task06

image classification

假设输入图片大小是固定的，如果大小不一样，我们会将所有图片都先rescale成大小一样，然后再丢到影像的辨识系统里面。

模型的目标是分类，把每一个类别表示成一个One-Hot的Vector，目标就叫做$\hat{y}$。

在这个one-hot的vector里面，假设现在类别是一个猫的话，那猫对应的dimension，它的数值就是1，其他的东西所对应的dimension数值就是0. dimension的长度决定了模型可以辨识出多少不同种类的东西。即如果向量长度为1000，则可以辨识出1000种不同的东西。

模型的输出通过softmax后，输出为y'，希望y'和$\hat{y}$的cross entropy越小越好，接下来的问题是怎么把一张影像当作一个模型的输入。

对于彩色图像，可以认为是三维的Tensor(可以理解为维度大于2的矩阵就是Tensor)。

三维的Tensor，分别代表图片的宽、高、channel数目。对于彩色的图片，三个Channel代表了RGB三个颜色。

接下来把一个三维的Tensor拉直，丢到一个network里面。

如100x100x3个数字，通通排成一排，100x100+100x100+100x100，每一维里面存的数值，就是某一个pixel某一个颜色的强度。

非常长的一个vector，假设第一个隐藏层的neuron的数目有1000个，那weight将由1000*100*100*3，即3x10的7次方。

参数越多会有什么问题呢?

虽然随着参数的增加，可以增加模型的弹性，增加它的能力，但也增加了overfitting的风险。如果模型的弹性越大，就越容易overfitting。

observation 1

对一个影像辨识的Neuron，侦测现在这张图片里面，有没有出现一些特别重要的pattern。

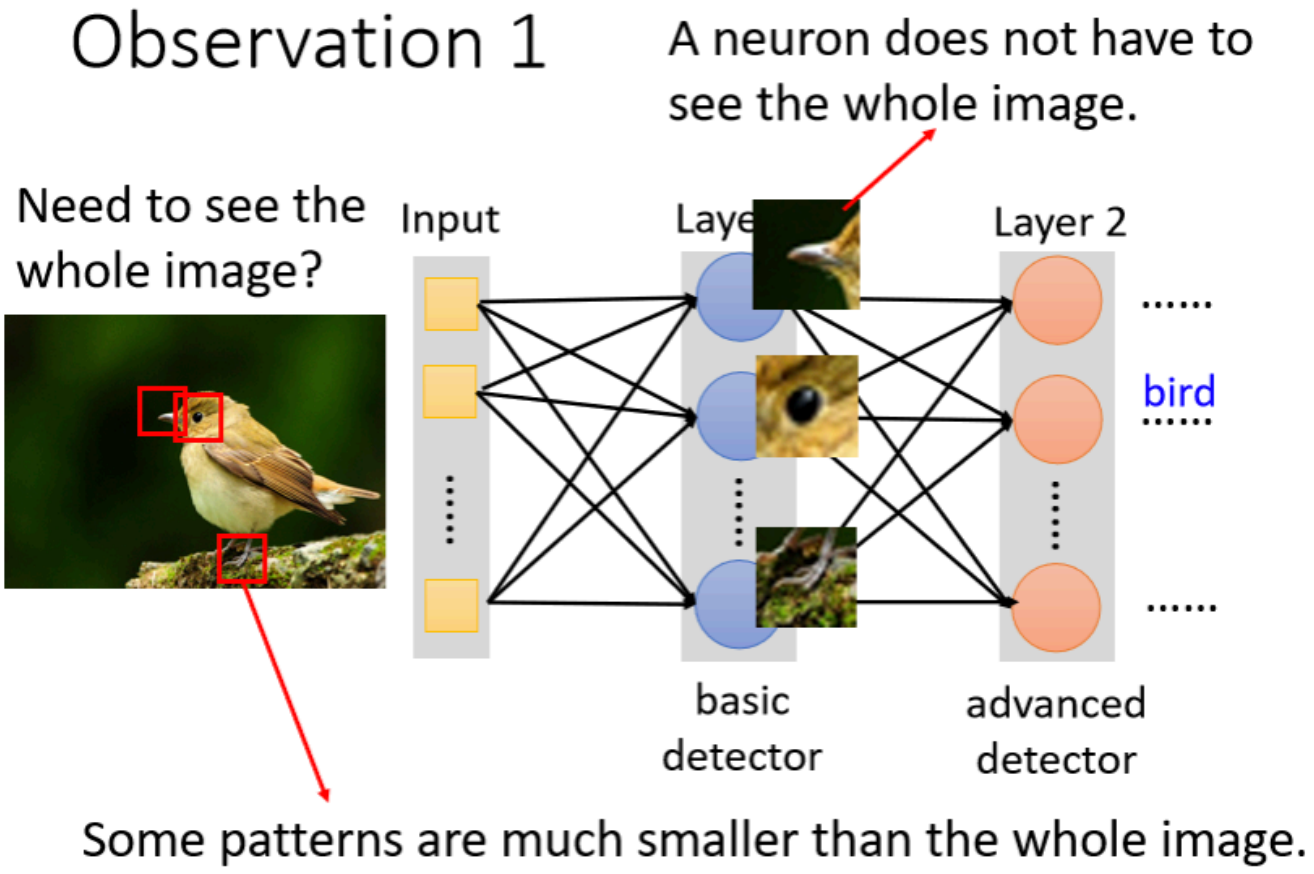
举例来说：

- 有一个Neuron，它看到鸟嘴这个pattern
- 有某个Neuron又说，它看到眼睛这个pattern
- 又有某个Neuron说，它看到鸟爪这个pattern

这些pattern综合起来可能就代表，我们看到了一只鸟，类神经网络可以告诉你说，因为看到了这些pattern，所以它看到了一只鸟。

仔细想想，人类是不是也用同样的方式，来看一张图片中有没有一张鸟呢。有个鸟嘴，有个眼睛，可能认为是一个乌鸦，但其实是一只猫。所以其实就算是人，我们在判断一个物件的时候，往往也是抓最重要的特征，然后看到这些特征以后，会很直觉的认为你看到了某种物件，对机器来说，也许也是一个有效的判断影像中有什么物件的方法。

