qwen2 进行对话。继续阅读文档，并尝试探索模型推理的更多高级用法！

1.4 llama.cpp

llama.cpp 是一个 C++ 库，用于简化 LLM 推理的设置。它使得在本地机器上运行 Qwen 成为可能。该库是一个纯 C/C++ 实现，不依赖任何外部库，并且针对 x86 架构提供了 AVX、AVX2 和 AVX512 加速支持。此外，它还提供了 2、3、4、5、6 以及 8 位量化功能，以加快推理速度并减少内存占用。对于大于总 VRAM 容量的大规模模型，该库还支持 CPU+GPU 混合推理模式进行部分加速。本质上，llama.cpp 的用途在于运行 GGUF（由 GPT 生成的统一格式）模型。欲了解更多详情，请参阅官方 GitHub 仓库。以下我们将演示如何使用 llama.cpp 运行 Qwen。

1.4.1 准备

这个示例适用于 Linux 或 MacOS 系统。第一步操作是："克隆仓库并进入该目录：

```
git clone https://github.com/ggerganov/llama.cpp
cd llama.cpp
```

然后运行 `make` 命令：

```
make
```

然后你就能使用 `llama.cpp` 运行 GGUF 文件。

1.4.2 运行 Qwen 的 GGUF 文件

我们在 Hugging Face 组织中提供了一系列 GGUF 模型，为了找到您需要的模型，您可以搜索仓库名称中包含 -GGUF 的部分。要下载所需的 GGUF 模型，请使用 huggingface-cli（首先需要通过命令 pip install huggingface_hub 安装它）：

```
huggingface-cli download <model_repo> <gguf_file> --local-dir <local_dir> --local-dir-
↪use-symlinks False
```

比如：

```
huggingface-cli download Qwen/Qwen2-7B-Instruct-GGUF qwen2-7b-instruct-q5_k_m.gguf --
↪local-dir . --local-dir-use-symlinks False
```

然后你可以用如下命令运行模型：

```
./main -m qwen2-7b-instruct-q5_k_m.gguf -n 512 --color -i -cml -f prompts/chat-with-
↪qwen.txt
```

-n 指的是要生成的最大 token 数量。这里还有其他超参数供你选择，并且你可以运行

```
./main -h
```

以了解它们。

1.4.3 生成你的 GGUF 文件

我们在 quantization/llama.cpp 中介绍了创建和量化 GGUF 文件的方法。您可以参考该文档获取更多信息。

1.4.4 PPL 评测

llama.cpp 为我们提供了评估 GGUF 模型 PPL 性能的方法。为了实现这一点，你需要准备一个数据集，比如“wiki 测试”。这里我们展示了一个运行测试的例子。

第一步，下载数据集：

```
wget https://s3.amazonaws.com/research.metamind.io/wikitext/wikitext-2-raw-v1.zip?
↪ref=salesforce-research -O wikitext-2-raw-v1.zip
unzip wikitext-2-raw-v1.zip
```

然后你可以用如下命令运行测试：

```
./perplexity -m models/7B/ggml-model-q4_0.gguf -f wiki.test.raw
```

输出如下所示

```
perplexity : calculating perplexity over 655 chunks
24.43 seconds per pass - ETA 4.45 hours
[1]4.5970,[2]5.1807,[3]6.0382,...
```

稍等一段时间你将得到模型的 PPL 评测结果。

1.4.5 在 LM Studio 使用 GGUF

如果你仍然觉得使用 llama.cpp 有困难，我建议你尝试一下 LM Studio 这个平台，它允许你搜索和运行本地的大规模语言模型。Qwen2 已经正式成为 LM Studio 的一部分。祝你使用愉快！

1.5 MLX-LM

mlx-lm 能让你在 Apple Silicon 上运行大型语言模型，适用于 MacOS。`mlx-lm` 已支持 Qwen 模型，此次我们提供直接可用的模型文件。

1.5.1 准备工作

首先需要安装 `mlx-lm` 包：

- 使用 `pip`：

```
pip install mlx-lm
```

- 使用 `conda`：

```
conda install -c conda-forge mlx-lm
```

1.5.2 使用 Qwen MLX 模型文件

我们已在 Hugging Face 提供了适用于 `mlx-lm` 的模型文件，请搜索带 `-MLX` 的存储库。

这里我们展示了一个代码样例，其中使用了 `apply_chat_template` 来应用对话模板。

```
from mlx_lm import load, generate

model, tokenizer = load('Qwen/Qwen2-7B-Instruct-MLX', tokenizer_config={"eos_token": "
↪<|im_end|>"})

prompt = "Give me a short introduction to large language model."
messages = [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": prompt}
]
text = tokenizer.apply_chat_template(
    messages,
    tokenize=False,
    add_generation_prompt=True
)

response = generate(model, tokenizer, prompt=text, verbose=True, top_p=0.8, temp=0.7,␣
↪repetition_penalty=1.05, max_tokens=512)
```

1.5.3 自行制作 MLX 格式模型

仅用一条命令即可制作 mlx 格式模型

```
mlx_lm.convert --hf-path Qwen/Qwen2-7B-Instruct --mlx-path mlx/Qwen2-7B-Instruct/ -q
```

参数含义分别是

- `--hf-path`: Hugging Face Hub 上的模型名或本地路径
- `--mlx-path`: 输出模型文件的存储路径
- `-q`: 启用量化

1.6 Ollama

Ollama 帮助您通过少量命令即可在本地运行 LLM。它适用于 MacOS、Linux 和 Windows 操作系统。现在，Qwen2 正式上线 Ollama，您只需一条命令即可运行它：

```
ollama run qwen2
```

接着，我们介绍在 Ollama 使用 Qwen2 模型的更多用法

1.6.1 快速开始

访问官方网站 Ollama ，点击 `Download` 以在您的设备上安装 Ollama。您还可以在网站上搜索模型，在这里您可以找到 Qwen2 系列模型。除了默认模型之外，您可以通过以下方式选择运行不同大小的 Qwen2-Instruct 模型：

- `ollama run qwen2:0.5b`
- `ollama run qwen2:1.5b`
- `ollama run qwen2:7b`
- `ollama run qwen2:72b`

1.6.2 在 Ollama 运行你的 GGUF 文件

有时您可能不想拉取模型，而是希望直接使用自己的 GGUF 文件来配合 Ollama。假设您有一个名为 `qwen2-7b-instruct-q5_0.gguf` 的 Qwen2 的 GGUF 文件。在第一步中，您需要创建一个名为 `Modelfile` 的文件。该文件的内容如下所示：

```
FROM qwen2-7b-instruct-q5_0.gguf

# set the temperature to 1 [higher is more creative, lower is more coherent]
PARAMETER temperature 0.7
PARAMETER top_p 0.8
PARAMETER repeat_penalty 1.05
PARAMETER top_k 20

TEMPLATE """{{ if and .First .System }}<|im_start|>system
{{ .System }}<|im_end|>
{{ end }}<|im_start|>user
```

(续下页)

(接上页)

```
{{ .Prompt }}<|im_end|>
<|im_start|>assistant
{{ .Response }}"""

# set the system message
SYSTEM """
You are a helpful assistant.
"""
```

然后通过运行下列命令来创建一个 ollama 模型

```
ollama create qwen2_7b -f Modelfile
```

完成后，你即可运行你的 ollama 模型：

```
ollama run qwen2_7b
```

1.7 Text Generation Web UI

Text Generation Web UI （简称 TGW，通常被称为 “oobabooga”）是一款流行的文本生成 Web 界面工具，类似于 AUTOMATIC1111/stable-diffusion-webui 。它拥有多个交互界面，并支持多种模型后端，包括 Transformers 、llama.cpp（通过 llama-cpp-python 实现）、ExLlamaV2 、AutoGPTQ 、AutoAWQ 、GPTQ-for-LLaMa 、CTransformers 以及 QuIP# 。在本节中，我们将介绍如何在本地环境中使用 TGW 运行 Qwen。

1.7.1 快速开始

最简单的运行 TGW（Text Generation WebUI）的方法是使用 repo 中提供的 Shell 脚本。首先，克隆 repo 并进去文件夹中：

```
git clone https://github.com/oobabooga/text-generation-webui
cd text-generation-webui
```

你可以根据你的操作系统直接运行相应的脚本，例如在 Linux 系统上运行 `start_linux.sh` ，在 Windows 系统上运行 `start_windows.bat` ，在 MacOS 系统上运行 `start_macos.sh` ，或者在 Windows 子系统 Linux（WSL）上运行 `start_wsl.bat` 。另外，你也可以选择手动在 conda 环境中安装所需的依赖项。这里以 MacOS 系统为例进行实践操作。

```
conda create -n textgen python=3.11
conda activate textgen
pip install torch torchvision torchaudio
```

接下来，您可以根据您的操作系统执行 `pip install -r` 命令来安装相应的依赖项，例如，

```
pip install -r requirements_apple_silicon.txt
```

对于 requirements 中的 `bitsandbytes` 和 `llama-cpp-python` ，我建议您直接通过 `pip` 进行安装。但是，暂时请不要使用 GGUF，因为其与 TGW 配合时的性能表现不佳。在完成所需包的安装之后，您需要准备模型，将模型文件或目录放在 `./models` 文件夹中。例如，您应按照以下方式将 `Qwen2-7B-Instruct` 的 transformers 模型目录放置到相应位置。

```
text-generation-webui
├── models
│   ├── Qwen2-7B-Instruct
│   │   ├── config.json
│   │   ├── generation_config.json
│   │   ├── model-00001-of-00004.safetensor
│   │   ├── model-00002-of-00004.safetensor
│   │   ├── model-00003-of-00004.safetensor
│   │   ├── model-00004-of-00004.safetensor
│   │   ├── model.safetensor.index.json
│   │   ├── merges.txt
│   │   ├── tokenizer_config.json
│   │   └── vocab.json
```

随后你需要运行

```
python server.py
```

来启动你的网页服务。请点击进入

```
`http://localhost:7860/?__theme=dark`
```

然后享受使用 Qwen 的 Web UI 吧！

1.7.2 下一步

TGW 中包含了许多更多用途，您甚至可以在其中享受角色扮演的乐趣，并使用不同类型的量化模型。您可以训练诸如 LoRA 这样的算法，并将 Stable Diffusion 和 Whisper 等扩展功能纳入其中。赶快去探索更多高级用法，并将它们应用于 Qwen 模型中吧！

1.8 AWQ

对于量化模型，我们推荐使用 AWQ 结合 AutoAWQ 。AWQ 即激活值感知的权重量化 (Activation-aware Weight Quantization)，是一种针对 LLM 的低比特权重量化的硬件友好方法。而 AutoAWQ 是一个易于使用的工具包，用于 4 比特量化模型。相较于 FP16，AutoAWQ 能够将模型的运行速度提升 3 倍，并将内存需求降低至原来的 1/3。AutoAWQ 实现了 AWQ 算法，可用于 LLM 的量化处理。在本文档中，我们将向您展示如何在 Transformers 框架下使用量化模型，以及如何对您自己的模型进行量化。

1.8.1 如何在 Transformers 中使用 AWQ 量化模型

现在，Transformers 已经正式支持 AutoAWQ，这意味着您可以直接在 Transformers 中使用量化模型。以下是一个非常简单的代码片段，展示如何运行量化模型 `Qwen2-7B-Instruct-AWQ`：

```
from transformers import AutoModelForCausalLM, AutoTokenizer
device = "cuda" # the device to load the model onto

model = AutoModelForCausalLM.from_pretrained(
    "Qwen/Qwen2-7B-Instruct-AWQ", # the quantized model
    device_map="auto"
)
tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen2-7B-Instruct-AWQ")
```

(续下页)

(接上页)

```
prompt = "Give me a short introduction to large language model."
messages = [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": prompt}
]
text = tokenizer.apply_chat_template(
    messages,
    tokenize=False,
    add_generation_prompt=True
)
model_inputs = tokenizer([text], return_tensors="pt").to(device)

generated_ids = model.generate(
    model_inputs.input_ids,
    max_new_tokens=512
)
generated_ids = [
    output_ids[len(input_ids):] for input_ids, output_ids in zip(model_inputs.input_
↪ids, generated_ids)
]

response = tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
```

1.8.2 如何在 vLLM 中使用 AWQ 量化模型

vLLM 已经支持了 AWQ，这意味着您可以直接使用我们提供的 AWQ 模型，或者是通过 AutoAWQ 训练得到的与 vLLM 兼容的模型。实际上，其用法与 vLLM 的基本用法相同。我们提供了一个简单的示例，展示了如何通过 vLLM 启动与 OpenAI API 兼容的接口，并使用 `Qwen2-7B-Instruct-AWQ` 模型：

```
python -m vllm.entrypoints.openai.api_server --model Qwen/Qwen2-7B-Instruct-AWQ
```

```
curl http://localhost:8000/v1/chat/completions  -H "Content-Type: application/json" -
↪d '{
   "model": "Qwen/Qwen2-7B-Instruct-AWQ",
   "messages": [
   {"role": "system", "content": "You are a helpful assistant."},
   {"role": "user", "content": "Tell me something about large language models."}
   ],
   }'
```

