

# 模型部署优化

## Accelerate

Accelerate是一个使用分布式开发技术，实现基于PyTorch平台的大模型训练和推理库。

Accelerate 是基于 `torch.xla` 和 `torch.distributed` 技术构建的。由于Accelerate 完成繁重的底层处理工作，所以我们反而不需要编写任何自定义代码来适应这些底层技术平台。Accelerate 利用 DeepSpeed 转换现有代码库，执行完全分片的数据并行，并自动支持混合精度训练！

## 安装

安装最新版的 Accelerate 需要 **Python 3.8** 以上的版本。

```
pip install accelerate
```

或

```
conda install -c conda-forge accelerate
```

## 大模型推理

Accelerate 对 **大模型推理** 提供了非常好的支持，它允许我们把无法完全装入显卡的模型进行加载并推理。

加载 PyTorch 模型的典型工作流程如下所示。 假设 `ModelClass` 是一个超过设备（mps、cuda 或 xpu）GPU 内存的模型。

```
import torch

my_model = ModelClass(...)
state_dict = torch.load(checkpoint_file)
my_model.load_state_dict(state_dict)
```

## 添加 Accelerate支持

在进行大模型推理时，首先使用 `init_empty_weights` 上下文管理器初始化一个模型的空框架。

这种创建方式不占用任何内存，因为 “没有参数”。

```
from accelerate import init_empty_weights

with init_empty_weights():
    my_model = ModelClass(...)
```

接着将 模型权重（参数） 加载到模型中进行推理。

```
from accelerate import load_checkpoint_and_dispatch

model = load_checkpoint_and_dispatch(
    model, checkpoint=checkpoint_file, device_map="auto"
)
```

`load_checkpoint_and_dispatch()` 方法会在空模型中加载检查点，并将各层的权重依次分配到所有可用设备上。

**分配的优先级：**速度最快的设备（GPU、MPS、XPU、NPU、MLU、SDAA、MUSA），然后再分配到速度较慢的设备（CPU 和硬盘）。

设置 `device_map="auto"` 会先自动填充 GPU 上所有可用空间，然后是 CPU，最后，如果内存仍然不足，则使用硬盘（最慢的选择）。

模型完全部署后，就可以进行推理了。

```
input = torch.randn(2,3)
device_type = next(iter(model.parameters())).device.type
```

```
input = input.to(device_type)
output = model(input)
```

每次输入通过一层时，它都会从CPU发送到GPU（或从磁盘发送到CPU再到GPU），计算输出，然后该层从GPU移除，沿线路返回。虽然这会给推理增加一些开销，但只要最大的层能在我们的GPU上运行，它就能让系统运行任意规模的模型。

还可以使用多个GPU，即“模型并行”(model parallelism)，但给定时刻只有一个GPU处于活动状态。这使得当前GPU等待前一个GPU向其发送输出。我们应该使用Python正常启动脚本，而不是使用 **torchrun** 和 **accelerate launch** 等其他工具。

## 小结

归纳一下自定义的模型训练，使用 Accelerate 进行设置的步骤：

1. 在脚本中通过 `init_empty_weights` 初始化 大模型类的对象（我们将要调用的对象）。
2. 使用 `load_checkpoint_and_dispatch` 方法加载模型权重，填充到之前创建的空模型对象。
3. 正常调用模型的推理或生成方法（accelerator 会自动帮我们完成模型的推理和数据传递）。



在使用多个GPU时，设备是依次被调用执行的。

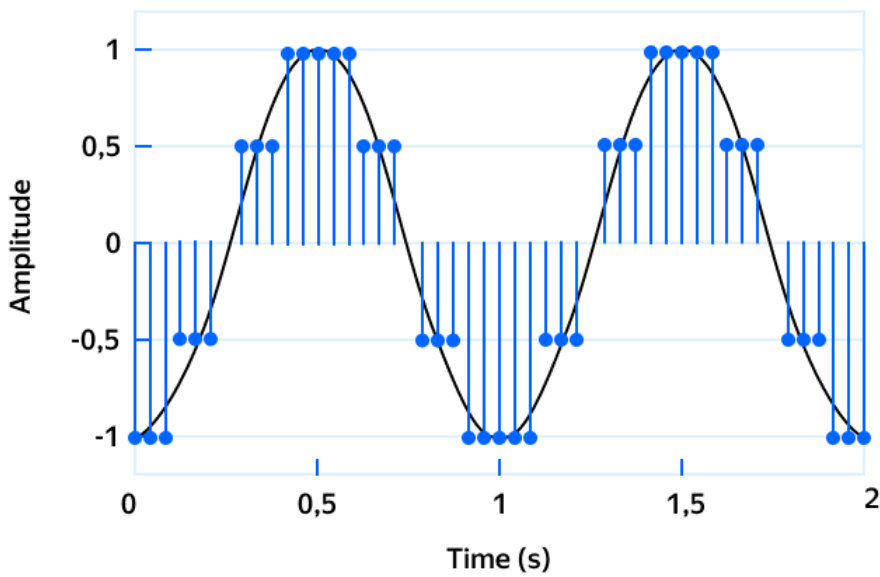
在大多数情况下，这就是整体的加速实现步骤就是这样。

# 模型量化

## 什么是量化以及它是如何工作的

量化是将大型信息表示的值（通常是连续集）转换为更紧凑的表示（通常是离散集）的过程。

量化的一个明显例子是模拟信号的采样，其中连续信号的每个值都被分配了来自预定离散集合的值。



模拟信号采样

在神经网络的上下文中环境中，量化意味着把占大量比特位的数据类型（例如 float32）转移到具有较小比特位的数据类型（例如 int8）的过程。下面讨论的内容，我们将更偏向于 LLM 来探讨量化神经网络模型的关键思想。

