

# Transformer 原理与实现

💡 近年来，**Transformer 模型**因其出色的序列建模能力和并行计算优势，成为自然语言处理领域的核心架构之一。与传统的循环神经网络（RNN）或卷积神经网络（CNN）相比，Transformer 基于 **自注意力机制（Self-Attention）** 来捕捉序列中任意位置的依赖关系，使得模型能够并行训练，并在捕捉长距离依赖和全局上下文信息上表现更强。《[The Annotated Transformer](#)》是哈佛大学NLP团队写的一个带注释的 PyTorch 实现教程，它通过逐行代码的形式复现了论文《[Attention Is All You Need](#)》中的 Transformer 模型。

本篇笔记基于该教程进行整理和总结(稍有不同)，旨在：

1. 梳理 Transformer 的核心组件及其原理，包括 **多头自注意力 (Multi-Head Attention)**、**前馈网络 (Feed Forward)**、**位置编码 (Positional Encoding)** 等。
2. 结合代码与公式，帮助读者理解模型的实现细节。
3. 展示模型在小规模翻译任务上的训练、推理和评估方法，为进一步学习大型预训练模型（如 BERT、GPT 系列）打下基础。

## 环境配置要求

🐱 需要安装的库在 **requirements.txt** 文件

```
--find-links https://download.pytorch.org/whl/torch\_stable.html pandas==1.3.5  
torch==1.11.0+cu113 torchdata==0.3.0 torchtext==0.12 spacy==3.2 altair==4.1  
jupyter==1.13 flake8 black GPUUtil wandb
```

代码块

```
1 #此外如果先前没有安装下面的，需要自行安装  
2 !python -m spacy download de_core_news_sm #下载并安装 spaCy 的德语小模型  
3 !python -m spacy download en_core_web_sm #下载并安装 spaCy 的英语小模型
```

## 预备内容

### 导入库和函数

代码块

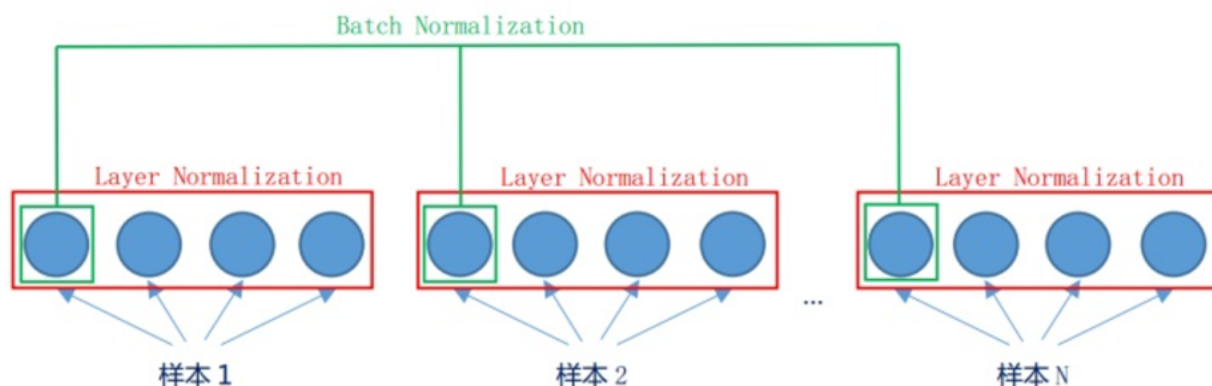
```
1  # 导入操作系统相关的工具，用于文件路径处理等
2  import os
3  from os.path import exists  # 判断文件/文件夹是否存在
4  # 导入 PyTorch 核心库
5  import torch
6  import torch.nn as nn  # 神经网络模块，包含各种层和结构
7  from torch.nn.functional import log_softmax, pad  # 函数式 API: 对数Softmax和张量填充
8  # 导入常用工具库
9  import math  # 数学函数
10 import copy  # 对象拷贝（深拷贝/浅拷贝）
11 import time  # 计时工具（常用于训练耗时统计）
12 # 学习率调度器，用于自定义动态调整学习率（如Transformer的 warmup 策略）
13 from torch.optim.lr_scheduler import LambdaLR
14 # 数据处理与可视化
15 import pandas as pd  # 数据分析工具
16 import altair as alt  # 可视化工具，绘制图表（如训练曲线、注意力可视化）
17 # TorchText: 文本处理库
18 from torchtext.data.functional import to_map_style_dataset  # 转换为 map-style 数据集
19 from torch.utils.data import DataLoader  # 数据加载器，批量处理和多线程读取
20 from torchtext.vocab import build_vocab_from_iterator  # 从迭代器构建词表
21 import torchtext.datasets as datasets  # 内置 NLP 数据集（如 Multi30k）
22 # NLP 工具
23 import spacy  # 用于分词和预处理（支持德语和英语）
24
25 # GPU 工具
26 import GPUtil  # GPU 使用率和显存监控工具
27
28 # 警告过滤
29 import warnings  # 用于忽略警告信息，避免日志过多
30
31 # 分布式训练相关
32 from torch.utils.data.distributed import DistributedSampler  # 在多GPU下分布式采样数据
33 import torch.distributed as dist  # 分布式训练的核心工具
34 import torch.multiprocessing as mp  # 多进程工具（分布式训练时常用）
35 from torch.nn.parallel import DistributedDataParallel as DDP  # 分布式数据并行包装器
36
37 # 设置全局参数
38 warnings.filterwarnings("ignore")  # 忽略所有警告信息
39 RUN_EXAMPLES = True  # 是否运行示例代码（调试时可设为 False）
40
41 # 一些在整个笔记本中都会用到的便捷辅助函数
42 def is_interactive_notebook():
43     # 判断当前运行环境是否为主程序 (__main__)
```

```

44     # 在 Jupyter Notebook 或交互式环境中, 这个判断一般为 False
45     return __name__ == "__main__"
46
47 def show_example(fn, args=[]):
48     # 如果在主程序中运行, 并且允许运行示例 (RUN_EXAMPLES=True)
49     # 那么执行并返回函数的结果, 用于演示例子
50     if __name__ == "__main__" and RUN_EXAMPLES:
51         return fn(*args)
52
53 def execute_example(fn, args=[]):
54     # 类似于 show_example, 但不会返回结果, 只是单纯执行
55     # 用于一些不需要结果展示的代码 (比如训练过程)
56     if __name__ == "__main__" and RUN_EXAMPLES:
57         fn(*args)
58
59 class DummyOptimizer(torch.optim.Optimizer):
60     # 一个“假的”优化器类, 用于在验证或推理阶段替代真实优化器
61     # 这样做的好处是复用训练代码结构, 而不用真的去更新参数
62
63     def __init__(self):
64         # 模拟一个 param_groups (真实优化器也有这个属性)
65         # 这里 lr=0 表示没有学习率
66         self.param_groups = [{"lr": 0}]
67         None
68
69     def step(self):
70         # 空方法, 不做任何参数更新
71         None
72
73     def zero_grad(self, set_to_none=False):
74         # 空方法, 不做梯度清零
75         None
76
77
78 class DummyScheduler:
79     # 一个“假的”学习率调度器, 用于验证/推理时占位
80     def step(self):
81         # 空方法, 不调整学习率
82         None

```

## 批归一化和层归一化对比



归一化方式	计算方式	主要用途	归一化范围
Batch Normalization (BN)	在 mini-batch 维度计算均值和标准差	CNN、深度网络等	在 batch 维度上
Layer Normalization (LayerNorm)	在特征维度计算均值和标准差	RNN、Transformer等	在单个样本的特征维度上

### 🔔 批归一化

优点：1. 有效抑制（协变量偏移）covariate shift，加速训练，提升深层网络性能。

缺点：1. 对 batch size 敏感，太小效果差。2. 在序列、变长输入等场景不适用。3. 推理时需提前统计均值方差，全局状态依赖强。

### 🔔 层归一化

优点：1. 对 batch size 不敏感，灵活，适合RNN、Transformer等需单步归一化场景。

缺点：1. 在大批量、多维空间特征场景下不如BatchNorm效果好。

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_1, \dots, x_m\}$ ;

Parameters to be learned:  $\gamma, \beta$

**Output:**  $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

**Algorithm 1:** Batch Normalizing Transform, applied to activation  $x$  over a mini-batch.

$$LN(\mathbf{z}; \alpha, \beta) = \frac{(\mathbf{z} - \mu)}{\sigma} \odot \alpha + \beta,$$

$$\mu = \frac{1}{D} \sum_{i=1}^D z_i, \quad \sigma = \sqrt{\frac{1}{D} \sum_{i=1}^D (z_i - \mu)^2},$$

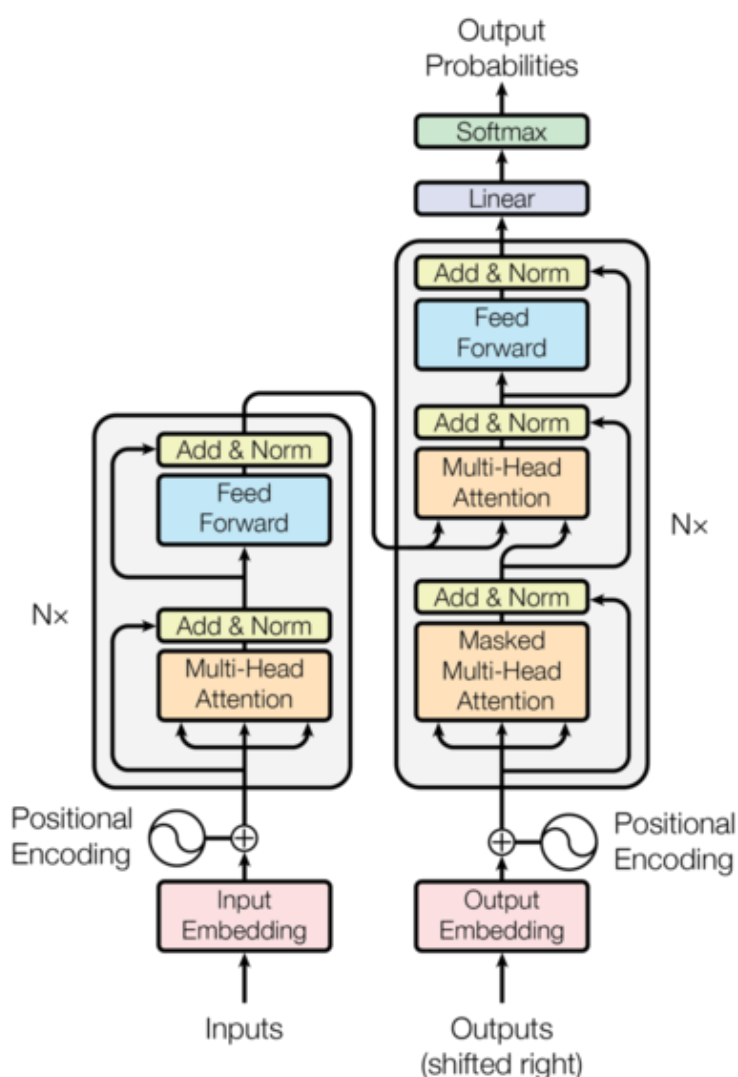
上面的式子即是层归一化的公式，归一化的步骤是对每个样本的所有特征进行归一化，其中  $D$  是特征维度， $\alpha$  和  $\beta$  是可学习的参数， $z_i$  是向量  $z$  的分量， $\mu$  是样本自身特征的均值， $\sigma$  是样本自身特征的标准差。最后如果想更加详细的了解层归一化的性能，可以直接参考 原论文的实验部分。

# 模型结构

## 编码器-解码器结构

大多数具有竞争力的神经序列转换模型都采用 **编码器-解码器 (encoder-decoder) 结构**。在这里，编码器将输入的符号序列表示  $(x_1, \dots, x_n)$  映射为连续表示序列  $\mathbf{z} = (z_1, \dots, z_n)$ 。在得到  $\mathbf{z}$  之后，**解码器**逐步生成输出符号序列  $(y_1, \dots, y_m)$ ，每次只生成一个元素。在生成序列的每一步中，模型采用 **自回归 (auto-regressive)** 的方式：在预测下一个符号时，会将已经生成的前序符号作为额外输入加以利用。

Transformer 遵循这种整体架构：编码器和解码器都由 **堆叠的自注意力层 (self-attention)** 和 **逐点全连接层 (point-wise fully connected layers)** 构成，如下图左半部分和右半部分所示。



代码块

```
1 class EncoderDecoder(nn.Module):
```

```

2      """
3      标准的编码器-解码器（Encoder-Decoder）架构。
4      这是许多 Transformer 类模型的基础。
5      """
6
7      def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
8          super(EncoderDecoder, self).__init__()
9          self.encoder = encoder      # 编码器模块
10         self.decoder = decoder      # 解码器模块
11         self.src_embed = src_embed  # 输入序列嵌入层 (source embedding)
12         self.tgt_embed = tgt_embed  # 输出序列嵌入层 (target embedding)
13         self.generator = generator  # 输出生成器 (linear + softmax)
14
15     def forward(self, src, tgt, src_mask, tgt_mask):
16         """
17         前向传播：
18         接收带 mask 的源序列和目标序列，并返回解码结果。
19         """
20         # 先编码源序列，再解码生成输出
21         return self.decode(self.encode(src, src_mask), src_mask, tgt, tgt_mask)
22
23     def encode(self, src, src_mask):
24         """
25         对源序列进行编码：
26         1. 先通过嵌入层得到向量表示
27         2. 再通过编码器处理，得到上下文表示
28         3. src_mask 的作用：对应源序列中 padding 的位置，
29            mask 会设置为很大的负值（通常是 -inf），使得 softmax 后对应注意力权重为 0。
30         """
31         return self.encoder(self.src_embed(src), src_mask)
32
33     def decode(self, memory, src_mask, tgt, tgt_mask):
34         """
35         对目标序列进行解码：
36         1. 先通过目标嵌入层得到向量表示
37         2. 使用编码器输出 memory 和 mask 信息进行解码
38         """
39         return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)
40
41
42     class Generator(nn.Module):
43         """
44         定义标准的输出生成步骤：
45         线性变换 + log_softmax，将解码器输出映射到词表概率分布
46         """
47
48         def __init__(self, d_model, vocab):

```

```

49         super(Generator, self).__init__()
50         self.proj = nn.Linear(d_model, vocab) # 将 d_model 维度映射到词表大小
51
52     def forward(self, x):
53         """
54         前向传播：
55         1. 先通过线性层
56         2. 再通过 log_softmax 转换为对数概率
57         """
58         return log_softmax(self.proj(x), dim=-1) # dim=-1 表示按最后一个维度归一
化

```

## 编码器



编码器由 **N=6** 个相同的层堆叠而成（如上图左半部分所示）。

代码块

```

1  import torch
2  import torch.nn as nn
3  import copy
4
5  def clones(module, N):
6      """
7      生成 N 个相同的层。
8      使用 deepcopy 确保每一层都是独立的参数。
9      """
10     return nn.ModuleList([copy.deepcopy(module) for _ in range(N)])
11
12
13 class Encoder(nn.Module):
14     """
15     Transformer 编码器 (Encoder) 核心：
16     由 N 个相同的层堆叠而成。
17     """
18
19     def __init__(self, layer, N):
20         super(Encoder, self).__init__()
21         self.layers = clones(layer, N) # 生成 N 个相同的编码器层
22         self.norm = LayerNorm(layer.size) # 最后的 LayerNorm, 用于层归一化输出
23
24     def forward(self, x, mask):
25         """
26         前向传播：
27         将输入 x (和 mask) 依次传入每一层编码器。

```

```

28         最后对输出进行 LayerNorm 归一化。
29         """
30         for layer in self.layers:
31             x = layer(x, mask) # 每一层处理后更新 x
32         return self.norm(x)    # 返回归一化后的最终输出

```



我们在每个子层外使用一个残差连接，然后再进行层归一化。


代码块

```

1  class LayerNorm(nn.Module):
2      """
3      构建一个 Layer Normalization 模块
4      """
5
6      def __init__(self, features, eps=1e-6):
7          """
8          :param features: 输入特征的维度 (D)，即每个样本的最后一个维度大小
9          :param eps: 防止除以 0 的微小常数，保证数值稳定性
10         """
11         super(LayerNorm, self).__init__()
12
13         # 可学习参数: 缩放因子 (gain), 初始化为 1
14         self.a_2 = nn.Parameter(torch.ones(features))
15
16         # 可学习参数: 平移因子 (bias), 初始化为 0
17         self.b_2 = nn.Parameter(torch.zeros(features))
18
19         self.eps = eps
20
21     def forward(self, x):
22         """
23         前向传播
24         输入 x: [batch_size, ..., features]
25         """
26         # 计算均值: 对最后一个维度 (features) 求均值
27         mean = x.mean(-1, keepdim=True)
28
29         # 计算标准差: 对最后一个维度 (features) 求标准差
30         std = x.std(-1, keepdim=True)
31
32         # 标准化 + 仿射变换
33         # (x - mean) / (std + eps) 保证最后一维特征均值为 0, 方差为 1
34         # 再通过 a_2 (缩放) 和 b_2 (平移) 恢复模型需要的分布
35         return self.a_2 * (x - mean) / (std + self.eps) + self.b_2


```



 就是说，每个子层的输出为： $LayerNorm(x + Sublayer(x))$ ，其中  $Sublayer(x)$  是子层自身实现的函数。我们在将子层输出与输入相加并进行归一化之前，对子层的输出应用 **dropout**。为了实现这些残差连接，模型中的所有子层，以及 embedding 层，输出的维度都是  $d_{model} = 512$ 。

代码块

```
1 class SublayerConnection(nn.Module):
2     """
3     子层连接模块 (Sublayer Connection) :
4     - 实现残差连接 (Residual Connection)
5     - 后接 LayerNorm (层归一化)
6     注意：为了代码简洁，这里先做归一化再输入子层 (Pre-LN) ,
7     而不是论文中的顺序 (Post-LN) 。
8     """
9
10    def __init__(self, size, dropout):
11        super(SublayerConnection, self).__init__()
12        self.norm = LayerNorm(size) # 初始化 LayerNorm, 规范化输入向量
13        self.dropout = nn.Dropout(dropout) # 初始化 Dropout, 防止过拟合
14
15    def forward(self, x, sublayer):
16        """
17        前向传播：
18        - x: 输入张量
19        - sublayer: 任意同维度的子层函数 (如 self-attention 或 feed-forward)
20        计算流程：
21        1. 对输入 x 做 LayerNorm: self.norm(x)
22        2. 将归一化后的 x 输入子层: sublayer(self.norm(x))
23        3. 对子层输出做 Dropout: self.dropout(...)
24        4. 与原始输入 x 相加，实现残差连接
25        返回：子层输出 + 残差
26        """
27        return x + self.dropout(sublayer(self.norm(x)))
```

