

# HW1: 编程部分 (在GPT-2模型中实现RoPE、GQA)

2024年9月12日 10:07

1

## 1 概述

[Rotary Positional Embeddings \(RoPE\)](#)

[Grouped Query Attention \(GQA\)](#)

[实验任务](#)

## 2 项目文件

[1. requirements.txt](#)

[2. input.txt](#)

[3. chargpt.py](#)

[4. mingpt/](#)

[参数配置 \(Flags\)](#)

## 3 本项目的模型剖析

### [3.1 源数据、数据集的制作](#)

[txt文本概述:](#)

[数据集制作流程:](#)

### [3.2 模型结构](#)

### [3.3 优化器、学习器、损失函数、超参数等](#)

## 4 Rotary Position Embeddings (RoPE)

### [4.1 讲义原文](#)

[背景](#)

[RoPE机制](#)

[实现:](#)

### [4.2 RoPE的代码实现](#)

### [4.3 RoPE问题](#)

[1 \(4分\)](#)

[2 \(4分\)](#)

[3 \(2分\)](#)

[4 \(2分\)](#)

[解答:](#)

[以下分别是训练600个step \(序列长度=128\) 和800个step \(序列长度=256\) 情况:](#)

[训练600个step后, 切换序列长度的训练结果可以直接参考CMU学生的文件:](#)

[文本生成情况展示:](#)

[GPT4的评估](#)

[GPT4的评估2](#)

## [5 Grouped Query Attention \(GQA\)](#)

### [5.1 讲义原文](#)

[GQA简介](#)

[实现细节:](#)

### [5.2 GQA的代码实现](#)

### [5.3 GQA问题](#)

[5 \(4分\)](#)

[6 \(4分\)](#)

[7 \(4分\)](#)

### [5.4 关于GQA跑实验的结论](#)

# 1 概述

在“Programming: RoPE and GQA”部分, 主要任务是通过结合RoPE (旋转位置嵌入) 和GQA (Grouped Query Attention, 分组查询注意力) 这两种机制, 改进现有的GPT模型, 并观察这些改进对模型性能的影响。以下是对RoPE和GQA的介绍:

## Rotary Positional Embeddings (RoPE)

RoPE是一种相对位置嵌入方法, 用来取代传统的绝对位置嵌入。在传统Transformer中, 位置信息通过将位置嵌入直接加到输入的词向量中进行传播。而RoPE直接在每一层注意力计算中引入相对位置信息, 旋转每个查询和键向量的一部分来嵌入这些信息。

RoPE的数学表达为: 通过旋转矩阵对查询向量和键向量进行旋转, 计算出新的旋转后的向量, 并基于这些旋转向量计算注意力得分。与传统的绝对位置编码不同, RoPE使得位置信息能够在不同层之间保持一致, 并且具有较好的相对位置信息表达能力。

你将在此部分作业中实现RoPE, 并通过与传统minGPT模型的对比, 分析其在训练过程中的表现。你需要在 `mingpt/model.py` 文件中的 `RotaryPositionalEmbeddings` 和 `CausalSelfAttention` 类中进行修改。

## Grouped Query Attention (GQA)

GQA是一种用于优化注意力机制的技术。它通过将查询头分组，每个组共享一个键和值头，减少了注意力计算中的资源消耗。GQA的设计提供了一种在计算效率和模型质量之间取得平衡的方案。

GQA的基本思想是，将查询头按组进行分组，每组查询头使用相同的键和值头，从而减少了键和值的计算量。与标准的多头注意力机制不同，GQA在保持较好的注意力表现的同时，能够大幅度减少计算开销和内存占用。

你将在 `mingpt/model.py` 文件中实现GQA。你需要在 `GroupedQueryAttention` 类中实现这一机制，并进行一系列实验，验证其性能和效率改进。

## 实验任务

1. **RoPE实验**：实现RoPE后，你需要绘制RoPE实现和传统minGPT在600次迭代中的训练损失对比图，并进一步分析RoPE在更长序列长度（128和256）下的训练表现。
2. **GQA实验**：实现GQA后，你需要测量GQA在不同键头数量（如1、2、3、6）下的计算时间和内存消耗，并与标准的多头注意力机制进行对比。此外，还需要观察GQA和多头注意力机制在600次迭代中的训练损失表现。

通过完成这些实验，你将对RoPE和GQA在提升模型性能和效率方面的效果有深入理解。

## 2 项目文件

该项目的起始代码文件包含在名为“hw1”的目录中，以下是这些文件及其功能的概述：

### 1. `requirements.txt`

- **功能**：列出了此项目所需的Python依赖包。
- **内容**：该文件列出了两种主要的依赖：`torch` 和 `einops`，这两个包分别用于深度学习框架和高效的张量操作。

### 2. `input.txt`

- **功能**：存储了用于训练模型的数据集。
- **内容**：这是完整的莎士比亚作品集，大小约为1.1MB。

### 3. `chargpt.py`

- **功能**：项目的主入口，用于训练Transformer模型。
- **内容**：通过命令 `python chargpt.py` 启动训练，可以通过添加标志来调整Transformer的配置。此文件使用提供的RoPE和GQA配置来调整模型。

### 4. `mingpt/`

这是项目中的核心代码库，包含以下文件：

a. `model.py`

- **功能**：定义了GPT模型的构建，包括Transformer的各个层和注意力机制。
- **内容**：这是作业中需要修改的文件，包含基础的Transformer实现。你需要在此文件中实现以下两个类：
  - i. **RotaryPositionalEmbeddings**：用于实现RoPE旋转位置嵌入。
  - ii. **GroupedQueryAttention**：用于实现分组查询注意力机制（GQA）。你还需要修改**CausalSelfAttention**类以整合RoPE。

b. `trainer.py`

- **功能**：定义了训练循环和模型优化逻辑。
- **内容**：这个文件包含训练GPT模型的逻辑，负责训练步骤、损失计算和参数更新。

c. `utils.py`

- **功能**：提供一些辅助功能，如保存日志和配置。
- **内容**：这个文件包含帮助函数，用于模型的日志记录、保存模型和配置的管理。

## 参数配置 (Flags)

- **用途**：通过向 `chargpt.py` 添加不同的标志，你可以修改训练时的各种参数配置。
- **示例**：你可以通过命令行标志设置序列长度、查询头和键/值头的数量等。例如：
  - `--data.block_size=128`：设置模型序列长度为128。
  - `--model.rope=True`：启用RoPE嵌入。
  - `--model.n_query_head=6`：设置查询头的数量为6。
  - `--model.n_kv_head=3`：设置键/值头的数量为3。

通过这些起始代码文件，你可以实现并训练一个基于RoPE和GQA的GPT模型，同时测试其在小型数据集上的性能。

## 3 本项目的模型剖析

### 3.1 源数据、数据集的制作

源数据就一个input.txt，里面是莎士比亚《科利奥兰纳斯》戏剧文字内容。内容长这样：

```

CMU_10423 > CMU_10423_2024S > HW1 > input.txt
1 First Citizen:
2 Before we proceed any further, hear me speak.
3
4 All:
5 Speak, speak.
6
7 First Citizen:
8 You are all resolved rather to die than to famish?
9
10 All:
11 Resolved. resolved.
12
13 First Citizen:
14 First, you know Caius Marcius is chief enemy to the people.
15
16 All:
17 We know't, we know't.
18
19 First Citizen:
20 Let us kill him, and we'll have corn at our own price.
21 Is't a verdict?
22
23 All:
24 No more talking on't; let it be done: away, away!
25
26 Second Citizen:
27 One word, good citizens.
28
29 First Citizen:
30 We are accounted poor citizens, the patricians good.
31 What authority surfeits on would relieve us: if they
32 would yield us but the superfluity, while it were
33 wholesome, we might guess they relieved us humanely;
34 but they think we are too dear: the leanness that
35 afflicts us, the object of our misery, is as an
36 inventory to particularise their abundance; our
37 sufferance is a gain to them Let us revenge this with
38 our pikes, ere we become rakes: for the gods know I

