

Homework 1: LLM 实现与微调

董博文 2300016604

本报告围绕 Tokenization、LLM Implementation 和 LoRA Fine-tuning 三个部分展开。

1. Tokenization

1.1 实现 BPE，训练 Tokenizer

1.1.1 BPE 算法概述

原始字符编码后在里面找出现频率最高的相邻字符对，把它们合并成一个新的编码符号。循环迭代进行这个合并的过程，每次都把频率最高的相邻字符对合起来，一直合到我们预设的词表大小为止，最终就得到了一个有限的词表。

1.1.2 基于 BPE 算法训练 LLM tokenizer 的流程概述

用 BPE 训练 tokenizer 的时候，先把原始的文本编码，然后反复执行 BPE 的合并操作，等训练结束，就得到了一个固定大小的词表，里面的每个 token 是一个字符或者字符组合。

1.1.3 基于 BPE 算法的 tokenizer 实现

基于 BPE 算法的 tokenizer 实现代码核心逻辑如下。

1. BPE 基于 bytes 合并，可以同时处理中英文，标点等各种符号。`vocab`初始化为包含 255 个单字节的字典，方便后续基于 bytes 的合并。

通过`get_stats`函数得出字符对的出现频数。`merge`函数合成新的字符对的编码表示。

`train`函数通过`get_stats`得出出现频数最高的字符对，通过`merge`函数把新的字符对的编码表示放入字典`merges`里面，最后合并到字典`vocab`里面。字典`vocab`就是训练成果，可以通过`vocab`进行 encode 和 decode 工作。

2. encode 的时候查看`merges`列表，如果字符对在`merges`里面有记录，就用`merges`里面的编码来 encode 字符对。

```
pair = min(states, key=lambda p: self.merges.get(p, float('inf')))
```

这里通过`min()`来查找最先被编码的字符对，因为之后编码的字符对可能依赖于之前的新编码。

3. decode 的时候通过`vocab`进行 decode，把编码转换成字节，通过 decode 用 UTF-8 解码，得出原来的文本。

综合上述要点，具体实现的代码如下：

```
class Tokenizer:
    def __init__(self, content, vocab_size=1024):
        self.vocab_size = vocab_size
        self.num_merges = vocab_size - 256
        self.ids = list(content.encode("utf-8"))
        self.merges = {}
        self.vocab = {i: bytes([i]) for i in range(256)}

    def get_stats(self, ids):
        counts = {}
        for pair in zip(ids, ids[1:]):
            counts[pair] = counts.get(pair, 0) + 1
        return counts

    def merge(self, ids, pair, idx):
        new_ids = []
        i = 0
        while i < len(ids):
            if i < len(ids) - 1 and ids[i] == pair[0] and ids[i + 1] == pair[1]:
                new_ids.append(idx)
                i += 2
            else:
                new_ids.append(ids[i])
                i += 1
        return new_ids

    def train(self):
        for i in range(self.num_merges):
            stats = self.get_stats(self.ids)
            pair = max(stats, key=stats.get)
            idx = 256 + i
            self.ids = self.merge(self.ids, pair, idx)
            self.merges[pair] = idx

        for (p0, p1), idx in self.merges.items():
            self.vocab[idx] = self.vocab[p0] + self.vocab[p1]

    def decode(self, ids):
        tokens = b"".join([self.vocab[idx] for idx in ids])
        text = tokens.decode('utf-8', errors='replace')
        return text

    def encode(self, text):
        tokens = list(text.encode('utf-8'))
        while len(tokens) >= 2:
            states = self.get_stats(tokens)
            pair = min(states, key=lambda p: self.merges.get(p, float('inf')))
            if pair not in self.merges:
                break
            idx = self.merges[pair]
            tokens = self.merge(tokens, pair, idx)
        return tokens
```

1.1.4 使用 manual.txt 训练 tokenizer

用 `hw1-code/bpe/manual.txt` 来训练 tokenizer, `vocab_size` 为 1024, 具体训练代码如下:

```
with open('manual.txt', 'r', encoding='utf-8') as f:
    content = f.read()
tokenizer = Tokenizer(content)
tokenizer.train()
```

执行完毕后, 就完成了 tokenizer 的训练。

1.1.5 观察 encode 再 decode manual.txt 的结果

用训练好的 tokenizer 来 encode 再 decode manual.txt, 代码实现如下:

```
encoded = tokenizer.encode(content)
decoded = tokenizer.decode(encoded)
```

