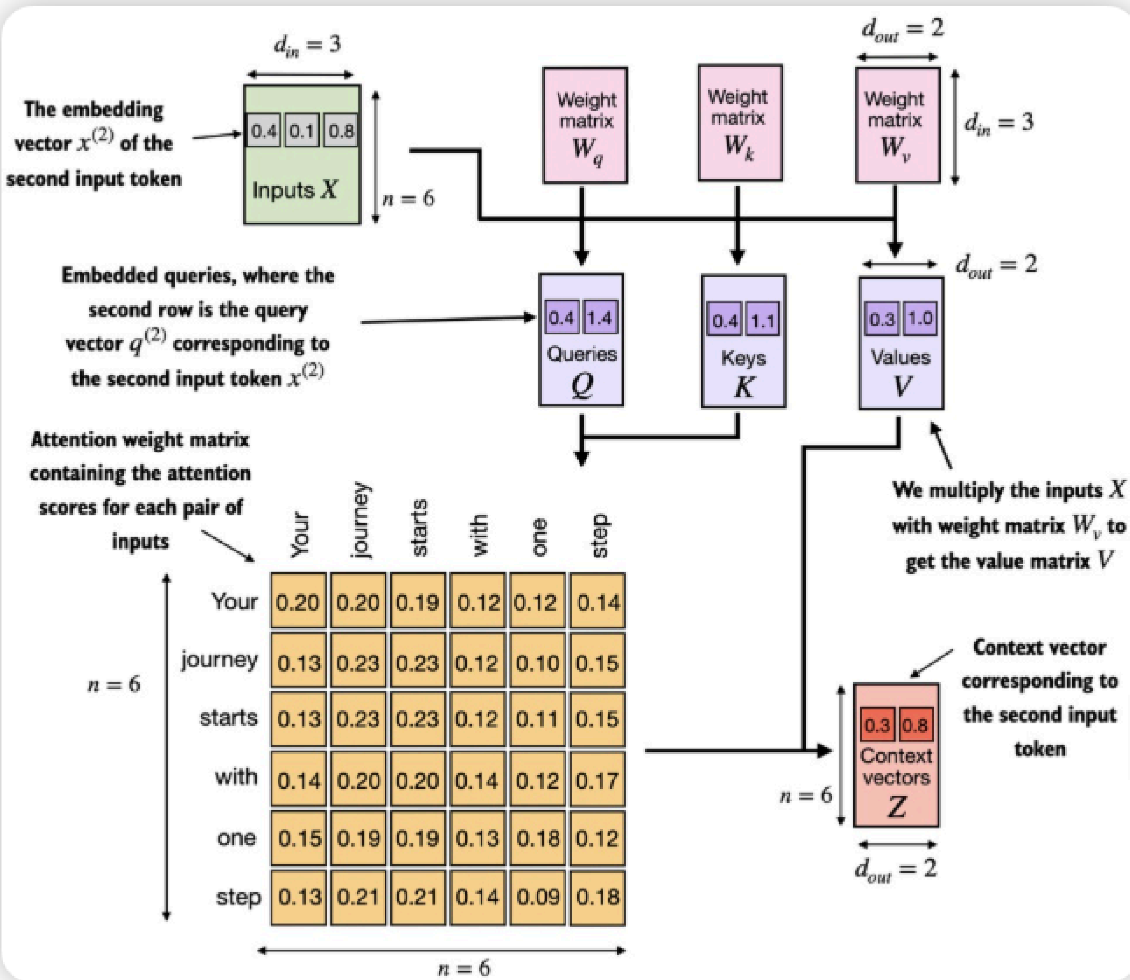
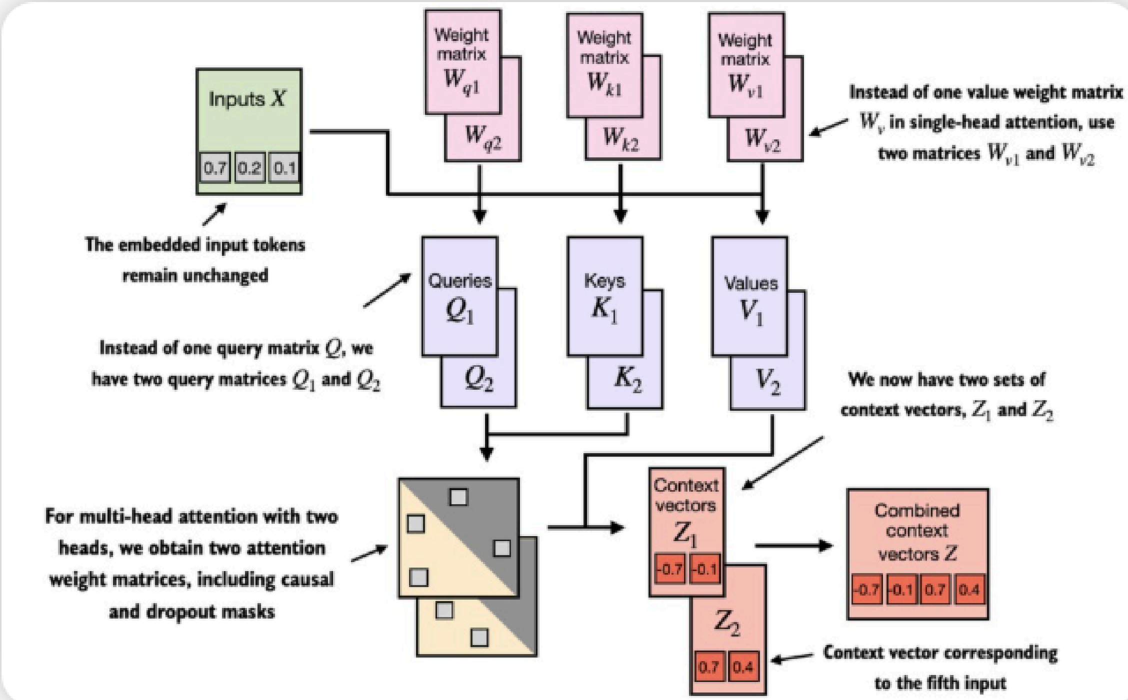
- 多头：将注意力机制分为多个“头”，每个头独立运作。单个因果注意力模块可以被视为单头注意力，其中只有一组注意力权重顺序处理输入。