 每一层包含 **两个子层**：

1. 第一个子层是 **多头自注意力机制 (multi-head self-attention)**
2. 第二个子层是 **简单的逐位置全连接前馈网络 (position-wise fully connected feed-forward network)**

```

代码块class EncoderLayer(nn.Module):
2     def __init__(self, size, self_attn, feed_forward, dropout):
3         """
4         初始化一个编码器层 (Encoder Layer)
5         参数:
6             size: 模型隐藏层的维度 (d_model)
7             self_attn: 多头自注意力机制模块
8             feed_forward: 前馈网络模块 (通常是两层全连接)
9             dropout: dropout比率, 用于正则化
10        """
11        super(EncoderLayer, self).__init__()
12        self.self_attn = self_attn          # 多头自注意力模块
13        self.feed_forward = feed_forward    # 前馈全连接模块
14        # 两个子层连接, 每个子层有残差连接和LayerNorm
15        # clones函数用于复制两个相同的SublayerConnection模块
16        self.sublayer = clones(SublayerConnection(size, dropout), 2)
17        self.size = size                    # 保存隐藏层维度信息
18
19    def forward(self, x, mask):
20        """
21        前向传播
22        参数:
23            x: 输入张量, 形状 [batch_size, seq_len, d_model]
24            mask: 注意力掩码, 用于屏蔽padding或未来信息
25        流程:
26            1. 第一子层: 多头自注意力 + 残差连接 + LayerNorm
27            2. 第二子层: 前馈网络 + 残差连接 + LayerNorm
28        """
29        # 第一个子层: 自注意力层
30        # lambda x: self.self_attn(x, x, x, mask) 表示把输入 x 作为查询、键、值
31        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
32        # 第二个子层: 前馈网络
33        x = self.sublayer[1](x, self.feed_forward)
34        return x

```

## 解码器



解码器 (decoder) 也由 N=6 个相同的层堆叠而成。

代码块

```

1 class Decoder(nn.Module):
2     # 通用的解码器, 由 N 层相同的 DecoderLayer 堆叠而成, 并支持掩码(masking)
3     def __init__(self, layer, N):
4         """

```

```

5         初始化解码器
6         参数：
7             layer: 一个 DecoderLayer 模块（包含自注意力、编码器-解码器注意力、前馈网
            络）
8             N: 解码器层数（通常 Transformer 原文是 6）
9         """
10        super(Decoder, self).__init__()
11        self.layers = clones(layer, N)      # 生成 N 个相同的解码器层
12        self.norm = LayerNorm(layer.size)   # 最后对输出做 LayerNorm 归一化
13
14    def forward(self, x, memory, src_mask, tgt_mask):
15        """
16        前向传播
17        参数：
18            x: 解码器输入张量，形状 [batch_size, tgt_seq_len, d_model]
19            memory: 编码器输出的记忆张量 (encoder output)
20            src_mask: 编码器输入掩码（屏蔽编码器padding）
21            tgt_mask: 解码器目标掩码（屏蔽未来信息，防止信息泄露）
22        流程：
23            对每一层解码器：
24                1. 自注意力层 (Masked Self-Attention)
25                2. 编码器-解码器注意力层 (Encoder-Decoder Attention)
26                3. 前馈网络层 (Feed Forward)
27                4. 每个子层都有残差连接和 LayerNorm
28            最后对所有层的输出做一次 LayerNorm
29        """
30        for layer in self.layers:
31            x = layer(x, memory, src_mask, tgt_mask)
32        return self.norm(x)  # 返回解码器最终输出

```



除了每个编码器层中的两个子层之外，解码器还插入了第三个子层，该子层对编码器堆栈的输出执行多头注意力（multi-head attention）。与编码器类似，我们在每个子层周围使用残差连接（residual connections），然后进行层归一化（layer normalization）。

代码块

```

1  class DecoderLayer(nn.Module):
2      # 解码器层由三个子层组成：
3      # 1. Masked 自注意力 (self-attention)
4      # 2. 编码器-解码器注意力 (encoder-decoder attention)
5      # 3. 前馈网络 (feed-forward network)
6      def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
7          """
8          初始化解码器层
9          参数：

```

```

10         size: 隐藏层维度 d_model
11         self_attn: Masked 自注意力模块
12         src_attn: 编码器-解码器注意力模块
13         feed_forward: 前馈网络模块
14         dropout: dropout 比率
15     """
16     super(DecoderLayer, self).__init__()
17     self.size = size # 保存隐藏层维度
18     self.self_attn = self_attn # Masked 自注意力模块
19     self.src_attn = src_attn # 编码器-解码器注意力模块
20     self.feed_forward = feed_forward # 前馈网络模块
21     # 三个子层，每个子层都有残差连接 + LayerNorm
22     self.sublayer = clones(SublayerConnection(size, dropout), 3)
23
24     def forward(self, x, memory, src_mask, tgt_mask):
25         """
26         前向传播
27         参数:
28             x: 解码器输入张量, 形状 [batch_size, tgt_seq_len, d_model]
29             memory: 编码器输出张量
30             src_mask: 源序列掩码 (屏蔽编码器 padding)
31             tgt_mask: 目标序列掩码 (屏蔽未来信息)
32         流程:
33             1. Masked 自注意力子层 (只看当前位置及之前位置)
34             2. 编码器-解码器注意力子层 (从编码器输出中获取信息)
35             3. 前馈网络子层
36             每个子层都有残差连接 + LayerNorm (通过 SublayerConnection 实现)
37         """
38         m = memory # 编码器输出
39         # 第一个子层: Masked 自注意力
40         x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, tgt_mask))
41         # 第二个子层: 编码器-解码器注意力
42         x = self.sublayer[1](x, lambda x: self.src_attn(x, m, m, src_mask))
43         # 第三个子层: 前馈网络
44         x = self.sublayer[2](x, self.feed_forward)
45         return x

```



我们还修改变码器堆栈中的自注意力子层，以防止当前位置关注后续位置。这个掩码（masking）机制，加上输出嵌入向量整体向后偏移一个位置，确保位置  $i$  的预测只能依赖于小于  $i$  的已知输出。

代码块

```

1 def subsequent_mask(size):
2     """

```

```

3     生成一个“后续位置掩码”（subsequent mask），用于解码器的自注意力，
4     以防止当前位置看到后面的词。
5     参数：
6         size: 序列长度 (tgt_seq_len)
7     返回：
8         mask: 布尔型张量，形状 [1, size, size]，上三角（未来位置）为 False，其余为
    True
9     """
10    # 创建一个全 1 张量，形状 [1, size, size]
11    attn_shape = (1, size, size)
12    # 上三角矩阵（对角线以上为1），表示要屏蔽的未来位置
13    subsequent_mask = torch.triu(torch.ones(attn_shape),
    diagonal=1).type(torch.uint8)
14    # 返回布尔掩码: True 表示可以关注, False 表示被屏蔽
15    return subsequent_mask == 0

```

#### 代码块

```

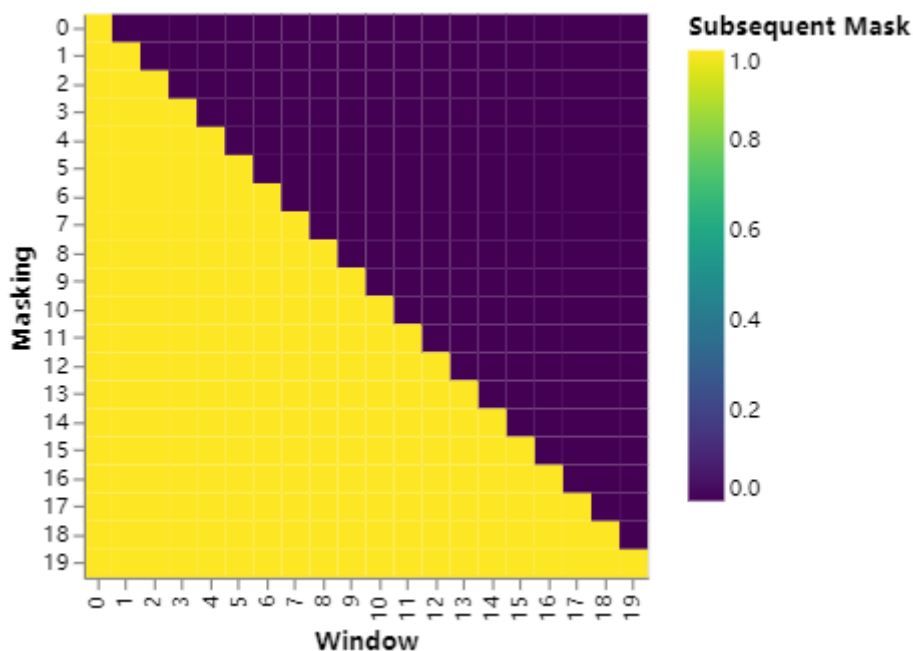
1  def example_mask():
2      """
3      可视化一个 20x20 的后续位置掩码（subsequent mask）。
4      用 Altair 绘制热力图，显示每个目标词在自注意力中允许关注的位置。
5      """
6
7      # 创建数据 DataFrame, 用于 Altair 绘图
8      LS_data = pd.concat(
9          [
10             pd.DataFrame(
11                 {
12                     # 后续掩码值, True 表示允许关注, False 表示屏蔽
13                     "Subsequent Mask": subsequent_mask(20)[0][x, y].flatten(),
14                     # 列坐标 (Window) : 表示目标词可以关注的列位置
15                     "Window": y,
16                     # 行坐标 (Masking) : 表示当前目标词所在的行
17                     "Masking": x,
18                 }
19             )
20             # 遍历 20x20 的矩阵, 生成每个 (行,列) 对应的数据
21             for y in range(20)
22             for x in range(20)
23         ]
24     )
25
26     # 使用 Altair 绘制热力图
27     return (
28         alt.Chart(LS_data)

```


```

29     .mark_rect() # 用矩形表示每个单元格
30     .properties(height=250, width=250) # 图像大小
31     .encode(
32         alt.X("Window:Q"), # 列坐标
33         alt.Y("Masking:Q"), # 行坐标
34         alt.Color("Subsequent Mask:Q", # 颜色表示掩码值
35                 scale=alt.Scale(scheme="viridis")), # 使用 viridis 颜色方
案
36     )
37     .interactive() # 支持交互缩放
38 )
39
40 # 调用 show_example 绘制图像
41 show_example(example_mask)

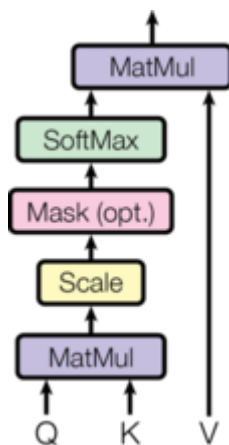
```



## 注意力机制

 注意力机制可以描述为将一个查询向量（query）和一组键-值对（key-value pairs）映射为一个输出向量，其中查询、键、值以及输出都是向量。输出向量是值向量的加权和，每个值的权重由查询向量与对应键向量的兼容性函数（compatibility function）计算得到。

我们称这种特定的注意力机制为‘缩放点积注意力（Scaled Dot-Product Attention）’。输入包括维度为  $d_k$  的查询（queries）和键（keys），以及维度为  $d_v$  的值（values）。我们计算查询向量与所有键向量的点积，然后除以  $\sqrt{d_k}$ ，再应用 softmax 函数，从而得到各个值向量的权重。



🔔 在实际应用中，我们会将一组查询向量同时计算注意力函数，并将它们打包成一个矩阵  $Q$ 。键向量和值向量也分别打包成矩阵  $K$  和  $V$ 。输出矩阵的计算公式为：

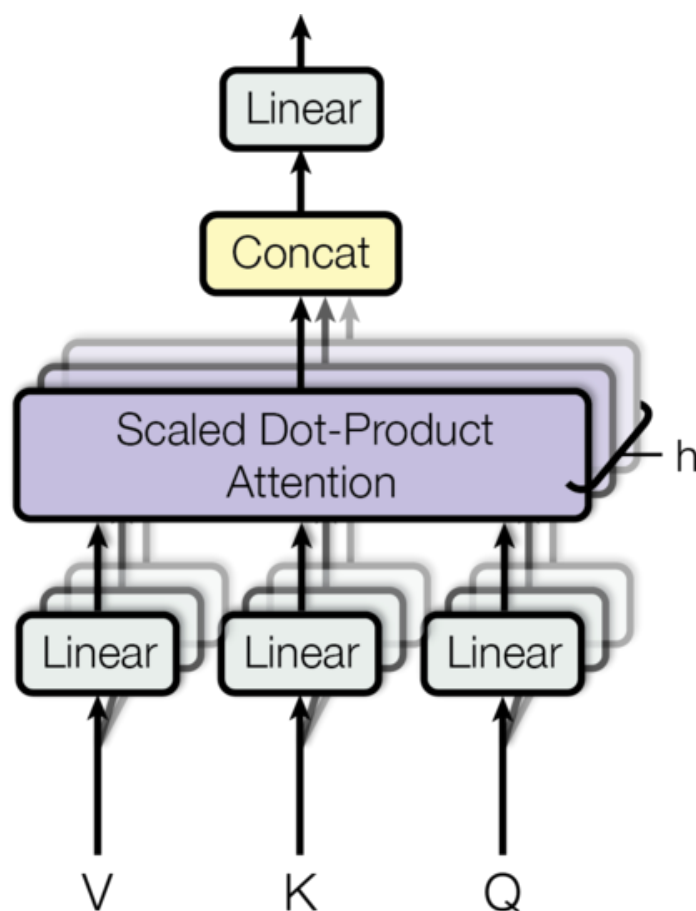
$$\text{Attention}(Q, K, V) = \text{softmax} \left( \frac{QK^{\top}}{\sqrt{d_k}} \right) V \quad (1)$$

代码块

```

1  def attention(query, key, value, mask=None, dropout=None):
2      # 获取 query 的最后一维大小，也就是每个向量的维度 d_k
3      d_k = query.size(-1)
4
5      # 计算注意力分数 (scores)
6      # query 与 key 的转置相乘，再除以 sqrt(d_k) 做缩放，防止维度过大导致 softmax 梯度
消失
7      scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
8
9      # 如果提供了 mask (通常用于 padding 或未来信息屏蔽)
10     # 将 mask 为 0 的位置对应的 scores 设置为一个很小的数 (-1e9)，这样 softmax 后几
乎为 0
11     if mask is not None:
12         scores = scores.masked_fill(mask == 0, -1e9)
13
14     # 对 scores 应用 softmax 得到注意力权重 p_attn
15     # dim=-1 表示对每行 (即每个 query 对所有 key) 进行 softmax
16     p_attn = scores.softmax(dim=-1)
17
18     # 如果提供了 dropout，应用在注意力权重上进行随机失活，用于正则化
19     if dropout is not None:
20         p_attn = dropout(p_attn)
21
22     # 最终的注意力输出是 p_attn 与 value 相乘
23     # 返回值：注意力输出和注意力权重 (可用于可视化或分析)
24     return torch.matmul(p_attn, value), p_attn
  
```

🔔 两种最常用的注意力函数是加性注意力（additive attention）和点积（乘法）注意力（dot-product attention）。点积注意力与我们的算法是相同的，唯一的区别是引入了一个缩放因子  $\frac{1}{\sqrt{d_k}}$ 。加性注意力（Additive Attention）通过一个带单隐藏层的前馈神经网络来计算兼容性函数。虽然理论复杂度上两者相似，但点积注意力（Dot-Product Attention）在实际应用中要快得多，且更节省空间，因为它可以利用高度优化的矩阵乘法来实现。当  $d_k$  较小时，这两种机制表现相似；但对于较大的  $d_k$  值，如果不进行缩放，加性注意力的性能优于点积注意力。我们怀疑，当  $d_k$  较大时，点积的数值会变得很大，这会使 softmax 函数进入梯度非常小的区域（为了说明为什么点积会变大，假设  $q$  和  $k$  的各个分量是均值为 0、方差为 1 的独立随机变量。那么它们的点积  $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$  的均值为 0，方差为  $d_k$ ）。为抵消这一影响，我们将点积除以  $\sqrt{d_k}$  进行缩放。



🔔 多头注意力允许模型在不同位置同时关注来自不同表示子空间的信息。而使用单个注意力头时，取平均会抑制这种能力。



$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W^O, \quad (2)$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \quad (3)$$

$$W_i^Q \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}, \quad W^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}.$$

在本工作中，我们使用  $h=8$  个并行的注意力层（或称注意力头）。对于每一个注意力头，我们设置  $d_k = d_v = d_{\text{model}}/h = 64$ 。由于每个头的维度减小，总的计算开销与使用全维度的单头注意力相当。

代码块

```

1  class MultiHeadedAttention(nn.Module):
2      def __init__(self, h, d_model, dropout=0.1):
3          "Take in model size and number of heads."
4          super(MultiHeadedAttention, self).__init__()
5          assert d_model % h == 0 # 确保模型维度可以被头数整除
6          # 我们假设 d_v 总是等于 d_k
7          self.d_k = d_model // h # 每个注意力头的维度
8          self.h = h # 注意力头的数量
9          self.linears = clones(nn.Linear(d_model, d_model), 4)
10         # 创建 4 个线性层，用于 Q、K、V 的投影以及最后的输出线性变换
11         self.attn = None # 用于存储注意力权重
12         self.dropout = nn.Dropout(p=dropout) # dropout 层，防止过拟合
13
14     def forward(self, query, key, value, mask=None):
15         if mask is not None:
16             # 相同的 mask 会应用到所有 h 个注意力头上
17             mask = mask.unsqueeze(1)
18             nbatches = query.size(0) # batch 大小
19
20             # 1) 对 Q、K、V 做线性投影，从 d_model => h x d_k
21             query, key, value = [
22                 lin(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
23                 for lin, x in zip(self.linears, (query, key, value))
24             ]
25             # lin(x): 对输入做线性变换
26             # view(): 改变形状为 [batch, seq_len, h, d_k]
27             # transpose(1, 2): 调整形状为 [batch, h, seq_len, d_k]，方便多头计算
28
29             # 2) 在所有投影后的向量上计算注意力
30             x, self.attn = attention(
31                 query, key, value, mask=mask, dropout=self.dropout
32             )
33             # attention 返回输出和注意力权重 self.attn
34

```

```

35         # 3) 将多头结果拼接起来，并应用最终线性变换
36         x = (
37             x.transpose(1, 2) # [batch, seq_len, h, d_k]
38             .contiguous()      # 保证内存连续
39             .view(nbatches, -1, self.h * self.d_k) # 拼接成 [batch, seq_len,
d_model]
40         )
41
42         del query
43         del key
44         del value # 释放内存
45
46         return self.linears[-1](x) # 最终输出线性变换

```



Transformer 在三种不同的方式使用多头注意力：

1. 在**编码器-解码器注意力**（encoder-decoder attention）层中，查询（queries）来自前一层解码器，而键（memory keys）和值（values）来自编码器的输出。这使得解码器中的每个位置都可以关注输入序列中的所有位置。这模仿了序列到序列（sequence-to-sequence）模型中常见的编码器-解码器注意力机制。
2. **编码器包含自注意力（self-attention）层**。在自注意力层中，所有的键（keys）、值（values）和查询（queries）都来自同一个来源，在这里是编码器前一层输出。编码器中的每个位置都可以关注编码器前一层的所有位置。
3. 类似地，**解码器中的自注意力（self-attention）层**允许解码器中的每个位置关注解码器中该位置及其之前的所有位置。为了保持自回归（auto-regressive）特性，我们需要阻止信息向左流动。在缩放点积注意力（scaled dot-product attention）中，我们通过屏蔽（mask）所有不合法的连接来实现这一点，即将对应 softmax 输入的值设为  $-\infty$ 。

