# Artificial Intelligence Experimental Manual

## 人工智能实验课程手册（下册）

作者：Yunlong Yu

组织：浙江大学信电学院

时间：July 10, 2023

浙江大学

# 目录

# 第 1 章 注意力机制

## 1.1 注意力机制

### 1.1.1 查询、键和值

自主性的与非自主性的注意力提示解释了人类的注意力的方式，下面来看看如何通过这两种注意力提示，用神经网络来设计注意力机制的框架，

首先，考虑一个相对简单的状况，即只使用非自主性提示。要想将选择偏向于感官输入，则可以简单地使用参数化的全连接层，甚至是非参数化的最大汇聚层或平均汇聚层。

因此，“是否包含自主性提示”将注意力机制与全连接层或汇聚层区别开来。在注意力机制的背景下，自主性提示被称为查询（query）。给定任何查询，注意力机制通过注意力汇聚（attention pooling）将选择引导至感官输入（sensory inputs，例如中间特征表示）。在注意力机制中，这些感官输入被称为值（value）。更通俗的解释，每个值都与一个键（key）配对，这可以想象为感官输入的非自主提示。如图1.1所示，可以通过设计注意力汇聚的方式，便于给定的查询（自主性提示）与键（非自主性提示）进行匹配，这将引导得出最匹配的值（感官输入）。

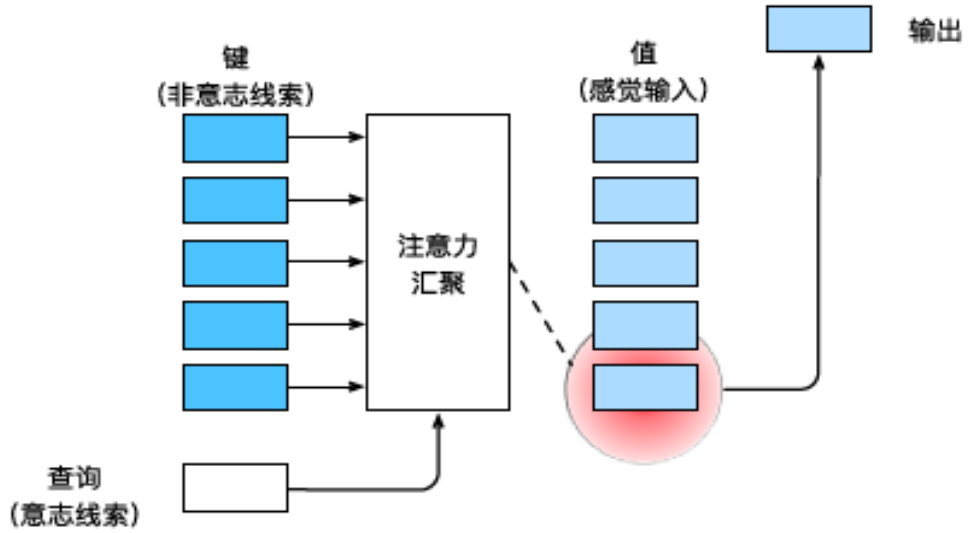



**图 1.1:** 注意力机制通过注意力汇聚将查询（自主性提示）和键（非自主性提示）结合在一起，实现对值（感官输入）的选择倾向。

### 1.1.2 注意力的可视化

平均汇聚层可以被视为输入的加权平均值，其中各输入的权重是一样的。实际上，注意力汇聚得到的是加权平均的总和值，其中权重是在给定的查询和不同的键之间计算得出的。

```
import torch
from d2l import torch as d2l
```

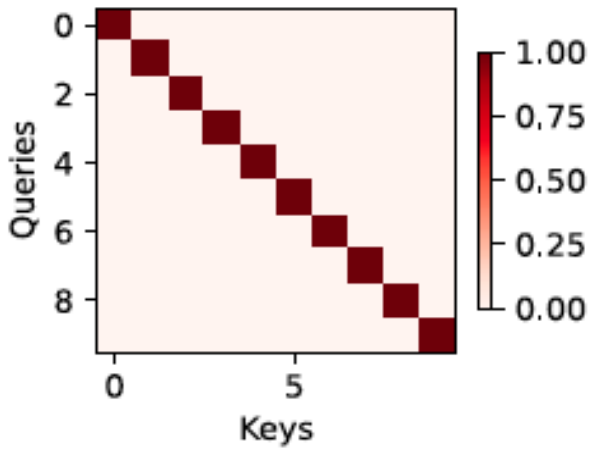
为了可视化注意力权重，需要定义一个 show_heatmaps 函数。其输入 matrices 的形状是（要显示的行数，要显示的列数，查询的数目，键的数目）。

```
#@save
def show_heatmaps(matrices, xlabel, ylabel, titles=None, figsize=(2.5, 2.5),
cmap='Reds'):
    """显示矩阵热图"""
```

```
    d2l.use_svg_display()
    num_rows, num_cols = matrices.shape[0], matrices.shape[1]
    fig, axes = d2l.plt.subplots(num_rows, num_cols, figsize=figsize,
    sharex=True, sharey=True, squeeze=False)
    for i, (row_axes, row_matrices) in enumerate(zip(axes, matrices)):
        for j, (ax, matrix) in enumerate(zip(row_axes, row_matrices)):
            pcm = ax.imshow(matrix.detach().numpy(), cmap=cmap)
            if i == num_rows - 1:
                ax.set_xlabel(xlabel)
            if j == 0:
                ax.set_ylabel(ylabel)
            if titles:
                ax.set_title(titles[j])
    fig.colorbar(pcm, ax=axes, shrink=0.6);
```

下面使用一个简单的例子进行演示。在本例子中，仅当查询和键相同时，注意力权重为 1，否则为 0。

```
attention_weights = torch.eye(10).reshape((1, 1, 10, 10))
show_heatmaps(attention_weights, xlabel='Keys', ylabel='Queries')
```




