# HW1


- Author: 刘智琦
- StudentID: 2300012860
- School: 信息科学技术学院

- Github **链接（点击跳转）**

## 1.1 BPE

**BPE** 是什么

- 它**寻**找文本里**总**是挨在一起出**现**的字符**组**合，并把它**们变**成一个新的 token（比如把 “t” 和 “h” 合并成 “th” 块）
- 目**标**是从最基本的子**节**开始，建立一个包含常用字符和它**们**常**见组**合的清单
- 合并之后，可以大大减少文本的 tokens 数

### 训练流程

- 准**备**大量文字数据
- 从每个字**节**作**为**一个 token 开始（vocab_size=256）
- 不断找到最常出**现**的相**邻词**块对，然后把它**们**合并成一个新的**词**块
- 重复这个合并过程，直到**词**块的**总**数量达到我**们**想要的大小（或**语**料用完了）
- 最后得到 tokens 列表

### 测试 my_tokenizor

**经过测试，**encode 再 decode manual.txt，与原始 manual.txt 完全一致

使用 checkout.py 中的 checkout_bpe 函数**测试**

比**较 hf_tokenizor** 和 **my_tokenizor**

| tokens | 英文句子 | 中文句子 |
|---|---|---|
| my_tokenizor | 942 | 118 |
| hf_tokenizor | 185 | 306 |

不同的原因主要是：

- my_tokenizor 仅使用中文**语**料来**训练**
- hf_tokenizor 主要在英文**语**料上**训练**

回答**问题**

- Python 中使用 ord()查看字符的 Unicode，chr()将 Unicode **转换成字符**
  - “北”: 21271
  - “大”: 22823

    - 22823: 大
    - 27169: 模
    - 22411: 型
- vocab size
    - 大
        - 优点：文本的 tokens 少，有助于模型直接学习常用词的语义
        - 缺点：训练更困难，参数量更大
    - 小
        - 优点：训练简单，参数量小
        - 缺点：文本的 tokens 多
- LLM 不能处理非常简单的字符串操作任务，比如反转字符串
    - 因为字符串被拆解为几个 tokens，一个 token 往往包含多个字母，LLM 不能直接知道这个 token 是由哪几个字母组成的
- LLM 在非英语语言（例如日语）上表现较差
    - 因为这些语言缺少天然分词
    - 英文只有 26 个字母，中文有大量汉字，不利于 BPE
    - 用于训练的英文语料多，其他语言的语料少
- LLM 在简单算术问题上表现不好
    - 因为多个连续的数字被编码成一个 token，LLM 不理解这个 token 的数学含义
- GPT-2 在编写 Python 代码时遇到比预期更多的困难
    - 因为在 GPT-2 中，连续的空格被视作多个单独的表示空格的 token，这对于 python 这种需要严格缩进且包含大量空格的语言不友好（缩进数不对，文本的 tokens 太长）
- LLM 遇到字符串 “<|endoftext|>” 时会突然中断
    - 因为这是一个特殊的 EOF Token，表示文本结束
- 当问 LLM 关于 “SolidGoldMagikarp” 的问题时 LLM 会崩溃
    - 因为 LLM 基于概率来预测输出，当遇到罕见的输入，LLM 预测下一个 token 的概率分布会变得混乱，导致它生成任何 token 的可能性都差不多，结果就是输出随机、重复或完全无关的文本
- 在使用 LLM 时应该更倾向于使用 YAML 而不是 JSON
    - 因为 YAML 格式对应的 tokens 更少
- LLM 实际上不是端到端的语言建模
    - LLM 的输入不是原始的文本字符，而是经过 Tokenizer 处理后得到的数值化 token 序列

## LLM Implementation

**Commit 1 “initial commit”**

实**现**：

- 定义 GPTConfig
- Self-Attention 模块
- MLP 模块
- 把 Self-Attention 和 MLP 拼成 Block
- 定义 GPT
- 加**载** Hugging-Face 上的 GPT-2

作用：

1. Token_Embedding
   - vocab.shape = (vocab_size, n_embd)
2. Position_Embedding
   - posit.shape = (block_size, n_embd)
3. Self-Attention
   - B: Batch Size (批量大小)
   - T: Sequence Length (序列**长**度)
   - C: Embedding Dimensionality (n_embd) (嵌入**维**度)

   1. In_Tensor.shape = (B, T, C)
   2. QKV.shape = (B, T, 3 * C)
   3. Q.shape = K.shape = V.shape = (B, T, C) -> (B, nh, T, hs)
   4. att.shape = (B, nh, T, T)
   5. y.shape = (B, nh, T, hs) -> (B, T, C)
4. MLP
   1. In_Tensor.shape = (B, T, C)
   2. Mid_Tensor.shape = (B, T, 4 * C)
   3. Out_Tensor.shape = (B, T, C)

