【手搓大模型】从零手写GPT2 — Attention

Understand Attention in an Intuitive Way

前文我们讲述了 token Embedding和position Embedding，不过embedding从本质上依然是关于token自身的向量。一旦训练结束，embedding就是固定的weights了。我们需要引入一种新的机制，去关注token与token之间的依赖关系，也就是token所处的context。

如下示例：

> “He sat by the **bank** of the river.”
>
> "The cat that the dog chased was black."

bank到底表示是银行还是河岸，取决于旁边的river；black虽然离dog更近，但我们依然知道说的其实是cat。

有侧重点地关注句子中不同token的影响，即关注语义相关性，正是Attention机制的出发点。也就是说，Attention机制的本质，是让模型在每一步中自主判断哪些token更相关，并依次构建context。

在Attention机制的论文中，创新地引入了Query/Key/Value这3个tensor：

> An attention function can be described as mapping a query and a set of key-value pairs to an output,
>
> where the query, keys, values, and output are all vectors. The output is computed as a weighted sum
>
> of the values, where the weight assigned to each value is computed by a compatibility function of the
>
> query with the corresponding key.

其中：Query代表问题或关注点，Key代表信息的索引，Value代表信息的具体值。

直观上理解，Attention 模拟了“查询-检索-提取”的过程，以下粗浅的示例帮助理解：

1）假设你去图书馆找书（query是你想知道的主题），

2）图书馆有很多书架标签（keys）和书（values），

3）你先看每个书架标签和你想要的主题有多相关（compatibility），

4）然后你根据相关度决定从哪些书架取多少书（weighted as compatibility），

5）把拿到的书内容综合起来(weighted sum)，就是你最终的答案（output）。

Scaled Dot-Product Attention

Attention Definition

Attention的严格定义如下：

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

其中Q,K,V分别代表query、key、value对应的matrix；而dk代表matrix的维度，用于做scaling。

定义很简洁，其实计算过程也特别清晰简单。

以下用5个token，嵌入维度是4，为例：

代码块

```python
import torch
import torch.nn as nn

torch.manual_seed(123)

tokens = ["Once", "upon", "a", "time", "there"]
token_to_idx = {token: idx for idx, token in enumerate(tokens)}
embedding_dim = 4

embedding_layer = nn.Embedding(num_embeddings=len(tokens), embedding_dim=embedding_dim)

input_indices = torch.tensor([token_to_idx[token] for token in tokens])  # [0,1,2,3,4]
X = embedding_layer(input_indices)
print("shape of input X:", X.shape)
print(X)
```

> shape of input X: torch.Size([5, 4])
>
> tensor([[0.3374, -0.1778, -0.3035, -0.5880],
>
> [1.5810, 1.3010, 1.2753, -0.2010],
>
> [-0.1606, -0.4015, 0.6957, -1.8061],
>
> [-1.1589, 0.3255, -0.6315, -2.8400],
>
> [-0.7849, -1.4096, -0.4076, 0.7953]], grad_fn=<EmbeddingBackward0>)

得到5*4的二维matrix，如下：

Dim 1	Dim 2	Dim 3	Dim 4	Token
0.3374	−0.1778	−0.3035	−0.5880	"Once"
1.5810	1.3010	1.2753	−0.2010	"upon"
−0.1606	−0.4015	0.6957	−1.8061	"a"
−1.1589	0.3255	−0.6315	−2.8400	"time"
−0.7849	−1.4096	−0.4076	0.7953	"there"

Q K V matrix

而依据次，创建出Q/K/V的matrix的方法特别简单，只需要指定维度，并随机初始化得到初始矩阵，并利用该矩阵对输入X做线性映射，如下：

代码块

```
torch.manual_seed(123)

W_Q = torch.nn.Parameter(torch.rand(embedding_dim, embedding_dim),
requires_grad=False)
W_K = torch.nn.Parameter(torch.rand(embedding_dim, embedding_dim),
requires_grad=False)
W_V = torch.nn.Parameter(torch.rand(embedding_dim, embedding_dim),
requires_grad=False)

print("shape of W_Q:", W_Q.shape)
print("W_Q:", W_Q)

Q = X @ W_Q
K = X @ W_K
V = X @ W_V

print("shape of Q:", Q.shape)
print("Q:", Q)
```

> shape of W_Q: torch.Size([4, 4])
>
> W_Q: Parameter containing:
>
> tensor([[0.2961, 0.5166, 0.2517, 0.6886],
>
> [0.0740, 0.8665, 0.1366, 0.1025],
>
> [0.1841, 0.7264, 0.3153, 0.6871],
>
> [0.0756, 0.1966, 0.3164, 0.4017]])
>
> shape of Q: torch.Size([5, 4])