## 1.2 多头注意力

在实践中，当给定相同的查询、键和值的集合时，我们希望模型可以基于相同的注意力机制学习到不同的行为，然后将不同的行为作为知识组合起来，捕获序列内各种范围的依赖关系（例如，短距离依赖和□距离依赖关系）。因此，允许注意力机制组合使用查询、键和值的不同子空间表示（representation subspaces）可能是有益的。

为此，与其只使用单独一个注意力汇聚，我们可以用独立学习得到的 h 组不同的线性投影（linear projections）来变换查询、键和值。然后，这 h 组变换后的查询、键和值将并行地送到注意力汇聚中。最后，将这 h 个注意力汇聚的输出拼接在一起，并且通过另一个可以学习的线性投影进行变换，以产生最终输出。这种设计被称为多头注意力（multihead attention）。对于 h 个注意力汇聚输出，每一个注意力汇聚都被称作一个头（head）。图1.2展示了使用全连接层来实现可学习的线性变换的多头注意力。

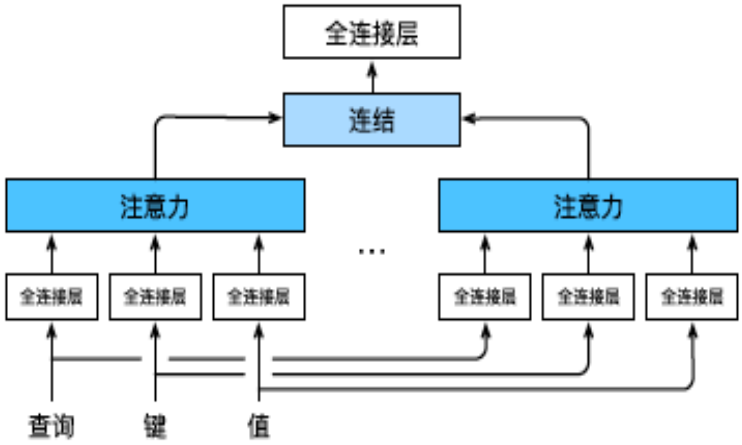



图 1.2: 多头注意力：多个头连结然后线性变换。

### 1.2.1 模型

在实现多头注意力之前，让我们用数学语言将这个模型形式化地描述出来。给定查询 $q \in \mathbb{R}^{d_q}$、键 $k \in \mathbb{R}^{d_k}$ 和值 $v \in \mathbb{R}^{d_v}$，每个注意力头 $h_i (i = 1, ..., h)$ 的计算方法为：

$$h_i = f(W_i^{(q)} q, W_i^{(k)} k, W_i^{(v)} v) \in \mathbb{R}^{p_v}, \tag{1.1}$$

其中，可学习的参数包括 $W_i^{(q)} \in \mathbb{R}^{p_q \times d_q}$、$W_i^{(k)} \in \mathbb{R}^{p_k \times d_k}$、$W_i^{(v)} \in \mathbb{R}^{p_v \times d_v}$，以及代表注意力汇聚的函数 $f$。

```
import math
import torch
from torch import nn
from d2l import torch as d2l
```

### 1.2.2 实现

在实现过程中通常选择缩放点积注意力作为每一个注意力头。为了避免计算代价和参数代价的大幅增长，我们设定 $p_q = p_k = p_v = p_o/h$。值得注意的是，如果将查询、键和值的线性变换的输出数量设置为 $p_q h = p_k h = p_v h = p_o$，则可以并行计算 $h$ 个头。在下面的实现中，$p_o$ 是通过参数 num_hiddens 指定的。

```
#@save
class MultiHeadAttention(nn.Module):
    """多头注意力"""
    def __init__(self, key_size, query_size, value_size, num_hiddens,
    num_heads, dropout, bias=False, **kwargs):
        super(MultiHeadAttention, self).__init__(**kwargs)
        self.num_heads = num_heads
        self.attention = d2l.DotProductAttention(dropout)
        self.W_q = nn.Linear(query_size, num_hiddens, bias=bias)
        self.W_k = nn.Linear(key_size, num_hiddens, bias=bias)
        self.W_v = nn.Linear(value_size, num_hiddens, bias=bias)
        self.W_o = nn.Linear(num_hiddens, num_hiddens, bias=bias)
    def forward(self, queries, keys, values, valid_lens):
        # queries，keys，values的形状：
        # (batch_size，查询或者“键 - 值”对的个数，num_hiddens)
```

```
        # valid_lens 的形状:
        # (batch_size, )或(batch_size, 查询的个数)
        # 经过变换后，输出的queries，keys，values 的形状:
        # (batch_size*num_heads，查询或者“键 - 值”对的个数，
        # num_hiddens/num_heads)
        queries = transpose_qkv(self.W_q(queries), self.num_heads)
        keys = transpose_qkv(self.W_k(keys), self.num_heads)
        values = transpose_qkv(self.W_v(values), self.num_heads)

        if valid_lens is not None:
            # 在轴0，将第一项（标量或者矢量）复制num_heads次，
            # 然后如此复制第二项，然后诸如此类。
            valid_lens = torch.repeat_interleave(valid_lens, repeats=self.num_heads, dim=0)
        # output的形状:(batch_size*num_heads，查询的个数，
        # num_hiddens/num_heads)
        output = self.attention(queries, keys, values, valid_lens)
        # output_concat的形状:(batch_size，查询的个数，num_hiddens)
        output_concat = transpose_output(output, self.num_heads)
        return self.W_o(output_concat)
```

