



# 《人工智能与Python程序设计》——卷积神经网络



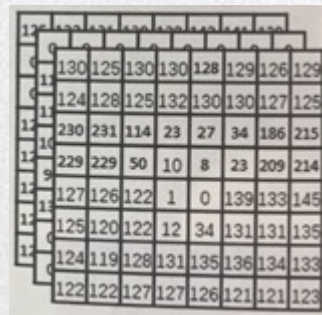
人工智能与Python程序设计 教研组

# 数字图像

- 数字图像是连续的光信号经过传感器的采样在空间域上的表达。一张图像是由一个包含若干个像素点的矩形框组成的。
- 每个小格子是一个像素
- 灰度值需要1个维度，表示像素的灰度值
- 彩色值需要3个维度，也就是3个图像通道：每个像素3个数字表示。



我们看到的图像



计算机看到的图像



# 复习

- 屏幕像素&屏幕分辨率



## 图像分辨率





# PyTorch实现卷积

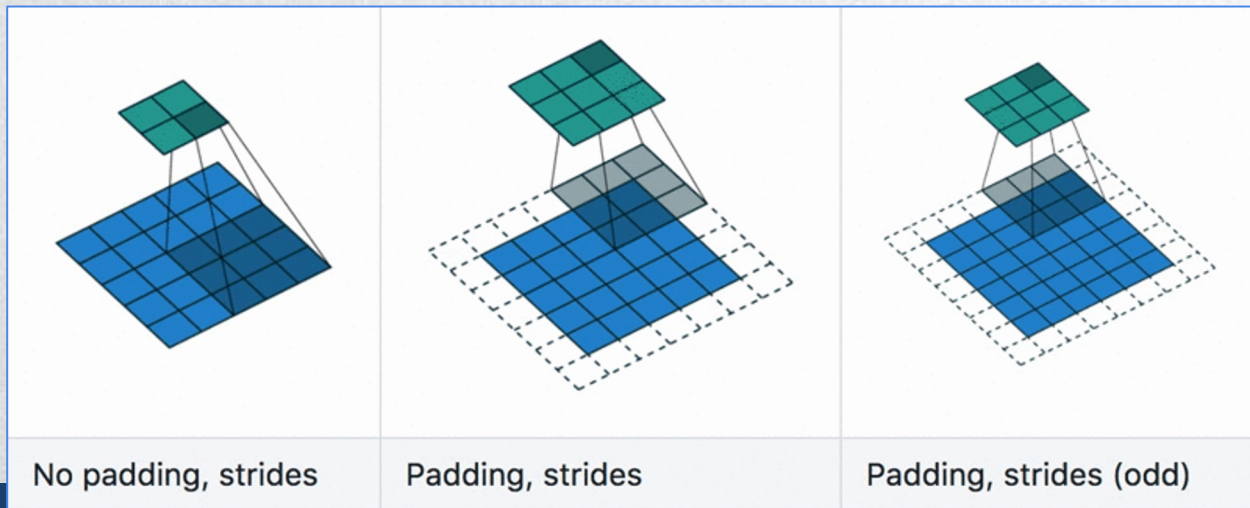
- `torch.nn.functional.conv2d(input, weight, bias=None, stride=1, padding=0, dilation=1, groups=1)`
  - `input`: 输入张量 (`batchsize x in_channels x iH x iW`) , `iH`和`iW`为输入tensor的高、宽
  - `weight`: 过滤器 (卷积核) 张量 (`out_channels, in_channels/groups, kH, kW`) , `kH`和`kW`为卷积核的高、宽
  - `bias`: 可选偏置张量 (`out_channels`)
  - `stride`: 卷积核的步长, 可以是单个数字或一个元组 (`sh x sw`)
  - `padding`: 输入上隐含零填充。可以是单个数字或元组
  - `groups`: 将输入分成组, `in_channels`应该被组数除尽



# 卷积计算输出大小

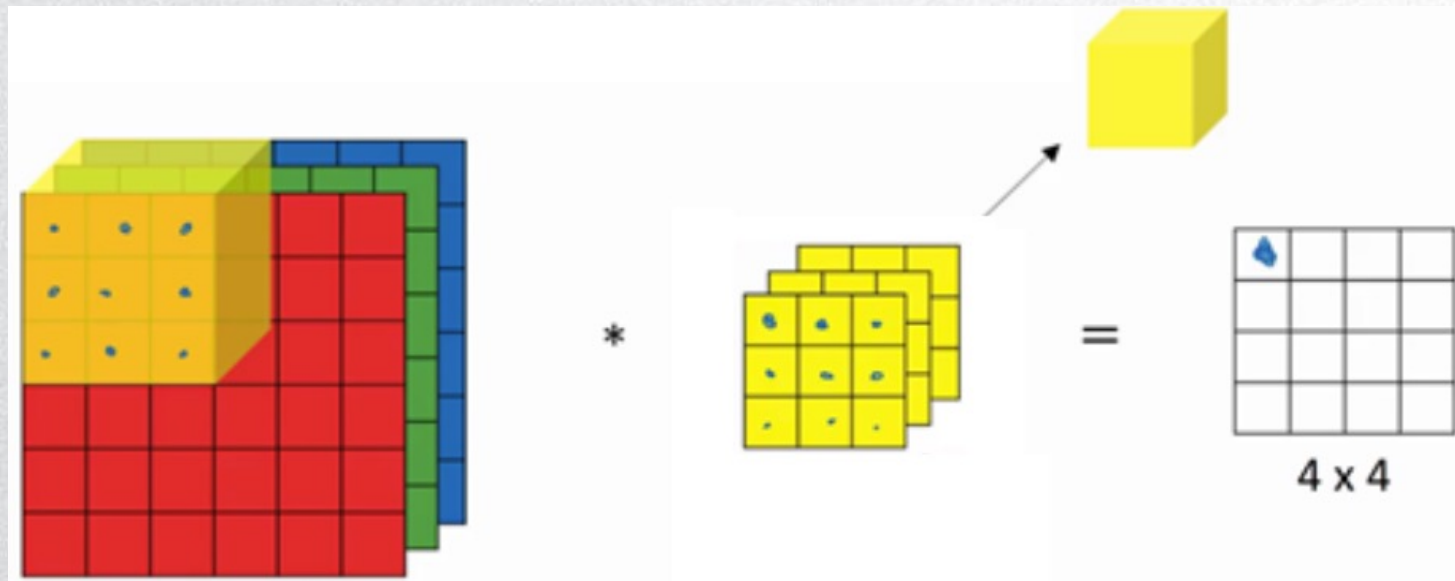
- $n \times n$  的图像，用  $f \times f$  的滤波器进行卷积，padding =  $p$ ，步长设为  $s$ ，

$$\begin{array}{cc} n \times n \text{ image} & f \times f \text{ filter} \\ \text{padding } p & \text{stride } s \\ \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \end{array}$$



# 多通道卷积

- 滤波器通道数和图像通道数要匹配
- 取出27个像素值，依次和滤波器对应参数相乘，求和



# 多通道卷积

# 多通道卷积 (RGB)

```
image = totensor(image).unsqueeze(0)
```

```
w = torch.tensor([[[[1., 0., -1.],  
——x——x——x——x [2., 0., -2.],  
——x——x——x——x [1., 0., -1.]]]])
```