> Q: tensor([[-0.0136, -0.3159, -0.2211, -0.2307],
>
> [0.7839, 2.8310, 0.9140, 2.0175],
>
> [-0.0858, -0.2806, -0.4474, -0.3992],
>
> [-0.6501, -1.3338, -1.3449, -2.3394],
>
> [-0.3515, -1.7666, -0.2669, -0.6454]], grad_fn=<MmBackward0>)

请注意X的维度是[5,4]，随机初始化得到的W_Q的维度是[4,4]，根据矩阵乘法，得到的Q的维度[5,4]。

这里使用了矩阵乘法（@等价与torch.matmul），相当于对原始的输入X做了线性投影。

值得注意的是，W_Q, W_K, W_V这3个都是可训练的参数。这就意味着其初始值并不重要，重要的是搭建出的空间自由度和信息的通路。

Similarity as Scores

依据上述公式， $\text{scores} = QK^T$ ，我们需要计算Q与K之间的点积，以计算二者之间的相关性或相似度。

代码块

```
scores = Q @ K.T
print("shape of scores:", scores.shape)
print("scores:", scores)
```

> shape of scores: torch.Size([5, 5])
>
> scores: tensor([[0.3101, -2.0474, 0.7024, 1.8280, 1.0647],
>
> [-2.5714, 17.4476, -5.5017, -14.6920, -9.3044],
>
> [0.6084, -2.9632, 1.4480, 3.1775, 1.4642],
>
> [2.8736, -14.6337, 6.4597, 14.7155, 7.4156],
>
> [0.9222, -8.1955, 1.8808, 5.9959, 4.5150]],
>
> grad_fn=<MmBackward0>)

上面例子中，Q与K的维度是[5,4]，对K做转置得到维度[4,5]，二者做点积，得到[5,5]的矩阵。

注：此处实际上是做的批量点积，即使用的是矩阵乘法。

Scaled Scores

根据上述公式， $\text{scaled\_scores} = \frac{QK^T}{\sqrt{d_k}}$ ，我们需要对得到的scores进行scaled，操作如下：

代码块

```
import math
```

```

attention_scores = scores / math.sqrt(embedding_dim)
print(attention_scores)
```

> tensor([[0.1551, -1.0237, 0.3512, 0.9140, 0.5323],
> [-1.2857, 8.7238, -2.7508, -7.3460, -4.6522],
> [0.3042, -1.4816, 0.7240, 1.5888, 0.7321],
> [1.4368, -7.3169, 3.2298, 7.3577, 3.7078],
> [0.4611, -4.0977, 0.9404, 2.9979, 2.2575]], grad_fn=<DivBackward0>)

那为什么要进行缩放，以及为什么要选择上面的值进行缩放呢？ 缩放主要是为了压缩score，避免后续softmax输出的分布太过极端，让梯度计算更流畅；之所以选择dk的平方根，应该有其数学统计意义。不过个人感觉还是经验与折中方案，除以其他值也是合理的，不必过分关注，本质上就只是一种数据正则化手段。

至此，我们就完成了Scaled Attention scores的计算。

Compute Attention Weights via Softmax

而从attention scores到可被使用的weights，还需要做进一步的归一化，即通过softmax操作：

$$\text{attention\_weights} = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

