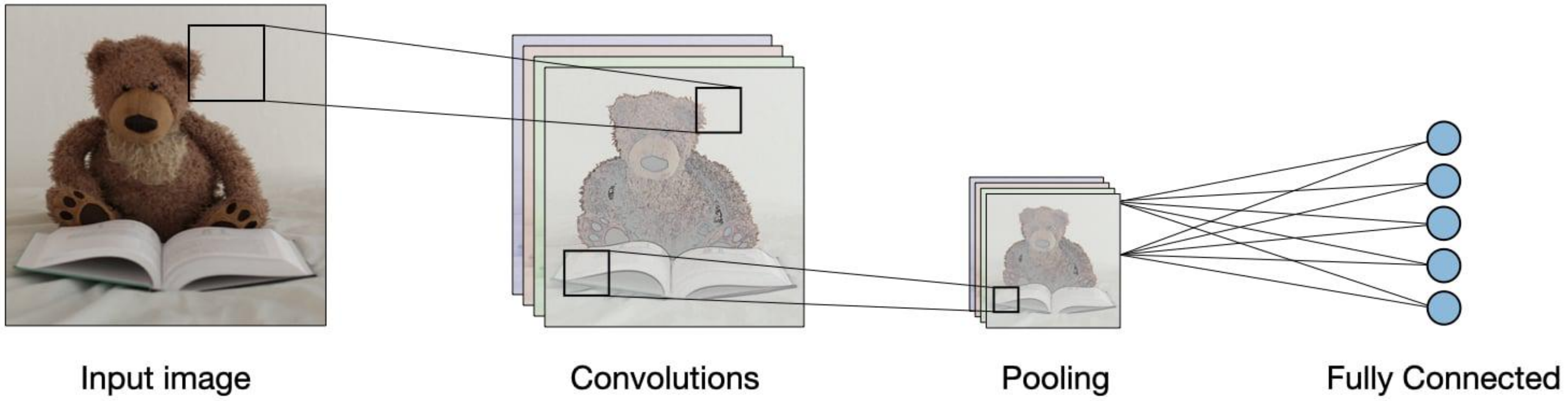
卷积结构及其计算

第三节

本节主要内容

- 卷积及其参数设计
- 经典卷积神经网络模型结构
- 卷积计算优化
- 自定义算子开发
- 练习

卷积神经网络（Convolutional Neural Network）

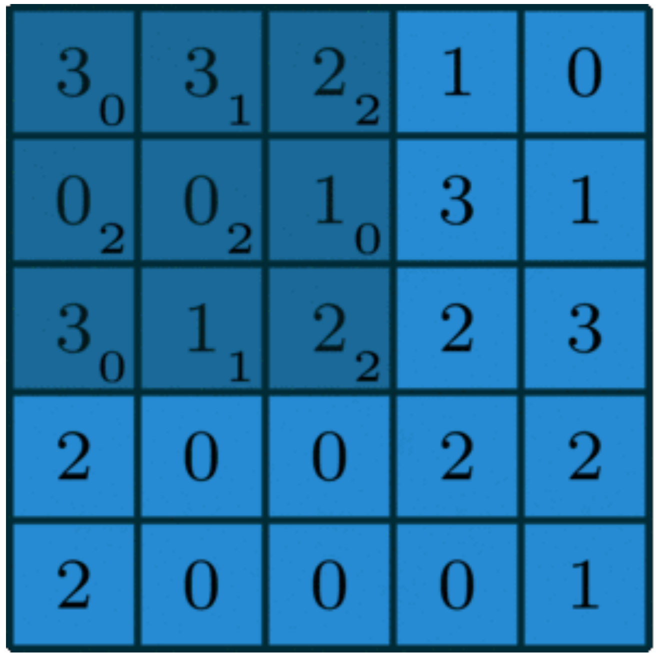


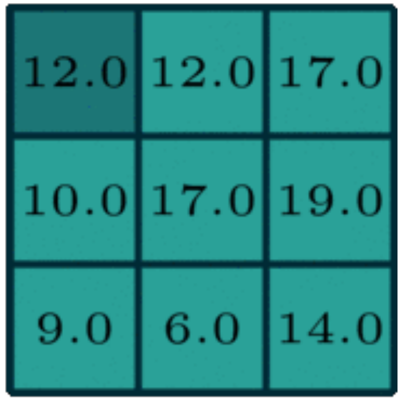
核心思想：

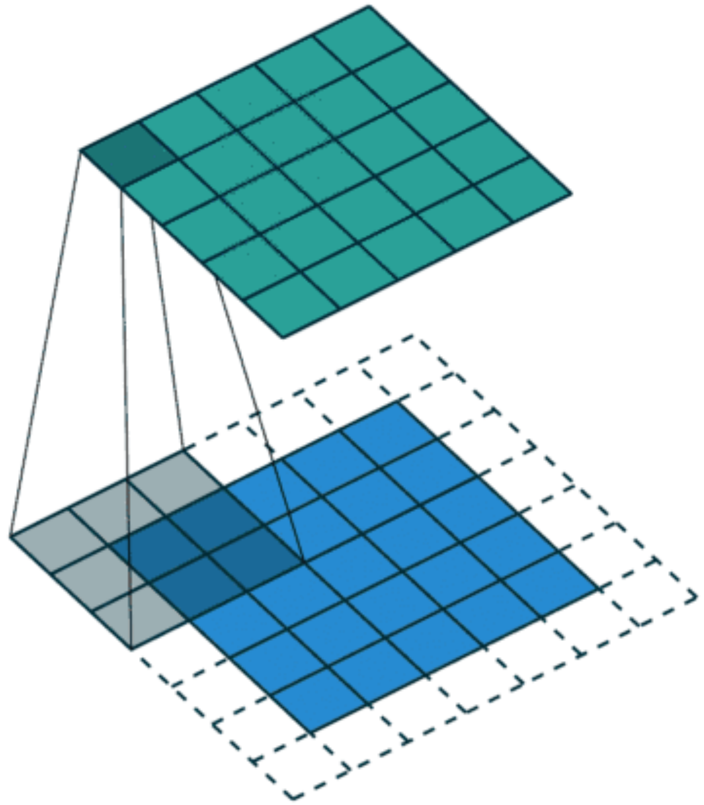
- **局部连接：视觉具有局部性，充分考虑领域信息，局部稠密连接。**
- **权重共享：每一组权重抽取图像中的一种特征。**

大幅度减少参数量、避免过拟合。

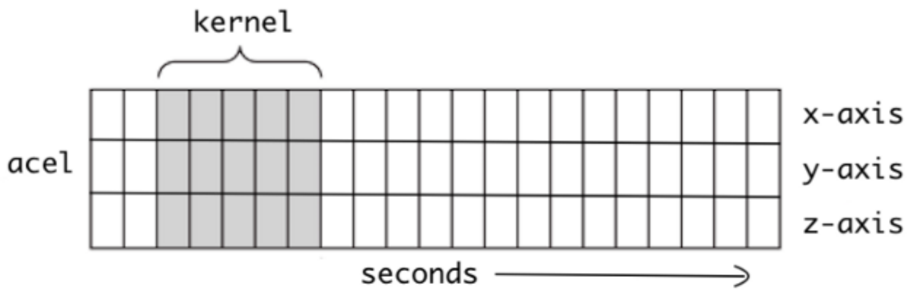
经典卷积计算[padding/stride]