或者你可以按照下面所示的方式，使用 `openai` Python 包中的 Python 客户端：

```
from openai import OpenAI
# Set OpenAI's API key and API base to use vLLM's API server.
openai_api_key = "EMPTY"
openai_api_base = "http://localhost:8000/v1"

client = OpenAI(
    api_key=openai_api_key,
    base_url=openai_api_base,
)

chat_response = client.chat.completions.create(
```

(续下页)

(接上页)

```
    model="Qwen/Qwen2-7B-Instruct-AWQ",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Tell me something about large language models."},
    ]
)
print("Chat response:", chat_response)
```

1.8.3 使用 AutoAWQ 量化你的模型

如果您希望将自定义模型量化为 AWQ 量化模型，我们建议您使用 AutoAWQ。推荐通过安装源代码来获取并安装该工具包的最新版本：

```
git clone https://github.com/casper-hansen/AutoAWQ.git
cd AutoAWQ
pip install -e .
```

假设你已经基于 `Qwen2-7B` 模型进行了微调，并将其命名为 `Qwen2-7B-finetuned`，且使用的是你自己的数据集，比如 Alpaca。若要构建你自己的 AWQ 量化模型，你需要使用训练数据进行校准。以下，我们将为你提供一个简单的演示示例以便运行：

```
from awq import AutoAWQForCausalLM
from transformers import AutoTokenizer

# Specify paths and hyperparameters for quantization
model_path = "your_model_path"
quant_path = "your_quantized_model_path"
quant_config = { "zero_point": True, "q_group_size": 128, "w_bit": 4, "version": "GEMM
↪" }

# Load your tokenizer and model with AutoAWQ
tokenizer = AutoTokenizer.from_pretrained(model_path)
model = AutoAWQForCausalLM.from_pretrained(model_path, device_map="auto",␣
↪safetensors=True)
```

接下来，您需要准备数据以进行校准。您需要做的就是将样本放入一个列表中，其中每个样本都是一段文本。由于我们直接使用微调数据来进行校准，所以我们首先使用 ChatML 模板对其进行格式化。例如：

```
data = []
for msg in messages:
    msg = c['messages']
    text = tokenizer.apply_chat_template(msg, tokenize=False, add_generation_
↪prompt=False)
    data.append(text.strip())
```

其中每个 `msg` 是一个典型的聊天消息，如下所示：

```
[
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "Tell me who you are."},
    {"role": "assistant", "content": "I am a large language model named Qwen..."}
]
```

然后只需通过一行代码运行校准过程：

```
model.quantize(tokenizer, quant_config=quant_config, calib_data=data)
```

最后，保存量化模型：

```
model.save_quantized(quant_path, safetensors=True, shard_size="4GB")
tokenizer.save_pretrained(quant_path)
```

然后你就可以得到一个可以用于部署的 AWQ 量化模型。玩得开心！

1.9 GPTQ

GPTQ 是一种针对类 GPT 大型语言模型的量化方法，它基于近似二阶信息进行一次性权重量化。在本文档中，我们将向您展示如何使用 `transformers` 库加载并应用量化后的模型，同时也会指导您如何通过 AutoGPTQ 来对您自己的模型进行量化处理。

1.9.1 在 Transformers 中使用 GPTQ 模型

现在，Transformers 正式支持了 AutoGPTQ，这意味着您能够直接在 Transformers 中使用量化后的模型。以下是一个非常简单的代码片段示例，展示如何运行 `Qwen2-7B-Instruct-GPTQ-Int8` （请注意，对于每种大小的 Qwen2 模型，我们都提供了 Int4 和 Int8 两种量化版本）：

```
from transformers import AutoModelForCausalLM, AutoTokenizer
device = "cuda" # the device to load the model onto

model = AutoModelForCausalLM.from_pretrained(
    "Qwen/Qwen2-7B-Instruct-GPTQ-Int8", # the quantized model
    device_map="auto"
)
tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen2-7B-Instruct-GPTQ-Int8")

prompt = "Give me a short introduction to large language model."
messages = [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": prompt}
]
text = tokenizer.apply_chat_template(
    messages,
    tokenize=False,
    add_generation_prompt=True
)
model_inputs = tokenizer([text], return_tensors="pt").to(device)

generated_ids = model.generate(
    model_inputs.input_ids,
    max_new_tokens=512
)
generated_ids = [
    output_ids[len(input_ids):] for input_ids, output_ids in zip(model_inputs.input_
↪ids, generated_ids)
]

response = tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
```

1.9.2 在 vLLM 中使用 GPTQ 量化模型

vLLM 已经支持了 GPTQ，这意味着您可以直接使用我们提供的 GPTQ 模型，或者那些通过 AutoGPTQ 训练得到的模型与 vLLM 结合使用。实际上，其用法与 vLLM 的基本用法相同。我们提供了一个简单的示例，展示了如何使用 vLLM 以及 `Qwen2-7B-Instruct-GPTQ-Int8` 模型启动与 OpenAI API 兼容的 API：

```
python -m vllm.entrypoints.openai.api_server --model Qwen/Qwen2-7B-Instruct-GPTQ-Int8
```

```
curl http://localhost:8000/v1/chat/completions  -H "Content-Type: application/json" -
↪d '{
   "model": "Qwen/Qwen2-7B-Instruct-GPTQ-Int8",
   "messages": [
   {"role": "system", "content": "You are a helpful assistant."},
   {"role": "user", "content": "Tell me something about large language models."}
   ],
   }'
```

或者你可以按照下面所示的方式，使用 `openai` Python 包中的 Python 客户端：

```
from openai import OpenAI
# Set OpenAI's API key and API base to use vLLM's API server.
openai_api_key = "EMPTY"
openai_api_base = "http://localhost:8000/v1"

client = OpenAI(
    api_key=openai_api_key,
    base_url=openai_api_base,
)

chat_response = client.chat.completions.create(
    model="Qwen/Qwen2-7B-Instruct-GPTQ-Int8",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Tell me something about large language models."},
    ]
)
print("Chat response:", chat_response)
```

1.9.3 使用 AutoGPTQ 量化你的模型

如果你想将自定义模型量化为 GPTQ 量化模型，我们建议你使用 AutoGPTQ 工具。推荐通过安装源代码的方式获取并安装最新版本的该软件包。

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

假设你已经基于 `Qwen2-7B` 模型进行了微调，并将该微调后的模型命名为 `Qwen2-7B-finetuned`，且使用的是自己的数据集，比如 Alpaca。要构建你自己的 GPTQ 量化模型，你需要使用训练数据进行校准。以下是一个简单的演示示例，供你参考运行：

```
from auto_gptq import AutoGPTQForCausalLM, BaseQuantizeConfig
from transformers import AutoTokenizer

# Specify paths and hyperparameters for quantization
```

(续下页)

(接上页)

```
model_path = "your_model_path"
quant_path = "your_quantized_model_path"
quantize_config = BaseQuantizeConfig(
    bits=8, # 4 or 8
    group_size=128,
    damp_percent=0.01,
    desc_act=False,  # set to False can significantly speed up inference but the␣
↪perplexity may slightly bad
    static_groups=False,
    sym=True,
    true_sequential=True,
    model_name_or_path=None,
    model_file_base_name="model"
)
max_len = 8192

# Load your tokenizer and model with AutoGPTQ
# To learn about loading model to multiple GPUs,
# visit https://github.com/AutoGPTQ/AutoGPTQ/blob/main/docs/tutorial/02-Advanced-
↪Model-Loading-and-Best-Practice.md
tokenizer = AutoTokenizer.from_pretrained(model_path)
model = AutoGPTQForCausalLM.from_pretrained(model_path, quantize_config)
```

但是，如果你想使用多 GPU 来读取模型，你需要使用 `max_memory` 而不是 `device_map`。下面是一段示例代码：

```
model = AutoGPTQForCausalLM.from_pretrained(
    model_path,
    quantize_config,
    max_memory={i:"20GB" for i in range(4)}
)
```

接下来，你需要准备数据进行校准。你需要做的是将样本放入一个列表中，其中每个样本都是一段文本。由于我们直接使用微调数据进行校准，所以我们首先使用 ChatML 模板对它进行格式化处理。例如：

```
import torch

data = []
for msg in messages:
    text = tokenizer.apply_chat_template(msg, tokenize=False, add_generation_
↪prompt=False)
    model_inputs = tokenizer([text])
    input_ids = torch.tensor(model_inputs.input_ids[:max_len], dtype=torch.int)
    data.append(dict(input_ids=input_ids, attention_mask=input_ids.ne(tokenizer.pad_
↪token_id)))
```

接下来，你需要准备数据以进行校准。你需要做的就是将样本放入一个列表中，其中每个样本都是文本。由于我们直接使用微调数据来进行校准，所以我们首先使用 ChatML 模板来格式化它。例如：

```
[
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "Tell me who you are."},
    {"role": "assistant", "content": "I am a large language model named Qwen..."}
]
```

然后只需通过一行代码运行校准过程：

```
import logging

logging.basicConfig(
    format="%(asctime)s %(levelname)s [%(name)s] %(message)s", level=logging.INFO,␣
↪datefmt="%Y-%m-%d %H:%M:%S"
)
model.quantize(data, cache_examples_on_gpu=False)
```

最后，保存量化模型：

```
model.save_quantized(quant_path, use_safetensors=True)
tokenizer.save_pretrained(quant_path)
```

很遗憾，`save_quantized` 方法不支持模型分片。若要实现模型分片，您需要先加载模型，然后使用来自 `transformers` 库的 `save_pretrained` 方法来保存并分片模型。除此之外，一切操作都非常简单。祝您使用愉快！

1.10 GGUF

最近，在社区中本地运行 LLM 变得越来越流行，其中使用 llama.cpp 处理 GGUF 文件是一个典型的例子。通过 llama.cpp，您不仅可以为自己的模型构建 GGUF 文件，还可以进行低比特量化操作。在 GGUF 格式下，您可以直接对模型进行量化而无需校准过程，或者为了获得更好的量化效果应用 AWQ scale，亦或是结合校准数据使用 imatrix 工具。在这篇文档中，我们将展示最简便的模型量化方法，以及如何在对 Qwen 模型进行量化时应用 AWQ 比例以优化其质量。

1.10.1 量化你的模型并生成 GGUF 文件

在进行量化操作之前，请确保你已经按照指导开始使用 llama.cpp。以下指引将不会提供有关安装和构建的步骤。现在，假设你要对 `Qwen2-7B-Instruct` 模型进行量化，首先需要按照如下所示的方式为 fp16 模型创建一个 GGUF 文件：

```
python convert-hf-to-gguf.py Qwen/Qwen2-7B-Instruct --outfile models/7B/qwen2-7b-
↪instruct-fp16.gguf
```

其中，第一个参数指代的是预训练模型所在的路径或者 HF 模型的名称，第二个参数则指的是你想要生成的 GGUF 文件的路径（此处我将其置于 `models/7B` 目录下）。请记住，在运行命令之前，需要先创建这个目录。通过这种方式，你已经为你的 fp16 模型生成了一个 GGUF 文件，接下来你需要根据实际需求将其量化至低比特位。以下是一个将模型量化至 4 位的具体示例：

```
./quantize models/7B/qwen2-7b-instruct-fp16.gguf models/7B/qwen2-7b-instruct-q4_0.
↪gguf q4_0
```

到现在为止，您已经完成了将模型量化为 4 比特，并将其放入 GGUF 文件中。这里的 `q4_0` 表示 4 比特量化。现在，这个量化后的模型可以直接通过 llama.cpp 运行。

1.10.2 利用 AWQ scales 来量化你的模型

要提升量化模型的质量，一种可能的解决方案是应用 AWQ scales，可以参考 ‘该脚本 <https://github.com/casper-hansen/AutoAWQ/blob/main/docs/examples.md>‘__。具体操作步骤如下：首先，在使用 AutoAWQ 运行 model.quantize() 时，请务必记得添加 `export_compatible=True` 参数，如下所示：

```
...
model.quantize(
    tokenizer,
    quant_config=quant_config,
    export_compatible=True
)
model.save_pretrained(quant_path)
...
```

