Deep Neural Networks
Convolutional Networks II

Bhiksha Raj

Fall 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 subnetwork 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 convolutional network
- For 1-D scan along time, it is called a Time-delay neural network

A little history

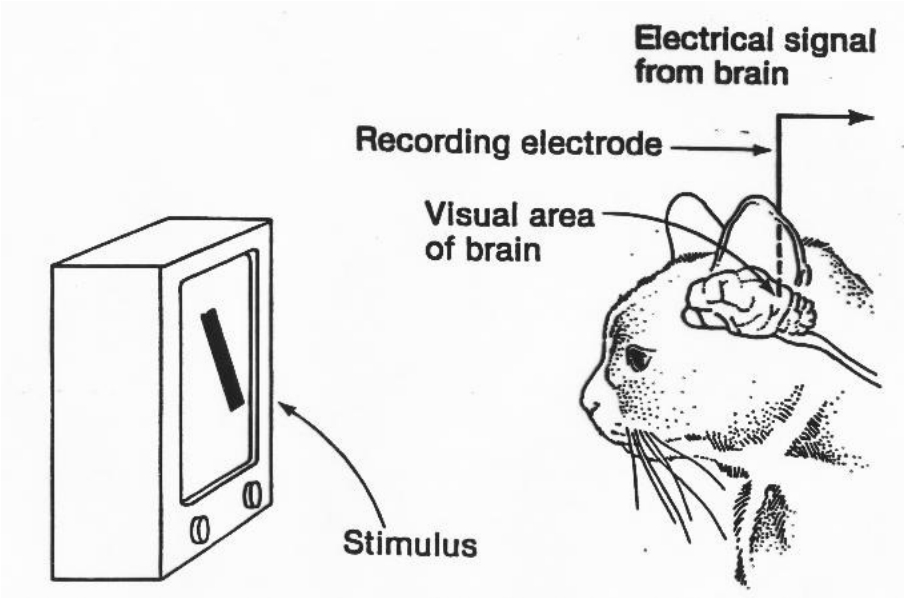
- How do animals see?
  - What is the neural process from eye to recognition?

- 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

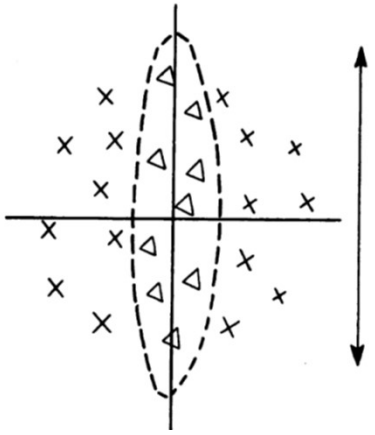
Hubel and Wiesel 1959



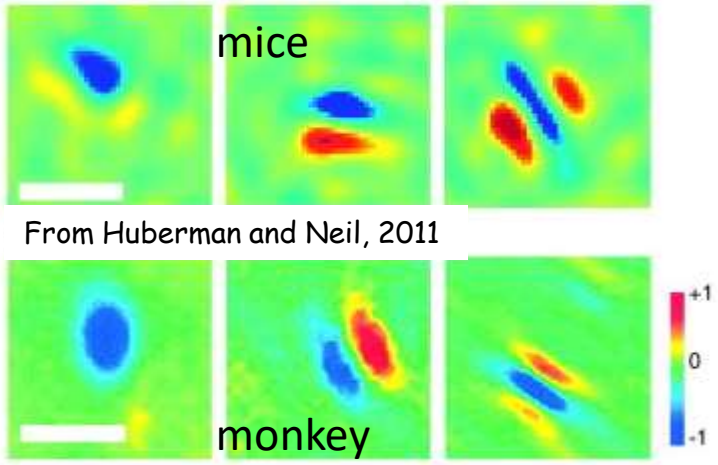
- Light of different wavelengths incident on the retina through fully open (slitted) Iris
  - Defines *immediate* (20ms) response of retinal cells

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

Hubel and Wiesel 1959



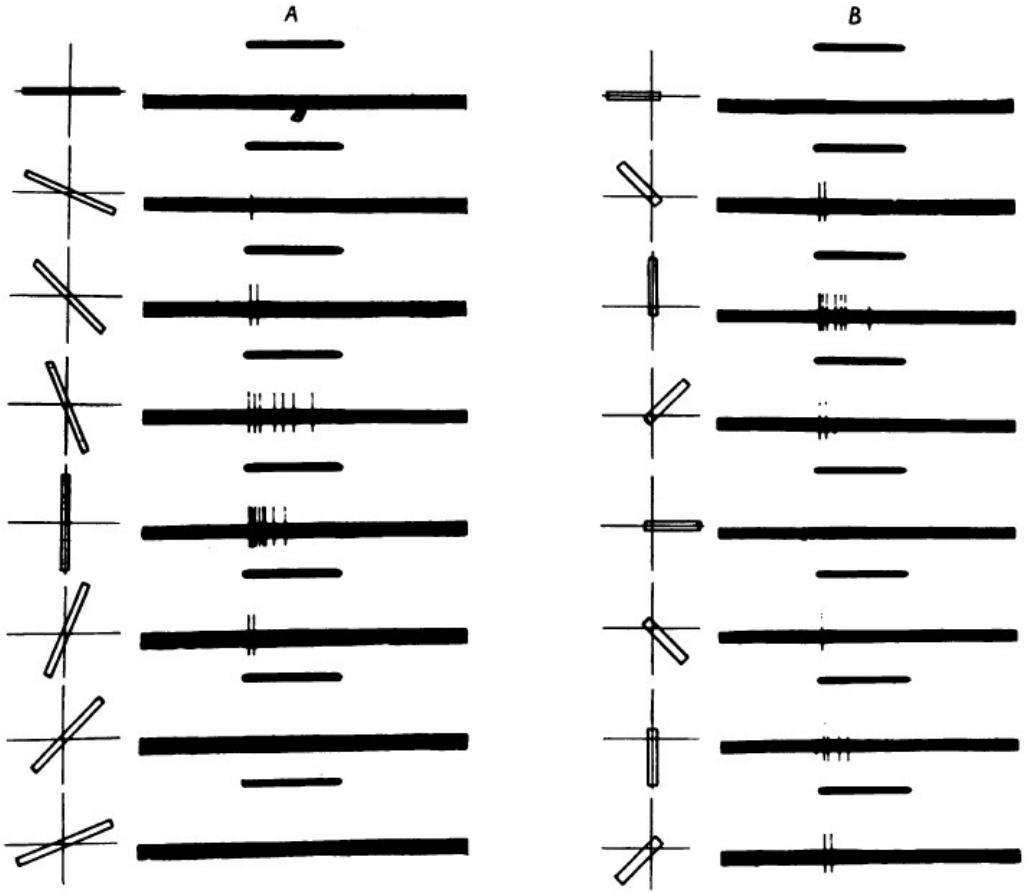
From Hubel and Wiesel



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
  - 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

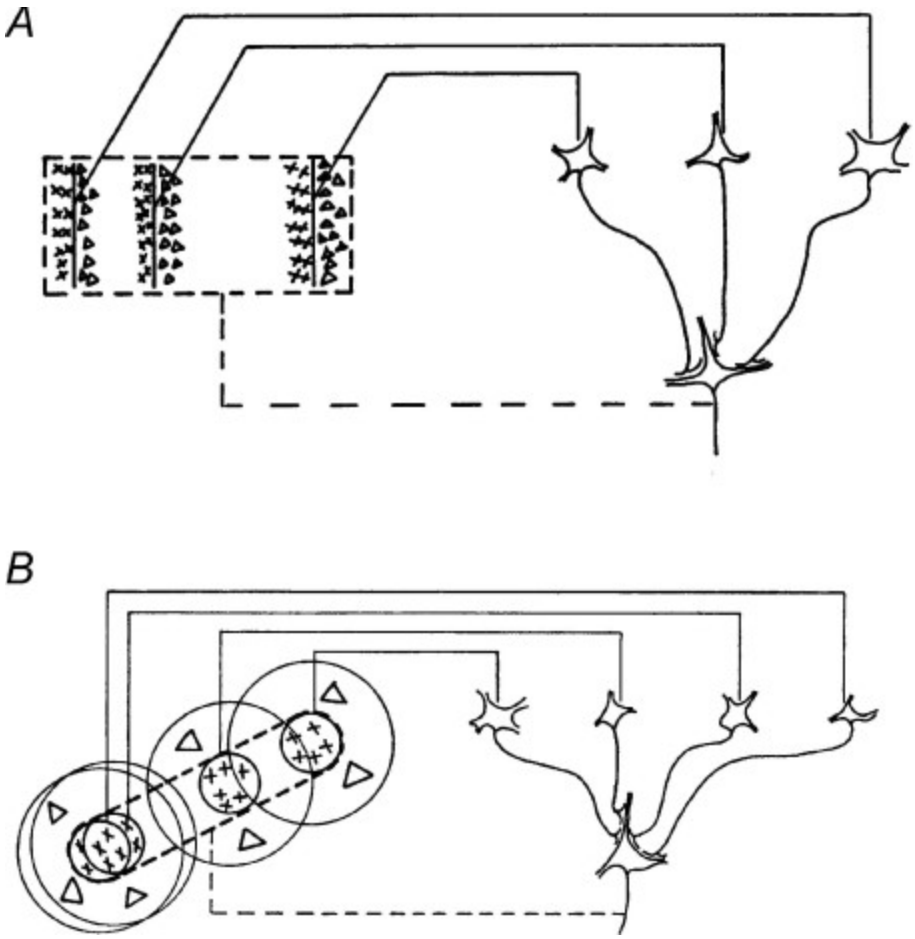
Hubel and Wiesel

- 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.”

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

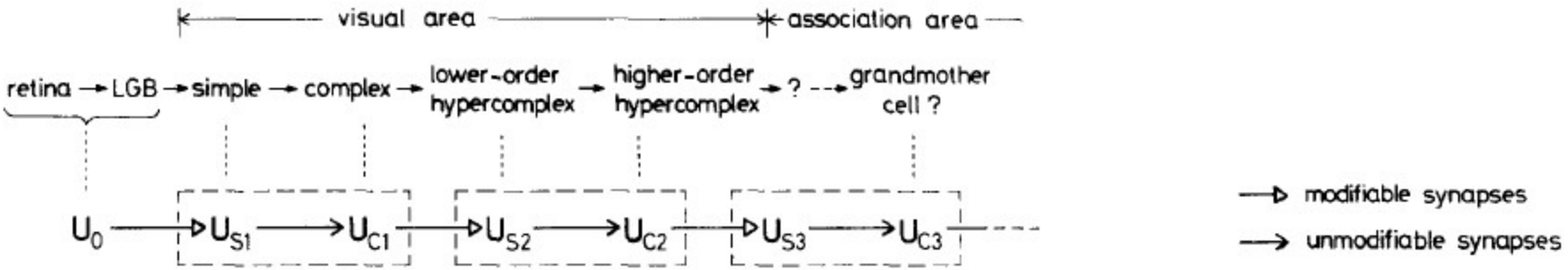


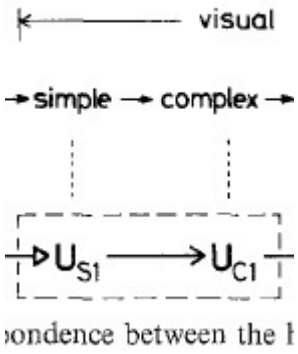
Fig. 1. Correspondence between the hierarchy model by Hubel and Wiesel, and the neural network of the neocognitron

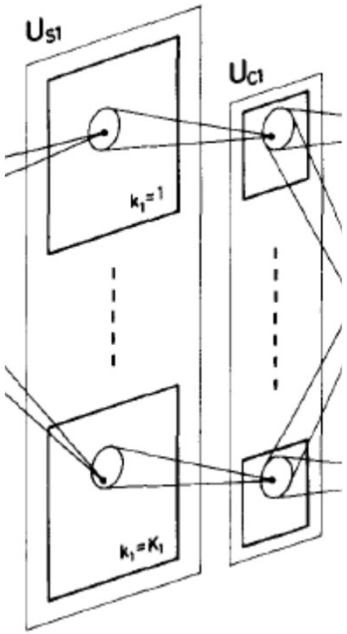
- Visual system consists of a hierarchy of modules, each comprising a layer of "S-cells" followed by a layer of "C-cells"
  - U_{Sl} is the l[th] layer of S cells, U_{Cl} is the l[th] layer of C cells

- 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 to the plane than the adjacent cells in the plane.

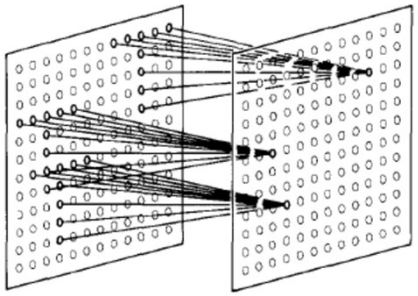
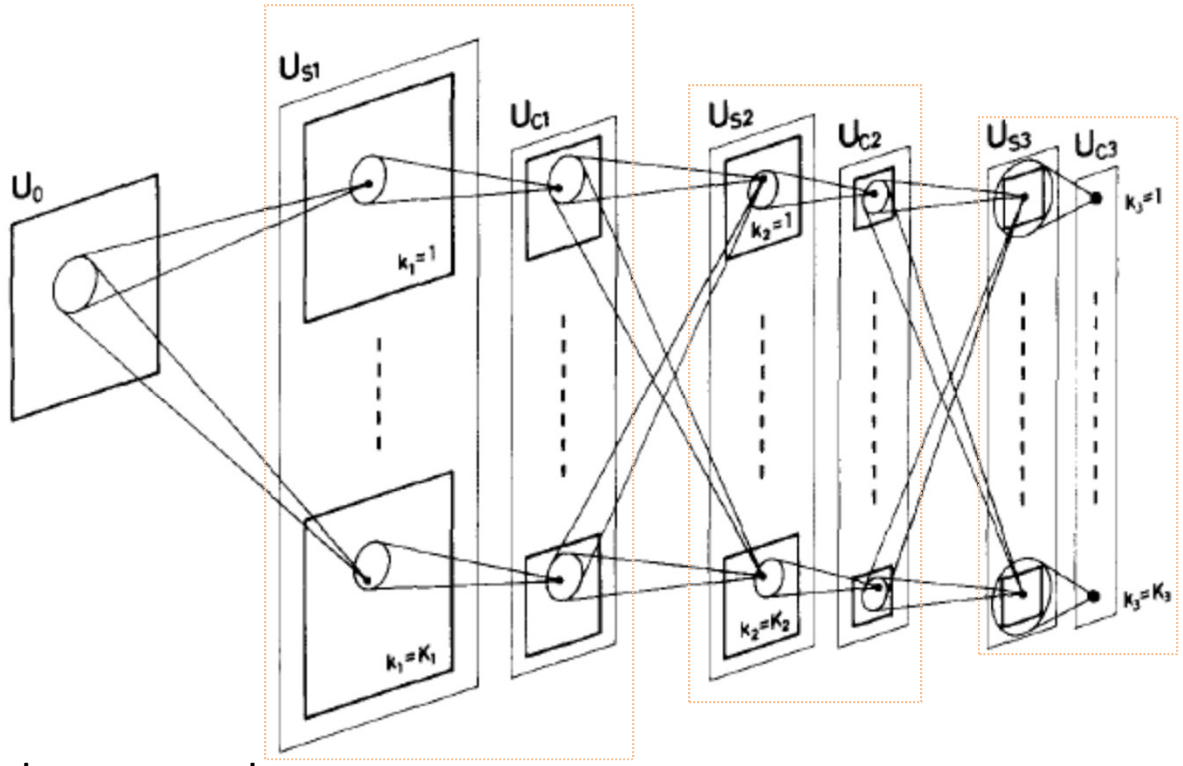


Fig. 3. Illustration showing the input interconnections to the cells within a single cell-plane

- 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

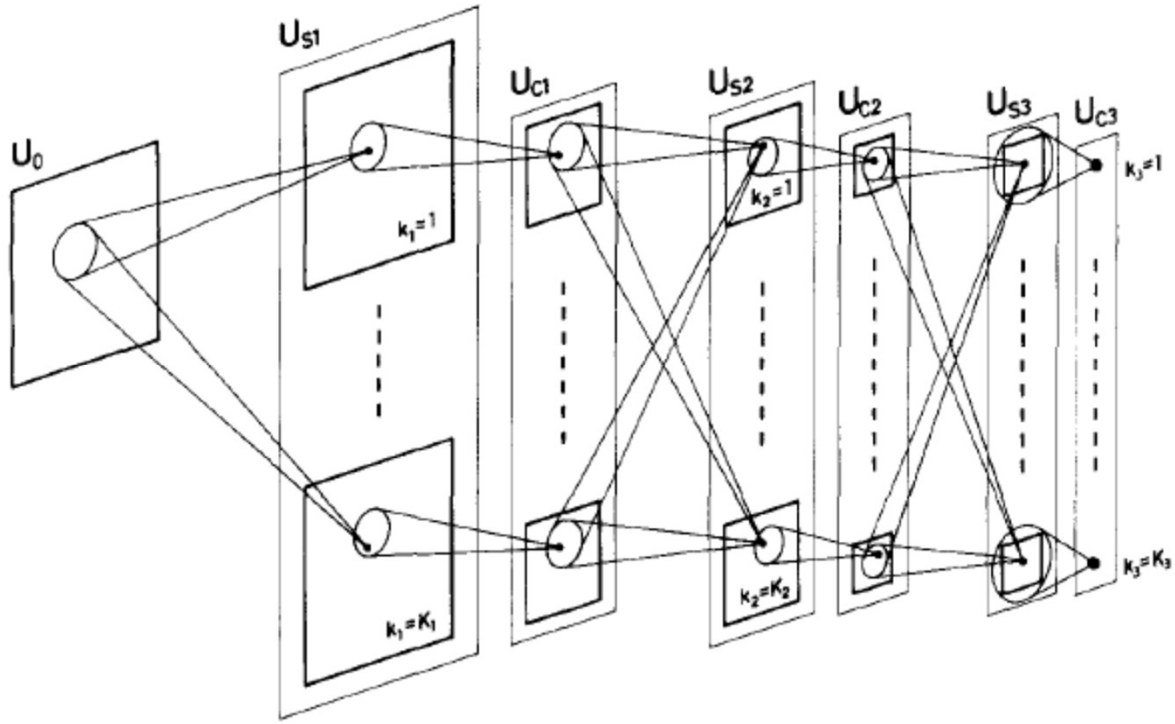
- C cells: Also RELU like, but with an inhibitory bias
  - Fires if weighted combination of S cells fires strongly enough

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

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

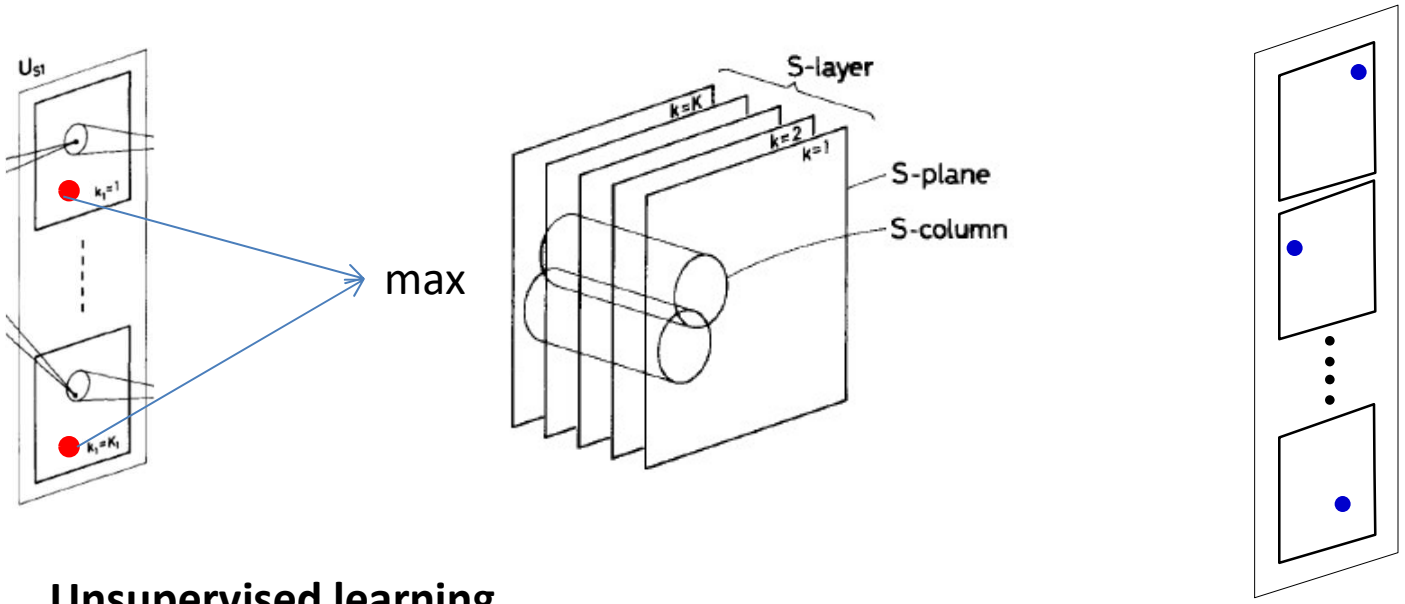
  - $\psi[x] = \varphi[x/(\alpha+x)]$

NeoCognitron



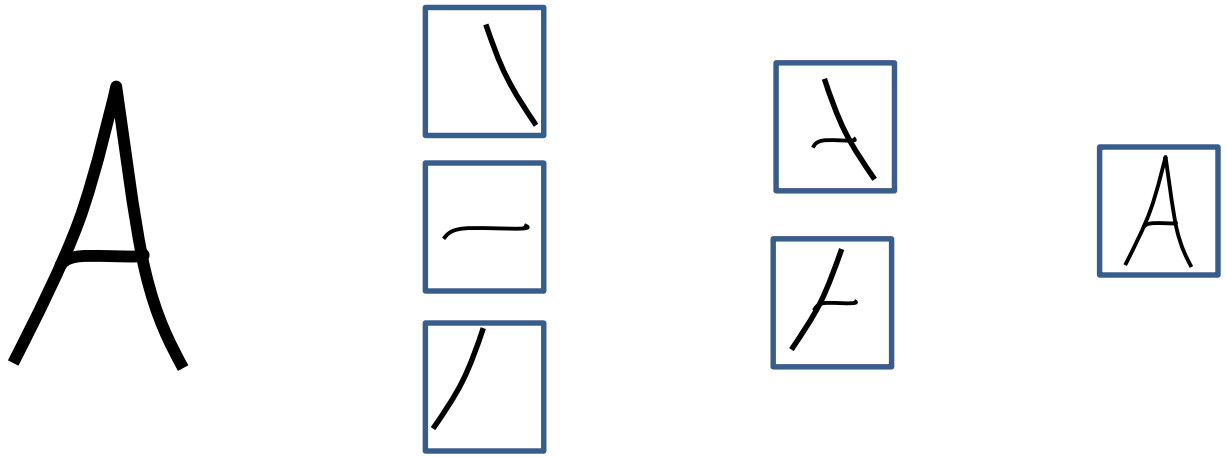
- The deeper the layer, the larger the receptive field of each neuron
  - Cell planes get smaller with layer number
  - Number of planes increases
    - i.e the number of complex pattern detectors increases with layer

Learning in the neocognitron



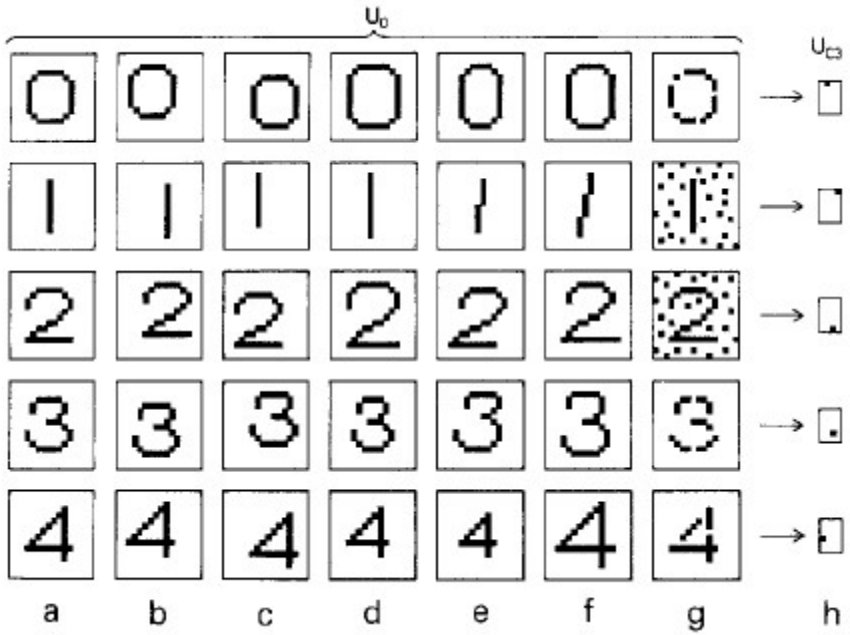
- **Unsupervised learning**
- Randomly initialize S cells, perform Hebbian learning updates in response to input
  - update = product of input and output : $\Delta w_{ij} = x_i y_j$
- Within any layer, at any position, only the maximum S from all the layers is selected for update
  - Also viewed as max-valued 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

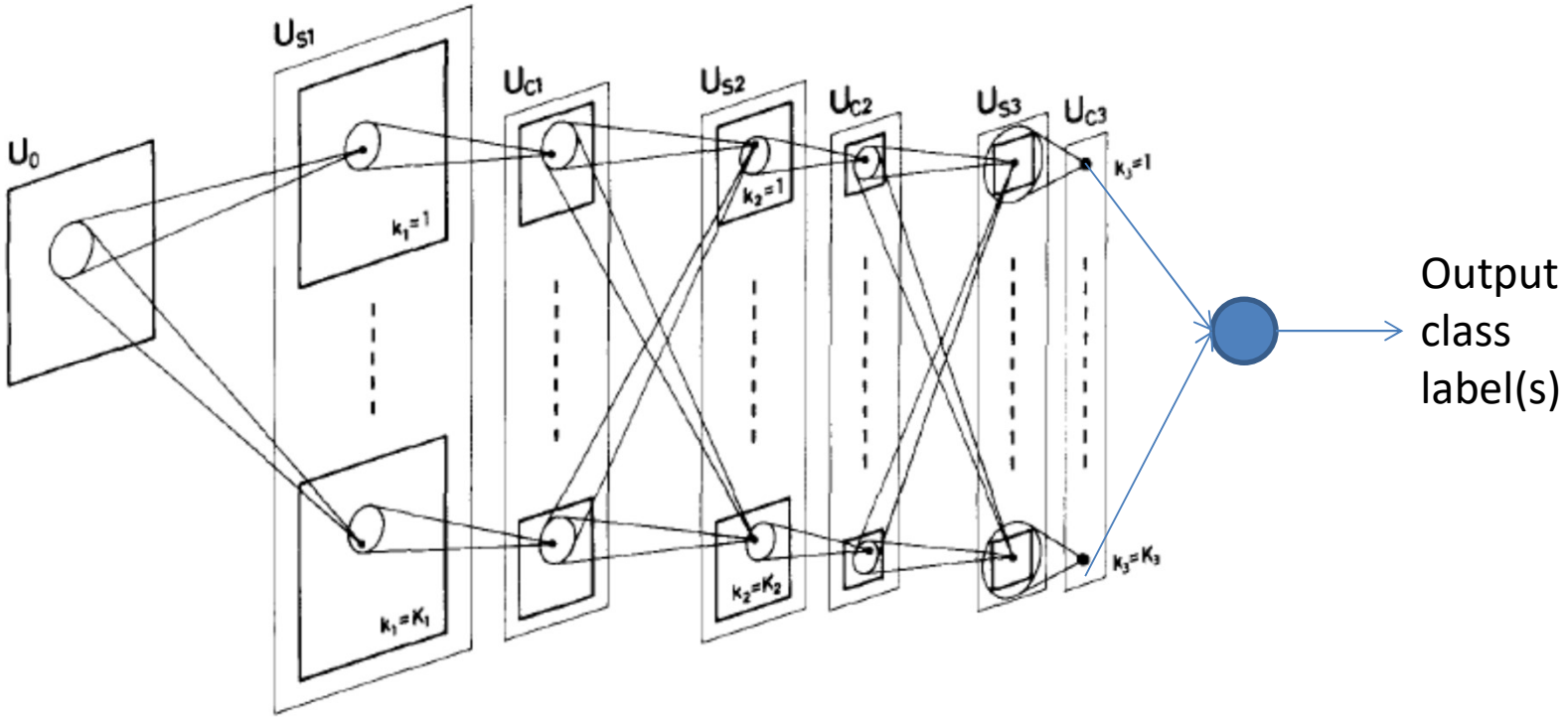


- Fukushima showed it successfully learns to cluster semantic visual concepts
  - E.g. number or characters, even in noise

Adding Supervision

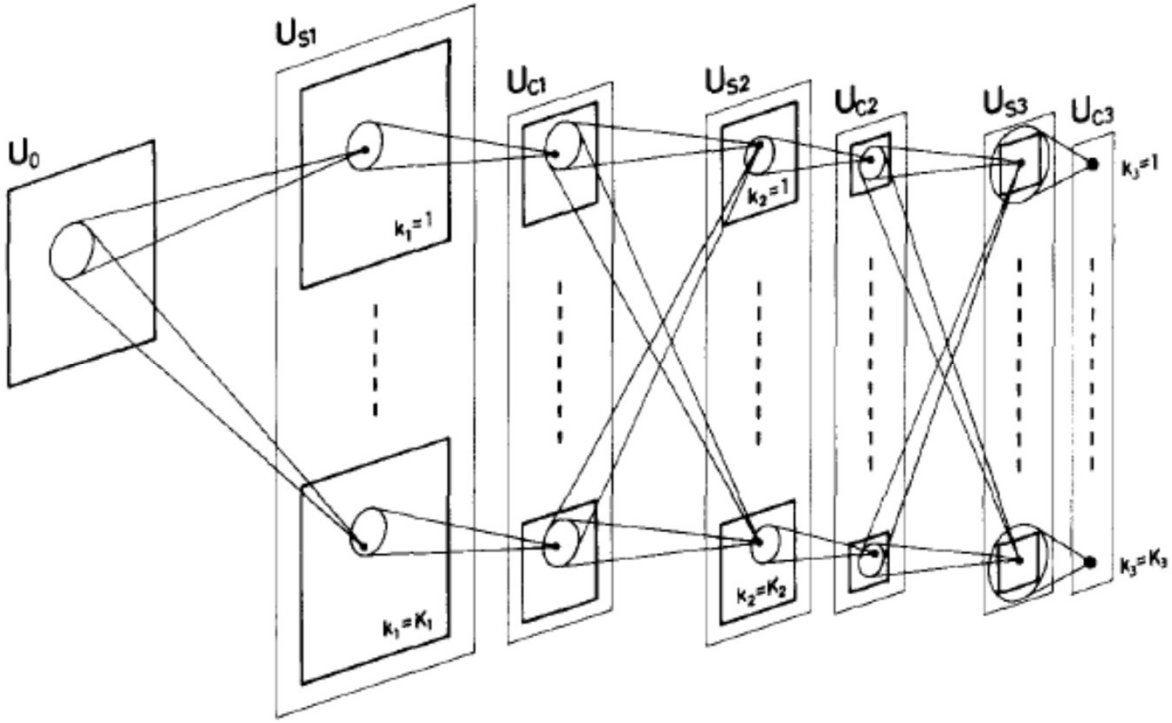
- The neocognitron is fully unsupervised
  - Semantic labels are automatically learned
- Can we add external supervision?
- Various proposals:
  - Temporal correlation: Homma, Atlas, Marks, '88
  - TDNN: Lang, Waibel et. al., 1989, '90
- Convolutional neural networks: LeCun

Supervising the neocognitron



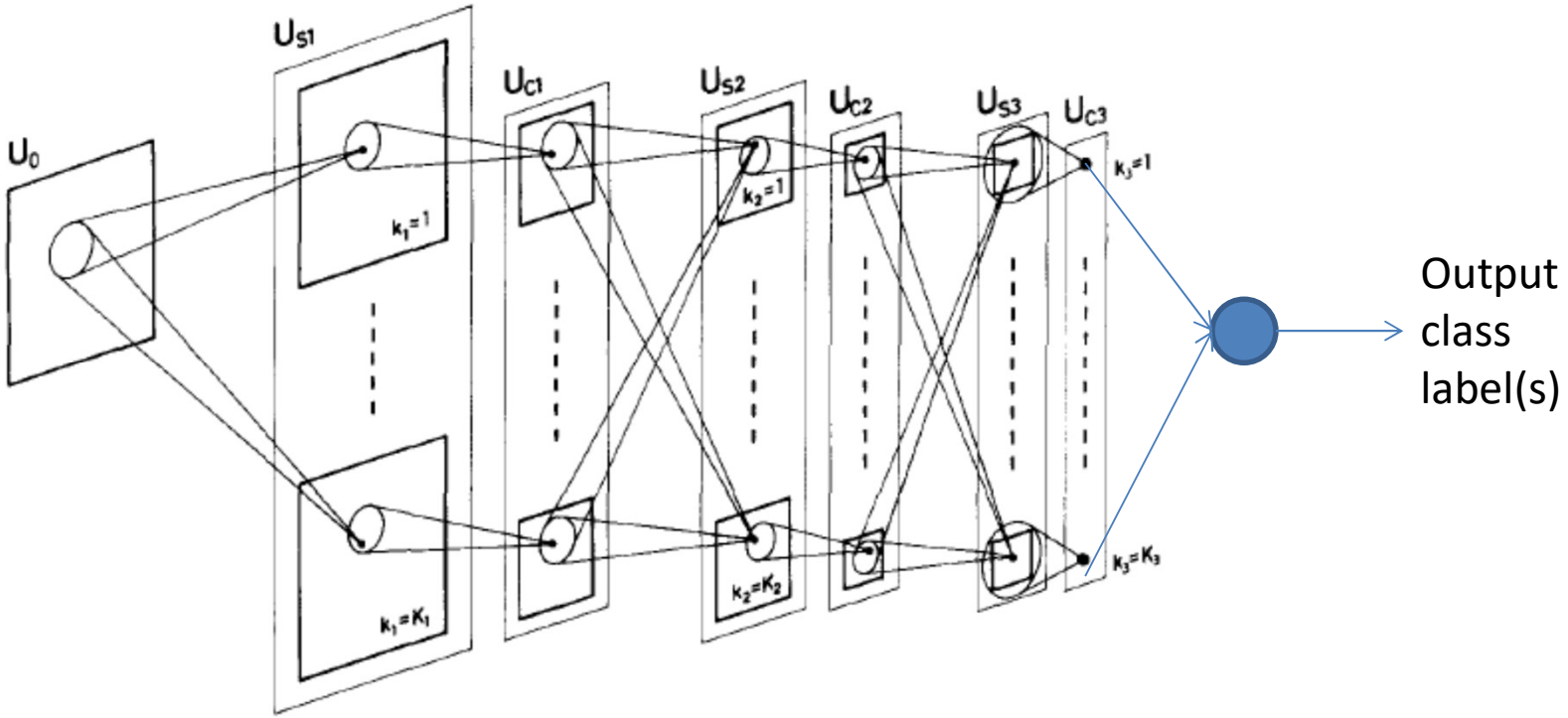
- Add an extra decision layer after the final C layer
  - Produces a class-label output
- We now have a fully feed forward MLP with shared parameters
  - All the S-cells within an S-plane have the same weights
- Simple backpropagation can now train the S-cell weights in every plane of every layer
  -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

Scanning vs. multiple filters



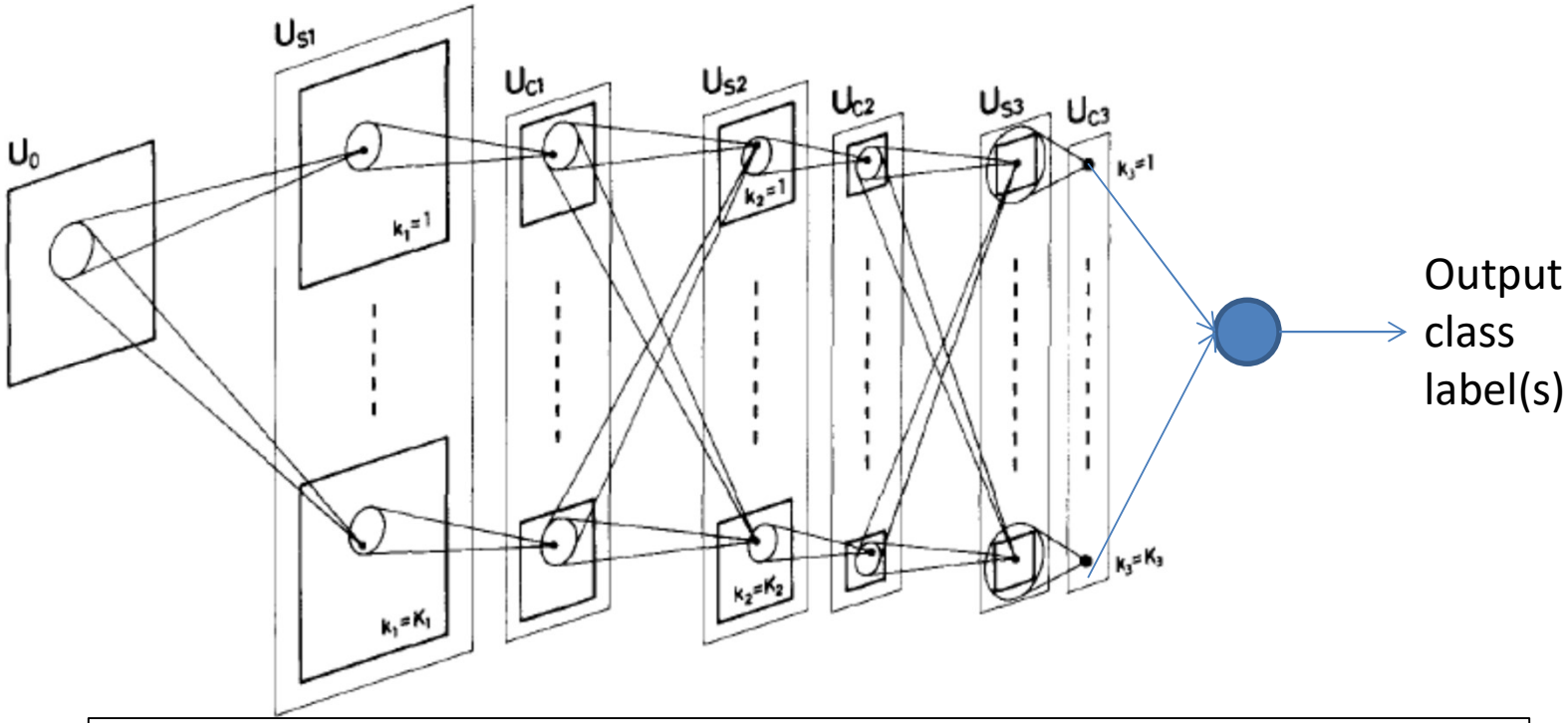
- **Note**: The original Neocognitron actually uses many identical copies of a neuron in each S and C plane

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

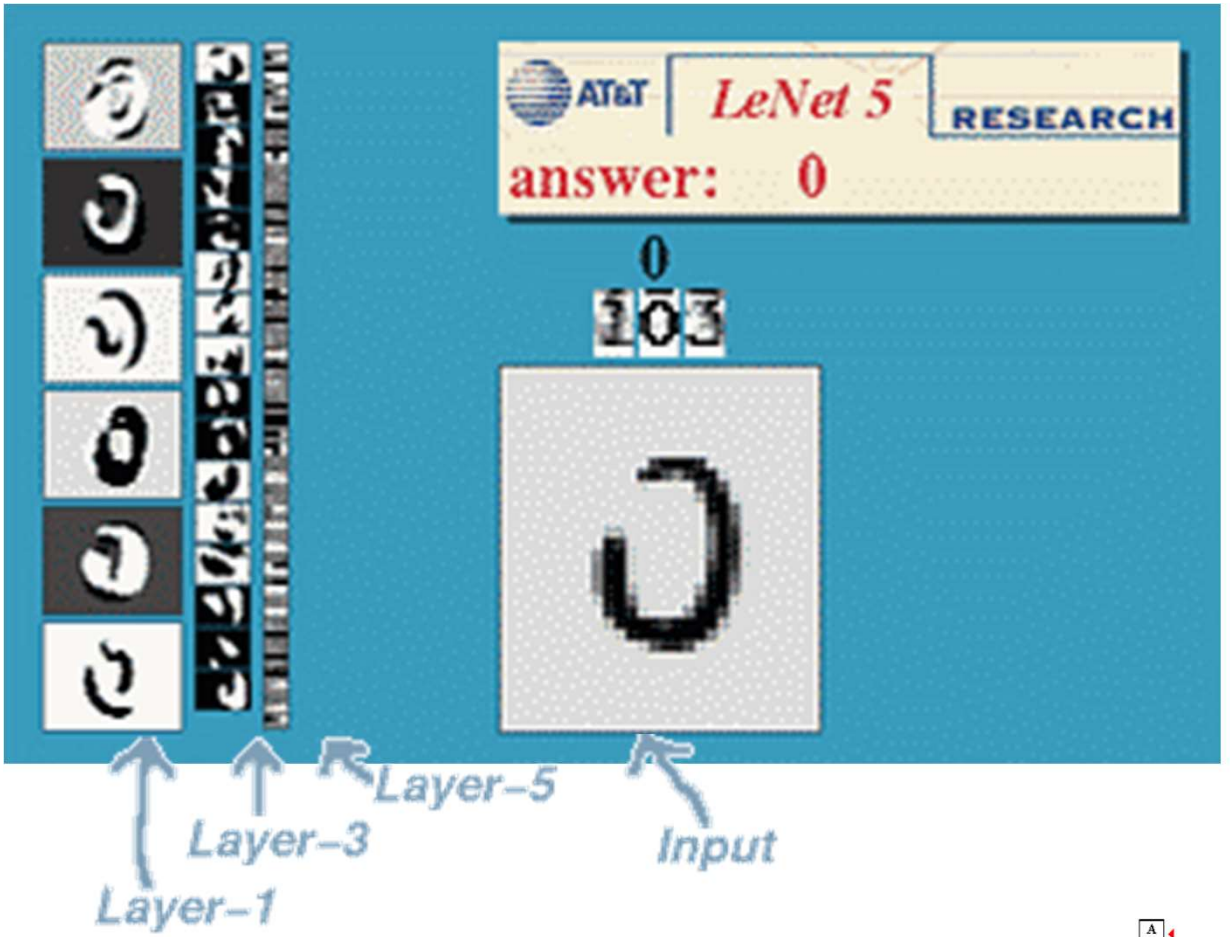


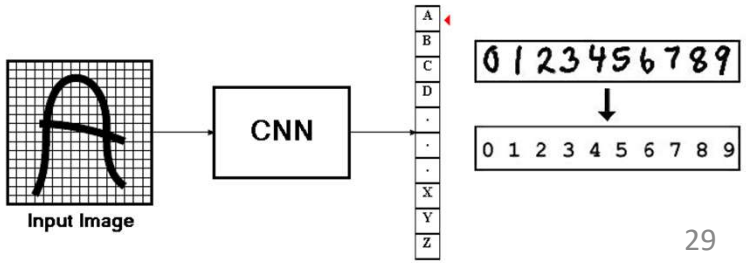
$$U_{S,l,n}(i,j) = \sigma\left(\sum_{p}\sum_{k=1}^{K_l}\sum_{l=1}^{K_l} w_{S,l,n}(p,k,l)\,U_{C,l-1,p}(i+l-1,j+k-1)\right)$$

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

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

Convolutional Neural Networks





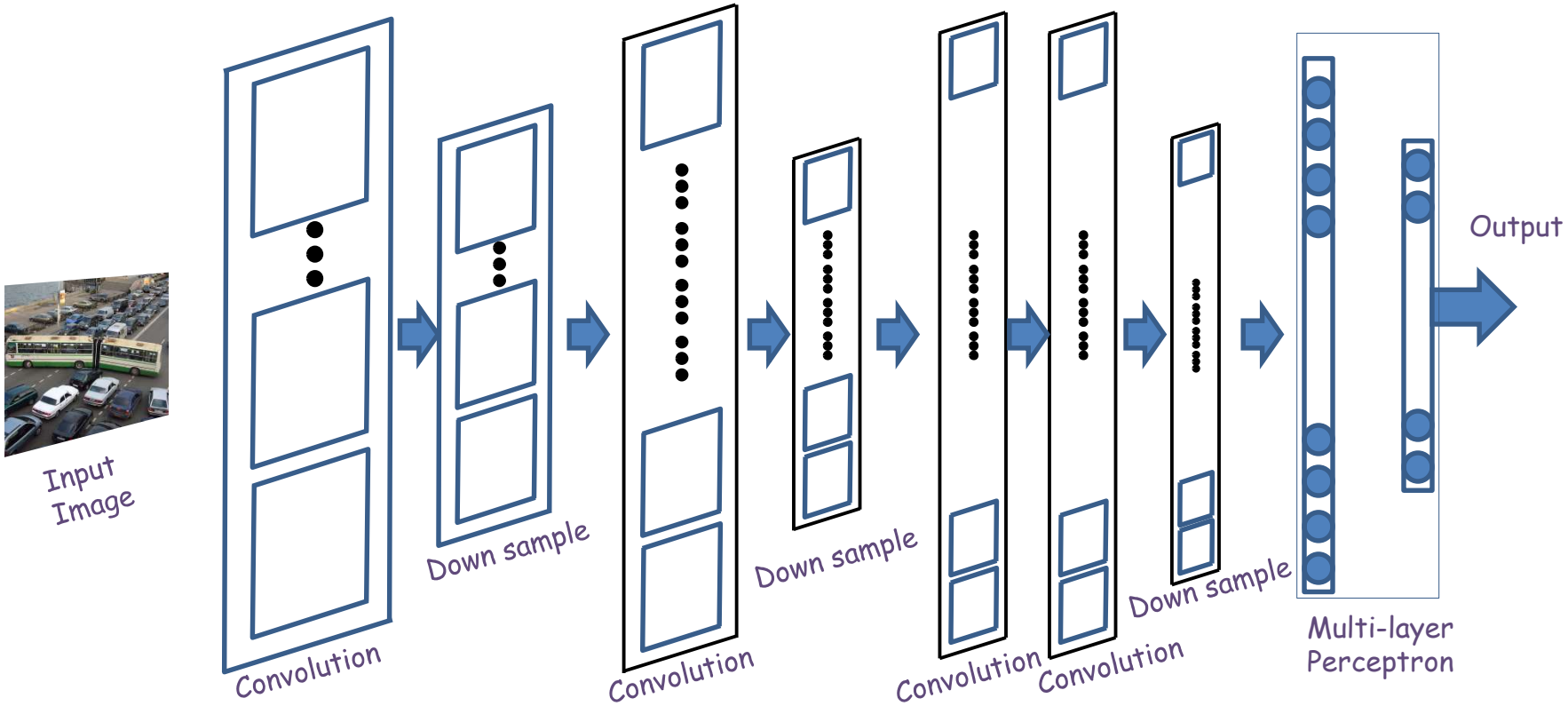
Story so far

- 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

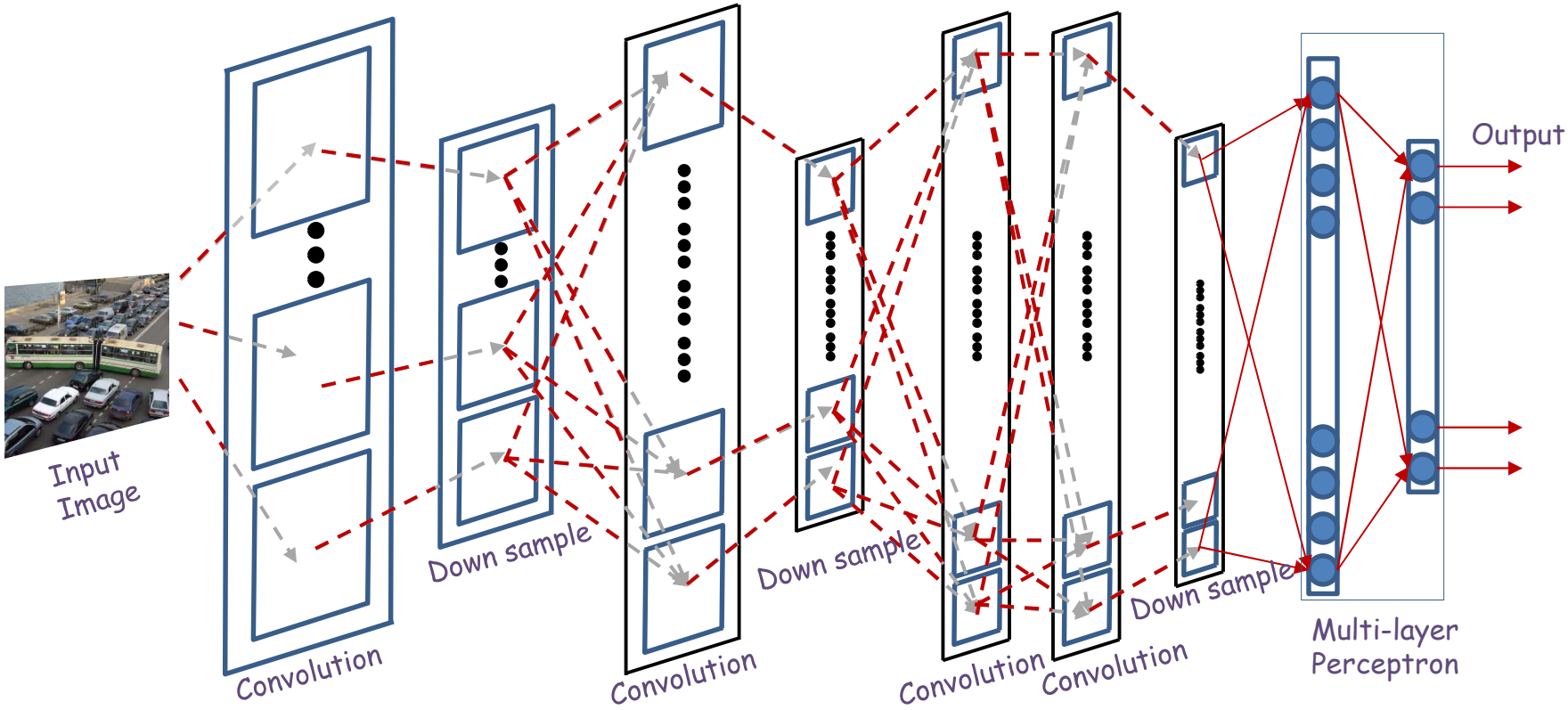
- LeCun's LeNet added external supervision to the neocognitron
  - S planes of cells with identical response are modelled by a scan (convolution) over image planes by a single neuron
  - C planes are emulated by cells that perform a max over groups of S cells
    - Reducing the size of the S planes
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

The general architecture of a convolutional neural network



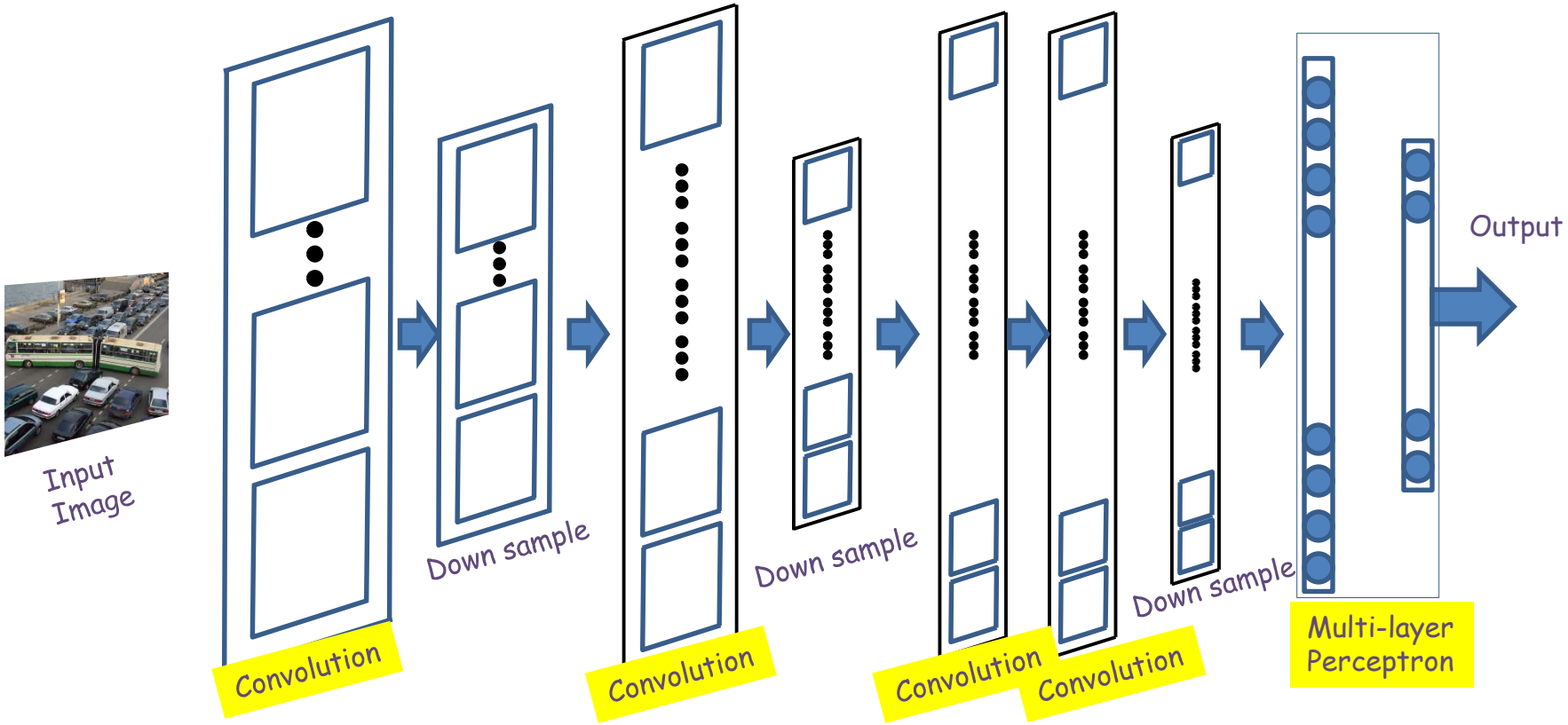
- A convolutional neural network comprises "convolutional" and "downsampling" layers
  - Convolutional layers comprise neurons that scan their input for patterns
    - Correspond to S planes
  - Downsampling layers perform max operations on groups of outputs from the convolutional layers
    - Correspond 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



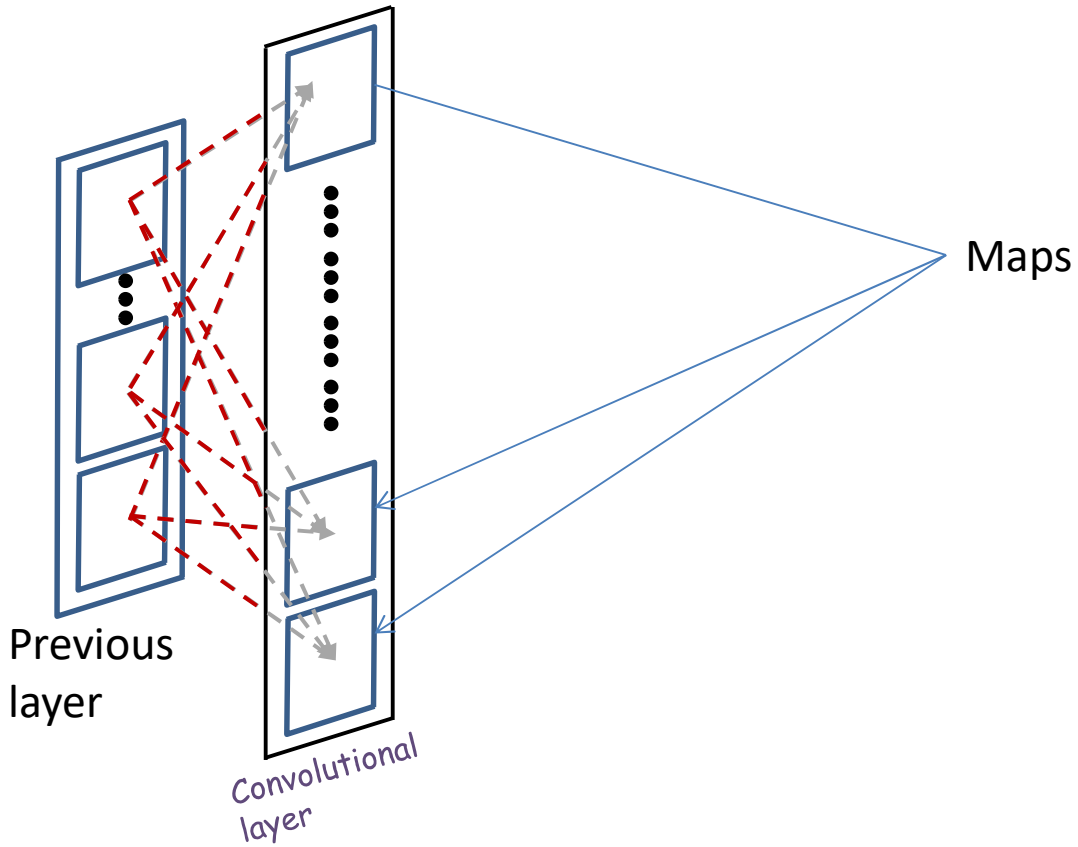
- A convolutional neural network comprises of "convolutional" and "downsampling" layers
  - The two may occur in any sequence, but typically they alternate
- Followed by an MLP with one or more layers

The general architecture of a convolutional neural network



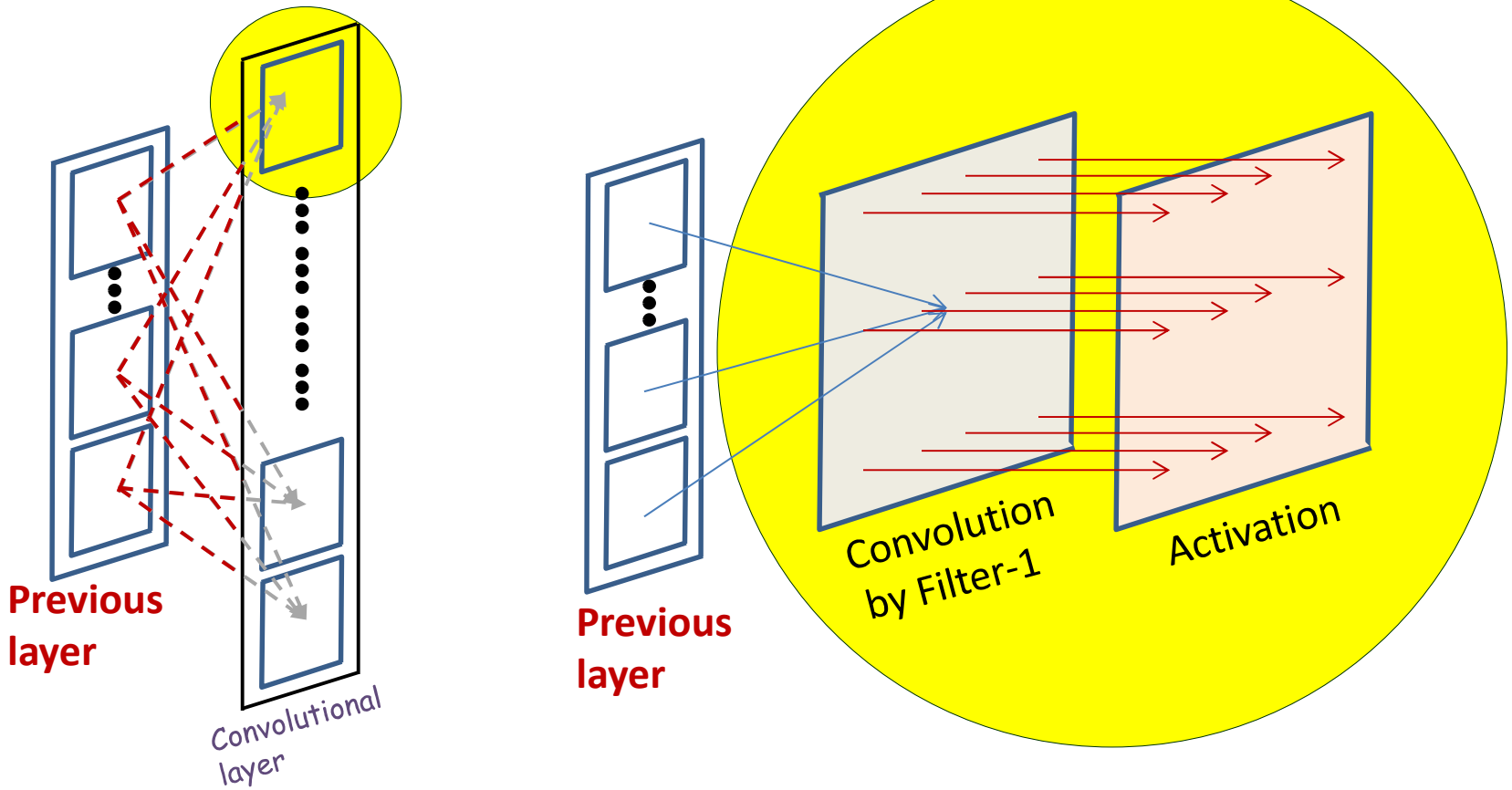
- **Convolutional layers and the MLP are *learnable***
  - Their parameters must be learned from training data for the target classification task
- Down-sampling layers are fixed and generally not learnable

A convolutional layer



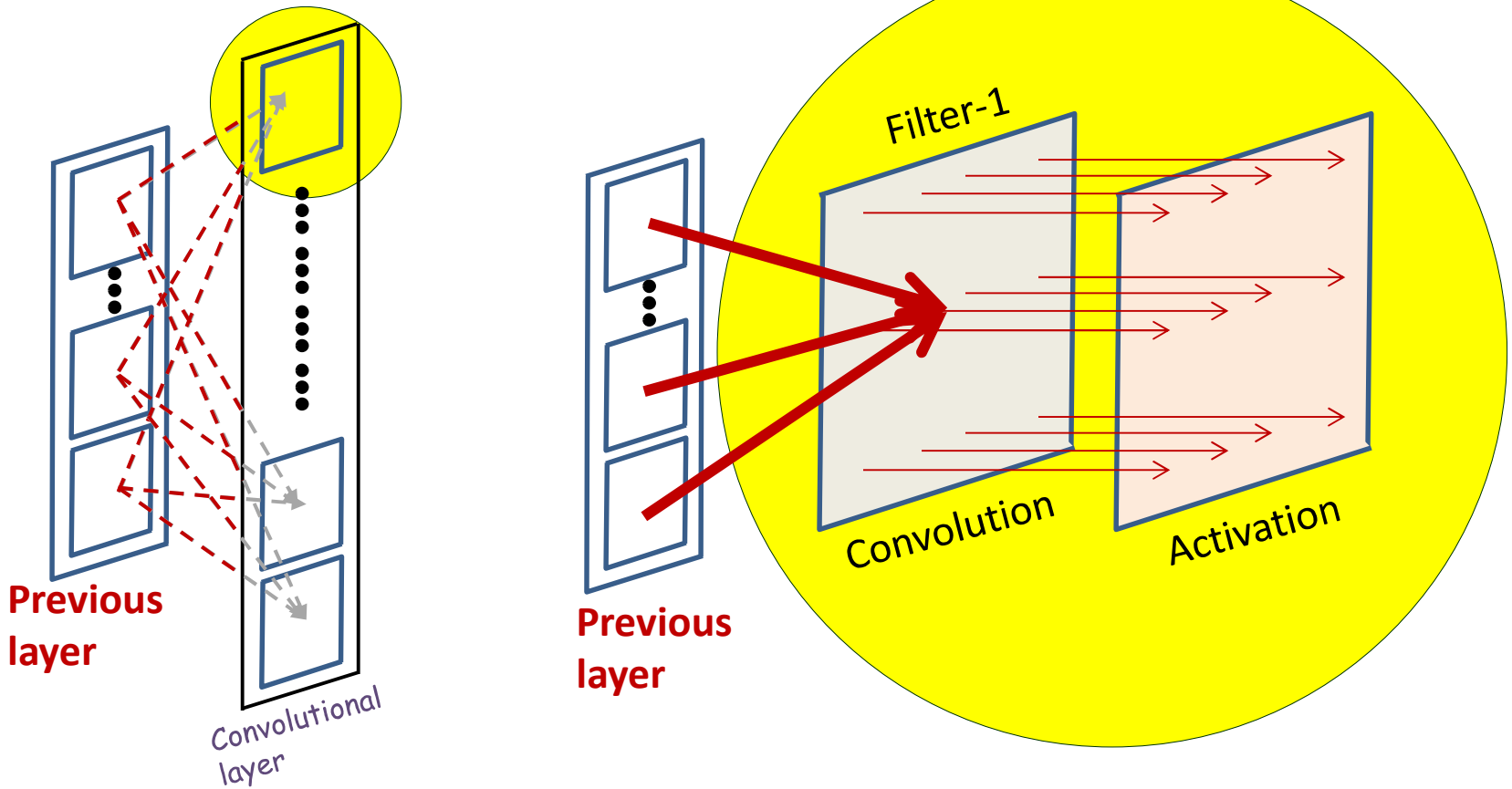
- A convolutional layer comprises of a series of "maps"
  - Corresponding the "S-planes" in the Neocognitron
  - Variously called feature maps or activation maps

A convolutional layer



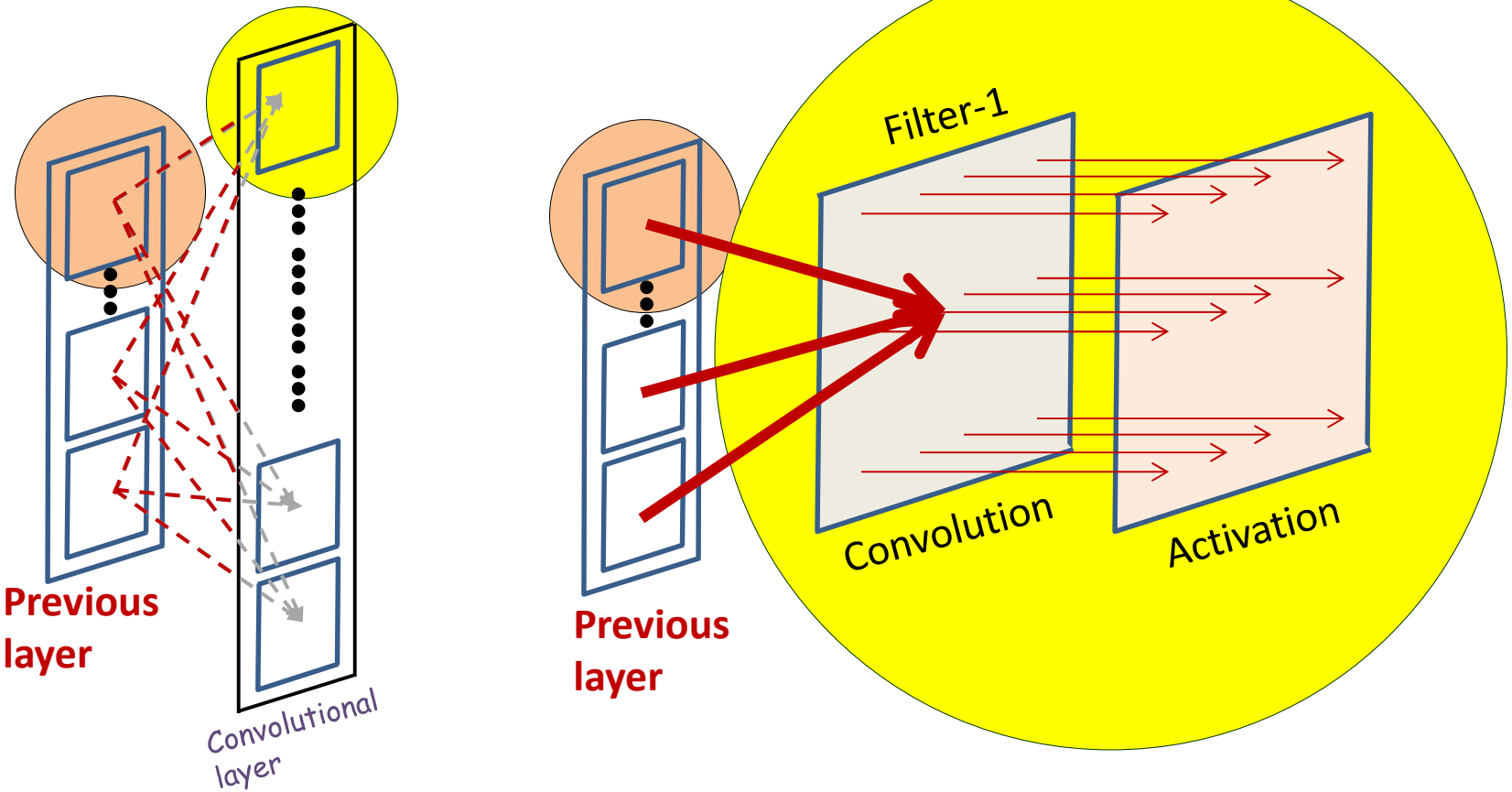
- Each activation map has two components
  - An *affine* map, obtained by *convolution* over maps in the previous layer
    - Each affine map has, associated with it, a ***learnable filter***
  - An *activation* that operates on the output of the convolution

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

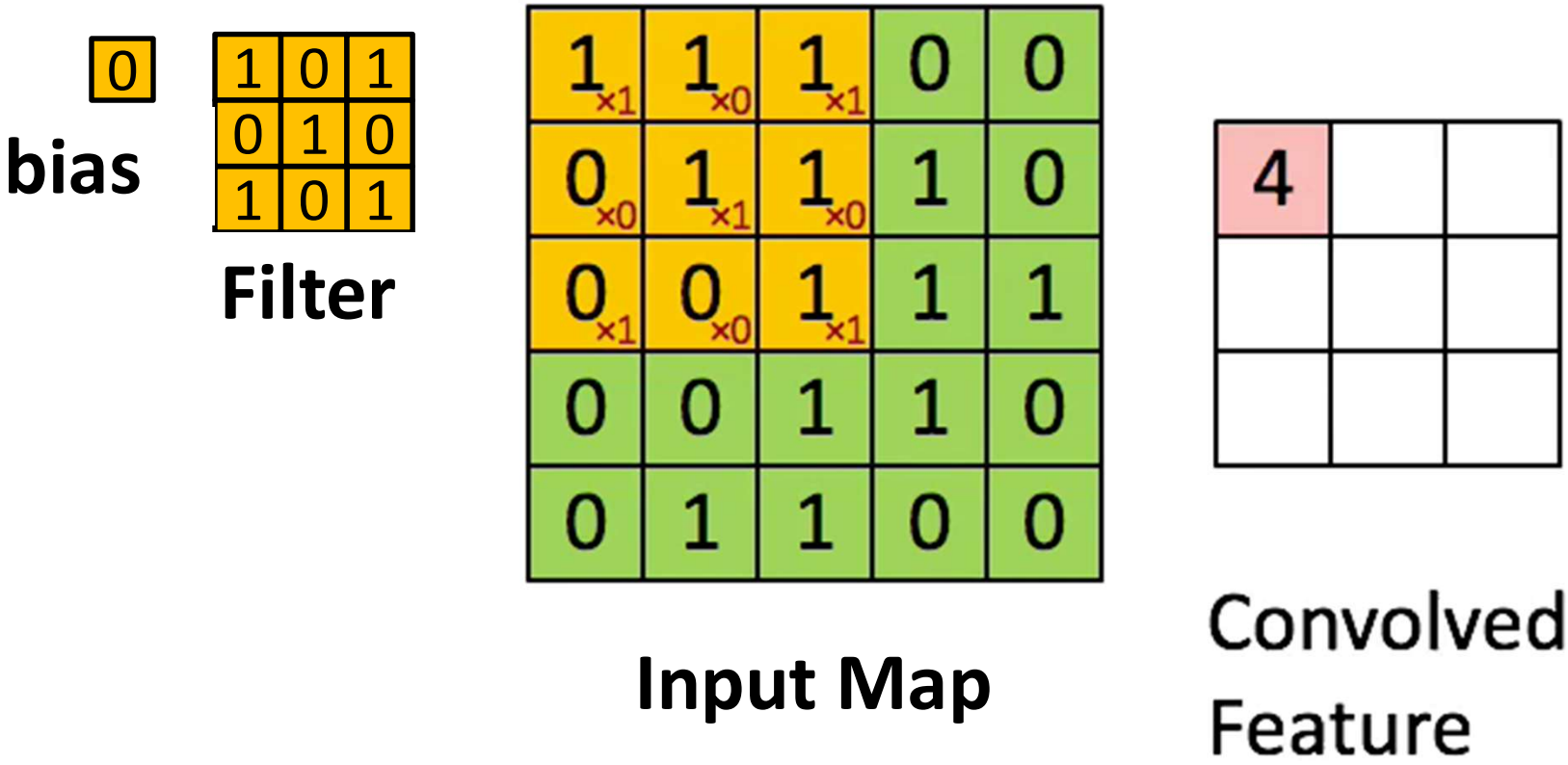
1	0	1
0	1	0
1	0	1

bias

0

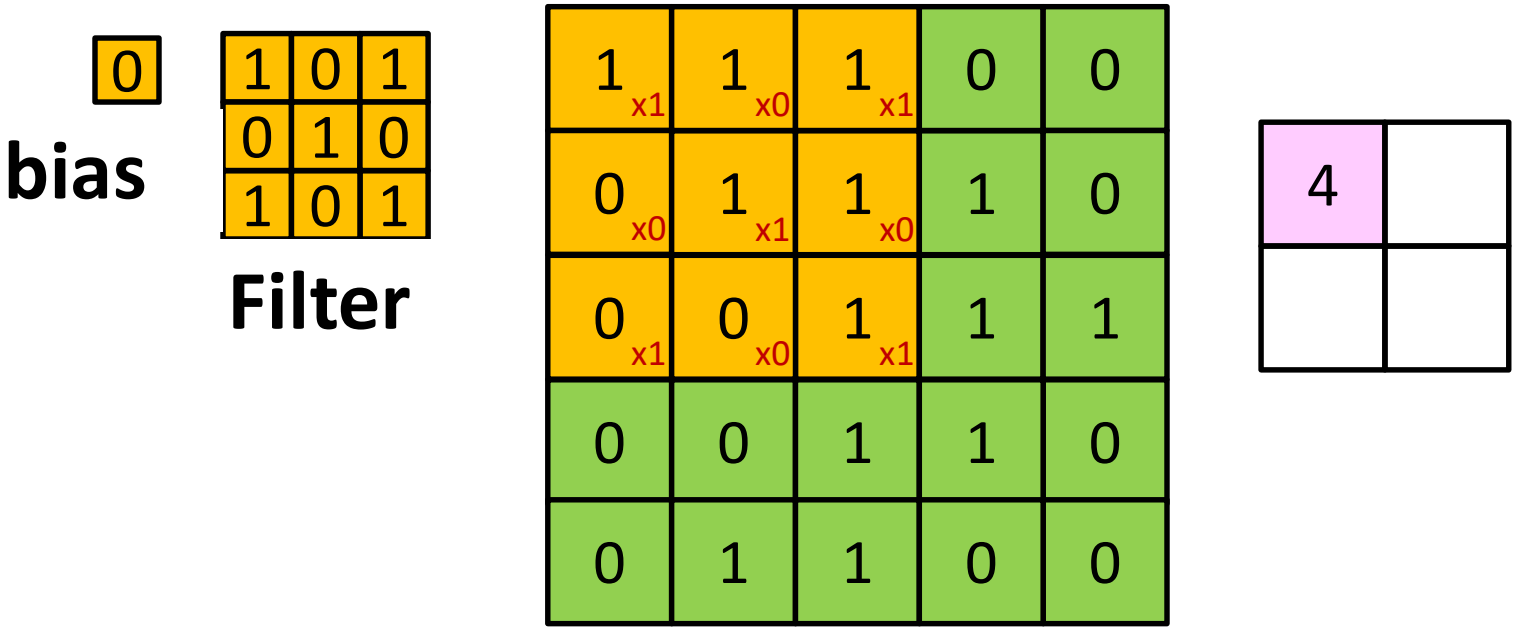
- Scanning an image with a “filter”
  - Note: a filter is really just a perceptron, with weights and a bias

What is a convolution



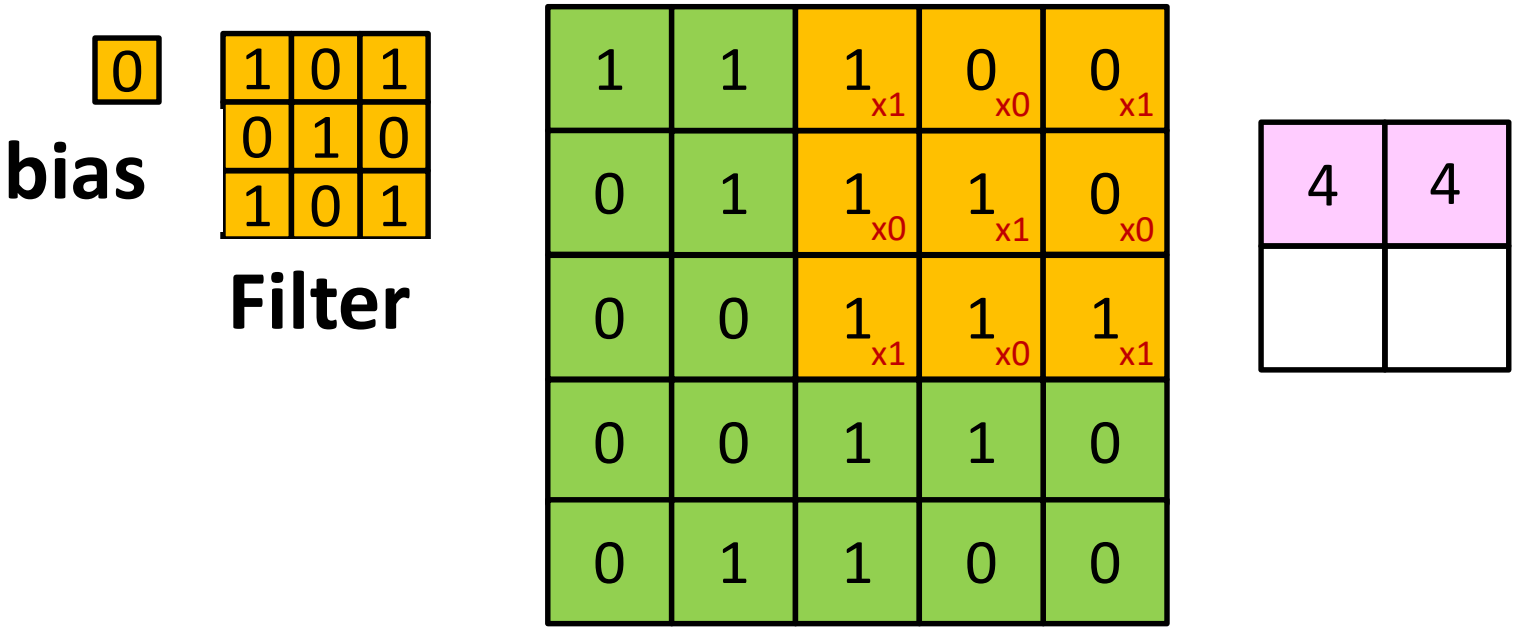
- Scanning an image with a “filter”
  - At each location, the “filter and the underlying map values are multiplied component wise, and the products are added along with the bias

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



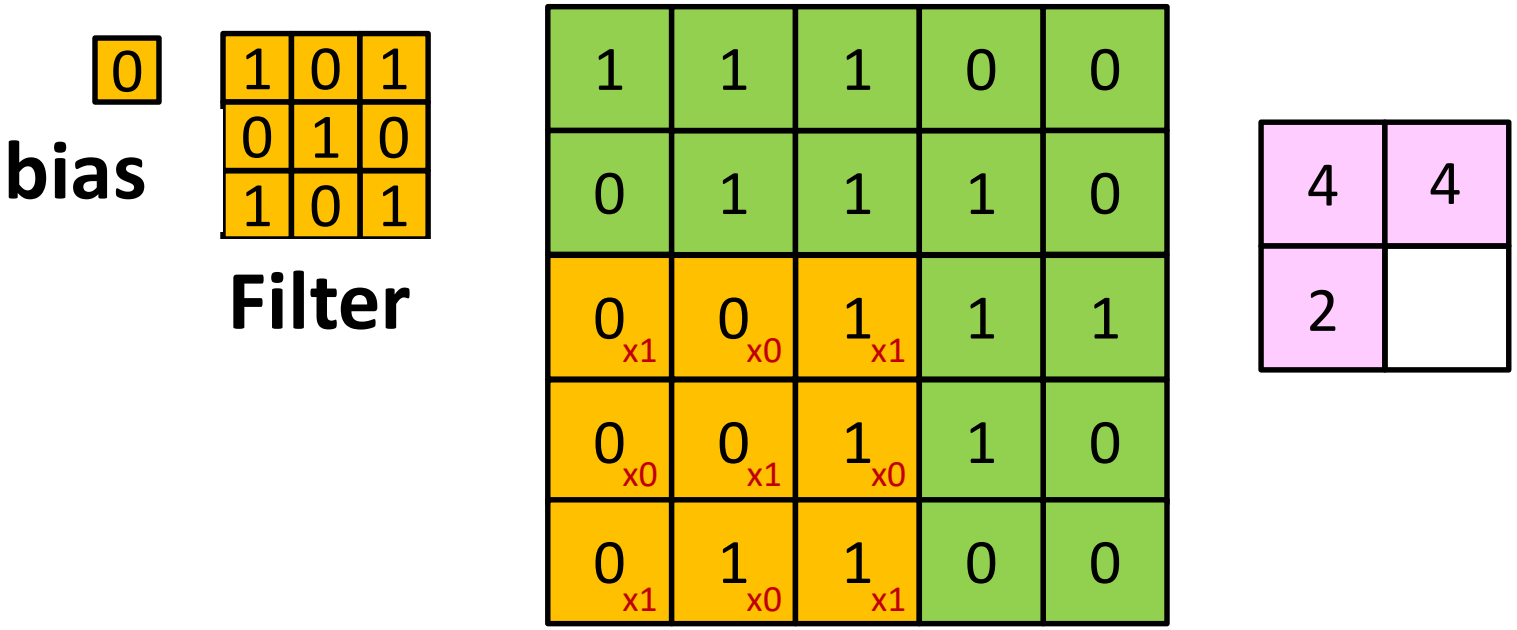
- Scanning an image with a "filter"
  - The filter may proceed by *more* than 1 pixel at a time
  - E.g. with a "stride" of *two* pixels per shift

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



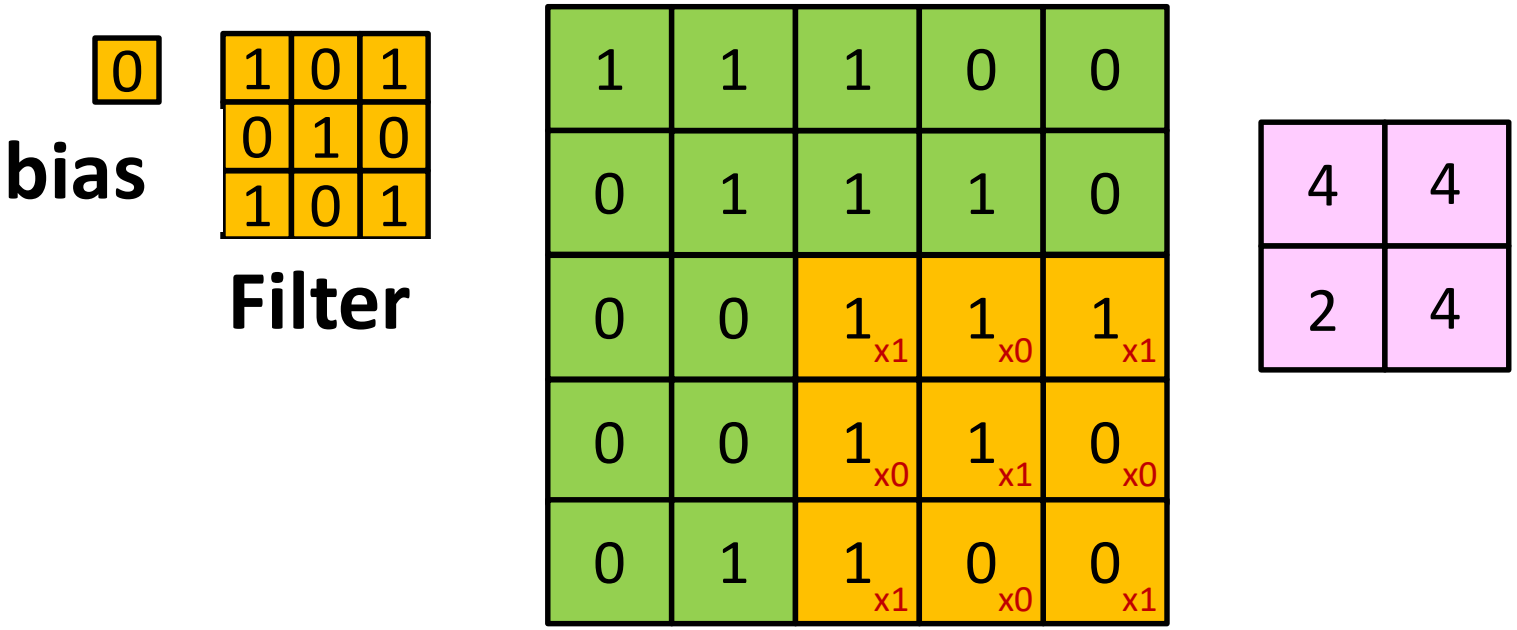
- Scanning an image with a "filter"
  - The filter may proceed by *more* than 1 pixel at a time
  - E.g. with a "hop" of *two* pixels per shift

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



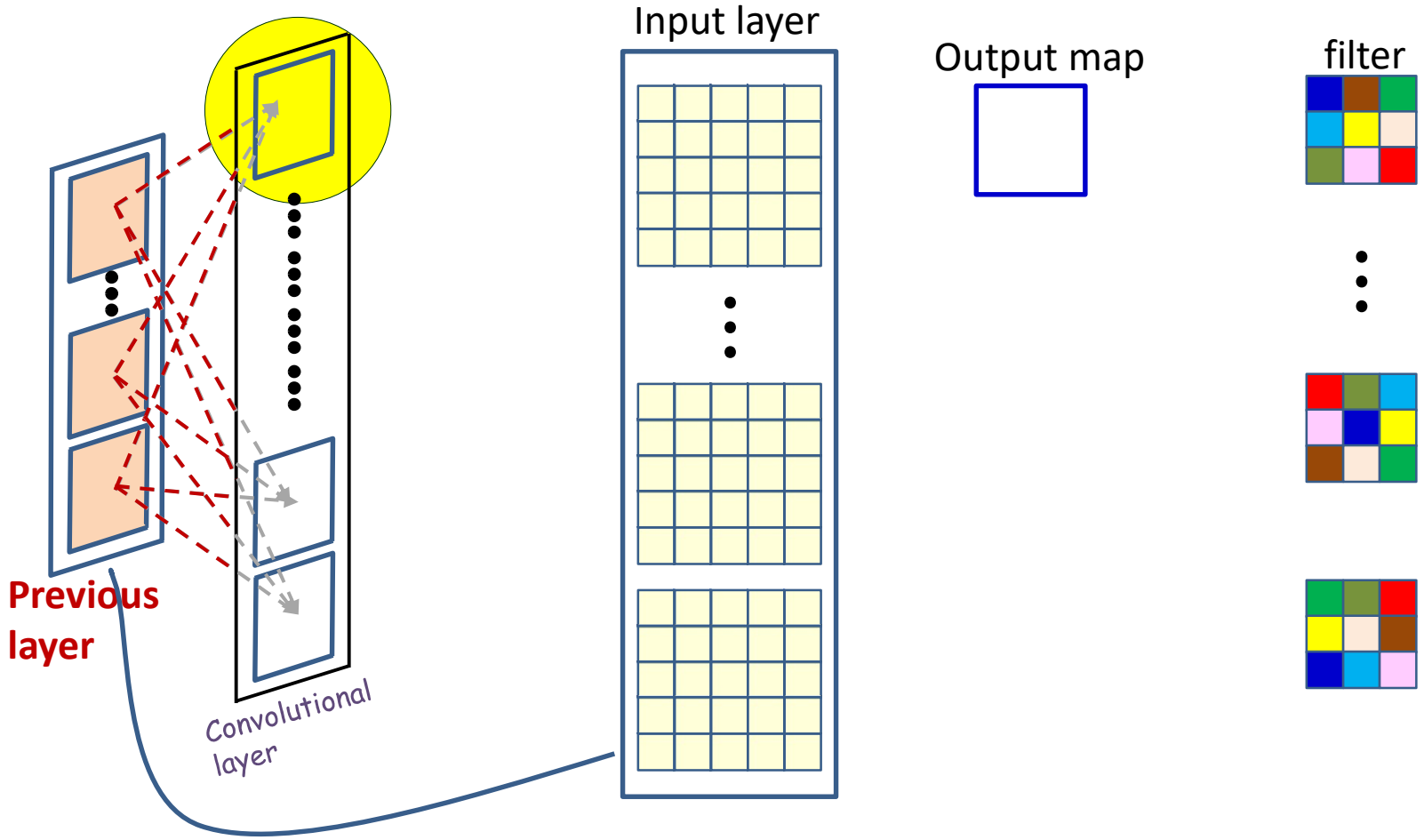
- Scanning an image with a "filter"
  - The filter may proceed by *more* than 1 pixel at a time
  - E.g. with a "hop" of *two* pixels per shift

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

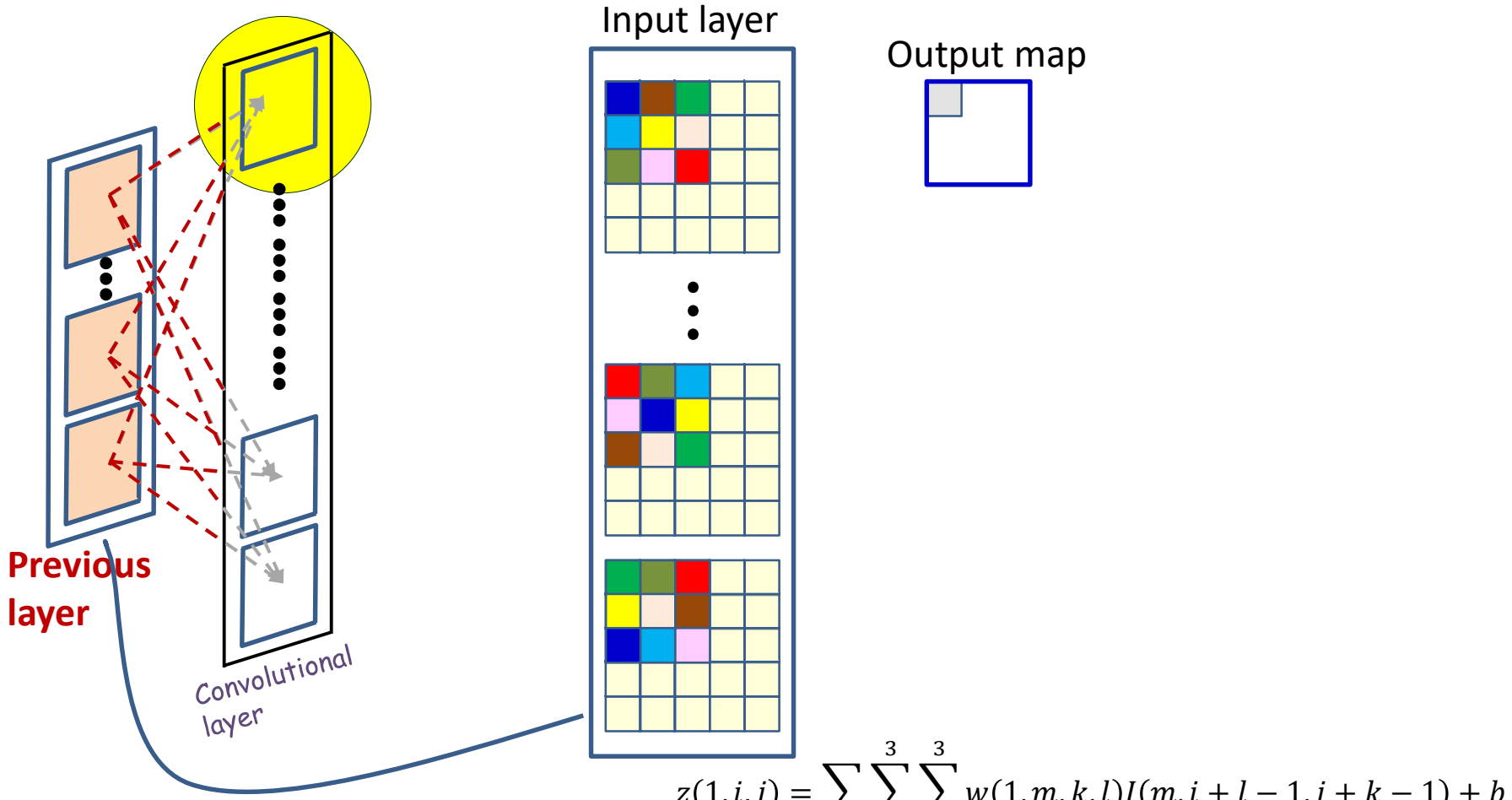
What really happens



- 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

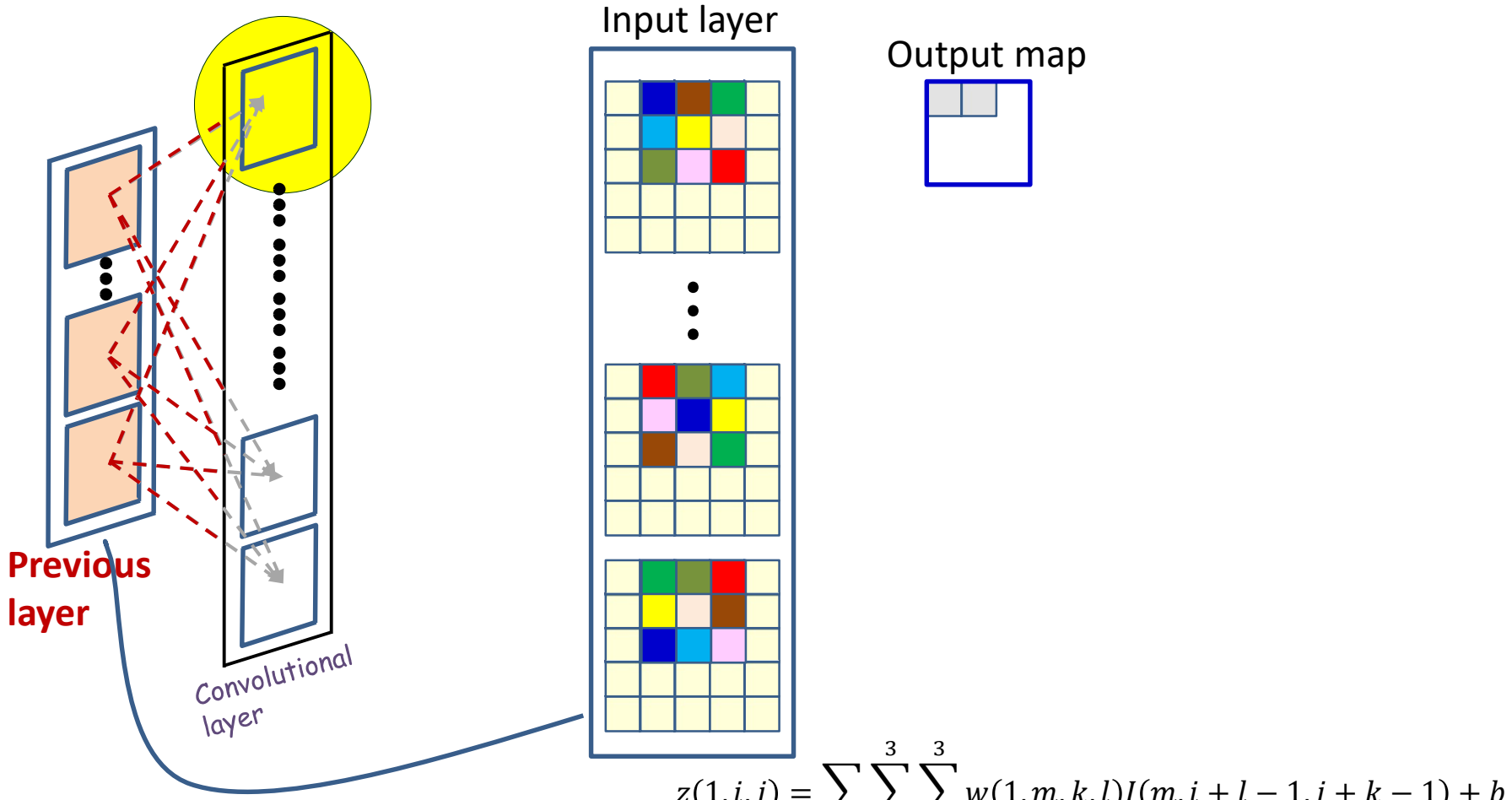


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- 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

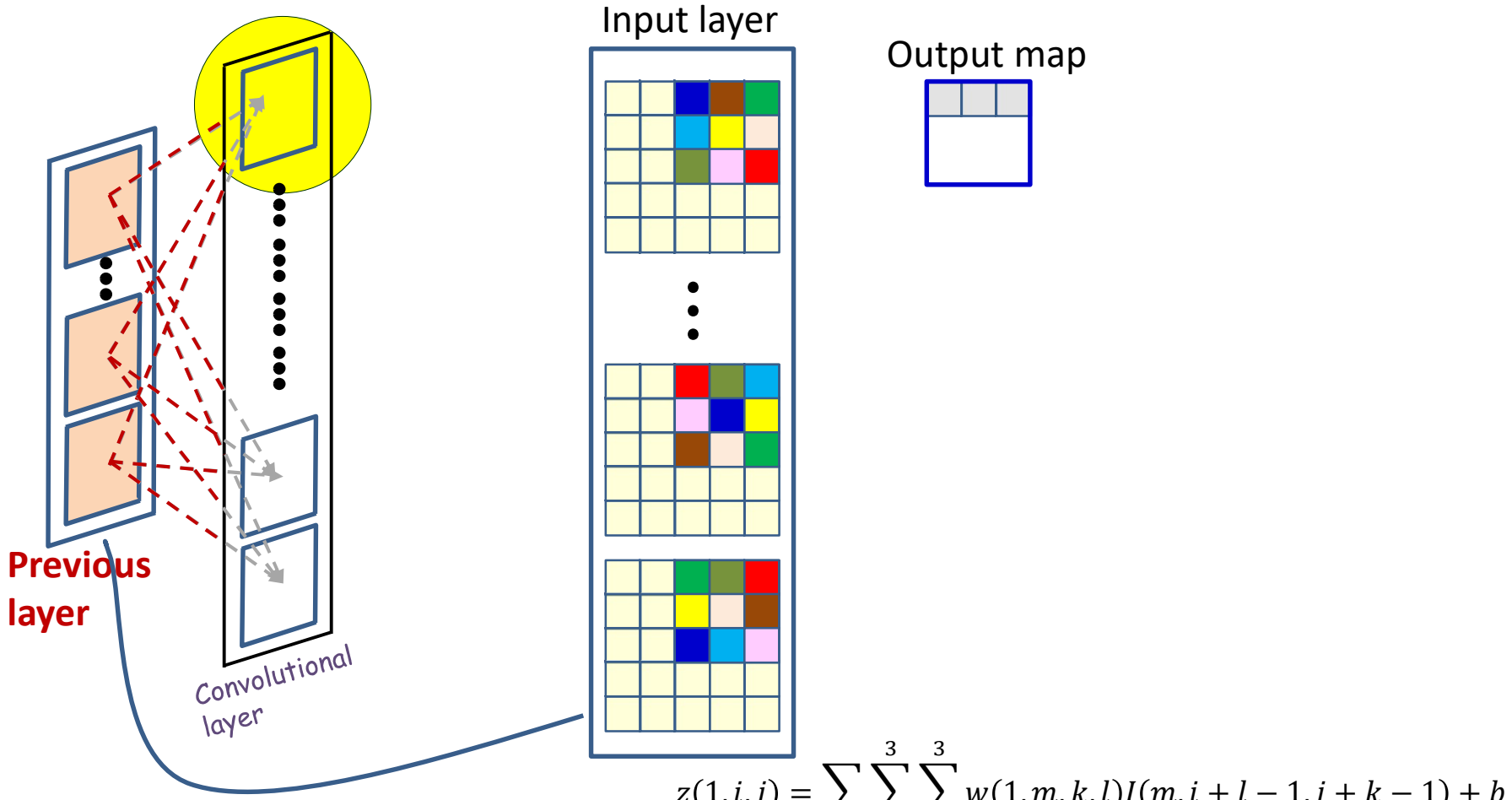


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- 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

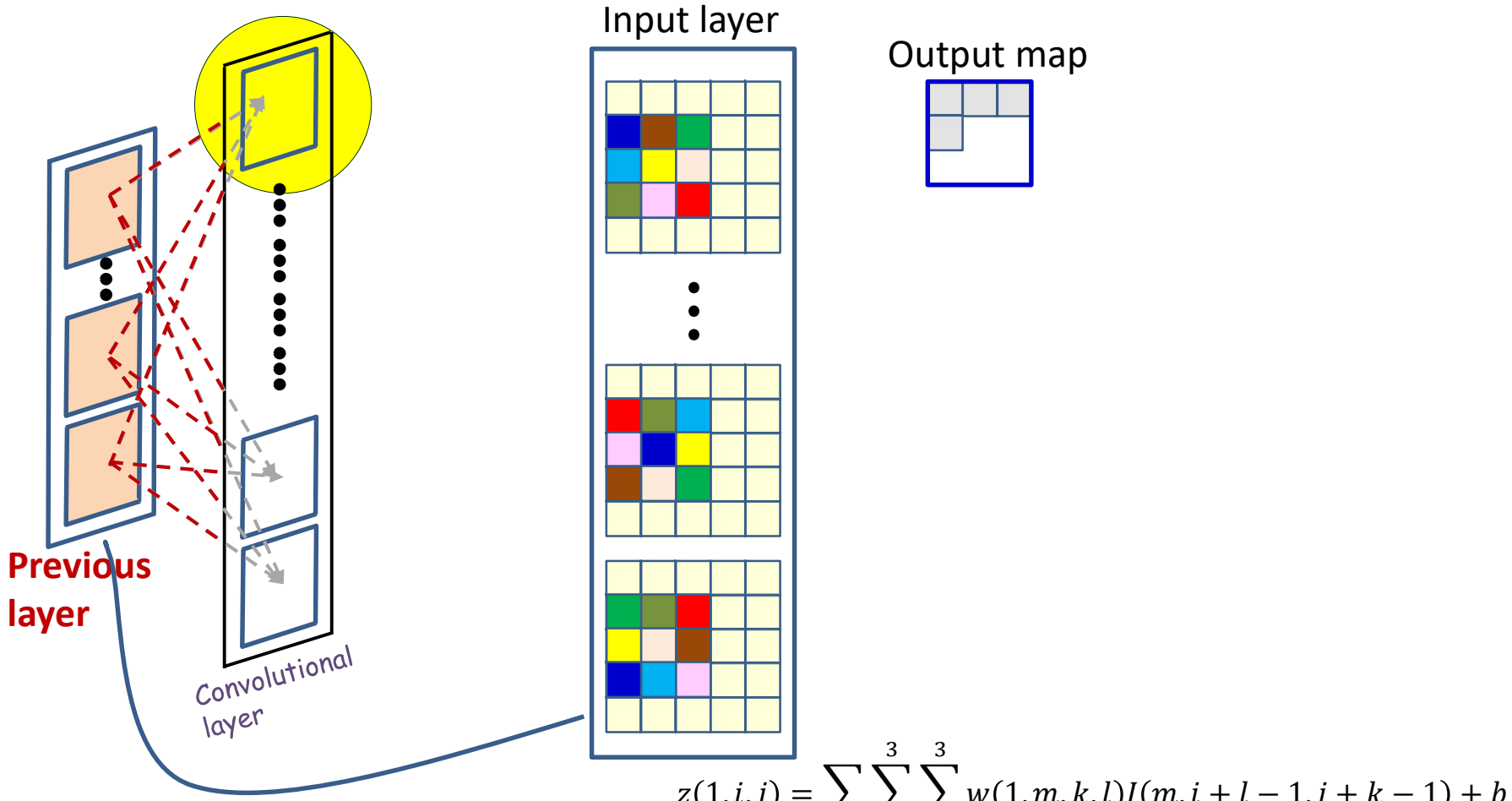


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- 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

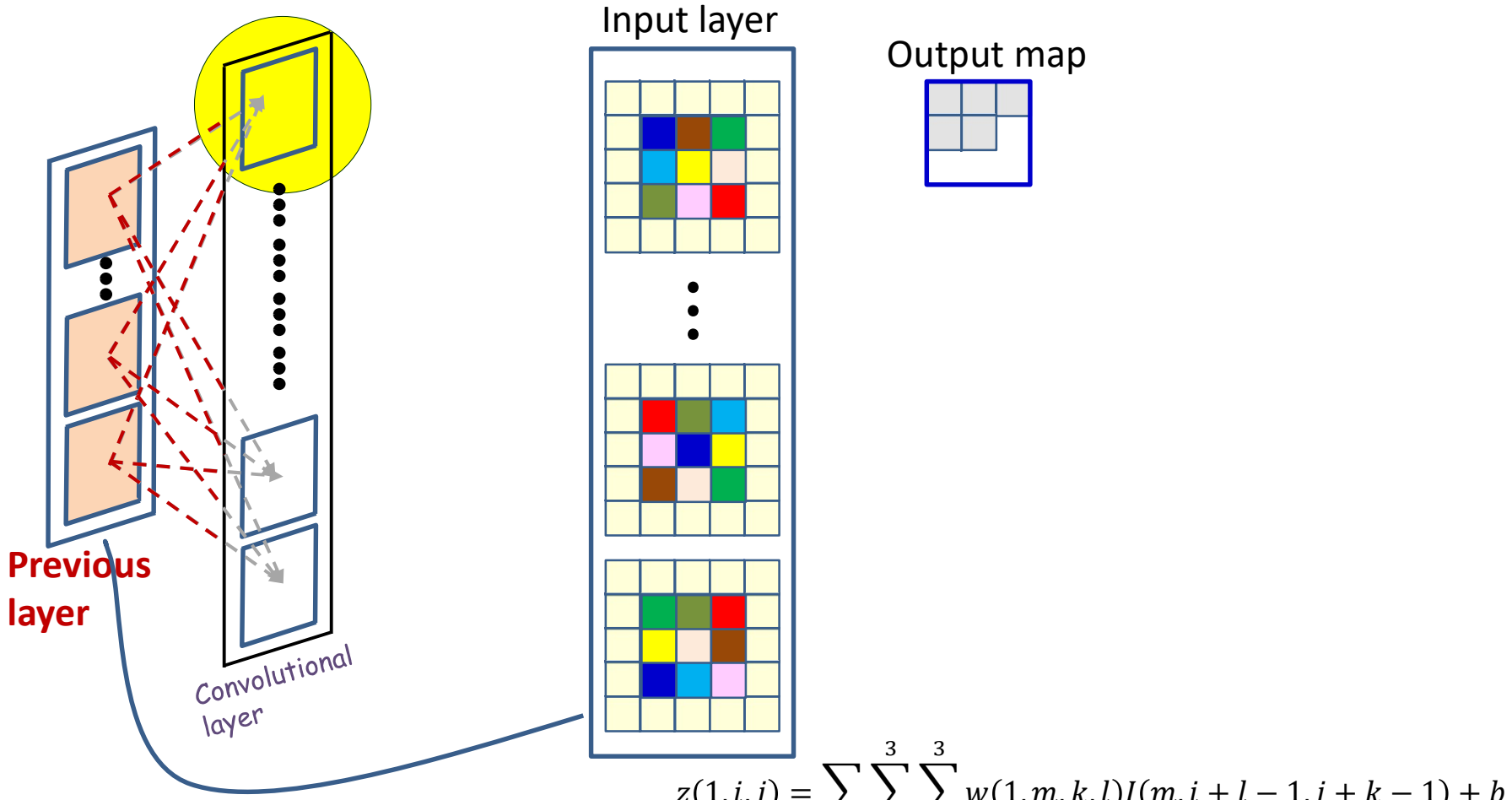


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- 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

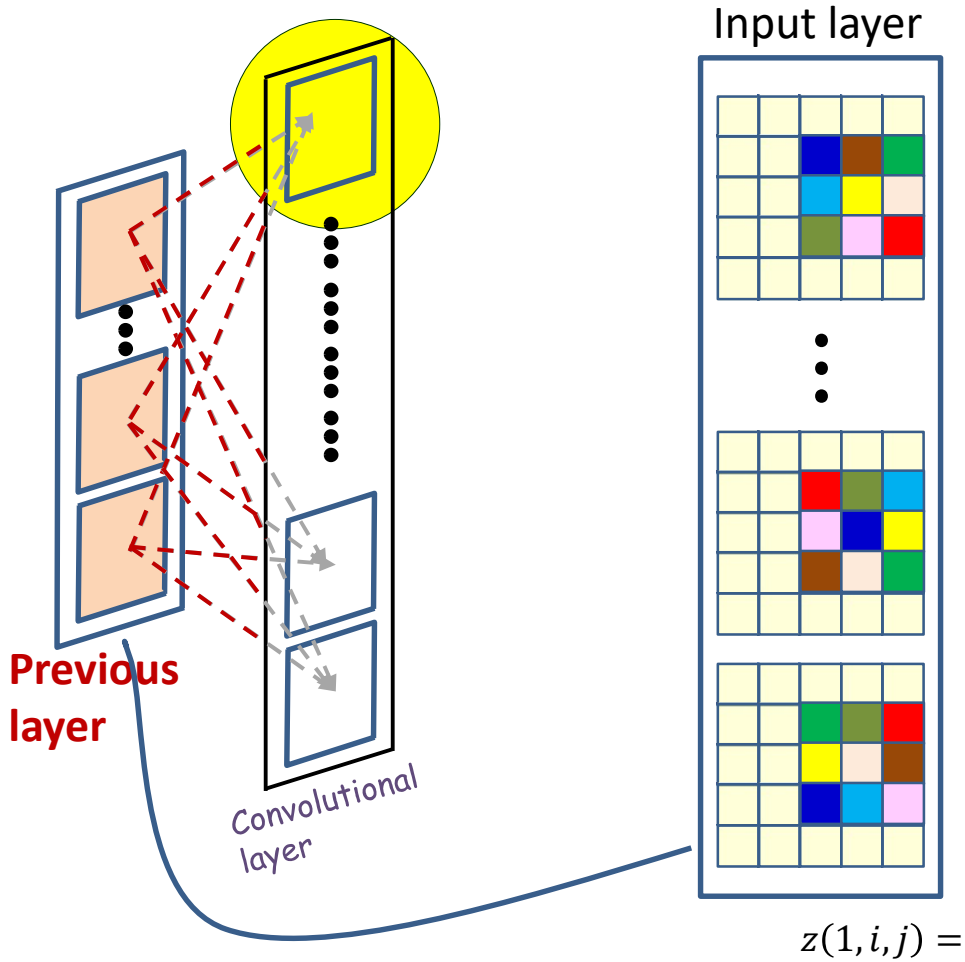


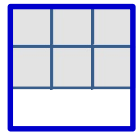
$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- 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



