

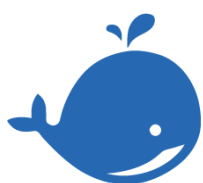
第六课-卷积和池化算子的实现

本课程赞助方：`Datawhale`

作者：傅莘莘，陈焕

主项目：<https://github.com/zjhelloworld/KuiperInfer> 欢迎大家点赞和PR.

课程代码：<https://github.com/zjhelloworld/kuiperdatawhale/course4>



Datawhale

KuiperInfer

上节课程回顾

在上一节课中，我带大家实现了算子的注册工厂和实例化，并且教大家如何实现了第一个简单的算子 `ReLU`。在上节课的基础上，本节课我将继续带着大家实现两个常见的算子——池化算子和卷积算子，它们在卷积网络中非常常见。

池化算子的定义

池化算子常用于缓解深度神经网络对位置的过度敏感性。

池化算子会在固定形状的窗口（即池化窗口）内对输入数据的元素进行计算，计算结果可以是池化窗口内元素的最大值或平均值，这种运算被称为最大池化或平均池化。

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 2 | 3 | 4 | 5 |
| 3 | 4 | 5 | 6 |
| 4 | 5 | 6 | 7 |
| | | | |
| 3 | 5 | | |
| 5 | 7 | | |

以上图为例，黄色区域表示一个 2×2 的池化窗口。在二维最大池化中，池化窗口从输入张量的左上角开始，按照从左往右、从上往下的顺序依次滑动（滑动的幅度称为 `stride`）。在每个池化窗口内，取窗口中的最大值或平均值作为输出张量相应位置的元素。

本例中采用最大池化，即每次都取窗口中的最大值，滑动的 `stride` 等于2。当时刻 t 等于2或3时，池化窗口的位置如下所示：

| | | | | | |
|---|---|---|---|-----|--|
| 1 | 2 | 3 | 4 | | |
| 2 | 3 | 4 | 5 | t=2 | |
| 3 | 4 | 5 | 6 | | |
| 4 | 5 | 6 | 7 | | |
| | | | | | |
| 1 | 2 | 3 | 4 | | |
| 2 | 3 | 4 | 5 | | |
| 3 | 4 | 5 | 6 | t=3 | |
| 4 | 5 | 6 | 7 | | |

当t等于2时，池化窗口的输出为5, 也就是窗口内最大的元素为5. 当t等于3时，池化窗口向左下移动，池化窗口的输出同样是5. 所以，该输入特征图的最大池化计算结果输出如下，不难看出，输出中的每个位置都对应于原始输入窗口中的最大值。

| | |
|---|---|
| 3 | 5 |
| 5 | 7 |

每次进行池化操作时，池化窗口都按照从左到右、从上到下的顺序进行滑动。窗口每次向下移动的步长为 `stride height`，向右移动的步长为 `stride width`. 池化操作的元素数量由 `pooling height` 和 `pooling width` 所组成的池化窗口决定。

根据图示，可知在本例中 `stride width` 的值为2，即窗口每次向右移动的距离为2个元素大小。我们在下图中展示了一个 `stride = 1` 的情况，即池化窗口每次滑动一个元素的位置。这意味着在进行池化操作时，池化窗口每次向下移动和向右移动的步长均为1个元素的大小，请参考下图以获得更直观的理解。

| | | | |
|----------|---|---|---|
| stride=1 | | | |
| | | | |
| 1 | 2 | 3 | 4 |
| 2 | 3 | 4 | 5 |
| 3 | 4 | 5 | 6 |
| 4 | 5 | 6 | 7 |

对于不带填充的池化算子，输入大小和输出大小之间有如下的等式关系：

$$output\ size = floor(\frac{input - pooling\ size}{stride} + 1)$$

对填充后的输入特征图求池化

在池化算子中，通常会先对输入特征进行填充(padding), 当padding的值等于2时，池化算子会先在输入特征的周围填充**2圈最小值元素**，然后再计算其对应的池化值。

| | | | | | | | |
|------|------|------|------|------|------|------|------|
| | | 1 | 2 | 3 | 4 | | |
| | | 2 | 3 | 4 | 5 | | |
| | | 3 | 4 | 5 | 6 | | |
| | | 4 | 5 | 6 | 7 | | |
| | | | | | | | |
| -inf | -inf | -inf | -inf | -inf | -inf | -inf | -inf |
| -inf | -inf | -inf | -inf | -inf | -inf | -inf | -inf |
| -inf | -inf | 1 | 2 | 3 | 4 | -inf | -inf |
| -inf | -inf | 2 | 3 | 4 | 5 | -inf | -inf |
| -inf | -inf | 3 | 4 | 5 | 6 | -inf | -inf |
| -inf | -inf | 4 | 5 | 6 | 7 | -inf | -inf |
| -inf | -inf | -inf | -inf | -inf | -inf | -inf | -inf |
| -inf | -inf | -inf | -inf | -inf | -inf | -inf | -inf |

对于带填充的池化算子，输出特征图的大小和输入特征图的大小之间有以下等式关系：

$$output\ size = floor(\frac{input\ size + 2 \times padding - pooling\ size}{stride} + 1)$$

多通道的池化算子定义

多通道的池化算子和单通道上的池化算子实现方式基本相同，只不过多通道池化需要对输入特征中的多个通道进行池化运算。

| | | | | | |
|---|---|---|---|--|-----------|
| 1 | 2 | 3 | 4 | | channel11 |
| 2 | 3 | 4 | 5 | | |
| 3 | 4 | 5 | 6 | | |
| 4 | 5 | 6 | 7 | | |
| | | | | | |
| 1 | 2 | 3 | 4 | | channel12 |
| 3 | 4 | 4 | 5 | | |
| 3 | 4 | 5 | 6 | | |
| 4 | 5 | 6 | 7 | | |
| | | | | | |
| 1 | 2 | 3 | 4 | | channel13 |
| 2 | 3 | 4 | 5 | | |
| 3 | 4 | 5 | 6 | | |
| 4 | 5 | 6 | 7 | | |

我们对通道数量为3的输入特征图进行最大池化操作时，使用滑动步长 (`stride`)为1和窗口大小为2. 下图以第3个通道为例进行说明。

| | | | | | | |
|-------|---|---|---|---|--|-----|
| max=3 | 1 | 2 | 3 | 4 | | |
| | 2 | 3 | 4 | 5 | | t=1 |
| | 3 | 4 | 5 | 6 | | |
| | 4 | 5 | 6 | 7 | | |
| | | | | | | |
| max=4 | 1 | 2 | 3 | 4 | | |
| | 2 | 3 | 4 | 5 | | t=2 |
| | 3 | 4 | 5 | 6 | | |
| | 4 | 5 | 6 | 7 | | |
| | | | | | | |
| max=5 | 1 | 2 | 3 | 4 | | |
| | 2 | 3 | 4 | 5 | | t=3 |
| | 3 | 4 | 5 | 6 | | |
| | 4 | 5 | 6 | 7 | | |
| | | | | | | |
| max=4 | 1 | 2 | 3 | 4 | | |
| | 2 | 3 | 4 | 5 | | t=3 |
| | 3 | 4 | 5 | 6 | | |
| | 4 | 5 | 6 | 7 | | |

在进行池化操作时，在不同的时刻，每次池化窗口的移动大小均为1。同时，窗口滑动需要按照先左后右、先上后下的顺序遍历整个输入通道。这保证了在求取最大池化值时，覆盖到所有可能的位置，并确保结果的准确性，同样的方式也适用于处理其他两个通道的数据。

最大池化算子的实现

最大池化指的是在一个窗口的范围内，取所有元素的最大值

最大池化算子的源代码位于 `course6` 文件夹下的 `maxpooling.cpp` 文件中，我们会逐一对其实现进行讲解。

在上节课中，我们讲到所有算子都会派生于 `Layer` 父类并重写其中带参数的 `Forward` 方法，子类的 `Forward` 方法会包含派生类算子的具体计算过程，一般包括以下几个步骤

1. 逐一从输入数组中提取输入张量，并对其进行空值和维度检查；
2. 根据算子的定义和输入张量的值，执行该算子的计算；
3. 将计算得到的结果写回到输出张量中。

```
1 InferStatus MaxPoolingLayer::Forward(  
2     const  
   std::vector<std::shared_ptr<Tensor<float>>>& inputs,  
3     std::vector<std::shared_ptr<Tensor<float>>>&  
   outputs) {  
4     if (inputs.empty()) {  
5         LOG(ERROR) << "The input tensor array in the max  
   pooling layer is empty";  
6         return InferStatus::kInferFailedInputEmpty;  
7     }  
8  
9     if (inputs.size() != outputs.size()) {  
10        LOG(ERROR)  
11            << "The input and output tensor array size  
   of the max pooling layer "  
12            "do not match";  
13        return  
   InferStatus::kInferFailedInputOutSizeMatchError;  
14    }
```

在以上的代码的第4行中，判断输入的张量数组是否为空，如果为空则返回对应的错误码。同样在第9行中，如果发现输入和输出的张量个数不相等，也返回相应的错误码。值得说明的是，我们将输入和输出的张量个数记作 `batch size`，也就是一个批次处理的数据数量。

```
1     for (uint32_t i = 0; i < batch; ++i) {
```

```

2     const std::shared_ptr<Tensor<float>>& input_data
    = inputs.at(i);
3     ...
4     ...
5     const uint32_t input_h = input_data->rows();
6     const uint32_t input_w = input_data->cols();
7     const uint32_t input_padded_h = input_data-
    >rows() + 2 * padding_h_;
8     const uint32_t input_padded_w = input_data-
    >cols() + 2 * padding_w_;
9
10    const uint32_t input_c = input_data->channels();
11
12    const uint32_t output_h = uint32_t(
13        std::floor((int(input_padded_h) -
    int(pooling_h)) / stride_h_ + 1));
14    const uint32_t output_w = uint32_t(
15        std::floor((int(input_padded_w) -
    int(pooling_w)) / stride_w_ + 1));
16