通过上述的 model.save_quantized()，一个带有 AWQ scales 的 fp16 模型将被保存。然后，当你运行 `convert-hf-to-gguf.py` 脚本时，请记得将模型路径替换为带有 AWQ scales 的 fp16 模型的路径，例如：

```
python convert-hf-to-gguf.py ${quant_path} --outfile models/7B/qwen2-7b-instruct-fp16-
↪awq.gguf
```

通过这种方式，您可以在 GGUF 格式的量化模型中应用 AWQ scales，这有助于提升模型的质量。

我们通常将 fp16 模型量化为 2、3、4、5、6 和 8 位模型。要执行不同低比特的量化，只需在命令中替换量化方法即可。例如，如果你想将你的模型量化为 2 位模型，你可以按照下面所示，将 `q4_0` 替换为 `q2_k`：

```
./quantize models/7B/qwen2-7b-instruct-fp16.gguf models/7B/qwen2-7b-instruct-q2_k.
↪gguf q2_k
```

我们现在提供了以下量化级别的 GGUF 模型：`q2_k`、`q3_k_m`、`q4_0`、`q4_k_m`、`q5_0`、`q5_k_m`、`q6_k` 和 `q8_0`。欲了解更多信息，请访问 llama.cpp 。

1.11 vLLM

我们建议您在部署 Qwen 时尝试使用 vLLM。它易于使用，且具有最先进的服务吞吐量、高效的注意力键值内存管理（通过 PagedAttention 实现）、连续批处理输入请求、优化的 CUDA 内核等功能。要了解更多关于 vLLM 的信息，请参阅 论文 和 文档 。

1.11.1 安装

默认情况下，你可以通过 `pip` 来安装 vLLM： `pip install vLLM>=0.4.0` ，但如果你正在使用 CUDA 11.8，请查看官方文档中的注意事项以获取有关安装的帮助（链接 ）。我们也建议你通过 `pip install ray` 安装 ray，以便支持分布式服务。

1.11.2 离线推理

Qwen2 代码支持的模型都被 vLLM 所支持。vLLM 最简单的使用方式是通过以下演示进行离线批量推理。

```
from transformers import AutoTokenizer
from vllm import LLM, SamplingParams

# Initialize the tokenizer
tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen2-7B-Instruct")

# Pass the default decoding hyperparameters of Qwen2-7B-Instruct
# max_tokens is for the maximum length for generation.
sampling_params = SamplingParams(temperature=0.7, top_p=0.8, repetition_penalty=1.05,␣
↪max_tokens=512)

# Input the model name or path. Can be GPTQ or AWQ models.
llm = LLM(model="Qwen/Qwen2-7B-Instruct")

# Prepare your prompts
prompt = "Tell me something about large language models."
messages = [
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": prompt}
]
text = tokenizer.apply_chat_template(
    messages,
    tokenize=False,
    add_generation_prompt=True
)

# generate outputs
outputs = llm.generate([text], sampling_params)

# Print the outputs.
for output in outputs:
    prompt = output.prompt
    generated_text = output.outputs[0].text
    print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")
```

1.11.3 适配 OpenAI-API 的 API 服务

借助 vLLM，构建一个与 OpenAI API 兼容的 API 服务十分简便，该服务可以作为实现 OpenAI API 协议的服务器进行部署。默认情况下，它将在 http://localhost:8000 启动服务器。您可以通过 `--host` 和 `--port` 参数来自定义地址。请按照以下所示运行命令：

```
python -m vllm.entrypoints.openai.api_server \
    --model Qwen/Qwen2-7B-Instruct
```

你无需担心 chat 模板，因为它默认会使用由 tokenizer 提供的 chat 模板。

然后，您可以利用 create chat interface 来与 Qwen 进行对话：

```
curl http://localhost:8000/v1/chat/completions -H "Content-Type: application/json" -d
↪'{
    "model": "Qwen/Qwen2-7B-Instruct",
    "messages": [
```

(续下页)

(接上页)

```
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": "Tell me something about large language models."}
    ]
    }'
```

或者您可以如下面所示使用 `openai` Python 包中的 Python 客户端：

```
from openai import OpenAI
# Set OpenAI's API key and API base to use vLLM's API server.
openai_api_key = "EMPTY"
openai_api_base = "http://localhost:8000/v1"

client = OpenAI(
    api_key=openai_api_key,
    base_url=openai_api_base,
)

chat_response = client.chat.completions.create(
    model="Qwen/Qwen2-7B-Instruct",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Tell me something about large language models."},
    ]
)
print("Chat response:", chat_response)
```

1.11.4 多卡分布式部署

要提高模型的处理吞吐量，分布式服务可以通过利用更多的 GPU 设备来帮助您。特别是对于像 `Qwen2-72B-Instruct` 这样的大模型，单个 GPU 无法支撑其在线服务。在这里，我们通过演示如何仅通过传入参数 `tensor_parallel_size` ，来使用张量并行来运行 `Qwen2-72B-Instruct` 模型：

```
from vllm import LLM, SamplingParams
llm = LLM(model="Qwen/Qwen2-72B-Instruct", tensor_parallel_size=4)
```

您可以通过传递参数 `--tensor-parallel-size` 来运行多 GPU 服务：

```
python -m vllm.entrypoints.api_server \
    --model Qwen/Qwen2-72B-Instruct \
    --tensor-parallel-size 4
```

1.11.5 部署量化模型

vLLM 支持多种类型的量化模型，例如 AWQ、GPTQ、SqueezeLLM 等。这里我们将展示如何部署 AWQ 和 GPTQ 模型。使用方法与上述基本相同，只不过需要额外指定一个量化参数。例如，要运行一个 AWQ 模型，例如 `Qwen2-72B-Instruct-AWQ`：

```
from vllm import LLM, SamplingParams
llm = LLM(model="Qwen/Qwen2-72B-Instruct-AWQ", quantization="awq")
```

或者是 GPTQ 模型比如 `Qwen2-72B-Instruct-GPTQ-Int8`：

```
llm = LLM(model="Qwen/Qwen2-72B-Instruct-GPTQ-Int4", quantization="gptq")
```

同样地，您可以在运行服务时添加 `--quantization` 参数，如下所示：

```
python -m vllm.entrypoints.openai.api_server \
    --model Qwen/Qwen2-72B-Instruct-AWQ \
    --quantization awq
```

或者

```
python -m vllm.entrypoints.openai.api_server \
    --model Qwen/Qwen2-72B-Instruct-GPTQ-Int8 \
    --quantization gptq
```

此外，vLLM 支持将 AWQ 或 GPTQ 模型与 KV 缓存量化相结合，即 FP8 E5M2 KV Cache 方案。例如：

```
llm = LLM(model="Qwen/Qwen2-7B-Instruct-GPTQ-Int8", quantization="gptq", kv_cache_
↪dtype="fp8_e5m2")
```

```
python -m vllm.entrypoints.openai.api_server \
    --model Qwen/Qwen2-7B-Instruct-GPTQ-Int8 \
    --quantization gptq \
    --kv-cache-dtype fp8_e5m2
```

1.11.6 常见问题

您可能会遇到令人烦恼的 OOM（内存溢出）问题。我们推荐您尝试两个参数进行修复。第一个参数是 `--max-model-len` 。我们提供的默认最大位置嵌入（max_position_embedding）为 32768，因此服务时的最大长度也是这个值，这会导致更高的内存需求。将此值适当减小通常有助于解决 OOM 问题。另一个您可以关注的参数是 `--gpu-memory-utilization` 。默认情况下，该值为 `0.9` ，您可以将其调高以应对 OOM 问题。这也是为什么您发现一个大型语言模型服务总是占用大量内存的原因。

1.12 TGI

Hugging Face 的 Text Generation Inference (TGI) 是一个专为部署大规模语言模型 (Large Language Models, LLMs) 而设计的生产级框架。TGI 提供了流畅的部署体验，并稳定支持如下特性：

- *推测解码 (Speculative Decoding)*： 提升生成速度。
- 张量并行 (Tensor Parallelism)： 高效多卡部署。
- 流式生成 (Token Streaming)： 支持持续性生成文本。
- 灵活的硬件支持：与 AMD ， Gaudi 和 AWS Inferentia 无缝衔接。

1.12.1 安装

通过 TGI docker 镜像使用 TGI 轻而易举。本文将主要介绍 TGI 的 docker 用法。

也可通过 Conda 实机安装或搭建服务。请参考 Installation Guide 与 CLI tool 以了解详细说明。

1.12.2 通过 TGI 部署 Qwen2

1. **选定 Qwen2 模型：** 从 the Qwen2 collection 中挑选模型。
2. **部署 TGI 服务：** 在终端中运行以下命令，注意替换 `model` 为选定的 Qwen2 模型 ID 、`volume` 为本地的数据路径：

```
model=Qwen/Qwen2-7B-Instruct
volume=$PWD/data # share a volume with the Docker container to avoid downloading␣
↪weights every run

docker run --gpus all --shm-size 1g -p 8080:80 -v $volume:/data ghcr.io/huggingface/
↪text-generation-inference:2.0 --model-id $model
```

1.12.3 使用 TGI API

一旦成功部署，API 将于选定的映射端口 (8080) 提供服务。

TGI 提供了简单直接的 API 支持流式生成：

```
curl http://localhost:8080/generate_stream -H 'Content-Type: application/json' \
        -d '{"inputs":"Tell me something about large language models.","parameters":{
↪"max_new_tokens":512}}'
```

也可使用 OpenAI 风格的 API 使用 TGI：

```
curl http://localhost:8080/v1/chat/completions -H "Content-Type: application/json" -d
↪'{
      "model": "",
      "messages": [
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": "Tell me something about large language␣
↪models."}
      ],
      "max_tokens": 512
}'
```

注意：json 中的 model 字段不会被 TGI 识别，您可传入任意值。完整 API 文档，请查阅 TGI Swagger UI 。

你也可以使用 Python 访问 API：

```
from openai import OpenAI

# initialize the client but point it to TGI
client = OpenAI(
   base_url="http://localhost:8080/v1/",  # replace with your endpoint url
   api_key="",  # this field is not used when running locally
)
chat_completion = client.chat.completions.create(
   model="",  # it is not used by TGI, you can put anything
```

(续下页)

(接上页)

```
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Tell me something about large language models."},
    ],
    stream=True,
    max_tokens=512,
)

# iterate and print stream
for message in chat_completion:
    print(message.choices[0].delta.content, end="")
```

1.12.4 量化

1. 依赖数据的量化方案（GPTQ 与 AWQ ）

GPTQ 与 AWQ 均依赖数据进行量化。我们提供了预先量化好的模型，请于 the Qwen2 collection 查找。你也可以使用自己的数据集自行量化，以在你的场景中取得更好效果。

以下是通过 TGI 部署 Qwen2-7B-Instruct-GPTQ-Int4 的指令：

```
model=Qwen/Qwen2-7B-Instruct-GPTQ-Int4
volume=$PWD/data # share a volume with the Docker container to avoid downloading␣
↪weights every run

docker run --gpus all --shm-size 1g -p 8080:80 -v $volume:/data ghcr.io/huggingface/
↪text-generation-inference:2.0 --model-id $model --quantize gptq
```

如果模型是 AWQ 量化的，如 Qwen/Qwen2-7B-Instruct-AWQ ，请使用 `--quantize awq` 。

2. 不依赖数据的量化方案

EETQ 是一种不依赖数据的量化方案，可直接用于任意模型。请注意，我们需要传入原始模型，并使用 `--quantize eetq` 标志。

```
model=Qwen/Qwen2-7B-Instruct
volume=$PWD/data # share a volume with the Docker container to avoid downloading␣
↪weights every run

docker run --gpus all --shm-size 1g -p 8080:80 -v $volume:/data ghcr.io/huggingface/
↪text-generation-inference:2.0 --model-id $model --quantize eetq
```

3. 延迟指标

以下为 Qwen2-7B-Instruct 量化模型在 4090 上的 token/s 结果：

- GPTQ int4: 6.8ms
- AWQ int4: 7.9ms
- EETQ int8: 9.7ms

1.12.5 多卡部署

使用 `--num-shard` 指定卡书数量。请务必传入 `--shm-size 1g` 让 NCCL 发挥最好性能 (说明)：

```
model=Qwen/Qwen2-7B-Instruct
volume=$PWD/data # share a volume with the Docker container to avoid downloading␣
↪weights every run

docker run --gpus all --shm-size 1g -p 8080:80 -v $volume:/data ghcr.io/huggingface/
↪text-generation-inference:2.0 --model-id $model --num-shard 2
```

1.12.6 推测性解码 (Speculative Decoding)

推测性解码 (Speculative Decoding) 通过预先推测下一 token 来节约每 token 需要的时间。使用 `--speculative-decoding` 设定预先推测 token 的数量（默认为 0，表示不预先推测)：

```
model=Qwen/Qwen2-7B-Instruct
volume=$PWD/data # share a volume with the Docker container to avoid downloading␣
↪weights every run

docker run --gpus all --shm-size 1g -p 8080:80 -v $volume:/data ghcr.io/huggingface/
↪text-generation-inference:2.0 --model-id $model --speculate 2
```