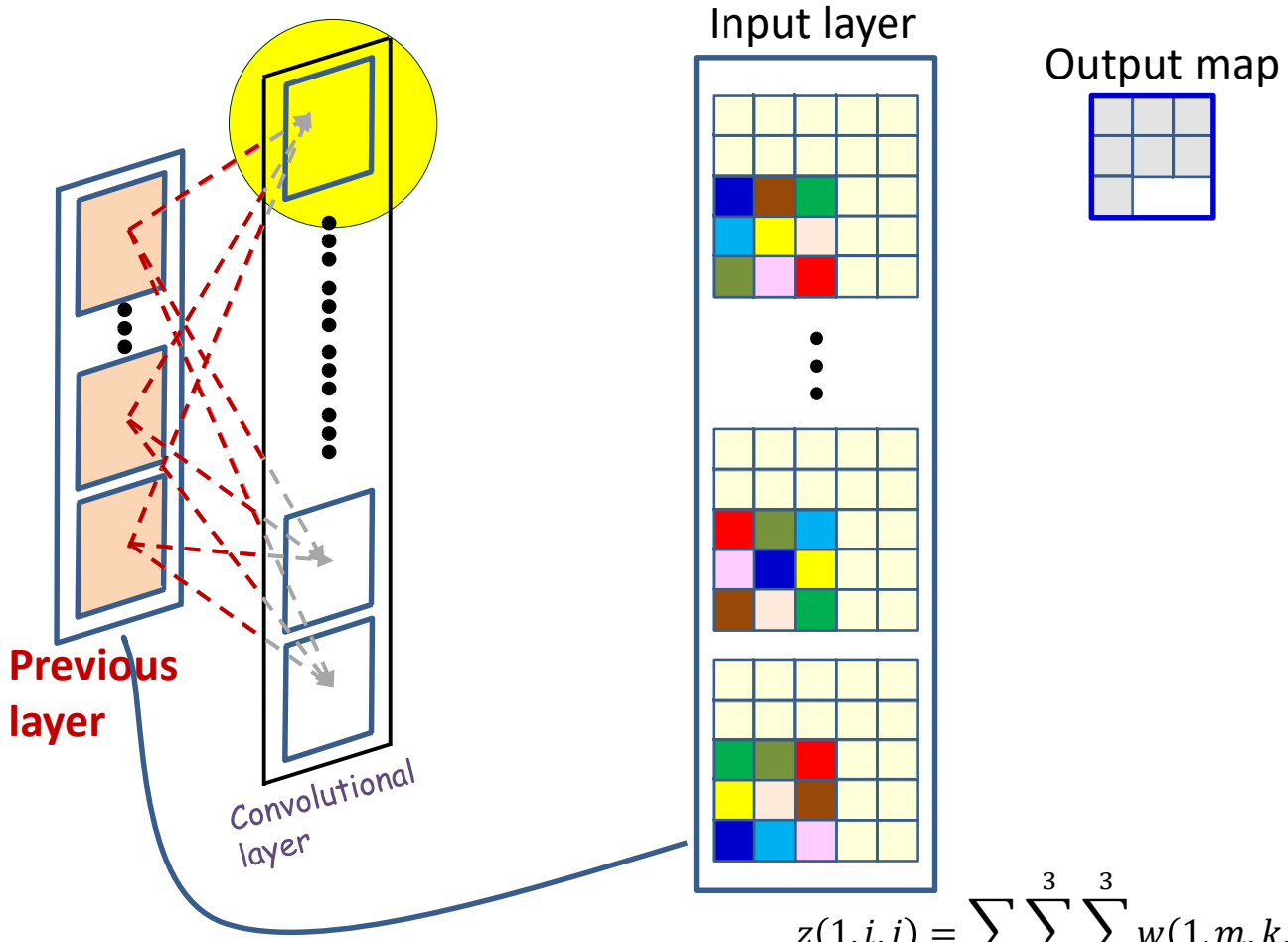


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- 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

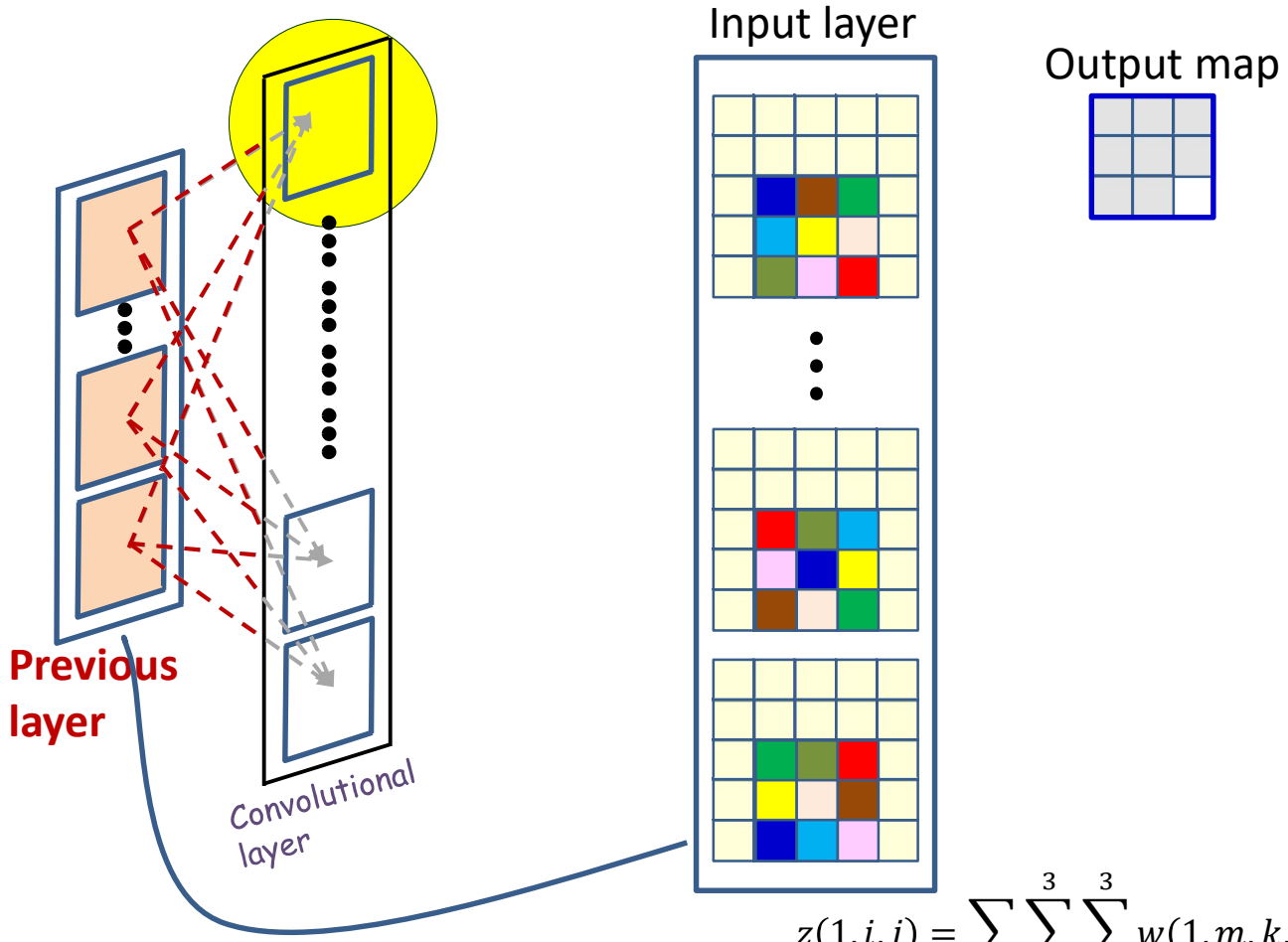


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- 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

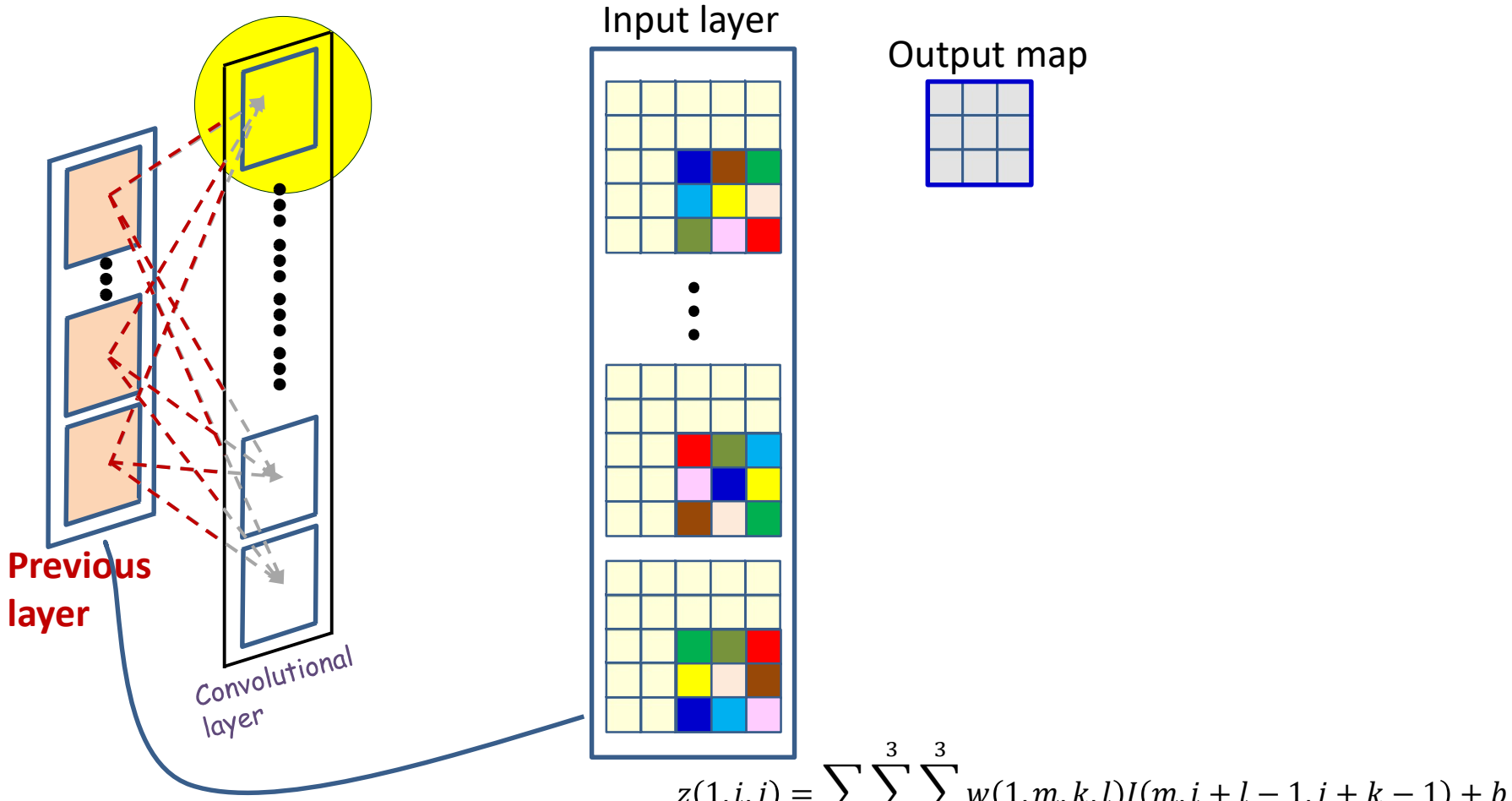


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- 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*

What really happens

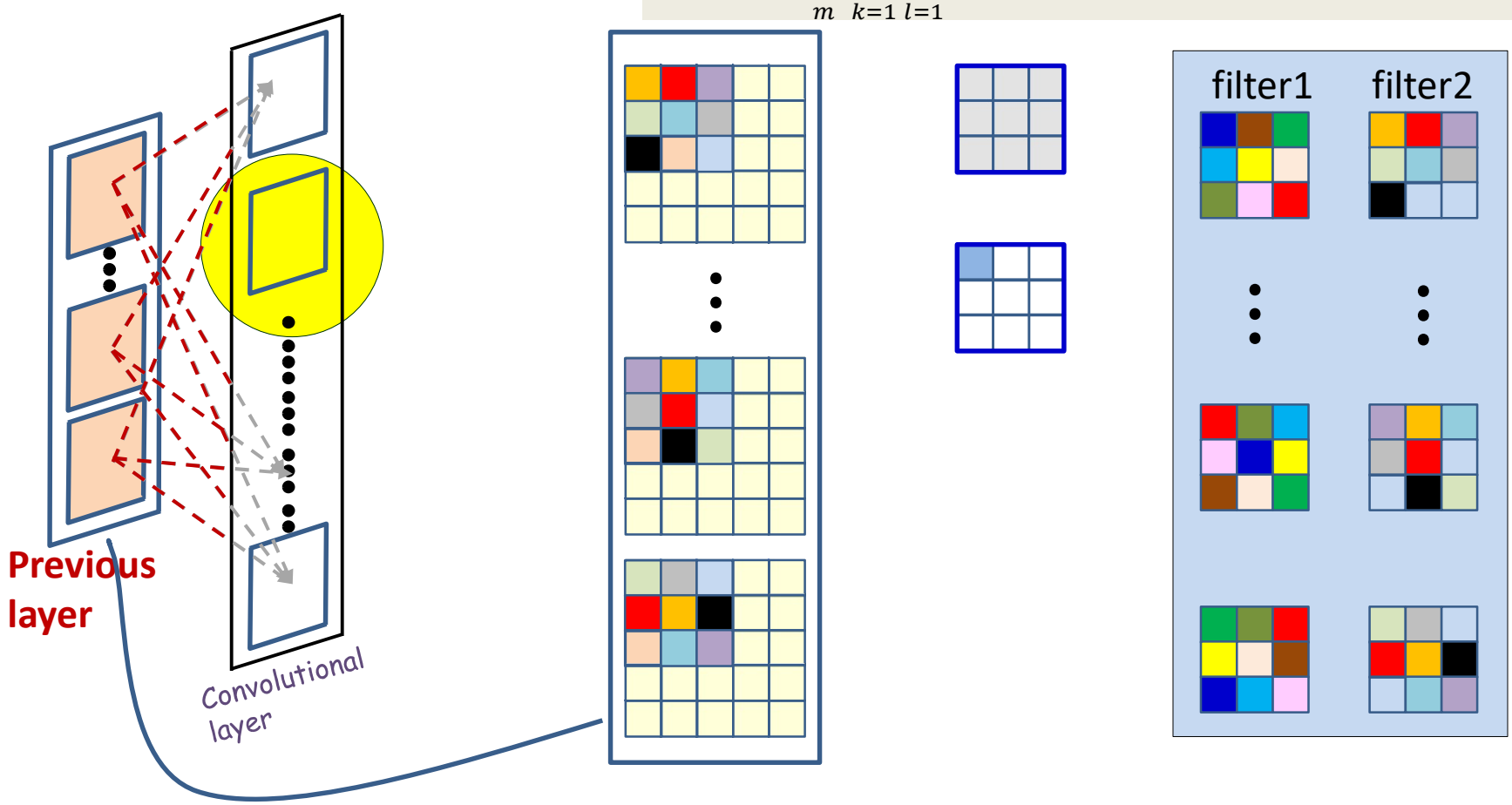


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- 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*

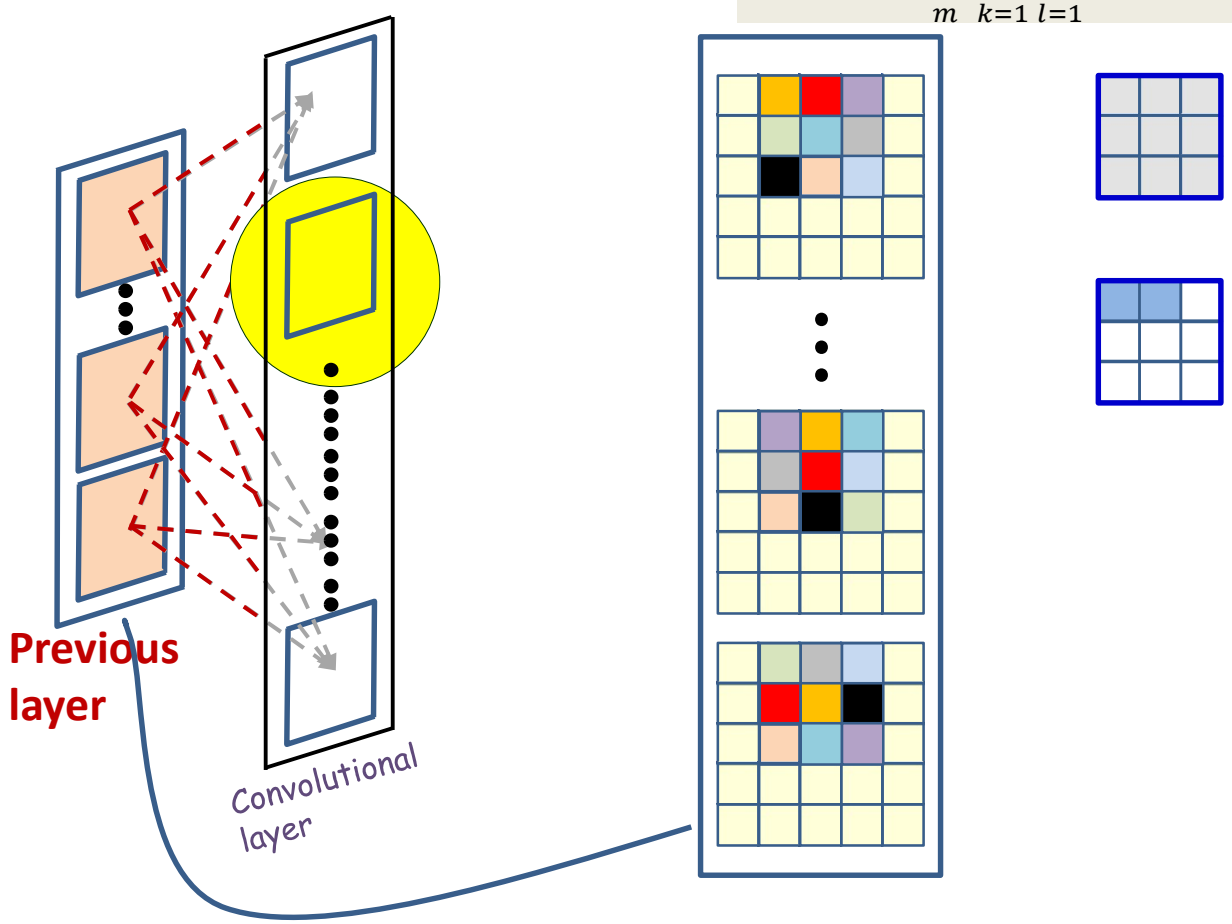
$$z(2,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(2,m,k,l) I(m, i+l-1, j+k-1) + b(2)$$



- 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*

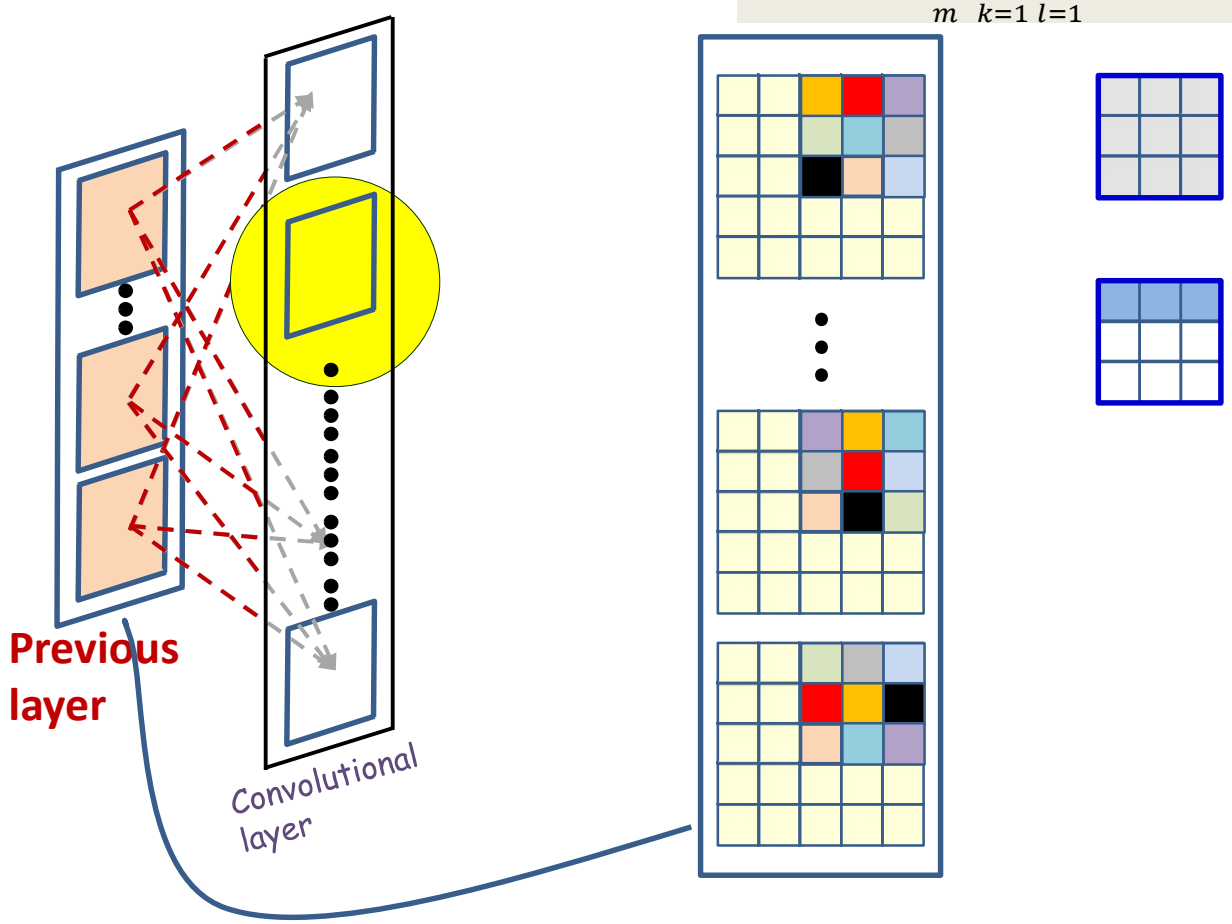
$$z(2,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(2,m,k,l) I(m, i+l-1, j+k-1) + b(2)$$



- 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*

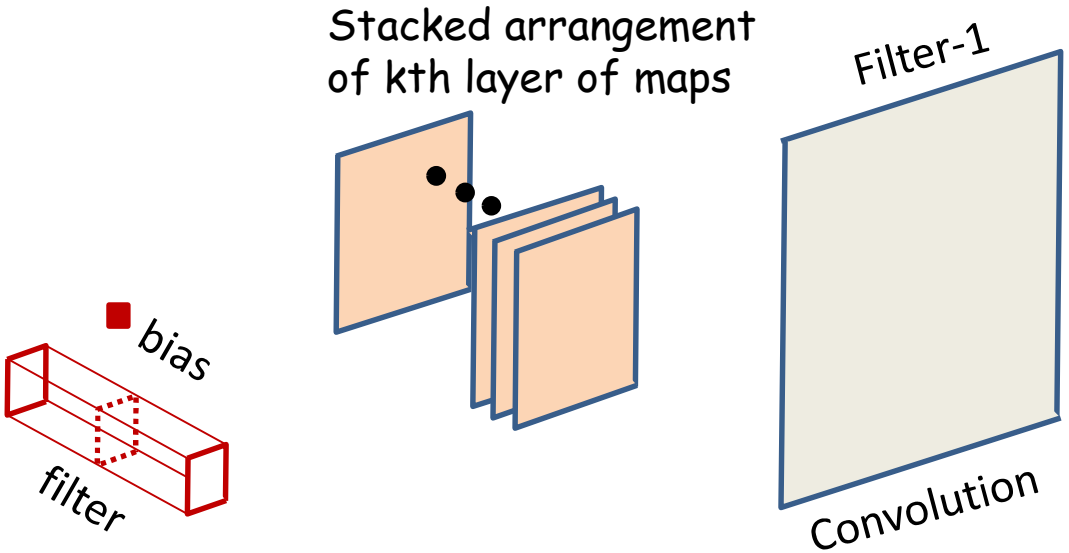
$$z(2,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(2,m,k,l) I(m, i+l-1, j+k-1) + b(2)$$



- 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*

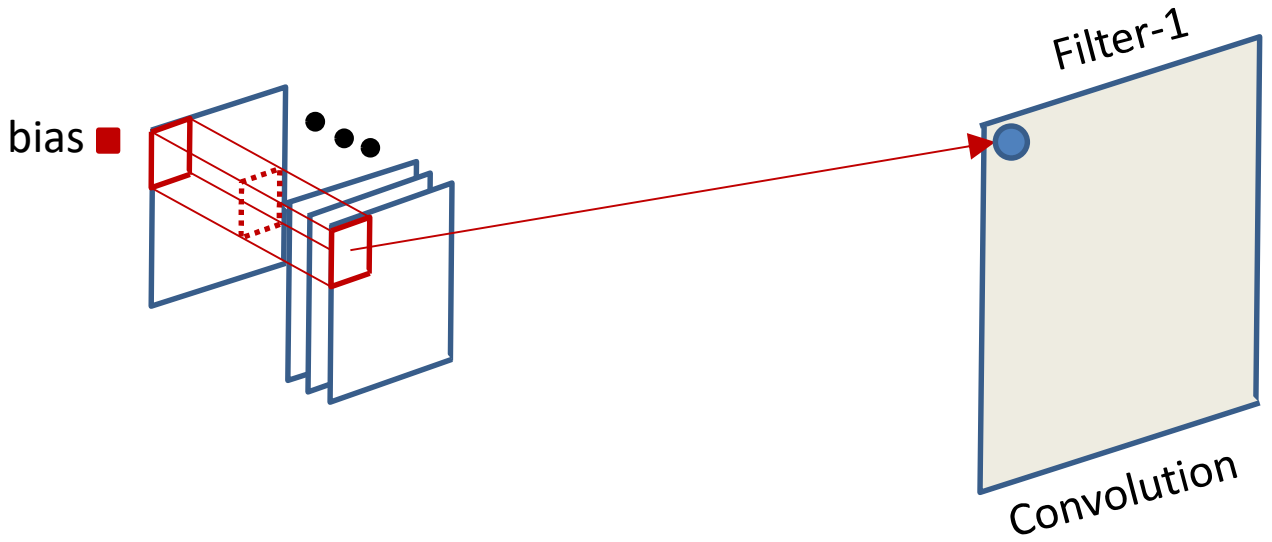
A different view



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

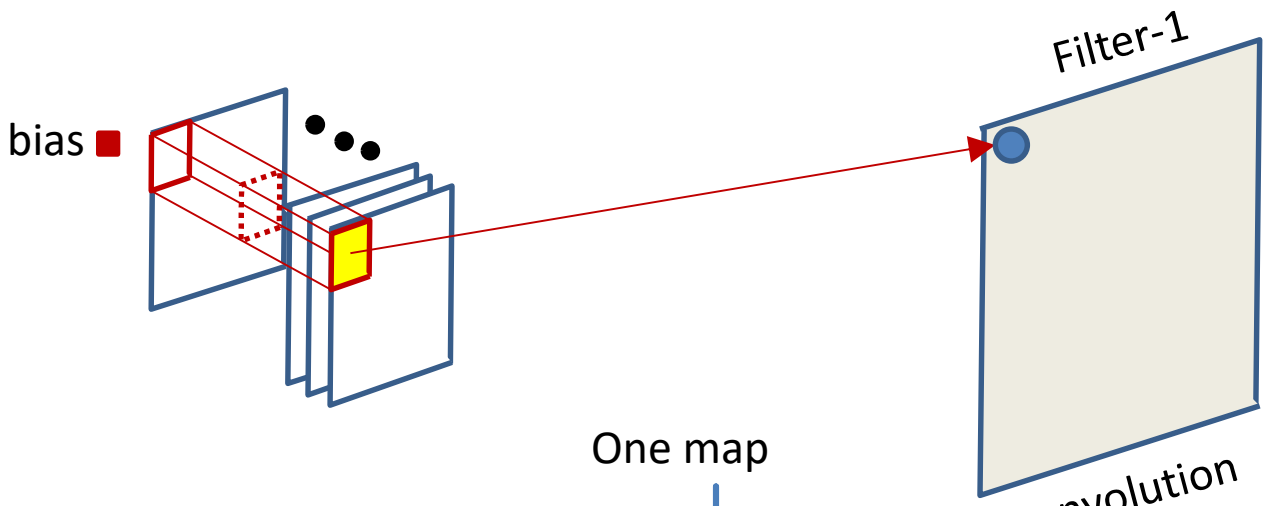
The "cube" view of input maps



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

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

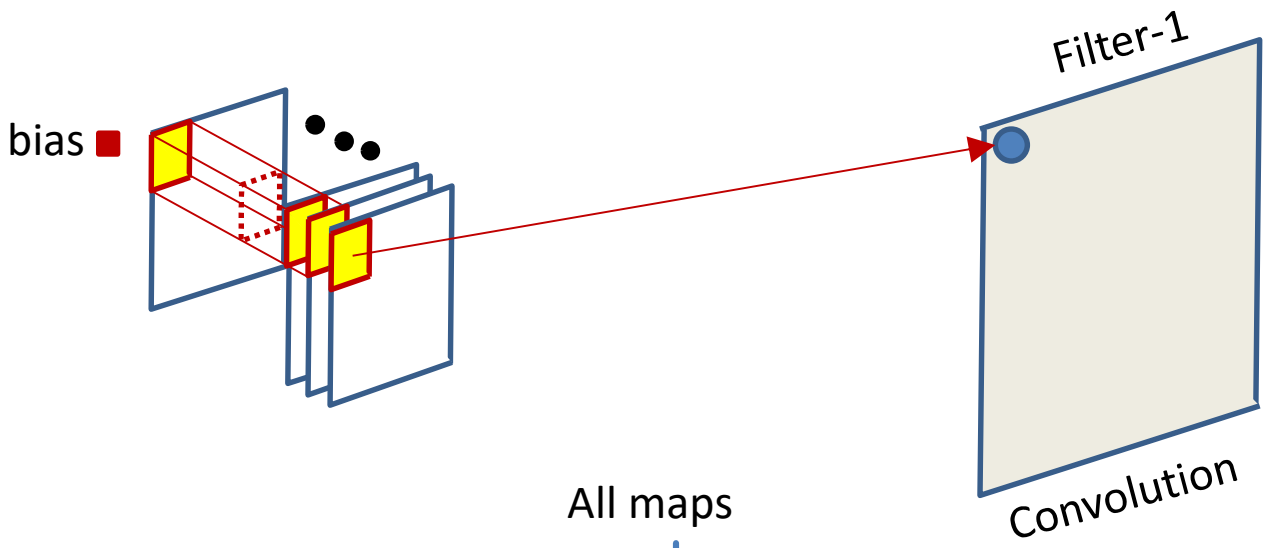
The "cube" view of input maps



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

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

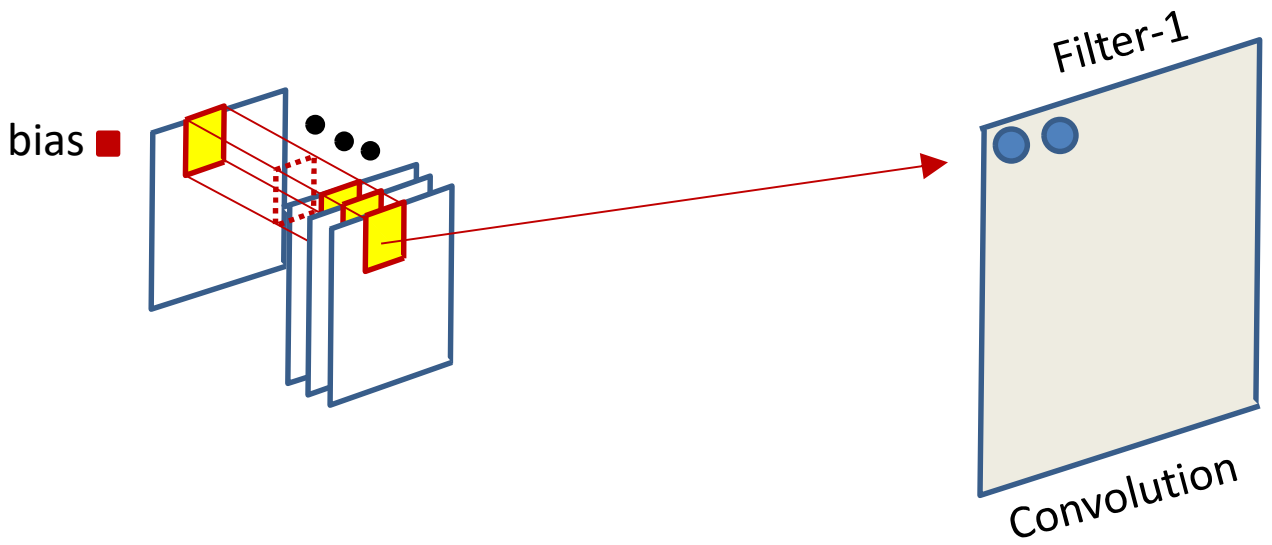
The "cube" view of input maps



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

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

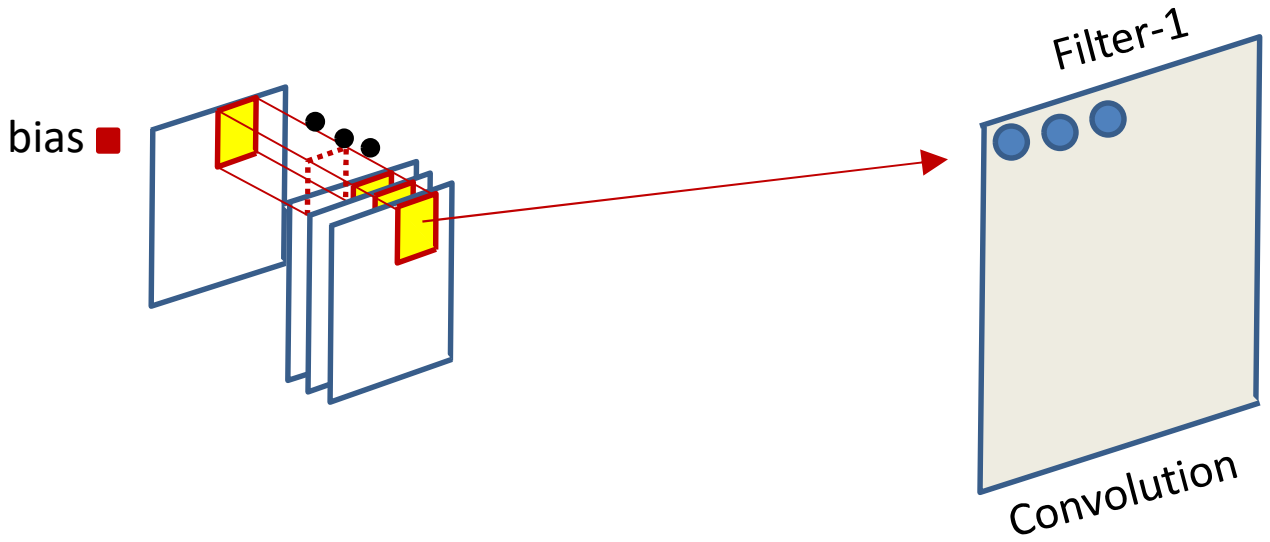
The "cube" view of input maps



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

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

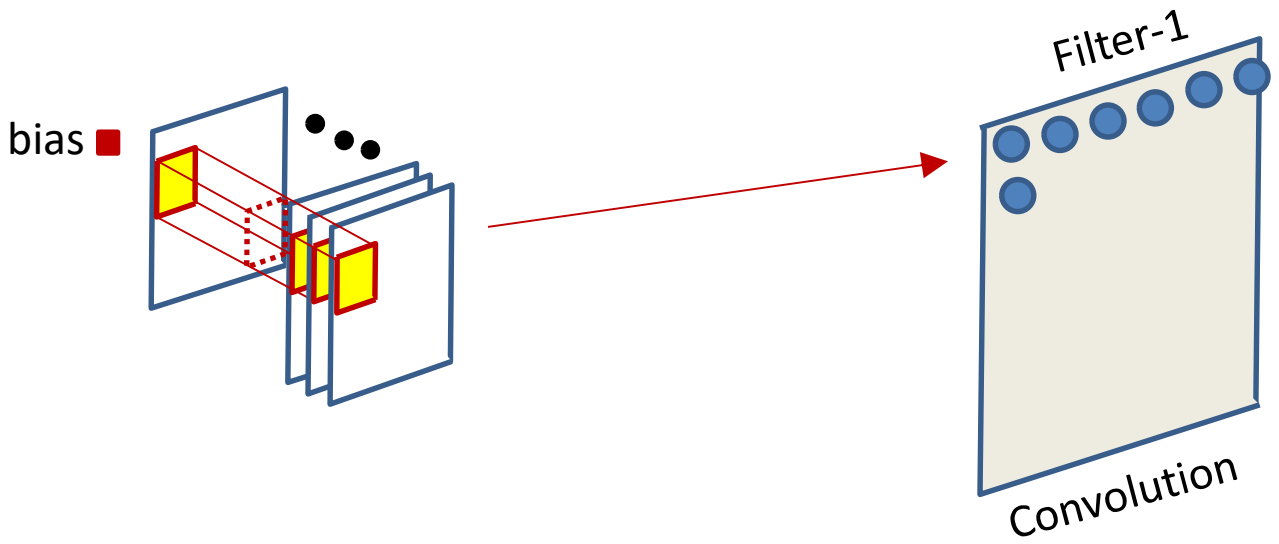
The "cube" view of input maps



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

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

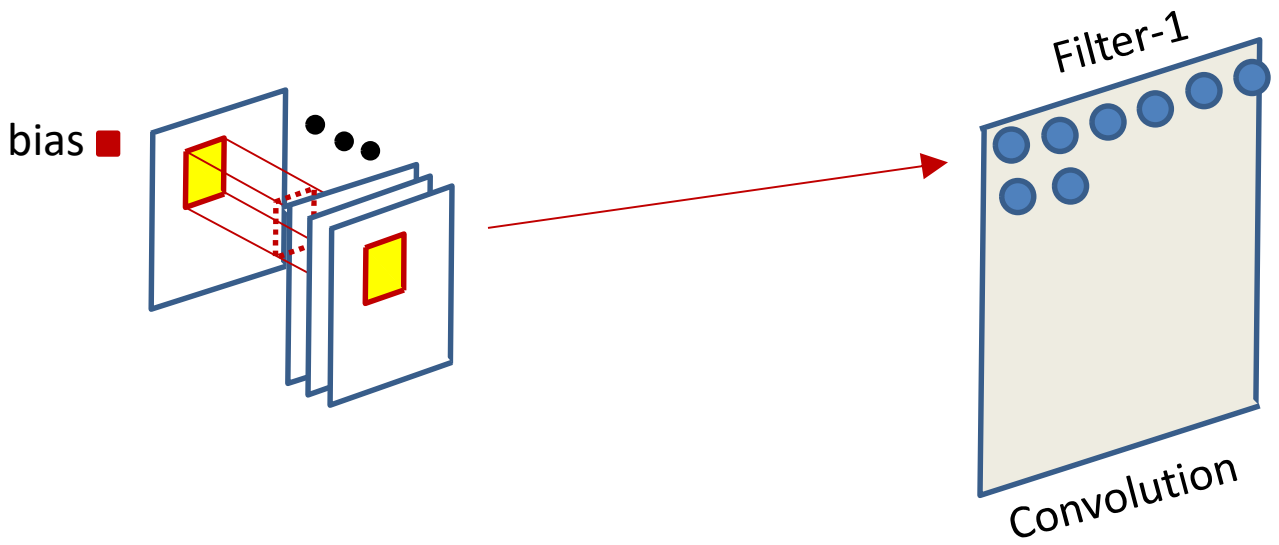
The "cube" view of input maps



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

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

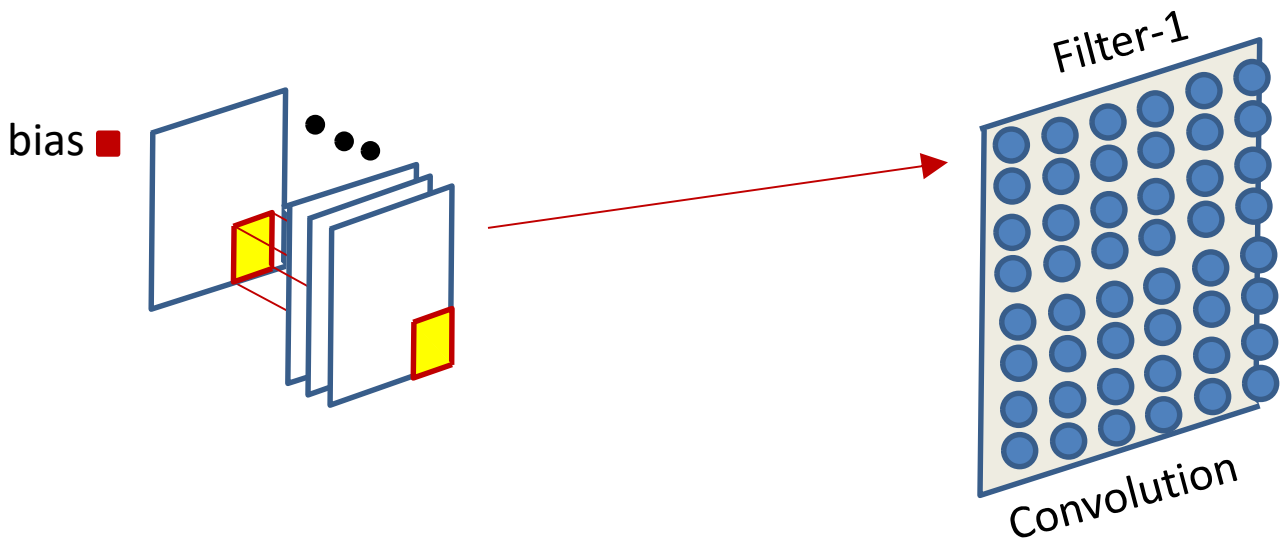
The "cube" view of input maps



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

- 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*

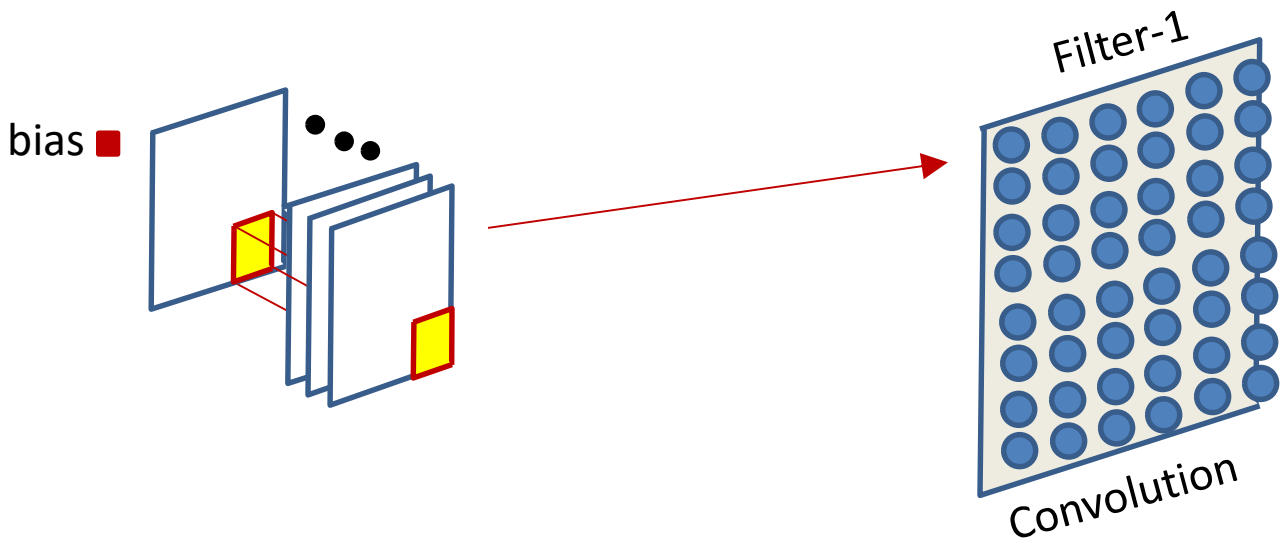
Convolutional neural net: Vector notation

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

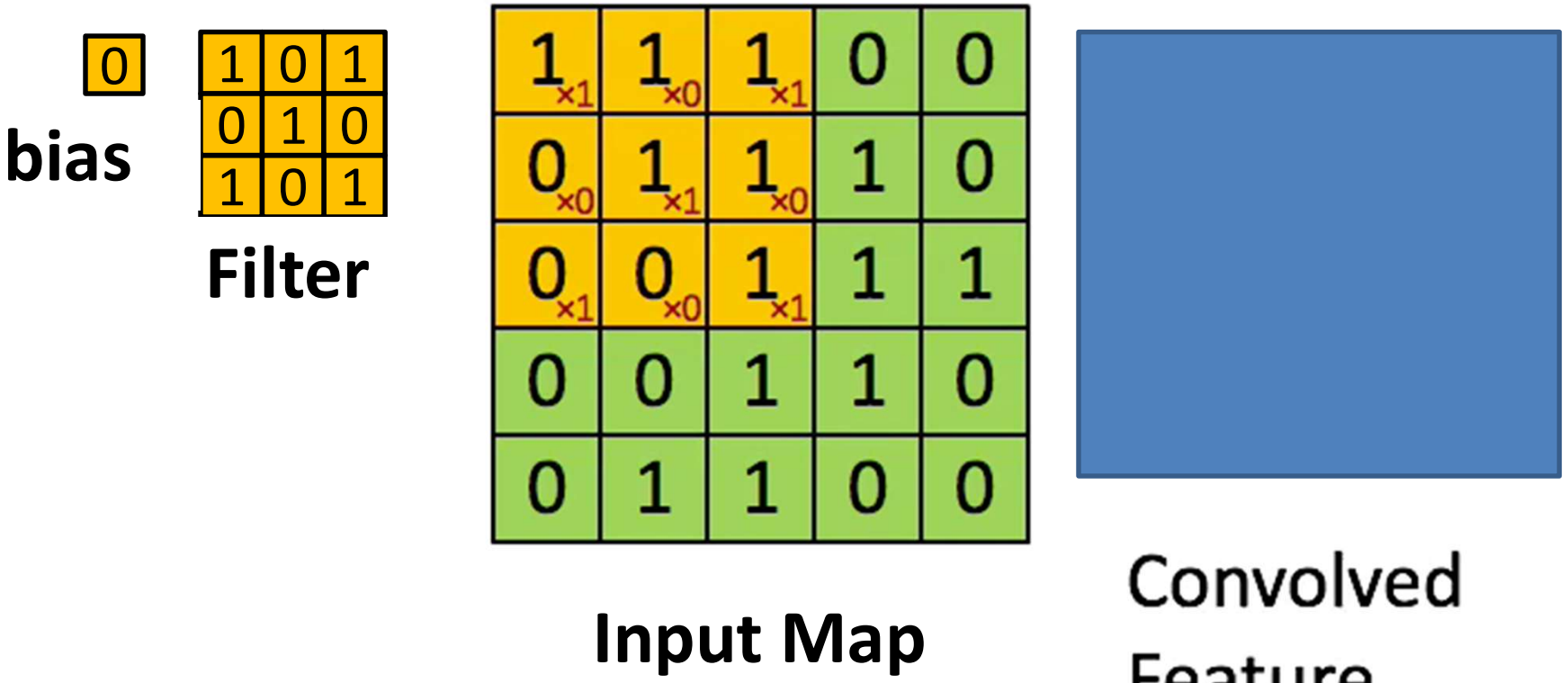
```
Y(0) = Image
for l = 1:L  # layers operate on vector at (x,y)
  for x = 1:W_{l-1}-K_l+1
    for y = 1:H_{l-1}-K_l+1
      for j = 1:D_l
        segment = Y(l-1,:,x:x+K_l-1,y:y+K_l-1) #3D tensor
        z(l,j,x,y) = W(l,j).segment #tensor inner prod.
        Y(l,j,x,y) = activation(z(l,j,x,y))

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

Engineering consideration: The size of the result of the convolution



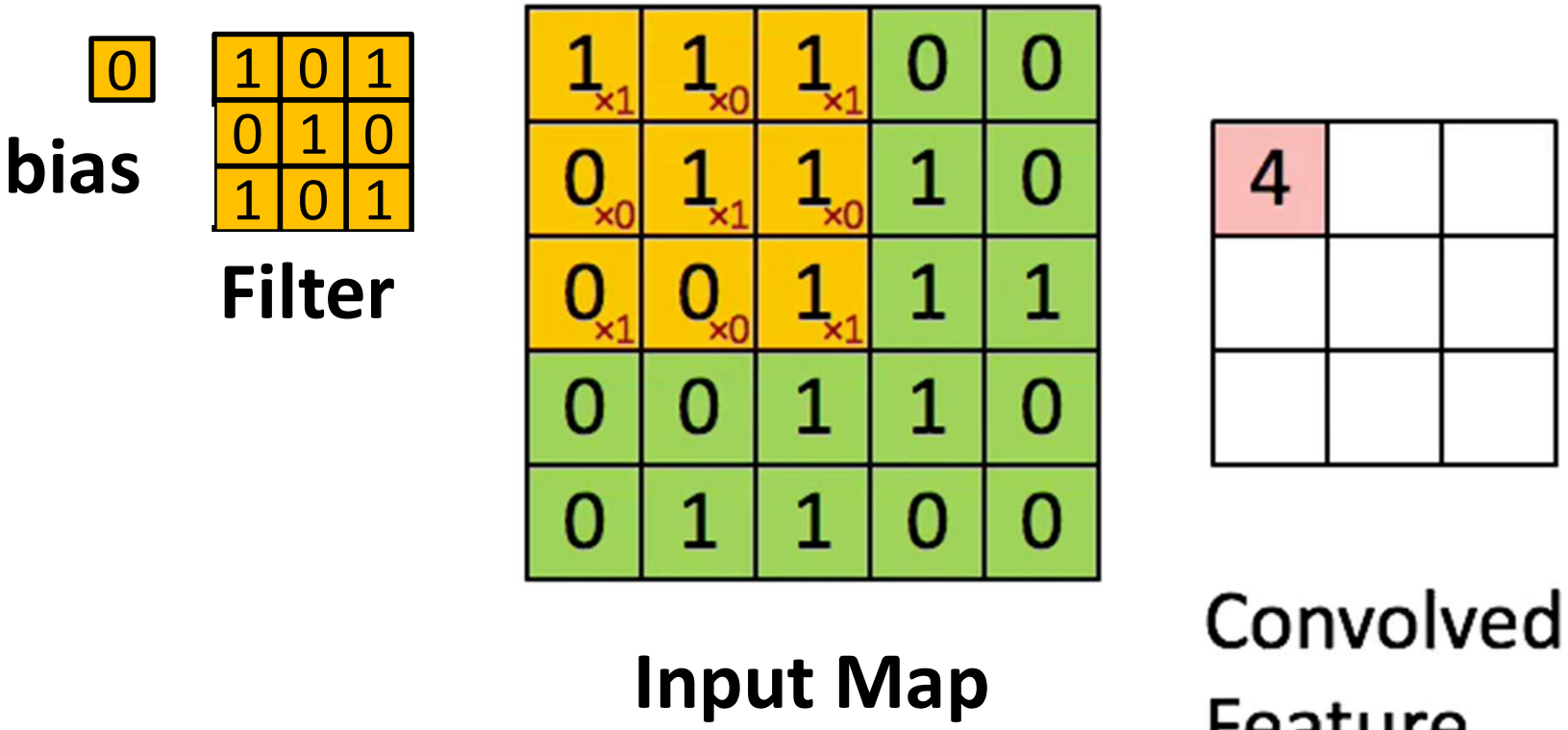
- The size of the output of the convolution operation depends on implementation factors
  - The size of the input, the size of the filter, and the stride
- And may not be identical to the size of the input
  - Let's take a brief look at this for completeness sake

The size of the convolution



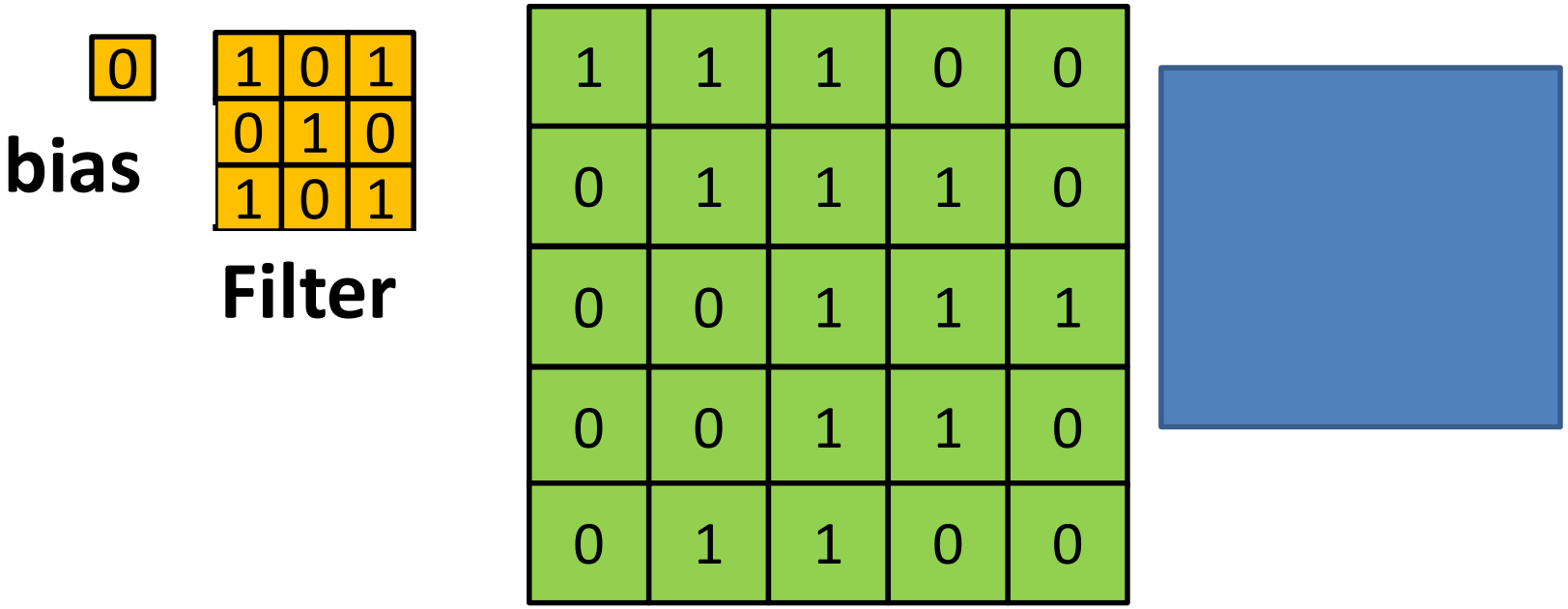
- Image size: 5x5
- Filter: 3x3
- “Stride”: 1
- Output size = ?

The size of the convolution



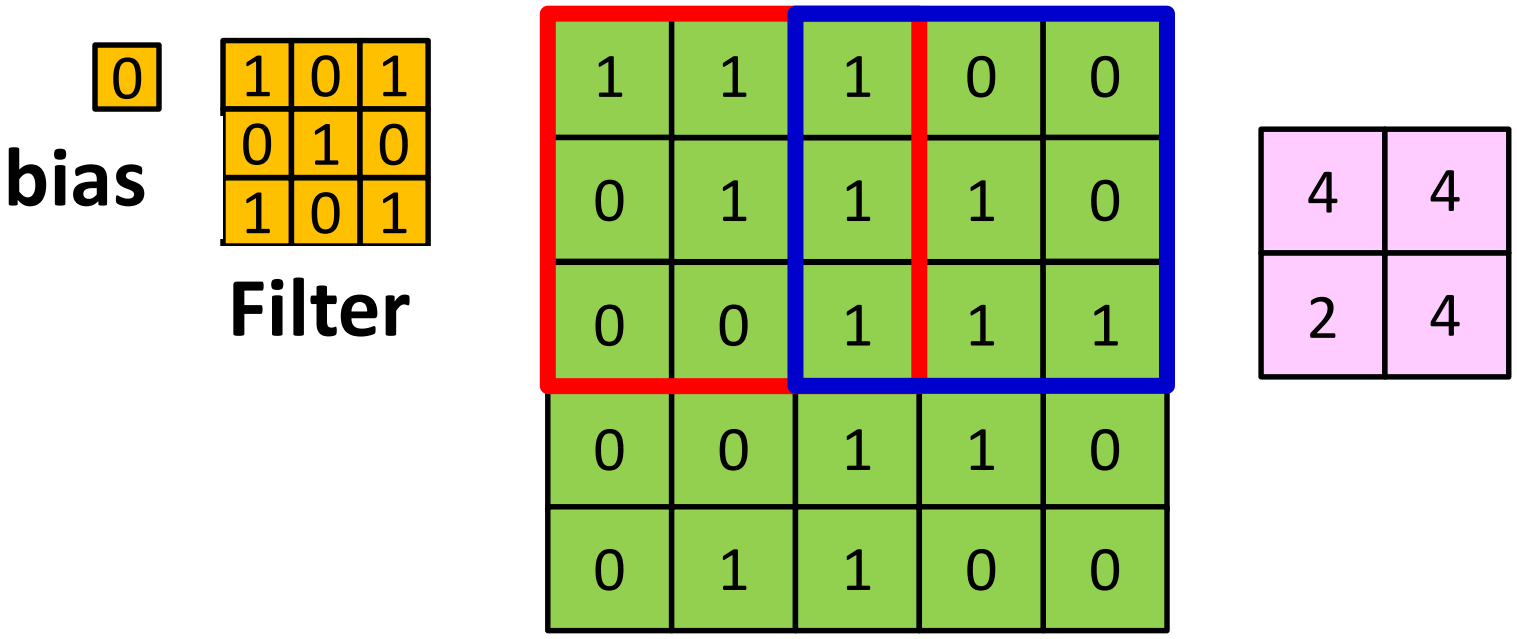
- Image size: 5x5
- Filter: 3x3
- Stride: 1
- Output size = ?

The size of the convolution



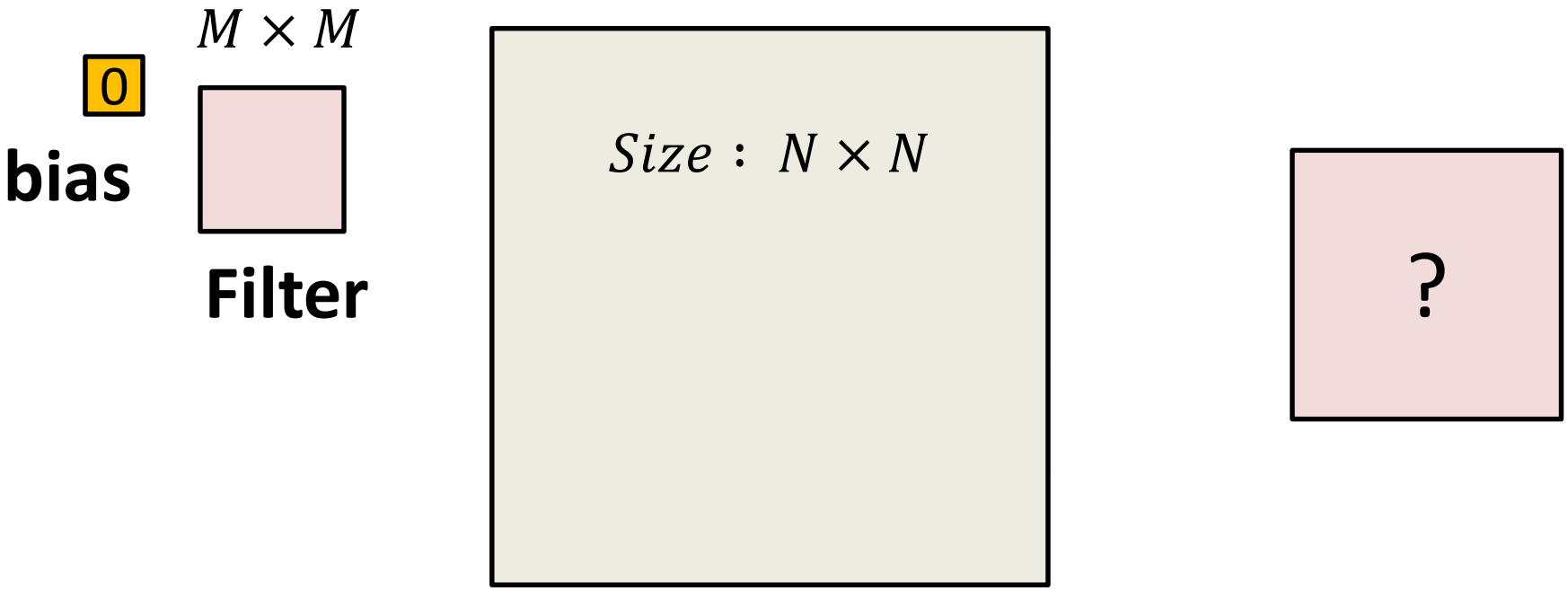
- Image size: 5x5
- Filter: 3x3
- Stride: 2
- Output size = ?

The size of the convolution



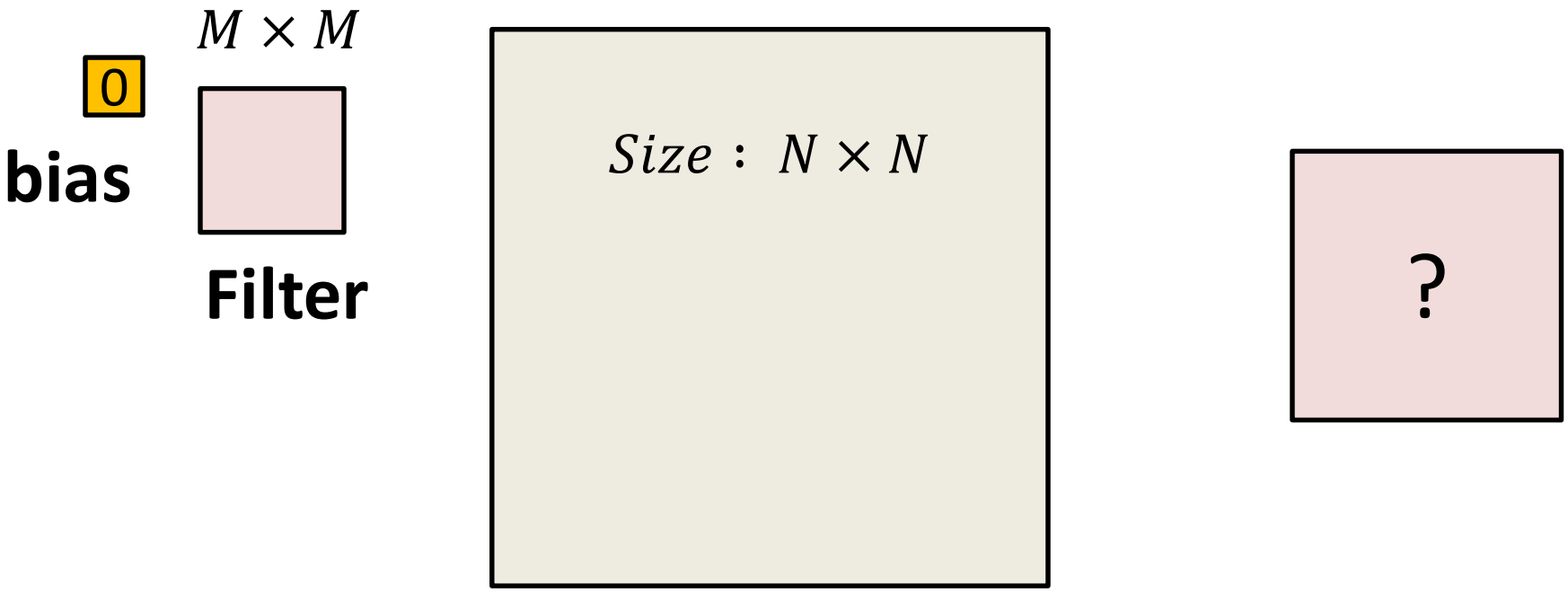
- Image size: 5x5
- Filter: 3x3
- Stride: 2
- Output size = ?

The size of the convolution



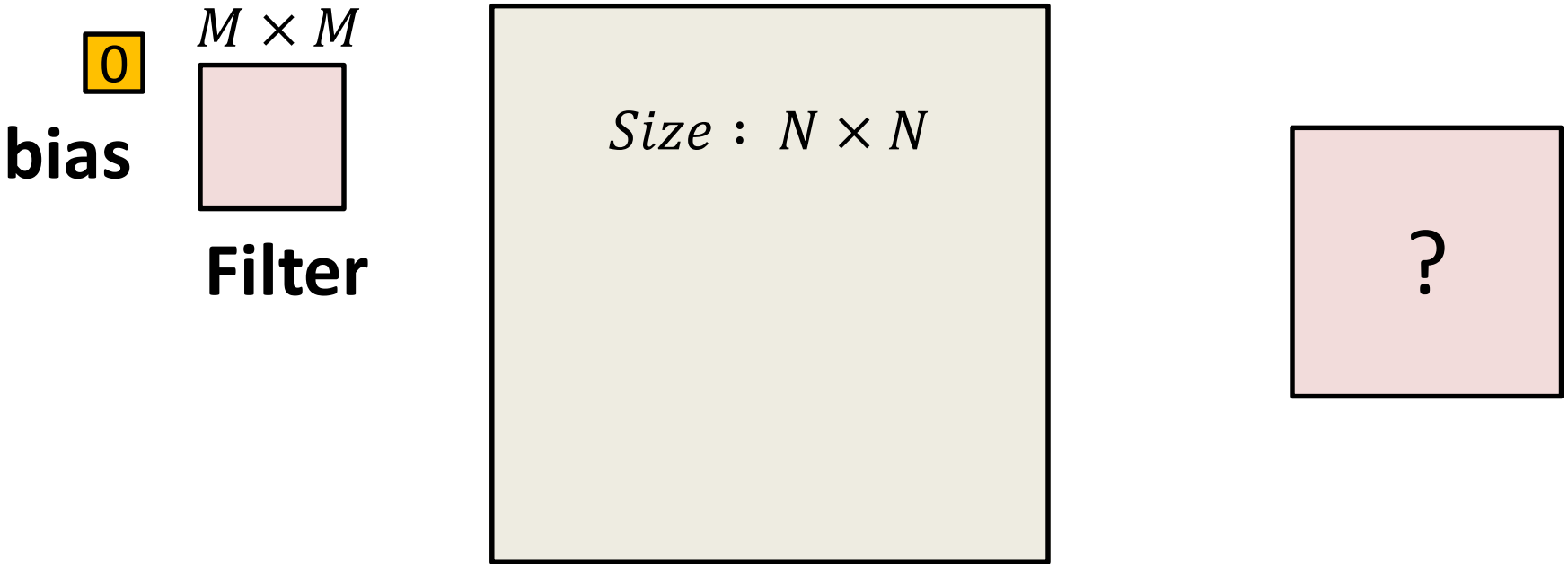
- Image size: $N \times N$
- Filter: $M \times M$
- Stride: 1
- Output size = (N-M)+1 on each side

The size of the convolution



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

The size of the convolution



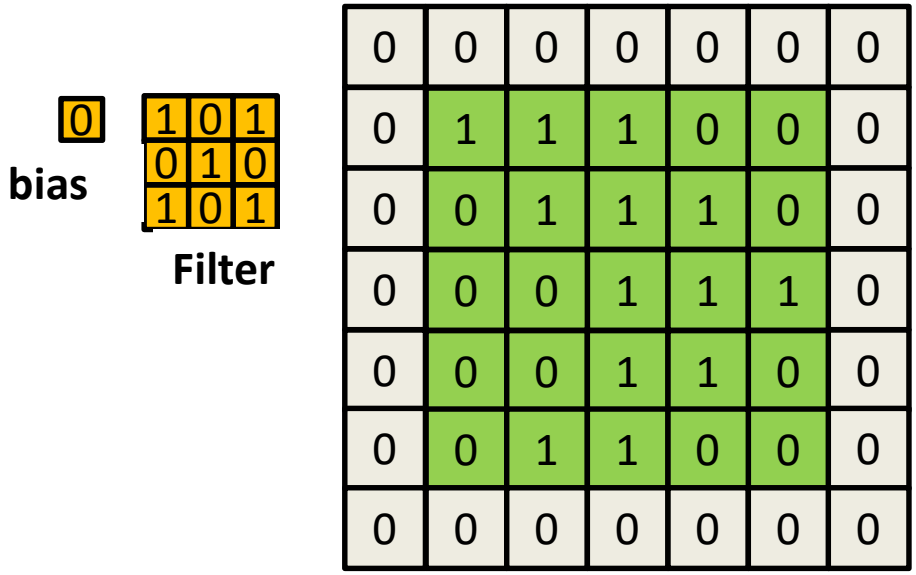
- Image size: $N \times N$
- Filter: $M \times M$
- Stride: S
- Output size (each side) = $\lfloor (N - M)/S \rfloor + 1$
  - Assuming you're not allowed to go beyond the edge of the input

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

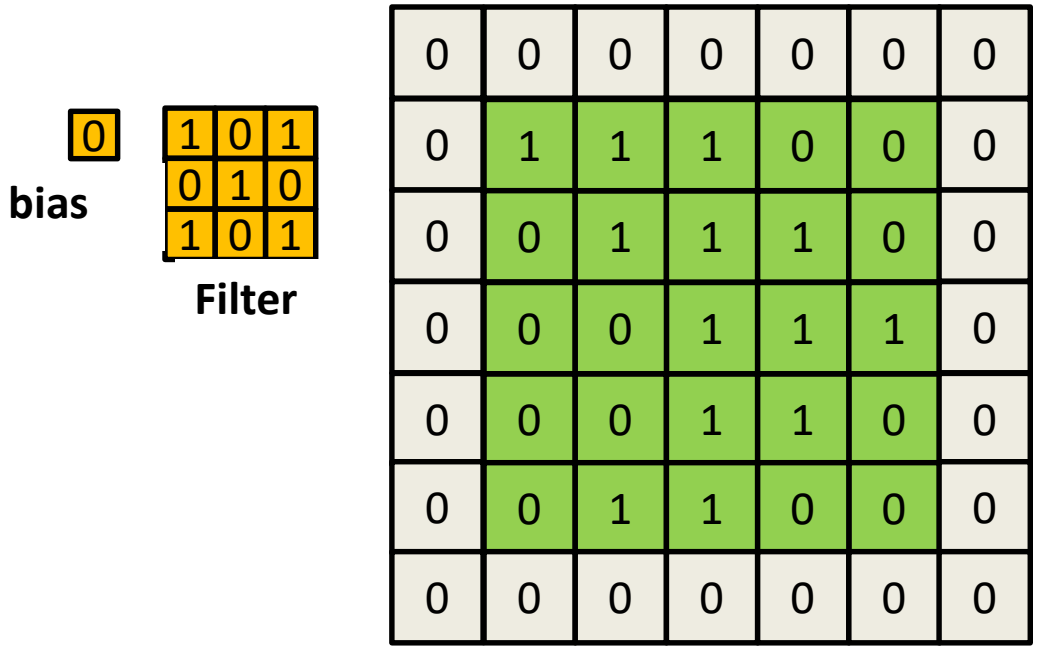
- Results in a reduction in the output size
  - Even if $S = 1$
  - Sometimes not considered acceptable
    - If there's no active downsampling, through max pooling and/or $S > 1$, then the output map should ideally be the same size as the input

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**

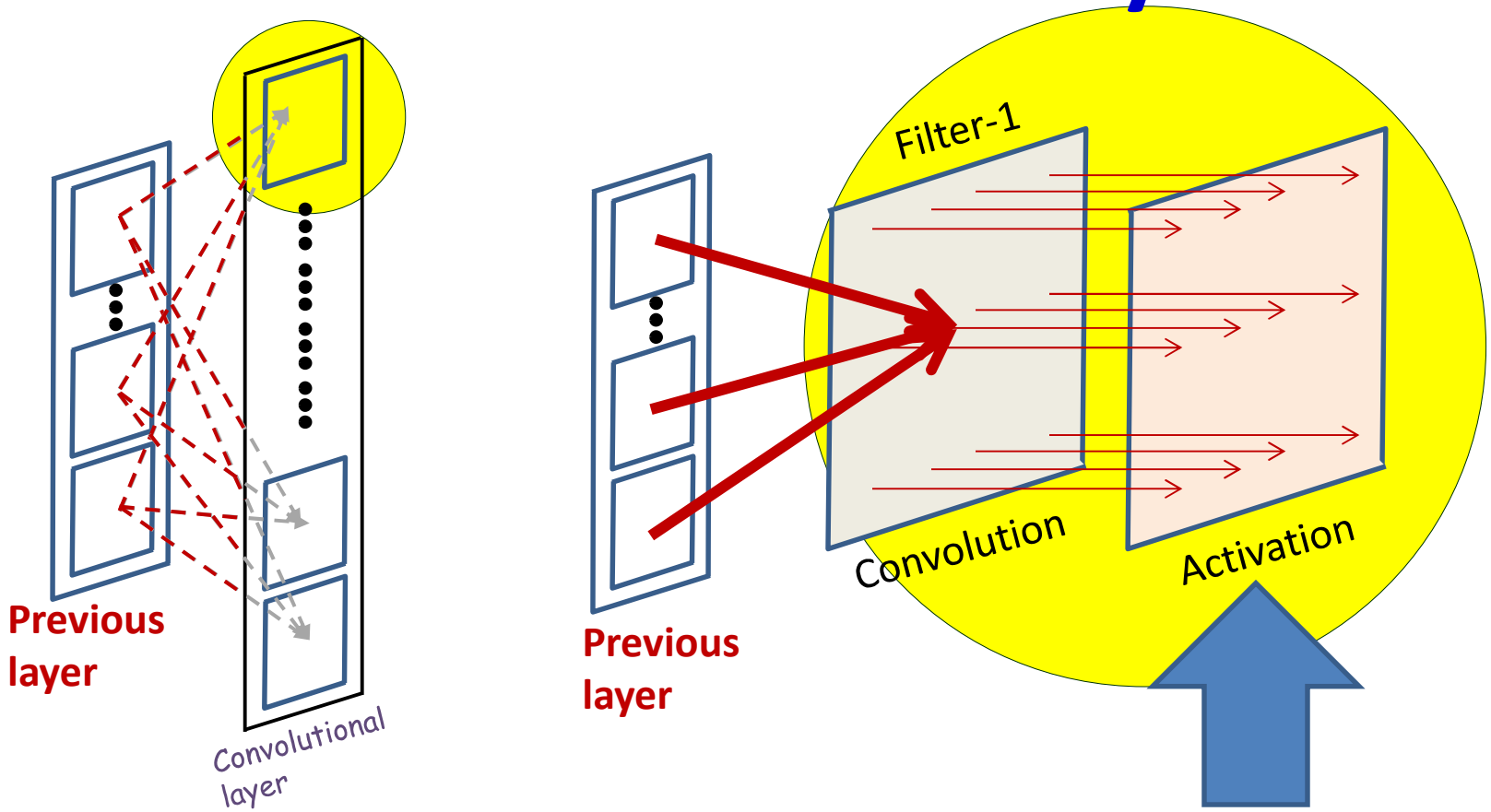
Zero padding

- 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

A convolutional layer



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

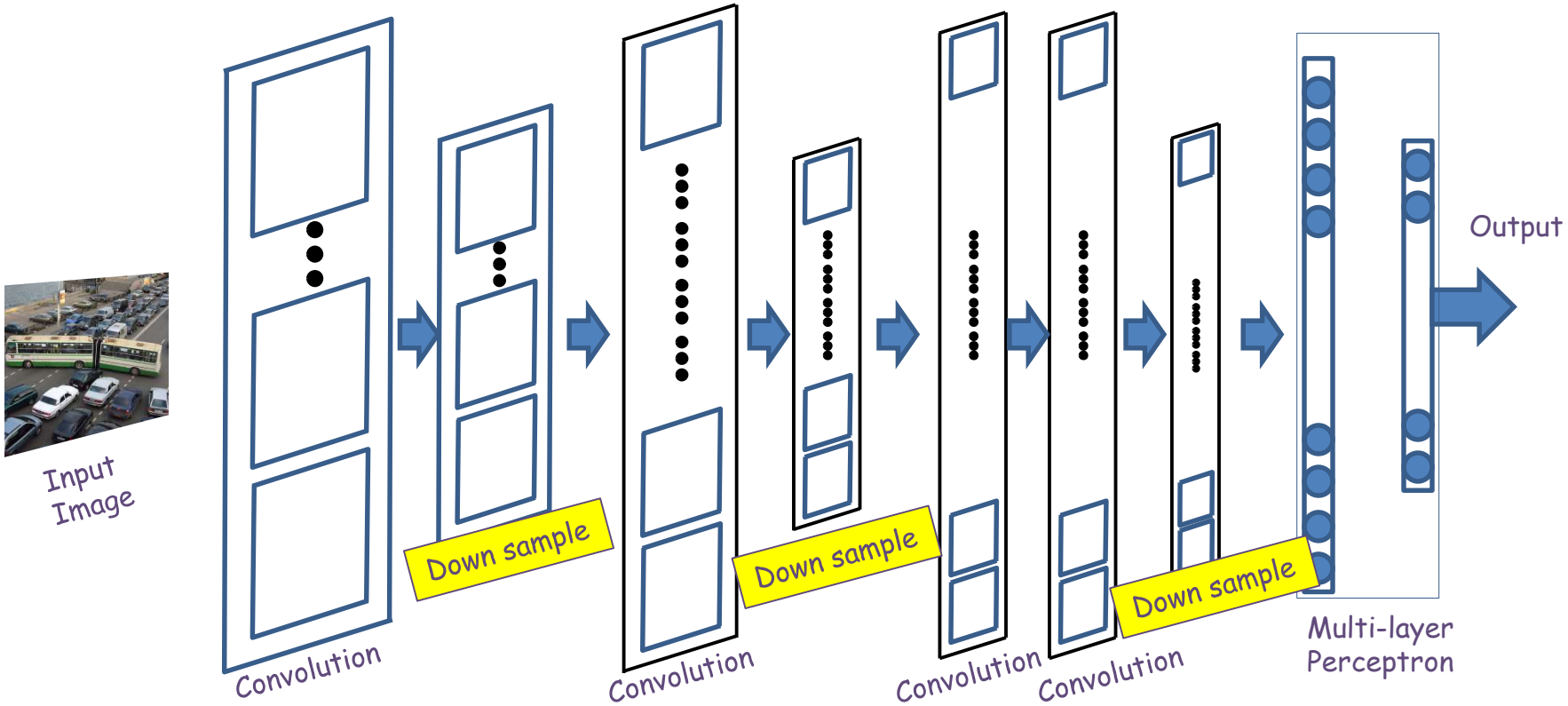
Convolutional neural net: Vector notation

The weight W(l,j) is now a 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

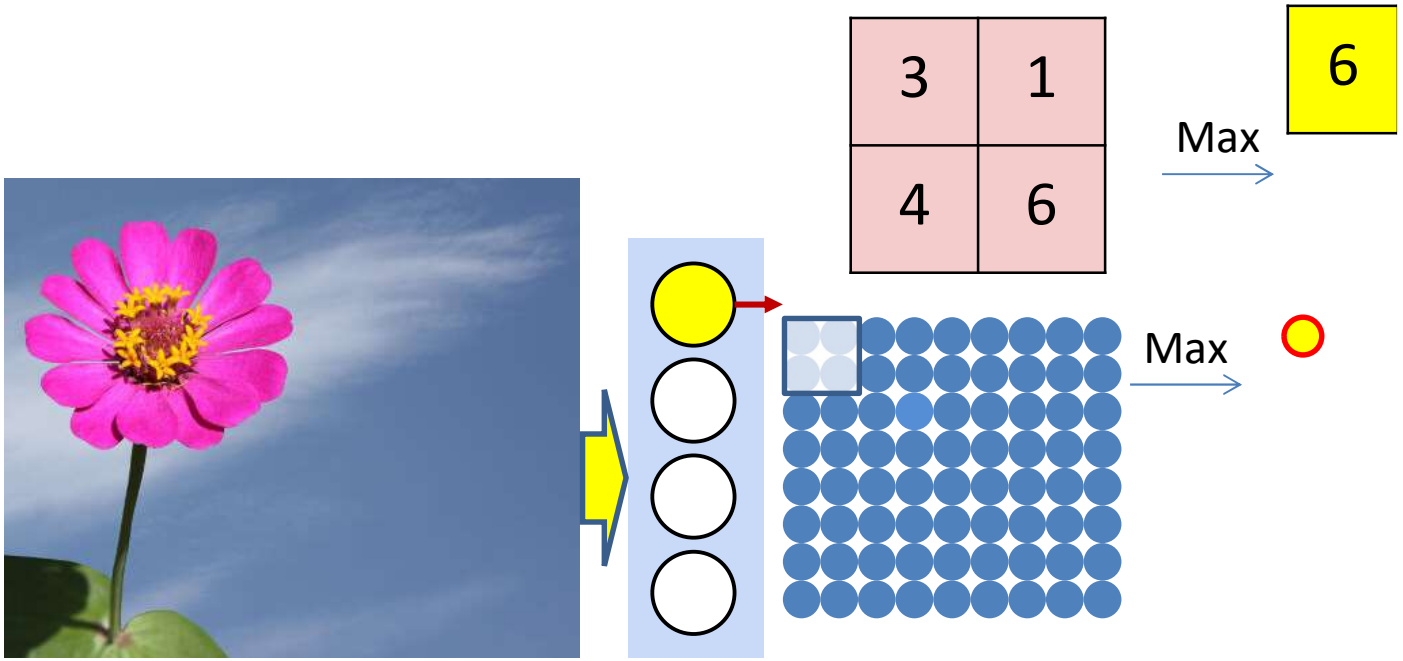
```
Y(0) = Image
for l = 1:L  # layers operate on vector at (x,y)
  for x = 1:W_{l-1}-K_l+1
    for y = 1:H_{l-1}-K_l+1
      for j = 1:D_l
        segment = Y(l-1,:,x:x+K_l-1,y:y+K_l-1) #3D tensor
        z(l,j,x,y) = W(l,j).segment #tensor inner prod.
        Y(l,j,x,y) = activation(z(l,j,x,y))

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

The other component
Downsampling/Pooling



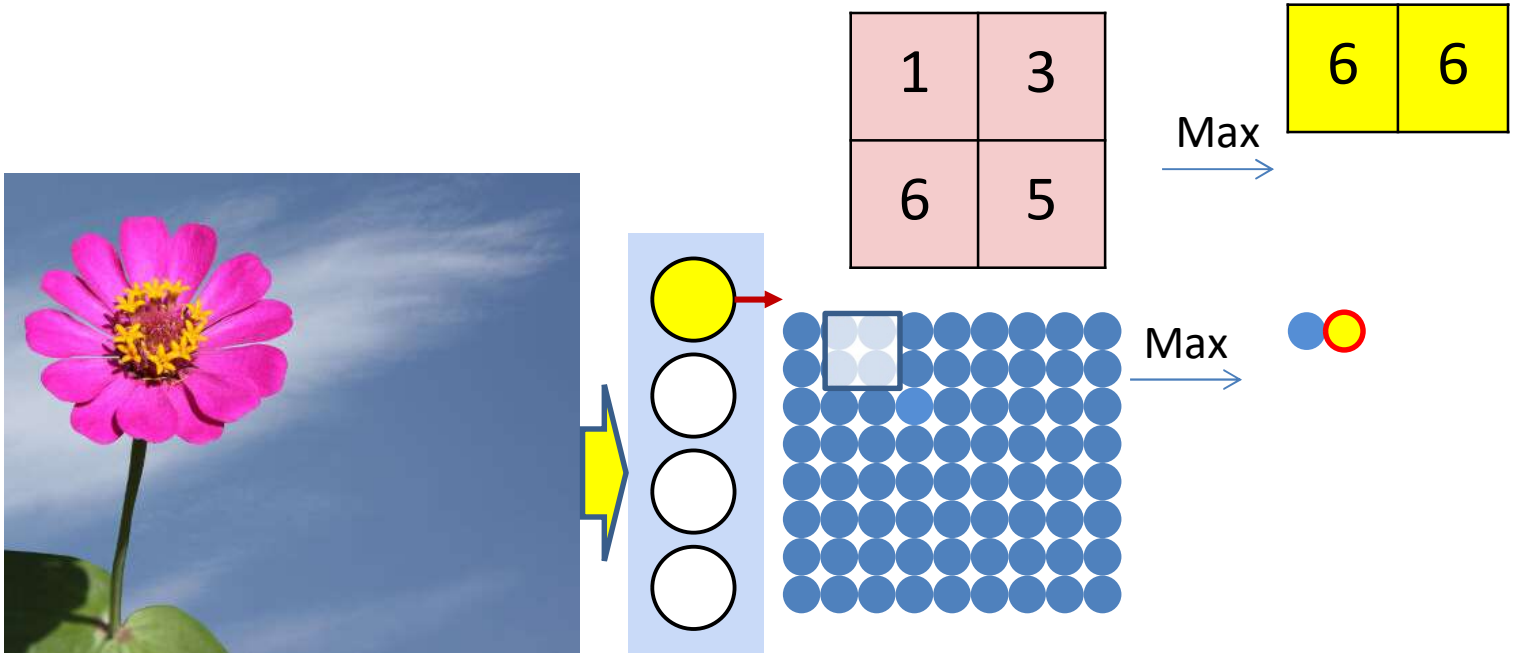
- Convolution (and activation) layers are followed intermittently by "downsampling" (or "pooling") layers
  - Typically (but not always) "max" pooling
  - Often, they alternate with convolution, though this is not necessary

Recall: Max pooling



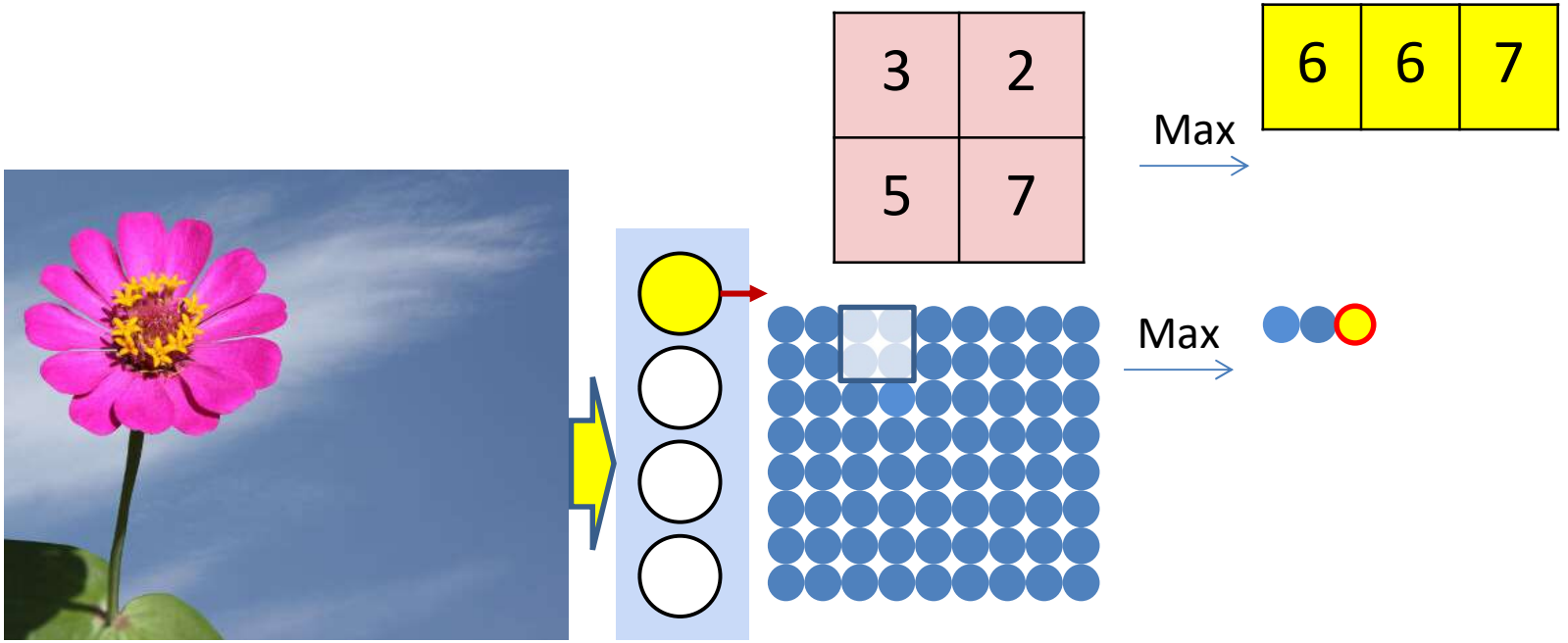
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

Recall: Max pooling



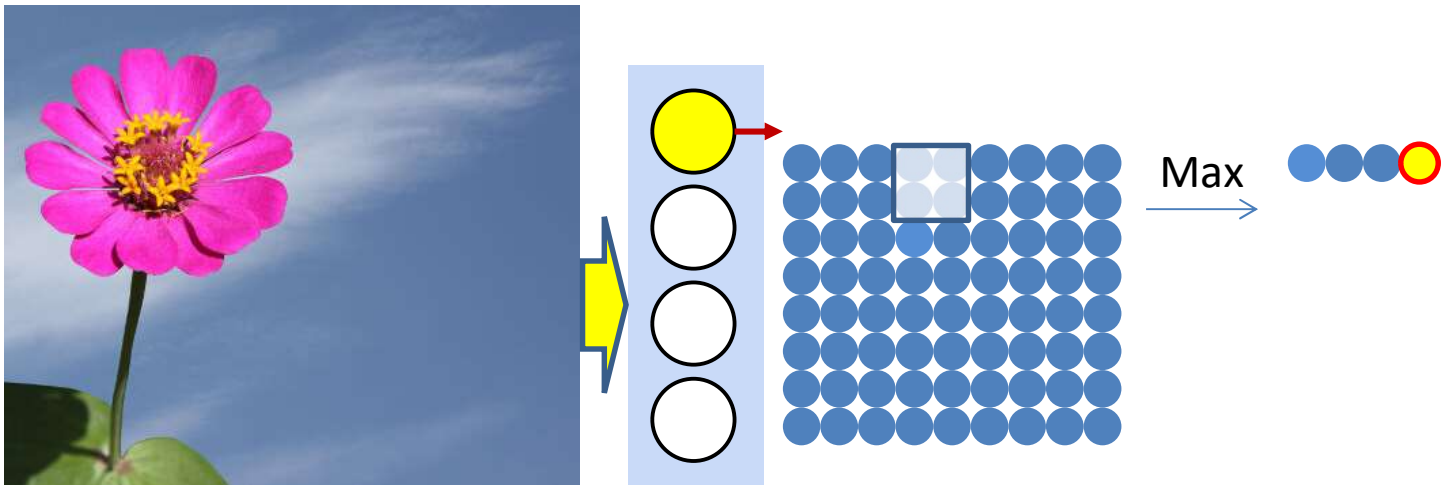
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

Recall: Max pooling



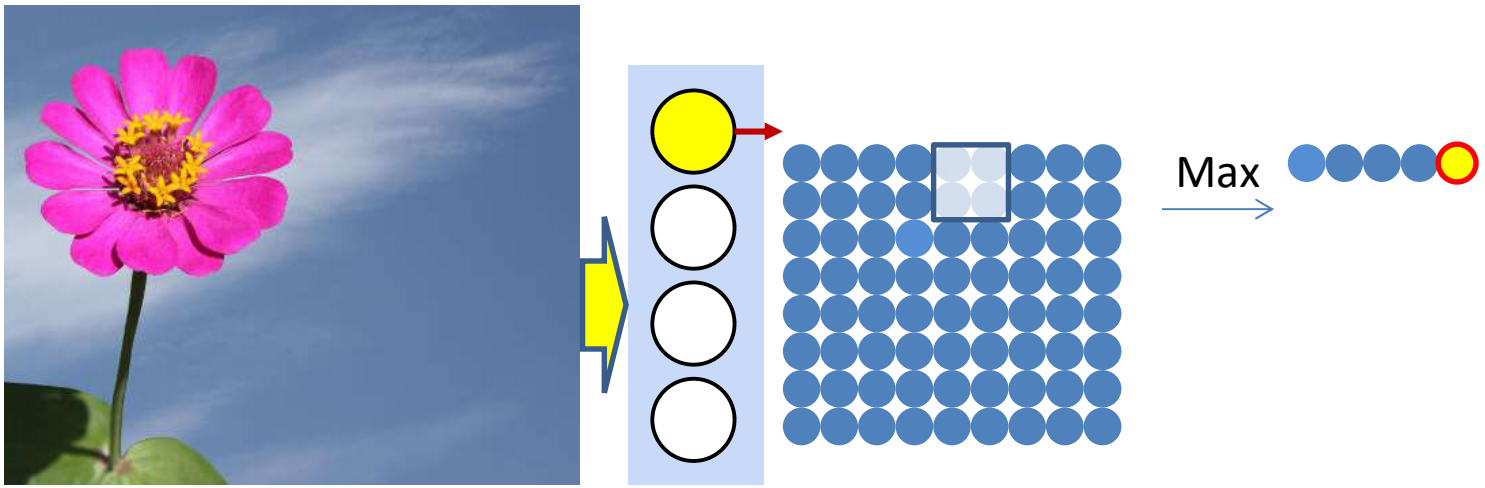
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

Recall: Max pooling



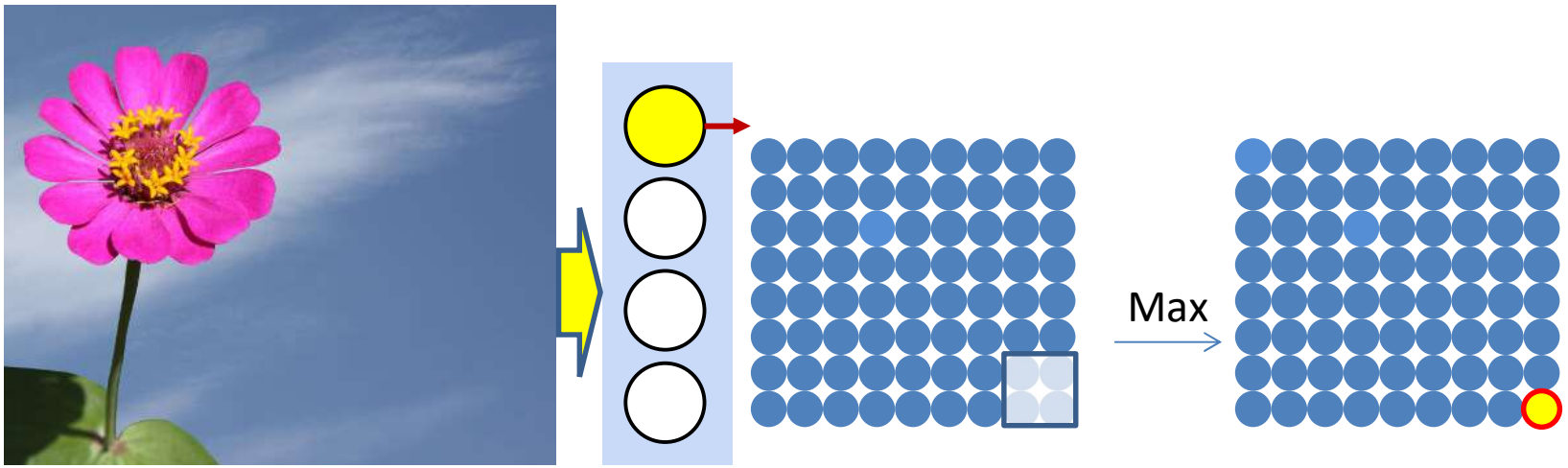
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

Recall: Max pooling



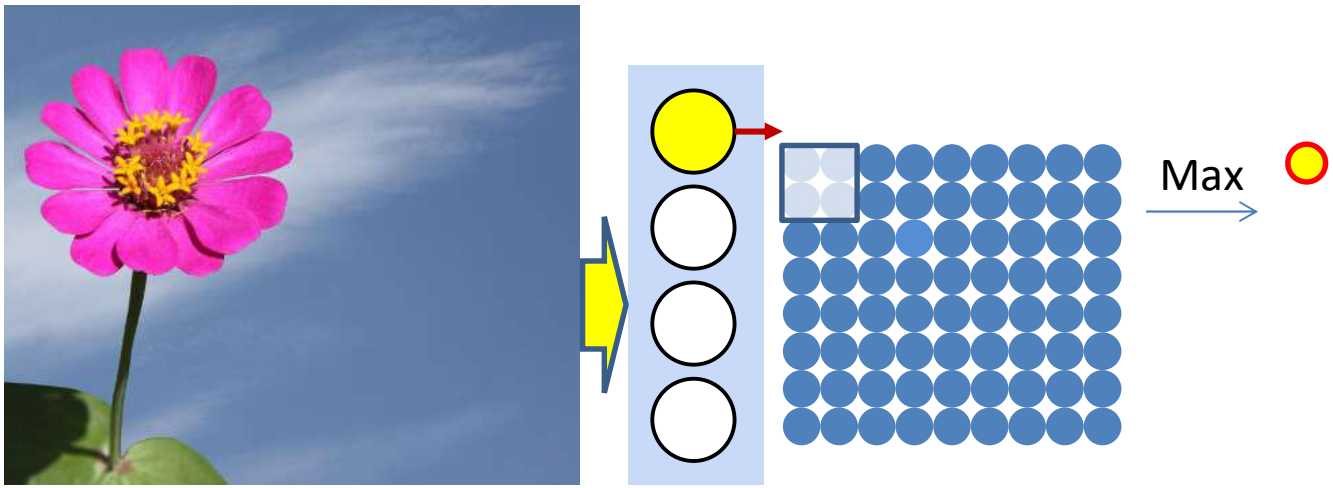
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

Recall: Max pooling



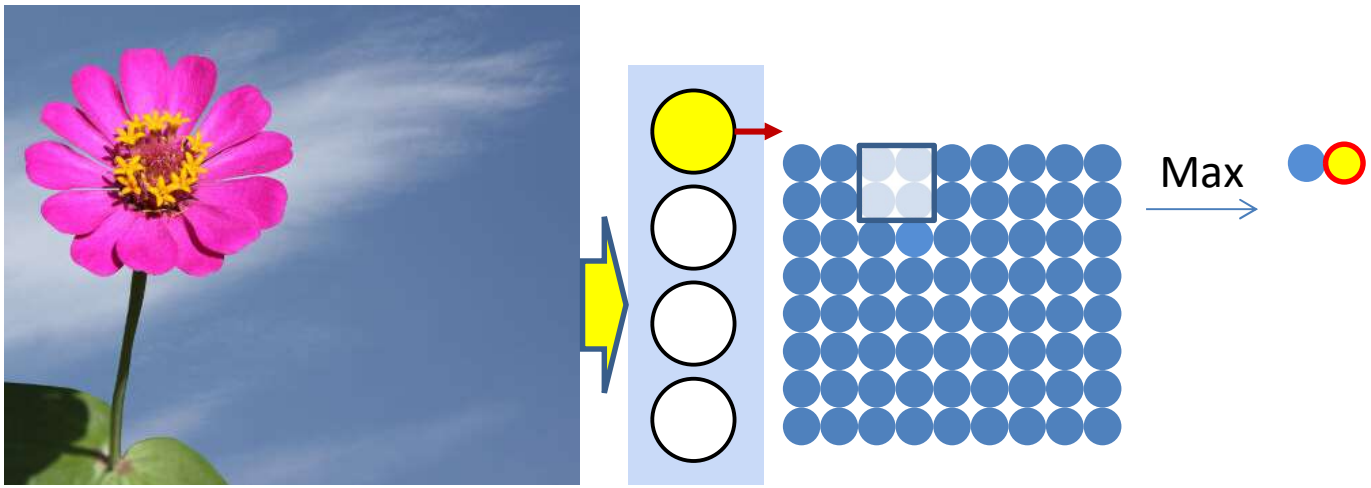
- Max pooling scans with a stride of 1 confer jitter-robustness, but do not constitute downsampling
- Downsampling requires a stride greater than 1

Downsampling requires Stride>1



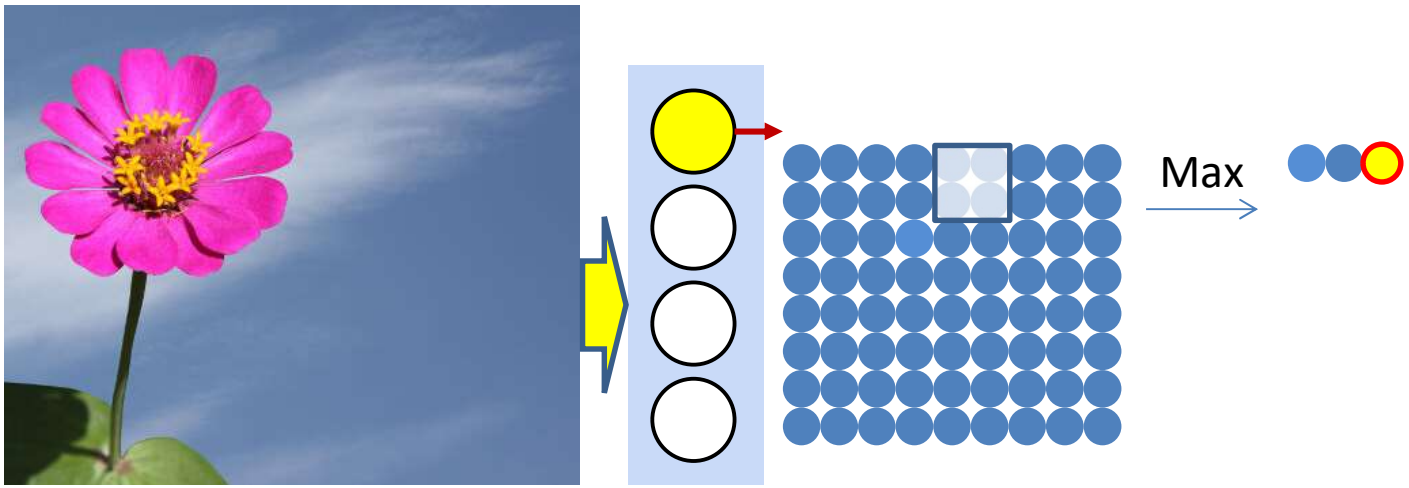
- The “max pooling” operation with “stride” greater than 1 results in an output smaller than the input
  - One output per stride
  - The output is “downsampled”

Downsampling requires Stride>1



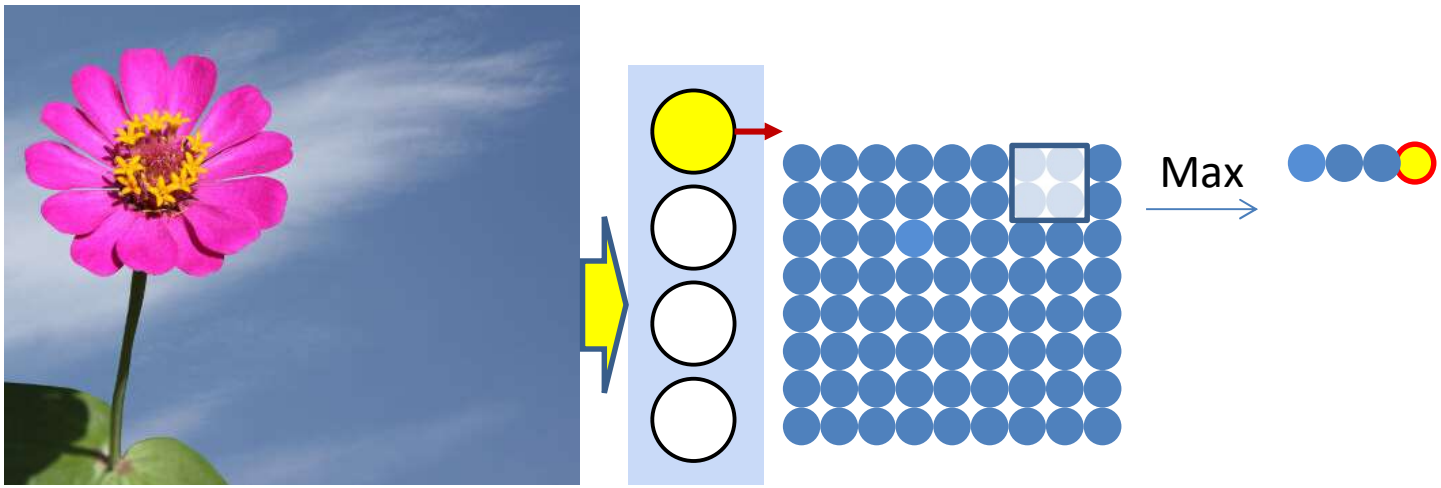
- The “max pooling” operation with “stride” greater than 1 results in an output smaller than the input
  - One output per stride
  - The output is “downsampled”

Downsampling requires Stride>1



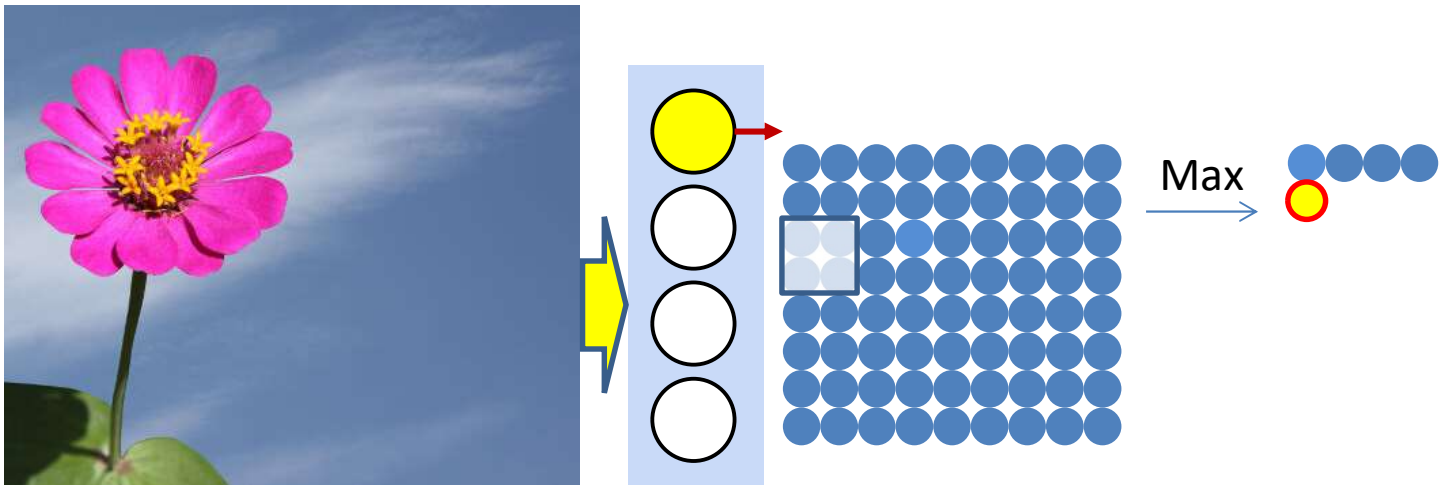
- The “max pooling” operation with “stride” greater than 1 results in an output smaller than the input
  - One output per stride
  - The output is “downsampled”

Downsampling requires Stride>1



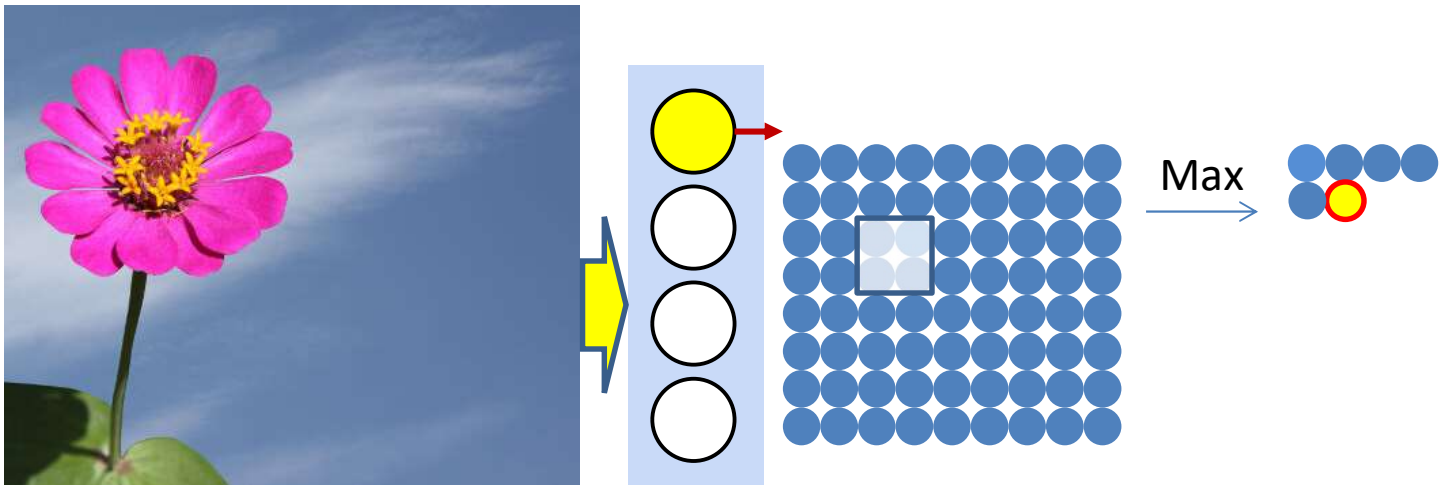
- The “max pooling” operation with “stride” greater than 1 results in an output smaller than the input
  - One output per stride
  - The output is “downsampled”

Downsampling requires Stride>1



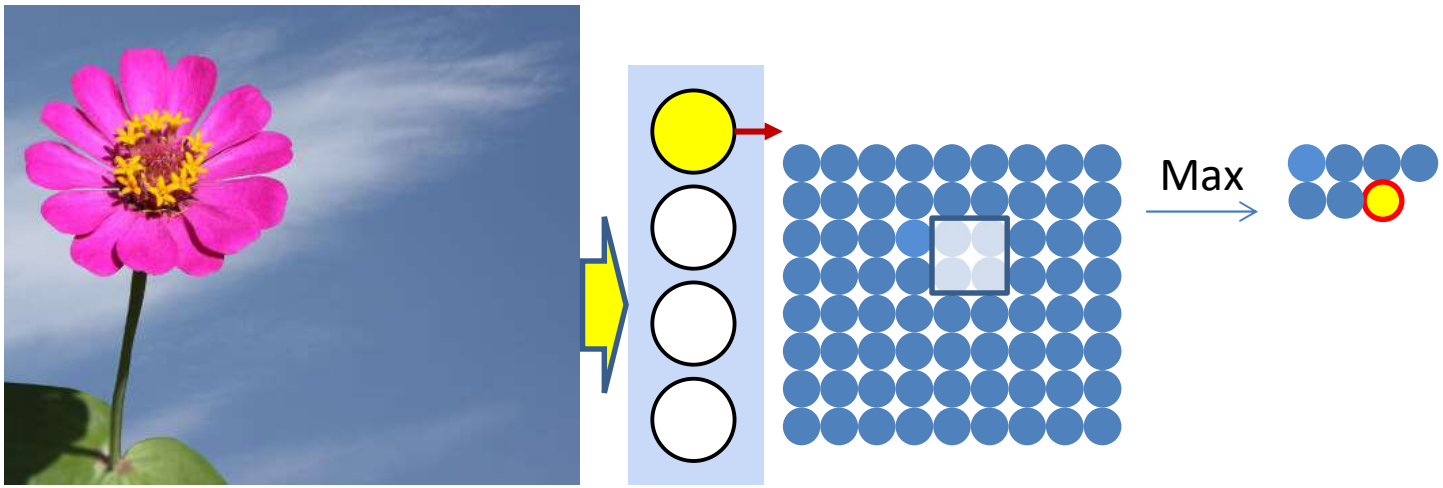
- The “max pooling” operation with “stride” greater than 1 results in an output smaller than the input
  - One output per stride
  - The output is “downsampled”

Downsampling requires Stride>1



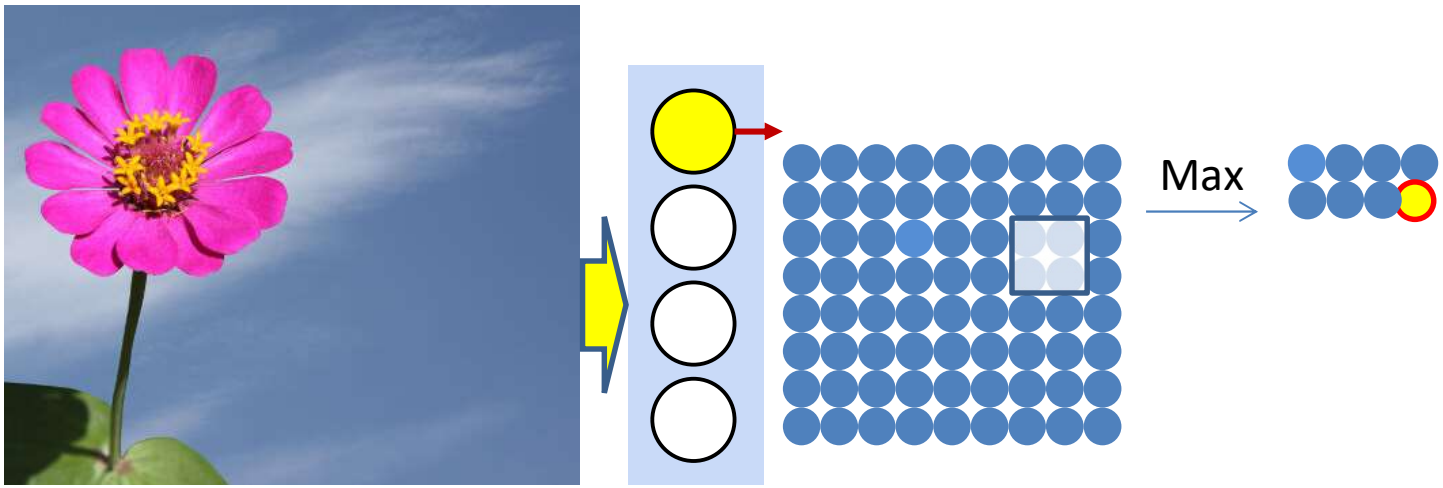
- The “max pooling” operation with “stride” greater than 1 results in an output smaller than the input
  - One output per stride
  - The output is “downsampled”

