Deep Neural Networks
Convolutional Networks II

Bhiksha Raj

Spring 2020

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 layer makes the final decision

- Deformations in the input can be handled by “max pooling”

- For 2-D (or higher-dimensional) scans, the structure is called a convnet
- 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?

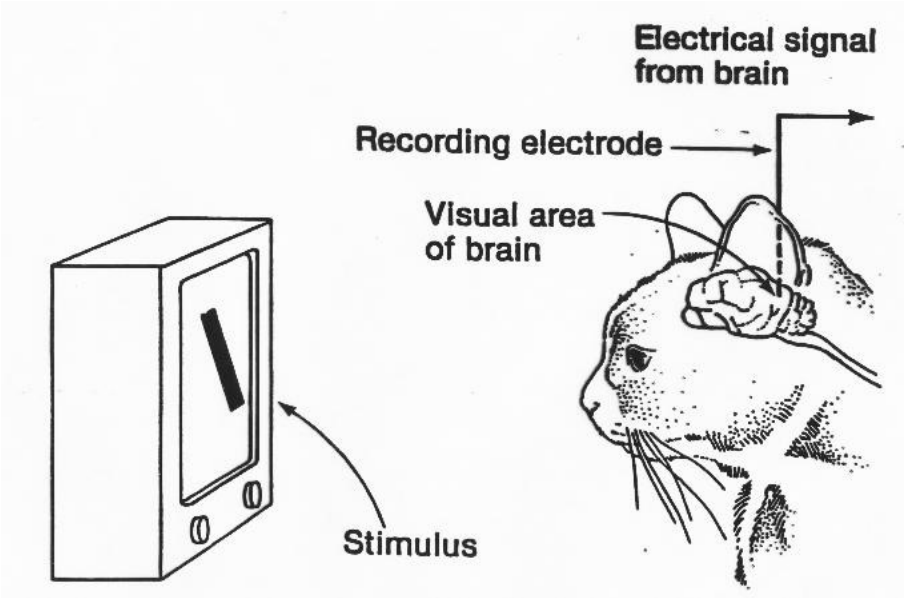
- Early research:
  - largely based on behavioral studies
    - Study behavioral judgment in response to visual stimulation
    - Visual illusions
  - and gestalt
    - Brain has innate tendency to organize disconnected bits into whole objects
  - But no real understanding of how the brain processed 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

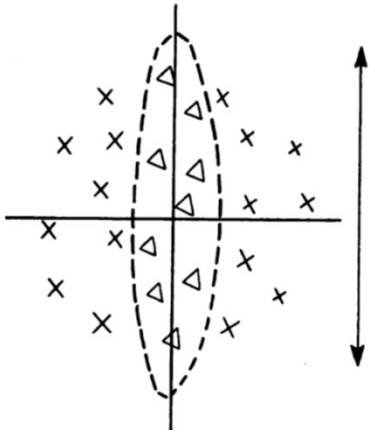
Hubel and Wiesel 1959



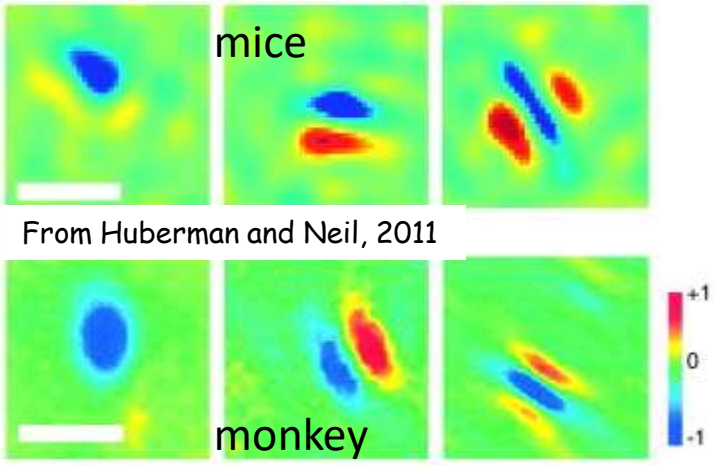
- Light of different wavelengths incident on the retina through fully open (slitted) Iris
  - Defines *immediate* (20ms) response of retinal cells

- Beamed light of different patterns into the eyes and measured neural responses in striate cortex

Hubel and Wiesel 1959



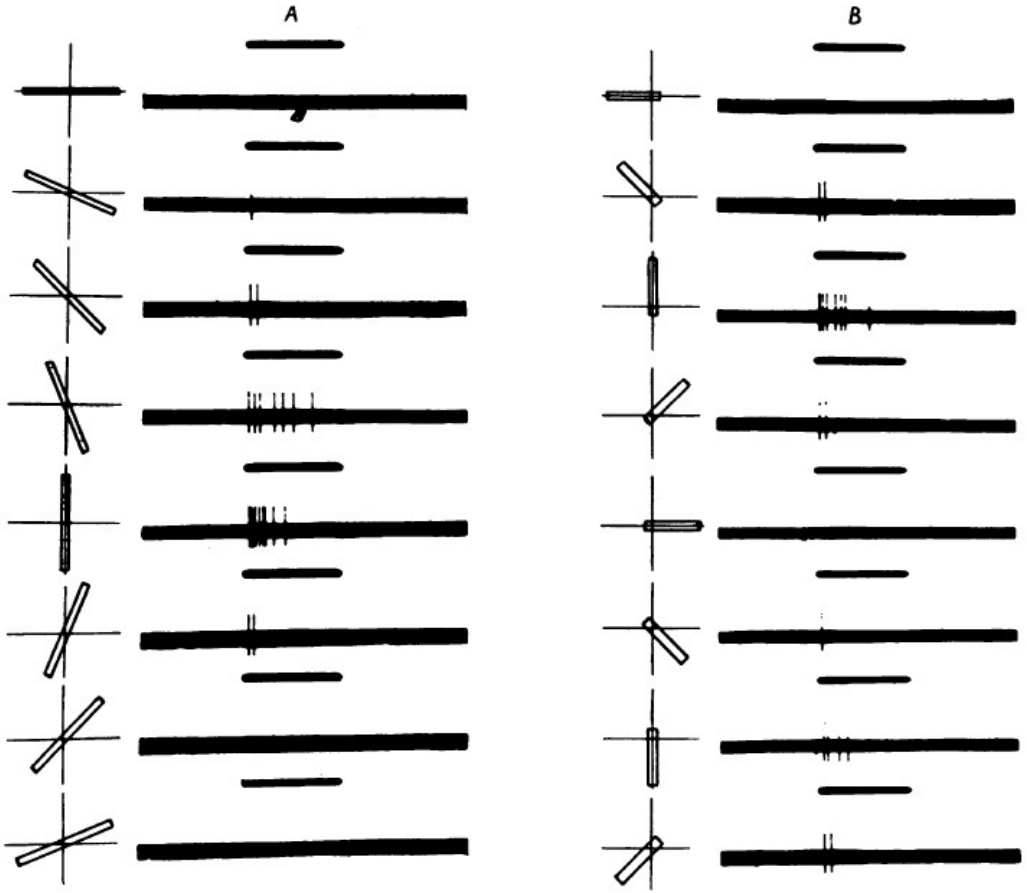
From Hubel and Wiesel



From Huberman and Neil, 2011

- 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
  - Receptive fields could be oriented in a vertical, horizontal or oblique manner.
    - Based on the arrangement of excitatory and inhibitory regions within receptive fields.
  - A spot of light gave greater response for some directions of movement than others.

Hubel and Wiesel 59



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

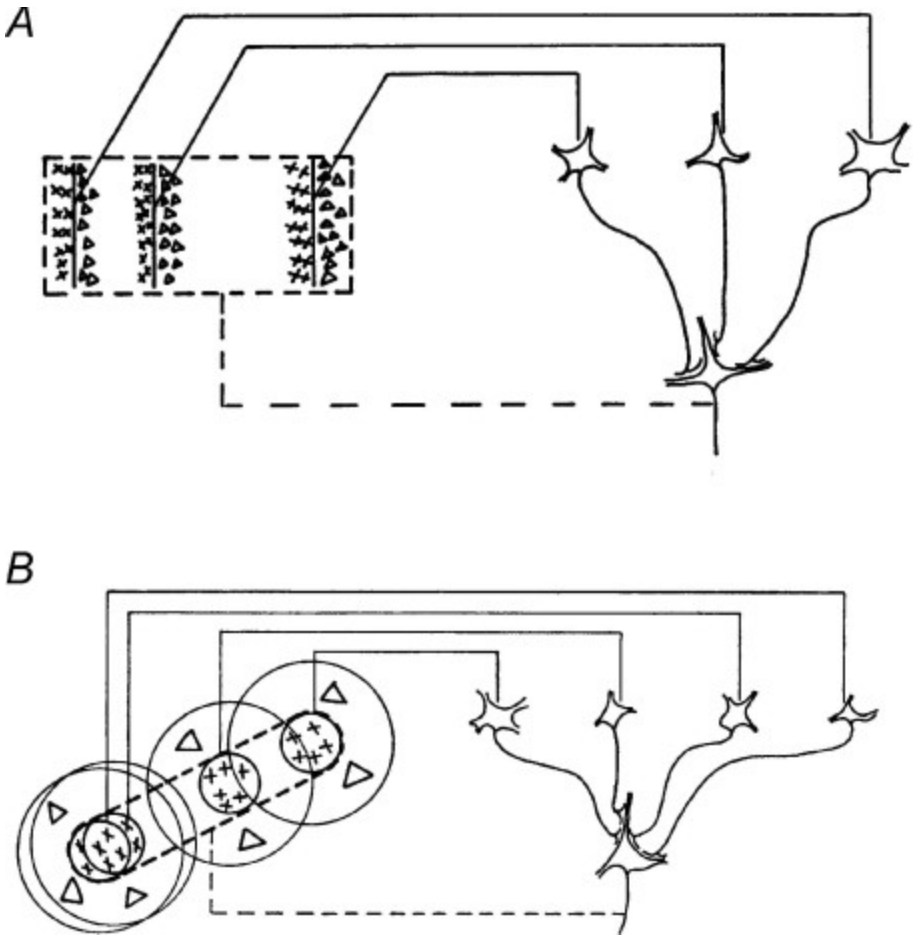
Hubel and Wiesel

- 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.”

Forward to 1980

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

Kunihiko 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

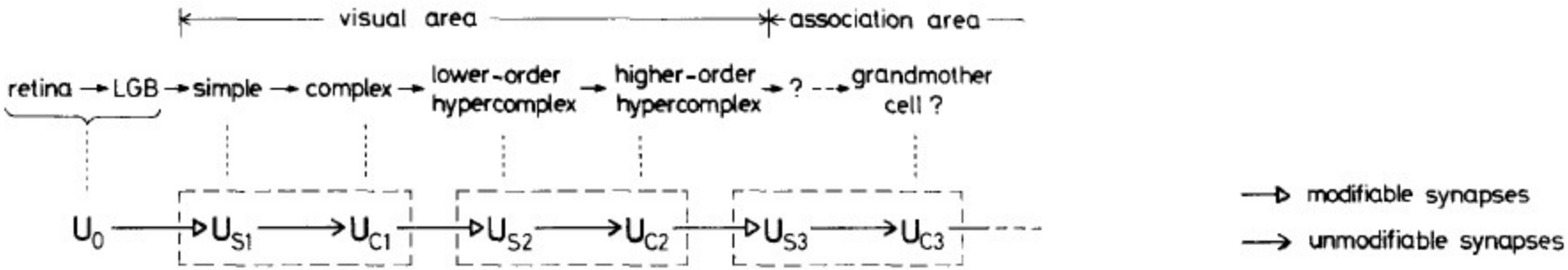
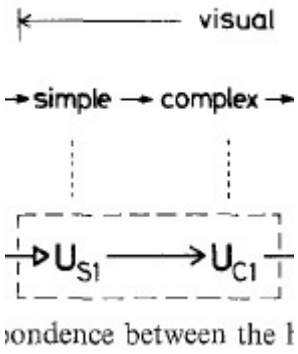
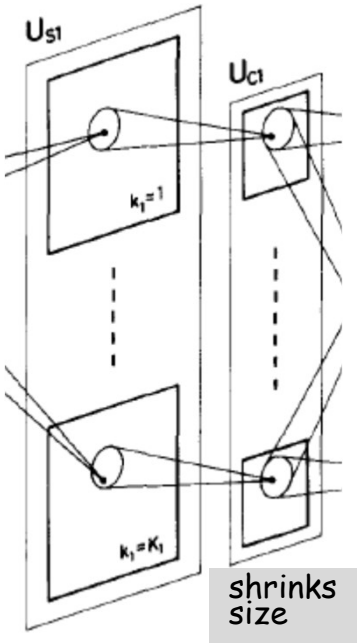


Fig. 1. Correspondence between the hierarchy model by Hubel and Wiesel, and the neural network of the neocognitron

- 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[th] layer of S cells, U_{Cl} is the l[th] layer of C cells
- Only S-cells are "plastic" (i.e. learnable), C-cells are fixed in their response
- S-cells ***respond*** to the signal in the previous layer
- C-cells ***confirm*** the S-cells' response

NeoCognitron





shrinks size

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

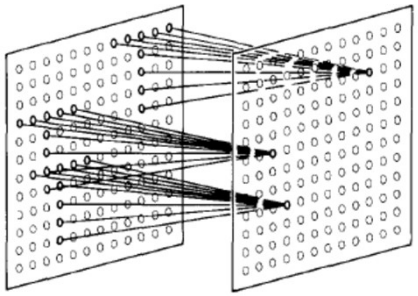
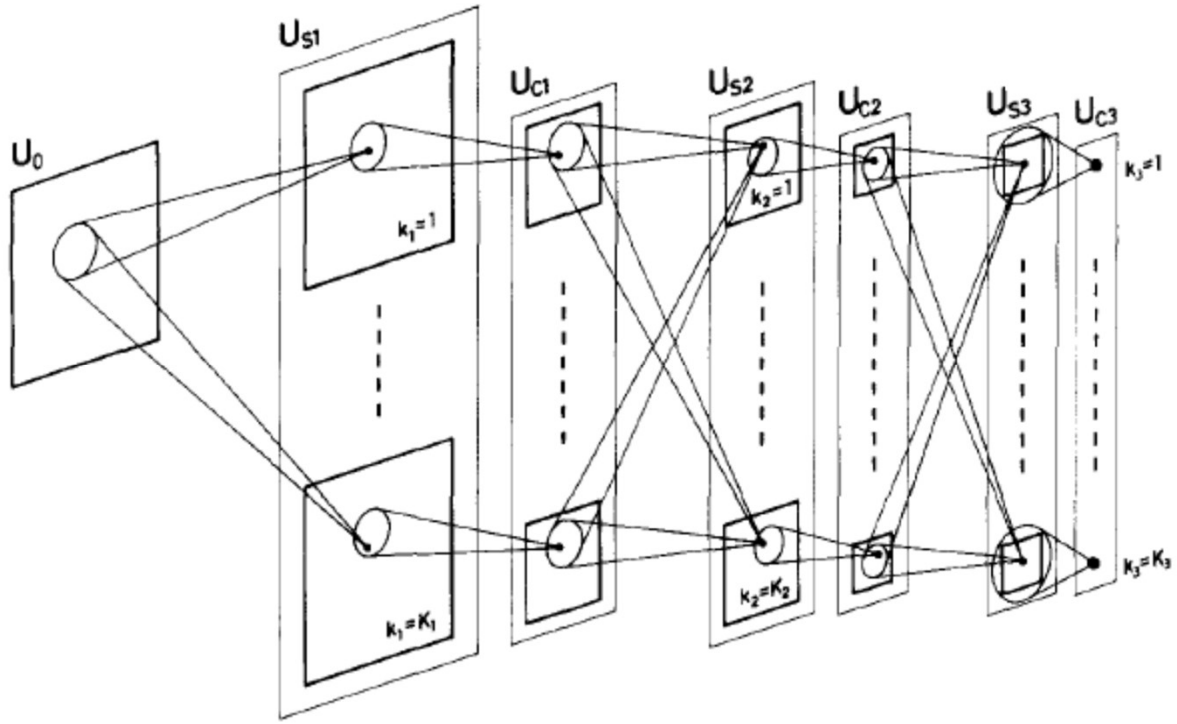


Fig. 3. Illustration showing the input interconnections to the cells within a single cell-plane

- 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

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

  - φ is a RELU

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

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

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

Neocognitron

- S cells: RELU like activation

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

  - φ is a RELU

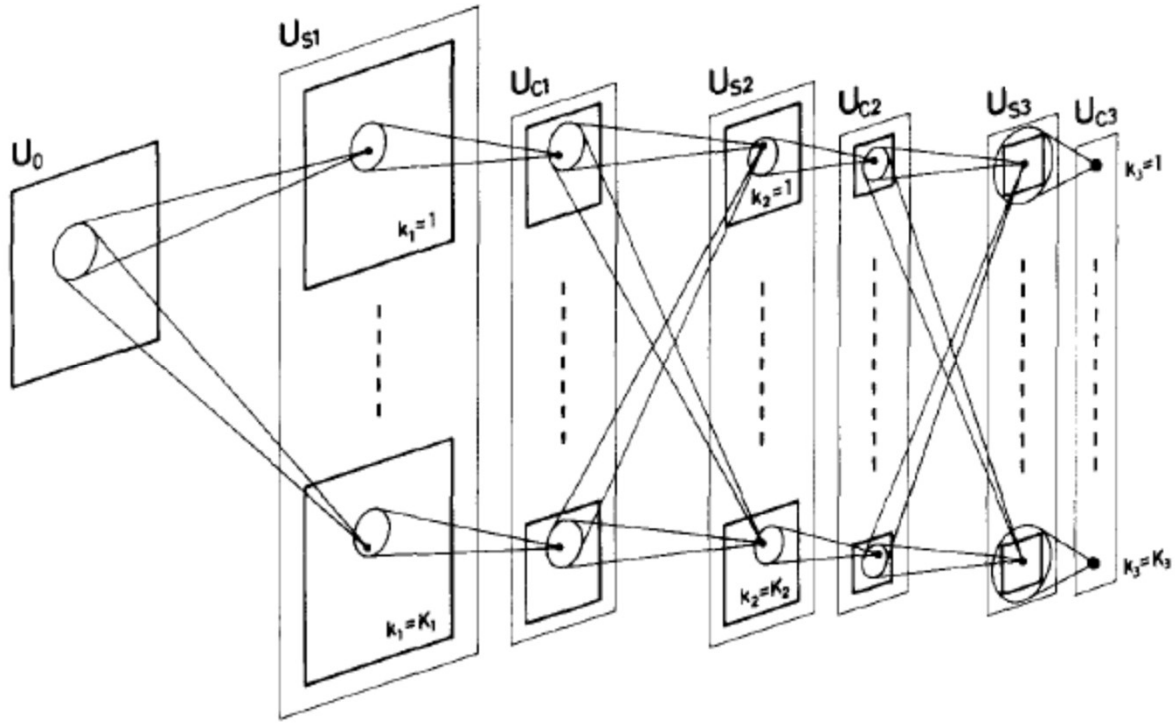
- C cells: Also RELU like, but with an inhibitory bias
  - Fires if weighted combination of S cells fires strongly enough

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

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

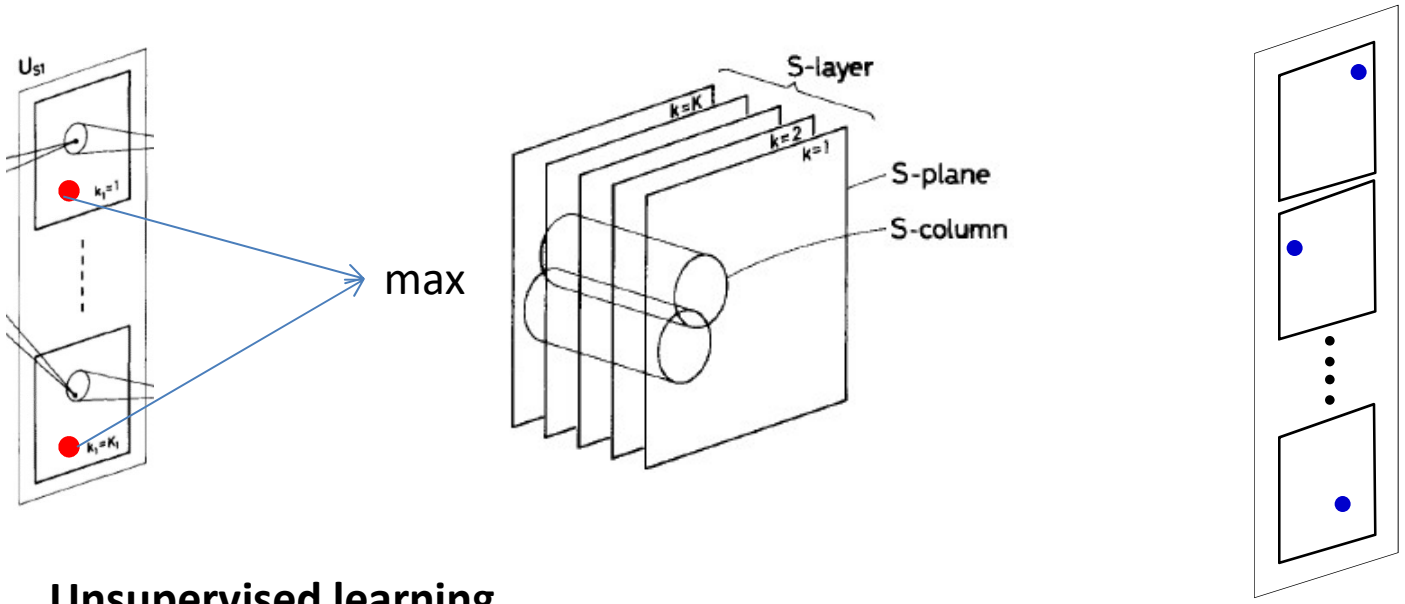
  - $\psi[x] = \varphi[x/(\alpha+x)]$

NeoCognitron



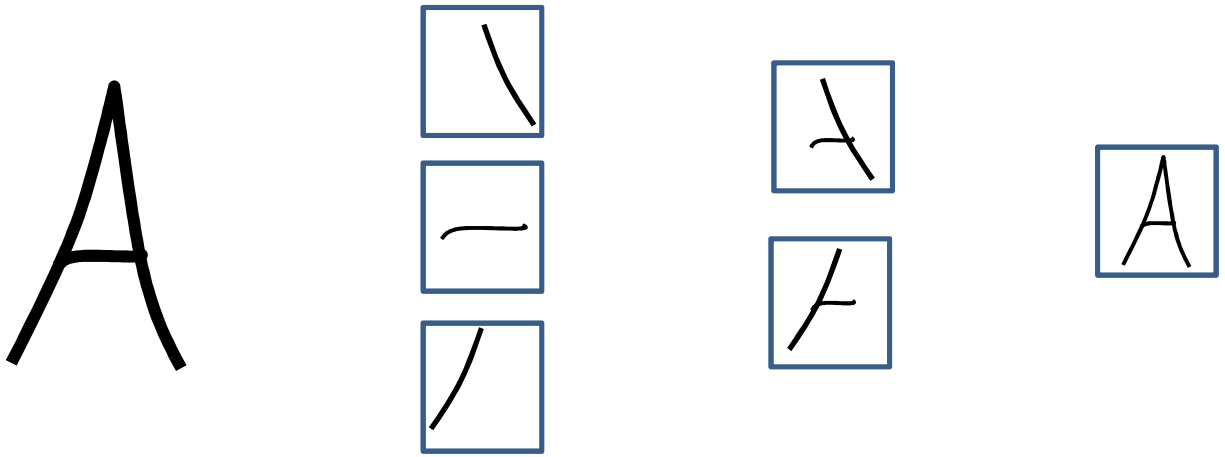
- The deeper the layer, the larger the receptive field of each neuron
  - Cell planes get smaller with layer number
  - Number of planes increases
    - i.e the number of complex pattern detectors increases with layer

Learning in the neo-cognitron



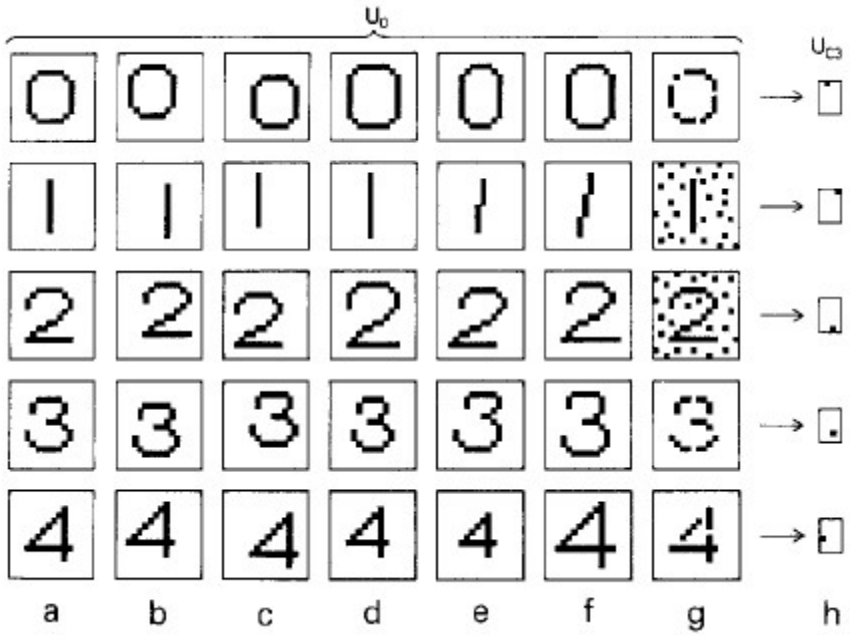
- **Unsupervised learning**
- Randomly initialize S cells, perform Hebbian learning updates in response to input
  - update = product of input and output : $\Delta w_{ij} = x_i y_j$
- Within any layer, at any position, only the maximum S from all the layers is selected for update
  - Also viewed as max-valued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

Learning in the neo-cognitron



- Ensures different planes learn different features
- Any plane learns only one feature
  - 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

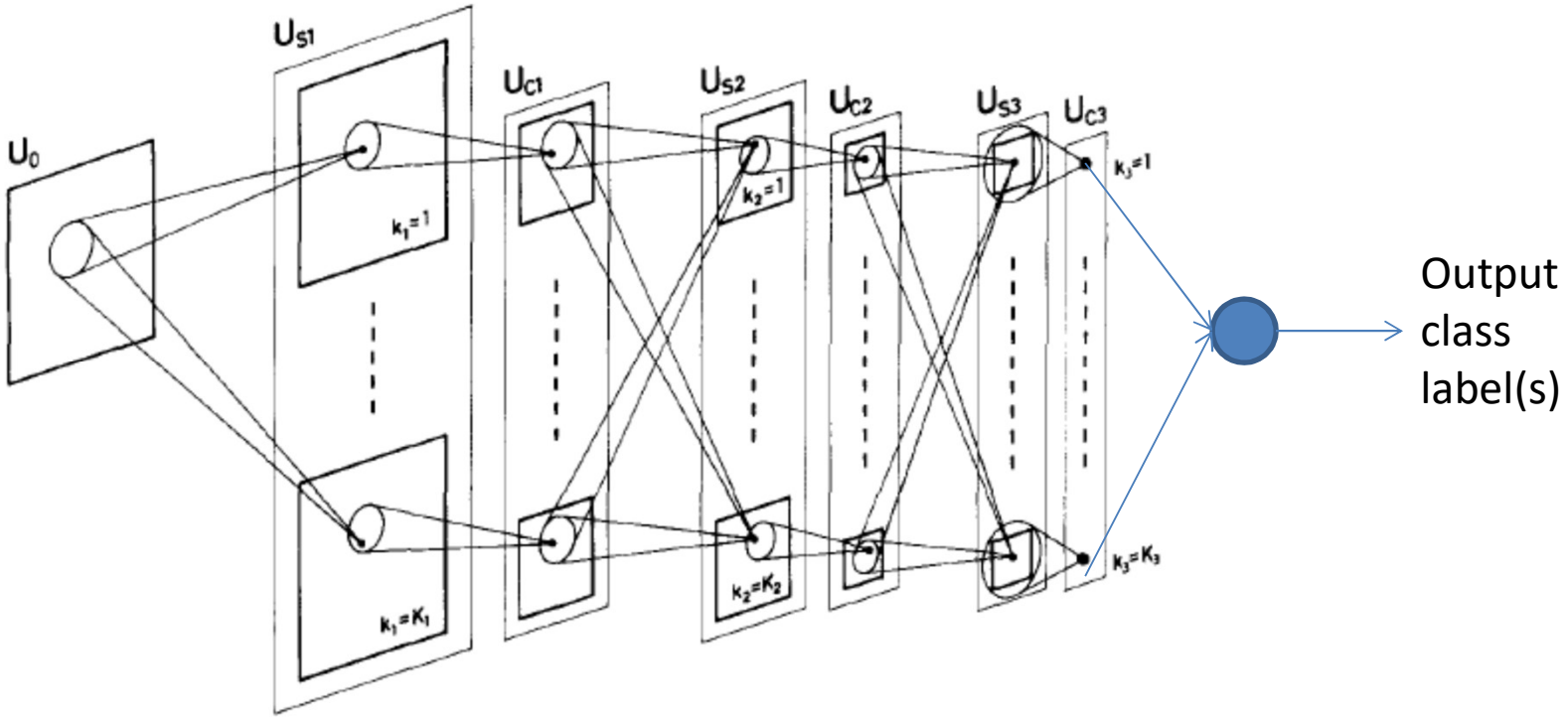


- Fukushima showed it successfully learns to cluster semantic visual concepts
  - E.g. number or characters, even in noise

Adding Supervision

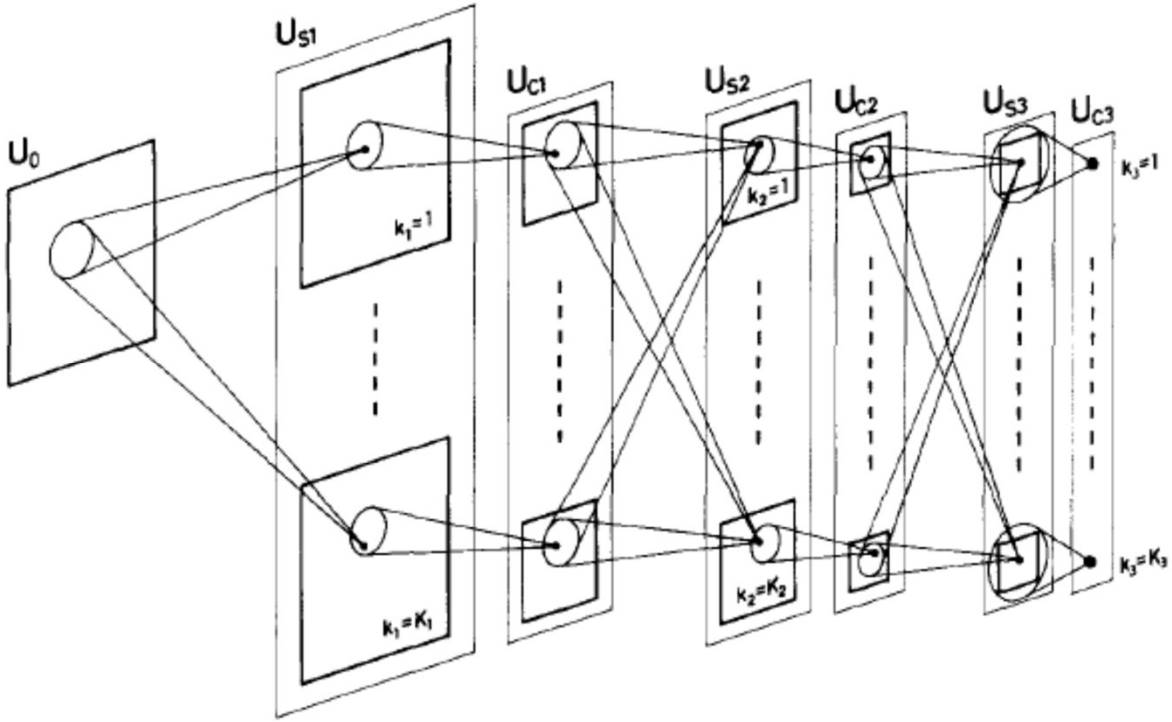
- The neocognitron is fully unsupervised
  - Semantic labels are automatically learned
- Can we add external supervision?
- Various proposals:
  - Temporal correlation: Homma, Atlas, Marks, '88
  - TDNN: Lang, Waibel et. al., 1989, '90
- Convolutional neural networks: LeCun

Supervising the neocognitron



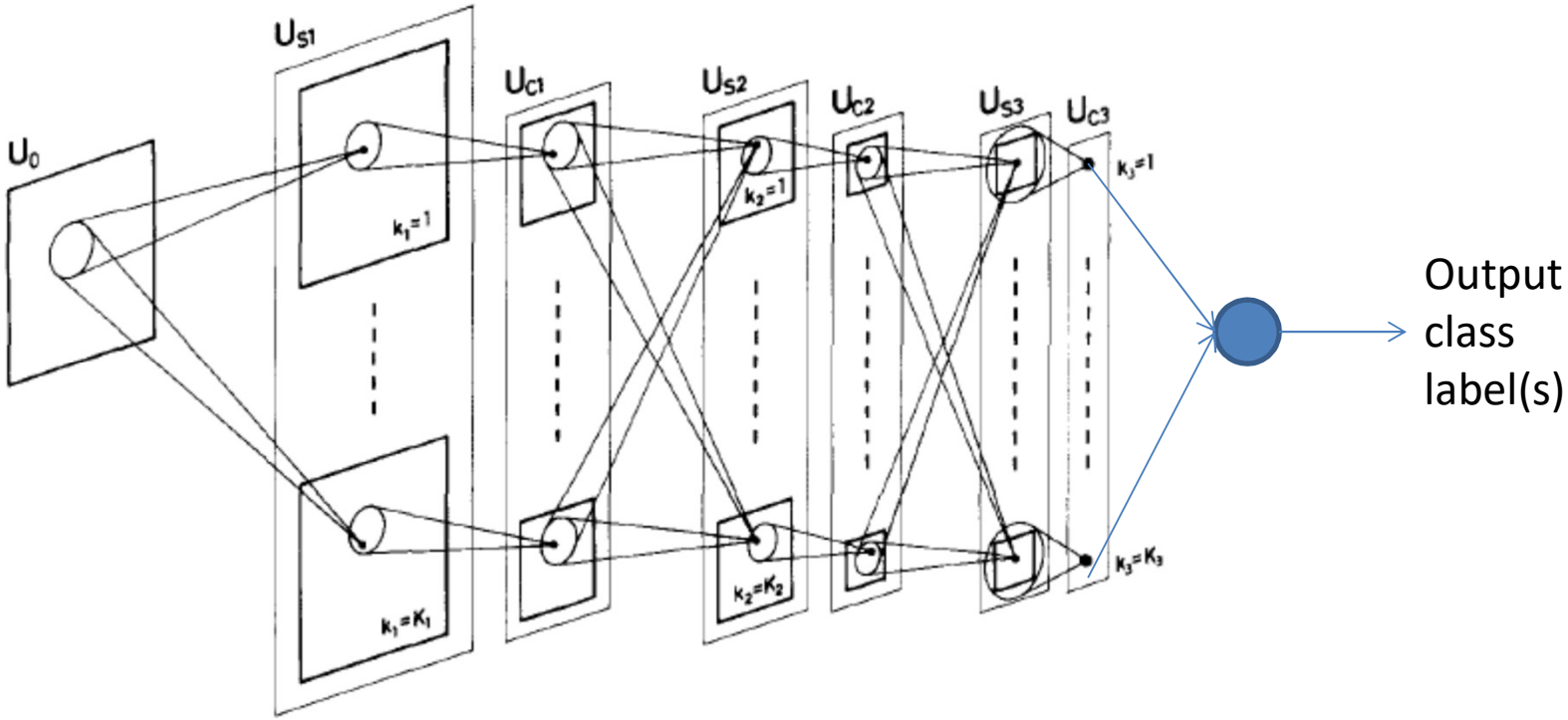
- Add an extra decision layer after the final C layer
  - Produces a class-label output
- We now have a fully feed forward MLP with shared parameters
  - All the S-cells within an S-plane have the same weights
- Simple backpropagation can now train the S-cell weights in every plane of every layer
  -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

Scanning vs. multiple filters



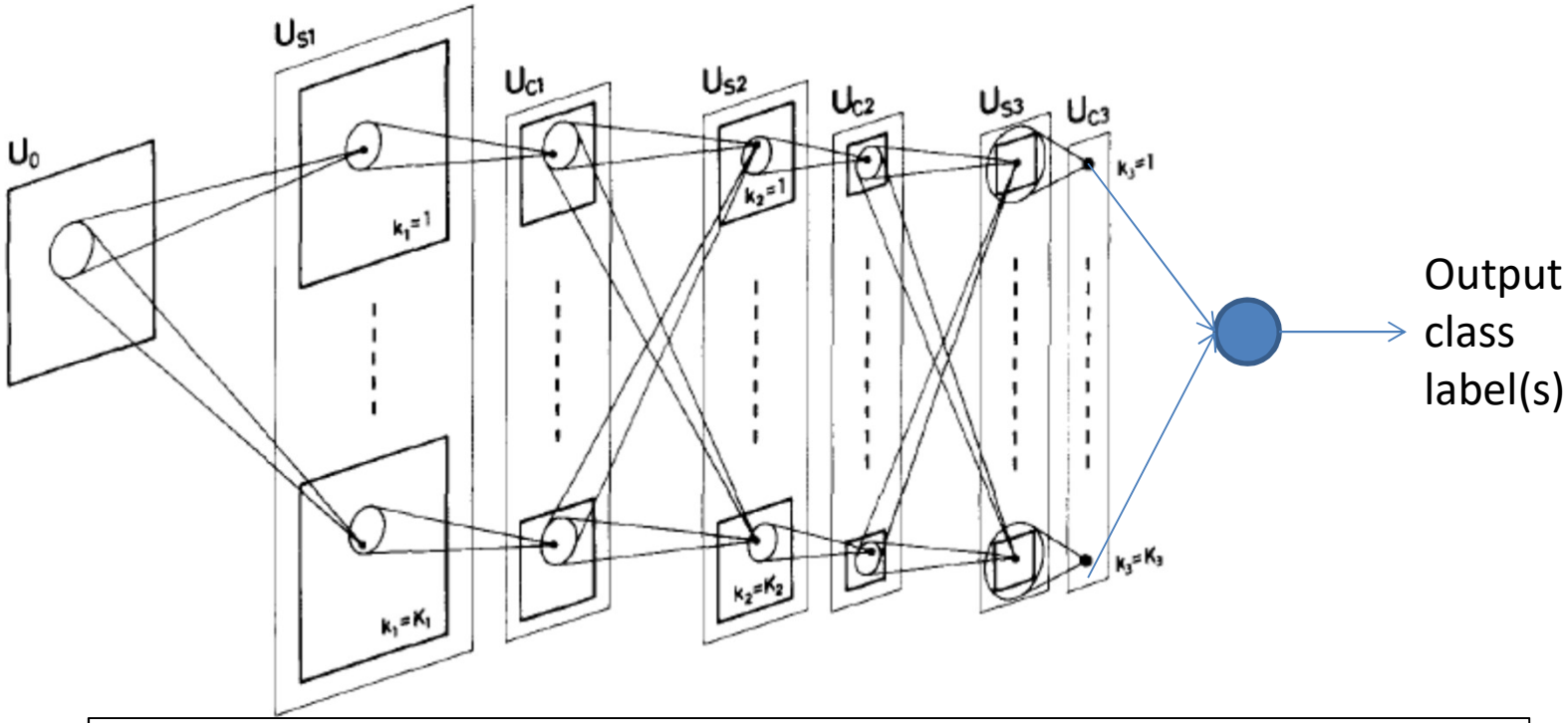
- **Note**: The original Neocognitron actually uses many identical copies of a neuron in each S and C plane