以下是 Qwen2-7B-Instruct 在 4090 GPU 上的实测指标：

- 无预先推测（默认)：17.4ms
- 预先推测 (`n=2`)：16.6ms

本例为代码生成，推测性解码相比于默认配置可加速 10%。推测性解码的加速效果依赖于任务类型，对于代码或重复性较高的文本生成任务，提速更明显。

更多说明可查阅 此文档 。

1.12.7 使用 HF Inference Endpoints 零代码部署

使用 Hugging Face Inference Endpoints 不费吹灰之力：

- **GUI interface:** https://huggingface.co/inference-endpoints/dedicated
- **Coding interface:** https://huggingface.co/blog/tgi-messages-api

一旦部署成功，服务使用与本地无异。

1.12.8 常见问题

Qwen2 支持长上下文，谨慎设定 `--max-batch-prefill-tokens` ，`--max-total-tokens` 和 `--max-input-tokens` 以避免 out-of-memory (OOM) 。如 OOM ，你将在启动 TGI 时收到错误提示。以下为修改这些参数的示例：

```
model=Qwen/Qwen2-7B-Instruct
volume=$PWD/data # share a volume with the Docker container to avoid downloading␣
↪weights every run

docker run --gpus all --shm-size 1g -p 8080:80 -v $volume:/data ghcr.io/huggingface/
```

(续下页)

(接上页)

```
↪text-generation-inference:2.0 --model-id $model --max-batch-prefill-tokens 4096 --
↪max-total-tokens 4096 --max-input-tokens 2048
```

1.13 SkyPilot

1.13.1 SkyPilot 是什么

SkyPilot 是一个可以在任何云上运行 LLM 、AI 应用以及批量任务的框架，旨在实现最大程度的成本节省、最高的 GPU 可用性以及受管理的执行过程。其特性包括：

- 通过跨区域和跨云充分利用多个资源池，以获得最佳的 GPU 可用性。
- 把费用降到最低——SkyPilot 在各区域和云平台中为您挑选最便宜的资源。无需任何托管解决方案的额外加价。
- 将服务扩展到多个副本上，所有副本通过单一 endpoint 对外提供服务
- 所有内容均保存在您的云账户中（包括您的虚拟机和 bucket ）
- 完全私密 - 没有其他人能看到您的聊天记录

1.13.2 安装 SkyPilot

我们建议您按照 指示 安装 SkyPilot 。以下为您提供了一个使用 `pip` 进行安装的简单示例：

```
# You can use any of the following clouds that you have access to:
# aws, gcp, azure, oci, lamabda, runpod, fluidstack, paperspace,
# cudo, ibm, scp, vsphere, kubernetes
pip install "skypilot-nightly[aws,gcp]"
```

随后，您需要用如下命令确认是否能使用云：

```
sky check
```

若需更多信息，请查阅官方文档，确认您的云账户设置是否正确无误。

或者，您也可以使用官方提供的 docker 镜像，可以自动克隆 SkyPilot 的主分支：

```
# NOTE: '--platform linux/amd64' is needed for Apple silicon Macs
docker run --platform linux/amd64 \
  -td --rm --name sky \
  -v "$HOME/.sky:/root/.sky:rw" \
  -v "$HOME/.aws:/root/.aws:rw" \
  -v "$HOME/.config/gcloud:/root/.config/gcloud:rw" \
  berkeleyskypilot/skypilot-nightly

docker exec -it sky /bin/bash
```

1.13.3 使用 SkyPilot 运行 Qwen2-72B-Instruct

1. serve-72b.yaml 中列出了支持的 GPU 。您可使用配备这类 GPU 的单个运算实例来部署 Qwen2-72B-Instruct 服务。该服务由 vLLM 搭建，并与 OpenAI API 兼容。以下为部署方法：

```
sky launch -c qwen serve-72b.yaml
```

2. 向该 endpoint 发送续写请求：

```
IP=$(sky status --ip qwen)

curl -L http://$IP:8000/v1/completions \
    -H "Content-Type: application/json" \
    -d '{
      "model": "Qwen/Qwen2-72B-Instruct",
      "prompt": "My favorite food is",
      "max_tokens": 512
  }' | jq -r '.choices[0].text'
```

3. 向该 endpoint 发送对话续写请求

```
curl -L http://$IP:8000/v1/chat/completions \
    -H "Content-Type: application/json" \
    -d '{
      "model": "Qwen/Qwen2-72B-Instruct",
      "messages": [
        {
          "role": "system",
          "content": "You are a helpful and honest chat expert."
        },
        {
          "role": "user",
          "content": "What is the best food?"
        }
      ],
      "max_tokens": 512
  }' | jq -r '.choices[0].message.content'
```

1.13.4 使用 SkyPilot Serve 扩展服务规模

1. 使用 SkyPilot Serve 扩展 Qwen 的服务规模非常容易，只需运行：

```
sky serve up -n qwen ./serve-72b.yaml
```

这将启动服务，使用多个副本部署在最经济的可用位置和加速器上。SkyServe 将自动管理这些副本，监控其健康状况，根据负载进行自动伸缩，并在必要时重启它们。

将返回一个 endpoint ，所有发送至该 endpoint 的请求都将被路由至就绪状态的副本。

2. 运行如下命令检查服务的状态：

```
sky serve status qwen
```

很快，您将看到如下输出：

```
Services
NAME        VERSION  UPTIME  STATUS         REPLICAS  ENDPOINT
Qwen  1         -       READY          2/2       3.85.107.228:30002

Service Replicas
SERVICE_NAME  ID  VERSION  IP  LAUNCHED    RESOURCES                    STATUS REGION
Qwen          1   1        -   2 mins ago  1x Azure({'A100-80GB': 8}) READY  eastus
Qwen          2   1        -   2 mins ago  1x GCP({'L4': 8})          READY  us-east4-
↪a
```

如下所示：该服务现由两个副本提供支持，一个位于 Azure 平台，另一个位于 GCP 平台。同时，已为服务选择云服务商提供的 **最经济实惠**的加速器类型。这样既最大限度地提升了服务的可用性，又尽可能降低了成本。

3. 要访问模型，我们使用带有 `curl -L`（用于跟随重定向），将请求发送到 endpoint：

```
ENDPOINT=$(sky serve status --endpoint qwen)

curl -L http://$ENDPOINT/v1/chat/completions \
    -H "Content-Type: application/json" \
    -d '{
      "model": "Qwen/Qwen2-72B-Instruct",
      "messages": [
        {
          "role": "system",
          "content": "You are a helpful and honest code assistant expert in Python."
        },
        {
          "role": "user",
          "content": "Show me the python code for quick sorting a list of integers."
        }
      ],
      "max_tokens": 512
  }' | jq -r '.choices[0].message.content'
```

1.13.5 使用 Chat GUI 调用 Qwen2

可以通过 FastChat 来使用 GUI 调用 Qwen2 的服务：

1. 开启一个 Chat Web UI

```
sky launch -c qwen-gui ./gui.yaml --env ENDPOINT=$(sky serve status --endpoint qwen)
```

2. 随后，我们可以通过返回的 gradio 链接来访问 GUI：

```
| INFO | stdout | Running on public URL: https://6141e84201ce0bb4ed.gradio.live
```

你可以通过使用不同的温度和 top_p 值来尝试取得更好的结果。

1.13.6 总结

通过 SkyPilot，你可以轻松地在任何云上部署 Qwen2 。我们建议您阅读 官方文档 了解更多用法和最新进展。

1.14 有监督微调

1.14.1 示例

在这里，我们提供了一个非常简单的有监督微调脚本，该脚本是基于 Fastchat 的训练脚本修改而来的。这个脚本用于使用 Hugging Face Trainer 对 Qwen 模型进行微调。你可以在以下链接查看这个脚本：这里 。这个脚本具有以下特点：

- 支持单卡和多卡分布式训练
- 支持全参数微调、LoRA 以及 Q-LoRA 。

下面，我们介绍脚本的更多细节。

安装

开始之前，确保你已经安装了以下代码库：

```
pip install peft deepspeed optimum accelerate
```

准备数据

我们建议你放在 jsonl 文件中，每一行是一个如下所示的字典：

```
{
    "type": "chatml",
    "messages": [
        {
            "role": "system",
            "content": "You are a helpful assistant."
        },
        {
            "role": "user",
            "content": "Tell me something about large language models."
        },
        {
            "role": "assistant",
            "content": "Large language models are a type of language model that is␣
↪trained on a large corpus of text data. They are capable of generating human-like␣
↪text and are used in a variety of natural language processing tasks..."
        }
    ],
    "source": "unknown"
}
```

```
{
    "type": "chatml",
    "messages": [
```

(续下页)

(接上页)

```
        {
            "role": "system",
            "content": "You are a helpful assistant."
        },
        {
            "role": "user",
            "content": "What is your name?"
        },
        {
            "role": "assistant",
            "content": "My name is Qwen."
        }
    ],
    "source": "self-made"
}
```

以上提供了该数据集中的每个样本的两个示例。每个样本都是一个 JSON 对象，包含以下字段：`type` 、`messages` 和 `source` 。其中，`messages` 是必填字段，而其他字段则是供您标记数据格式和数据来源的可选字段。`messages` 字段是一个 JSON 对象列表，每个对象都包含两个字段：`role` 和 `content` 。其中，`role` 可以是 `system` 、`user` 或 `assistant` ，表示消息的角色；`content` 则是消息的文本内容。而 `source` 字段代表了数据来源，可能包括 `self-made` 、`alpaca` 、`open-hermes` 或其他任意字符串。

你需要用 `json` 将一个字典列表存入 jsonl 文件中：

```
import json

with open('data.jsonl', 'w') as f:
    for sample in samples:
        f.write(json.dumps(sample) + '\n')
```

快速开始

为了让您能够快速开始微调，我们直接提供了一个 shell 脚本，您可以无需关注具体细节即可运行。针对不同类型的训练（例如单 GPU 训练、多 GPU 训练、全参数微调、LoRA 或 Q-LoRA），您可能需要不同的超参数设置。

```
cd examples/sft
bash finetune.sh -m <model_path> -d <data_path> --deepspeed <config_path> [--use_lora␣
↪True] [--q_lora True]
```

为您的模型指定 `<model_path>` ，为您的数据指定 `<data_path>` ，并为您的 Deepspeed 配置指定 `<config_path>` 。如果您使用 LoRA 或 Q-LoRA，只需根据您的需求添加 `--use_lora True` 或 `--q_lora True` 。这是开始微调的最简单方式。如果您想更改更多超参数，您可以深入脚本并修改这些参数。

高级用法

在这个部分中，我们介绍 python 脚本以及 shell 脚本的相关细节

Shell 脚本

在展示 Python 代码之前，我们先对包含命令的 Shell 脚本做一个简单的介绍。我们在 Shell 脚本中提供了一些指南，并且此处将以 `finetune.sh` 这个脚本为例进行解释说明。

要为分布式训练(或单 GPU 训练)设置环境变量，请指定以下变量：`GPUS_PER_NODE`、`NNODES`、`NODE_RANK`、`MASTER_ADDR` 和 `MASTER_PORT`。不必过于担心这些变量，因为我们为您提供了默认设置。在命令行中，您可以通过传入参数 `-m` 和 `-d` 来分别指定模型路径和数据路径。您还可以通过传入参数 `--deepspeed` 来指定 Deepspeed 配置文件。我们为您提供针对 ZeRO2 和 ZeRO3 的两种配置文件，您可以根据需求选择其中之一。在大多数情况下，我们建议在多 GPU 训练中使用 ZeRO3，但针对 Q-LoRA，我们推荐使用 ZeRO2。

有一系列需要调节的超参数。您可以向程序传递 `--bf16` 或 `--fp16` 参数来指定混合精度训练所采用的精度级别。此外，还有其他一些重要的超参数如下：

- `--output_dir`: the path of your output models or adapters.
- `--num_train_epochs`: the number of training epochs.
- `--gradient_accumulation_steps`: the number of gradient accumulation steps.
- `--per_device_train_batch_size`: the batch size per GPU for training, and the total batch size is equalt to `per_device_train_batch_size` × `number_of_gpus` × `gradient_accumulation_steps`.
- `--learning_rate`: the learning rate.
- `--warmup_steps`: the number of warmup steps.
- `--lr_scheduler_type`: the type of learning rate scheduler.
- `--weight_decay`: the value of weight decay.
- `--adam_beta2`: the value of β_2 in Adam.
- `--model_max_length`: the maximum sequence length.
- `--use_lora`: whether to use LoRA. Adding `--q_lora` can enable Q-LoRA.
- `--gradient_checkpointing`: whether to use gradient checkpointing.