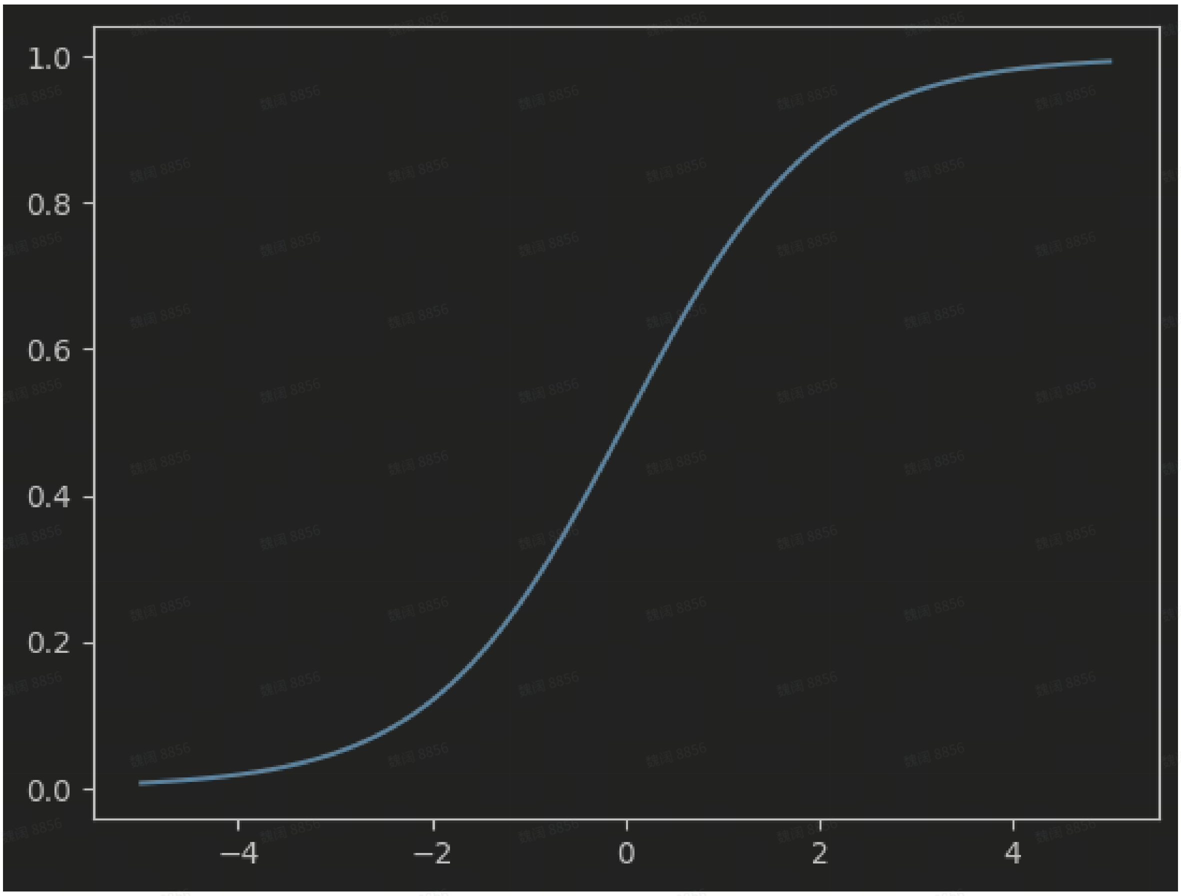
画图看一眼softmax函数，特别简单，如下：

代码块

```
import torch
import matplotlib.pyplot as plt

x = torch.linspace(-5, 5, 200)
scores = torch.stack([x, torch.zeros_like(x)], dim=1)
softmax_vals = torch.softmax(scores, dim=1)

plt.plot(x.numpy(), softmax_vals[:,0].numpy())
plt.show()
```



可见softmax把所有的输入都压缩到(0,1)之间，使之看起来更像概率值。

注：softmax本质是一种数据归一化，也可以替换成其他相似函数。

我们直接使用pytorch自带的softmax函数计算如下：

代码块

```
attention_weights = torch.softmax(attention_scores, dim=-1)
print("shape of attention_weights:", attention_weights.shape)
print(attention_weights)
```

> shape of attention_weights: torch.Size([5, 5])
>
> tensor([[1.6344e-01, 5.0283e-02, 1.9885e-01, 3.4910e-01, 2.3833e-01],
>
> [4.4966e-05, 9.9994e-01, 1.0389e-05, 1.0494e-07, 1.5519e-06],
>
> [1.2761e-01, 2.1395e-02, 1.9418e-01, 4.6106e-01, 1.9576e-01],
>
> [2.5676e-03, 4.0538e-07, 1.5426e-02, 9.5713e-01, 2.4878e-02],
>
> [4.6963e-02, 4.9191e-04, 7.5844e-02, 5.9361e-01, 2.8309e-01]],
>
> grad_fn=<SoftmaxBackward0>)

可见得到的weight均在(0,1)，很适合用来做加权计算。

Output as weighted sum

依据Attention定义，得到weights矩阵后，需要与Value矩阵相乘，得到最终的Attention ouput：

$$\text{output} = \text{attention\_weights} \cdot V$$

代码块

```python
# Final output of self-attention
output = attention_weights @ V
print("shape of output:", output.shape)
print(output)
```

> shape of output: torch.Size([5, 4])
>
> tensor([[-1.0221, -1.1318, -1.0966, -1.2475],
>
> [1.6613, 1.7716, 2.1347, 2.5049],
>
> [-1.3064, -1.3985, -1.3982, -1.5418],
>
> [-2.2928, -2.2490, -2.4211, -2.5138],
>
> [-1.6010, -1.6693, -1.7563, -1.9028]], grad_fn=<MmBackward0>)

