

# **Deep Neural Networks**

## **Convolutional Networks II**

**Bhiksha Raj**

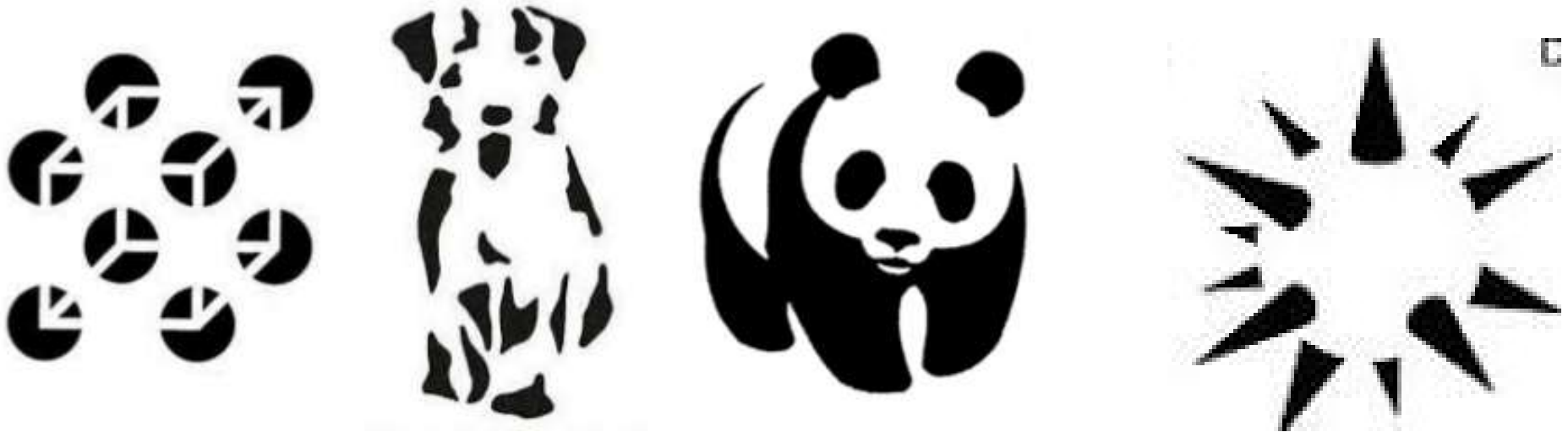
**Spring 2022**

# Story so far

- Pattern classification tasks such as “does this picture contain a cat”, or “does this recording include HELLO” are best performed by scanning for the target pattern
- Scanning an input with a network and combining the outcomes is equivalent to scanning with individual neurons hierarchically
  - First level neurons scan the input
  - Higher-level neurons scan the “maps” formed by lower-level neurons
  - A final “decision” unit or subnetwork makes the final decision
- Deformations in the input can be handled by “pooling”
- For 2-D (or higher-dimensional) scans, the structure is called a convolutional network
- For 1-D scan along time, it is called a Time-delay neural network



# A little history



- How do animals see?
  - What is the neural process from eye to recognition?
- Early research:
  - largely based on behavioral studies
    - Study behavioral judgment in response to visual stimulation
    - Visual illusions
  - and gestalt
    - Brain has innate tendency to organize disconnected bits into whole objects
  - But no real understanding of how the brain processed images

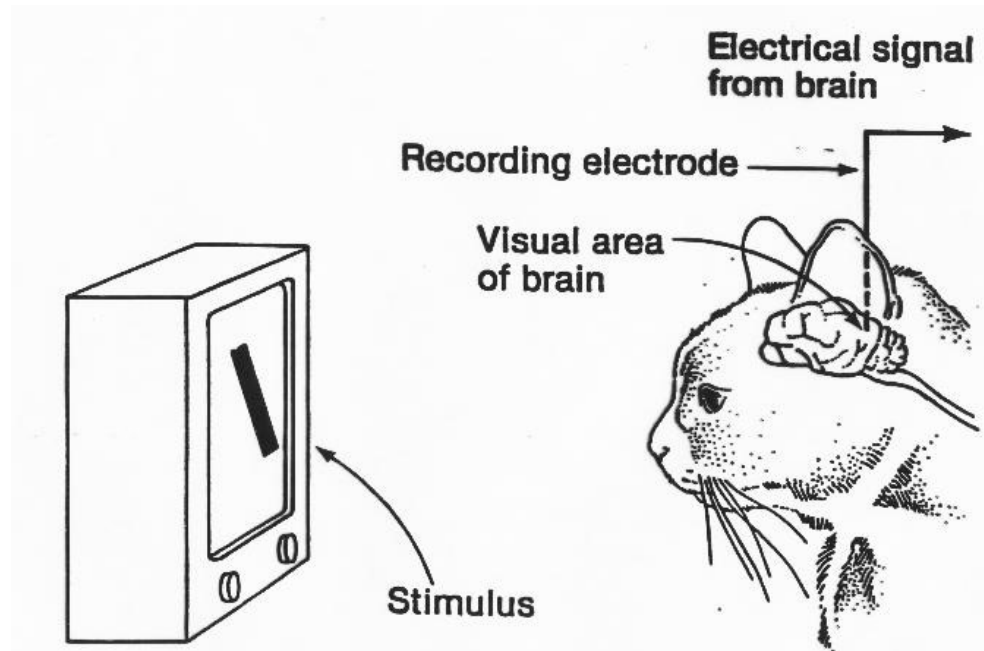
# Hubel and Wiesel 1959



- First study on neural correlates of vision.
  - “Receptive Fields in Cat Striate Cortex”
    - “Striate Cortex”: Approximately equal to the V1 visual cortex
      - “Striate” – defined by structure, “V1” – functional definition
- 24 cats, anaesthetized, immobilized, on artificial respirators
  - Anaesthetized with truth serum
  - Electrodes into brain
    - Do not report if cats survived experiment, but claim brain tissue was studied

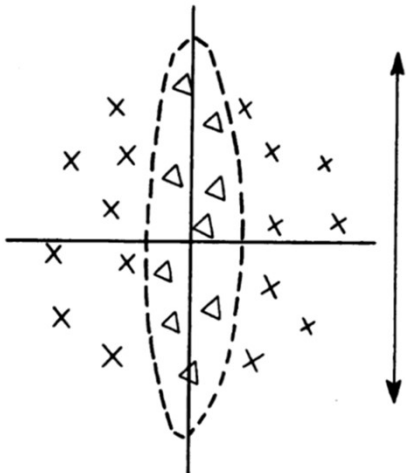


# Hubel and Wiesel 1959

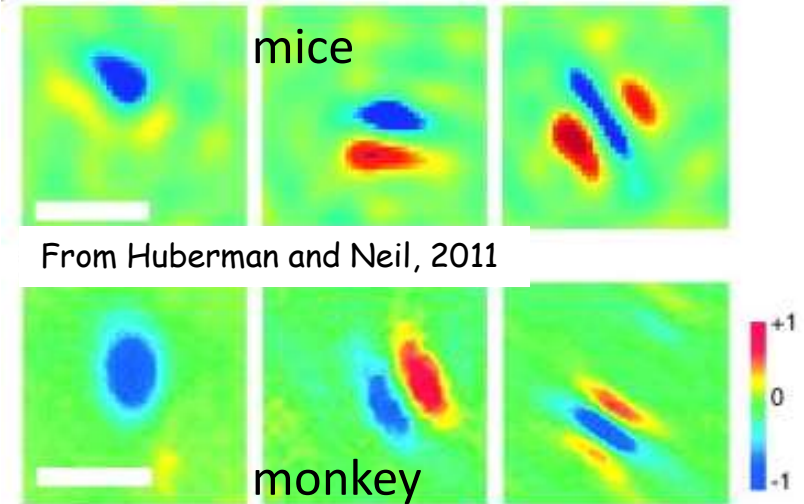


- Light of different wavelengths incident on the retina through fully open (slitted) Iris
  - Defines *immediate* (20ms) response of retinal cells
- Beamed light of different patterns into the eyes and measured neural responses in striate cortex

# Hubel and Wiesel 1959

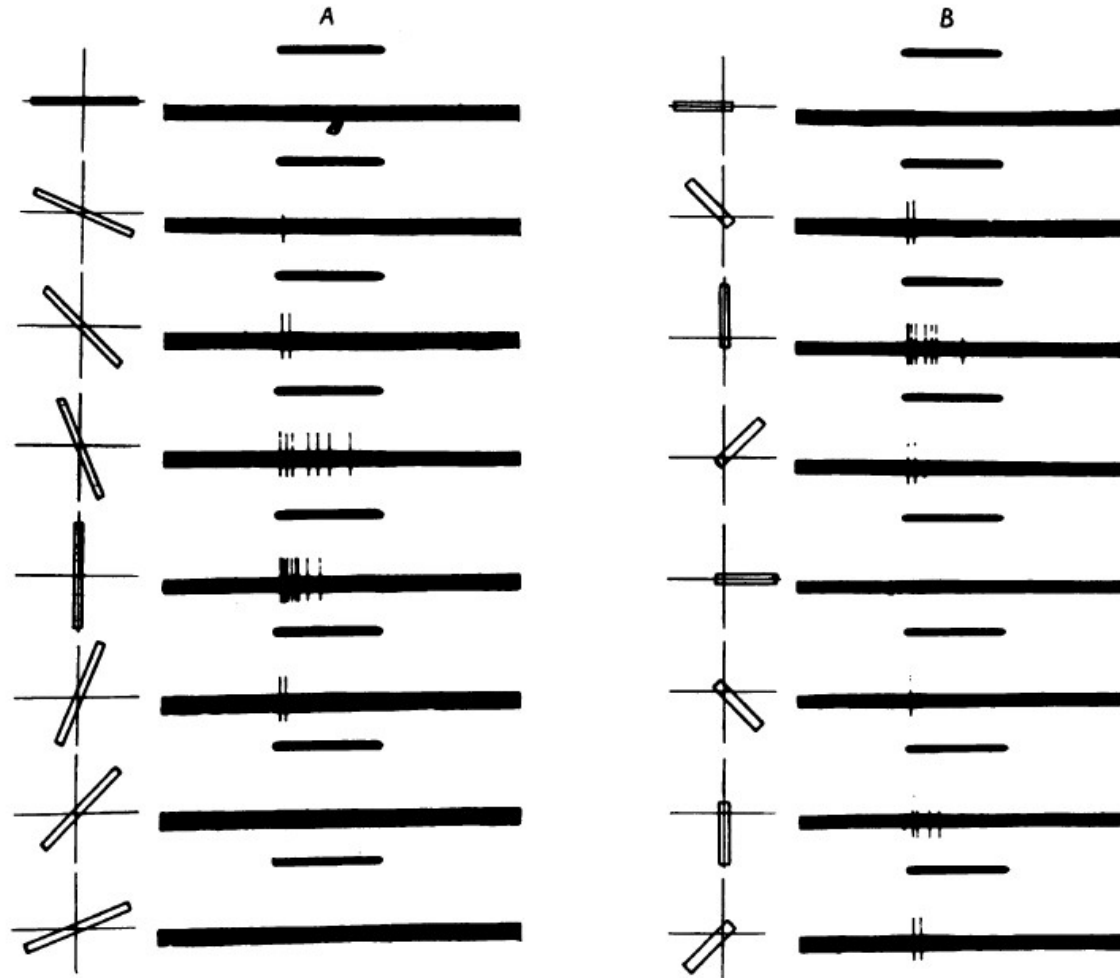


From Hubel and Wiesel



- Restricted retinal areas which on illumination influenced the firing of single cortical units were called **receptive fields**.
  - These fields were usually subdivided into excitatory and inhibitory regions.
- Findings:
  - A light stimulus covering the whole receptive field, or diffuse illumination of the whole retina, was ineffective in driving most units, as excitatory regions cancelled inhibitory regions
    - Light must fall on excitatory regions and NOT fall on inhibitory regions, resulting in clear patterns
  - A spot of light gave greater response for some directions of movement than others.
    - Can be used to determine the receptive field
  - Receptive fields could be oriented in a vertical, horizontal or oblique manner.
    - Based on the arrangement of excitatory and inhibitory regions within receptive fields.

# Hubel and Wiesel 59



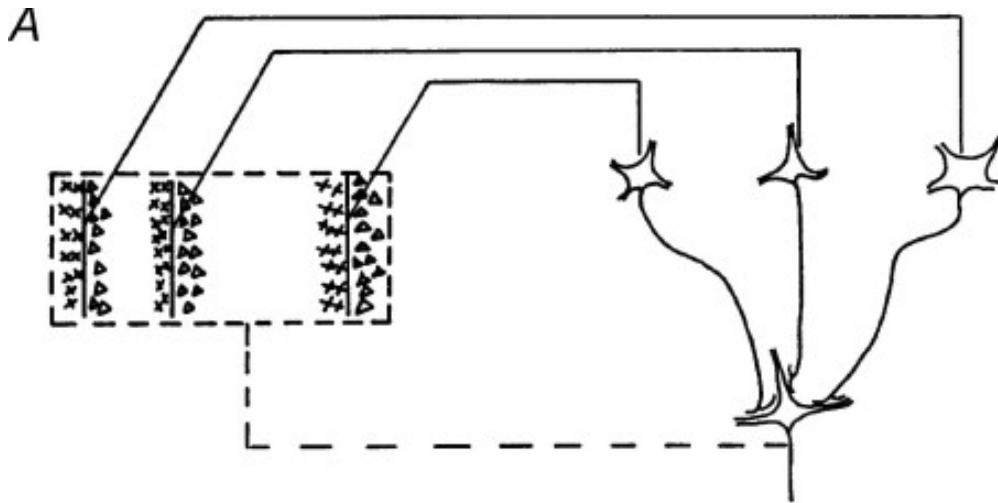
- Response as orientation of input light rotates
  - Note spikes – this neuron is sensitive to vertical bands



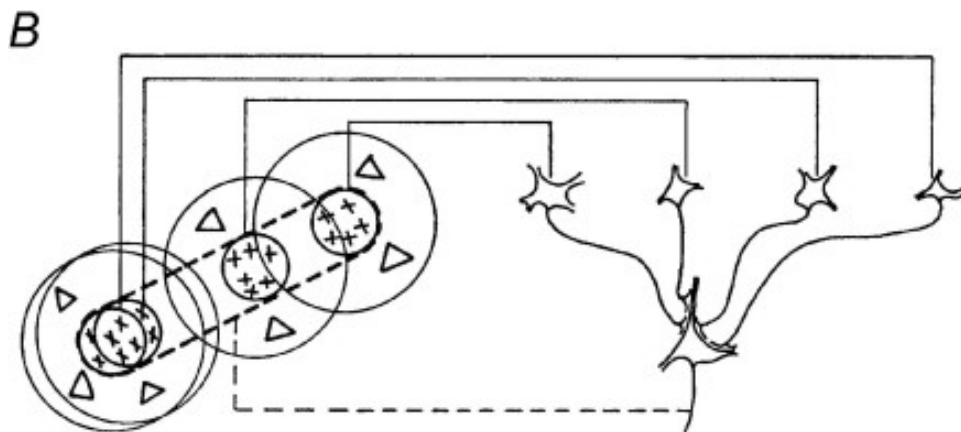
# Hubel and Wiesel

- Oriented slits of light were the most effective stimuli for activating striate cortex neurons
- The orientation selectivity resulted from *the previous level of input* because lower-level neurons responding to a slit also responded to patterns of spots if they were aligned with the same orientation as the slit.
- In a later paper (Hubel & Wiesel, 1962), they showed that within the striate cortex, two levels of processing could be identified
  - Between neurons referred to as *simple* S-cells and *complex* C-cells.
  - Both types responded to oriented slits of light, but complex cells were not “confused” by spots of light while simple cells could be confused

# Hubel and Wiesel model



Composition of complex receptive fields from simple cells. The C-cell responds to the largest output from a bank of S-cells to achieve oriented response that is robust to distortion



Transform from circular retinal receptive fields to elongated fields for simple cells. The simple cells are susceptible to fuzziness and noise

# Hubel and Wiesel

- Complex C-cells build from similarly oriented simple cells
  - They “fine-tune” the response of the simple cell
- Show complex buildup – building *more complex patterns* by composing early neural responses
  - Successive transformation through Simple-Complex combination layers
- Demonstrated more and more complex responses in later papers
  - Later experiments were on waking macaque monkeys
    - Too horrible to recall



# Adding insult to injury..

- “However, this model cannot accommodate the color, spatial frequency and many other features to which neurons are tuned. The exact organization of all these cortical columns within V1 remains a hot topic of current research.”



# Poll 1

**@673**

- According to Hubel and Wiesel which type of cells found patterns in the input and which cells “cleaned” up these patterns?
  - S cells find patterns and C cells clean them up
  - C cells find patterns and S cells clean them up

# Poll 1

According to Hubel and Wiesel which type of cells found patterns in the input and which cells “cleaned” up these patterns?

- **S cells find patterns and C cells clean them up**
- C cells find patterns and S cells clean them up

# Forward to 1980

- Kunihiro Fukushima
- Recognized deficiencies in the Hubel-Wiesel model

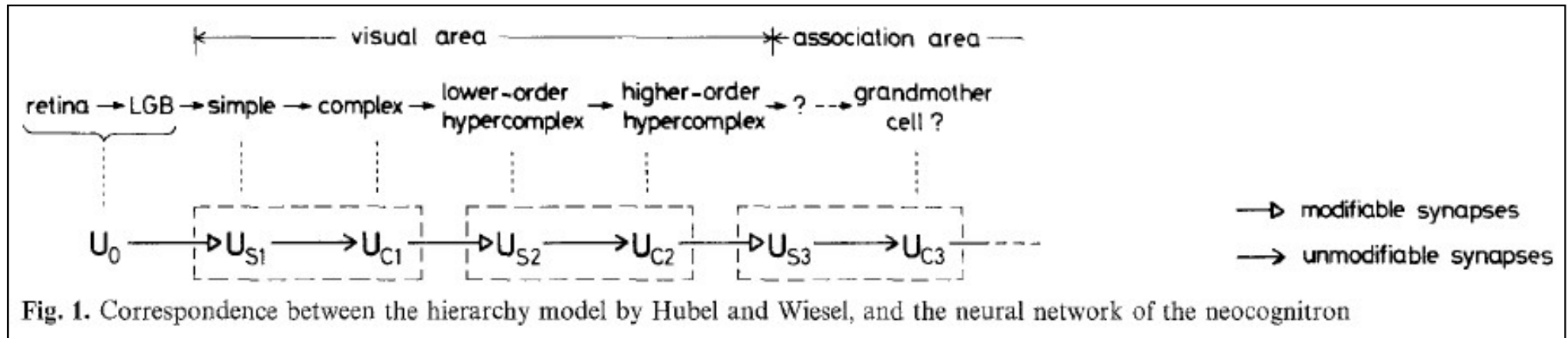


Kunihiro Fukushima

- One of the chief problems: Position invariance of input
  - Your grandmother cell fires even if your grandmother moves to a different location in your field of vision

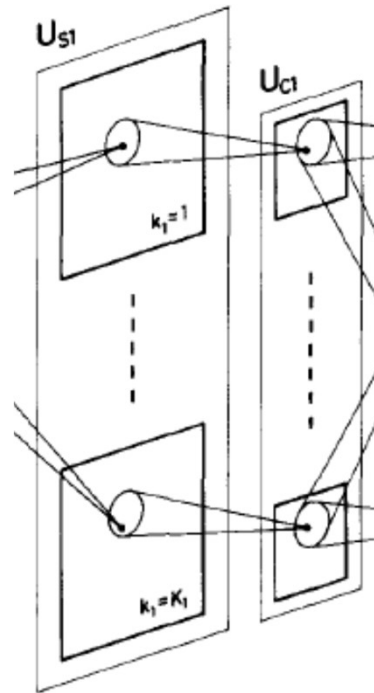
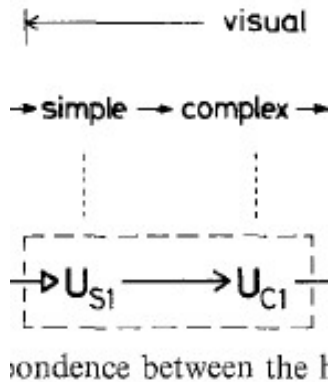
# NeoCognitron

Figures from Fukushima, '80



- Visual system consists of a hierarchy of modules, each comprising a layer of “S-cells” followed by a layer of “C-cells”
  - $U_{Sl}$  is the  $l^{\text{th}}$  layer of S cells,  $U_{Cl}$  is the  $l^{\text{th}}$  layer of C cells
- S-cells **respond** to the signal in the previous layer
- C-cells **confirm** the S-cells’ response
- Only S-cells are “plastic” (i.e. learnable), C-cells are fixed in their response

# NeoCognitron



Each cell in a plane "looks" at a slightly shifted region of the **input** than the adjacent cells in the plane.

... "through" the previous layer planes

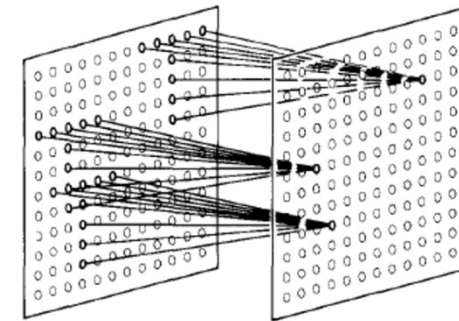
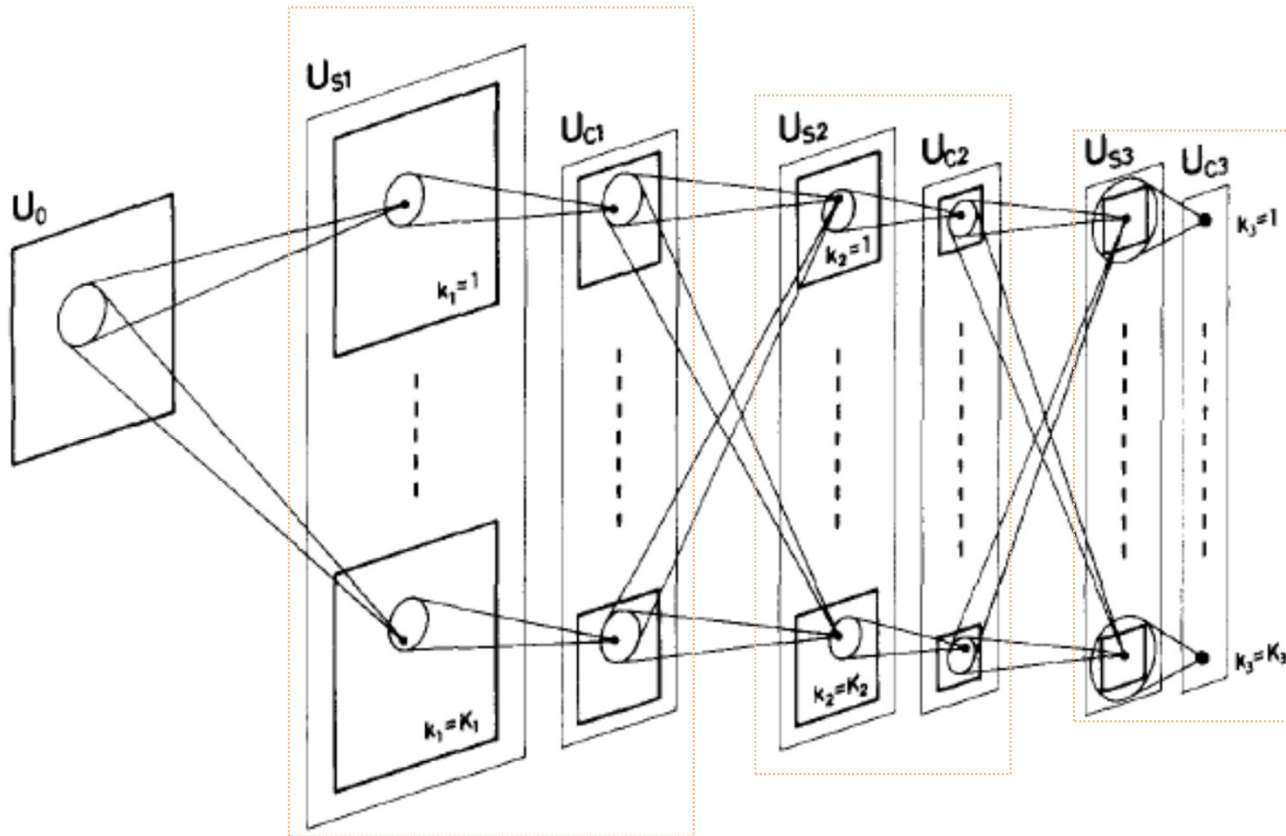


Fig. 3. Illustration showing the input interconnections to the cells within a single cell-plane

- Each simple-complex module includes a layer of S-cells and a layer of C-cells
- S-cells are organized in rectangular groups called S-planes.
  - All the cells within an S-plane have identical learned responses
- C-cells too are organized into rectangular groups called C-planes
  - One C-plane per S-plane
  - All C-cells have identical fixed response
- In Fukushima's original work, each C and S cell "looks" at an elliptical region in the previous plane



# NeoCognitron



- The complete network
- $U_0$  is the retina
- In each subsequent module, the planes of the S layers detect plane-specific patterns in the previous layer (C layer or retina)
- The planes of the C layers “refine” the response of the corresponding planes of the S layers

# Neocognitron

- S cells: RELU like activation

$$u_{SI}(k_t, \mathbf{n}) = r_t \cdot \varphi \left[ \frac{1 + \sum_{k_{l-1}=1}^{K_{l-1}} \sum_{\mathbf{v} \in S_l} a_l(k_{l-1}, \mathbf{v}, k_t) \cdot u_{CI-1}(k_{l-1}, \mathbf{n} + \mathbf{v})}{1 + \frac{2r_t}{1+r_t} \cdot b_l(k_t) \cdot v_{CI-1}(\mathbf{n})} - 1 \right]$$

–  $\varphi$  is a RELU

- C cells: Also RELU like, but with an inhibitory bias
  - Fires if weighted combination of S cells fires strongly enough

$$u_{CI}(k_t, \mathbf{n}) = \psi \left[ \frac{1 + \sum_{\mathbf{v} \in D_l} d_l(\mathbf{v}) \cdot u_{SI}(k_t, \mathbf{n} + \mathbf{v})}{1 + v_{SI}(\mathbf{n})} - 1 \right]$$

$$\psi[x] = \varphi[x/(\alpha + x)]$$

# Neocognitron

- S cells: RELU like activation

$$u_{SI}(k_l, \mathbf{n}) = r_l \cdot \varphi \left[ \frac{1 + \sum_{k_{l-1}=1}^{K_{l-1}} \sum_{\mathbf{v} \in S_l} a_l(k_{l-1}, \mathbf{v}, k_l) \cdot u_{CI-1}(k_{l-1}, \mathbf{n} + \mathbf{v})}{1 + \frac{2r_l}{1+r_l} \cdot b_l(k_l) \cdot v_{CI-1}(\mathbf{n})} - 1 \right]$$

–  $\varphi$  is a RELU

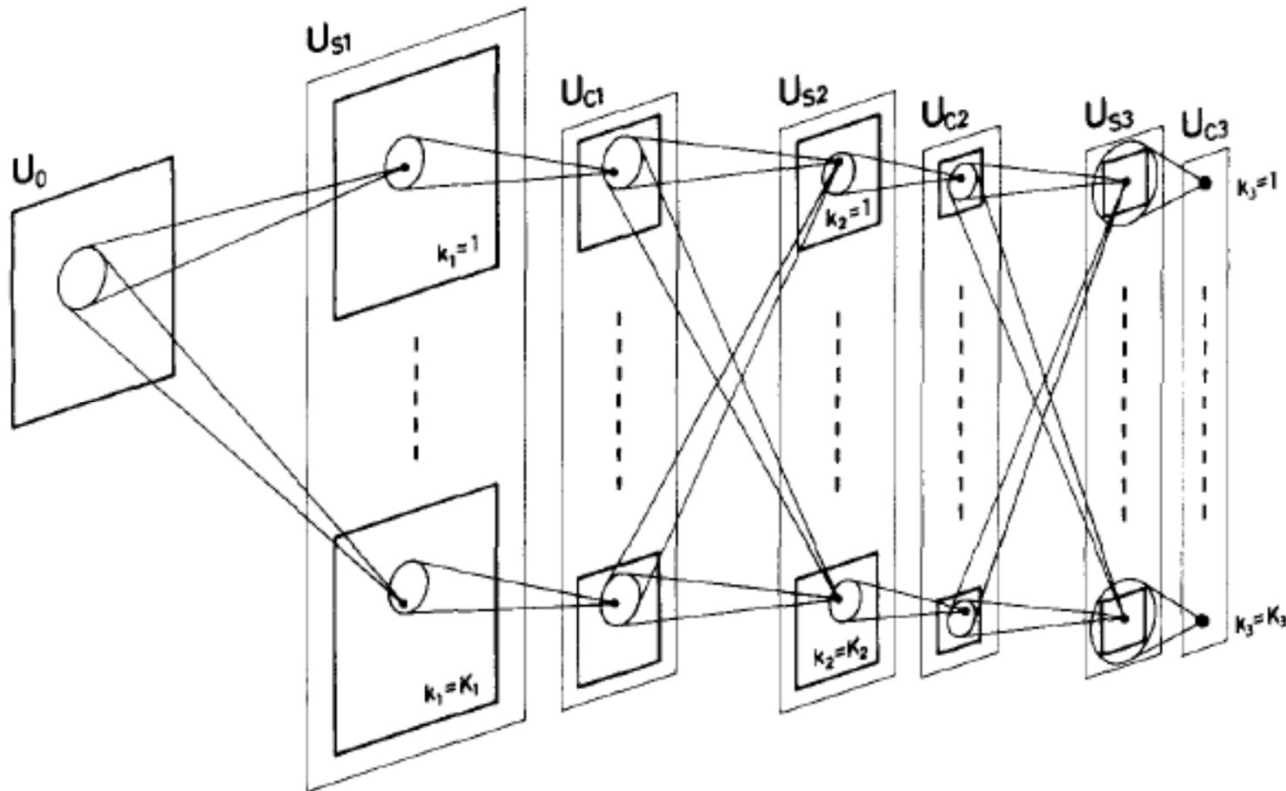
- C cells: Also RELU like, but with an inhibitory bias
  - Fires if weighted combination of S cells fires strongly enough

$$u_{CI}(k_l, \mathbf{n}) = \psi \left[ \frac{1 + \sum_{\mathbf{v} \in D_l} d_l(\mathbf{v}) \cdot u_{SI}(k_l, \mathbf{n} + \mathbf{v})}{1 + v_{SI}(\mathbf{n})} - 1 \right]$$

$$\psi[x] = \varphi[x/(\alpha + x)]$$

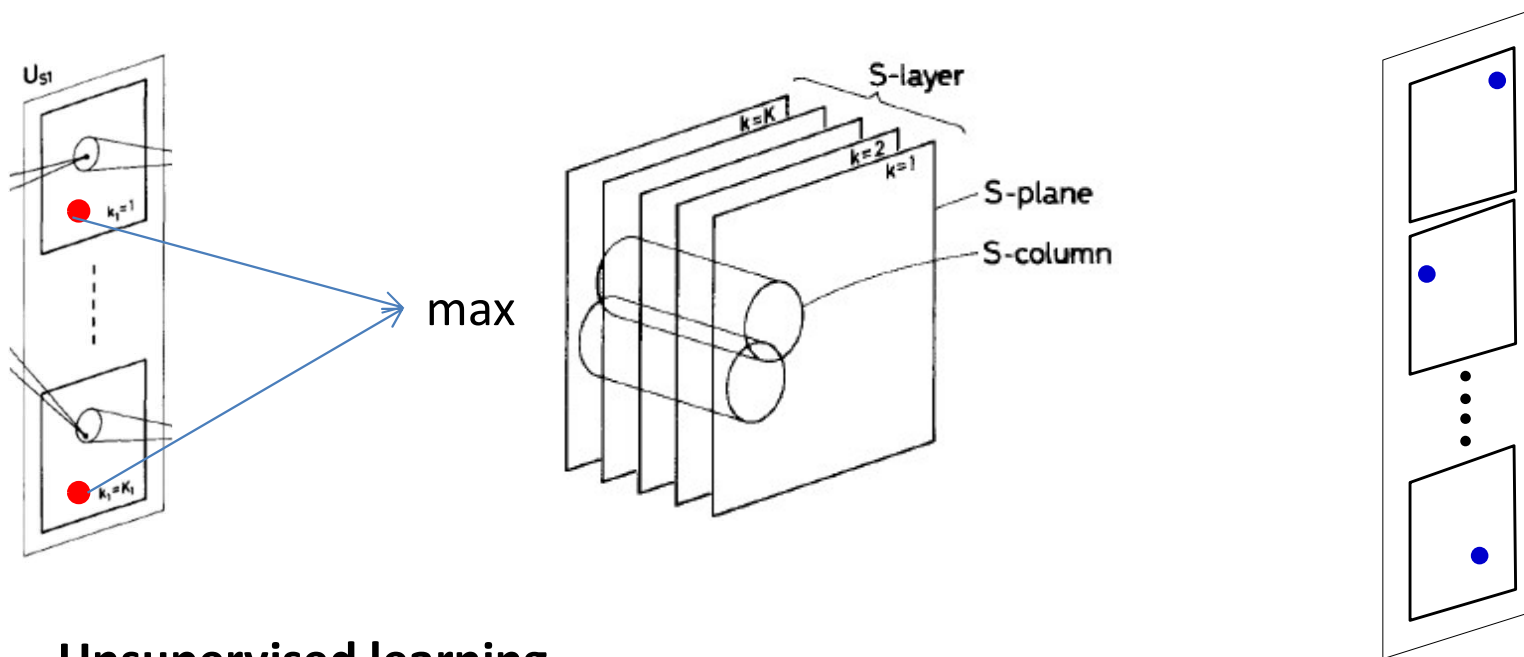
Could simply replace these strange functions with a RELU and a max

# NeoCognitron



- The deeper the layer, the larger the receptive field of each neuron
  - Cell planes get smaller with layer number
  - Number of planes increases
    - i.e the number of complex pattern detectors increases with layer

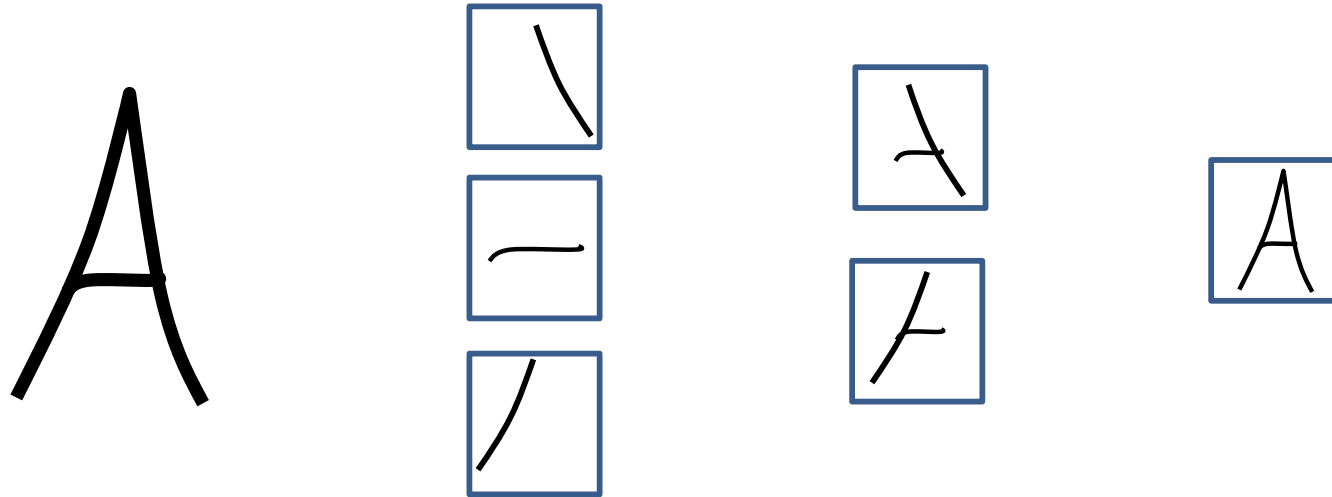
# Learning in the neocognitron



- **Unsupervised learning**
- Randomly initialize S cells, perform Hebbian learning updates in response to input
  - update = product of input and output :  $\Delta w_{ij} = x_i y_j$
- Within any layer, at any position, only the maximum S from all the layers is selected for update
  - Also viewed as max-valued cell from each *S column*
    - Ensures only one of the planes picks up any feature
    - If multiple max selections are on the same plane, only the largest is chosen
  - But across all positions, multiple planes will be selected
- Updates are distributed across all cells within the plane

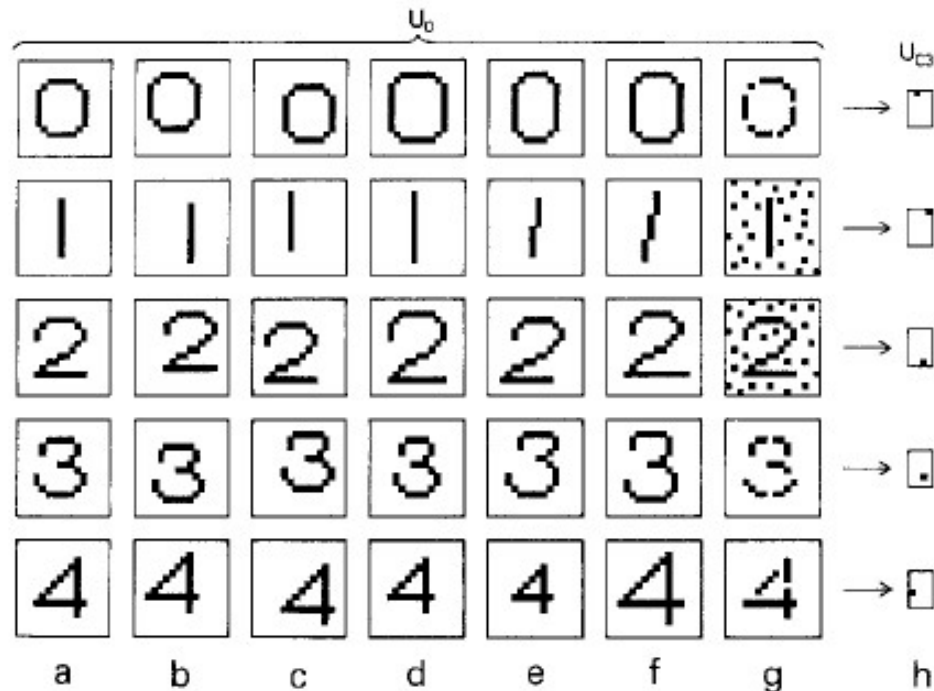


# Learning in the neocognitron



- Ensures different planes learn different features
  - E.g. Given many examples of the character “A” the different cell planes in the S-C layers may learn the patterns shown
    - Given other characters, other planes will learn their components
  - Going up the layers goes from local to global receptor fields
- Winner-take-all strategy makes it robust to distortion
- Unsupervised: Effectively clustering

# Neocognitron – finale



- Fukushima showed it successfully learns to cluster semantic visual concepts
  - E.g. number or characters, even in noise

# Poll 2

**@674, @675**

Fukushima's model is an unsupervised CNN, true or false

- True
- False

Supervision can be added to Fukushima's model, true or false

- True
- False

# Poll 2

Fukushima's model is an unsupervised CNN, true or false

- **True**
- False

Supervision can be added to Fukushima's model, true or false

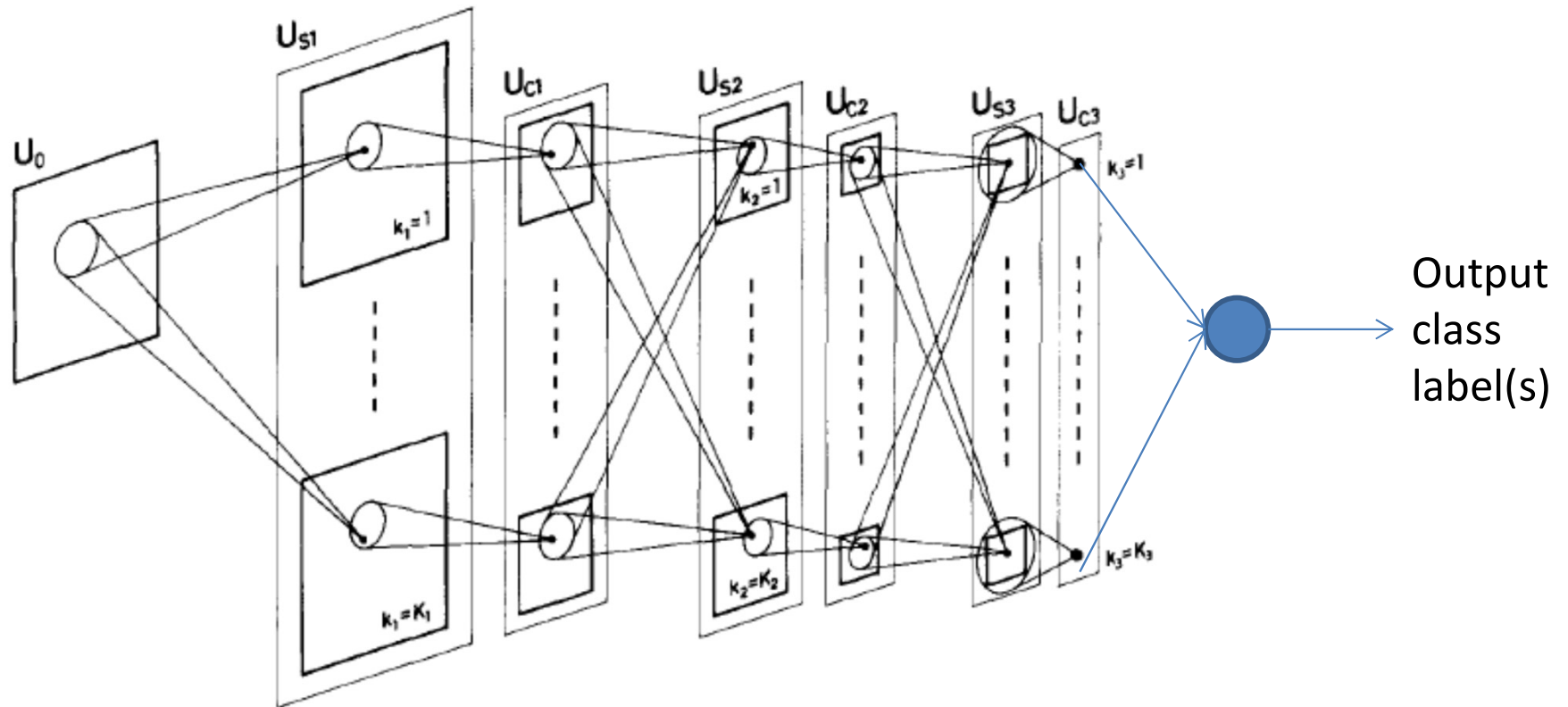
- **True**
- False

# Adding Supervision

- The neocognitron is fully unsupervised
  - Semantic labels are automatically learned
- Can we add external supervision?
- Various proposals:
  - Temporal correlation: Homma, Atlas, Marks, '88
  - TDNN: Lang, Waibel et. al., 1989, '90
- Convolutional neural networks: LeCun

