动手实现 LoRA - LoRA from scratch


Xode
没啥想法，到处看看

继上一篇练习文章**《从头预训练一只超迷你 LLaMA 3——复现 Tiny Stories》**后，这次练习一下从零实现 **LoRA**，也就是用 torch 从头实现一遍 LoRA 微调的流程。LoRA 是目前应用最广的微调方法，现在许多微调方法都是 LoRA 及其变体。

在本文你将看到：

- 如何用 pytorch 从零实现 LoRA，表现会近似于 huggingface 的 peft 库（代码实现不一定）
- 如何卸载保存自实现的 LoRA 适配器和重载自实现的 LoRA 适配器
- 如何用自实现的 LoRA 微调 LLM，支持 hugging face 格式的模型
- 如何合并自实现的 LoRA 适配器，并上传到 hugging face
- （待更新）不同的 LoRA 初始化方法
- （待更新）不同的 LoRA 改进方法

在本文看不到：

- LoRA 的应用原理详解
- LoRA 的优化原理详解
- QLoRA、INT8 等量化 LoRA 方式

全文代码主要是以 jupyter notebook 的形式展开的，并不是 `.py` 文件的形式，也就是说前面执行的变量会在中间储存下来。

这次除了 torch，用到的库有：

```
transformers
peft
datasets
accelearate
```

```
tqdm
einops
```

具体可以见仓库的 `requirements.txt` 。

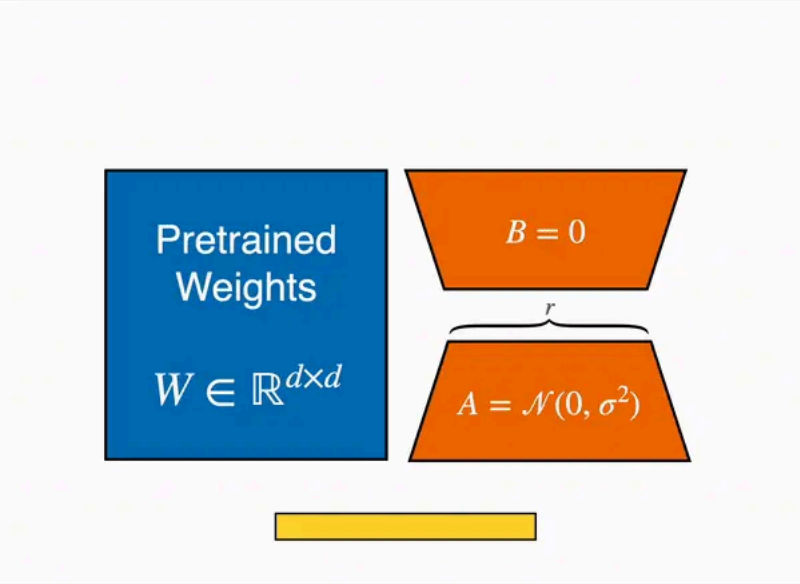
这次的代码和上一次文章的代码整合到一起，放在这里：**Mxoder/LLM-from-scratch**。

---

1. LoRA 原理

先稍微回顾一下 LoRA 原理：

通俗地说，LoRA 基于一个低秩假设：大模型的参数是冗余和稀疏的，其内在秩（可以理解为表征能力）是非常低的，因此可以用一种“降维”的方式来优化学习。



LoRA 原理示意动图，来源：https://huggingface.co/blog/4bit-transformers-bitsandbytes

上面这张图想必大家都见得非常多了。简单来说，LoRA 其实就是用两个更低秩的矩阵 $\boldsymbol{A_{r\times k}}$ 和 $\boldsymbol{B_{d\times r}}$ 来近似原参数矩阵 $\boldsymbol{W_{d\times k}}$ 的效果，将原矩阵冻结以后，前向结果是原矩阵与新的近似矩阵结果的和，但反向的时候只更新 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 两个低秩矩阵，从而大大降低了需要训练的参数，因此属于参数高效微调（PEFT，Parameter Efficient Fine-tuning）的一种。

所以，我们如果要对一个模型应用 LoRA，其实就是把 $\boldsymbol{W}$ 替换为 $\boldsymbol{W + BA}$，非常简单！

那么，这个 $\boldsymbol{W}$ 是什么呢？我们要对什么参数矩阵上 LoRA 呢？其实，模型中**所有的线性层**都是我们可以替换的目标，一个线性层其实就是一个矩阵（先不考虑偏置），例如一个这样的线性层：

```
W = nn.Linear(in_features, out_features, bias=False)
```

它其实就是一个形状为 `(out_features, in_features)` 的权重矩阵。

假设我们有个输入 `x` ，形状为 `(batch_size, in_features)` ，那么输入这个线性层 `W` 后，就相当于做一次矩阵乘法，出来的结果形状就是 `(batch_size, out_features)` 。