```

在以上的代码中，我们对 `batch size` 个输入张量进行遍历处理。在第5 - 6行的代码中，我们获得该输入张量的高度和宽度，并在12 - 15行代码中，我们根据以下的公式对池化的输出大小进行计算。

$$output\ size = floor(\frac{input\ size + 2 \times padding - pooling\ size}{stride} + 1)$$


```

1  for (uint32_t ic = 0; ic < input_c; ++ic) {
2      const arma::fmat& input_channel = input_data-
>slice(ic);
3      arma::fmat& output_channel = output_data-
>slice(ic);
4      for (uint32_t c = 0; c < input_padded_w -
pooling_w + 1; c += stride_w_) {
5          for (uint32_t r = 0; r < input_padded_h -
pooling_h + 1;
6              r += stride_h_) {
7              循环中的内容
8          }
9      }

```

第1行是对多通道的输入特征图进行逐通道的处理，每次进行一个通道上的最大池化操作，首先获取当前输入张量上的第 `ic` 个通道 `input_channel`。

在第4 - 5行的代码中，我们在输入特征图中进行窗口移动（就如同我们上面的图示一样），每次纵向移动的步长为 `stride_h`，每次横向移动的步长为 `stride_w`。

```

1  float* output_channel_ptr =
    output_channel.colptr(int(c / stride_w_));
2  float max_value =
    std::numeric_limits<float>::lowest();
3  for (uint32_t w = 0; w < pooling_w; ++w) {
4      for (uint32_t h = 0; h < pooling_h; ++h) {
5          float current_value = 0.f;
6          if ((h + r >= padding_h_ && w + c >=
padding_w_) &&
7              (h + r < input_h + padding_h_ &&
8              w + c < input_w + padding_w_)) {
9              const float* col_ptr =
input_channel.colptr(c + w - padding_w_);
10             current_value = *(col_ptr + r + h -
padding_h_);

```

```

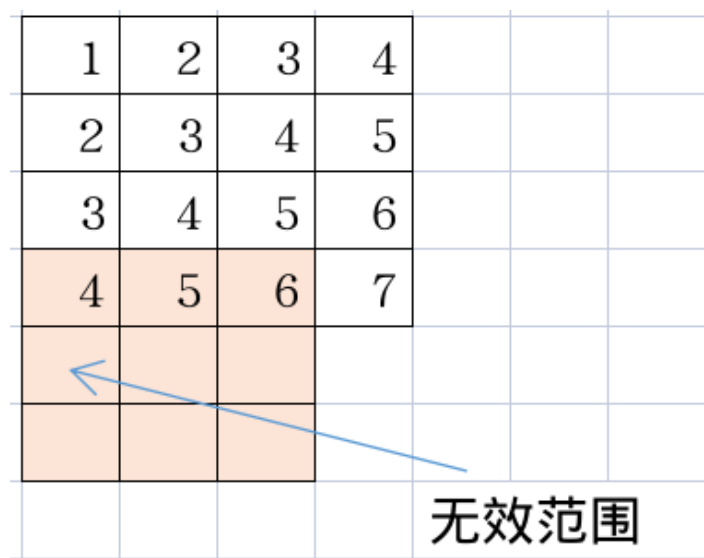
11         } else {
12             current_value =
std::numeric_limits<float>::lowest();
13         }
14         max_value = max_value > current_value ?
max_value : current_value;
15     }
16 }
17 *(output_channel_ptr + int(r / stride_h_)) =
max_value;

```

在第3 - 4行的内部循环中，我们对一个窗口内的 `pooling_h` × `pooling_w` 个元素求得最大值。在这个内部循环中，有两种情况（使用 `if` 判断的地方）：

- 第一种情况(第6行)是当前遍历的元素位于有效范围内，我们将该元素的值记录下来；
- 第二种情况(第12行)是当前遍历的元素超出了输入特征图的范围，在这种情况下，我们将该元素的值赋值为一个最小值。

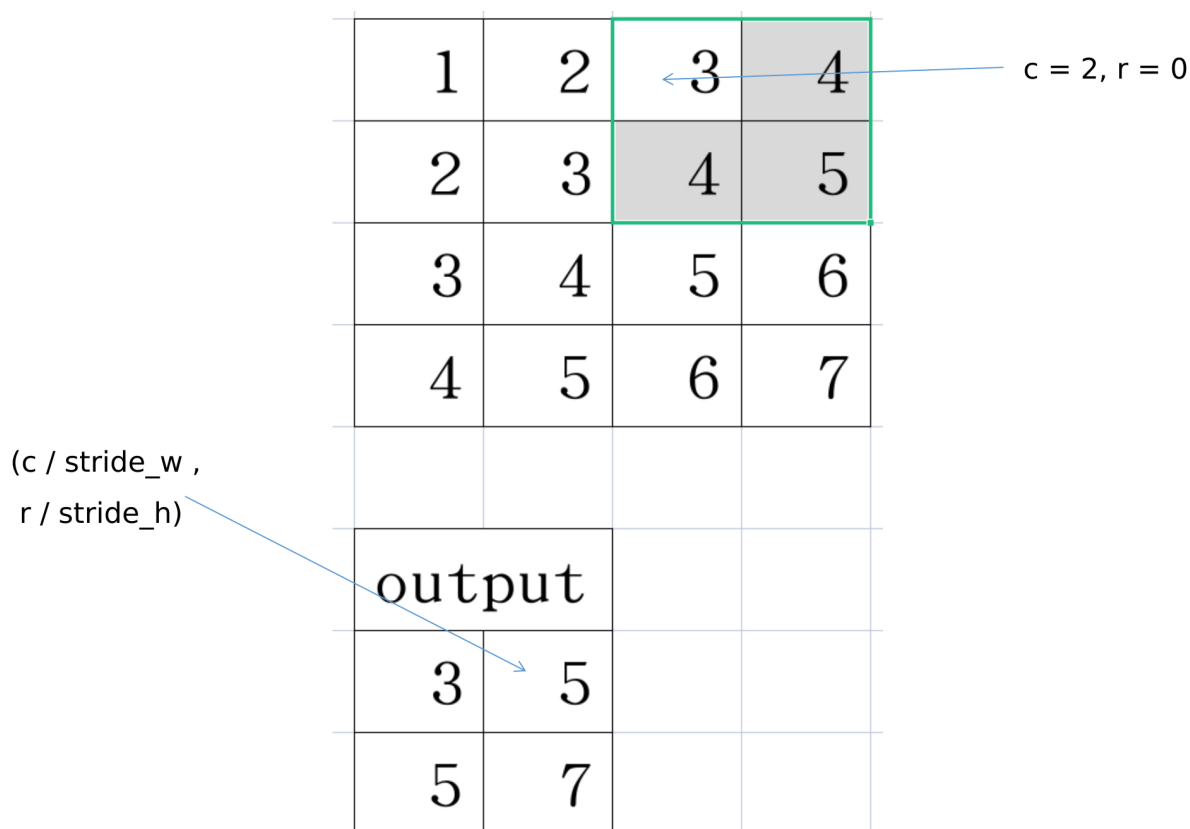
以下是关于无效值范围的图示。我们可以观察到，在一个窗口大小为3的池化算子中，当该窗口移动到特定位置时，就会出现无效范围，即超出了输入特征图的尺寸限制。



在内部循环中，当获取到窗口的最大值后，我们需要将该最大值写入到对应的输出张量中。以下是输出张量写入的位置索引：

1. `output_data->slice(ic);` 获得第 `ic` 个输出张量;
2. `output_channel.colptr(int(c / stride_w));` 计算第 `ic` 个张量的**输出列位置**，列的值和当前窗口的位置有关，`c` 表示滑动窗口当前所在的列数。
3. `*(output_channel_ptr + int(r / stride_h_)) = max_value;` 计算输出的行位置，行的位置同样与当前窗口的滑动有关，`h` 表示当前滑动窗口所在的行数。

我们以下面的图示作为一个例子来讲解对 `output` 的输出位置：



在以上情况中，窗口的滑动步长为2，窗口此时所在的列 `c` 等于2，窗口所在的行 `r` 等于0。通过观察下面的输出张量，我们可以看出，在该时刻要写入的输出值为5，其写入位置的计算方法如下：

1. 写入位置的列 $= c / stride_w = 2 / 2 = 1$
2. 写入位置的行 $= r / stride_h = 0 / 2 = 0$

所以对于该时刻，池化窗口求得的最大值为5，写入的位置为 `output(0, 1)`。

池化算子的注册

我们在 `maxpooling.cpp` 的最后，使用上节课中提到的算子注册类，将最大池化的初始化过程注册到了推理框架中。接下来我们来看一下最大池化算子的初始化过程 `MaxPoolingLayer::GetInstance.`

```
1 LayerRegistererWrapper
  kMaxPoolingGetInstance("nn.MaxPool2d",
2
  MaxPoolingLayer::GetInstance);
```

池化算子的实例化函数

```
1 ParseParameterAttrStatus
  MaxPoolingLayer::GetInstance(
2     const std::shared_ptr<RuntimeOperator>& op,
3     std::shared_ptr<Layer>& max_layer) {
4     ...
5     const std::map<std::string,
  std::shared_ptr<RuntimeParameter>>& params =
6         op->params;
7 }
```

传入参数之一是 `RuntimeOperator`，回顾前两节课，这个数据结构中包含了算子初始化所需的参数和权重信息，我们首先访问并获取该算子中的所有参数 `params`。

```

1  if (params.find("stride") == params.end()) {
2      LOG(ERROR) << "Can not find the stride
parameter";
3      return
ParseParameterAttrStatus::kParameterMissingStride;
4  }
5
6      auto stride =
7
std::dynamic_pointer_cast<RuntimeParameterIntArray>
(params.at("stride"));
8      if (!stride) {
9          LOG(ERROR) << "Can not find the stride
parameter";
10         return
ParseParameterAttrStatus::kParameterMissingStride;
11     }
12     ...