Python 脚本

在本脚本中，我们主要使用来自 HF 的 `trainer` 和 `peft` 来训练我们的模型。同时，我们也利用 `deepspeed` 来加速训练过程。该脚本非常简洁且易于理解。

```
@dataclass
@dataclass
class ModelArguments:
    model_name_or_path: Optional[str] = field(default="Qwen/Qwen2-7B")


@dataclass
class DataArguments:
    data_path: str = field(
        default=None, metadata={"help": "Path to the training data."}
    )
```

(续下页)

(接上页)

```
    eval_data_path: str = field(
        default=None, metadata={"help": "Path to the evaluation data."}
    )
    lazy_preprocess: bool = False


@dataclass
class TrainingArguments(transformers.TrainingArguments):
    cache_dir: Optional[str] = field(default=None)
    optim: str = field(default="adamw_torch")
    model_max_length: int = field(
        default=8192,
        metadata={
            "help": "Maximum sequence length. Sequences will be right padded (and␣
↪possibly truncated)."
        },
    )
    use_lora: bool = False


@dataclass
class LoraArguments:
    lora_r: int = 64
    lora_alpha: int = 16
    lora_dropout: float = 0.05
    lora_target_modules: List[str] = field(
        default_factory=lambda: [
            "q_proj",
            "k_proj",
            "v_proj",
            "o_proj",
            "up_proj",
            "gate_proj",
            "down_proj",
        ]
    )
    lora_weight_path: str = ""
    lora_bias: str = "none"
    q_lora: bool = False
```

参数类允许你为模型、数据和训练指定超参数，如果使用 LoRA 或 Q-LoRA 训练模型，还会包含这两个方法的相关超参数。具体来说，`model-max-length` 是一个关键的超参数，它决定了训练数据的最大序列长度。

`LoRAArguments` includes the hyperparameters for LoRA or Q-LoRA:

- `lora_r`: the rank for LoRA;
- `lora_alpha`: the alpha value for LoRA;
- `lora_dropout`: the dropout rate for LoRA;
- `lora_target_modules`: the target modules for LoRA. By default we tune all linear layers;
- `lora_weight_path`: the path to the weight file for LoRA;
- `lora_bias`: the bias for LoRA;
- `q_lora`: whether to use Q-LoRA.

```
def maybe_zero_3(param):
    if hasattr(param, "ds_id"):
        assert param.ds_status == ZeroParamStatus.NOT_AVAILABLE
        with zero.GatheredParameters([param]):
            param = param.data.detach().cpu().clone()
    else:
        param = param.detach().cpu().clone()
    return param


# Borrowed from peft.utils.get_peft_model_state_dict
def get_peft_state_maybe_zero_3(named_params, bias):
    if bias == "none":
        to_return = {k: t for k, t in named_params if "lora_" in k}
    elif bias == "all":
        to_return = {k: t for k, t in named_params if "lora_" in k or "bias" in k}
    elif bias == "lora_only":
        to_return = {}
        maybe_lora_bias = {}
        lora_bias_names = set()
        for k, t in named_params:
            if "lora_" in k:
                to_return[k] = t
                bias_name = k.split("lora_")[0] + "bias"
                lora_bias_names.add(bias_name)
            elif "bias" in k:
                maybe_lora_bias[k] = t
        for k, t in maybe_lora_bias:
            if bias_name in lora_bias_names:
                to_return[bias_name] = t
    else:
        raise NotImplementedError
    to_return = {k: maybe_zero_3(v) for k, v in to_return.items()}
    return to_return


def safe_save_model_for_hf_trainer(
    trainer: transformers.Trainer, output_dir: str, bias="none"
):
    """Collects the state dict and dump to disk."""
    # check if zero3 mode enabled
    if deepspeed.is_deepspeed_zero3_enabled():
        state_dict = trainer.model_wrapped._zero3_consolidated_16bit_state_dict()
    else:
        if trainer.args.use_lora:
            state_dict = get_peft_state_maybe_zero_3(
                trainer.model.named_parameters(), bias
            )
        else:
            state_dict = trainer.model.state_dict()
    if trainer.args.should_save and trainer.args.local_rank == 0:
        trainer._save(output_dir, state_dict=state_dict)
```

方法 `safe_save_model_for_hf_trainer` 通过使用 `get_peft_state_maybe_zero_3` 有助于解决在保存采用或未采用 ZeRO3 技术训练的模型时遇到的问题。

```
def preprocess(
```

(续下页)

(接上页)

```
    messages,
    tokenizer: transformers.PreTrainedTokenizer,
    max_len: int,
) -> Dict:
    """Preprocesses the data for supervised fine-tuning."""

    texts = []
    for i, msg in enumerate(messages):
        texts.append(
            tokenizer.apply_chat_template(
                msg,
                tokenize=True,
                add_generation_prompt=False,
                padding=True,
                max_length=max_len,
                truncation=True,
            )
        )
    input_ids = torch.tensor(texts, dtype=torch.int)
    target_ids = input_ids.clone()
    target_ids[target_ids == tokenizer.pad_token_id] = IGNORE_TOKEN_ID
    attention_mask = input_ids.ne(tokenizer.pad_token_id)

    return dict(
        input_ids=input_ids, target_ids=target_ids, attention_mask=attention_mask
    )
```

在数据预处理阶段，我们使用 `preprocess` 来整理数据。具体来说，我们应用 ChatML 模板对文本进行处理。如果您倾向于使用其他 chat 模板，您也可以选择其他的，例如，仍然通过 “apply_chat_template()“函数配合另一个 tokenizer 进行应用。Chat 模板存储在 HF 仓库中的 `tokenizer_config.json` 文件中。此外，我们还将每个样本的序列填充到最大长度，以便于训练。

```
class SupervisedDataset(Dataset):
    """Dataset for supervised fine-tuning."""

    def __init__(
        self, raw_data, tokenizer: transformers.PreTrainedTokenizer, max_len: int
    ):
        super(SupervisedDataset, self).__init__()

        rank0_print("Formatting inputs...")
        messages = [example["messages"] for example in raw_data]
        data_dict = preprocess(messages, tokenizer, max_len)

        self.input_ids = data_dict["input_ids"]
        self.target_ids = data_dict["target_ids"]
        self.attention_mask = data_dict["attention_mask"]

    def __len__(self):
        return len(self.input_ids)

    def __getitem__(self, i) -> Dict[str, torch.Tensor]:
        return dict(
            input_ids=self.input_ids[i],
            labels=self.labels[i],
            attention_mask=self.attention_mask[i],
```

(续下页)

(接上页)

```
        )


class LazySupervisedDataset(Dataset):
    """Dataset for supervised fine-tuning."""

    def __init__(
        self, raw_data, tokenizer: transformers.PreTrainedTokenizer, max_len: int
    ):
        super(LazySupervisedDataset, self).__init__()
        self.tokenizer = tokenizer
        self.max_len = max_len

        rank0_print("Formatting inputs...Skip in lazy mode")
        self.tokenizer = tokenizer
        self.raw_data = raw_data
        self.cached_data_dict = {}

    def __len__(self):
        return len(self.raw_data)

    def __getitem__(self, i) -> Dict[str, torch.Tensor]:
        if i in self.cached_data_dict:
            return self.cached_data_dict[i]

        ret = preprocess([self.raw_data[i]["messages"]], self.tokenizer, self.max_len)
        ret = dict(
            input_ids=ret["input_ids"][0],
            labels=ret["target_ids"][0],
            attention_mask=ret["attention_mask"][0],
        )
        self.cached_data_dict[i] = ret

        return ret


def make_supervised_data_module(
    tokenizer: transformers.PreTrainedTokenizer,
    data_args,
    max_len,
) -> Dict:
    """Make dataset and collator for supervised fine-tuning."""
    dataset_cls = (
        LazySupervisedDataset if data_args.lazy_preprocess else SupervisedDataset
    )
    rank0_print("Loading data...")

    train_data = []
    with open(data_args.data_path, "r") as f:
        for line in f:
            train_data.append(json.loads(line))
    train_dataset = dataset_cls(train_data, tokenizer=tokenizer, max_len=max_len)

    if data_args.eval_data_path:
        eval_data = []
        with open(data_args.eval_data_path, "r") as f:
            for line in f:
```

(续下页)

(接上页)

```
                eval_data.append(json.loads(line))
        eval_dataset = dataset_cls(eval_data, tokenizer=tokenizer, max_len=max_len)
    else:
        eval_dataset = None

    return dict(train_dataset=train_dataset, eval_dataset=eval_dataset)
```

然后我们利用 make_supervised_data_module ，通过使用 SupervisedDataset 或 LazySupervisedDataset 来构建数据集。

```
def train():
    global local_rank

    parser = transformers.HfArgumentParser(
        (ModelArguments, DataArguments, TrainingArguments, LoraArguments)
    )
    (
        model_args,
        data_args,
        training_args,
        lora_args,
    ) = parser.parse_args_into_dataclasses()

    # This serves for single-gpu qlora.
    if (
        getattr(training_args, "deepspeed", None)
        and int(os.environ.get("WORLD_SIZE", 1)) == 1
    ):
        training_args.distributed_state.distributed_type = DistributedType.DEEPSPEED

    local_rank = training_args.local_rank

    device_map = None
    world_size = int(os.environ.get("WORLD_SIZE", 1))
    ddp = world_size != 1
    if lora_args.q_lora:
        device_map = {"": int(os.environ.get("LOCAL_RANK") or 0)} if ddp else "auto"
        if len(training_args.fsdp) > 0 or deepspeed.is_deepspeed_zero3_enabled():
            logging.warning("FSDP or ZeRO3 is incompatible with QLoRA.")

    model_load_kwargs = {
        "low_cpu_mem_usage": not deepspeed.is_deepspeed_zero3_enabled(),
    }

    compute_dtype = (
        torch.float16
        if training_args.fp16
        else (torch.bfloat16 if training_args.bf16 else torch.float32)
    )

    # Load model and tokenizer
    config = transformers.AutoConfig.from_pretrained(
        model_args.model_name_or_path,
        cache_dir=training_args.cache_dir,
    )
    config.use_cache = False
```

(续下页)

(接上页)

```
    model = AutoModelForCausalLM.from_pretrained(
        model_args.model_name_or_path,
        config=config,
        cache_dir=training_args.cache_dir,
        device_map=device_map,
        quantization_config=BitsAndBytesConfig(
            load_in_4bit=True,
            bnb_4bit_use_double_quant=True,
            bnb_4bit_quant_type="nf4",
            bnb_4bit_compute_dtype=compute_dtype,
        )
        if training_args.use_lora and lora_args.q_lora
        else None,
        **model_load_kwargs,
    )
    tokenizer = AutoTokenizer.from_pretrained(
        model_args.model_name_or_path,
        cache_dir=training_args.cache_dir,
        model_max_length=training_args.model_max_length,
        padding_side="right",
        use_fast=False,
    )

    if training_args.use_lora:
        lora_config = LoraConfig(
            r=lora_args.lora_r,
            lora_alpha=lora_args.lora_alpha,
            target_modules=lora_args.lora_target_modules,
            lora_dropout=lora_args.lora_dropout,
            bias=lora_args.lora_bias,
            task_type="CAUSAL_LM",
        )
        if lora_args.q_lora:
            model = prepare_model_for_kbit_training(
                model, use_gradient_checkpointing=training_args.gradient_checkpointing
            )

        model = get_peft_model(model, lora_config)

        # Print peft trainable params
        model.print_trainable_parameters()

        if training_args.gradient_checkpointing:
            model.enable_input_require_grads()

    # Load data
    data_module = make_supervised_data_module(
        tokenizer=tokenizer, data_args=data_args, max_len=training_args.model_max_
↪length
    )

    # Start trainer
    trainer = Trainer(
        model=model, tokenizer=tokenizer, args=training_args, **data_module
    )
```

(续下页)

(接上页)

```
    # `not training_args.use_lora` is a temporary workaround for the issue that there␣
↪are problems with
    # loading the checkpoint when using LoRA with DeepSpeed.
    # Check this issue https://github.com/huggingface/peft/issues/746 for more␣
↪information.
    if (
        list(pathlib.Path(training_args.output_dir).glob("checkpoint-*"))
        and not training_args.use_lora
    ):
        trainer.train(resume_from_checkpoint=True)
    else:
        trainer.train()
    trainer.save_state()

    safe_save_model_for_hf_trainer(
        trainer=trainer, output_dir=training_args.output_dir, bias=lora_args.lora_bias
    )
```

