

1. 卷积神经网络

1.1 CNN 张量理解

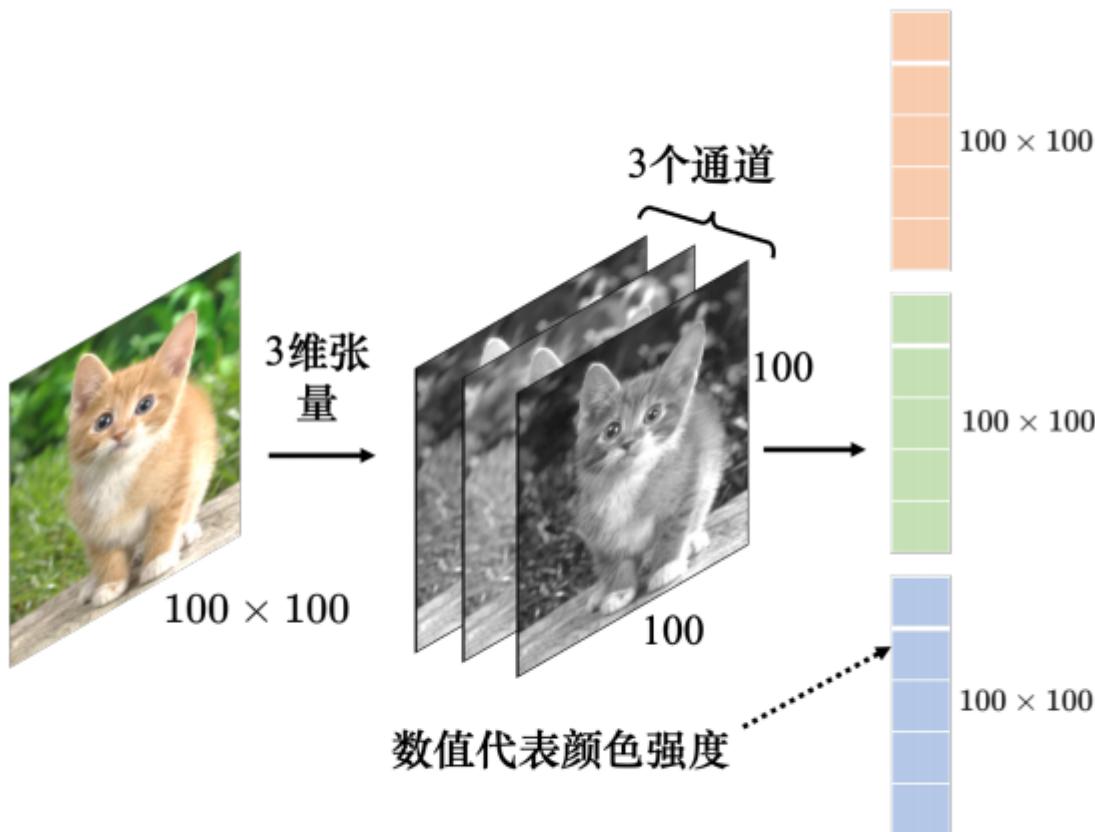


图 4.1 把图像作为模型输入

Tensor就是按照 **通道** 方向继续叠加多个矩阵

1.1.2 一维张量

代表这向量，也分行向量和列向量

```
true_w = torch.tensor([2, -3.4]) # 这个就是一维张量，也就是向量
```

1.1.2.1 通过arrange生成

```
# 使用张量arrange设定步长
x = torch.arange(0, 10, 2) # arange是左闭右开
print("x: ", x)
print("x.shape", x.shape)
print("x.size:", x.size())
```

```
x: tensor([0, 2, 4, 6, 8])
x.shape torch.Size([5])
x.size: torch.Size([5])
```

1.1.3 二维张量

代表着矩阵

1.1.3.1 随机二维张量构建

```
...
Author: Jean_Leung
Date: 2024-09-02 11:00:52
LastEditors: Jean_Leung
LastEditTime: 2024-09-02 11:01:10
FilePath: \LinearModel\tensorpratice\tensor_practise.py
Description: CNN 张量理解

Copyright (c) 2024 by ${robotlive limit}, All Rights Reserved.
...

import torch

x = torch.randn(3,4) # 随机生成的数据
print("x: ", x)
```

```
x: tensor([[ 0.0704,  0.8286, -1.3293, -0.9820],
           [ 0.5927,  0.5964,  1.3840, -0.7827],
           [ 0.2429, -1.1099,  0.3849,  0.3374]])
```

1.1.3.2 全0二维张量构建

```
import torch
x = torch.zeros(4, 3, dtype=torch.long)
print(x)
```

```
tensor([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]])
```

1.1.3.3 全1二维张量构建

```
# 全一二维张量
x = torch.ones(4, 3, dtype=torch.float32)
print("x : ", x)
```

```
x : tensor([[1., 1., 1.],
           [1., 1., 1.],
           [1., 1., 1.],
           [1., 1., 1.]])
```

1.1.3.3 tensor构建

构建张量数据

```
# 通过tensor构建

x = torch.tensor([[1,2,3],[4,5,6],[7,8,9]])
print("x: ", x)

x:  tensor([[1, 2, 3],
           [4, 5, 6],
           [7, 8, 9]])
```

基于之前存在tensor构建新tensor

```
# 基于之前存在tensor构建新tensor
x = torch.ones(4, 4, dtype=torch.float32)
print("x: ", x)
# 基于该张量的size重新搞一个随机张量
x = torch.rand_like(x, dtype=torch.double)
print("x: ", x)
print("x.shape:", x.shape)
print("x.size:", x.size())
```

1.1.3.4 normal构建

标准正态分布中，均值为0，标准差为0.1的矩阵(全部数据都满足正态分布的矩阵)

```
# 使用normal设定
x = torch.normal(mean=0, std=1, size=(4, 4))
print("x: ", x)

x:  tensor([[-0.0531,  0.0341, -0.0902,  0.1080],
           [ 0.0705,  0.0873, -0.0074, -0.0487],
           [-0.0991,  0.1540,  0.1141, -0.1020],
           [ 0.1001, -0.2636, -0.0637, -0.0545]])
```

使每个数据都满足标准正态分布，均值为从0, 1, 2, 3，标准差为 1,0.9,0.8,0.7,重塑矩阵为(2,2)大小

```
x = torch.normal(mean=torch.arange(4.), std=torch.arange(1., 0.6, -0.1)).reshape(2, 2)
print("x: ", x)
```

1.1.4 tensor 构建总结

函数	功能
Tensor(sizes)	基础构造函数
tensor(data)	类似于np.array
ones(sizes)	全1
zeros(sizes)	全0
eye(sizes)	对角为1, 其余为0
arange(s,e,step)	从s到e, 步长为step
linspace(s,e,steps)	从s到e, 均匀分成step份
rand/randn(sizes)	rand是[0,1]均匀分布; randn是服从N(0, 1)的正态分布
normal(mean,std)	正态分布(均值为mean, 标准差是std)
randperm(m)	随机排列

1.2 CNN模型的原理理解

1.2.1 Receptive field(感受野)

需要解决问题: 如果 **一个神经元** 负责 **整个图片** 的识别,那么需要抓取特征(例如鸟的嘴巴, 鸟的眼睛等等, 那么效率会大大折扣)

1.2.1.1 解决方法

卷积神经网络会设定一个区域:感受野(receptive filed)

每个神经元都只关心自己的感受野里面发生的事情, 感受野是由我们自己决定的

该感受野 是这个神经元负责, 那么相当于此 **神经元** 只需要负责

$3 \times 3 \times 3$ (后面的3是通道数)的部分的张量

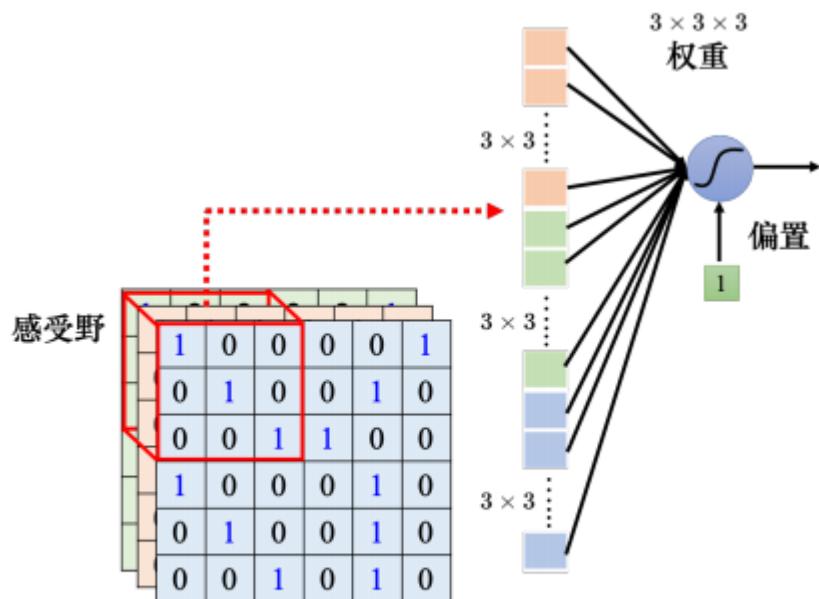


图 4.6 感受野

多个神经元检测同一个感受野 = 一个感受野由不同的神经元去守备

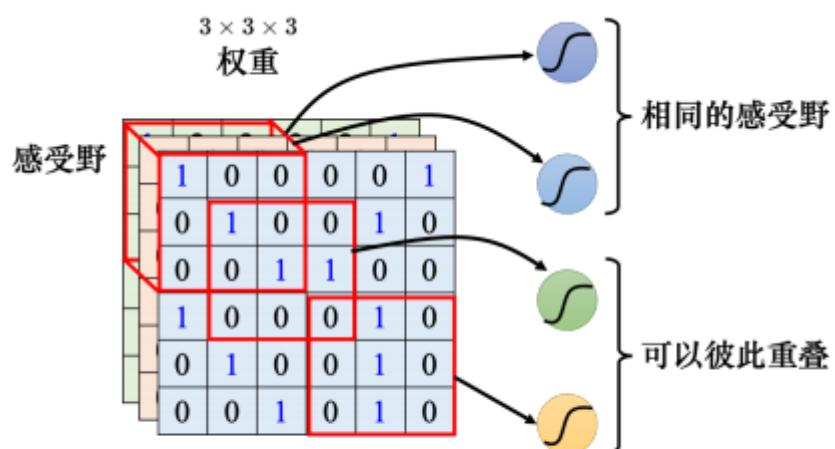


图 4.7 感受野彼此重叠

部分 神经元只负责 **一个通道** 的守备,比如某个神经元只负责守备R通道或者只负责守备G通道

1.2.2 filter(卷积核)

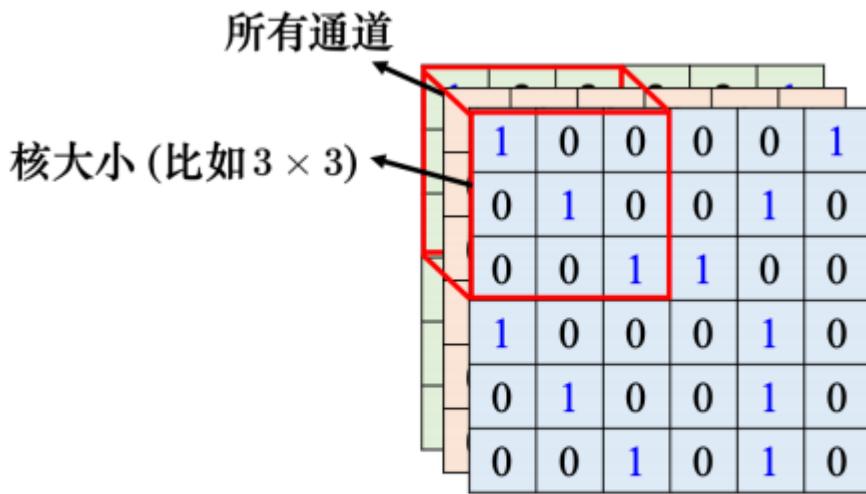


图 4.8 卷积核

核大小不会设太大 3×3 的核大小差不多足够

卷积核的深度=通道数，所以一般只讲核大小(宽*高)

1.2.3 stride(步幅)

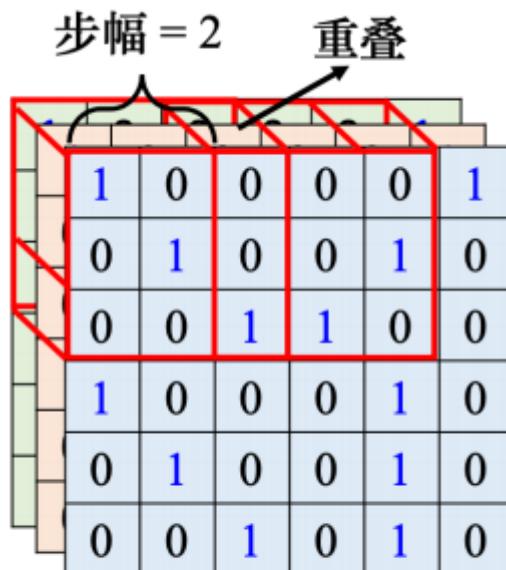


图 4.9 步幅

现在来探讨不同感受野之间的关系

感受野移动的量=步幅

希望各个感受野之间是重叠的，因为没重叠的话，会导致两个感受野之间的交界上没有检测到

Q: 为什么希望感受野之间是有重叠的呢? A: 因为假设感受野完全没有重叠, 如果有一个模式正好出现在两个感受野的交界上面, 就没有任何神经元去检测它, 这个模式可能会丢失, 所以希望感受野彼此之间有高度的重叠。如令步幅 = 2, 感受野就会重叠。

1.2.4 padding(填充)

需要解决的问题: 如果感受野超出图像的范围, 就是没有神经元去检测边界, 这么会漏掉图像边界的地方

解决方法: 零填充zero padding, 超出范围补0

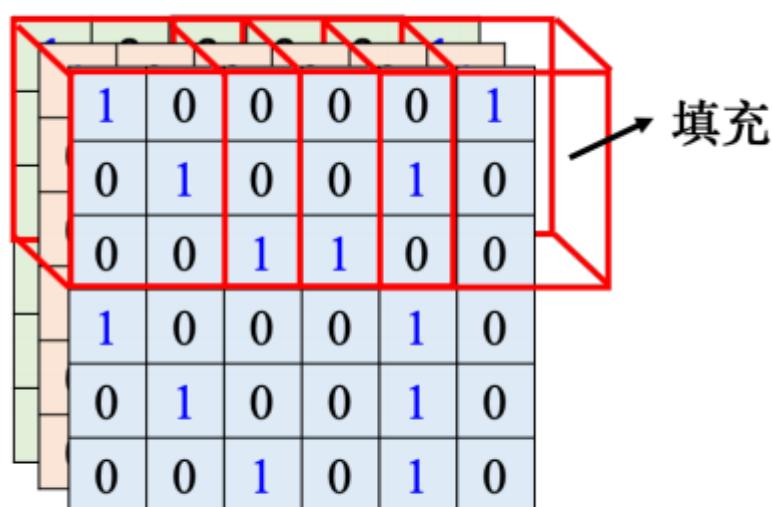


图 4.10 填充

1.2.5 (parameter sharing) 共享参数

需要解决的问题: 同一个特征(例如鸟嘴)出现在图片不同的区域, 也就是说每个感受野都有可能出现鸟嘴, 那么是否需要每个感受野都要设置鸟嘴的检测器??? 也就是神经元组的参数是都要设置不同参数????

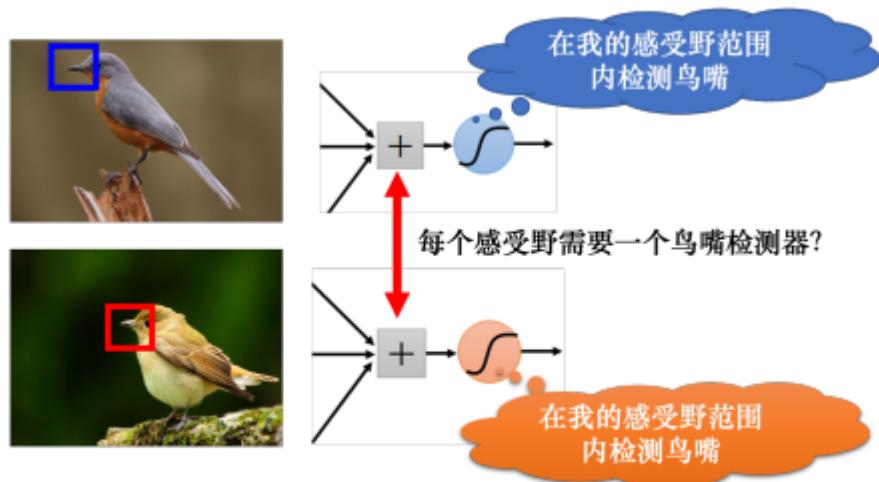


图 4.12 每个感受野都放一个鸟嘴检测器

解决方法: 共享参数= 不同的感受野的神经元共享参数

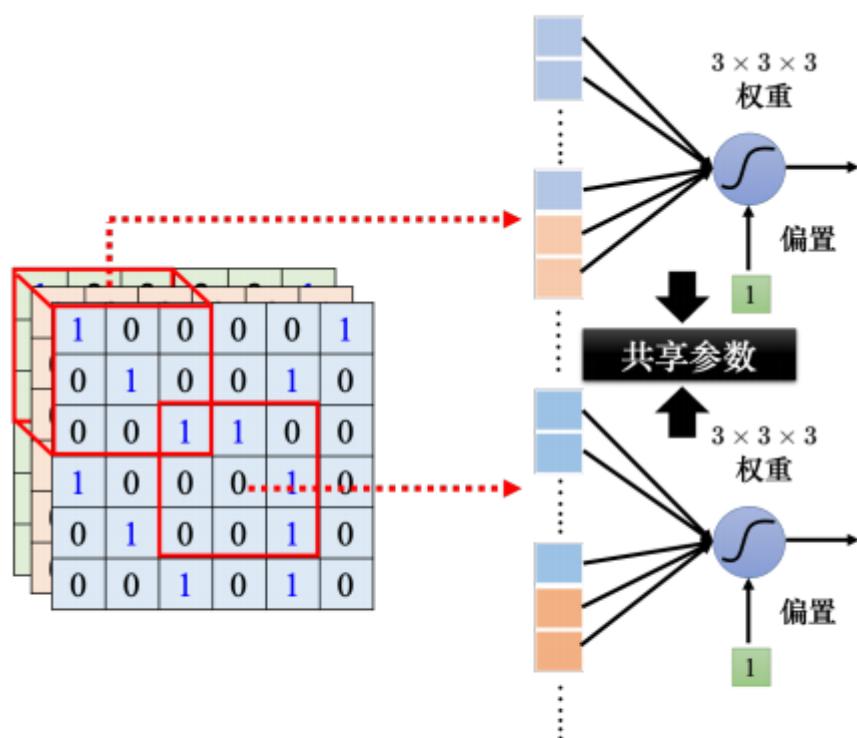


图 4.13 共享参数

1.2.5.1 两个神经元共享参数举例

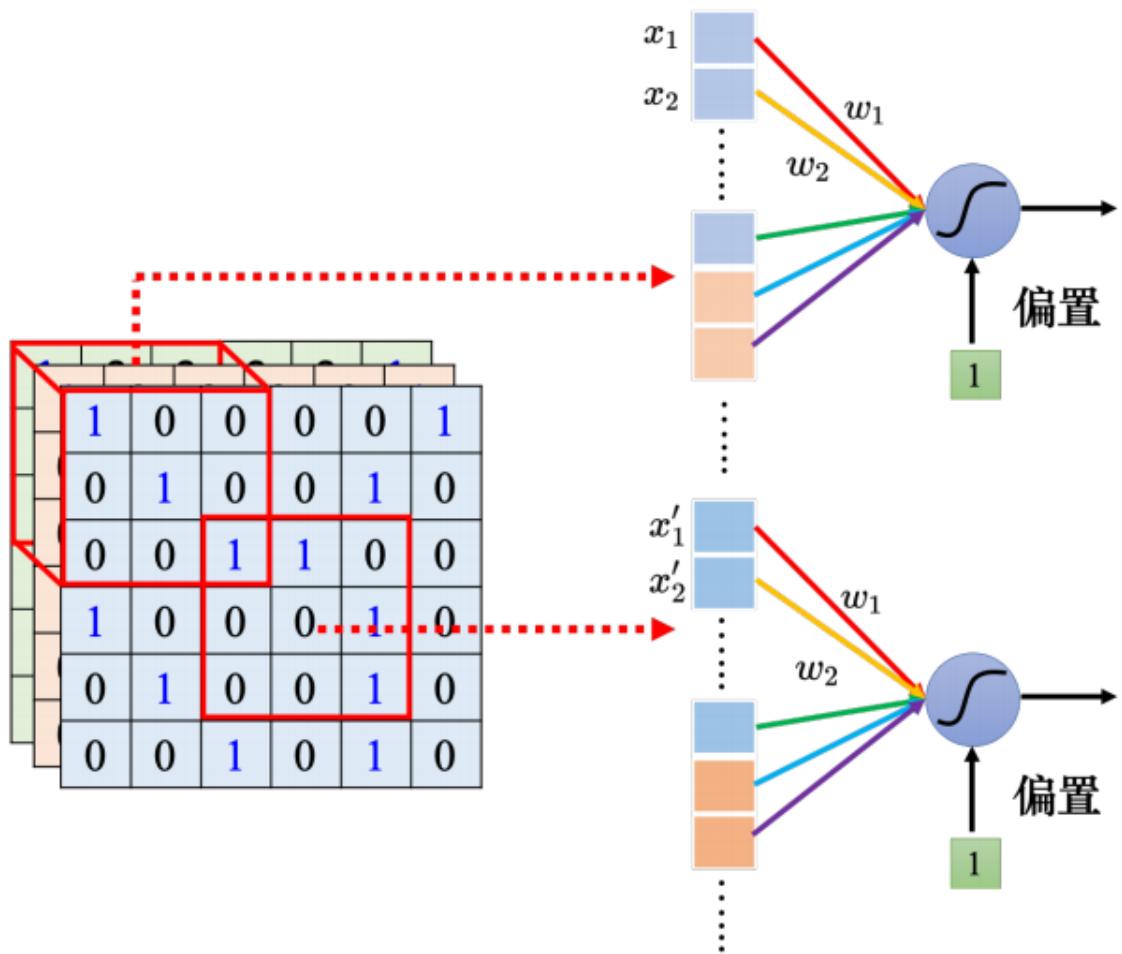


图 4.14 两个神经元共享参数

例如 这两个感受野可能都有出现同一个特征,那么我们就要实现共享参数

上面的输入参数为

$$x_1 \dots x_{27}$$

由之前非线性相加的公式可以得到,上面神经元的输入是 x_1, x_2, \dots, x_7 ,

下面神经元输入是 $x'_1, x'_2, \dots, x'_{13}$

$$\mathbf{H}^{(1)} = \sigma_1 \left(\mathbf{XW}^{(1)} + \mathbf{b}^{(1)} \right) \text{ 和 } \mathbf{H}^{(2)} = \sigma_2 \left(\mathbf{H}^{(1)} \mathbf{W}^{(2)} + \mathbf{b}^{(2)} \right)$$

$$\begin{aligned} & \sigma(w_1 x_1 + w_2 x_2 + \dots + 1) \\ & \sigma(w_1 x'_1 + w_2 x'_2 + \dots + 1) \end{aligned}$$

也就是说W的参数是一致的

1.2.6 二维卷积过程 (2D Convolutional)