Supervising the neocognitron



- The Math
  - Assuming *square* receptive fields, rather than elliptical ones
  - Receptive field of S cells in lth layer is $K_l \times K_l$
  - Receptive field of C cells in lth layer is $L_l \times L_l$

Supervising the neocognitron

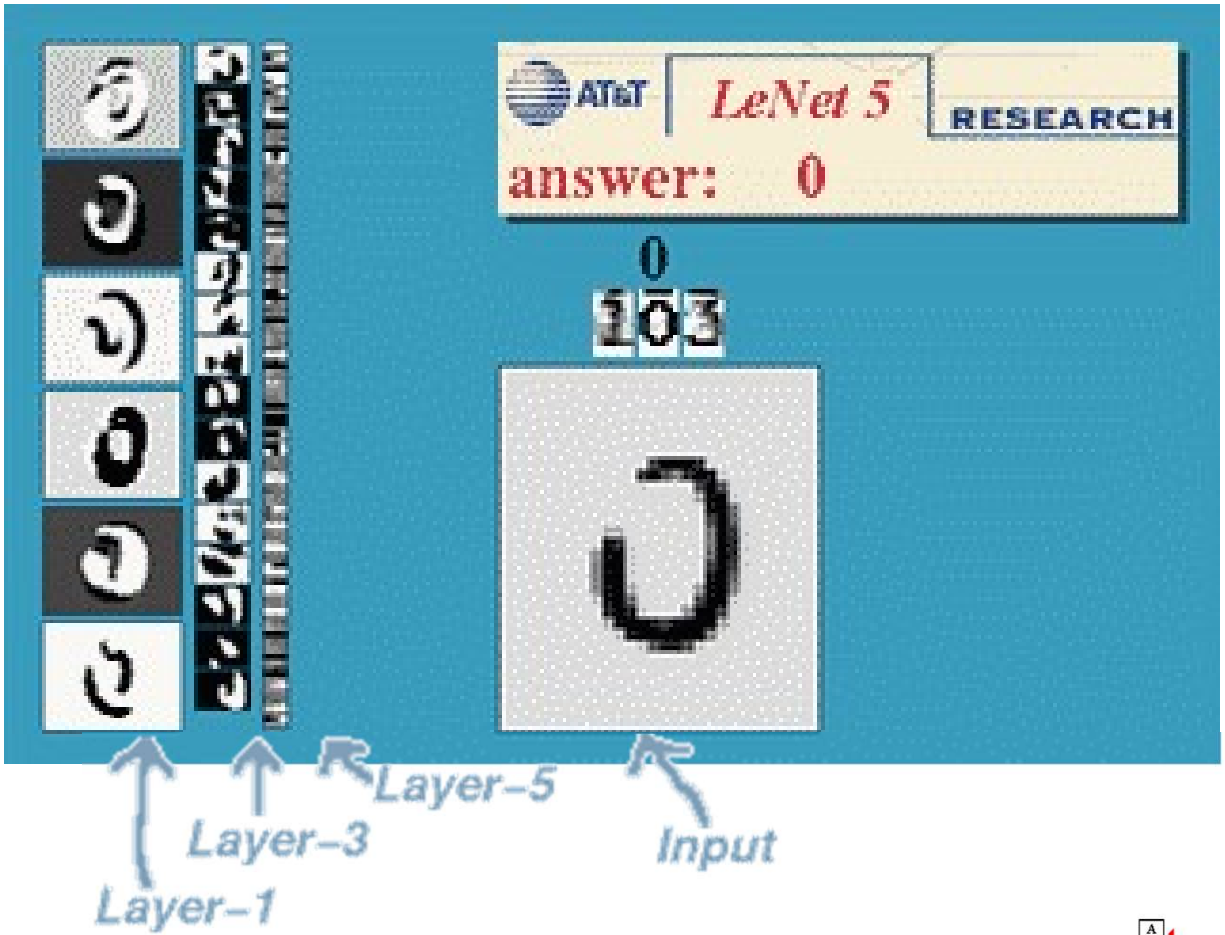


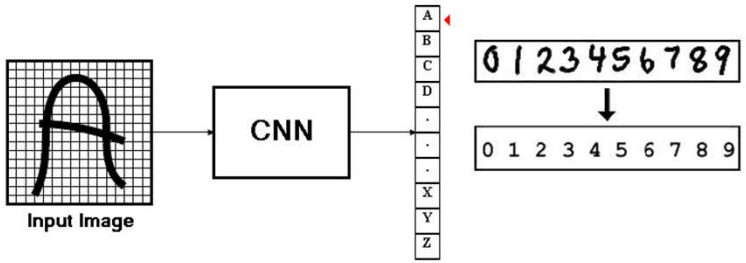
$$U_{S,l,n}(i,j) = \sigma\left(\sum_{p}\sum_{k=1}^{K_l}\sum_{l=1}^{K_l} w_{S,l,n}(p,k,l)\,U_{C,l-1,p}(i+l-1, j+k-1)\right)$$

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

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

Convolutional Neural Networks





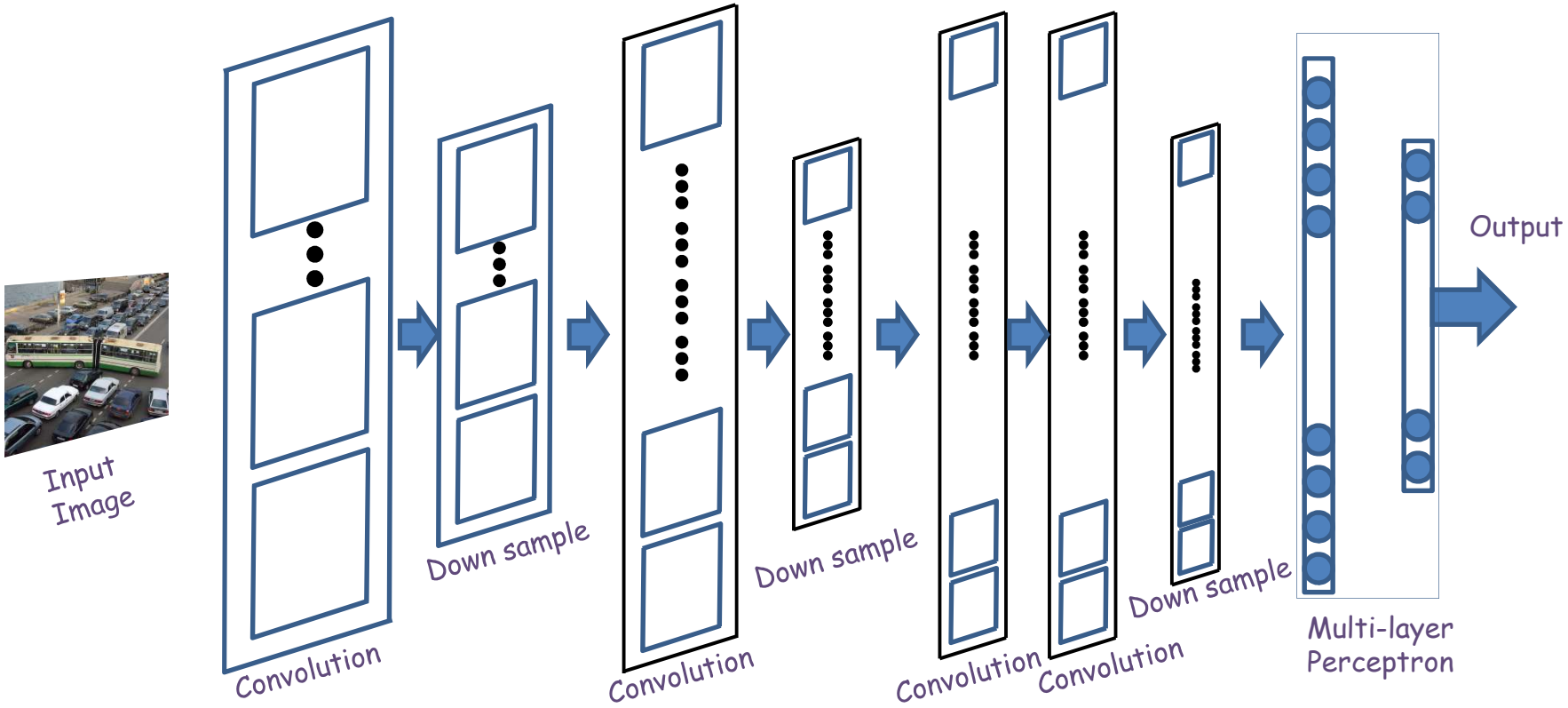
Story so far

- 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

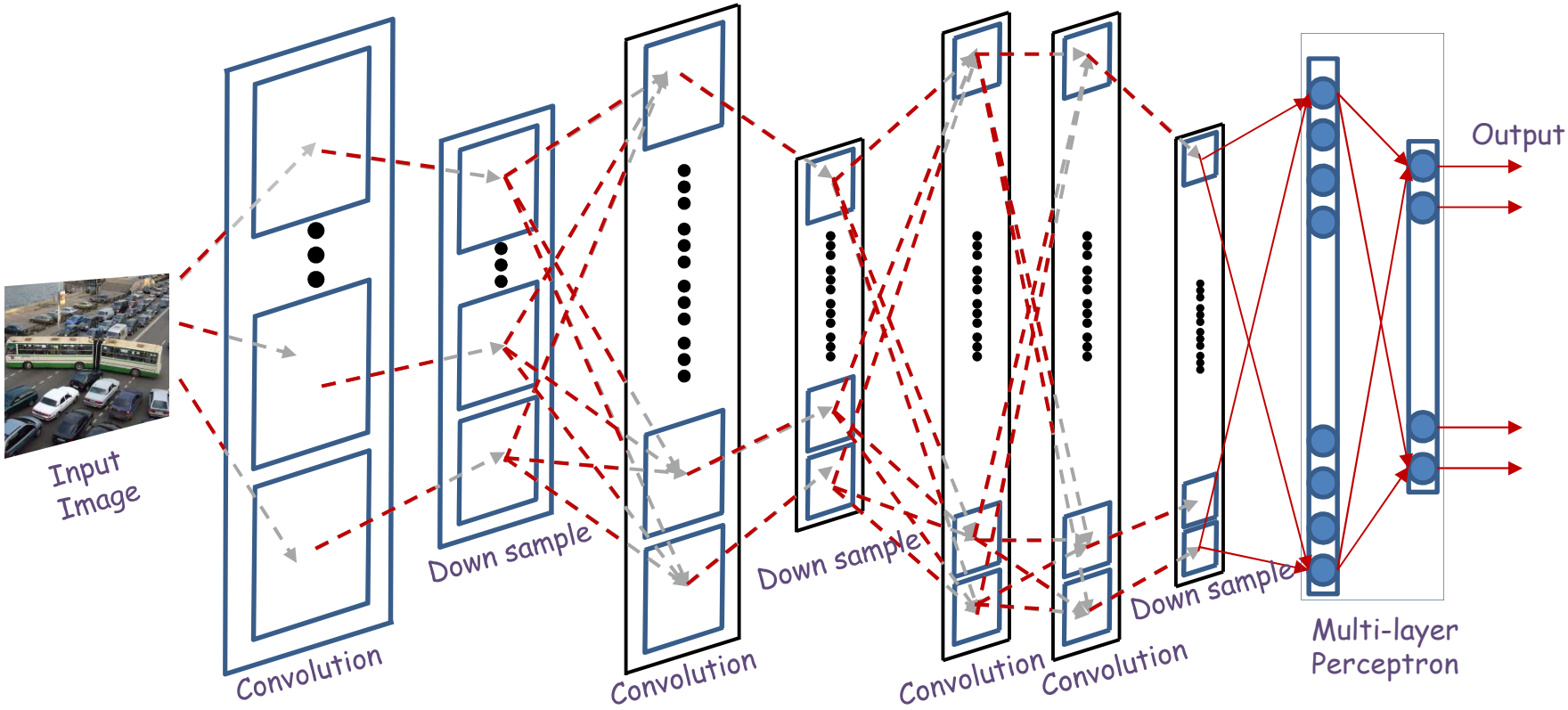
- LeCun’s LeNet added external supervision to the neocognitron
  - S planes of cells with identical response are modelled by a scan (convolution) over image planes by a single neuron
  - C planes are emulated by cells that perform a max over groups of S cells
    - Reducing the size of the S planes
  - Giving us a “Convolutional Neural Network”

The general architecture of a convolutional neural network



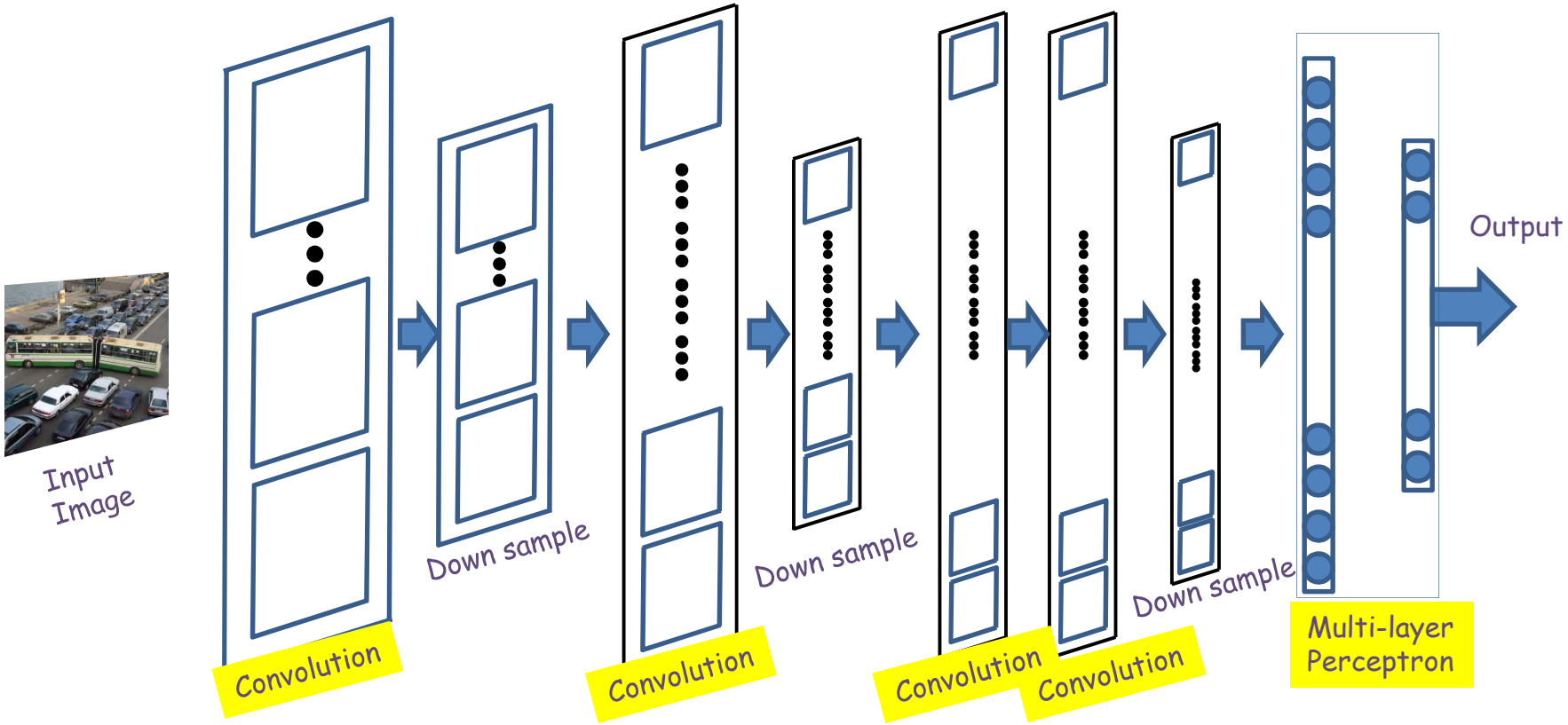
- A convolutional neural network comprises “convolutional” and “downsampling” layers
  - Convolutional layers comprise neurons that scan their input for patterns
  - Downsampling layers perform max operations on groups of outputs from the convolutional layers
  - The two may occur in any sequence, but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

The general architecture of a convolutional neural network



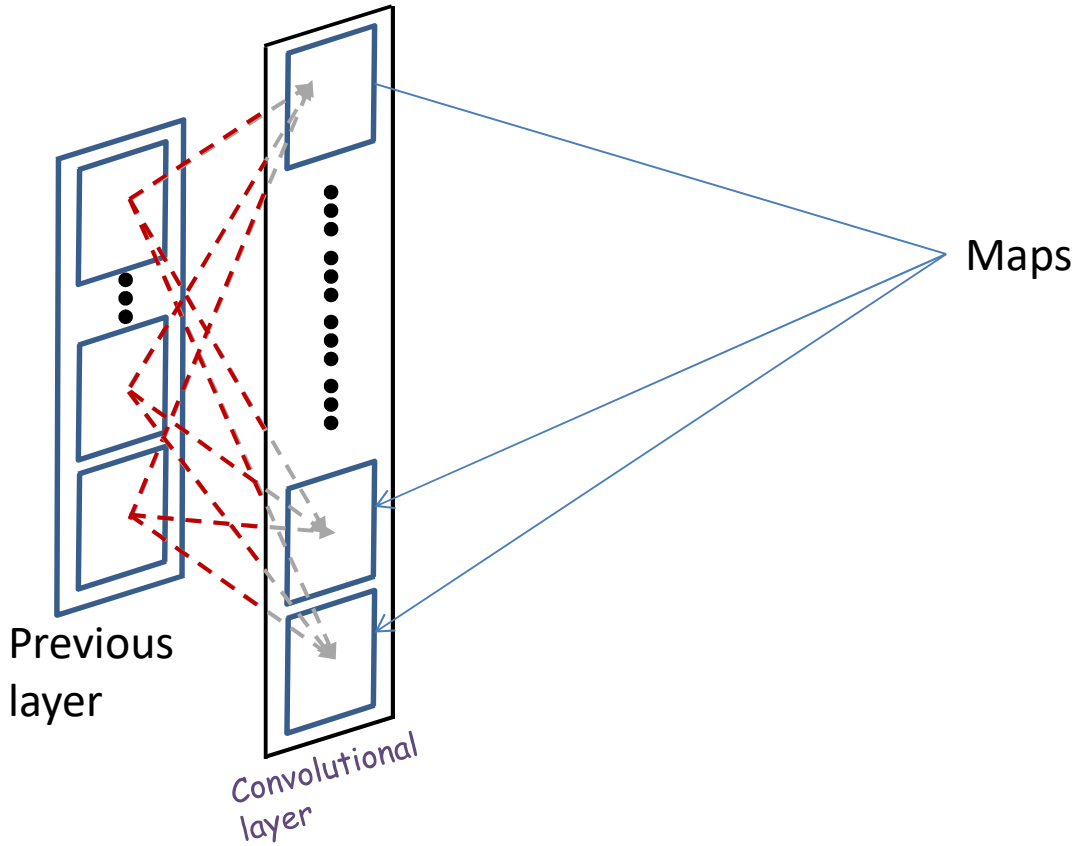
- A convolutional neural network comprises of "convolutional" and "downsampling" layers
  - The two may occur in any sequence, but typically they alternate
- Followed by an MLP with one or more layers

The general architecture of a convolutional neural network



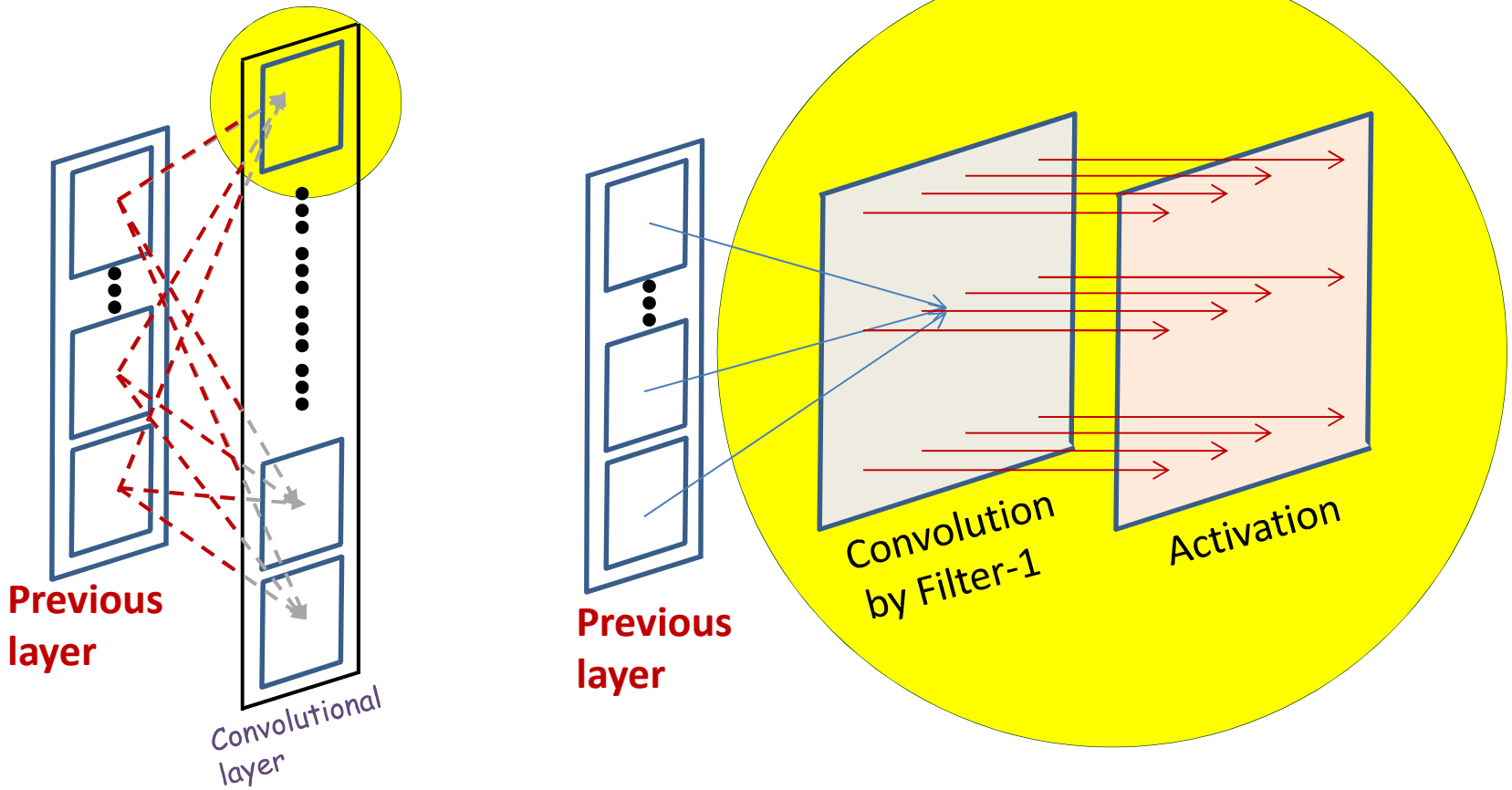
- **Convolutional layers and the MLP are *learnable***
  - Their parameters must be learned from training data for the target classification task
- Down-sampling layers are fixed and generally not learnable

A convolutional layer



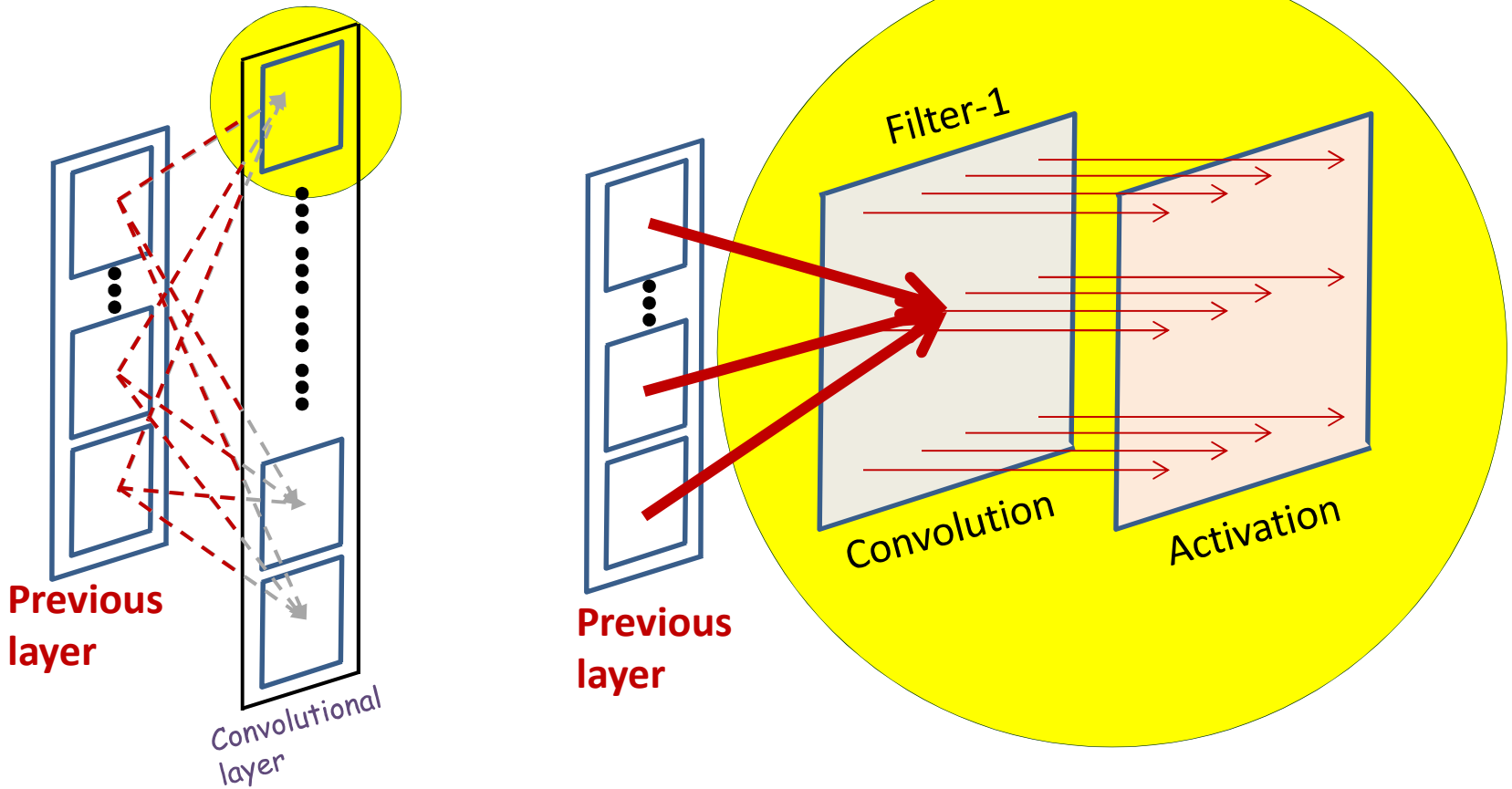
- A convolutional layer comprises of a series of “maps”
  - Corresponding the “S-planes” in the Neocognitron
  - Variously called feature maps or activation maps

A convolutional layer



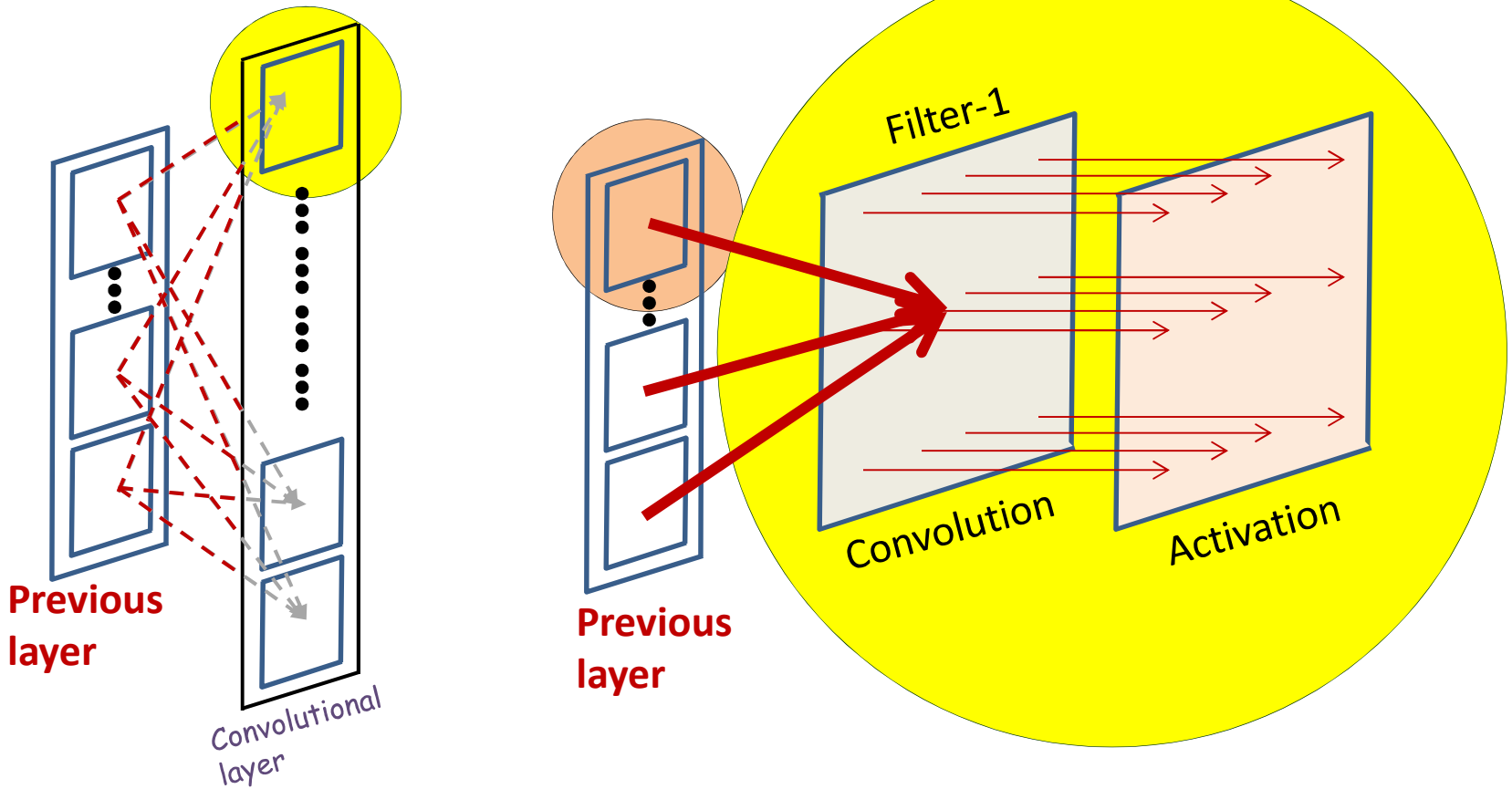
- Each activation map has two components
  - A *linear* map, obtained by *convolution* over maps in the previous layer
    - Each linear map has, associated with it, a ***learnable filter***
  - An *activation* that operates on the output of the convolution

A convolutional layer



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

A convolutional layer



- 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

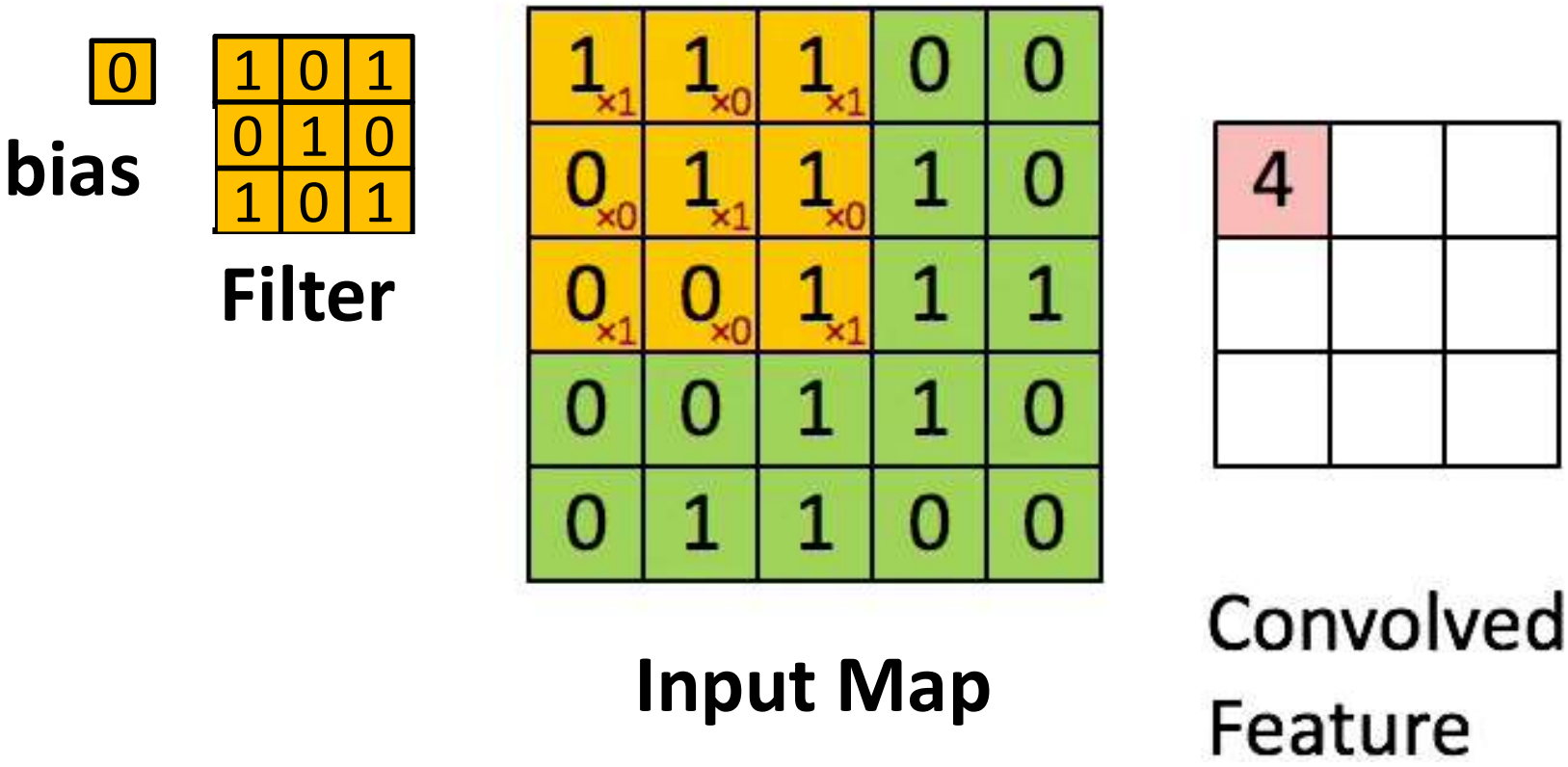
1	0	1
0	1	0
1	0	1

bias

0

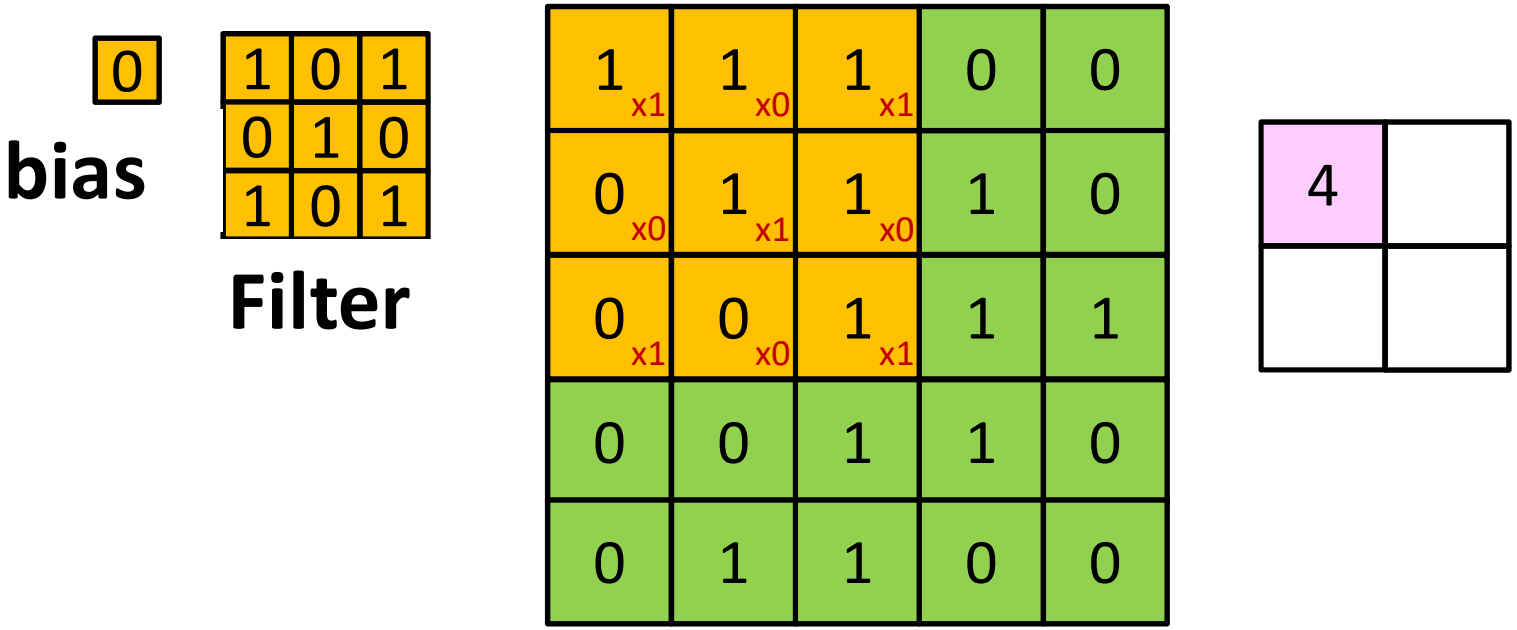
- Scanning an image with a “filter”
  - Note: a filter is really just a perceptron, with weights and a bias