## 逐位置前馈网络



除了注意力子层之外，我们的编码器和解码器中的每一层都包含一个全连接前馈网络（fully connected feed-forward network），该网络对每个位置**单独且相同地**应用。这个网络由两次线性变换组成，中间夹着一个 ReLU 激活函数。

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \quad (4)$$

虽然线性变换在不同位置上是相同的，但**每一层使用的参数是不同的**。另一种描述方式是，将其看作是两个卷积核大小为 1 的卷积操作。输入和输出的维度为  $d_{\text{model}} = 512$ ，而中间层的维度为  $d_{\text{ff}} = 2048$ 。这里采用 ReLU 的原因：

1. ReLU 对正数部分梯度恒为 1，不容易出现梯度消失（vanishing gradient），Sigmoid/tanh 在输入过大或过小时梯度会接近 0，训练深层网络可能不稳定。
2. 对大规模 Transformer 或深层网络训练来说，速度快。

代码块

```
1  class PositionwiseFeedForward(nn.Module):
2      """
3      实现 Transformer 中的逐位置前馈网络（Position-wise Feed-Forward Network,
4      FFN）。
5
6      对输入序列的每个位置（token）独立应用相同的前馈网络。
7      通常公式为：
8          
$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

9      其中 ReLU 激活函数应用于第一层线性变换的输出。
10     """
11     def __init__(self, d_model, d_ff, dropout=0.1):
12         """
13         初始化前馈网络。
14
15         参数：
16         d_model -- 输入和输出的特征维度（模型维度）
17         d_ff    -- 前馈网络隐藏层维度（通常大于 d_model）
18         dropout -- dropout 概率，用于防止过拟合
19         """
20     super(PositionwiseFeedForward, self).__init__()
21     self.w_1 = nn.Linear(d_model, d_ff)  # 第 1 个线性层，将 d_model 映射到
22     d_ff
23     self.w_2 = nn.Linear(d_ff, d_model)  # 第 2 个线性层，将 d_ff 映射回
24     d_model
25     self.dropout = nn.Dropout(dropout)   # Dropout 层
26
27     def forward(self, x):
28         """
29         前向传播。
30
31         输入：
32         x -- 形状为 (batch_size, seq_len, d_model) 的张量
33
34         步骤：
35         1. 对输入 x 进行线性变换 w_1
36         2. 经过 ReLU 激活函数
37         3. 应用 dropout
38         4. 再经过线性变换 w_2
39         5. 输出与输入形状相同
```

```

38         """
39         return self.w_2(self.dropout(self.w_1(x).relu()))

```

## 嵌入层

🔦 类似于其他序列转换（sequence transduction）模型，我们使用 **可学习的嵌入向量（learned embeddings）**。将输入 token 和输出 token 转换为维度为  $d_{model}$  的向量。我们还使用常规的可学习线性变换和 Softmax 函数将解码器（decoder）的输出转换为预测的下一个 token 的概率。在我们的模型中，我们将两个嵌入层与 Softmax 之前的线性变换共享同一个权重矩阵，这与文献中提到的方法类似。在嵌入层中，我们将这些权重乘以  $\sqrt{d_{model}}$ 。

代码块

```

1  class Embeddings(nn.Module):
2      """
3      Transformer 中的嵌入层（Embedding Layer）。
4
5      将输入的 token ID 转换为 d_model 维的向量表示，并进行缩放。
6      公式：
7          Embedding(x) = sqrt(d_model) * lookup(x)
8      其中 lookup(x) 为 nn.Embedding 查询对应 token 的向量。
9      """
10
11     def __init__(self, d_model, vocab):
12         """
13         初始化嵌入层。
14         参数：
15         d_model -- 输出向量的维度（模型维度）
16         vocab    -- 词表大小，即输入 token 的总数
17         """
18         super(Embeddings, self).__init__()
19         self.lut = nn.Embedding(vocab, d_model) # nn.Embedding: 查找 token 对应
            的向量
20         self.d_model = d_model # 保存模型维度，用于缩放
21
22     def forward(self, x):
23         """
24         前向传播。
25         输入：
26         x -- 形状为 (batch_size, seq_len) 的 token ID 张量
27         输出：
28         对应 token 的嵌入向量，形状为 (batch_size, seq_len, d_model)，
29         并乘以 sqrt(d_model) 进行缩放。

```

```

30         """
31         return self.lut(x) * math.sqrt(self.d_model)

```

## 位置编码

### 绝对位置编码

☀ 由于我们的模型 **不包含循环结构 (recurrence) 也不包含卷积 (convolution)**，为了让模型能够利用序列的顺序信息，我们必须向输入中注入关于 token **相对或绝对位置** 的信息。为此，我们在 **编码器和解码器堆栈的底部** 将“位置编码 (positional encodings)”加到输入嵌入 (input embeddings) 上。位置编码的维度与嵌入向量相同，即  $d_{model}$ ，这样二者可以直接相加。位置编码有多种选择，可以是 **可学习的 (learned)** 或 **固定的 (fixed)**。可学习的位置编码不能外推到更长的序列，但是正余弦编码可以。

在本工作中，我们使用 **不同频率的正弦 (sine) 和余弦 (cosine) 函数 (绝对位置编码-Absolute Position Encoding)**。

$$\begin{aligned}
 PE_{(pos, 2i)} &= \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right) \\
 PE_{(pos, 2i+1)} &= \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)
 \end{aligned} \tag{5}$$

其中  $pos$  表示位置， $i$  表示维度。也就是说，位置编码的每一个维度都对应一个正弦函数。这些正弦函数的波长形成了从  $2\pi$  到  $10000 * 2\pi$  的几何级数。我们选择这个函数的原因是，假设它可以让模型更容易地根据 **相对位置** 学习注意力，因为对于任意固定偏移量  $k$ ， $PE_{pos+k}$  可以表示为  $PE_{pos}$  的线性函数。

此外，我们在 **编码器和解码器堆栈中**，对 **嵌入向量与位置编码的和** 应用 *dropout*。对于基础模型，我们使用的 *dropout* 概率为  $P_{drop} = 0.1$ 。

代码块

```

1  class PositionalEncoding(nn.Module):
2      """
3      Transformer 中的位置编码 (Positional Encoding, PE) 实现。
4      将正弦/余弦位置编码添加到输入嵌入中，使模型能够感知序列中 token 的顺序。
5      """
6      def __init__(self, d_model, dropout, max_len=5000):
7          """
8          初始化位置编码。
9          参数：
10             d_model -- 嵌入向量的维度 (模型维度)
11             dropout -- dropout 概率
12             max_len -- 支持的最大序列长度 (默认 5000)

```

```

13         """
14         super(PositionalEncoding, self).__init__()
15         self.dropout = nn.Dropout(p=dropout) # 对加上位置编码后的向量应用 dropout
16
17         # 计算位置编码（一次性计算好，存储在 pe 中）
18         pe = torch.zeros(max_len, d_model) # 存储位置编码的张量，形状
(max_len, d_model)
19         position = torch.arange(0, max_len).unsqueeze(1) # 位置索引 0~max_len-
1, 形状 (max_len, 1)
20
21         # 计算每个维度的缩放因子，在log空间中计算
22         div_term = torch.exp(
23             torch.arange(0, d_model, 2) * -(math.log(10000.0) / d_model)
24         ) # shape=(d_model/2,)
25
26         # 偶数维使用 sin
27         pe[:, 0::2] = torch.sin(position * div_term)
28         # 奇数维使用 cos
29         pe[:, 1::2] = torch.cos(position * div_term)
30
31         pe = pe.unsqueeze(0) # 增加 batch 维度，形状变为 (1, max_len, d_model)
32
33         # 注册为 buffer，不参与梯度更新，但会随着模型保存/加载
34         self.register_buffer("pe", pe)
35
36     def forward(self, x):
37         """
38         前向传播。
39
40         输入：
41         x -- 输入嵌入向量，形状 (batch_size, seq_len, d_model)
42
43         步骤：
44         1. 将对应序列长度的 PE 加到输入嵌入上
45         2. 对结果应用 dropout
46         """
47         x = x + self.pe[:, : x.size(1)].requires_grad_(False) # 加上对应长度的位
置编码
48         return self.dropout(x)
49

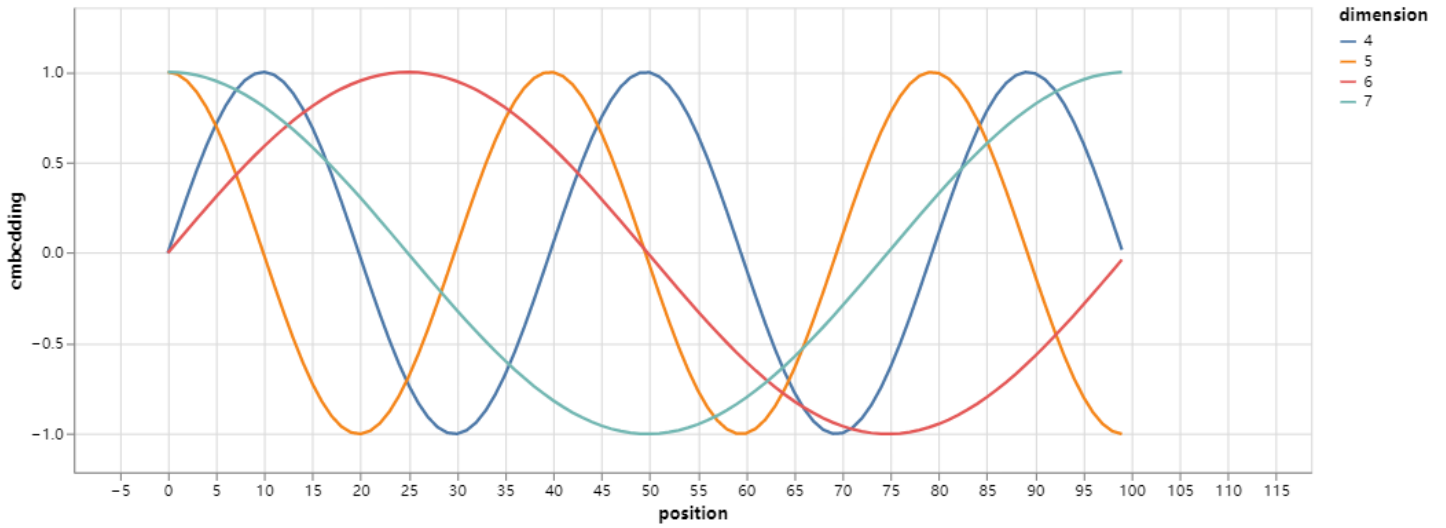
```

🔦 在位置编码中，会根据 **位置** 添加一个正弦波。每个维度的正弦波 **频率和相位偏移** 都不同。

```

1  def example_positional():
2      """
3      演示 PositionalEncoding 的效果，绘制不同维度的正弦/余弦位置编码曲线。
4      """
5
6      # 创建一个 PositionalEncoding 对象
7      # d_model=20 -> 嵌入维度为20
8      # dropout=0 -> 不使用 dropout
9      pe = PositionalEncoding(20, 0)
10
11     # 创建一个全零张量，形状 (1, 100, 20)
12     # 表示 batch_size=1, 序列长度=100, 嵌入维度=20
13     # 经过位置编码后，y 的每个位置会加上对应的正弦/余弦编码
14     y = pe.forward(torch.zeros(1, 100, 20))
15
16     # 构建绘图数据
17     # 只取第 4、5、6、7 维来绘图，方便观察正弦/余弦波形
18     data = pd.concat(
19         [
20             pd.DataFrame(
21                 {
22                     "embedding": y[0, :, dim],          # 对应维度的编码值
23                     "dimension": dim,                  # 维度编号
24                     "position": list(range(100)),       # 位置索引 0~99
25                 }
26             )
27             for dim in [4, 5, 6, 7]                    # 选择要绘制的维度
28         ]
29     )
30
31     # 使用 Altair 绘制折线图
32     # x 轴：位置，y 轴：编码值，不同维度用不同颜色
33     # interactive() 允许缩放和平移
34     return (
35         alt.Chart(data)
36         .mark_line()
37         .properties(width=800)
38         .encode(x="position", y="embedding", color="dimension:N")
39         .interactive()
40     )
41
42     # 调用 show_example 绘制示意图
43     show_example(example_positional)
44

```



## 相对位置编码



相对位置编码起源于论文《Self-Attention with Relative Position Representations》，与传统的 **绝对位置编码 (absolute positional encoding)** 不同，它强调序列中 **两个 token 之间的相对距离**，而不是 token 的全局位置。首先，假设输入序列

$x = (x_1, x_2, \dots, x_n)$ ,  $x_i \in \mathbb{R}^{d_x}$ , 逐步生成的输出序列是

$z = (z_1, z_2, \dots, z_n)$ ,  $z_i \in \mathbb{R}^{d_z}$ , 先考虑一般自注意力：

$$\begin{cases} q_i = x_i W^Q \\ k_j = x_j W^K \\ v_j = x_j W^V \\ a_{ij} = \text{softmax} \left( \frac{q_i k_j^\top}{\sqrt{d_z}} \right) \\ z_i = \sum_{j=1}^n a_{ij} v_j = \sum_{j=1}^n a_{ij} (x_j W^V) \end{cases} \quad (6)$$

其中权重向量

$$a_{ij} = \text{softmax}(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{l=1}^n \exp(e_{il})} = \frac{\exp \left( \frac{q_i k_j^\top}{\sqrt{d_z}} \right)}{\sum_{l=1}^n \exp \left( \frac{q_i k_l^\top}{\sqrt{d_z}} \right)} \quad (7)$$

此外， $W^Q, W^K, W^V \in \mathbb{R}^{d_x \times d_z}$ 。然后在论文中引入了两种边的表示：(1)  $a_{ij}^V$  用于 value 方向（即信息传递的内容）。(2)  $a_{ij}^K$  用于 key 方向（即注意力权重的计算）。随后将上面的式子修改成下面的形式。

$$z_i = \sum_{j=1}^n \alpha_{ij} (x_j W^V + a_{ij}^V) \quad (8)$$



$$e_{ij} = \frac{q_i k_j^\top}{\sqrt{d_z}} = \frac{x_i W^Q (x_j W^K + a_{ij}^K)^T}{\sqrt{d_z}} \quad (9)$$

对于线性序列，边可以捕捉输入元素之间的相对位置差的信息。我们考虑的最大相对位置会被截断到绝对值不超过  $k$ 。文章中假设是超过一定距离的精确相对位置信息并没有太大用处。截断最大距离还能使模型对训练中未见过的序列长度具有更好的泛化能力。因此，文章中考虑  $2k + 1$  个唯一的边标签。

$$a_{ij}^K = w_{\text{clip}(j-i;k)}^K, \quad a_{ij}^V = w_{\text{clip}(j-i;k)}^V, \quad \text{clip}(x; k) = \max(-k, \min(k, x)) \quad (10)$$

然后学习相对位置向量序列：

$$w^K = (w_{-k}^K, \dots, w_k^K), \quad w^V = (w_{-k}^V, \dots, w_k^V), \quad \text{where } w_i^K, w_i^V \in \mathbb{R}^{d_a} \quad (11)$$

在CSDN上也有人用带绝对位置编码的注意力来做[例子讲解](#)，怎么一步一步变到相对位置编码。同时也可以去看看《[ROFORMER: ENHANCED TRANSFORMER WITH ROTARY POSITION EMBEDDING](#)》论文中提到相对位置编码的小节内容。

## 旋转位置编码



旋转式位置编码 (Rotary Position Embedding)，通过绝对位置编码的方式实现相对位置编码，综合了绝对位置编码和相对位置编码的优点。具体可见《[ROFORMER: ENHANCED TRANSFORMER WITH ROTARY POSITION EMBEDDING](#)》。主要就是对 attention 中的 q、k 向量注入了绝对位置信息，然后用更新的 q、k 向量做 attention 中的内积就会引入相对位置信息了。RoPE 广泛应用在当前的大模型生态中。二维情况下，对于向量 q 用复数表示的 RoPE 如下所示：

$$f_q(x_m, m) = (W_q x_m) e^{im\theta} \quad (12)$$

$$f_k(x_n, n) = (W_k x_n) e^{in\theta} \quad (13)$$

$$g(x_m, x_n, m - n) = \text{Re} \left[ (W_q x_m) (W_k x_n)^* e^{i(m-n)\theta} \right] \quad (14)$$

其中  $\text{Re}[\cdot]$  表示复数的实部，而  $(W_k x_n)^*$  表示  $(W_k x_n)$  的共轭复数。 $\theta \in \mathbb{R}$  是一个预设的非零常数。我们可以进一步将  $f_{\{q,k\}}$  写成矩阵乘法形式：

$$f_{\{q,k\}}(x_m, m) = \begin{bmatrix} \cos(m\theta) & -\sin(m\theta) \\ \sin(m\theta) & \cos(m\theta) \end{bmatrix} \begin{bmatrix} W_{\{q,k\}}^{11} & W_{\{q,k\}}^{12} \\ W_{\{q,k\}}^{21} & W_{\{q,k\}}^{22} \end{bmatrix} \begin{bmatrix} x_m^{(1)} \\ x_m^{(2)} \end{bmatrix} \quad (15)$$