笔**记**：

- **计算** QKV 时，可以使用 shape = (n_embd, 3 * n_embd) 的**线**性**层**来一次性**计**算所有头的 QKV 矩**阵**，然后再沿最后一个**维**度 split() 得到 QKV
- nn.Linear(in_dim, out_dim) 要求**输**入的**张**量的最后一个**维**度 = in_dim，其余**维**度不重要，即 In_Tensor.shape = (*, in_dim)

  Out_Tensor = nn.Linear(In_Tensor), Out_Tensor.shape = (*, out_dim)
- 在 Numpy 中，M.T 交**换**所有**维**度，在 PyTorch 中，M.T 交**换**后两个**维**度
- Tensor.view() 要求 Tensor 在内存中的存**储**是**连续**的，但**经**过 transpose(), permute() 或复**杂**的切片/索引操作后，Tensor 往往是非**连续**的

  所以**经**过这些操作后需要先**应**用 contiguous() 再 view()
- n_head * head_size = n_embd

多头注意力实际上是把 n_embd 分割成 n_head 个片段，每个 head 处理一个片段，最后再把所有处理过的片段合起来。这样可以让不同头捕捉不同的依赖关系，并提高并行效率

- Block 中需要定义两个 nn.LayerNorm。虽然它们参数相同，但是运算过程中梯度等不能共用，所以需要分开
- 为了正确地管理 nn.Module，需要使用 nn.ModuleList 和 nn.ModuleDict 来存储
- buffer 用于存储不需要学习的参数。在使用 state_dict 时，需要过滤掉 buffer，一方面是 buffer 是固定的，不需要保存，另一方面是不同来源的模型的 buffer 的键名或形状可能不匹配，从而在应用时出错
- 在 Pytorch 中，使用 with torch.nograd(): 来临时禁用梯度计算，此时 PyTorch 会停止追踪这个块内部所有张量上的操作，因此不会构建计算图，也不会计算梯度
- 在 PyTorch 中，方法名后面带有下划线（如 copy_, add_, zero_ 等）通常表示这是一个 in-place (原地) 操作。这意味着这个方法会直接修改调用它的 Tensor，而不是返回一个新的 Tensor
- hugging-face 的矩阵的维度和 Karpathy 的代码中的维度相反，所以需要 transpose

**Commit 2 “add forward() function of GPT2 nn.Module”**

实现：

- GPT 的 forward()

作用：

- 前向传播

笔记：

- torch.arange 初始化时，可能会默认使用 torch.int32，而 nn.Embedding 要求输入的索引是 torch.long 类型，所以需要在初始化时声明 dtype=torch.long

**Commit 3 “generate from the model”**

实现：

- 加载 GPT-2 的 tokenizer
- 生成了几句话来测试模型是否正确

作用：

- 现在可以做推理了

笔记：

- 在 PyTorch 中，模型的层（例如 Dropout 和 Batch Normalization）在训练和评估模式下的行为是不同的
  - 训练模式 (model.train()): 层会根据需要进行更新（例如，Dropout 层会随机丢弃神经元，Batch Normalization 会计算并更新移动平均统计量）
  - 评估模式 (model.eval()): 在评估或推理过程中，需要确定性的行为。model.eval() 会关闭 Dropout，并使用训练期间计算得到的移动平均统计量来执行 Batch Normalization，而不是使用当前批次的统计量。这确保了模型在推理时输出是确定的，并且不受批次大小的影响
- 记得把 Tensor 移动到正确的 device 上
- tensor.unsqueeze(i) 用于在 i 处增加一个维度
- tensor.repeat(n_1, n_2, ..., n_k) 由于在各个维度复制 n_j 次
- torch.manual_seed(42) 用于设置 CPU
- torch.cuda.manual_seed(42) 用于设置 GPU
- logits[:, i, :] 代表基于前 i+1 个 tokens，对第 i+2 个 token 的预测
- Top-K 采样是推理阶段的策略，训练时不用。K 越大，文本越具有创意，K 越小，文本越不容易出错
- torch.topk 用于 Top-K 采样
- torch.multinomial 不要求输入概率和为 1，依概率生成指定个输出
- torch.gather 给定“查找表”和“索引表”，收集数据
- torch.cat 拼接 Tensor

