哈尔滨工业大学

**自然语言处理**

**实验报告**

|  |  |
| --- | --- |
| **姓名：** | **陈铠** |
| **学号：** | **2023112419** |
| **学院：** | **未来技术学院** |
| **日期：** | **2025年10月16日** |
| **教师：** | **车万翔** |

实验一：从零开始实现基于Transformer的语言模型

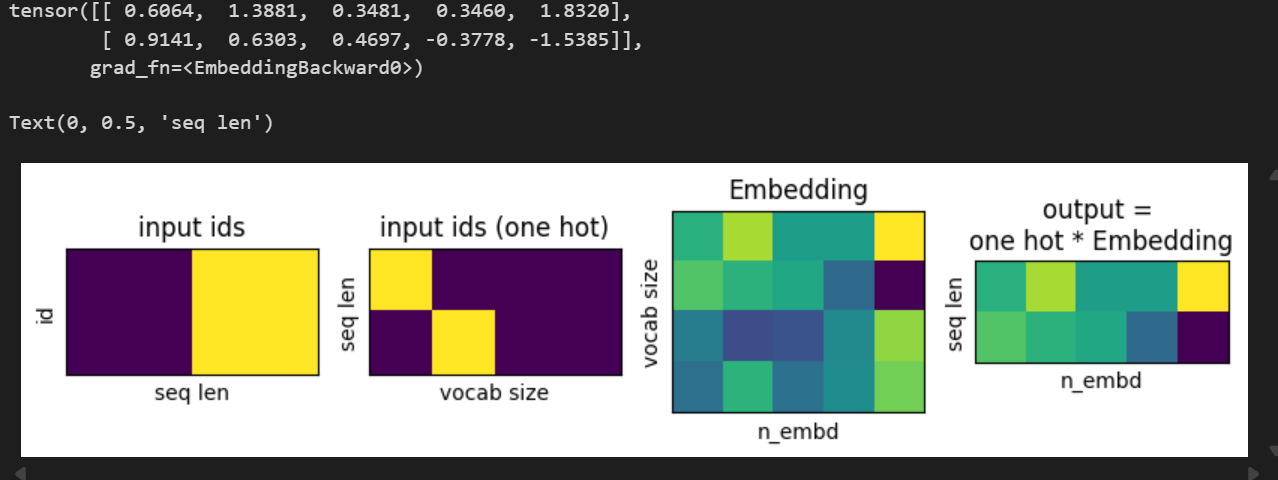
## 一、实验过程及结果

（一）Token Embedding

Token Embedding的作用是将输入的token ID转换为高维向量表示。具体实现方法是通过nn.Embedding实现，该层会将每个token ID映射到一个固定维度的向量。。

wte = nn.Embedding(vocab\_size, n\_embd)

output = wte(input\_ids) # shape = [T, n\_embd]

input\_id为输入的token ID序列，wte为词向量映射层，输出为对应的词向量。

（二）Positional Encoding

位置编码在Transformer模型中注入位置信息，使得模型能够处理顺序信息。GPT使用的是可学习的位置编码，通过nn.Embedding层来实现。

wpe = nn.Embedding(seq\_len, n\_embd)