为了将我们在二维情况下的结果推广到任意偶数维的  $x_i \in \mathbb{R}^d$ ，我们将  $d$  维空间划分为  $d/2$  个子空间，并利用内积的线性性质将它们组合起来，从而将  $f_{\{q,k\}}$  转化为：

$$f_{q,k}(x_m, m) = R_{\Theta, m}^d W_{q,k} x_m \quad (16)$$

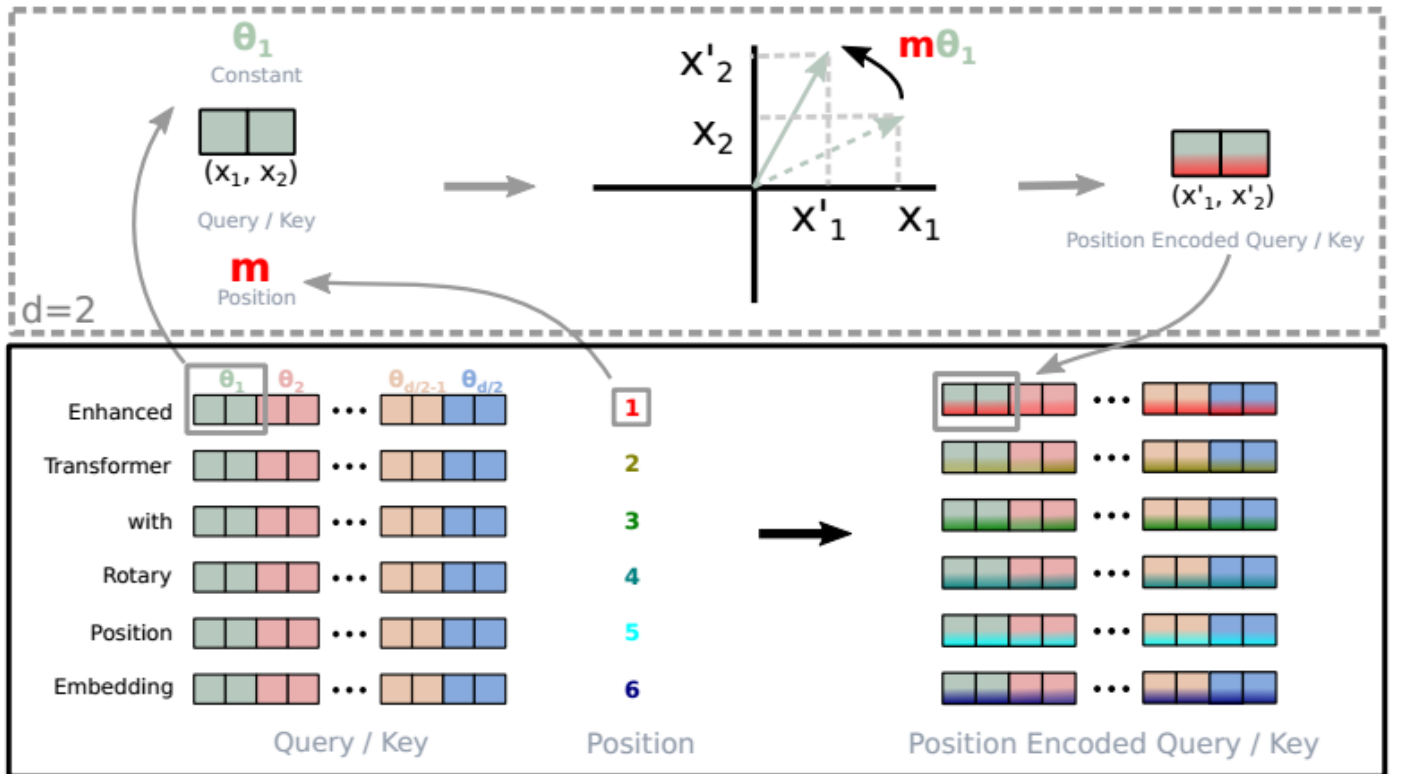
其中  $R_{\Theta, m}^d$  如下

$$R_{\Theta, m}^d = \begin{pmatrix} \cos(m\theta_1) & -\sin(m\theta_1) & 0 & 0 & \cdots & 0 & 0 \\ \sin(m\theta_1) & \cos(m\theta_1) & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos(m\theta_2) & -\sin(m\theta_2) & \cdots & 0 & 0 \\ 0 & 0 & \sin(m\theta_2) & \cos(m\theta_2) & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos(m\theta_{d/2}) & -\sin(m\theta_{d/2}) \\ 0 & 0 & 0 & 0 & \cdots & \sin(m\theta_{d/2}) & \cos(m\theta_{d/2}) \end{pmatrix} \quad (17)$$

它是一个旋转矩阵，其中参数集合为

$$\Theta = \left\{ \theta_i = 10000^{-2(i-1)/d}, i \in [1, 2, \dots, d/2] \right\} \quad (18)$$

具体详细证明可以到原论文去看或者可以到CSDN看一个博主的[详细证明](#)。



## 完整模型

在这里，我们定义一个从超参数到完整模型的函数。

代码块

```

1  def make_model(
2      src_vocab, tgt_vocab, N=6, d_model=512, d_ff=2048, h=8, dropout=0.1):
3      """
4      构建完整的 Transformer 模型 (Encoder-Decoder 结构)。
5
6      参数:
7      src_vocab -- 源语言词表大小
8      tgt_vocab -- 目标语言词表大小
9      N -- 编码器和解码器的层数 (默认 6)
10     d_model -- 词嵌入维度 (模型维度, 默认 512)
11     d_ff -- 前馈网络隐藏层维度 (默认 2048)
12     h -- 多头注意力头数 (默认 8)
13     dropout -- dropout 概率 (默认 0.1)
14     """
15     c = copy.deepcopy      # 深拷贝函数, 用于复制模块实例, 避免权重共享
16
17     # 定义基础组件
18     attn = MultiHeadedAttention(h, d_model)      # 多头自注意力层
19     ff = PositionwiseFeedForward(d_model, d_ff, dropout) # 前馈全连接层
20     position = PositionalEncoding(d_model, dropout) # 位置编码层
21
22     # 搭建 Encoder-Decoder 框架
23     model = EncoderDecoder(
24         # 编码器部分: N 层 EncoderLayer 堆叠
25         Encoder(EncoderLayer(d_model, c(attn), c(ff), dropout), N),
26         # 解码器部分: N 层 DecoderLayer 堆叠 (注意力层要深拷贝两份)
27         Decoder(DecoderLayer(d_model, c(attn), c(attn), c(ff), dropout), N),
28         # 源词嵌入 + 位置编码
29         nn.Sequential(Embeddings(d_model, src_vocab), c(position)),
30         # 目标词嵌入 + 位置编码
31         nn.Sequential(Embeddings(d_model, tgt_vocab), c(position)),
32         # 输出层 (把 d_model 维映射到目标词表大小)
33         Generator(d_model, tgt_vocab),
34     )
35
36     # 参数初始化: 使用 Xavier 均匀分布 (Glorot 初始化)
37     # 这样有助于稳定训练, 避免梯度消失或爆炸
38     for p in model.parameters():
39         if p.dim() > 1: # 只初始化权重矩阵 (不处理偏置等一维参数)
40             nn.init.xavier_uniform_(p)
41
42     return model

```

## 推理



这里进行一次前向传播，用模型生成预测结果。我们尝试使用 Transformer 来记忆输入。正如你将看到的，由于模型尚未经过训练，输出是随机生成的。在下一个教程中，我们将构建训练函数，并尝试训练模型去记忆从 1 到 10 的数字。

代码块

```

1  def inference_test():
2      # 构造一个测试用的 Transformer 模型
3      # src_vocab=11, tgt_vocab=11, N=2 (层数=2, 简化模型)
4      test_model = make_model(11, 11, 2)
5      test_model.eval() # 切换到 eval 模式 (关闭 dropout 等训练特性)
6
7      # 构造输入序列: 1 到 10
8      src = torch.LongTensor([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]) # shape=(1,10)
9      src_mask = torch.ones(1, 1, 10) # 全 1 的 mask (没有屏蔽任何位置)
10
11     # 编码器处理输入
12     memory = test_model.encode(src, src_mask)
13
14     # 初始化解码器输入 ys, 起始符号 (0 表示起始 token)
15     ys = torch.zeros(1, 1).type_as(src)
16
17     # 逐步生成序列 (这里生成 9 个 token)
18     for i in range(9):
19         # 解码器输出: 输入为 memory (编码器输出)、mask、已生成序列 ys
20         out = test_model.decode(
21             memory,
22             src_mask,
23             ys,
24             subsequent_mask(ys.size(1)).type_as(src.data) # 防止看到未来 token
25         )
26
27         # 取解码器最后一个位置的输出, 通过生成器得到预测概率分布
28         prob = test_model.generator(out[:, -1])
29
30         # 从概率分布中选择最大概率的 token
31         _, next_word = torch.max(prob, dim=1)
32         next_word = next_word.data[0]
33
34         # 把预测到的新 token 拼接到 ys 序列里
35         ys = torch.cat(
36             [ys, torch.empty(1, 1).type_as(src.data).fill_(next_word)], dim=1
37         )
38

```

```

39     # 打印模型预测的结果（由于模型未训练，输出是随机的）
40     print("Example Untrained Model Prediction:", ys)
41
42
43     def run_tests():
44         # 运行 10 次推理测试
45         for _ in range(10):
46             inference_test()
47
48     # 展示推理示例
49     show_example(run_tests)
50
51     #输出结果
52     #Example Untrained Model Prediction: tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
53     #Example Untrained Model Prediction: tensor([[0, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4]])
54     #Example Untrained Model Prediction: tensor([[ 0, 10, 10, 10,  3,  2,  5,  7,
55     9,  6]])
56     #Example Untrained Model Prediction: tensor([[ 0,  4,  3,  6, 10, 10,  2,  6,
57     2,  2]])
58     #Example Untrained Model Prediction: tensor([[ 0,  9,  0,  1,  5, 10,  1,  5,
59     10,  6]])
60     #Example Untrained Model Prediction: tensor([[ 0,  1,  5,  1, 10,  1, 10, 10,
61     10, 10]])
62     #Example Untrained Model Prediction: tensor([[ 0,  1, 10,  9,  9,  9,  9,  9,
63     1,  5]])
64     #Example Untrained Model Prediction: tensor([[ 0,  3,  1,  5, 10, 10, 10, 10,
65     10, 10]])
66     #Example Untrained Model Prediction: tensor([[ 0,  3,  5, 10,  5, 10,  4,  2,
67     4,  2]])
68     #Example Untrained Model Prediction: tensor([[0, 5, 6, 2, 5, 6, 2, 6, 2, 2]])

```

## 模型训练



这一节描述了模型的训练机制。在此之前，先插入一个简短的过渡，介绍一些用于训练标准编码器-解码器模型的工具。首先，定义一个 **Batch** 对象，用来保存训练时的源句子和目标句子，并负责构建相应的 **mask**。

## 批处理与掩码

代码块

```

1     class Batch:
2         """用于在训练中保存一个批次的数据及对应掩码的对象"""

```

```

3
4     def __init__(self, src, tgt=None, pad=2): # pad=2 表示填充符 <blank>
5         # 保存源序列
6         self.src = src
7         # 为源序列生成掩码：非填充位置为 True，填充位置为 False
8         # unsqueeze(-2) 增加一个维度，以便在注意力计算中广播
9         # 原本形状 (batch_size, seq_len) → (batch_size, 1, seq_len)
10        self.src_mask = (src != pad).unsqueeze(-2)
11
12        if tgt is not None:
13            # 解码器的输入序列：去掉目标序列的最后一个 token
14            self.tgt = tgt[:, :-1]
15            # 解码器的目标输出序列：去掉目标序列的第一个 token
16            # 用于计算 loss
17            self.tgt_y = tgt[:, 1:]
18            # 为目标序列生成掩码，屏蔽填充位置和未来位置
19            self.tgt_mask = self.make_std_mask(self.tgt, pad)
20            # 统计目标序列中非填充 token 的数量，用于 loss 归一化
21            self.ntokens = (self.tgt_y != pad).data.sum()
22
23        @staticmethod
24        def make_std_mask(tgt, pad):
25            """为目标序列创建标准掩码（屏蔽填充位置和未来 token）"""
26            # 首先屏蔽填充位置
27            tgt_mask = (tgt != pad).unsqueeze(-2)
28            # 再应用下三角掩码，防止解码器看到未来 token
29            tgt_mask = tgt_mask & subsequent_mask(tgt.size(-1)).type_as(
30                tgt_mask.data
31            )
32            return tgt_mask
33

```



接下来，我们创建一个通用的训练和评估函数，用于跟踪损失（loss）。我们会传入一个通用的 **loss 计算函数**，它同时负责参数的更新。

## 训练循环

代码块

```

1     class TrainState:
2         """用于跟踪训练状态，包括步骤数、样本数和处理的 token 数"""
3         step: int = 0 # 当前 epoch 中的梯度步数（每个 batch 都增加）
4         accum_step: int = 0 # 梯度累积步数（用于梯度累积优化）
5         samples: int = 0 # 处理的样本总数

```

```

6     tokens: int = 0      # 处理的 token 总数
7
8     def run_epoch(
9         data_iter,        # 数据迭代器, 按 batch 提供训练/验证数据
10        model,            # Transformer 模型
11        loss_compute,     # 计算损失的函数, 同时可以更新参数
12        optimizer,        # 优化器
13        scheduler,        # 学习率调度器
14        mode="train",     # 模式, 可选 "train" / "train+log" / "eval"
15        accum_iter=1,     # 梯度累积步数
16        train_state=TrainState(), # 保存训练状态
17    ):
18        """训练或验证单个 epoch"""
19        start = time.time() # 记录时间
20        total_tokens = 0    # 本 epoch 总 token 数 (用于平均 loss)
21        total_loss = 0     # 本 epoch 总 loss
22        tokens = 0         # 用于计算每秒处理 token 数
23        n_accum = 0        # 已累积优化步数
24
25        for i, batch in enumerate(data_iter):
26            # 前向传播
27            out = model.forward(
28                batch.src, batch.tgt, batch.src_mask, batch.tgt_mask
29            )
30
31            # 计算 loss, 并返回用于反向传播的节点 loss_node
32            loss, loss_node = loss_compute(out, batch.tgt_y, batch.ntokens)
33            # 如果使用梯度累积, 可以除以 accum_iter
34            # loss_node = loss_node / accum_iter
35
36            if mode == "train" or mode == "train+log":
37                # 反向传播
38                loss_node.backward()
39                train_state.step += 1 # 增加全局步数
40                train_state.samples += batch.src.shape[0] # 增加处理样本数
41                train_state.tokens += batch.ntokens # 增加处理 token 数
42
43            # 梯度累积到指定步数后更新参数
44            if i % accum_iter == 0:
45                optimizer.step() # 更新参数
46                optimizer.zero_grad(set_to_none=True) # 清空梯度
47                n_accum += 1
48                train_state.accum_step += 1
49
50            # 更新学习率
51            scheduler.step()
52

```

```

53     # 累加总 loss 和 token 数, 用于 epoch 统计
54     total_loss += loss
55     total_tokens += batch.ntokens
56     tokens += batch.ntokens
57
58     # 每 40 个 batch 打印一次训练信息
59     if i % 40 == 1 and (mode == "train" or mode == "train+log"):
60         lr = optimizer.param_groups[0]["lr"] # 当前学习率
61         elapsed = time.time() - start # 距离上一次打印时间
62         print(
63             (
64                 "Epoch Step: %6d | Accumulation Step: %3d | Loss: %6.2f "
65                 + "| Tokens / Sec: %7.1f | Learning Rate: %6.1e"
66             )
67             % (i, n_accum, loss / batch.ntokens, tokens / elapsed, lr)
68         )
69         start = time.time() # 重置计时器
70         tokens = 0 # 重置 token 计数
71
72     # 释放内存
73     del loss
74     del loss_node
75
76     # 返回平均 loss 和训练状态
77     return total_loss / total_tokens, train_state

```

## 优化器

 我们使用 **Adam 优化器** 进行训练, 参数设置为:

$$\beta_1 = 0.9, \quad \beta_2 = 0.98, \quad \epsilon = 10^{-9} \quad (19)$$

在训练过程中, 我们使用以下公式调整学习率:

$$\text{lr} = d_{\text{model}}^{-1/2} \cdot \min \left( \text{step\_num}^{-1/2}, \text{step\_num} \cdot \text{warmup\_steps}^{-3/2} \right) \quad (20)$$

这对应于在训练的前 `warmup_steps` 步中, 线性地增加学习率; 之后则按照步数的平方根的倒数来降低学习率。我们使用了 `warmup_steps=4000`。

注意: 这一部分非常重要。训练模型时需要使用这种设置。下图展示了不同模型规模以及不同优化超参数下, 该模型学习率曲线的示例。



```

1  def rate(step, model_size, factor, warmup):
2      """
3      we have to default the step to 1 for LambdaLR function
4      to avoid zero raising to negative power.
5      """
6      if step == 0:
7          step = 1
8      return factor * (
9          model_size ** (-0.5) * min(step ** (-0.5), step * warmup ** (-1.5))
10     )

```

代码块

```

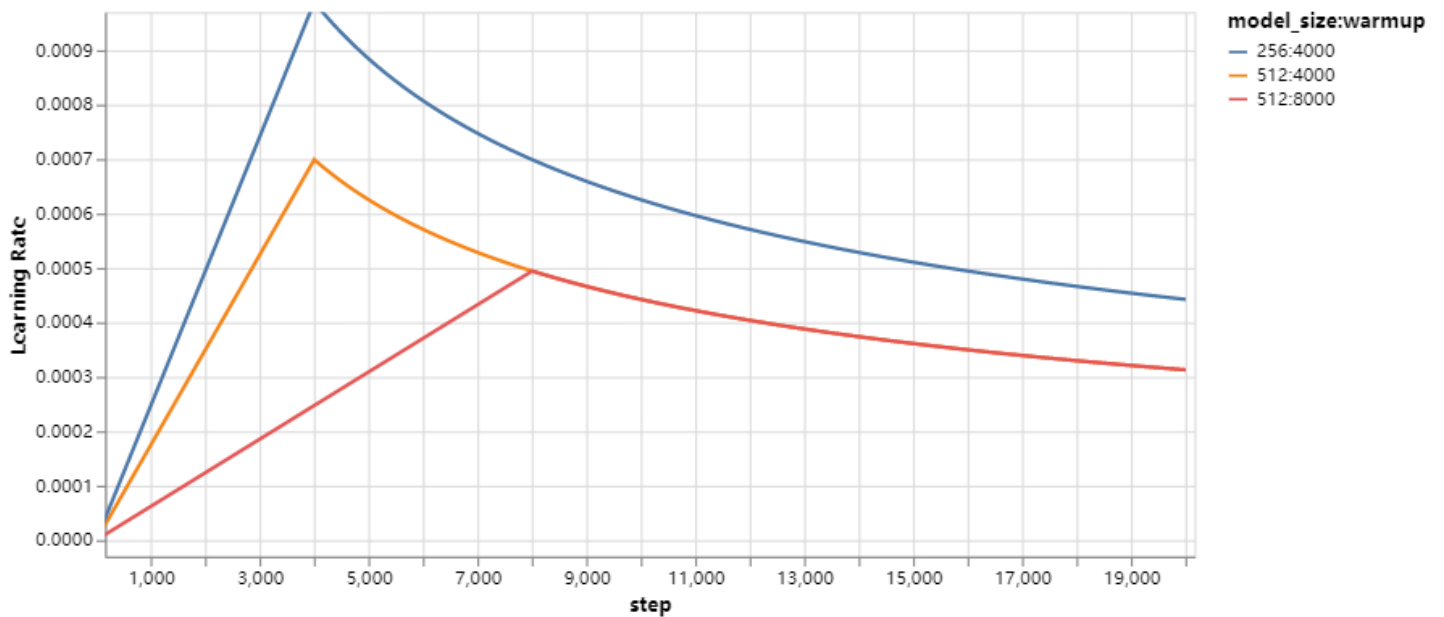
1  def example_learning_schedule():
2      # 定义三个不同的学习率设置，每个子列表包含 [模型维度, 缩放因子, warmup_steps]
3      opts = [
4          [512, 1, 4000], # 示例 1
5          [512, 1, 8000], # 示例 2
6          [256, 1, 4000], # 示例 3
7      ]
8
9      # 创建一个简单的线性模型作为占位模型 (dummy model)
10     dummy_model = torch.nn.Linear(1, 1)
11
12     # 用于存储不同设置下每个训练步的学习率曲线
13     learning_rates = []
14
15     # 遍历 opts 列表中的三个不同学习率设置
16     for idx, example in enumerate(opts):
17         # 使用 Adam 优化器初始化模型参数，初始 lr=1, beta1=0.9, beta2=0.98,
18         # epsilon=1e-9
19         optimizer = torch.optim.Adam(
20             dummy_model.parameters(), lr=1, betas=(0.9, 0.98), eps=1e-9
21         )
22
23         # 使用 LambdaLR 定义自定义学习率调度策略
24         # rate 函数根据 step 和当前 example 的参数计算学习率
25         lr_scheduler = LambdaLR(
26             optimizer=optimizer, lr_lambda=lambda step: rate(step, *example)
27         )
28
29         # 临时存储当前 example 的学习率
30         tmp = []
31
32         # 模拟 20000 个训练步，记录每一步的学习率
33         for step in range(20000):

