项目实战7: QLoRA微调 + Flash Attention微调大模型

在这个项目中，我们将会基于ShareGPT数据，使用QLoRA微调Qwen2 7B模型，并且在微调过程中使用Flash Attention技术。 最终实现一个简单的对齐后模型，QLoRA量化技术让我们可以使用低成本的GPU进行大模型的微调。

项目预计所需时间：2-4小时

项目目标

- 理解模型微调整体流程，以及模型低参数量微调相关技术
- 理解FlashAttention技术的使用方法
- 理解数据的准备和预处理流程
- 掌握分解挑战和制定解决方案的方法
- 使用huggingface全家桶 (transformers, PEFT) 等基础工具

如何为一个大模型微调项目制定解决方案

1. 需求分析

定制大模型微调方案的第一步是明确大模型的需求，这一过程通常需要和业务方沟通得到。在需求分析阶段，通常需要关注如下几个方面的问题：

首先需要了解的是下游任务所属的领域，从而为微调任务选择合适的基座模型。大部分情况下，我们并不会考虑从零开始训练模型。选择合适的基座模型，并在基座模型上做finetune会让下游任务的性能更好。比如实现电商领域的问答系统，那么就需要尝试去寻找电商领域的大模型底座。如果需要开发医疗领域的智能客服，那么就需要寻找医疗领域的大模型底座。

其次需要了解模型所要具备的能力，因为大模型所需要具备的能力直接决定了我们所制定的技术方案，这些能力维度一方面决定了我们需要收集那些微调数据，另一方面决定了我们应该使用什么类型的评测数据。如果对于某一种特定能力，没有合适的开源数据（比如适用于特定人设的角色扮演能力，或者适用于特定领域的知识问答能力），那么我们需要单独制定微调数据和评测数据的收集方案。

2. 基础模型的选择

模型选择一般从如下几个维度考虑：

1. **语言**：下游任务的重要语言是什么？中文还是英文？一般来说，如果需要应对的是英文任务，可以考虑选择llama系列的模型。如果需要考虑中文场景，则可以考虑通义千问系列。或者直接无脑选择Qwen。因为Qwen的确是在各个维度上性能都比较高的一个模型系列（此断言适用于2024年9月前后）。在此之前的开源模型都没有很好的后续迭代。

2. **base模型还是chat模型**：选择base模型还是chat模型主要由你手头的数据量所决定。如果微调阶段可以获取的数据量较大（如10B以上的token），或者有很多专业知识要注入，那么可以考虑使用base模型，通过二阶段预训练先注入相关知识，然后使用微调数据激发模型的指令微调能力。如果手头上只有较少的指令微调数据，那么chat模型是一个不错的选择。

3. 数据准备

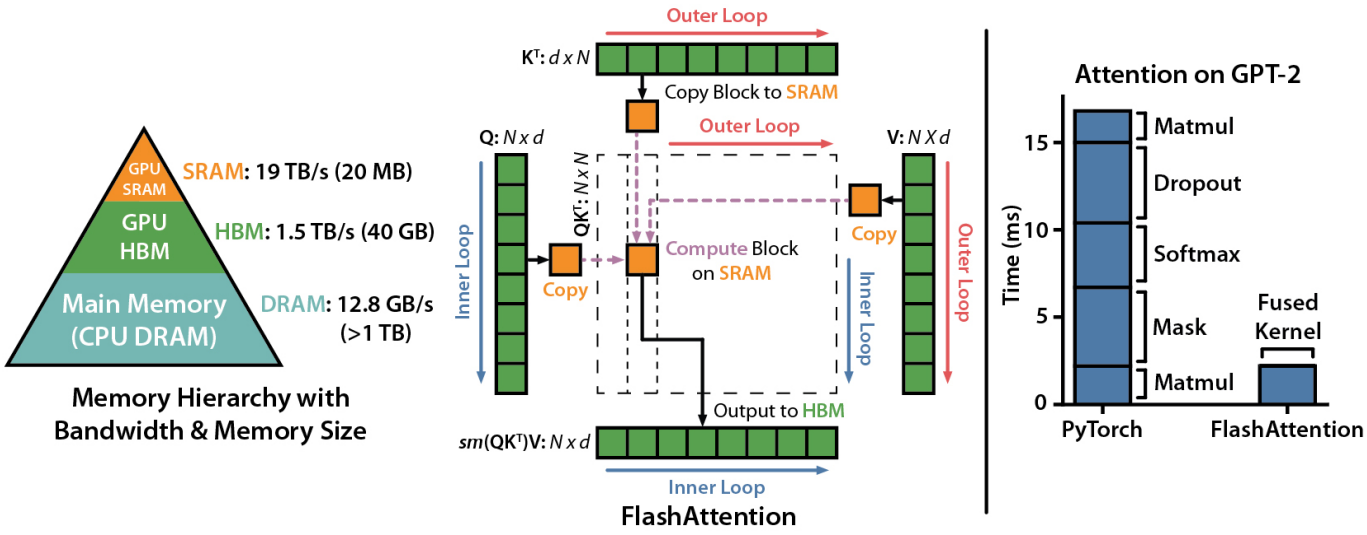
模型训练前，最重要的步骤就是准备数据。这里包括寻找开源数据，构造领域专有数据，数据清洗和数据预处理。 开源数据的寻找可以多借助Huggingface等平台，或者各类github上的资源repo。构造领域专有数据的主要过程是根据特定的领域应用，基于实际业务需求构造所适用的指令微调数据，比如对于医疗领域问答应用，可以寻找特定科室的常见病症，并基于所要回答的问题，构造相应的答案。同理，数据清洗和预处理也通常和具体的领域应用强相关，在本项目中，我们会给大家展示一个智能客服应用的数据准备流程。