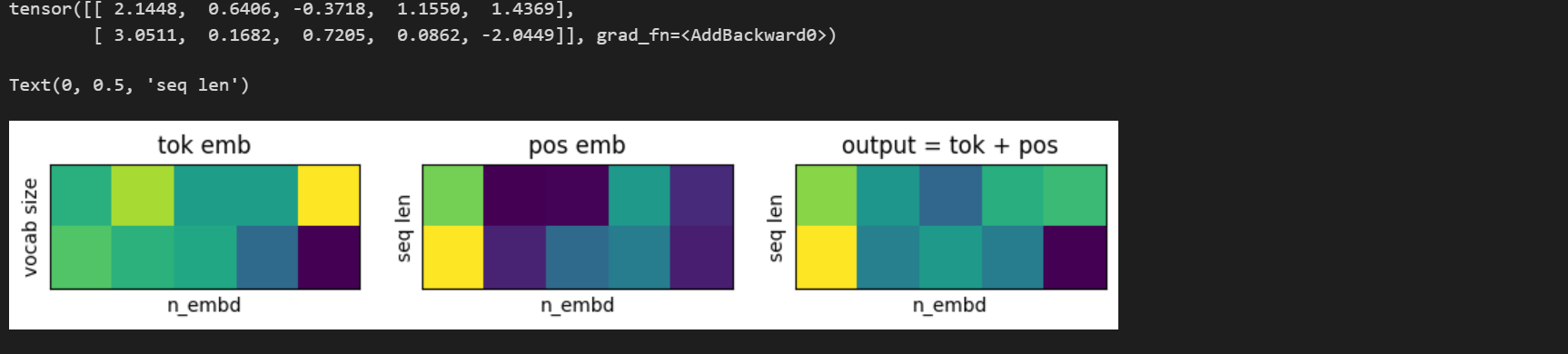
pos = torch.arange(0, seq\_len, dtype=torch.long)

tok\_emb = wte(input\_ids)  # shape = [T, n\_embd]

pos\_emb = wpe(pos)  # shape = [T, n\_embd]

output = tok\_emb + pos\_emb

pos\_emb是位置编码向量，将其与token的词向量tok\_emb相加，得到每个token的最终表示。



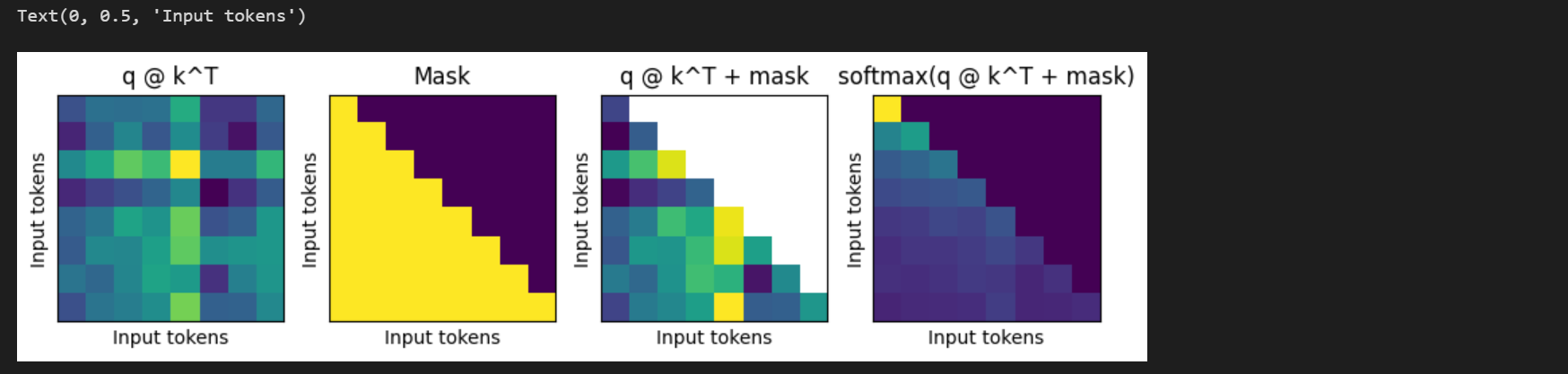
（三）简易Attention Mask

在训练时，为了确保模型的自注意力机制只能关注前面的token，需要创建一个遮蔽矩阵。使用torch.tril可以生成一个上三角矩阵，用负无穷填充未来位置，从而实现自回归的效果。

mask = torch.tril(torch.ones(seq\_len, seq\_len, device=q.device)).bool()

att = att.masked\_fill(~mask, float("-inf"))

通过torch.tril生成一个三角矩阵，然后用masked\_fill将未来的tokens遮蔽掉。



（四）多头自注意力

多头自注意力（Multi-Head Attention）可以在多个子空间中并行捕捉信息。首先，通过线性变换获得Q、K、V矩阵，然后计算自注意力分数。接着使用softmax进行归一化，再与V矩阵相乘获得最终输出。

self.head\_dim = config.n\_embd // config.n\_head

self.c\_attn = nn.Linear(config.n\_embd, 3 \* config.n\_embd,bias=False)

self.c\_proj = nn.Linear(config.n\_embd, config.n\_embd, bias=False)

