# 知识工程-作业10 英法中文翻译

2024214500 叶璨铭

## 代码与文档格式说明

本文档使用Jupyter Notebook编写，遵循Diátaxis 系统 Notebook实践 <https://nbdev.fast.ai/tutorials/best_practices.html，所以同时包括了实验文档和实验代码>。

本文档理论上支持多个格式，包括ipynb, docx, pdf 等。您在阅读本文档时，可以选择您喜欢的格式来进行阅读，建议您使用 Visual Studio Code (或者其他支持jupyter notebook的IDE, 但是VSCode阅读体验最佳) 打开 ipynb格式的文档来进行阅读。

为了记录我们自己修改了哪些地方，使用git进行版本控制，这样可以清晰地看出我们基于助教的代码在哪些位置进行了修改，有些修改是实现了要求的作业功能，而有些代码是对原本代码进行了重构和优化。我将我在知识工程课程的代码，在作业截止DDL之后，开源到 <https://github.com/2catycm/THU-Coursework-Knowledge-Engineering.git> ，方便各位同学一起学习讨论。

## 代码规范说明

在我们实现函数过程中，函数的docstring应当遵循fastai规范而不是numpy规范，这样简洁清晰，不会Repeat yourself。相应的哲学和具体区别可以看 <https://nbdev.fast.ai/tutorials/best_practices.html#keep-docstrings-short-elaborate-in-separate-cells>

为了让代码清晰规范，在作业开始前，使用 ruff format格式化助教老师给的代码;

alt text

很好，这次代码格式化没有报错。

Pylance 似乎也没有明显问题。

## 实验环境准备

采用上次的作业专属环境，为了跑通最新方法，使用3.12 和 torch 2.7

conda create -n assignments python=3.12  
conda activate assignments  
pip install -r ../requirements.txt  
pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu126  
pip install -U git+https://github.com/TorchRWKV/flash-linear-attention

本次作业似乎没有新的依赖，只是用到了 torch

## 原理回顾和课件复习

课上详细介绍了 概述、传统、统计、神经方法

概述中，注意到“平行语料库”这个概念，平行语料库（Parallel Corpus）是指在两个或多个语言之间具有对齐关系的文本集合。每对对应的文本在不同语言中表达了相同或相似的意义，因而可以用于语言间的比较、翻译模型的训练、语言学研究以及其他自然语言处理任务。罗塞塔石碑就是典型例子，罗塞塔石碑是一块刻有相同内容但用三种不同书写系统（古埃及象形文字、埃及民用文和古希腊文）的石碑。正是因为这些文本内容一致，学者们才能利用已知语言（古希腊文）的信息，逐步破译不明的古埃及象形文字，揭开了古埃及语言和文化的神秘面纱。

1947年，机器翻译认为是解读密码。1966年陷入低迷，1978年恢复。

难点是语言表达有歧义、文化有差异、翻译和知识、常识有关、解不唯一、新词和专有名词。

简单直接翻译方法直接替换已知的单词、短语、句子，然后调整顺序。

基于规则的方法用规则描述语法，对句子进行词法分析（把连续的字符序列划分成独立的词或符号（即“词元”或“标记”））、句法分析（在词法分析的基础上，利用预先定义的语法规则构造句子的句法结构（如语法树），确定不同词汇之间的组合规则和结构关系。）、语义分析（不仅关注词语的基本含义，还要判断它们在上下文中的语境作用，识别歧义、隐含意义和语义角色）。生成译文的句子结构（两个语言表达顺序不同），然后选择词法。 也叫作基于Transform的方法，这是独立分析两个语言的结构。缺点是语言写得不符合预定义的语法的时候处理不了。

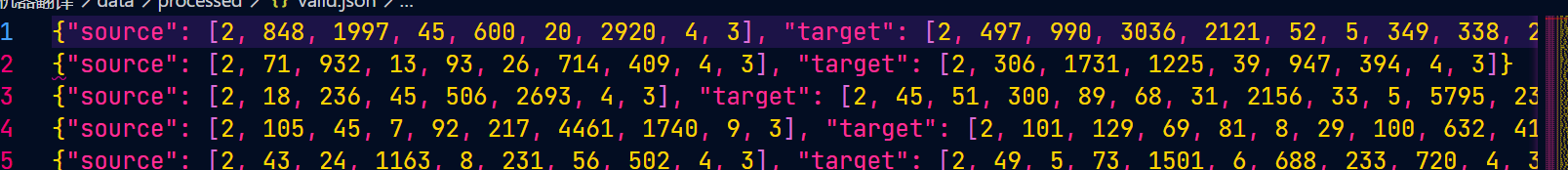
注意，基于规则的方法无需依赖双语平行语料。（？需要词典，词典不算吗）

基于实例的方法，会类比已有的标准翻译实例，然后拼凑新的翻译。

基于统计的翻译，使用噪声信道模型。从S翻译到T，认为是T经过噪声干扰变成了S。求 P(T|S)， 可以求 P(S|T) 和 P(T) 后者直接统计词频。

## 数据准备

助教已经帮我们preprocess好了数据，注意到有四个json文件。有两个是单词编码为id，词典对照。 train和valid是 jsonl 的格式， 一行是一个 句子翻译到一个句子。