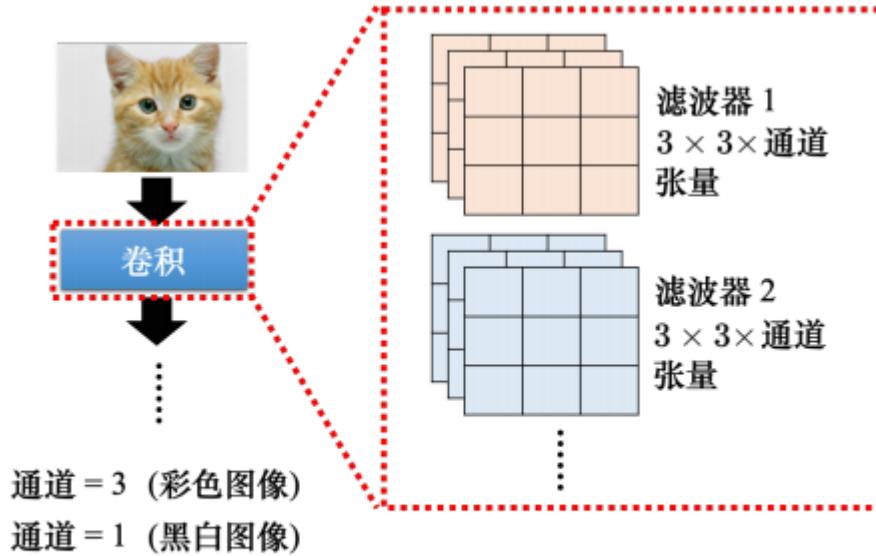
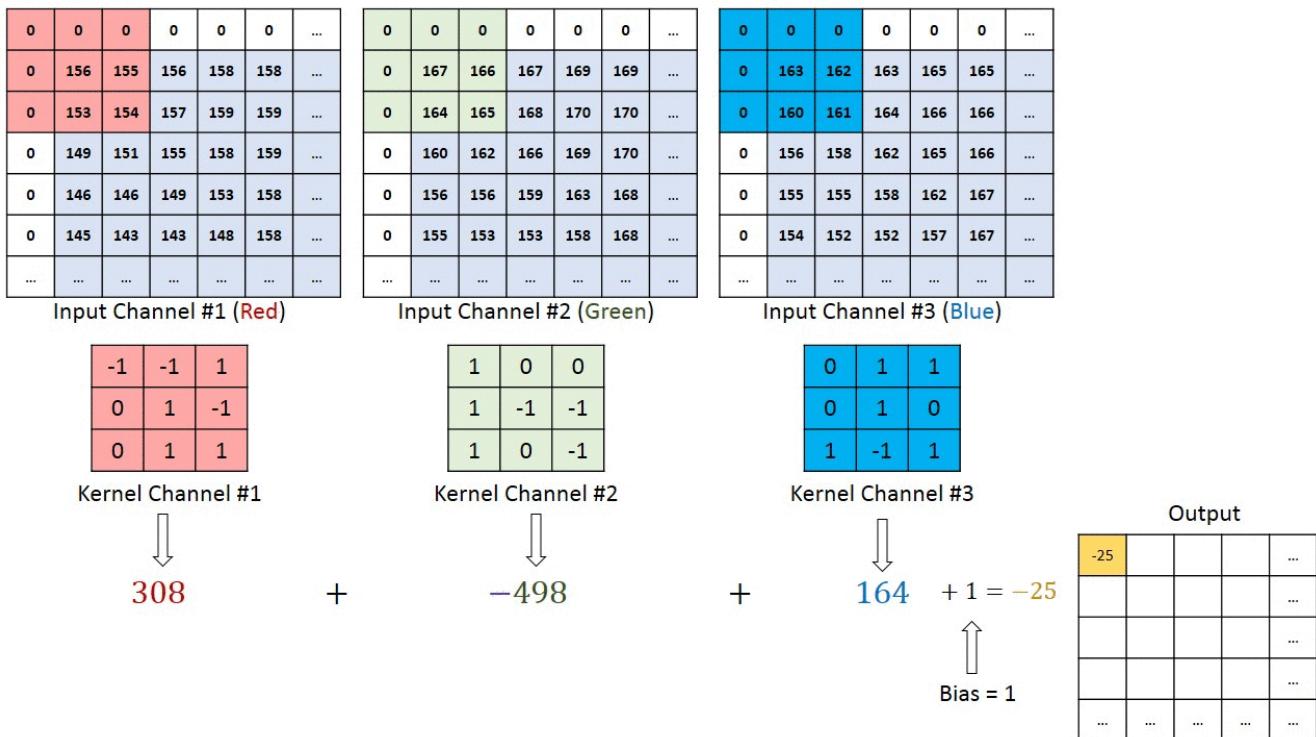


图 4.18 卷积层中的滤波器



图是对一个3通道的图片做卷积操作，卷积核的大小为 3×3 ，卷积核的数目为3，此时过滤器指的就是这三个卷积核的集合，维度是 $3 \times 3 \times 3$ ，前面的 3×3 指的是卷积核的高度 (H) 和宽度 (W)，后面的那个 3 指的是卷积核的数目 (通道数)。

上面的操作是对三个通道分别做卷积操作，然后将卷积的结果相加，最后输出一个特征图。

即：一个过滤器就对应一个特征图。

重要的事情说三遍！！！

二维卷积说的是：生成的结果矩阵是二维的

1.2.6.1 黑白照片(通道为1)计算过程

计算案例的数据：

图片：设置黑白图片($10 * 10 * 1$)，1是通道数

图片： $W_{in} * H_{out} * 1$ (1为通道数)

滤波器1和滤波器2：设置 $3 * 3 * 1$ 的张量

注：这些滤波器里面的数值其实是未知的，它是可以通过学习找出来的。假设这些滤波器里面的数值已经找出来了。

1	0	0	0	0	0	1
0	1	0	0	1	0	0
0	0	1	1	0	0	0
1	0	0	0	1	0	0
0	1	0	0	1	0	0
0	0	1	0	1	0	0

6×6 图像

1	-1	-1
-1	1	-1
-1	-1	1

滤波器 1

-1	1	-1
-1	1	-1
-1	1	-1

滤波器 2

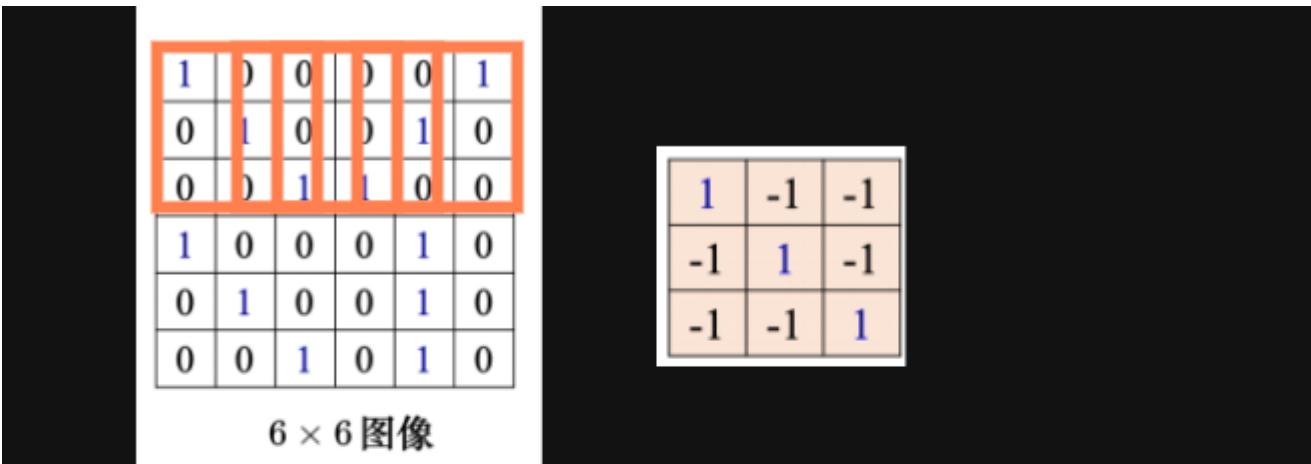
⋮

图 4.19 滤波器示例

1. 单滤波器计算演示

就是根据张量的，就是 矩阵 的 内积 运算

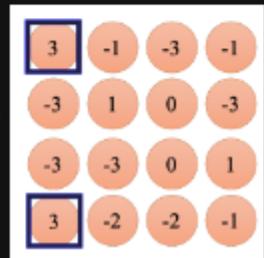
逐渐从左上角往右下角移动，最后因为 步幅是1 所以生成的矩阵为 $4 * 4 * 1$ 的结果矩阵



$$1 \times 1 + 0 \times -1 + 0 \times -1 + 1 \times 1 + 0 \times -1 + 0 \times -1 + 0 \times -1 + 1 \times 1 = 3$$

$$0 \times 1 + 0 \times -1 + 0 \times -1 + 1 \times -1 + 0 \times 1 + 0 \times -1 + 0 \times -1 + 1 \times 1 = -1$$

等等



2. 单层卷积 多个滤波器 计算演示

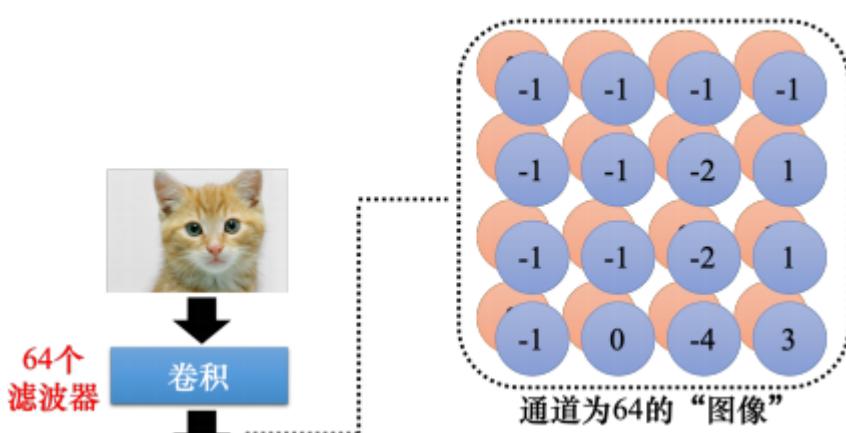
多个滤波器 是指滤波器数量，并非滤波器通道数

如下图，存在多个滤波器的时候

是 **单层卷积** 的时候

结果生成就是 **4*4*滤波器数量** 的结果矩阵，假设有 **64** 个滤波器，那么生成就是 **4 * 4 * 64** 的结果矩阵，就是下图的通道为64的图像，这只猫通过64个滤波器，生成了通道为64的“图像”

卷积核: $k_w * k_h * 1$ (1为通道数) * **NUM1(多个卷积核数量)**



注意:此时生成的结果为64的图像是作为下一层卷积的输入

输出结果: $W_{out} * H_{out} * 1$ (1为通道数) * NUM1 (多个卷积核数量)

3. 双层卷积 多个滤波器 计算演示

多个滤波器 是指滤波器数量，并非滤波器通道数

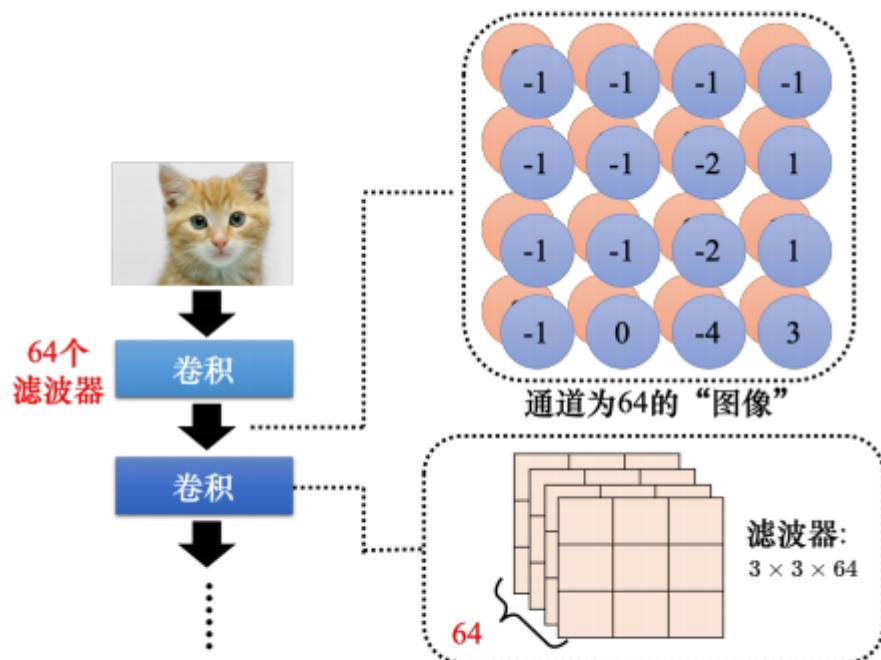


图 4.22 对图像进行卷积

如上图，因为第一层卷积生成了 $4 * 4 * 64$ 的结果图像

那么如果还需要通过下一层卷积，就需要通过 $3 * 3 * 64$ 的滤波器

注:因为第一层卷积后生成的通道(channel)是64，那么第二层卷积的滤波器通道数必须要和第一层的通道数相同

1.2.6.2 彩色照片(通道为3)计算过程

假设现在有输入图片为彩色图片，分为RGB通道三个通道

输入图片为: $5 * 5 * 3$ 大小通道的图片

1. 单滤波器计算演示

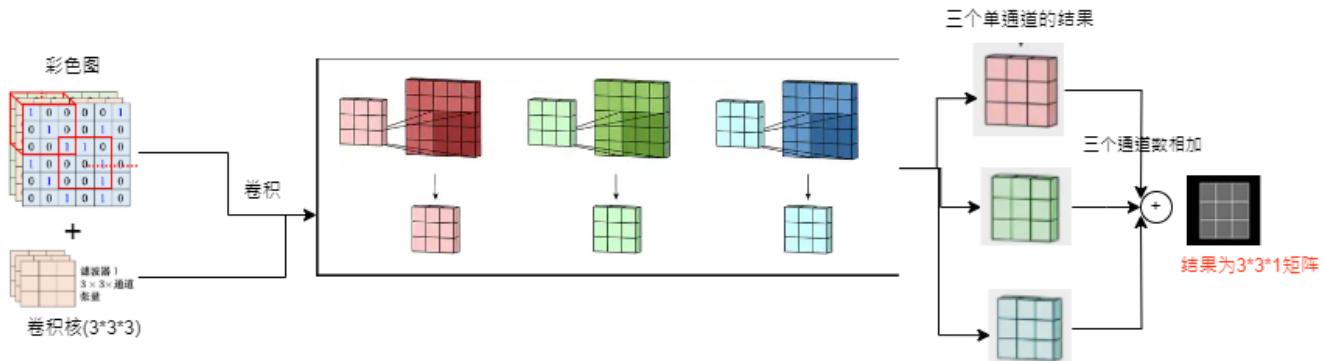
彩色图: $5 * 5 * 3$

卷积核: $3 * 3 * 3$ (3为通道数，并非个数)

最后生成的结果为: $3 * 3 * 1$ 的矩阵，因为三个单通道结果需要叠加

注：单滤波器指的是卷积核的个数，并非通道数

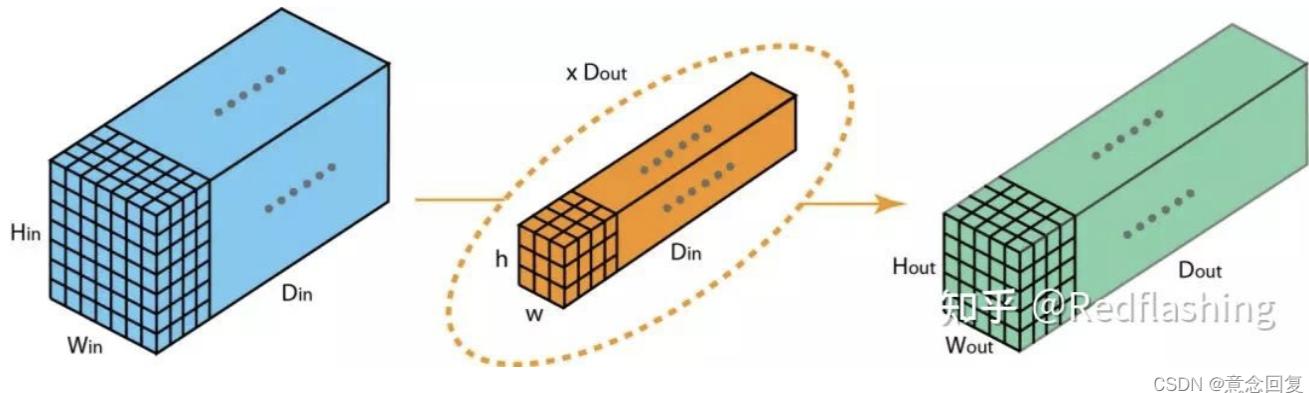
卷积核: $k_w \times k_h \times c$ (c 为通道数)



输出结果: $w_{out} \times w_{in} \times 1$ (通道数)

2. 单层卷积 多个滤波器 计算演示

多个滤波器的意思就是：滤波器的数量是多个的，和通道数无关



进一步地，我们就能非常轻易地理解如何在不同深度的层（Layer）进行转换。假设输入层有 X_{in} 个通道，而输出层需要得到 D_{out} 个通道。只需要将 D_{out} 个过滤器对输入层进行处理，而每一个过滤器有 X_{in} 个卷积核。每个过滤器提供一个输出通道。完成该过程将得到 D_{out} 个通道组成输出层。

1.2.6.3 总结

输入图片的规格:

$$W_{in} * H_{in} * D_{in}$$

超参数：

过滤器个数： k

过滤器中卷积核维度： $w * h$

滑动步长(*Stride*)： S

填充值(*Padding*)： p

输出层： $W_{\text{out}} * H_{\text{out}} * D_{\text{out}}$

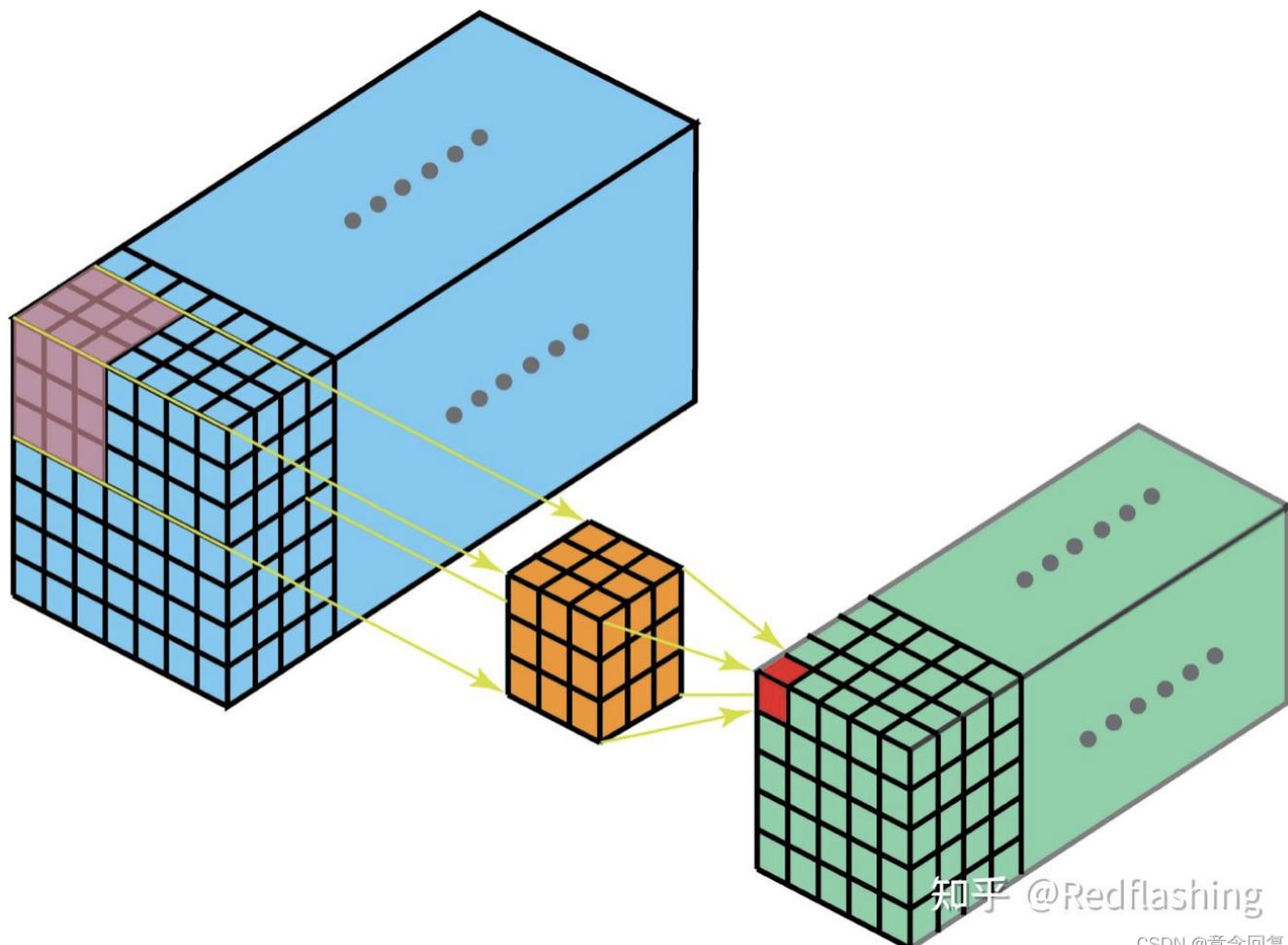
输出数据的规格

$$\begin{cases} W_{\text{out}} = (W_{\text{in}} + 2p - w) / s + 1, \\ H_{\text{out}} = (H_{\text{in}} + 2p - h) / s + 1, \\ D_{\text{out}} = k \end{cases}$$

1.2.7 三维卷积过程(3D Convolutional)

在上一个插图中，可以看出，这实际上是在完成3D卷积。

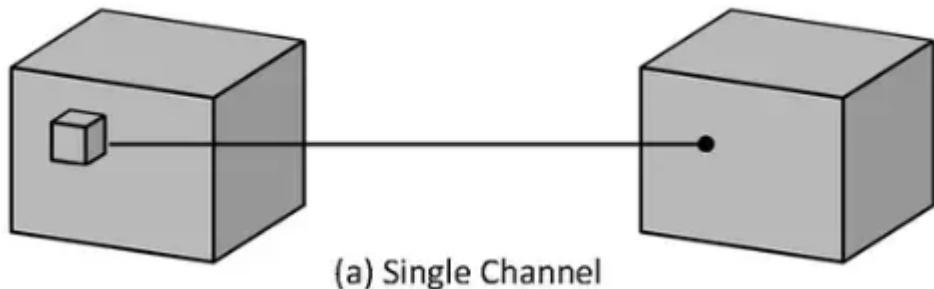
但通常意义上，仍然称之为深度学习 的2D卷积。因为将滤波器深度和输入层深度相同，3D滤波器仅在2个维度上移动（例如图像的高度和宽度），得到的结果为单通道。



知乎 @Redflashing

CSDN @意念回复

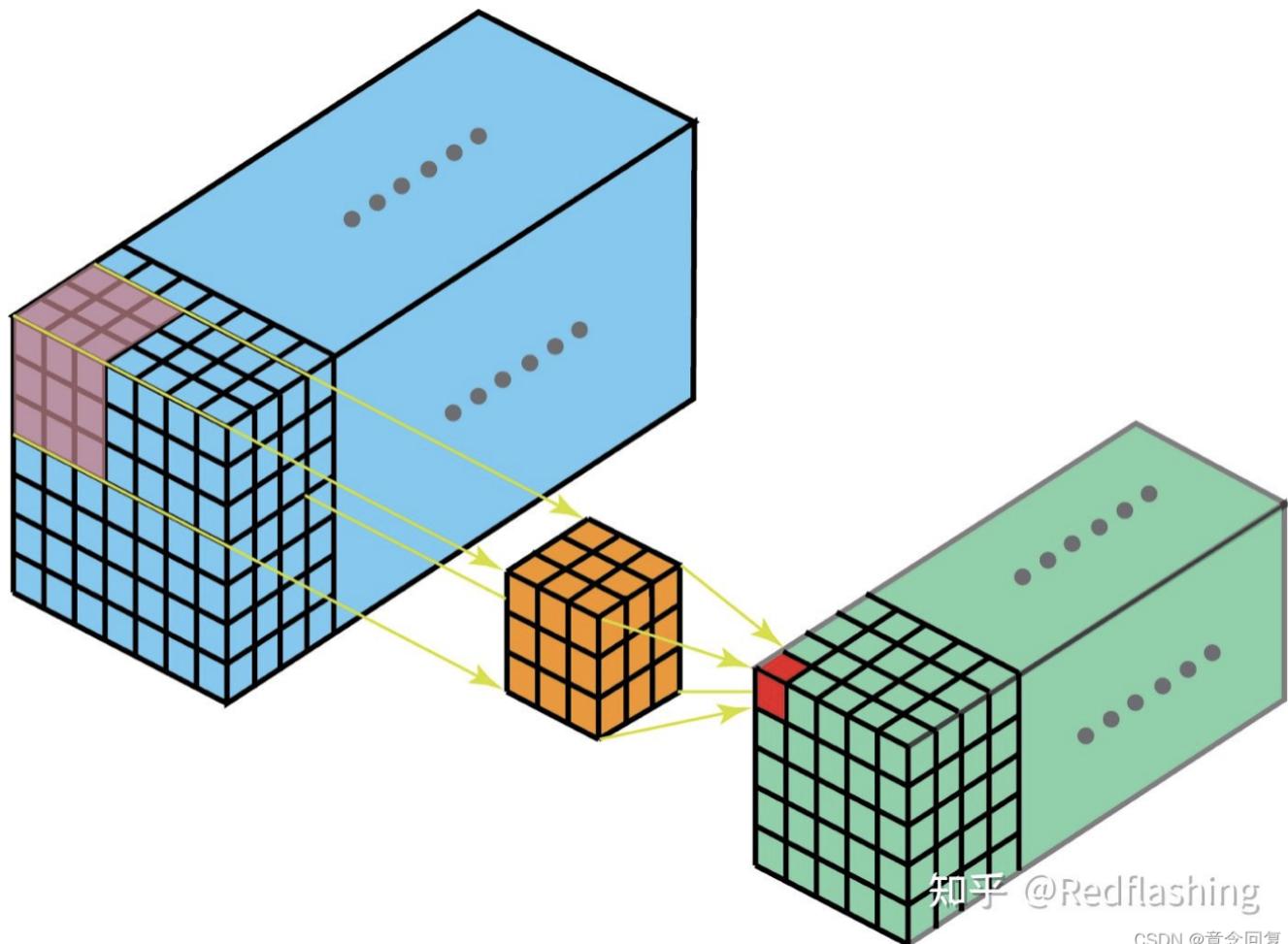
1.2.7.1 单通道计算过程



输入数据: depth * height * width * 1 (1为通道数)