为了能够使多个头并行计算，上面的 MultiHeadAttention 类将使用下面定义的两个转置函数。具体来说，transpose_output 函数反转了 transpose_qkv 函数的操作。

```
#@save
def transpose_qkv(X, num_heads):
    """为了多注意力头的并行计算而变换形状"""
    # 输入X的形状:(batch_size，查询或者“键 - 值”对的个数，num_hiddens)
    # 输出X的形状:(batch_size，查询或者“键 - 值”对的个数，num_heads，
    # num_hiddens/num_heads)
    X = X.reshape(X.shape[0], X.shape[1], num_heads, -1)
    # 输出X的形状:(batch_size，num_heads，查询或者“键 - 值”对的个数，
    # num_hiddens/num_heads)
    X = X.permute(0, 2, 1, 3)
    # 最终输出的形状:(batch_size*num_heads,查询或者“键 - 值”对的个数，
    # num_hiddens/num_heads)
    return X.reshape(-1, X.shape[2], X.shape[3])

#@save
def transpose_output(X, num_heads):
    """逆转transpose_qkv函数的操作"""
    X = X.reshape(-1, num_heads, X.shape[1], X.shape[2])
    X = X.permute(0, 2, 1, 3)
    return X.reshape(X.shape[0], X.shape[1], -1)
```

下面使用键和值相同的小例子来测试我们编写的 MultiHeadAttention 类。多头注意力输出的形状是（batch_size，num_queries，num_hiddens）。

```
num_hiddens, num_heads = 100, 5
attention = MultiHeadAttention(num_hiddens, num_hiddens, num_hiddens,
num_hiddens, num_heads, 0.5)
attention.eval()
```

```
MultiHeadAttention((attention): DotProductAttention(
(dropout): Dropout(p=0.5, inplace=False)
)
(W_q): Linear(in_features=100, out_features=100, bias=False)
(W_k): Linear(in_features=100, out_features=100, bias=False)
(W_v): Linear(in_features=100, out_features=100, bias=False)
(W_o): Linear(in_features=100, out_features=100, bias=False)
)
```

```
batch_size, num_queries = 2, 4
num_kvpairs, valid_lens = 6, torch.tensor([3, 2])
X = torch.ones((batch_size, num_queries, num_hiddens))
Y = torch.ones((batch_size, num_kvpairs, num_hiddens))
attention(X, Y, Y, valid_lens).shape
```

```
torch.Size([2, 4, 100])
```

## 1.3 自注意力和位置编码

在深度学习中，经常使用卷积神经网络（CNN）或循环神经网络（RNN）对序列进行编码。想象一下，有了注意力机制之后，我们将词元序列输入注意力池化中，以便同一组词元同时充当查询、键和值。具体来说，每个查询都会关注所有的键 - 值对并生成一个注意力输出。由于查询、键和值来自同一组输入，因此被称为自注意力（self-attention），也被称为内部注意力（intra-attention）。本节将使用自注意力进行序列编码，以及如何使用序列的顺序作为补充信息。

```
import math
import torch
from torch import nn
from d2l import torch as d2l
```

### 1.3.1 自注意力

给定一个由词元组成的输入序列 $x_1, ..., x_n$，其中任意 $x_i \in \mathbb{R}^d$ $1 \le i \le n$。该序列的自注意力输出为一个长度相同的序列 $y_1, ..., y_n$，其中：

$$y_i = f(x_i, (x_1, x_1), ..., (x_n, x_n)) \in \mathbb{R}^d \tag{1.2}$$

$f$ 注意力汇聚函数。下面的代码片段是基于多头注意力对一个张量完成自注意力的计算，张量的形状为（批量大小，时间步的数目或词元序列的长度，$d$）。输出与输入的张量形状相同。

```
num_hiddens, num_heads = 100, 5
attention = d2l.MultiHeadAttention(num_hiddens, num_hiddens, num_hiddens,
num_hiddens, num_heads, 0.5)
attention.eval()
```

```
MultiHeadAttention(
(attention): DotProductAttention(
(dropout): Dropout(p=0.5, inplace=False)
)
(W_q): Linear(in_features=100, out_features=100, bias=False)
(W_k): Linear(in_features=100, out_features=100, bias=False)
(W_v): Linear(in_features=100, out_features=100, bias=False)
(W_o): Linear(in_features=100, out_features=100, bias=False)
)
```

```
batch_size, num_queries, valid_lens = 2, 4, torch.tensor([3, 2])
X = torch.ones((batch_size, num_queries, num_hiddens))
attention(X, X, X, valid_lens).shape
```

```
torch.Size([2, 4, 100])
```

### 1.3.2 比较卷积神经网络、循环神经网络和自注意力

接下来比较下面几个架构，目标都是将由 n 个词元组成的序列映射到另一个🄵度相等的序列，其中的每个输入词元或输出词元都由 d 维向量表示。具体来说，将比较的是卷积神经网络、循环神经网络和自注意力这几个架构的计算复杂性、顺序操作和最大路径🄵度。请注意，顺序操作会妨碍并行计算，而任意的序列位置组合之间的路径越短，则能更轻松地学习序列中的远距离依赖关系。

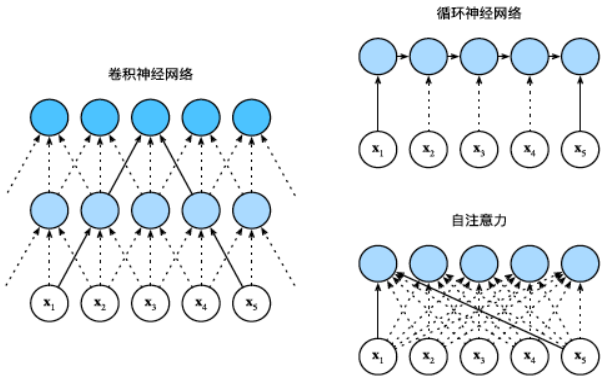



**图 1.3:** 比较卷积神经网络（填充词元被忽略）、循环神经网络和自注意力三种架构。

考虑一个卷积核大小为 $k$ 的卷积层。在后面的章节将提供关于使用卷积神经网络处理序列的更多详细信息。目前只需要知道的是，由于序列长度是 $n$，输入和输出的通道数量都是 $d$，所以卷积层的计算复杂度为 $O(knd^2)$。如图1.3所示，卷积神经网络是分层的，因此为有 $O(1)$ 个顺序操作，最大路径🄵度为 $O(n/k)$。例如，$x_1$ 和 $x_5$ 处于图1.3中卷积核大小为 3 的双层卷积神经网络的感受野内。