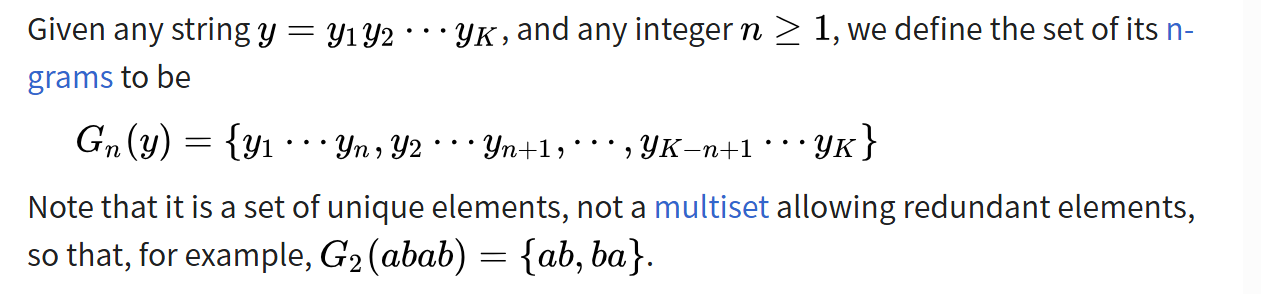


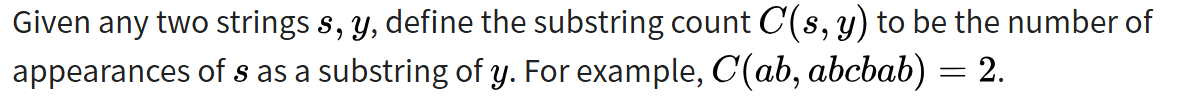
## 补充完成./metrics.py中BLEU的计算

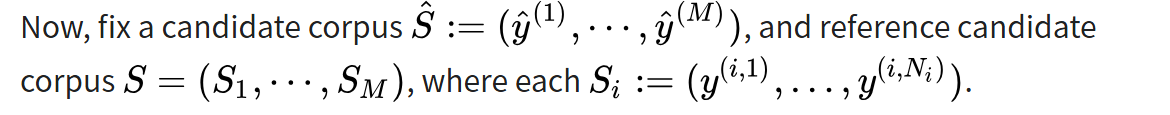
首先了解一下BLEU的概念, 参考课件和 <https://en.wikipedia.org/wiki/BLEU> 。

BLEU 全称 bilingual evaluation understudy，雙語替換評測，“Understudy”在这里指的是“替补演员”，在戏剧、表演等领域，指那些在正式演员因故无法出演时，能够随时顶替其角色的演员。

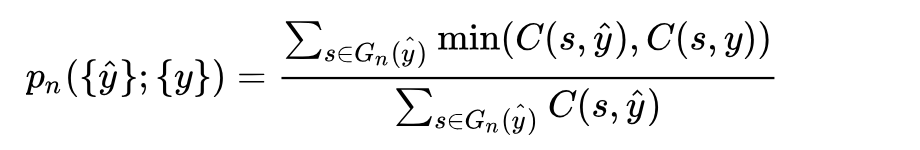
想要评测 Quality/correspondence/accuracy。 wiki说是2001, 课件是2002， IBM发明。不考虑可理解性、语法正确性，只考虑与参考答案（有一组， a set of good quality reference translations）像不像。输出[0,1]。数据集=语料库 corpus，有多个翻译预测和多组翻译参考答案。

首先需要指定 N-Gram 的 N，对于字符串，可以找到 其 N-Grams 的（不重复）集合。 

然后定义出现次数C 

S有M个预测答案，M组参考答案。 

首先定义 Modified N-Gram Precision，既然是Precision，不是Recall，所以就是从预测答案来看

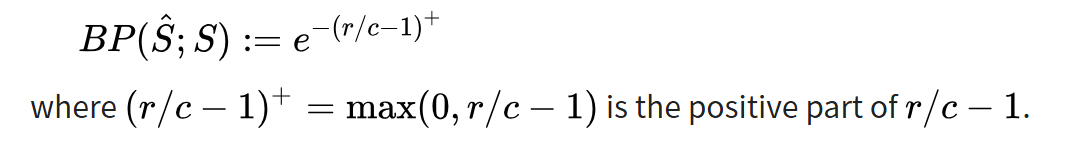
在预测的答案中，每一个N-Gram出现了很多次，对于每一个N-Gram而言，想看看在标准答案里面出现多少次，如果比我少，那我不准，我做多了，可能凭空翻译了新东西。有一组标准答案，所以里面对我最好的那个（出现我的ngram最多的那个）和我比。 

这个式子另一个角度看，是为了衡量，参考句子多少个n gram时候是在 候选句子中有的，有多少次。

有了 Modified N-Gram Precision 之后，BLEU引入 Brevity penalty 简洁性惩罚（不是惩罚之后变简洁，而是简洁的被惩罚）。

因为刚才的指标不恰当地（unduly）会奖励那种为了拿分全部N-Gram都说一遍（telegraphic）的模型。

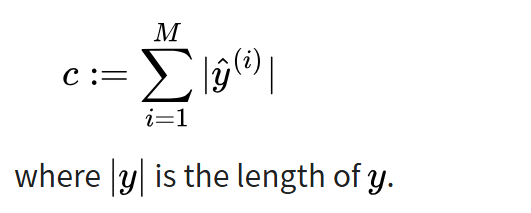
仔细看了看，我理解错了，惩罚的是有N-Gram，但是只说一遍，后面忘记说了的模型。