注意维度的变化，[5,5] * [5,4]得到最终output的形状是[5,4]，这与输入X的形状刚好是一致的。也就是说，经过了Attention变化之后，输出的维度依然与输入相同。

至此，我们完成了Attention的完整计算。

Simple Self-Attention Code

理解了上述过程，我们可以使用pytorch非常方便地构建self-attention模块，如下：

代码块

```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class SimpleSelfAttention(nn.Module):
    def __init__(self, d_in, d_out):
        super().__init__()
        # (d_in, d_out)
        self.W_Q = nn.Linear(d_in, d_out, bias=False)
        self.W_K = nn.Linear(d_in, d_out, bias=False)
        self.W_V = nn.Linear(d_in, d_out, bias=False)

    def forward(self, x):
        # (seq_len, d_in) x (d_in, d_out) -> (seq_len, d_out)
        Q = self.W_Q(x)   # equal to: x @ W_Q.T
        K = self.W_K(x)
```

```python
        V = self.W_V(x)

        # (seq_len, d_out) x (d_out, seq_len) -> (seq_len, seq_len)
        scores = Q @ K.transpose(-2, -1) / K.shape[-1]**0.5
        # (seq_len, seq_len)
        weights = F.softmax(scores, dim=-1)
        # (seq_len, seq_len) x (seq_len, d_out) -> # (seq_len, d_out)
        context = weights @ V
        return context
```

代码块

```python
torch.manual_seed(123)
sa = SelfAttentionV2(4, 4)
output = sa(X)
print(output)
```

> tensor([[0.1318, -0.1000, -0.4239, -0.0858],
>
> [-0.0532, 0.2164, -0.8386, -0.1107],
>
> [0.2318, -0.2270, -0.4083, -0.0919],
>
> [0.4762, -0.5514, -0.2901, -0.0859],
>
> [0.0700, -0.0399, -0.3281, -0.0728]], grad_fn=<MmBackward0>)

请特别注意其中tensor维度的变化。

注：此处使用了nn.Linear构建线性层来初始化Q权重，也可以使用nn.Parameter(torch.rand(d_in, d_out))手动创建参数矩阵。不过二者的内部初始化方式略有不同。

Casual Attention: Mask future words

上述Attention weights的计算包含了整个context，但这与生成式大模型的训练过程并不一致，如：

> “He sat by the **bank** of the river.”

当模型正在尝试生成bank的时候，context中只能包含前面的单词，而不能包含后面的river等单词。因为如果在训练阶段，我们允许模型看到全部context，那么训练出来的模型泛化能力较差；当遇到真正的生成任务的时候，效果不佳。因此，我们需要把模型不应该看到的“未来词”挡住，以更好地提升模型能力。

在Embedding章节，我们已经知道大模型的训练是一个自回归过程，如下：

> Once --> upon
>
> Once upon --> a
>
> Once upon a --> time

> Once upon a time --> there
>
> Once upon a time there --> were

那其实遮挡未来词就变得非常简单，只需要把上述Attention中对角线以上的元素全部去掉即可。

如，我们可以借助如下的下三角矩阵，通过矩阵运算，轻松mask掉未来token。mask矩阵如下：

代码块

```python
import torch

context_size = attention_scores.shape[0]
# Lower triangular mask
mask = torch.tril(torch.ones(context_size, context_size))
print(mask)
mask = mask.masked_fill(mask == 0, float('-inf')).masked_fill(mask == 1, 0.0)
print(mask)
```

> tensor([[1., 0., 0., 0., 0.],
>
> [1., 1., 0., 0., 0.],
>
> [1., 1., 1., 0., 0.],
>
> [1., 1., 1., 1., 0.],
>
> [1., 1., 1., 1., 1.]])
>
> tensor([[0., -inf, -inf, -inf, -inf],
>
> [0., 0., -inf, -inf, -inf],
>
> [0., 0., 0., -inf, -inf],
>
> [0., 0., 0., 0., -inf],
>
> [0., 0., 0., 0., 0.]])