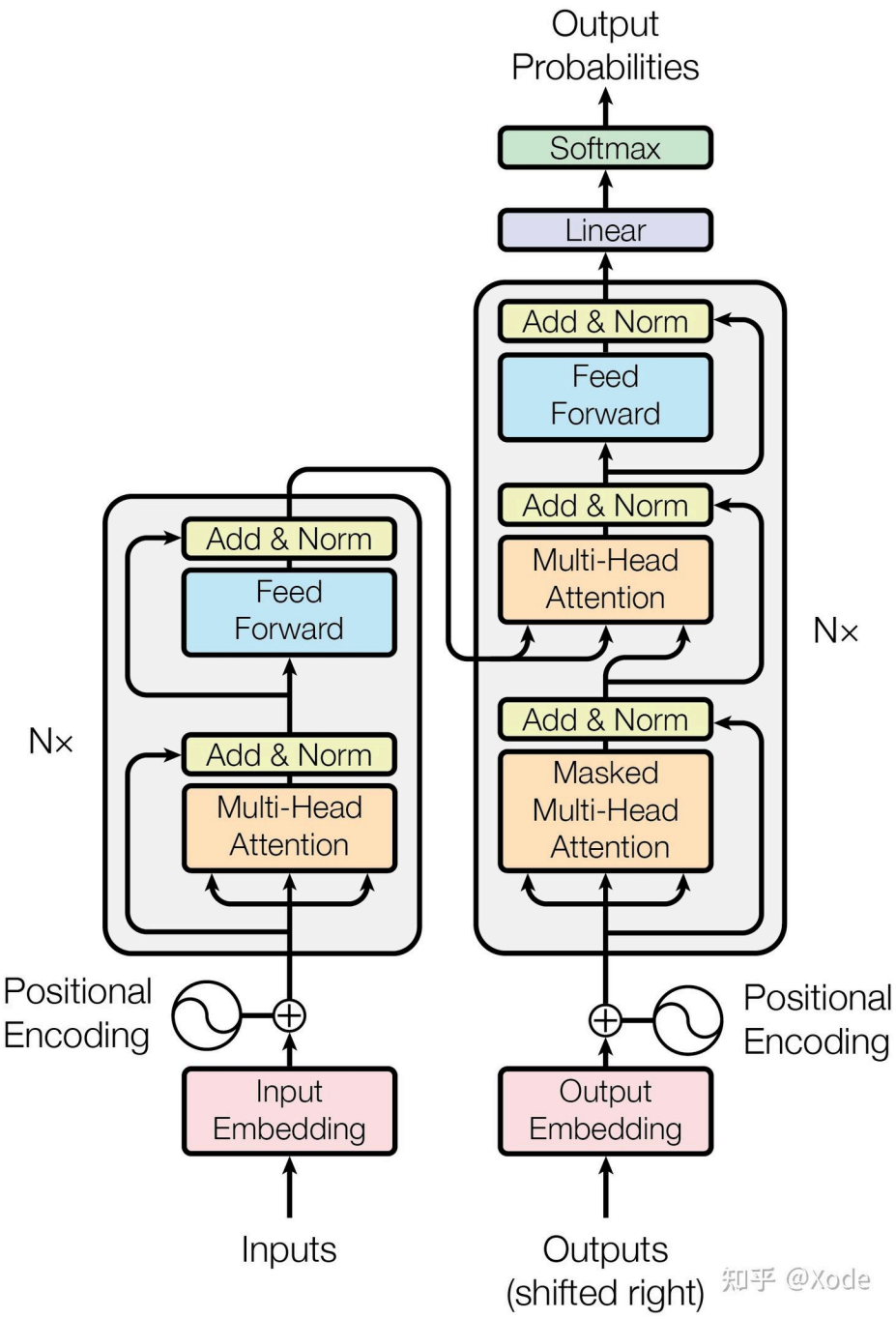
和上面 LoRA 的示意图一样，我们只需要给这个 `W` 加上两个近似线性层（矩阵） `A` 和 `B` 就好了：

```
W = nn.Linear(in_features, out_features, bias=False)

# 新增的 A 和 B，此处 r 即对应原论文中的低秩 rank
lora_A = nn.Linear(in_features, r, bias=False)
```

```
lora_B = nn.Linear(r, out_features, bias=False)

# 原来的前向: Wx
W(x)
# 新的前向: Wx + BAx
W(x) + lora_B(lora_A(x))
```

既然这么简单，我们只需要找到想替换的线性层就好了。那么对于一个模型——特别是我们常见的 decoder-only 架构的 Causal LM，有哪些线性层可以替换呢？

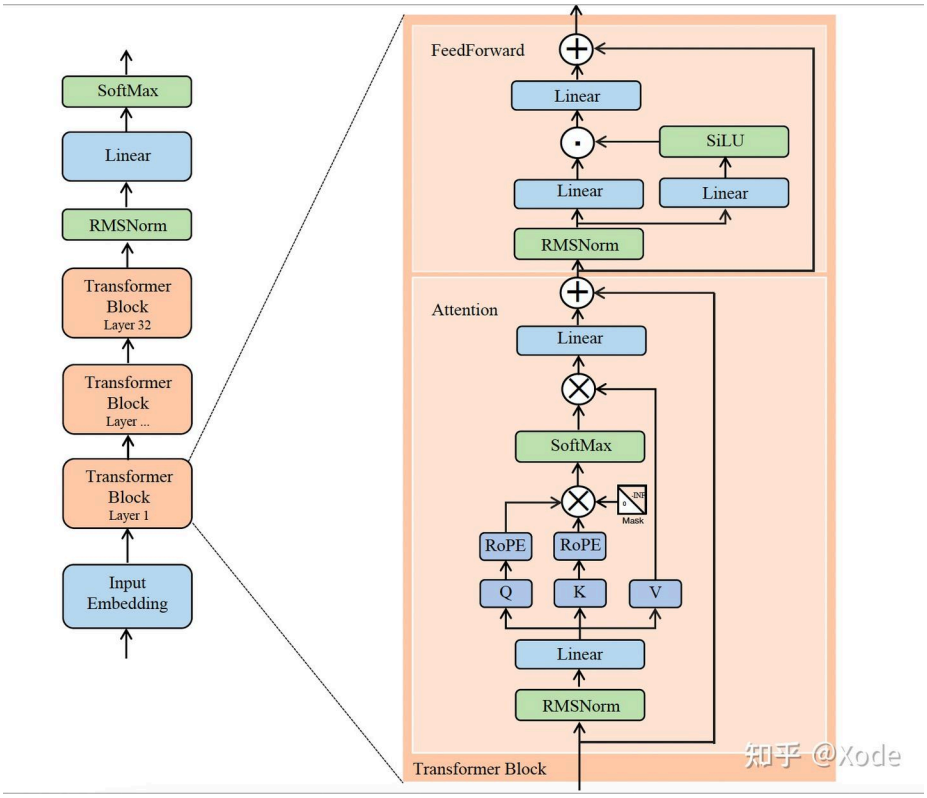
先看看传统的 Transformer Decoder 的 GPT 架构（GPT2、OPT、Bloom 等）：



Transformer 架构示意图，来源：https://arxiv.org/abs/1706.03762

这里注意力 Attention 部分的 Q、K、V 是线性层，全连接 FFN 部分也有两个线性层（一个升维和一个降维），所以它们是可以被替换的目标。

再看看现在更流行的 LLaMA 架构（Mistral、Qwen 等）：



LLaMA 架构示意图，来源：https://zhuanlan.zhihu.com/p/649756898

这里注意力的 Q、K、V 仍然是线性层，多了一个加和后的 O，它也是线性层。LLaMA 类架构中 FFN 被换成了 GLU，例如 SwiGLU、GeGLU 等，它们有三个线性层，例如图中的 up、gate、down。上面就是全部可替换的线性层。

除此以外，其实作为因果模型，最后一般都有一个形状为 `(vocab_size, hidden_size)` 的因果头，一般叫 `lm_head`，它的作用就是把最后输出的 hidden_state 映射为词表范围，用于输出。它也是线性层，但我们一般不会去对它做 LoRA 替换。huggingface 的 Peft 库中的实现也是默认不考虑它的。