`train` 方法是训练过程的关键所在。通常情况下，它会通过 `AutoTokenizer.from_pretrained()` 和 `AutoModelForCausalLM.from_pretrained()` 来加载 tokenizer 和模型。如果使用了 LoRA，该方法将会用 `LoraConfig` 初始化 LoRA 配置。若应用 Q-LoRA，则应当采用 `prepare_model_for_kbit_training` 。请注意，目前还不支持对 LoRA 的续训（resume）。接下来的任务就交给 trainer 处理，此时不妨喝杯咖啡稍作休息！

下一步

现在，您可以使用一个非常简单的脚本来执行不同类型的 SFT。另外，您还可以选择使用更高级的训练库，例如 Axolotl 或 LLaMA-Factory ，以享受更多的功能。进一步来说，在完成 SFT 之后，您可以考虑采用 RLHF（强化学习与人类反馈）来使您的模型更好地与人类偏好对齐！请继续关注我们接下来关于 RLHF 的教程！

1.14.2 LLaMA-Factory

我们将介绍如何使用 LLaMA-Factory 微调 Qwen2 模型。本脚本包含如下特点：

- 支持单卡和多卡分布式训练
- 支持全参数微调、LoRA、Q-LoRA 和 DoRA 。

下文将介绍更多关于脚本的用法。

安装

开始之前，确保你已经安装了以下代码库：

1. 根据 LLaMA-Factory 官方指引构建好你的环境
2. 安装下列代码库（可选）：

```
pip install deepspeed
pip install flash-attn --no-build-isolation
```

3. 如你使用 FlashAttention-2 ，请确保你的 CUDA 版本在 11.6 以上。

准备数据

LLaMA-Factory 在 data 文件夹中提供了多个训练数据集，您可以直接使用它们。如果您打算使用自定义数据集，请按照以下方式准备您的数据集。

1. 请将您的数据以 json 格式进行组织，并将数据放入 data 文件夹中。LLaMA-Factory 支持以 alpaca 或 sharegpt 格式的数据集。

- alpaca 格式的数据集应遵循以下格式：

```
[
  {
    "instruction": "user instruction (required)",
    "input": "user input (optional)",
    "output": "model response (required)",
    "system": "system prompt (optional)",
    "history": [
      ["user instruction in the first round (optional)", "model response in the first␣
↪round (optional)"],
      ["user instruction in the second round (optional)", "model response in the␣
↪second round (optional)"]
    ]
  }
]
```

- sharegpt 格式的数据集应遵循以下格式：

```
[
  {
    "conversations": [
      {
        "from": "human",
        "value": "user instruction"
      },
      {
        "from": "gpt",
        "value": "model response"
      }
    ],
    "system": "system prompt (optional)",
    "tools": "tool description (optional)"
  }
]
```

2. 在 data/dataset_info.json 文件中提供您的数据集定义，并采用以下格式：

- 对于 alpaca 格式的数据集，其 dataset_info.json 文件中的列应为：

```
"dataset_name": {
  "file_name": "dataset_name.json",
  "columns": {
    "prompt": "instruction",
    "query": "input",
    "response": "output",
    "system": "system",
    "history": "history"
  }
}
```

- 对于 sharegpt 格式的数据集，dataset_info.json 文件中的列应该包括：

```
"dataset_name": {
    "file_name": "dataset_name.json",
    "formatting": "sharegpt",
    "columns": {
      "messages": "conversations",
      "system": "system",
      "tools": "tools"
    },
    "tags": {
      "role_tag": "from",
      "content_tag": "value",
      "user_tag": "user",
      "assistant_tag": "assistant"
    }
  }
```

训练

执行下列命令：

```
DISTRIBUTED_ARGS="
    --nproc_per_node $NPROC_PER_NODE \
    --nnodes $NNODES \
    --node_rank $NODE_RANK \
    --master_addr $MASTER_ADDR \
    --master_port $MASTER_PORT
  "

torchrun $DISTRIBUTED_ARGS src/train.py \
    --deepspeed $DS_CONFIG_PATH \
    --stage sft \
    --do_train \
    --use_fast_tokenizer \
    --flash_attn \
    --model_name_or_path $MODEL_PATH \
    --dataset your_dataset \
    --template qwen \
    --finetuning_type lora \
    --lora_target q_proj,v_proj\
    --output_dir $OUTPUT_PATH \
    --overwrite_cache \
    --overwrite_output_dir \
    --warmup_steps 100 \
    --weight_decay 0.1 \
    --per_device_train_batch_size 4 \
    --gradient_accumulation_steps 4 \
    --ddp_timeout 9000 \
    --learning_rate 5e-6 \
    --lr_scheduler_type cosine \
    --logging_steps 1 \
    --cutoff_len 4096 \
    --save_steps 1000 \
    --plot_loss \
    --num_train_epochs 3 \
    --bf16
```

并享受训练过程。若要调整您的训练，您可以通过修改训练命令中的参数来调整超参数。其中一个需要注意的参数是 cutoff_len，它代表训练数据的最大长度。通过控制这个参数，可以避免出现 OOM（内存溢出）错误。

合并 LoRA

如果你使用 LoRA 训练模型，可能需要将 adapter 参数合并到主分支中。请运行以下命令以执行 LoRA adapter 的合并操作。

```
CUDA_VISIBLE_DEVICES=0 llamafactory-cli export \
    --model_name_or_path path_to_base_model \
    --adapter_name_or_path path_to_adapter \
    --template qwen \
    --finetuning_type lora \
    --export_dir path_to_export \
    --export_size 2 \
    --export_legacy_format False
```

结语

上述内容是使用 LLaMA-Factory 训练 Qwen 的最简单方法。欢迎通过查看官方仓库深入了解详细信息！

1.15 Function Calling

在 Qwen-Agent 中，我们提供了一个专用封装器，旨在实现通过 dashscope API 与 OpenAI API 进行的函数调用。

1.15.1 使用示例

```
import json
import os
from qwen_agent.llm import get_chat_model


# Example dummy function hard coded to return the same weather
# In production, this could be your backend API or an external API
def get_current_weather(location, unit='fahrenheit'):
    """Get the current weather in a given location"""
    if 'tokyo' in location.lower():
        return json.dumps({
            'location': 'Tokyo',
            'temperature': '10',
            'unit': 'celsius'
        })
    elif 'san francisco' in location.lower():
        return json.dumps({
            'location': 'San Francisco',
            'temperature': '72',
            'unit': 'fahrenheit'
        })
    elif 'paris' in location.lower():
```

(续下页)

(接上页)

```
        return json.dumps({
            'location': 'Paris',
            'temperature': '22',
            'unit': 'celsius'
        })
    else:
        return json.dumps({'location': location, 'temperature': 'unknown'})


def test():
    llm = get_chat_model({
        # Use the model service provided by DashScope:
        'model': 'qwen-max',
        'model_server': 'dashscope',
        'api_key': os.getenv('DASHSCOPE_API_KEY'),

        # Use the model service provided by Together.AI:
        # 'model': 'Qwen/Qwen2-7B-Instruct',
        # 'model_server': 'https://api.together.xyz',  # api_base
        # 'api_key': os.getenv('TOGETHER_API_KEY'),

        # Use your own model service compatible with OpenAI API:
        # 'model': 'Qwen/Qwen2-72B-Instruct',
        # 'model_server': 'http://localhost:8000/v1',  # api_base
        # 'api_key': 'EMPTY',
    })

    # Step 1: send the conversation and available functions to the model
    messages = [{
        'role': 'user',
        'content': "What's the weather like in San Francisco?"
    }]
    functions = [{
        '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'],
        },
    }]

    print('# Assistant Response 1:')
    responses = []
    for responses in llm.chat(messages=messages,
                              functions=functions,
                              stream=True):
```

(续下页)

(接上页)

```
        print(responses)

    messages.extend(responses)  # extend conversation with assistant's reply

    # Step 2: check if the model wanted to call a function
    last_response = messages[-1]
    if last_response.get('function_call', None):

        # Step 3: call the function
        # Note: the JSON response may not always be valid; be sure to handle errors
        available_functions = {
            'get_current_weather': get_current_weather,
        }  # only one function in this example, but you can have multiple
        function_name = last_response['function_call']['name']
        function_to_call = available_functions[function_name]
        function_args = json.loads(last_response['function_call']['arguments'])
        function_response = function_to_call(
            location=function_args.get('location'),
            unit=function_args.get('unit'),
        )
        print('# Function Response:')
        print(function_response)

        # Step 4: send the info for each function call and function response to the␣
↪model
        messages.append({
            'role': 'function',
            'name': function_name,
            'content': function_response,
        })  # extend conversation with function response

        print('# Assistant Response 2:')
        for responses in llm.chat(
                messages=messages,
                functions=functions,
                stream=True,
        ):  # get a new response from the model where it can see the function response
            print(responses)


if __name__ == '__main__':
    test()
```

1.16 Qwen-Agent

Qwen-Agent 是一个基于 Qwen 的指令跟随、工具使用、计划和记忆能力来开发 LLM 应用程序的框架。它还附带了一些示例应用程序，例如浏览器助手、代码解释器和自定义助手。

1.16.1 安装

```
git clone https://github.com/QwenLM/Qwen-Agent.git
cd Qwen-Agent
pip install -e ./
```

1.16.2 开发您自己的智能体

Qwen-Agent 提供包括语言模型和提示词等原子级组件，及智能体等高级组件在内的多种组件。以下示例选取助理组件进行展示，阐述了如何整合自定义工具以及如何迅速开发出一个能够应用这些工具的代理程序。

```
import json
import os

import json5
import urllib.parse
from qwen_agent.agents import Assistant
from qwen_agent.tools.base import BaseTool, register_tool

llm_cfg = {
    # Use the model service provided by DashScope:
    'model': 'qwen-max',
    'model_server': 'dashscope',
    # 'api_key': 'YOUR_DASHSCOPE_API_KEY',
    # It will use the `DASHSCOPE_API_KEY' environment variable if 'api_key' is not␣
↪set here.

    # Use your own model service compatible with OpenAI API:
    # 'model': 'Qwen/Qwen2-72B-Instruct',
    # 'model_server': 'http://localhost:8000/v1',  # api_base
    # 'api_key': 'EMPTY',

    # (Optional) LLM hyperparameters for generation:
    'generate_cfg': {
        'top_p': 0.8
    }
}
system = 'According to the user\'s request, you first draw a picture and then␣
↪automatically run code to download the picture ' + \
         'and select an image operation from the given document to process the image'

# Add a custom tool named my_image_gen:
@register_tool('my_image_gen')
class MyImageGen(BaseTool):
    description = 'AI painting (image generation) service, input text description,␣
↪and return the image URL drawn based on text information.'
    parameters = [{
        'name': 'prompt',
        'type': 'string',
        'description': 'Detailed description of the desired image content, in English
↪',
        'required': True
    }]

    def call(self, params: str, **kwargs) -> str:
```

(续下页)

(接上页)

```
        prompt = json5.loads(params)['prompt']
        prompt = urllib.parse.quote(prompt)
        return json.dumps(
            {'image_url': f'https://image.pollinations.ai/prompt/{prompt}'},
            ensure_ascii=False)


tools = ['my_image_gen', 'code_interpreter']  # code_interpreter is a built-in tool␣
↪in Qwen-Agent
bot = Assistant(llm=llm_cfg,
                system_message=system,
                function_list=tools,
                files=[os.path.abspath('doc.pdf')])

messages = []
while True:
    query = input('user question: ')
    messages.append({'role': 'user', 'content': query})
    response = []
    for response in bot.run(messages=messages):
        print('bot response:', response)
    messages.extend(response)
```

该框架还为开发者提供了更多的原子组件以供组合使用。欲了解更多示例，请参见 examples。

1.17 LlamaIndex

为了实现 Qwen2 与外部数据（例如文档、网页等）的连接，我们提供了 LlamaIndex 的详细教程。本指南旨在帮助用户利用 LlamaIndex 与 Qwen2 快速部署检索增强生成（RAG）技术。

1.17.1 环境准备

为实现检索增强生成（RAG），我们建议您首先安装与 LlamaIndex 相关的软件包。

以下是一个简单的代码示例：

```
pip install llama-index
pip install llama-index-llms-huggingface
pip install llama-index-readers-web
```

1.17.2 设置参数

现在，我们可以设置语言模型和向量模型。Qwen2-Instruct 支持包括英语和中文在内的多种语言对话。您可以使用 “bge-base-en-v1.5“模型来检索英文文档，下载 “bge-base-zh-v1.5“模型以检索中文文档。根据您的计算资源，您还可以选择 “bge-large“或 “bge-small“作为向量模型，或调整上下文窗口大小或文本块大小。Qwen2 模型系列支持最大 32K 上下文窗口大小（Qwen2-7B-Instruct 与 Qwen2-72B-Instruct 可支持 128K 上下文，但需要额外配置）

```
import torch
from llama_index.core import Settings
from llama_index.core.node_parser import SentenceSplitter
```

