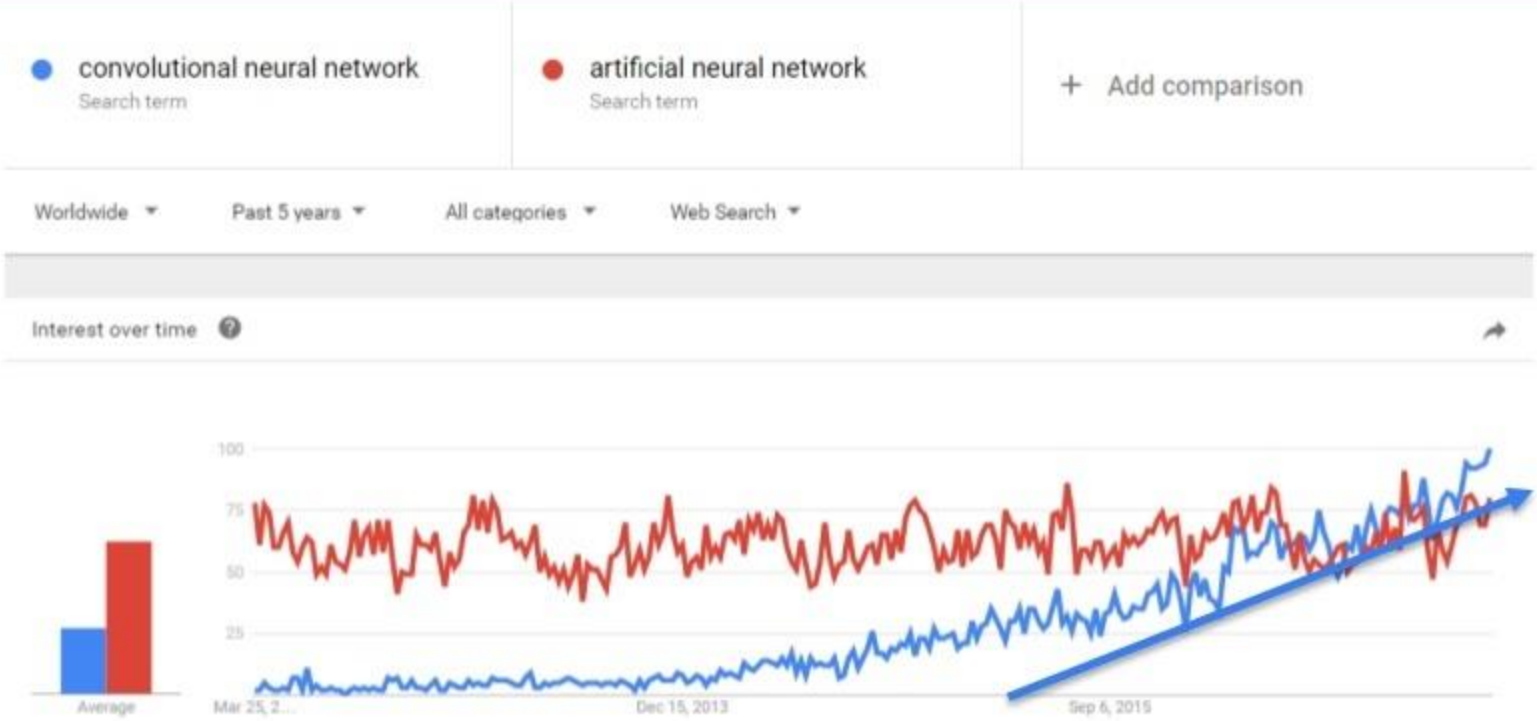
Convolutional Neural Networks

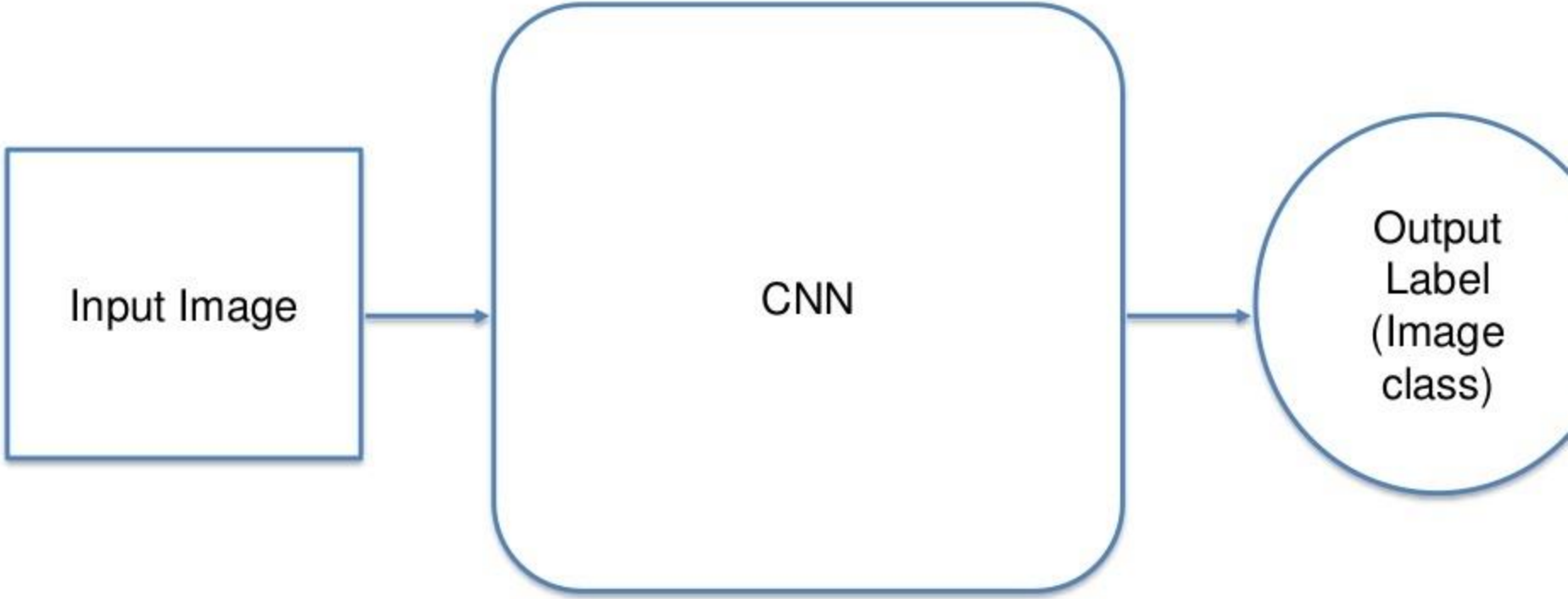
Dr. B M Mainul Hossain
mainul@iit.du.ac.bd

Convolutional Neural Networks

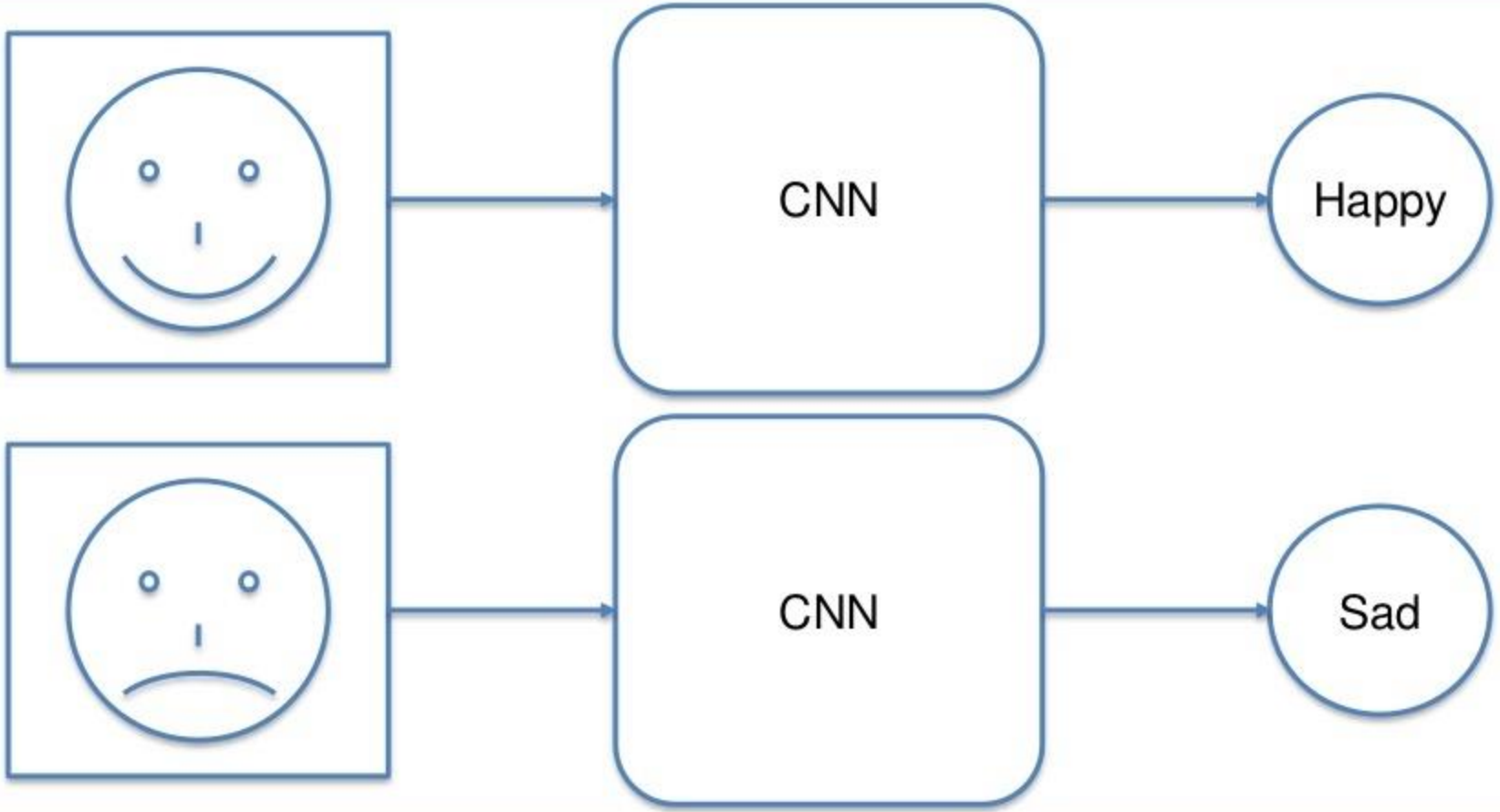


Source: google t

Convolutional Neural Networks



Convolutional Neural Networks



Convolutional Neural Networks

B / W Image 2x2px

Pixel 1	Pixel 2
Pixel 3	Pixel 4

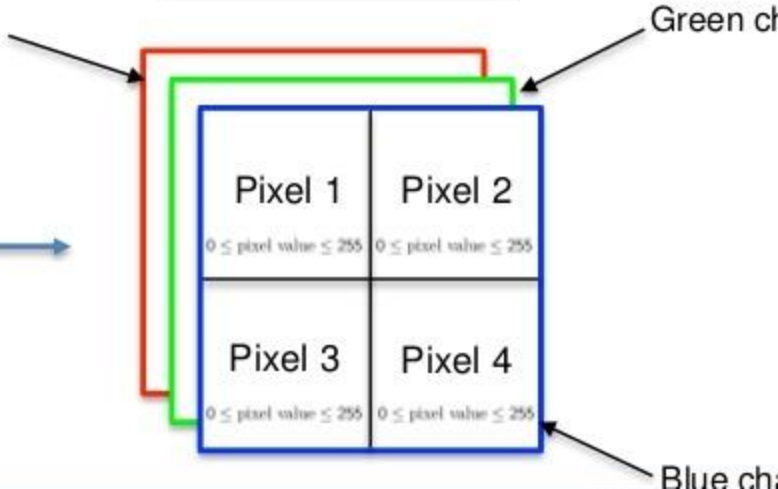
2d array

Pixel 1	Pixel 2
$0 \leq$ pixel value ≤ 255	$0 \leq$ pixel value ≤ 255
Pixel 3	Pixel 4
$0 \leq$ pixel value ≤ 255	$0 \leq$ pixel value ≤ 255

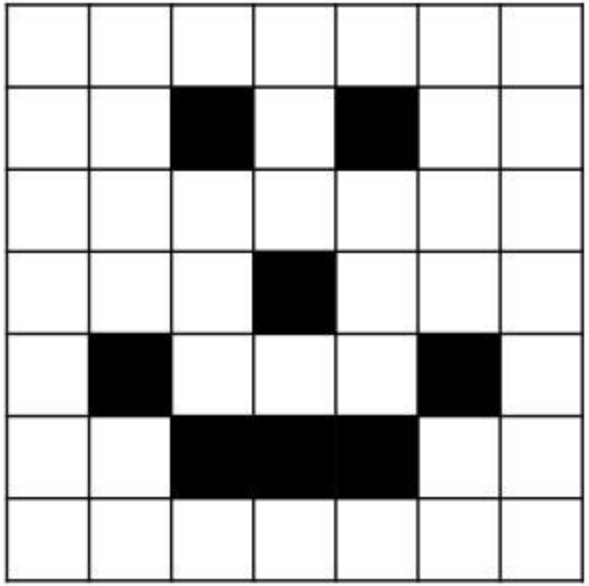
Colored Image 2x2px

Pixel 1	Pixel 2
Pixel 3	Pixel 4

3d array



Convolutional Neural Networks



0	0	0	0	0	0	0
0	1	0	0	0	1	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	1	0	0	0	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

- Suppose our image dimensions are 256 * 256
- Neural network input: 256 * 256 * 3 = 196,608 features in total. (That's a lot of features!)
- From these 196,608 features, answer- is there a cat in the picture?
- 196,908 features means 196,908 + 1 = 196,909 parameters in one neuron.
- Neuron first takes some linear combination of the input features before applying an activation function.

Summary: Images are a 3-dimensional array of features: each pixel in the 2-D space contains three numbers from 0–255 (inclusive) corresponding to the Red, Green and Blue channels. Often, image data contains a lot of input features.

. . .

Recall from above that the nature of images is such that:

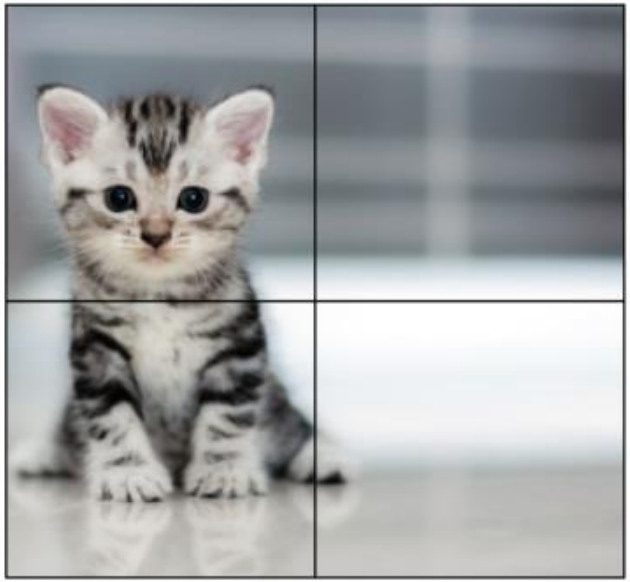
1. There are a lot of 'input features', each corresponding to the R, G and B value of each pixel, which thus requires a lot of parameters.
2. A cat in the top left or a cat in the bottom right of the image should give similar outputs.

At this point, perhaps we can consider the following method. Suppose we have an image we want to test:

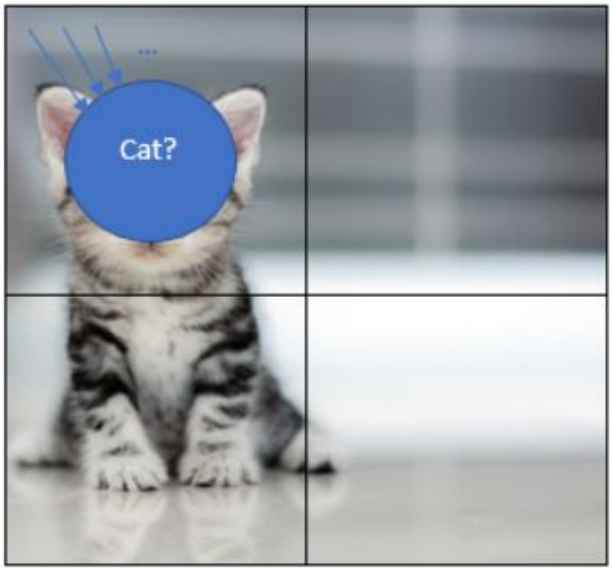
An image of size 256x256 we wish to test whether there is a cat or not

Here is our algorithm:

Step 1: Split the image into four equal quadrants. Let's take the image size to originally be 256 * 256 * 3(channels). Then, each quadrant of the image will have 128 * 128 * 3 features.

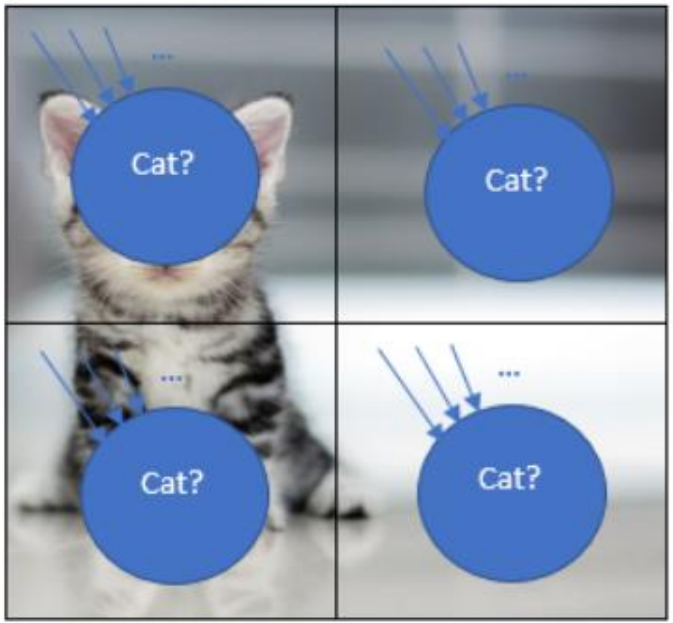


Step 2: Apply a neuron for the top-left quadrant to convert the 128 * 128 * 3 features into one single number. Just for intuition's sake (although this is not entirely accurate), let's say this neuron is in charge of recognizing a cat within the 128 * 128 * 3 features:



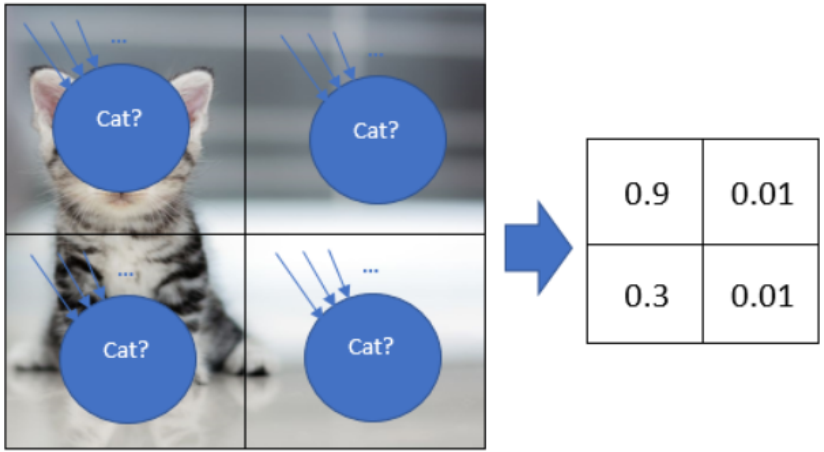
We apply the neuron which takes in the 128 * 128 * 3 features in the top-left quadrant

Step 3: Apply the exact same neuron for the top-right quadrant, the bottom-left quadrant and the bottom-right quadrant. This is called *parameter sharing*, since we use the exact same neuron for all four quadrants.



We apply the exact same neuron with the exact same parameters to all four quadrants of the image

Step 4: After applying that neuron for all four quadrants, we have four different numbers (intuitively speaking, these numbers represent whether there is a cat or not in each of the quadrants).



We get four different output numbers even though we apply the same neuron since the inputs are different (although the parameters are the same)

Remember that we get four different numbers because we put different input features, even though the function (and the parameters remain the same).

What does this algorithm do in terms of addressing our earlier concerns?