**Commit 4 “autodetect device, and switch to a random model”**

实现：

- 添加了 device 检测机制

作用：

- 自动选择合适的 device

笔记：

- 使用 device = “mps” 来在 Apple Silicon GPU 上运行
- 部分 Pytorch 可能不包含 “mps”，所以检查 torch.backends.mps.is_available() 前最好先检查 hasattr(torch.backends, “mps”)

**Commit 5 “add tiny shakespeare and create an example little batch out of it”**

实**现**：

- 添加 Shakepear 数据集

作用：

- **为训练做准备**

笔**记**：

- 由于需要对比“**预测**”和“实**际**”来**训练**，所以**应该**取前 B * T + 1 个 token

**Commit 6 “calculate the loss function: cross entropy loss”**

实**现**：

- **计算** loss

作用：

- 可以**计**算 loss 了

笔**记**：

- view(-1) 中 –1 是一个特殊的占位符，view() 会根据其他指定的**维**度大小，自动**计**算出 –1 的大小
- 可以使用 targets=None 可选参数和 if targets is not None: 判断**语**句来用一个 forward() 同时实**现**推理和**训练**

**Commit 7 “little loop crushes a little batch”**

实**现**：

- 添加 optimizer

作用：

- 可以更新 grad 了

笔**记**：

- **训练步骤**
  1. 清零梯度
  2. 前向**传**播
  3. 反向**传**播
  4. 更新参数

**Commit 8 “add a DataLoaderLite”**

实现：

- 实现了 DataLoader

作用：

- 加载数据
- 逐 batch 训练

笔记：

- 注意 Input 和 Target 之间差一个索引
- 由于 DataLoader 在 CPU 上，所以训练时每次循环都要调用一次 to(device)

  把 DataLoader 放在 device 上可以优化这个问题

**Commit 9 “weight tie the embedding and unembedding matrix”**

实现：

- 添加 wte.weight 和 lm_head.weight 的绑定

作用：

- 提高模型的性能和泛化能力

笔记：

- 将一个词从离散的 token 空间映射到连续的嵌入空间（词嵌入层），以及将模型在嵌入空间的输出映射回离散的 token 空间（语言模型头），这两个任务是高度相关的。共享权重鼓励模型学习这种对称性，可以帮助模型更好地理解词汇之间的关系，尤其是在词汇表很大的情况下，有助于提高模型的性能和泛化能力
- 当 wte.weight = lm_head.weight ，logits = self.lm_head(x) 类似于让 x 和每个 token 对应的向量表示做点积，
- 在 PyTorch 中，执行 tensor_a = tensor_b，tensor_a 将指向 tensor_b 所指向的同一个底层数据

  于是 wte.weight = lm_head.weight 相当于让 2 个层共享权重，在更新时先累计梯度再更新

**Commit 10 “gpt-2 initialization”**

实现：

- 添加初始化功能

作用：
- 初始化参数

笔**记**：

- Module.apply(fn) 可以递归地把 fn() **应用于自身和自身的每个子模块**
- 把残差**连**接前的全**连**接**层**的方差适当减小，有助于提升**训练**初期的**稳**定性
- 在 Python 中，**给**实例不存在的属性**赋**值，实例将**创**建这个新的属性
- 利用 NANOGPT_SCALE_INIT 这个**标记**，可以区分残差**连**接前的全**连**接**层**和普通全**连**接**层**，从而 _init_weights() 可以用一个代**码**块处理两种全**连**接**层**，减少代**码**量

**Commit 11 “also add jupyter notebook i think”**

实**现**：
- 添加 .ipynb 文件

作用：
- 可以通过 .ipynb 文件来运行程序

笔**记**：
- 无

**Commit 12 “add TensorFloat32 tf32 matmuls”**

实**现**：

- 添加**计**时**装**置
- **设置** float32 矩**阵**乘法的精度**为** “high”

作用：
- **计时**

笔**记**：
- 无

**Commit 13 “add bfloat16”**

实**现**：

- 添加自动混合精度**训练**

作用：

- 自动混合精度**训练**

笔**记**：

- torch.autocast() 用于启用自动混合精度**训练**
- 推理时使用低精度，反向**传**播和更新梯度时使用高精度

**Commit 14 “add torch compile”**

实**现**：

- 添加 torch.compile()