但现在假设我们用neuron做的事情，其实就是判断说现在有没有某种pattern出现，这样也许不需要每一个neuron都去看一张完整的图片。

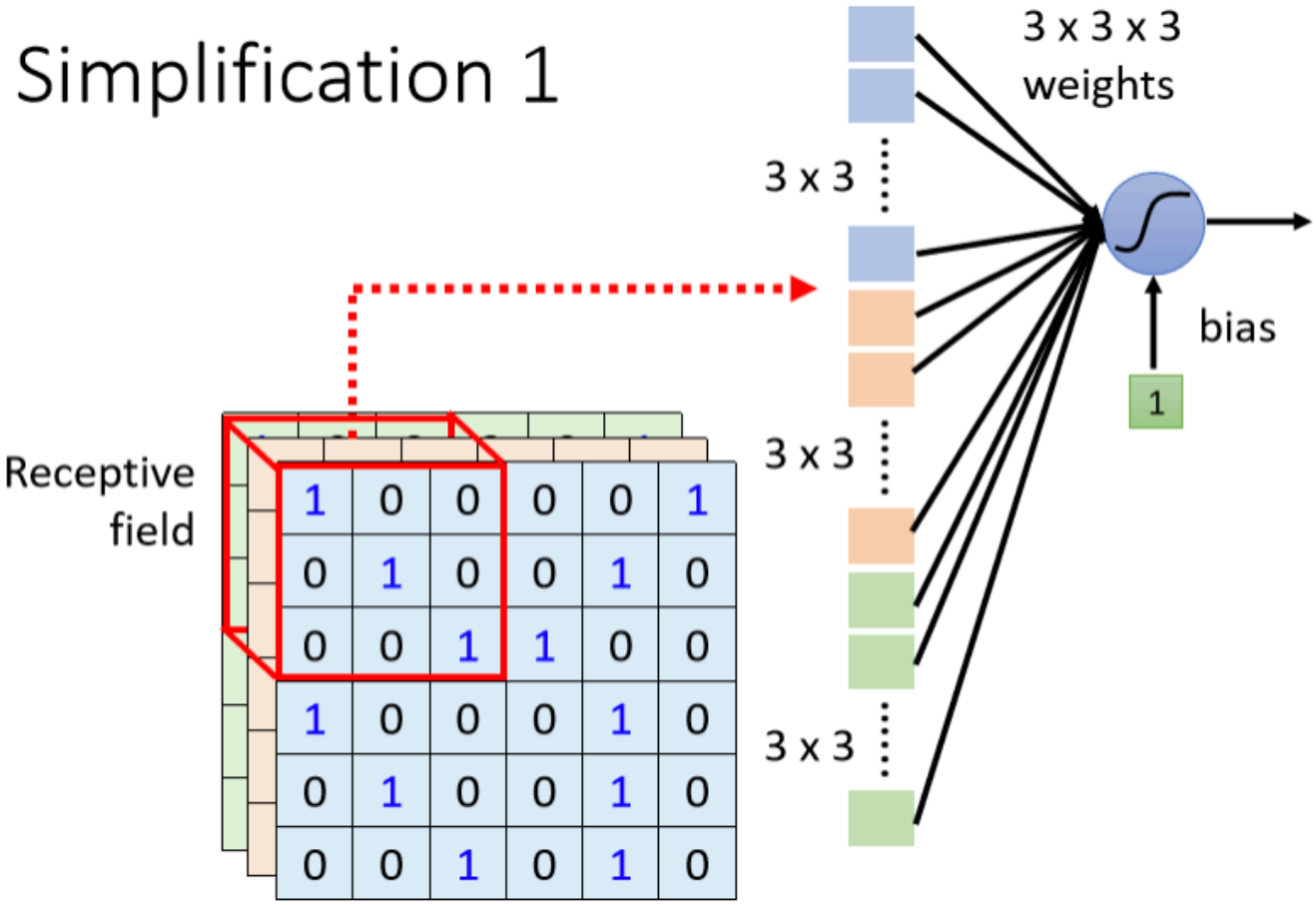


因为一些重要的pattern，比如说鸟嘴、眼睛、鸟爪，并不需要看整张完整的图片，才能够得到这些资讯。所以这些Neuron也许根本就不需要把整张图片当作输入，他们只需要把图片的一小部分当作输入。

Simplification 1

设定一个区域Receptive Field(感受野)，每一个Neuron都只关心自己的Receptive Field里面发生的事情就好了。

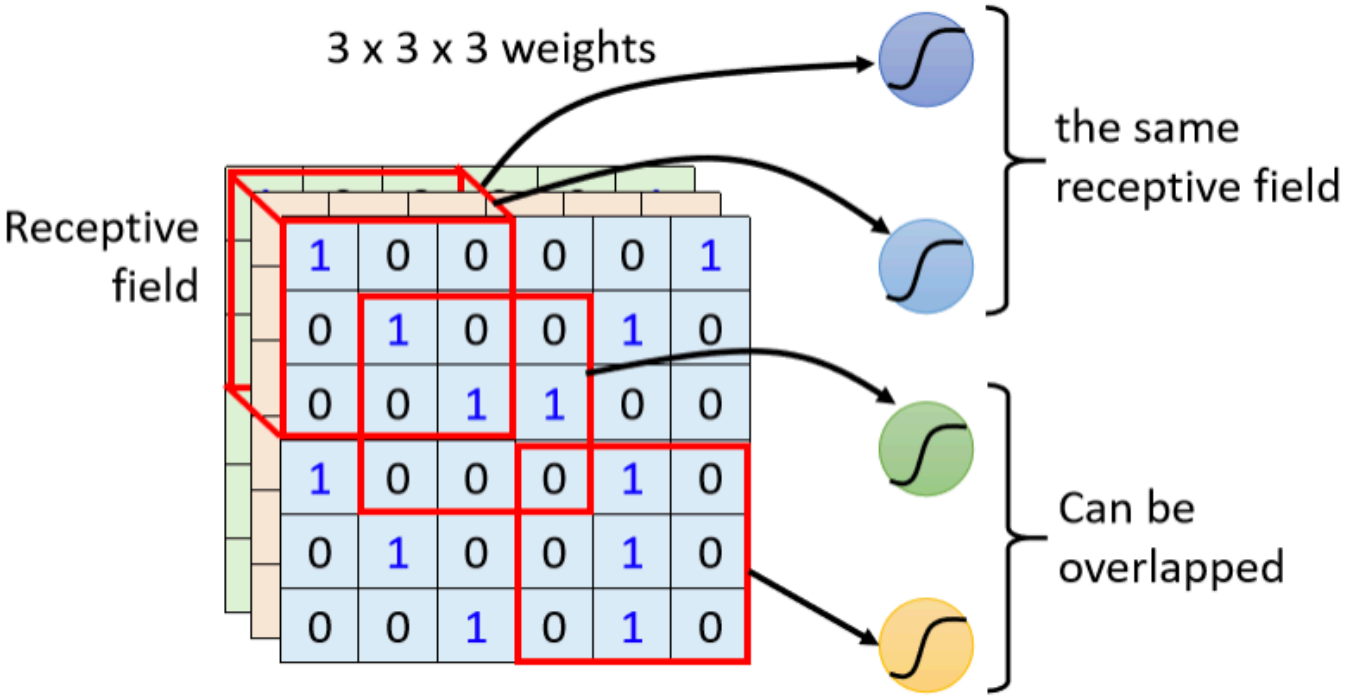
举例来说，先定义说这个蓝色的Neuron，它的守备范围就是这一个Receptive Field，那这个感受野里面有333个数值，对蓝色的Neuron来说，只需要关心这一个小范围就好，不需要在意整张图片里面有什么东西。



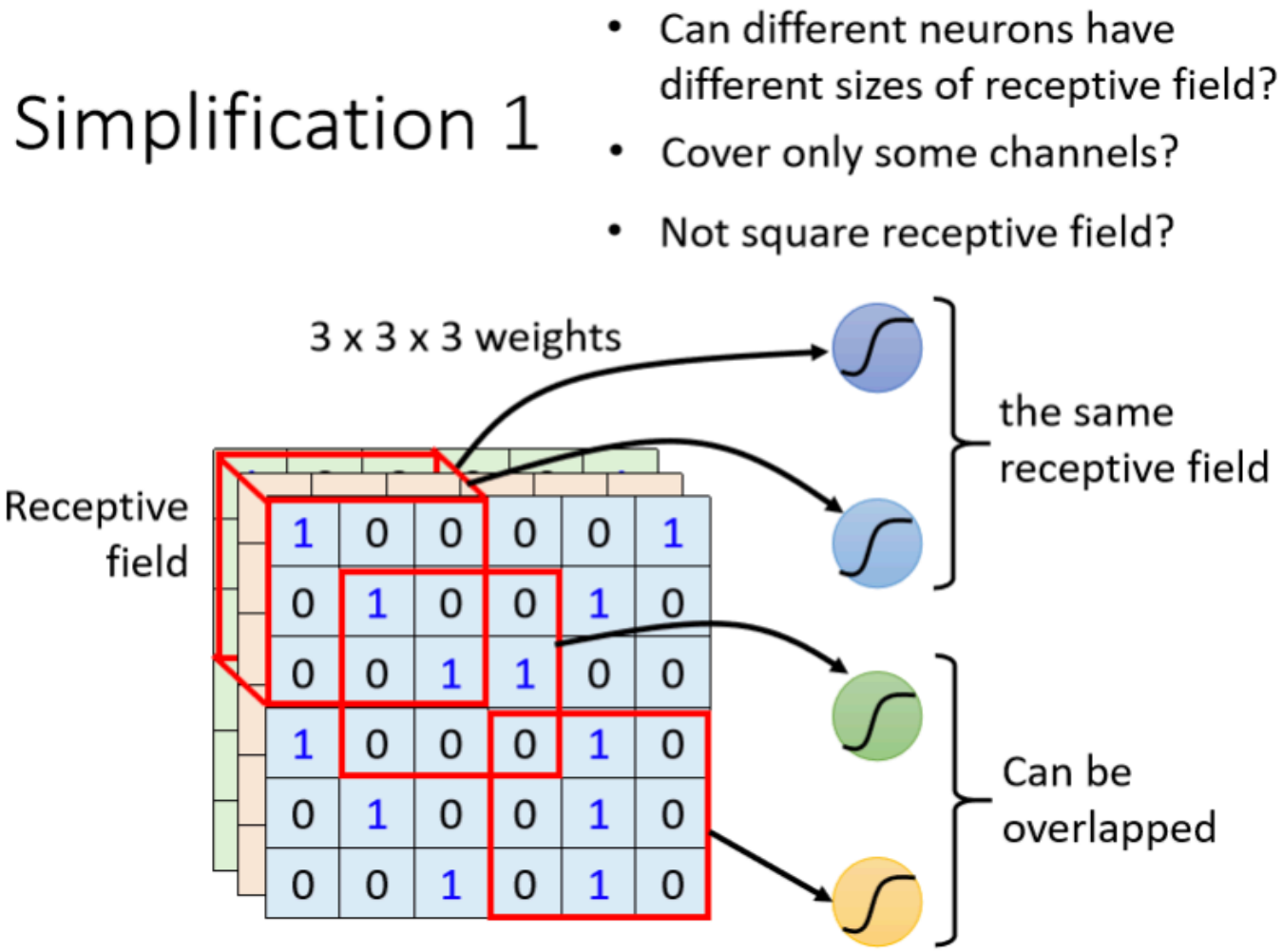
要做的事情为：

- 把这3x3x3多数值拉直，变成一个长度为3x3x3也就是27维的向量，再把这27维多向量作为Neuron的输入。
- 这个Neuron会给27维的向量，每一个Dimension一个weight，这个Neuron有3x3x3 ，27个weight
- 再加上bias得到的输出，再送个下一层的Neuron当作输入。