### 1.3.3 位置编码

在处理词元序列时，循环神经网络是逐个的重复地处理词元的，而自注意力则因为并行计算而放弃了顺序操作。为了使用序列的顺序信息，通过在输入表示中添加位置编码（positional encoding）来注入绝对的或相对的位置信息。位置编码可以通过学习得到也可以直接固定得到。接下来描述的是基于正弦函数和余弦函数的固定位置编码。

假设输入表示 $X \in 2\mathbb{R}^{n\times d}$ 包含一个序列中 $n$ 个词元的 $d$ 维嵌入表示。位置编码使用相同形状的位置嵌入矩阵 $P \in \mathbb{R}^{n\times d}$ 输出 $X+P$，矩阵第 $i$ 行、第 $2j$ 列和 $2j+1$ 列上的元素为：

$$p_{i,2j} = sin(\frac{i}{10000^{2j/d}}) \tag{1.3}$$

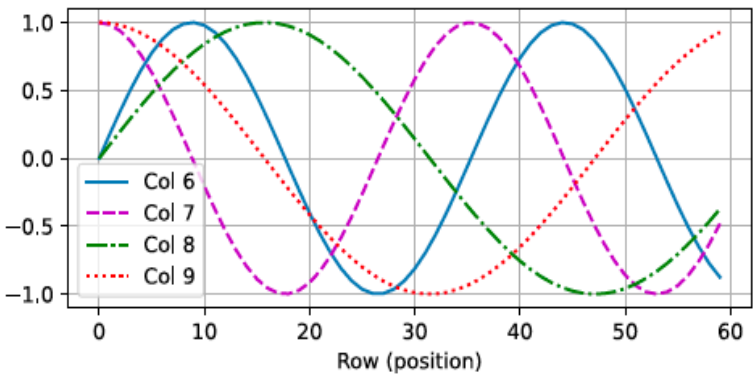
$$p_{i,2j+1} = cos(\frac{i}{10000^{2j/d}}) \tag{1.4}$$

乍一看，这种基于三🄵函数的设计看起来很奇怪。在解释这个设计之前，让我们先在下面的 PositionalEncoding 类中实现它。

```
#@save
class PositionalEncoding(nn.Module):
    """位置编码"""
    def __init__(self, num_hiddens, dropout, max_len=1000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(dropout)
        # 创建一个足够长的P
        self.P = torch.zeros((1, max_len, num_hiddens))
        X = torch.arange(max_len, dtype=torch.float32).reshape(
        -1, 1) / torch.pow(10000, torch.arange(
        0, num_hiddens, 2, dtype=torch.float32) / num_hiddens)
        self.P[:, :, 0::2] = torch.sin(X)
        self.P[:, :, 1::2] = torch.cos(X)
    def forward(self, X):
        X = X + self.P[:, :X.shape[1], :].to(X.device)
        return self.dropout(X)
```

在位置嵌入矩阵 P 中，行代表词元在序列中的位置，列代表位置编码的不同维度。从下面的例子中可以看到位置嵌入矩阵的第 6 列和第 7 列的频率高于第 8 列和第 9 列。第 6 列和第 7 列之间的偏移量（第 8 列和第 9 列相同）是由于正弦函数和余弦函数的交替。

```
encoding_dim, num_steps = 32, 60
pos_encoding = PositionalEncoding(encoding_dim, 0)
pos_encoding.eval()
X = pos_encoding(torch.zeros((1, num_steps, encoding_dim)))
P = pos_encoding.P[:, :X.shape[1], :]
d2l.plot(torch.arange(num_steps), P[0, :, 6:10].T, xlabel='Row (position)',
figsize=(6, 2.5), legend=["Col %d" % d for d in torch.arange(6, 10)])
```



# 1.4 Transformer

Transformer 模型完全基于注意力机制，没有任何卷积层或循环神经网络层。尽管 Transformer 最初是应用于在文本数据上的序列到序列学习，但现在已经推广到各种现代的深度学习中，例如语言、视觉、语音和强化学习领域。

## 1.4.1 模型

Transformer 作为编码器 - 解码器架构的一个实例，其整体架构图在图1.4中展示。正如所见到的，Transformer 是由编码器和解码器组成的。Transformer 的编码器和解码器是基于自注意力的模块叠加而成的，源（输入）序列和目标（输出）序列的嵌入（embedding）表示将加上位置编码（positional encoding），再分别输入到编码器和解码器中。