Downsampling requires Stride>1



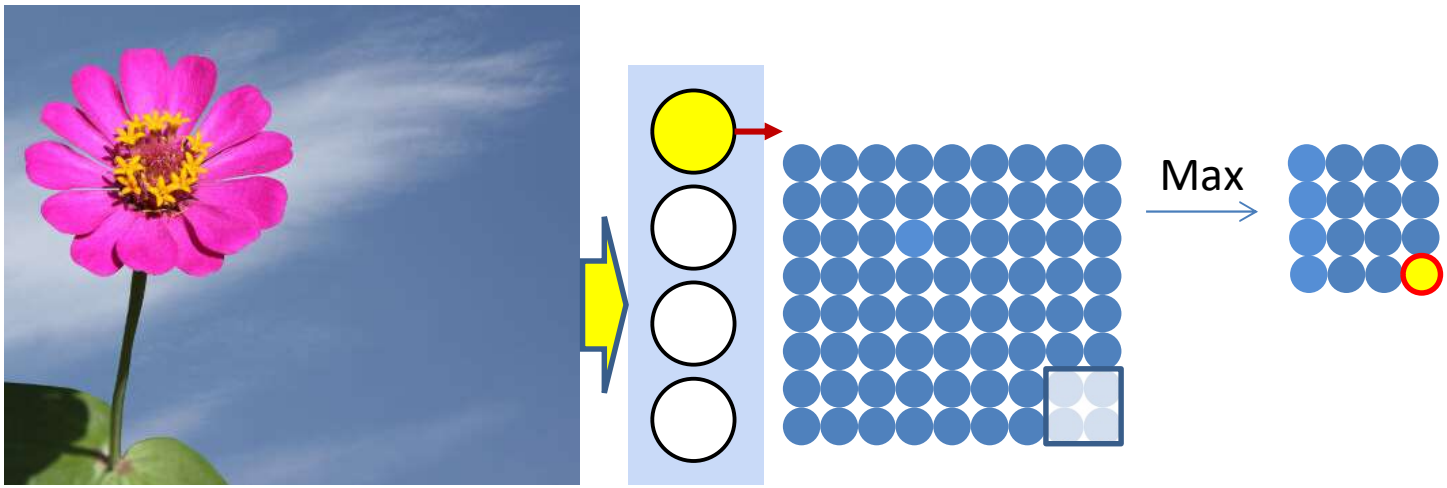
- The “max pooling” operation with “stride” greater than 1 results in an output smaller than the input
  - One output per stride
  - The output is “downsampled”

Downsampling requires Stride>1



- The “max pooling” operation with “stride” greater than 1 results in an output smaller than the input
  - One output per stride
  - The output is “downsampled”

Downsampling requires Stride>1



- The “max pooling” operation with “stride” greater than 1 results in an output smaller than the input
  - One output per stride
  - The output is “downsampled”

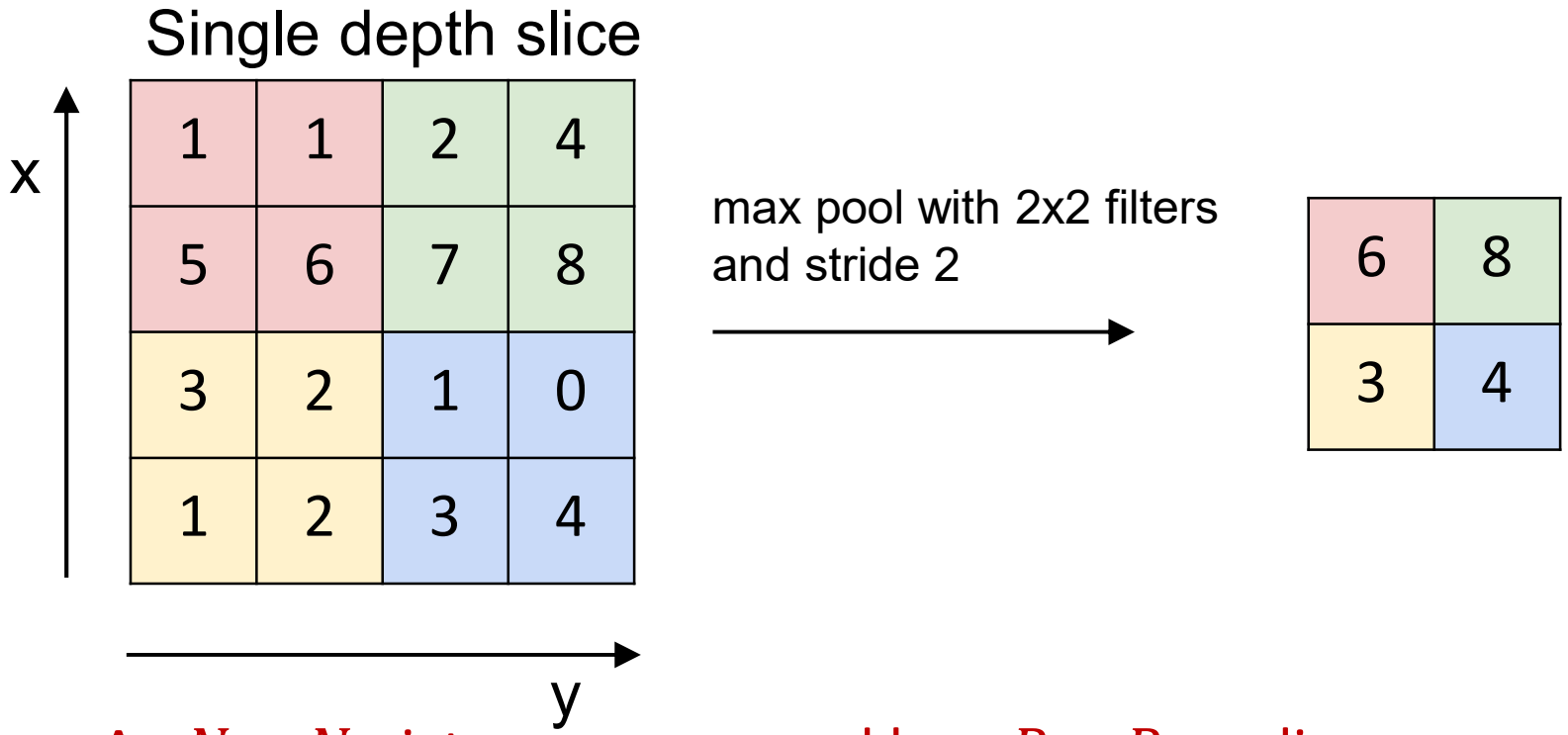
Max Pooling layer at layer l

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

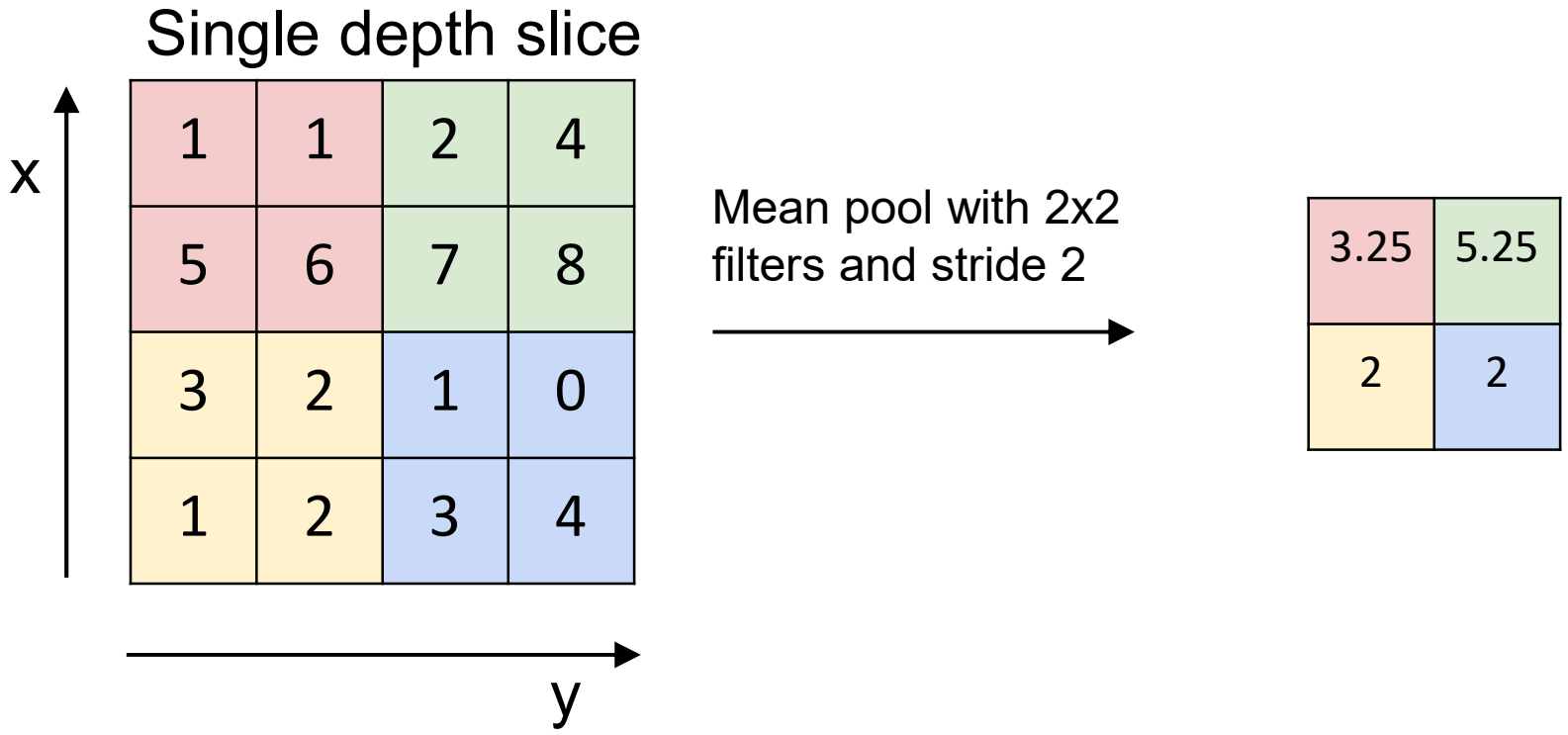
```
for j = 1:D_l
   m = 1
   for x = 1:stride(l):W_{l-1}-K_l+1
     n = 1
     for y = 1:stride(l):H_{l-1}-K_l+1
       pidx(l,j,m,n) = maxidx(Y(l-1,j,x:x+K_l-1,y:y+K_l-1))
       Y(l,j,m,n) = Y(l-1,j,pidx(l,j,m,n))
       n = n+1
     m = m+1
```

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

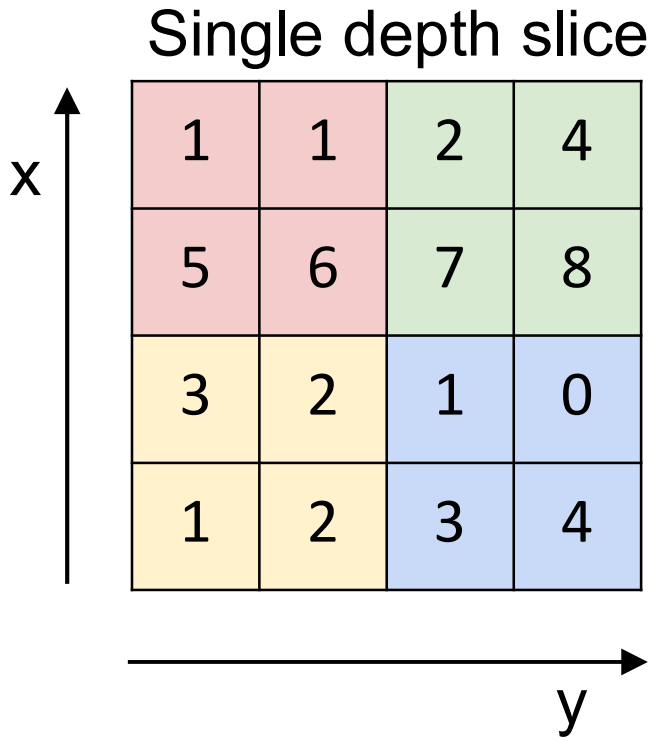
Mean Pooling layer at layer l

a) Performed separately for every map (j)

Mean pooling

```
for j = 1:D_l
    m = 1
    for x = 1:stride(l):W_{l-1}-K_l+1
        n = 1
        for y = 1:stride(l):H_{l-1}-K_l+1
            Y(l,j,m,n) = mean(Y(l-1,j,x:x+K_l-1,y:y+K_l-1))
            n = n+1
        m = m+1
```

Alternative to Max pooling: p-norm



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

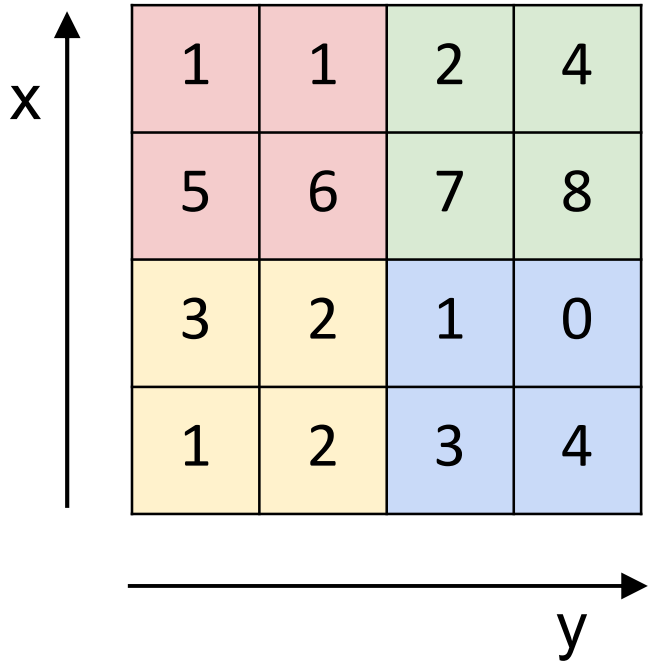
$$y = \sqrt[p]{\frac{1}{K^2}\sum_{i,j} x_{ij}^p}$$

4.86	8
2.38	3.16

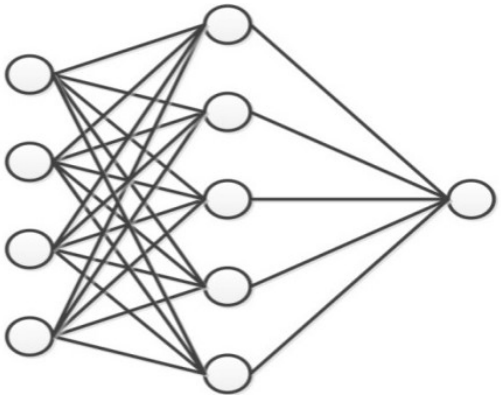
- Compute a p-norm of the pool

Other options

Single depth slice



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

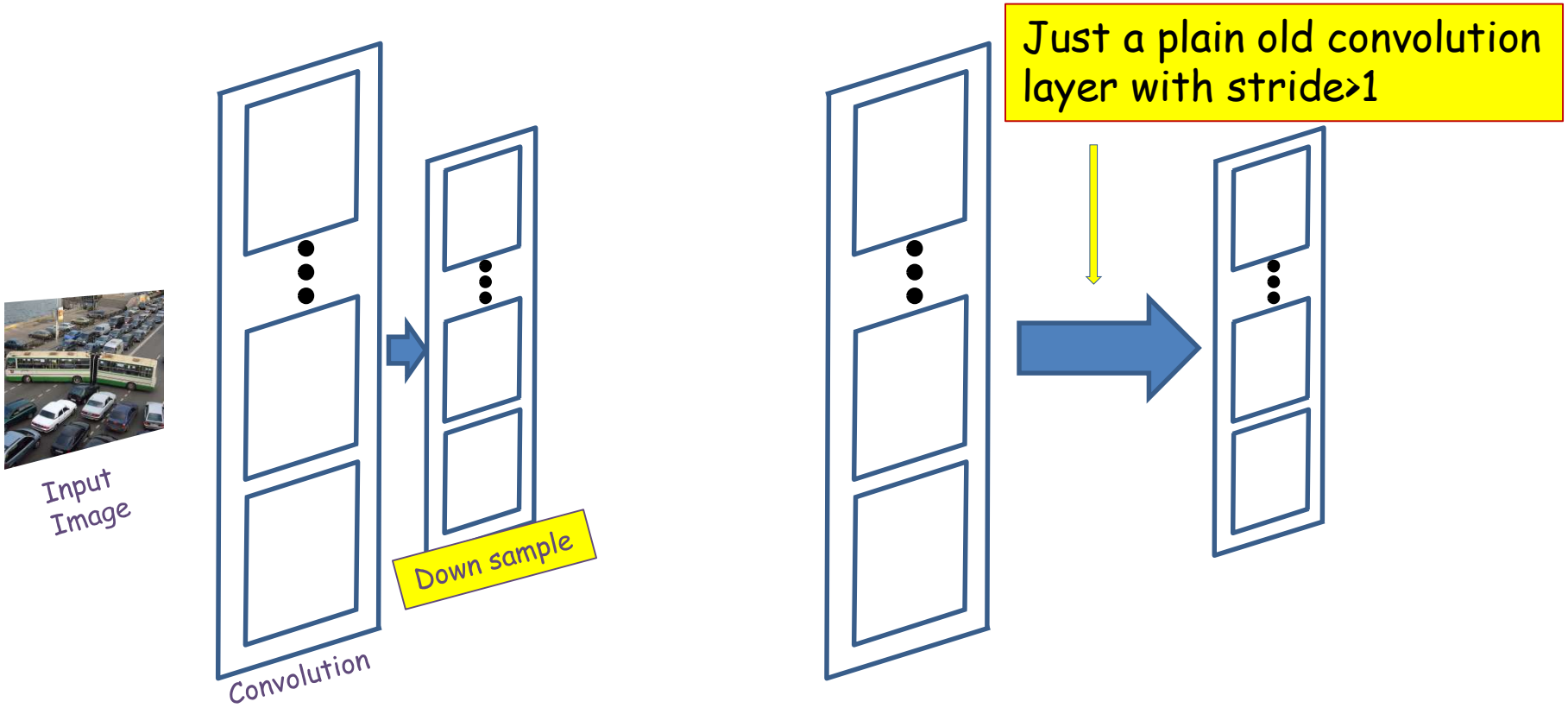


6	8
3	4

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}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 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
       for j = 1:D_l
          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..

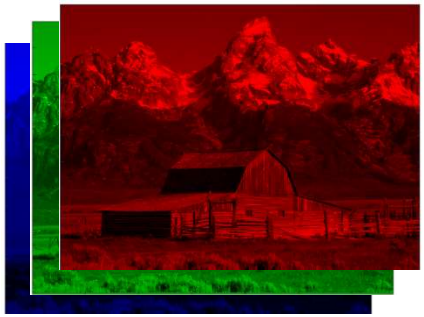
Convolutional Neural Networks

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

Convolutional Neural Networks

- Input: 3 pictures

Convolutional Neural Networks

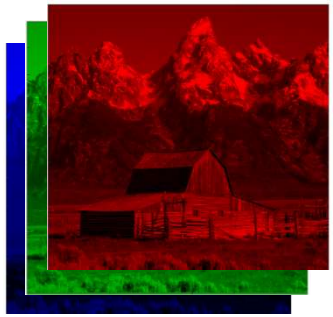


- Input: 3 pictures

Preprocessing

- Large images are a problem
  - Too much detail
  - Will need big networks
- Typically scaled to small 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

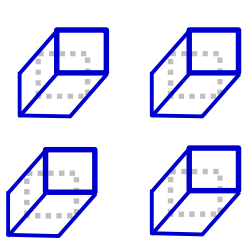
Convolutional Neural Networks



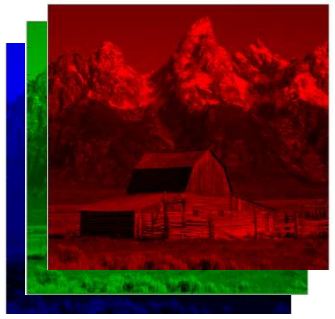
$I \times I$ *image*

- Input: 3 pictures

Convolutional Neural Networks



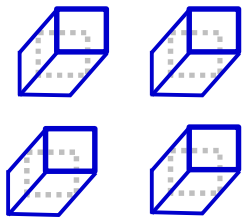
K_1 total filters
Filter size: $L \times L \times 3$



$I \times I$ *image*

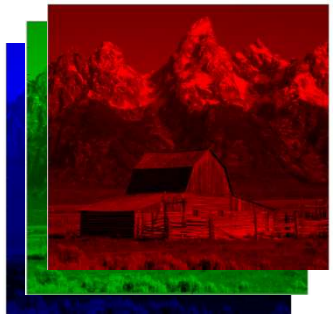
- Input is convolved with a set of K_1 filters
  - Typically K_1 is a power of 2, e.g. 2, 4, 8, 16, 32,..
  - Filters are typically 5x5, 3x3, or even 1x1

Convolutional Neural Networks



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

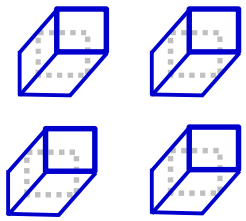
Small enough to capture fine features
(particularly important for scaled-down images)



$I \times I$ *image*

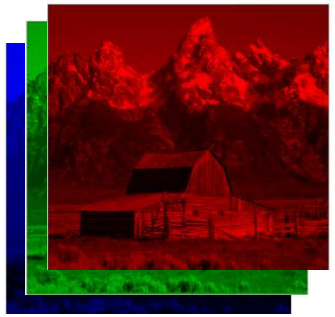
- Input is convolved with a set of K_1 filters
  - Typically K_1 is a power of 2, e.g. 2, 4, 8, 16, 32,..
  - Filters are typically 5x5, 3x3, or even 1x1

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)

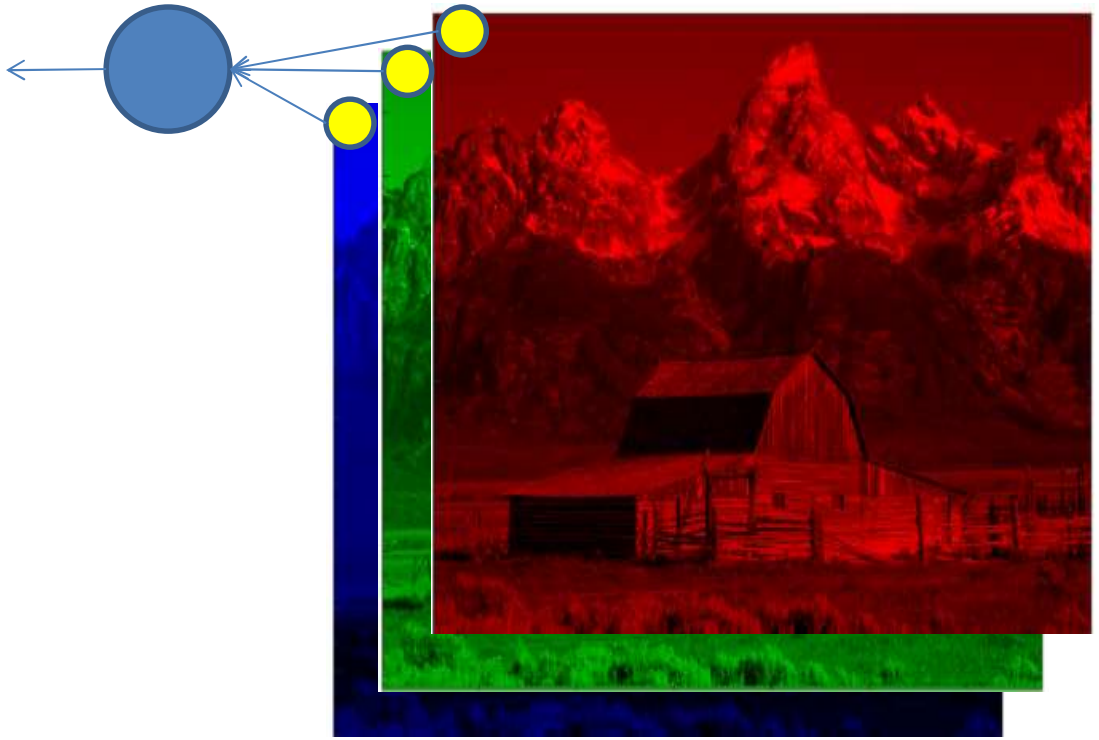


$I \times I$ *image*

What on earth is this?

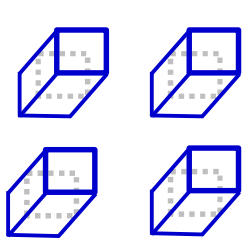
- Input is convolved with a set of K_1 filters
  - Typically K_1 is a power of 2, e.g. 2, 4, 8, 16, 32,..
  - Filters are typically 5x5, 3x3, or even 1x1

The 1x1 filter

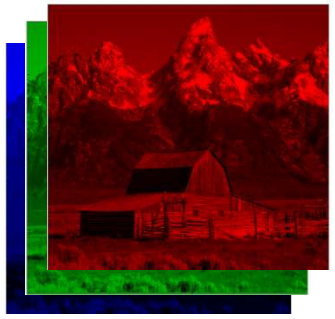


- A 1x1 filter is simply a perceptron that operates over the *depth* of the stack of maps, but has no spatial extent
  - Takes one pixel from each of the maps (at a given location) as input

Convolutional Neural Networks



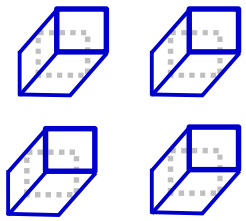
K_1 total filters
Filter size: $L \times L \times 3$



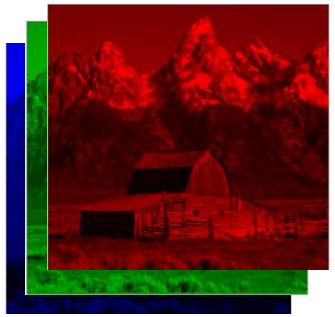
$I \times I$ *image*

- 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*

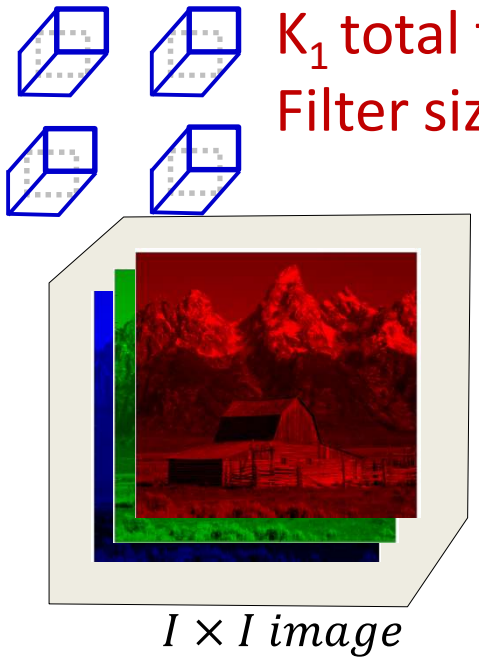
Parameters to choose: K_1, L and S

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: $K_1(3L^2 + 1)$

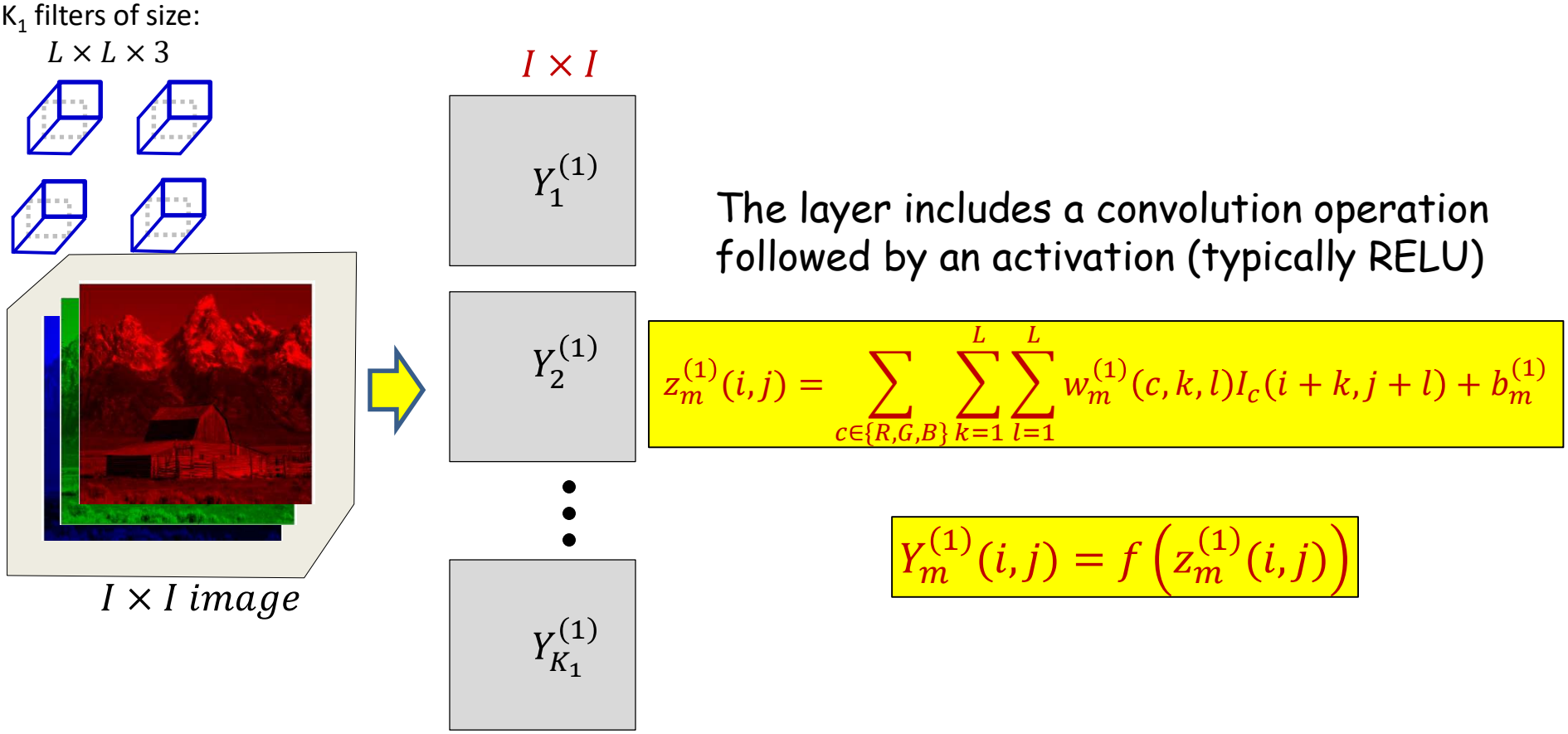
- Input is convolved with a set of K_1 filters
  - Typically K_1 is a power of 2, e.g. 2, 4, 8, 16, 32,..
  - **Better notation:** Filters are typically 5x5(x3), 3x3(x3), or even 1x1(x3)
  - ***Typical stride:*** 1 or 2

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

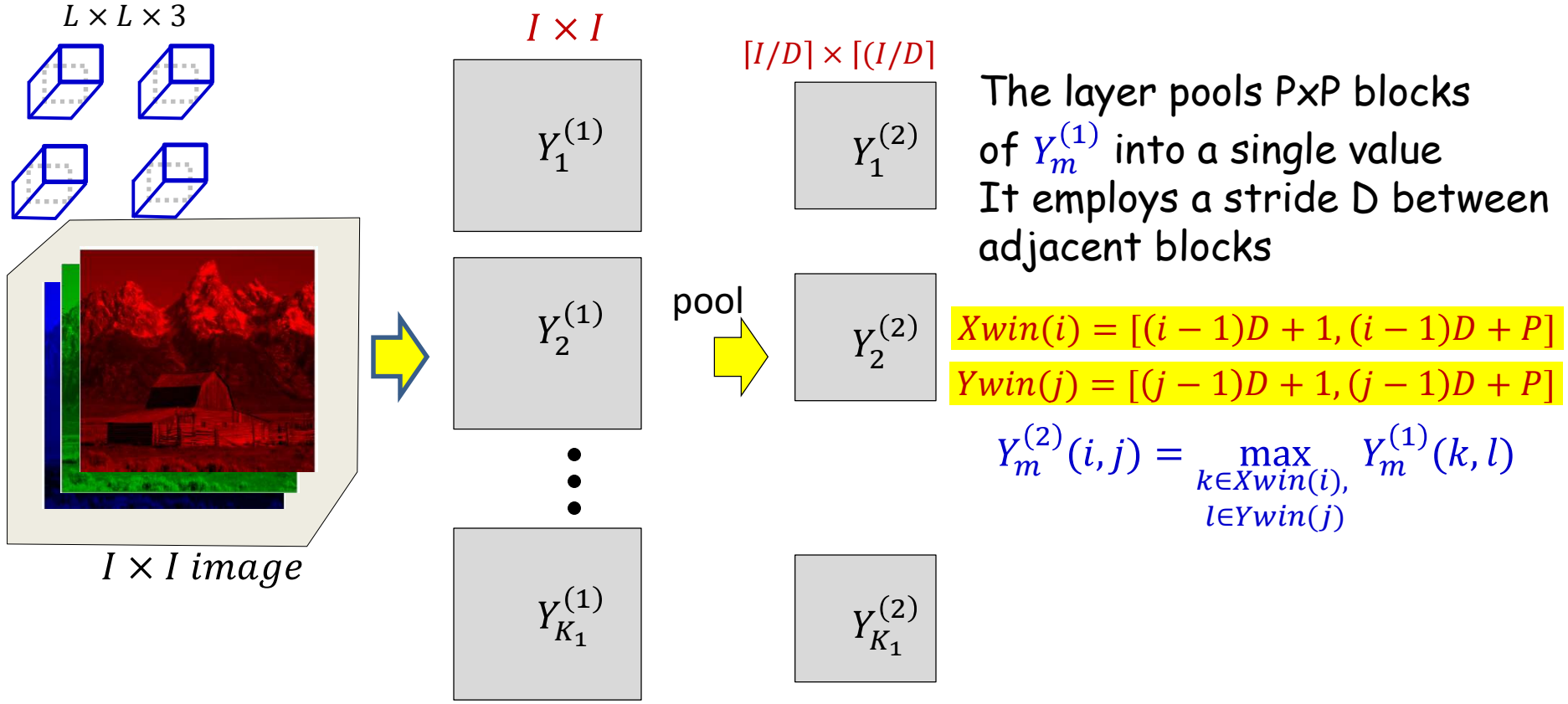
Learnable parameters in the first convolutional layer

- 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

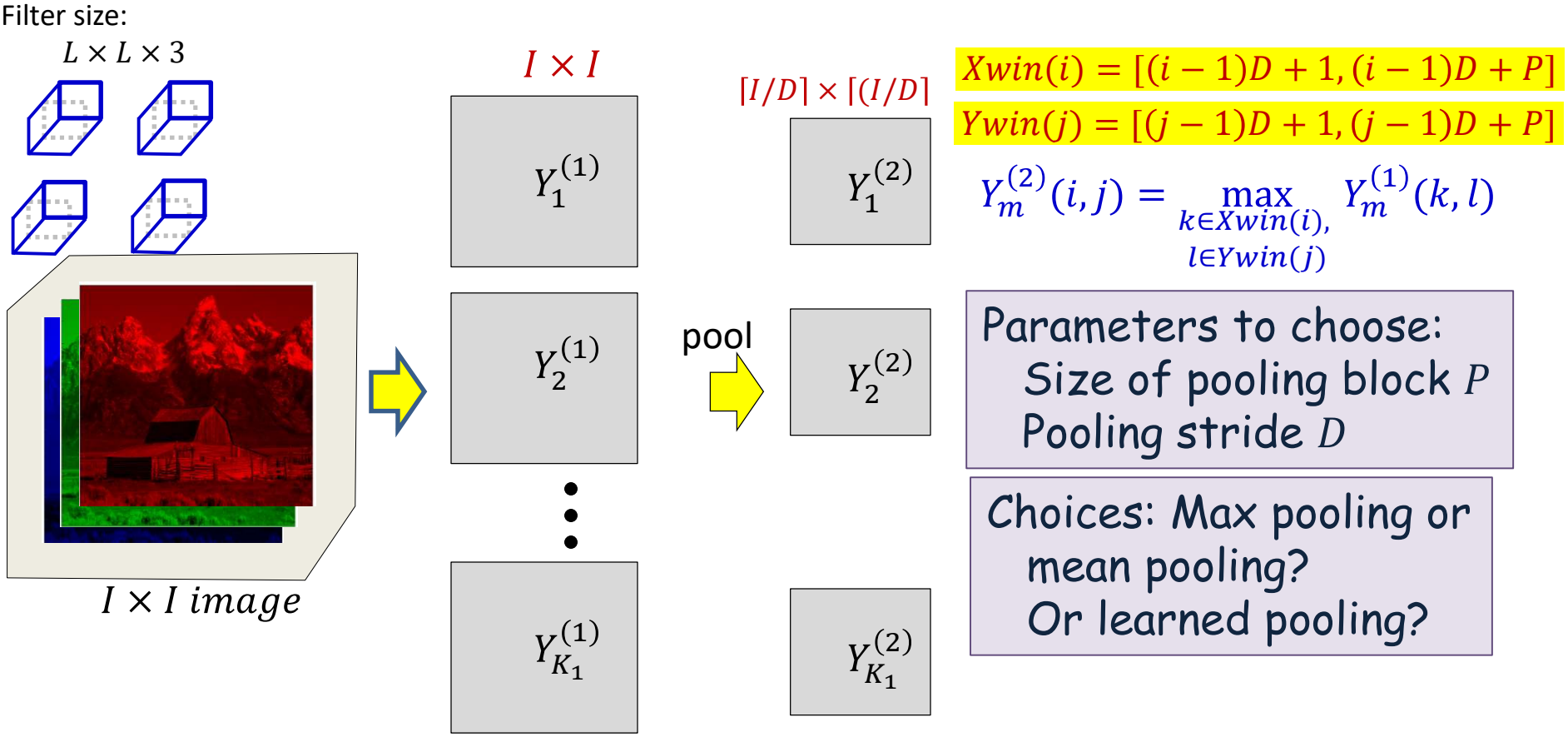
- All of these parameters must be learned

Convolutional Neural Networks



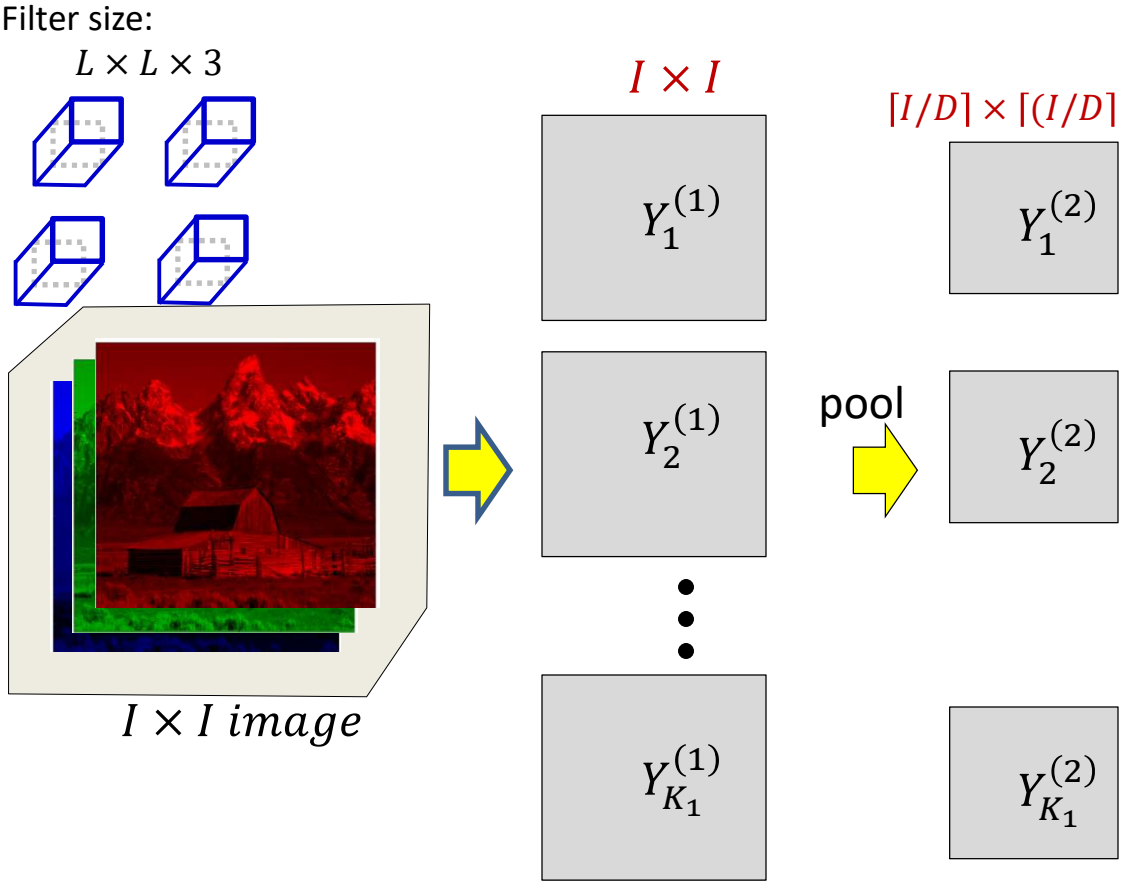
- **First downsampling layer:** From each $P \times P$ block of each map, *pool* down to a single value
  - For max pooling, during training keep track of which position had the highest value

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



$$Xwin(i) = [(i-1)D + 1, (i-1)D + P]$$

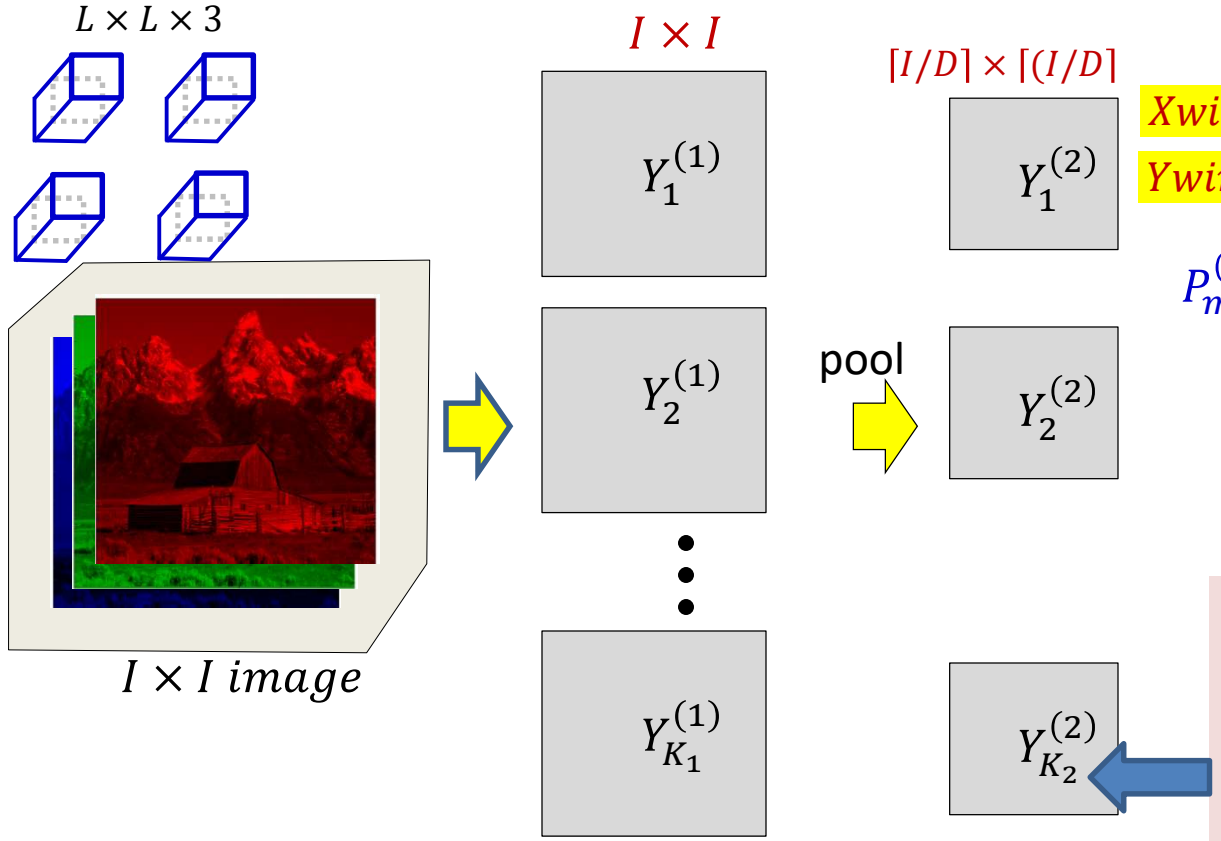
$$Ywin(j) = [(j-1)D + 1, (j-1)D + P]$$

$$P_m^{(2)}(i,j) = \underset{\substack{k \in Xwin(i), \\ l \in Ywin(j)}}{\operatorname{argmax}} Y_m^{(1)}(k,l)$$

$$Y_m^{(2)}(i,j) = Y_m^{(1)}(P_m^{(2)}(i,j))$$

- **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



$$Xwin(i) = [(i-1)D + 1, (i-1)D + P]$$

$$Ywin(j) = [(j-1)D + 1, (j-1)D + P]$$

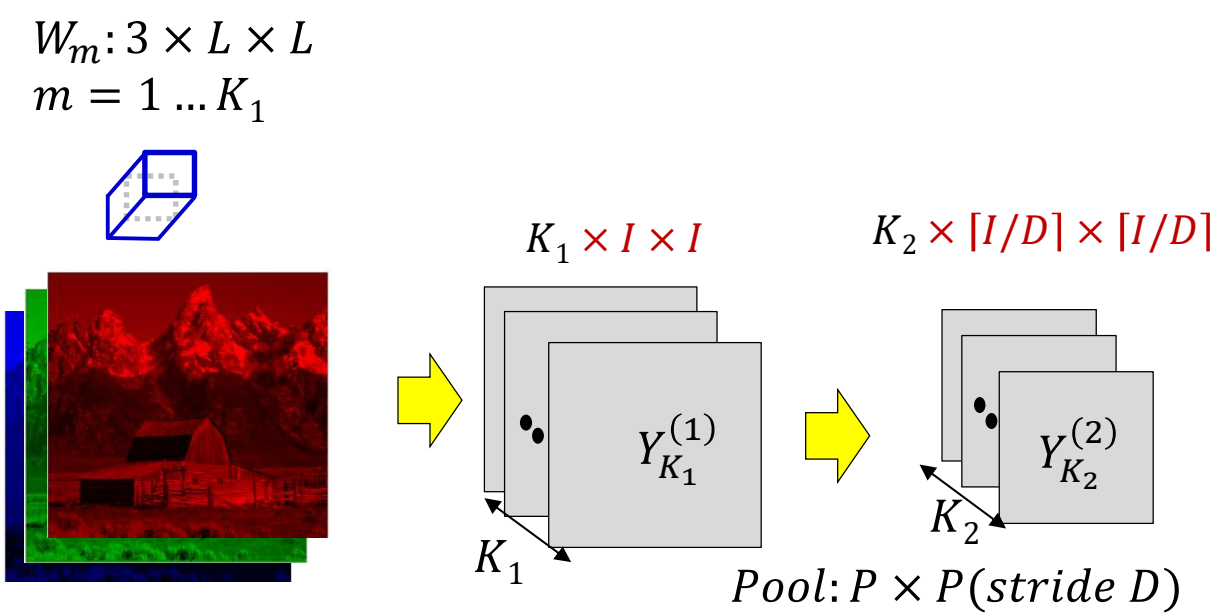
$$P_m^{(2)}(i,j) = \underset{\substack{k \in Xwin(i), \\ l \in Ywin(j)}}{\operatorname{argmax}} Y_m^{(1)}(k,l)$$

$$Y_m^{(2)}(i,j) = Y_m^{(1)}(P_m^{(2)}(i,j))$$

$K_2 = K_1$. Just using the new index K_2 for notational uniformity.
Pooling layers do not change the number of maps because pooling is performed individually on each of the maps in the previous layer.

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

Convolutional Neural Networks

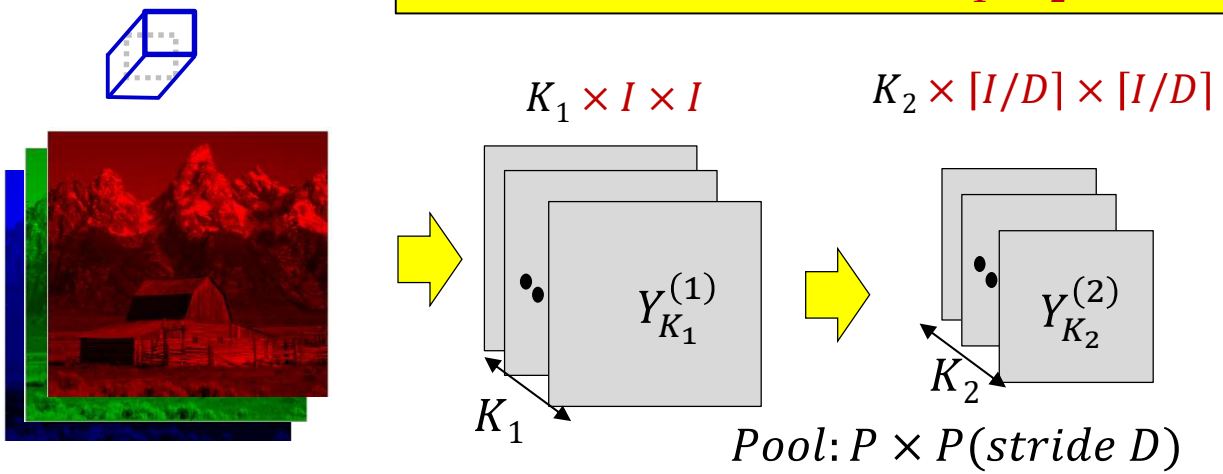


- **First pooling layer:** Drawing it differently for convenience

Convolutional Neural Networks

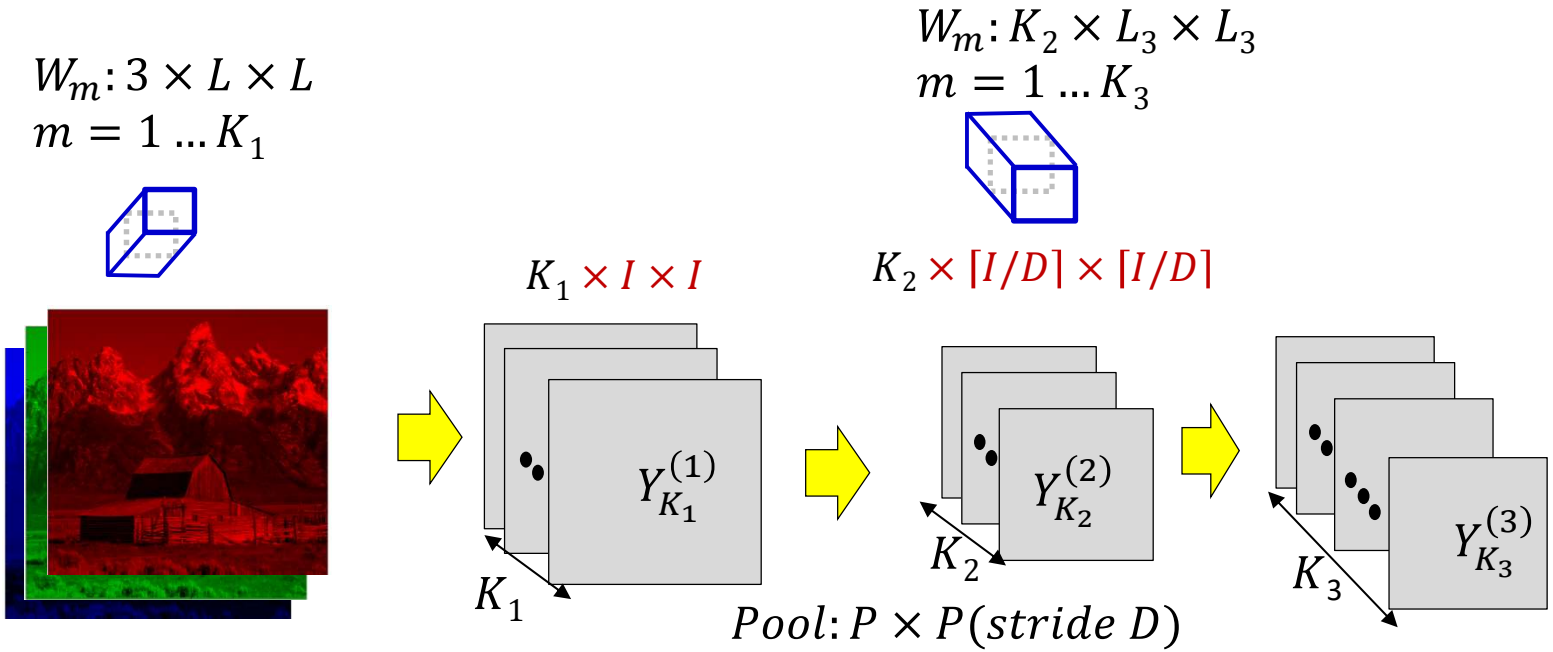
$W_m: 3 \times L \times L$
$m = 1 \ldots K_1$

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

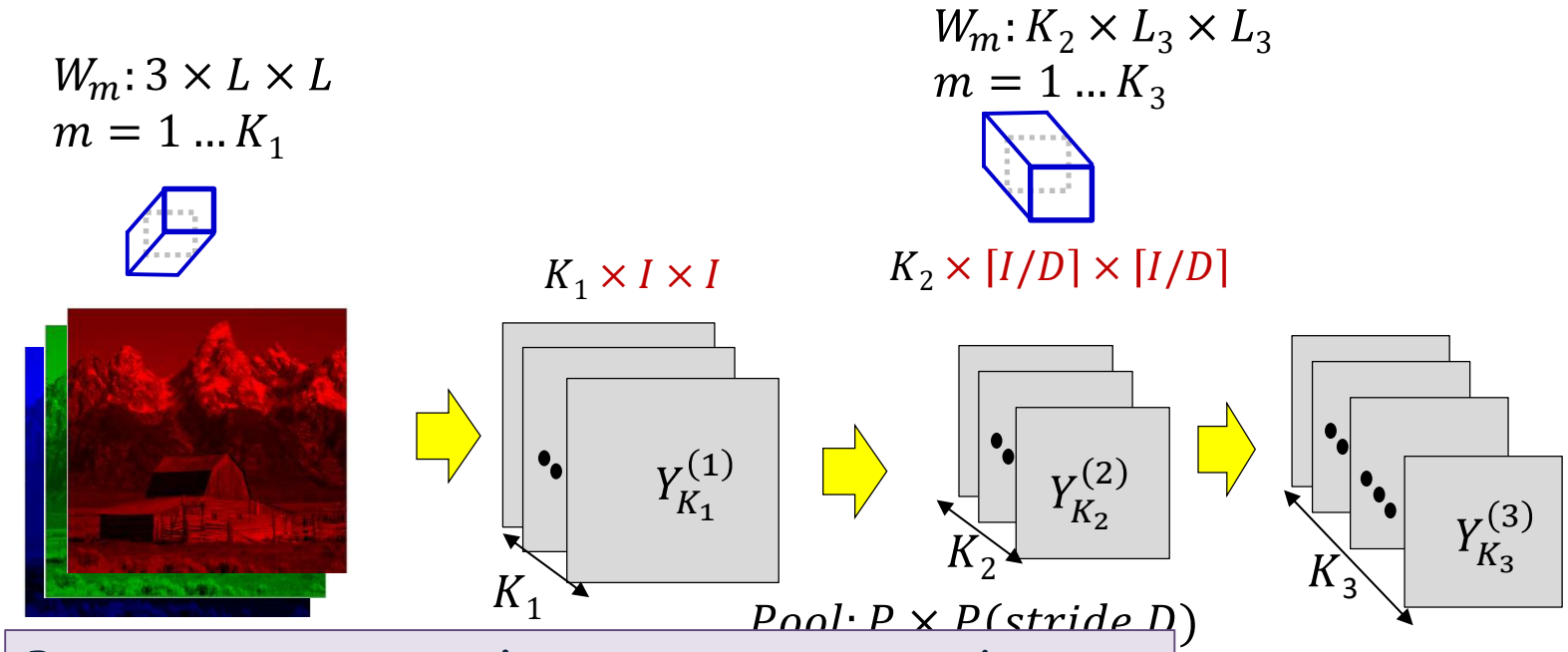


$$z_m^{(n)}(i,j) = \sum_{r=1}^{K_{n-1}} \sum_{k=1}^{L_n} \sum_{l=1}^{L_n} w_m^{(n)}(r,k,l) Y_r^{(n-1)}(i+k, j+l) + b_m^{(n)}$$

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

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

Convolutional Neural Networks



Parameters to choose: K_3, L_3 and S_3

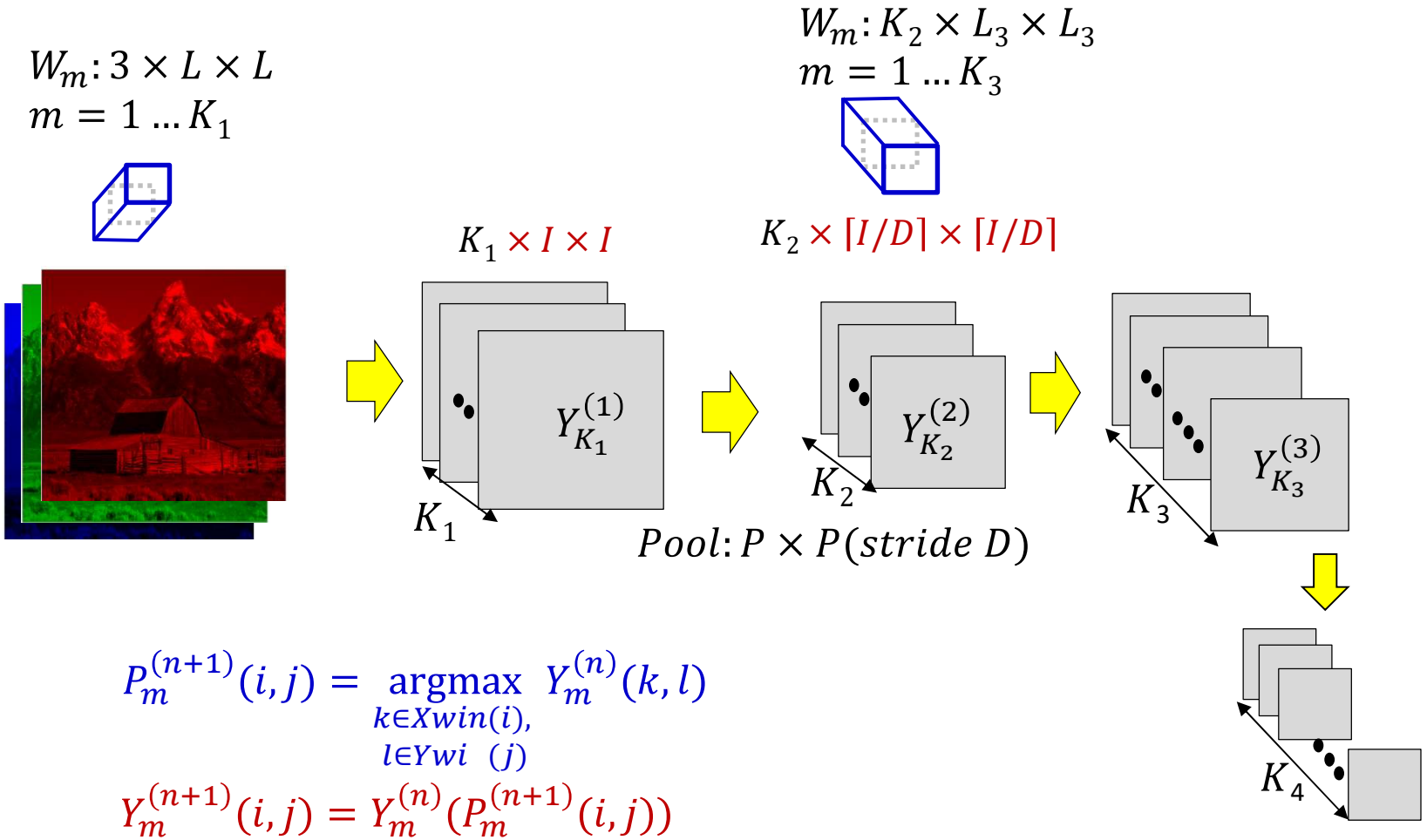
1. Number of filters K_3
2. Size of filters $L_3 \times L_3 \times K_2 + bias$
3. *Stride* of convolution S_3

Total number of parameters: $K_3(K_2 L_3^2 + 1)$

All these parameters must be learned

ng in K_3 2-D maps

Convolutional Neural Networks

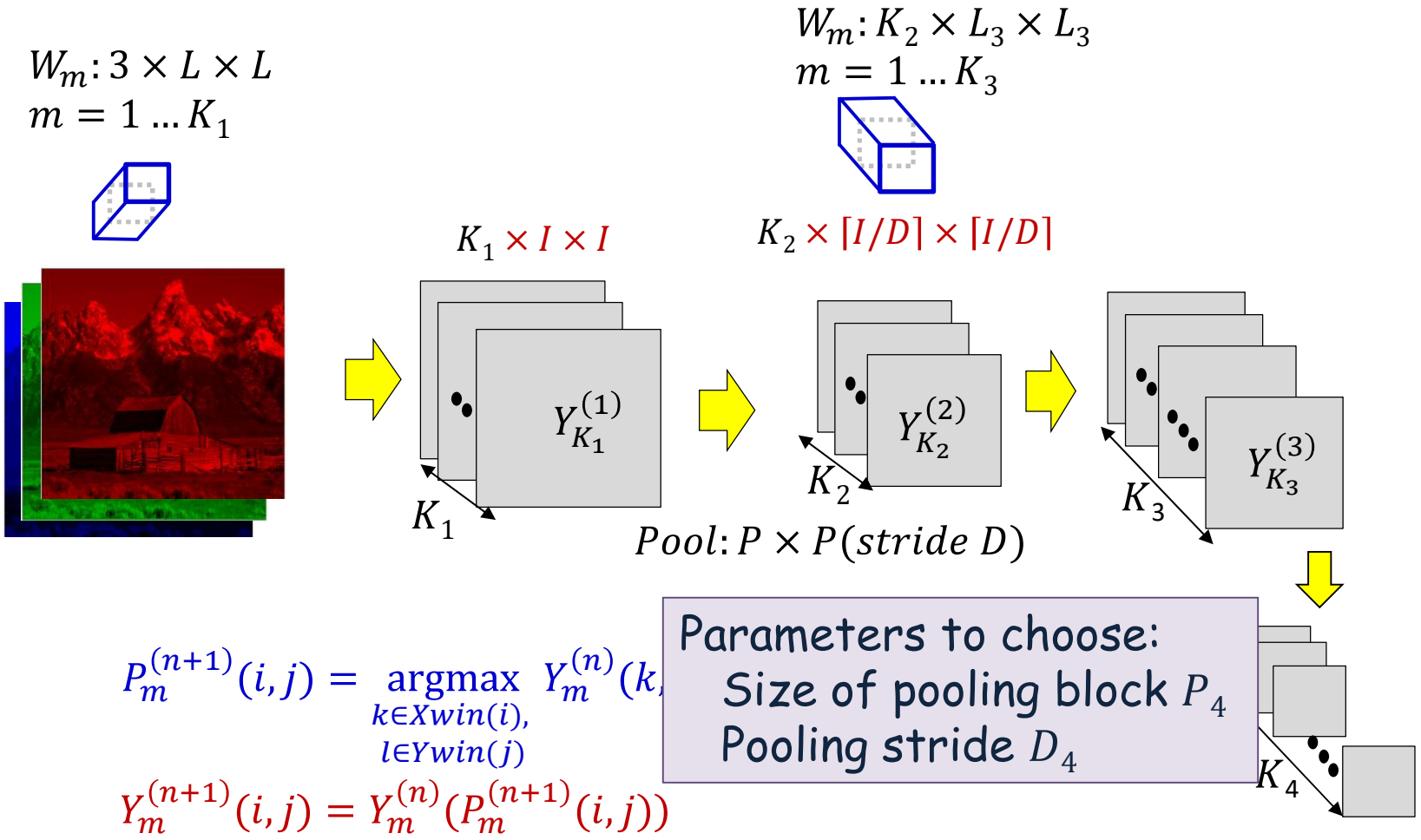


$$P_m^{(n+1)}(i,j) = \underset{\substack{k \in Xwin(i), \\ l \in Ywi\ (j)}}{\operatorname{argmax}} Y_m^{(n)}(k,l)$$

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

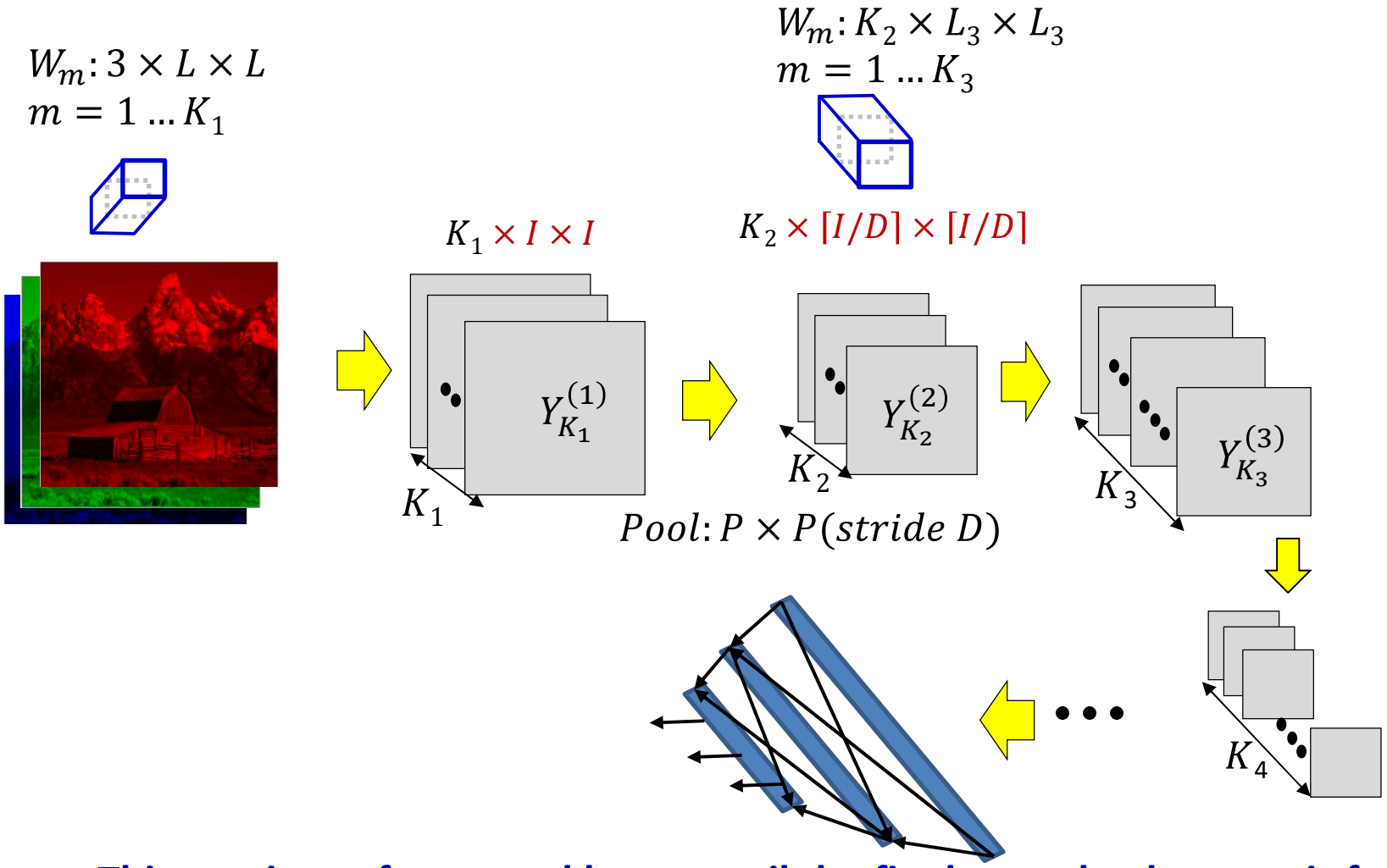
- **Second convolutional layer:** K_2 3-D filters resulting in K_2 2-D maps
- **Second pooling layer**: K_2 Pooling operations: outcome K_2 reduced 2D maps

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**

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)

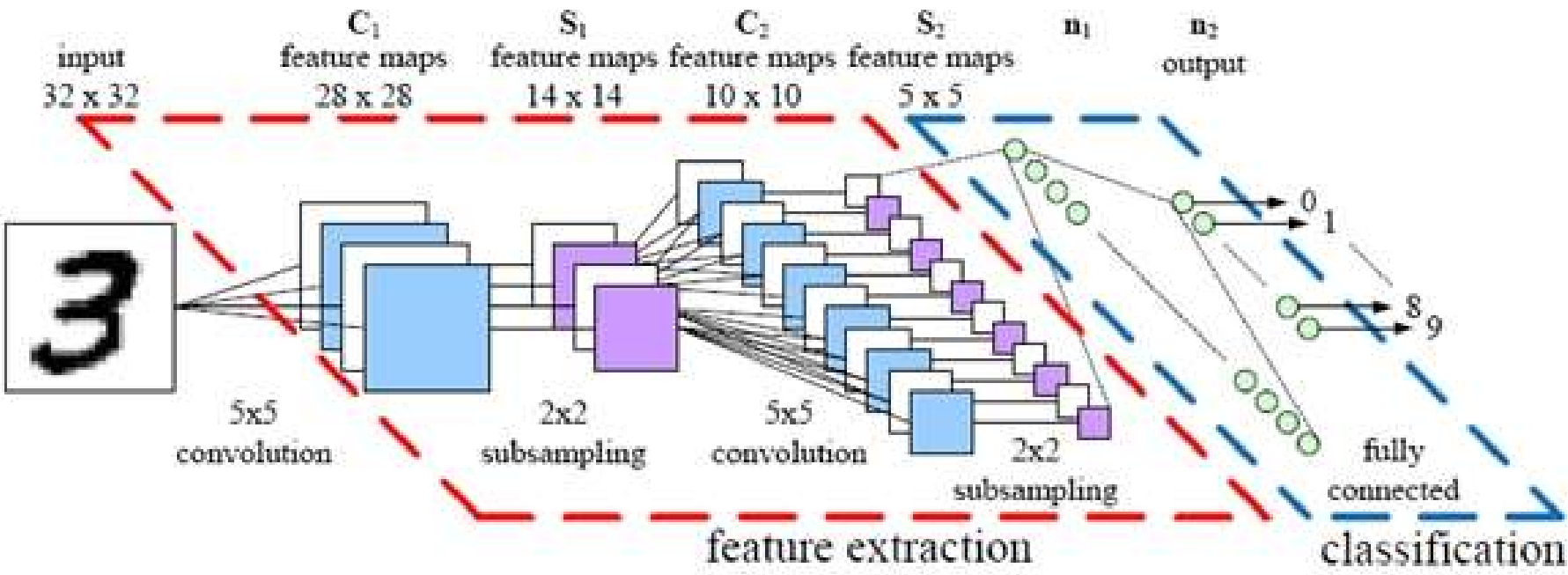
- Number of convolutional and downsampling layers
  - And arrangement (order in which they follow one another)

- 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

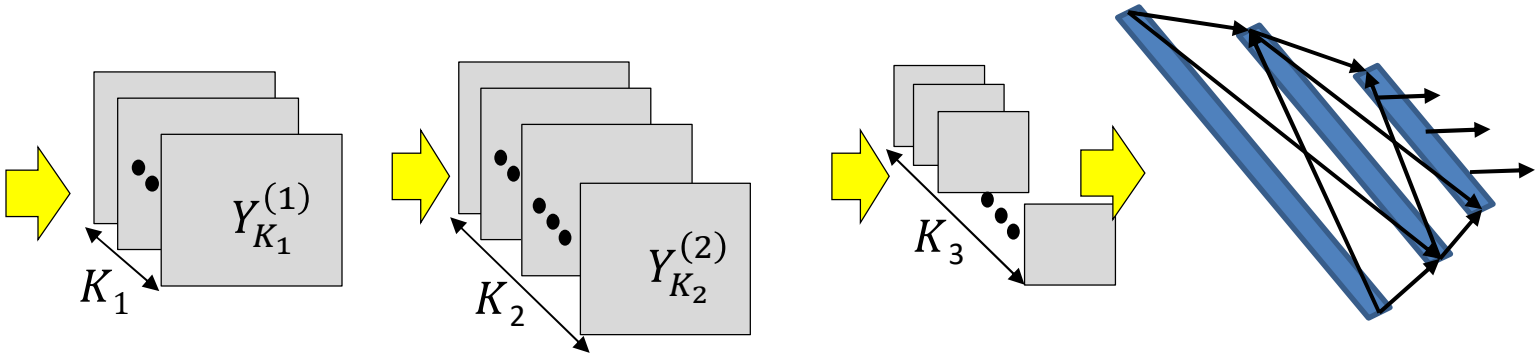
- For each downsampling/pooling layer:
  - Spatial extent of filter $P_i \times P_i$
  - The stride D_i

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

Digit classification



Training

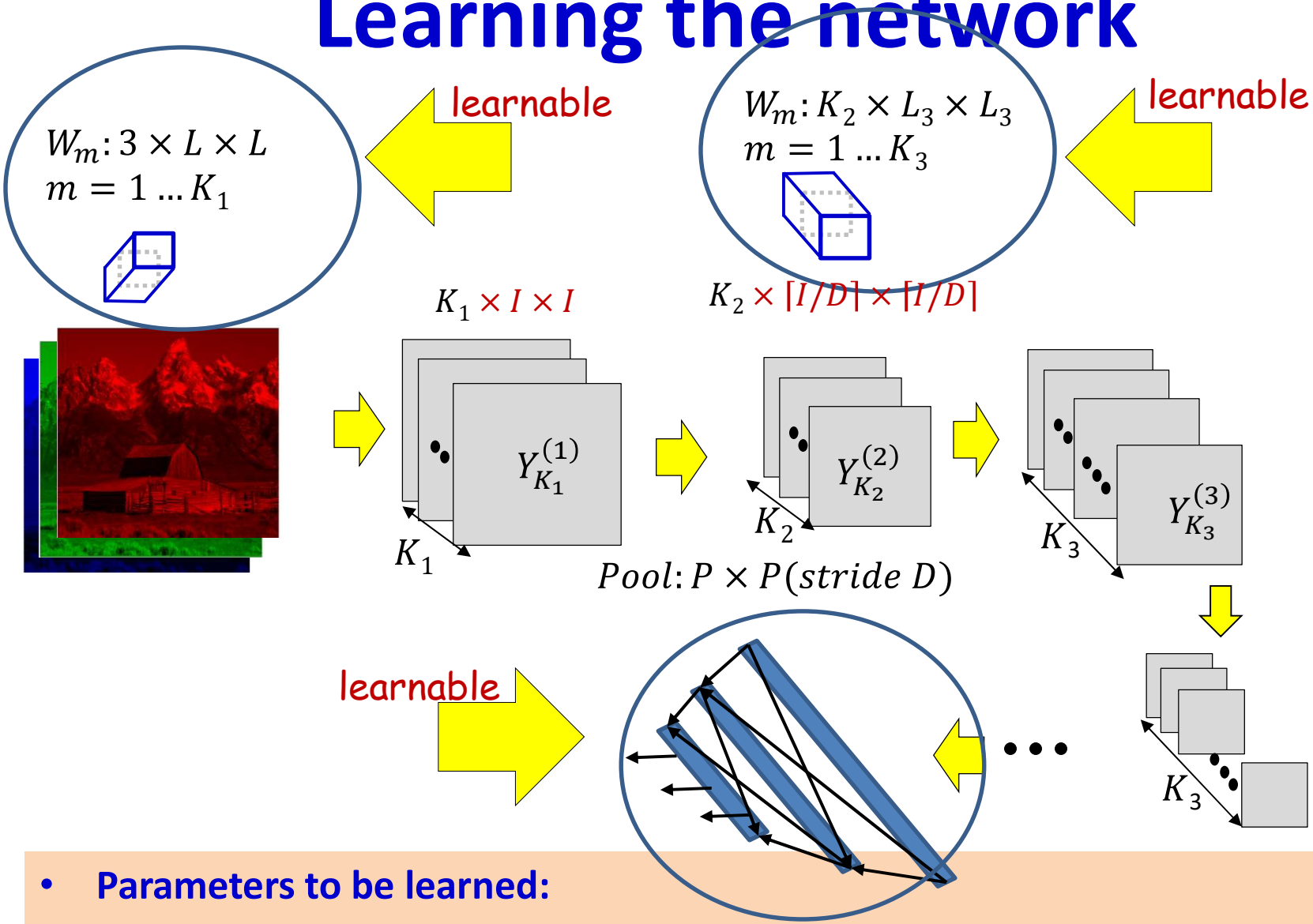


- Training is as in the case of the regular MLP
  - The *only* difference is in the *structure* of the network
- **Training examples of (Image, class) are provided**
- Define a divergence between the desired output and true output of the network in response to any input
- **Network parameters are trained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

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

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**

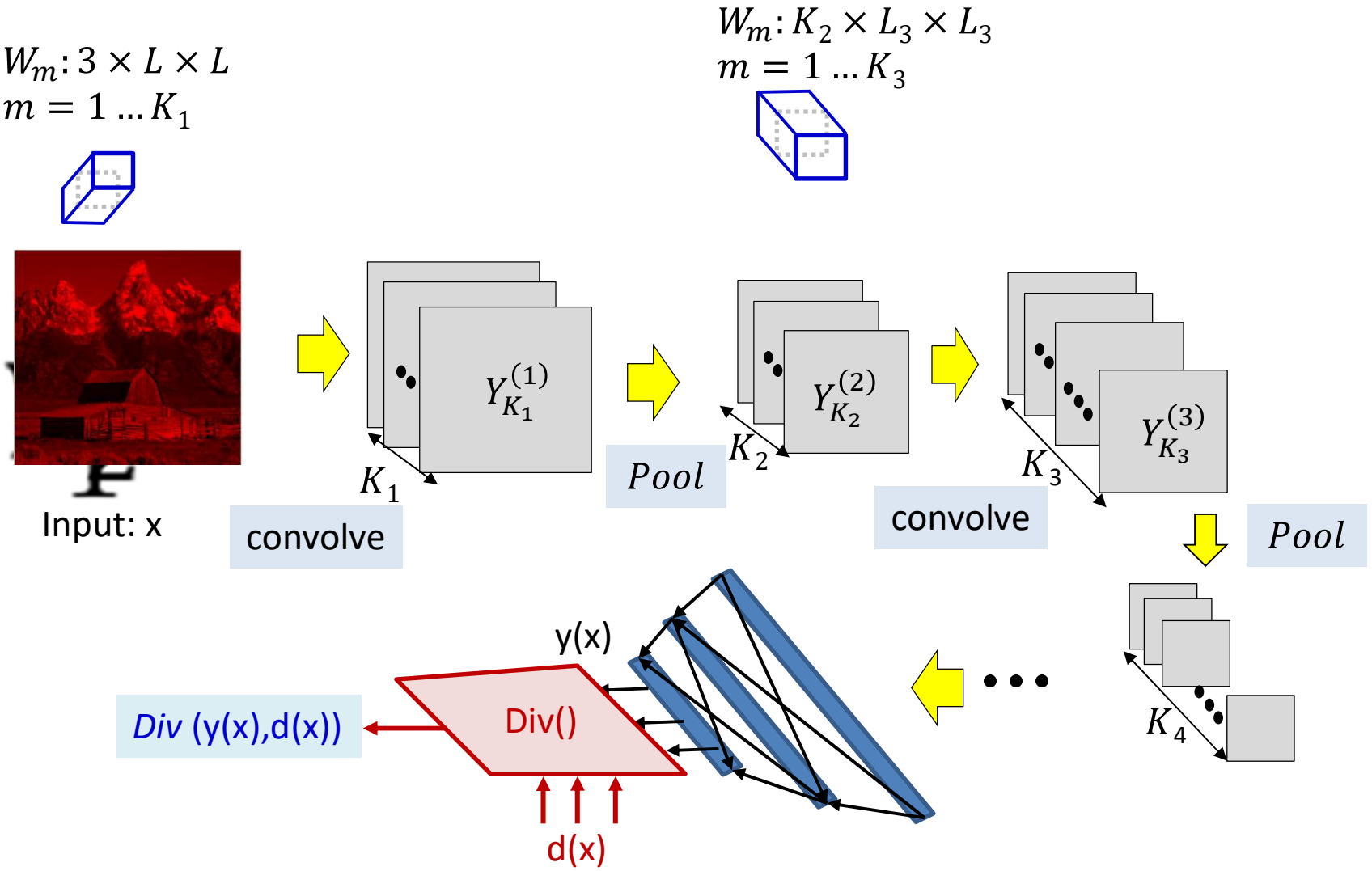
Recap: Learning the CNN

- 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 loss:**
  - Define a divergence between the desired output and true output of the network in response to any input
  - The loss aggregates the divergences of the training set

- **Network parameters are trained to minimize the loss**
  - Through variants of gradient descent
  - Gradients are computed through backpropagation

Defining the loss



- The loss for a single instance

Recap: Problem Setup

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

- The divergence on the ith instance is $div(Y_i, d_i)$
- The aggregate Loss

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

- Minimize $Loss$ w.r.t $\{W_m, b_m\}$
  - Using gradient descent

Recap: The derivative

Total training loss:

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

- Computing the derivative

Total derivative:

$$\frac{dLoss}{dw} = \frac{1}{T}\sum_{i} \frac{dDiv(Y_i, d_i)}{dw}$$

Recap: The derivative

Total training loss:

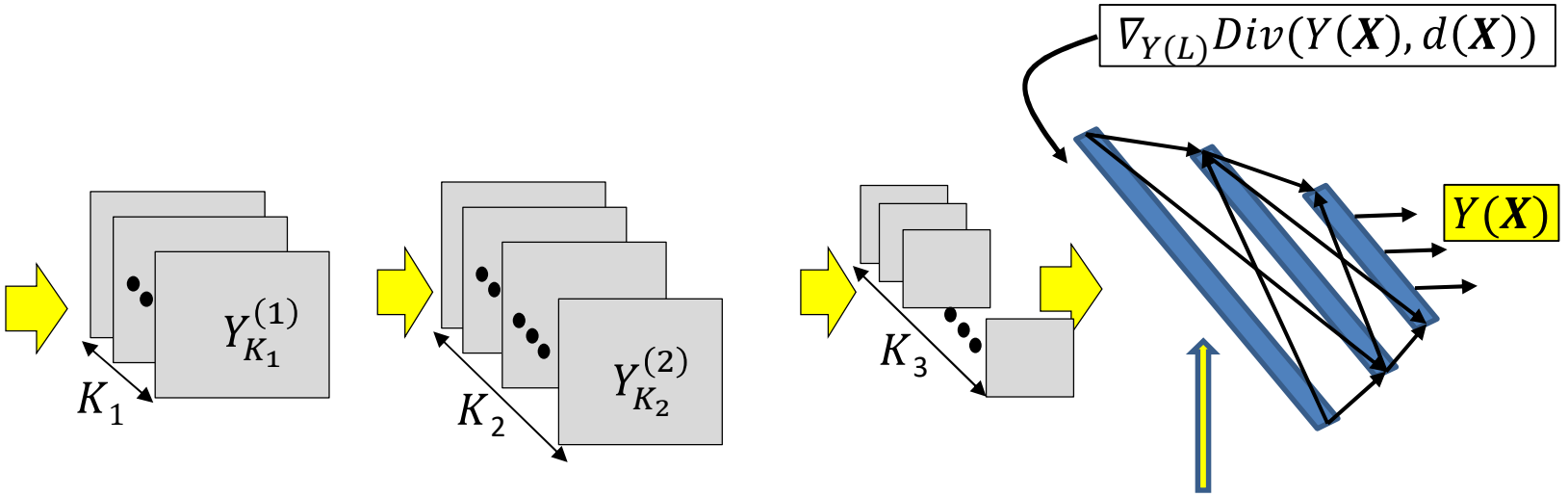
$$Loss = \frac{1}{T}\sum_{i} Div(Y_i, d_i)$$

- Computing the derivative

Total derivative:

$$\frac{dLoss}{dw} = \frac{1}{T}\sum_{i} \frac{dDiv(Y_i, d_i)}{dw}$$

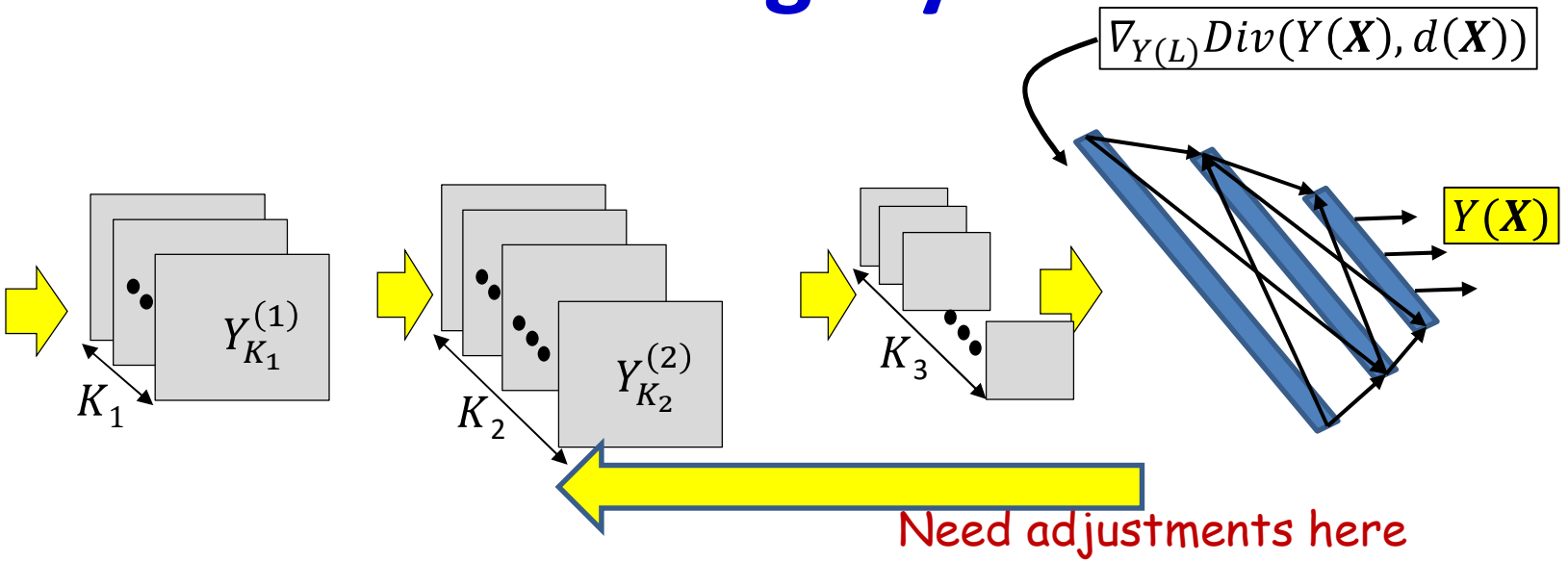
Backpropagation: Final flat layers



- For each training instance: First, a forward pass through the net
- Then the backpropagation of the derivative of the divergence

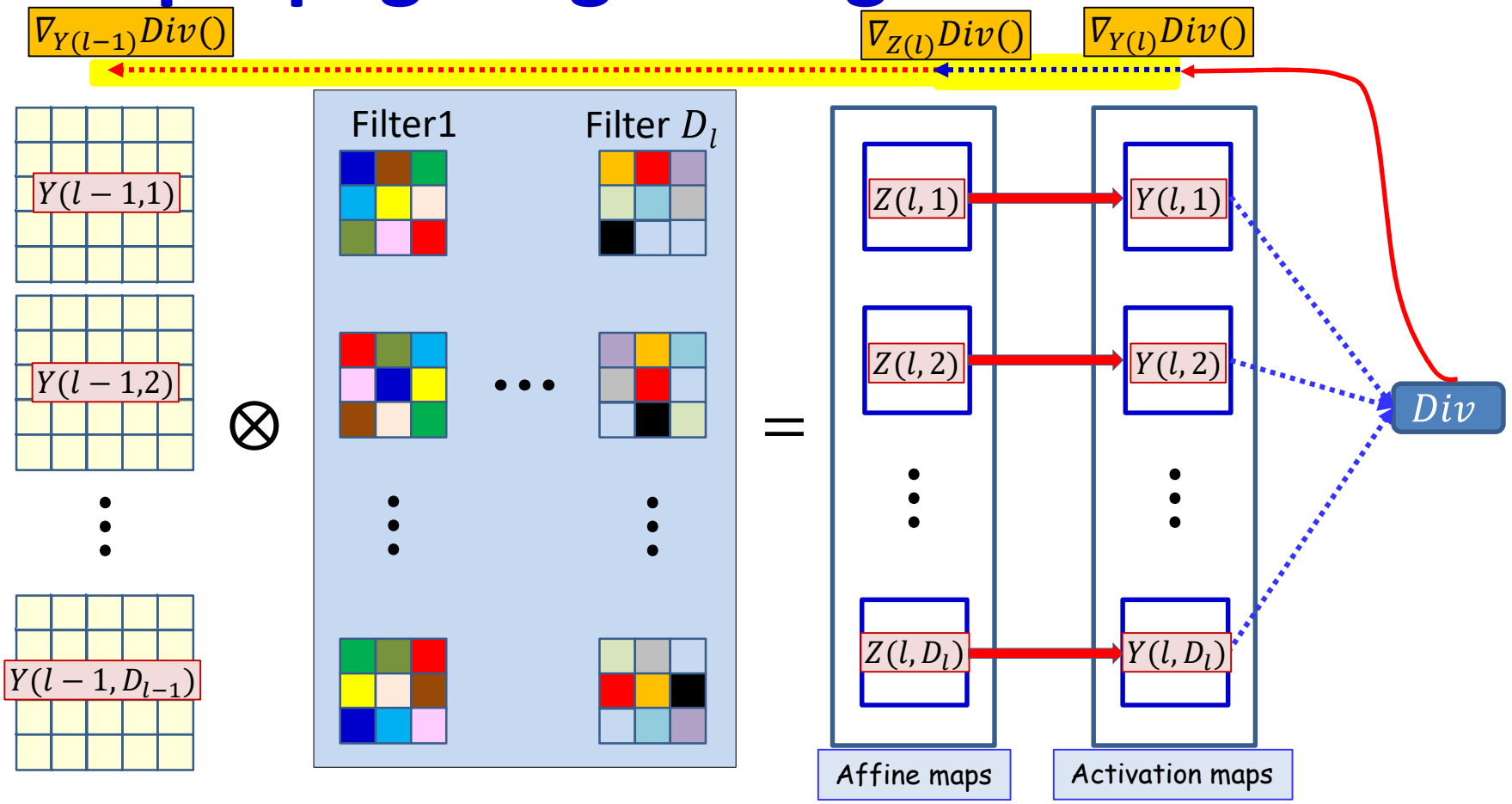
- Backpropagation continues in the usual manner until the computation of the derivative of the divergence w.r.t the inputs to the first "flat" layer
  - Important to recall: the first flat layer is only the "unrolling" of the maps from the final convolutional layer

Backpropagation: Convolutional and Pooling layers



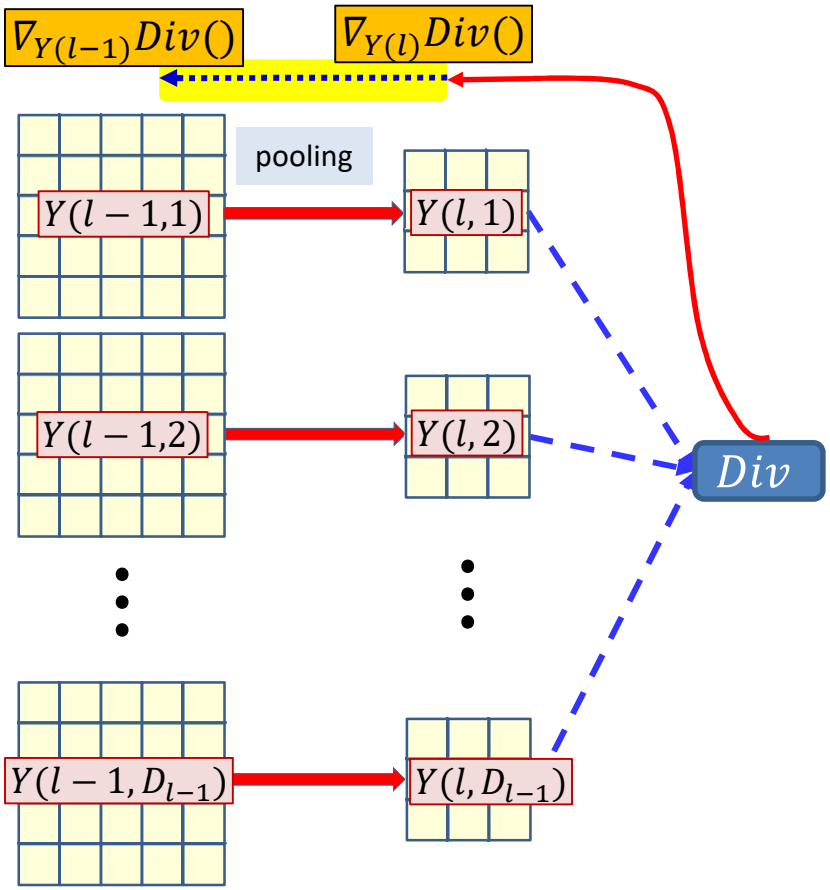
- Backpropagation from the flat MLP requires special consideration of
  - The shared computation in the convolution layers
  - The pooling layers (particularly maxpooling)

Backpropagating through the convolution



- **Convolution layers:**
- We already have the derivative w.r.t (all the elements of) activation map $Y(l,*)$
  - Having backpropagated it from the divergence
- We must backpropagate it through the activation to compute the derivative w.r.t. $Z(l,*)$ and further back to compute the derivative w.r.t the filters and $Y(l-1,*)$

Backprop: Pooling and D/S layer



- **Pooling and downsampling layers:**
- We already have the derivative w.r.t $Y(l,*)$
  - Having backpropagated it from the divergence
- We must compute the derivative w.r.t $Y(l-1,*)$

Backpropagation: Convolutional and Pooling layers

- **Assumption:** We already have the derivatives w.r.t. the elements of the maps output by the final convolutional (or pooling) layer
  - Obtained as a result of backpropagating through the flat MLP

- **Required:**
  - **For convolutional layers**:
    - Given derivative w.r.t. activation $Y(l)$ compute the derivatives w.r.t. the affine combination $Z(l)$ maps
    - From derivative w.r.t. $Z(l)$ compute the derivative w.r.t. $Y(l-1)$ and $w(l)$
  - **For pooling layers**:
    - How to compute the derivative w.r.t. $Y(l-1)$ given derivatives w.r.t. $Y(l)$

Backpropagation: Convolutional and Pooling layers

- **Assumption:** We already have the derivatives w.r.t. the elements of the maps output by the final convolutional (or pooling) layer
  - Obtained as a result of backpropagating through the flat MLP

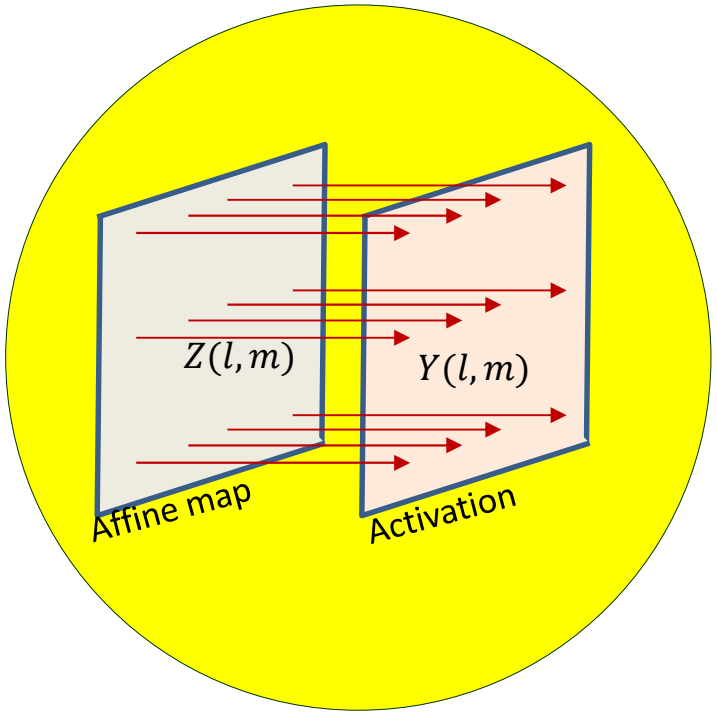
- **Required:**
  - **For convolutional layers**:
    - Given derivative w.r.t. activation $Y(l)$ compute the derivatives w.r.t. the affine combination $Z(l)$ maps
    - From derivative w.r.t. $Z(l)$ compute the derivative w.r.t. $Y(l-1)$ and $w(l)$
  - **For pooling layers**:
    - How to compute the derivative w.r.t. $Y(l-1)$ given derivatives w.r.t. $Y(l)$

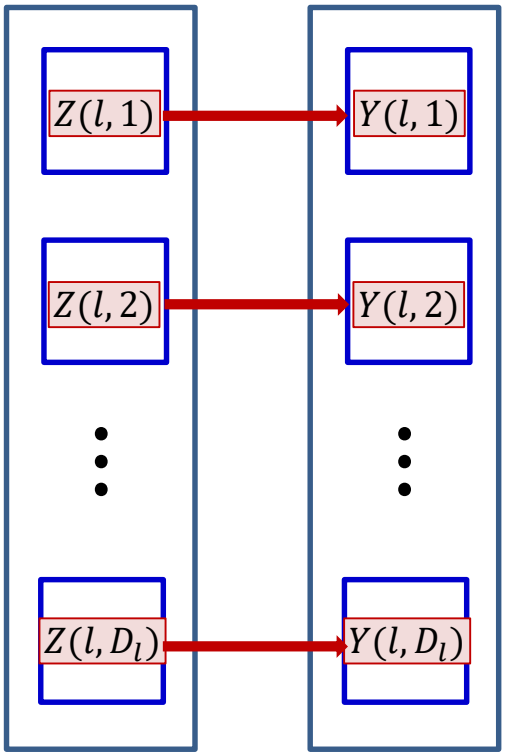
Backpropagation: Convolutional and Pooling layers

- **Assumption:** We already have the derivatives w.r.t. the elements of the maps output by the final convolutional (or pooling) layer
  - Obtained as a result of backpropagating through the flat MLP

- **Required:**
  - **For convolutional layers**:
    - Given derivative w.r.t. activation $Y(l)$ compute the derivatives w.r.t. the affine combination $Z(l)$ maps
    - From derivative w.r.t. $Z(l)$ compute the derivative w.r.t. $Y(l-1)$ and $w(l)$
  - **For pooling layers**:
    - How to compute the derivative w.r.t. $Y(l-1)$ given derivatives w.r.t. $Y(l)$

Backpropagating through the activation



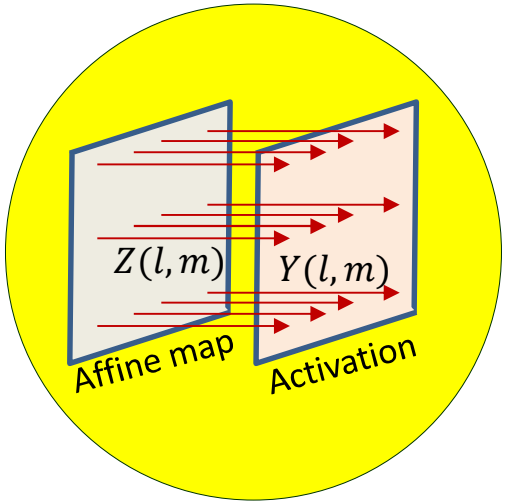


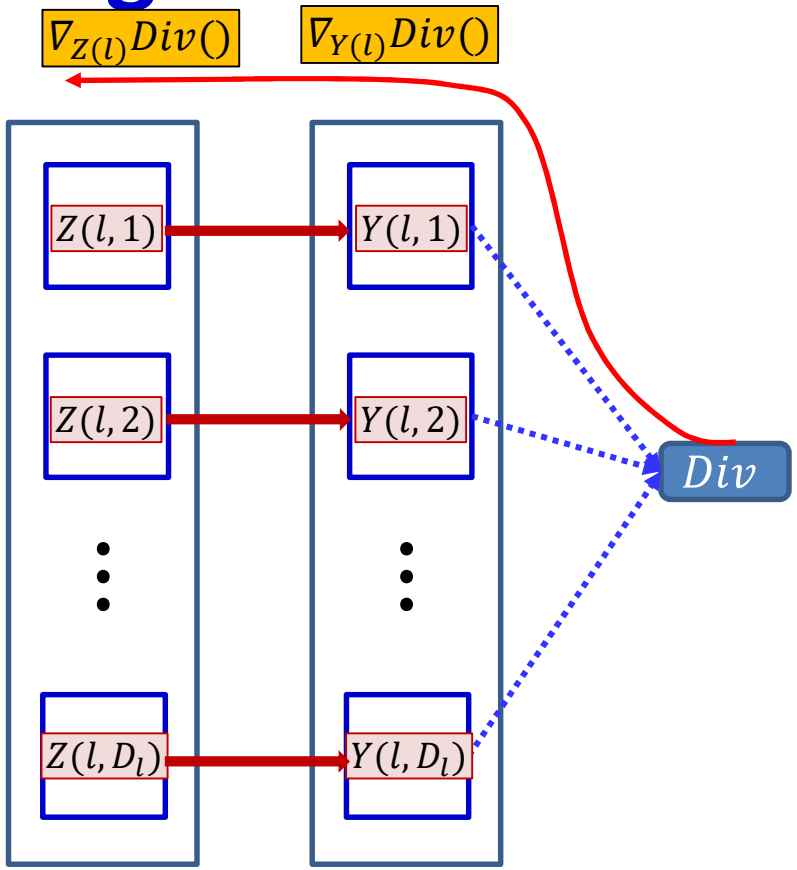
- **Forward computation:** The activation maps are obtained by point-wise application of the activation function to the affine maps

$$y(l,m,x,y) = f(z(l,m,x,y))$$

  - The affine map entries $z(l,m,x,y)$ have already been computed via convolutions over the previous layer

Backpropagating through the activation





$$y(l,m,x,y) = f(z(l,m,x,y))$$

$$\frac{dDiv}{dz(l,m,x,y)} = \frac{dDiv}{d\,y(l,m,x,y)} f'(z(l,m,x,y))$$

- **Backward computation:** For every map $Y(l,m)$ for every position (x,y), we already have the derivative of the divergence w.r.t. $y(l,m,x,y)$
  - Obtained via backpropagation
- We obtain the derivatives of the divergence w.r.t. $z(l,m,x,y)$ using the chain rule:

$$\frac{dDiv}{dz(l,m,x,y)} = \frac{dDiv}{d\,y(l,m,x,y)} f'(z(l,m,x,y))$$

  - Simple component-wise computation

Backpropagation: Convolutional and Pooling layers

- **Assumption:** We already have the derivatives w.r.t. the elements of the maps output by the final convolutional (or pooling) layer
  - Obtained as a result of backpropagating through the flat MLP

- **Required:**
  - **For convolutional layers**:
    - ✔ Given derivative w.r.t. activation $Y(l)$ compute the derivatives w.r.t. the affine combination $Z(l)$ maps
    - From derivative w.r.t. $Z(l)$ compute the derivative w.r.t. $Y(l-1)$ and $w(l)$
  - **For pooling layers**:
    - How to compute the derivative w.r.t. $Y(l-1)$ given derivatives w.r.t. $Y(l)$

Backpropagating through affine map

- Forward affine computation:
  - Compute affine maps $\mathrm{z}(l, n, x, y)$ from previous layer maps $y(l-1, m, x, y)$ and filters $w_l(m, n, x, y)$

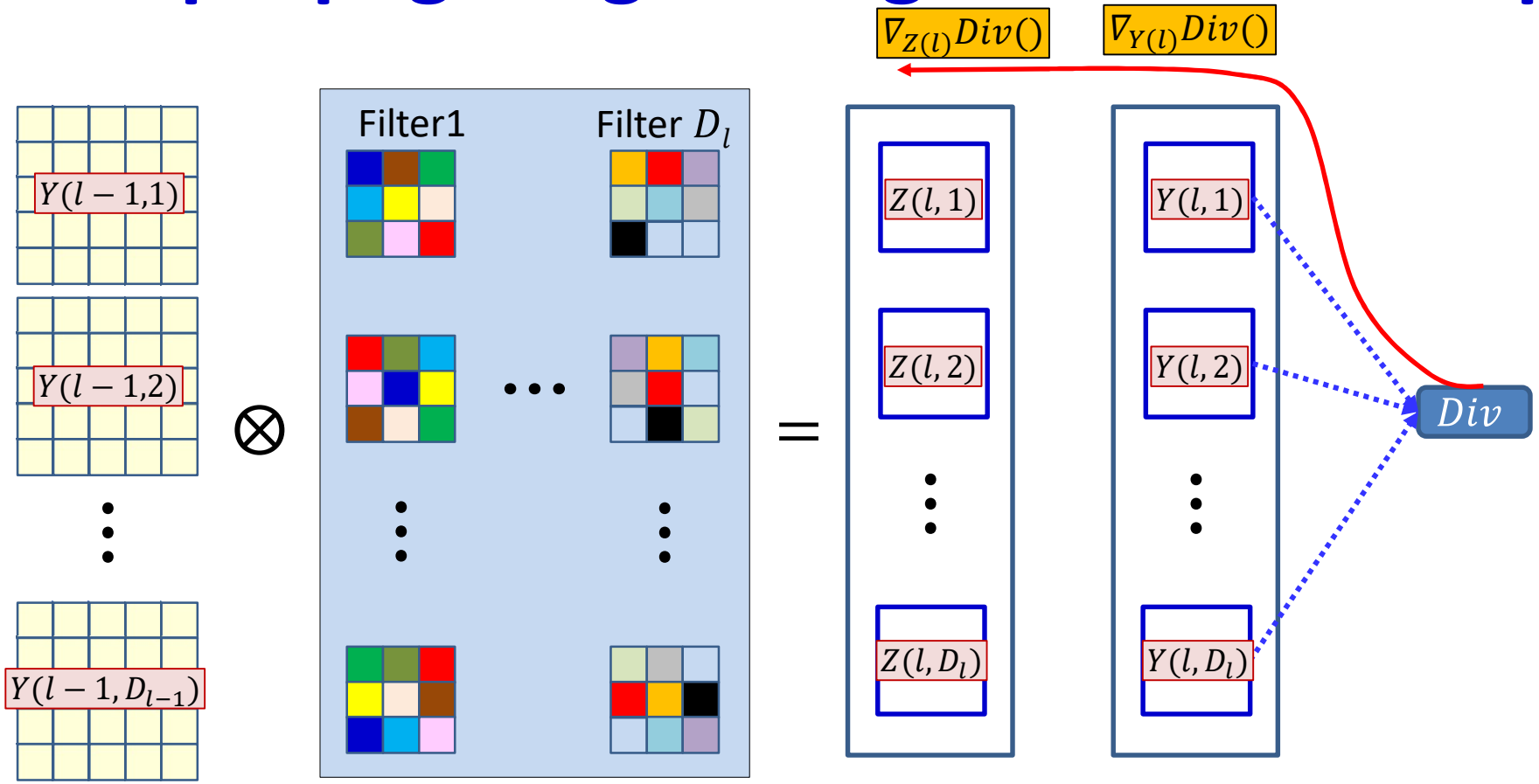
- Backpropagation: Given $\frac{dDiv}{dz(l,n,x,y)}$
  - Compute derivative w.r.t. $y(l-1, m, x, y)$
  - Compute derivative w.r.t. $w_l(m, n, x, y)$

Backpropagating through affine map

- Forward affine computation:
  - Compute affine maps $z(l,n,x,y)$ from previous layer maps $y(l-1,m,x,y)$ and filters $w_l(m,n,x,y)$

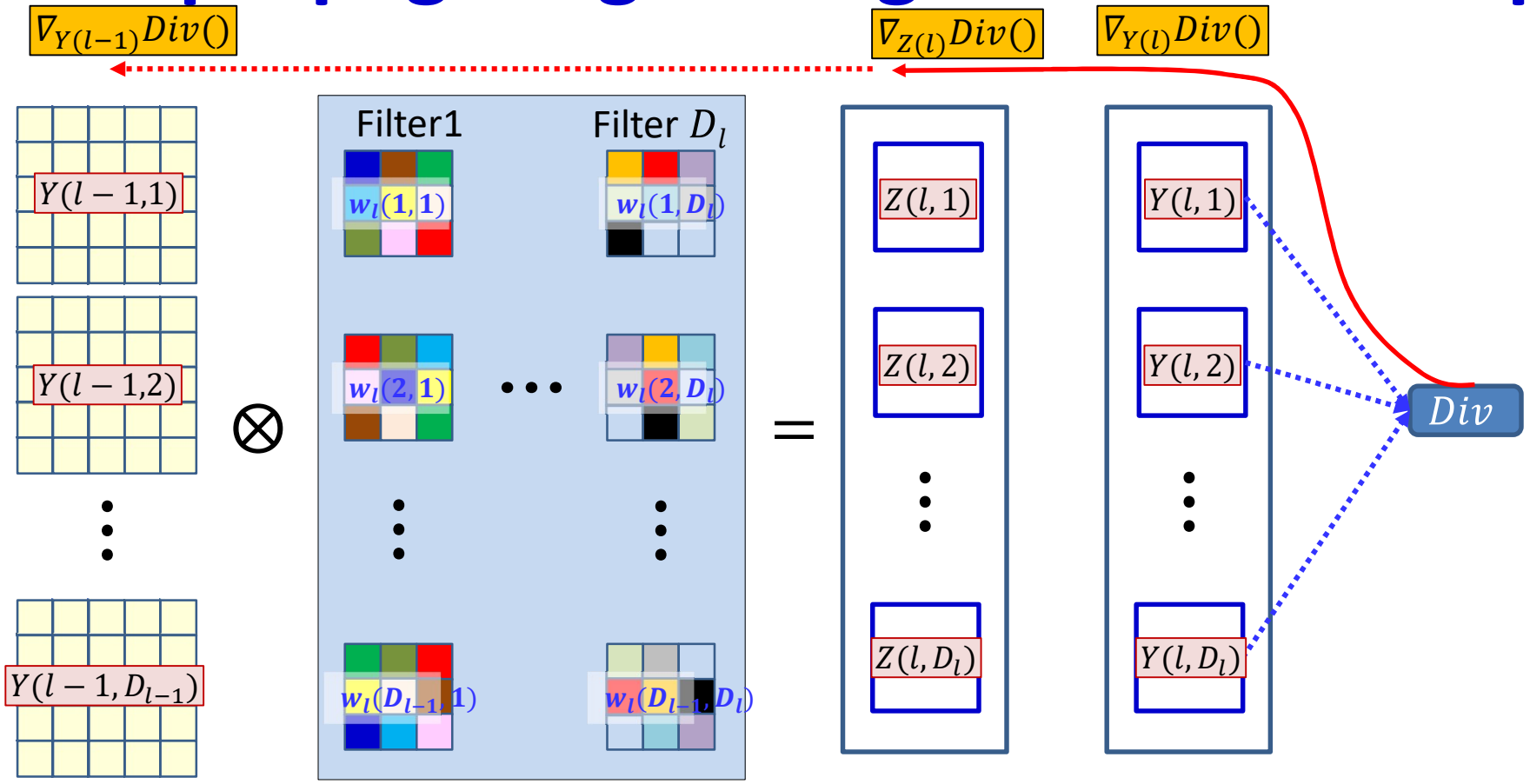
- Backpropagation: Given $\frac{dDiv}{dz(l,n,x,y)}$
  - Compute derivative w.r.t. $y(l-1,m,x,y)$
  - Compute derivative w.r.t. $w_l(m,n,x,y)$