为什么需要量化？它可以节省计算资源（省钱）。由于量化模型需要更少的硬件计算资源并且速度更快。所以，通过量化技术，既能节省资金还能改善用户对于模型的应用体验。

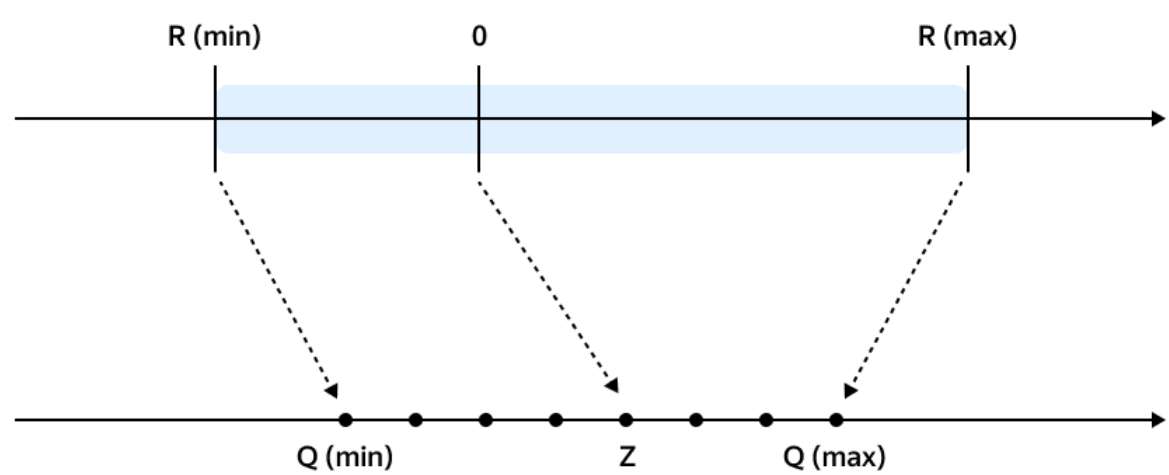
没有个人 GPU 集群的研究人员和爱好者现在有机会尝试大型现代模型。这就意味着我们可以直接在用户本地设备上高效地执行计算。

## 线性量化

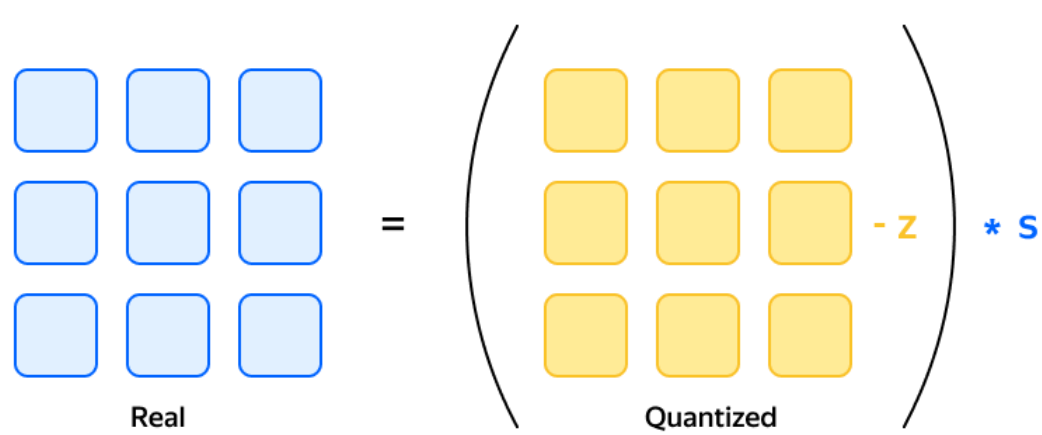
量化方法有很多。例如，通过聚类或矩阵分解等途径。我们这里将重点放到目前最流行且经过验证的线性量化方法作为主要内容。

- 仿射量化

仿射（或单端）量化将单端范围映射到 k 位数据类型。让我们考虑一个实际的取值范围 $[R_{min}, R_{max}]$ 。



当有了有一个量化张量后，还原成真实张量也非常简单。只需要将 `zero-point` 结果减去 z 并乘以 `scale`。从这里我们也很容易理解如何从真实张量转换成量化张量的计算过程。



