​**小白从0开始手动搭建6层transformer模型全记录**

**一、 任务目标**

  手动搭建6层transformer模型，训练模型，能够使模型实现中文到英文的翻译功能，最终评测模型性能。

**二、 准备工作**

1. 学习台大李宏毅教授自注意力机制和Transformer详解视频，见YouTube或b站

2. 学习理解经典论文 《Attention is all you need》

3. 查找他人手动搭建transformer模型的经验贴，自己动手跑一下他人实现好的项目。主要观察其代码工程应该分为哪些部分，数据集如何构建，模型参数如何设置等。

**三、 数据集**

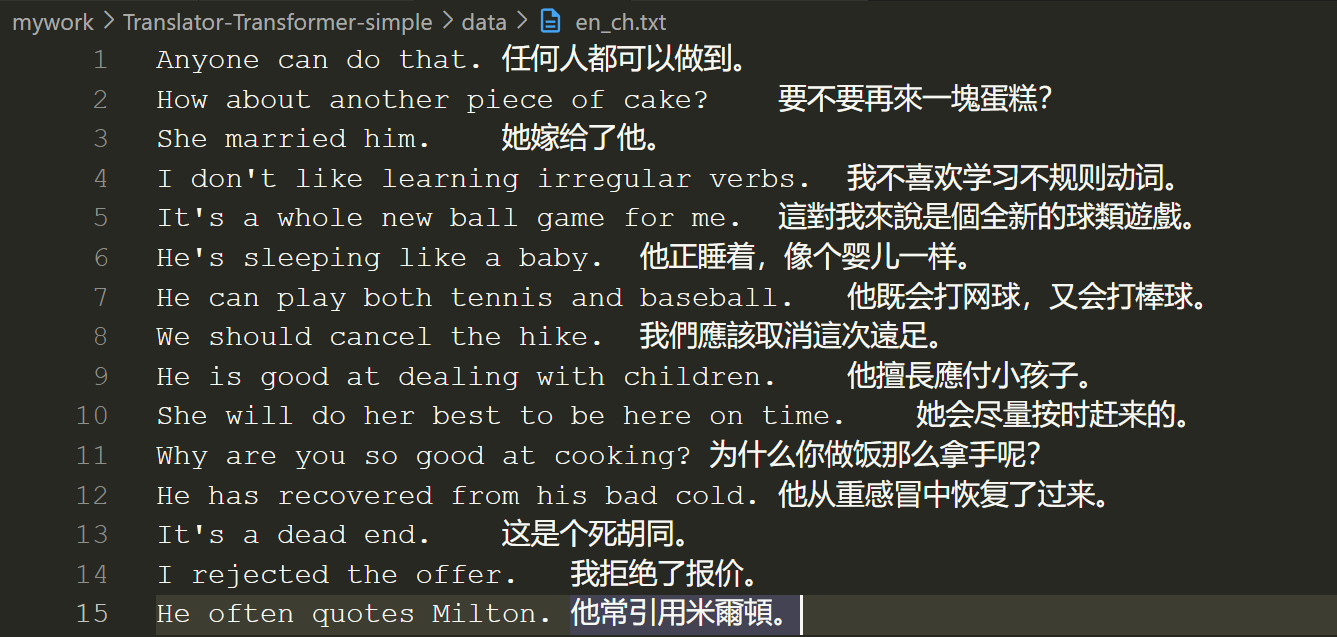
本项目数据来源如下：

|  |  |  |
| --- | --- | --- |
| 数据集名称 | 类型 | 来源 |
| en\_ch.txt | 中英短句子对 | *https://github.com/cuicaihao/Annotated-Transformer-English-to-Chinese-Translator* |
| en\_vocab.txt | 英文词汇表 | 根据en\_ch.txt与test.txt手动构建 |
| ch\_vocab.txt | 中文词汇表 | 根据en\_ch.txt与test.txt手动构建 |

下面举例说明上述数据集内容格式：

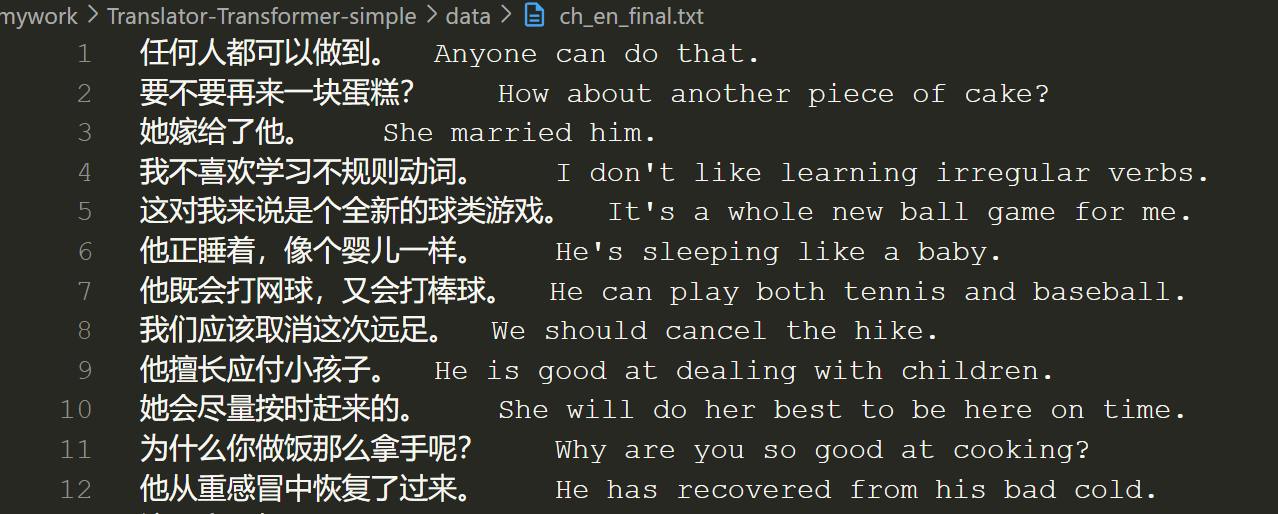
**1.en\_ch.txt**

类似如下[英文 繁体中文]格式，共 14533行。



对上述数据集使用**preprocess.py**进行预处理，使用opencc库将繁体中文转换为简体中文格式，又因为本任务方便输入是将中文转换为英文，所以调整为[简体中文 英文]格式，得到ch\_en\_final.txt文件。

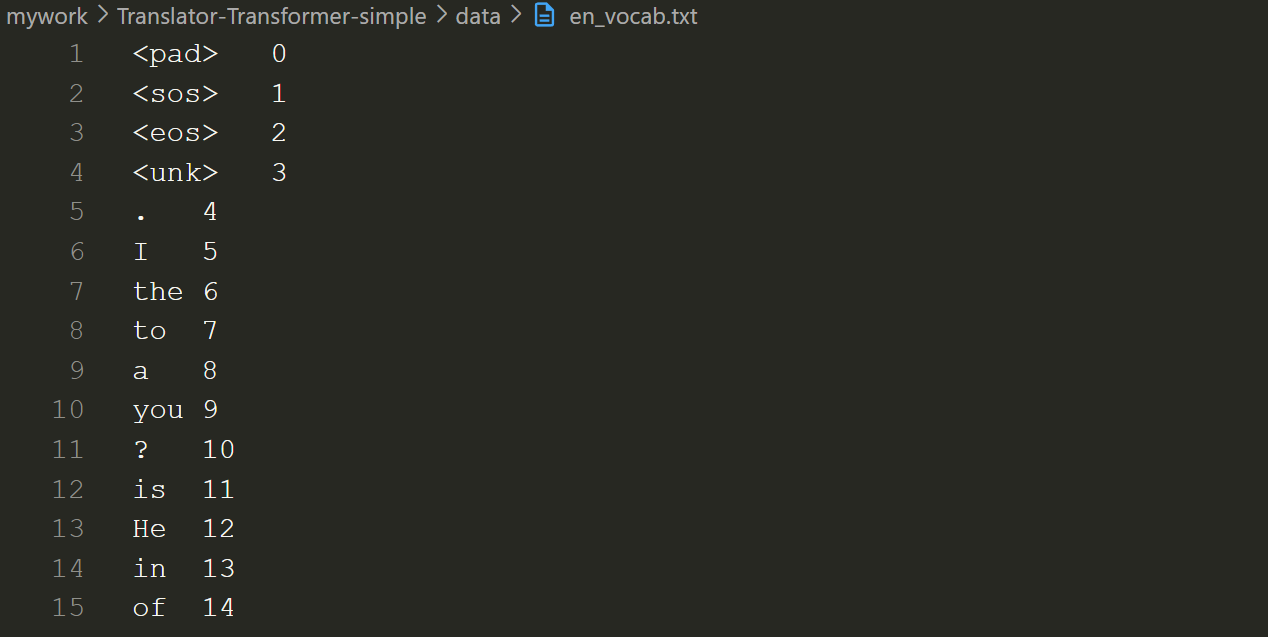
具体格式如下：



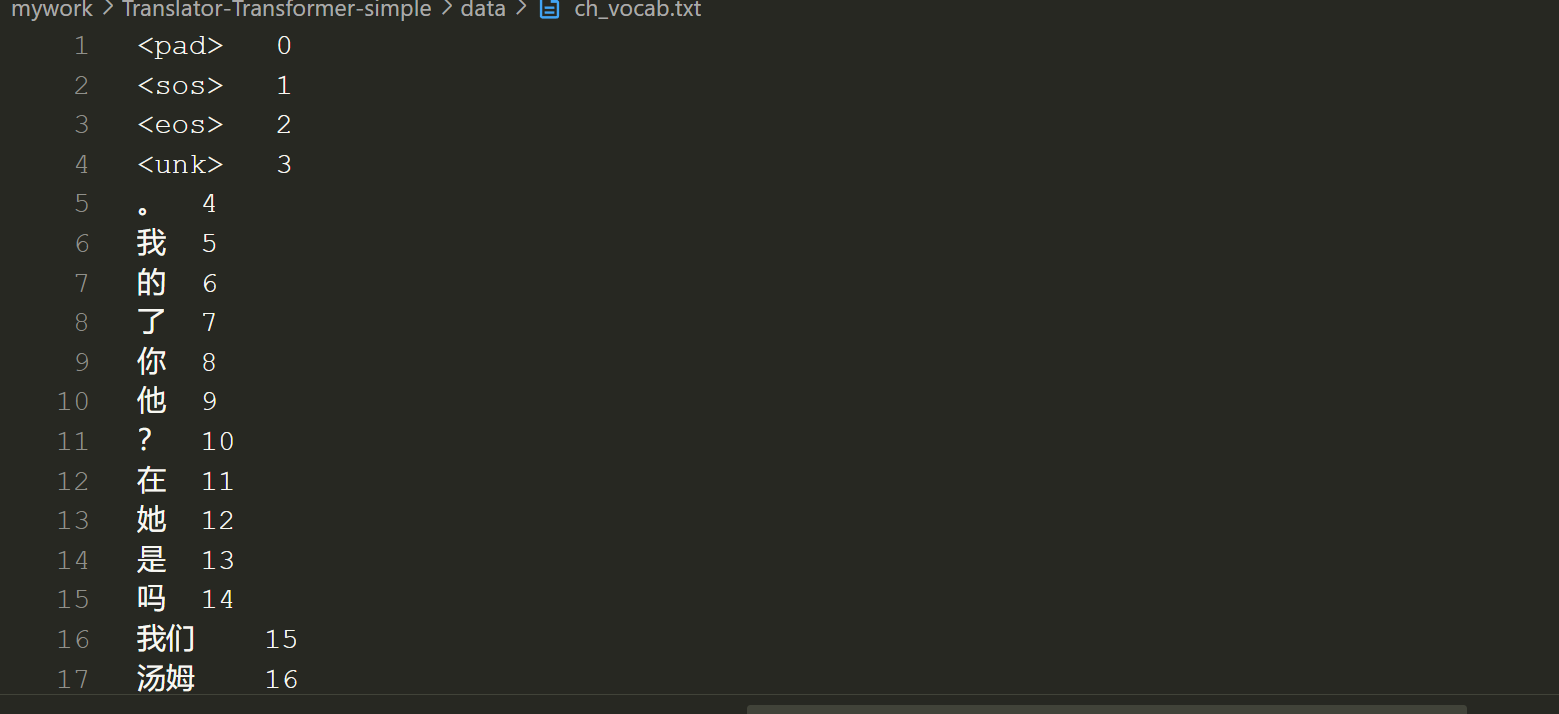
**2. en\_vocab/ch\_vocab**

分别使用jieba库和空格+标点符号正则表达式对中文和英文进行分词，然后统计频次，按照频次大小排序，频次大的token映射对在前。

最终，en\_vocab格式如下：



ch\_vocab格式如下：



**3. 说明**

关于中英文词汇表特殊符号：

|  |
| --- |
| <pad> 0 #作填充用  <sos> 1 #标志句子开始  <eos> 2 #标志句子结束  <unk> 3 #未知词（词汇表中没有的词） |

**四、 Transformer模型的手动搭建**

具体实现见本人CSDN博客：

*https://blog.csdn.net/m0\_67192720/article/details/143477113?spm=1001.2014.3001.5501*

**五、 transformer模型的训练**

**1. 模型相关参数设置**

|  |
| --- |
| ch\_vocab = load\_vocab(ch\_vocab\_file)  en\_vocab = load\_vocab(en\_vocab\_file)  src\_vocab\_size = len(ch\_vocab) # embedding操作的参数  trg\_vocab\_size = len(en\_vocab) # embedding操作的参数  embed\_size = 512 # 常用256、512...  num\_layers = 6 # transformer模型层数  forward\_expansion = 4 # 扩展隐藏层维度，前馈全连接网络的参数  heads = 8 # 512可以被8整除，512/8=64  dropout = 0.1 # 随机化，防止过拟合  max\_length = 15 # 最大序列长度  num\_epochs = 50 #训练轮次，可视loss变化情况而定  learning\_rate = 3e-4 # 学习率  *batch\_size*=128 #每批样本数量为128个    # 优化器Adam 优化算法来更新模型的参数  optimizer = optim.Adam(model.parameters(), *lr*=learning\_rate)    # 学习率调度器，监测loss在无明显变化时减少学习率  scheduler = ReduceLROnPlateau(optimizer, *mode*='min', *factor*=0.5, *patience*=5, *verbose*=True)    # 损失函数，交叉熵损失  criterion = nn.CrossEntropyLoss(*ignore\_index*=trg\_pad\_idx) |