```

```

33         tmp.append(optimizer.param_groups[0]["lr"]) # 保存当前学习率
34         optimizer.step() # 假装做一次参数更新 (这里没有实际 loss)
35         lr_scheduler.step() # 更新学习率
36
37         # 将当前 example 的学习率曲线保存到列表
38         learning_rates.append(tmp)
39
40         # 将学习率列表转换为 torch.Tensor, 方便后续处理
41         learning_rates = torch.tensor(learning_rates)
42
43         # Altair 默认限制行数为 5000, 这里禁用限制以支持 20000 步数据
44         alt.data_transformers.disable_max_rows()
45
46         # 构建用于绘图的 DataFrame, 每行包含 step、Learning Rate 和 model_size:warmup
47         opts_data = pd.concat(
48             [
49                 pd.DataFrame(
50                     {
51                         "Learning Rate": learning_rates[warmup_idx, :], # 学习率
52                         "model_size:warmup": ["512:4000", "512:8000", "256:4000"][
53                             warmup_idx
54                         ], # 标注当前 example
55                         "step": range(20000), # 步数
56                     }
57                 )
58                 for warmup_idx in [0, 1, 2] # 对三个 example 构建 DataFrame
59             ]
60         )
61
62         # 使用 Altair 绘制折线图, x 轴为 step, y 轴为 Learning Rate, 不同颜色表示不同设置
63         return (
64             alt.Chart(opts_data)
65             .mark_line()
66             .properties(width=600)
67             .encode(x="step", y="Learning Rate", color="model_size:warmup:N")
68             .interactive() # 启用交互功能, 可以缩放和平移
69         )
70
71         # 调用函数绘制学习率曲线
72         example_learning_schedule()

```



## 正则化

### 标签平滑

🔧 在训练过程中，我们使用了标签平滑（label smoothing），平滑系数为  $\epsilon_{ls} = 0.1$ 。这会对困惑度（perplexity）产生一定影响，因为模型会变得不那么自信，但能够提高准确率（accuracy）和 BLEU 分数。我们通过 KL 散度（KL divergence）损失实现标签平滑。与使用 one-hot 标签不同，我们构建了一个分布：对正确单词赋予较高的置信度（confidence），其余的平滑概率均匀分配到整个词表中。

#### 代码块

```
1 class LabelSmoothing(nn.Module):
2     "实现标签平滑 (Label Smoothing) "
3
4     def __init__(self, size, padding_idx, smoothing=0.0):
5         """
6         size: 词表大小 (类别总数)
7         padding_idx: 用于 padding 的索引 (忽略在 loss 计算中)
8         smoothing: 平滑系数  $\epsilon$  (例如 0.1)
9         """
10        super(LabelSmoothing, self).__init__()
11        # 使用 KL 散度计算损失, reduction="sum" 表示对 batch 内元素求和
12        self.criterion = nn.KLDivLoss(reduction="sum")
13        self.padding_idx = padding_idx
14        self.confidence = 1.0 - smoothing # 正确标签的置信度
15        self.smoothing = smoothing # 平滑值
16        self.size = size # 类别总数
17        self.true_dist = None # 保存平滑后的目标分布 (可用于调试)
18
```

```

19     def forward(self, x, target):
20         """
21         x: 模型输出 logits (未经过 softmax)
22             形状: [batch_size, vocab_size]
23         target: 真实标签索引
24             形状: [batch_size]
25         """
26         assert x.size(1) == self.size # 确保输出维度与词表大小一致
27
28         # 1. 初始化目标分布为平滑后的概率
29         true_dist = x.data.clone() # 复制 x 的形状
30         true_dist.fill_(self.smoothing / (self.size - 2)) # 给每个类别分配平滑概
率 (排除 padding 和正确类别)
31
32         # 2. 对正确标签赋予较高的置信度
33         true_dist.scatter_(1, target.data.unsqueeze(1), self.confidence)
34
35         # 3. padding 位置的概率设为 0
36         true_dist[:, self.padding_idx] = 0
37
38         # 4. 找到所有 padding 的位置
39         mask = torch.nonzero(target.data == self.padding_idx)
40         if mask.dim() > 0:
41             # 将 padding 对应的行全部置为 0, 不参与 loss 计算
42             true_dist.index_fill_(0, mask.squeeze(), 0.0)
43
44         # 5. 保存平滑后的目标分布 (可调试或可视化)
45         self.true_dist = true_dist
46
47         # 6. 使用 KL 散度计算 loss
48         # x 需要先经过 log_softmax 后才能和 true_dist 对比, 这里假设外部传入的 x 已经
是 log_prob
49         return self.criterion(x, true_dist.clone().detach())

```



这里我们可以看到一个示例，展示了根据置信度（confidence）如何将概率质量分配到各个单词上。

代码块

```

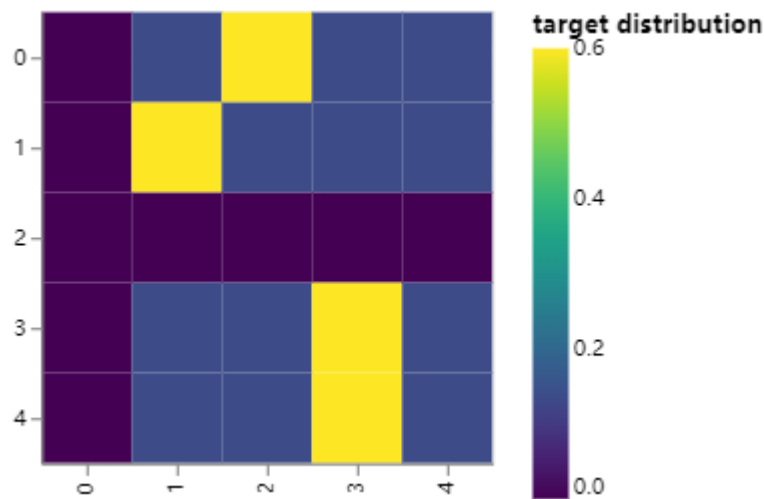
1 # Example of label smoothing.
2
3 def example_label_smoothing():
4     crit = LabelSmoothing(5, 0, 0.4)
5     predict = torch.FloatTensor(
6         [

```

```

7         [0, 0.2, 0.7, 0.1, 0],
8         [0, 0.2, 0.7, 0.1, 0],
9         [0, 0.2, 0.7, 0.1, 0],
10        [0, 0.2, 0.7, 0.1, 0],
11        [0, 0.2, 0.7, 0.1, 0],
12    ]
13 )
14 crit(x=predict.log(), target=torch.LongTensor([2, 1, 0, 3, 3]))
15 LS_data = pd.concat(
16     [
17         pd.DataFrame(
18             {
19                 "target distribution": crit.true_dist[x, y].flatten(),
20                 "columns": y,
21                 "rows": x,
22             }
23         )
24         for y in range(5)
25         for x in range(5)
26     ]
27 )
28
29 return (
30     alt.Chart(LS_data)
31     .mark_rect(color="Blue", opacity=1)
32     .properties(height=200, width=200)
33     .encode(
34         alt.X("columns:O", title=None),
35         alt.Y("rows:O", title=None),
36         alt.Color(
37             "target distribution:Q", scale=alt.Scale(scheme="viridis")
38         ),
39     )
40     .interactive()
41 )
42
43 show_example(example_label_smoothing)

```



如果上面还不能直观的感受标签平滑具体怎么做，这里举个简单的例子来帮助理解：

在 Label Smoothing 中，我们首先初始化目标分布 `true_dist`，将每个非正确类别的概率赋为平滑值  $\epsilon / (V - 2)$ ，其中  $V$  是类别总数，`-2` 表示排除正确类别和 padding。例如，假设 `vocab_size=5`, `smoothing=0.1`, `padding_idx=0`, `batch_size=3`，则每个位置初始概率约为 0.0333，得到：

```
[
  [0.0333, 0.0333, 0.0333, 0.0333, 0.0333],
  [0.0333, 0.0333, 0.0333, 0.0333, 0.0333],
  [0.0333, 0.0333, 0.0333, 0.0333, 0.0333]
]
```

接着，用 `scatter_` 将每个样本的正确类别赋予较高的置信度 `confidence = 1 - smoothing`。假设 `target = [2, 0, 3]`，`confidence=0.9`，则结果为：

```
[
  [0.0333, 0.0333, 0.9, 0.0333, 0.0333],
  [0.9, 0.0333, 0.0333, 0.0333, 0.0333],
  [0.0333, 0.0333, 0.0333, 0.9, 0.0333]
]
```

为了忽略 padding 列的贡献，用 `true_dist[:, padding_idx] = 0` 将 padding 列置为 0，不参与 loss 计算，得到：

```
[
  [0, 0.0333, 0.9, 0.0333, 0.0333],
  [0, 0.0333, 0.0333, 0.0333, 0.0333],
  [0, 0.0333, 0.0333, 0.9, 0.0333]
]
```

然后，找出 batch 中 target 本身就是 padding 的样本，用 `index_fill_` 将整行置 0，确保这些样本不参与梯度计算。对于 `target = [2, 0, 3]`，第二个样本是 padding，因此整行置 0，得到最终 `true_dist`：

```
[
  [0,      0.0333, 0.9,      0.0333, 0.0333], # 样本 0
  [0,      0,      0,      0,      0],        # 样本 1 padding
  [0,      0.0333, 0.0333, 0.9,      0.0333]  # 样本 2
]
```

最后将平滑后的目标分布保存到 `self.true_dist`，供调试或可视化使用。整个过程完成后，得到的 `true_dist` 就是 **平滑后的目标分布**，可用于 KL 散度或交叉熵计算。



标签平滑实际上会在模型对某个选项过于自信时对其进行惩罚。

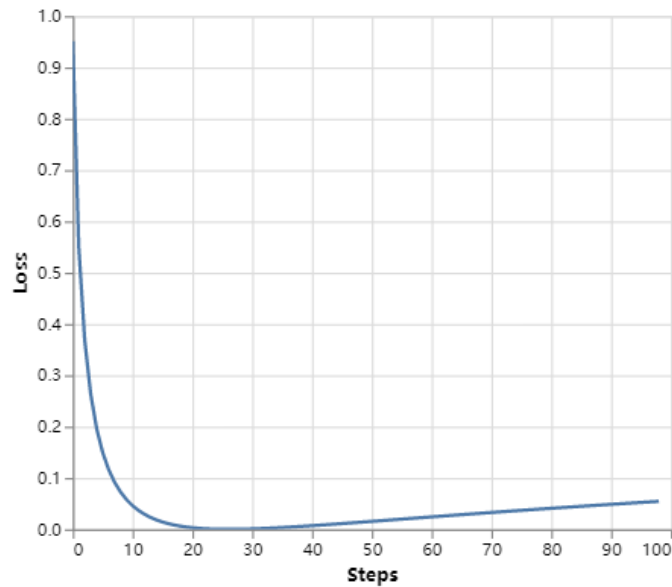
代码块

```
1  def loss(x, crit):
2      # 计算一个简单的归一化值 d，用于构造预测概率
3      d = x + 3 * 1
4      # 构造预测向量 predict
5      # 预测向量长度为 5 (vocab_size=5)
6      # 第 0 个位置为 0 (通常对应 padding)
7      # 第 1 个位置为 x/d (这是正确标签对应的值)
8      # 剩余位置均为 1/d (其他类别平滑分布)
9      predict = torch.FloatTensor([[0, x / d, 1 / d, 1 / d, 1 / d]])
10     # 对预测向量取 log，传入 LabelSmoothing 计算损失
11     # 目标标签是索引 1
12     # 返回标量 loss 值
13     return crit(predict.log(), torch.LongTensor([1])).data
14
15  def penalization_visualization():
16     # 初始化 Label Smoothing 对象
17     # vocab_size=5, padding_idx=0, smoothing=0.1
18     crit = LabelSmoothing(5, 0, 0.1)
19
20     # 构造 DataFrame，用于可视化 loss 随预测变化的曲线
21     loss_data = pd.DataFrame(
22         {
23             # 计算 loss 从 x=1 到 x=99
24             "Loss": [loss(x, crit) for x in range(1, 100)],
25             # 步数对应 x 的取值
26             "Steps": list(range(99)),
27         }
28     ).astype("float")
29     # 使用 Altair 绘制折线图
30     return (
31         alt.Chart(loss_data)
32         .mark_line() # 折线图
```

```

33     .properties(width=350) # 图表宽度
34     .encode(
35         x="Steps",          # x 轴：步骤或预测值变化
36         y="Loss",          # y 轴：对应的损失值
37     )
38     .interactive()         # 支持交互操作（缩放、平移）
39 )
40 # 调用绘图函数，显示 Label Smoothing 对 loss 的惩罚效果
41 show_example(penalization_visualization)

```



## 示例

### 简单示例



首先从一个简单的 **复制任务（copy-task）** 开始：给定一个来自小词表的随机输入符号序列，目标是生成与输入完全相同的符号序列。

### 生成数据

代码块

```

1  def data_gen(V, batch_size, nbatches):
2      """
3      为 src-tgt 复制任务生成随机数据。
4
5      参数：
6      - V: 词表大小（符号总数）
7      - batch_size: 每个 batch 的序列数量
8      - nbatches: 生成 batch 的总数

```



```

9
10     功能：
11     1. 生成形状为 [batch_size, 10] 的随机整数张量，每个元素在 [1, V) 范围内，表示词表
索引。
12     2. 将每个序列的第一个 token 固定为 1（起始标记）。
13     3. 构造 src 和 tgt，复制任务中两者相同。
14     4. 返回 Batch 对象，padding token index 为 0。
15     """
16     for i in range(nbatches):
17         data = torch.randint(1, V, size=(batch_size, 10)) # 随机生成 batch
18         data[:, 0] = 1 # 第一个 token 固定为
1
19         src = data.requires_grad_(False).clone().detach() # 源序列
20         tgt = data.requires_grad_(False).clone().detach() # 目标序列
21         yield Batch(src, tgt, 0) # 返回 Batch

```

## 损失计算

代码块

```

1  class SimpleLossCompute:
2      "一个简单的 loss 计算和训练辅助类。"
3
4      def __init__(self, generator, criterion):
5          """
6          初始化 SimpleLossCompute。
7
8          参数：
9          - generator: 模型的输出生成器（通常是一个线性层 + softmax/log_softmax）
10         - criterion: 损失函数，例如 LabelSmoothing 或 CrossEntropyLoss
11         """
12         self.generator = generator
13         self.criterion = criterion
14
15     def __call__(self, x, y, norm):
16         """
17         计算给定输出和目标的损失。
18
19         参数：
20         - x: 模型输出，形状通常为 [batch_size, seq_len, d_model]
21         - y: 目标序列索引，形状为 [batch_size, seq_len]
22         - norm: 归一化系数（例如 batch 内 token 总数）
23
24         功能：
25         1. 将模型输出 x 经过 generator（通常生成 log_prob 或 logits）。
26         2. 将 x 和 y 展平 (view)，方便损失函数逐 token 计算：

```

```

27         - x: [batch_size*seq_len, vocab_size]
28         - y: [batch_size*seq_len]
29     3. 计算损失 sloss, 并除以归一化系数 norm。
30     4. 返回两个值:
31         - sloss.data * norm: 未归一化的损失标量 (仅用于记录/打印)
32         - sloss: 可用于 backward() 的可微损失
33     """
34     x = self.generator(x) # 通过生成器转换输出
35     sloss = (
36         self.criterion(
37             x.contiguous().view(-1, x.size(-1)), # 展平成 [batch*seq_len,
vocab_size]
38             y.contiguous().view(-1) # 展平成 [batch*seq_len]
39         )
40         / norm # 按归一化系数缩放
41     )
42     return sloss.data * norm, sloss # 返回标量 loss 和可反向传播的 loss

```

## 贪心解码



这段代码使用 **贪心解码** 来预测翻译, 目的是为了简化处理。

### 代码块

```

1  def greedy_decode(model, src, src_mask, max_len, start_symbol):
2      """
3      使用贪心解码从模型生成序列。
4
5      参数:
6      - model: Transformer 或其他序列生成模型
7      - src: 输入序列张量, 形状 [1, src_len]
8      - src_mask: 输入序列掩码, 用于忽略 padding
9      - max_len: 最大生成长度
10     - start_symbol: 序列起始标记的索引
11
12     返回:
13     - ys: 生成的目标序列 (包含起始符和生成的 token)
14     """
15
16     # 对输入序列进行编码, 得到 memory 表示
17     memory = model.encode(src, src_mask)
18
19     # 初始化生成序列 ys, 起始符为 start_symbol
20     # ys 形状为 [1, 1]
21     ys = torch.zeros(1, 1).fill_(start_symbol).type_as(src.data)

```

```

22
23     # 循环生成序列，每次生成一个 token
24     for i in range(max_len - 1):
25         # 对当前生成序列 ys 解码，得到输出表示 out
26         # subsequent_mask 用于保证每个位置只能看到当前位置及之前的位置
27         out = model.decode(
28             memory,
29             src_mask,
30             ys,
31             subsequent_mask(ys.size(1)).type_as(src.data)
32         )
33
34         # 通过 generator 得到最后一个位置的概率分布
35         prob = model.generator(out[:, -1])
36
37         # 贪心选择概率最大的 token 作为下一步输出
38         _, next_word = torch.max(prob, dim=1)
39         next_word = next_word.data[0] # 提取整数索引
40
41         # 将新生成的 token 拼接到 ys 后面
42         ys = torch.cat(
43             [ys, torch.zeros(1, 1).type_as(src.data).fill_(next_word)], dim=1
44         )
45
46     # 返回生成的完整序列 ys
47     return ys
48

```

#### 代码块

```

1     # 训练一个简单的序列复制任务 (copy task)
2
3     def example_simple_model():
4         """
5         构建并训练一个小型 Transformer 模型用于序列复制任务。
6         输入序列随机生成，目标是输出与输入相同的序列。
7         """
8
9         # 定义词表大小
10        V = 11 # 包含 0 ~ 10 共 11 个符号
11
12        # 定义损失函数，这里不使用标签平滑 (smoothing=0.0)
13        criterion = LabelSmoothing(size=V, padding_idx=0, smoothing=0.0)
14
15        # 构建 Transformer 模型
16        # 输入词表大小 = 输出词表大小 = V，编码/解码层数 N=2