---

2. 代码实践

简单温习完原理后，我们就来上手试试。

注意：下面我们会对所有线性层做 LoRA 替换，这是遵循 **QLoRA 的风格**，会增加可训练参数、增强表征能力。

2.1 定义 LoRA 线性层

我们先来创建用于替换的 LoRA 核心部分，我们用一个类 `LoraLinear` 来包装：

```
class LoraLinear(nn.Module):
    def __init__(
        self,
        base_layer: nn.Linear,      # 原来的线性层
        r: int = 8,                 # lora rank
        alpha: int = 16,            # lora alpha
        dropout_p: float = 0.0,     # lora dropout
        test_mode: bool = False,    # 测试模式，用于控制 lora_B 是否为全零
    ):
        super(LoraLinear, self).__init__()
        self.base_layer = copy.deepcopy(base_layer)
        self.r = r
```

```
        self.alpha = alpha
        self.dropout = nn.Dropout(dropout_p)

        # 定义 lora_A 和 lora_B 为 Parameter
        self.lora_A = nn.Parameter(torch.empty((r, base_layer.in_features), dty
        self.lora_B = nn.Parameter(torch.empty((base_layer.out_features, r), dt

        # 初始化 lora 矩阵
        nn.init.normal_(self.lora_A, mean=0.0, std=0.02)
        if test_mode:
            nn.init.normal_(self.lora_B, mean=0.0, std=0.02)
        else:
            nn.init.zeros_(self.lora_B)

        # 冻结原来的层的参数
        for param in self.base_layer.parameters():
            param.requires_grad = False

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        scaling = float(self.alpha) / float(self.r)     # lora 缩放系数
        lora_adjustment = F.linear(self.dropout(x), self.lora_A)
        lora_adjustment = F.linear(lora_adjustment, self.lora_B)
        return self.base_layer(x) + lora_adjustment * scaling
```

这里的 alpha 和 r 都遵循原论文设计，r 就是近似的低秩矩阵的 rank，alpha 是用于缩放的一个因子。这里多了个 `test_mode` 字段，它会用在我们后面的验证环节，并不是主要实现，不用过于关注。

中间核心部分：我们主要关注中间的核心部分，这里我用 `nn.Parameter` 来定义了 A 和 B 矩阵，当然也可以用 `nn.Linear` ，没关系。重要的是 A 和 B 的形状不要弄错，A 是 `(in_features, r)` 、B 是 `(r, out_features)` ，所以代码实现里就要反过来。这里还加了个 dropout，默认为 0。

初始化：原论文对 A 进行高斯初始化、对 B 进行零初始化，这样的作用是为了让 BA 的初始前向结果为零，从而保证一开始的结果和原矩阵结果一致。原论文其实也讨论了 A 零初始化、B 高斯初始化，但认为没什么区别。现在已经有了很多新的初始化方法，我们只是学习型复现，所以先不考虑太多，只用最简单的初始化方法。

这里的 `test_mode` 就是控制 B 是否为全零初始化，主要用于后面的验证部分，正式实现中不会用到的。

冻结参数：最后，特别注意要把原矩阵冻结。

前向：前向的时候，原论文提到了一个缩放因子 $\frac{\alpha}{r}$，所以我们实现一下。然后就是非常简单的 Wx + BAx，如果前面有 dropout 的话注意带上 dropout。

2.2 替换 LoRA 层

实现了 `LoraLinear` 后，我们就需要替换目标线性层了。就像前面说的，我们这里会找到所有线性层（除了 `lm_head` ）并替换，当然也可以只替换 Attention 部分或者 FFN/GLU 部分，可以自定义尝试。

我们根据名称来找线性层：

```
import torch.nn as nn

def replace_linear_with_lora(
    module: nn.Module,
    r: int = 8,
    alpha: int = 16,
    dropout_p: float = 0.0,
    embed_requires_grad: bool = False,      # embedding 层是否训练
```

```
    norm_requires_grad: bool = False,       # norm 层是否训练
    head_requires_grad: bool = False,       # lm_head 层是否训练（Causal LM才有）
    test_mode: bool = False,                # 测试模式，用于控制 lora_B 是否为全零
):
    """
    找到 module 中所有线性层并递归替换
    """
    for name, child in module.named_children():
        # 先处理额外的层，lm_head 也是 linear，所以先处理
        if any(s in name for s in ['embed', 'norm', 'lm_head']):
            requires_grad = embed_requires_grad if 'embed' in name \
                            else norm_requires_grad if 'norm' in name \
                            else head_requires_grad
            for param in child.parameters():
                param.requires_grad = requires_grad
        # 替换所有线性层，QLoRA 做法
        elif isinstance(child, nn.Linear):
            lora_linear = LoraLinear(child, r=r, alpha=alpha, dropout_p=dropout
            setattr(module, name, lora_linear)
        # 递归向下替换
        else:
            replace_linear_with_lora(
                child, r, alpha, dropout_p,
                embed_requires_grad, norm_requires_grad, head_requires_grad,
                test_mode=test_mode
            )
```

递归查找： `nn.Module` 的 `named_children()` 会返回名称和**下一级**的子模块。注意，这不会递归返回全部子模块，`named_modules()` 才是递归返回所有的子模块。所以我们需要递归向下查找替换。（注意：这里只考虑了语言模型，如果涉及到卷积、池化等其它类型的参数模块，需要在递归的时候默认把当前参数冻结，遇到 LoRA 替换才开启。）

> 为什么不用 `named_modules()`？
> 因为我们不仅仅是遍历，还需要替换参数，如果根据 `named_modules()` 来遍历，就会出现经典的**迭代中变更元素**的错误，所以我们手动向下 DFS。

替换：我们遇到 `nn.Linear` 就替换，在替换的时候用 python 的 `setattr` 方法。

额外层：其实这对应 huggingface peft 库中的 `modules_to_save` 字段，就是额外保存训练的参数，我们这里默认都不动，所以遍历到的时候都设为不需要梯度。

2.3 LoRA 尝试

我们的 LoRA 主体部分已经实现完了！非常简单！

现在，我们来尝试一下使用，我们先实现一个用于检查的函数：

```
def print_trainable_parameters(model: nn.Module):
    """
    打印可训练参数，表现和 PeftModel 的 print_trainable_parameters 方法类似
    """
    total_params = sum(p.numel() for p in model.parameters())
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_
    trainable_percentage = 100 * trainable_params / total_params

    # 返回可训练参数量、所有参数量、可训练参数量占比（百分比）
    print(f"trainable params: {trainable_params:,} || all params: {total_params
```

接下来，我们创建一个小模型，还是以 LLaMA 为例：

```python
from transformers import AutoConfig

config = AutoConfig.for_model('llama')
config.hidden_size = 24
config.intermediate_size = config.hidden_size * 4
config.num_attention_heads = 4
config.num_hidden_layers = 4
config.num_key_value_heads = 2
config.vocab_size = 128
```