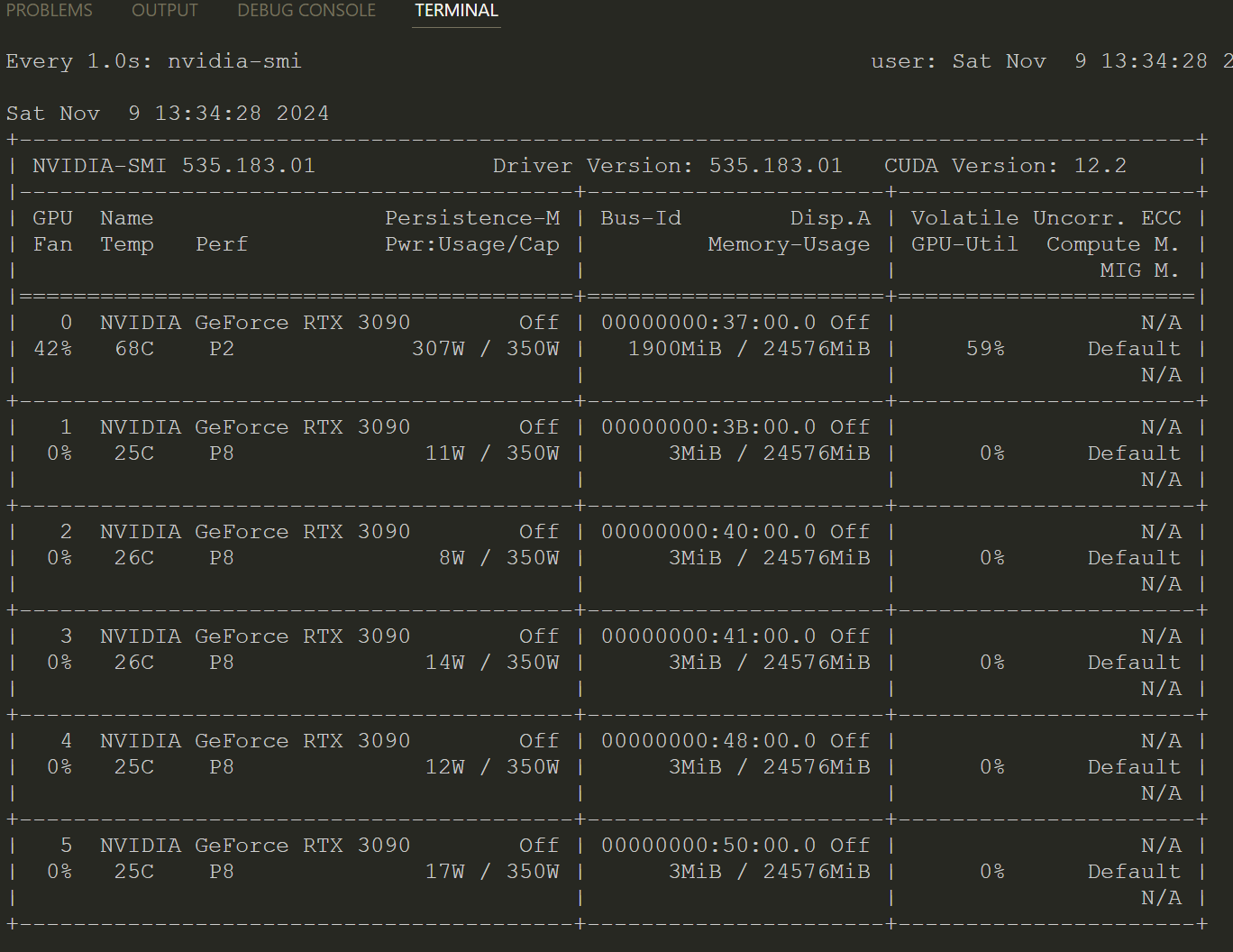


**2. 模型训练效果**

**（1） 显存使用情况与利用率**

[1] 以上参数训练50轮过程中，各个显卡的显存使用情况以及利用率如下

使用命令：watch -n 1 nvidia-smi

​

分析：

1. GPU 设备使用情况：

6 块 NVIDIA GeForce RTX 3090 显卡，每块显卡的显存容量为 24 GB。只有 GPU 0 正在使用，其他5块 GPU 处于空闲状态。

1. 显存使用情况与利用率：

GPU 0显存使用 1900 MiB(约 1991.33 MB/仅占1.8 GB左右)，处于较高负载状态，GPU利用率为56% 上下。其余GPU显存使用非常低，仅为 3 MiB，没有在处理任何显著的任务。

1. 温度与功耗：

GPU 0温度为 69°C，功耗 307W，处于较高的负载状态，正在进行计算密集型任务。其它GPU温度较低（26°C - 28°C），功耗在 8W - 15W 范围内，基本处于空闲状态。

1. 性能状态：

GPU 0性能状态为 P2，表示该卡在处理负载任务时正在优化能耗与性能。其它GPU性能状态为 P8，处于最低的性能状态，这种状态下 GPU 的时钟速度很低，以便节省功耗。

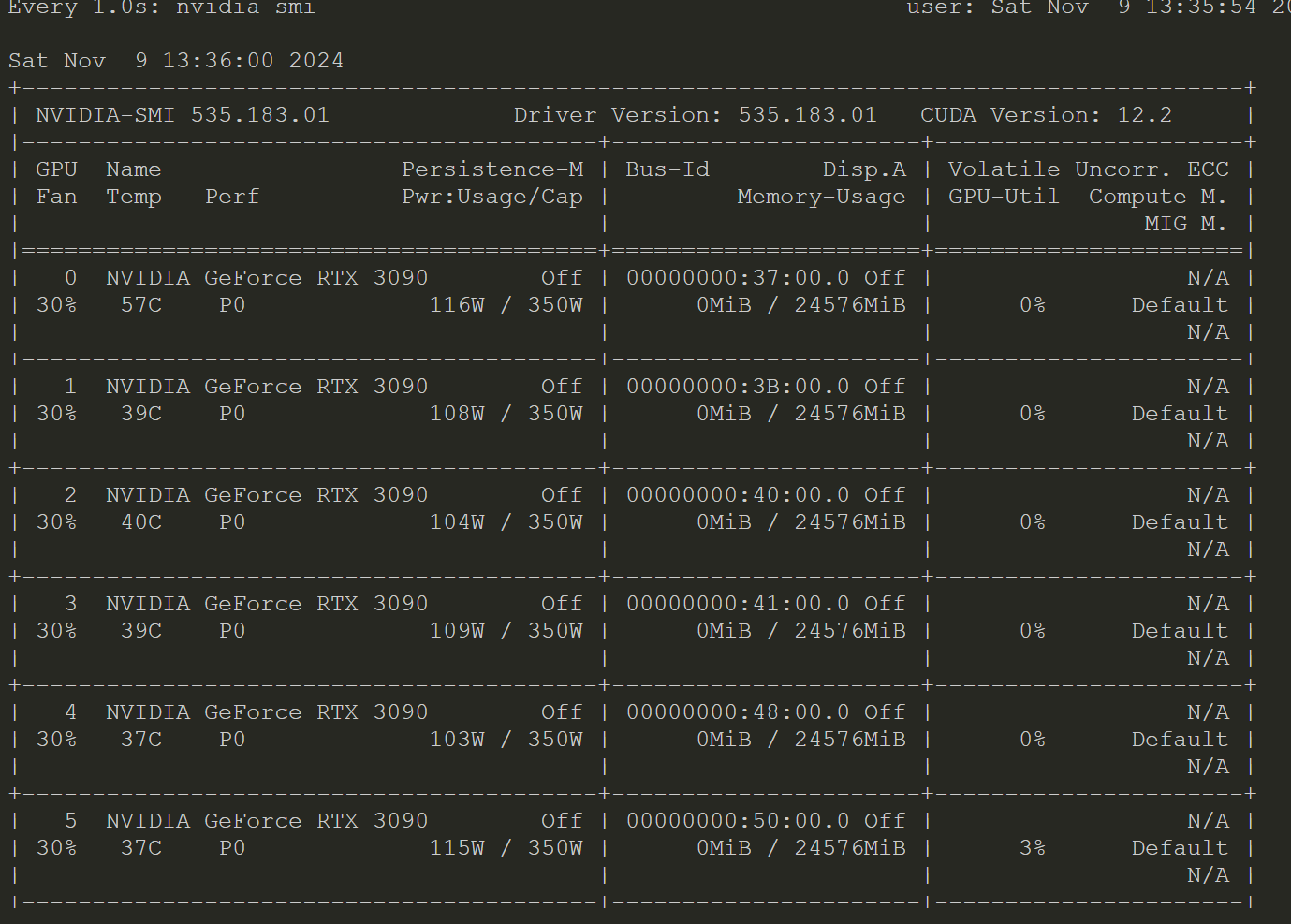
[2] GPU利用率的跳变

由于上图每秒都会更新，因此我们也可以观察到GPU利用率虽然常在59%左右，但也会发生跳变，跳变到很低甚至到0%。

事实上，这是GPU正在等待CPU传输数据。从总线传输到GPU 后，GPU逐渐开始计算，利用率会突然提高。但是，GPU非常强大，处理速度很快，因此利用率将再次下降，等待下一批进入。因此GPU利用率也取决于内存带宽和CPU的性能。

[3] 训练完成后：

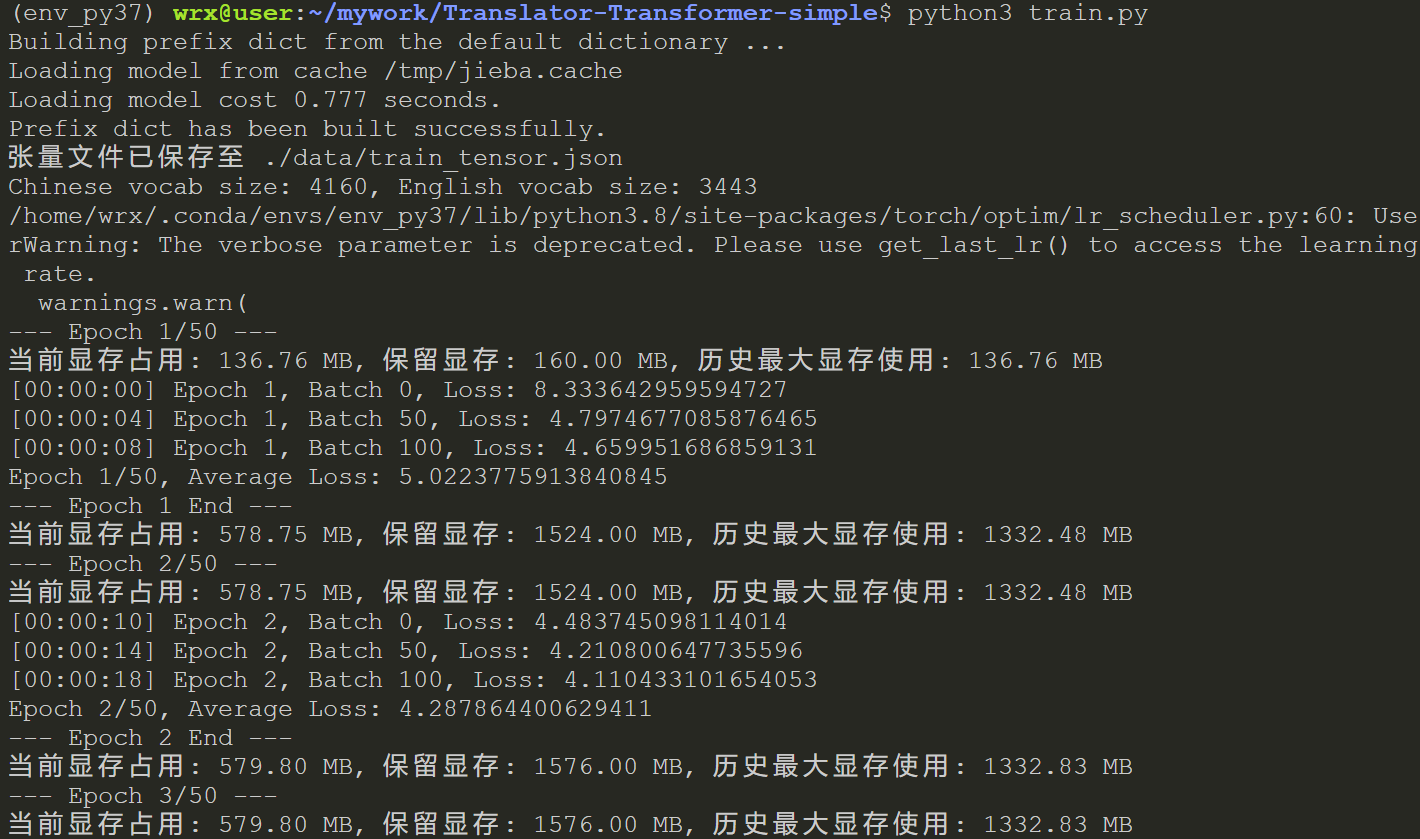
释放了显卡内存，温度逐渐降低。

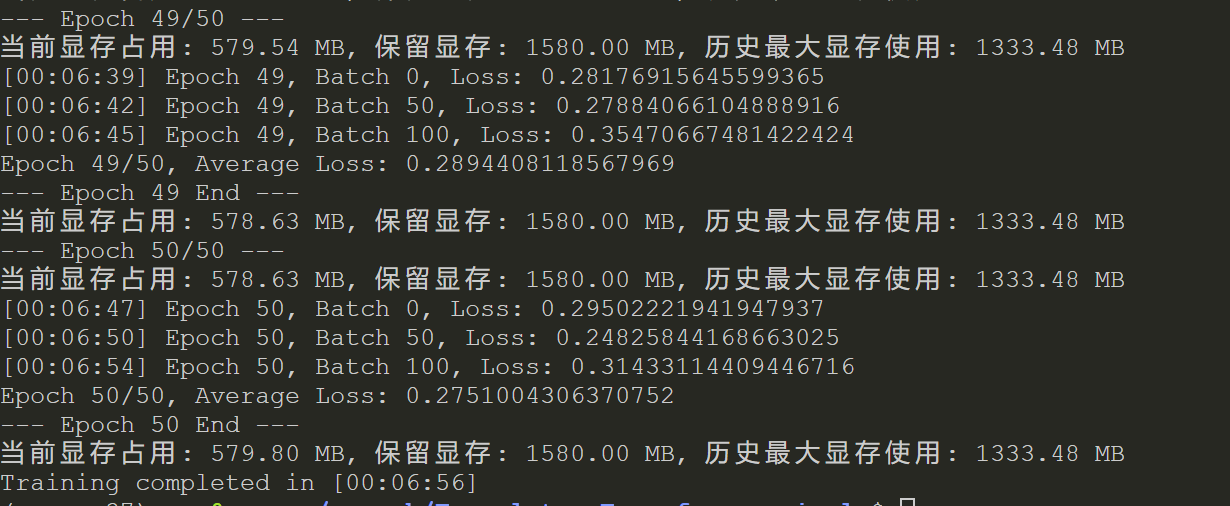


**（2） PyTorch函数获取显存信息(PyTorch 在当前模型和数据处理中的使用量)**

引入pytorch显存管理机制：

Pytorch为减少碎片化影响。当有多个 Tensor 可以被释放时，可以优先释放那些在内存空间上前后有空闲内存的 Tensor。这样便于 PyTorch 整理这些空闲块组成的碎片，让三个块组成一整个更大的块。





分析：

每轮训练开始和结束的时候当前显存占用基本保持在578MB左右，保留显存(1580MB)和历史最大显存(1333.48MB)也基本保持在一个平稳范围内。

因为模型的结构、参数大小、批次（batch）大小在每个epoch中都保持不变，每个批次的计算所需的显存基本相同。保留显存 = 当前显存占用 + 分配出的空余显存。历史最大显存使用是指 PyTorch 在整个训练过程中，某一时刻所用的最大显存量。这可能会比当前显存占用大，因为历史上曾经分配过一些更大的张量或中间结果，但这些张量可能在后续的计算过程中被释放或重新分配。