4. 模型训练和评估

数据准备完毕之后就进入了模型训练过程，模型训练时可以选择全量训练（Full Fine-tuning）或低参数量微调（PEFT）。关于具体的微调算法在我们的课程中已经有过详细的介绍，大家可以通过课程回放视频进行回顾。

如何使用FlashAttention

下面我们先介绍如何使用FlashAttention，FlashAttention基于Online Attention算法将Attention计算分解为不同的chunk，然后在执行具体Attention计算的过程中优化了大量的IO操作，从而在保证结果无损的情况下，极大地加速了Attention的计算速度。



FlashAttention在使用的过程中主要带来两方面的优势：

1. 会极大提升Attention计算速度，从而加速模型训练和推理过程。
2. 可以节省计算过程中的显存占用，主要原因在于计算每个chunk的attention时不再将中间计算结果保存 在HBM中。 但是这里需要注意的是，目前常见的FlashAttention实现在Nvidia硬件上并不能做到计算结 果的严格可复现(Not Bitwise Reproducible)，其主要原因在于为了优化计算速度，在大部分的实现中 会动态选择不同的GEMM Kernal，从而使得矩阵乘法的计算结果会有细微差异。不过这种差异很小， 一般不会影响模型训练和推理的精度。

安装FlashAttention

FlashAttention的官方实现以Python Package的方式提供，使用前可以参考 官方Github Repo 进行安装。具体来说，FlashAttention的使用需要满足以下前置条件（截止2024年9月）：

CUDA 11.7 and above
PyTorch 1.12 and above

-
-

- `packaging` Python package
- `ninja` Python package
- 推荐使用Linux环境

FlashAttention的安装可以使用如下命令：

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

更多安装选项请参考FlashAttention的官方Gihub Repo。

FlashAttention接口函数说明

FlashAttention的使用相对来说较为简单，它提供了用于计算Attention的CUDA Kernel，并且绑定到了Python API上，只需要使用这个库中所提供的函数替换Pytorch实现中的Attention函数即可，常用的函数包括：

- `flash_attn_func`: 在给定的`q`,`k`,`v`矩阵上执行Attention操作。
- `flash_attn_qkvpacked_func`: 在给定`qkv`矩阵上执行Attention操作，其中`qkv`被pack成了同一个矩阵。
- `flash_attn_with_kvcache`: 在给定的 `kv`缓存上执行Attention操作，常用于增量式解码阶段。