可使用 `difflib` 库比较 decode 后的内容与原先内容的异同点:

```
import difflib
if content == decoded:
    print("完全相同")

diff = difflib.ndiff(content, decoded)
for d in diff:
    if d[0] != ' ':
        print(d)
```

经过比较可知, encode 再 decode 的内容与原始内容完全一致。

1.1.6 GPT-2 tokenizer 研究

使用 GPT-2 的 tokenizer 和上面训练好的 tokenizer 分别 encode 句子一、二, 比较二者输出可知, 其在长度和具体 token 上均有所不同。

1. 长度的不同

对于中文文本, GPT-2 的 tokenizer encode 后的长度为 248, 训练好的 tokenizer encode 之后的长度为 158, GPT-2 的 tokenizer encode 后的长度更长。相反, 对于英文文本, GPT-2 的 tokenizer encode 后的长度为 239, 训练好的 tokenizer encode 之后的长度为 645, 上面训练好的 tokenizer encode 之后的长度更长。

究其原因, 中文文本是从 `manual.txt` 里面抽出来的, 英文文本是对 `manual.txt` 内容的转译。

`manual.txt` 里几乎没有英文文本, 所以合并的英文字符组很少, 基于此训练出来的 tokenizer encode 英文时得出的结果长度更长。GPT-2 的词表为 50257, 支持更细粒度的英语词汇压缩, 最终展现出的

encode 结果更短。而 GPT-2 的 tokenizer 训练时使用的中文文本较少，合并的中文字符组少，encode 中文时得出的结果长度更长。

2. 具体 token 的不同 由于 BPE 在训练时所采用的语料不同，不同语料中字符的组合频率和模式存在差异，因此在进行合并操作时，所得到的字符组合或者说子词单元也会有所不同。这种差异会导致最终生成的词汇表在构成上有明显的区别，影响分词效果，造成具体 token 的不同

1.2 回答问题

1.2.1 Python 中字符的 Unicode

在 Python 中，可以使用 `ord()` 函数查看字符的 Unicode，使用 `chr()` 函数将 Unicode 转换成字符。例如：

```
print(ord("北")) # 输出: 21271
print(ord("大")) # 输出: 22823
print(chr(22823)) # 输出: 大
print(chr(27169)) # 输出: 模
print(chr(22411)) # 输出: 型
```

1.2.2 Tokenizer 的 vocab size 大小的优缺点

1. 大 vocab size:

- 优点：可以更高效地编码常见的词汇，减少 token 数量。
- 缺点：需要更多的内存和存储空间，训练和推理速度可能变慢。

2. 小 vocab size:

- 优点：模型更轻量化，适合资源受限的环境。
- 缺点：编码时可能需要更多的 token，导致上下文窗口更快耗尽。

1.2.3 LLM 不能处理非常简单的字符串操作任务（如反转字符串）的原因

其核心是 tokenizer 的机制原因，LLM 看到的并不是初始的单个字符，而是合并的字符串。而反转字符串是单个字符级别的任务，所以 LLM 不能处理。当把字符串拆解为一个一个字符处理时，LLM 就可以成功执行反转字符串的任务了。

1.2.4 LLM 在非英语语言（例如日语）上表现较差的原因

主要原因有两个，其一是语言模型在训练时见到的其他语言的数据较少，其二是 LLM 的 tokenizer 没有根据其他特定的语言进行适配的训练。

1.2.5 LLM 在简单算术问题上表现不好的原因

主要原因是 tokenizer 可能把几个数字合并成一个表示，LLM 见到的并不是原始的数字，简单算术问题是基于每个数字进行运算的，LLM 处理不好。此外，还有训练数据，模型能力等等多方面原因。

1.2.6 GPT-2 在编写 Python 代码时遇到比预期更多困难的原因

tokenizer 对空格的处理效率很低，大大减少了模型可以处理的上下文的长度。同样的，也有模型架构，数据集，模型能力等多方面的原因。

1.2.7 LLM 遇到字符串 `<|endoftext|>` 时会突然中断的原因

`<|endoftext|>` 是 GPT 模型的特殊标记，表示文本结束。模型遇到该标记时会自动停止生成。

1.2.8 LLM 在处理字符串 “SolidGoldMagikarp” 时崩溃的原因

该字符串或许是在 tokenizer 训练过程中经常出现的词，或许由于数据选取的问题导致训练数据集中出现了大量的“SolidGoldMagikarp”，从而为该字符串分配了一个专门的编码。但该字符串可能在模型的训练数据中非常罕见或不存在，该标记从来未被激活，模型没有学到如何对这个字符串进行后续的处理，导致模型输出混乱且随机，无法生成合理的上下文。

