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
>
> 主项目：https://github.com/zjhellofss/KuiperInfer 欢迎大家点赞和PR.
>
> 课程代码：https://github.com/zjhellofss/kuiperdatawhale/course4

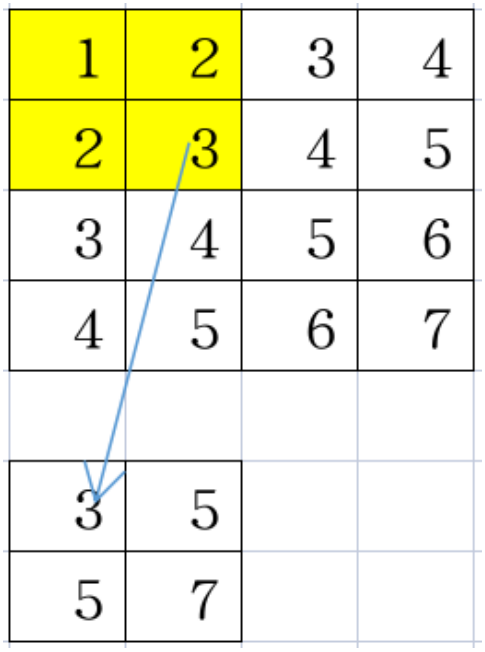
上节课程回顾

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

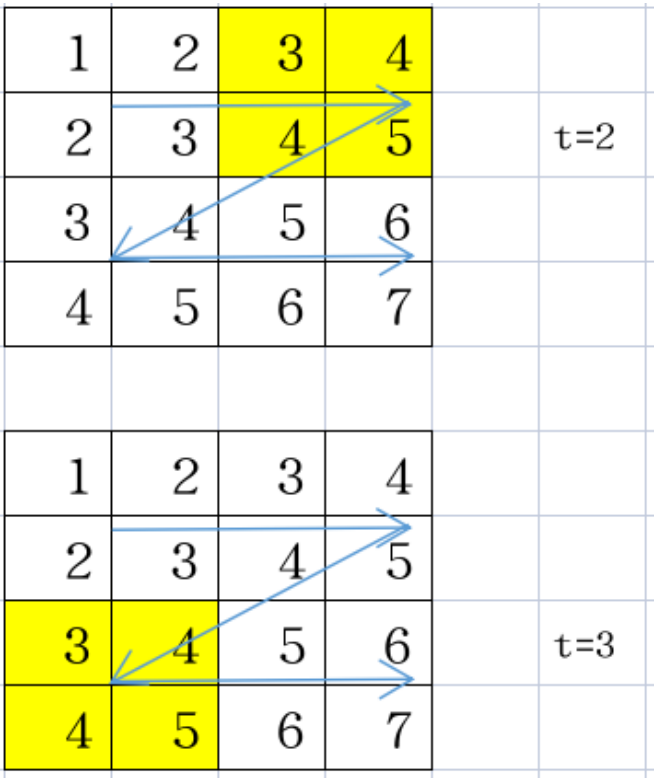
池化算子常用于缓解深度神经网络对位置的过度敏感性。

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。



以上图为例，黄色区域表示一个2 x 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：

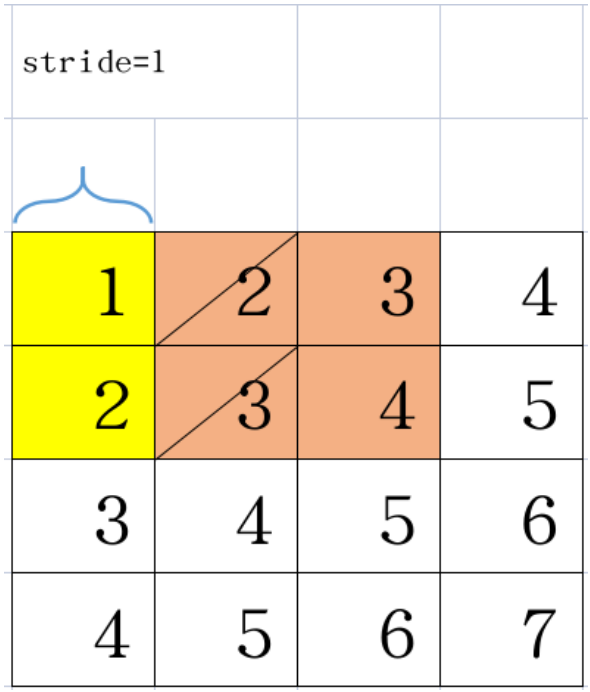


当t等于2时，池化窗口的输出为5, 也就是窗口内最大的元素为5. 当t等于3时，池化窗口向左下移动，池化窗口的输出同样是5. 所以，该输入特征图的最大池化计算结果输出如下，不难看出，输出中的每个位置都对应于原始输入窗口中的最大值。