这里列出了`FlashAttention v2.6.3`版本中关于上述函数的使用示例：

```
flash_attn_func(q, k, v, dropout_p=0.0, softmax_scale=None, causal=False,
                window_size=(-1, -1), alibi_slopes=None,
deterministic=False):
"""dropout_p should be set to 0.0 during evaluation
Supports multi-query and grouped-query attention (MQA/GQA) by passing in
KV with fewer heads
than Q. Note that the number of heads in Q must be divisible by the number
of heads in KV.
For example, if Q has 6 heads and K, V have 2 heads, head 0, 1, 2 of Q
will attention to head
0 of K, V, and head 3, 4, 5 of Q will attention to head 1 of K, V.
If window_size != (-1, -1), implements sliding window local attention.
Query at position i
will only attend to keys between
[i + seqlen_k - seqlen_q - window_size[0], i + seqlen_k - seqlen_q +
window_size[1]] inclusive.

Arguments:
    q: (batch_size, seqlen, nheads, headdim)
    k: (batch_size, seqlen, nheads_k, headdim)
    v: (batch_size, seqlen, nheads_k, headdim)
    dropout_p: float. Dropout probability.
    softmax_scale: float. The scaling of QK^T before applying softmax.
        Default to 1 / sqrt(headdim).
    causal: bool. Whether to apply causal attention mask (e.g., for auto-
regressive modeling).
    window_size: (left, right). If not (-1, -1), implements sliding window
local attention.
```

```
    alibi_slopes: (nheads,) or (batch_size, nheads), fp32. A bias of
        (-alibi_slope * |i + seqlen_k - seqlen_q - j|)
        is added to the attention score of query i and key j.
    deterministic: bool. Whether to use the deterministic implementation
of the backward pass,
        which is slightly slower and uses more memory. The forward pass is
always deterministic.
Return:
    out: (batch_size, seqlen, nheads, headdim).
"""
```

```
flash_attn_qkvpacked_func(qkv, dropout_p=0.0, softmax_scale=None,
causal=False,
                          window_size=(-1, -1), alibi_slopes=None,
deterministic=False):
"""dropout_p should be set to 0.0 during evaluation
If Q, K, V are already stacked into 1 tensor, this function will be faster
than
calling flash_attn_func on Q, K, V since the backward pass avoids explicit
concatenation
of the gradients of Q, K, V.
If window_size != (-1, -1), implements sliding window local attention.
Query at position i
will only attend to keys between [i - window_size[0], i + window_size[1]]
inclusive.
Arguments:
    qkv: (batch_size, seqlen, 3, nheads, headdim)
    dropout_p: float. Dropout probability.
    softmax_scale: float. The scaling of QK^T before applying softmax.
        Default to 1 / sqrt(headdim).
    causal: bool. Whether to apply causal attention mask (e.g., for auto-
regressive modeling).
    window_size: (left, right). If not (-1, -1), implements sliding window
local attention.
    alibi_slopes: (nheads,) or (batch_size, nheads), fp32. A bias of (-
alibi_slope * |i - j|) is added to
        the attention score of query i and key j.
    deterministic: bool. Whether to use the deterministic implementation
of the backward pass,
        which is slightly slower and uses more memory. The forward pass is
always deterministic.
Return:
    out: (batch_size, seqlen, nheads, headdim).
"""
```

```
def flash_attn_with_kvcache(
    q,
    k_cache,
    v_cache,
```

```
    k=None,
    v=None,
    rotary_cos=None,
    rotary_sin=None,
    cache_seqlens: Optional[Union[(int, torch.Tensor)]] = None,
    cache_batch_idx: Optional[torch.Tensor] = None,
    block_table: Optional[torch.Tensor] = None,
    softmax_scale=None,
    causal=False,
    window_size=(-1, -1),  # -1 means infinite context window
    rotary_interleaved=True,
    alibi_slopes=None,
):
    """
    If k and v are not None, k_cache and v_cache will be updated *inplace*
with the new values from
    k and v. This is useful for incremental decoding: you can pass in the
cached keys/values from
    the previous step, and update them with the new keys/values from the
current step, and do
    attention with the updated cache, all in 1 kernel.

    If you pass in k / v, you must make sure that the cache is large
enough to hold the new values.
    For example, the KV cache could be pre-allocated with the max sequence
length, and you can use
    cache_seqlens to keep track of the current sequence lengths of each