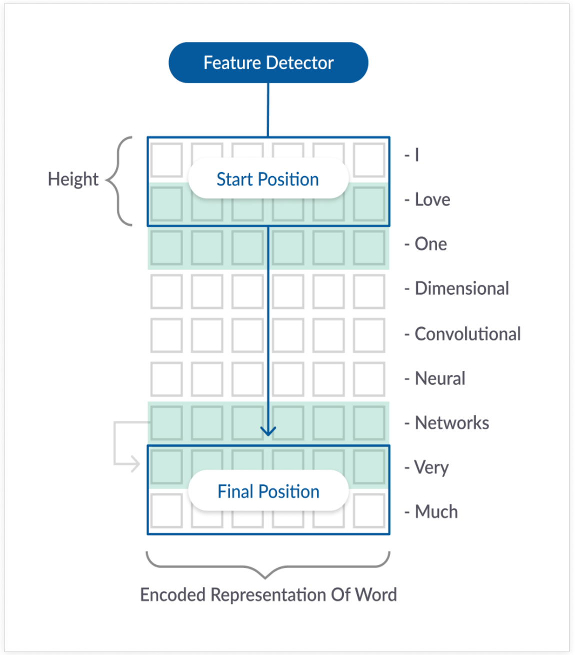


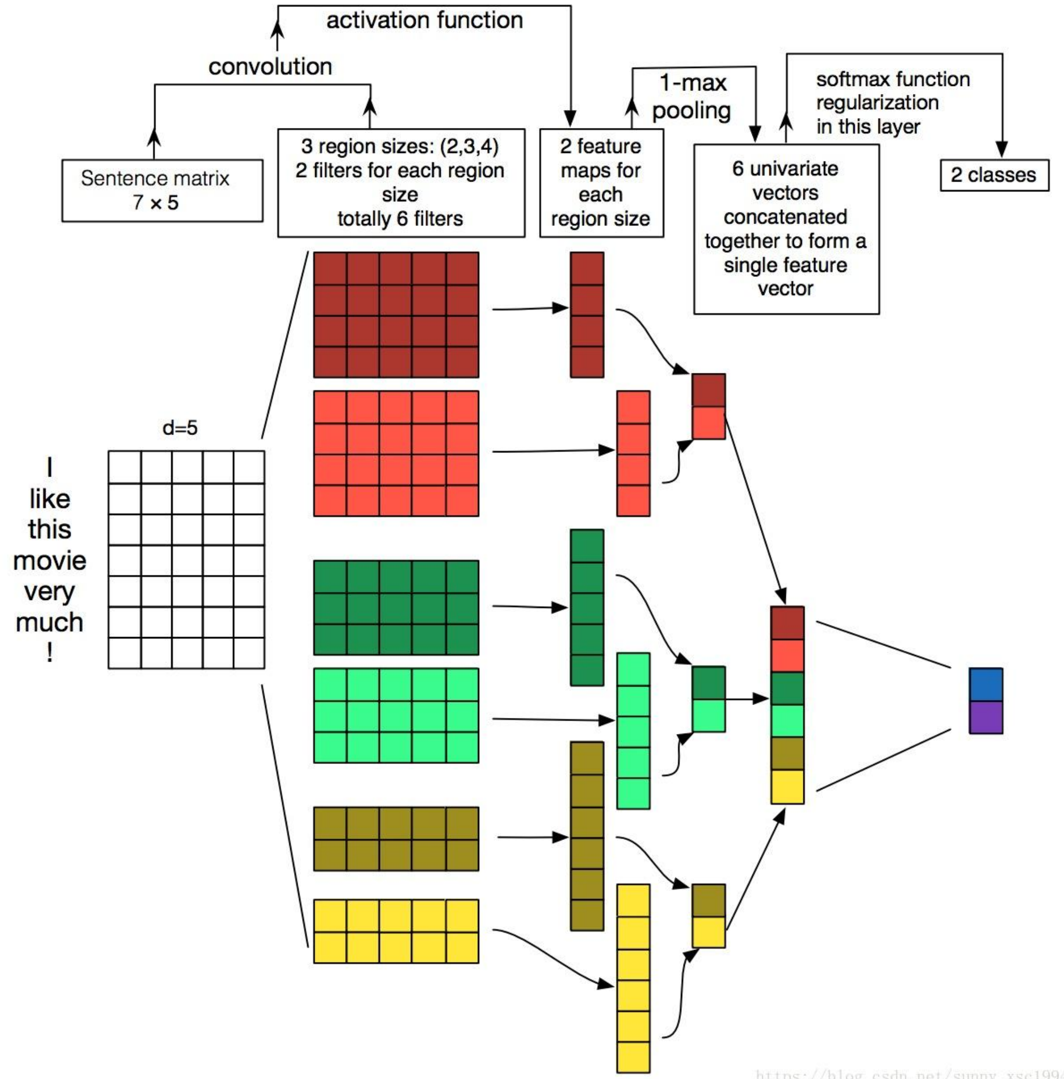


1D CONV



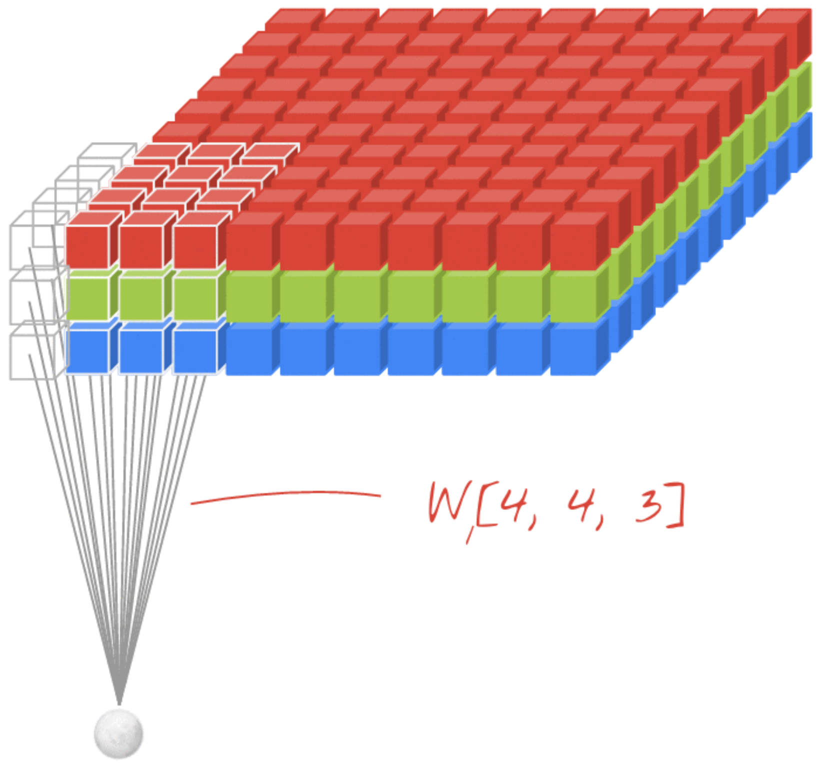
1D CONVOLUTIONAL - EXAMPLE





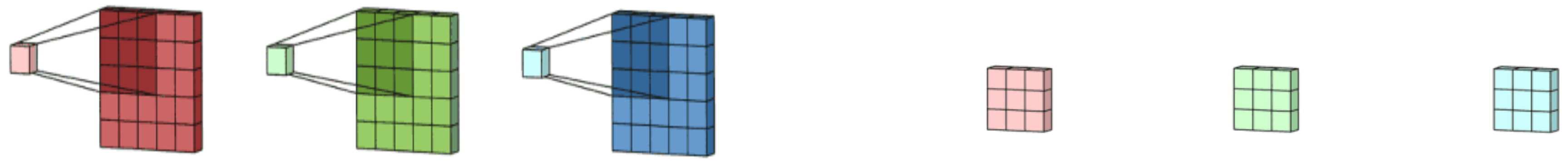
activation function
convolution
1-max pooling
softmax function regularization in this layer
Sentence matrix 7 × 5
3 region sizes: (2,3,4) 2 filters for each region size totally 6 filters
2 feature maps for each region size
6 univariate vectors concatenated together to form a single feature vector
2 classes
d=5
I like this movie very much !

2D CONV

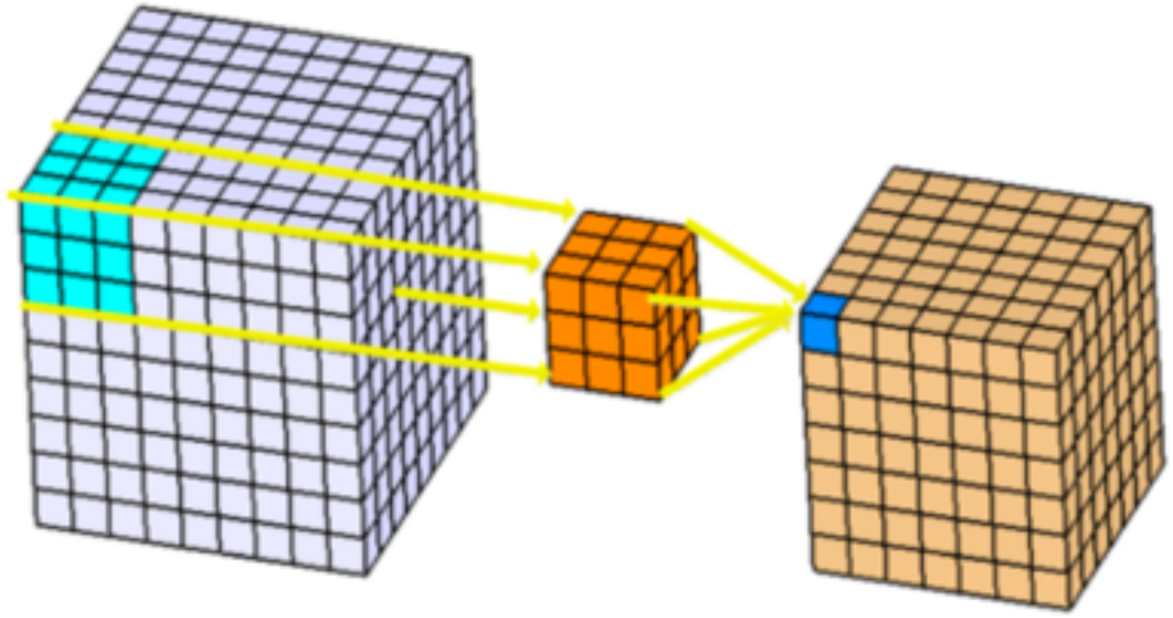


W, [4, 4, 3]