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。



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		

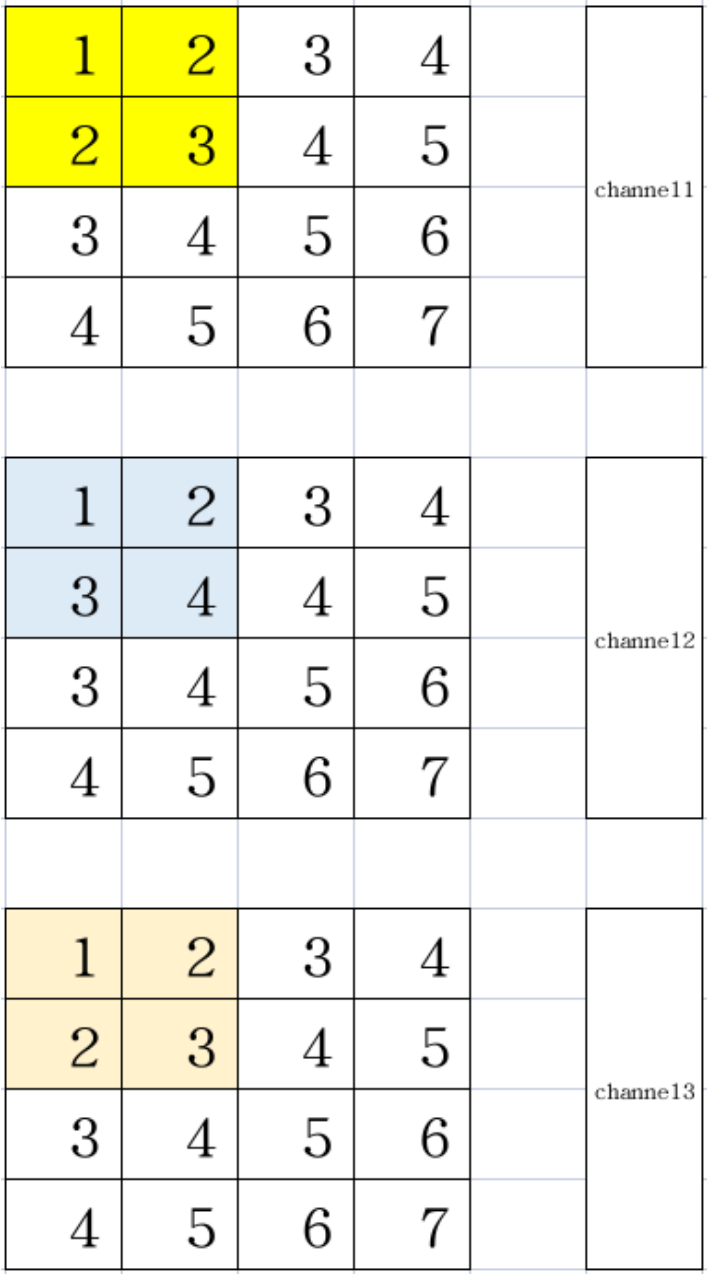
-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf
-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf
-inf	-inf	1	2	3	4	-inf	-inf
-inf	-inf	2	3	4	5	-inf	-inf
-inf	-inf	3	4	5	6	-inf	-inf
-inf	-inf	4	5	6	7	-inf	-inf
-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf
-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf

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。



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**遍历完整个输入通道。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`.