所以每一个Neuron只考虑自己的Receptive field，但这个感受野怎么决定出来，就要问自己了。



- 可以说这边有个蓝色的Neuron，就看左上角的范围，这就是它的Receptive field
- 另外又有一个黄色的Neuron，看右下角的这个3x3x3的范围
- 那Receptive field彼此之间可以是重叠的，比如说画一个receptive field，这个地方是绿色的，Neuron的守备范围，它跟蓝色的跟黄色的都有一些重叠的空间。
- 那甚至可以两个不同的neuron，守备看到的范围是一样的，也许一个范围使用一个neuron来守备，没有办法侦测所有的pattern，所以同个范围可以有多个不同的neuron，同个Receptive field可以有多个



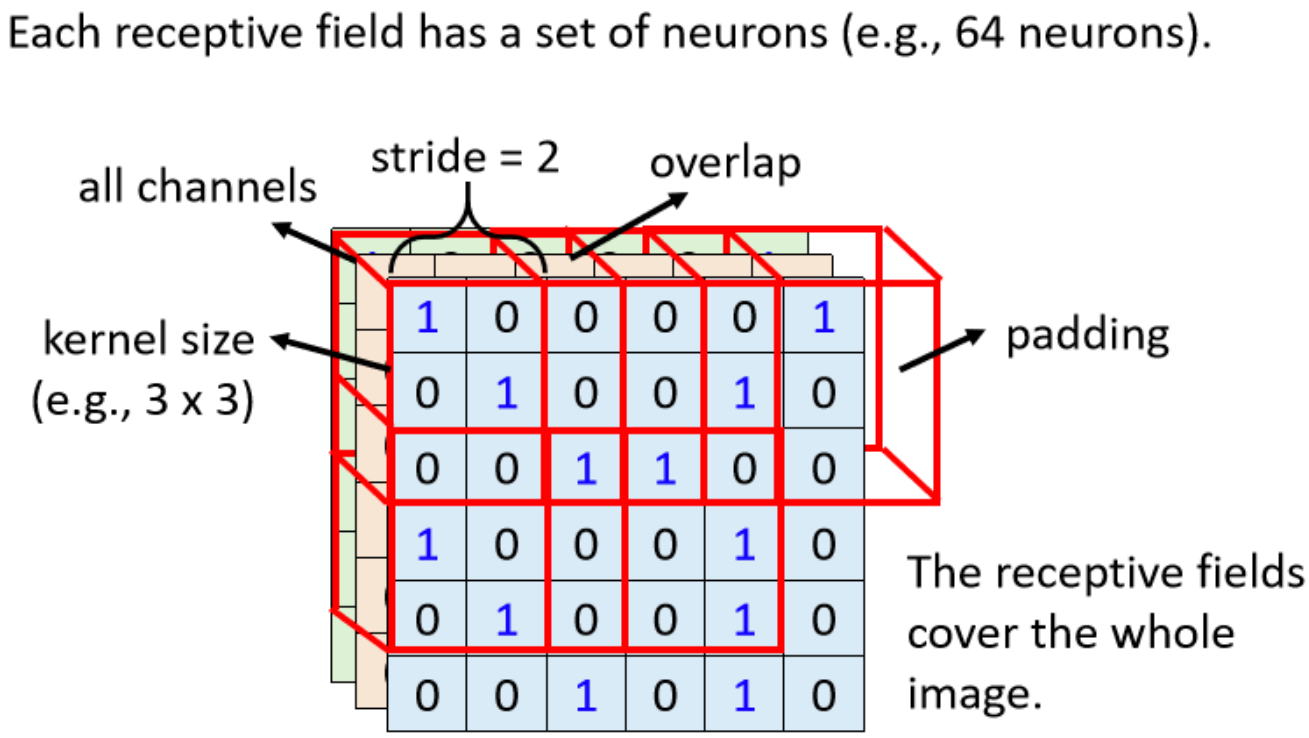
- Receptive field可不可以有大有小？ 因为pattern有的比较小、有的比较大，需要11x11的范围才能被侦测出来。
- Receptive field可不可以只考虑某些channel？一般不考虑，但也存在这样子的做法。比如有些只在蓝色的channel出现。
- Receptive field通常是正方形的，也可以自己定义成长方形的

simplification 1 - typical setting

虽然可以任意设计，但也有最经典的Receptive Field的安排方式。

1. 看所有的channel， kernel size

   展示了3x3的kernel size。一般同一个Receptive field不会只有一个Neuron去关照他，往往会有一组、一排Neuron去守备它。比如64个。



2. Stride,

   把最左上角的感受野往右移一点，制造另外一个感受野，这个移动的量为stride，为超参数，由自己设定

3. Padding

   超出影像的范围，可以用0补齐

4. 除了横着移动，还有垂直方向的移动

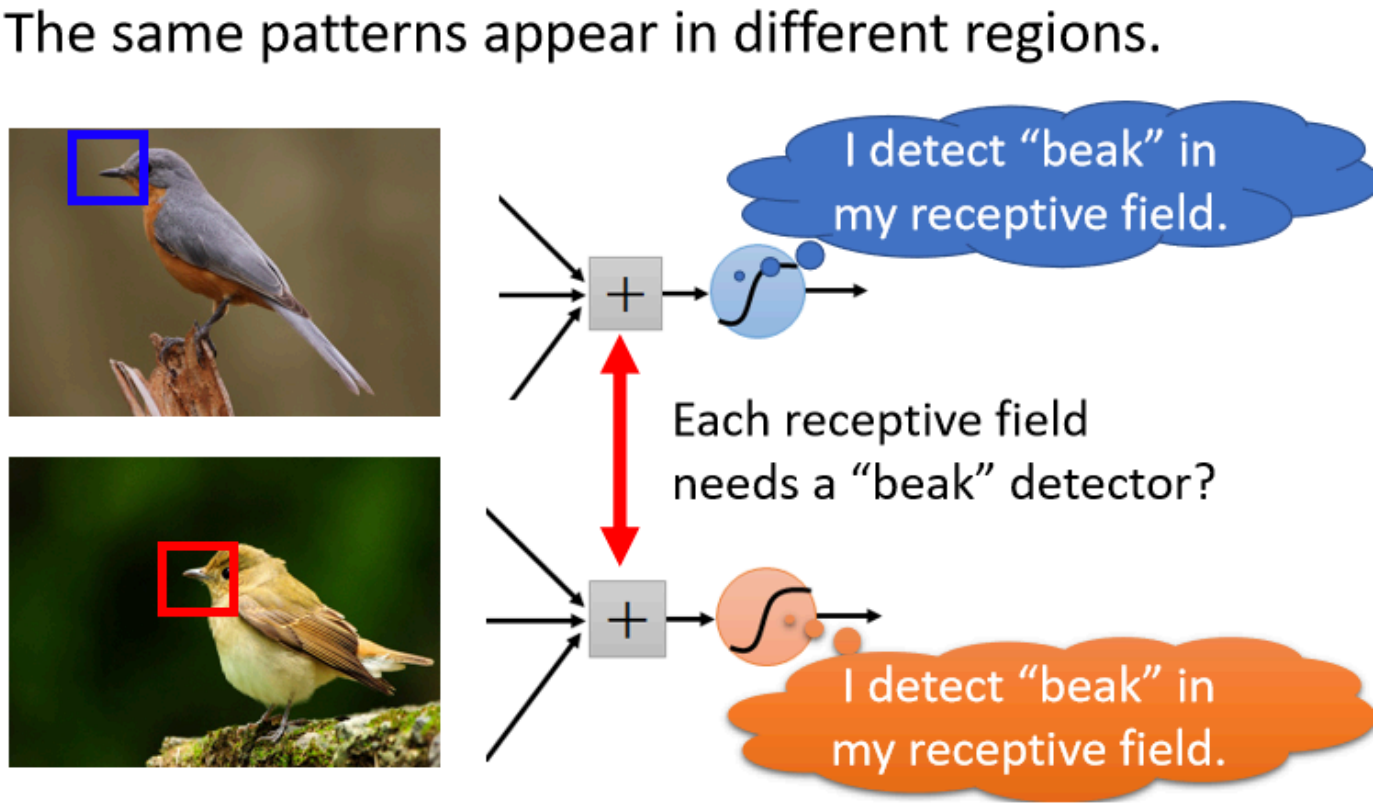
observation 2

第二个观察，同样的pattern，可能会出现在图片的不同区域里面

比如同样一个鸟嘴，可能出现在左上角，也可能出现在中间。

有个问题，是否真的需要每一个守备范围，都去放一个侦测鸟嘴的Neuron吗?