显存主要用于每轮训练中模型的参数和一个batch的计算所需的资源每轮结束后，旧的计算图会被释放回收，腾出来的显存可以继续分配给下一轮计算。

显存占用一般包括：

①模型参数。即包含了模型的权重（weights）、偏置（biases）等参数。无论是前向传播还是反向传播过程中，模型的参数始终需要存储在显存中。

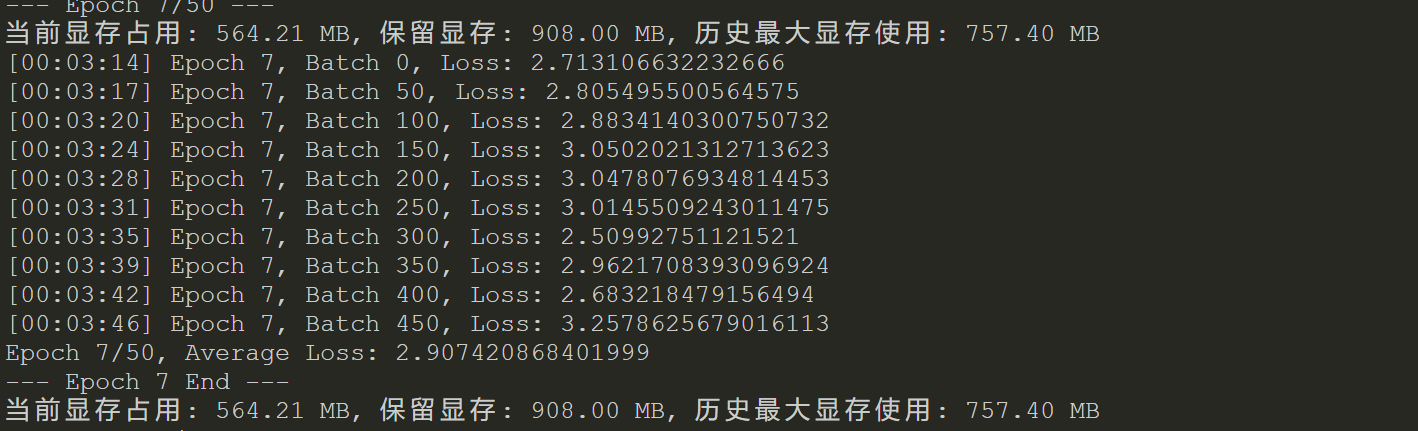
|  |
| --- |
| 参数占用显存 = 参数数目\*n，其中n为参数数据类型，如float32时，n=4  说明：1> 网络模型的占用的显存主要是来自于所有有参数的层，包括：卷积、全连接、BN 等；而不占用显存的有：激活函数、池化层以及 Dropout 等  2> 模型参数的显存占用与 batchsize 无关，但是在训练阶段与优化器有关 |

② 输入输出，神经网络的每一层输入输出都需要保存下来，用来反向传播。（但在某些特殊情况下，可以不保存输入。比如ReLU，在PyTorch中，使用nn.ReLU(inplace = True)能将激活函数ReLU的输出直接覆盖保存于模型的输入之中，节省不少显存）。

也就是说，训练过程中的中间变量即训练过程需要保存每一层输入输出的中间变量，一般后续反向传导，更新梯度，因此显存需要保存模型每一层输出的特征图（占用较多）。（因为反向传播中需要对中间层的特征图求导，所以中间层的输出特征图不会被释放）。

|  |
| --- |
| 与batch\_size有关， 输入张量(N, value\_len, heads, head\_dim)其中N就是一批次内样本数量，输入张量的内存占用就等于输入张量元素数量乘以每个元素占用的字节数（如使用 float32 类型时每个元素占 4 字节）。因此，批量大小越大，输入输出占用的显存也越大。 |

图中batch\_size等于128的显存占用，可以看到batch\_size = 32情况如下：



分析：

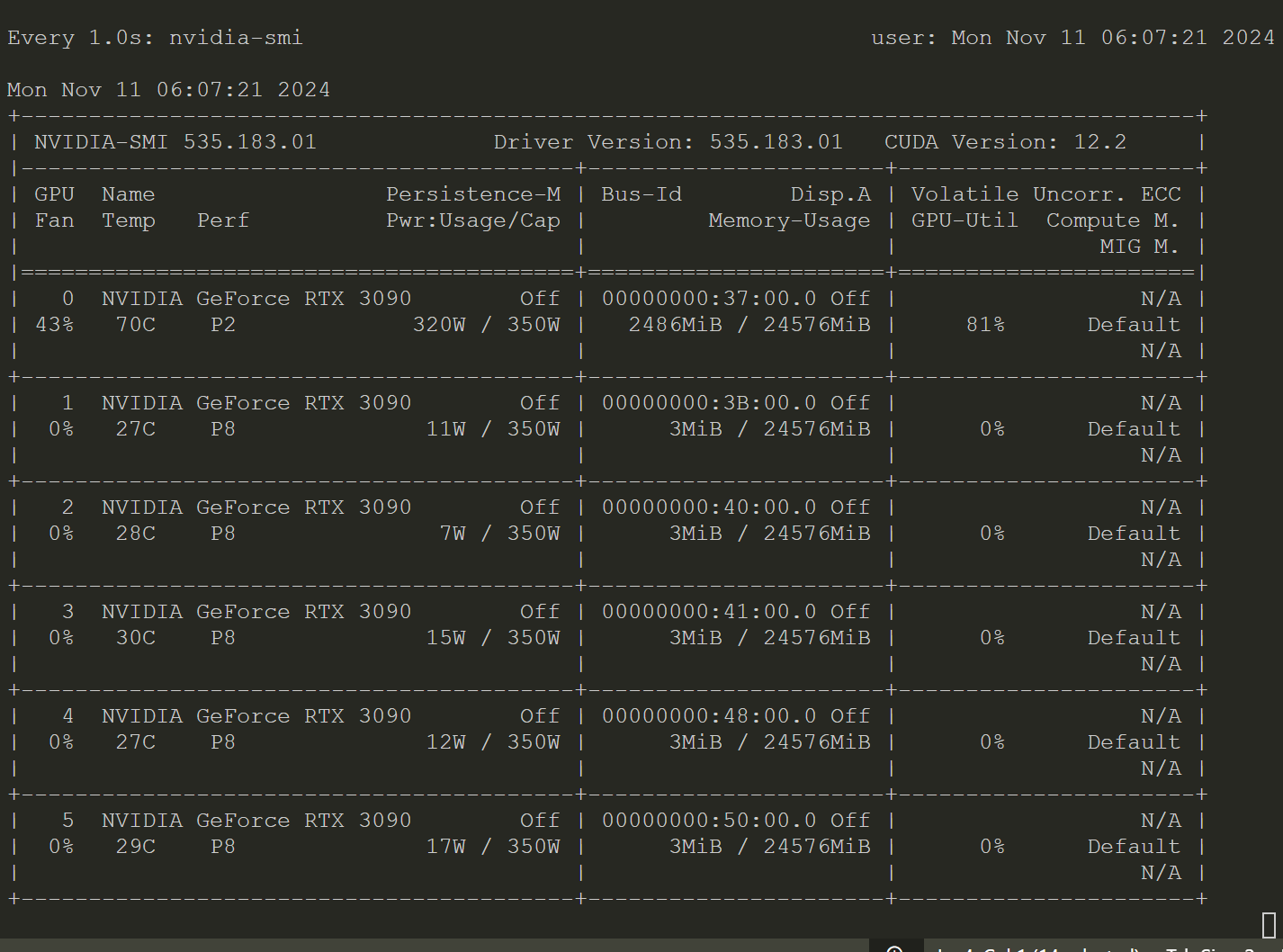
保留显存(1580MB->908MB)和历史最大显存使用(1333.48MB->757.40MB)都降低了近乎一半，说明训练历史上曾经分配过最大的张量也是远小于batch\_size=128的时候。

|  |
| --- |
| 与max\_seq\_length有关， 输入张量(N, value\_len, heads, head\_dim)其中value\_len就是序列长度，其决定矩阵的大小，决定输入输出张量的大小。因此，seq\_length越大，输入输出占用的显存也越大。 |

图中为max\_length = 15时的显存占用，可以看到max\_length = 25情况如下：



此外，全局显卡资源情况如下：



分析：

保留显存(1580MB->2166MB)和历史最大显存使用(1333.48MB->1880.80MB)都有明显增加，GPU显存使用(1900MiB->2486MiB)和利用率(56%->81%)也明显增加。可见序列长度增加，张量也增大，因此输入输出所占用的显存也有所增大。

③ 优化器状态(梯度、动量)。一些优化器（例如Adam）会为每个参数维护多个状态变量（如一阶动量和二阶动量），以辅助计算梯度更新。这些状态变量以及梯度也需要显存。

**梯度**：是模型参数相对于损失函数的导数。它表示了每个参数的值增加或减少对损失函数的影响程度和方向。梯度是通过反向传播计算得出的，用于指导模型在优化方向上更新参数。

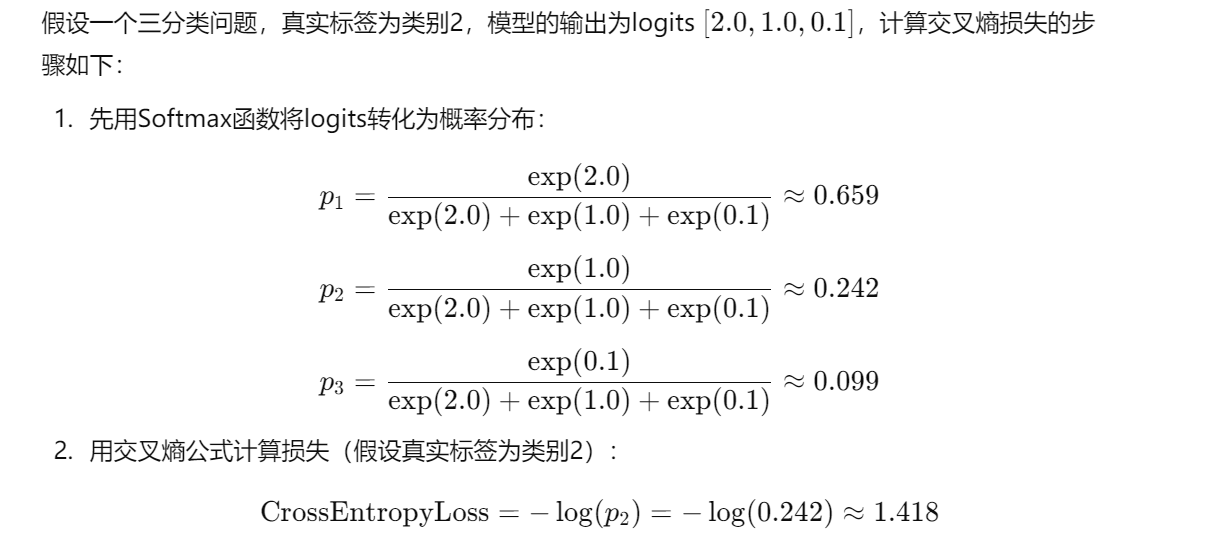
**动量：**是优化器在更新参数时使用的一种技巧，目的是加快收敛速度并减少震荡。动量会记录之前几个梯度的方向和大小，形成“动量”，从而在一定程度上保留之前的更新趋势，使得参数不容易被小的梯度波动打断，从而在合适的方向上继续前进。

保存梯度是因为，反向传播的核心是逐层计算每个参数的梯度。得到梯度后，需要用其更新参数值来优化模型。保存动量是为了加速收敛。

**（3）训练过程中的损失函数的变化情况**

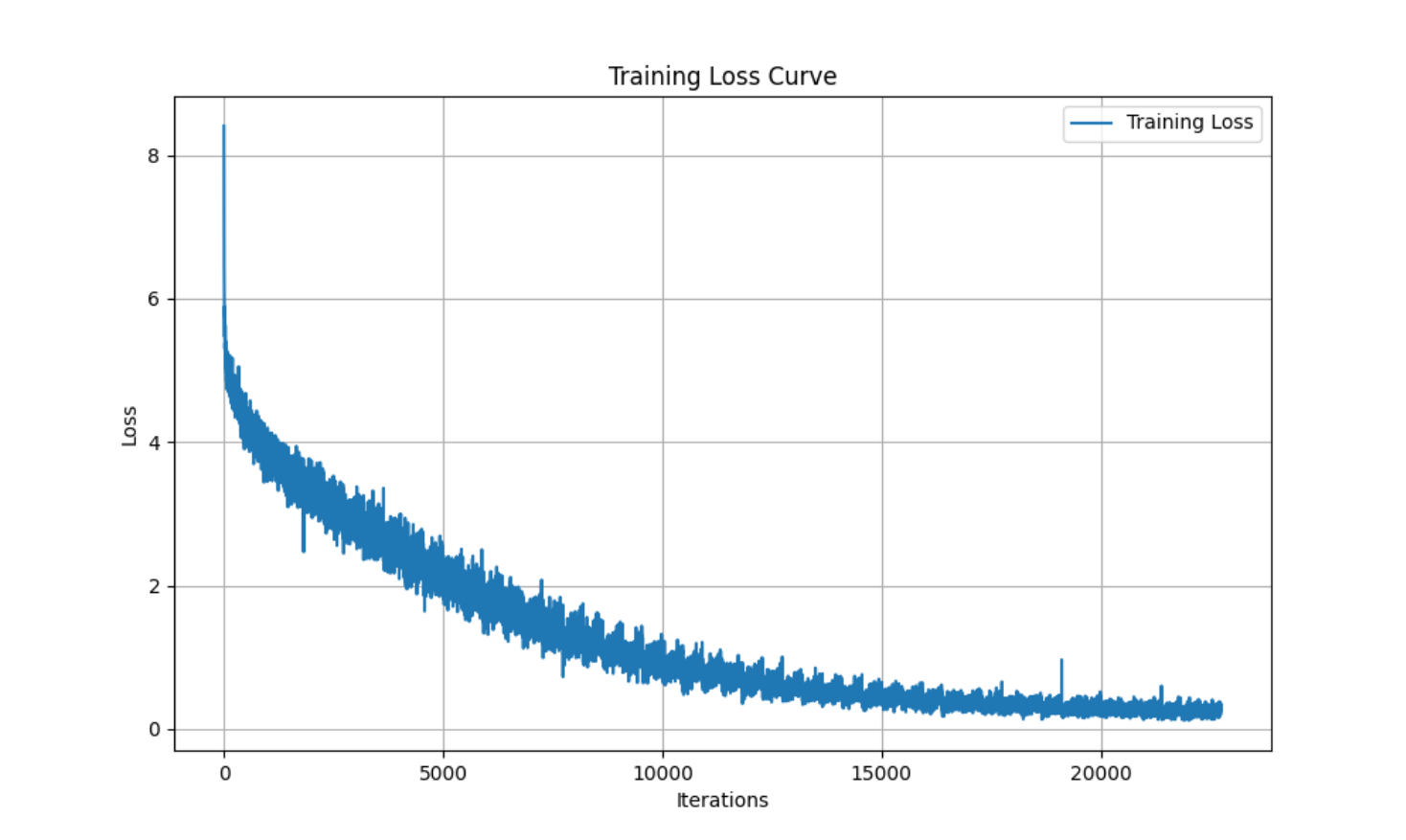
采用CrossEntropyLoss(交叉熵损失)作为损失函数。本模型的目标是将输入序列（源语言）转换为目标序列（目标语言）。对于每个时间步，模型会输出一个预测概率分布，表示当前单词属于目标词汇表中每个词的概率分布。CrossEntropyLoss 用于计算模型预测与真实目标单词之间的差距较为合适。