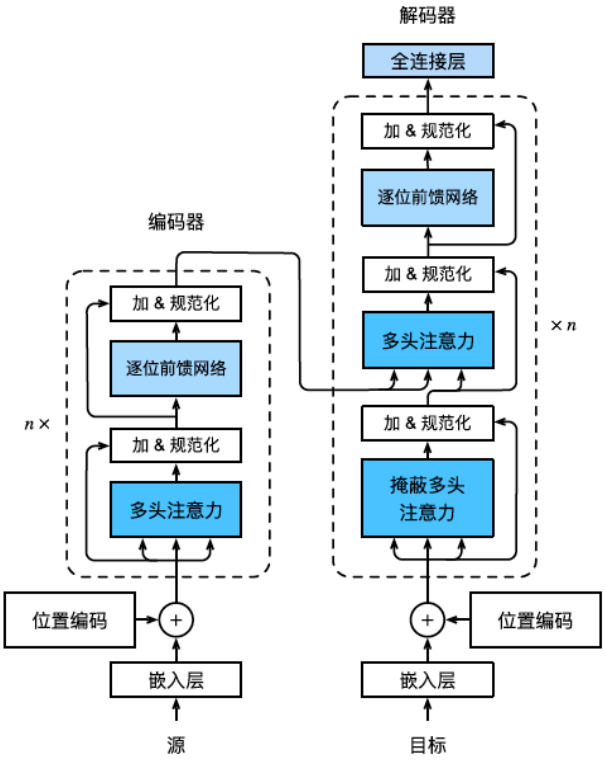



**图 1.4:** Transformer 架构。

图图1.4中概述了 Transformer 的架构。从宏观角度来看，Transformer 的编码器是由多个相同的层叠加而成的，每个层都有两个子层（子层表示为 sublayer）。第一个子层是多头自注意力（multi-head self-attention）汇聚；第二个子层是基于位置的前馈网络（positionwise feed-forward network）。具体来说，在计算编码器的自注意力时，查询、键和值都来自前一个编码器层的输出。受残差网络的启发，每个子层都采用了残差连接（residual connection）。在 Transformer 中，对于序列中任何位置的任何输入 $x \in \mathbb{R}^d$，都要求满足 $sublayer(x) \in \mathbb{R}^d$，以便残差连接满足 $x + sublayer(x) \in \mathbb{R}^d$。在残差连接的加法计算之后，紧接着应用层规范化（layer normalization）。因此，输入序列对应的每个位置，Transformer 编码器都将输出一个 d 维表示向量。

Transformer 解码器也是由多个相同的层叠加而成的，并且层中使用了残差连接和层规范化。除了编码器中描述的两个子层之外，解码器还在这两个子层之间插入了第三个子层，称为编码器 - 解码器注意力（encoderdecoder attention）层。在编码器 - 解码器注意力中，查询来自前一个解码器层的输出，而键和值来自整个编码器的输出。在解码器自注意力中，查询、键和值都来自上一个解码器层的输出。但是，解码器中的每个位置只能考虑该位置之前的所有位置。这种掩蔽（masked）注意力保留了自回归（auto-regressive）属性，确保预测仅依赖于已生成的输出词元。

```
import math
import pandas as pd
import torch
from torch import nn
from d2l import torch as d2l
```

### 1.4.2 基于位置的前馈网络

基于位置的前馈网络对序列中的所有位置的表示进行变换时使用的是同一个多层感知机（MLP），这就是称前馈网络是基于位置的（positionwise）的原因。在下面的实现中，输入 X 的形状（批量大小，时间步数或序列🄵度，隐单元数或特征维度）将被一个两层的感知机转换成形状为（批量大小，时间步数，ffn_num_outputs）的输出张量。

```
#@save
class PositionWiseFFN(nn.Module):
"""基于位置的前馈网络"""
    def __init__(self, ffn_num_input, ffn_num_hiddens, ffn_num_outputs,
    **kwargs):
        super(PositionWiseFFN, self).__init__(**kwargs)
        self.dense1 = nn.Linear(ffn_num_input, ffn_num_hiddens)
        self.relu = nn.ReLU()
        self.dense2 = nn.Linear(ffn_num_hiddens, ffn_num_outputs)
    def forward(self, X):
        return self.dense2(self.relu(self.dense1(X)))
```

下面的例子显示，改变张量的最里层维度的尺寸，会改变成基于位置的前馈网络的输出尺寸。因为用同一个多层感知机对所有位置上的输入进行变换，所以当所有这些位置的输入相同时，它们的输出也是相同的。

```
ffn = PositionWiseFFN(4, 4, 8)
ffn.eval()
ffn(torch.ones((2, 3, 4)))[0]
```

```
tensor([[ 0.3407, -0.0869, -0.3967, 0.7588, 0.3862, 0.2616, 0.1842, -0.0328],
[ 0.3407, -0.0869, -0.3967, 0.7588, 0.3862, 0.2616, 0.1842, -0.0328],
[ 0.3407, -0.0869, -0.3967, 0.7588, 0.3862, 0.2616, 0.1842, -0.0328]],
grad_fn=<SelectBackward0>)
```

### 1.4.3 残差连接和层规范化

现在让我们关注图1.4中的加法和规范化（addnorm）组件。正如在本节开头所述，这是由残差连接和紧随其后的层规范化组成的。两者都是构建有效的深度架构的关键。

前面解释了在一个小批量的样本内基于批量规范化对数据进行重新中心化和重新缩放的调整。层规范化和批量规范化的目标相同，但层规范化是基于特征维度进行规范化。尽管批量规范化在计算机视觉中被广泛应用，但在自然语言处理任务中（输入通常是变🄵序列）批量规范化通常不如层规范化的效果好。

以下代码对比不同维度的层规范化和批量规范化的效果。

```
ln = nn.LayerNorm(2)
bn = nn.BatchNorm1d(2)
```

```
X = torch.tensor([[1, 2], [2, 3]], dtype=torch.float32)
# 在训练模式下计算X的均值和方差
print('layer norm:', ln(X), '\nbatch norm:', bn(X))
```

```
layer norm: tensor([[-1.0000, 1.0000],
[-1.0000, 1.0000]], grad_fn=<NativeLayerNormBackward0>)
batch norm: tensor([[-1.0000, -1.0000],
[ 1.0000, 1.0000]], grad_fn=<NativeBatchNormBackward0>)
```

现在可以使用残差连接和层规范化来实现 AddNorm 类。暂退法也被作为正则化方法使用。

```
#@save
class AddNorm(nn.Module):
    """残差连接后进行层规范化"""
    def __init__(self, normalized_shape, dropout, **kwargs):
        super(AddNorm, self).__init__(**kwargs)
        self.dropout = nn.Dropout(dropout)
        self.ln = nn.LayerNorm(normalized_shape)
    def forward(self, X, Y):
        return self.ln(self.dropout(Y) + X)
```

残差连接要求两个输入的形状相同，以便加法操作后输出张量的形状相同。

```
add_norm = AddNorm([3, 4], 0.5)
add_norm.eval()
add_norm(torch.ones((2, 3, 4)), torch.ones((2, 3, 4))).shape
```

```
torch.Size([2, 3, 4])
```