我们最终得到了下三角是0、上三角是负无穷的矩阵。

而mask的过程就是简单的矩阵加法，如下：

代码块

```python
print("original scores: \n", attention_scores)
# Apply mask to scores
masked_scores = attention_scores + mask
print("masked scores:\n", masked_scores)
```

> original scores:
>
> tensor([[0.1551, -1.0237, 0.3512, 0.9140, 0.5323],

> [-1.2857, 8.7238, -2.7508, -7.3460, -4.6522],
>
> [0.3042, -1.4816, 0.7240, 1.5888, 0.7321],
>
> [1.4368, -7.3169, 3.2298, 7.3577, 3.7078],
>
> [0.4611, -4.0977, 0.9404, 2.9979, 2.2575]], grad_fn=<DivBackward0>)
>
> masked scores:
>
> tensor([[0.1551, -inf, -inf, -inf, -inf],
>
> [-1.2857, 8.7238, -inf, -inf, -inf],
>
> [0.3042, -1.4816, 0.7240, -inf, -inf],
>
> [1.4368, -7.3169, 3.2298, 7.3577, -inf],
>
> [0.4611, -4.0977, 0.9404, 2.9979, 2.2575]], grad_fn=<AddBackward0>)

可见，我们只保留了Attention scores的下三角部分，上三角被填充为-inf。使用-inf是因为后续需要做softmax运算，而softmax(-inf)=0，对weights的计算就是零贡献。

Dropout

另外，为了提升模型泛化能力，经常使用的一种技术是随机丢弃，即dropout。我们用pytorch代码简单示例dropout如下：

代码块

```
import torch

torch.manual_seed(123)

# Create a dropout layer with 20% dropout rate
dropout = torch.nn.Dropout(0.2)
dropout.train()  # Explicitly set to training mode to enable dropout

example = torch.ones(5, 5)
print("Input tensor:\n",example)

# Apply dropout to the input tensor
output = dropout(example)
print("tensor after Dropout:\n",output)
print(f"Number of zeros in output: {(output == 0).sum().item()}")
print(f"Output mean value (should be ~1.0 due to scaling): {output.mean().item():.4f}")
```

> Input tensor:
>
> tensor([[1., 1., 1., 1., 1.],

> [1., 1., 1., 1., 1.],
>
> [1., 1., 1., 1., 1.],
>
> [1., 1., 1., 1., 1.],
>
> [1., 1., 1., 1., 1.]])
>
> tensor after Dropout:
>
> tensor([[1.2500, 1.2500, 1.2500, 1.2500, 1.2500],
>
> [1.2500, 1.2500, 1.2500, 0.0000, 1.2500],
>
> [0.0000, 1.2500, 1.2500, 1.2500, 1.2500],
>
> [1.2500, 1.2500, 1.2500, 1.2500, 1.2500],
>
> [1.2500, 1.2500, 1.2500, 1.2500, 1.2500]])
>
> Number of zeros in output: 2
>
> Output mean value (should be ~1.0 due to scaling): 1.1500

可见，在5x5的全1矩阵中，部分值被置为了0，且剩余值变成了1.25。这是因为pytorch在做dropout的时候，为了保持整体的均值不变，按照scale=1/(1-drop_rate)做了缩放。

不过看起来上述结果并不太符合预期，这是因为示例用的矩阵维度太小了，别忘了统计概率只对大数据生效。只需要把上面的维度改为500x500，就可以看到mean依然非常接近1。在gpt2的真实环境中，如前所述，这一步weights的维度是seq_len x seq_len，也就1024 x 1024，其实是非常巨大的。

在Attention机制中，dropout是作用在softmax后得到的weights上的，代码如下：

代码块

```
weights = F.softmax(masked_scores, dim=-1)
print("weights after mask: \n", weights)
torch.manual_seed(123)
output = dropout(weights)
print("weights after Dropout: \n", output)
```

> weights after mask:
>
> tensor([[1.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00],
>
> [4.4967e-05, 9.9996e-01, 0.0000e+00, 0.0000e+00, 0.0000e+00],
>
> [3.7185e-01, 6.2345e-02, 5.6581e-01, 0.0000e+00, 0.0000e+00],
>
> [2.6332e-03, 4.1573e-07, 1.5819e-02, 9.8155e-01, 0.0000e+00],
>
> [4.6963e-02, 4.9191e-04, 7.5844e-02, 5.9361e-01, 2.8309e-01]],
>
> grad_fn=<SoftmaxBackward0>)

> weights after Dropout:
>
> tensor([[1.2500e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00],
>
> [5.6209e-05, 1.2499e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00],
>
> [0.0000e+00, 7.7931e-02, 7.0726e-01, 0.0000e+00, 0.0000e+00],
>
> [3.2914e-03, 5.1966e-07, 1.9774e-02, 1.2269e+00, 0.0000e+00],
>
> [5.8704e-02, 6.1489e-04, 9.4804e-02, 7.4201e-01, 3.5387e-01]],
>
> grad_fn=<MulBackward0>)