What is a convolution



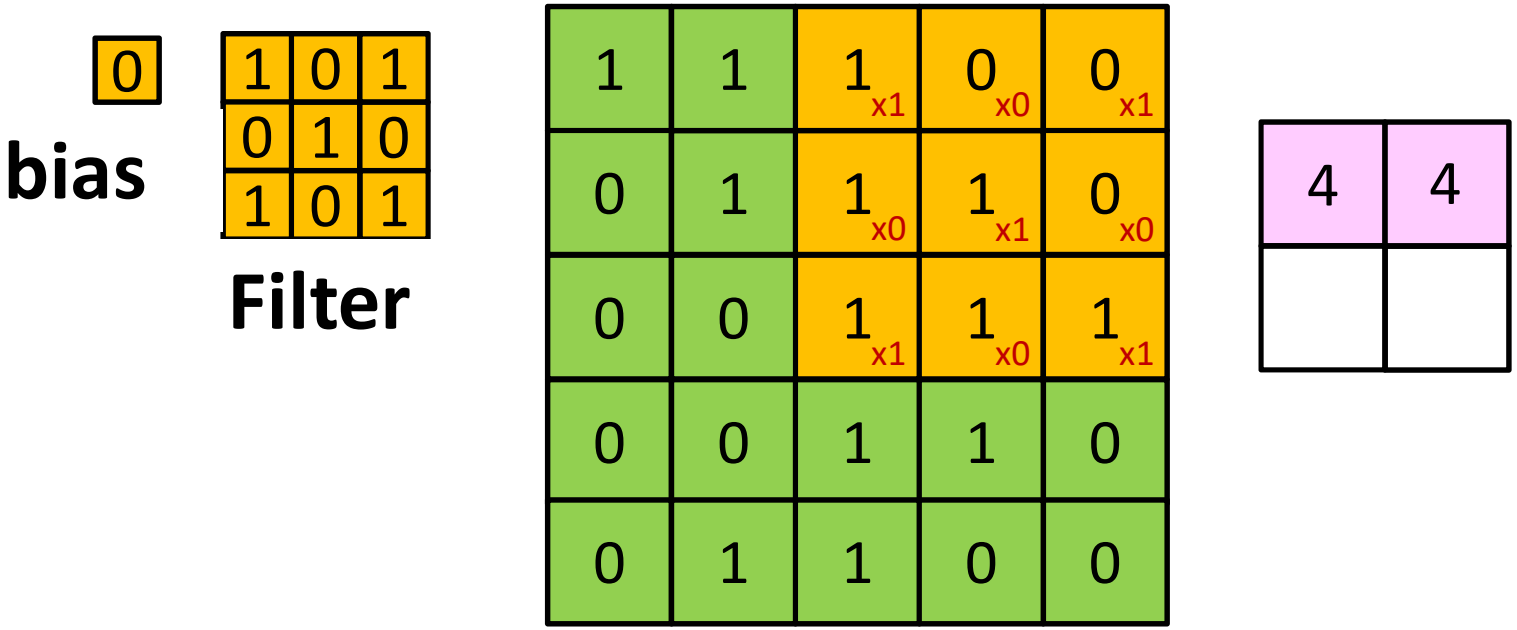
- Scanning an image with a “filter”
  - At each location, the “filter and the underlying map values are multiplied component wise, and the products are added along with the bias

The “Stride” between adjacent scanned locations need not be 1



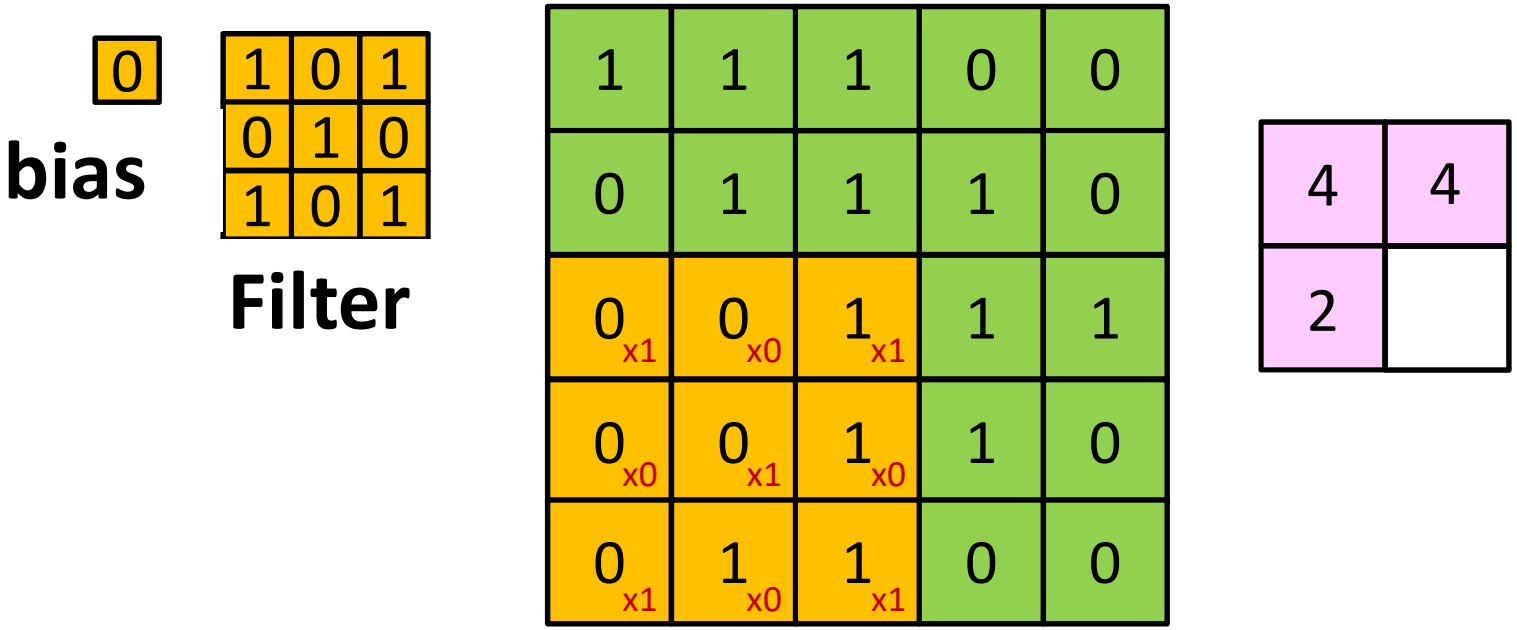
- Scanning an image with a “filter”
  - The filter may proceed by *more* than 1 pixel at a time
  - E.g. with a “stride” of *two* pixels per shift

The "Stride" between adjacent scanned locations need not be 1



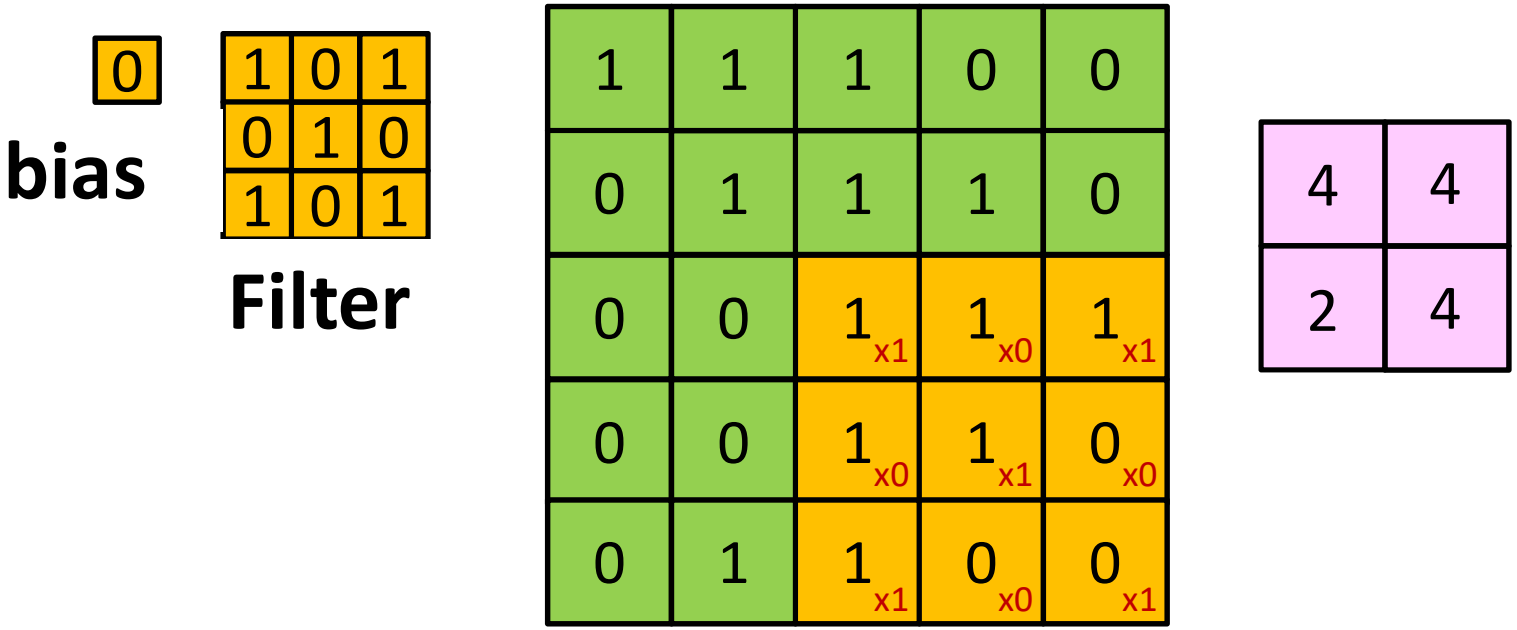
- Scanning an image with a "filter"
  - The filter may proceed by *more* than 1 pixel at a time
  - E.g. with a "hop" of *two* pixels per shift

The "Stride" between adjacent scanned locations need not be 1



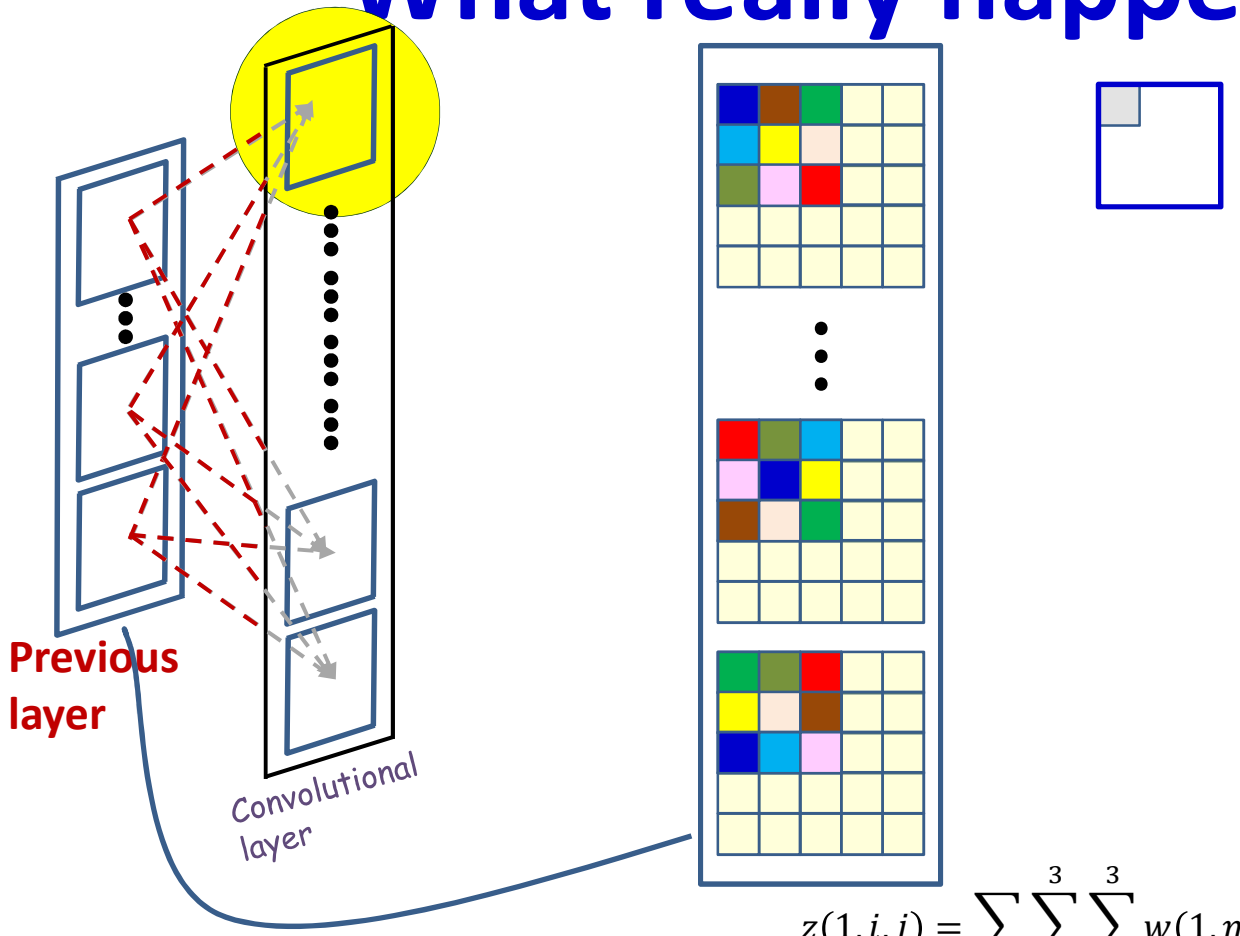
- Scanning an image with a "filter"
  - The filter may proceed by *more* than 1 pixel at a time
  - E.g. with a "hop" of *two* pixels per shift

The "Stride" between adjacent scanned locations need not be 1



- Scanning an image with a "filter"
  - The filter may proceed by *more* than 1 pixel at a time
  - E.g. with a "hop" of *two* pixels per shift

What really happens

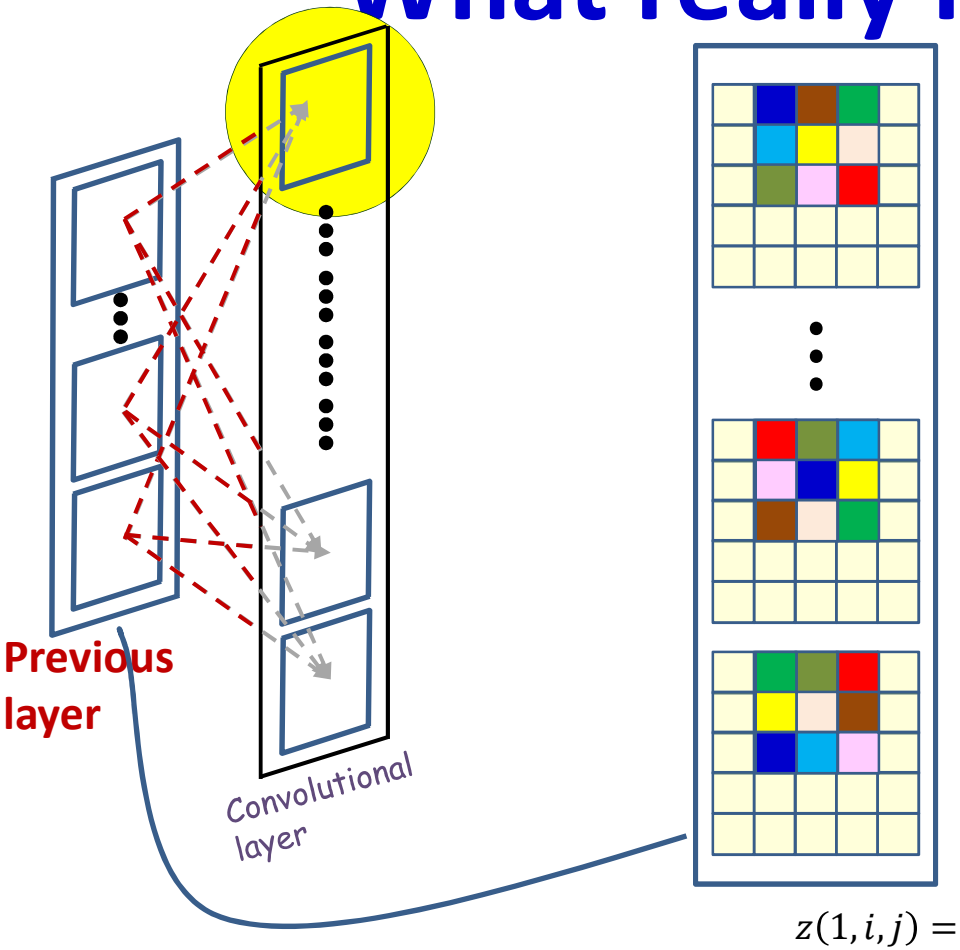


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- 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

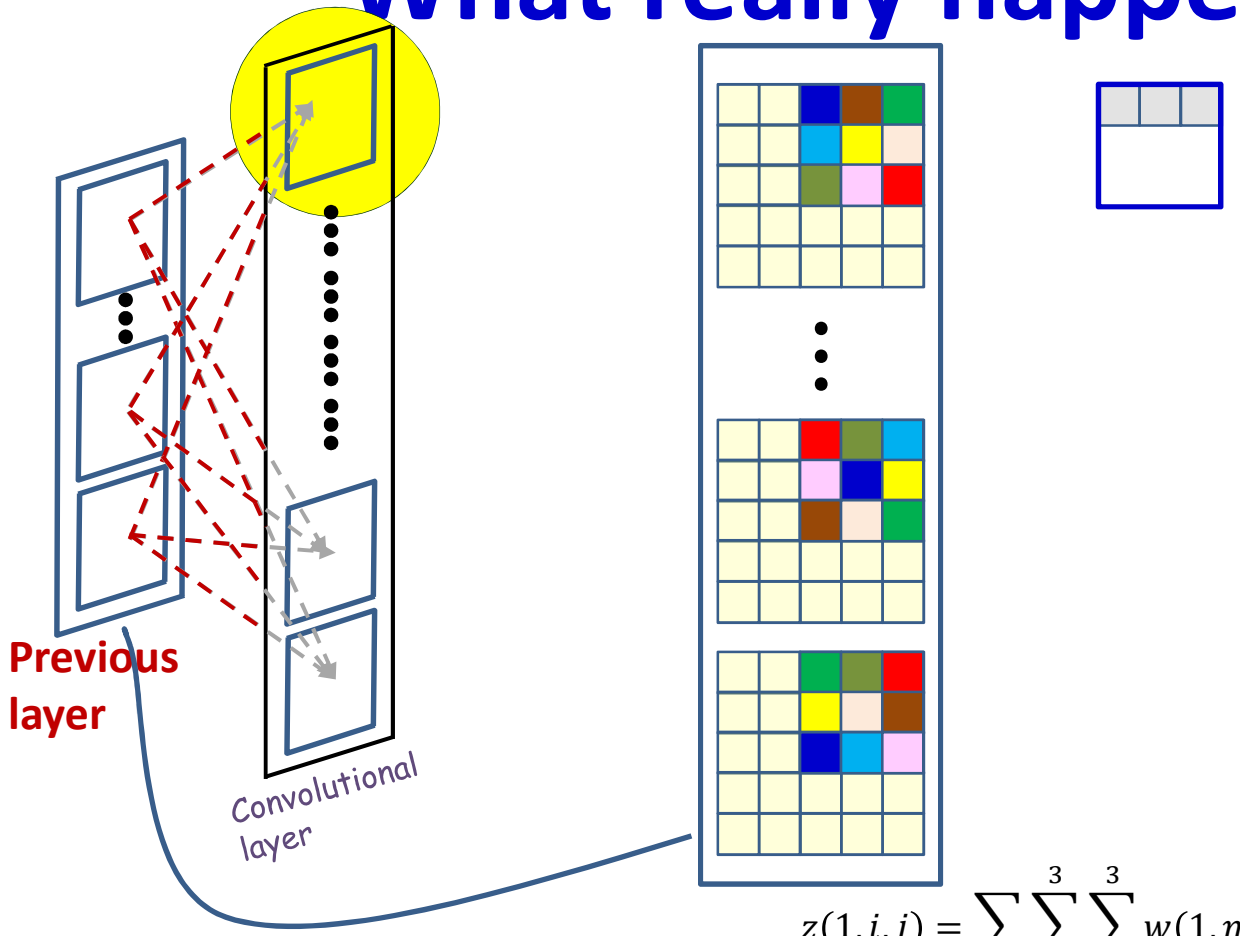


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- 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

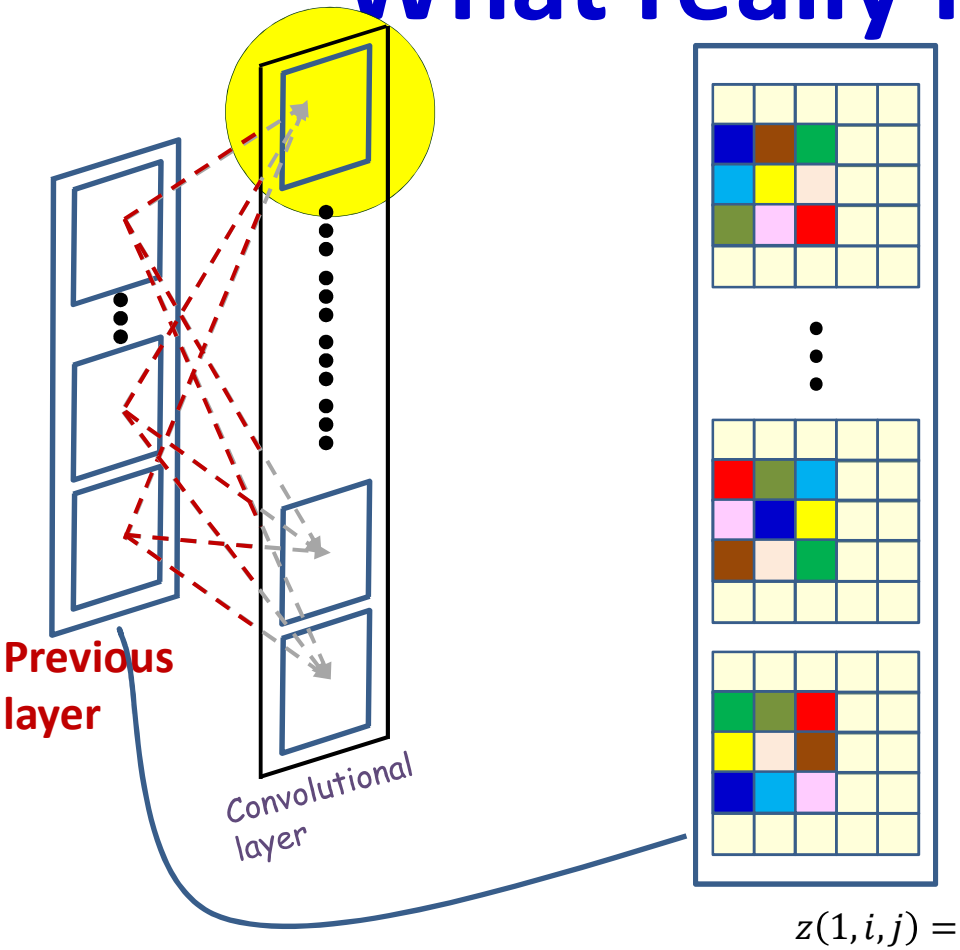


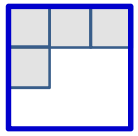
$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- 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



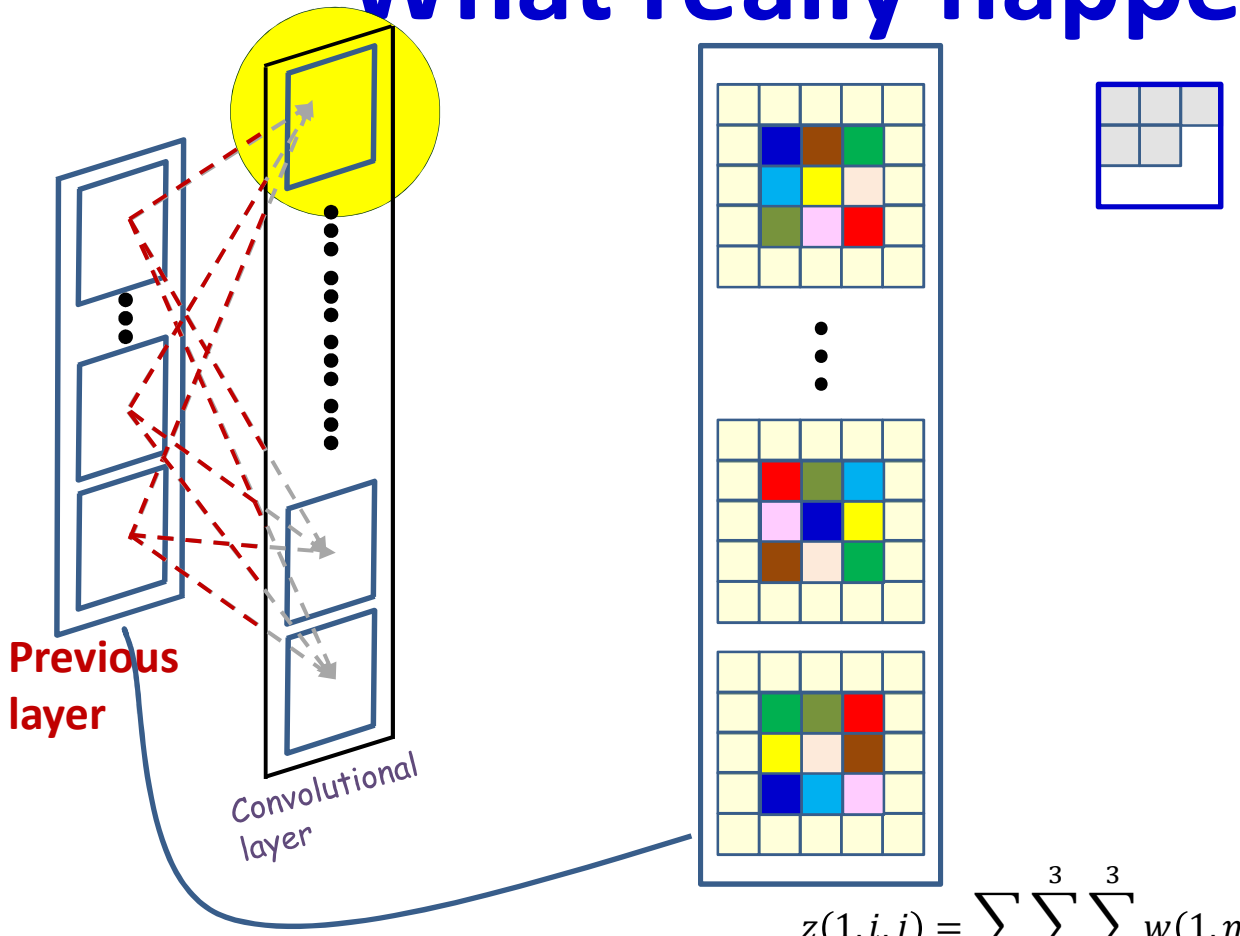


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- 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

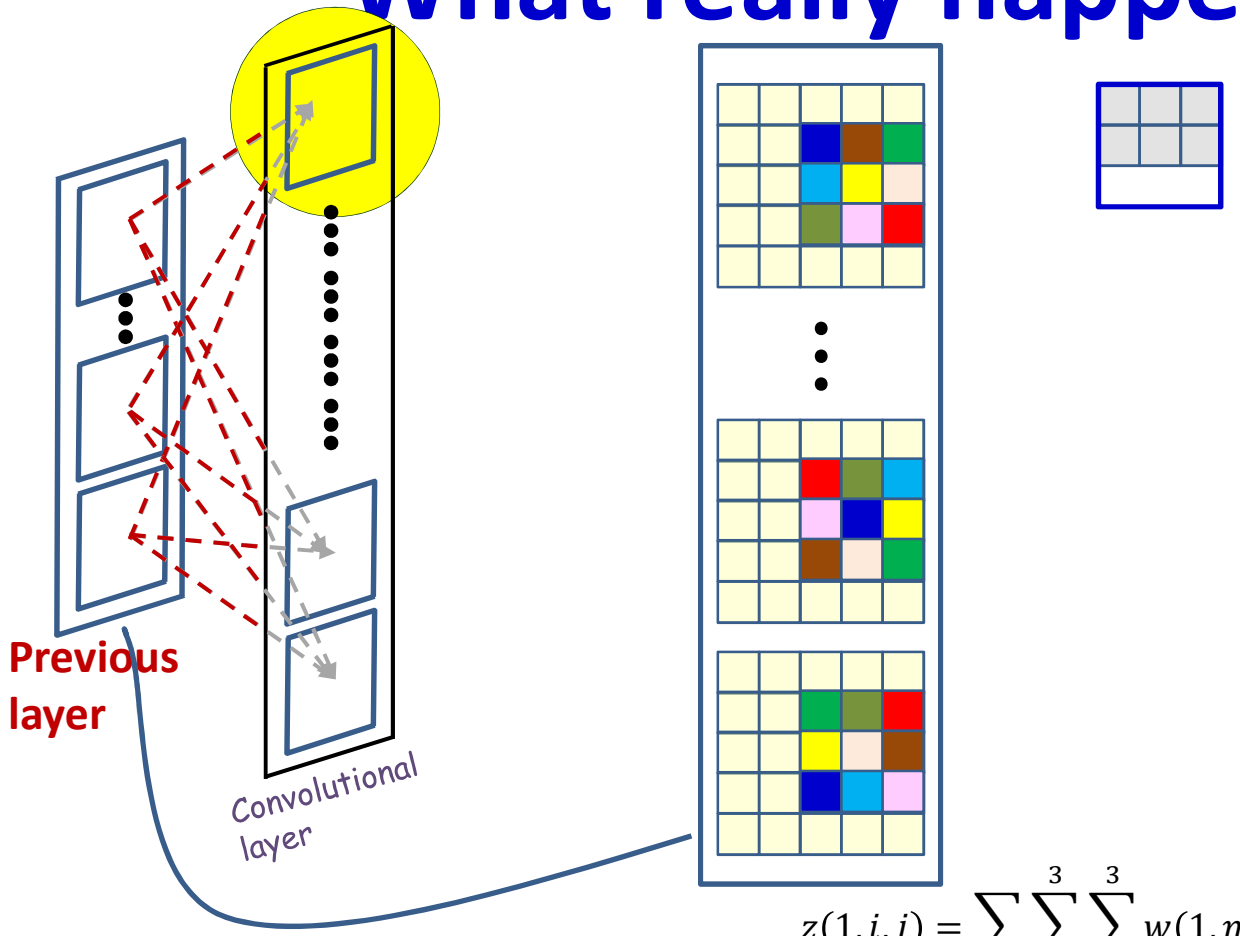


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- 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

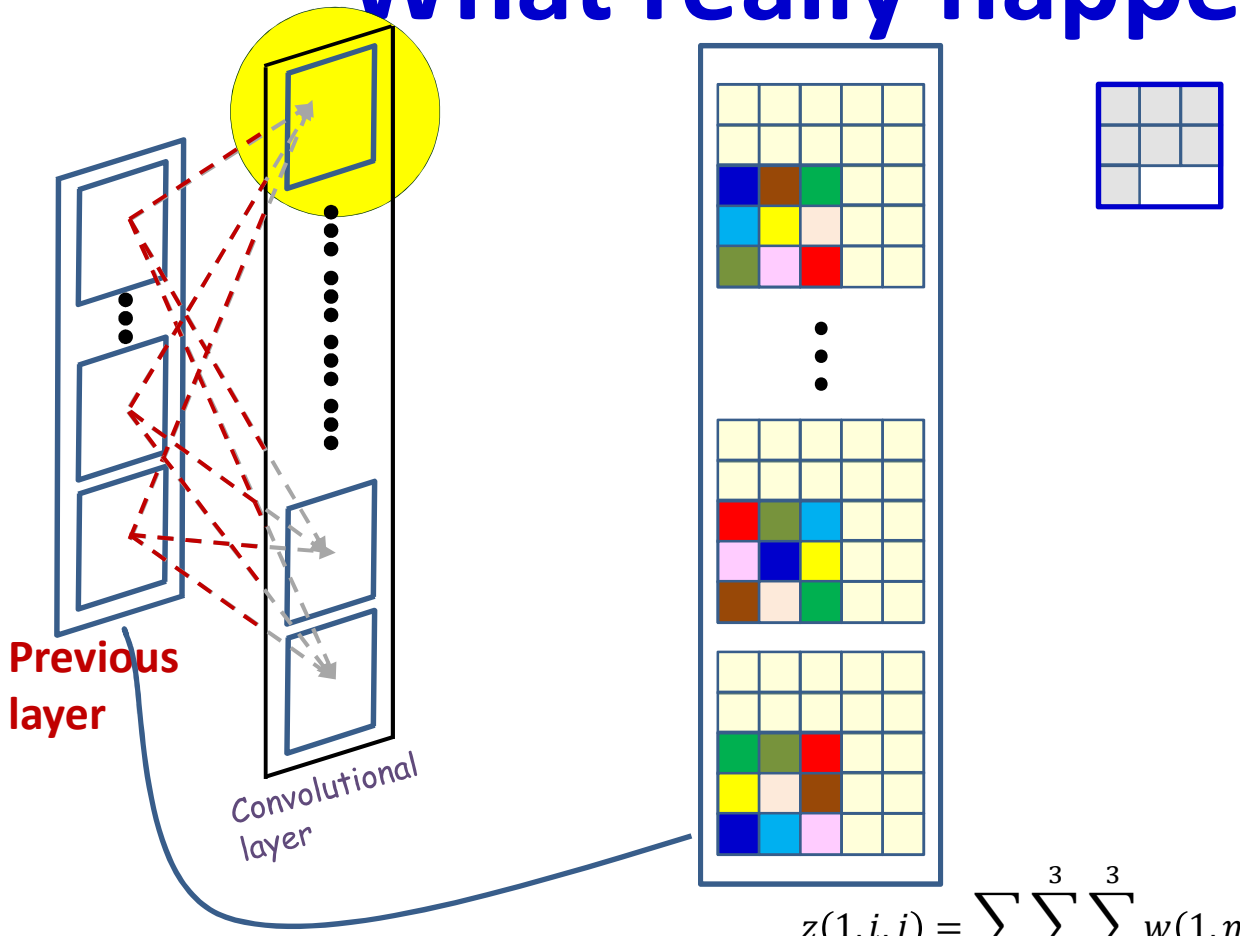


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- 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

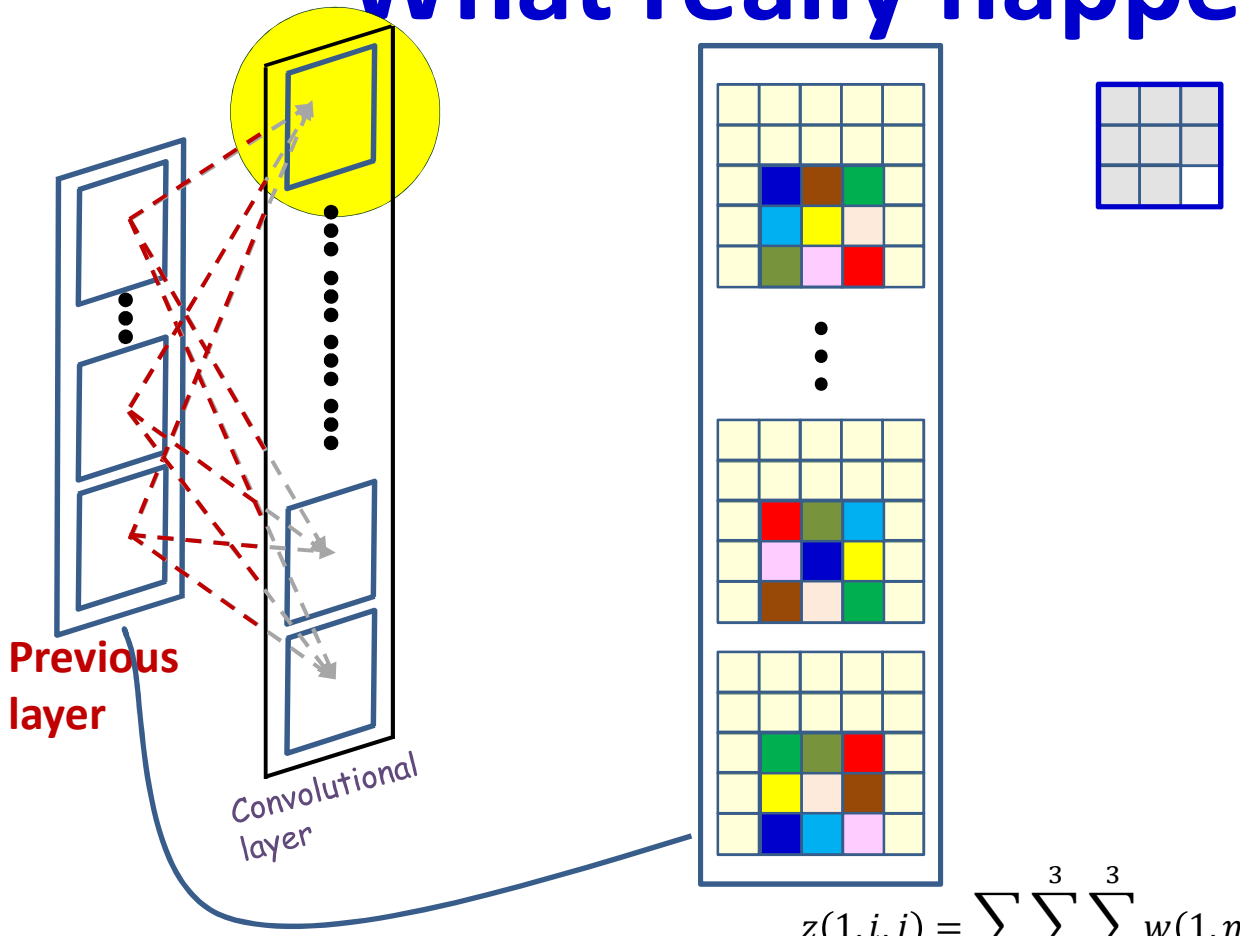


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- 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