多通道卷积运算

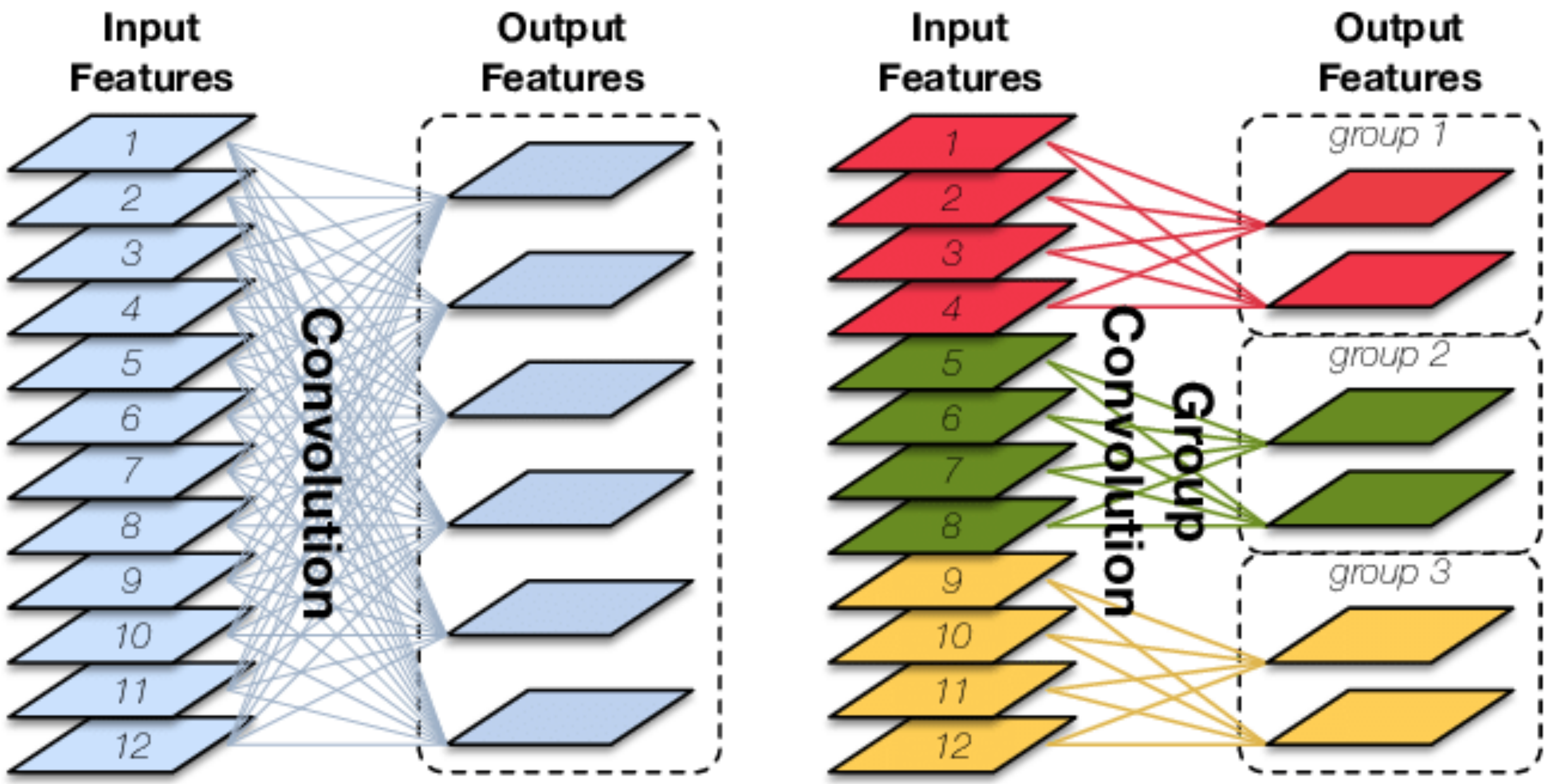


3D CONV

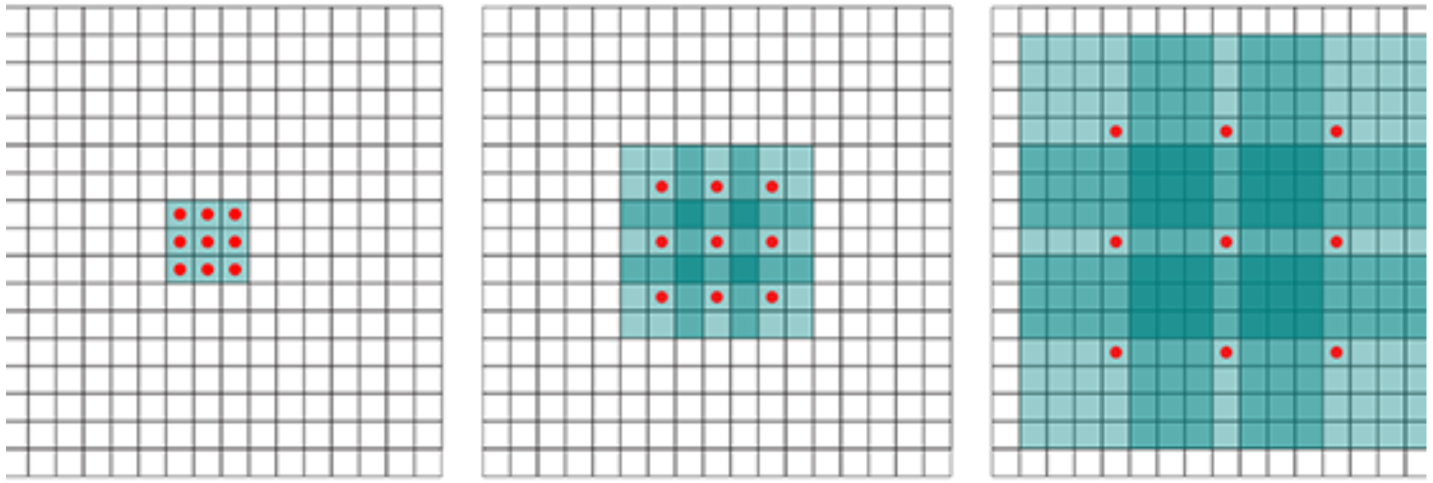


3D卷积将3维过滤器应用于数据集，并且过滤器将3方向（x，y，z）移动以计算低级特征表示。它们的输出形状是3维的体积空间，例如立方体或长方体。它们有助于视频，3D医学图像等中的事件检测。它们不仅限于3d空间，还可以应用于2d空间输入（例如图像）。

Group Convolution



Dilated Convolution

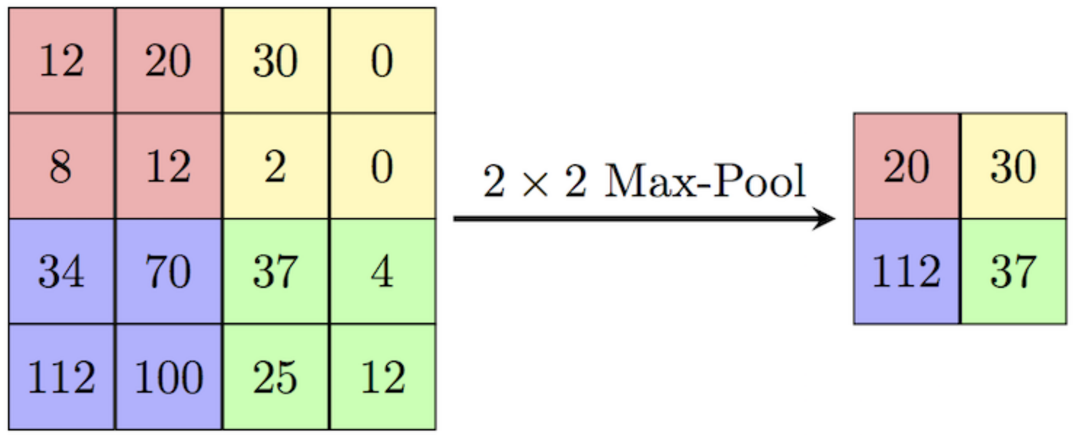


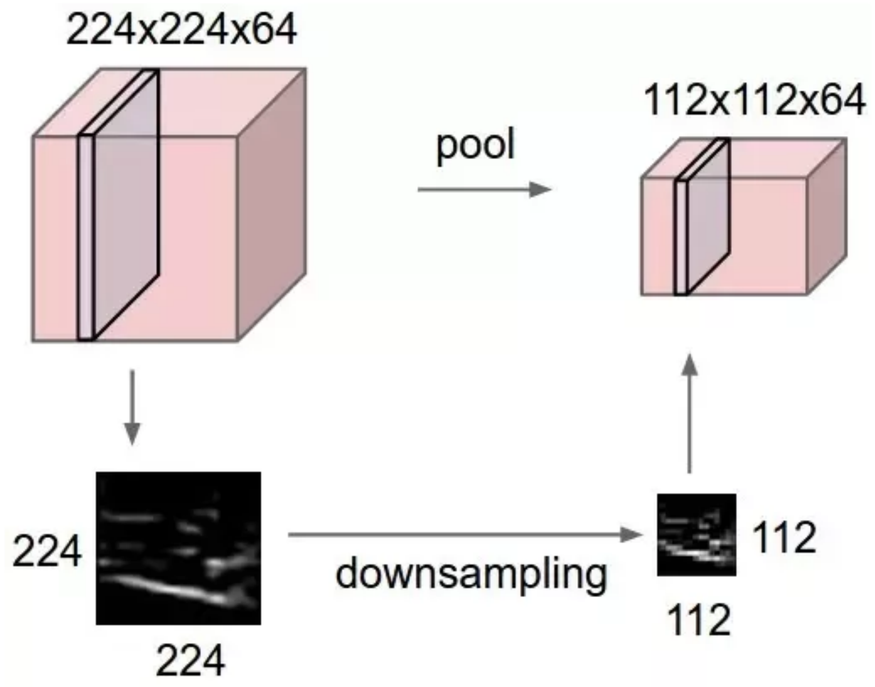
Conv2D类的dilation_rate参数是2-元组，用于控制膨胀卷积的膨胀率。

池化（Pooling）

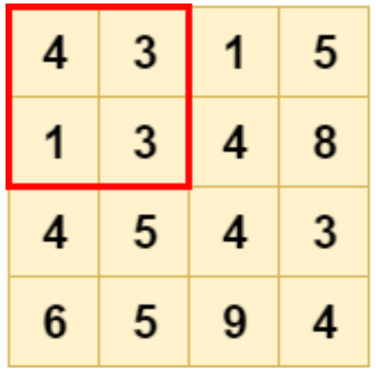
- 池化是使用某一位置相邻输出的总体统计特征代替网络在该位置的输出。
- 池化(POOL)是一个向下采样操作，通常应用于卷积层之后，卷积层执行一些空间不变性。其中，最大池和平均池是特殊类型的池，分别取最大值和平均值。