Backpropagating through the affine map



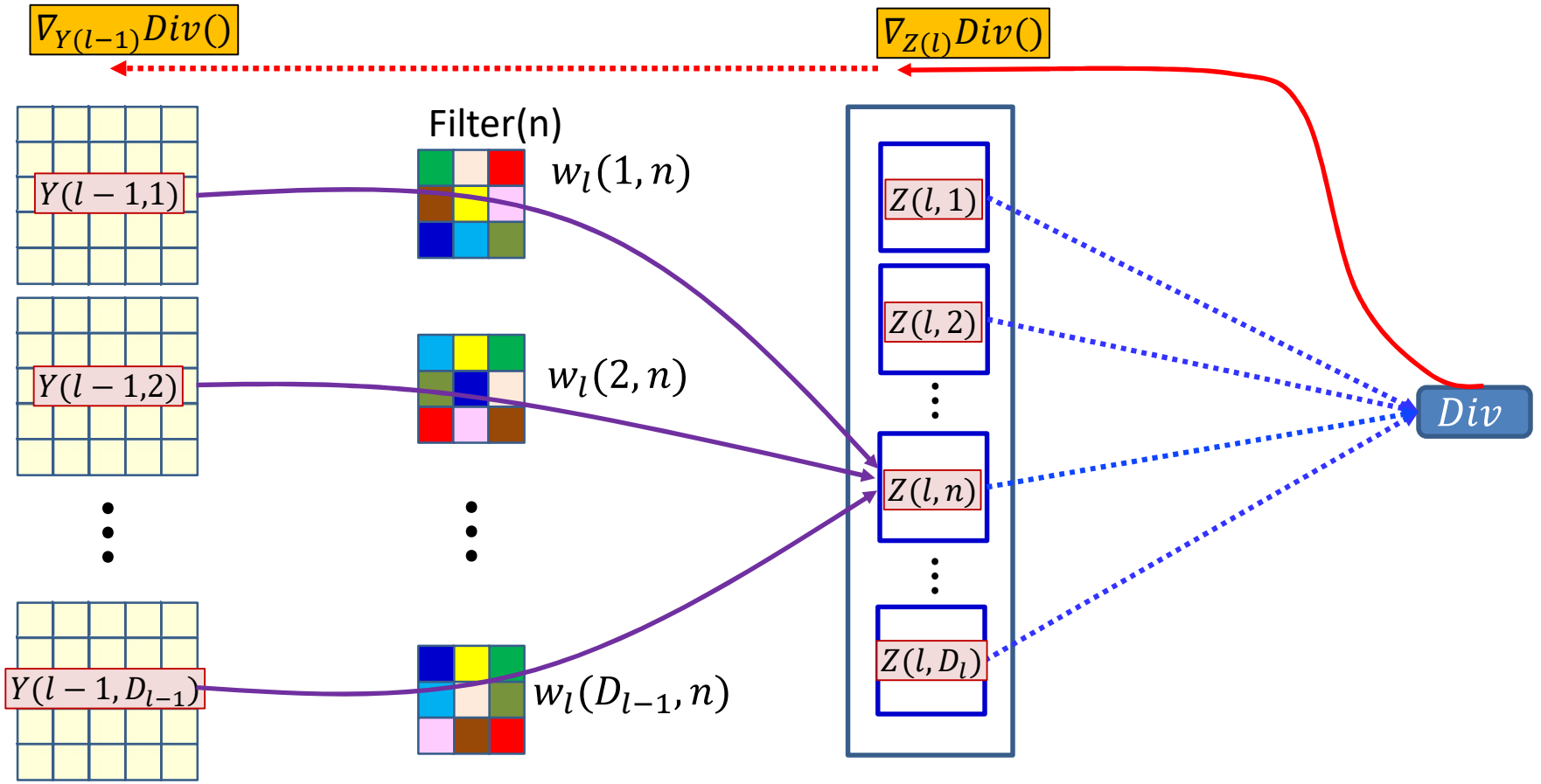
- We already have the derivative w.r.t $Z(l,*)$
  - Having backpropagated it past $Y(l,*)$

Backpropagating through the affine map



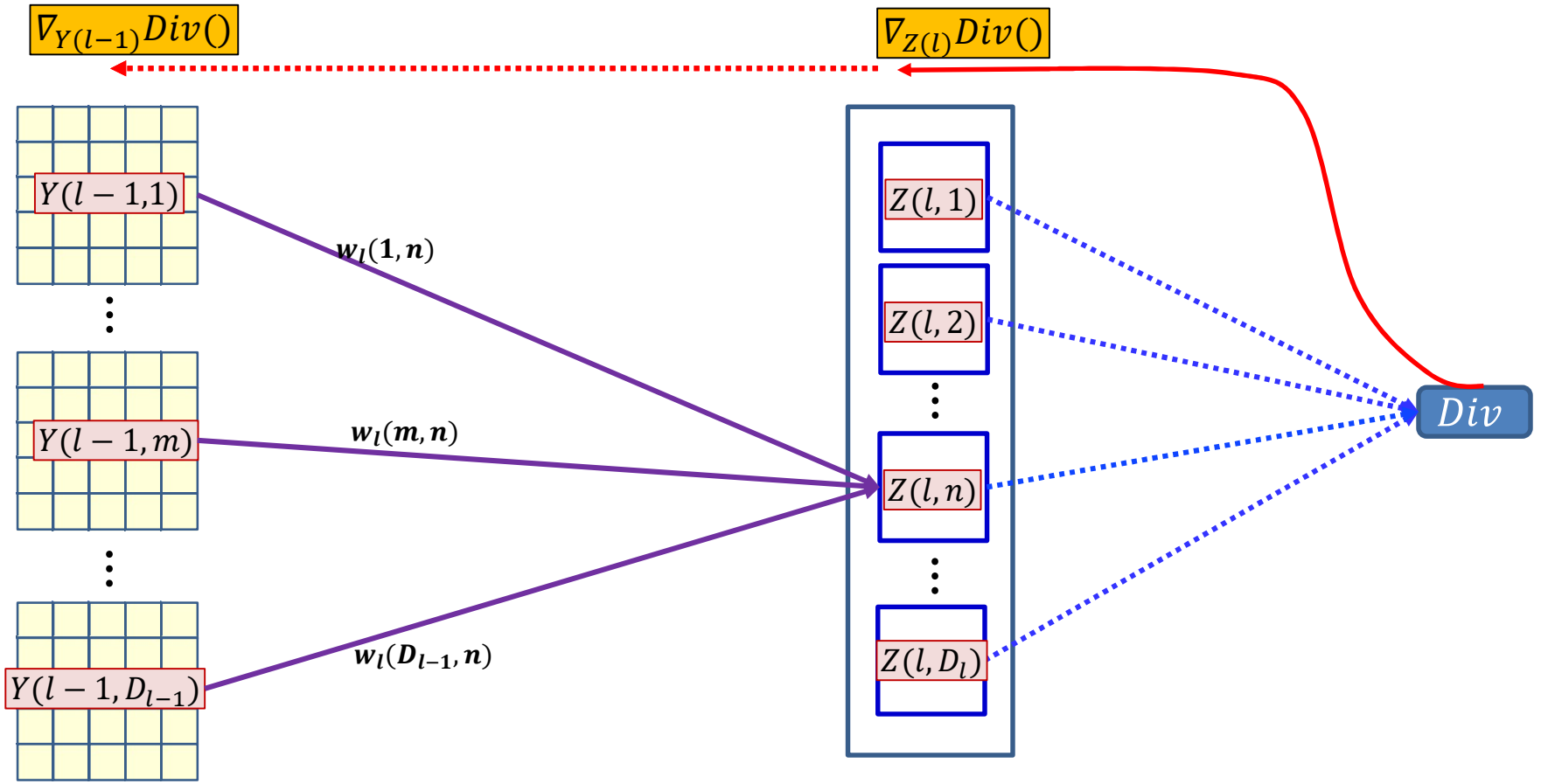
- We already have the derivative w.r.t $Z(l,*)$
  - Having backpropagated it past $Y(l,*)$
- We must compute the derivative w.r.t $Y(l-1,*)$

Dependency between Z(l,n) and Y(l-1,*)



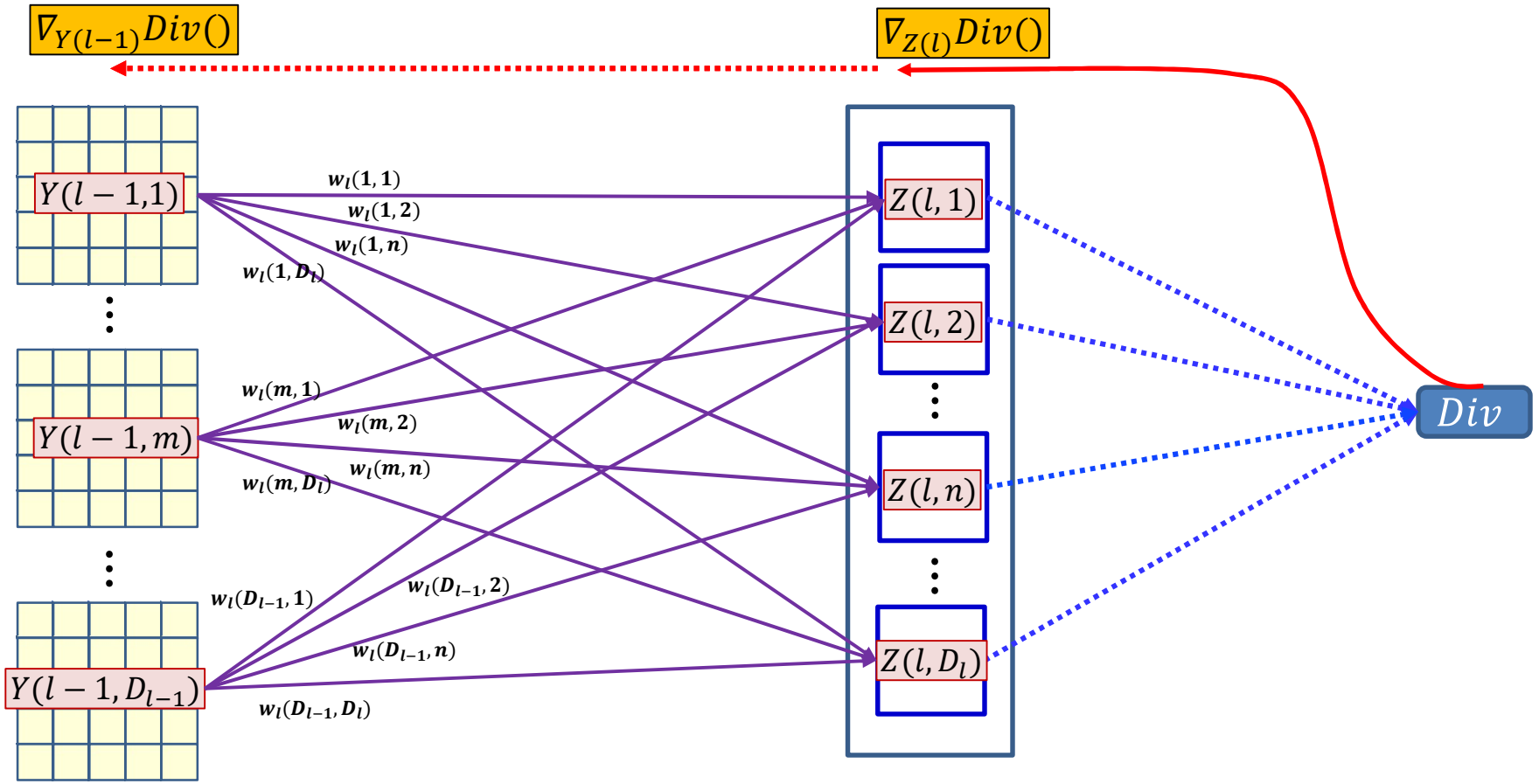
- Each $Y(l-1,m)$ map influences $Z(l,n)$ through the mth "plane" of the nth filter $w_l(m,n)$

Dependency between Z(l,n) and Y(l-1,*)



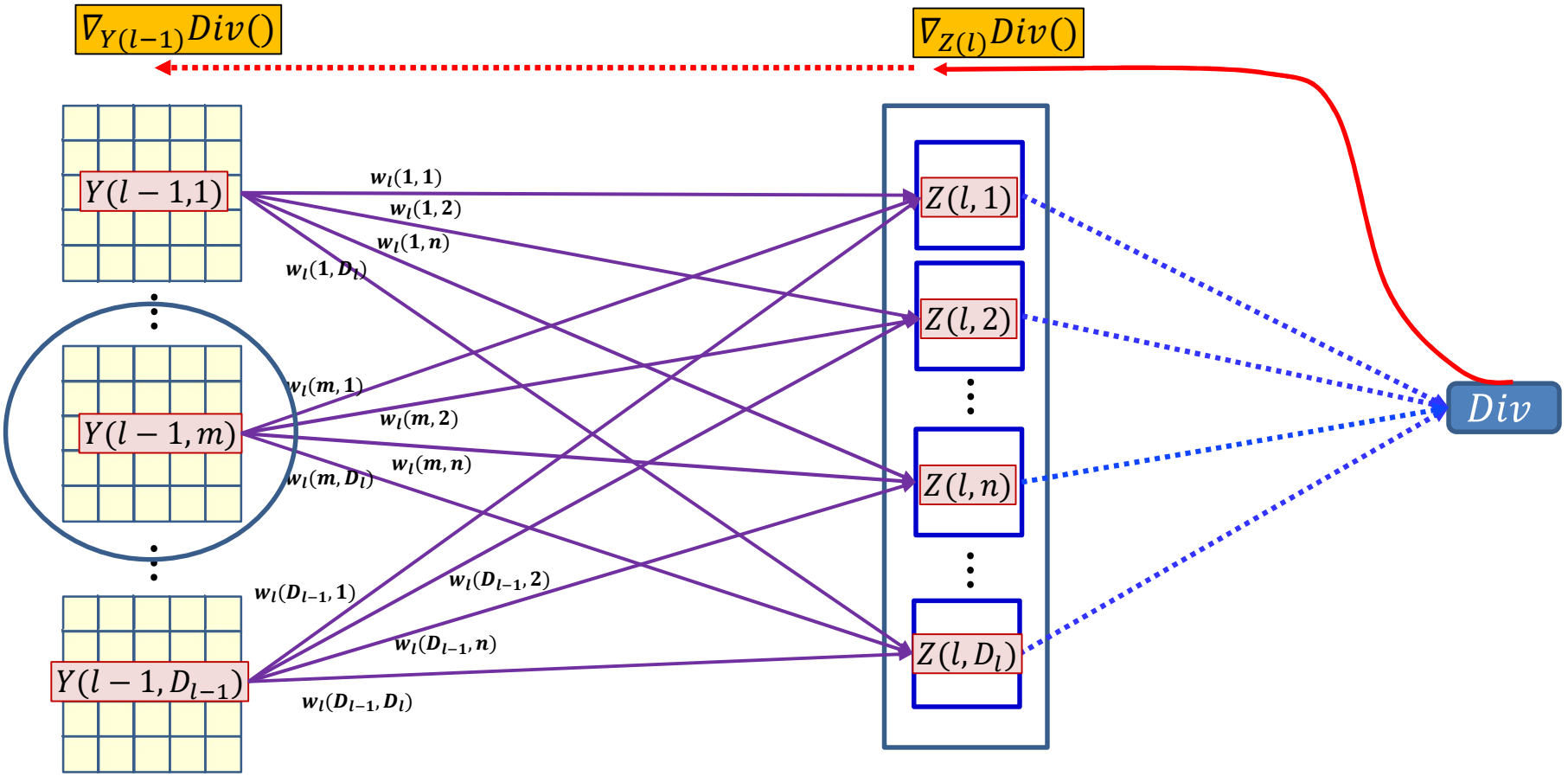
- Each $Y(l-1,m)$ map influences $Z(l,n)$ through the mth "plane" of the nth filter $w_l(m,n)$

Dependency between Z(l,*) and Y(l-1,*)



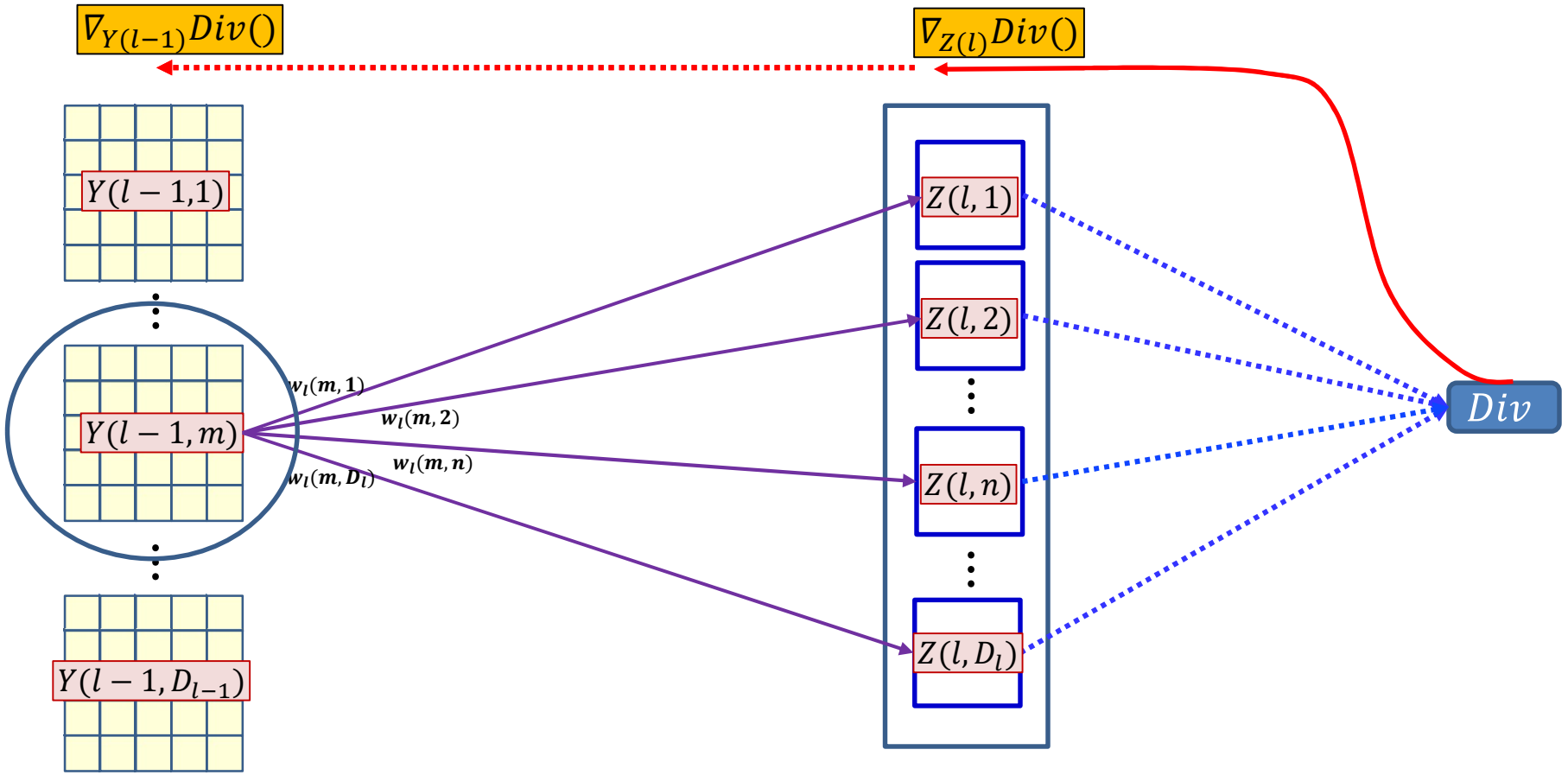
- Each $Y(l-1,m)$ map influences $Z(l,n)$ through the mth "plane" of the nth filter $w_l(m,n)$

Dependency between Z(l,*) and Y(l-1,*)



- Each $Y(l-1,m)$ map influences $Z(l,n)$ through the mth "plane" of the nth filter $w_l(m,n)$

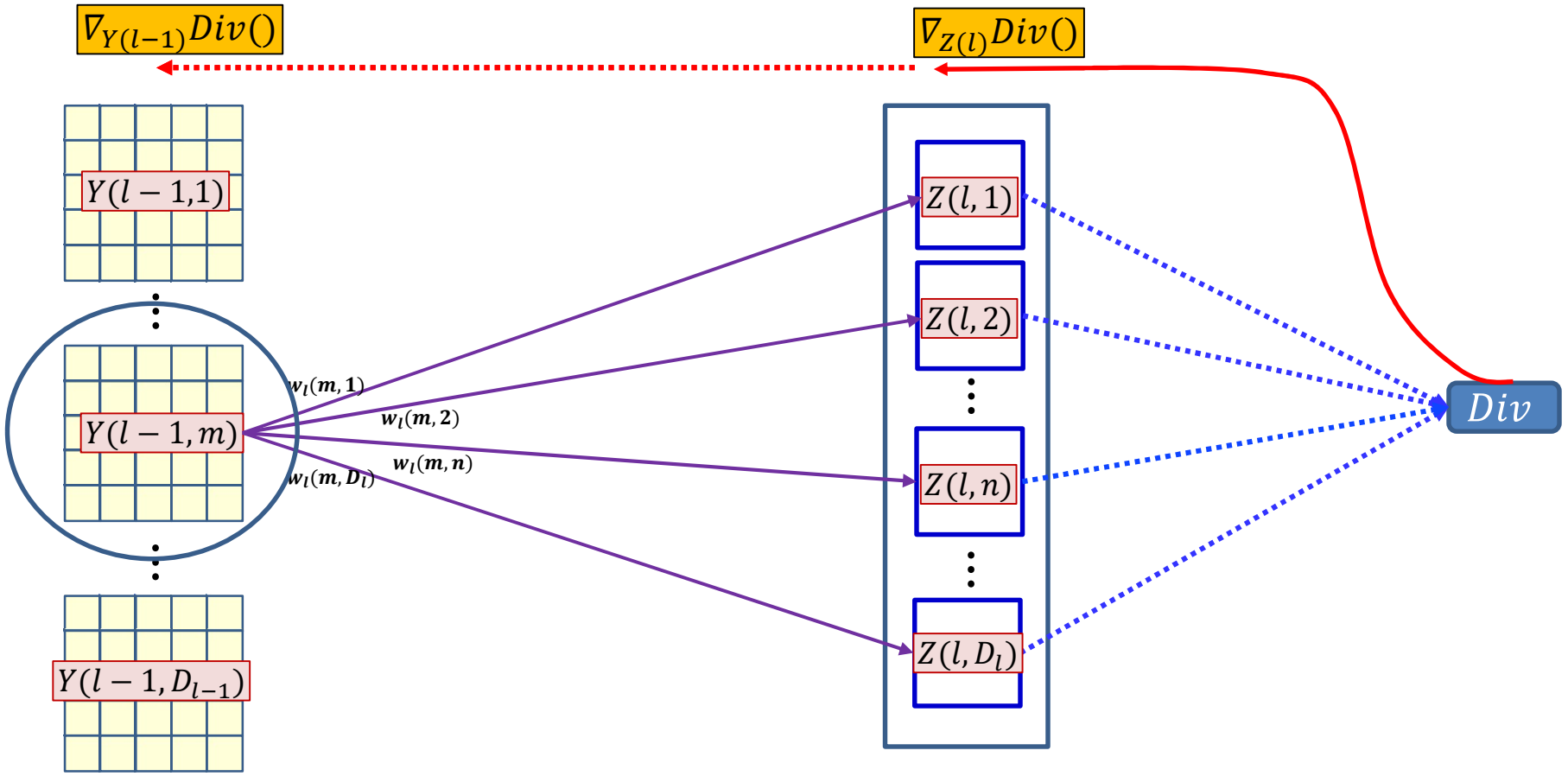
Dependency diagram for a single map



- Each $Y(l-1,m)$ map influences $Z(l,n)$ through the mth "plane" of the nth filter $w_l(m,n)$

- $Y(l-1,m,*,*)$ influences the divergence through all $Z(l,n,*,*)$ maps

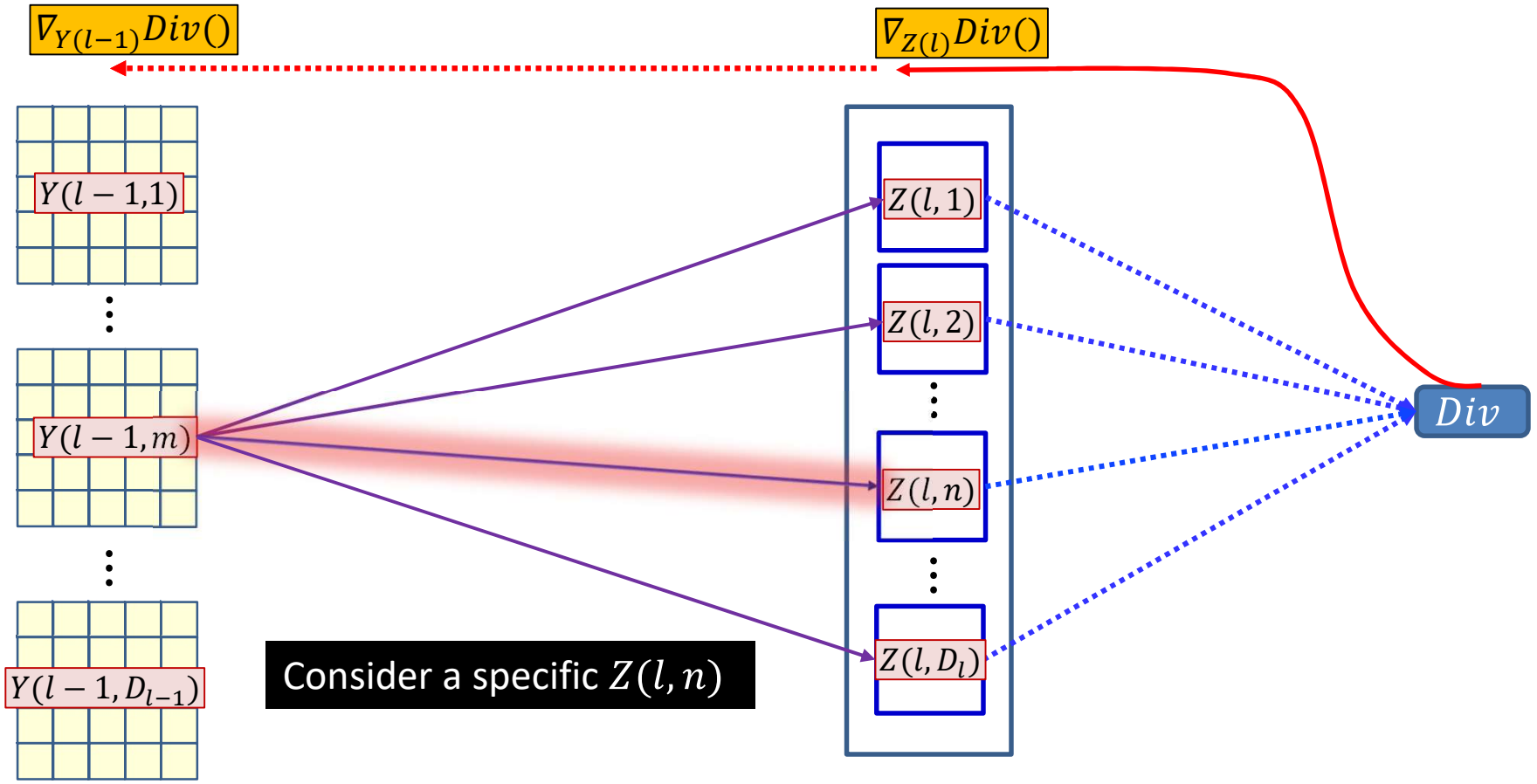
Dependency diagram for a single map



$$\nabla_{Y(l-1,m)}Div(.) = \sum_n \nabla_{Z(l,n)}Div(.)\,\nabla_{Y(l-1,m)}Z(l,n)$$

- Need to compute $\nabla_{Y(l-1,m)}Z(l,n)$, the derivative of $Z(l,n)$ w.r.t. $Y(l-1,m)$ to complete the computation of the formula

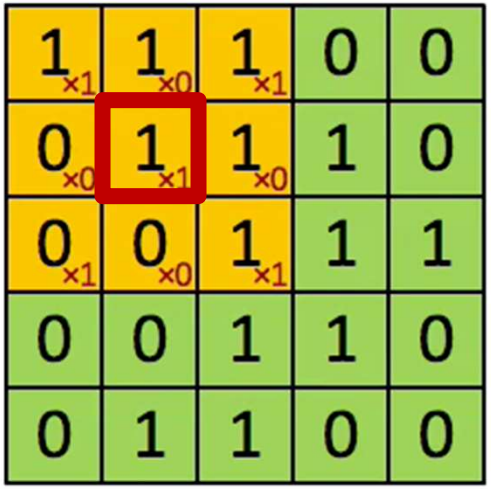
Dependency diagram for a single map



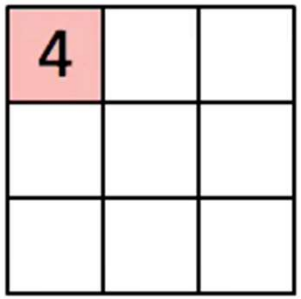
$$\nabla_{Y(l-1,m)}Div(.) = \sum_n \nabla_{Z(l,n)}Div(.)\,\nabla_{Y(l-1,m)}Z(l,n)$$

- Need to compute $\nabla_{Y(l-1,m)}Z(l,n)$, the derivative of $Z(l,n)$ w.r.t. $Y(l-1,m)$ to complete the computation of the formula

BP: Convolutional layer



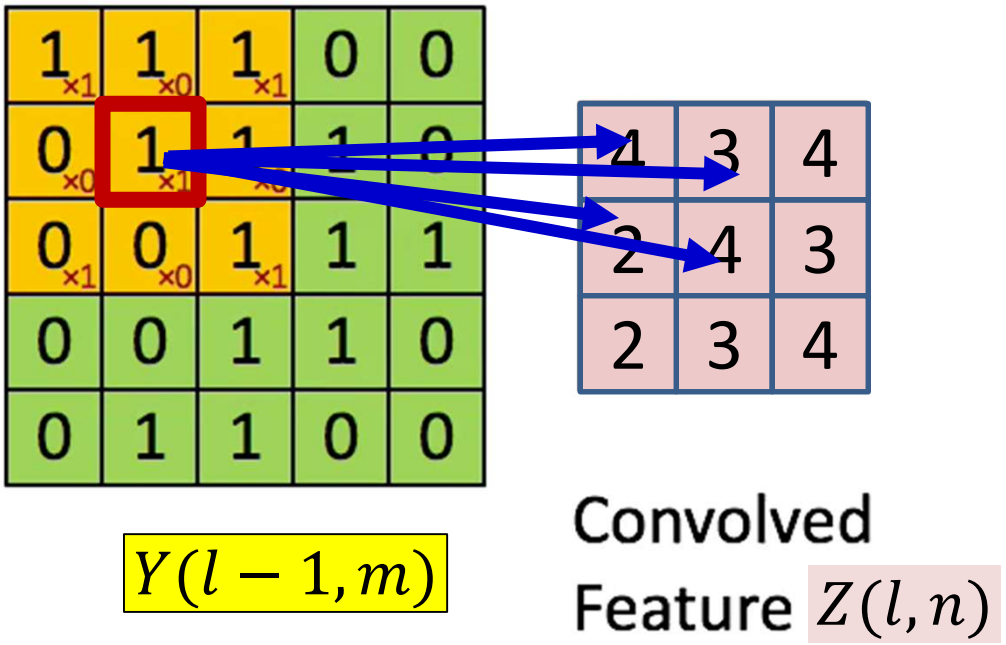
$Y(l-1,m)$



Convolved Feature $Z(l,n)$

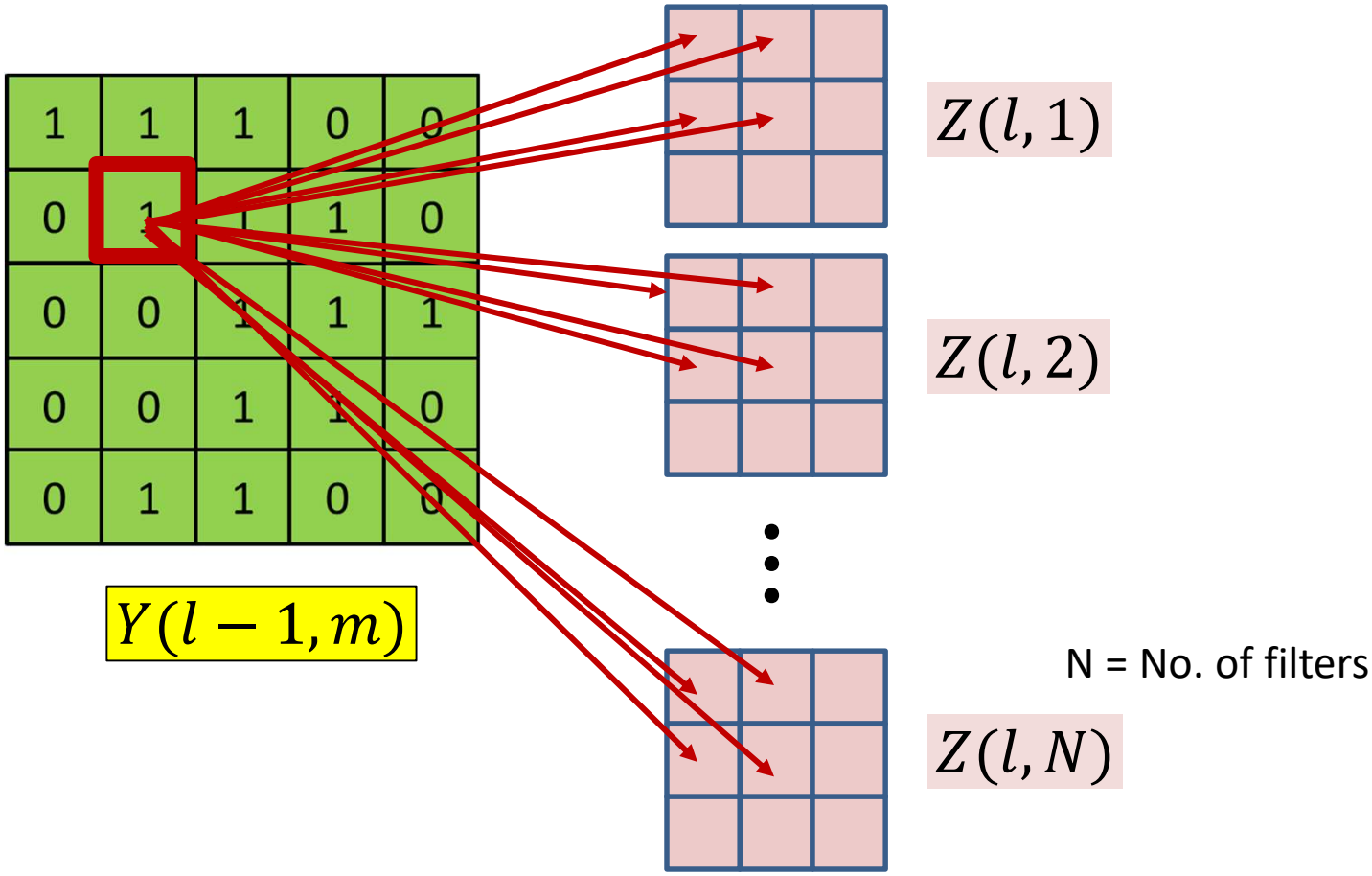
- Each $Y(l-1,m,x,y)$ affects several $z(l,n,x',y')$ terms

BP: Convolutional layer



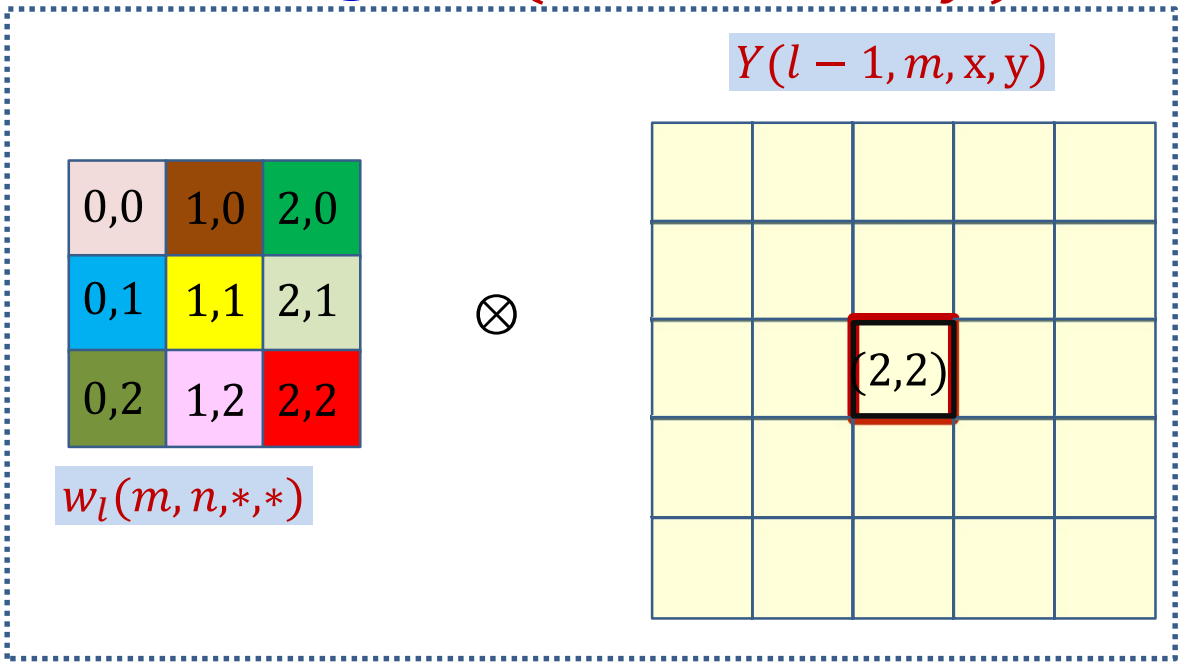
- Each $Y(l-1,m,x,y)$ affects several $z(l,n,x',y')$ terms

BP: Convolutional layer



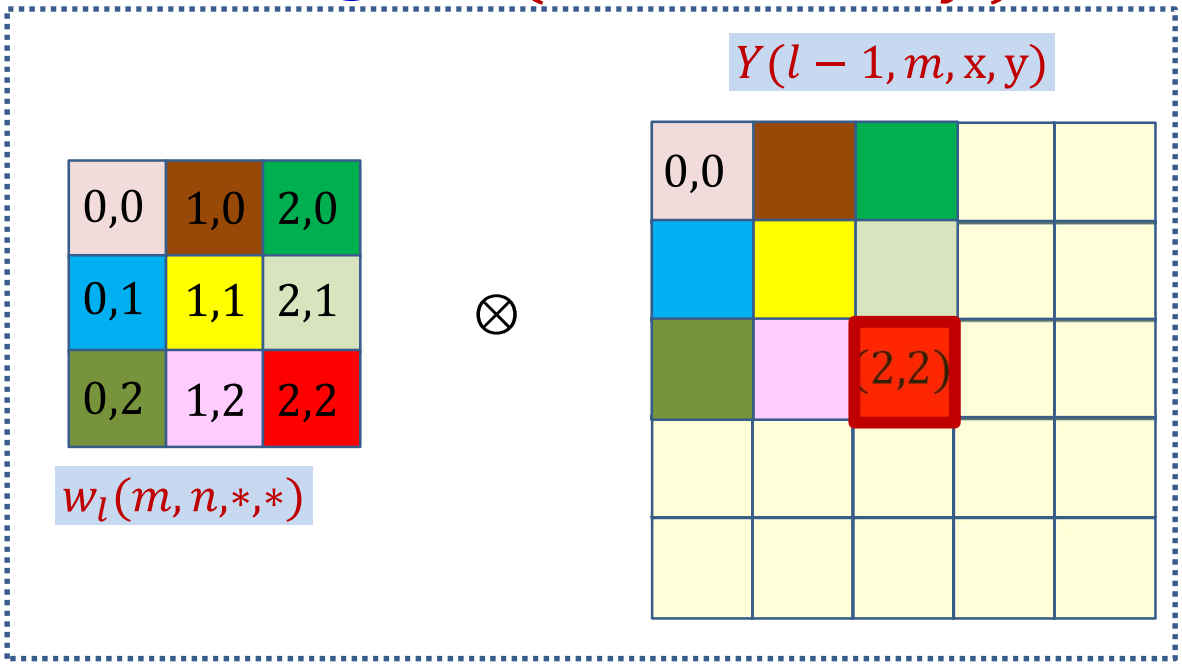
- Each $Y(l-1,m,x,y)$ affects several $z(l,n,x',y')$ terms
  - Affects terms in *all* l th layer Z maps
  - But how?

How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$



- Compute how *each* x, y in Y influences various locations of z
  - We will have to reverse the direction of influence to compute the derivative w.r.t that x, y component of Y

How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$

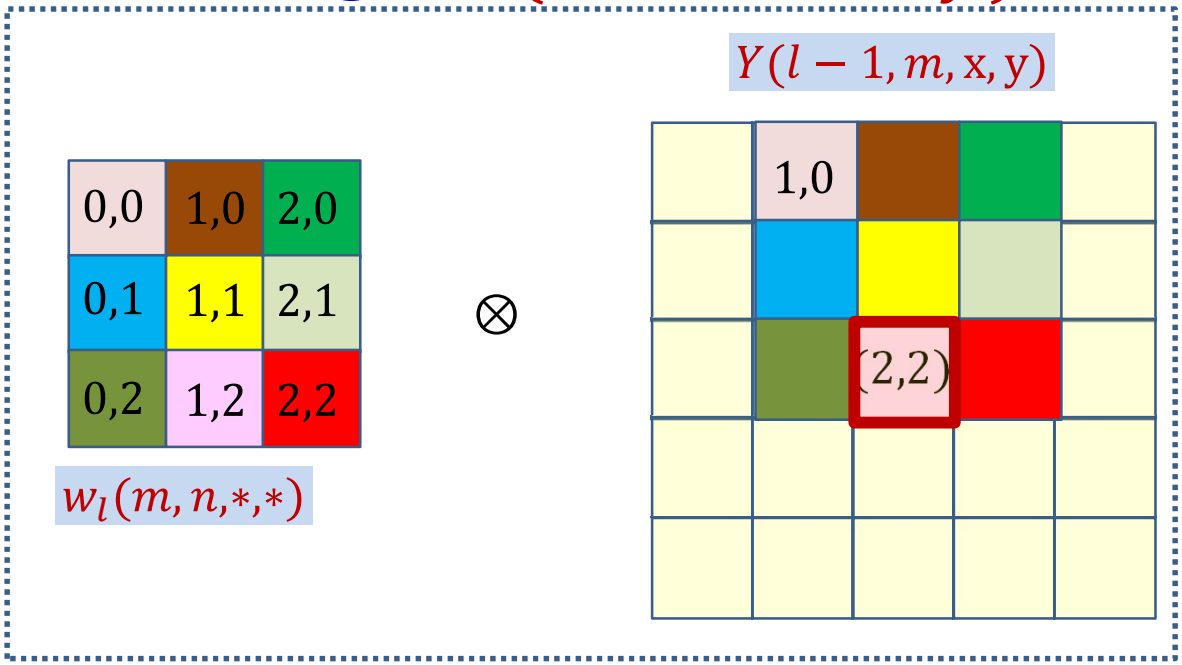


$$z(l,n,0,0) \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,2,2)$$

- **Note:** The coordinates of $z(l,n)$ and $w_l(m,n)$ sum to the coordinates of $Y(l-1,m)$

$$z(l,n,x',y') \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,2-x',2-y')$$

How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$

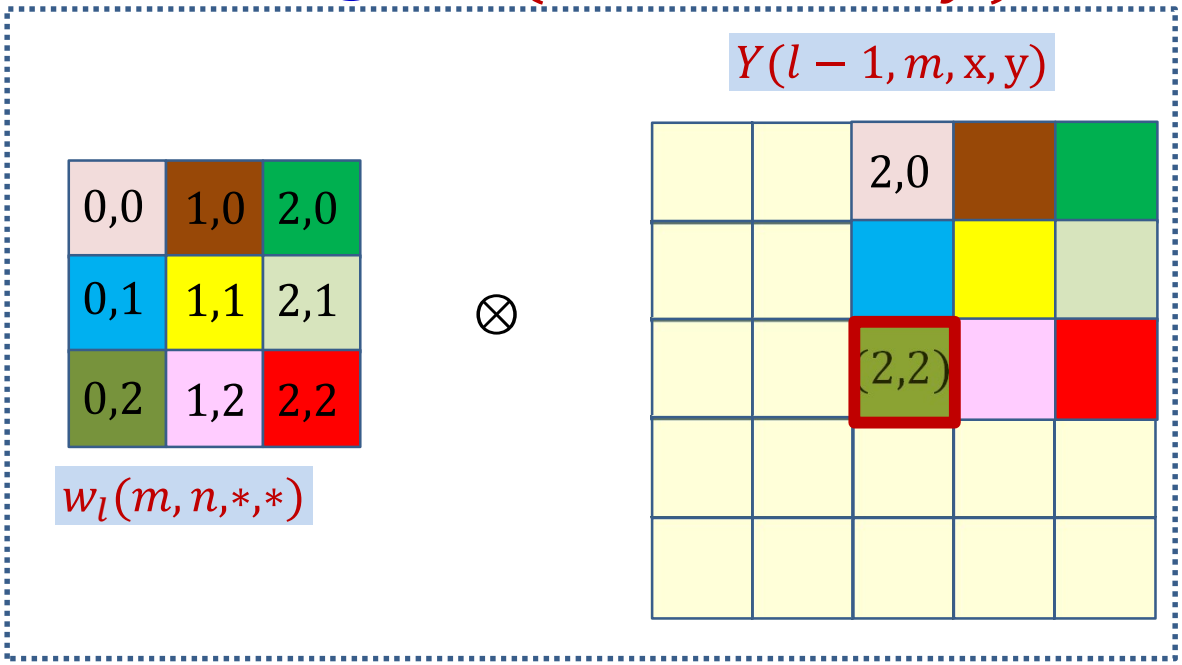


$$z(l,n,1,0) \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,1,2)$$

- **Note:** The coordinates of $z(l,n)$ and $w_l(m,n)$ sum to the coordinates of $Y(l-1,m)$

$$z(l,n,x',y') \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,2-x',2-y')$$

How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$

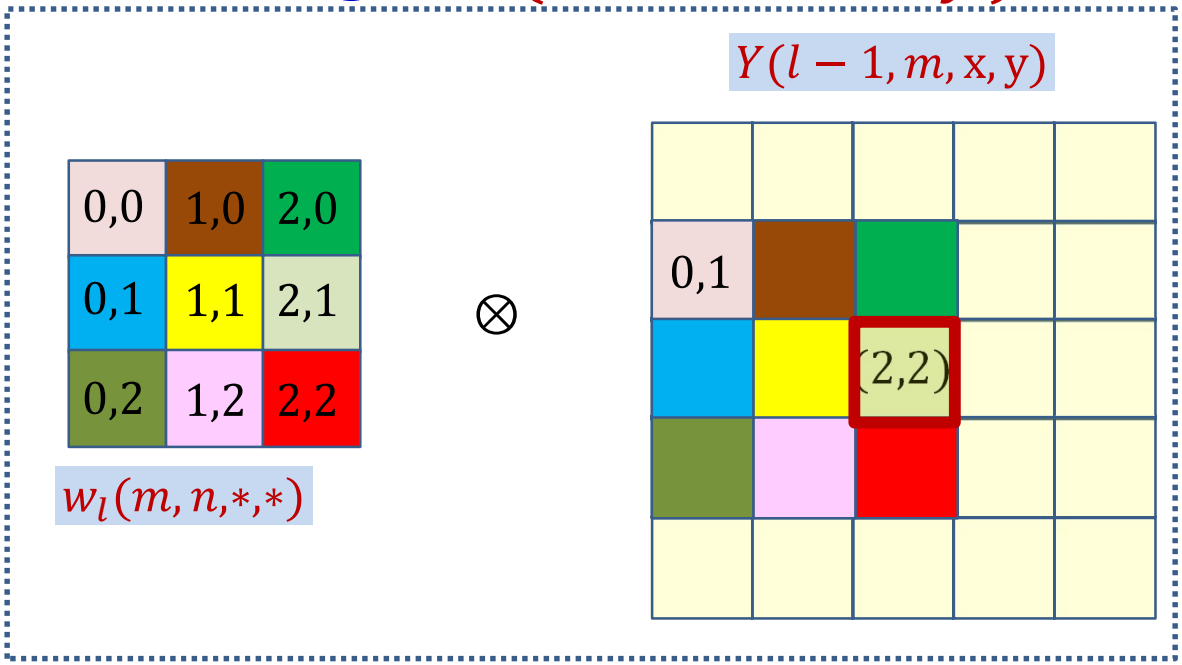


$$z(l,n,2,0) \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,0,2)$$

- **Note:** The coordinates of $z(l,n)$ and $w_l(m,n)$ sum to the coordinates of $Y(l-1,m)$

$$z(l,n,x',y') \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,2-x',2-y')$$

How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$

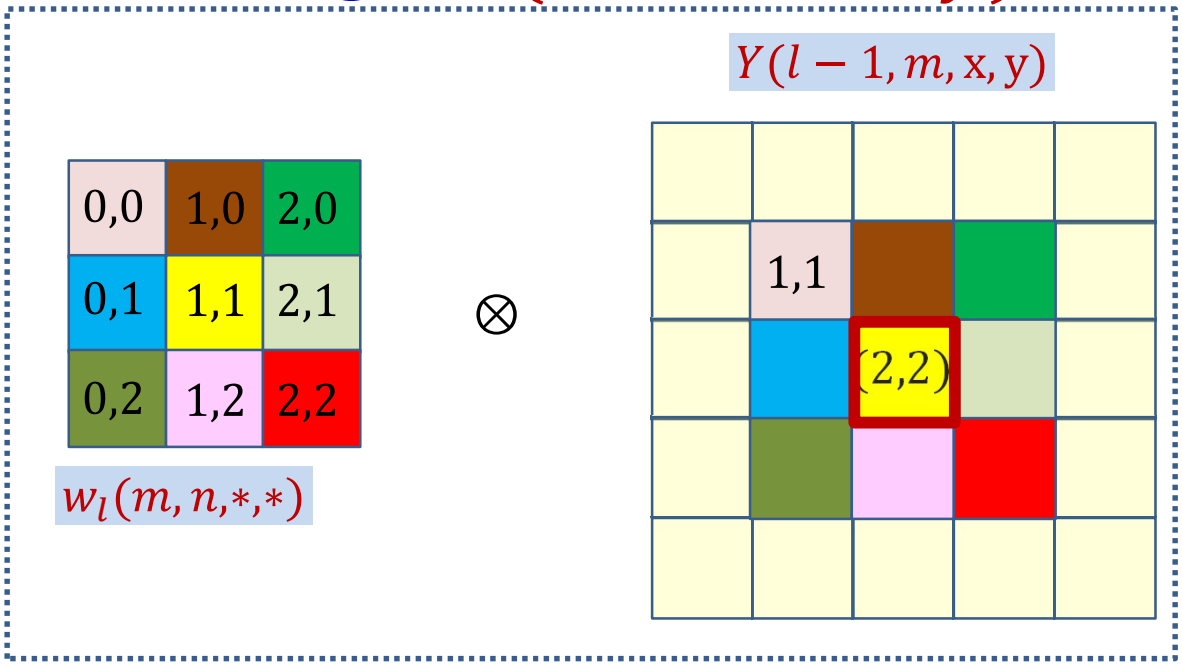


$$z(l,n,0,1) \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,2,1)$$

- **Note:** The coordinates of $z(l,n)$ and $w_l(m,n)$ sum to the coordinates of $Y(l-1,m)$

$$z(l,n,x',y') \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,2-x',2-y')$$

How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$

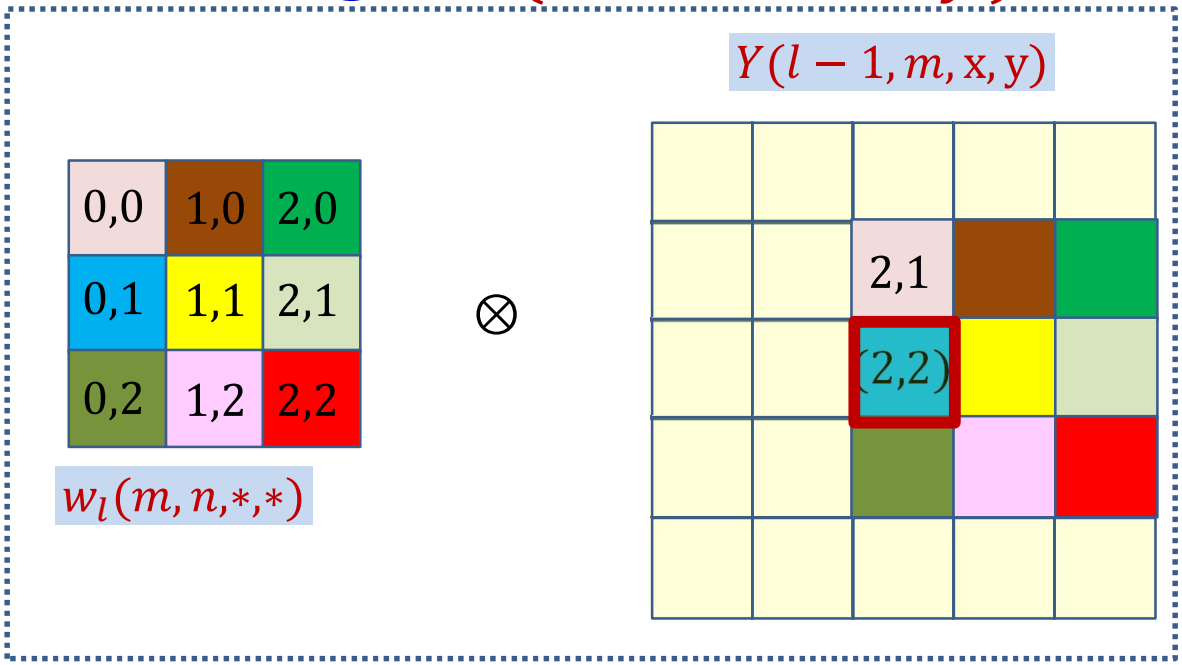


$$z(l,n,1,1) \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,1,1)$$

- **Note:** The coordinates of $z(l,n)$ and $w_l(m,n)$ sum to the coordinates of $Y(l-1,m)$

$$z(l,n,x',y') \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,2-x',2-y')$$

How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$

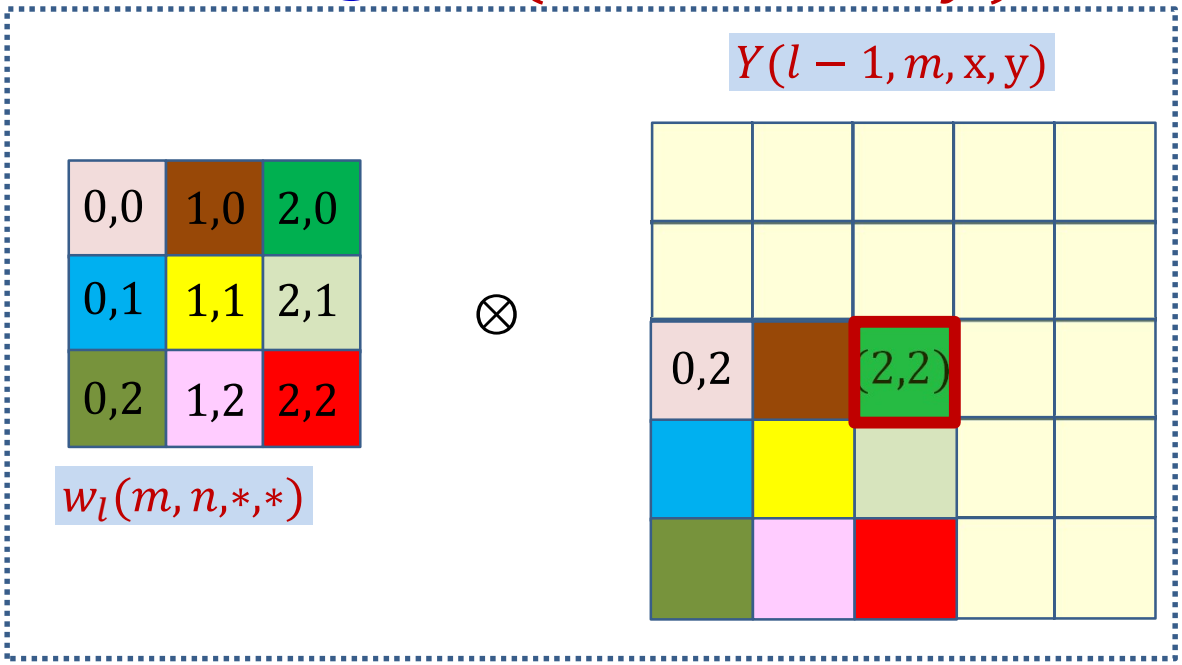


$$z(l,n,2,1) \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,0,1)$$

- **Note:** The coordinates of $z(l,n)$ and $w_l(m,n)$ sum to the coordinates of $Y(l-1,m)$

$$z(l,n,x',y') \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,2-x',2-y')$$

How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$

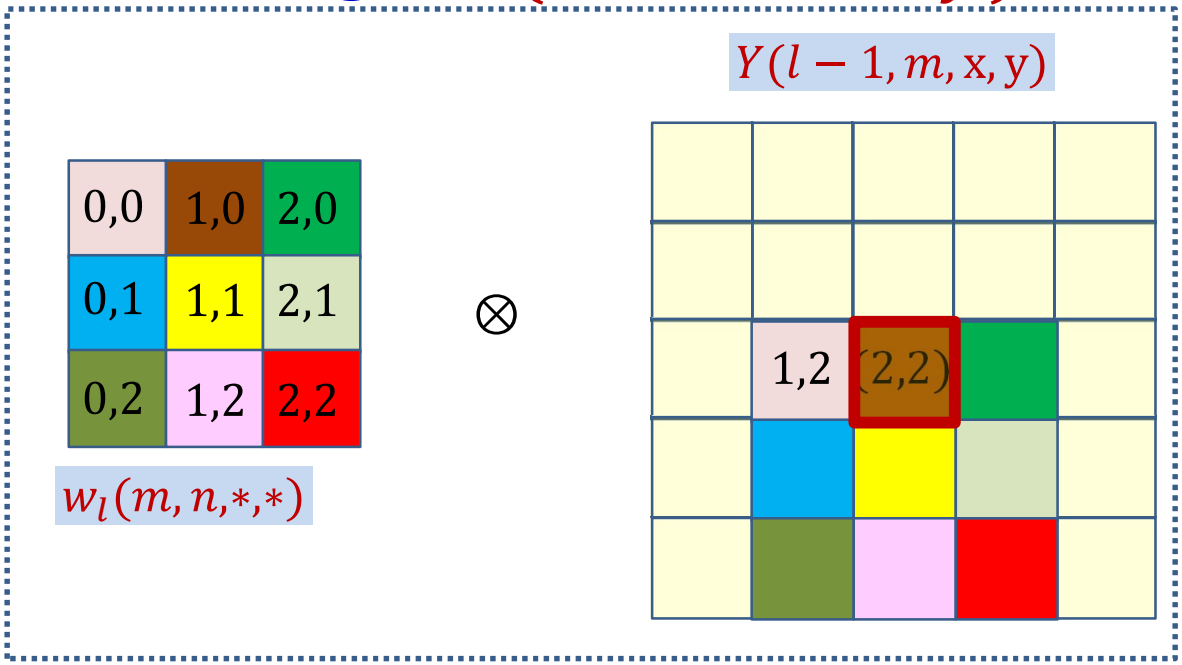


$$z(l,n,0,2) \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,2,0)$$

- **Note:** The coordinates of $z(l,n)$ and $w_l(m,n)$ sum to the coordinates of $Y(l-1,m)$

$$z(l,n,x',y') \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,2-x',2-y')$$

How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$

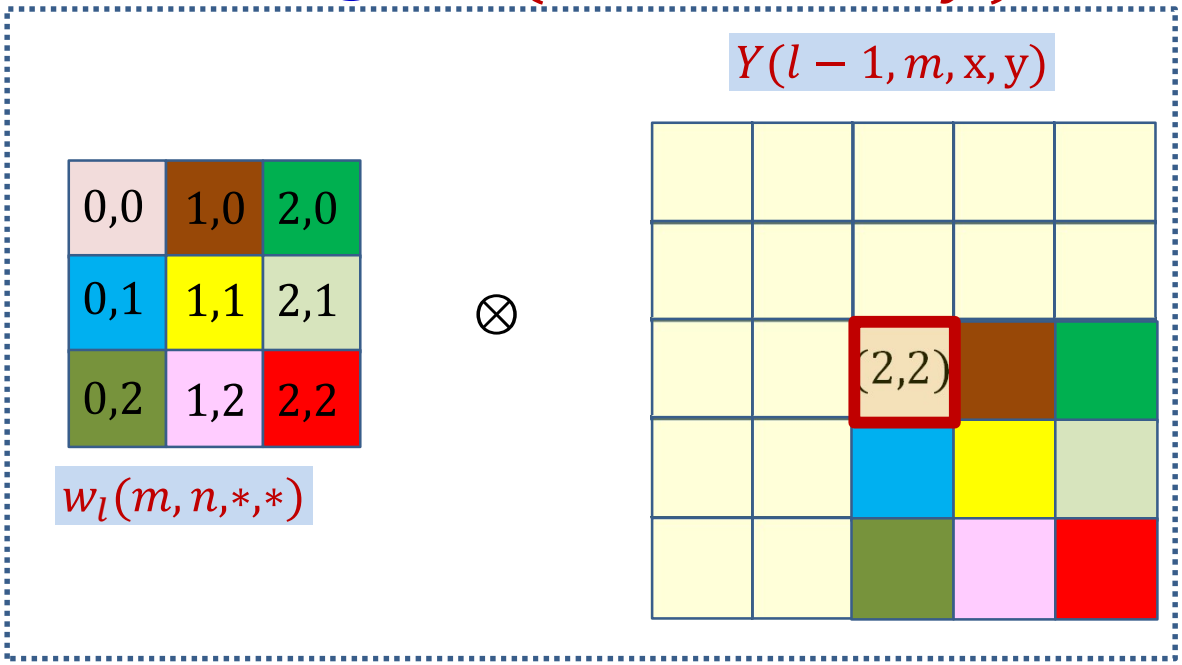


$$z(l,n,1,2) \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,2,1)$$

- **Note:** The coordinates of $z(l,n)$ and $w_l(m,n)$ sum to the coordinates of $Y(l-1,m)$

$$z(l,n,x',y') \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,2-x',2-y')$$

How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$

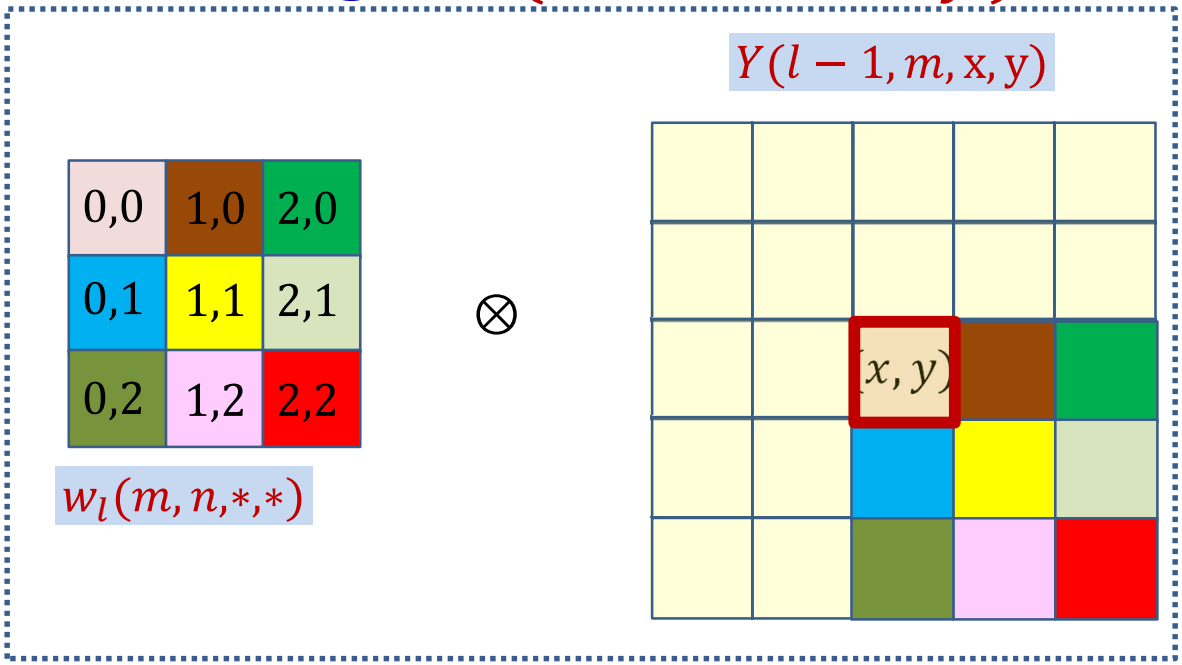


$$z(l,n,2,2) \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,0,0)$$

- **Note:** The coordinates of $z(l,n)$ and $w_l(m,n)$ sum to the coordinates of $Y(l-1,m)$

$$z(l,n,x',y') \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,2-x',2-y')$$

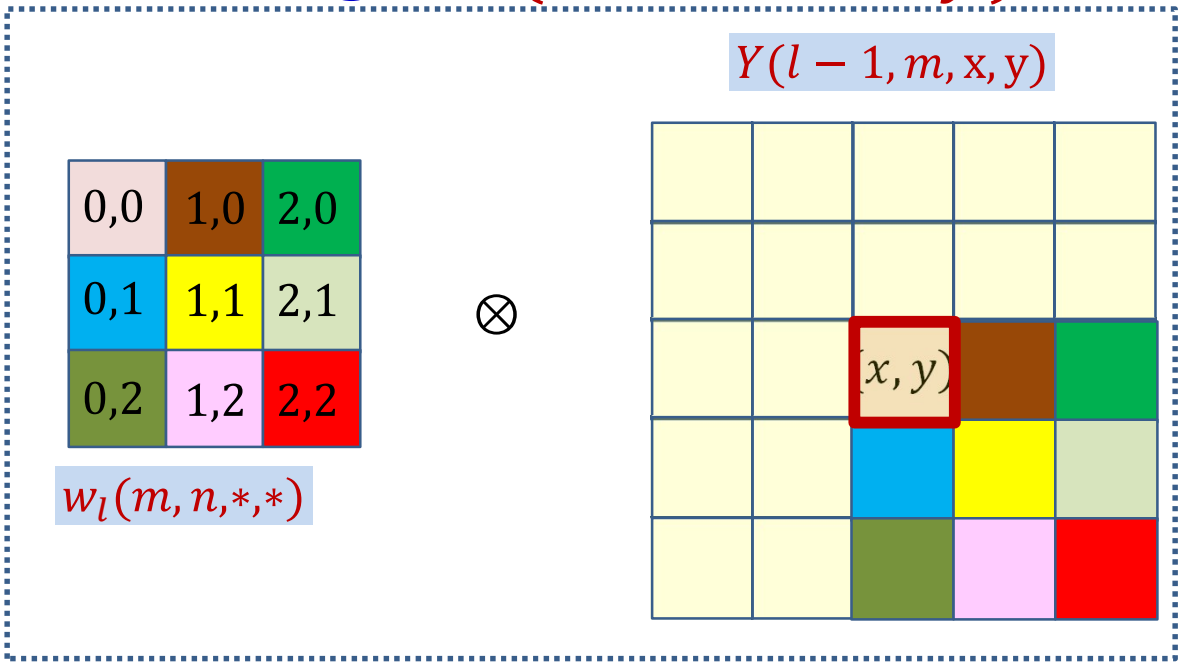
How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$



$$z(l,n,x',y') \mathrel{+}= Y(l-1,m,x,y)w_l(m,n,x-x',y-y')$$

- **Note:** The coordinates of $z(l,n)$ and $w_l(m,n)$ sum to the coordinates of $Y(l-1,m)$

How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$

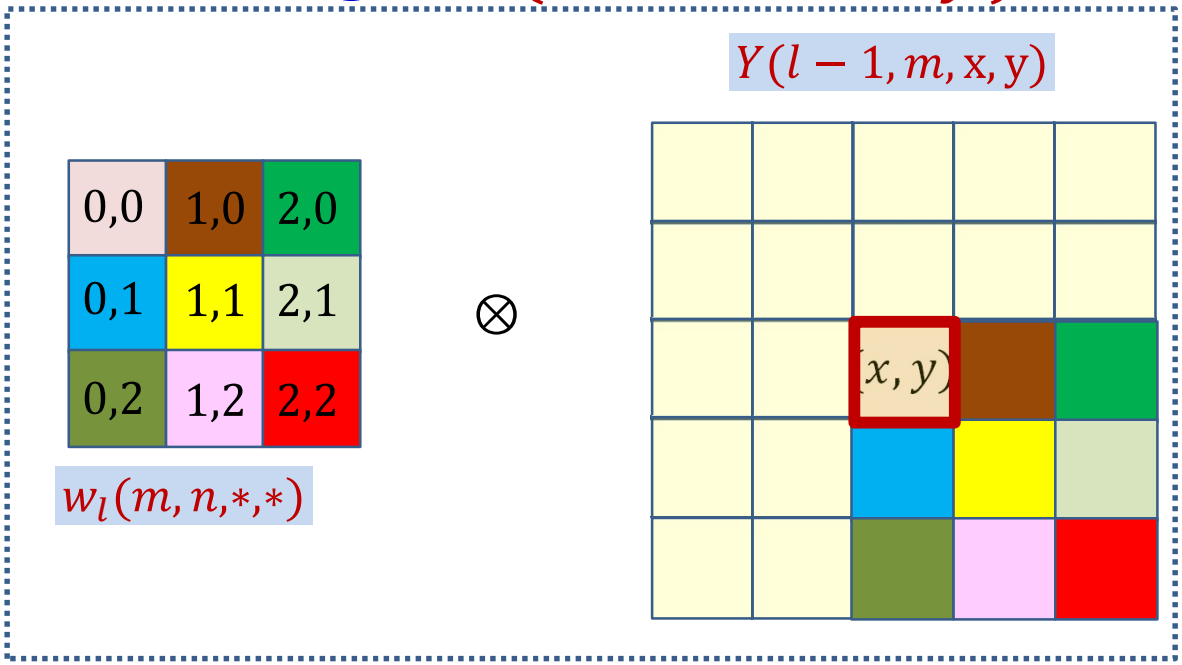


$$z(l,n,x',y') \mathrel{+}= Y(l-1,m,x,y)\,w_l(m,n,x-x',y-y')$$

$$\frac{dDiv}{dY(l-1,m,x,y)} \mathrel{+}= \frac{dDiv}{dz(l,n,x',y')}\,w_l(m,n,x-x',y-y')$$

Contribution of a single position (x', y')

How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$

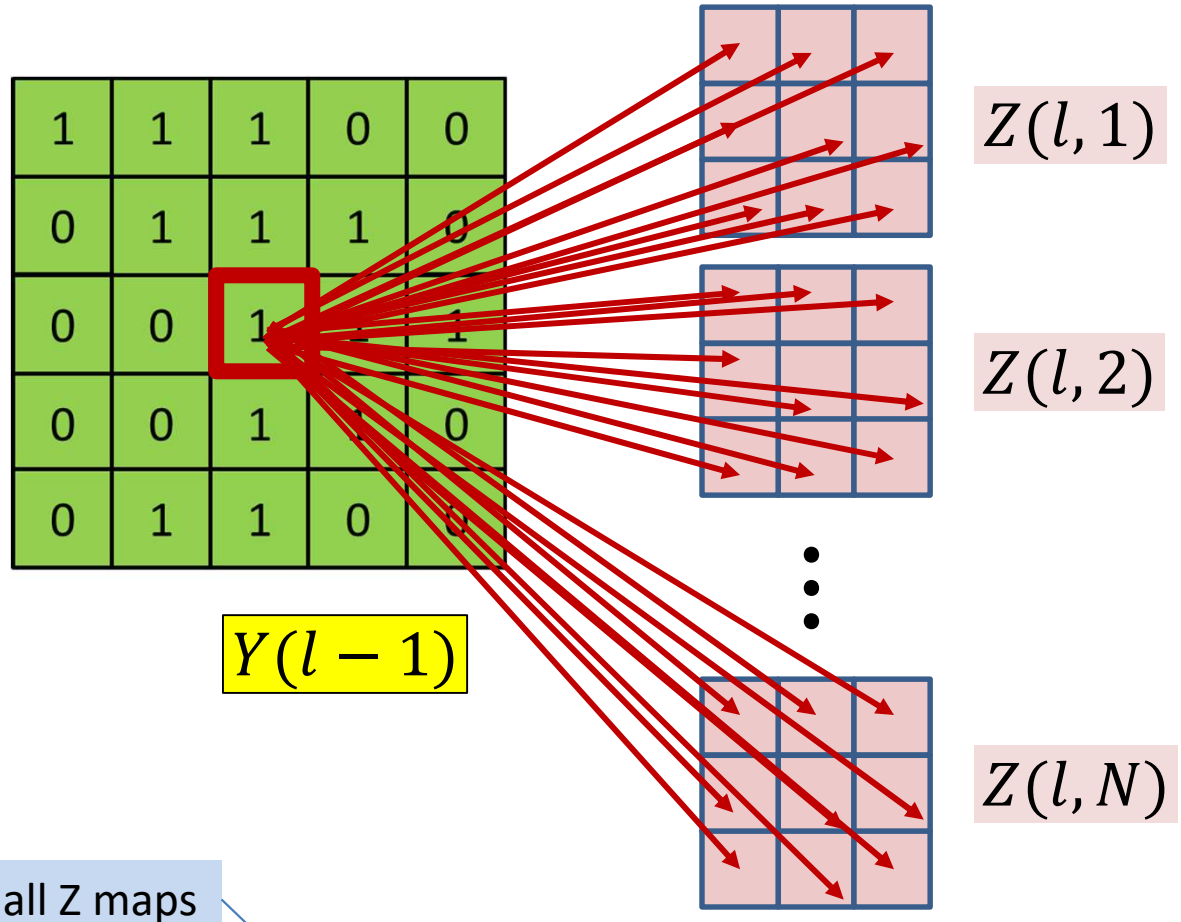


$$z(l,n,x',y') \mathrel{+}= Y(l-1,m,x,y)\,w_l(m,n,x-x',y-y')$$

$$\frac{dDiv}{dY(l-1,m,x,y)} \mathrel{+}= \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')}\, w_l(m,n,x-x',y-y')$$

Contribution of the entire nth affine map $z(l,n)$

BP: Convolutional layer



Summing over all Z maps

$$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_{n} \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$$

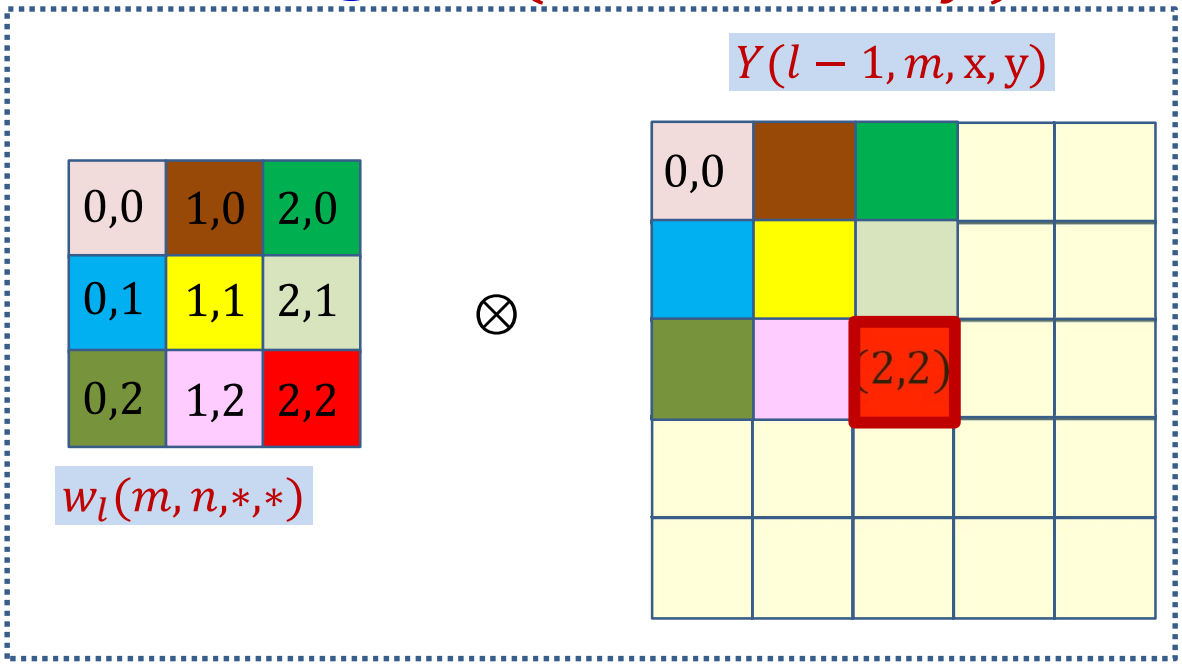
Computing derivative for $\boldsymbol{Y(l-1,m,*,*)}$

- The derivatives for every element of every map in $Y(l-1)$ by direct implementation of the formula:

$$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_{n}\sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$$

- But this is actually a convolution!
  - Let's see how

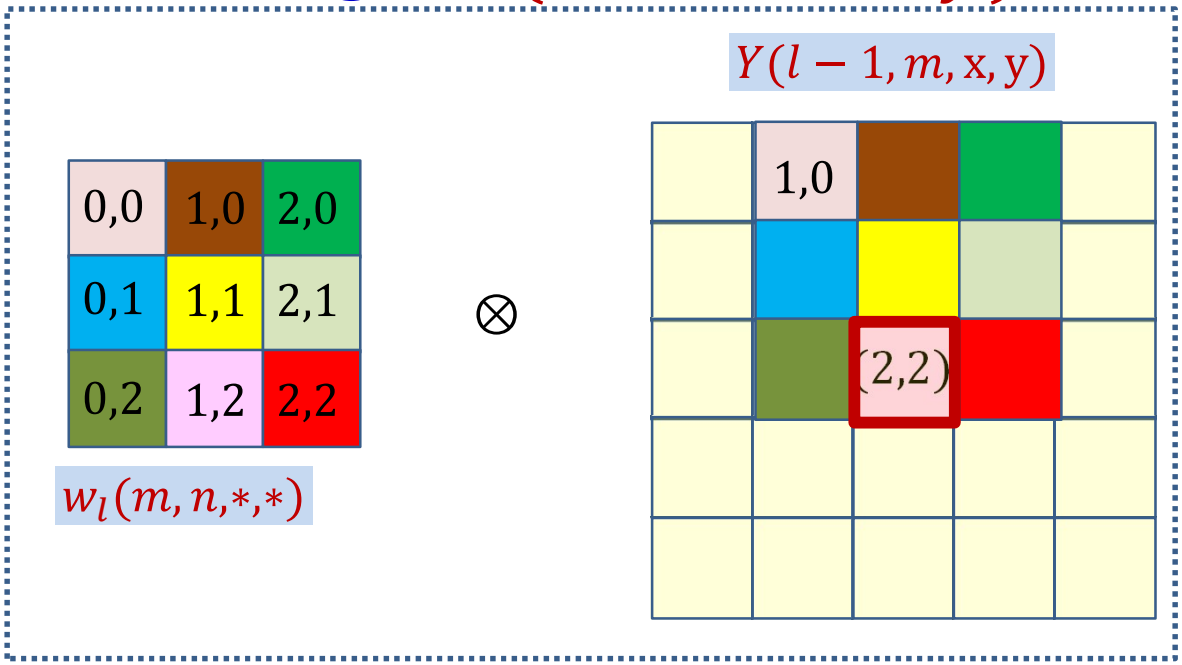
How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$

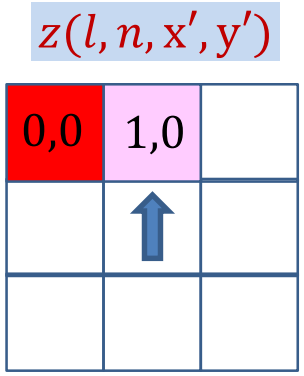


$$z(l,n,0,0) \mathrel{+}= Y(l-1,m,2,2)w_l(m,n,2,2)$$

$$\frac{dDiv}{dY(l-1,m,2,2)} \mathrel{+}= \frac{dDiv}{dz(l,n,0,0)} w_l(m,n,2,2)$$

How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$

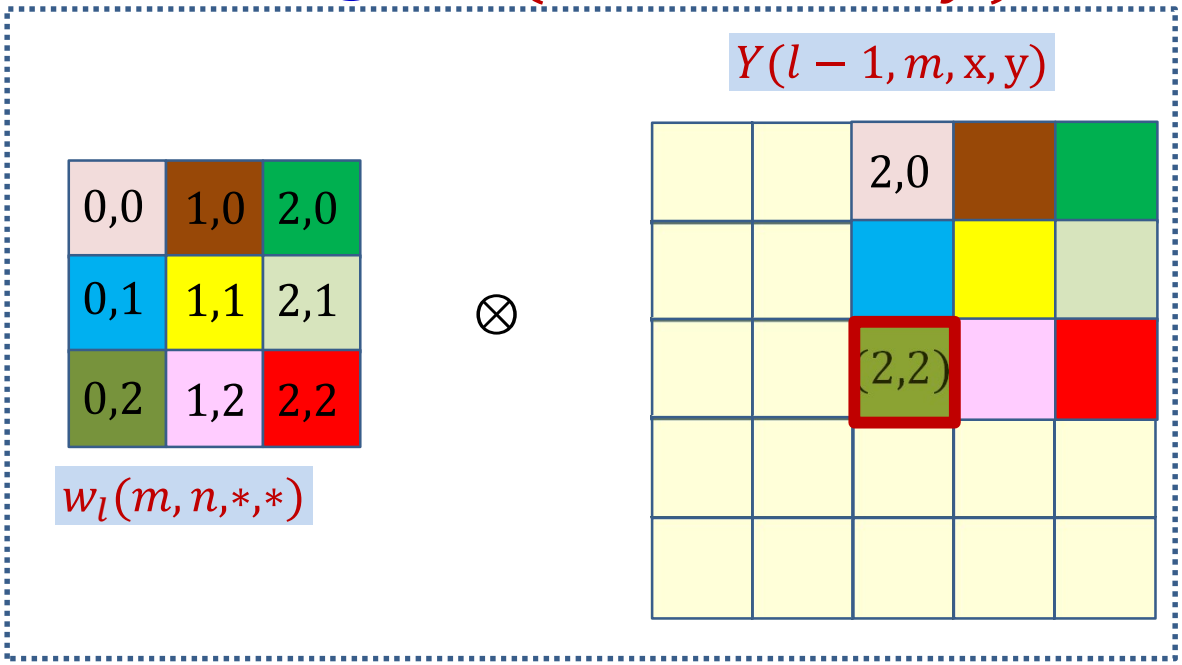




$$z(l,n,1,0) \mathrel{+}= Y(l-1,m,2,2)\,w_l(m,n,1,2)$$

$$\frac{dDiv}{dY(l-1,m,2,2)} \mathrel{+}= \frac{dDiv}{dz(l,n,1,0)}\,w_l(m,n,1,2)$$

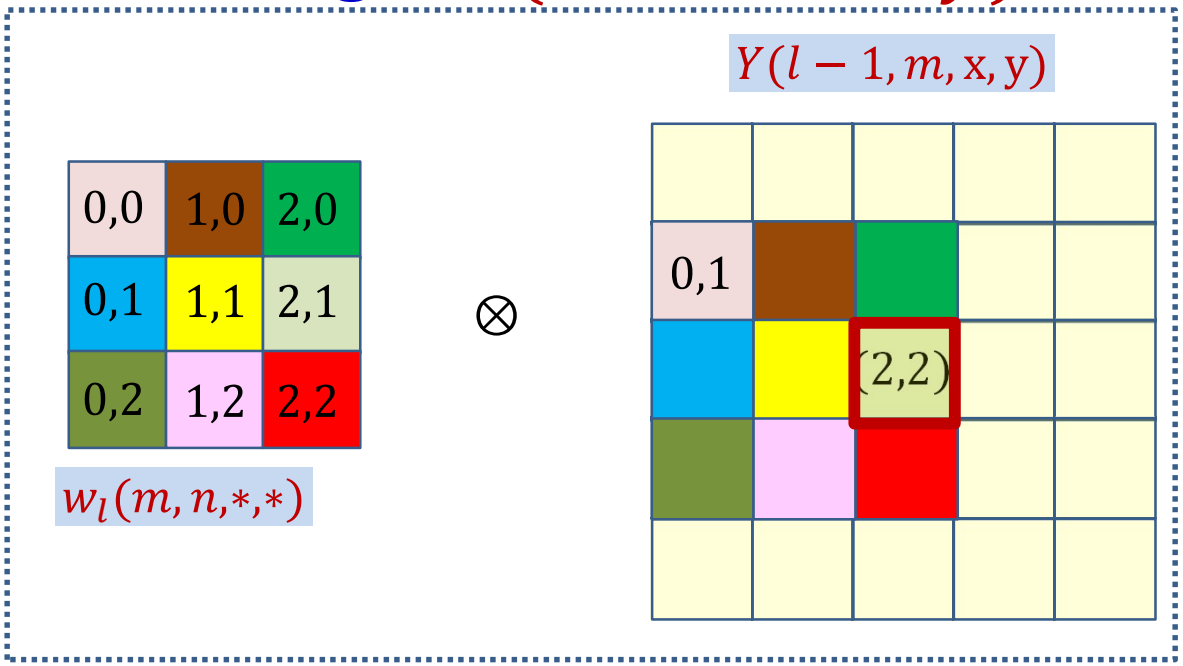
How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$



$$z(l,n,2,0) \mathrel{+}= Y(l-1,m,2,2)\,w_l(m,n,0,2)$$

$$\frac{dDiv}{dY(l-1,m,2,2)} \mathrel{+}= \frac{dDiv}{dz(l,n,2,0)}\,w_l(m,n,0,2)$$

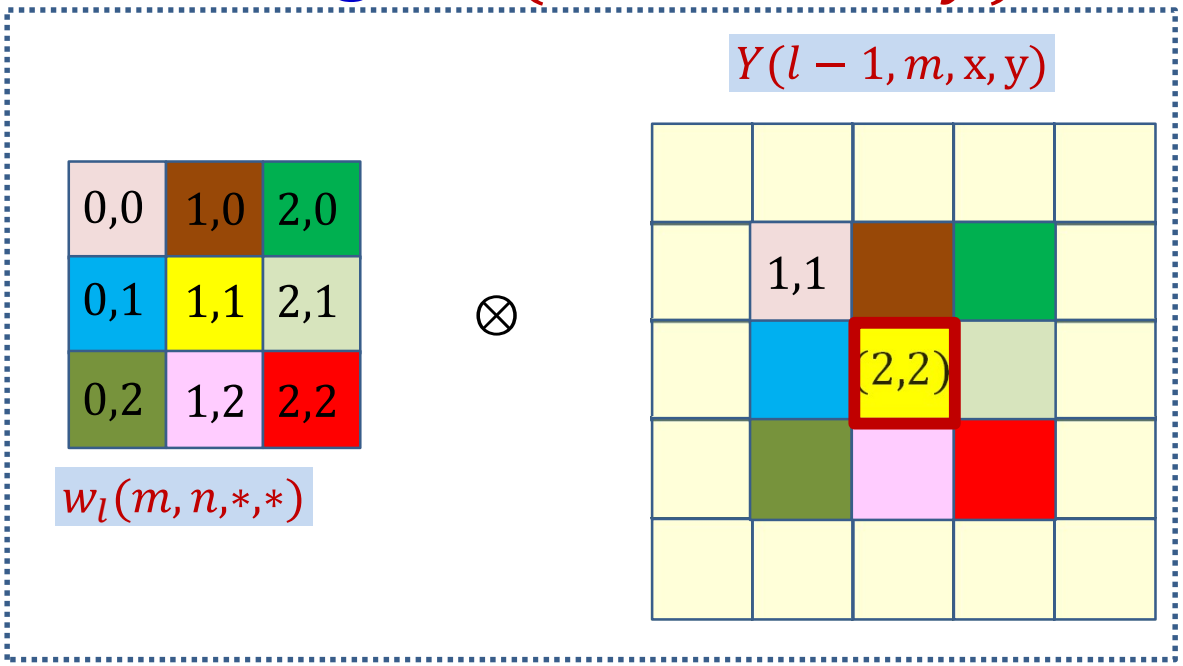
How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$



$$z(l,n,0,1) \mathrel{+}= Y(l-1,m,2,2)\,w_l(m,n,2,1)$$

$$\frac{dDiv}{dY(l-1,m,2,2)} \mathrel{+}= \frac{dDiv}{dz(l,n,0,1)}\,w_l(m,n,2,1)$$

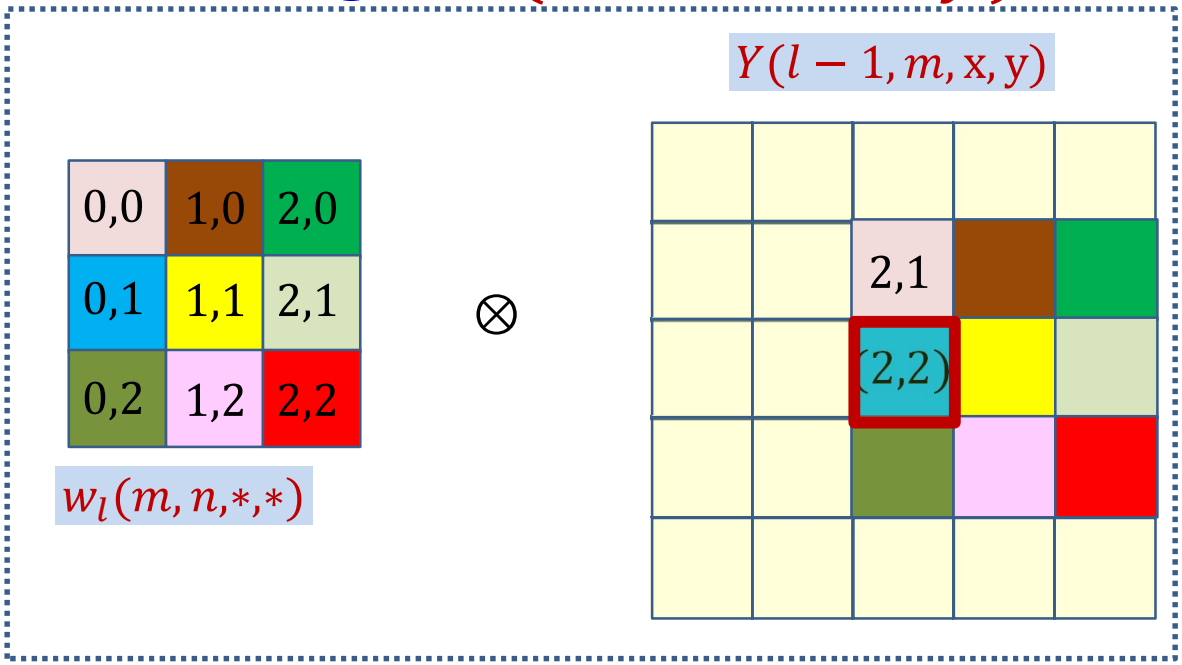
How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$



$$z(l,n,1,1) \mathrel{+}= Y(l-1,m,2,2)\,w_l(m,n,1,1)$$

$$\frac{dDiv}{dY(l-1,m,2,2)} \mathrel{+}= \frac{dDiv}{dz(l,n,1,1)}\,w_l(m,n,1,1)$$

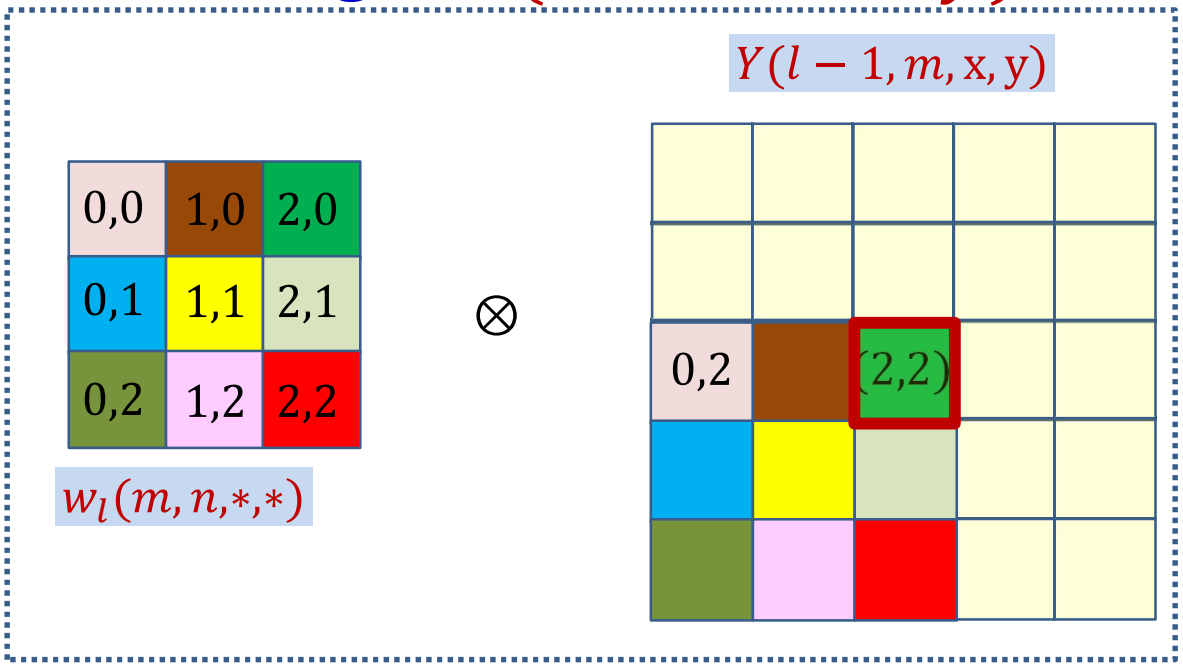
How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$



$$z(l,n,2,1) \mathrel{+}= Y(l-1,m,2,2)\,w_l(m,n,0,1)$$

$$\frac{dDiv}{dY(l-1,m,2,2)} \mathrel{+}= \frac{dDiv}{dz(l,n,2,1)}\,w_l(m,n,0,1)$$

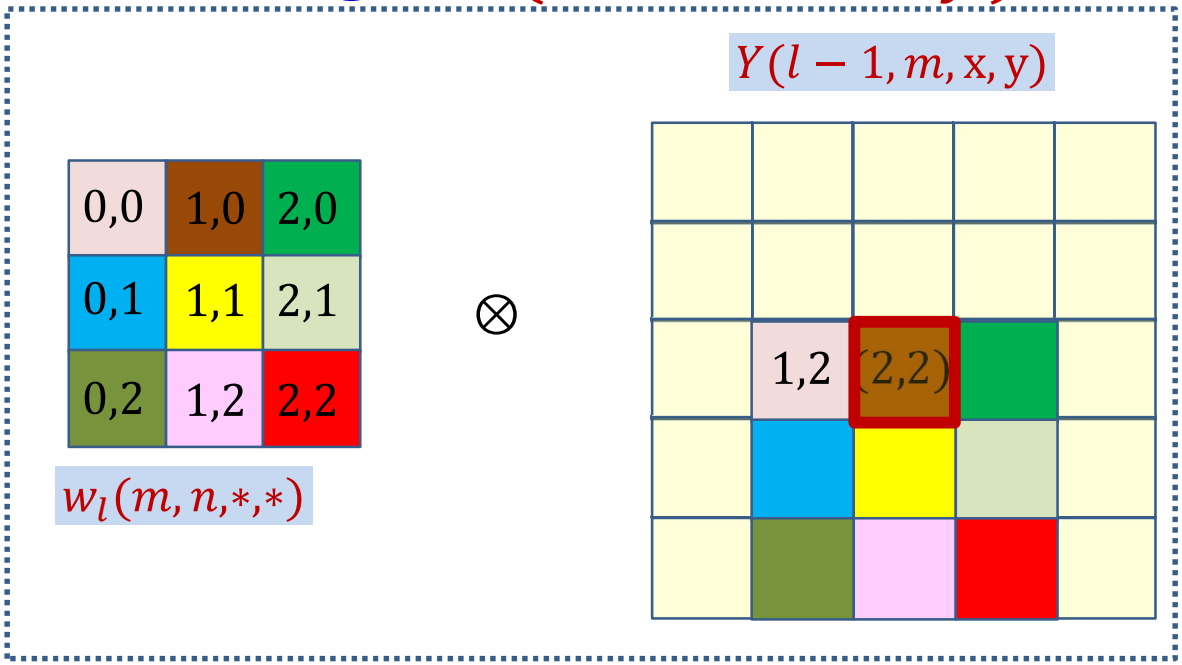
How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$



$$z(l,n,0,2) \mathrel{+}= Y(l-1,m,2,2)\,w_l(m,n,2,0)$$

$$\frac{dDiv}{dY(l-1,m,2,2)} \mathrel{+}= \frac{dDiv}{dz(l,n,0,2)}\,w_l(m,n,2,0)$$

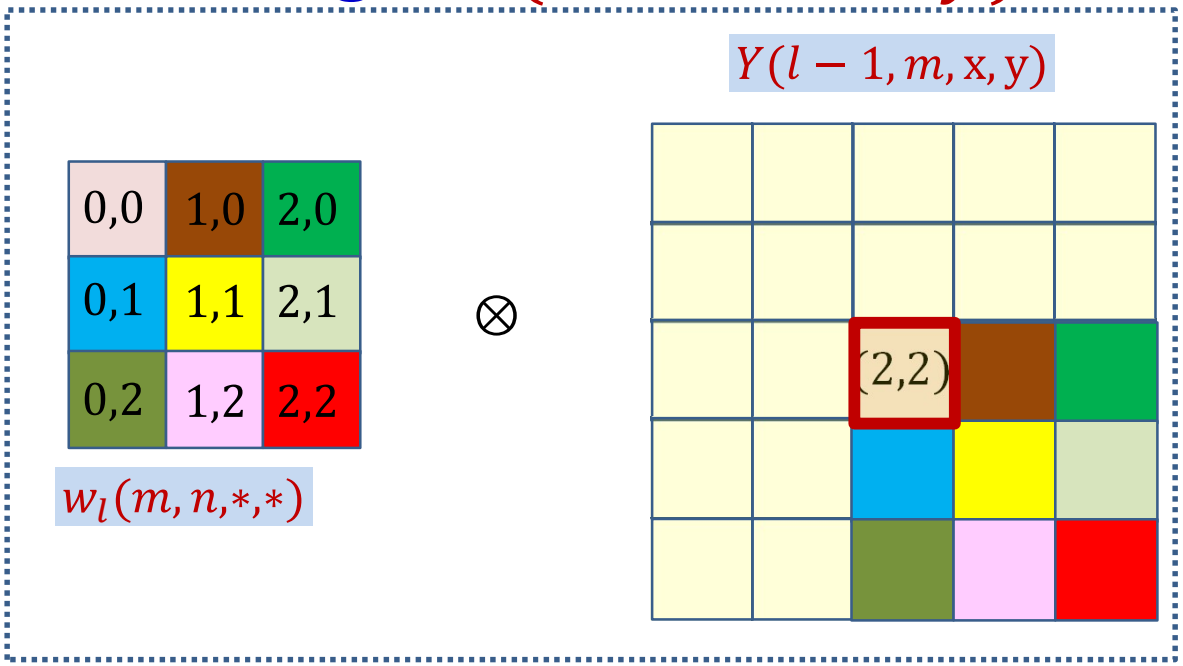
How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$



$$z(l,n,1,2) \mathrel{+}= Y(l-1,m,2,2)\,w_l(m,n,2,1)$$

$$\frac{dDiv}{dY(l-1,m,2,2)} \mathrel{+}= \frac{dDiv}{dz(l,n,1,2)}\,w_l(m,n,1,0)$$

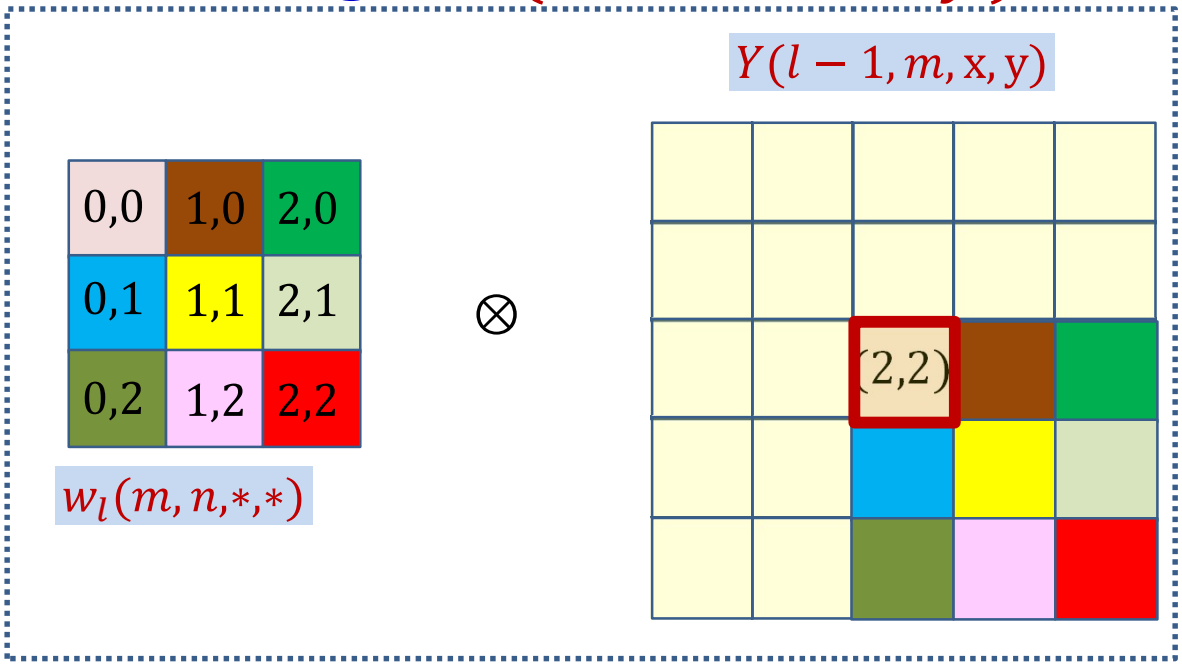
How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$



$$z(l,n,2,2) \mathrel{+}= Y(l-1,m,2,2)\,w_l(m,n,0,0)$$

$$\frac{dDiv}{dY(l-1,m,2,2)} \mathrel{+}= \frac{dDiv}{dz(l,n,2,2)}\,w_l(m,n,0,0)$$

How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$

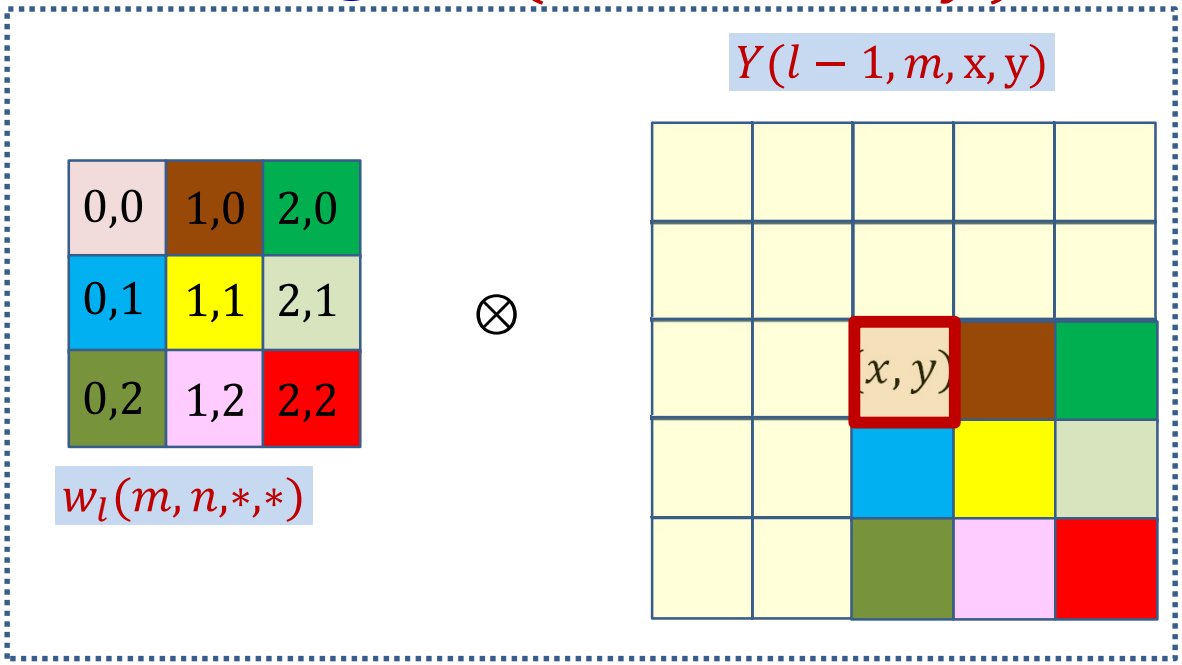


$$z(l,n,x',y') \mathrel{+}= Y(l-1,m,2,2)\,w_l(m,n,2-\mathrm{x}',2-\mathrm{y}')$$

$$\frac{dDiv}{dY(l-1,m,2,2)} \mathrel{+}= \frac{dDiv}{dz(l,n,x',y')}\,w_l(m,n,2-x',2-y')$$

- The derivative at $Y(l-1,m,2,2)$ is the sum of component-wise product of the elements of the *flipped* filter and the elements of the derivative at $z(l,m,.,.)$
- The flipped filter is positioned with its bottom right square at (2,2) on the Z derivative map

How a single $Y(l-1,m,x,y)$ influences $z(l,n,x',y')$

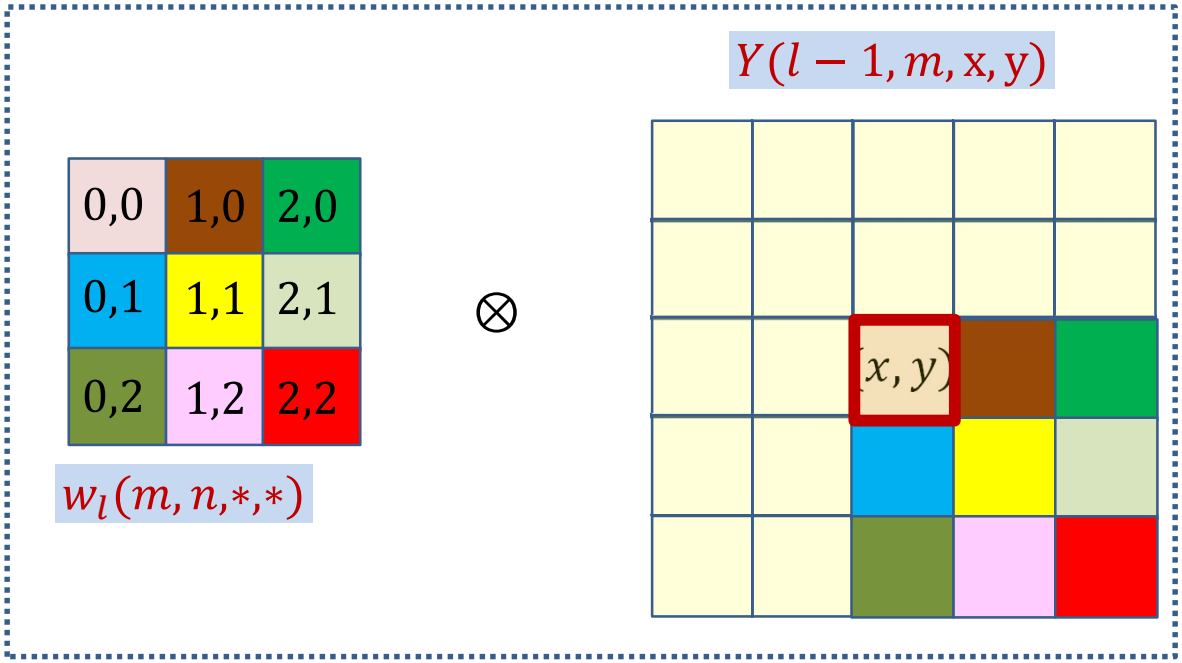


$$z(l,n,x',y') \mathrel{+}= Y(l-1,m,x,y)w_l(m,n,x-x',y-y')$$

$$\frac{dDiv}{dY(l-1,m,x,y)} \mathrel{+}= \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$$

- The derivative at $Y(l-1,m,x,y)$ is the sum of component-wise product of the flipped filter and the elements of the derivative at $z(l,m,.,.)$
- The flipped filter is positioned with its bottom right corner at (x,y)

Derivative at $Y(l-1,m,x,y)$ from a single $Z(l,n)$ map



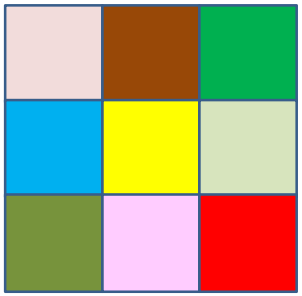
$$z(l,n,x',y') \mathrel{+}= Y(l-1,m,x,y)\,w_l(m,n,x-x',y-y')$$

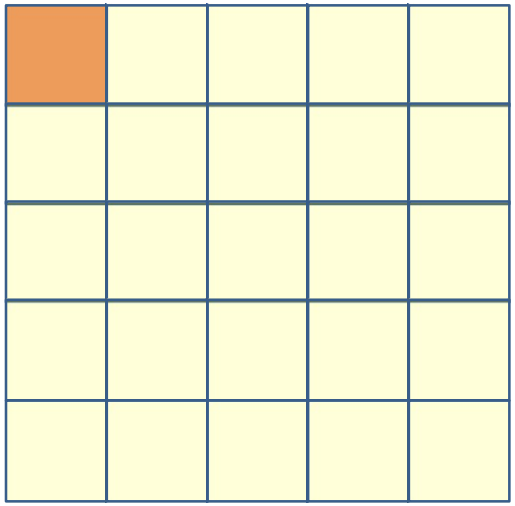
$$\frac{dDiv}{dY(l-1,m,x,y)} \mathrel{+}= \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')}\, w_l(m,n,x-x',x-y')$$

Contribution of the entire nth affine map $z(l,n,*,*)$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

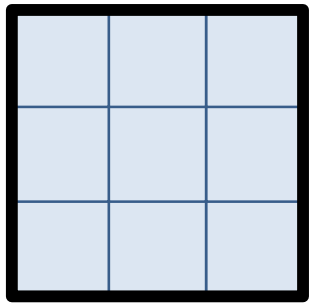
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

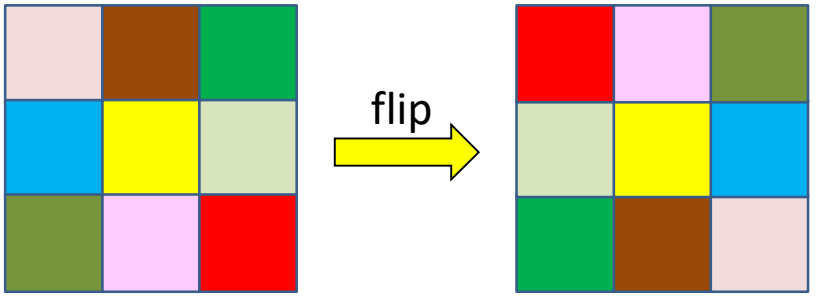
$=$

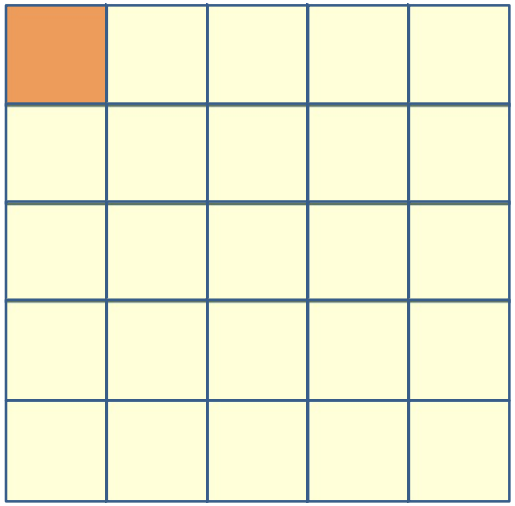


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

$w_l(m,n,*,*)$

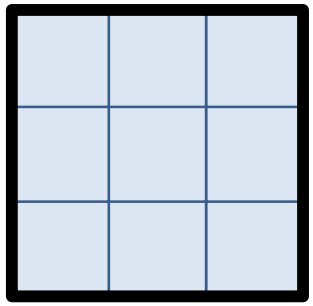




$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

=

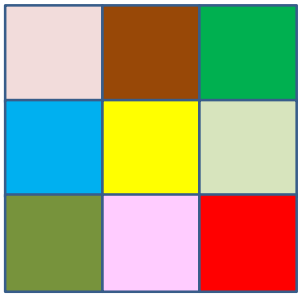
Zero pad with K-1 rows and cols on every side