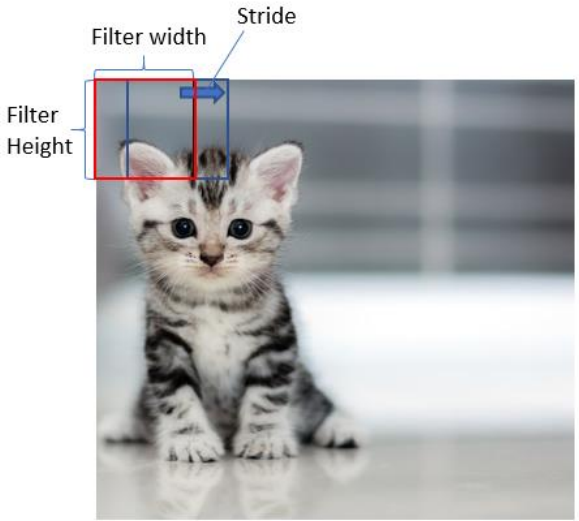
- There were too many features and therefore too many parameters. We'd need 256 * 256 * 3 + 1 = 196,609 parameters for each neuron. If we split this into four different quadrants and use the exact same parameters for all four quadrants, we only need 128 * 128 * 3 + 1 = 49,153 parameters — a reduction by almost four times!

- It doesn't matter where the cat is in the image, all it matters is that there is a cat in the image. By using the same neuron for recognizing a cat in all four quadrants, we address this issue since the 'cat-recognizing neuron' should tell us which quadrant has a cat!

A Convolution layer has these few hyper-parameters that we can specify:

- ***Filter size.*** This corresponds to how many input features in the width and height dimensions one neuron takes in. In our earlier example, the filter size was 128 * 128 because each neuron looked at 128 * 128 pixels spatially (width and height). We always assume that we do not split up the image by its depth (or the channels), only the width and height. So if we specify the filter size, the number of parameters in our neuron is filter_width * filter_height * input_depth + 1. In our example, the number of parameters are 128 * 128 * 3 + 1 = 49,153. Typically though, a reasonable filter size might be more along the order of 3 * 3 or 5 * 5.

- ***Stride.*** Sometimes a cat doesn't appear nicely in the quadrants but might appear somewhere in the middle of two (or more) quadrants. In that case, perhaps we should apply our neuron not just exclusively in the four quadrants, but we want to apply the neuron in overlapping regions as well. Stride is simply how many pixels we want to move (towards the right/down direction) when we apply the neuron again. In our earlier example, we moved with stride 128 so we went to the next quadrant immediately without visiting any overlapping region. More commonly, we typically move with stride 1 or 2.



A convolution layer with filter width, filter height and stride. Note that while the red and blue boxes look at different areas, they use the same parameters.

- ***Depth.*** In our earlier example, we applied just one neuron to identify whether there was a cat or not and share the parameters by applying the same neuron in each quadrant. Suppose we wanted another neuron to identify whether there was a dog or not as well. This neuron would be applied in the same way as the cat-identifying neuron, but have different parameters and therefore a different output for each quadrant. How would this change our parameter and output size? Well, if we had two such neurons, we'd have (128 * 128 * 3 + 1) * 2 = 98,306 parameters. And at the end of Step 4, we'll have 2 * 2 * 2 = 8 output numbers. The first two terms, 2 * 2, refers to the height and width (of our four quadrant areas) and the last term, 2, refers to the fact that we had two different neurons applied to each quadrant. This last term is what we call **depth**.

QUESTION: Suppose we have an image of input size 256 * 256 * 3. We apply a conv layer with filter size 3 * 3, stride 1, and depth 64.

- How many parameters do we have in our conv layer?
- What are the output dimensions of this conv layer?

Answer:

Number of parameters

We work out the case for depth = 1, since that's just one neuron applied throughout. This neuron takes in 3 * 3 * 3 (filter size * input channels) features, and so the number of parameters for this one neuron are 3 * 3 * 3 + 1 = 28. We know that depth = 64, meaning there are 64 such neurons. This gives us a total of 28 * 64 = 1,792 parameters.

Answer:

Output dimensions

Let's think of it in the dimension of width first. We have a row of 256 pixels in our original input image. At the start, the center of our filter (what the neuron takes as input) will be at pixel 2, since we have a 3 * 3 filter. Thus, since the leftmost side of the filter will be at pixel 1, the center of the filter will be at pixel 2.

This filter moves rightwards by 1 pixel at each time to apply the neuron(s). At the end of all our steps, the center of our filter will be at pixel 255, again because we have a 3 * 3 filter (so pixel 256 will be taken up by the rightmost side of the filter).

So given that the center of our filter starts at pixel 2 and ends at 255 while moving 1 pixel each step, the math suggests that we've applied the neuron 254 times across the width. Similarly, we've applied the neuron 254 times across the height. And since we have 64 neurons doing that (depth = 64), our output dimensions are 254 * 254 * 64.

At this point, you might be wondering: well, what if I wanted output dimensions of 256 * 256 * 64 so that the height and width of our output remains the same as the input dimensions? Here, I will introduce a new concept to deal with that exactly:

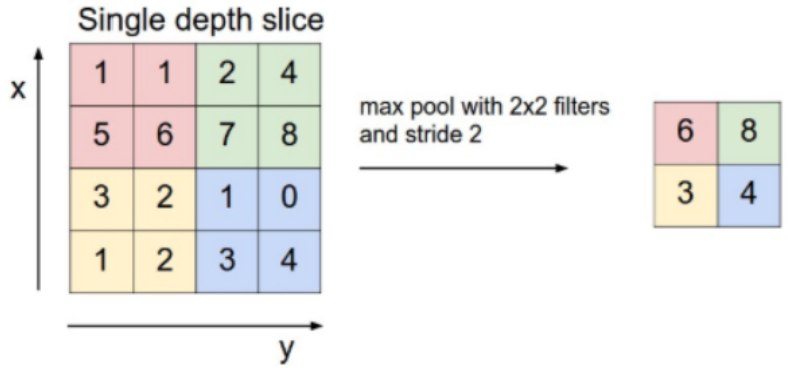
- **Padding.** Recall that the center of the 3x3 filter started at pixel 2 (instead of at pixel 1) and ended at pixel 255 (instead of at pixel 256). To make the center the filter start at pixel 1, we can pad the image with a border of '0's, like this:

0	0	0	0	0
0	12	34	15	0
0	32	63	1	0
0	8	77	98	0
0	0	0	0	0

This is an example of padding a 3x3 image with a padding border of size 1. This ensures that the center of a 3x3 filter will begin at pixel 1 of the original image.

Step 4 & 5 Pooling

More generally, a pooling layer has a filter size and a stride, similar to a convolution layer. Let's take the simple example of an input with depth 1 (i.e. it only has 1 depth slice). If we apply a max-pool with filter size 2x2 and stride 2, so there is no overlapping region, we get:



Max-pooling with filter 2 and stride 2. Note that a max-pool layer of filter 2 and stride 2 is commonly seen in many models today. Image taken from CS231N notes: http://cs231n.github.io/convolutional-networks/

This max-pool seems very similar to a conv layer, except that there are no parameters (since it just takes the maximum of the four numbers it sees within the filter). When we introduce depth, however, we see more differences between the pooling layer and the conv layer.

Pooling Layer

- The pooling layer applies to each individual depth channel separately.
- That is, the max-pooling operation does not take the maximum across the different depths; it only takes the maximum in a single depth channel.
- This is unlike the conv layer, which combines inputs from all the depth channels.
- This also means that the depth size of our output layer does not and cannot change, unlike the conv layer where the output depth might be different from input depth.
- The purpose of the pooling layer, ultimately, is to reduce the spatial size (width and height) of the layers and it does not touch on the depth at all.
- This reduces the number of parameters (and thus computation) required in future layers after this pooling layer

QUESTION: Suppose after our first conv layer (with pooling), we have an output dimension of 256 * 256 * 64. We now apply a max-pooling (with filter size 2x2 and stride 2) operation to this.

- what are the output dimensions after the max pooling layer?

ANSWER: 128 * 128 * 64, since the max-pool operator reduces the dimensions on the width and height by half, while leaving the depth dimension unchanged.

Fully Connected Layer (FC)

The last layer that commonly appears in CNNs is one that we've seen before in earlier parts — and that is the Fully-Connected (FC) layer. The FC layer is the same as our standard neural network — every neuron in the next layer takes as input every neuron in the previous layer's output. Hence, the name Fully Connected, since all neurons in the next layer are always connected to all the neurons in the previous layer. To show a familiar diagram we've seen in Part 1a:

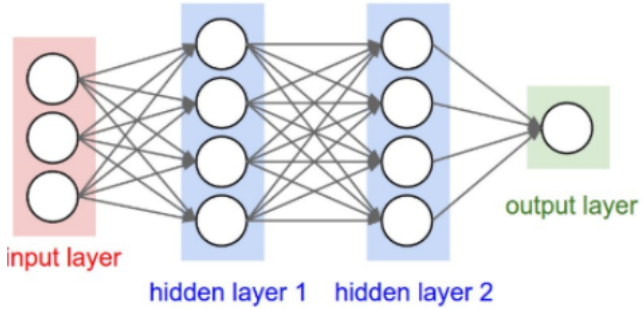


Image taken from CS231N Notes (http://cs231n.github.io/neural-networks-1/)

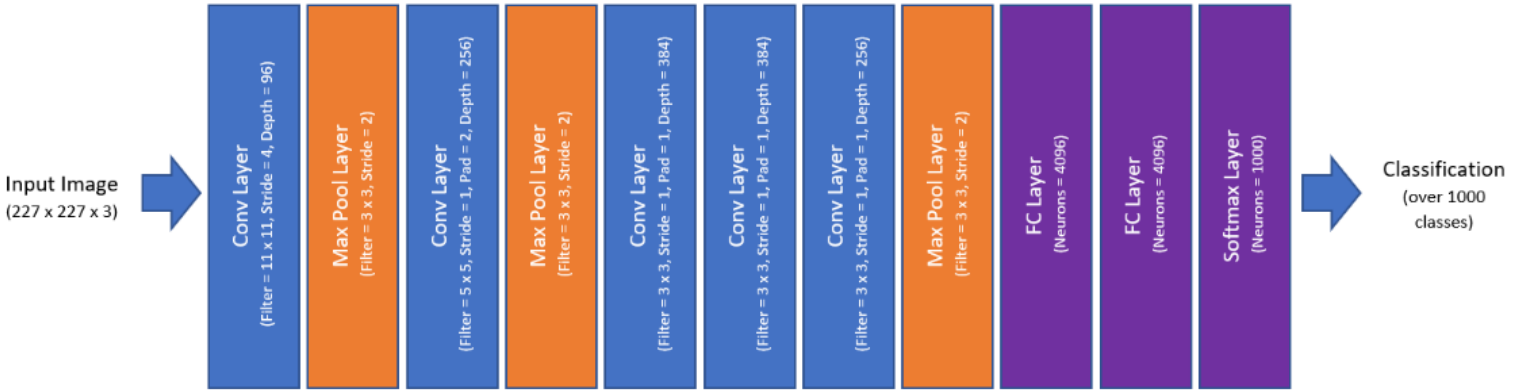
We usually use FC layers at the very end of our CNNs. So when we reach this stage, we can flatten the neurons into a one-dimensional array of features. If the output of the previous layer was 7 * 7 * 5, we can flatten them into a row of 7*7*5 = 245 features as our input layer in the above diagram. Then, we apply the hidden layers as per usual.

- One important benchmark that is commonly used amongst researchers in Computer Vision is this challenge called ImageNet Large Scale Visual Recognition Challenge (ILSVRC).

- ImageNet refers to a huge database of images, and the challenge of ILSVRC is to accurately classify an input image into 1,000 separate object categories.

- One of the models that was hailed at the turning point in using deep learning is **AlexNet**, which won the ILSVRC in 2012

AlexNet's architecture can be summarized somewhat as follows:



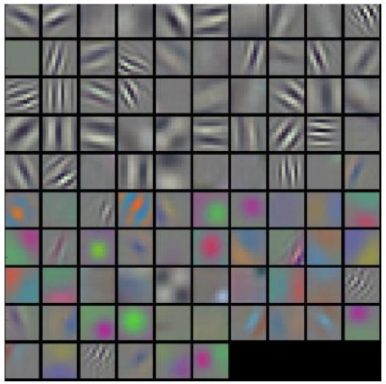
A simplistic view of the AlexNet Architecture, where some details have been omitted

As you can see, AlexNet is simply made out of the building blocks of:

- Conv Layers (with ReLU acitvations)
- Max Pool Layers
- FC Layers
- Softmax Layers

- Why does stacking so many layers together work, and what is each layer really doing?

We can visualize some of the intermediate layers. This is a visualization of the first conv layer of AlexNet:

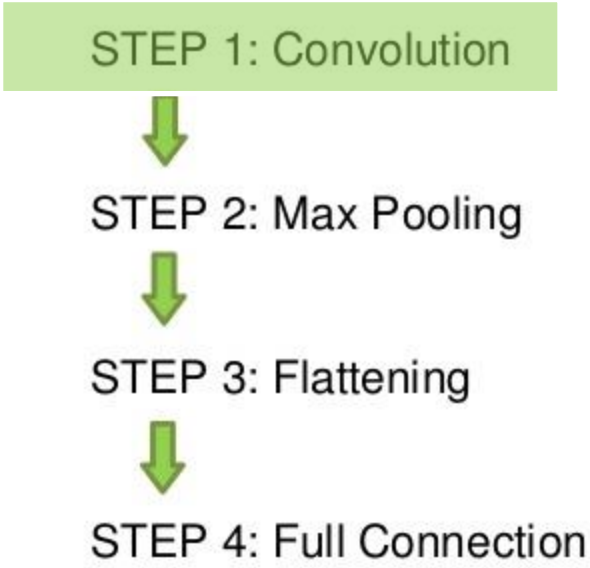


A visualization of the first conv layer in AlexNet. Image taken from CS231N notes: http://cs231n.github.io/understanding-cnn/

- ✓ In the first few layers, the neural network tries to extract out some low-level features. These first few layers then combine in subsequent layers to form more and more complex features, and in the end, figure out what represents objects like cats, dogs etc.

- Why did the neural network pick out those features in particular in the first layer?
- ✓ It just figured out that these are the best parameters to characterize the first few layers; they simply produced the minimal loss.

Convolutional Neural Networks



Step 1 – Convolution

Step 1 - Convolution

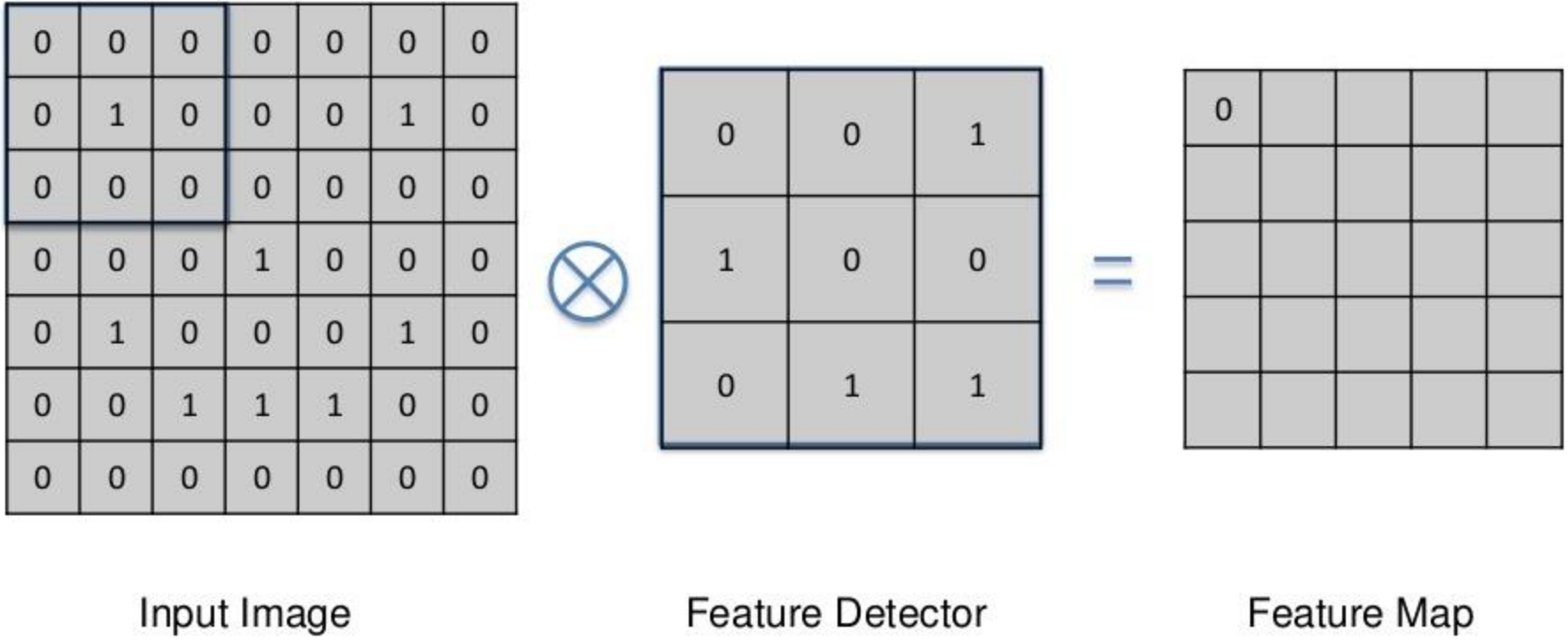
0	0	0	0	0	0	0
0	1	0	0	0	1	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	1	0	0	0	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