Max Pooling

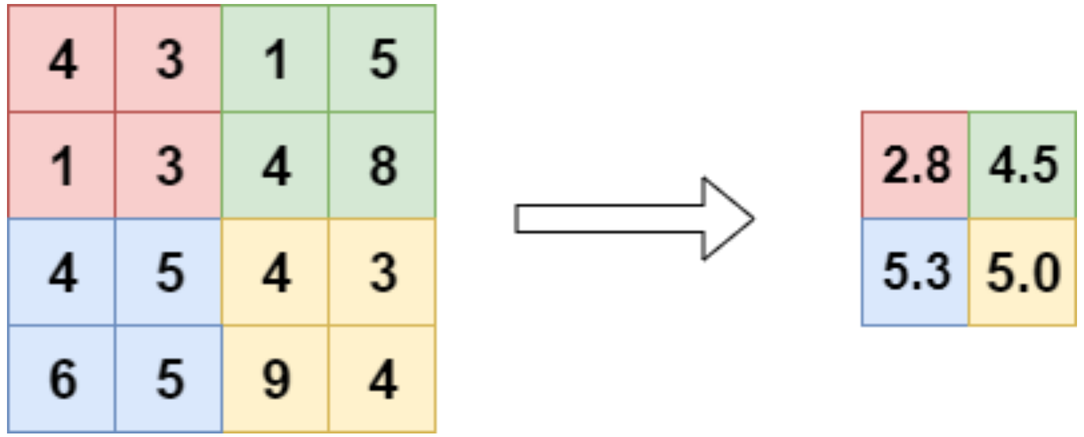

12 20 30 0
8 12 2 0
34 70 37 4
112 100 25 12
2 × 2 Max-Pool
20 30
112 37


224x224x64
pool
112x112x64
224
224
downsampling
112
112

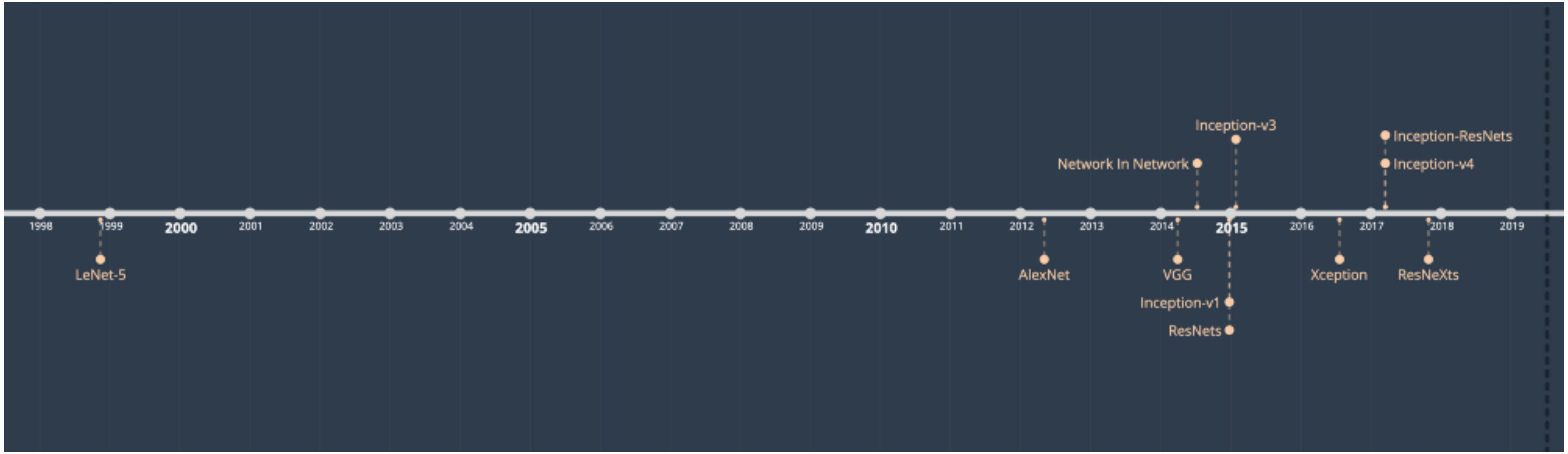
Average Pooling



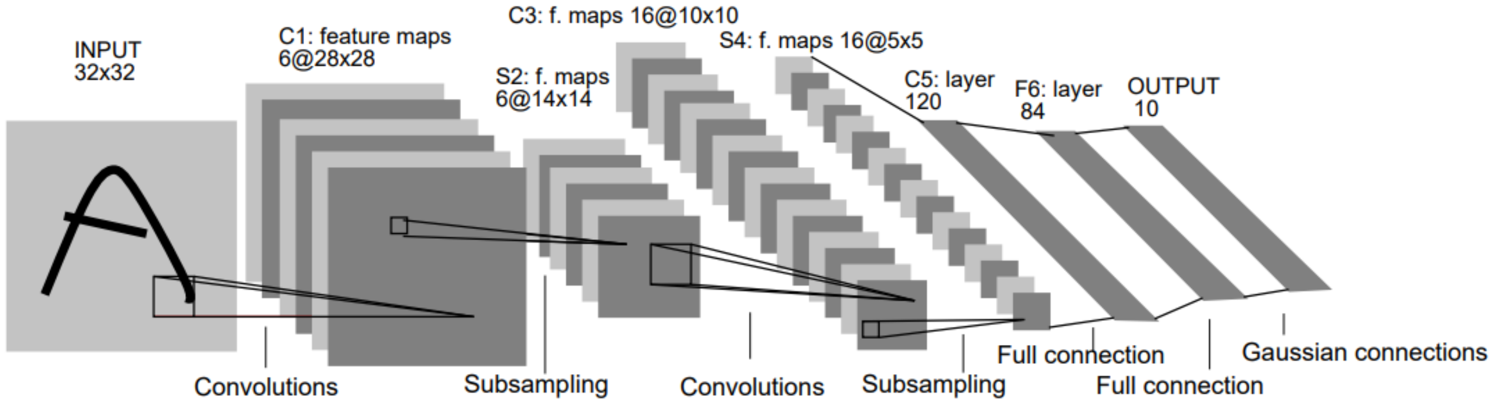
Avg([4, 3, 1, 3]) = 2.75



卷积网络演进

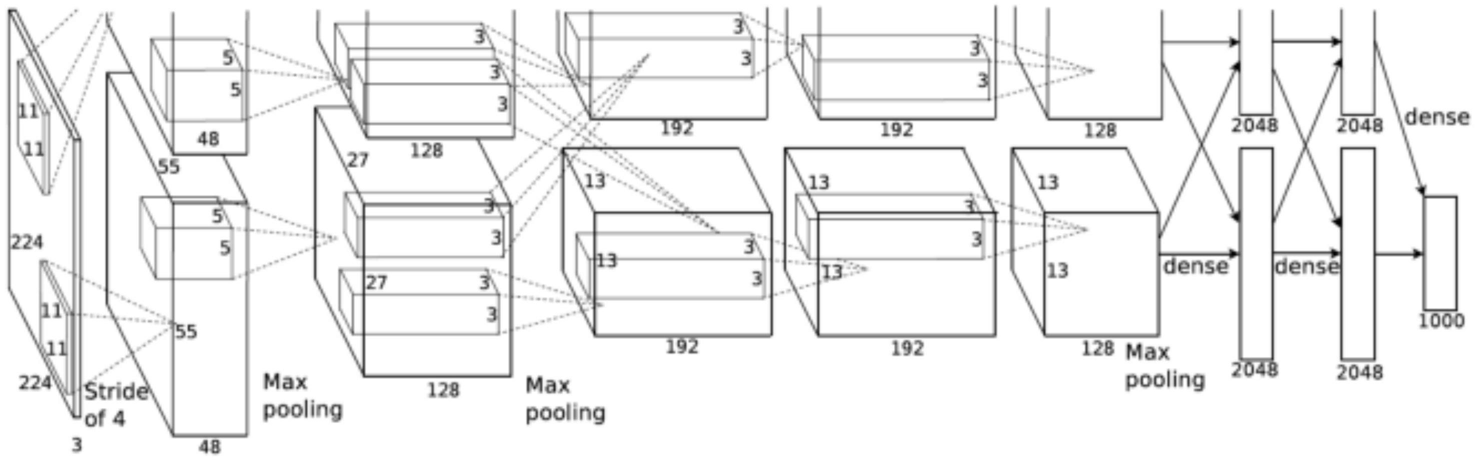

1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
LeNet-5
AlexNet
Network In Network
VGG
Inception-v3
Inception-v1
ResNets
Xception
Inception-ResNets
Inception-v4
ResNeXts

LeNet-5



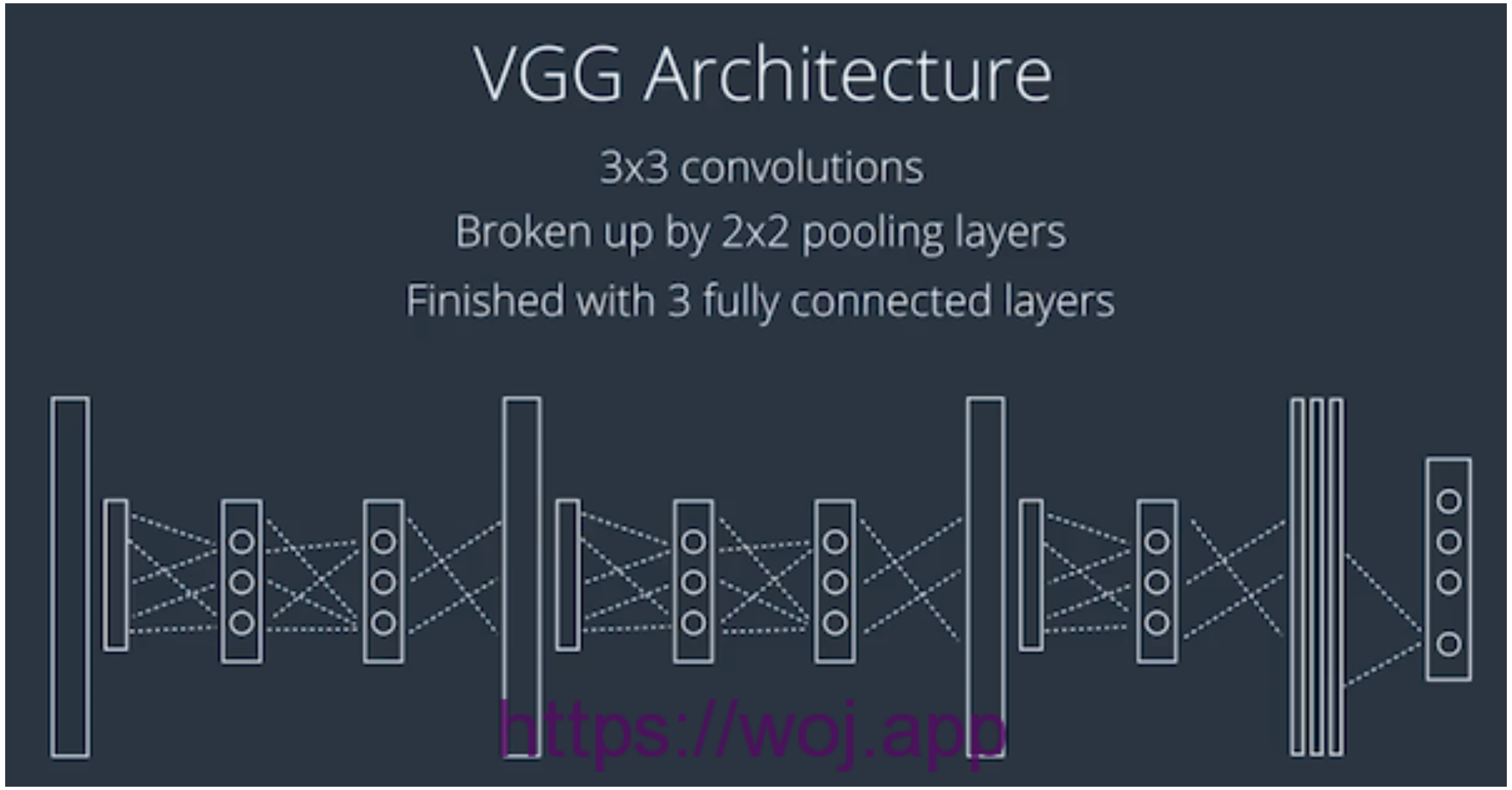
- 诞生于 1994 年
- LeNet-5是Yann LeCun等人在多次研究后提出的最终卷积神经网络结构，一般LeNet即指代LeNet-5。
- LeNet-5包含七层，不包括输入，每一层都包含可训练参数（权重），当时使用的输入数据是32*32像素的图像。下面逐层介绍LeNet-5的结构，并且，卷积层将用Cx表示，子采样层则被标记为Sx，完全连接层被标记为Fx，其中x是层索引。