可见，softmax作用于masked scores，得到的是仅有下三角的weights；weights经过dropout时候，有20%的值被置为0，剩余值被sacle到1.25倍。

Casual Self-Attention Code

我们在前面SimpleSelfAttention的基础上，增加mask和dropout，得到完整代码如下：

代码块

```
import torch
import torch.nn as nn

class CausalAttention(nn.Module):
    """
    Implements single-head causal self-attention with optional dropout.
    """
    def __init__(self, d_in, d_out, context_length, dropout=0.0, qkv_bias=False):
        super().__init__()
        # (d_in, d_out)
        self.W_Q = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_K = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_V = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.dropout = nn.Dropout(dropout)

        # Create a fixed causal mask (upper triangular) [1 means "mask"]
        mask = torch.triu(torch.ones(context_length, context_length), diagonal=1)
        self.register_buffer("mask", mask.bool())

    def forward(self, x):
        # x: shape (batch_size, seq_len, d_in)
        batch_size, seq_len, _ = x.size()
        Q = self.W_Q(x)
        K = self.W_K(x)
        V = self.W_V(x)
```

```
        # Compute attention scores
        scores = Q @ K.transpose(-2, -1) / (d_out ** 0.5)  # (batch_size, seq_len, seq_len)

        # Apply causal mask
        scores = scores.masked_fill(self.mask[:seq_len, :seq_len], -torch.inf)

        # Compute softmax weights and apply dropout
        weights = torch.softmax(scores, dim=-1)
        weights = self.dropout(weights)

        # Compute output
        output = weights @ V  # (batch_size, seq_len, d_out)
        return output
```

在这段代码中，为了更与真实环境贴合，我们为输入X增加了一个batch的维度，其维度变为：(batch_size, seq_len, d_in)，而最终得到的output的维度也相应变为(batch_size, seq_len, d_out)。

我们通过生成2批次、最大长度为5、维度是4的随机矩阵，来模拟通过上述CausalAttention来计算最终的context vector，代码如下：

代码块

```
torch.manual_seed(123)

batch = torch.randn(2, 5, 4)  # (batch_size=2, seq_len=5, d_in=4)
d_in = 4
d_out = 4
context_length = batch.size(1)

ca = CausalAttention(d_in, d_out, context_length, dropout=0.0)
context_vecs = ca(batch)

print("context_vecs.shape:", context_vecs.shape)
print("context_vecs:\n", context_vecs)
```

> context_vecs.shape: torch.Size([2, 5, 4])
>
> context_vecs:
>
> tensor([[[-0.0487, -0.0112, 0.0449, 0.3506],
>
> [0.0439, 0.1278, 0.1848, 0.1733],
>
> [-0.2467, -0.1078, 0.2722, 0.5128],
>
> [-0.1638, 0.0053, 0.3753, 0.3111],

[0.0264, 0.1455, 0.3622, 0.0182]],

[[0.0960, 0.4257, 1.7419, 0.2045],
[-0.0967, 0.2774, 1.1946, 0.5023],
[0.1017, 0.2037, 0.4849, 0.1862],
[-0.0775, 0.1062, 0.3737, 0.3387],
[-0.1181, -0.0113, 0.1070, 0.2743]]], grad_fn=<UnsafeViewBackward0>)

