深度之眼
deepshare.net

卷积神经网络简介

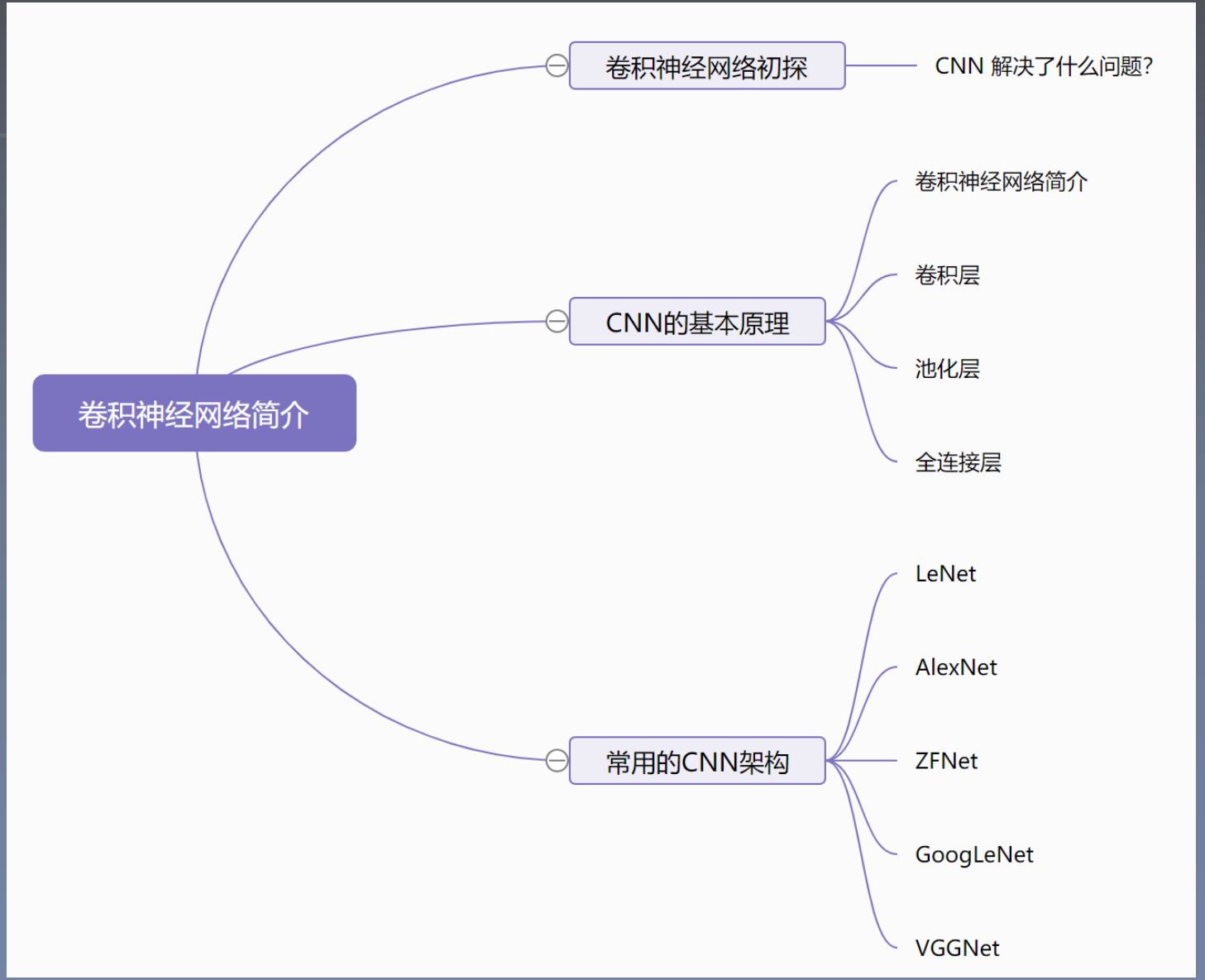
导师：GAUSS

深度之眼
deepshare.net

目录

知识树

Knowledge tree



深度之眼
deepshare.net

卷积神经网络初探

CNN 解决了什么问题？

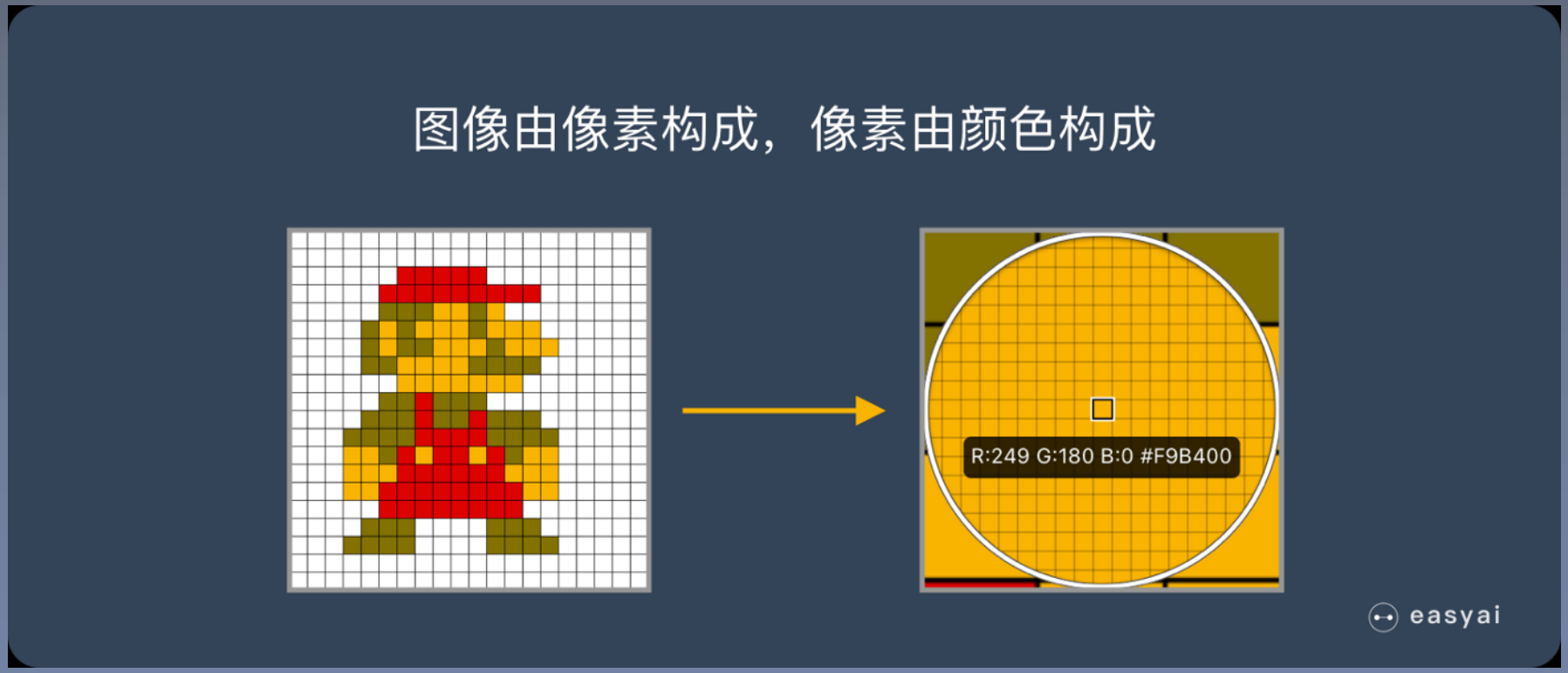
在 CNN 出现之前，图像对于人工智能来说是一个难题，有2个原因：

- 图像需要处理的**数据量太大**，导致成本很高，效率很低
- 图像在数字化的过程中**很难保留原有的特征**，导致图像处理的准确率不高

下面就详细说明一下这2个问题：

需要处理的数据量太大

图像是由像素构成的，每个像素又是由颜色构成的。



需要处理的数据量太大