AlexNet



- 2012年
- Alex Krizhevsky、Ilya Sutskever在多伦多大学Geoff Hinton的实验室设计
- AlexNet有6亿个参数和650,000个神经元，包含5个卷积层，]3个全连接层

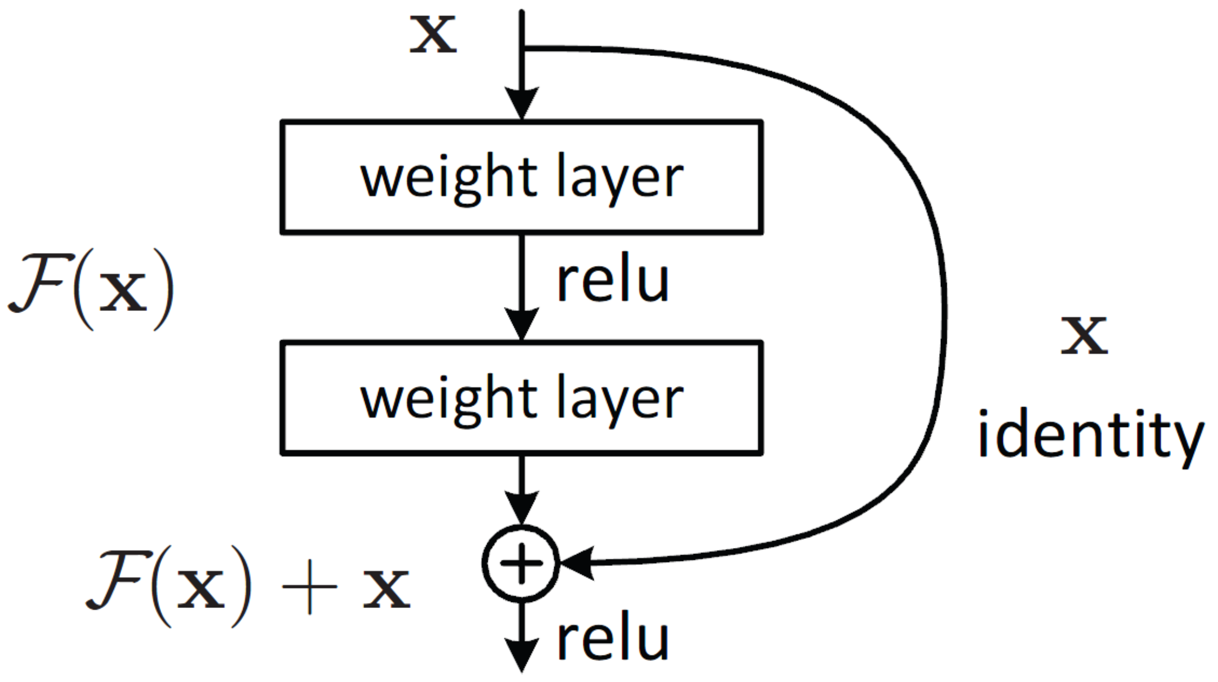
VGG


VGG Architecture
3x3 convolutions
Broken up by 2x2 pooling layers
Finished with 3 fully connected layers

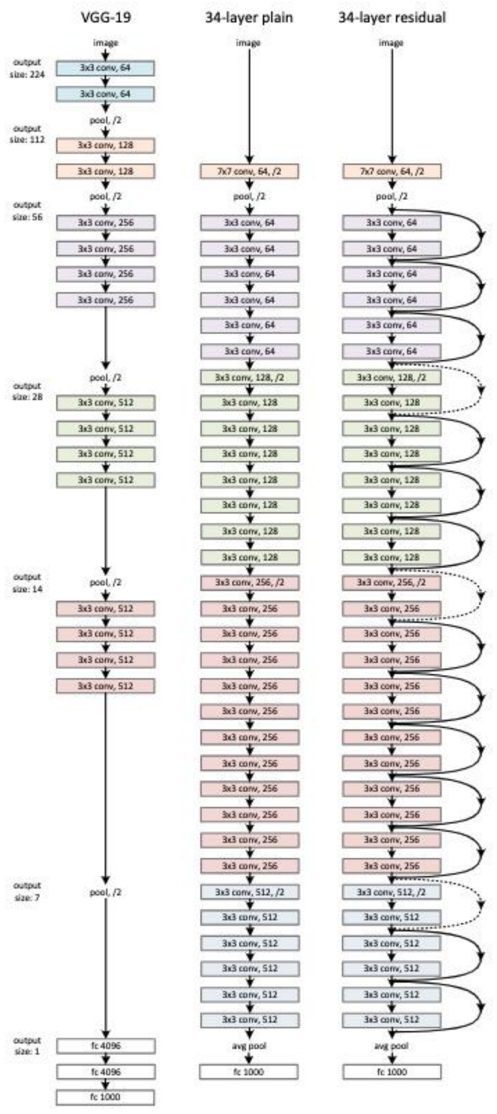
- 2014 年
- 牛津大学的视觉几何
- VGG16 和 VGG19，分别有 16 个层级和19个层级。
- 由一个很长的 3 \times 33×3 卷积序列，穿插着 2 \times 22×2 的池化层，最后是 3 个全连接层。

GooLeNet

ResNet



$\mathbf{x}$
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
$\mathbf{x}$
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

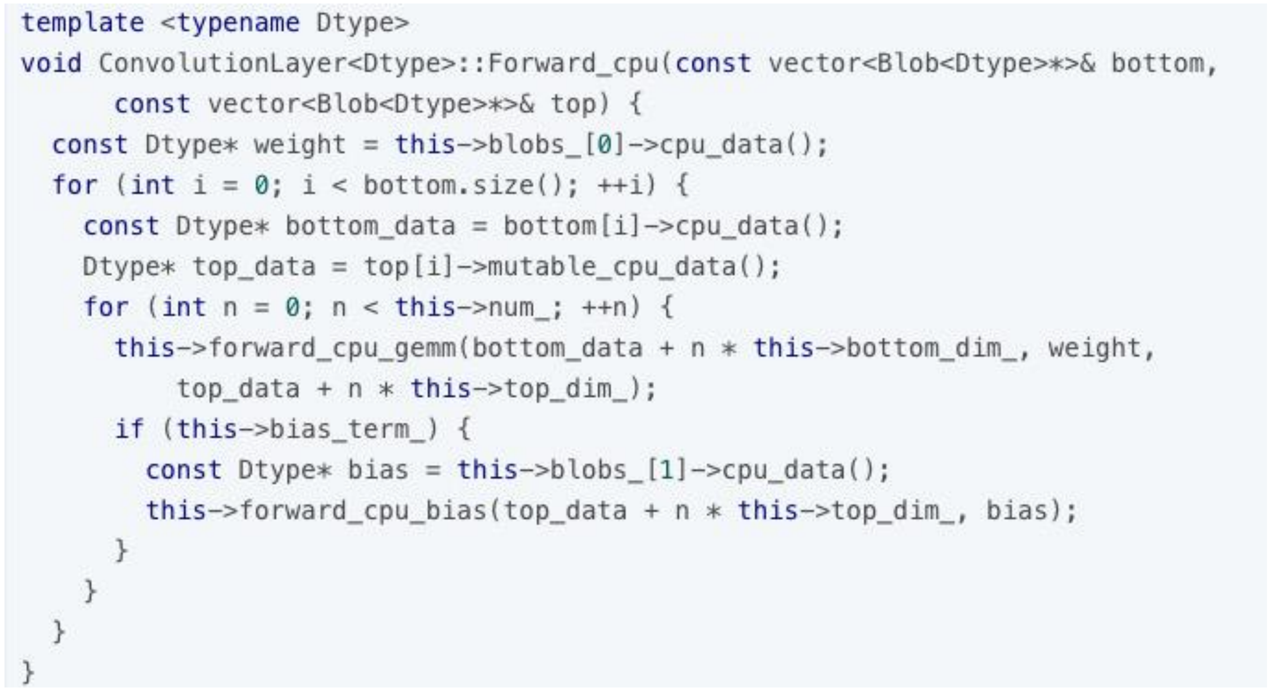


VGG-19
34-layer plain
34-layer residual
image
output size: 224
3x3 conv, 64
3x3 conv, 64
pool, /2
output size: 112
3x3 conv, 128
3x3 conv, 128
pool, /2
output size: 56
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
pool, /2
output size: 28
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 14
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 7
output size: 1
fc 4096
fc 4096
fc 1000
image
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
avg pool
fc 1000
image
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
avg pool
fc 1000