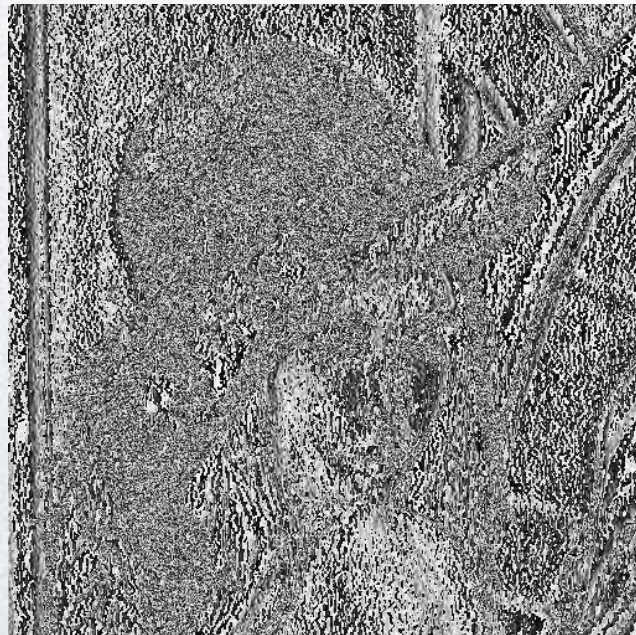
```
w = w.repeat(1, 3, 1, 1)
```

```
convv = F.conv2d(image, w, stride=1, padding=0)
```

```
convv = convv.squeeze(0)
```

```
convv = toPIL(convv)
```

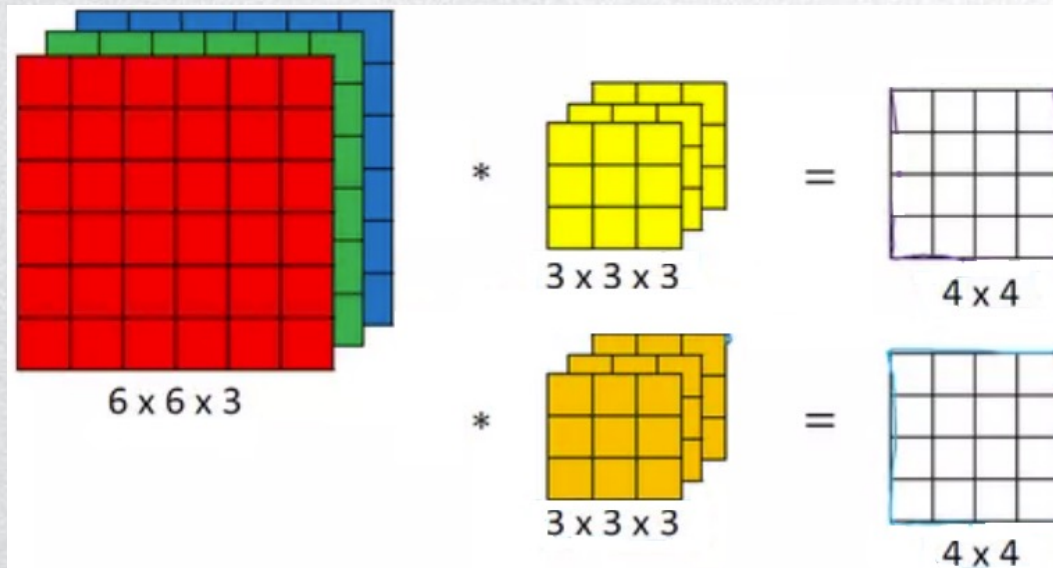
```
convv.show()
```





# 多卷积核

- 不仅检测垂直边缘，还需要同时检测垂直边缘和水平边缘
- 同时用多个过滤器：把输出堆叠在一起，得到 $4 \times 4 \times 2$ 的输出立方体







# 多卷积核



# 多卷积核

```
w2 = torch.tensor([[[[1., 2., 1.],  
——x——x——x——x [0., 0., 0.],  
——x——x——x——x [-1., -2., -1.]]]]])
```

```
w2 = w2.repeat(1, 3, 1, 1)
```

```
w3 = torch.cat((w, w2), 0)
```

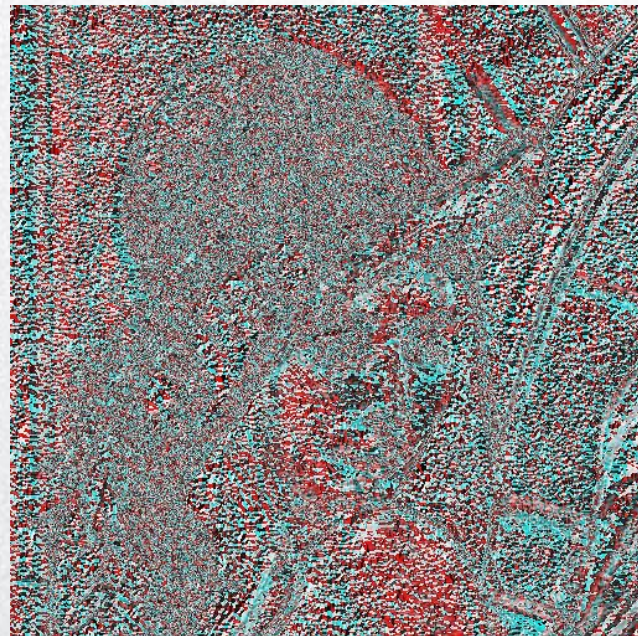
```
w3 = torch.cat((w3, w2), 0)
```

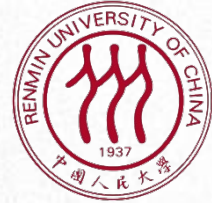
```
convh = F.conv2d(image, w3, stride=1, padding=0)
```

```
convh = convh.squeeze(0)
```

```
convh = toPIL(convh)
```

```
convh.show()
```





# 提纲



## 卷积神经网络

- □ 单个卷积层
- □ 池化层
- □ 卷积神经网络
-

# 手写数字识别

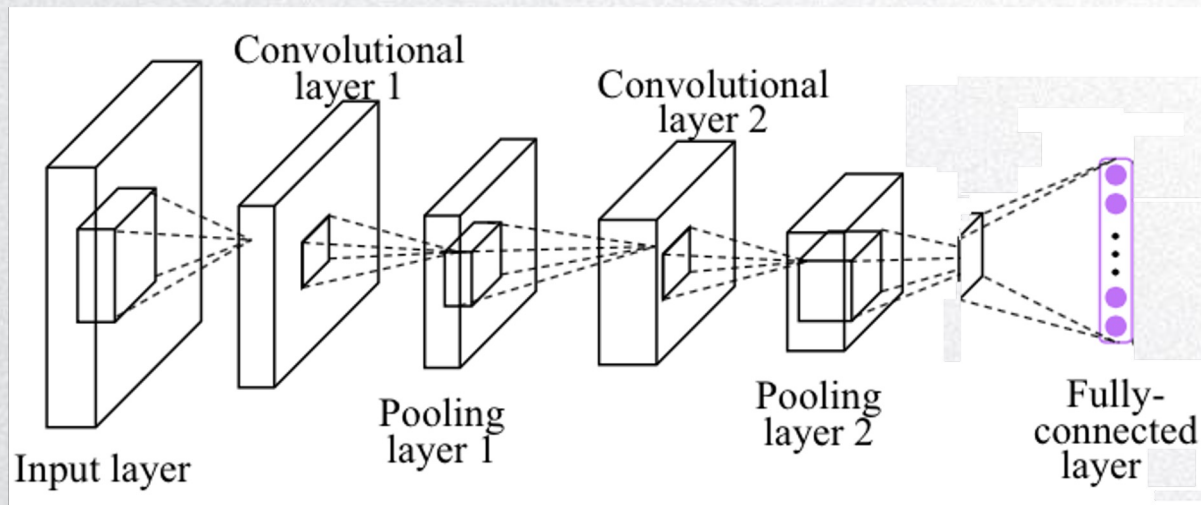
- $3 \times 32 \times 32$ 的RGB图片中含有某个数字，识别它是从0-9这10个数字中的哪一个
- 分类问题