现在随随便便一张图片都是 1000×1000 像素以上的， 每个像素都有RGB 3个参数来表示颜色信息。

假如我们处理一张 1000×1000 像素的图片，我们就需要**处理3百万个参数**！

1000×1000×3=3,000,000

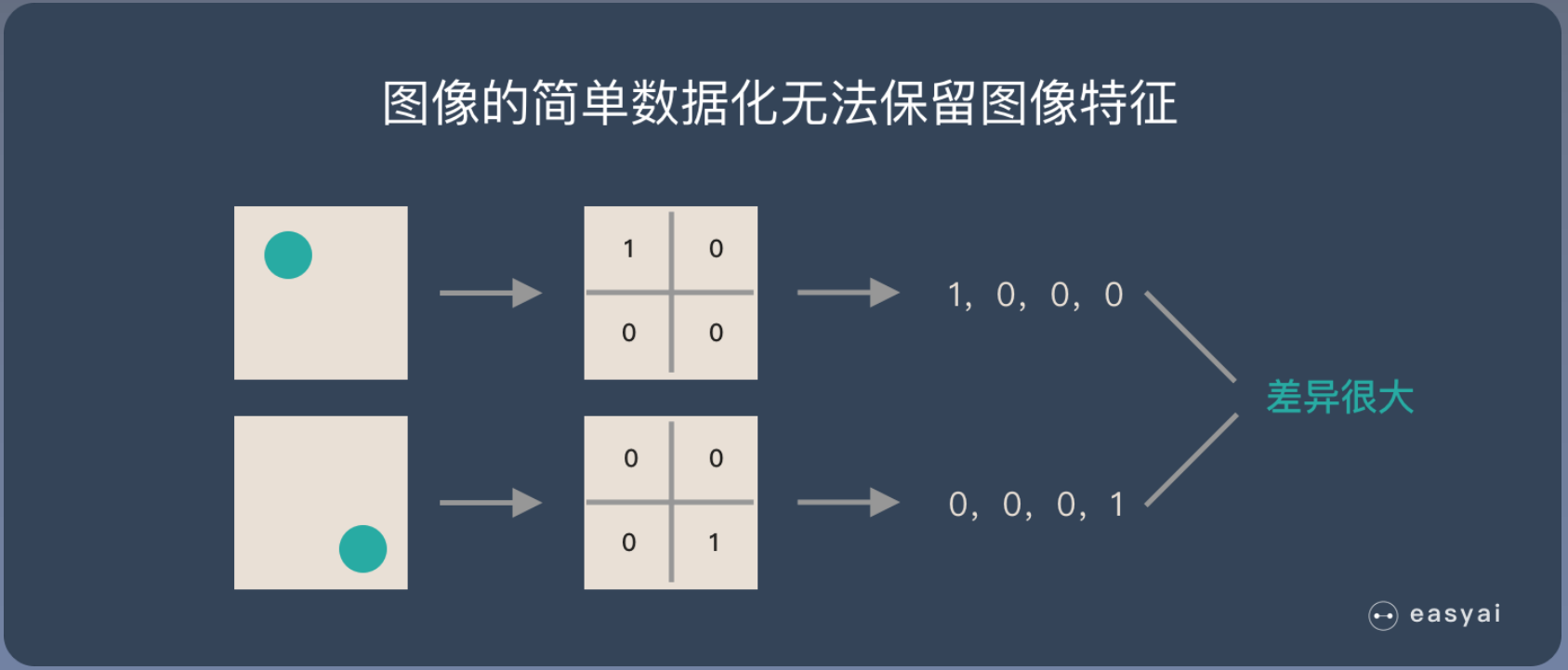
这么大量的数据处理起来是非常消耗资源的，而且这只是一张不算太大的图片！

卷积神经网络 — CNN 解决的第一个问题就是「将复杂问题简化」，把**大量参数降维成少量参数**，再做处理。

更重要的是：我们在大部分场景下，降维并不会影响结果。比如1000像素的图片缩小成200像素，并不影响肉眼认出来图片中是一只猫还是一只狗，机器也是如此。

保留图像特征

图片数字化的传统方式我们简化一下，就类似下图的过程：



保留图像特征

假如有圆形是1，没有圆形是0，那么圆形的位置不同就会产生完全不同的数据表达。但是从视觉的角度来看，**图像的内容（本质）并没有发生变化，只是位置发生了变化。**

