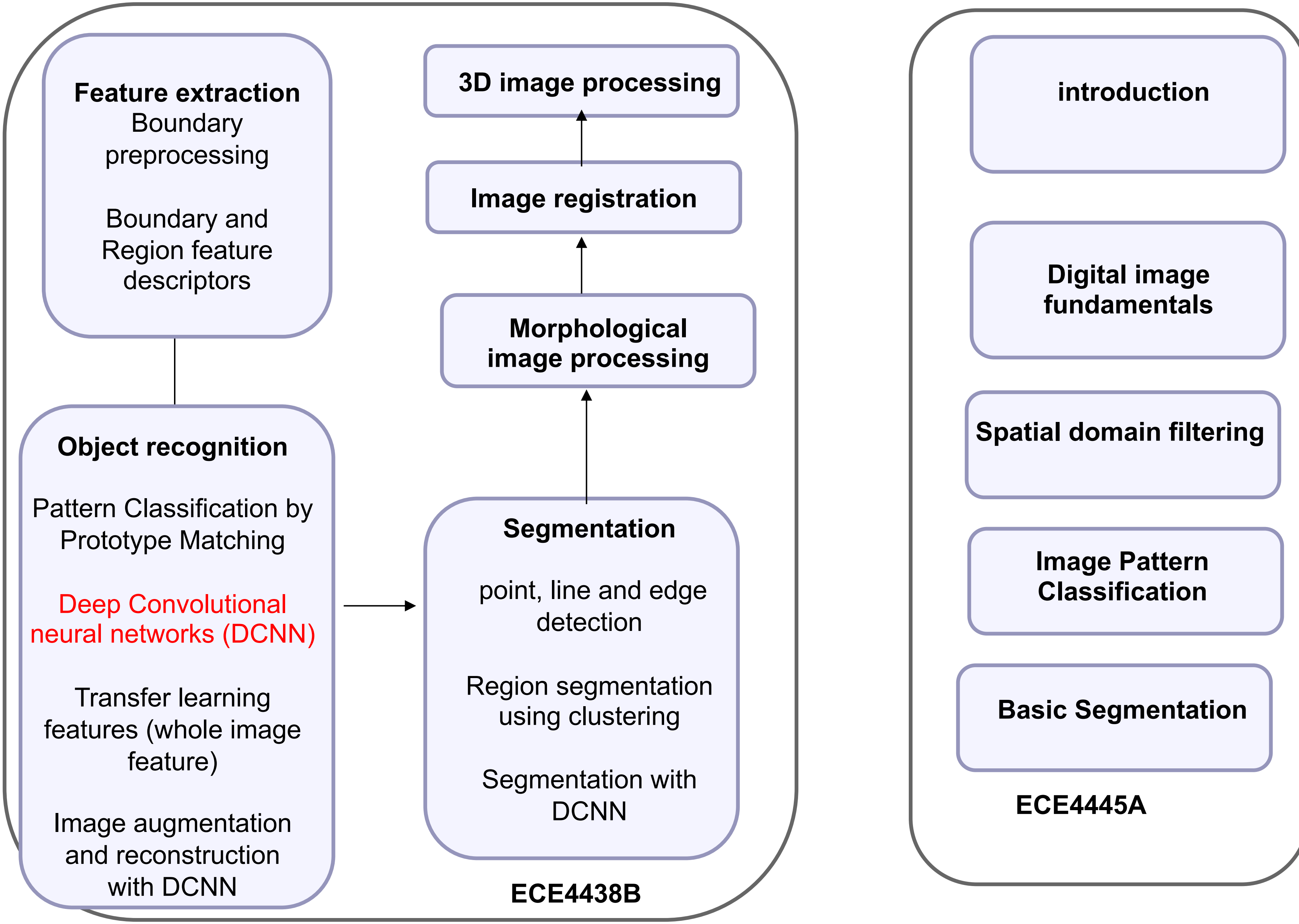
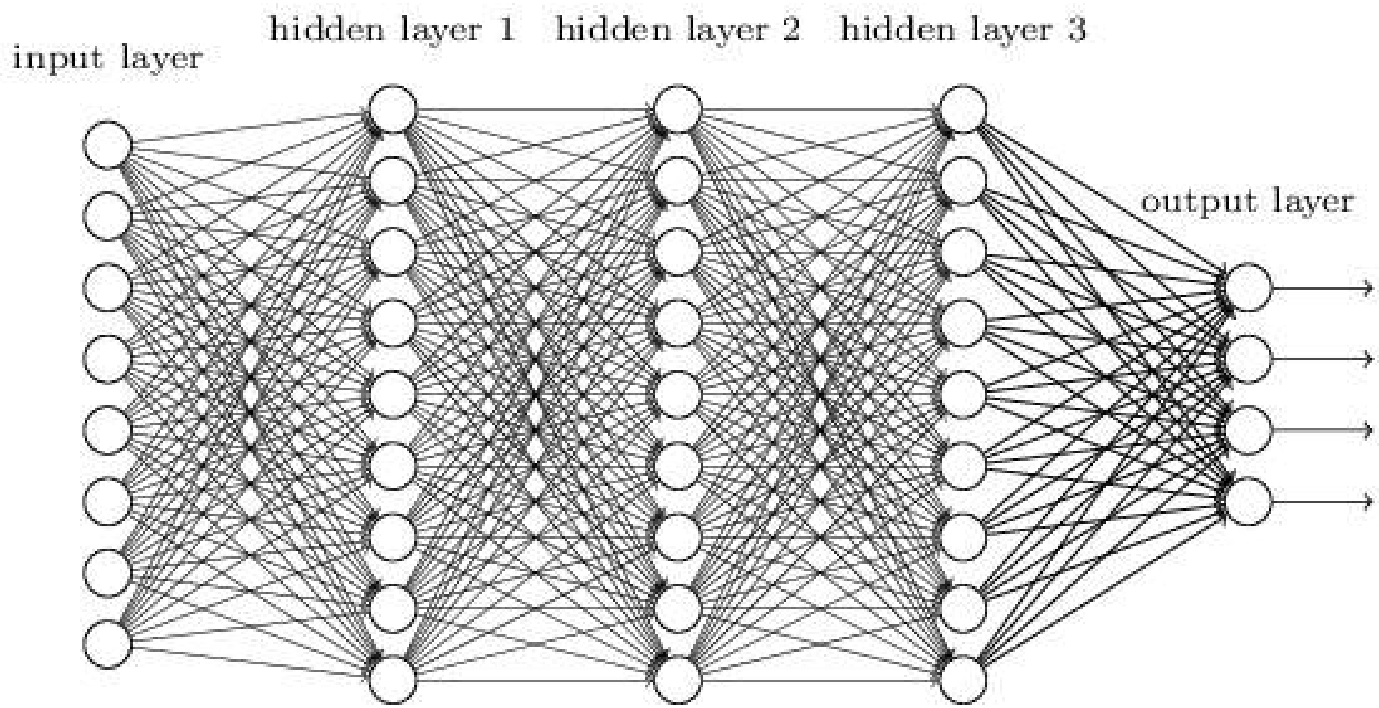
Courses topics at a glance

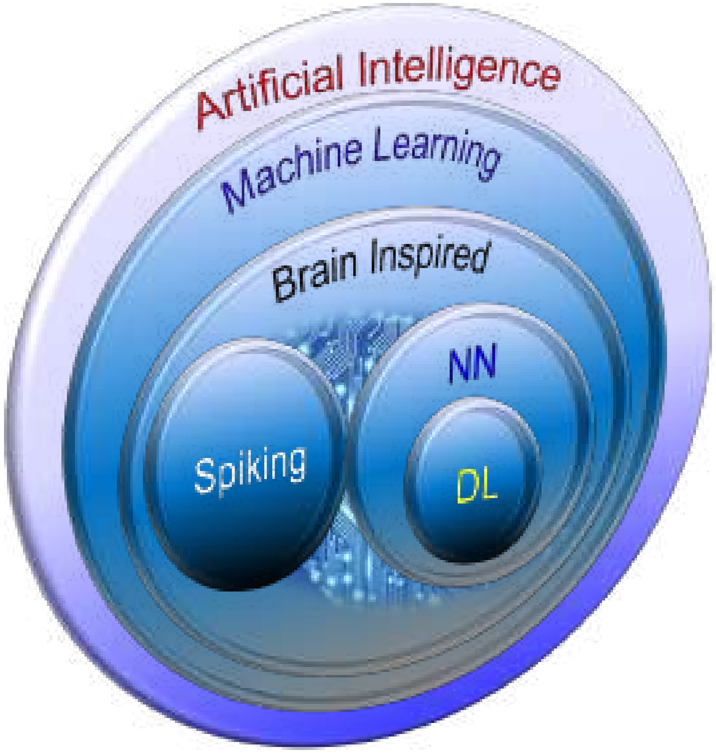


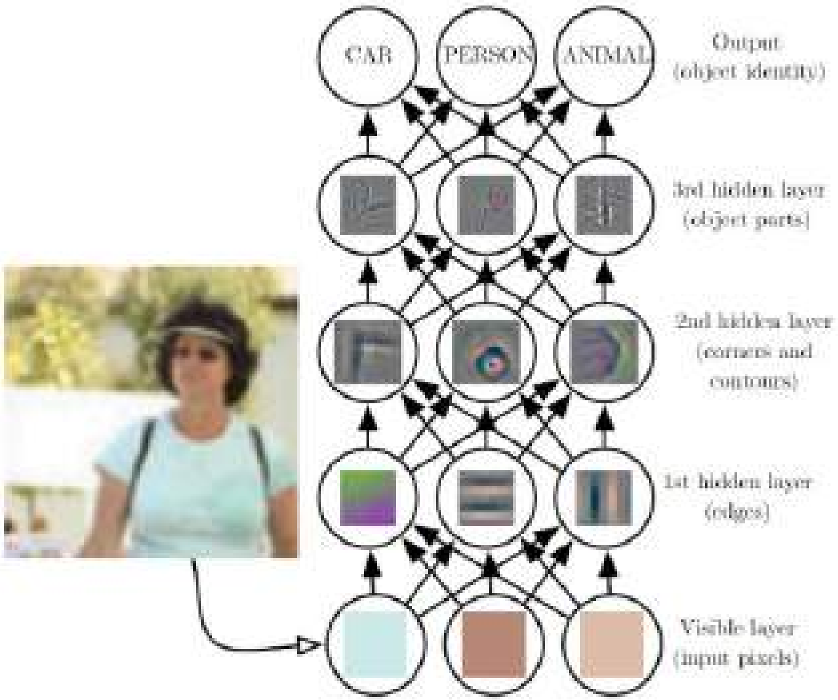
Deep Convolutional Neural Network

- We know it is good to learn a small model.
- From this fully connected model, do we really need all the edges?
- Can some of these be shared?

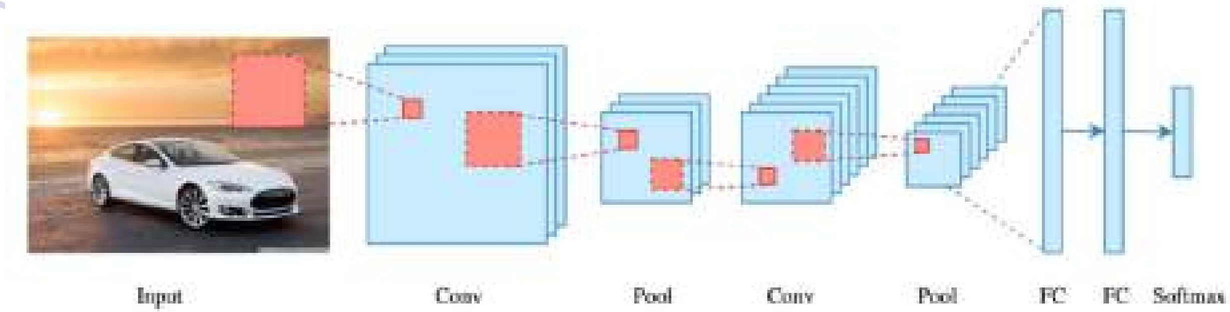


Convolutional Neural Networks





Convolutional Neural Networks



- All CNN models follow a similar architecture, as shown in the figure below.
- There is an input image that we're working with. We perform a series convolution + pooling operations, followed by a number of fully connected layers. If we are performing multiclass classification the output is softmax. We will now dive into each component.

Convlution

- The main building block of CNN is the convolutional layer. Convolution is a mathematical operation to merge two sets of information. In our case the convolution is applied on the input data using a *convolution filter* to produce a *feature map*.

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

Input

1	0	1
0	1	0
1	0	1

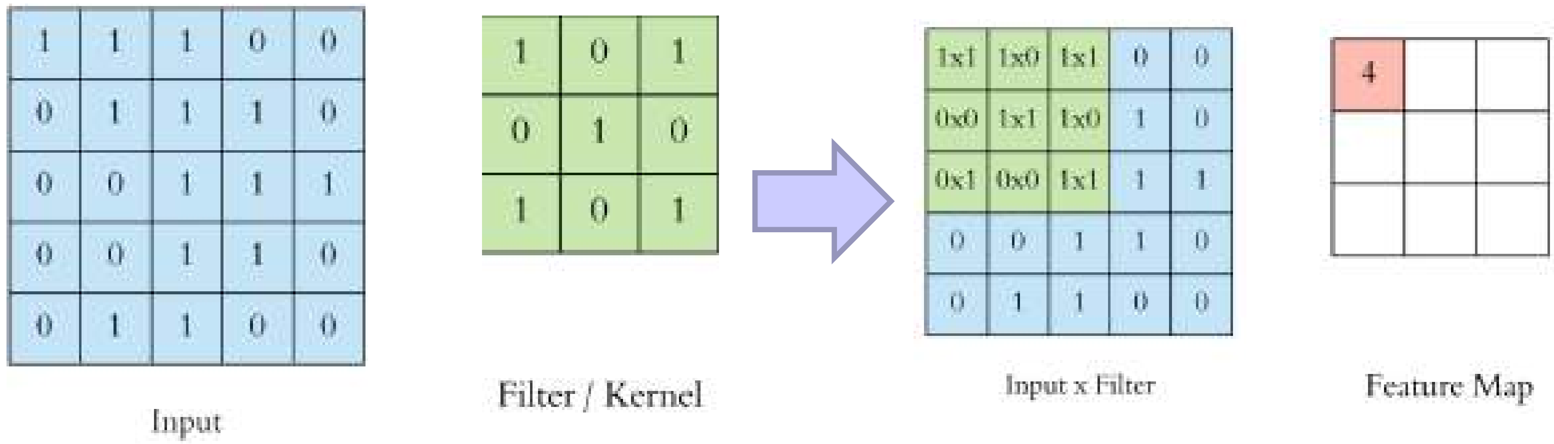
Filter / Kernel

uses trainable feature extraction method.

- On the left side is the input to the convolution layer, for example the input image. On the right is the convolution filter, also called the kernel, we will use these terms interchangeably. This is called a 3x3 convolution due to the shape of the filter.

Convlution

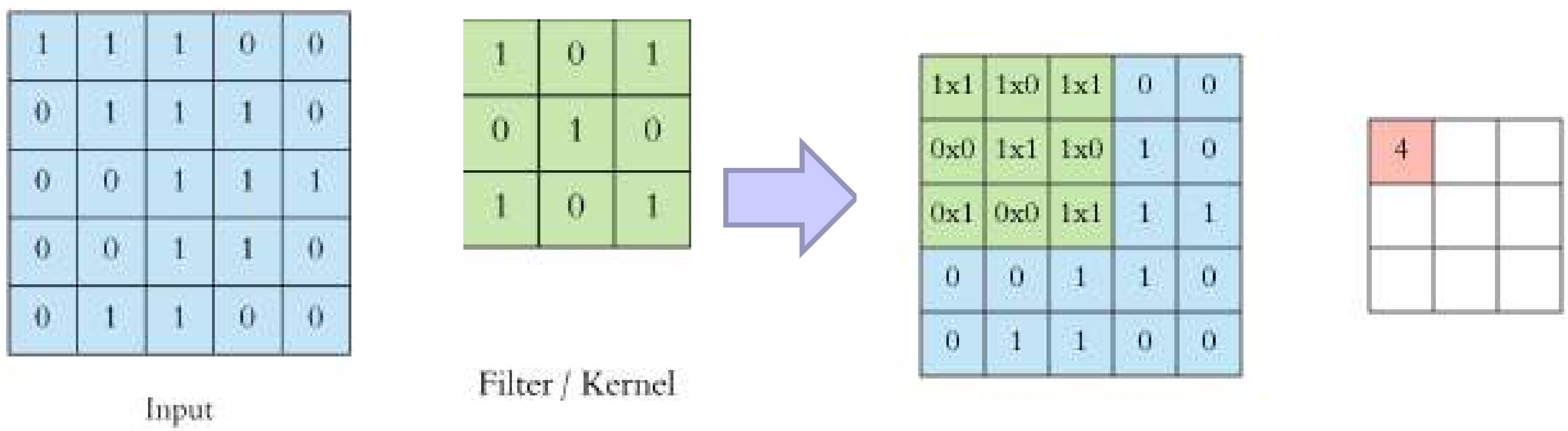
- We perform the convolution operation by sliding this filter over the input. At every location, we do element-wise matrix multiplication and sum the result. This sum goes into the feature map. The green area where the convolution operation takes place is called the *receptive field*. Due to the size of the filter the receptive field is also 3x3.



- Here the filter is at the top left, the output of the convolution operation “4” is shown in the resulting feature map. We then slide the filter to the right and perform the same operation, adding that result to the feature map as well.

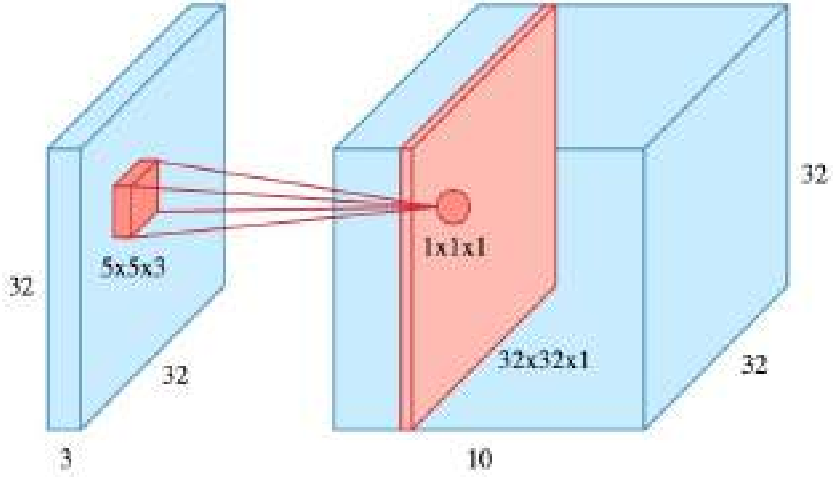
Convlution

- We continue like this and aggregate the convolution results in the feature map. Here’s an animation that shows the entire convolution operation.