$S$  和  $Z$  是量化常数，即过程中计算的参数。

$S$  即 `scale` 来负责变换的缩放比。

$$S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}}$$

由于精度缩放的要求，结果值还是使用原始的真实数据类型进行存储。

$Z$  即 `zero-point`，对应于零值，其中  $\lceil \cdot \rceil$  代表四舍五入。

$$Z = \lceil q_{min} - \frac{r_{min}}{S} \rceil$$

对于神经网络来说，零的准确表示非常重要。我们可以采用不同的方式舍入：向下舍入、随机舍入到最接近的整数。

$Z$  通常也是以量化类型存储的。

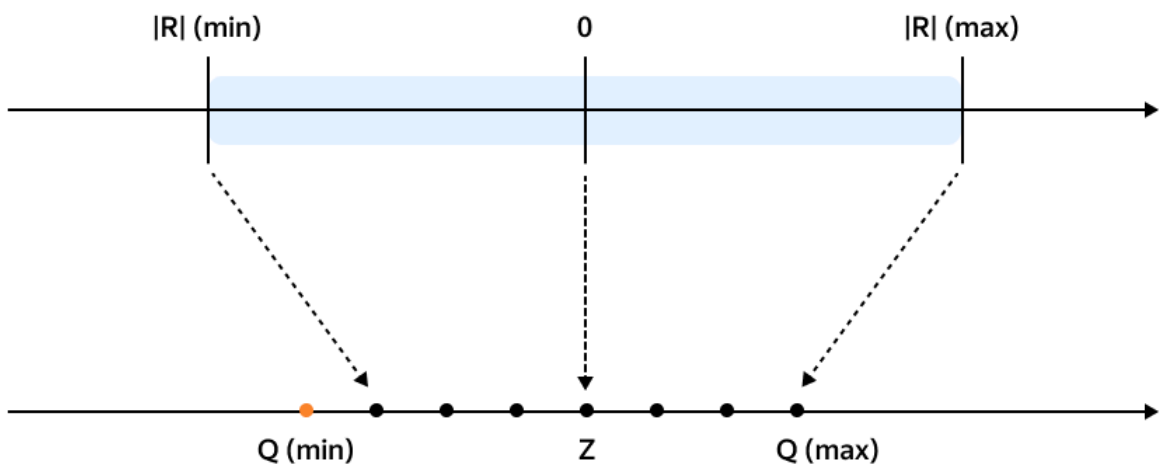
**量化：**  $X_q = \lceil \frac{X}{S} + Z \rceil$

**反量化：**  $X = S(X_q - Z)$

仿射量化非常适合非对称分布，例如 ReLU 的输出。

• **对称量化**

对称量化显示一个关于零对称的范围。



实数类型的零对应量化类型的零。量化范围的边界被确定为量化值的最大模数  $|R_{max}|$ 。

要使类型对称，就需要放弃量化数据类型中的一个值。例如，取值范围 `signed int8: [-128, 127]` 将变为 `[-127, 127]`。

量化常数  $S$ 、 $Z$ ：

$$S = \frac{|r|_{max}}{2^{N-1} - 1}$$
$$Z = 0$$

**量化：**  $X_q = \lceil \frac{X}{S} \rceil$

**反量化：**  $X = SX_q$

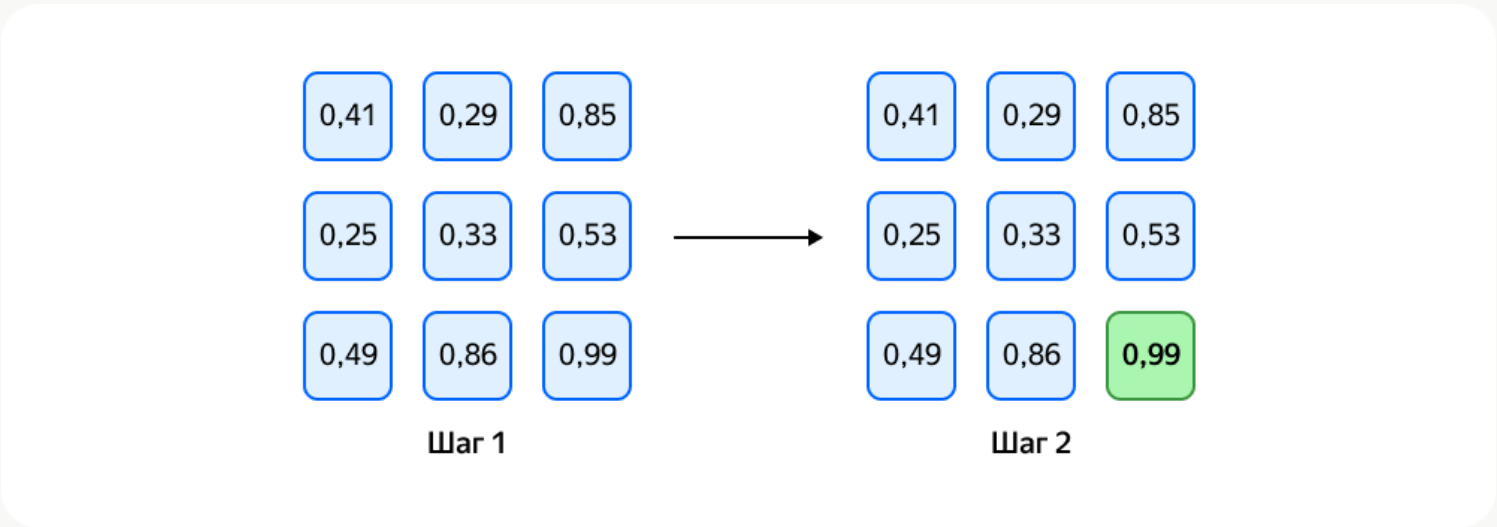
对称量化与仿射量化公式的区别在于缺少  $Z$ 。仿射量化的优点是可以更准确、更好地处理非对称分布，而对称量化则受益于简单性和速度。使用这种方法，无需考虑存储零点，对于反量化来说，将张量乘以一个常数就足够了。