(续下页)

(接上页)

```
from llama_index.llms.huggingface import HuggingFaceLLM
from llama_index.embeddings.huggingface import HuggingFaceEmbedding

# Set prompt template for generation (optional)
from llama_index.core import PromptTemplate

def completion_to_prompt(completion):
   return f"<|im_start|>system\n<|im_end|>\n<|im_start|>user\n{completion}<|im_end|>\n
↪<|im_start|>assistant\n"

def messages_to_prompt(messages):
    prompt = ""
    for message in messages:
        if message.role == "system":
            prompt += f"<|im_start|>system\n{message.content}<|im_end|>\n"
        elif message.role == "user":
            prompt += f"<|im_start|>user\n{message.content}<|im_end|>\n"
        elif message.role == "assistant":
            prompt += f"<|im_start|>assistant\n{message.content}<|im_end|>\n"

    if not prompt.startswith("<|im_start|>system"):
        prompt = "<|im_start|>system\n" + prompt

    prompt = prompt + "<|im_start|>assistant\n"

    return prompt

# Set Qwen2 as the language model and set generation config
Settings.llm = HuggingFaceLLM(
    model_name="Qwen/Qwen2-7B-Instruct",
    tokenizer_name="Qwen/Qwen2-7B-Instruct",
    context_window=30000,
    max_new_tokens=2000,
    generate_kwargs={"temperature": 0.7, "top_k": 50, "top_p": 0.95},
    messages_to_prompt=messages_to_prompt,
    completion_to_prompt=completion_to_prompt,
    device_map="auto",
)

# Set embedding model
Settings.embed_model = HuggingFaceEmbedding(
    model_name = "BAAI/bge-base-en-v1.5"
)

# Set the size of the text chunk for retrieval
Settings.transformations = [SentenceSplitter(chunk_size=1024)]
```

1.17.3 构建索引

现在我们可以从文档或网站构建索引。

以下代码片段展示了如何为本地名为’document’的文件夹中的文件（无论是 PDF 格式还是 TXT 格式）构建索引。

```
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader

documents = SimpleDirectoryReader("./document").load_data()
index = VectorStoreIndex.from_documents(
    documents,
    embed_model=Settings.embed_model,
    transformations=Settings.transformations
)
```

以下代码片段展示了如何为一系列网站的内容构建索引。

```
from llama_index.readers.web import SimpleWebPageReader
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader

documents = SimpleWebPageReader(html_to_text=True).load_data(
    ["web_address_1","web_address_2",...]
)
index = VectorStoreIndex.from_documents(
    documents,
    embed_model=Settings.embed_model,
    transformations=Settings.transformations
)
```

要保存和加载已构建的索引，您可以使用以下代码示例。

```
from llama_index.core import StorageContext, load_index_from_storage

# save index
storage_context = StorageContext.from_defaults(persist_dir="save")

# load index
index = load_index_from_storage(storage_context)
```

1.17.4 检索增强（RAG）

现在您可以输入查询，Qwen2 将基于索引文档的内容提供答案。

```
query_engine = index.as_query_engine()
your_query = "<your query here>"
print(query_engine.query(your_query).response)
```

1.18 Langchain

本教程旨在帮助您利用 “Qwen2-7B-Instruct“与 “langchain“，基于本地知识库构建问答应用。目标是建立一个知识库问答解决方案。

1.18.1 基础用法

您可以仅使用您的文档配合 “langchain“来构建一个问答应用。该项目的实现流程包括加载文件 -> 阅读文本 -> 文本分段 -> 文本向量化 -> 问题向量化 -> 将最相似的前 k 个文本向量与问题向量匹配 -> 将匹配的文本作为上下文连同问题一起纳入提示 -> 提交给 Qwen2-7B-Instruct 生成答案。以下是一个示例：

```
pip install langchain==0.0.174
pip install faiss-gpu
```

```
from transformers import AutoModelForCausalLM, AutoTokenizer
from abc import ABC
from langchain.llms.base import LLM
from typing import Any, List, Mapping, Optional
from langchain.callbacks.manager import CallbackManagerForLLMRun
device = "cuda" # the device to load the model onto

model = AutoModelForCausalLM.from_pretrained(
    "Qwen/Qwen2-7B-Instruct",
    torch_dtype="auto",
    device_map="auto"
)
tokenizer = AutoTokenizer.from_pretrained("Qwen/Qwen2-7B-Instruct")

class Qwen(LLM, ABC):
     max_token: int = 10000
     temperature: float = 0.01
     top_p = 0.9
     history_len: int = 3

     def __init__(self):
         super().__init__()

     @property
     def _llm_type(self) -> str:
         return "Qwen"

     @property
     def _history_len(self) -> int:
         return self.history_len
```

(续下页)

(接上页)

```
    def set_history_len(self, history_len: int = 10) -> None:
        self.history_len = history_len

    def _call(
        self,
        prompt: str,
        stop: Optional[List[str]] = None,
        run_manager: Optional[CallbackManagerForLLMRun] = None,
    ) -> str:
        messages = [
            {"role": "system", "content": "You are a helpful assistant."},
            {"role": "user", "content": prompt}
        ]
        text = tokenizer.apply_chat_template(
            messages,
            tokenize=False,
            add_generation_prompt=True
        )
        model_inputs = tokenizer([text], return_tensors="pt").to(device)
        generated_ids = model.generate(
            model_inputs.input_ids,
            max_new_tokens=512
        )
        generated_ids = [
            output_ids[len(input_ids):] for input_ids, output_ids in zip(model_
↪inputs.input_ids, generated_ids)
        ]

        response = tokenizer.batch_decode(generated_ids, skip_special_tokens=True)[0]
        return response

    @property
    def _identifying_params(self) -> Mapping[str, Any]:
        """Get the identifying parameters."""
        return {"max_token": self.max_token,
                "temperature": self.temperature,
                "top_p": self.top_p,
                "history_len": self.history_len}
```

加载 Qwen2-7B-Instruct 模型后，您可以指定需要用于知识库问答的 txt 文件。

```
import os
import re
import torch
import argparse
from langchain.vectorstores import FAISS
from langchain.embeddings.huggingface import HuggingFaceEmbeddings
from typing import List, Tuple
import numpy as np
from langchain.document_loaders import TextLoader
from langchain.text_splitter import CharacterTextSplitter
from langchain.docstore.document import Document
from langchain.prompts.prompt import PromptTemplate
from langchain.chains import RetrievalQA

class ChineseTextSplitter(CharacterTextSplitter):
    def __init__(self, pdf: bool = False, **kwargs):
```

(续下页)

(接上页)

```
        super().__init__(**kwargs)
        self.pdf = pdf

    def split_text(self, text: str) -> List[str]:
        if self.pdf:
            text = re.sub(r"\n{3,}", "\n", text)
            text = re.sub('\s', ' ', text)
            text = text.replace("\n\n", "")
        sent_sep_pattern = re.compile(
            '([🄵🄵🄵🄵. 。！ ？][”’ ” 」 』 ]{0,2}|(?=[“ ‘ “ 「 『]{1,2}|$))')
        sent_list = []
        for ele in sent_sep_pattern.split(text):
            if sent_sep_pattern.match(ele) and sent_list:
                sent_list[-1] += ele
            elif ele:
                sent_list.append(ele)
        return sent_list


def load_file(filepath):
    loader = TextLoader(filepath, autodetect_encoding=True)
    textsplitter = ChineseTextSplitter(pdf=False)
    docs = loader.load_and_split(textsplitter)
    write_check_file(filepath, docs)
    return docs


def write_check_file(filepath, docs):
    folder_path = os.path.join(os.path.dirname(filepath), "tmp_files")
    if not os.path.exists(folder_path):
        os.makedirs(folder_path)
    fp = os.path.join(folder_path, 'load_file.txt')
    with open(fp, 'a+', encoding='utf-8') as fout:
        fout.write("filepath=%s,len=%s" % (filepath, len(docs)))
        fout.write('\n')
        for i in docs:
            fout.write(str(i))
            fout.write('\n')
        fout.close()


def separate_list(ls: List[int]) -> List[List[int]]:
    lists = []
    ls1 = [ls[0]]
    for i in range(1, len(ls)):
        if ls[i - 1] + 1 == ls[i]:
            ls1.append(ls[i])
        else:
            lists.append(ls1)
            ls1 = [ls[i]]
    lists.append(ls1)
    return lists


class FAISSWrapper(FAISS):
    chunk_size = 250
    chunk_conent = True
```

(续下页)

(接上页)

```
    score_threshold = 0

    def similarity_search_with_score_by_vector(
            self, embedding: List[float], k: int = 4
    ) -> List[Tuple[Document, float]]:
        scores, indices = self.index.search(np.array([embedding], dtype=np.float32),␣
↪k)
        docs = []
        id_set = set()
        store_len = len(self.index_to_docstore_id)
        for j, i in enumerate(indices[0]):
            if i == -1 or 0 < self.score_threshold < scores[0][j]:
                # This happens when not enough docs are returned.
                continue
            _id = self.index_to_docstore_id[i]
            doc = self.docstore.search(_id)
            if not self.chunk_conent:
                if not isinstance(doc, Document):
                    raise ValueError(f"Could not find document for id {_id}, got {doc}
↪")
                doc.metadata["score"] = int(scores[0][j])
                docs.append(doc)
                continue
            id_set.add(i)
            docs_len = len(doc.page_content)
            for k in range(1, max(i, store_len - i)):
                break_flag = False
                for l in [i + k, i - k]:
                    if 0 <= l < len(self.index_to_docstore_id):
                        _id0 = self.index_to_docstore_id[l]
                        doc0 = self.docstore.search(_id0)
                        if docs_len + len(doc0.page_content) > self.chunk_size:
                            break_flag = True
                            break
                        elif doc0.metadata["source"] == doc.metadata["source"]:
                            docs_len += len(doc0.page_content)
                            id_set.add(l)
                if break_flag:
                    break
        if not self.chunk_conent:
            return docs
        if len(id_set) == 0 and self.score_threshold > 0:
            return []
        id_list = sorted(list(id_set))
        id_lists = separate_list(id_list)
        for id_seq in id_lists:
            for id in id_seq:
                if id == id_seq[0]:
                    _id = self.index_to_docstore_id[id]
                    doc = self.docstore.search(_id)
                else:
                    _id0 = self.index_to_docstore_id[id]
                    doc0 = self.docstore.search(_id0)
                    doc.page_content += " " + doc0.page_content
            if not isinstance(doc, Document):
                raise ValueError(f"Could not find document for id {_id}, got {doc}")
            doc_score = min([scores[0][id] for id in [indices[0].tolist().index(i)␣
```

(续下页)

(接上页)

```
↪for i in id_seq if i in indices[0]]])
            doc.metadata["score"] = int(doc_score)
            docs.append((doc, doc_score))
        return docs


if __name__ == '__main__':
    # load docs (pdf file or txt file)
    filepath = 'your file path'
    # Embedding model name
    EMBEDDING_MODEL = 'text2vec'
    PROMPT_TEMPLATE = """Known information:
    {context_str}
    Based on the above known information, respond to the user's question concisely␣
↪and professionally. If an answer cannot be derived from it, say 'The question␣
↪cannot be answered with the given information' or 'Not enough relevant information␣
↪has been provided,' and do not include fabricated details in the answer. Please␣
↪respond in English. The question is {question}"""
    # Embedding running device
    EMBEDDING_DEVICE = "cuda"
    # return top-k text chunk from vector store
    VECTOR_SEARCH_TOP_K = 3
    CHAIN_TYPE = 'stuff'
    embedding_model_dict = {
        "text2vec": "your text2vec model path",
    }
    llm = Qwen()
    embeddings = HuggingFaceEmbeddings(model_name=embedding_model_dict[EMBEDDING_
↪MODEL],model_kwargs={'device': EMBEDDING_DEVICE})

    docs = load_file(filepath)

    docsearch = FAISSWrapper.from_documents(docs, embeddings)

    prompt = PromptTemplate(
        template=PROMPT_TEMPLATE, input_variables=["context_str", "question"]
    )

    chain_type_kwargs = {"prompt": prompt, "document_variable_name": "context_str"}
    qa = RetrievalQA.from_chain_type(
        llm=llm,
        chain_type=CHAIN_TYPE,
        retriever=docsearch.as_retriever(search_kwargs={"k": VECTOR_SEARCH_TOP_K}),
        chain_type_kwargs=chain_type_kwargs)

    query = "Give me a short introduction to large language model."
    print(qa.run(query))
```

