## 安装

网址：<https://tensorflow.google.cn/>

TensorFlow™ 是一个开放源代码软件库，用于进行高性能数值计算。借助其灵活的架构，用户可以轻松地将计算工作部署到多种平台（CPU、GPU、TPU）和设备（桌面设备、服务器集群、移动设备、边缘设备等）。TensorFlow™ 最初是由 Google Brain 团队（隶属于 Google 的 AI 部门）中的研究人员和工程师开发的，可为机器学习和深度学习提供强力支持，并且其灵活的数值计算核心广泛应用于许多其他科学领域。

SciKit-learn与TensorFlow 的主要区别是：TensorFlow 更底层。而 SciKit-learn 提供了执行机器学习算法的模块化方案，很多算法模型直接就能用

其他框架：

Pytorch：

<https://pytorch.org/>

Torch 非常适用于卷积神经网络。它的开发者认为，Torch 的原生交互界面比其他框架用起来更自然、更得心应手。

其次，第三方的扩展工具包提供了丰富的递归神经网络（ RNN）模型。

因为这些强项，许多互联网巨头开发了定制版的 Torch，以助力他们的 AI 研究。这其中包括 Facebook、Twitter，和被谷歌招安前的 DeepMind。

Theano：

Theano 在深度学习框架中是祖师级的存在。它的开发始于 2007，早期开发者包括传奇人物 Yoshua Bengio 和 Ian Goodfellow。但随着 Tensorflow 在谷歌的支持下强势崛起，Theano 日渐式微，使用的人越来越少。**这过程中的标志性事件是：创始者之一的 Ian Goodfellow 放弃 Theano 转去谷歌开发 Tensorflow**

Caffe：

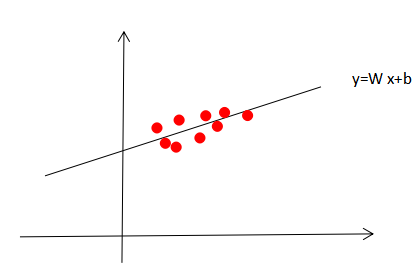
Caffe，全称Convolutional Architecture for Fast Feature Embedding。是一种常用的深度学习框架，主要应用在视频、图像处理方面的应用上。caffe是一个清晰，可读性高，快速的深度学习框架。作者是贾扬清，加州大学伯克利的ph.D，现就职于Facebook。caffe的官网是<http://caffe.berkeleyvision.org/>

<https://tensorflow.google.cn/install/>



## 实现线性回归

在数据上选择一条直线y=Wx+b，在这条直线上附件随机生成一些数据点如下图，让TensorFlow建立回归模型，去学习什么样的W和b能更好去拟合这些数据点。



随机生成1000个数据点，围绕在y=0.1x+0.3 周围，设置W=0.1，b=0.3，届时看构建的模型是否能学习到w和b的值。

import numpy as np

import tensorflow as tf

import matplotlib.pyplot as plt

num\_points=1000

vectors\_set=[]

for i in range(num\_points):

#横坐标，进行随机高斯处理化，以0为均值，以0.55为标准差

x1=np.random.normal(0.0,0.55)

#纵坐标，数据点在y1=x1\*0.1+0.3上小范围浮动

y1=x1\*0.1+0.3+np.random.normal(0.0,0.03)

vectors\_set.append([x1,y1])

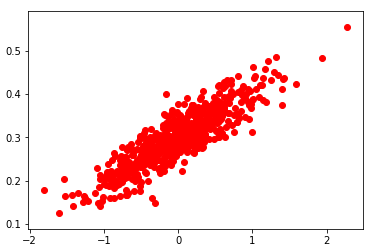
x\_data=[v[0] for v in vectors\_set]

y\_data=[v[1] for v in vectors\_set]

plt.scatter(x\_data,y\_data,c='r')

plt.show()

构造数据如下图



2）构造线性回归模型，学习上面数据图是符合一个怎么样的W和b

# 生成1维的W矩阵，取值是[-1,1]之间的随机数

W = tf.Variable(tf.random\_uniform([1], -1.0, 1.0), name='W')

# 生成1维的b矩阵，初始值是0

b = tf.Variable(tf.zeros([1]), name='b')

# 经过计算得出预估值y

y = W \* x\_data + b

# 以预估值y和实际值y\_data之间的均方误差作为损失

loss = tf.reduce\_mean(tf.square(y - y\_data), name='loss')

# 采用梯度下降法来优化参数 学习率为0.5

optimizer = tf.train.GradientDescentOptimizer(0.5)

# 训练的过程就是最小化这个误差值

train = optimizer.minimize(loss, name='train')

sess = tf.Session()

init = tf.global\_variables\_initializer()

sess.run(init)

# 初始化的W和b是多少

print ("W =", sess.run(W), "b =", sess.run(b), "loss =", sess.run(loss))

for step in range(20): # 执行20次训练

sess.run(train)