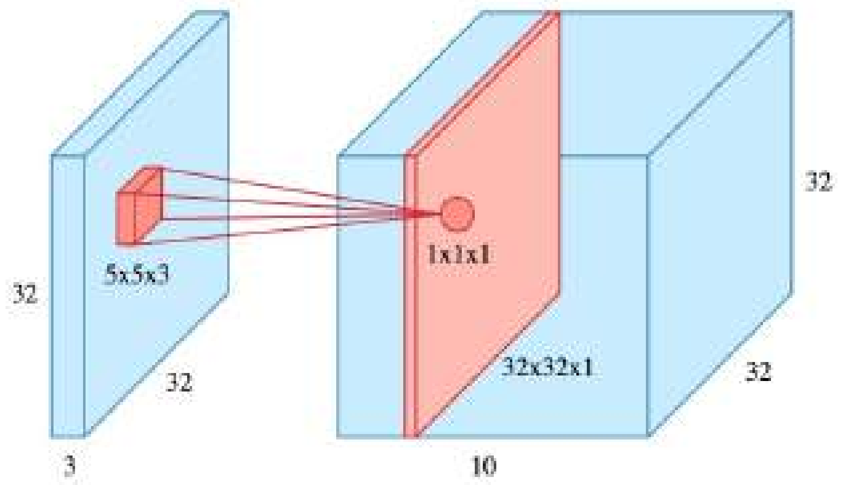
- This was an example convolution operation shown in 2D using a 3x3 filter. But in reality these convolutions are performed in 3D. In reality an image is represented as a 3D matrix with dimensions of height, width and depth, where depth corresponds to color channels (RGB). A convolution filter has a specific height and width, like 3x3 or 5x5, and by design it covers the entire depth of its input so it needs to be 3D as well.

Convlution



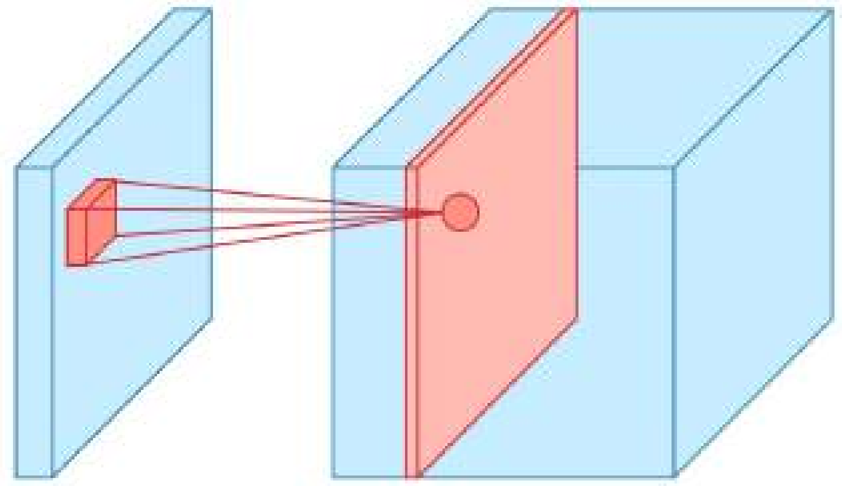
- Let’s say we have a 32x32x3 image and we use a filter of size 5x5x3 (note that the depth of the convolution filter matches the depth of the image, both being 3). When the filter is at a particular location it covers a small volume of the input, and we perform the convolution operation described above. The only difference is this time we do the sum of matrix multiply in 3D instead of 2D, but the result is still a scalar. We slide the filter over the input like above and perform the convolution at every location aggregating the result in a feature map. This feature map is of size 32x32x1, shown as the red slice on the right.

Convlution



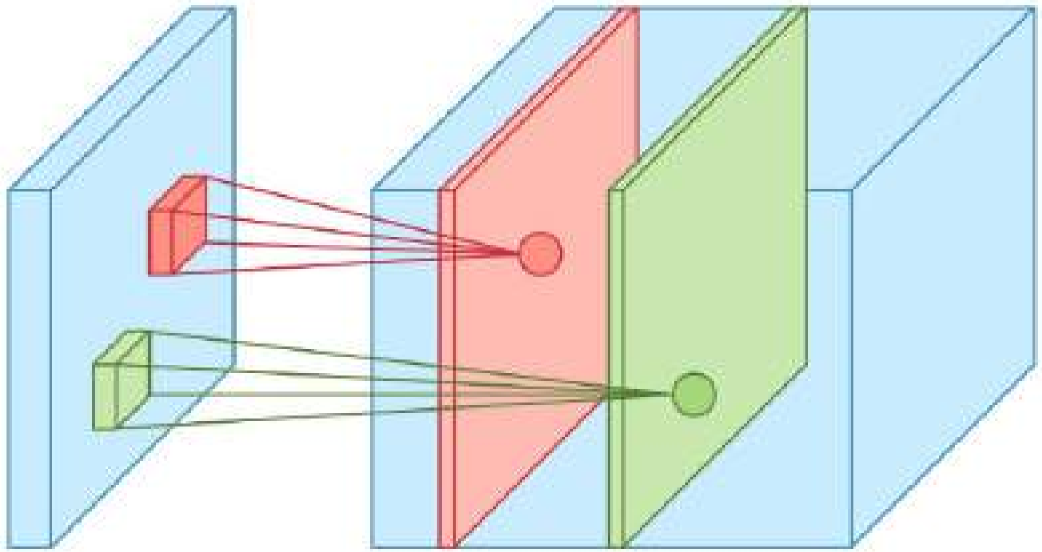
- ❑ If we used 10 different filters we would have 10 feature maps of size 32x32x1 and stacking them along the depth dimension would give us the final output of the convolution layer: a volume of size 32x32x10.
- ❑ Note that the height and width of the feature map are unchanged and still 32, it’s due to padding and we will elaborate on that shortly.

Convlution



- ❑ The animation shows the sliding operation at 4 locations, but in reality it's performed over the entire input.
- ❑ Note that the height and width of the feature map are unchanged and still 32, it's due to padding and we will elaborate on that shortly.

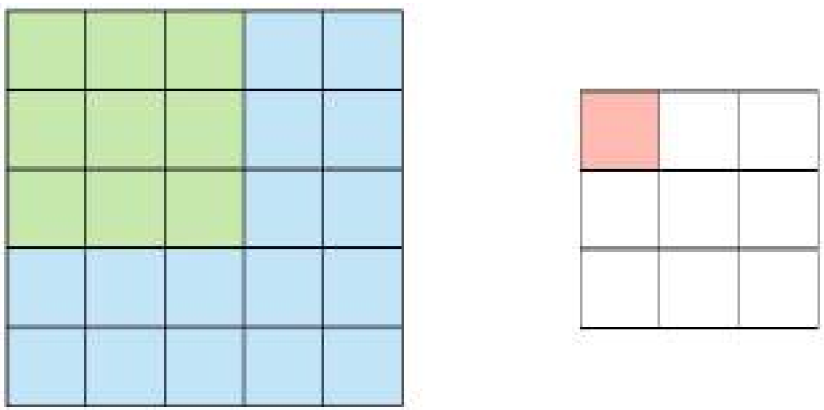
Convlution



- ❑we can see how two feature maps are stacked along the depth dimension. The convolution operation for each filter is performed independently and the resulting feature maps are disjoint.

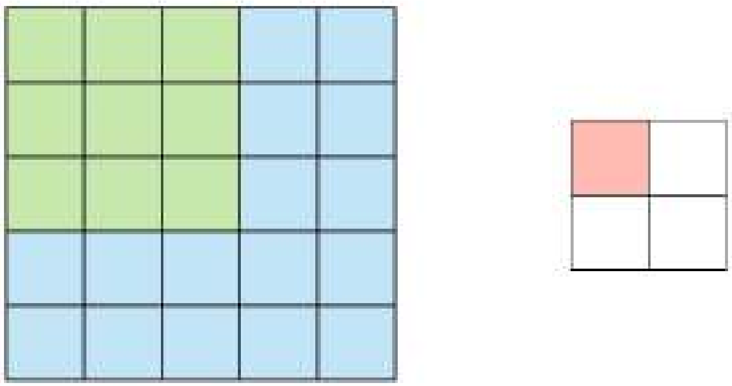
Stride and Padding

- *Stride* specifies how much we move the convolution filter at each step. By default the value is 1, as you can see in the figure below.



Stride 1 Feature Map

- We can have bigger strides if we want less overlap between the receptive fields. This also makes the resulting feature map smaller since we are skipping over potential locations. The following figure demonstrates a stride of 2. Note that the feature map got smaller.



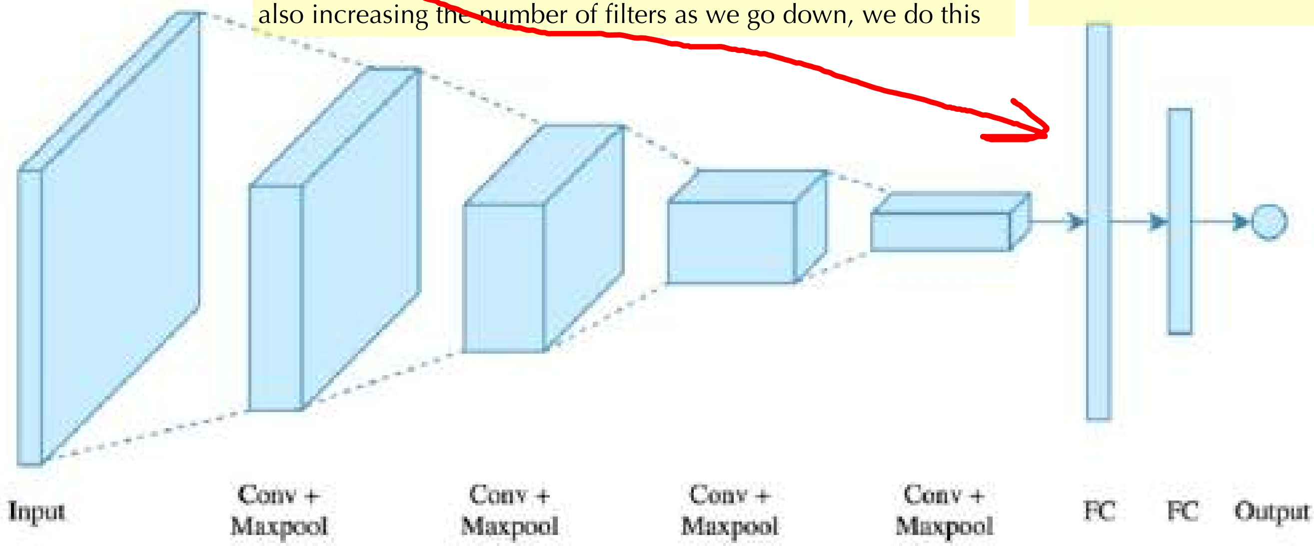
Stride 2 Feature Map

Convlution

As you go down it is a good idea to increase the channel size. i.e increase the kernel channel when convolving.
Recommendation: always use odd size of filters.
7x7 for the first layer and 3 x 3 on the consequent layers. We can also increasing the number of filters as we go down, we do this

!! know the formula to know the amount of padding size to keep the size of original image.

padding size = (kernel_size * - 1) / 2

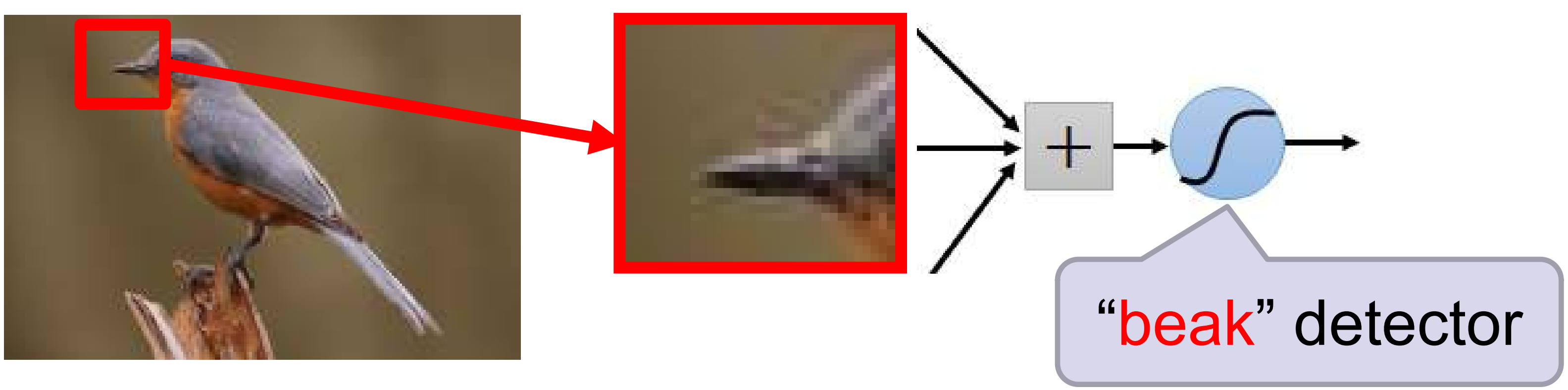


- ❑we can see how two feature maps are stacked along the depth dimension. The convolution operation for each filter is performed independently and the resulting feature maps are disjoint.

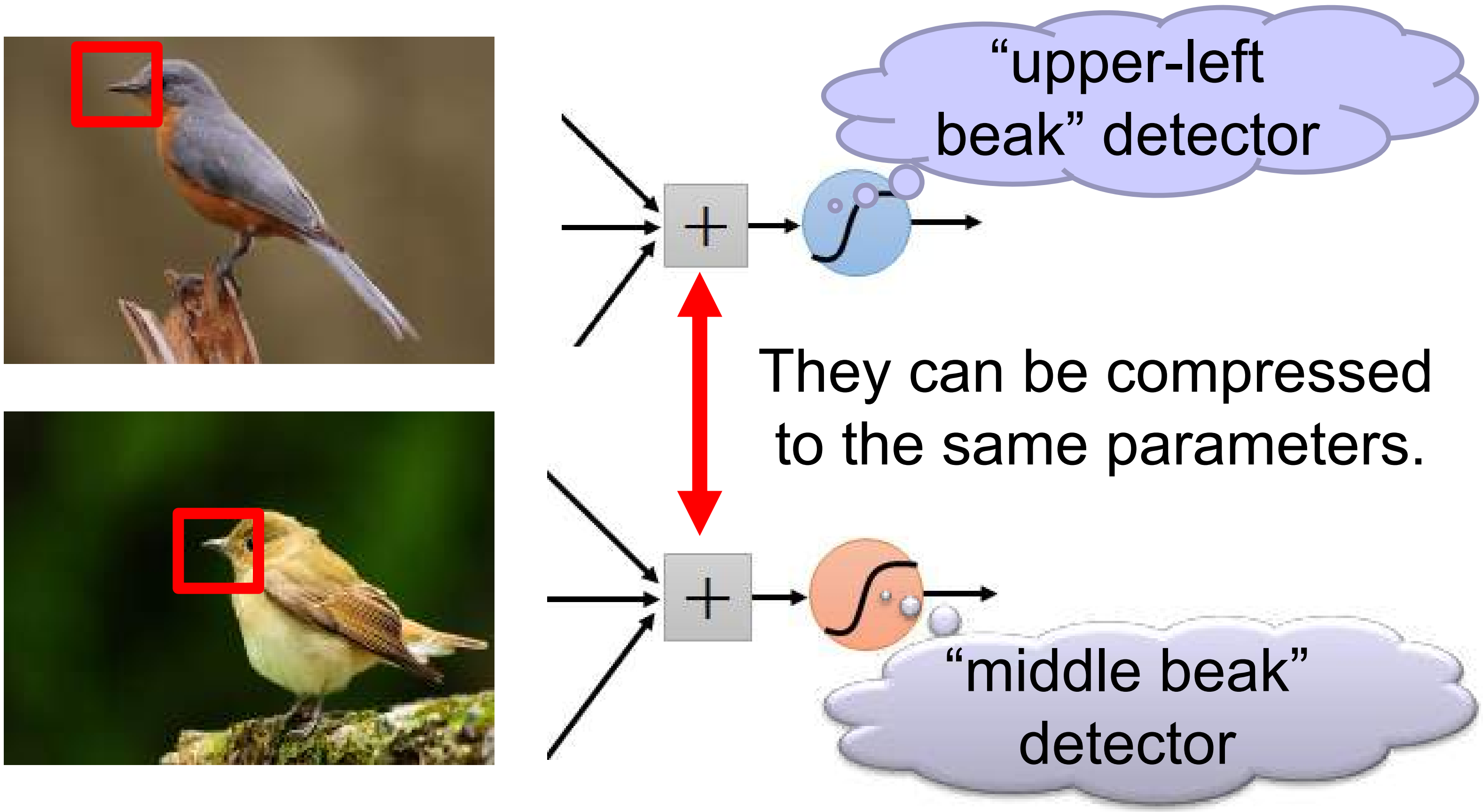
Consider learning an image:

- Some patterns are much smaller than the whole image

Can represent a small region with fewer parameters

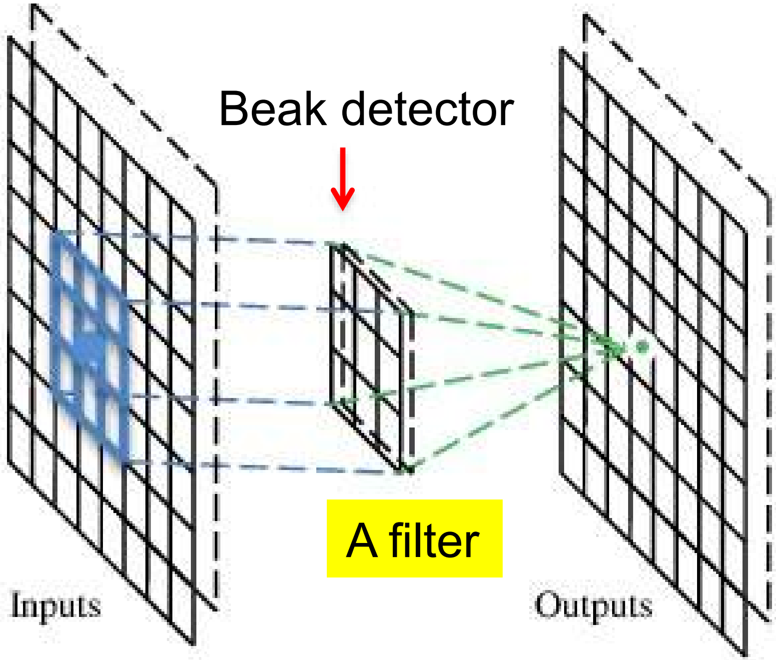


Same pattern appears in different places:
They can be compressed!
What about training a lot of such "small" detectors and each detector must "move around".