作用：

- 自动优化，提升性能

笔**记**：

- torch.compile() 可自动提升性能

**Commit 15 “switch to flash attention”**

实**现**：

- **转换为** Flash Attention

作用：

- **节省内存**
- 加速

笔**记**：

- Flash Attention 可以大大**节**省内存并加速运算

**Commit 16 “vocab size 50257 -> 50304 nice”**

实**现**：

- 使 vocab_size 因数分解后包含 2 的高次幂

作用：

- 加快**计**算速度

笔**记**：

- 参数尽量包含 2 的高次幂

**Commit 17 “make print nice:”**

实**现**：

- 改**进输**出

作用：

- 美化**输**出

笔**记**：

- :.6f 表示保留 6 位小数
  - 2.0 -> 2.000000
  - 3.1415926 -> 3.141593
- :4d 表示整数**宽**度**为** 4
  - “100” -> “ 100”
  - “1000” -> “1000”
- 上述操作需要用 f{} 来**应**用

**Commit 18 “AdamW params and grad clipping set”**

实**现**：

- **调整** optimizer 参数
- 添加 gradient clipping

作用：

- gradient clipping 限制梯度的最大范数，防止梯度爆炸

笔**记**：

- AdamW 的 beta1 越大，历史梯度影响越大， beta2 越大，越关注最近的梯度变化
- torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0) 由于 Gradient Clipping，限制梯度的最大范数（norm）为 1.0，防止梯度爆炸，使训练更稳定

  如果总范数超过 1.0，则按比例缩小所有梯度；如果总范数小于 1.0，则保持梯度不变

**Commit 19 “add learning rate scheduler”**

实现：

- 添加学习率调控装置

作用：

- 使不同训练阶段采用不同的学习率，从而兼顾模型训练稳定性和速度

笔记：

- 在不同训练阶段采用不同的学习率，可以兼顾模型训练稳定性和速度
- 一种学习率设置方法是分为三个阶段
  1. 热身阶段，步数少，学习率快速线性上升
  2. 学习率以余弦函数下降
  3. 学习率不变

**Commit 20 “add weight decay, only for 2D params, and add fused AdamW”**

实现：

- 新增配置优化器方法
- 新增 fused AdamW

作用：

- 可以为 2D 参数添加权重衰减，可以通过 GPT 获取优化器
- 若可行，则启动 fused AdamW

笔记：

- 为 2D 参数添加权重衰减可以提高泛化性

  获取优化器的函数可以写在 GPT 中
- fused AdamW 可以提高更新参数的速度

**Commit 21 “add gradient accumulation”**

实现：

- 添加梯度累计

作用：

- 使用梯度累计来在不增加显存占用的情况下模拟更大的 batch

笔记：

- 运用梯度累计时需要除以 grad_accum_steps
- 监控累计损失时，需要确保 loss_accum 不在计算图中，除了可以使用 .detach()，也可以直接用一个 float 变量来累计

**Commit 22 “add DistributedDataParallel training”**

实现：

- 分布式训练

作用：

- 实现分布式训练

笔记：

- 启动方式
  - 简单的启动方式:

    python train_gpt2.py
  - DDP (分布式数据并行) 启动方式，例如 8 个 GPU:

    torchrun –standalone –nproc_per_node=8 train_gpt2.py
- torchrun 命令会自动设置环境变量 RANK, LOCAL_RANK, 和 WORLD_SIZE
  - RANK (全局排名)
    - 当前进程在整个分布式训练任务中的唯一标识符（ID）。它是一个从 0 到 WORLD_SIZE - 1 的整数
    - RANK=0 的进程通常被指定为主进程（master process），负责一些协调工作，例如初始化、日志记录、保存模型等
    - 其他进程（RANK > 0）是工作进程（worker processes），主要负责模型训练任务的不同部分
  - LOCAL_RANK (本地排名)
    - 当前进程在当前节点（例如，一台机器）内的唯一标识符（ID）。它是一个从 0 到 N - 1 的整数，其中 N 是当前节点上的进程数（通常等于该节点上的可用 GPU 数量）

- WORLD_SIZE (全局大小)
  - 参与当前分布式训练任务的总进程数（或参与训练的 GPU 总数）

- 计算时应当根据 self.process_rank 作一定调整（相比单 GPU）

**Commit 23 “add the grad accum mini example to ipynb file”**

实现：

- 在 .ipynb 中添加梯度累计的演示

作用：

- 演示梯度累计的写法为什么是示例中那样

笔记：

- 无

**Commit 24 “switch to FineWeb EDU”**

实现：

- 数据分块预处理

作用：

- 把过大的数据集分块成较小的元素组成的列表

笔记：

- tqdm 库用于在循环或迭代过程中显示智能进度条
- Python 中，科学计数法（如 1e8）是浮点数，可以用 int() 转换为整数（int(1e8)）
- os 可以实现系统级的操作，比如 os.path 可用于获取路径、智能拼接路径等
- 由于一个训练集太大，所以需要拆分为多个 shard
- 设置进程数，通常为 CPU 核心数的一半，以避免过多的进程导致系统资源耗尽