```

网上找了翻译大概看了下，如下：

市民甲 在我们继续前进之前，先听我说句话。

众人 说，说。

市民甲 你们都下了决心，宁愿死，不愿挨饿吗？

众人 我们都下了决心了，我们都下了决心了。

市民甲 第一，你们知道卡厄斯·马歇斯是人民的最大公敌。

众人 我们知道，我们知道。

市民甲 让我们杀死他，然后我们要多少谷就有多少谷。我们就这样决定了吗？

众人 不用多说；就这么干。走，走！

市民乙 各位好市民，听我说一句话。市民甲 我们都是苦百姓，贵族才是好市民。那些有权有势的人吃饱了，装不下的东西就可以救济我们。他们只要把吃剩下的东西趁着新鲜的时候赏给我们，我们就会以为他们是出于人道之心来救济我们；可是在他们看来，我们都不值得救济的。我们的痛苦饥寒，我们的枯瘦憔悴，就像是列载着他们的富裕的一张清单；他们享福就是靠了我们受苦。让我们举起我们的武器来复仇，趁我们还没有瘦得只剩几根骨头。天神知道我说这样的话，只是迫于没有面包吃的饥饿，不是因为渴于复仇。

## txt文本概述：

- **数据量**：总计 40000 行文本。
- **字符种类**：65种字符，包括大小写字母、标点符号、数字、空格、换行符。
- **总字符数**：文本中包含 1,115,394 个字符（包括空格和换行符）。

## 数据集制作流程：

### 1. 样本构建：

- 首先，将 txt 文件中的所有文本行首尾相接，形成一个连续的字符序列，总长度为 1,115,394 个字符。
- 在训练过程中，每次从该连续字符序列中，**在随机索引位置**提取 128 个（这是个超参数）连续的字符，作为一个训练样本。

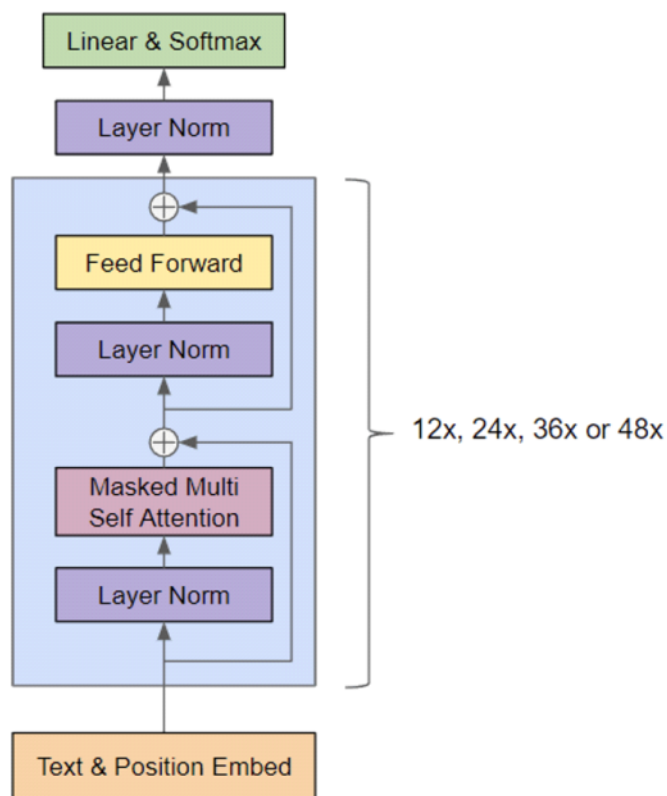
### 2. 标签生成：

- 标签与样本对应，但标签内容相对样本内容往后挪了一位。因为要训练模型基于前面的字符预测下一个字符。

## 3.2 模型结构

模型主体采用了GPT-2的基本结构，包含6层解码器。本项目有两种位置编码方式：

1. 第一种方法是先生成一个 `[0, 1, ..., 127]` 的序列，然后通过 `nn.Embedding()` 进行编码，其结果再与样本的 `nn.Embedding()` 编码结果进行融合，相当于把128个字符的索引位置编码进去了。
2. 第二种方法就是在attention里面应用RoPE位置编码。



GPT-2

模型架构完整参数如下：

Layer (type:depth-idx)	Input Shape	Output Shape	Param #	Param %
GPT	[64, 128]	[64, 128, 65]	--	--
└ModuleDict: 1-1	--	--	--	--
└Embedding: 2-1 词嵌入层	[64, 128]	[64, 128, 192]	12,480	0.46%
└Embedding: 2-2 位置嵌入层	[1, 128]	[1, 128, 192]	24,576	0.90%
└Dropout: 2-3	[64, 128, 192]	[64, 128, 192]	--	--
└ModuleList: 2-4	--	--	--	--
└Block: 3-1	[64, 128, 192]	[64, 128, 192]	444,864	16.36%
└Block: 3-2	[64, 128, 192]	[64, 128, 192]	444,864	16.36%
└Block: 3-3	[64, 128, 192]	[64, 128, 192]	444,864	16.36%
└Block: 3-4	[64, 128, 192]	[64, 128, 192]	444,864	16.36%
└Block: 3-5	[64, 128, 192]	[64, 128, 192]	444,864	16.36%
└Block: 3-6	[64, 128, 192]	[64, 128, 192]	444,864	16.36%
└LayerNorm: 2-5	[64, 128, 192]	[64, 128, 192]	384	0.01%
└Linear: 1-2	[64, 128, 192]	[64, 128, 65]	12,480	0.46%
=====				
Total params: 2,719,104				
Trainable params: 2,719,104				
Non-trainable params: 0				
Total mult-adds (M): 172.47				
=====				
Input size (MB): 0.07				
Forward/backward pass size (MB): 860.09				
Params size (MB): 10.88				
Estimated Total Size (MB): 871.04				
=====				

## 3.3 优化器、学习器、损失函数、超参数等

优化器：adamW

学习率调度器：没用

损失函数：交叉熵

学习率：3e-4

batchsize：64

## 4 Rotary Position Embeddings (RoPE)

### 4.1 讲义原文

在本部分中，你将实现旋转位置嵌入（Rotary Position Embeddings, RoPE）（Su et al., 2021）。

#### 背景

在标准的Transformer语言模型中，绝对位置嵌入（Absolute Position Embeddings）被添加到词嵌入的第一层中。随后的层从底层向上传播位置信息。

传统的注意力机制定义如下：

$$q_j = W_q^T x_j, \forall j$$

$$k_j = W_k^T x_j, \forall j$$

$$s_{t,j} = \frac{k_j^T q_t}{\sqrt{d_k}}, \forall j, t$$



$$a_t = \text{softmax}(s_t), \forall t$$

其中  $d_k$  是查询/键/值向量的大小。

## RoPE机制

旋转位置嵌入（Rotary Position Embeddings, RoPE）（Su et al., 2021）直接将位置信息融入到每一层的注意力计算中。如果下一层注意力机制的输入是  $X = [x_1, \dots, x_N]^T$ ，那么我们引入两个函数  $f_q(x_j, j)$  和  $f_k(x_j, j)$ ，分别计算位置感知的查询和键。然后，注意力得分计算如下：

$$q_j = W_q^T x_j, \forall j$$

$$k_j = W_k^T x_j, \forall j$$

$$\tilde{q}_j = R_{\Theta, j} q_j, \tilde{k}_j = R_{\Theta, j} k_j$$

$$s_{t, j} = \frac{\tilde{k}_j^T \tilde{q}_t}{\sqrt{d_k}}, \forall j, t$$

$$a_t = \text{softmax}(s_t), \forall t$$

其中  $d = \frac{d_k}{2}$ ， $W_k, W_q \in \mathbb{R}^{d_{\text{model}} \times d_k}$ 。对于某个固定的绝对位置  $m$ ，旋转矩阵  $R_{\Theta, m} \in \mathbb{R}^{d_k \times d_k}$  被定义为：

$$R_{\Theta, m} = \begin{pmatrix} \cos(m\theta_1) & -\sin(m\theta_1) & 0 & 0 & \dots & 0 & 0 \\ \sin(m\theta_1) & \cos(m\theta_1) & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & \cos(m\theta_2) & -\sin(m\theta_2) & \dots & 0 & 0 \\ 0 & 0 & \sin(m\theta_2) & \cos(m\theta_2) & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \dots & \cos(m\theta_{d_k/2}) & -\sin(m\theta_{d_k/2}) \\ 0 & 0 & 0 & 0 & \dots & \sin(m\theta_{d_k/2}) & \cos(m\theta_{d_k/2}) \end{pmatrix}$$