具体可以参考下面：



1. 批次大小batch\_size = 32, 训练轮次epch = 50轮时：

loss函数变化如下图：

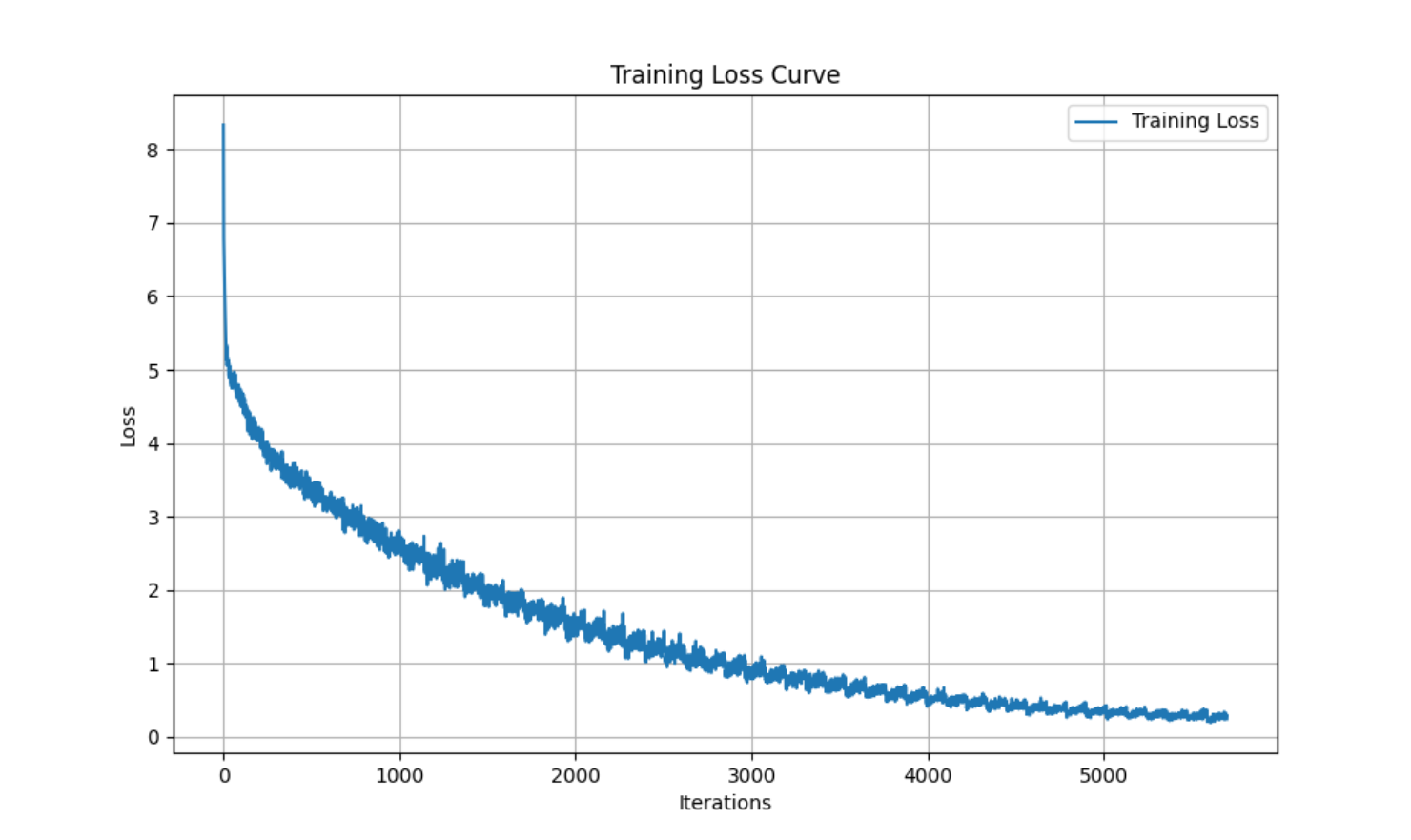


其中，横坐标Iterations表示loss的迭代次数，纵坐标表示损失函数的值，反映了模型预测误差的大小。损失值越低，表示模型性能越好。。可以看到训练到50轮，loss函数已经趋近于平稳。没有出现大幅度的损失波动，说明学习率和其他超参数设置较合理，没有出现不稳定或发散的问题。

此外，曲线中有一些小的波动，这可能是由于使用了小批量训练，导致不同批次之间损失值有一些小的随机变化。

1. 批次大小batch\_size = 128，训练轮次epch = 50轮时：

loss函数变化如下图：



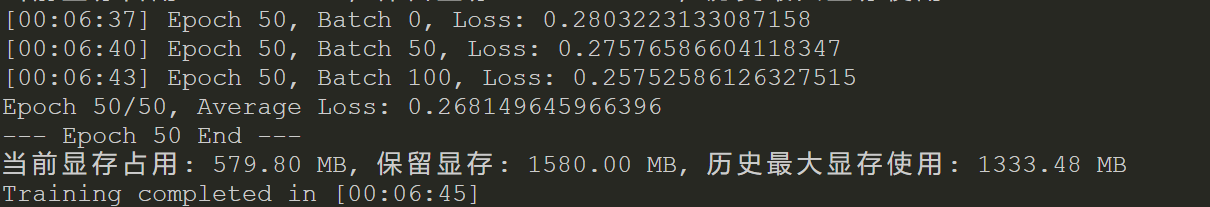
分析：

在初始阶段，损失值从大约8迅速下降，模型在早期阶段学习速度较快。随着迭代次数增加，损失值继续逐渐下降，并逐渐趋于平稳，模型接近收敛状态。

此外，与batch\_size=32的loss变化图做比较，使用了较大批次大小后，有效减少损失的波动，波动幅度相对较小，曲线更平滑。意味着较大的批次通常会产生较稳定的梯度更新，因为每次更新参数的样本数多，梯度估计也比较准确，使得损失曲线下降得更为平滑，有助于模型以更稳定的方式收敛。

**（4）训练用时情况**（数据集样本数为14533个）

1) 批次大小batch\_size = 128，训练轮次epch = 50轮时



可以看到，每轮训练用时约为6s,最终总用时6min 45s

2）批次大小batch\_size = 32, 训练轮次epch = 50 轮时：



可以看到，每轮训练用时约为32s,最终总用时26min 30s

总结：可以看到在GPU显存足够的情况下，batch\_size越大处理的越快。

**七、运行结果示例**

**1. 期望：**

来自训练集的部分数据为橙色表格，如下所示：

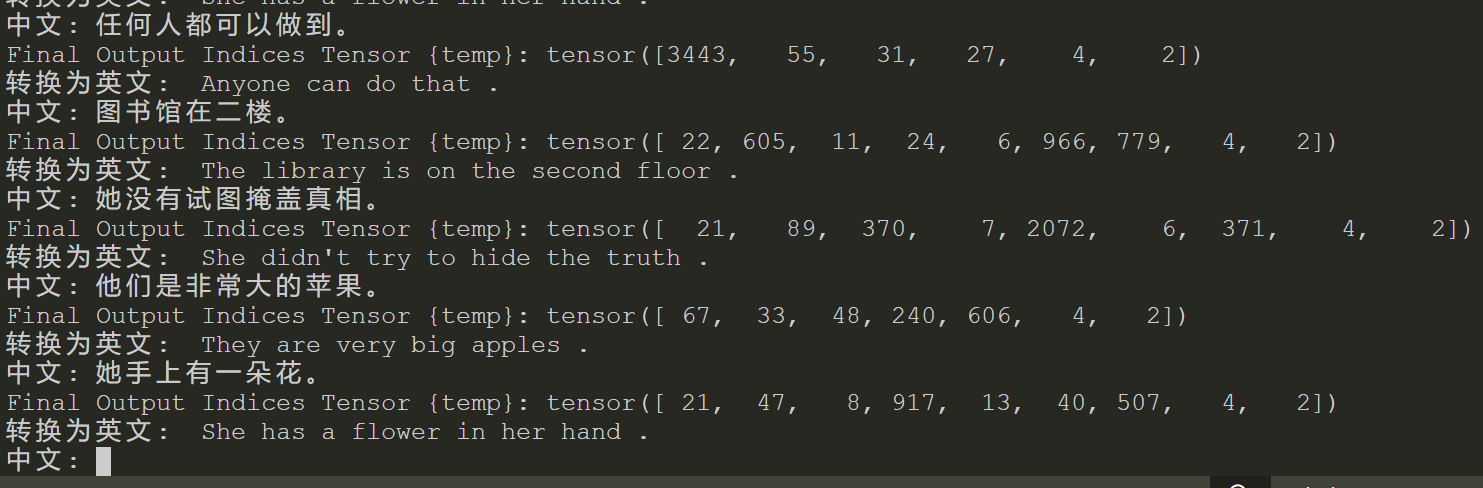
|  |  |
| --- | --- |
| input | output |
| 任何人都可以做到 | Anyone can do that. |
| 图书馆在二楼 | The library is on the second floor. |
| 她没有试图掩盖真相 | She didn't try to hide the truth. |
| 他们是非常大的苹果。 | They are very big apples. |
| 她手上有一朵花。 | She has a flower in her hand. |

来自测试集的部分数据为绿色表格，如下所示：

|  |  |
| --- | --- |
| input | output |
| 由你来决定。 | It's up to you. |
| 你应该帮他工作的。 | You should have helped him with his work. |
| 他几乎对那个动物一无所知。 | He knows almost nothing about that animal. |
| 我宁愿自己独立生活也不要他来指使我。 | I would rather live by myself than do as he tells me to do. |
| 上个月我们这儿没有下很多雨。 | We didn't have much rain last month. |
| 毛皮大衣在打折。 | Fur coats are on sale. |

**2. 运行结果如下：**

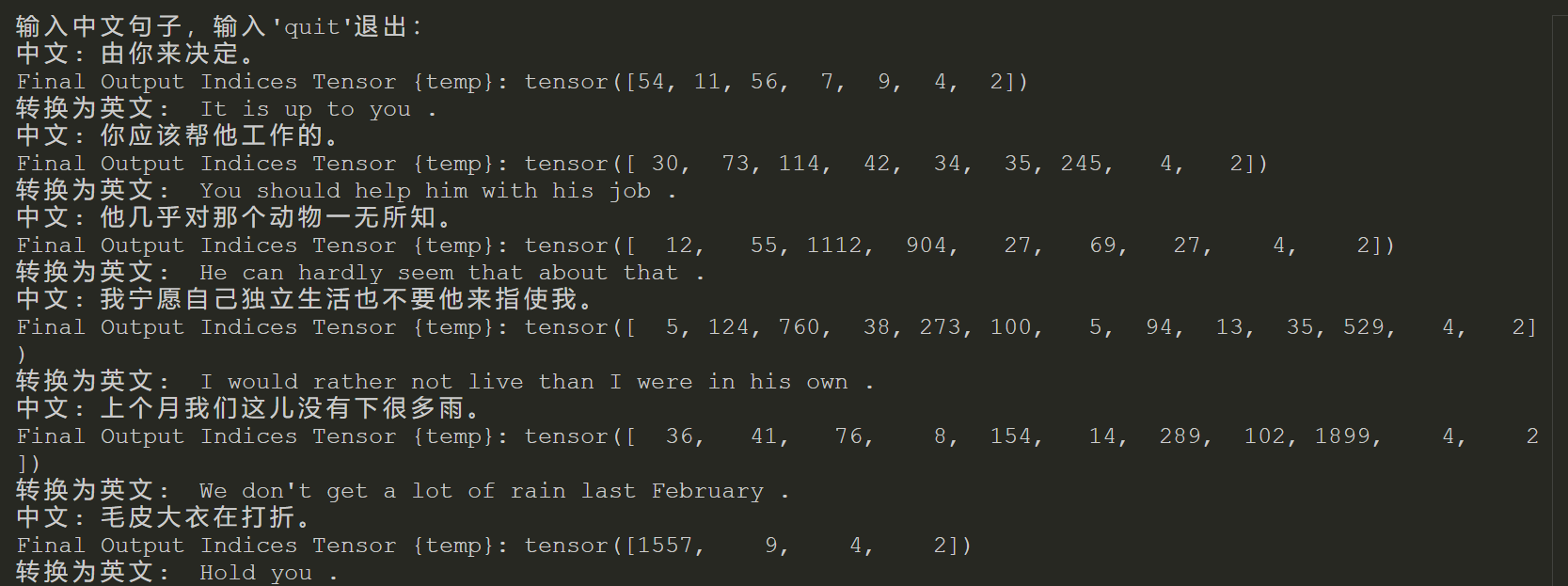
对训练集数据：



分析：

模型能够完美完成任务，模型正常工作。

对测试集数据：



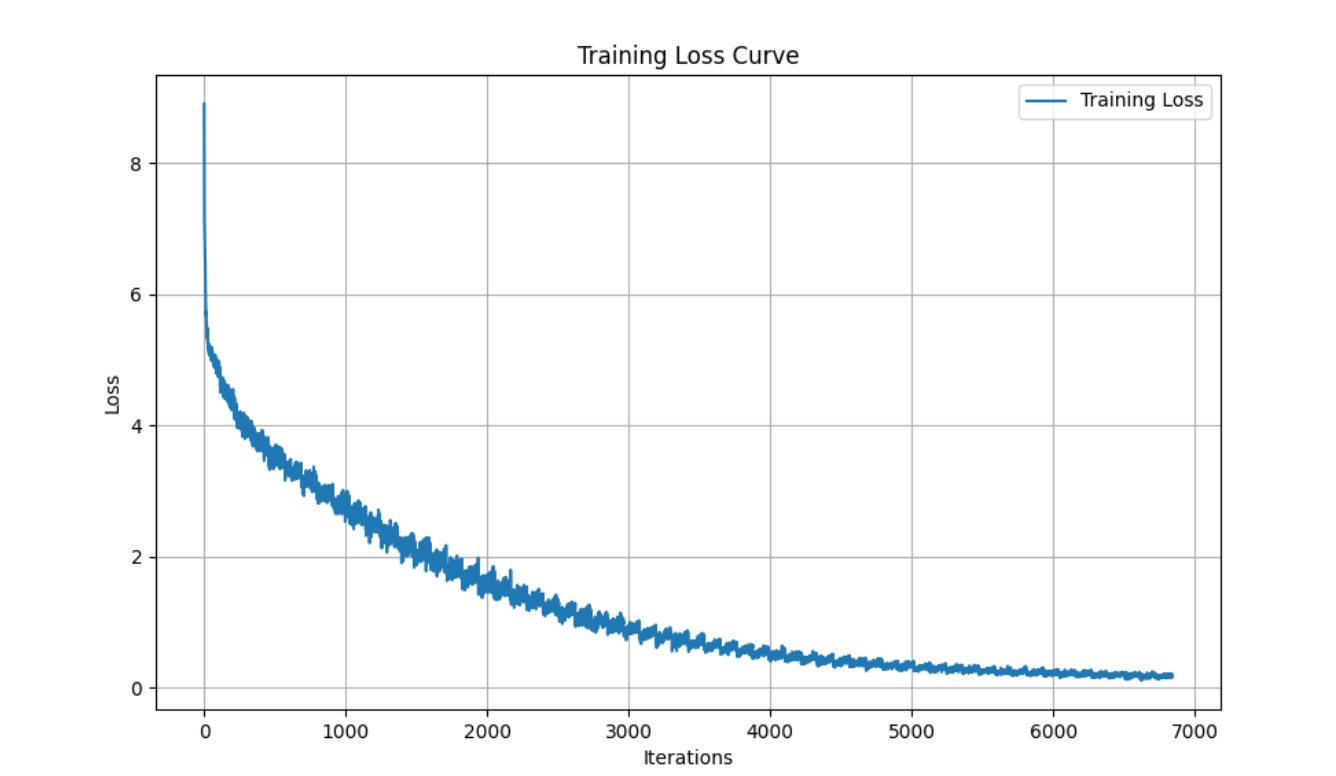
分析：

模型基本能完成任务，且会正常截断（博客内容任务之前不能正常截断，是因为只给训练集目标向量加了标志开始与结束，源向量没有加入，没有<sos>导致错位多了一位，所以模型总少第一位，没有<eos>导致模型无法正常截断），并且对句子有自己的理解，大体具有语法结构的捕捉能力。（这里需要注意，输入文本转换成张量的时候，一定要手动补上<sos>标志开始，否则也会缺少位。）此外，词汇表质量可能还不够好，模型对一些长句子或者低频词句子的翻译还不到位，并且语法结构还需要进一步训练，当然这可能与模型层数和训练集质量也有关系。