Input Image

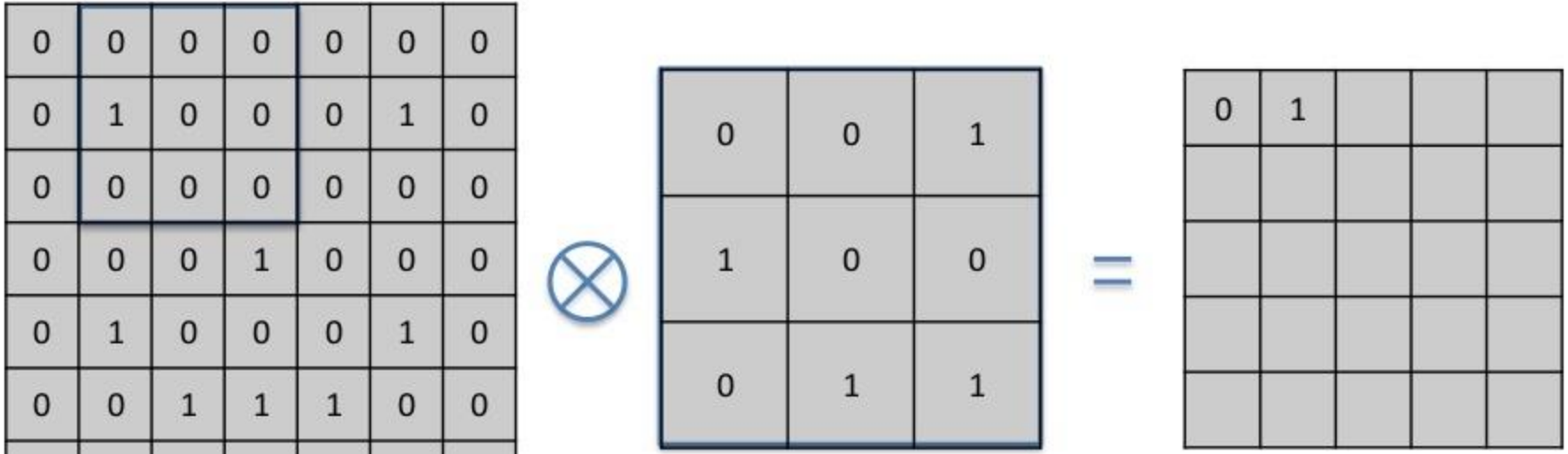
0	0	1
1	0	0
0	1	1

Feature Detector

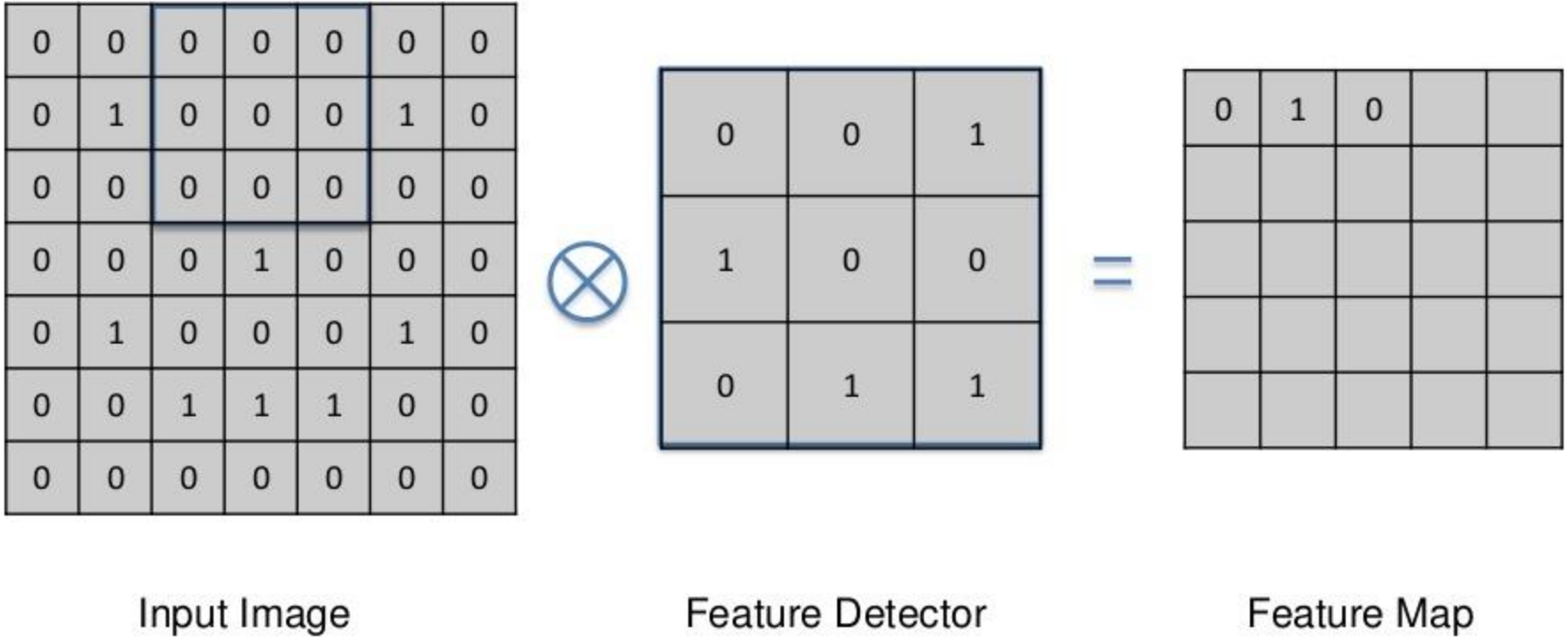
Step 1 - Convolution



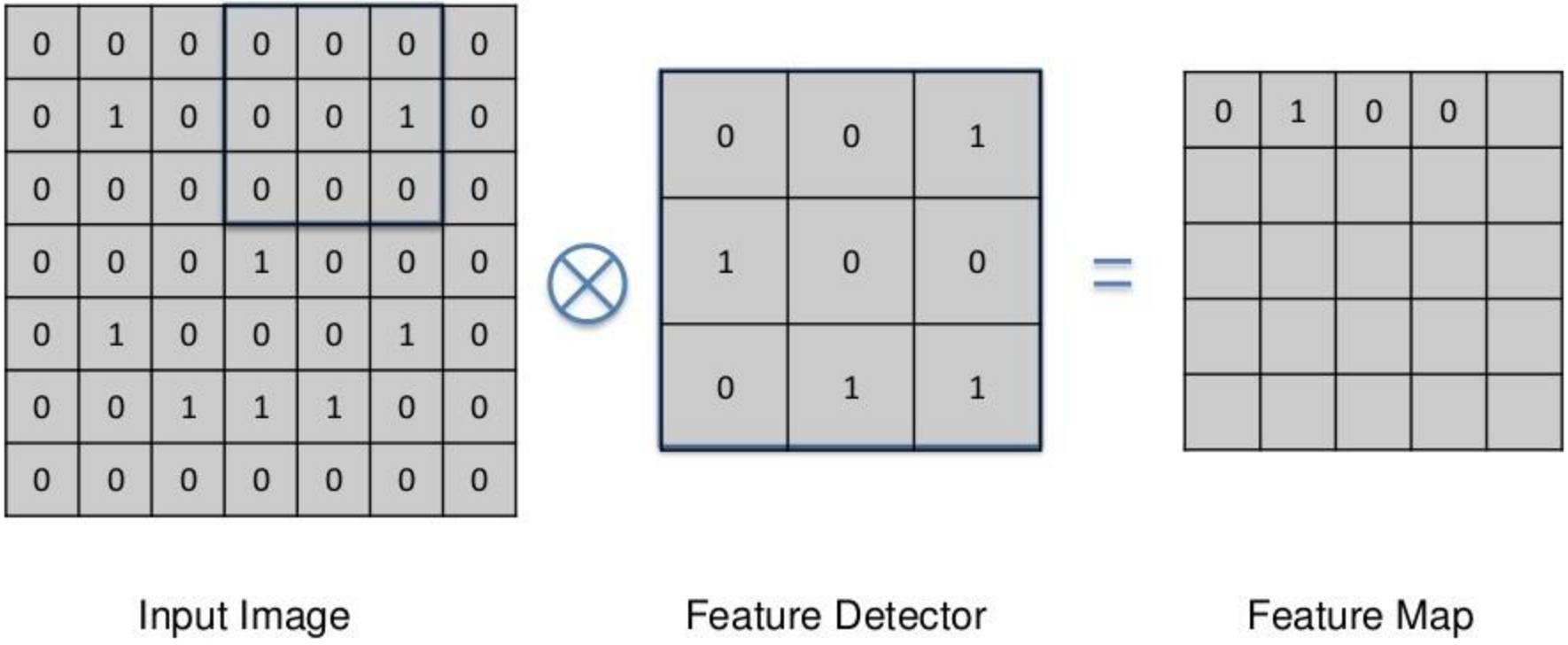
Step 1 - Convolution



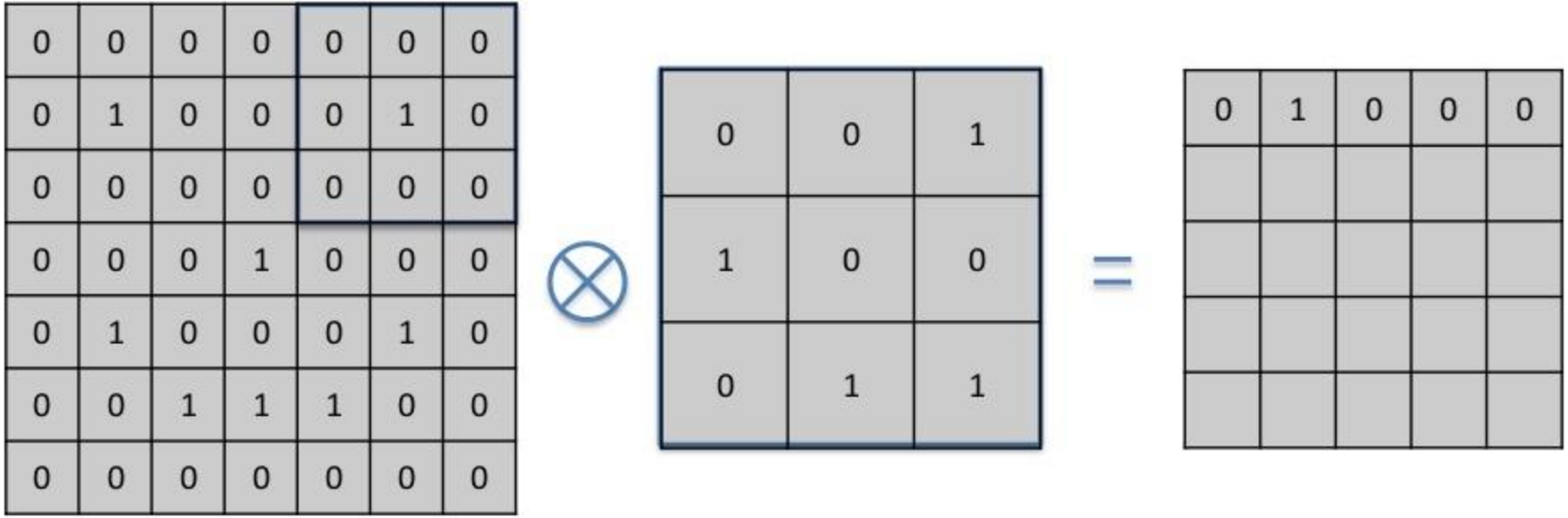
Step 1 - Convolution



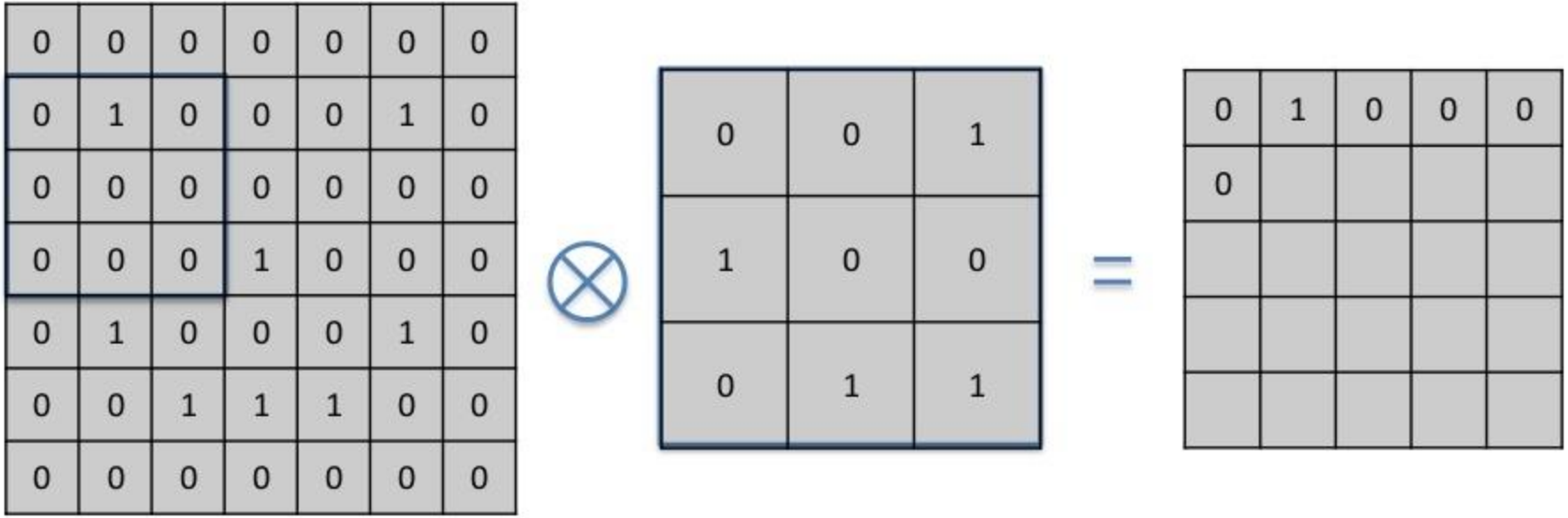
Step 1 - Convolution



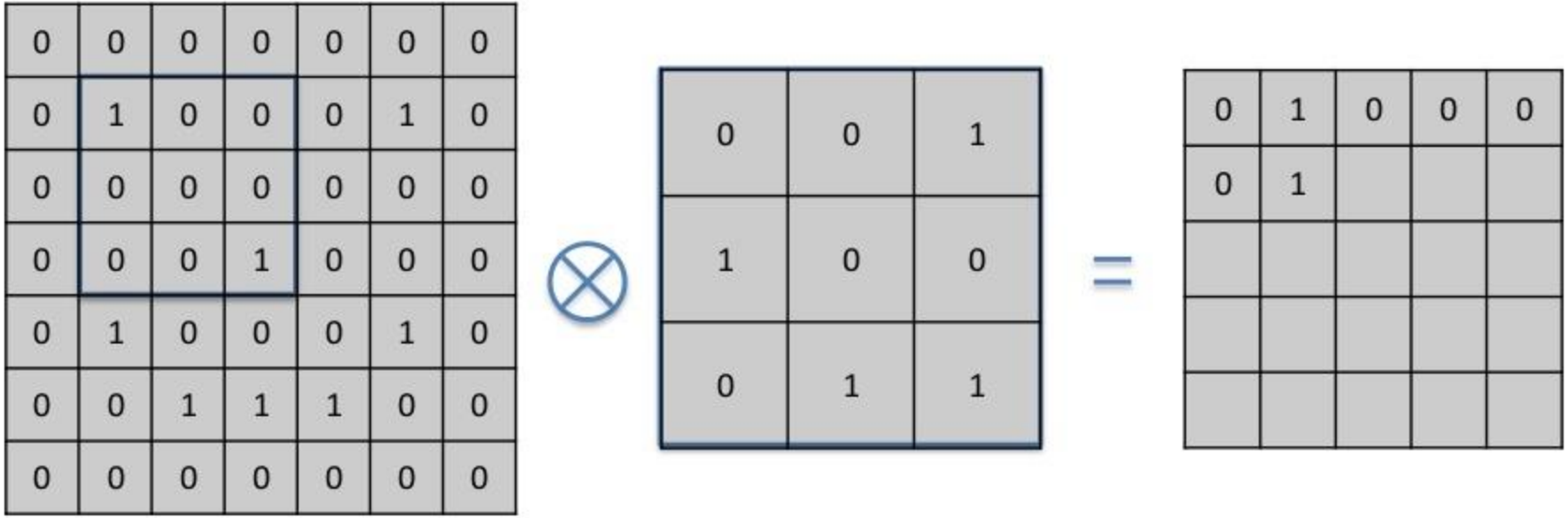
Step 1 - Convolution



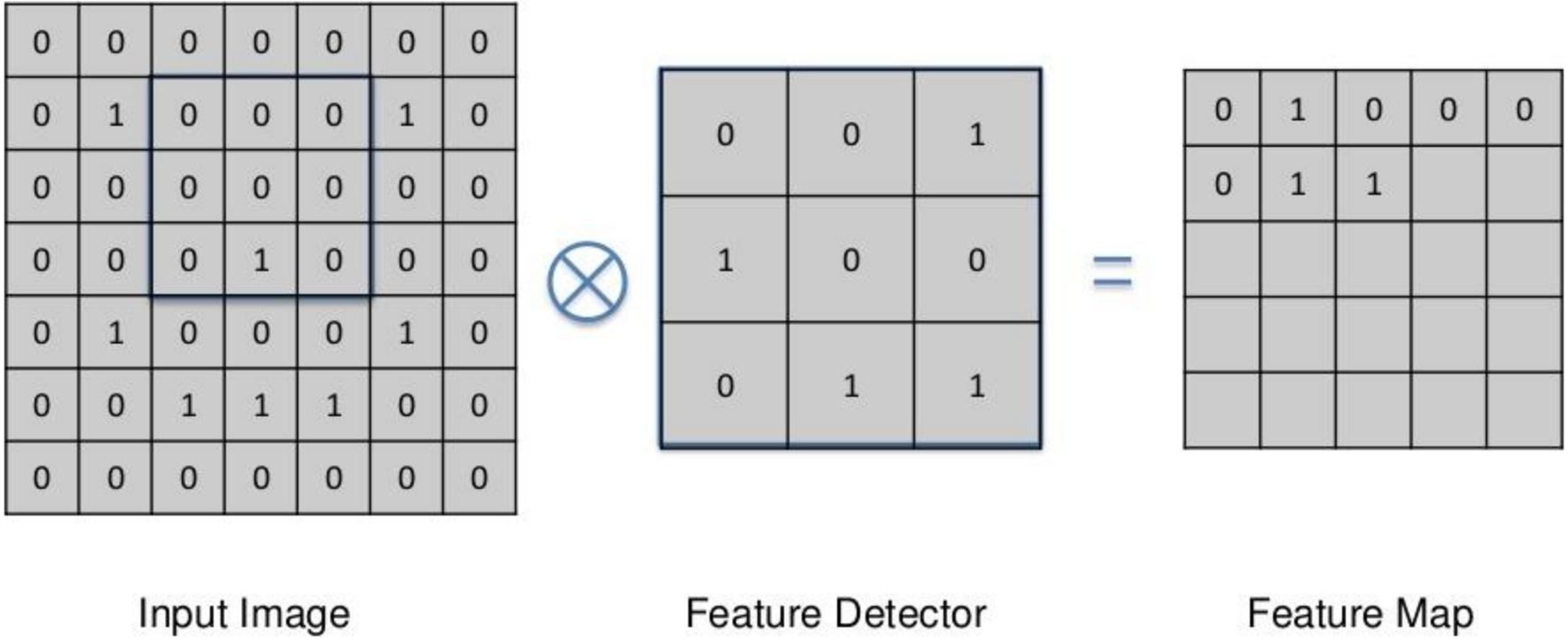
Step 1 - Convolution



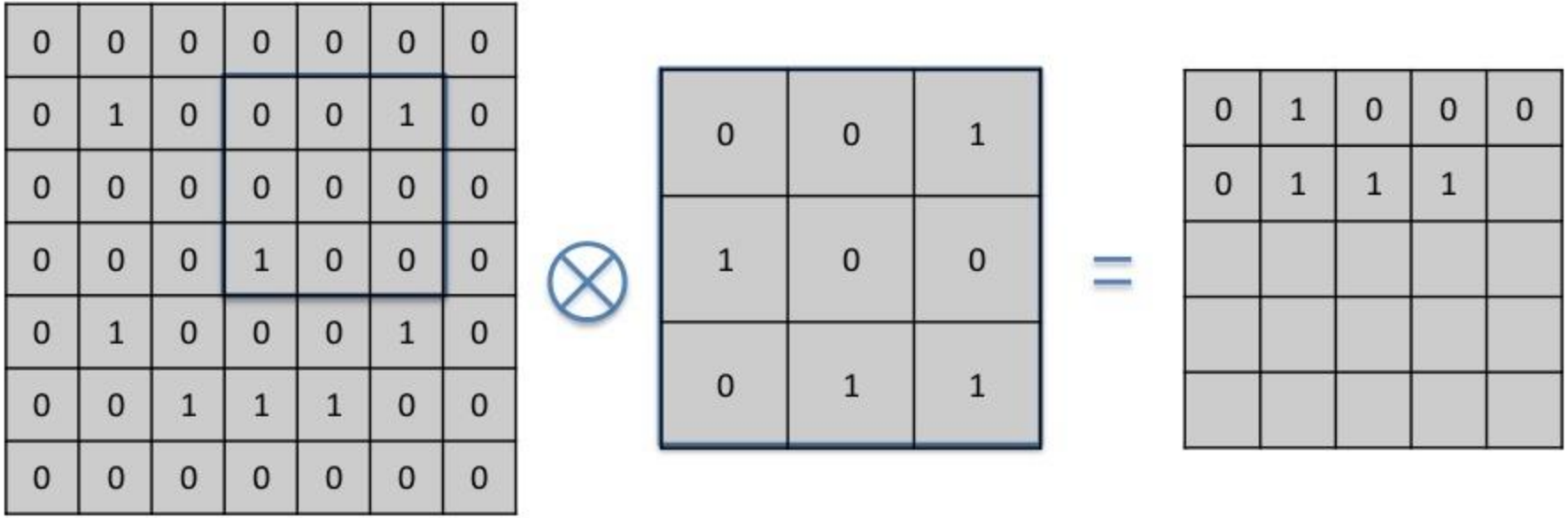
Step 1 - Convolution



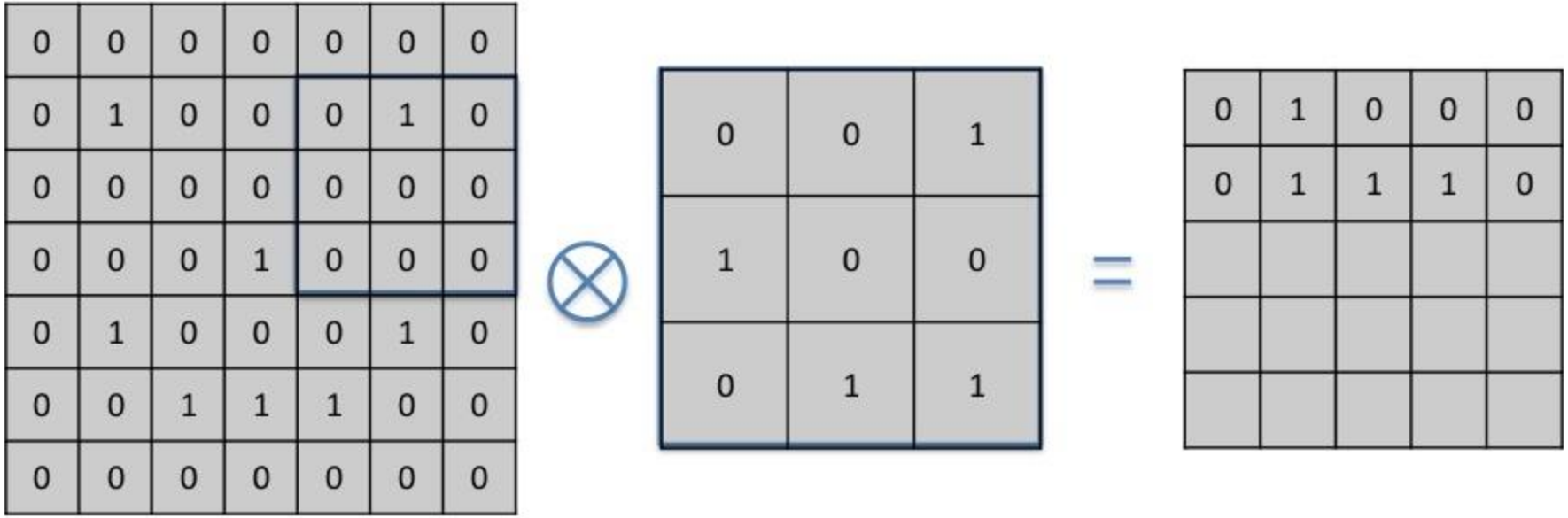