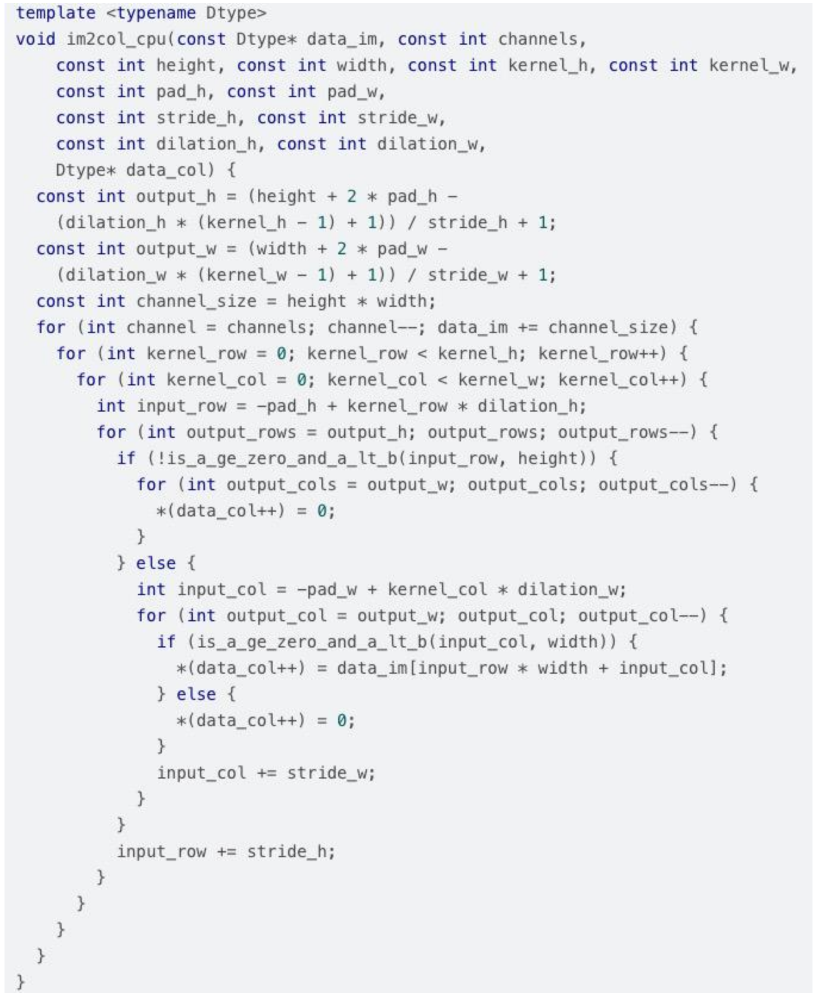
卷积计算常用方法

- 滑动窗口：计算比较慢，一般不采用。
- im2col：主流计算框架包括 Caffe, MXNet 等都实现了该方法。该方法把整个卷积过程转化成了 GEMM 过程， GEMM 在各种 BLAS 库中都是被极致优化的，一般来说，速度较快。
- FFT: 傅里叶变换和快速傅里叶变化是在经典图像处理里面经常使用的计算方法，但是，在 ConvNet 中通常不采用，主要是因为在 ConvNet 中的卷积模板通常都比较小，例如 3×3 等，这种情况下，FFT 的时间开销反而更大。
- Winograd：Winograd 方法都显示和较大的优势，目前 CUDNN 中计算卷积就使用了该方法。

卷积计算-Forward



```
template <typename Dtype>
void ConvolutionLayer<Dtype>::Forward_cpu(const vector<Blob<Dtype>*>& bottom,
      const vector<Blob<Dtype>*>& top) {
  const Dtype* weight = this->blobs_[0]->cpu_data();
  for (int i = 0; i < bottom.size(); ++i) {
    const Dtype* bottom_data = bottom[i]->cpu_data();
    Dtype* top_data = top[i]->mutable_cpu_data();
    for (int n = 0; n < this->num_; ++n) {
      this->forward_cpu_gemm(bottom_data + n * this->bottom_dim_, weight,
          top_data + n * this->top_dim_);
      if (this->bias_term_) {
        const Dtype* bias = this->blobs_[1]->cpu_data();
        this->forward_cpu_bias(top_data + n * this->top_dim_, bias);
      }
    }
  }
}
```



```
template <typename Dtype>
void im2col_cpu(const Dtype* data_im, const int channels,
    const int height, const int width, const int kernel_h, const int kernel_w,
    const int pad_h, const int pad_w,
    const int stride_h, const int stride_w,
    const int dilation_h, const int dilation_w,
    Dtype* data_col) {
  const int output_h = (height + 2 * pad_h -
    (dilation_h * (kernel_h - 1) + 1)) / stride_h + 1;
  const int output_w = (width + 2 * pad_w -
    (dilation_w * (kernel_w - 1) + 1)) / stride_w + 1;
  const int channel_size = height * width;
  for (int channel = channels; channel--; data_im += channel_size) {
    for (int kernel_row = 0; kernel_row < kernel_h; kernel_row++) {
      for (int kernel_col = 0; kernel_col < kernel_w; kernel_col++) {
        int input_row = -pad_h + kernel_row * dilation_h;
        for (int output_rows = output_h; output_rows; output_rows--) {
          if (!is_a_ge_zero_and_a_lt_b(input_row, height)) {
            for (int output_cols = output_w; output_cols; output_cols--) {
              *(data_col++) = 0;
            }
          } else {
            int input_col = -pad_w + kernel_col * dilation_w;
            for (int output_col = output_w; output_col; output_col--) {
              if (is_a_ge_zero_and_a_lt_b(input_col, width)) {
                *(data_col++) = data_im[input_row * width + input_col];
              } else {
                *(data_col++) = 0;
              }
              input_col += stride_w;
            }
          }
          input_row += stride_h;
        }
      }
    }
  }
}
```

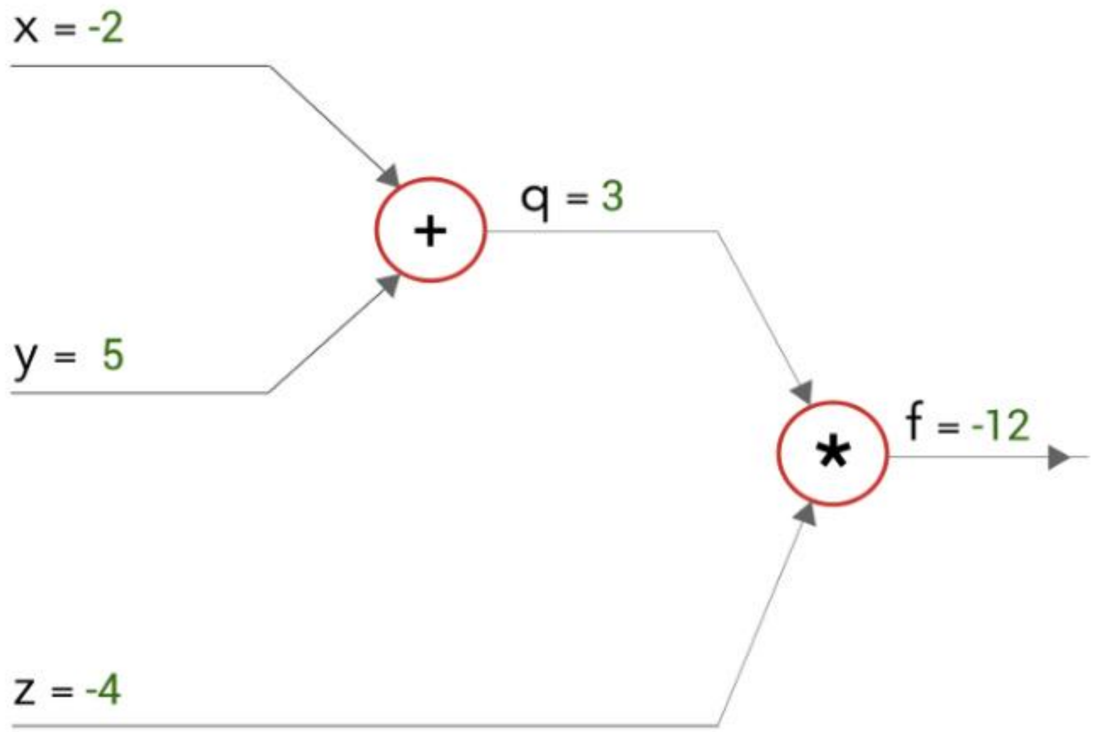
反向传播基础1 （BackPropagation ）

$f(x,y,z) = (x + y)z$

$$f(x,y,z) = (x+y)z$$

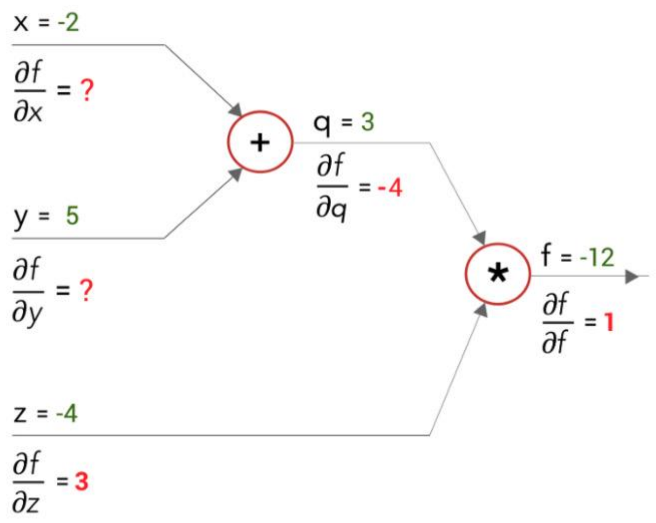
$$q = x + y$$

$$f = q * z$$



Computational Graph of f = q*z where q = x + y

反向传播算法基础2



$x = -2$
$\frac{\partial f}{\partial x} = ?$
$y = 5$
$\frac{\partial f}{\partial y} = ?$
$z = -4$
$\frac{\partial f}{\partial z} = 3$
+
$q = 3$
$\frac{\partial f}{\partial q} = -4$
*
$f = -12$
$\frac{\partial f}{\partial f} = 1$