sequence in the batch.

    Also apply rotary embedding if rotary_cos and rotary_sin are passed
in. The key @k will be
    rotated by rotary_cos and rotary_sin at indices cache_seqlens,
cache_seqlens + 1, etc.
    If causal or local (i.e., window_size != (-1, -1)), the query @q will
be rotated by rotary_cos
    and rotary_sin at indices cache_seqlens, cache_seqlens + 1, etc.
    If not causal and not local, the query @q will be rotated by
rotary_cos and rotary_sin at
    indices cache_seqlens only (i.e. we consider all tokens in @q to be at
position cache_seqlens).

    See tests/test_flash_attn.py::test_flash_attn_kvcache for examples of
how to use this function.

    Supports multi-query and grouped-query attention (MQA/GQA) by passing
in KV with fewer heads
    than Q. Note that the number of heads in Q must be divisible by the
number of heads in KV.
    For example, if Q has 6 heads and K, V have 2 heads, head 0, 1, 2 of Q
will attention to head
    0 of K, V, and head 3, 4, 5 of Q will attention to head 1 of K, V.

    If causal=True, the causal mask is aligned to the bottom right corner
of the attention matrix.
```

```
    For example, if seqlen_q = 2 and seqlen_k = 5, the causal mask (1 =
keep, 0 = masked out) is:
        1 1 1 1 0
        1 1 1 1 1
    If seqlen_q = 5 and seqlen_k = 2, the causal mask is:
        0 0
        0 0
        0 0
        1 0
        1 1
    If the row of the mask is all zero, the output will be zero.

    If window_size != (-1, -1), implements sliding window local attention.
Query at position i
    will only attend to keys between
    [i + seqlen_k - seqlen_q - window_size[0], i + seqlen_k - seqlen_q +
window_size[1]] inclusive.

    Note: Does not support backward pass.

    Arguments:
        q: (batch_size, seqlen, nheads, headdim)
        k_cache: (batch_size_cache, seqlen_cache, nheads_k, headdim) if
there's no block_table,
            or (num_blocks, page_block_size, nheads_k, headdim) if there's
a block_table (i.e. paged KV cache)
            page_block_size must be a multiple of 256.
        v_cache: (batch_size_cache, seqlen_cache, nheads_k, headdim) if
there's no block_table,
            or (num_blocks, page_block_size, nheads_k, headdim) if there's
a block_table (i.e. paged KV cache)
        k [optional]: (batch_size, seqlen_new, nheads_k, headdim). If not
None, we concatenate
            k with k_cache, starting at the indices specified by
cache_seqlens.
        v [optional]: (batch_size, seqlen_new, nheads_k, headdim). Similar
to k.
        rotary_cos [optional]: (seqlen_ro, rotary_dim / 2). If not None,
we apply rotary embedding
            to k and q. Only applicable if k and v are passed in.
rotary_dim must be divisible by 16.
        rotary_sin [optional]: (seqlen_ro, rotary_dim / 2). Similar to
rotary_cos.
        cache_seqlens: int, or (batch_size,), dtype torch.int32. The
sequence lengths of the
            KV cache.
        block_table [optional]: (batch_size, max_num_blocks_per_seq),
dtype torch.int32.
        cache_batch_idx: (batch_size,), dtype torch.int32. The indices
used to index into the KV cache.
            If None, we assume that the batch indices are [0, 1, 2, ...,
batch_size - 1].
            If the indices are not distinct, and k and v are provided, the
values updated in the cache
```

```
                might come from any of the duplicate indices.
        softmax_scale: float. The scaling of QK^T before applying softmax.
            Default to 1 / sqrt(headdim).
        causal: bool. Whether to apply causal attention mask (e.g., for
auto-regressive modeling).
        window_size: (left, right). If not (-1, -1), implements sliding
window local attention.
        rotary_interleaved: bool. Only applicable if rotary_cos and
rotary_sin are passed in.
            If True, rotary embedding will combine dimensions 0 & 1, 2 &
3, etc. If False,
            rotary embedding will combine dimensions 0 & rotary_dim / 2, 1
& rotary_dim / 2 + 1
            (i.e. GPT-NeoX style).
        alibi_slopes: (nheads,) or (batch_size, nheads), fp32. A bias of
            (-alibi_slope * |i + seqlen_k - seqlen_q - j|)
            is added to the attention score of query i and key j.

    Return:
        out: (batch_size, seqlen, nheads, headdim).
    """
```