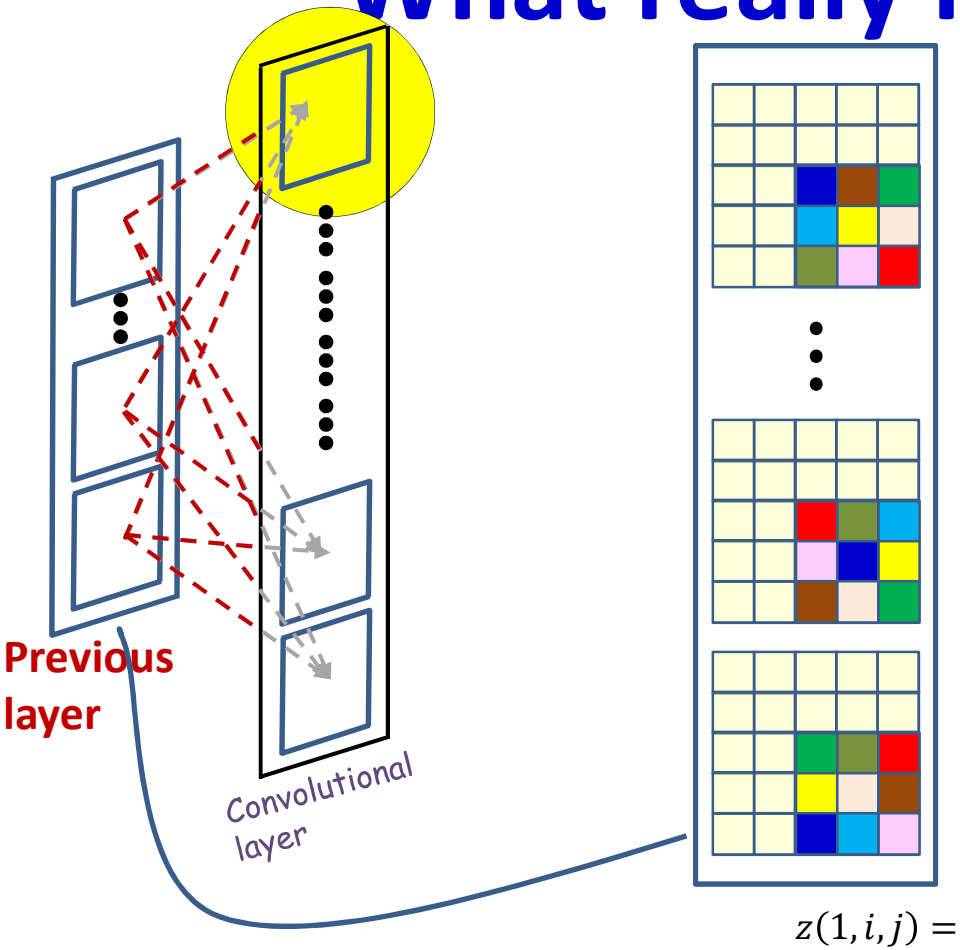


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- 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

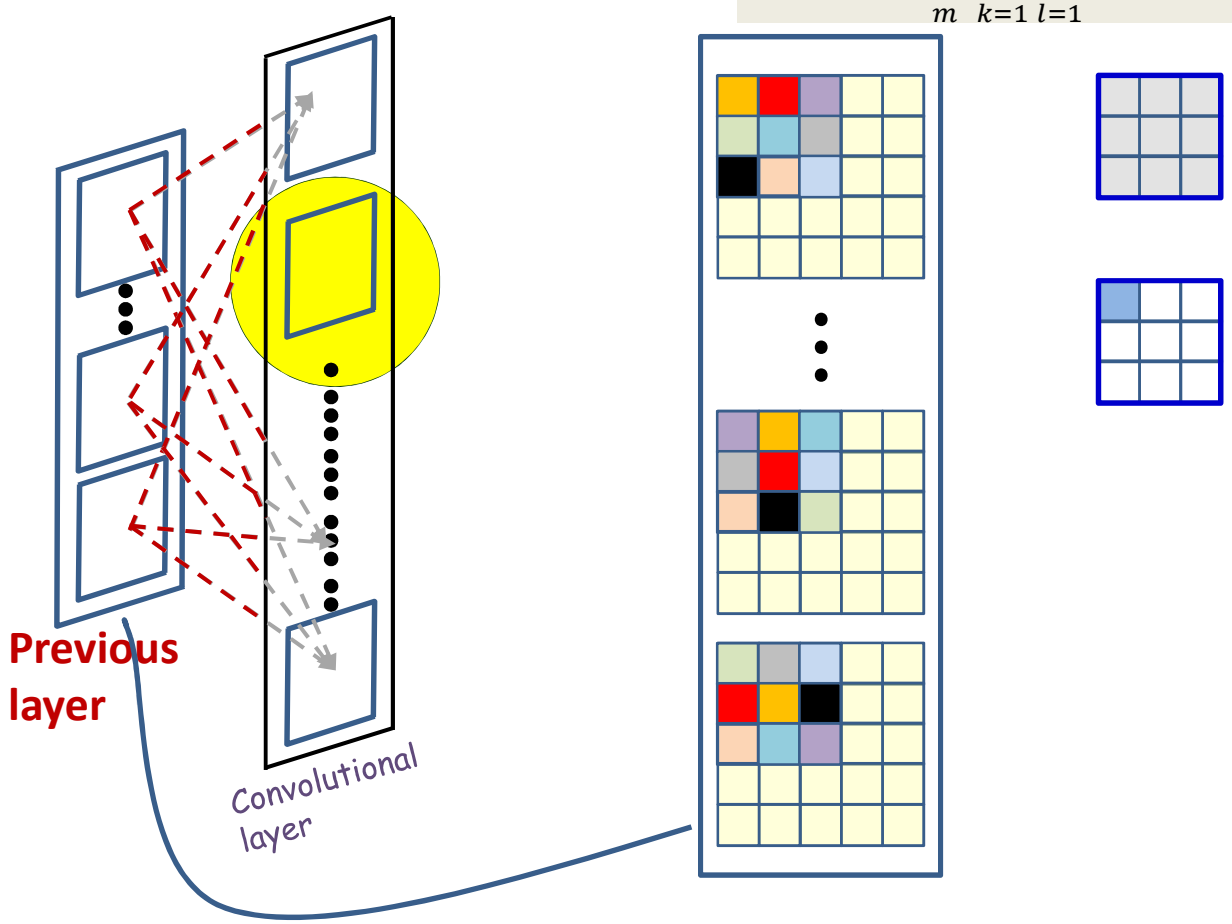


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- 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*

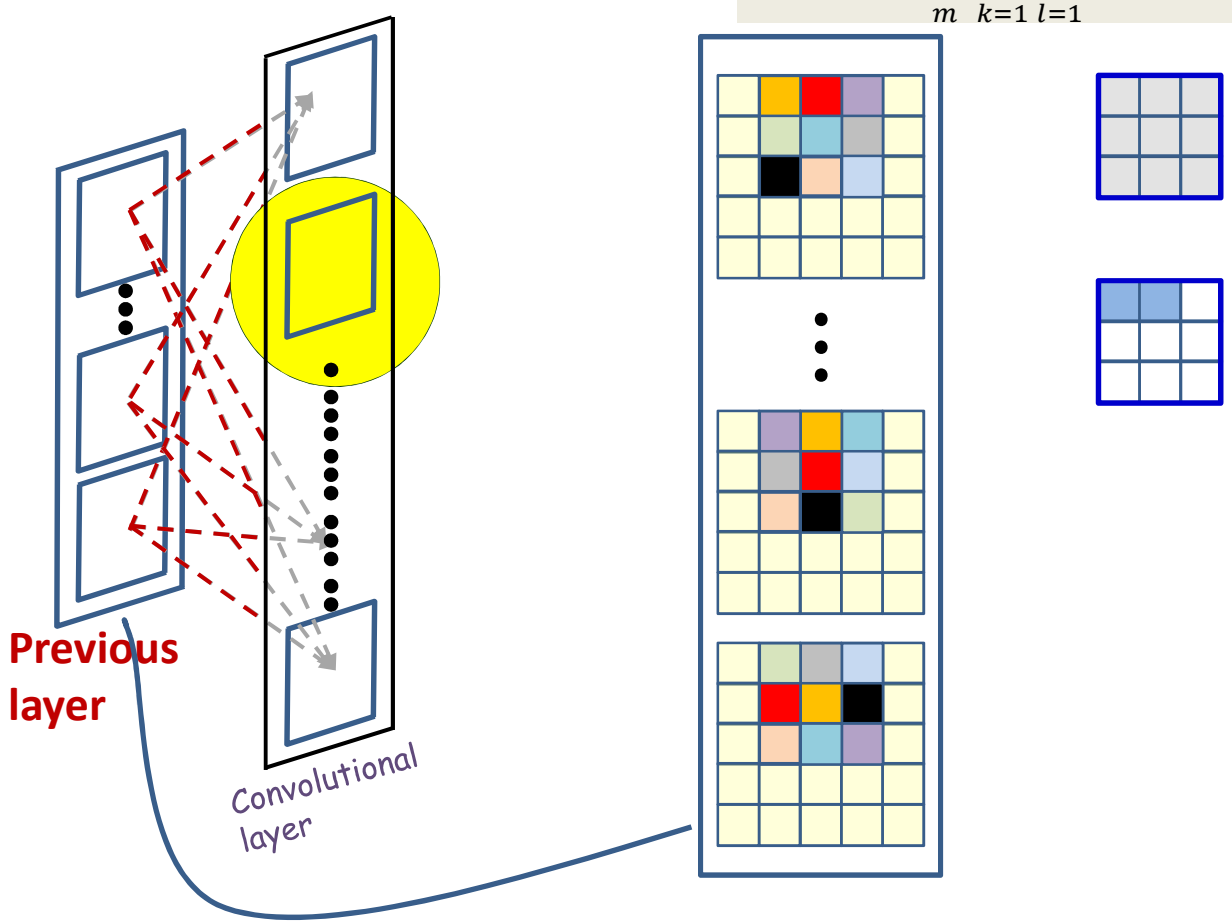
$$z(2,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(2,m,k,l) I(m, i+l-1, j+k-1) + b(2)$$



- 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*

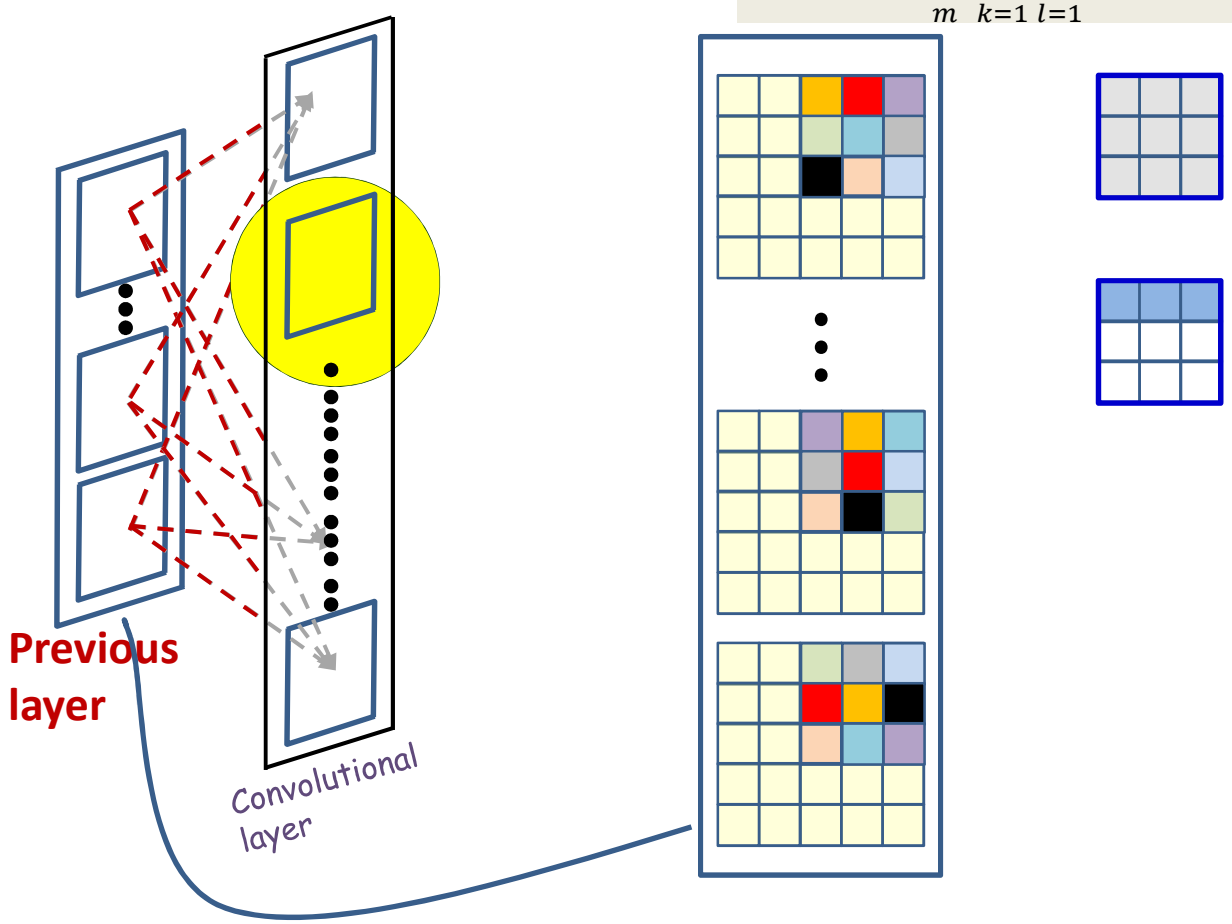
$$z(2,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(2,m,k,l) I(m, i+l-1, j+k-1) + b(2)$$



- 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*

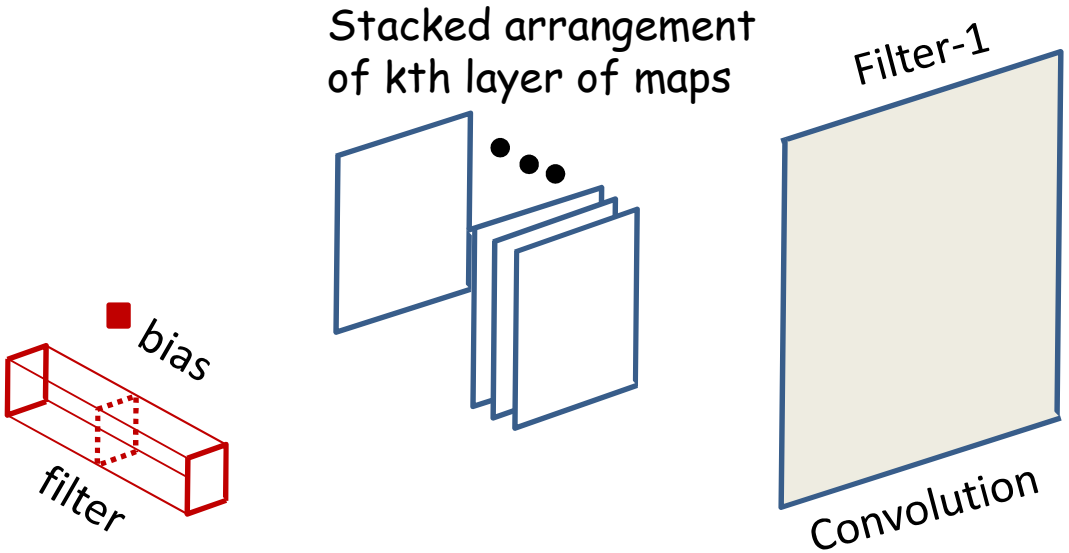
$$z(2,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(2,m,k,l) I(m, i+l-1, j+k-1) + b(2)$$



- 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*

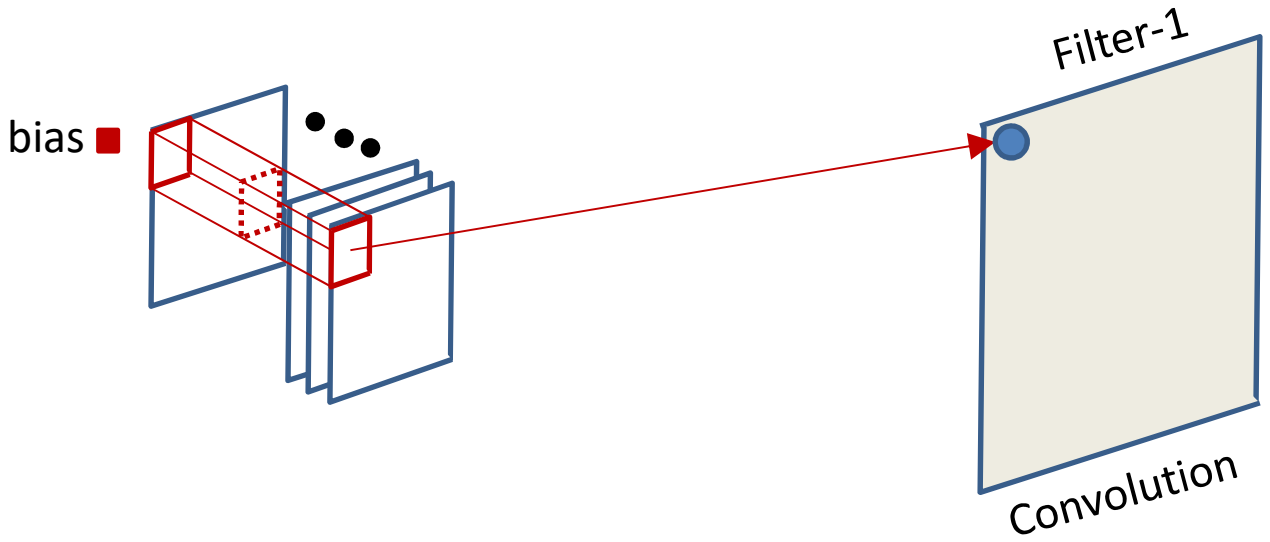
A different view



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

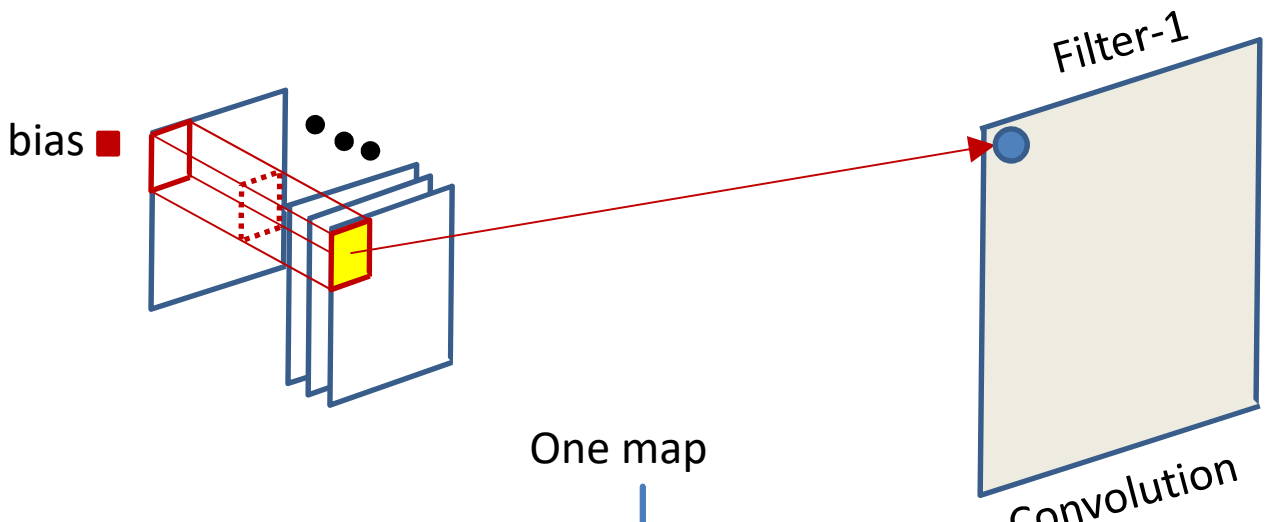
Extending to multiple input maps



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

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

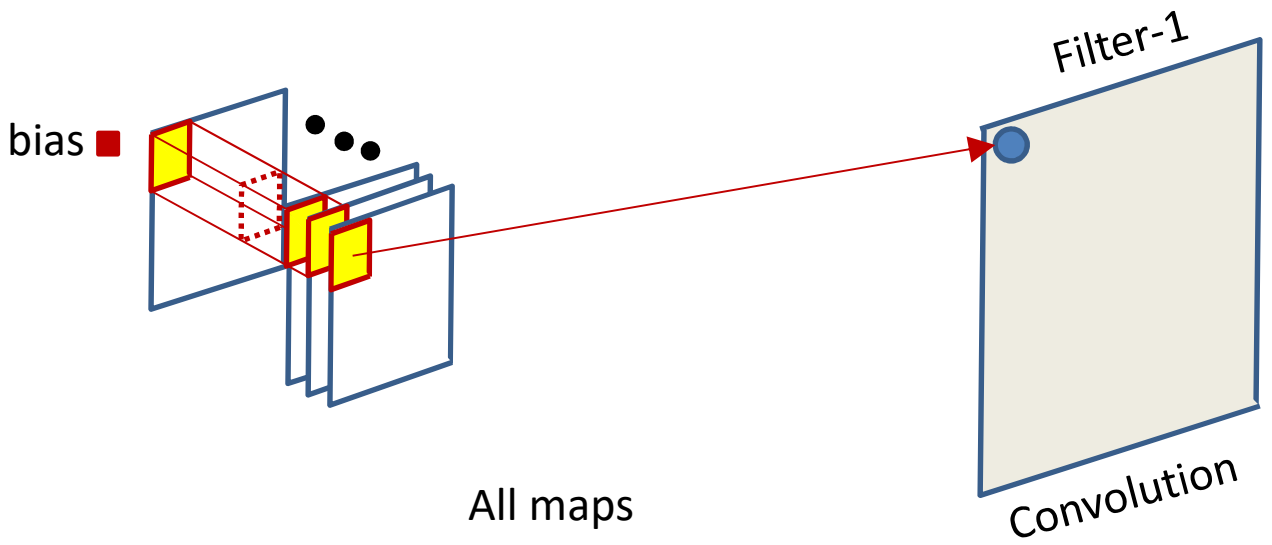
Extending to multiple input maps



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

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

Extending to multiple input maps

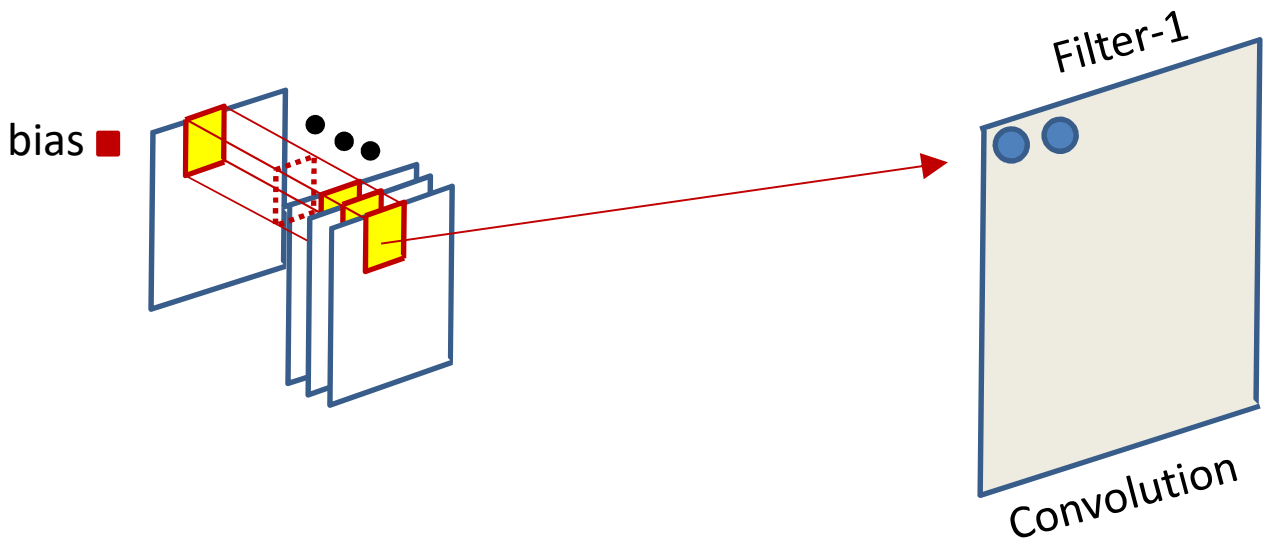


All maps

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

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

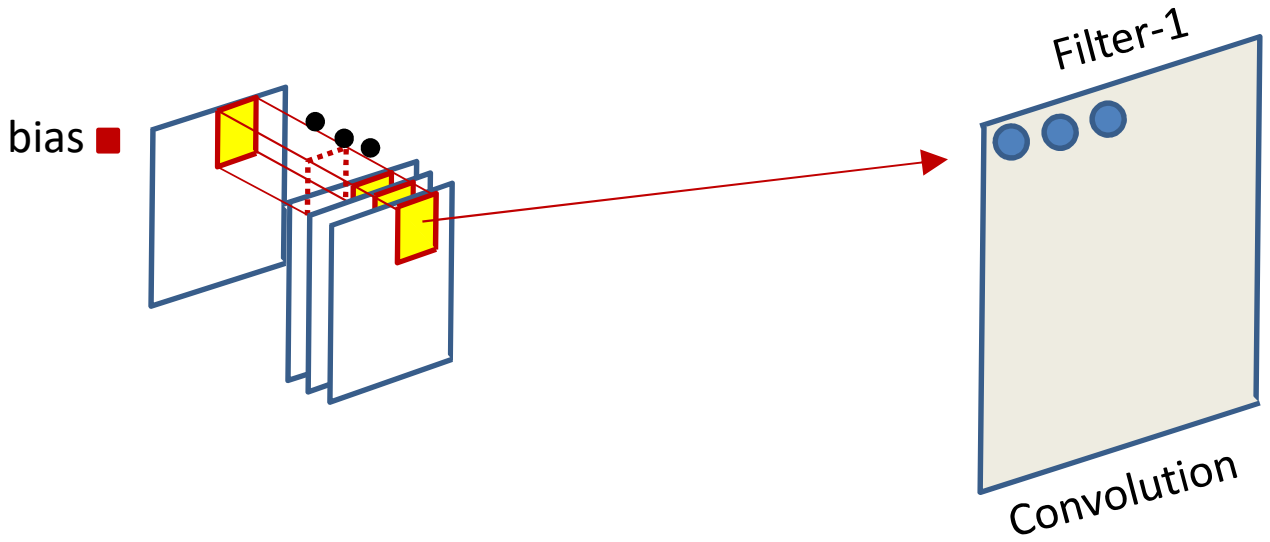
Extending to multiple input maps



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

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

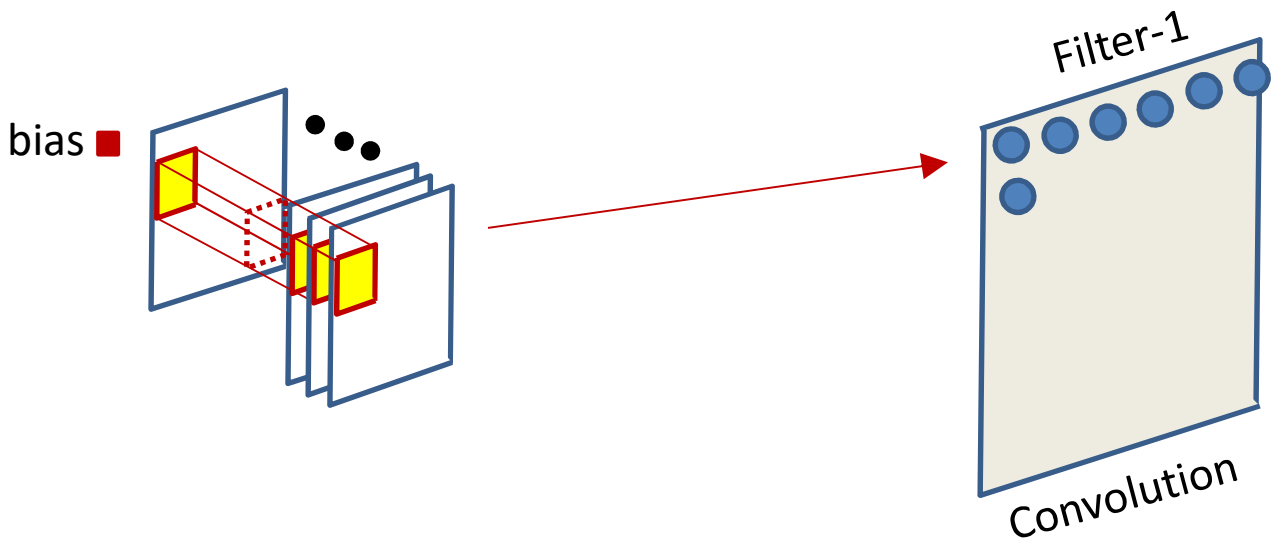
Extending to multiple input maps



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

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

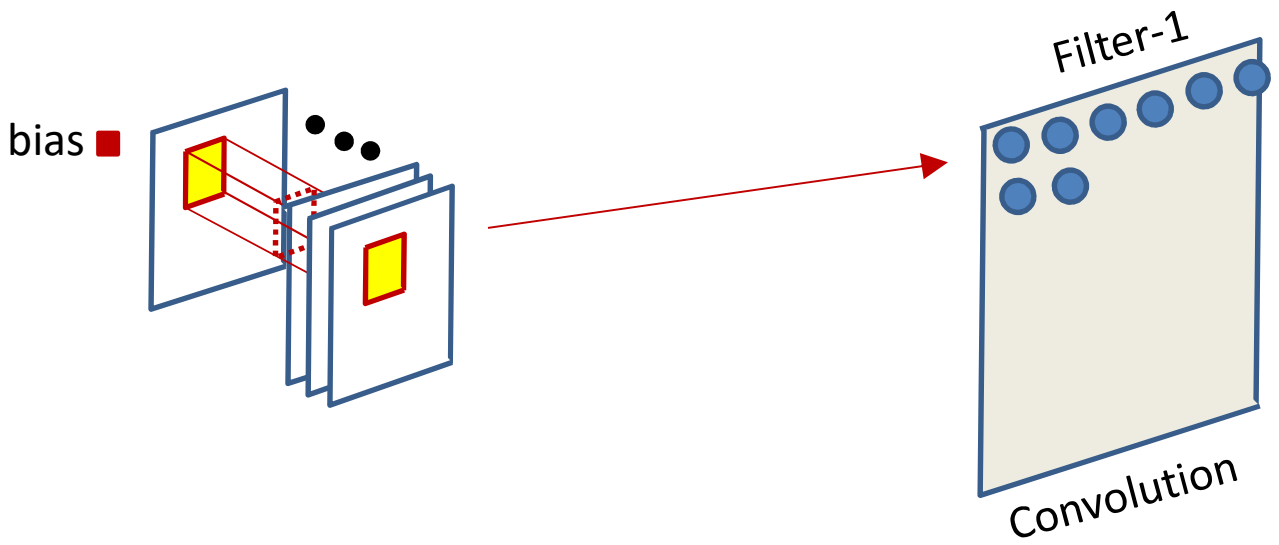
Extending to multiple input maps



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

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

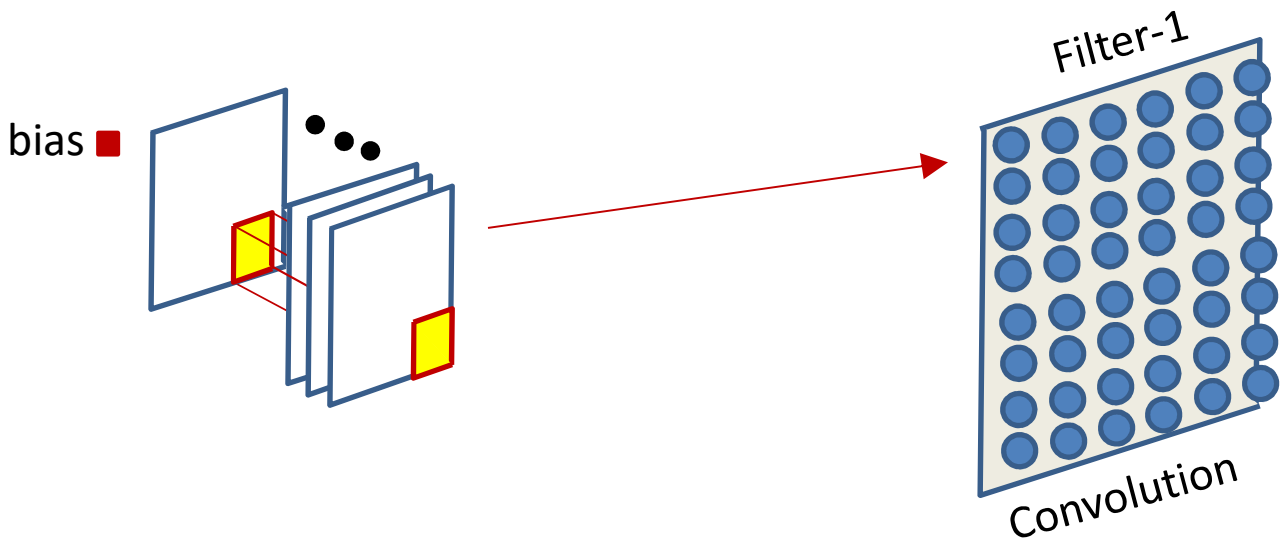
Extending to multiple input maps



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

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

Extending to multiple 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 convolutive map at any location *sums* the convolutive outputs *at all planes*

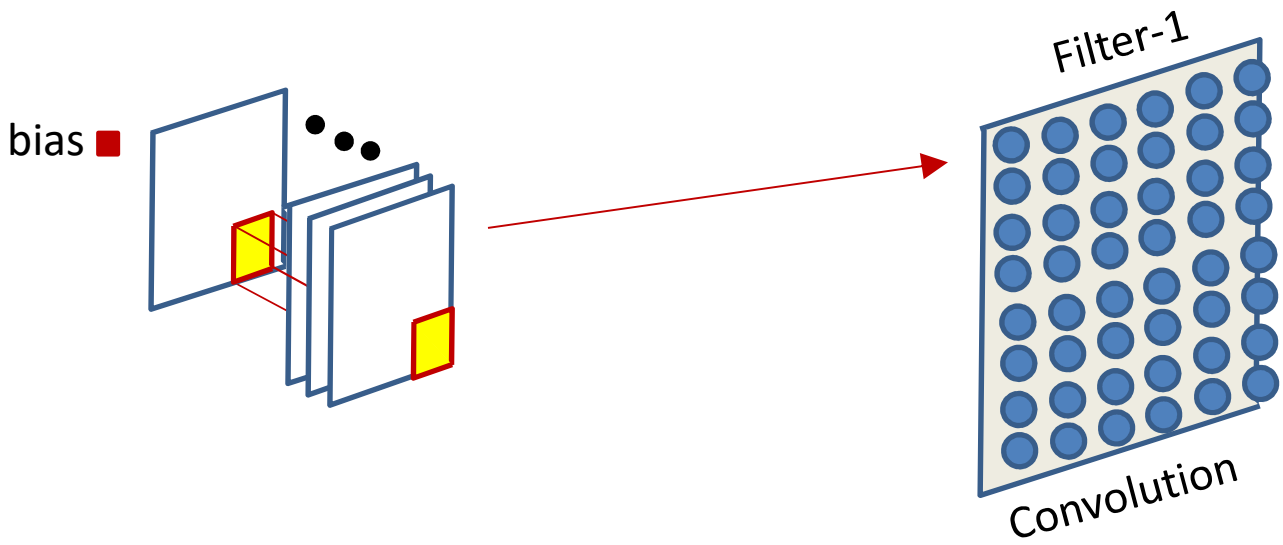
Convolutional neural net: Vector notation