请特别注意，最终生成的context vector的维度一定是与输入vector的维度是完全一致的。

至此，我们完成了Attention论文中单头注意力的完整计算。

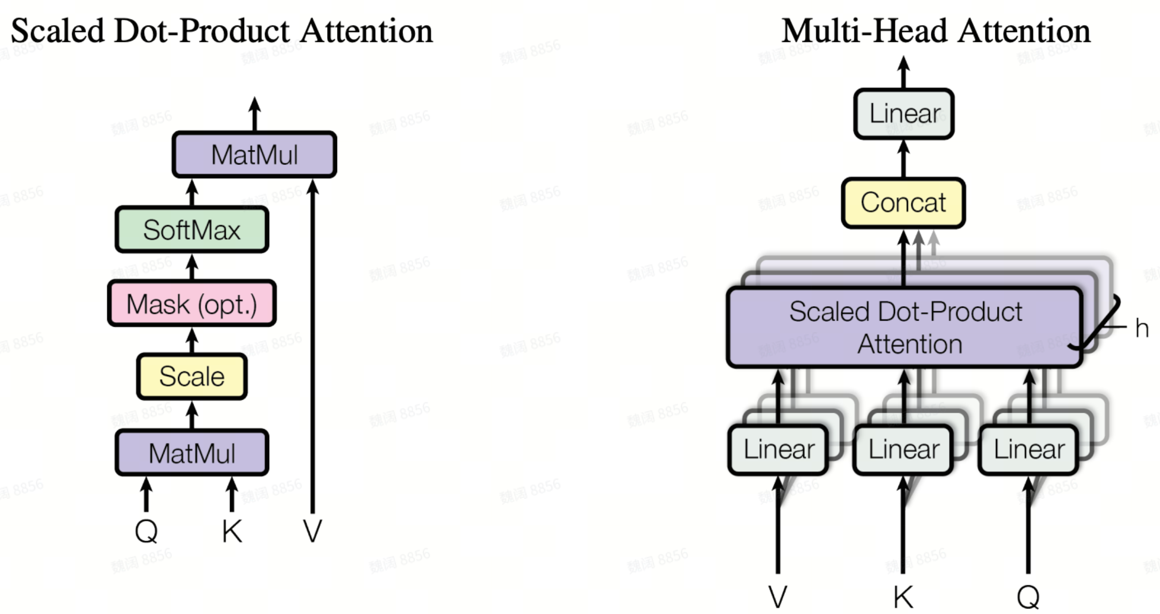


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

Multi-Head Attention

前面我们完成了单头注意力的计算，为了进一步提升模型的表达能力，又进一步引入多头注意力计算，如上图的Attention论文图片所示。

比如在

> “The cat that the dog chased was black.”

这个例子中，利用多个注意力头，可以分别关注不同的语义结构：

> "cat" <------ "was"（Head 1 强 attention）Head 1：关注主语-谓语（句子主干）
>
> "that", "dog", "chased" ----> "cat"（Head 2 强 attention）Head 2：关注定语从句的修饰结构

> "dog" ----> "chased"（Head 3 强 attention）Head 3：关注宾语结构
>
> "cat" <---- "black"（Head 4 attention）Head 4：关注形容词修饰关系

Concat heads code

多头的最直接实现，是直接把上述单头重复多次，然后堆叠在一起，代码如下：

代码块

```
class MultiHeadAttentionWrapper(nn.Module):
    """
    Implements multi-head self-attention by stacking multiple heads.
    """

    def __init__(self, d_in, d_out, context_length, dropout, num_heads, qkv_bias=False):
        super().__init__()
        assert d_out % num_heads == 0, "d_out must be divisible by num_heads"
        self.head_dim = d_out // num_heads
        self.heads = nn.ModuleList(
            [CausalAttention(d_in, self.head_dim, context_length, dropout, qkv_bias) for _ in range(num_heads)])

    def forward(self, x):
        output = torch.cat([head(x) for head in self.heads], dim=-1)
        return output
```

代码块

```
torch.manual_seed(123)

batch = torch.randn(2, 5, 6)   # (batch_size=2, seq_len=5, d_in=6)
d_in = 6
d_out = 6
context_length = batch.size(1)

mha = MultiHeadAttentionWrapper(d_in, d_out, context_length, dropout=0.0,num_heads=2)
context_vecs = mha(batch)

print("context_vecs.shape:", context_vecs.shape)
print("context_vecs:\n", context_vecs)
```

> context_vecs.shape: torch.Size([2, 5, 6])
>
> context_vecs:

> tensor([[[-0.0067, -0.0370, 0.2712, -0.5243, -0.0242, -0.0438],
>
> [-0.1782, 0.0173, -0.0166, -0.2391, -0.0284, 0.2177],
>
> [-0.1541, 0.2878, -0.2018, 0.2535, 0.0242, 0.3002],
>
> [-0.2817, 0.5219, -0.0699, 0.5508, -0.2767, 0.3709],
>
> [-0.0355, -0.1721, 0.0981, 0.2389, -0.1460, 0.1938]],
>
> [[0.7943, -1.9382, 0.2171, -1.6710, 0.7970, -1.3094],
>
> [0.2519, -1.1446, 0.2991, -1.5203, 0.3135, -0.9541],
>
> [0.1920, -0.8646, 0.3794, -0.9135, 0.0203, -0.5454],
>
> [0.2565, -0.8320, 0.1292, -0.9259, 0.2156, -0.4762],
>
> [0.1519, -0.5043, 0.1079, -0.3281, 0.1523, -0.1446]]],
>
> grad_fn=<CatBackward0>)