参数  $\theta_i$  在训练之前被固定，定义如下：

$$\Theta = \{\theta_i = 10000^{-\frac{2i-1}{d}}, i \in [1, 2, \dots, d/2]\}$$

由于  $R_{\Theta, m}$  中的稀疏结构，我们可以更高效地计算  $R_{\Theta, m}$  与任意向量  $y$  的矩阵-向量乘法：

$$R_{\Theta, m} y = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_{d-1} \\ y_d \end{pmatrix} \otimes \begin{pmatrix} \cos(m\theta_1) \\ \cos(m\theta_1) \\ \cos(m\theta_2) \\ \cos(m\theta_2) \\ \vdots \\ \cos(m\theta_{d/2}) \\ \cos(m\theta_{d/2}) \end{pmatrix} + \begin{pmatrix} -y_2 \\ y_1 \\ -y_4 \\ y_3 \\ \vdots \\ -y_d \\ y_{d-1} \end{pmatrix} \otimes \begin{pmatrix} \sin(m\theta_1) \\ \sin(m\theta_1) \\ \sin(m\theta_2) \\ \sin(m\theta_2) \\ \vdots \\ \sin(m\theta_{d/2}) \\ \sin(m\theta_{d/2}) \end{pmatrix}$$

在PyTorch中高效地实现这一过程仍然需要一些技巧。如果我们有一个嵌入矩阵  $Y = [y_1, \dots, y_N]^T \in \mathbb{R}^{N \times d_k}$ （在实际操作中，这个  $Y$  可能是查询  $Q$  或键  $K$ ），那么我们想要构建一个新的矩阵  $\tilde{Y} = g(Y, \Theta)$ ，其中  $\tilde{Y}_{m, \cdot} = R_{\Theta, m} y_m$ 。为了简化，我们可以将向量的索引重新排列，使得可以分别处理向量的前半部分和后半部分。简记为  $d = d_k$ ：

$$\tilde{Y} = g(Y, \Theta) = \begin{bmatrix} Y_{1,1} & \dots & Y_{1,d/2} & Y_{1,d/2+1} & \dots & Y_{1,d} \\ \vdots & \dots & \vdots & \vdots & \dots & \vdots \\ Y_{N,1} & \dots & Y_{N,d/2} & Y_{N,d/2+1} & \dots & Y_{N,d} \end{bmatrix} \otimes \begin{bmatrix} \cos(1\theta_1) & \dots & \cos(1\theta_{d/2}) \\ \vdots & \dots & \vdots \\ \cos(N\theta_1) & \dots & \cos(N\theta_{d/2}) \end{bmatrix} + \begin{bmatrix} -Y_{1,d/2+1} & \dots & -Y_{1,d} & Y_{1,1} & \dots & Y_{1,d/2} \\ \vdots & \dots & \vdots & \vdots & \dots & \vdots \\ -Y_{N,d/2+1} & \dots & -Y_{N,d} & Y_{N,1} & \dots & Y_{N,d/2} \end{bmatrix} \otimes \begin{bmatrix} \sin(1\theta_1) & \dots & \sin(1\theta_{d/2}) \\ \vdots & \dots & \vdots \\ \sin(N\theta_1) & \dots & \sin(N\theta_{d/2}) \end{bmatrix}$$

或者更简洁地表示为：



$$C = \begin{bmatrix} 1\theta_1 & \dots & 1\theta_{d/2} \\ \vdots & \dots & \vdots \\ N\theta_1 & \dots & N\theta_{d/2} \end{bmatrix}$$

$$\tilde{Y} = g(Y, \Theta) = [Y_{:,1:d/2} \quad Y_{:,d/2+1:d}] \otimes \cos(C) + [-Y_{:,d/2+1:d} \quad Y_{:,1:d/2}] \otimes \sin(C)$$

现在，我们可以高效地计算RoPE嵌入：

$$Q = XW_q, K = XW_k$$

$$\tilde{Q} = g(Q, \Theta), \tilde{K} = g(K, \Theta)$$

$$S = \frac{\tilde{Q} \cdot \tilde{K}^T}{\sqrt{d_k}}$$

$$A = \text{softmax}(S)$$

你不需要理解论文中的所有数学细节，但可以阅读以便对RoPE形成直觉。

**实现：**

你将在 `minGPT` 中实现RoPE。为此，你需要在 `mingpt/model.py` 文件中修改 `RotaryPositionalEmbeddings` 和 `CausalSelfAttention` 类。

## 4.2 RoPE的代码实现

```
class RotaryPositionalEmbeddings(nn.Module):
    """
    实现RoPE（旋转位置嵌入）。该模块对输入的查询和键应用旋转位置嵌入，主要通过构建旋转角度的
    cos 和 sin 矩阵来完成。
    """

    def __init__(self, d: int, base: int = 10_000):
        """
        初始化RoPE模块。

        参数:
        d (int): 输入的嵌入维度。
        base (int): 用于生成旋转角度的基数，默认为10_000。
        """
        super().__init__()
        self.d = d # 嵌入维度 (例如, d_model 或 d_query)
        self.base = base # 基数, 控制旋转角度的幅度
        self.cosine_mat = None # 用于缓存 cos 矩阵
        self.sine_mat = None # 用于缓存 sin 矩阵

    def _build_cache(self, x: torch.Tensor):
        """
```

计算和缓存 cos 和 sin 矩阵，这些矩阵用于对输入进行旋转嵌入。

此步骤根据输入的维度（序列长度和嵌入维度）动态生成。

参数：

x (torch.Tensor): 输入张量，包含批次大小、头数、序列长度和嵌入维度。

"""

device = x.device # 获取输入的设备（例如 CPU 或 GPU）

self.N = x.shape[-2] # 获取序列长度 N

# 生成 theta 作为旋转角度，它基于嵌入维度的前半

# positions 是嵌入维度的一半（即  $d // 2$ ）个位置，从 0 到  $(d // 2 - 1)$

positions = torch.arange(0, self.d // 2, device=device)

# theta 是旋转频率的参数，随着位置增大而衰减

theta = torch.pow(self.base, -2 \*\* (positions) / self.d)

# 生成矩阵 c\_matrix，其尺寸为 (N,  $d // 2$ )，用于储存每个序列位置的旋转频率

# arange\_N 是一个从 1 到 N 的向量，它表示每个位置（从 1 开始）

arange\_N = torch.arange(1, self.N + 1, device=device).unsqueeze(1)

# c\_matrix 是每个位置乘以 theta，用于生成 cos 和 sin 值

c\_matrix = arange\_N \* theta.unsqueeze(0)

# 将 c\_matrix 复制两份，形成 (N, d) 维度的矩阵，其中前半部分和后半部分相同

# 生成对应的 cos 和 sin 矩阵，并扩展维度为 (1, 1, N, d)，以便广播操作

self.cosine\_mat = torch.cos(torch.cat([c\_matrix, c\_matrix],  
dim=-1)).unsqueeze(0).unsqueeze(0)

self.sine\_mat = torch.sin(torch.cat([c\_matrix, c\_matrix],  
dim=-1)).unsqueeze(0).unsqueeze(0)

def forward(self, x: torch.Tensor):