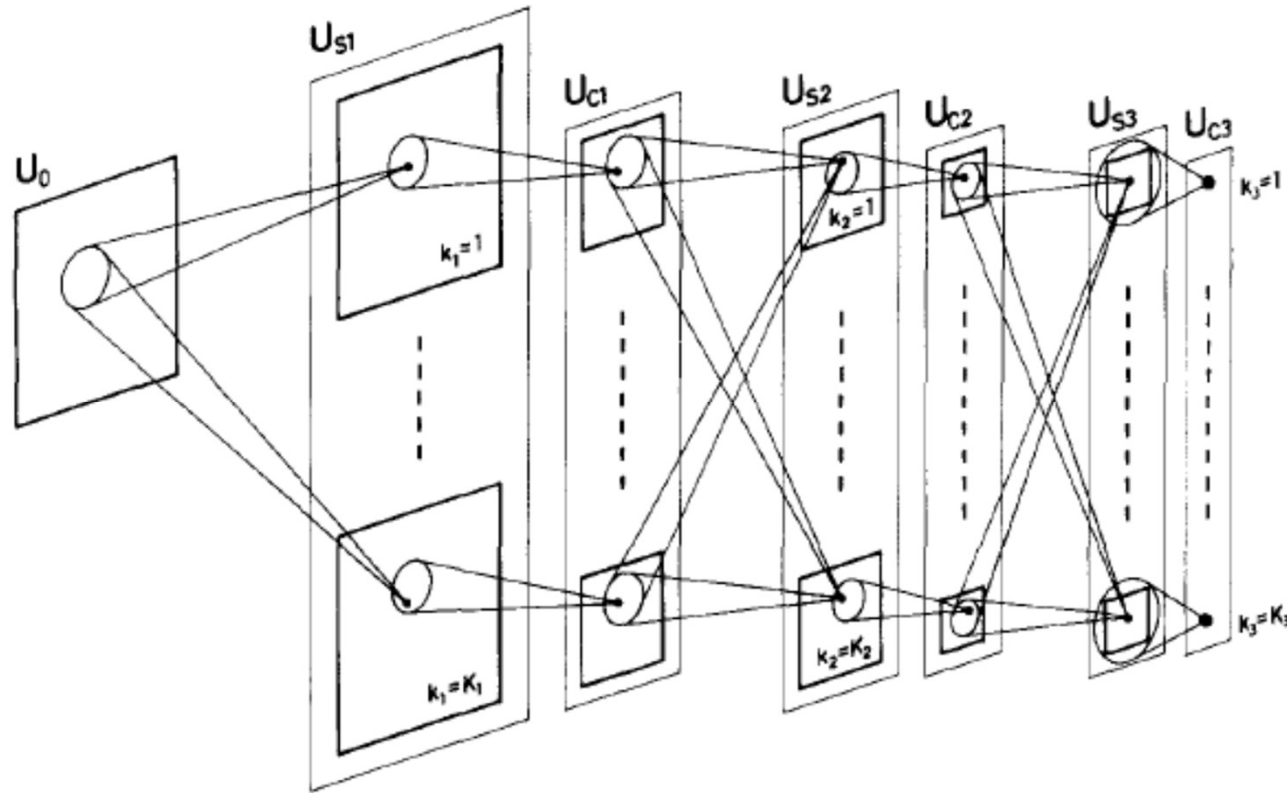


# Supervising the neocognitron



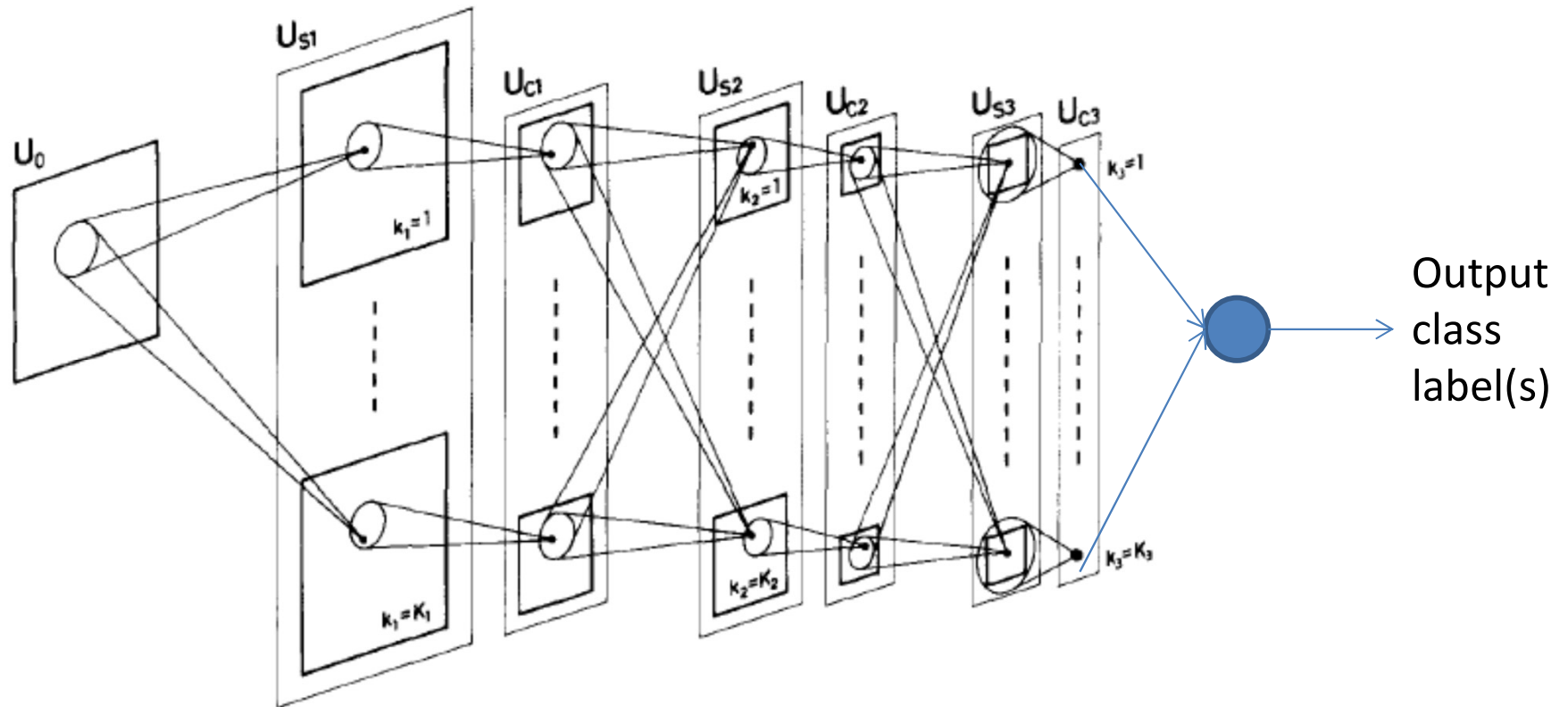
- Add an extra decision layer after the final C layer
  - Produces a class-label output
- We now have a fully feed forward MLP with shared parameters
  - All the S-cells within an S-plane have the same weights
- Simple backpropagation can now train the S-cell weights in every plane of every layer
  - C-cells are not updated

# Scanning vs. multiple filters



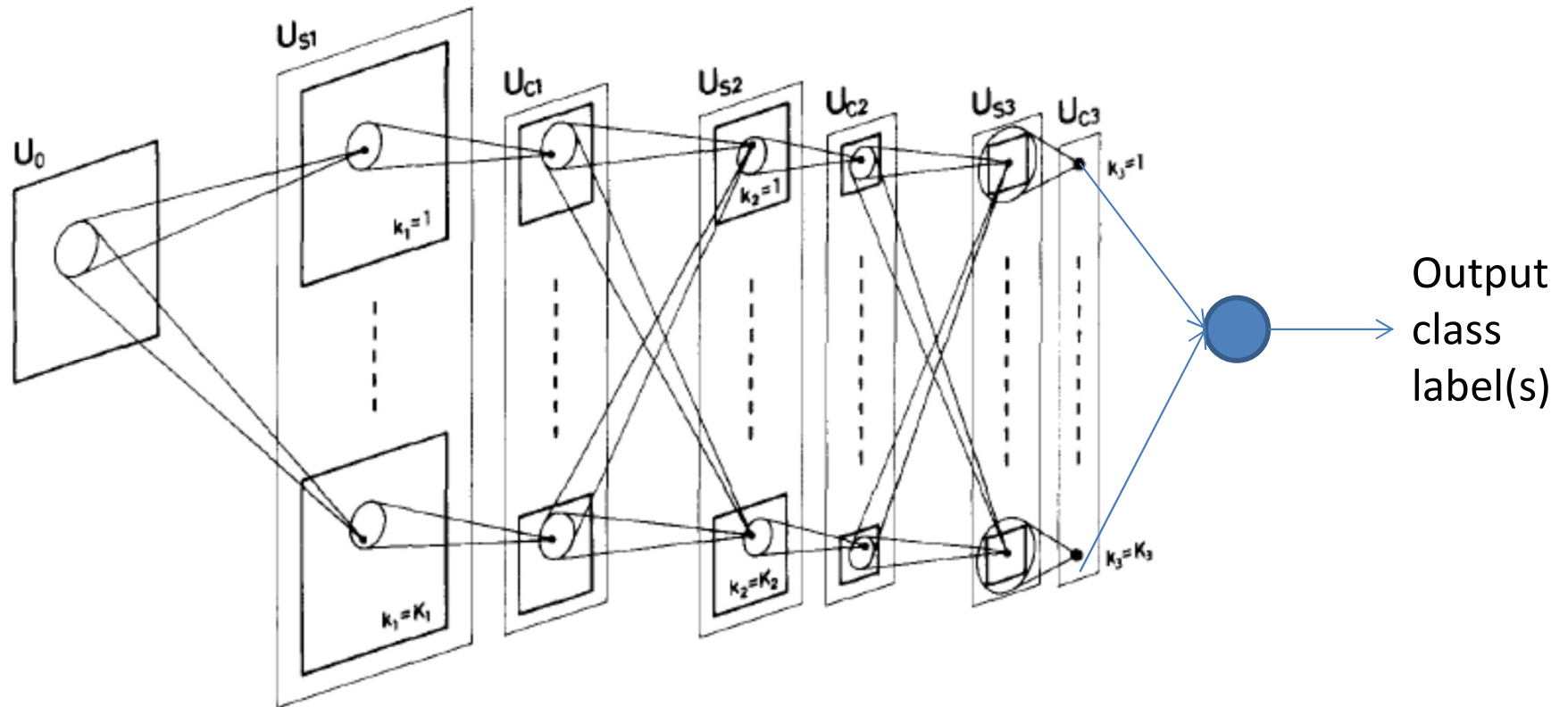
- **Note:** The original Neocognitron actually uses many identical copies of a neuron in each S and C plane
  - Mathematically identical to “scanning” with a single copy

# Supervising the neocognitron



- The Math
  - Assuming *square* receptive fields, rather than elliptical ones
  - Receptive field of S cells in  $l$ th layer is  $K_l \times K_l$
  - Receptive field of C cells in  $l$ th layer is  $L_l \times L_l$ 
    - **C cells “stride” by more than one pixel, resulting in a shrinking, or “downsampling” of the maps**

# Supervising the neocognitron

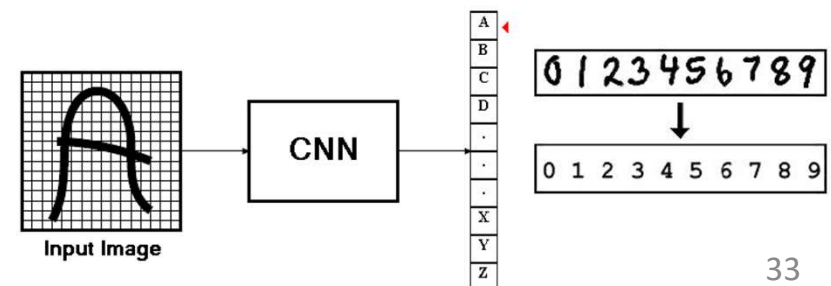
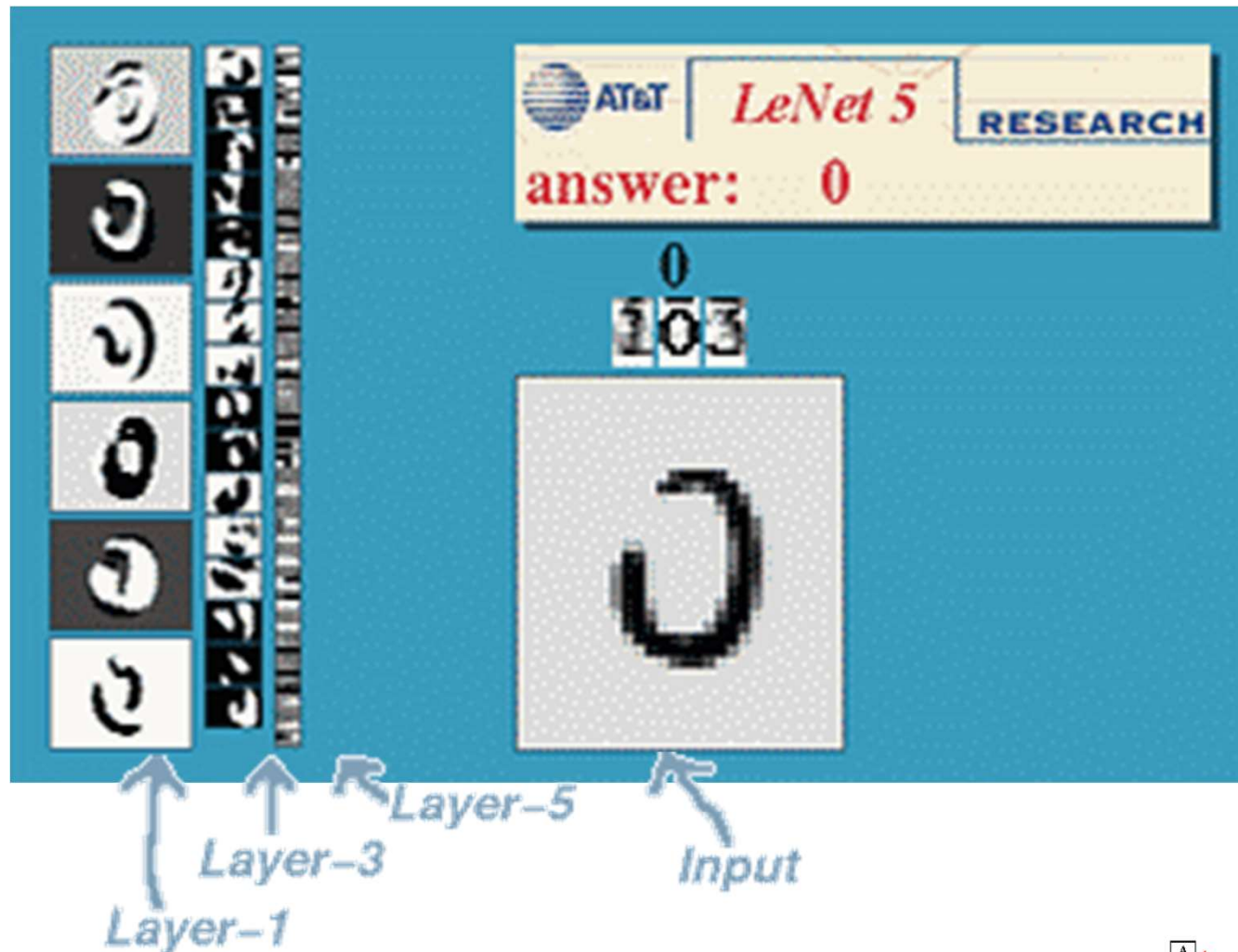


$$U_{S,l,n}(i,j) = \sigma \left( \sum_p \sum_{k=1}^{K_l} \sum_{l=1}^{K_l} w_{S,l,n}(p,k,l) U_{C,l-1,p}(i+l-1, j+k-1) \right)$$

$$U_{C,l,n}(i,j) = \max_{k \in (i, i+L_l), j \in (l, l+L_l)} (U_{S,l,n}(i,j))$$

- This is, identical to “scanning” (convolving) with a single neuron/filter (what LeNet actually did)

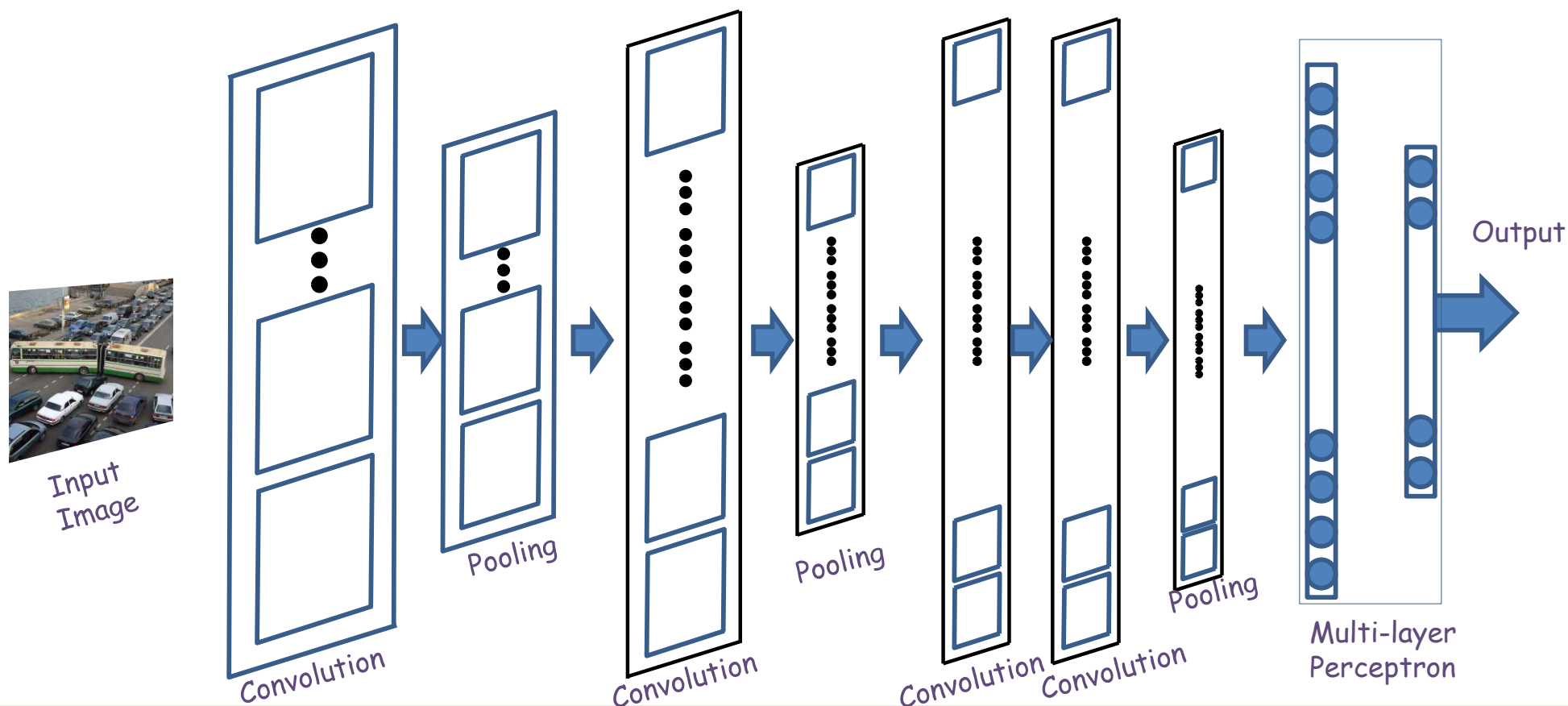
# Convolutional Neural Networks



# Story so far

- The mammalian visual cortex contains of S cells, which capture oriented visual patterns and C cells which perform a “majority” vote over groups of S cells for robustness to noise and positional jitter
- The neocognitron emulates this behavior with planar banks of S and C cells with identical response, to enable shift invariance
  - Only S cells are learned
  - C cells perform the equivalent of a max over groups of S cells for robustness
  - Unsupervised learning results in learning useful patterns
- LeCun’s LeNet added external supervision to the neocognitron
  - S planes of cells with identical response are modelled by a scan (convolution) over image planes by a single neuron
  - C planes are emulated by cells that perform a max over groups of S cells
    - Reducing the size of the S planes
  - Giving us a “Convolutional Neural Network”

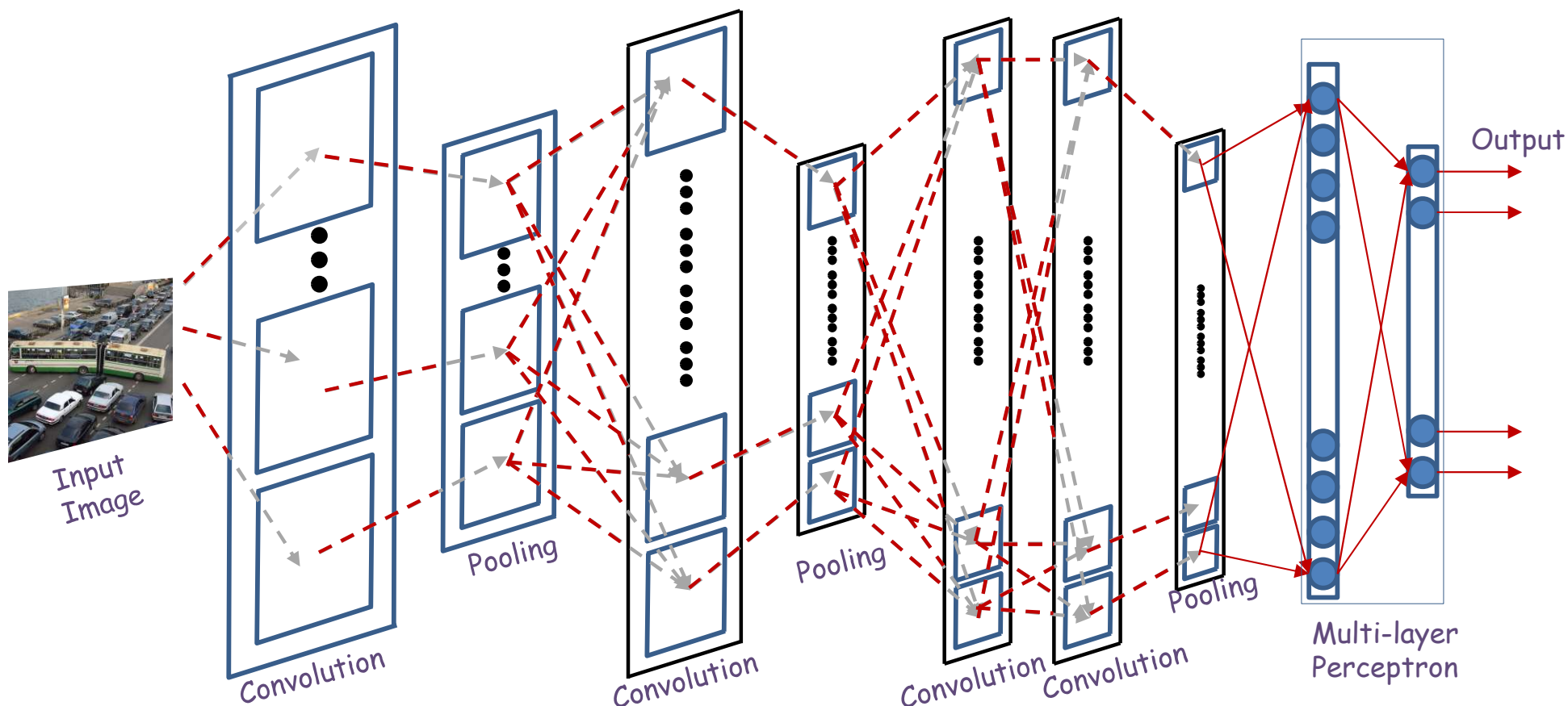
# The general architecture of a convolutional neural network



- A convolutional neural network comprises “convolutional” and “pooling” layers
  - Convolutional layers comprise neurons that scan their input for patterns
    - Correspond to S planes
  - Pooling layers perform max operations on groups of outputs from the convolutional layers
    - Correspond to C planes
  - The two may occur in any sequence, but typically they alternate
- Followed by an MLP with one or more layers



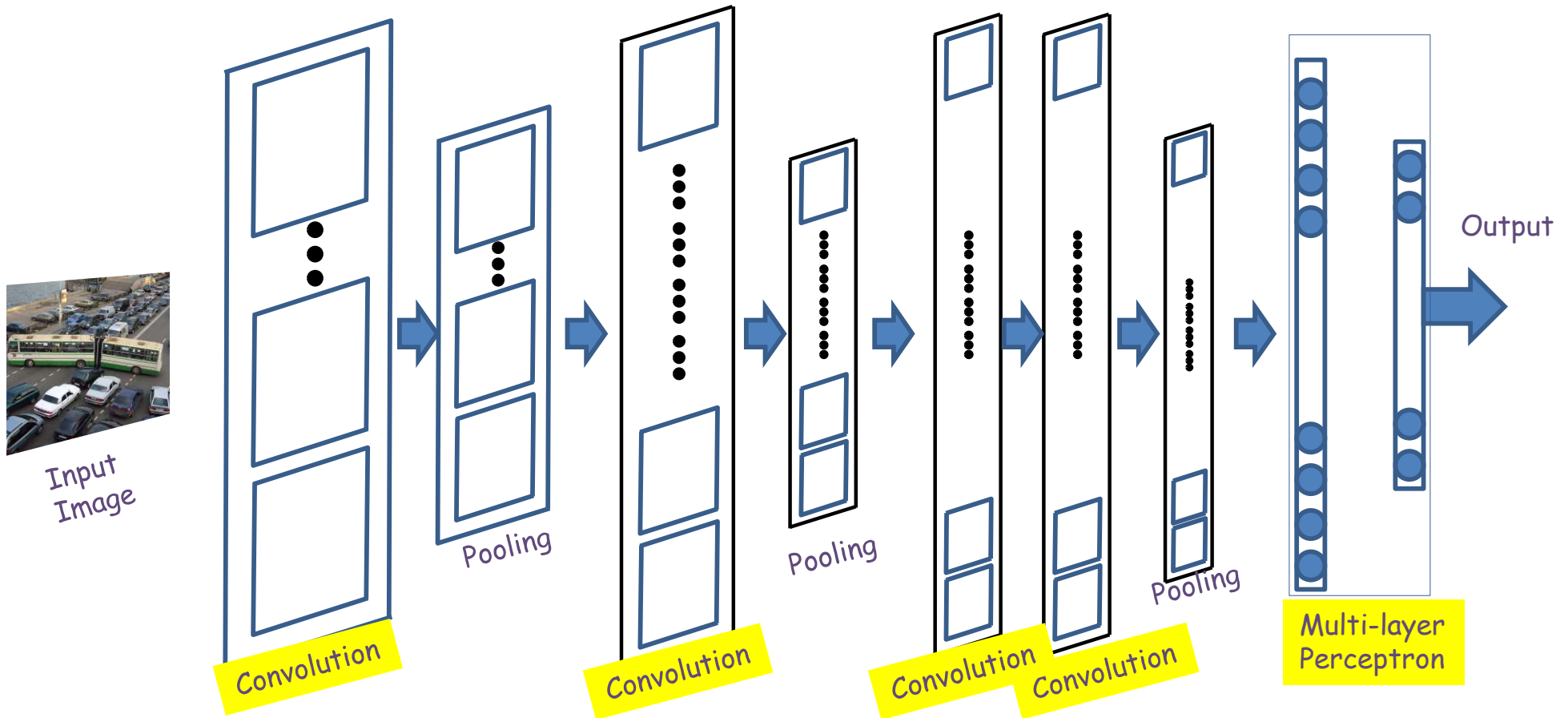
# The general architecture of a convolutional neural network



- A convolutional neural network comprises of “pooling” and “downsampling” layers
  - The two may occur in any sequence, but typically they alternate
- Followed by an MLP with one or more layers

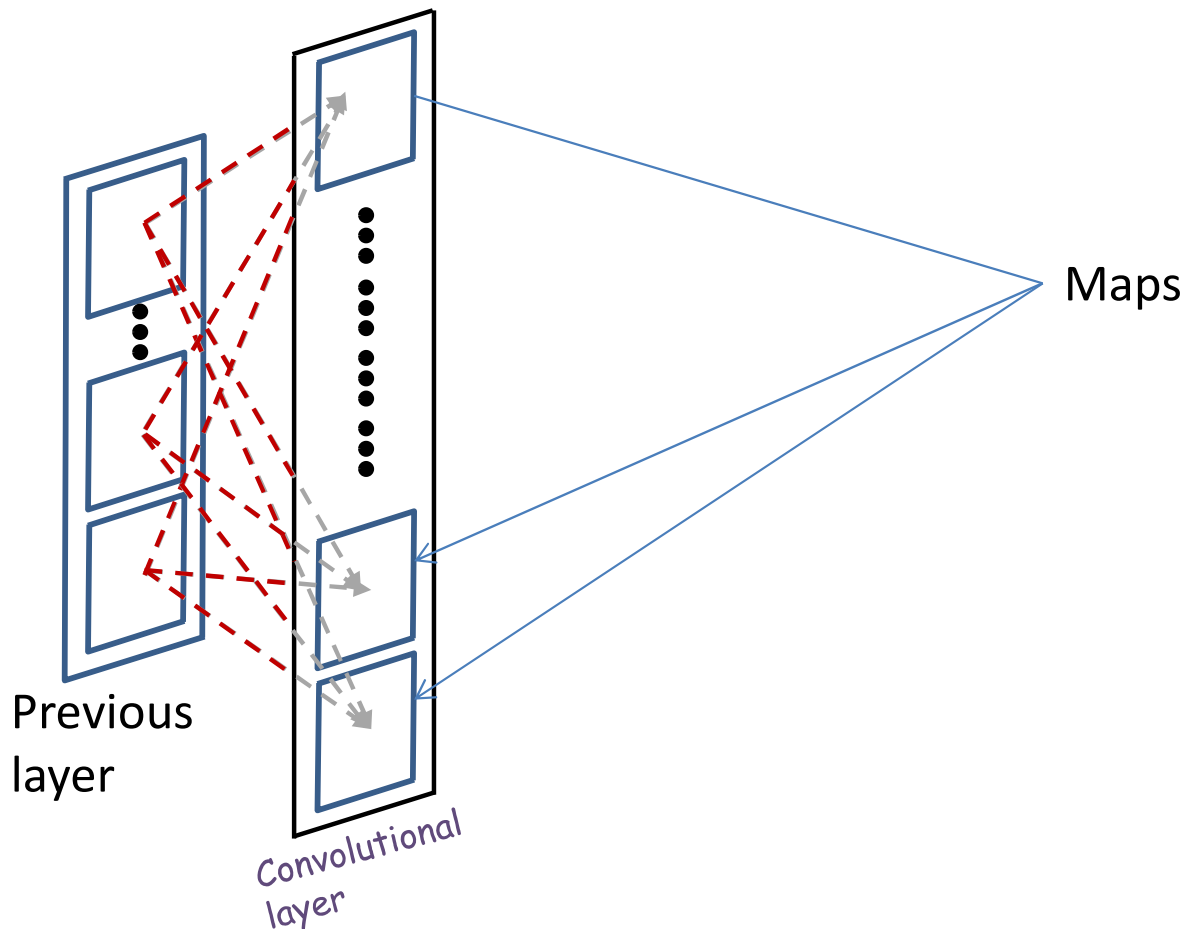


# The general architecture of a convolutional neural network



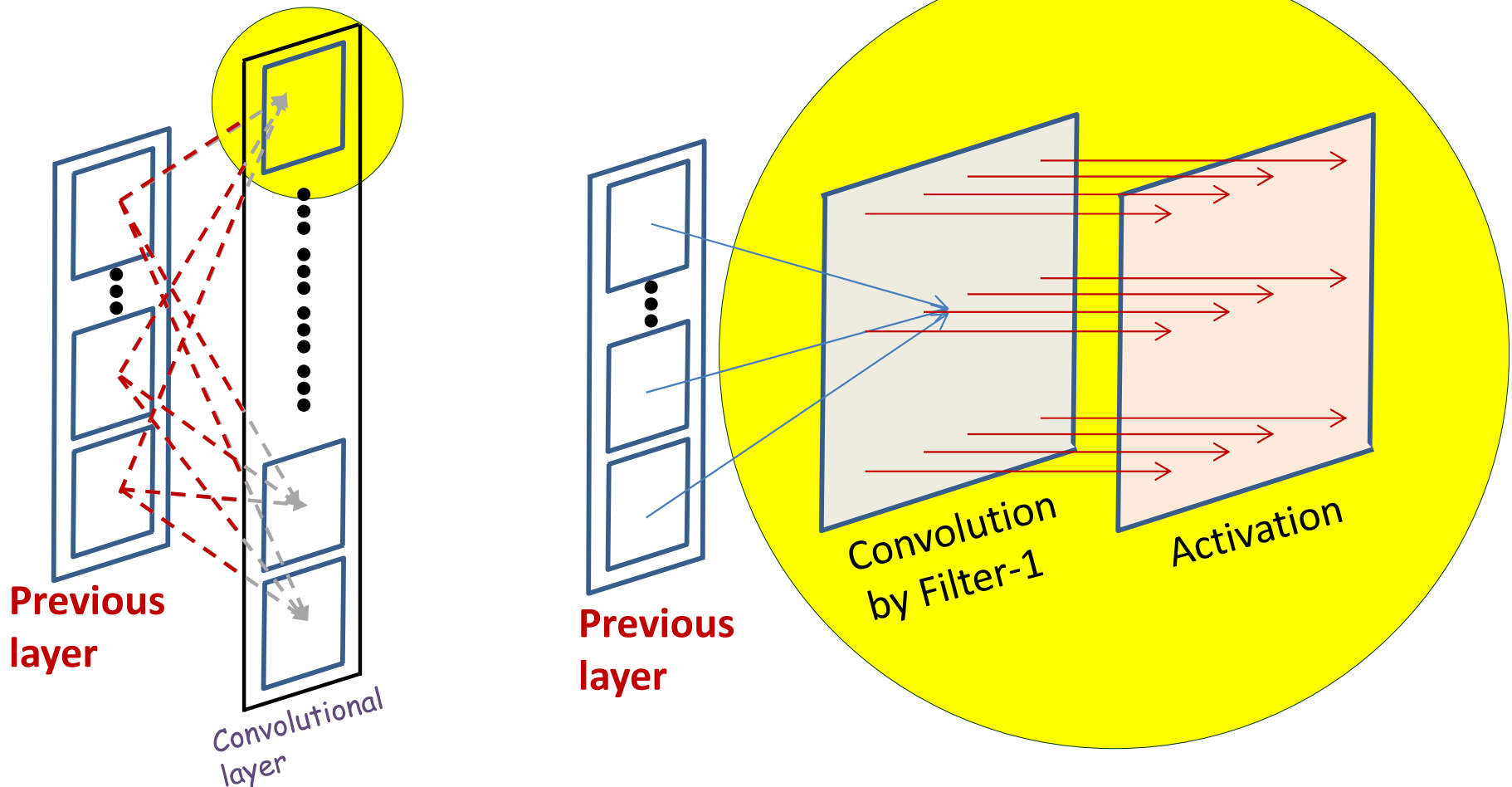
- **Convolutional layers and the MLP are *learnable***
  - Their parameters must be learned from training data for the target classification task
- Pooling layers are fixed and generally not learnable

# A convolutional layer



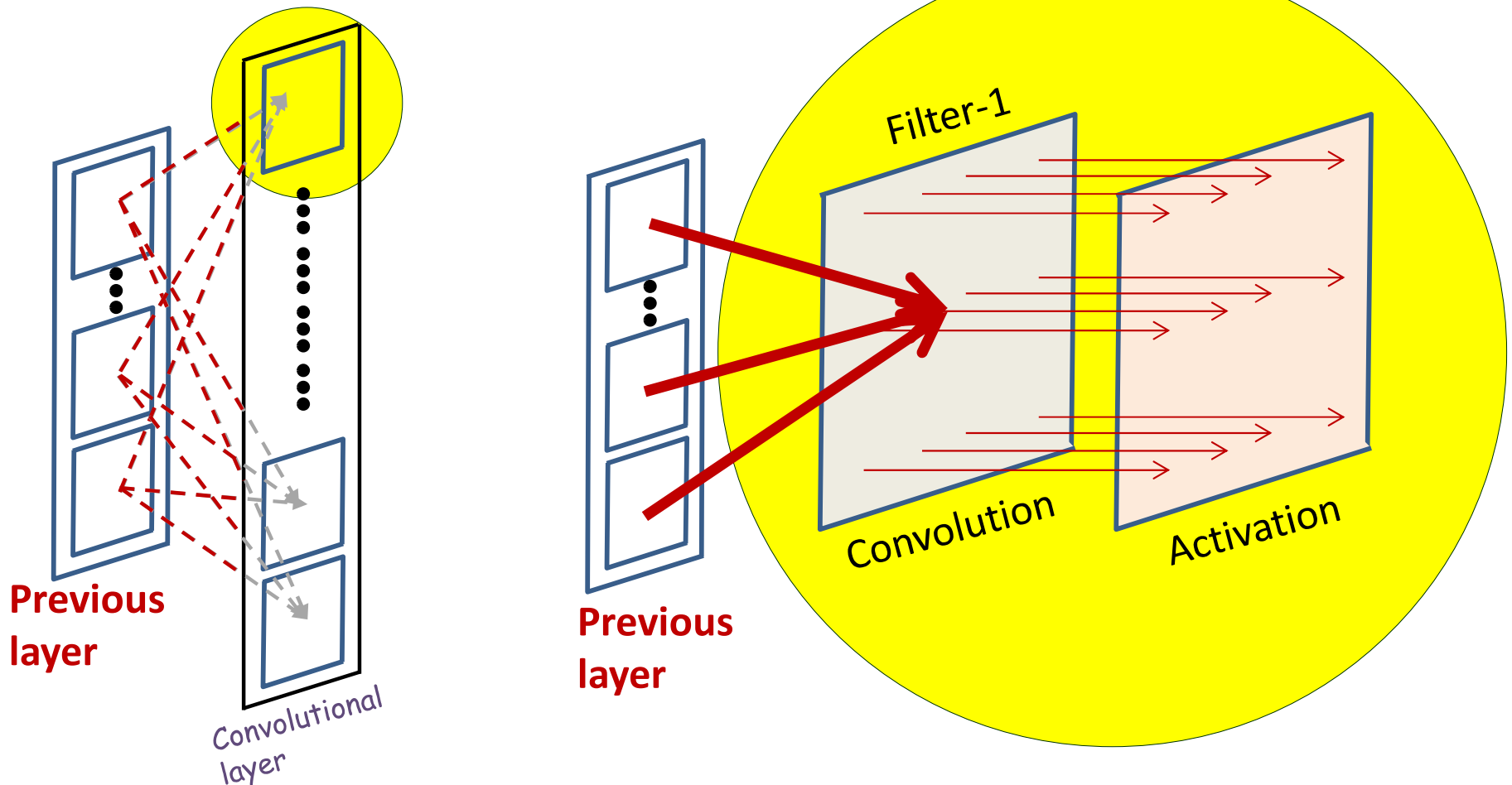
- A convolutional layer comprises of a series of “maps”
  - Corresponding the “S-planes” in the Neocognitron
  - Various called feature maps or activation maps