所以当我们移动图像中的物体，用传统的方式得出来的参数会差异很大！这是不符合图像处理的要求的。

而 CNN 解决了这个问题，他用类似视觉的方式保留了图像的特征，当图像做翻转，旋转或者变换位置时，它也能有效的识别出来是类似的图像。

深度之眼
deepshare.net

CNN 的基本原理

卷积神经网络简介

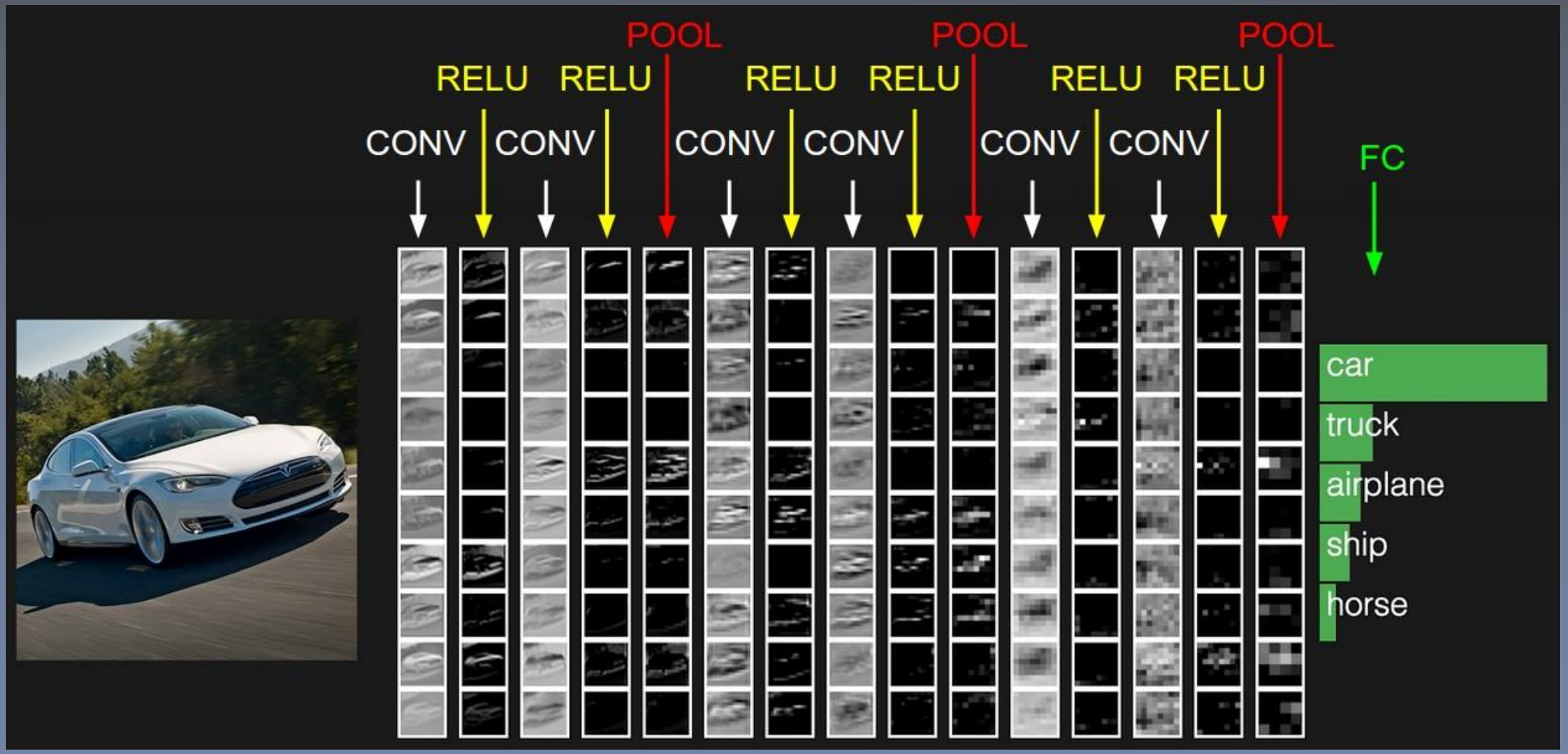
典型的卷积神经网络（CNN） 由3个部分构成：

- **卷积层**
- **池化层**
- **全连接层**

如果简单来描述的话：

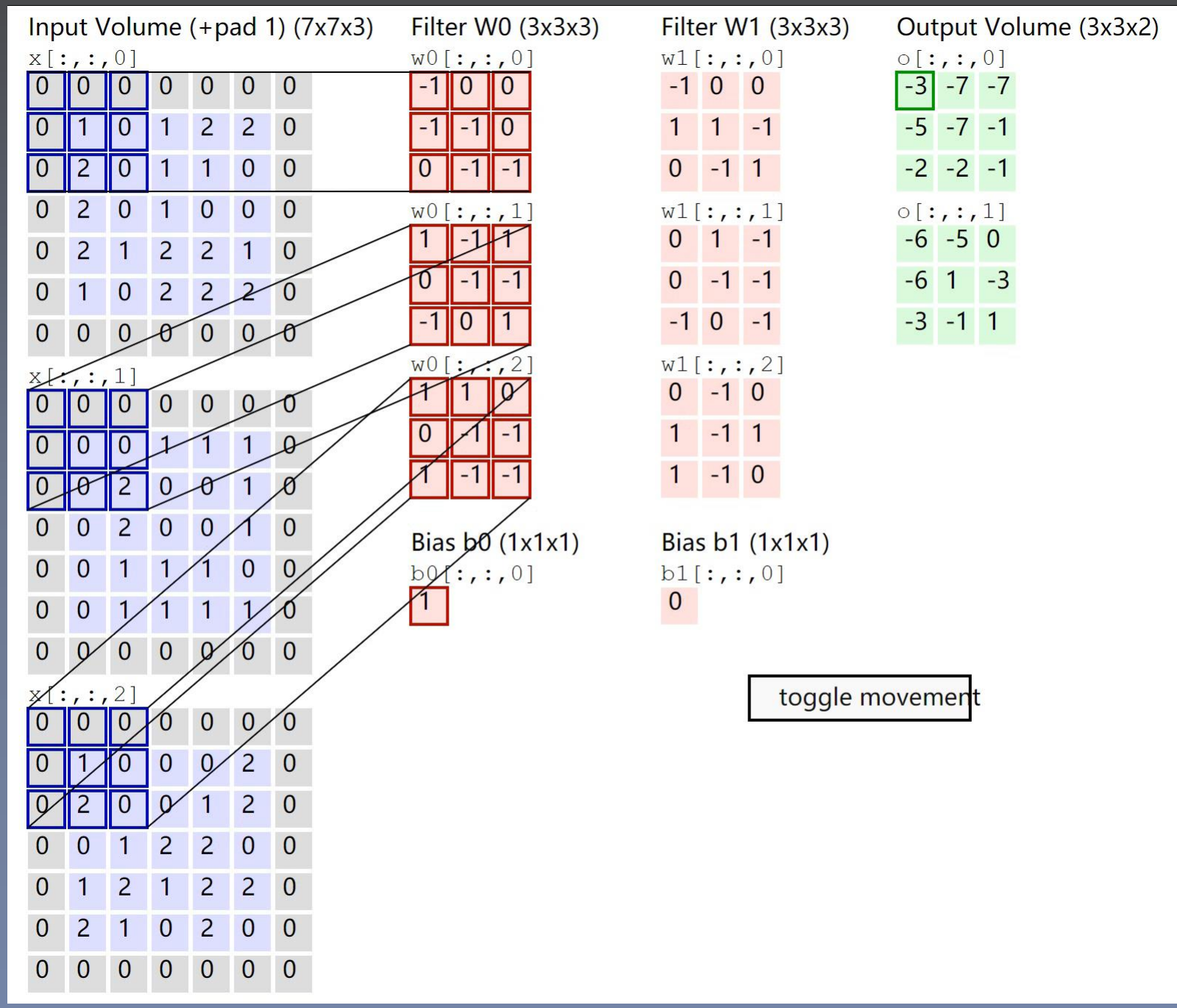
卷积层负责提取图像中的局部特征；池化层用来大幅降低参数量级(降维)；全连接层类似传统神经网络的部分，用来输出想要的结果。