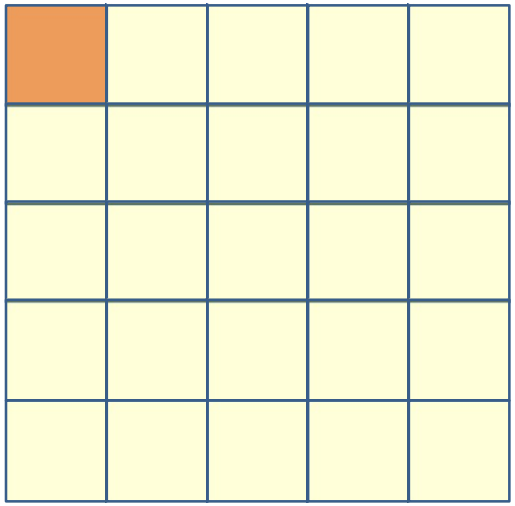


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

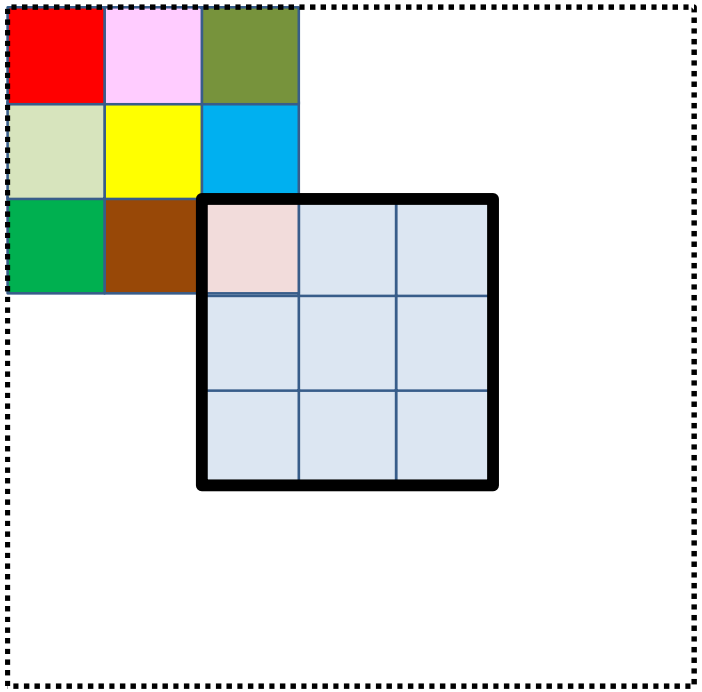
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

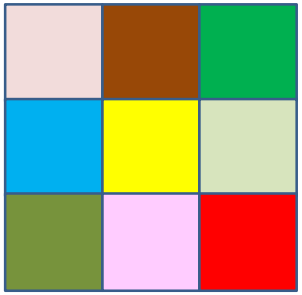
=

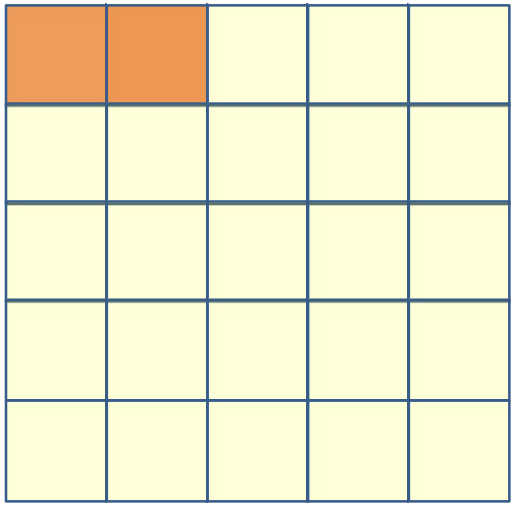


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

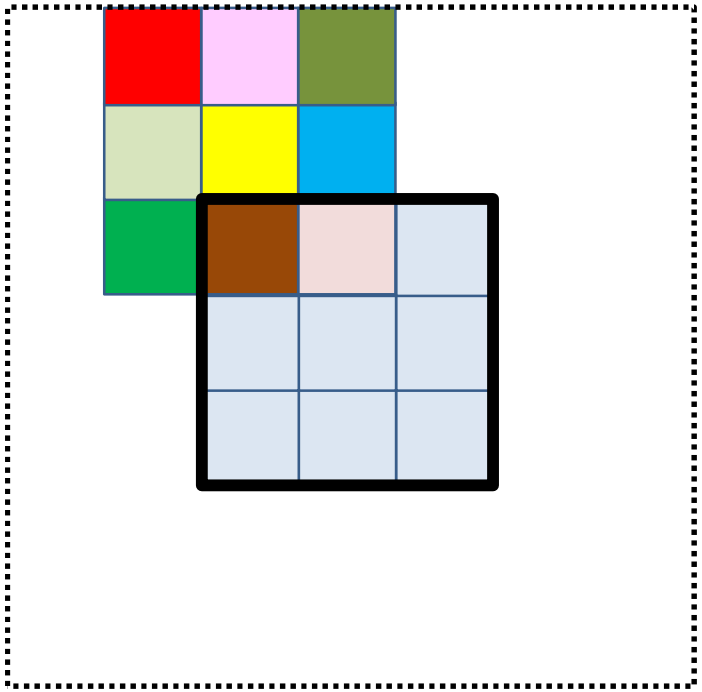
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

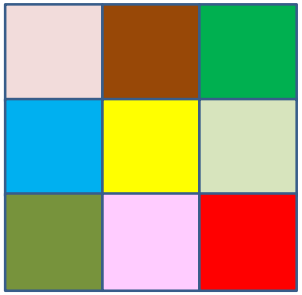
=

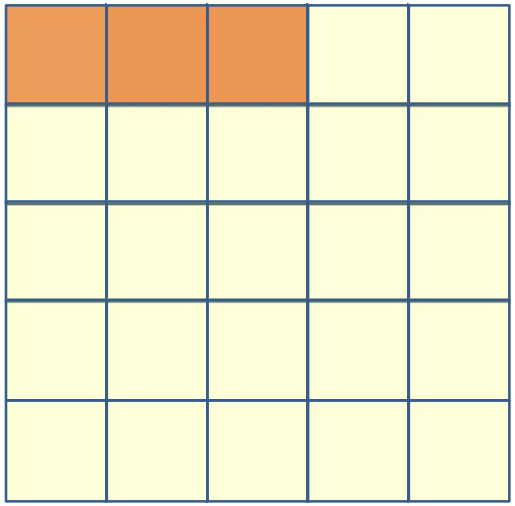


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

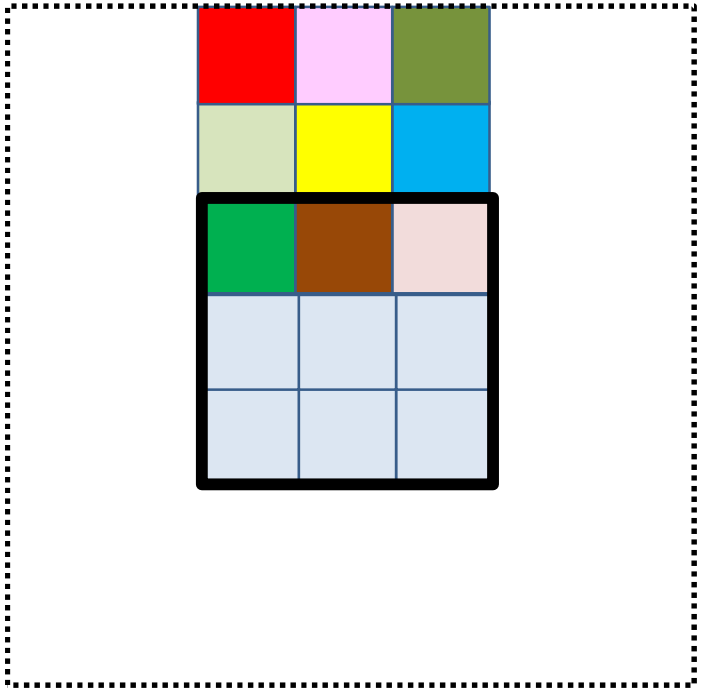
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

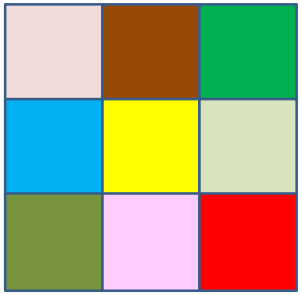
=

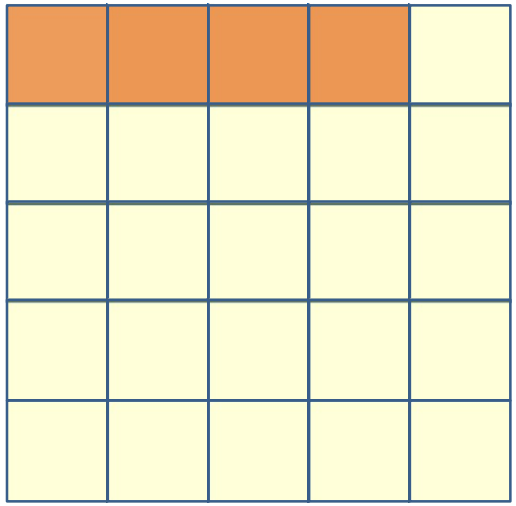


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

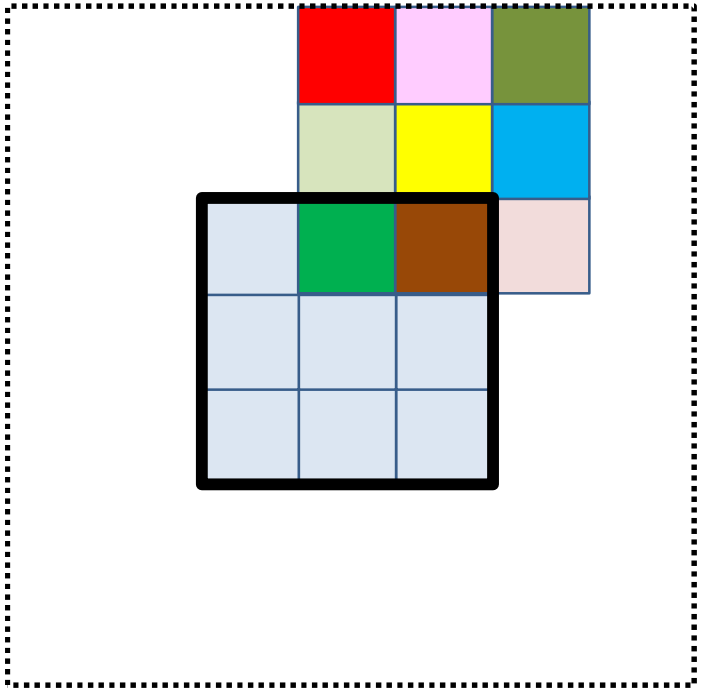
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

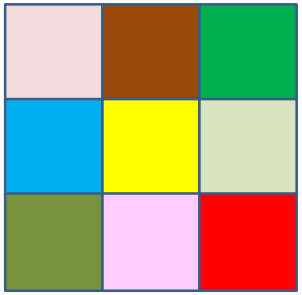
=

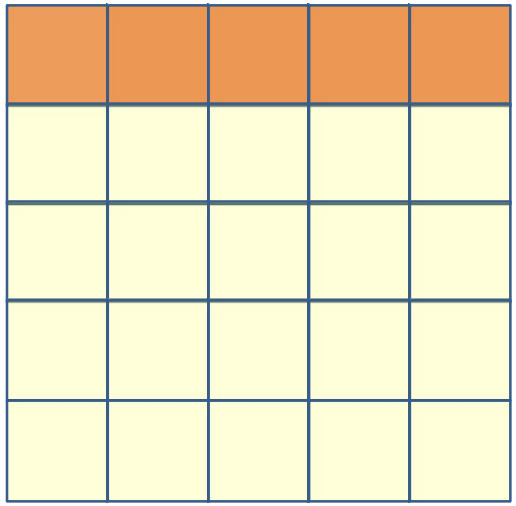


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

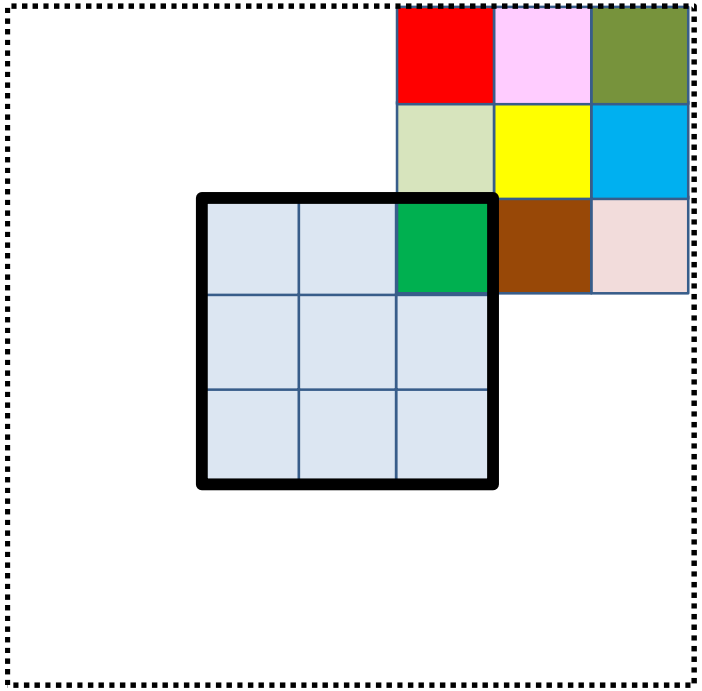
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

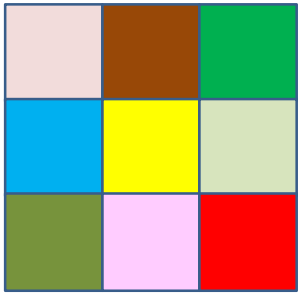
=

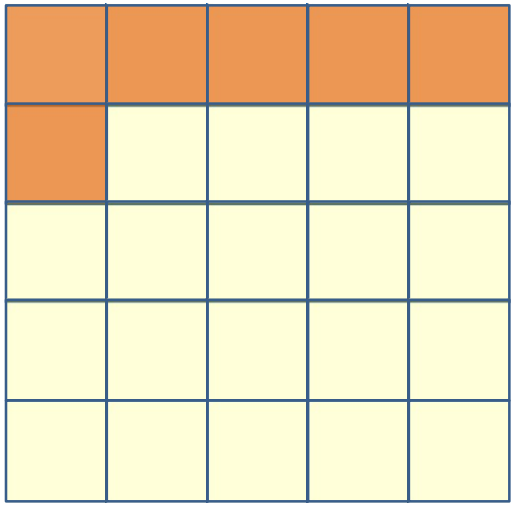


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

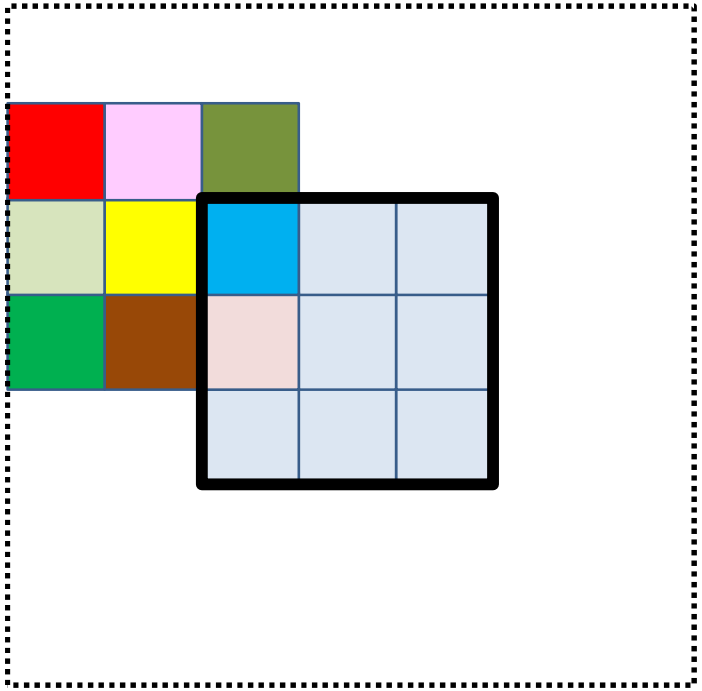
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

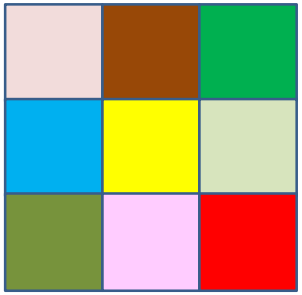
=

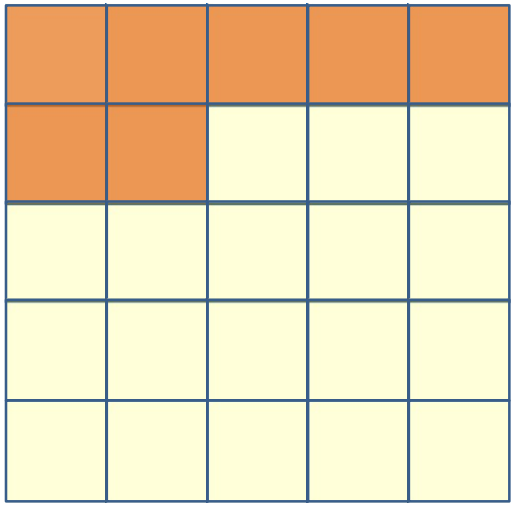


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

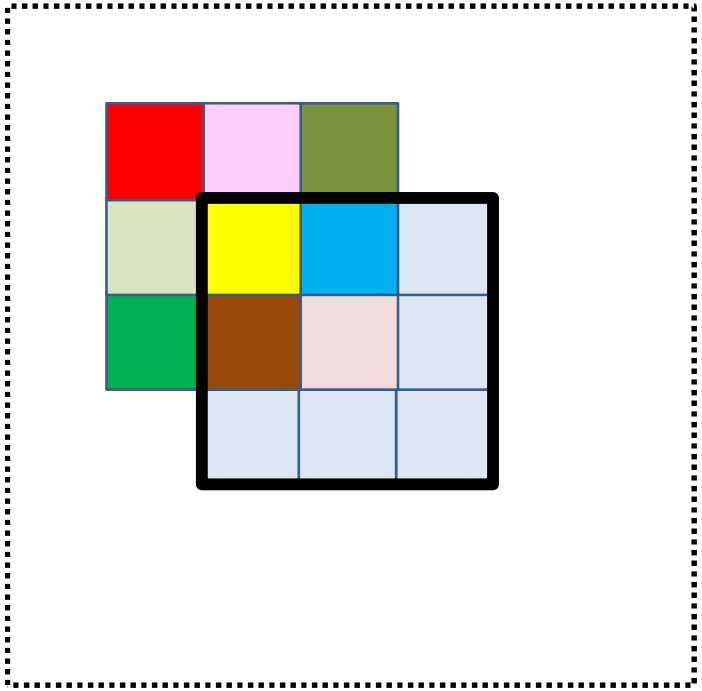
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

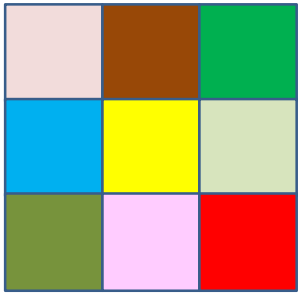
=

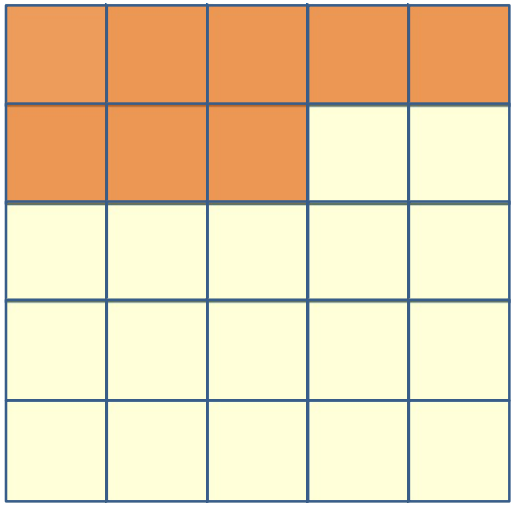


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

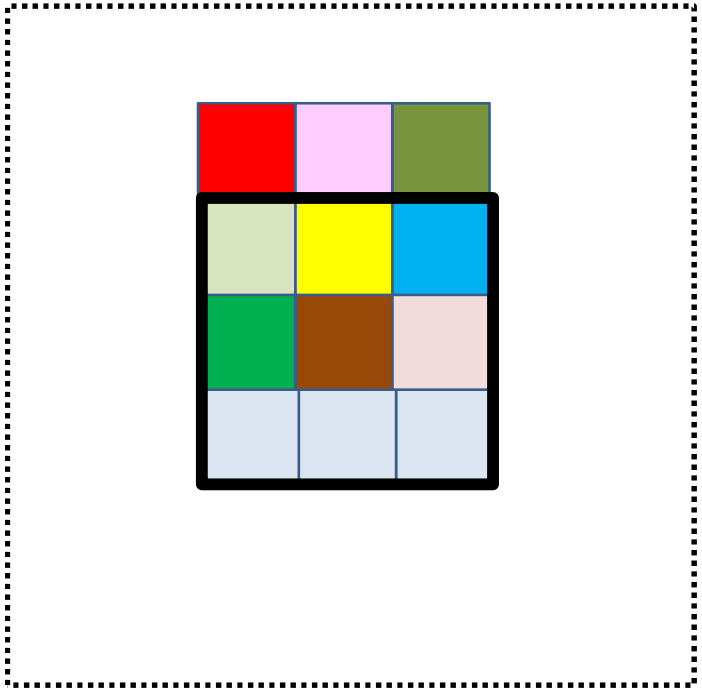
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

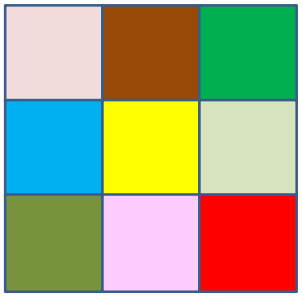
=

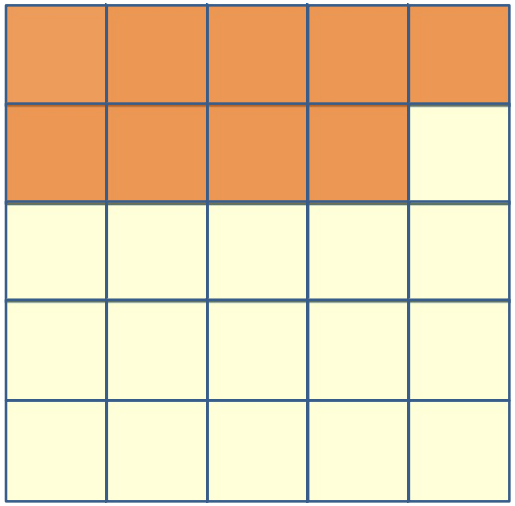


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

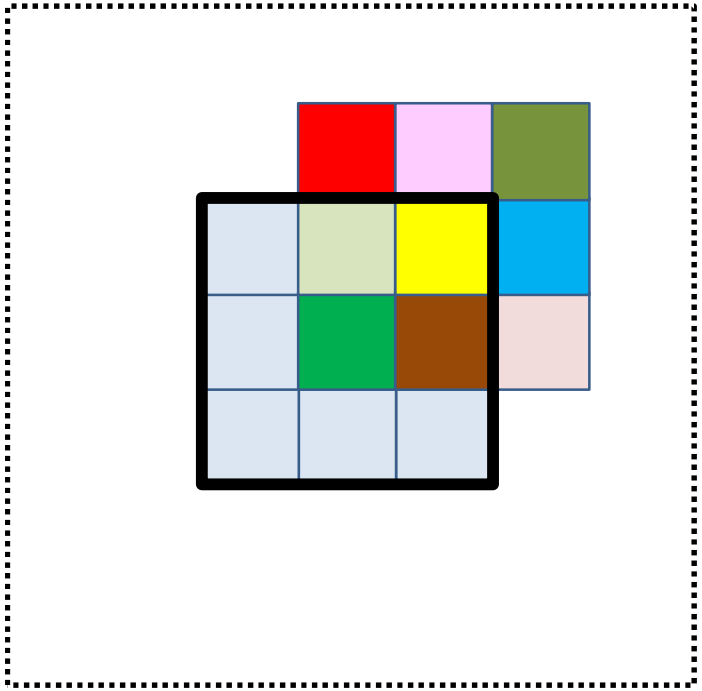
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

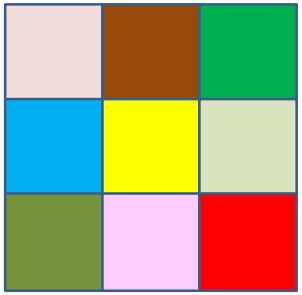
=

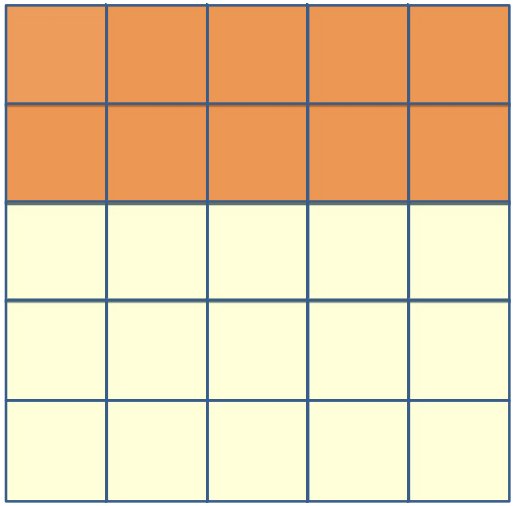


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

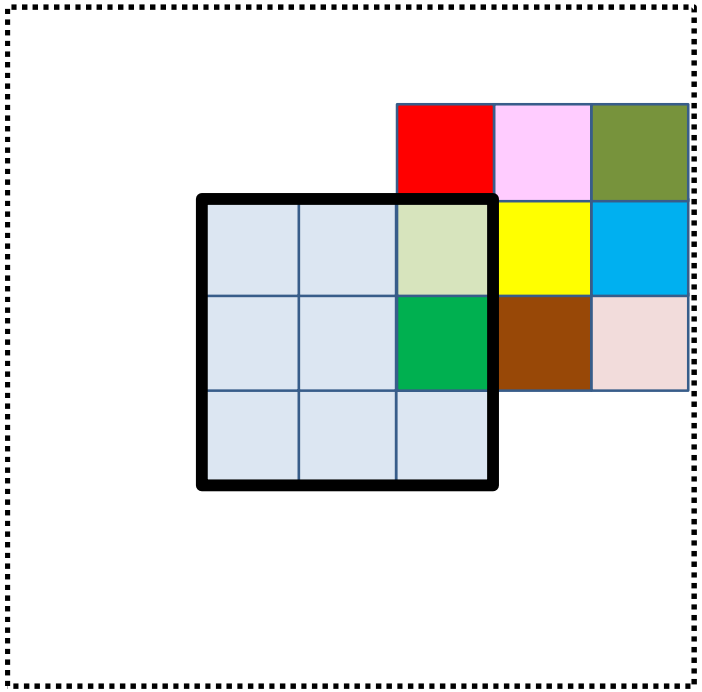
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

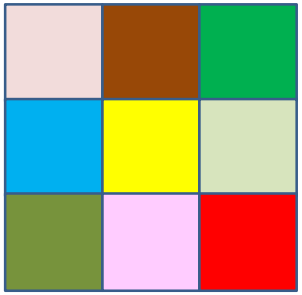
=

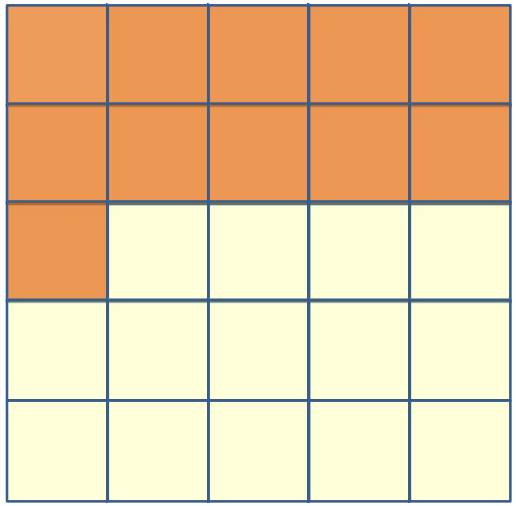


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

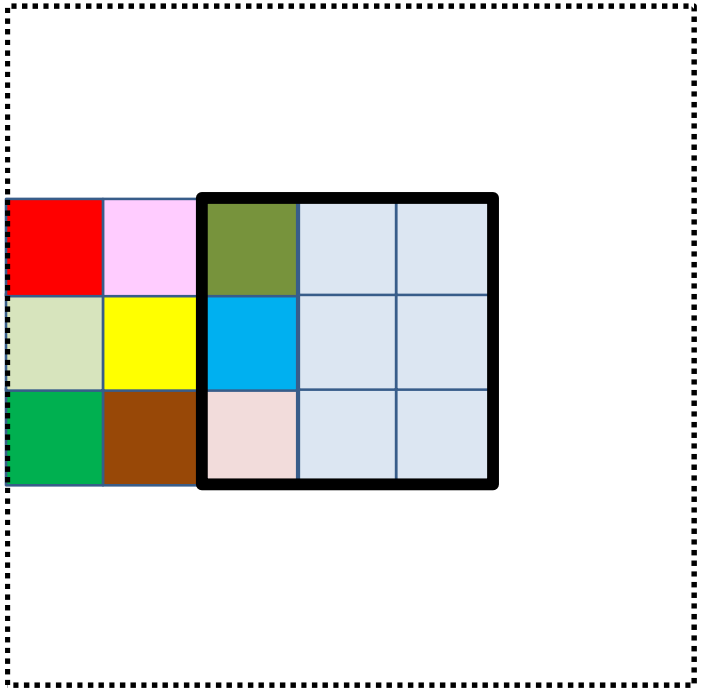
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

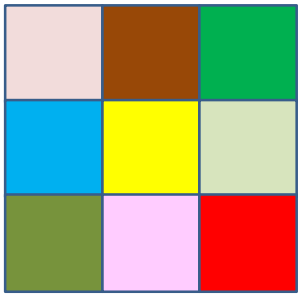
=

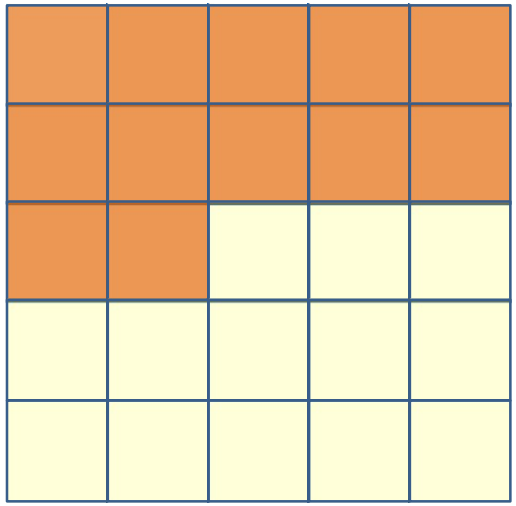


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

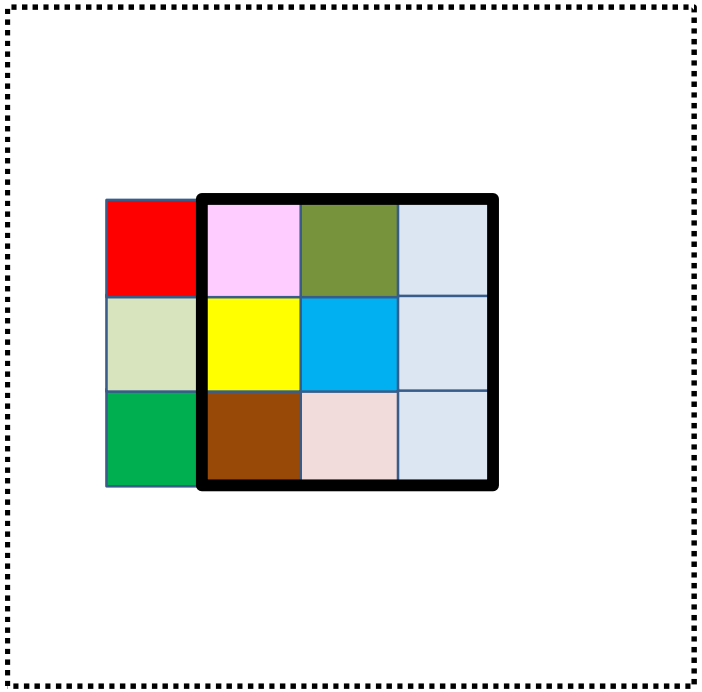
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

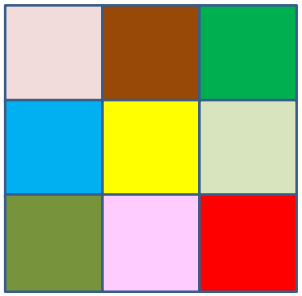
=

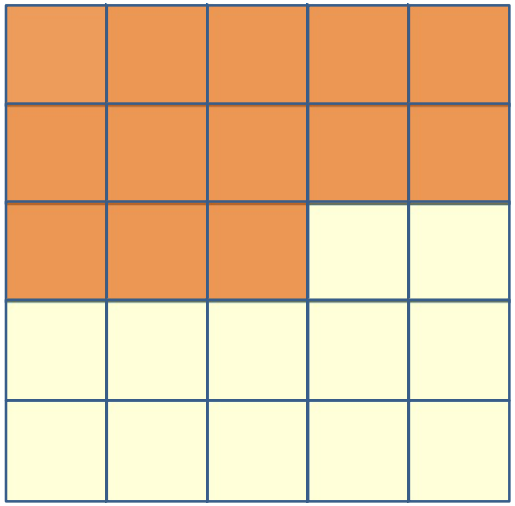


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

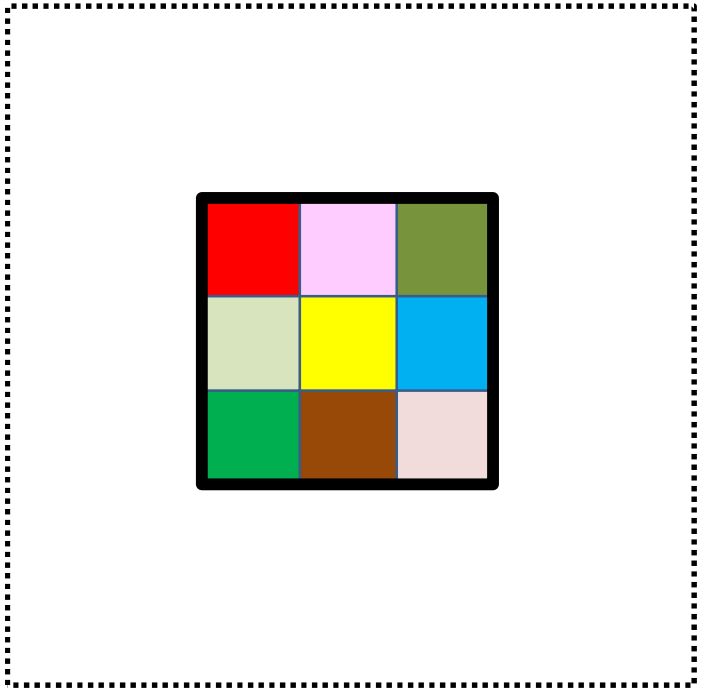
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

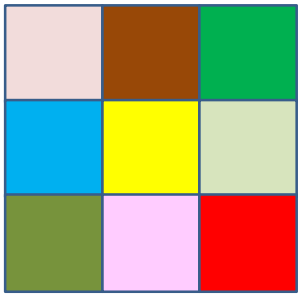
=

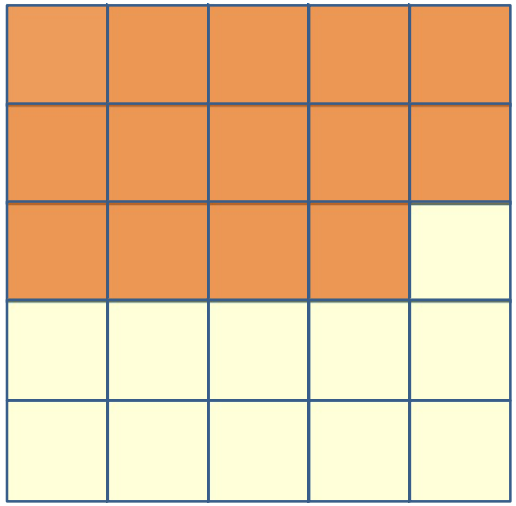


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

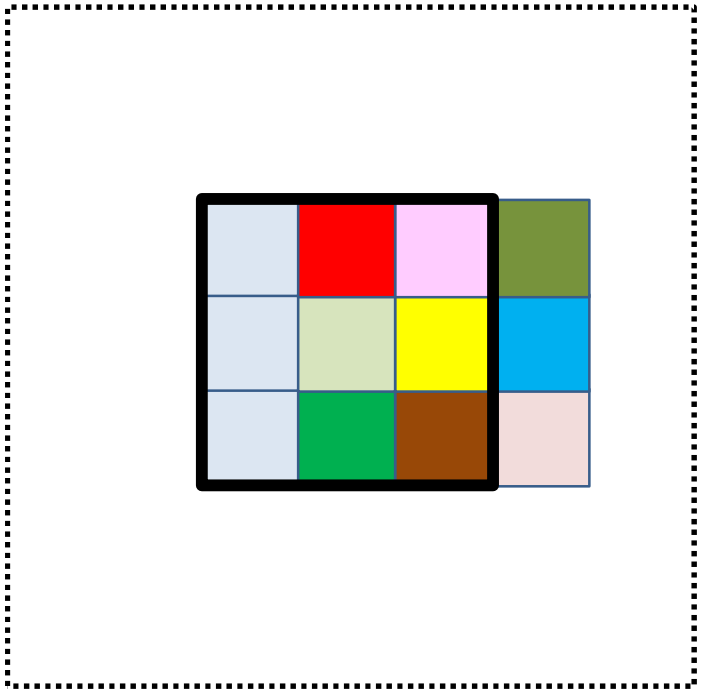
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

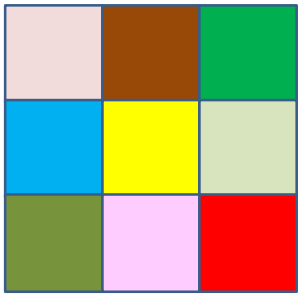
=

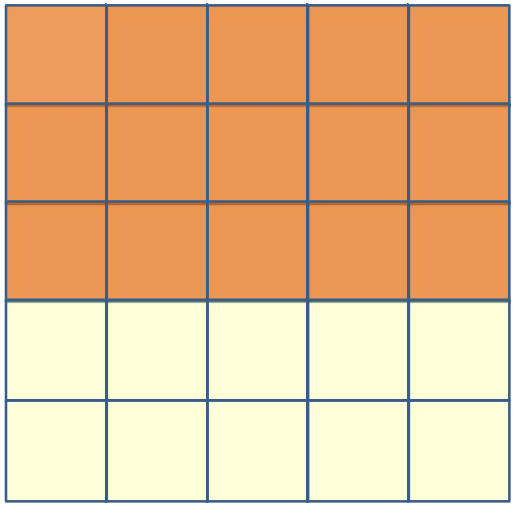


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

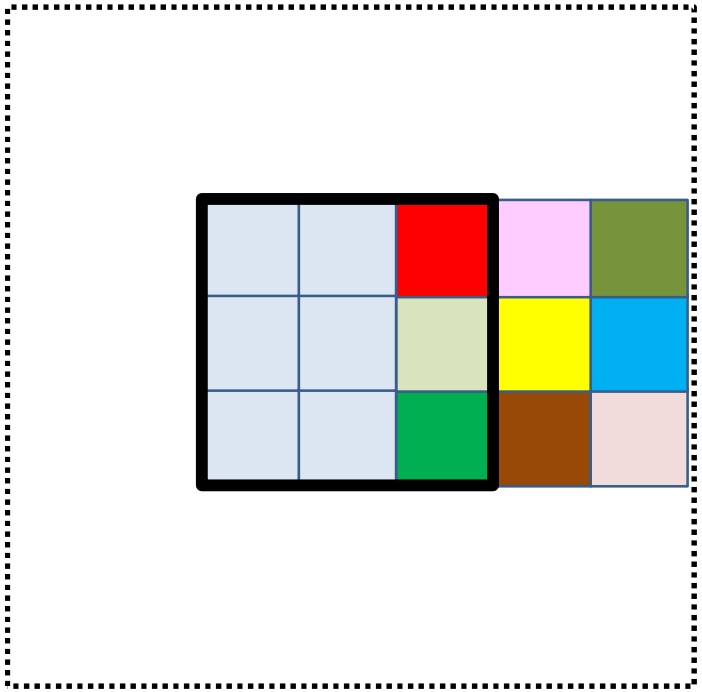
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

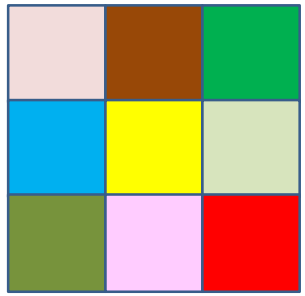
=

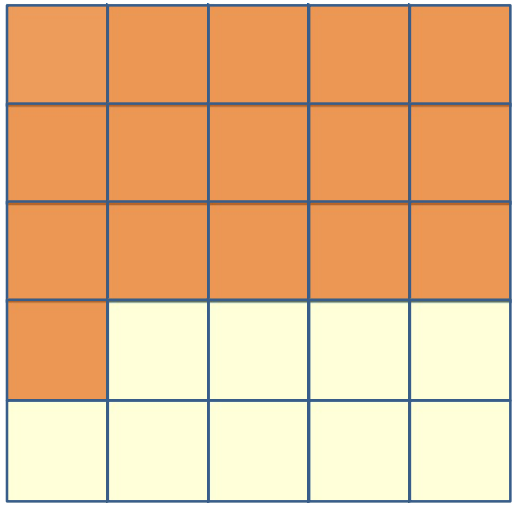


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

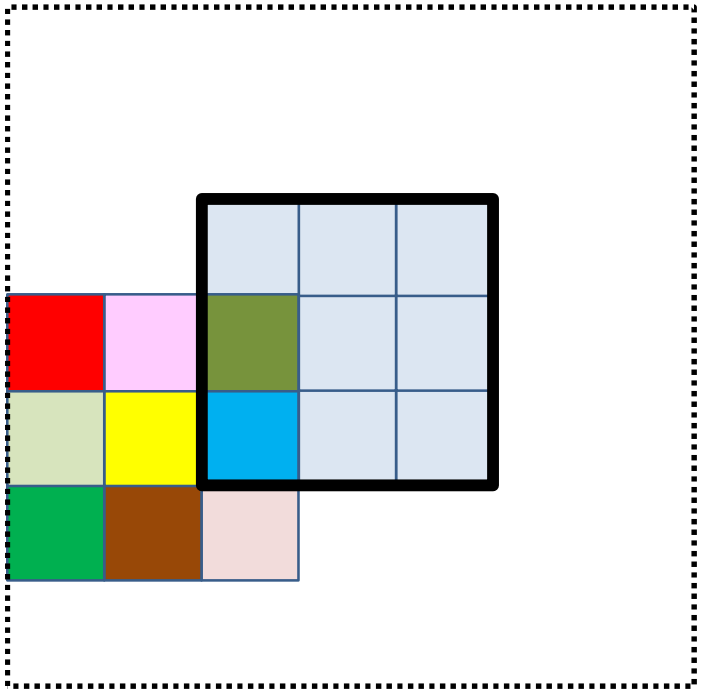
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

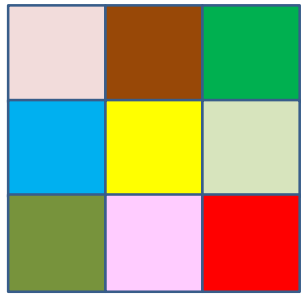
=

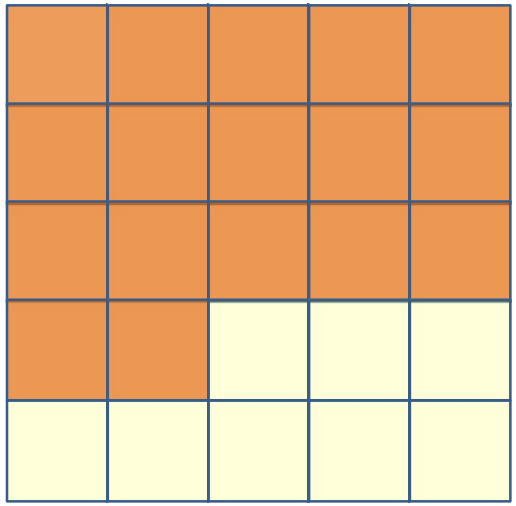


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

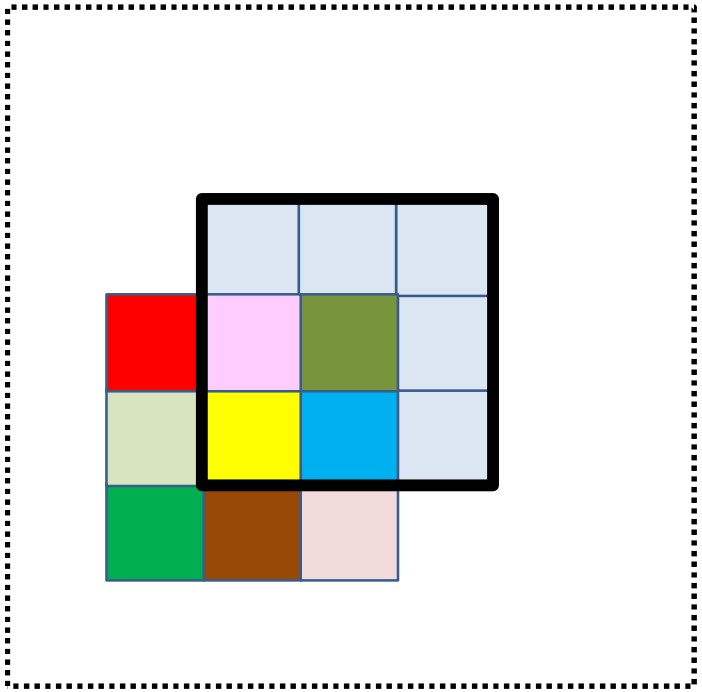
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

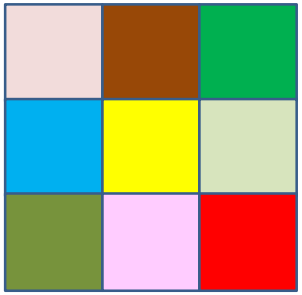
=

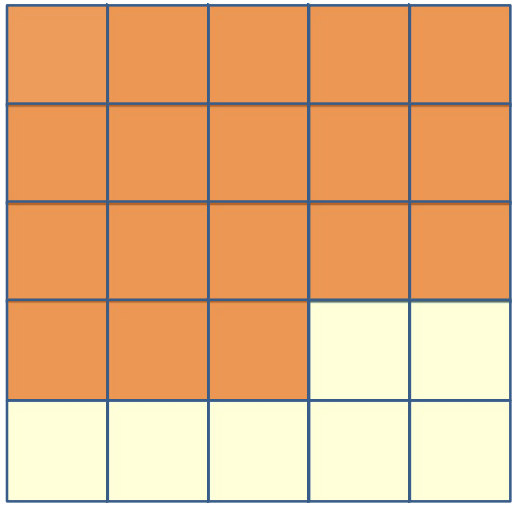


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

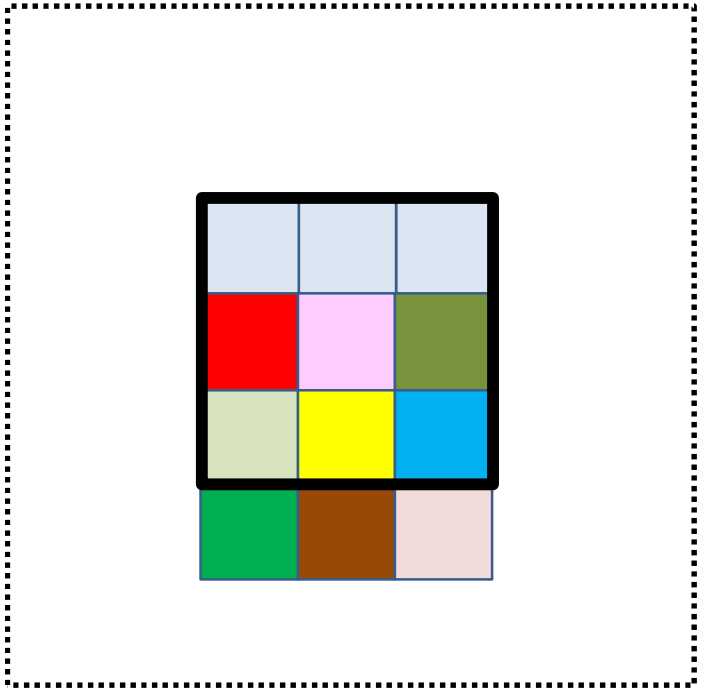
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

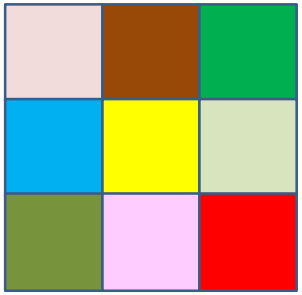
=

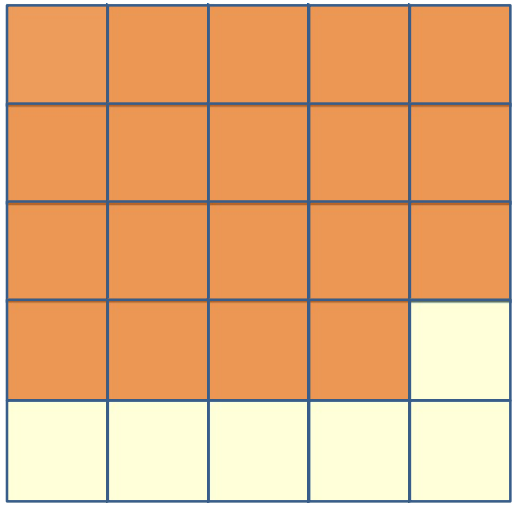


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

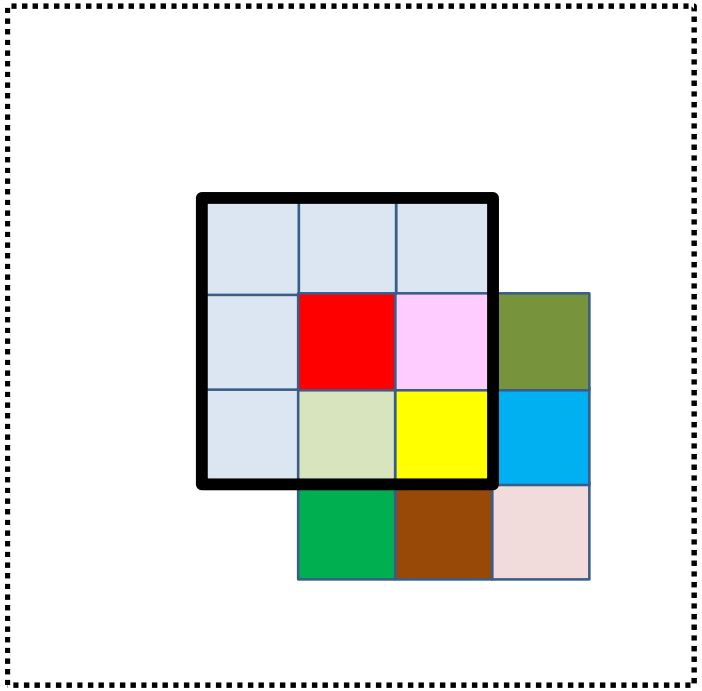
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

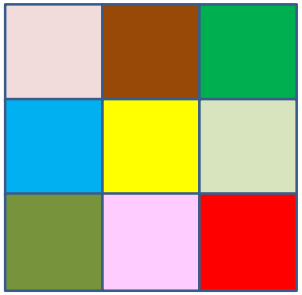
=

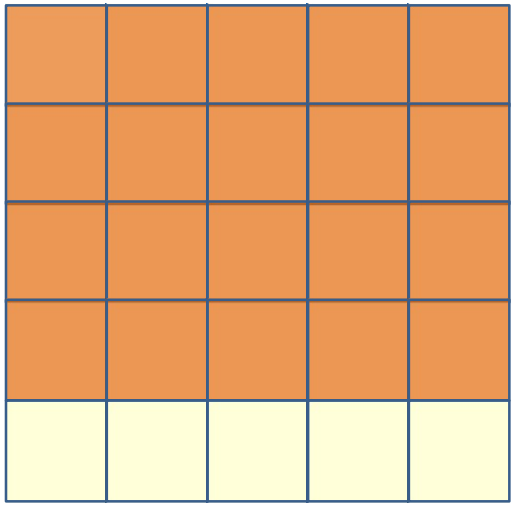


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

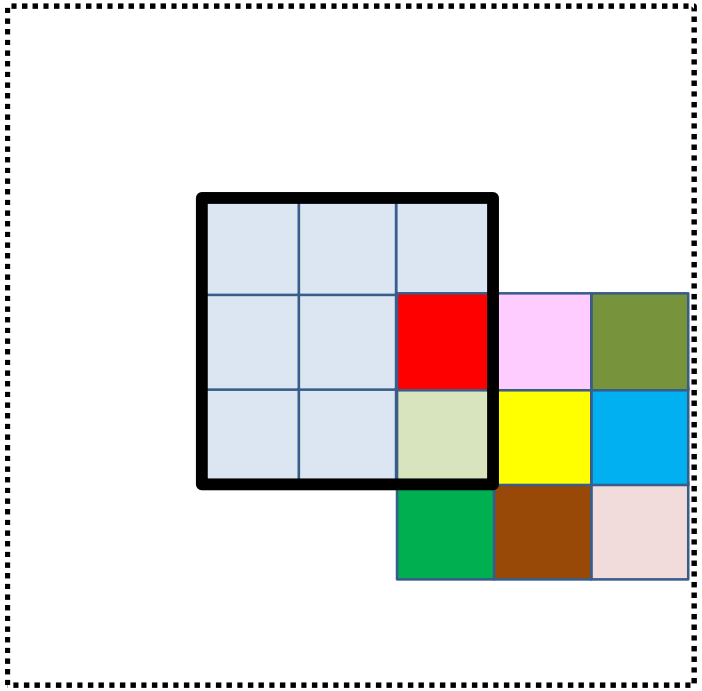
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

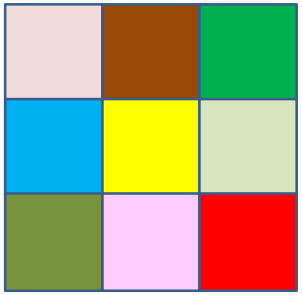
=

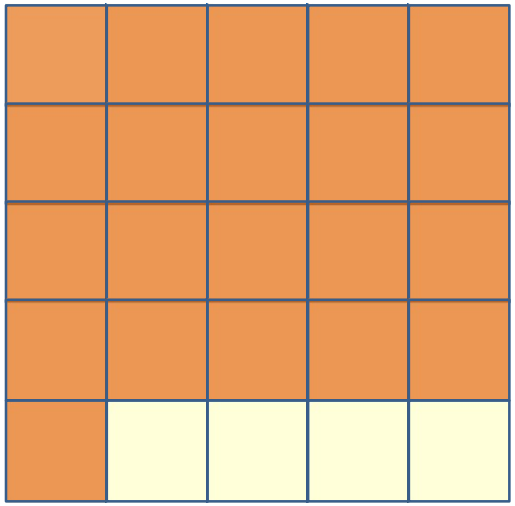


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

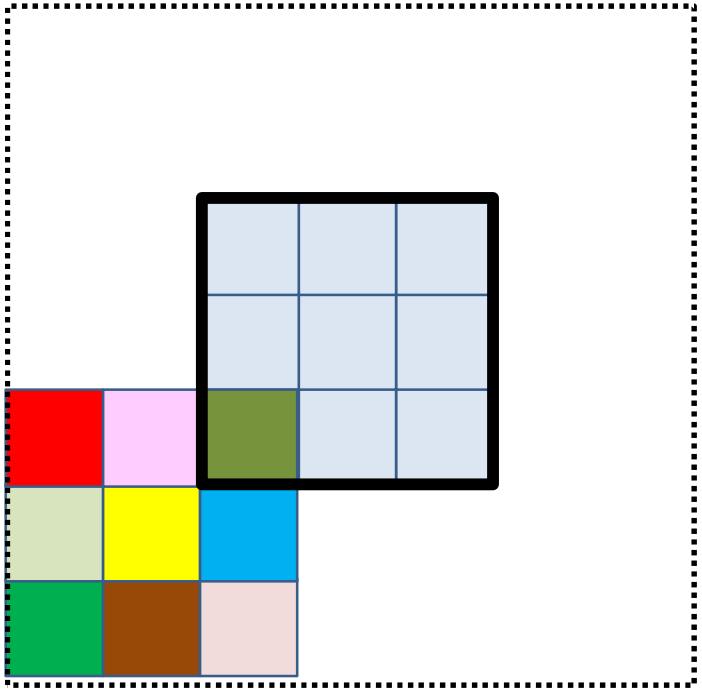
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

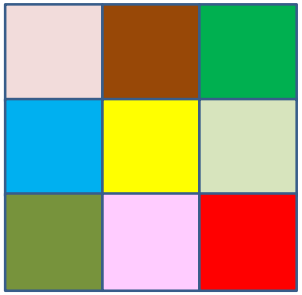
=

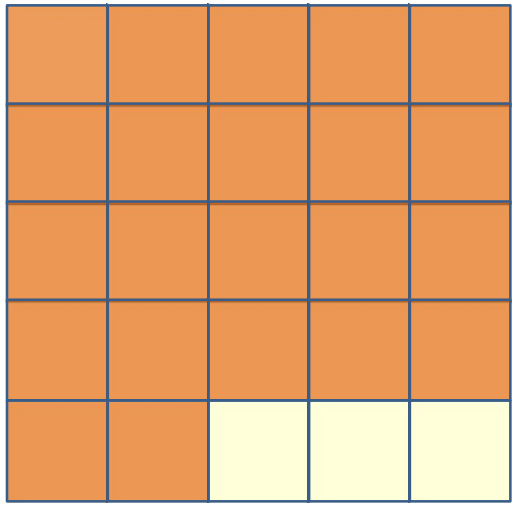


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

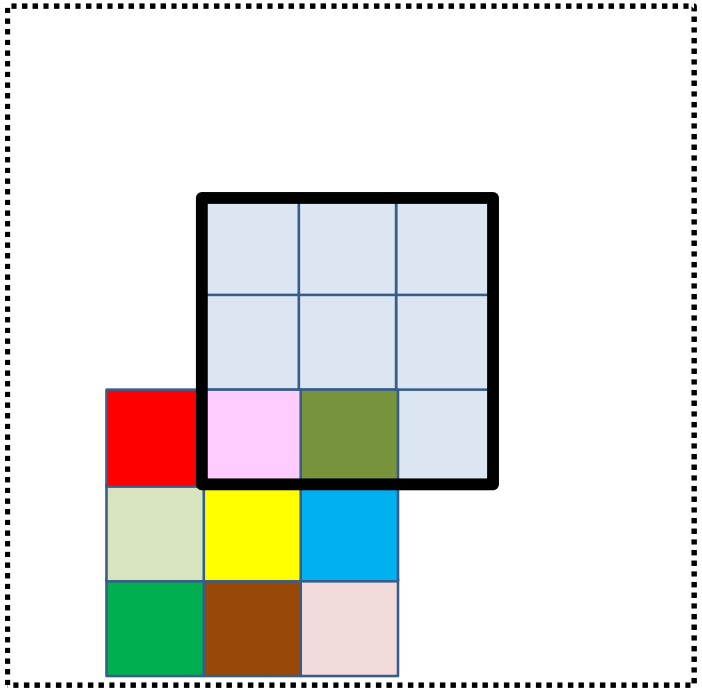
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

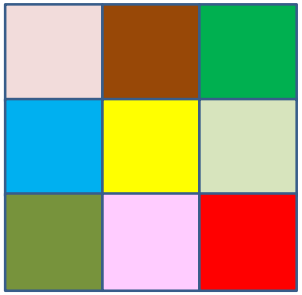
=

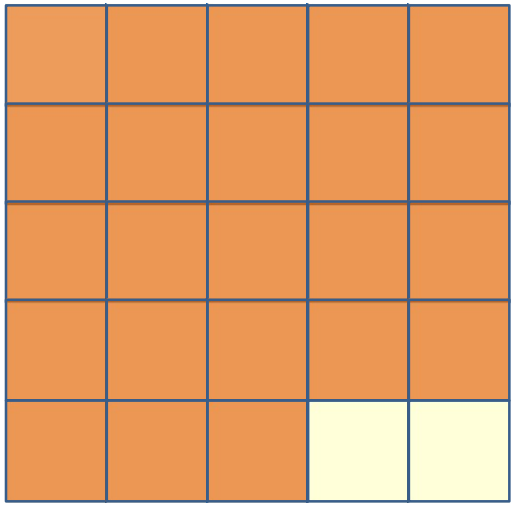


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

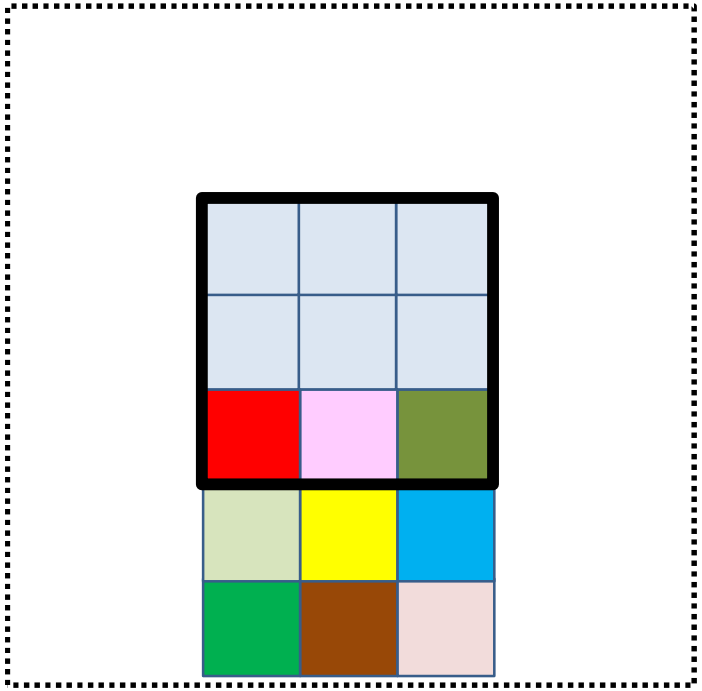
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

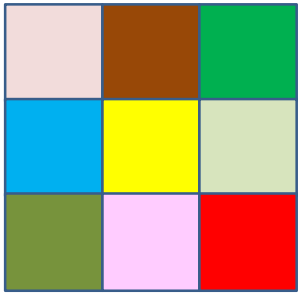
=

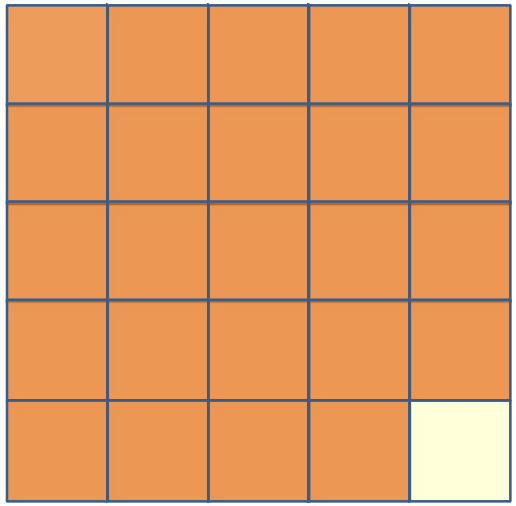


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

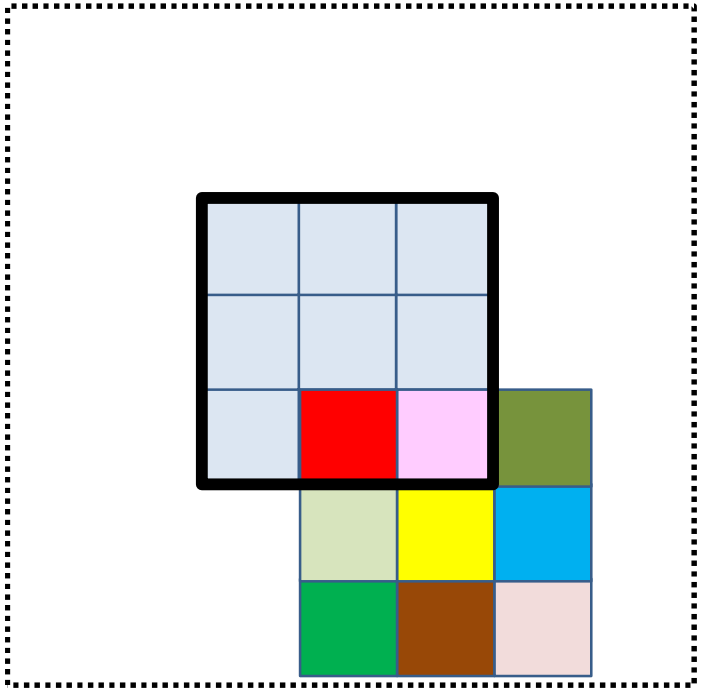
$w_l(m,n,*,*)$





$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

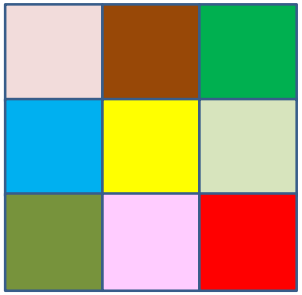
=

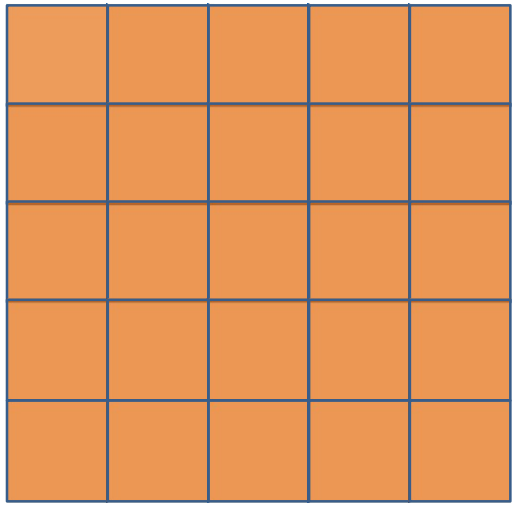


$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

Derivative at $Y(l-1,m)$ from a single $Z(l,n)$ map

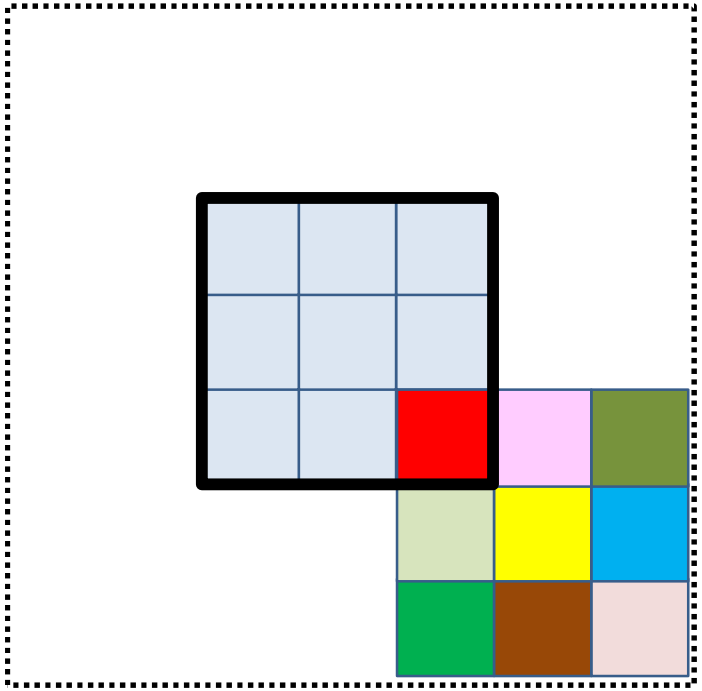
$w_l(m,n,*,*)$





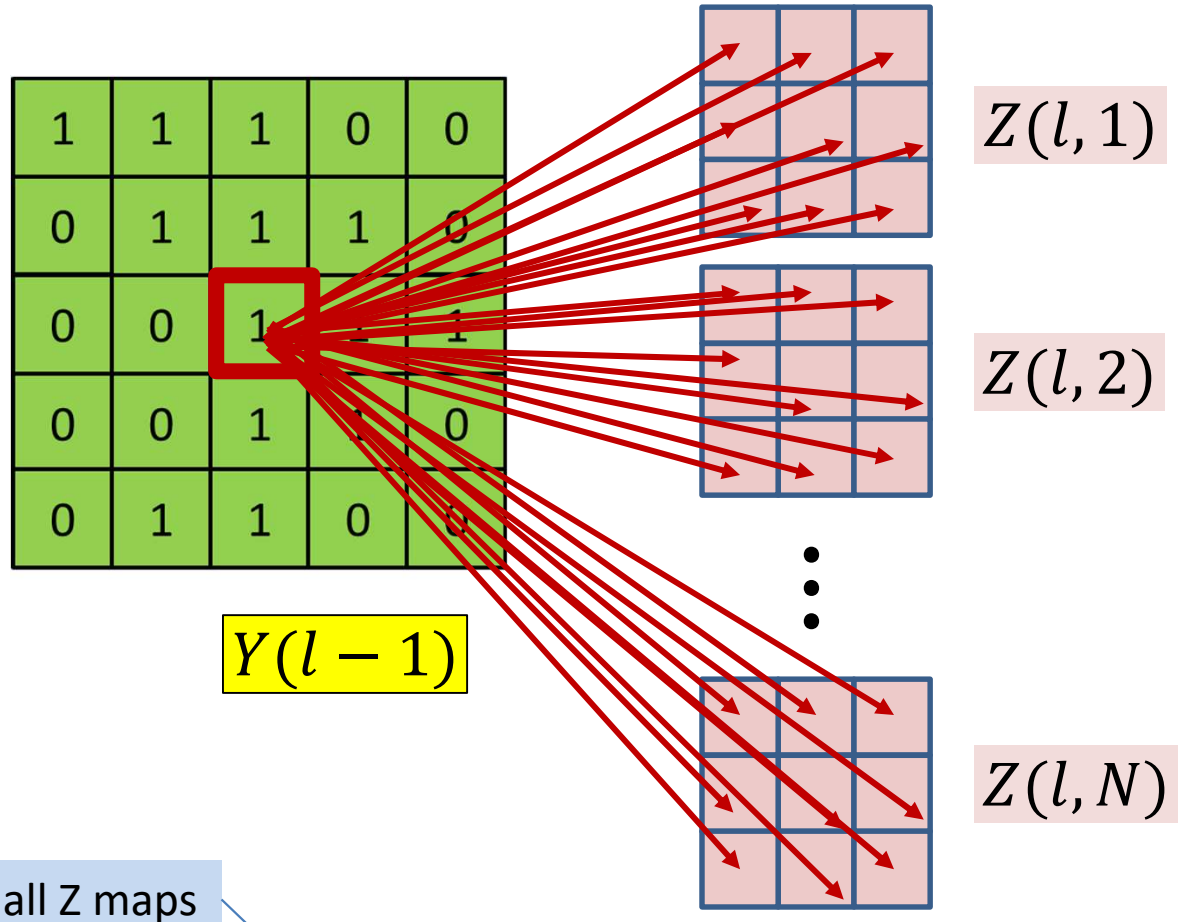
$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

=



$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

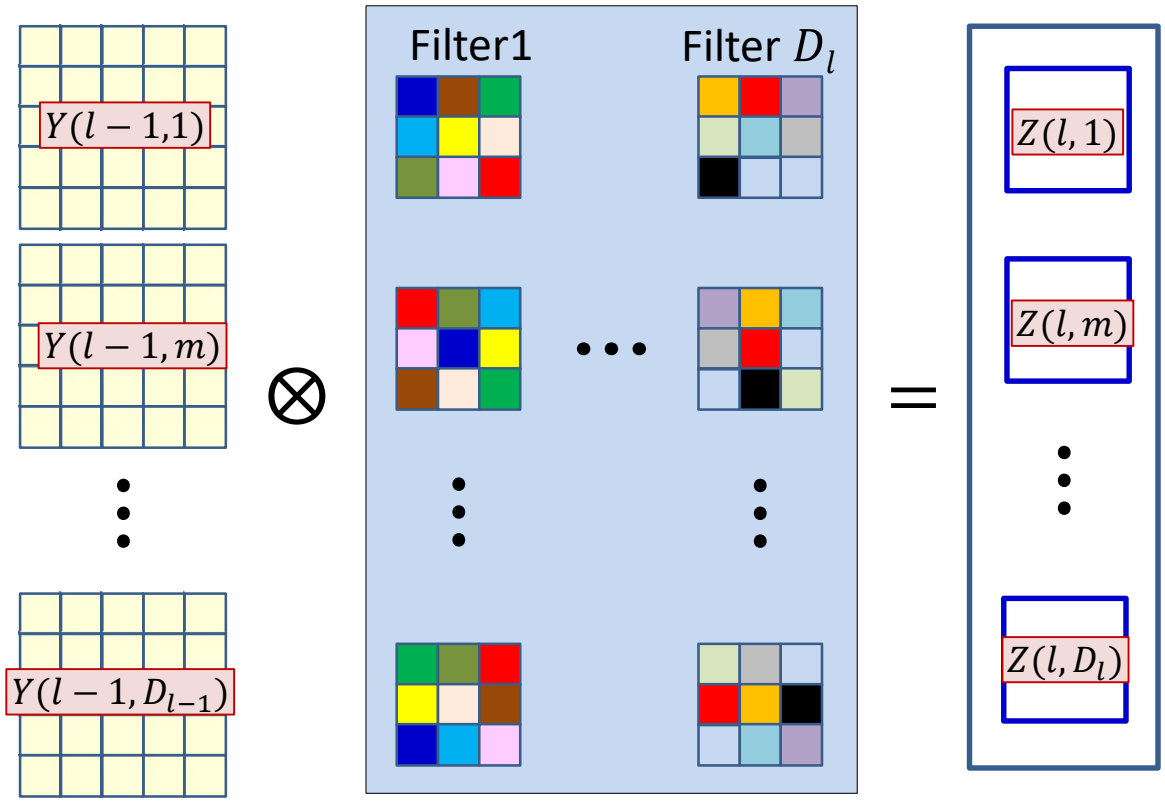
BP: Convolutional layer



Summing over all Z maps

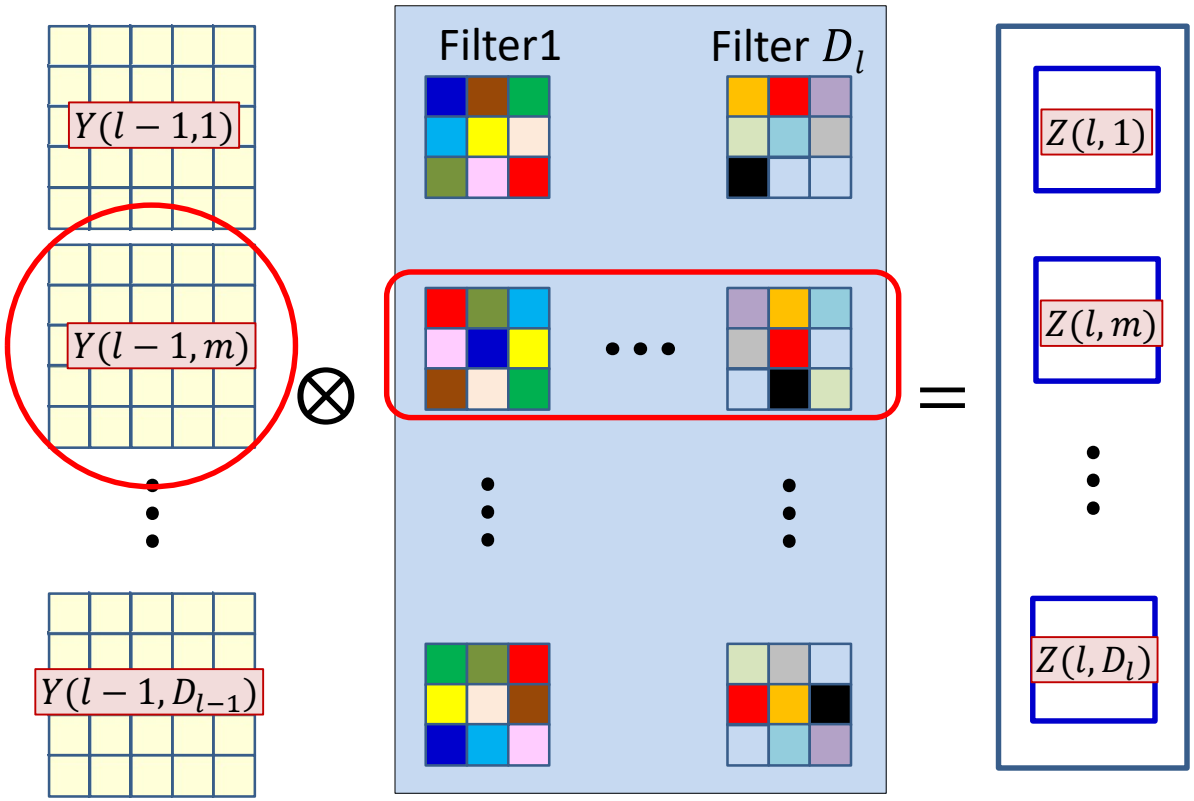
$$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_{n} \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$$

The actual convolutions



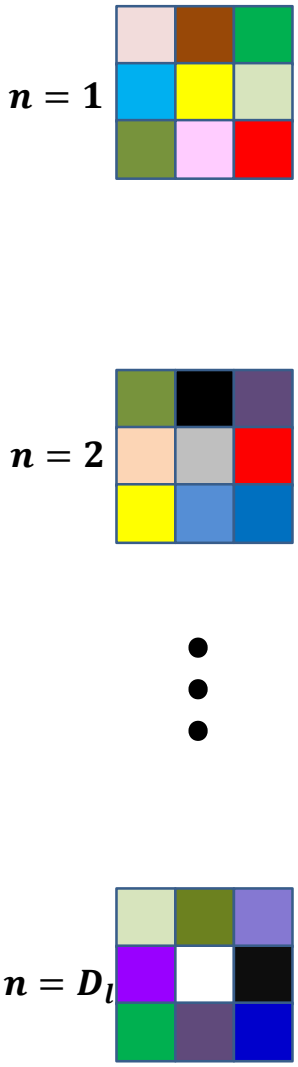
- The D_l affine maps are produced by convolving with D_l filters

The actual convolutions



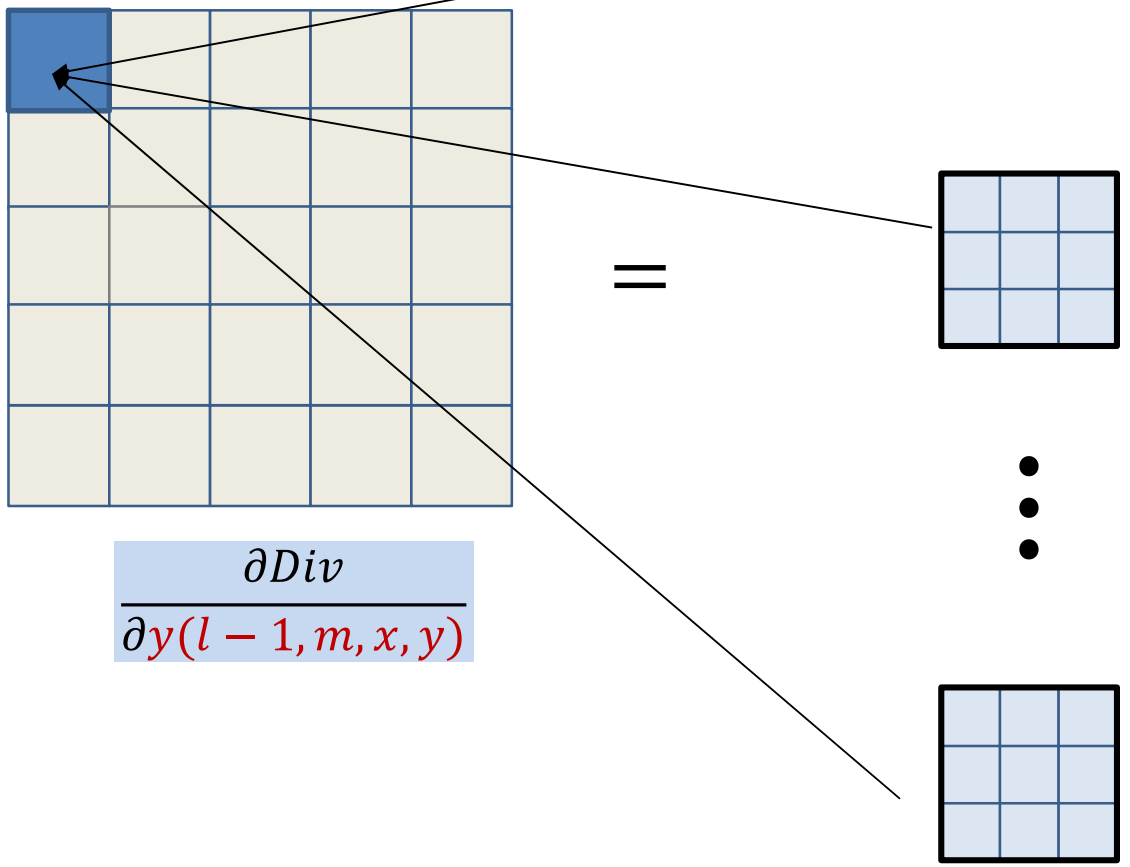
- The D_l affine maps are produced by convolving with D_l filters
- The m^{th} Y map always convolves the m^{th} plane of the filters
- The derivative for the m^{th} Y map will invoke the m^{th} plane of *all* the filters

$w_l(m,n,x,y)$

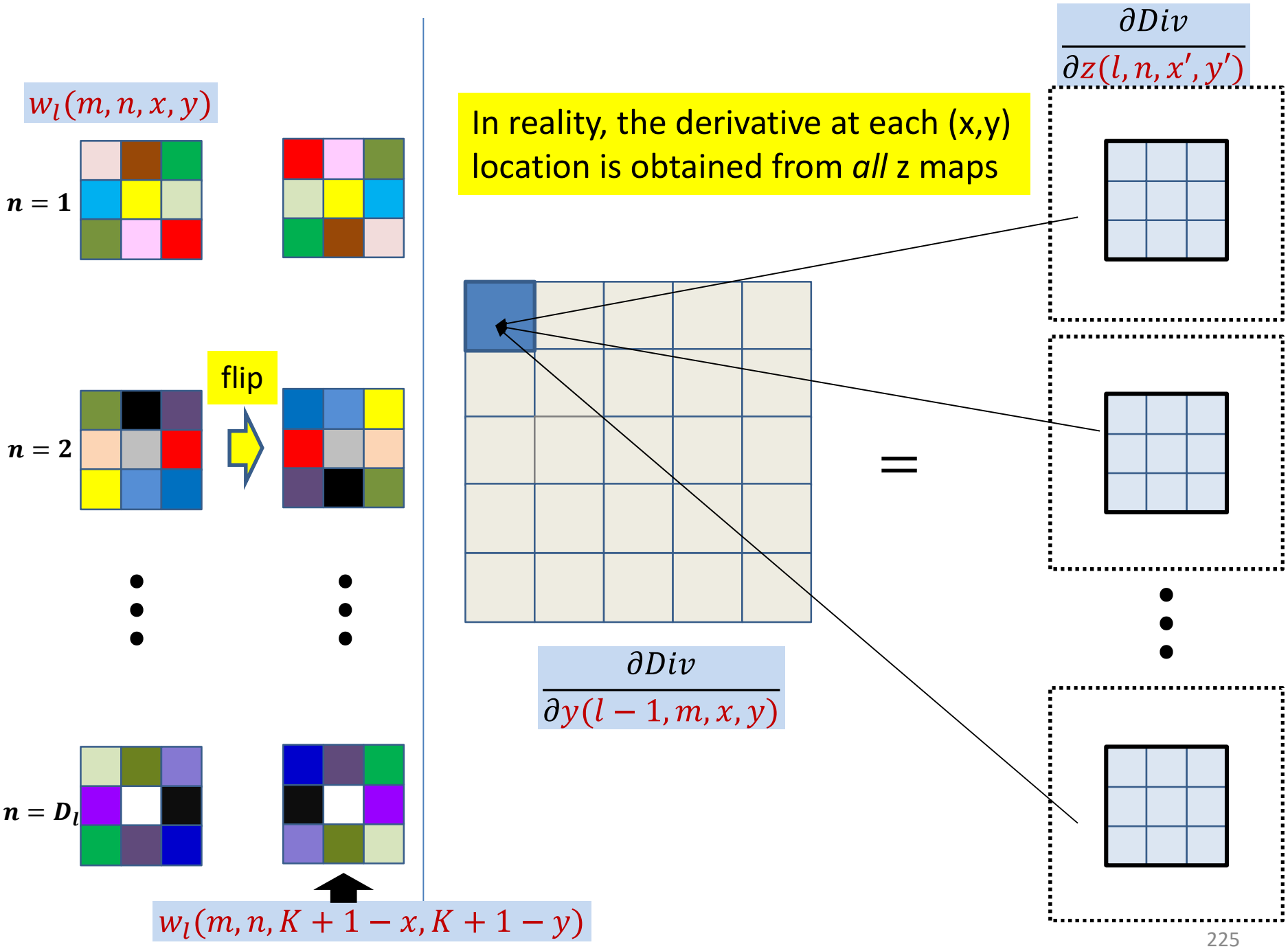


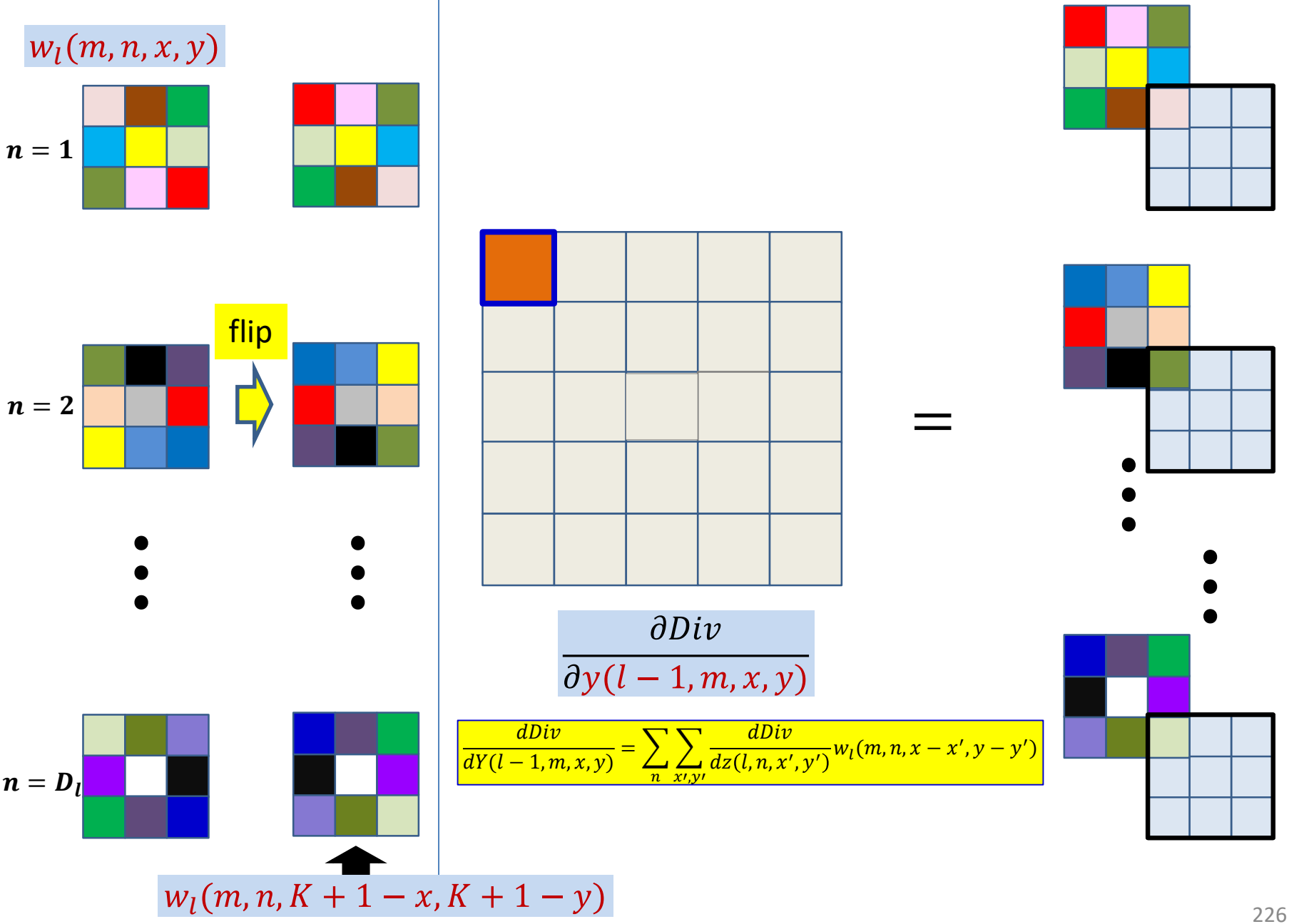
In reality, the derivative at each (x,y) location is obtained from *all* z maps

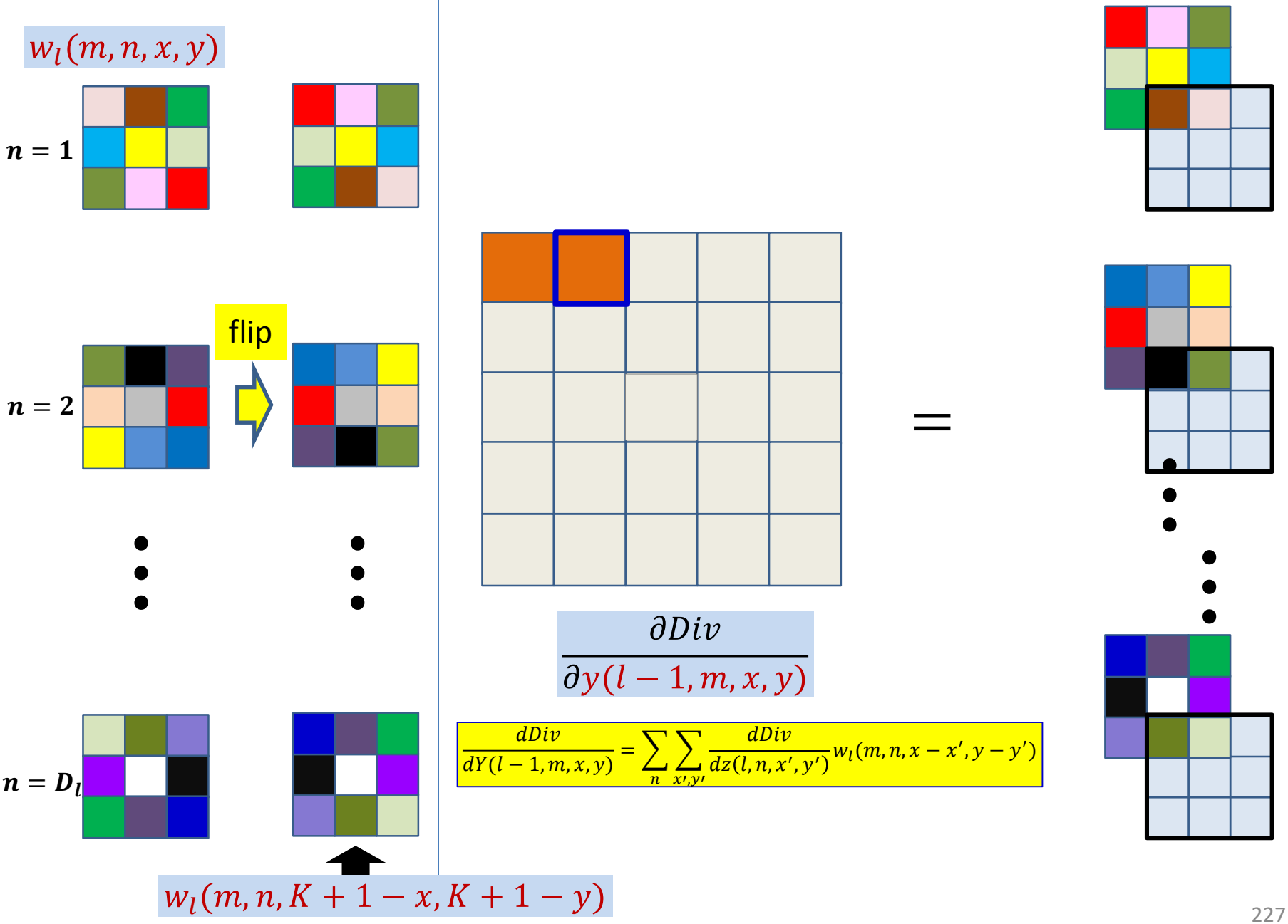
$$\frac{\partial Div}{\partial z(l,n,x',y')}$$

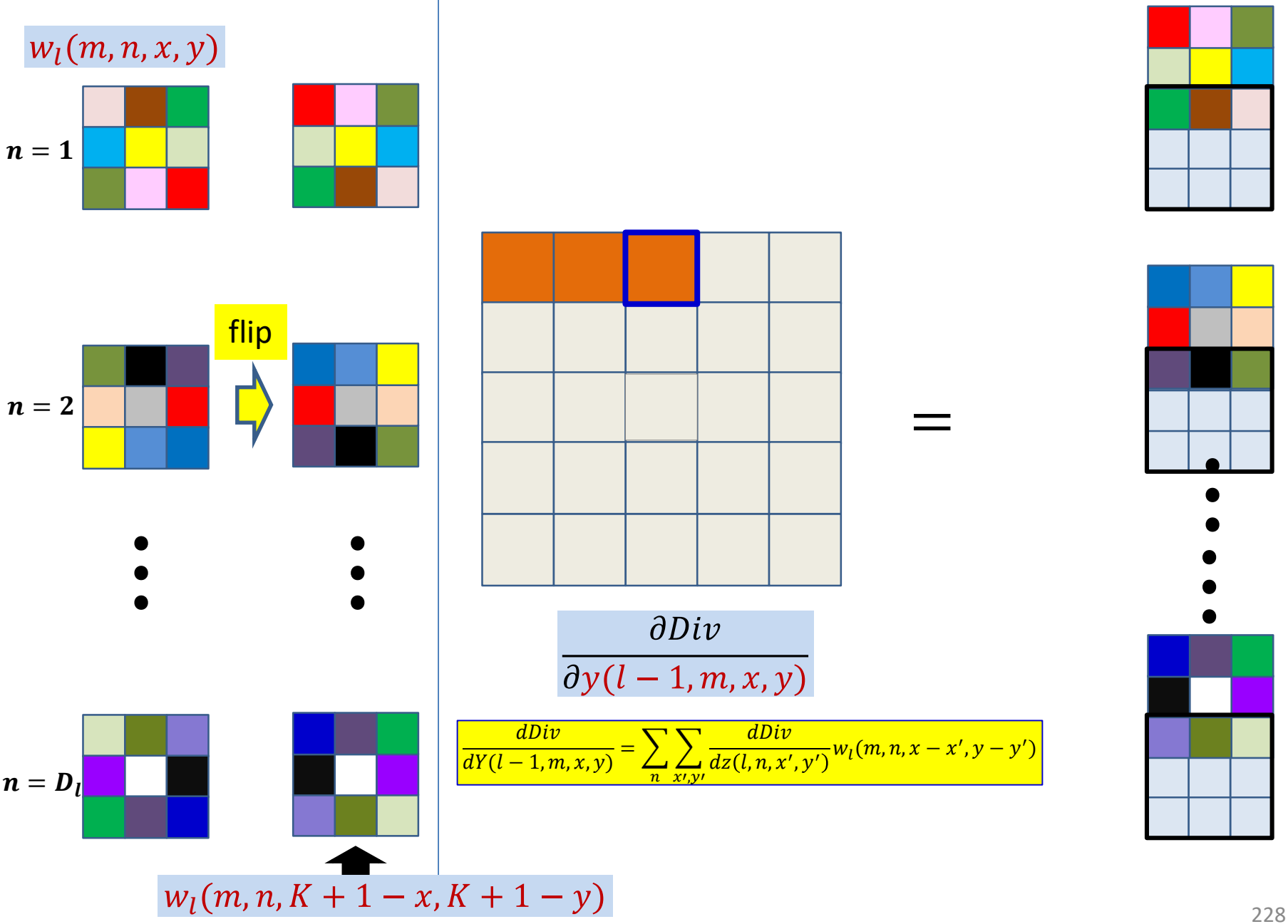


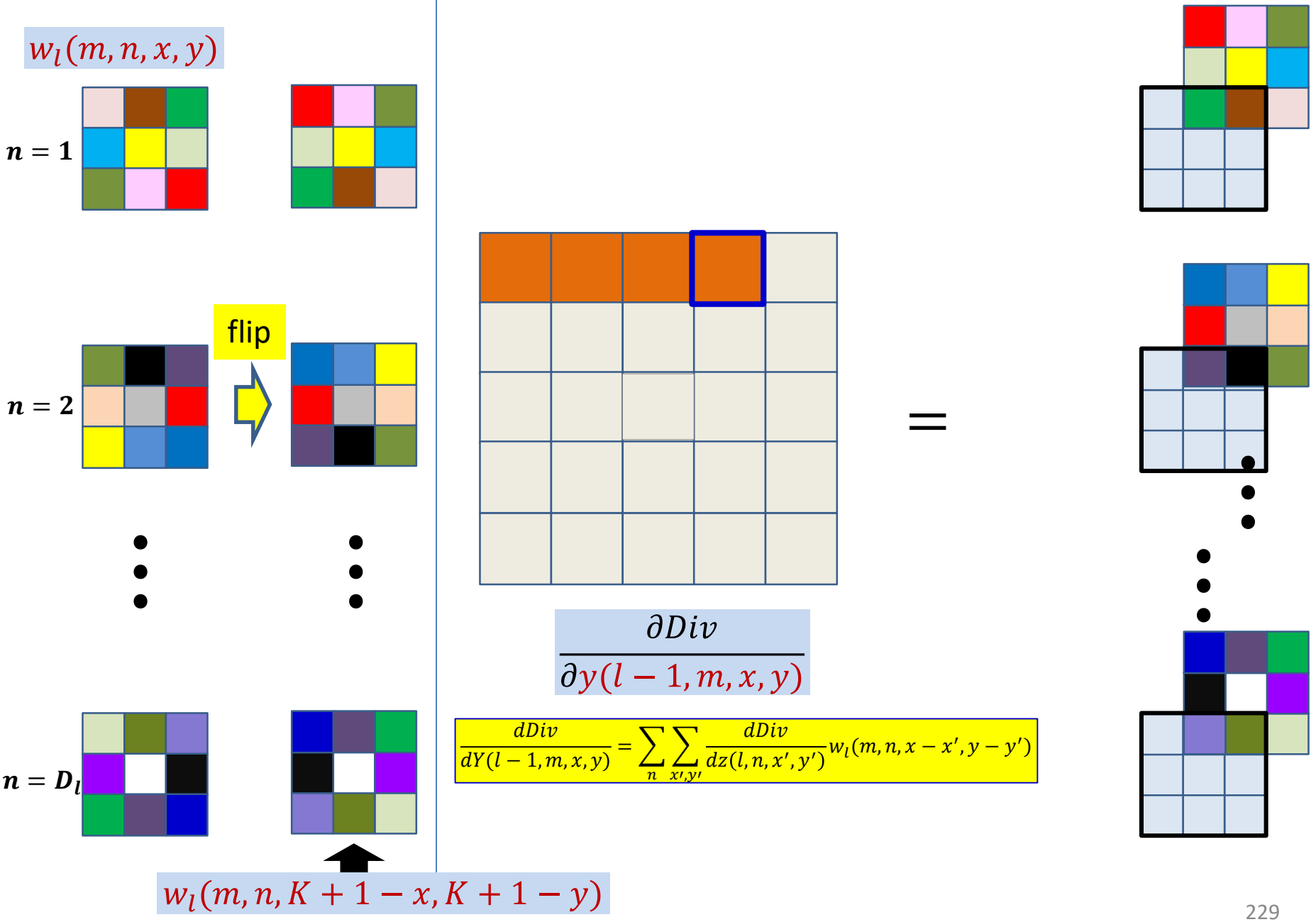
$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$


$w_l(m,n,x,y)$
$n = 1$
$n = 2$
flip
$n = D_l$
$w_l(m,n,K+1-x,K+1-y)$
In reality, the derivative at each (x,y) location is obtained from *all* z maps
$\frac{\partial Div}{\partial y(l-1,m,x,y)}$
=
$\frac{\partial Div}{\partial z(l,n,x',y')}$


$w_l(m,n,x,y)$
$n = 1$
$n = 2$
flip
$n = D_l$
$w_l(m,n,K+1-x,K+1-y)$
$\frac{\partial Div}{\partial y(l-1,m,x,y)}$
$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$


$w_l(m,n,x,y)$
$n = 1$
$n = 2$
flip
$n = D_l$
$w_l(m,n,K+1-x,K+1-y)$
=
$\frac{\partial Div}{\partial y(l-1,m,x,y)}$
$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$


$w_l(m,n,x,y)$
$n = 1$
$n = 2$
flip
$n = D_l$
$w_l(m,n,K+1-x,K+1-y)$
$\frac{\partial Div}{\partial y(l-1,m,x,y)}$
$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$


$w_l(m,n,x,y)$
$n = 1$
$n = 2$
flip
$n = D_l$
$w_l(m,n,K+1-x,K+1-y)$
=
$\frac{\partial Div}{\partial y(l-1,m,x,y)}$
$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$

$w_l(m,n,x,y)$

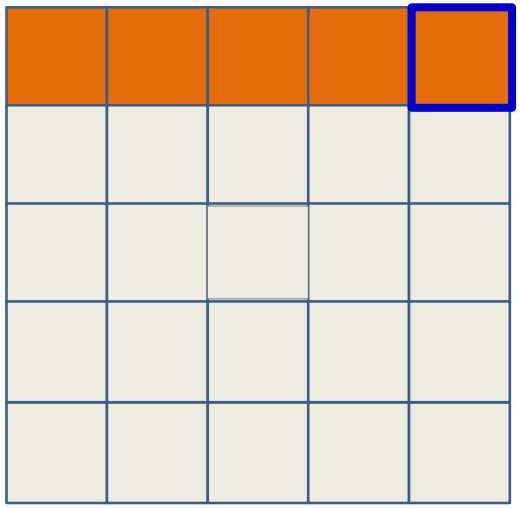
$\boldsymbol{n=1}$

$\boldsymbol{n=2}$

flip

$\boldsymbol{n=D_l}$

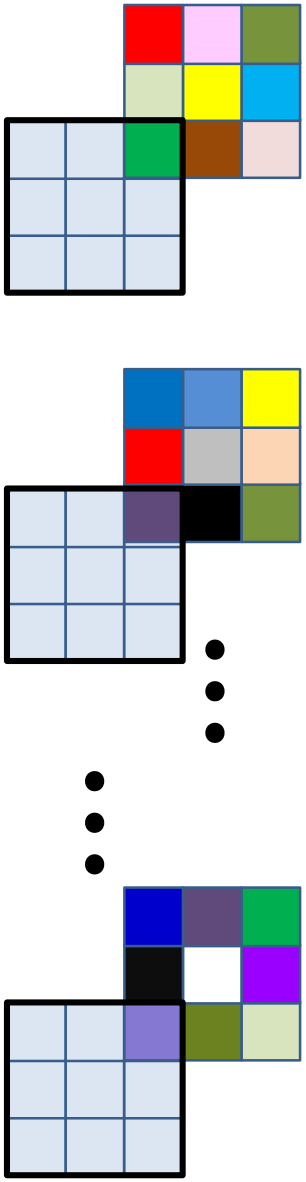
$w_l(m,n,K+1-x,K+1-y)$

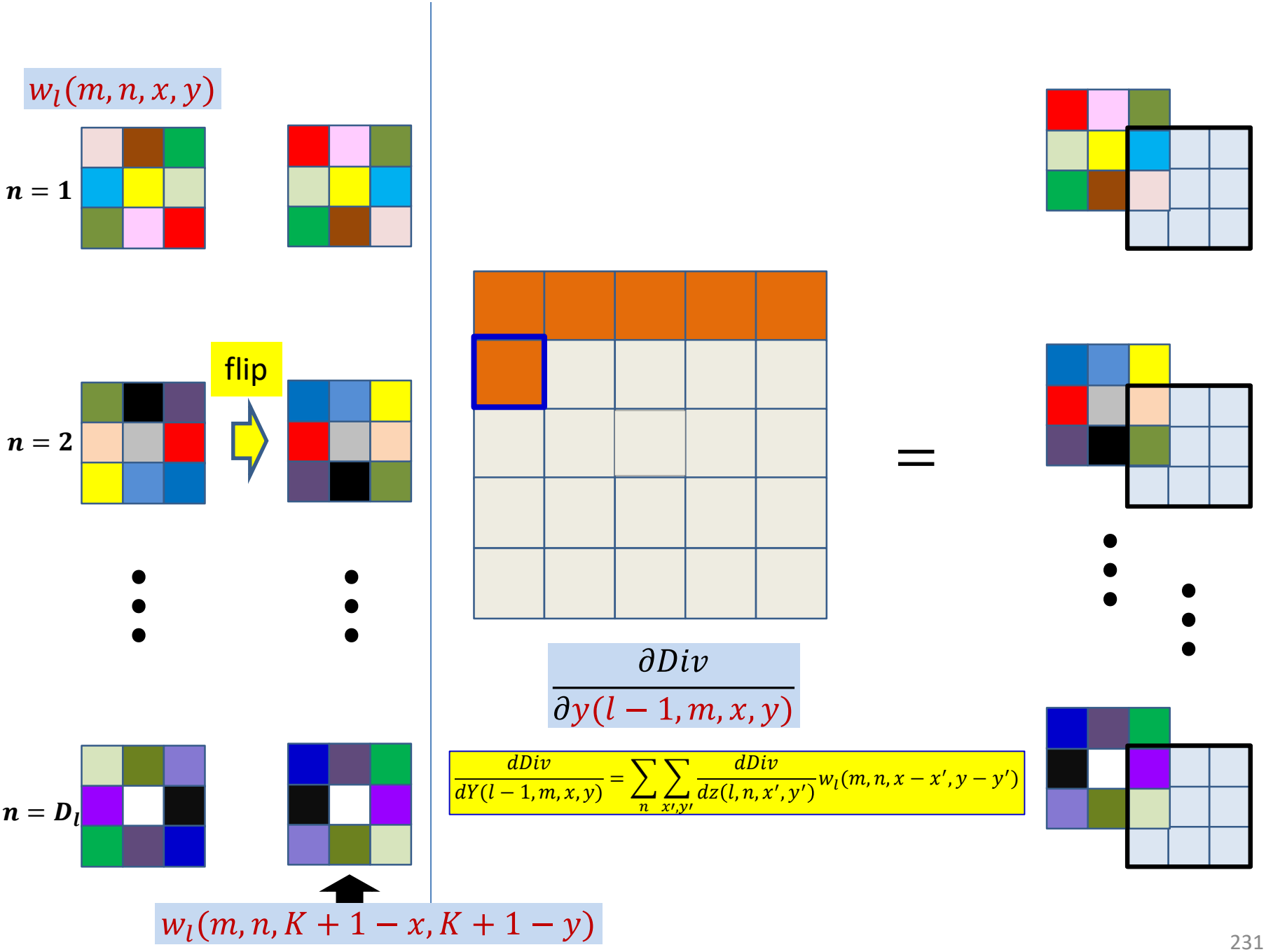


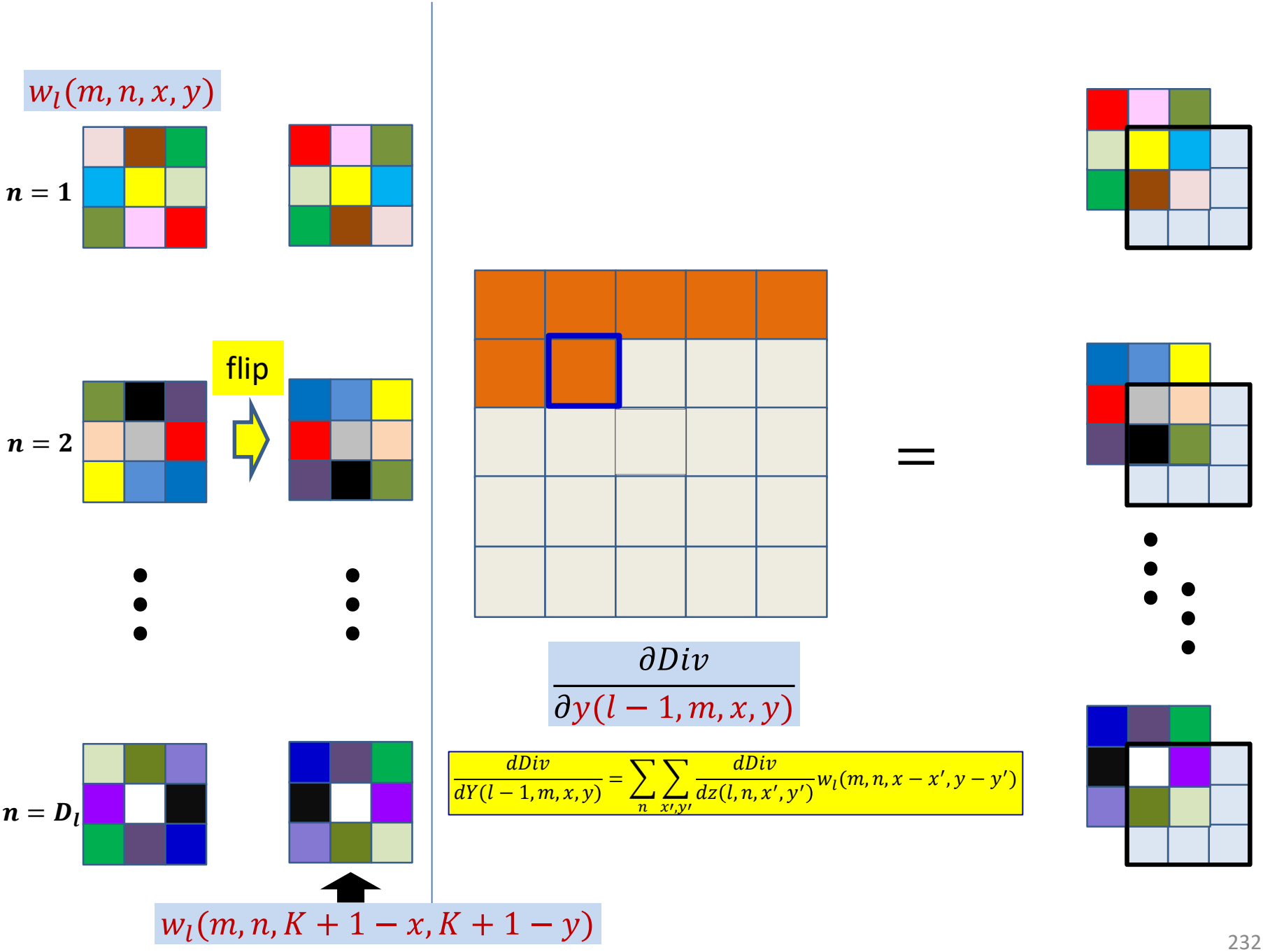
$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