# A convolutional layer



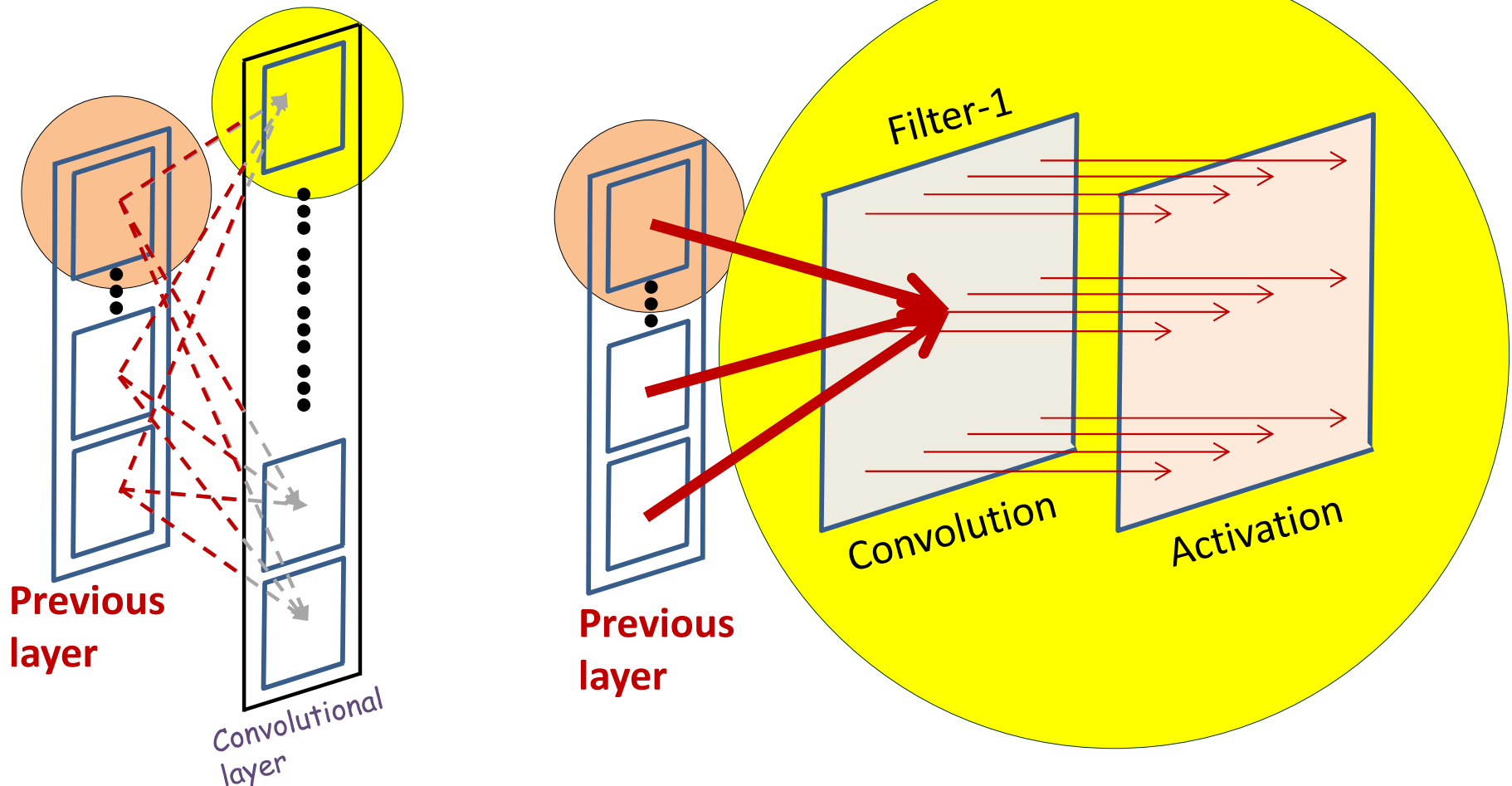
- Each activation map has two components
  - An *affine* map, obtained by *convolution* over maps in the previous layer
    - Each affine map has, associated with it, a **learnable filter**
  - An *activation* that operates on the output of the convolution

# A convolutional layer: affine map



- All the maps in the previous layer contribute to each convolution

# A convolutional layer: affine map



- All the maps in the previous layer contribute to each convolution
  - Consider the contribution of a *single* map

# What is a convolution

Example 5x5 image with binary pixels

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

Example 3x3 filter

1	0	1
0	1	0
1	0	1

bias

0
---

- Scanning an image with a “filter”
  - Note: a filter is really just a perceptron, with weights and a bias

# What is a convolution

**0**  
**bias**

1	0	1
0	1	0
1	0	1

**Filter**

1 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	0	0
0 <sub>x0</sub>	1 <sub>x1</sub>	1 <sub>x0</sub>	1	0
0 <sub>x1</sub>	0 <sub>x0</sub>	1 <sub>x1</sub>	1	1
0	0	1	1	0
0	1	1	0	0

**Input Map**

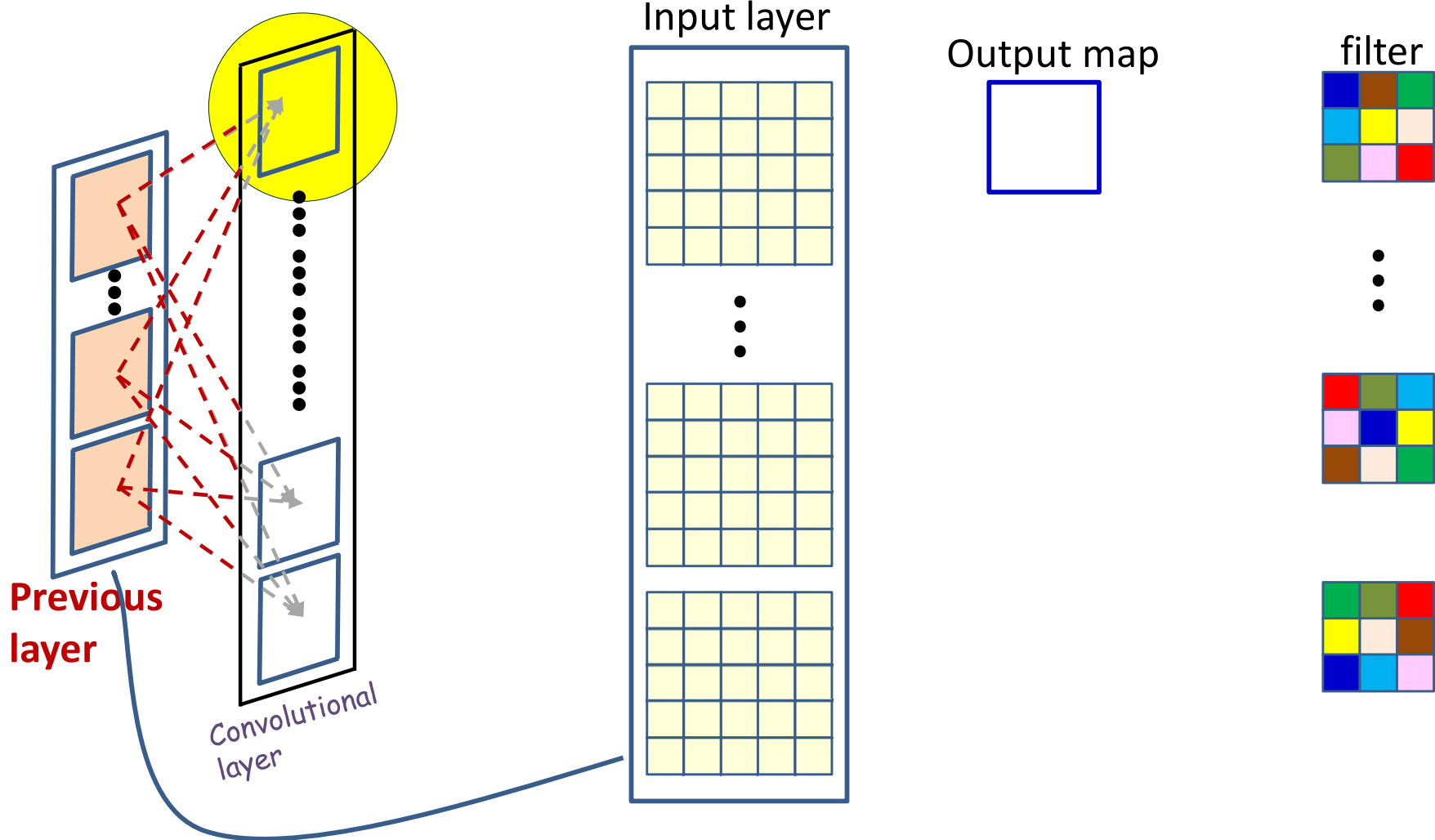
4		

**Convolved  
Feature**

- Scanning an image with a “filter”
  - At each location, the “filter and the underlying map values are multiplied component wise, and the products are added along with the bias

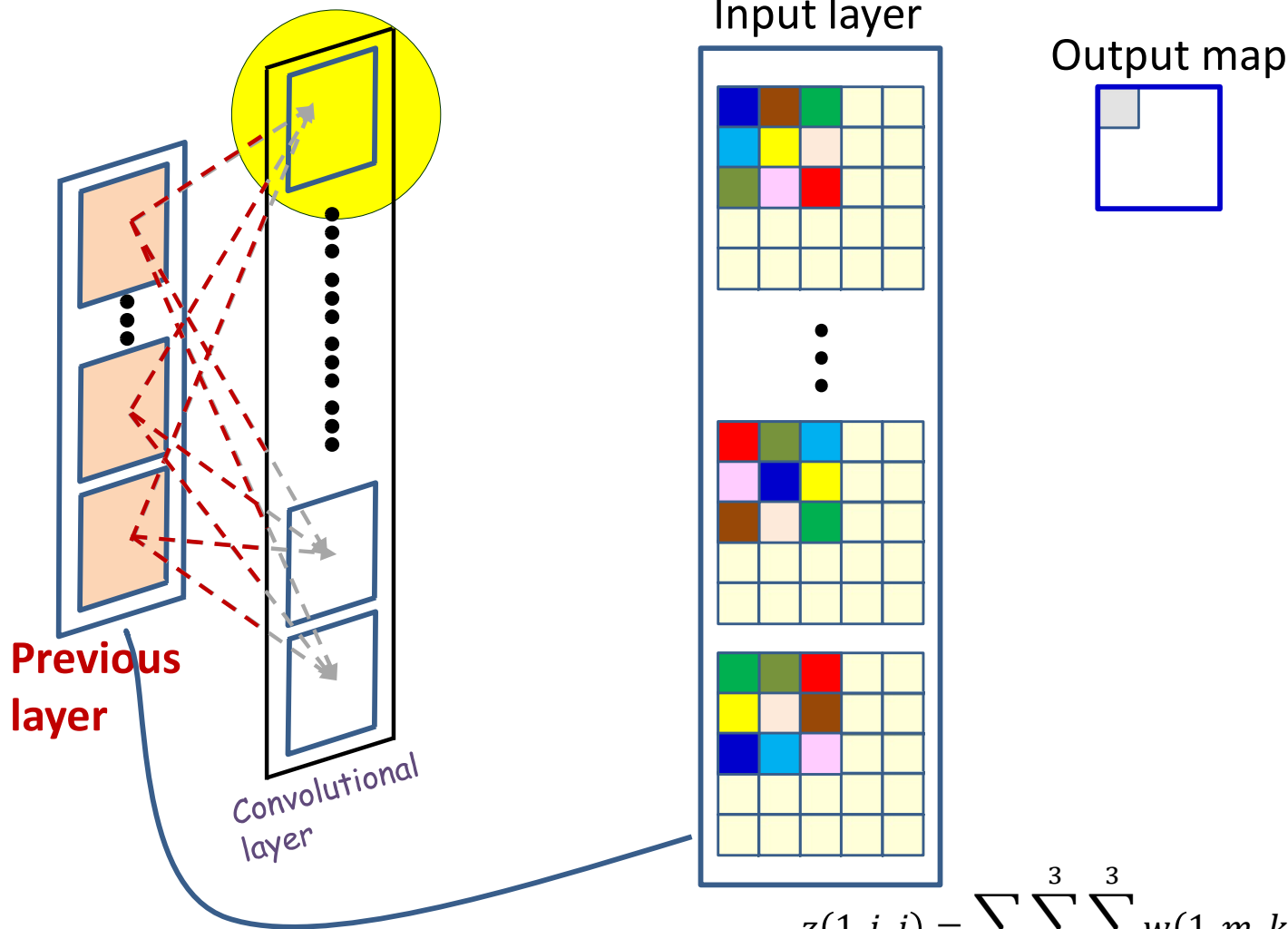


# What really happens



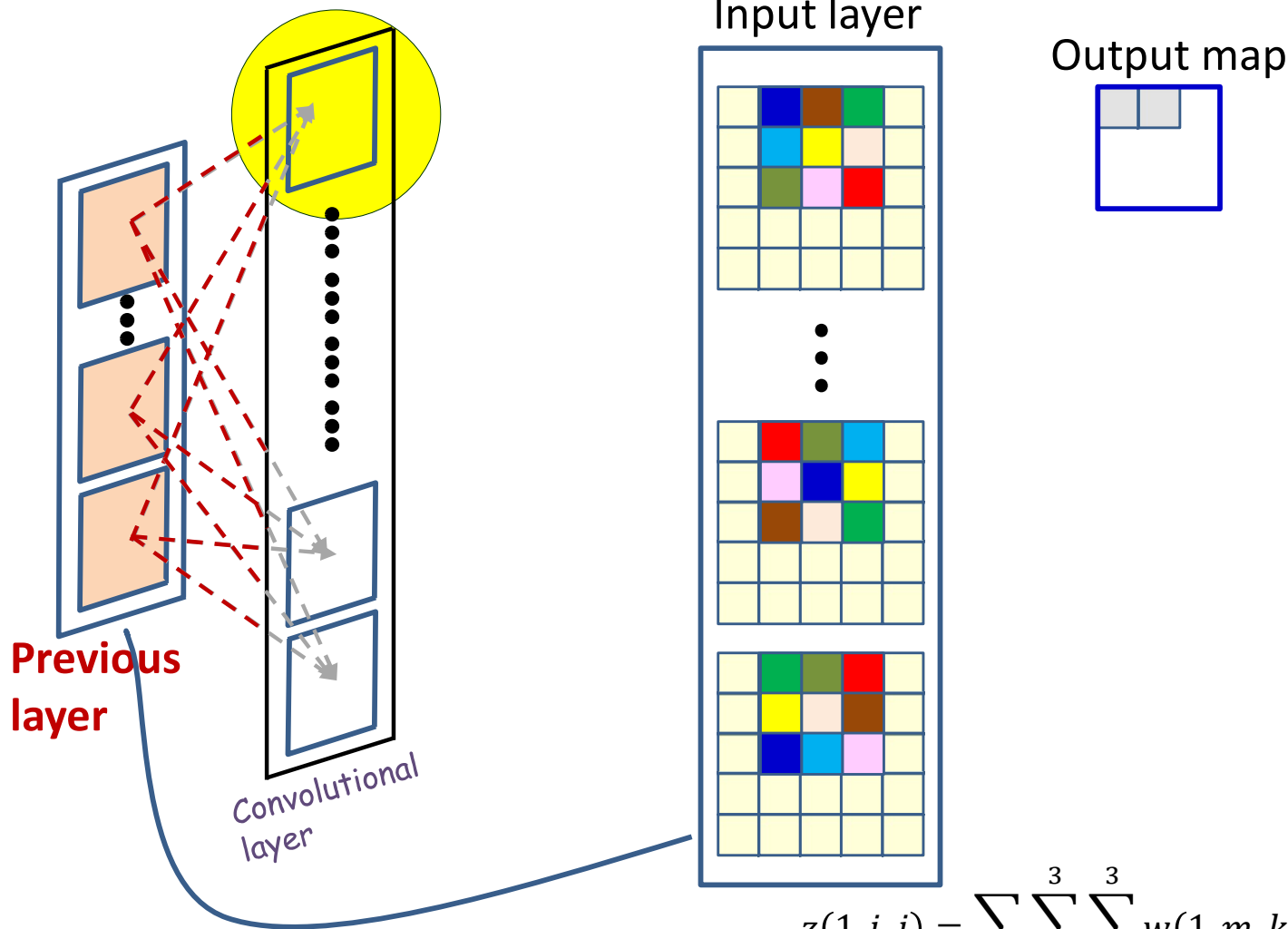
- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as *size of the filter* x *no. of maps in previous layer*

# What really happens



- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as *size of the filter* x *no. of maps in previous layer*

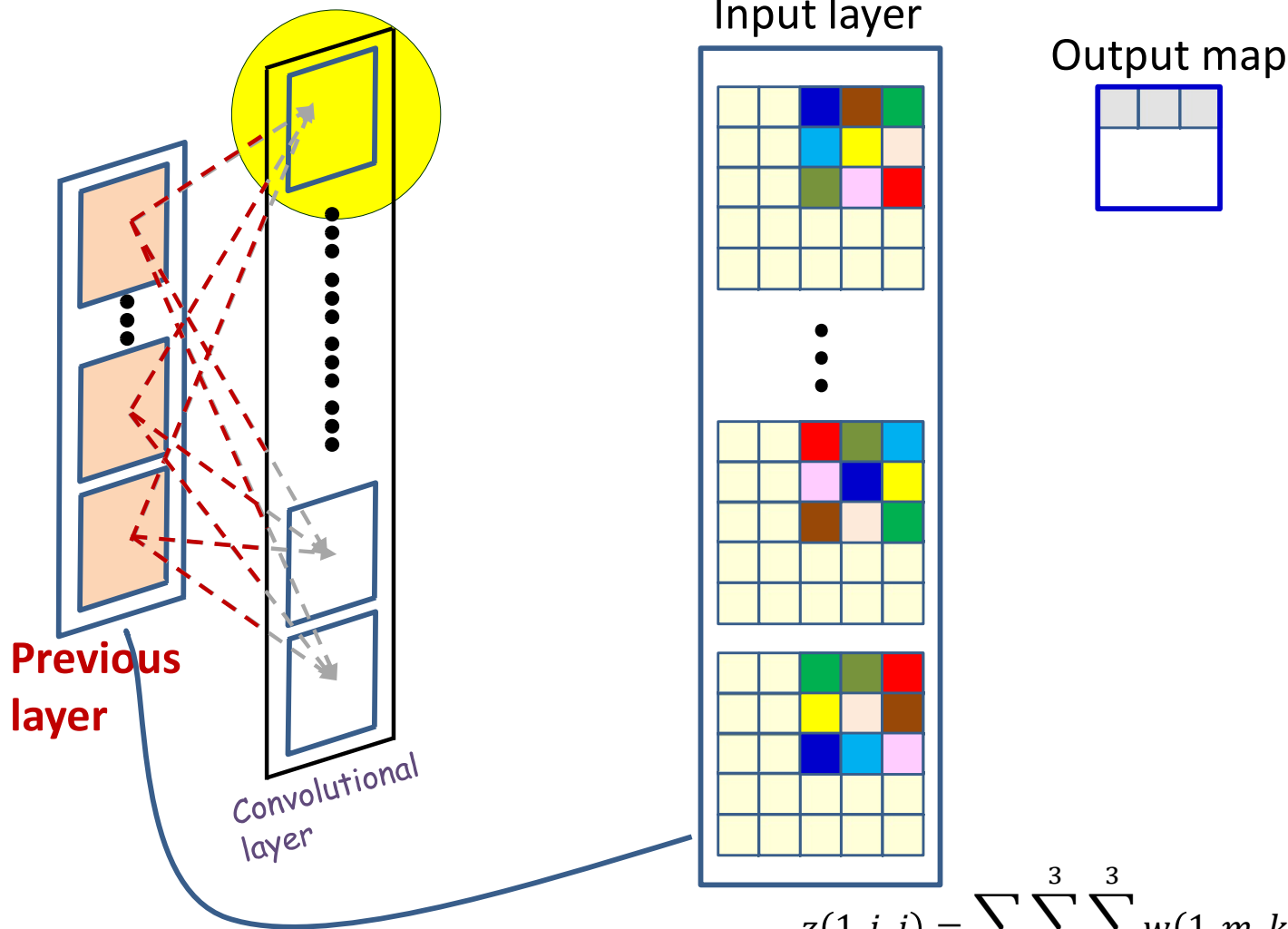
# What really happens



$$z(1, i, j) = \sum_m \sum_{k=1}^3 \sum_{l=1}^3 w(1, m, k, l) I(m, i + l - 1, j + k - 1) + b$$

- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as *size of the filter* x *no. of maps in previous layer*

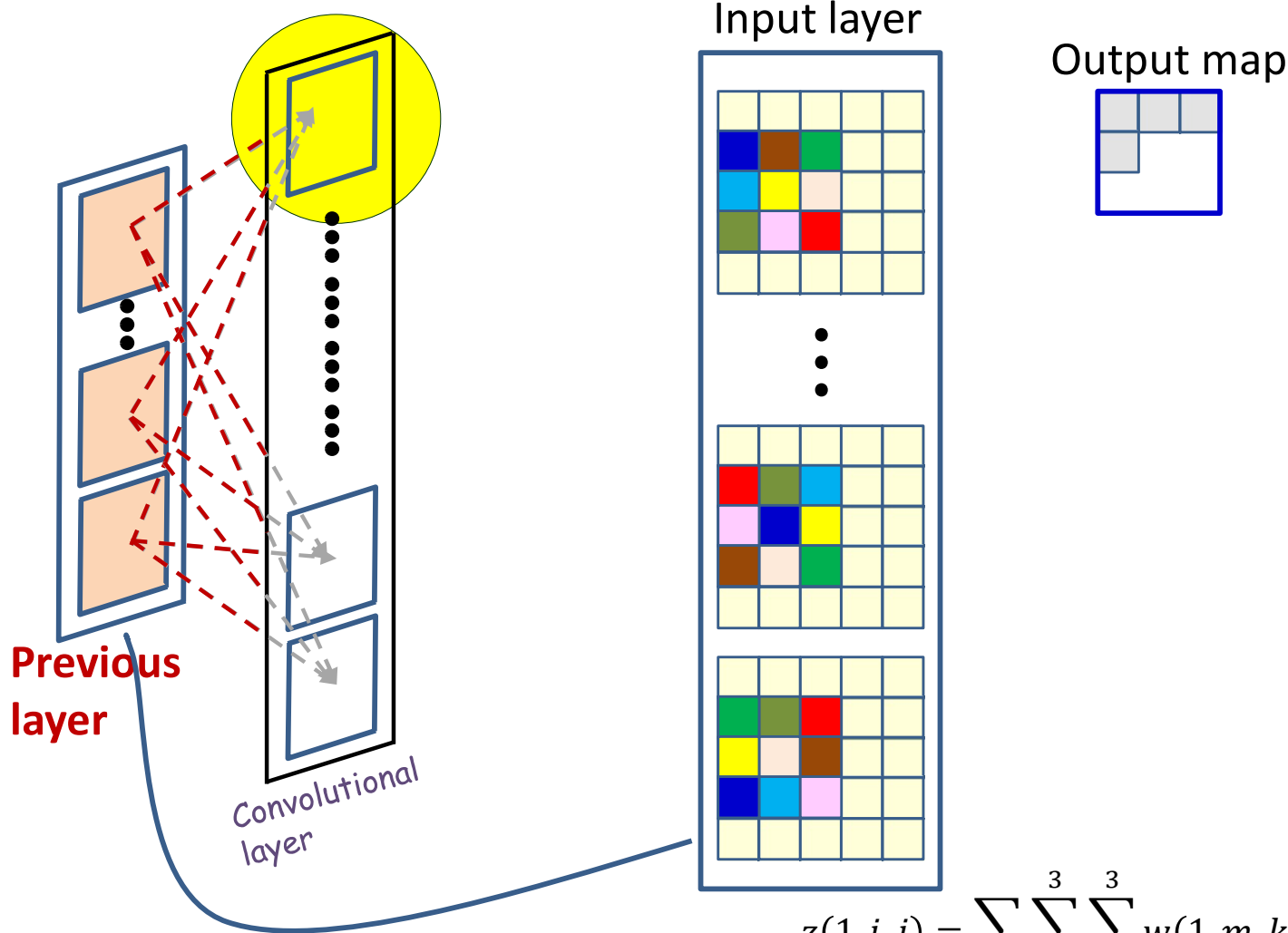
# What really happens



$$z(1, i, j) = \sum_m \sum_{k=1}^3 \sum_{l=1}^3 w(1, m, k, l) I(m, i + l - 1, j + k - 1) + b$$

- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as *size of the filter* x *no. of maps in previous layer*

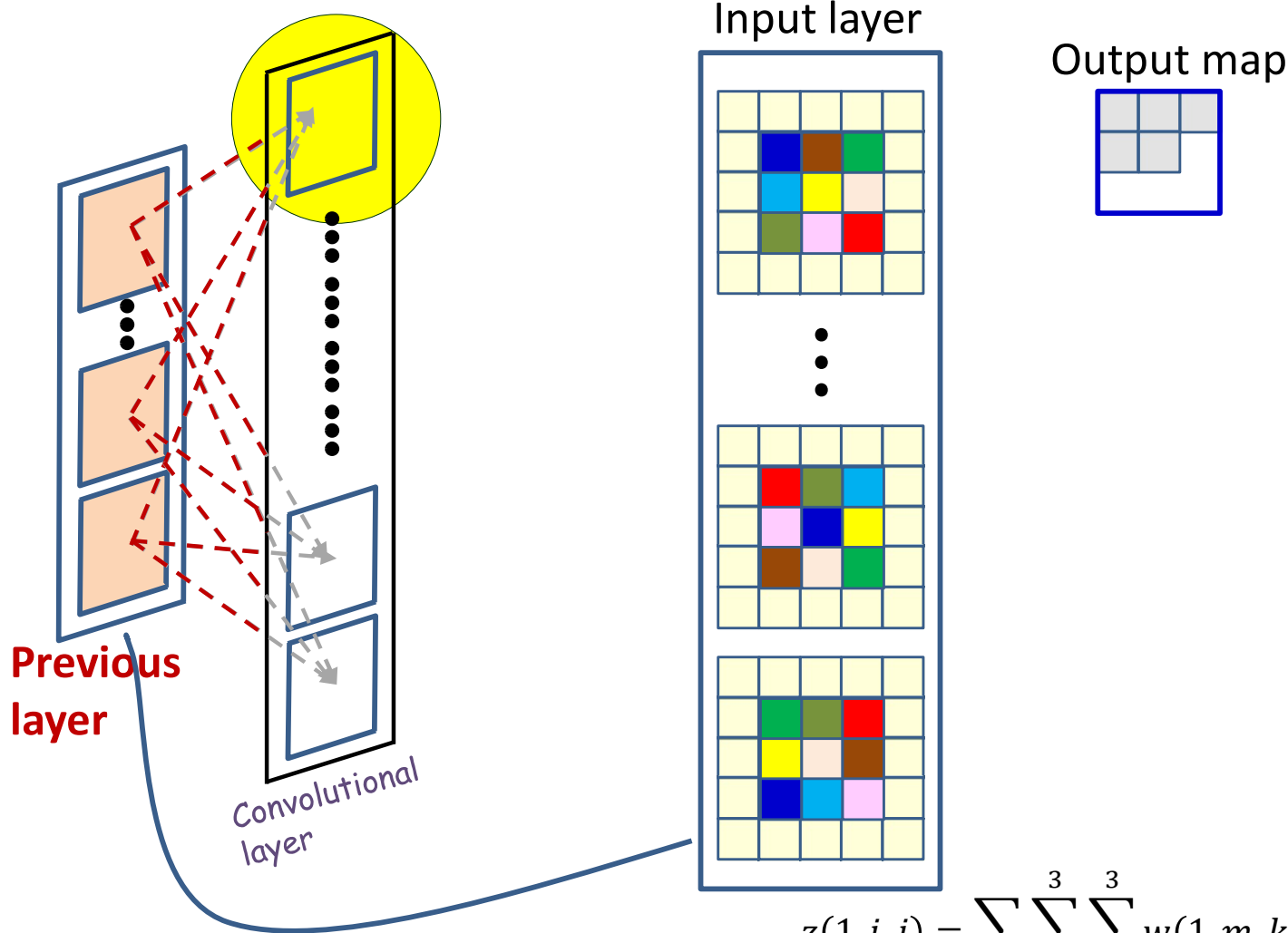
# What really happens



$$z(1, i, j) = \sum_m \sum_{k=1}^3 \sum_{l=1}^3 w(1, m, k, l) I(m, i + l - 1, j + k - 1) + b$$

- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as *size of the filter* x *no. of maps in previous layer*

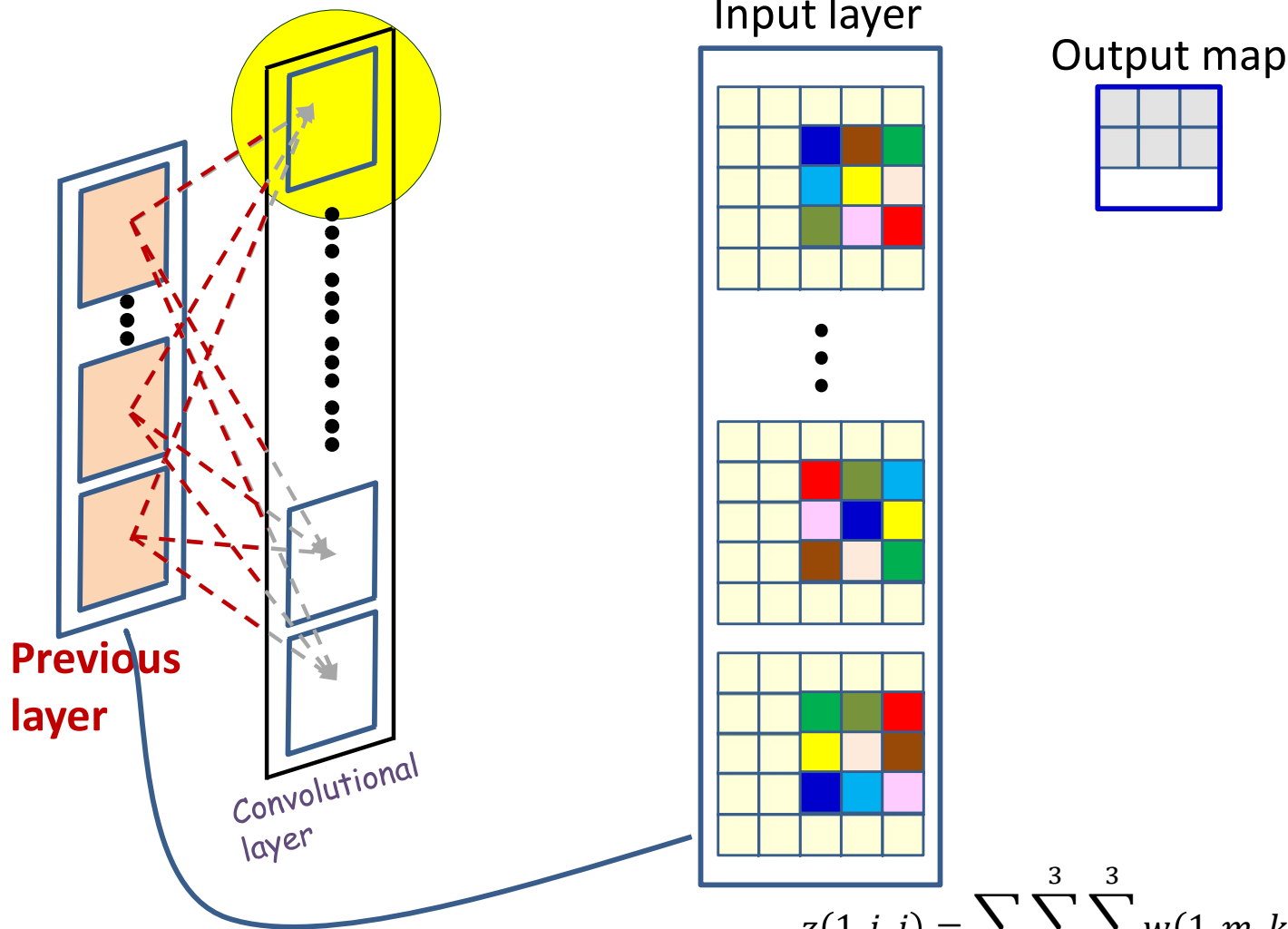
# What really happens



$$z(1, i, j) = \sum_m \sum_{k=1}^3 \sum_{l=1}^3 w(1, m, k, l) I(m, i + l - 1, j + k - 1) + b$$

- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as *size of the filter* x *no. of maps in previous layer*

# What really happens

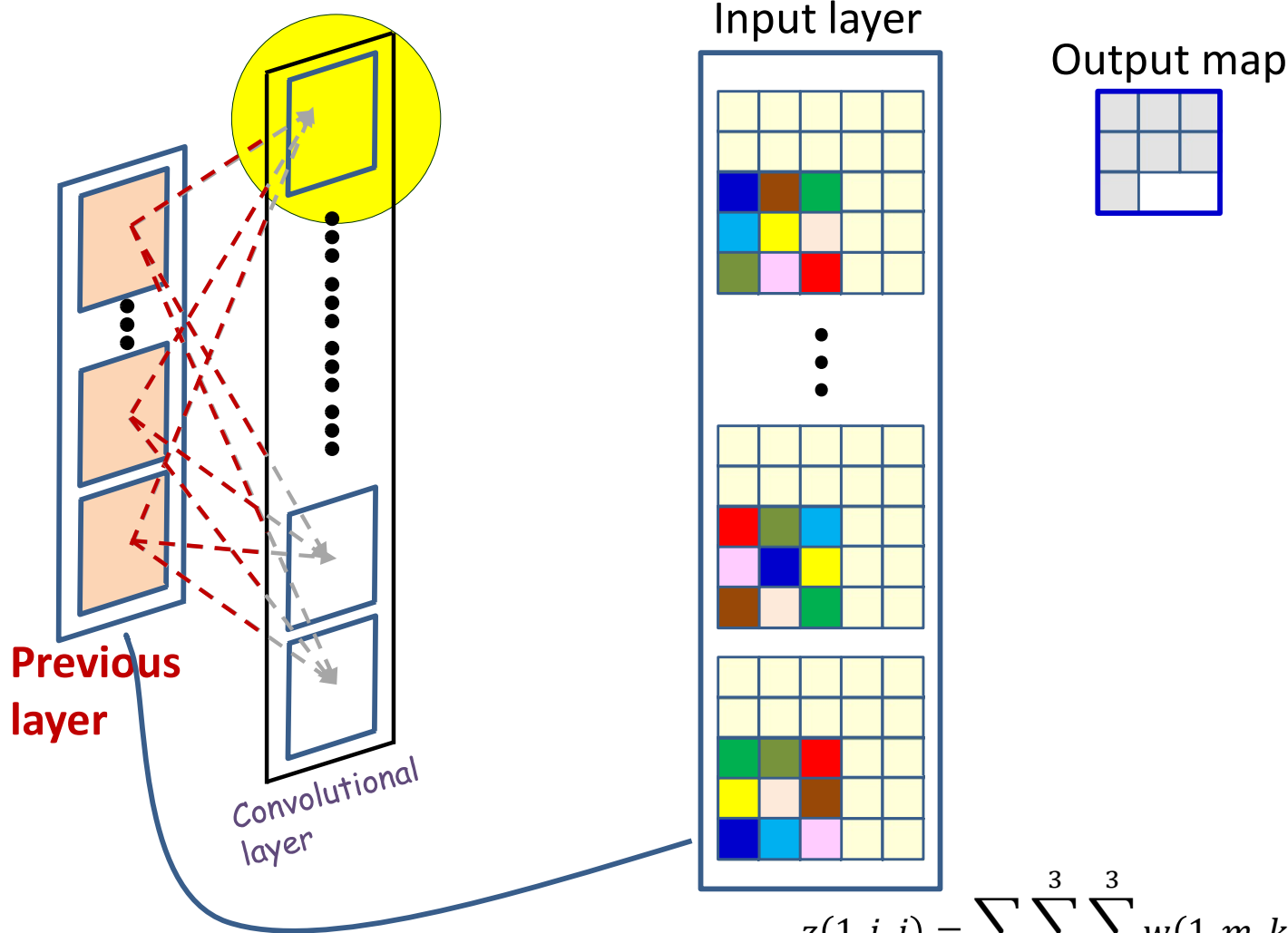


$$z(1, i, j) = \sum_m \sum_{k=1}^3 \sum_{l=1}^3 w(1, m, k, l) I(m, i + l - 1, j + k - 1) + b$$

- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as *size of the filter* x *no. of maps in previous layer*

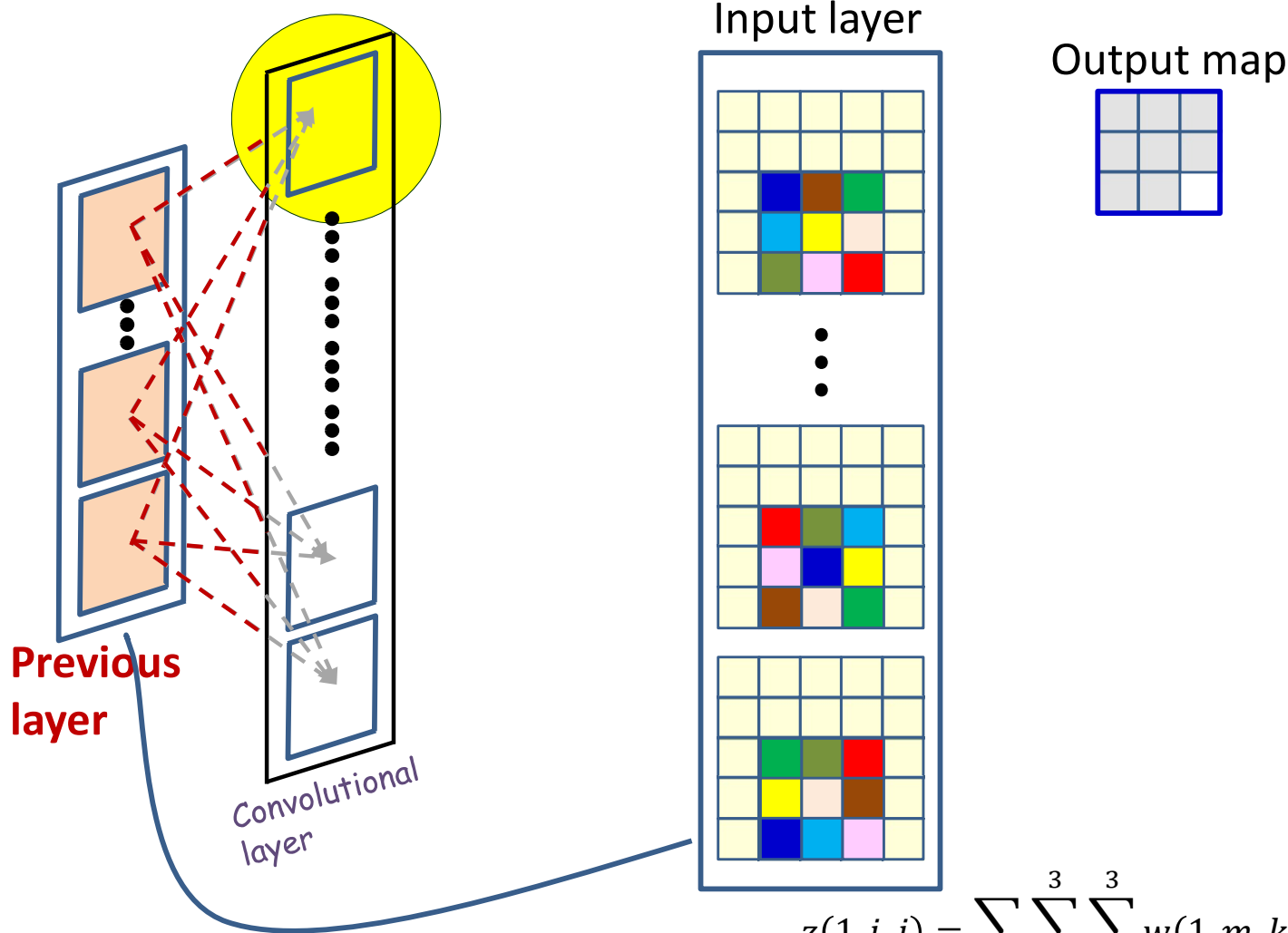


# What really happens



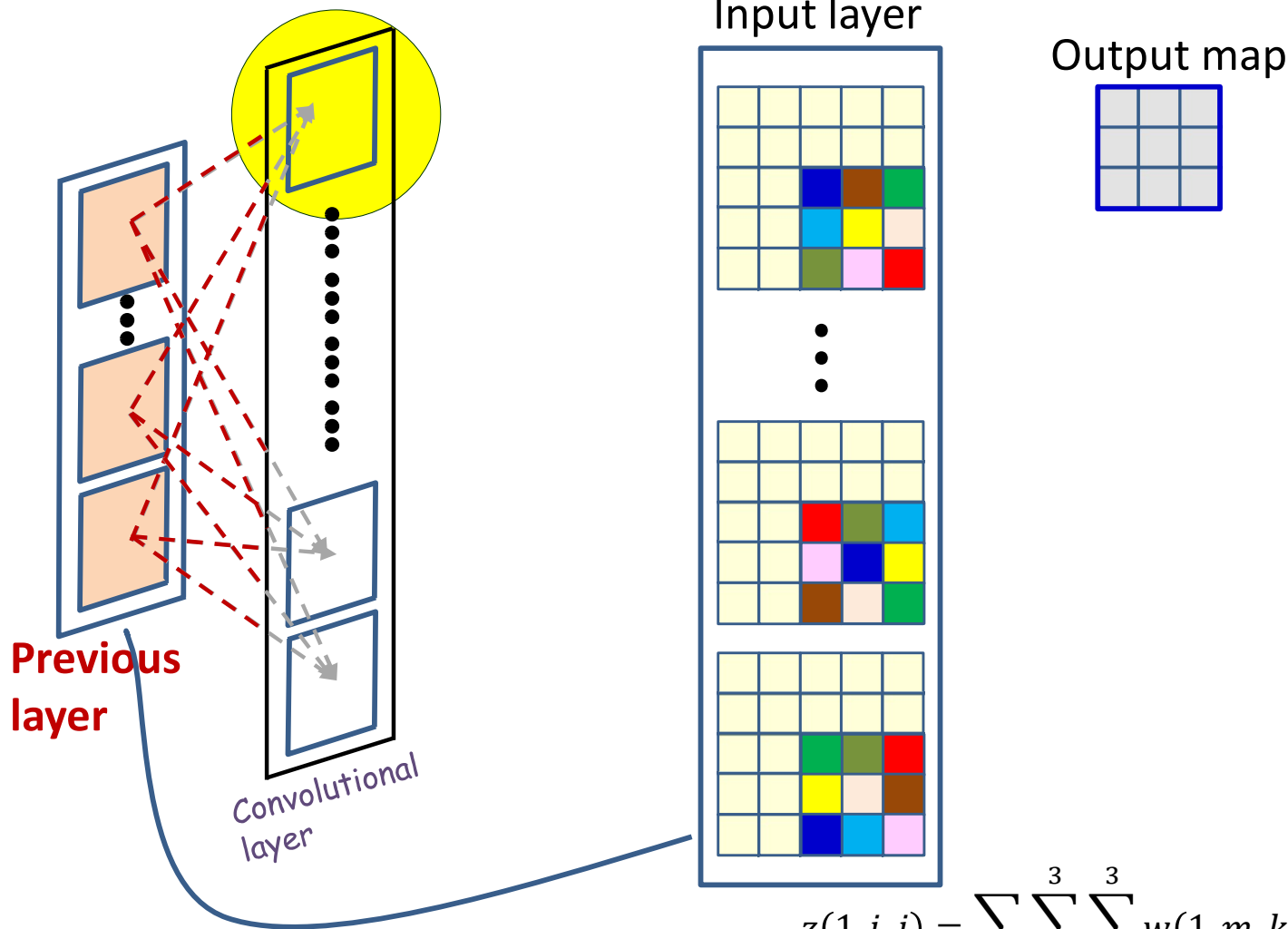
- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as *size of the filter* x *no. of maps in previous layer*