A convolutional layer

A CNN is a neural network with some convolutional layers (and some other layers). A convolutional layer has a number of filters that does convolutional operation.



Convolution

These are the network parameters to be learned.

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

6 x 6 image

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

⋮ ⋮

Each filter detects a small pattern (3 x 3).

Convolution

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

stride=1

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

Dot product

3 -1

6 x 6 image

Convolution

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

If stride=2

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

6 x 6 image

3 -3

Convolution

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

stride=1

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

6 x 6 image

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

Convolution

-1	1	-1
-1	1	-1
-1	1	-1

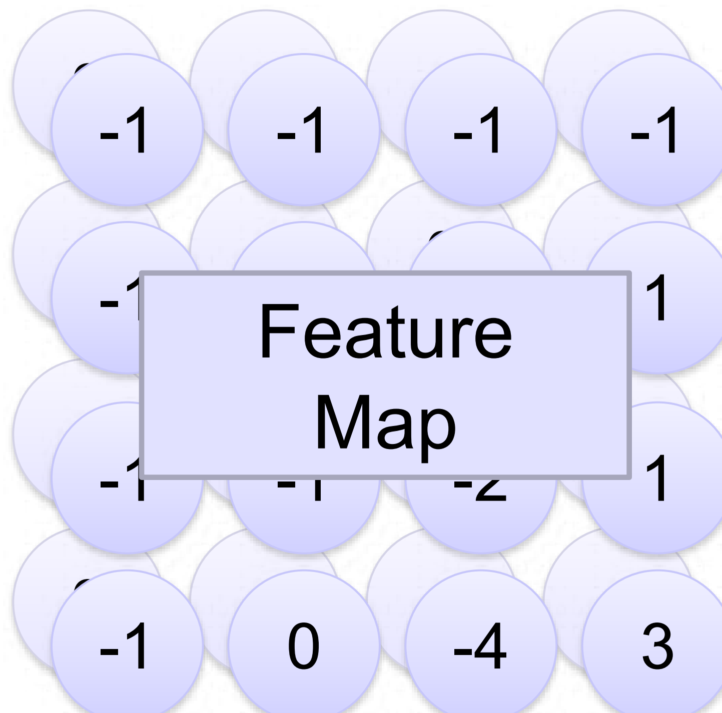
Filter 2

stride=1

Repeat this for each filter

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

6 x 6 image



Two 4 x 4 images
Forming 2 x 4 x 4 matrix

Color image: RGB 3 channels

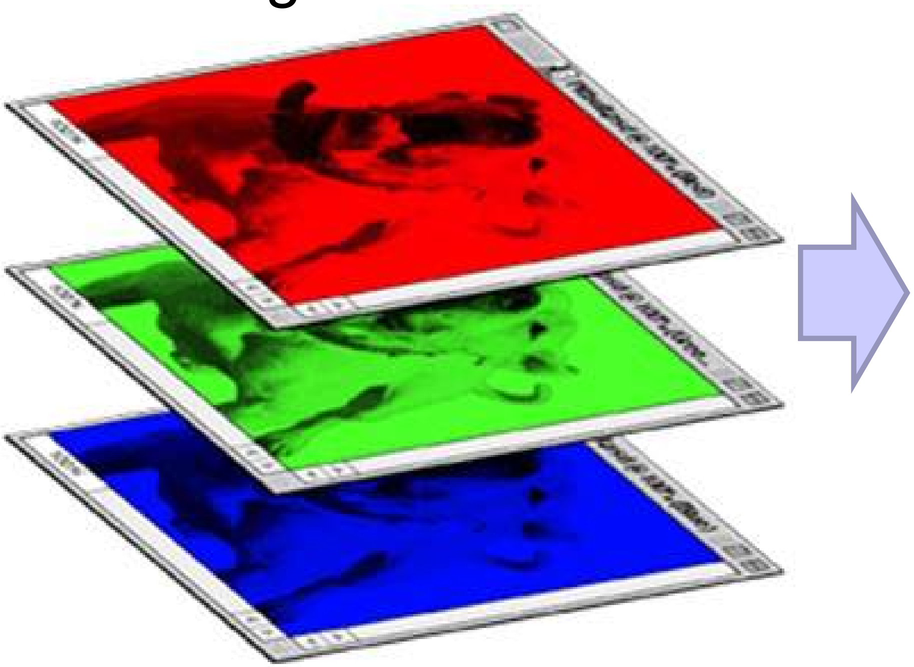
1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