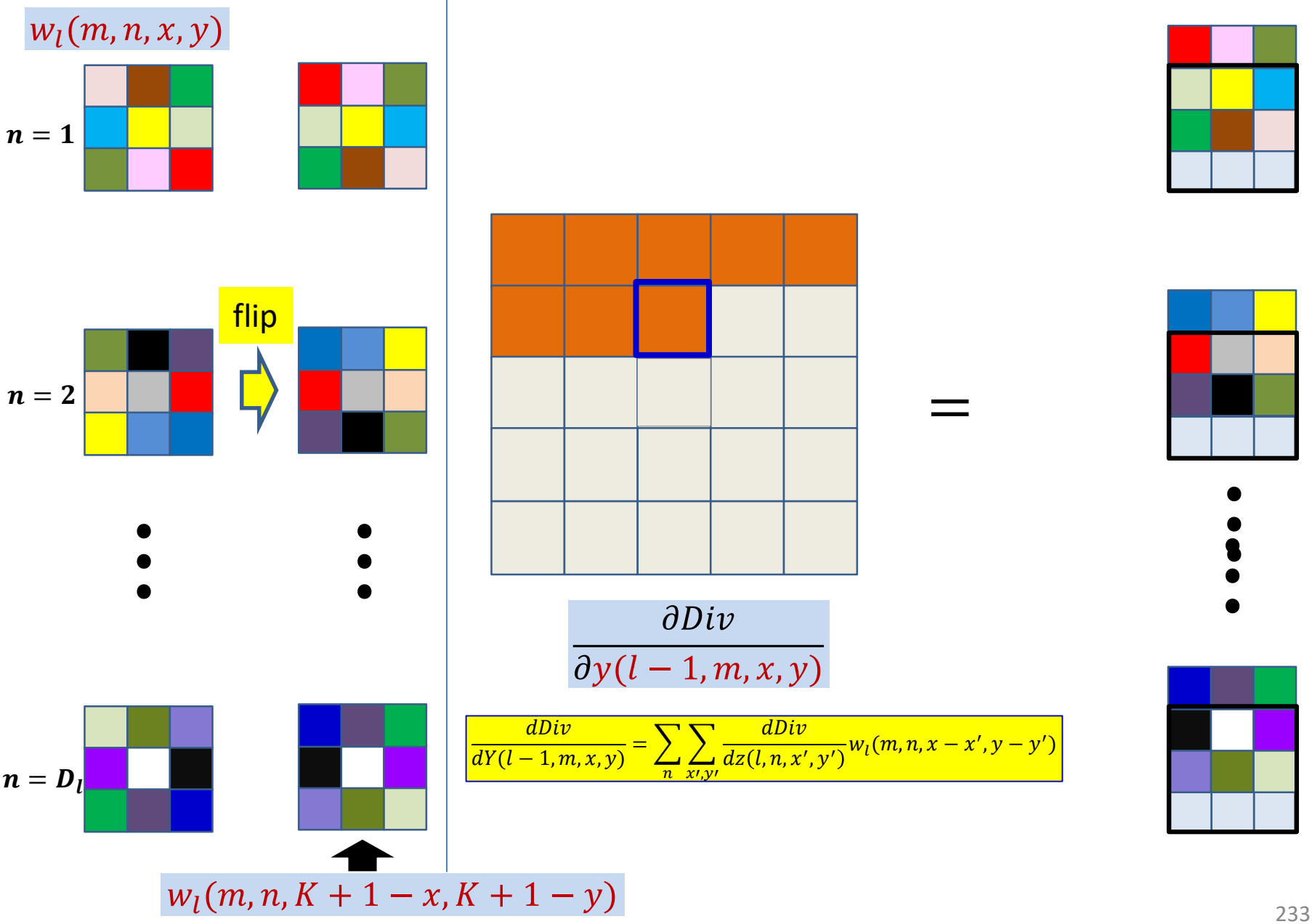
$$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$$

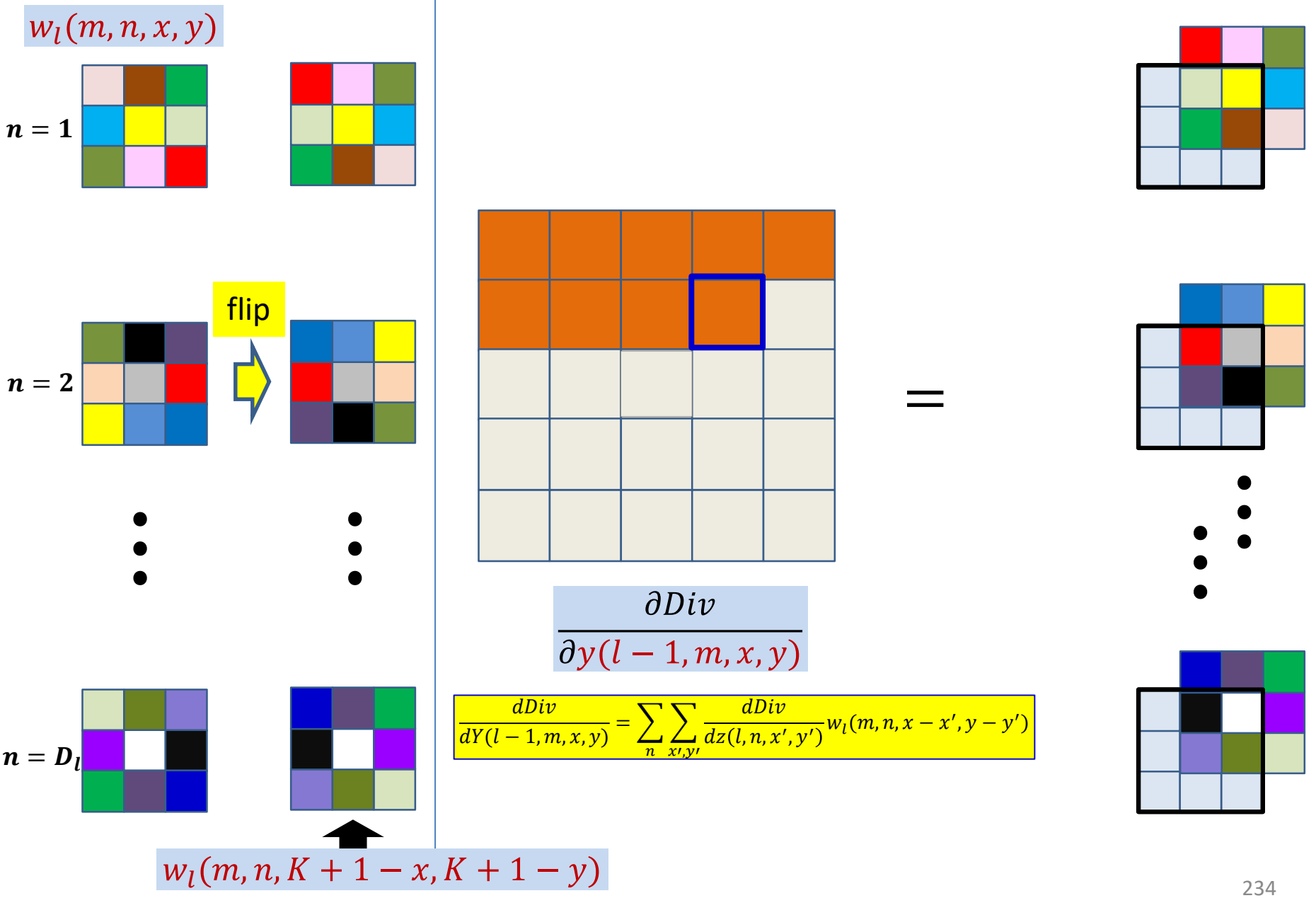
=

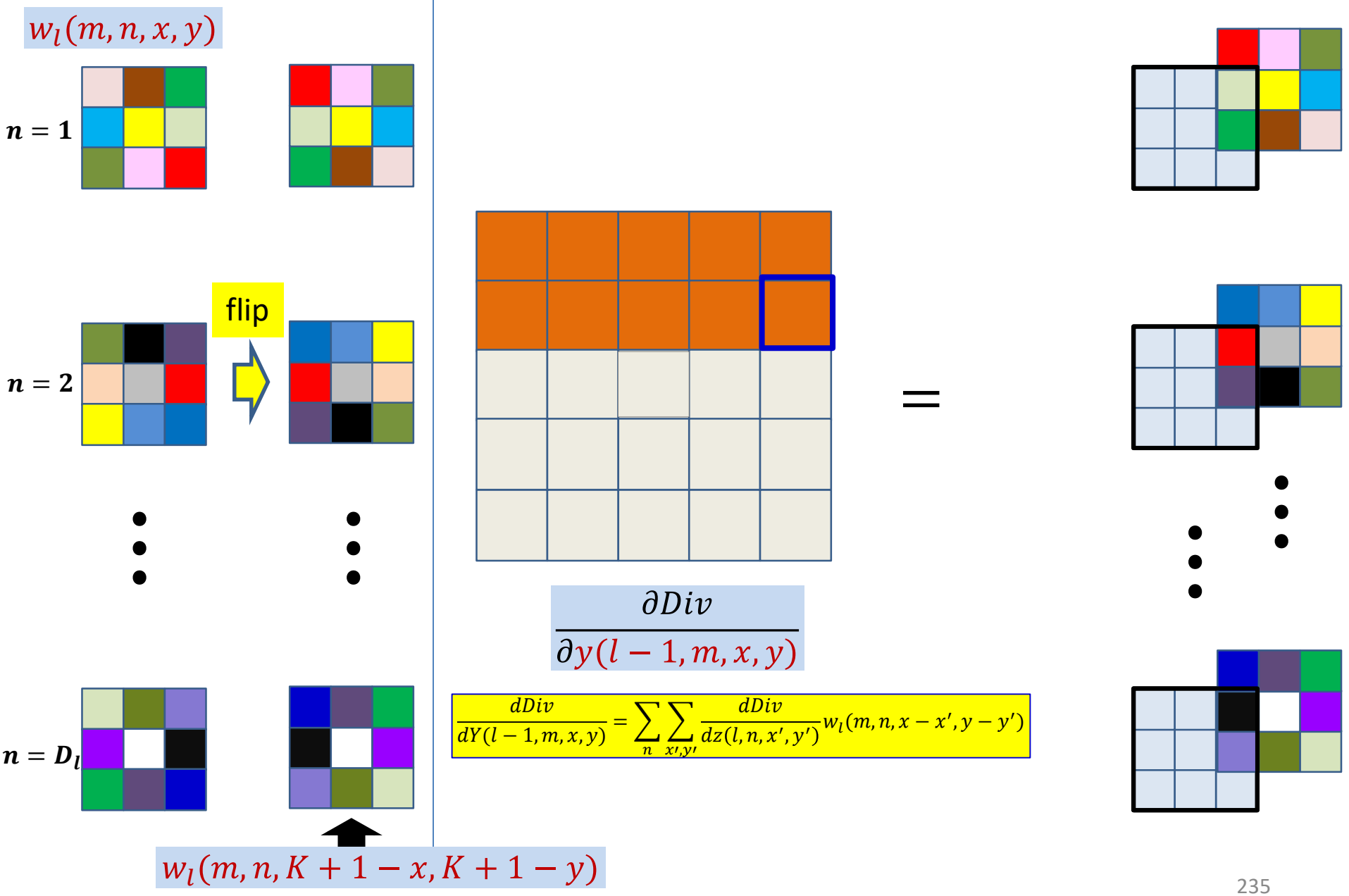


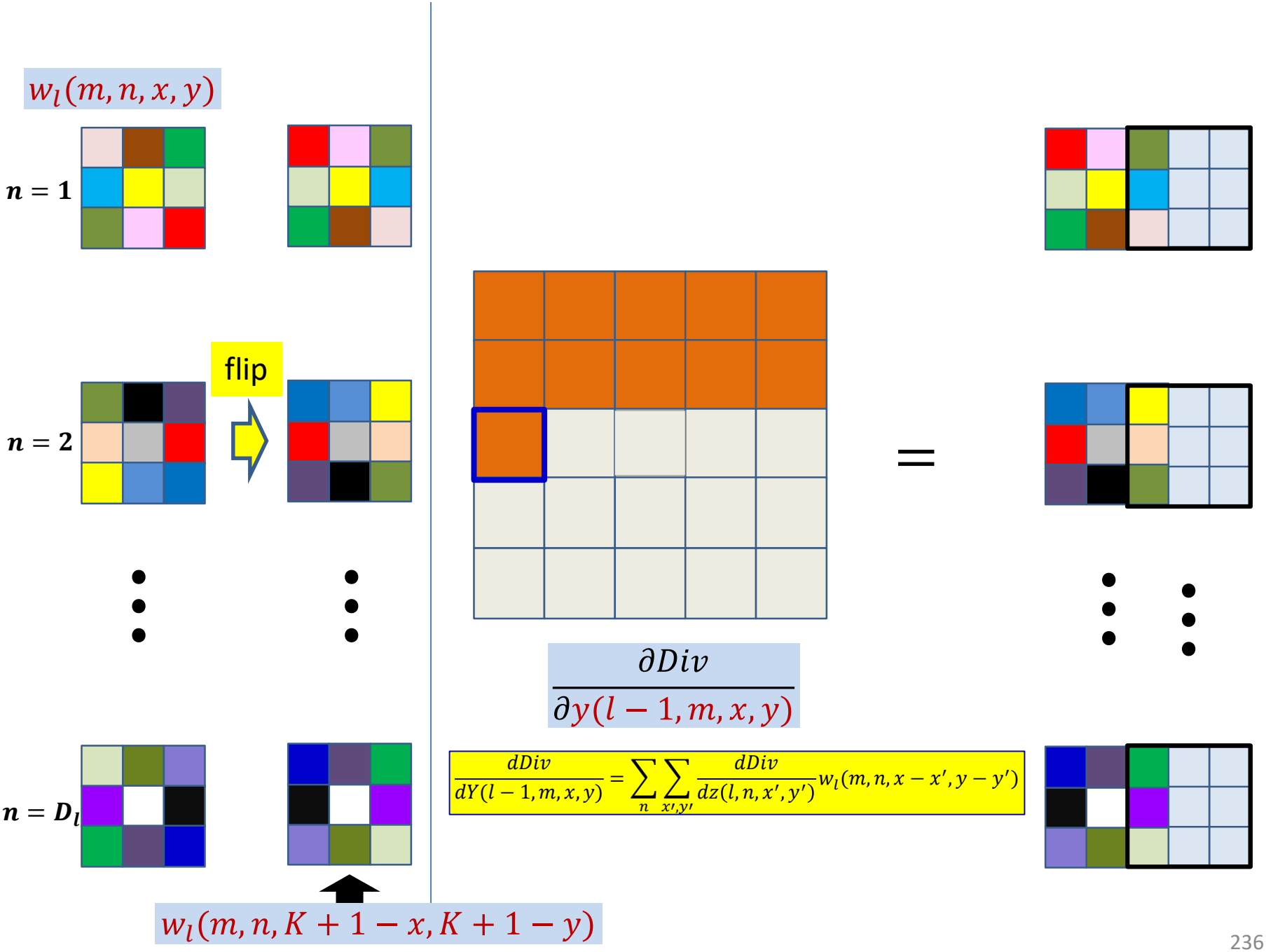

$w_l(m,n,x,y)$
$n = 1$
$n = 2$
flip
$n = D_l$
$w_l(m,n,K+1-x,K+1-y)$
=
$\frac{\partial Div}{\partial y(l-1,m,x,y)}$
$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$

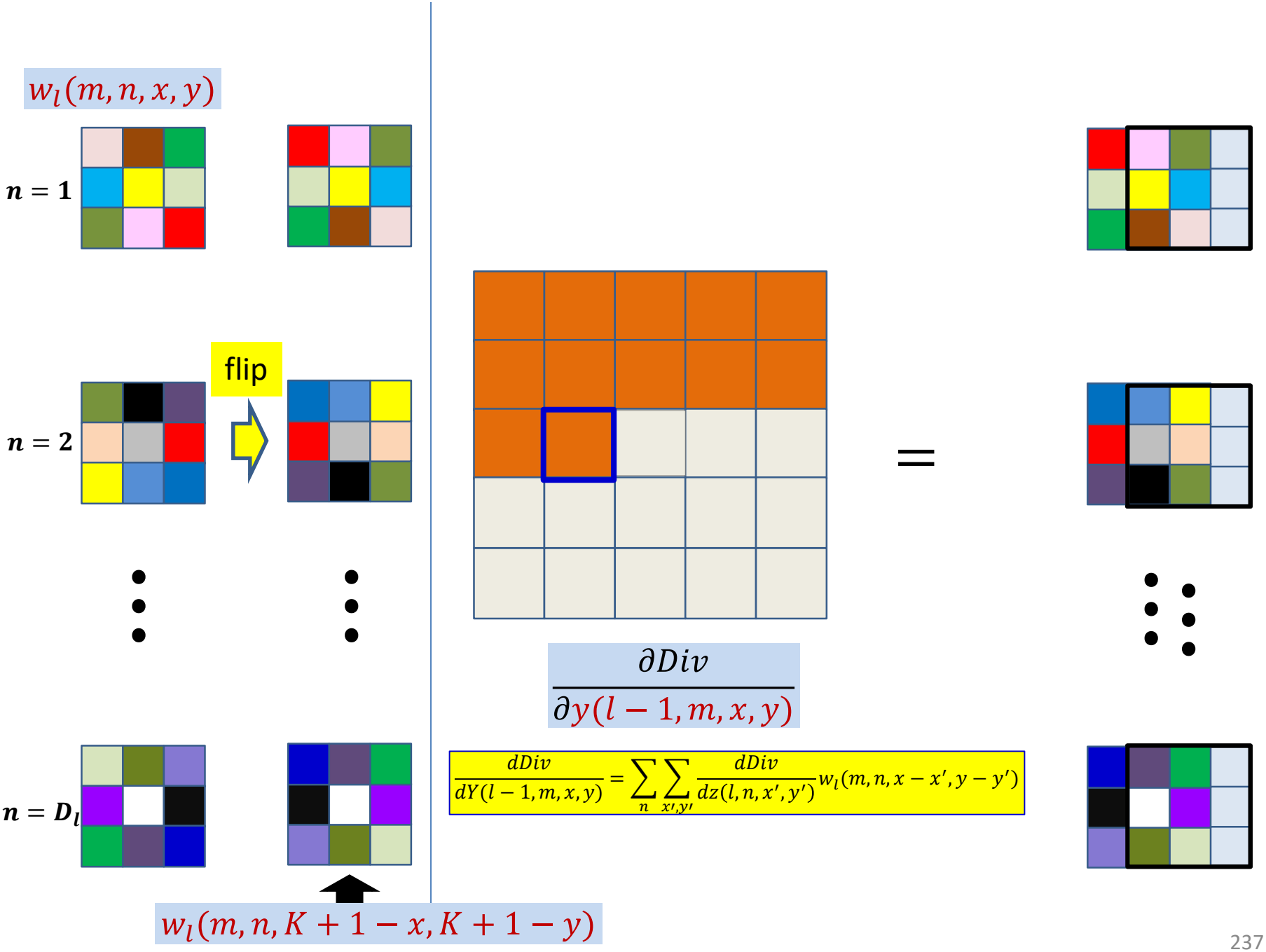

$w_l(m,n,x,y)$
$n = 1$
$n = 2$
flip
$n = D_l$
$w_l(m,n,K+1-x,K+1-y)$
$\frac{\partial Div}{\partial y(l-1,m,x,y)}$
$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$

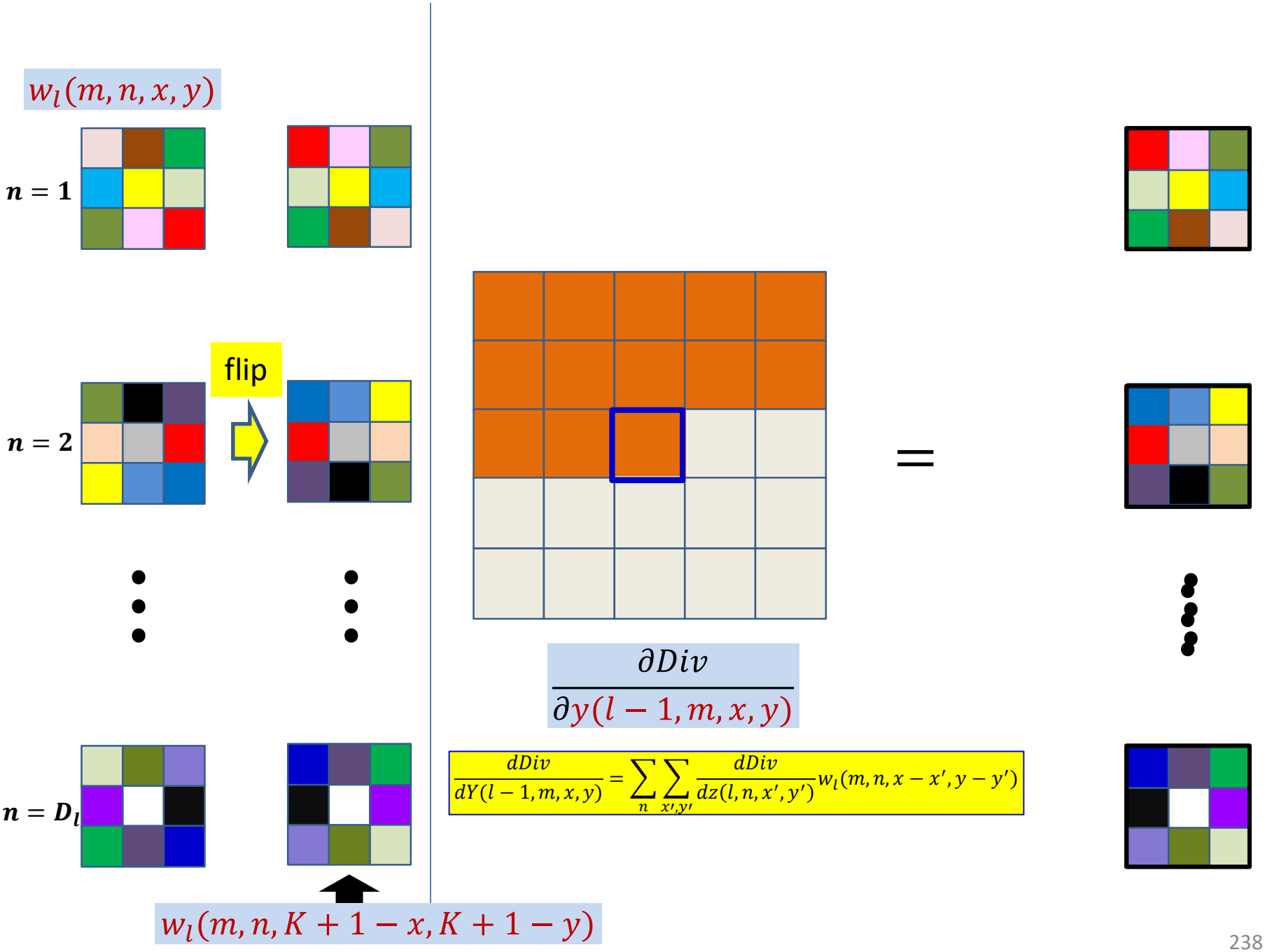

$w_l(m,n,x,y)$
$n = 1$
$n = 2$
flip
$n = D_l$
$w_l(m,n,K+1-x,K+1-y)$
=
$\frac{\partial Div}{\partial y(l-1,m,x,y)}$
$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$


$w_l(m,n,x,y)$
$n = 1$
$n = 2$
flip
$n = D_l$
$w_l(m,n,K+1-x,K+1-y)$
=
$\frac{\partial Div}{\partial y(l-1,m,x,y)}$
$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$


$w_l(m,n,x,y)$
$n = 1$
$n = 2$
flip
$n = D_l$
$w_l(m,n,K+1-x,K+1-y)$
=
$\frac{\partial Div}{\partial y(l-1,m,x,y)}$
$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$


$w_l(m,n,x,y)$
$n = 1$
$n = 2$
flip
$n = D_l$
=
$\frac{\partial Div}{\partial y(l-1,m,x,y)}$
$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$
$w_l(m,n,K+1-x,K+1-y)$


$w_l(m,n,x,y)$
$n = 1$
$n = 2$
flip
$n = D_l$
=
$\frac{\partial Div}{\partial y(l-1,m,x,y)}$
$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$
$w_l(m,n,K+1-x,K+1-y)$


$w_l(m,n,x,y)$
$n = 1$
$n = 2$
flip
$n = D_l$
$w_l(m,n,K+1-x,K+1-y)$
=
$\frac{\partial Div}{\partial y(l-1,m,x,y)}$
$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$

$w_l(m,n,x,y)$

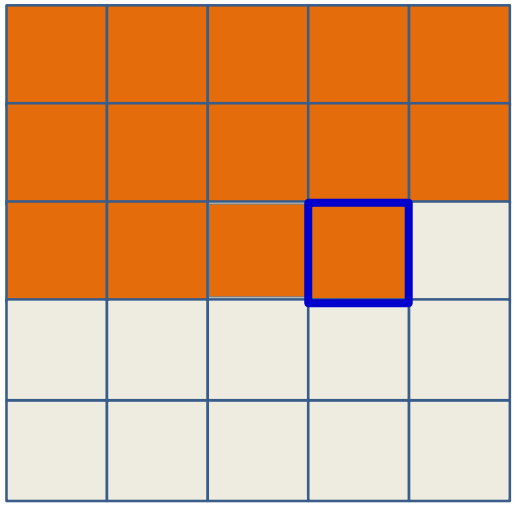
$\boldsymbol{n=1}$

$\boldsymbol{n=2}$

flip

$\boldsymbol{n=D_l}$

$w_l(m,n,K+1-x,K+1-y)$



$=$

$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

$$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$$

$w_l(m,n,x,y)$

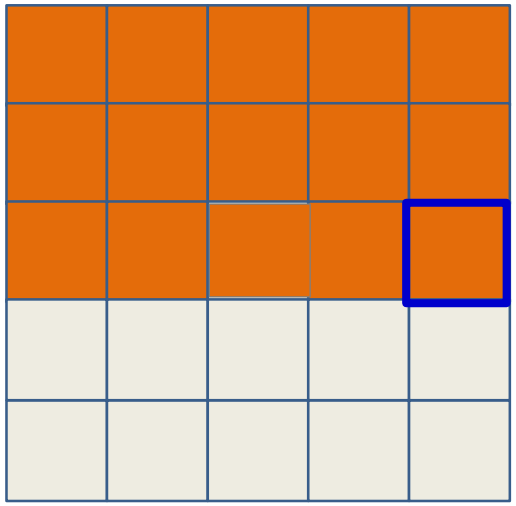
$\boldsymbol{n=1}$

$\boldsymbol{n=2}$

flip

$\boldsymbol{n=D_l}$

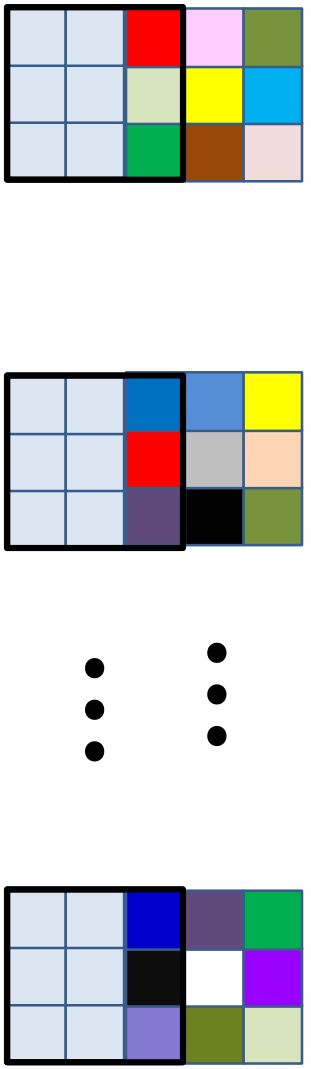
$w_l(m,n,K+1-x,K+1-y)$

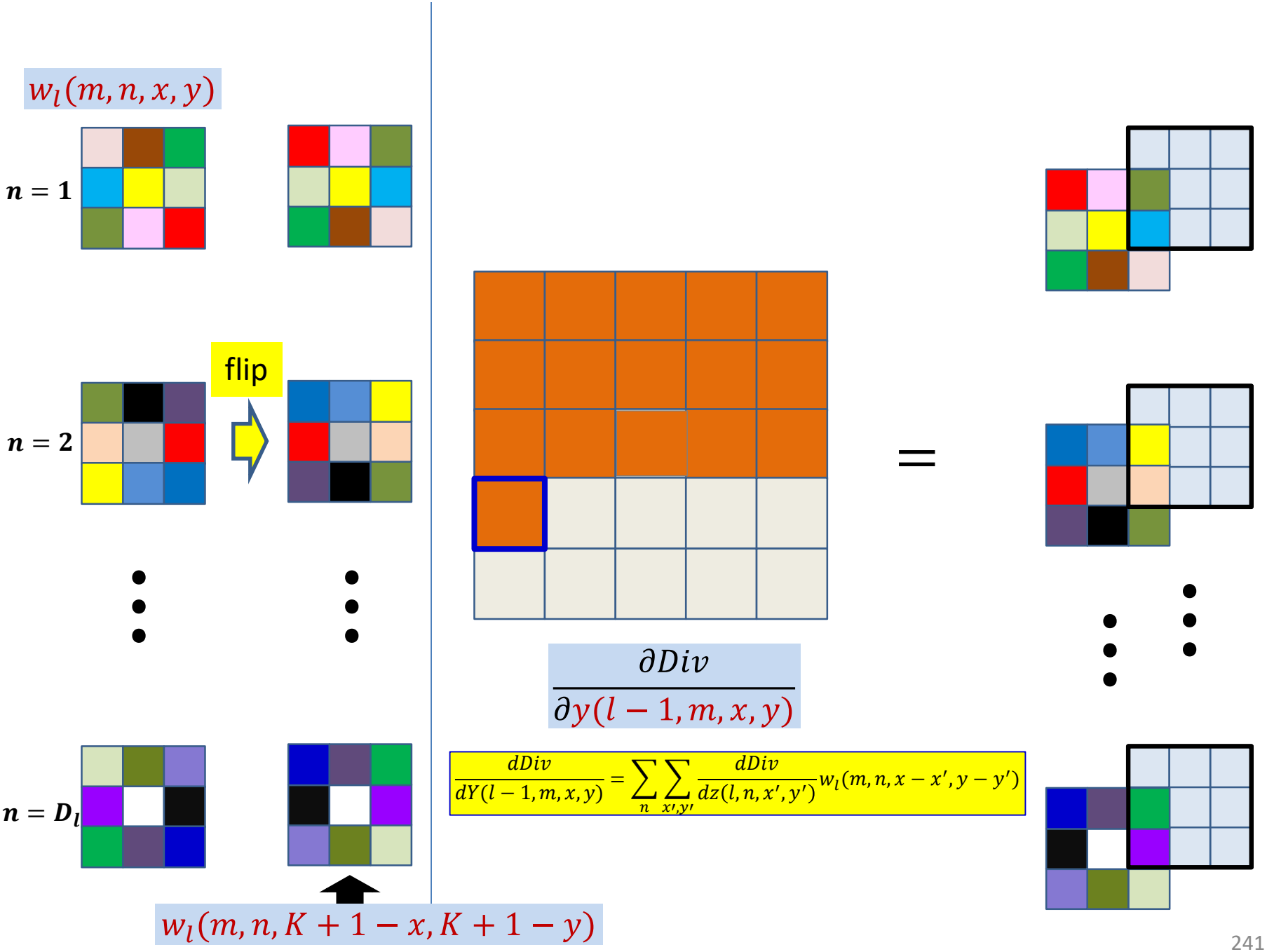


$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

$$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$$

=




$w_l(m,n,x,y)$
$n = 1$
$n = 2$
flip
$n = D_l$
$w_l(m,n,K+1-x,K+1-y)$
=
$\frac{\partial Div}{\partial y(l-1,m,x,y)}$
$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$

$w_l(m,n,x,y)$

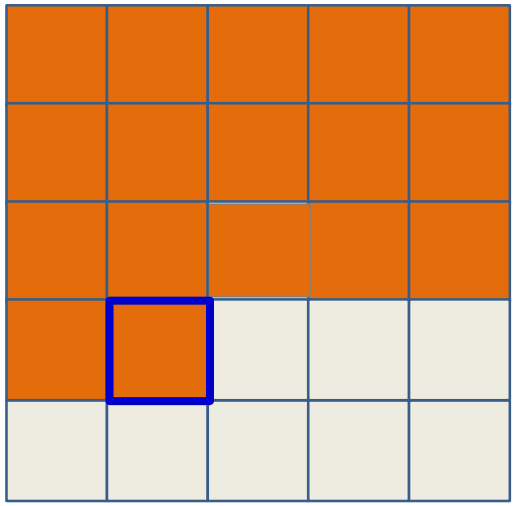
$\boldsymbol{n=1}$

$\boldsymbol{n=2}$

flip

$\boldsymbol{n=D_l}$

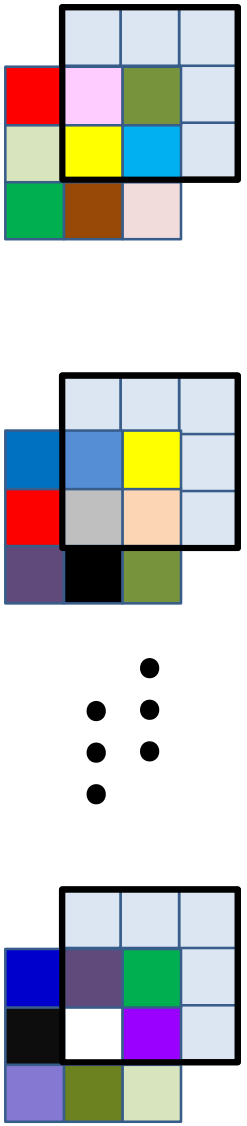
$w_l(m,n,K+1-x,K+1-y)$



$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

$$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$$

=



$w_l(m,n,x,y)$

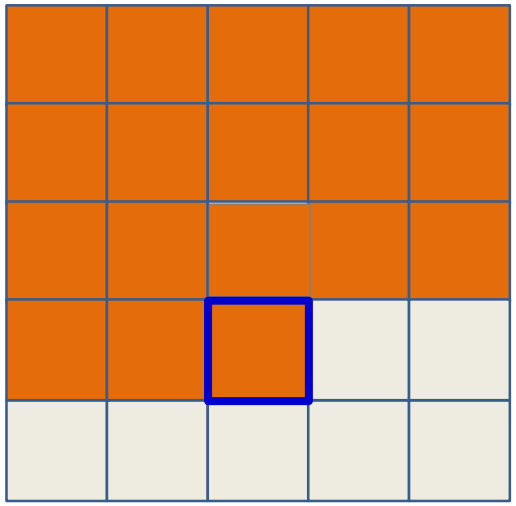
$\boldsymbol{n=1}$

$\boldsymbol{n=2}$

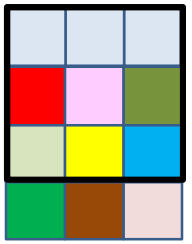
flip

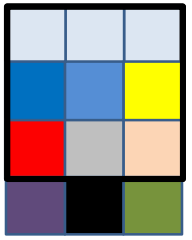
$\boldsymbol{n=D_l}$

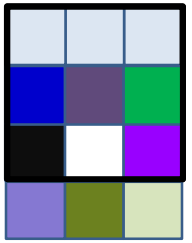
$w_l(m,n,K+1-x,K+1-y)$



=







$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

$$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$$

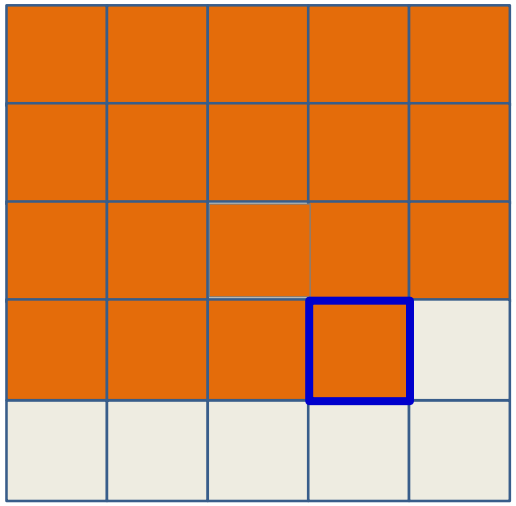
$w_l(m,n,x,y)$

$n = 1$

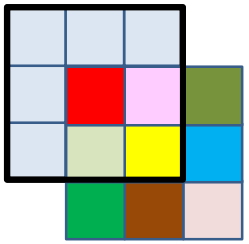
$n = 2$

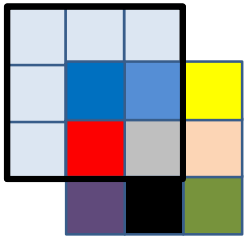
flip

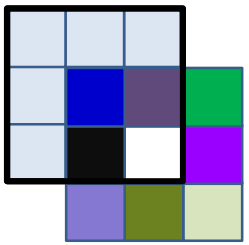
$n = D_l$



=



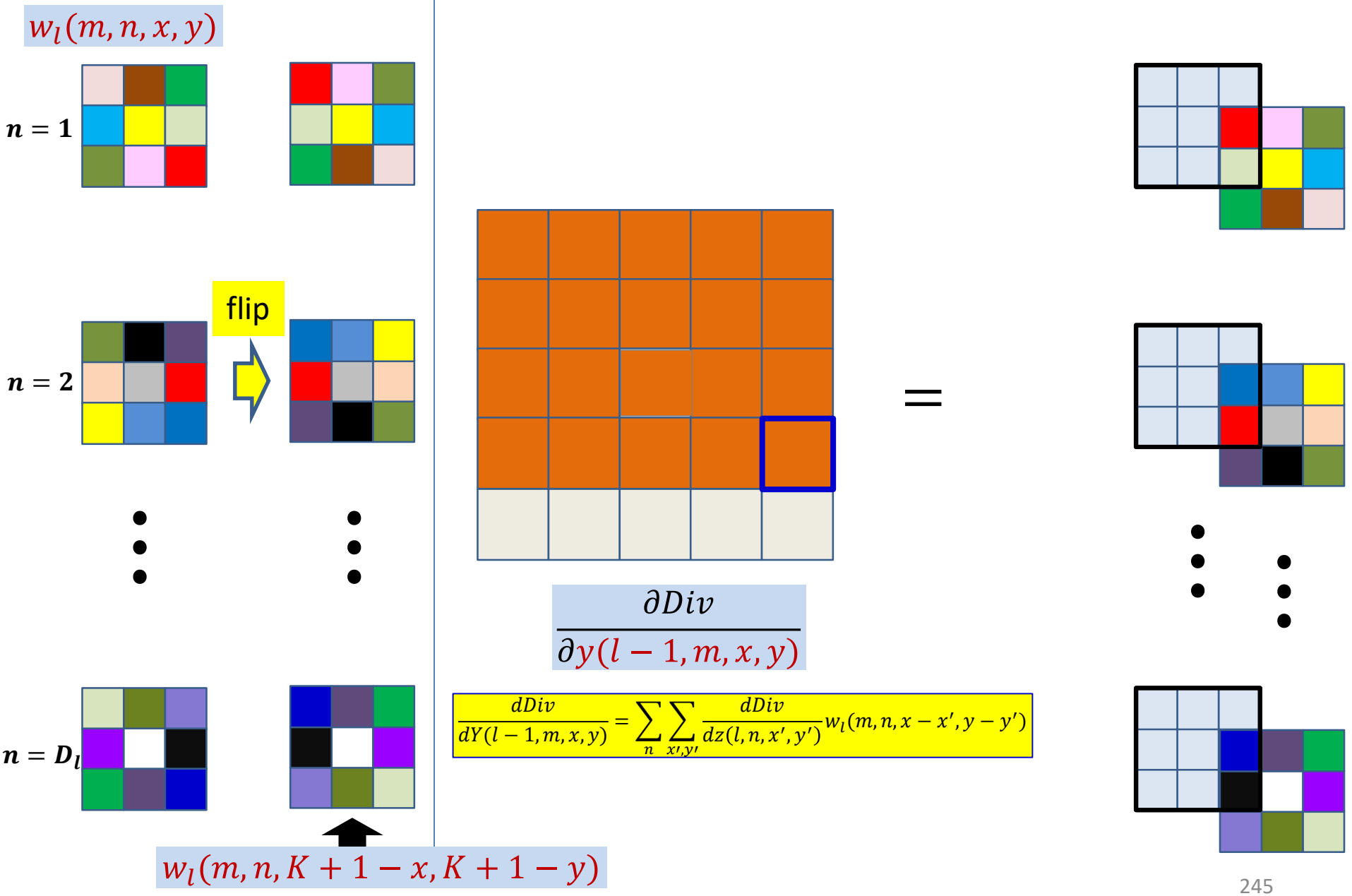


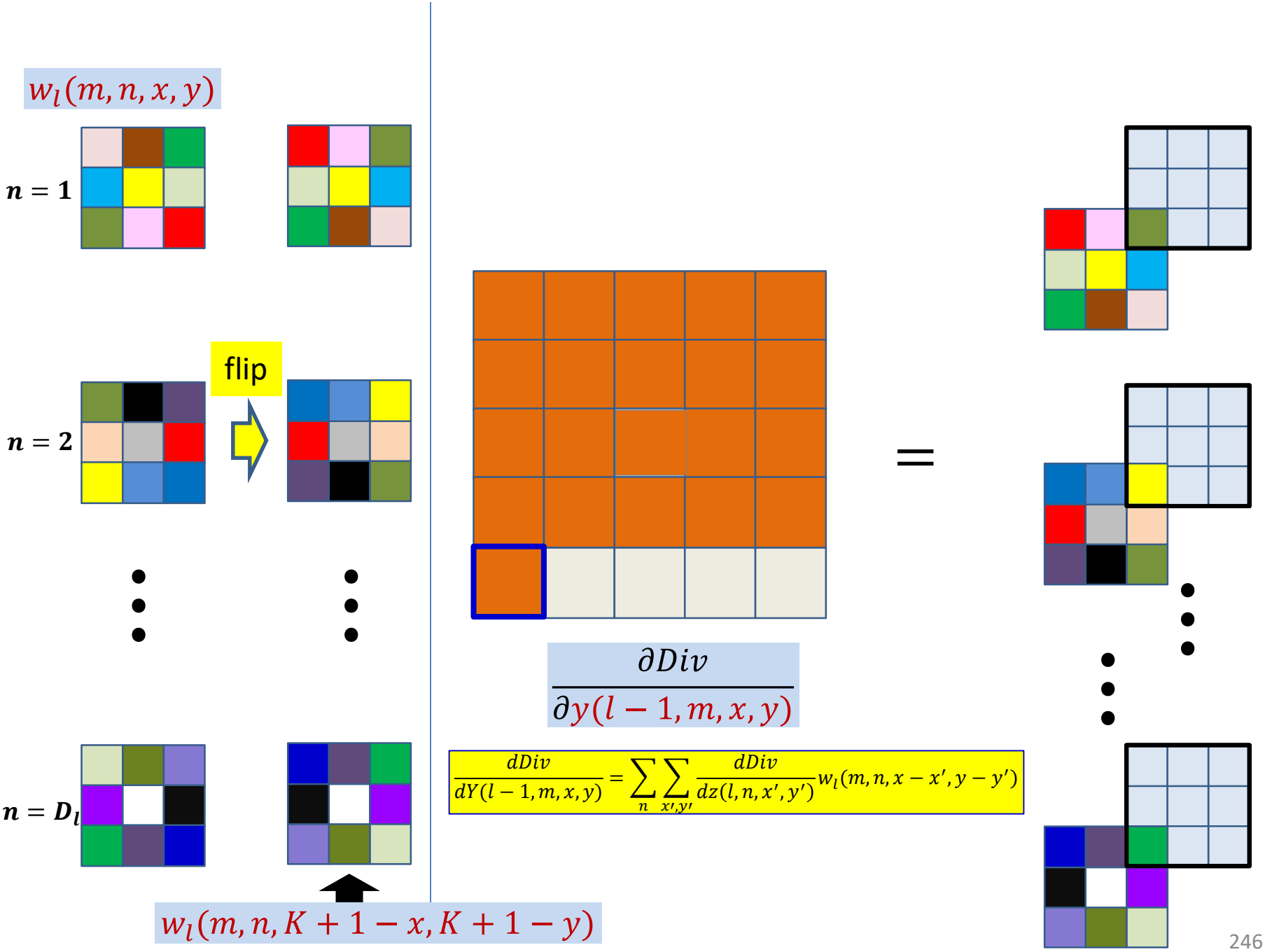


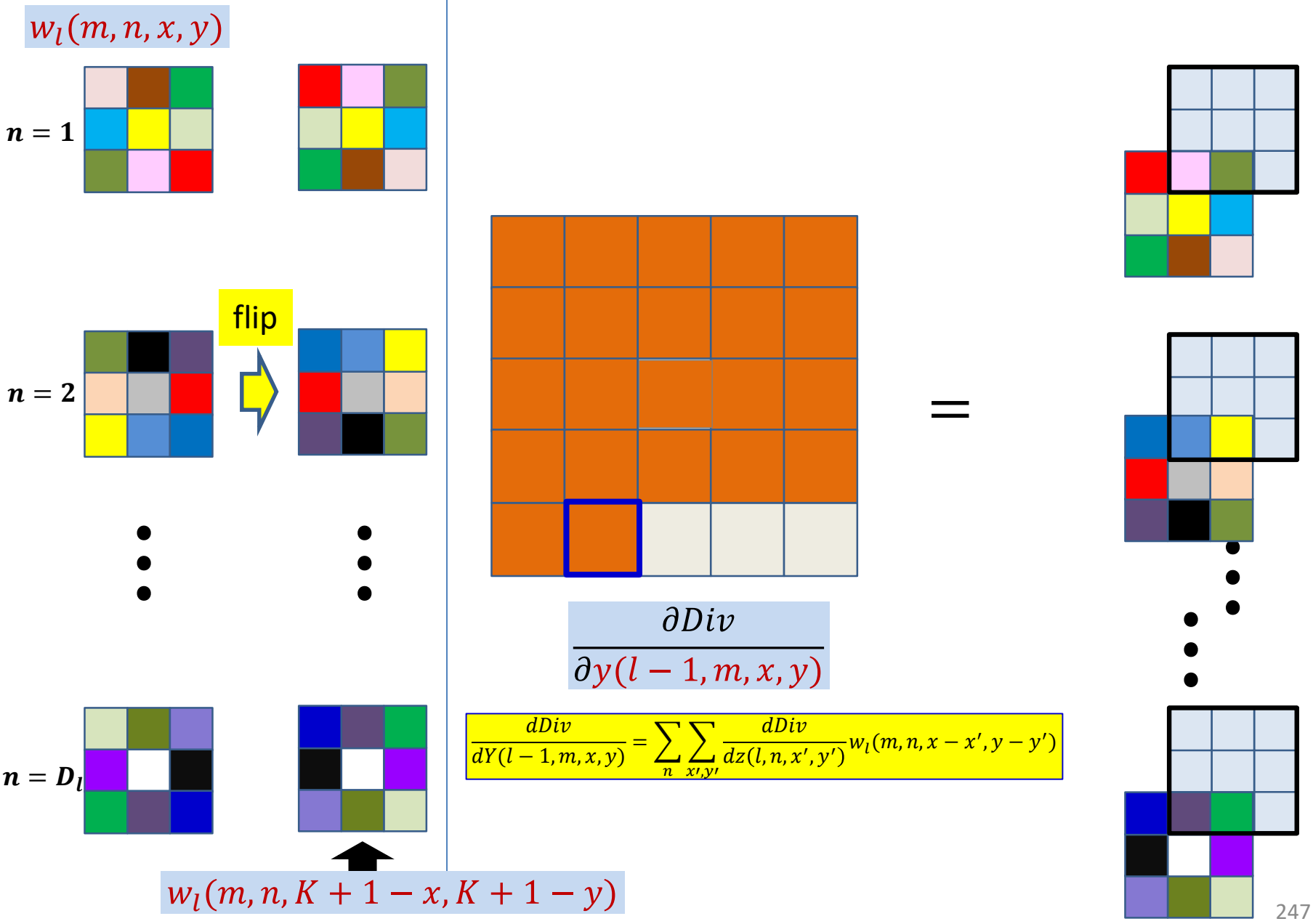
$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

$$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$$

$w_l(m,n,K+1-x,K+1-y)$


$w_l(m,n,x,y)$
$n = 1$
$n = 2$
flip
$n = D_l$
$w_l(m,n,K+1-x,K+1-y)$
=
$\frac{\partial Div}{\partial y(l-1,m,x,y)}$
$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$


$w_l(m,n,x,y)$
$n = 1$
$n = 2$
flip
$n = D_l$
$w_l(m,n,K+1-x,K+1-y)$
=
$\frac{\partial Div}{\partial y(l-1,m,x,y)}$
$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$


$w_l(m,n,x,y)$
$n = 1$
$n = 2$
flip
$n = D_l$
$w_l(m,n,K+1-x,K+1-y)$
=
$\frac{\partial Div}{\partial y(l-1,m,x,y)}$
$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$

$w_l(m,n,x,y)$

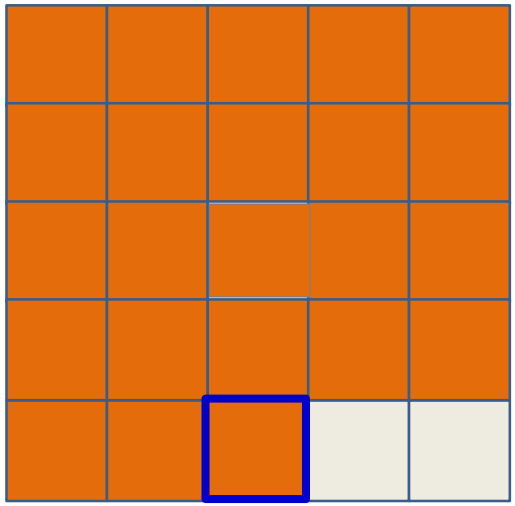
$\boldsymbol{n=1}$

$\boldsymbol{n=2}$

flip

$\boldsymbol{n=D_l}$

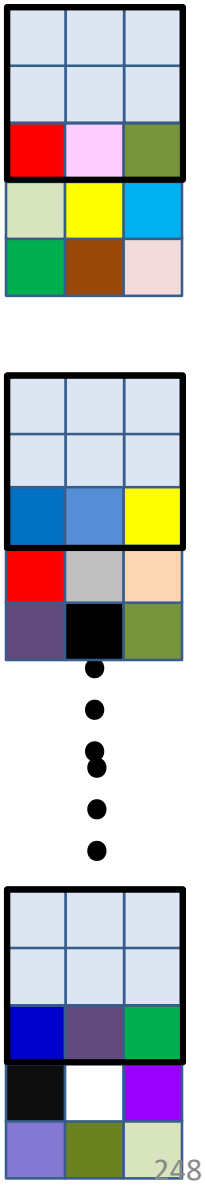
$w_l(m,n,K+1-x,K+1-y)$



$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

$$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$$

=



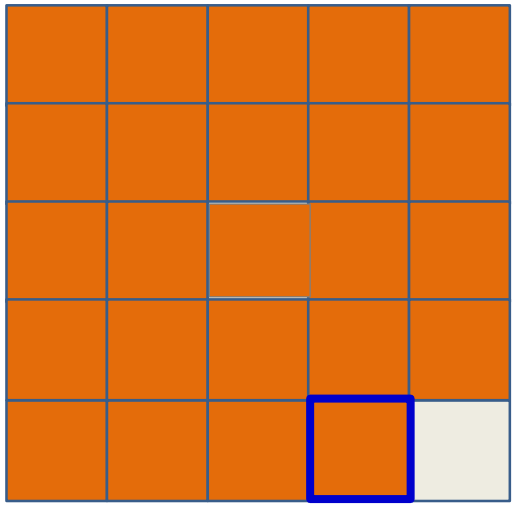
$w_l(m,n,x,y)$

$n = 1$

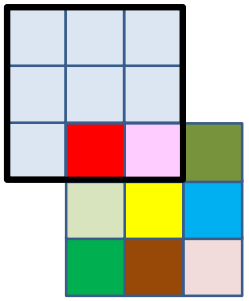
$n = 2$

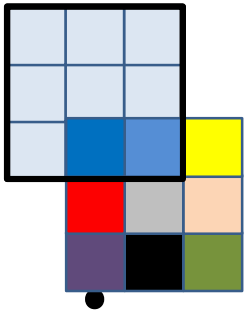
flip

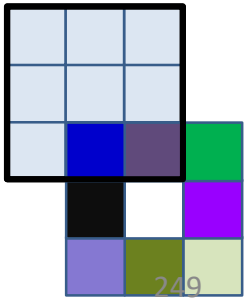
$n = D_l$



=







$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

$$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$$

$w_l(m,n,K+1-x,K+1-y)$

$w_l(m,n,x,y)$

$n = 1$

$n = 2$

flip

$n = D_l$

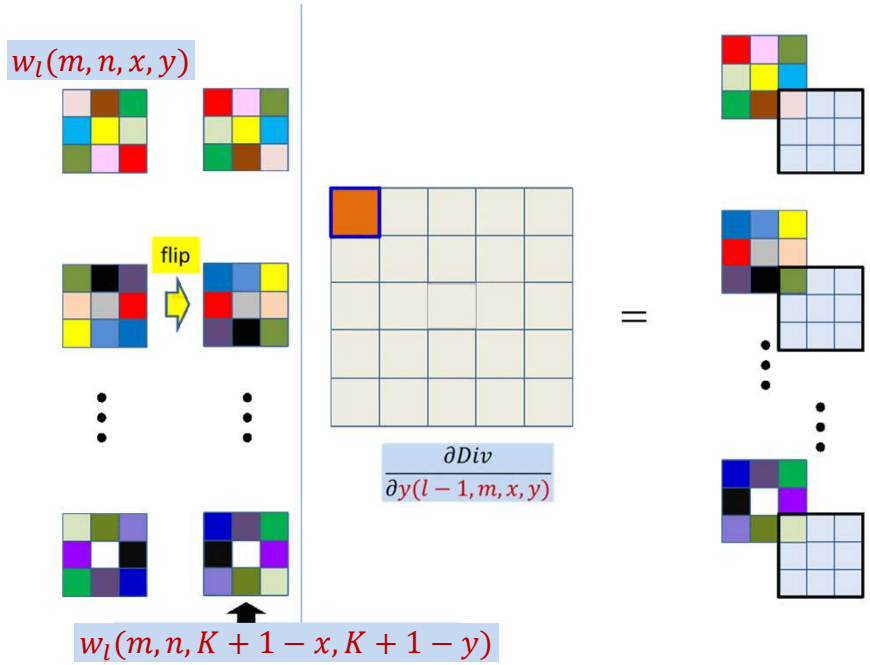
$w_l(m,n,K+1-x,K+1-y)$

$$\frac{\partial Div}{\partial y(l-1,m,x,y)}$$

$$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l,n,x',y')} w_l(m,n,x-x',y-y')$$

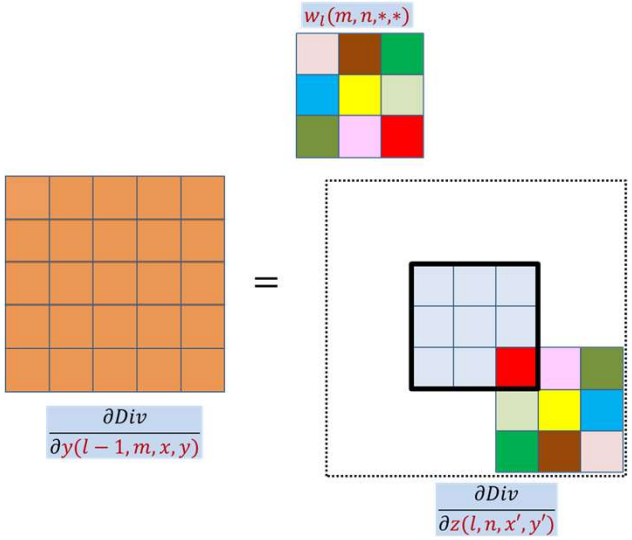
=

Computing the derivative for $Y(l-1,m)$



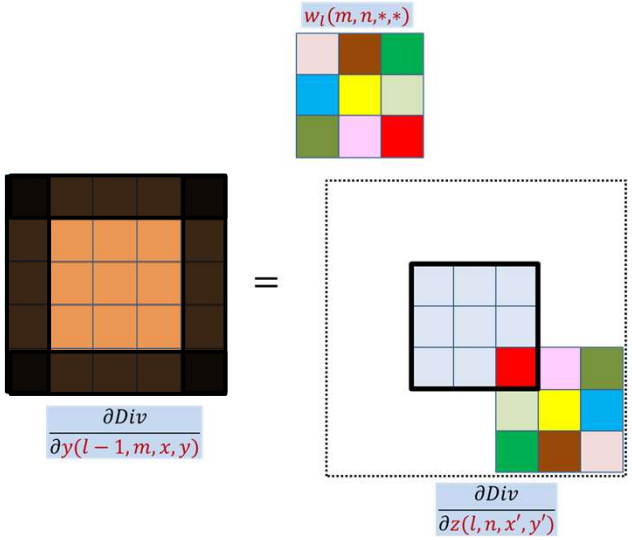
- This is just a convolution of the zero-padded $\frac{\partial Div}{\partial z(l,n,x,y)}$ maps by the transposed and flipped filter
  - After zero padding it first with $K-1$ zeros on every side

The size of the Y-derivative map



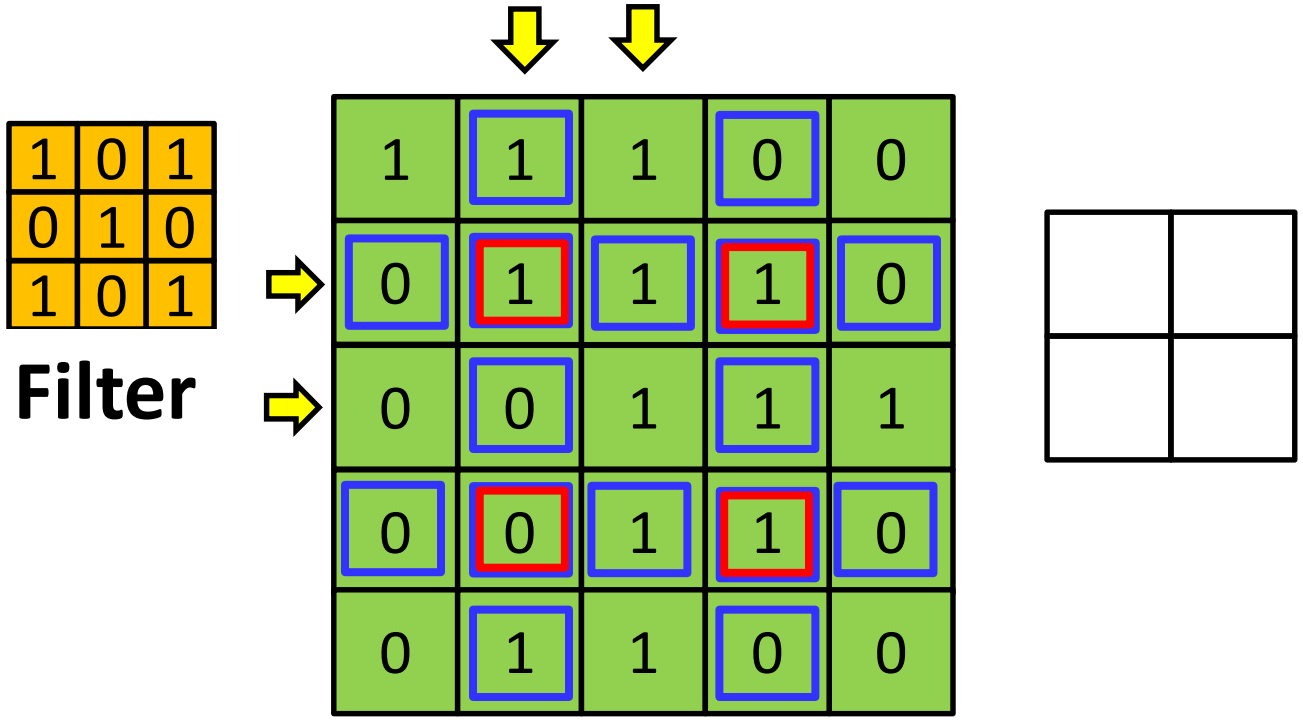
- We continue to compute elements for the derivative Y map as long as the (flipped) filter has at least one element in the (unpadded) derivative Zmap
  - I.e. so long as the Y derivative is non-zero
- The size of the Y derivative map will be $(H + K - 1) \times (W + K - 1)$
  - H and W are heidght and width of the Zmap
- This will be the size of the actual Y map that was originally convolved

The size of the Y-derivative map



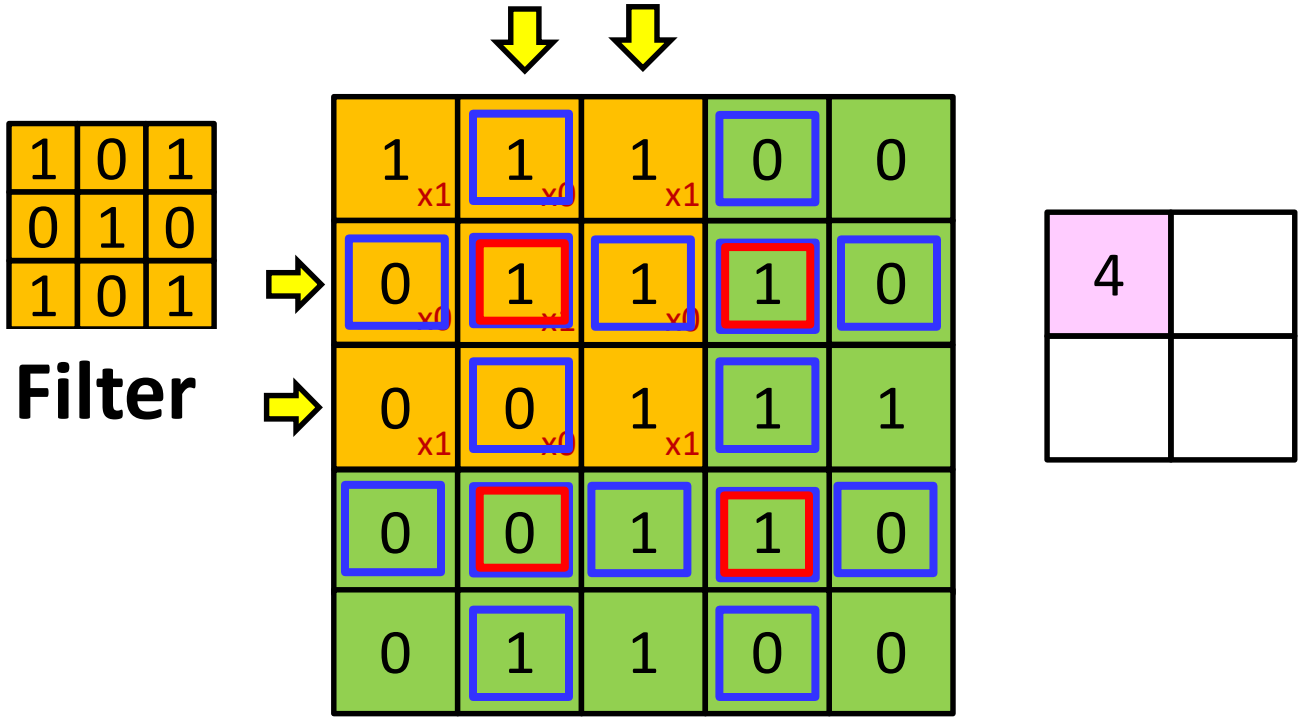
- If the Y map was zero-padded in the forward pass, the derivative map will be the size of the *zero-padded* map
  - The zero padding regions must be deleted before further backprop

When the stride is more than 1?



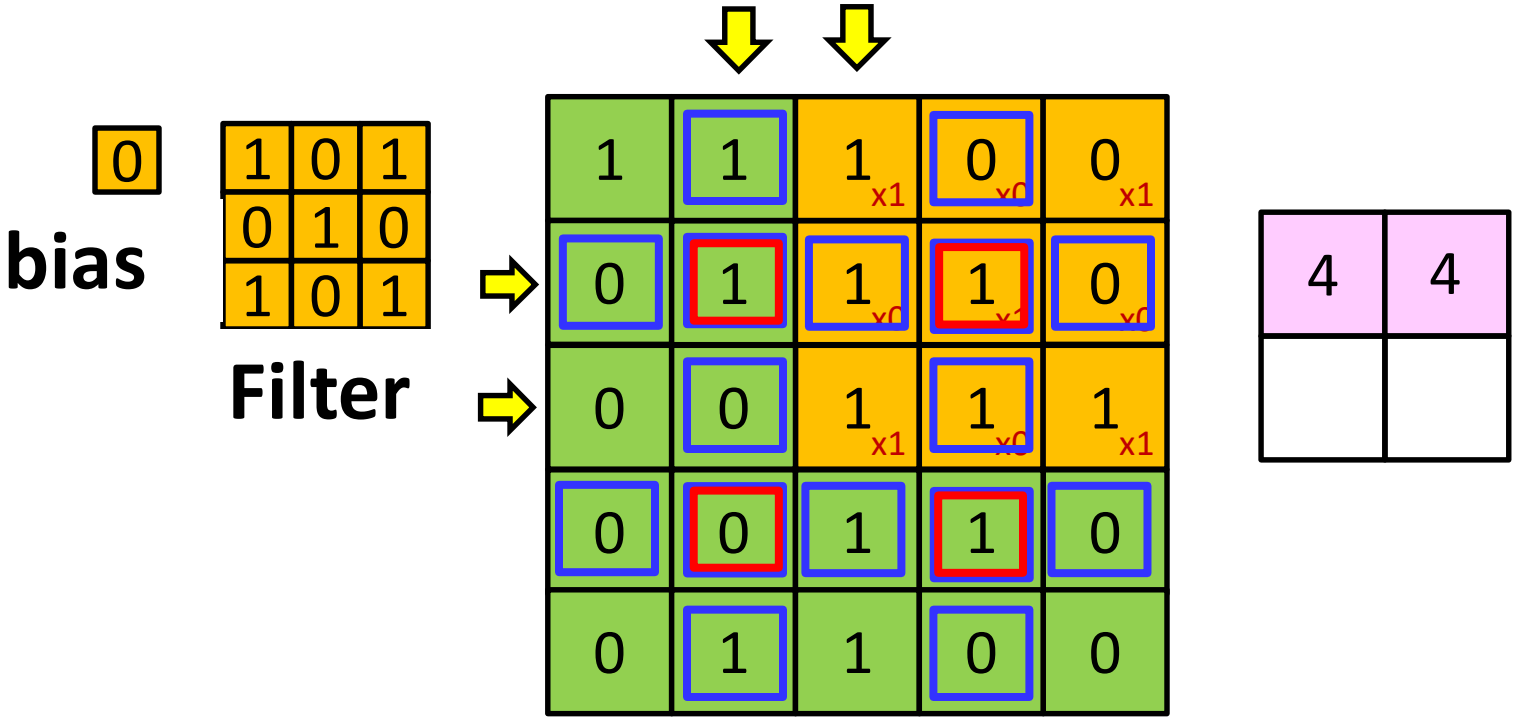
- When the stride is greater than 1, some positions of $Y(l-1,m)$ contribute to more locations on the $Z(l,n)$ maps than others
  - With a stride of 2, the boxed-in-blue $Y(l-1,m)$ locations contribute to half as many $Z(l,n)$ locations as the unboxed locations
  - The double-boxed (blue and red boxes) $Y(l-1,m)$ locations contribute to only a quarter as many $Z(l,n)$ locations as the unboxed ones

When the stride is more than 1?



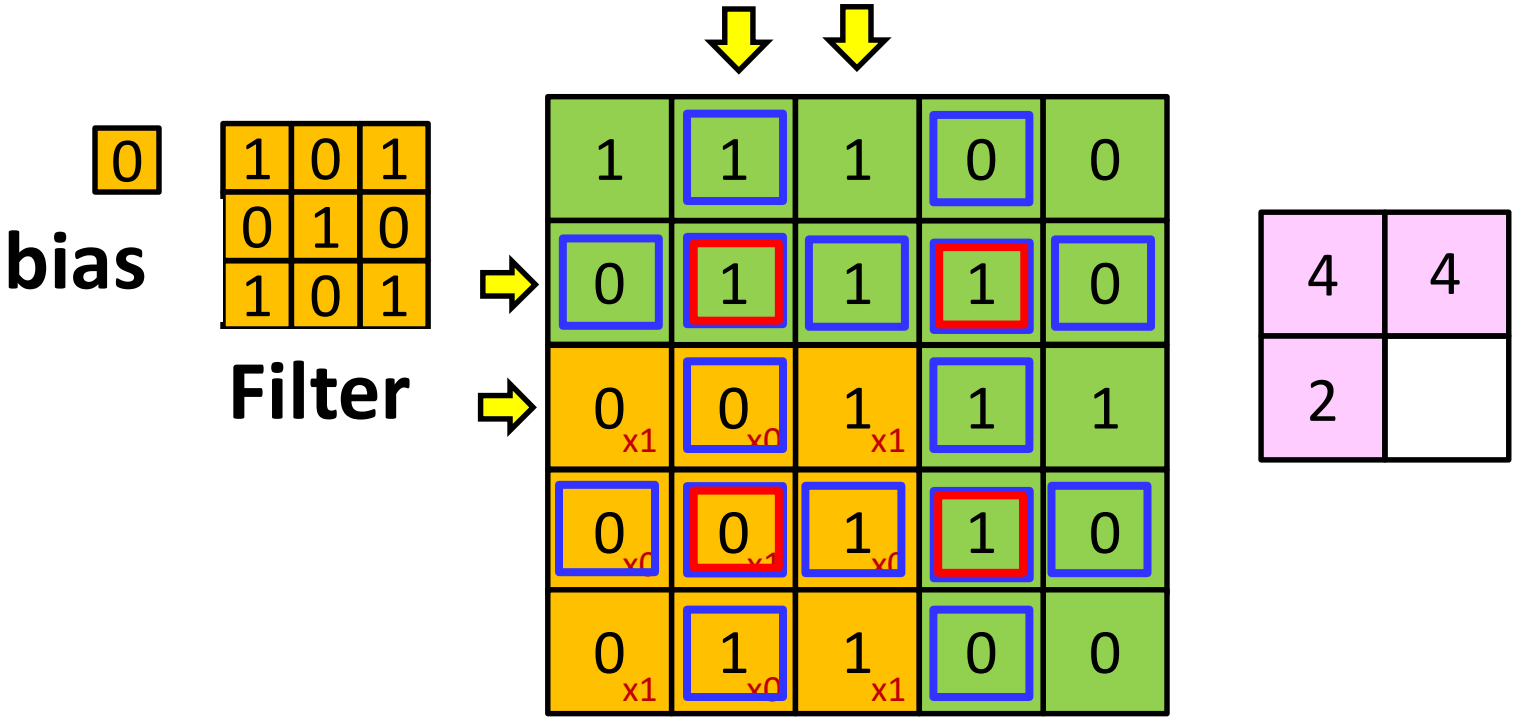
- When the stride is greater than 1, some positions of $Y(l-1,m)$ contribute to more locations on the $Z(l,n)$ maps than others
  - With a stride of 2, the boxed-in-blue $Y(l-1,m)$ locations contribute to half as many $Z(l,n)$ locations as the unboxed locations
  - The double-boxed (blue and red boxes) $Y(l-1,m)$ locations contribute to only a quarter as many $Z(l,n)$ locations as the unboxed ones

When the stride is more than 1?



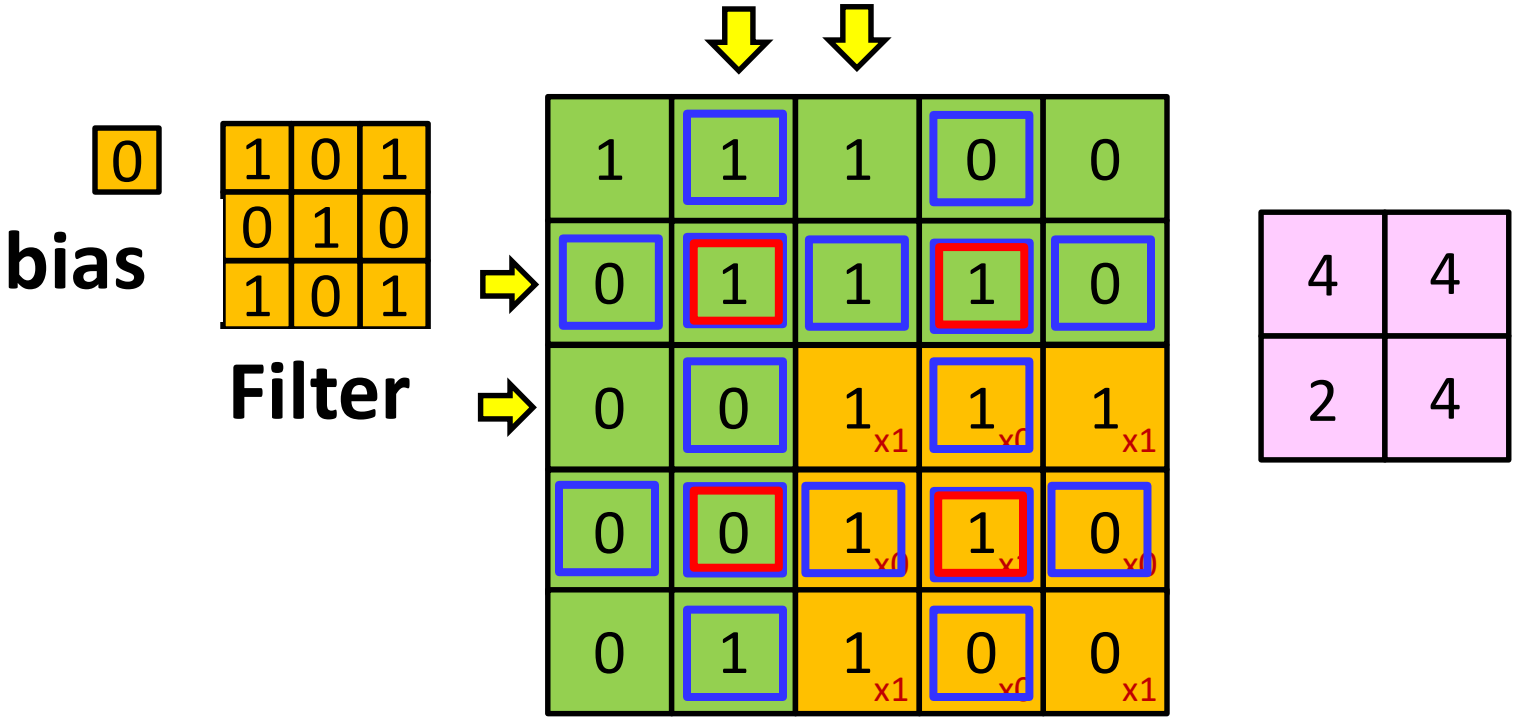
- When the stride is greater than 1, some positions of $Y(l-1,m)$ contribute to more locations on the $Z(l,n)$ maps than others
  - With a stride of 2, the boxed-in-blue $Y(l-1,m)$ locations contribute to half as many $Z(l,n)$ locations as the unboxed locations
  - The double-boxed (blue and red boxes) $Y(l-1,m)$ locations contribute to only a quarter as many $Z(l,n)$ locations as the unboxed ones

When the stride is more than 1?



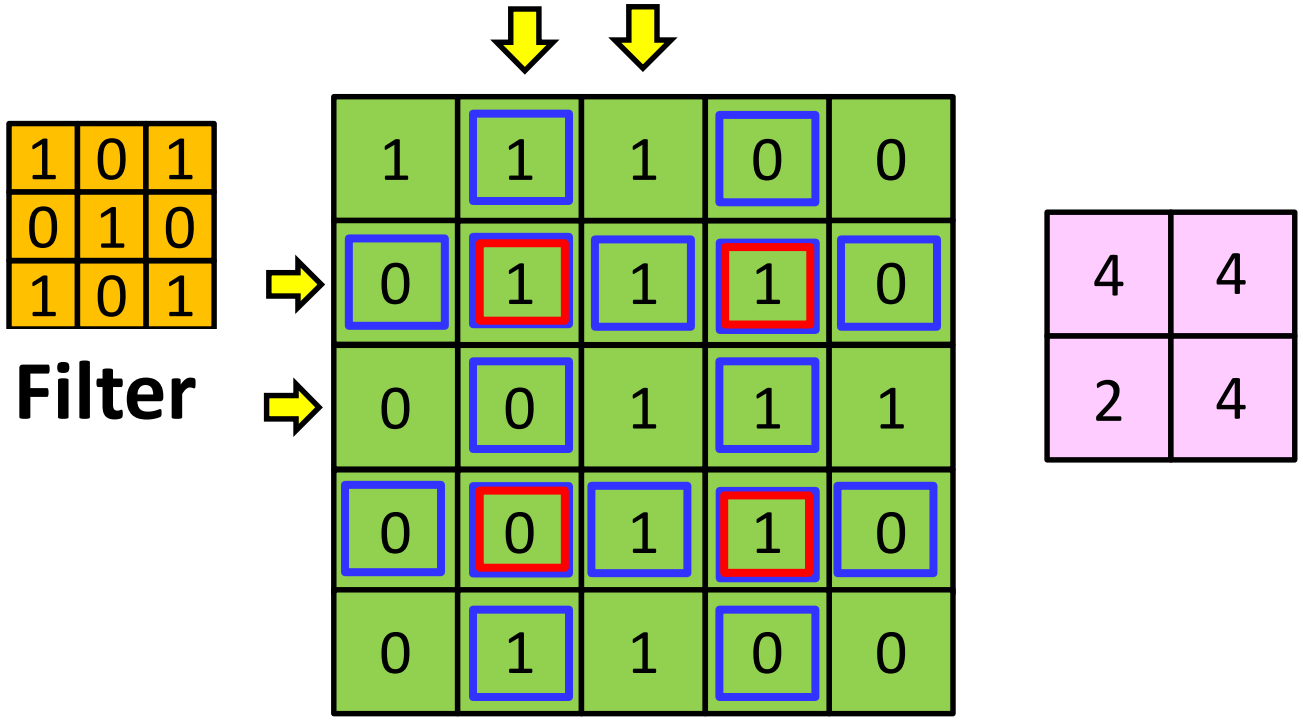
- When the stride is greater than 1, some positions of $Y(l-1,m)$ contribute to more locations on the $Z(l,n)$ maps than others
  - With a stride of 2, the boxed-in-blue $Y(l-1,m)$ locations contribute to half as many $Z(l,n)$ locations as the unboxed locations
  - The double-boxed (blue and red boxes) $Y(l-1,m)$ locations contribute to only a quarter as many $Z(l,n)$ locations as the unboxed ones

When the stride is more than 1?



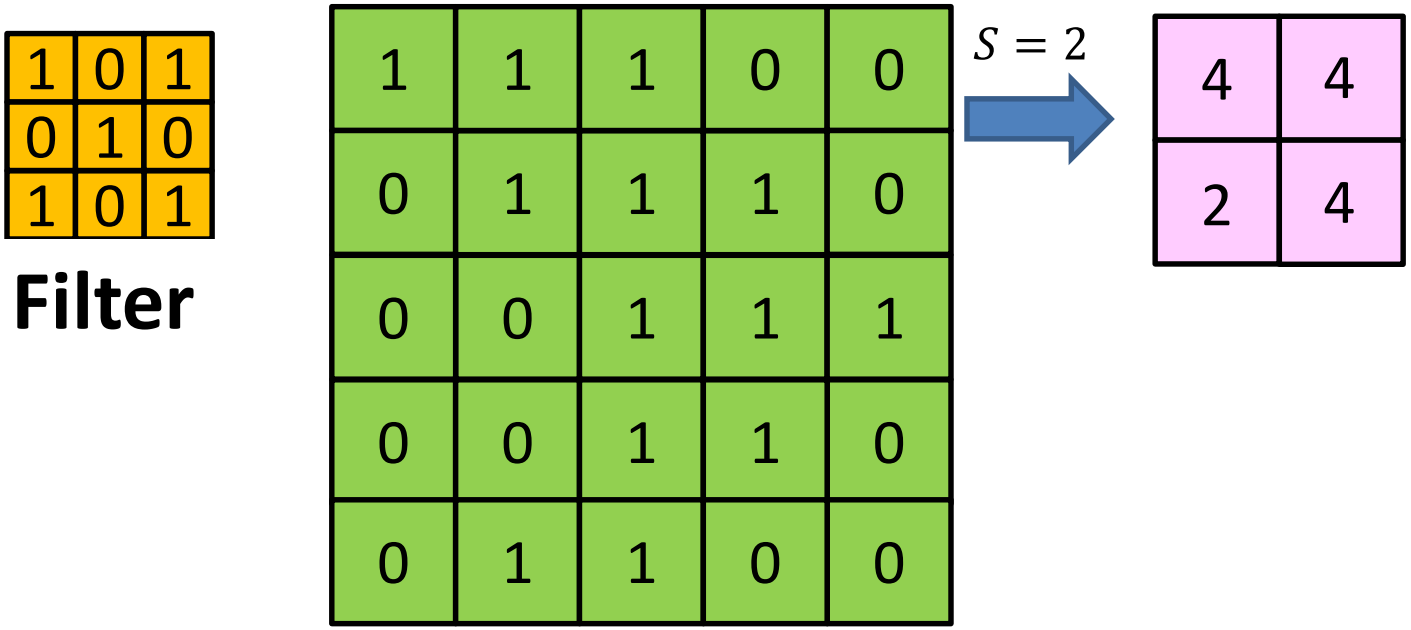
- When the stride is greater than 1, some positions of $Y(l-1,m)$ contribute to more locations on the $Z(l,n)$ maps than others
  - With a stride of 2, the boxed-in-blue $Y(l-1,m)$ locations contribute to half as many $Z(l,n)$ locations as the unboxed locations
  - The double-boxed (blue and red boxes) $Y(l-1,m)$ locations contribute to only a quarter as many $Z(l,n)$ locations as the unboxed ones

When the stride is more than 1?



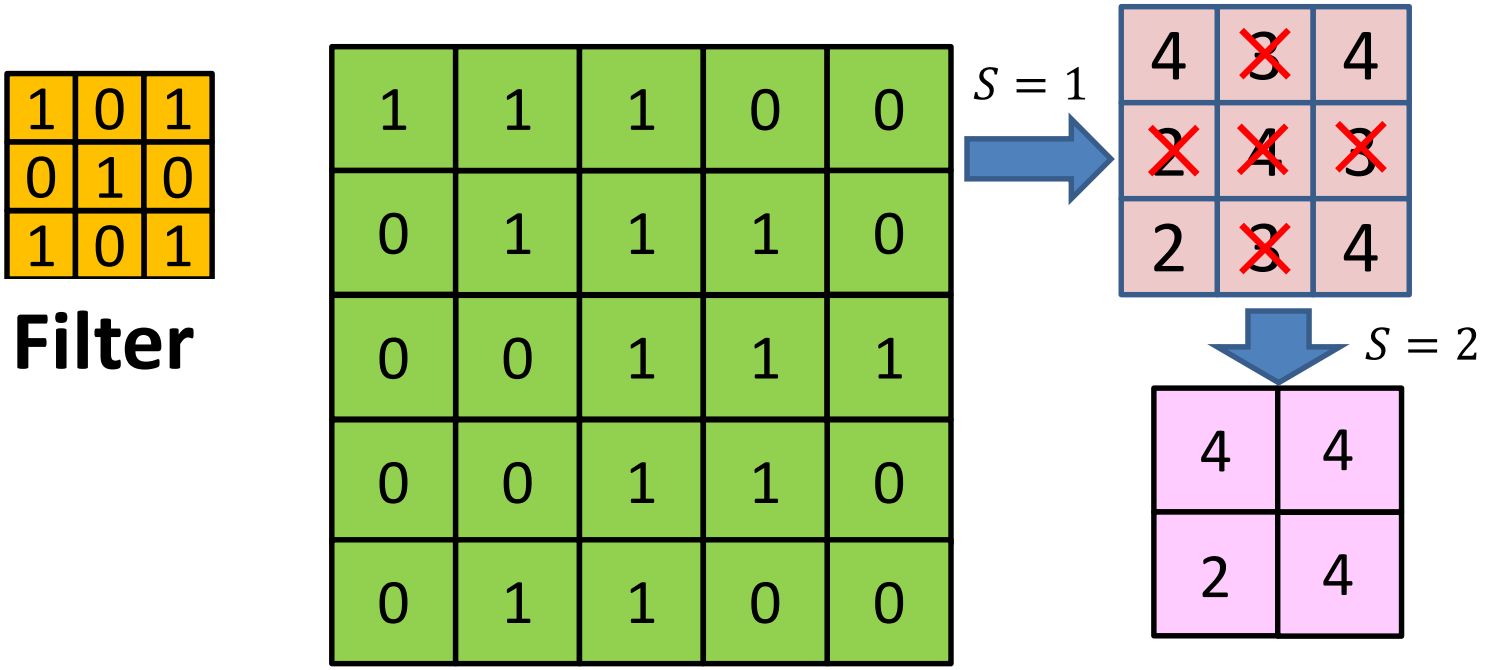
- We must make adjustments for when the stride is greater than 1.

Stride greater than 1



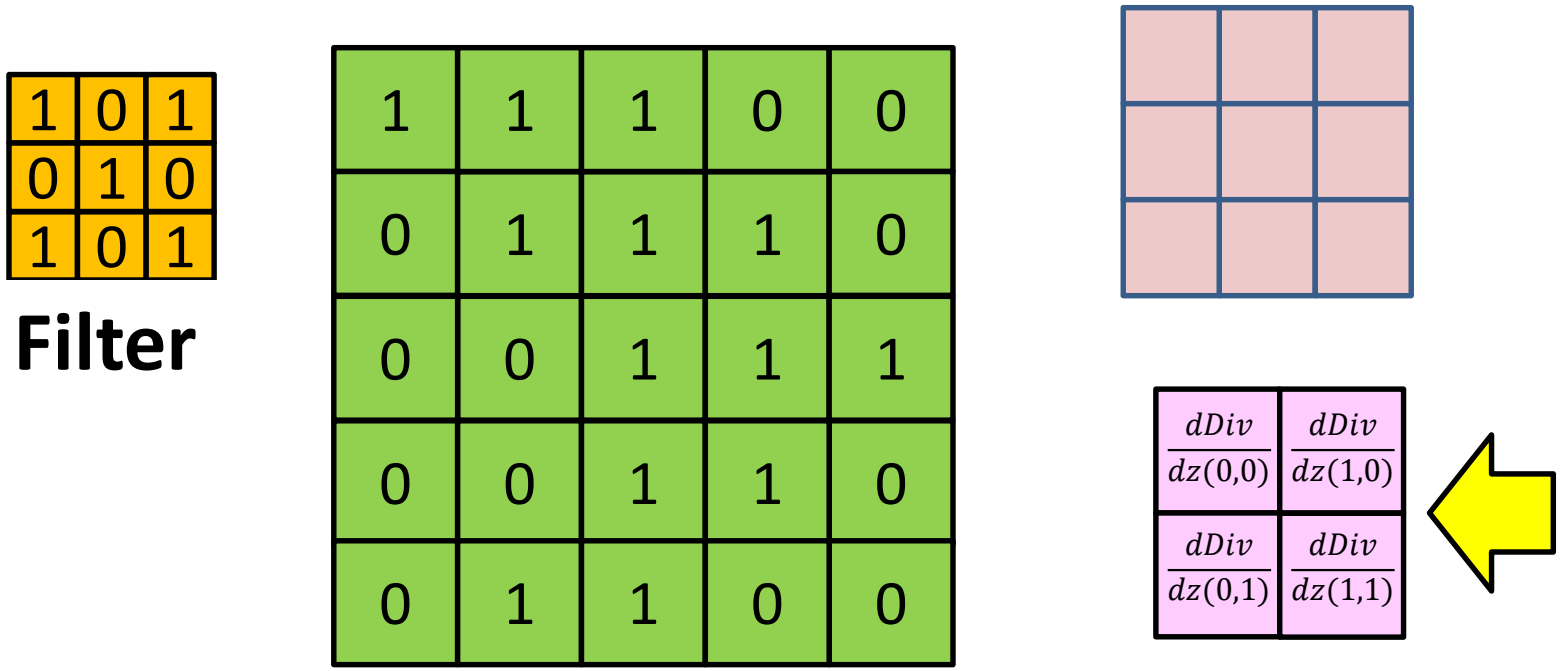
- **How do we adjust the formulae when we downsample by S?**
  - The output Z map is now a factor S smaller on every side
  - We have not computed all the terms in the usual formula

Stride greater than 1



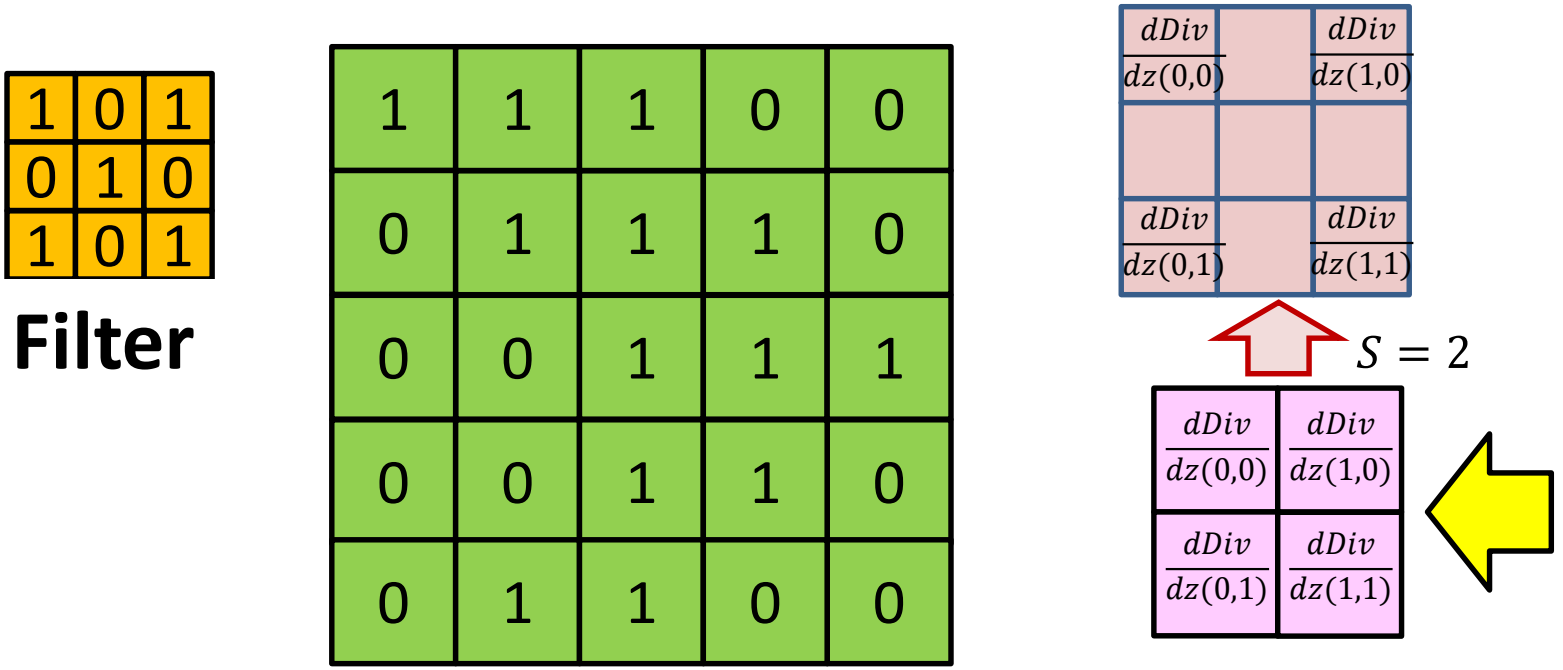
- **Observation:** Convolving with a stride S greater than 1 is the same as convolving with stride 1 and "dropping" $S - 1$ out of every S rows, and $S - 1$ of every S columns
  - **Downsampling by S**
  - E.g. for stride 2, it is the same as convolving with stride 1 and dropping every 2nd entry

Derivatives with Stride greater than 1



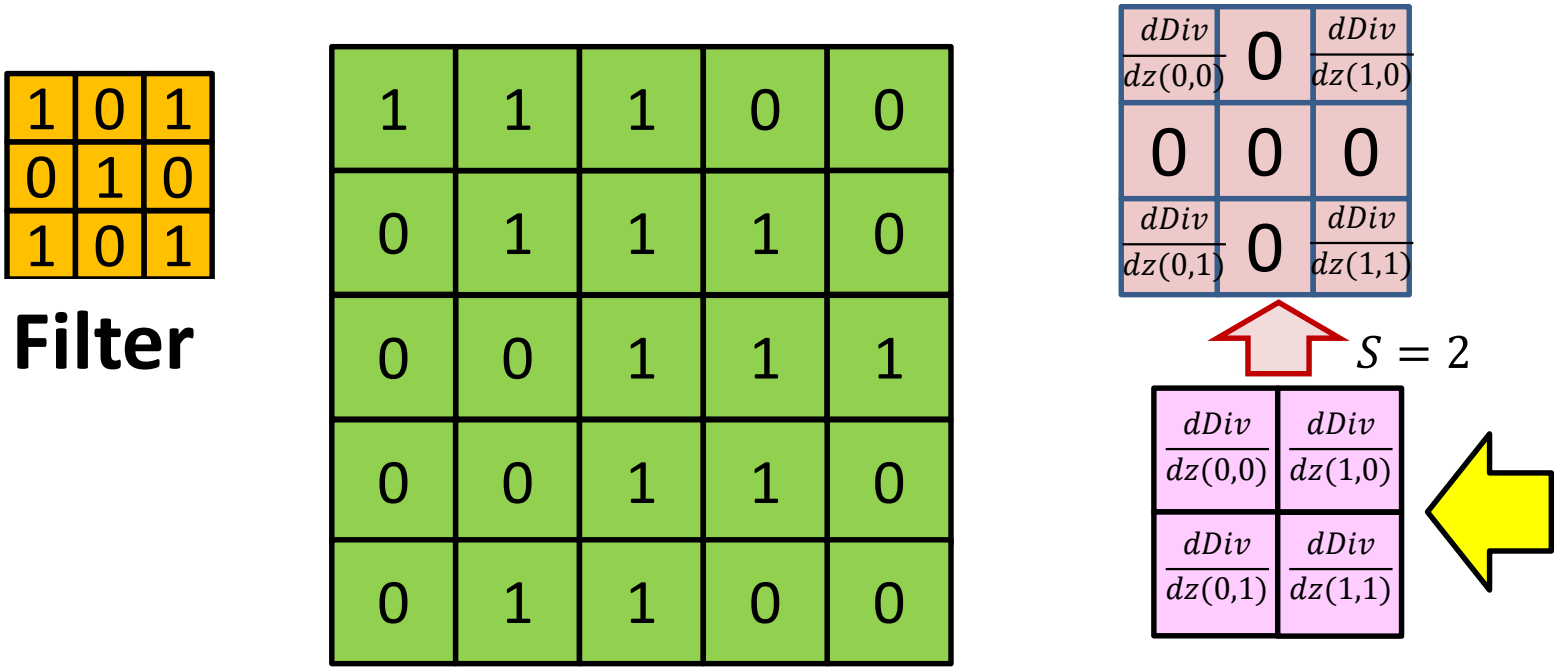
- **Derivatives:** Backprop gives us the derivatives of the divergence with respect to the elements of the *downsampled* (strided) Z map

Derivatives with Stride greater than 1



- **Derivatives:** Backprop gives us the derivatives of the divergence with respect to the elements of the *downsampled* (strided) Z map
- We can place these derivative values back into their original locations of the full-sized Z map

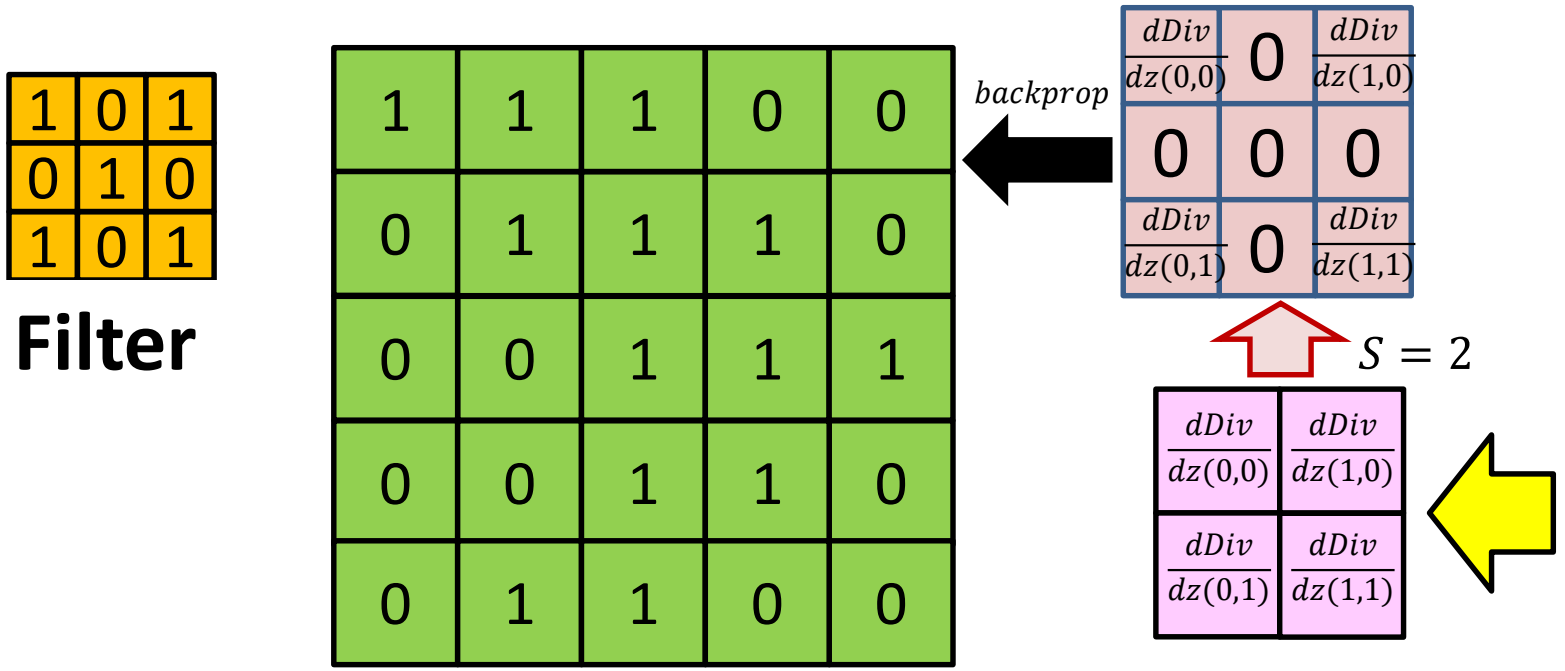
Derivatives with Stride greater than 1



- **Derivatives:** Backprop gives us the derivatives of the divergence with respect to the elements of the *downsampled* (strided) Z map
- We can place these values back into their original locations of the full-sized Z map

- The remaining entries of the Z map do not affect the divergence
  - Since they get dropped out
- The derivative of the divergence w.r.t. these values is 0

Computing derivatives with Stride > 1



- **Upsampling derivative map:**
  - Upsample the downsampled derivatives
  - Insert zeros into the "empty" slots
  - This gives us the derivatives w.r.t. all the entries of a full-sized (stride 1) Z map
- We can compute the derivatives for Y, using the full map

Overall algorithm for computing derivatives w.r.t. $Y(l-1)$

- Given the derivatives $\frac{dDiv}{dz(l,n,x,y)}$
- If stride $S > 1$, upsample derivative map

$$\hat{z}(l,n,Sx,Sy) = \frac{dDiv}{dz(l,n,x,y)}$$

$$\hat{z}(l,n,x,y) = 0 \quad \forall\, x\,, y \neq \textit{integer multiples of } S$$

- For $S = 1$,

$$\hat{z}(l,n,x,y) = \frac{dDiv}{dz(l,n,x,y)}$$

- Compute derivatives using:

$$\frac{dDiv}{dY(l-1,m,x,y)} = \sum_{n}\sum_{x',y'} \hat{z}(l,n,x',y') w_l(m,n,x-x',y-y')$$

Can be computed by convolution with flipped filter

Derivatives for a single layer l: Vector notation

```
# The weight W(l,m) is a 3D D_{l-1} x K_l x K_l
# Assuming dz has already been obtained via backprop
if (stride > 1)  #upsample
    dz = upsample(dz,stride,W_{l-1},H_{l-1},K_l)

dzpad = zeros(D_l x (H_l+2(K_l-1)) x (W_l+2(K_l-1))) # zeropad
for j = 1:D_l
    for i = 1:D_{l-1}   # Transpose and flip
        Wflip(i,j,:,:) = flipLeftRight(flipUpDown(W(l,i,j,:,:)))
    dzpad(j,K_l:K_l+H_l-1,K_l:K_l+W_l-1) = dz(l,j,:,:) #center map
end
```

```
for j = 1:D_{l-1}
  for x = 1:W_{l-1}
    for y = 1:H_{l-1}
       segment = dzpad(:, x:x+K_l-1, y:y+K_l-1) #3D tensor
       dy(l-1,j,x,y) = Wflip.segment #tensor inner prod.
```

Upsampling

```
# Upsample dz to the size it would be if stride was 1
function upsample(dz, S, W, H, K)
    if (S > 1)  #Insert S-1 zeros between samples
        Hup = H - K + 1
        Wup = W - K + 1
        dzup = zeros(Wup, Hup)
        for x  = 1:H
            for y = 1:W
                dzup((x-1)S+1,(y-1)S+1) = dz(x,y)
    else
        dzup = dz
    return dzup
```

Upsampling

```
function upsample(dz, S)
    if (S > 1)  #Insert S-1 zeros between samples
        dzup = zeros((H-1)xS+1, (W-1)xS+1)
        for x  = 1:H
        for y = 1:W
            dzup((x-1)S+1,(y-1)S+1) = dz(x,y)
    else
        dzup = dz
    return dzup
```

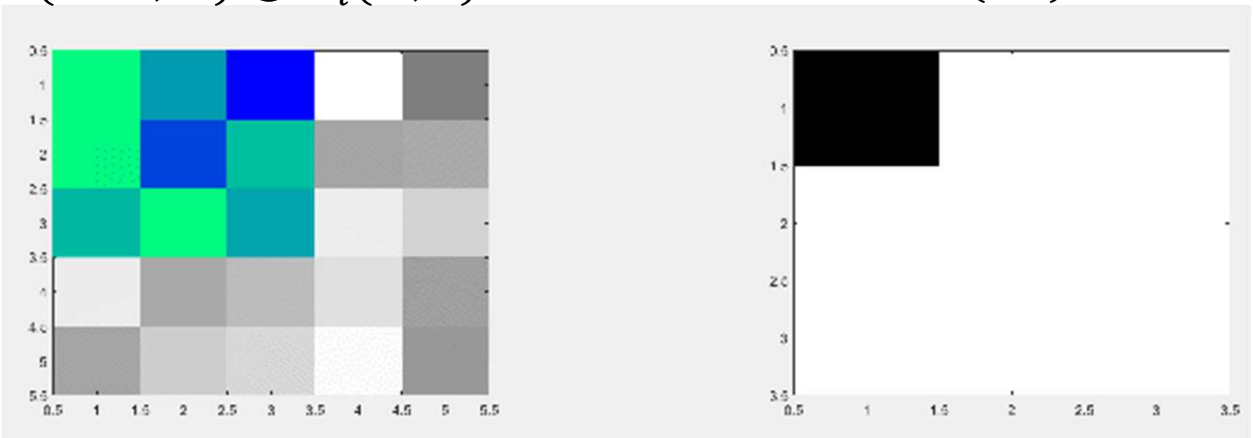
Backpropagating through affine map

- Forward affine computation:
  - Compute affine maps $z(l,n,x,y)$ from previous layer maps $y(l-1,m,x,y)$ and filters $w_l(m,n,x,y)$

- Backpropagation: Given $\frac{dDiv}{dz(l,n,x,y)}$
  -  Compute derivative w.r.t. $y(l-1,m,x,y)$
  - Compute derivative w.r.t. $w_l(m,n,x,y)$

The derivatives for the weights

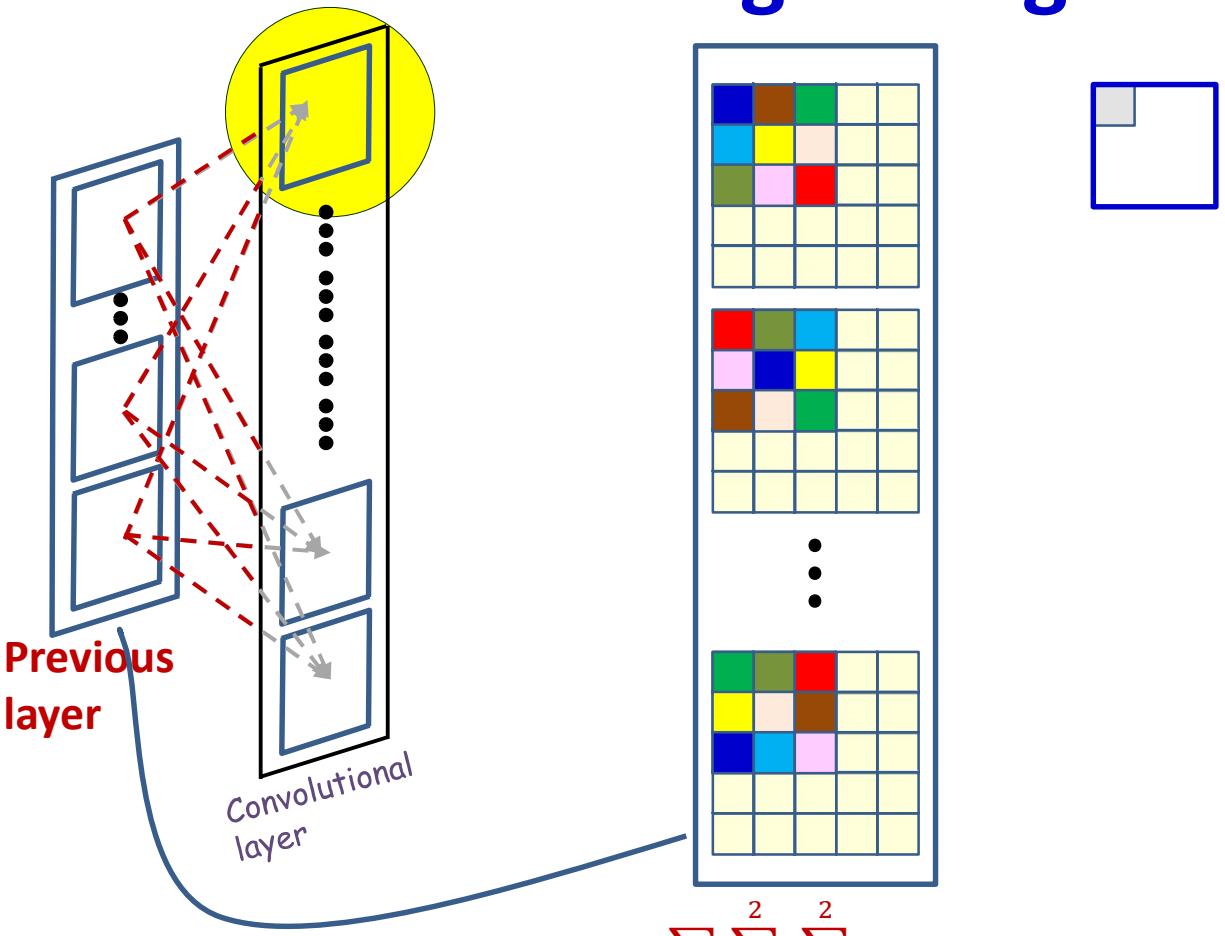
$Y(l-1,m) \otimes w_l(m,n)$ $Z(l,n)$



$$z(l,n,x,y) = \sum_m \sum_{x',y'} w_l(m,n,x',y') y(l-1,m,x+x',y+y') + b_l(n)$$

- Each ***weight*** $w_l(m,n,x',y')$ affects several $z(l,n,x,y)$
  - Consider the contribution of one filter components: $w_l(m,n,i,j)$ (e.g. $w_l(m,n,1,2)$)

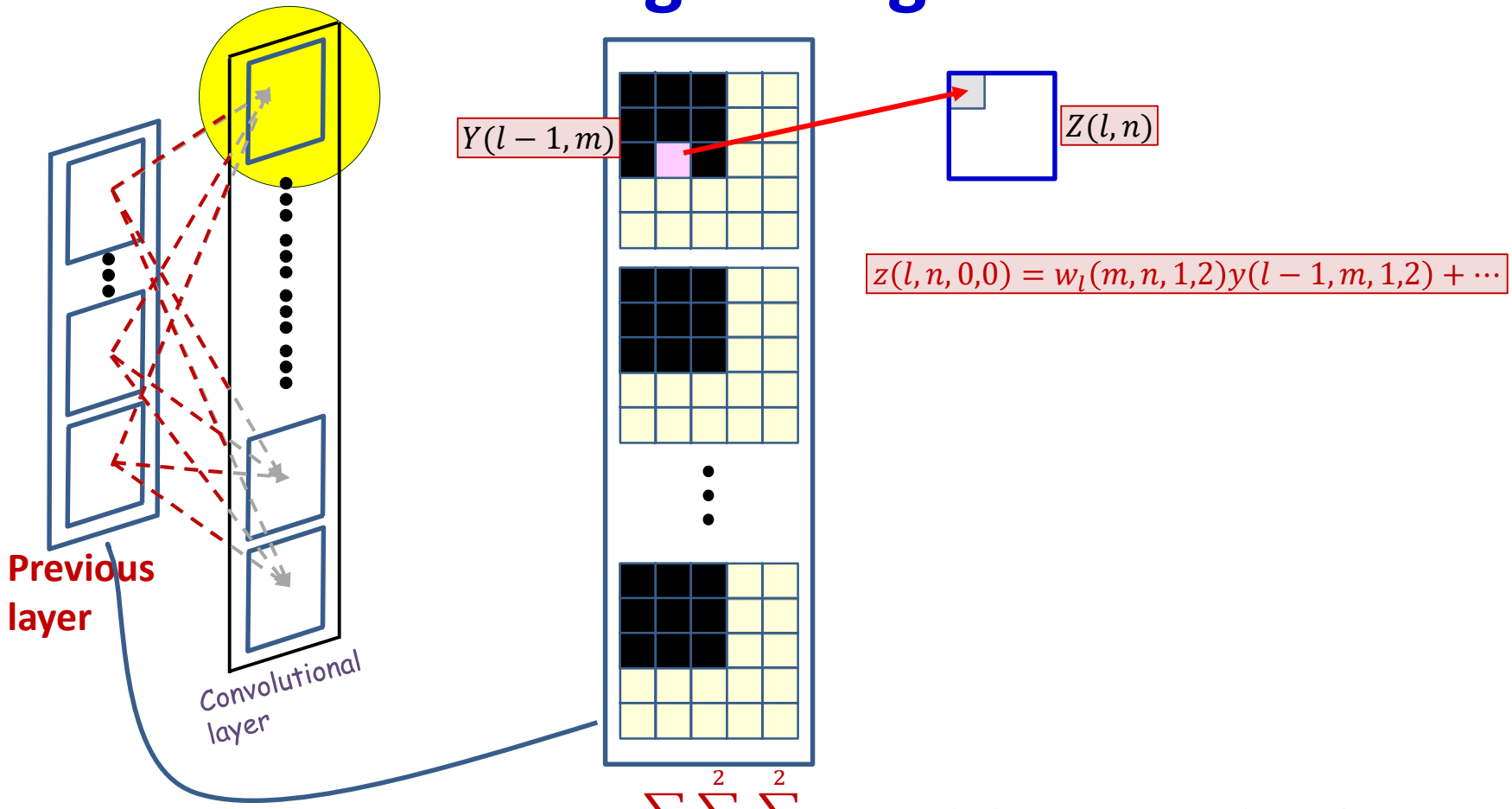
Convolution: the contribution of a single weight



$$z(l,n,x,y) = \sum_{m} \sum_{x'=0}^{2} \sum_{y'=0}^{2} w_l(m,n,x',y') y(l-1,m,x+x',y+x') + b_l(n)$$

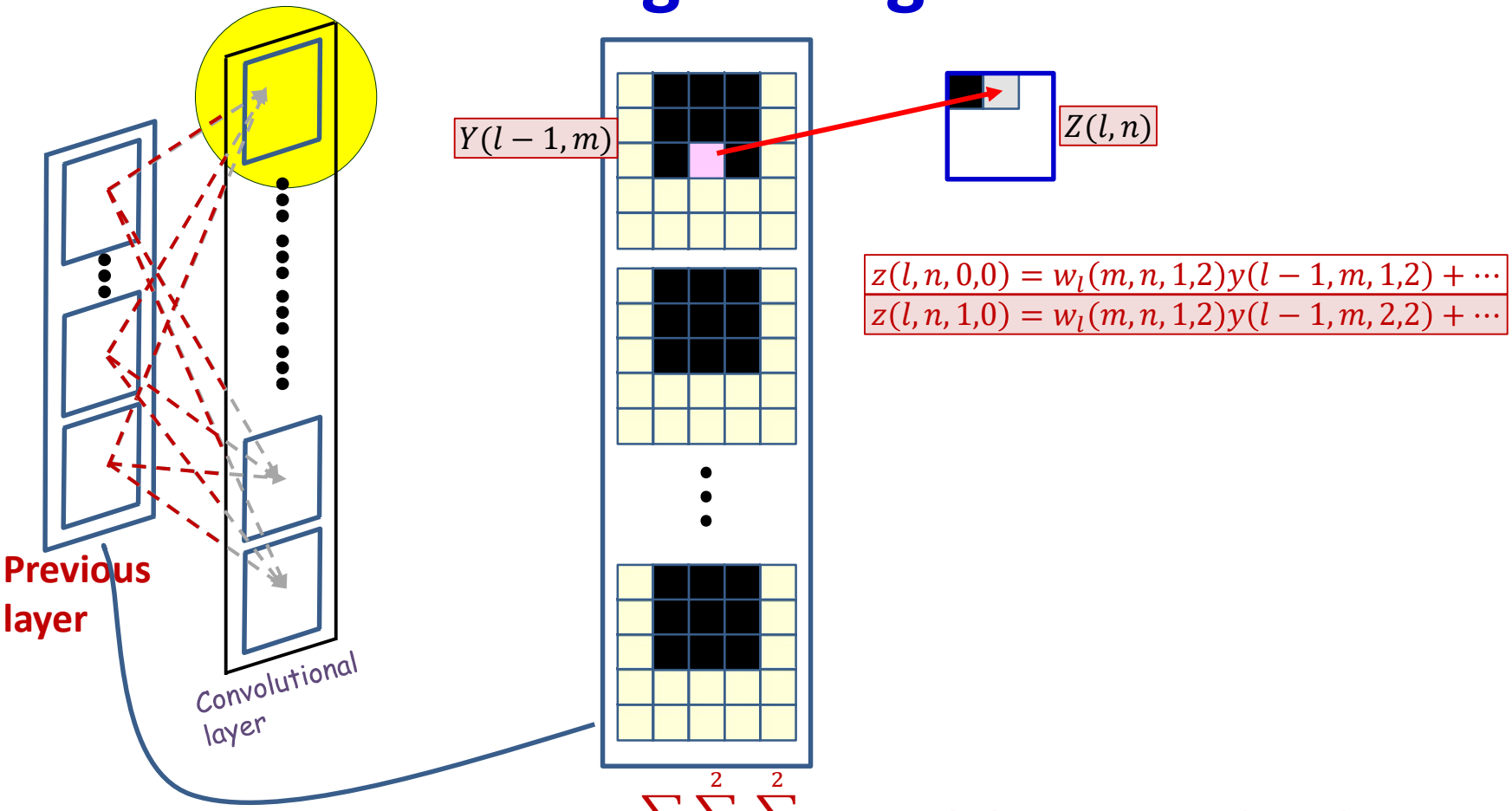
- Each affine output is computed from multiple input maps simultaneously
- Each ***weight*** $w_l(m,n,i,j)$ affects several $z(l,n,x,y)$