1. 单滤波器计算演示

卷积核: $k_d * k_h * k_w * 1$ (1为通道数), 且 这时候只有一个卷积核 , 也就是单滤波器的单是卷积核的数量 , 并非通道数



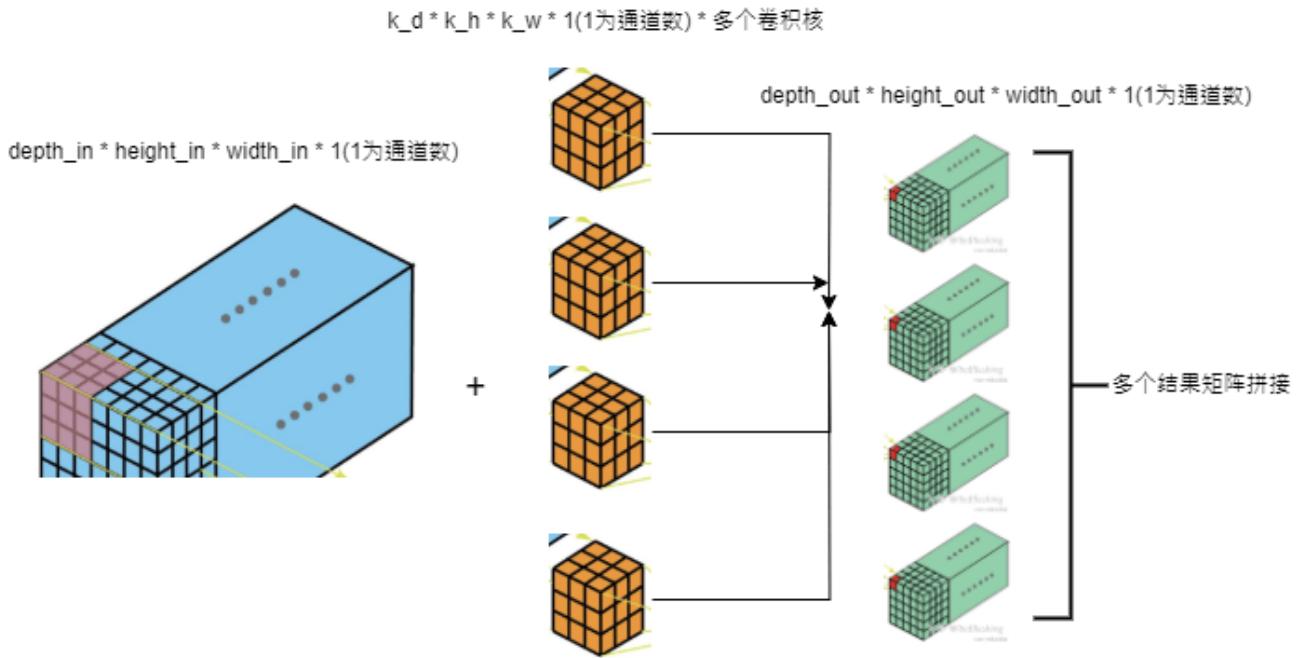
知乎 @Redflashing

CSDN @意念回复

输出结果: depth_out * height_out * width_out * 1(1为通道数)

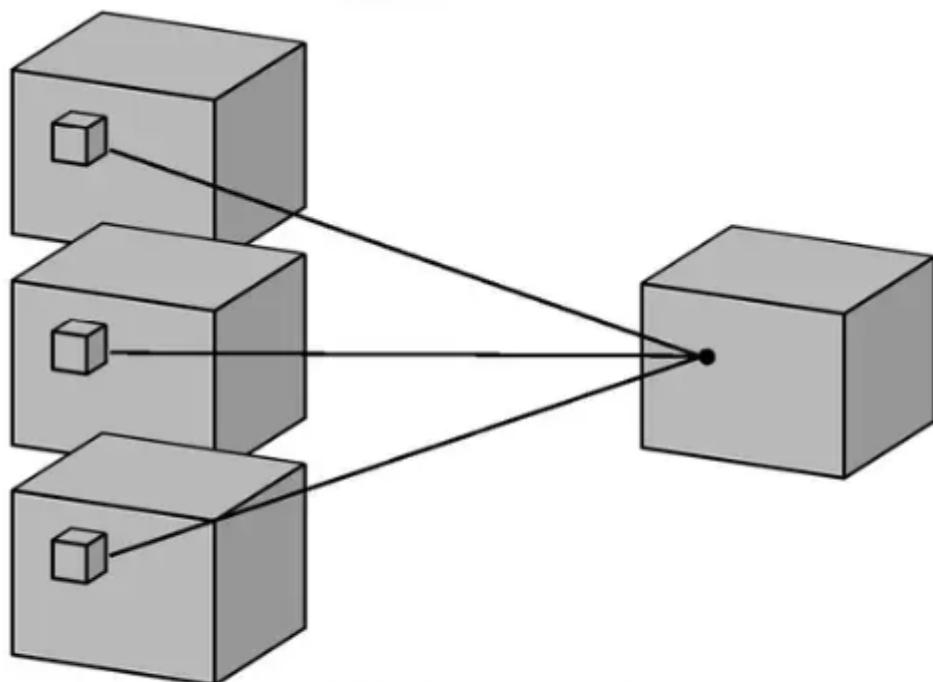
2. 单层卷积 多滤波器 计算演示

卷积核: $k_d * k_h * k_w * 1$ (**1为通道数**) * 多个卷积核



输出结果: $depth_out * height_out * width_out * 1$ (**1为通道数**) * **多个卷积核个数**

1.2.7.2 多通道计算过程

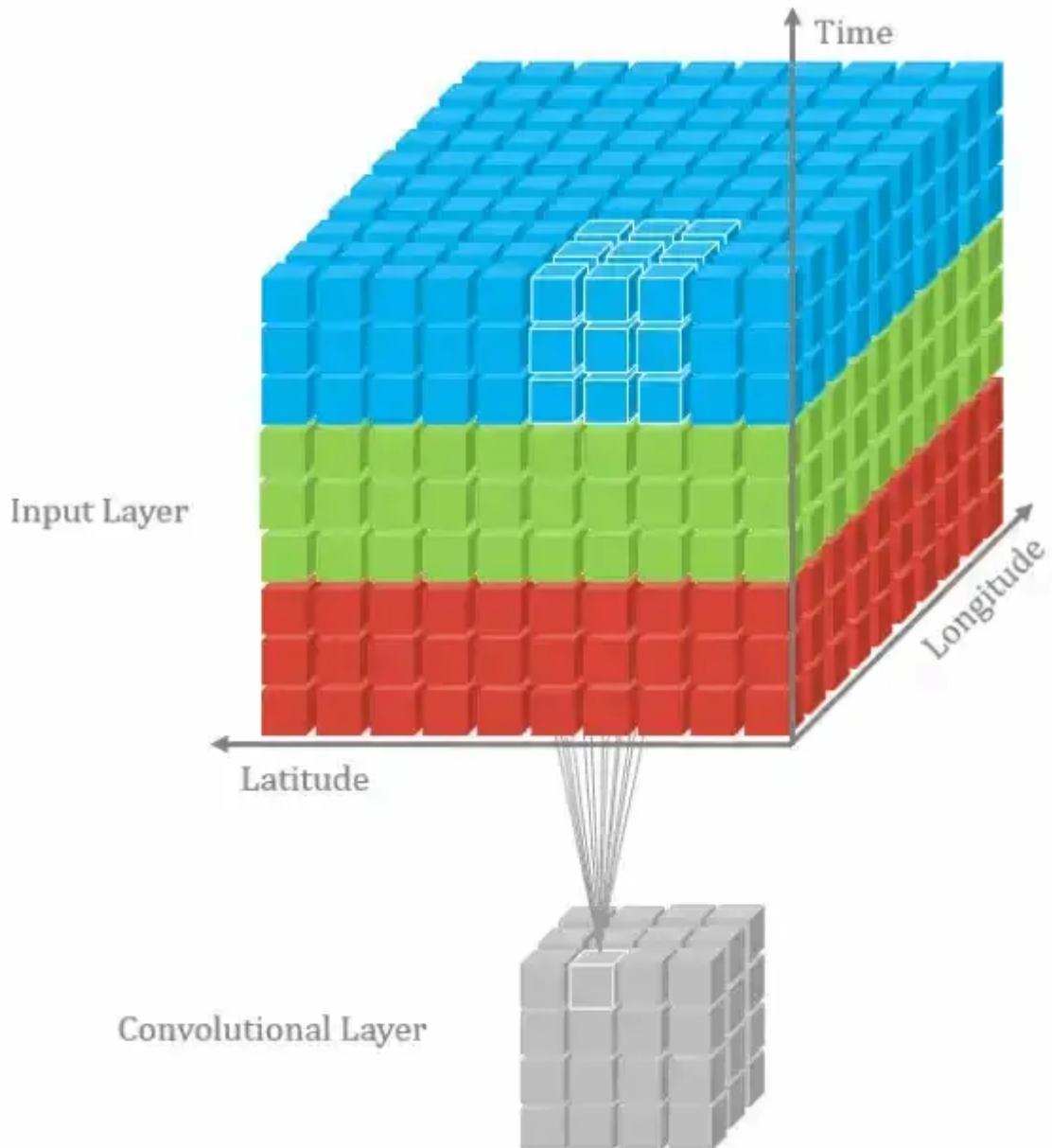


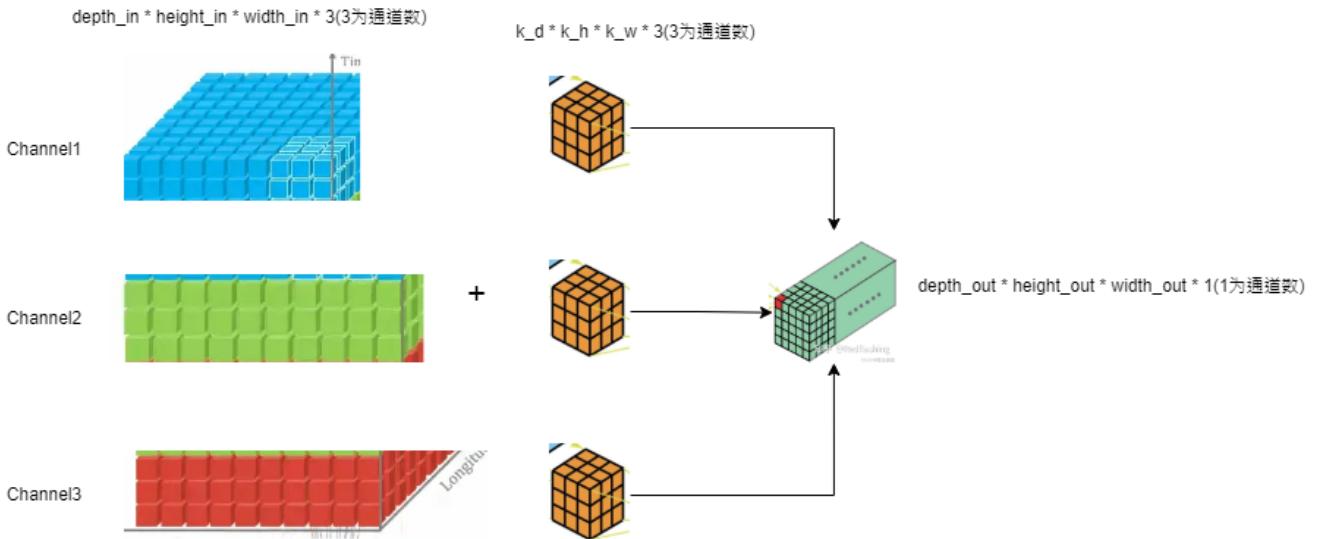
知乎 (

输入数据: $depth * height * width * 3$ (**3为通道数**)

1. 单滤波器计算演示

卷积核: $k_d * k_h * k_w * 3$ (3 为通道数)

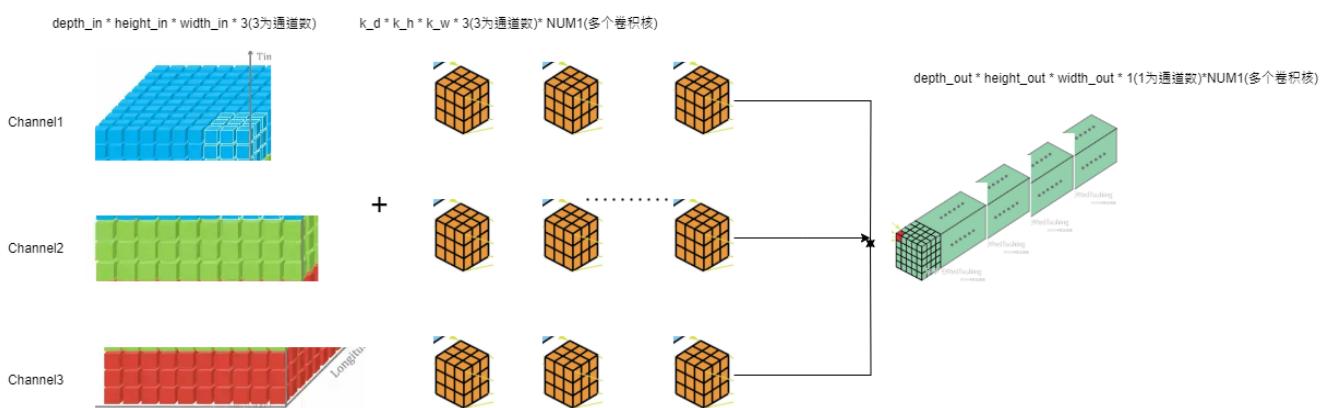




输出结果: $depth_{out} \times height_{out} \times width_{out} \times 1$ (1为通道数)

2. 单层卷积多滤波器计算演示

卷积核: $k_d \times k_h \times k_w \times 3$ (3为通道数) * NUM1(多个卷积核数量)



输出结果: $depth_{out} \times height_{out} \times width_{out} \times 1$ (1为通道数) * NUM1(多个卷积核数量)

1.2.7.3 总结

输入层:

$$W_{in} * H_{in} * D_{in} * C_{in}$$

超参数:

超参数 :

过滤器个数 : k

过滤器中卷积核维度 : $w * h * d$

滑动步长 (Stride) : S

填充值 (Padding) : p

输出层 : $W_{out} * H_{out} * D_{out} * C_{out}$

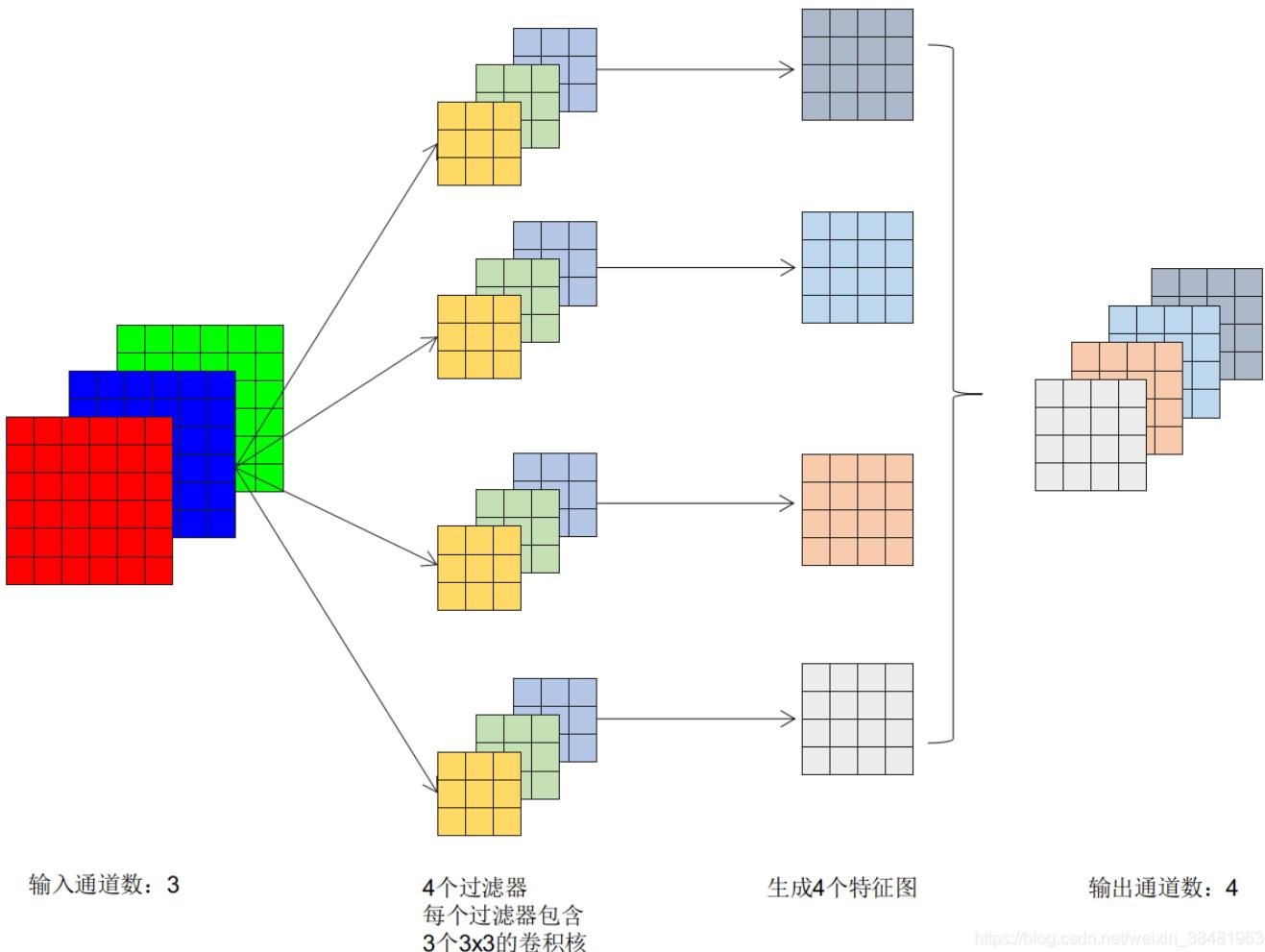
输出层规格

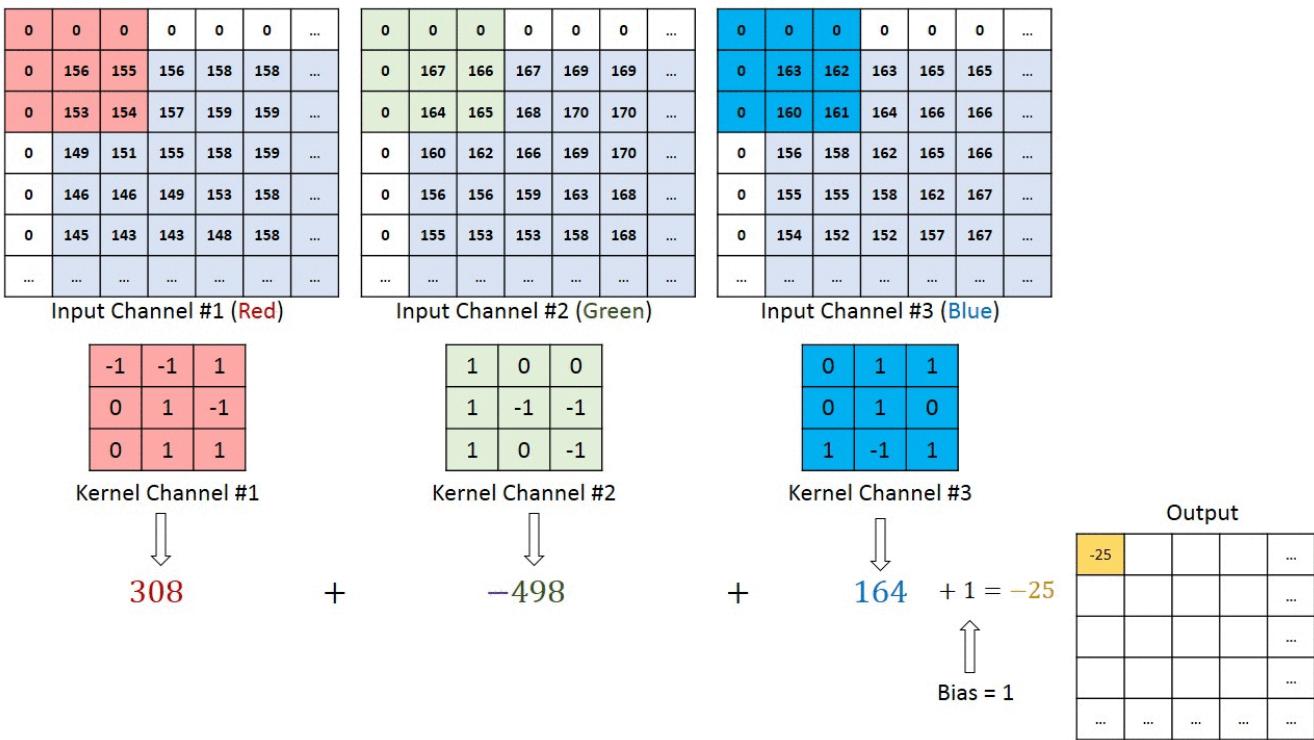
$$\begin{cases} W_{\text{out}} = (W_{\text{in}} + 2p - w) / s + 1 \\ H_{\text{out}} = (H_{\text{in}} + 2p - h) / s + 1, \\ D_{\text{out}} = (D_{\text{in}} + 2p - d) / s + 1, \\ C_{\text{out}} = k \end{cases}$$

调参数量:

$$(w * h * d + 1) * k$$

- 对于最初输入图片样本的通道数 `in_channels` 取决于图片的类型，如果是彩色的，即RGB类型，这时候通道数固定为3，如果是灰色的，通道数为1。
- 卷积完成之后，`输出的通道数 out_channels` 取决于过滤器的数量。从这个方向理解，这里的 `out_channels` 设置的就是过滤器的数目。
- 对于 第二层或者更多层的卷积，此时的 `in_channels` 就是上一层的 `out_channels`，`out_channels` 还是取决于过滤器数目。





1.2.8 滤波器守备范围探讨

Q: 如果滤波器的大小一直设 3×3 , 会不会让网络没有办法看比较大范围的模式呢?