。 **如何将实张量量化为 int8**

可以把神经网络看做数字张量的序列。让我们看一个说明性示例，了解如何将 32 位实数的真实张量量化为 8 位整数。

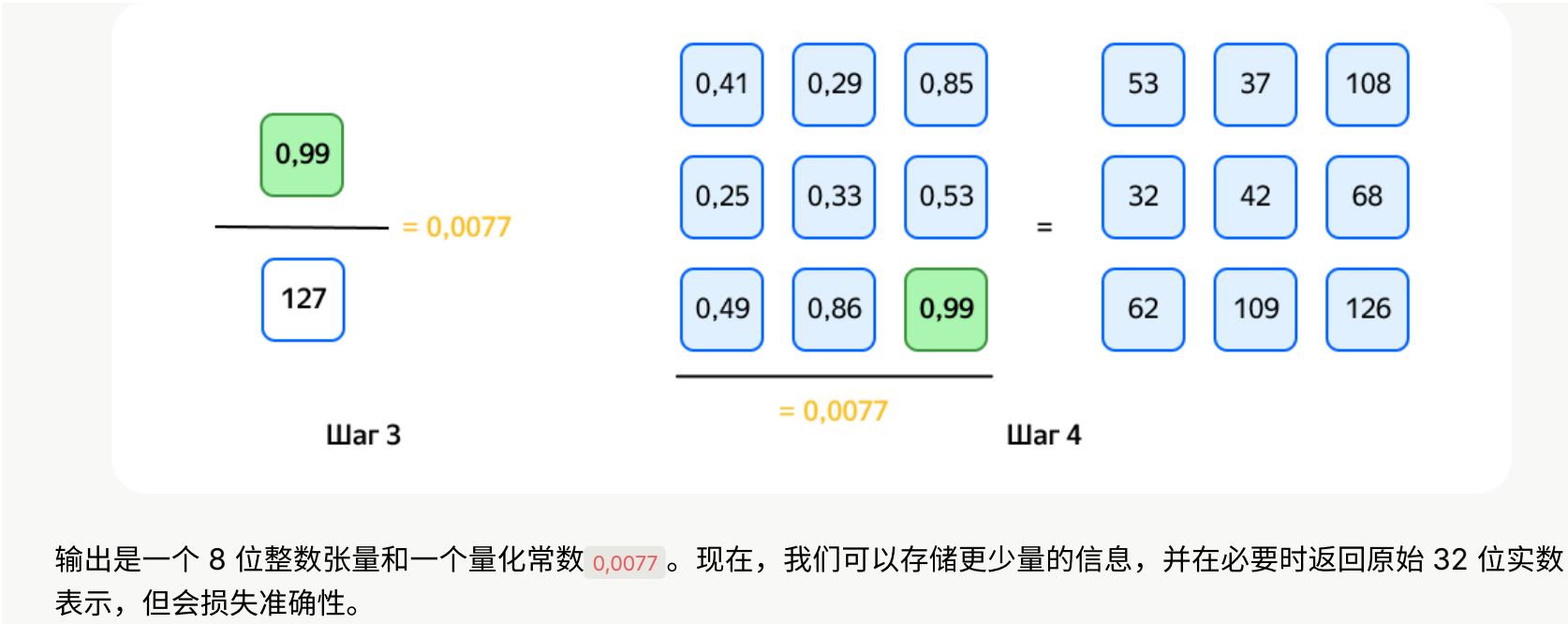
**步骤 1.** 取一个实数张量。

**步骤 2.** 找到最大值。



**步骤 3.**  $S$ 使用公式计算。

**步骤 4.** 量化。



通过量化来提高模型效率

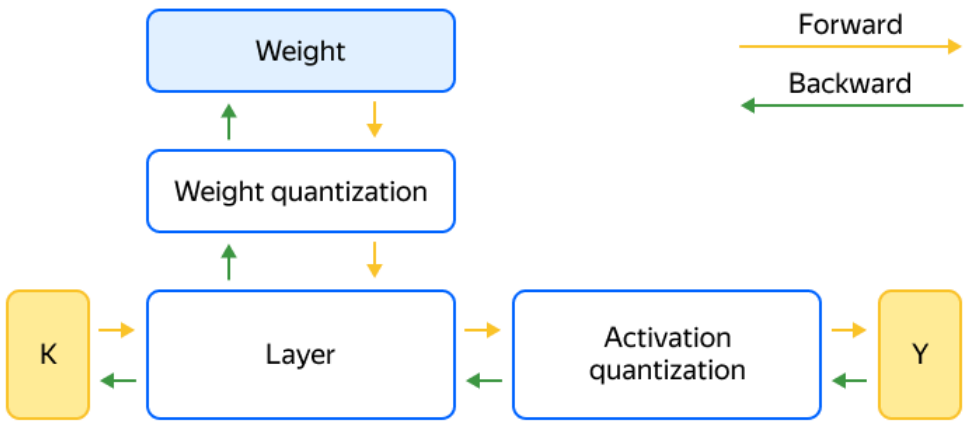
标准的实现方案是量化模型的**权重**。不需要额外的操作，只需套用公式就可以得到量化的结果。

另一种方案，就是量化**激活层输出**。由于激活函数的输出结果范围并不确定，所以又分为两种方法：

- 训练后**静态量化**：每次激活的范围是在**量化时**提前计算的。我们通过预训练的神经网络训练数据集中的数据，并收集统计数据，再通过这些数据算出量化常数。
- 训练后**动态量化**：每次激活的范围是在**运行时**动态计算的。在模型每次进行推理时，根据结果动态计算量化常数。动态量化过程更加复杂且计算成本更高，但常数和量化结果始终保持相关性。

什么时候是量化模型的最佳时机？

我们可以在训练过程中准备网络进行量化；这种方法称为 **量化感知**。为了实现这个功能，神经网络种需要内置特殊块，并在训练期间模拟量化推理。



量化感知训练

模型的层中存储原始的真实权重。在执行 **forward-pass** 之前，它们被使用公式的量化权重所替换。该操作是不可微的，因此梯度直接投射到真实张量中。类似地，特殊块用于量化激活。

量化感知训练很复杂，需要更多的计算资源，但输出是一个“适应”量化值的模型，并且可能更准确。

在后训练的情况下，已经训练的模型被量化。为了量化激活，来自校准数据集的数据另外通过经过训练的网络，收集张量的统计数据，然后进行量化。如果仅量化权重，则不需要数据，因为所有信息已经在张量中。此方法比 Quantize-Aware 更简单、更快，但不够准确。

bits and bytes (BNB)

bitsandbytes 通过 k 位量化为 PyTorch 启用大型语言模型的可访问性提供了实现。bitsandbytes 提供了三个主要功能，可大幅减少推理和训练的内存消耗：