```

在以上的代码中，我们获取到了用于初始化池化算子的 `stride` 参数，即窗口滑动步长。如果参数中没有 `stride` 参数，则返回对应的错误代码。

```

1   if (params.find("padding") == params.end()) {
2       LOG(ERROR) << "Can not find the padding
padding
parameter";
3       return
ParseParameterAttrStatus::kParameterMissingPadding;
4   }
5
6   auto padding =
7
std::dynamic_pointer_cast<RuntimeParameterIntArray>
(params.at("padding"));
8   if (!padding) {
9       LOG(ERROR) << "Can not find the padding
padding
parameter";
10      return
ParseParameterAttrStatus::kParameterMissingPadding;
11  }

```

对于获得算子其他的初始化（如以上的填充大小padding）参数同理，我们可以先判断params键值对中是否存在名为padding的键。如果存在，则进行访问。

```

1   auto kernel_size =
std::dynamic_pointer_cast<RuntimeParameterIntArray>(
2       params.at("kernel_size"));
3   if (!kernel_size) {
4       LOG(ERROR) << "Can not find the kernel size
padding
parameter";
5       return
ParseParameterAttrStatus::kParameterMissingKernel;
6   }
7   const auto& padding_values = padding->value;
8   const auto& stride_values = stride->value;
9   const auto& kernel_values = kernel_size->value;

```

在以上代码中，首先我们尝试获取 `params` 中的 `kernel size` 参数。如果该参数存在，则进行访问；如果不存在，则返回相应的错误码。随后，我们可以分别获取填充大小、步长大小和滑动窗口大小的参数值。

```
1  max_layer = std::make_shared<MaxPoolingLayer>(
2      padding_values.at(0), padding_values.at(1),
   kernel_values.at(0),
3      kernel_values.at(1), stride_values.at(0),
   stride_values.at(1));
```

最后，我们可以使用这些参数对池化算子进行初始化，并将其赋值给 `max_layer` 参数进行返回。

池化算子的单元测试

第1个单元测试的目的是测试该算子（池化算子）是否已经被成功注册到算子列表中。如果已经注册成功，则通过测试。

```
1  TEST(test_registry, create_layer_poolingforward) {
2      std::shared_ptr<RuntimeOperator> op =
   std::make_shared<RuntimeOperator>();
3      op->type = "nn.MaxPool2d";
4      std::vector<int> strides{2, 2};
5      std::shared_ptr<RuntimeParameter> stride_param =
   std::make_shared<RuntimeParameterIntArray>(strides);
6      op->params.insert({"stride", stride_param});
7
8      std::vector<int> kernel{2, 2};
9      std::shared_ptr<RuntimeParameter> kernel_param =
   std::make_shared<RuntimeParameterIntArray>(strides);
10     op->params.insert({"kernel_size", kernel_param});
11
12     std::vector<int> paddings{0, 0};
13     std::shared_ptr<RuntimeParameter> padding_param =
   std::make_shared<RuntimeParameterIntArray>
   (paddings);
14     op->params.insert({"padding", padding_param});
```

```

15
16     std::shared_ptr<Layer> layer;
17     layer = LayerRegisterer::CreateLayer(op);
18     ASSERT_NE(layer, nullptr);
19 }

```

在以上的代码中，我们将 `RuntimeOperator` 的类型设置为池化，将池化大小参数设置为 `3 x 3`，将填充大小设置为0，并将滑动窗口的步长设置为2。

设置完毕后，我们将使用 `CreateLayer` 函数传递该参数，并返回相应的池化算子。这一步骤已在上节课中进行了详细的讲解。

在下面的单元测试中，我们同样将池化窗口的大小设置为2，并且将滑动窗口的步长设置为2。同时，关于池化算子的输入值，我们做如下图的设置：

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 2 | 3 | 4 | 5 |
| 3 | 4 | 5 | 6 |
| 4 | 5 | 6 | 7 |
| | | | |
| 3 | 5 | | |
| 5 | 7 | | |

```

1  TEST(test_registry, create_layer_poolingforward_1) {
2      ...
3      ...
4      std::shared_ptr<Layer> layer;
5      layer = LayerRegisterer::CreateLayer(op);
6      ASSERT_NE(layer, nullptr);
7

```



```

8   sftensor tensor = std::make_shared<ftensor>(1, 4,
9   arma::fmat input = arma::fmat("1,2,3,4;"
10                                  "2,3,4,5;"
11                                  "3,4,5,6;"
12                                  "4,5,6,7");
13   tensor->data().slice(0) = input;
14   std::vector<sftensor> inputs(1);
15   inputs.at(0) = tensor;
16   std::vector<sftensor> outputs(1);
17   layer->Forward(inputs, outputs);
18
19   ASSERT_EQ(outputs.size(), 1);
20   outputs.front()->Show();
21 }

```

我们在 `layer->Forward(inputs, outputs)` 中使用初始化完毕的池化算子，对特定的输入值进行推理，池化算子的预测结果如下所示，可以看出符合我们之前的逻辑推导过程。

```

1 I20230721 14:05:09.855405 3224 tensor.cpp:201]
2     3.0000    5.0000
3     5.0000    7.0000

```

卷积算子的定义

卷积是信号处理和图像处理中常用的运算操作之一。它通过将输入信号（如图像、音频等）与一个卷积核（也称为滤波器或权重）进行相乘和累加的过程，用于在深度神经网络中提取特定的特征。因此，可以说卷积是最常用的算子之一。

- 在离散情况下，一维卷积操作可以定义为：

$$y[n] = \sum_{k=-\infty}^{\infty} x[k] \cdot h[n - k]$$

其中， x 表示输入信号， h 表示卷积核， y 表示输出信号。这个公式表示输出信号中的每个元素 $y[n]$ 是通过将输入信号 x 进行加权累加计算得到的，计算过程中使用了卷积核 h 确定的权重。

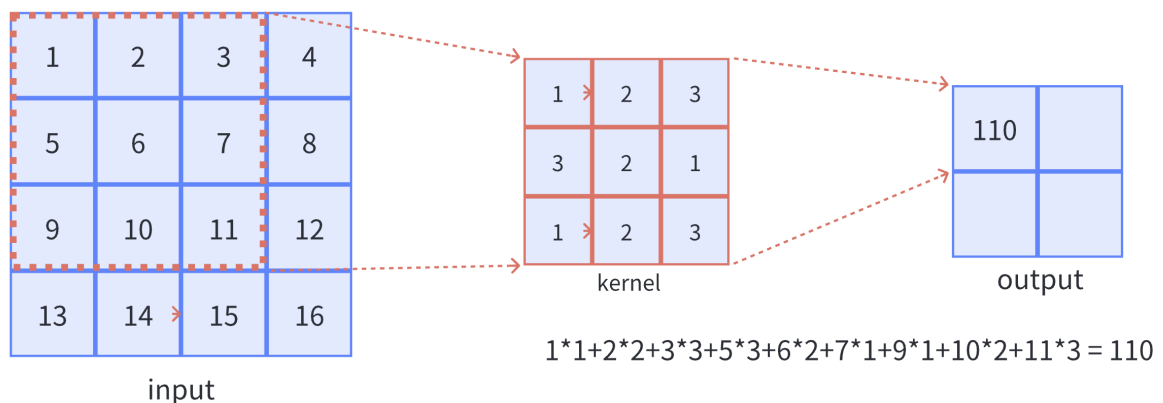
- 更多的场景中我们一般面对的是二维输入（多通道，此处以单通道为例），这时候我们只需要将卷积定义拓展为二维：

$$Y[i, j] = \sum_m \sum_n H[m, n] \cdot X[i + m, j + n]$$

其中， X 表示输入矩阵， H 表示卷积核， Y 表示输出矩阵， i 和 j 表示输出矩阵中的输出像素坐标， m 和 n 表示卷积核中的坐标， $i + m$ 和 $j + n$ 用于将卷积核和输入矩阵进行对齐，分别表示输入图像中的某个元素坐标。通过这两个偏移量，我们可以确定卷积核在输入矩阵中的位置，并将其与对应位置的像素值相乘，然后求和得到输出矩阵的每个元素 $Y[i, j]$ 。

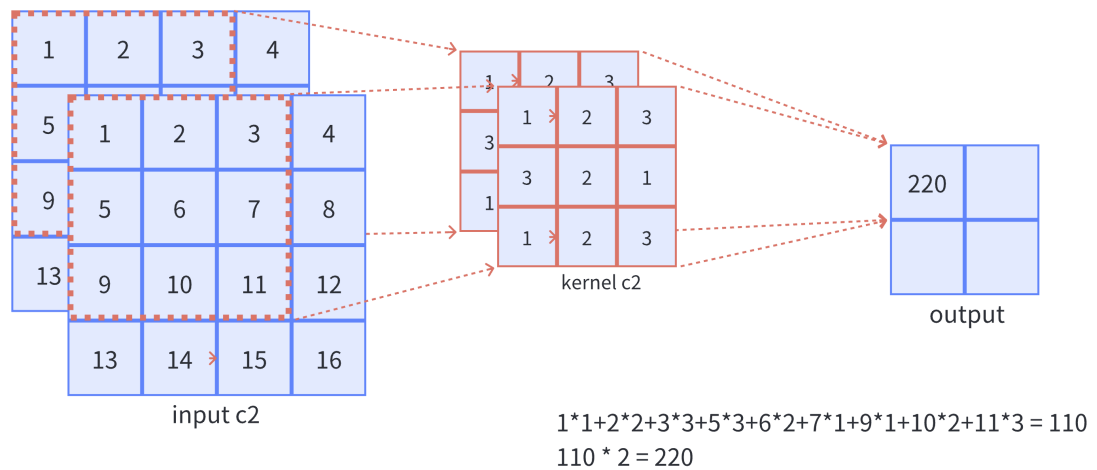
直观展示

- 为了直观解释，我们对二维卷积的计算过程进行直观展示，如下图，卷积核以滑动窗口的形式，从输入中划过，计算点积并求和，得到卷积后的输出存于 `output` 中。