1.18.2 下一步

现在，您可以在您自己的文档上与 Qwen2 进行交流。继续阅读文档，尝试探索模型检索的更多高级用法！

1.19 量化模型效果评估

本部分介绍 Qwen2 量化模型（包括 GPTQ 与 AWQ 量化方案）的效果评估，有以下数据集

- MMLU （准确率）
- C-Eval （准确率）
- IFEval （提示词级的严格准确率，Strict Prompt-Level Accuracy）

所有模型均使用贪心解码。

	量化模型	平均	MMLU	C-Eval	IFEval
Qwen2-72B-Instruct	BF16	81.3	82.3	83.8	77.6
	GPTQ-Int8	80.7	81.3	83.4	77.5
	GPTQ-Int4	81.2	80.8	83.9	78.9
	AWQ	80.4	80.5	83.9	76.9
Qwen2-7B-Instruct	BF16	66.9	70.5	77.2	53.1
	GPTQ-Int8	66.2	69.1	76.7	52.9
	GPTQ-Int4	64.1	67.8	75.2	49.4
	AWQ	64.1	67.4	73.6	51.4
Qwen2-1.5B-Instruct	BF16	48.4	52.4	63.8	29.0
	GPTQ-Int8	48.1	53.0	62.5	28.8
	GPTQ-Int4	45.0	50.7	57.4	27.0
	AWQ	46.5	51.6	58.1	29.9
Qwen2-0.5B-Instruct	BF16	34.4	37.9	45.2	20.0
	GPTQ-Int8	32.6	35.6	43.9	18.1
	GPTQ-Int4	29.7	33.0	39.2	16.8
	AWQ	31.1	34.4	42.1	16.7

1.20 效率评估

本部分介绍 Qwen2 模型（原始模型和量化模型）的效率测试结果，包括推理速度 (tokens/s) 与不同上下文长度时的显存占用 (GB)。

测试 HuggingFace `transformers` 时的环境配置：

- NVIDIA A100 80GB
- CUDA 11.8
- Pytorch 2.1.2+cu118
- Flash Attention 2.3.3
- Transformers 4.38.2
- AutoGPTQ 0.7.1
- AutoAWQ 0.2.4

测试 vLLM 时的环境配置：

- NVIDIA A100 80GB
- CUDA 11.8
- Pytorch 2.3.0+cu118
- Flash Attention 2.5.6
- Transformers 4.40.1
- vLLM 0.4.2

注意：

- batch size 设置为 1，使用 GPU 数量尽可能少
- 我们测试生成 2048 tokens 时的速度与显存占用，输入长度分别为 1、6144、14336、30720、63488、129024 tokens。(超过 32K 长度仅有 Qwen2-72B-Instuct 与 Qwen2-7B-Instuct 支持)
- 对于 vLLM，由于 GPU 显存预分配，实际显存使用难以评估。默认情况下，统一设定为 “gpu_memory_utilization=0.9 max_model_len=32768 enforce_eager=False“。
- 0.5B (Transformer)

模型	输入长度	量化	GPU 数量	速度 (tokens/s)	显存占用 (GB)
Qwen2-0.5B-Instruct	1	BF16	1	49.94	1.17
		GPTQ-Int8	1	36.35	0.85
		GPTQ-Int4	1	49.56	0.68
		AWQ	1	38.78	0.68
	6144	BF16	1	50.83	6.42
		GPTQ-Int8	1	36.56	6.09
		GPTQ-Int4	1	49.63	5.93
		AWQ	1	38.73	5.92
	14336	BF16	1	49.56	13.48
		GPTQ-Int8	1	36.23	13.15
		GPTQ-Int4	1	48.68	12.97
		AWQ	1	38.94	12.99
	30720	BF16	1	49.25	27.61
		GPTQ-Int8	1	34.61	27.28
		GPTQ-Int4	1	48.18	27.12
		AWQ	1	38.19	27.11

- 0.5B (vLLM)

模型	输入长度	量化	GPU 数量	速度 (tokens/s)
Qwen2-0.5B-Instruct	1	BF16	1	270.49
		GPTQ-Int8	1	235.95
		GPTQ-Int4	1	240.07
		AWQ	1	233.31
	6144	BF16	1	256.16
		GPTQ-Int8	1	224.30
		GPTQ-Int4	1	226.41
		AWQ	1	222.83
	14336	BF16	1	108.89
		GPTQ-Int8	1	108.10
		GPTQ-Int4	1	106.51
		AWQ	1	104.16
	30720	BF16	1	97.20
		GPTQ-Int8	1	94.49
		GPTQ-Int4	1	93.94
		AWQ	1	92.23

- 1.5B (Transformer)

模型	输入长度	量化	GPU 数量	速度 (tokens/s)	显存占用 (GB)
Qwen2-1.5B-Instruct	1	BF16	1	40.89	3.44
		GPTQ-Int8	1	31.51	2.31
		GPTQ-Int4	1	42.47	1.67
		AWQ	1	33.62	1.64
	6144	BF16	1	40.86	8.74
		GPTQ-Int8	1	31.31	7.59
		GPTQ-Int4	1	42.78	6.95
		AWQ	1	32.90	6.92
	14336	BF16	1	40.08	15.92
		GPTQ-Int8	1	31.19	14.79
		GPTQ-Int4	1	42.25	14.14
		AWQ	1	33.24	14.12
	30720	BF16	1	34.09	30.31
		GPTQ-Int8	1	28.52	29.18
		GPTQ-Int4	1	31.30	28.54
		AWQ	1	32.16	28.51

- 1.5B (vLLM)

模型	输入长度	量化	GPU 数量	速度 (tokens/s)
Qwen2-1.5B-Instruct	1	BF16	1	175.55
		GPTQ-Int8	1	172.28
		GPTQ-Int4	1	184.58
		AWQ	1	170.87
	6144	BF16	1	166.23
		GPTQ-Int8	1	164.32
		GPTQ-Int4	1	174.04
		AWQ	1	162.81
	14336	BF16	1	83.67
		GPTQ-Int8	1	98.63
		GPTQ-Int4	1	97.65
		AWQ	1	92.48
	30720	BF16	1	77.69
		GPTQ-Int8	1	86.42
		GPTQ-Int4	1	87.49
		AWQ	1	82.88

- 7B (Transformer)

模型	输入长度	量化	GPU 数量	速度 (tokens/s)	显存占用 (GB)
Qwen2-7B-Instruct	1	BF16	1	37.97	14.92
		GPTQ-Int8	1	30.85	8.97
		GPTQ-Int4	1	36.17	6.06
		AWQ	1	33.08	5.93
	6144	BF16	1	34.74	20.26
		GPTQ-Int8	1	31.13	14.31
		GPTQ-Int4	1	33.34	11.40
		AWQ	1	30.86	11.27
	14336	BF16	1	26.63	27.71
		GPTQ-Int8	1	24.58	21.76
		GPTQ-Int4	1	25.81	18.86
		AWQ	1	27.61	18.72
	30720	BF16	1	17.49	42.62
		GPTQ-Int8	1	16.69	36.67
		GPTQ-Int4	1	17.17	33.76
		AWQ	1	17.87	33.63

- 7B (vLLM)

模型	输入长度	量化	GPU 数量	速度 (tokens/s)
Qwen2-7B-Instruct	1	BF16	1	80.45
		GPTQ-Int8	1	114.32
		GPTQ-Int4	1	143.40
		AWQ	1	96.65
	6144	BF16	1	76.41
		GPTQ-Int8	1	107.02
		GPTQ-Int4	1	131.55
		AWQ	1	91.38
	14336	BF16	1	66.54
		GPTQ-Int8	1	89.72
		GPTQ-Int4	1	97.93
		AWQ	1	76.87
	30720	BF16	1	55.83
		GPTQ-Int8	1	71.58
		GPTQ-Int4	1	81.48
		AWQ	1	63.62
	63488	BF16	1	41.20
		GPTQ-Int8	1	49.37
		GPTQ-Int4	1	54.12
		AWQ	1	45.89
	129024	BF16	1	25.01
		GPTQ-Int8	1	27.73
		GPTQ-Int4	1	29.39
		AWQ	1	27.13

- 57B-A14B (Transformer)

模型	输入长度	量化	GPU 数量	速度 (tokens/s)	显存占用 (GB)
Qwen2-57B-A14B-Instruct	1	BF16	2	4.76	110.29
		GPTQ-Int4	1	5.55	30.38
	6144	BF16	2	4.90	117.80
		GPTQ-Int4	1	5.44	35.67
	14336	BF16	2	4.58	128.17
		GPTQ-Int4	1	5.31	43.11
	30720	BF16	2	4.12	163.77
		GPTQ-Int4	1	4.72	58.01

- 57B-A14B (vLLM)

模型	输入长度	量化	GPU 数量	速度 (tokens/s)
Qwen2-57B-A14B-Instruct	1	BF16	2	31.44
	6144	BF16	2	31.77
	14336	BF16	2	21.25
	30720	BF16	2	20.24

混合专家模型 (Mixture-of-Experts, MoE) 与稠密模型相比，当批大小较大时，吞吐量更大。下表展示了有关数据：

模型	量化	请求数	请求每秒 (QPS)	速度 (tokens/s)
Qwen1.5-32B-Chat	BF16	100	6.68	7343.56
Qwen2-57B-A14B-Instruct	BF16	100	4.81	5291.15
Qwen1.5-32B-Chat	BF16	1000	7.99	8791.35
Qwen2-57B-A14B-Instruct	BF16	1000	5.18	5698.37

数据由 vLLM 吞吐量测试脚本测得，可通过以下命令复现

```
python vllm/benchmarks/benchmark_throughput.py --input-len 1000 --output-len
100 --model <model_path> --num-prompts <number of prompts> --enforce-eager -tp
2
```

- 72B (Transformer)

模型	输入长度	量化	GPU 数量	速度 (tokens/s)	显存占用 (GB)
Qwen2-72B-Instruct	1	BF16	2	7.45	134.74
		GPTQ-Int8	2	7.30	71.00
		GPTQ-Int4	1	9.05	41.80
		AWQ	1	9.96	41.31
	6144	BF16	2	5.99	144.38
		GPTQ-Int8	2	5.93	80.60
		GPTQ-Int4	1	6.79	47.90
		AWQ	1	7.49	47.42
	14336	BF16	3	4.12	169.93
		GPTQ-Int8	2	4.43	95.14
		GPTQ-Int4	1	4.87	57.79
		AWQ	1	5.23	57.30
	30720	BF16	3	2.86	209.03
		GPTQ-Int8	2	2.83	124.20
		GPTQ-Int4	2	3.02	107.94
		AWQ	2	1.85	88.60

- 72B (vLLM)

模型	输入长度	量化	GPU 数量	速度 (tokens/s)	Setting
Qwen2-72B-Instruct	1	BF16	2	17.68	[Setting 1]
		BF16	4	30.01	•
		GPTQ-Int8	2	27.56	•
		GPTQ-Int4	1	29.60	[设定 2]
		GPTQ-Int4	2	42.82	•
		AWQ	2	27.73	•

续下页

表 1 – 接上页

模型	输入长度	量化	GPU 数量	速度 (tokens/s)	Setting
	6144	BF16	4	27.98	•
		GPTQ-Int8	2	25.46	•
		GPTQ-Int4	1	25.16	[设定 3]
		GPTQ-Int4	2	38.23	•
		AWQ	2	25.77	•
	14336	BF16	4	21.81	•
		GPTQ-Int8	2	22.71	•
		GPTQ-Int4	2	26.54	•
		AWQ	2	21.50	•
	30720	BF16	4	19.43	•
		GPTQ-Int8	2	18.69	•
		GPTQ-Int4	2	23.12	•
		AWQ	2	18.09	•
	30720	BF16	4	19.43	•
		GPTQ-Int8	2	18.69	•
		GPTQ-Int4	2	23.12	•
		AWQ	2	18.09	•

续下页

表 1 – 接上页

模型	输入长度	量化	GPU 数量	速度 (tokens/s)	Setting
	63488	BF16	4	17.46	•
		GPTQ-Int8	2	15.30	•
		GPTQ-Int4	2	13.23	•
		AWQ	2	13.14	•
	129024	BF16	4	11.70	•
		GPTQ-Int8	4	12.94	•
		GPTQ-Int4	2	8.33	•
		AWQ	2	7.78	•

- [默认设定]=(gpu_memory_utilization=0.9 max_model_len=32768 enforce_eager=False)
- [设定 1]=(gpu_memory_utilization=0.98 max_model_len=4096 enforce_eager=True)
- [设定 2]=(gpu_memory_utilization=1.0 max_model_len=4096 enforce_eager=True)
- [设定 3]=(gpu_memory_utilization=1.0 max_model_len=8192 enforce_eager=True)