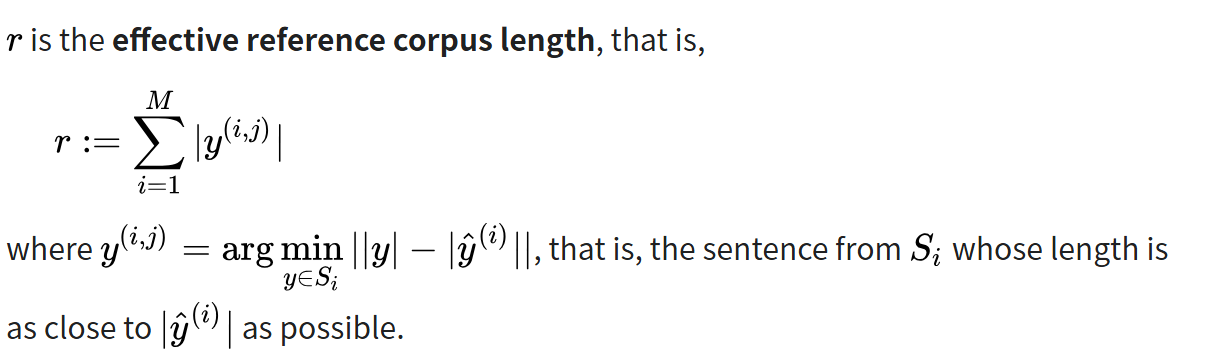


乍一看， r是 real 的长度（一组里面最接近c的那个），c是candidate的长度。

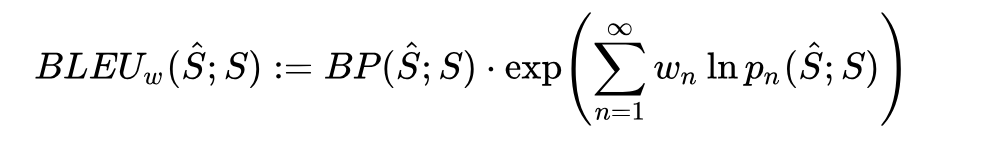
如果c比r长就不惩罚了（我说的那个问题好像不是这里解决）

实际上， 需要注意BP是对整个语料库算的，不是对单个句子！是求了个和！整体进行惩罚！





最终，BLEU 认为 很多 N Gram都重要，要加权算，所以枚举n=1到无穷，有wn分布来加权，得到



几何平均数是希望，模型不是只在一个N Gram上表现好，而是大部分都好。

原本论文只考虑 n=1, 2, 3, 4, w = 1/4 。

批评BLEU的意见指出，没有分词边界的语言，或者英语使用不同的token方案，会导致BLEU分数差异很大，不可比较。

现在我们来实现，首先观察助教给的函数签名，顿时发现了问题

candidate\_corpus: List[List[str]] # 形如 [[cand1\_token1, cand1\_token2, …], [cand2\_…], …] references\_corpus: List[List[str]] # 同样是 [[ref1\_token1, ref1\_token2, …], [ref2\_…], …]

首先每一个 str 是 一个 token， 不是 句子哦， List[str] 是一个句子。

第二，这里是一对一的，没有上面概念里面的 一个 candidate，多个 reference 的情况。

def bleu\_score(  
 candidate\_corpus: List[List[str]], # 候选翻译的token列表的列表。  
 references\_corpus: List[  
 List[str]  
 ], # 参考翻译的token列表的列表，数量应与候选一一对应。  
 max\_n=4, # 最大的ngram的数量，默认4。  
 weights: List[float] = [0.25]  
 \* 4, # 用于计算加权几何平均时的权重列表，长度应为max\_n。  
 verbose: bool = True,  
) -> float: # BLEU分数（0到1之间）。  
 """  
 计算候选翻译语料库和参考翻译语料库之间的BLEU分数。  
 """  
 assert len(candidate\_corpus) == len(references\_corpus), (  
 "候选翻译和参考翻译的数量必须一致。"  
 )  
 assert len(weights) == max\_n, "权重列表的长度必须等于最大的ngram数量。"  
  
 total\_clip\_count = [0] \* max\_n  
 total\_candidate\_ngrams = [0] \* max\_n  
 total\_candidate\_length = 0  
 total\_reference\_length = 0  
  
 for candidate, references in zip(candidate\_corpus, references\_corpus):  
 candidate\_ngrams = \_compute\_ngram\_counter(candidate, max\_n)  
 reference\_ngrams = [\_compute\_ngram\_counter(ref, max\_n) for ref in references]  
 max\_reference\_ngrams = collections.Counter()  
 for ref\_ngrams in reference\_ngrams:  
 for ngram, count in ref\_ngrams.items():  
 max\_reference\_ngrams[ngram] = max(max\_reference\_ngrams[ngram], count)  
  
 for n in range(1, max\_n + 1):  
 for ngram, count in candidate\_ngrams.items():  
 if len(ngram) == n:  
 total\_candidate\_ngrams[n - 1] += count  
 total\_clip\_count[n - 1] += min(count, max\_reference\_ngrams[ngram])  
  
 candidate\_length = len(candidate)  
 total\_candidate\_length += candidate\_length  
 reference\_lengths = [len(ref) for ref in references]  
 closest\_ref\_length = min(  
 reference\_lengths, key=lambda x: abs(x - candidate\_length)  
 )  
 total\_reference\_length += closest\_ref\_length  
  
 precisions = []  
 for clip\_count, candidate\_ngrams in zip(total\_clip\_count, total\_candidate\_ngrams):  
 if candidate\_ngrams == 0:  
 precisions.append(0)  
 else:  
 precisions.append(clip\_count / candidate\_ngrams)  
  
 if verbose:  
 print(f"Precisions: {precisions}")  
 print(f"Total candidate length: {total\_candidate\_length}")  
 print(f"Total reference length: {total\_reference\_length}")  
  
 if total\_candidate\_length == 0:  
 return 0  
  
 brevity\_penalty = (  
 1  
 if total\_candidate\_length >= total\_reference\_length  
 else math.exp(1 - total\_reference\_length / total\_candidate\_length)  
 )  
  
 if verbose:  
 print(f"Brevity penalty: {brevity\_penalty}")  
  
 log\_precisions = [math.log(p) if p > 0 else float("-inf") for p in precisions]  
 bleu = brevity\_penalty \* math.exp(  
 sum(w \* p for w, p in zip(weights, log\_precisions))  
 )  
  
 return bleu

简单测试一下。

from metrics import bleu\_score  
  
# 示例候选翻译语料库  
candidate\_corpus = [["the", "cat", "sat", "on", "the", "mat"], ["hello", "world"]]  
# 示例参考翻译语料库  
references\_corpus = [  
 [["the", "cat", "is", "sitting", "on", "the", "mat"]],  
 [["hello", "world"]],  
]  
  
score = bleu\_score(candidate\_corpus, references\_corpus)  
score

Precisions: [0.875, 0.6666666666666666, 0.25, 0.0]  
Total candidate length: 8  
Total reference length: 9  
Brevity penalty: 0.8824969025845955