- 8 位优化器使用逐块量化，以极小的内存成本维持 32 位的性能。

- LLM.Int() 或 8 位量化在不降低性能的前提下，仅需一半的内存就可实现大型语言模型推理。此方法基于向量方式的量化，将大多数特征量化为 8 位，并使用 16 位矩阵乘法单独处理异常值。
- QLoRA 或 4 位量化可通过多种节省内存的技术实现大型语言模型训练，且不会影响性能。此方法将模型量化为 4 位，并插入一小组可训练的低秩自适应 (LoRA) 权重矩阵以进行训练。

## 安装

### 系统环境依赖

bitsandbytes 最新的版本基于

OS	CUDA	Compiler
Linux	11.7 - 12.3	GCC 11.4
	12.4+	GCC 13.2
Windows	11.7 - 12.4	MSVC 19.38+ (VS2022 17.8.0+)

MacOS 的支持功能仍在开发中

### 设备依赖

对于 Linux 系统，使用 bitsandbytes 功能需要确保硬件满足以下要求：

Feature	Hardware requirement
LLM.int8()	NVIDIA Turing (RTX 20 series, T4) or Ampere (RTX 30 series, A4-A100) GPUs
8-bit optimizers/quantization	NVIDIA Kepler (GTX 780 or newer)

bitsandbytes 0.39.1 及之后的版本在 pip 安装中不再包含开Kepler的二进制文件。

### 安装指令

```
pip install bitsandbytes
```

## 集成

bitsandbytes 可以与 Hugging Face 以及 PyTorch 生态系统中的许多库集成。

有关更多详细信息，可以参考每个库的关联文档。

## Transformers

使用 Transformer 模型，可以非常轻松地以 4 位或 8 位实时量化的方式加载任何模型。要配置量化参数，只需要在 `transformers.BitsAndBytesConfig` 类中进行定制。

例如，将模型加载并量化为 4 位，并使用 bfloat16 数据类型进行计算：



如果硬件能够支持，bfloat16 是理想的 `compute_dtype`。虽然默认的 `compute_dtype` (float32) 可以确保向后兼容性（支持更多的硬件设备）和数值稳定性，但它占用的空间更大且会减慢计算速度。相比之下，float16 更小且更快，但可能会导致数值不稳定。bfloat16 结合了这两者的优点；它提供了 float32 的数值稳定性以及 16 位数据类型的更少的内存占用和计算速度。

检查设备硬件是否支持 bfloat16，并使用

`transformers.BitsAndBytesConfig` 中的 `bnb_4bit_compute_dtype` 参数进行配置！

```
from transformers import AutoModelForCausalLM, BitsAndBytesConfig
```

```
quantization_config = BitsAndBytesConfig(  
    load_in_4bit=True,
```

```
bnb_4bit_compute_dtype=torch.bfloat16)

model_4bit = AutoModelForCausalLM.from_pretrained(
    "bigscience/bloom-1b7",
    device_map=device_map,
    quantization_config=quantization_config,
)
```

## Accelerate

在 Accelerate 中 bitsandbytes 可以方便地被调用。我们可以通过传递一个带有对应设置的 `accelerate.utils.BnbQuantizationConfig` 来量化任何 PyTorch 模型，然后调用 `accelerate.utils.load_and_quantize_model` 函数对其进行量化。

```
from accelerate import init_empty_weights
from accelerate.utils import BnbQuantizationConfig, load_and_quantize_model
from mingpt.model import GPT

model_config = GPT.get_default_config()
model_config.model_type = 'gpt2-xl'
model_config.vocab_size = 50257
model_config.block_size = 1024

# 创建空模型（没有初始化参数模型）
with init_empty_weights():
    empty_model = GPT(model_config)

bnb_quantization_config = BnbQuantizationConfig(
    load_in_4bit=True,
    bnb_4bit_compute_dtype=torch.bfloat16, # optional
    bnb_4bit_use_double_quant=True,      # optional
    bnb_4bit_quant_type="nf4"           # optional
)

quantized_model = load_and_quantize_model(
    empty_model,
    weights_location=weights_location,
    bnb_quantization_config=bnb_quantization_config,
    device_map = "auto"
)
```

# LLM加速 - vLLM

## vLLM 简介

vLLM是一个专为大语言模型（LLM）加速而设计的框架，旨在提高训练和推理的效率。通过优化计算资源和减少内存占用，VLLM能够显著加快模型的处理速度。

该框架通过以下几种方式实现加速：

- 分布式计算：** 利用多台计算机和多块GPU进行并行计算，从而加速模型训练和推理过程。通过分布式计算，VLLM可以将计算任务分配到多个处理单元上，充分利用硬件资源，显著缩短计算时间。
- 内存优化：** 通过高效的内存管理技术，减少内存占用，提升模型处理效率。内存优化技术包括内存压缩、内存共享和内存复用等，可以有效降低模型的内存需求，使得在相同硬件条件下，能够运行更大规模的模型。
- 自适应批处理：** 动态调整处理批次的大小，以优化计算资源的使用。自适应批处理可以根据运行时的资源情况，灵活调整批处理大小，避免资源浪费，提高整体计算效率。

这些特性使得vLLM在处理大规模语言模型时，能够提供显著的性能提升，是大语言模型开发者的重要工具。

## vLLM的优势



vLLM不仅在性能上有显著提升，还在灵活性和易用性方面具备优势：

- **灵活性**：vLLM支持多种硬件配置，可以在不同的计算环境中运行，包括单机、多机、多GPU等，从而适应不同的应用场景。
- **易用性**：vLLM提供了简洁易用的API接口，开发者可以方便地将其集成到现有的项目中，无需大量的代码修改。
- **兼容性**：vLLM兼容主流的深度学习框架，如TensorFlow和PyTorch，开发者可以在熟悉的环境中使用VLLM，降低学习成本。