# 输出训练好的W和b

print ("W =", sess.run(W), "b =", sess.run(b), "loss =", sess.run(loss))

打印每一次结果，如下图，随着迭代进行，训练的W、b越来越接近0.1、0.3，说明构建的回归模型确实学习到了之间建立的数据的规则。loss一开始很大，后来慢慢变小，说明模型表达效果随着迭代越来越好。

W = [-0.9676645] b = [0.] loss = 0.45196822

W = [-0.6281831] b = [0.29385352] loss = 0.17074569

W = [-0.39535886] b = [0.29584622] loss = 0.07962803

W = [-0.23685378] b = [0.2972129] loss = 0.03739688

W = [-0.12894464] b = [0.2981433] loss = 0.017823622

W = [-0.05548081] b = [0.29877672] loss = 0.008751821

### TensorFlow的一些基本概念

#### 1.placehoder（占位符）

tf.placeholder(dtype, shape=None, name=None)

Args:

    dtype: The type of elements in the tensor to be fed.

    shape: The shape of the tensor to be fed (optional). If the shape is not specified, you can feed a tensor of any shape.

    name: A name for the operation (optional).

dytpe:占位符的数据类型

shape:占位符的纬度，例如[2,2]代表2x2的二维矩阵，None可以代表任意维度，例如[None,2]则代表任意行数，2列的二维矩阵

name:占位符的名字

变量在定义时要初始化，但可能有些变量我们一开始定义的时候并不一定知道该变量的值，只有当真正开始运行程序的时候才由外部输入，比如我们需要训练的数据，所以就用占位符来占个位置，告诉TensorFlow，等到真正运行的时候再通过输入数据赋值。

例如

x = tf.placeholder(tf.float32, [2, 2])

就是生成了一个2x2的二维矩阵，矩阵中每个元素的类型都是tf.float32（也就是浮点型）。

有时候定义需要训练的参数时候，会定义一个[input\_size,output\_size]大小的矩阵，其中input\_size数输入数据的维度，output\_size是输出数据的维度

#### 2.Variable（变量）

变量在声明的时候要有一个初始值

x = tf.Variable(tf.zeros([2,2])) # 声明一个2x2的矩阵，并将矩阵中的所有元素的值赋为0，默认每个元素都是tf.float32类型的数据

y = tf.Variable(1.0, tf.float32) # 声明一个tf.float32的变量，并将初始值设为1.0

我们一般还需要运行下global\_variables\_initializer真正在TensorFlow的Session中初始化所有变量，后面的样例中也会有体现。

#### 3.Constant（常量）

x = tf.constant(3.0, tf.float32) # 定义一个值为3.0的浮点型常量

#### 4.Session（会话）

TensorFlow所有的操作都必须在Session中运行，才能真正起作用，可以将Session当作TensorFlow运行的环境，Session运行完需要close～

#用close()关闭

sess = tf.Session()

sess.run(...)

sess.close()

#使用with..as..语句关闭

with tf.Session() as sess:

     sess.run(...)

## 实现CNN模型

**step 0 导入TensorFlow**

1 import tensorflow as tf

2 from tensorflow.examples.tutorials.mnist import input\_data

**step 1 加载数据集mnist**

声明两个placeholder，用于存储神经网络的输入，输入包括image和label。这里加载的image是(784,)的shape。

1 mnist = input\_data.read\_data\_sets('MNIST\_data/', one\_hot=True)

2 x = tf.placeholder(tf.float32,[None, 784])

3 y\_ = tf.placeholder(tf.float32, [None, 10])

**step 2 定义weights和bias**

为了使代码整洁，这里把weight和bias的初始化封装成函数。

[复制代码](javascript:void(0);)

1 #----Weight Initialization---#

2 #One should generally initialize weights with a small amount of noise for symmetry breaking, and to prevent 0 gradients

3 def weight\_variable(shape):

4 initial = tf.truncated\_normal(shape, stddev=0.1)

5 return tf.Variable(initial)

6 def bias\_variable(shape):

7 initial = tf.constant(0.1, shape=shape)

8 return tf.Variable(initial)

[复制代码](javascript:void(0);)

**step 3 定义卷积层和maxpooling**

同样，为了代码的整洁，将卷积层和maxpooling封装起来。padding=‘SAME’表示使用padding，不改变图片的大小。

[复制代码](javascript:void(0);)

1 #Convolution and Pooling

2 #Our convolutions uses a stride of one and are zero padded so that the output is the same size as the input.