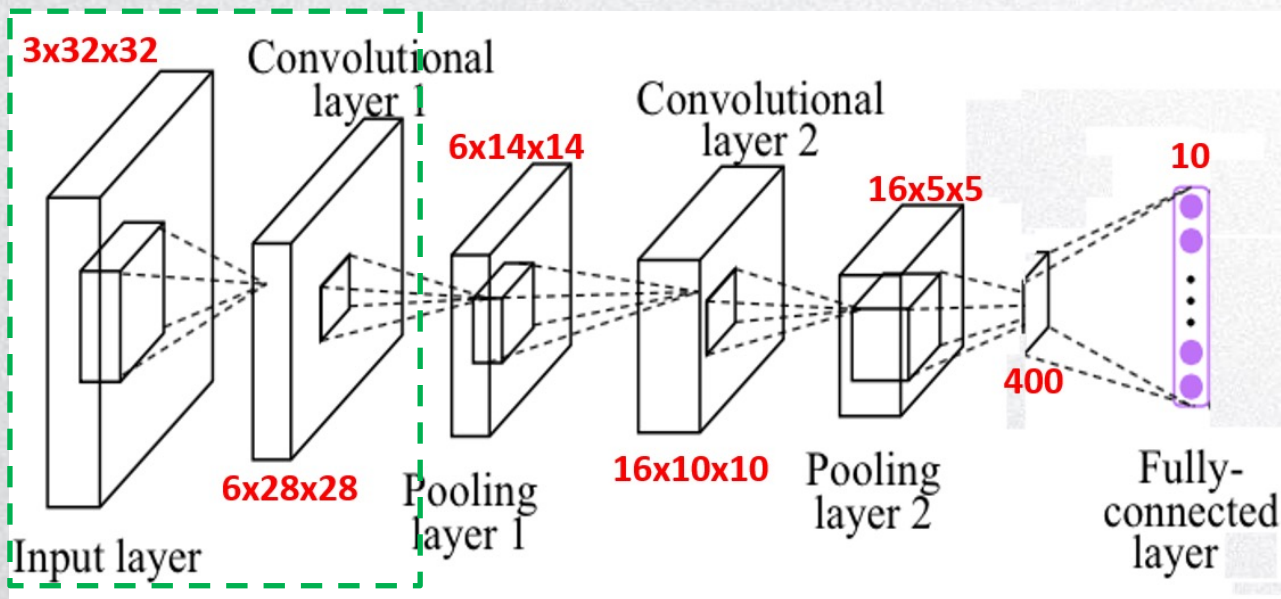
# 卷积神经网络

- 类似LeNet-5 (Yann LeCun创建)



# 卷积神经网络

- 输入是 $3 \times 32 \times 32$ 的矩阵，第一层使用滤波器大小为 $5 \times 5$ ，步幅是1，padding是0，滤波器个数为6，那么输出为 $6 \times 28 \times 28$ 。将这层标记为CONV1，增加偏差，应用非线性函数（如ReLU）后输出结果。