将隐层维度n\_embd均分为多个较小的空间，每个空间捕捉不同类型的依赖关系。c\_attn从输入向量中同时生成Query（Q）、Key（K）和Value（V）矩阵。c\_proj将多个头的输出拼接后，恢复到原始的n\_embd维度。

qkv = self.c\_attn(x)

q, k, v = qkv.split(self.n\_embd, dim=2)

q = q.view(B, T, self.n\_head, self.head\_dim).transpose(1, 2)

k = k.view(B, T, self.n\_head, self.head\_dim).transpose(1, 2)

v = v.view(B, T, self.n\_head, self.head\_dim).transpose(1, 2)

q, k, v通过.view()被重新组织成形状[batch\_size, n\_head, seq\_len, head\_dim]。 然后，使用.transpose(1, 2)交换n\_head和seq\_len维度，从而使每个头的计算可以并行进行。

att = (q @ k.transpose(-2, -1)) \* (1.0 / math.sqrt(self.head\_dim))

mask = torch.tril(torch.ones(T, T, device=x.device)).view(1, 1, T, T).bool()

att = att.masked\_fill(~mask, float("-inf"))

att = F.softmax(att, dim=-1)

y = att @ v

att计算Q和K的点积，得到注意力分数，再依次应用掩码、softmax。然后计算最终的注意力输出。

（五）FFN层

前馈网络（Feed Forward Network，FFN）通过两层全连接层处理自注意力的输出。中间使用GELU激活函数增加非线性。

self.c\_fc = nn.Linear(config.n\_embd, 4 \* config.n\_embd)

self.act = nn.GELU()

self.c\_proj = nn.Linear(4 \* config.n\_embd, config.n\_embd)

FFN的第一层扩展维度，有助于模型增加表示能力。FFN的第二层将高维的特征映射回n\_embd维度，以便与后续的层对接。

（六）Decoder Block

Decoder Block由两个主要部分组成：多头自注意力和前馈网络FFN，并且每一部分都包含LayerNorm层来做归一化，同时具有残差连接。

self.ln1 = nn.LayerNorm(config.n\_embd)

self.attn = MultiHeadSelfAttention(config)

self.ln2 = nn.LayerNorm(config.n\_embd)

self.ffn = FFN(config)

（七）多头自注意力

将各个组件（Embedding、Positional Encoding、Decoder Block）组合在一起，构建完整的GPT模型，从而进行语言生成任务。

wte = nn.Embedding(config.vocab\_size, config.n\_embd),

wpe = nn.Embedding(config.block\_size, config.n\_embd),

drop = nn.Dropout(config.dropout),

blocks =

nn.ModuleList([DecoderBlock(config) for \_ in range(config.n\_layer)]),

ln\_f = nn.LayerNorm(config.n\_embd),

计算过程：

tok\_emb = self.transformer.wte(idx)

pos = torch.arange(0, T, dtype=torch.long, device=device)

pos\_emb = self.transformer.wpe(pos)[None, :, :]

x = tok\_emb + pos\_emb

x = self.transformer.drop(x)

for block in self.transformer.blocks:

    x = block(x)

x = self.transformer.ln\_f(x)

logits = self.lm\_head(x)