3 #Our pooling is plain old max pooling over 2x2 blocks

4 def conv2d(x, W):

5 return tf.nn.conv2d(x, W, strides=[1,1,1,1], padding='SAME')

6 def max\_pool\_2x2(x):

7 return tf.nn.max\_pool(x, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')



tf.nn.conv2d(input, filter, strides, padding, use\_cudnn\_on\_gpu=None, name=None)

除去name参数用以指定该操作的name，与方法有关的一共五个参数：

第一个参数input：指需要做卷积的输入图像，它要求是一个Tensor，具有[batch, in\_height, in\_width, in\_channels]这样的shape，具体含义是[训练时一个batch的图片数量, 图片高度, 图片宽度, 图像通道数]，注意这是一个4维的Tensor，要求类型为float32和float64其中之一

第二个参数filter：相当于CNN中的卷积核，它要求是一个Tensor，具有[filter\_height, filter\_width, in\_channels, out\_channels]这样的shape，具体含义是[卷积核的高度，卷积核的宽度，图像通道数，卷积核个数]，要求类型与参数input相同，有一个地方需要注意，第三维in\_channels，就是参数input的第四维

第三个参数strides：卷积时在图像每一维的步长，这是一个一维的向量，长度4

第四个参数padding：string类型的量，只能是"SAME","VALID"其中之一，这个值决定了不同的卷积方式（后面会介绍）

第五个参数：use\_cudnn\_on\_gpu:bool类型，是否使用cudnn加速，默认为true

结果返回一个Tensor，这个输出，就是我们常说的feature map，shape仍然是[batch, height, width, channels]这种形式。

**step 4 reshape image数据**

为了神经网络的layer可以使用image数据，我们要将其转化成4d的tensor: (Number, width, height, channels)

1 #To apply the layer, we first reshape x to a 4d tensor, with the second and third dimensions corresponding to image width and height,

2 #and the final dimension corresponding to the number of color channels.

3 x\_image = tf.reshape(x, [-1,28,28,1])

**step 5 搭建第一个卷积层**

使用32个5x5的filter，然后通过maxpooling。

[复制代码](javascript:void(0);)

1 #----first convolution layer----#

2 #he convolution will compute 32 features for each 5x5 patch. Its weight tensor will have a shape of [5, 5, 1, 32].

3 #The first two dimensions are the patch size,

4 #the next is the number of input channels, and the last is the number of output channels.

5 W\_conv1 = weight\_variable([5,5,1,32])

6

7 #We will also have a bias vector with a component for each output channel.

8 b\_conv1 = bias\_variable([32])

9

10 #We then convolve x\_image with the weight tensor, add the bias, apply the ReLU function, and finally max pool.

11 #The max\_pool\_2x2 method will reduce the image size to 14x14.

12 h\_conv1 = tf.nn.relu(conv2d(x\_image, W\_conv1) + b\_conv1)

13 h\_pool1 = max\_pool\_2x2(h\_conv1)

[复制代码](javascript:void(0);)

**step 6 第二层卷积**

使用64个5x5的filter。

[复制代码](javascript:void(0);)

1 #----second convolution layer----#

2 #The second layer will have 64 features for each 5x5 patch and input size 32.

3 W\_conv2 = weight\_variable([5,5,32,64])

4 b\_conv2 = bias\_variable([64])

5

6 h\_conv2 = tf.nn.relu(conv2d(h\_pool1, W\_conv2) + b\_conv2)

7 h\_pool2 = max\_pool\_2x2(h\_conv2)

[复制代码](javascript:void(0);)

**step 7 构建全链接层**

需要将上一层的输出，展开成1d的神经层。

[复制代码](javascript:void(0);)

1 #----fully connected layer----#

2 #Now that the image size has been reduced to 7x7, we add a fully-connected layer with 1024 neurons to allow processing on the entire image

3 W\_fc1 = weight\_variable([7\*7\*64, 1024])

4 b\_fc1 = bias\_variable([1024])

5

6 h\_pool2\_flat = tf.reshape(h\_pool2, [-1, 7\*7\*64])

7 h\_fc1 = tf.nn.relu(tf.matmul(h\_pool2\_flat,W\_fc1) + b\_fc1)

[复制代码](javascript:void(0);)

**step 8 添加Dropout**

加入Dropout层，可以防止过拟合问题。注意，这里使用了另外一个placeholder，可以控制在训练和预测时是否使用Dropout。

[复制代码](javascript:void(0);)

1 #-----dropout------#

2 #To reduce overfitting, we will apply dropout before the readout layer.

3 #We create a placeholder for the probability that a neuron's output is kept during dropout.

4 #This allows us to turn dropout on during training, and turn it off during testing.