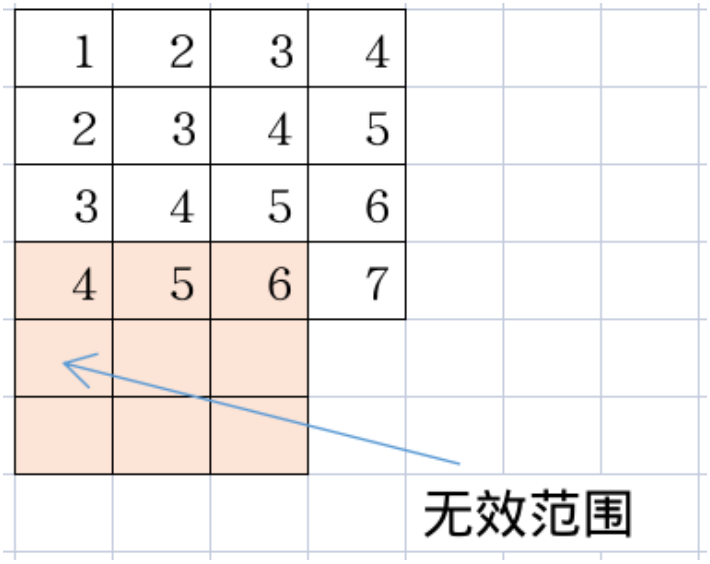
```
1  float* output_channel_ptr = output_channel.colptr(int(c / stride_w_));
2  float max_value = std::numeric_limits<float>::lowest();
3  for (uint32_t w = 0; w < pooling_w; ++w) {
4      for (uint32_t h = 0; h < pooling_h; ++h) {
5          float current_value = 0.f;
6          if ((h + r >= padding_h_ && 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。

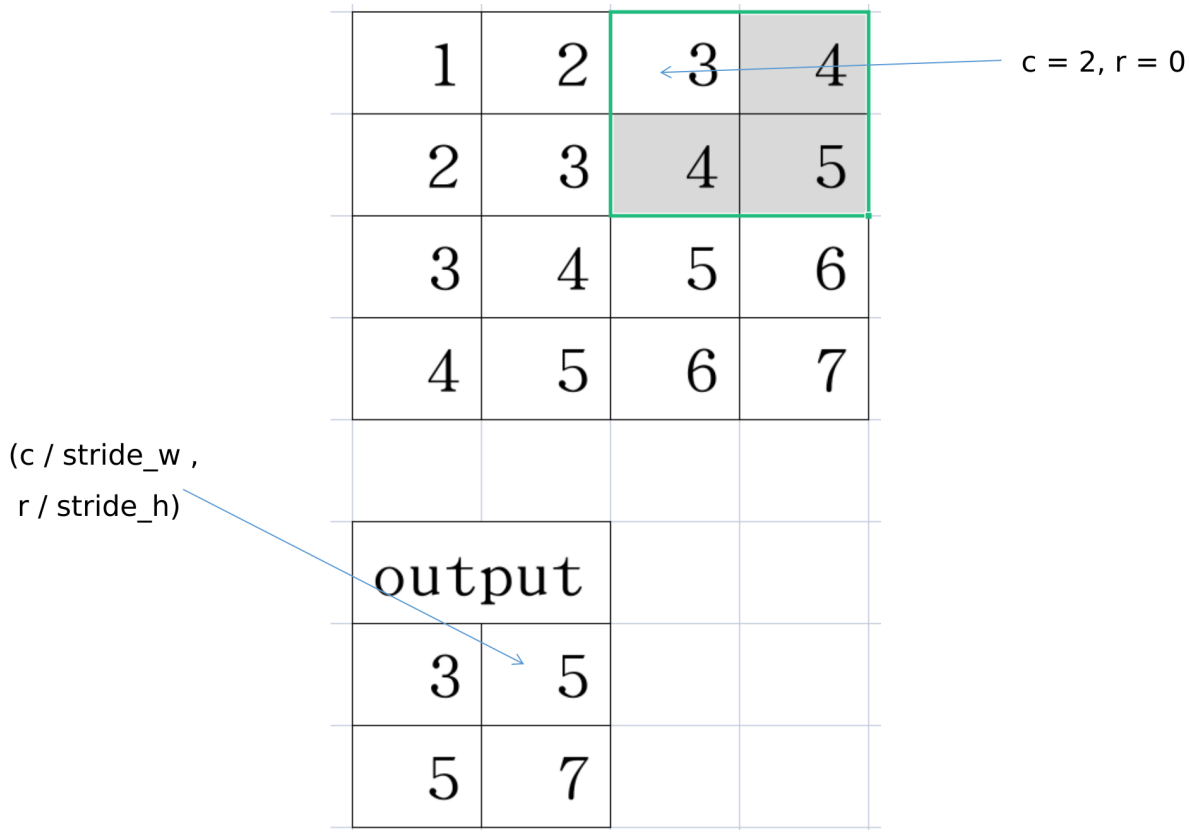
以下是关于无效值范围的图示。我们可以观察到，在一个窗口大小为3的池化算子中，**当该窗口移动到特定位置时，就会出现无效范围，即超出了输入特征图的尺寸限制**。



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**.`

```
LayerRegistererWrapper
kMaxPoolingGetInstance("nn.MaxPool2d",

 MaxPoolingLayer::GetInstance);
```

池化算子的实例化函数

```
ParseParameterAttrStatus
MaxPoolingLayer::GetInstance(
    const std::shared_ptr<RuntimeOperator>& op,
    std::shared_ptr<Layer>& max_layer) {
  ...
  const std::map<std::string,
std::shared_ptr<RuntimeParameter>>& params =
      op->params;

```

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`.