卷积神经网络简介



简单了解卷积层

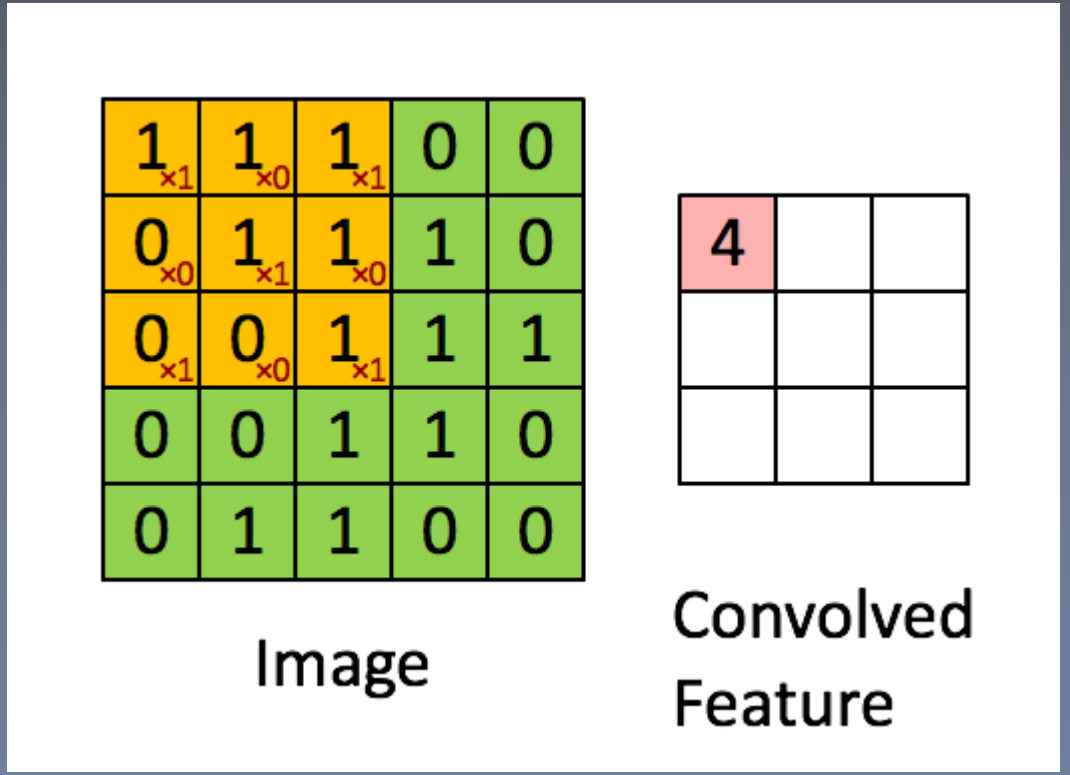
卷积层的前向传播以及计算方式：



卷积层

卷积层的运算过程如右图，用一个卷积核扫完整张图片：

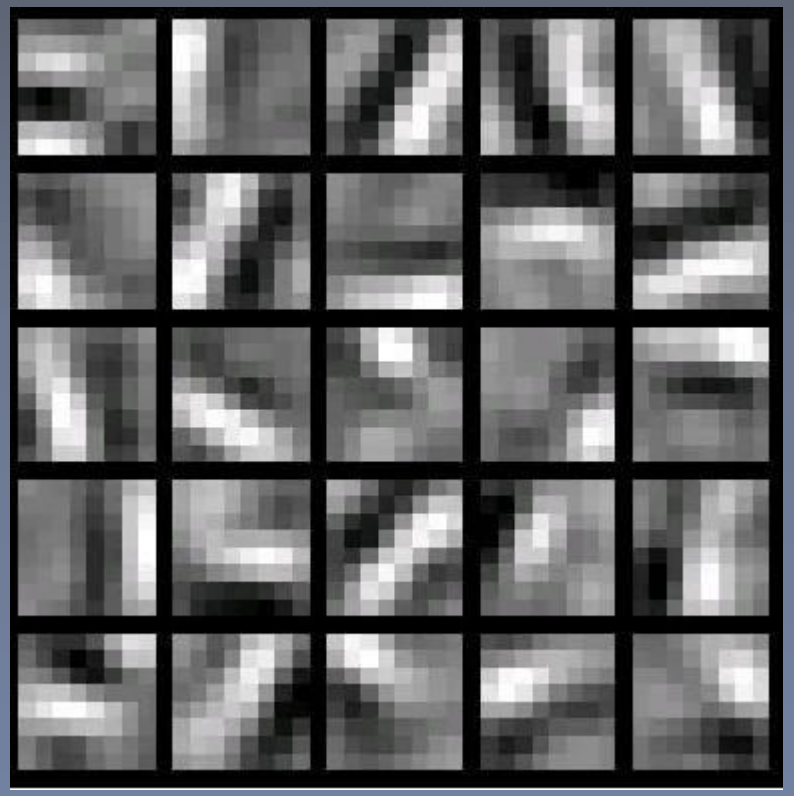
这个过程我们可以理解为我们使用一个过滤器（卷积核）来过滤图像的各个小区域，从而得到这些小区域的特征值。



卷积层

在具体应用中，往往有**多个卷积核**，可以认为，每个卷积核代表了一种图像模式，如果某个图像块与此卷积核卷积出的值大，则认为此图像块十分接近于此卷积核。如果我们**设计了6个卷积核**，可以理解为我们认为这个图像上有**6种底层纹理模式**，也就是我们**用6种基础模式就能描绘出一副图像。**

右图就是25种不同的卷积核的示例：



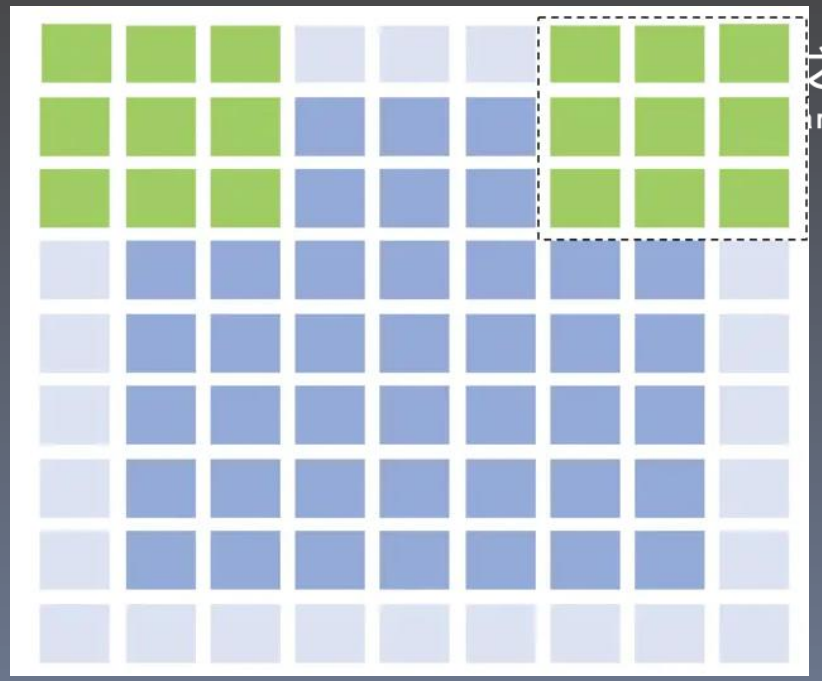
卷积层计算公式

输入张量格式：四个维度，依次为：样本数、图像高度、图像宽度、图像通道数

输出张量格式：与输出矩阵的维度顺序和含义相同，但是后三个维度（图像高度、图像宽度、图像通道数）的尺寸发生变化。

卷积核格式：同样是四个维度，但维度的含义与上面两者都不同，为：卷积核高度、卷积核宽度、输入通道数、输出通道数（卷积核个数）

卷积层计算公式



输入张量、**卷积核**、**输出张量**这三者之间的相互关系：

卷积核的输入通道数由**输入张量**的通道数所决定。

输出张量的通道数由**卷积核**的输出通道数所决定。

输出张量的高度和宽度这两个维度的尺寸由输入张量、卷积核、strides(每一步的步长)所共同决定。计算公式如下：

$$\begin{cases} height_{out} = (height_{in} - height_{kernel} + 2 * padding) \ / \ stride + 1 \\ width_{out} = (width_{in} - width_{kernel} + 2 * padding) \ / \ stride + 1 \end{cases}$$