上述三个函数有其所对应的三个变种，用于处理变长的序列：

- flash_attn_varlen_func
- flash_attn_varlen_qkvpacked_func
- flash_attn_varlen_kvpacked_func

这些函数在实现的过程中可以节省不必要的Attention操作，从而节省计算量，提升运算速度。 这里列出了FlashAttention v2.6.3版本中关于flash_attn_varlen_func函数的函数签名：

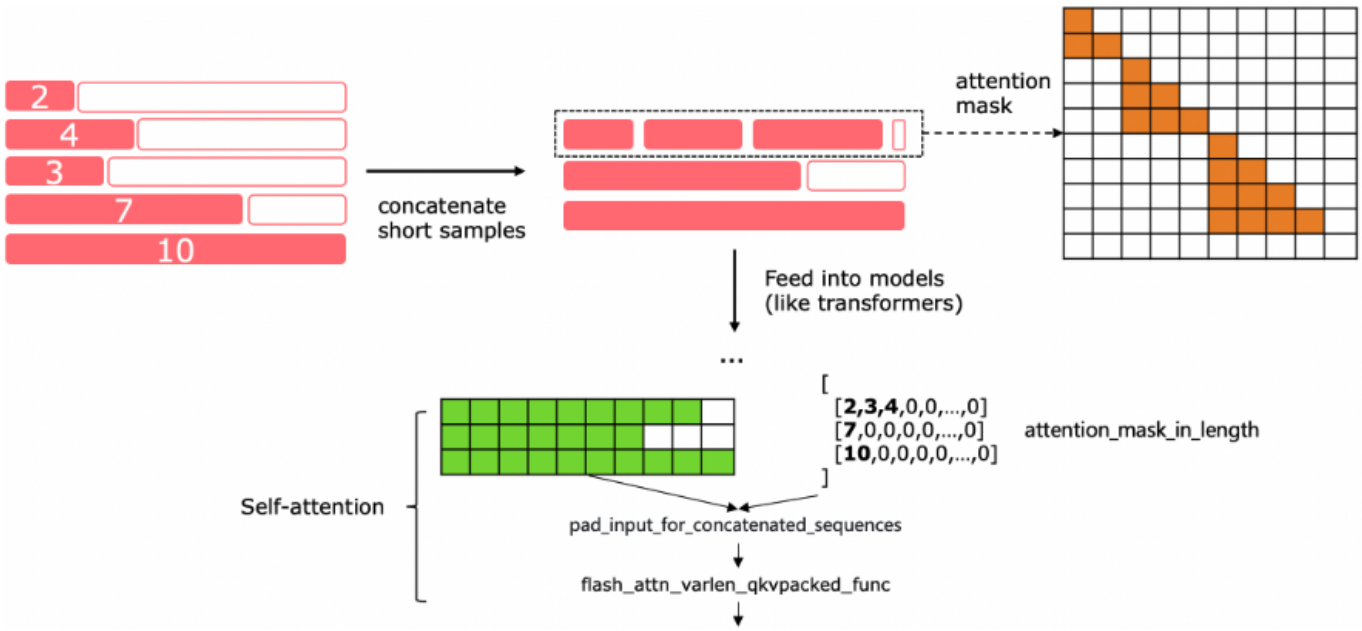
```
def flash_attn_varlen_func(
    q,
    k,
    v,
    cu_seqlens_q,
    cu_seqlens_k,
    max_seqlen_q,
    max_seqlen_k,
    dropout_p=0.0,
    softmax_scale=None,
    causal=False,
    window_size=(-1, -1),  # -1 means infinite context window
    softcap=0.0, # 0.0 means deactivated
    alibi_slopes=None,
    deterministic=False,
    return_attn_probs=False,
    block_table=None,
):
    """dropout_p should be set to 0.0 during evaluation
    Supports multi-query and grouped-query attention (MQA/GQA) by passing
```

```
in K, V with fewer heads
    than Q. Note that the number of heads in Q must be divisible by the
number of heads in KV.
    For example, if Q has 6 heads and K, V have 2 heads, head 0, 1, 2 of Q
will attention to head
    0 of K, V, and head 3, 4, 5 of Q will attention to head 1 of K, V.

    If causal=True, the causal mask is aligned to the bottom right corner
of the attention matrix.
    For example, if seqlen_q = 2 and seqlen_k = 5, the causal mask (1 =
keep, 0 = masked out) is:
        1 1 1 1 0
        1 1 1 1 1
    If seqlen_q = 5 and seqlen_k = 2, the causal mask is:
        0 0
        0 0
        0 0
        1 0
        1 1
    If the row of the mask is all zero, the output will be zero.

    If window_size != (-1, -1), implements sliding window local attention.
Query at position i
    will only attend to keys between
    [i + seqlen_k - seqlen_q - window_size[0], i + seqlen_k - seqlen_q +
window_size[1]] inclusive.

    Arguments:
        q: (total_q, nheads, headdim), where total_q = total number of
query tokens in the batch.
        k: (total_k, nheads_k, headdim), where total_k = total number of
key tokens in the batch.
        v: (total_k, nheads_k, headdim), where total_k = total number of
key tokens in the batch.
        cu_seqlens_q: (batch_size + 1,), dtype torch.int32. The cumulative
sequence lengths
           of the sequences in the batch, used to index into q.
        cu_seqlens_k: (batch_size + 1,), dtype torch.int32. The cumulative
sequence lengths
           of the sequences in the batch, used to index into kv.
        max_seqlen_q: int. Maximum query sequence length in the batch.
        max_seqlen_k: int. Maximum key sequence length in the batch.
        dropout_p: float. Dropout probability.
        softmax_scale: float. The scaling of QK^T before applying softmax.
            Default to 1 / sqrt(headdim).
        causal: bool. Whether to apply causal attention mask (e.g., for
auto-regressive modeling).
        window_size: (left, right). If not (-1, -1), implements sliding
window local attention.
        softcap: float. Anything > 0 activates softcapping attention.
        alibi_slopes: (nheads,) or (batch_size, nheads), fp32. A bias of
            (-alibi_slope * |i + seqlen_k - seqlen_q - j|)
            is added to the attention score of query i and key j.
        deterministic: bool. Whether to use the deterministic
```

```
    implementation of the backward pass,
                which is slightly slower and uses more memory. The forward
    pass is always deterministic.
            return_attn_probs: bool. Whether to return the attention
    probabilities. This option is for
               testing only. The returned probabilities are not guaranteed to
    be correct
               (they might not have the right scaling).
        Return:
            out: (total, nheads, headdim).
            softmax_lse [optional, if return_attn_probs=True]: (nheads,
    total_q_seqlen). The
                logsumexp of each row of the matrix QK^T * scaling (e.g., log
    of the softmax
                normalization factor).
            S_dmask [optional, if return_attn_probs=True]: (batch_size,
    nheads, seqlen, seqlen).
                The output of softmax (possibly with different scaling). It
    also encodes the dropout
                pattern (negative means that location was dropped, nonnegative
    means it was kept).
        """
```