### 1.4.4 编码器

有了组成 Transformer 编码器的基础组件，现在可以先实现编码器中的一个层。下面的 EncoderBlock 类包含两个子层：多头自注意力和基于位置的前馈网络，这两个子层都使用了残差连接和紧随的层规范化。

```
#@save
class EncoderBlock(nn.Module):
    """Transformer编码器块"""
    def __init__(self, key_size, query_size, value_size, num_hiddens,norm_shape, ffn_num_input, ffn_
         num_hiddens, num_heads,dropout, use_bias=False, **kwargs):
        super(EncoderBlock, self).__init__(**kwargs)
        self.attention = d2l.MultiHeadAttention(
        key_size, query_size, value_size, num_hiddens, num_heads, dropout,
        use_bias)
        self.addnorm1 = AddNorm(norm_shape, dropout)
        self.ffn = PositionWiseFFN(
        ffn_num_input, ffn_num_hiddens, num_hiddens)
        self.addnorm2 = AddNorm(norm_shape, dropout)
    def forward(self, X, valid_lens):
        Y = self.addnorm1(X, self.attention(X, X, X, valid_lens))
        return self.addnorm2(Y, self.ffn(Y))
```

正如从代码中所看到的，Transformer 编码器中的任何层都不会改变其输入的形状。

```
X = torch.ones((2, 100, 24))
valid_lens = torch.tensor([3, 2])
encoder_blk = EncoderBlock(24, 24, 24, 24, [100, 24], 24, 48, 8, 0.5)
encoder_blk.eval()
encoder_blk(X, valid_lens).shape
```

```
torch.Size([2, 100, 24])
```

下面实现的 Transformer 编码器的代码中，堆叠了 num_layers 个 EncoderBlock 类的实例。由于这里使用的是值范围在-1 和 1 之间的固定位置编码，因此通过学习得到的输入的嵌入表示的值需要先乘以嵌入维度的平方根进行重新缩放，然后再与位置编码相加。

```
#@save
class TransformerEncoder(d2l.Encoder):
    """Transformer编码器"""
    def __init__(self, vocab_size, key_size, query_size, value_size,
    num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens,
    num_heads, num_layers, dropout, use_bias=False, **kwargs):
        super(TransformerEncoder, self).__init__(**kwargs)
        self.num_hiddens = num_hiddens
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add_module("block"+str(i),
            EncoderBlock(key_size, query_size, value_size, num_hiddens,
            norm_shape, ffn_num_input, ffn_num_hiddens,
            num_heads, dropout, use_bias))
    def forward(self, X, valid_lens, *args):
        # 因为位置编码值在-1和1之间，
        # 因此嵌入值乘以嵌入维度的平方根进行缩放，
        # 然后再与位置编码相加。
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self.attention_weights = [None] * len(self.blks)
        for i, blk in enumerate(self.blks):
            X = blk(X, valid_lens)
            self.attention_weights[        i] = blk.attention.attention.attention_weights
        return X
```

下面我们指定了超参数来创建一个两层的 Transformer 编码器。Transformer 编码器输出的形状是（批量大小，时间步数目，num_hiddens）。

```
encoder = TransformerEncoder(
200, 24, 24, 24, 24, [100, 24], 24, 48, 8, 2, 0.5)
encoder.eval()
encoder(torch.ones((2, 100), dtype=torch.long), valid_lens).shape
```

```
torch.Size([2, 100, 24])
```

### 1.4.5 解码器

如图1.4所示，Transformer 解码器也是由多个相同的层组成。在 DecoderBlock 类中实现的每个层包含了三个子层：解码器自注意力、“编码器-解码器” 注意力和基于位置的前馈网络。这些子层也都被残差连接和紧随的层规范化围绕。正如在本节前面所述，在掩蔽多头解码器自注意力层（第一个子层）中，查询、键和值都来自上一个解码器层的输出。关于序列到序列模型（sequence-to-sequence model），在训练阶段，其输出序列的所有位置（时间步）的词元都是已知的；然而，在预测阶段，其输出序列的词元是逐个生成的。因此，在任何解码器时间步中，只有生成的词元才能用于解码器的自注意力计算中。为了在解码器中保留自回归的属性，其掩蔽自注意力设定了参数 dec_valid_lens，以便任何查询都只会与解码器中所有已经生成词元的位置（即直到该查询位置为止）进行注意力计算。

```
class DecoderBlock(nn.Module):
    """解码器中第i个块"""
    def __init__(self, key_size, query_size, value_size, num_hiddens,
    norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
    dropout, i, **kwargs):
        super(DecoderBlock, self).__init__(**kwargs)
        self.i = i
        self.attention1 = d2l.MultiHeadAttention(
        key_size, query_size, value_size, num_hiddens, num_heads, dropout)
        self.addnorm1 = AddNorm(norm_shape, dropout)
        self.attention2 = d2l.MultiHeadAttention(
        key_size, query_size, value_size, num_hiddens, num_heads, dropout)
        self.addnorm2 = AddNorm(norm_shape, dropout)
        self.ffn = PositionWiseFFN(ffn_num_input, ffn_num_hiddens,
        num_hiddens)
        self.addnorm3 = AddNorm(norm_shape, dropout)
    def forward(self, X, state):
        enc_outputs, enc_valid_lens = state[0], state[1]
        # 训练阶段，输出序列的所有词元都在同一时间处理，
        # 因此state[2][self.i]初始化为None。
        # 预测阶段，输出序列是通过词元一个接着一个解码的，
        # 因此state[2][self.i]包含着直到当前时间步第i个块解码的输出表示
        if state[2][self.i] is None:
            key_values = X
        else:
            key_values = torch.cat((state[2][self.i], X), axis=1)
        state[2][self.i] = key_values

        if self.training:
            batch_size, num_steps, _ = X.shape
            # dec_valid_lens的开头:(batch_size,num_steps),
            # 其中每一行是[1,2,...,num_steps]
            dec_valid_lens = torch.arange(
            1, num_steps + 1, device=X.device).repeat(batch_size, 1)
        else:
            dec_valid_lens = None
        # 自注意力
        X2 = self.attention1(X, key_values, key_values, dec_valid_lens)
```

```
        Y = self.addnorm1(X, X2)
        # 编码器 - 解码器注意力。
        # enc_outputs的开头:(batch_size,num_steps,num_hiddens)
        Y2 = self.attention2(Y, enc_outputs, enc_outputs, enc_valid_lens)
        Z = self.addnorm2(Y, Y2)
        return self.addnorm3(Z, self.ffn(Z)), state
```