```

```

17 model = make_model(V, V, N=2)
18
19 # 定义优化器
20 optimizer = torch.optim.Adam(
21     model.parameters(), lr=0.5, betas=(0.9, 0.98), eps=1e-9
22 )
23
24 # 定义学习率调度器 (使用前面定义的 rate 函数和 warmup)
25 lr_scheduler = LambdaLR(
26     optimizer=optimizer,
27     lr_lambda=lambda step: rate(
28         step, model_size=model.src_embed[0].d_model, factor=1.0, warmup=400
29     ),
30 )
31
32 # 训练参数
33 batch_size = 80 # 每个 batch 序列数
34
35 # 训练 20 个 epoch
36 for epoch in range(20):
37     # 模型训练模式
38     model.train()
39     run_epoch(
40         data_gen(V, batch_size, 20), # 生成训练数据
41         model, # 模型
42         SimpleLossCompute(model.generator, criterion), # 损失计算
43         optimizer, # 优化器
44         lr_scheduler, # 学习率调度器
45         mode="train", # 训练模式
46     )
47
48     # 模型评估模式
49     model.eval()
50     run_epoch(
51         data_gen(V, batch_size, 5), # 生成验证数据
52         model,
53         SimpleLossCompute(model.generator, criterion),
54         DummyOptimizer(), # 不更新参数
55         DummyScheduler(), # 不调度学习率
56         mode="eval", # 评估模式
57     )[0]
58
59 # 使用训练好的模型进行一次贪心解码测试
60 model.eval()
61 src = torch.LongTensor([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]) # 测试输入序列
62 max_len = src.shape[1] # 最大生成长度 =

```

输入长度

```

63     src_mask = torch.ones(1, 1, max_len) # 输入掩码 (全 1,
      表示无 padding)
64     print(
65         greedy_decode(model, src, src_mask, max_len=max_len, start_symbol=0)
66     ) # 打印模型生成序列
67
68     execute_example(example_simple_model) # 可调用执行整个例子

```

```

Epoch Step: 1 | Accumulation Step: 2 | Loss: 3.06 | Tokens / Sec: 1414.3 | Learning Rate: 5.5e-06
Epoch Step: 1 | Accumulation Step: 2 | Loss: 2.09 | Tokens / Sec: 1648.9 | Learning Rate: 6.1e-05
Epoch Step: 1 | Accumulation Step: 2 | Loss: 1.72 | Tokens / Sec: 1652.2 | Learning Rate: 1.2e-04
Epoch Step: 1 | Accumulation Step: 2 | Loss: 1.49 | Tokens / Sec: 1501.8 | Learning Rate: 1.7e-04
Epoch Step: 1 | Accumulation Step: 2 | Loss: 0.99 | Tokens / Sec: 1691.6 | Learning Rate: 2.3e-04
Epoch Step: 1 | Accumulation Step: 2 | Loss: 0.60 | Tokens / Sec: 1729.7 | Learning Rate: 2.8e-04
Epoch Step: 1 | Accumulation Step: 2 | Loss: 0.34 | Tokens / Sec: 1674.2 | Learning Rate: 3.4e-04
Epoch Step: 1 | Accumulation Step: 2 | Loss: 0.21 | Tokens / Sec: 1782.1 | Learning Rate: 3.9e-04
Epoch Step: 1 | Accumulation Step: 2 | Loss: 0.12 | Tokens / Sec: 1740.3 | Learning Rate: 4.5e-04
Epoch Step: 1 | Accumulation Step: 2 | Loss: 0.10 | Tokens / Sec: 1467.8 | Learning Rate: 5.0e-04
Epoch Step: 1 | Accumulation Step: 2 | Loss: 0.12 | Tokens / Sec: 1755.9 | Learning Rate: 5.6e-04
Epoch Step: 1 | Accumulation Step: 2 | Loss: 0.24 | Tokens / Sec: 1685.8 | Learning Rate: 6.1e-04
Epoch Step: 1 | Accumulation Step: 2 | Loss: 0.10 | Tokens / Sec: 1636.6 | Learning Rate: 6.7e-04
Epoch Step: 1 | Accumulation Step: 2 | Loss: 0.09 | Tokens / Sec: 1752.0 | Learning Rate: 7.2e-04
Epoch Step: 1 | Accumulation Step: 2 | Loss: 0.10 | Tokens / Sec: 1739.3 | Learning Rate: 7.8e-04
Epoch Step: 1 | Accumulation Step: 2 | Loss: 0.14 | Tokens / Sec: 1786.8 | Learning Rate: 8.3e-04
Epoch Step: 1 | Accumulation Step: 2 | Loss: 0.10 | Tokens / Sec: 1807.6 | Learning Rate: 8.9e-04
Epoch Step: 1 | Accumulation Step: 2 | Loss: 0.17 | Tokens / Sec: 1694.8 | Learning Rate: 9.4e-04
Epoch Step: 1 | Accumulation Step: 2 | Loss: 0.09 | Tokens / Sec: 1628.0 | Learning Rate: 1.0e-03
Epoch Step: 1 | Accumulation Step: 2 | Loss: 0.06 | Tokens / Sec: 1703.3 | Learning Rate: 1.1e-03
tensor([[0, 2, 3, 2, 4, 5, 6, 7, 8, 9]])

```

## 复杂示例



现在我们考虑一个真实的例子 —— 使用 **德语-英语翻译任务**（这里与原来的数据不一致，因为**下载不了**）。这个任务相比论文中使用的 **WMT 翻译任务** 要小得多，但它能够完整地展示整个系统的运行过程。

我们还会演示如何使用 **多 GPU 并行处理** 来显著加快训练速度。

## 数据加载



我们将从 **github** 上加载数据集，并使用 **spacy** 进行分词。

代码块

```

1  def load_tokenizers():
2      """
3      加载德语和英语的 spaCy 分词器
4      如果本地没有对应的模型，则自动下载
5      """

```

```

6     try:
7         spacy_de = spacy.load("de_core_news_sm") # 加载德语分词器
8     except IOError:
9         os.system("python -m spacy download de_core_news_sm") # 如果没有就下载
10        spacy_de = spacy.load("de_core_news_sm")
11
12    try:
13        spacy_en = spacy.load("en_core_web_sm") # 加载英语分词器
14    except IOError:
15        os.system("python -m spacy download en_core_web_sm") # 如果没有就下载
16        spacy_en = spacy.load("en_core_web_sm")
17
18    return spacy_de, spacy_en

```

代码块

```

1  def yield_tokens(data_iter, tokenizer, index):
2      """
3      从数据迭代器中逐句取出并分词，返回生成器
4      index=0 表示取德语，index=1 表示取英语
5      """
6      for from_to_tuple in data_iter:
7          yield tokenizer(from_to_tuple[index])

```

代码块

```

1  import torchtext.datasets as datasets
2  from torchtext.datasets import Multi30k, multi30k
3
4
5  def build_vocabulary(spacy_de, spacy_en):
6      """
7      构建德语和英语的词表
8      使用 Multi30k 数据集 (train + valid)，并设置词频阈值 min_freq=2
9      """
10
11     def tokenize_de(text):
12         return tokenize(text, spacy_de)
13
14     def tokenize_en(text):
15         return tokenize(text, spacy_en)
16
17     print("Building German Vocabulary ...")
18
19     # 修改默认的 Multi30k 数据下载地址，指向 GitHub

```

```

20     multi30k.URL["train"] =
    "https://raw.githubusercontent.com/neycher/small_DL_repo/master/datasets/Multi3
    0k/training.tar.gz"
21     multi30k.URL["valid"] =
    "https://raw.githubusercontent.com/neycher/small_DL_repo/master/datasets/Multi3
    0k/validation.tar.gz"
22
23     # 加载 train 和 valid 数据集
24     train = datasets.Multi30k(split='train', language_pair=('de', 'en'))
25     val = datasets.Multi30k(split='valid', language_pair=('de', 'en'))
26
27     # 构建德语词表
28     vocab_src = build_vocab_from_iterator(
29         yield_tokens(train + val, tokenize_de, index=0),
30         min_freq=2, # 词频小于 2 的词丢弃
31         specials=["<s>", "</s>", "<blank>", "<unk>"], # 特殊符号
32     )
33
34     print("Building English Vocabulary ...")
35
36     # 再次加载数据集 (因为上面迭代器已经被消耗)
37     train = datasets.Multi30k(split='train', language_pair=('de', 'en'))
38     val = datasets.Multi30k(split='valid', language_pair=('de', 'en'))
39
40     # 构建英语词表
41     vocab_tgt = build_vocab_from_iterator(
42         yield_tokens(train + val, tokenize_en, index=1),
43         min_freq=2,
44         specials=["<s>", "</s>", "<blank>", "<unk>"],
45     )
46
47     # 设置默认索引 (如果遇到未登录词, 则替换为 <unk>)
48     vocab_src.set_default_index(vocab_src["<unk>"])
49     vocab_tgt.set_default_index(vocab_tgt["<unk>"])
50
51     return vocab_src, vocab_tgt
52
53
54 def load_vocab(spacy_de, spacy_en):
55     """
56     加载词表, 如果不存在则重新构建并保存
57     """
58     if not exists("vocab.pt"):
59         vocab_src, vocab_tgt = build_vocabulary(spacy_de, spacy_en)
60         torch.save((vocab_src, vocab_tgt), "vocab.pt")
61     else:
62         vocab_src, vocab_tgt = torch.load("vocab.pt")

```

```

63
64     print("Finished.\nVocabulary sizes:")
65     print(len(vocab_src))  # 输出德语词表大小
66     print(len(vocab_tgt))  # 输出英语词表大小
67     return vocab_src, vocab_tgt
68
69
70 if is_interactive_notebook():
71     # 在 notebook 环境下运行示例
72     spacy_de, spacy_en = show_example(load_tokenizers)
73     vocab_src, vocab_tgt = show_example(load_vocab, args=[spacy_de, spacy_en])
74
75 #结果如下
76 #Building German Vocabulary ...
77 #Building English Vocabulary ...
78 #Finished.
79 #Vocabulary sizes:
80 #8185
81 #6291

```



批处理对速度影响非常大。我们希望批次划分得非常均匀，并且填充（padding）尽可能少。为了做到这一点，我们必须对默认的 TorchText 批处理机制做一些改动。这个代码对它的默认批处理进行了修补，确保我们在寻找紧凑批次时能遍历足够多的句子。

## 迭代器

代码块

```

1  def collate_batch(
2      batch,
3      src_pipeline,  # 处理源语言文本的函数（如分词函数）
4      tgt_pipeline,  # 处理目标语言文本的函数
5      src_vocab,     # 源语言词表
6      tgt_vocab,     # 目标语言词表
7      device,        # 张量所在设备（CPU 或 GPU）
8      max_padding=128, # 最大序列长度（用于 padding）
9      pad_id=2,       # 填充 token 的索引，一般对应 <blank>
10 ):
11     # <s> token id, 用于序列起始
12     bs_id = torch.tensor([0], device=device)
13     # </s> token id, 用于序列结束
14     eos_id = torch.tensor([1], device=device)
15
16     # 存放处理后的源语言和目标语言序列

```



```

17     src_list, tgt_list = [], []
18
19     # 遍历 batch 中的每个样本, batch 中每个元素是 (_src, _tgt)
20     for (_src, _tgt) in batch:
21         # 处理源语言序列
22         processed_src = torch.cat(
23             [
24                 bs_id, # 在开头加 <s>
25                 torch.tensor(
26                     src_vocab(src_pipeline(_src)), # 分词并转换为索引
27                     dtype=torch.int64,
28                     device=device,
29                 ),
30                 eos_id, # 在末尾加 </s>
31             ],
32             0, # 沿着第 0 维拼接
33         )
34
35         # 处理目标语言序列
36         processed_tgt = torch.cat(
37             [
38                 bs_id,
39                 torch.tensor(
40                     tgt_vocab(tgt_pipeline(_tgt)),
41                     dtype=torch.int64,
42                     device=device,
43                 ),
44                 eos_id,
45             ],
46             0,
47         )
48
49         # 对源语言序列进行 padding, 使其长度为 max_padding
50         # 注意: 如果序列长度超过 max_padding, 会出现负数 padding 长度, 可能覆盖原值
51         src_list.append(
52             pad(
53                 processed_src,
54                 (0, max_padding - len(processed_src)), # 在末尾填充
55                 value=pad_id,
56             )
57         )
58
59         # 对目标语言序列进行同样的 padding
60         tgt_list.append(
61             pad(
62                 processed_tgt,
63                 (0, max_padding - len(processed_tgt)),

```

```

64         value=pad_id,
65     )
66 )
67
68 # 将列表中的序列堆叠成 batch 张量, 形状为 [batch_size, max_padding]
69 src = torch.stack(src_list)
70 tgt = torch.stack(tgt_list)
71
72 return (src, tgt) # 返回处理好的源语言和目标语言 batch

```

#### 代码块

```

1  def create_data_loaders(
2      device,          # 模型运行设备 (CPU 或 GPU)
3      vocab_src,        # 源语言词表
4      vocab_tgt,        # 目标语言词表
5      spacy_de,        # 德语 spaCy 分词器
6      spacy_en,        # 英语 spaCy 分词器
7      batch_size=12000, # 批次大小
8      max_padding=128,  # 最大序列长度, 用于 padding
9      is_distributed=True, # 是否使用分布式训练
10 ):
11     # 定义德语分词函数
12     def tokenize_de(text):
13         return tokenize(text, spacy_de)
14
15     # 定义英语分词函数
16     def tokenize_en(text):
17         return tokenize(text, spacy_en)
18
19     # 定义 collate_fn, 用于 DataLoader 合并 batch
20     def collate_fn(batch):
21         return collate_batch(
22             batch,
23             tokenize_de,          # 源语言分词
24             tokenize_en,          # 目标语言分词
25             vocab_src,             # 源语言词表
26             vocab_tgt,             # 目标语言词表
27             device,               # 张量设备
28             max_padding=max_padding, # 最大 padding 长度
29             pad_id=vocab_src.get_stoi()["<blank>"], # 填充 token 索引
30         )
31
32     # 加载 Multi30k 数据集, 返回迭代器
33     train_iter, valid_iter, test_iter = datasets.Multi30k(
34         language_pair=("de", "en")

```

```

35     )
36
37     # 转换为 map-style dataset (支持索引访问) ,
38     # DistributedSampler 需要知道 dataset 的长度
39     train_iter_map = to_map_style_dataset(train_iter)
40     # 分布式训练使用 DistributedSampler, 否则为 None
41     train_sampler = (
42         DistributedSampler(train_iter_map) if is_distributed else None
43     )
44
45     # 验证集也做同样处理
46     valid_iter_map = to_map_style_dataset(valid_iter)
47     valid_sampler = (
48         DistributedSampler(valid_iter_map) if is_distributed else None
49     )
50
51     # 创建训练 DataLoader
52     train_dataloader = DataLoader(
53         train_iter_map,
54         batch_size=batch_size,
55         shuffle=(train_sampler is None), # 如果没有 sampler, 则随机打乱
56         sampler=train_sampler,          # 分布式采样器
57         collate_fn=collate_fn,          # 合并 batch 函数
58     )
59
60     # 创建验证 DataLoader
61     valid_dataloader = DataLoader(
62         valid_iter_map,
63         batch_size=batch_size,
64         shuffle=(valid_sampler is None),
65         sampler=valid_sampler,
66         collate_fn=collate_fn,
67     )
68
69     # 返回训练和验证 DataLoader
70     return train_dataloader, valid_dataloader

```

## 训练

代码块

```

1  def train_worker(gpu, ngpus_per_node, vocab_src, vocab_tgt, spacy_de,
2     spacy_en, config,
3     is_distributed=False,
4     ):
5     # 打印当前进程使用的 GPU

```

```

5     print(f"Train worker process using GPU: {gpu} for training", flush=True)
6     torch.cuda.set_device(gpu) # 设置当前进程使用的 GPU
7
8     pad_idx = vocab_tgt["<blank>"] # 目标词表中的填充 token 索引
9     d_model = 512 # Transformer 模型的隐藏维度
10    model = make_model(len(vocab_src), len(vocab_tgt), N=6) # 构建 Transformer
    模型
11    model.cuda(gpu) # 将模型移动到当前 GPU
12    module = model # 在分布式训练时, 需要访问原始 model.module
13    is_main_process = True # 是否为主进程, 用于保存模型
14    if is_distributed:
15        # 初始化分布式进程组
16        dist.init_process_group(
17            "nccl", init_method="env://", rank=gpu, world_size=ngpus_per_node
18        )
19        model = DDP(model, device_ids=[gpu]) # 包装模型用于分布式训练
20        module = model.module # DDP 包装后的原始模型
21        is_main_process = gpu == 0 # GPU0 为主进程
22    # 定义损失函数: 标签平滑交叉熵
23    criterion = LabelSmoothing(
24        size=len(vocab_tgt), padding_idx=pad_idx, smoothing=0.1
25    )
26    criterion.cuda(gpu) # 将损失函数移动到 GPU
27    # 创建训练和验证数据加载器
28    train_dataloader, valid_dataloader = create_data_loaders(
29        gpu,
30        vocab_src,
31        vocab_tgt,
32        spacy_de,
33        spacy_en,
34        batch_size=config["batch_size"] // ngpus_per_node,
35        max_padding=config["max_padding"],
36        is_distributed=is_distributed,
37    )
38    # 定义优化器和学习率调度器
39    optimizer = torch.optim.Adam(
40        model.parameters(), lr=config["base_lr"], betas=(0.9, 0.98), eps=1e-9
41    )
42    lr_scheduler = LambdaLR(
43        optimizer=optimizer,
44        lr_lambda=lambda step: rate(
45            step, d_model, factor=1, warmup=config["warmup"]
46        ),
47    )
48    train_state = TrainState() # 用于记录训练状态 (例如步数、损失等)
49
50    for epoch in range(config["num_epochs"]):

```

```

51     if is_distributed:
52         # 分布式训练需要为每个 epoch 设置 sampler 的随机种子
53         train_data_loader.sampler.set_epoch(epoch)
54         valid_data_loader.sampler.set_epoch(epoch)
55
56     model.train() # 切换到训练模式
57     print(f"[GPU{gpu}] Epoch {epoch} Training ===", flush=True)
58
59     # 运行训练循环
60     _, train_state = run_epoch(
61         (Batch(b[0], b[1], pad_idx) for b in train_data_loader),
62         model,
63         SimpleLossCompute(module.generator, criterion),
64         optimizer,
65         lr_scheduler,
66         mode="train+log",
67         accum_iter=config["accum_iter"],
68         train_state=train_state,
69     )
70
71     GPUUtil.showUtilization() # 显示 GPU 使用情况
72
73     if is_main_process:
74         # 保存每个 epoch 的模型
75         file_path = "%s%.2d.pt" % (config["file_prefix"], epoch)
76         torch.save(module.state_dict(), file_path)
77
78     torch.cuda.empty_cache() # 清理 GPU 缓存
79
80     # 验证阶段
81     print(f"[GPU{gpu}] Epoch {epoch} Validation ===", flush=True)
82     model.eval() # 切换到评估模式
83     sloss = run_epoch(
84         (Batch(b[0], b[1], pad_idx) for b in valid_data_loader),
85         model,
86         SimpleLossCompute(module.generator, criterion),
87         DummyOptimizer(),
88         DummyScheduler(),
89         mode="eval",
90     )
91     print(sloss) # 打印验证损失
92     torch.cuda.empty_cache() # 清理 GPU 缓存
93     # 训练结束后保存最终模型
94     if is_main_process:
95         file_path = "%sfinal.pt" % config["file_prefix"]
96         torch.save(module.state_dict(), file_path)