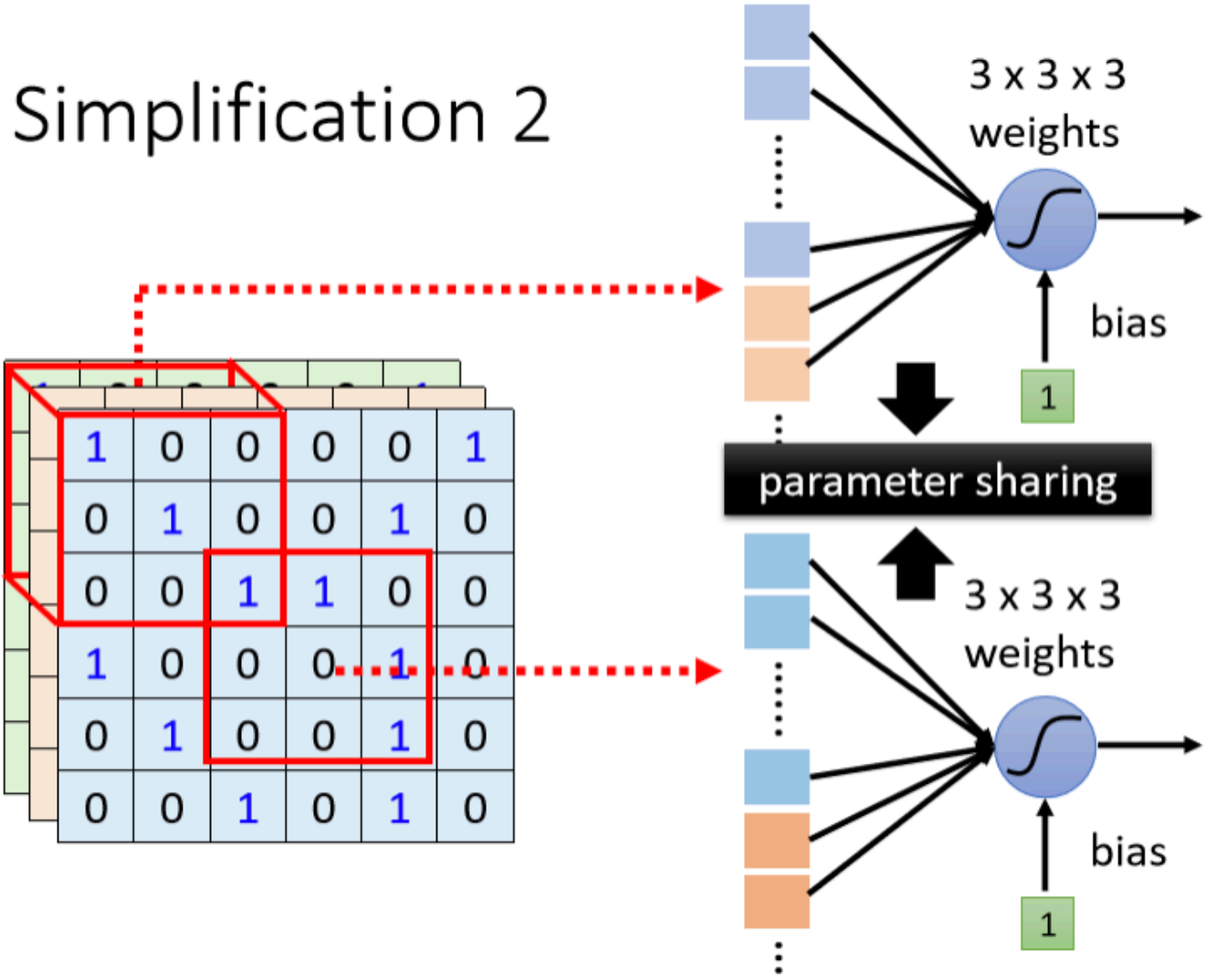
如果需要，那参数量不会太多了吗?

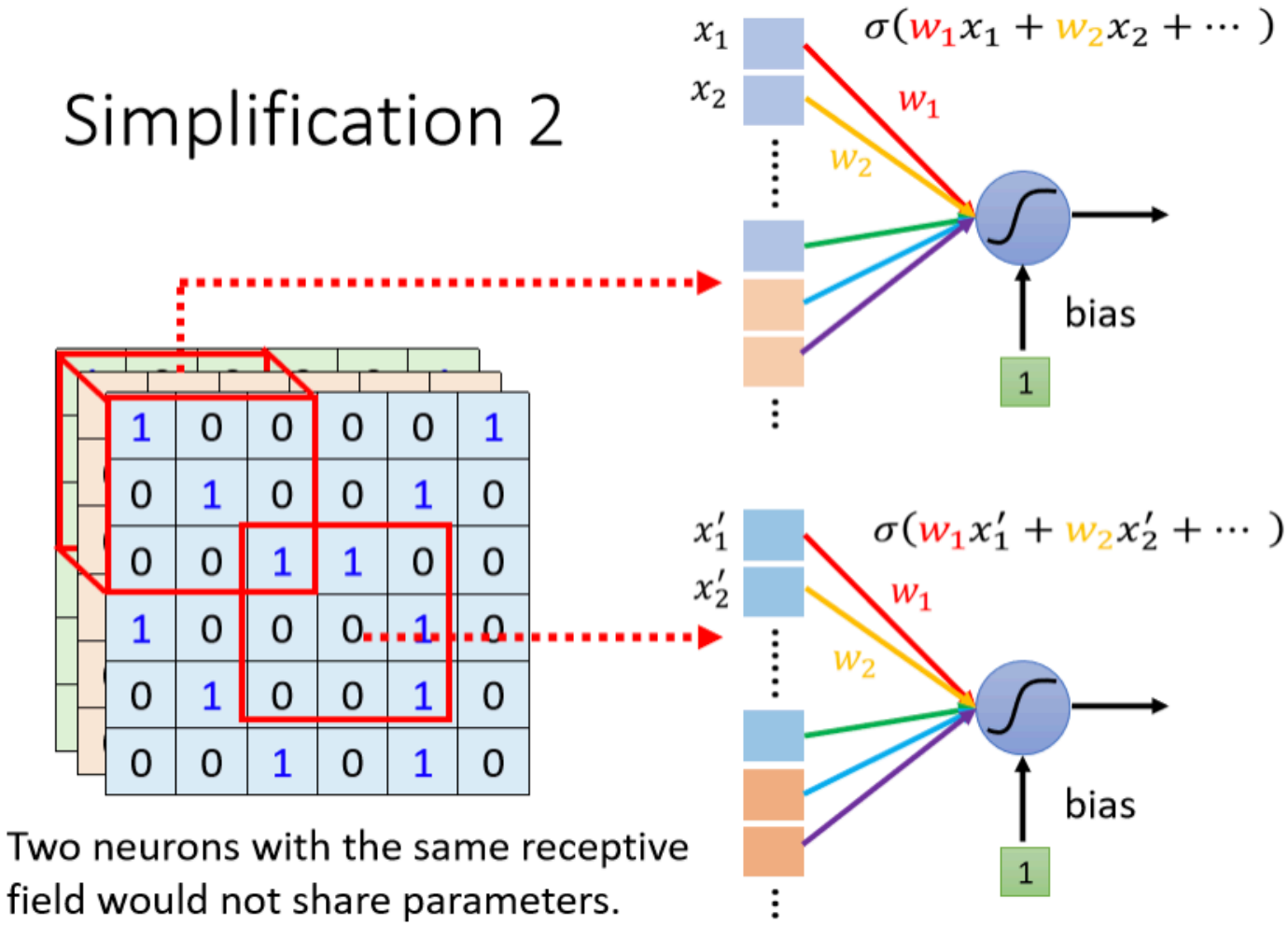


simplification 2

不同receptive field的Neuron共享参数，也就是做Parameter sharing **权值共享**：即两个Neuron它们的weights完全是一样的。