1.2.9 使用 LLM 时更倾向于使用 YAML 而不是 JSON 的原因

JSON 的 tokens 表示方面较密集，而 YAML 效率更高，YAML 经过 tokenizer 编码后的 token 数往往更少，使用的资源更少，效率更高。

1.2.10 LLM 实际上不是端到端语言建模的原因

这一问题有三个核心原因。

- 1. LLM 依赖大量预处理和后处理步骤（如分词、去噪等）。
- 2. LLM 在建模之前先把连续的字符流分开成离散 token，这一步骤由人类设计并固定，而不是由模型共同学习。因此，真正被训练和推断的对象是 token 序列，而非原始文本本身。
- 3. 它们的生成过程通常需要额外的控制逻辑（如温度调节、采样策略）。

2. LLM Implementation

2.1 git commit 记录

Commits on Jun 16, 2025

[2300016604] section-4 commit

jennie-dong committed 17 minutes ago

a278caf

<>

Commits on May 24, 2025

[2300016604] section-3 commit

jennie-dong committed 3 weeks ago

09101e1

<>

[2300016604]section-2 commit

jennie-dong committed 3 weeks ago

384685f

<>

make it fast

jennie-dong committed 3 weeks ago

0c26895

<>

Commits on May 4, 2025

[2300016604] section-1 commit

jennie-dong committed on May 4

a7a0031

<>

Initial commit

jennie-dong authored on May 4

Verifieda0ad521

<>

2.2 笔记

2.2.1 Section 1 GPT-2 整体架构

本节说明了 GPT-2 的整体模块组成结构，介绍了关键层级及其工作流程。

In_f 层的添加

先是有编码层，位置层和最后的线性映射。编码层通常指的是 token embedding，将每个 token 映射为向量；位置层即 positional embedding，用来加入位置信息。两者直接相加后作为 transformer 的初始输入。然后又新加了一个 In_f 层，这是一种最终输出的 LayerNorm，对整个模型的输出做归一化，使得下游任务更容易收敛。

之后是中间的隐藏 block。---规范化，然后再经过 attention 层，然后再规范化，之后输入 mlp 层。这是 transformer block 的标准流程：先 LayerNorm，然后进入 multi-head self-attention 计算，attention 后再 LayerNorm，再进入 mlp 层。整个结构加上残差连接，形成了深层信息提取的闭环。

(attn 层具体是怎么组织的) 每个 attention 层会对输入向量分别做线性变换生成 QKV (query, key, value)，计算注意力分数，对 value 加权求和。多头机制是将向量分成多个子空间，分别做注意力，再拼接起来增强表达力。

mlp 层是线性层 + gelu + 线性层 (gelu 总是不会让局部梯度为 0，更为灵活)。这里的 gelu 激活函数相比 relu 更平滑，可以避免部分神经元永久失活的问题，提升模型泛化能力。

细节：采样，循环采样。设置设备。计算 logits，计算 loss。optimize loop 数据循环链。采样时使用前一个 token 的输出作为下一个 token 的输入，形成循环采样；训练时则是并行计算 logits (预测分布)，用真实标签计算 cross entropy loss。设备如 CUDA 会在训练前设置好，整个 optimize loop 会不断读取数据，反向传播，再优化更新模型参数。

有很多个 attention 块，每个块之内有很多注意力层。每个注意力层关注每个 token 不同的向量段。也就是说，在多头机制下，每个 head 聚焦 token 向量的不同子空间，相当于“多角度”理解当前 token 与上下文的关系。