在上面代码中，我们手动模拟了2个head，并通过d_out // num_heads调整了单个head的维度。

Weight split code

但其实上面代码并不是真正的MHA(Multi-Head-Attention)的实现，堆叠矩阵的方式效率较低，更好的方式是先用大矩阵做一次大投影，然后再拆分，相当于weight splits。

代码如下：

代码块

```python
import torch
import torch.nn as nn

class MultiHeadAttention(nn.Module):
    """
    Implements multi-head attention by splitting the attention matrix into multiple heads.
    """

    def __init__(self, d_in, d_out, context_length, dropout, num_heads, qkv_bias=False):
        super().__init__()
        assert d_out % num_heads == 0, "d_out must be divisible by num_heads"
        self.num_heads = num_heads
        self.head_dim = d_out // num_heads

        self.W_Q = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_K = nn.Linear(d_in, d_out, bias=qkv_bias)
```

```python
        self.W_V = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_O = nn.Linear(d_out, d_out)
        self.dropout = nn.Dropout(dropout)

        mask = torch.triu(torch.ones(context_length, context_length), diagonal=1)
        self.register_buffer("mask", mask.bool())

    def forward(self, x):
        # shape (batch_size, seq_len, d_in)
        batch_size, seq_len, _ = x.size()

        # Split Q, K, V into multiple heads
        # (batch_size, seq_len, d_in) -> (batch_size, seq_len, d_out) ->
        # -> (batch_size, seq_len, num_heads, head_dim) -> (batch_size, num_heads, seq_len, head_dim)
        Q = self.W_Q(x).view(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1, 2)
        K = self.W_K(x).view(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1, 2)
        V = self.W_V(x).view(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1, 2)

        # Compute attention scores
        scores = Q @ K.transpose(-2, -1) / (d_out ** 0.5)  # (batch_size, num_heads, seq_len, seq_len)

        # Apply causal mask
        scores = scores.masked_fill(self.mask[:seq_len, :seq_len], -torch.inf)

        # Compute softmax weights and apply dropout
        weights = torch.softmax(scores, dim=-1)
        weights = self.dropout(weights)

        # Compute output
        output = weights @ V  # (batch_size, num_heads, seq_len, head_dim)
        # Concatenate heads and project to output dimension
        # (batch_size, num_heads, seq_len, head_dim) -> (batch_size, seq_len, num_heads, head_dim)
        # ->   (batch_size, seq_len, d_out)
        output = output.transpose(1, 2).contiguous().view(batch_size, seq_len, -1)
        # Should be helpful, but not strictly necessary.
        output = self.W_O(output)
        return output
```

代码块

```
torch.manual_seed(123)

batch = torch.randn(2, 5, 6)  # (batch_size=2, seq_len=5, d_in=6)
d_in = 6
d_out = 6
context_length = batch.size(1)

mha = MultiHeadAttention(d_in, d_out, context_length, dropout=0.0,num_heads=2)
context_vecs = mha(batch)

print("context_vecs.shape:", context_vecs.shape)
print("context_vecs:\n", context_vecs)
```

> context_vecs.shape: torch.Size([2, 5, 6])
>
> context_vecs:
>
> tensor([[[-0.5829, -0.5644, 0.1930, -0.1541, 0.2518, -0.2252],
>
> [-0.2962, -0.2681, 0.1179, 0.1136, 0.0953, -0.4015],
>
> [-0.2039, -0.0745, 0.1557, -0.0494, 0.1125, -0.5282],
>
> [-0.2540, 0.1181, 0.2729, -0.1182, 0.0321, -0.5292],
>
> [-0.2007, 0.0280, 0.1645, -0.0798, 0.1264, -0.5020]],
>
> [[-0.2307, -1.7354, -0.4065, 0.3778, 0.9090, -0.1498],
>
> [-0.5355, -1.2480, -0.0049, 0.1522, 0.5635, -0.0269],
>
> [-0.4674, -0.8466, 0.0176, 0.1337, 0.4053, -0.2230],
>
> [-0.3683, -0.6768, 0.0088, 0.0933, 0.3034, -0.3600],
>
> [-0.2545, -0.5944, -0.0236, 0.0762, 0.3629, -0.3780]]],
>
> grad_fn=<ViewBackward0>)

相比之前：

1） 使用了统一的大矩阵W_Q做投影，然后通过view操作split到多个head中。

2）对output额外做了一层线性映射，以进一步融合多头。不过这一步并非严格必须的。

在操作过程中，我们应该特别留意tensor维度的变化。只要看懂tensor维度的变化，就基本搞清楚了整个计算的逻辑。

至此，我们已经完成了MHA的代码实现，也是GPT2的Transformer架构中最核心的实现。