Simplification 2





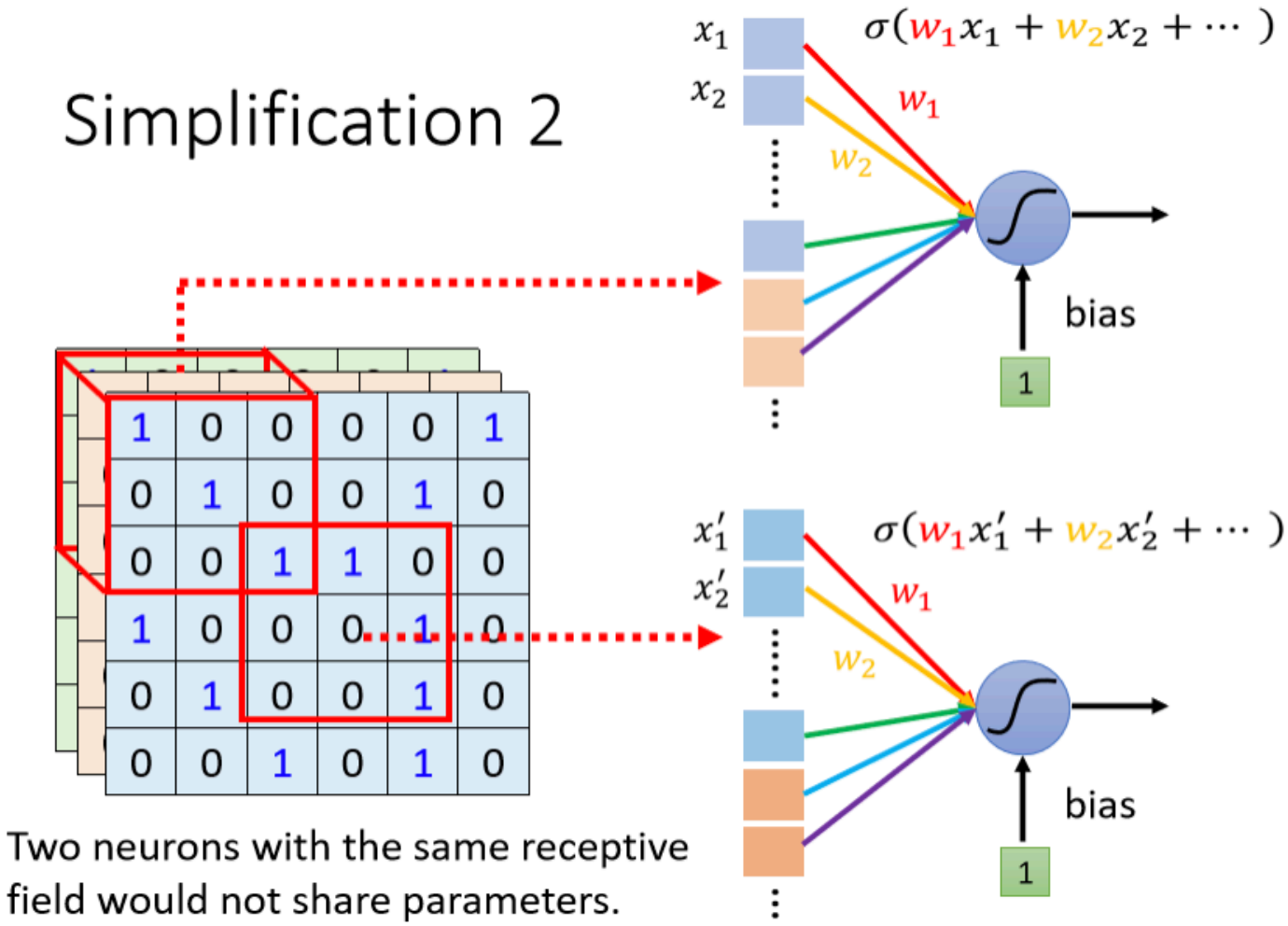
- 上面这个Neuron的一个weight叫做w1，下面这个Neuron的第一个weight也是w1，它们是同一个weight，用红色来表示。
- 上面这个Neuron的第二个weight是w2，下面这个Neuron的第二个weight也是w2，都用黄色来表示

这两个Neuron守备的Receptive field不一样，但它们的参数是一摸一样的。

那它的输出会永远一样吗？ 不会，因为它们的输入不一样，即使参数一摸一样，但它们照顾的范围不一样。

Simplification 2 -Typical Setting

怎么共享呢？可以完全自己决定，但存在常见的共享方法设定。



比如说64个Neuron，所以这个感受野有64个Neuron。用一样的颜色代表这两个Neuron共享一样的参数。

所以每一个receptive field都只有一组参数而已。

benefit of convolutional layer

讲了fully connected的network，它弹性最大，但有时候不需要看整张图片，只要看图片的一小部分就可以侦测出重要的pattern，所以有了receptive field的概念。

当强制一个neuron只能看到一张图片里的一个范围，它的弹性是变小的。如果是全连接，可以决定看整张图片还是只看一部分，如果只看一部分，就把许多weights设为0，所以加入receptive field，弹性是变小的。

权值共享又是如何限制弹性的呢?

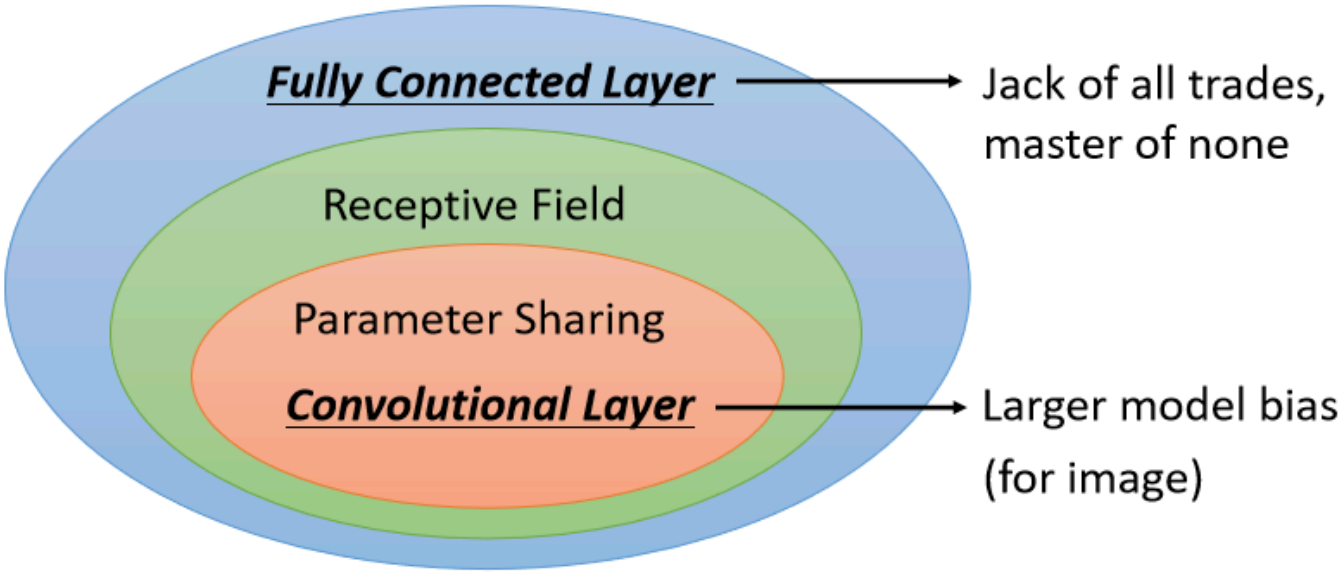
每一个neuron可以各自有不同的参数，可以正好学出一摸一样的参数，也可以有不一样的参数。

但加入参数共享以后，某一些neuron参数要一摸一样，所以这又更增加了对Neuron的限制。而Receptive field加上parameter sharing 就是convolutional layer。

有用到convolutional layer的network就叫cnn。可以看出cnn的bias比较大，model的bias比较大。

model bias大，不一定是坏事

- 小的话，model 的flexibility很高时，容易overfitting，全连接层可以做各式各样的，但可能没有办法在任何特定的任务上做好
- 而卷基层，专门为影像设计的，虽然bias很大，但在影像上不是问题。

