MHA_MQA_GQA

1.总结

- 在 **MHA（Multi Head Attention）** 中，每个头有自己单独的 key-value 对；标准的多头注意力机制，h个Query、Key 和 Value 矩阵。
- 在 **MQA（Multi Query Attention）** 中只会有一组 key-value 对；多查询注意力的一种变体，也是用于自回归解码的一种注意力机制。与MHA不同的是，**MQA 让所有的头之间共享同一份 Key 和 Value 矩阵，每个头只单独保留了一份 Query 参数，从而大大减少 Key 和 Value 矩阵的参数量。**
- 在 **GQA（Grouped Query Attention）**中，会对 attention 进行分组操作，query 被分为 N 组，每个组共享一个 Key 和 Value 矩阵**GQA将查询头分成G组，每个组共享一个Key 和 Value 矩阵。**GQA-G是指具有G组的grouped-query attention。GQA-1具有单个组，因此具有单个Key 和 Value，等效于MQA。而GQA-H具有与头数相等的组，等效于MHA。

GQA介于MHA和MQA之间。GQA 综合 MHA 和 MQA，既不损失太多性能，又能利用 MQA 的推理加速。不是所有 Q 头共享一组 KV，而是分组一定头数 Q 共享一组 KV，比如上图中就是两组 Q 共享一组 KV。

2. 代码实现

2.1 MHA

多头注意力机制是Transformer模型中的核心组件。在其设计中，"多头"意味着该机制并不只计算一种注意力权重，而是并行计算多种权重，每种权重都从不同的"视角"捕获输入的不同信息。

为输入序列中的每个元素计算 q, k, v，这是通过将输入词向量与三个权重矩阵相乘实现的：

$$q = xW_q$$

$$k = xW_k$$

$$v = xW_v$$

其中，x 是输入词向量，W_q、W_k 和 W_v 分别是 q、k、v 的权重矩阵。

计算 q、k 的注意力得分：$\text{score}(q, k) = \frac{q \cdot k^T}{\sqrt{d_k}}$其中，$d_k$ 是 k 的维度。

使用 softmax 得到注意力权重：$\text{Attention}(q, K) = \text{softmax}(\text{score}(q, k))$

使用注意力权重和 V，计算输出：$\text{Output} = \text{Attention}(q, K) \cdot V$

拼接多头输出，乘以 W_O，得到最终输出：
$\text{MultiHeadOutput} = \text{Concat}(\text{Output}_1, \text{Output}_2, \ldots, \text{Output}_H)W_O$

代码实现

```
import torch
from torch import nn
class MutiHeadAttention(torch.nn.Module):
    def __init__(self, hidden_size, num_heads):
        super(MutiHeadAttention, self).__init__()
        self.num_heads = num_heads
        self.head_dim = hidden_size // num_heads
```

```
        ## 初始化Q、K、V投影矩阵
        self.q_linear = nn.Linear(hidden_size, hidden_size)
        self.k_linear = nn.Linear(hidden_size, hidden_size)
        self.v_linear = nn.Linear(hidden_size, hidden_size)

        ## 输出线性层
        self.o_linear = nn.Linear(hidden_size, hidden_size)

    def forward(self, hidden_state, attention_mask=None):
        batch_size = hidden_state.size()[0]

        query = self.q_linear(hidden_state)
        key = self.k_linear(hidden_state)
        value = self.v_linear(hidden_state)

        query = self.split_head(query)
        key = self.split_head(key)
        value = self.split_head(value)

        ## 计算注意力分数
        attention_scores = torch.matmul(query, key.transpose(-1, -2)) /
torch.sqrt(torch.tensor(self.head_dim))

        if attention_mask != None:
            attention_scores += attention_mask * -1e-9

        ## 对注意力分数进行归一化
        attention_probs = torch.softmax(attention_scores, dim=-1)

        output = torch.matmul(attention_probs, value)

        ## 对注意力输出进行拼接
        output = output.transpose(-1, -2).contiguous().view(batch_size, -1,
self.head_dim * self.num_heads)

        output = self.o_linear(output)

        return output


    def split_head(self, x):
        batch_size = x.size()[0]
        return x.view(batch_size, -1, self.num_heads,
self.head_dim).transpose(1,2)
```