Step 1 - Convolution



Step 1 - Convolution



Step 1 - Convolution

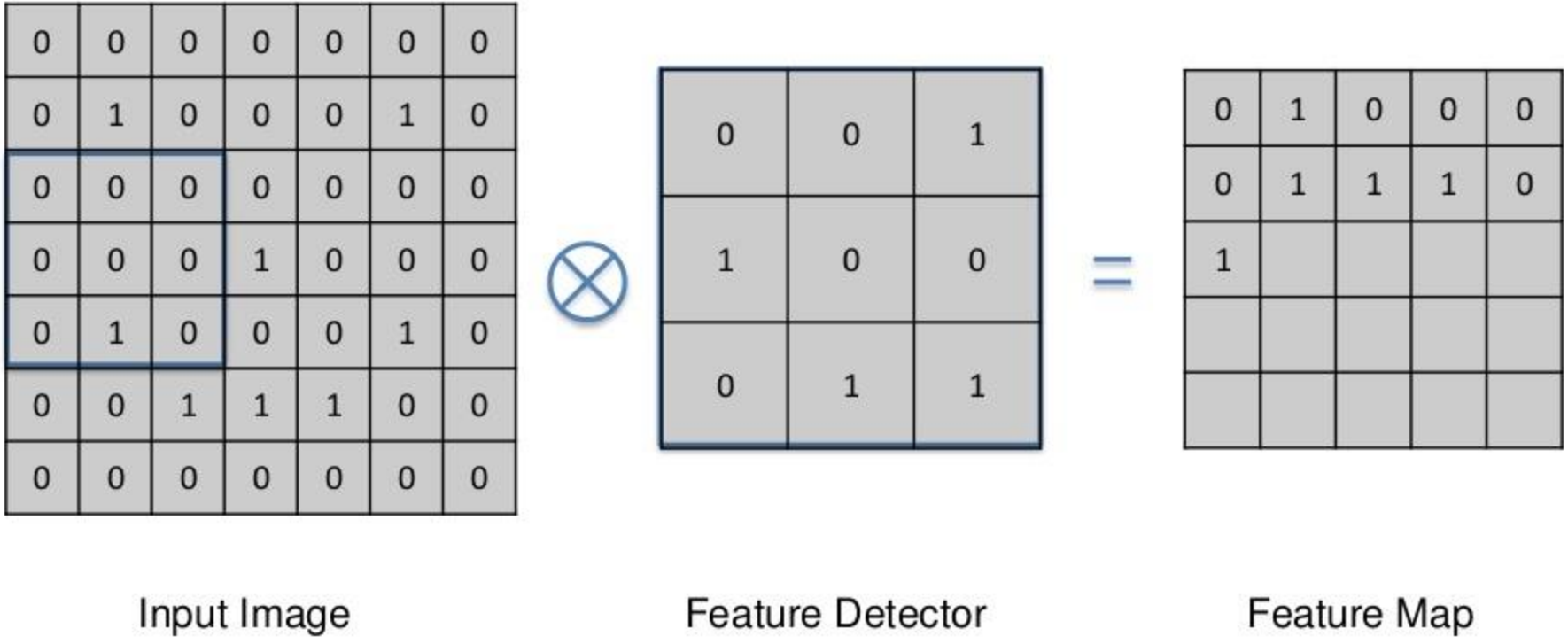


Input Image

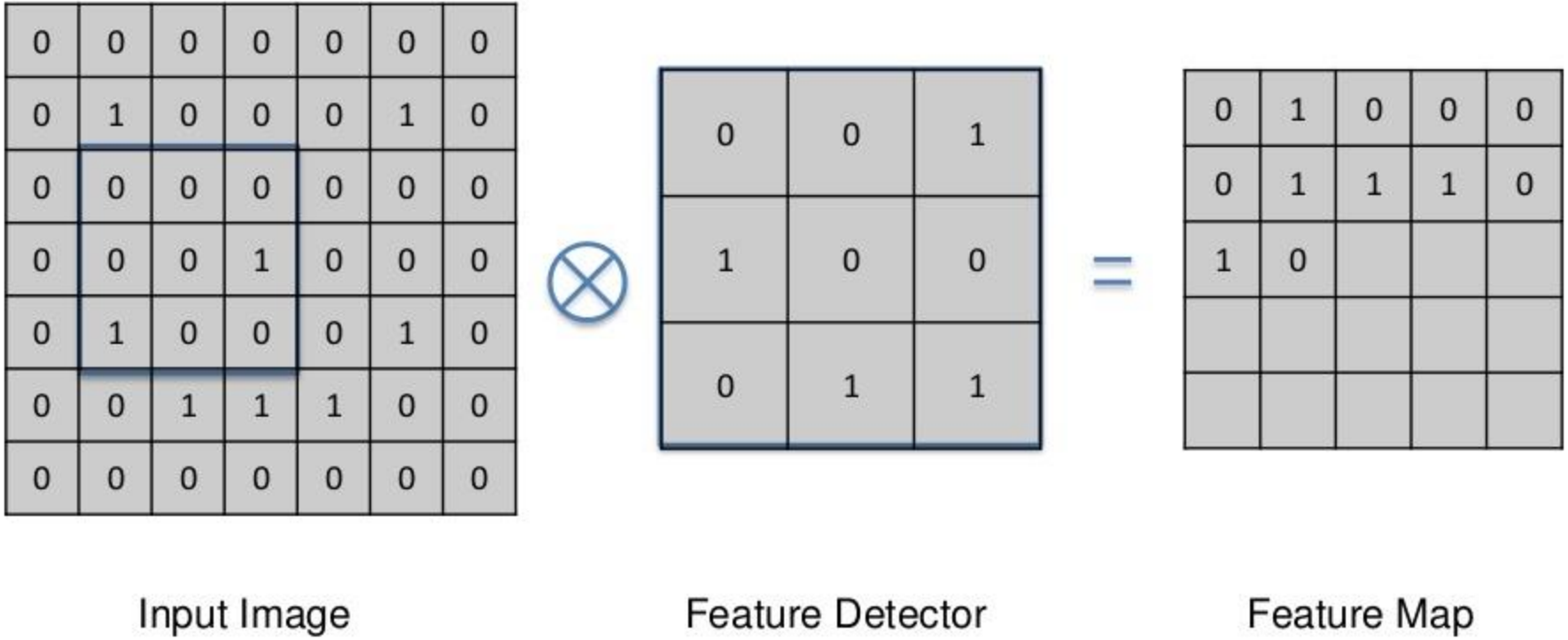
Feature Detector

Feature Map

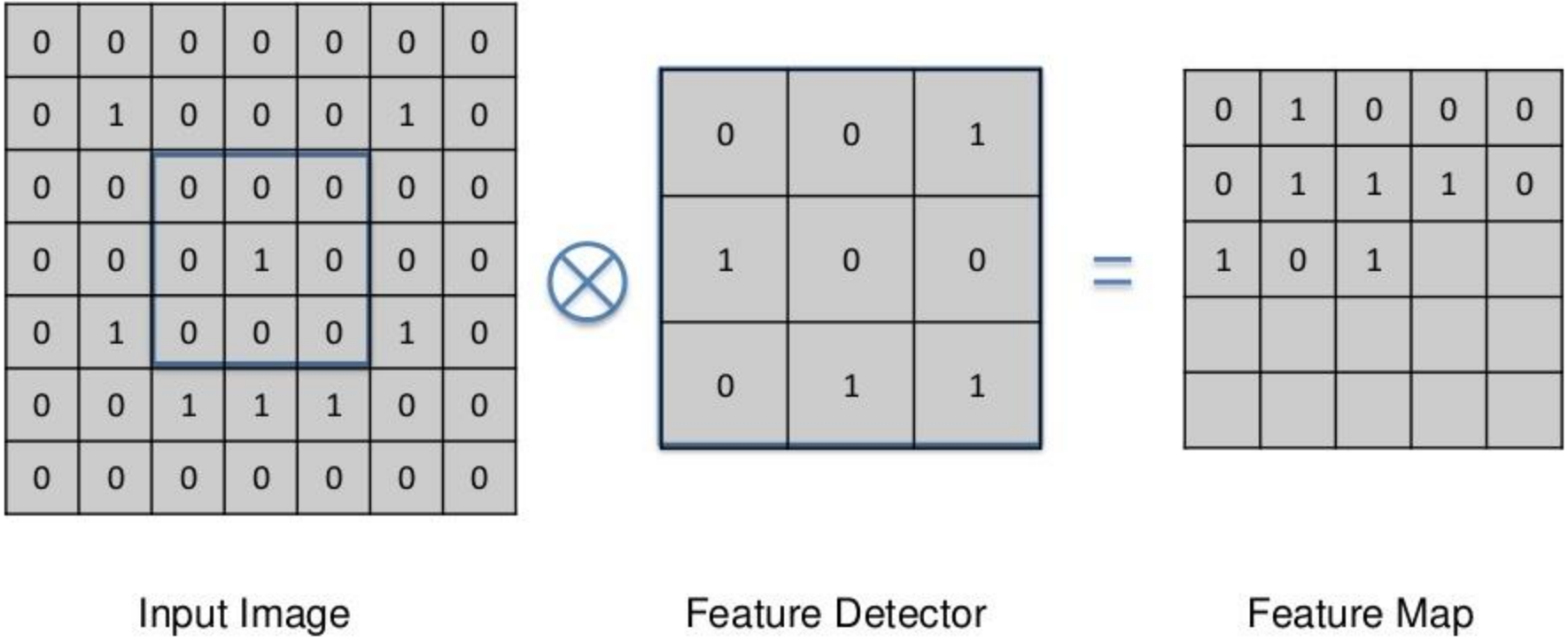
Step 1 - Convolution



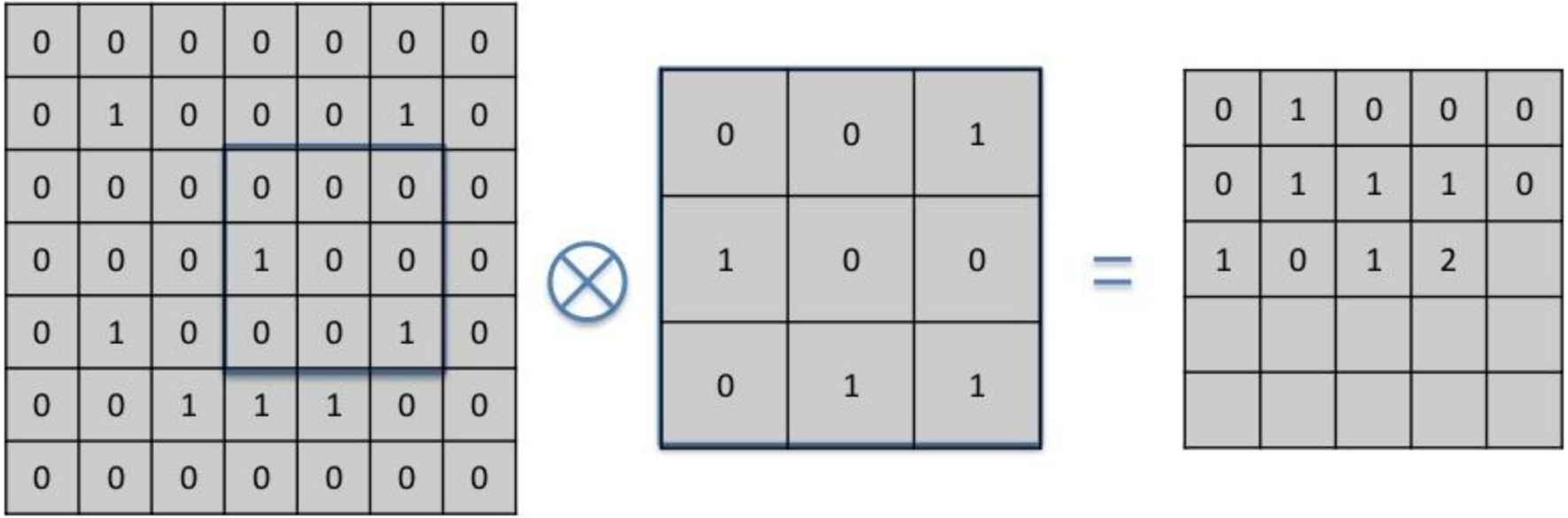
Step 1 - Convolution



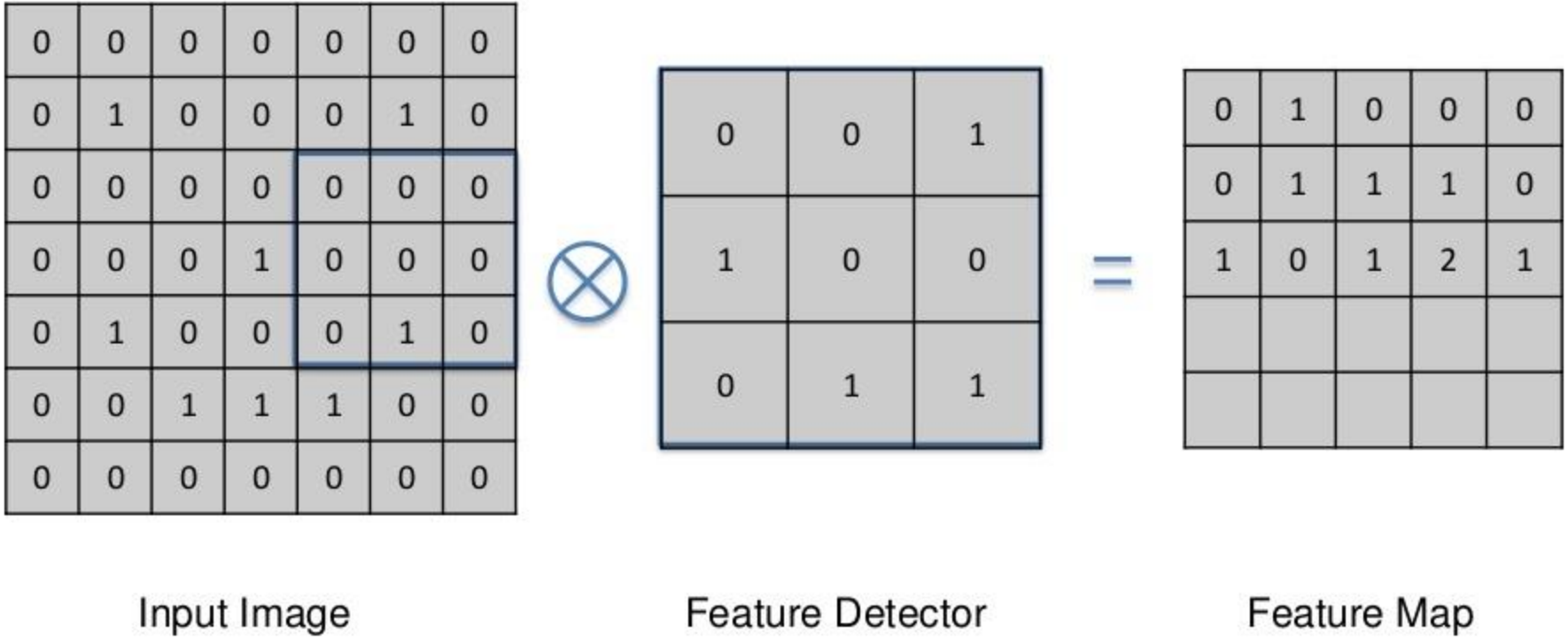
Step 1 - Convolution



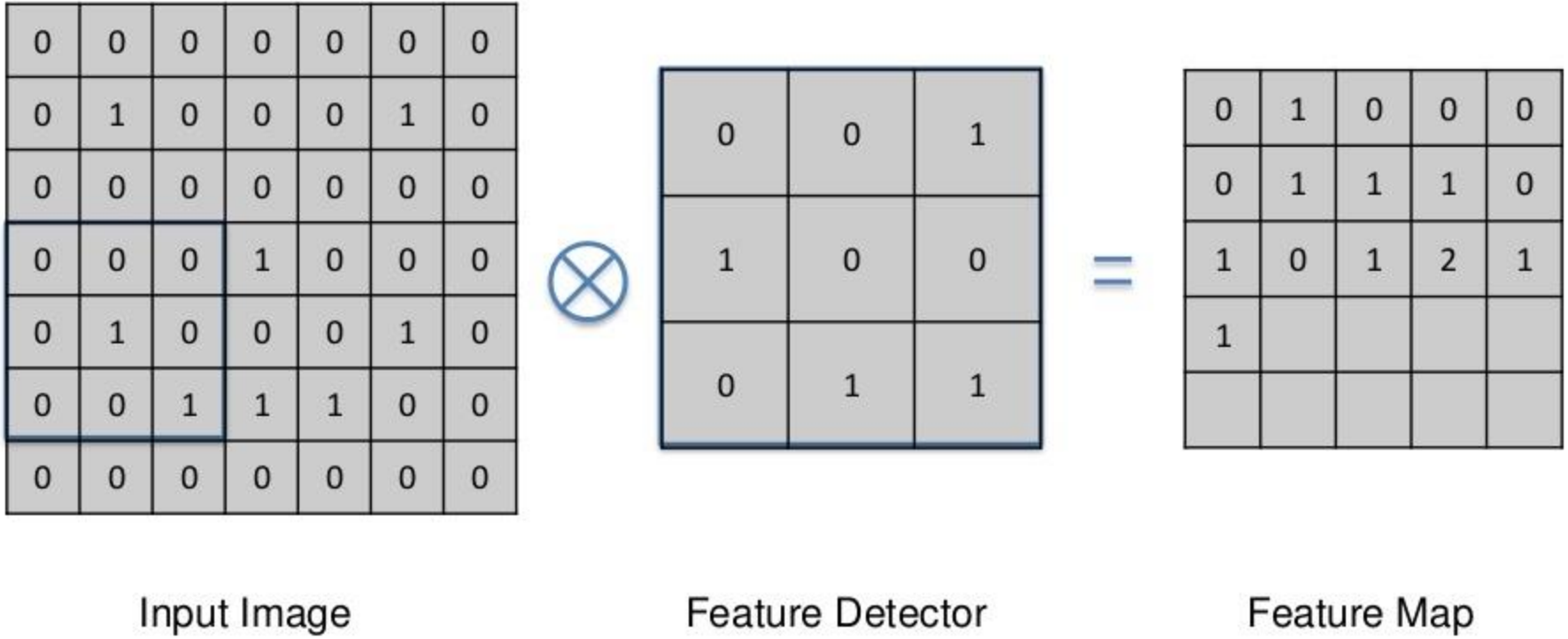
Step 1 - Convolution



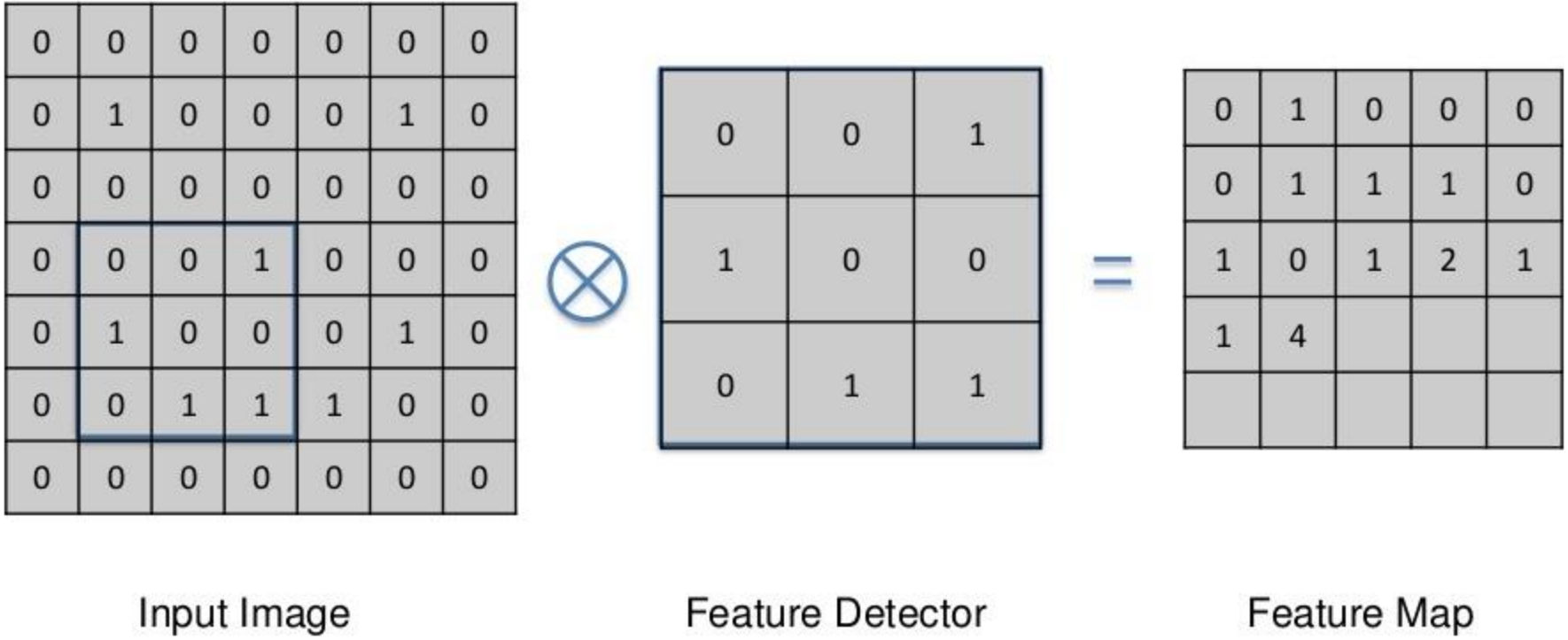
Step 1 - Convolution



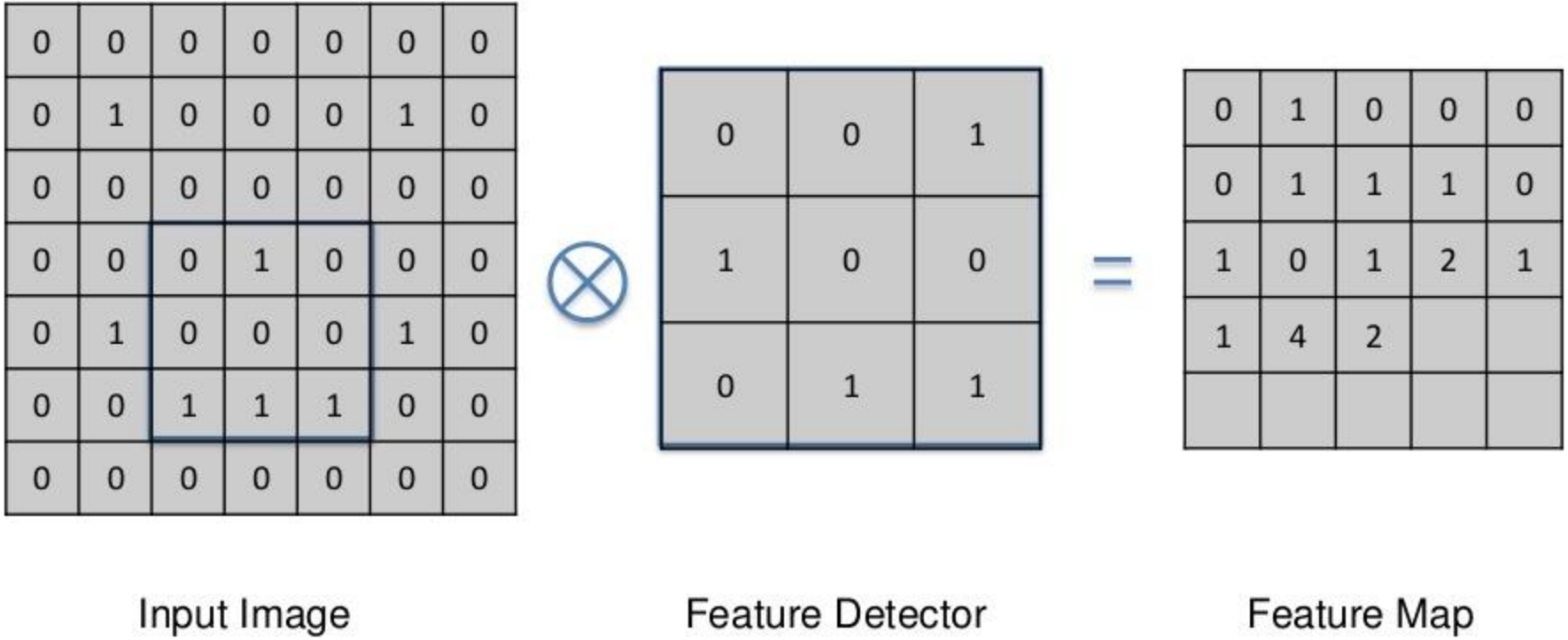
Step 1 - Convolution