A: 不会。如图 4.23 所示, 如果在第 2 层卷积层滤波器的大小一样设 3×3 , 当我们看第 1 个卷积层输出的特征映射的 3×3 的范围的时候, 在原来的图像上是考虑了一个 5×5 的范围。虽然滤波器只有 3×3 , 但它在图像上考虑的范围是比较大的是 5×5 。因此网络叠得越深, 同样是 3×3 的大小的滤波器, 它看的范围就会越来越大。所以网络够深, 不用怕检测不到比较大的模式。

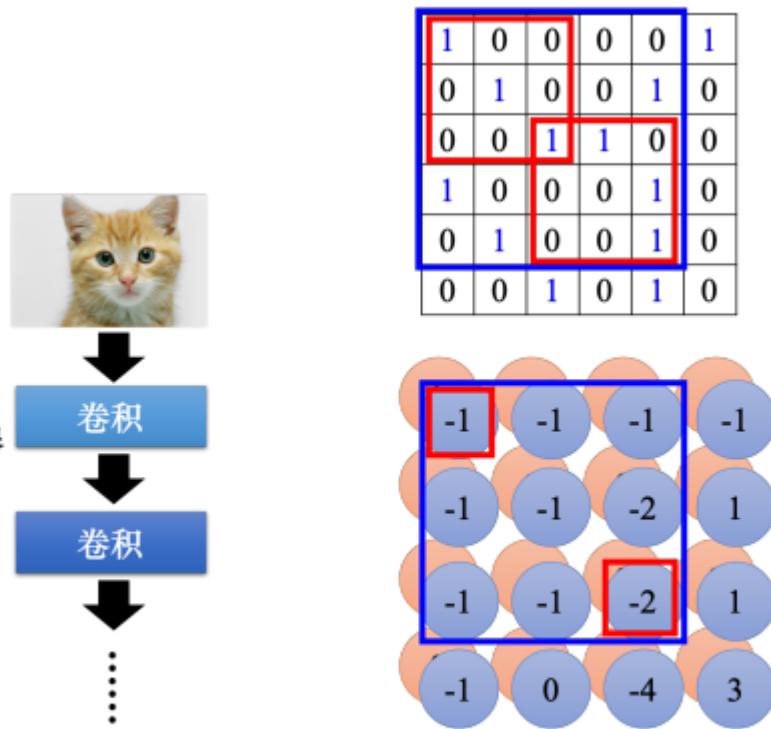


图 4.23 网络越深，可以检测的模式越大

1.2.9 Pooling(池化层、汇聚)

汇聚最主要的作用是减少运算量，通过下采样把图像变小，从而减少运算量

1. invariance(**不变性**)，这种不变性包括translation(平移), rotation(旋转), scale(尺度)
2. 保留主要的特征同时减少参数(降维，效果类似PCA)和计算量，防止过拟合，提高模型泛化能力

参数:

超参数 :

滑动步长 (*Stride*) : s

规格 *size* : f

1.2.9.1 Max Pooling(最大化池化层)

如下图

池化层的参数是 $f = 2, s = 2$

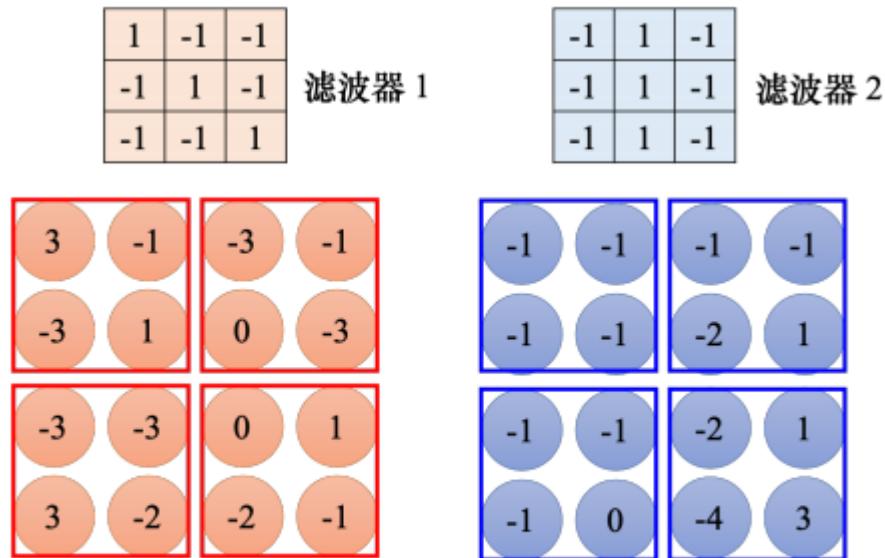


图 4.27 最大汇聚示例

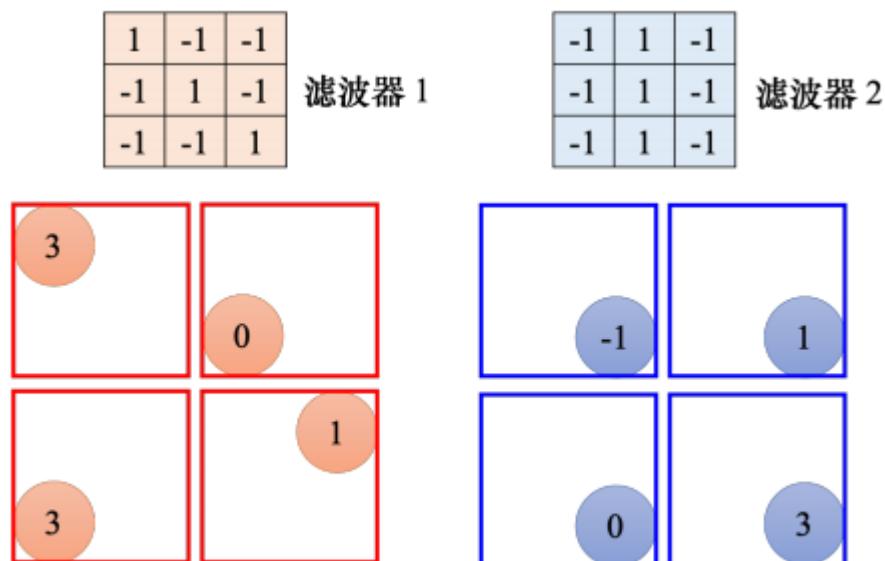
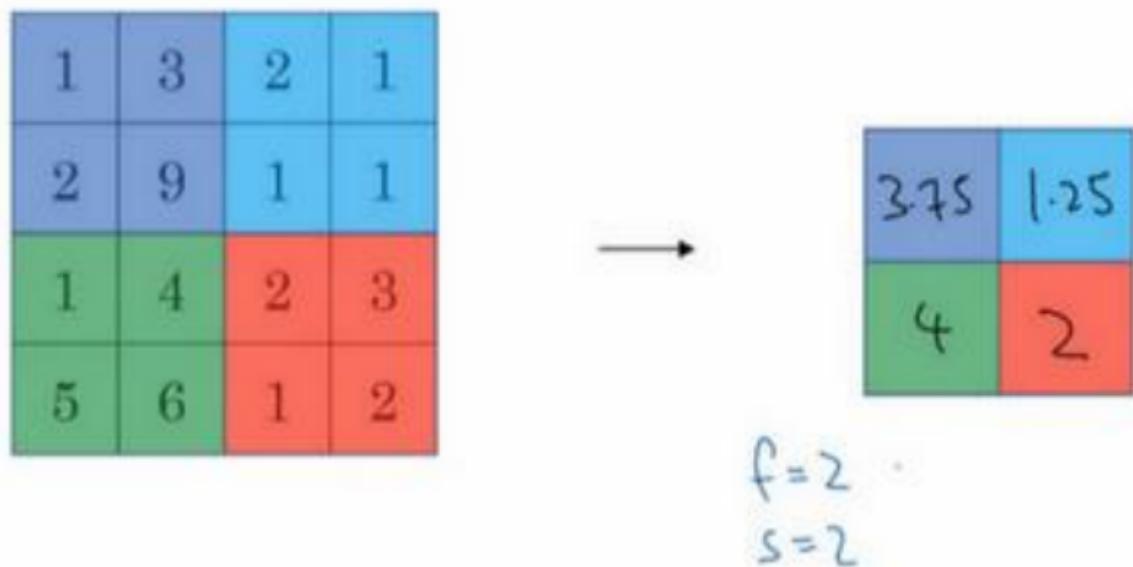


图 4.28 最大汇聚结果

1.2.9.2 Average Pooling(平均化池化层)

如下图

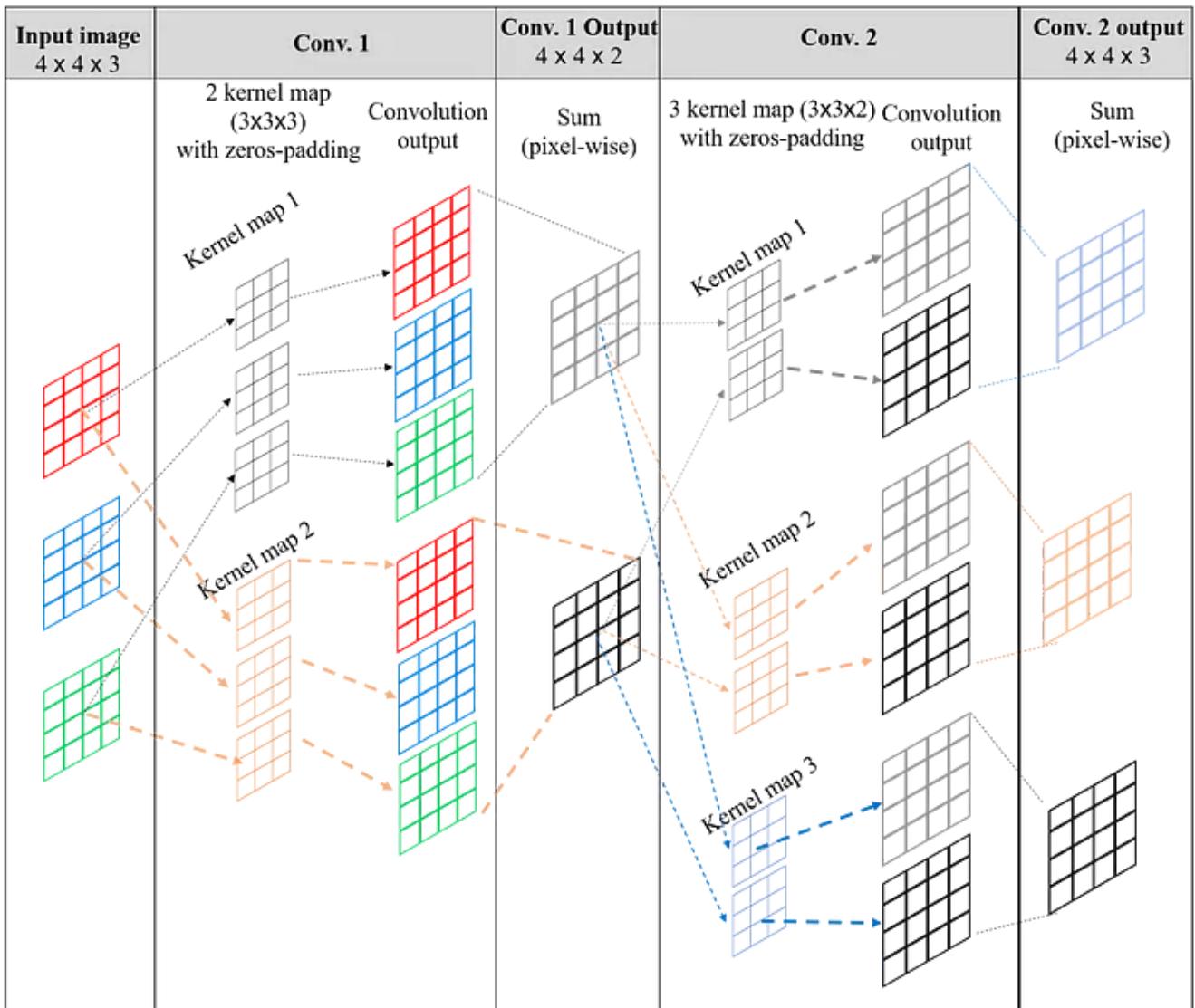
池化层的规格是 $f = 2, s = 2$



1.2.9.3 参考链接

CNN网络的pooling层有什么用？

1.2.10 Flatten(扁平化、拉直)



但是2維的tensor是沒辦法被神經元接收的，所以我們必須把2維的tensor拉平，拉成長~~~長的一條直線，變成一維的矩陣(就是一個數列)。

1.2.11 FC(全连接层)

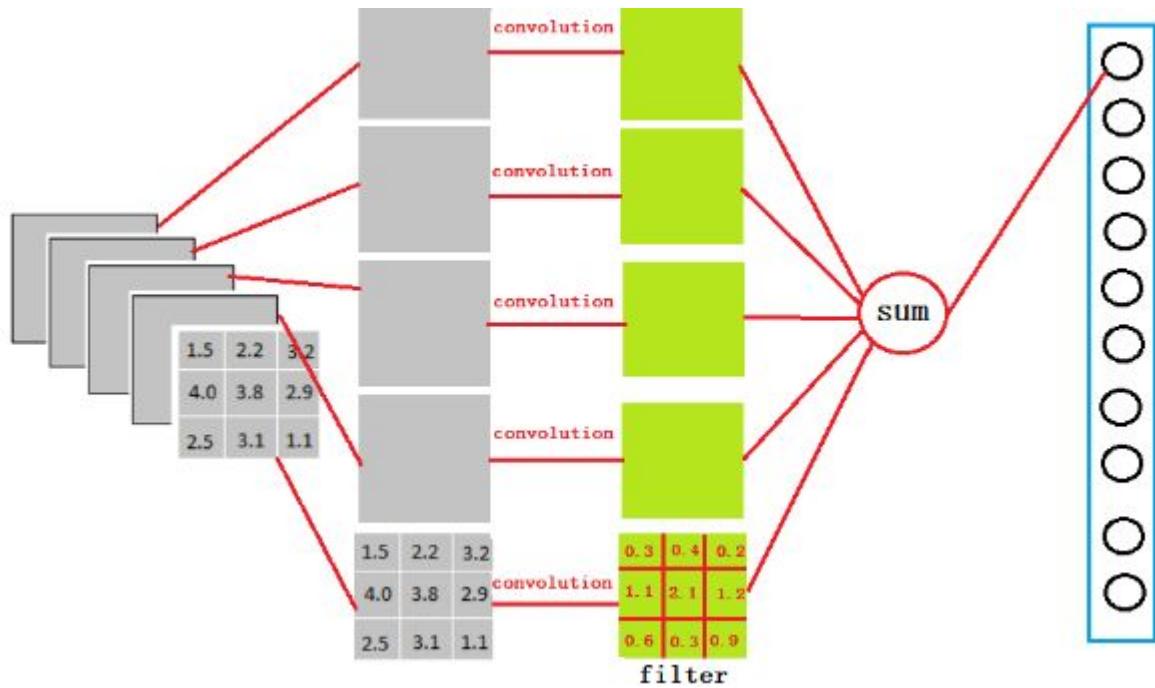
全连接层 Fully Connected Layer 一般位于整个卷积神经网络的最后，负责将卷积输出的二维特征图转化成一维的一个向量，由此实现了端到端的学习过程（即：输入一张图像或一段语音，输出一个向量或信息）。

全连接层的每一个结点都与上一层的所有结点相连因而称之为全连接层。由于其全相连的特性，一般全连接层的参数也是最多的

1.2.11.1 作用

全连接层的主要作用就是将前层（卷积、池化等层）计算得到的特征空间映射样本标记空间。

简单的说就是将特征表示整合成一个值，其优点在于减少特征位置对于分类结果的影响，提高了整个网络的鲁棒性。



1.2.11.2 FC(全连接层)参考文章链接

[CNN入门讲解：什么是全连接层（Fully Connected Layer）？](#)

1.2.9 CNN参考文章链接

[卷积神经网络中二维卷积核与三维卷积核有什么区别](#)

1.2.9 总结

Convolutional Layer

<i>Neuron Version Story</i>	<i>Filter Version Story</i>
Each neuron only considers a receptive field.	There are a set of filters detecting small patterns.
The neurons with different receptive fields share the parameters.	Each filter convolves over the input image.

They are the same story.

两个版本的故事

每个神经元只负责一个感受野 = 有很多滤波器负责检测小的特征

神经元共享参数 = 滤波器卷积过程

番外-RNN(循环神经网络模型 Recurrent Neural Network)

Slot filling介绍

先介绍什么是Slot filling:

Example 1: User Input: "I want to book a flight from New York to London on September 5th."

Slots to be filled:

- Departure City: New York
- Destination City: London
- Date: September 5th

Filled Slots:

Slot	Value
Departure City	New York
Destination City	London
Date	September 5th

Example 2: Restaurant Reservation

User Input: "Book a table for 4 people at a Chinese restaurant in downtown at 7 PM."

Slots to be filled:

- **Cuisine:** Chinese
- **Location:** Downtown
- **Number of People:** 4
- **Time:** 7 PM

Filled Slots:

Slot	Value
Cuisine	Chinese
Location	Downtown
Number of People	4
Time	7 PM

Example 3: Hotel Booking

User Input: "I need a hotel room in Paris from June 10th to June 15th."

Slots to be filled:

- **Location:** Paris
- **Check-in Date:** June 10th
- **Check-out Date:** June 15th

Filled Slots:

Slot	Value
Location	Paris
Check-in Date	June 10th
Check-out Date	June 15th

Example 4: Movie Booking

User Input: "I want two tickets for the 8 PM showing of Oppenheimer tomorrow."

Slots to be filled:

- **Movie Title:** Oppenheimer
- **Showtime:** 8 PM
- **Number of Tickets:** 2
- **Date:** Tomorrow

Filled Slots:

Slot	Value
Movie Title	Oppenheimer
Showtime	8 PM
Number of Tickets	2
Date	Tomorrow

总结:

slot filling 就是给次打上label, 比如I want to book a flight from New York to London on September 5th
需要打上"Depature City"的slot是New York, "destination City"的slot是London, "Date"的Slot是September 5th

One-Hot(独热编码)

问题的产生:

回归, 分类, 聚类或者NLP等问题的时候, 通常很多数据都是无法直接利用的。例如一个学生信息数据集中样本有三种类别, 每个类别分别对应不同种类的标签: “性别”(男、女)、“班级”(1班、2班、3班)、“年级”(一年级、二年级、三年级、四年级)。

One-Hot编码, 又称为一位有效编码, 主要是采用N位状态寄存器来对N个状态进行编码, 每个状态都由他独立的寄存器位, 并且在任意时候只有一位有效。

性别:[“男”,“女”]

只有两个特征, 所以N为2, 下面同理。

男=>10

女=>01

班级:[“1班”,“2班”,“3班”]

1班=>100

2班=>010

3班=>001

年纪:[“一年级”,“二年级”,“三年级”,“四年级”]

一年级=>1000

二年级=>0100

三年级=>0010

四年级=>0001

所以如果一个样本为[“男”,“2班”,“四年级”的时候,完整的特征数字化的结果为:

[1,0,0,1,0,0,0,0,1]

到这里我们就实现了对数据集样本的编码,就可以顺利进行后续的回归、聚类或者其他操作。

优点:

能够处理非连续型数值特征,也就是离散值

缺点:

(1)如果原本的标签编码是有序的,那one hot编码就不合适了——会丢失顺序信息。

(2)如果特征的特征值数目特别多,特征向量就会非常大,且非常稀疏。

RNN-1.需要解决的问题

场景一:

假设用户1说的是:在6月1号抵达上海

用户2说的是:在6月1号离开上海

如果采用前馈神经网络,假设把单词表示为向量,把这个向量丢到前馈神经网络里面去,在该任务里面,输出是一个概率分布,该概率分布代表着输入单词属于每一个槽的概率,比如“上海”属于目的地的概率和“上海”属于出发地的概率

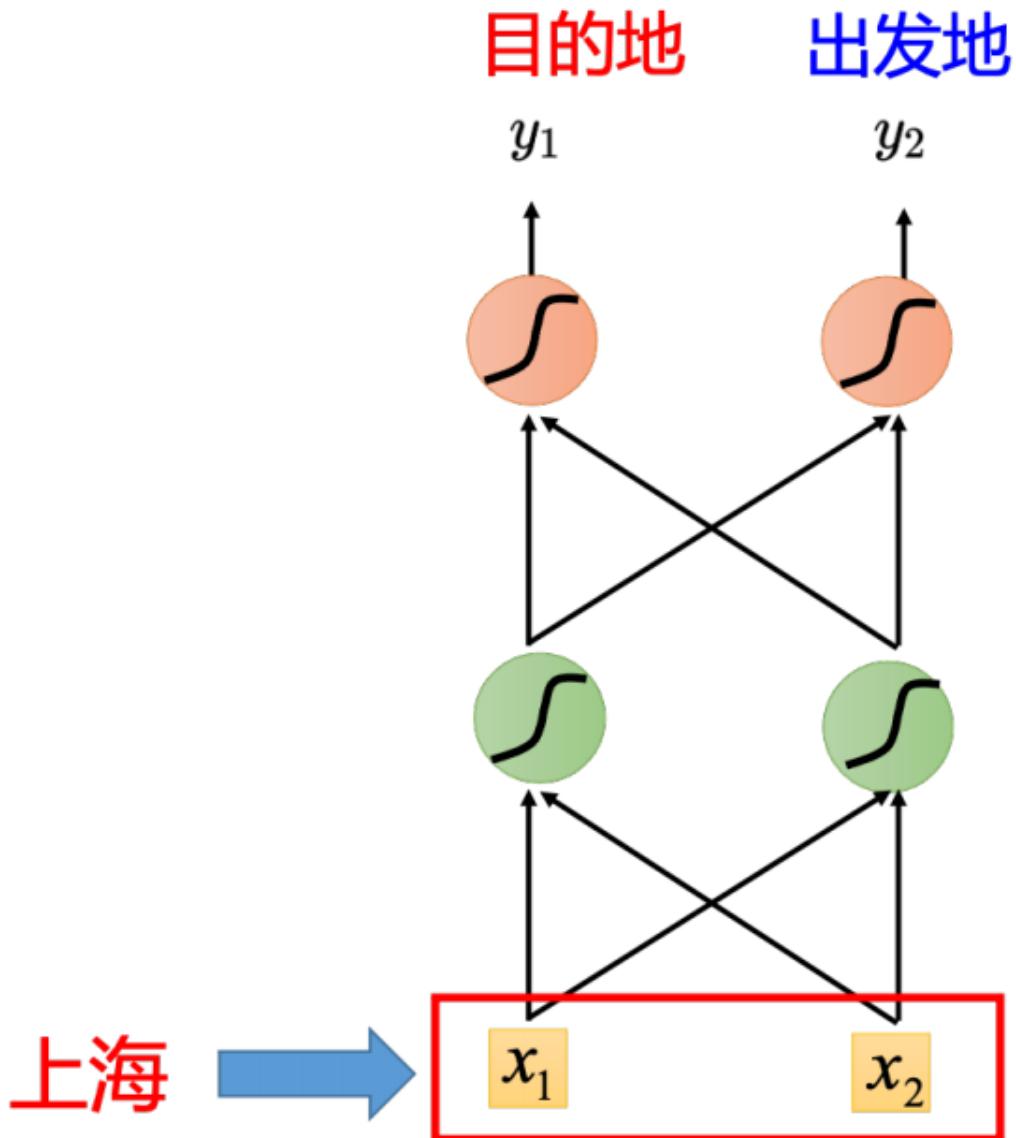


图 5.4 使用前馈神经网络预测概率分布

也就是说,用户1说:"在 6 月 1 号抵达上海"。用户2说:"在 6 月 1 号离开上海", 这个时候 上海 就变成了出发地

but, 对于神经网络来说, 输出要么让目的地概率最高, 要么让出发地概率高, 不能一会出发地概率高, 一会目的地概率高

如果神经网络有记忆力的, 它记得它看过“抵达”, 在看到“上海”之前; 或者它记得它已经看过“离开”, 在看到“上海”之前。通过记忆力, 它可以根据上下文产生不同的输出。如果让神经网络是有记忆力, 其就可以解决输入不同的单词, 输出不同的问题

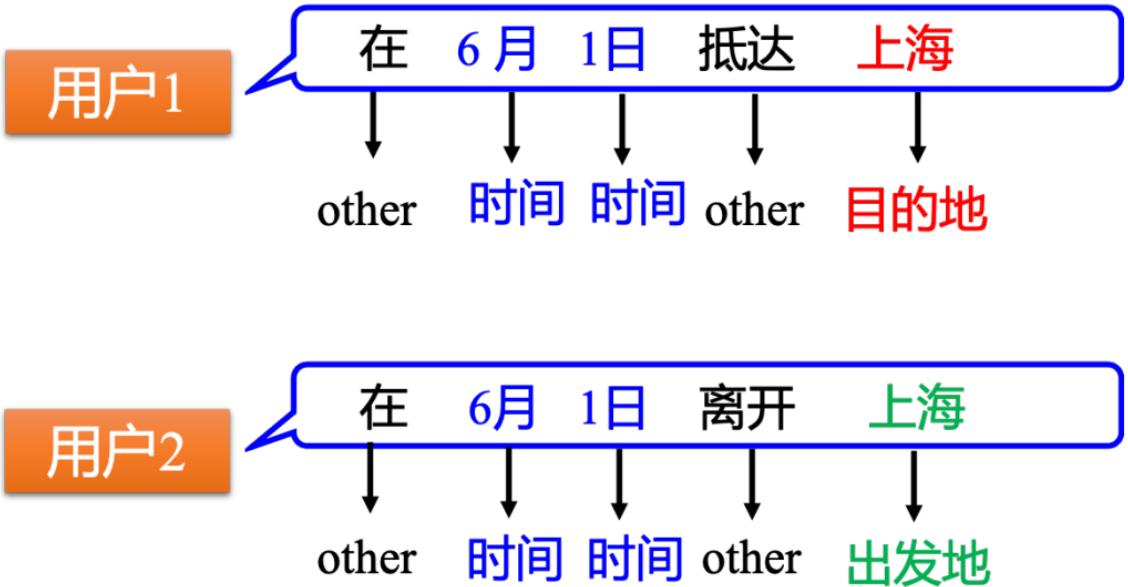


图 5.5 前馈神经网络的问题

场景二：

我们先来看一个NLP很常见的问题，**命名实体识别**，举个例子，现在有两句话：

第一句话：I like eating apple! （我喜欢吃苹果！）

第二句话：The Apple is a great company! （苹果真是一家很棒的公司！）

现在的任务是要给apple打Label，我们都应该知道第一个apple是一种水果，第二个apple是苹果公司，假设我们现在有大量的已经标记好的数据以供训练模型，当我们使用全连接的神经网络时，我们做法是把apple这个单词的**特征向量**输入到我们的模型中（如下图），在输出结果时，让我们的label里，正确的label概率最大，来训练模型，但我们的**语料库**中，有的apple的label是水果，有的label是公司，这将导致，模型在训练的过程中，预测的准确程度，取决于训练集中哪个label多一些，这样的模型对于我们来说完全没有作用。问题就出在了我们没有结合上下文去训练模型，而是单独的在训练apple这个单词的label，这也是全连接神经网络模型所不能做到的，于是就有了我们的循环神经网络。