# What really happens



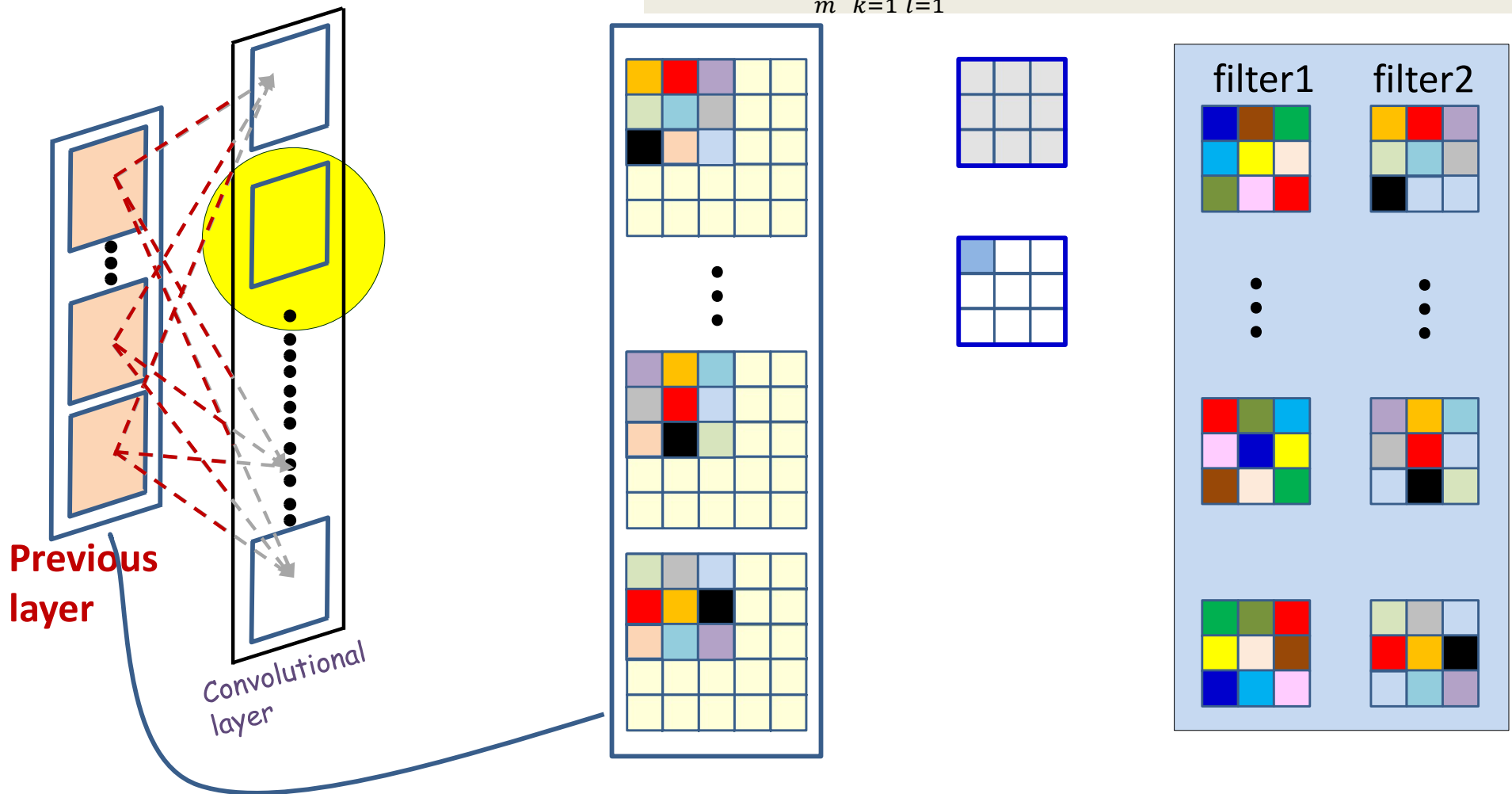
- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as *size of the filter* x *no. of maps in previous layer*

# What really happens

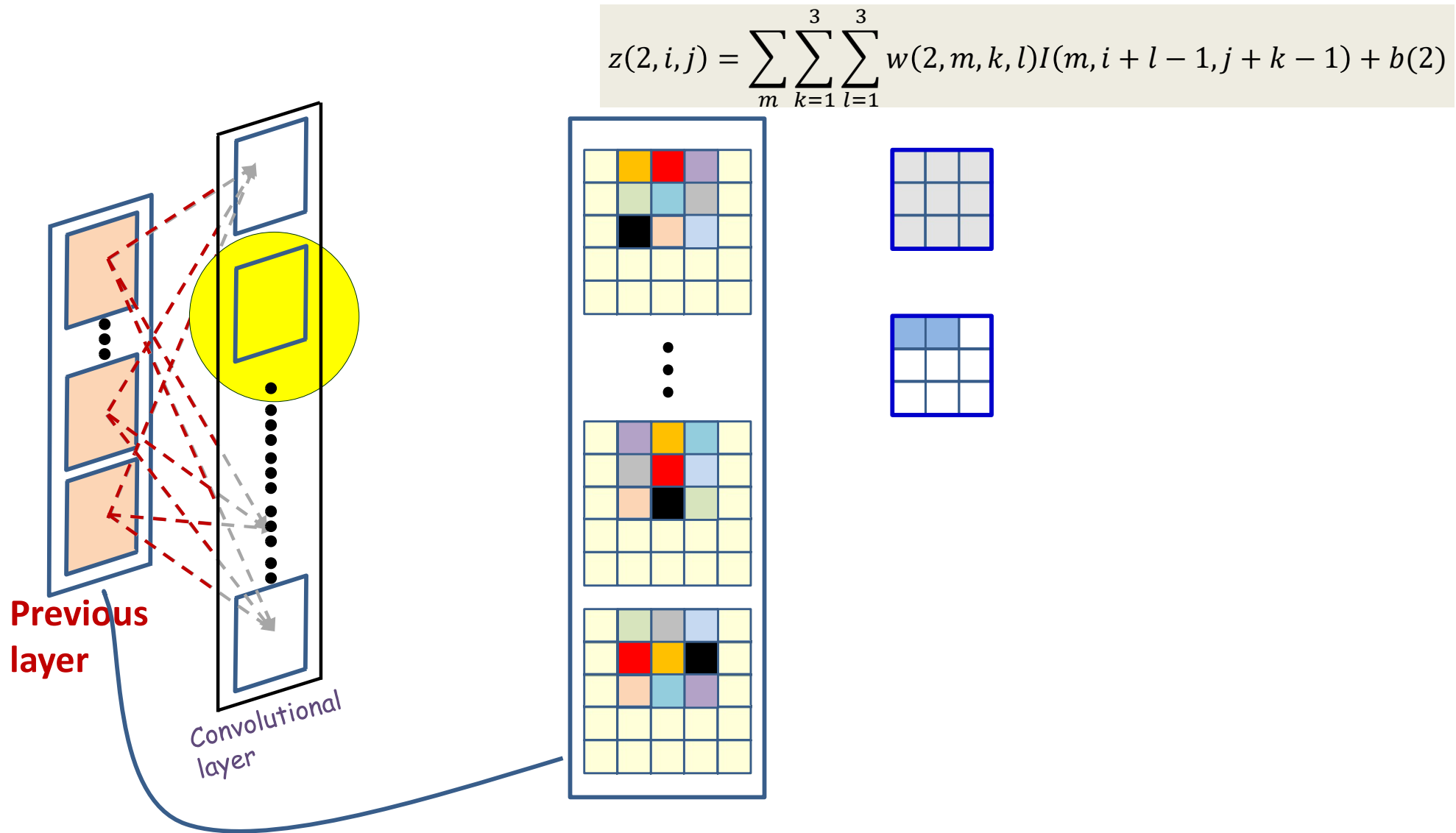


- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as *size of the filter* x *no. of maps in previous layer*

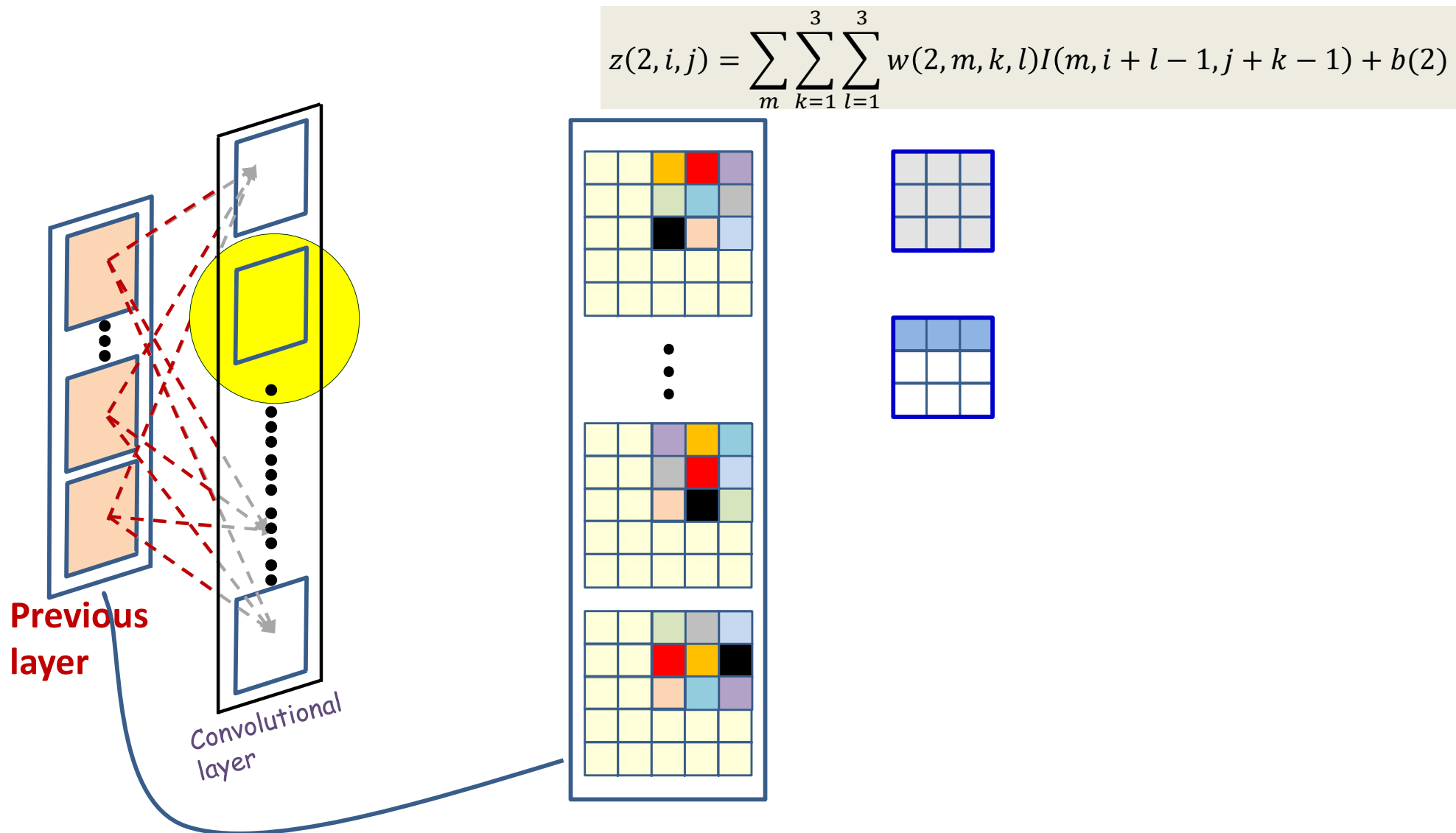
$$z(2, i, j) = \sum_m \sum_{k=1}^3 \sum_{l=1}^3 w(2, m, k, l) I(m, i + l - 1, j + k - 1) + b(2)$$



- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as *size of the filter* x *no. of maps in previous layer*

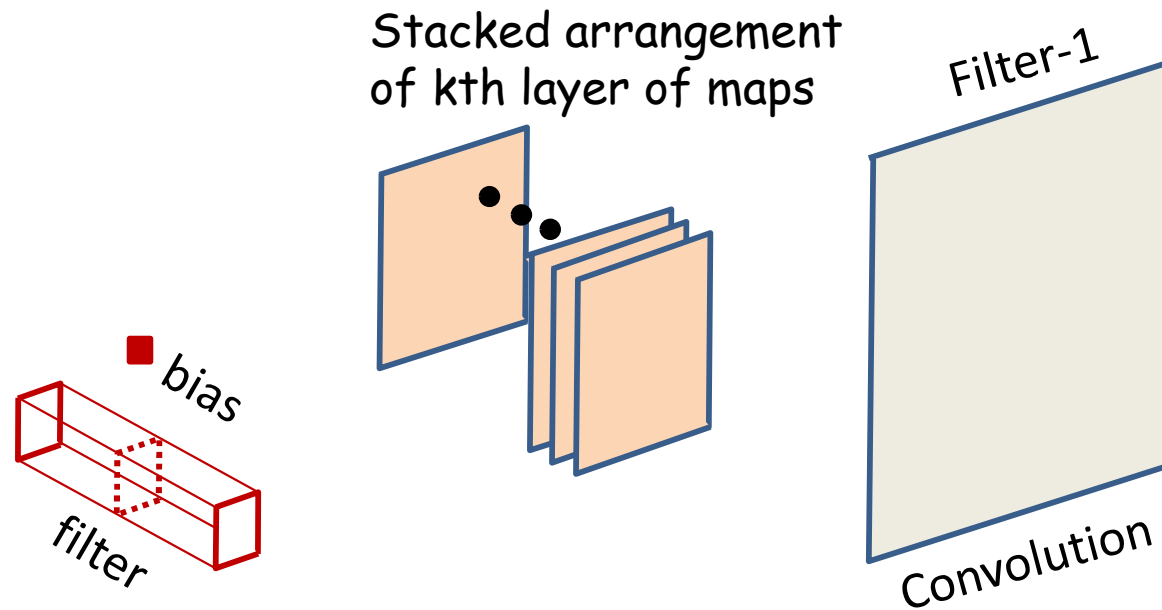


- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as *size of the filter* x *no. of maps in previous layer*



- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as *size of the filter* x *no. of maps in previous layer*

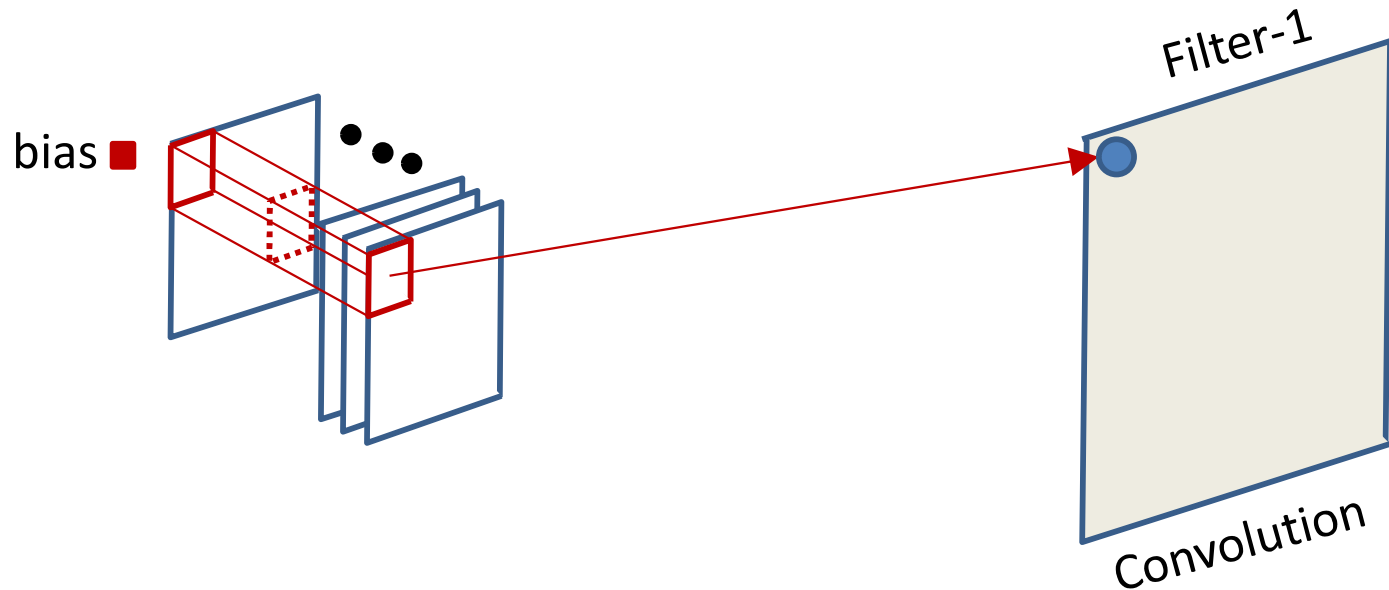
# A different view



Filter applied to kth layer of maps  
(convolutive component plus bias)

- ..A *stacked* arrangement of planes
- We can view the joint processing of the various maps as processing the stack using a three-dimensional filter

# The “cube” view of input maps

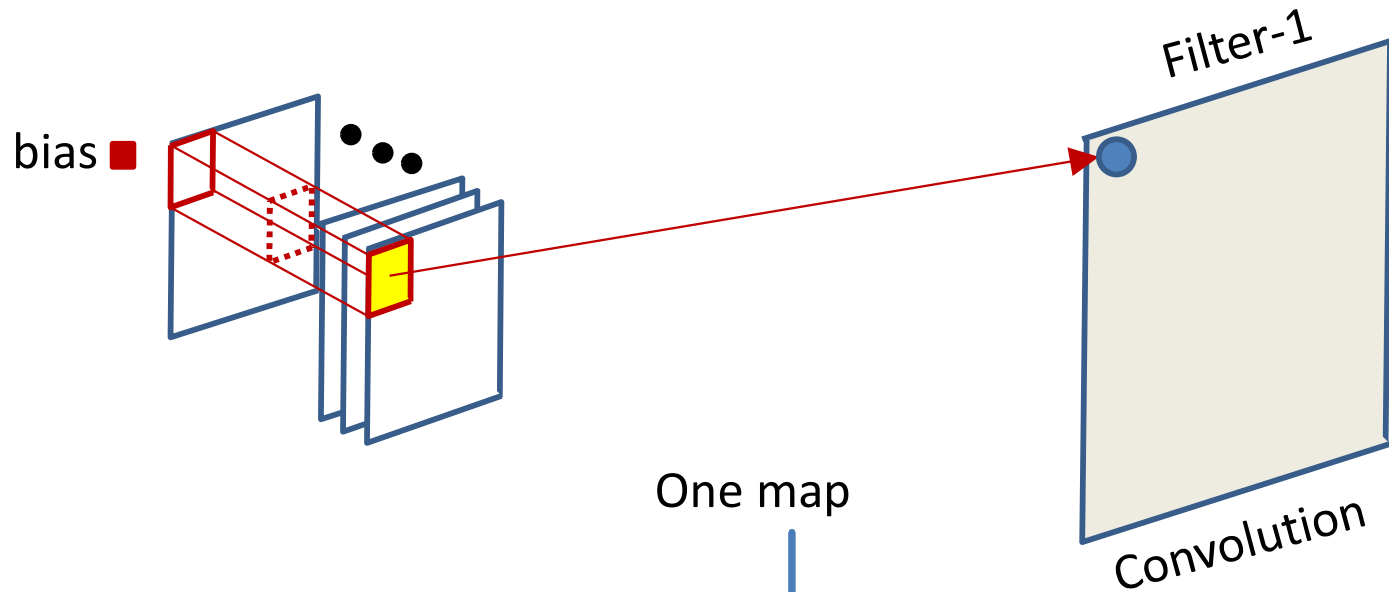


$$z(s, i, j) = \sum_p \sum_{k=1}^L \sum_{l=1}^L w(s, p, k, l) Y(p, i + l - 1, j + k - 1) + b(s)$$

- The computation of the convolutional map at any location *sums* the convolutional outputs *at all planes*



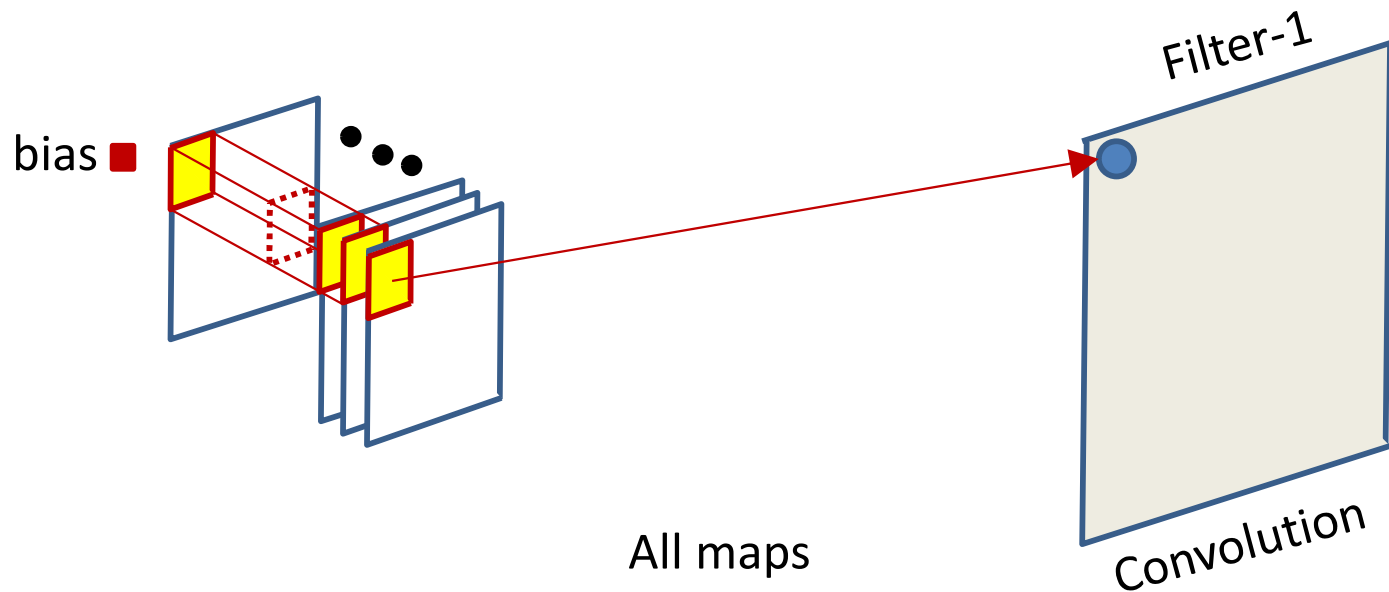
# The “cube” view of input maps



$$z(s, i, j) = \sum_p \sum_{k=1}^L \sum_{l=1}^L w(s, p, k, l) Y(p, i + l - 1, j + k - 1) + b(s)$$

- The computation of the convolutional map at any location *sums* the convolutional outputs *at all planes*

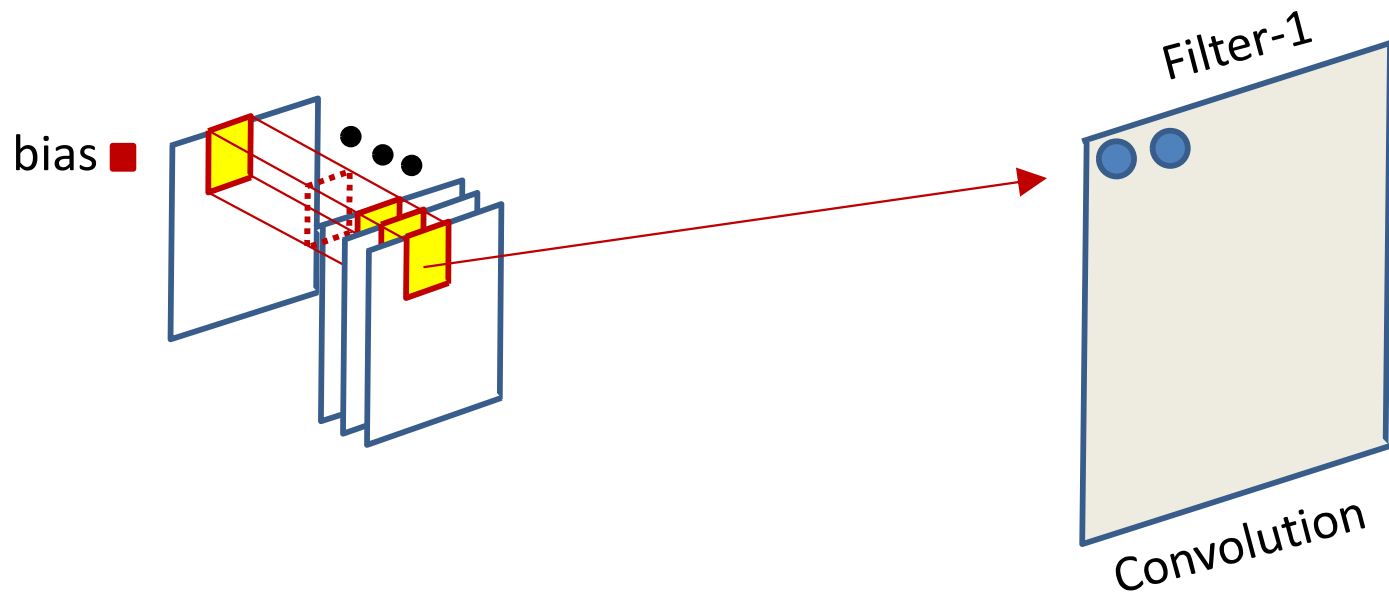
# The “cube” view of input maps



$$z(s, i, j) = \sum_p \sum_{k=1}^L \sum_{l=1}^L w(s, p, k, l) Y(p, i + l - 1, j + k - 1) + b(s)$$

- The computation of the convolutional map at any location *sums* the convolutional outputs *at all planes*

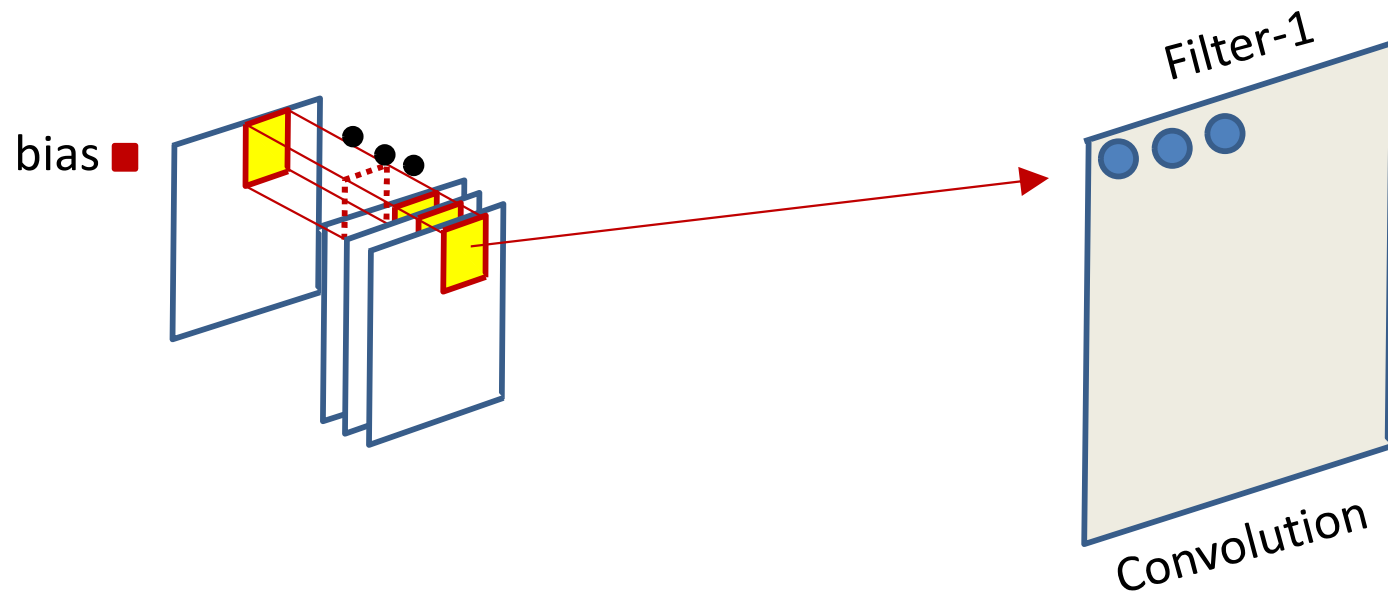
# The “cube” view of input maps



$$z(s, i, j) = \sum_p \sum_{k=1}^L \sum_{l=1}^L w(s, p, k, l) Y(p, i + l - 1, j + k - 1) + b(s)$$

- The computation of the convolutional map at any location *sums* the convolutional outputs *at all planes*

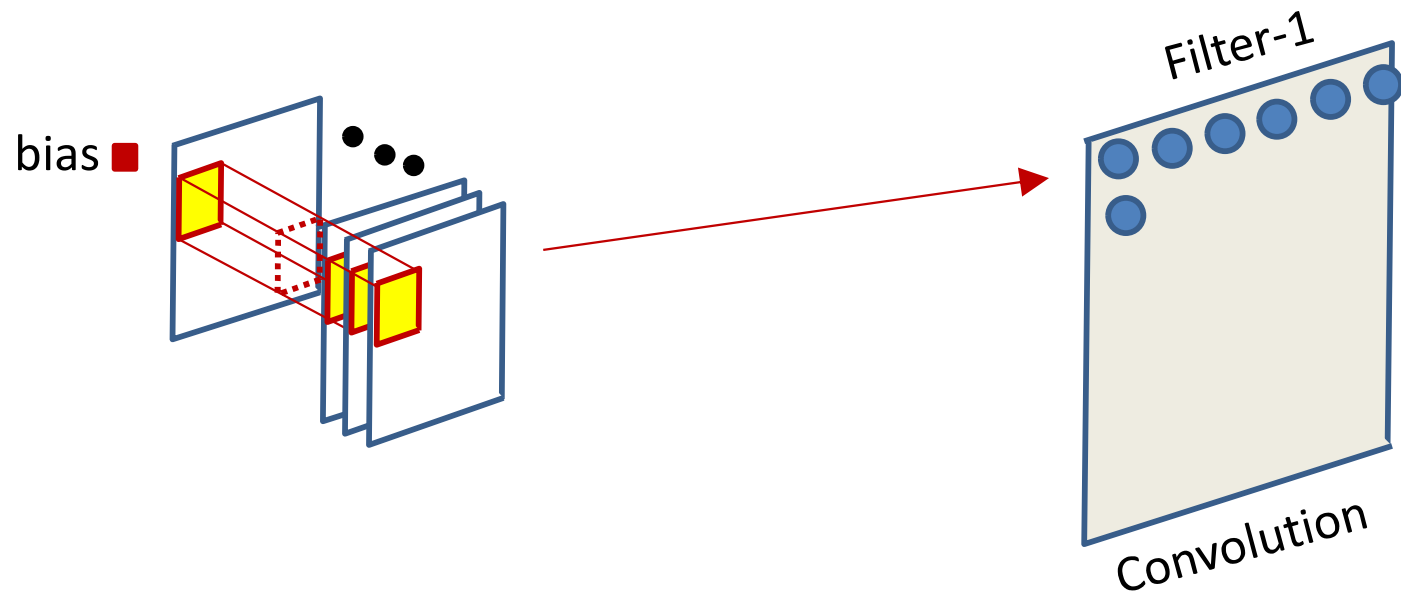
# The “cube” view of input maps



$$z(s, i, j) = \sum_p \sum_{k=1}^L \sum_{l=1}^L w(s, p, k, l) Y(p, i + l - 1, j + k - 1) + b(s)$$

- The computation of the convolutional map at any location *sums* the convolutional outputs *at all planes*

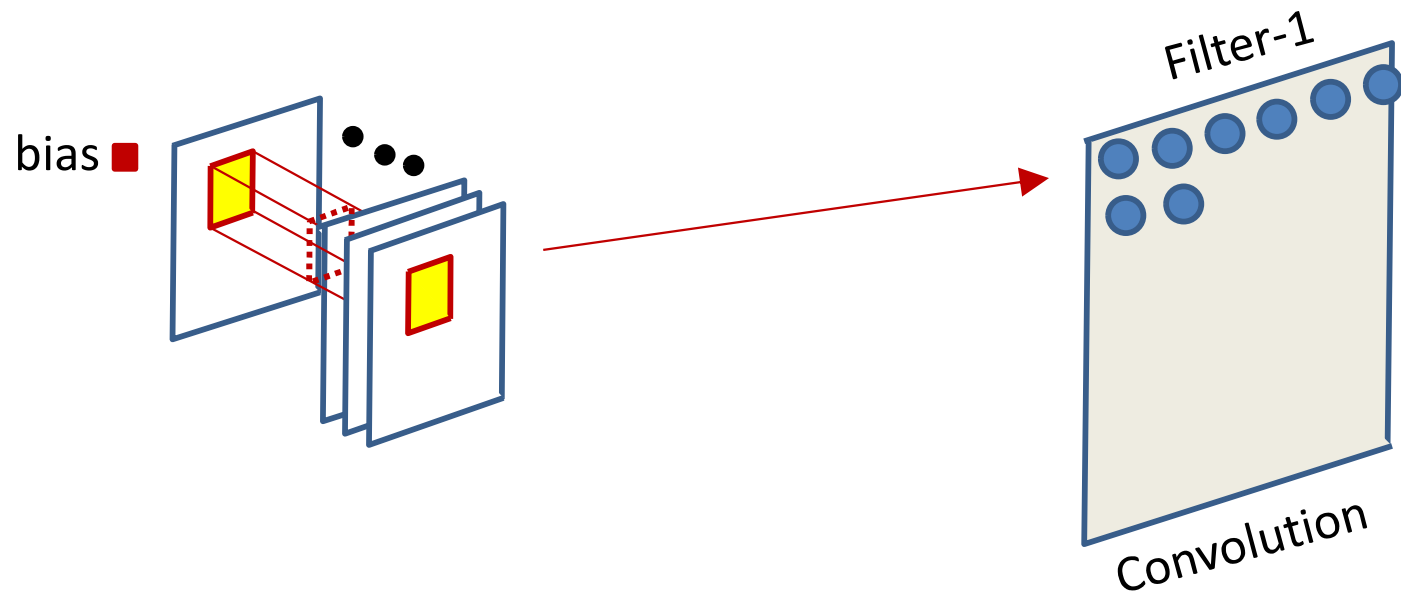
# The “cube” view of input maps



$$z(s, i, j) = \sum_p \sum_{k=1}^L \sum_{l=1}^L w(s, p, k, l) Y(p, i + l - 1, j + k - 1) + b(s)$$

- The computation of the convolutional map at any location *sums* the convolutional outputs *at all planes*

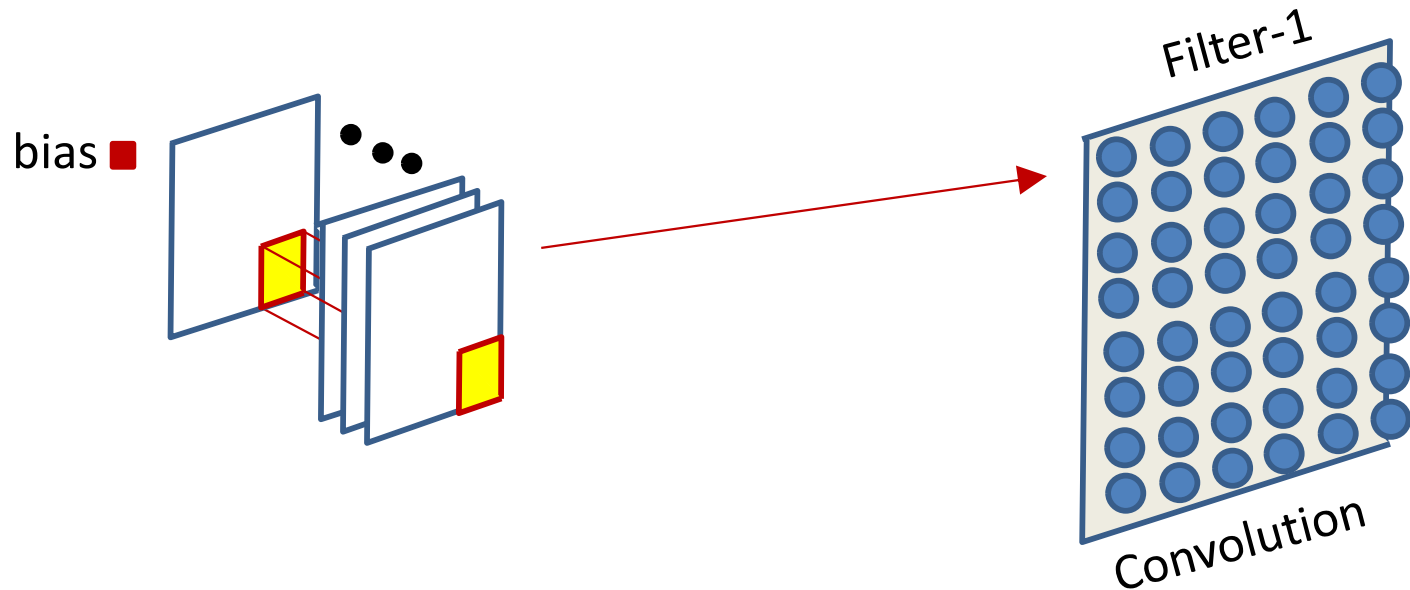
# The “cube” view of input maps



$$z(s, i, j) = \sum_p \sum_{k=1}^L \sum_{l=1}^L w(s, p, k, l) Y(p, i + l - 1, j + k - 1) + b(s)$$

- The computation of the convolutional map at any location *sums* the convolutional outputs *at all planes*

# The “cube” view of input maps



$$z(s, i, j) = \sum_p \sum_{k=1}^L \sum_{l=1}^L w(s, p, k, l) Y(p, i + l - 1, j + k - 1) + b(s)$$

- The computation of the convolutional map at any location *sums* the convolutional outputs *at all planes*

# CNN: Vector notation to compute a single output map

The weight  $W(l, j)$  is now a 3D  $D_{l-1} \times K_1 \times K_1$  tensor (assuming square receptive fields)

The product in blue is a tensor inner product with a scalar output

$Y(0) = \text{Image}$

for  $l = 1:L$     **# layers operate on vector at (x,y)**

```
for x = 1:Wl-1-K1+1
```

```
  for y = 1:Hl-1-K1+1
```

```
    for j = 1:Dl
```

```
      segment =  $Y(l-1, :, x:x+K_1-1, y:y+K_1-1)$     #3D tensor
```

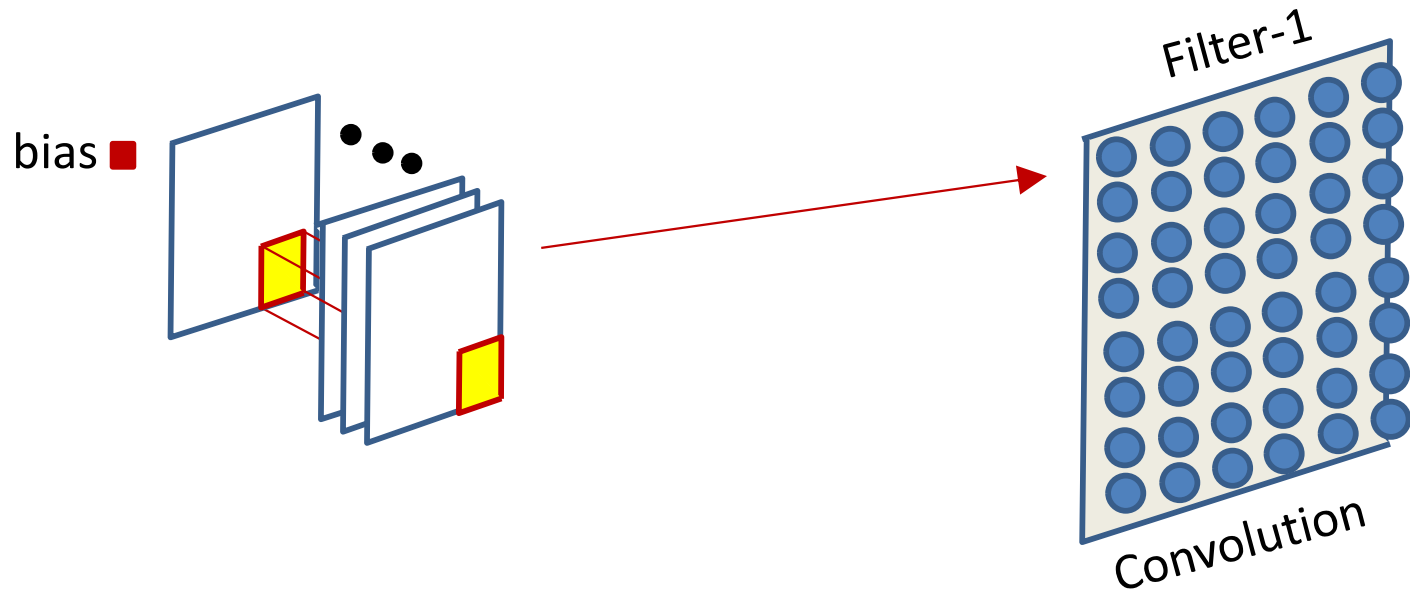
```
       $z(l, j, x, y)$  =  $W(l, j)$  . segment    #tensor inner prod.
```

```
       $Y(l, j, x, y)$  = activation( $z(l, j, x, y)$ )
```

```
Y = softmax( {  $Y(L, :, :, :)$  } )
```



# Engineering consideration: The size of the result of the convolution



- The size of the output of the convolution operation depends on implementation factors
  - The size of the input, the size of the filter
- And may not be identical to the size of the input
  - Let's take a brief look at this for completeness sake

# The size of the convolution

**0**  
bias

1	0	1
0	1	0
1	0	1

**Filter**

1 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	0	0
0 <sub>x0</sub>	1 <sub>x1</sub>	1 <sub>x0</sub>	1	0
0 <sub>x1</sub>	0 <sub>x0</sub>	1 <sub>x1</sub>	1	1
0	0	1	1	0
0	1	1	0	0

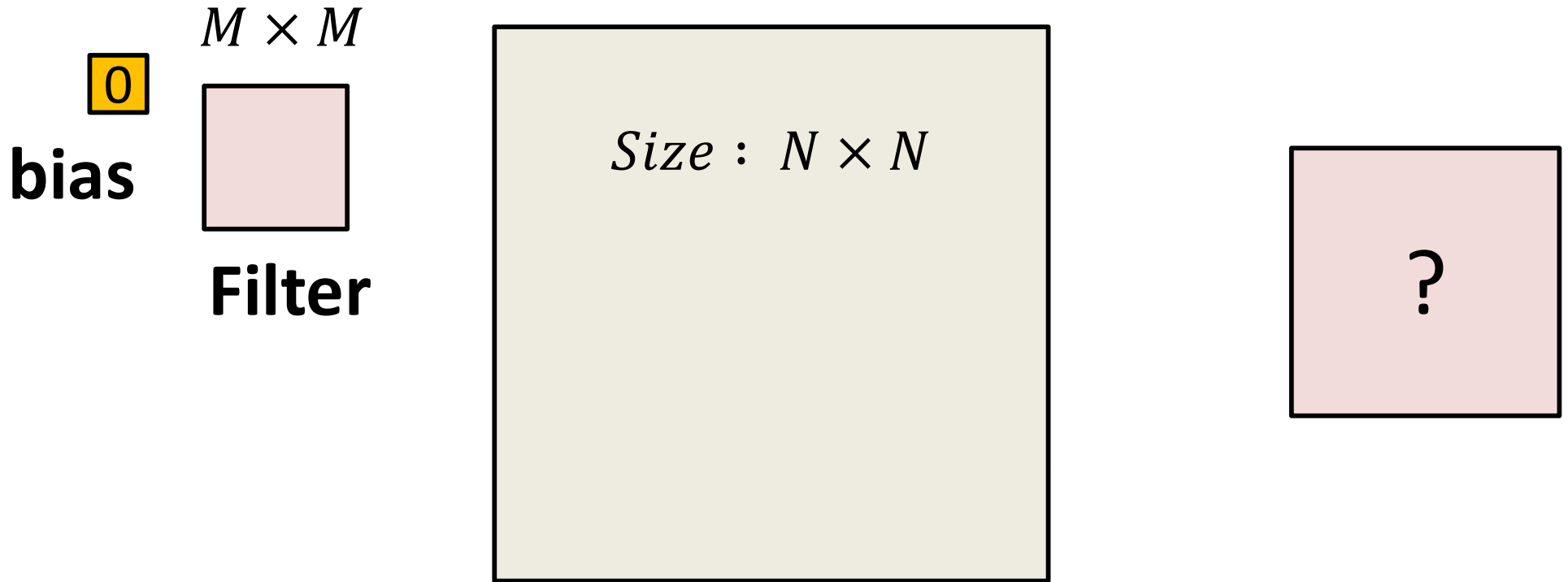
**Input Map**

4		

**Convolved  
Feature**

- Image size: 5x5
- Filter: 3x3
- Output size = ?