**Commit 25 “add validation split”**

实现：

- 添加训练过程中的评估功能

作用：
- 查看随着训练的进行，模型的输出的变化

笔记：
- 无

**Commit 26 “move up the samepling code into the main loop, but disable it because it doesn’t work with torch.compile and i’m not sure why :(“**

实现：
- 禁用训练过程中周期性生成文本样本

作用：
- 防止和 torch.compile() 出现兼容性 bug

笔记：
- 使用 torch.compile() 可能会出现兼容性 bug

**Commit 27 “hellaswag adeval d to main train file, and logging”**

实现：
- 添加 HellaSwag 评估集
- 在训练过程中监控在 HellaSwag 上的性能

作用：
- 现在可以使用 HellaSwag 来评估模型的性能了

笔记：
- 无

**Commit 28 “add readme v1”**

实现：
- 添加 Readme

作用：

- 添加 Readme

笔**记**：
- 无

**Commit 29 “add discord and discussions”**

实**现**：
- 添加 discord 和 discussions

作用：
- 添加 discord 和 discussions

笔**记**：
- 无

**Commit 30 “add example generations from 124M”**

实**现**：
- 添加模型生成示例

作用：
- 展示模型能力

笔**记**：
- 无

**Commit 31 “readme tweaks more infos”**

实**现**：
- Readme 添加更多信息

作用：
- Readme 添加更多信息

笔**记**：
- 无

**Commit 32-44 “**美化 **&** 小修小**补”**
实**现**：

- **为程序做了一些小修小补**
- 美化
- 添加 youtube **链接等**

作用：
- 小改**进**

笔**记**：
- 无

## LoRA Fine-tuning

**IMDB** 数据集超参数分析

为了找到在 IMDB 影评数据集上微调 GPT-2 模型的最佳 LoRA 超参数，我们进行了一系列实验。实验探索了不同的 `rank`, `alpha`, 和 `dropout` 组合对最终模型损失（final loss）的影响。