"""

前向传播：对输入张量 x 进行旋转位置嵌入。

参数：

x (torch.Tensor): 输入张量，形状为 (batch\_size, n\_heads, seq\_len, d)。

返回：

torch.Tensor: 应用旋转位置嵌入后的张量。

"""

# 检查是否需要重新生成 cos 和 sin 矩阵（基于输入的序列长度和嵌入维度）

if (self.cosine\_mat is None or x.shape[-2] != self.cosine\_mat.shape[-2] or  
x.shape[-1] != self.cosine\_mat.shape[-1]):

self.\_build\_cache(x) # 如果缓存的矩阵尺寸不匹配，则重新构建

# 获取嵌入维度的一半，便于将张量分成两部分

half\_d = self.d // 2

```

# 计算 x 的 cos 部分：每个位置的值乘以相应的 cos 值
x_cos = x * self.cosine_mat

# 计算 x 的 sin 部分：
# 将 x 的后半部分移到前面，前半部分移到后面，并乘以相应的 sin 值
x_sin = torch.cat([-x[:, :, :, half_d:], x[:, :, :, :half_d]], dim=3) *
self.sine_mat

# 返回两个部分的和，作为旋转嵌入后的输出
return x_cos + x_sin

```

## 4.3 RoPE问题

### 1 (4 分)

绘制你实现的 RoPE 和原始 minGPT 模型在序列长度为 128 时，训练 600 次迭代的训练损失图。

### 2 (4 分)

绘制你实现的 RoPE 和原始 minGPT 模型在总计 800 次训练迭代中的训练损失图：前 600 次迭代使用序列长度 128，接下来的 200 次迭代使用序列长度 256。

### 3 (2 分)

提供一个来自你训练了 600 次迭代（序列长度为 128）的 RoPE 模型的样本。样本的生成应基于你最喜欢的莎士比亚剧本中的第一行作为条件。

### 4 (2 分)

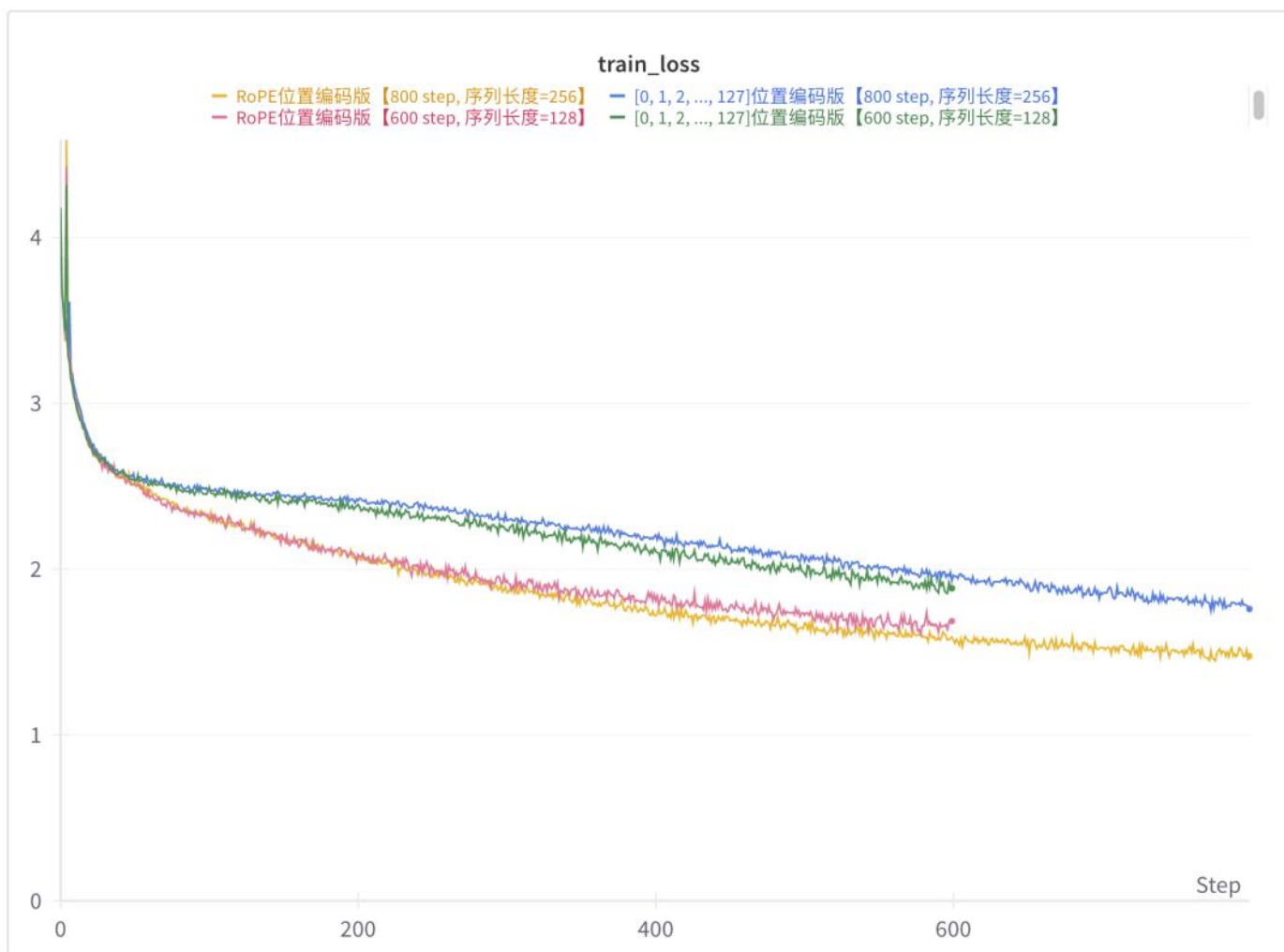
提供一个来自你训练了 600 次迭代（序列长度为 128），然后又训练了 200 次迭代（序列长度为 256）的 RoPE 模型的样本。样本的生成应基于你最喜欢的莎士比亚剧本中的第一行作为条件。

**解答：**

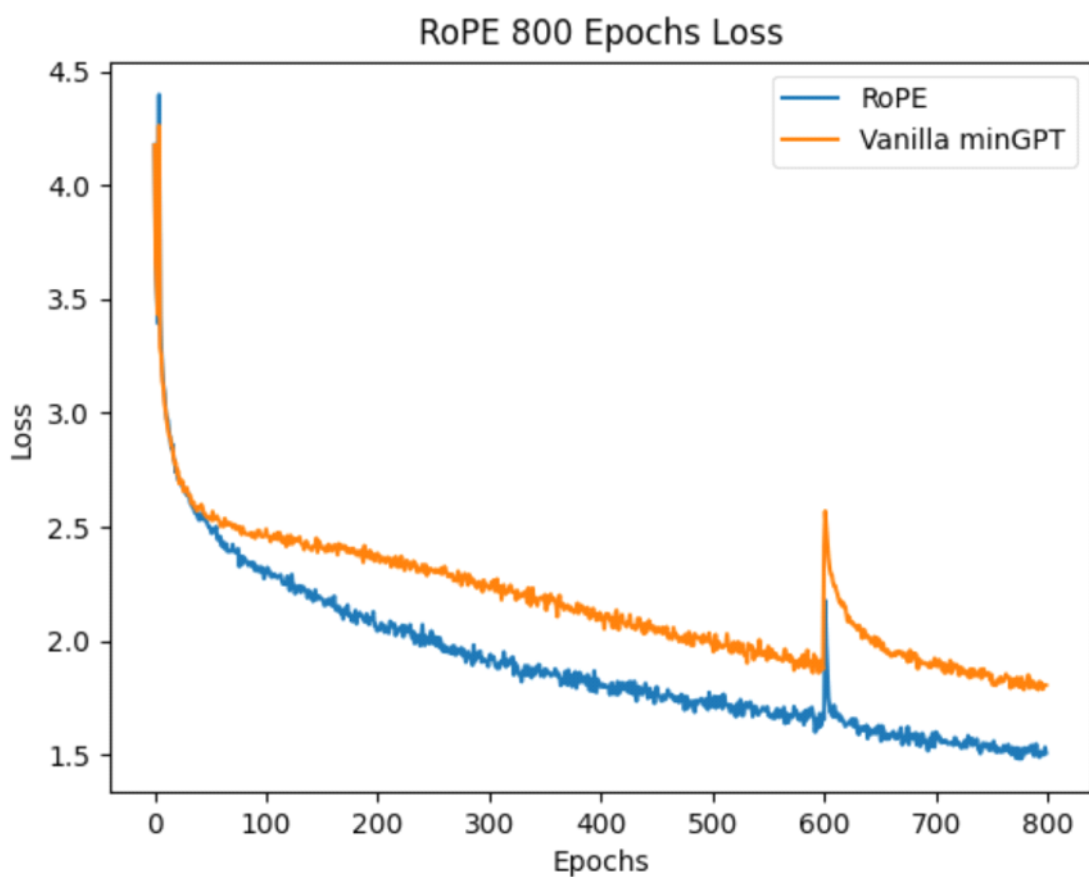
**以下分别是训练600个step（序列长度=128）和800个step（序列长度=256）情况：**