5 keep\_prob = tf.placeholder(tf.float32)

6 h\_fc1\_dropout = tf.nn.dropout(h\_fc1, keep\_prob)

[复制代码](javascript:void(0);)

**step 9 输入层**

没有什么特别的，就是输出一个线性结果。

1 #----read out layer----#

2 W\_fc2 = weight\_variable([1024,10])

3 b\_fc2 = bias\_variable([10])

4 y\_conv = tf.matmul(h\_fc1\_dropout, W\_fc2) + b\_fc2

**step 10 训练和评估**

首先，需要指定一个cost function --cross\_entropy，在输出层使用softmax。然后指定optimizer--adam。需要特别指出的是，一定要记得

tf.global\_variables\_initializer().run()初始化变量

[复制代码](javascript:void(0);)

1 #------train and evaluate----#

2 cross\_entropy = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(labels=y\_, logits=y\_conv))

3 train\_step = tf.train.AdamOptimizer(1e-4).minimize(cross\_entropy)

4 accuracy = tf.reduce\_mean(tf.cast(tf.equal(tf.argmax(y\_, 1), tf.argmax(y\_conv, 1)), tf.float32))

5 with tf.Session() as sess:

6 tf.global\_variables\_initializer().run()

7 for i in range(3000):

8 batch = mnist.train.next\_batch(50)

9 if i % 100 == 0:

10 train\_accuracy = accuracy.eval(feed\_dict = {x: batch[0],

11 y\_: batch[1],

12 keep\_prob: 1.})

13 print('setp {},the train accuracy: {}'.format(i, train\_accuracy))

14 train\_step.run(feed\_dict = {x: batch[0], y\_: batch[1], keep\_prob: 0.5})

15 test\_accuracy = accuracy.eval(feed\_dict = {x: mnist.test.images, y\_: mnist.test.labels, keep\_prob: 1.})

16 print('the test accuracy :{}'.format(test\_accuracy))

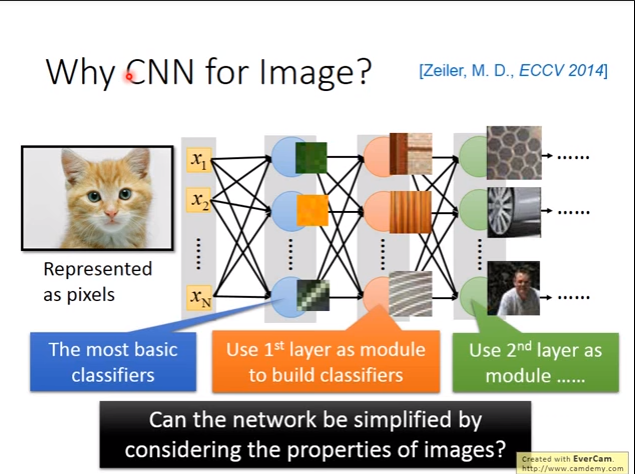
17 saver = tf.train.Saver()

18 path = saver.save(sess, './my\_net/mnist\_deep.ckpt')

19 print('save path: {}'.format(path))