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

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

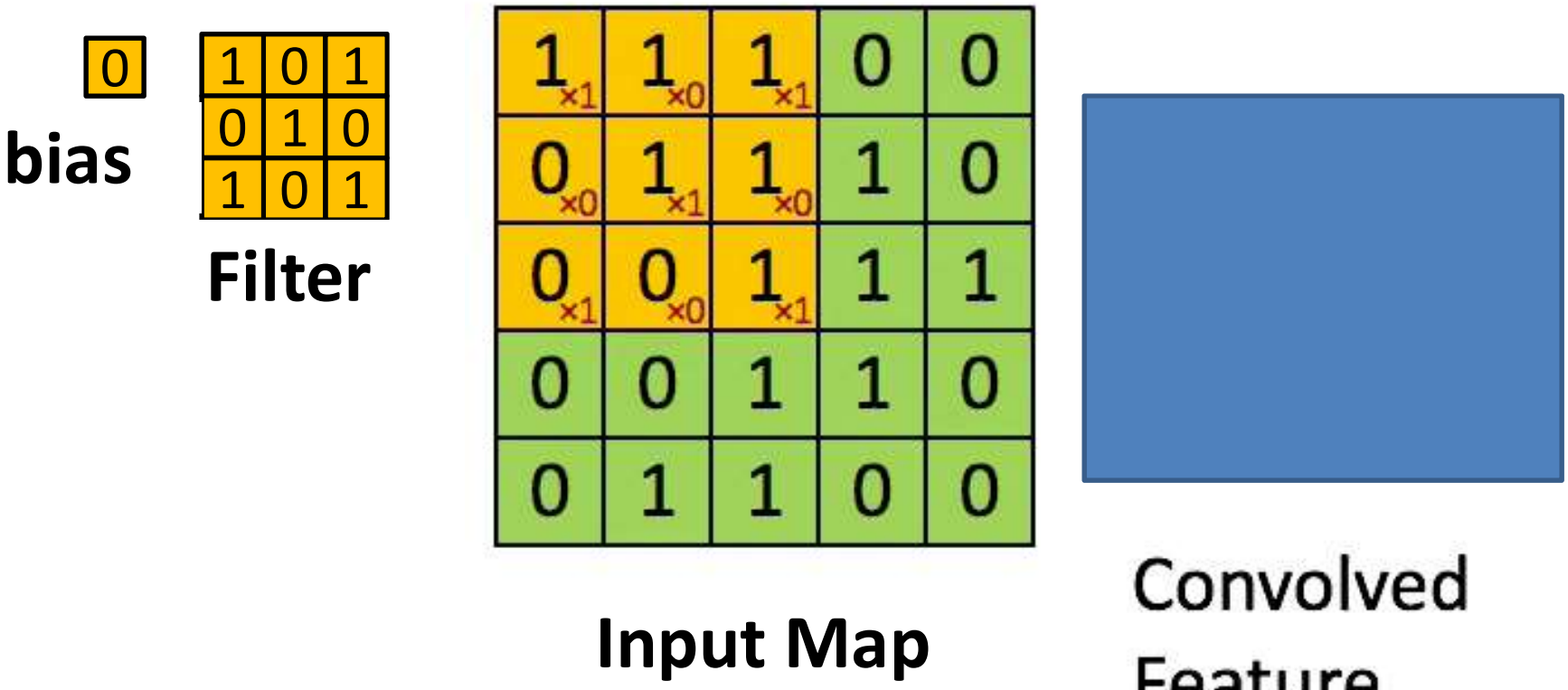
```
Y(0) = Image
for l = 1:L  # layers operate on vector at (x,y)
  for j = 1:D_l
      for x = 1:W_{l-1}-K_l+1
         for y = 1:H_{l-1}-K_l+1
            segment = Y(l-1,:,x:x+K_l-1,y:y+K_l-1) #3D tensor
            z(l,j,x,y) = W(l,j).segment #tensor inner prod.
            Y(l,j,x,y) = activation(z(l,j,x,y))

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

Engineering consideration: The size of the result of the convolution



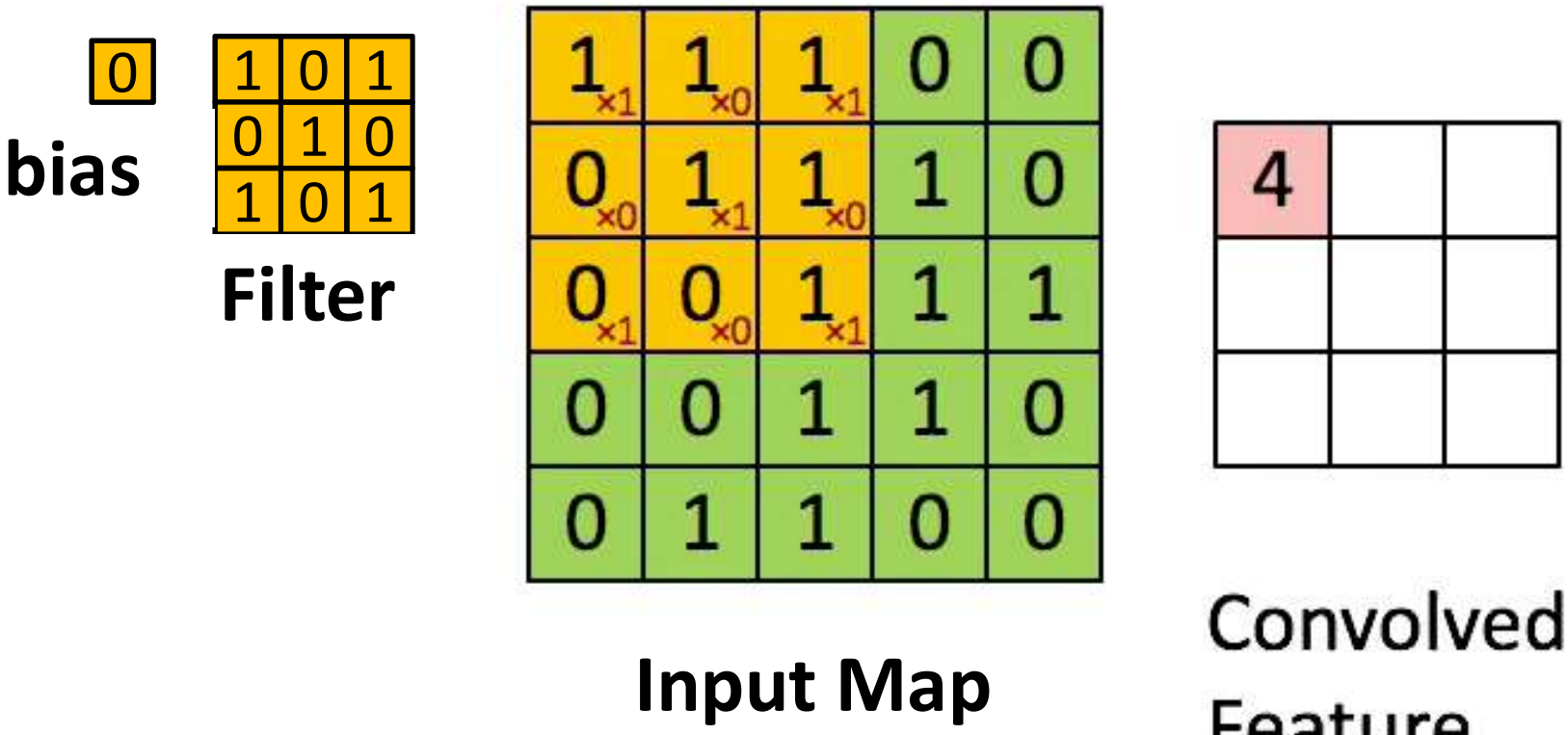
- **Recall**: the "stride" of the convolution may not be one pixel
  - I.e. the scanning neuron may "stride" more than one pixel at a time
- The size of the output of the convolution operation depends on implementation factors
  - And may not be identical to the size of the input
  - Lets take a brief look at this for completeness sake

The size of the convolution



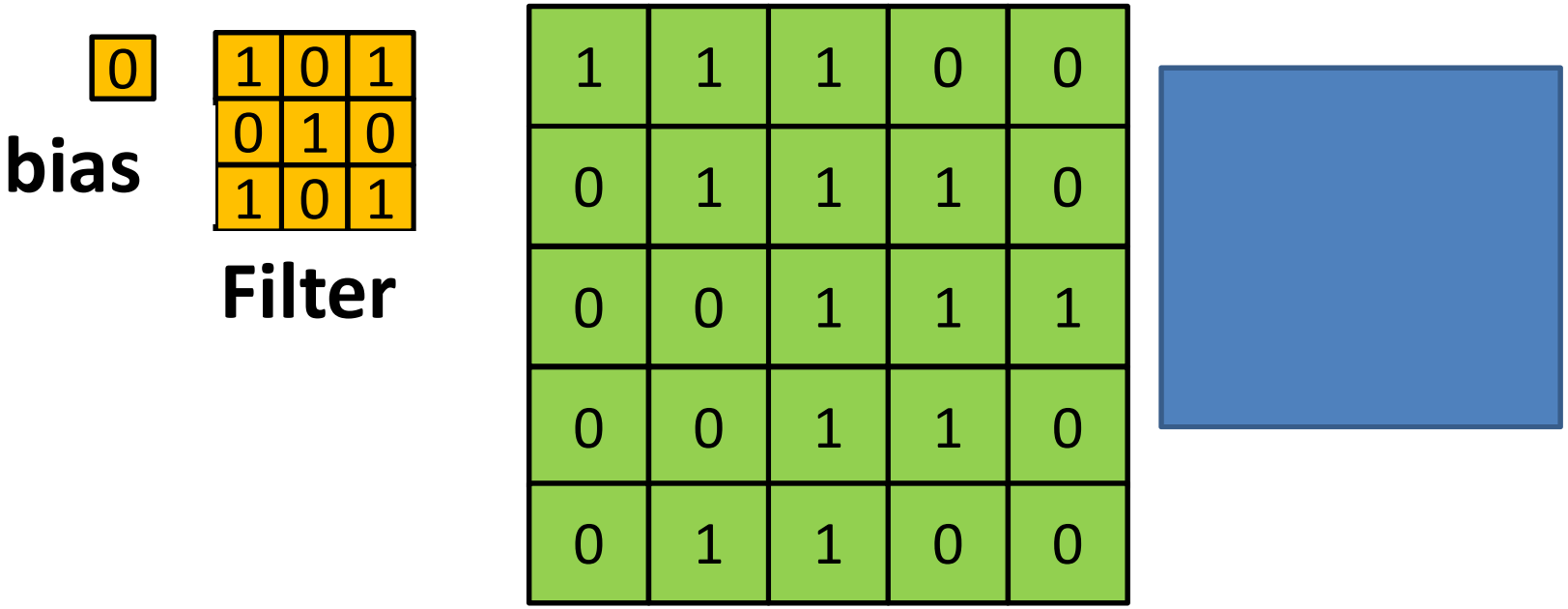
- Image size: 5x5
- Filter: 3x3
- “Stride”: 1
- Output size = ?

The size of the convolution



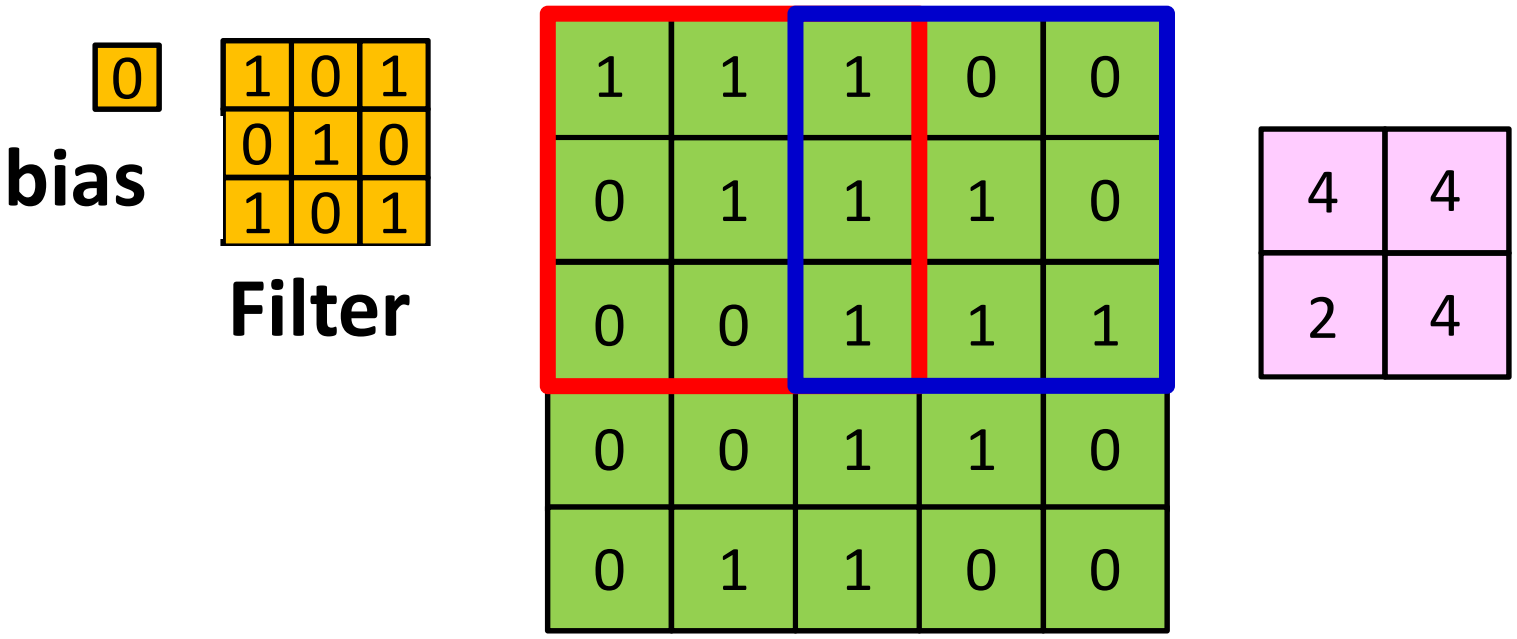
- Image size: 5x5
- Filter: 3x3
- Stride: 1
- Output size = ?

The size of the convolution



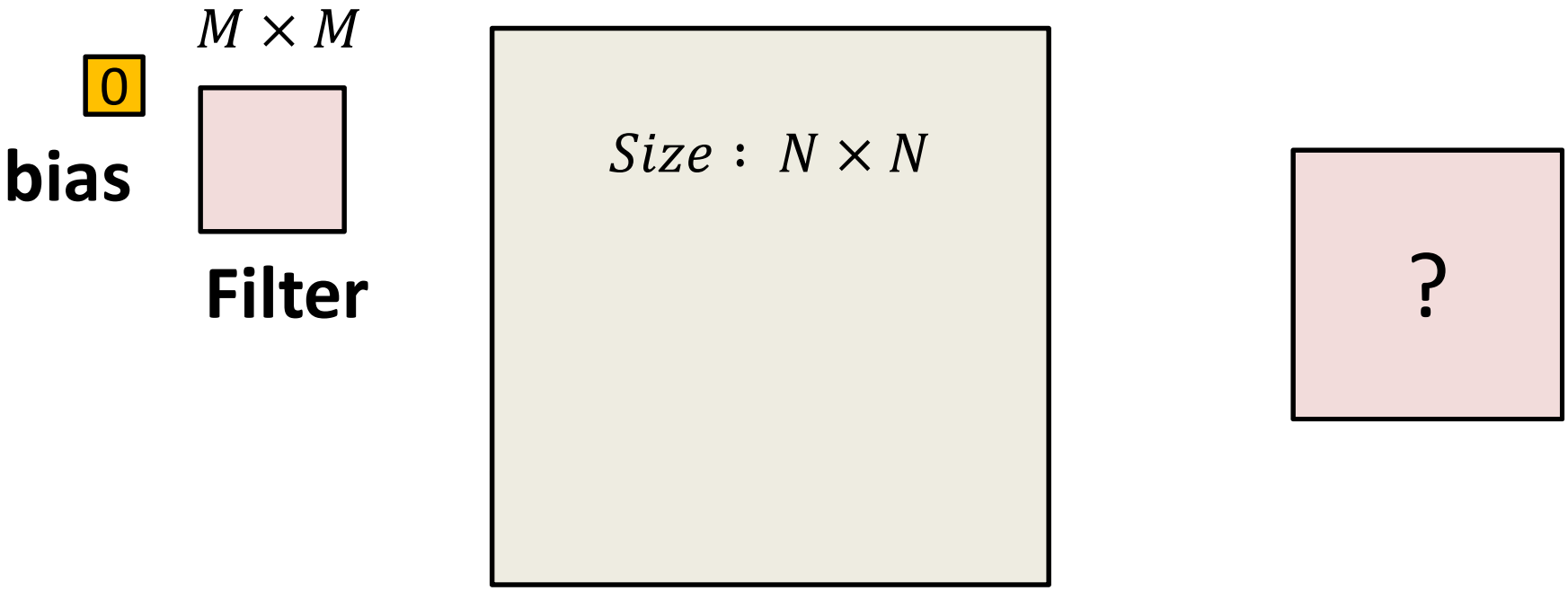
- Image size: 5x5
- Filter: 3x3
- Stride: 2
- Output size = ?

The size of the convolution



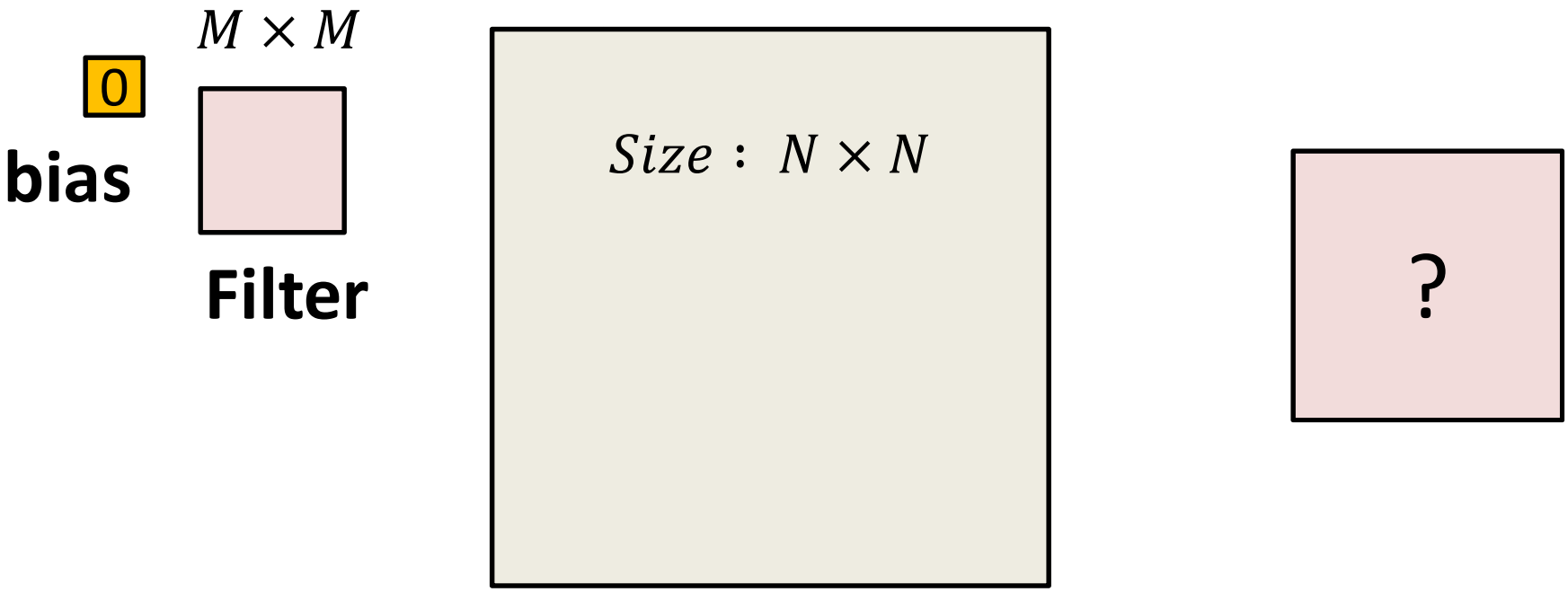
- Image size: 5x5
- Filter: 3x3
- Stride: 2
- Output size = ?

The size of the convolution



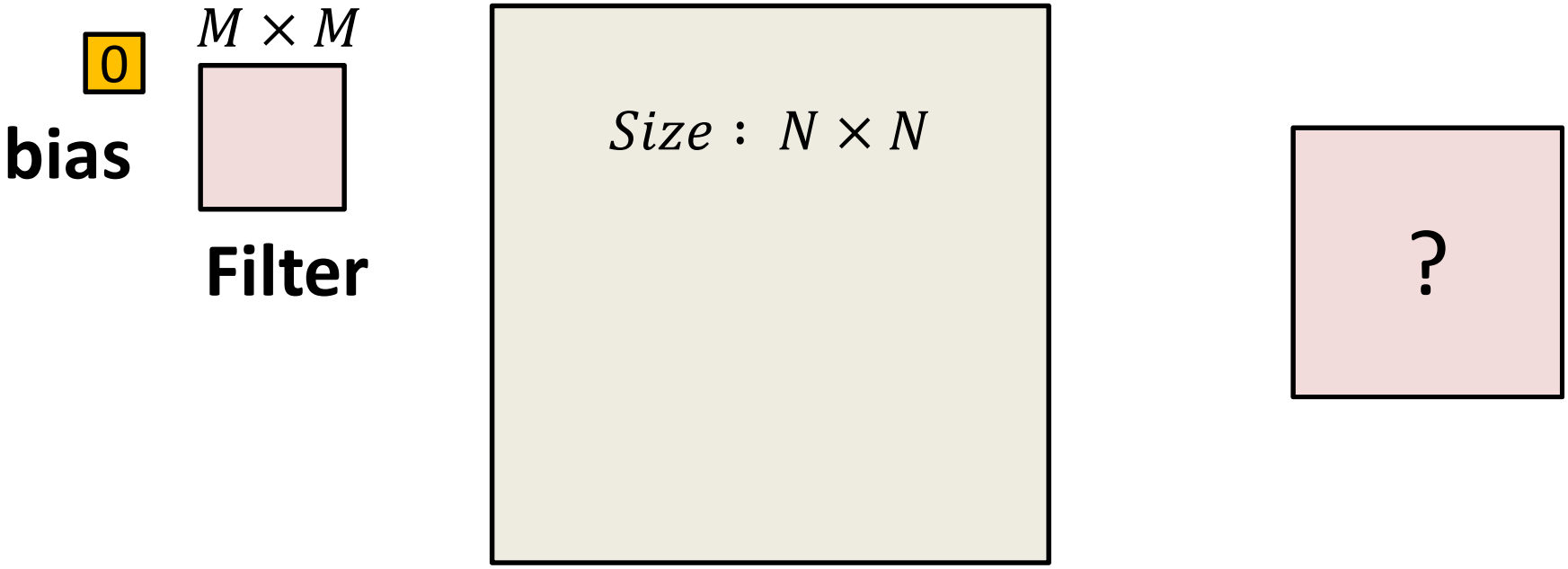
- Image size: $N \times N$
- Filter: $M \times M$
- Stride: 1
- Output size = ?

The size of the convolution



- Image size: $N \times N$
- Filter: $M \times M$
- Stride: S
- Output size = ?

The size of the convolution

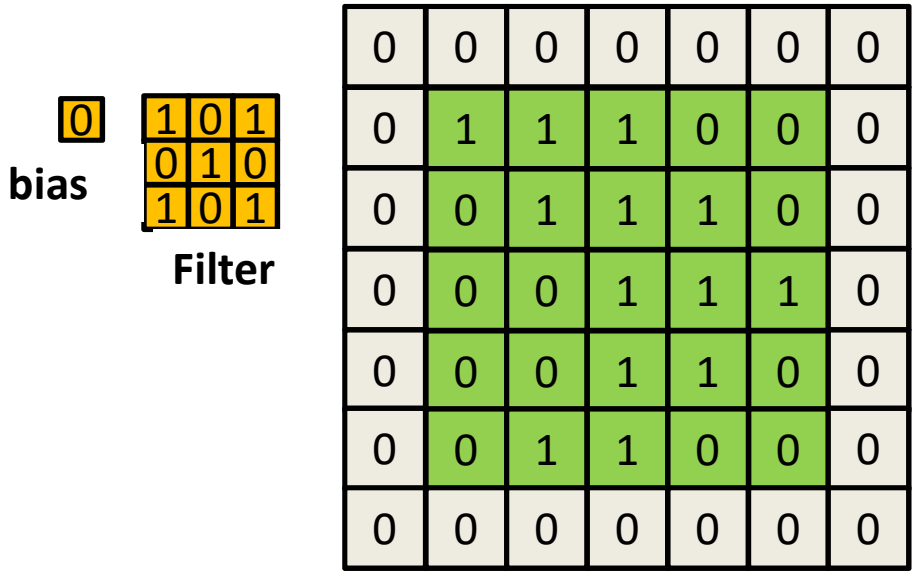


- Image size: $N \times N$
- Filter: $M \times M$
- Stride: S
- Output size (each side) = $\lfloor (N - M)/S \rfloor + 1$
  - Assuming you're not allowed to go beyond the edge of the input

Convolution Size

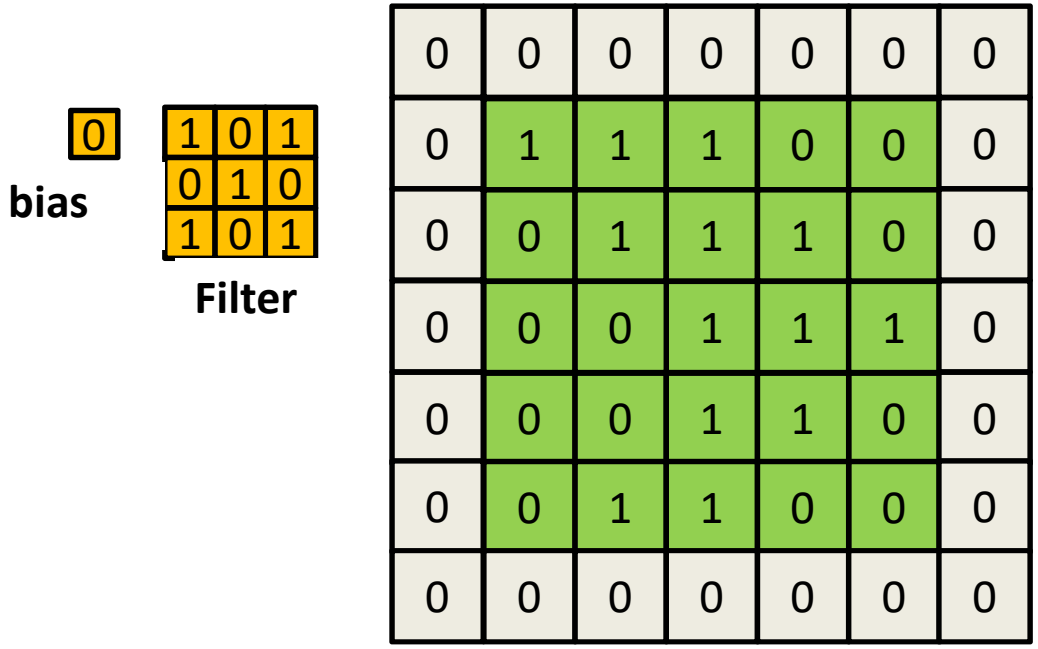
- Simple convolution size pattern:
  - Image size: $N \times N$
  - Filter: $M \times M$
  - Stride: S
  - **Output size (each side)** = $\lfloor (N - M)/S \rfloor + 1$
    - Assuming you're not allowed to go beyond the edge of the input
- Results in a reduction in the output size
  - Even if $S = 1$
  - Sometimes not considered acceptable
    - If there's no active downsampling, through max pooling and/or $S > 1$, then the output map should ideally be the same size as the input

Solution



- Zero-pad the input
  - Pad the input image/map all around
    - Add P_L rows of zeros on the left and P_R rows of zeros on the right
    - Add P_L rows of zeros on the top and P_L rows of zeros at the bottom
  - P_L and P_R chosen such that:
    - $P_L = P_R$ OR $|P_L - P_R| = 1$
    - $P_L + P_R = M-1$
      - **For stride 1, the result of the convolution is the same size as the original image**

Solution



- Zero-pad the input
  - Pad the input image/map all around
  - Pad as symmetrically as possible, such that..
  - **For stride 1, the result of the convolution is the same size as the original image**

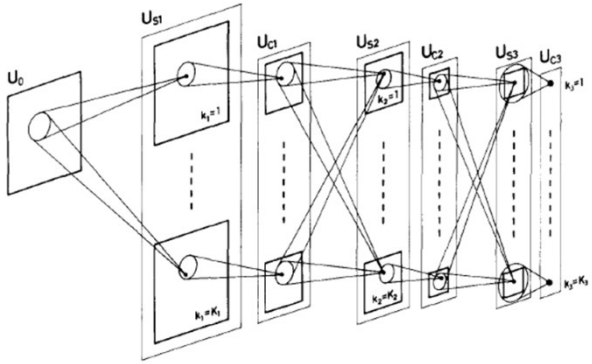
Zero padding

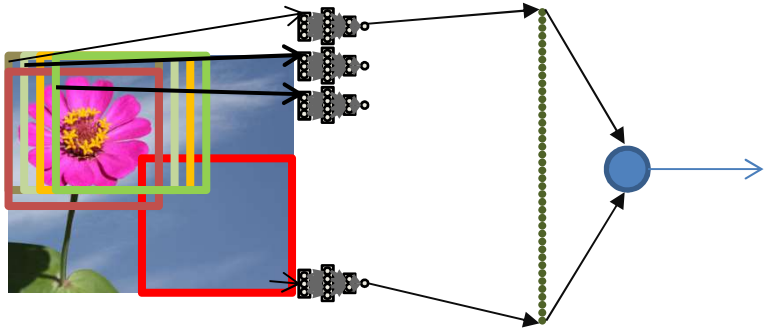
- For an L width filter:
  - Odd L : Pad on both left and right with $(L-1)/2$ columns of zeros
  - Even L : Pad one side with$L/2$ columns of zeros, and the other with $\frac{L}{2}-1$ columns of zeros
  - The resulting image is width $N+L-1$
  - The result of the convolution is width N

- The top/bottom zero padding follows the same rules to maintain map height after convolution

- For hop size $S>1$, zero padding is adjusted to ensure that the size of the convolved output is $\lceil N/S \rceil$
  - Achieved by *first* zero padding the image with $S\lceil N/S \rceil - N$ columns/rows of zeros and then applying above rules

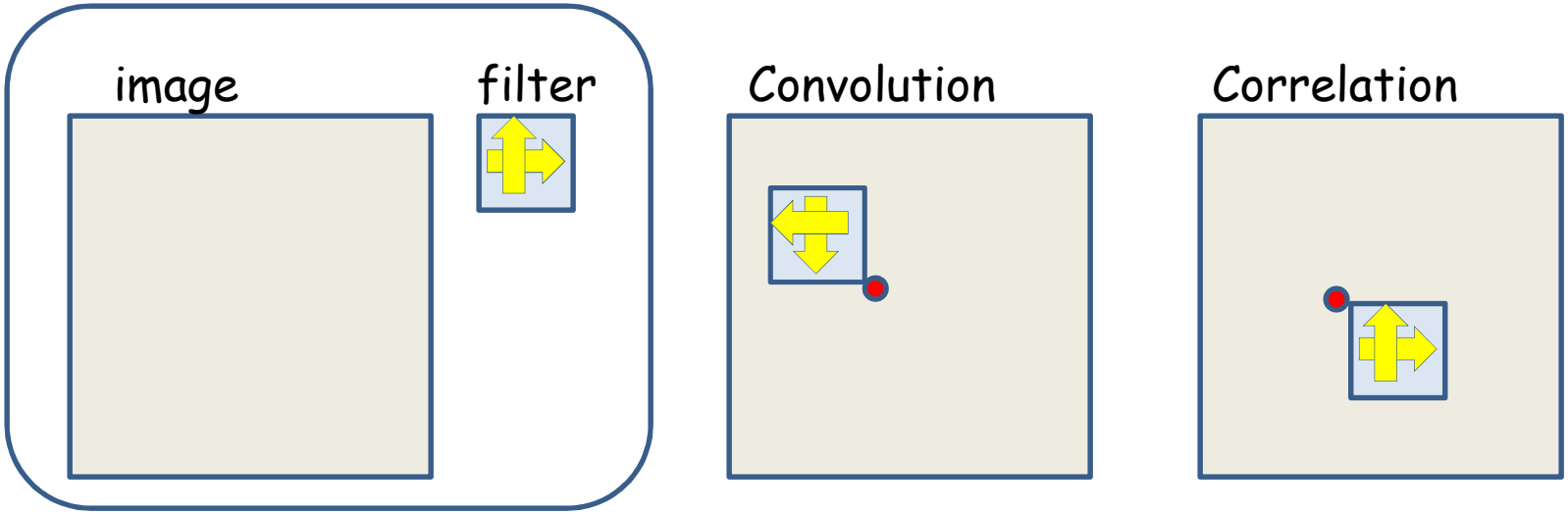
Why convolution?





- Convolutional neural networks are, in fact, equivalent to *scanning* with an MLP
  - Just run the entire MLP on each block separately, and combine results
    - As opposed to scanning (convolving) the picture with individual neurons/filters
  - Even computationally, the number of operations in both computations is identical
    - The neocognitron in fact views it equivalently to a scan
- So why convolutions?

Correlation, not Convolution



- The operation performed is technically a correlation, not a convolution
- **Correlation**:

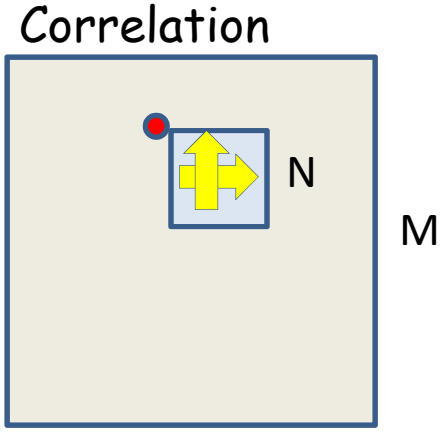
$$y(i,j) = \sum_l \sum_m x(i+l, j+m) w(l,m)$$

  - Shift the "filter" w to "look" at the input x block *beginning* at (i,j)
- **Convolution**:

$$y(i,j) = \sum_l \sum_m x(i-l, j-m) w(l,m)$$

- Effectively "flip" the filter, right to left, top to bottom

Cost of Correlation

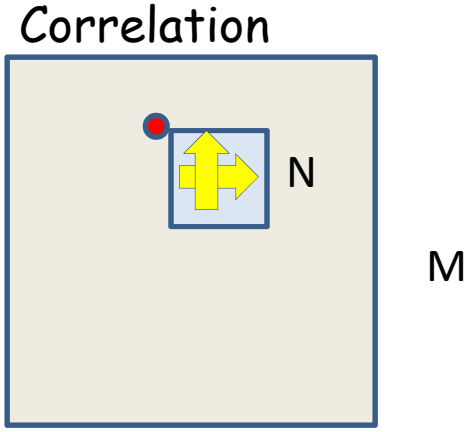


- **Correlation**:

$$y(i,j) = \sum_{l} \sum_{m} x(i+l, j+m) w(l,m)$$

- Cost of scanning an $M \times M$ image with an $N \times N$ filter: $\mathrm{O}(M^2 N^2)$
  - N^2 multiplications at each of M^2 positions
    - Not counting boundary effects
  - Expensive, for large filters

Correlation in Transform Domain



- **Correlation usind DFTs**:

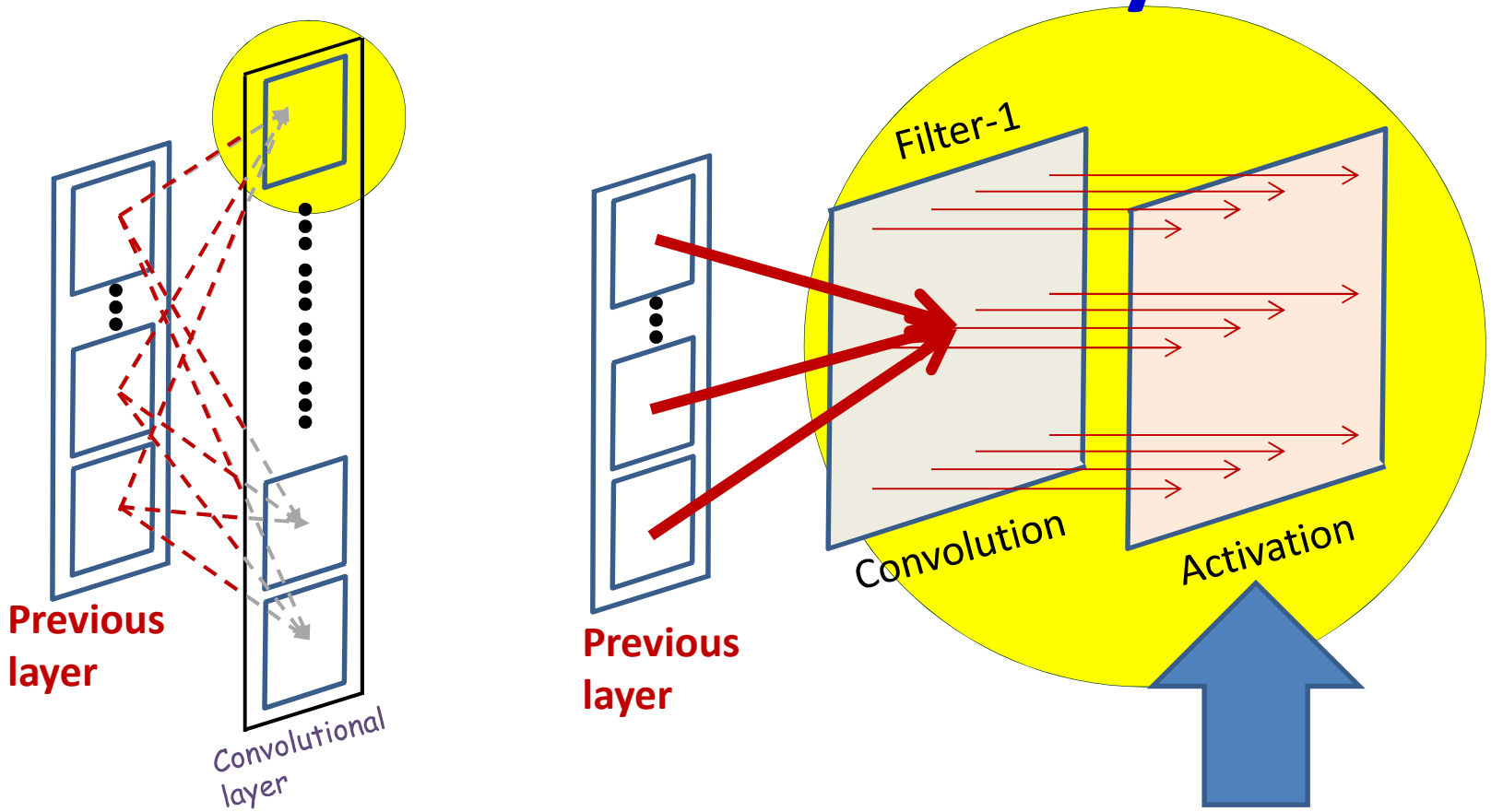
$$\mathrm{Y} = IDFT2\big(DFT2(X) \circ conj(DFT2(W))\big)$$

- Cost of doing this using the Fast Fourier Transform to compute the DFTs: $\mathrm{O}(M^2 log N)$
  - Significant saving for large filters
  - Or if there are many filters

Returning to our problem

- … From the world of size engineering …

A convolutional layer



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

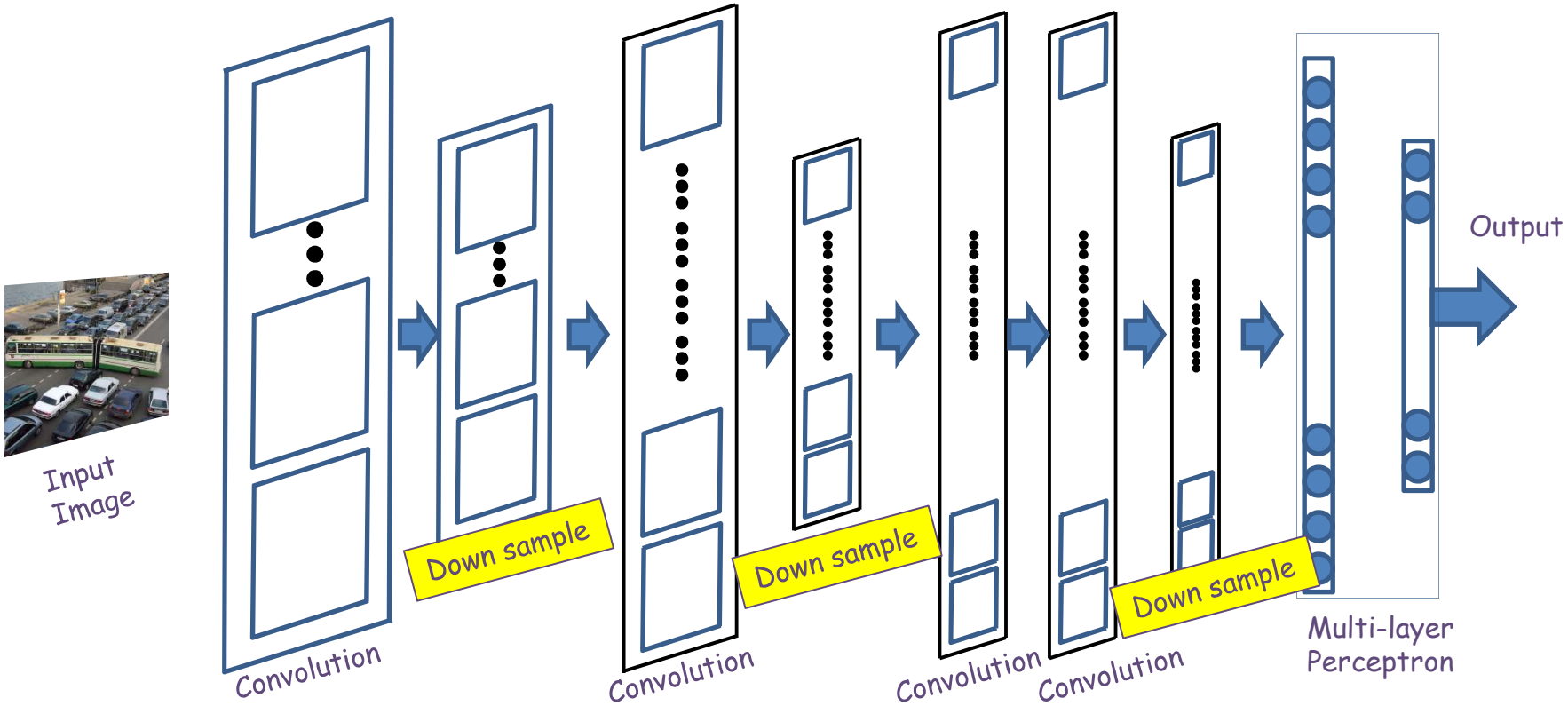
Convolutional neural net:

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

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

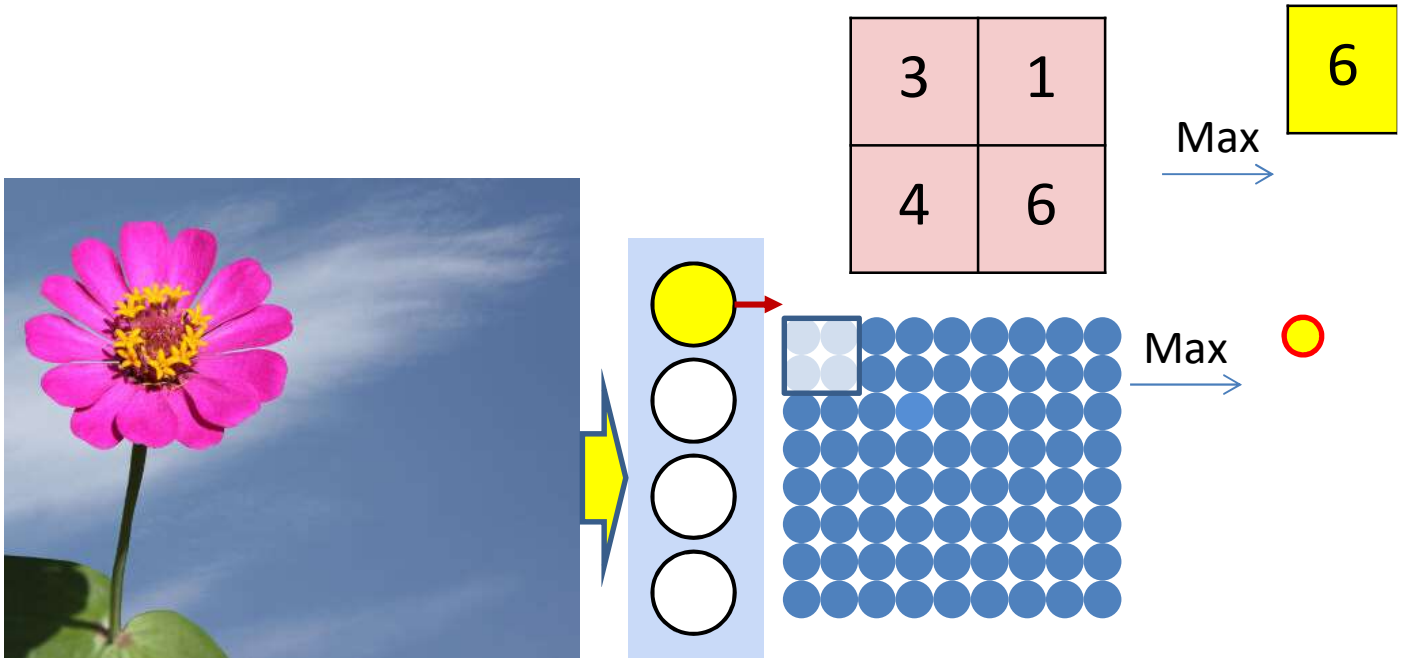
```
Y(0) = Image
for l = 1:L  # layers operate on vector at (x,y)
  for j = 1:D_l
     for x = 1:W_{l-1}-K_l+1
        for y = 1:H_{l-1}-K_l+1
          segment = Y(l-1,:,x:x+K_l-1,y:y+K_l-1) #3D tensor
          z(l,j,x,y) = W(l,j).segment #tensor inner prod.
          Y(l,j,x,y) = activation(z(l,j,x,y))