**八、 模型性能评估**

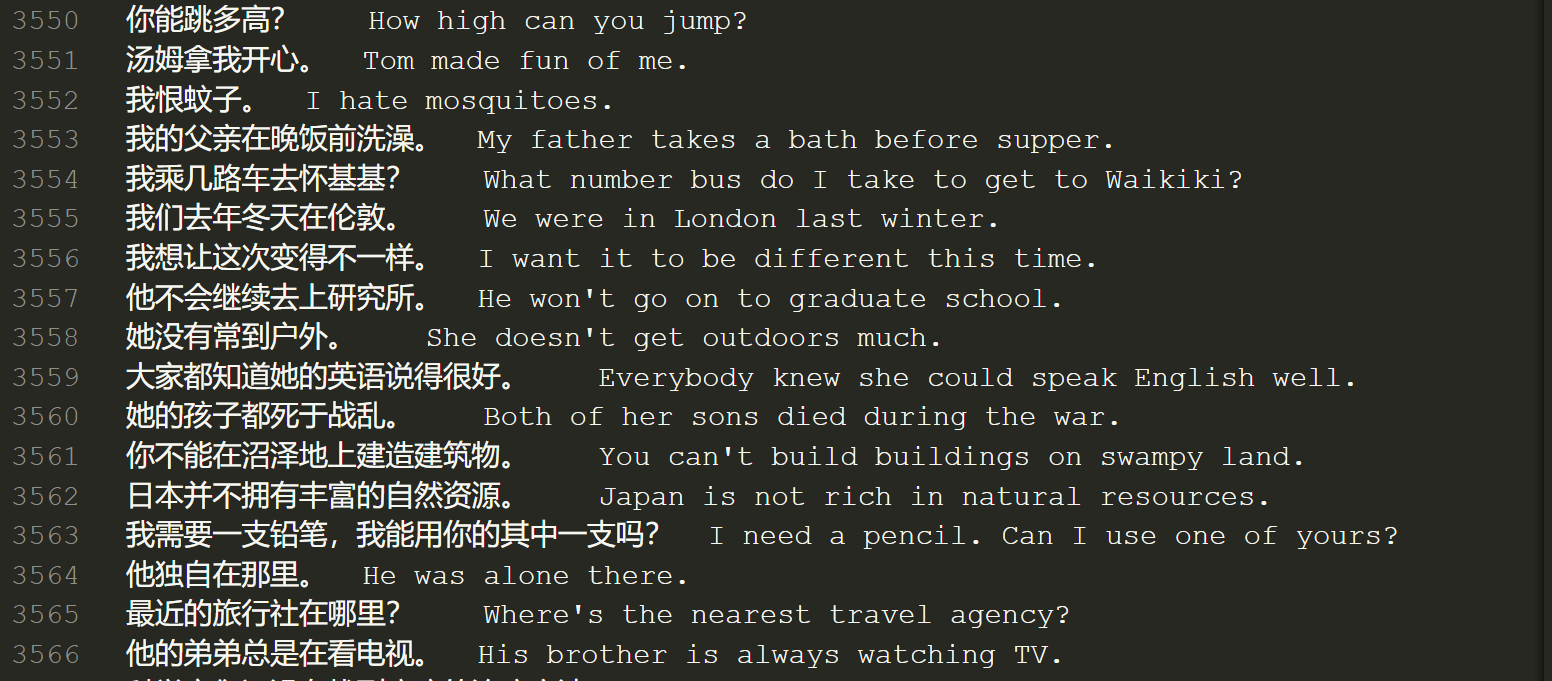
考虑到序列长度，最终进行测试的模型是max\_length从15增大到25、训练轮次num\_epochs从50变成60次训练得到的，变化的参数及损失情况如下。

|  |
| --- |
| max\_length = 25 # 最大序列长度  num\_epochs = 60 #训练轮次，可视loss变化情况而定 |



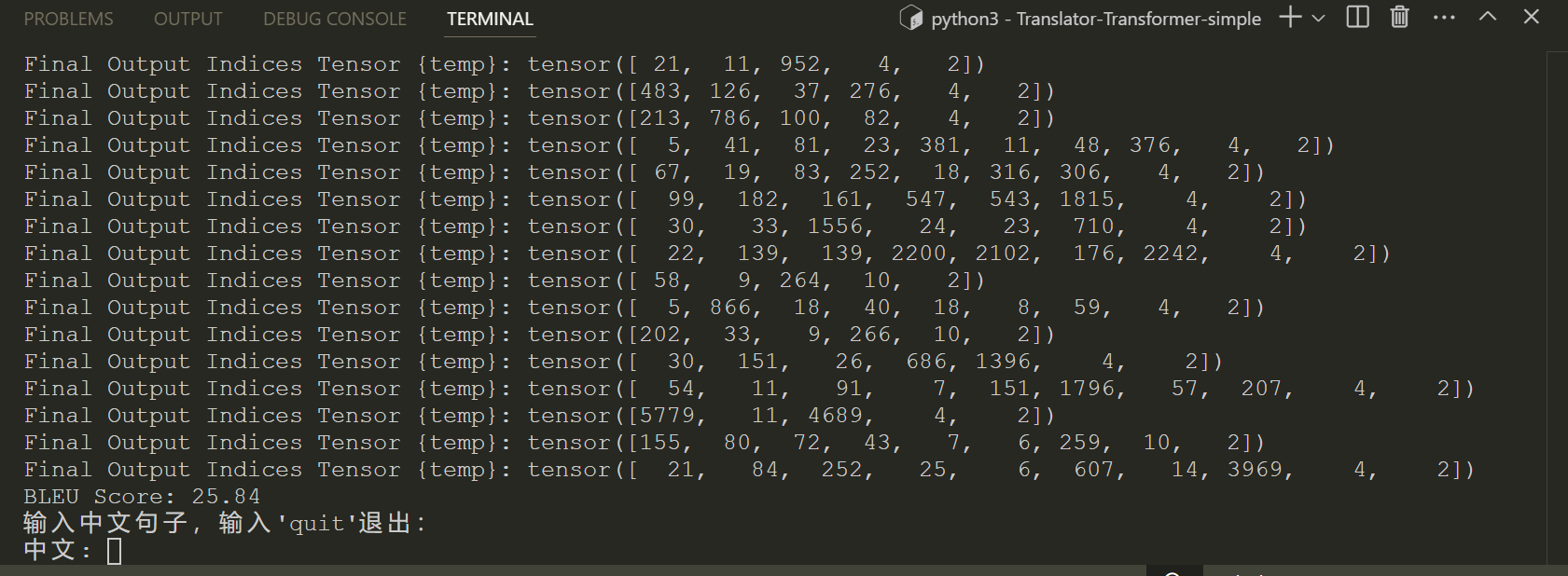
1. **测试集构建**

将测试集en\_ch\_test.txt使用preprocess.py文件进行预处理，使用opencc库将繁体中文转换为简体中文格式，又因为本任务方便输入是将中文转换为英文，所以调整为[简体中文 英文]格式，得到ch\_en\_test\_final.txt文件,具体格式如下：



测试集样本数为3644行，训练集：测试集 = 8 : 2。

**2．BLEU分数**



分析：

BLEU分数为25.48(这里从百分比方法为0-100内的分数)。对于不使用预训练而是手动搭建的transformer模型、且训练集只有18000多个样本来说，分数还可以，但仍然有较大的改进空间。

（20-30之间算好，20以下算一般，10以下很差）

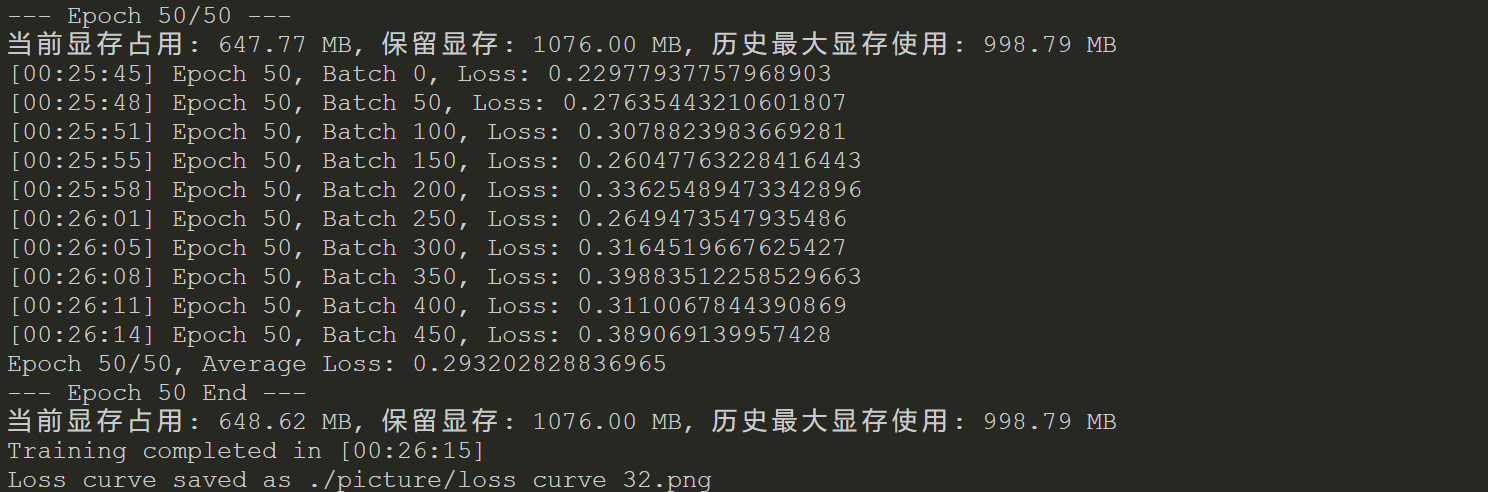
**3. 训练速度、所需容量与batch\_size的关系**

说明：

学习率：3e-4 、序列长度max\_length = 25、词汇表大小8888和6135

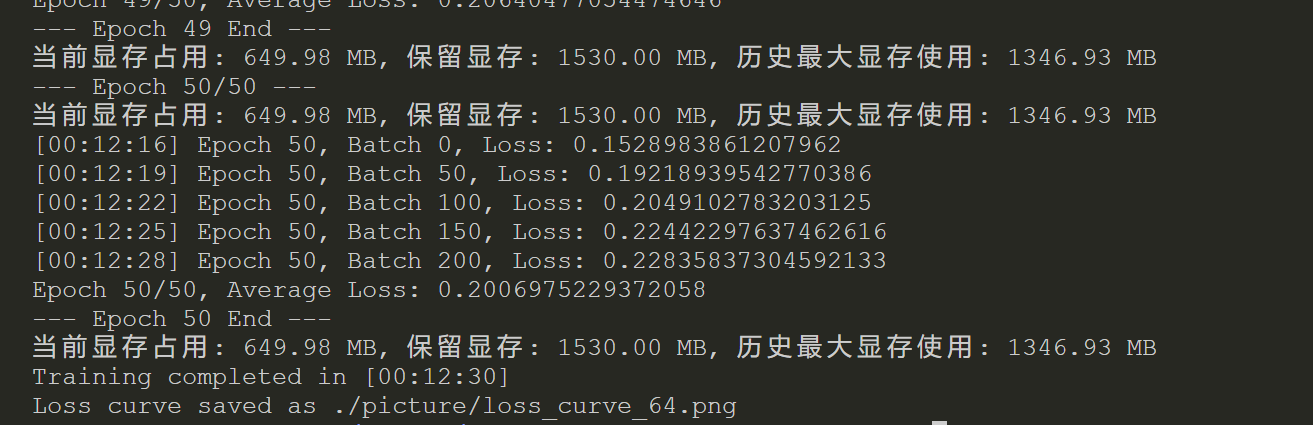
torch.cuda.memory\_allocated()可以精准地反馈当前进程中torch.Tensor所占用的GPU显存(注意是只包括torch.Tensor，eg模型参数、激活值、梯度、动量等)和torch.cuda.max\_memory\_allocated()，前者可以精准地反馈当前进程中torch.Tensor所占用的GPU显存(注意是只包括torch.Tensor，eg模型参数、激活值、梯度、动量等)，后者则可以告诉我们到调用函数为止所达到的最大的显存占用字节数，可能额外包含一些只存在很短时间的临时性显存。

（1）批次大小batch\_size = 32，训练轮次epoch = 50:



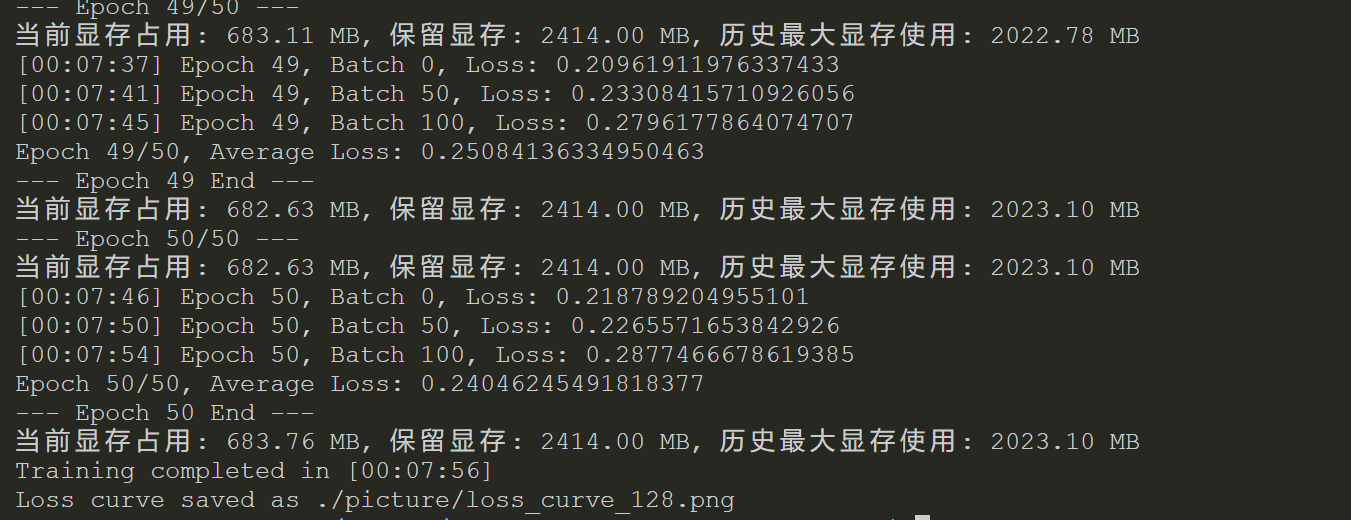
分析：当前显存占用为648.62MB，最大历史显存占用为998.79MB左右，每轮训练用时约为30s,最终总用时26min 15s。

（2）批次大小batch\_size = 64，训练轮次epoch = 50:



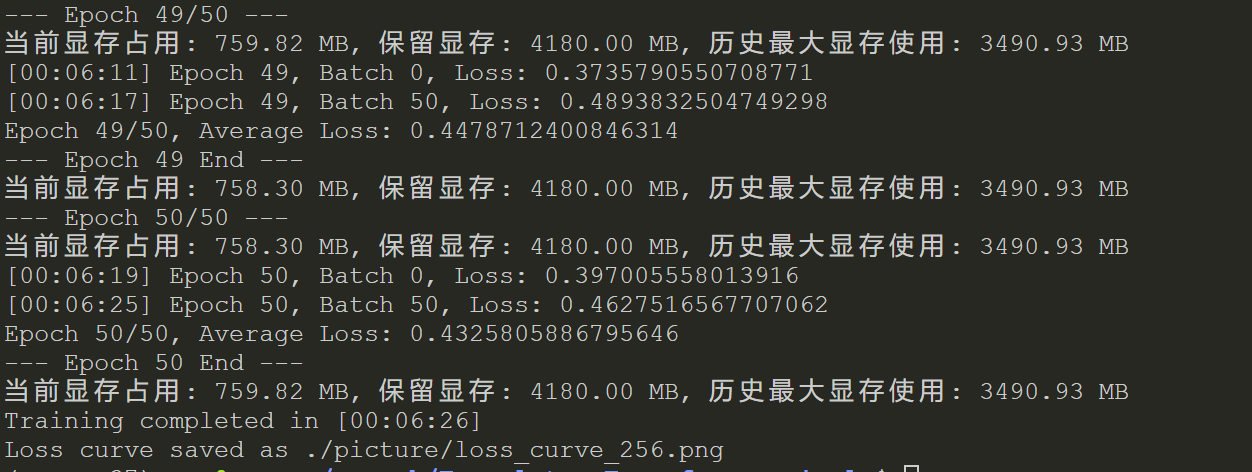
分析：当前显存占用为649.98MB，最大历史显存占用为1346.93MB左右，每轮训练用时约为13s,最终总用时12min 30s