具体使用示例可以参考这里。

一般来说，上述三个处理变长序列Attention的函数的输入可以使用`unpad_input_for_concatenated_sequences`函数辅助处理。这一函数可以帮助将若干短序列拼接在一起，生成上述三个变长序列Attention函数的输入，具体的使用示例如下：



这里提供一段伪代码：

```
        qkv = torch.stack(
            [query_states, key_states, value_states], dim=2
        )  # [bsz, nh, 3, q_len, hd]
```

```
        qkv = qkv.transpose(1, 3)  # [bsz, q_len, 3, nh, hd]
        nheads = qkv.shape[-2]
        x = rearrange(qkv, "b s three h d -> b s (three h d)")
        x_unpad, indices, cu_q_lens, max_s =
unpad_input_for_concatenated_sequences(x, attention_mask_in_length)
        x_unpad = rearrange(
            x_unpad, "nnz (three h d) -> nnz three h d", three=3, h=nheads
        )
        output_unpad = flash_attn_varlen_qkvpacked_func(
            x_unpad, cu_q_lens, max_s, 0.0, softmax_scale=None,
causal=True
        )
        output = rearrange(
              pad_input(
                  rearrange(output_unpad, "nnz h d -> nnz (h d)"),
indices, bsz, q_len
              ),
              "b s (h d) -> b s h d",
              h=nheads,
          )
        attn_output = rearrange(output, "b s h d -> b s (h d)")
```

FlashAttention使用示例

这里我们提供了一套示例代码来演示如何使用`flash-attn`所提供的官方实现计算self-attention的值。详见`flash-attn.ipynb`。 注意，运行该示例需要一台配备了Nivida GPU的服务器，因为`flash-attn`官方实现中要求配置为`CUDA 11.7 and above`。

微调Qwen2.5-3B-Instruct模型

接下来，我们会正式演示如何使用QLoRA方法和Flash Attention微调Qwen系列模型。 这里我们选择了一个参数量并不大的模型作为演示：Qwen2.5-3B-Instruct。

下载模型

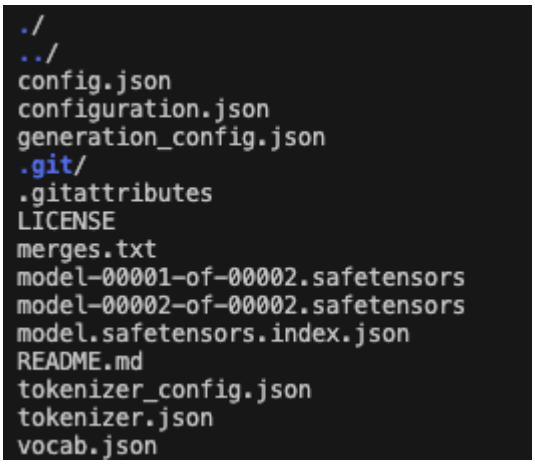
首先我们需要下载该模型，大家可以在Huggingface的模型仓库中下载这一模型：

```
git lfs install
# 需科学上网，不然会因为神秘力量下载很慢
git clone https://huggingface.co/Qwen/Qwen2.5-3B-Instruct
```

如果不能顺畅地科学上网的话，可以选择在ModelScope上下载Qwen模型：

```
git lfs install
git clone https://www.modelscope.cn/Qwen/Qwen2.5-3B-Instruct.git
```

下载完之后的模型目录如图所示



配置环境

在微调模型前需要准备相应的环境：

```
%pip install transformers=='4.45.2' peft=='0.13.1' accelerate=='1.0.0'
tiktoken=='0.8.0' bitsandbytes=='0.44.1'
```