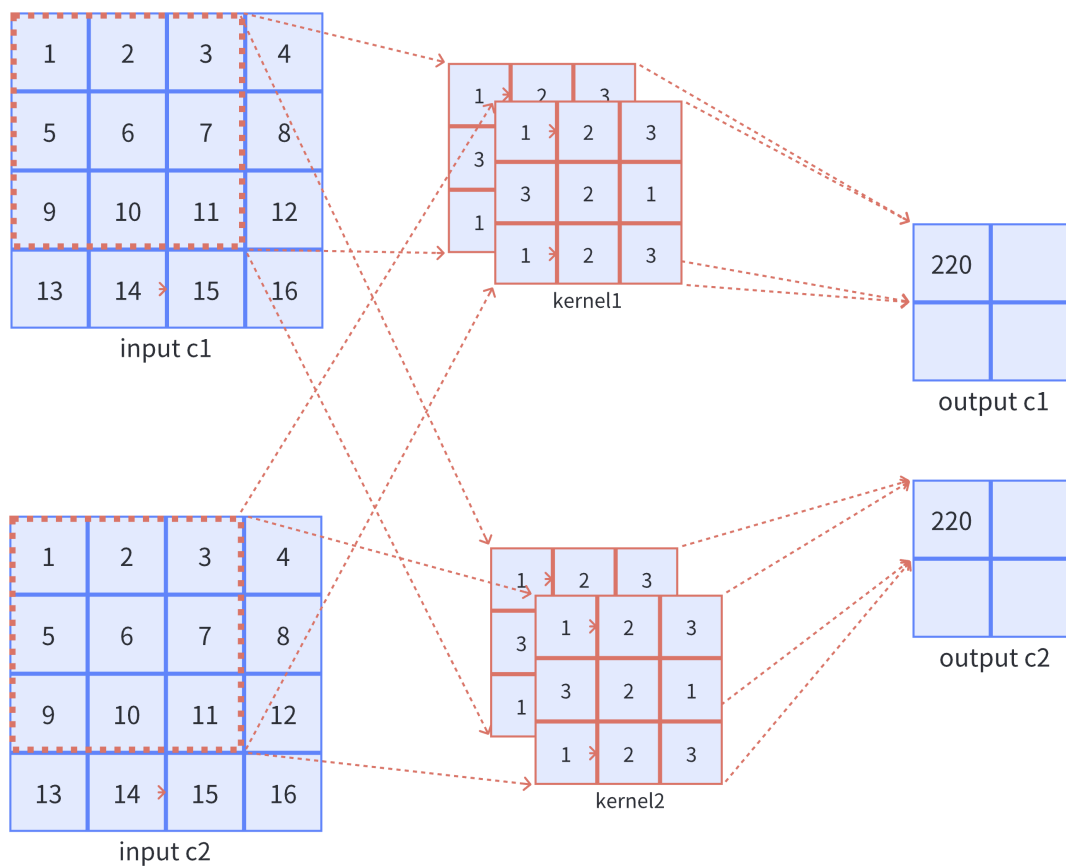
- 单通道可以直观地被拓展成多通道，只需对多个单通道的卷积结果求和即可（请注意，下图中的kernel属于同一个卷积核中的不同通道），此时需要注意的是输入的通道数与卷积核的通道数需要保持一致。

如下图所示，可以看到一个多通道的输入和一个多通道的卷积核进行卷积计算，最后得到了一个单通道的输出 `output`。

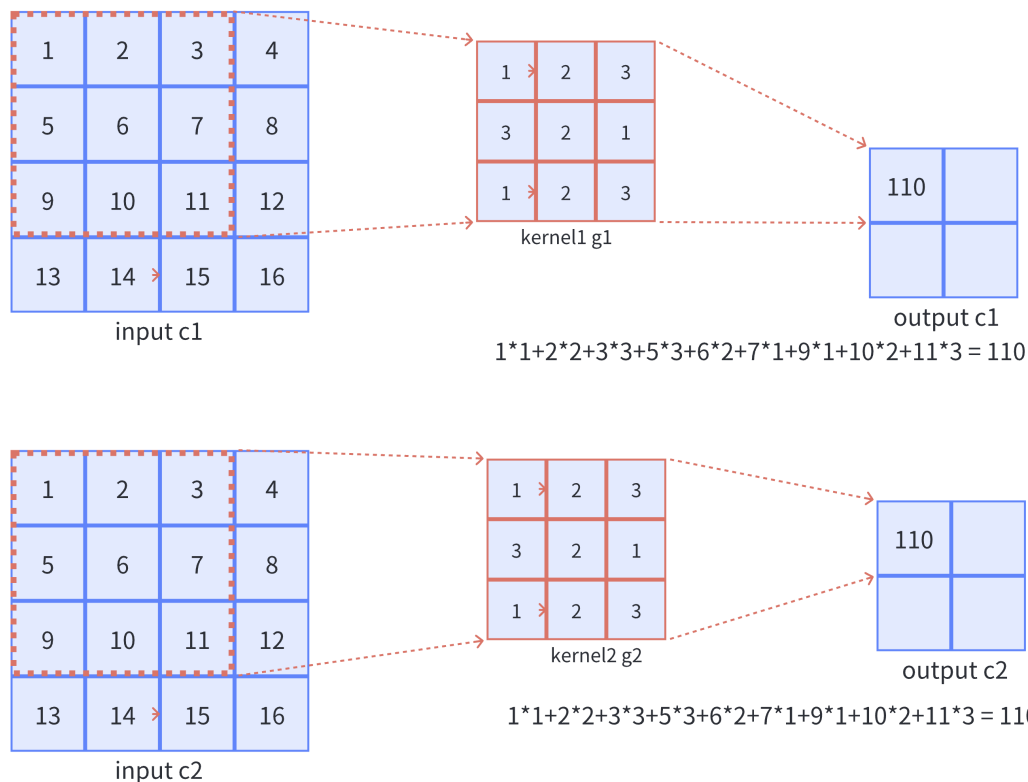


- 对于单通道输出，我们只需要一个卷积核就可以完成，如果想要使得输出为多通道，则需使用多个不同的卷积核，即卷积核个数对应输出通道个数。

如下图所示，可以看到，如果使用两个卷积核，最后会产生一个多通道的输出 **output**，它有两个通道，分别为 **c1** 和 **c2**。



- 组卷积（**group conv**），顾名思义就是将卷积分组，即在深度上进行分组，假设 $group=2$ ，则表示我们将原有的输入数据分成2组，如上图图所示，原本一个卷积核管全部通道，当分组之后，一个卷积核只需要管 $\frac{input\ channel}{group} = 2/2 = 1$ 个通道，即如下图所示。
- 分组卷积早在 **AlexNet** 便得到了应用，**Alex**认为组卷积能够增加卷积核之间的对角相关性，并减少训练参数，不容易过拟合，达到类似正则的效果。从下图可以看出，如果对一个多通道的输入运用组卷积，最后得到了一个多通道的输出 **output**，它有两个通道，分别为 **c1** 和 **c2**。



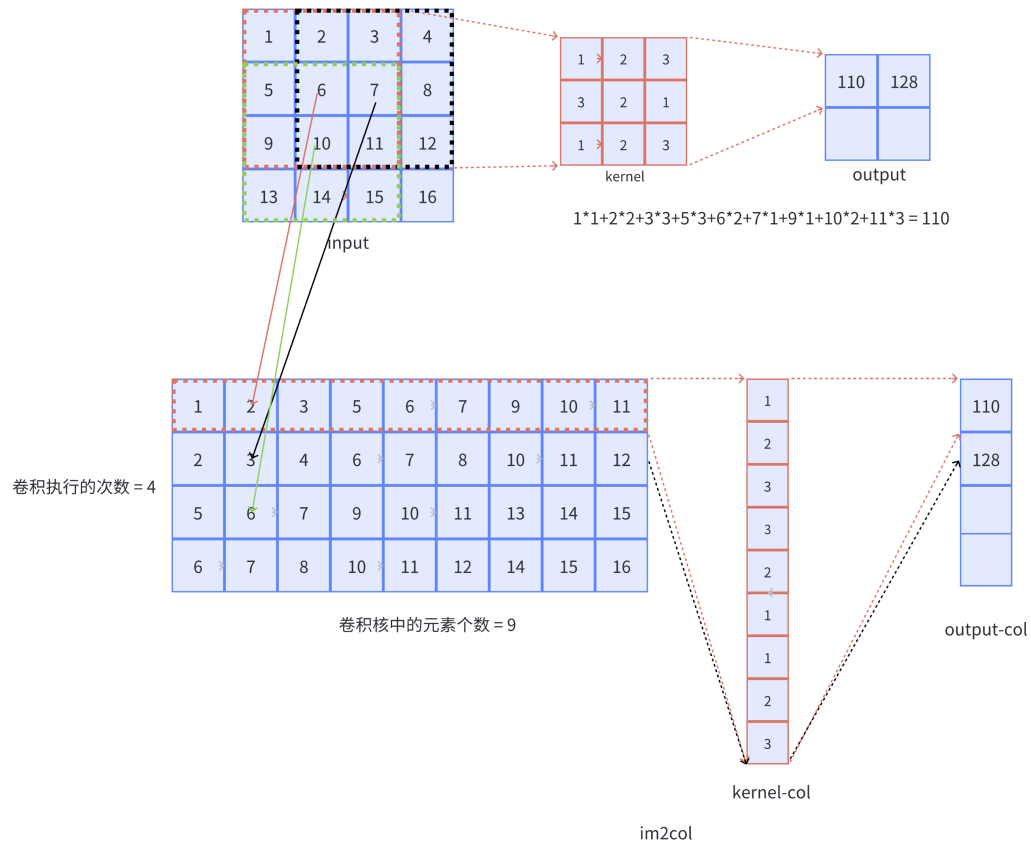
在本节中，我们描述了一维和二维卷积的基本定义，并通过直观地解释了二维卷积的运行过程。在这个过程中，我们发现卷积核的通道数需要与输入数据的通道数保持一致，而卷积核的数量则代表了输出数据的通道数。在卷积计算中，输入输出大小的维度有以下的对应关系：

$$output\ size = floor(\frac{input\ size + padding - kernel\ size}{stride} + 1)$$

