Convolution Layer

严格来说，卷积层是个错误的叫法，因为它所表达的运算其实是互相关运算（cross-correlation），而不是卷积运算。

Convolution Operation

在卷积操作中，卷积核在输入矩阵上移动，并在每个位置进行加权和运算。

公式推导

- **输入图像**：大小为 H×W
- **卷积核**：大小为 K×K
- **步幅（Stride）**：卷积核在输入图像上滑动的步长。
- **填充（Padding）**：在输入图像的边缘填充的像素数。

卷积核的每次移动都会计算一个输出像素。步幅决定了卷积核每次移动的距离。

$$\text{Output Height} = \frac{H + 2 \times P - K}{S} + 1$$

$$\text{Output Weight} = \frac{W + 2 \times P - K}{S} + 1$$

使输出大小等于输入大小的条件

为了使卷积层的输出大小与输入大小相同，通常需要适当地设置填充。我们设定步幅 S=1，卷积核大小为 K×K

在这种情况下，为了保持输出大小与输入大小相同，填充的计算公式为：

$$\text{Padding} = \frac{K - 1}{2}$$

Traditional Convolutional Layer Code

```
# 互相关运算
import torch
from torch import nn
# 定义一个stride=1，padding=0的卷积层
def corr2d(X, K):   # X为输入，K为卷积核
    h, w = K.shape
    Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i:i + h, j:j + w] * K).sum() # 图片的小方块区域与卷积核做点积
    return Y

# 验证上述二维互相关运算的输出
X = torch.tensor([[0.0,1.0,2.0],[3.0,4.0,5.0],[6.0,7.0,8.0]])
K = torch.tensor([[0.0,1.0],[2.0,3.0]])
print(corr2d(X,K))
```

图像中目标的边缘检测

```
# 由X生成Y的卷积核
conv2d = nn.Conv2d(1, 1, kernel_size=(1,2), bias=False) # 单个矩阵，输入通道为1，黑白图片通道为1，彩色图片通道为3。这里输入通道为1，输出通道为1.
X = X.reshape((1,1,6,8)) # 通道维：通道数，RGB图3通道，灰度图1通道，批量维就是样本维，就是样本数
Y = Y.reshape((1,1,6,7))
for i in range(10):
    Y_hat = conv2d(X)
    l = (Y_hat - Y) ** 2
    conv2d.zero_grad()
    l.sum().backward()
    conv2d.weight.data[:] -= 3e-2 * conv2d.weight.grad # 3e-2是学习率
    if(i+1) % 2 == 0:
        print(f'batch {i+1},loss {l.sum():.3f}')

# 所学的卷积核的权重张量
print(conv2d.weight.data.reshape((1,2)))
'''
Y_hat = conv2d(X)`：通过卷积层计算 `X` 的预测输出 `Y_hat`。

`l = (Y_hat - Y) ** 2`：计算 `Y_hat` 和目标 `Y` 之间的均方误差。

`conv2d.zero_grad()`：清除卷积层的梯度。

`l.sum().backward()`：反向传播，计算梯度。

`conv2d.weight.data[:] -= 3e-2 * conv2d.weight.grad`：使用学习率 3e-23e-23e-2 更新卷积核的权重。

每两个批次打印一次损失值 `l.sum()`，以监控训练过程。
'''
```