注：以下计算演示均省略掉了偏置（Bias），严格来说其实每个卷积核都还有一个 Bias 参数。

tf自带的2D卷积层


```
tf.keras.layers.Conv2D(
    filters, kernel_size, strides=(1, 1), padding='valid', data_format=None,
    dilation_rate=(1, 1), activation=None, use_bias=True,
    kernel_initializer='glorot_uniform', bias_initializer='zeros',
    kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None,
    kernel_constraint=None, bias_constraint=None, **kwargs
)
```

输入：

如果data_format = ‘channels_first ‘，则4D张量的形状：(batch_size, channels, rows, cols) **或**

如果data_format =’ channels_last ‘，则4D张量的形状： (batch_size, rows, cols, channels)

输出：

如果data_format = ‘channels_first ‘，则4D张量的形状:(batch_size, filters, new_rows, new_cols) **或**

如果data_format =’ channels_last ‘，则形状的4D张量: (batch_size, new_rows, new_cols, filters) 。

其中rows和cols值可能由于填充而发生了变化。

tf自带的2D卷积层

- filters：输出空间的维数（即卷积中输出滤波器的数量）。
- kernel_size：过滤器的大小(一个整数或2个整数的元组/列表)。
- strides： 横向和纵向的步长。
- padding： valid表示不够卷积核大小的块,则丢弃; same表示不够卷积核大小的块就补0,所以输出和输入形状相同。
- data_format： 一个字符串,表示输入中维度的顺序.支持channels_last(默认)和channels_first。channels_last对应于具有形状(batch, height, width, channels)的输入,而channels_first对应于具有形状(batch, channels, height, width)的输入。

tf自带的2D卷积层

- dilation_rate :表示使用扩张卷积时的扩张率， 注意：strides 不等于1 和 dilation_rate 不等于1 这两种情况不能同时存在。（一般扩张卷积中使用）
- activation：激活函数。
- use_bias ： 是否使用偏置。
- kernel_initializer ： 卷积核的初始化。
- bias_initializer ： 偏差向量的初始化。

tf自带的2D卷积层

- kernel_regularizer ： 卷积核的正则项。
- bias_regularizer ： 偏差向量的正则项。
- activity_regularizer ： 输出层的正则化函数。
- kernel_constraint ： 约束函数应用于卷积核。
- bias_constraint ： 将约束函数应用于偏差向量。

动手实现卷积层Numpy版本

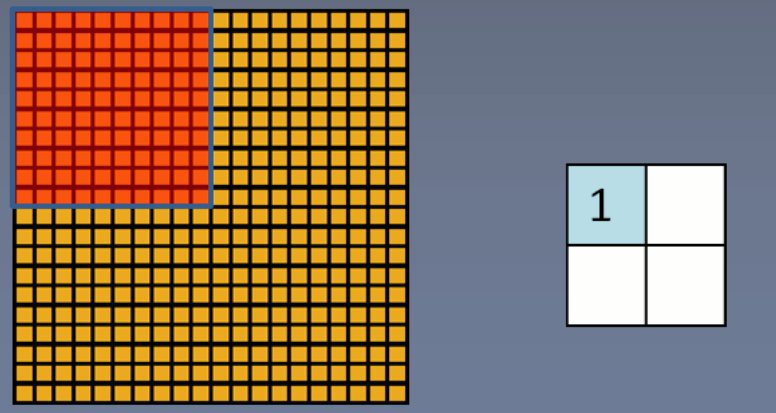

```python
def conv_numpy(x, w, b, pad,strides):
    out = None

    N, H, W, C = x.shape
    F,  HH, WW, C = w.shape

    X = np.pad(x, ((0, 0),(pad, pad), (pad, pad), (0, 0) ), 'constant')

    Hn = 1 + int((H + 2 * pad - HH) / strides[0])
    Wn = 1 + int((W + 2 * pad - WW) / strides[1])

    out = np.zeros((N, Hn, Wn, F))

    for n in range(N):
        for m in range(F):
            for i in range(Hn):
                for j in range(Wn):
                    data = X[n, i * strides[0]:i * strides[0] + HH, j * strides[1]:j * strides[1] + WW, :].reshape(1, -1)
                    filt = w[m].reshape(-1, 1)
                    out[n, i, j, m] = data.dot(filt) + b[m]

    return out
```

动手实现卷积层tf2.0版本


```
def corr2d(x, w, b, pad, stride):
    N, H, W, C = tf.shape(x)
    F, HH, WW, C = tf.shape(w)

    x = tf.pad(x, ((0, 0),(pad, pad), (pad, pad), (0, 0) ), 'constant')
    Hn = 1 + int((H + 2 * pad - HH) / stride[0])
    Wn = 1 + int((W + 2 * pad - WW) / stride[1])
    Y = tf.Variable(tf.zeros((N, Hn, Wn, F),dtype=tf.float32))

    for m in range(F):
        for i in range(Hn):
            for j in range(Wn):
                data = x[:, i * stride[0]:i * 1 + HH, j * stride[1]:j * 1 + WW, :]
                filt = w[m,:,:,:]
                Y[:, i, j, m].assign(tf.reduce_sum(tf.multiply(data,filt),axis=(1,2,3))+b[m])

    return Y
```