```

代码块

```
1  def train_distributed_model(vocab_src, vocab_tgt, spacy_de, spacy_en, config):
2      from the_annotated_transformer import train_worker
3
4      ngpus = torch.cuda.device_count() # 获取可用 GPU 数量
5      os.environ["MASTER_ADDR"] = "localhost" # 分布式通信主机地址
6      os.environ["MASTER_PORT"] = "12356" # 分布式通信端口
7      print(f"Number of GPUs detected: {ngpus}")
8      print("Spawning training processes ...")
9
10     # 使用多进程训练, 每个 GPU 一个进程
11     mp.spawn(
12         train_worker,
13         nprocs=ngpus,
14         args=(ngpus, vocab_src, vocab_tgt, spacy_de, spacy_en, config, True),
15     )
```

代码块

```
1  def train_model(vocab_src, vocab_tgt, spacy_de, spacy_en, config):
2      # 根据配置选择分布式或单 GPU 训练
3      if config["distributed"]:
4          train_distributed_model(
5              vocab_src, vocab_tgt, spacy_de, spacy_en, config
6          )
7      else:
8          train_worker(
9              0, 1, vocab_src, vocab_tgt, spacy_de, spacy_en, config, False
10         )
11
12
13  def load_trained_model():
14      # 训练配置
15      config = {
16          "batch_size": 32,
17          "distributed": False,
18          "num_epochs": 8,
19          "accum_iter": 10,
20          "base_lr": 1.0,
21          "max_padding": 72,
22          "warmup": 3000,
23          "file_prefix": "multi30k_model_",
24      }
25      model_path = "multi30k_model_final.pt"
26
```

```

27     # 如果模型不存在，则先训练
28     if not exists(model_path):
29         train_model(vocab_src, vocab_tgt, spacy_de, spacy_en, config)
30
31     # 加载模型
32     model = make_model(len(vocab_src), len(vocab_tgt), N=6)
33     model.load_state_dict(torch.load("multi30k_model_final.pt"))
34     return model
35
36
37 if is_interactive_notebook():
38     # 如果在 Notebook 中运行，则直接加载训练好的模型
39     model = load_trained_model()

```

## 结果

代码块

```

1  # 函数：检查模型在验证集上的输出
2  def check_outputs(
3      valid_data_loader, # 验证集 DataLoader
4      model,             # 已训练的翻译模型
5      vocab_src,          # 源语言词表
6      vocab_tgt,          # 目标语言词表
7      n_examples=15,     # 要查看的样本数量
8      pad_idx=2,         # padding token 的索引
9      eos_string="</s>", # 句子结束符
10 ):
11     # 用于保存结果的列表，每个元素是一个元组
12     results = [()] * n_examples
13
14     # 遍历指定数量的样本
15     for idx in range(n_examples):
16         print("\nExample %d =====\n" % idx)
17
18         # 获取一个 batch 数据 (这里 batch_size = 1)
19         b = next(iter(valid_data_loader))
20
21         # 将 batch 数据转换为 Batch 对象 (通常封装了 src, tgt, mask 等)
22         rb = Batch(b[0], b[1], pad_idx)
23
24         # 对输入进行贪心解码 (这里生成长度最大为 64 的序列)
25         greedy_decode(model, rb.src, rb.src_mask, 64, 0)[0]
26
27         # 将源句子 token id 转为对应的单词，并去掉 pad

```

```

28     src_tokens = [vocab_src.get_itos()[x] for x in rb.src[0] if x !=
pad_idx]
29
30     # 将目标句子 token id 转为对应的单词, 并去掉 pad
31     tgt_tokens = [vocab_tgt.get_itos()[x] for x in rb.tgt[0] if x !=
pad_idx]
32
33     # 打印源句子
34     print("Source Text (Input) : " + " ".join(src_tokens).replace("\n",
""))
35
36     # 打印真实目标句子
37     print("Target Text (Ground Truth) : " + "
".join(tgt_tokens).replace("\n", ""))
38
39     # 模型输出解码 (生成长度最大为 72)
40     model_out = greedy_decode(model, rb.src, rb.src_mask, 72, 0)[0]
41
42     # 将模型输出的 token id 转为单词, 并截取到第一个句子结束符
43     model_txt = (
44         " ".join([vocab_tgt.get_itos()[x] for x in model_out if x !=
pad_idx])
45         .split(eos_string, 1)[0] + eos_string
46     )
47
48     # 打印模型预测输出
49     print("Model Output : " + model_txt.replace("\n", ""))
50
51     # 保存结果: 包含 batch、源句子、目标句子、模型输出 token id、模型输出文本
52     results[idx] = (rb, src_tokens, tgt_tokens, model_out, model_txt)
53
54     return results # 返回所有样本结果
55

```

#### 代码块

```

1  # 函数: 运行模型并查看几个示例输出
2  def run_model_example(n_examples=5):
3      global vocab_src, vocab_tgt, spacy_de, spacy_en
4
5      print("Preparing Data ...")
6
7      # 创建验证集 DataLoader (这里 batch_size=1, CPU 运行)
8      _, valid_data_loader = create_data_loaders(
9          torch.device("cpu"),
10         vocab_src,

```



```
11         vocab_tgt,
12         spacy_de,
13         spacy_en,
14         batch_size=1,
15         is_distributed=False,
16     )
17
18     print("Loading Trained Model ...")
19
20     # 创建模型 (与训练时相同架构)
21     model = make_model(len(vocab_src), len(vocab_tgt), N=6)
22
23     # 加载训练好的模型参数
24     model.load_state_dict(
25         torch.load("multi30k_model_final.pt", map_location=torch.device("cpu"))
26     )
27
28     print("Checking Model Outputs:")
29
30     # 检查模型输出, 返回示例数据
31     example_data = check_outputs(
32         valid_data_loader, model, vocab_src, vocab_tgt, n_examples=n_examples
33     )
34
35     return model, example_data # 返回模型和示例数据
36 run_model_example()
```

Preparing Data ...

Loading Trained Model ...

Checking Model Outputs:

Example 0 =====

Source Text (Input) : <s> Zwei Terrier spielen zu Hause auf dem Holzboden . </s>  
Target Text (Ground Truth) : <s> Two terriers play on the wood floor of their home . </s>  
Model Output : <s> Two black dogs play on a wooden floor . </s>

Example 1 =====

Source Text (Input) : <s> Eine Frau in einer blauen Bluse und Jeans wirft etwas in einen Müllcontainer . </s>  
Target Text (Ground Truth) : <s> A woman in a blue shirt and jeans is throwing something away in a dumpster . </s>  
Model Output : <s> A woman in a blue shirt and jeans throws something into a dumpster . </s>

Example 2 =====

Source Text (Input) : <s> Ein Boot mit Menschen und ihrem Hab und Gut befindet sich im Wasser . </s>  
Target Text (Ground Truth) : <s> A boat with people and their belongings is in the water . </s>  
Model Output : <s> A boat with people on and their belongings in the water . </s>

Example 3 =====

Source Text (Input) : <s> Eine Frau in einer gelben Jacke folgt zwei anderen Frauen . </s>  
...

Source Text (Input) : <s> Ein Rennkatamaran wird im Wasser auf einen <unk> gehoben . </s>  
Target Text (Ground Truth) : <s> A racing catamaran is lifted onto one <unk> in the water . </s>  
Model Output : <s> A racing biker in the water is being <unk> by the <unk> . </s>

```
(EncoderDecoder(
  (encoder): Encoder(
    (layers): ModuleList(
      (0): EncoderLayer(
        (self_attn): MultiHeadedAttention(
          (linears): ModuleList(
            (0): Linear(in_features=512, out_features=512, bias=True)
            (1): Linear(in_features=512, out_features=512, bias=True)
            (2): Linear(in_features=512, out_features=512, bias=True)
            (3): Linear(in_features=512, out_features=512, bias=True)
          )
          (dropout): Dropout(p=0.1, inplace=False)
        )
        (feed_forward): PositionwiseFeedForward(
          (w_1): Linear(in_features=512, out_features=2048, bias=True)
          (w_2): Linear(in_features=2048, out_features=512, bias=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
        (sublayer): ModuleList(
          (0): SublayerConnection(
            (norm): LayerNorm()
            (dropout): Dropout(p=0.1, inplace=False)
          )
          (1): SublayerConnection(
            (norm): LayerNorm()
            ...
            1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
            1,  1,  1,  5,  1,  1,  1,  1,  1,  1,  1,  5,  1,  1,
            1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  1,
            1,  1]),
          '<s> A racing biker in the water is being <unk> by the <unk> . </s>'))])
```

Output is truncated. View as a [scrollable element](#) or open in a [text editor](#). Adjust cell output [settings](#)...



即使使用贪心解码，翻译效果也相当不错。我们还可以进一步对其进行可视化，以观察注意力机制在每一层的表现情况。

代码块

```
1 import matplotlib.pyplot as plt
2 import seaborn as sns
3 import pandas as pd
4 import numpy as np
5
6 # 将矩阵转换成 pandas DataFrame (可选，主要用于生成行列标签，方便可视化)
7 def mt2df(m, max_row, max_col, row_tokens, col_tokens):
8     # 取矩阵的行列数，不能超过指定的最大值
9     rows = min(max_row, m.shape[0])
10    cols = min(max_col, m.shape[1])
11
12    # 创建 DataFrame，行列索引使用 token 名称 + 行号/列号，超出范围用 '<blank>'
13    df = pd.DataFrame(
```

```

14         m[:rows, :cols],
15         index=[f"{r:03d} {row_tokens[r] if len(row_tokens) > r else '<blank>'}"
    for r in range(rows)],
16         columns=[f"{c:03d} {col_tokens[c] if len(col_tokens) > c else
'<blank>'}" for c in range(cols)],
17     )
18     return df
19
20
21 # 可视化 Transformer 模型某一层的所有注意力头 (attention heads)
22 def visualize_layer_heads_together(model, layer, getter_fn, ntokens,
    row_tokens, col_tokens, max_dim=30):
23     # 使用 getter 函数获取指定层的注意力矩阵
24     attn = getter_fn(model, layer)
25
26     # 注意力矩阵维度: [batch, n_heads, seq_len, seq_len]
27     n_heads = attn.shape[1]
28     rows = min(max_dim, len(row_tokens))
29     cols = min(max_dim, len(col_tokens))
30
31     # 布局: 3 行, 每行显示大约 n_heads / 3 个 head
32     ncols = (n_heads + 1) // 3
33     nrows = 3
34     fig, axes = plt.subplots(nrows, ncols, figsize=(4*ncols, 4*nrows))
35     axes = axes.flatten() # 拉平成一维数组, 方便循环索引
36
37     for h in range(n_heads):
38         # 取第 0 个样本的第 h 个 head 的注意力矩阵
39         mat = attn[0, h].detach().cpu().numpy()[:rows, :cols].astype(float)
40
41         # 行列标签
42         row_labels = [f"{r:03d} {row_tokens[r] if len(row_tokens) > r else
'<blank>'}" for r in range(rows)]
43         col_labels = [f"{c:03d} {col_tokens[c] if len(col_tokens) > c else
'<blank>'}" for c in range(cols)]
44
45         # 使用 seaborn 绘制 heatmap
46         sns.heatmap(
47             mat,
48             cmap="viridis",
49             ax=axes[h],
50             xticklabels=col_labels,
51             yticklabels=row_labels
52         )
53         axes[h].set_title(f"Head {h+1}")
54         axes[h].set_xlabel("Cols")
55         axes[h].set_ylabel("Rows")

```

```

56         axes[h].tick_params(axis='x', rotation=90, labelsz=6)
57         axes[h].tick_params(axis='y', rotation=0, labelsz=6)
58
59         # 如果 head 数量不是 n_rows*n_cols, 剩余 subplot 关闭显示
60         for ax in axes[n_heads:]:
61             ax.axis("off")
62
63         plt.suptitle(f"Layer {layer+1} Attention Heads", fontsize=16)
64         plt.tight_layout(rect=[0, 0, 1, 0.95])
65         plt.show()
66
67
68     # 获取 encoder 某一层的自注意力矩阵
69     def get_encoder(model, layer):
70         return model.encoder.layers[layer].self_attn.attn
71
72     # 获取 decoder 某一层的自注意力矩阵
73     def get_decoder_self(model, layer):
74         return model.decoder.layers[layer].self_attn.attn
75
76     # 获取 decoder 某一层的 encoder-decoder 注意力矩阵 (对 encoder 输出的注意力)
77     def get_decoder_src(model, layer):
78         return model.decoder.layers[layer].src_attn.attn

```

## 编码器自注意力

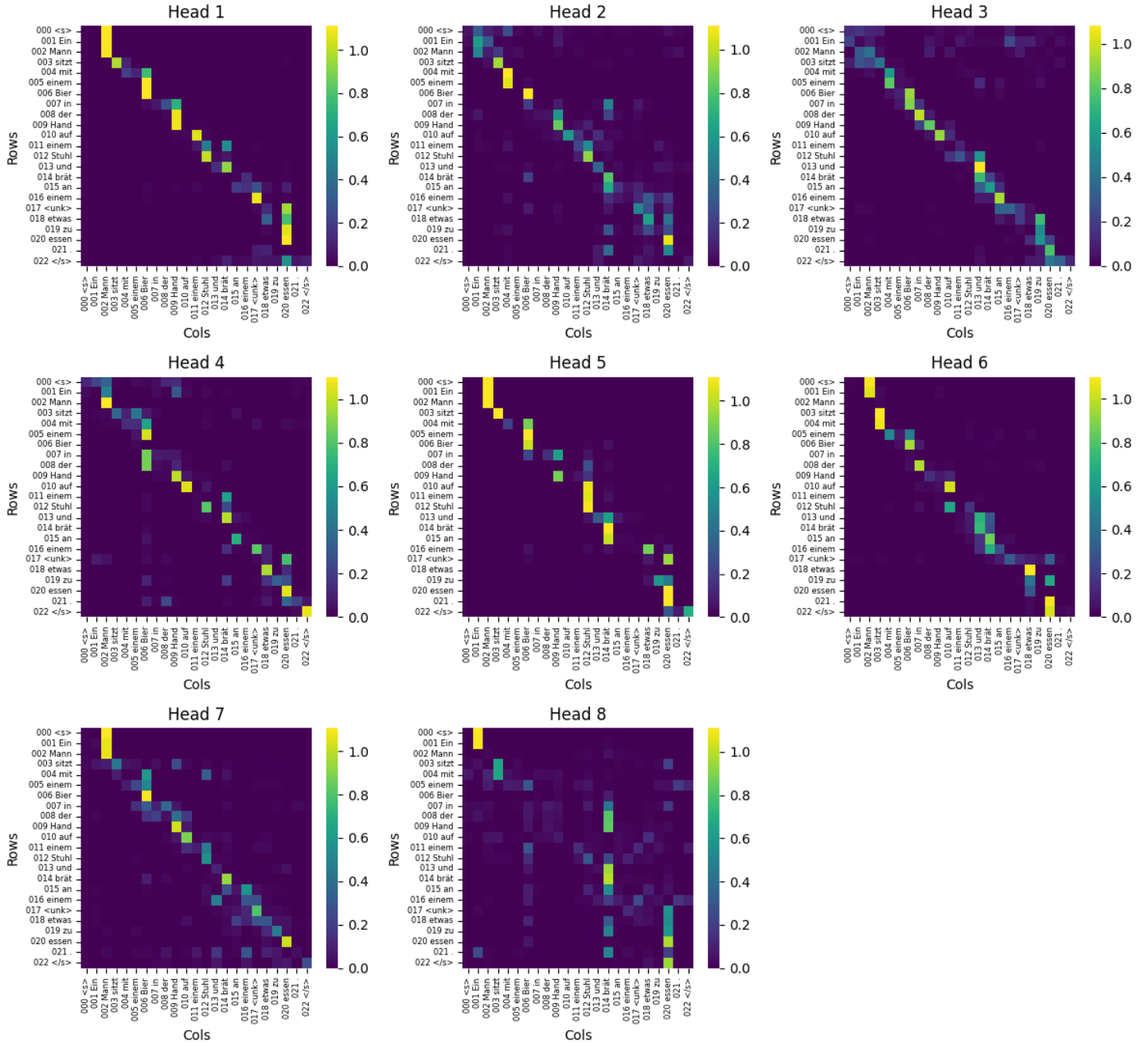
代码块

```

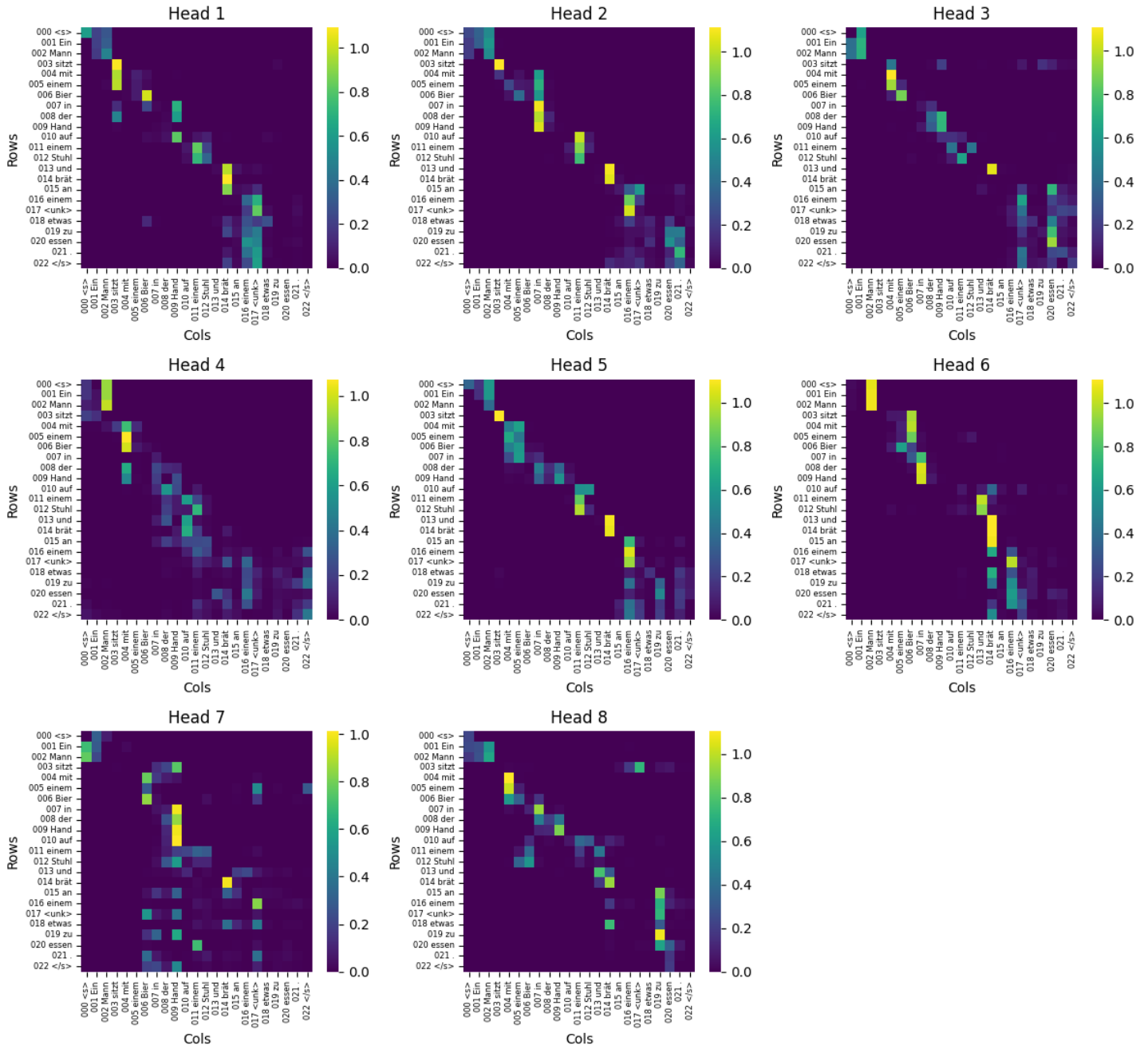
1     # 绘制指定层的 encoder 注意力
2     def viz_encoder_self_matplotlib():
3         model, example_data = run_model_example(n_examples=1)
4         example = example_data[-1]
5         # 绘制第 0,2 层, 每层所有 heads 放在同一张图里
6         for layer in [0,2]:
7             visualize_layer_heads_together(model, layer, get_encoder,
len(example[1]), example[1], example[1])
8         # Notebook 里直接显示
9         viz_encoder_self_matplotlib()