**结果：**

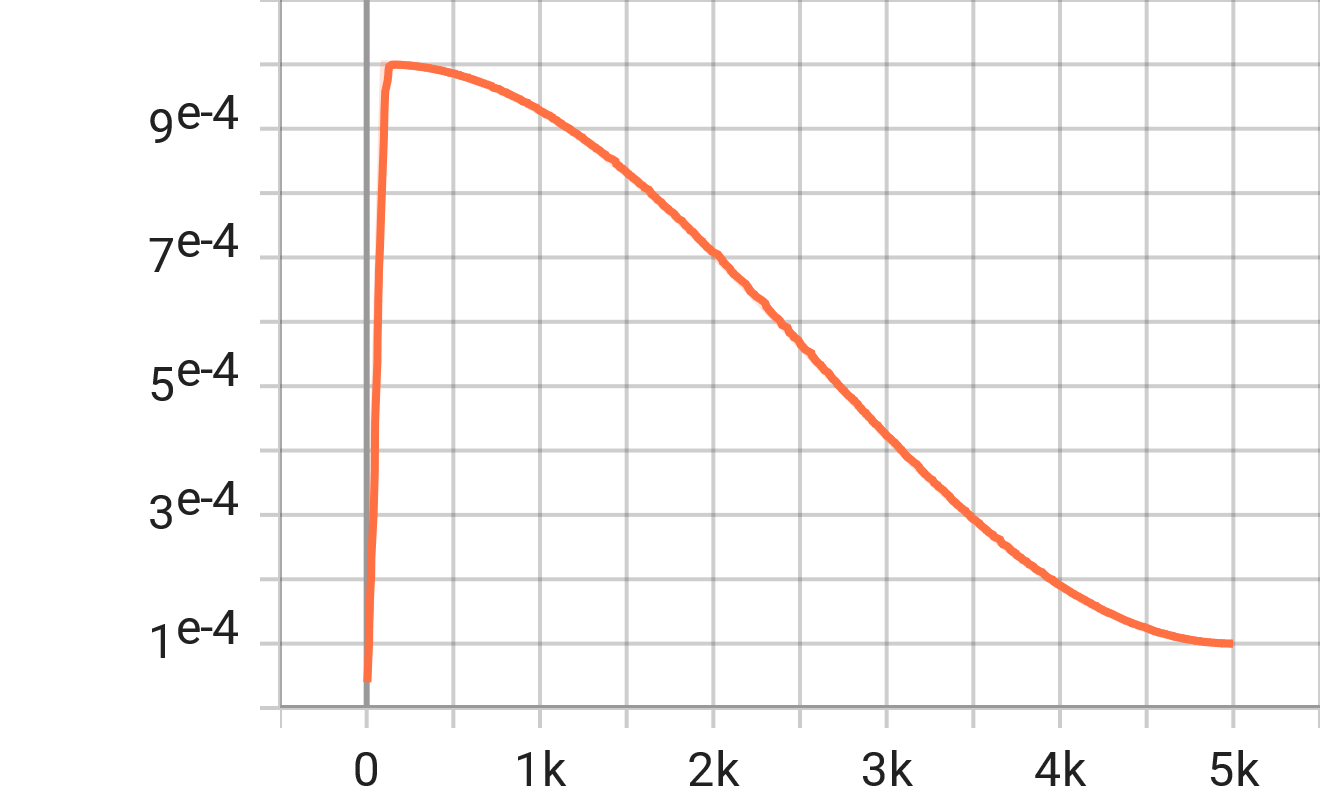


图 1 lr变化图

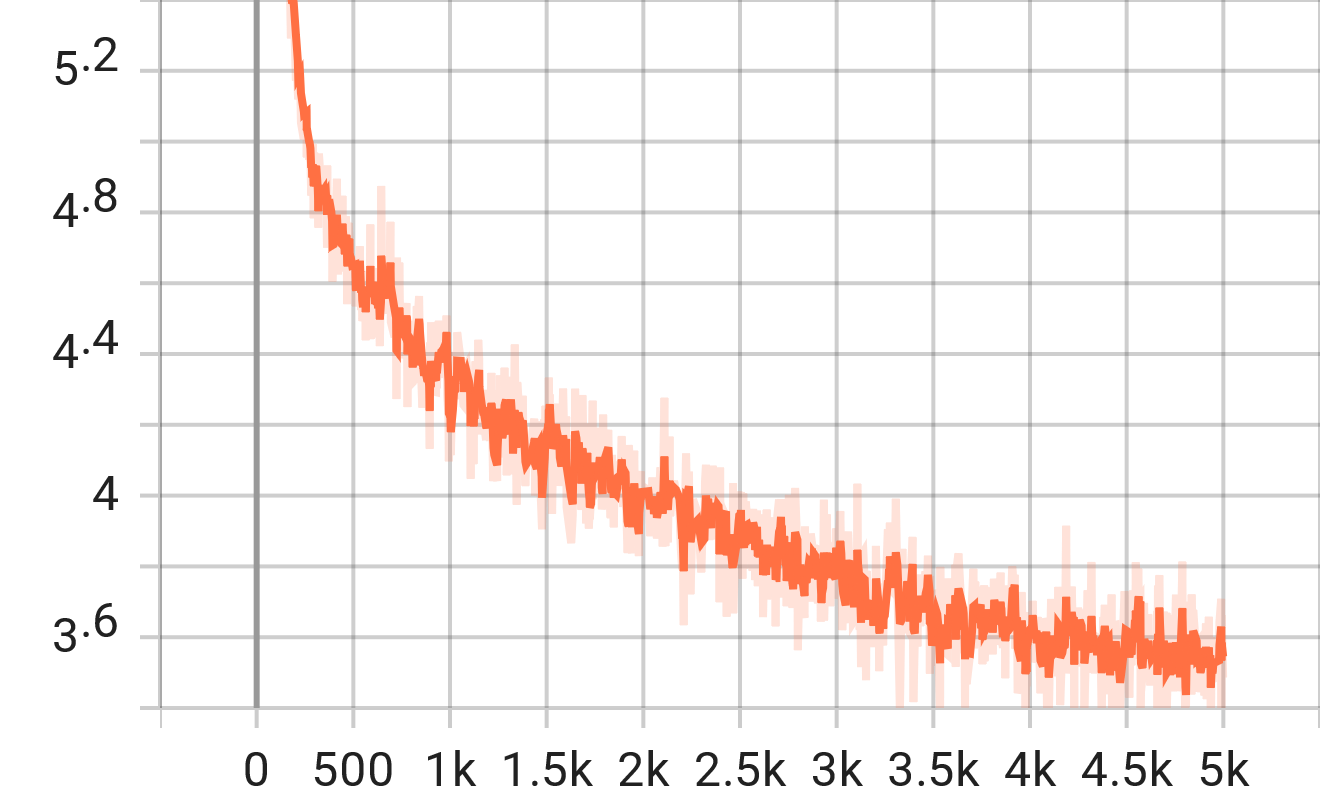


图 2 train loss 变化图

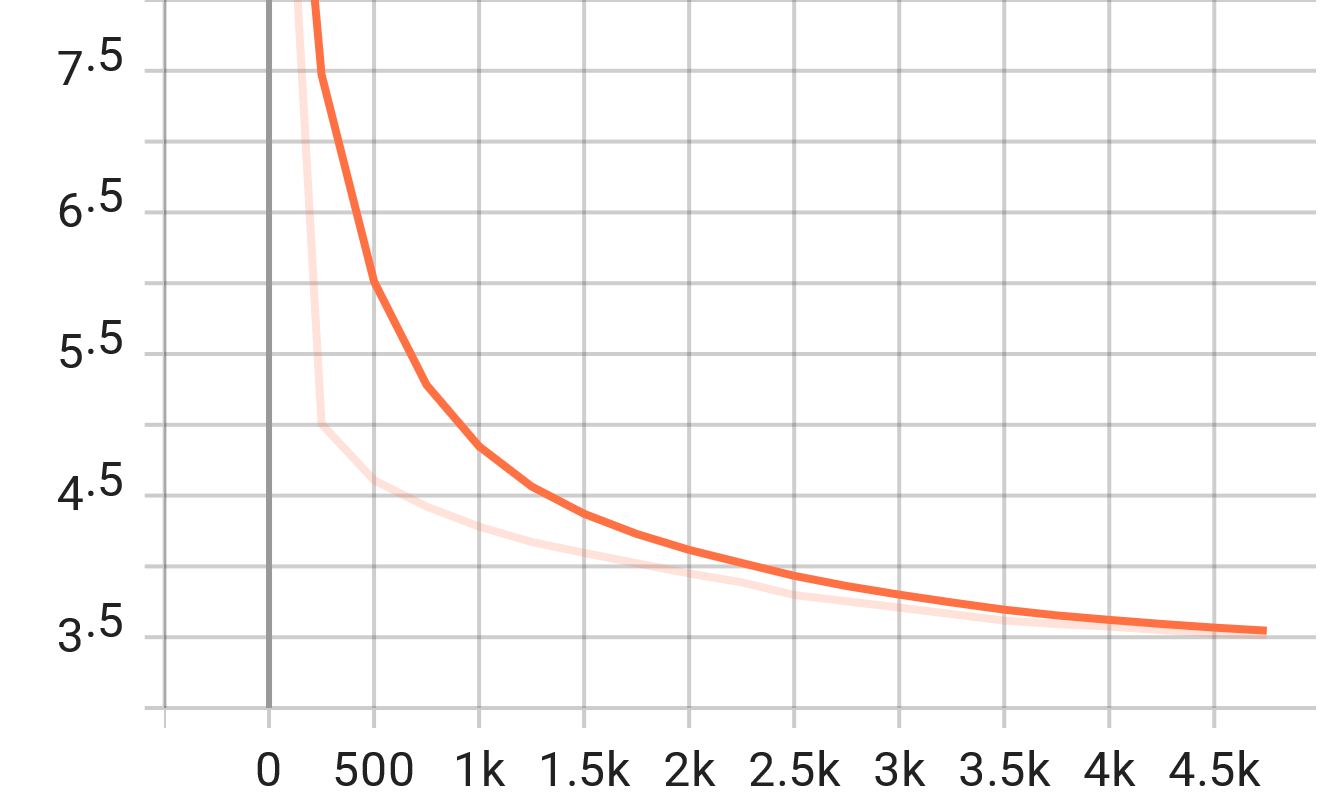