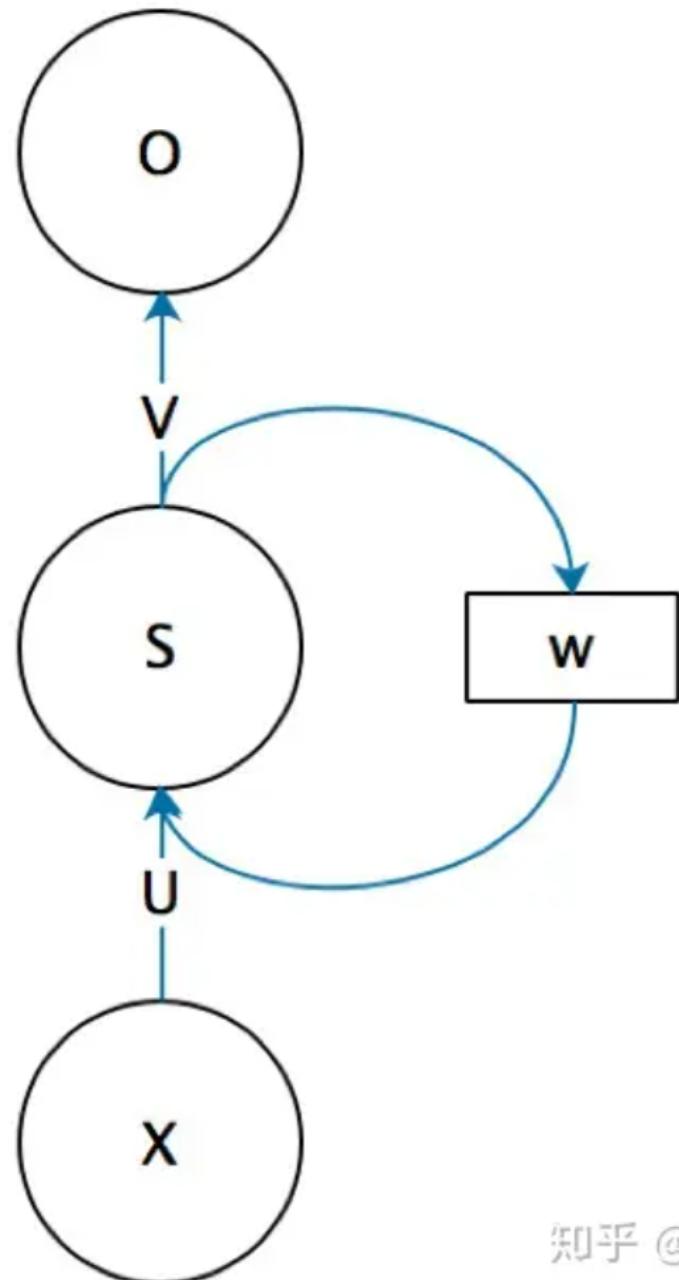
RNN-2 基础RNN的结构

RNN对具有序列特性的数据非常有效，它能挖掘数据中的时序信息以及语义信息

输出层

隐藏层

输入层



知乎 @韦伟

(RNN结构)

因为当前时刻的隐状态使用与上一时刻隐状态相同的定义，所以隐状态的计算是循环的（recurrent），基于循环计算的隐状态神经网络被称为循环神经网络。

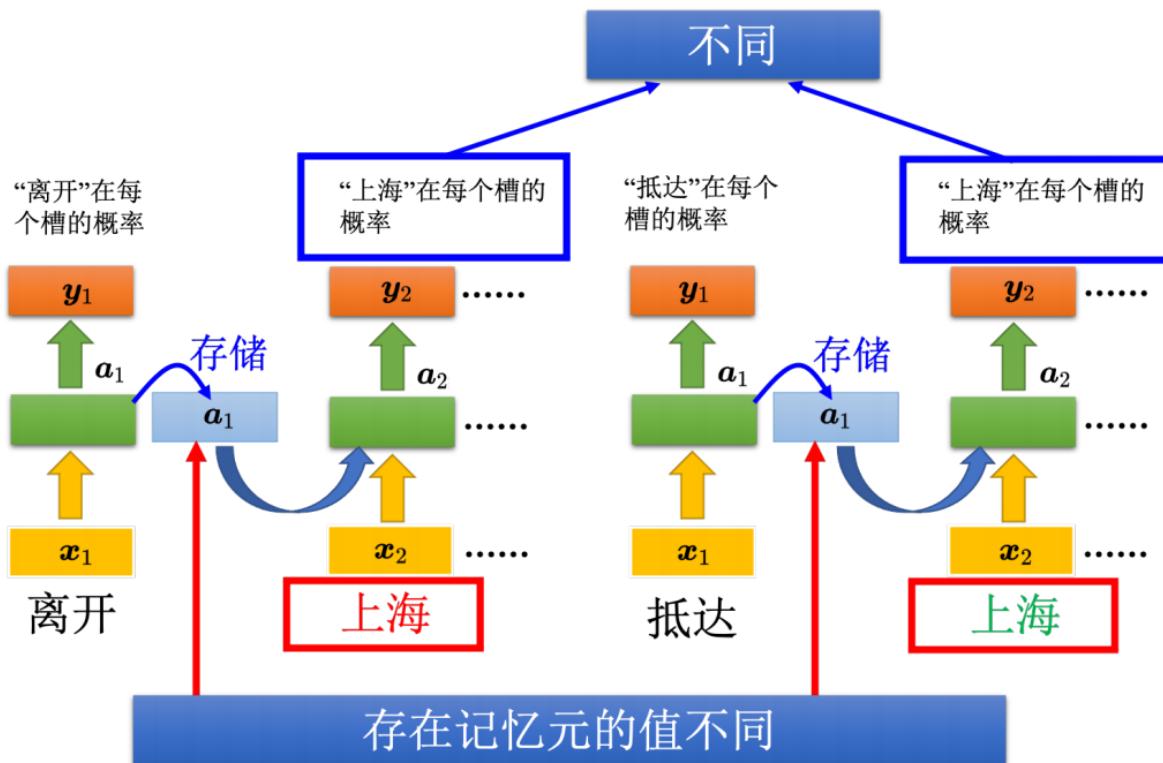


图 5.8 输入相同，输出不同示例

RNN-3 基础RNN的梯度爆炸和梯度消失现象

RNN结构是存在梯度爆炸和梯度消失现象的，证明需要从RNN结构的反向传播求偏导来看

粗略证明：

RNN-3 Elman网络、Jordan网络和双向循环神经网络(基础RNN网络的变形)

只需了解

RNN-4 LSTM(Long Short-Term Memory network)长短期记忆网络

存在问题：

1. 基础RNN网络是短时记忆的，假设很长的序列句子输入，那么基础RNN不会记得句子开头的内容详细内容。
2. 且RNN网络存在梯度爆炸和梯度消失现象

LSTM解决了梯度爆炸和梯度消失问题

番外-Seq2Seq(序列到序列模型)

番外-Encoder-Decoder经典模型架构

Encoder-Decoder是一个模型结构，是一类算法统称；

编码（encode）由一个编码器将输入序列转化成一个固定维度的稠密向量，解码（decode）阶段将这个激活状态生成目标译文

Encoder(编码器)

2. 自注意力机制(self-attention model)

图像处理上，输入看作一个向量(经过CONV 和 Flatten)之后生成向量

回归问题：就是输出一个标量

分类问题：就是输出一个类别

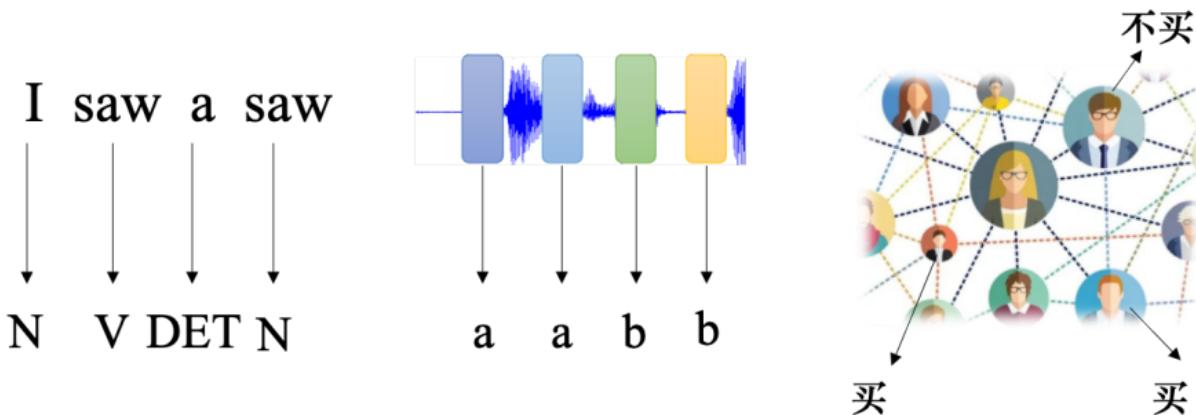
2.1 输入是向量序列的情况(Seq2Seq模型)

2.1.1 输入与输出数量相同

模型输入是一组向量(可以是文字、语音、图)，输入的向量要和输出的向量数量相同

1. 输入4个向量，输出4个标签-----分类问题
2. 输入4个向量，输出4个标量-----回归问题
3. 如下图

- 找出 "I saw a saw" 每个单词的词性
- 一段声音都有一串向量，找出每串向量是从属于音标
- 每个结点有什么样的特性，比如某个人会不会买某个商品



2.1.2 输入是一个序列，输出是一个标签

第二种可能的输出如图 6.8 所示，整个序列只需要输出一个标签就好。



图 6.8 类似 2：输入是一个序列，输出是一个标签

应用场景：

情感分析，根据贴文，让机器看贴文，通过情感分析出这篇文章是消极还是积极的

分子亲水性分析：给定一个分子，预测该分子的亲水性

2.1.3 输入是序列，输出也是序列



图 6.10 类型 3：序列到序列任务

我们不知道应该输出多少个标签，机器要自己决定输出多少个标签。如图 6.10 所示，输入是 N 个向量，输出可能是 N' 个标签。

翻译就是满足此类工作

2.2 自注意力的运作原理

存在问题:

sequence labeling(序列标注)

1. 如果给 "I saw a saw" 每个词判断词性，第一直觉是用 FC(全连接层) 进行，但是全连接层输入的东西一样，输出的东西必然是一样的，也就是 saw 这个单词无法会混淆
2. 也就是每个单词需要解决通过 "预先" 知道整个序列的情况下进行输入

2.2.1 自注意力模型的运作方式

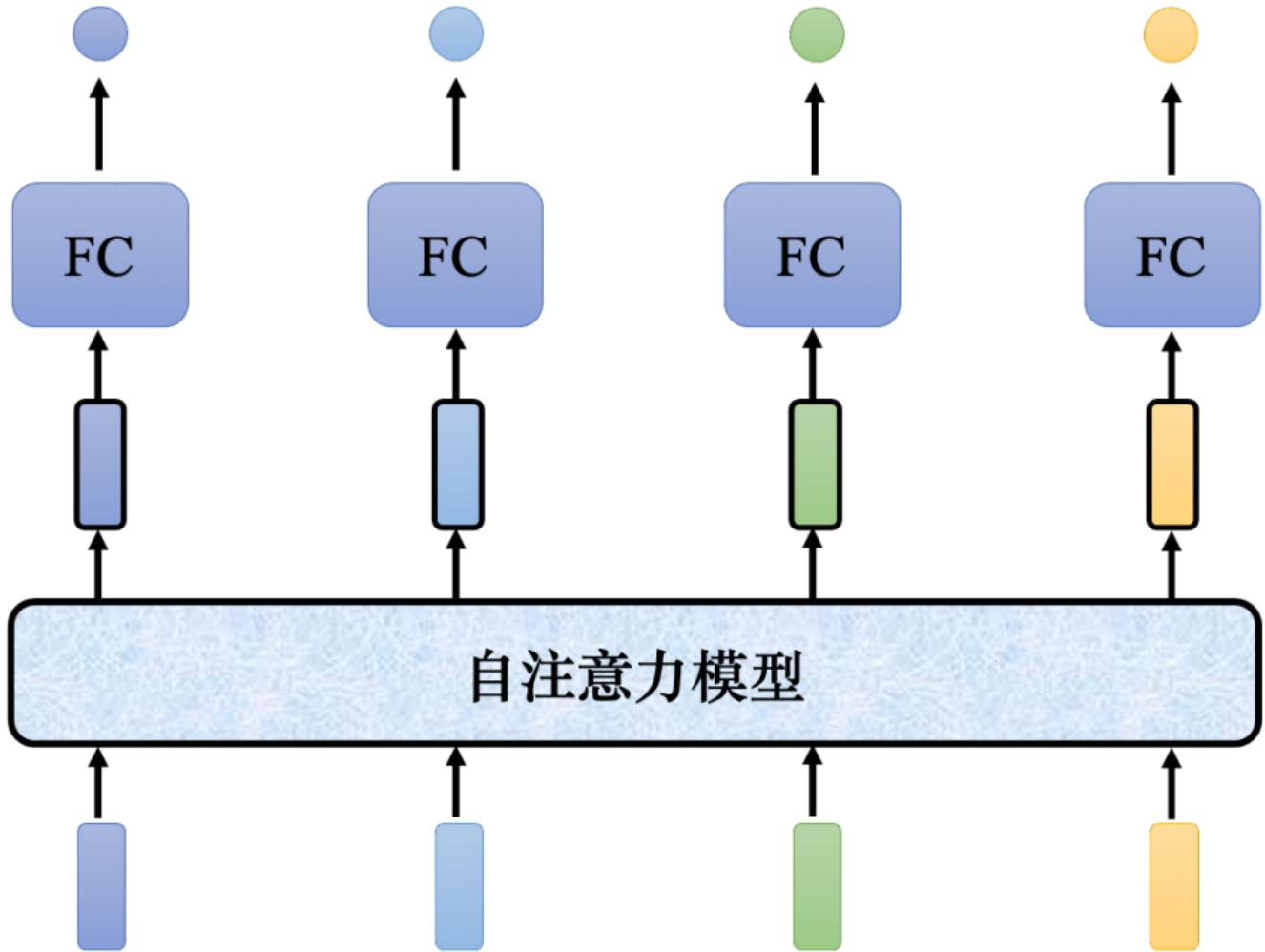
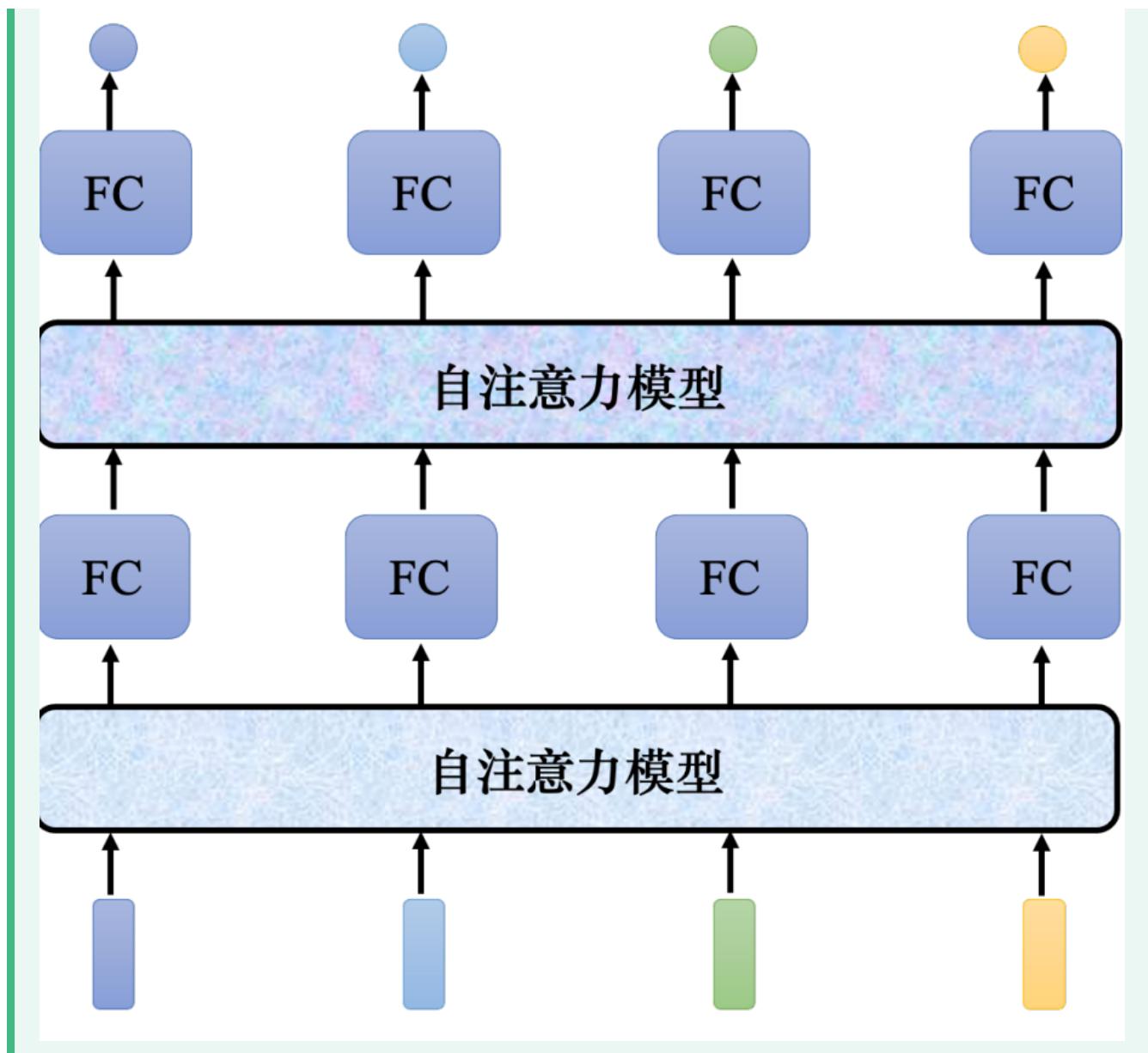


图 6.14 自注意力模型的运作方式

1. Input Sequence->Self Attention:
2. 如上图，生成的结果是带着 **黑框** 的，生成的结果是通过 **预先** 知道整个序列而生成
3. 生成的结果进入FC层，因此全连接网络不是只考虑一个非常小的范围或一个小的窗口，而是考虑整个序列信息，再来决定现在应该要输出什么样的结果，这就是自注意力模型。
4. 可以通过多个自注意力模型和FC层



2.2.2 自注意力模型的计算方式(图解方式)

先提出问题：首先自注意力模型是如何做到让一个输入向量提前预知到整个序列？？

1. 通过向量与其他所有向量之间的关联性入手，利用一个 **关联因子 a** 进行关联
2. 如何衡量本向量与整个序列的关系，通过生成一个 b 来衡量本向量与整个序列的关联性

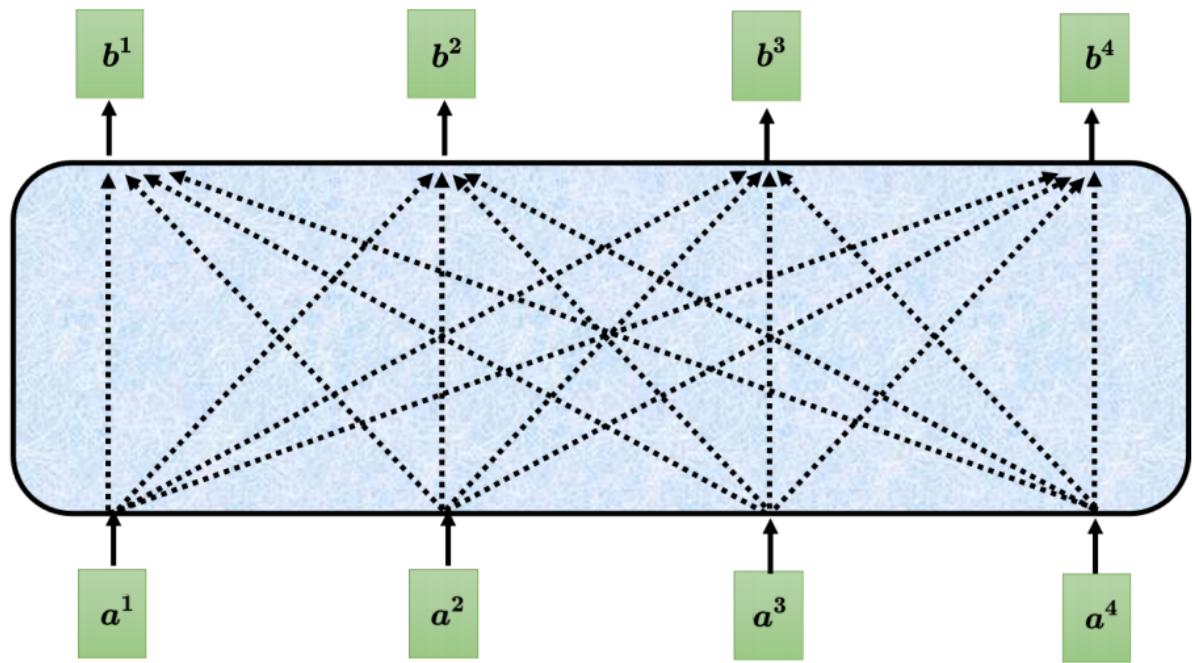
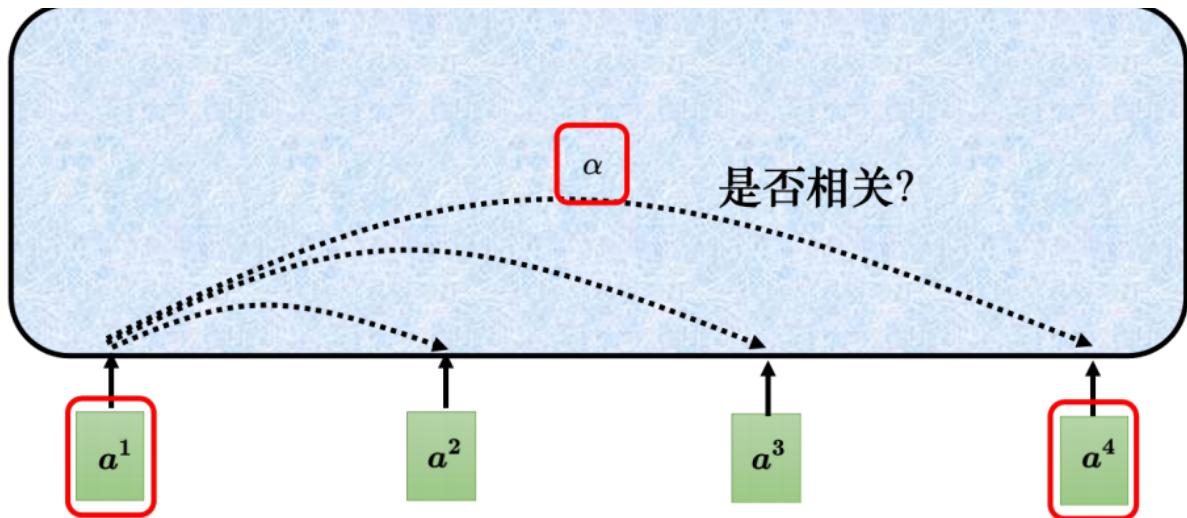


图 6.16 自注意力模型的运作方式

1. 如何计算 a_1 与其他向量的关联性(如何计算 a_1 与其他向量的注意力分数)



采用(Query-Key-Value, QKV模式)

计算步骤如下:

$$\text{计算 } a^1 \text{ 与其他向量之间的关联 : } \left\{ \begin{array}{l} q^1 = W^q a^1, \quad k^2 = W^k a^2, \quad k^3 = W^k a^3, \quad k^4 = W^k a^4 \\ \alpha_{1,2} = q^1 \cdot k^2 \\ \alpha_{1,3} = q^1 \cdot k^3 \\ \alpha_{1,4} = q^1 \cdot k^4 \end{array} \right.$$