2.2.2 Section 2 加快训练速度

本节聚焦于提升训练效率的工程技巧，包括计算优化和框架支持等内容。

不同精度的计算所需时间比较。tensorcore。gradient scaler 用什么来加快模型速度，释放内存。使用 fp16 混合精度训练可以让显存占用更小，tensorcore 通过优化矩阵计算极大加快速度。为了避免 fp16 下数值精度不稳定，gradient scaler 会动态放大梯度再缩放回去，确保数值安全，提升训练稳定性。

torch.compile() 加速的核心，就是通过消除 Python 调度开销 (Python overhead) 和算子融合 (kernel fusion)，生成大块高效代码，提升模型运行速度。这种方式将多个小操作融合为一个 kernel，避免不必要的数据搬运和中间结果存储。

flash atten 找到了一些 torch.compile 找不到的 kernel fusion。Flash Attention 是针对传统 attention 的底层重写，更省显存，计算效率更高，可以处理更长的序列，属于一种 kernel-level 的精度和性能改进。它补足了 torch.compile 无法自动优化的一部分。

vocab size 50257-50304。GPT-2 使用的是 byte-level BPE tokenizer，基础词表为 50257，50304 和 2 有关，有助于提升训练效率。

2.2.3 Section 3 训练中的 trick

本节介绍训练过程中的关键技巧，从学习率调度到大规模模型的并行优化。

lr schedule

先 warm up，然后按照 lr schedule 的规定在训练中更新参数。warm up 一般指在训练开始的前几个 epoch 里线性上升学习率，防止训练初期梯度不稳定；之后按设定的 lr schedule（如 cosine decay）平滑降低学习率，帮助模型更稳地收敛。

`configure_optimizers()` 创建优化器，控制参数更新逻辑，`lr schedule + warmup` 控制优化器的学习率随训练进度如何变化，它们配合使用，提升训练效率与稳定性。这个函数不仅设置 optimizer 类型（如 AdamW），还决定了 scheduler 怎么接入训练循环，是学习率动态变化的控制中心。

梯度累积 = 多次反向传播 + 一次优化器更新，让你用有限显存模拟大 batch 训练，是大模型训练的重要技巧。尤其当一个 batch 太大会爆显存时，可以累积多个小 batch 的梯度，最后一次性做优化器的参数更新，达到 batch size 增大的效果。

数据并行 = 多 GPU 同时处理不同数据，模型结构一样，梯度同步后统一更新参数，是训练提速和扩容的关键技术。比如使用 PyTorch 的 `DataParallel` 或 `DistributedDataParallel`，每个 GPU 处理一部分数据，训练完后做 AllReduce 汇总梯度，从而保持多个副本的一致性。

2.2.4 Section 4 继续补充

本节用于延展补充未在前文细讲的细节内容，进一步完善对 GPT-2 训练过程的理解，包括断点续联机制与训练参数动态调整策略。

断点续联 (checkpoint & resume) 是训练长时间模型时的重要机制，确保中断后可以从保存点继续训练，避免算力浪费。每隔固定步数保存模型状态（包括权重、optimizer、lr schedule 状态等），当任务被 kill 或崩溃时，可以通过加载这些状态无缝恢复训练。

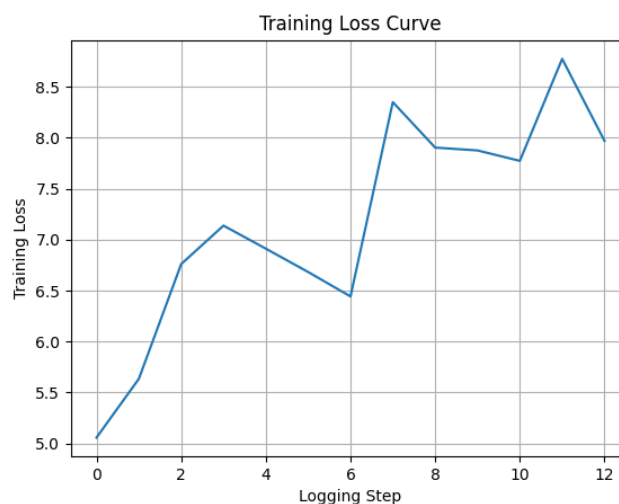
训练参数的调整 在训练过程中根据观察结果动态修改如 dropout rate、batch size、学习率等参数。

3. LoRA Fine-tuning

3.1 运行记录

保留了运行记录的 ipynb 文件在 `hw1-code/lora.ipynb` 中。

3.2 原始数据集的首次运行结果



训练参数占比：

```
trainable params: 466,944 || all params: 559,607,808 || trainable%: 0.0834
```