```

## Layer 1 Attention Heads



## Layer 3 Attention Heads

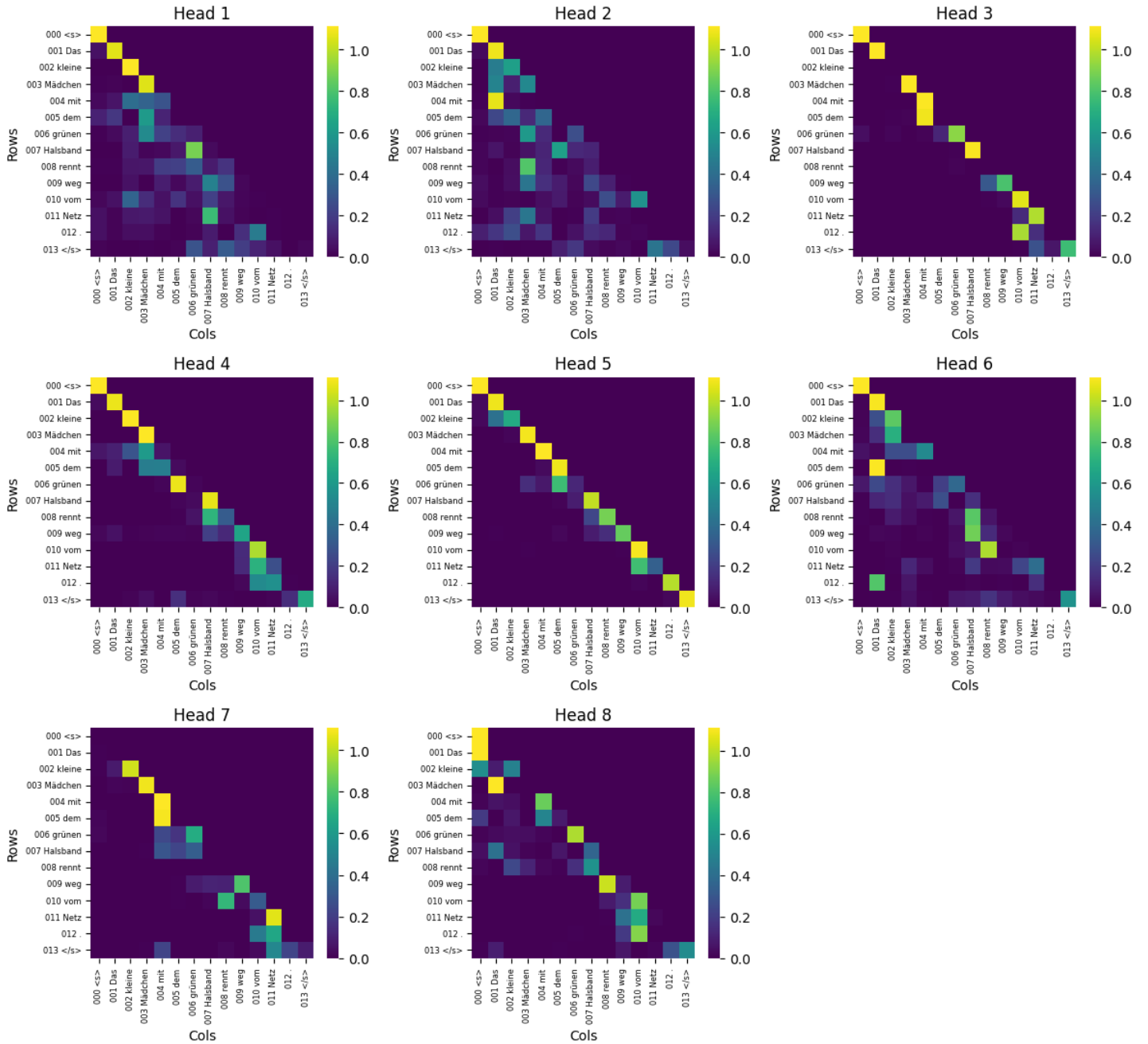


## 解码器自注意力

代码块

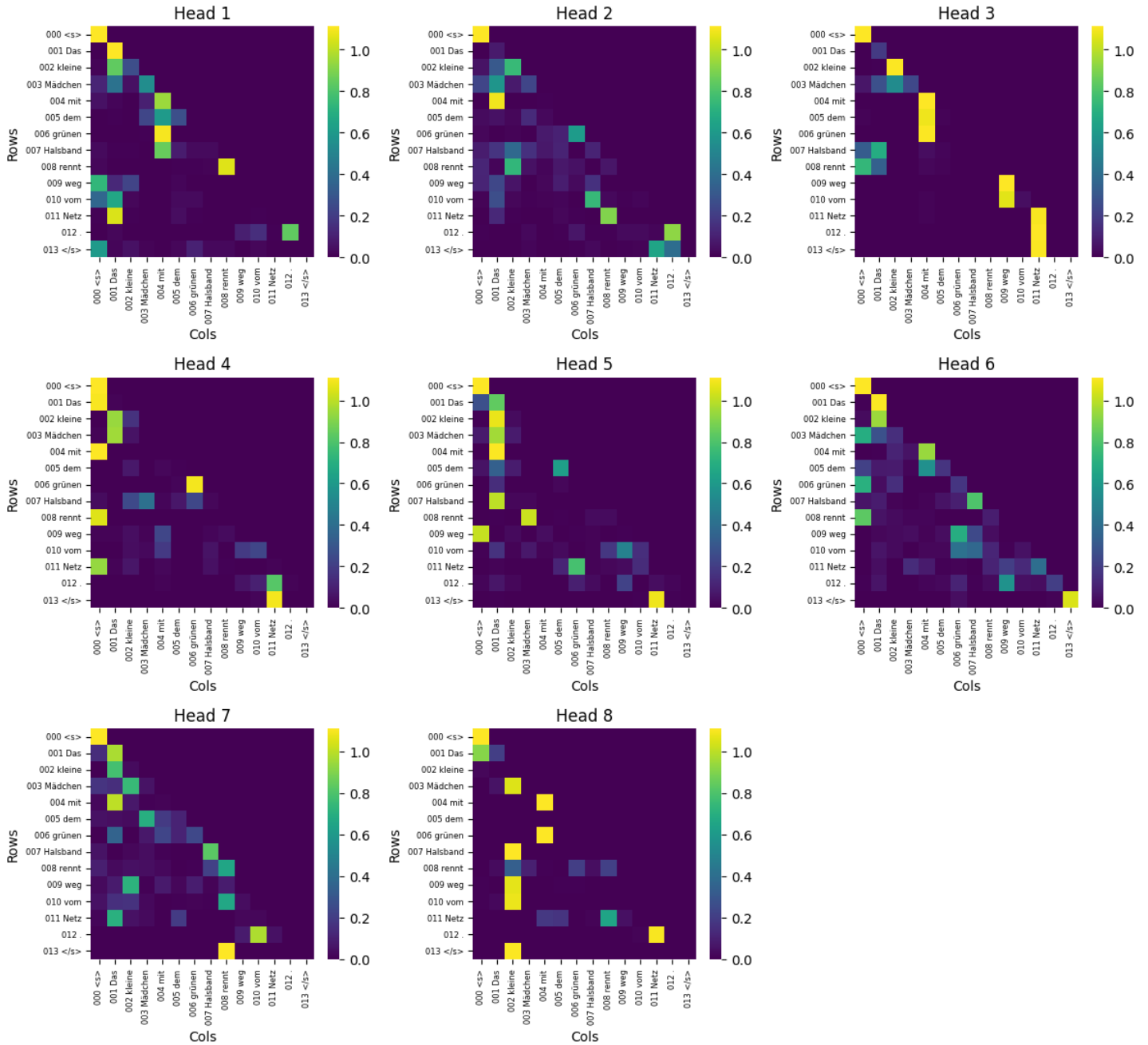
```
1 # 绘制指定层的 decoder 自注意力
2 def viz_decoder_self_matplotlib():
3     model, example_data = run_model_example(n_examples=1)
4     example = example_data[-1]
5     # 绘制第 0, 2 层, 每层所有 heads 放在同一张图里
6     for layer in [0, 2]:
7         visualize_layer_heads_together(model, layer, get_decoder_self,
8                                         len(example[1]), example[1], example[1])
9     # Notebook 里直接显示
```

## Layer 1 Attention Heads





## Layer 3 Attention Heads



## 解码器对源序列的注意力

代码块

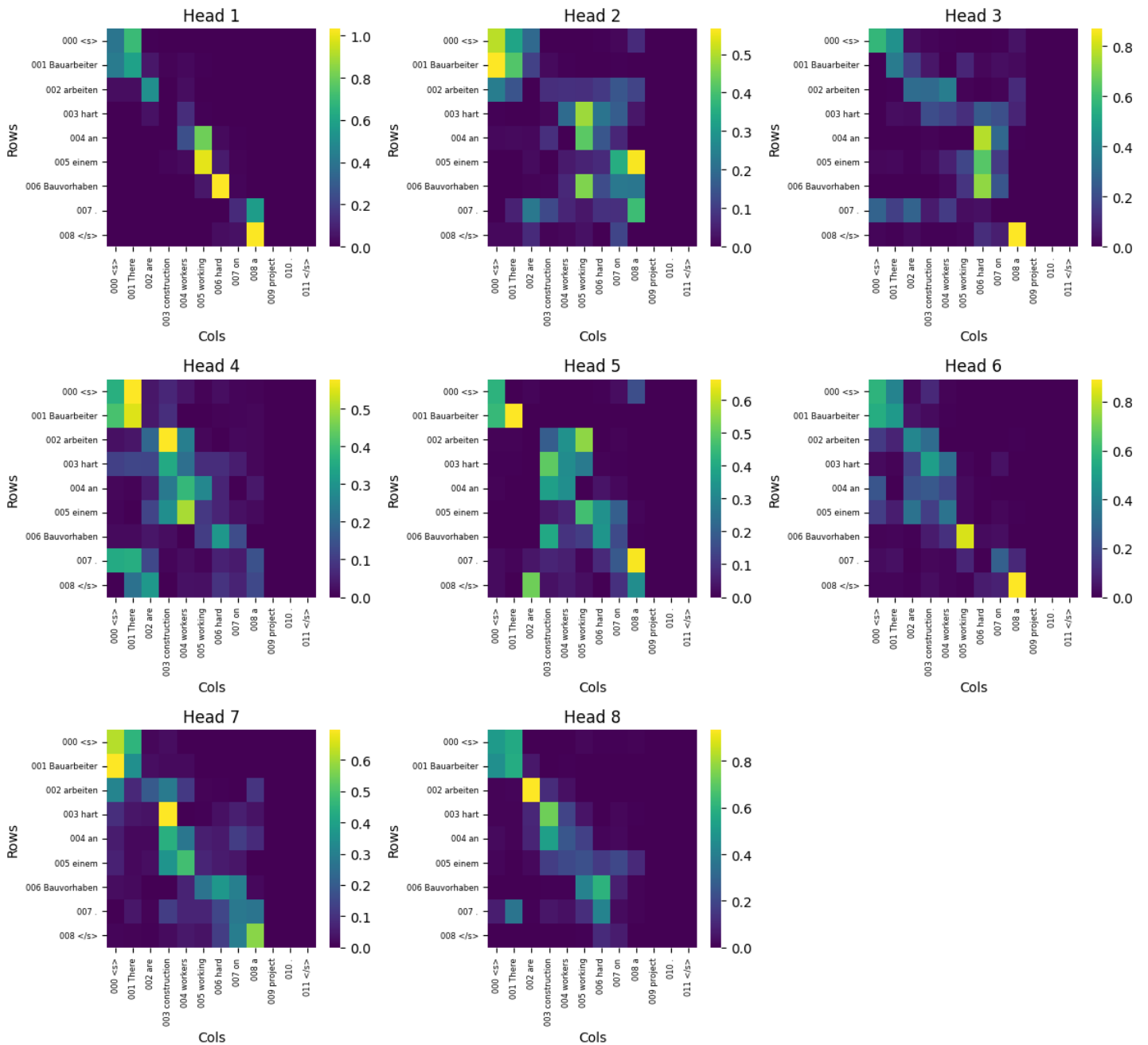
```
1 # 绘制指定层的 decoder 源-目标注意力 (Matplotlib 风格)
2 def viz_decoder_src_matplotlib():
3     model, example_data = run_model_example(n_examples=1)
4     example = example_data[-1] # 保留原来的最后一个样例
5
6     # 绘制第 0, 2 层, 每层所有 heads 放在同一张图里
7     for layer in [0, 2]:
8         visualize_layer_heads_together(
9             model,
10             layer,
```

```

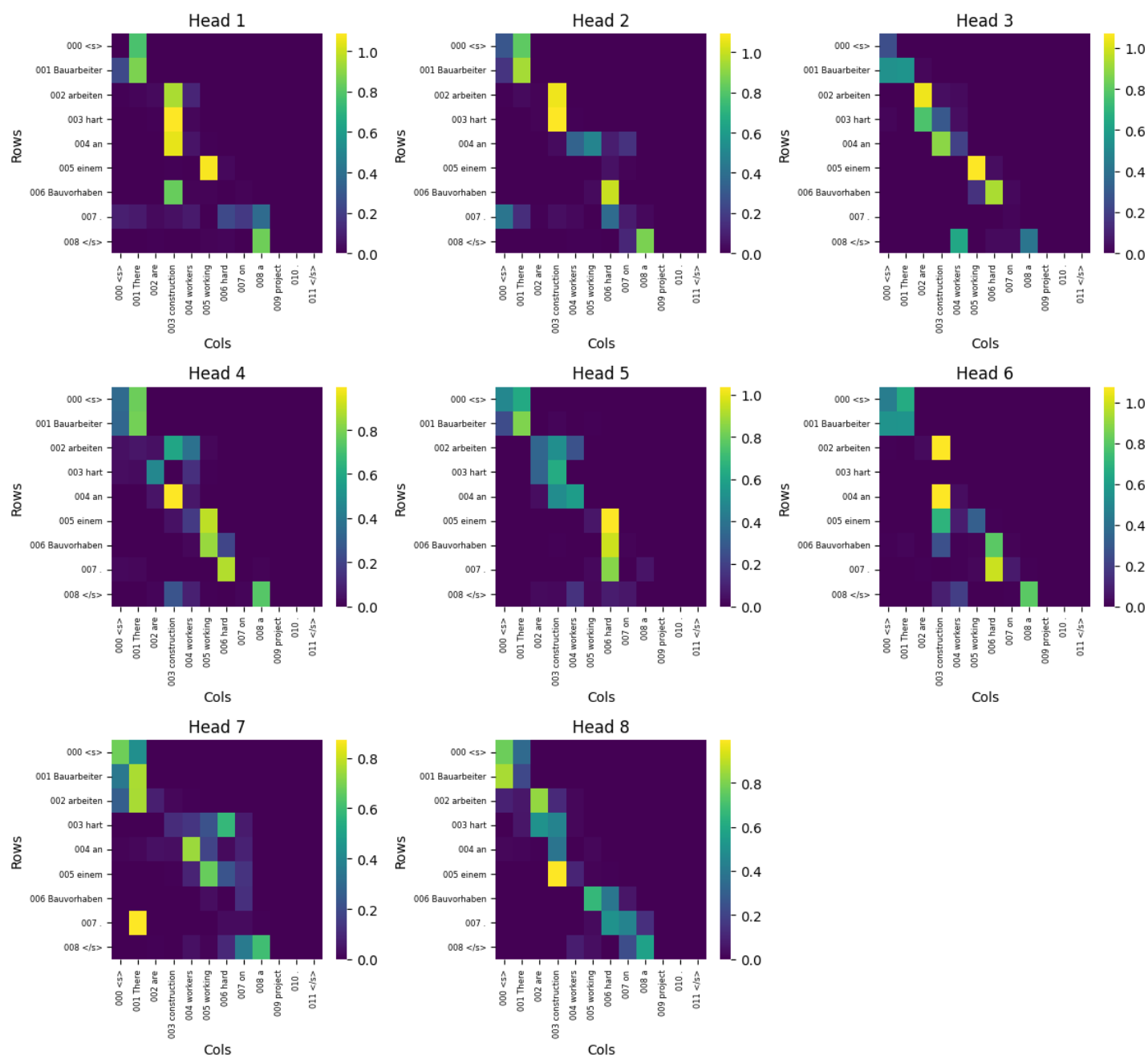
11     get_decoder_src,
12     max(len(example[1]), len(example[2])), # 保留原来的维度
13     example[1],
14     example[2]
15 )
16
17 # Notebook 里直接显示
18 viz_decoder_src_matplotlib()

```

## Layer 1 Attention Heads



## Layer 3 Attention Heads



## 其他组成部分



这部分主要介绍了 Transformer 模型本身。还有四个方面我们没有明确讲到。同时，在 **OpenNMT-py** 中，这些额外功能也已经实现了。

1. **BPE / Word-piece**: 我们可以先用一个库把数据预处理成 **子词单元**。可以参考 Rico Sennrich 提供的 **subword-nmt** 实现。
2. **Shared Embeddings**: 当使用 **BPE** 并采用共享词表时，我们可以在 **源端 / 目标端 / 生成器** 之间共享同一套权重向量。具体细节可参考相关文献。
3. **束搜索 (Beam Search)**: 这部分内容比较复杂，这里不展开讲。可以参考 **OpenNMT-py** 中的 PyTorch 实现。

4. **模型平均 (Model Averaging)**：论文中通过对最后的  $k$  个 checkpoint 进行平均来实现集成效果。如果我们手上有多个模型，也可以在训练后进行同样的操作。