## vLLM的应用场景

由于其卓越的性能和灵活性，vLLM在许多领域都有广泛的应用：

- **自然语言处理**：在自然语言生成、机器翻译、情感分析等任务中，vLLM能够加速模型的训练和推理，提高模型的实际应用效果。
- **智能客服**：通过加速语言模型的响应速度，vLLM可以提高智能客服系统的响应效率，提升用户体验。
- **数据分析**：在大规模数据分析任务中，vLLM可以加速数据处理过程，提高分析结果的及时性和准确性。

总之，vLLM作为一个高效的大语言模型加速框架，为开发者提供了强大的工具，使得在处理复杂语言任务时，能够显著提升效率和性能。

作为一个开源项目，vLLM 可以让我们下载模型的权重文件并将其传递给 vLLM 加载，通过它提供的 Python 编程接口来运行推理;

下面是官方文档中的一段代码示例：

```
from vllm import LLM, SamplingParams

prompts = [
    "Hello, my name is",
    "The president of the United States is",
    "The capital of France is",
    "The future of AI is",
]

# 初始化
sampling_params = SamplingParams(temperature=0.8, top_p=0.95)
llm = LLM(model="facebook/opt-125m")

# 进行推理
outputs = llm.generate(prompts, sampling_params)

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

vLLM 类似于 Huggingface 的 **transformers** 库，以下是使用 transformers 对同一模型进行推理的方法：

```
from transformers import pipeline

generator = pipeline('text-generation', model="facebook/opt-125m")
generator("Hello, my name is")
```

在生产环境中，我们通常会提供一个简单的调用接口来与模型交互，以便系统的其它模块可以轻松调用它。一个很好的解决方案是通过 API 公开我们的模型。

假设我们想要借助 Flask 来构建一个 REST API 来为模型提供服务：

```
from flask import Flask, request, jsonify
from vllm import LLM, SamplingParams
```



```
app = Flask(__name__)
sampling_params = SamplingParams(temperature=0.8, top_p=0.95)
llm = LLM(model="facebook/opt-125m")

@app.route('/generate', methods=['POST'])
def generate():
    data = request.get_json()
    prompts = data.get('prompts', [])

    outputs = llm.generate(prompts, sampling_params)

    # 存储输出结果
    results = []

    for output in outputs:
        prompt = output.prompt
        generated_text = output.outputs[0].text
        results.append({
            'prompt': prompt,
            'generated_text': generated_text
        })

    return jsonify(results)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=6000)
```

用户可以通过 `/generate` 路由来调用我们的模型。

但是，这里还存在着很多限制和需求：

- 如果许多用户同时调用，Flask 在尝试并发运行时，会非常容易引起系统崩溃。
- 我们可能还需要实现自定义的身份验证（authentication）机制。
- 添加必要交互性操作辅助：用户需要阅读模型 的 REST API 文档才能与我们的模型进行交互。


而这就是 vLLM 的服务最厉害地方，它为我们提供了以上所有这些问题的解决方案。

如果说 vLLM 的 Python 编程接口类似于 transformers 库，那么 vLLM 的服务器则类似于 IGL。

## 安装 vLLM

安装 vLLM 很简单：

```
pip install vllm
```

 vLLM 需要 Linux 环境且 Python 版本  $\geq 3.8$ 。此外，它还需要一个计算能力  $\geq 7.0$  的 GPU（例如，V100、T4、RTX20xx、A100、L4、H100）。

vLLM 是使用 CUDA 12.1 编译的，因此我们还需要检查一下机器运行的 CUDA 版本。

```
nvcc --version
```

如果运行输出的不是 CUDA 12.1，那么就需要安装当前正在运行的 CUDA 版本编译的 vLLM 版本（请参阅 安装说明 以了解更多信息），或者安装 CUDA 12.1。

## 验证安装

```
python -c 'import torch; print(torch.cuda.is_available())'
```

下一步，创建 `check-vllm.py` 文件并在其中编写如下代码：

```
from vllm import LLM, SamplingParams

prompts = [
    "Mexico is famous for ",
    "The largest country in the world is ",
]

sampling_params = SamplingParams(temperature=0.8, top_p=0.95)
llm = LLM(model="facebook/opt-125m")
responses = llm.generate(prompts, sampling_params)

for response in responses:
    print(response.outputs[0].text)
```

运行脚本，在vLLM 加载模型后，我们将会看到一些输出内容：

```
~~national~~ cultural and artistic art. They've already worked with him.

~~the country~~ a capitalist system with the highest GDP per capita in the world
```


## 启动 vLLM 服务器

在完成了 VLLM 安装后，我们启动服务器。