这里简单介绍一下这几个包的用处：

1. `transformers`: 这个库提供了常见预训练模型的实现，并且基本上已经成为了自然语言处理任务的标准库。被广泛地应用在各类NLP相关应用中。
2. `peft`: 这个库和`transformers`一样，都是由`huggingface`这个组织所维护和开发的。它的主要功能是提供了各类低参数量微调算法的实现，比如Lora等。
3. `bitsandbytes`: 这个库提供了很多高效的位操作和字节级别的处理能力，非常适合深度学习任务中对于性能要求比较高的场景，并且集成了各类低比特量化算法的实现。
4. `accelerate`: 这个库也是由`huggingface`这个组织所维护和开发的，是`huggingface`全家桶中的重要一员。它提供了多种并行训练功能，并且支持多GPU训练。这个库已经被深度集成在`huggingface`全家桶中。

导入数据

本项目使用一个医疗对话数据集演示QLoRA + FlashAttention的微调流程。

医疗对话数据集地址：https://github.com/Toyhom/Chinese-medical-dialogue-data

微调自我认知数据集：https://github.com/SmartFlowAI/Llama3-Tutorial/blob/main/data/self_cognition.json

自我认知数据集的目的是想要让我们的模型知道自己的名字。并且由于资源有限，我们只加载了一份儿科数据进行微调。如果大家的资源足够充足，可以加载更多数据进行微调训练。具体处理处理对应的代码可以参考`flash-attn.ipynb`代码文件，关键部分添加了详细注释。

配置模型

下一步就是对模型进行一些LoRA和量化参数的配置。我们使用如下量化配置：

```
compute_dtype = getattr(torch, "float16")

quant_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=compute_dtype,
    bnb_4bit_use_double_quant=True,
)
```

其中

- compute_dtype 指定了训练所用的数据类型，这里使用`fp16`
- `BitsAndBytesConfig`是用于配置量化参数的类
- load_in_4bit=True 这个参数指定了模型在加载时是否使用4bit量化。设置为`True`意味着模型在加载时会进行4bit量化
- `bnb_4bit_quant_type="nf4"` 这个参数定义了4bit量化的类型。 `nf4` 代表`non-functional 4-bit`量化，这是一种非功能性的量化方法，通常用于测试和实验，因为它不会改变模型的权重值。
- `bnb_4bit_compute_dtype=compute_dtype` 这个参数设置了在4bit位量化计算时使用的数值类型。这里使用了之前定义的`compute_dtype` ，即 `float16` 。这意味4bit权重在进行计算时，会先将权重反量化为16位浮点数，然后使用16位浮点数对应的运算指令计算。
- `bnb_4bit_use_double_quant=True` 这个参数指定是否使用双重量化。双重量化是一种技术，即将量化缩放系数再次量化，可以在不损失太多精度的情况下进一步减少模型大小。

关于LoRA参数的设置请参见代码中的注释。

模型训练

模型训练前我们需要先加载模型，并且初始化我们要微调的LoRA参数。预训练模型的加载可以直接使用`transformers`库所提供的`from_pretrained`工具函数实现。具体加载命令请参考代码注释。在加载模型时，可以使用`attn_implementation`参数指定所要使用的attention实现是哪一种，在这里可以直接指定`flash_attention_2`，在`transformers`库中的QWen模型的实现代码中，会根据这一参数决定使用那种实现方式，在QWen模型的实现代码中，这一逻辑如下所示：

```
QWEN2_ATTENTION_CLASSES = {
    "eager": Qwen2Attention,
    "flash_attention_2": Qwen2FlashAttention2,
    "sdpa": Qwen2SdpaAttention,
}


class Qwen2DecoderLayer(nn.Module):
    def __init__(self, config: Qwen2Config, layer_idx: int):
        super().__init__()
        self.hidden_size = config.hidden_size

        if config.sliding_window and config._attn_implementation !=
"flash_attention_2":
            logger.warning_once(
                f"Sliding Window Attention is enabled but not implemented
```

```
for `{config._attn_implementation}`; "
                "unexpected results may be encountered."
            )
        self.self_attn =
QWEN2_ATTENTION_CLASSES[config._attn_implementation](config, layer_idx)

        self.mlp = Qwen2MLP(config)
        self.input_layernorm = Qwen2RMSNorm(config.hidden_size,
eps=config.rms_norm_eps)
        self.post_attention_layernorm = Qwen2RMSNorm(config.hidden_size,
eps=config.rms_norm_eps)
...
```