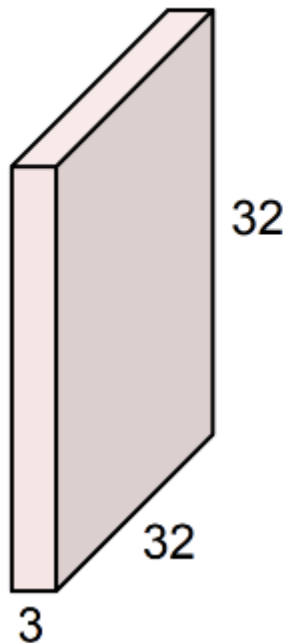


# 单个卷积层

- 输入图片大小是 $3 \times 32 \times 32$
- 用 $3 \times 5 \times 5$ 的卷积核

- 输出:  $(32+0-5) / 1 + 1$

32x32x3 image



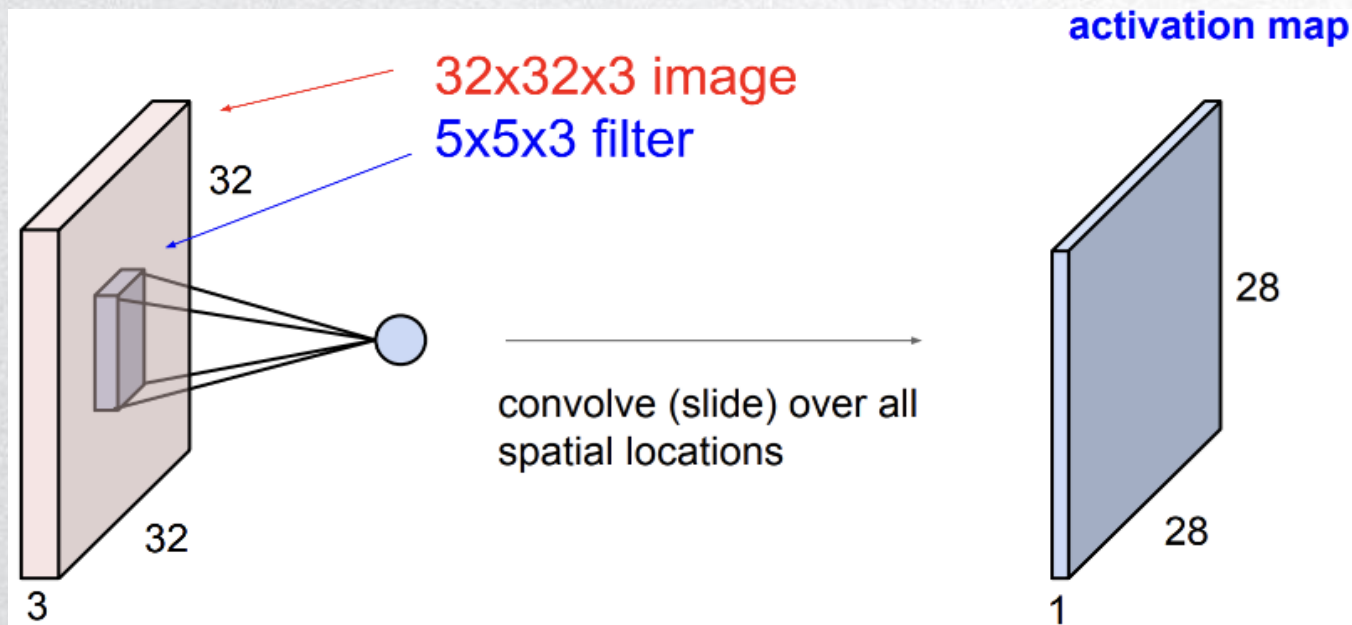
5x5x3 filter





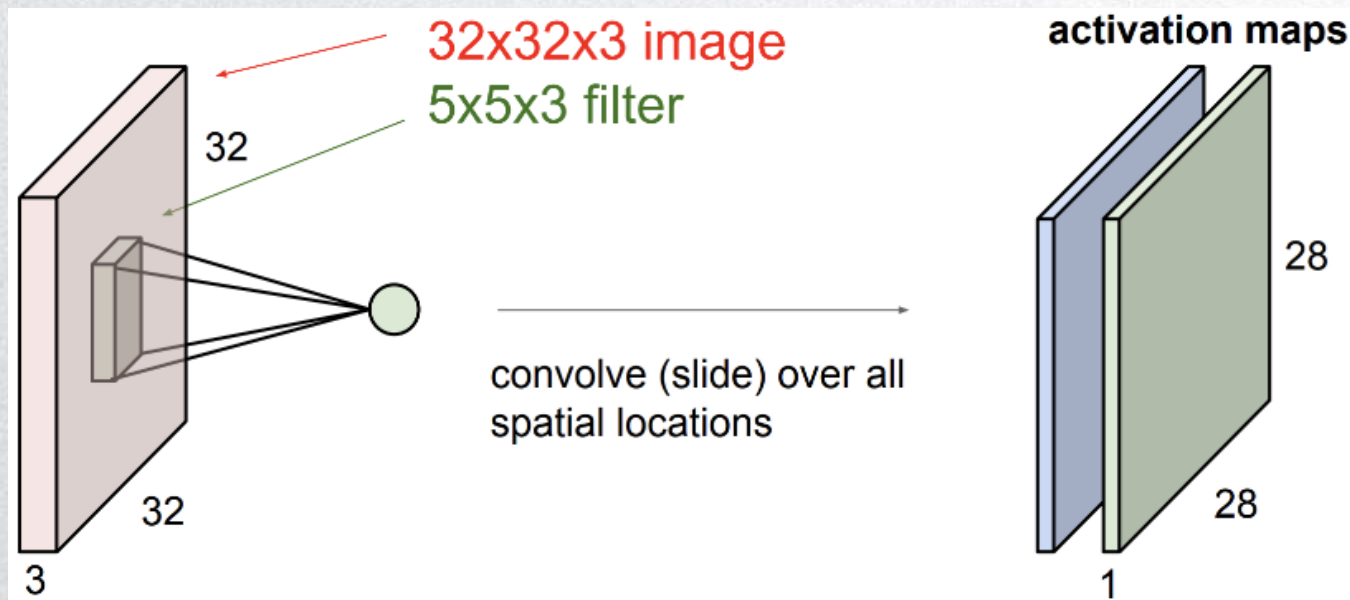
# 单个卷积层

- 单个卷积核得到特征图



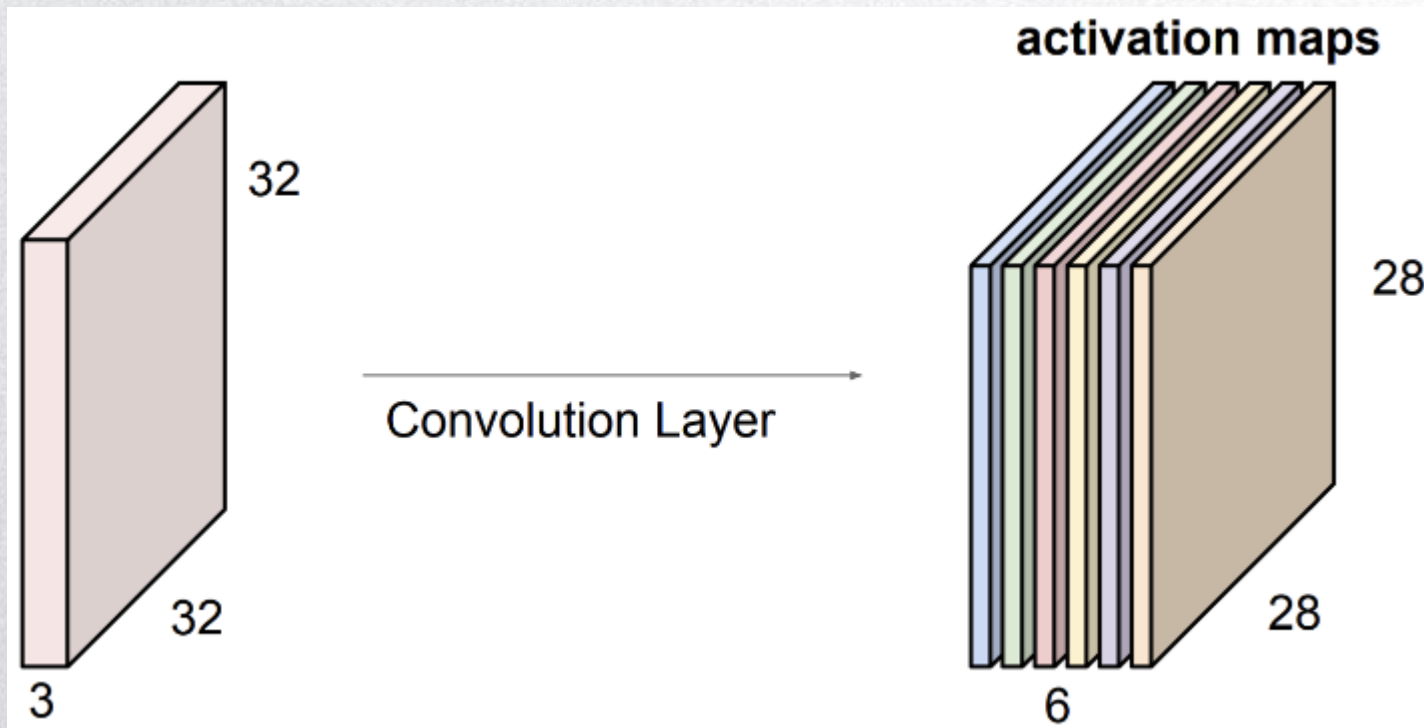
# 单个卷积层

- 每个卷积核得到一个特征图



# 单个卷积层

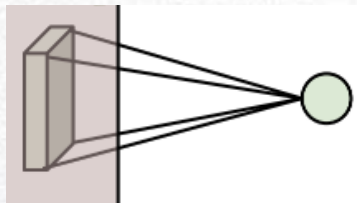
- 总共有6个卷积核





# 单个卷积层

- 前向传播（在每个局部）  $z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$
  - 非线性函数： $a^{[1]} = g(z^{[1]})$
  - 滤波器用变量  $W^{[1]}$  表示。
  - 每个局部共进行了多少次乘法运算和加法运算？
- 
- 应用激活函数ReLU，得到的 $6 \times 28 \times 28$ 矩阵成为神经网络的下一层的输入。
  - 通过这个过程把一个 $3 \times 32 \times 32$ 维度的输入图像变换为一个 $6 \times 28 \times 28$ 维度的特征图，这就是卷积神经网络的一层





# Pytorch卷积层

- Pytorch实现这些只需要一行代码
- `nn.Conv2d(self, in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True)`
  - `in_channel`: 输入数据的通道数;
  - `out_channel`: 输出数据的通道数;
  - `kennel_size`: 卷积核大小, 可以是int, 或tuple;
  - `stride`: 步长;
  - `padding`



# Pytorch卷积层

```
1  import torch
2  import torch.nn as nn
3  import numpy as np
4  from PIL import Image
5  from torchvision import transforms
6  import torch.nn.functional as F
7
8  image = Image.open('5.jpg').convert('RGB')
9  input = transforms.ToTensor()(image).unsqueeze(0)
10 print(input.shape)
11 # Conv-1
12 Conv1 = nn.Conv2d(in_channels=3,out_channels=6,kernel_size=5,stroke=1,padding=0)
13 Nf1 = nn.ReLU()
14 output1 = Nf1(Conv1(input))
15 print(output1.shape)
```

```
torch.Size([1, 3, 32, 32])
torch.Size([1, 6, 28, 28])
```