下图展示了不同超参数配置下的最终损失和可训练参数量的对比。

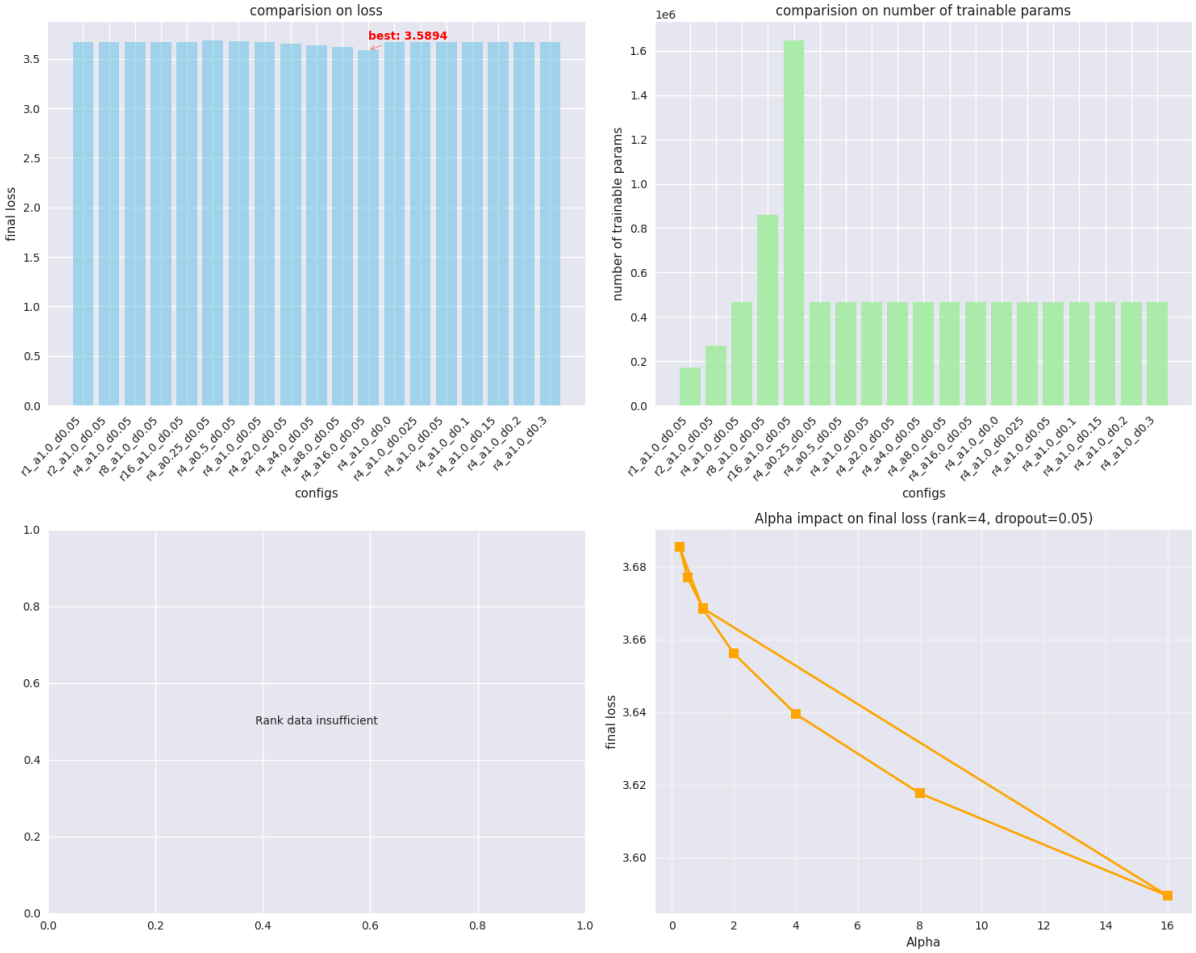


图 1　IMDB 数据集超参数搜索结果。左上图展示了不同配置的最终损失，右上图是对应的可训练参数量，右下图展示了固定 rank=4 和 dropout=0.05 时，alpha 对损失的影响。

实验结果汇总:

| rank | alpha | dropout | final_loss | trainable_params | train_time |
|---|---|---|---|---|---|
| 1 | 1.00 | 0.050 | 3.6693 | 172,032 | 12.8s |
| 2 | 1.00 | 0.050 | 3.6692 | 270,336 | 12.9s |
| 4 | 1.00 | 0.050 | 3.6684 | 466,944 | 13.0s |
| 8 | 1.00 | 0.050 | 3.6696 | 860,160 | 13.6s |
| 16 | 1.00 | 0.050 | 3.6695 | 1,646,592 | 13.0s |
| 4 | 0.25 | 0.050 | 3.6854 | 466,944 | 13.1s |
| 4 | 0.50 | 0.050 | 3.6772 | 466,944 | 12.9s |
| 4 | 2.00 | 0.050 | 3.6561 | 466,944 | 14.0s |

| rank | alpha | dropout | final_loss | trainable_params | train_time |
|---|---|---|---|---|---|
| 4 | 4.00 | 0.050 | 3.6394 | 466,944 | 13.0s |
| 4 | 8.00 | 0.050 | 3.6176 | 466,944 | 12.9s |
| 4 | 16.00 | 0.050 | 3.5894 | 466,944 | 13.1s |
| 4 | 1.00 | 0.000 | 3.6682 | 466,944 | 12.9s |
| 4 | 1.00 | 0.025 | 3.6685 | 466,944 | 12.8s |
| 4 | 1.00 | 0.100 | 3.6686 | 466,944 | 13.0s |
| 4 | 1.00 | 0.150 | 3.6687 | 466,944 | 13.3s |
| 4 | 1.00 | 0.200 | 3.6684 | 466,944 | 13.2s |
| 4 | 1.00 | 0.300 | 3.6680 | 466,944 | 13.5s |

根据上图和表格数据，IMDB 数据集上的最佳配置为:

- **Rank:** 4
- **Alpha:** 16.0
- **Dropout:** 0.05
- 最终损失 **(Final Loss):** 3.5894
- 可训练参数 **(Trainable Params):** 466,944

