三维数据空间维度转换

Dezeming Family

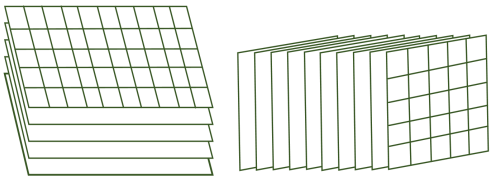
2022 年 6 月 28 日

DezemingFamily 系列文章和电子书**全部都有免费公开的电子版**，可以很方便地进行修改和重新发布。如果您获得了 DezemingFamily 的系列电子书，可以从我们的网站 [https://dezeming.top/] 找到最新的版本。对文章的内容建议和出现的错误也欢迎在网站留言。

目录

一　目的

我们用到的三维数据，比如医学影像数据，是许多张数据构成的，其中每张数据的大小是长 × 宽。而 CT 图像的拍摄顺序是不太一样的，比如有些数据看起来是对人体从上到下扫描，有些数据看起来是对人体从左到右扫描，我们希望将一种形式转换到另外一种：

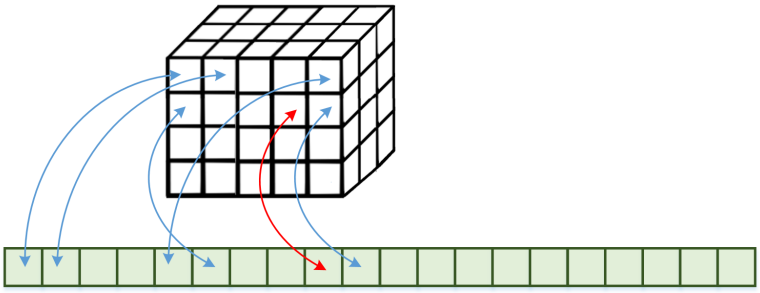


上图左，长 10，宽 5，高为 5；上图右，长 5，宽 5，高为 10。这里的高表示图像的数量。

我们默认数据在内存中的存储是一维的，三维只是我们理解和访问的方式。

二　定义 stride

设 $5 \times 5 \times 3$ 的三维数据，为了访问其中的某个数据，我们需要定义 stride，stride 表示数据索引与其在数组中位置的关系，对于坐标 $c(x, y, z)$，计算式为 $x * stride[0] + y * stride[1] + z * stride[2]$。比如下面的 stride 设置为 (1,5,20)：



其在内存中的排列形式是（此时，图像长为 5，宽为 4，高为 3）：

```
第一张图像的第一行，第一张图像的第二行，第一张图像的第三行，第一张图像的第四行，第二张图像的第一行，第二张图像的第二行，第二张图像的第三行，第二张图像的第四行，第三张图像的第一行，第三张图像的第二行，第三张图像的第三行，第三张图像的第四行
```

每行有 5 个数据，总共 60 个数据。

这里默认竖着 5×4 大小的图像是第一张，对于坐标 $c(3, 1, 0)$（红色箭头位置），对应到 stride 的计算是：

```
3*stride[0]+1*stride[1]+0*stride[2]=8
```

如果我们想交换数据的维度，只交换 stride 即可。比如还是按照当前的内存排列形式，我们把 stride 定义为 (1,20,5)，注意此时数据在一维空间中还是像上面那样排列的，只是对于访问坐标 c 来说，之前的 $(3, 1, 0)$ 位置变为了 $(3, 0, 1)$。

三　数据转换思路

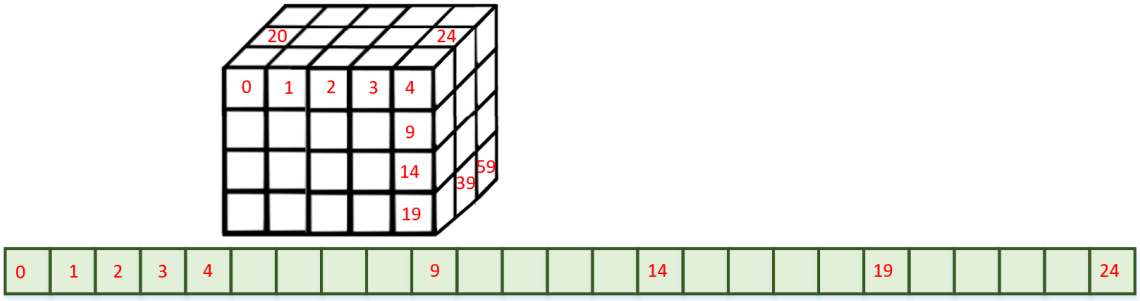
首先定义坐标访问方式（cord 表示坐标）：

```
// stride:int[3]  cord:int[3]
int getIndex(int*stride, int*cord) {
    return cord[0] * stride[0] + cord[1] * stride[1] + cord[2] * stride[2];
}
```