```
python -m vllm.entrypoints.openai.api_server --model=MODELTORUN
```

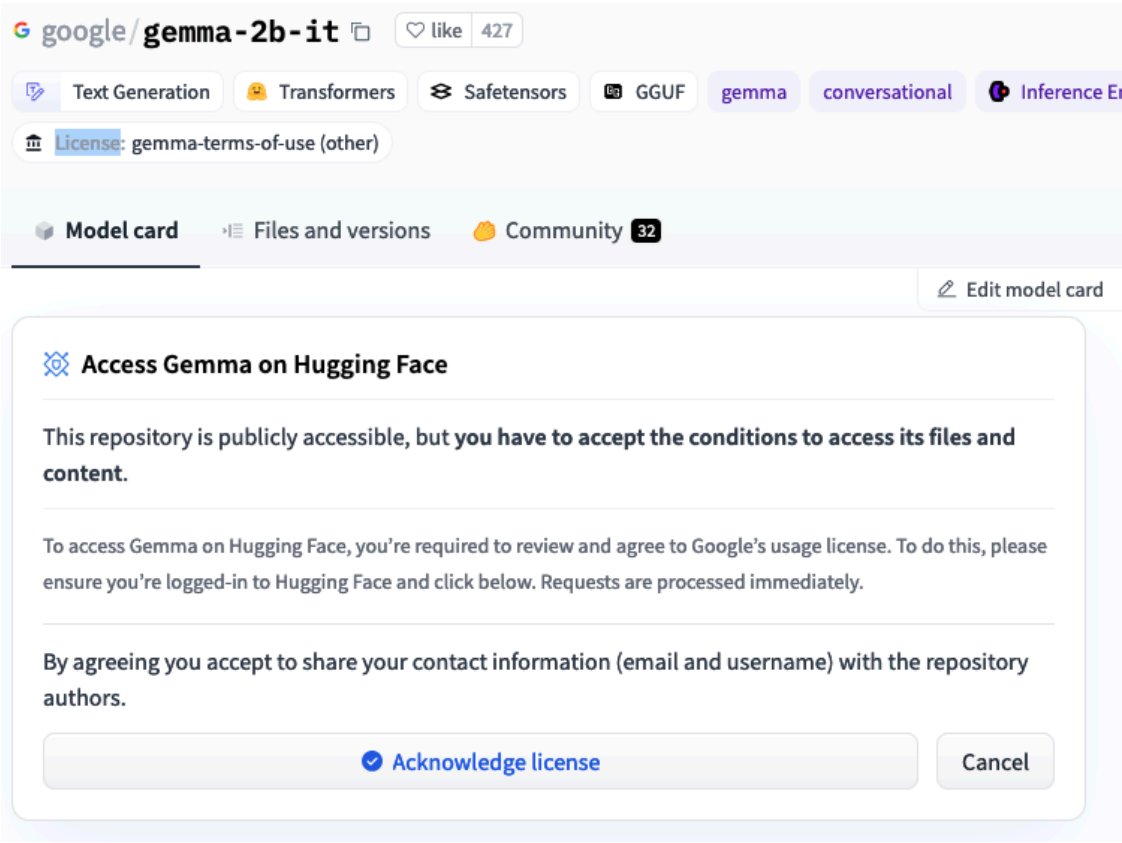
指令中的 `MODELTORUN` 即是我们想要提供服务的模型，例如要提供服务的模型为 `google/gemma-2b`。那么指令就会写成

```
python -m vllm.entrypoints.openai.api_server --model=google/gemma-2b
```



**注意** 某些模型（例如 `google/gemma-2b`）需要我们接受他们的 License，因此，我们需要创建一个 HuggingFace 帐户，接受该模型的 License，并生成一个 Token。

例如，在 HugginFace 上打开 `google/gemma-2b` 时（需要登录），会看到：



接受许可证后，前往 [token section](#) 获取令牌，然后在启动 vLLM 之前，按如下方式设置token：

```
export HF_TOKEN=YOURTOKEN
```

设置 token 后，我们可以顺利地启动服务器了。

💡 即使下载了模型权重，运行时也需要token。否则，会发生以下错误：

```
File "/opt/conda/lib/python3.10/site-packages/huggingface_hub/hf_file_system.py", line 863, in _raise_file_not_found
    raise FileNotFoundError(msg) from err
FileNotFoundError: google/gemma-2b (repository not found)
```

## 设置 dtype

运行一个模型，首先需要考虑的一个重要参数就是 `dtype`，它控制模型权重的数据类型。我们可以需要根据自身 GPU 调整此参数，例如，尝试运行 `google/gemma-2b`：

```
# --dtype=auto is the default value
python -m vllm.entrypoints.openai.api_server --model=google/gemma-2b --dtype=auto
```

在 NVIDIA Tesla T4 上，运行就会产生以下错误：

```
ValueError: Bfloat16 is only supported on GPUs with compute capability of at least 8.0.
Your Tesla T4 GPU has compute capability 7.5. You can use float16 instead by explicitly
setting the `dtype` flag in CLI, for example: --dtype=half.
```

更改 `--dtype` 参数，以便让模型能够在 T4 上运行：

```
python -m vllm.entrypoints.openai.api_server --model=google/gemma-2b --dtype=half
```

如果是第一次使用 `--model` 参数指定模型并启动 vLLM，会需要等待几分钟时间，因为 vLLM 会先下载模型权重。下载模型权重会存储在 `~/.cache` 目录中。之后的初始化加载模型的速度就会快很多了。

由于 Model 必须加载到内存中，因此仍然需要一些时间来加载（取决于模型大小）。

如果我们看到如下消息：

```
INFO: Started server process [428]
INFO: Waiting for application startup.
INFO: Application startup complete.
INFO: Uvicorn running on http://0.0.0.0:80 (Press CTRL+C to quit)
```

则表示 vLLM 已准备好接受客户端发起的请求！

## 发送请求

服务器运行后，我们就可以向其发送请求。

以下是使用 `google/gemma-2b` 和 Python `requests` 库的示例：

```
# 安装requests包: pip install requests
import requests
import json

# 改成自己的主机地址
VLLM_HOST = "https://autumn-snow-1380.ploomberapp.io"
url = f"{VLLM_HOST}/v1/completions"

headers = {"Content-Type": "application/json"}
data = {
    "model": "google/gemma-2b",
    "prompt": "JupySQL is",
    "max_tokens": 100,
    "temperature": 0
}

response = requests.post(url, headers=headers, data=json.dumps(data))

print(response.json()["choices"][0]["text"])
```

得到的response：

```
JupySQL is a Python library that allows you to create and run SQL queries in Jupyter notebooks. It is a powerful too

How does JupySQL work?