## 卷积网络

**Why CNN for Image**



图片是由像素点组成的，可以这样来解释深度神经网络对图片的处理。

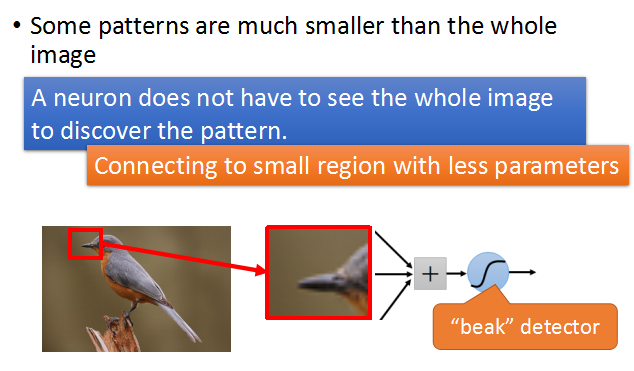
第一层的layer是最基本的分类器，区分一些基本的特征，比如颜色、是否有斜线。

第二层的layer会检测更加复杂的东西，比如一些简单的组合线条；

后面的layer也会越来越复杂……

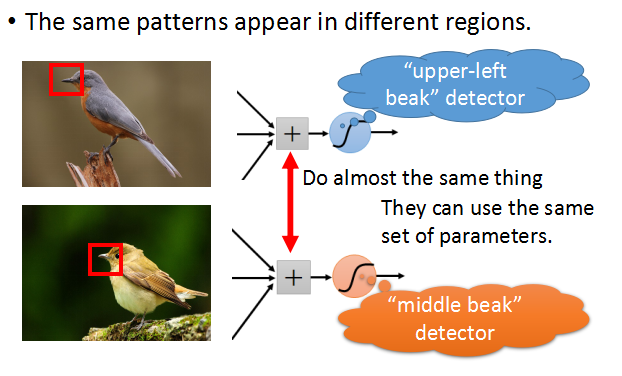
我们可以通过思考图像的特征来简化网络。

1.图片中一些特征通常比整个图片要小，比如要检测图像中是否有鸟嘴。



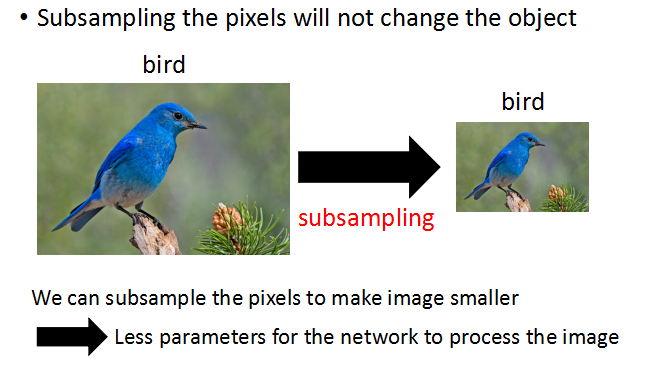
 我们的neuron不需要看整个图像来发现某些特征，所以我们只需要把鸟嘴那一小部分的图片，用很少的参数跟neuron关联起来。

 2.同样的特征可能出现在图片的不同位置。



我们不会为每个不同位置的特征单独训练一个neuron，因为它做的都是同样的事情，就是检测是否出现鸟嘴，只是出现的位置不一样罢了。

3.subsampling 可以使图片缩小，但不影响图片的表达。

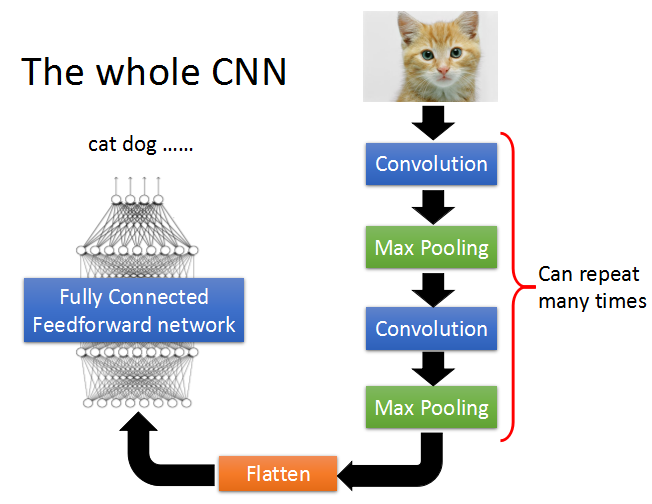


每隔一行、一列删除一行pixel，就是subsampling，我们同样可以看到图片表达的信息，就是一只鸟。

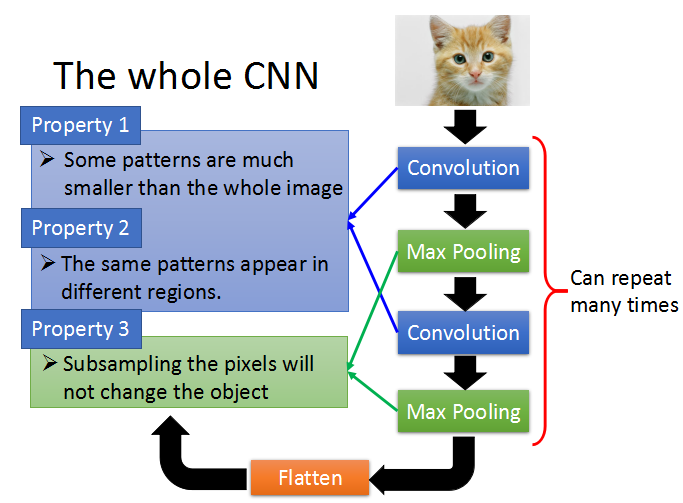
由于图片缩小了，这样又可以减少参数了。

**The whole CNN**

来看看整个CNN的架构



从图片作为输入开始，经过多层的Convolution层+MaxPooling的组合，然后是Flatten层，最后经过一个Fully Connected network。