通过上式可得, 将 a_1 与其他向量关联因子 α 算出

注意力分数是通过点积生成的标量

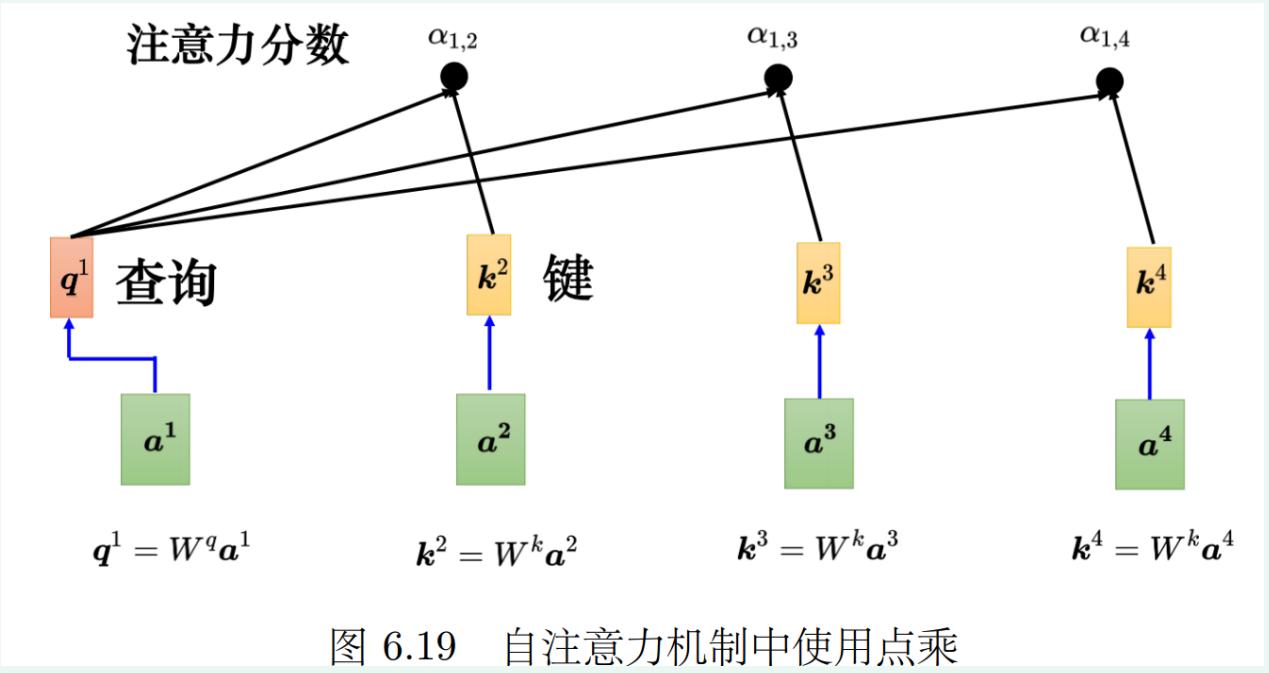


图 6.19 自注意力机制中使用点乘

注: W^k 和 W^q 矩阵下面继续解释

2. 如何衡量 \mathbf{a}_1 与其他向量之间的比重权重

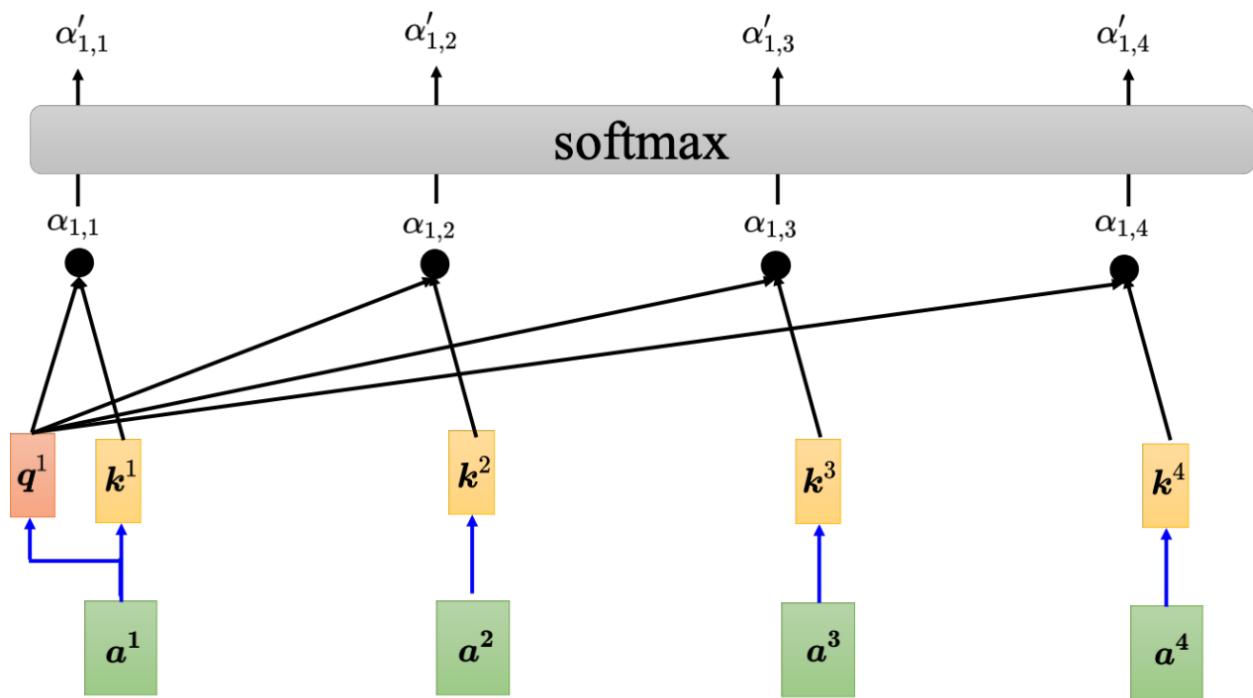
2.1 通过激活函数

因为算出的 α 可能存在负数，我们如果要衡量 \mathbf{a}_1 与其他向量之间的比重，那么就需要通过激活函数，将注意力分数组值限制在某个区域之间

- 通过 softmax 函数

$$\alpha'_{1,i} = \exp(\alpha_{1,i}) / \sum_j \exp(\alpha_{1,j}) \quad (1)$$

通过 softmax 函数可以将 α 限制在某个区域，就是说可以方便描述



$$q^1 = W^q a^1$$

$$k^2 = W^k a^2$$

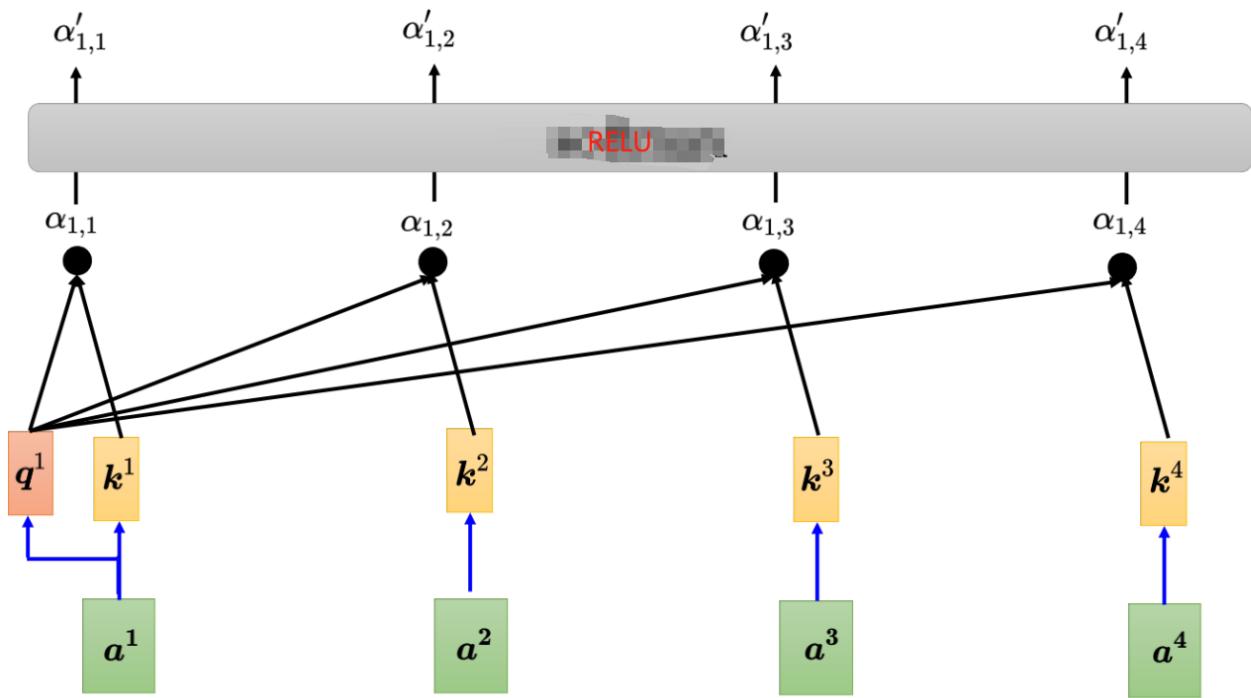
$$k^3 = W^k a^3$$

$$k^4 = W^k a^4$$

$$k^1 = W^k a^1$$

图 6.20 添加 softmax

- 通过RELU函数



$$q^1 = W^q a^1 \quad k^2 = W^k a^2 \quad k^3 = W^k a^3 \quad k^4 = W^k a^4$$

$$k^1 = W^k a^1$$

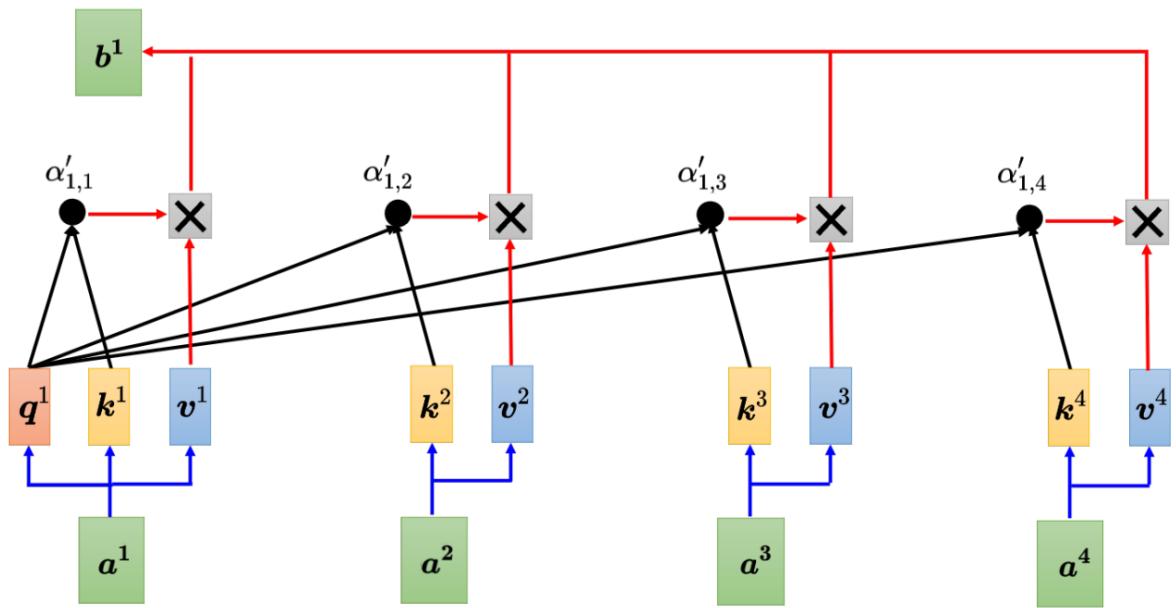
图 6.20 添加 softmax

$$\alpha'_{1,i} = R\text{Elu}(\alpha_{1,i}, 0)$$

2.2 再根据 α' 进行加权和，根据加权和来判断哪个关联性最强

$$\begin{cases} v^1 = W^v a^1, & v^2 = W^v a^2, & v^3 = W^v a^3, & v^4 = W^v a^4 \\ b^1 = \sum_i \alpha'_{1,i} v^i \end{cases}$$

注意: v 是矩阵, b 也是矩阵, α' 是标量



$$\mathbf{v}^1 = W^v \mathbf{a}^1$$

$$\mathbf{v}^2 = W^v \mathbf{a}^2$$

$$\mathbf{v}^3 = W^v \mathbf{a}^3$$

$$\mathbf{v}^4 = W^v \mathbf{a}^4$$

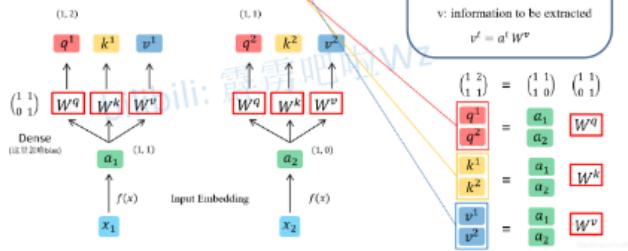
图 6.21 根据 α' 抽取序列中重要的信息

2.2.3 自注意力模型的计算方式(矩阵计算方式)

第一步：乘积生成Q K V矩阵

Self-Attention

To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{model} = 512$



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

$$q^i = a^i W^q$$

$$k^i = a^i W^k$$

$$v^i = a^i W^v$$

- q 代表 query，后续会去和每一个 k 进行匹配

- k 代表 key，后续会被每个 q 匹配

- v 代表从 a 中提取得到的信息

- 后续 q 和 k 匹配的过程可以理解成计算两者的相关性，相关性越大对应 v 的权重也就越大

$$\begin{cases} I = [a^1 \quad a^2] \text{ 代表输入的向量} \\ W^q \quad W^k \quad W^v \text{ 代表变换矩阵} \end{cases}$$

假设 $a_1 = (1, 1), a_2 = (1, 0), W^q = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$ 那么：

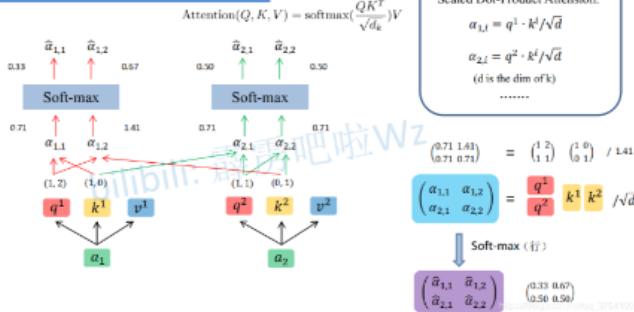
$$q^1 = (1, 1) \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = (1, 2), \quad q^2 = (1, 0) \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = (1, 1)$$

也就是说 可以简化为 $Q = I^T W^q = \begin{pmatrix} a^1 \\ a^2 \end{pmatrix} W^q = \begin{pmatrix} q^1 \\ q^2 \end{pmatrix} = \begin{pmatrix} 1, 2 \\ 1, 1 \end{pmatrix}$

同理 $K = I^T W^k = \begin{pmatrix} a^1 \\ a^2 \end{pmatrix} W^k = \begin{pmatrix} k^1 \\ k^2 \end{pmatrix}, V = I^T W^v = \begin{pmatrix} a^1 \\ a^2 \end{pmatrix} W^v = \begin{pmatrix} v^1 \\ v^2 \end{pmatrix}$

第二步：QK匹配和Softmax回归处理

Self-Attention



$$\text{Scaled Dot-Product Attention:}$$

$$\alpha_{1,i} = q^1 \cdot k^1 / \sqrt{d}$$

$$\alpha_{2,i} = q^2 \cdot k^i / \sqrt{d}$$

(d 是 dim of k)

现在得到了 Q K V 三个矩阵，
怎么能衡量各个向量之间的相关性呢？
需要做的是 Q 与 K 进行逐一匹配
因为 Q 是 query 查询，K 是 key 值

QK匹配

拿 q^1 和 q^2 去分别匹配所有的 k 能得到 $\alpha_{1,i}$ 和 $\alpha_{2,i}$ ，统一写成矩阵乘法形式：

$$\begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} \\ \alpha_{2,1} & \alpha_{2,2} \end{pmatrix} = \frac{\begin{pmatrix} q^1 \\ q^2 \end{pmatrix} \begin{pmatrix} k^1 \\ k^2 \end{pmatrix}^T}{\sqrt{d}}$$

$$\alpha_{1,1} = \frac{q^1 \cdot k^1}{\sqrt{d}} = \frac{1 \times 1 + 2 \times 0}{\sqrt{2}} = 0.71$$

$$\alpha_{1,2} = \frac{q^1 \cdot k^2}{\sqrt{d}} = \frac{1 \times 0 + 2 \times 1}{\sqrt{2}} = 1.41$$

$$\alpha_{2,1} = \frac{q^2 \cdot k^1}{\sqrt{d}} = \frac{1 \times 1 + 1 \times 0}{\sqrt{2}} = 0.71$$

$$\alpha_{2,2} = \frac{q^2 \cdot k^2}{\sqrt{d}} = \frac{1 \times 0 + 1 \times 1}{\sqrt{2}} = 0.71$$

Q : 为什么需要进行除于根号d?

A: 因为需要进行缩小，论文中的解释是“进行点乘后的数值很大，导致通过softmax后梯度变的很小”，所以通过除以根号d来进行缩放

$$\text{softmax}\left(\frac{q^1 k^1 T}{\sqrt{d_k}}\right) = \text{softmax}(\alpha_{1,1}) = \text{softmax}\left(\frac{1 \times 1 + 2 \times 0}{\sqrt{2}}\right) = \hat{\alpha}_{1,1}$$

$$\text{softmax}\left(\frac{q^1 k^2 T}{\sqrt{d_k}}\right) = \text{softmax}(\alpha_{1,2}) = \text{softmax}\left(\frac{1 \times 0 + 2 \times 1}{\sqrt{2}}\right) = \hat{\alpha}_{1,2}$$

$$\text{softmax}\left(\frac{q^2 k^1 T}{\sqrt{d_k}}\right) = \text{softmax}(\alpha_{2,1}) = \text{softmax}\left(\frac{1 \times 1 + 1 \times 0}{\sqrt{2}}\right) = \hat{\alpha}_{2,1}$$

$$\text{softmax}\left(\frac{q^2 k^2 T}{\sqrt{d_k}}\right) = \text{softmax}(\alpha_{2,2}) = \text{softmax}\left(\frac{1 \times 0 + 1 \times 1}{\sqrt{2}}\right) = \hat{\alpha}_{2,2}$$

Softmax回归处理

Q : 为什么需要进行Softmax处理？

A: Softmax处理

第三步：V运算

Self-Attention

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

$$\begin{aligned} b^1 &= \sum_i \hat{\alpha}_{1,i} \times v^i \\ b^2 &= \sum_i \hat{\alpha}_{2,i} \times v^i \end{aligned}$$

上面已经计算得到 α ，即针对每个 v 的权重，接着进行加权得到最终结果：

$$b_1 = \hat{\alpha}_{1,1} \times v^1 + \hat{\alpha}_{1,2} \times v^2 = (0.33, 0.67)$$

$$b_2 = \hat{\alpha}_{2,1} \times v^1 + \hat{\alpha}_{2,2} \times v^2 = (0.50, 0.50)$$

统一写成矩阵乘法形式：

$$\begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} \hat{\alpha}_{1,1} & \hat{\alpha}_{1,2} \\ \hat{\alpha}_{2,1} & \hat{\alpha}_{2,2} \end{pmatrix} \begin{pmatrix} v^1 \\ v^2 \end{pmatrix}$$

总结就是这个公式

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

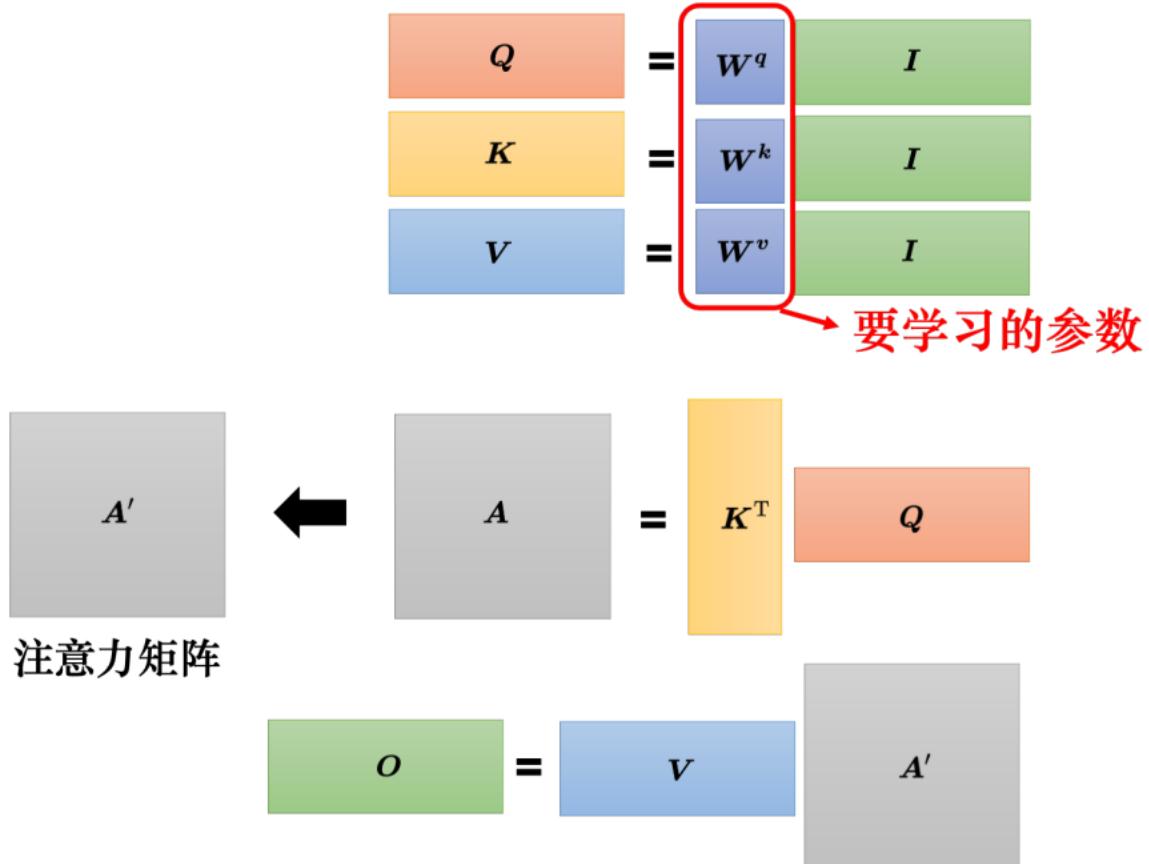


图 6.28 从矩阵乘法的角度来理解注意力

一个字就是 W^q W^k W^v 都是要调整的参数，才能影响整个Attention模型

2.2.4 自注意力机制的起源

参考这篇文章: Transformer: 注意力机制 (attention) 和自注意力机制 (self-attention) 的学习总结

说的是通过 平均汇聚方法 来加和

2.3 Multi-Head Attention(多头注意力模型)

详解Transformer中Self-Attention以及Multi-Head Attention

2.4 总结

Self attention解决了可以让所有的单独的向量提前预知整个序列的向量，且计算出与其最相关的向量是哪个向量

2.4 (positional encoding)位置编码

位置编码为每一个位置设定一个向量，即位置向量 (positional vector)。位置向量用 e_i 来表示，上标 i 代表位置，不同的位置就有不同的向量，不同的位置都有一个专属的 e ，把 e 加到 a_i 上面就结束了。这相当于告诉自注意力位置的信息，如果看到 a_i 被加上 e_i ，它就知道现在出现的位置应该是在 i 这个位置。

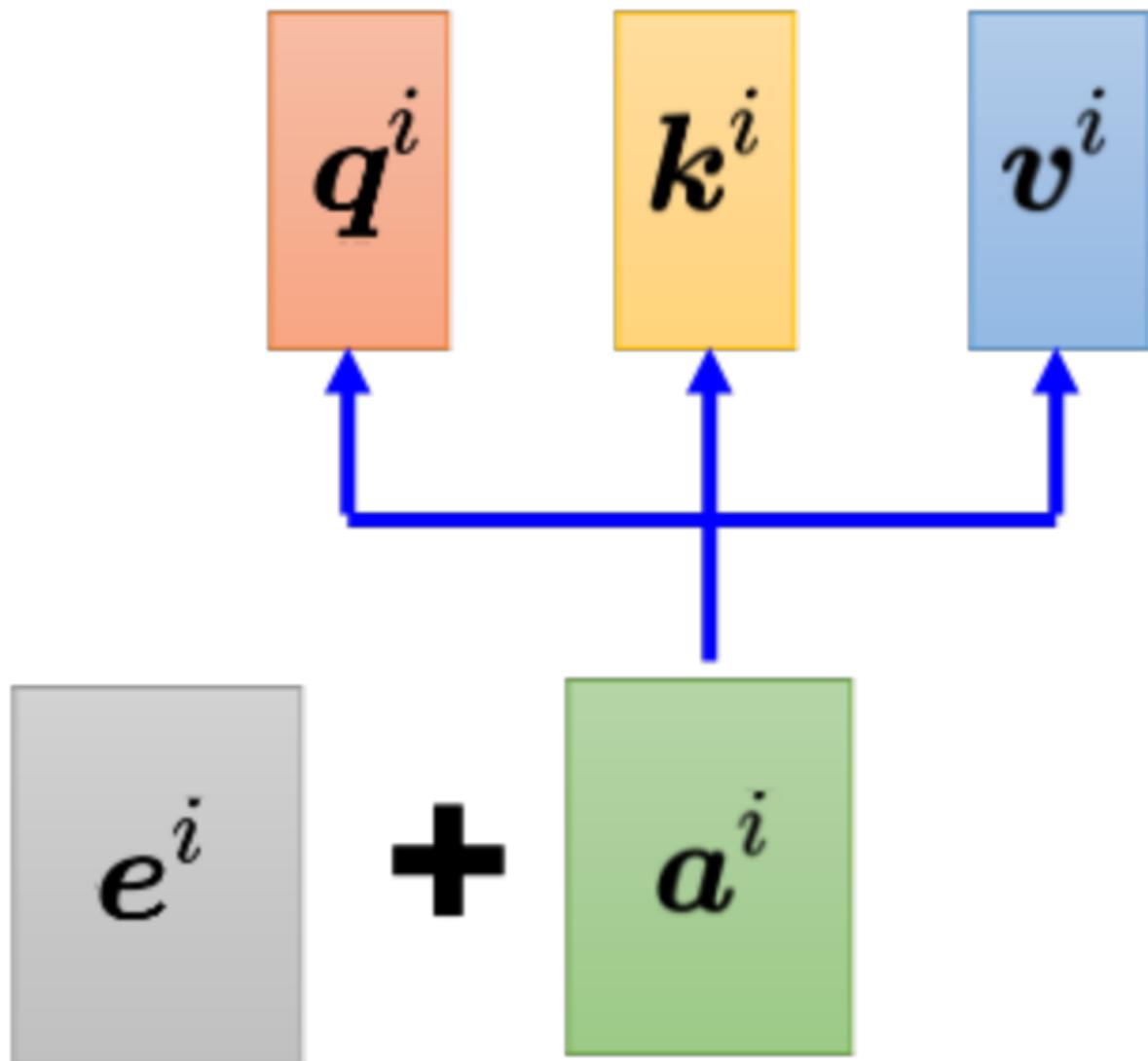


图 6.32 位置编码

Q：为什么要通过正弦函数和余弦函数产生向量，有其他选择吗？为什么一定要这样产生手工的位置向量呢？

A：不一定要通过正、余弦函数来产生向量，我们可以提出新的方法。此外，不一定要这样产生手工的向量，位置编码仍然是一个尚待研究的问题，甚至位置编码是可以根据数据学出来的。有关位置编码，可以参考论文“Learning to Encode Position for Transformer with Continuous Dynamical Model”，该论文比较了不同的位置编码方法并提出了新的位置编码。