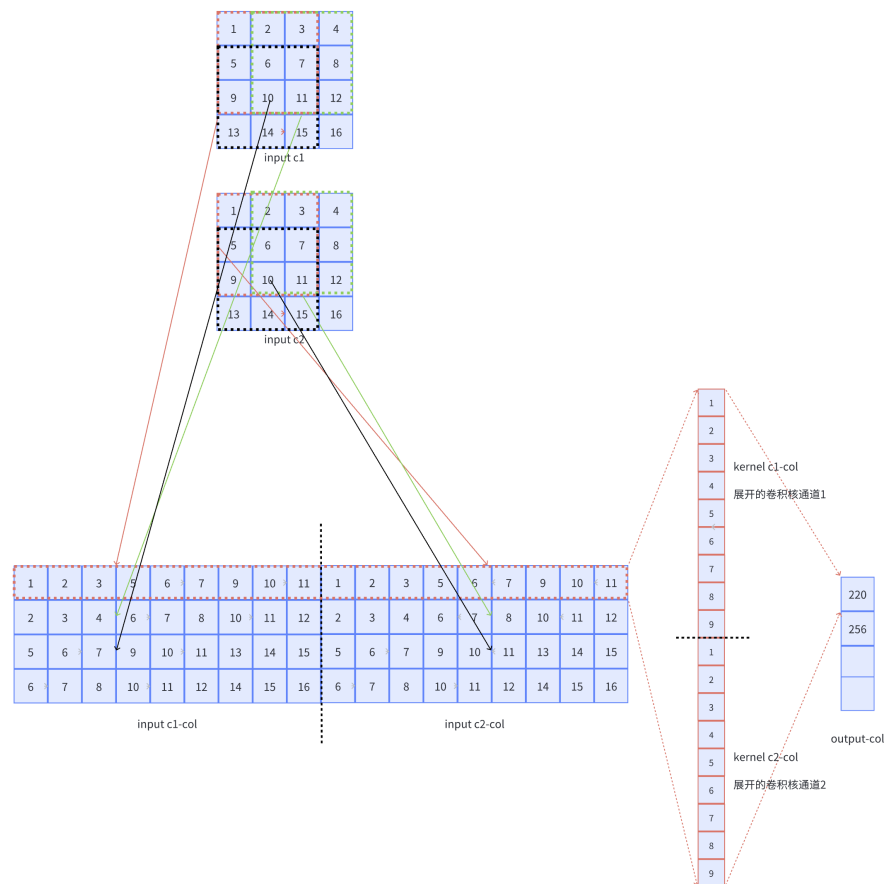
Im2col优化卷积计算

- **Im2col**的核心思想是将卷积计算转化成矩阵计算，利用现有较为成熟的矩阵加速方法实现卷积运算的加速
- 我们仍然从最简单的单通道开始：
 - 下图向我们展示单通道卷积的 **Im2col** 展开方式：对于输入，每行代表一个卷积核窗口扫过的区域，展开的行数为 **input-col**，也就是卷积总执行次数，每行的列数为一个卷积核元素和，本例中卷积核被展开为一个 4×9 的矩阵；

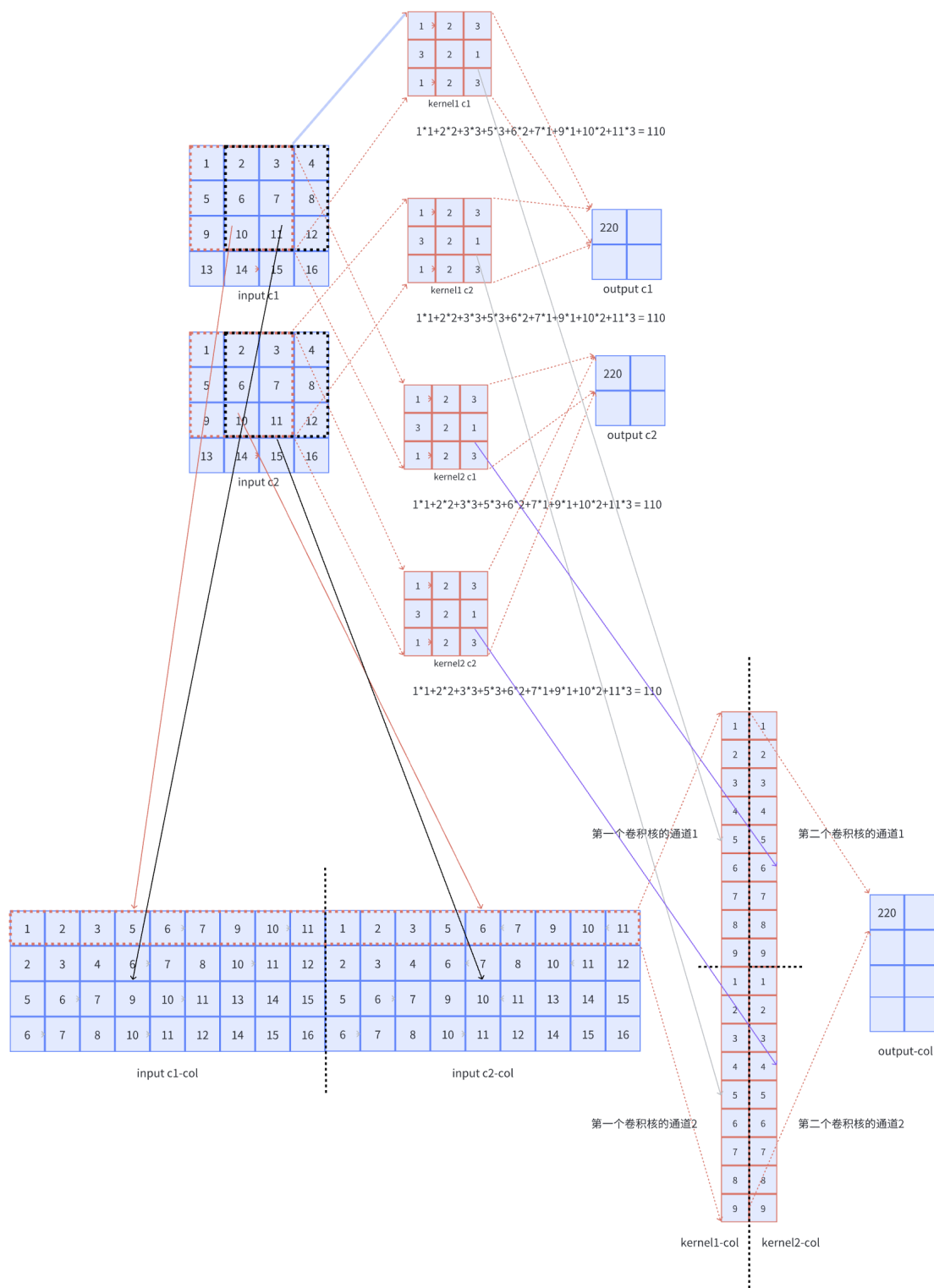
- 对于卷积核，一个卷积核展成一列，本例中卷积核被展开为一个 9×1 的向量；
- 如此，卷积就被成功转化成矩阵计算，我们来看下面的图示。



- 如果是对**多通道**的输入进行卷积计算呢？



- 同理，对于多通道输出(有多个卷积核的情况)，则有：



KuiperInfer中的分组卷积

我们假设 `group` 的数量为2, 如果输入特征图和卷积核的通道数为4, 共有4个卷积核.

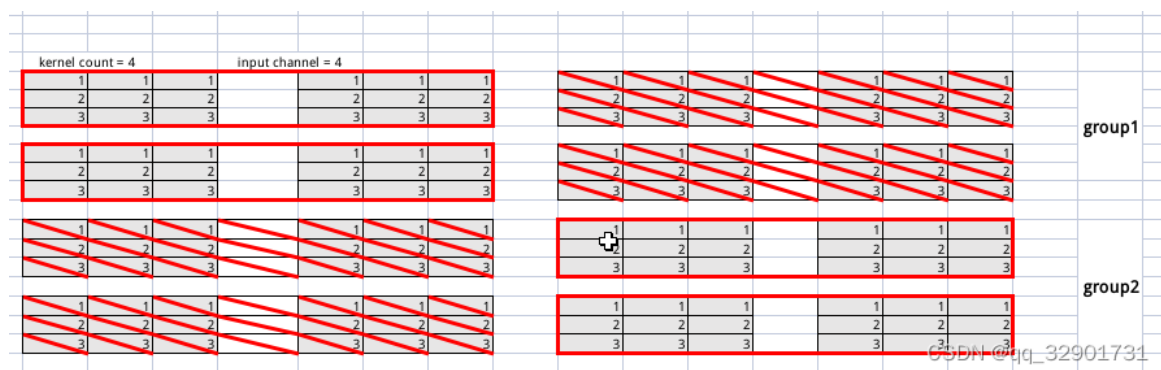
当 `group` 等于2时, 我们将输入特征图的4个通道分为2组, 同时将4个卷积核也分为2组, 每组分配到两个卷积核, 每个卷积核的通道数为2.

1. `group` 等于2, 我们将输入特征图的4个通道一分为2.

