

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

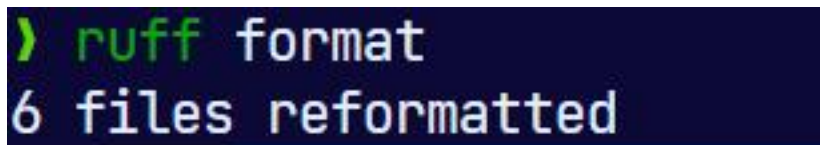
<https://github.com/2catycm/THU-Coursework-Knowledge-Engineering.git>，方便各位同学一起学习讨论。

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；



```
> ruff format
6 files reformatted
```

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。

```
1 {"source": [2, 848, 1997, 45, 600, 20, 2920, 4, 3], "target": [2, 497, 990, 3036, 2121, 52, 5, 349, 338, 2]
2 {"source": [2, 71, 932, 13, 93, 26, 714, 409, 4, 3], "target": [2, 306, 1731, 1225, 39, 947, 394, 4, 3]}
3 {"source": [2, 18, 236, 45, 506, 2693, 4, 3], "target": [2, 45, 51, 300, 89, 68, 31, 2156, 33, 5, 5795, 23]
4 {"source": [2, 105, 45, 7, 92, 217, 4461, 1740, 9, 3], "target": [2, 101, 129, 69, 81, 8, 29, 100, 632, 41]
5 {"source": [2, 43, 24, 1163, 8, 231, 56, 502, 4, 3], "target": [2, 49, 5, 73, 1501, 6, 688, 233, 720, 4, 3]}
```

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。

Given any string $y = y_1 y_2 \cdots y_K$, and any integer $n \geq 1$, we define the set of its **n-grams** to be

$$G_n(y) = \{y_1 \cdots y_n, y_2 \cdots y_{n+1}, \cdots, y_{K-n+1} \cdots y_K\}$$

Note that it is a set of unique elements, not a **multiset** allowing redundant elements, so that, for example, $G_2(abab) = \{ab, ba\}$.

然后定义出现次数 C

Given any two strings s, y , define the substring count $C(s, y)$ to be the number of appearances of s as a substring of y . For example, $C(ab, abcbab) = 2$.

S 有 M 个预测答案，M 组参考答案。

Now, fix a candidate corpus $\hat{S} := (\hat{y}^{(1)}, \cdots, \hat{y}^{(M)})$, and reference candidate corpus $S = (S_1, \cdots, S_M)$, where each $S_i := (y^{(i,1)}, \cdots, y^{(i,N_i)})$.

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。

$$p_n(\{\hat{y}\}; \{y\}) = \frac{\sum_{s \in G_n(\hat{y})} \min(C(s, \hat{y}), C(s, y))}{\sum_{s \in G_n(\hat{y})} C(s, \hat{y})}$$

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。

$$BP(\hat{S}; S) := e^{-(r/c-1)^+}$$

where $(r/c - 1)^+ = \max(0, r/c - 1)$ is the positive part of $r/c - 1$.

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！

$$c := \sum_{i=1}^M |\hat{y}^{(i)}|$$

where $|y|$ is the length of y .

r is the **effective reference corpus length**, that is,

$$r := \sum_{i=1}^M |y^{(i,j)}|$$

where $y^{(i,j)} = \arg \min_{y \in S_i} ||y| - |\hat{y}^{(i)}||$, that is, the sentence from S_i whose length is as close to $|\hat{y}^{(i)}|$ as possible.

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 w_n 分布来加权，得到

$$BLEU_w(\hat{S}; S) := BP(\hat{S}; S) \cdot \exp \left(\sum_{n=1}^{\infty} w_n \ln p_n(\hat{S}; S) \right)$$

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_corpu
s):
    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_can
didate_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 ($\log(0) = -\text{inf}$)，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, weight
s=[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

- **query** (*Tensor*) – Query tensor; shape (N, \dots, H_q, L, E) .
- **key** (*Tensor*) – Key tensor; shape (N, \dots, H, S, E) .
- **value** (*Tensor*) – Value tensor; shape (N, \dots, H, S, E_v) .
- **attn_mask** (*optional Tensor*) – Attention mask; shape must be broadcastable to the shape of attention weights, which is (N, \dots, L, S) . Two types of masks are supported. A boolean mask where a value of True indicates that the element *should* take part in attention. A float mask of the same type as query, key, value that is added to the attention score.