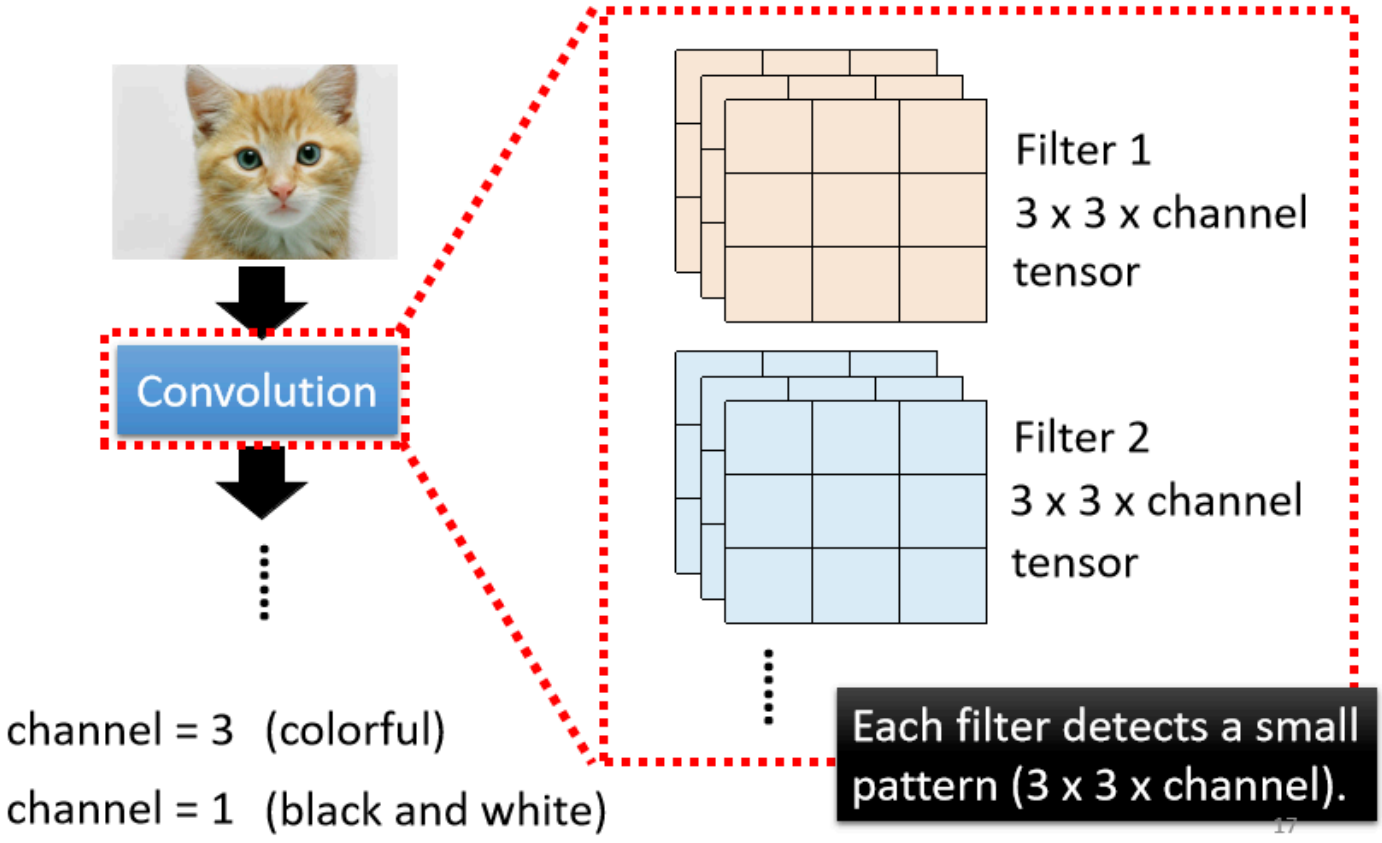


- Some patterns are much smaller than the whole image.
- The same patterns appear in different regions.

convolutional layer

以上讲的影像用的特性为cnn的某一种介绍方式，第二种介绍方式跟刚才讲的介绍方式一模一样，知识同一个故事，用不同版本来说明。

convolutional 的layer里面有很多的filter。

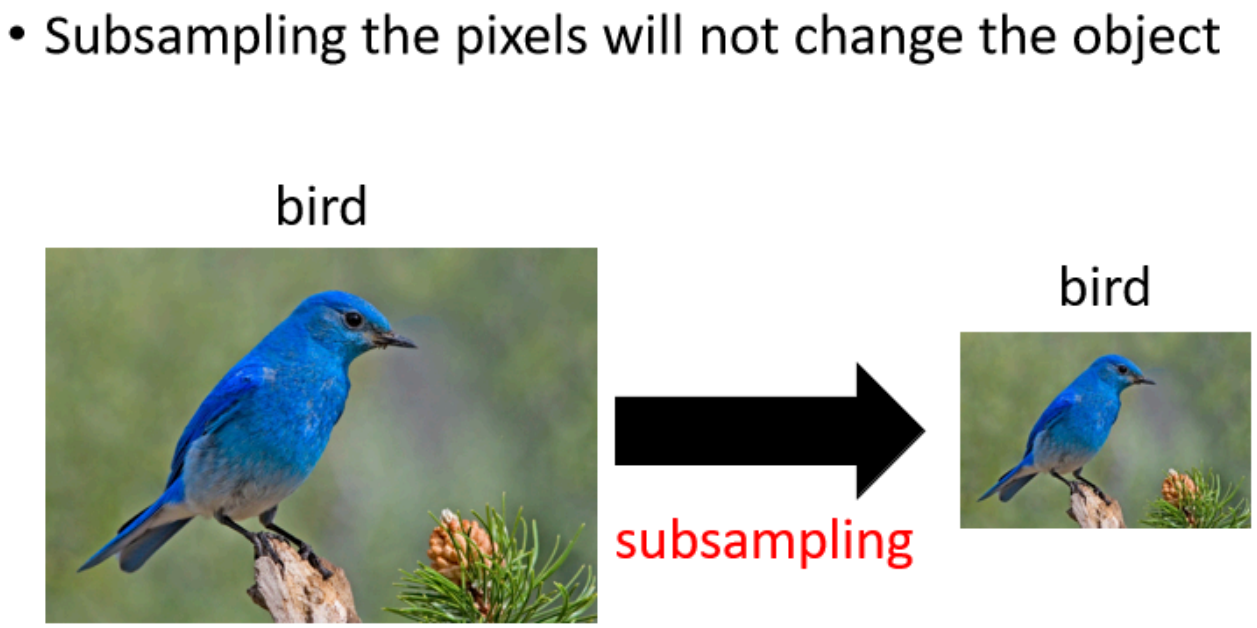


xxxxxx

xxxxxx

observation 3

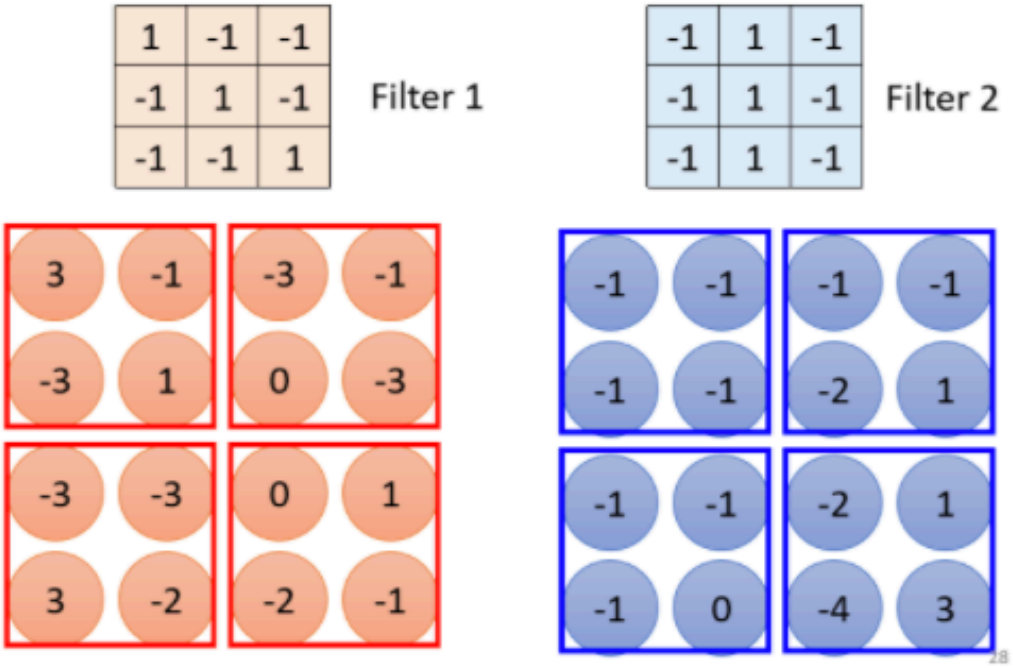
第三个常用的东西，pooling。



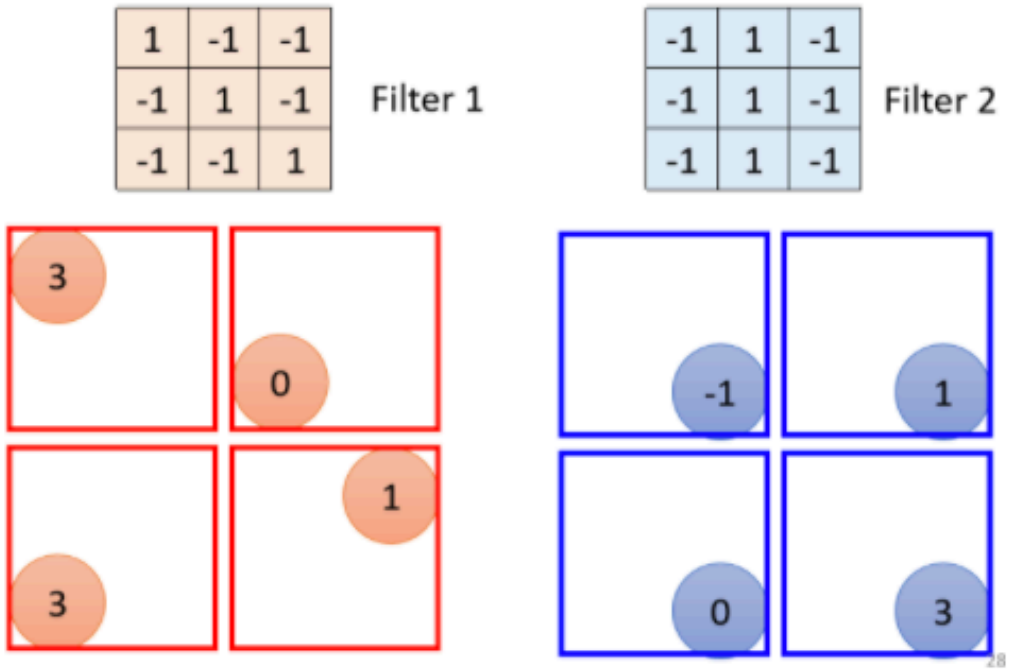
把比较大的图片做subsampling，举例来说把偶数的column都拿掉，奇数的row都拿掉，图片变成为原来的1/4，但不会影像里面是什么东西。把一张大的图片缩小，是一只鸟，这张小的图片看起来还是一只鸟。

pooling本身没有参数，所以它不是一个layer，没有weight，没有要learn的东西。 比较像activation function，比较像是sigmoid，relu那些。就是一个operator，行为都是固定好的。