-1	1	-1
-1	1	-1
-1	1	-1

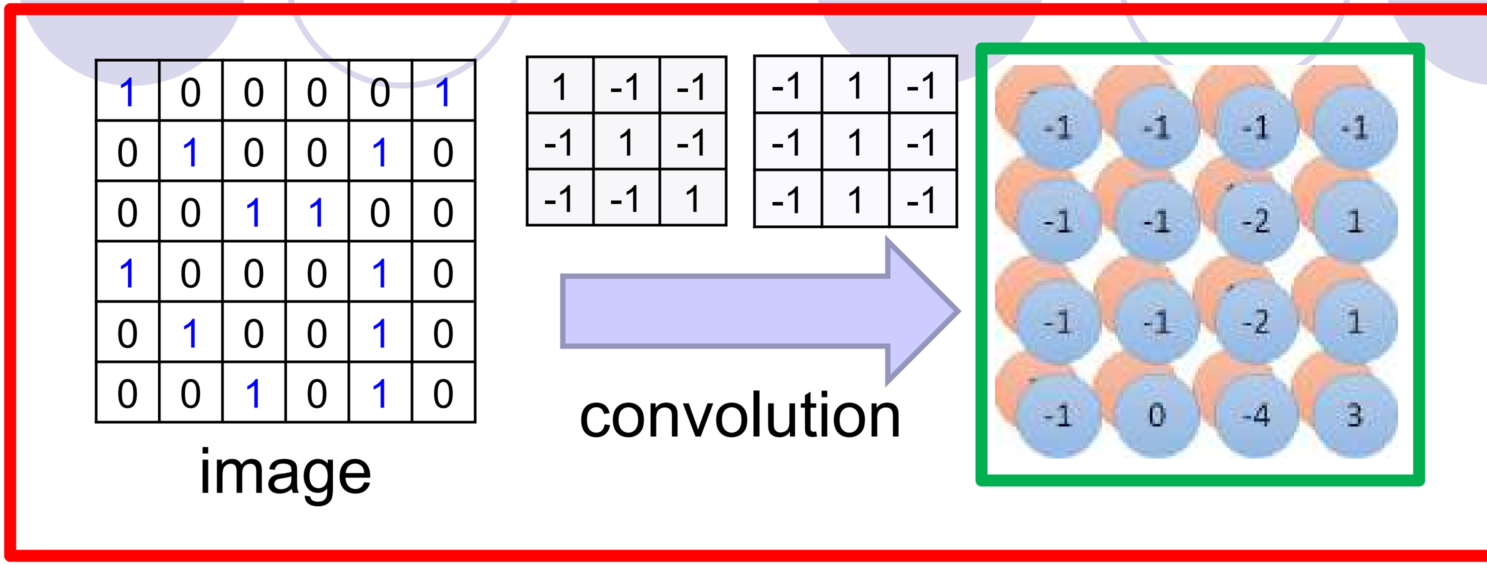
Filter 2

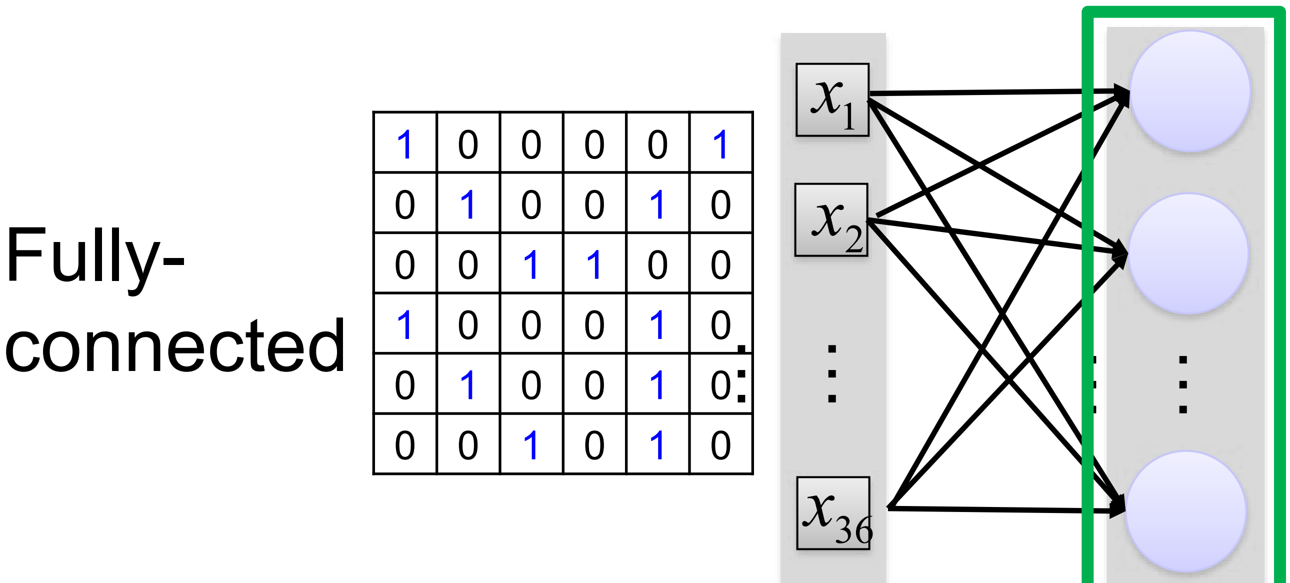
Color image

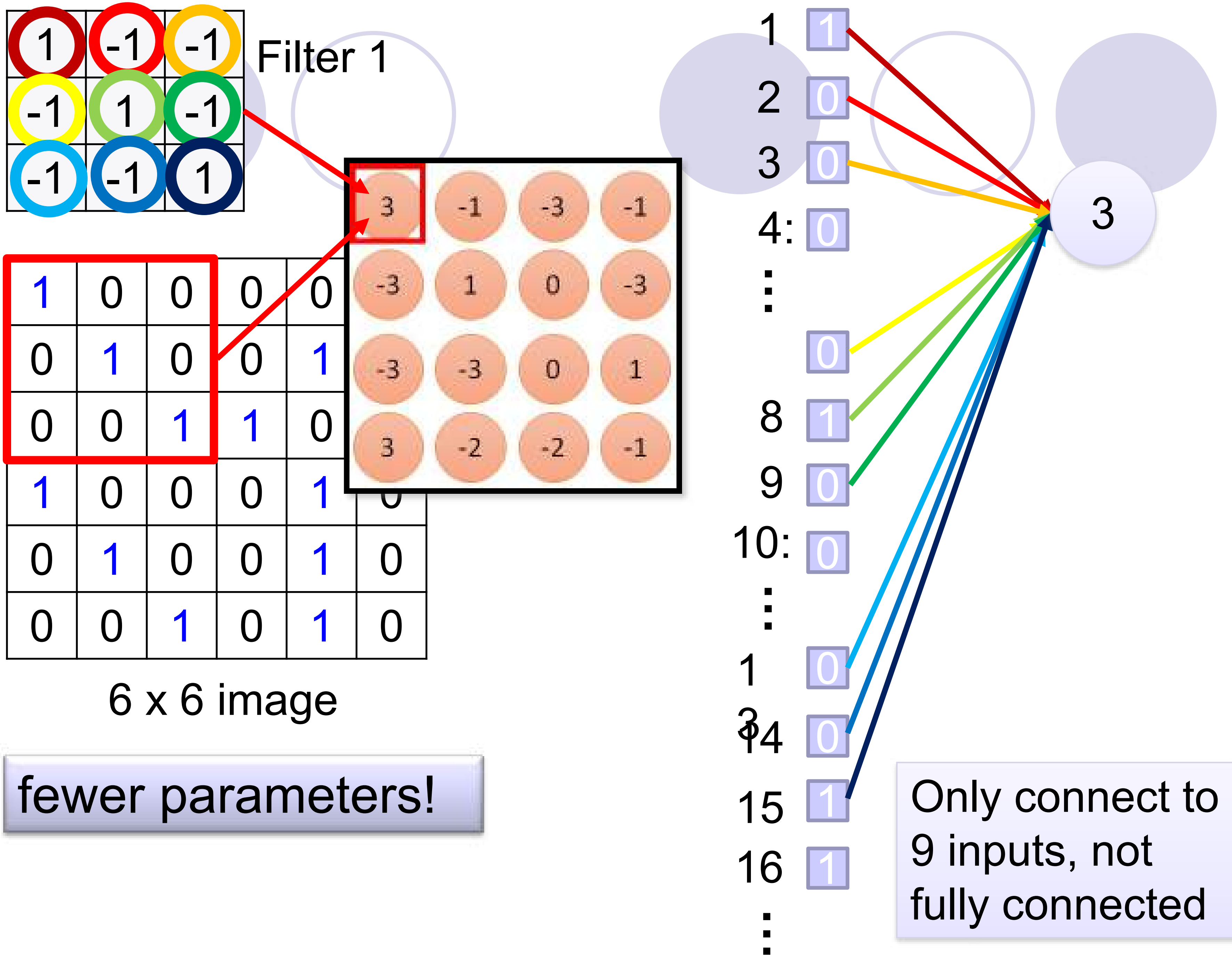


1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

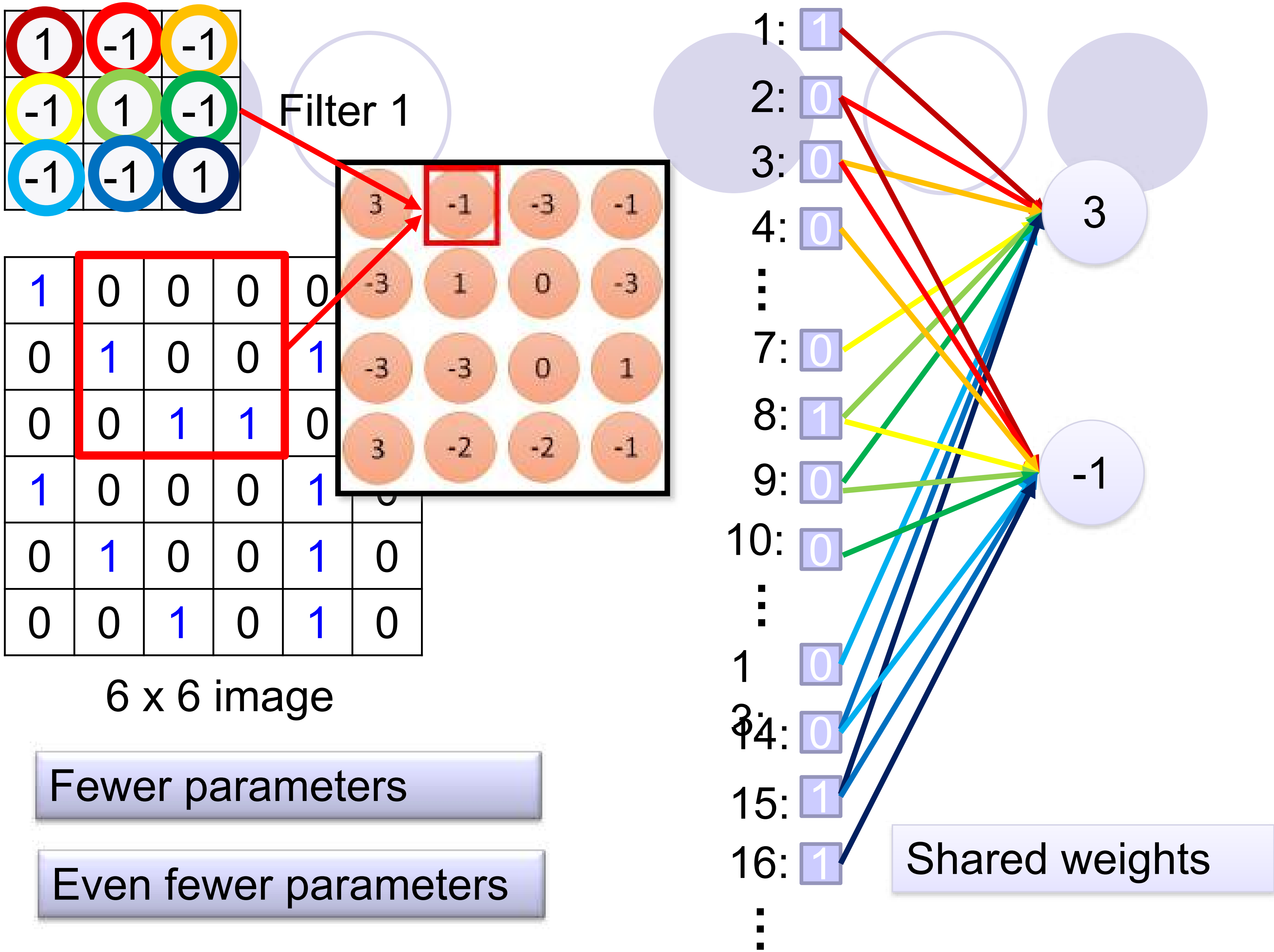
Convolution v.s. Fully Connected



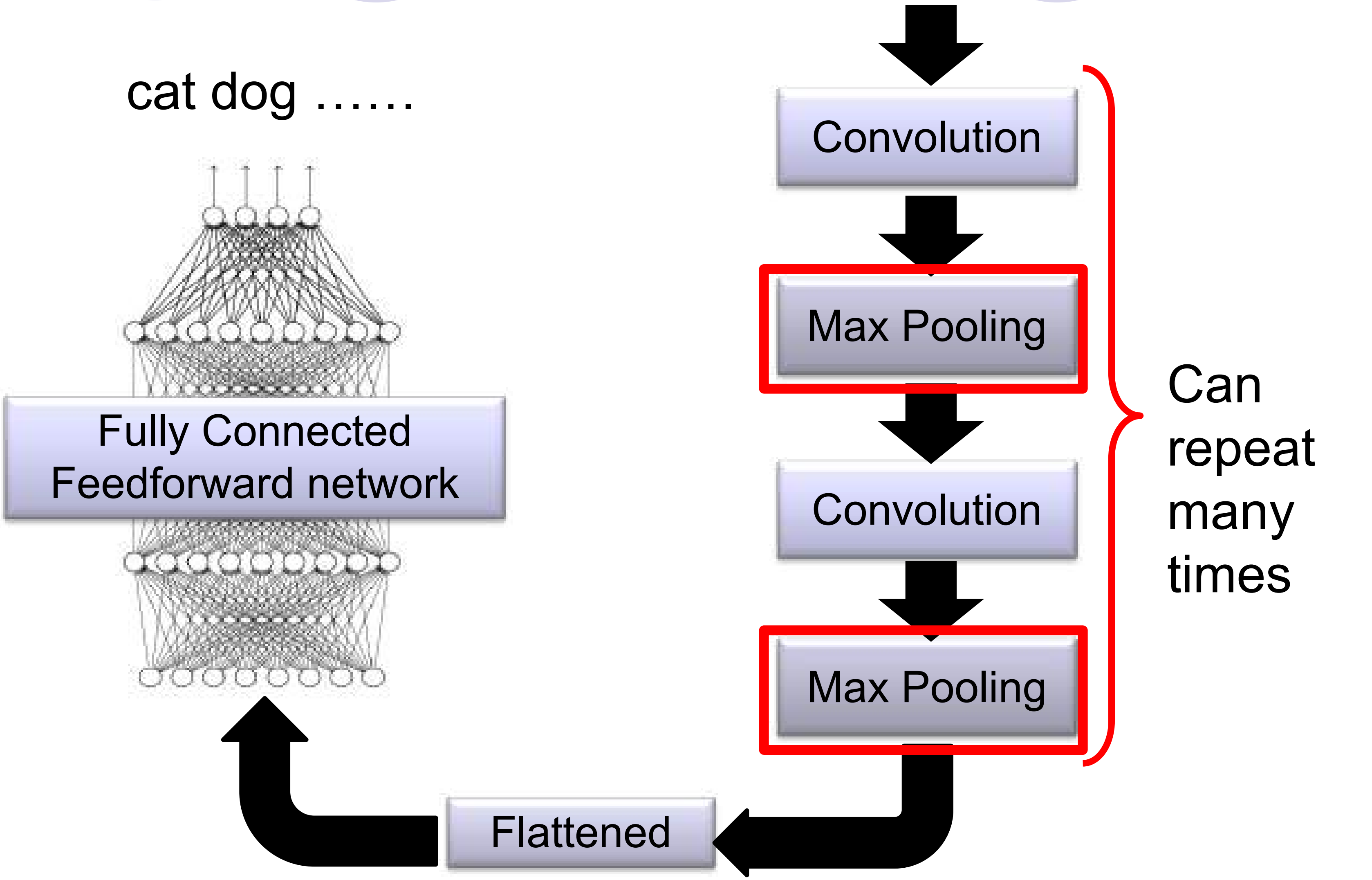




Filter 1
6 x 6 image
fewer parameters!
Only connect to 9 inputs, not fully connected


Filter 1
6 x 6 image
Fewer parameters
Even fewer parameters
Shared weights
3
-1
1:
2:
3:
4:
7:
8:
9:
10:
13:
14:
15:
16:

The whole CNN



Max Pooling

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

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

-1	-1	-1	-1
-1	-1	-2	1
-1	-1	-2	1
-1	0	-4	3

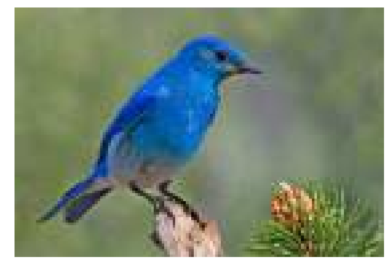
Why Pooling

- Subsampling pixels will not change the object

bird

Subsampling

bird

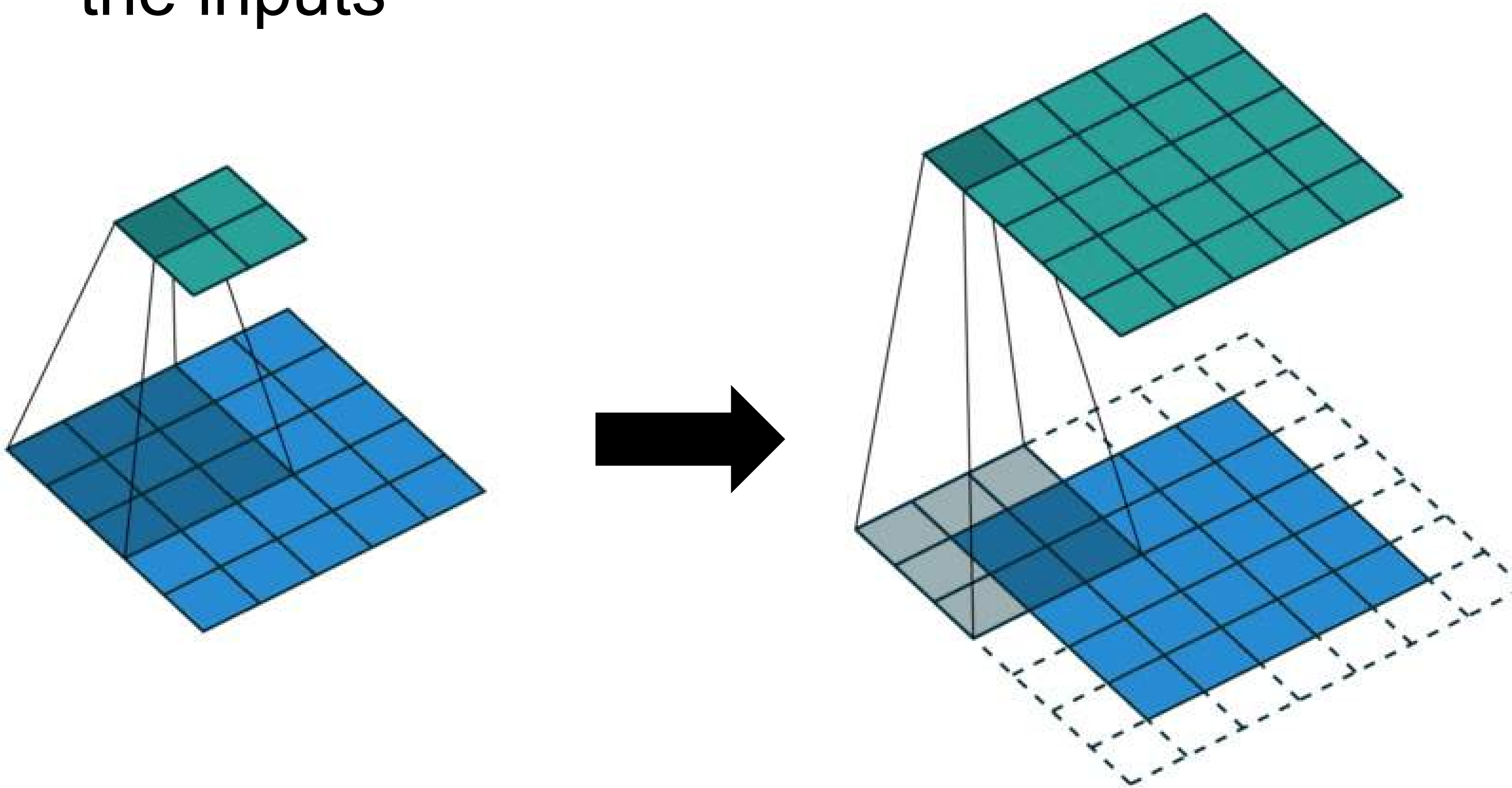


We can subsample the pixels to make image smaller

$\Rightarrow$ fewer parameters to characterize the image

Padding

- We do not want to lose any information from the inputs



Batch Normalization

When training NN, we usually normalize the image data before feeding to the NN. Lets say we have 40 millions of dataset. normalizing these amount data might be not of a good use. So we use batch normalization.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

A CNN compresses a fully connected network in two ways:

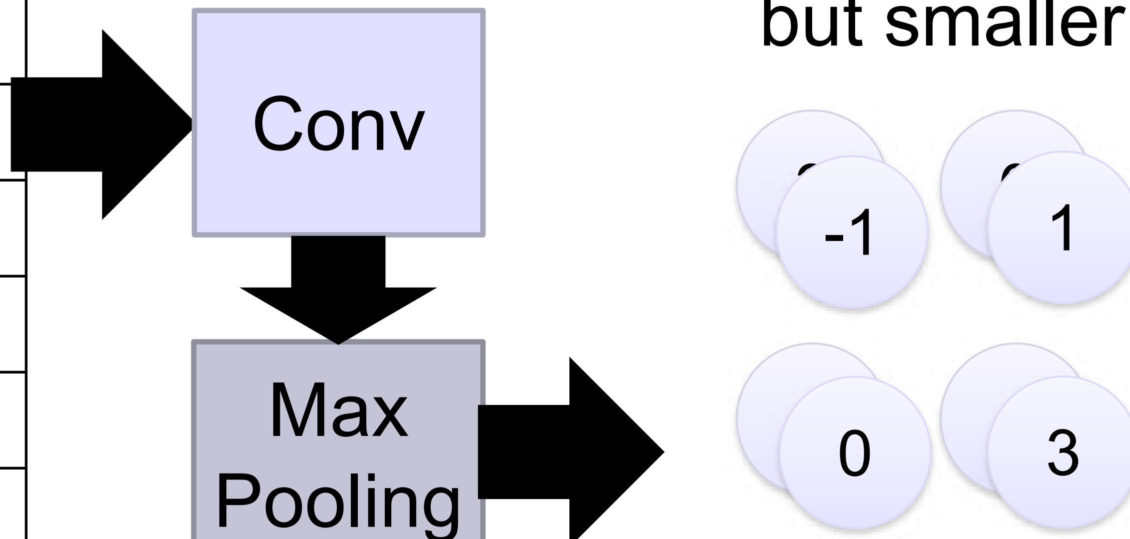
- Reducing number of connections
- Shared weights on the edges
- Max pooling further reduces the complexity