从下图可看出2点：

- RoPE位置嵌入远好于绝对位置嵌入。
- 使用绝对位置嵌入，当序列长度增大时，同样step情况下loss会变大。而如果使用RoPE位置嵌入，当序列长度增大时，训练时的loss会更小。



训练600个step后，切换序列长度的训练结果可以直接参考CMU学生的文件：



## 文本生成情况展示：

我同时测试了把训练集《科利奥兰纳斯》的第一句话作为输入，以及《罗密欧与朱丽叶》第一句话作为输入，评估模型生成文本的效果。

以下2张图是《罗密欧与朱丽叶》原文：

ACT I	
ACT I	
SCENE I. Verona. A Public Place.	
Enter SAMPSON and GREGORY, armed with swords and bucklers.	GREGORY To move is to stir; and to be valiant is to stand: therefore, if thou art moved, thou runnest away.
SAMPSON Gregory, o'my word, we'll not carry coals.	SAMPSON A dog of that house shall move me to stand: I will take the wall of any man or maid of Montague's.
GREGORY No, for then we should be colliers.	GREGORY That shows thee a weak slave; for the weakest goes to the wall.
SAMPSON I mean, as we be in choler, we'll draw.	SAMPSON 'Tis true; and therefore women, being the weaker vessels, are ever thrust to the wall: therefore I will push Montague's men from the wall, and thrust his maids to the wall.
GREGORY Ay, while you live, draw your neck out o'the collar.	GREGORY The quarrel is between our masters and us their men.
SAMPSON I strike quickly, being moved.	SAMPSON 'Tis all one, I will show myself a tyrant: when I have fought with the men, I will be cruel with the maids, and cut off their heads.
GREGORY But thou art not quickly moved to strike.	
SAMPSON A dog of the house of Montague moves me.	

第一幕	
第一幕	
第一场 维罗纳广场	
山普孙及葛莱古里各持盾剑上。	汉。
山普孙 葛莱古里，咱们可真的不能让人家当作苦力一样欺侮。	山普孙 我见了他们家里的狗子，就会站住不动；蒙太古家的人，不论男女，碰到了我就像碰到墙壁一样。
葛莱古里 对了，咱们不是可以随便给人欺侮的。	葛莱古里 这正说明你是个不中用的家伙；只有不中用的家伙，才会躲到墙底。
山普孙 我说，咱们要是发起脾气来，就会拔刀子动武。	山普孙 不错；所以没用的女人，就老是被人逼得不能动：我见了蒙太古家里人来，是男人，我就把他们从墙边推出去；是女人，我就把她们朝着墙壁摔过去。
葛莱古里 对了，你可不要把脖子缩进领口里去。	葛莱古里 吵架是咱们两家主仆男人们的事，与她们女人有什么相干？
山普孙 我一动性子，我的剑是不认人的。	山普孙 那我不管，我要做一个杀人不眨眼的魔王；一面跟男人们打架，一面对娘儿们也不留情面，我要割掉她们的头。
葛莱古里 可是你不大容易动性子。	葛莱古里 割掉娘儿们的头吗？
山普孙 我见了蒙太古家的狗子就生气。	山普孙 对了，娘儿们的头，或是她们的奶头，你爱怎么说就怎
葛莱古里 有胆量的，生了气就应当站住不动；逃跑的不是好	

600step+RoPE编码的模型的预测结果如下：



<p>Before we proceed any further, hear me speak.</p> <p>Petives:</p> <p>There bear and the from well that see his countak Shall him by disher; and the may best thy sea; The well and musts thou dries in his heaven, foe; What I mone to day'd and and treeng many Tit his suppetion. You wife a marry foe: Who see you, sir. Have I have him hand of thou love, To some your mate my be folloon forewith was I do a deather in toudious, and this were with you.</p> <p>AUTOLYCA:</p> <p>And, arm about henger and honours: and ambutions, Beaings the for our forler may head, thou had 注意力计算耗时 0.78ms, 序列长度为 128 Gregory, o'my word, we'll not carry coals. Stime, honour fare many tell shall not a faith, We looks and his worth's daughter's than what her?</p> <p>GLOUCESTER:</p> <p>Is must of hence talk, swords of hath wall, And prayand the harmatt baw the copparliess Hell, the wiss weak of were a man show, Thou damant thiu sto hope, the bring of.</p> <p>KING RICHARD III:</p> <p>Shall I the sever in auther's bishoud Supfully wouly sighty with soves, Or then mest stress, while, my broth, therefore!</p> <p>ANGELO:</p> <p>Shall not he, need's with to make thee hath my mine: The dreservest t 注意力计算耗时 0.78ms, 序列长度为 128</p>	<p>在我们继续之前, 听我说。</p> <p>Petives:</p> <p>那里的熊和井里的人看见他的脸, 就用盘子;愿你的大海最美;你在他的天堂里驾井, 驾船;我今天所做的一切都是他的猜想。你的妻子是一个结婚的敌人:谁看见你了, 先生。我得到了你的爱的手, 对你的某个伴侣, 我将永远追随, 我将死在你身边。</p> <p>奥托利卡:</p> <p>你有充足的荣耀和兵器, 为我们的先人作预备</p> <p>The attention calculation took 0.78ms and the sequence length was 128</p> <p>格里高利, 我保证, 我们不会搬煤的。有时, 荣誉会告诉我们, 我们的容貌和他的女儿的价值比她的什么?</p> <p>格洛斯特:</p> <p>他必须这样说话, 刀剑有墙, 祈祷和伤害有议会, 地狱, 智慧的弱点是一个人的显示, 你诅咒你的希望, 带来的。</p> <p>理查三世:我是该在作家的教堂里祈祷, 祈祷上帝保佑我的眼睛, 还是我的压力, 而我的肉汤, 因此!</p> <p>天使:</p> <p>难道他不需要和你一起占有我的吗? 最保守的 The attention calculation took 0.78ms and the sequence length was 128</p>
--	---

## GPT4的评估

根据你提供的第一句“Gregory, o'my word, we'll not carry coals.”以及后续生成的文本, 我可以从以下几个方面评估大模型的文本生成效果:

### 1. 语言风格的匹配性 (权重: 40%) :

- 这句话来自莎士比亚的作品, 因此具有古英语的语法和风格。大模型生成的文本尝试模仿这种古英语的形式, 比如使用了词汇如“thou”, “hath”, “shall”等。
- 然而, 整体生成的句子语法显得相当混乱, 句子结构也不太连贯, 难以真正展现莎士比亚那种优雅的风格。语言的流畅性和一致性有待提高。

