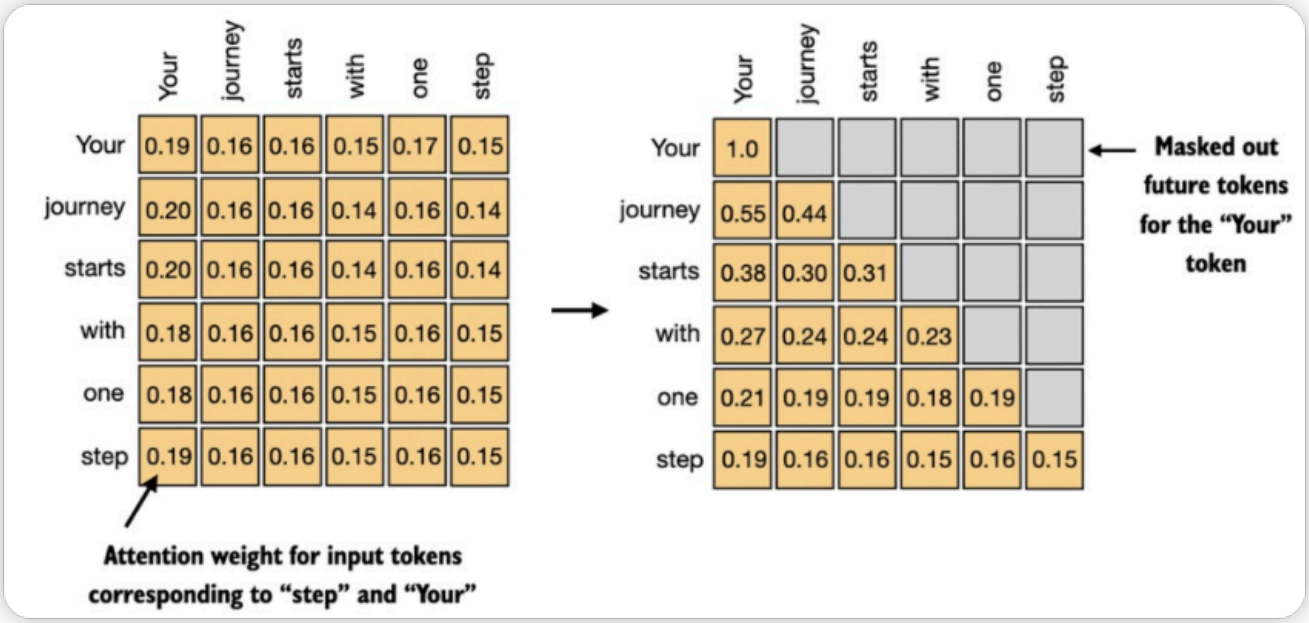
只考虑序列中当前Token或之前出现的Token。



> 自然想到引入遮蔽矩阵（下三角矩阵）

步骤

1. 前面与3-2得到注意力权重步骤相同。
2. 对每个当前token后的注意力权重进行遮蔽，**两种遮蔽方式**：
    1. 3-3因果注意力 > 遮蔽方式一
    2. 3-3因果注意力 > 遮蔽方式二
3. 计算上下文向量：通过注意力权重和值 values 向量，与3-2相同
    1. `context_vec = attn_weights @ values`

遮蔽方式一

1. `attn_weights * mask_simple` 与遮蔽矩阵（下三角矩阵）做乘法

2. 再归一化

```
# 引入遮蔽矩阵（下三角矩阵）:
context_length = attn_weights.shape[0]
mask_simple = torch.tril(torch.ones(context_length,
context_length))

# 将遮蔽与注意力权重相乘，将对角线以上的值归零:
masked_simple = attn_weights * mask_simple

# 再归一化:
row_sums = masked_simple.sum(dim=1, keepdim=True) # 求每一行的和
masked_simple_norm = masked_simple / row_sums
```

伪 - 信息泄漏问题

> 可能出现打算遮蔽的Token仍影响当前Token，因它们的值时softmax函数计算的一部分。

但是softmax的数学优雅之处在于，尽管在最初的计算中分母包含了所有位置，但在遮蔽和重新归一化之后，被遮蔽的位置的*影响被消除了*

遮蔽方式二

利用softmax函数特性：负无穷趋近于0。

```
# mask（上三角矩阵），主对角线为0
mask = torch.triu(torch.ones(context_length, context_length),
diagonal=1)
# 填充attn_scores张量中的上三角部分为负无穷
masked = attn_scores.masked_fill(mask.bool(), -torch.inf)
```

```
attn_weights = torch.softmax(masked / keys.shape[-1]**0.5, dim=1)
```

引入Dropout防止过拟合

[Layer > Dropout 层]

```
# 例子:
torch.manual_seed(123)

# 为了补偿活跃元素的减少，矩阵中剩余元素的值被放大了 1/0.5 = 2 倍
dropout = torch.nn.Dropout(0.5) # 丢弃率50%
attn_weights = dropout(attn_weights)
```

实现 Casual Attention 类

```
import torch.nn as nn

class CausalAttention(nn.Module):

    def __init__(self, d_in, d_out, context_length, dropout,
qkv_bias=False):
        super().__init__()
        self.d_out = d_out
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.dropout = nn.Dropout(dropout)
        # register_buffer 不需要手动确保这些张量与模型参数在同一设备上，从
而避免设备不匹配错误
        self.register_buffer('mask',
torch.triu(torch.ones(context_length, context_length), diagonal=1))

    def forward(self, x):
        b, num_tokens, d_in = x.shape
```

```
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)
        # 将keys的第二维和第三维进行交换，下面有说明！！！
        attn_scores = queries @ keys.transpose(1, 2)
        attn_scores.masked_fill_(
            self.mask.bool()[:num_tokens, :num_tokens], -torch.inf)
        attn_weights = torch.softmax(attn_scores / keys.shape[-1]
** 0.5, dim=-1)
        attn_weights = self.dropout(attn_weights)

        context_vec = attn_weights @ values
        return context_vec
```

keys.transpose(1, 2)说明

进行第二维和第三维交换：

变量	维度
inputs	`([b, num_tokens, d_in])`
W_ke, W_query, W_value	`([d_in, d_out])`
keys, queries, values	`([b, num_tokens, d_out])`
attn_scores, attn_weights	`([b, num_tokens, num_tokens])`
context_vec	`([b, num_tokens, d_out])`

- queries 为 `([b, num_tokens, d_out])`
- keys为 `([b, num_tokens, d_out])` 第二维和第三维交换后 `([b, d_out, num_tokens])` 然后才对齐执行矩阵乘法，得到attn_scores `([b, num_tokens, num_tokens])`