我们的模型设置得非常小，因为只是尝试验证。接下来我们把它实例化：

```python
from transformers import AutoModel, AutoModelForCausalLM

raw_model = AutoModel.from_config(config)  # 没带因果头
# raw_model = AutoModelForCausalLM.from_config(config)  # 带了因果头
print(raw_model)

"""
LlamaModel(
  (embed_tokens): Embedding(128, 24)
  (layers): ModuleList(
    (0-3): 4 x LlamaDecoderLayer(
      (self_attn): LlamaSdpaAttention(
        (q_proj): Linear(in_features=24, out_features=24, bias=False)
        (k_proj): Linear(in_features=24, out_features=12, bias=False)
        (v_proj): Linear(in_features=24, out_features=12, bias=False)
        (o_proj): Linear(in_features=24, out_features=24, bias=False)
        (rotary_emb): LlamaRotaryEmbedding()
      )
      (mlp): LlamaMLP(
        (gate_proj): Linear(in_features=24, out_features=96, bias=False)
        (up_proj): Linear(in_features=24, out_features=96, bias=False)
        (down_proj): Linear(in_features=96, out_features=24, bias=False)
        (act_fn): SiLU()
      )
      (input_layernorm): LlamaRMSNorm()
      (post_attention_layernorm): LlamaRMSNorm()
    )
  )
  (norm): LlamaRMSNorm()
)
"""
```

可以看到，模型已经实例化了，中间的 `q_proj` 、 `k_proj` 等线性层就是我们的替换目标。

我们先看看模型现在的参数情况：

```python
print_trainable_parameters(raw_model)

"""
trainable params: 37,848 || all params: 37,848 || trainable%: 100.0000
"""
```

模型小小的，只有几万参数，并且现在因为什么操作都没做，所以默认全部参数都可以训练。

接下来，我们替换目标线性层：

```python
import copy

lora_model = copy.deepcopy(raw_model)  # 深克隆，独立一个新模型
replace_linear_with_lora(lora_model, r=8, alpha=16)  # 替换
print_trainable_parameters(lora_model) # 打印参数情况
print(lora_model)
```

```
"""
trainable params: 16,896 || all params: 54,744 || trainable%: 30.8637

LlamaModel(
  (embed_tokens): Embedding(128, 24)
  (layers): ModuleList(
    (0-3): 4 x LlamaDecoderLayer(
      (self_attn): LlamaAttention(
        (q_proj): LoraLinear(
          (base_layer): Linear(in_features=24, out_features=24, bias=False)
          (dropout): Dropout(p=0.0, inplace=False)
        )
        (k_proj): LoraLinear(
          (base_layer): Linear(in_features=24, out_features=12, bias=False)
          (dropout): Dropout(p=0.0, inplace=False)
        )
        (v_proj): LoraLinear(
          (base_layer): Linear(in_features=24, out_features=12, bias=False)
          (dropout): Dropout(p=0.0, inplace=False)
        )
        (o_proj): LoraLinear(
          (base_layer): Linear(in_features=24, out_features=24, bias=False)
          (dropout): Dropout(p=0.0, inplace=False)
        )
        (rotary_emb): LlamaRotaryEmbedding()
      )
      (mlp): LlamaMLP(
        (gate_proj): LoraLinear(
          (base_layer): Linear(in_features=24, out_features=96, bias=False)
          (dropout): Dropout(p=0.0, inplace=False)
        )
        (up_proj): LoraLinear(
          (base_layer): Linear(in_features=24, out_features=96, bias=False)
          (dropout): Dropout(p=0.0, inplace=False)
        )
        (down_proj): LoraLinear(
          (base_layer): Linear(in_features=96, out_features=24, bias=False)
          (dropout): Dropout(p=0.0, inplace=False)
        )
        (act_fn): SiLU()
      )
      (input_layernorm): LlamaRMSNorm()
      (post_attention_layernorm): LlamaRMSNorm()
    )
  )
  (norm): LlamaRMSNorm()
)
"""
```

看上去成功了！由于我们的模型本身就小，所以可训练参数占比达到了 30%+，实际上在大模型中 LoRA 的参数占比非常小，一般在 0.1% 量级。

2.4 验证 LoRA

我们的 LoRA 替换看上去成功了，可要怎么知道我们的替换是真的有效和符合预期的呢？下面给出三种方法，其中第三种涉及到了 **LoRA 的卸载与重载**。

（1）检查是不是只有 LoRA 层是可训练的

```
def print_model_parameters(model):
    """
    查看模型参数的 requires_grad 情况
    """
    print("Layer Name & Parameters")
    print("----------------------------")
```

```
    for name, parameter in model.named_parameters():
        print(f"{name:50} | Requires_grad: {parameter.requires_grad}")
```

这里会打印出所有参数及其 `requires_grad` 的值，可以人工检查一下。

（2）和 hugging face 的 peft 对比

```
from peft import LoraConfig, get_peft_model

lora_config = LoraConfig(
    r=8,
    lora_alpha=16,
    target_modules='all-linear', # 太低版本的 peft 不支持这种做法
)
peft_lora_model = copy.deepcopy(raw_model)
peft_lora_model = get_peft_model(peft_lora_model, lora_config)
peft_lora_model.print_trainable_parameters()

"""
trainable params: 16,896 || all params: 54,744 || trainable%: 30.8637
"""
```

可以看到，我们的 lora_model 参数情况和 hugging face 的 peft 表现一致。

（3）卸载 LoRA 和重载 LoRA

除了参数，我们还要验证我们的前向是否正确。由于 LoRA 实现特别简单，所以其额外参数部分也很容易卸载。考虑到这个，我们很直接地想到下面四种前向情况：

- 原始模型
- LoRA 适配后的模型
- 卸载 LoRA 后的模型
- 重载 LoRA 后的模型

如果实现正确，情况一和三的前向结果应该是一致的，情况二和四的前向结果应该是一致的。如果考虑到 BA 不做零初始化，那么这两类前向结果各自应该是不一样的。

因此，我们前面的 `test_mode` 字段派上用场，我们对 A 和 B 都做高斯初始化，让 BA 非零，改变模型的前向结果。下面来测试，先写好 `unload_lora` 和 `load_lora` 。

- `unload_lora`：

```
from typing import List

def unload_lora(module: nn.Module, adapter_name: str = 'adapter'):
    """
    卸载 lora 参数，并将原模型恢复至加载 lora 前的样子
    """
    lora_parameters = {}
    def search_lora_linear(module: nn.Module, prefix: List[str]):
        for name, child in module.named_children():
            new_prefix = prefix + [name]
            if isinstance(child, LoraLinear):
                # 保存 lora 参数
                lora_parameters['.'.join(new_prefix)] = {
                    "lora_A_weight": child.lora_A.data.cpu(),
                    "lora_B_weight": child.lora_B.data.cpu(),
                    "r": child.r,
                    "alpha": child.alpha,
                    "dropout_p": child.dropout.p,
                }
                setattr(module, name, child.base_layer)
            else:
                search_lora_linear(child, new_prefix)
```

```
    search_lora_linear(module, [])
    # 解冻原模型
    for name, param in module.named_parameters():
        param.requires_grad = True

    torch.save(lora_parameters, f"{adapter_name}.pt")
```