# 提纲

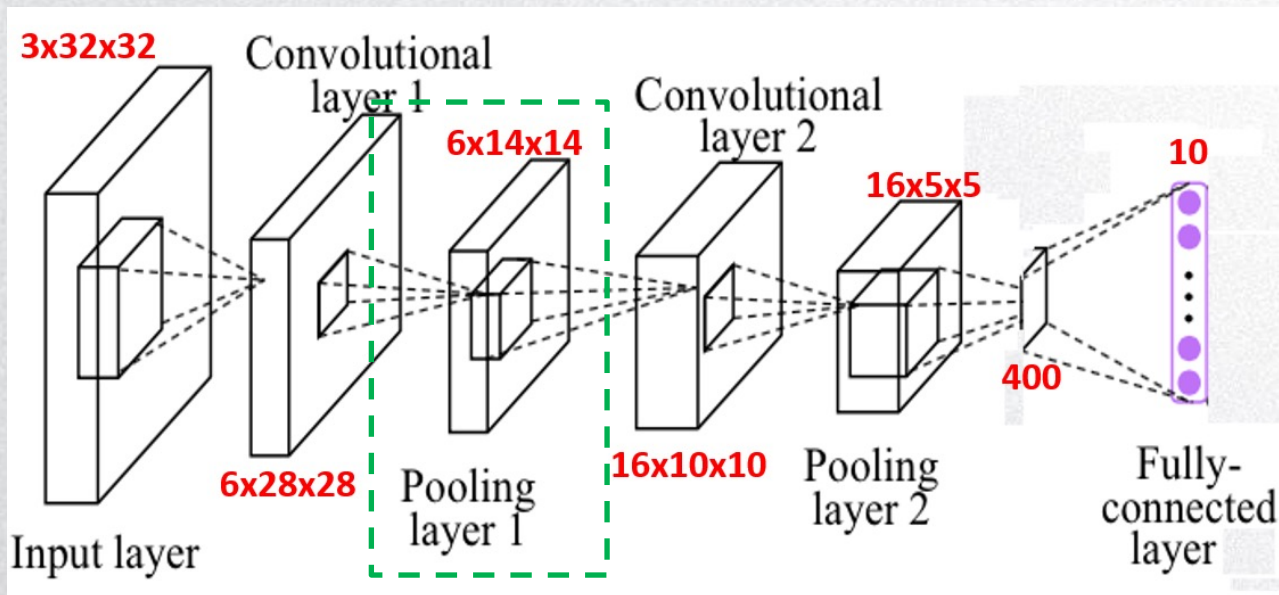


## 卷积神经网络

- □ 单个卷积层
- □ 池化层
- □ 卷积神经网络
-

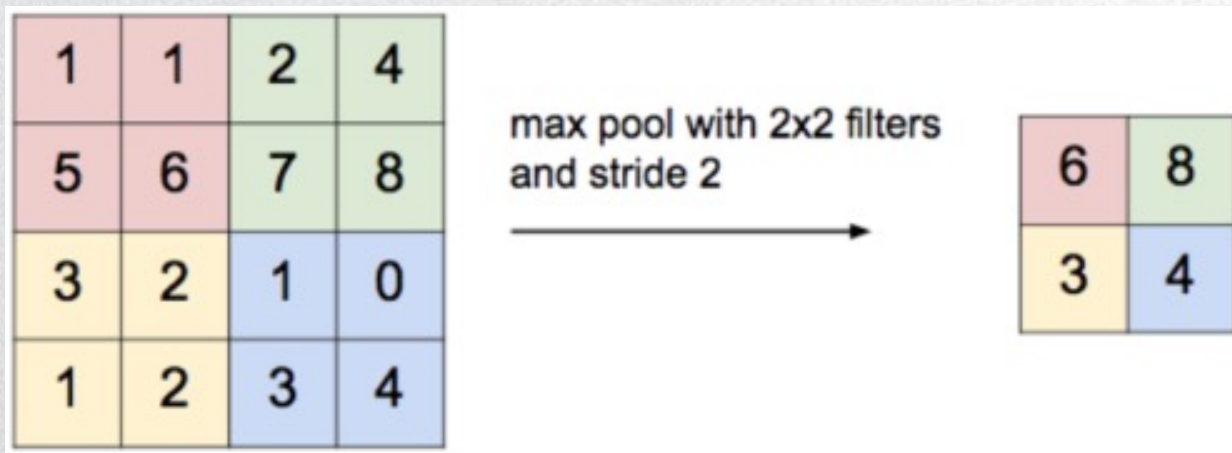
# 卷积神经网络

- 在第一个卷积层之后，构建一个池化层，这里选最大池化，过滤器为 $2 \times 2$ ，步幅为2，padding为0。最终输出为 $6 \times 14 \times 14$ ，将该层标记为**POOL1**。



# 池化层

- 特征图展开成向量可能维度很大
- 池化：缩减模型的大小；提高计算速度；提高所提取特征的鲁棒性
- Max pooling:





# 最大池化

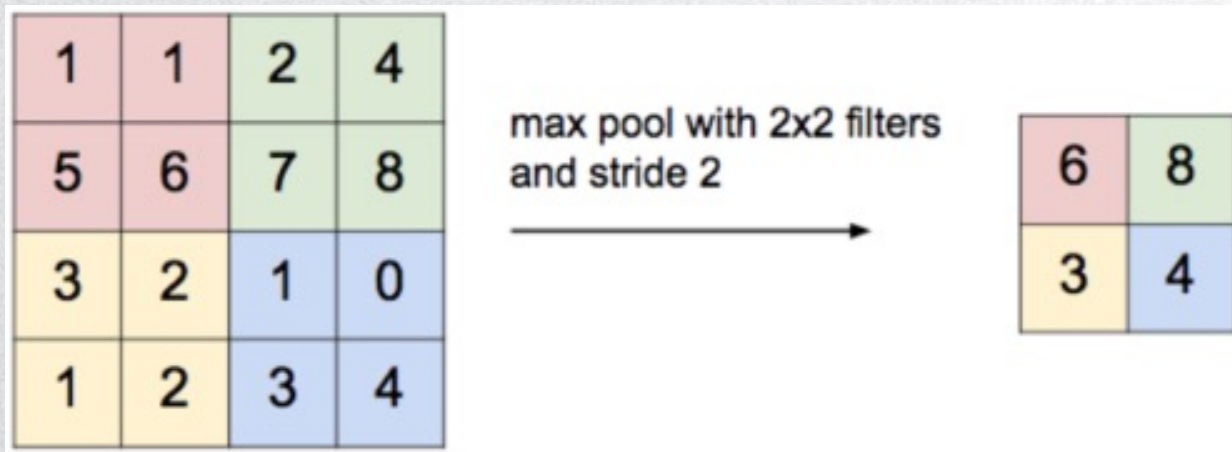
- 最大池化

3	3	2	1	0
0	0	1	3	1
3	1	2	2	3
2	0	0	2	2
2	0	0	0	1

3.0	3.0	3.0
3.0	3.0	3.0
3.0	2.0	3.0

# 最大池化

- 输入可能是神经网络中某一层的非激活值，可以看作是某些特征的集合。数字大意味着可能探测到了某些特定的特征。
- 如果在过滤器中提取到某个特征，那么保留其最大值。如果没有提取到这个特征，可能在左下象限中不存在这个特征，那么其中的最大值也还是很小。





# 最大池化

- 最大池化有一组超参数，但并没有参数需要学习。一旦确定了 $f$ 和 $s$ ，它就是一个固定运算，梯度下降无需改变任何值。
- 计算卷积层输出大小的公式同样适用于最大池化。
- 如果输入是三维的，那么输出也是三维的。例如，输入是 $nc \times 5 \times 5$ ， $f=3$ ， $s=1$ ，那么输出是 $nc \times 3 \times 3$ 。计算最大池化的方法就是分别对每个通道执行刚刚的计算过程。



# 平均池化

- 选取的不是每个过滤器的最大值，而是平均值
- $f=3, s=1$

3	3	2	1	0
0	0	1	3	1
3	1	2	2	3
2	0	0	2	2
2	0	0	0	1

1.7	1.7	1.7
1.0	1.2	1.8
1.1	0.8	1.3



# Pytorch池化层

- `nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False)`
  - `kernel_size`: 卷积核大小, 可以是int, 或tuple;
  - `stride`: 步长;
  - `padding`
- `nn.AvgPool2d(kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True)`

# Pytorch池化层

```
1  import torch
2  import torch.nn as nn
3
4  x = torch.tensor([[3., 3., 2., 1., 0.], [0., 0., 1., 3., 1.]
5  x = x.view(1,5,5)
6  MP = nn.MaxPool2d(kernel_size=3, stride=1)
7  y = MP(x)
8  print(y)
9
10 MP2 = nn.AvgPool2d(3,1)
11 y = MP2(x)
12 print(y)
```

```
tensor([[[[3., 3., 3.],
          [3., 3., 3.],
          [3., 2., 3.]]]])
tensor([[[[1.6667, 1.6667, 1.6667],
          [1.0000, 1.2222, 1.7778],
          [1.1111, 0.7778, 1.3333]]]])
```





# Pytorch池化层

```
a = torch.randn(3,5,10)
MP3 = nn.MaxPool2d((5,1))
c = MP3(a)
print(c.shape)
```

```
x = torch.rand(1,3,7,7)
MP4 = nn.AvgPool2d(kernel_size=2, stride=2)
print(MP4.forward(x).shape)
```

```
torch.Size([3, 1, 10])
torch.Size([1, 3, 3, 3])
```