为了便于在“编码器 - 解码器”注意力中进行缩放点积计算和残差连接中进行加法计算，编码器和解码器的特征维度都是 num_hiddens。

```
decoder_blk = DecoderBlock(24, 24, 24, 24, [100, 24], 24, 48, 8, 0.5, 0)
decoder_blk.eval()
X = torch.ones((2, 100, 24))
state = [encoder_blk(X, valid_lens), valid_lens, [None]]
decoder_blk(X, state)[0].shape
```

```
torch.Size([2, 100, 24])
```

现在我们构建了由 num_layers 个 DecoderBlock 实例组成的完整的 Transformer 解码器。最后，通过一个全连接层计算所有 vocab_size 个可能的输出词元的预测值。解码器的自注意力权重和编码器解码器注意力权重都被存储下来，方便日后可视化的需要。

```
class TransformerDecoder(d2l.AttentionDecoder):
    def __init__(self, vocab_size, key_size, query_size, value_size,
    num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens,
    num_heads, num_layers, dropout, **kwargs):
        super(TransformerDecoder, self).__init__(**kwargs)
        self.num_hiddens = num_hiddens
        self.num_layers = num_layers
        self.embedding = nn.Embedding(vocab_size, num_hiddens)

        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add_module("block"+str(i),
            DecoderBlock(key_size, query_size, value_size, num_hiddens,
            norm_shape, ffn_num_input, ffn_num_hiddens,
            num_heads, dropout, i))
        self.dense = nn.Linear(num_hiddens, vocab_size)
    def init_state(self, enc_outputs, enc_valid_lens, *args):
        return [enc_outputs, enc_valid_lens, [None] * self.num_layers]


    def forward(self, X, state):
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self._attention_weights = [[None] * len(self.blks) for _ in range (2)]
        for i, blk in enumerate(self.blks):
            X, state = blk(X, state)
            # 解码器自注意力权重
            self._attention_weights[0][
```

```
            i] = blk.attention1.attention.attention_weights
            # “编码器 - 解码器” 自注意力权重
            self._attention_weights[1][
            i] = blk.attention2.attention.attention_weights
        return self.dense(X), state

    @property
    def attention_weights(self):
        return self._attention_weights
```

### 1.4.6 训练

依照 Transformer 架构来实例化编码器 - 解码器模型。在这里，指定 Transformer 的编码器和解码器都是 2 层，都使用 4 头注意力。为了进行序列到序列的学习，下面在 “英语 - 法语” 机器翻译数据集上训练 Transformer 模型。

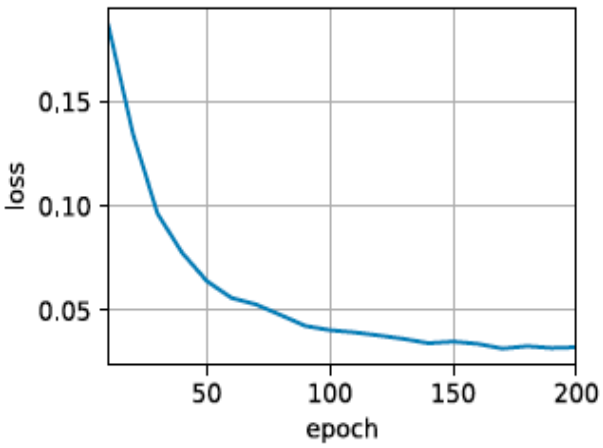
```
num_hiddens, num_layers, dropout, batch_size, num_steps = 32, 2, 0.1, 64, 10
lr, num_epochs, device = 0.005, 200, d2l.try_gpu()
ffn_num_input, ffn_num_hiddens, num_heads = 32, 64, 4
key_size, query_size, value_size = 32, 32, 32
norm_shape = [32]

train_iter, src_vocab, tgt_vocab = d2l.load_data_nmt(batch_size, num_steps)
encoder = TransformerEncoder(len(src_vocab), key_size, query_size, value_size, num_hiddens,
norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
num_layers, dropout)

decoder = TransformerDecoder(
len(tgt_vocab), key_size, query_size, value_size, num_hiddens,
norm_shape, ffn_num_input, ffn_num_hiddens, num_heads,
num_layers, dropout)

net = d2l.EncoderDecoder(encoder, decoder)

d2l.train_seq2seq(net, train_iter, lr, num_epochs, tgt_vocab, device)
```

```
loss 0.032, 5679.3 tokens/sec on cuda:0
```

训练结束后，使用 Transformer 模型将一些英语句子翻译成法语，并且计算它们的 BLEU 分数。

```
engs = ['go .', "i lost .", 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
for eng, fra in zip(engs, fras):
    translation, dec_attention_weight_seq = d2l.predict_seq2seq(net, eng, src_vocab, tgt_vocab, num_
        steps, device, True)
    print(f'{eng} => {translation}, ',f'bleu {d2l.bleu(translation, fra, k=2):.3f}')
```

```
go . => va !, bleu 1.000
i lost . => j'ai perdu ., bleu 1.000
he's calm . => il est calme ., bleu 1.000
```




```
i'm home . => je suis chez moi ., bleu 1.000
```