2.5 (truncated self-attention) 截断自注意力

可以处理向量序列长度过大的问题

如果要辨识某个位置有什么样的音标，这个位置有什么样的内容并不需要看整句话，只要看这句话以及它前后一定范围之内的信息，就可以判断。在做自注意力的时候，也许没有必要让自注意力考虑一整个句子，只需要考虑一个小范围就好，这样就可以加快运算的速度。这就是截断自注意力。

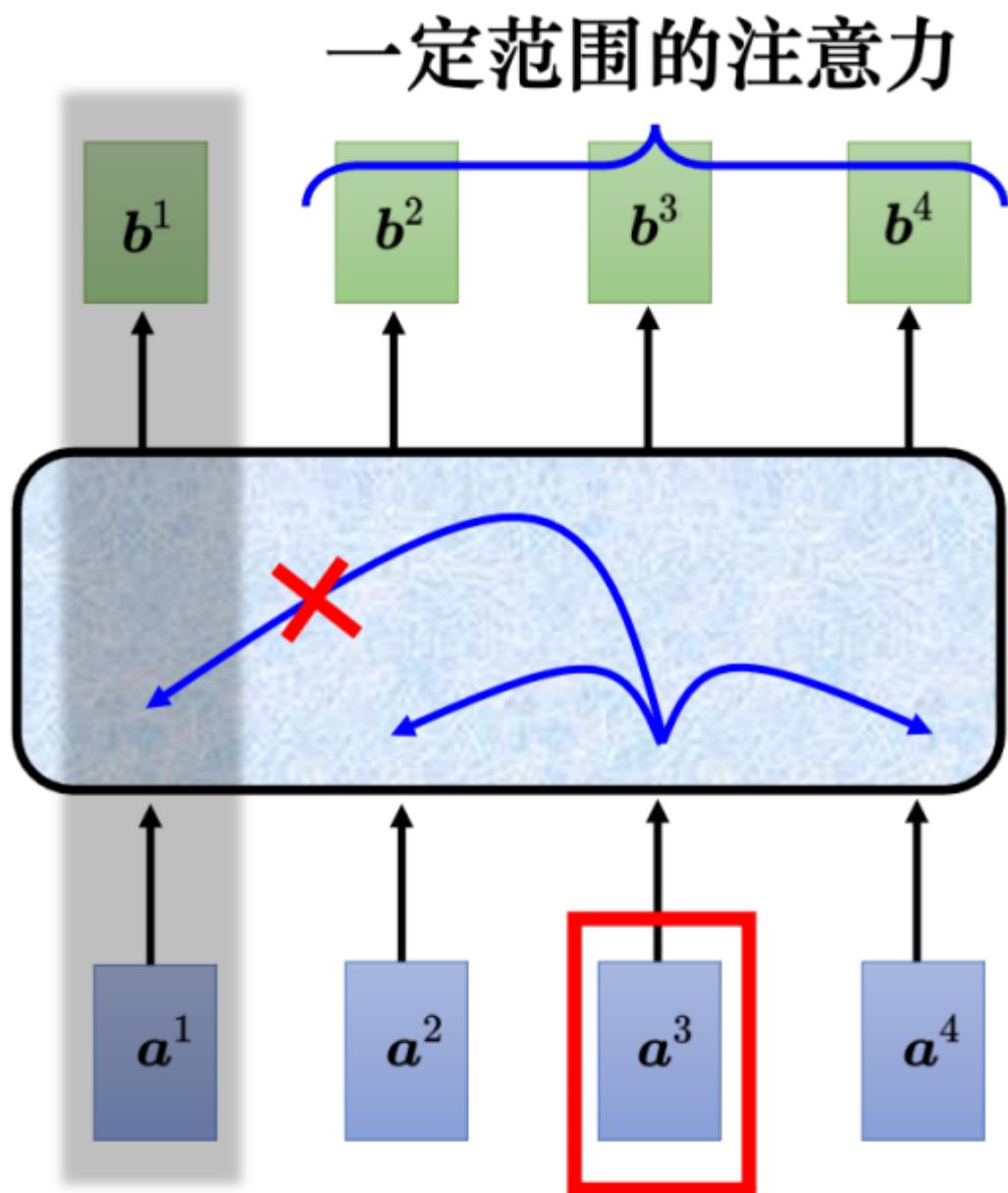


图 6.35 截断自注意力

2.6 自注意力与卷积神经网络对比

自注意力还可以被用在图像上, 一张图像可以看作是一个向量序列,

3. Transformer

Transformer 是一个基于自注意力的序列到序列模型, 与基于循环神经网络的序列到序列模型不同, 其可以能够并行计算。

3.1 序列到序列模型

1. 输入和输出长度一样
2. 机器决定输出的长度

应用场景:

1. 语音识别: 输入是声音信号, 输出是语音识别的结果, 即 输入的这段声音信号所对应的文字。
2. 机器翻译: 机器 输入一个语言的句子, 输出另外一个语言的句子, 输入句子的长度是N, 输出句子的长度是N', N 跟 N'之间的关系由机器决定。
3. 语音翻译: 我们对机器说一句话, 比如“machine learning”, 机器直接把听到的英语的声音信号翻译成中文。
4. 语音合成: 输入文字、输出声音信号就是语音合成 (Text-To-Speech, TTS)。现在还没有真的做端到端 (end-to-end) 的模型, 以闽南语的语音合成为例, 其使用的模型还是分成两阶, 首先模型会先把白话文的文字转成闽南语的拼音, 再把闽南语的拼音转成声音信号。从闽南语的拼音转成声音信号这一段是通过序列到序列模型 echotron 实现的。

5. 聊天机器人：



图 7.2 聊天机器人的例子

6. 问答任务：

序列到序列模型在自然语言处理的领域的应用很广泛，而很多自然语言处理的任务都可以想成是问答（Question Answering, QA）的任务，比如下面是一些例子。

- 翻译。机器读的文章是一个英语句子，问题是这个句子的德文翻译是什么？输出的答案就是德文。
- 自动做摘要：给机器读一篇长的文章，让它把长的文章的重点找出来，即给机器一段文字，问题是这段文字的摘要是什么。
- 情感分析：机器要自动判断一个句子是正面的还是负面的。如果把情感分析看成是问答的问题，问题是给定句子是正面还是负面的，希望机器给出答案。

问答就是给机器读一段文字，问机器一个问题，希望它可以给出一个正确的答案。

7. 句法分析

在句法分析的任务中，输入是一段文字，输出是一个树状的结构，而一个树状的结构可以看成一个序列，该序列代表了这个树的结构。把树的结构转成一个序列以后，我们就可以用序列到序列模型来做句法分析，具体可参考论文“Grammar as a Foreign Language”

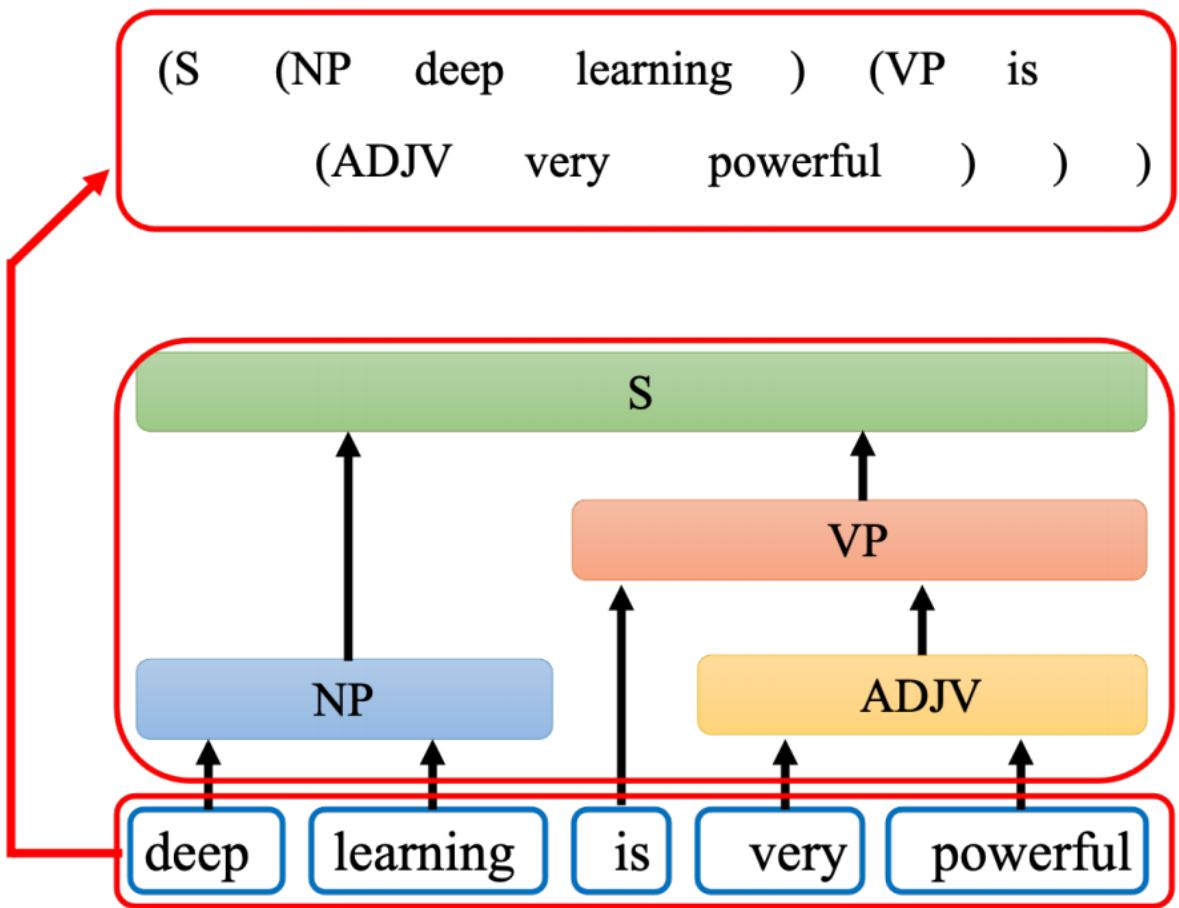


图 7.4 树状结构对应的序列

8. 多标签分类

多分类问题（multi-class classification）是指分类的类别数大于 2。而多标签分类是指同一个东西可以属于多个类。

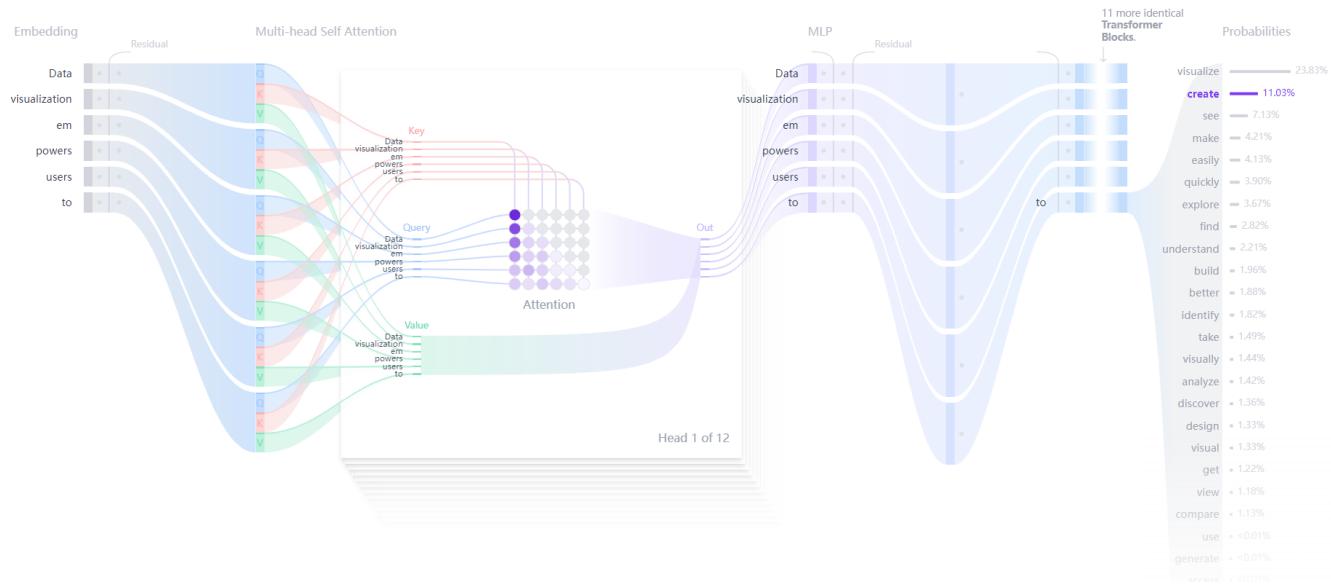
文章 1	文章 2	文章 3	文章 4
类 1 类 3	类 1	类 3 类 9	类 10 类 17

图 7.5 多标签分类示例



图 7.6 序列到序列模型来解多标签分类问题

3.2 Transformer 结构



整个Transformer结构如这个结构化所示

3.2.1 Embedding layer (输入嵌入)

假设输入一句 "Hello! This is an example of a paragraph that has been split into its basic components. I wonder what will come next! Any guesses?"

通俗的理解：输入是序列只是一段句子，我们需要将他转化为张量才能输入到模型中，那么就需要进行词的分割，且转化为张量

精准的定义：Embedding layer可以将看成将序列数据从 **高维空间** 转化为 **低维空间**，或者理解为 **高维的稀疏矩阵** 转化为 **低维的稠密矩阵**

Embedding layer的方法：

- **ONE-HOT**
- PCA降维
- **Word2Vec**

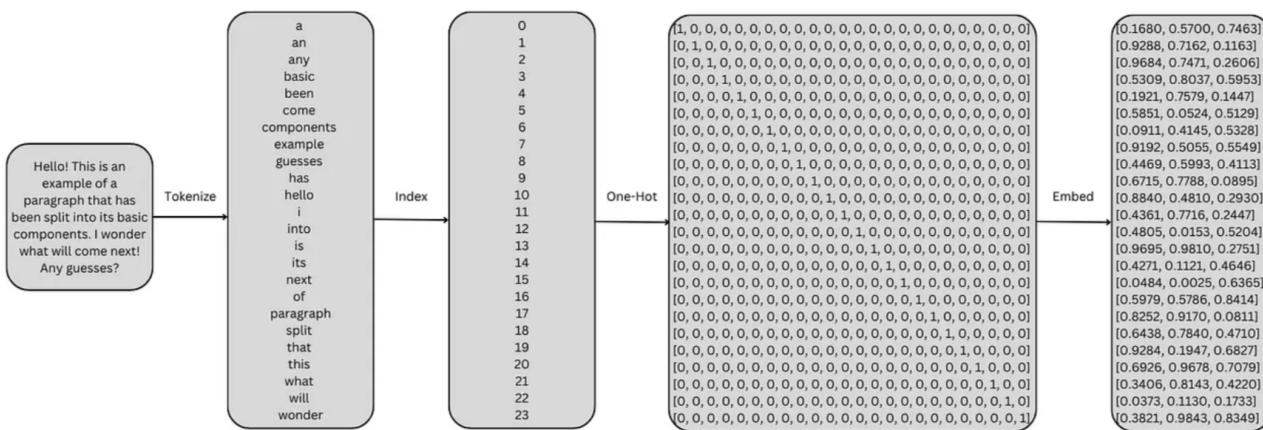
- Item2Vec
- 矩阵分解
- 基于深度学习的协同过滤

Let us create the embedding from scratch

1. Vocabulary ONE-HOT 2 lower dim Vector

我们需要先建立vocabulary 词典的映射关系，因为embedding layer是需要建立词典的映射关系，然后后续的句子是通过词典里面来寻找的

因为句子是一个一个单词组成的，经典的思想就是将"句子"转化为词数组，这个时候就需要使用到我们经典的ONE-HOT编码



根据上图，

1. 先Tokenize,那么Tokenize的代码如下

```
...
Author: Jean_Leung
Date: 2024-09-06 10:10:58
LastEditors: Jean_Leung
LastEditTime: 2024-09-06 11:21:21
FilePath: \LinearModel\transformer_pratice\positional_embedding.py
Description: 搞懂Transformer的Positional_embedding layer的运作方式
```

```
Copyright (c) 2024 by ${robotlive limit}, All Rights Reserved.
```

```
...
# importing required libraries
import math
import copy
import numpy as np

# torch packages
import torch
```

```

import torch.nn as nn
import torch.nn.functional as F
from torch import Tensor

# visualization packages
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt

example = "Hello! This is an example of a paragraph that has been split into its basic components. I wonder what will come next! Any guesses?"

# Tokenize the input text into individual words
# 第一步先 tokenize 字符串，经典的就是按照每个词进行 tokenize
# tokenize 就是将文本分割成离散的单元
def tokenize(sequence):
    # remove punctuation
    # 移除标点符号
    for punc in ["!", ".", "?"]:
        sequence = sequence.replace(punc, "") # 用无字符取代标点位置
    # 返回一个字符数组
    return [token.lower() for token in sequence.split(" ")]

print(tokenize(example))

```

输出结果:

```

['hello', 'this', 'is', 'an', 'example', 'of', 'a', 'paragraph', 'that', 'has', 'been', 'split',
'into', 'its', 'basic', 'components', 'i', 'wonder', 'what', 'will', 'come', 'next', 'any',
'guesses']

```

2. 先用ONE-HOT建立词库映射，word:index 的关系

```

# 下一步按照词的开头 ASCII 进行排序，从 a-z 进行排序

def build_vocab(data):
    # 构建一个单词到 index 的映射
    vocab = list(set(tokenize(data)))
    # 进行排序，默认快排，从小到大排序
    # vocab.sort(key=lambda x: (ord(x[0]), x))
    vocab.sort()
    # 建立一个 index 到单词的映射
    word_to_idx = {word: idx for idx, word in enumerate(vocab)}
    return word_to_idx

stoi = build_vocab(example) # 建立词库映射
print(stoi)

```

输出结果:

```

{'a': 0, 'an': 1, 'any': 2, 'basic': 3, 'been': 4, 'come': 5, 'components': 6, 'example': 7,
'guesses': 8, 'has': 9, 'hello': 10, 'i': 11, 'into': 12, 'is': 13, 'its': 14, 'next': 15, 'of': 16, 'paragraph': 17, 'split': 18, 'that': 19, 'this': 20, 'what': 21, 'will': 22, 'wonder': 23}

```

3. 现在已经根据映射建立了key:value的mapping关系，也就是one-hot编码，这个是词库的映射，接下我们需要生成 **低维度的稠密矩阵** 那么应该怎么生成呢？

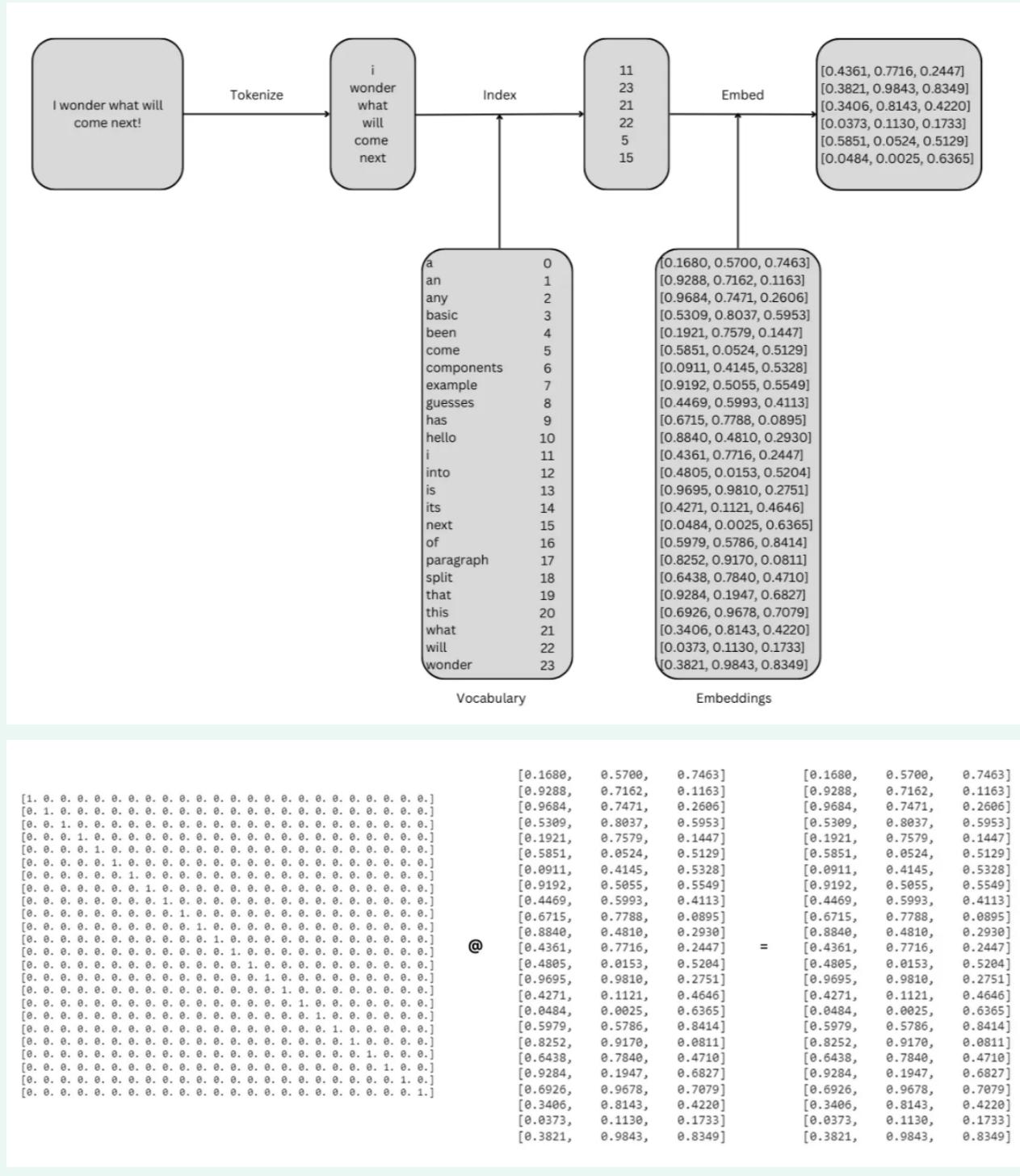
实现代码可以参考这篇文章生成embedding matrix [word2vec的原理及实现（附github代码）](#)

实现简洁的基础版本代码参考这篇文章: [The Embedding Layer](#)

原理可以参考这篇文章 [Word Embedding教程](#)

