2.text_generation_inference

1.简介

Text Generation Inference（TGI）是 HuggingFace 推出的一个项目，作为支持 HuggingFace Inference API 和 Hugging Chat 上的LLM 推理的工具，旨在支持大型语言模型的优化推理

2.主要特性

- 支持张量并行推理
- 支持传入请求 Continuous batching 以提高总吞吐量

> **Note**
>
> Continuous Batching 如何工作?
>
> - 每生成一个 token 后，调度器立即：
>   1. **释放已完成请求的资源**（如 KV 缓存）；
>   2. **将新到达的请求加入当前 batch**（只要显存允许）；
>   3. **重组 batch**，只包含当前仍在生成的请求。
> - 所有请求**异步推进**，互不阻塞。

- 使用 flash-attention 和 Paged Attention 在主流的模型架构上优化用于推理的 transformers 代码。**注意：并非所有模型都内置了对这些优化的支持。**
- 使用bitsandbytes(LLM.int8())和GPT-Q进行量化

> **Note**
>
> **bitsandbytes**（bnb）
>
> - **类型**：运行时动态量化（支持训练 + 推理）
> - 核心技术：
>   - **LLM.int8()**：8-bit 量化，对 >6B 模型几乎无损；
>   - **4-bit NormalFloat** (NF4)：专为 LLM 设计的 4-bit 格式，配合 double quantization。
> - 优点：
>   - 与 Hugging Face Transformers **无缝集成**（`load_in_4bit=True`）；
>   - 支持 **QLoRA 微调**（训练时也能量化）；
>   - 开箱即用，无需预转换模型。
> - 缺点：
>   - 推理时需**动态反量化** → 速度略慢于静态量化；
>   - 小模型（<3B）可能掉点明显。
>
> **GPT-Q**（Generative Post-Training Quantization）
>
> - **类型**：后训练静态量化（仅推理）
> - 核心技术：
>   - 逐层贪心优化，将权重压缩为 **4-bit**；
>   - 使用少量校准数据补偿量化误差；

    - 保留 per-group scale（类似分组量化）。
- 优点：
    - **精度极高**（接近 FP16）；
    - **推理速度快**（权重已静态量化，可配合高效 CUDA kernel）；
    - Hugging Face Hub 上有大量社区预量化模型（如 `TheBloke/Llama-2-13B-GPTQ`）。
- 缺点：
    - **仅支持推理**，不能用于训练；
    - 量化过程慢（需数小时）；
    - 必须使用**预量化模型**，不能动态加载原始 HF 模型。

- 内置服务评估，可以监控服务器负载并深入了解其性能
- 轻松运行自己的模型或使用任何 HuggingFace 仓库的模型
- 自定义提示生成：通过提供自定义提示来指导模型的输出，轻松生成文本
- 使用 Open Telemetry，Prometheus 指标进行分布式跟踪

> **Tip**
>
> **Prometheus 指标**（Metrics）
>
> - **用途**：**监控系统性能**（如吞吐、延迟、资源使用）。
> - 工作方式：
>   - 推理服务暴露 `/metrics` 端点（如 `http://localhost:8000/metrics`）；
>   - 输出结构化指标，例如：
>
>       ```
>       tgi_request_success_total{model="llama2"} 1200
>
>       tgi_batch_current_size 8
>
>       tgi_queue_time_seconds_bucket{le="0.1"} 500
>       ```
>
> - **集成**：可接入 **Grafana** 可视化，构建实时监控面板。
> - **价值**：帮助你了解 **GPU 利用率、batch 大小、请求队列、错误率** 等关键指标。
>
> **OpenTelemetry**（OTel）
>
> - **用途**：**分布式追踪**（Tracing） + 日志 + 指标（三位一体可观测性）。
> - 工作方式：
>   - 为每个请求生成唯一 **trace ID**；
>   - 记录请求在系统中的完整路径：
>   `API Gateway → Prefill → Decoding Step 1 → Step 2 → ... → Response`
>   - 每个环节记录耗时、错误、上下文。
> - **集成**：可对接 **Jaeger、Zipkin、Datadog、New Relic** 等后端。
> - 价值：
>   - 定位**慢请求瓶颈**（是 prefill 慢？还是 decoding 慢？）；
>   - 分析**跨服务调用链**（如 LangChain → LLM Server → 向量数据库）；
>   - 实现 **SLO/SLI 监控**（如“95% 请求应在 2 秒内完成”）。

3.支持的模型

- BLOOM
- FLAN-T5
- Galactica
- GPT-Neox
- Llama
- OPT
- SantaCoder
- Starcoder
- Falcon 7B
- Falcon 40B
- MPT
- Llama V2
- Code Llama