Pooling – Max Pooling

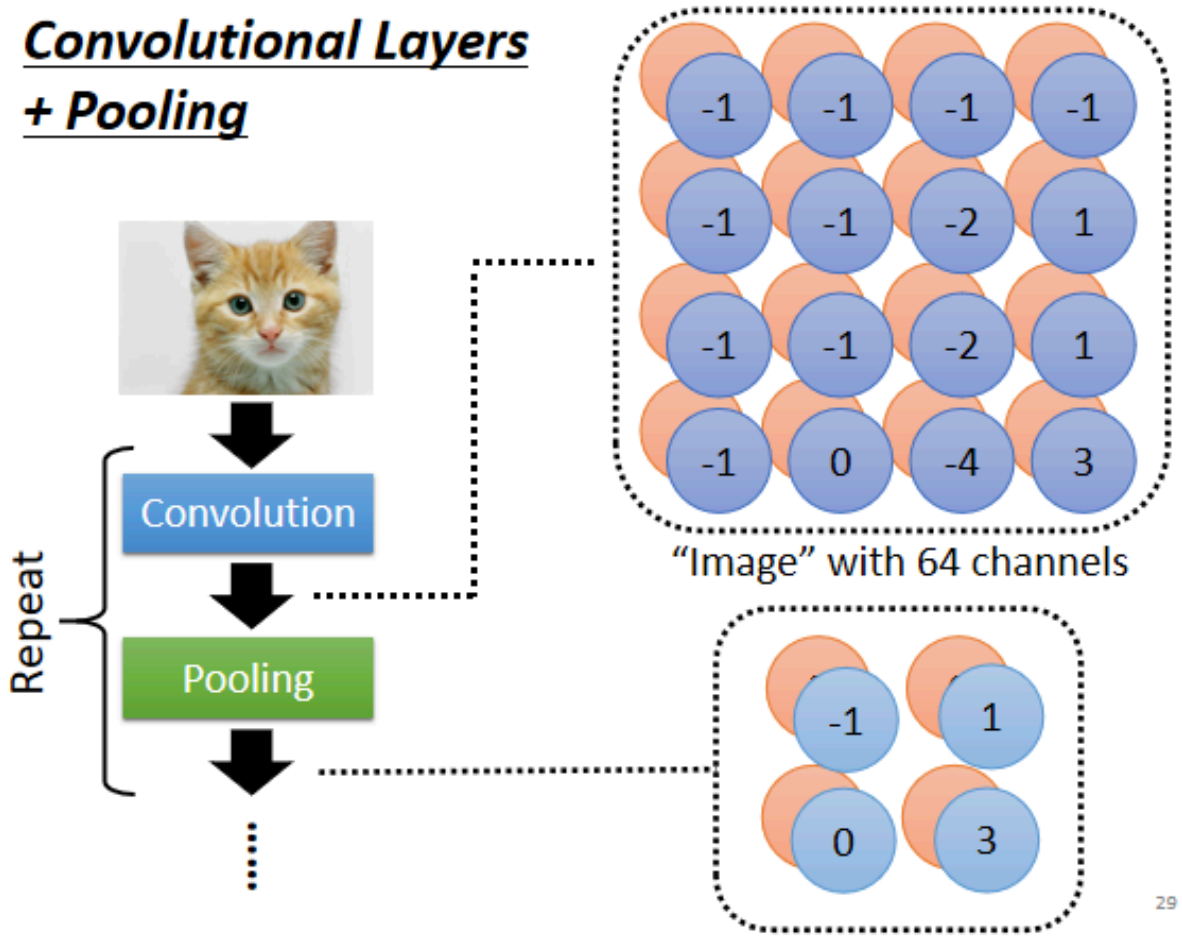


Pooling – Max Pooling



Convolutional Layers + Pooling

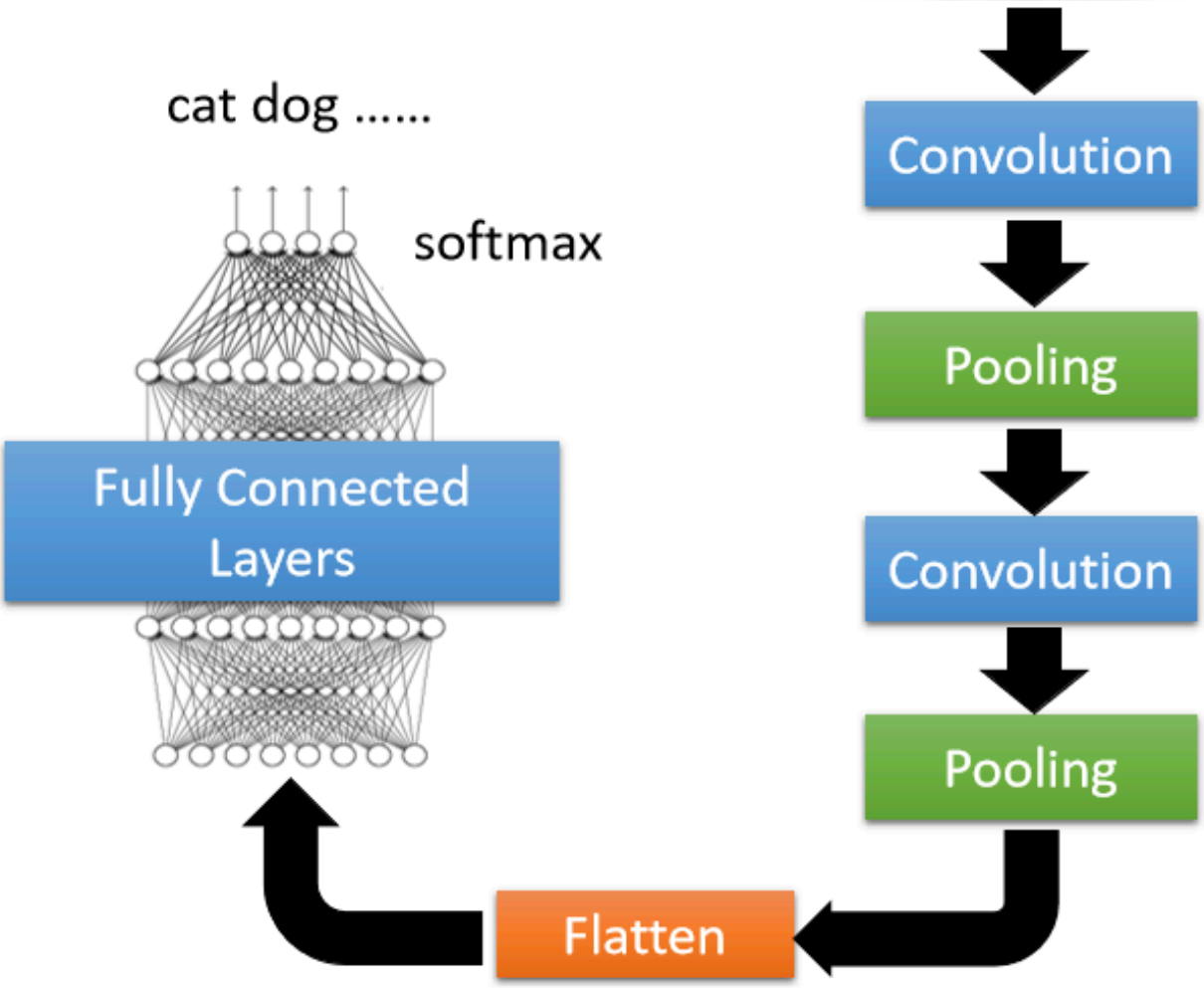
pooling做的事情就是把图片变小。一般convolution跟pooling交替使用。pooling会对性能造成一定伤害，最主要的理由是为了较少运算量。如果今天的运算资源足够支撑不做pooling的话，可以不用pooling。