这是演示过程，那么我们可以默认已经随机生成了E矩阵，然后通过

这个过程就是高维稀疏矩阵通过矩阵乘法映射成低维稠密矩阵



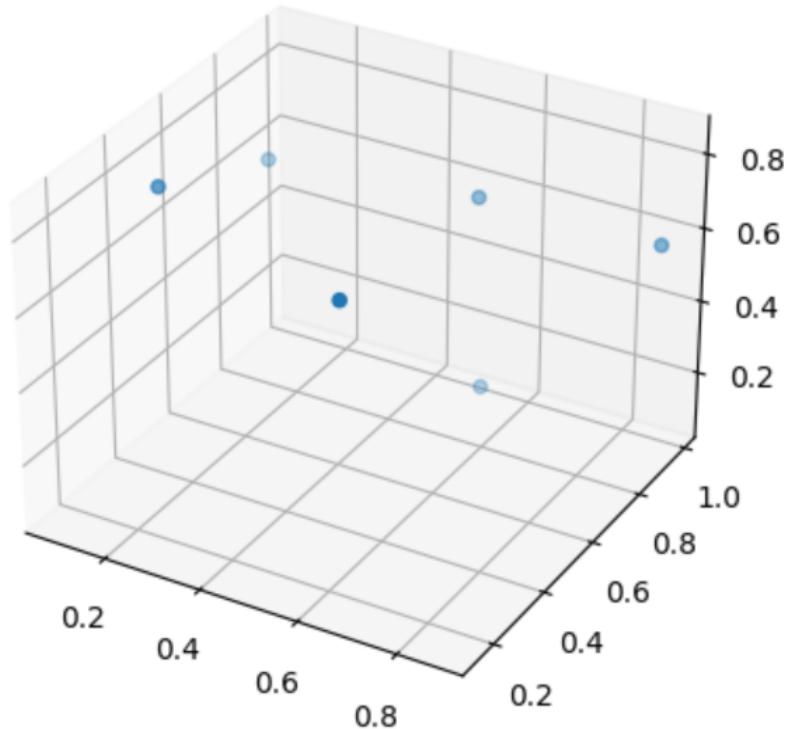
```
sequence = [stoi[word] for word in tokenize("I wonder what will come next!")]
print(sequence)

# Positional Encoding layer
# vocab size
# 获得 vocab_size
vocab_size = len(stoi)
# d_model 隐藏单元的个数
d_model = 3 # 数据是三维,这个时候embedding矩阵是(24, 3)的

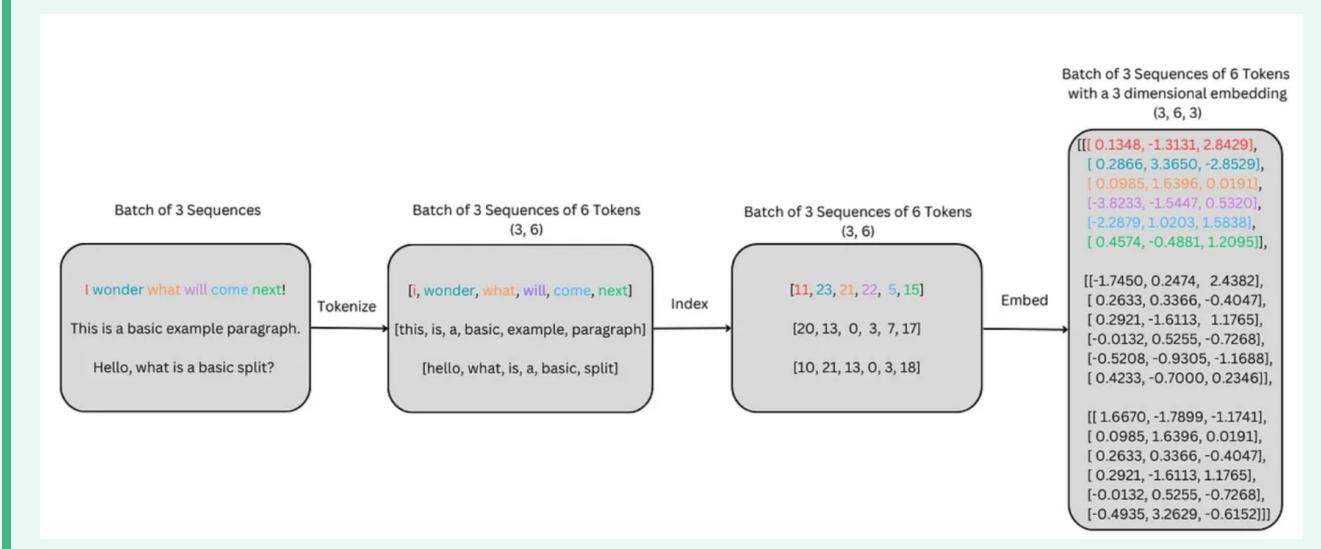
# 构建一个position的embedding
embedding = torch.rand(vocab_size,d_model) # size为(24, 3)矩阵, 因为需要进行运算
print(embedding)
embedded_sequence = embedding[sequence]
print("embedded_sequence:", embedded_sequence)
print("embedded_sequence.shape:", embedded_sequence.shape) # size为(6, 3)
```

```
embedded_sequence: tensor([[ 0.6384,  0.0727,  0.9703],
                           [ 0.3984,  0.1473,  0.1994],
                           [ 0.8784,  0.3126,  0.1634],
                           [ 0.5357,  0.7002,  0.5487],
                           [ 0.0914,  0.3244,  0.3288],
                           [ 0.1613,  0.3047,  0.1960]])
embedded_sequence.shape: torch.Size([6, 3])
```

Figure 1



4. 利用Pytorch自带的embedding进行生成矩阵



```
sequence = [stoi[word] for word in tokenize("I wonder what will come next!")]
# print(sequence)
```

```

# Positional Encoding layer
# vocab size
# 获得 vocab_size
vocab_size = len(stoi)
# d_model 隐藏单元的个数
d_model = 3 # 数据是三维

# 构建一个position的embedding
# embedding = torch.rand(vocab_size,d_model) # size为(24, 3)矩阵, 因为需要进行运算
lut = nn.Embedding(vocab_size,d_model) # nn自带的Embedding类
lut.state_dict()['weight']
indices = torch.Tensor(sequence).long()
embedding = lut(indices)
print(embedding)

```

- 总代码

```

...
Author: Jean_Leung
Date: 2024-09-06 10:10:58
LastEditors: Jean_Leung
LastEditTime: 2024-09-06 14:35:22
FilePath: \LinearModel\transformer_practise\_token_embedding.py
Description: 搞懂Transformer的Positional_embedding layer的运作方式
    1. 从零开始建立词库映射
        1.1 先有单词库, 然后我们利用One-hot映射成一个稀疏的高维矩阵
        1.2 再用一个转化矩阵将稀疏的高维矩阵转化为稠密的低维矩阵

```

Copyright (c) 2024 by \${robotlive limit}, All Rights Reserved.

```

# importing required libraries
import math
import copy
import numpy as np

# torch packages
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch import Tensor

# visualization packages
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt

example = "Hello! This is an example of a paragraph that has been split into its basic components. I wonder what will come next! Any guesses?"

# Tokenize the input text into individual words
# 第一步先 tokenize字符串, 经典的就是按照每个词进行 tokenize
# tokenize 就是将文本分割成离散的单元
def tokenize(sequence):

```

```

# remove punctuation
# 移除标点符号
for punc in["!", ".", "?"]:
    sequence = sequence.replace(punc, "")
# 返回一个字符数组
return [token.lower() for token in sequence.split(" ")]

# print(tokenize(example))

# 下一步按照词的开头ASCII进行排序，从a-z进行排序

def build_vocab(data):
    # 构建一个单词到index的映射
    vocab = list(set(tokenize(data)))
    # 进行排序，默认快排，从小到大排序
    # vocab.sort(key=lambda x: (ord(x[0]), x))
    vocab.sort()
    # 建立一个index到单词的映射
    word_to_idx = {word: idx for idx, word in enumerate(vocab)}
    return word_to_idx

stoi = build_vocab(example) # 建立词库映射
# print(stoi)

sequence = [stoi[word] for word in tokenize("I wonder what will come next!")]
# print(sequence)

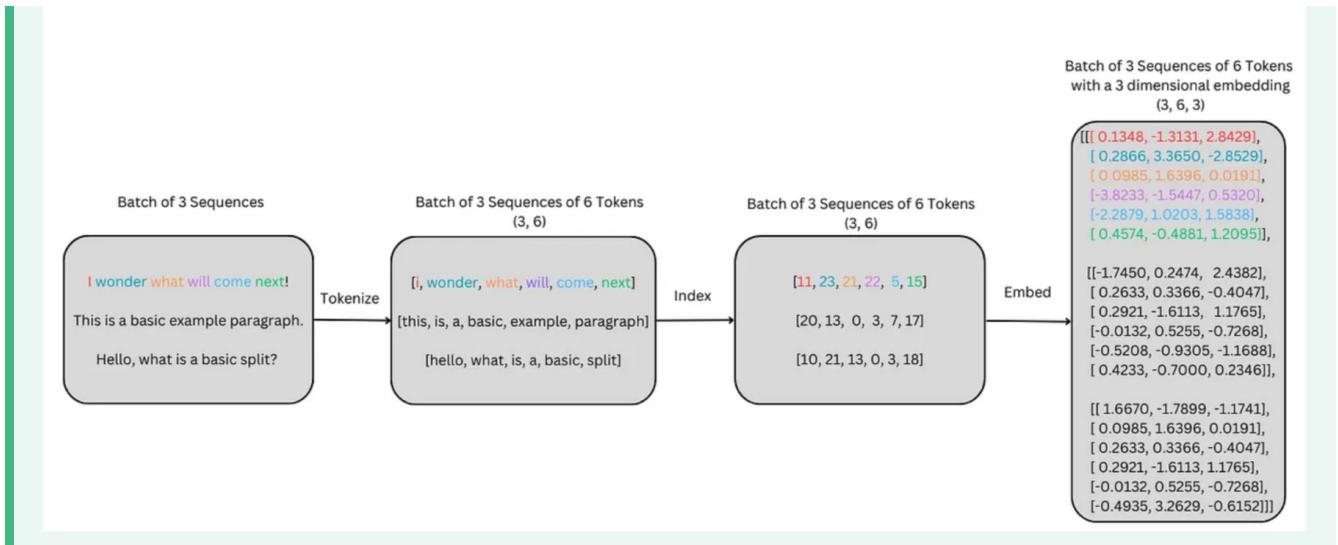
# Positional Encoding layer
# vocab size
# 获得 vocab_size
vocab_size = len(stoi)
# d_model 隐藏单元的个数
d_model = 3 # 数据是三维

# 构建一个position的embedding
# embedding = torch.rand(vocab_size,d_model) # size为(24,3)矩阵，因为需要进行运算
lut = nn.Embedding(vocab_size,d_model)
lut.state_dict()['weight'] # 返回字典dict
indices = torch.Tensor(sequence).long()
embedding = lut(indices)
print(embedding)
# embedded_sequence = embedding[sequence]
# print("embedded_sequence:", embedded_sequence)
# print("embedded_sequence.shape:", embedded_sequence.shape) # size为(6,3)
# 3D visualization of the positional encoding
# x,y,z = embedded_sequence[:,0],embedded_sequence[:,1],embedded_sequence[:,2]
# ax = plt.axes(projection='3d')
# ax.scatter3D(x,y,z)
# plt.show()

```

假设输入三段序列

```
sequences = ["I wonder what will come next!", "This is a basic example paragraph.", "Hello, what is a basic split?"]
```



```
...
Author: Jean_Leung
Date: 2024-09-06 10:10:58
LastEditors: Jean_Leung
LastEditTime: 2024-09-06 14:50:31
FilePath: \LinearModel\transformer_practise\_token_embedding.py
Description: 搞懂Transformer的Positional_embedding layer的运作方式
    1. 从零开始建立词库映射
        1.1 先有单词库，然后我们利用One-hot映射成一个稀疏的高维矩阵
        1.2 再用一个转化矩阵将稀疏的高维矩阵转化为稠密的低维矩阵
```

Copyright (c) 2024 by \${robotlive limit}, All Rights Reserved.

```
# importing required libraries
import math
import copy
import numpy as np

# torch packages
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch import Tensor

# visualization packages
from mpl_toolkits import mplot3d
import matplotlib.pyplot as plt

example = "Hello! This is an example of a paragraph that has been split into its basic components.  
I wonder what will come next! Any guesses?"

# Tokenize the input text into individual words
# 第一步先 tokenize 字符串，经典的就是按照每个词进行 tokenize
# tokenize 就是将文本分割成离散的单元
def tokenize(sequence):
    # remove punctuation
```

```

# 移除标点符号
for punc in["!", ".", "?", ","]:
    sequence = sequence.replace(punc, "")
# 返回一个字符数组
return [token.lower() for token in sequence.split(" ")]

# print(tokenize(example))

# 下一步按照词的开头ASCII进行排序，从a-z进行排序

def build_vocab(data):
    # 构建一个单词到index的映射
    vocab = list(set(tokenize(data)))
    # 进行排序，默认快排，从小到大排序
    # vocab.sort(key=lambda x: (ord(x[0]), x))
    vocab.sort()
    # 建立一个index到单词的映射
    word_to_idx = {word: idx for idx, word in enumerate(vocab)}
    return word_to_idx

stoi = build_vocab(example) # 建立词库映射
# print(stoi)

# sequence = [stoi[word] for word in tokenize("I wonder what will come next!")]
sequences = ["I wonder what will come next!",
             "This is a basic example paragraph.",
             "Hello, what is a basic split?"]

# tokenize the sequences
tokenized_sequences = [tokenize(seq) for seq in sequences]
print('tokenized sequences', tokenized_sequences)
# index the sequences
indexed_sequences = [[stoi[word] for word in seq] for seq in tokenized_sequences]
print('indexed sequences', indexed_sequences)
vocab_size = len(stoi) # 词向量行长度
d_model = 3
lut = nn.Embedding(vocab_size,d_model)
lut.state_dict()['weight']
# convert the sequences to a tensor
# 将sequence转化为tensor
tensor_sequences = torch.tensor(indexed_sequences).long()
embedding = lut(tensor_sequences) # 进行匹配
print('embedding', embedding)

```

输出结果

```

tokenized sequences [['i', 'wonder', 'what', 'will', 'come', 'next'], ['this', 'is', 'a', 'basic', 'example', 'paragraph'], ['hello', 'what', 'is', 'a', 'basic', 'split']]

indexed sequences [[11, 23, 21, 22, 5, 15], [20, 13, 0, 3, 7, 17], [10, 21, 13, 0, 3, 18]]

embedding tensor([[[ 0.3408, -1.5560, -1.9805],
                  [-0.0814,  0.3851, -0.6627],
                  [ 1.2521, -1.3471,  1.7690],

```

```
[ 1.4194, -0.2584,  0.2271],  
[ 0.4447,  0.4902, -0.9708],  
[-0.5985,  1.8578, -0.4236]],  
  
[[ 0.0246, -0.7061,  0.3169],  
[ 0.4956,  1.2468, -0.0811],  
[ 0.2099,  0.1917, -1.3613],  
[ 0.7798, -0.5529, -0.8695],  
[ 0.2796, -0.1313, -0.2061],  
[ 0.6807, -0.3493,  0.2217]],  
  
[[-1.0525,  0.0185,  0.3689],  
[ 1.2521, -1.3471,  1.7690],  
[ 0.4956,  1.2468, -0.0811],  
[ 0.2099,  0.1917, -1.3613],  
[ 0.7798, -0.5529, -0.8695],  
[ 0.4003,  0.7833,  0.4854]]], grad_fn=<EmbeddingBackward0>)
```

2. 总结

- 关键在于embedding matrix的生成原理要搞懂,上面的例子是通过nn.Embedding()模块生成的,原理可以参考上面那两篇文章
- 要理解One-Hot矩阵通过与Embedding矩阵相乘,转化为低维的矩阵

3.2.2 Positional Encoding(位置编码)

参考这篇文章,已经讲的很清楚

[【Transformer系列】深入浅出理解Positional Encoding位置编码](#)

代码实现

下面来讲代码实现:

代码实现参考这篇文章:

[Positional Encoding](#)

```
...  
Author: Jean_Leung  
Date: 2024-09-06 16:15:20  
LastEditors: Jean_Leung  
LastEditTime: 2024-09-06 17:57:42  
FilePath: \LinearModel\transformer_practise\_positional_encoding.py  
Description: 位置编码练习  
1. 相对位置编码:  
    1.1 正余弦位置编码  
    1.2 复数函数位置编码  
2. 绝对位置编码:
```

```
Copyright (c) 2024 by ${robotlive limit}, All Rights Reserved.  
...  
  
# importing required libraries  
import math  
import copy  
import numpy as np  
  
# torch packages  
import torch  
import torch.nn as nn  
import torch.nn.functional as F  
from torch import Tensor  
  
# visualization packages  
from mpl_toolkits import mplot3d  
import matplotlib.pyplot as plt  
  
# 建立词库  
# -----词典序列-----  
# 词典包含了所有所有接下来准备出现的单词  
example = "Hello! This is an example of a paragraph that has been split into its basic components.  
I wonder what will come next! Any guesses?"  
  
# 将词库中所有的词进行tokenize  
  
def tokenize(sequence):  
    # remove punctuation  
    # 移除标点符号  
    for punc in["!", ".", "?", ",":]  
        sequence = sequence.replace(punc, "")  
    # 返回一个字符数组  
    # print('sequence:', sequence)  
    return [token.lower() for token in sequence.split(" ")]  
  
# 返回了tokenize之后的数组  
# tokens = tokenize(example)  
# print('tokens:', tokens)  
  
# 对词典进行排序，且进行word:index映射  
def build_vocab(data):  
    vocab = list(set(tokenize(data))) # 先用集合将序列数组进行去重，然后再用list集合存储  
    vocab.sort() # 排序  
    # 建立word:index的映射  
    word_to_idx = {word: idx for idx, word in enumerate(vocab)}  
    return word_to_idx  
  
# 间接建立了one-hot矩阵，因为不是用矩阵存储的，所以说是间接  
stoi = build_vocab(example)  
# print("stoi: ", stoi)  
#  
  
#-----测试序列-----
```

```

# sequence = [stoi[word] for word in tokenize("I wonder what will come next!")]
sequences = ["I wonder what will come next!",
             "This is a basic example paragraph.",
             "Hello, what is a basic split?"]

# 得到了tokenized的序列
def tokenize_sequence(sequences, stoi):
    # sequences_tokens: [[['i', 'wonder', 'what', 'will', 'come', 'next'], ['this', 'is', 'a', 'basic', 'example', 'paragraph']], [['hello', 'what', 'is', 'a', 'basic', 'split']]]
    sequences_tokens = [tokenize(sequence) for sequence in sequences] # 得到sequence token的数组例如
    # sequences_indices: [[11, 23, 21, 22, 5, 15], [20, 13, 0, 3, 7, 17], [10, 21, 13, 0, 3, 18]]
    # 得到sequences里面的词的位置向量，是一个(3,6)的矩阵
    sequences_indices = [[stoi[word] for word in sequence_tokens] for sequence_tokens in
    sequences_tokens]
    return sequences_indices

# sequences_tokens = [tokenize(sequence) for sequence in sequences] # 得到sequence token的数组例如
# # print('sequences_tokens:', sequences_tokens)
# sequences_indices = [[stoi[word] for word in sequence_tokens] for sequence_tokens in
# sequences_tokens]
# print('sequences_indices:', sequences_indices)

vocab_size = len(stoi) # 词向量行长度
d_model = 4 # embedding 的d_model维度
max_length = 10 # maximum sequence length
n = 100

# -----embedding-----
# # 调用nn里面的embedding模块
# lut = nn.Embedding(vocab_size,d_model) # look-up model
# lut.state_dict()['weight']
# # for param in lut.state_dict():
# #     print('parameter',param)
# # 根据官方文档，接受的是张量，所以我们需要将sequence_indices转化为张量
# sequences_indices = tokenize_sequence(sequences=sequences,stoi=stoi)
# tensor_sequences = torch.tensor(sequences_indices).long() #转化为张量在输入到nn.Embedding里面，且需要
long化
# embedding = lut(tensor_sequences) # 得到embedding矩阵
# print('embedding:',embedding)
def embedding_(vocab_size,d_model):
    lut = nn.Embedding(vocab_size,d_model) # look-up model, 里面是基于word2Vec的思想进行生成矩阵
    # 得仔细研读
    lut.state_dict()['weight'] # 字典对象筛选出来，只剩tensor对象
    sequences_indices = tokenize_sequence(sequences=sequences,stoi=stoi)
    tensor_sequences = torch.tensor(sequences_indices).long() #转化为张量在输入到nn.Embedding里面，且需
要long化
    embedding = lut(tensor_sequences) # 得到embedding矩阵
    return embedding
# 生成embedding之后的矩阵
embeddings = embedding_(vocab_size=vocab_size, d_model=d_model)
# print('embeddings:', embeddings)

```

```

# -----相对位置编码-----
# set the output to 2 decimal places without scientific notation
torch.set_printoptions(precision=2, sci_mode=False)

def gen_pe(max_length,d_model,n):
    # 初始化pe,将pe初始化成(max_length,d_model)的零矩阵张量
    pe = torch.zeros(max_length, d_model)
    position = torch.arange(0, max_length)
    # print('position.shape: ',position.shape)
    # print('position without unsqueeze: ',position)
    position = position.unsqueeze(-1)
    # print('position.shape: ',position.shape)
    # print('position with unsqueeze: ', position)
    div_term = torch.exp(torch.arange(0, d_model, 2) * -(math.log(n) / d_model))
    pe[:, 0::2] = torch.sin(position * div_term)
    pe[:, 1::2] = torch.cos(position * div_term)
    # add a dimension
    # 增加一个维度
    pe = pe.unsqueeze(0)
    # the output has a shape of (1, max_length, d_model)
    return pe

# encodings = gen_pe(max_length=max_length, d_model=d_model, n=n)

# print('encodings:', encodings)
# print('encodings.shape:', encodings.shape)
# select the first six tokens
# seq_length = embeddings.shape[1]
# print('encodings: ',encodings[:, :seq_length])
# print(embeddings + encodings[:, :seq_length])
# encodings[:seq_length]
# print('seq_length', seq_length)
# print(encodings[:seq_length])

#-----使用pytorch里面的框架-----
# 下面的代码是官方文档的代码
class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, dropout: float = 0.1, max_length: int = 5000):
        """
        Args:
            d_model: dimension of embeddings
            dropout: randomly zeroes-out some of the input
            max_length: max sequence length
        """
        # inherit from Module
        super().__init__()

        # initialize dropout
        self.dropout = nn.Dropout(p=dropout)

        # create tensor of 0s
        pe = torch.zeros(max_length, d_model)

```

```

# create position column
k = torch.arange(0, max_length).unsqueeze(1)

# calc divisor for positional encoding
div_term = torch.exp(
    torch.arange(0, d_model, 2) * -(math.log(10000.0) / d_model)
)

# calc sine on even indices
pe[:, 0::2] = torch.sin(k * div_term)

# calc cosine on odd indices
pe[:, 1::2] = torch.cos(k * div_term)

# add dimension
pe = pe.unsqueeze(0)

# buffers are saved in state_dict but not trained by the optimizer
self.register_buffer("pe", pe)

def forward(self, x: Tensor):
    """
    Args:
        x: embeddings (batch_size, seq_length, d_model)

    Returns:
        embeddings + positional encodings (batch_size, seq_length, d_model)
    """
    # add positional encoding to the embeddings
    x = x + self.pe[:, :x.size(1)].requires_grad_(False)

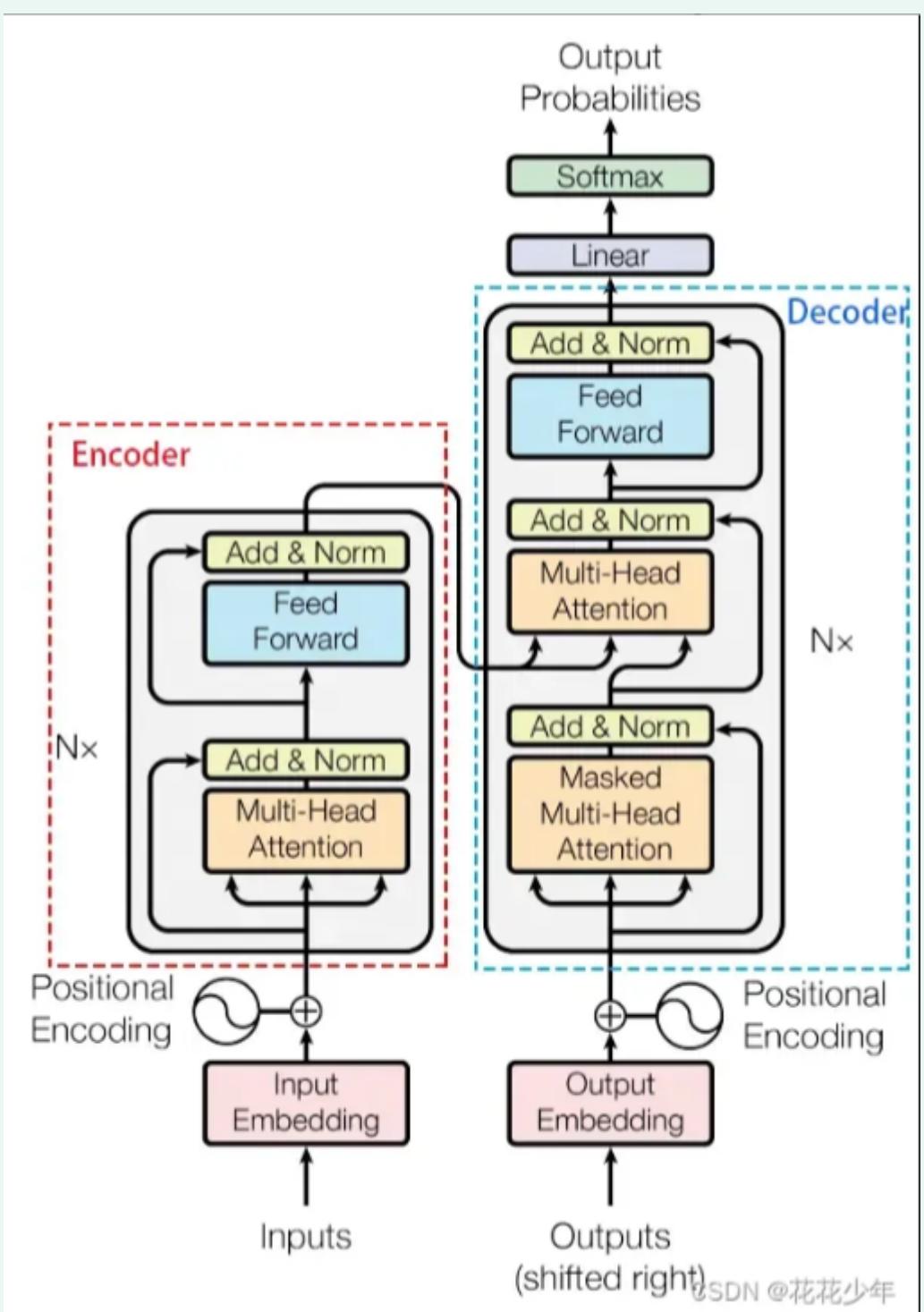
    # perform dropout
    return self.dropout(x)

pe = PositionalEncoding(d_model=d_model, dropout=0.1, max_length=max_length)
# print(pe.state_dict())
# print(pe(embeddings))
pe(embeddings)

```

总结

3.2.3 Encoder(编码器)



Encoder包括:

- Multi-Head Attention(多头注意力机制)
- residual connection(残差连接)
- layer normalization(层归一化)
- Feed Forward connection (全反馈神经网络连接)

1. Multi-Head Attention

参照前面Self-Attention那一章

代码实现

总结

2. residual connection(残差连接)

代码实现

总结

3.layer normalization(层归一化)

代码实现

总结

4.FNN(全连接前馈神经网络)

代码实现

总结