模型输出：

```
["I love this movie because them: them it the them; these your still they such one
though each our where that yet great Polish their there few he stabbing novel
thief like need needs work being having when how mean means policemen country new
its some too very example later included expect those my course giving telling
behind promise even really once lying They you trying attending!). writing )\n
both she little detail practice heat several many developing separate different
multiple more less easily quite have don't between after while if actually The St
watching every either probably waiting middle Britain at military his"]
```

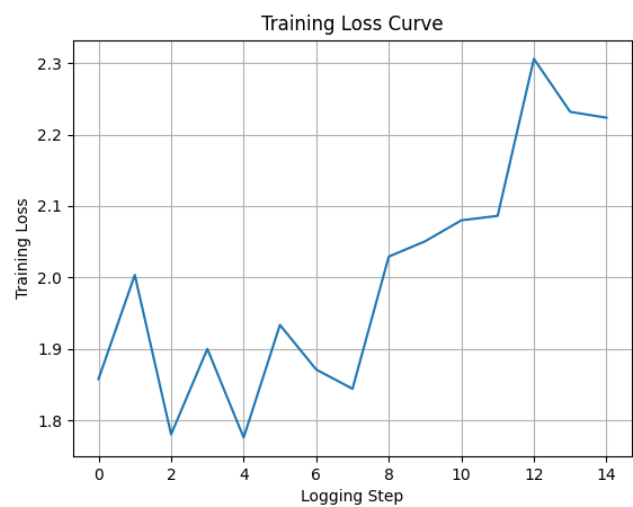
观察实验结果可知，loss 整体较高并且 loss 呈现上升趋势，训练效果不好。并且模型输出混乱无逻辑。一方面是learning rate太高，模型学习太快。一方面是 Lora 参与训练的参数只有 0.08%。

3.3 原始数据集上 lora 超参数对模型效果的影响

3.3.1 learning rate

尽管 learning rate 并不是 lora 超参数，但是对训练过程同样有关键作用。为保证训练过程顺利开展，首先试图寻找合适的 learning rate。我定性发现，在原始超参数设定基础上适度降低learning rate后，loss 在训练过程中呈下降趋势。因此，对 learning rate 分别为1e-3和1e-4进行了实验。

1. lr=1e-3

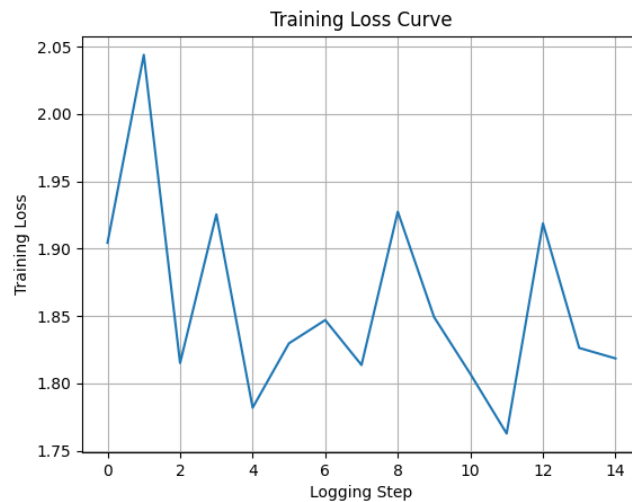


输出结果：["I love this movie because I think it is a great film. It was so good, but not quite as bad!

The plot has been very well done and the acting really did have some of its own quirks too much to be true storytelling with lots more than that... The characters are all pretty decent (and yes they were also excellent in

their performances). But that's just what you get for your money!! And then there's another one which isn't even remotely worth watching if you're going crazy about them"]

1. $lr=1e-4$



["I love this movie because it is so good. I have to say that the first time we saw The Last of Us was when he played a guy who had been in jail for years and then went on his own, but that's not what happened here; he's just as bad at killing people with no reason or purpose (and even if you were going out there looking like you're trying really hard), it's all about how much fun they make him feel by being able do things without any intention whatsoever - which makes me think they're doing"]

通过实验可知，学习率为 $1e-3$ 时,发现 loss 曲线依旧呈现上升趋势；学习率减小到 $1e-4$ 时，loss 曲线呈现下降趋势。经更多实验尝试，最终确定学习率 $1e-4$ 为最优参数。

3.3.2 其他训练超参数

同时，为了让模型训练更稳定，效果更好，我经过实验，对一系列训练参数进行了优化，调整了 `num_train_epochs`, `warmup_steps`, `gradient_accumulation_steps`, `weight_decay` 等。