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!
```

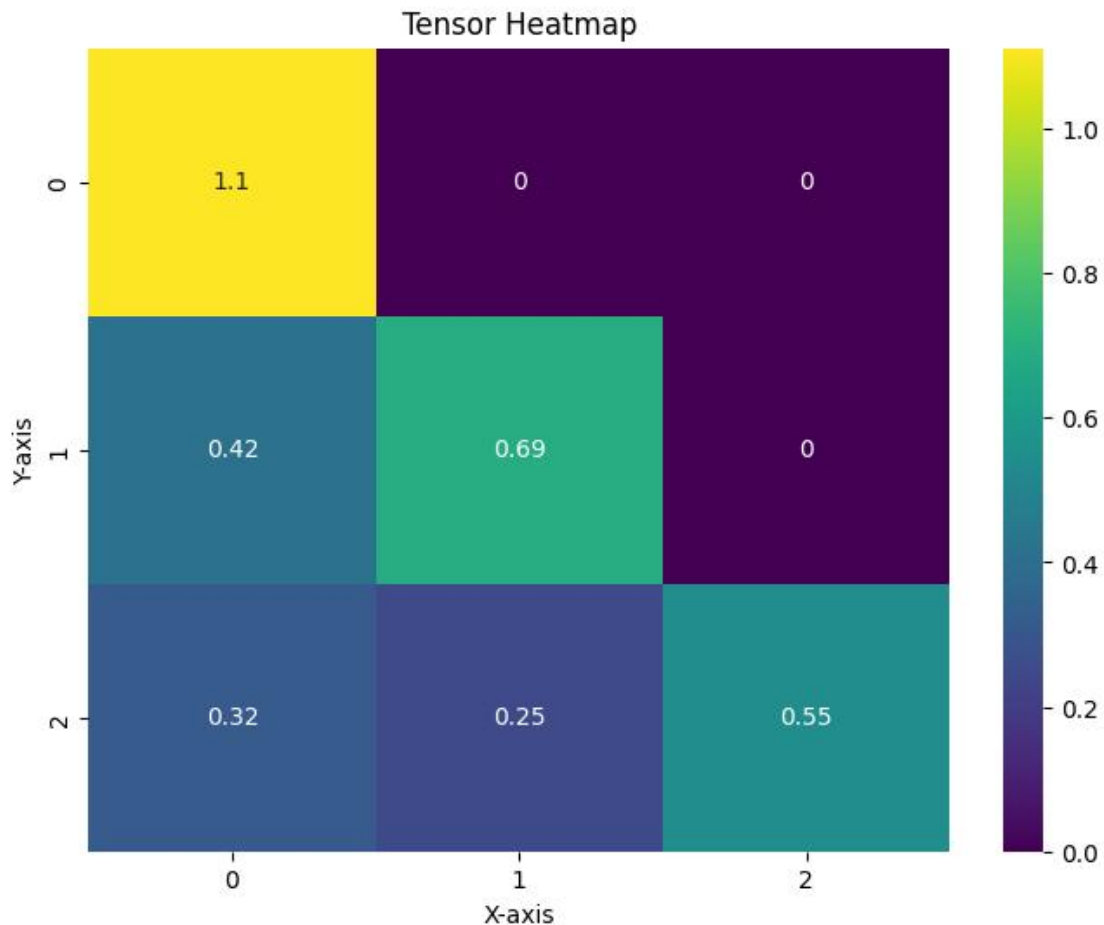
我们可以可视化一下 attention score

```

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

```

> python main.py
0%|          | 0/4800 [00:00<?, ?it/s]scores.shape torch
.Size([32, 12, 20, 20])
mask.shape torch.Size([32, 1, 20])
0%|          | 0/4800 [00:00<?, ?it/s]
Traceback (most recent call last):
  File "main.py", line 105, in <module>
    output = model(source_ids, source_mask, target_ids, target_mask)
  File "/home/ye_canning/micromamba/envs/assignments/lib/python3.8/site-packages/torch/nn/modules/module.py", line 1553, in _wrapped_call_impl
    return self._call_impl(*args, **kwargs)
  File "/home/ye_canning/micromamba/envs/assignments/lib/python3.8/site-packages/torch/nn/modules/module.py", line 1562, in _call_impl
    return forward_call(*args, **kwargs)
  File "/home/ye_canning/repos/assignments/THU-Coursework-Knowledge-Engineering/10.英法机器翻译/model/transformer.py", line 29, in forward
    encoder_output = self.encoder(source_ids, source_mask)

```

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 `causal` 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 `causal`，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.pre
pare(
    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。

```
File "/home/ye_canning/micromamba/envs/assignments/lib/python3.8/site-packages/onnxscript/onnx_types.py", line 12, in <module>
    import onnxscript.ir
File "/home/ye_canning/micromamba/envs/assignments/lib/python3.8/site-packages/onnxscript/ir/__init__.py", line 84, in <module>
    from onnxscript.ir import convenience, external_data, passes, serde, tape, traversal
File "/home/ye_canning/micromamba/envs/assignments/lib/python3.8/site-packages/onnxscript/ir/tape.py", line 13, in <module>
    from onnxscript.ir._tape import Tape
File "/home/ye_canning/micromamba/envs/assignments/lib/python3.8/site-packages/onnxscript/ir/_tape.py", line 19, in <module>
    UsedOpsets = set[Tuple[str, Optional[int]]]
TypeError: 'type' object is not subscriptable
```

无奈卸载。我们不用了。我们就 `accelerate` 多卡就好。

```
accelerate config
accelerate launch main.py
```

但是正好没多卡了, 还是单卡算了。

```
100%| 4795/4800 [07:16:00:00, 4797/4800 [07:16:00:00, 4797/4800 [07:16:00:00, 4797/4800 [07:16:00:00, 4799/4800 [07:17:00:00, 4799/4800 [07:17:00:00, 4800/4800 [07:17:00:00, 4800/4800 [07:17:00:00,
10.98it/s, loss=0.342]
100%| 4800/4800 [09:42:00:00, 8.25it/s, loss=0.409]
100%| 4800/4800 [09:34:00:00, 8.35it/s, loss=0.514]
100%| 4800/4800 [09:39:00:00, 8.28it/s, loss=0.382]
8%| 394/4800 [00:57:11:54, 6.16it/s, loss=0.439]
```

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

```
new file: Please open an issue on Github for any issues related to this experimental feature.
torch.load(os.path.join(args.model_path, "model_" + str(argsckpt) + ".bin"))
0%| 45/17065 [00:49:44:09:36, 1.16it/s]
0%| 45/17065 [00:49:45:14:32, 1.11it/s]
```

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`