# The size of the convolution

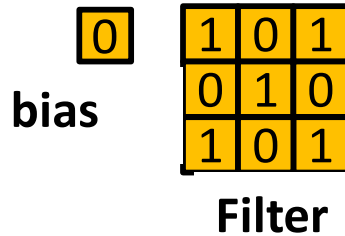


- Image size:  $N \times N$
- Filter:  $M \times M$
- Output size =  $(N-M)+1$  on each side

# Convolution Size

- Simple convolution size pattern:
  - Image size:  $N \times N$
  - Filter:  $M \times M$
  - **Output size (each side)**  $= (N - M) + 1$ 
    - Assuming you're not allowed to go beyond the edge of the input
- Results in a reduction in the output size
  - Sometimes not considered acceptable

# Solution



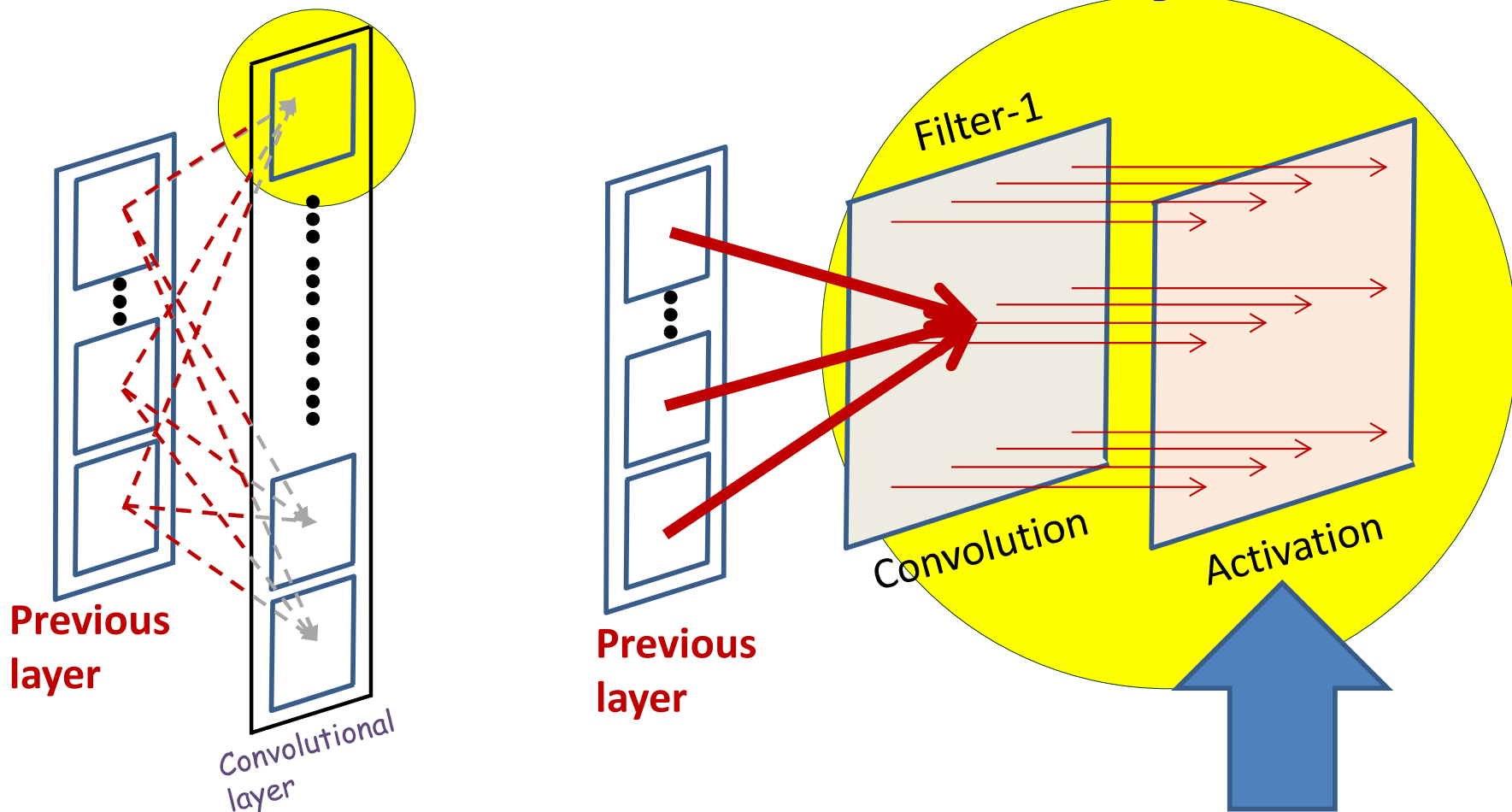
0	0	0	0	0	0	0
0	1	1	1	0	0	0
0	0	1	1	1	0	0
0	0	0	1	1	1	0
0	0	0	1	1	0	0
0	0	1	1	0	0	0
0	0	0	0	0	0	0

- Zero-pad the input
  - Pad the input image/map all around
  - Pad as symmetrically as possible, such that..
  - **The result of the convolution is the same size as the original image**

# Zero padding

- For an  $L$  width filter:
  - Odd  $L$  : Pad on both left and right with  $(L - 1)/2$  columns of zeros
  - Even  $L$  : Pad one side with  $L/2$  columns of zeros, and the other with  $\frac{L}{2} - 1$  columns of zeros
  - The resulting image is width  $N + L - 1$
  - The result of the convolution is width  $N$
- The top/bottom zero padding follows the same rules to maintain map height after convolution

# A convolutional layer



- The convolution operation results in an affine map
- An *Activation* is finally applied to every entry in the map

# Convolutional neural net:

## Vector notation

The weight  $W(l, j)$  is now a 4D  $D_1 \times D_{l-1} \times K_1 \times K_1$  tensor (assuming square receptive fields)

The product in blue is a tensor inner product with a scalar output

$Y(0) = \text{Image}$

for  $l = 1:L$     **# layers operate on vector at (x,y)**

```
for x = 1:Wl-1-K1+1
    for y = 1:Hl-1-K1+1
        segment =  $Y(l-1, :, x:x+K_1-1, y:y+K_1-1)$  #3D tensor
         $z(l, :, x, y)$  =  $W(l)$  .segment #tensor inner prod.
         $Y(l, :, x, y)$  = activation( $z(l, :, x, y)$ )
```

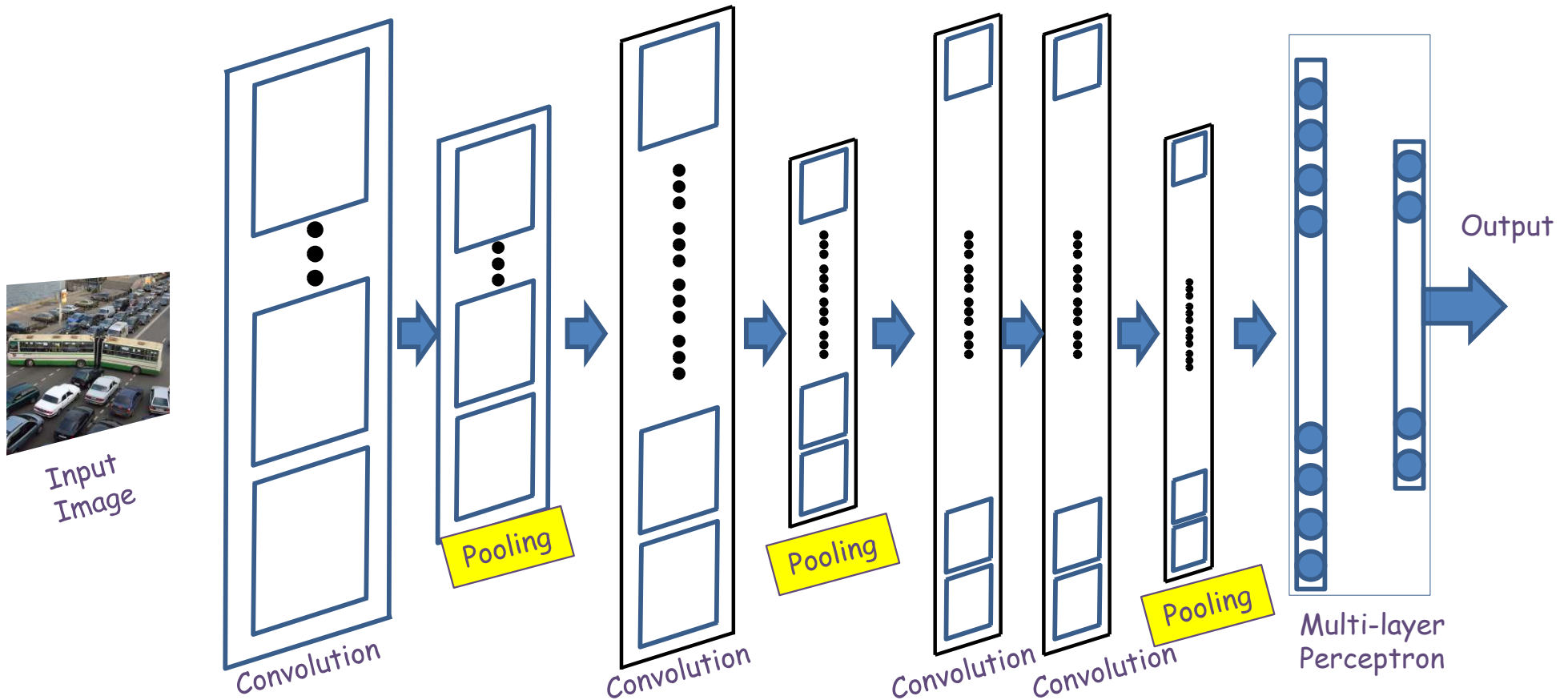
$Y = \text{softmax}(\{Y(L, :, :, :)\})$



# Convolution Summary

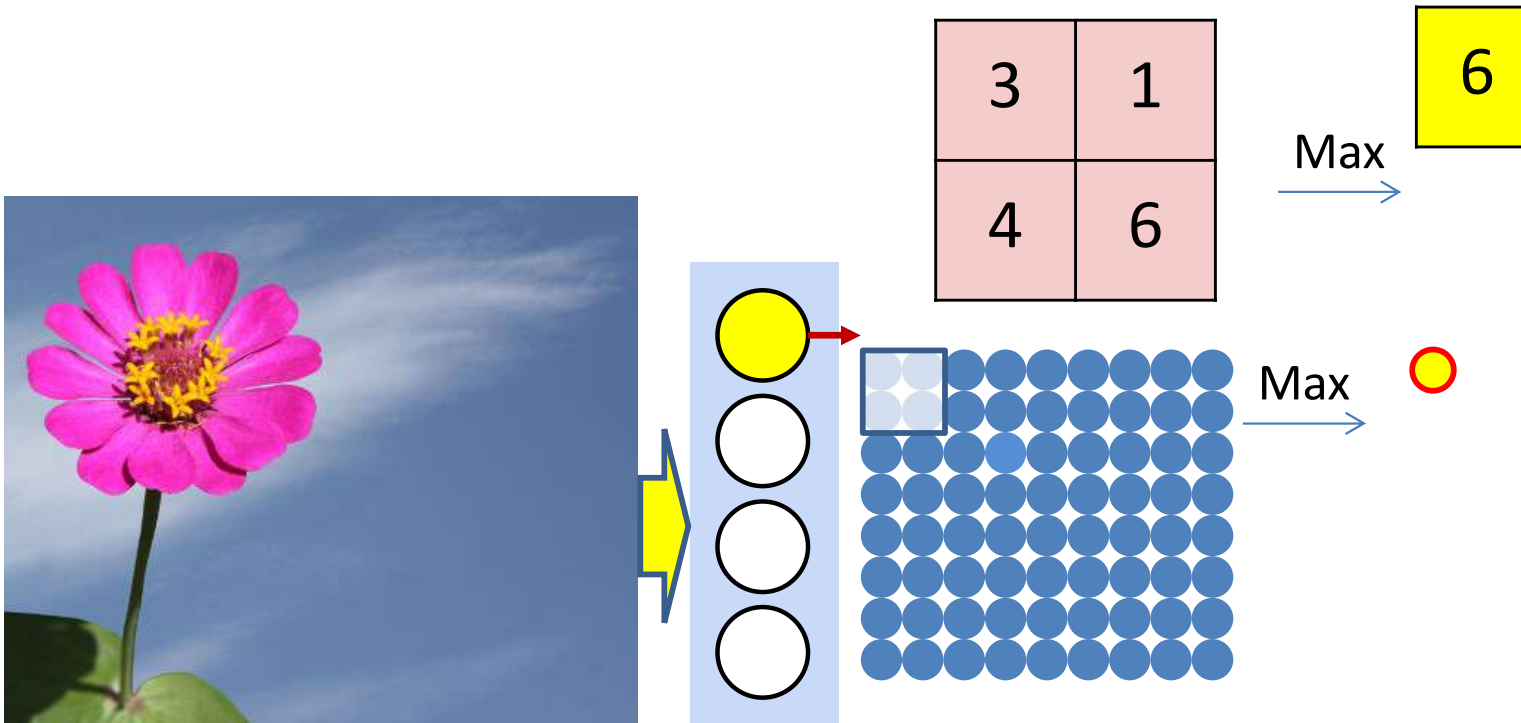
- Convolutional layers “scan” the input using a bank of “filters”
  - A “filter” is just a neuron in a scanning layer
- Each filter jointly scans the maps in the previous layer to produce an output “map”
  - As many output maps as ***filters*** (one output map per filter)
    - Regardless of the number of input maps
- We may have to pad the edges of the input maps to ensure that the output maps are the same size as input maps
  - If not, convolution loses rows/columns at the edges of the scan

# The other component: Pooling



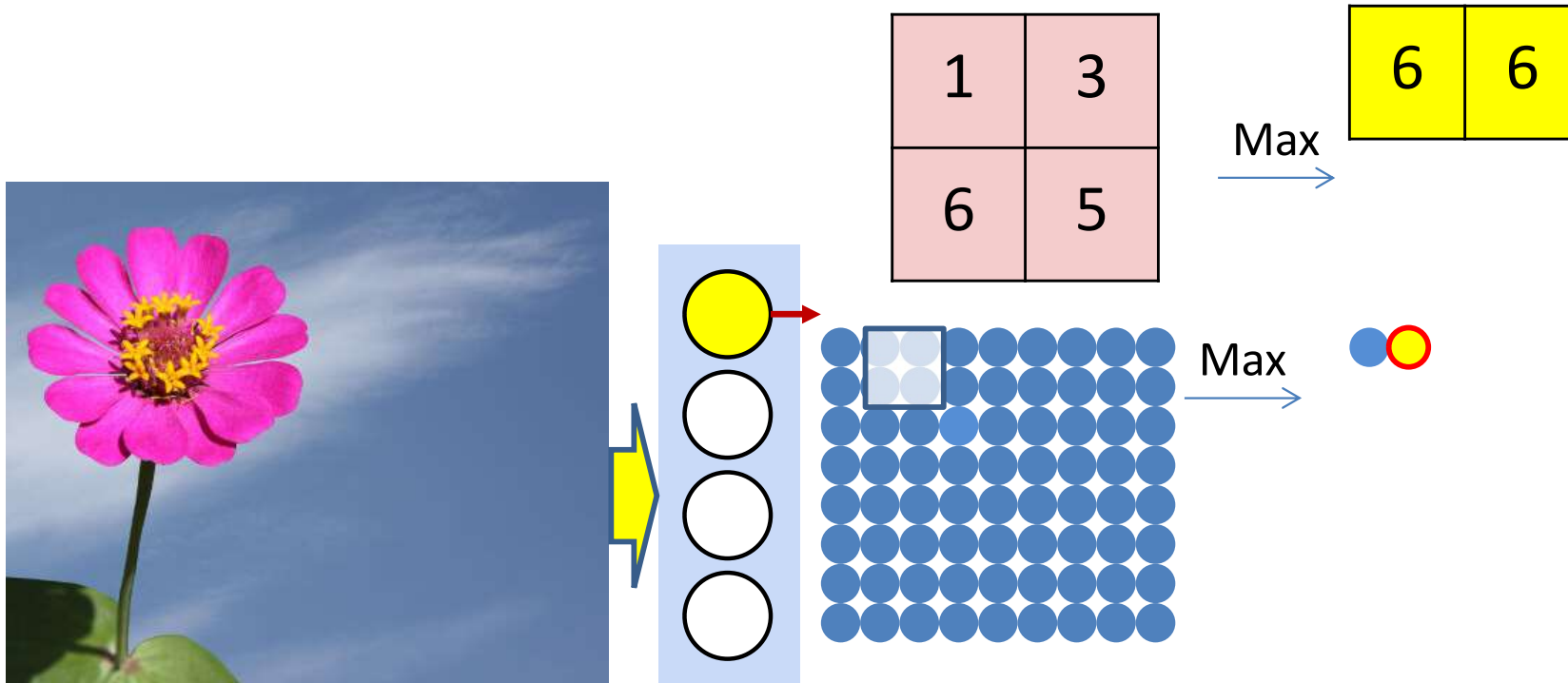
- Convolution (and activation) layers are followed intermittently by “pooling” layers
  - Typically (but not always) “max” pooling
  - Often, they alternate with convolution, though this is not necessary

# Max pooling



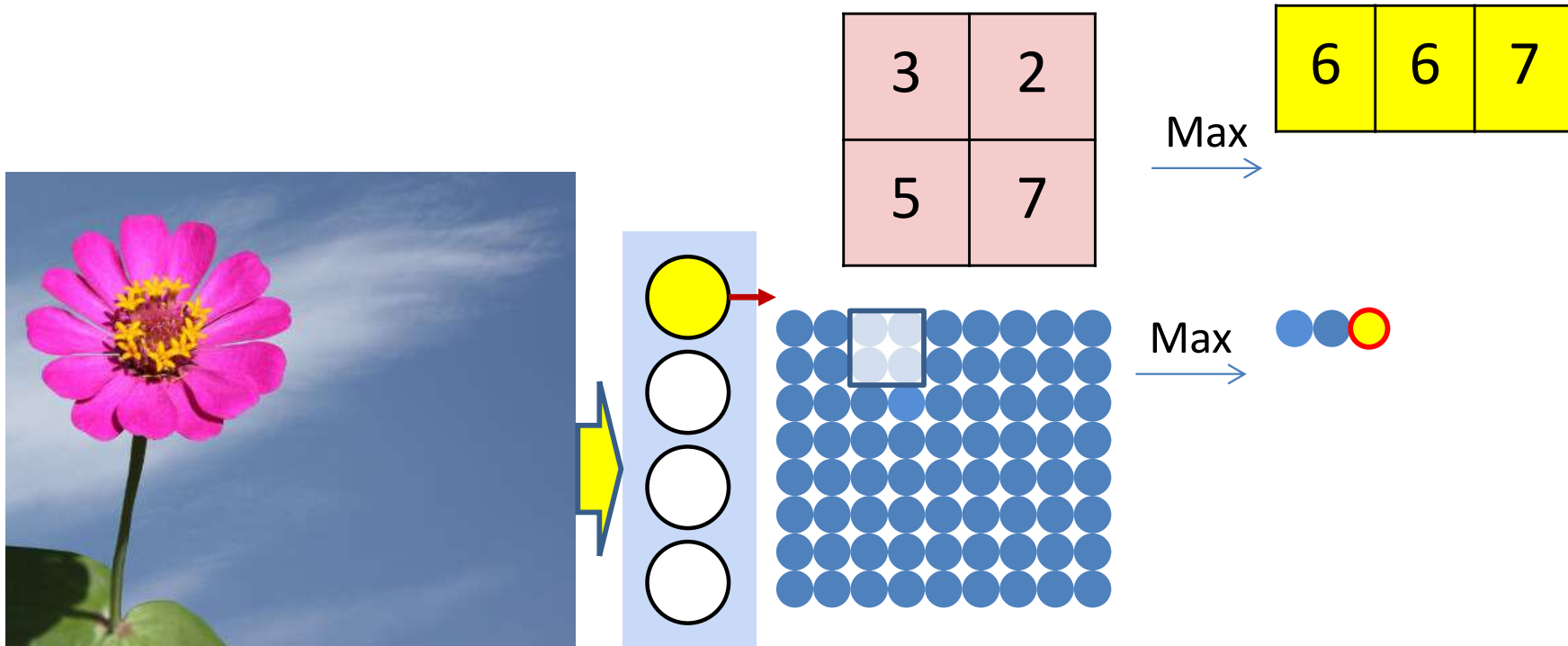
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

# Recall: Max pooling



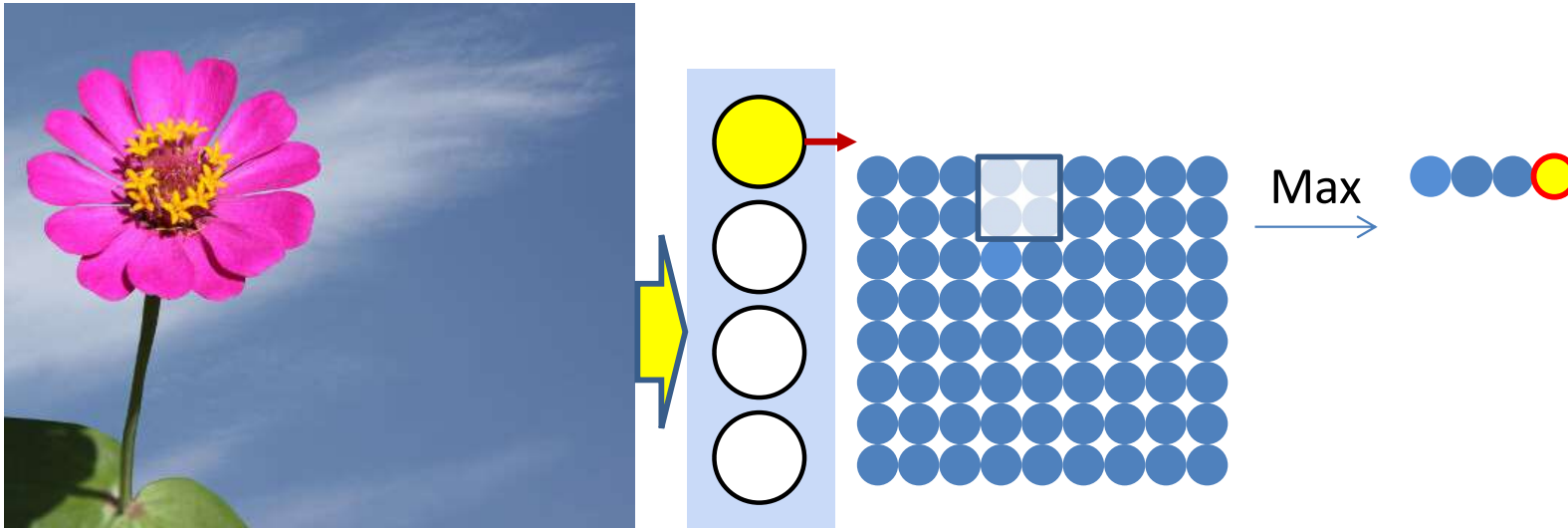
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

# Recall: Max pooling



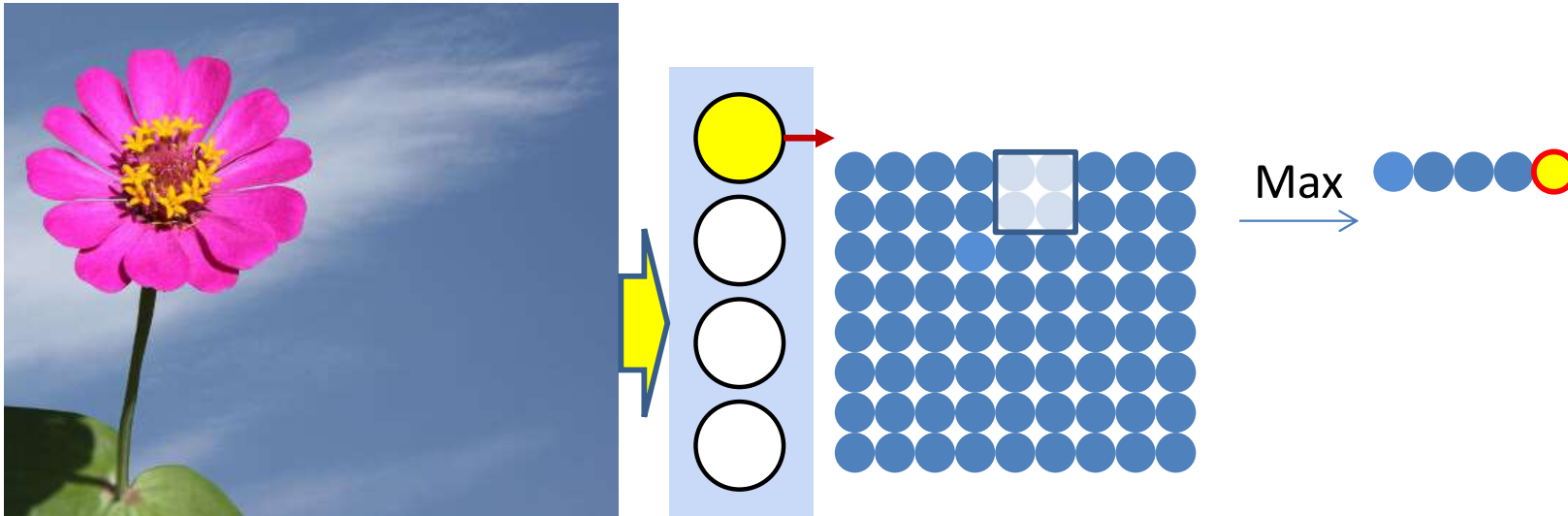
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

# Recall: Max pooling



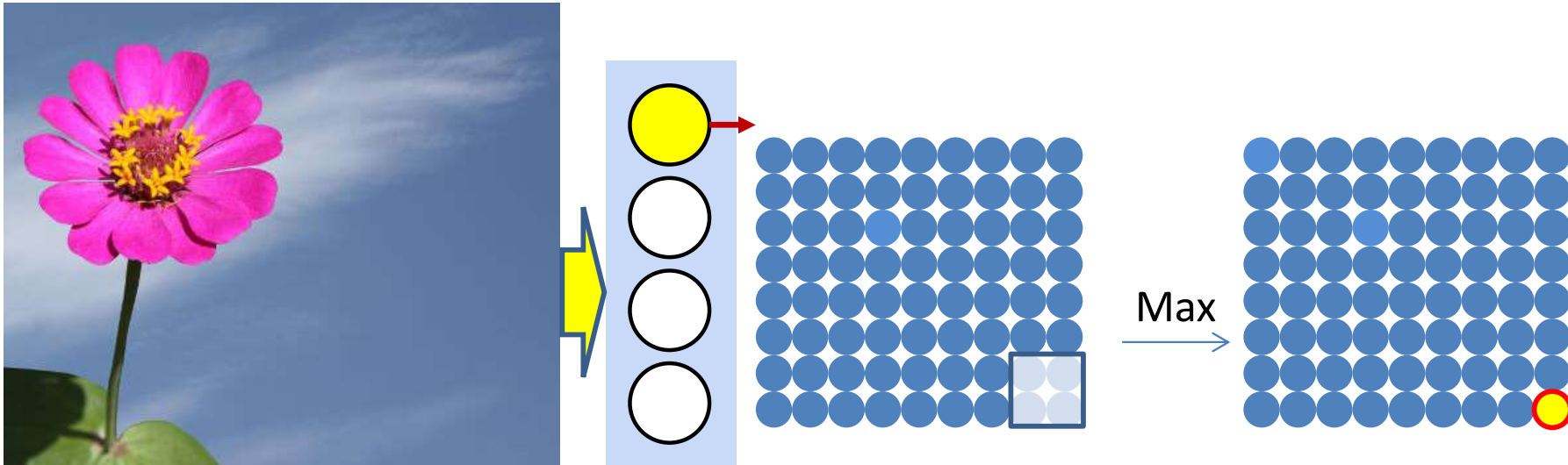
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

# Recall: Max pooling



- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

# Recall: Max pooling



- Max pooling scans with a stride of 1 confer jitter-robustness
  - Typically performed with a stride  $> 1$ , whereupon it also results in “downsampling”
    - We return to downsampling shortly



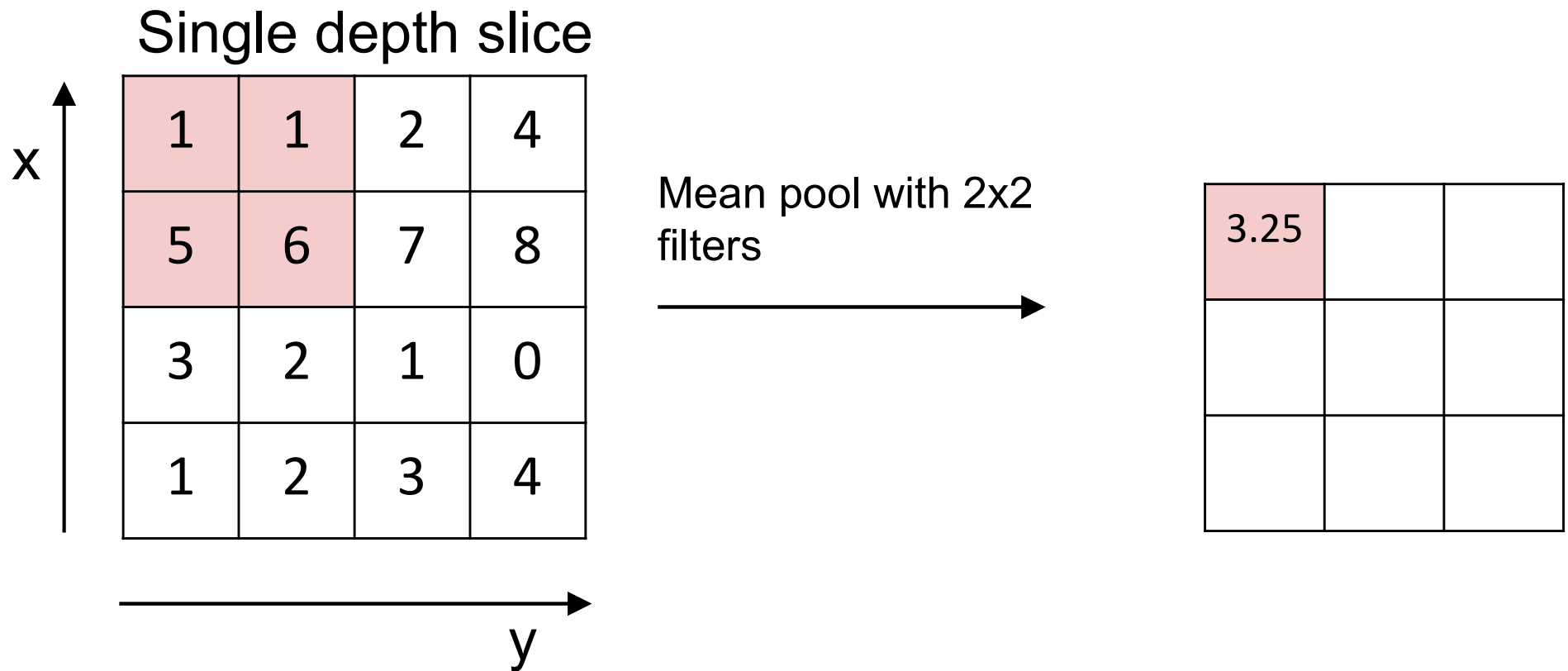
# Max Pooling layer at layer $l$

- a) Performed separately for every map ( $j$ ).
  - \*) Not combining multiple maps within a single max operation.
- b) Keeping track of location of max

## Max pooling

```
for j = 1:D1
    for x = 1:Wl-1-K1+1
        for y = 1:Hl-1-K1+1
            pidx(l,j,x,y) = maxidx(Y(l-1,j,x:x+K1-1,y:y+K1-1))
            Y(l,j,x,y) = Y(l-1,j,pidx(l,j,x,y))
```

# Alternative to Max pooling: Mean Pooling



- Compute the mean of the pool, instead of the max

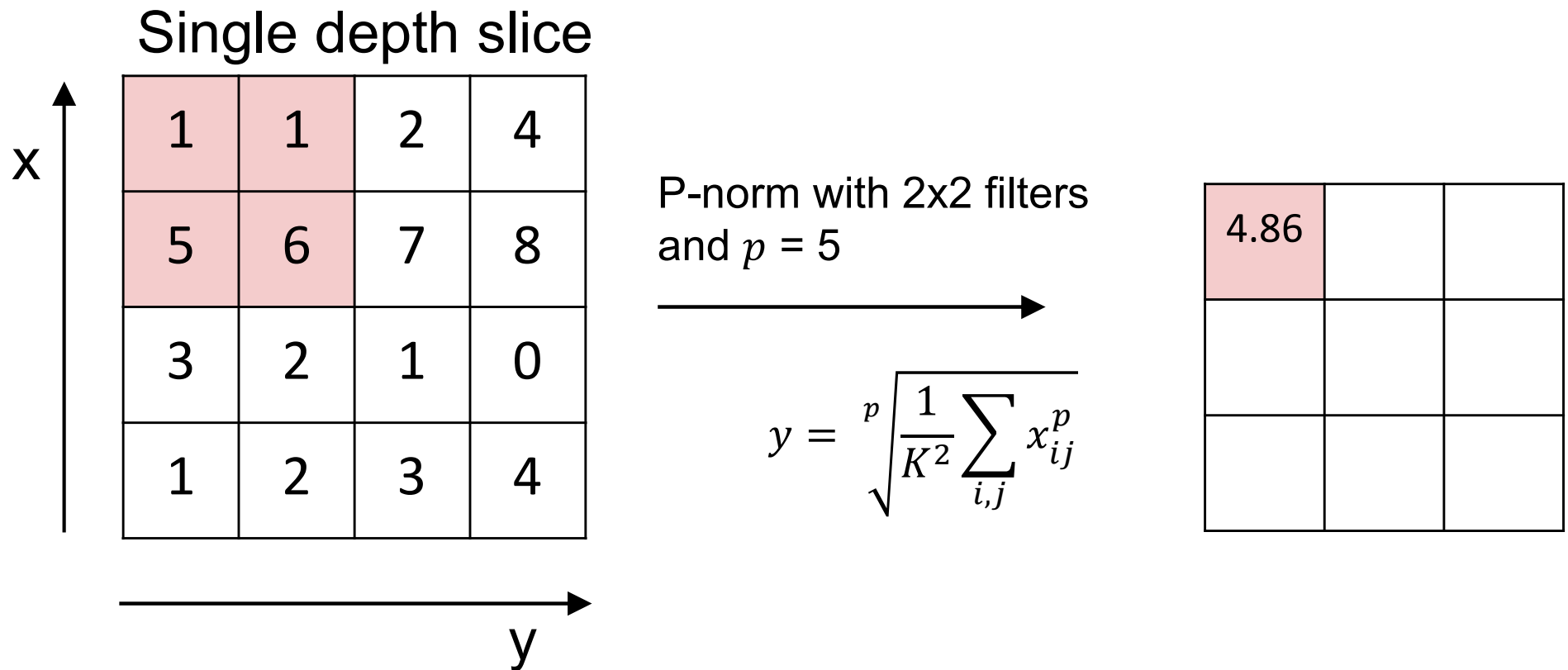
# Mean Pooling layer at layer $l$

Performed separately for every map ( $j$ ).