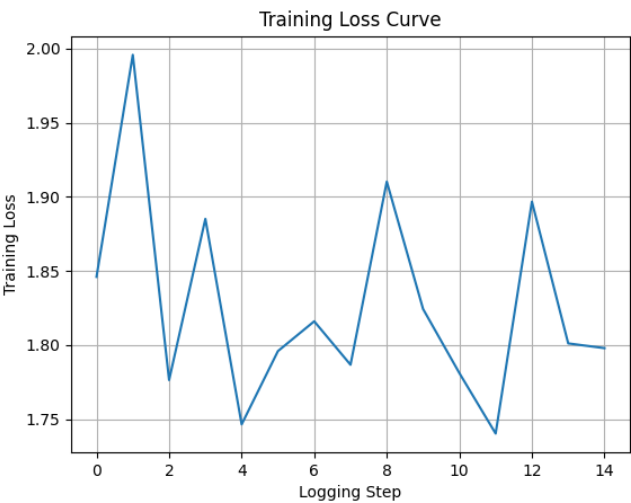
调整后，训练超参数设定如下：

```
learning_rate = 1e-4
num_train_epochs = 5
warmup_steps = 15
gradient_accumulation_steps = 2
weight_decay = 0.01
```

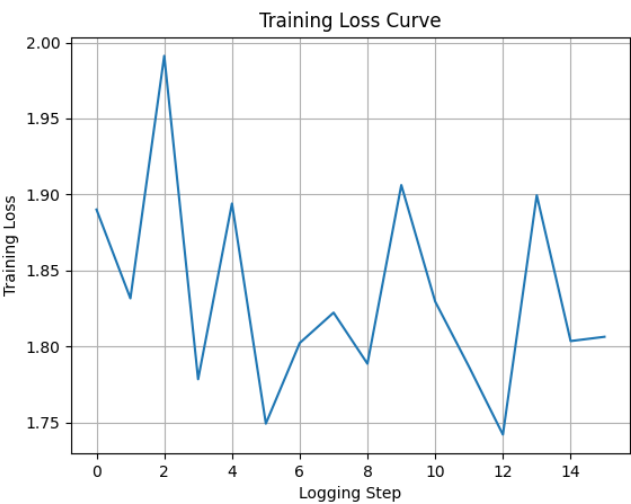
3.3.3 lora 参数 r 的作用

为了探究 lora 参数 r 的作用，我固定其他参数不变，分别设定 $r=a$ 和 $r=b$ 进行了测试，结果如下：

1. $r=32$



1. $r=64$

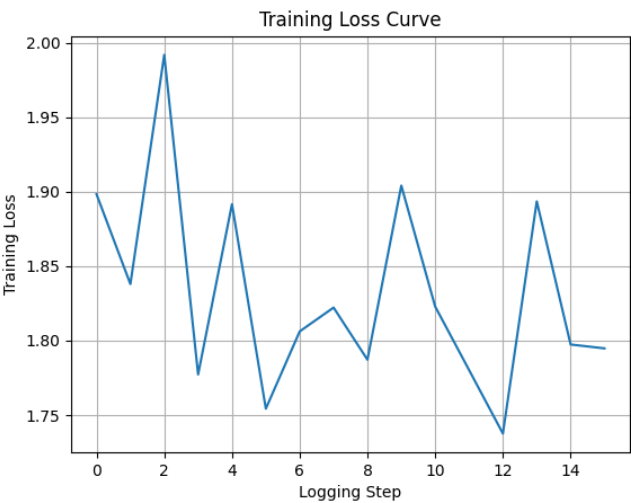


由此可见，改变 r 对训练结果影响不大。

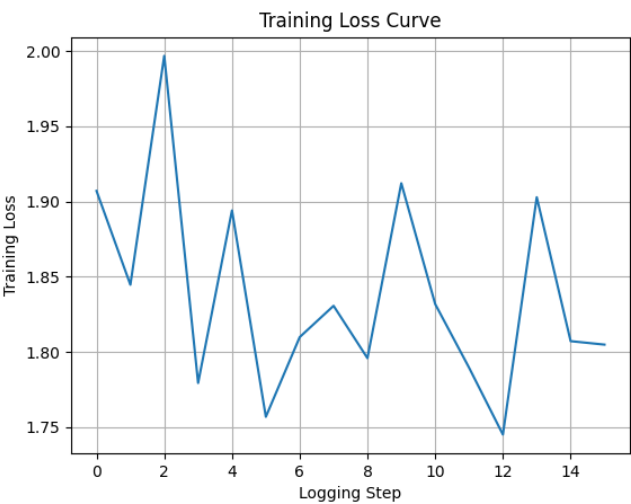
3.3.4 lora 参数lora_alpha的作用

为了探究 lora 参数lora_alpha的作用，我固定其他参数不变，分别设定了 lora_alpha=32和lora_alpha=64进行了测试，结果如下：

1. lora_alpha=32



1. `lora_alpha=64`

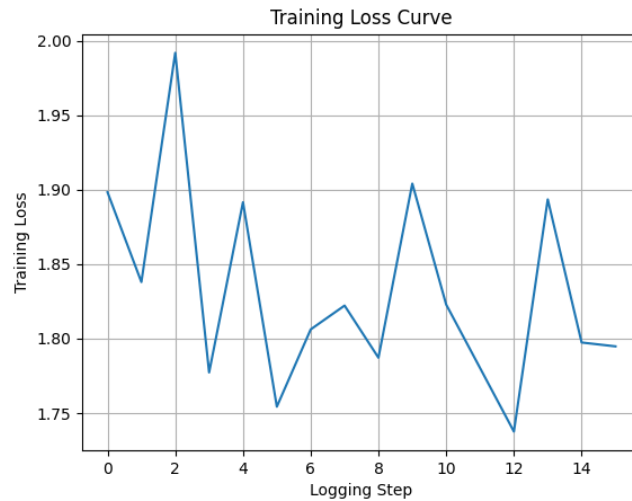


由此可见，改变`lora_alpha`对训练结果影响不大。

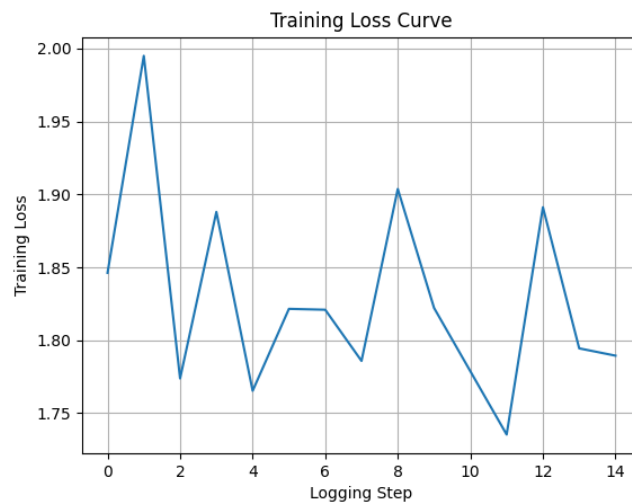
3.3.5 `lora` 参数`target_modules`的作用