**Alpaca** 数据集超参数分析

为了验证超参数的普适性，我们更换为 Alpaca（指令微调）数据集，并进行了相同的超参数搜索实验。

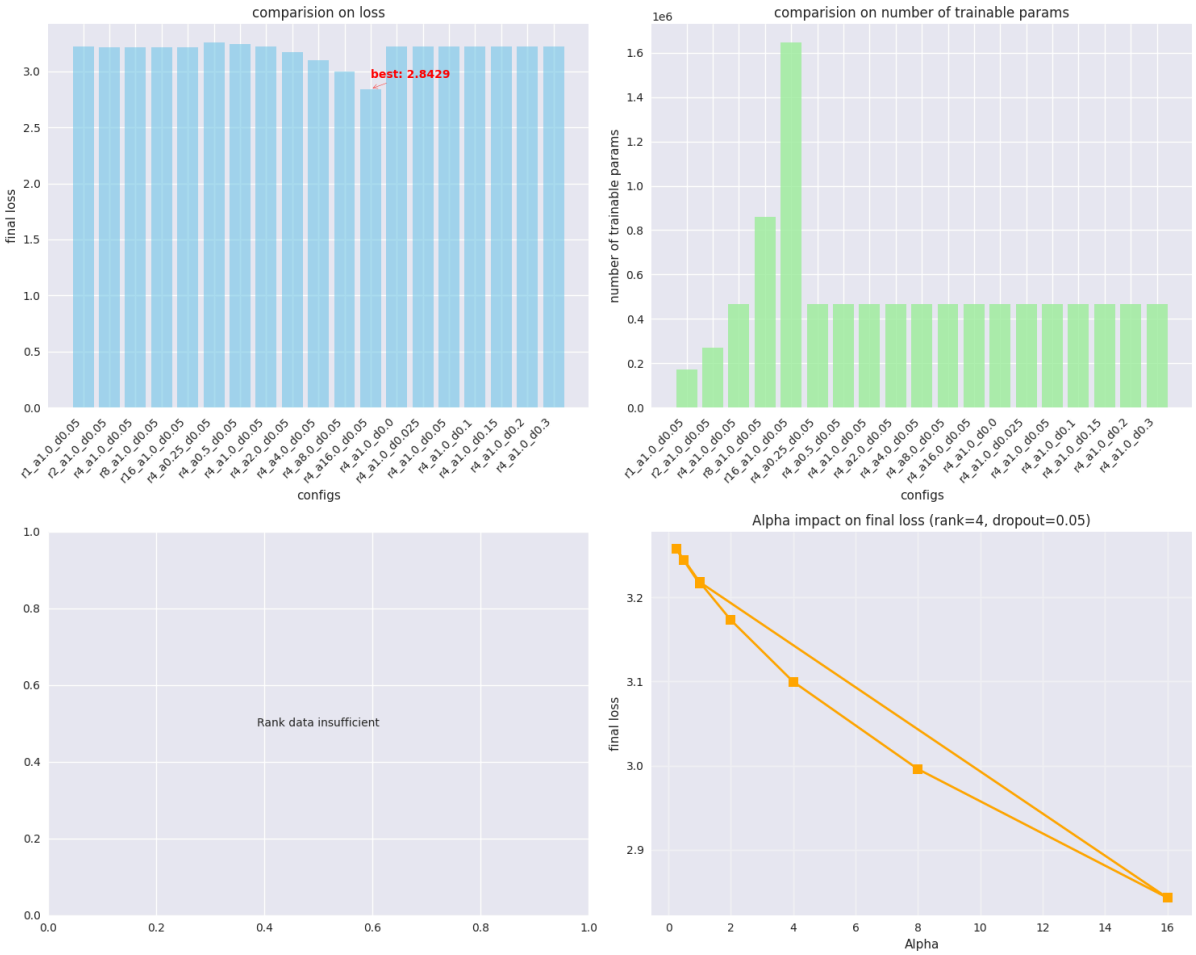


图 2　Alpaca 数据集超参数搜索结果。布局与 IMDB 实验图一致。

从 图 2 中可以看出，在 Alpaca 数据集上，损失的整体值更低。根据图中标注，我们得到了 Alpaca 数据集上的最佳配置：

- **Rank:** 4
- **Alpha:** 16.0
- **Dropout:** 0.05
- 最终损失 **(Final Loss):** 2.8429
- 可训练参数 **(Trainable Params):** 466,944