Max Pooling

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

6 x 6 image

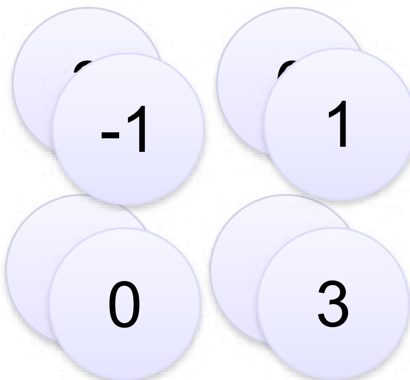
New image but smaller



2 x 2 image

Each filter is a channel

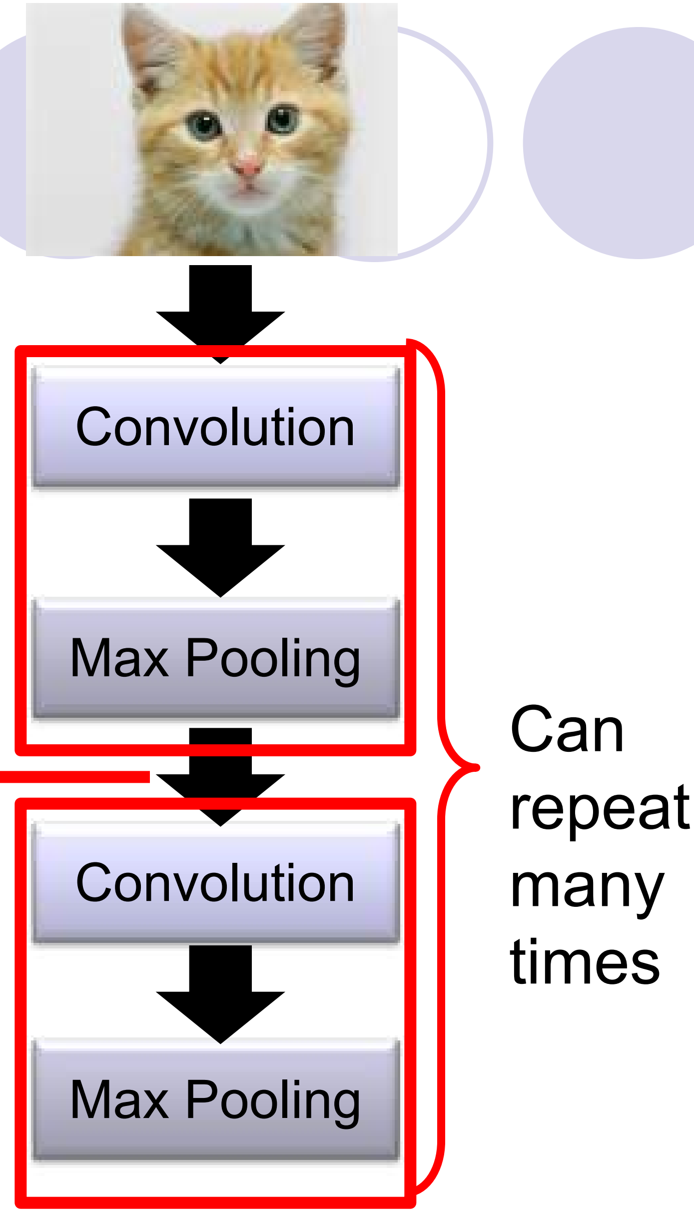
The whole CNN



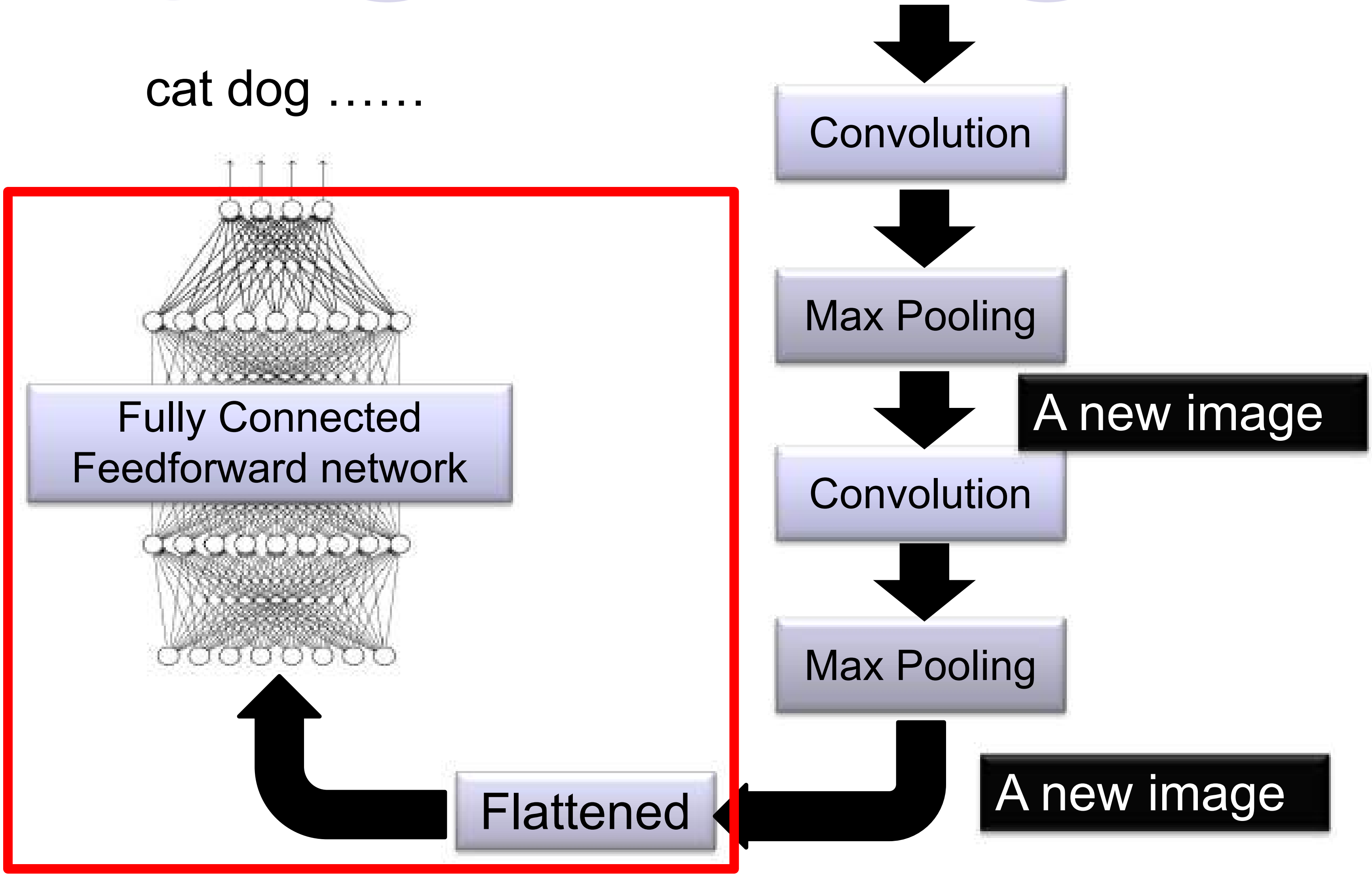
A new image

Smaller than the original image

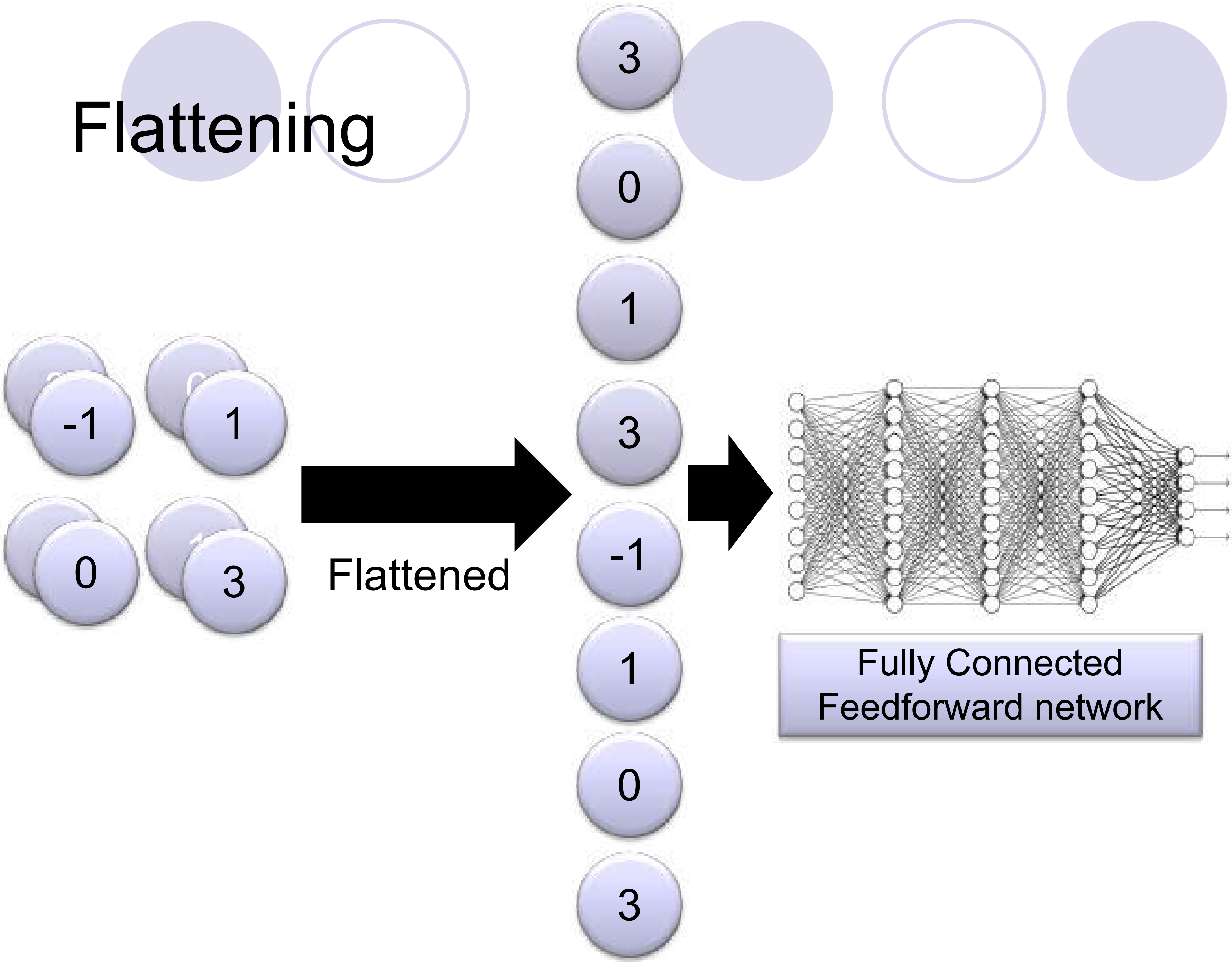
The number of channels is the number of filters



The whole CNN



Flattening

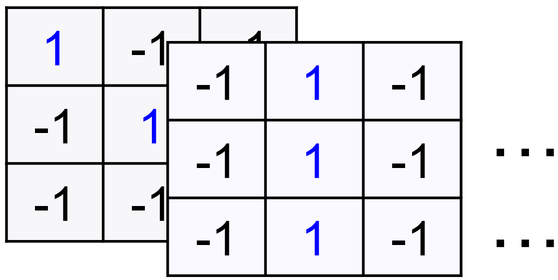


-1
1
0
3
Flattened
3
0
1
3
-1
1
0
3
Fully Connected
Feedforward network

CNN in Keras

Only modified the ***network structure*** and ***input format (vector -> 3-D tensor)***

```
model2.add( Convolution2D( 25,3,3,
            input_shape=(28,28,1)) )
```



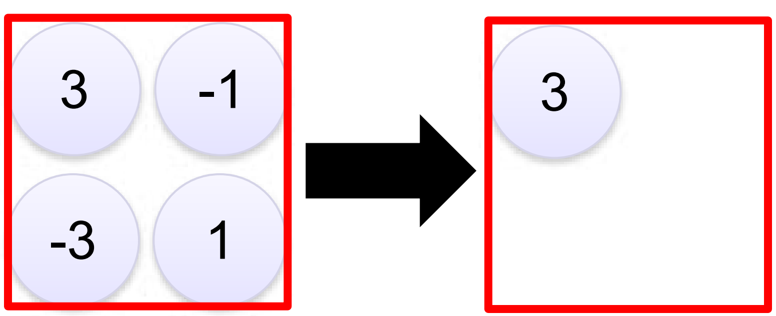
There are 25 3x3 filters.

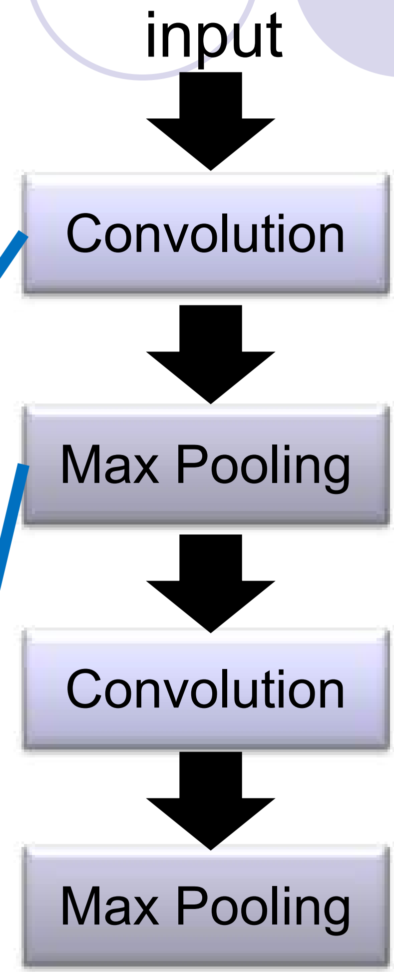
Input_shape = (28 , 28 , 1)

28 x 28 pixels

1: black/white, 3: RGB

```
model2.add(MaxPooling2D((2,2)))
```





CNN in Keras

Only modified the ***network structure*** and ***input format (vector -> 3-D array)***

Exam alert! - compute the number of parameters in each layer and their output.

Input

1 x 28 x 28

Convolution

```
model2.add( Convolution2D( 25,3,3,
            input_shape=(28,28,1)) )
```

How many parameters for each filter? 9

25 x 26 x 26

Max Pooling

```
model2.add(MaxPooling2D((2,2)))
```

25 x 13 x 13

Convolution

```
model2.add(Convolution2D(50,3,3))
```

How many parameters for each filter? 225= 25x9

50 x 11 x 11

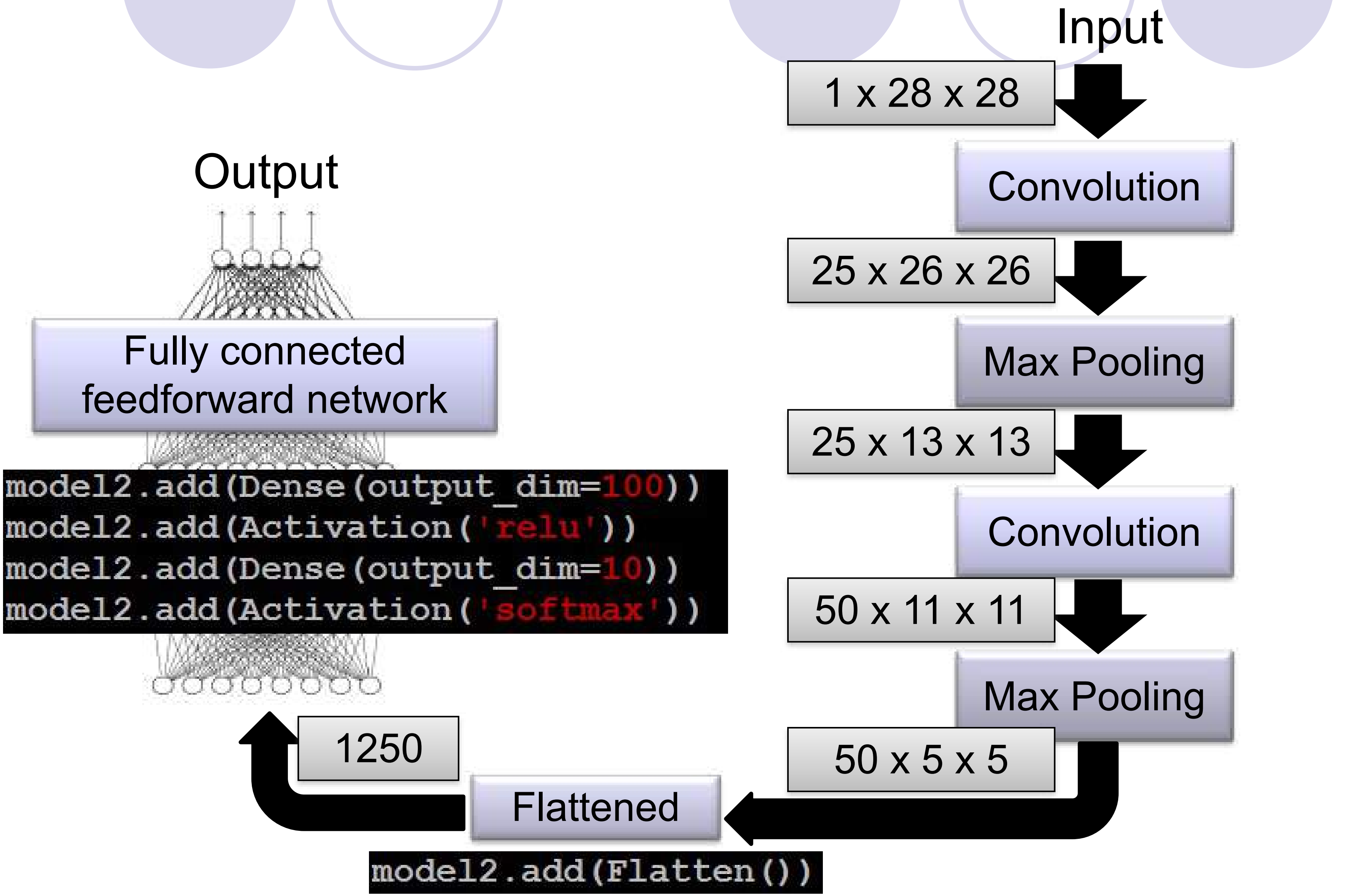
Max Pooling

```
model2.add(MaxPooling2D((2,2)))
```

50 x 5 x 5

CNN in Keras

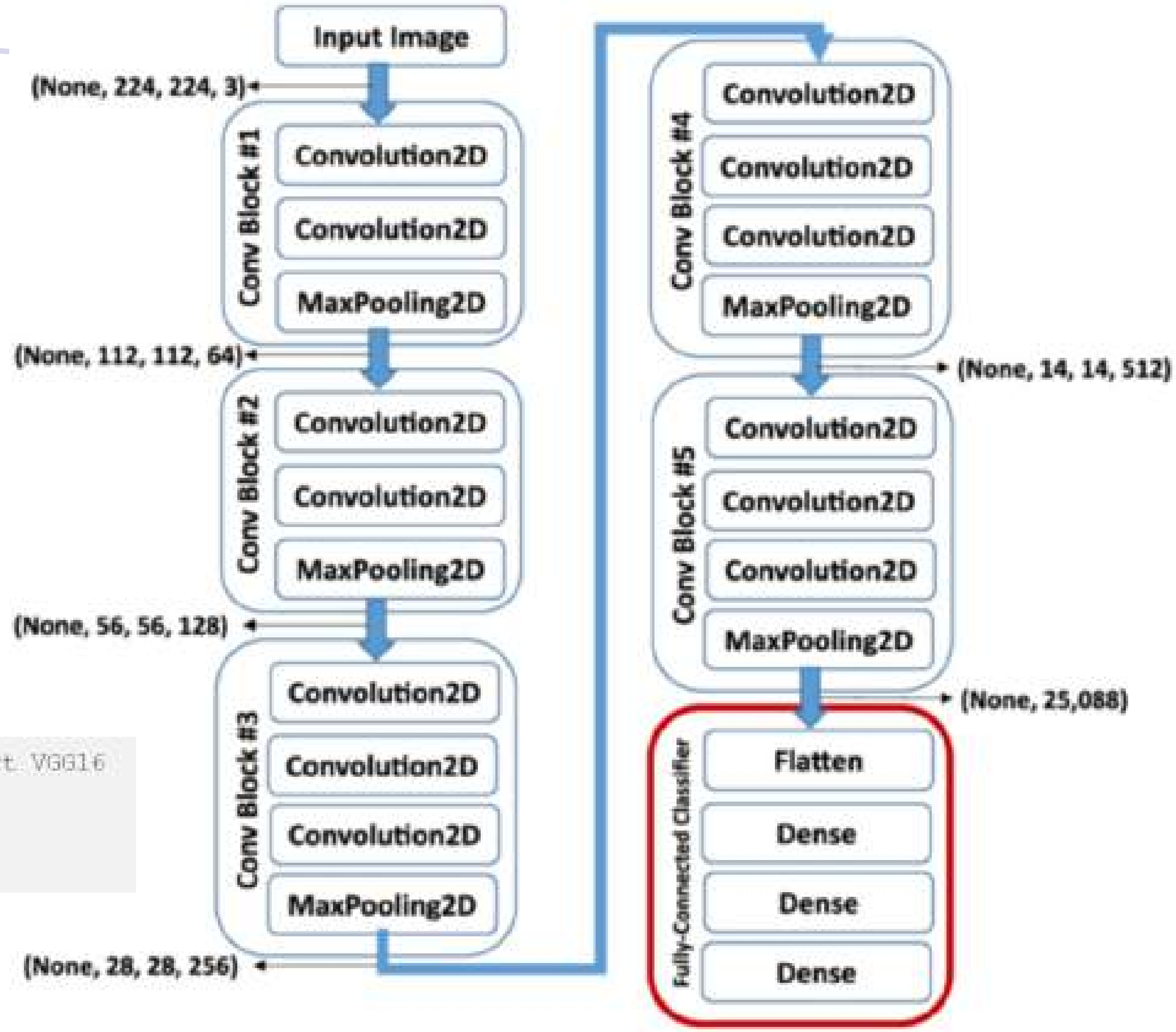
Only modified the ***network structure*** and ***input format (vector -> 3-D array)***




```
model2.add(Flatten())
```

```
model2.add(Dense(output_dim=100))
model2.add(Activation('relu'))
model2.add(Dense(output_dim=10))
model2.add(Activation('softmax'))
```

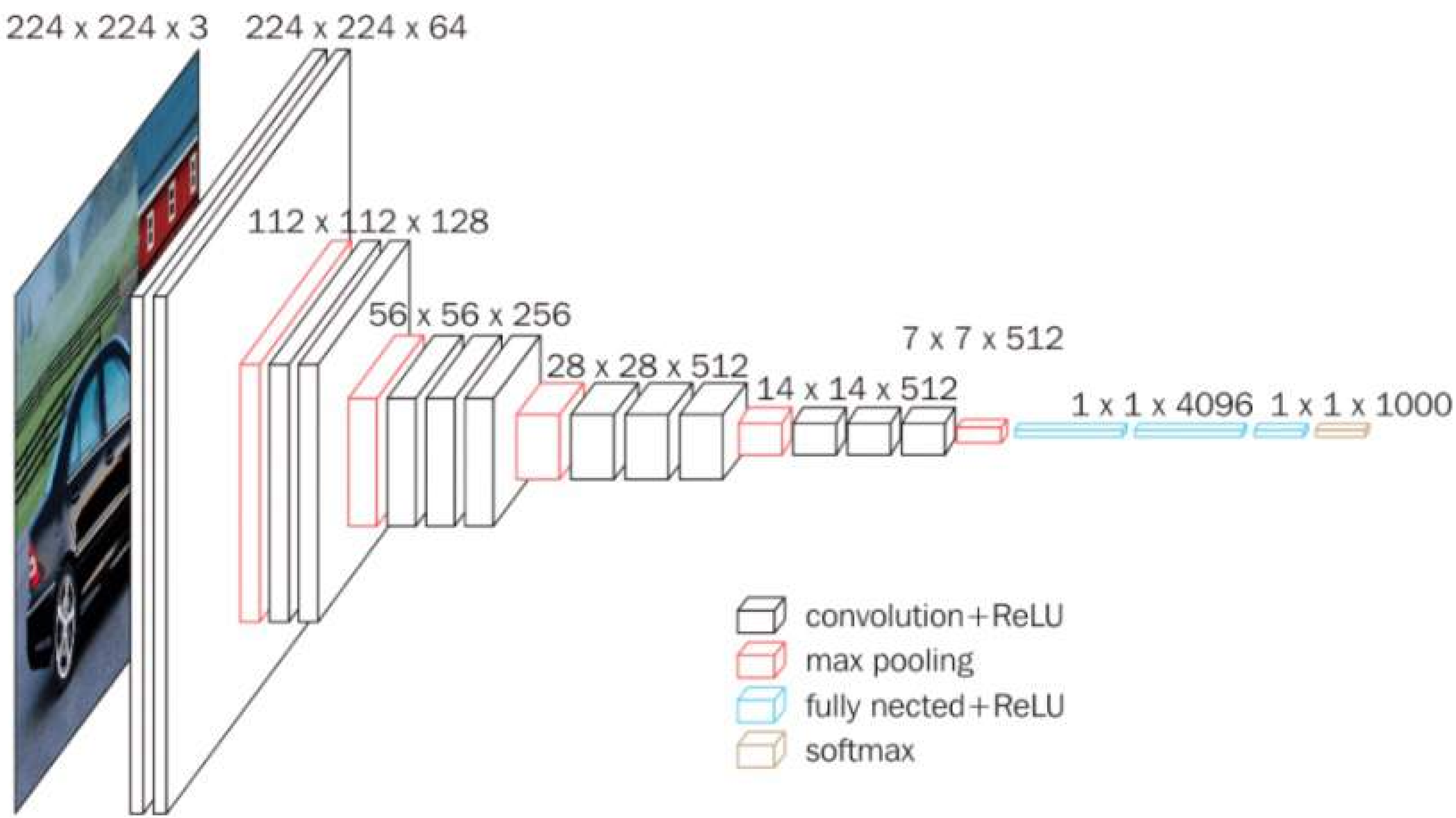
VGG16




```
from keras.applications.vgg16 import VGG16

#build model
mod = VGG16()
```