**Mean pooling**

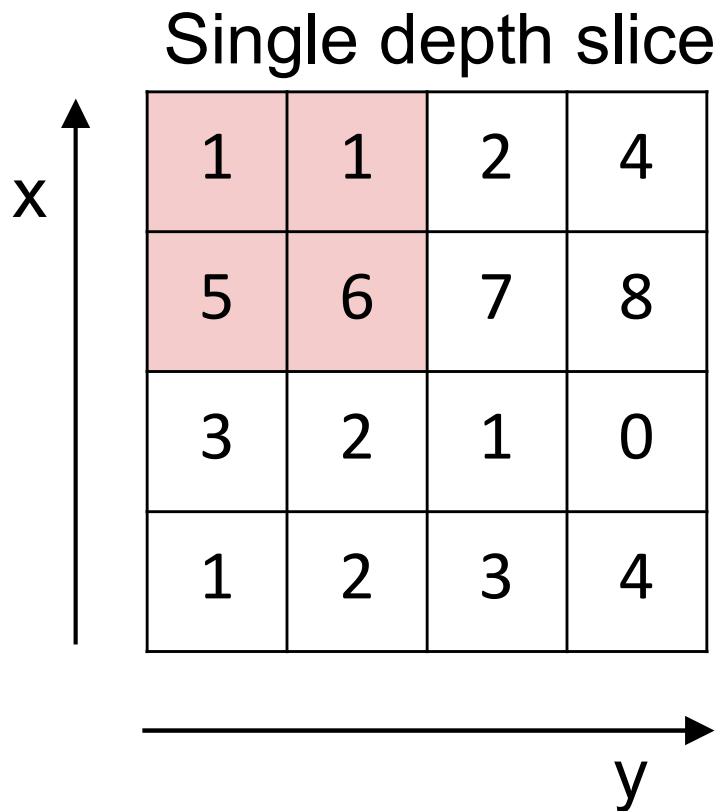
```
for j = 1:Dl
    for x = 1:Wl-1-Kl+1
        for y = 1:Hl-1-Kl+1
            Y(l,j,x,y) = mean(Y(l-1,j,x:x+Kl-1,y:y+Kl-1))
```

# Alternative to Max pooling: *p*-norm

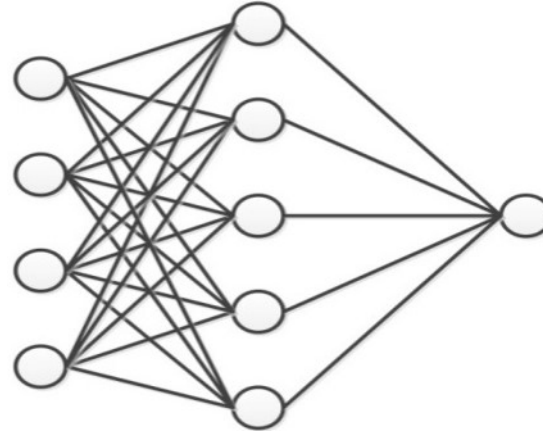


- Compute a  $p$ -norm of the pool

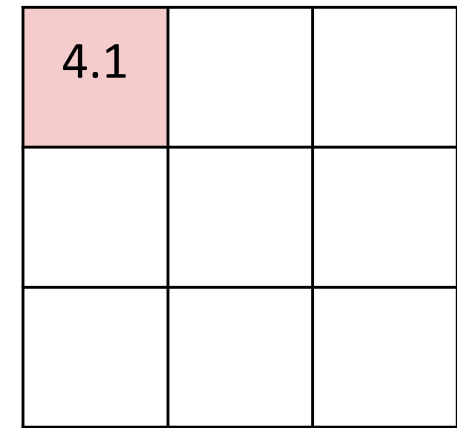
# Other options



Network applies to each  
2x2 block in this example



Network in network

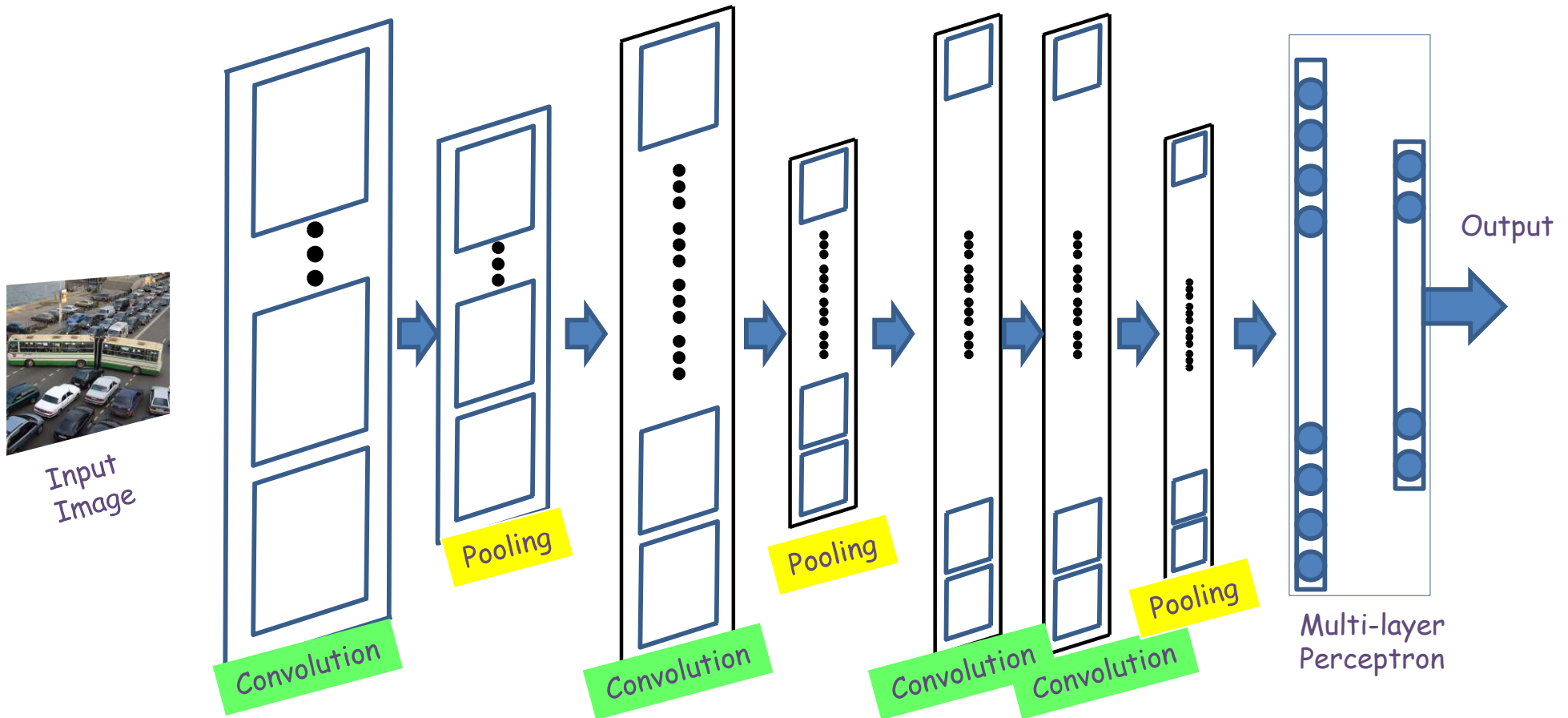


- The pooling may even be a *learned* filter
  - The *same* network is applied on each block
    - (Again, a shared parameter network)

# Pooling Summary

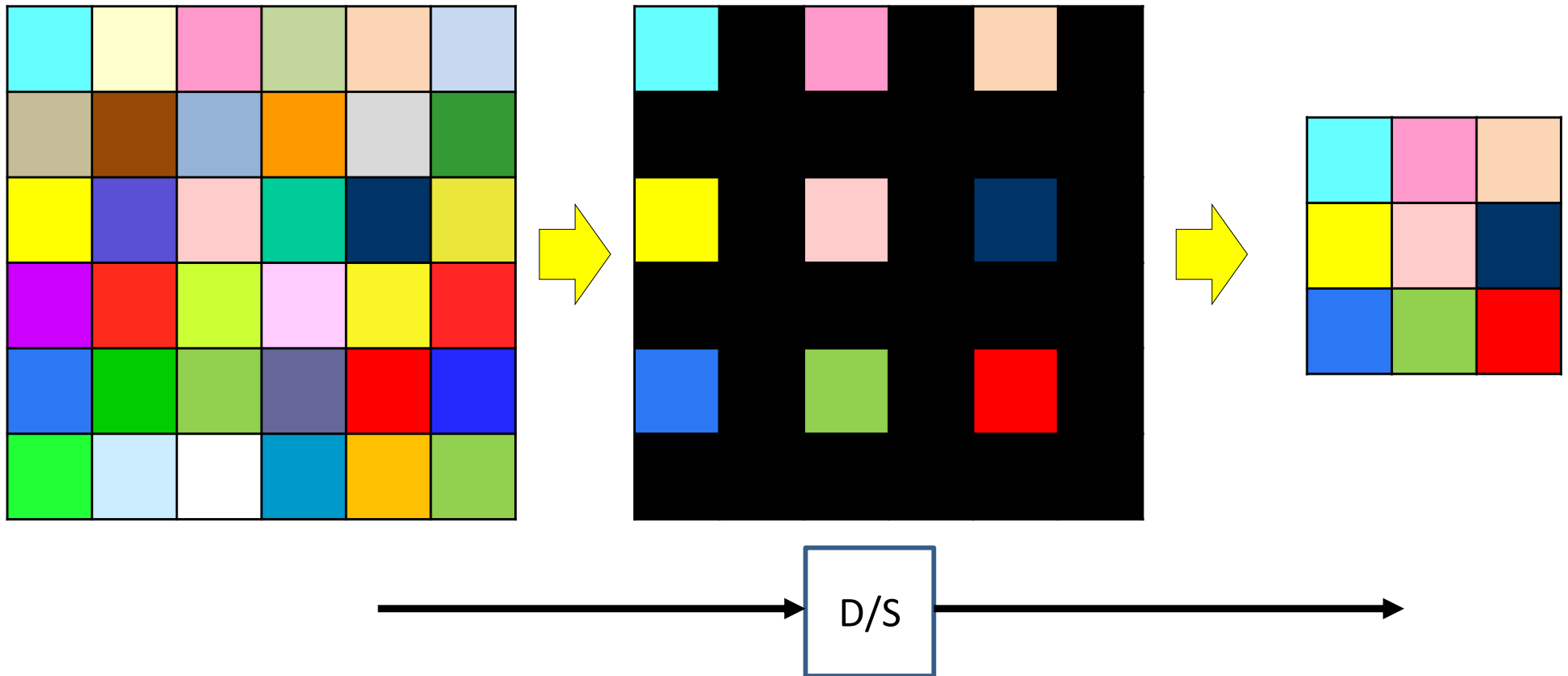
- Pooling layers “scan” the input using a “pooling” operation
  - E.g. selecting the max from a  $K \times K$  block of input
- Each “pooling filter” scans an *individual* maps in the previous layer to produce an output “pooled map”
  - As many output maps as input maps
- For pooling we do not generally pad the edges
  - The zeros may result in bogus pooled values, e.g. when all inputs are –ve and we apply max pooling

# The types of layers considered so far



- So far we have only considered layers where the output size is approximately equal to input size
- There are two other operations that *change* the size of the output

# The Downsampling Layer



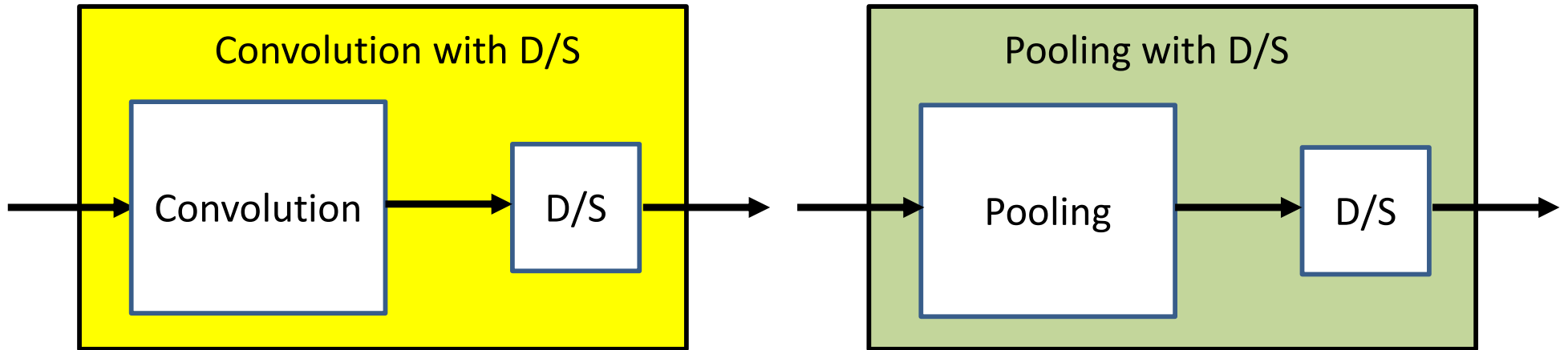
- A *downsampling* layer simply “drops”  $S - 1$  of  $S$  rows and columns for every map in the layer
  - Effectively reducing the size of the map by factor  $S$  in every direction



# Downsampling Pseudocode

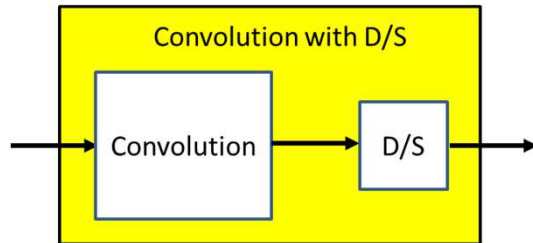
```
m = 1
for i = 1:S:W
    n = 1
    for j = 1:S:H
        y(m,n) = x(i,j)
        n++
    end
    m++
end
```

# Downsampling in practice



- In practice, the downsampling is combined with the layers just before it
  - Which could be convolutional or pooling layers

# Downsampling after Convolution



**bias**

0
---

**Filter**

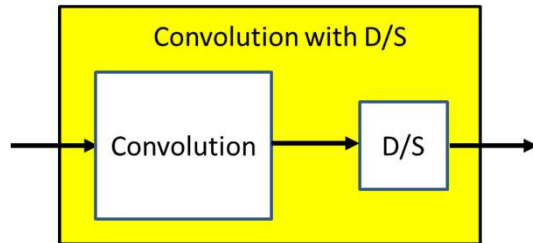
1	0	1
0	1	0
1	0	1

1 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	0	0
0 <sub>x0</sub>	1 <sub>x1</sub>	1 <sub>x0</sub>	1	0
0 <sub>x1</sub>	0 <sub>x0</sub>	1 <sub>x1</sub>	1	1
0	0	1	1	0
0	1	1	0	0

4	

- A downsampling layer can be combined with a convolutional layer into a single convolutional layer with convolution stride  $S$

# Downsampling after Convolution



**bias**

0
---

**Filter**

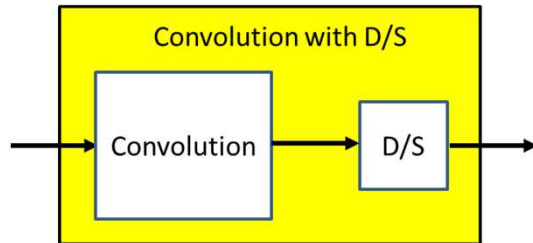
1	0	1
0	1	0
1	0	1

1	1	1 <sub>x1</sub>	0 <sub>x0</sub>	0 <sub>x1</sub>
0	1	1 <sub>x0</sub>	1 <sub>x1</sub>	0 <sub>x0</sub>
0	0	1 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>
0	0	1	1	0
0	1	1	0	0

4	4

- A downsampling layer can be combined with a convolutional layer into a single convolutional layer with convolution stride S

# Downsampling after Convolution



**bias**

0
---

**Filter**

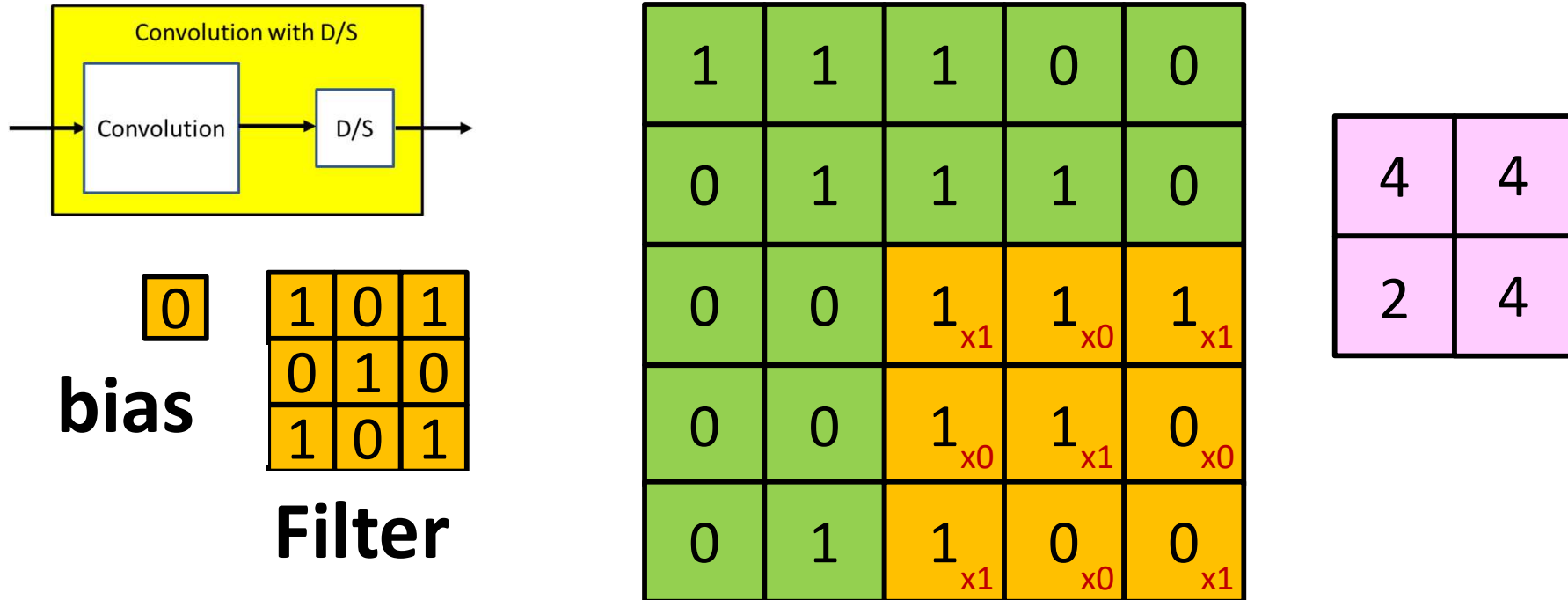
1	0	1
0	1	0
1	0	1

1	1	1	0	0
0	1	1	1	0
0 <sub>x1</sub>	0 <sub>x0</sub>	1 <sub>x1</sub>	1	1
0 <sub>x0</sub>	0 <sub>x1</sub>	1 <sub>x0</sub>	1	0
0 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	0	0

4	4
2	

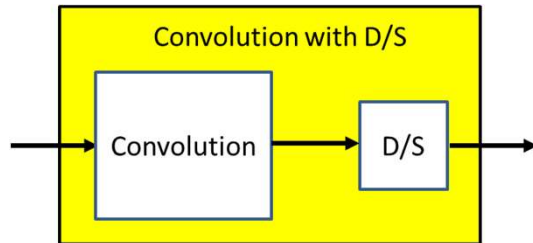
- A downsampling layer can be combined with a convolutional layer into a single convolutional layer with convolution stride S

# Downsampling after Convolution



- A downsampling layer can be combined with a convolutional layer into a single convolutional layer with convolution stride S

# Downsampling after Convolution



<b>bias</b>	0
-------------	---

1	0	1
0	1	0
1	0	1

**Filter**

1	1	1	0	0
0	1	1	1	0
0	0	1 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>
0	0	1 <sub>x0</sub>	1 <sub>x1</sub>	0 <sub>x0</sub>
0	1	1 <sub>x1</sub>	0 <sub>x0</sub>	0 <sub>x1</sub>

4	4
2	4

For an input of size  $N \times N$  and filters of size  $M \times M$  and stride  $S$ , the output size will be  $\left\lfloor \frac{N-M}{S} \right\rfloor + 1$  on every side

# Convolution with downsampling

The weight  $W(l, j)$  is now a 4D  $D_1 \times D_{l-1} \times K_1 \times K_1$  tensor

The product in blue is a tensor inner product with a scalar output

$Y(0) = \text{Image}$

for  $l = 1:L$  # layers operate on vector at  $(x, y)$

```
m = 1
```

```
for x = 1:S:Wl-1-K1+1
```

```
  n = 1
```

```
  for y = 1:S:Hl-1-K1+1
```

```
    segment = Y(l-1, :, x:x+K1-1, y:y+K1-1) #3D tensor
```

```
    z(l, :, m, n) = W(l).segment #tensor inner prod.
```

```
    Y(l, :, m, n) = activation(z(l, :, m, n))
```

```
    n++
```

```
  m++
```

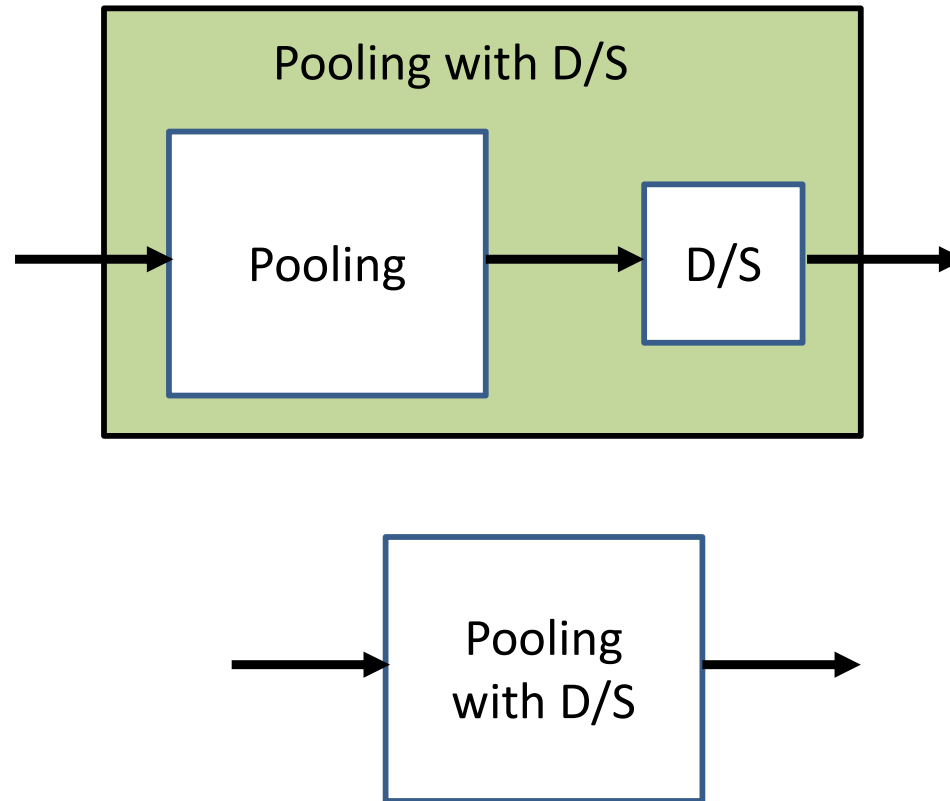
Downsampled indices

STRIDE

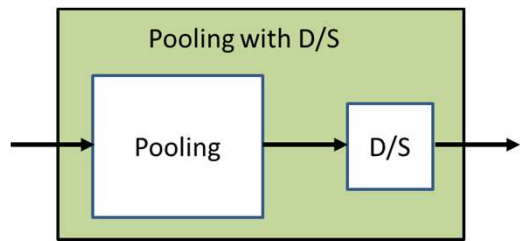
```
Y = softmax( {Y(L, :, :, :)} )
```



# Downsampling and Pooling



- Downsampling after a pooling layer can be merged with it to obtain pooling with stride  $S$ 
  - More on pooling later...



# Max Pooling

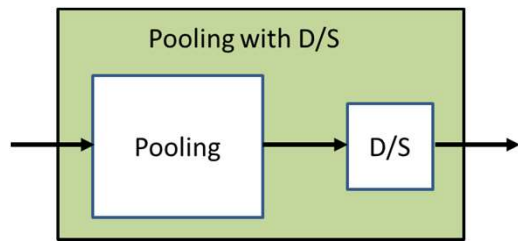
Single depth slice

x ↑	1	1	2	4
	5	6	7	8
	3	2	1	0
	1	2	3	4
→ y				

max pool with 2x2 filters  
and stride 2

6	

- Find the max in each block and stride by 2



# Max Pooling

Single depth slice

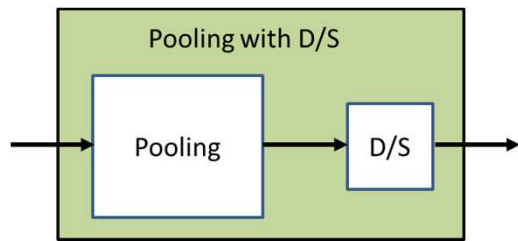
x ↑	1	1	2	4
	5	6	7	8
	3	2	1	0
	1	2	3	4
→ y				

max pool with 2x2 filters  
and stride 2

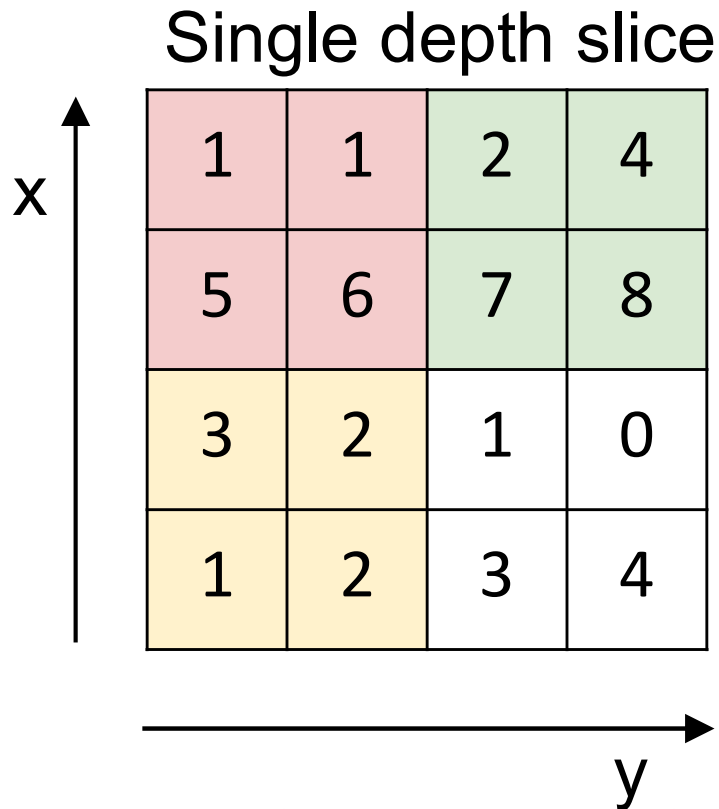


6	8

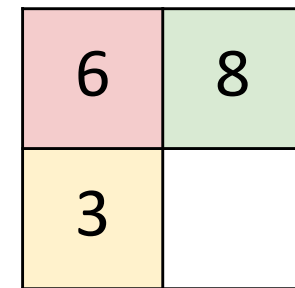
- Find the max in each block and stride by 2



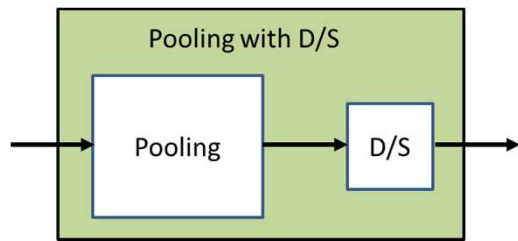
# Max Pooling



max pool with 2x2 filters  
and stride 2



- Find the max in each block and stride by 2



# Max Pooling

Single depth slice

x ↑	1	1	2	4
	5	6	7	8
	3	2	1	0
	1	2	3	4
→ y				

max pool with 2x2 filters  
and stride 2



6	8
3	4

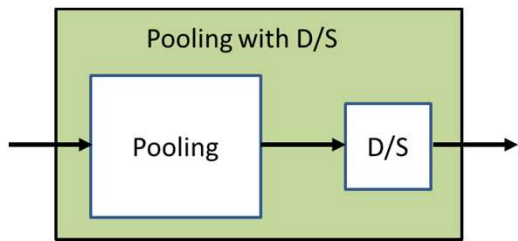
- Find the max in each block and stride by 2

# Max Pooling layer at layer $l$

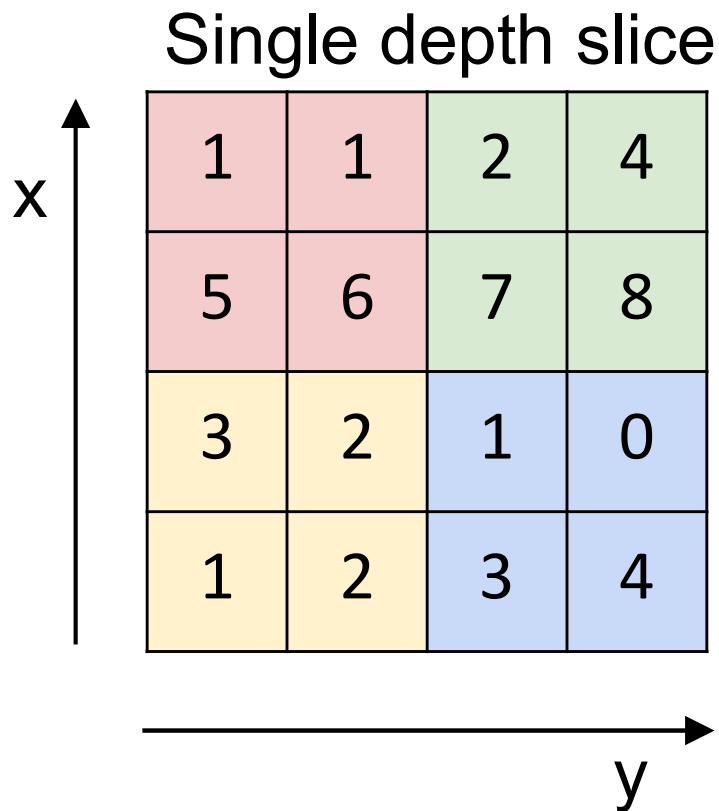
- a) Performed separately for every map ( $j$ ).
  - \*) Not combining multiple maps within a single max operation.
- b) Keeping track of location of max

## Max pooling

```
for j = 1:D1
    m = 1
    for x = 1:stride(1):W1-1-K1+1
        n = 1
        for y = 1:stride(1):H1-1-K1+1
            pidx(l,j,m,n) = maxidx(Y(l-1,j,x:x+K1-1,y:y+K1-1))
            Y(l,j,m,n) = Y(l-1,j,pidx(l,j,m,n))
            n = n+1
        end
        m = m+1
    end
end
```



# Mean Pooling



Mean pool with 2x2  
filters and stride 2



3.25	5.25
2	2

- Compute the mean of the pool, instead of the max


# Mean Pooling layer at layer $l$

Performed separately for every map ( $j$ ).

\*) Not combining multiple maps within a single operation.

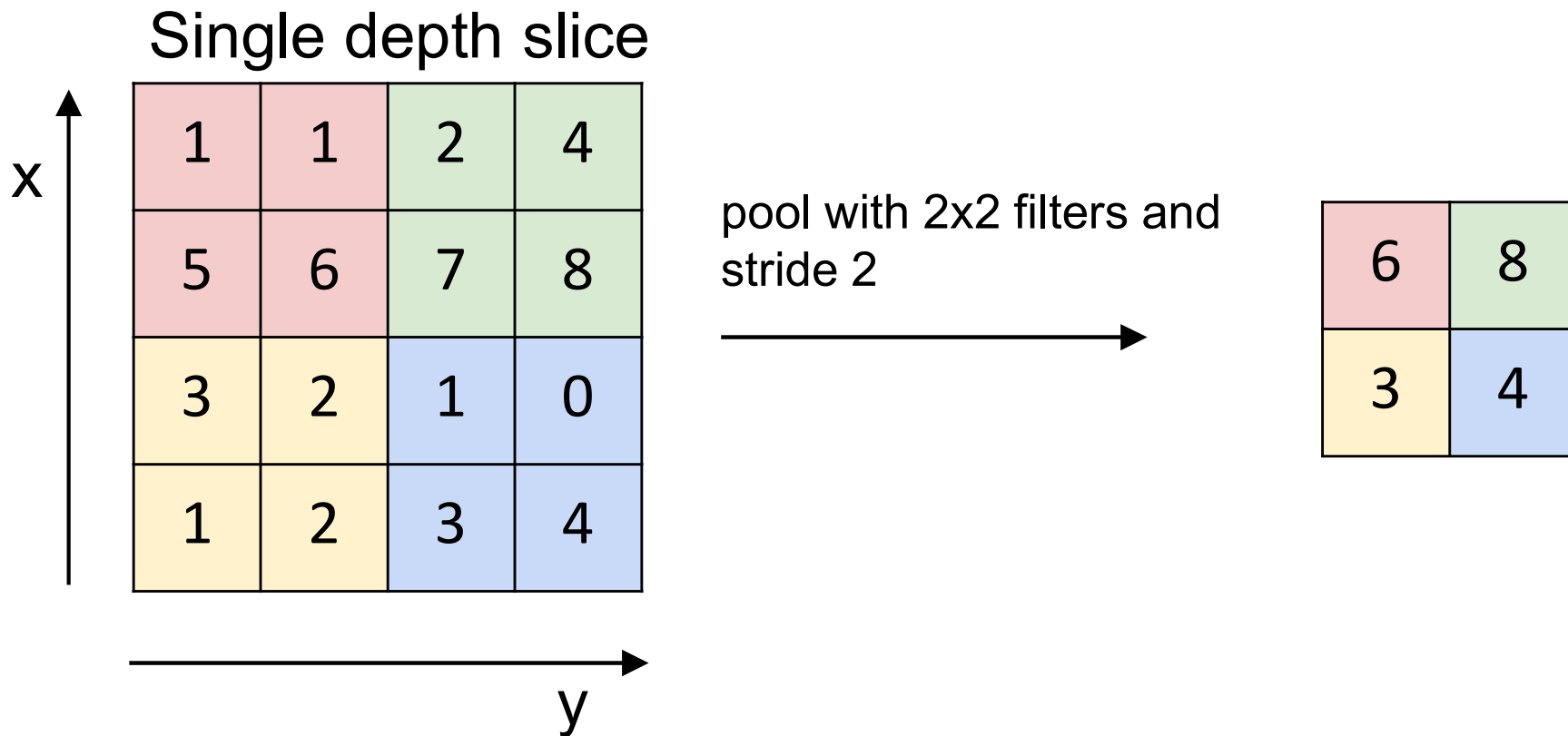
## Mean pooling

```
for j = 1:D1
    m = 1
    for x = 1:stride(1):Wl-1-K1+1
        n = 1
        for y = 1:stride(1):Hl-1-K1+1
            Y(l,j,m,n) = mean(Y(l-1,j,x:x+K1-1,y:y+K1-1))
            n = n+1
        end
        m = m+1
    end
end
```



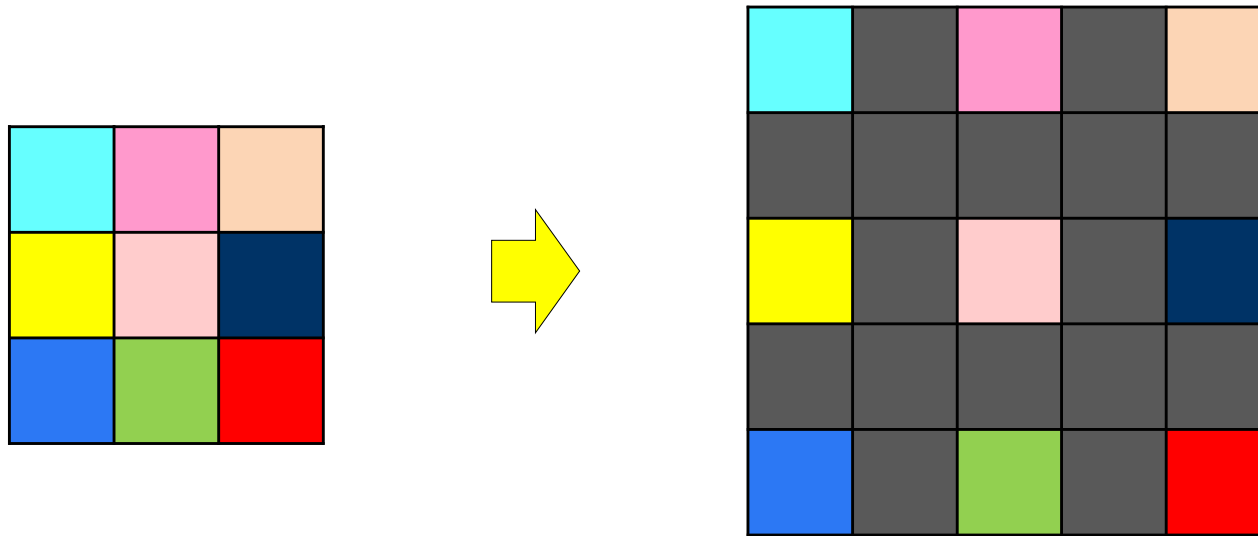


# Downsampling: Size of output



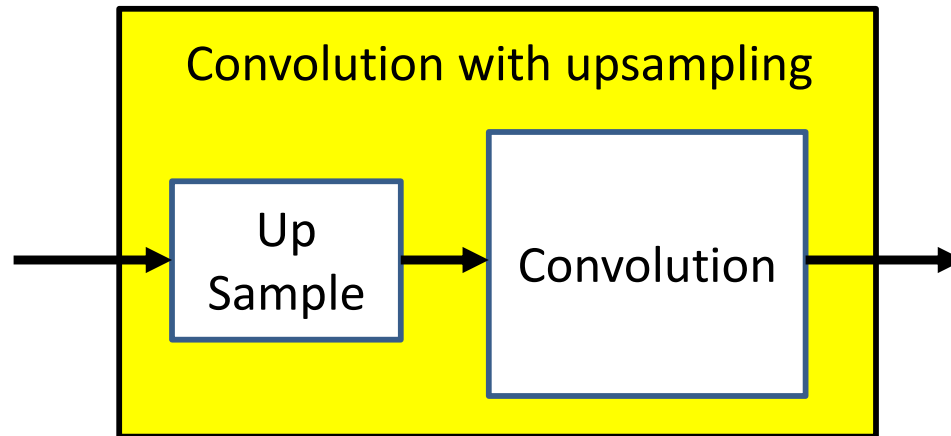
- An  $N \times N$  picture compressed by a  $P \times P$  pooling filter with stride  $D$  results in an output map of side  $\lceil (N - P)/D \rceil + 1$ 
  - Typically do not zero pad

# The Upsampling Layer



- A *upsampling* (or dilation) layer simply introduces  $S - 1$  rows and columns for every map in the layer
  - Effectively *increasing* the size of the map by factor  $S$  in every direction
- Used explicitly to increase the map size by a uniform factor

# The Upsampling Layer



- A *upsampling* layer is generally followed by a CNN layer
  - It is not useful to follow it by a pooling layer
  - It is also not useful as the *final* layer of a CNN

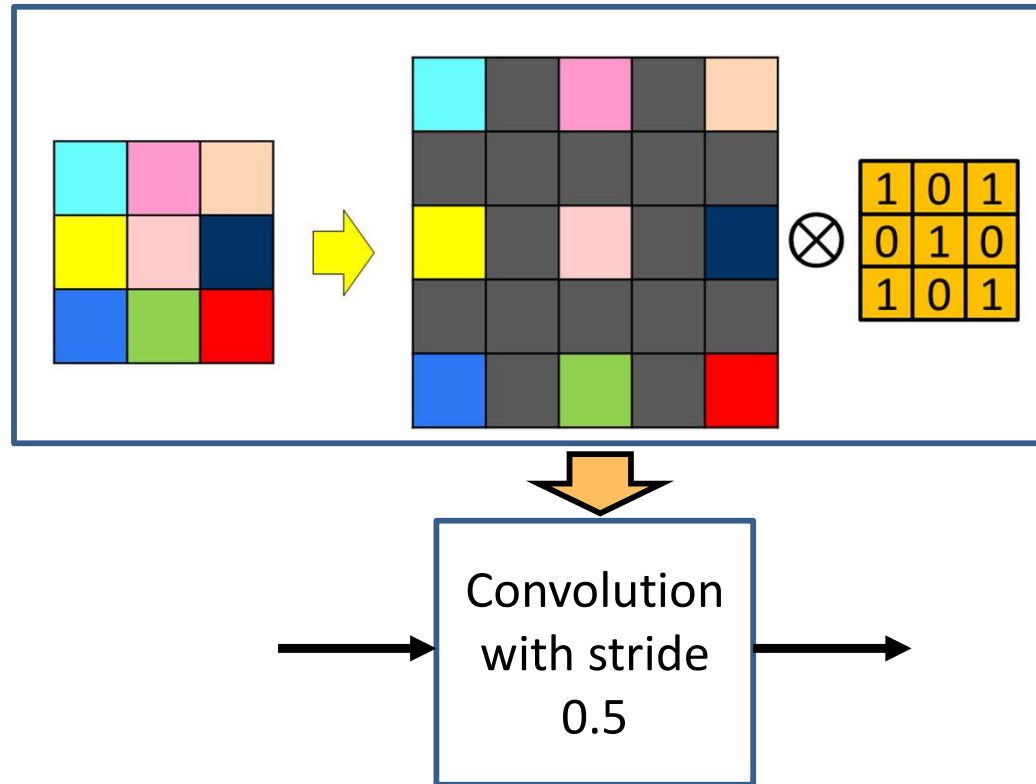
# Upsampling

**X** is a 3D  $D_1 \times W \times H$  tensor

Assuming  $D_1$ ,  $W$  and  $H$  are also passed in

```
function dilate(X)
    Xup = zeros(D1, (W-1)S+1, (H-1)S+1)
    for i = 1:W
        for j = 1:H
            Xup(:, (i-1)S+1, (j-1)S+1) = X(:, i, j)
        end
    end
    return Xup
```

# The Upsampling Layer



- Upsampling layers followed by a convolutional layer are also often viewed as convolving with a fractional stride
  - Upsampling by factor  $S$  is the same as striding by factor  $1/S$

# Resampling Summary

- Map sizes can be changed by downsampling or upsampling
  - Downsampling: Drop  $S-1$  of  $S$  rows and columns
  - Upsampling: Insert  $S-1$  zeros between every two rows / columns
- Downsampling typically *follows* convolution or pooling
  - Reduces the size of the maps
- Upsampling occurs *before* convolution
  - Increases the size of the map
  - Does not generally occur before pooling layers

# Story so far

- The convolutional neural network is a supervised version of a computational model of mammalian vision
- It includes
  - Convolutional layers comprising learned filters that scan the outputs of the previous layer
  - “Pooling” layers that vote over groups of outputs from the convolutional layer
- Convolution can change the size of the output. This may be controlled via zero padding.
- Pooling layers may perform max, p-norms, or be learned
- Resampling layers increase or decrease the size of the map
  - Downsampling can be merged with the preceding operation, by simply scanning with a stride  $> 1$
  - Upsampling must occur before convolution operations

# Poll 3

@676, @677

Mark all that are true of upsampling layers

- Upsampling by  $S$  introduces  $S-1$  zeroes between adjacent rows and columns
- Upsampling layers are generally followed by pooling layers
- Upsampling layers are generally followed by convolutional layers
- Upsampling layers can be combined with the convolutional or pooling layers before them into a single operation with fractional stride

Mark all that are true of downsampling layers

- Downsampling by  $S$  replaces every  $S$ th row and column with zeros
- Downsampling by  $S$  deletes  $S-1$  consecutive rows/columns and retains only every  $S$ th row/column
- Downsampling layers are usually merged with the convolution or pooling layers after them, which are then modified to convolution or pooling with a stride
- Downsampling layers are usually merged with the convolution or pooling layers before them, which are then modified to convolution or pooling with a stride



# Poll 3

Mark all that are true of upsampling layers

- **Upsampling by  $S$  introduces  $S-1$  zeroes between adjacent rows and columns**
- Upsampling layers are generally followed by pooling layers
- **Upsampling layers are generally followed by convolutional layers**
- Upsampling layers can be combined with the convolutional or pooling layers before them into a single operation with fractional stride

Mark all that are true of downsampling layers

- Downsampling by  $S$  replaces every  $S$ th row and column with zeros
- **Downsampling by  $S$  deletes  $S-1$  consecutive rows/columns and retains only every  $S$ th row/column**
- Downsampling layers are usually merged with the convolution or pooling layers after them, which are then modified to convolution or pooling with a stride
- **Downsampling layers are usually merged with the convolution or pooling layers before them, which are then modified to convolution or pooling with a stride**

# Setting everything together

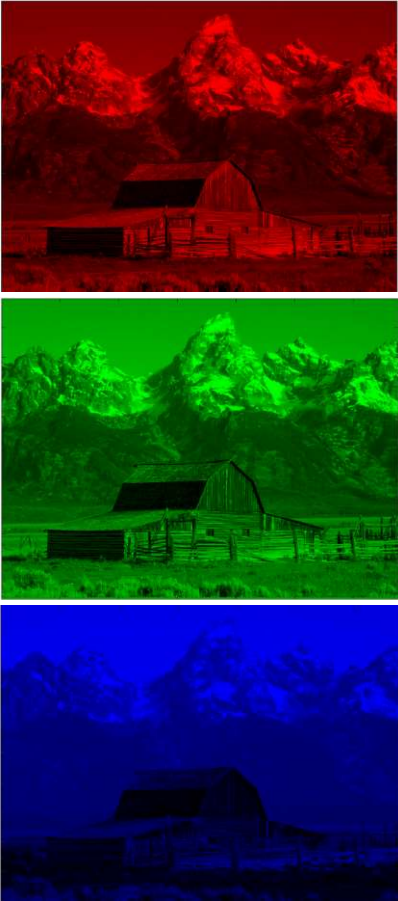
- Typical image classification task
  - Assuming maxpooling..