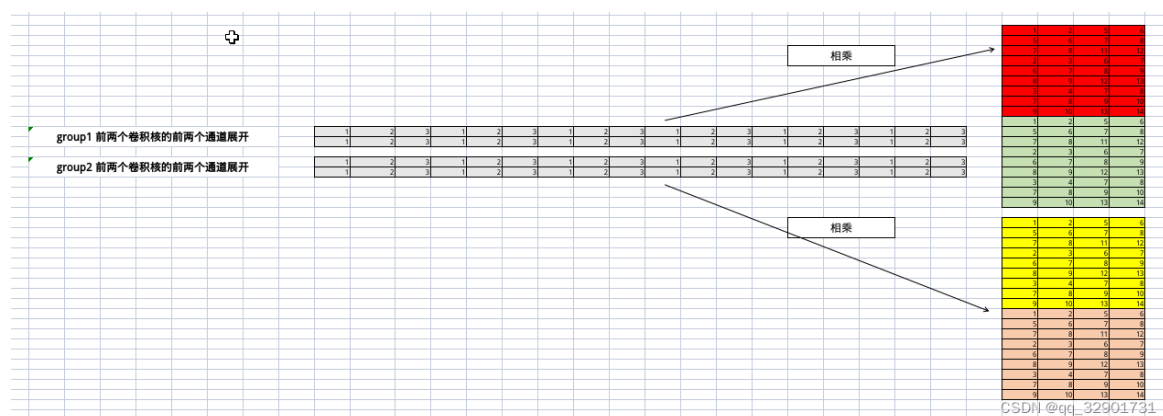


2. 将4个卷积核一分为2, 分成两组, 每组分配到两个卷积核, 每个卷积核的通道数为2, 它们分别是 `group1` 和 `group2`.

`group1` 处理第1份通道数为2的输入特征图, `group2` 处理第2份通道数为2的输入特征图



3. 做分组卷积，属于 `group1` 的两个卷积核与属于 `group1` 的输入特征图通道展开后进行相乘，属于 `group2` 的两个卷积核与属于 `group2` 的输入特征图通道展开后进行相乘。



KuiperInfer中的Im2Col实现

因为 `armadillo` 自身是列主序的，所以我们这里相当于在做 `Im2Row`，但是这个不影响我们的课程内容，大家记住这个奇怪の設定就好。

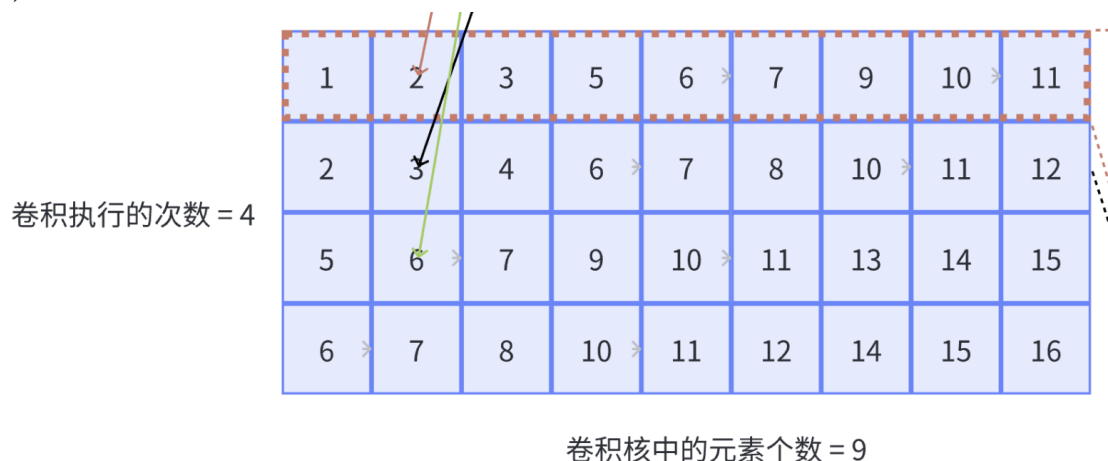
`Im2Col` 的代码位于 `convolution.cpp` 中。

```
1 arma::fmat ConvolutionLayer::Im2Col(sftensor input,
   uint32_t kernel_w,
   uint32_t kernel_h, uint32_t input_w,
   uint32_t input_h,
   uint32_t input_c_group,
   uint32_t group,
   uint32_t row_len,
   uint32_t col_len)
2 {
3     const {
4         arma::fmat input_matrix(input_c_group * row_len,
5             col_len);
6         const uint32_t input_padded_h = input_h + 2 *
7             padding_h_;
8         const uint32_t input_padded_w = input_w + 2 *
9             padding_w_;
10        const float padding_value = 0.f;
```

我们首先来看一下传入到这个函数中的参数，依次是：

1. `input`: 输入特征图像
2. `kernel_*`: 卷积核的大小
3. `input_*`: 输入特征图像的尺寸大小，也就是`input`的尺寸大小
4. `input_c_group`: 每个`group`处理的通道数量，如前文所叙，我们会将输入特征图的通道按照组数进行切分
5. `group`: 当前进行`Im2Col`的组数(`group`)
6. `row_len`: $kernel_w \times kernel_h$, 也就是上一节中卷积核展开后的列数或输入展开后的行数
7. `col_len`: 卷积计算的次数，也就是卷积窗口滑动的次数

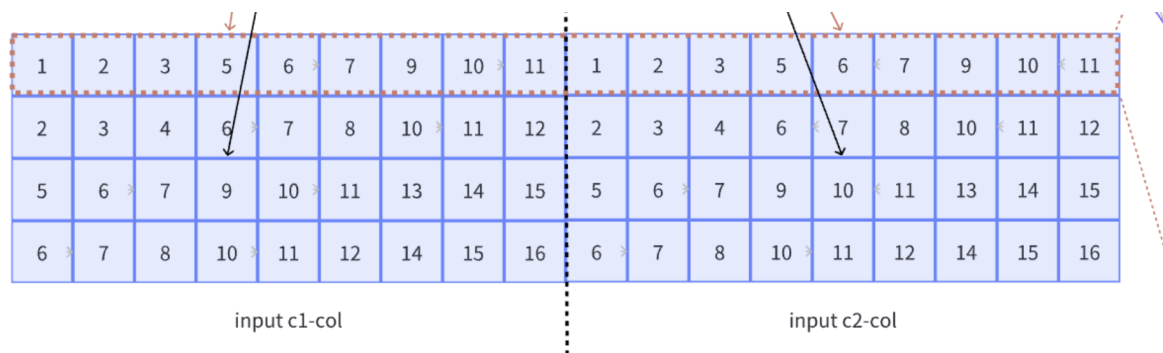
对于下图的情况，`row_len`等于9，表示展开后的行数；`col_len`等于4，表示卷积窗口滑动的次数。



```
1   arma::fmat input_matrix(input_c_group * row_len,
    col_len);
2   const uint32_t input_padded_h = input_h + 2 *
    padding_h_;
3   const uint32_t input_padded_w = input_w + 2 *
    padding_w_;
```

`input_matrix`用于存储对输入图像展开后的矩阵，`input_padded_*`表示输入填充后的尺寸大小。为什么这里的`input_matrix`行数等于 $input_c_group \times row_len$ 呢，我们从下方的图中可以看出，对于多输入通道的情况，它的列数等于输入通道数和卷积核相乘（因为我们是

列主序的，实际执行 **Im2Row**，所以行列相反），它的行数等于 **col_len**（行列相反），也就是卷积窗口进行滑动的次数。

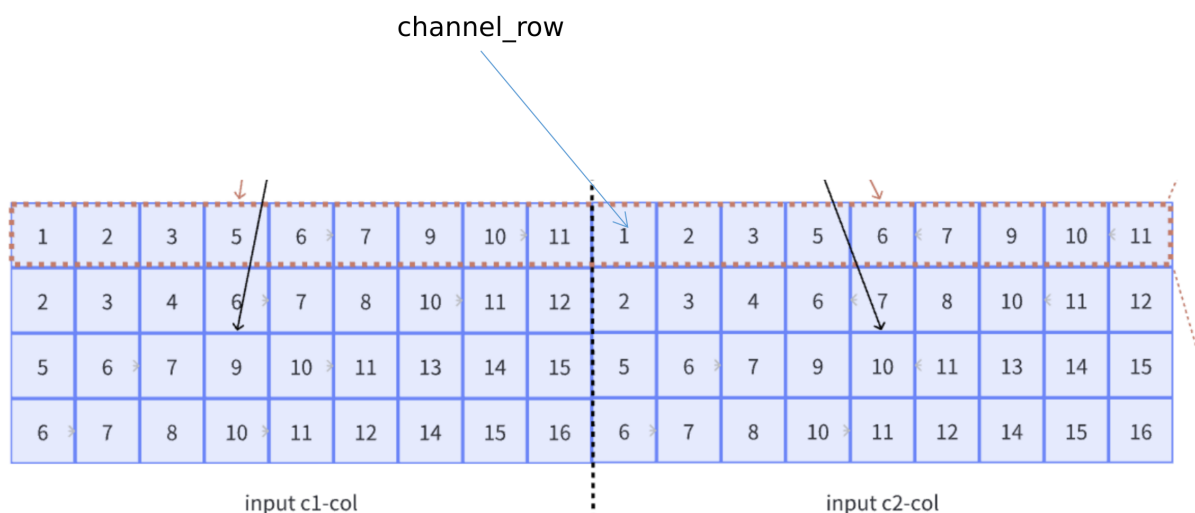


```

1  for (uint32_t ic = 0; ic < input_c_group; ++ic) {
2      float* input_channel_ptr =
3          input->matrix_row_ptr(ic + group *
input_c_group);
4      uint32_t current_col = 0;
5      uint32_t channel_row = ic * row_len;
6      for (uint32_t w = 0; w < input_padded_w -
kernel_w + 1; w += stride_w_) {
7          for (uint32_t r = 0; r < input_padded_h -
kernel_h + 1; r += stride_h_) {
8              float* input_matrix_ptr =
9                  input_matrix.colptr(current_col) +
channel_row;

```

我们需要提取当前的输入通道，并将该通道的起始指针赋值给 **input_channel_ptr**。其中， **channel_row** 表示我们当前展开通道后开始摆放的起始位置，以第2个通道($ic = 2$)为例：