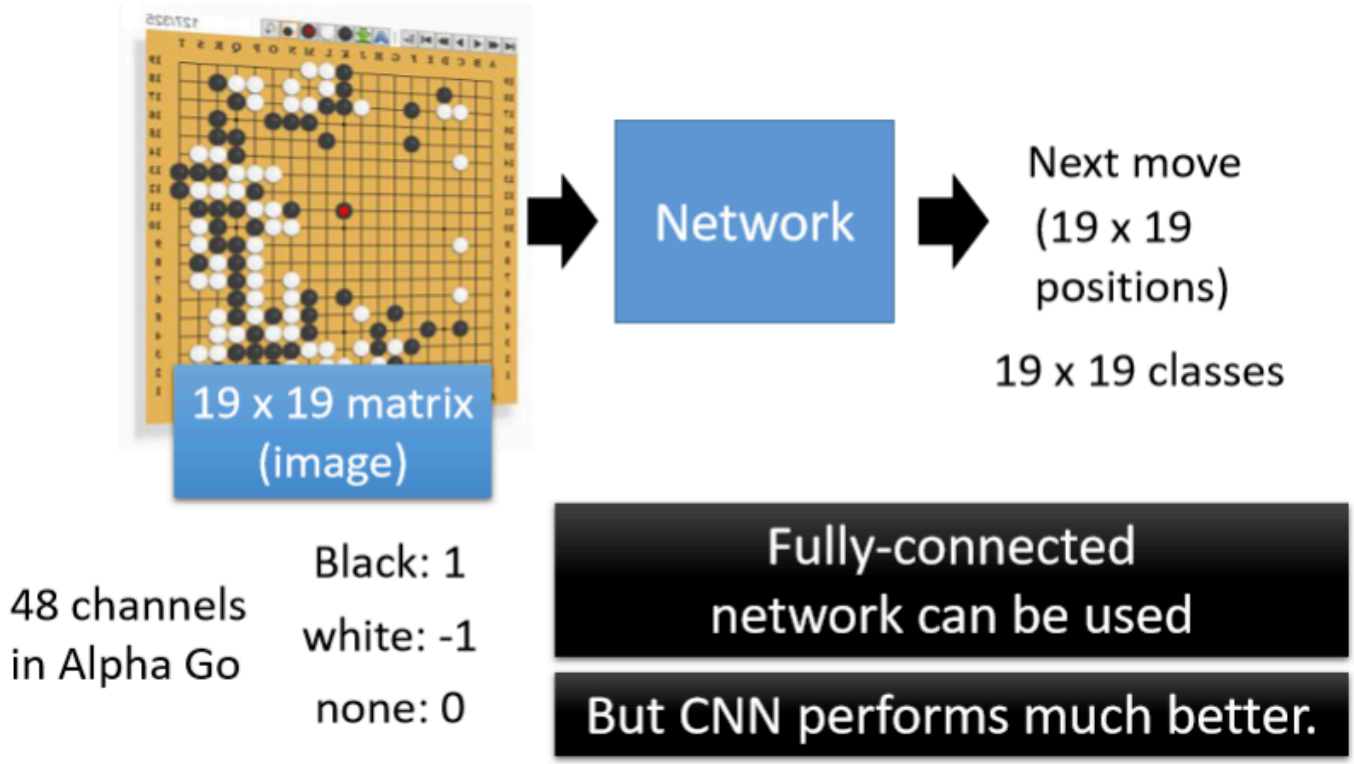
The whole CNN

做完几次convolution以后，接下来会把pooling的output做一件事情，叫做flatten。意思就是把这个影像里面的本来排成矩阵样子的东西拉直，把所有数值拉直成一个向量，再丢进全连接层。

The whole CNN



Application： Alpha Go



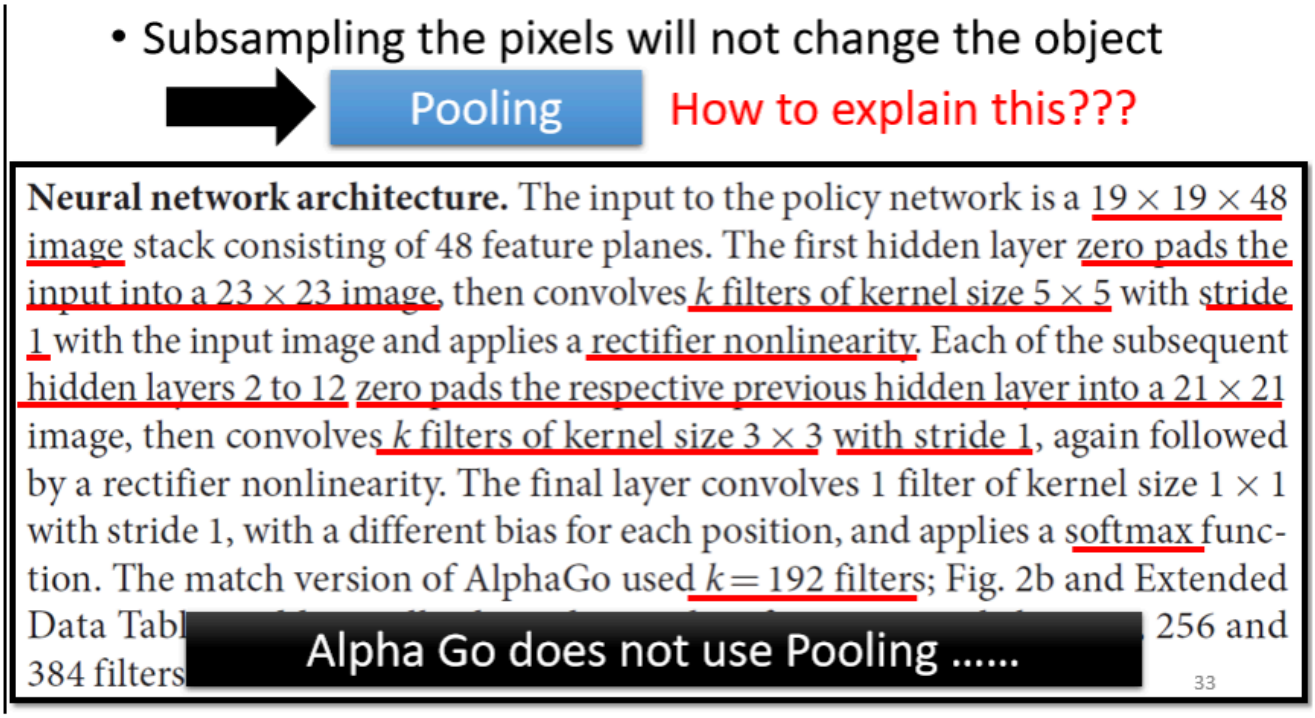
第一个观察是,很多重要的 Pattern,你只需要看小范围就知道,下围棋是不是也是一样呢

第二个观察是,同样的 Pattern 可能会出现在不同的位置,在下围棋裡面是不是也是一样呢

让人想不透的地方是,在做影像的时候我们说我们都会做 Pooling,也就是一张**影像做 Subsampling 以后,并不会影响我们对影像中物件的判读**

但是棋盘是这个样子吗,你可以把棋盘上的奇数行跟偶数列拿掉,还是同一个棋局吗,听起来好像不是对不对,下围棋这麼精细的任务,你随便拿掉一个 Column 拿掉一个 Row,整个棋整个局势就不一样啦,怎麼可能拿掉一个 Row 拿掉一个 Column,还会没有问题呢

- 我们把一个棋盘,看作 19 × 19 × 48 大小的 Image
- 接下来它说它有做 Zero Padding,Padding 这件事我们有讲嘛,就是你的 Filter 如果超出影像的范围就补 0,Zero Padding 就是超出范围就补 0 的意思
- 它说它的 Filter 的大小啊,Kernel Size 就是 Filter 的大小是 5 × 5
- 然后有 k 个 Filter,k 是多少,k 是 192,这当然是试出来的啦,它也试了 128 跟 256 发现 192 最好了,好 这是第一层
- 然后 Stride=1,Stride 是什麼 我们刚才也解释过了
- 然后这边有用了 Rectifier Nonlinearity,这是什麼,这个就是 ReLU 啦,这个就是 ReLU
- 然后在第二层呢,到第 12 层都有做 Zero Padding,然后呢 这个 Kernel Size 都是 3 × 3,一样是 k 个 Filter,也就是每一层都是 192 个 Filter,Stride 呢 一样设 1,就这样叠了很多层以后呢,因為是一个分类的问题
- 最后加上了一个 Softmax



你不要看影像上面都有用 Pooling,就觉得 Pooling 一定是好的,在下围棋的时候就是不适合用 Pooling,所以你要想清楚说,你今天用一个 Network 架构的时候,我这个 Network 的架构到底代表什麼意思,它适不适合用在我现在这个任务上,

other

CNN 并没有你想像的那麼强,那就是為什麼在做影像辨识的时候,往往都要做 Data Augmentation,所谓 Data Augmentation 的意思就是说,你把你的训练资料,每张图片都裡面截一小块出来放大,让 CNN 有看过不同大小的 Pattern,然后把图片旋转,让它有看过说,某一个物件旋转以后长什麼样子,CNN 才会做到好的结果

那你说 欸 CNN 这个不能够处理 Scaling,跟 Rotation 的问题啊,那有没有什麼 Network 架构,是可以处理这个问题呢,其实有的,有一个架构叫 **Special Transformer Layer**。