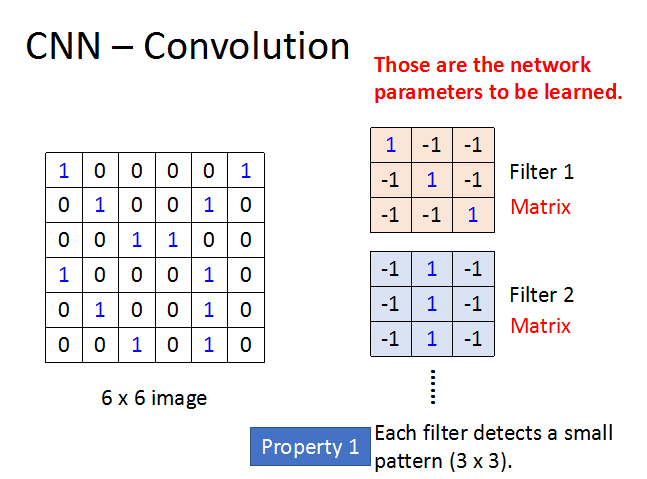


 其中，上面讨论的关于图片的三个特点，在CNN的不同层中有相应处理。

Property1、2是小的特征和 特征的不同位置，通过Convolution层进行处理；

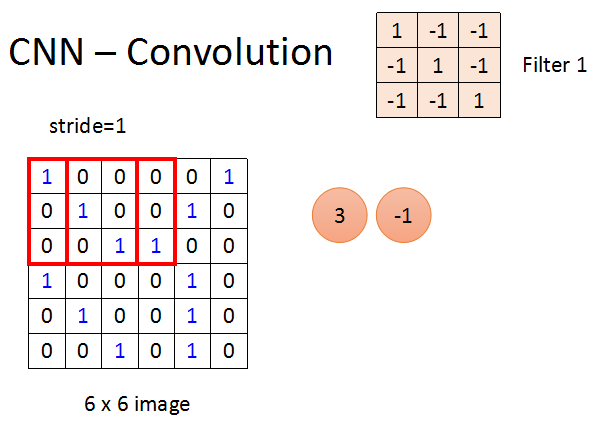
Property3 Subsampling通过MaxPooling处理。

**CNN – Convolution**

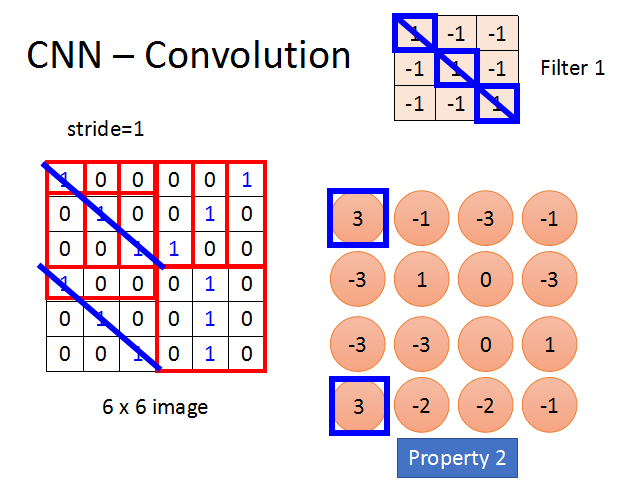


这里需要提到的是Filter。Filter其实就是一个矩阵，它们是神经网络需要学习的参数。

每个Filter在图片中进行扫描，检测3\*3的特征。

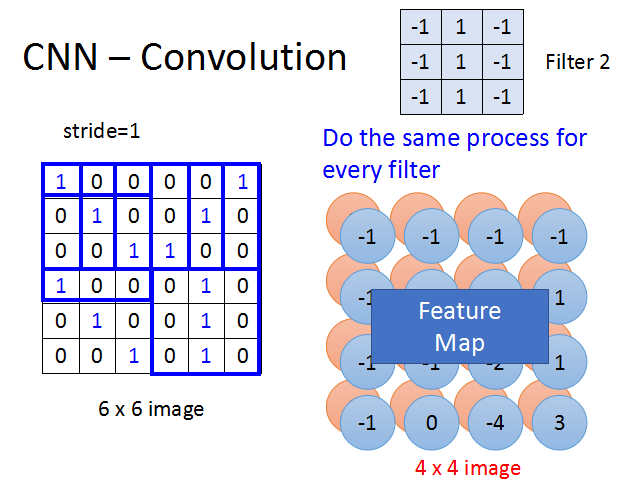


Filter从图片的左上角开始，以stride为步长进行图片扫描，图片中每3\*3的子图会和Filter作内积，然后得到一个输出值。



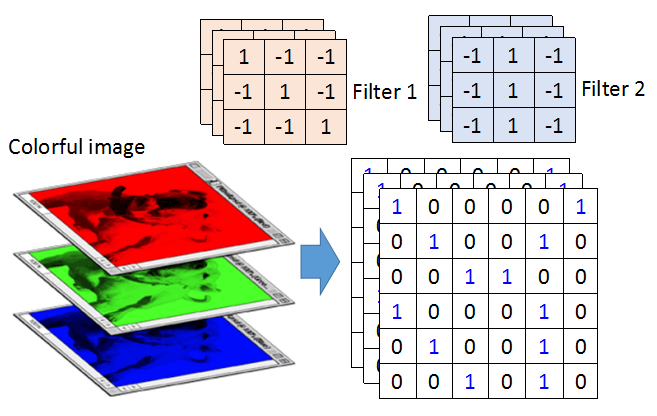