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

The other component
Downsampling/Pooling



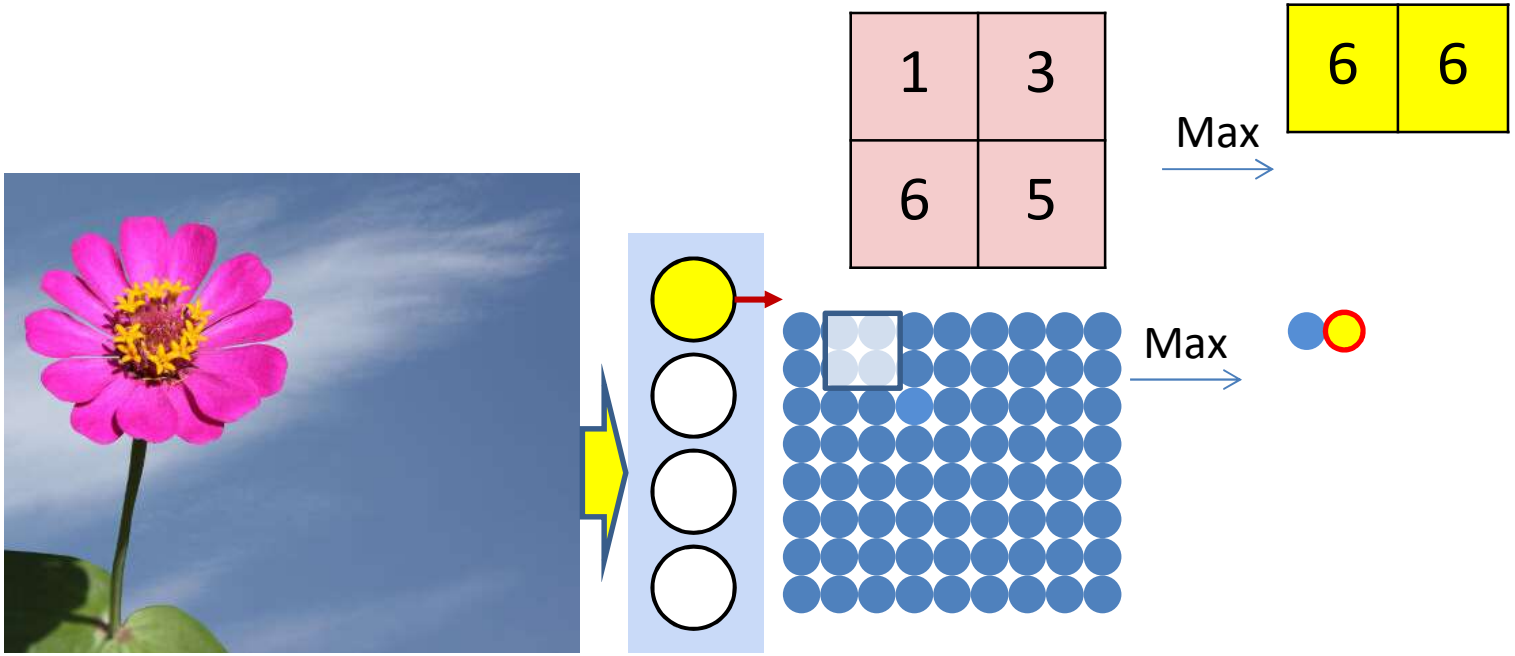
- Convolution (and activation) layers are followed intermittently by "downsampling" (or "pooling") layers
  - Often, they alternate with convolution, though this is not necessary

Recall: Max pooling



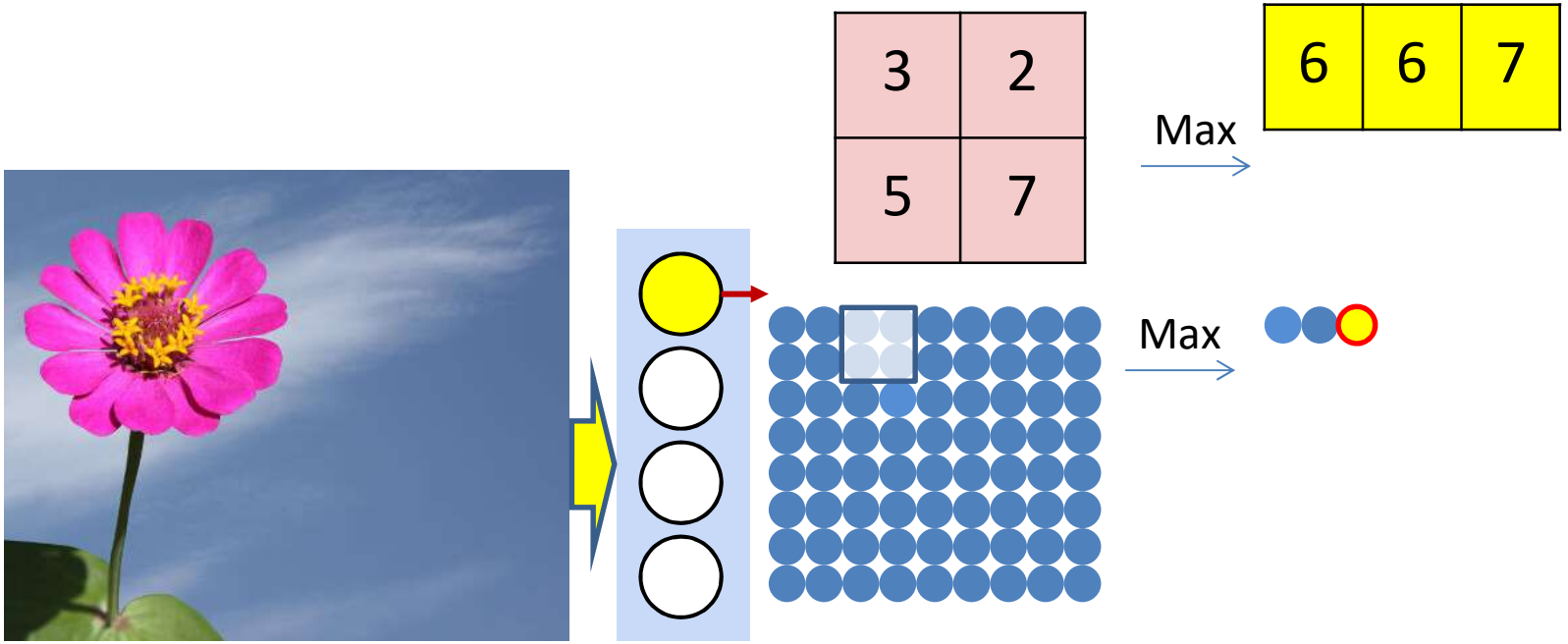
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

Recall: Max pooling



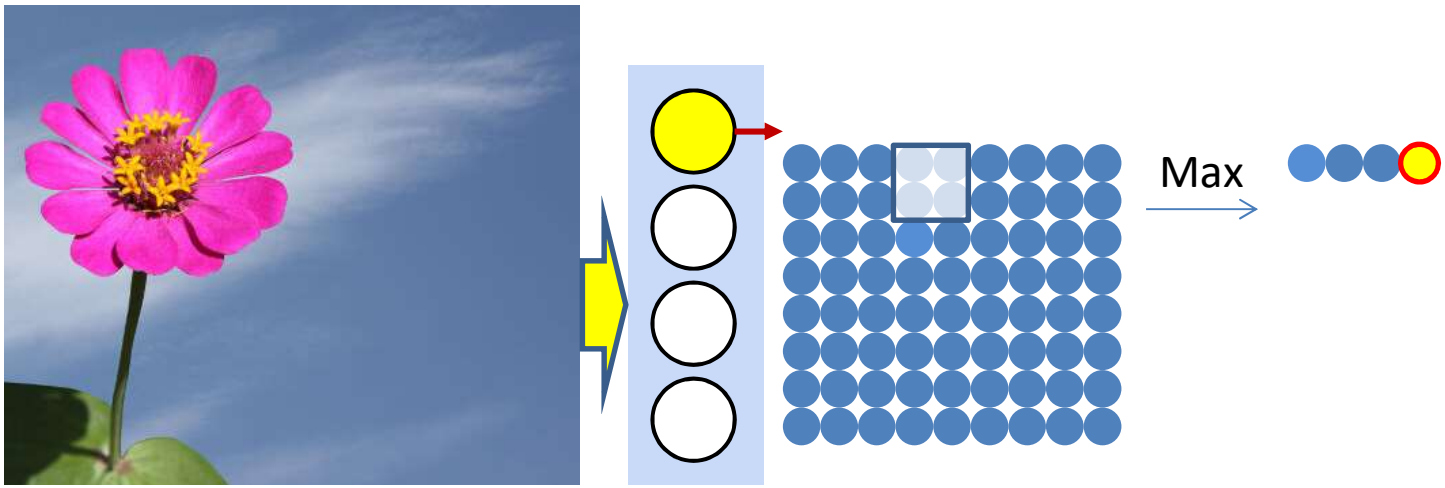
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

Recall: Max pooling



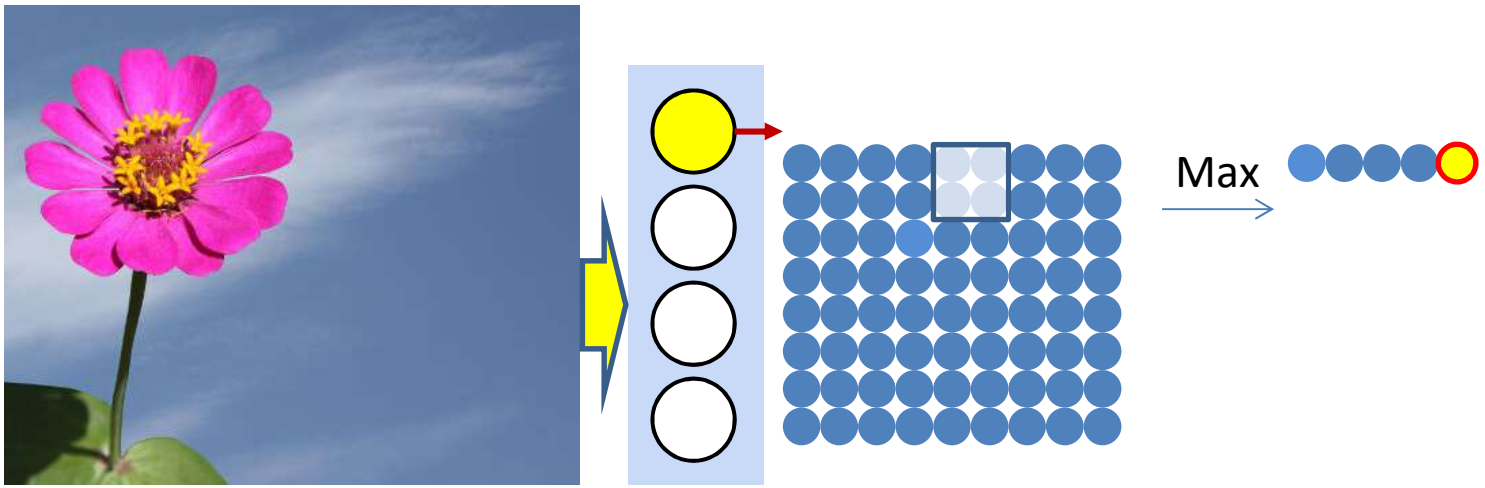
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

Recall: Max pooling



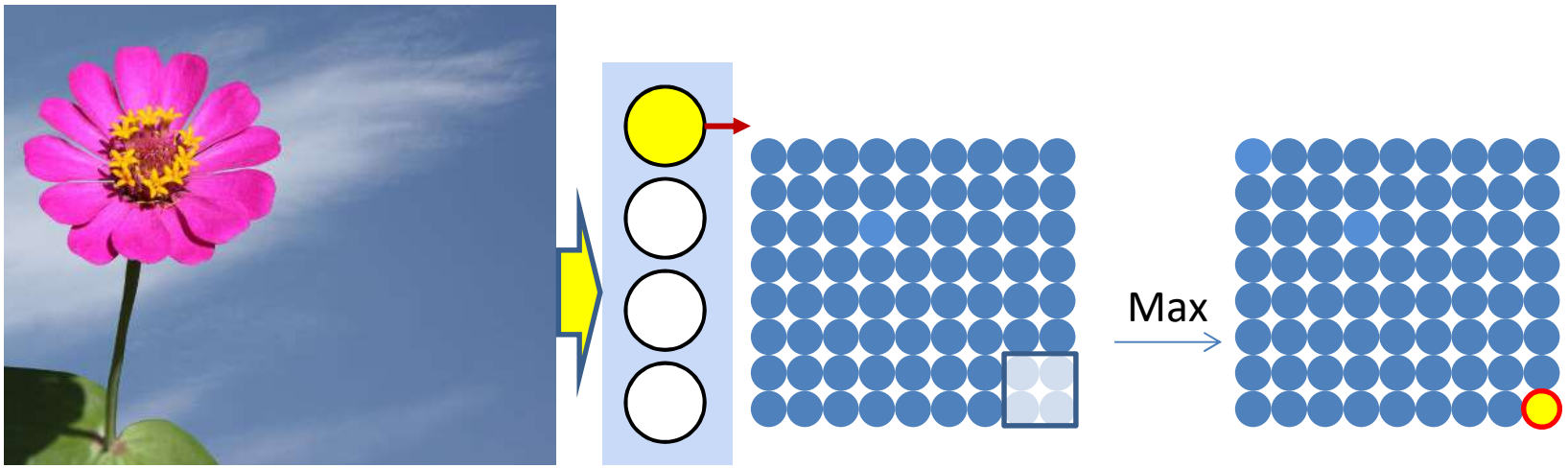
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

Recall: Max pooling



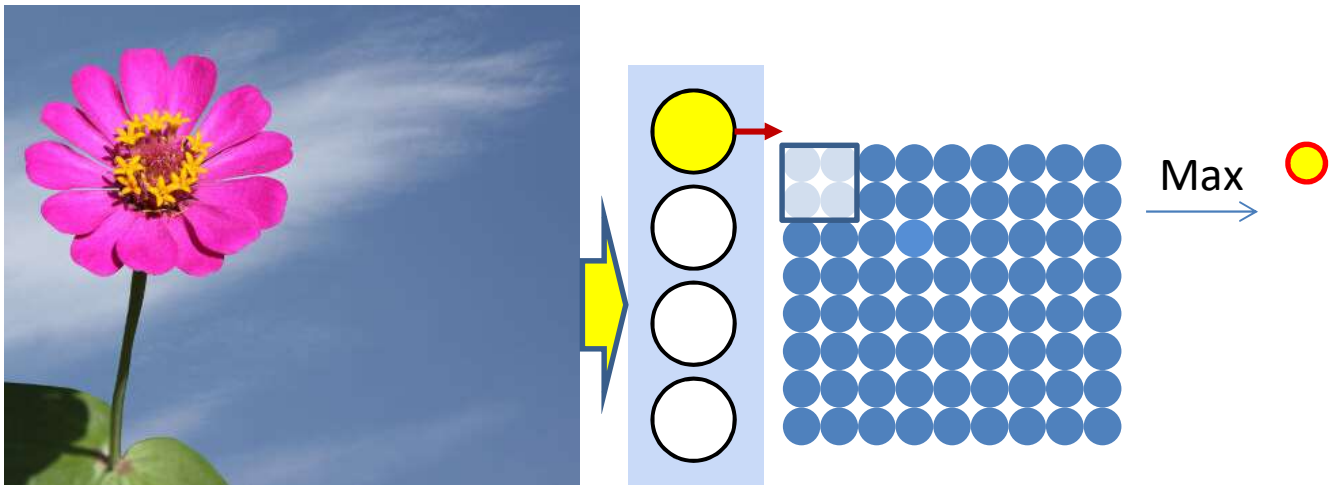
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

Recall: Max pooling



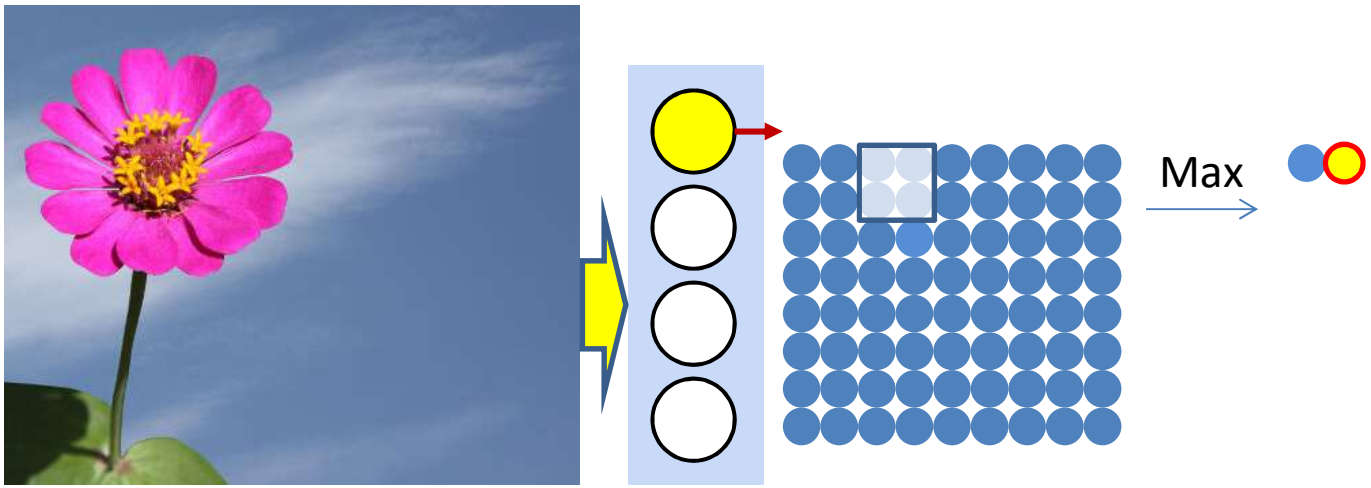
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

“Strides”



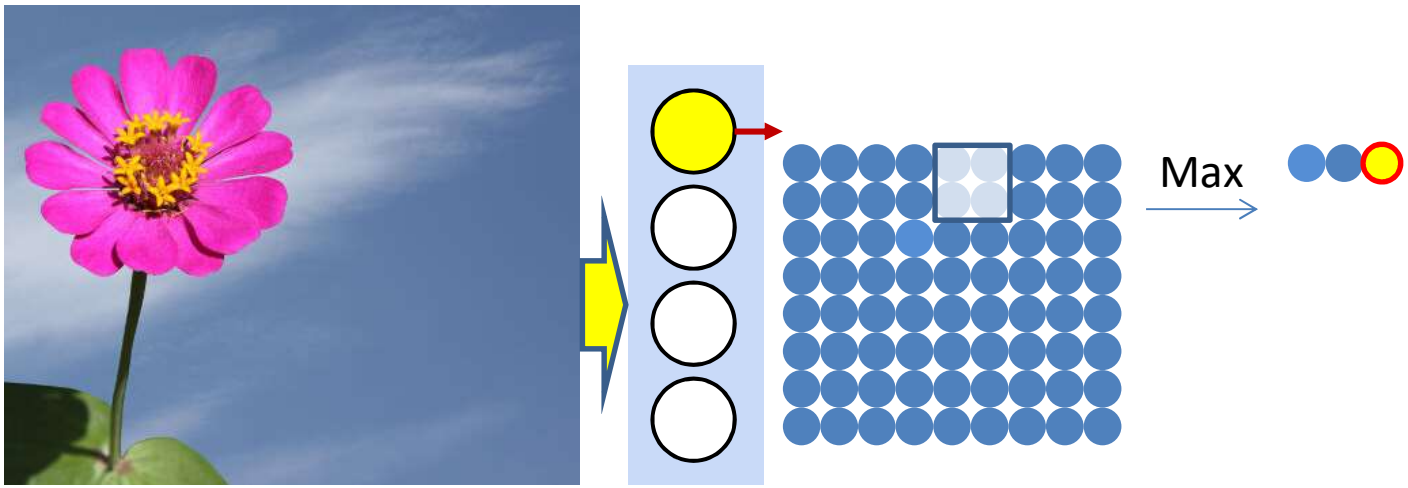
- The “max” operations may “stride” by more than one pixel

“Strides”



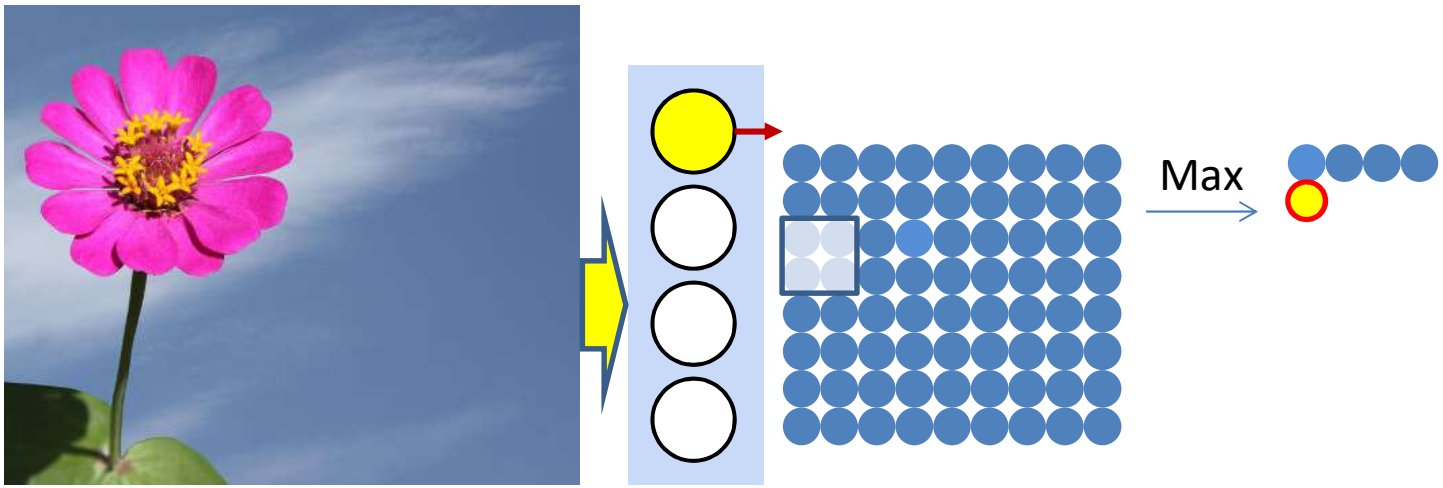
- The “max” operations may “stride” by more than one pixel

“Strides”



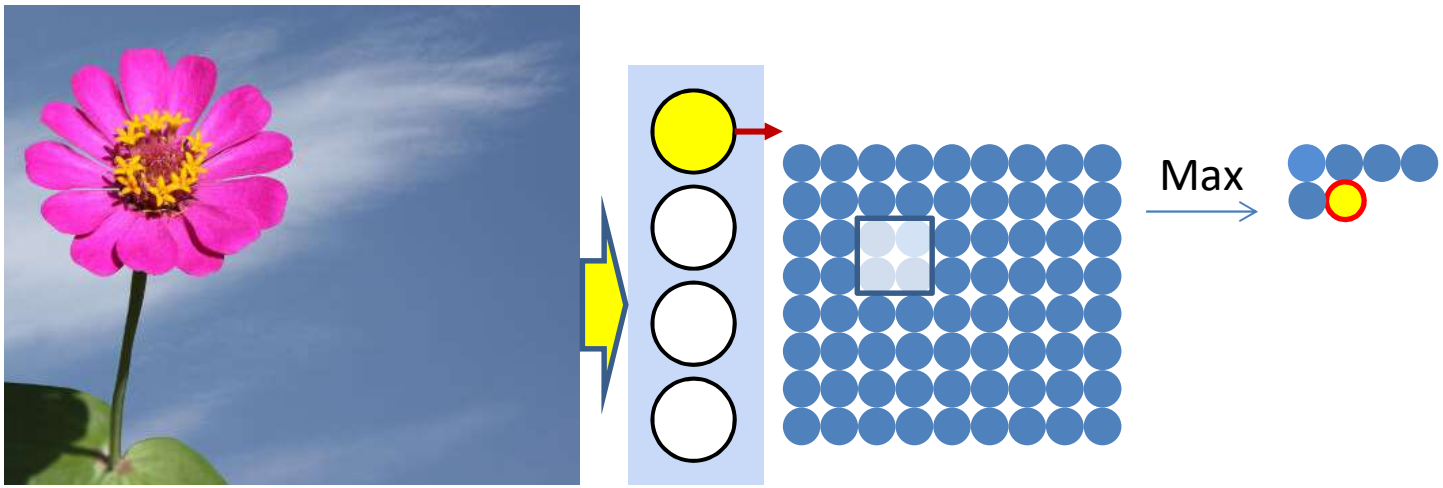
- The “max” operations may “stride” by more than one pixel

“Strides”



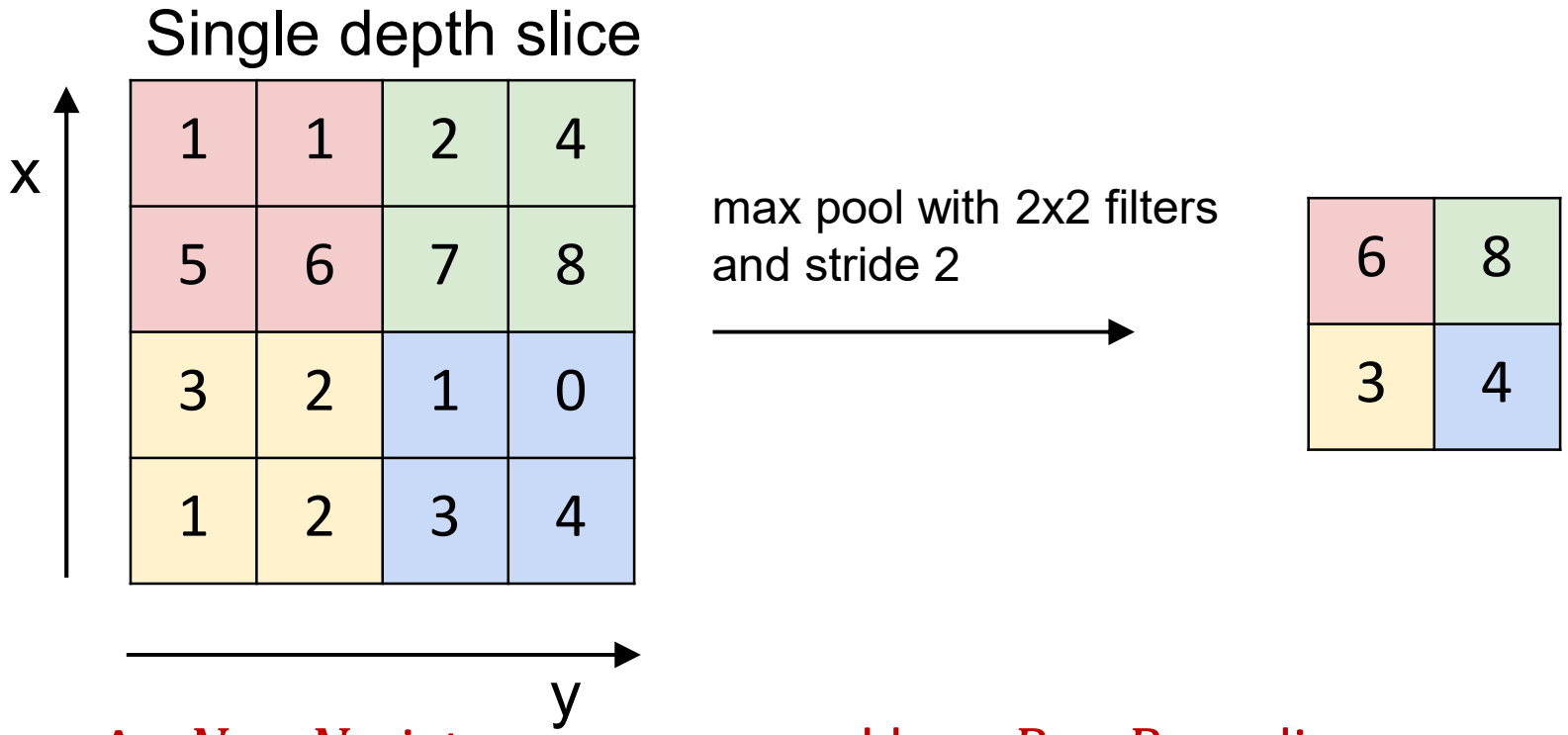
- The “max” operations may “stride” by more than one pixel

“Strides”



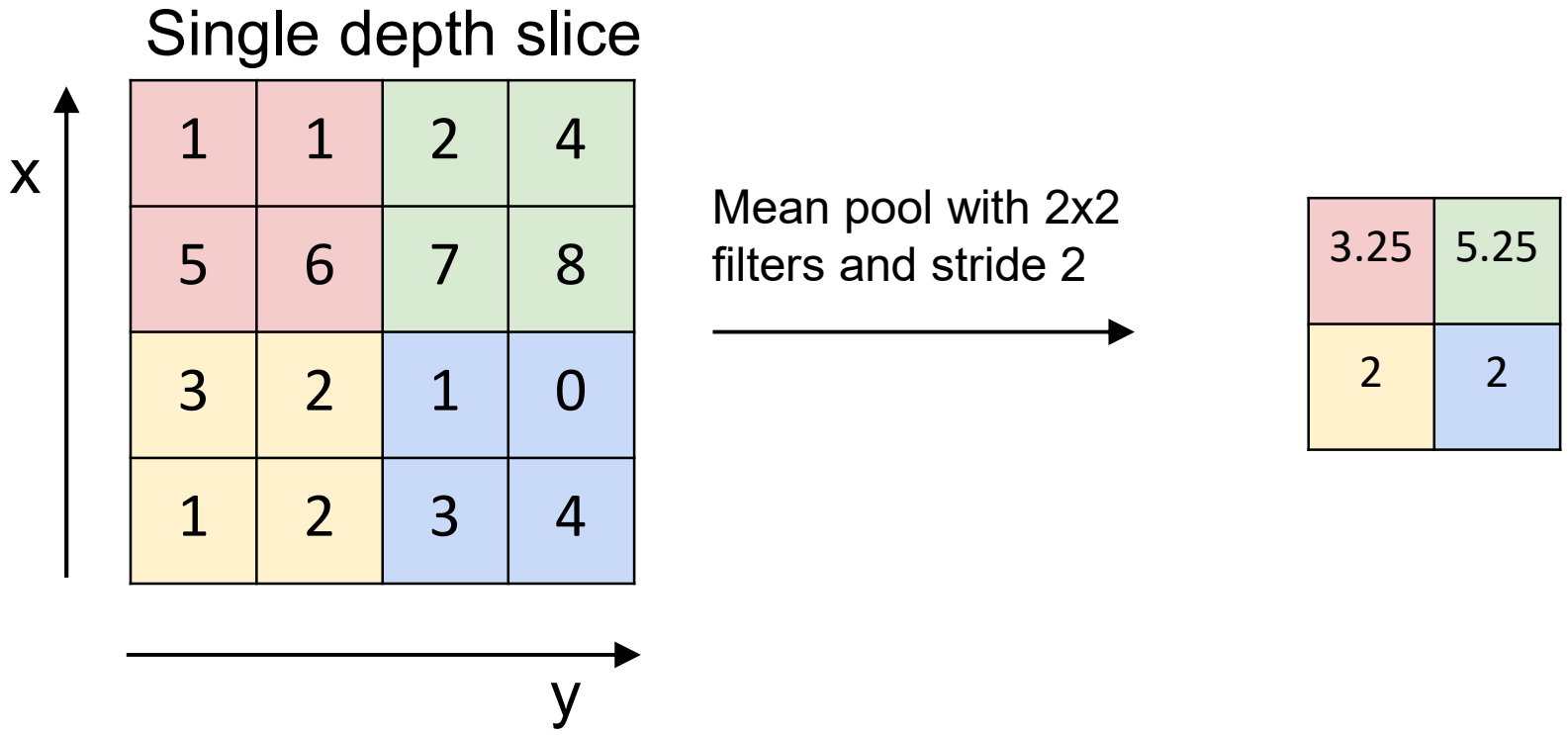
- The “max” operations may “stride” by more than one pixel

Pooling: Size of output



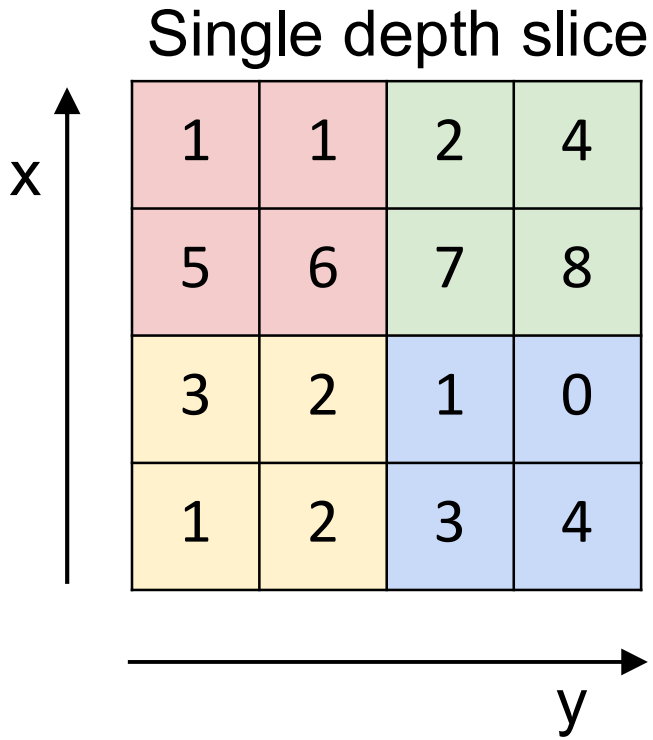
- An $N \times N$ picture compressed by a $P \times P$ pooling filter with stride D results in an output map of side $\lceil (N - P)/D \rceil + 1$
  - Typically do not zero pad