图 3 val loss 变化图

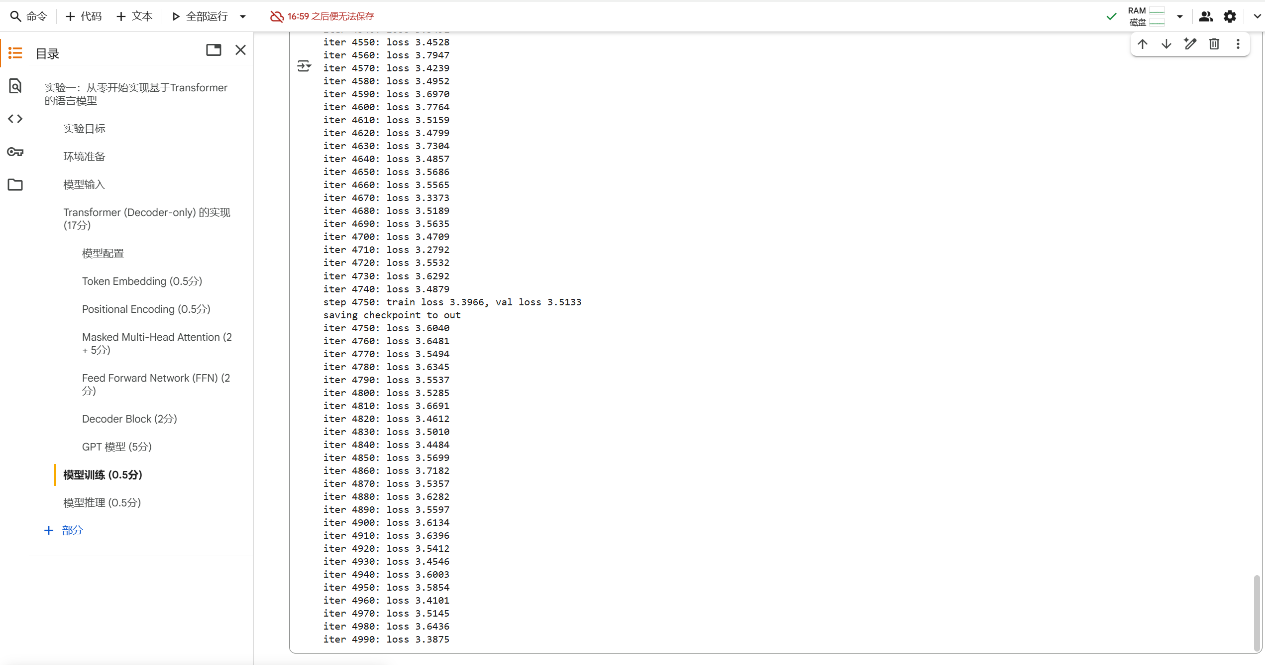
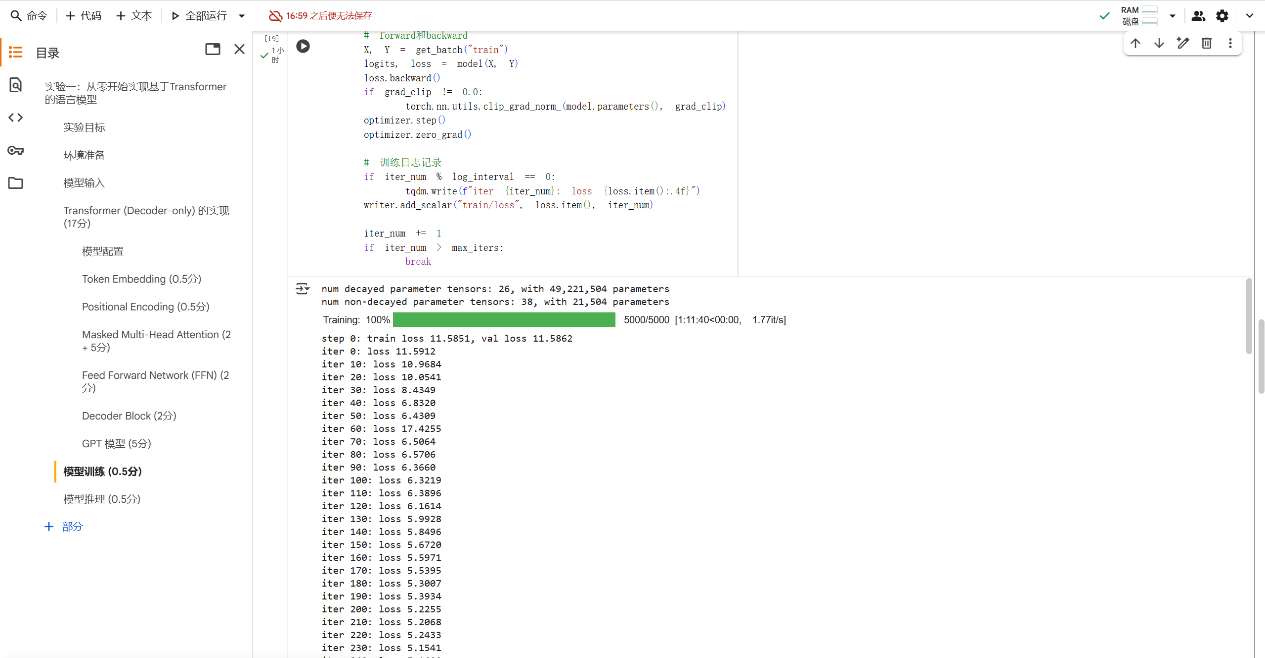


图 4 训练过程图

最终loss在3.3-3.7左右



图 5 输出图

## 二、实验心得

通过本实验，成功实现了一个基于Transformer架构的GPT模型。通过逐步完成每个组件（Token Embedding、Positional Encoding、多头自注意力、FFN层等），对Transformer的工作原理有了更深入的理解。

**遇到的困难**

在训练过程中，发现模型生成的文本是有一定的逻辑性，但还存在一些瑕疵，主要是因为模型的规模较小，生成的内容可能缺乏连贯性。

1.模型训练时间长：由于模型规模较大，训练时间较长。解决方式是适当调整模型参数，缩小batch size，使用更强的硬件加速。

2.生成文本质量不高：模型规模较小，导致生成的文本缺乏逻辑。可以通过增加模型层数或隐藏单元数来提高生成质量。

|  |  |
| --- | --- |
| 实验成绩： |  |
| 教师签名： |  |