有趣的是，尽管两个数据集的任务（影评情感 vs. 指令跟随）和数据分布不同，但取得最佳性能的超参数组合惊人地一致。

**IMDB** 与 **Alpaca** 结果对比与结论

为了更直观地比较 LoRA 超参数在两个不同数据集上的表现，我们将结果汇总在下图中。

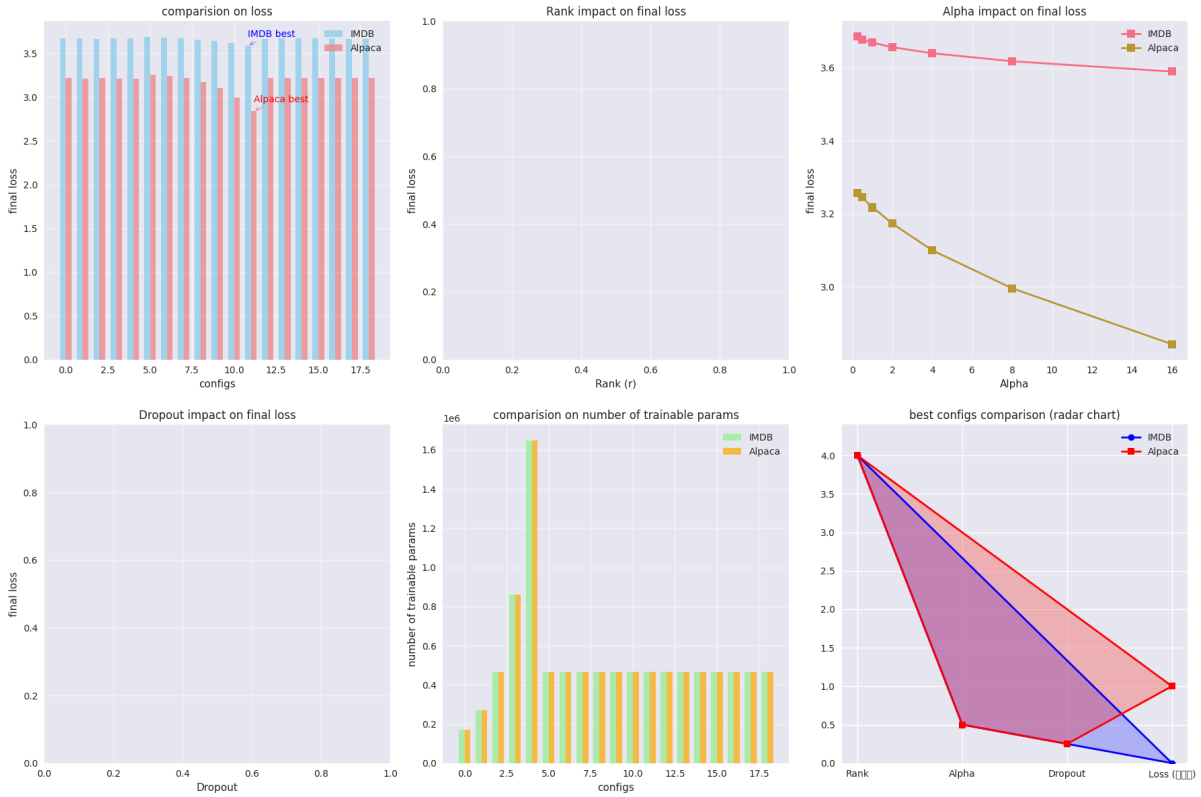


图 3　IMDB (蓝色/粉色) 与 Alpaca (红色/金色) 数据集在不同超参数下的性能对比。雷达图直观展示了两个数据集上最佳配置的异同。

从对比图中我们可以看到，两个数据集的最佳超参数一致，都是 rank $= 4$, $\alpha = 16$, dropout $=$ 0.05。这组参数的 rank 和 dropout 较均衡，而 alpha 值则较高。结果表明，较高的 alpha 有助于降低 loss，特别是对于需要模型行为发生更显著改变的 Alpaca 数据集，效果尤其明显。

该现象表明，不论具体下游任务是什么，最佳的 LoRA 超参数配置可能存在一定的一致性。我们可以从 LoRA 的更新公式来分析其原因：

$$W_{\text{new}} = W_{\text{original}} + (\text{LoRA}_B \times \text{LoRA}_A) \times \left(\frac{\alpha}{\text{rank}}\right)$$

**1. Rank=4:** 中等秩的选择

- **为什么选择中等 rank**？
  - 足够的表达能力: rank $= 4$ 提供了足够的低秩空间来捕获特定于任务的模式，而不会像更高 rank（如 16）那样引入过多可能导致过拟合的参数。
  - **计算效率:** 相比高 rank，训练更快，内存占用更少。

**2. Alpha=16:** 高 **Alpha** 值的作用

- 对 **IMDB** 数据集: 高 alpha 值让 LoRA 适配器产生的权重更新具有更大的影响力，能够有效地将预训练模型中性的文本续写风格，调整为带有明确情感色彩的影评风格。
- 对 **Alpaca** 数据集 (效果更明显): 指令跟随任务要求模型行为发生根本性的改变——从“续写文本”转变为“遵循指令”。预训练模型本身不具备这种能力，因此需要一个非常强的信号来重塑其行为。高 $\alpha = 16$ 提供了足够强的权重调整，促使模型完成了这一关键转变。

**3. Dropout=0.05: 轻度正则化**

- **为什么选择较低 dropout**？

- LoRA 本身参数量就很少，学习能力有限，过高的 dropout（如 >0.1）可能会抑制其学习过程。
- 对于我们使用的较小数据集，`0.05` 的 dropout 提供了基础的正则化，可以在不损害学习效率的情况下，有效防止过拟合。