池化层

池化层简单说就是下采样，他可以大大降低数据的维度。其过程如下：



池化层

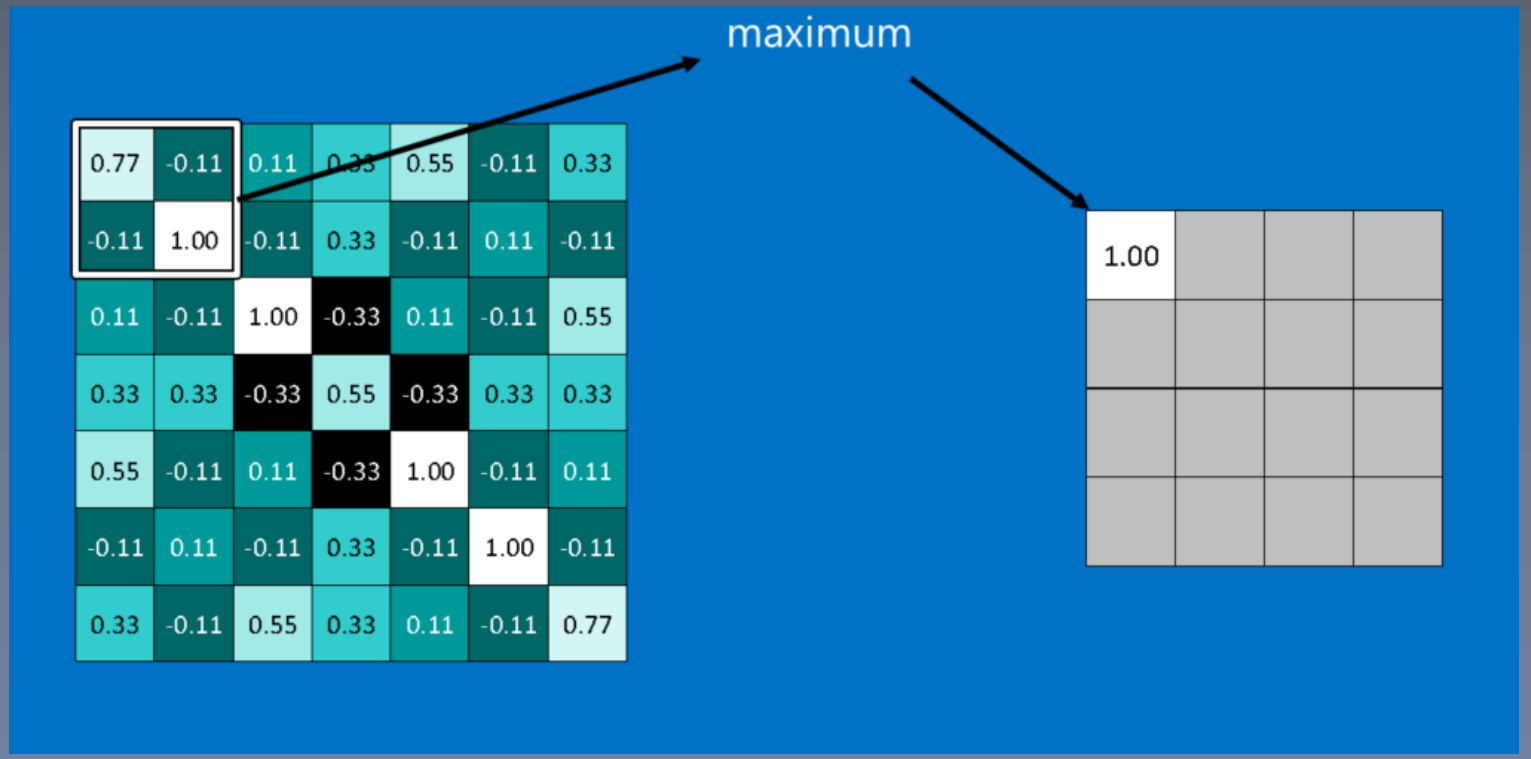
上图中，我们可以看到，原始图片是20×20的，我们对其进行下采样，采样窗口为10×10，最终将其下采样成为一个2×2大小的特征图。

之所以这么做的原因，是因为即使做完了卷积，图像仍然很大（因为卷积核比较小），所以为了降低数据维度，就进行下采样。

总结：池化层相比卷积层可以更有效的降低数据维度，这么做不但可以大大减少运算量，还可以有效的避免过拟合。

池化层

https://www.youtube.com/watch?v=FmpDIaiMIeA&feature=youtu.be&t=9m20s



池化层计算公式

通过pool_size为每个窗口的张量计算最大值，从而实现对输入张量进行下采样，窗口strides在每个维度上移动。使用“valid”填充选项时的结果输出的形状（行或列数）为:

$$height_{out} = (height_{in} - height_{pool\_size} + 1)/stride$$

$$width_{out} = (width_{in} - width_{pool\_size} + 1)/stride$$

tf自带的MaxPool2D


```
tf.keras.layers.MaxPool2D(
    pool_size=(2, 2), strides=None, padding='valid', data_format=None, **kwargs
)
```

输入形状： 如果data_format='channels_last'：具有形状的4D张量(batch_size, rows, cols, channels)。

如果data_format='channels_first'：具有形状的4D张量(batch_size, channels, rows, cols)。

输出形状： 如果data_format='channels_last'：具有形状的4D张量(batch_size, pooled_rows, pooled_cols, channels)。

如果data_format='channels_first'：具有形状的4D张量(batch_size, channels, pooled_rows, pooled_cols)。

tf自带的MaxPool2D

pool_size：2个整数的元组/列表：(pool_height,pool_width),用于指定池窗口的大小。可以是单个整数，这样所有空间维度具有相同值。

strides：2个整数的元组/列表,用于指定池操作的步幅.可以是单个整数，这样所有空间维度具有相同值。注意：当strides=None，大小和pool_size一样

padding：一个字符串,表示填充方法,“valid”或“same”,不区分大小写.

data_format：一个字符串,表示输入中维度的顺序.支持channels_last(默认)和channels_first；channels_last对应于具有形状(batch, height, width, channels)的输入,而channels_first对应于具有形状(batch, channels, height, width)的输入.

动手实现池化层numpy版本

numpy版本池化层(MaxPool2D)

```
def max_pool_forward_naive(x, pool_size=(2,2),strides=(1,1)):

    N, H, W, C = x.shape
    h_p, w_p = pool_size
    h_s, w_s = strides


    Hn = 1 + int((H - h_p) / h_s)
    Wn = 1 + int((W - w_p) / w_s)
    out = np.zeros((N, Hn, Wn, C))
    for i in range(Hn):
        for j in range(Wn):
            out[:, i, j, :] = np.max(x[:, i*h_s:i*h_s+h_p, j*w_s:j*w_s+w_p, :], axis=(1,2))
    return out
```