0.0

可以发现，因为句子太短，没有 4-Gram! BLEU 分数为 0 的原因是 4-gram 的精确率为 0，导致其对数为负无穷（log(0) = -inf），最终加权平均后的指数部分为负无穷，使得整体结果为 0。这是符合数学定义的，并非代码错误。

测试用例中，第一个候选翻译 ["the", "cat", "sat", "on", "the", "mat"] 和参考翻译 ["the", "cat", "is", "sitting", "on", "the", "mat"] 的 4-gram 必然无法匹配（候选长度为 6，4-gram 数量为 3；参考长度为 7，4-gram 数量为 4），因此 4-gram 精确率为 0 是合理的。 BLEU 分数的数学定义中，只要任意 n-gram 的精确率为 0，其对数会拉低整个指数项，导致结果趋近于 0。这是正常现象。

我们可以让 max\_n=3

score = bleu\_score(candidate\_corpus, references\_corpus, max\_n=3, weights=[1 / 3] \* 3)  
score

Precisions: [0.875, 0.6666666666666666, 0.25]  
Total candidate length: 8  
Total reference length: 9  
Brevity penalty: 0.8824969025845955

0.464513981711853

## 阅读model/transformer.py并补充完成multihead attention

拿到助教给我们的代码，首先注意到分为 context 和 attention两个部分。

context, attention = None, None  
# TODO  
return context, attention

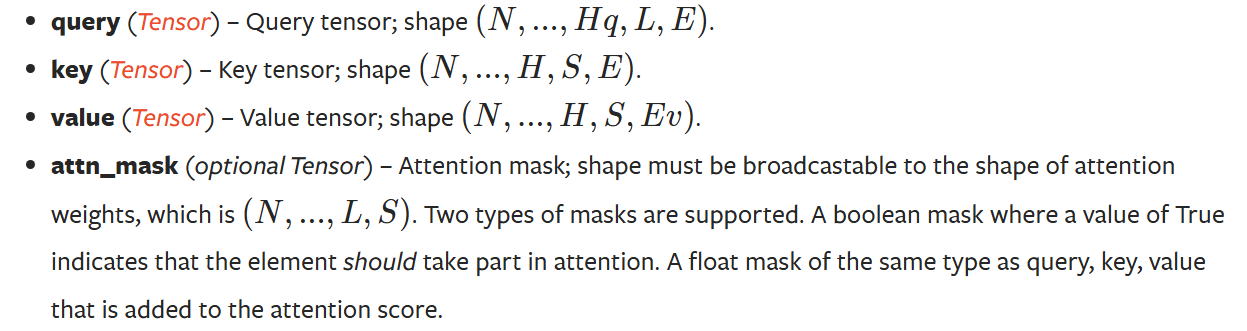
其实 context就是输出的hidden states, attention 分数是中间结果，可能要可视化吧。

由于维度比较高，难以思考，所以我们决定用 某次作业助教用到的 einops 来尝试实现 这个 attention。

import torch  
import torch.nn as nn  
from einops import einsum  
  
  
class ScaledDotProductAttention(nn.Module):  
 def \_\_init\_\_(self, dropout):  
 """实现Scaled Dot-Product Attention"""  
 super(ScaledDotProductAttention, self).\_\_init\_\_()  
 self.dropout = nn.Dropout(dropout)  
 self.softmax = nn.Softmax(dim=-1)  
  
 def forward(  
 self,  
 q: torch.Tensor,  
 k: torch.Tensor,  
 v: torch.Tensor, # [batch\_size, num\_heads, seq\_len, hidden\_size / num\_heads]  
 mask: torch.Tensor # [batch\_size, 1, seq\_len, seq\_len]  
 ):  
 """  
 output:  
 - context: 输出值  
 - attention: 计算得到的注意力矩阵  
 """  
 d\_k = q.size(-1)  
 sqrt\_d\_k = torch.sqrt(torch.tensor(d\_k, dtype=torch.float32))  
 # 计算点积，使用einops的einsum  
   
 # b,h 独立，做乘法的是 i,j，  
 # d 维度会进行求和操作，因为它只在输入中出现，不在输出中出现（被reduce掉了）。  
 scores = einsum(q, k, 'b h i d, b h j d -> b h i j') / sqrt\_d\_k   
  
 if mask is not None:  
 scores = scores.masked\_fill(mask == 0, -1e9) # mask到的位置不能被 attention 注意到，本来是赋值为0，但是待会有softmax，应该给-inf。  
  
 attention = self.softmax(scores)  
 attention = self.dropout(attention)  
  
 # 计算context，使用einops的einsum  
 # b,h 独立，i， d乘法；对 j 求和  
 context = einsum(attention, v, 'b h i j, b h j d -> b h i d')  
 return context, attention

注意掩码约定为masks 的形状为 (batch, i, j)，i：代表 查询（Query）的位置（例如，序列中每个 token 作为查询时的索引）；j：代表 键（Key）的位置（例如，序列中每个 token 作为键时的索引）。 掩码 masks[b, i, j] 表示：第 b 个批次中，查询位置 i 能否关注键位置 j（通常为 0 或 1，或用于缩放的权重）。

decoder中，不允许 i 关注 >i 的 j， 所以 mask 是 下三角矩阵，i可以大于j，j不可以大于i。

参考 torch 官网文档 <https://pytorch.org/docs/stable/generated/torch.nn.functional.scaled_dot_product_attention.html> 