为了探究 `lora` 参数`target_modules`的作用，我固定其他参数不变，分别设定了 `target_modules=["query_key_value"]`和`target_modules=["query_key_value", "dense"]`进行了测试，结果如下：

1. `target_modules=["query_key_value"]`



2. `target_modules=["query_key_value", "dense"]`



由此可见，改变`target_modules`对训练结果有一定影响，加入 `dense` 层训练效果变好。

本实验所使用的模型规模较小，且训练数据量有限，同时 `bias` 与 `lora_dropout` 两个参数本身在 LoRA 中的可调空间相对较小，因此在本轮实验中并未对这两个参数进行深入调优，仅采用了默认设置（`bias="lora_only"`，`lora_dropout=0.05`）。在后面的实验中尝试对这两个参数进行调整（设置 `bias="none"` 以及 `lora_dropout=0.1`），但观察到其对模型性能的影响仍较为有限，未体现出显著的提升或变化。

3.3.8 各个 `lora` 超参数作用分析

`r`是决定加入多少新参数，值大一点模型能学得多，但也容易过拟合。`lora_alpha`控制这些新参数对结果的影响，`target_modules`就是设定模型要改哪些部分，`lora_dropout`在训练中加入随机性，防止模型死记硬背，`bias="lora_only"`，就是只改 LoRA 那部分，别的地方不动，比较稳。`lora_alpha` 和 `r` 一直保持 2 的比率，试着大幅改变这一比率但结果变化不大，可能是模型比较小或者数据量太小的原因，所以维持该比率调整`r`以及层数`target_modules`。