动手实现池化层tf版本

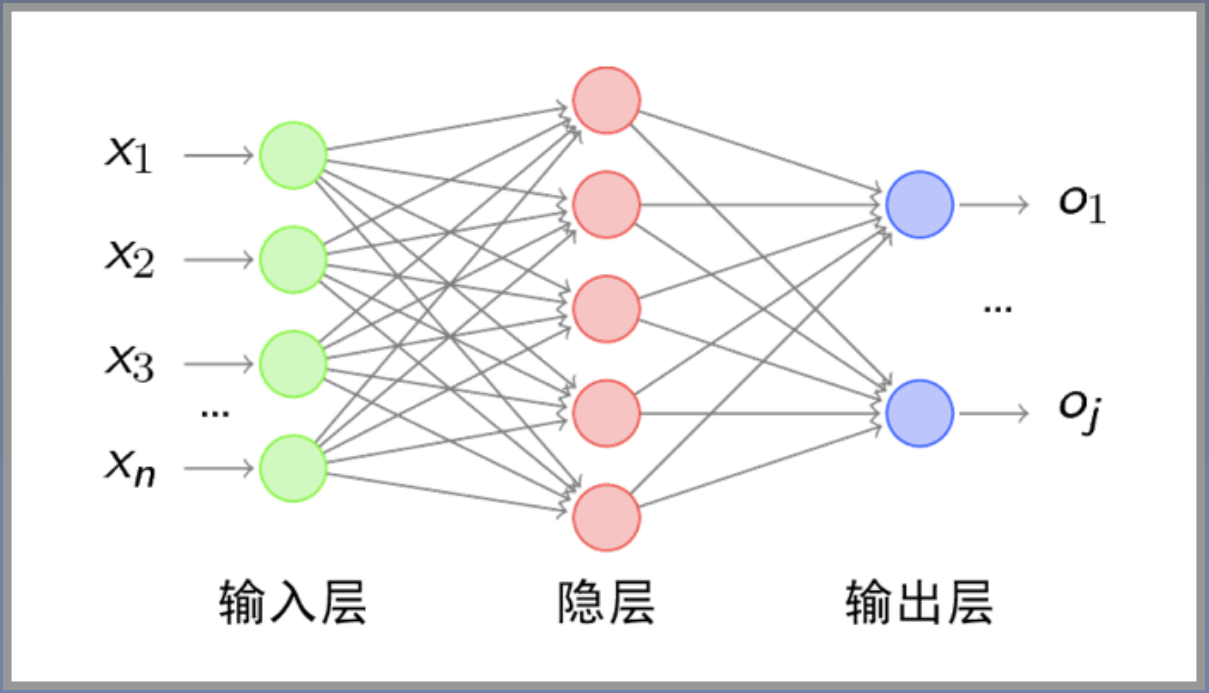
tf版本池化层(MaxPool2D)

```
def pool2d(X, pool_size=(2,2),strides=(1,1)):
    N, H, W, C = tf.shape(X)
    p_h, p_w = pool_size
    s_h, s_w = strides
    Y = tf.zeros((N, (H - p_h + 1)//s_h, (W - p_w +1)//s_w, C))
    Y = tf.Variable(Y)

    for i in tf.range(tf.shape(Y)[1]):
        for j in tf.range(tf.shape(Y)[2]):
            Y[:,i,j,:].assign(tf.math.reduce_max(X[:,i*s_h:i*s_h+p_h, j*s_w:j*s_w+p_w,:],axis=(1,2),keepdims=False))
    return Y
```

全连接层

这个部分就是最后一步了，经过卷积层和池化层处理过的数据输入到全连接层，得到最终想要的结果。

经过卷积层和池化层降维过的数据，全连接层才能”跑得动”，不然数据量太大，计算成本高，效率低下。



深度之眼
deepshare.net

常用的CNN架构

卷积神经网络架构

卷积神经网络领域中有几种具有名称的架构。最常见的是：

LeNet： Yann LeCun在1990年代开发了卷积网络的第一个成功应用程序。其中，最著名的是LeNet体系结构，该体系结构用于读取邮政编码，数字等。

AlexNet： 由Alex Krizhevsky，Ilya Sutskever和Geoff Hinton开发的AlexNet是在计算机视觉中普及卷积网络的第一部作品。AlexNet 于2012年参加了ImageNet ILSVRC挑战赛，并大大超越了第二名（前5名的错误率为16%，而第二名的错误率为26%）。网络具有与LeNet非常相似的体系结构，但是更深，更大，并且具有彼此堆叠的卷积层（以前通常只有一个CONV层总是紧随其后是POOL层）。

卷积神经网络架构

ZF Net：2013年ILSVRC冠军是Matthew Zeiler和Rob Fergus的卷积网络。它被称为ZFNet（Zeiler＆Fergus Net的缩写）。通过调整体系结构超参数，特别是通过扩展中间卷积层的大小并减小第一层的步幅和过滤器大小，对AlexNet进行了改进。

GoogLeNet：2014年ILSVRC获奖者是来自Google的Szegedy等人的卷积网络。它的主要贡献是开发了一个Inception模块，该模块大大减少了网络中的参数数量（4M，而AlexNet为60M）。此外，本文使用平均池而不是ConvNet顶部的全连接层，从而消除了似乎无关紧要的大量参数。GoogLeNet也有多个后续版本，最近的是Inception-v4。

卷积神经网络架构

VGGNet: 2014年ILSVRC的亚军是来自Karen Simonyan和Andrew Zisserman的网络，该网络被称为VGGNet。它的主要贡献在于表明网络深度是获得良好性能的关键因素。他们最终的最佳网络包含16个CONV / FC层，并且吸引人的是，它具有极其均匀的体系结构，从头到尾仅执行3x3卷积和2x2池化。他们的预训练模型可用于Caffe中的即插即用功能。VGGNet的缺点是评估成本更高，并且使用更多的内存和参数（140M）。这些参数中的大多数都位于第一个完全连接的层中，并且由于发现这些FC层可以在不降低性能的情况下被删除，从而大大减少了必要参数的数量。

卷积神经网络架构

ResNet：由Kaiming He等人开发的残差网络。是ILSVRC 2015的获胜者。它具有特殊的跳过连接和大量使用批标准化的功能。该体系结构还缺少网络末端的完全连接的层。读者还可以参考Kaiming的演示文稿（视频，幻灯片），以及一些最近的实验，这些实验在Torch中再现了这些网络。ResNets目前是最先进的卷积神经网络模型，并且是在实践中使用ConvNets的默认选择（截至2016年5月10日）。特别是，还可以看到更多的最新进展，这些变化将原始架构从何开明等。深度残留网络中的身份映射（2016年3月发布）。