⚠️ 核心：多组查询 W_q，键 W_k，值 W_v 权重矩阵。

单头：



多头：



Casual Attention 实现

顺序处理

- 缺点：需在forward方法中 `[head(x) for head in self.heads]` 顺序处理。

```
class MultiHeadAttentionWrapper(nn.Module):
    def __init__(self, d_in, d_out, context_length,
                 dropout, num_heads, qkv_bias=False):
        super().__init__()
        self.heads = nn.ModuleList(
            [CausalAttention(d_in, d_out, context_length, dropout,
qkv_bias)
             for _ in range(num_heads)]
        )

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

并行处理

矩阵乘法代替for循环：

> 关键操作是将 d_out 维度分割为 num_heads 和 head_dim，其中 head_dim = d_out / num_heads。这种分割随后通过 .view 方法实现：将维度为 (b, num_tokens, d_out) 的张量重塑为维度 (b, num_tokens, num_heads, head_dim)

- 优点：只需要一次矩阵乘法就可以计算出键。

> 带维度注释

```
import torch
import torch.nn as nn

class MultiHeadAttention(nn.Module):

    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" # A
        self.d_out = d_out
        self.num_heads = num_heads # A
        self.head_dim = d_out // num_heads # A

        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.out_proj = nn.Linear(d_out, d_out) # B
        self.dropout = nn.Dropout(dropout)
        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[(b, num_tokens, d_out)] 重塑为 [(b, num_tokens,
num_heads, num_dim)]
        keys = keys.view(b, num_tokens, self.num_heads,
self.head_dim)
        queries = queries.view(b, num_tokens, self.num_heads,
self.head_dim)
        values = values.view(b, num_tokens, self.num_heads,
self.head_dim)

        # 将[(b, num_tokens, num_heads, num_dim)] 转置为 [(b,
num_heads, num_tokens, num_dim)]
        keys = keys.transpose(1, 2)
        queries = queries.transpose(1, 2)
        values = values.transpose(1, 2)

        # [(b, num_heads, num_tokens, num_dim)] @ [(b, num_heads,
num_dim, num_tokens)]
        # => attn_scores: [(b, num_heads, num_tokens, num_tokens)]
        attn_scores = queries @ keys.transpose(2, 3)
        mask_bool = self.mask.bool()[:num_tokens, :num_tokens]
        attn_scores.masked_fill_(mask_bool, -torch.inf)
        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1] ** 0.5, dim=-1)
        attn_weights = self.dropout(attn_weights)

        # [(b, num_heads, num_tokens, num_tokens)] @ [(b,
num_heads, num_tokens, num_dim)]
        # =>[(b, num_heads, num_tokens, num_dim)] 再转置
        # => context_vec: [(b, num_tokens, num_heads, num_dim)]
        context_vec = (attn_weights @ values).transpose(1, 2)
        # [(b, num_tokens, num_heads, num_dim)] 重塑为 [(b,
num_tokens, d_out)]
        context_vec = context_vec.contiguous().view(b, num_tokens,
self.d_out) # F
```

```
        context_vec = self.out_proj(context_vec) # F
        return context_vec
```