- load_lora：

```
def load_lora(module: nn.Module, adapter_name: str = 'adapter'):
    """
    加载 lora 参数
    """
    lora_parameters = torch.load(f"{adapter_name}.pt")

    for name, lora_params in lora_parameters.items():
        child = dict(module.named_modules())[name]
        if isinstance(child, nn.Linear):
            lora_linear = LoraLinear(child, lora_params['r'], lora_params['alph
            lora_linear.lora_A.data = lora_params["lora_A_weight"].to(lora_line
            lora_linear.lora_B.data = lora_params["lora_B_weight"].to(lora_line

            # 名称示例：layers.0.self_attn.q_proj
            # 根据名称循环找到所需 module
            parts = name.split(".")
            obj = module
            for part in parts[:-1]:  # 不包括最后一级
                obj = getattr(obj, part)
            setattr(obj, parts[-1], lora_linear)

    # 恢复原来的冻结方式，这里简单地除了 lora 全冻结
    for name, param in module.named_parameters():
        if any(s in name for s in ['embed', 'norm', 'lm_head']):
            param.requires_grad = False
```

下面创建一个测试张量：

```
# 创建一个测试 tensor
bsz = 2
seq_len = 8
test_tensor = torch.randint(0, config.vocab_size, (bsz, seq_len))
```

然后做 LoRA 替换：

```
# 开测试模式，让 BA 非零
lora_model = copy.deepcopy(raw_model)
replace_linear_with_lora(lora_model, r=8, alpha=16, test_mode=True)
```

再做四次前向：

```
# 原模型的前向结果
raw_model.eval()
print_trainable_parameters(raw_model)   # 检查参数和可训练情况
raw_res = raw_model(test_tensor).last_hidden_state
"""
trainable params: 37,848 || all params: 37,848 || trainable%: 100.0000
"""

# 第一次直接初始化 lora 的前向结果
lora_model.eval()
print_trainable_parameters(lora_model)  # 检查参数和可训练情况
```

```
before_unload_res = lora_model(test_tensor).last_hidden_state
"""
trainable params: 16,896 || all params: 54,744 || trainable%: 30.8637
"""

# 卸载 lora 后的前向结果
unload_lora(lora_model)
lora_model.eval()
print_trainable_parameters(lora_model)  # 检查参数和可训练情况
unload_res = lora_model(test_tensor).last_hidden_state
"""
trainable params: 37,848 || all params: 37,848 || trainable%: 100.0000
"""

# 重新装载 lora 后的前向结果
load_lora(lora_model)
lora_model.eval()
print_trainable_parameters(lora_model)  # 检查参数和可训练情况
load_res = lora_model(test_tensor).last_hidden_state
"""
trainable params: 16,896 || all params: 54,744 || trainable%: 30.8637
"""
```

如果实现正确，这时候文件夹里应该有 `adapter.pt` ，并且一、三的参数可训练情况一致，二、四的参数可训练情况一致。

最后检查一下前向结果：

```
print(torch.allclose(raw_res, unload_res, atol=1e-6))           # 应为 True
print(torch.allclose(before_unload_res, load_res, atol=1e-6))   # 应为 True
print(torch.allclose(raw_res, load_res, atol=1e-6))             # 应为 False
"""
True
True
False
"""
```

非常好！经过验证，我们的 LoRA 实现无误！

---

3. LoRA 实战

虽然验证上我们的 LoRA 实现没有问题，但我们还是需要实际微调模型，来试试看是不是真的准确实现了。我们这里用的模型是 hugging face transformers 的模型，但**不用 hugging face 的 Trainer**，而是**用 torch 来写训练循环**，因此最基本的 torch 模型也是可以支持的。

下面导入需要的库：

```
import json
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F

from tqdm import tqdm
from typing import List
from einops import rearrange
from datasets import load_dataset
from torch.utils.data import Dataset, DataLoader
from transformers import AutoConfig, AutoTokenizer, AutoModel, AutoModelForCaus
```

```
device = 'cuda' if torch.cuda.is_available() else 'cpu'
dtype = torch.bfloat16 if device != 'cpu' and torch.cuda.is_bf16_supported() el
print(f'device: {device}\ndtype: {dtype}')
```

3.1 模型

我们选用一个小模型，Qwen1.5-0.5B：**Qwen1.5-0.5B**。

选这个而不选新出的 Qwen2-0.5B 是因为怕表现太强了不用微调也很好，所以选一个弱一点的。

这里的模型和任务都非常简单，只是用于学习验证，当然也可以换成任何想尝试的其他模型。我们先加载原始模型：

```
# 模型路径可以改成本地
model_name_or_path = 'Qwen/Qwen1.5-0.5B'

# 加载原始模型
tokenizer = AutoTokenizer.from_pretrained(model_name_or_path)
tokenizer.pad_token_id = tokenizer.eos_token_id
tokenizer.padding_side = 'left'

model = AutoModelForCausalLM.from_pretrained(model_name_or_path, torch_dtype=dt
```

然后我们获取自己写好的 LoRA 模型：

```
# 获取 lora model
replace_linear_with_lora(model, r=16, alpha=32, dropout_p=0.0)
model.to(device)

# 查看可训练参数
print_trainable_parameters(model)
"""
trainable params: 3,784,704 || all params: 467,772,416 || trainable%: 0.8091
"""
```

这里可以看到，可训练参数占比只有 0.8% 左右，非常小。

3.2 数据集

为了避免数据污染，我们选一个新一点的数据集。这里选了一个社区开源的 **BioInstruct**，是关于生物医学的问答数据集，格式是经典的 alpaca，即单轮问答 Instruction、Input 和 Output 的形式。

这里我们就不用 hugging face 那一套流程了（但还是借用了它的 Dataset），而是用 torch。

```
# 数据路径可以改成本地
data_name_or_path = 'bio-nlp-umass/bioinstruct'

# 定义训练数据集
class SFTDataset(Dataset):
    def __init__(self,
        tokenizer: AutoTokenizer,
        data_path: str,
        load_local: bool = False,
        max_len: int = 256,
        split_len: str = '1%',
    ):
        super().__init__()
        self.tokenizer = tokenizer
```

```
        if load_local:
            ds = load_dataset('json', data_dir=data_path, split=f'train[:{split
        else:
            ds = load_dataset(data_path, split=f'train[:{split_len}]')
        self.max_len = max_len

        def process_func(example):
            # 提取 instruction 和 input
            instruction = example['instruction'].strip()
            input = example['input'].strip()
            output = example['output'].strip()

            # 构造模板
            instruction_msg = [
                {"role": "user", "content": (instruction + f"\n{input}") if len
            ]
            tokenized_instruction = tokenizer.apply_chat_template(instruction_m
            tokenized_output = tokenizer(output + "<|im_end|>" + f"{tokenizer.e

            # 截断，最大不超过 max_len
            tokenized_prompt = (tokenized_instruction + tokenized_output)[:self

            # 构造 input_ids, attention_mask, labels
            input_ids = tokenized_prompt[:-1]
            padding_mask = ([0] * len(tokenized_instruction) + [1] * (len(token
            labels = tokenized_prompt[1:]

            return {
                'input_ids': input_ids,
                'attention_mask': padding_mask,
                'labels': labels,
            }

        self.ds = ds.map(
            process_func,
            batched=False,
            remove_columns=ds.column_names,
            desc='Processing dataset',
        )

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

    def __getitem__(self, index: int):
        return self.ds[index]
```