深度之眼
deepshare.net

CNN 有哪些实际应用？

图像分类、检索

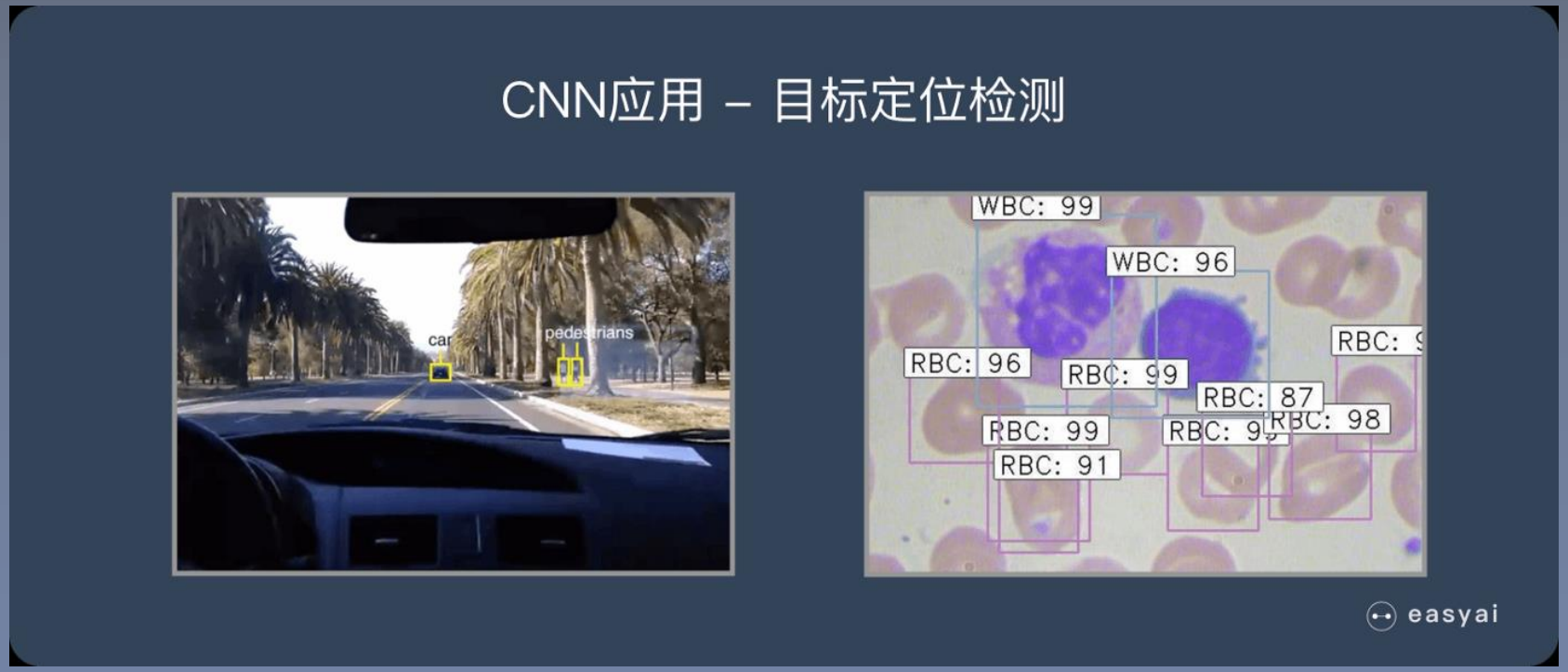
图像分类是比较基础的应用，他可以节省大量的人工成本，将图像进行有效的分类。对于一些特定领域的图片，分类的准确率可以达到 95%+，已经算是一个可用性很高的应用了。

典型场景：图像搜索...

目标定位检测

可以在图像中定位目标，并确定目标的位置及大小。

典型场景：自动驾驶、安防、医疗...



目标分割

简单理解就是一个像素级的分类。

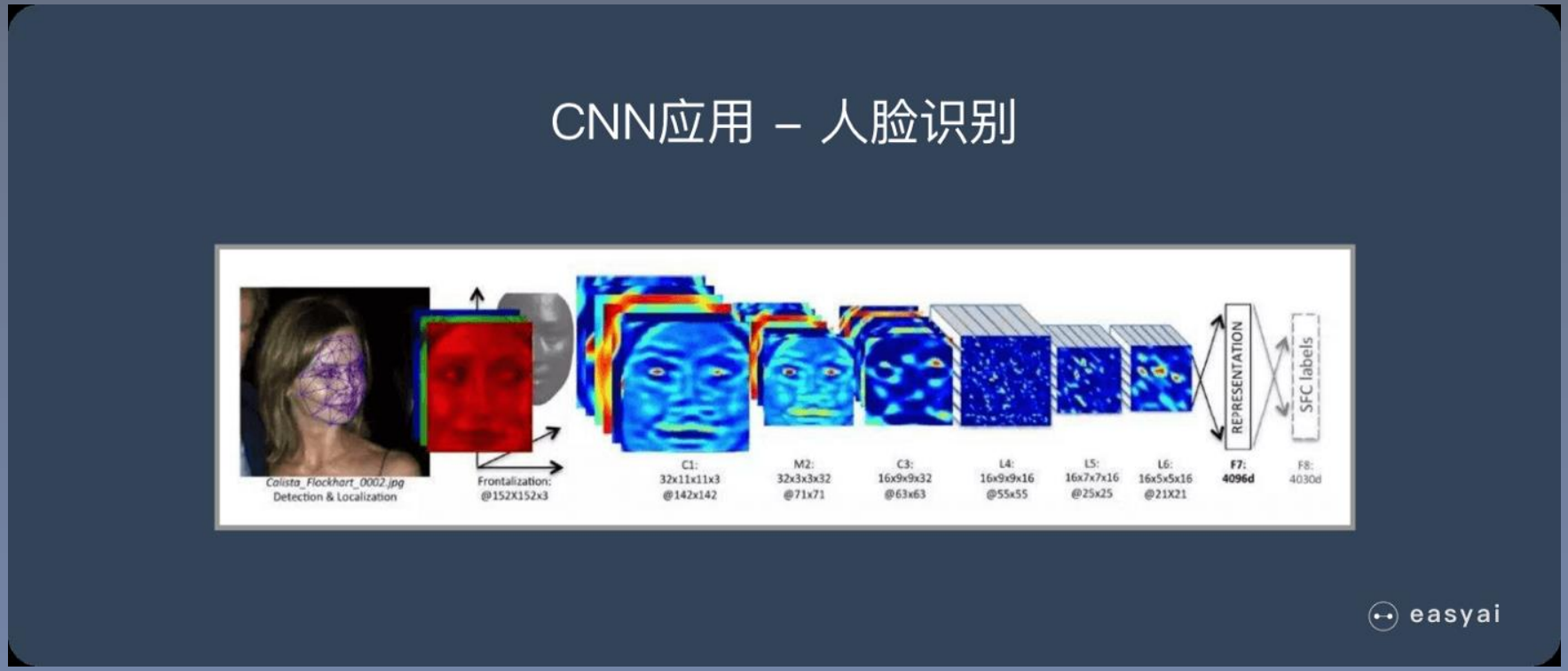
他可以对前景和背景进行像素级的区分、再高级一点还可以识别出目标并且对目标进行分类。

典型场景：美图秀秀、视频后期加工、图像生成...

人脸识别

人脸识别已经是一个非常普及的应用了，在很多领域都有广泛的应用。

典型场景：安防、金融、生活...



骨骼识别

骨骼识别是可以识别身体的关键骨骼，以及追踪骨骼的动作。

典型场景：安防、电影、图像视频生成、游戏...

深度之眼
deepshare.net

总结

总结

CNN 的基本原理：

- 卷积层 — 主要作用是保留图片的特征
- 池化层 — 主要作用是把数据降维，可以有效的避免过拟合
- 全连接层 — 根据不同任务输出我们想要的结果

CNN 的价值：

- 能够将大数据量的图片有效的降维成小数据量(并不影响结果)
- 能够保留图片的特征，类似人类的视觉原理

总结

CNN 的实际应用：

- 图片分类、检索
- 目标定位检测
- 目标分割
- 人脸识别
- 骨骼识别

本节小结

Summary

卷积神经网络简介	卷积神经网络初探	
	CNN 的基本原理	卷积层
		池化层
		全连接层
	常用的CNN架构	
	CNN 有哪些实际应用	

结语

我 说

看过千万代码，不如实践一把！

深度之眼
deepshare.net

联系我们：

电话：18001992849

邮箱：service@deepshare.net

Q Q： 2677693114

公众号

客服微信