### • 语义连贯性 (权重: 30%) :

- 文本中的句子大多是独立的, 缺乏上下文的连贯性。例如“Shall him by disher; and the may best thy sea;”这类句子含义模糊, 且难以理解。语义上, 文本没有明确传递某个主题或逻辑进展。
- 生成的内容尽管模仿了古英语的形式, 但没有形成一个有意义的段落, 语义比较混乱。

### • 创造力 (权重: 20%) :

- 尽管生成的文本不连贯, 但模型展现了一定的创造力, 生成了许多词汇组合和结构。比如“Tit his suppetion”和“deather in toudious”, 尽管没有明显的意义, 但能看出模型在尝试创造新词和表达。
- 这种创造力在文本生成中是很重要的, 但它需要在连贯的框架下展现出来。

### • 模型对特定任务的适应性 (权重: 10%) :

- 如果这是一个模仿莎士比亚风格的任务，模型生成的文本还未能完全达到目标。虽然句子表面上看起来具有古英语的风格，但它缺乏莎士比亚作品中的逻辑、语义和深度。

总结：

大模型在语言风格上做了一些尝试，使用了莎士比亚时代的词汇和表达，但生成的文本缺乏语义上的连贯性，句子结构也比较混乱。要提升效果，模型可能需要更好地理解上下文，并生成更符合人类阅读习惯的连贯文本。

以下是训练800个step情况（序列长度=256，使用RoPE）的生成结果：

```
Before we proceed any further, hear me speak.

COMINIUS:
But I'll dead young of chooly, that yet our horls,
And not of, that they since to tay.
What I warrant or time the word, her beggar such
and praise are at thy hopest from and throw men,
Which scorn'd to a distress and so came, so-making
sweet hath sleign all, that there and being to straight
safe be and day? there that, a was, all with her;
Or three his poor against thy guard of help, all had
sure, men thumbate man from help-pack of ploses,
To so-prayed against the comparly, but they
注意力计算耗时 0.72ms, 序列长度为 256
Gregory, o'my word, we'll not carry coals.

KINCE EDWARD IV:
Why, where is with the wear of once to there,
By man of a cars in sorrow our lustice,
I to the gone old man him of a stay,
Sweet was on their wench, or seent oftracial.

BRALTANUS:
But we did, for the blood he is necking fine:
Would not her soon of crumph; or that is he doour see:
I wive the here setters is to a crown;
And stoop innoce to the subrovings their friend?
What, master death frown that art in thouse
He thou serve hangs, and impring too.

PETRUCHIO:
Awake this heart!
```

## GPT4的评估2

### 1. 语言风格的匹配性（权重：40%）：

- **本次评估：**本次生成的文本继续模仿莎士比亚式的古英语风格，使用了词汇如“hath”、“thou”、“art”等。然而，与第一次相比，仍然有很多词汇组合显得不自然，且句子结构比较混乱。例如“sweet hath sleign all, that there and being to straight”这样的句子，语法不正确，句子的逻辑也不连贯。
- **对比第一次：**与第一次生成相比，语言风格的模仿没有显著提升，模型依然局限于表面形式的仿古英语词汇堆砌，而无法真正模仿莎士比亚句子结构的优雅和严谨。

### 2. 语义连贯性（权重：30%）：

- **本次评估：**与第一次相比，生成的文本仍然缺乏语义连贯性。比如句子“her beggar such and praise are at thy hopest from and throw men”显得无意义，单词之间的关联很弱，整体句子的逻辑性和可读性较差。并且许多句子是独立生成的，无法形成完整的情节或上下文。
- **对比第一次：**本次生成的文本在语义连贯性上与第一次的表现相似，没有明显的改进。句子依旧是单独生成的，未能形成连贯的对话或叙述。



### 3. 创造力（权重：20%）：

- **本次评估：**模型在生成文本时展现了创造性，继续生成了一些词汇和表达，虽然部分词汇组合不符合语法，但体现了一些语言的灵活性。例如“to so-prayed against the comparly”虽然没有意义，但模型在创造新的短语和组合上表现了一定的创造力。
- **对比第一次：**与第一次相比，模型的创造力水平保持相近，生成了一些新的组合。不过这些创造性的表达并未带来实际的语义提升，仍然是“词汇实验”。

### 4. 模型对任务的适应性（权重：10%）：

- **本次评估：**如果目标是模仿莎士比亚风格，模型本次生成的文本与第一次一样，仍然远未达到目标。尽管使用了古英语词汇，但句子逻辑和意义仍不连贯，无法形成有意义的段落。
- **对比第一次：**与第一次相比，本次生成的文本在适应任务方面没有太大改进，依然主要表现在表面上的风格模仿，而不是语义上的理解和生成。

### 综合对比：

- **语言风格：**两次生成的风格模仿相似，都使用了古英语词汇和句式，但缺乏句法的正确性和句子之间的逻辑关联。
- **语义连贯性：**两次生成的文本语义上都缺乏连贯性，句子之间没有明确的主题或关联。
- **创造力：**两次生成的文本都有一定的创造性，但这种创造性主要体现在无意义的词汇组合上，没有在语义上带来提升。
- **适应性：**两次生成的文本都没有很好地完成模仿莎士比亚风格的任务。

### 总结：

总体而言，本次生成的效果与第一次生成相似，在语言风格和创造性上都有一定的表现，但在语义连贯性和适应性上仍有很大不足。两次生成的文本在莎士比亚风格的模仿上都仅停留在表面，未能深入到句子的逻辑结构和语义层面。

## 5 Grouped Query Attention (GQA)

### 5.1 讲义原文

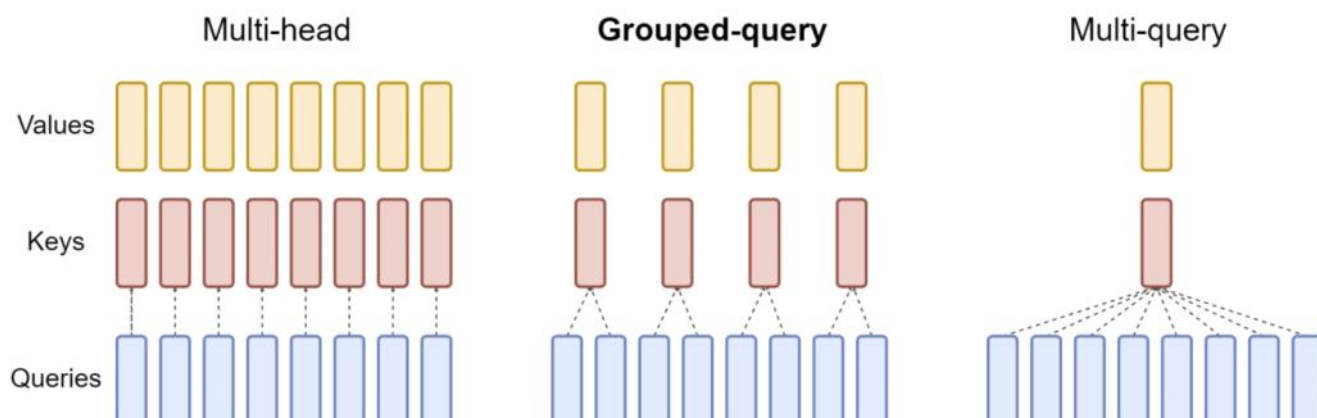


图 1：注意力机制的示意图，展示了具有每个头单独键和值的多头注意力（Multi-head attention），具有分组查

询共享公共键和值的分组查询注意力（Grouped-query attention），以及使用单一键和值的多查询注意力（Multi-query attention）。