我们的掩码约定和其一样。

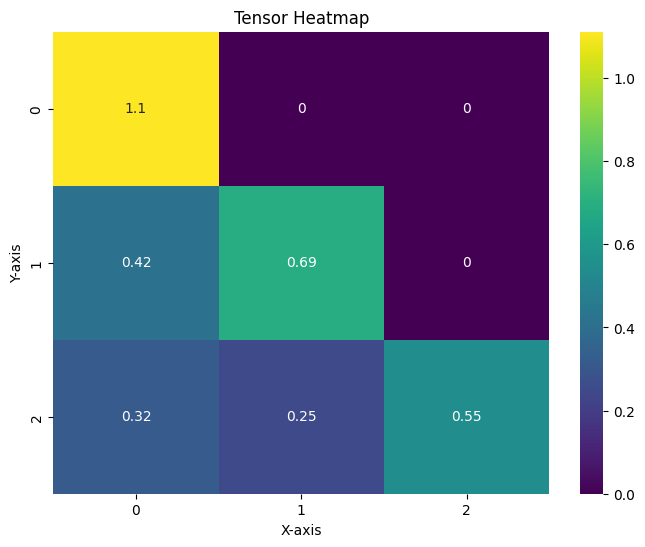
我们可以简单测试一下我们的代码

# 定义参数  
batch\_size = 2  
num\_heads = 4  
seq\_len = 3  
hidden\_size\_per\_head = 8  
dropout = 0.1  
  
# 创建模拟输入  
q = torch.randn(batch\_size, num\_heads, seq\_len, hidden\_size\_per\_head)  
k = torch.randn(batch\_size, num\_heads, seq\_len, hidden\_size\_per\_head)  
v = torch.randn(batch\_size, num\_heads, seq\_len, hidden\_size\_per\_head)  
mask = torch.tril(torch.ones(batch\_size, 1, seq\_len, seq\_len))  
  
# 初始化模型  
attention\_module = ScaledDotProductAttention(dropout)  
  
# 前向传播  
context, attention = attention\_module(q, k, v, mask)  
  
print(f"Context shape: {context.shape}")  
print(f"Attention shape: {attention.shape}")  
print("Test passed!")

Context shape: torch.Size([2, 4, 3, 8])  
Attention shape: torch.Size([2, 4, 3, 3])  
Test passed!

我们可以可视化一下 attention score

import numpy as np  
import seaborn as sns  
import matplotlib.pyplot as plt  
  
# 使用seaborn绘制热力图  
plt.figure(figsize=(8, 6))  
sns.heatmap(attention[0, 0], annot=True, cmap='viridis')  
plt.title('Tensor Heatmap')  
plt.xlabel('X-axis')  
plt.ylabel('Y-axis')  
plt.show()



现在继续实现多头注意力

class MultiHeadAttention(nn.Module):  
 def \_\_init\_\_(self, hidden\_size, num\_heads, dropout):  
 """实现Multi-Head Attention"""  
 super(MultiHeadAttention, self).\_\_init\_\_()  
 self.hidden\_size = hidden\_size  
 self.num\_heads = num\_heads  
 self.dropout = dropout  
 self.linear\_q = nn.Linear(hidden\_size, hidden\_size)  
 self.linear\_k = nn.Linear(hidden\_size, hidden\_size)  
 self.linear\_v = nn.Linear(hidden\_size, hidden\_size)  
 self.linear = nn.Linear(hidden\_size, hidden\_size)  
 self.scaled\_dot\_product\_attention = ScaledDotProductAttention(dropout)  
 self.dropout\_layer = nn.Dropout(dropout)  
 self.layer\_norm = nn.LayerNorm(hidden\_size)  
  
 def forward(  
 self,  
 q: torch.Tensor,  
 k: torch.Tensor,  
 v: torch.Tensor, # [batch\_size, num\_heads, seq\_len, hidden\_size / num\_heads]  
 mask: torch.Tensor # [batch\_size, 1, seq\_len, seq\_len]  
 ):  
 residual = q  
 batch\_size = q.size(0)  
  
 # 线性变换  
 q = self.linear\_q(q)  
 k = self.linear\_k(k)  
 v = self.linear\_v(v)  
  
 # 分割成多个头  
 head\_dim = self.hidden\_size // self.num\_heads  
 q = q.view(batch\_size, -1, self.num\_heads, head\_dim).transpose(1, 2)  
 k = k.view(batch\_size, -1, self.num\_heads, head\_dim).transpose(1, 2)  
 v = v.view(batch\_size, -1, self.num\_heads, head\_dim).transpose(1, 2)  
  
 # 应用缩放点积注意力  
 context, attention = self.scaled\_dot\_product\_attention(q, k, v, mask)  
  
 # 拼接多个头的输出  
 context = context.transpose(1, 2).contiguous().view(batch\_size, -1, self.hidden\_size)  
  
 # 通过线性层  
 output = self.linear(context)  
 output = self.dropout\_layer(output)  
  
 # 残差连接和层归一化  
 output = self.layer\_norm(output + residual)  
  
 return output, attention

# 定义参数  
batch\_size = 2  
seq\_len = 3  
hidden\_size = 8  
num\_heads = 4  
dropout = 0.1  
  
# 创建模拟输入  
q = torch.randn(batch\_size, seq\_len, hidden\_size)  
k = torch.randn(batch\_size, seq\_len, hidden\_size)  
v = torch.randn(batch\_size, seq\_len, hidden\_size)  
mask = torch.tril(torch.ones(batch\_size, 1, seq\_len, seq\_len))  
  
# 初始化多头注意力模块  
multi\_head\_attention = MultiHeadAttention(hidden\_size, num\_heads, dropout)  
  
# 前向传播  
output, attention = multi\_head\_attention(q, k, v, mask)  
  
# 打印输出形状  
print(f"Output shape: {output.shape}")  
print(f"Attention shape: {attention.shape}")  
print("Test passed!")

Output shape: torch.Size([2, 3, 8])  
Attention shape: torch.Size([2, 4, 3, 3])  
Test passed!