# 池化层总结

- 池化的超参数包括过滤器大小 $f$ 、步幅 $s$ 、池化方式（最大池化和平均池化），常用的参数值为 $f=2$ ， $s=2$ ，效果相当于高度和宽度缩减一半。
- 也可以增加表示padding的其他超参数，但很少这么用。
- 输入通道与输出通道个数相同，因为对每个通道都做了池化。
- 池化过程中没有需要学习的参数，只有这些超参数，可以是手动设置的，也可以是通过交叉验证设置的。



# 卷积神经网络

- 构建一个池化层，这里选最大池化，过滤器为 $2 \times 2$ ，步幅为2，padding为0。最终输出为 $6 \times 14 \times 14$ ，将该输出标记为**POOL1**。
- 在计算神经网络有多少层时，通常只统计具有权重和参数的层。这里，把CONV1和POOL1共同作为一个卷积，并标记为Layer1。

```
16 # Pool-1
17 Pool1 = nn.MaxPool2d((2,2),stride=2,padding=0)
18 output1 = Pool1(output1)
19 print(output1.shape)
```

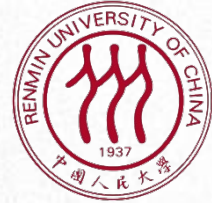
```
torch.Size([1, 6, 14, 14])
```





# 层的划分

- 在卷积神经网络文献中，卷积有两种分类，这与所谓层的划分有关。
- 一类卷积是一个卷积层和一个池化层一起作为一层，这就是神经网络的**Layer1**。
- 另一类卷积是把卷积层作为一层，而池化层单独作为一层。
- 在计算神经网络有多少层时，通常只统计具有权重和参数的层。因为池化层没有权重和参数，只有一些超参数。
- 在阅读网络文章或研究报告时，可能会看到卷积层和池化层各为一层的情况，这只是两种不同的标记术语。这里我们用**CONV1**和**POOL1**来标记，两者都是神经网络**Layer1**的一部分。



# 提纲

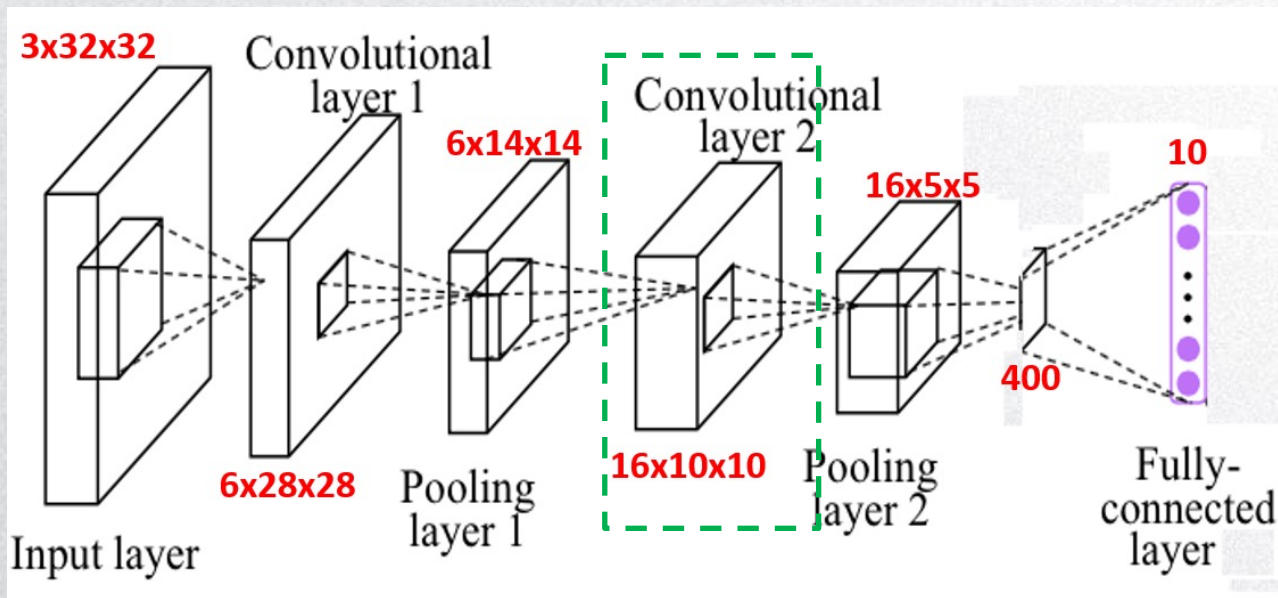


## 卷积神经网络

- □ 单个卷积层
- □ 池化层
- □ 卷积神经网络
-

# 卷积神经网络

- 再构建一个卷积层，过滤器大小为 $5 \times 5$ ，步幅为1，这次用16个过滤器，最后输出一个 $16 \times 10 \times 10$ 的矩阵，标记为CONV2。







## 练习：CONV2

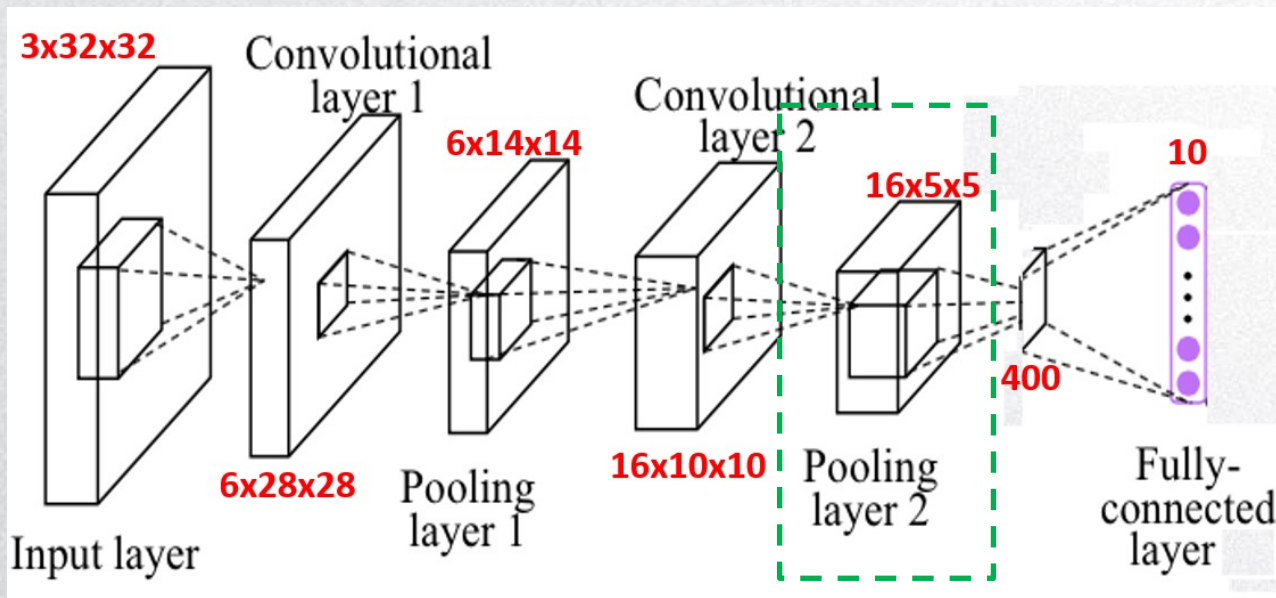
- 第二层卷积，16个5\*5的卷积核

```
20 # Conv-2
21 Conv2 = nn.Conv2d(in_channels=6,out_channels=16,kernel_size=5,stride=1,padding=0)
22 Nf2 = nn.ReLU()
23 output2 = Nf2(Conv2(output1))
24 print(output2.shape)
```

```
torch.Size([1, 16, 10, 10])
```

# 卷积神经网络

- 最大池化层,  $f=2$ ,  $s=2$ , 最后输出为  $16 \times 5 \times 5$ , 标记为POOL2
- 第二个卷积层和最大池化层组成**Layer2**