在本部分中，你将实现分组查询注意力（Grouped Query Attention, GQA）（Ainslie 等, 2023）。

## GQA简介

分组查询注意力（GQA）是一种用于神经网络架构的技术，它修改了模型（如Transformer）中使用的注意力机制。该机制将查询头划分为多个组，每组共享一个键和值头。这种方法可以在多查询注意力（MQA）和多头注意力（MHA）之间进行插值，提供了计算效率与模型质量之间的平衡[图 1]。

令  $h_q$  表示查询头的数量， $h_{kv}$  表示键/值头的数量。我们假设  $h_q$  可以被  $h_{kv}$  整除，并且  $g = \frac{h_q}{h_{kv}}$  是每组的大小（即每个键/值向量对应的查询向量数量）。

我们对 GQA 的参数矩阵大小保持一致： $W_q^{(g,i)}, W_k^{(g)}, W_v^{(g)} \in \mathbb{R}^{d_{\text{model}} \times d_k}$ ，其中  $d_k = \frac{d_{\text{model}}}{h_q}$ 。然而，现在我们有不同数量的查询、键和值头：

$$\begin{aligned} X &= [x_1, \dots, x_T]^T \\ V^{(i)} &= XW_v^{(i)}, \forall i \in \{1, \dots, d_{kv}\} \\ K^{(i)} &= XW_k^{(i)}, \forall i \in \{1, \dots, d_{kv}\} \\ Q^{(i,j)} &= XW_q^{(i,j)}, \forall i \in \{1, \dots, d_{kv}\}, \forall j \in \{1, \dots, g\} \end{aligned}$$

上面，我们定义了比键/值向量多  $g$  倍的查询向量。然后，我们计算每个查询向量  $(i,j)$  与其对应键  $(i)$  的缩放点积，并在每组内对查询求和，以得到相似度得分。使用这些相似度得分计算注意力矩阵，但只使用  $h_{kv}$  个头：

$$\begin{aligned} S^{(i)} &= \sum_{j=1}^g Q^{(i,j)} (K^{(i)})^T / \sqrt{d_k} \\ A^{(i)} &= \text{softmax}(S^{(i)}) \\ X'^{(i)} &= A^{(i)} V^{(i)} \end{aligned}$$

## 实现细节：

你将在 `mingpt/model.py` 文件中的 `GroupedQueryAttention` 类中实现 GQA。你编写的代码大部分会与 `CausalSelfAttention` 类中的内容相似。

**提示：**你可能会发现，首先使用 `einops.rearrange()` 重新实现 `CausalSelfAttention` 类比使用 `tensor.view()` 或 `tensor.transpose()` 更加容易；此外，使用 `einops.einsum()` 替代 `@` 操作符。如果你在这种形式下扩展实现 `GroupedQueryAttention`，可能会更直接。

- **初始化：**
  - 熟悉初始化注意力机制的配置设置，包括查询头、键/值头的数量和嵌入维度。
  - 确保嵌入维度可以被查询头和键/值头的数量整除。
- **正则化：**
  - 添加注意力和残差的 dropout 层，以防止过拟合。
- **维度和投影：**

- 实现查询、键和值的线性投影层，考虑维度约束和机制的分组性质。
- **旋转位置嵌入：**
  - 如果启用了旋转位置嵌入（RoPE），将 RoPE 集成到查询和键的投影中。（注意：将 RoPE 和 GQA 结合是可选的，但实现相对简单。）
- **前向传播：**
  - 在 `forward` 方法中，按照查询、键和值的投影变换输入。
  - 通过计算分组缩放点积注意力来应用注意力机制。
  - 对注意力进行掩码操作，以确保因果性（即避免未来的 tokens 被关注）。
  - 将注意力与值进行聚合，并将输出投影回嵌入维度。
- **内存效率：**
  - 在注意力操作前后，监控并记录 CUDA 内存分配，以分析 GQA 的内存效率。在 `CausalSelfAttention` 类中有用于监控内存的参考代码。

## 5.2 GQA的代码实现

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class GroupedQueryAttention(nn.Module):
    """
    实现分组查询注意力机制。
    """

    def __init__(self, config):
        super().__init__()

        # 确保嵌入维度能够整除查询头和键/值头的数量
        assert config.n_embd % config.n_query_head == 0
        assert config.n_query_head % config.n_kv_head == 0

        # 分组数量g（每个键/值头对应多少个查询头）
        self.g = config.n_query_head // config.n_kv_head

        # 键和值的线性投影层，生成键和值的向量，大小为 2 * n_embd
        self.vk_attn = nn.Linear(config.n_embd, 2 * config.n_embd)

        # 查询的线性投影层，生成查询的向量，维度为 g * n_embd
        self.q_attn = nn.Linear(config.n_embd, self.g * config.n_embd)
```

```

# 输出的线性投影层，用于将注意力输出映射回嵌入维度
self.c_proj = nn.Linear(config.n_embd, config.n_embd)

# 正则化，防止过拟合
self.attn_dropout = nn.Dropout(config.attn_pdrop)
self.resid_dropout = nn.Dropout(config.resid_pdrop)

# 因果遮罩，确保注意力仅关注到当前位置及之前的序列
self.register_buffer("bias", torch.tril(torch.ones(config.block_size,
config.block_size))
                                .view(1, 1, config.block_size, config.block_size))

# 存储键/值头数量和查询头数量
self.n_head = config.n_kv_head
self.n_query = config.n_query_head
self.n_embd = config.n_embd

# 是否使用RoPE（旋转位置嵌入）
self.rope = config.rope
if self.rope:
    # 如果使用RoPE，则初始化RoPE模块，维度为每个头的嵌入维度
    self.custom_rope = RotaryPositionalEmbeddings(self.n_embd // self.n_head)

def forward(self, x):
    """
    前向传播，计算分组查询注意力输出。

    参数:
        x (torch.Tensor): 输入张量，形状为 (batch, seq_len, n_embd)。

    返回:
        torch.Tensor: 注意力输出，形状为 (batch, seq_len, n_embd)。
        int: 显存消耗。
    """
    # 获取输入的批次大小 (B)、序列长度 (T) 和嵌入维度 (C)
    B, T, C = x.size()

    # 通过线性层生成键和值，将嵌入维度分为两部分
    v, k = self.vk_attn(x).split(self.n_embd, dim=2)

    # 通过查询的线性层生成查询
    q = self.q_attn(x)

    # 将键和值 reshaped 为多头形式，维度为 (B, nh, T, d)，其中 nh 表示头的数量，d 表示每个头的维度
    k = k.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T,

```

```

d)
    v = v.view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T,
d)

    # 将查询 reshaped 为多头和分组形式, 维度为 (B, nh, g, T, d), g 表示查询的分组
    q = q.view(B, T, self.n_head, self.g, C // self.n_head).transpose(1,
2).transpose(2, 3) # (B, nh, g, T, d)

    # 如果启用了 RoPE, 则将旋转位置嵌入应用于查询和键
    if self.rope:
        q = self.custom_rope(q)
        k = self.custom_rope(k)

    # 清除缓存, 确保准确计算显存消耗
    torch.cuda.empty_cache()
    start_memory = torch.cuda.memory_allocated() # 获取开始时的显存消耗

    # 使用 einsum 计算注意力分数, 维度为 (B, nh, g, T, T)
    att = torch.einsum('ijklm,ijmn->ijln', q, k.transpose(-2, -1)) * (1.0 /
math.sqrt(k.size(-1)))