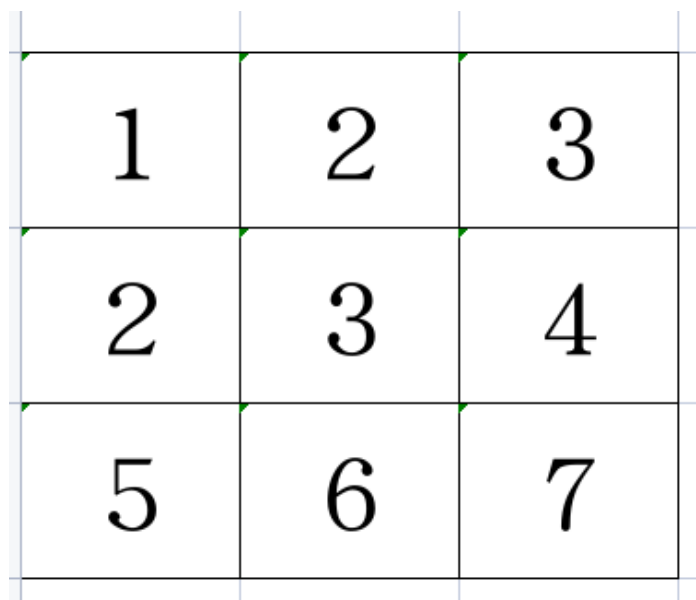


在第6 - 7行中，我们对一个输入通道进行窗口滑动，

`input_matrix_ptr`表示当前元素展开后存放的位置。

```
1  for (uint32_t kw = 0; kw < kernel_w; ++kw) {
2      const uint32_t region_w = input_h * (w + kw -
padding_w_);
3      for (uint32_t kh = 0; kh < kernel_h; ++kh) {
4          if ((kh + r >= padding_h_ && kw + w >=
padding_w_) &&
5              (kh + r < input_h + padding_h_ &&
6              kw + w < input_w + padding_w_)) {
7              float* region_ptr =
8                  input_channel_ptr + region_w + (r +
kh - padding_h_);
9                  *input_matrix_ptr = *region_ptr;
10         }
11         ...
12         input_matrix_ptr += 1;
```

`input_matrix_ptr`依次指向上图中`channel_row`之后的各个位置，用于存放展开后的各元素。



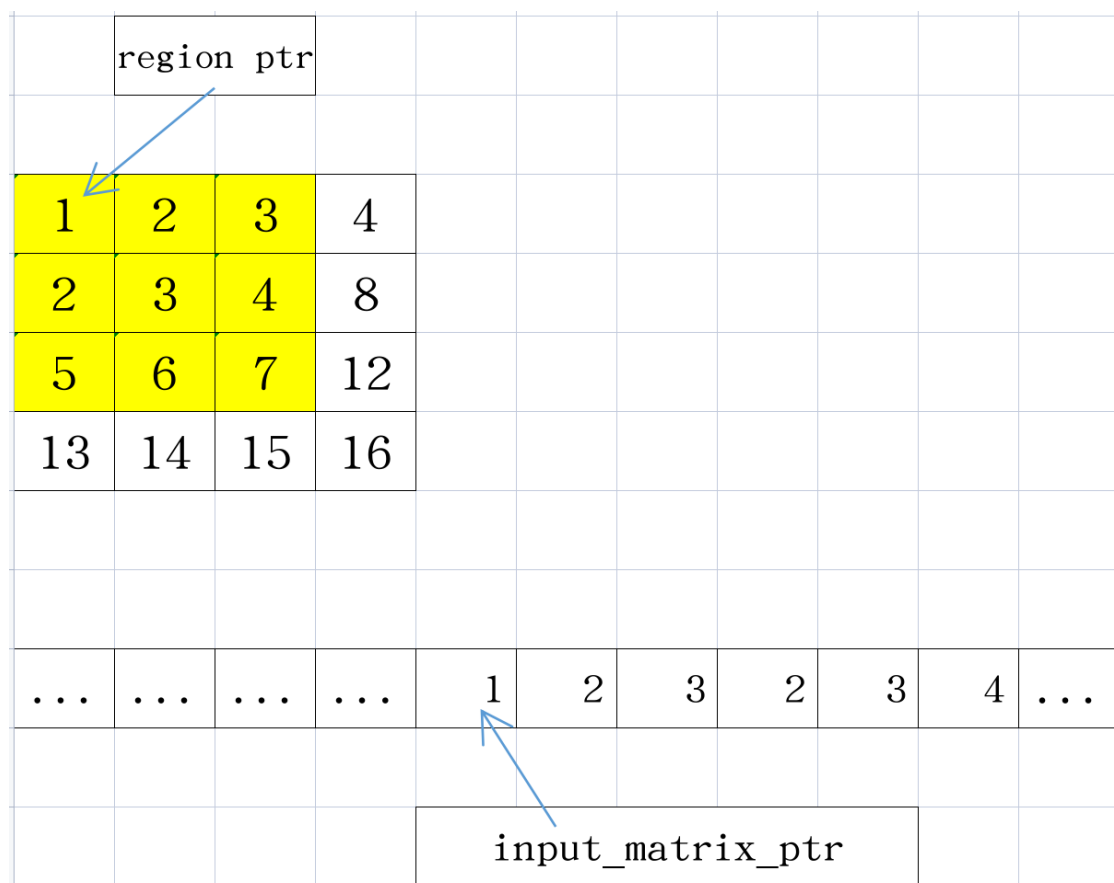
| | | |
|---|---|---|
| 1 | 2 | 3 |
| 2 | 3 | 4 |
| 5 | 6 | 7 |

接下来，我们需要处理卷积窗口内的所有元素。首先，我们要定位到当前需要展开的元素`region_ptr`，也就是黄色范围。

定位的方法是先将指针定位到输入通道的列，即

$input_channel_ptr + (w + kw - padding_w)$ ，然后再将指针定位到输入通道数的行，即

$input_channel_ptr + (r + kh - padding_h)$ 。这样，我们就能得到当前需要处理元素的内存地址。



`region_ptr` 表示正在展开的输入特征图的内存索引位置，而

`input_matrix_ptr` 表示展开后存放位置的指针。

卷积算子实现的主流程

```
1 InferStatus ConvolutionLayer::Forward(  
2     const  
   std::vector<std::shared_ptr<Tensor<float>>>& inputs,  
3     std::vector<std::shared_ptr<Tensor<float>>>&  
   outputs)
```

我们来看卷积算子中的主要实现流程，首先是对批次数据逐个处理。

```
1   for (uint32_t i = 0; i < batch_size; ++i) {
2       ...
3       ...
4   }
```

```
1   const uint32_t input_c = input->channels();
2   const uint32_t input_padded_h = input->rows() +
3   2 * padding_h_;
4   const uint32_t input_padded_w = input->cols() +
5   2 * padding_w_;
6
7   const uint32_t output_h =
8       std::floor((int(input_padded_h) -
9   int(kernel_h)) / stride_h_ + 1);
10  const uint32_t output_w =
11       std::floor((int(input_padded_w) -
12   int(kernel_w)) / stride_w_ + 1);
13
14  ...
15  uint32_t col_len = output_h * output_w;
```

在以上的代码中，我们计算了输入张量的尺寸，输出张量的尺寸以及 `col_len`（也就是卷积窗口执行计算的次数）。

```

1  uint32_t input_c_group = input_c / groups_;
2
3  for (uint32_t g = 0; g < groups_; ++g) {
4      const auto& input_matrix =
5          // 展开属于一个group中的所有输入通道
6          Im2Col(input, kernel_w, kernel_h, input-
7              >cols(), input->rows(),
8              input_c_group, g, row_len, col_len);
9      std::shared_ptr<Tensor<float>> output_tensor =
10         outputs.at(i);
11      if (output_tensor == nullptr || output_tensor-
12          >empty()) {
13          output_tensor =
14              std::make_shared<Tensor<float>>
15              (kernel_count, output_h, output_w);
16          outputs.at(i) = output_tensor;
17      }
18  }

```

在以上的代码中，我们对 `group` 进行迭代遍历，其中 `g` 表示当前的组号，`input_c_group` 表示每组卷积需要处理的通道数，`Im2Col` 函数中会对属于该组的输入通道进行展开。

```

1  for (uint32_t g = 0; g < groups_; ++g) {
2      ...
3      ...
4      const uint32_t kernel_count_group_start =
5          kernel_count_group * g;
6      for (uint32_t k = 0; k < kernel_count_group;
7          ++k) {
8          arma::frowvec kernel;
9          if (groups_ == 1) {
10             // 非分组的情况
11             kernel = kernel_matrix_arr_.at(k);
12         } else {
13             // 分组卷积的情况
14             kernel =
15                 kernel_matrix_arr_.at(kernel_count_group_start + k);
16         }
17     }
18 }

```



```

13         }
14         ConvGemmBias(input_matrix, output_tensor, g,
15                       k, kernel_count_group,
16                       kernel, output_w, output_h);
17     }

```

`kernel_count_group` 是分组卷积中，每组卷积分得的卷积核个数，所以在第3行的for循环中，我们使用 `kernel_count_goup`, `k` 等变量定位到对应的卷积核。

$$kernel\ index = k + g \times kernel\ count\ group$$

这里我们对属于同一组的输入通道和卷积核（每组卷积核的个数是 `kernel_count_group` 个）进行逐一的相乘，也就是说，这里进行相乘的卷积核和输入通道都属于同一个 `group`。

小结：

1. 我们先计算得到了每个组需要处理的通道数

`input_channel_group`，随后再对该组这些输入通道进行展开（`Im2Col` 函数中）。

2. 获取到1中对应的卷积核组，将它们逐个取出，取出的方式为：

$$kernel\ index + group \times kernel\ count\ group$$

3. 然后，我们将取出的同组每个卷积核与步骤1中得到的同一组输入通道进行相乘，得到结果。

KuiperInfer中的GEMM实现

```
1 void ConvolutionLayer::ConvGemmBias(  
2     const arma::fmat& input_matrix, sftensor  
   output_tensor, uint32_t group,  
3     uint32_t kernel_index, uint32_t  
   kernel_count_group,  
4     const arma::frowvec& kernel, uint32_t output_w,  
   uint32_t output_h) const {  
5
```