（3）批次大小batch\_size = 128，训练轮次epoch = 50:



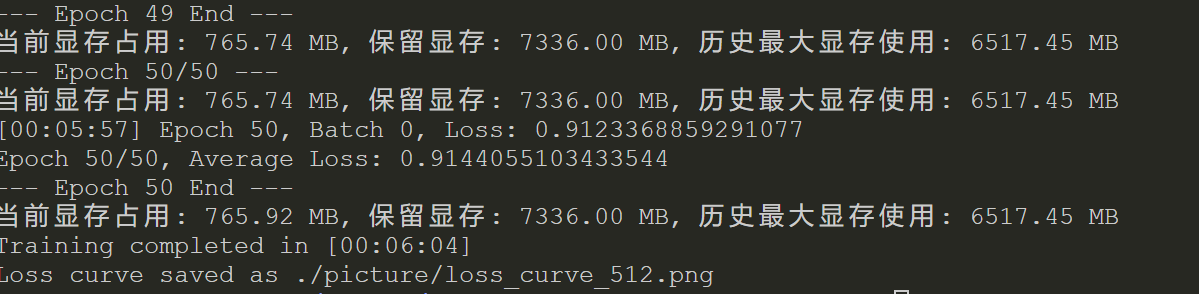
分析：当前显存占用为683.76MB，最大历史显存占用为2022.7MB左右，每轮训练用时约为8s,最终总用时7min 56s

（4）批次大小batch\_size = 256，训练轮次epoch = 50:



分析：当前显存占用为759.82MB，最大历史显存占用为3490.93MB左右，每轮训练用时约为6s,最终总用时6min 26s

（5）批次大小batch\_size = 512，训练轮次epoch = 50:



分析：当前显存占用为765.92MB，最大历史显存占用为6517.45MB左右，每轮训练用时约为4s,最终总用时6min 04s

Batch\_size与显存的关系:

Batch\_size与训练时间的关系:

分析：

Batch\_size与最大历史显存呈现明显的线性关系，而对于当前显存，虽然也随着Batch\_size增大而增大，但幅度不是明显线性。

训练时间随着Batch\_size的逐渐增大而减少，即训练速度加快，开始时候训练速度增加的快，随着batch\_size逐渐变大，其训练速度增加的幅度变缓。

**4. 显存大小的公式验证**

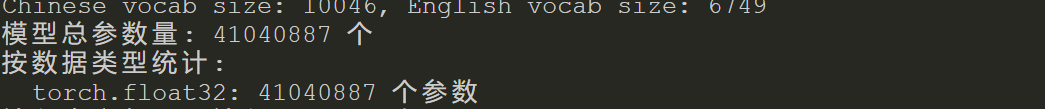
（1）公式1：

可粗略估计显存占用部分为：

1. 模型参数，占用大小为P\*4 bytes，其中 P为参数数目，本模型参数总数为41040887，具体如下：



类型如下：



1. 梯度，与参数对应，每个参数在反向传播时都有对应梯度，占用大小为P\*4 bytes
2. 优化器状态，例如Adam存储每个参数的梯度和动量，2\*P\*4
3. 中间激活(在前向传播中计算的中间结果，这些结果在反向传播中也会用到，用于计算梯度)：

计算公式为：



其中该公式计算结果精度为fp16，计算得到的单位为字节B。公式中b为批次大小，s为序列长度，d为隐藏层维度，d = embed\_size\* forward\_expansion（前馈全连接网络的第一层就是把词嵌入维度embed\_size映射为embed\_size\* forward\_expansion），h为多头注意力的头数。

分为以下部分：

**[1] Self-Attention 块的中间激活值**

Self-Attention 计算的中间激活值显存占用大小总和为:11bsd+5bs2h。

bs2h矩阵大小解释：其中dk是切分后每个头的维度



**[2] MLP 块的中间激活值**

MLP 通常包含两个全连接层和一个非线性激活函数，中间激活值占用显存大小为：19bsd

**[3] layernorm标准化层**(计算平均值和标准差)，需要保存其输入大小为2bsd，

2个layer norm 需要保存的中间激活为4bsd

打印本模型部分中间激活值数据类型：



最终公式为，总显存占用 = 4P+ L\*(34bsd + 5bs2h)，可以推测显存占用是和批次大小呈现线性关系的。

估计本模型不同批次大小时候的显存占用，得到如下表格：

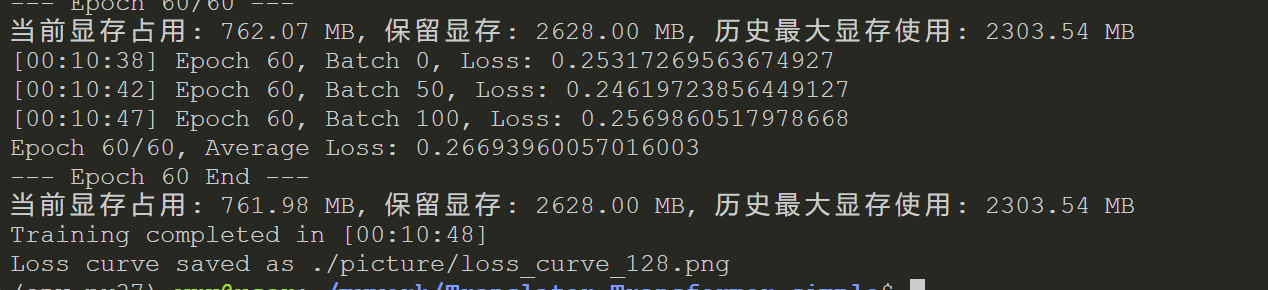
|  |  |  |  |
| --- | --- | --- | --- |
| Batch\_size大小 | 当前显存/MB | 最大历史显存/MB | 公式计算显存/MB |
| 32 | 648.62 | 998.79 | 949.5618744 |
| 64 | 649.98 | 1346.93 | 1272.889511 |
| 128 | 683.76 | 2022.7 | 1919.544785 |
| 256 | 759.82 | 3490.93 | 3212.855331 |
| 512 | 765.92 | 6517.45 | 5799.476425 |

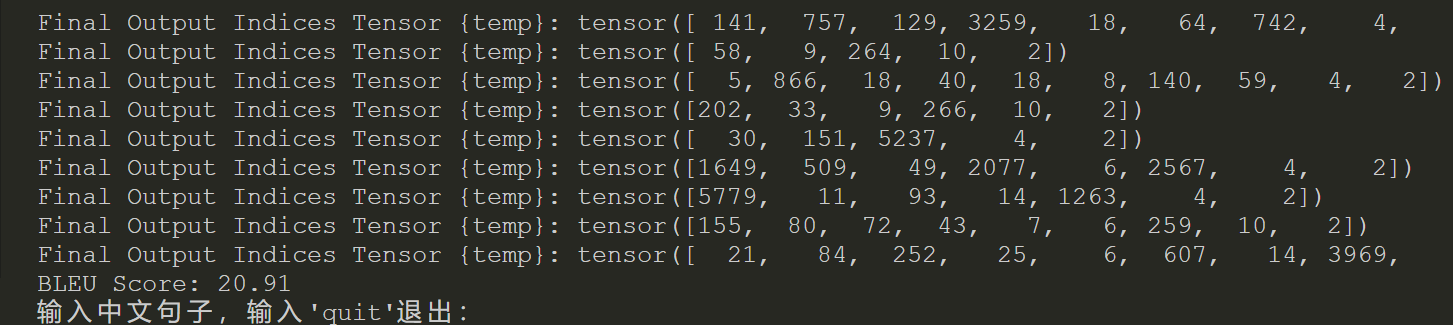
**5. transformer 层数增加对模型效果的影响(用BLEU分数衡量**)

说明：

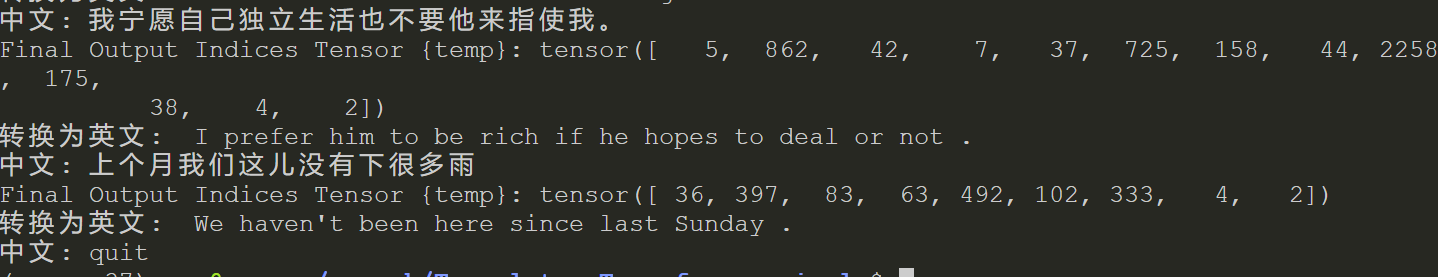
Transformer模型层数是6层时，采用本文档第五点中训练参数，训练后的模型对测试集进行翻译，得到的BLEU分数为25.48。