VGG16

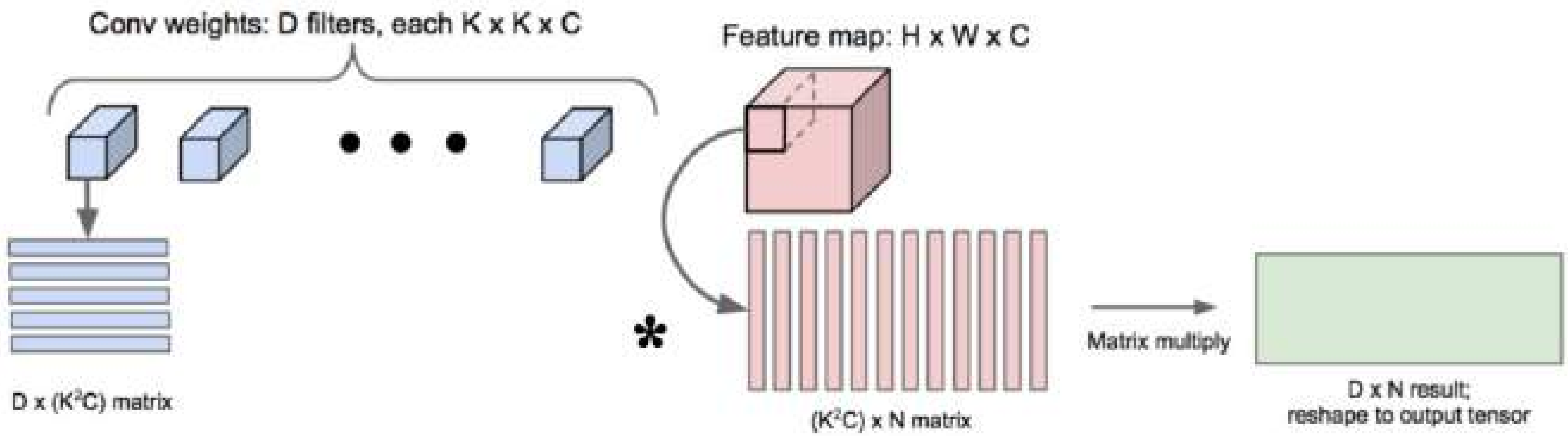


224 x 224 x 3
224 x 224 x 64
112 x 112 x 128
56 x 56 x 256
28 x 28 x 512
14 x 14 x 512
7 x 7 x 512
1 x 1 x 4096
1 x 1 x 1000
convolution+ReLU
max pooling
fully nected+ReLU
softmax

Additional information regarding forward and backward

https://www.youtube.com/watch?v=f0t-OCG79-U

Additional information regarding forward and backward



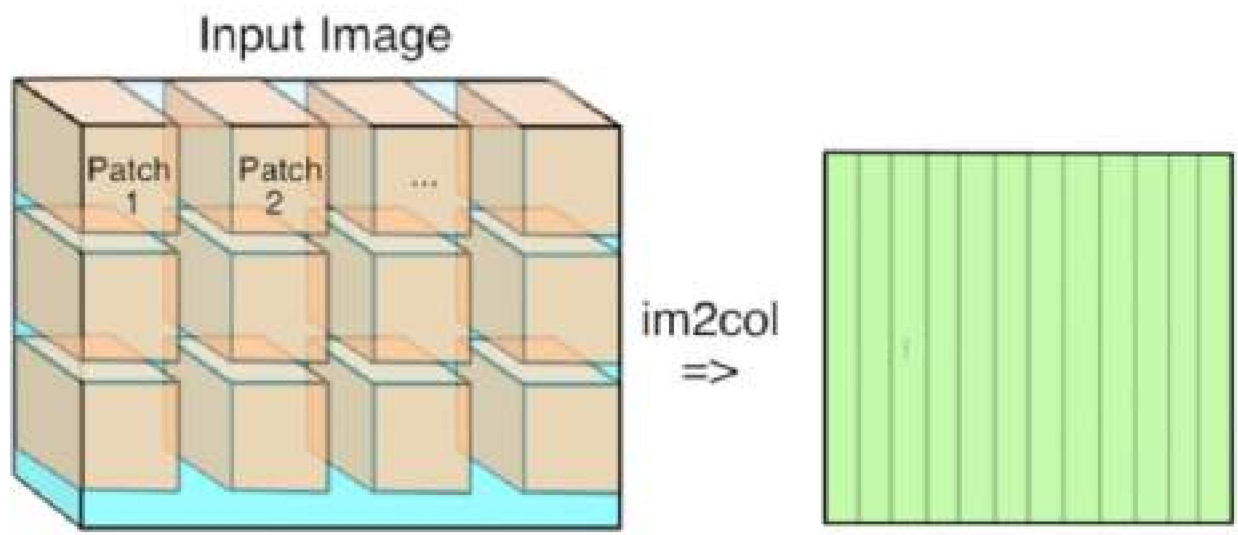
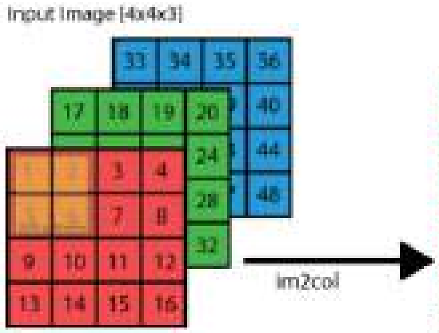
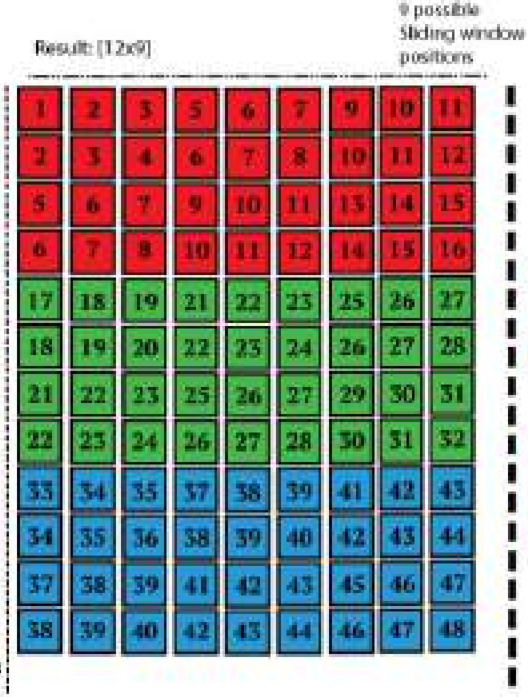


Image to column operation (im2col)
Slide the input image like a convolution but each patch become a column vector.



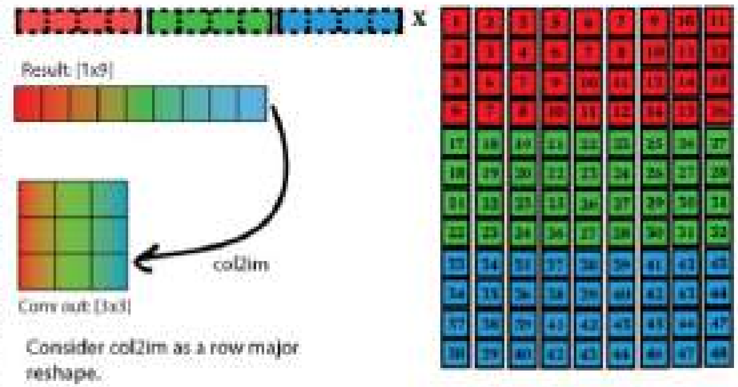


Kernel Width:2
Kernel Height:2
Stride:1.
Padding:0

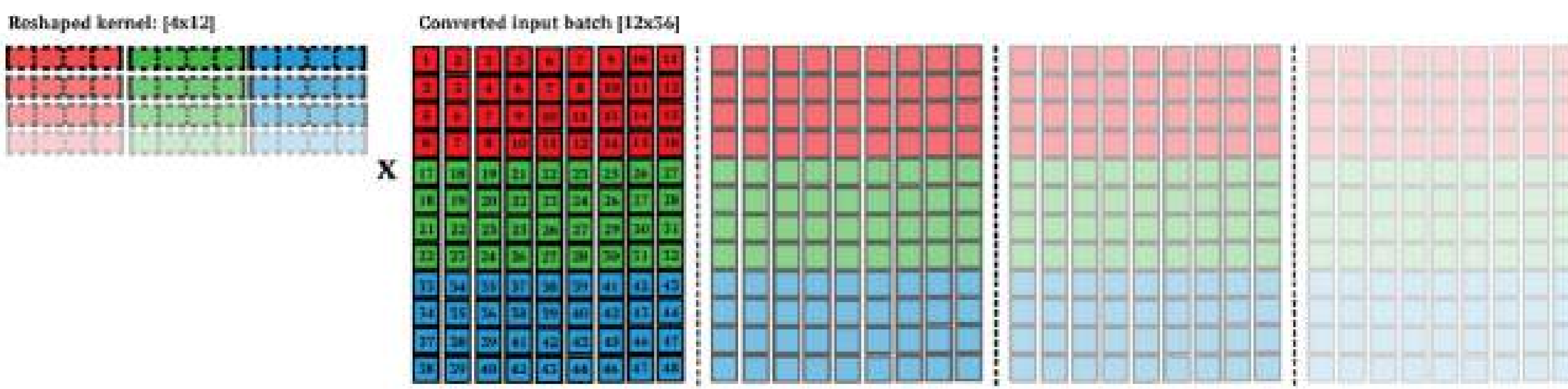
W_out=(W_in - kW + 2*P)/S + 1
H_out=(H_in - kH + 2*P)/S + 1

W_out=(4-2)/1+1=3
H_out=(4-2)/1+1=3

We can multiply this result matrix [12x9]
with a kernel [1x12].
result = kernel x matrix
The result would be a row vector [1x9].
We need another operation that will convert
this row vector into a image [3x3].



We get true performance gain
when the kernel has a large number of filters, ie: F=4
and/or you have a batch of images (N=4). Example for the input batch [4x4x3x4], convolved with 4 filters [2x2x3x2] .
The only problem with this approach is the amount of memory



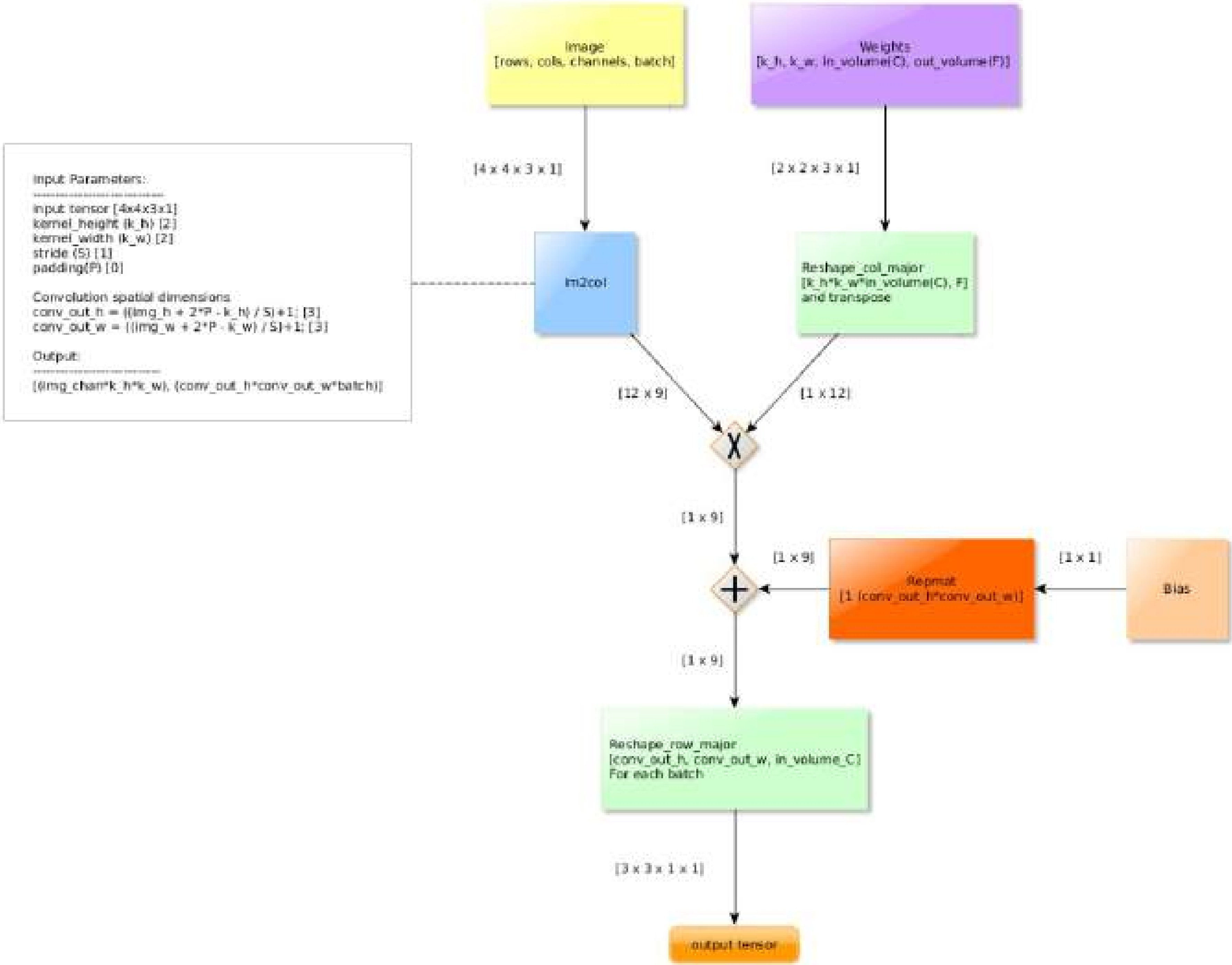
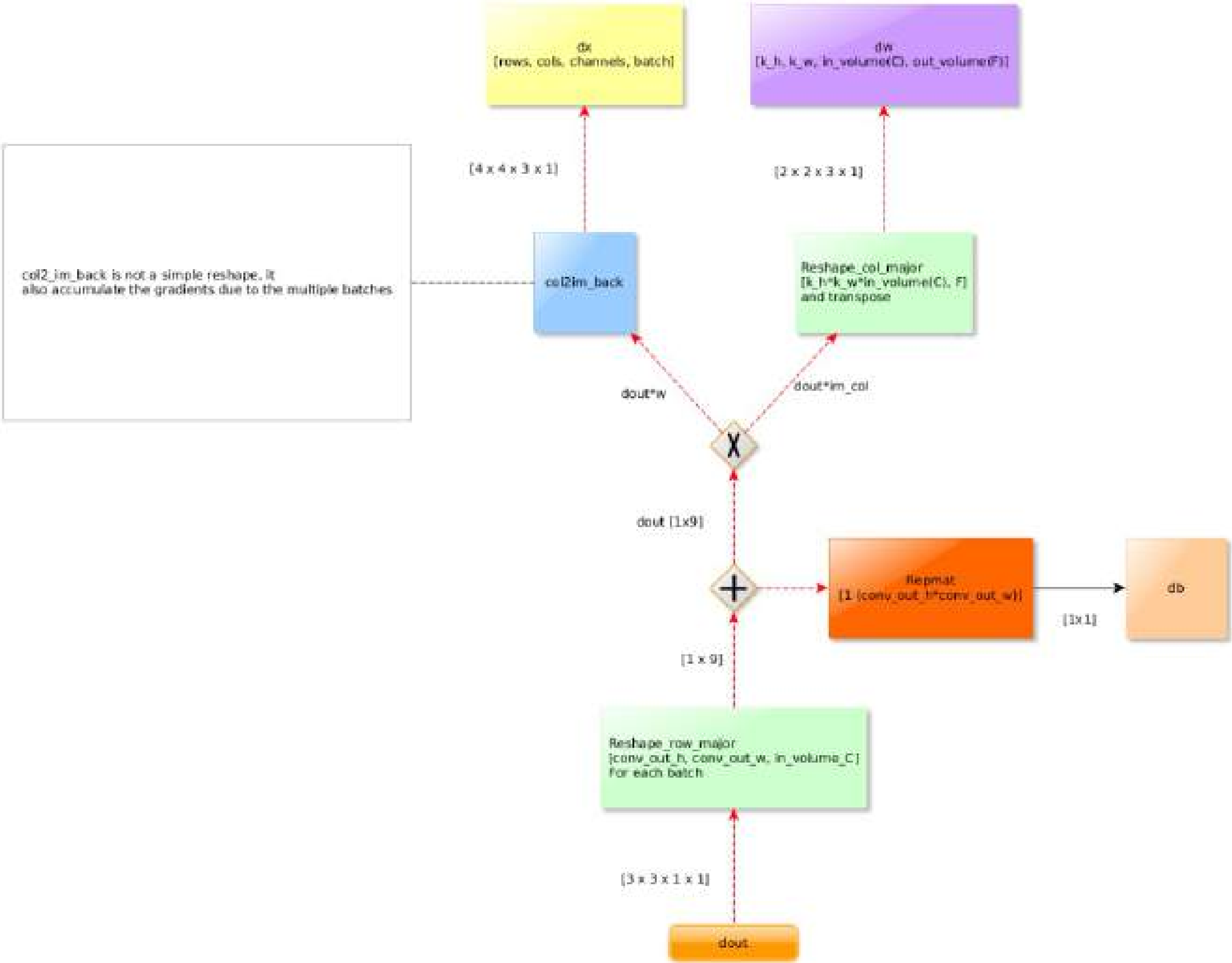
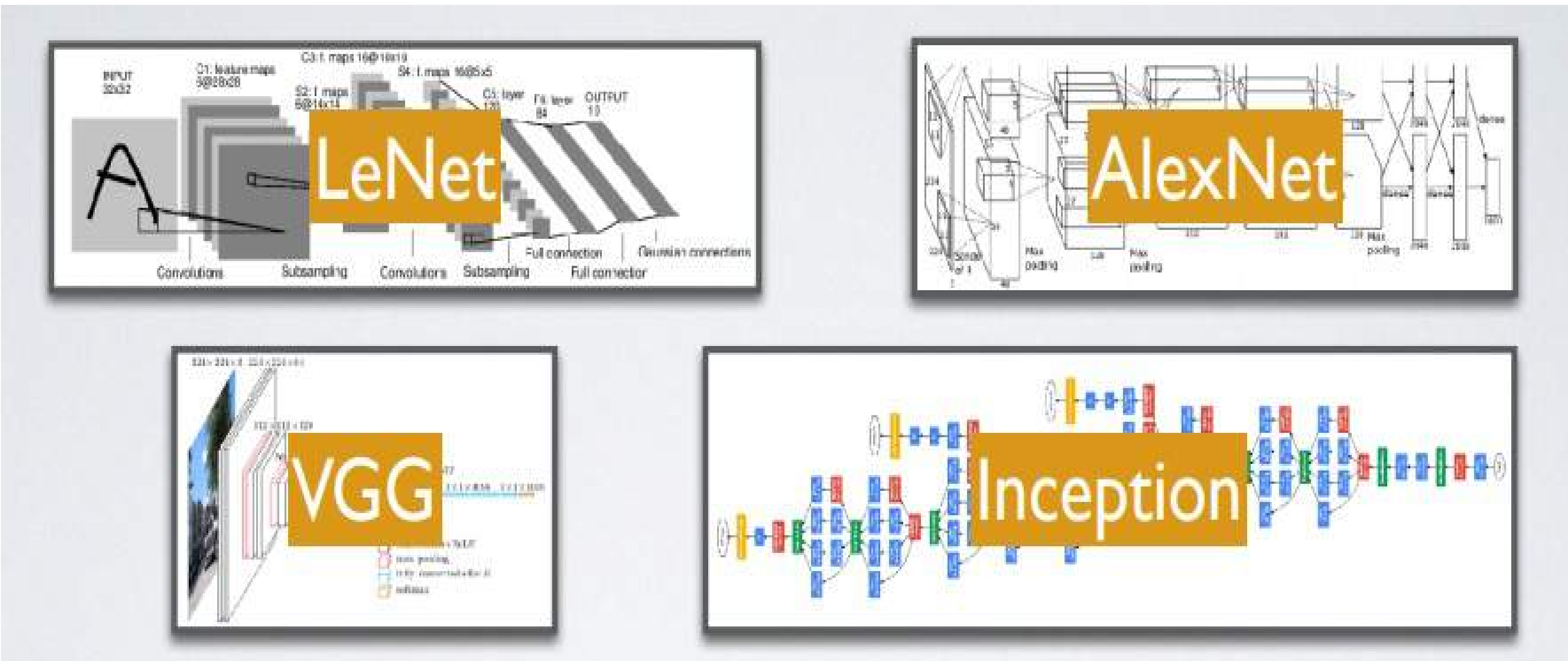