Convolution: the contribution of a single weight



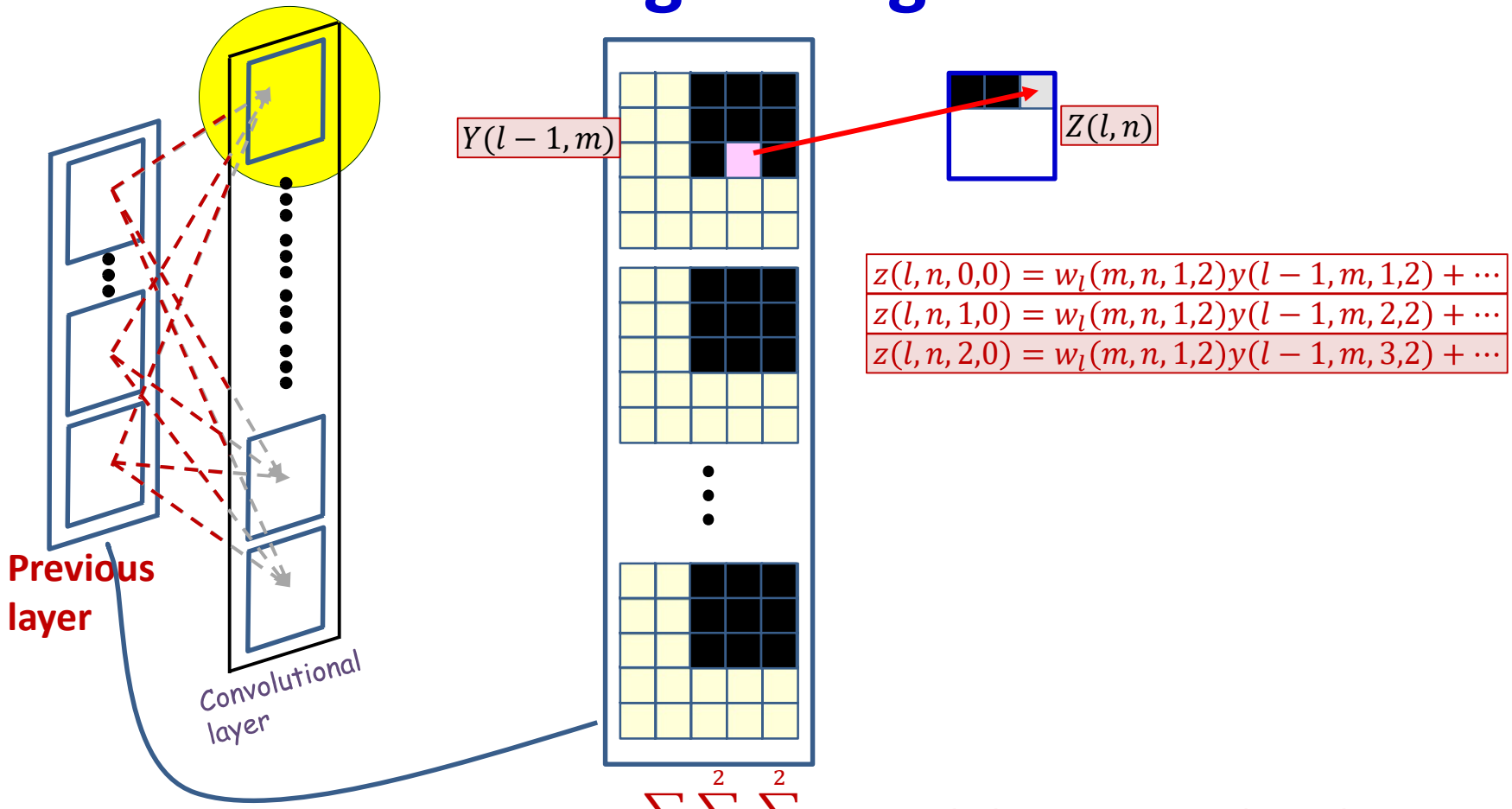
- Each weight $w_l(m,n,i,j)$ affects several $z(l,n,x,y)$
  - Consider the contribution of one filter components: e.g. $w_l(m,n,1,2)$

Convolution: the contribution of a single weight



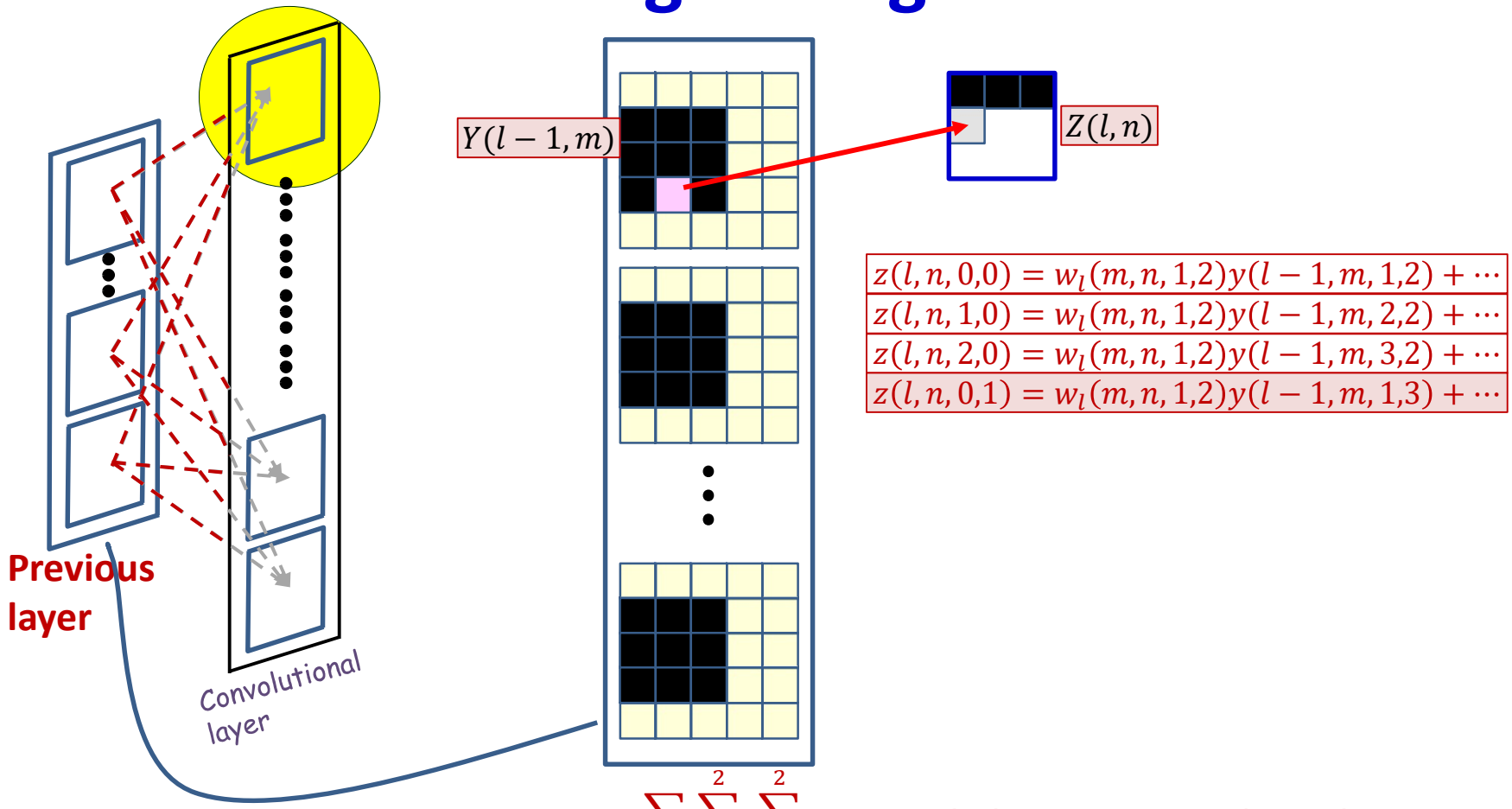
- Each weight $w_l(m,n,i,j)$ affects several $z(l,n,x,y)$
  - Consider the contribution of one filter components: e.g. $w_l(m,n,1,2)$

Convolution: the contribution of a single weight



- Each weight $w_l(m,n,i,j)$ affects several $z(l,n,x,y)$
  - Consider the contribution of one filter components: e.g. $w_l(m,n,1,2)$

Convolution: the contribution of a single weight

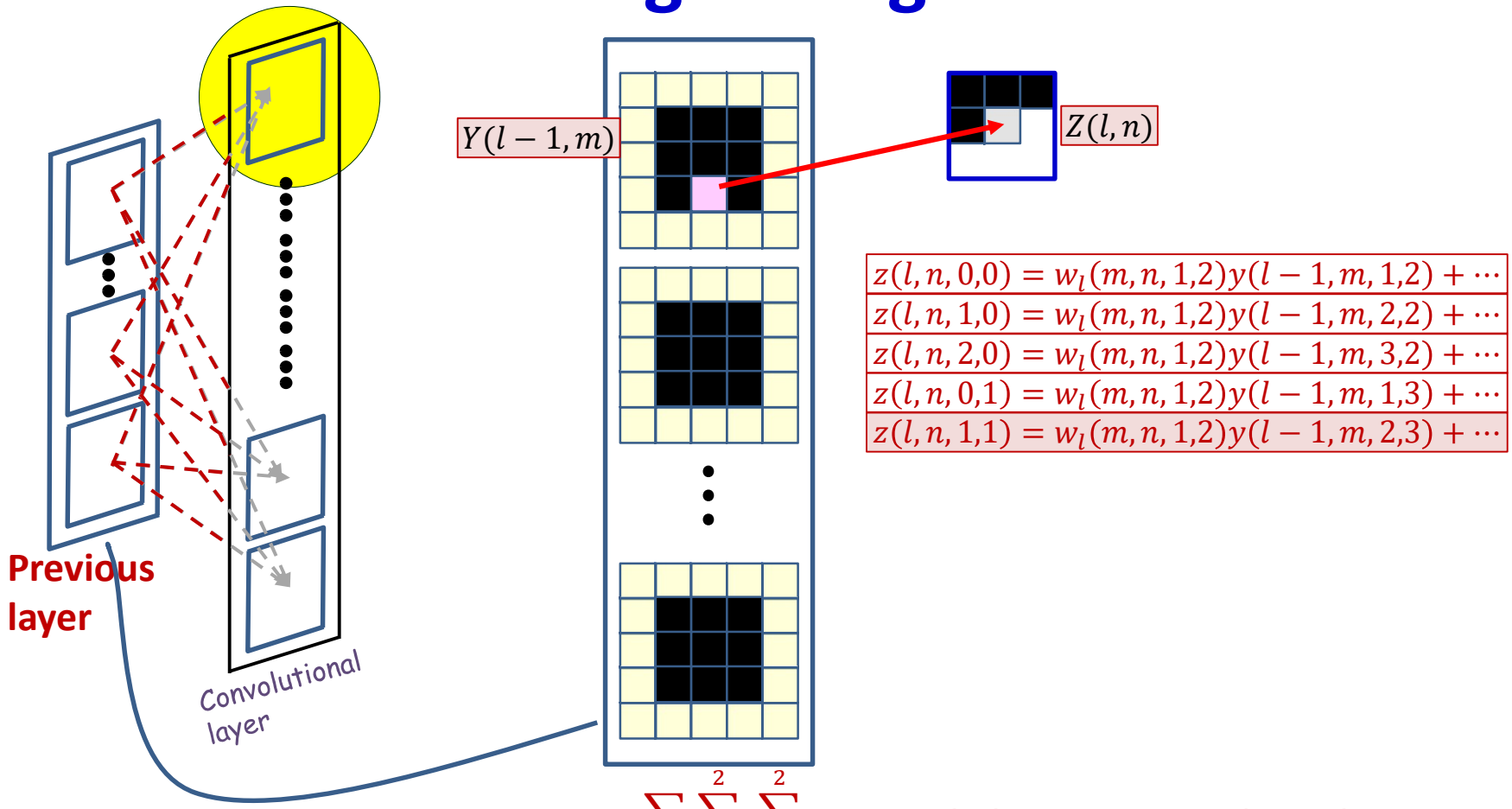


$$z(l,n,0,0) = w_l(m,n,1,2)y(l-1,m,1,2) + \cdots$$
$$z(l,n,1,0) = w_l(m,n,1,2)y(l-1,m,2,2) + \cdots$$
$$z(l,n,2,0) = w_l(m,n,1,2)y(l-1,m,3,2) + \cdots$$
$$z(l,n,0,1) = w_l(m,n,1,2)y(l-1,m,1,3) + \cdots$$

$$z(l,n,x,y) = \sum_m \sum_{x'=0}^{2} \sum_{y'=0}^{2} w_l(m,n,x',y')y(l-1,m,x+x',y+x') + b_l(n)$$

- Each weight $w_l(m,n,i,j)$ affects several $z(l,n,x,y)$
  - Consider the contribution of one filter components: e.g. $w_l(m,n,1,2)$

Convolution: the contribution of a single weight

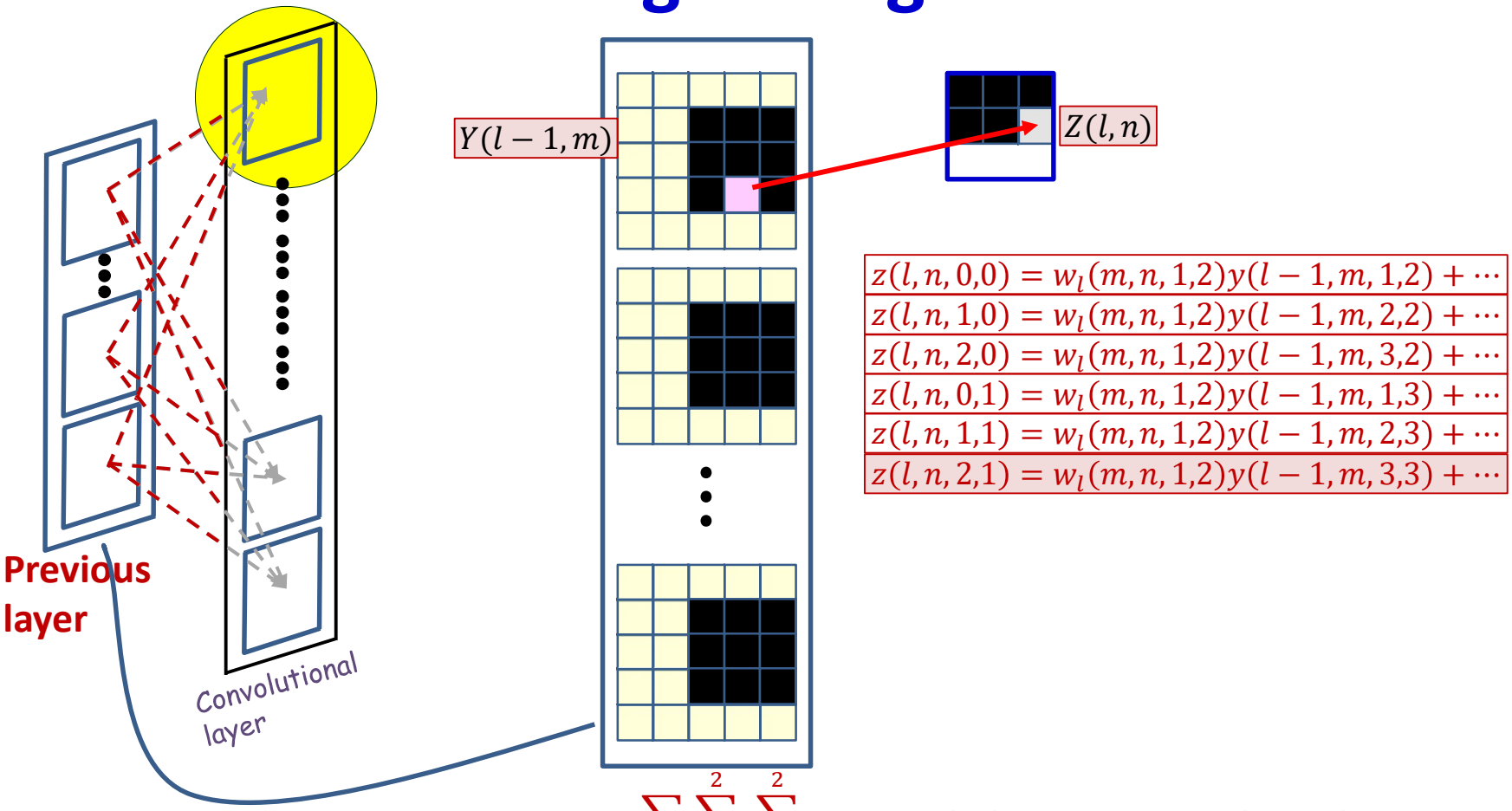


$$z(l,n,0,0) = w_l(m,n,1,2)y(l-1,m,1,2) + \cdots$$
$$z(l,n,1,0) = w_l(m,n,1,2)y(l-1,m,2,2) + \cdots$$
$$z(l,n,2,0) = w_l(m,n,1,2)y(l-1,m,3,2) + \cdots$$
$$z(l,n,0,1) = w_l(m,n,1,2)y(l-1,m,1,3) + \cdots$$
$$z(l,n,1,1) = w_l(m,n,1,2)y(l-1,m,2,3) + \cdots$$

$$z(l,n,x,y) = \sum_m \sum_{x'=0}^{2} \sum_{y'=0}^{2} w_l(m,n,x',y')y(l-1,m,x+x',y+x') + b_l(n)$$

- Each weight $w_l(m,n,i,j)$ affects several $z(l,n,x,y)$
  - Consider the contribution of one filter components: e.g. $w_l(m,n,1,2)$

Convolution: the contribution of a single weight

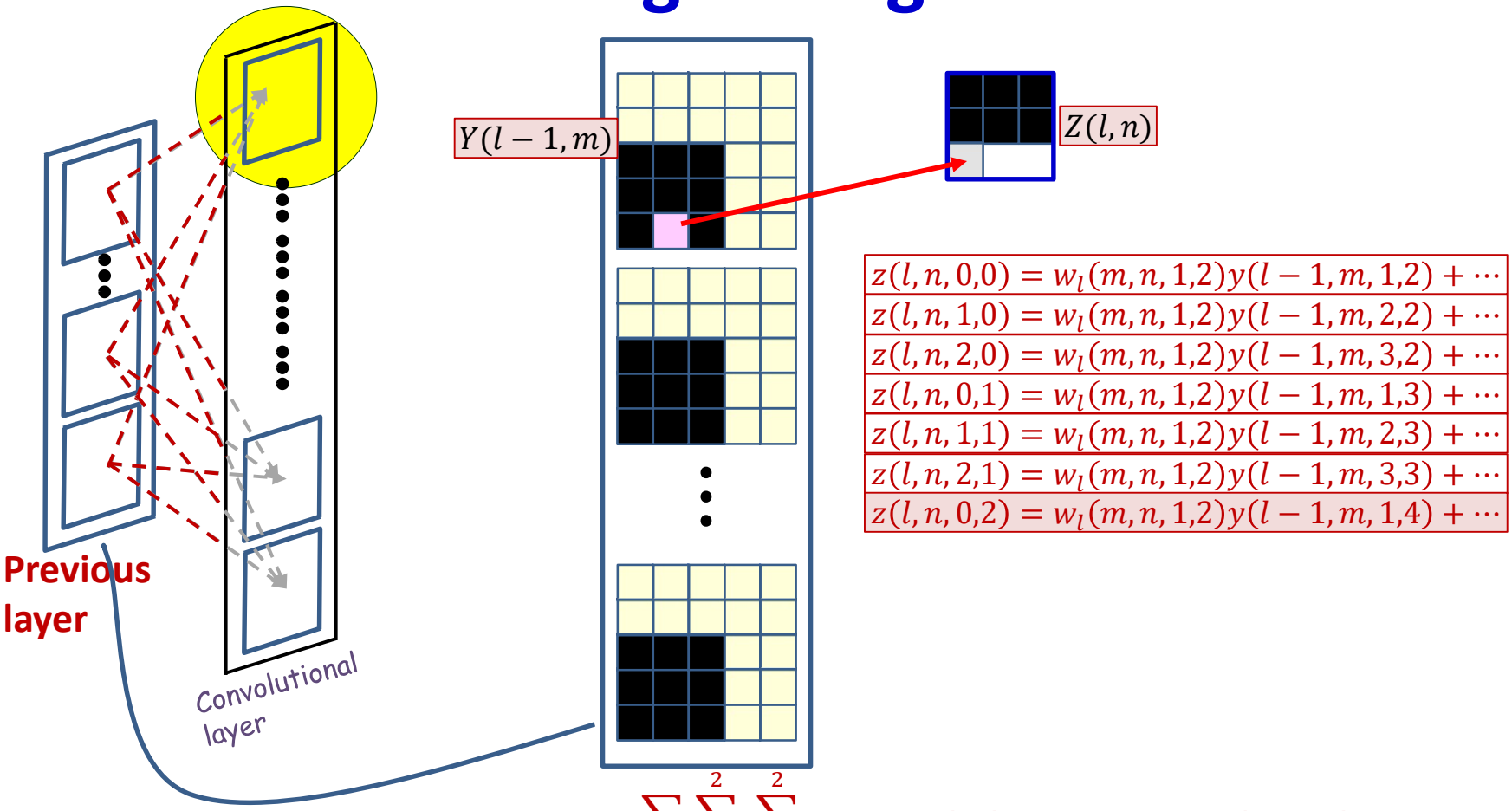


$$z(l,n,0,0) = w_l(m,n,1,2)y(l-1,m,1,2) + \cdots$$
$$z(l,n,1,0) = w_l(m,n,1,2)y(l-1,m,2,2) + \cdots$$
$$z(l,n,2,0) = w_l(m,n,1,2)y(l-1,m,3,2) + \cdots$$
$$z(l,n,0,1) = w_l(m,n,1,2)y(l-1,m,1,3) + \cdots$$
$$z(l,n,1,1) = w_l(m,n,1,2)y(l-1,m,2,3) + \cdots$$
$$z(l,n,2,1) = w_l(m,n,1,2)y(l-1,m,3,3) + \cdots$$

$$z(l,n,x,y) = \sum_m \sum_{x'=0}^{2} \sum_{y'=0}^{2} w_l(m,n,x',y')y(l-1,m,x+x',y+x') + b_l(n)$$

- Each weight $w_l(m,n,i,j)$ affects several $z(l,n,x,y)$
  - Consider the contribution of one filter components: e.g. $w_l(m,n,1,2)$

Convolution: the contribution of a single weight

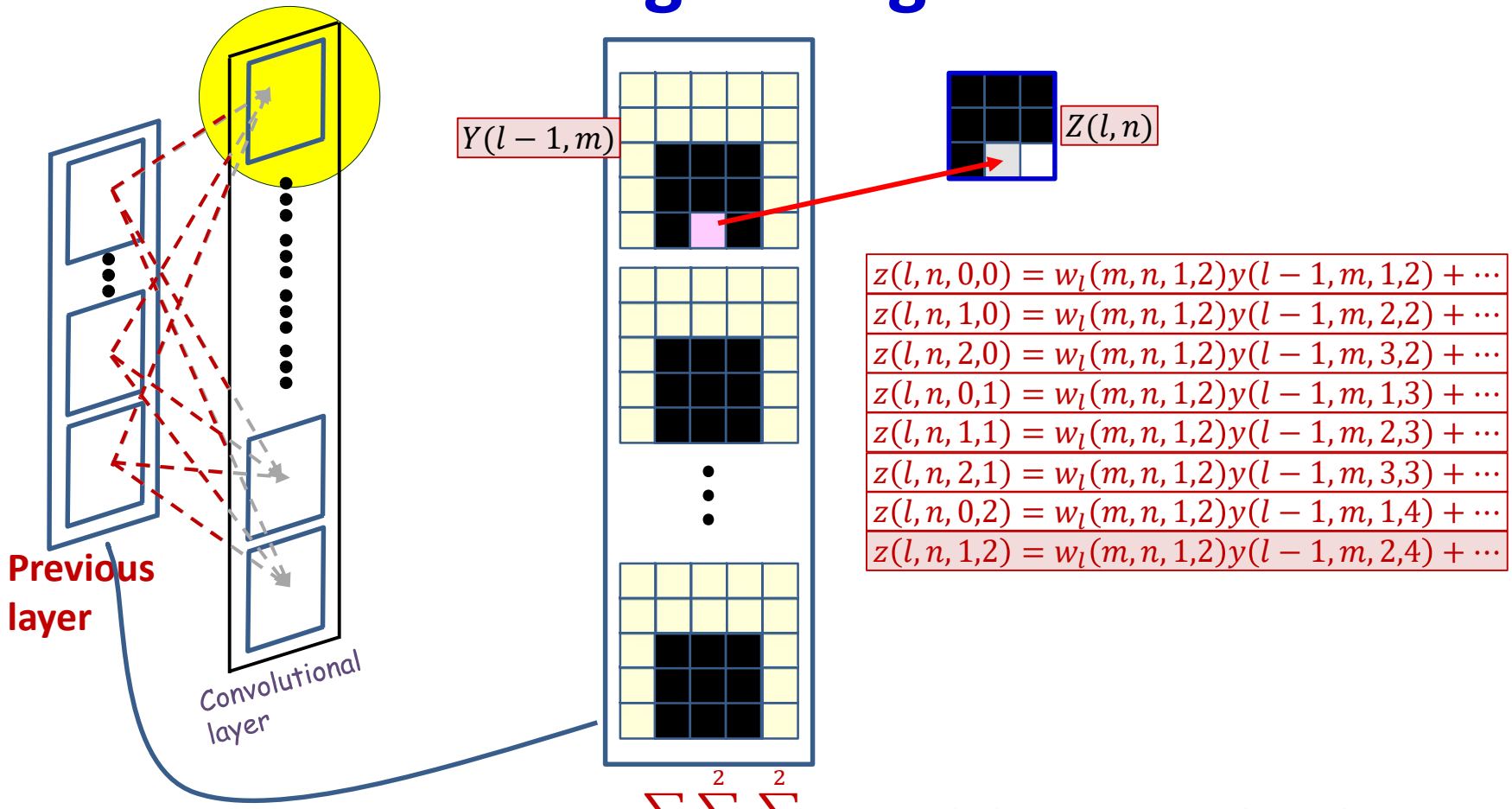


$$z(l,n,0,0) = w_l(m,n,1,2)y(l-1,m,1,2) + \cdots$$
$$z(l,n,1,0) = w_l(m,n,1,2)y(l-1,m,2,2) + \cdots$$
$$z(l,n,2,0) = w_l(m,n,1,2)y(l-1,m,3,2) + \cdots$$
$$z(l,n,0,1) = w_l(m,n,1,2)y(l-1,m,1,3) + \cdots$$
$$z(l,n,1,1) = w_l(m,n,1,2)y(l-1,m,2,3) + \cdots$$
$$z(l,n,2,1) = w_l(m,n,1,2)y(l-1,m,3,3) + \cdots$$
$$z(l,n,0,2) = w_l(m,n,1,2)y(l-1,m,1,4) + \cdots$$

$$z(l,n,x,y) = \sum_m \sum_{x'=0}^{2} \sum_{y'=0}^{2} w_l(m,n,x',y')y(l-1,m,x+x',y+x') + b_l(n)$$

- Each weight $w_l(m,n,i,j)$ affects several $z(l,n,x,y)$
  - Consider the contribution of one filter components: e.g. $w_l(m,n,1,2)$

Convolution: the contribution of a single weight

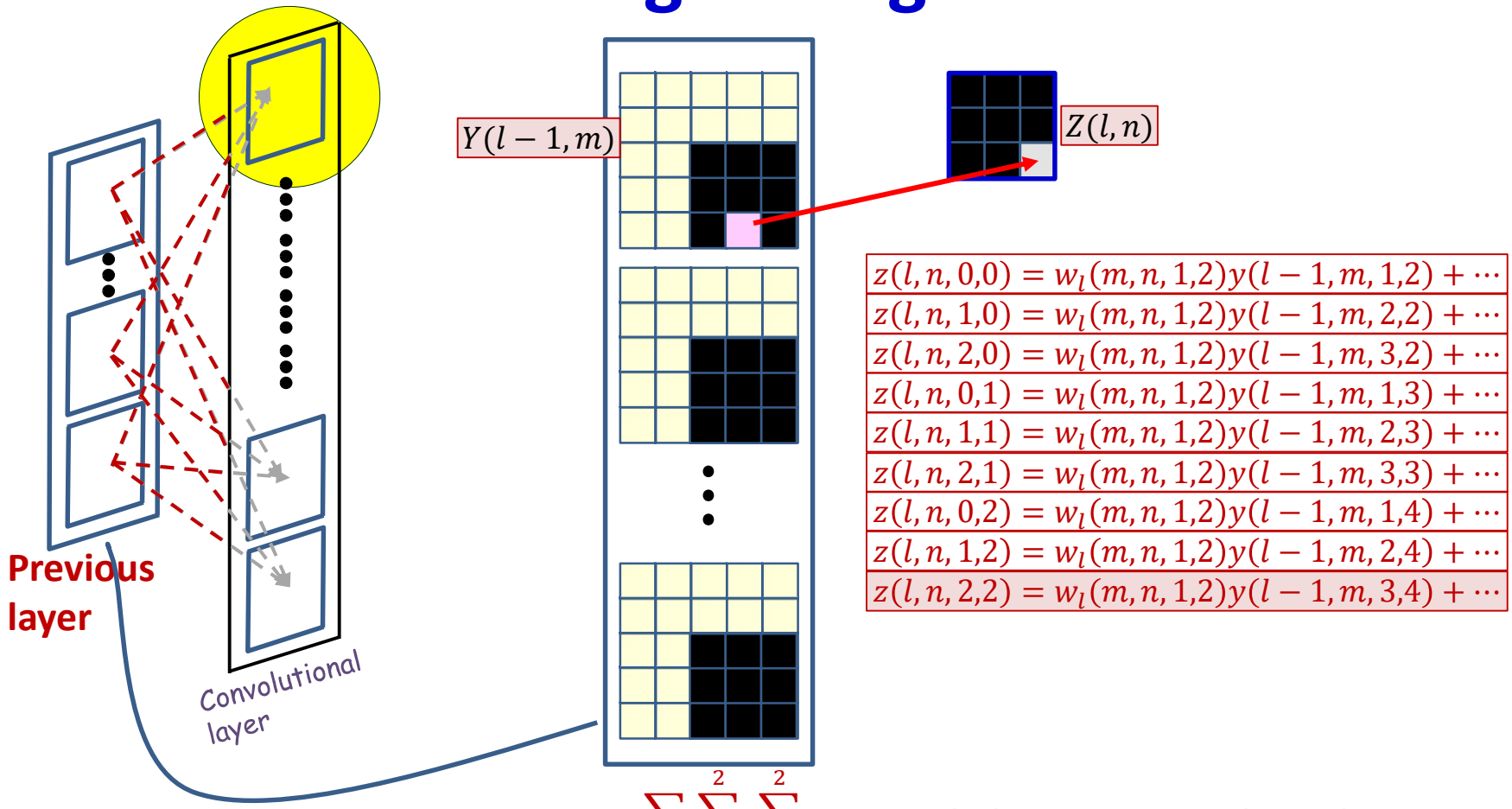


$$z(l,n,0,0) = w_l(m,n,1,2)y(l-1,m,1,2) + \cdots$$
$$z(l,n,1,0) = w_l(m,n,1,2)y(l-1,m,2,2) + \cdots$$
$$z(l,n,2,0) = w_l(m,n,1,2)y(l-1,m,3,2) + \cdots$$
$$z(l,n,0,1) = w_l(m,n,1,2)y(l-1,m,1,3) + \cdots$$
$$z(l,n,1,1) = w_l(m,n,1,2)y(l-1,m,2,3) + \cdots$$
$$z(l,n,2,1) = w_l(m,n,1,2)y(l-1,m,3,3) + \cdots$$
$$z(l,n,0,2) = w_l(m,n,1,2)y(l-1,m,1,4) + \cdots$$
$$z(l,n,1,2) = w_l(m,n,1,2)y(l-1,m,2,4) + \cdots$$

$$z(l,n,x,y) = \sum_m \sum_{x'=0}^{2} \sum_{y'=0}^{2} w_l(m,n,x',y')y(l-1,m,x+x',y+x') + b_l(n)$$

- Each weight $w_l(m,n,i,j)$ affects several $z(l,n,x,y)$
  - Consider the contribution of one filter components: e.g. $w_l(m,n,1,2)$

Convolution: the contribution of a single weight

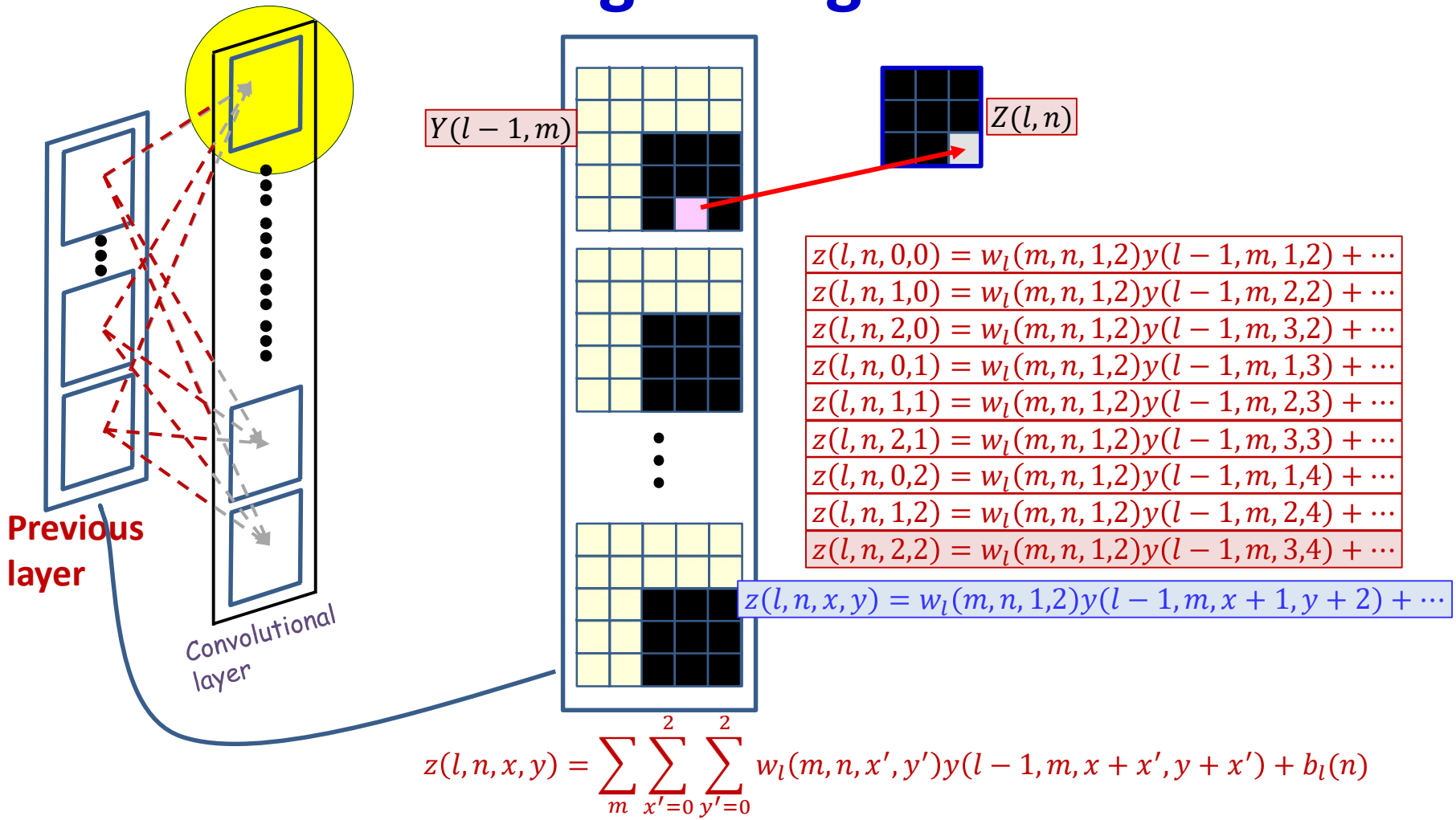


$$z(l,n,0,0) = w_l(m,n,1,2)y(l-1,m,1,2) + \cdots$$
$$z(l,n,1,0) = w_l(m,n,1,2)y(l-1,m,2,2) + \cdots$$
$$z(l,n,2,0) = w_l(m,n,1,2)y(l-1,m,3,2) + \cdots$$
$$z(l,n,0,1) = w_l(m,n,1,2)y(l-1,m,1,3) + \cdots$$
$$z(l,n,1,1) = w_l(m,n,1,2)y(l-1,m,2,3) + \cdots$$
$$z(l,n,2,1) = w_l(m,n,1,2)y(l-1,m,3,3) + \cdots$$
$$z(l,n,0,2) = w_l(m,n,1,2)y(l-1,m,1,4) + \cdots$$
$$z(l,n,1,2) = w_l(m,n,1,2)y(l-1,m,2,4) + \cdots$$
$$z(l,n,2,2) = w_l(m,n,1,2)y(l-1,m,3,4) + \cdots$$

$$z(l,n,x,y) = \sum_m \sum_{x'=0}^{2} \sum_{y'=0}^{2} w_l(m,n,x',y')y(l-1,m,x+x',y+x') + b_l(n)$$

- Each weight $w_l(m,n,i,j)$ affects several $z(l,n,x,y)$
  - Consider the contribution of one filter components: e.g. $w_l(m,n,1,2)$

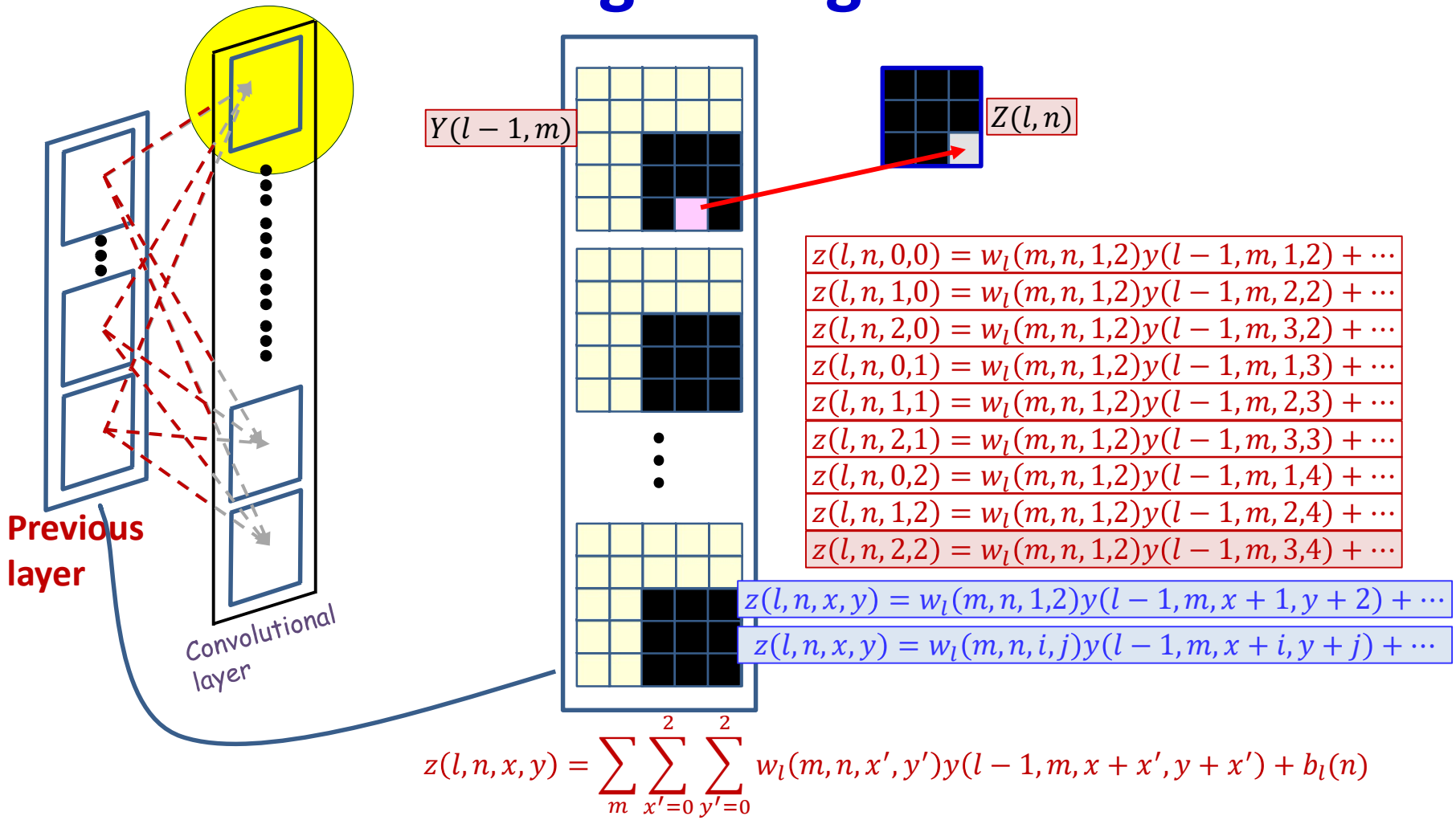
Convolution: the contribution of a single weight



$$z(l,n,x,y) = \sum_m \sum_{x'=0}^{2} \sum_{y'=0}^{2} w_l(m,n,x',y')y(l-1,m,x+x',y+x') + b_l(n)$$

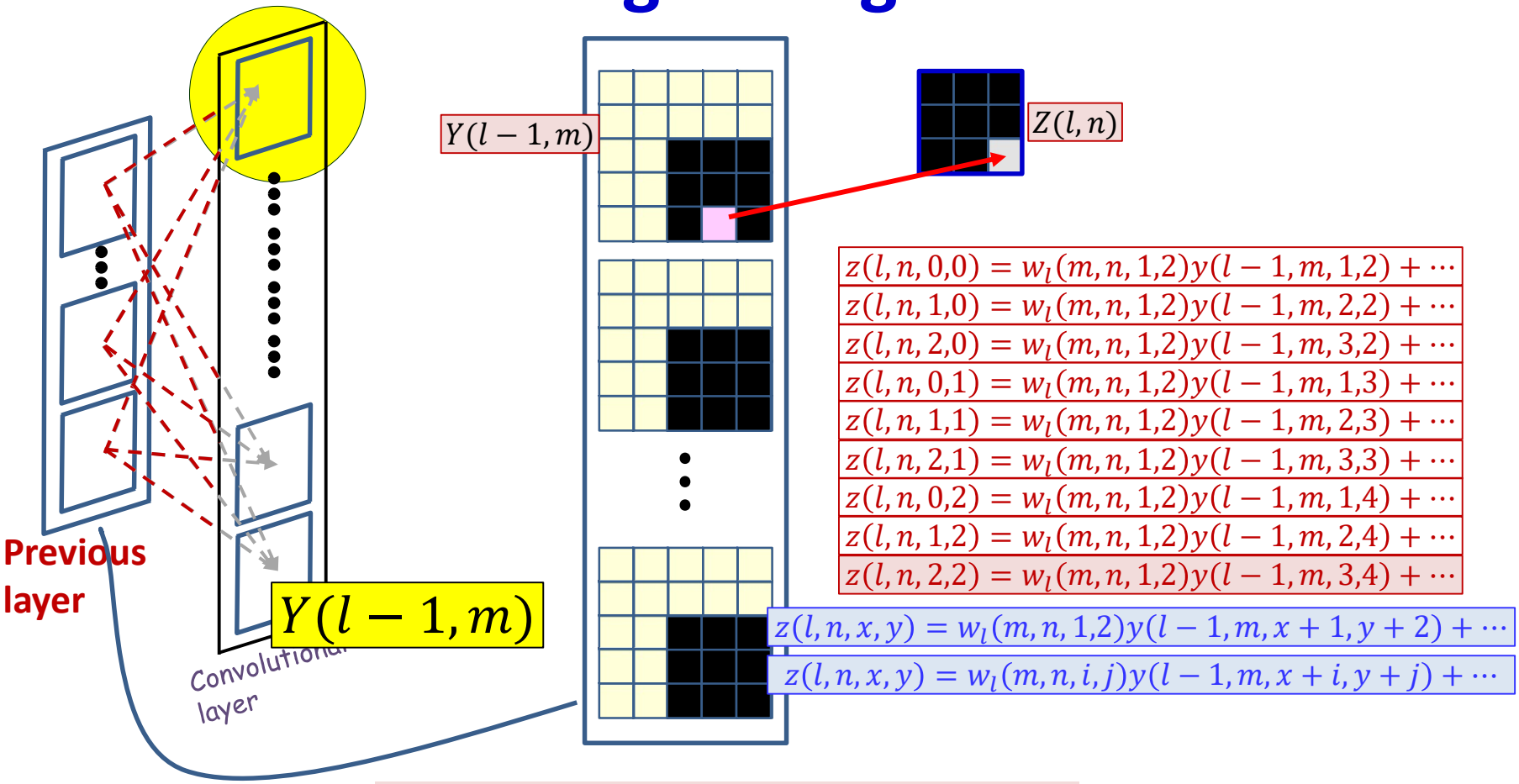
- Each weight $w_l(m,n,i,j)$ affects several $z(l,n,x,y)$
  - Consider the contribution of one filter components: e.g. $w_l(m,n,1,2)$

Convolution: the contribution of a single weight



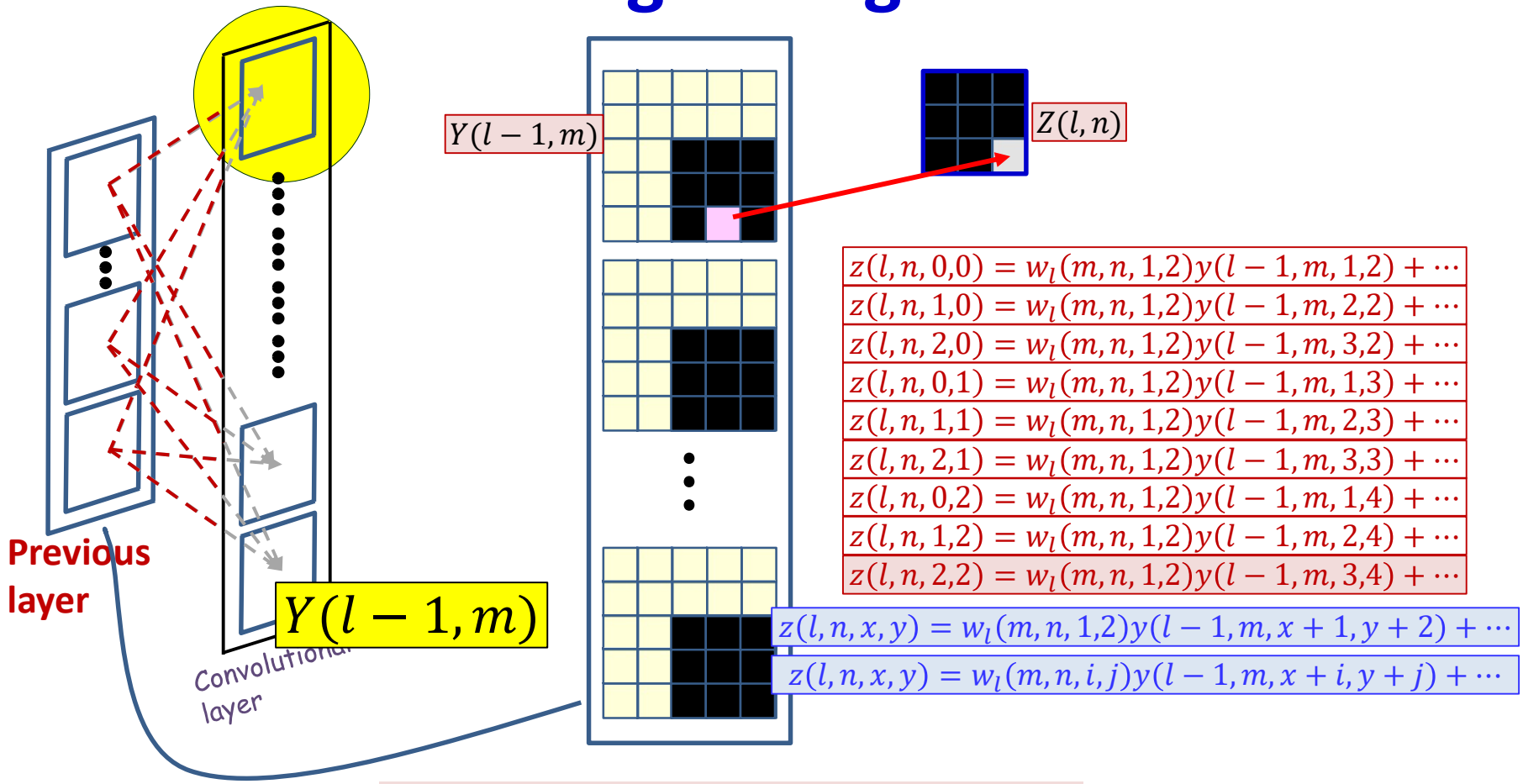
- Each weight $w_l(m,n,i,j)$ affects several $z(l,n,x,y)$
  - Consider the contribution of one filter components: e.g. $w_l(m,n,1,2)$

Convolution: the contribution of a single weight



$$\frac{dz(l,n,x,y)}{dw_l(m,n,i,j)} = y(l-1,m,x+i,y+j)$$

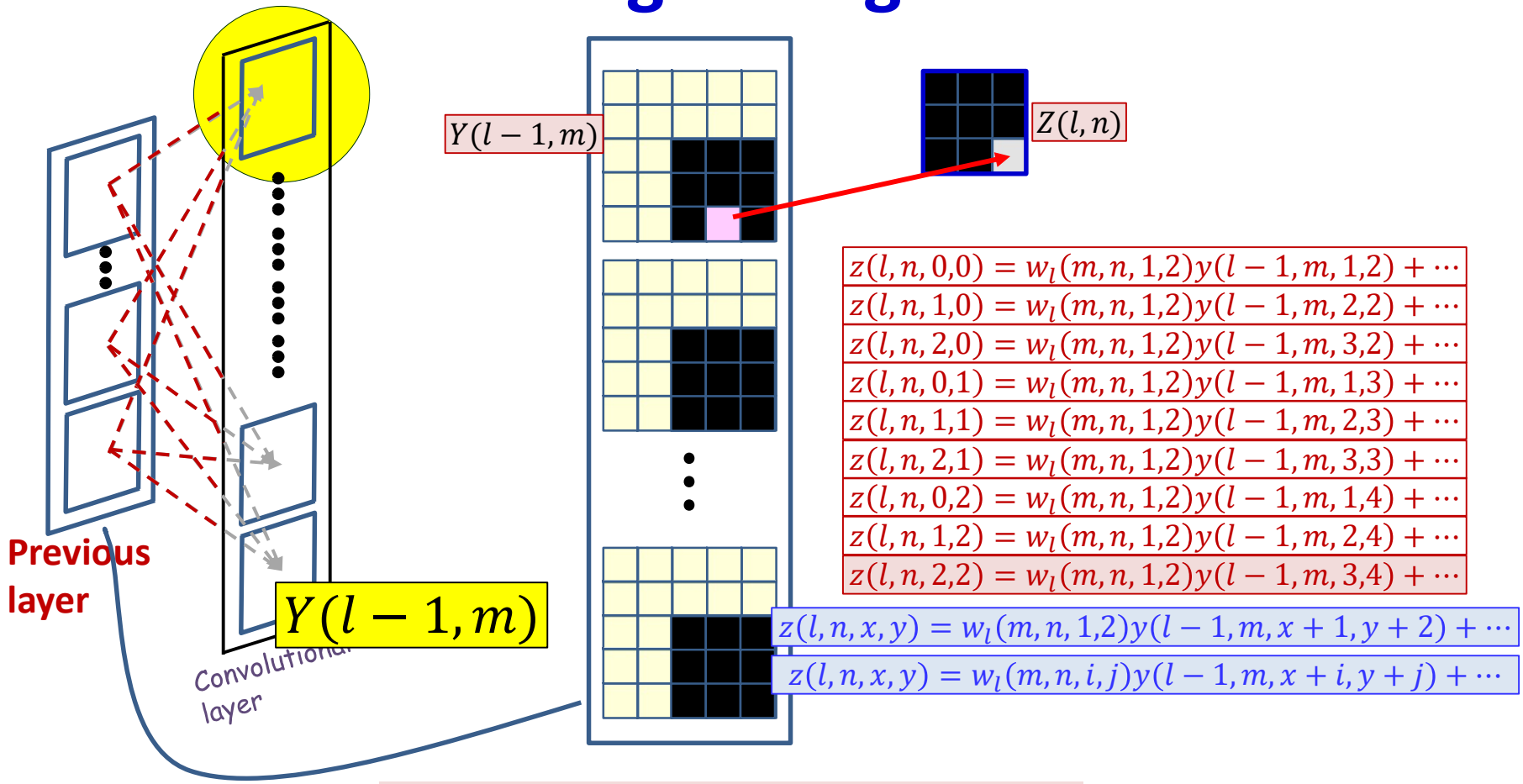
Convolution: the contribution of a single weight



$$\frac{dz(l,n,x,y)}{dw_l(m,n,i,j)} = y(l-1,m,x+i,y+j)$$

$$\frac{dDiv}{dw_l\,(m,n,i,j)} \mathrel{+}= \frac{dDiv}{dz(l,n,x,y)} \frac{dz(l,n,x,y)}{dw_l\,(m,n,i,j)}$$

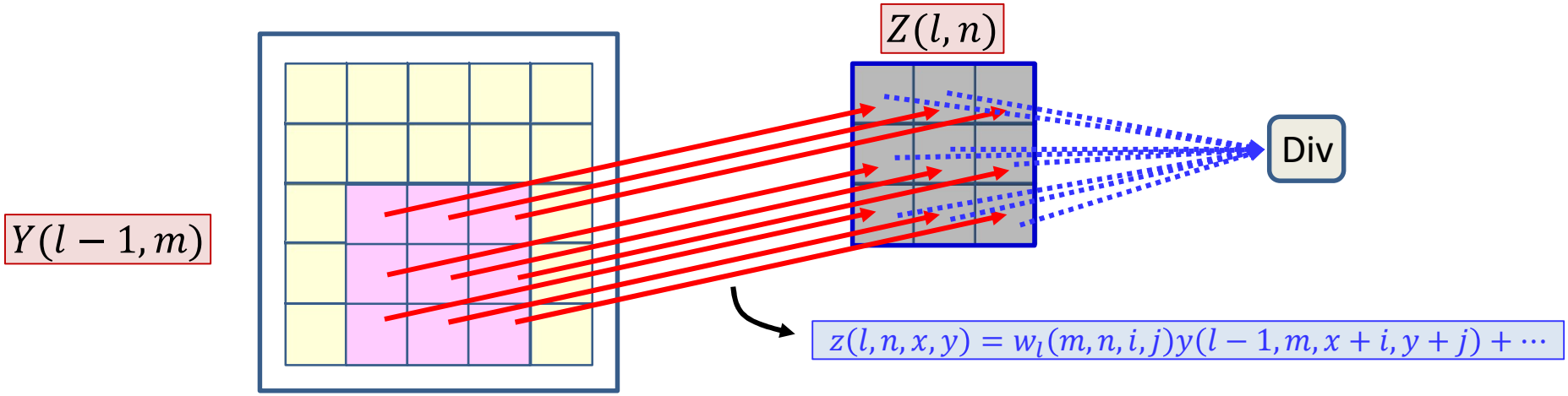
Convolution: the contribution of a single weight



$$\frac{dz(l,n,x,y)}{dw_l(m,n,i,j)} = y(l-1,m,x+i,y+j)$$

$$\frac{dDiv}{dw_l\,(m,n,i,j)} \mathrel{+}= \frac{dDiv}{dz(l,n,x,y)} y(l-1,m,x+i,y+j)$$

The derivative for a single weight



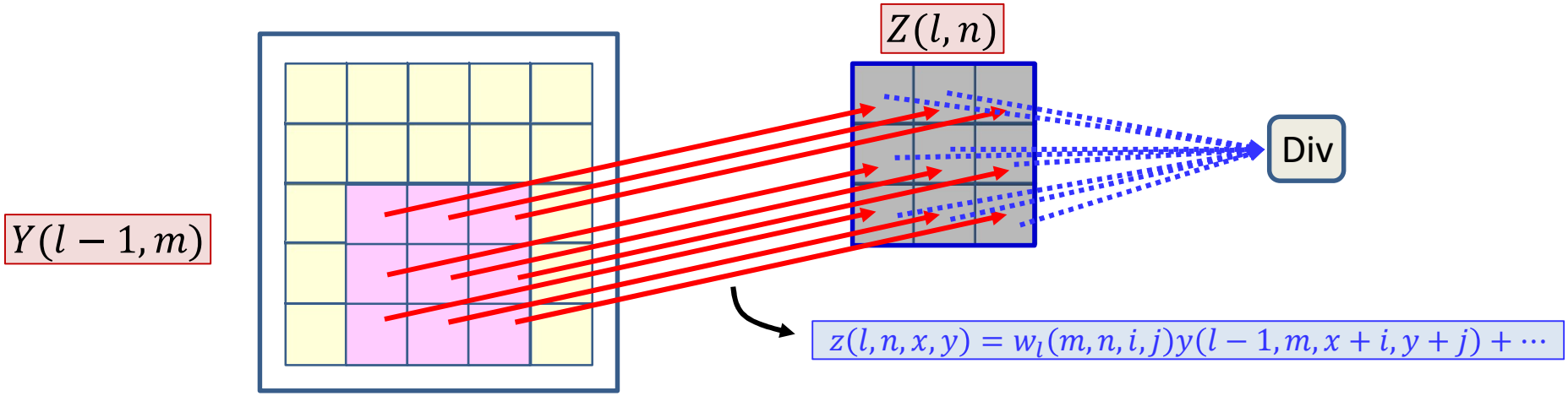
- Each filter component $w_l(m,n,i,j)$ affects several $z(l,n,x,y)$
  - The derivative of each $z(l,n,x,y)$ w.r.t. $w_l(m,n,i,j)$ is given by

$$\frac{dz(l,n,x,y)}{dw_l(m,n,i,j)} = y(l-1,m,x+i,y+j)$$

- The final divergence is influenced by *every* $z(l,n,x,y)$
- The derivative of the divergence w.r.t $w_l(m,n,i,j)$ must sum over all $z(l,n,x,y)$ terms it influences

$$\frac{dDiv}{dw_l(m,n,i,j)} = \sum_{x,y} \frac{dDiv}{dz(l,n,x,y)} y(l-1,m,x+i,y+j)$$

The derivative for a single weight



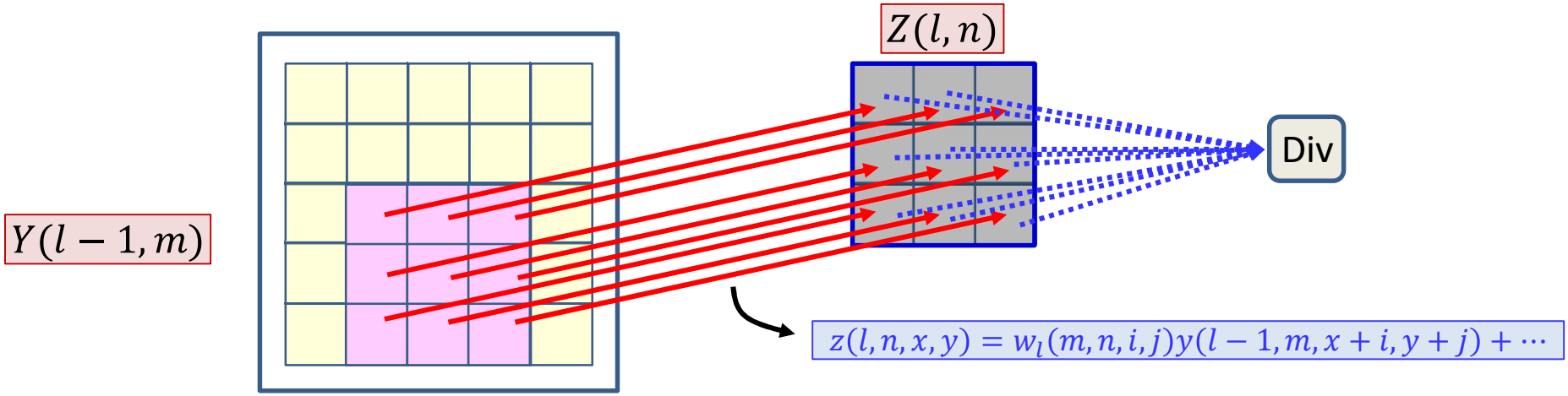
- Each filter component $w_l(m,n,i,j)$ affects several $z(l,n,x,y)$
  - The derivative of each $z(l,n,x,y)$ w.r.t. $w_l(m,n,i,j)$ is given by

$$\frac{dz(l,n,x,y)}{dw_l(m,n,i,j)} = y(l-1,m,x+i,y+j)$$

- The final divergence is influenced by *every* $z(l,n,x,y)$
- The derivati[Already computed]e w.r.t $w_l(m,n,i,j)$ must sum over all $z(l,n,x,y)$ terms it influences

$$\frac{dDiv}{dw_l(m,n,i,j)} = \sum_{x,y} \frac{dDiv}{dz(l,n,x,y)} y(l-1,m,x+i,y+j)$$

The derivative for a single weight



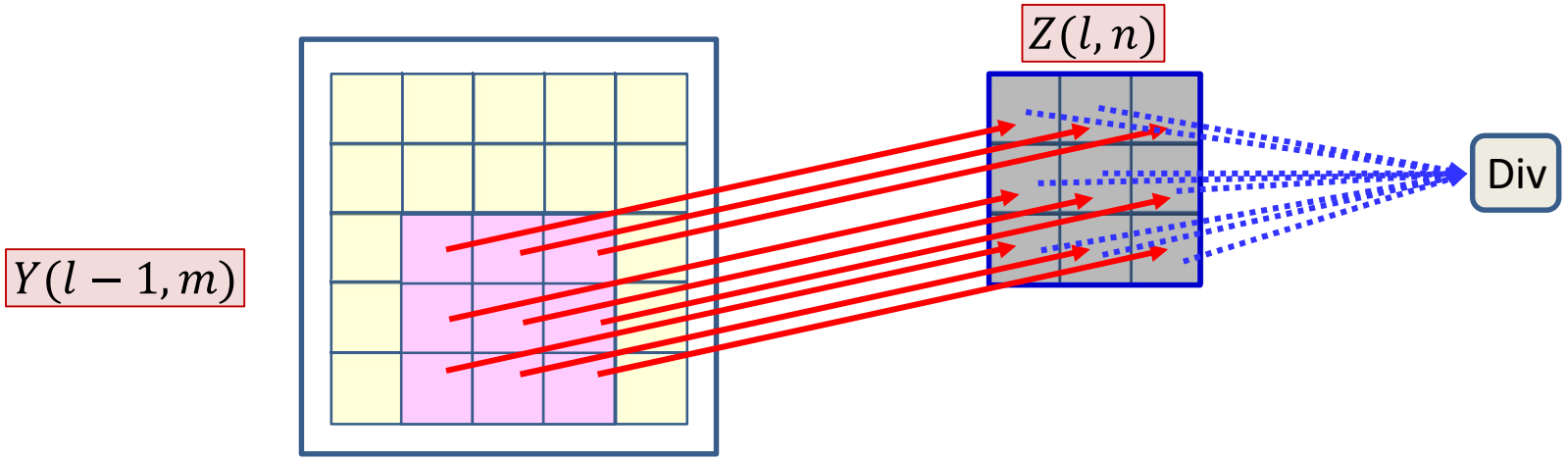
- Each filter component $w_l(m,n,i,j)$ affects several $z(l,n,x,y)$
  - The derivative of each $z(l,n,x,y)$ w.r.t. $w_l(m,n,i,j)$ is given by

$$\frac{dz(l,n,x,y)}{dw_l(m,n,i,j)} = y(l-1,m,x+i,y+j)$$

- The final divergence is influenced by *every* $z(l,n,x,y)$
- The derivati[Already computed]e w.r.t $w_l(m,n,i,j)$ must sum over all $z(l,n,x,y)$ terms it influences

$$\frac{dDiv}{dw_l(m,n,i,j)} = \sum_{x,y} \frac{dDiv}{dz(l,n,x,y)} y(l-1,m,x+i,y+j)$$

The derivative for a single weight



- Each filter component $w_l(m,n,i,j)$ affects several $z(l,n,x,y)$
  - The derivative of each $z(l,n,x,y)$ w.r.t. $w_l(m,n,i,j)$ is given by

$$\frac{dz(l,n,x,y)}{dw_l(m,n,i,j)} = y(l-1,m,x+i,y+j)$$

- The final divergence is influenced by *every* $z(l,n,x,y)$
- The derivative of the divergence w.r.t $w_l(m,n,i,j)$ must sum over all $z(l,n,x,y)$ terms it influences

$$\frac{dDiv}{dw_l(m,n,i,j)} = \sum_{x,y} \frac{dDiv}{dz(l,n,x,y)} y(l-1,m,x+i,y+j)$$

But this too is a convolution

$$\frac{dDiv}{dw_l\,(m,n,i,j)} = \sum_{x,y} \frac{dDiv}{dz(l,n,x,y)} y(l-1,m,x+i,y+j)$$

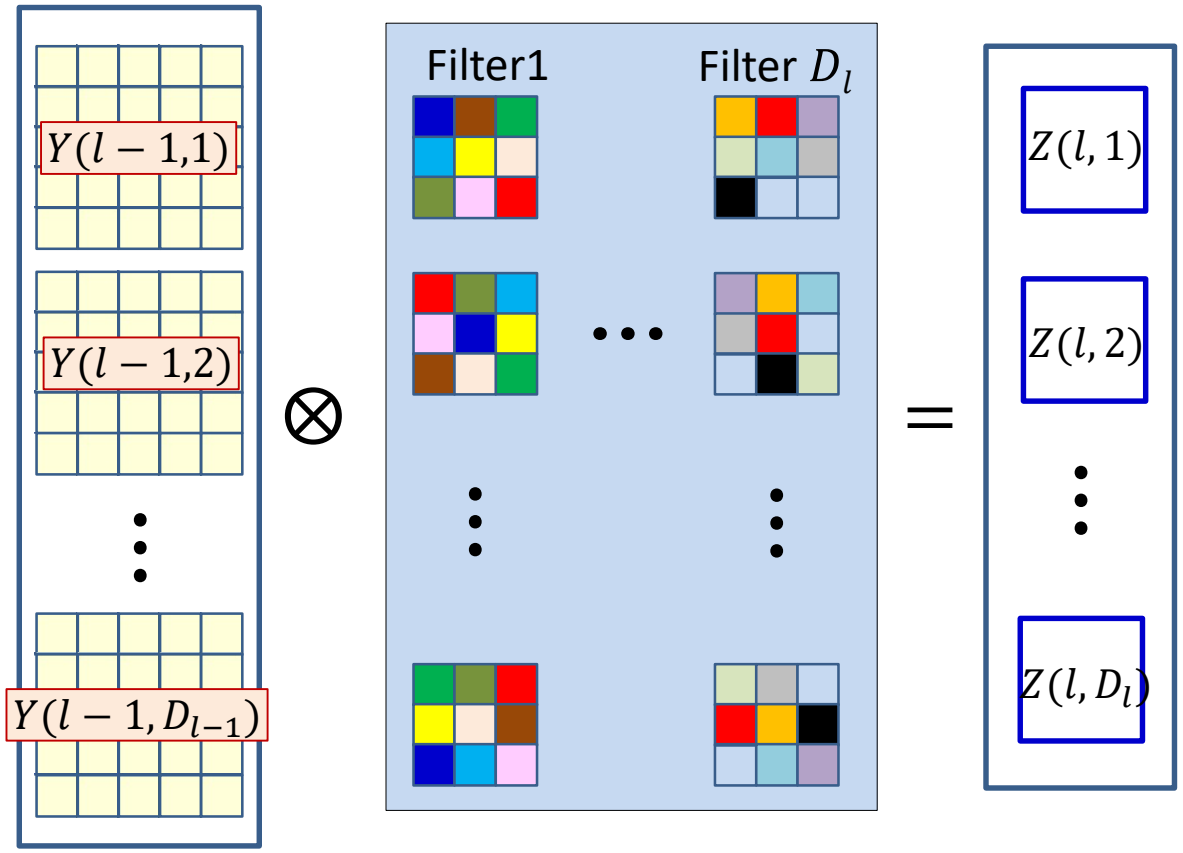
- The derivatives for all components of all filters can be computed directly from the above formula

- In fact it is just a convolution

$$\frac{dDiv}{dw_l\,(m,n,i,j)} = \frac{dDiv}{dz(l,n)} \otimes y(l-1,m)$$

- How?

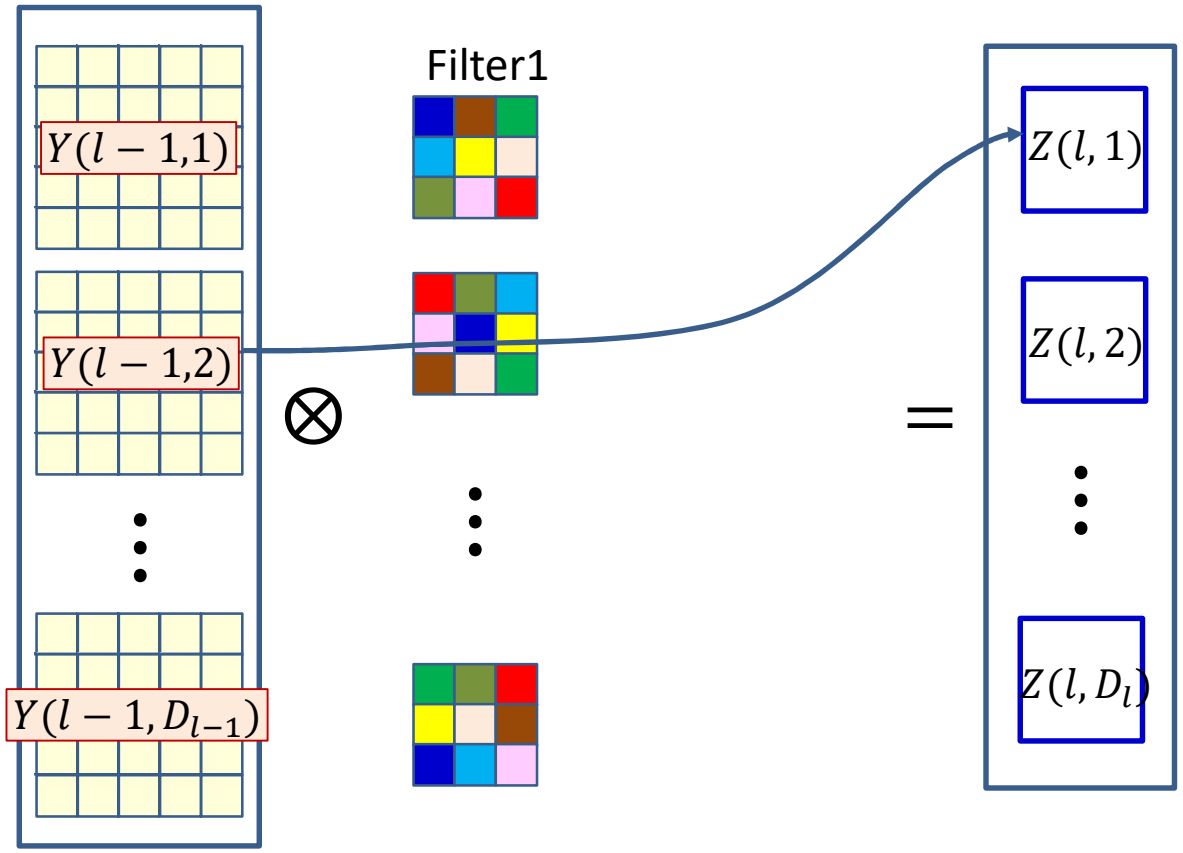
Recap: Convolution



$$z(l,n,x,y) = \sum_{m} \sum_{i=0}^{2} \sum_{j=0}^{2} w_l(m,n,i,j) y(l-1,m,x+i,y+j) + b_l(n)$$

- Forward computation: Each filter produces an affine map

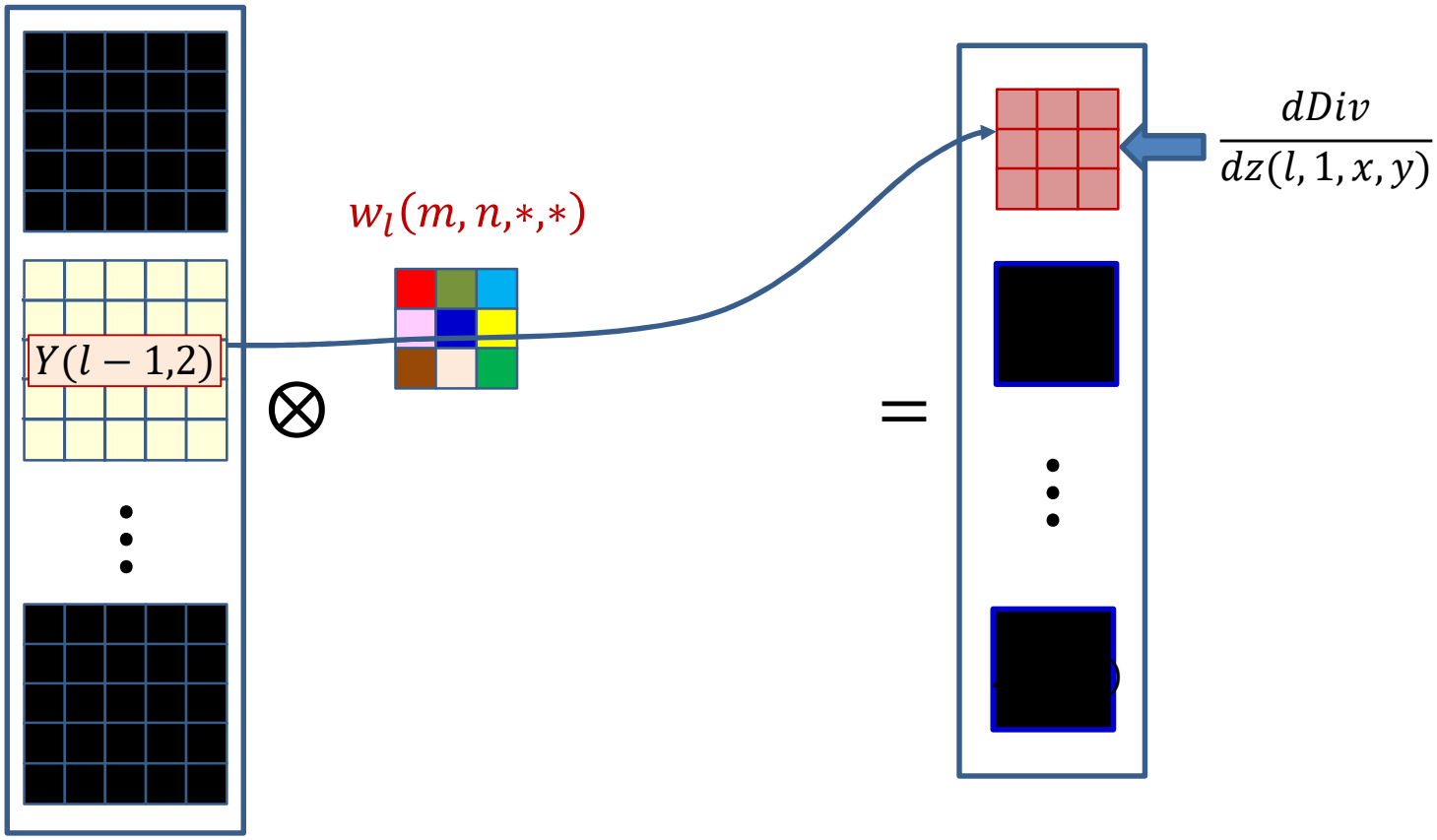
Recap: Convolution



$$z(l,n,x,y) = \sum_{m} \sum_{i=0}^{2} \sum_{j=0}^{2} w_l(m,n,i,j) y(l-1,m,x+i,y+j) + b_l(n)$$

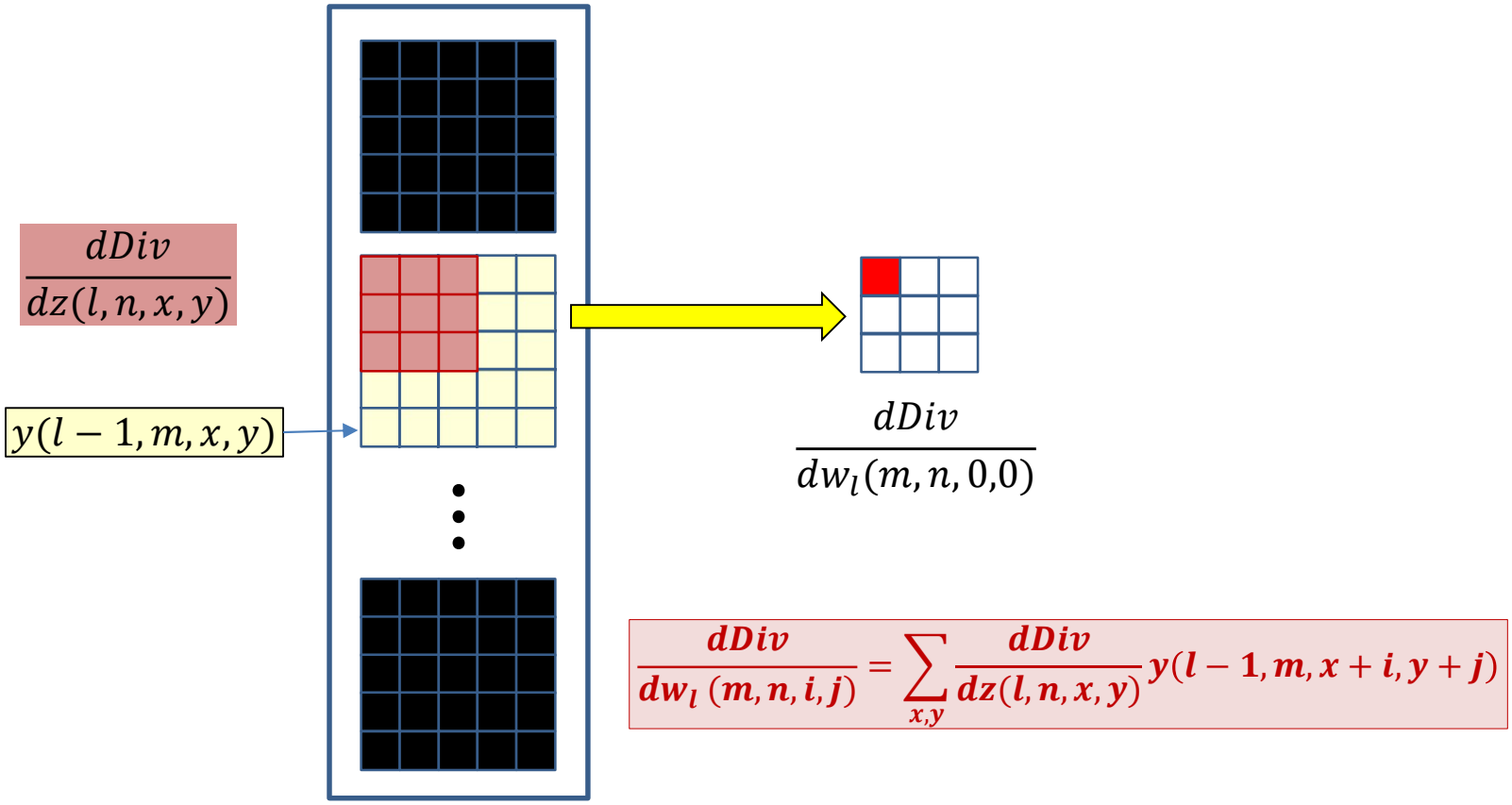
- $Y(l-1,m)$ influences $Z(l,n)$ through $w_l(m,n)$

The filter derivative



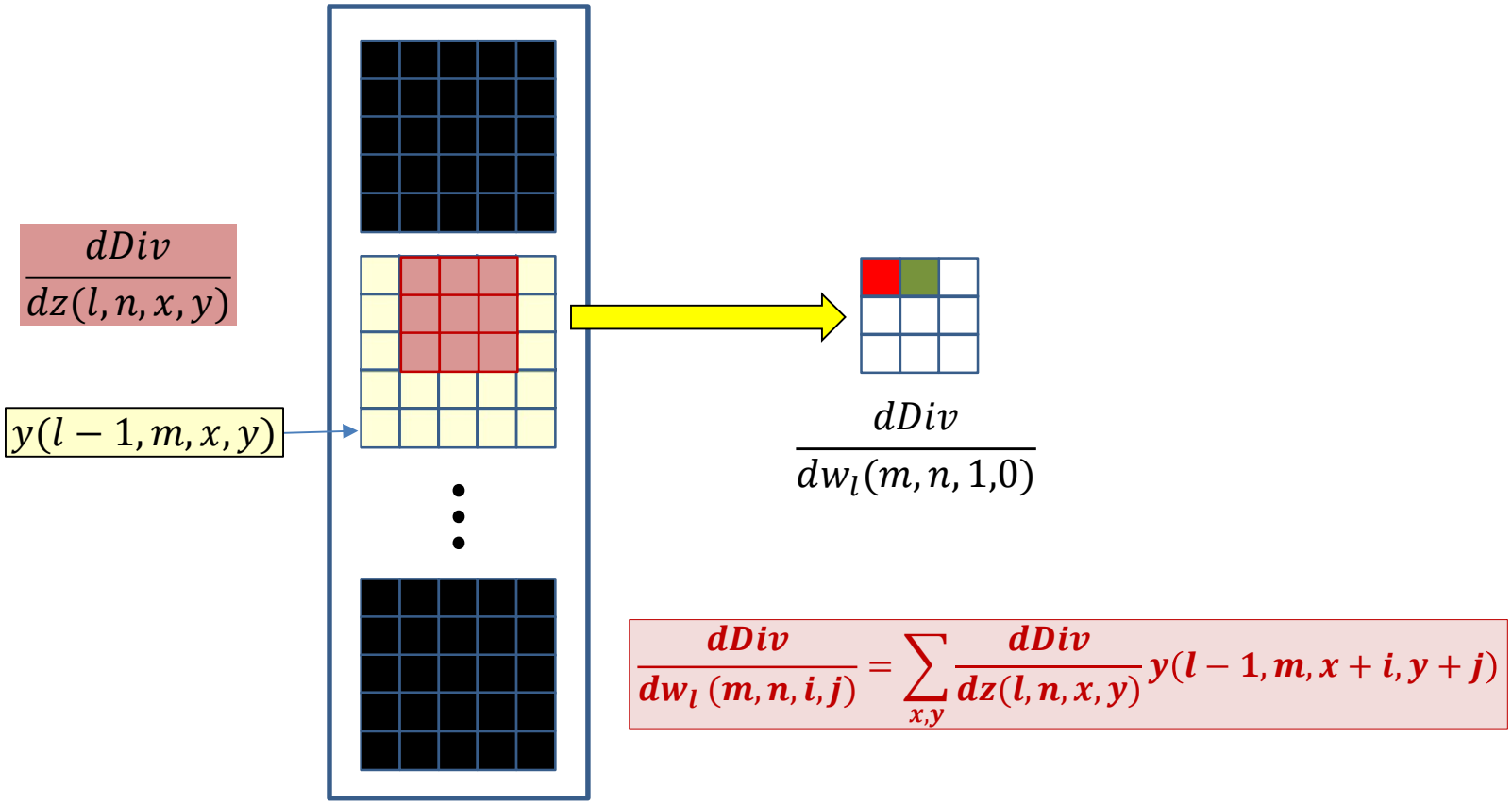
- The derivatives of the divergence w.r.t. every element of $Z(l,n)$ is known
  - Must use them to compute the derivative for $w_l(m,n,*,*)$

The filter derivative



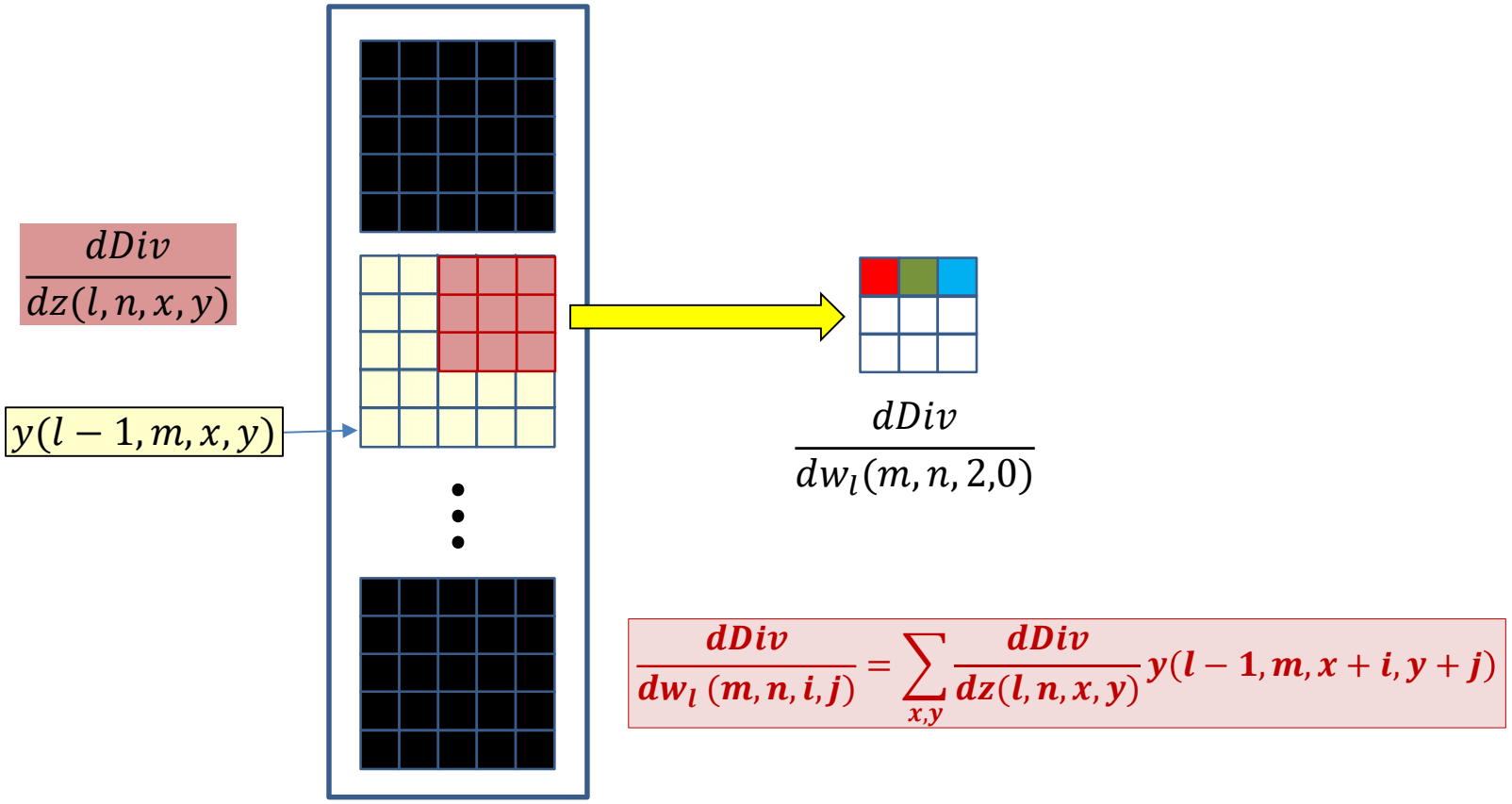
- The derivatives of the divergence w.r.t. every element of $Z(l,n)$ is known
  - Must use them to compute the derivative for $w_l(m,n,*,*)$

The filter derivative



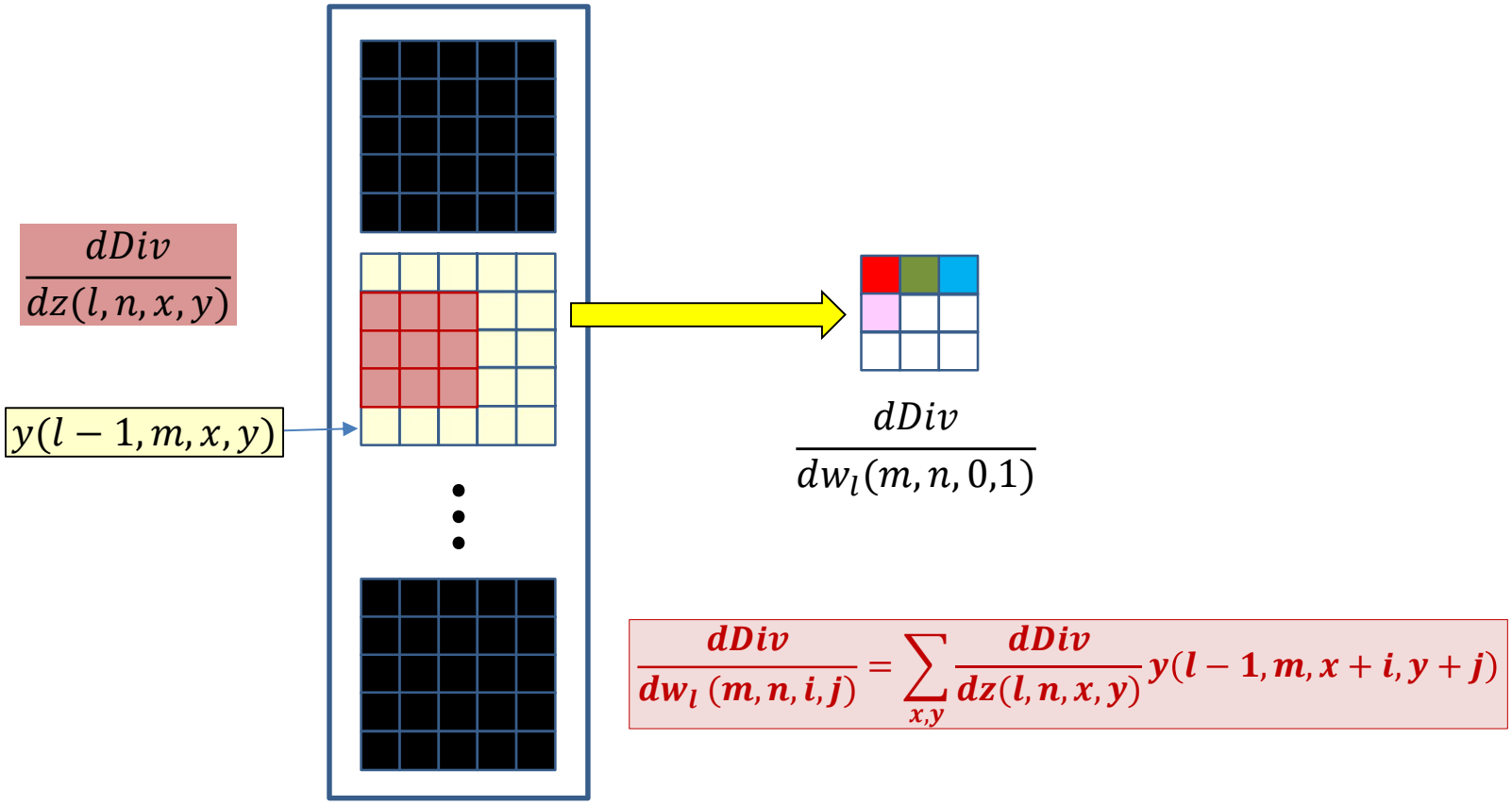
- The derivatives of the divergence w.r.t. every element of $Z(l,n)$ is known
  - Must use them to compute the derivative for $w_l(m,n,*,*)$

The filter derivative



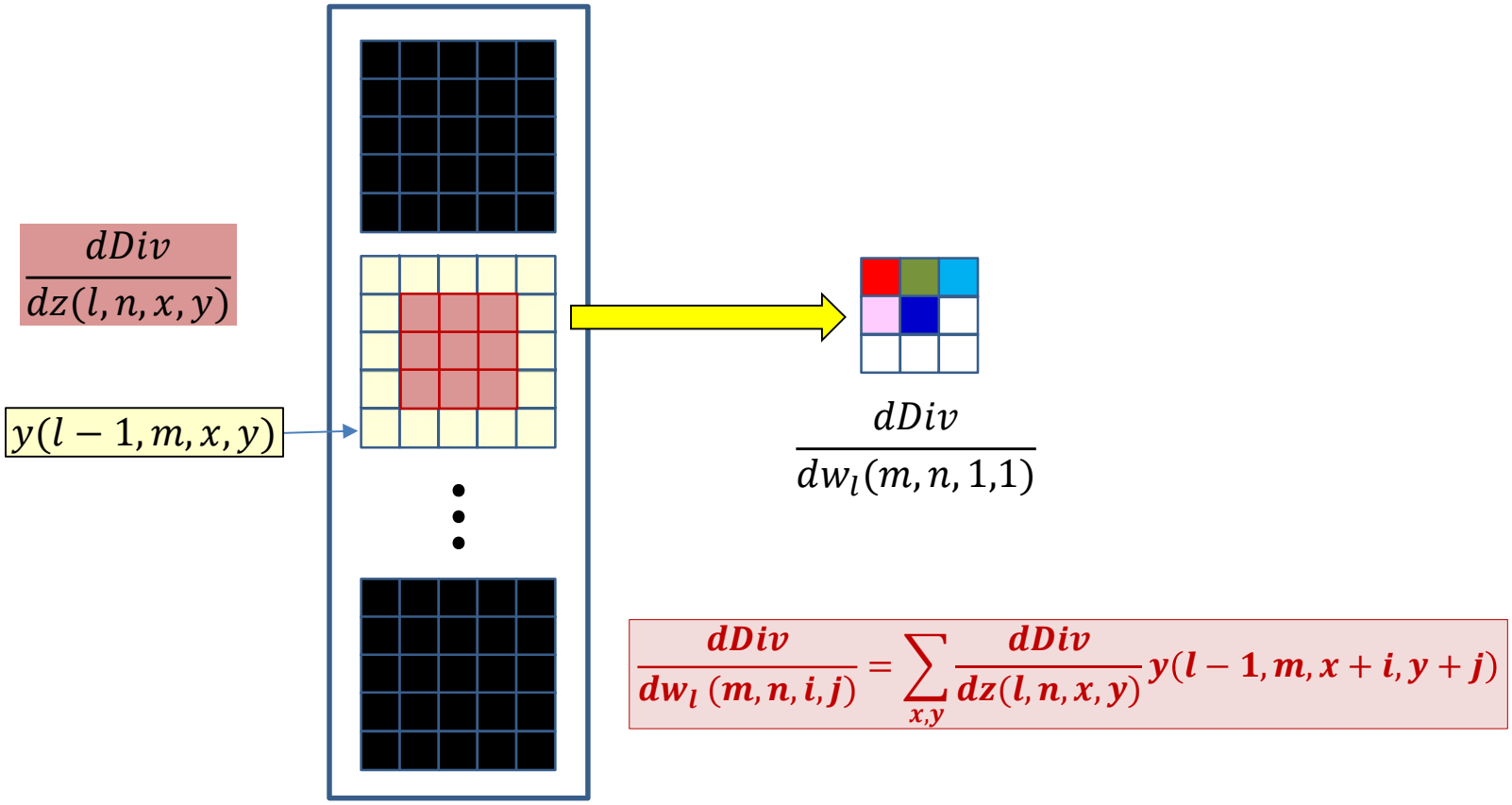
- The derivatives of the divergence w.r.t. every element of $Z(l,n)$ is known
  - Must use them to compute the derivative for $w_l(m,n,*,*)$

The filter derivative



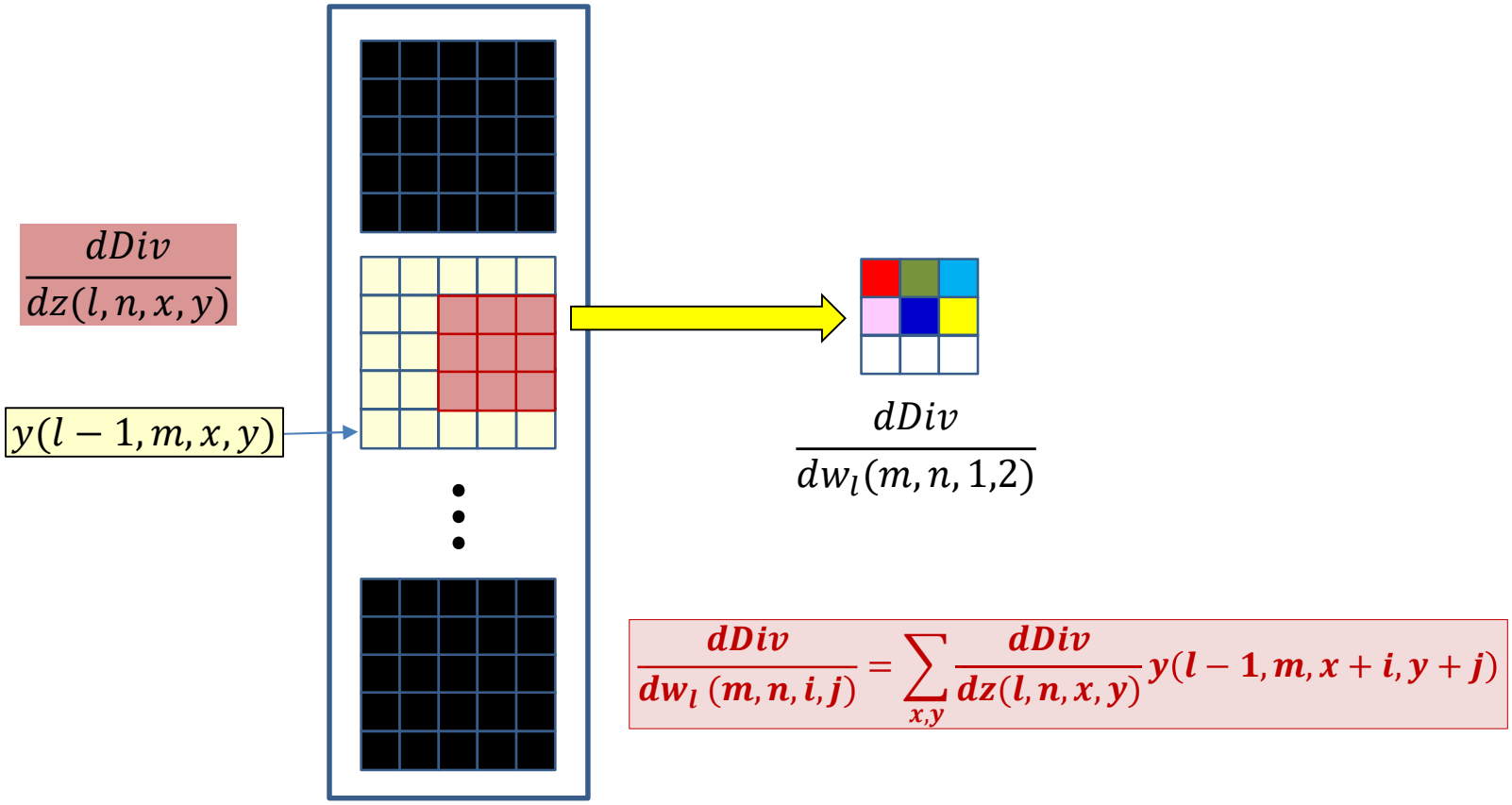
- The derivatives of the divergence w.r.t. every element of $Z(l,n)$ is known
  - Must use them to compute the derivative for $w_l(m,n,*,*)$

The filter derivative



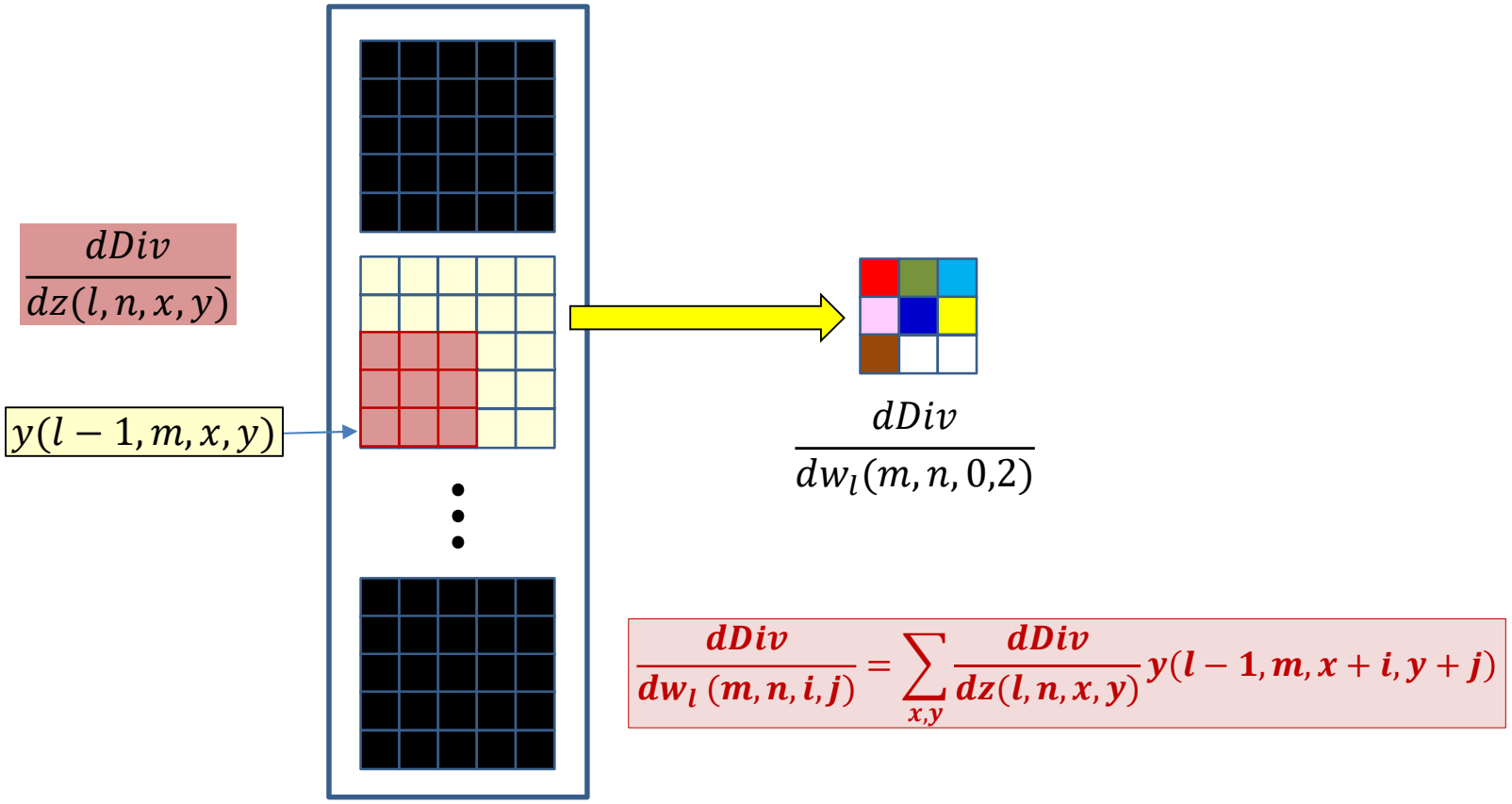
- The derivatives of the divergence w.r.t. every element of $Z(l,n)$ is known
  - Must use them to compute the derivative for $w_l(m,n,*,*)$

The filter derivative



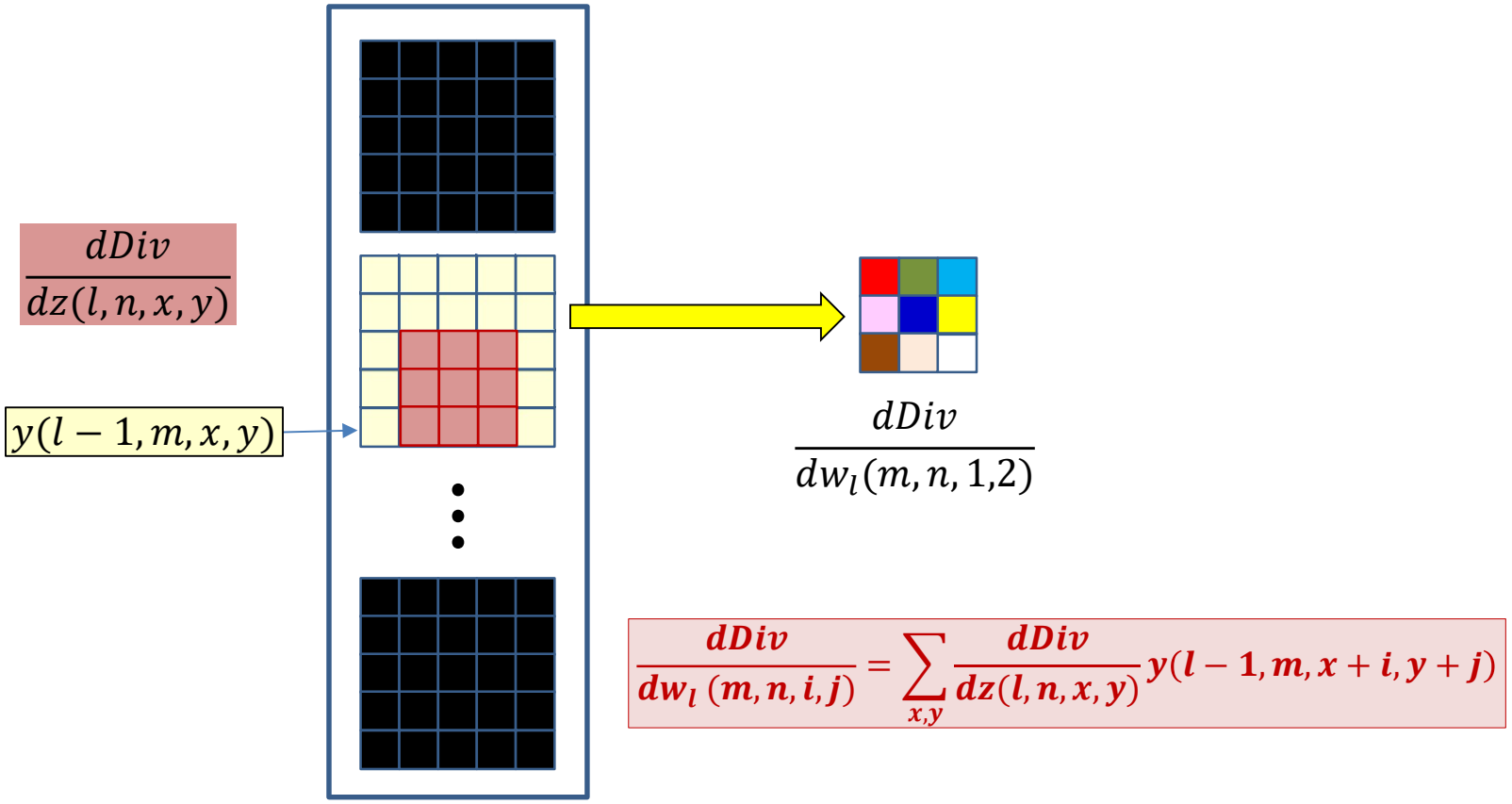
- The derivatives of the divergence w.r.t. every element of $Z(l,n)$ is known
  - Must use them to compute the derivative for $w_l(m,n,*,*)$

The filter derivative



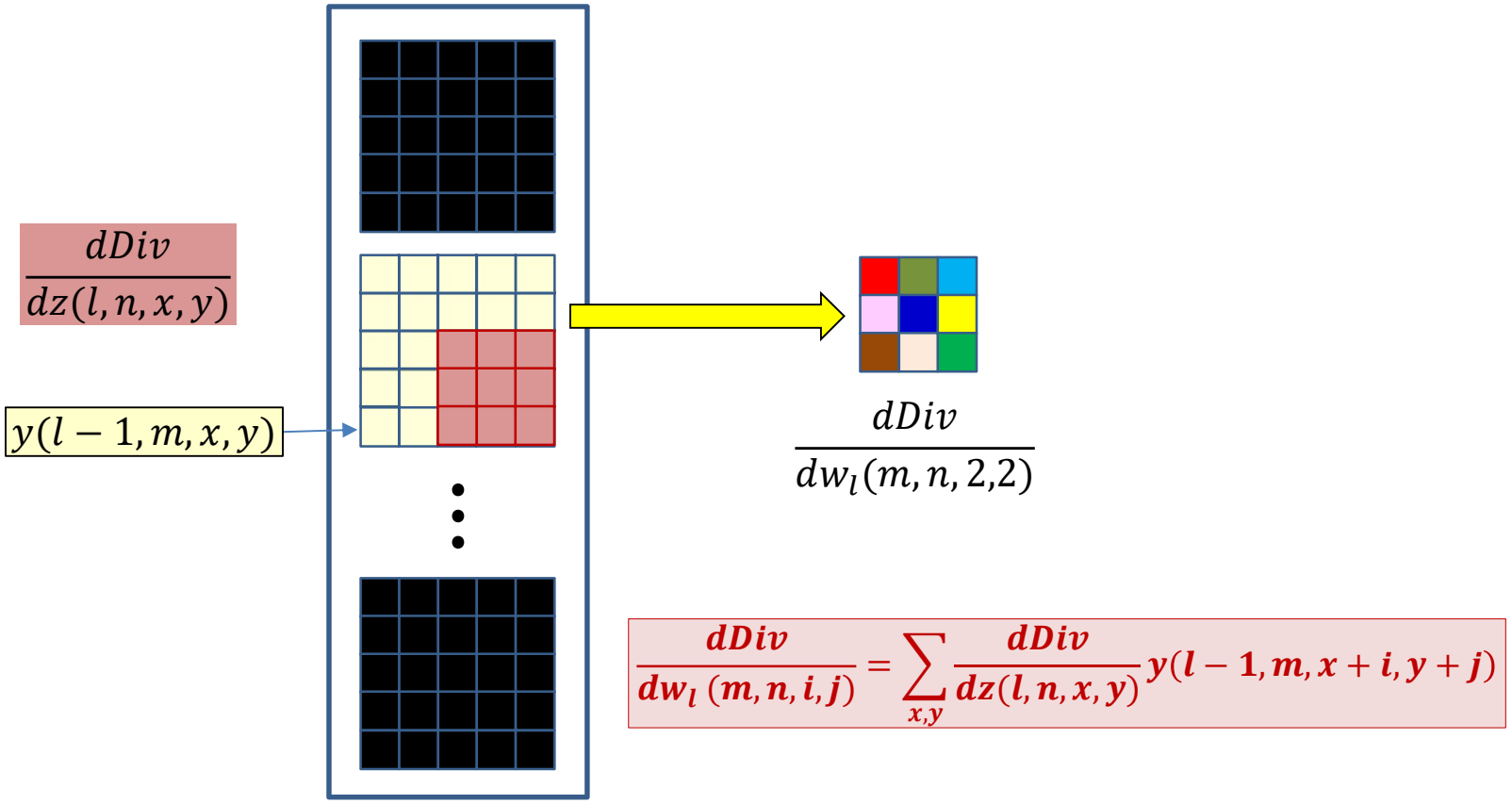
- The derivatives of the divergence w.r.t. every element of $Z(l,n)$ is known
  - Must use them to compute the derivative for $w_l(m,n,*,*)$

The filter derivative



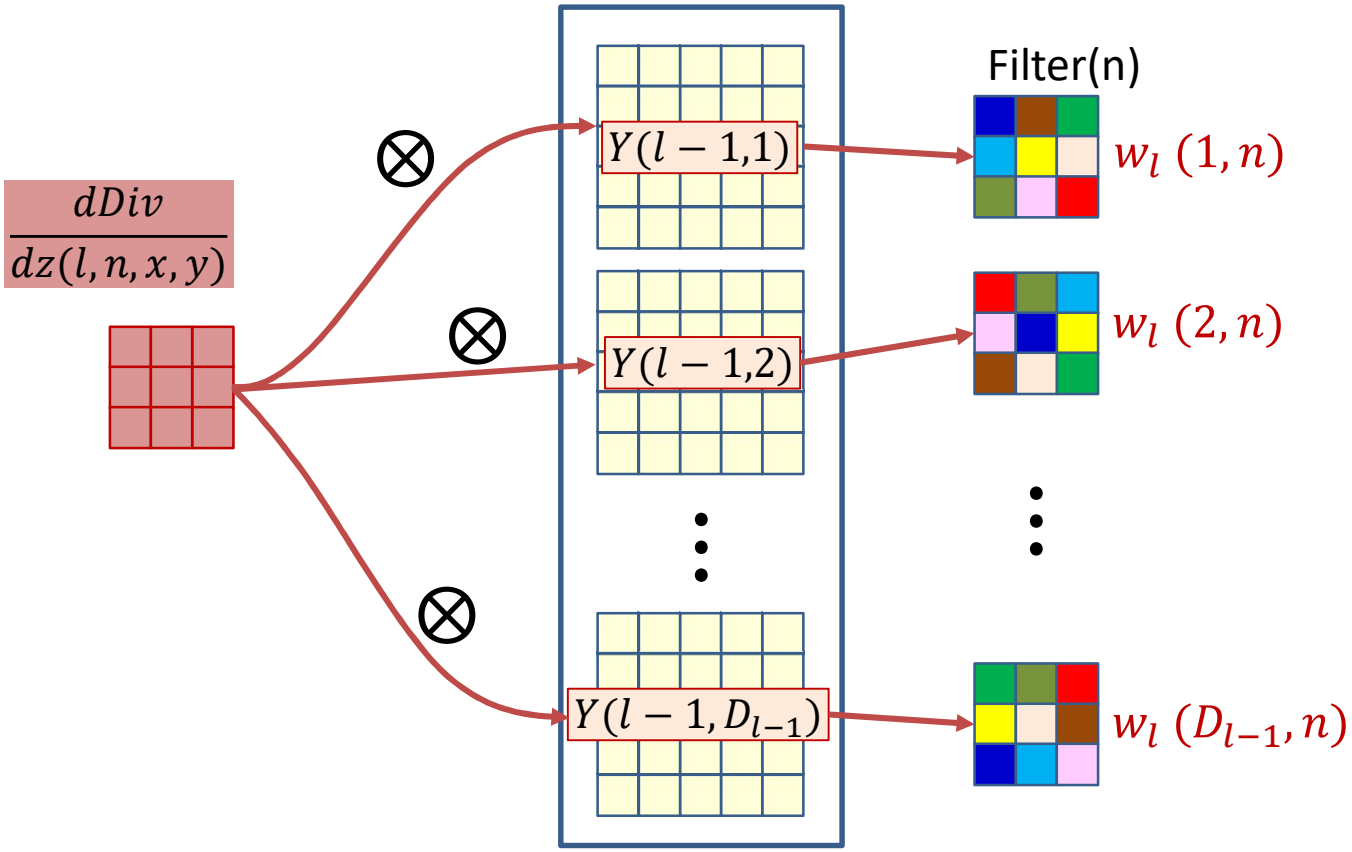
- The derivatives of the divergence w.r.t. every element of $Z(l,n)$ is known
  - Must use them to compute the derivative for $w_l(m,n,*,*)$

The filter derivative



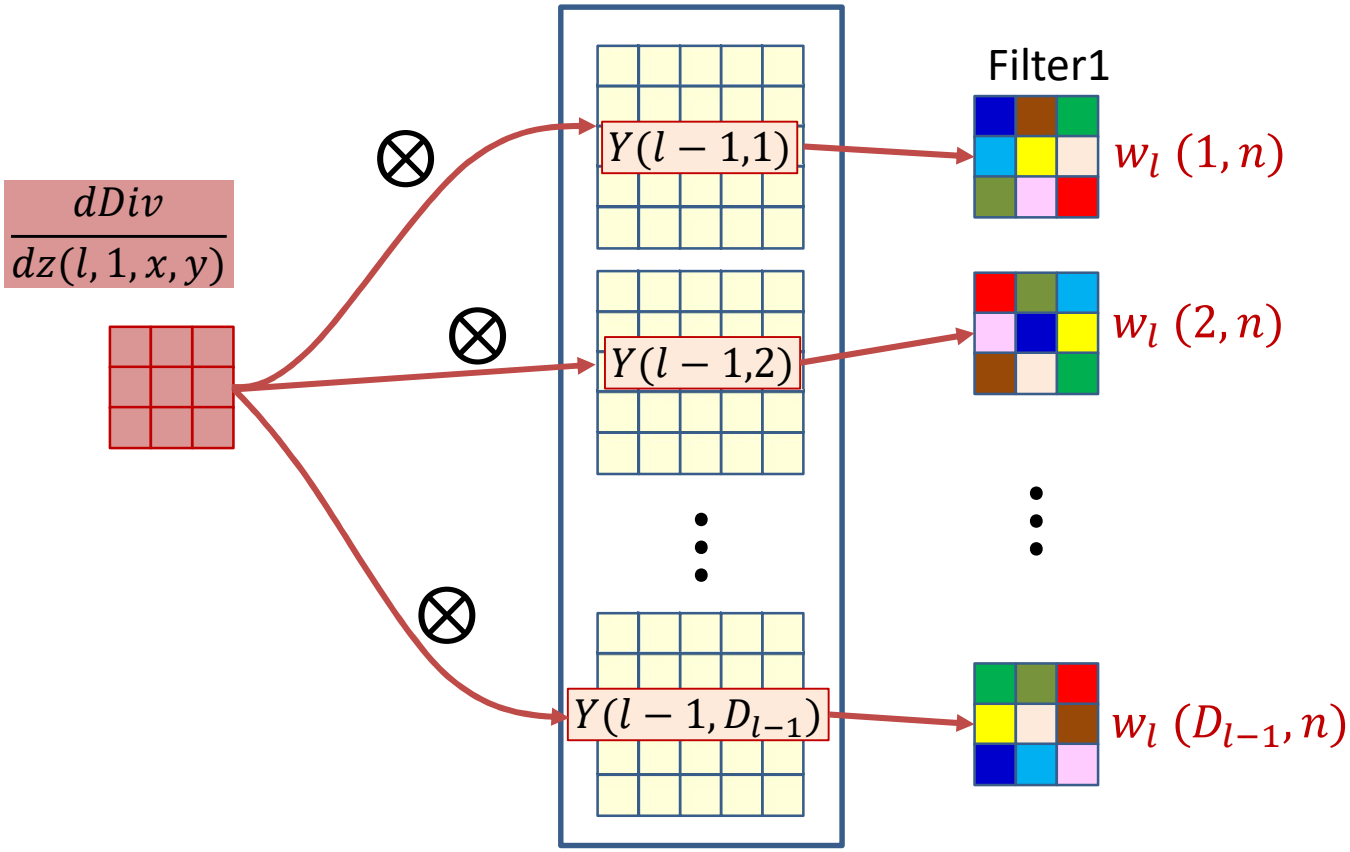
- The derivatives of the divergence w.r.t. every element of $Z(l,n)$ is known
  - Must use them to compute the derivative for $w_l(m,n,*,*)$

The filter derivative



- The derivative of the n^{th} affine map $Z(l,n)$ convolves with every output map $Y(l-1,m)$ of the $(l-1)^{\text{th}}$ layer, to get the derivative for $w_l(m,n)$, the m^{th} "plane" of the n^{th} filter

The filter derivative



$$\frac{dDiv}{dw_l\ (m,n,i,j)} = \sum_{x,y} \frac{dDiv}{dz(l,n,x,y)} y(l-1,m,x+i,y+j)$$

$$= \frac{dDiv}{dz(l,n)} \otimes y(l-1,m)$$

$\frac{dDiv}{dw_l\ (m,n,i,j)}$ must be upsampled if the stride was greater than 1 in the forward pass

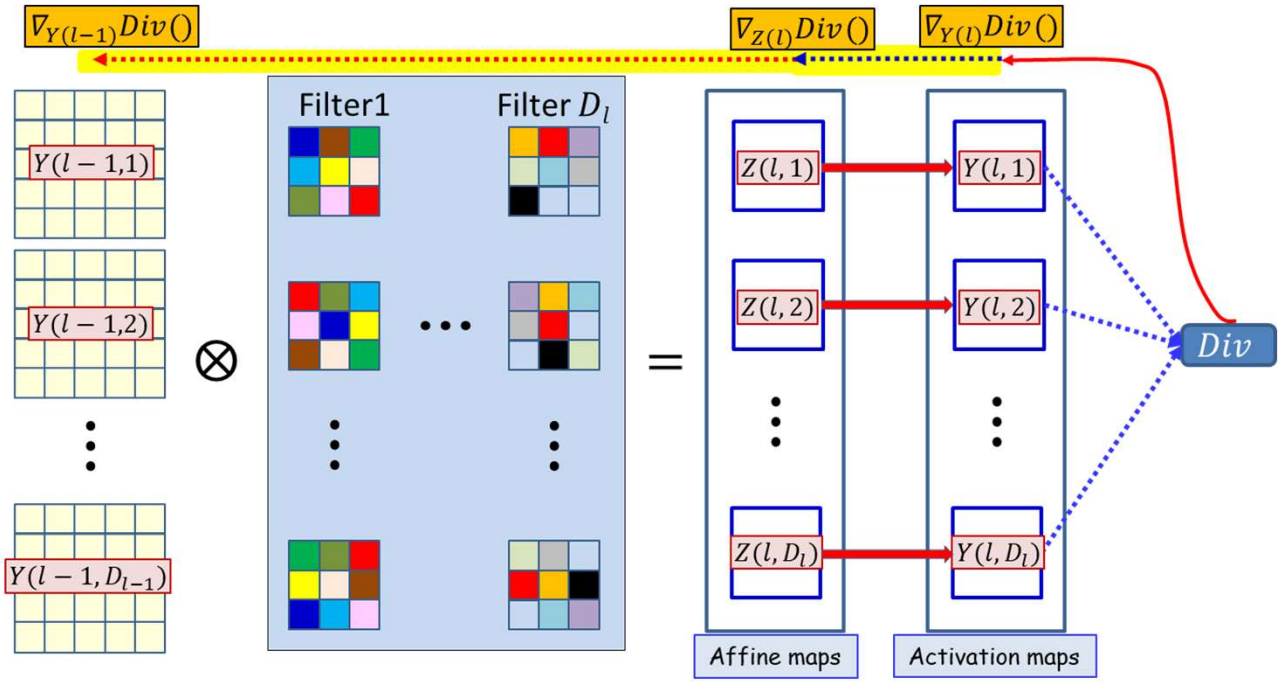
If $Y(l-1,m)$ was zero padded in the forward pass, it must be zero padded for backprop

Derivatives for the filters at layer l: Vector notation

```
# The weight W(l,j)is a 3D D_{l-1}xK_lxK_l
# Assuming that derivative maps have been upsampled
#    if stride > 1
# Also assuming y map has been zero-padded if this was
#    also done in the forward pass
```

```
for n = 1:D_l
  for x = 1:K_l
    for y = 1:K_l
      for m = 1:D_{l-1}
        dw(l,m,n,x,y) = dz(l,n,:,:).        #dot product
                        y(l-1,m,x:x+K_l-1,y:y+K_l-1)
```

Backpropagation: Convolutional layers



- **For convolutional layers**:

  - How to compute the derivatives w.r.t. the affine combination $Z(l)$ maps from the activation output maps $Y(l)$
  - How to compute the derivative w.r.t. $Y(l-1)$ and $w(l)$ given derivatives w.r.t. $Z(l)$

CNN: Forward

```
Y(0,:,:,:) = Image
for l = 1:L  # layers operate on vector at (x,y)
   for x = 1:W-K+1
      for y = 1:H-K+1
         for j = 1:D_l
            z(l,j,x,y) = 0
            for i = 1:D_{l-1}
               for x' = 1:K_l
                  for y' = 1:K_l
                     z(l,j,x,y) += w(l,j,i,x',y')
                                   Y(l-1,i,x+x'-1,y+y'-1)
            Y(l,j,x,y) = activation(z(l,j,x,y))

Y = softmax( Y(L,:,1,1)..Y(L,:,W-K+1,H-K+1) )
```

Switching to 1-based indexing with appropriate adjustments

Backward layer l

```
dw(l) = zeros(D_l x D_{l-1} x K_l x K_l)
dY(l-1) = zeros(D_{l-1} x W_{l-1} x H_{l-1})
for x = 1:W_{l-1}-K_l+1
  for y = 1:H_{l-1}-K_l+1
    for j = 1:D_l
      dz(l,j,x,y) = dY(l,j,x,y).f'(z(l,j,x,y))
      for i = 1:D_{l-1}
        for x' = 1:K_l
          for y' = 1:K_l
            dY(l-1,i,x+x'-1,y+y'-1) +=
                        w(l,j,i,x',y')dz(l,j,x,y)
            dw(l,j,i,x',y') +=
                  dz(l,j,x,y)Y(l-1,i,x+x'-1,y+y'-1)
```

Complete Backward (no pooling)

```
dY(L) = dDiv/dY(L)
for l = L:downto:1  # Backward through layers
    dw(l) = zeros(D_l x D_{l-1} x K_l x K_l)
    dY(l-1) = zeros(D_{l-1} x W_{l-1} x H_{l-1})
    for x = 1:W_{l-1}-K_l+1
        for y = 1:H_{l-1}-K_l+1
            for j = 1:D_l
                dz(l,j,x,y) = dY(l,j,x,y).f'(z(l,j,x,y))
                for i = 1:D_{l-1}
                    for x' = 1:K_l
                        for y' = 1:K_l
                            dY(l-1,i,x+x'-1,y+y'-1) +=
                                w(l,j,i,x',y')dz(l,j,x,y)
                            dw(l,j,i,x',y') +=
                            dz(l,j,x,y)y(l-1,i,x+x'-1,y+y'-1)
```

Complete Backward (no pooling)

```
dY(L) = dDiv/dY(L)
for l = L:downto:1   # Backward through layers
    dw(l) = zeros(D_l x D_{l-1} x K_l x K_l)
    dY(l-1) = zeros(D_{l-1} x W_{l-1} x H_{l-1})
    for x = 1:W_{l-1}-K_l+1
        for y = 1:H_{l-1}-K_l+1
            for j = 1:D_l
                dz(l,j,x,y) = dY(l,j,x,y).f'(z(l,j,x,y))
                for i = 1:D_{l-1}
                    for x' = 1:K_l
                        for y' = 1:K_l
                            dY(l-1,i,x+x'-1,y+y'-1) +=
                                w(l,j,i,x',y')dz(l,j,x,y)
                            dw(l,j,i,x',y') +=
                            dz(l,j,x,y)y(l-1,i,x+x'-1,y+y'-1)
```

Multiple ways of recasting this as tensor/ vector operations.