（1）数据组织格式

这里的构造的数据是按照 ChatML 格式构造的，即：

```
<|im_start|>system
You are a helpful assistant<|im_end|>
<|im_start|>user
{指令}<|im_end|>
<|im_start|>assistant
{回复}<|im_end|>
```

由于 Qwen 本身用的也是 ChatML 格式，所以 tokenizer 已经实现了 `apply_chat_template` 方法，可以直接用。

（2）Loss 计算

要注意：我们的数据集是单轮问答 QA 形式，不同于预训练对所有位置都计算 loss，在做 SFT 的时候，我们只计算 Answer 部分的 loss，Question 部分我们会 mask 掉。

> 当然，对于小模型的 SFT，也有做法是和预训练一样，全部位置都计算 loss，我们这里只是用最经典的方式，不关注太多其他技巧。

（3）加载

下面加载验证一下。

我这里只取了很小的一部分（1%），实际上这个数据集本身就特别小，所以全部加载也没问题。

```
ds = SFTDataset(tokenizer, data_name_or_path, load_local=False, split_len="1%")

print(len(ds[0]['input_ids']))
print(len(ds[0]['attention_mask']))
print(len(ds[0]['labels']))

print(tokenizer.decode(ds[0]['input_ids']))
print(ds[0]['attention_mask'])
print(tokenizer.decode(ds[0]['labels']))

"""
79
79
79

<|im_start|>system
You are a helpful assistant<|im_end|>
<|im_start|>user
Identify the main conclusion from the provided medical report excerpt.
The patient's blood test results showed an elevation in liver enzymes, specific
<|im_start|>assistant
The patient has signs of liver damage and a fatty liver.<|im_end|><|endoftext|>

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

system
You are a helpful assistant<|im_end|>
<|im_start|>user
Identify the main conclusion from the provided medical report excerpt.
The patient's blood test results showed an elevation in liver enzymes, specific
<|im_start|>assistant
The patient has signs of liver damage and a fatty liver.<|im_end|><|endoftext|>
"""
```

可以看到，数据是符合预期的。其中 output 部分我多加了一个 `<|endoftext|>`，即 Qwen 的 eos_token。

（4）`collate_fn`

接下来写 `collate_fn`，用于处理 batch。

这里由于截断已经在 Dataset 里做好了，所以 `collate_fn` 里主要是做填充。

```
def collate_fn(batch: List, tokenizer):
    max_len = max(len(item['input_ids']) for item in batch)

    input_ids = []
    attention_mask = []
    labels = []

    for item in batch:
        input_id = item['input_ids']
        attention_mask_item = item['attention_mask']
        label = item['labels']
```

```
        # 计算填充长度
        pad_len = max_len - len(input_id)

        # 左填充
        input_ids.append([tokenizer.eos_token_id] * pad_len + input_id)
        attention_mask.append([0] * pad_len + attention_mask_item)
        labels.append([tokenizer.eos_token_id] * pad_len + label)

    # 将 list 转换为 tensor
    input_ids = torch.LongTensor(input_ids)
    attention_mask = torch.LongTensor(attention_mask)
    labels = torch.LongTensor(labels)

    return {
        'input_ids': input_ids,
        'attention_mask': attention_mask,
        'labels': labels,
    }
```

要注意这里是左填充，具体原因在**第一篇文章**里也有解释：

> 另外注意这里 `padding_side='left'` ，如果不是的话需要设置 `tokenizer.padding_side='left'` ，即批量填充的时候从左边开始填充，这对于 decoder-only 的模型做生成任务是必要的，因为我们本质上做的是 next token prediction，如果 pad 挡在了生成序列的右边，会影响到模型生成。
>
> # 假设 pad_token 就是 eos_token(</s>)
> # 从右边填充 Once upon a time </s></s></s></s>...
> # 从左边填充 </s></s></s></s>Once upon a time ...

（5）dataloader

下面我们加载一下 dataloader：

```
bsz = 8

dataloader = DataLoader(ds, batch_size=bsz, shuffle=True, collate_fn=lambda bat

for batch in dataloader:
    print(batch)
    break

# 这里的三个张量形状应该一致，都为 [bsz, seq_len]
"""
{
  "input_ids": ...,
  "attention_mask": ...,
  "labels": ...
}
"""
```

由于我们这次主要是学习 LoRA，因此数据处理是非常简单地截断填充，没有做一些排序、拼接等操作。

3.3 训练

（1）超参数设置

设置超参数和优化器如下，我们就不做学习率调度了，直接保持到底：

```
lr = 5e-4       # 学习率，不要太大
num_epochs = 3    # 训练轮数
```

```
logging_steps = 5   # 每隔多少步输出
max_grad_norm = 1.0  # 最大梯度范数，用于剪裁

optimizer = optim.AdamW(model.parameters(), lr=lr) # AdamW 优化器，常用
```

(2) Training Loop

```
model.train()

total_loss = 0
total_step = 0
for epoch in range(num_epochs):
    for step, batch in enumerate(tqdm(dataloader, desc=f"Epoch {epoch+1}/{num_e
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        optimizer.zero_grad()

        # 重整 logits, mask, labels
        outputs = model(input_ids, attention_mask=attention_mask, labels=labels
        logits = outputs.logits
        rearranged_logits = rearrange(logits, 'bsz seq_len vocab_size -> (bsz s
        rearranged_attention_mask = rearrange(attention_mask, 'bsz seq_len -> (
        rearranged_labels = rearrange(labels, 'bsz seq_len -> (bsz seq_len)')

        # 按照 mask 手动计算 loss
        sum_loss = F.cross_entropy(rearranged_logits, rearranged_labels, ignore
        loss = torch.sum(sum_loss * rearranged_attention_mask) / torch.sum(rear
        loss.backward()

        # 计算梯度范数并裁剪
        total_norm = nn.utils.clip_grad_norm_(model.parameters(), max_grad_norm

        optimizer.step()
        total_loss += loss.item()

        total_step += 1
        if total_step % logging_steps == 0:
            avg_loss = total_loss / total_step
            print(f"Step: {step+1}/{len(dataloader)}, Loss: {avg_loss:.4f}, Gra


    # 打印每个 epoch 结束的累计损失
    print(f"Epoch {epoch+1} finished, Average Loss: {total_loss / total_step:.4
```