Alternative to Max pooling: Mean Pooling



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

Alternative to Max pooling: P-norm



P-norm with 2x2 filters and stride 2, $p = 5$

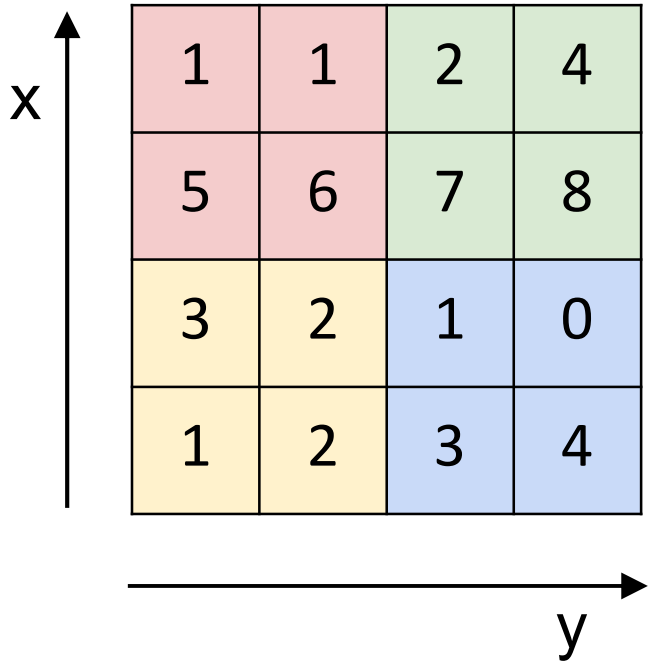
$$y = \sqrt[p]{\frac{1}{P^2}\sum_{i,j} x_{ij}^p}$$

4.86	8
2.38	3.16

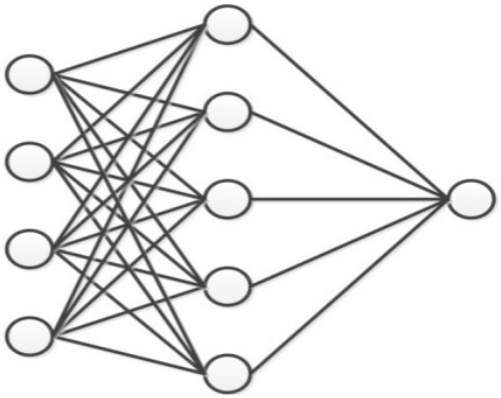
- Compute a p-norm of the pool

Other options

Single depth slice



Network applies to each 2x2 block and strides by 2 in this example

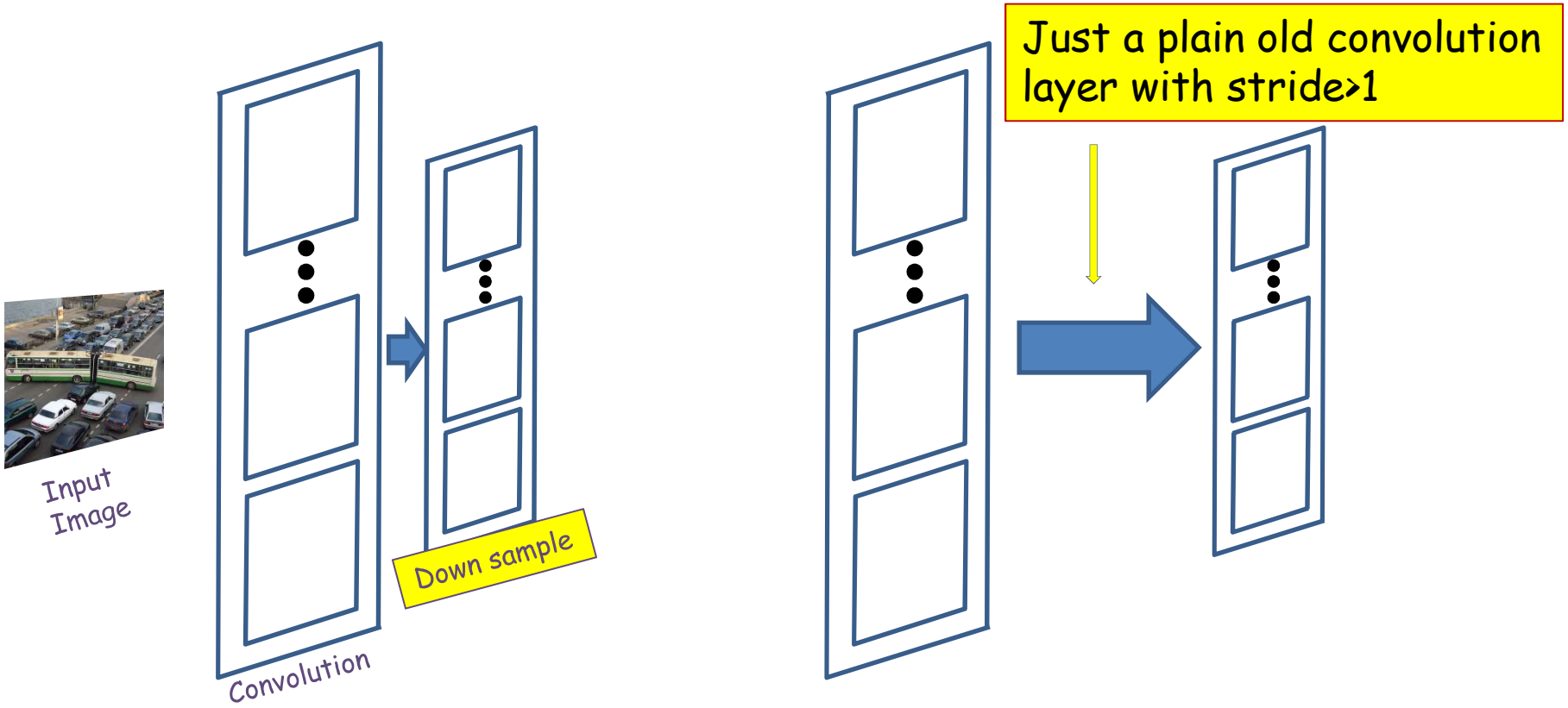


6	8
3	4

Network in network

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

Or even an "all convolutional" net



- Downsampling may even be done by a simple convolution layer with stride larger than 1
  - Replacing the maxpooling layer with a conv layer

Fully convolutional network (no pooling)