    # 应用因果遮罩, 确保注意力只关注到当前位置及之前的序列
    att = att.masked_fill(self.bias[:, :, :T, :T] == 0, float('-inf'))

    # 使用 softmax 计算注意力权重, 并应用 dropout 进行正则化
    att = F.softmax(att, dim=-1)
    att = self.attn_dropout(att)

    # 计算注意力输出, 将注意力权重与值进行点积, 得到 (B, nh, T, d)
    y = att @ v

    # 获取结束时的显存消耗
    end_memory = torch.cuda.memory_allocated()

    # 重塑注意力输出, 将多头输出拼接, 返回 (B, T, C)
    y = y.transpose(1, 2).contiguous().view(B, T, C)

    # 输出线性变换, 并应用残差 dropout 进行正则化
    y = self.resid_dropout(self.c_proj(y))

    # 返回注意力输出和显存消耗差值
    return y, end_memory - start_memory

```

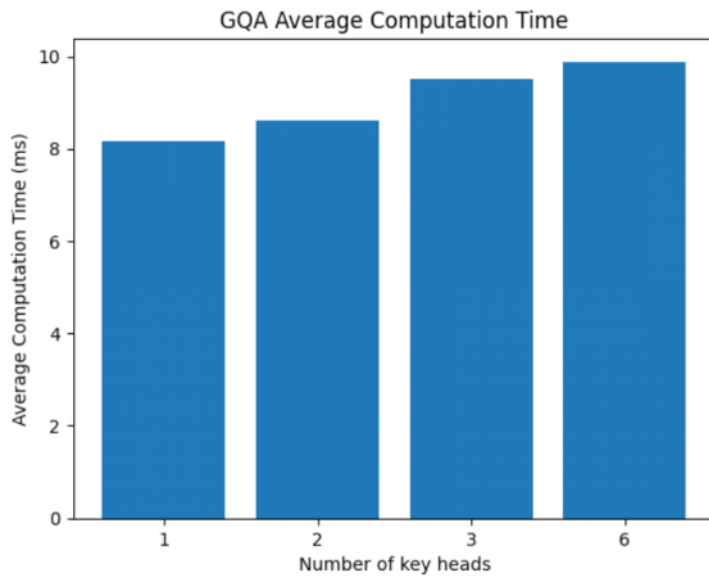
## 5.3 GQA问题

备注: 下文5,6题我懒得去修改代码了, 直接引用的CMU学生跑的结果图, 第7问是我自己跑的结果图。

以下问题假设你使用的是绝对位置嵌入，而不是 RoPE。

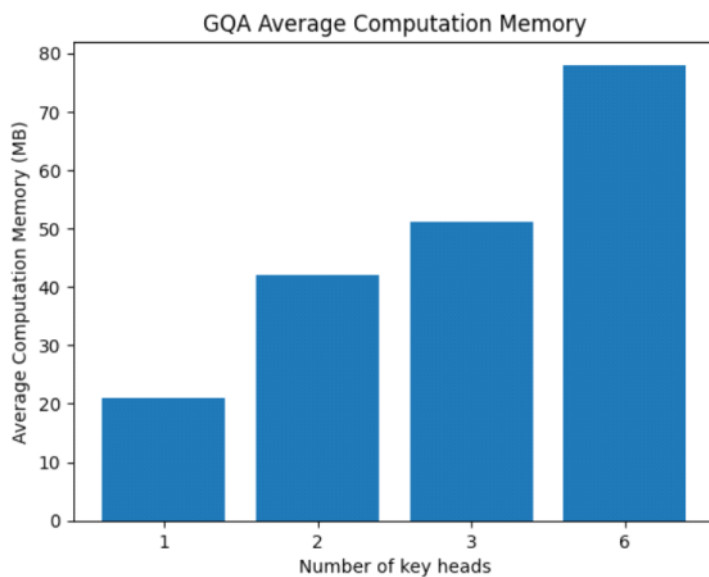
## 5 (4分)

绘制每次迭代中计算注意力所需的平均时间（毫秒）在不同K头数量 {1, 2, 3, 6} 下的变化。



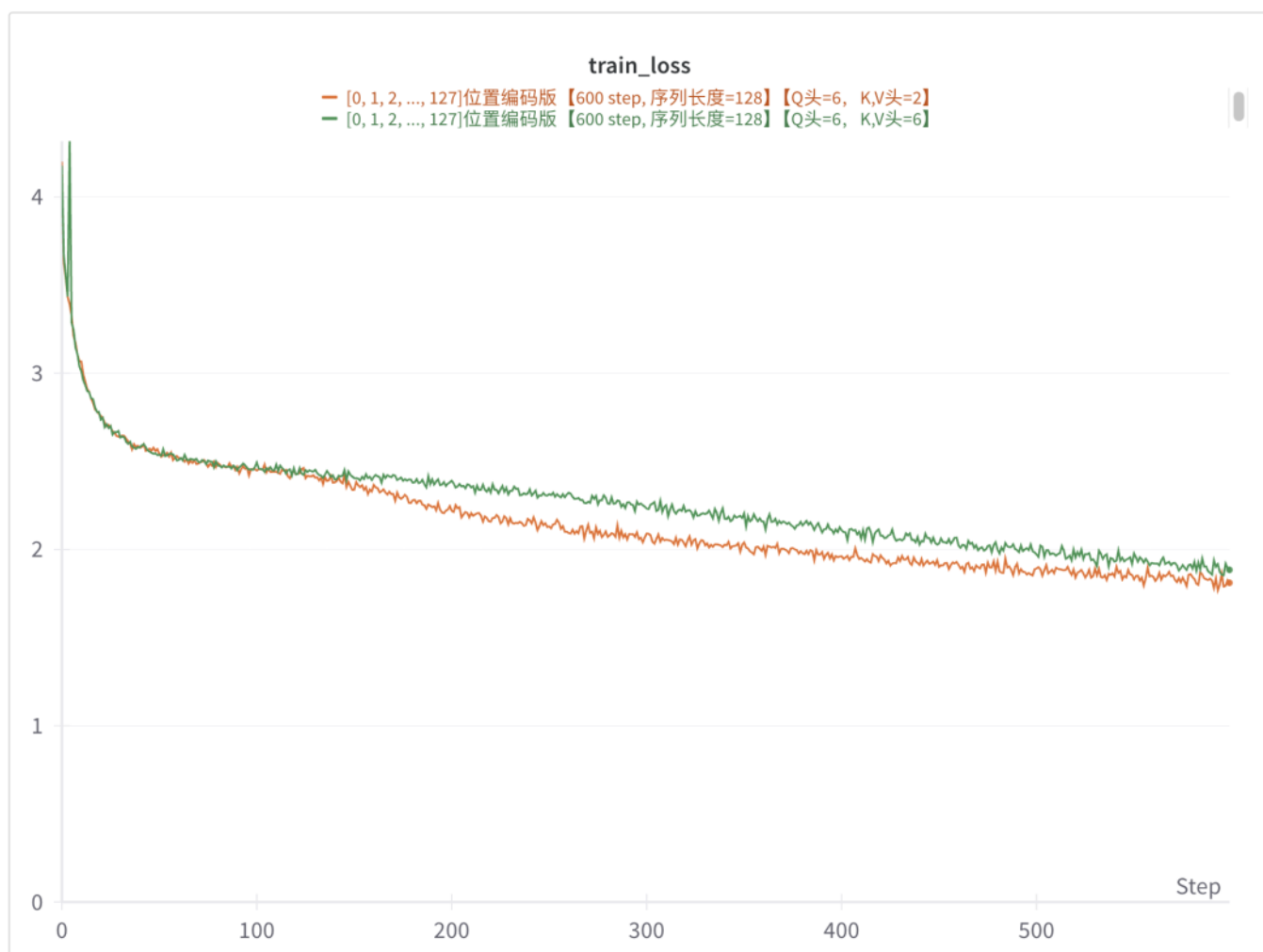
## 6 (4分)

绘制每次迭代的内存消耗（MB）在不同K头数量 {1, 2, 3, 6} 下的变化。



## 7 (4分)

绘制你实现的 GQA 模型在 2 个K头下与原始（多头注意力）minGPT 在序列长度为 128 时训练 600 次迭代的训练损失图。



## 5.4 关于GQA跑实验的结论

从5.3可看出，使用QGA，降低K头数量，虽然计算时间降得很少，但是显存消耗降得非常多，并且训练速度或精度竟然还能提升！