```
if (params.find("stride") == params.end()) {
    LOG(ERROR) << "Can not find the stride parameter";
    return ParseParameterAttrStatus::kParameterMissingStride;
  }

  auto stride =
 std::dynamic_pointer_cast<RuntimeParameterIntArray>(params.at("stride"));
  if (!stride) {
    LOG(ERROR) << "Can not find the stride parameter";
    return ParseParameterAttrStatus::kParameterMissingStride;
  }
  ...
```

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。

```
  if (params.find("padding") == params.end()) {
    LOG(ERROR) << "Can not find the padding parameter";
    return ParseParameterAttrStatus::kParameterMissingPadding;
  }

  auto padding =
 std::dynamic_pointer_cast<RuntimeParameterIntArray>(params.at("padding"));
  if (!padding) {
    LOG(ERROR) << "Can not find the padding parameter";
    return ParseParameterAttrStatus::kParameterMissingPadding;
  }
```

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。

```
  auto kernel_size = std::dynamic_pointer_cast<RuntimeParameterIntArray>(
      params.at("kernel_size"));
  if (!kernel_size) {
    LOG(ERROR) << "Can not find the kernel size parameter";
    return ParseParameterAttrStatus::kParameterMissingKernel;
  }
  const auto& padding_values = padding->value;
  const auto& stride_values = stride->value;
  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。

```
 max_layer = std::make_shared<MaxPoolingLayer>(
      padding_values.at(0), padding_values.at(1),
kernel_values.at(0),
      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。

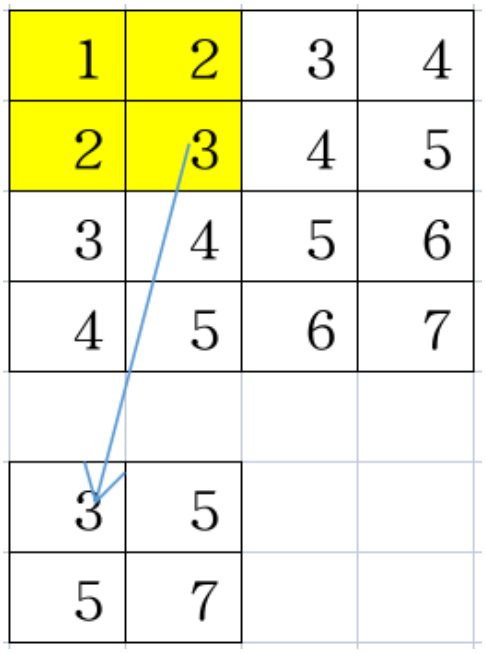
```
TEST(test_registry, create_layer_poolingforward) {
  std::shared_ptr<RuntimeOperator> op =
std::make_shared<RuntimeOperator>();
  op->type = "nn.MaxPool2d";
  std::vector<int> strides{2, 2};
  std::shared_ptr<RuntimeParameter> stride_param =
std::make_shared<RuntimeParameterIntArray>(strides);
  op->params.insert({"stride", stride_param});

  std::vector<int> kernel{2, 2};
  std::shared_ptr<RuntimeParameter> kernel_param =
std::make_shared<RuntimeParameterIntArray>(strides);
  op->params.insert({"kernel_size", kernel_param});

  std::vector<int> paddings{0, 0};
  std::shared_ptr<RuntimeParameter> padding_param =
std::make_shared<RuntimeParameterIntArray>
(paddings);
  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3，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  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
```

```
  sftensor tensor = std::make_shared<ftensor>(1, 4, 4);
  arma::fmat input = arma::fmat("1,2,3,4;"
                                "2,3,4,5;"
                                "3,4,5,6;"
                                "4,5,6,7");
  tensor->data().slice(0) = input;
  std::vector<sftensor> inputs(1);
  inputs.at(0) = tensor;
  std::vector<sftensor> outputs(1);
  layer->Forward(inputs, outputs);

  ASSERT_EQ(outputs.size(), 1);
  outputs.front()->Show();
}
```

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。

```
I20230721 14:05:09.855405  3224 tensor.cpp:201]
   3.0000   5.0000
   5.0000   7.0000
```

卷积算子的定义

Todo