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

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

```
Y(0) = Image
for l = 1:L  # layers operate on vector at (x,y)
  for j = 1:D_l
     for x,m = 1:stride(l):W_{l-1}-K_l+1 # double indices
        for y,n = 1:stride(l):H_{l-1}-K_l+1
          segment = y(l-1,:,x:x+K_l-1,y:y+K_l-1) #3D tensor
          z(l,j,m,n) = W(l,j).segment #tensor inner prod.
          Y(l,j,m,n) = activation(z(l,j,m,n))

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

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 vote over groups of outputs from the convolutional layer
- Convolution can change the size of the output. This may be controlled via zero padding.
- Downsampling layers may perform max, p-norms, or be learned downsampling networks
- Regular convolutional layers with stride > 1 also perform downsampling
  - Eliminating the need for explicit downsampling layers

Setting everything together

- Typical image classification task
  - Assuming maxpooling..

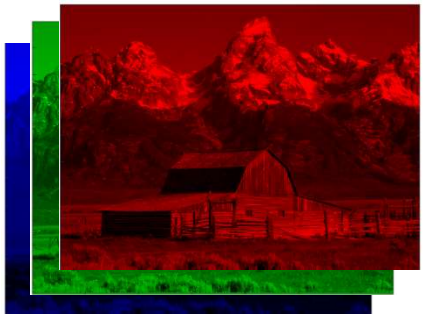
Convolutional Neural Networks

- Input: 1 or 3 images
  - Black and white or color
  - Will assume color to be generic

Convolutional Neural Networks

- Input: 3 pictures

Convolutional Neural Networks



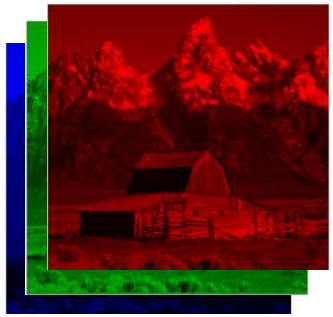
- Input: 3 pictures

Preprocessing

- Typically works with *square* images
  - Filters are also typically square

- Large networks are a problem
  - Too much detail
  - Will need big networks
- Typically scaled to small sizes, e.g. 32x32 or 128x128
  - Based on how much will fit on your GPU

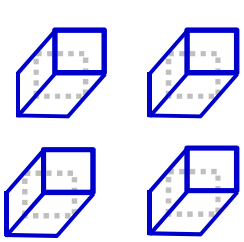
Convolutional Neural Networks



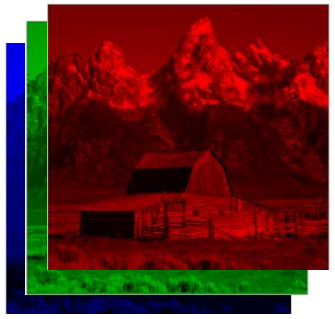
$I \times I$ *image*

- Input: 3 pictures

Convolutional Neural Networks



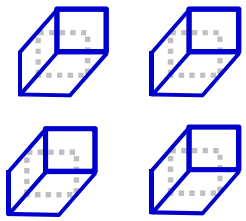
K_1 total filters
Filter size: $L \times L \times 3$



$I \times I$ *image*

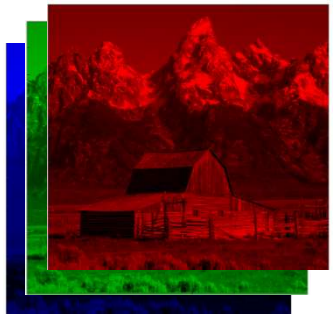
- Input is convolved with a set of K_1 filters
  - Typically K_1 is a power of 2, e.g. 2, 4, 8, 16, 32,..
  - Filters are typically 5x5, 3x3, or even 1x1

Convolutional Neural Networks



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

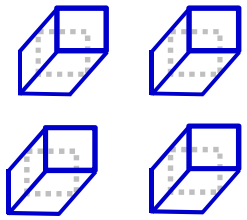
Small enough to capture fine features
(particularly important for scaled-down images)



$I \times I$ *image*

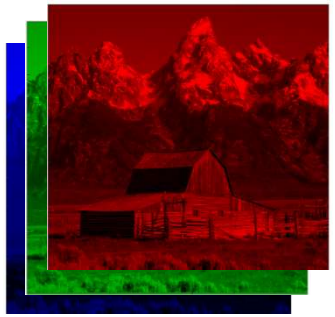
- Input is convolved with a set of K_1 filters
  - Typically K_1 is a power of 2, e.g. 2, 4, 8, 16, 32,..
  - Filters are typically 5x5, 3x3, or even 1x1

Convolutional Neural Networks



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

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

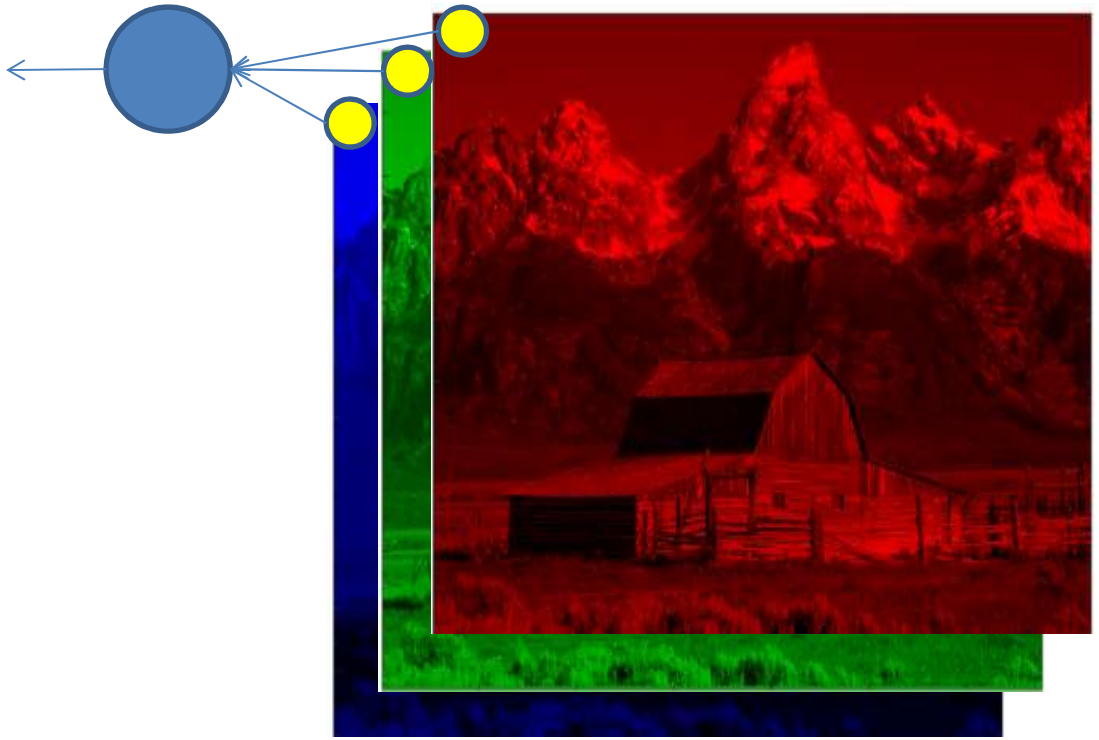


$I \times I$ *image*

What on earth is this?

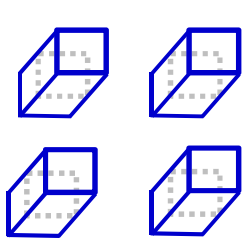
- Input is convolved with a set of K_1 filters
  - Typically K_1 is a power of 2, e.g. 2, 4, 8, 16, 32,..
  - Filters are typically 5x5, 3x3, or even 1x1

The 1x1 filter

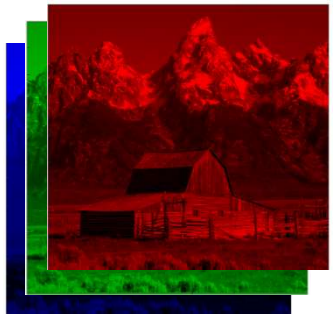


- A 1x1 filter is simply a perceptron that operates over the *depth* of the map, but has no spatial extent
  - Takes one pixel from each of the maps (at a given location) as input

Convolutional Neural Networks



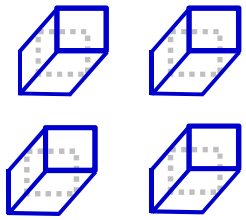
K_1 total filters
Filter size: $L \times L \times 3$



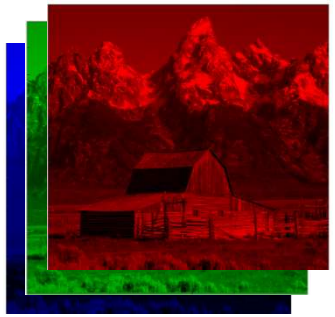
$I \times I$ *image*

- 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,..
  - **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*

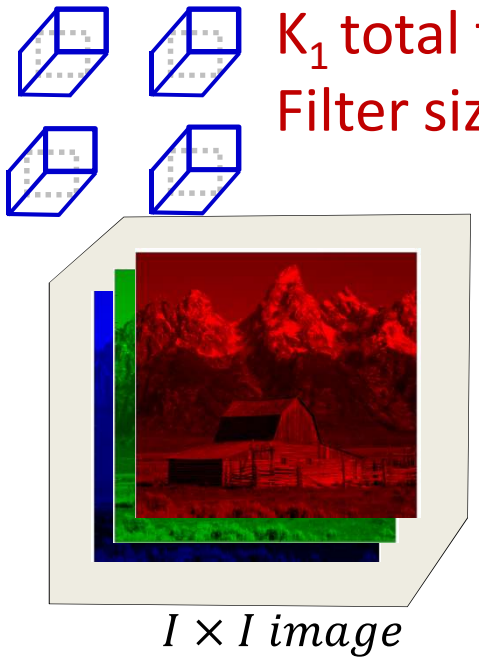
Parameters to choose: K_1, L and S

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

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

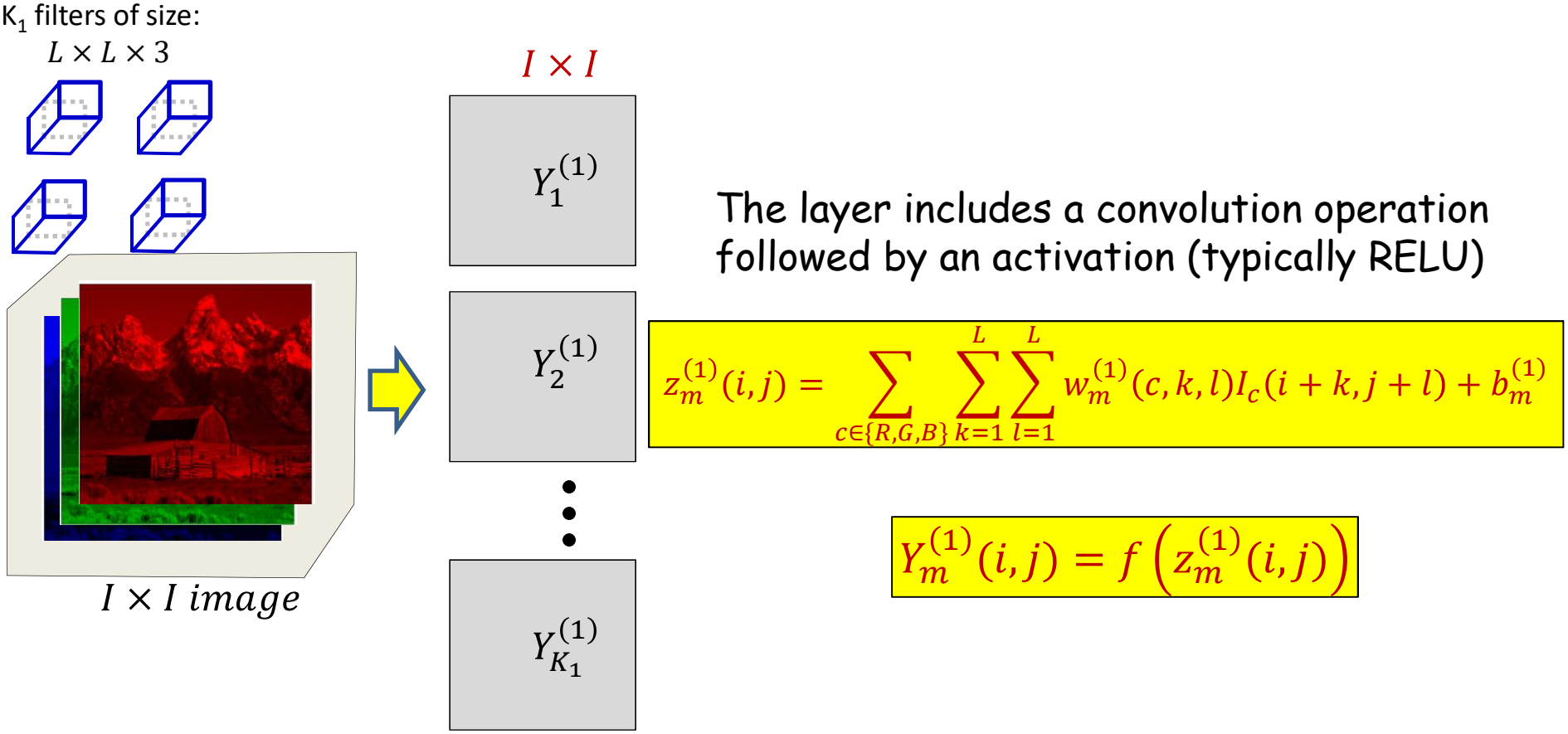
- Input is convolved with a set of K_1 filters
  - Typically K_1 is a power of 2, e.g. 2, 4, 8, 16, 32,..
  - **Better notation:** Filters are typically 5x5(x3), 3x3(x3), or even 1x1(x3)
  - ***Typical stride:*** 1 or 2

Convolutional Neural Networks



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

Convolutional Neural Networks



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

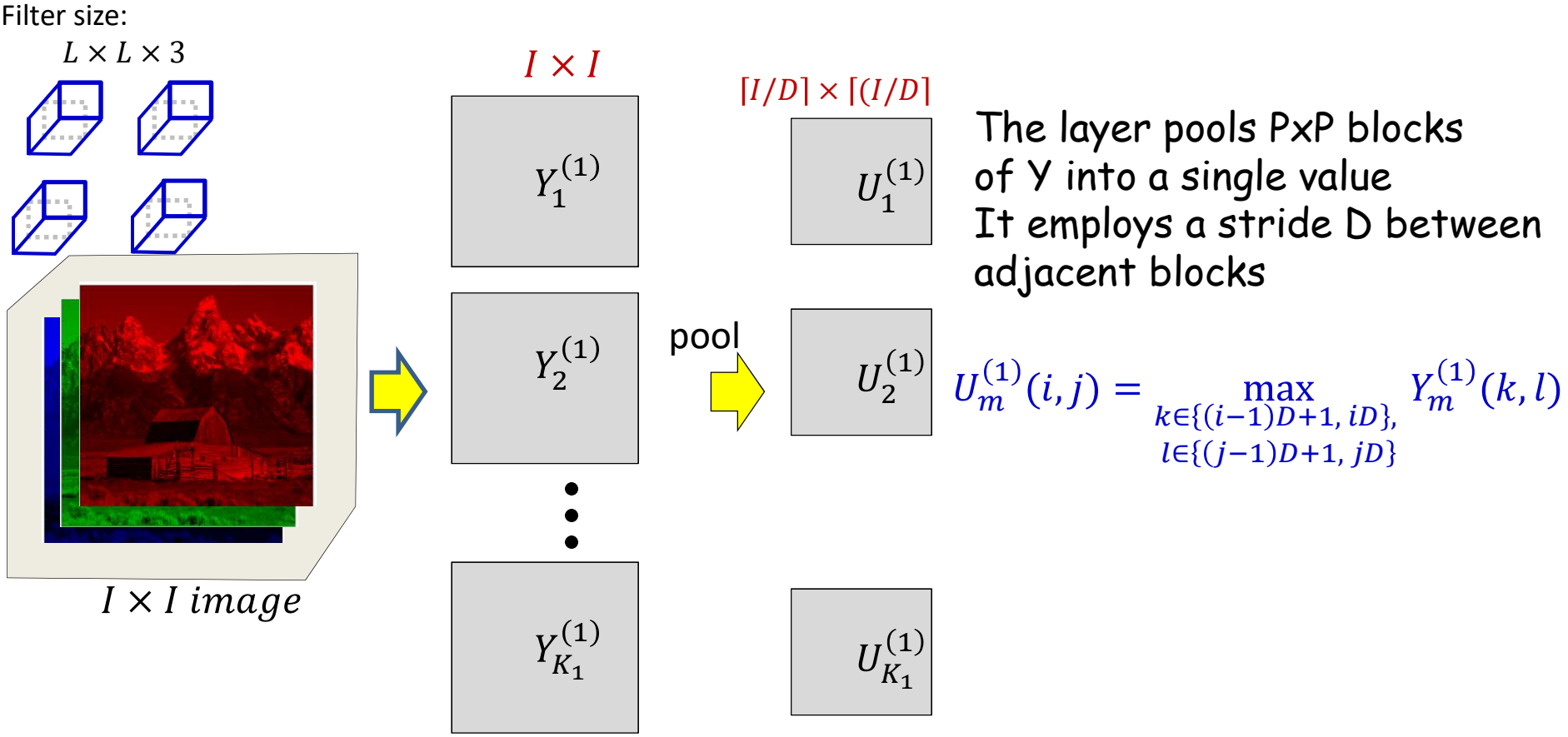
Learnable parameters in the first convolutional layer

- 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

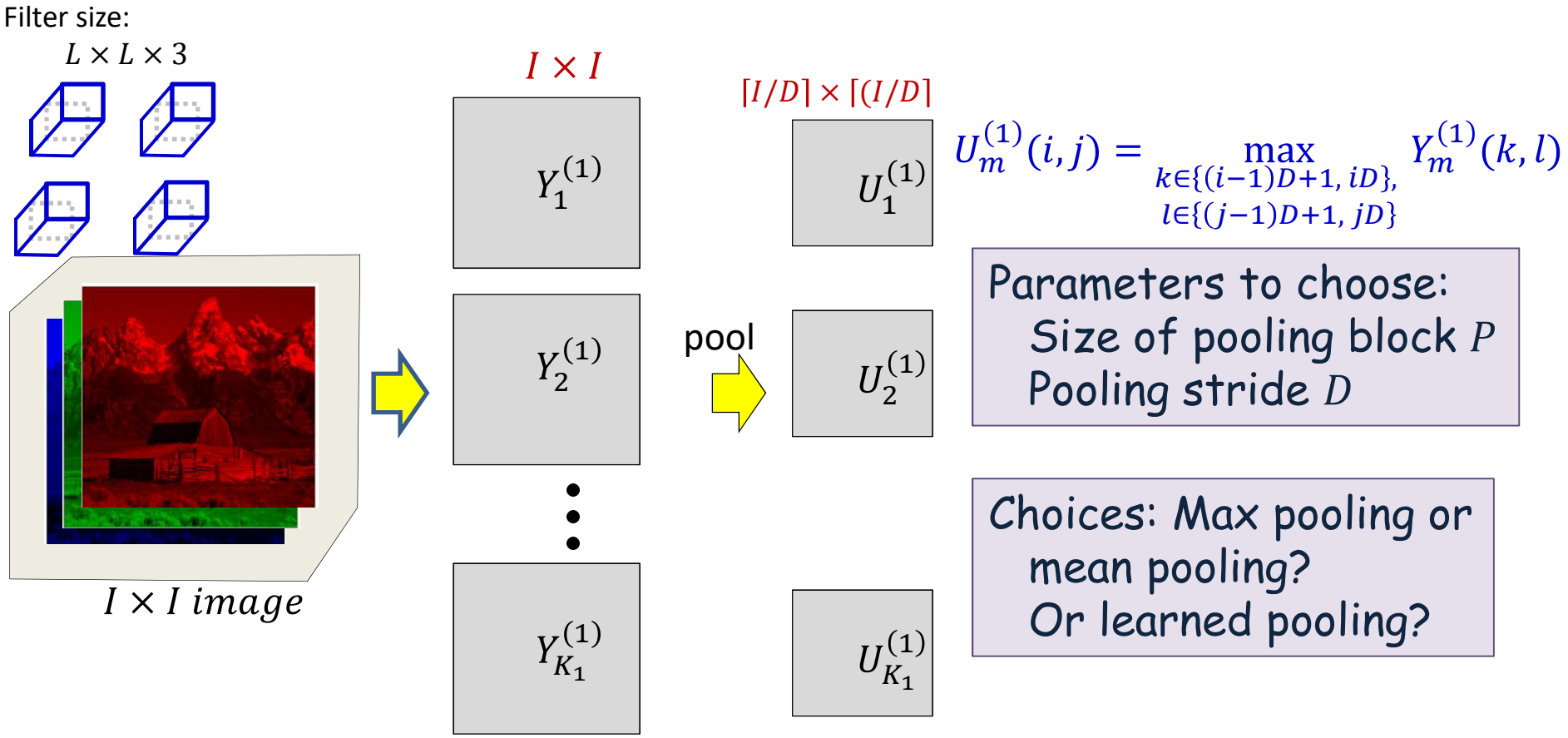
- All of these parameters must be learned

Convolutional Neural Networks



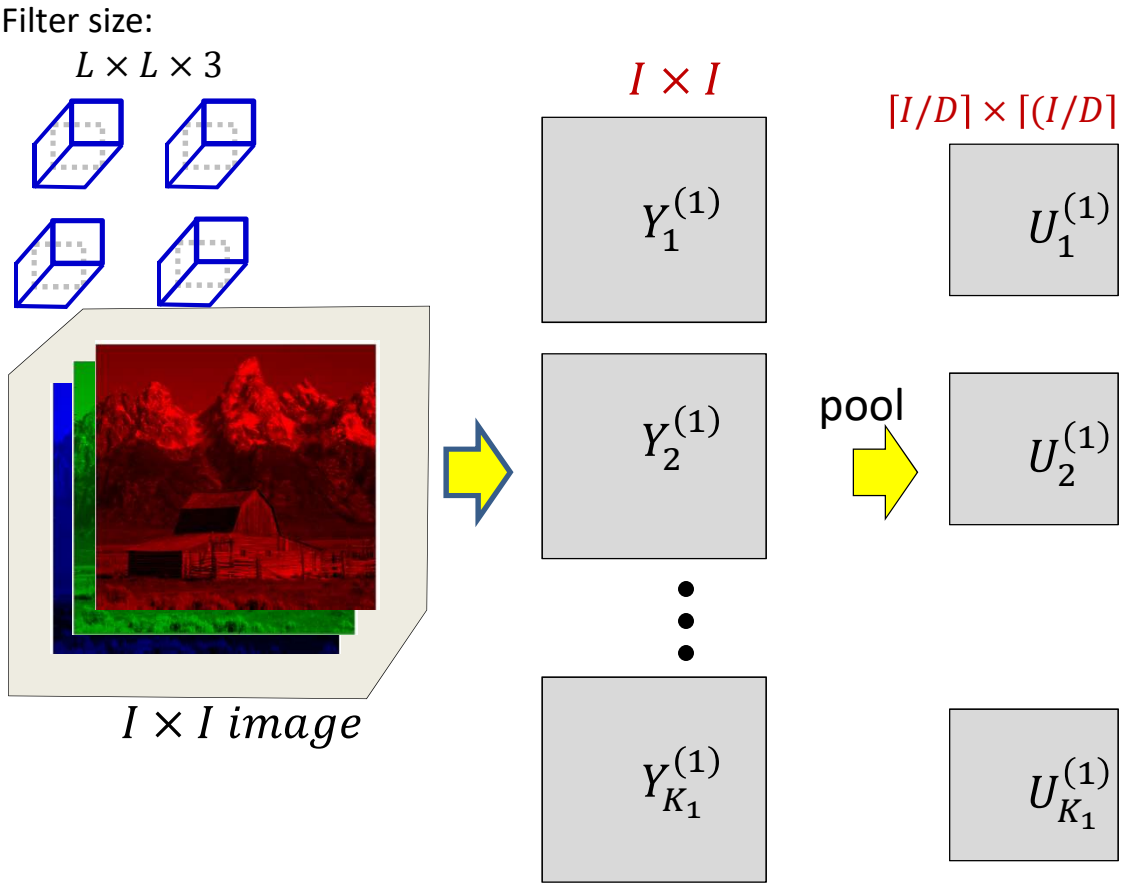
- **First downsampling layer:** From each $P \times P$ block of each map, *pool* down to a single value
  - For max pooling, during training keep track of which position had the highest value

Convolutional Neural Networks



- **First 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

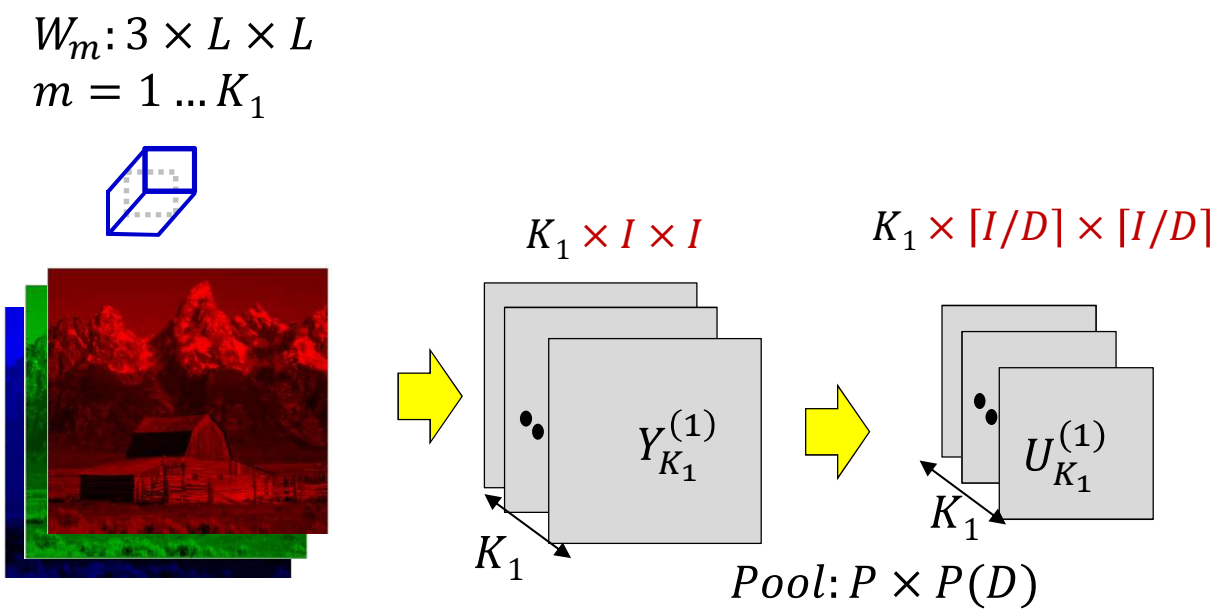


$$P_m^{(1)}(i,j) = \underset{\substack{k \in \{(i-1)D+1,\, iD\},\\ l \in \{(j-1)D+1,\, jD\}}}{\operatorname{argmax}} Y_m^{(1)}(k,l)$$

$$U_m^{(1)}(i,j) = Y_m^{(1)}(P_m^{(1)}(i,j))$$

- **First 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

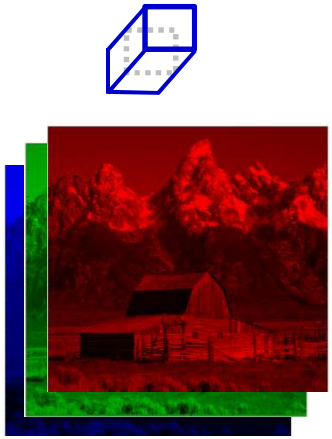


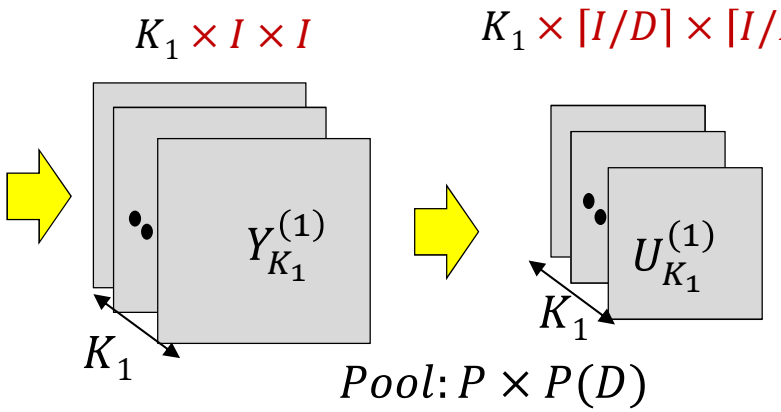
- **First pooling layer:** Drawing it differently for convenience

Convolutional Neural Networks

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

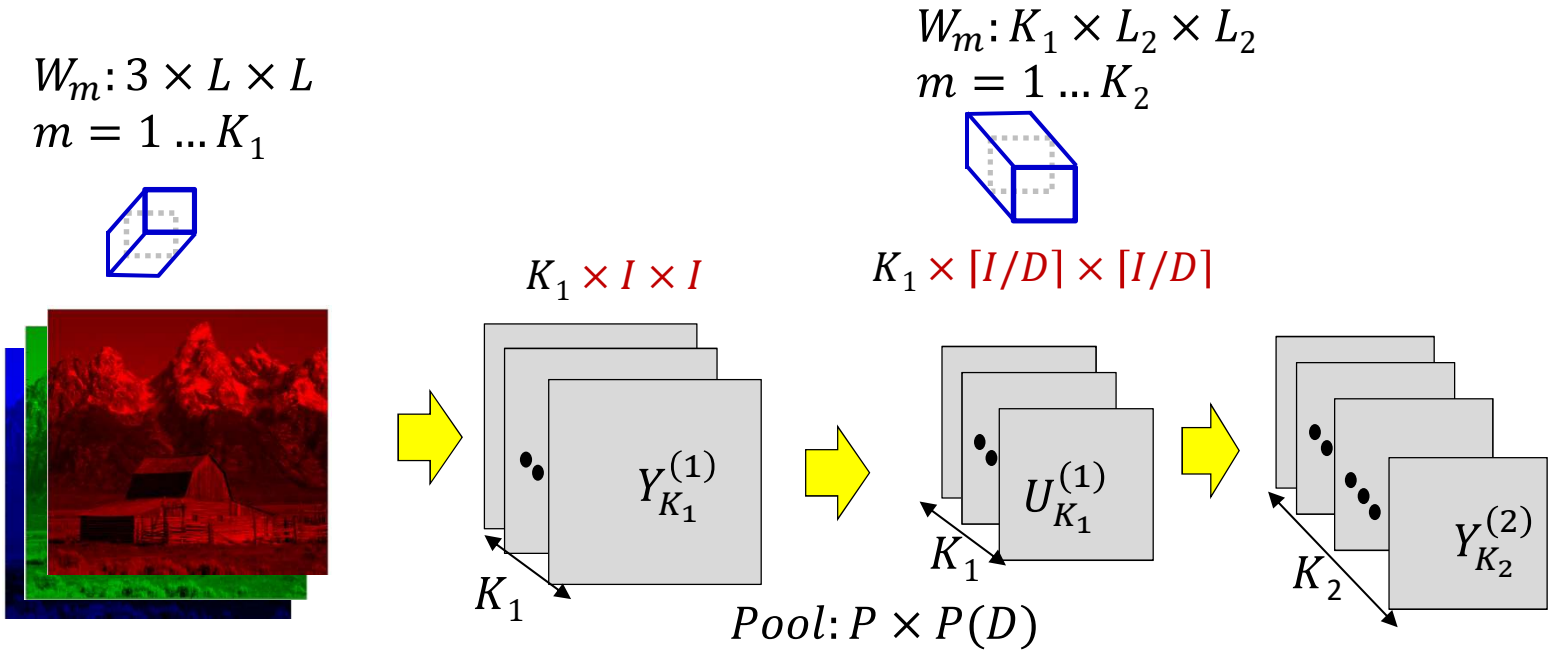
Jargon: Filters are often called "Kernels"
The outputs of individual filters are called "channels"
The number of filters (K_1, K_2, etc) is the number of channels





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

Convolutional Neural Networks

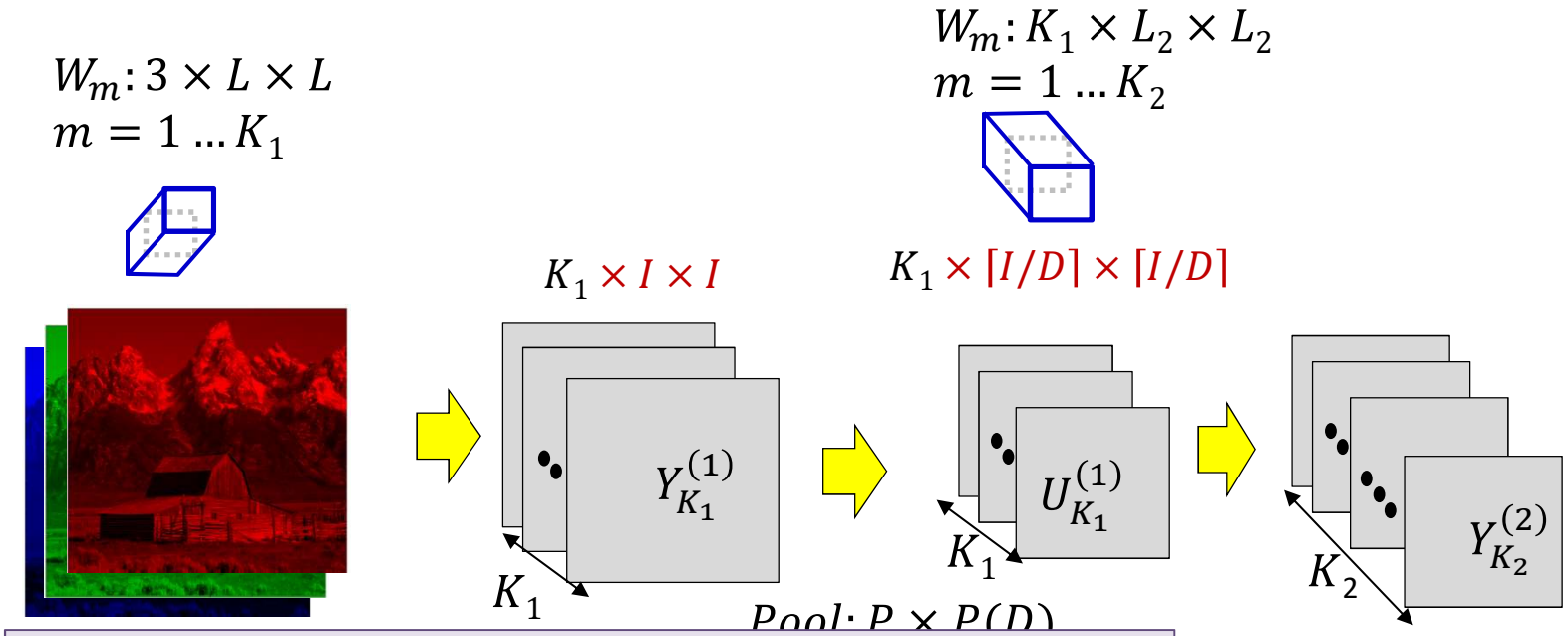


$$z_m^{(n)}(i,j) = \sum_{r=1}^{K_{n-1}} \sum_{k=1}^{L_n} \sum_{l=1}^{L_n} w_m^{(n)}(r,k,l) U_r^{(n-1)}(i+k, j+l) + b_m^{(n)}$$

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

- **Second convolutional layer:** K_2 3-D filters resulting in K_2 2-D maps
  - Alternately, a kernel with K_2 output channels

Convolutional Neural Networks



Parameters to choose: K_2, L_2 and S_2

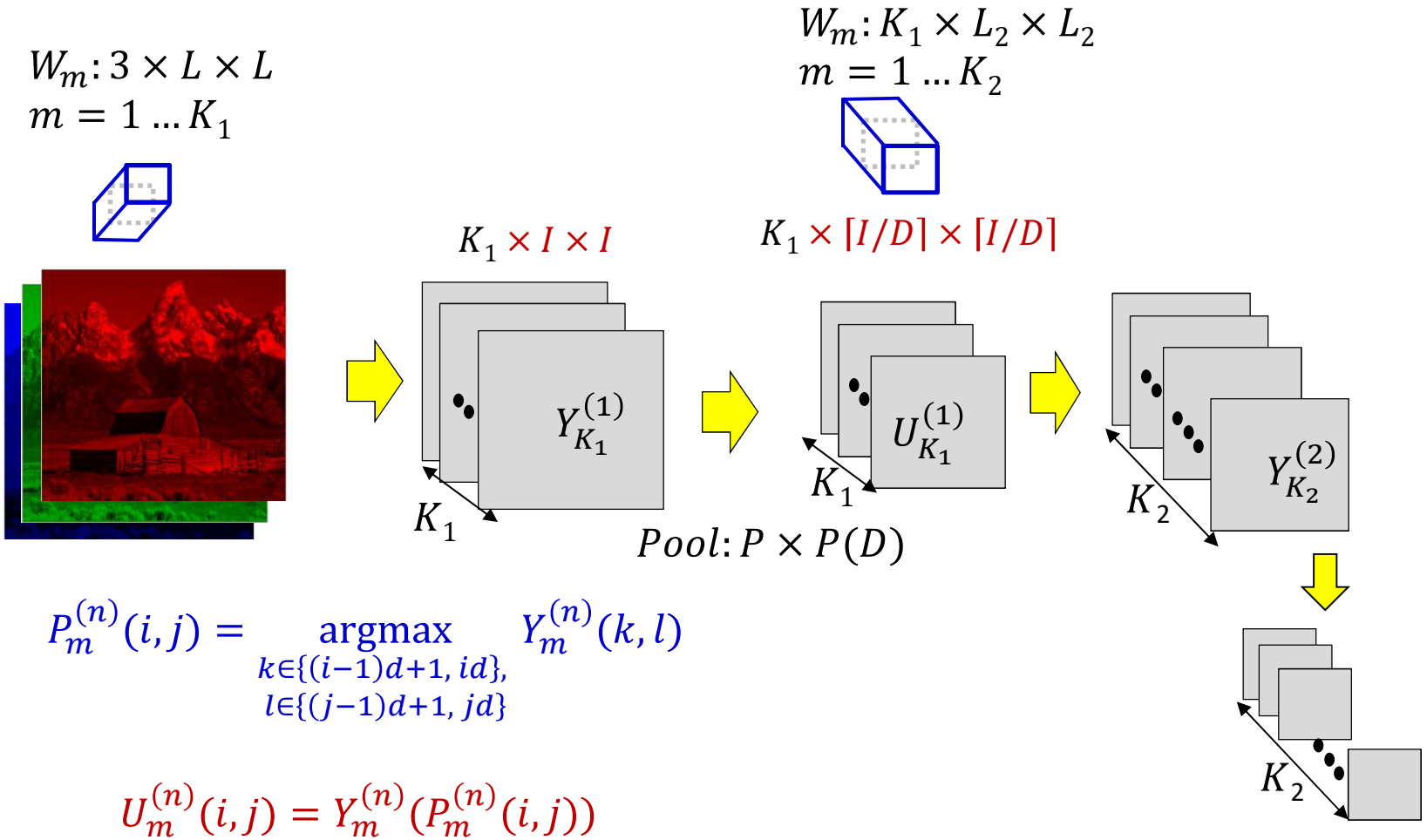
1. Number of filters K_2
2. Size of filters $L_2 \times L_2 \times K_1 + bias$
3. *Stride* of convolution S_2

Total number of parameters: $K_2(K_1 L_2^2 + 1)$

All these parameters must be learned

ng in K_2 2-D maps

Convolutional Neural Networks

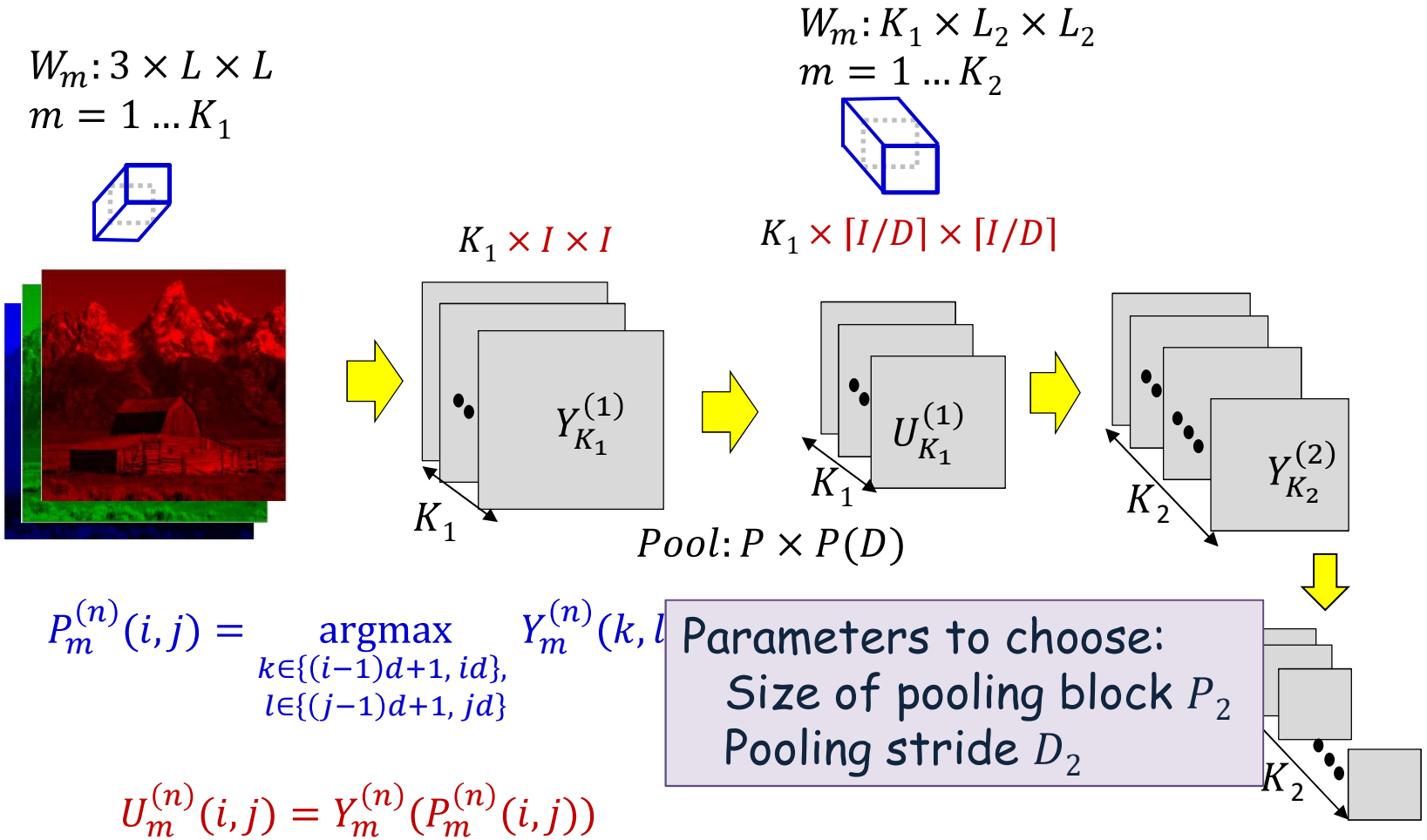


$$P_m^{(n)}(i,j) = \underset{\substack{k \in \{(i-1)d+1,\, id\},\\ l \in \{(j-1)d+1,\, jd\}}}{\operatorname{argmax}} Y_m^{(n)}(k,l)$$

$$U_m^{(n)}(i,j) = Y_m^{(n)}(P_m^{(n)}(i,j))$$

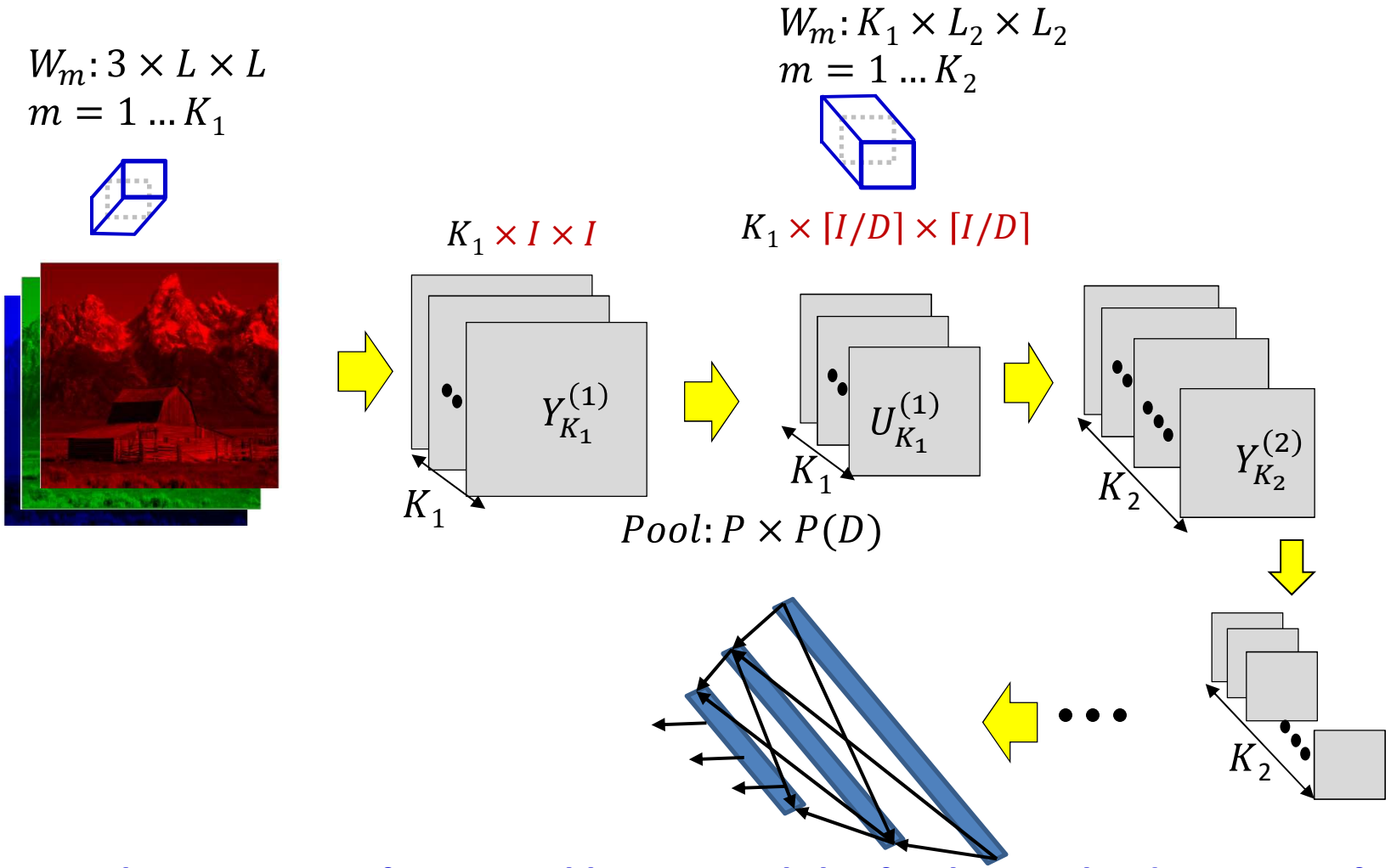
- **Second convolutional layer:** K_2 3-D filters resulting in K_2 2-D maps
- **Second pooling layer**: K_2 Pooling operations: outcome K_2 reduced 2D maps

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 i

The Size of the Layers

- Each convolution layer 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 may *increase* the *number* of maps from the previous layer
- Each pooling layer with hop 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)

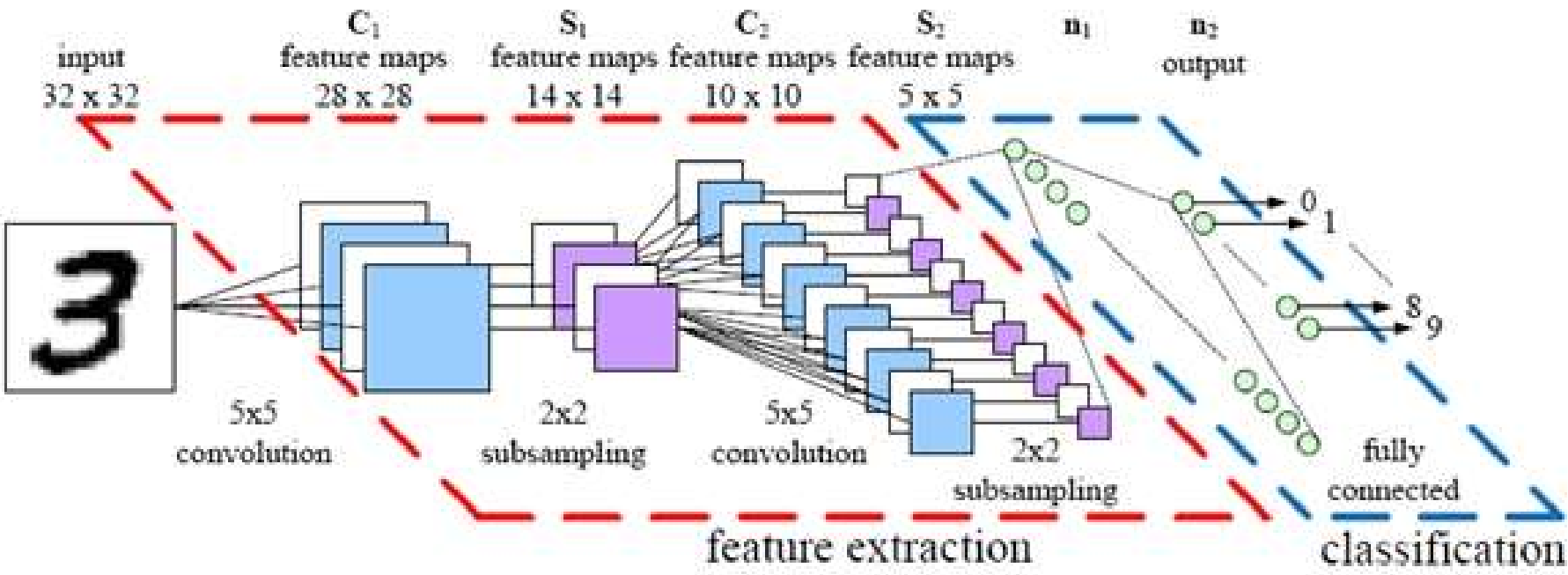
- Number of convolutional and downsampling layers
  - And arrangement (order in which they follow one another)

- 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

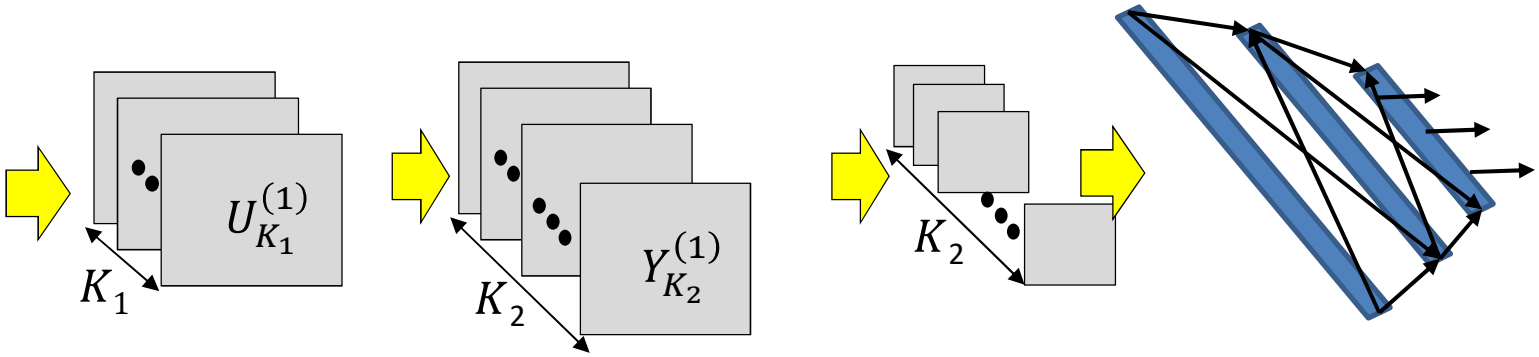
- For each downsampling/pooling layer:
  - Spatial extent of filter $P_i \times P_i$
  - The stride D_i

- For the final MLP:
  - Number of layers, and number of neurons in each layer

Digit classification

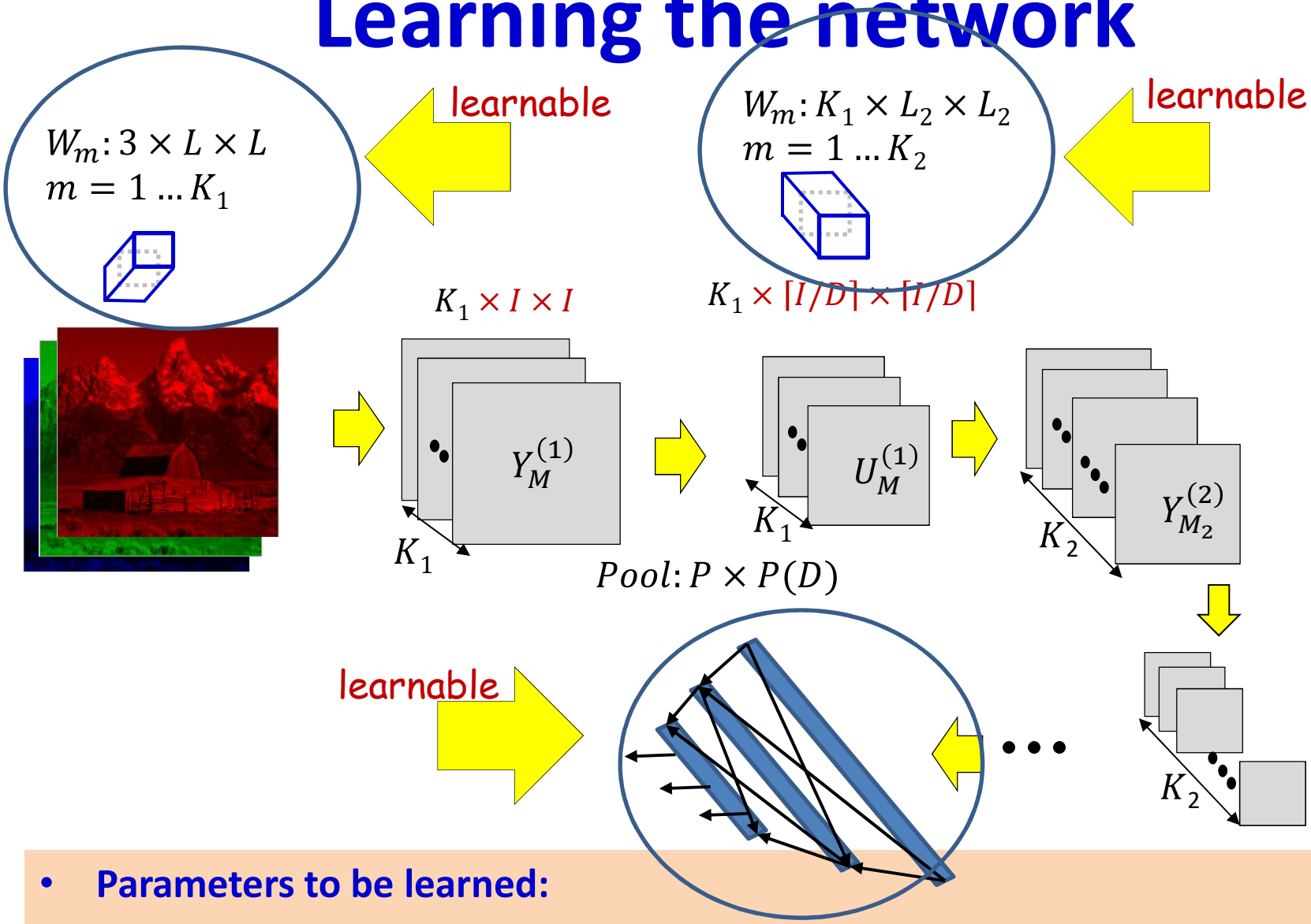


Training



- 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



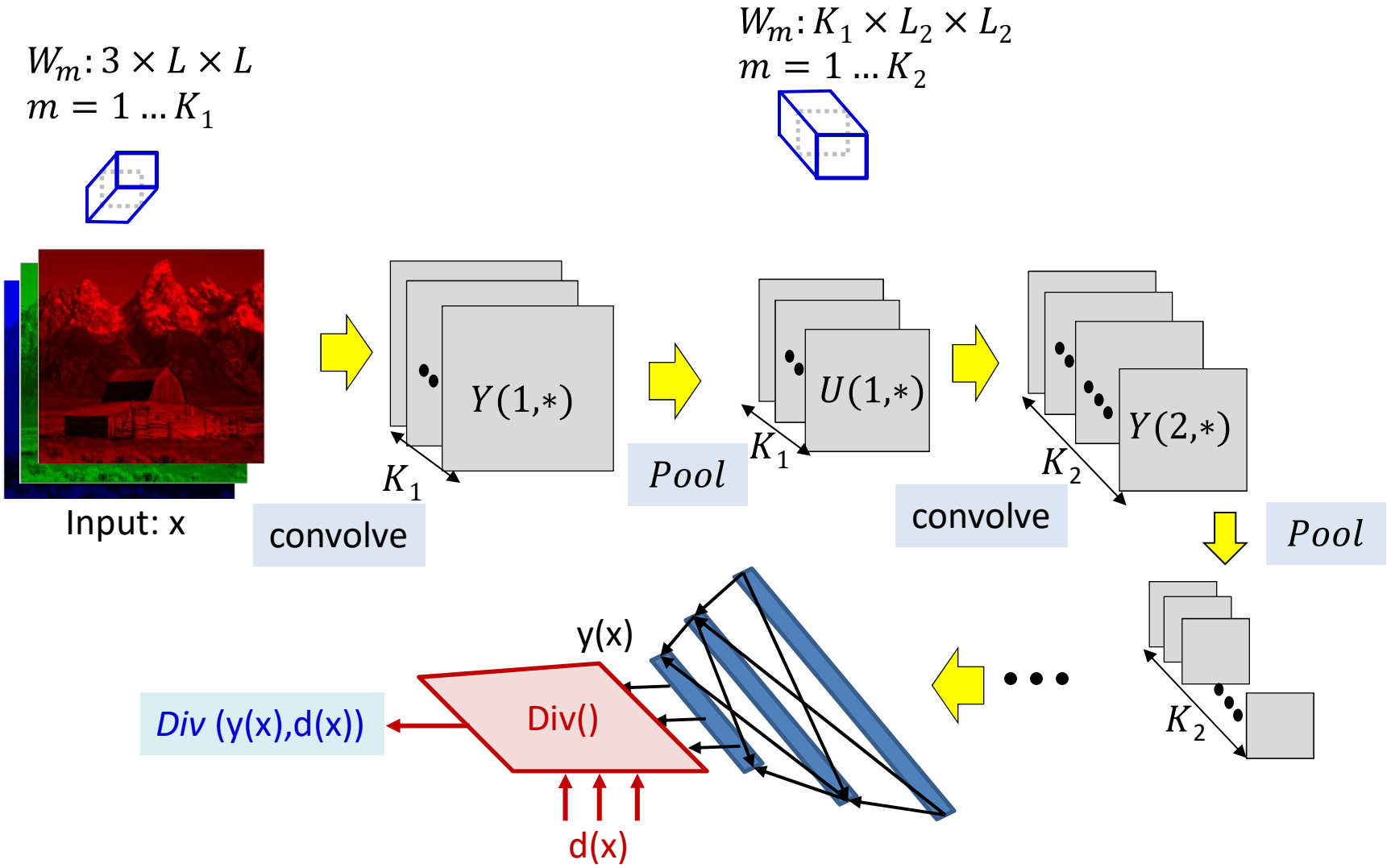
- **Parameters to be learned:**
  - **The weights of the neurons in the final MLP**
  - **The (weights and biases of the) filters for every *convolutional* layer**

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^{th} layer be size $L_J \times L_J$
- For the J^{th} layer we will require $K_J\left(K_{J-1}L_J^2 + 1\right)$ filter parameters
- Total parameters required for the *convolutional* layers:
  $\sum_{J \in convolutional\ layers} K_J\left(K_{J-1}L_J^2 + 1\right)$

Defining the loss



- The loss for a single instance

Problem Setup

- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)$

- The loss on the ith instance is $div(Y_i, d_i)$
- The total loss

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

- Minimize $Loss$ 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:

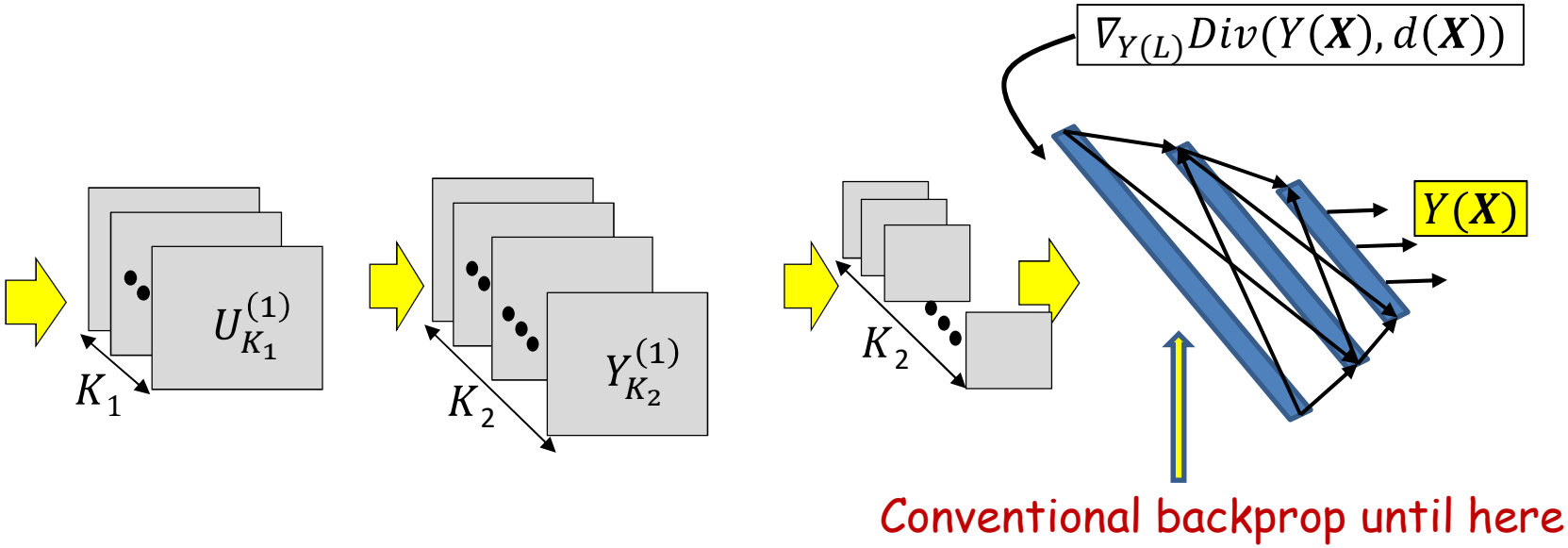
$$Loss = \frac{1}{T}\sum_{i} Div(Y_i, d_i)$$

- Computing the derivative

Total derivative:

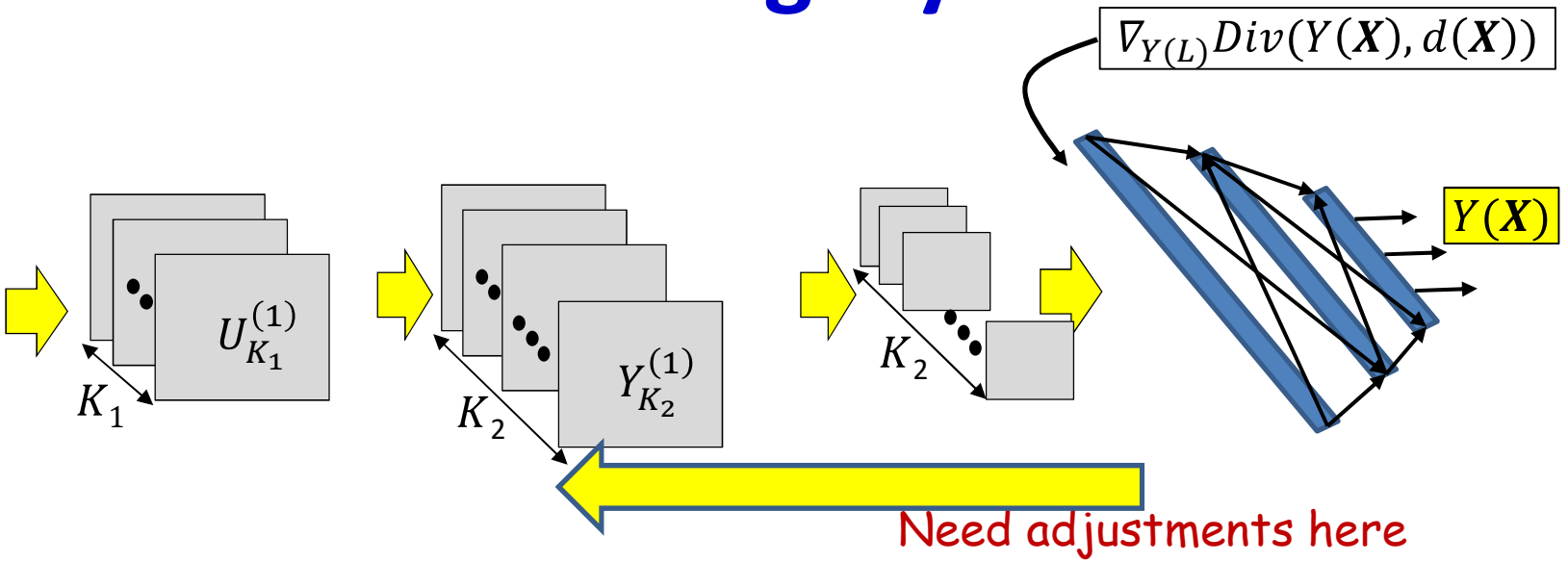
$$\frac{dLoss}{dw(l,m,j,x,y)} = \frac{1}{T}\sum_{i} \frac{dDiv(Y_i, d_i)}{dw(l,m,j,x,y)}$$

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 "unrolling" of the maps from the final convolutional layer

Backpropagation: Convolutional and Pooling layers

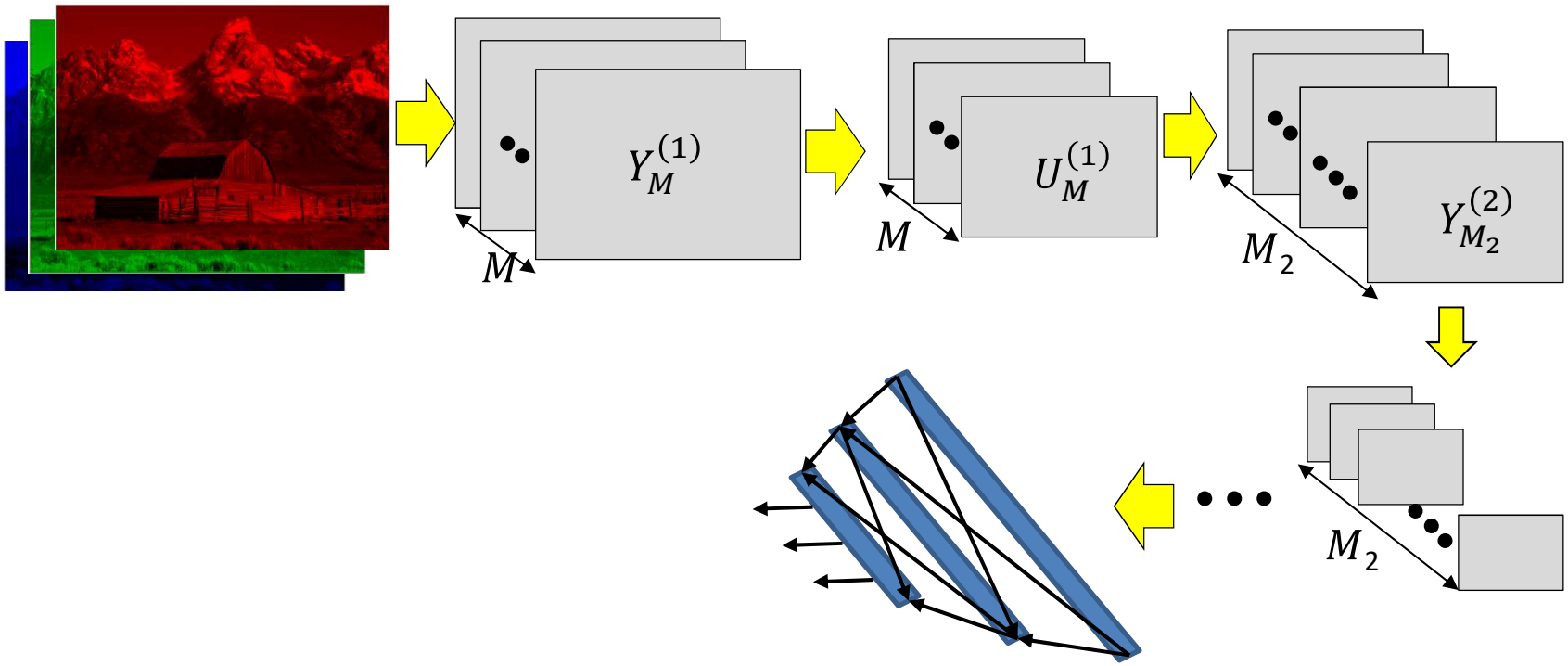


- Backpropagation from the flat MLP requires special consideration of
  - The shared computation in the convolution layers
  - The pooling layers (particularly maxout)

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..