$$f = q * z$$

$$\frac{\partial f}{\partial q} = z \mid z = -4$$

$$\frac{\partial f}{\partial z} = q \mid q = 3$$

$$q = x + y$$

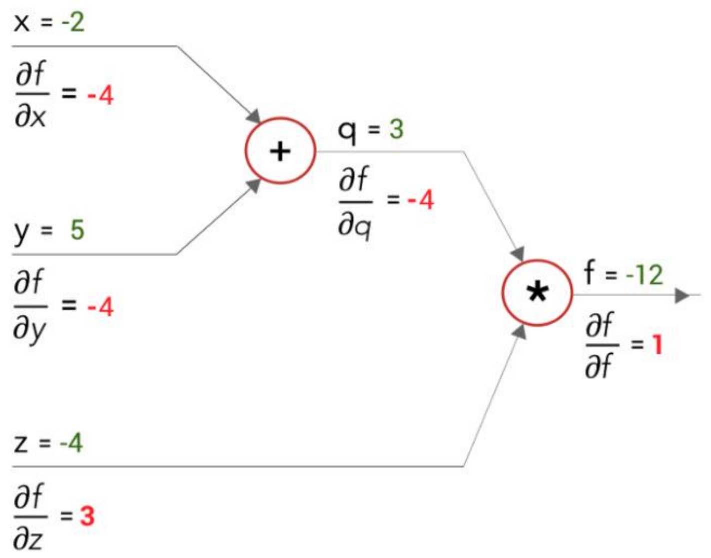
$$\frac{\partial q}{\partial x} = 1 \qquad \frac{\partial q}{\partial y} = 1$$

Using chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} * \frac{\partial q}{\partial x}$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} * \frac{\partial q}{\partial x} = -4 * 1 = -4$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} * \frac{\partial q}{\partial y} = -4 * 1 = -4$$



$x = -2$
$\frac{\partial f}{\partial x} = -4$
$y = 5$
$\frac{\partial f}{\partial y} = -4$
$z = -4$
$\frac{\partial f}{\partial z} = 3$
+
$q = 3$
$\frac{\partial f}{\partial q} = -4$
*
$f = -12$
$\frac{\partial f}{\partial f} = 1$

Max-pooling Backpropagation

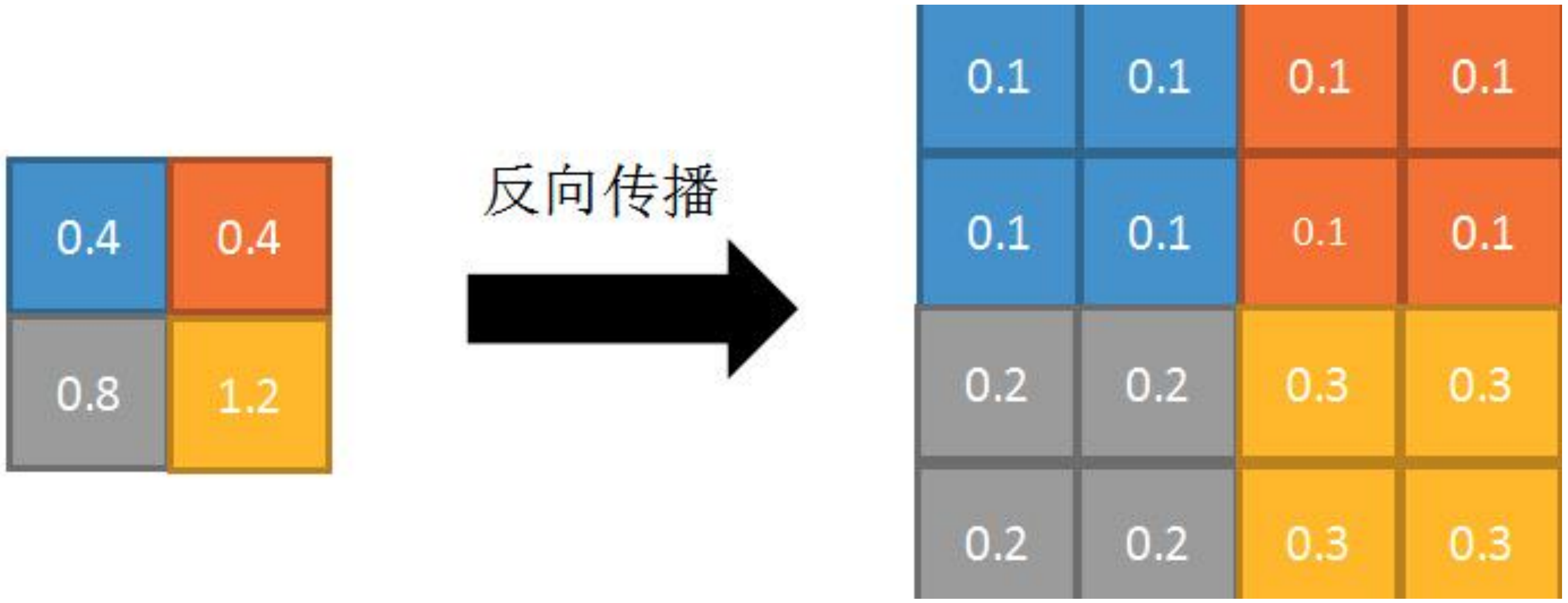
$$y = \max(x_1, x_2, \cdots, x_n)$$

$$y = \sum_{i=1}^{n} w_i x_i$$

$$\begin{aligned} \frac{\partial y}{\partial x_i} &= w_i \\ &= \begin{cases} 1 & \text{if } x_i = \max(x_1, x_2, \cdots, x_n) \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

$$w_i = \begin{cases} 1 & \text{if } x_i = \max(x_1, x_2, \cdots, x_n) \\ 0 & \text{otherwise} \end{cases}$$

Average-Pooling



卷积反向传播1

卷积反向传播2

卷积反向传播3

卷积反向传播4

$$\frac{\partial L}{\partial X_{11}} = \frac{\partial L}{\partial O_{11}} * F_{11}$$

$$\frac{\partial L}{\partial X_{12}} = \frac{\partial L}{\partial O_{11}} * F_{12} + \frac{\partial L}{\partial O_{12}} * F_{11}$$

$$\frac{\partial L}{\partial X_{13}} = \frac{\partial L}{\partial O_{12}} * F_{12}$$

$$\frac{\partial L}{\partial X_{21}} = \frac{\partial L}{\partial O_{11}} * F_{21} + \frac{\partial L}{\partial O_{21}} * F_{11}$$

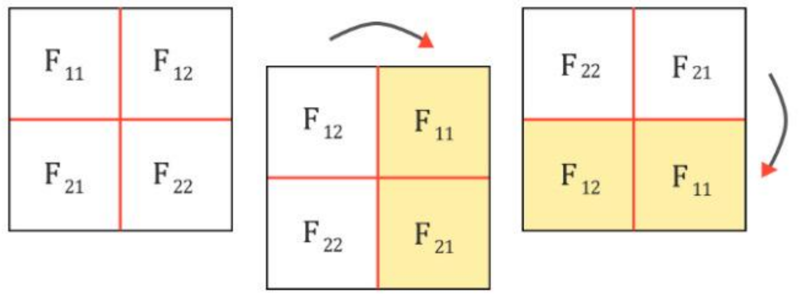
$$\frac{\partial L}{\partial X_{22}} = \frac{\partial L}{\partial O_{11}} * F_{22} + \frac{\partial L}{\partial O_{12}} * F_{21} + \frac{\partial L}{\partial O_{21}} * F_{12} + \frac{\partial L}{\partial O_{22}} * F_{11}$$

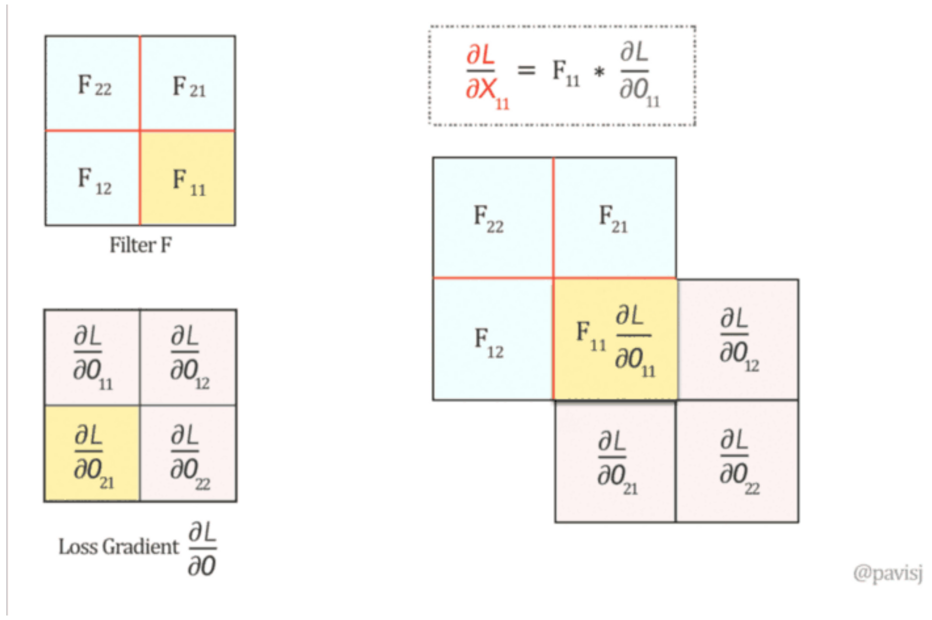
$$\frac{\partial L}{\partial X_{23}} = \frac{\partial L}{\partial O_{12}} * F_{22} + \frac{\partial L}{\partial O_{22}} * F_{12}$$

$$\frac{\partial L}{\partial X_{31}} = \frac{\partial L}{\partial O_{21}} * F_{21}$$