JupySQL works by connecting to a database server and executing SQL queries. It supports a wide range of databas

Once you have connected to a database, you can create and run SQL queries in
```

## 使用 OpenAI 客户端

vLLM 公开了一个模仿 OpenAI 的 API 代码，这意味着我们可以使用 OpenAI 的 Python 包来直接调用自定义的 vLLM 服务器。

让我们看一个例子：

```
# NOTE: 安装openai包: pip install openai
from openai import OpenAI

# 我们没有配置身份验证，所以传递一个虚拟值
openai_api_key = "EMPTY"
# 改成自己的主机地址，末尾要添加 "/v1"
openai_api_base = "https://autumn-snow-1380.ploomberapp.io/v1"

client = OpenAI(
    api_key=openai_api_key,
```

```
base_url=openai_api_base,
)
completion = client.completions.create(model="google/gemma-2b",
                                       prompt="JupySQL is",
                                       max_tokens=20)
print(completion.choices[0].text)
```

得到的输出：

```
a powerful SQL editor and IDE. It integrates with Google Jupyter Notebook,
which allows users to create and
```

## 使用聊天 API

前面的示例使用了 completions API。

但我们可能更熟悉聊天 API。请注意，如果我们使用聊天 API，则必须确保使用指令优化模型。

由于 `google/gemma-2b` 并未针对这项功能进行过微调，所以我们改用 `google/gemma-2b-it`，让我们启动 vLLM 服务器以应此模型：

```
python -m vllm.entrypoints.openai.api_server \
--host 0.0.0.0 --port 80 \
--model google/gemma-2b \
--dtype=half
```

现在我们可以使用 `client.chat.completions.create` 函数来向模型发送信息：

```
# NOTE: 安装openai包: pip install openai
from openai import OpenAI

openai_api_key = "EMPTY"
openai_api_base = "https://autumn-snow-1380.ploomberapp.io/v1"

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

chat_response = client.chat.completions.create(
    model="google/gemma-2b-it",
    messages=[
        {"role": "user", "content": "Tell me in one sentence what Mexico is famous for"},
    ]
)
print(chat_response.choices[0].message.content)
```

输出：

```
Mexico is known for its rich culture, vibrant cities, stunning natural beauty,
and delicious cuisine.
```

如果我们之前使用过 OpenAI 的 API，应该还记得 `messages` 参数通常包含一些带有 `{"role": "system", "content": ...}` 的消息：

```
chat_response = client.chat.completions.create(
    model="google/gemma-2b-it",
    messages=[
        {"role": "system", "content": "You're a helpful assistant."},
        {"role": "user", "content": "Tell me in one sentence what Mexico is famous for"},
    ]
)
```

但是，有些模型不支持系统角色，例如 `google/gemma-2b-it` 返回以下信息：

```
BadRequestError: Error code: 400 - {'object': 'error', 'message': 'System role not supported', 'type': 'BadRequestError', 'param': None, 'code': 400}
```

我们可以查看模型的文档，了解如何使用聊天 API。

## 安全设置

默认情况下，vLLM的服务器不会进行任何身份验证。如果我们需要将服务器暴露在 Internet 上，那么就需要设置 API 密钥。

可以按如下方式生成一个密钥：

```
export VLLM_API_KEY=$(python -c 'import secrets; print(secrets.token_urlsafe())')
# print the API key
echo $VLLM_API_KEY
```

启动 vLLM：

```
python -m vllm.entrypoints.openai.api_server --model google/gemma-2b-it --dtype=half
```

现在，我们的服务器将受到保护，所有没有 API key 的请求都将被拒绝。

请注意，在上面的指令中，我们没有传递 `--api-key` 参数，因为 vLLM会自动读取 `VLLM_API_KEY` 环境变量。

通过使用之前的 Python 代码段调用来测试服务器，查看是否具有 API 密钥身份验证，我们将得到以下错误信息：

```
No key: `AuthenticationError: Error code: 401 - {'error': 'Unauthorized'}`
```

要解决这个问题，需要使用正确的 API 密钥初始化 `OpenAI` 客户端：

```
from openai import OpenAI

openai_api_key = "THE_ACTUAL_API_KEY"
openai_api_base = "https://autumn-snow-1380.ploomberapp.io/v1"

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

还有一个基本的安全要求是通过 HTTPS 协议来发布我们的 API。

但是，这需要额外的配置，例如获取 TLS 证书。如果您想跳过这个麻烦的步骤，[请跳到最后一部分](#)，我们将在后面展示用于安全部署 vLLM 服务器的一键式解决方案。

## 生产部署的注意事项

- 部署 vLLM 时，必须确保 API 在崩溃时重新启动（或者物理服务器重新启动）。我们可以使用 `systemd` 等工具执行此操作。
- 为了使我们的模型部署具备可移植性，建议使用 `docker`。
- 请确保固定所有 Python 依赖项，以便更新操作不会破坏已配置好的安装环境（例如，使用 `pip freeze`）。

## 使用 PyTorch 的 docker 镜像

建议使用 [PyTorch 的官方 Docker 镜像](#)，因为它已经预装了 `torch` 和 CUDA 驱动程序。



以下是一个可用的 Dockerfile 示例：

```
FROM pytorch/pytorch:2.1.2-cuda12.1-cudnn8-devel

WORKDIR /srv
RUN pip install vllm==0.3.3 --no-cache-dir

# if the model you want to serve requires you to accept the license terms,
# you must pass a HF_TOKEN environment variable, also ensure to pass a VLLM_API_KEY
# environment variable to authenticate your API
ENTRYPOINT ["python", "-m", "vllm.entrypoints.openai.api_server", \
    "--host", "0.0.0.0", "--port", "80", \
    "--model", "google/gemma-2b-it", \
    # depending on your GPU, you might or might not need to pass --dtype
    "--dtype=half"]
```