## 运行./main.py

首先我们遇到了错误



我以为mask应该是 20x20的，但是这里实际上不是。

我们反过来，首先确认一下 source\_mask 和 target\_mask 是什么？在代码中是怎么产生的？首先我们打log出来，看到结果

Encoder source\_mask.shape is torch.Size([32, 15])  
scores.shape torch.Size([32, 12, 15, 15])  
mask.shape torch.Size([32, 1, 15])

首先 source\_mask 通常是一个布尔类型的张量，形状为 (batch\_size, seq\_len)。在填充位置上的值为 False，非填充位置上的值为 True。这个是为了防止用到padding token的信息。

target\_mask 要结合填充掩码和未来信息掩码。

我们整体阅读一下代码，追踪 mask 的来源

一开始是在 util.py

source\_mask 和 target\_mask 由 util.py 中的 collate\_fn 生成，本质是填充掩码（Padding Mask），用于标记序列中的填充位置（值为 0）和有效位置（值为 1）。生成逻辑：通过 pad\_list 函数对批次内的序列进行填充，有效位置生成 1，填充位置生成 0，最终转换为 PyTorch 张量。

在推理阶段（mode="eval"）： source\_mask 同样由 collate\_fn 生成（处理验证集数据）。 但是target\_mask 在 greedy\_decoder 中动态生成，为全 1 张量（torch.ones\_like(decoder\_input)），因为推理时逐词生成，无需遮盖未来信息（通过循环控制生成停止条件）。

关键调用链： 数据加载 → collate\_fn 生成填充掩码 → Transformer 模块传递掩码 → MultiHeadAttention 处理掩码形状 → ScaledDotProductAttention 使用掩码。

这中间还有关键的一步 ，

class Encoder(nn.Module):  
 ...  
 def forward(self, source\_ids, source\_mask):  
 ...  
 source\_mask = source\_mask.unsqueeze(1)  
 ...  
...  
class Decoder(nn.Module):  
 ...  
 def forward(self, target\_ids, encoder\_output, source\_mask, target\_mask):  
 ...  
 source\_mask = source\_mask.unsqueeze(1)  
 target\_mask = target\_mask.unsqueeze(1)  
 ...

所以实际上助教给的注释有误，

原本我们拿到的其实是

batch 个 [1, ..., 1, 0, ..., 0] 这种形式的 mask, unsqueeze 之后变成了 batch 个 [[1, ..., 1, 0, ..., 0]] 而已，并不是代码中规定的 mask: torch.Tensor # [batch\_size, 1, seq\_len, seq\_len]

如果原来是 [1, 0], 应该变成 [[1, 0] [1,0]]

另一个要怀疑的问题是，是否要对 num\_heads 做处理，这个其实不用， multihead 的 mask 直接往下面传递就可以了，可以利用 PyTorch 的广播（broadcast）机制把头维度自动扩张到 num\_heads 。

所以助教给的 transformer.py 的Encoder 和Decoder 逻辑不足，我决定从这里入手修复代码问题。

对于 Decoder而言，如果mask是 [1, 1, 0], 那就是 可以11，21， 22， 其他都是0，那矩阵应该是 [[1, 0, 0], [1, 1, 0], [0, 0, 0] ]

如果是 Encoder， 如果 mask 是 [1, 1, 0], 那 意味着最后一行一列是0，其他都是1。

import torch  
from functools import partial  
  
def make\_attention\_mask(pad\_mask: torch.Tensor, is\_decoder: bool) -> torch.Tensor:  
 """  
 生成 Encoder 或 Decoder 的 [batch, 1, seq, seq] 自注意力 mask。  
  
 参数：  
 pad\_mask: [batch, seq\_len]，1 表示有效 token，0 表示 padding。  
 is\_decoder: True 则生成下三角 causal mask，否则全 1。  
  
 返回：  
 mask: [batch, 1, seq\_len, seq\_len]，bool dtype。  
 """  
 batch, seq\_len = pad\_mask.shape  
 device = pad\_mask.device  
  
 # —— 第一步：底板 mask（bool）  
 if is\_decoder:  
 # 下三角 causal  
 base = torch.tril(torch.ones((seq\_len, seq\_len),   
 dtype=torch.bool,   
 device=device))  
 else:  
 # 全 1  
 base = torch.ones((seq\_len, seq\_len),   
 dtype=torch.bool,   
 device=device)  
  
 # 扩展到 batch 维度  
 base = base.unsqueeze(0).expand(batch, seq\_len, seq\_len) # [b, seq, seq]  
  
 # —— 第二步：根据 pad\_mask 屏蔽行 & 列  
 # row\_ok[b,i,j] = pad\_mask[b,i]  
 row\_ok = pad\_mask.bool().unsqueeze(2).expand(batch, seq\_len, seq\_len)  
 # col\_ok[b,i,j] = pad\_mask[b,j]  
 col\_ok = pad\_mask.bool().unsqueeze(1).expand(batch, seq\_len, seq\_len)  
  
 mask2d = base & row\_ok & col\_ok # [b, seq, seq]  
 return mask2d.unsqueeze(1) # -> [b, 1, seq, seq]  
  
make\_decoder\_mask = partial(make\_attention\_mask, is\_decoder=True)  
make\_encoder\_mask = partial(make\_attention\_mask, is\_decoder=False)

pad\_mask = torch.tensor([[1, 1, 0],  
 [1, 0, 1]], dtype=torch.uint8)  
make\_decoder\_mask(pad\_mask), make\_encoder\_mask(pad\_mask)

(tensor([[[[ True, False, False],  
 [ True, True, False],  
 [False, False, False]]],  
   
   
 [[[ True, False, False],  
 [False, False, False],  
 [ True, False, True]]]]),  
 tensor([[[[ True, True, False],  
 [ True, True, False],  
 [False, False, False]]],  
   
   
 [[[ True, False, True],  
 [False, False, False],  
 [ True, False, True]]]]))

