

USER

我正在学习 Deep Learning , 你能详细解释一下 Transformer 模型中的 Self-Attention 机制吗 ?

如果可以的话 , 请给我一个基于 PyTorch 的简化代码实现 , 方便我理解它的矩阵运算过程。

ASSISTANT

没问题 ! Self-Attention (自注意力机制) 是 Transformer 的核心组件 , 它解决了 RNN 无法并行计算和长距离依赖的问题。

1. 数学公式

注意力机制的计算公式如下 , 这是一个行内公式示例 : $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$ 。

其中 , d_k 是 Key 向量的维度。除以 $\sqrt{d_k}$ 是为了防止点积结果过大导致 Softmax 梯度消失。

下面是一个更复杂的独立公式块示例 :

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

其中每个 head 的计算方式为 :

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

2. PyTorch 代码实现

下面是一个简化版的 Multi-Head Self-Attention 实现。即便代码较长 , 现在的布局也能完美展示 :

```
import torch
import torch.nn as nn

class SelfAttention(nn.Module):
    def __init__(self, embed_size, heads):
        super(SelfAttention, self).__init__()
        self.embed_size = embed_size
        self.heads = heads
        self.head_dim = embed_size // heads

        assert (
            self.head_dim * heads == embed_size
        )
```

```
), "Embedding size needs to be divisible by heads"

# 定义线性层用来生成 Q, K, V
self.values = nn.Linear(self.head_dim, self.head_dim, bias=False)
self.keys = nn.Linear(self.head_dim, self.head_dim, bias=False)
self.queries = nn.Linear(self.head_dim, self.head_dim,
bias=False)
self.fc_out = nn.Linear(heads * self.head_dim, embed_size)
```

```
def forward(self, values, keys, query, mask):
    # 获取 batch size
    N = query.shape[0]
    value_len, key_len, query_len = values.shape[1], keys.shape[1],
query.shape[1]

    # 拆分 Heads
    values = values.reshape(N, value_len, self.heads, self.head_dim)
    keys = keys.reshape(N, key_len, self.heads, self.head_dim)
    query = query.reshape(N, query_len, self.heads, self.head_dim)

    # Einsum 矩阵乘法: query * keys
    energy = torch.einsum("nqhd,nkhd->nhqk", [query, keys])

    # 如果有掩码 (Mask) , 将不需要关注的位置设为负无穷
    if mask is not None:
        energy = energy.masked_fill(mask == 0, float("-1e20"))

    # Softmax 归一化
    attention = torch.softmax(energy / (self.embed_size ** (1 / 2)),
dim=3)

    # 加权求和: attention * values
    out = torch.einsum("nhql,nlhd->nqhd", [attention,
values]).reshape(
        N, query_len, self.heads * self.head_dim
    )

    return self.fc_out(out)
```

这个实现涵盖了矩阵拆分、爱因斯坦求和约定 (einsum) 以及 Mask 的处理。

USER

非常有帮助！代码里的 torch.einsum 用法确实很简洁。既然你提到了 Mask，如果是 Decoder 部分的 Mask，代码会有什么不同吗？

ASSISTANT

在 Decoder 中，我们需要用到 Look-ahead Mask (前瞻掩码)。

这是为了防止模型在预测当前词时“偷看”到后面的词。我们通常使用一个上三角矩阵来实现：

```
# 创建一个上三角矩阵掩码
trg_len = 10
trg_mask = torch.tril(torch.ones((trg_len, trg_len))).expand(
    N, 1, trg_len, trg_len
)

# 在 Attention 计算前应用
energy = energy.masked_fill(trg_mask == 0, float("-1e20"))
```

这样，位置 i 的 query 就只能关注到 0 到 i 位置的 key，而无法看到 $i+1$ 之后的信息。