上面就是FIlter1扫描整个6×6图片后得到的4×4的结果矩阵。

可以进一步理解的是，该Filter对角线全为1，表示检测图像是否出现类似的斜线，出现斜线的地方在结果矩阵中的值为最大。



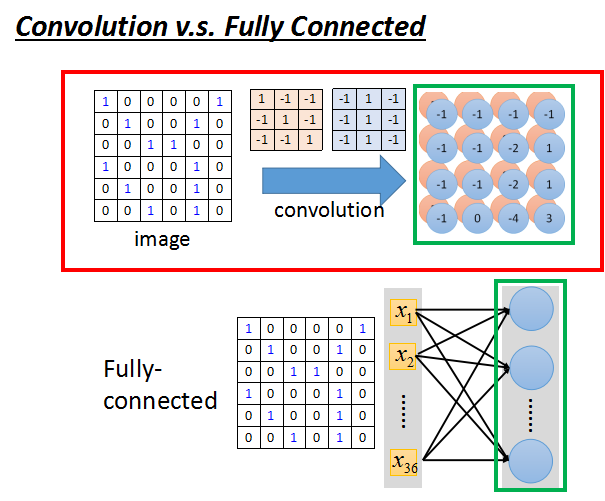
对于一张图片，我们会同时检测很多特征，每个filter只做一件相同的事情，所以需要有很多的Filter，

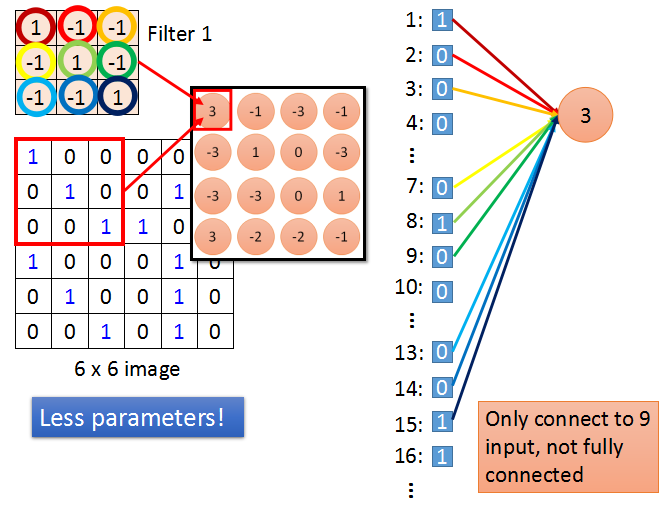
它们放在一起就叫做Feature Map。



对于彩色图片，一个Filter是3维的，如上图，Filter是3×3×3的立方体(tensor).

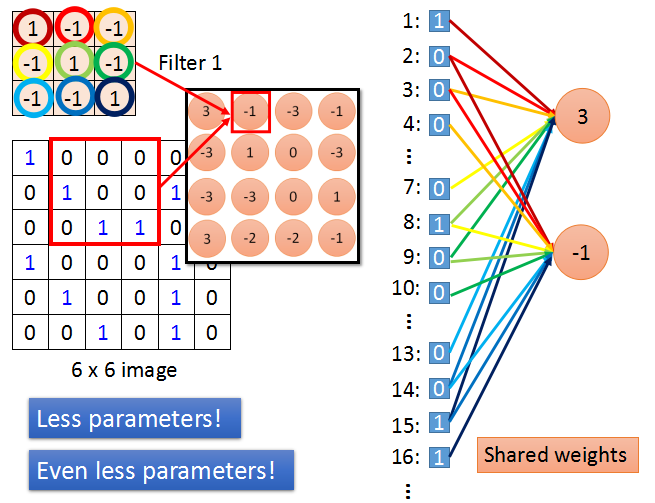
下面将Convolution层和Fully Connected连接对应理解。





将图片拉直成一个列向量，上面的Filter连接的是1，2，3，7，8，9，13，14，15的输入单元，而不是全连接，Filter的每一个分量可以看作是全连接网络中的w和b。