表面上我们彻底解决了问题，但是实际上，有个更加棘手的问题在于，mask实际上也不是 [batch, 1, seq\_len, seq\_len] 而是[batch, 1, query\_seq\_len, key\_seq\_len], 因为 需要考虑 Decoder中 cross Attention的情况！

这下我们陷入了绝境，我们首先仔细想一想，最终我们搞半天mask的目的是什么？是为了Attention分数 <https://pytorch.org/docs/stable/generated/torch.Tensor.masked_fill_.html#torch.Tensor.masked_fill_>

实际上unsqueeze两次可能就对了，直觉上！我们来验证一下

首先我们推导 self attention

scores 为 b h enc\_seq enc\_seq

mask 首先是 b enc\_seq 然后 变成 b, 1, 1, enc\_seq 广播相当于 复制 h, enc\_seq 份， 把 查询到的 enc\_seq 变0

cross attention， 也是这样！(decoder 中先自己self，然后再对enc做cross)

scores 为 b h dec\_seq enc\_seq dec 去 查询 enc mask 是 对 enc\_seq 做的，为什么不用对 dec\_seq做呢，这是因为 dec\_seq 没生成到的位置直接不存在，然后这里查询的时候查询的是左边，随便查。

那么 mask 也是 b, 1, 1, enc\_seq 同样 把 每一个dec\_seq（1广播过来的）想attend的enc\_seq 都进行了检验。

这下突然发现问题了，decoder中的self attention应该有causual才是对的，这里好像没考虑？

仔细检查代码， self.attention\_1(x, x, x, target\_mask) 按照广播的逻辑，传入的 target\_mask 是 b, 1, 1, dec\_seq dec\_seq 是慢慢生成的

这里面确实没有限制 causual，新的token确实没有问题，但是旧的token之间可以互相attend到了啊？

其实不然，我们这个问题是好问题，这就是K V Cache！ 前面的token自己乱attend，得到的结果没有用（和前面生成过的一样）

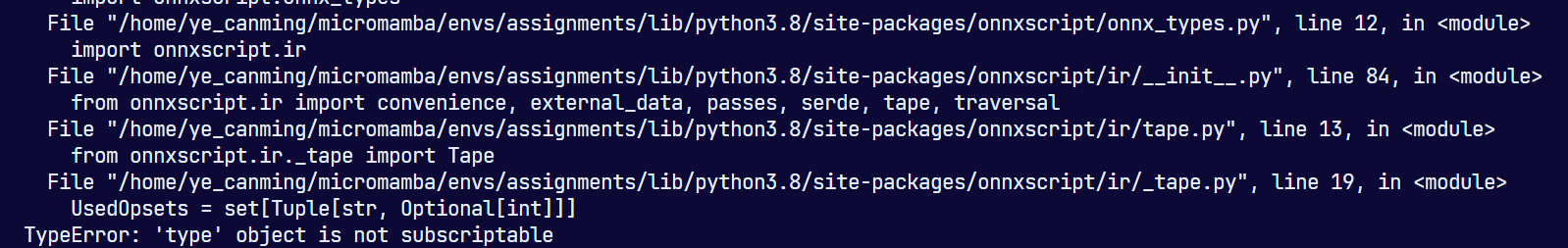
确认推导正确后，我们终于可以跑通啦！但是一跑这个速度让我傻眼了，我们的main.py好像是在cpu上跑！

一不做二不休，我直接改到accelerator上面跑，加上 onnxrt 编译

pip install numpy onnx onnxscript onnxruntime-training

from accelerate import Accelerator  
accelerator = Accelerator(  
 dynamo\_backend = "onnxrt"  
)  
...  
lr\_scheduler, optimizer, dataloader, criterion, model = accelerator.prepare(  
 lr\_scheduler, optimizer, dataloader, criterion, model  
 )

然而我们这次作业因为服务器原因更换了环境为Python 3.8， onnx直接崩溃。



无奈卸载。我们不用了。我们就 accelerate 多卡就好。

accelerate cofnig  
accelerate launch main.py

但是正好没多卡了，还是单卡算了。



训练完毕

现在是评估阶段，注意device问题

encoder\_output = model.encoder(source\_ids, source\_mask)  
target\_mask = torch.ones\_like(decoder\_input).to(decoder\_input.device)