## 练习: POOL2

- POOL2

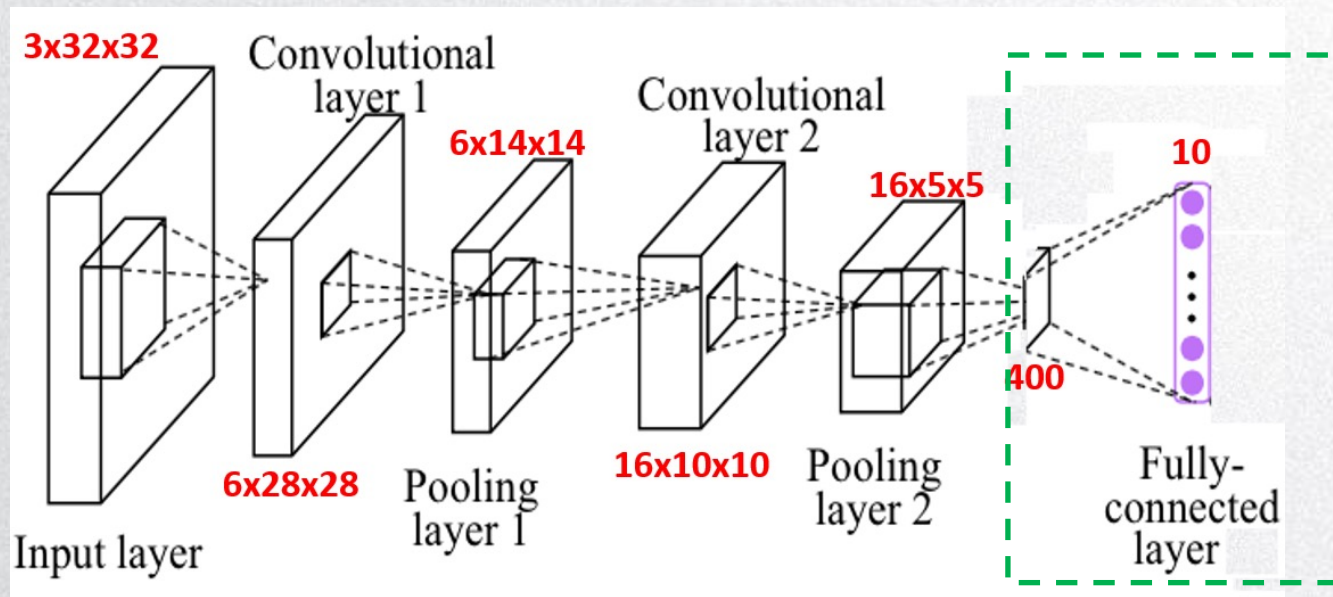
```
25 # Pool-2
26 Pool2 = nn.MaxPool2d((2,2),stride=2,padding=0)
27 output2 = Pool2(output2)
28 print(output2.shape)
```

```
torch.Size([1, 16, 5, 5])
```



# 卷积神经网络

- $16 \times 5 \times 5$  矩阵包含 400 个元素，平整化为一个大小为 400 的一维向量
- 400 个单元作为输入，10 个单元作为输出，构建全连接层 (FC)





# 分类概率基础

- **二分类**概率分布

- 样本空间：一个事件所有发生的可能情况
  - $S = \{\text{发生}, \text{不发生}\}$
  - $S = \{\text{第0类}, \text{第1类}\}$
- 样本点概率：  $P(s)$ , 满足  $P(s) \geq 0$  且  $P(s) \leq 1$
- 定义在一个样本空间 $S$ 的概率分布, 满足
  - $P(s) \geq 0, \forall s \in S$
  - $P(\text{第0类}) + P(\text{第1类}) = 1$
  - 可以用2个概率值（一个概率分布）表示对于一个事件发生的概率的估计
    - $\{0.1, 0.9\}$
    - $\{1, 0\}$  必定为第0类的概率
    - $\{0, 1\}$  必定为第1类的概率



# 分类概率基础

- **多分类**概率分布

- 样本空间：一个事件所有发生的可能情况
  - $S = \{\text{第0类}, \text{第1类}, \dots, \text{第}C-1\text{类}\}$
- 样本点概率：  $P(s)$ , 满足  $P(s) \geq 0$  且  $P(s) \leq 1$
- 定义在一个样本空间 $S$ 的概率分布, 满足
  - $P(s) \geq 0, \forall s \in S$
  - $P(\text{第0类}) + P(\text{第1类}) + \dots + P(\text{第}C-1\text{类}) = 1$
  - 可以用 $K$ 个概率数表示对于一个事件发生的概率的估计
    - $\{0.1, 0.3, 0.2, 0.4\}$
    - $\{1, 0, 0, 0\}$  必定为第0类的概率
    - $\{0, 1, 0, 0\}$  必定为第1类的概率

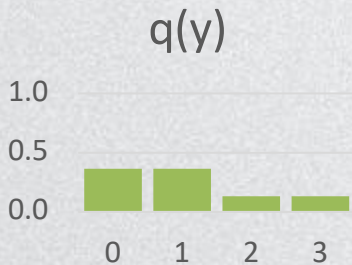


# 使用Softmax函数得到多分类概率估计

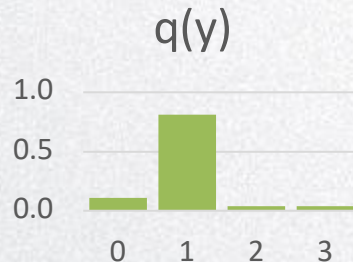
- 多分类： 假设总共有C类,  $y \in \{0, 1, 2, \dots, C - 1\}$ 
  - $q(y = i)$ : 当前样本属于第*i*类的概率的估计值
- 问题： 如何得到 $q(y = 1)$ ?
  - 模型输出一个C维向量 $\mathbf{z} = [z[0], z[1], \dots, z[C - 1]] \in R^C$
  - 利用softmax函数计算:

$$q(y = i) = \frac{e^{z[i]}}{\sum_j e^{z[j]}}$$

y	z	exp(z)	q(y)
0	1.000	2.718	0.366
1	1.000	2.718	0.366
2	0.000	1.000	0.134
3	0.000	1.000	0.134



y	z	exp(z)	q(y)
0	1.000	2.718	0.110
1	3.000	20.086	0.810
2	0.000	1.000	0.040
3	0.000	1.000	0.040





# 卷积神经网络

- $16 \times 5 \times 5$  矩阵包含400个元素，平整化为一个大小为400的一维向量
- 400个单元作为输入，10个单元作为输出，构建全连接层（FC）
- “全连接”：维度为 $400 \times 10$ ，这400个单元与这10个单元的每一项连接，有10个输出；对输出做SoftMax，得到属于各个数字的概率

`#Flatten`

```
output3 = output2.flatten(1)
print(output3.shape)
```

`#FC`

```
FC = nn.Linear(400, 10)
output3 = FC(output3)
print(output3.shape)
```

`#SoftMax`

```
output = torch.softmax(output3, dim=1)
print(output)
```

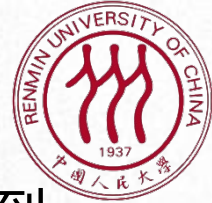
```
torch.Size([1, 400])
torch.Size([1, 10])
tensor([[0.0967, 0.1120, 0.0888, 0.0889, 0.0936, 0.1018, 0.1022, 0.1085, 0.1035,
         0.1040]], grad_fn=<SoftmaxBackward0>)
```



# Softmax函数

- PyTorch提供的Softmax函数:
  - `torch.softmax(input, dim)`
  - 输入:
    - `input`: tensor  $Z$
    - `dim`: 维数index, 沿着哪一维做softmax
  - 如果训练使用交叉熵损失函数, 则在forward里不需要反对输出施加Softmax
    - 会在下节课介绍如何训练CNN时讲解