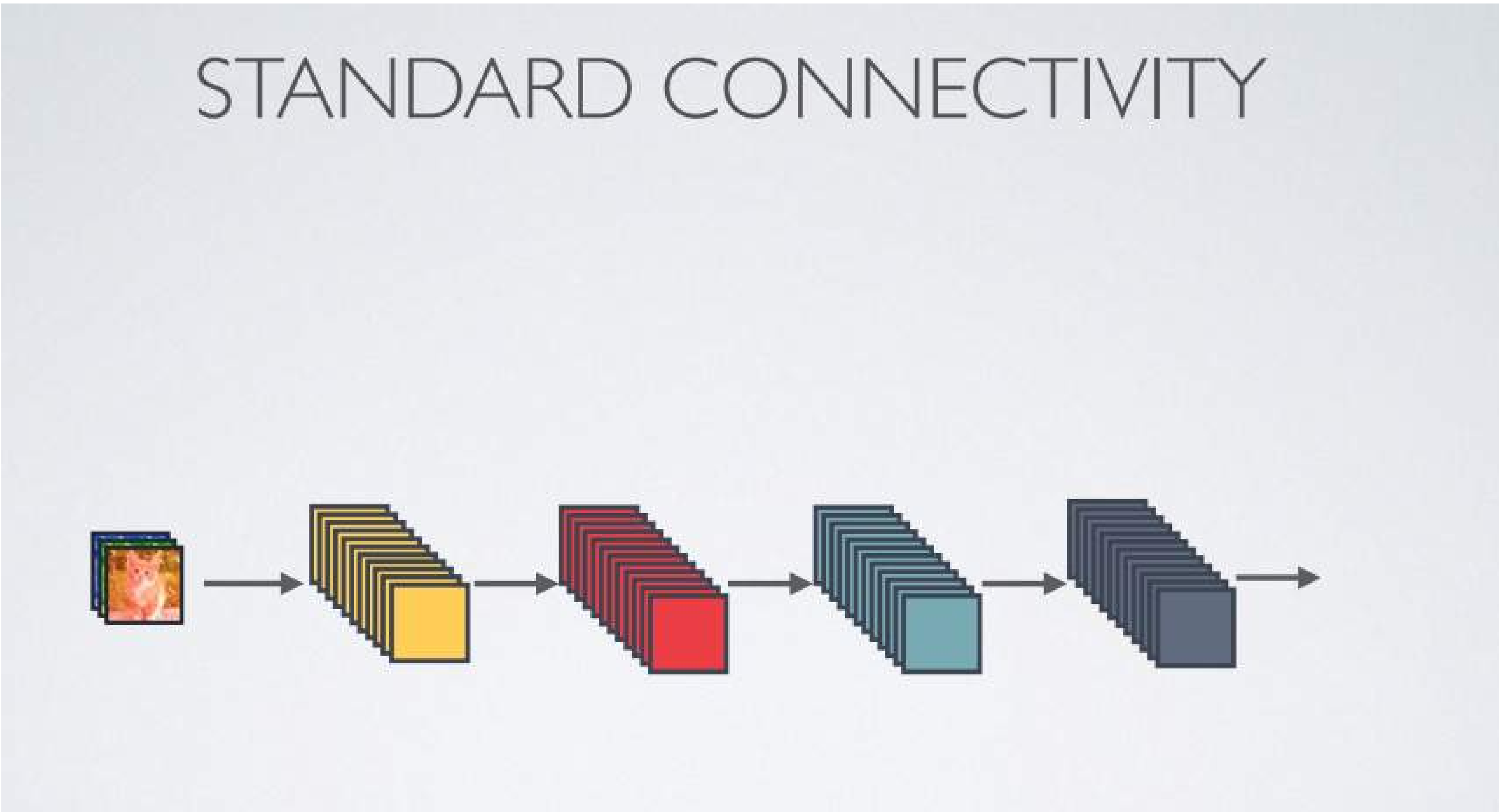
Image
[rows, cols, channels, batch]
Weights
[k_h, k_w, in_volume(C), out_volume(F)]
Input Parameters:
input tensor [4x4x3x1]
kernel_height (k_h) [2]
kernel_width (k_w) [2]
stride (S) [1]
padding(P) [0]
Convolution spatial dimensions
conv_out_h = ((img_h + 2*P - k_h) / S)+1; [3]
conv_out_w = ((img_w + 2*P - k_w) / S)+1; [3]
Output:
[(img_chan*k_h*k_w), (conv_out_h*conv_out_w*batch)]
[4 x 4 x 3 x 1]
[2 x 2 x 3 x 1]
im2col
Reshape_col_major
[k_h*k_w*in_volume(C), F]
and transpose
[12 x 9]
[1 x 12]
X
[1 x 9]
[1 x 9]
Repmat
[1 (conv_out_h*conv_out_w)]
[1 x 1]
Bias
[1 x 9]
Reshape_row_major
[conv_out_h, conv_out_w, in_volume_C]
For each batch
[3 x 3 x 1 x 1]
output tensor


dx
[rows, cols, channels, batch]
dw
[k_h, k_w, in_volume(C), out_volume(F)]
[4 x 4 x 3 x 1]
[2 x 2 x 3 x 1]
col2_im_back is not a simple reshape. It
also accumulate the gradients due to the multiple batches
col2im_back
Reshape_col_major
[k_h*k_w*in_volume(C), F]
and transpose
dout*w
dout*im_col
X
dout [1x9]
Repmat
[1 (conv_out_h*conv_out_w)]
db
[1x1]
[1 x 9]
Reshape_row_major
[conv_out_h, conv_out_w, in_volume_C]
For each batch
[3 x 3 x 1 x 1]
dout

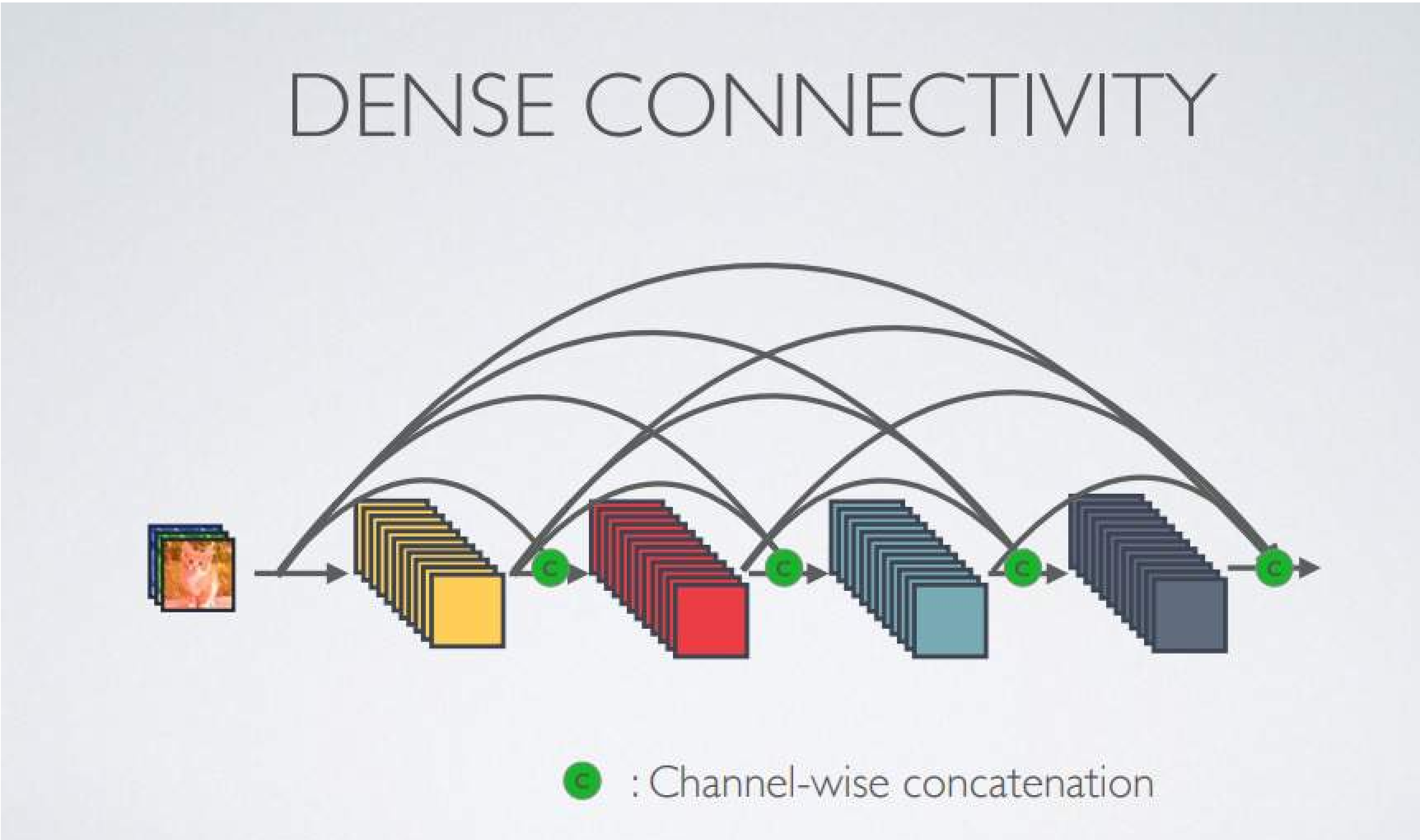
DenseNet


LeNet
AlexNet
VGG
Inception
Convolutions
Subsampling
Convolutions
Subsampling
Full connection
Full connection
OUTPUT