4.适用场景

依赖 HuggingFace 模型，并且不需要为核心模型增加多个adapter的场景。

5.项目架构

整个项目由三部分组成：

- launcher
- router
- serve

Launcher、Router和Server（Python gRPC服务）都是服务的组成部分，它们各自承担不同的职责，共同提供一个完整的文本生成推理服务。以下是它们之间的关系：

- **Launcher**：这是服务的启动器，它负责启动和运行服务。它可能会启动 Router，并设置好所有的路由规则。然后，它会监听指定的地址和端口，等待并处理来自客户端的连接。当接收到一个连接时，它会将连接转发给Router 进行处理。
- **Router**：这是服务的中间件，它的主要职责是**路由和调度请求。**当客户端发送一个请求时，Router 会接收这个请求，然后根据请求的内容和当前的系统状态，决定将请求路由到哪个处理器进行处理。这个处理器可能是Server 中的一个 gRPC 方法。Router 的目的是有效地管理和调度系统资源，提高系统的并发处理能力和响应速度。
- **Server（Python gRPC服务）**：这是服务的核心部分，它实现了文本生成推理的主要逻辑。它提供了一些 gRPC 方法，如 Info、Health、ServiceDiscovery、ClearCache、FilterBatch、Prefill 和 Decode，这些方法用于处理客户端的请求，执行文本生成的推理任务，并返回结果。这个服务可能运行在一个单独的服务器上，独立于Launcher 和 Router。

5.1 launcher 启动器

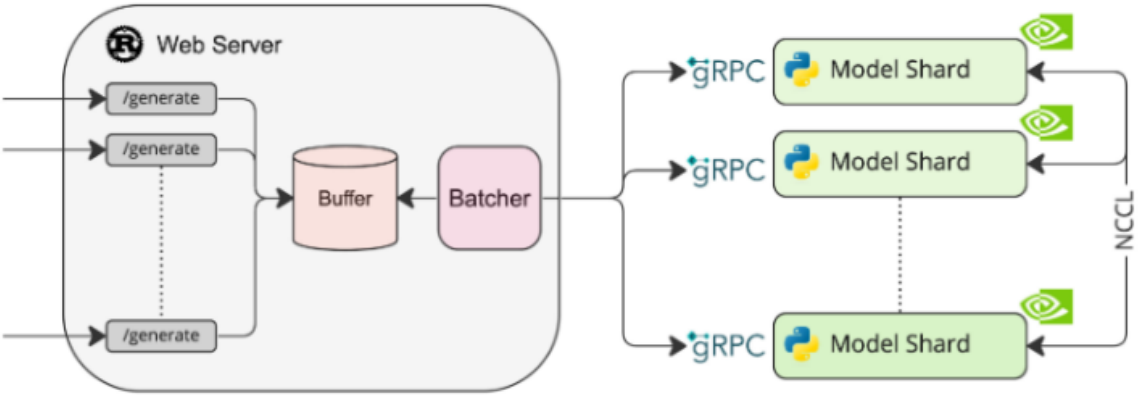
顾名思义，launcher 启动器，就是负责启动的程序，主要做以下工作：(在 launcher/src/main.rs 中)

1. 通过 serve 的命令下载模型，代码中执行的函数为：`download_convert_model(&args, running.clone())?;`
2. 启动 serve ，代码中执行的函数为：`spawn_shards(...)`
3. 启动 router，代码中执行的函数为：`spawn_webserver(args, shutdown.clone(), &shutdown_receiver)?;`

所以，router 和 serve 负责主要的逻辑处理与模型调用。

> **Tip**
>
> 在 LLM 推理框架（如 **Hugging Face Text Generation Inference, TGI**）中：
>
> - Router
>
> （Web Server）运行在 CPU 或前端进程，负责：
>
> - 接收 HTTP/gRPC 请求（来自用户或 API 客户端）；
> - 将请求暂存到 buffer；
> - 根据调度策略（如 Continuous Batching）组装成 batch。
> - Serve
>
> （推理引擎）运行在 GPU 进程（可能在同一台机器，也可能在远程节点），负责：
>
> - 执行 `Prefill` 和 `Decode` 等计算密集型操作；
> - 返回生成的 token。

在项目中有一个架构图，可以更加直观的认识到它们之间的关系，其架构如下图所示：



5.2 router 路由

可以看到 router 这个 webserver 负责接收请求，然后放在 buffer 中，等收集到一定量的数据后，一个 batch 一个 batch 的以 **rpc （远程过程调用）** 的方式发送给 serve 的去处理。

对外暴露的 url 很少同时也很精简，只有四个：

1. `/generate`：一次性生成所有回答的 token
2. `/generate_stream`： 流式的生成所回答的 token (就类似于 chatgpt 一样，一个字一个字的显现)
3. `/metrics`： 获取该服务的 `metrics` 信息。
4. `/info`： 获取模型的相关信息