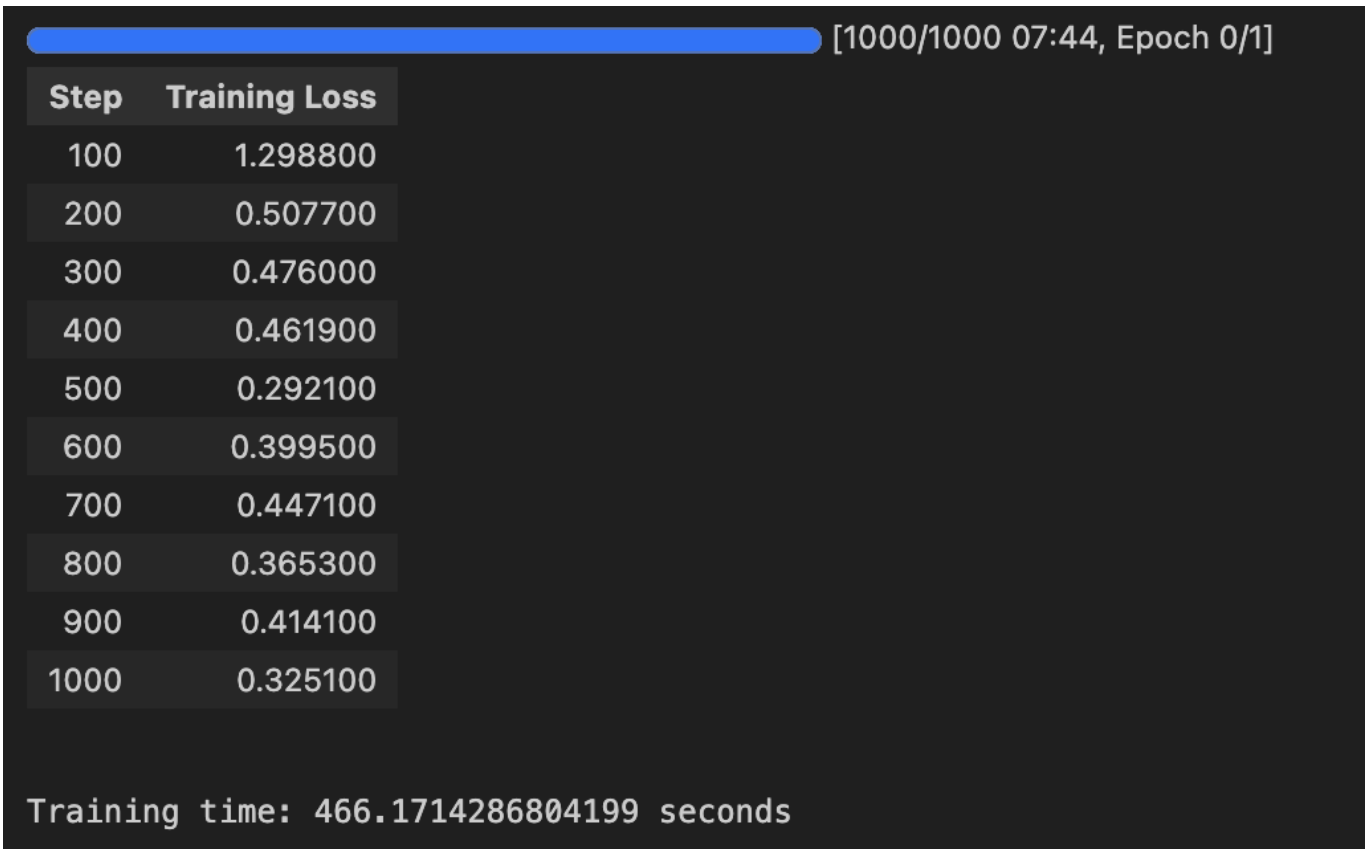
从上述代码中可以看出，指定了`flash_attention_2`之后，代码会使用`Qwen2FlashAttention2`这个类来实现attention操作，其具体实现可以参考源码

模型的训练过程使用`transformers`库所提供的`Trainer`类实现，这个类中实现了非常完备的训练循环，并且可以通过各种参数设置训练过程。具体所使用的参数请参考代码注释。

开始训练的命令如下：

```
peft_trainer.train()
```

可以从Notebook中看到，训练可以如期进行，并且loss也在逐渐下降。



总结

本项目演示了如何结合`transformers`库，使用了QLoRA + FlashAttention微调Qwen模型，由于LLM社区近些年非常活跃，因此相关的开源库发展很快，所以大部分的功能已经非常完善，大家可以通过简单的几行代码调用即可实现相应的微调功能。比如只需要指定一个参数就可以在微调模型的时候轻松使用到FlashAttention所实现的注意力机制。建议大家在实践过程中尽可能多地熟悉`transformers`全家桶。

本项目也演示了如何调用原始的flash-attn库中所提供的attention函数。