我们首先来看一下传入到这个函数中的参数，依次是：

1. `input_matrix`: 展开后的输入特征
2. `output_tensor`: 用于存放输出的矩阵
3. `group`: 当前进行 `Im2Col` 的组(group)数
4. `kernel*`: 用于定位当前展开后的卷积核
5. `output_*`: 输出矩阵的尺度大小

```
1     arma::fmat output(  
2         output_tensor->matrix_raw_ptr(kernel_index +  
   group * kernel_count_group),  
3         output_h, output_w, false, true);
```

以上的代码中定位了输出矩阵所在的位置，`kernel_count_group`表示属于一个 `group` 的卷积核数量。

如前文所叙。`kernel_index` 表示当前 `group` 内的卷积核索引，`group` 表示当前的组数，所以我们使用

$kernel\ index + group \times kernel\ count\ group$ 索引到当前卷积核对应的输出矩阵位置。

```

1  if (!this->bias_.empty() && this->use_bias_) {
2      std::shared_ptr<Tensor<float>> bias;
3      bias = this->bias_.at(kernel_index);
4      if (bias != nullptr && !bias->empty()) {
5          float bias_value = bias->index(0);
6          output = kernel * input_matrix + bias_value;
7      } else {
8          LOG(FATAL) << "Bias tensor is empty or
    nullptr";
9      }
10 } else {
11     output = kernel * input_matrix;
12 }

```

接下来，我们将展开后的输入矩阵 `input_matrix` 和展开后的卷积核 `kernel` 相乘，得到结果矩阵 `output`。

卷积算子的实例化

我们在 `convolution.cpp` 的最后将该算子的实例化过程注册到了深度学习推理框架中。

```

1  LayerRegistererWrapper kConvGetInstance("nn.Conv2d",
2
    ConvolutionLayer::GetInstance);

```

具体的实例化函数如下，同样是先得到初始化算子需要的参数 `params`：

```

1 ParseParameterAttrStatus
  ConvolutionLayer::GetInstance(
2     const std::shared_ptr<RuntimeOperator>& op,
3     std::shared_ptr<Layer>& conv_layer) {
4     CHECK(op != nullptr) << "Convolution operator is
  nullptr";
5     const std::map<std::string,
  std::shared_ptr<RuntimeParameter>>& params =
6         op->params;
7

```

这些参数的初始化过程，我们就不做详细的描述了，因为它和池化算子的初始化过程大同小异。

```

1     conv_layer = std::make_shared<ConvolutionLayer>(
2         out_channel->value, in_channel->value,
  kernels.at(0), kernels.at(1),
3         paddings.at(0), paddings.at(1), strides.at(0),
  strides.at(1),
4         groups->value, use_bias->value);

```

这里我们根据以上的得到的参数对卷积算子进行了初始化，**但是在这里，我们还没有对卷积算子中的权重(weight 和 bias)进行加载。**

我们重点来看一下卷积算子中的权重和偏移量初始化、加载过程：

```

1  const std::map<std::string,
std::shared_ptr<RuntimeAttribute>>& attrs =
2      op->attribute;
3      ...
4      const auto& bias = attrs.at("bias");
5      const std::vector<int>& bias_shape = bias-
>shape;
6      if (bias_shape.empty() || bias_shape.at(0) !=
out_channel->value) {
7          LOG(ERROR) << "The attribute of bias shape is
wrong";
8          return
ParseParameterAttrStatus::kAttrMissingBias;
9      }
10
11     const std::vector<float>& bias_values = bias-
>get<float>();
12     conv_layer->set_bias(bias_values);
13 }

```

在第1行中，我们获取到了 `RuntimeOperator` 中的权重变量 `attrs`。然后我们尝试找到其中的 `bias` 键。如果该键存在，就获取其中的偏移量权重（`bias`），并将其赋值给 `conv_layer`。

```

1  const auto& weight = attrs.at("weight");
2  const std::vector<int>& weight_shape = weight-
>shape;
3  if (weight_shape.empty()) {
4      LOG(ERROR) << "The attribute of weight shape is
wrong";
5      return
ParseParameterAttrStatus::kAttrMissingWeight;
6  }
7
8  const std::vector<float>& weight_values = weight-
>get<float>();
9  conv_layer->set_weights(weight_values);

```

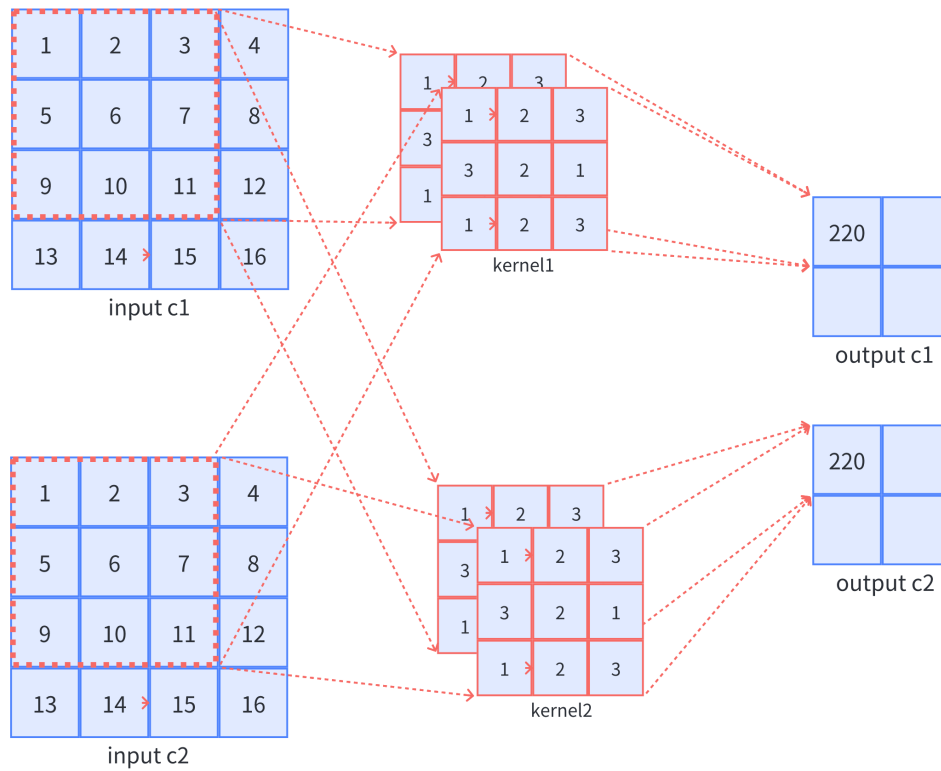
```
10
11     auto conv_layer_derived =
12         std::dynamic_pointer_cast<ConvolutionLayer>
13         (conv_layer);
14     CHECK(conv_layer_derived != nullptr);
15     conv_layer_derived->InitIm2ColWeight();
16     return
17         ParseParameterAttrStatus::kParameterAttrParseSuccess
18     ;
```

同样在以上的第一行中，我们尝试获取 `RuntimeOperator` 中的 `weight` 键，如果该键存在，就获取其对应的权重变量，并将它赋值给 `conv_layer`。至此，我们完成了卷积算子的实例化。对了，我们可以看到，这里有一句 `InitIm2ColWeight`，它是在展开该卷积算子对应的卷积核。

因为对于一个卷积算子来说，它的输入是不确定的，所以我们需要在运行时再调用 `Im2Col` 进行展开，而一个卷积算子中的权重是固定的，所以可以在初始化的时候进行展开。

卷积算子的单元测试

我们单元测试对应的卷积计算图示如下，是一个多输入通道(`input c1` 和 `input c2`)和多输出通道(`output` 的 `c1` 和 `c2`)的卷积算子。



```

1 TEST(test_registry, create_layer_convforward) {
2     const uint32_t batch_size = 1;
3     std::vector<sftensor> inputs(batch_size);
4     std::vector<sftensor> outputs(batch_size);
5
6     const uint32_t in_channel = 2;
7     for (uint32_t i = 0; i < batch_size; ++i) {
8         sftensor input = std::make_shared<ftensor>
9         (in_channel, 4, 4);
10         input->data().slice(0) = "1,2,3,4;"
11                                   "5,6,7,8;"
12                                   "9,10,11,12;"
13                                   "13,14,15,16;";
14         input->data().slice(1) = "1,2,3,4;"
15                                   "5,6,7,8;"
16                                   "9,10,11,12;"
17                                   "13,14,15,16;";
18         inputs.at(i) = input;
19     }

```

```

20     const uint32_t kernel_h = 3;
21     const uint32_t kernel_w = 3;
22     const uint32_t stride_h = 1;
23     const uint32_t stride_w = 1;
24     const uint32_t kernel_count = 2;
25     std::vector<sftensor> weights;
26     for (uint32_t i = 0; i < kernel_count; ++i) {
27         sftensor kernel = std::make_shared<ftensor>
(in_channel, kernel_h, kernel_w);
28         kernel->data().slice(0) = arma::fmat("1,2,3;"
29                                             "3,2,1;"
30                                             "1,2,3;");
31         kernel->data().slice(1) = arma::fmat("1,2,3;"
32                                             "3,2,1;"
33                                             "1,2,3;");
34         weights.push_back(kernel);
35     }
36     ConvolutionLayer conv_layer(kernel_count,
in_channel, kernel_h, kernel_w, 0,
37                               0, stride_h, stride_w,
1, false);
38     conv_layer.set_weights(weights);
39     conv_layer.Forward(inputs, outputs);
40     outputs.at(0)->Show();
41 }

```

课程作业

1. 仔细阅读这个课件并调试其中的代码，观察状态和变量的变换。因为本节课内容非常难，跟着看一遍不一定能全部掌握
2. 我们的单元测试只测试了group = 1的情况，针对分组卷积，即group != 1的情况，请自己构造单元测试，并逐步调试代码。