我们看看训练结果：

```
Epoch 1/3:  12%|█         | 4/32 [00:08<00:56,  2.03s/it]
Step: 5/32, Loss: 2.0748, Grad Norm: 1.9531
Epoch 1/3:  28%|███       | 9/32 [00:16<00:35,  1.56s/it]
Step: 10/32, Loss: 1.9878, Grad Norm: 1.7812
Epoch 1/3:  44%|████▍     | 14/32 [00:24<00:31,  1.75s/it]
Step: 15/32, Loss: 1.9209, Grad Norm: 1.7109
Epoch 1/3:  59%|█████▉    | 19/32 [00:33<00:23,  1.78s/it]
Step: 20/32, Loss: 1.9113, Grad Norm: 2.2188
Epoch 1/3:  75%|███████▌  | 24/32 [00:41<00:12,  1.52s/it]
Step: 25/32, Loss: 1.8852, Grad Norm: 1.6406
Epoch 1/3:  91%|█████████ | 29/32 [00:49<00:04,  1.65s/it]
Step: 30/32, Loss: 1.8730, Grad Norm: 1.9922
Epoch 1/3: 100%|██████████| 32/32 [00:52<00:00,  1.64s/it]
Epoch 1 finished, Average Loss: 1.8794

Epoch 2/3:   6%|▋         | 2/32 [00:02<00:37,  1.25s/it]
```

```
Step: 3/32, Loss: 1.8285, Grad Norm: 1.3828
Epoch 2/3:  22%|██        | 7/32 [00:10<00:41,  1.64s/it]
Step: 8/32, Loss: 1.7702, Grad Norm: 1.5156
Epoch 2/3:  38%|███▊      | 12/32 [00:19<00:35,  1.78s/it]
Step: 13/32, Loss: 1.7095, Grad Norm: 1.5547
Epoch 2/3:  53%|█████▎    | 17/32 [00:26<00:20,  1.40s/it]
Step: 18/32, Loss: 1.6636, Grad Norm: 1.9219
Epoch 2/3:  69%|██████▉   | 22/32 [00:34<00:15,  1.57s/it]
Step: 23/32, Loss: 1.6245, Grad Norm: 1.5625
Epoch 2/3:  84%|████████▍ | 27/32 [00:42<00:08,  1.64s/it]
Step: 28/32, Loss: 1.5951, Grad Norm: 1.4766
Epoch 2/3: 100%|██████████| 32/32 [00:50<00:00,  1.57s/it]
Epoch 2 finished, Average Loss: 1.5732

Epoch 3/3:   0%|          | 0/32 [00:00<?, ?it/s]
Step: 1/32, Loss: 1.5647, Grad Norm: 1.8828
Epoch 3/3:  16%|█▌        | 5/32 [00:08<00:43,  1.61s/it]
Step: 6/32, Loss: 1.5122, Grad Norm: 2.1094
Epoch 3/3:  31%|███       | 10/32 [00:16<00:33,  1.54s/it]
Step: 11/32, Loss: 1.4668, Grad Norm: 2.2969
Epoch 3/3:  47%|████▋     | 15/32 [00:25<00:29,  1.72s/it]
Step: 16/32, Loss: 1.4207, Grad Norm: 1.5234
Epoch 3/3:  62%|██████▎   | 20/32 [00:33<00:20,  1.71s/it]
Step: 21/32, Loss: 1.3811, Grad Norm: 2.3906
Epoch 3/3:  78%|███████▊  | 25/32 [00:41<00:11,  1.60s/it]
Step: 26/32, Loss: 1.3481, Grad Norm: 2.0625
Epoch 3/3:  94%|█████████▍| 30/32 [00:48<00:03,  1.59s/it]
Step: 31/32, Loss: 1.3177, Grad Norm: 1.8594
Epoch 3/3: 100%|██████████| 32/32 [00:50<00:00,  1.59s/it]
Epoch 3 finished, Average Loss: 1.3119
```

Loss 逐步递减，最后收敛到 1.3 左右，表现不错！

（3）手动计算 Loss

这里特别提一下为什么要手动计算 loss。

我们这里用的是 hugging face transformers 格式的模型，在**第一篇文章**中提到，他们的 loss 已经实现了 input_ids 和 labels 的移位：

> 所以我们只需要把 input_ids 复制一份、再偏移一位，就可以作为 labels 了。
> ……再等等，让我们看看这条问题：**Shifting ids to the right when training GPT-2 on text generation? - Beginners - Hugging Face Forums**：
>
> 这里 sgugger 提到，在 Hugging Face 的实现里，training 时已经实现好了偏移一位的逻辑，不需要我们再手动实现了。我们也可以在 transformers 的源码里看到这一点，例如 **LLaMA 的实现**。

所以尽管 `outputs = model(input_ids, attention_mask=attention_mask, labels=labels)` 的 outputs 中有 loss，我们也不能直接拿来用。我们可以做一个小实验验证一下：

```
# 创建测试 tensor
test_text = 'Hello, world!'
test_tensor = tokenizer(test_text, return_tensors='pt').to(device)
test_tensor['input_ids'].shape
inputs = tokenizer(test_text)

# 手动移位，对应我们上面的实现
input_ids = torch.LongTensor([inputs['input_ids'][:-1]]).to(device)
attention_mask = torch.LongTensor([inputs['attention_mask'][:-1]]).to(device)
labels = torch.LongTensor([inputs['input_ids'][1:]]).to(device)

# 前向
outputs = model(
```

```
    input_ids=input_ids,
    attention_mask=attention_mask,
    labels=labels,
)
loss, logits = outputs.loss, outputs.logits
loss.item()
"""
9.069315910339355
"""
```

这里模型自带传出来的 loss 是 9 左右。我们再看看手动计算的 loss：

```
# 用 einops 重整 logits、mask、labels，和上面一样
rearranged_logits = rearrange(logits, 'b seq_len vocab_size -> (b seq_len) voca
rearranged_attention_mask = rearrange(attention_mask, 'b seq_len -> (b seq_len)
rearranged_labels = rearrange(labels, 'b seq_len -> (b seq_len)')

# 手动计算 loss
sum_loss = F.cross_entropy(rearranged_logits, rearranged_labels, ignore_index=0
print(sum_loss)
# 按 mask 计算最后的 loss，当然这里没有被 mask 的区域，相当于加和平均
torch.sum(sum_loss * rearranged_attention_mask) / torch.sum(rearranged_attentic
"""
tensor([14.0533,  1.8505, 10.2932,  3.9565], grad_fn=<NllLossBackward0>)
tensor(7.5384, grad_fn=<DivBackward0>)
"""
```