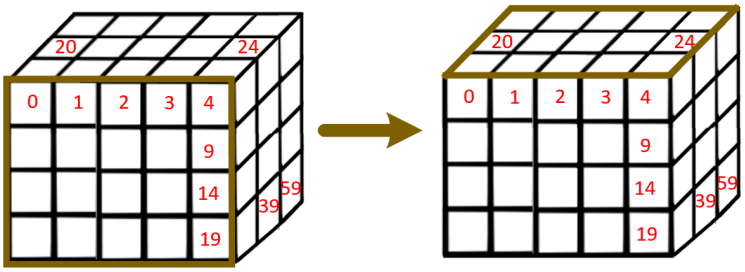
现在我们希望数据能换一种排列方式。我们构造数据：

```
int arr1[60];
int shape_arr1[3] = { 5,4,3 };
int stride_arr1[3] = { 1,5,20 };
for (int i = 0; i < 60; i++) arr1[i] = i;
```

排列示意图如下：



我们希望数据横着作为第一张，也就是如下转换：



转换后的内存形式是：

转换的程序如下：

```
int arr2[60];
int shape_arr2[3] = { 5,3,4 };
for (int i = 0; i < 5; i++)
    for (int j = 0; j < 3; j++)
        for (int k = 0; k < 4; k++) {
            // arr2的坐标
            int offset2 = i + j * 5 + k * 15;
            int cord1[3] = {i,k,j};
            arr2[offset2] = arr1[getIndex(stride_arr1, cord1)];
        }
```

注意 arr2 的坐标形式：

```
i*1 + j*shape_arr2[0] + k*shape_arr2[0]*shape_arr2[1]
```

生成后，stride_arr2 应当是 (1,5,15)。即 (1,shape_arr[0],shape_arr[0]*shape_arr[1])。我们可以看到：shape_arr[2]>shape_arr[1]>shape_arr[0]。

四　更一般的思路

我们定义交换顺序 permute。比如：

```
int permute[3] = {0,2,1};
```

表示交换后的顺序中，第 0 个维度保持不变，第 2 个维度移动到维度 1，第 1 个维度移动到维度 2。也就是把维度 1 和维度 2 交换一下顺序。

还是用前面的例子，设三维数据一共 3 张，每张图像有 4 行 5 列。排列顺序依旧是：

```
第一张图像的第一行，第一张图像的第二行，第一张图像的第三行，第一张图像的第四行，第二张图像的第一行，第二张图像的第二行，第二张图像的第三行，第二张图像的第四行，第三张图像的第一行，第三张图像的第二行，第三张图像的第三行，第三张图像的第四行
```

定义维度数组：

```
int width = 5;
int height = 4;
int imageNum = 3;
int dimM[3] = { width, height, imageNum };
```

然后，生成转换以后的维度数组 dimM_new：

```
for (int i = 0; i < 3; i++) {
    dimM_new[i] = dimM[permute[i]];
}
```

定义原三维数据的 stride：

```
int stride[3] = {1, width, width * height};
```

注意 arr2 是我们要生成的目标：

```
for (int i = 0; i < dimM_new[0]; i++) {
    for (int j = 0; j < dimM_new[1]; j++) {
        for (int k = 0; k < dimM_new[2]; k++) {
            int offsetnew = i + j * dimM_new[0] + k * dimM_new[0] * dimM_new[1];
            int offset = i * stride[permute[0]] + j * stride[permute[1]] + k * stride[permute[2]];
            arr2[offsetnew] = arr1[offset];
        }
    }
}
```

注意上面 offset 和 offsetnew 的计算方式。在 offset 的计算需要重返原维度的 stride；offsetnew 则是根据新的维度数组来计算位置。

五　连续 stride 与非连续 stride

我们的数据访问方式有时候是（第几张图像, 第几行，第几个元素），比如 (i,j,k) 表示第 i 张图像的第 j 行的第 k 个元素。这种序号方式与前面相反，注意前面的 (i,j,k) 应该读作“第 j 行的第 i 个元素，在第 k 个图像上”。

根据排列来说，由于 stride[2]>stride[1]>stride[0]，我们就称之为连续 stride；否则，比如 stride(1,20,5) 就是非连续 stride。

参考文献

[1] https://minitorch.github.io/module2.html