5.3 serve

在图中，也可以看到，在每个卡上都启动了一个 serve，被叫做 shard，这也是 launcher 的作用之一，通过参数来决定 serve 启动的情况。

在 serve 端的代码，有两个命令行启动脚本（`serve/text_generation_server/cli.py`）：

```
# 下载模型权重的方法
@app.command()
def download_weights(
...
)
...

# 启动 serve 服务的方法
@app.command()
def serve(
...
)
...
```

其实内部逻辑也很简单，稍微处理一下数据后，直接调用 model 的接口来处理。

`Server` 对外暴露了一下接口：（这里说的对外，指的是 router ）

1. Info : 返回 model 信息
2. Health : 检查 serve 的健康状况
3. ServiceDiscovery : 服务发现，实现也很简单，将所有的 serve 的地址发送出去
4. ClearCache : 清除 cache 中的数据 （cache 的功能再看）
5. FilterBatch
6. Prefill
7. Decode

> cache 中的存储单位是 batch （在 router 中提过，router 就是一个 batch 一个 batch 来传的。）

> **Note**
>
> 典型的请求流程如下：
>
> 1. 客户端 → Router：发送 prompt；
> 2. Router → Server：调用 `Info` 确认模型能力；
> 3. Router → Server：调用 `Prefill` 处理 prompt，得到第一个 token；
> 4. Router 将请求加入“活跃队列”；
> 5. 每次迭代：
>   - Router 调用 `Decode` 生成新 token；
>   - 调用 `FilterBatch` 移除已完成请求；
>   - 将新到达的请求通过 `Prefill` 加入 batch；
> 6. 若需运维：
>   - 调用 `Health` 监控状态；

- 调用 `ClearCache` 释放显存；
- 调用 `ServiceDiscovery` 动态扩缩容。

5.4 内部接口的含义

再然后，就剩下最重要的三个功能：FilterBatch、Prefill、Decode

FilterBatch 流程如下：（使用场景还不太清楚）

先从 cache 中以 batch_id 获取特定的 batch 再从 batch 中过滤出我们想要留下的 request_ids（这里的 request_id 指的是 客户端发送的请求 id ） 过滤后，再将 batch 放回 cache 中。

Prefill 的主要功能是：

1. 从 router 接收 batch ,然后根据模型给的 `from_pb` 方法整理一下 batch 中的信息 并且 通过 `tokenizer` 来将相应的词转化成词向量。（from_pb 方法之后在说）
2. 将 整理后的 batch 信息，通过 model 的 generate_token 方法，生成新的 token (也就是预测的词)，同时也会返回 next_batch。（generate_token 方法之后在说）
3. 将 next_batch 存放到 cache 中。
4. 返回消息。

> **Note**
>
> TGI 采用 **Rust（Router） + Python（Serve）** 的混合架构：
>
> - **Router**（用 Rust 写）：接收 HTTP/gRPC 请求，做负载均衡、流控、Continuous Batching 调度；
> - **Serve**（用 Python 写）：运行在 GPU 上，执行 `Prefill` / `Decode` 等推理逻辑。
>
> 两者通过 **gRPC** 通信，而 gRPC 默认使用 **Protocol Buffers** (Protobuf) 作为数据格式。
>
> 因此，Serve 收到的 batch 数据是 Protobuf 对象（如 `Batch`、`Request`），不能直接用于 PyTorch 推理。
> → 需要一个转换函数：`from_pb`。
>
> ## 总结
>
> | 项目 | 说明 |
> | --- | --- |
> | `pb` | Protocol Buffer（gRPC 默认序列化格式） |
> | `from_pb` | 类方法，用于将 Protobuf 对象反序列化为 Python 对象 |
> | **目的** | 实现 Rust Router 与 Python Serve 之间的 |

Decode 的功能也很简单，主要功能是：

1. 通过 request 传入的 batch.id 从 cache 中获取 batch
2. 将这些 batch 通过 model 的 generate_token 方法，生成新的 token，同时会返回 next_batch。
3. 将 next_batch 存放到 cache 中。
4. 返回消息。

主要是第一步，从缓存中获取 batch，这样有两个好处：**第一，request 不需要传输历史的 信息，上下文都在 cache 中；第二，cache 中缓存的是 词向 的信息，所以，在每次预测词的时候，只需要将传入的信息通过词嵌入转化成词向量，其他的信息就不需要再做转化了，减少了大量的计算工作。**

参考资料：

- LLM-text_generation_interfence
- huggingface/text-generation-inference
- 目前业界大模型推理框架很多，各有什么优缺点，应该如何选择？