# Convolutional Neural Networks



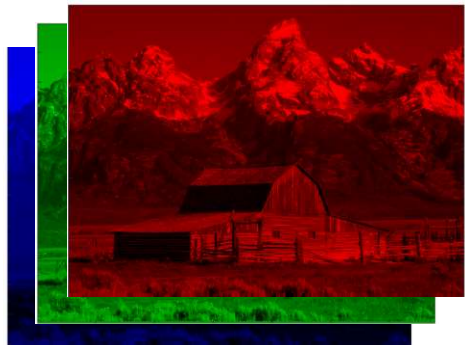
- Input: 1 or 3 images
  - Grey scale or color
  - Will assume color to be generic

# Convolutional Neural Networks



- Input: 3 pictures

# Convolutional Neural Networks

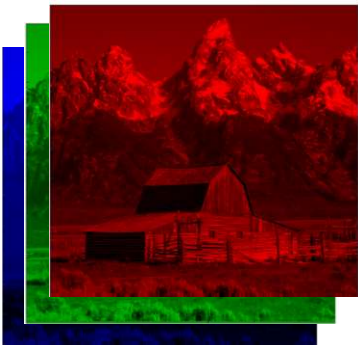


- Input: 3 pictures

# Preprocessing

- Large images are a problem
  - Too large
  - Compute and memory intensive to process
- Sometimes scaled to smaller sizes, e.g. 128x128 or even 32x32
  - Based on how much will fit on your GPU
  - Typically cropped to *square* images
  - Filters are also typically square

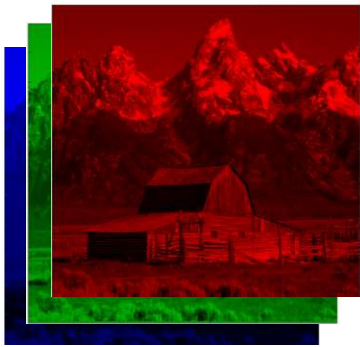
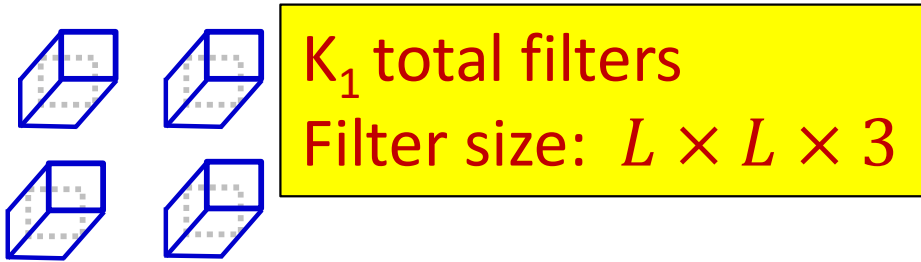
# Convolutional Neural Networks



$I \times I$  image

- Input: 3 pictures

# Convolutional Neural Networks

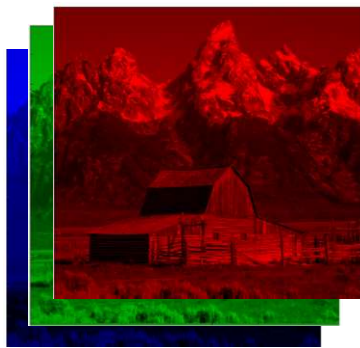
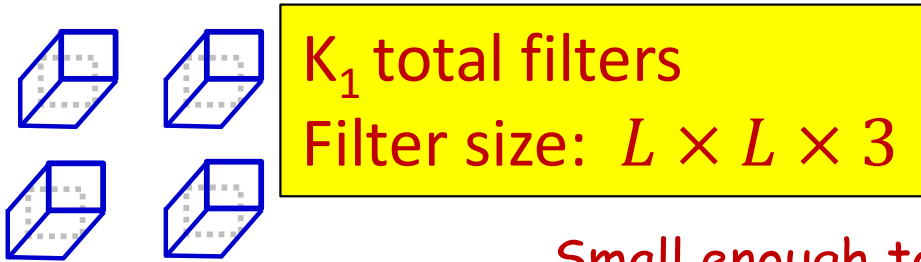


$I \times I$  image

- Input is convolved with a set of  $K_1$  filters
  - Typically  $K_1$  is a power of 2, e.g. 2, 4, 8, 16, 32,...
  - Filters are typically 5x5, 3x3, or even 1x1



# Convolutional Neural Networks

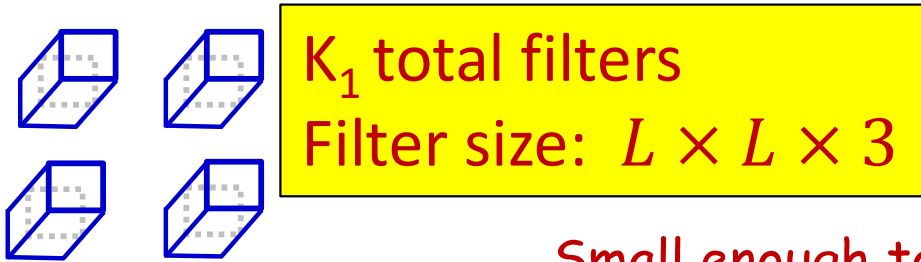


$I \times I$  image

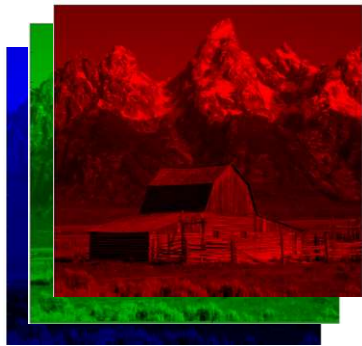
Small enough to capture fine features  
(particularly important for scaled-down images)

- Input is convolved with a set of  $K_1$  filters
  - Typically  $K_1$  is a power of 2, e.g. 2, 4, 8, 16, 32,...
  - Filters are typically 5x5, 3x3, or even 1x1

# Convolutional Neural Networks



Small enough to capture fine features  
(particularly important for scaled-down images)

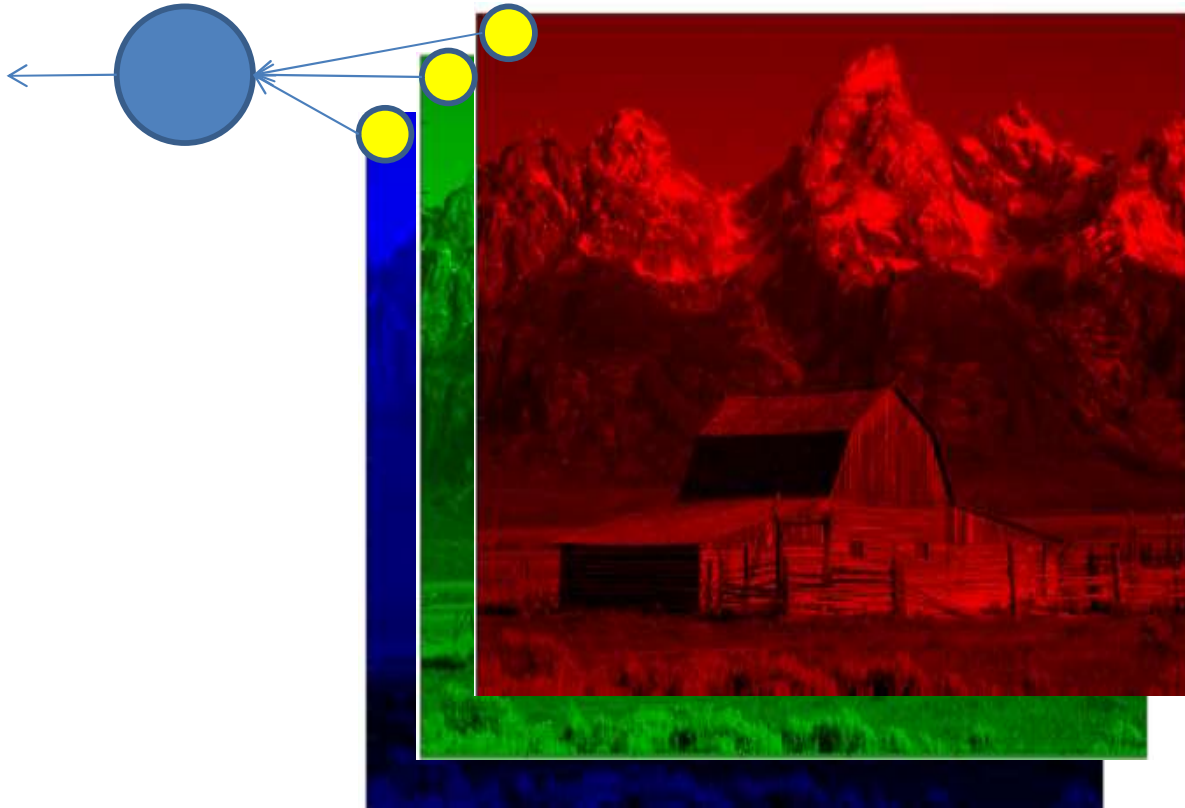


$I \times I$  image

What on earth is this?

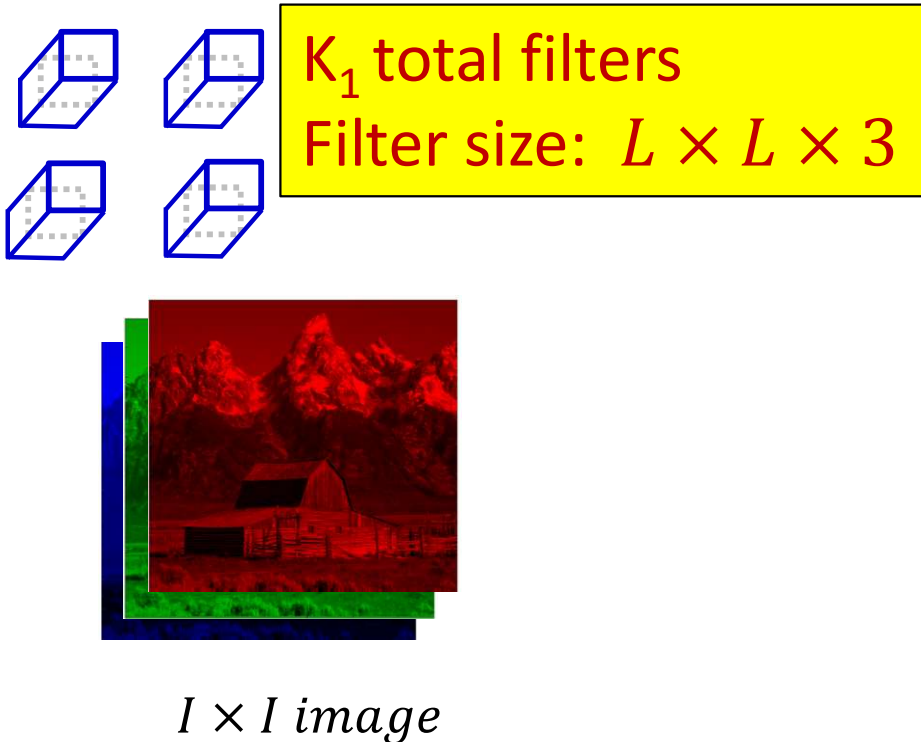
- Input is convolved with a set of  $K_1$  filters
  - Typically  $K_1$  is a power of 2, e.g. 2, 4, 8, 16, 32,...
  - Filters are typically 5x5, 3x3, or even 1x1

# The 1x1 filter



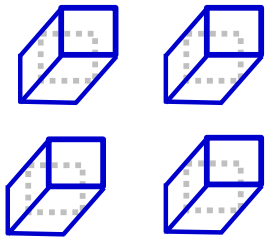
- A 1x1 filter is simply a perceptron that operates over the *depth* of the stack of maps, but has no spatial extent
  - Takes one pixel from each of the maps (at a given location) as input
  - A *non-distributed* layer of the scanning MLP

# Convolutional Neural Networks

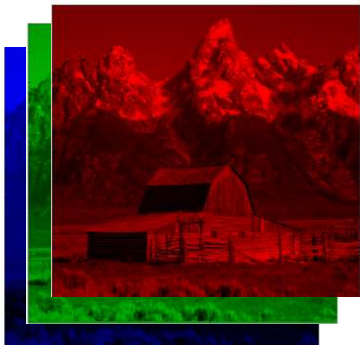


- Input is convolved with a set of  $K_1$  filters
  - Typically  $K_1$  is a power of 2, e.g. 2, 4, 8, 16, 32,...
  - **Better notation:** Filters are typically 5x5(x3), 3x3(x3), or even 1x1(x3)

# Convolutional Neural Networks



$K_1$  total filters  
Filter size:  $L \times L \times 3$



$I \times I$  image

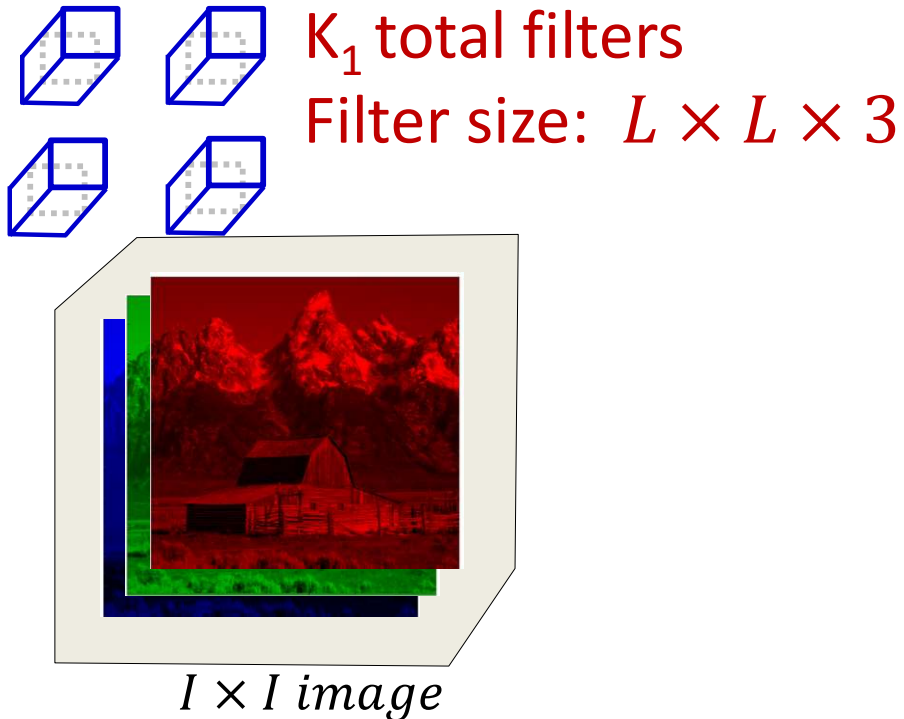
Parameters to choose:  $K_1$ ,  $L$  and  $S$

1. Number of filters  $K_1$
2. Size of filters  $L \times L \times 3 + \text{bias}$
3. Stride of convolution  $S$

Total number of parameters:  $K_1(3L^2 + 1)$

- Input is convolved with a set of  $K_1$  filters
  - Typically  $K_1$  is a power of 2, e.g. 2, 4, 8, 16, 32,...
  - **Better notation:** Filters are typically 5x5(x3), 3x3(x3), or even 1x1(x3)
  - **Typical stride:** 1 or 2

# Convolutional Neural Networks

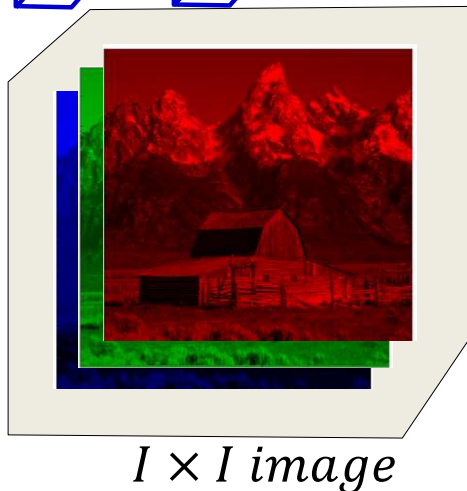
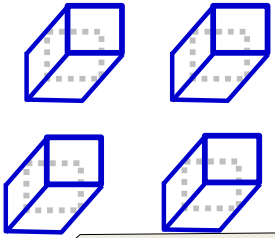


- The input may be zero-padded according to the size of the chosen filters

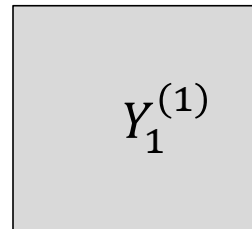
# Convolutional Neural Networks

$K_1$  filters of size:

$$L \times L \times 3$$



$$I \times I$$



The layer includes a convolution operation followed by an activation (typically RELU)

$$z_m^{(1)}(i, j) = \sum_{c \in \{R, G, B\}} \sum_{k=1}^L \sum_{l=1}^L w_m^{(1)}(c, k, l) I_c(i + k, j + l) + b_m^{(1)}$$

$$Y_m^{(1)}(i, j) = f\left(z_m^{(1)}(i, j)\right)$$

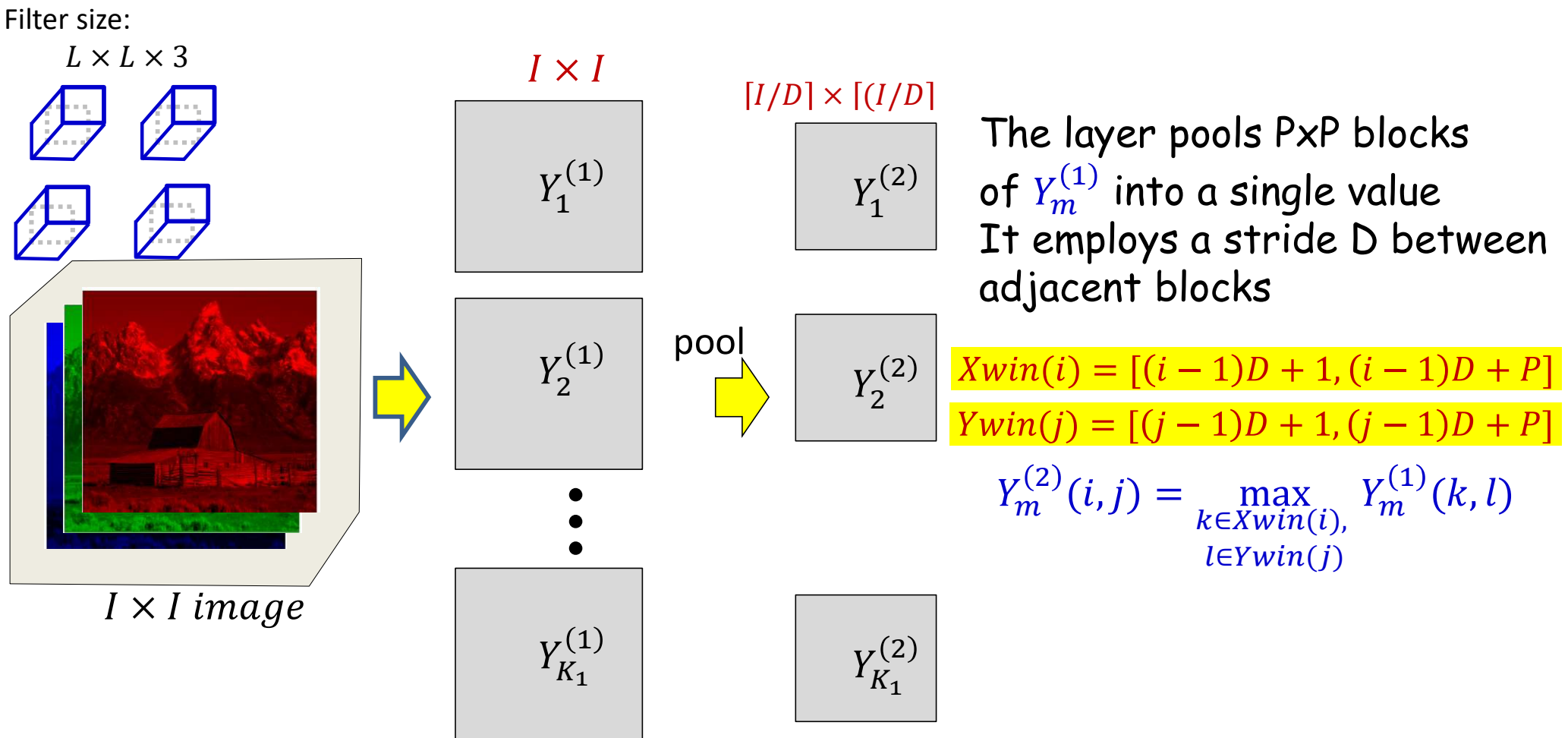
- **First convolutional layer:** Several convolutional filters
  - Filters are “3-D” (third dimension is color)
  - Convolution followed typically by a RELU activation
- Each filter creates a single 2-D output map

# Learnable parameters in the first convolutional layer

- The first convolutional layer comprises  $K_1$  filters, each of size  $L \times L \times 3$ 
  - Spatial span:  $L \times L$
  - Depth : 3 (3 colors)
- This represents a total of  $K_1(3L^2 + 1)$  parameters
  - “+ 1” because each filter also has a bias
- All of these parameters must be learned

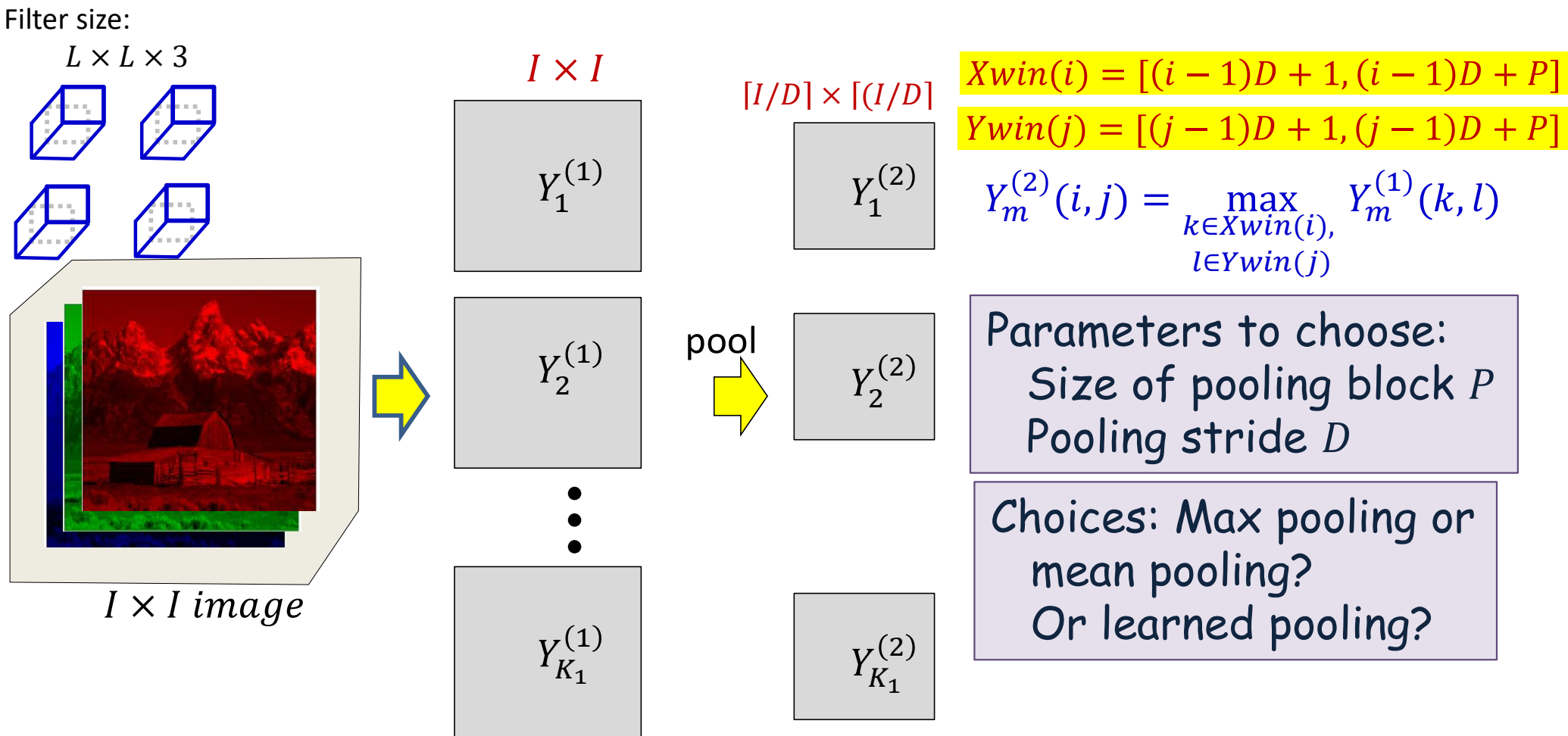


# Convolutional Neural Networks



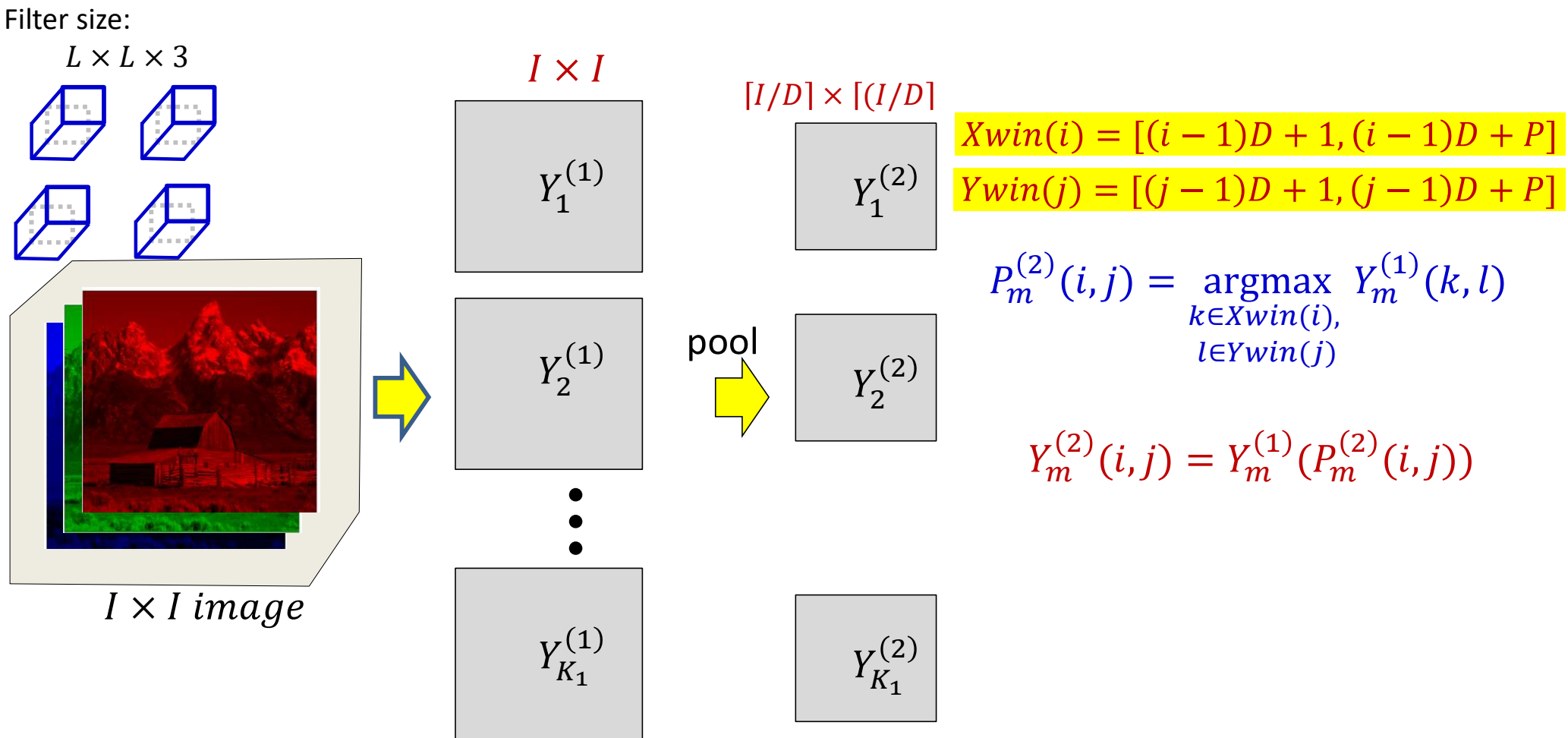
- **First pooling/downsampling layer:** From each  $P \times P$  block of each map, *pool* down to a single value
  - For max pooling, during training keep track of which position had the highest value

# Convolutional Neural Networks



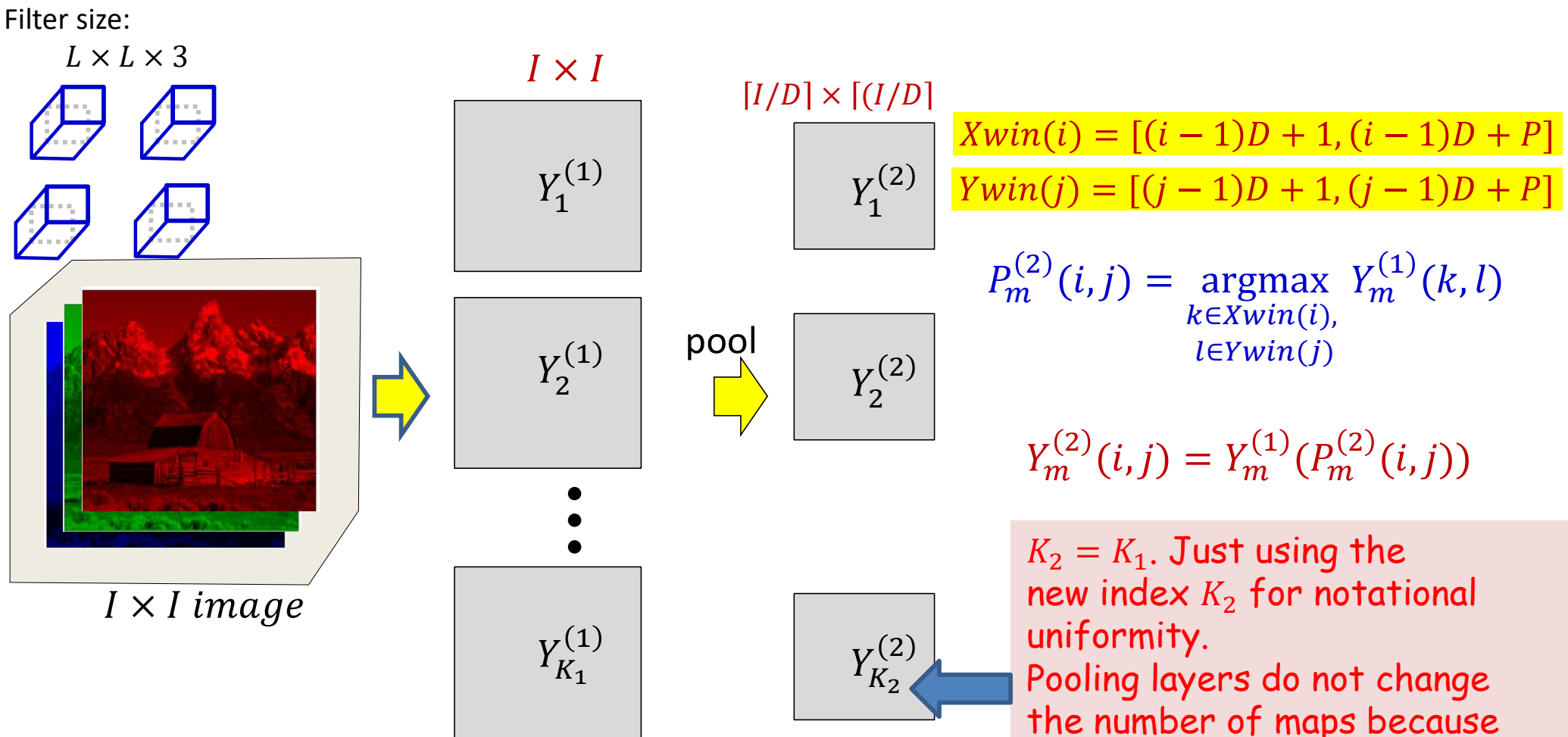
- **First pooling/downsampling layer:** From each  $P \times P$  block of each map, *pool* down to a single value
  - For max pooling, during training keep track of which position had the highest value

# Convolutional Neural Networks



- **First pooling/downsampling layer:** From each  $P \times P$  block of each map, *pool* down to a single value
  - **For max pooling, during training keep track of which position had the highest value**

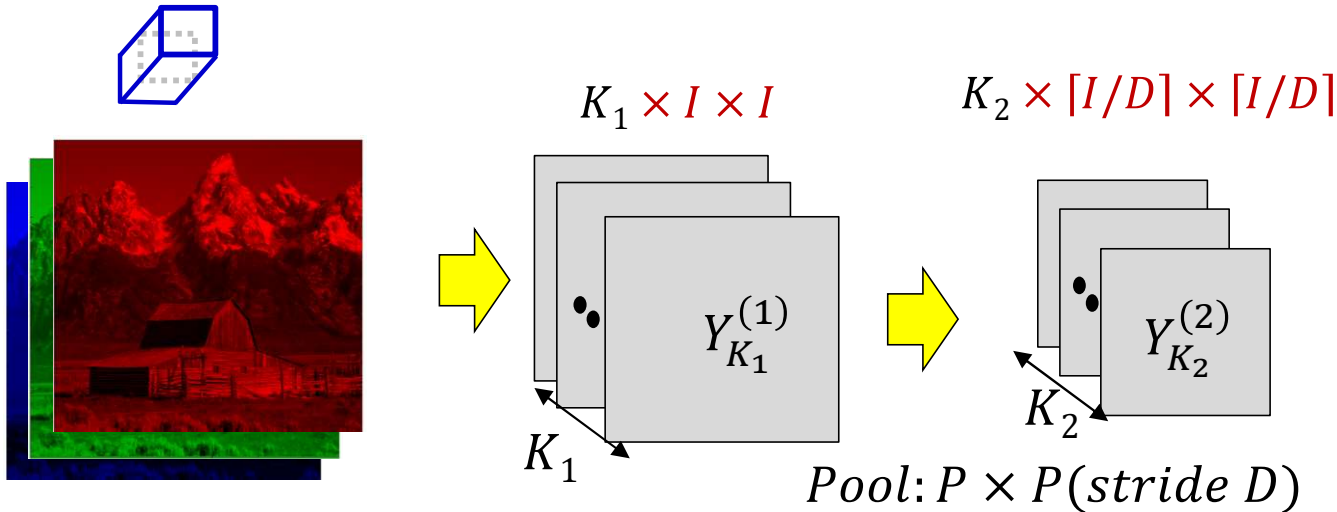
# Convolutional Neural Networks



- **First pooling/downsampling layer:** From each map, *pool* down to a single value
  - For max pooling, during training keep track of which position had the highest value

# Convolutional Neural Networks

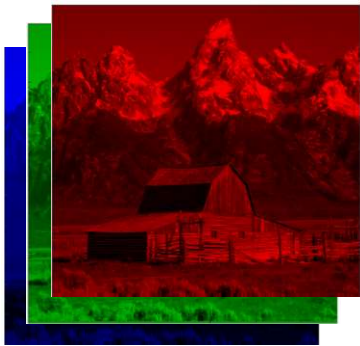
$$W_m: 3 \times L \times L$$
$$m = 1 \dots K_1$$



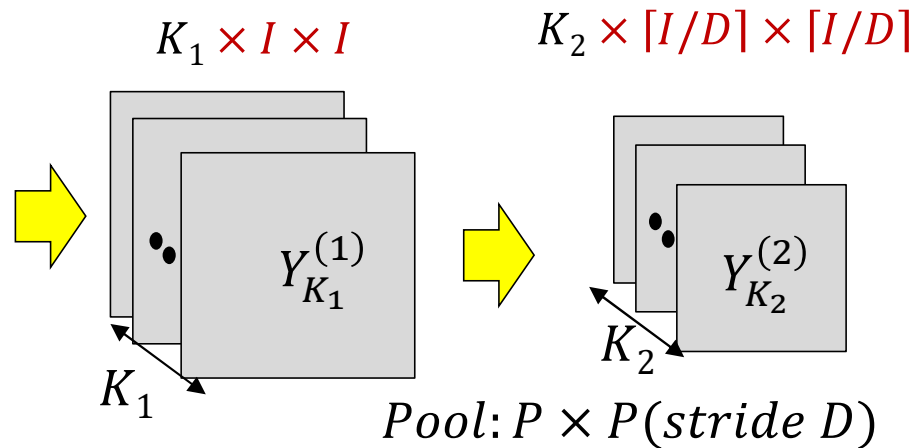
- **First pooling layer:** Drawing it differently for convenience

# Convolutional Neural Networks

$$W_m: 3 \times L \times L$$
$$m = 1 \dots K_1$$



Jargon: Filters are often called "Kernels"  
The outputs of individual filters are called "channels"  
The number of filters ( $K_1$ ,  $K_2$ , etc) is the number of channels

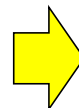
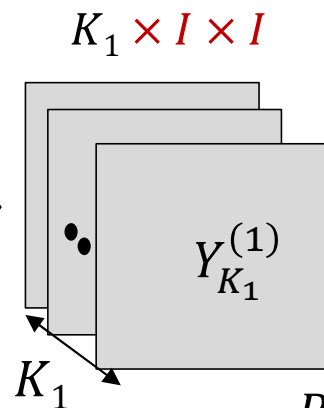
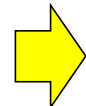
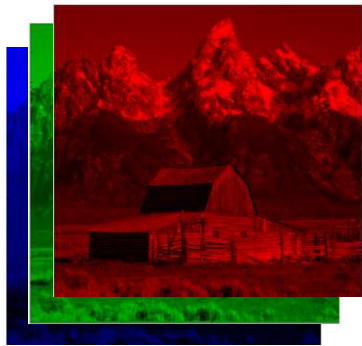


- **First pooling layer:** Drawing it differently for convenience

# Convolutional Neural Networks

$$W_m: 3 \times L \times L$$

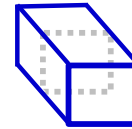
$$m = 1 \dots K_1$$



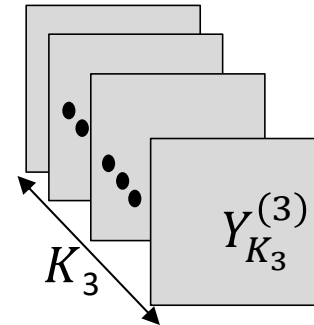
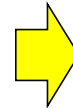
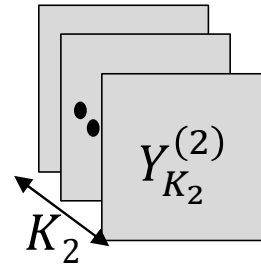
*Pool:  $P \times P$  (stride  $D$ )*

$$W_m: K_2 \times L_3 \times L_3$$

$$m = 1 \dots K_3$$



$$K_2 \times [I/D] \times [I/D]$$



$$z_m^{(n)}(i, j) = \sum_{r=1}^{K_{n-1}} \sum_{k=1}^{L_n} \sum_{l=1}^{L_n} w_m^{(n)}(r, k, l) Y_r^{(n-1)}(i + k, j + l) + b_m^{(n)}$$