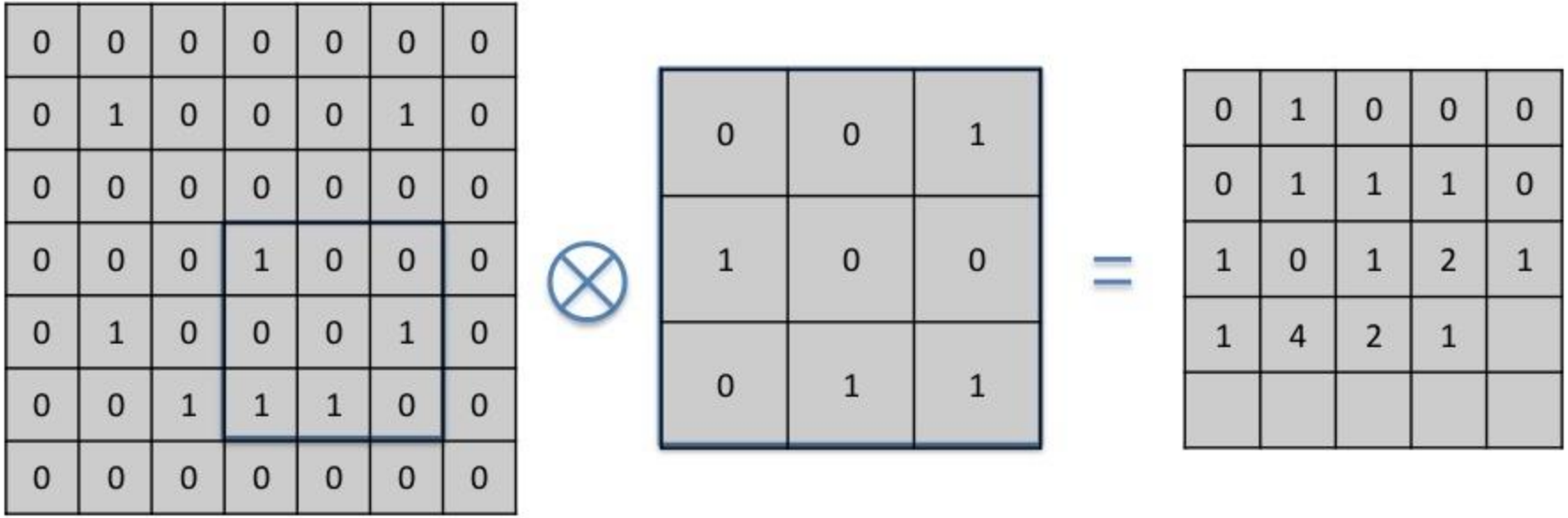
Step 1 - Convolution



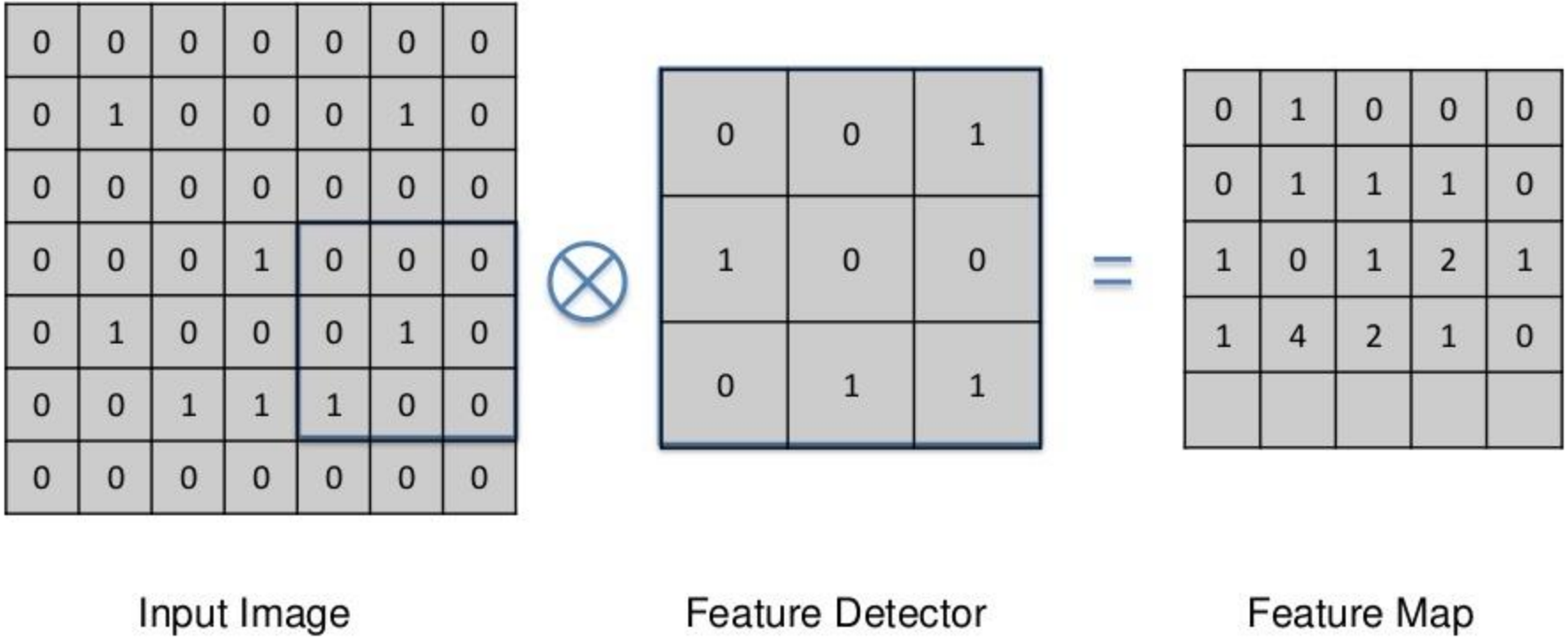
Step 1 - Convolution



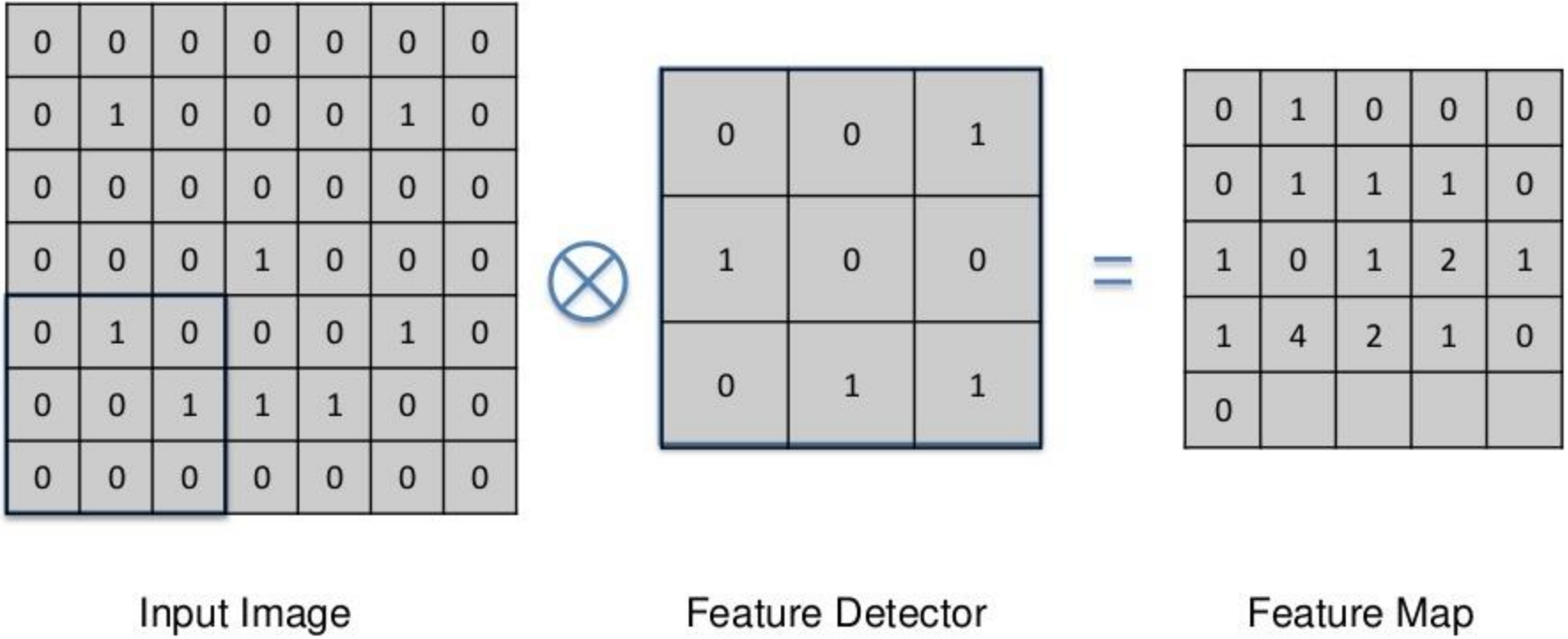
Step 1 - Convolution



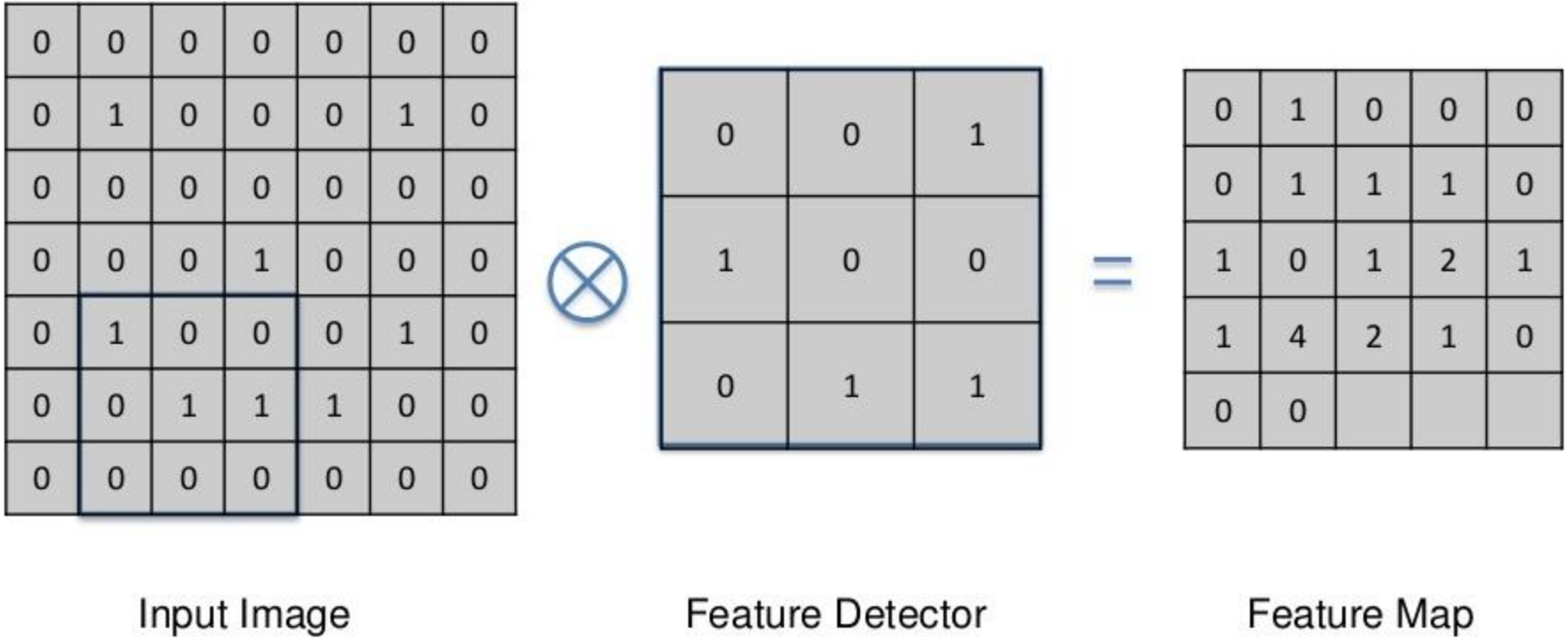
Step 1 - Convolution



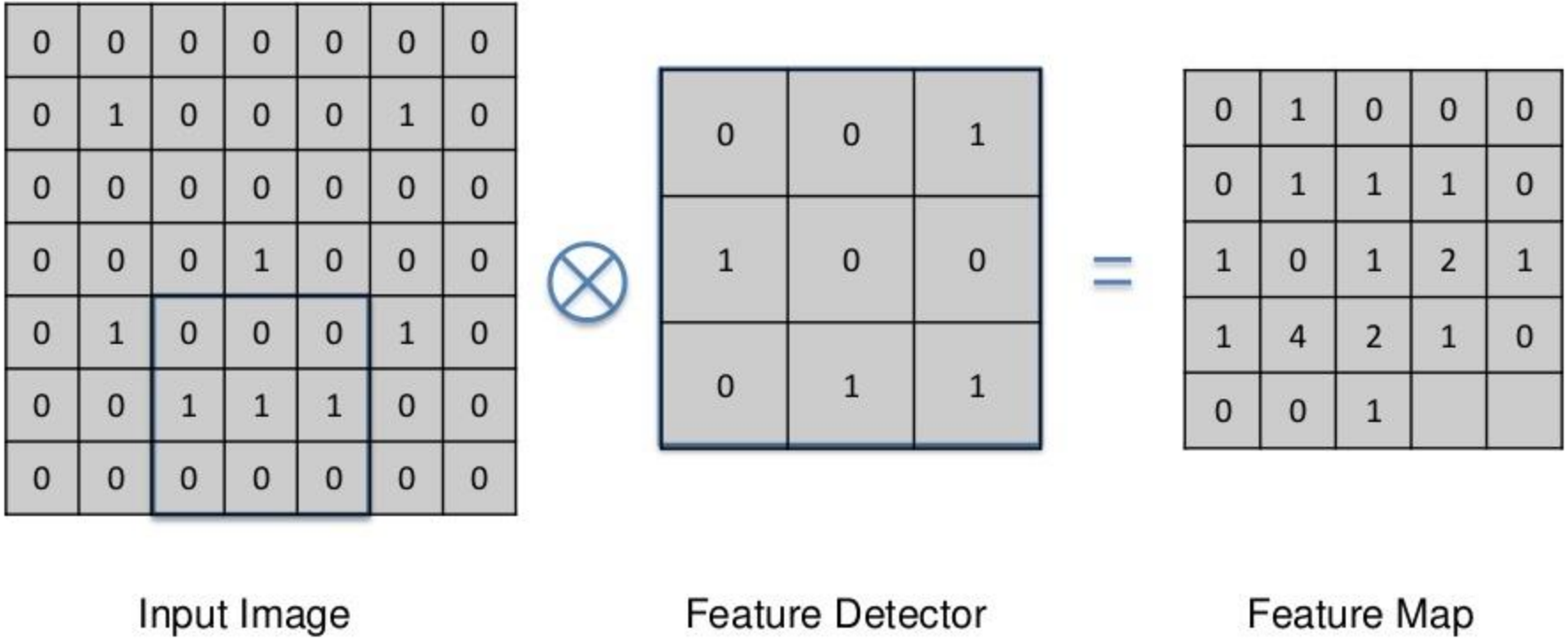
Step 1 - Convolution



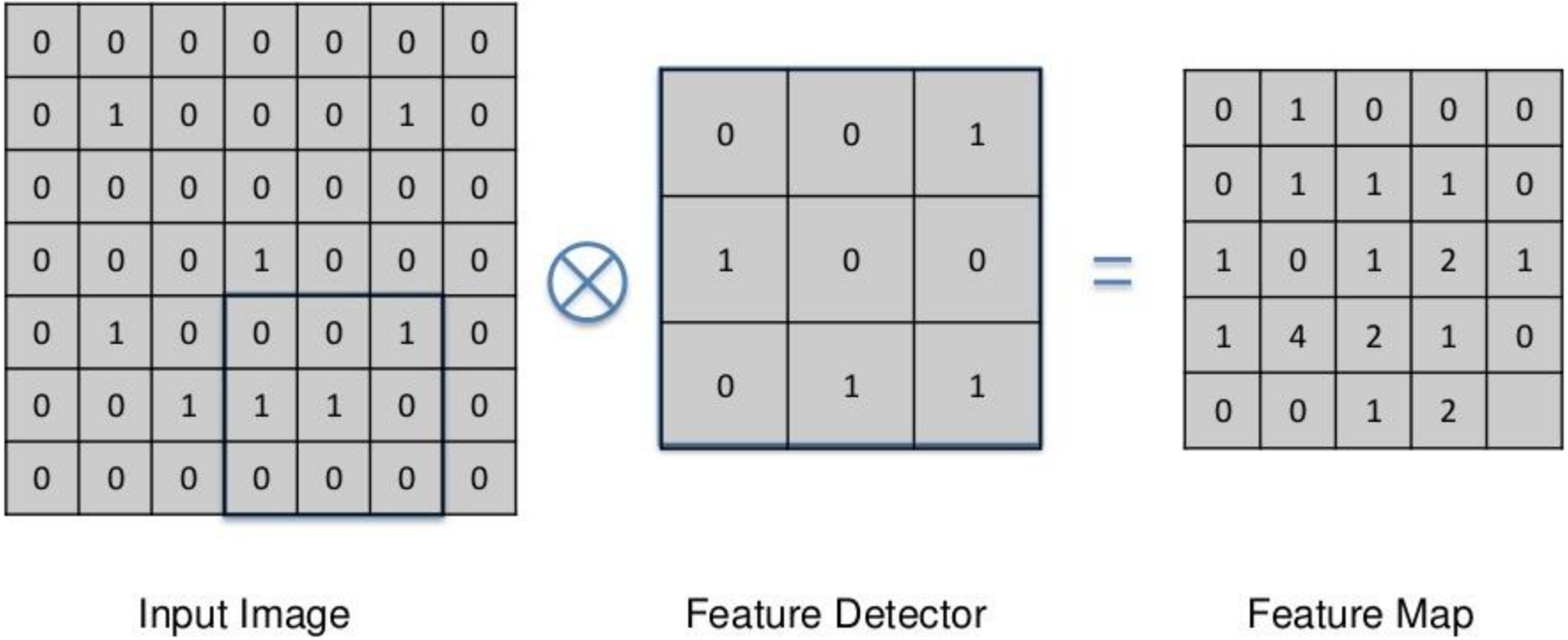
Step 1 - Convolution



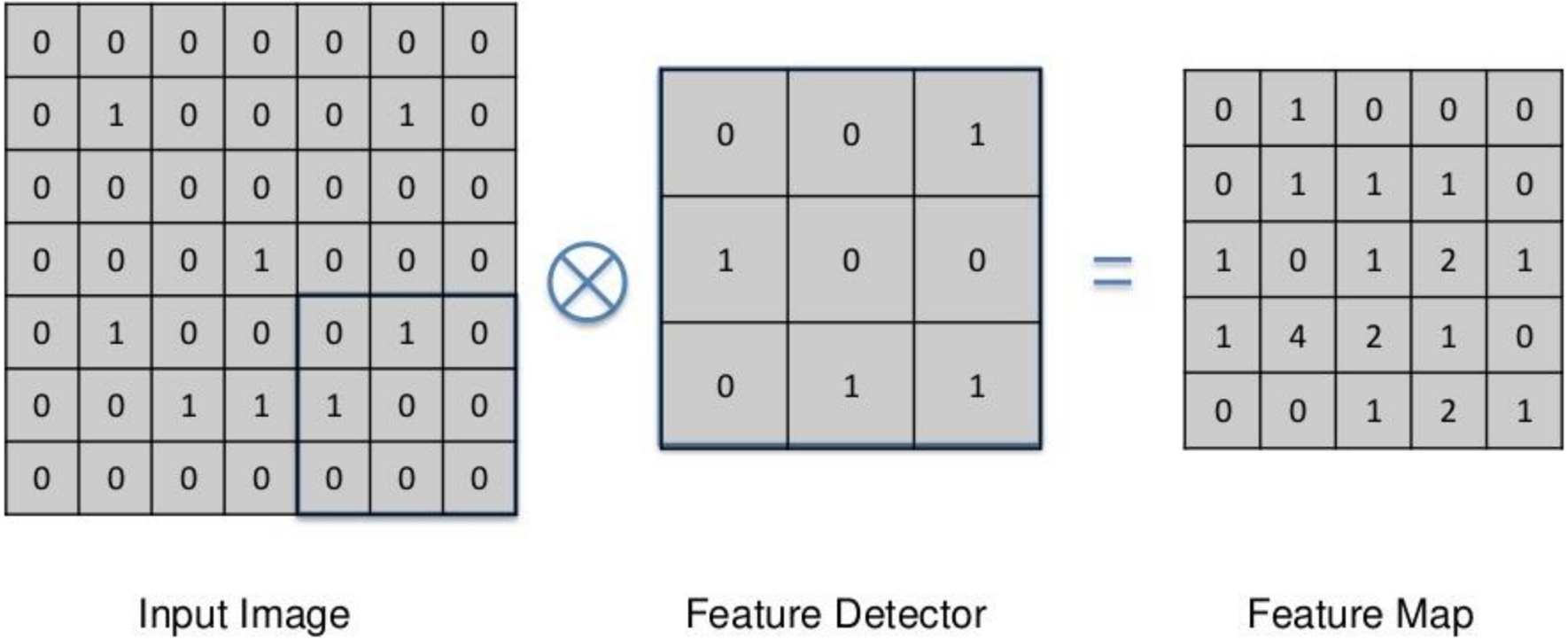
Step 1 - Convolution



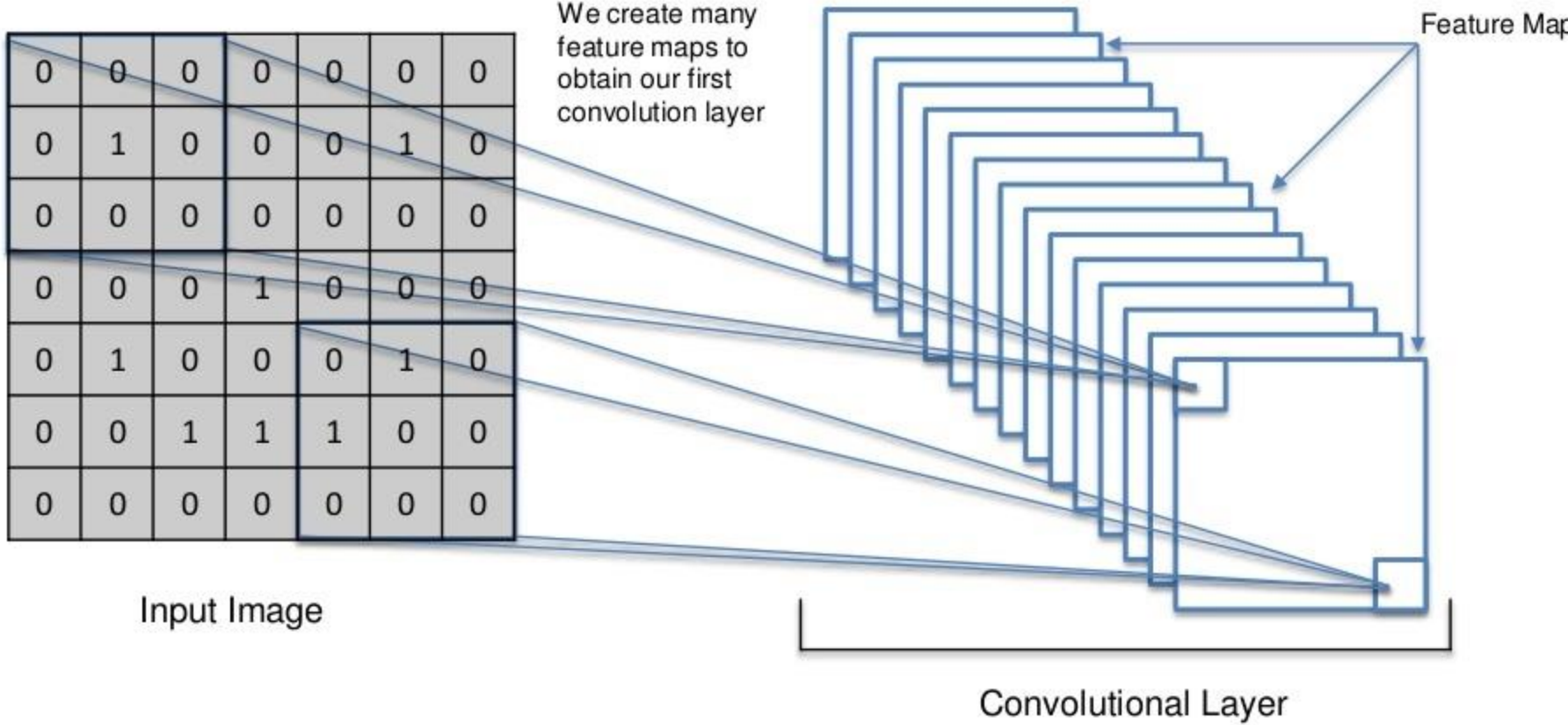
Step 1 - Convolution



Step 1 - Convolution



Step 1 - Convolution



Step 1 - Convolution

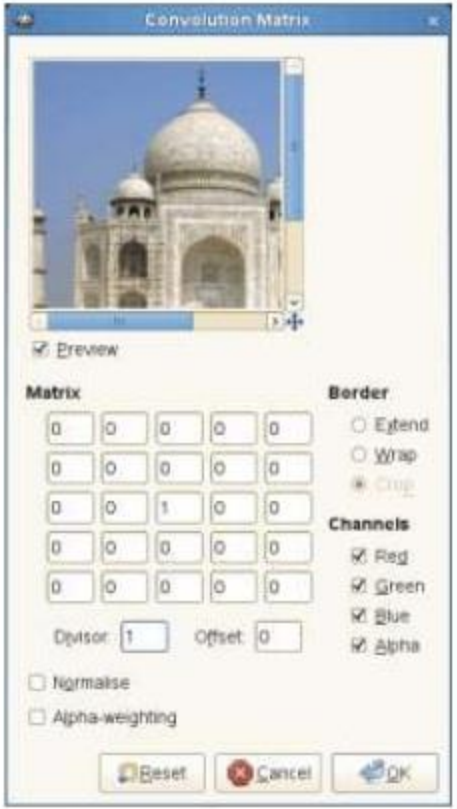


Image Source: docs.gimp.org/en/plug-in-convmatrix

Step 1 - Convolution

Sharpen:

0	0	0	0	0
0	0	-1	0	0
0	-1	5	-1	0
0	0	-1	0	0
0	0	0	0	0

Image Source: docs.gimp.org/en/plug-in-convmatrix

Step 1 - Convolution

Blur:

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

Image Source: docs.gimp.org/en/plug-in-convmatrix

Step 1 - Convolution

Edge Enhance:

	0	0	0	
	-1	1	0	
	0	0	0	

Image Source: docs.gimp.org/en/plug-in-convmatrix

Step 1 - Convolution

Edge Detect:

	0	1	0	
	1	-4	1	
	0	1	0	

Image Source: docs.gimp.org/en/plug-in-convmatrix

Step 1 - Convolution

Emboss:

	-2	-1	0	
	-1	1	1	
	0	1	2	

Image Source: docs.gimp.org/en/plug-in-convmatrix

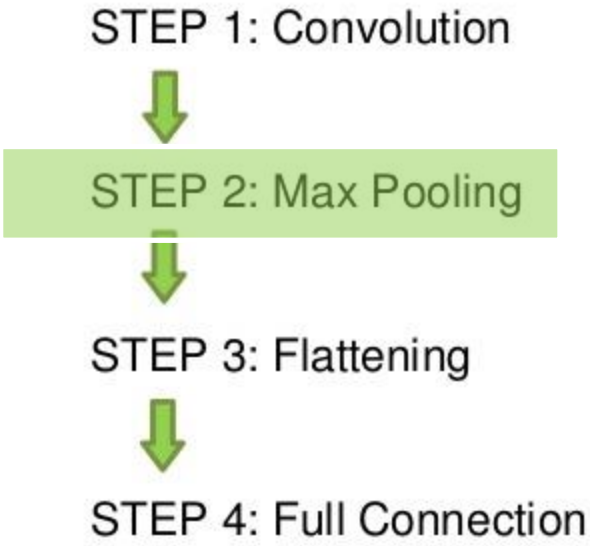
Step 1 - Convolution

$*$

1	0	-1
2	0	-2
1	0	-1

Image Source: eonardoaraujosantos.gitbo

Convolutional Neural Networks



Step 2 – Max Pooling

Step 2 - Max Pooling

0	1	0	0	0
0	1	1	1	0
1	0	1	2	1
1	4	2	1	0
0	0	1	2	1

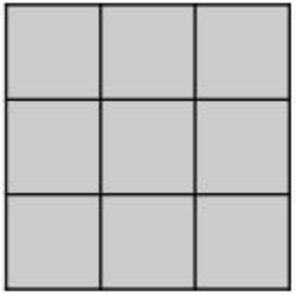
Feature Map

Step 2 - Max Pooling

0	1	0	0	0
0	1	1	1	0
1	0	1	2	1
1	4	2	1	0
0	0	1	2	1

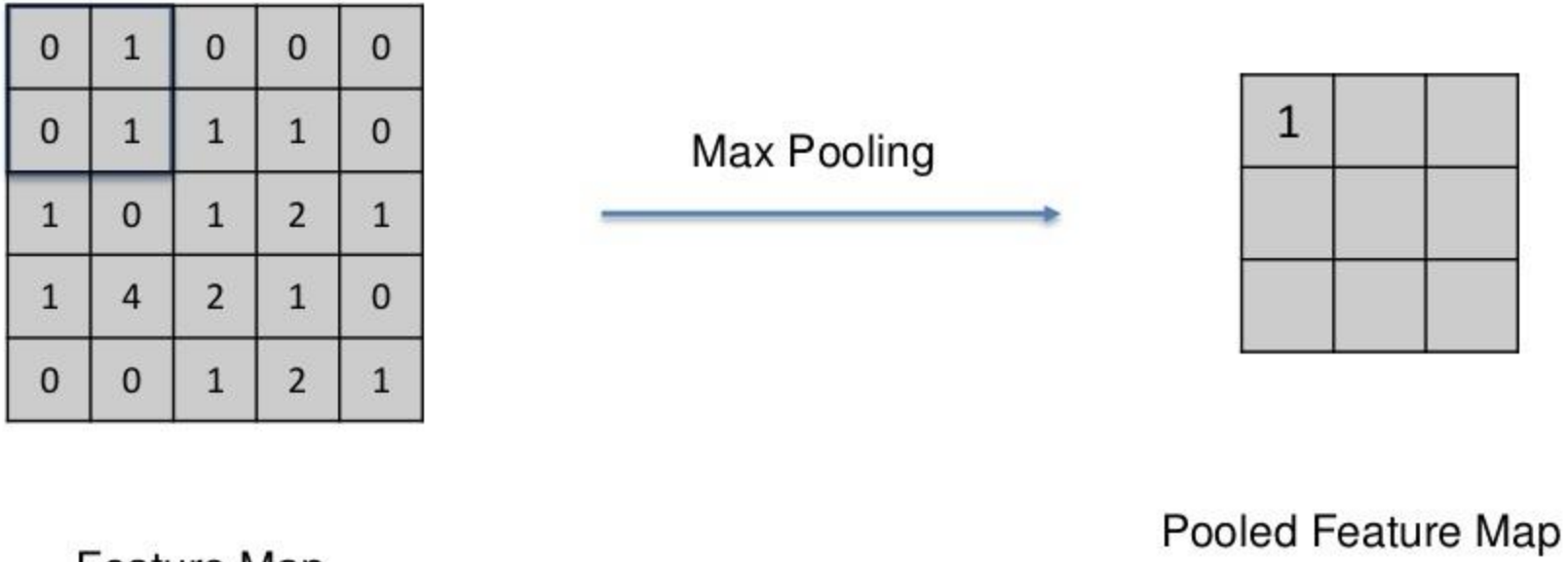
Feature Map

Max Pooling

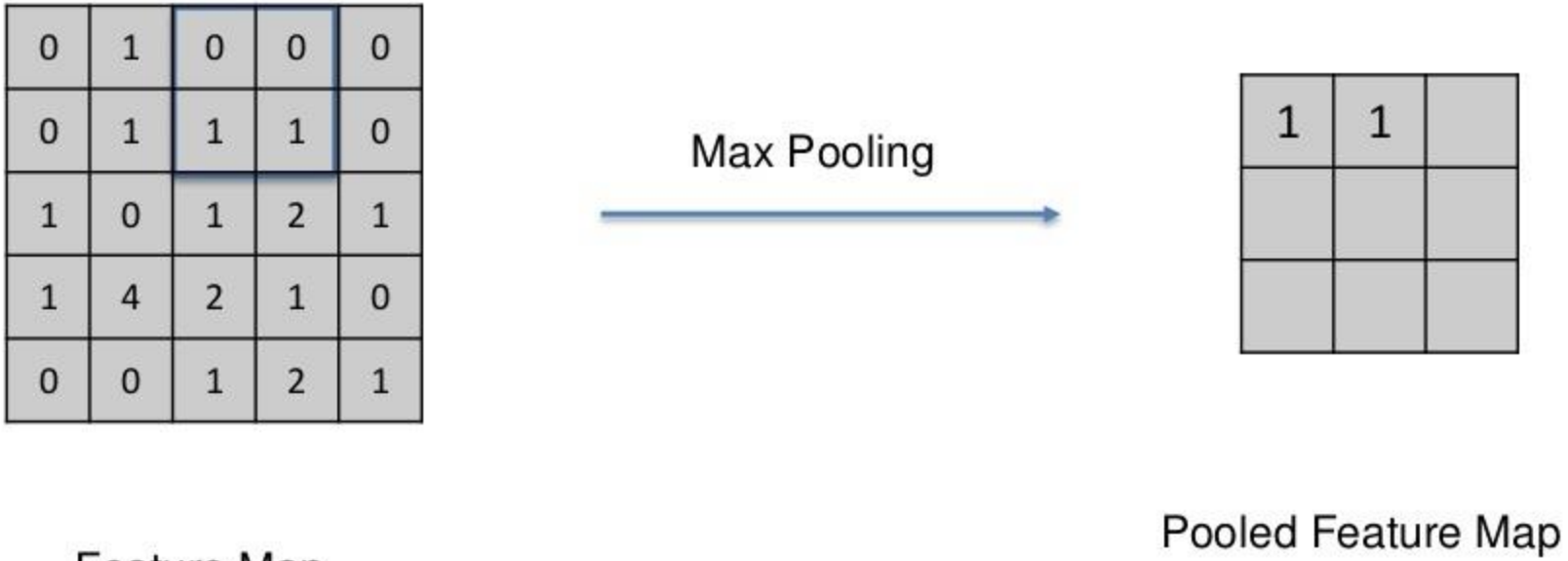


Pooled Feature Map

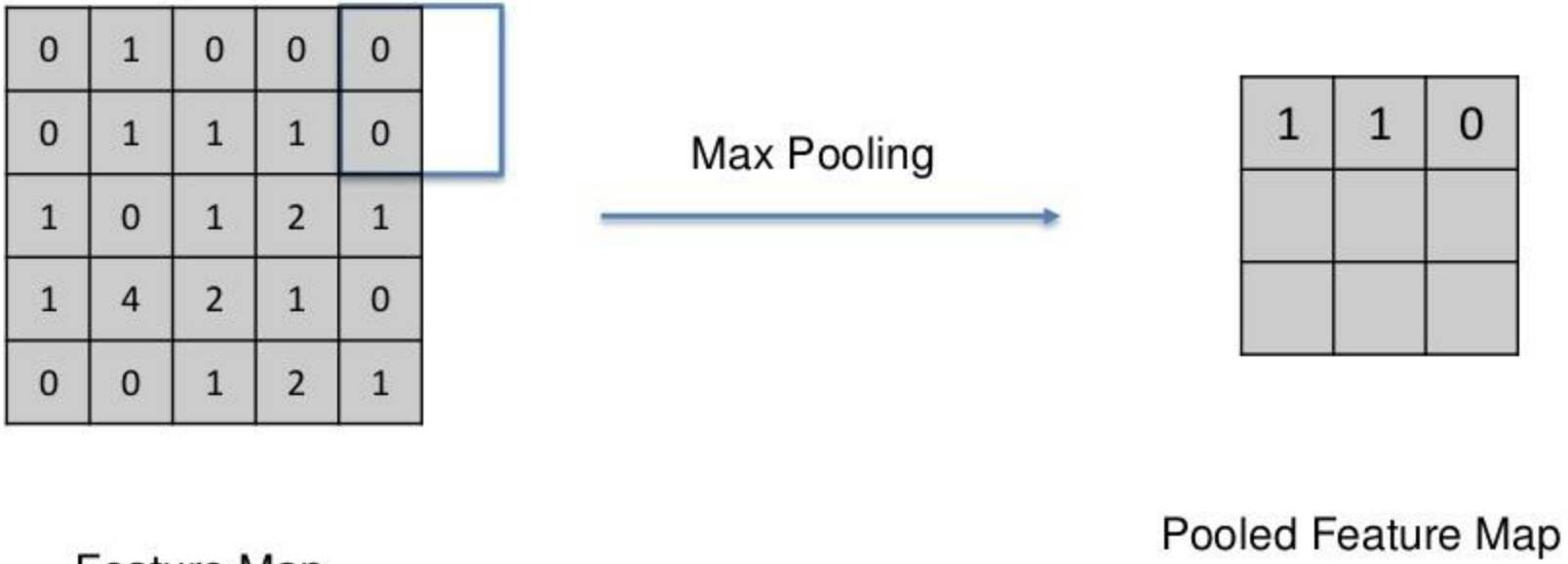
Step 2 - Max Pooling



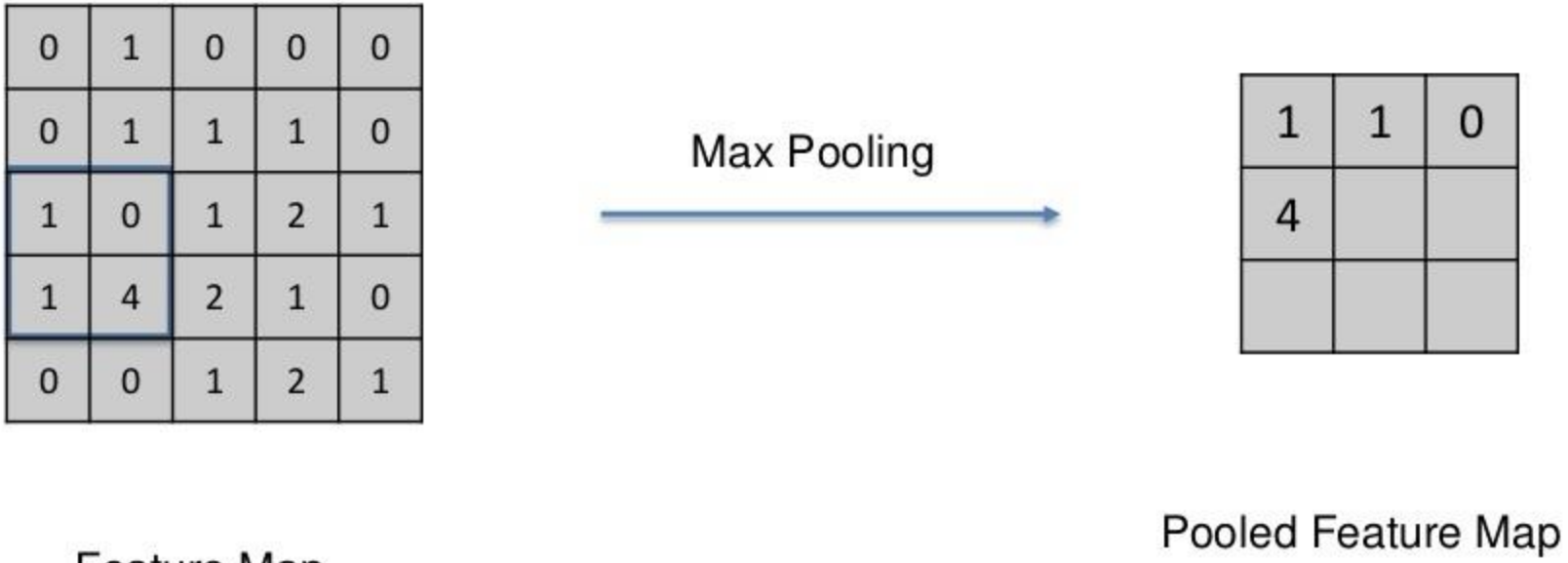
Step 2 - Max Pooling



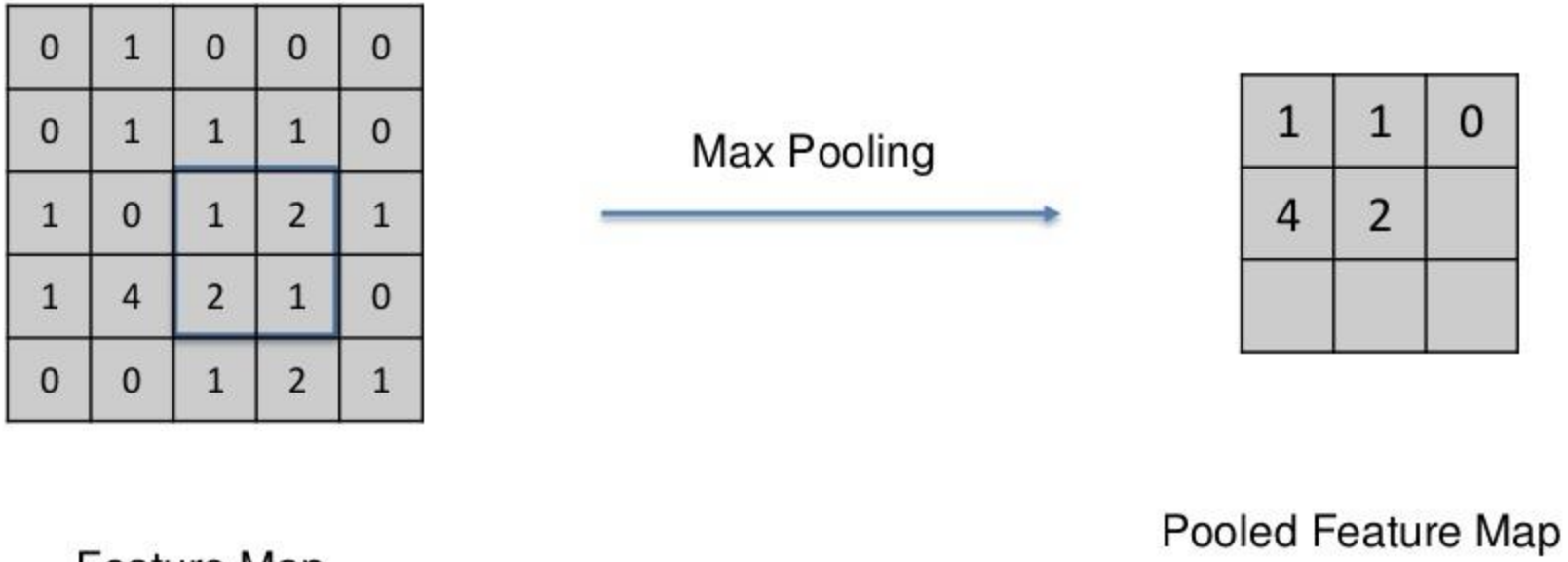
Step 2 - Max Pooling



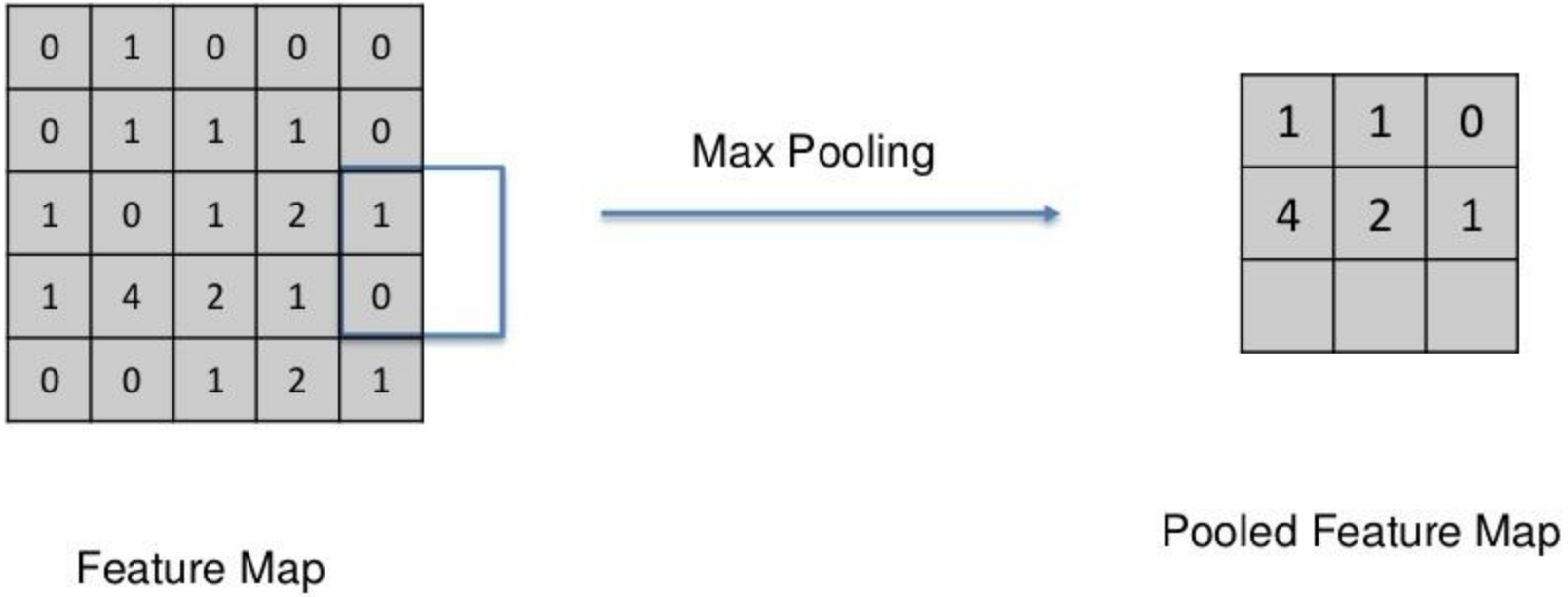
Step 2 - Max Pooling



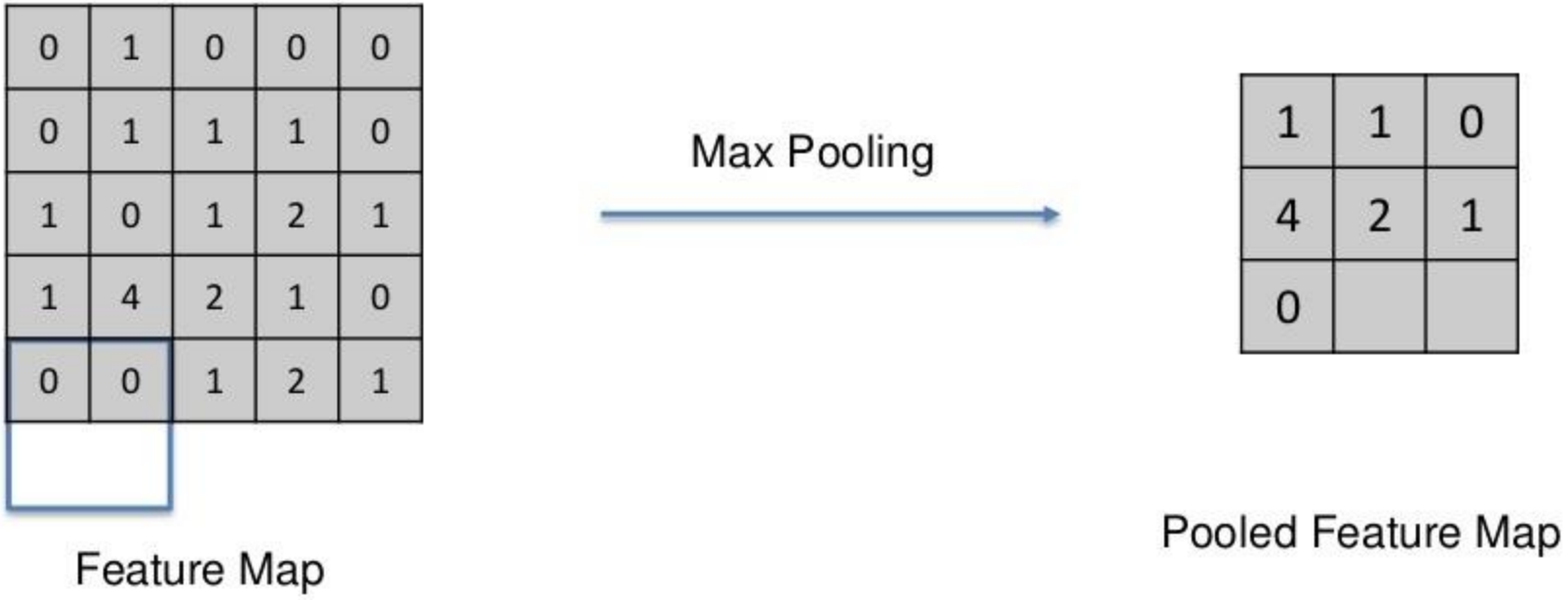
Step 2 - Max Pooling



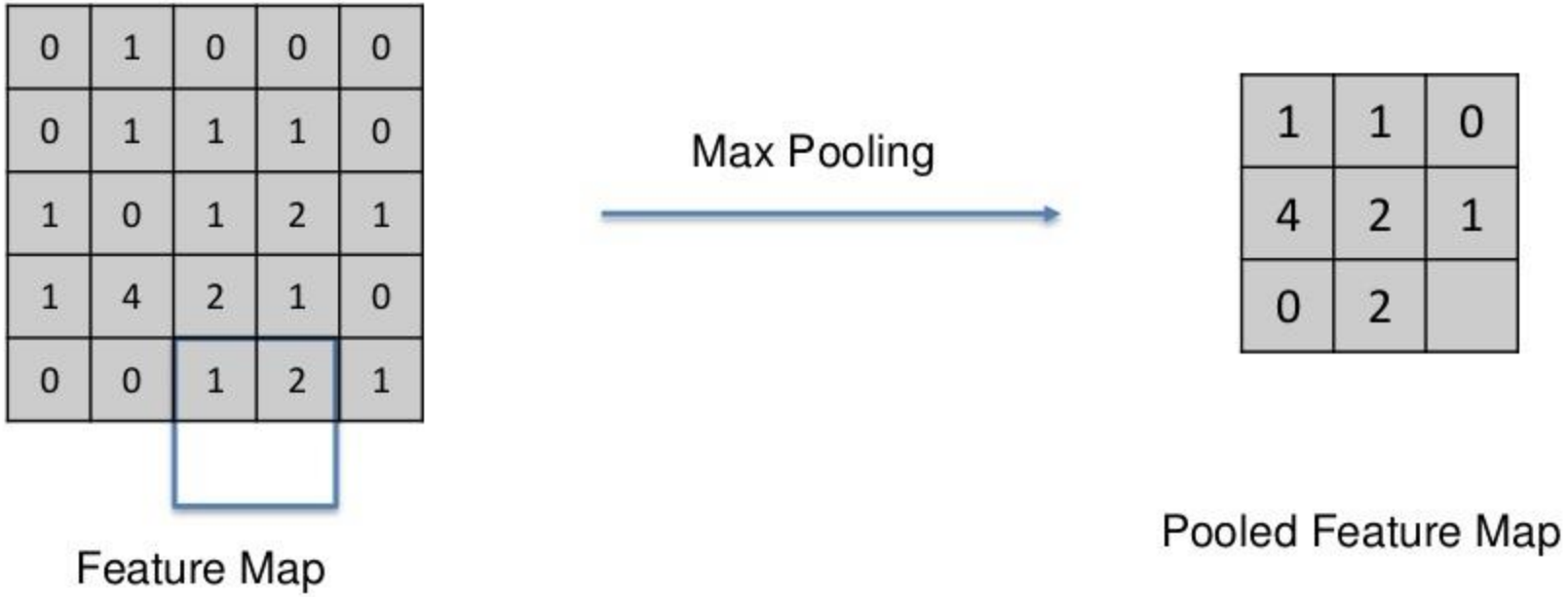
Step 2 - Max Pooling



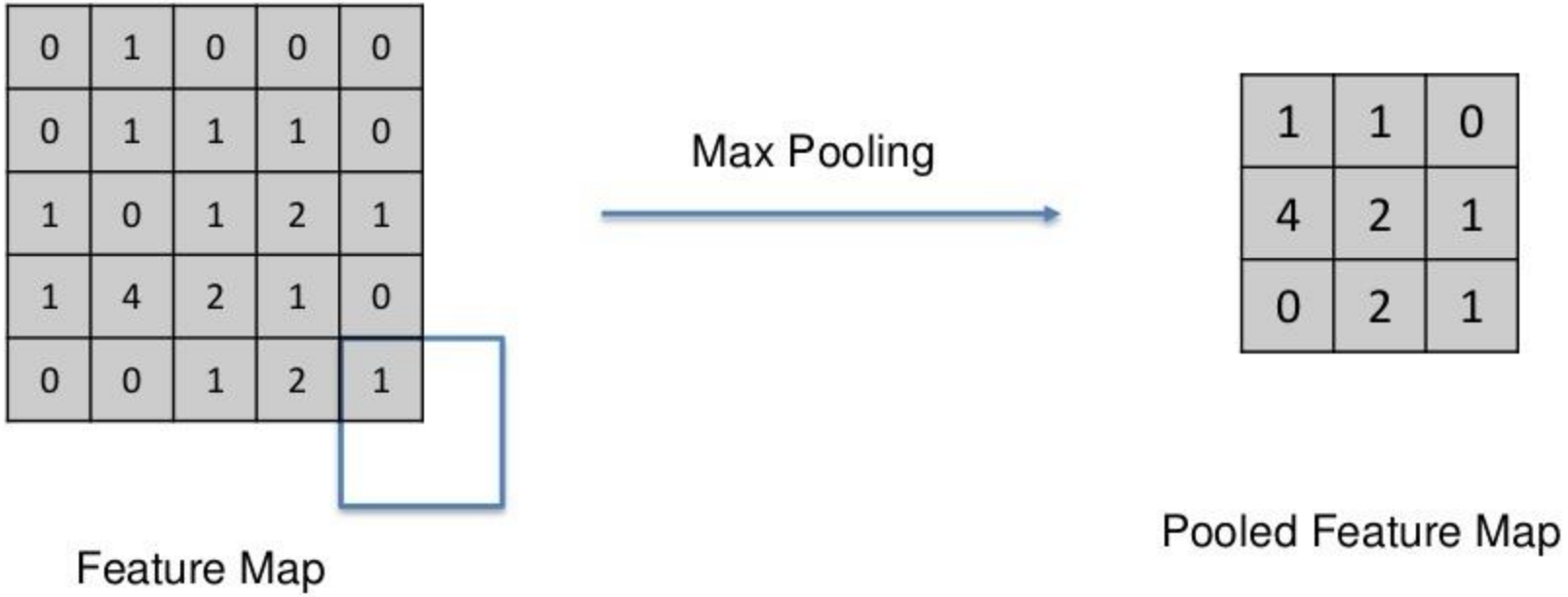
Step 2 - Max Pooling



Step 2 - Max Pooling



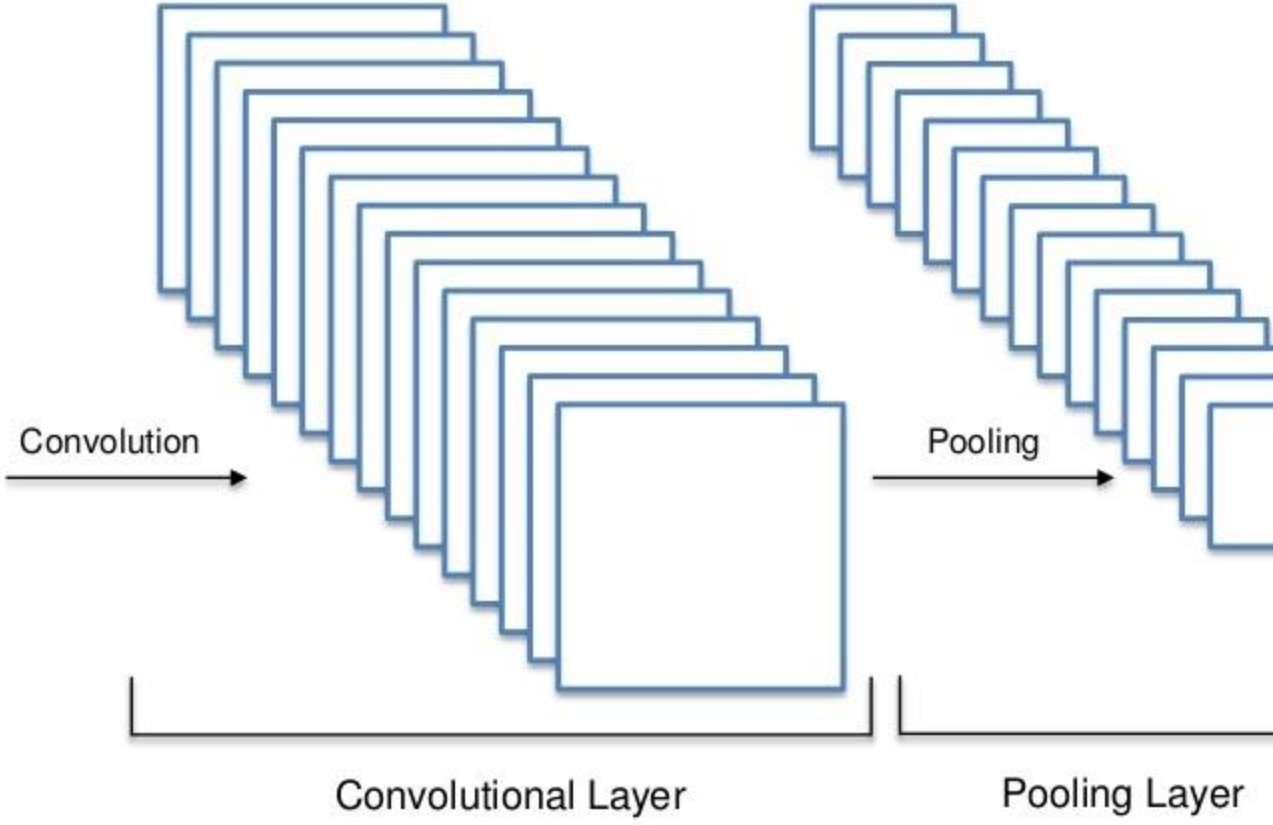
Step 2 - Max Pooling



Step 2 - Max Pooling

0	0	0	0	0	0	0
0	1	0	0	0	1	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	1	0	0	0	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

Input Image



Convolutional Neural Networks

Step 3 – Flattening

Step 3 - Flattening

1	1	0
4	2	1
0	2	1

Pooled Feature Map

Step 3 - Flattening

1	1	0
4	2	1
0	2	1

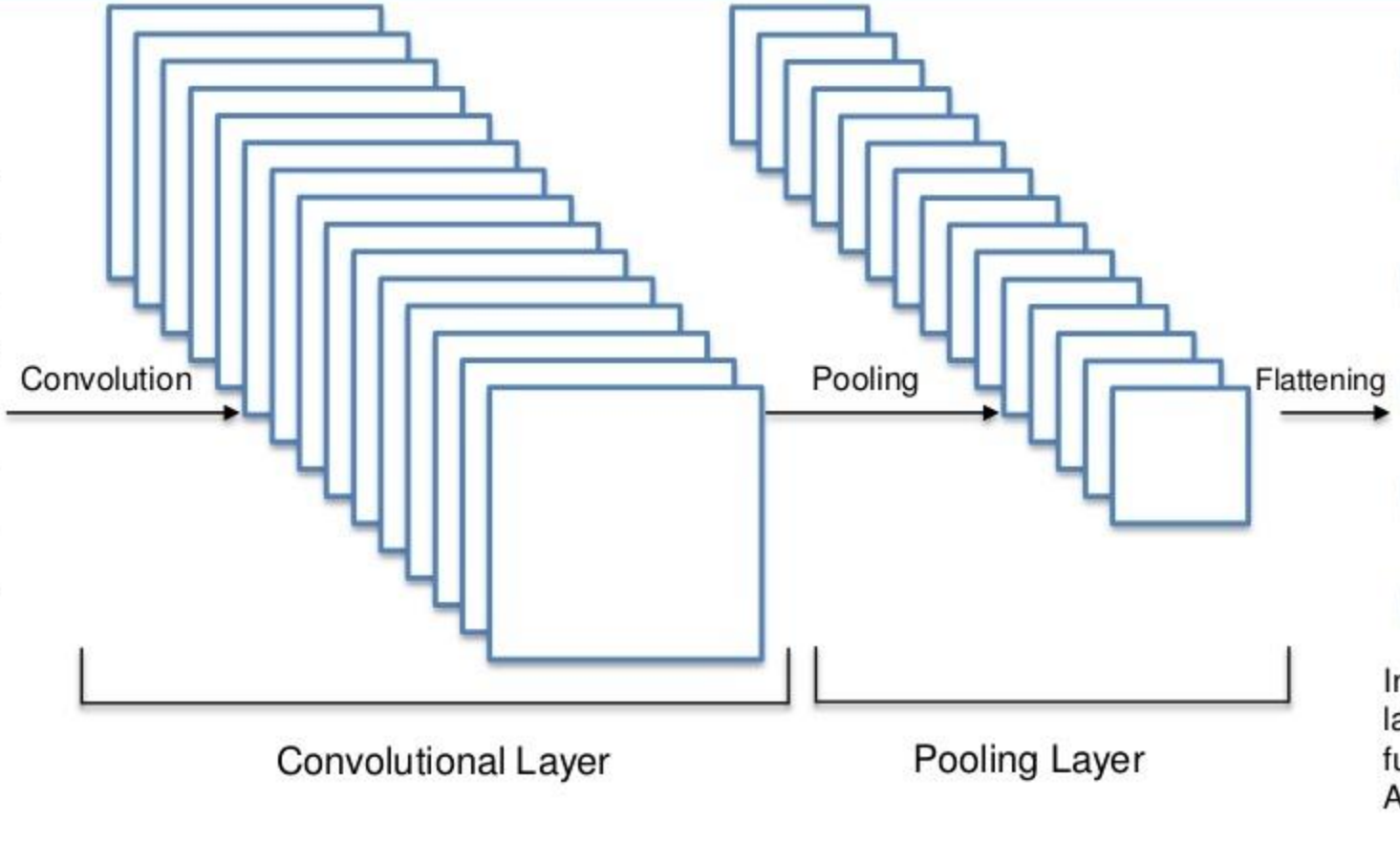
Pooled Feature Map

1
1
0
4
2
1
0
2
1

Step 3 - Flattening

0	0	0	0	0	0	0
0	1	0	0	0	1	0
0	0	0	0	0	0	0