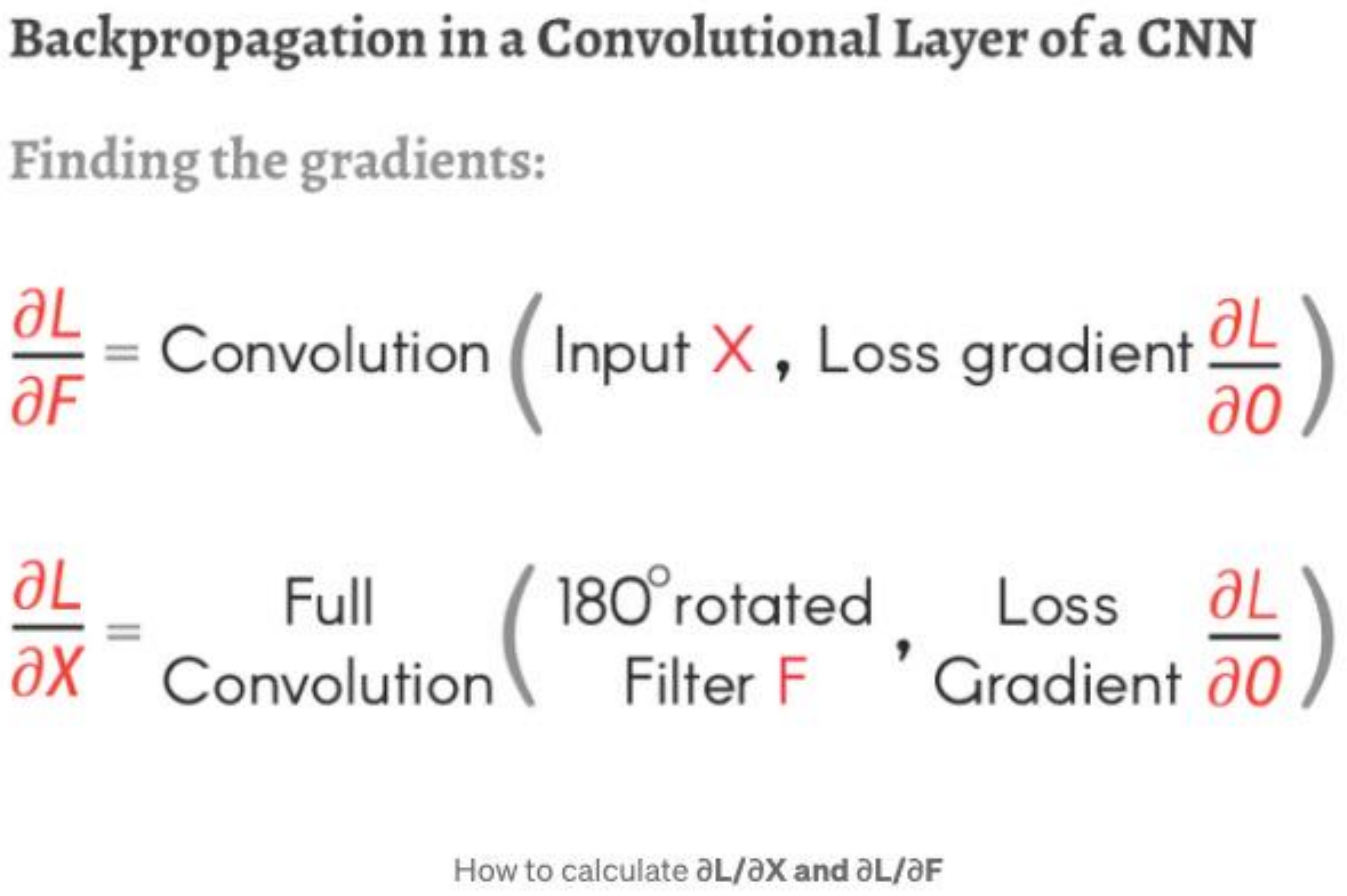
$$\frac{\partial L}{\partial X_{32}} = \frac{\partial L}{\partial O_{21}} * F_{22} + \frac{\partial L}{\partial O_{22}} * F_{21}$$

$$\frac{\partial L}{\partial X_{33}} = \frac{\partial L}{\partial O_{22}} * F_{22}$$




F_{22} F_{21} F_{12} F_{11}
Filter F
$\frac{\partial L}{\partial O_{11}}$ $\frac{\partial L}{\partial O_{12}}$ $\frac{\partial L}{\partial O_{21}}$ $\frac{\partial L}{\partial O_{22}}$
Loss Gradient $\frac{\partial L}{\partial O}$
$\frac{\partial L}{\partial X_{11}} = F_{11} * \frac{\partial L}{\partial O_{11}}$
@pavisj

卷积反向传播5



Backpropagation in a Convolutional Layer of a CNN

Finding the gradients:

$$\frac{\partial L}{\partial F} = \text{Convolution}\left(\text{Input } X,\ \text{Loss gradient } \frac{\partial L}{\partial O}\right)$$

$$\frac{\partial L}{\partial X} = \begin{matrix}\text{Full}\\ \text{Convolution}\end{matrix}\left(\begin{matrix}180^{\circ}\text{rotated}\\ \text{Filter } F\end{matrix},\ \begin{matrix}\text{Loss}\\ \text{Gradient}\end{matrix}\ \frac{\partial L}{\partial O}\right)$$

How to calculate **∂L/∂X and ∂L/∂F**

Winograd

计算复杂度：

$$\mu\left(F\left(m \times n, r \times s\right)\right) = (m + n - 1) \times (n + s - 1)$$

计算方法：

$$\boxed{Y = A^T \left[\left[G g G^T\right] \odot \left[B^T d B\right]\right] A}$$

Winograd缺点

- Depthwise conv 中其优势不明显了。
- 在 tile 较大的时候，Winograd 方法不适用，inverse transform 计算开销抵消了 Winograd 带来的计算节省。
- Winograd 会产生误差

Pytorch自定义算子

```python
class ScipyConv2dFunction(Function):
    @staticmethod
    def forward(ctx, input, filter, bias):
        # detach so we can cast to NumPy
        input, filter, bias = input.detach(), filter.detach(), bias.detach()
        result = correlate2d(input.numpy(), filter.numpy(), mode='valid')
        result += bias.numpy()
        ctx.save_for_backward(input, filter, bias)
        return torch.as_tensor(result, dtype=input.dtype)

    @staticmethod
    def backward(ctx, grad_output):
        grad_output = grad_output.detach()
        input, filter, bias = ctx.saved_tensors
        grad_output = grad_output.numpy()
        grad_bias = np.sum(grad_output, keepdims=True)
        grad_input = convolve2d(grad_output, filter.numpy(), mode='full')
        # the previous line can be expressed equivalently as:
        # grad_input = correlate2d(grad_output, flip(flip(filter.numpy(), axis=0), axis=1),
        grad_filter = correlate2d(input.numpy(), grad_output, mode='valid')
        return torch.from_numpy(grad_input), torch.from_numpy(grad_filter).to(torch.float),
```

Pytorch自定义算子

```python
class ScipyConv2d(Module):
    def __init__(self, filter_width, filter_height):
        super(ScipyConv2d, self).__init__()
        self.filter = Parameter(torch.randn(filter_width, filter_height))
        self.bias = Parameter(torch.randn(1, 1))

    def forward(self, input):
        return ScipyConv2dFunction.apply(input, self.filter, self.bias)
```

Pytorch自定义算子

```python
from setuptools import setup, Extension
from torch.utils import cpp_extension

setup(name='lltm_cpp',
      ext_modules=[cpp_extension.CppExtension('lltm_cpp', ['lltm.cpp'])],
      cmdclass={'build_ext': cpp_extension.BuildExtension})
```

```python
Extension(
   name='lltm_cpp',
   sources=['lltm.cpp'],
   include_dirs=cpp_extension.include_paths(),
   language='c++')
```

```cpp
PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
  m.def("forward", &lltm_forward, "LLTM forward");
  m.def("backward", &lltm_backward, "LLTM backward");
}
```

```cpp
#include <vector>

std::vector<at::Tensor> lltm_forward(
    torch::Tensor input,
    torch::Tensor weights,
    torch::Tensor bias,
    torch::Tensor old_h,
    torch::Tensor old_cell) {
  auto X = torch::cat({old_h, input}, /*dim=*/1);

  auto gate_weights = torch::addmm(bias, X, weights.transpose(0, 1));
  auto gates = gate_weights.chunk(3, /*dim=*/1);

  auto input_gate = torch::sigmoid(gates[0]);
  auto output_gate = torch::sigmoid(gates[1]);
  auto candidate_cell = torch::elu(gates[2], /*alpha=*/1.0);

  auto new_cell = old_cell + candidate_cell * input_gate;
  auto new_h = torch::tanh(new_cell) * output_gate;

  return {new_h,
          new_cell,
          input_gate,
          output_gate,
          candidate_cell,
          X,
          gate_weights};
}
```

作业

- 2D Conv的 Pytorch插件实现

要求：

(1) 支持padding和stride的参数化

(2) 支持正向和反向两个过程

签到&反馈