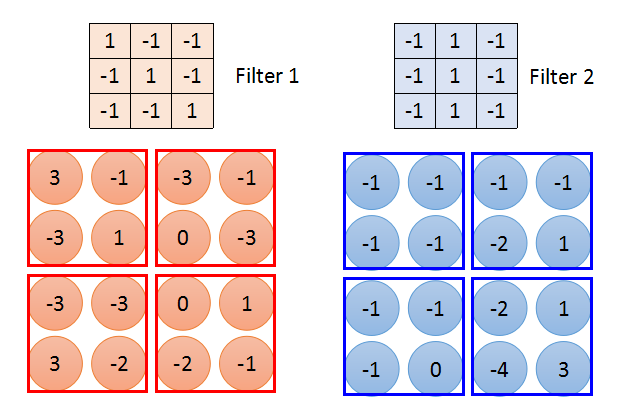
这样相比于全连接的网络，就只需要更少的参数。



而且，之前讨论过，检测同一个特征只使用相同的Filter，所以每一个neuron共用相同的参数，这就是Shared weights。

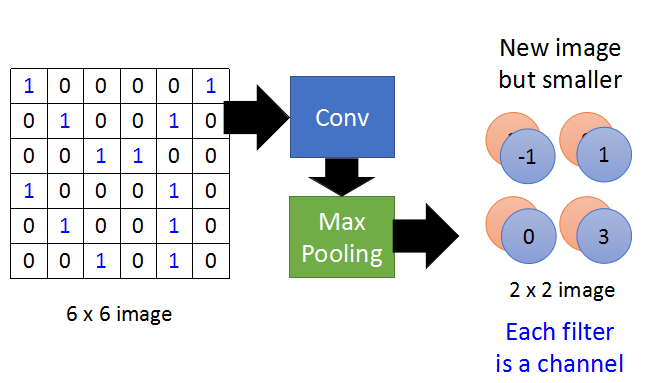
这会使CNN的参数变得更少。

**CNN – Max Pooling**



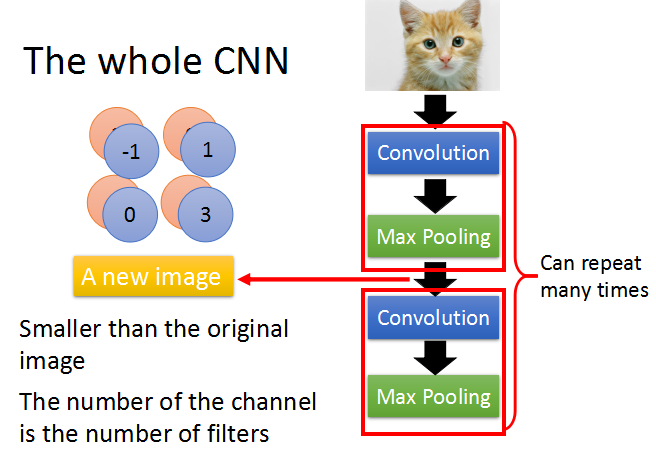
将每一个Filter检测后的结果，划分成2×2的小块，在每一块中可以取均值或最大值，代替这四个值，这样就实现了Subsampling的功能。

取最大值的方法就是Maxpooling。

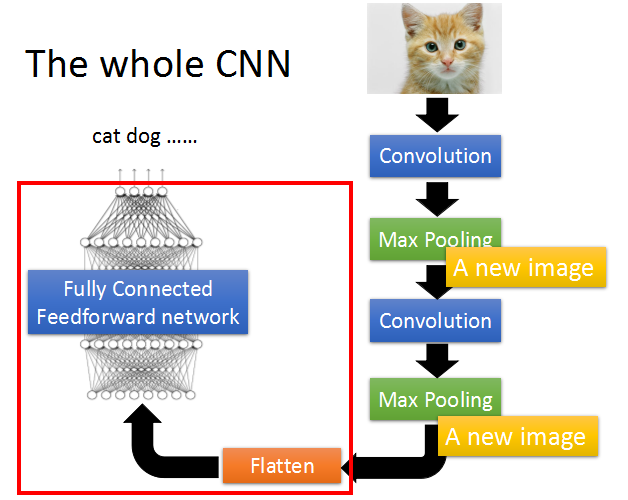


一副图像，经过Conv和Maxpooling后，会变成一幅小的新图像。可以再它之上继续进行Conv和Maxpooling。

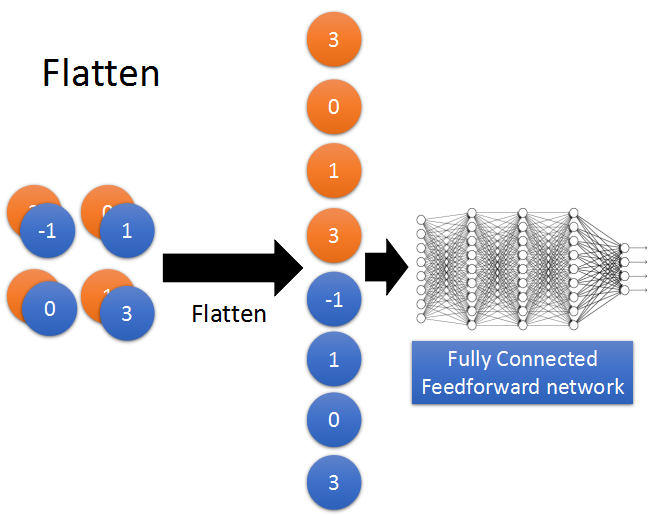
经过MaxPooling处理后会产生和Filter数目相同的“新图像”，每一个Filter都可以看作是处理之前图像的一个channel。



**Flatten**



最后的Flatten就是将上一层Maxpooling得到的image拉直成列向量，作为全连接网络的输入。



以上就是一个CNN神经网络的所有模块简介。最后附一张全图。