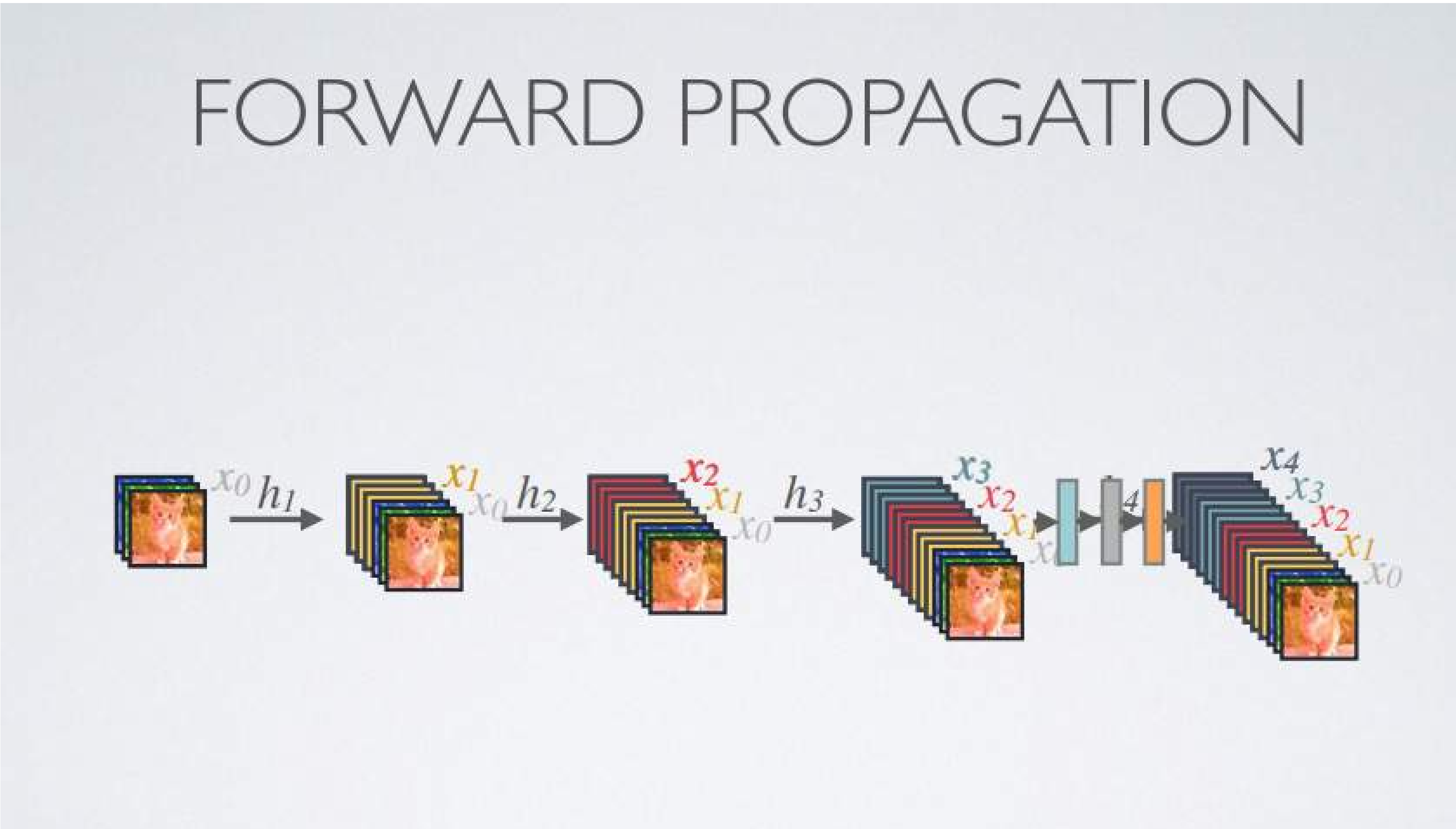
DenseNet



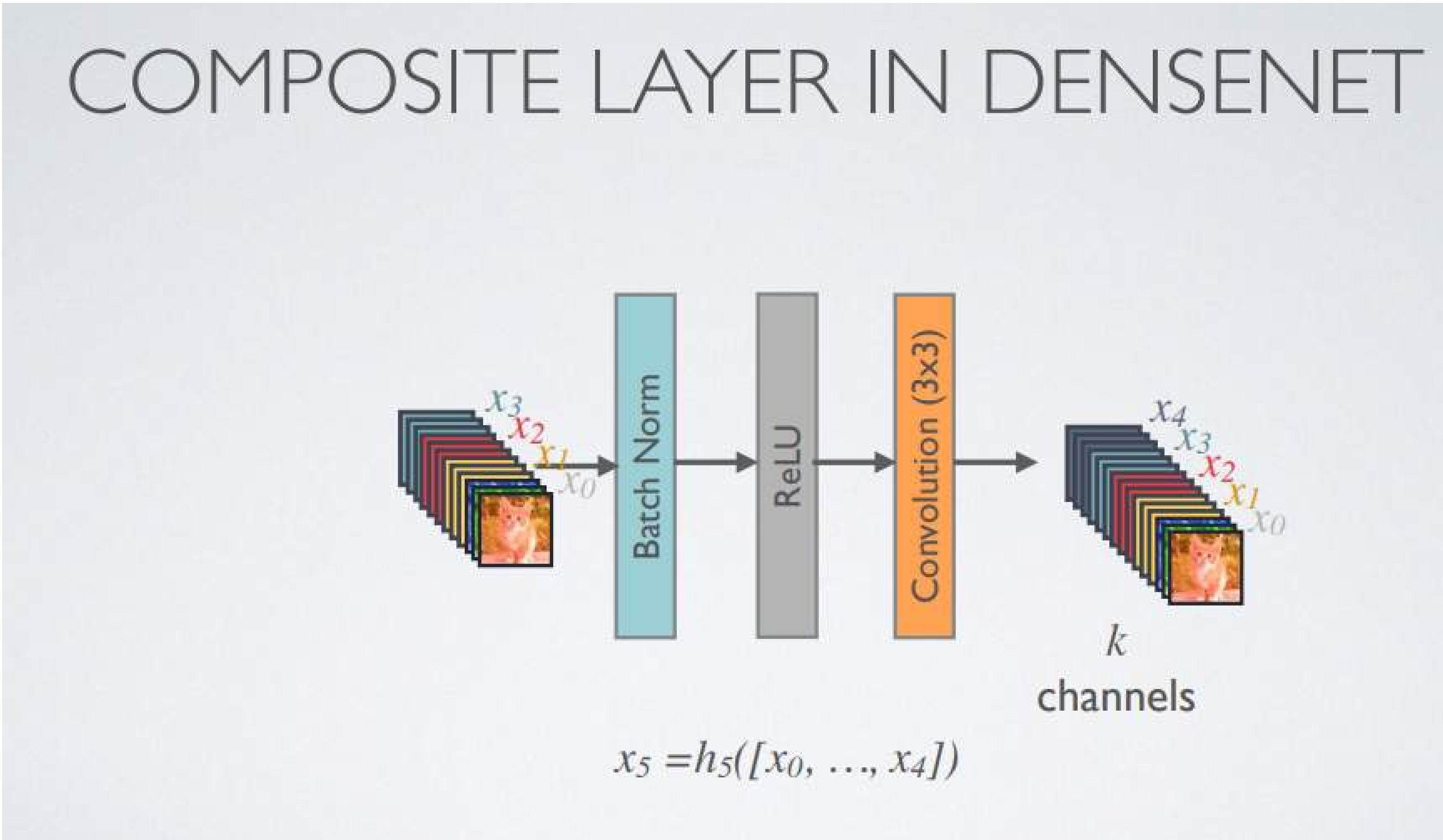
DenseNet



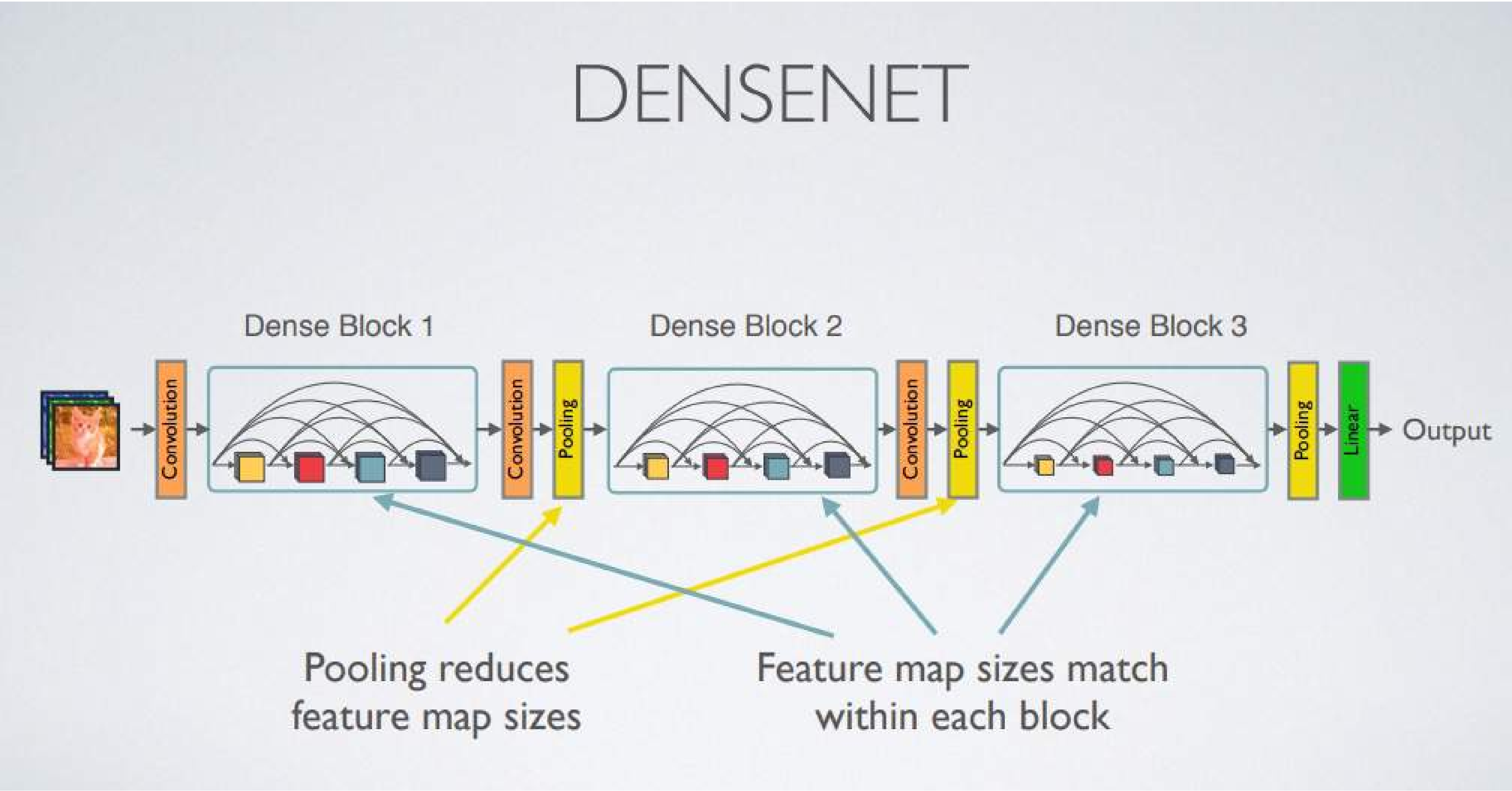
DenseNet



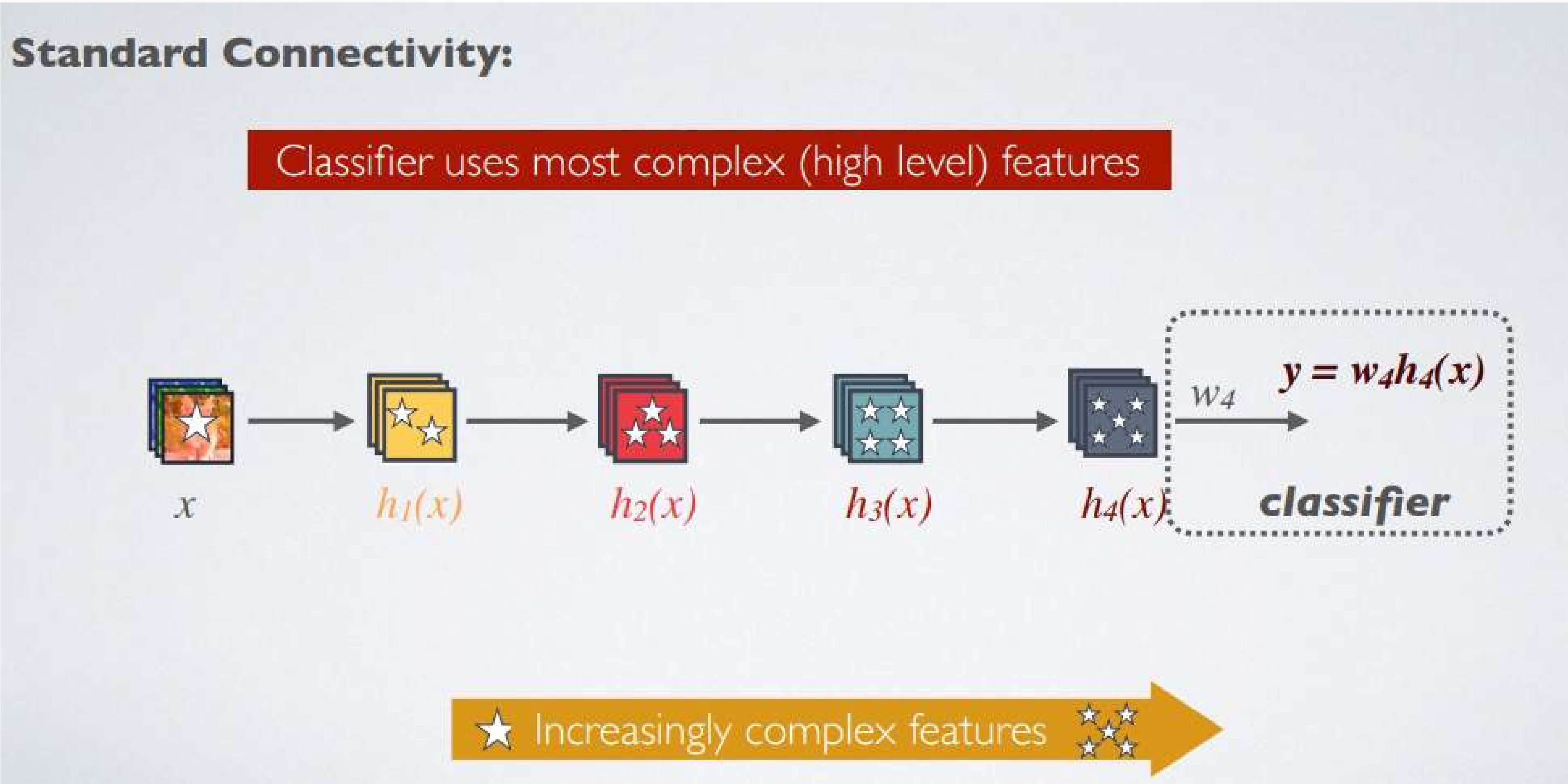
DenseNet


COMPOSITE LAYER IN DENSENET
x3
x2
x1
x0
Batch Norm
ReLU
Convolution (3x3)
x4
x3
x2
x1
x0
k
channels
x5 =h5([x0, …, x4])

DenseNet

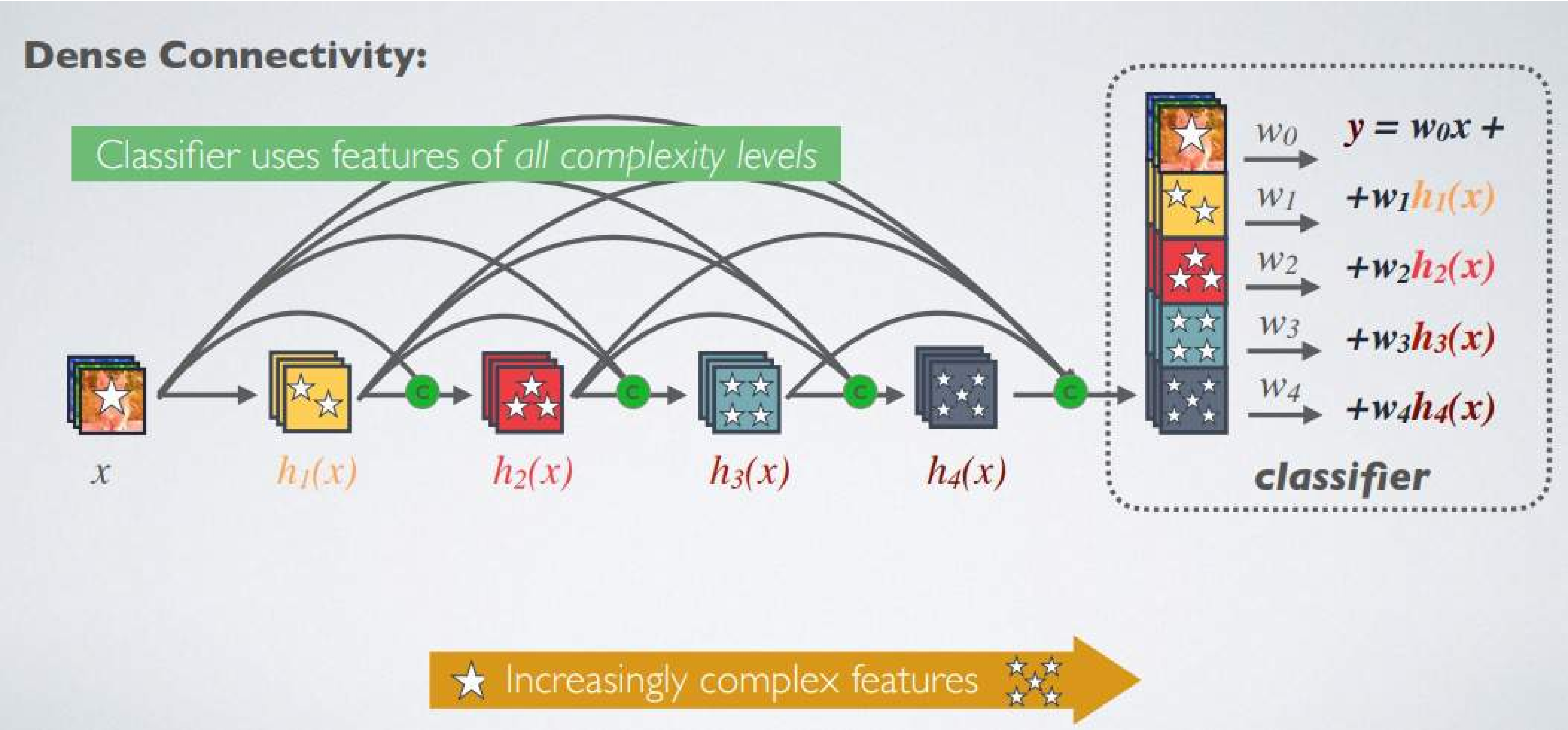

DENSENET
Dense Block 1
Dense Block 2
Dense Block 3
Convolution
Convolution
Pooling
Convolution
Pooling
Pooling
Linear
Output
Pooling reduces feature map sizes
Feature map sizes match within each block

DenseNet



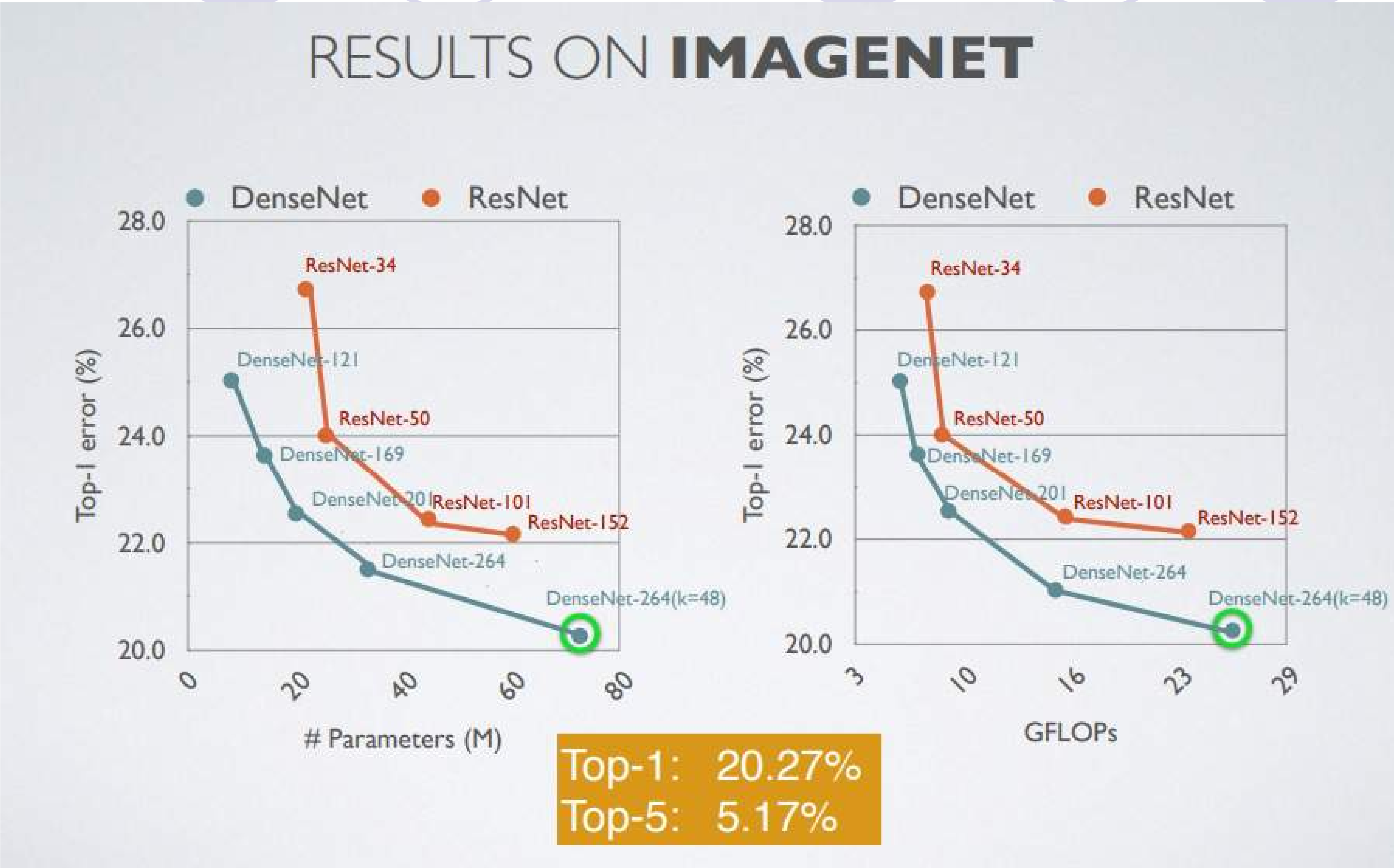
Standard Connectivity:
Classifier uses most complex (high level) features
x
$h_1(x)$
$h_2(x)$
$h_3(x)$
$h_4(x)$
w_4
$y = w_4 h_4(x)$
classifier
Increasingly complex features

DenseNet



Dense Connectivity:
Classifier uses features of *all complexity levels*
x
h1(x)
h2(x)
h3(x)
h4(x)
w0
w1
w2
w3
w4
y = w0x +
+w1h1(x)
+w2h2(x)
+w3h3(x)
+w4h4(x)
classifier
Increasingly complex features

DenseNet


RESULTS ON IMAGENET
DenseNet
ResNet
28.0
26.0
24.0
22.0
20.0
Top-1 error (%)
ResNet-34
ResNet-50
ResNet-101
ResNet-152
DenseNet-121
DenseNet-169
DenseNet-201
DenseNet-264
DenseNet-264(k=48)
0
20
40
60
80
Parameters (M)
3
10
16
23
29
GFLOPs
Top-1: 20.27%
Top-5: 5.17%