0	0	0	1	0	0	0
0	1	0	0	0	1	0
0	0	1	1	1	0	0
0	0	0	0	0	0	0

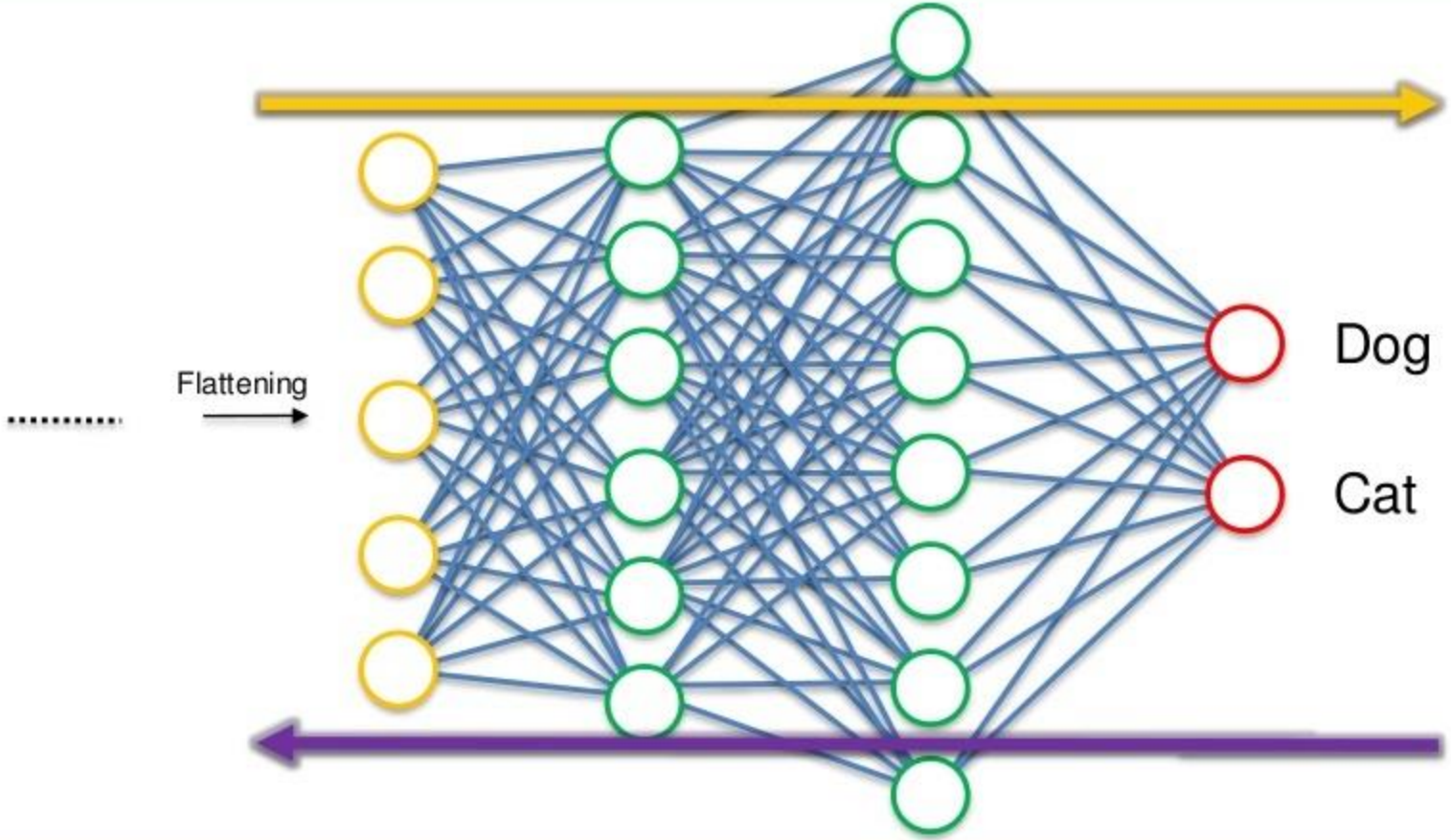
Input Image



Convolutional Neural Networks

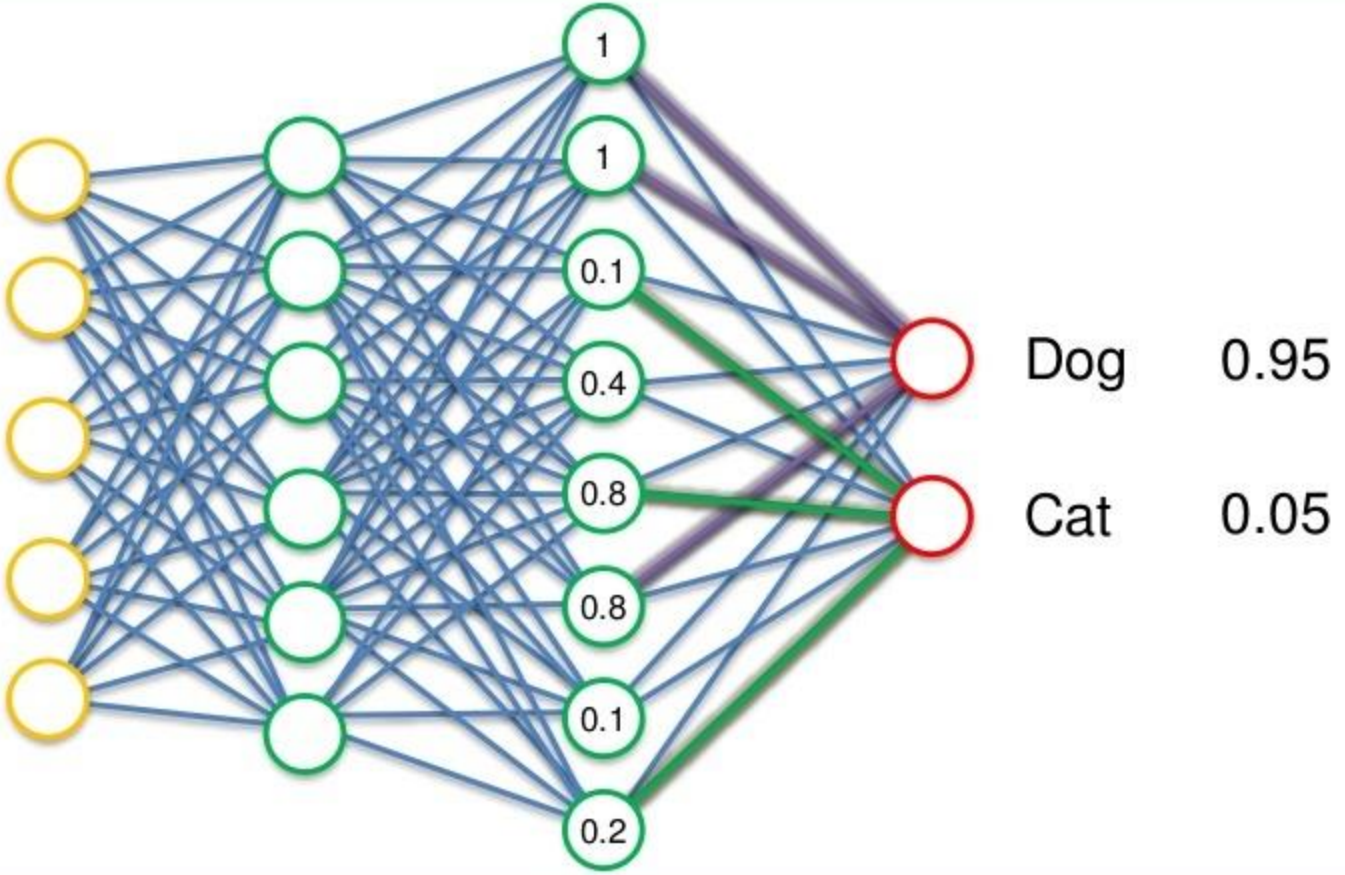
Step 4 – Full Connection

Step 4 - Full Connection



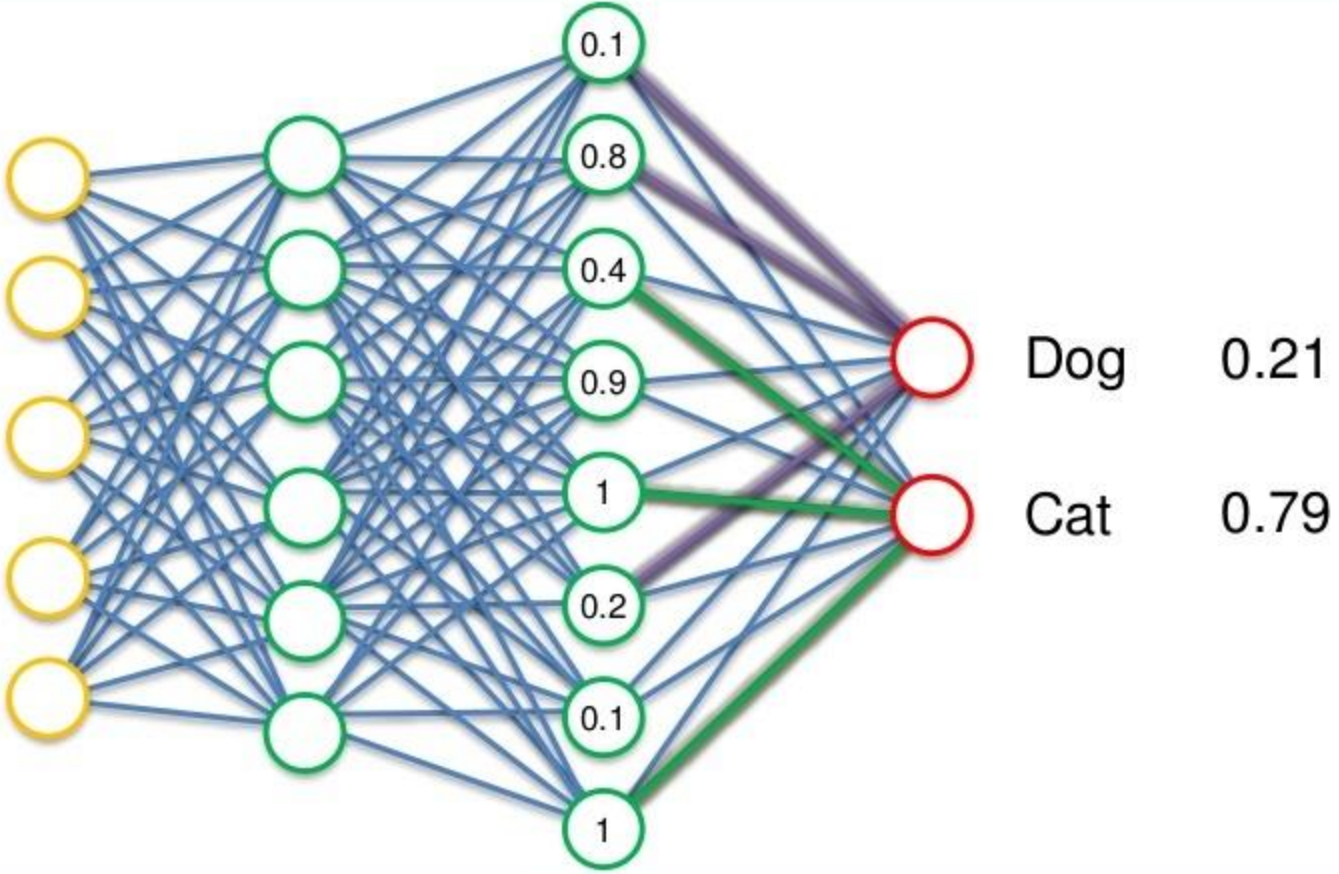
Step 4 - Full Connection

Flattening



Step 4 - Full Connection

Flattening



Summary

- Images are a 3-dimensional array of features: each pixel in the 2-D space contains three numbers from 0–255 corresponding to the Red, Green and Blue channels.
- Often, image data contains a lot of input features. A layer common in CNNs in the Conv layer, which is defined by the *filter size*, *stride*, *depth* and *padding*.
- The Conv layer uses the same parameters and applies the same neuron(s) across different regions of the image, thereby reducing the number of parameters.
- Another common layer in CNNs is the max-pooling layer, defined by the *filter size* and *stride*, which reduces the spatial size by taking the maximum of the numbers within its filter.
- We also typically use our traditional Fully-Connected layers at the end of our CNNs.
- AlexNet was a CNN which revolutionized the field of Deep Learning, and is built from conv layers, max-pooling layers and FC layers.
- When many layers are put together, the earlier layers learn low-level features and combine them in later layers for more complex representations.

ACKNOWLEDGEMENTS

https://towardsdatascience.com/intuitive-deep-learning-part-2-cnns-for-computer-vision-472bbb2c8060

https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-1-convolution-operation