2.2 MQA

在计算多头注意力的时候，query仍然进行分头，和多头注意力机制相同，而key和value只有一个头。

正常情况在计算多头注意力分数的时候，query、key的维度是相同的，所以可以直接进行矩阵乘法，但是在多查询注意力（MQA）中，query的维度为`[batch_size, num_heads, seq_len, head_dim]`，key和value的维度为`[batch_size, 1, seq_len, head_dim]`。这样就无法直接进行矩阵的乘法，为了完成这一乘法，可以采用torch的广播乘法。

```
## 多查询注意力
import torch
from torch import nn
class MutiQueryAttention(torch.nn.Module):
    def __init__(self, hidden_size, num_heads):
        super(MutiQueryAttention, self).__init__()
        self.num_heads = num_heads
        self.head_dim = hidden_size // num_heads

        ## 初始化Q、K、V投影矩阵
        self.q_linear = nn.Linear(hidden_size, hidden_size)
        self.k_linear = nn.Linear(hidden_size, self.head_dim) ###
        self.v_linear = nn.Linear(hidden_size, self.head_dim) ###

        ## 输出线性层
        self.o_linear = nn.Linear(hidden_size, hidden_size)

    def forward(self, hidden_state, attention_mask=None):
        batch_size = hidden_state.size()[0]

        query = self.q_linear(hidden_state)
        key = self.k_linear(hidden_state)
        value = self.v_linear(hidden_state)

        query = self.split_head(query)
        key = self.split_head(key, 1)
        value = self.split_head(value, 1)

        ## 计算注意力分数
        attention_scores = torch.matmul(query, key.transpose(-1, -2)) /
torch.sqrt(torch.tensor(self.head_dim))

        if attention_mask != None:
            attention_scores += attention_mask * -1e-9

        ## 对注意力分数进行归一化
        attention_probs = torch.softmax(attention_scores, dim=-1)

        output = torch.matmul(attention_probs, value)

        output = output.transpose(-1, -2).contiguous().view(batch_size, -1,
self.head_dim * self.num_heads)

        output = self.o_linear(output)

        return output




    def split_head(self, x, head_num=None):

        batch_size = x.size()[0]

        if head_num == None:
```

```
            return x.view(batch_size, -1, self.num_heads,
self.head_dim).transpose(1,2)
        else:
            return x.view(batch_size, -1, head_num, self.head_dim).transpose(1,2)
```

相比于多头注意力，多查询注意力在W_k和W_v的维度映射上有所不同，还有就是计算注意力分数采用的是广播机制，计算最后的output也是广播机制，其他的与多头注意力完全相同。

2.3 GQA

GQA将MAQ中的key、value的注意力头数设置为一个能够被原本的注意力头数整除的一个数字，也就是group数。

不同的模型使用GQA有着不同的实现方式，但是总体的思路就是这么实现的，注意，***设置的组一定要能够被注意力头数整除。***

```
## 分组注意力查询
import torch
from torch import nn
class GroupQueryAttention(torch.nn.Module):
    def __init__(self, hidden_size, num_heads, group_num):
        super(MutiQueryAttention, self).__init__()
        self.num_heads = num_heads
        self.head_dim = hidden_size // num_heads
        self.group_num = group_num

        ## 初始化Q、K、V投影矩阵
        self.q_linear = nn.Linear(hidden_size, hidden_size)
        self.k_linear = nn.Linear(hidden_size, self.group_num * self.head_dim)
        self.v_linear = nn.Linear(hidden_size, self.group_num * self.head_dim)

        ## 输出线性层
        self.o_linear = nn.Linear(hidden_size, hidden_size)

    def forward(self, hidden_state, attention_mask=None):
        batch_size = hidden_state.size()[0]

        query = self.q_linear(hidden_state)
        key = self.k_linear(hidden_state)
        value = self.v_linear(hidden_state)

        query = self.split_head(query)
        key = self.split_head(key, self.group_num)
        value = self.split_head(value, self.group_num)

        ## 计算注意力分数
        attention_scores = torch.matmul(query, key.transpose(-1, -2)) /
torch.sqrt(torch.tensor(self.head_dim))

        if attention_mask != None:
            attention_scores += attention_mask * -1e-9

        ## 对注意力分数进行归一化
        attention_probs = torch.softmax(attention_scores, dim=-1)
```

```
        output = torch.matmul(attention_probs, value)

        output = output.transpose(-1, -2).contiguous().view(batch_size, -1,
self.head_dim * self.num_heads)

        output = self.o_linear(output)

        return output




    def split_head(self, x, group_num=None):

        batch_size,seq_len = x.size()[:2]

        if group_num == None:
            return x.view(batch_size, -1, self.num_heads,
self.head_dim).transpose(1,2)
        else:
            x = x.view(batch_size, -1, group_num, self.head_dim).transpose(1,2)
            x = x[:, :, None, :, :].expand(batch_size, group_num, self.num_heads
// group_num, seq_len, self.head_dim).reshape(batch_size, self.num_heads //
group_num * group_num, seq_len, self.head_dim)
            return x
```