（1）将模型层数改为7层，num\_layers = 7，其余参数不变，BLEU分数如下：



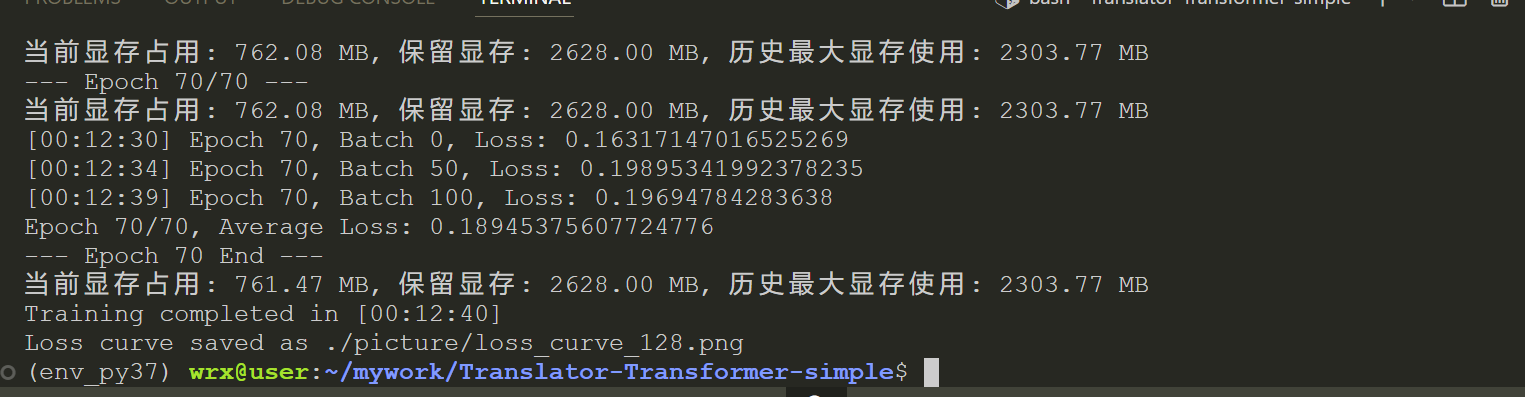


BLEU分数为20.91，对同一句子的效果也变差：

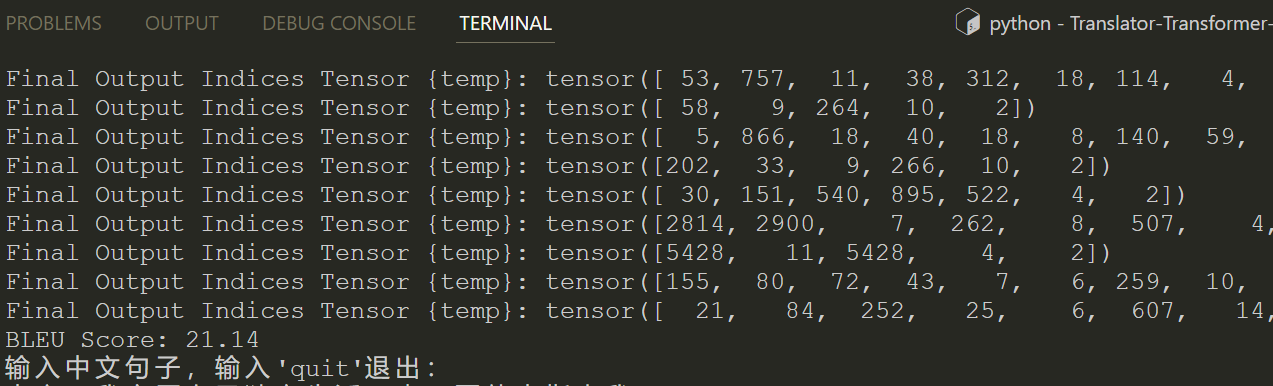


增大学习率，效果更差，目前学习率对当前批次大小较为合适。

继续增加轮次判断，改为70轮次。

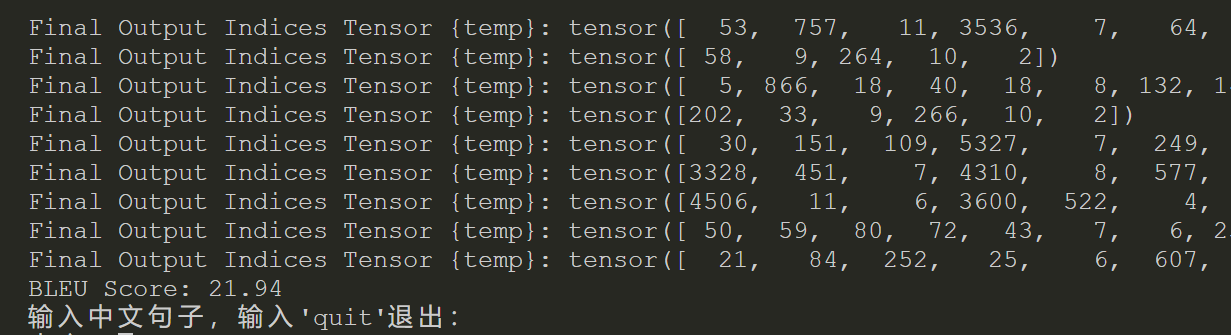


BLEU分数如下，为21.14：

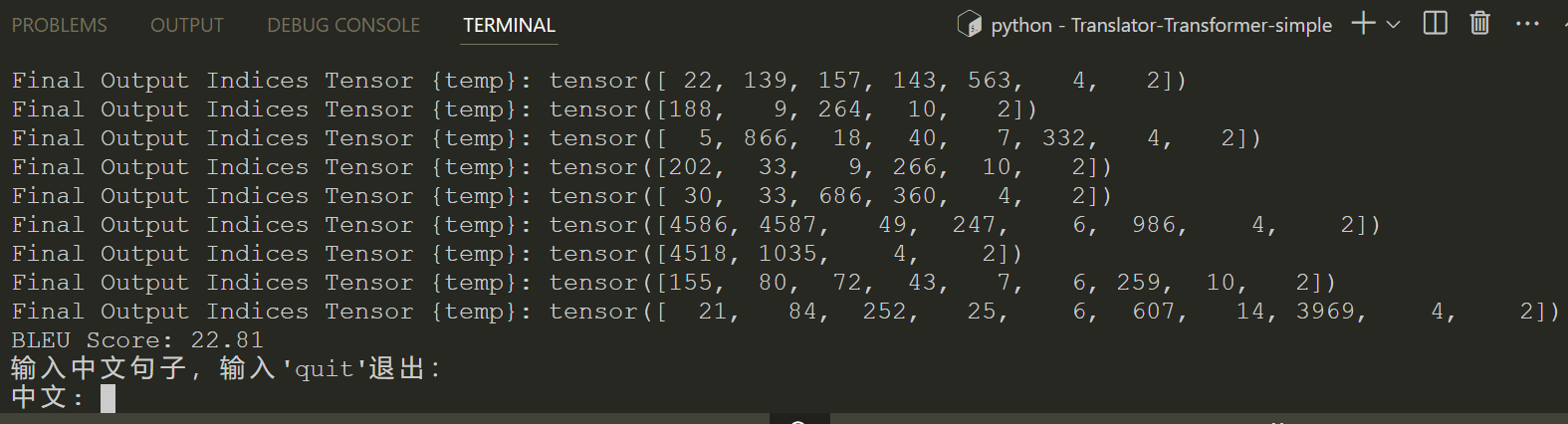


可能是数据量不足以支持这么复杂的模型。

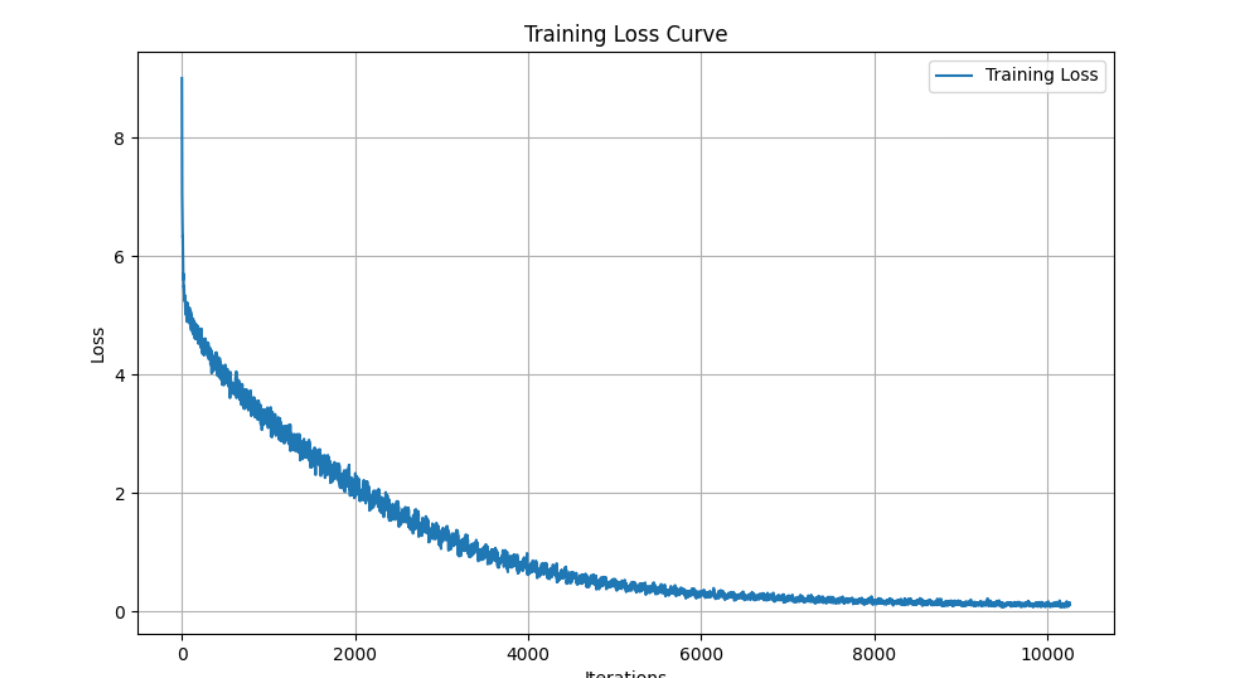
训练轮次增加到80轮次，bleu分数如下，为21.94：



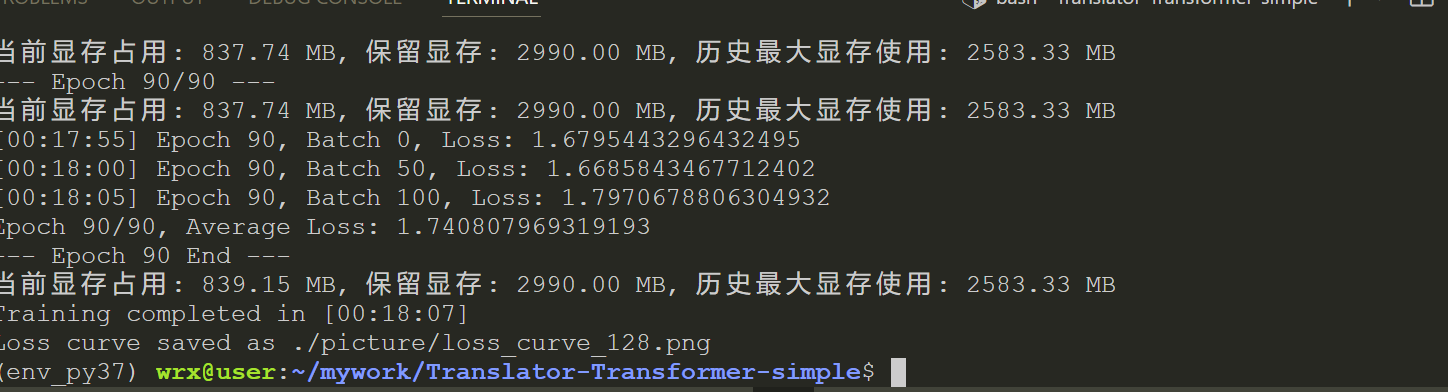
训练轮次增加到90轮次，bleu分数如下，为22.81：

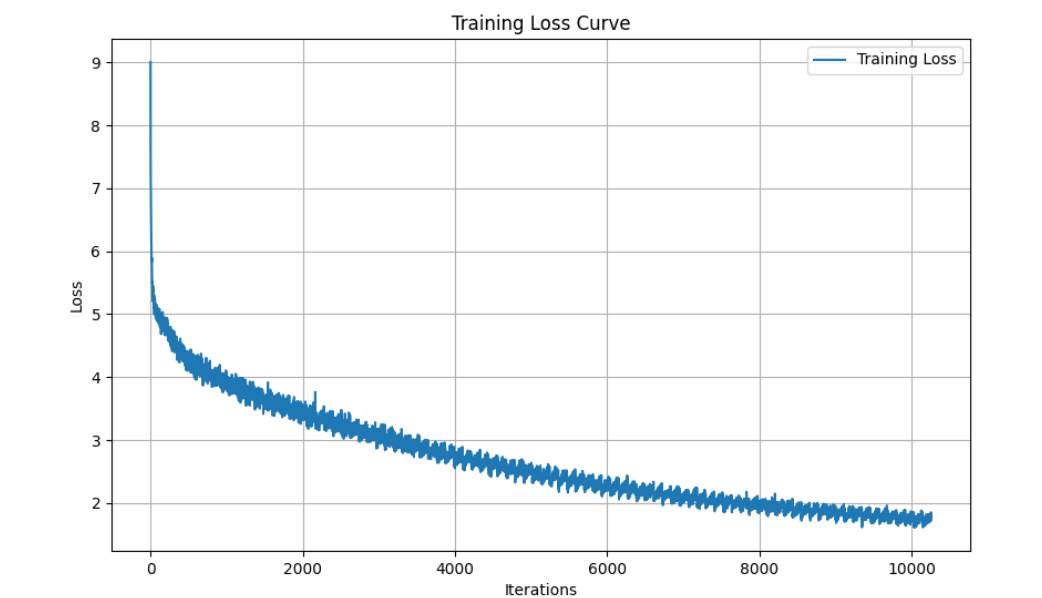


可以推测继续提升训练轮次，提升不大，loss损失值也已经平稳。

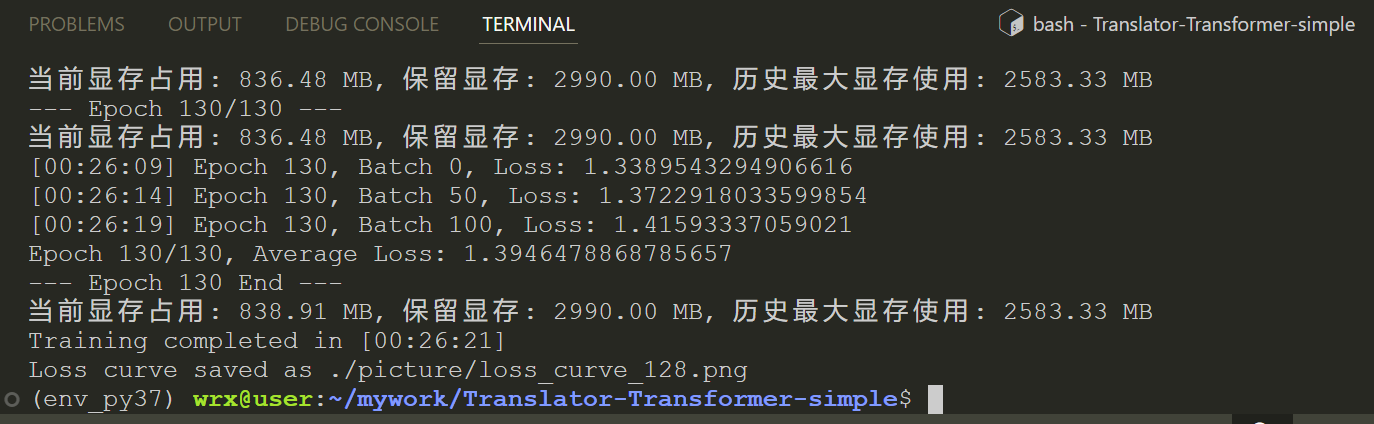


（2）提升模型num\_layers到8层，训练90轮次，loss还未收敛：





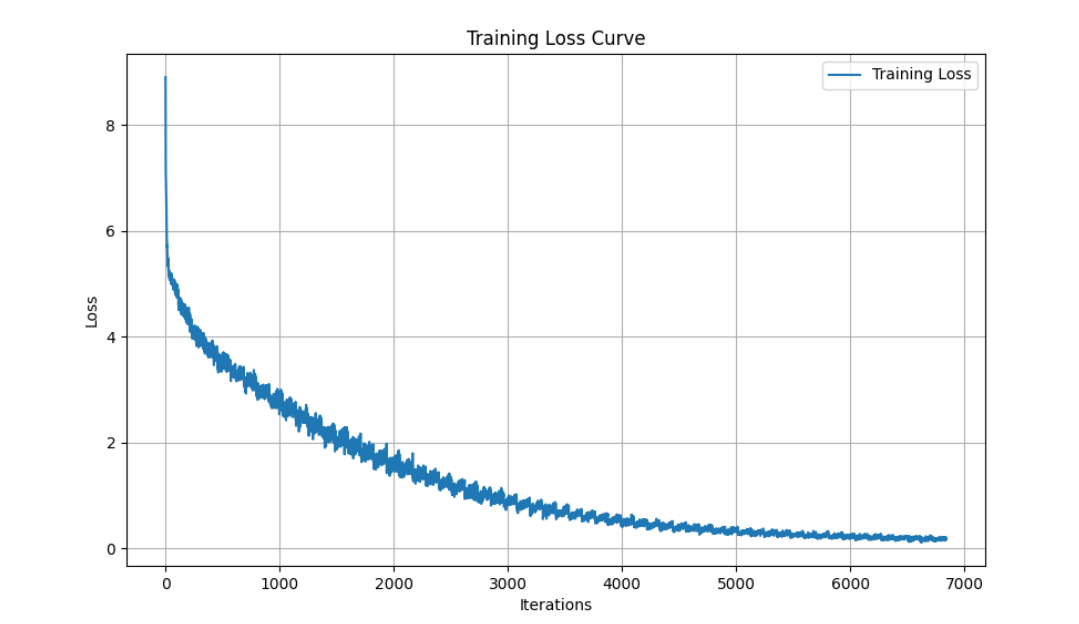
增大到130轮次,训练情况如下：

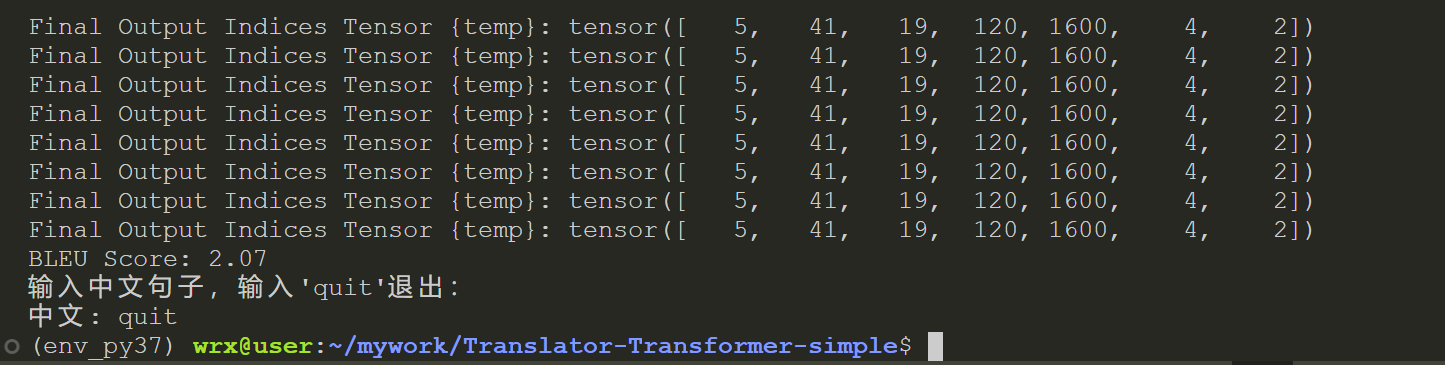


可以看到loss值没有降低的很好，仍未收敛，且波动很大。

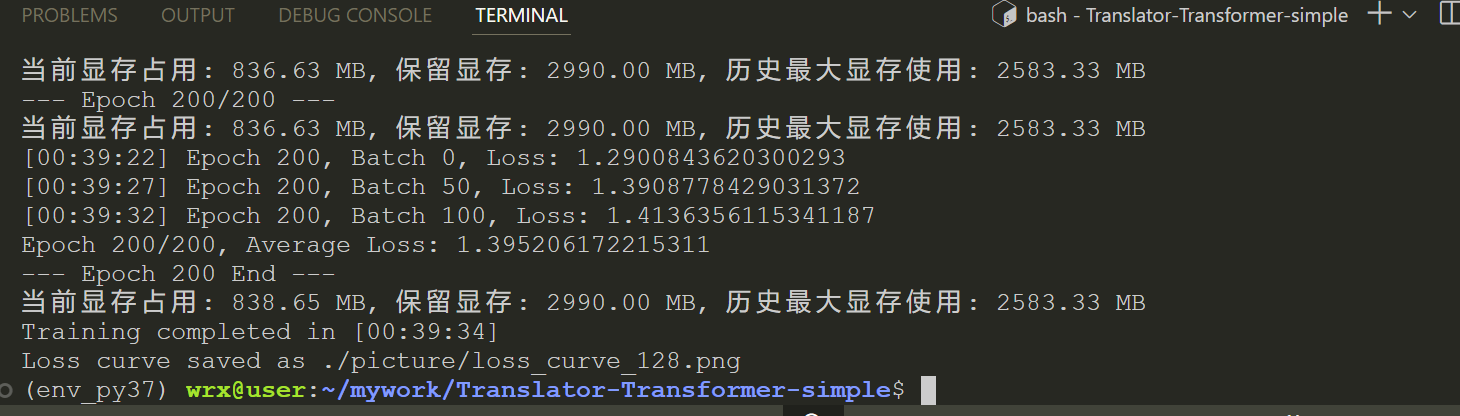


可以对比层数为6层的模型训练情况:

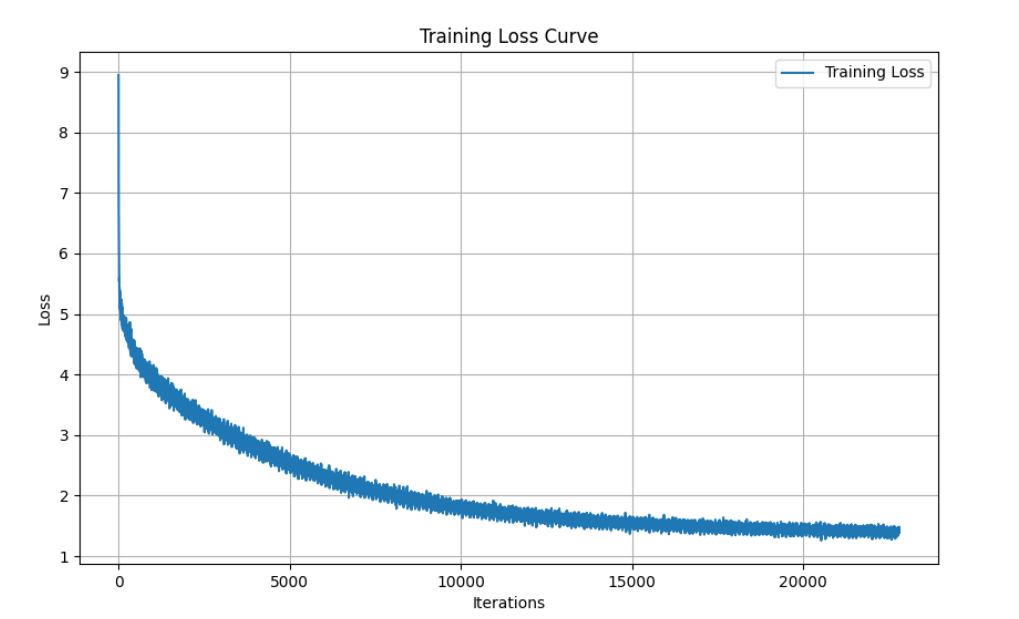




继续训练200轮次：



模型收敛至loss值为1.3-1.4左右。



增大学习率至2倍,loss值收敛至4，较高的学习率会导致模型参数在接近最优解附近来回振荡，(参数更新的幅度较大，导致训练初期参数的变化较为剧烈)，始终无法找到最佳解，使得损失值在较高的水平上震荡甚至停留。（梯度爆炸）

梯度爆炸与梯度消失相反，是指在网络反向传播过程中，由于链式求导法则的累乘效应，当每一层梯度都大于1时，某些参数的梯度会变得非常大。这导致在更新这些梯度时，参数变化过大，进而使得损失函数出现震荡现象。



**(3) 加深Transformer模型层数存在训练难度**

①局部最优状态。

局部最优状态：在深层网络中，由于层数增多，模型容易在初期就陷入局部最优解。这是因为深层网络中的优化过程更复杂(参数空间增加，损失函数复杂)，梯度更新路径更加曲折和不稳定(梯度消失与梯度爆炸)，特别是如果模型没有足够的正则化或初始化手段(本模型使用dropout正则化以及pytorch默认初始化方法设置参数初始值)，模型很容易在训练的早期 “卡住”，难以进一步提高性能。

② 梯度消失问题：

梯度消失，是指在网络反向传播过程中，由于链式求导法则的累积效应，当每一层的梯度都小于1时，会出现某些参数的梯度变得非常小的现象。这导致在更新这些梯度时，参数值几乎不发生变化，从而使得网络训练停滞，模型无法继续优化。

在标准的Transformer结构中，Layer Normalization通常位于每一层的自注意力和前馈网络之后（即Post-LN）。尽管标准化层Layer Normalization（LN）可以确保数据分布更加稳定，避免了极端值，有助于激活值稳定，从而有助于缓解梯度消失问题，但它并不能完全解决问题。

即使标准化了每一层的输出，模型的参数空间仍然很大，残差连接部分的标准化较少，这使得这些部分的权重变得更大，从而影响梯度的流动，进而导致训练不稳定；此外，它并没有解决跨层之间的依赖性问题。如果层间连接存在不合适的初始化或者某些层过于复杂，梯度仍然可能在某些情况下变得非常小或非常大，导致梯度传播不稳定。

改善方法：将Layer Normalization放在自注意力和前馈网络之前（即Pre-LN），可以改善模型训练的稳定性。这种结构能够帮助网络在训练初期保持更稳定的梯度流动，从而减少梯度消失的风险。Pre-LN通过在每一层的输入而不是输出上进行标准化，有助于保持梯度的大小，使得训练过程更加平稳。(论文)

Xshell

**6. 更换为特定某领域数据集(使得模型不能准确预测的回答我不知道)**

（1） 可采用的方法

① 置信度/平均置信度

当置信度低于设定的阙值，说明模型没有把握准确预测回答，因此回答我不知道。设定的阙值可以用回答目标领域内问题的置信度来决定。

② 场景限定校验

如果生成的词或句子超出领域词表或约束规则，可以认定为不确定。例如：

如果输出中包含未定义的标记、翻译与特定词典或语义规则不匹配。

构建领域词表，将模型的翻译结果逐词检查是否存在于词表中，如果检测到不在词表中的单词或标记，认定为不确定；此外词表检验通过后，还要使用特定的工具进行语法规则校验，不符合语法规则也认为不确定。

③ 其它语言模型校验

调用别的成熟模型对模型预测的回答进行打分，分值低则本模型回答我不知道

（2） 实际实现(采用平均置信度方法)

① 特定领域数据集

由于没有找到合适的特定领域的中英文翻译数据集，使用GPT生成了1031条餐饮场景下的数据(再继续生成内容同质化严重)，并据此构建了中英文词汇表。

|  |  |
| --- | --- |
| 文件名 | 内容 |
| rest\_ch\_en.txt | 餐厅场景下中英文翻译训练集，共1031条 |
| rest\_ch\_vocab.txt | 根据训练集构建的中文词汇表 |
| rest\_en\_vocab.txt | 根据训练集构建的英文词汇表 |

② 训练参数的调整

由于训练集规模很小，训练参数调整如下：

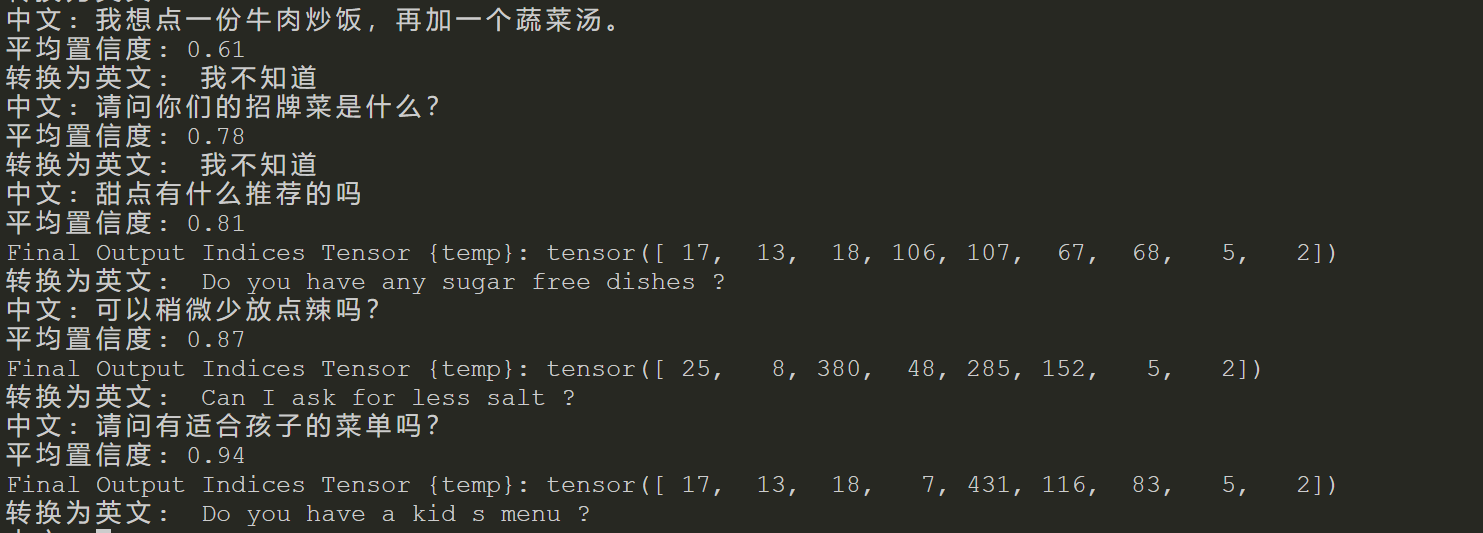
|  |
| --- |
| embed\_size = 512  # 常用256、512...      num\_layers = 6  # transformer模型层数      forward\_expansion = 4 # 扩展隐藏层维度，前馈全连接网络的参数      heads = 8  # 512可以被8整除，512/8=64      dropout = 0.1  # 随机化，防止过拟合      max\_length = 25  # 最大序列长度      num\_epochs = 60  # 训练轮次      learning\_rate = 3e-4  # 学习率 |

③ 利用置信概率实现模型对自己输出正确性的预测

|  |
| --- |
| # 获取当前(最后一个)时间步的输出分布  logits\_last\_step = output[:, -1, :]  # 转化为概率分布，归1化  probabilities = F.softmax(logits\_last\_step, *dim*=-1)  # 获取概率最大的索引  next\_token = probabilities.argmax(*dim*=-1).item()  # 记录置信度（最大概率值）  confidence = probabilities[0, next\_token].item()  confidences.append(confidence) |

由于transformer模型会选择token中概率最大的作为输出，因此我们取其概率作为置信度，累加后除以步长，即可得到平均置信度。当平均置信度小于某阙值，则表示模型比较没有信心，则输出“我不知道”。

1. 结果展示：置信概率0.8

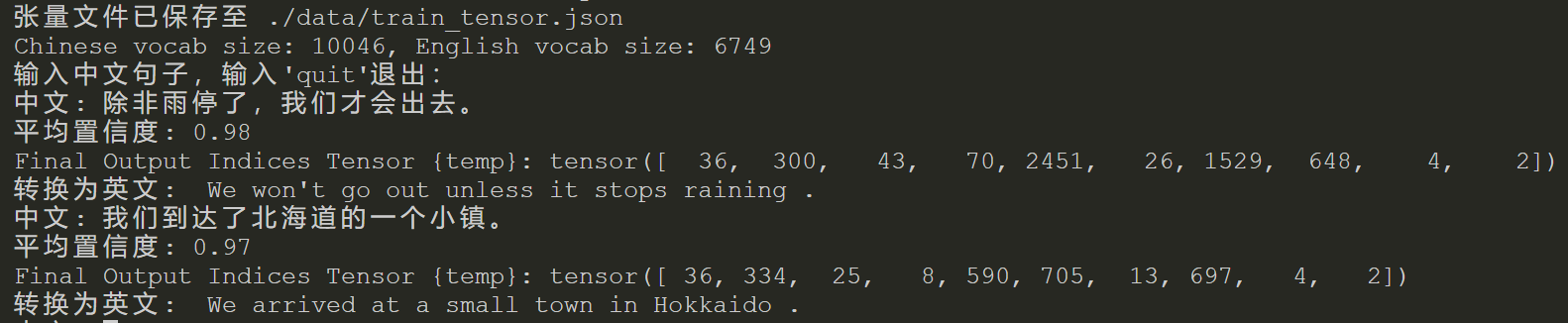
对于训练集，完全符合。

对于测试的句子，部分虽然输出但仍有错误，部分回答我不知道。

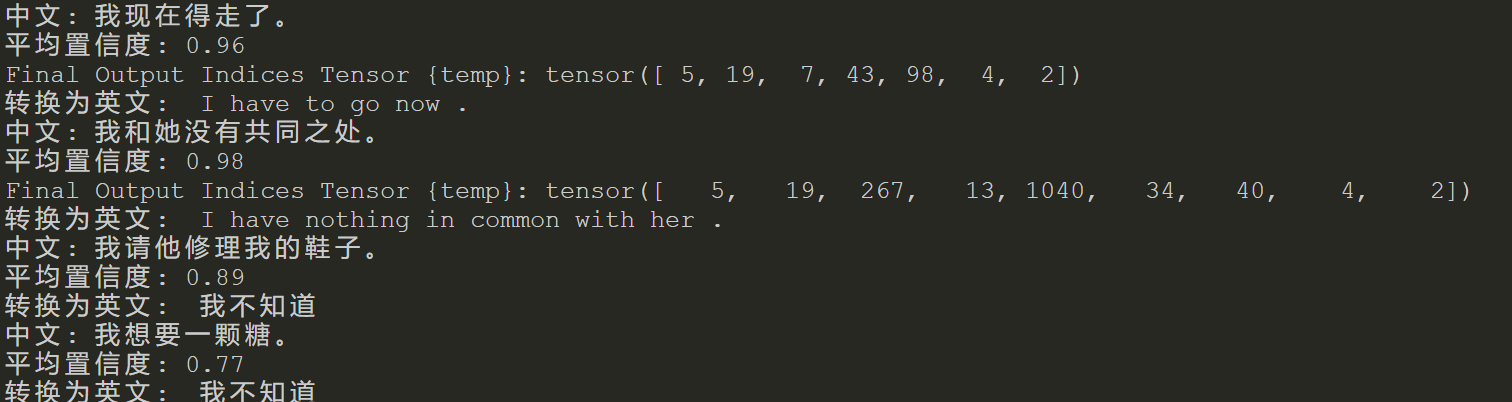
1. 其他测试，使用原数据集和模型：

置信概率0.9

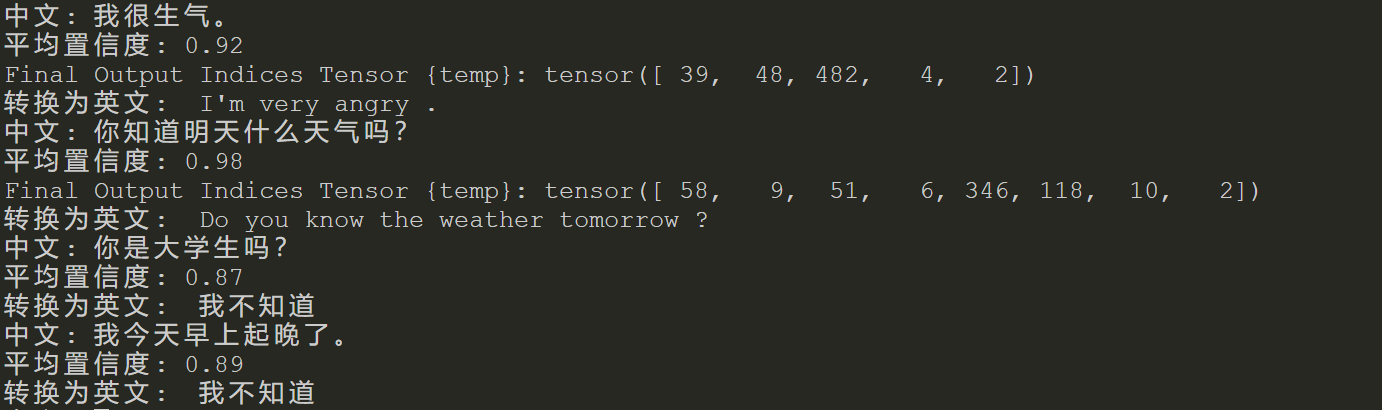
对于训练集数据：



对于测试集数据：



随机数据：



对于模型自信能预测的句子，用词和语法几乎没有问题，极少数情况下也会出错。缺点是能准确预测的数量太少，更多都是一些简单的句子结构预测结果较好。

(对于一些0.85-0.9置信度之间的数据，模型预测出来的结果往往只差一个单词的不匹配、缺失或增多，但也还是错误的，所以按照不知道来看。)

1. 公式验证
2. 中英文翻译 （一类书，翻译同类的其他书,IT）预训练
3. 大模型，人机对话