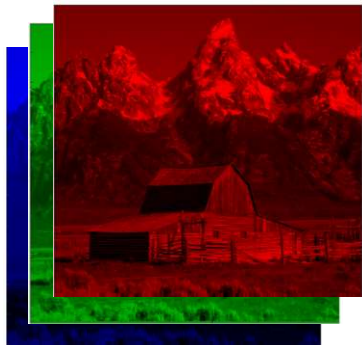
$$Y_m^{(n)}(i, j) = f(z_m^{(n)}(i, j))$$

- **Second convolutional layer:**  $K_3$  3-D filters resulting in  $K_3$  2-D maps
  - Alternately, a kernel with  $K_3$  output channels

# Convolutional Neural Networks

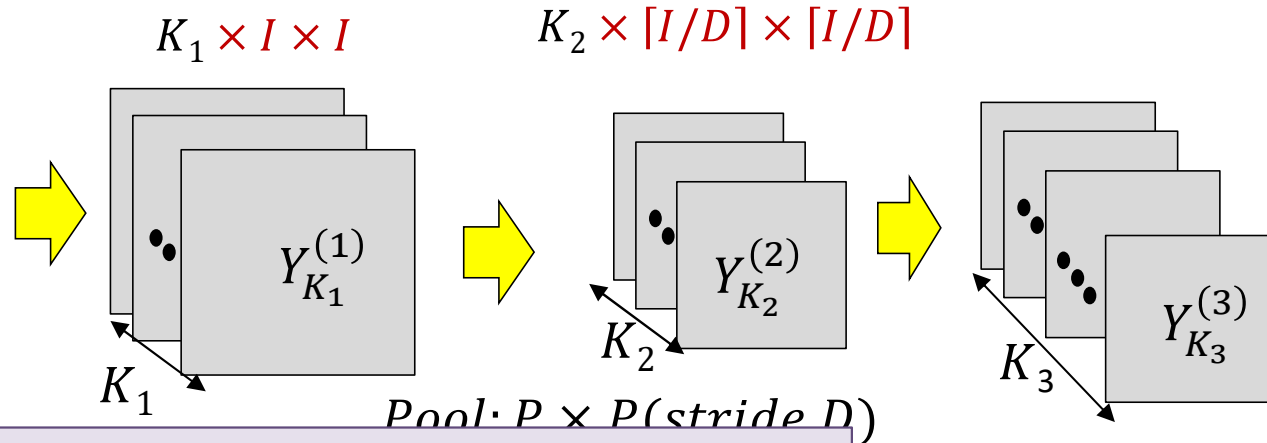
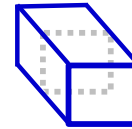
$$W_m: 3 \times L \times L$$

$$m = 1 \dots K_1$$



$$W_m: K_2 \times L_3 \times L_3$$

$$m = 1 \dots K_3$$



Parameters to choose:  $K_3$ ,  $L_3$  and  $S_3$

1. Number of filters  $K_3$
2. Size of filters  $L_3 \times L_3 \times K_2 + bias$
3. Stride of convolution  $S_3$

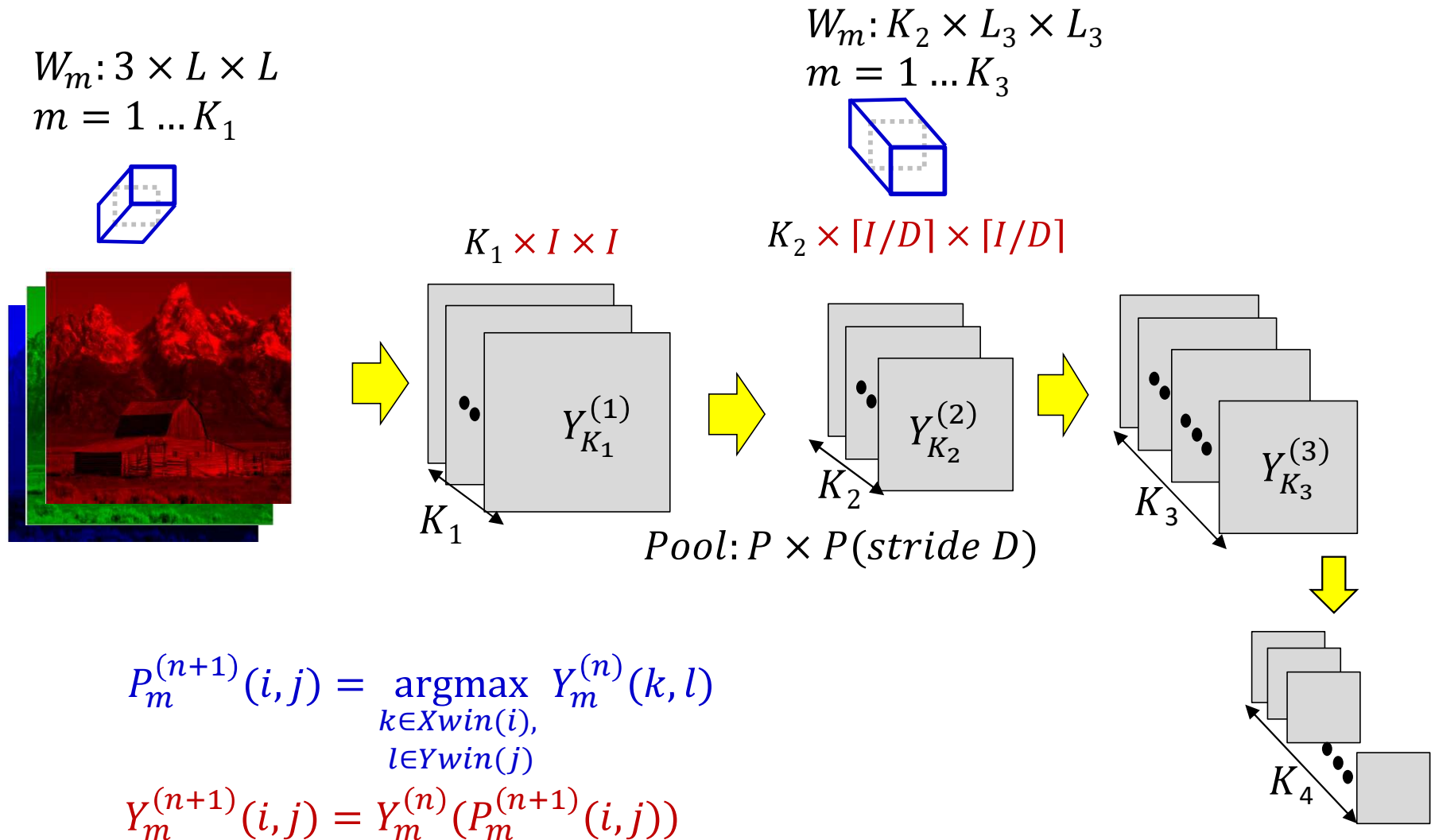
$(n)$   
 $m$

Total number of parameters:  $K_3(K_2L_3^2 + 1)$   
All these parameters must be learned

ing in  $K_3$  2-D maps

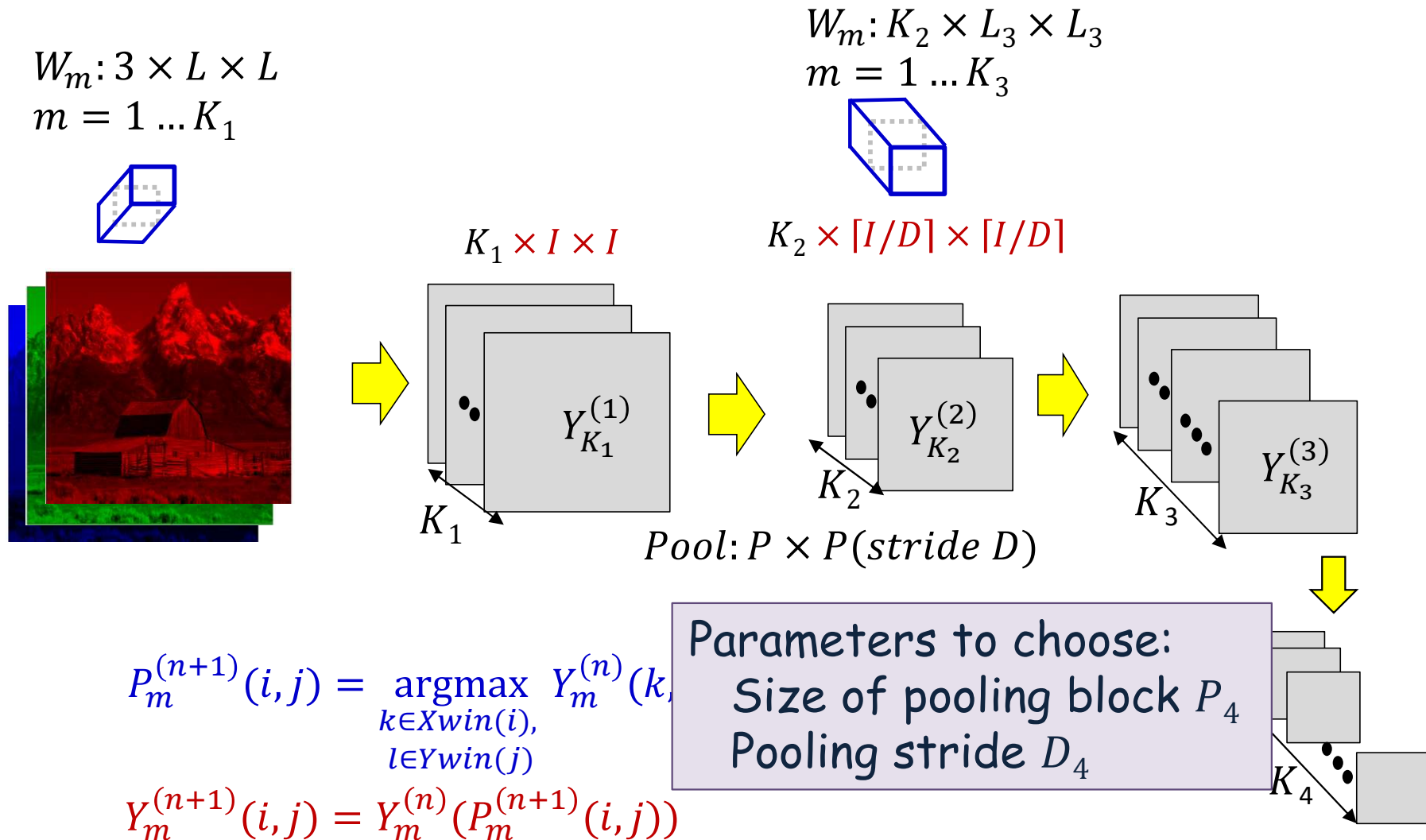


# Convolutional Neural Networks



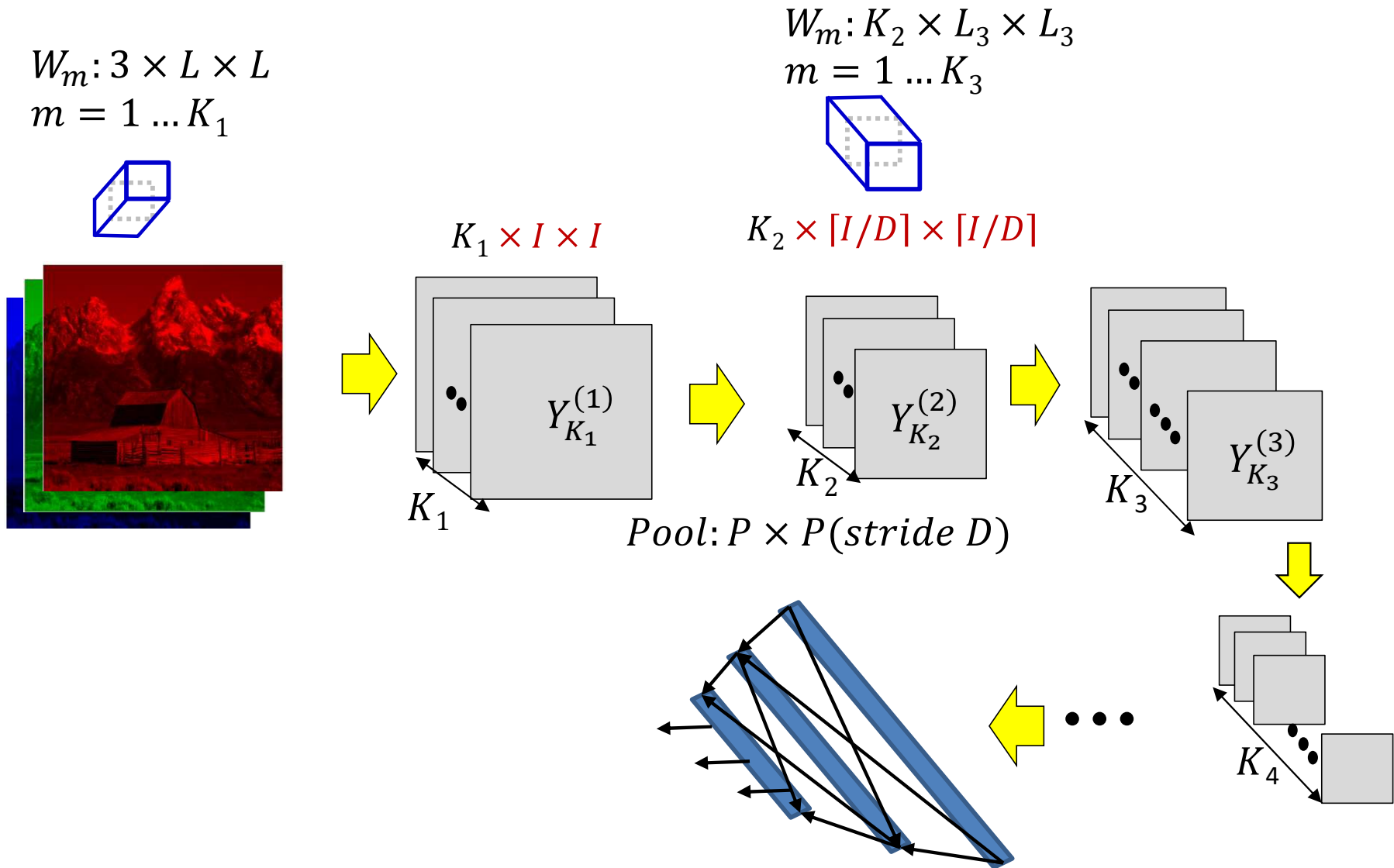
- **Second convolutional layer:**  $K_2$  3-D filters resulting in  $K_2$  2-D maps
- **Second pooling layer:**  $K_2$  Pooling operations: outcome  $K_2$  reduced 2D maps

# Convolutional Neural Networks



- **Second convolutional layer:**  $K_2$  3-D filters resulting in  $K_2$  2-D maps
- **Second pooling layer:**  $K_2$  Pooling operations: outcome  $K_2$  reduced 2D maps

# Convolutional Neural Networks



- This continues for several layers until the final convolved output is fed to a softmax
  - Or a full MLP

# The Size of the Layers

- Each convolution layer with stride 1 typically maintains the size of the image
  - With appropriate zero padding
  - If performed *without* zero padding it will decrease the size of the input
- Each convolution layer will generally *increase* the *number* of maps from the previous layer
  - Increasing layers reduces the amount of information lost by subsequent downsampling
- Each pooling layer with stride  $D$  *decreases* the *size* of the maps by a factor of  $D$
- Filters within a layer must all be the same size, but sizes may vary with layer
  - Similarly for pooling,  $D$  may vary with layer
- In general the number of convolutional filters increases with layers

# Parameters to choose (design choices)

- Number of convolutional and downsampling layers
  - And arrangement (order in which they follow one another)
- For each convolution layer:
  - Number of filters  $K_i$
  - Spatial extent of filter  $L_i \times L_i$ 
    - The “depth” of the filter is fixed by the number of filters in the previous layer  $K_{i-1}$
  - The stride  $S_i$
- For each downsampling/pooling layer:
  - Spatial extent of filter  $P_i \times P_i$
  - The stride  $D_i$
- For the final MLP:
  - Number of layers, and number of neurons in each layer

# Poll 4

@678

What is the relationship between the number of channels in the output of a convolutional layer and the number of neurons in the corresponding layer of a scanning MLP

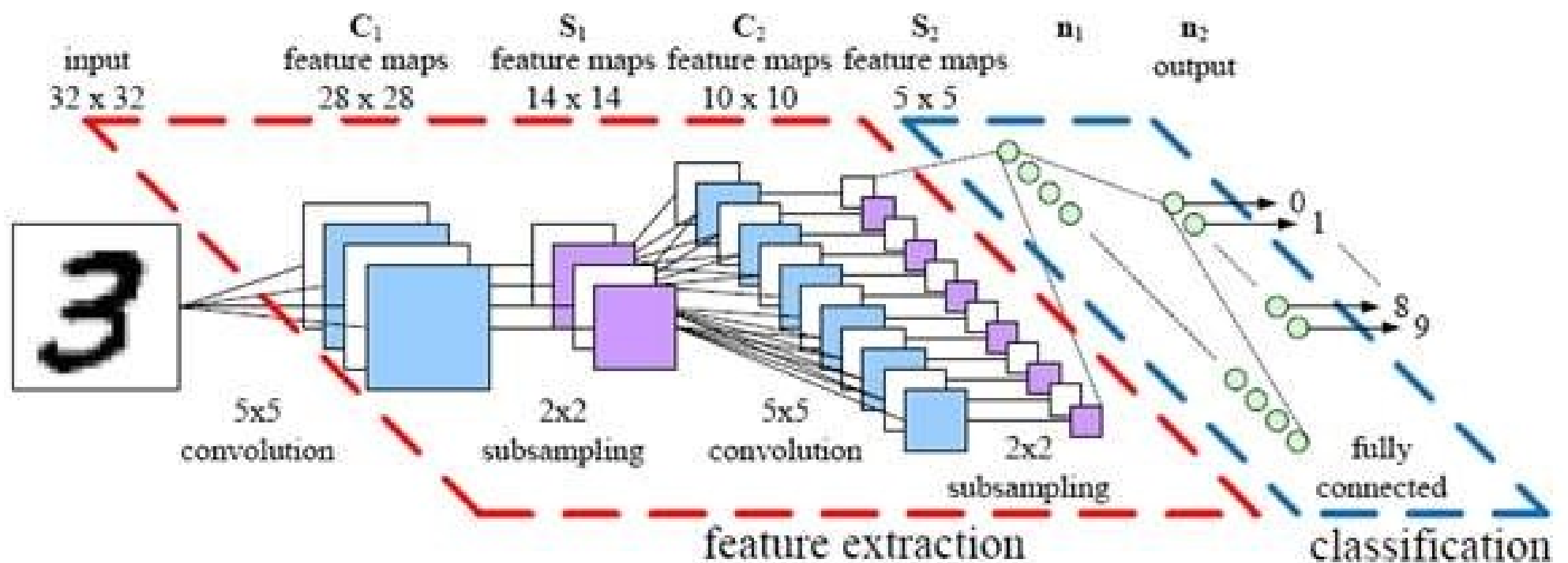
- They are the same
- The two are not related.

# Poll 4

What is the relationship between the number of channels in the output of a convolutional layer and the number of neurons in the corresponding layer of a scanning MLP

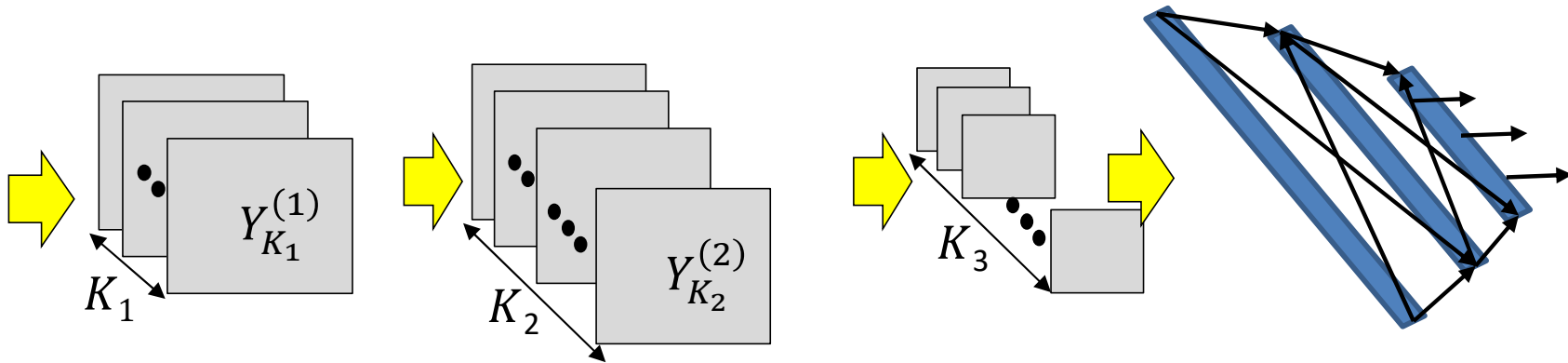
- **They are the same**
- The two are not related.

# Digit classification



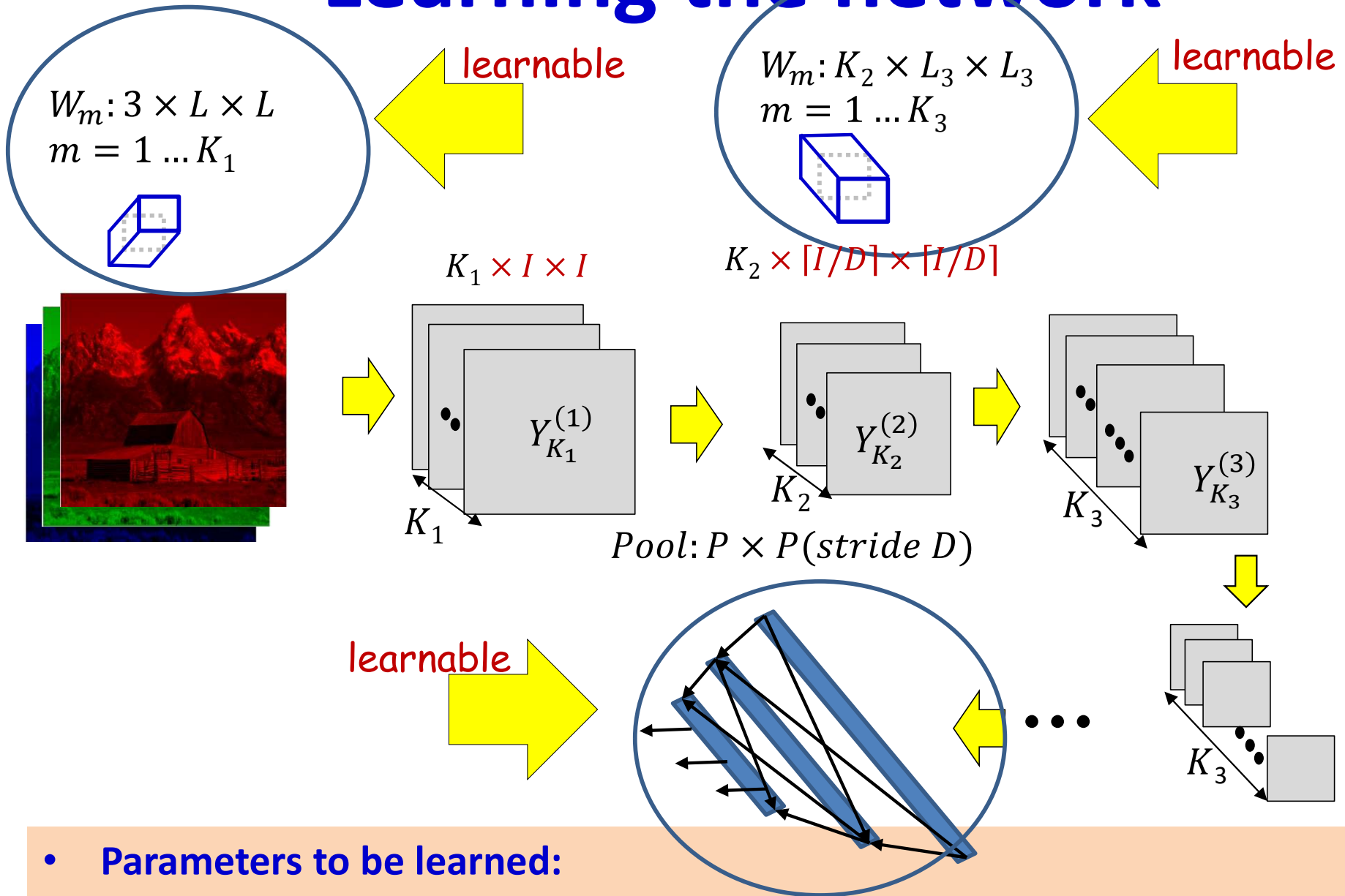


# Training



- Training is as in the case of the regular MLP
  - The *only* difference is in the *structure* of the network
- **Training examples of (Image, class) are provided**
- Define a divergence between the desired output and true output of the network in response to any input
- **Network parameters are trained through variants of gradient descent**
- **Gradients are computed through backpropagation**

# Learning the network

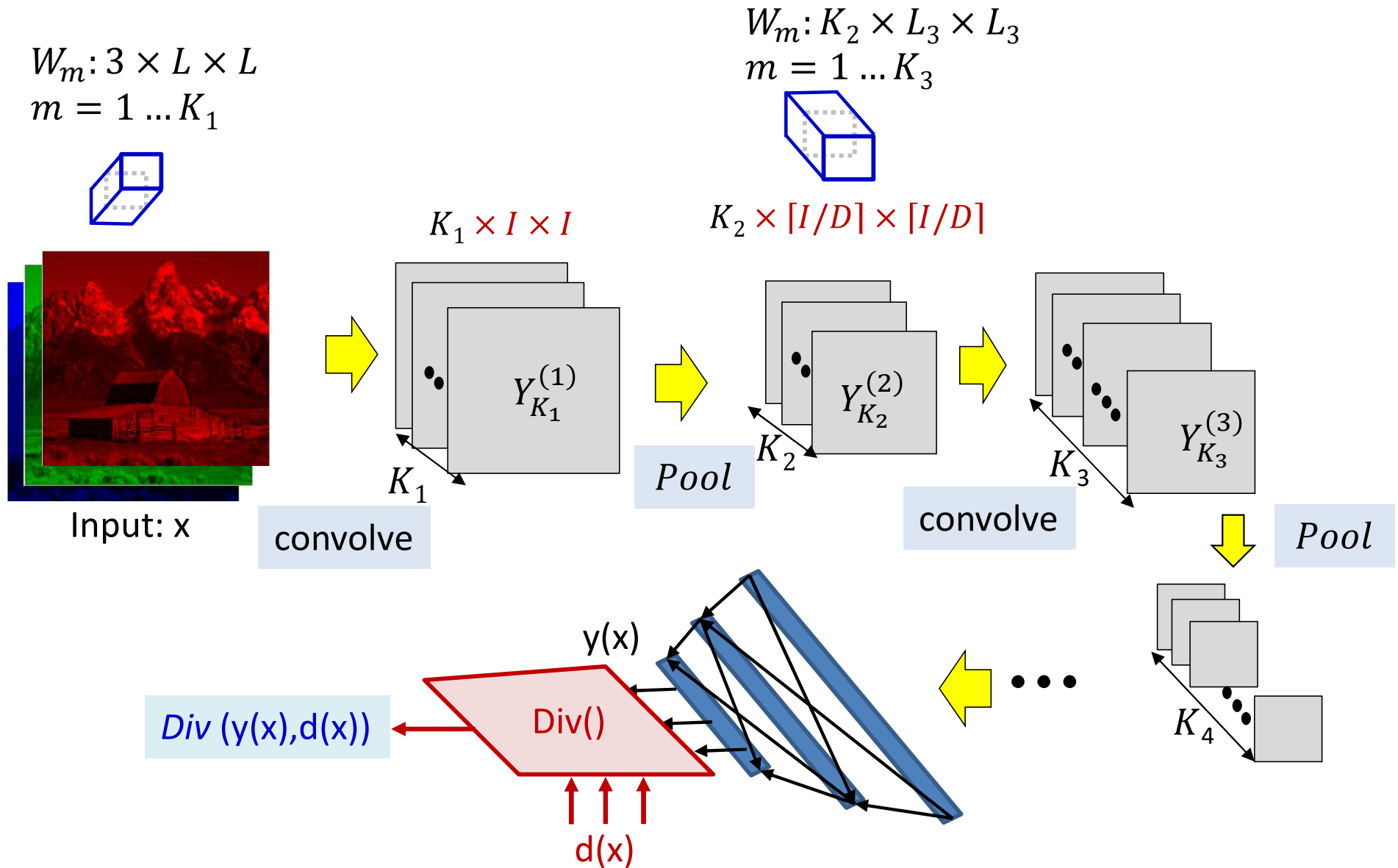


- Parameters to be learned:
  - The weights of the neurons in the final MLP
  - The (weights and biases of the) filters for every *convolutional* layer

# Learning the CNN

- In the final “flat” multi-layer perceptron, all the weights and biases of each of the perceptrons must be learned
- In the *convolutional layers* the filters must be learned
- Let each layer  $J$  have  $K_J$  maps
  - $K_0$  is the number of maps (colours) in the input
- Let the filters in the  $J^{\text{th}}$  layer be size  $L_J \times L_J$
- For the  $J^{\text{th}}$  layer we will require  $K_J(K_{J-1}L_J^2 + 1)$  filter parameters
- Total parameters required for the *convolutional layers*:  
$$\sum_{J \in \text{convolutional layers}} K_J(K_{J-1}L_J^2 + 1)$$

# Defining the loss



- The loss for a single instance

# Problem Setup

- Given a training set of input-output pairs  $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- The loss on the  $i^{\text{th}}$  instance is  $\text{div}(Y_i, d_i)$
- The total loss

$$\text{Loss} = \frac{1}{T} \sum_{i=1}^T \text{div}(Y_i, d_i)$$

- Minimize  $\text{Loss}$  w.r.t  $\{W_m, b_m\}$

# Training CNNs through Gradient Descent

Total training loss:

$$Loss = \frac{1}{T} \sum_{i=1}^T div(Y_i, d_i)$$

Assuming the bias is also represented as a weight

- Gradient descent algorithm:
- Initialize all weights and biases  $\{w(:, :, :, :, :)\}$
- Do:
  - For every layer  $l$  for all filter indices  $m$ , update:
    - $w(l, m, j, x, y) = w(l, m, j, x, y) - \eta \frac{dLoss}{dw(l, m, j, x, y)}$
- Until *Loss* has converged

# Training CNNs through Gradient Descent

Total training loss:

$$Loss = \frac{1}{T} \sum_{i=1}^T div(Y_i, d_i)$$

Assuming the bias is also represented as a weight

- Gradient descent algorithm:
- Initialize all weights and biases  $\{w(:, :, :, :, :)\}$
- Do:
  - For every layer  $l$  for all filter indices  $m$ , update:
    - $w(l, m, j, x, y) = w(l, m, j, x, y) - \eta \frac{dLoss}{dw(l, m, j, x, y)}$
- Until *Loss* has converged

# The derivative

Total training loss:

$$Loss = \frac{1}{T} \sum_i Div(Y_i, d_i)$$

- Computing the derivative

Total derivative:

$$\frac{dLoss}{dw(l, m, j, x, y)} = \frac{1}{T} \sum_i \frac{dDiv(Y_i, d_i)}{dw(l, m, j, x, y)}$$



# The derivative

Total training loss:

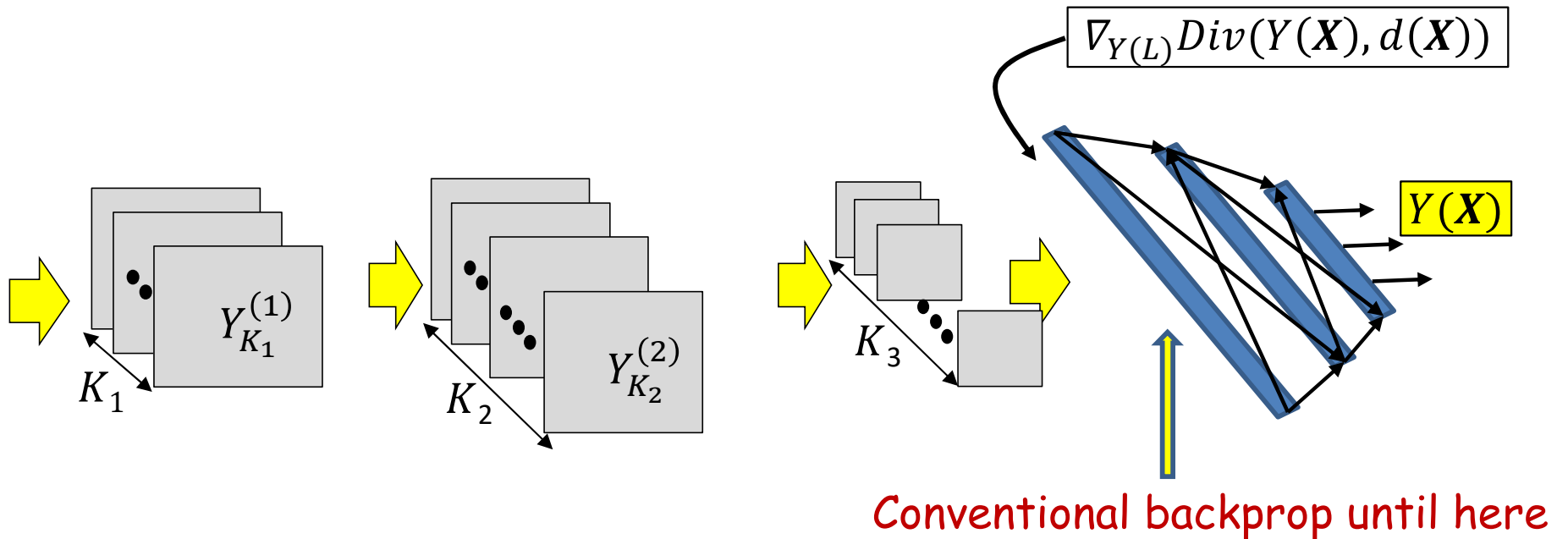
$$Loss = \frac{1}{T} \sum_i Div(Y_i, d_i)$$

- Computing the derivative

Total derivative:

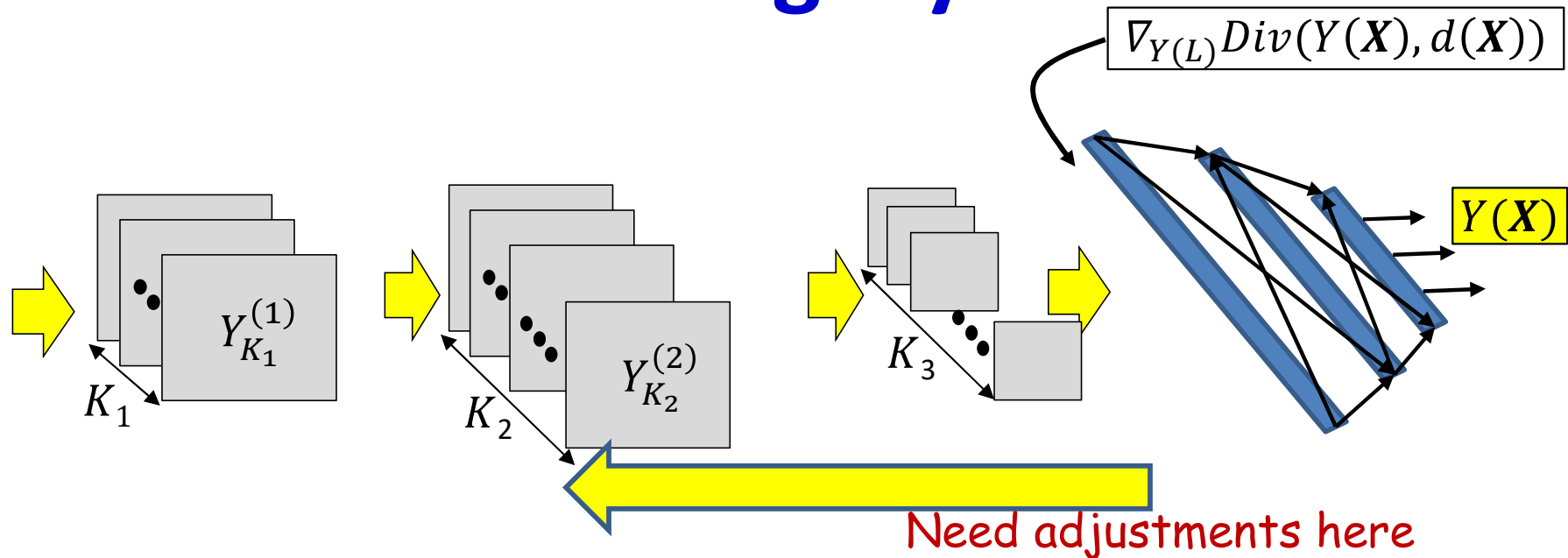
$$\frac{dLoss}{dw(l, m, j, x, y)} = \frac{1}{T} \sum_i \frac{dDiv(Y_i, d_i)}{dw(l, m, j, x, y)}$$

# Backpropagation: Final flat layers



- Backpropagation continues in the usual manner until the computation of the derivative of the divergence w.r.t the inputs to the first “flat” layer
  - Important to recall: the first flat layer is only the “flattening” of the maps from the final convolutional layer

# Backpropagation: Convolutional and Pooling layers

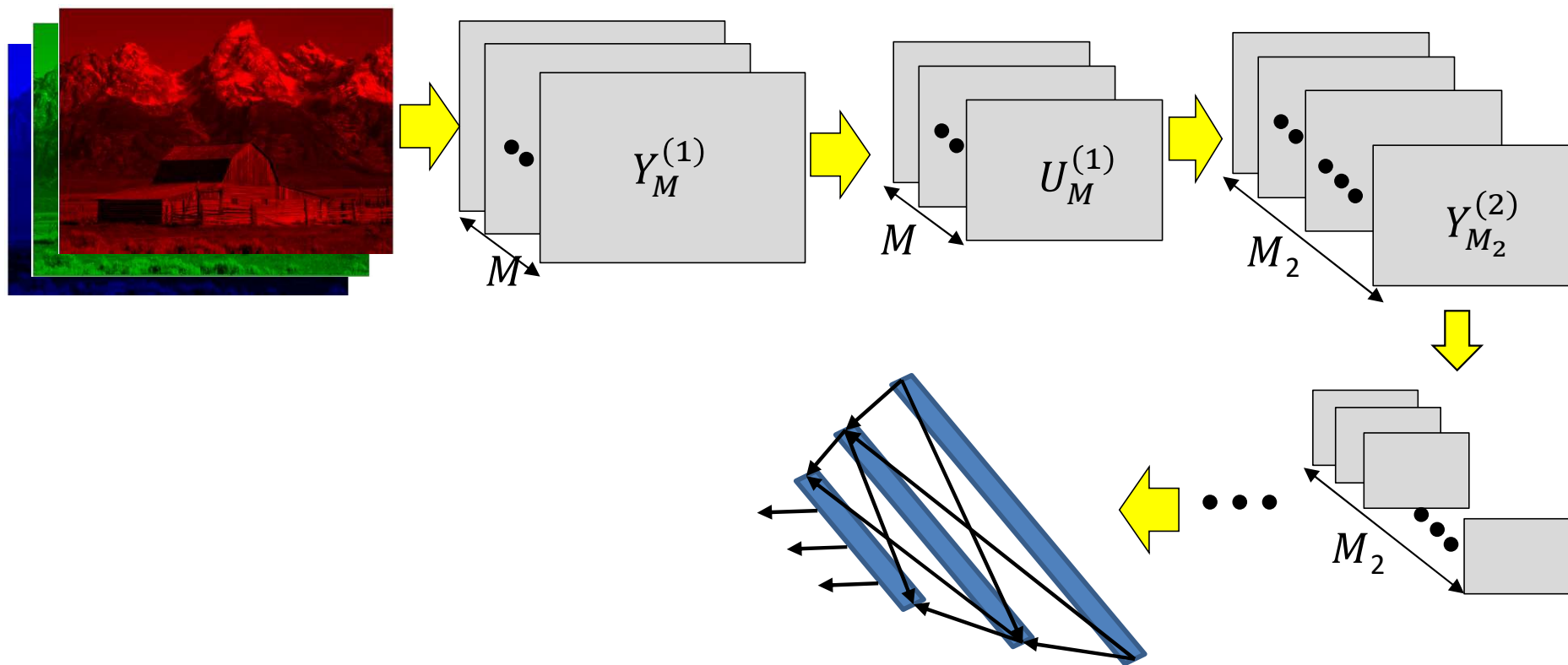


- Backpropagation from the flat MLP requires special consideration of
  - The shared computation in the convolutional layers
  - The pooling layers (particularly maxout)

# Backprop through a CNN

- In the next class...

# Learning the network



- Have shown the derivative of divergence w.r.t every intermediate output, and every free parameter (filter weights)
- Can now be embedded in gradient descent framework to learn the network

# Story so far

- The convolutional neural network is a supervised version of a computational model of mammalian vision
- It includes
  - Convolutional layers comprising learned filters that scan the outputs of the previous layer
  - Downsampling layers that operate over groups of outputs from the convolutional layer to reduce network size
- The parameters of the network can be learned through regular back propagation
  - Continued in next lecture..