修改了不合理的代码后，终于能在GPU上跑起来了， 原本CPU不知道要跑到猴年马月。

python main.py --mode eval

但是仍然需要4h，太慢了



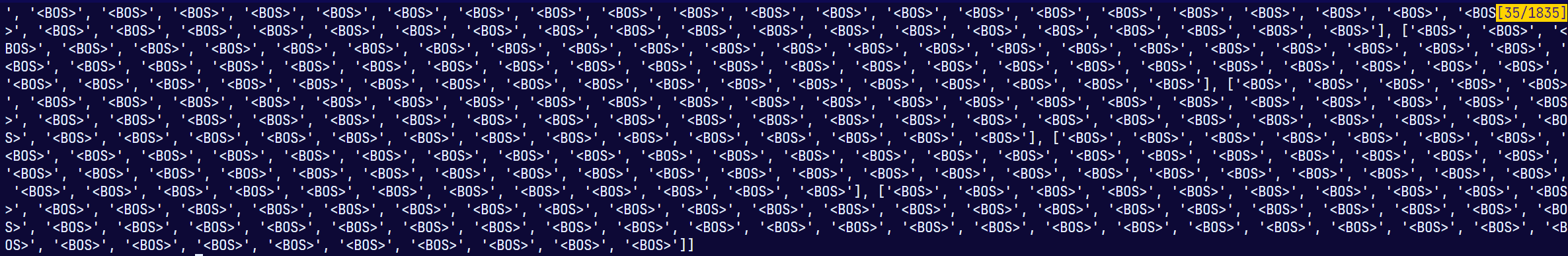
因为代码写得有问题，batch size 被强制设置为了 1。

其实decoder 的token生成逻辑有误，我们再次强制修改代码。

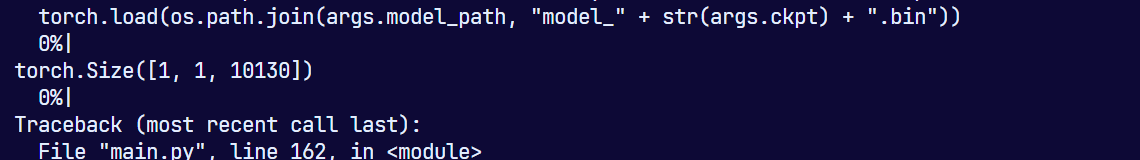
我们发现生成的逻辑严重被限制了，因为有结束生成，所以没法batch

没办法了，随机采样一些 dataloader 来评估我们的模型吧。

from torch.utils.data import Dataset, Subset, DataLoader  
dataset = MyDataset(args.valid\_data\_path)  
subset\_indices = range(1000)   
dataset = Subset(dataset, subset\_indices)

首先采样了五个，立马发现了utils.py的函数错漏百出 居然全部给了 BOS 

原因是维度没有做对

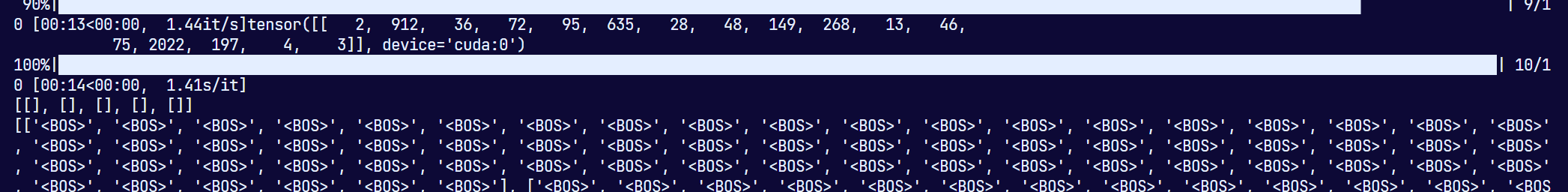


把代码 next\_token = torch.argmax(logits[:, -1, :]).item() 改成 next\_token = torch.argmax(logits[:, :, -1]).item()

这样逻辑仍然是错误的，目标是找 argmax，不是在里面-1，这里代码错漏摆出。

正确答案应该是 next\_token = torch.argmax(logits.flatten()).item()

很快我们发现main.py 也错漏百出，无法信任。target\_ids明明是存在的

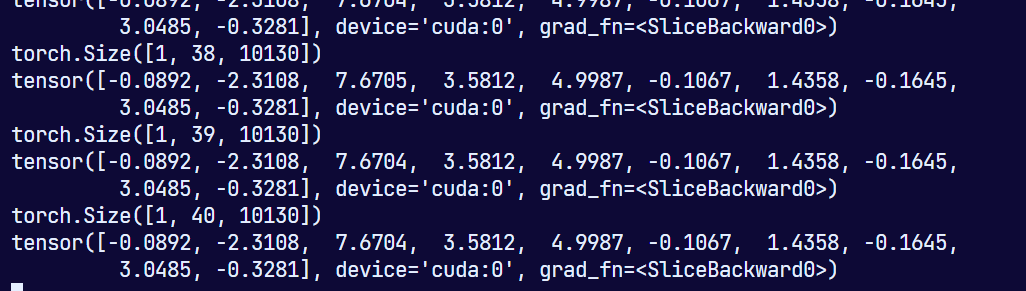
解码的时候直接解码失败了，连实例都没有。 

这是因为粗心的main.py 再次搞错了代码 references\_corpus.append( decode(config\_dict["target\_vocab\_path"], target\_ids[1:-1]) )

[1:-1 ]直接就把答案搞没了，很无语。

references\_corpus.append( decode(config\_dict["target\_vocab\_path"], target\_ids[0][1:-1]) )

现在至少参考答案对了 但是预测输出问题很大，2的概率太大了



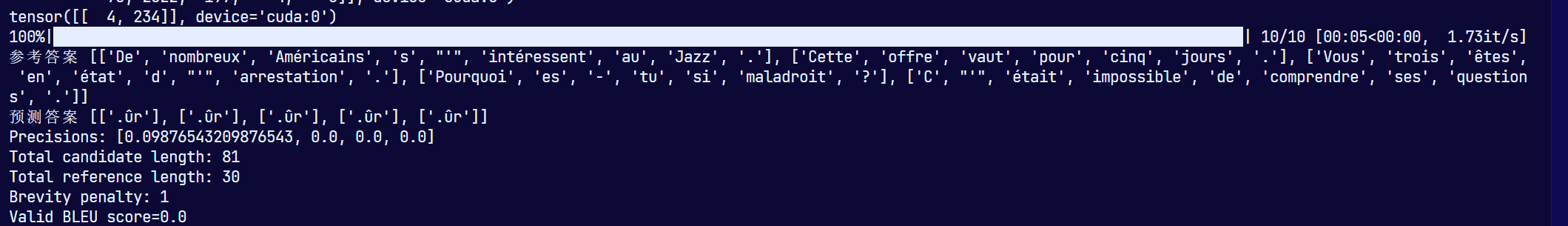
我们尝试用 top2 来代替 top1 看看会保护好有改进。q

next\_token = torch.topk(logits.flatten(), k=2).indices[1].item()

最终整体逻辑是

logits = logits[:, -1, :]  
 assert len(logits.flatten()) == len(target\_vocab)  
 next\_token = torch.topk(logits.flatten(), k=2).indices[1].item()

修复了 大量错误答案之后，我们终于勉强得到一个评测结果



可以看到模型明显学坏了，多个epoch让模型只学会了词频，最大的完全就是那个奇怪的单词，我们前面的辛苦训练了几个小时其实参数不好。

但是整体的代码流程是对的，只是需要调参。