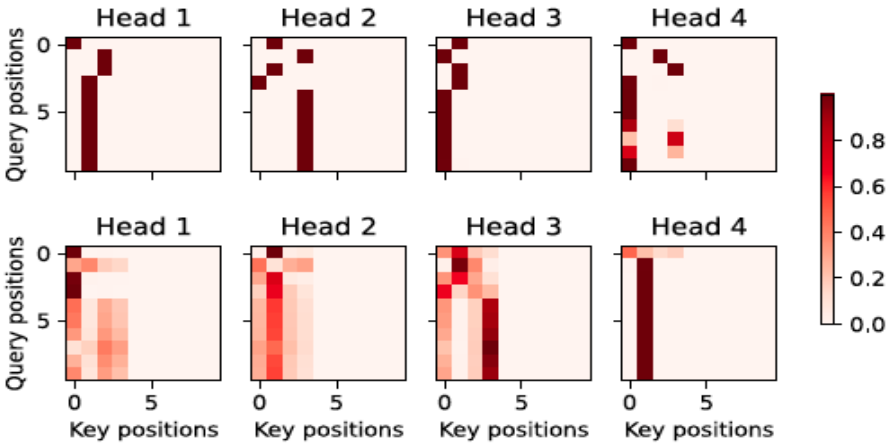
当进行最后一个英语到法语的句子翻译工作时，让我们可视化 Transformer 的注意力权重。编码器自注意力权重的形状为（编码器层数，注意力头数，num_steps 或查询的数目，num_steps 或“键 - 值”对的数目）。

```
enc_attention_weights = torch.cat(net.encoder.attention_weights, 0).reshape((num_layers, num_heads,
-1, num_steps))
enc_attention_weights.shape
```

```
torch.Size([2, 4, 10, 10])
```

在编码器的自注意力中，查询和键都来自相同的输入序列。因为填充词元是不携带信息的，因此通过指定输入序列的有效长度可以避免查询与使用填充词元的位置计算注意力。接下来，将逐行呈现两层多头注意力的权重。每个注意力头都根据查询、键和值的不同的表示子空间来表示不同的注意力。

```
d2l.show_heatmaps(enc_attention_weights.cpu(), xlabel='Key positions',
ylabel='Query positions', titles=['Head %d' % i for i in range(1, 5)],
figsize=(7, 3.5))
```



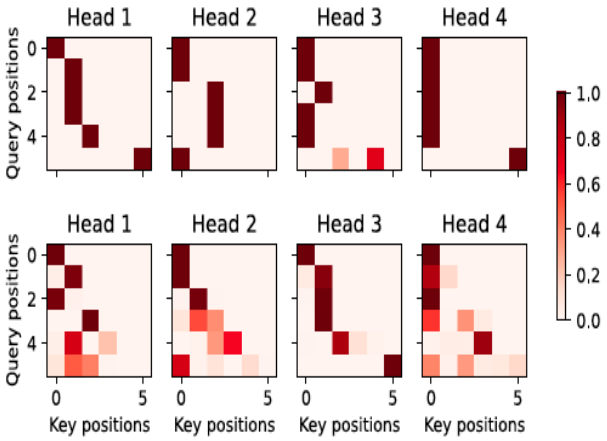

为了可视化解码器的自注意力权重和“编码器 - 解码器”的注意力权重，我们需要完成更多的数据操作工作。例如用零填充被掩蔽住的注意力权重。值得注意的是，解码器的自注意力权重和“编码器 - 解码器”的注意力权重都有相同的查询：即以序列开始词元（beginning-of-sequence, BOS）打头，再与后续输出的词元共同组成序列。

```
dec_attention_weights_2d = [head[0].tolist()
                            for step in dec_attention_weight_seq
                            for attn in step for blk in attn for head in blk]
dec_attention_weights_filled = torch.tensor(pd.DataFrame(dec_attention_weights_2d).fillna(0.0).values)
dec_attention_weights = dec_attention_weights_filled.reshape((-1, 2, num_layers, num_heads, num_steps)
    )
dec_self_attention_weights, dec_inter_attention_weights = \
dec_attention_weights.permute(1, 2, 3, 0, 4)
dec_self_attention_weights.shape, dec_inter_attention_weights.shape
```

```
(torch.Size([2, 4, 6, 10]), torch.Size([2, 4, 6, 10]))
```

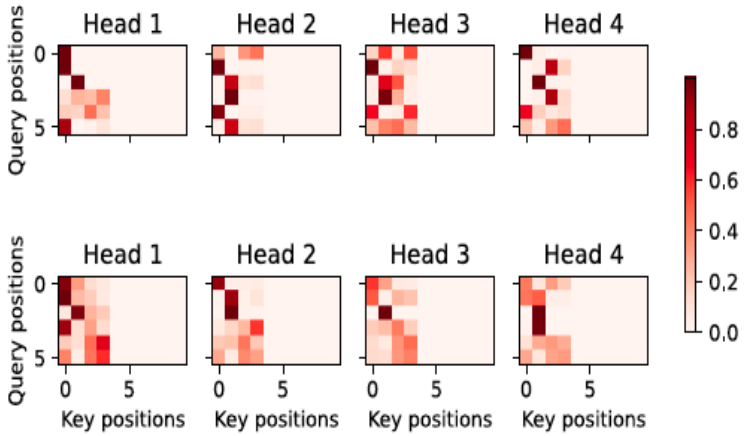
由于解码器自注意力的自回归属性，查询不会对当前位置之后的“键 - 值”对进行注意力计算。

```
# Plusonetoincludethebeginning-of-sequencetoken
d2l.show_heatmaps(dec_self_attention_weights[:, :, :, :len(translation.split()) + 1],
xlabel='Key positions', ylabel='Query positions',
titles=['Head %d' % i for i in range(1, 5)], figsize=(7, 3.5))
```




与编码器的自注意力的情况类似，通过指定输入序列的有效长度，输出序列的查询不会与输入序列中填充位置的词元进行注意力计算。

```
d2l.show_heatmaps(dec_inter_attention_weights, xlabel='Key positions',
ylabel='Query positions', titles=['Head %d' % i for i in range(1, 5)],
figsize=(7, 3.5))
```


Head 1
Head 2
Head 3
Head 4
Query positions
0
5
Head 1
Head 2
Head 3
Head 4
Query positions
0
5
0
5
Key positions
0
5
Key positions
0
5
Key positions
0
5
Key positions
0.8
0.6
0.4
0.2
0.0