可以看到，我们手动计算的 loss 是 7.5 左右，和上面不一致！

那要怎么样才能一致呢？很简单，我们不移位、让模型内部自己移位就行了：

```
inputs = tokenizer(test_text)

# 这次不移位
input_ids = torch.LongTensor([inputs['input_ids']]).to(device)
attention_mask = torch.LongTensor([inputs['attention_mask']]).to(device)
labels = torch.LongTensor([inputs['input_ids']]).to(device)

outputs = model(
    input_ids=input_ids,
    attention_mask=attention_mask,
    labels=labels,
)
loss, logits = outputs.loss, outputs.logits
loss
"""
tensor(7.5384, grad_fn=<NllLossBackward0>)
"""
```

这次 loss 就和我们前面手动计算的结果一样了。

3.4 推理 & 保存

（1）自实现 `inference`

上面训练完以后，我们可以尝试推理一下。

注意这是 jupyter notebook 保存了每个中间变量结果，所以 model 就是训练好的 model。

下面的推理方法用了流式输出，具体实现我们会在第三篇文章中解读（挖坑：

```python
def inference(
    model,
    tokenizer,
    text: str,
    max_new_tokens: int = 160,
    do_sample: bool = True,
    temperature: float = 0.3,
    print_inputs: bool = True,
    streaming: bool = False,
):
    # 构建输入，模板要和 Dataset 中一致
    prompt_msg = [
        {"role": "user", "content": text}
    ]
    prompt = tokenizer.apply_chat_template(prompt_msg, tokenize=False, add_gene
    inputs = tokenizer(prompt, return_tensors='pt', add_special_tokens=False).t
    input_ids = inputs['input_ids']
    im_end_id = tokenizer.encode("<|im_end|>")[0]

    # 是否打印输入部分
    if print_inputs:
        print(prompt, end='')

    # 生成
    stop_words = [tokenizer.eos_token_id, im_end_id]
    generated_tokens = []

    for _ in range(max_new_tokens):
        with torch.no_grad():
            outputs = model(input_ids)

        logits = outputs.logits[:, -1, :]

        # 不同采样方式
        if do_sample:
            logits = logits / temperature
            probs = F.softmax(logits, dim=-1)
            next_token = torch.multinomial(probs, num_samples=1)
        else:
            # 贪婪解码
            next_token = torch.argmax(logits, dim=-1, keepdim=True)
        if next_token.item() in stop_words:
            break
        generated_tokens.append(next_token.item())
        # 流式输出
        if streaming:
            yield tokenizer.decode(generated_tokens)

        # 更新输入
        input_ids = torch.cat([input_ids, next_token], dim=-1)

    generated_text = tokenizer.decode(generated_tokens)
    return generated_text
```

(2) 推理

```python
model.eval()

for test_text in [
    'Describe the process of bacterial conjugation and its significance in the
    'Explain the role of insulin in the body and how insulin resistance affects
    'Provide recommendations for lifestyle changes that can help improve the ov
]:
    print('=' * 80)
    last_text = ''
```

```
    for text in inference(model, tokenizer, test_text, streaming=True):
        cur_text = text.replace(last_text, '')
        print(cur_text, end='', flush=True)
        last_text = text
    print('\n')

"""
================================================================================
<|im_start|>system
You are a helpful assistant<|im_end|>
<|im_start|>user
Describe the process of bacterial conjugation and its significance in the conte
<|im_start|>assistant
Bacterial conjugation is a process by which bacteria exchange genetic material

================================================================================
<|im_start|>system
You are a helpful assistant<|im_end|>
<|im_start|>user
Explain the role of insulin in the body and how insulin resistance affects bloc
<|im_start|>assistant
Insulin is a hormone produced by the pancreas that helps regulate blood sugar l

================================================================================
<|im_start|>system
You are a helpful assistant<|im_end|>
<|im_start|>user
Provide recommendations for lifestyle changes that can help improve the overall
<|im_start|>assistant
1. Maintain a healthy diet: Eat a balanced diet with lean protein, whole grains
2. Exercise regularly: Aim for at least 150 minutes of moderate-intensity exerc
3. Maintain a healthy weight: Focus on making healthy lifestyle choices and bei
4. Limit alcohol intake: Limit alcohol intake to no more than 150 ml per day, a
5. Manage stress: Practice stress management techniques like deep breathing

"""
```

可以看到大体上推理效果是不错的，考虑到这个数据集是新的，因此可以说我们的 LoRA 是有效的。

（3）保存

最后，我们可以用前面写好的 `unload` 来卸载和保存 LoRA 参数，下次再用 `load` 加载即可。

当然，我们也可以合并 LoRA 参数：

```
def merge_lora(module: nn.Module):
    """
    将 lora 参数合并到原来的 base_layer 中，并将 lora 层替换回原来的 nn.Linear 层
    """
    def search_lora_linear(module: nn.Module, prefix: List[str]):
        for name, child in module.named_children():
            new_prefix = prefix + [name]
            if isinstance(child, LoraLinear):
                # 合并 lora 参数到 base_layer
                with torch.no_grad():
                    lora_adjustment = torch.matmul(child.lora_B, child.lora_A)
                    child.base_layer.weight.add_(lora_adjustment)

                # 替换回原来的 base_layer
                setattr(module, name, child.base_layer)
            else:
                search_lora_linear(child, new_prefix)

    search_lora_linear(module, [])
```

```
    # 解冻原模型
    for name, param in module.named_parameters():
        param.requires_grad = True

# 合并
merge_lora(model)
```

可以上传到 hugging face：

```
from huggingface_hub import notebook_login

notebook_login()

model.push_to_hub('Qwen1.5-0.5B-LoRA-bioinstruct')
```

---

4. 结语

LoRA 的从零实现到这里就结束了。

很高兴上次练习预训练的文章发出来受到大家喜欢，这次也是分享一下自己学习和复现的过程，欢迎大家交流讨论！

编辑于 2024-06-12 01:02 · IP 属地北京

内容所属专栏

LLM from scratch
LLM相关的从零实现的笔记

订阅专栏

lora训练　大模型　PyTorch

理性发言，友善互动

12 条评论　默认　最新

Kiren Wang
好文。提一个小问题：感觉作者对 padding 的理解似乎有点小错误？tokenizer 的 padding 方向其实对单条 input 没什么影响。而你在构造 sft 样本的时候，由于不需要进行推理，inputs_ids 已经是完整长度，所以 left和 right 的 padding 对训练貌似不会有什么影响。padding 方向真正印象的应该是 batch 推理阶段吧。
10-31 · 上海　回复　喜欢

Xode 作者
是的，padding side只影响batch inference，这篇文章写的比较早了哈哈，当时还在学习阶段
10-31 · 北京　回复　喜欢

zfszfs
学长，太感谢您了，写的太棒了，泪目

07-31 · 河南　回复　喜欢

是海绵宝宝先生吗