3.4 Alpaca 数据集上使用最优超参数的训练结果

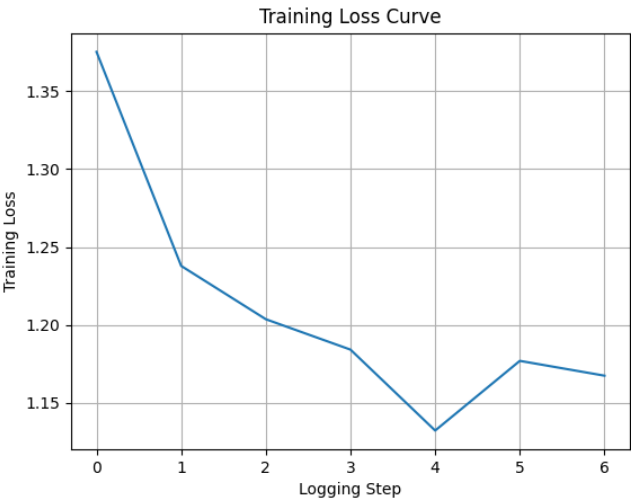
说明

这里是指令微调，是不同的任务。拼接instruction和input作为模型的输入（prompt），以output作为生成目标，同时通过设置label中prompt部分为-100，这样模型只在输出部分计算loss。训练时报维度错误，所以对所有输入进行了手动padding，让所有输入长度一致，同时返回Python的list类型而非tensor，以满足Hugging Face datasets.map()的要求。按照上个数据集的参数，loss呈下降趋势但未收敛，继续进行调整。

3.5 Alpaca 数据集上 lora 超参数对模型效果的影响

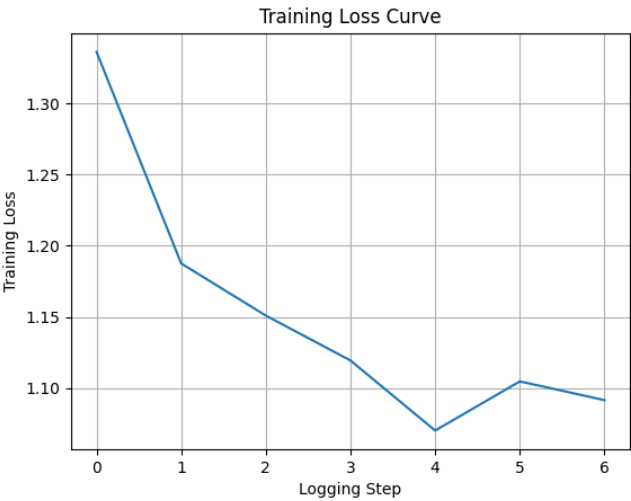
首先就是r越高 loss 的表现越好，这里写出 32（+0.1），64（+dense）的效果

32（+lora_dropout=0.1）：



[I love this movie because it is so funny and I am sure that the characters are very well developed. The acting was excellent, especially in terms of supporting roles such as Tom Hanks (the voice actor), who played a great role with his natural charm to make him seem like he could be trusted by anyone but himself! It also made me think about how much we can learn from each other when faced together on difficult times or even if there were any differences between us at one point; all these things helped bring out our'] trainable params: 3,219,456 || all params: 562,360,320 || trainable%: 0.5725

64（+dense）：



[I love this movie because it is so funny and I am sure that everyone will enjoy the ending. It was a great film!'] trainable params: 9,535,488 || all params: 568,651,776 || trainable%: 1.6769

分析

在第一轮的基础上再次调整部分 Lora 参数看效果：由于指令微调相对于原本的任务较为复杂，可能需要更高的 r 效果会更好。改动 `bias=None` 没用，loss 基本没动，猜测是模型本身包含 `bias` 的层就很少，调整 `bias` 对模型的能力影响有限。`lora_dropout` 调整起来有用一点点，固定在 0.1，提升了模型训练的鲁棒性。 $r = 64, lora_alpha = 128$ 似乎越大效果越好，但感觉是过拟合。必须要加上 `dense` 层，属于注意力层的一部分，加了之后训练效果变好，loss 有所下降。因为二者任务本身有区别，loss 也差别很大。