# 卷积神经网络

- 随着神经网络深度的加深，高度和宽度通常都会减少（从 $32 \times 32$ 到 $28 \times 28$ ，到 $14 \times 14$ ，到 $10 \times 10$ ，再到 $5 \times 5$ ），而通道数量会增加（从3到6到16不断增加），然后使用一个全连接层。
- 在神经网络中，另一种常见模式就是一个或多个卷积后面跟随一个池化层，然后一个或多个卷积层后面再跟一个池化层，然后是几个全连接层，最后是一个softmax。

# 搭建卷积神经网络



```
# A class implementation
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=6, kernel_size=5, stride=1, padding=0)
        self.pool1 = nn.MaxPool2d((2,2), stride=2, padding=0)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1, padding=0)
        self.pool2 = nn.MaxPool2d((2,2), stride=2, padding=0)
        self.fc = nn.Linear(400, 10)
        self.nf = nn.ReLU()

    def forward(self, x):
        output1 = self.nf(self.conv1(x))
        print(output1.shape)
        output1 = self.pool1(output1)
        print(output1.shape)
        output2 = self.nf(self.conv2(output1))
        print(output2.shape)
        output2 = self.pool2(output2)
        print(output2.shape)
        # Flatten
        output3 = output2.flatten(1)
        print(output3.shape)
        # FC
        output3 = self.fc(output3)
        print(output3.shape)
        # SoftMax
        output = torch.softmax(output3, dim=1)
        return output

image = Image.open('5.jpg').convert('RGB')
input = transforms.ToTensor()(image).unsqueeze(0)
print(input.shape)

model = CNN()
output = model(input)
print(output)
```



# 搭建卷积神经网络

- 前向计算

```
image = Image.open('5.jpg').convert('RGB')
input = transforms.ToTensor()(image).unsqueeze(0)
print(input.shape)

model = CNN()
output = model(input)
print(output)

torch.Size([1, 3, 32, 32])
torch.Size([1, 6, 28, 28])
torch.Size([1, 6, 14, 14])
torch.Size([1, 16, 10, 10])
torch.Size([1, 16, 5, 5])
torch.Size([1, 400])
torch.Size([1, 10])
tensor([[0.1037, 0.1030, 0.0927, 0.1005, 0.0962, 0.1268, 0.0881, 0.0966, 0.0935,
        0.0988]], grad_fn=<SoftmaxBackward0>)
```





# 搭建卷积神经网络

- 前向计算, batch\_size > 1

```
image1 = Image.open('5.jpg').convert('RGB')
input1 = transforms.ToTensor()(image).unsqueeze(0)
image2 = Image.open('3.jpg').convert('RGB')
input2 = transforms.ToTensor()(image).unsqueeze(0)
input = torch.cat([input1, input2], 0)
print(input.shape)

model = CNN()
output = model(input)
print(output)

torch.Size([2, 3, 32, 32])
torch.Size([2, 6, 28, 28])
torch.Size([2, 6, 14, 14])
torch.Size([2, 16, 10, 10])
torch.Size([2, 16, 5, 5])
torch.Size([2, 400])
torch.Size([2, 10])
tensor([[0.1000, 0.1035, 0.0951, 0.0957, 0.1055, 0.0962, 0.0966, 0.0993, 0.1078,
         0.1002],
        [0.1000, 0.1035, 0.0951, 0.0957, 0.1055, 0.0962, 0.0966, 0.0993, 0.1078,
         0.1002]], grad_fn=<SoftmaxBackward0>)
```



## 练习

- $3 \times 32 \times 32$  的图片，第一层用6个大小为  $5 \times 5$  的滤波器，这一层有多少个参数？



# 为什么使用卷积


- 参数数量
- $3 \times 32 \times 32$  的图片，假设用6个大小为  $5 \times 5$  的滤波器，输出维度为  $6 \times 28 \times 28$ ，每个滤波器有  $3 \times 5 \times 5 + 1 = 76$  个参数，共  $76 \times 6 = 456$  个参数。
- 如果用全连接层，输入  $3 \times 32 \times 32 = 3072$  个节点，输出  $6 \times 28 \times 28 = 4704$  个节点，参数数量  $4074 \times 3072 \approx 1400$  万
- 如果图片大小为  $1000 \times 1000$ ，全连接权重矩阵会变得非常大。卷积参数数量保持不变。



# 为什么使用卷积


- 参数共享
- 特征检测如垂直边缘检测如果适用于图片的某个区域，那么它也可能适用于图片的其他区域。每个特征检测器都可以在输入图片的不同区域中使用同样的参数。

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0




\*

1	0	-1
1	0	-1
1	0	-1



=


0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0



# 为什么使用卷积


- 稀疏连接
- 例：0是通过 $3 \times 3$ 的卷积计算得到的，它只依赖于这个 $3 \times 3$ 的输入的单位格，右边这个输出单元（元素0）仅与36个输入特征中9个相连接；其它像素值都不会对输出产生任何影响。

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0




\*

1	0	-1
1	0	-1
1	0	-1



=

0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0





# 为什么使用卷积

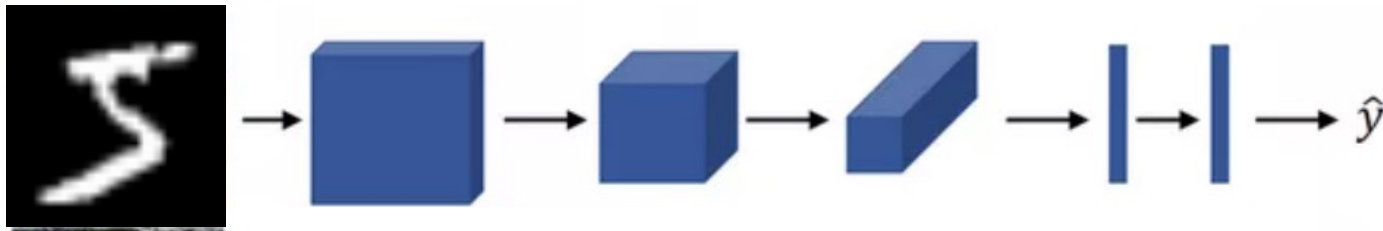
- 卷积神经网络通过参数共享和稀疏连接这两种机制减少参数，以便使用更少的训练数据来训练，防止过拟合。
- 平移不变性：向右移动两个像素，图片中的猫依然清晰可见；因为神经网络的卷积结构使得即使移动几个像素，这张图片依然具有非常相似的特征。



# 卷积神经网络训练

- 训练集:  $x$ 表示一张图片,  $y$ 是类别标注
- 损失函数:
  - [nn.CrossEntropyLoss](#)
  - 注意: PyTorch实现的多分类交叉熵不需要做softmax
- 优化器: 例如梯度下降法或Adam

Training set  $(x^{(1)}, y^{(1)}) \dots (x^{(m)}, y^{(m)})$ .



$$\text{Cost } J = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

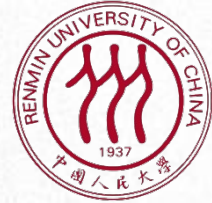


# PyTorch中的损失函数

- PyTorch提供的交叉熵损失函数：
  - [nn.CrossEntropyLoss](#)
  - 输入：
    - $N \times C$ 维矩阵 $\mathbf{Z}$ ，其中每一行为 $\mathbf{z} = [z[0], z[1], \dots, z[C - 1]] \in \mathbb{R}^C$ 
      - 每个数据点在 $C$ 个类别上的“确信度”
    - $N$ 维向量 $\mathbf{y}$ ，其中每个元素为 $y \in \{0, 1, 2, \dots, C - 1\}$ 
      - 标准答案

$$\text{loss}(\mathbf{Z}, \mathbf{y}) = - \sum_{i=0}^{N-1} \log\left(\frac{e^{\mathbf{Z}[i, \mathbf{y}[i]]}}{\sum_j e^{\mathbf{Z}[i, j]}}\right)$$

- 注意：该损失函数同时计算了softmax函数和交叉熵函数：



谢谢！