Will not discuss here

Complete Backward (with strides)

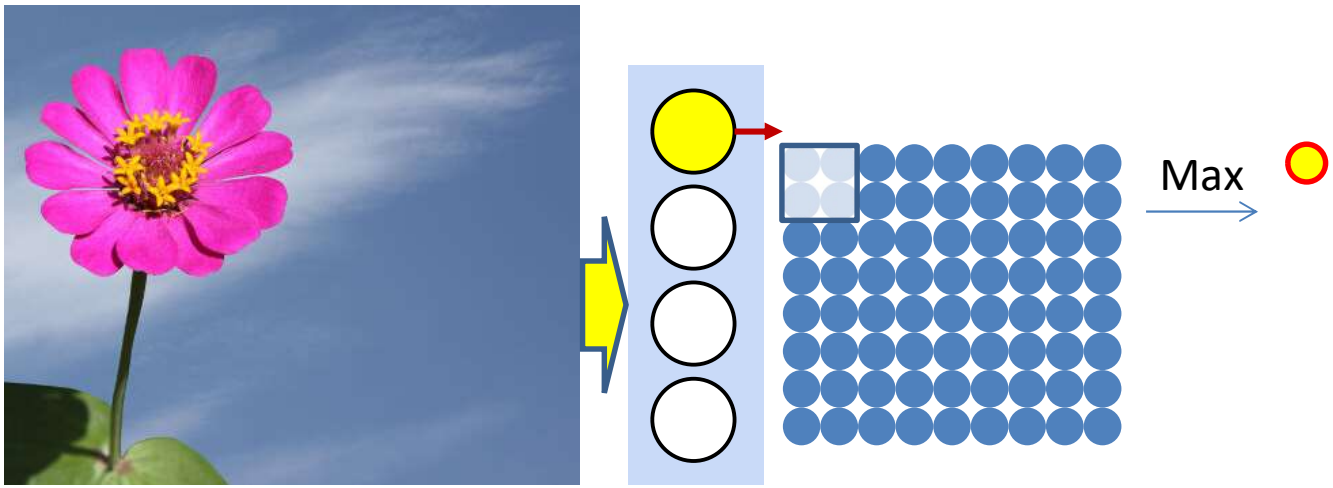
```
dY(L) = dDiv/dY(L)
for l = L:1  # Backward through layers
   dw(l) = zeros(D_l x D_{l-1} x K_l x K_l)
   dY(l-1) = zeros(D_{l-1} x W_{l-1} x H_{l-1})
   for x = 1:stride:W_l
      m = (x-1)stride
      for y = 1:stride: H_l
         n = (y-1)stride
         for j = 1:D_l
            dz(l,j,x,y) = dY(l,j,x,y).f'(z(l,j,x,y))
            for i = 1:D_{l-1}
               for x' = 1:K_l
                  for y' = 1:K_l
                     dY(l-1,i,m+x',n+y') +=
                        w(l,j,i,x',y')dz(l,j,x,y)
                     dw(l,j,i,x',y') +=
                        dz(l,j,x,y)y(l-1,i,m+x',n+y')
```

Backpropagation: Convolutional and Pooling layers

- **Assumption:** We already have the derivatives w.r.t. the elements of the maps output by the final convolutional (or pooling) layer
  - Obtained as a result of backpropagating through the flat MLP

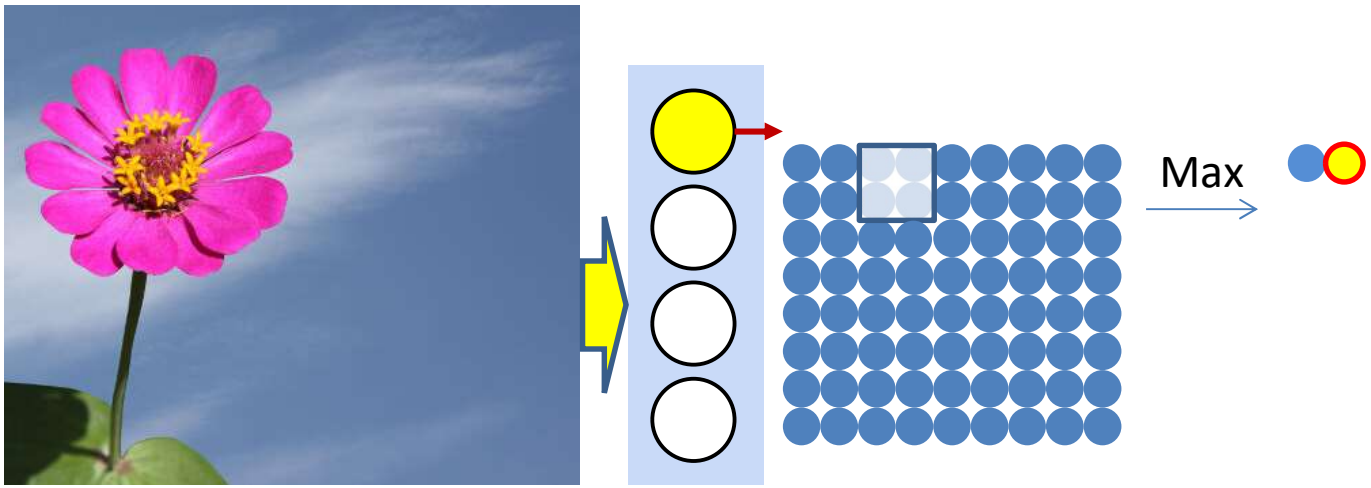
- **Required:**
  - **For convolutional layers**:
    - How to compute the derivatives w.r.t. the affine combination $Z(l)$ maps from the activation output maps $Y(l)$
    - How to compute the derivative w.r.t. $Y(l-1)$ and $w(l)$ given derivatives w.r.t. $Z(l)$
  - **For pooling layers**:
    - How to compute the derivative w.r.t. $Y(l-1)$ given derivatives w.r.t. $Y(l)$

Pooling and downsampling



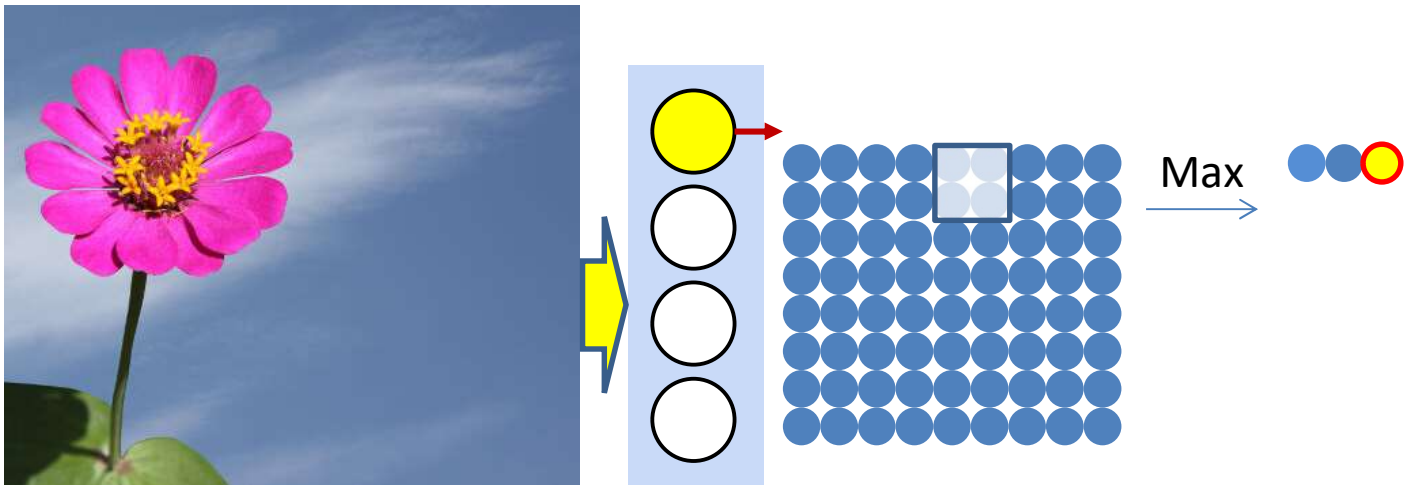
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



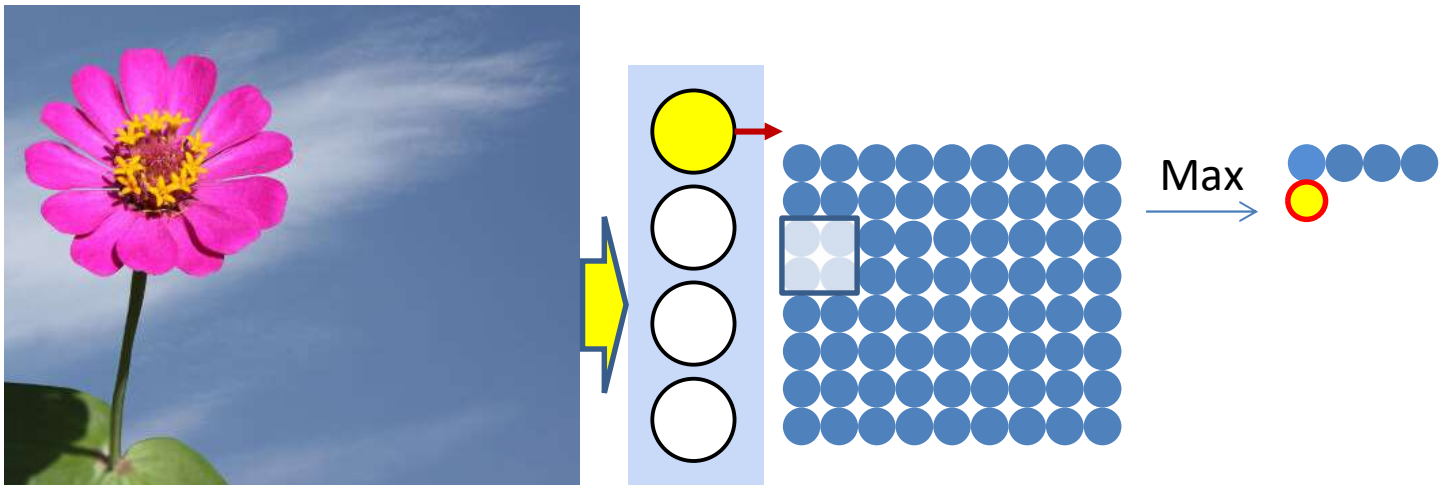
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



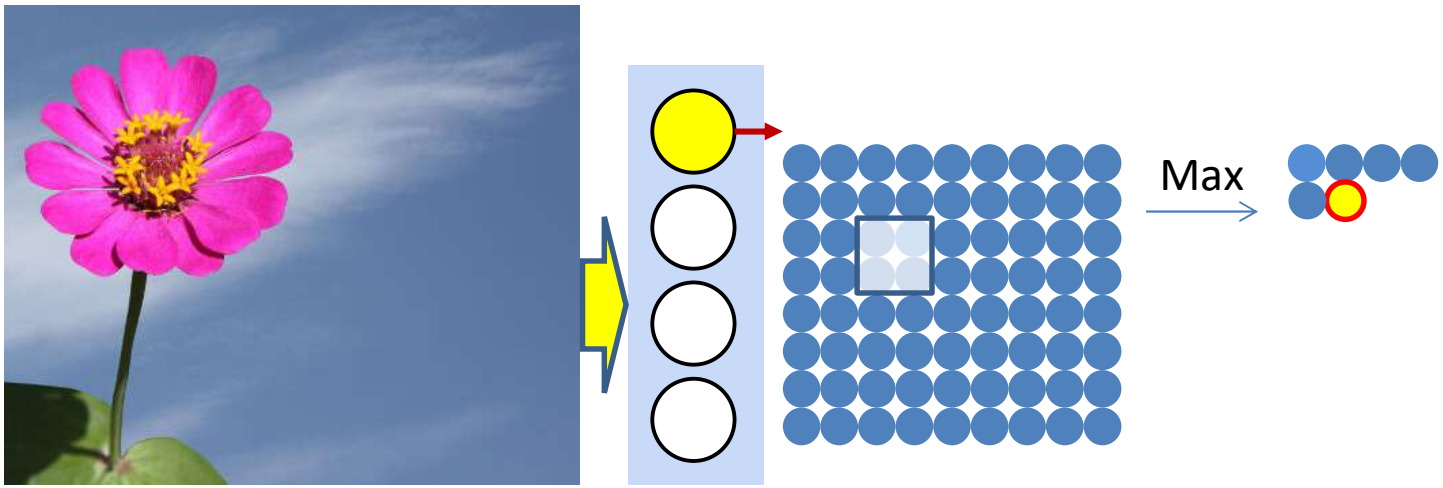
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



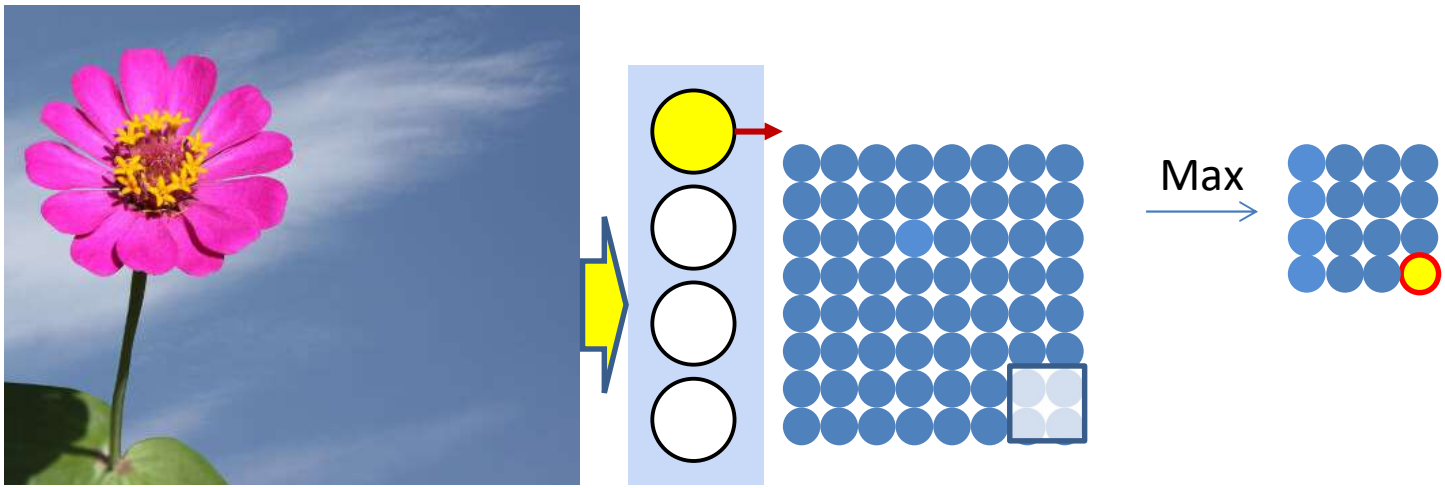
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



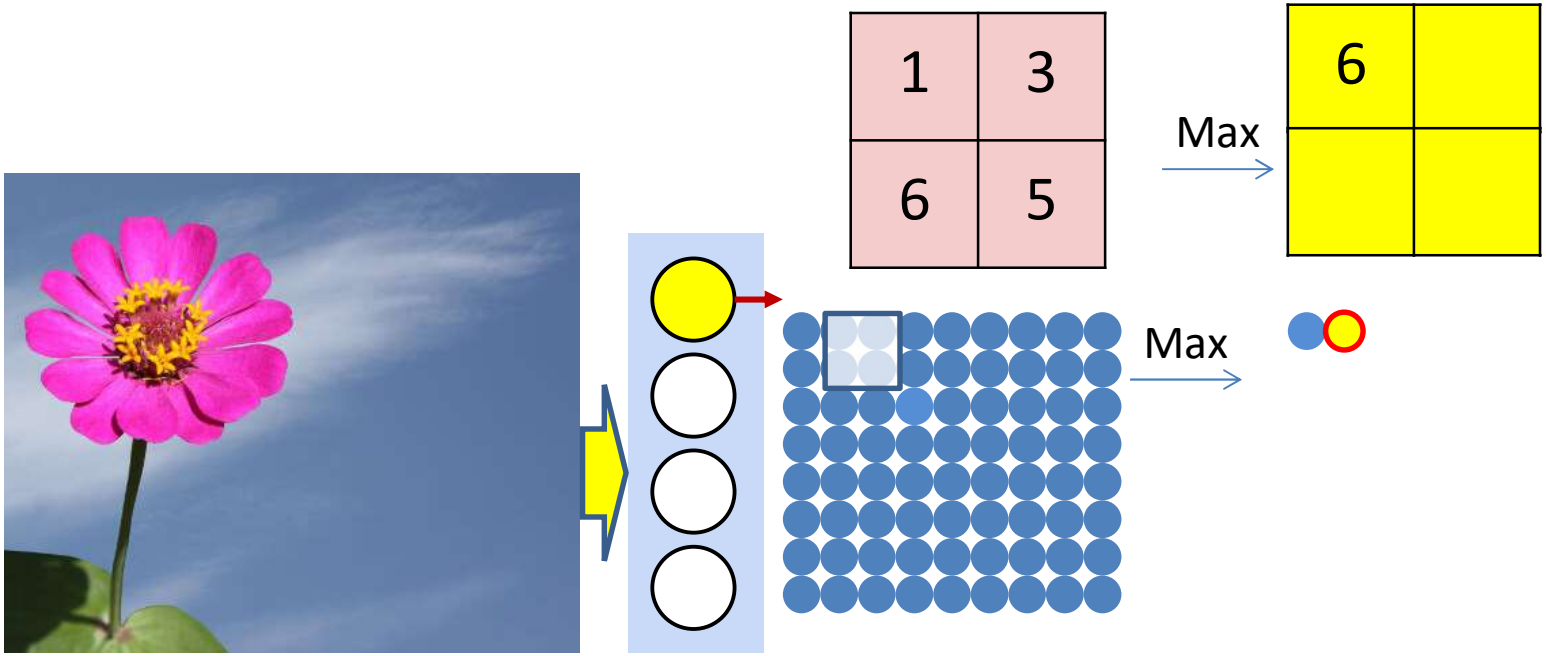
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Max pooling

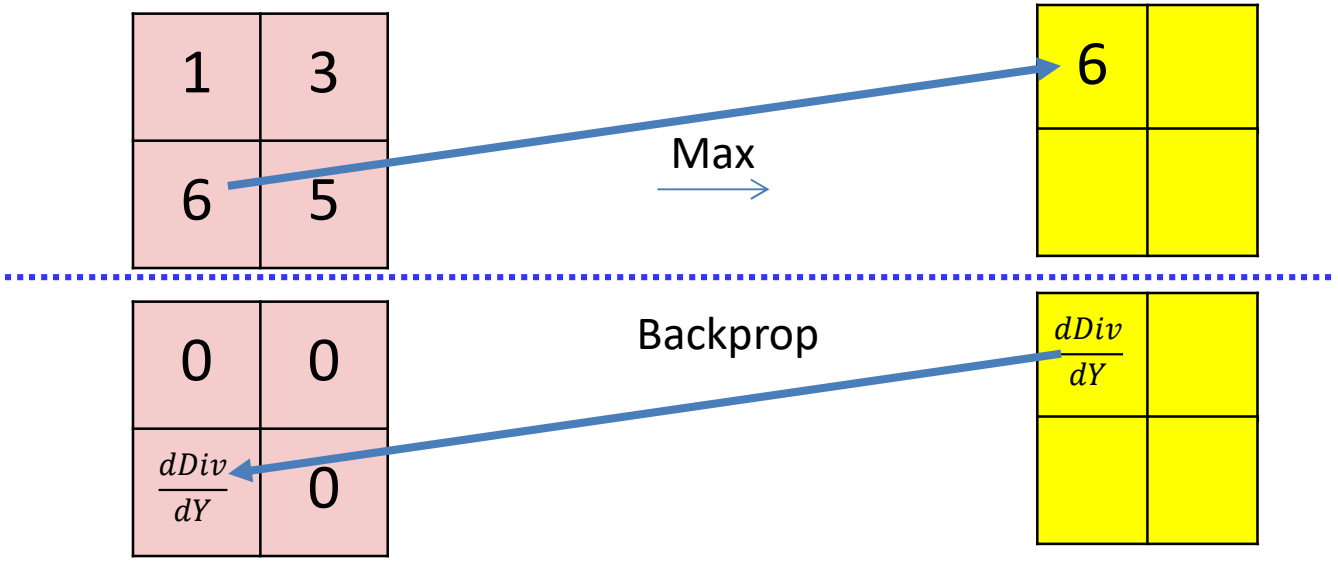


- Max pooling selects the largest from a pool of elements
- Pooling is performed by "scanning" the input

$$P(l,m,i,j) = \underset{\substack{k\in\{(i-1)d+1,\,(i-1)d+K_{lpool}\},\\ n\in\{(j-1)d+1,(j-1)d+K_{lpool}\}}}{\operatorname{argmax}} Y(l-1,m,k,n)$$

$$Y(l,m,i,j) = Y(l-1,m,P(l,m,i,j))$$

Derivative of Max pooling



$$\frac{dDiv}{dy(l-1,m,k,l)} = \begin{cases} \frac{dDiv}{dy(l,m,i,j)} & if\ (k,l) = P(l,m,i,j) \\ 0\ otherwise \end{cases}$$

- Max pooling selects the largest from a pool of elements

$$P(l,m,i,j) = \underset{\substack{k\in\{(i-1)d+1,\,(i-1)d+K_{lpool}\},\\ n\in\{(j-1)d+1,(j-1)d+K_{lpool}\}}}{\operatorname{argmax}} y(l-1,m,k,n)$$

$$y(l,m,i,j) = y(l-1,m,P(l,m,i,j))$$

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:D_l
  m = 1
  for x = 1:stride(l):W_l-1-K_l+1
    n = 1
    for y = 1:stride(l):H_l-1-K_l+1
      pidx(l,j,m,n) = maxidx(y(l-1,j,x:x+K_l-1,y:y+K_l-1))
      y(l,j,m,n) = y(l-1,j,pidx(l,j,m,n))
      n = n+1
    m = m+1
```

Derivative of max pooling layer at layer l

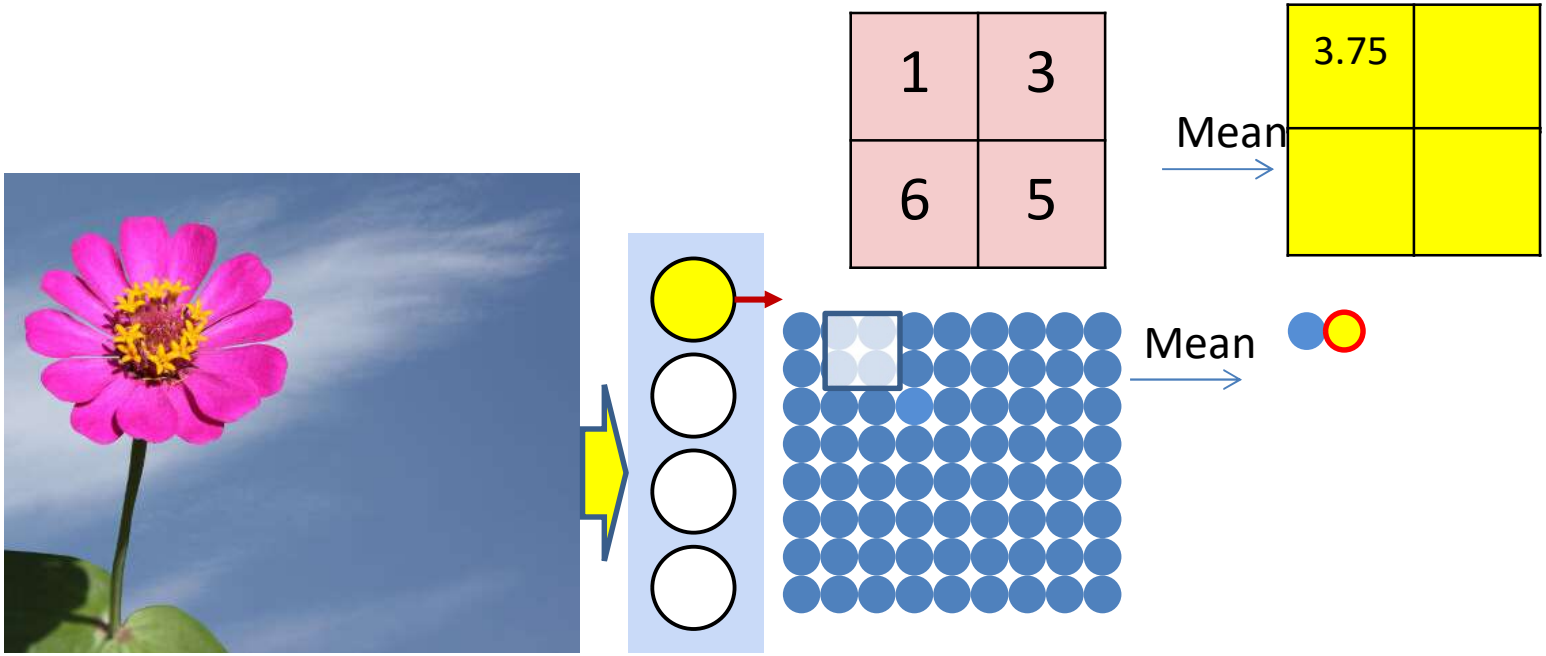
a) Performed separately for every map (j).
*) Not combining multiple maps within a single max operation.
b) Keeping track of location of max

Max pooling

```
dy(:,:,:) = zeros(D_l x W_l x H_l)
for j = 1:D_l
    for x = 1:W_l_downsampled
        for y = 1:H_l_downsampled
            dy(l-1,j,pidx(l,j,x,y)) += dy(l,j,x,y)
```

"+=" because this entry may be selected in multiple adjacent overlapping windows

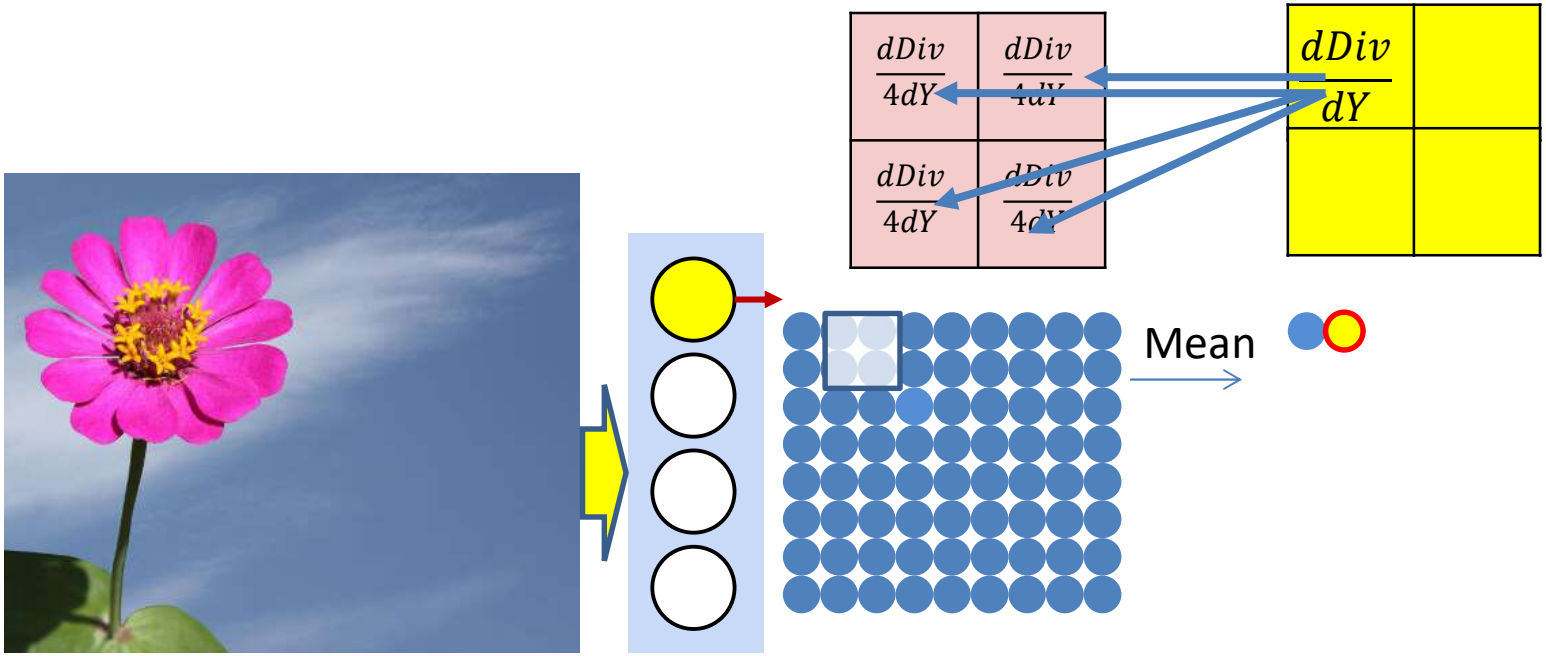
Mean pooling



- Mean pooling compute the mean of a pool of elements
- Pooling is performed by “scanning” the input

$$y(l,m,i,j) = \frac{1}{K_{lpool}^2} \sum_{\substack{k\in\{(i-1)d+1,\,(i-1)d+K_{lpool}\},\\ n\in\{(j-1)d+1,(j-1)d+K_{lpool}\}}} y(l-1,m,k,n)$$

Derivative of mean pooling



- The derivative of mean pooling is distributed over the pool

$$k \in \{(i-1)d+1, (i-1)d+K_{lpool}\},$$
$$n \in \{(j-1)d+1, (j-1)d+K_{lpool}\}$$
$$dy(l-1,m,k,n) = \frac{1}{K_{lpool}^2} dy(l,m,k,n)$$

Mean Pooling layer at layer l

a) Performed separately for every map (j).
*) Not combining multiple maps within a single mean operation.

Mean pooling

```
for j = 1:D_l  #Over the maps
    m = 1
    for x = 1:stride(l):W_l-1-K_l+1  #K_l = pooling kernel size
        n = 1
        for y = 1:stride(l):H_l-1-K_l+1
            y(l,j,m,n) = mean(y(l-1,j,x:x+K_l-1,y:y+K_l-1))
            n = n+1
        m = m+1
```

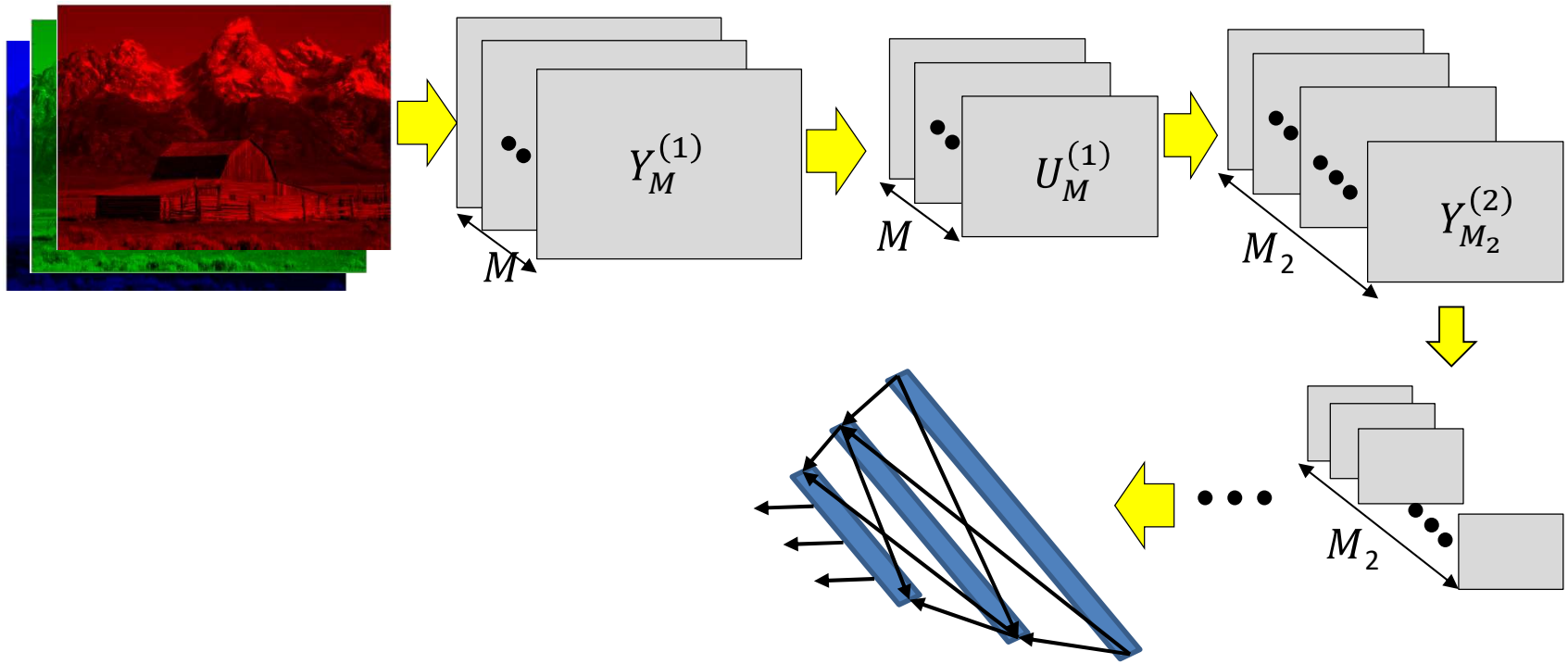
Derivative of mean pooling layer at layer l

Mean pooling

```
dy(:,:,:) = zeros(D_l x W_l x H_l)
for j = 1:D_l
  for x = 1:W_l_downsampled
    n = (x-1)*stride
    for y = 1:H_l_downsampled
      m = (y-1)*stride
      for i = 1:K_lpool
        for j = 1:K_lpool
          dy(l-1,j,p,n+i,m+j) += (1/K^2_lpool)y(l,j,x,y)
```

"+=" because adjacent windows may overlap

Learning the network



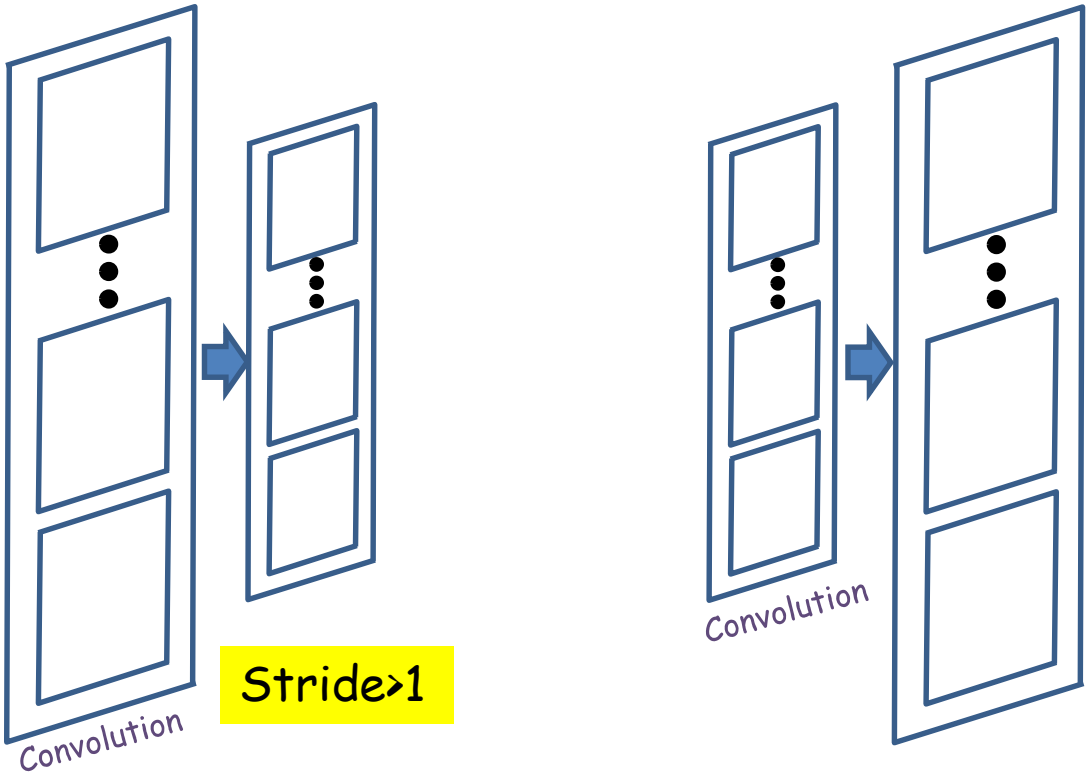
- Have shown the derivative of divergence w.r.t every intermediate output, and every free parameter (filter weights)
- Can now be embedded in gradient descent framework to learn the network

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

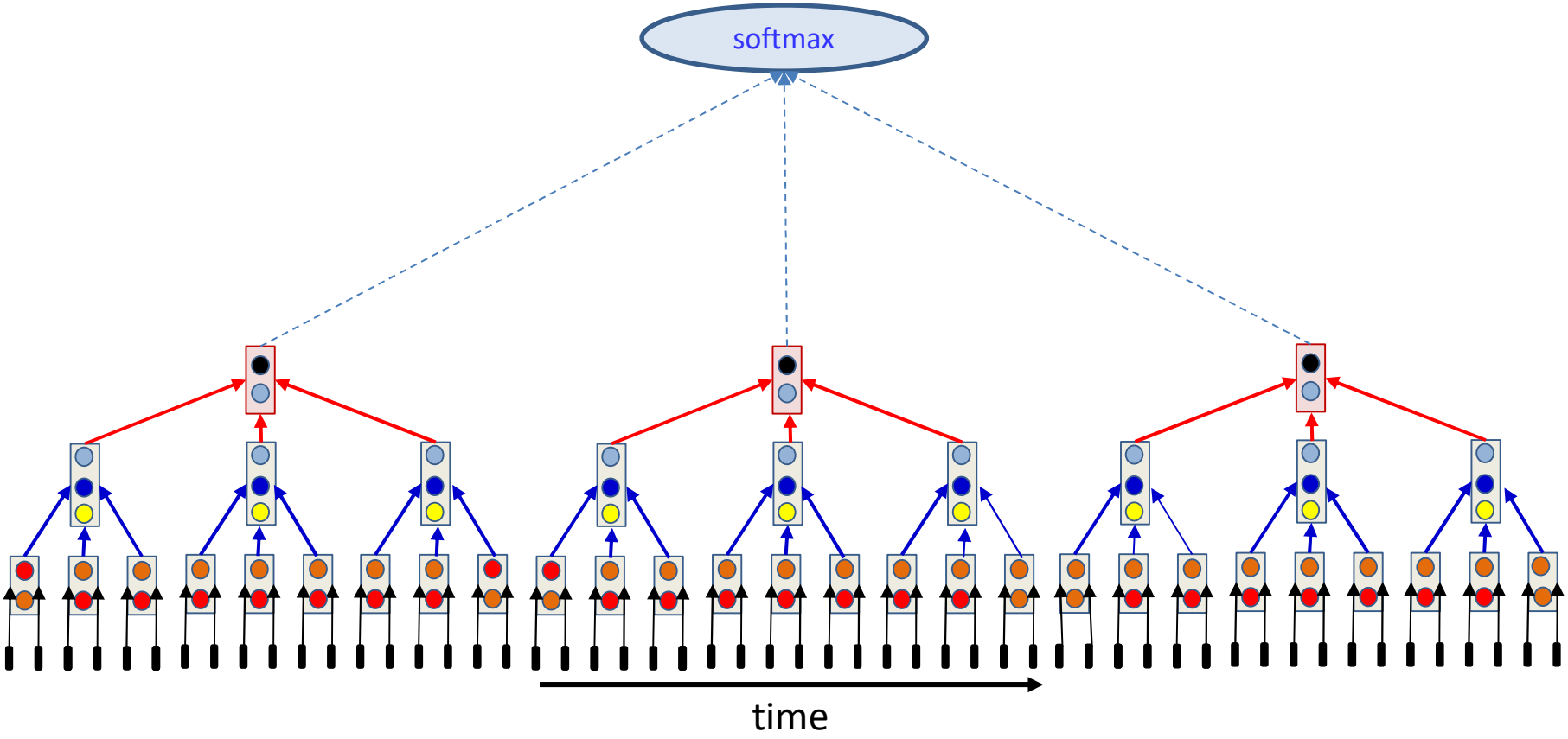
- The parameters of the network can be learned through regular back propagation
  - Maxpooling layers must propagate derivatives only over the maximum element in each pool
    - Other pooling operators can use regular gradients or subgradients
  - Derivatives must sum over appropriate sets of elements to account for the fact that the network is, in fact, a shared parameter network

An implicit assumption



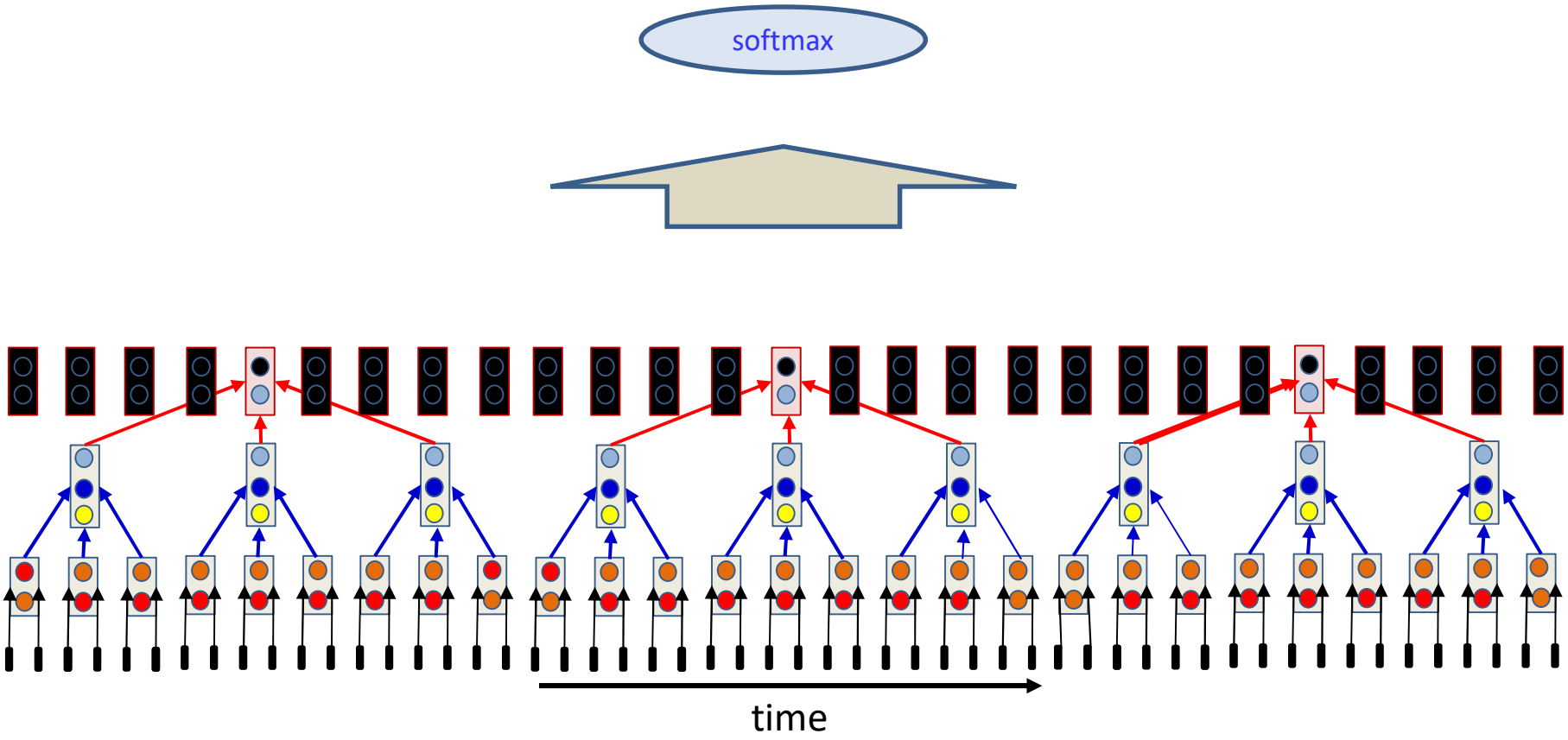
- We've always assumed that subsequent steps *shrink* the size of the maps
- Can subsequent maps *increase* in size?

1-D scans



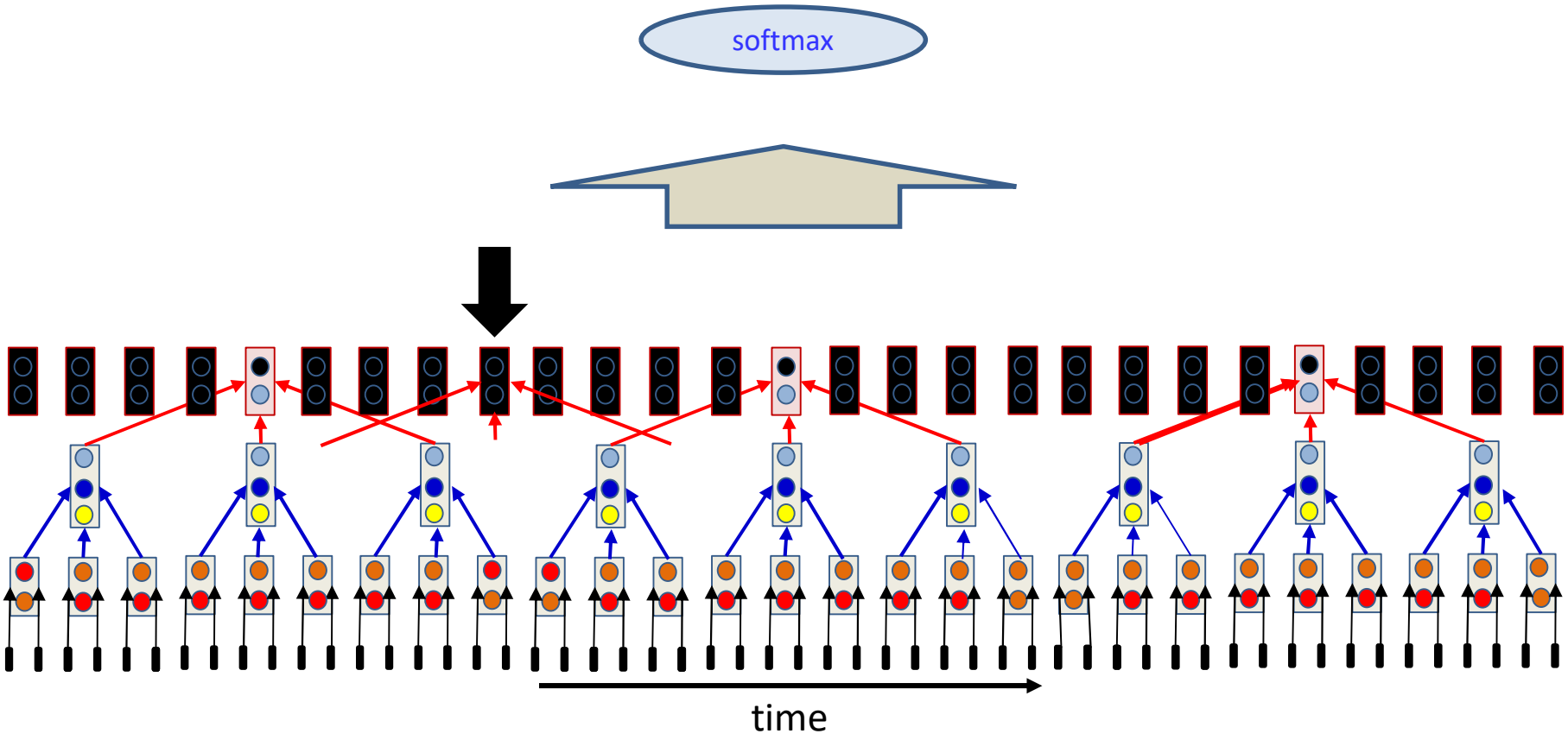
- The number of "bars" in each layer is usually the same or *smaller* than the bars in the previous layer
  - Scanning maintains or reduces the time resolution of the signal at each layer

Upsampling 1-D scans



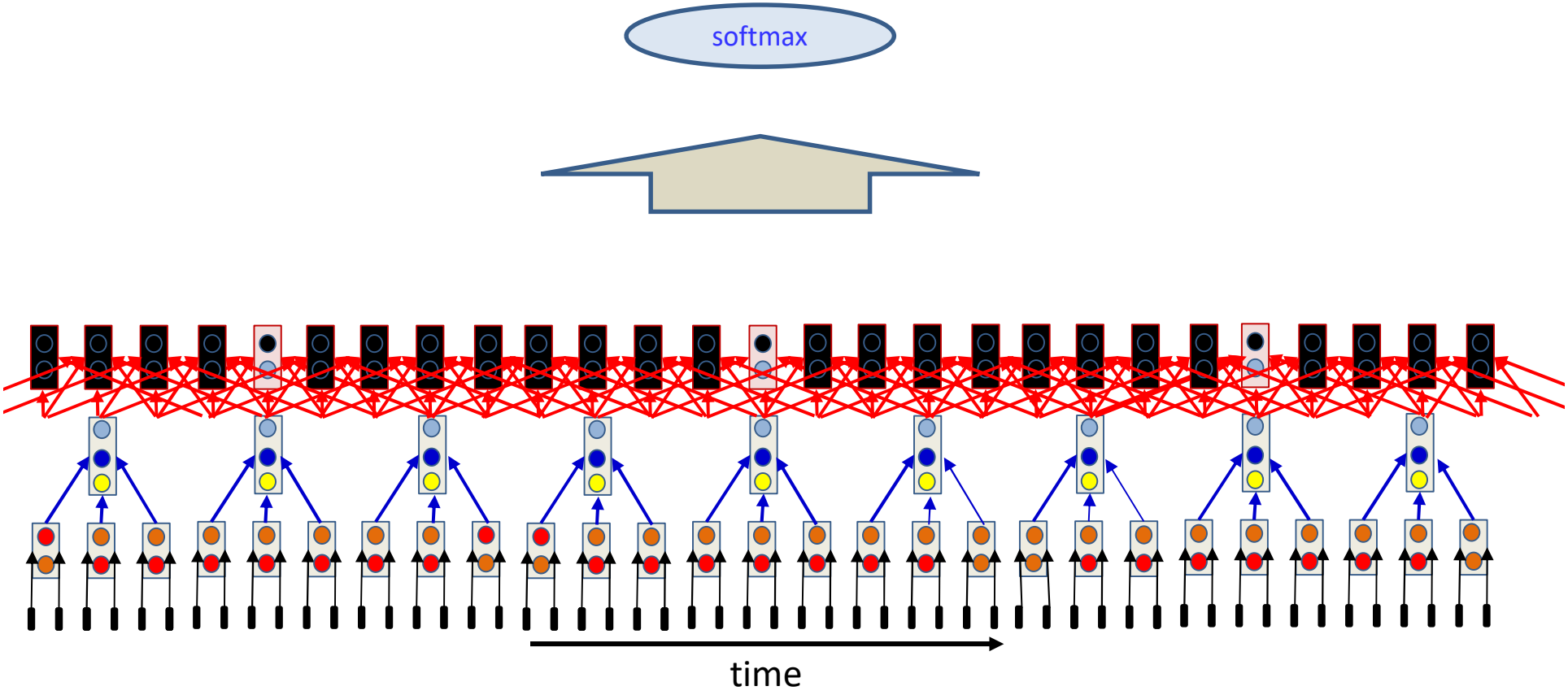
- The number of "bars" in each layer is usually the same or *smaller* than the bars in the previous layer
  - Scanning maintains or reduces the time resolution of the signal at each layer
- What if we want to *increase the* time resolution with layers?

Upsampling 1-D scans



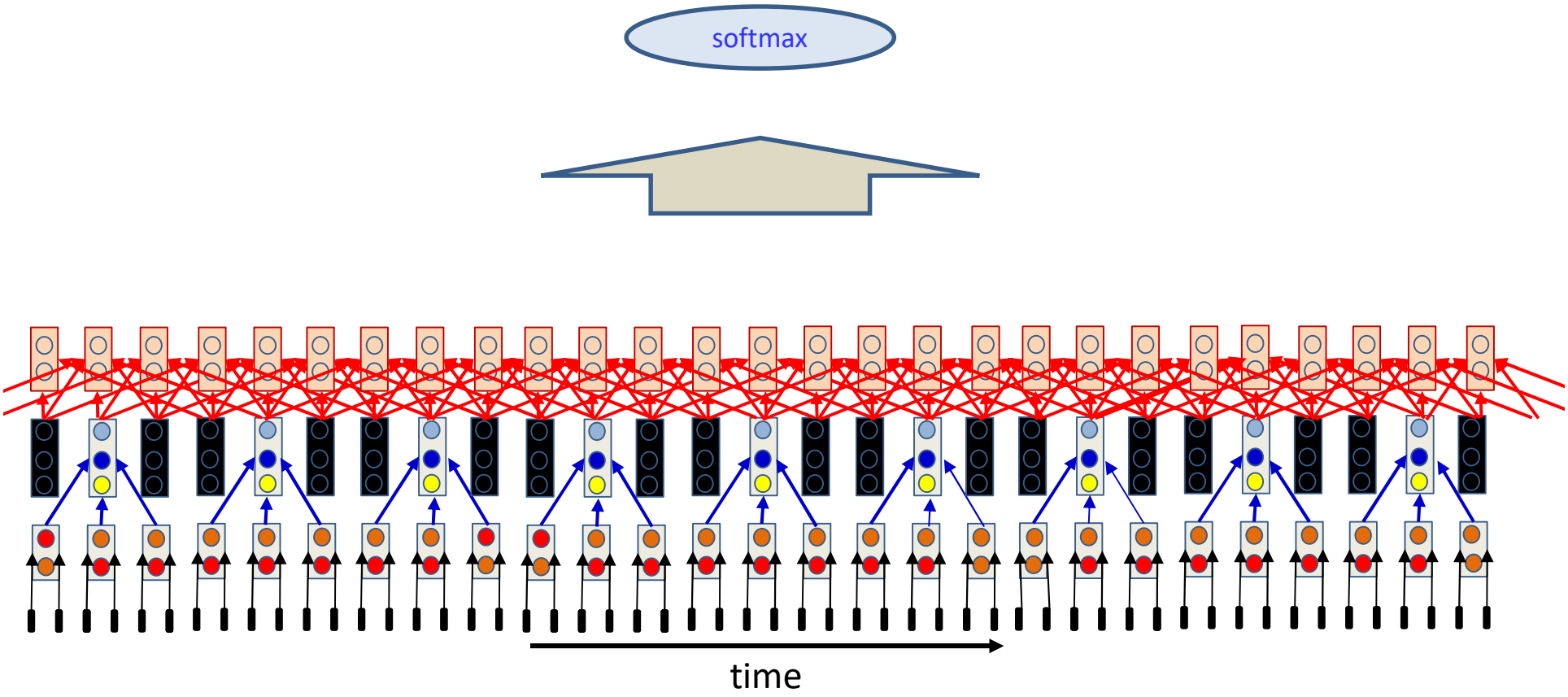
- The number of "bars" in each layer is usually the same or *smaller* than the bars in the previous layer
  - Scanning maintains or reduces the time resolution of the signal at each layer
- What if we want to *increase the* time resolution with layers?

Upsampling 1-D scans



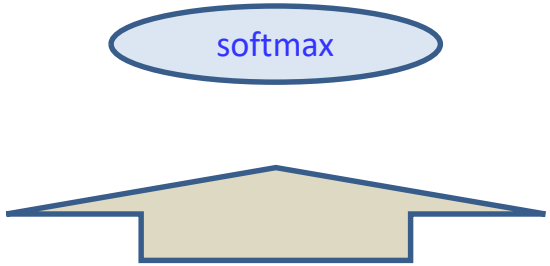
- **Problem:** The values required to compute the intermediate values are missing from the previous layer!

Upsampling 1-D scans

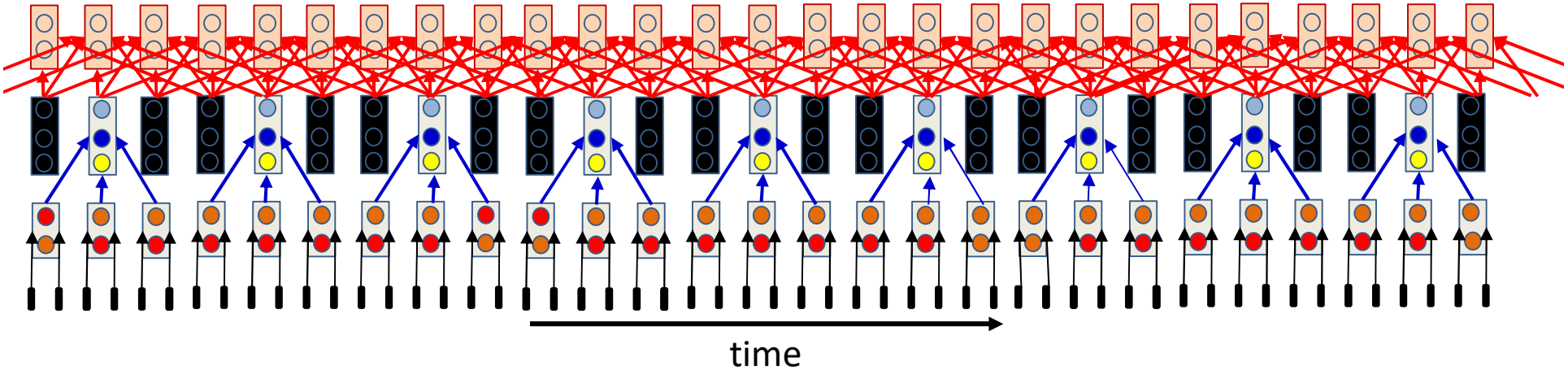


- **Problem:** The values required to compute the intermediate values are missing from the previous layer!
- **Solution:** Synthetically fill in the missing intermediate values of the previous layer
  - With zeros
    - Could also fill them in with linear or spline interpolation of neighbors, but it will complicate backprop

Upsampling 1-D scans

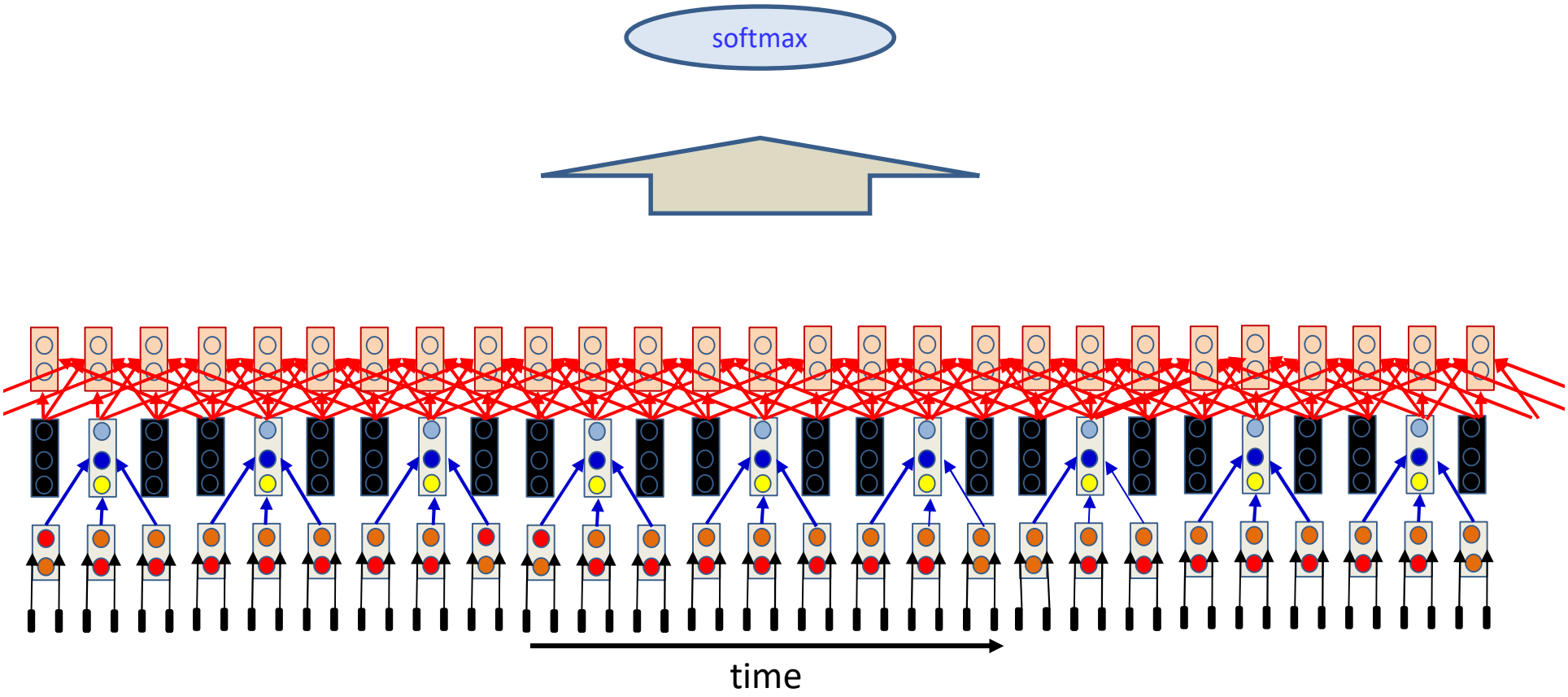


This is exactly analogous to the upsampling performed during backprop when forward convolution uses stride > 1



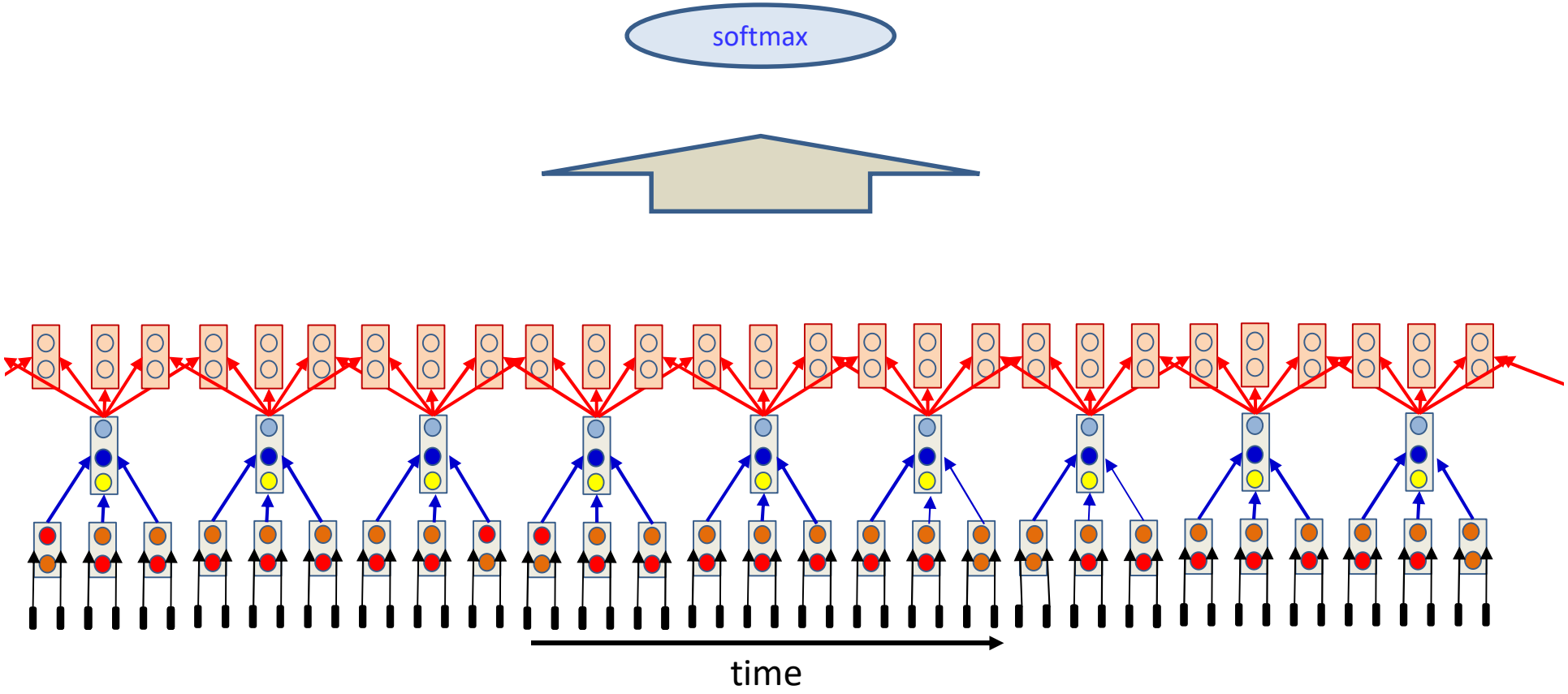
- **Problem:** The values required to compute the intermediate values are missing from the previous layer!
- **Solution:** Synthetically fill in the missing intermediate values of the previous layer
  - With zeros
    - Could also fill them in with linear or spline interpolation of neighbors, but it will complicate backprop

Upsampling 1-D scans



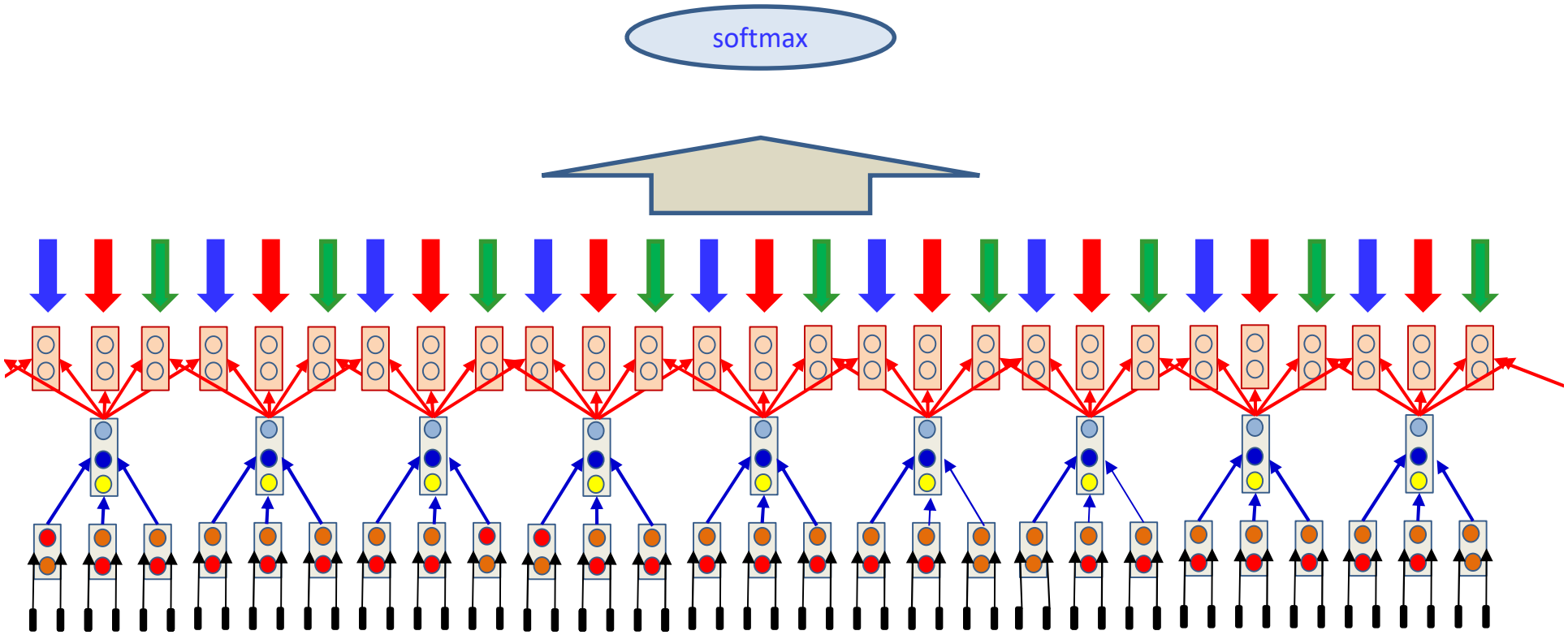
- The 0-valued interpolated inputs do not really provide any input
- They, and their connections can be removed without changing the computation

Upsampling 1-D scans



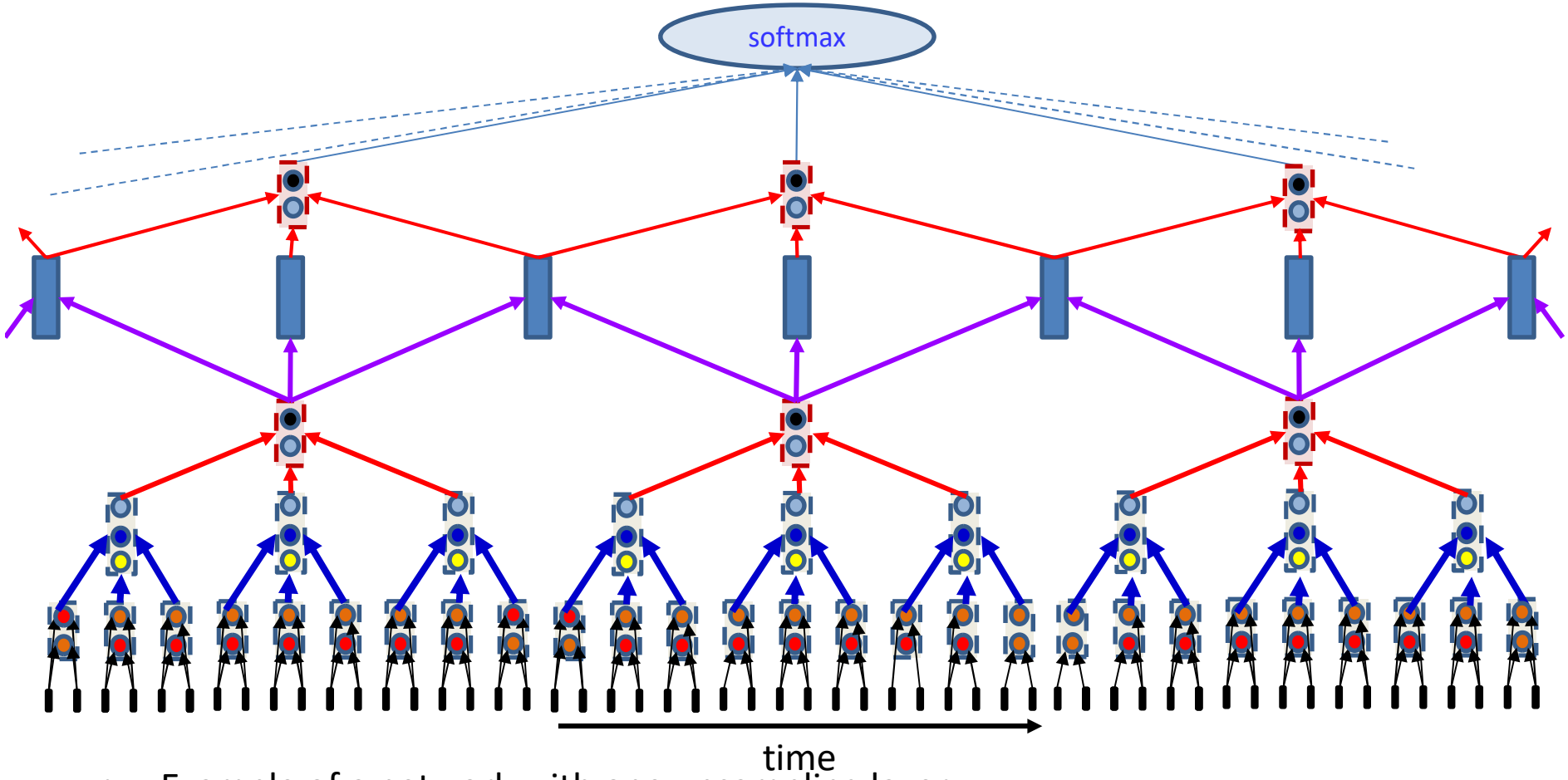
- The 0-valued interpolated inputs do not really provide any input
- They, and their connections can be removed without changing the computation
- *This* is the actual computation performed

Upsampling 1-D scans



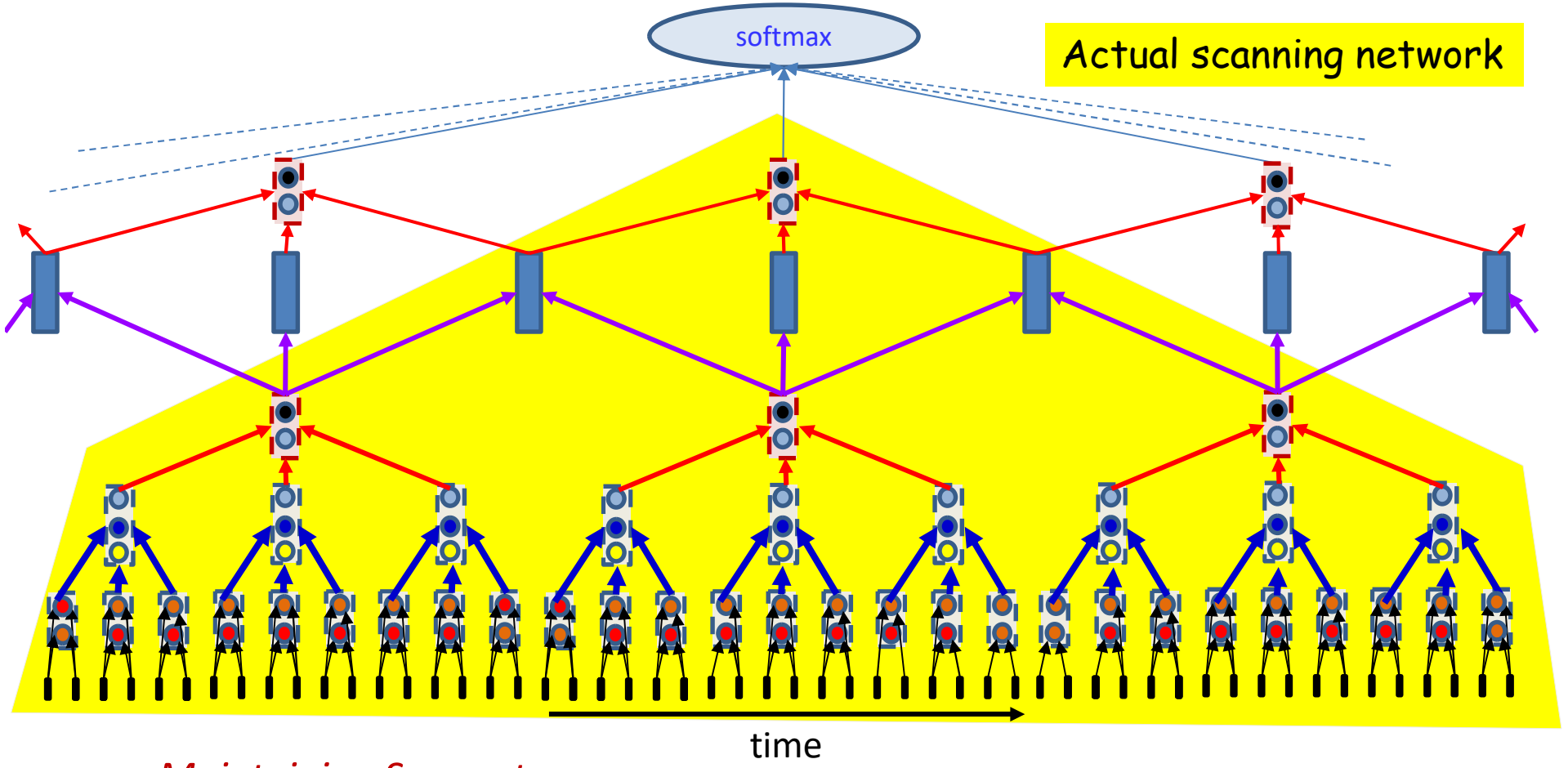
- Key difference from downsampling layers
  - All the “columns” in the regular/downsampling layers are identical
    - Their *incoming* weight patterns are identical
  - The columns in the upsampling layers are *not* identical
    - The *outgoing* weight patterns of the *lower* layer columns are identical

Upsampling as a scanning network



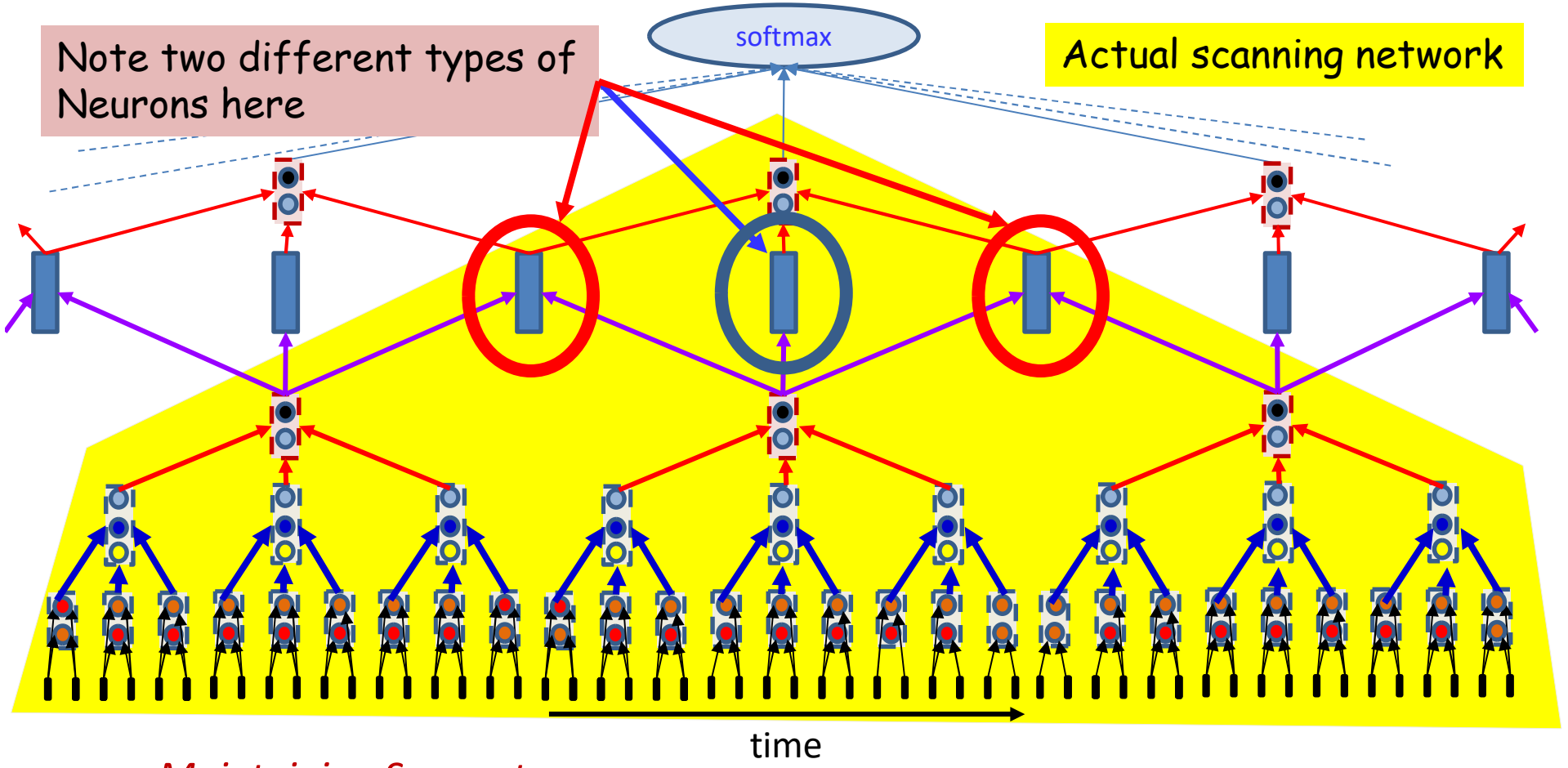
- Example of a network with one upsampling layer
- *Maintaining Symmetry:*
  - Vertical bars in the 4th layer are regularly arranged w.r.t. bars of layer 3
  - The pattern of values of upward weights for each of the three pink (3rd layer) bars is identical

Upsampling as a scanning network



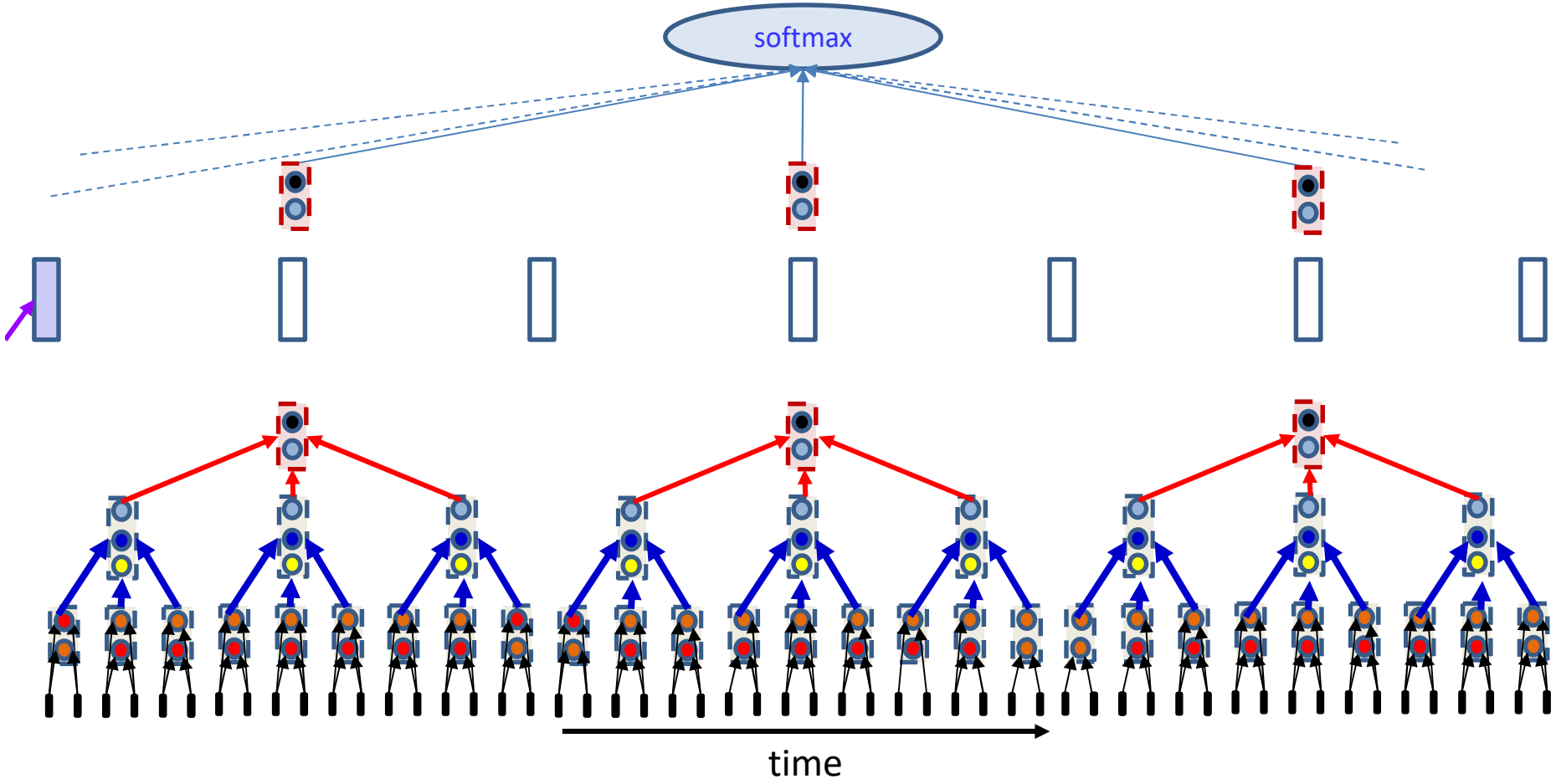
- *Maintaining Symmetry:*
  - Vertical bars in the 4th layer are regularly arranged w.r.t. bars of layer 3
  - The pattern of values of upward weights for each of the three pink (3rd layer) bars is identical

Upsampling as a scanning network



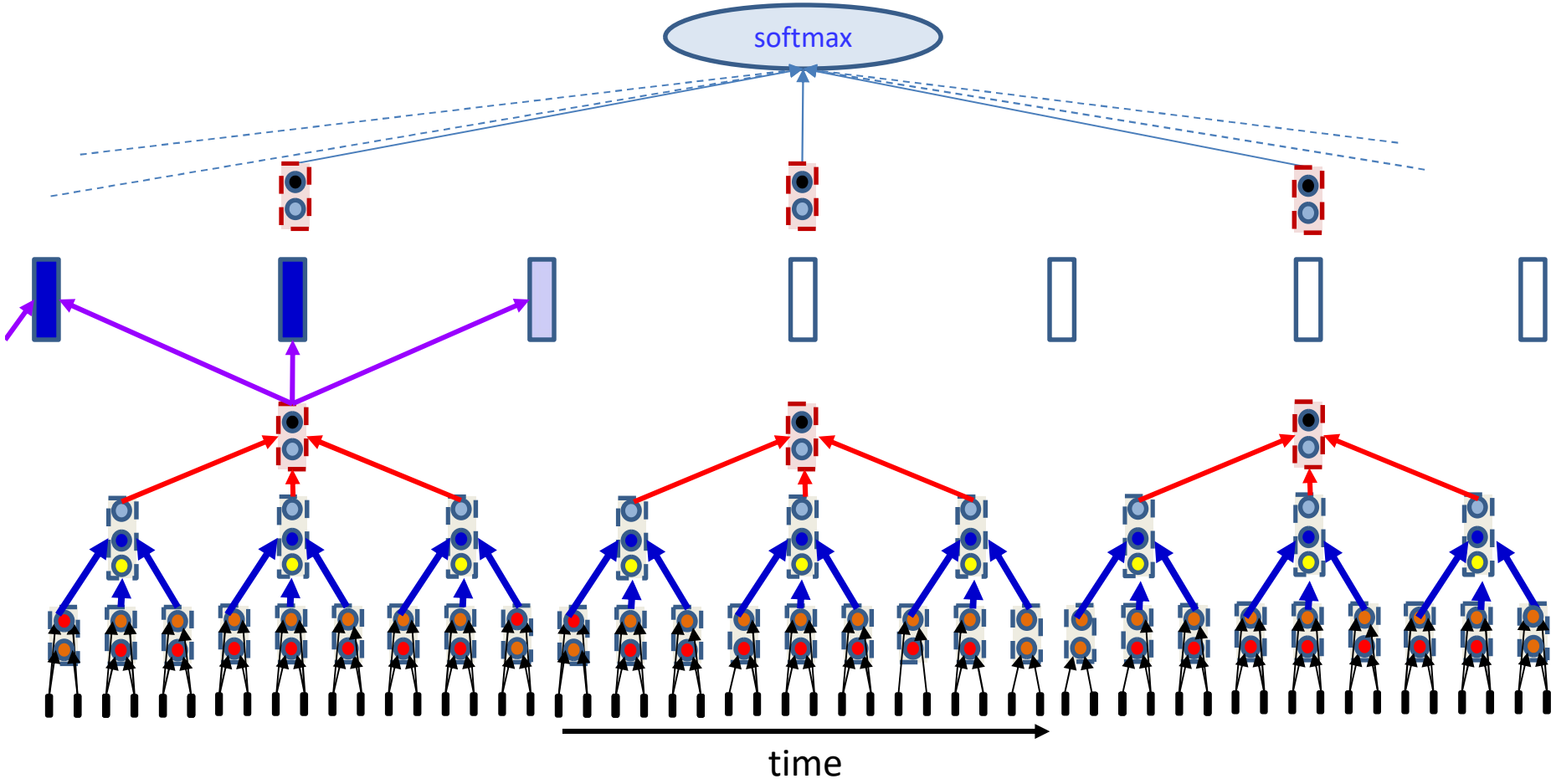
- *Maintaining Symmetry:*
  - Vertical bars in the 4th layer are regularly arranged w.r.t. bars of layer 3
  - The pattern of values of upward weights for each of the three pink (3rd layer) bars is identical

Scanning with increased-res layer



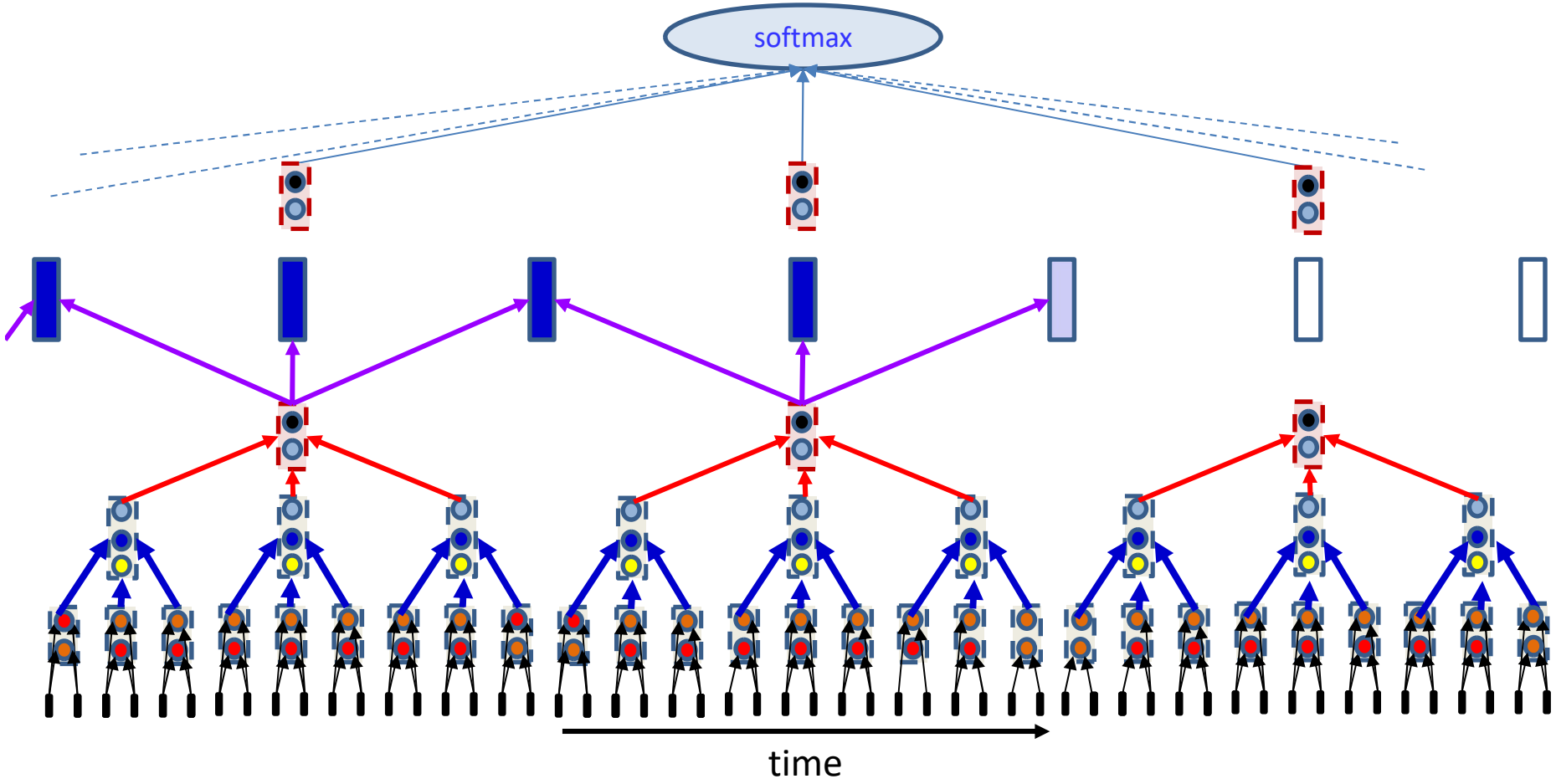
- Flow of info from bottom to top when implemented as a left-to-right scan
  - Note: Arrangement of vertical bars is predetermined by architecture

With layer of increased size



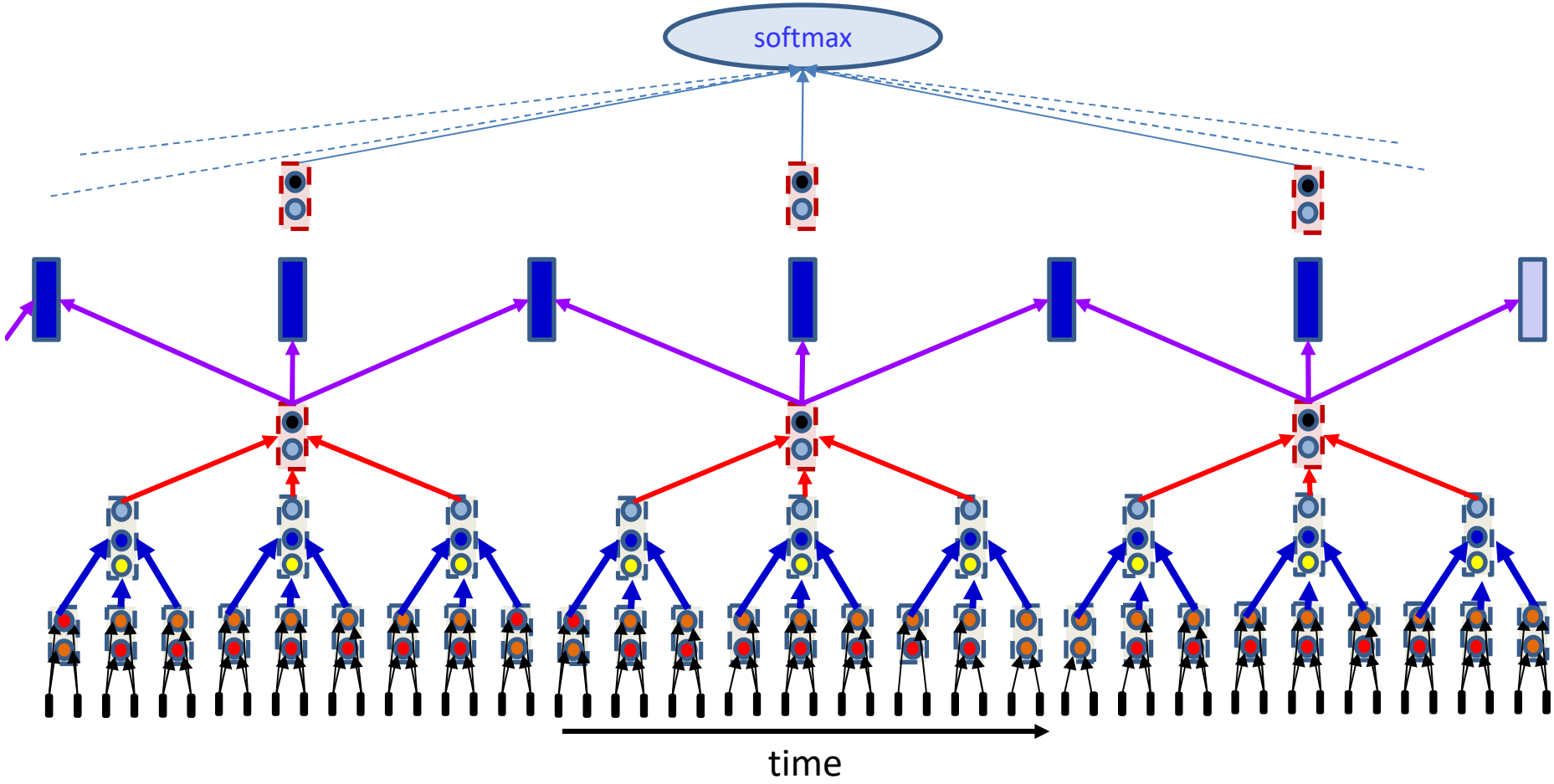
- Flow of info from bottom to top when implemented as a left-to-right scan
  - Note: Arrangement of vertical bars is predetermined by architecture

With layer of increased size



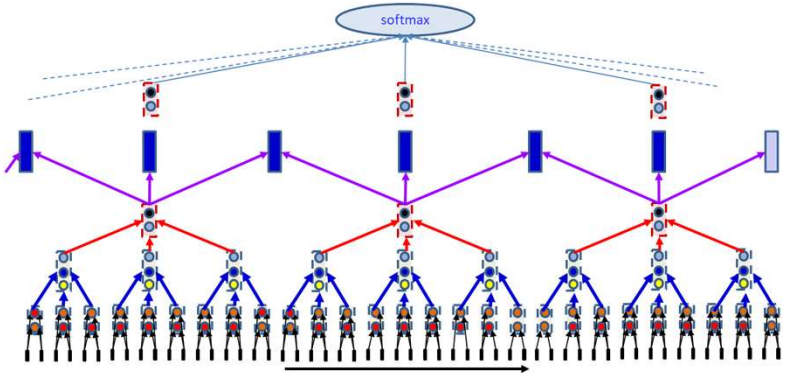
- Flow of info from bottom to top when implemented as a left-to-right scan
  - Note: Arrangement of vertical bars is predetermined by architecture

With layer of increased size



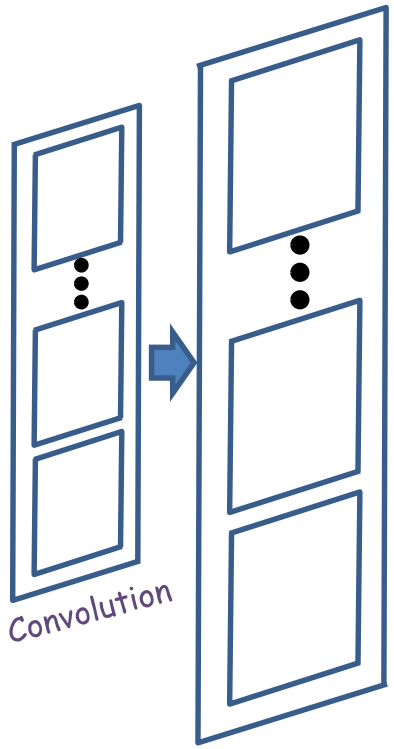
- Flow of info from bottom to top when implemented as a left-to-right scan
  - Note: Arrangement of vertical bars is predetermined by architecture

Transposed convolution



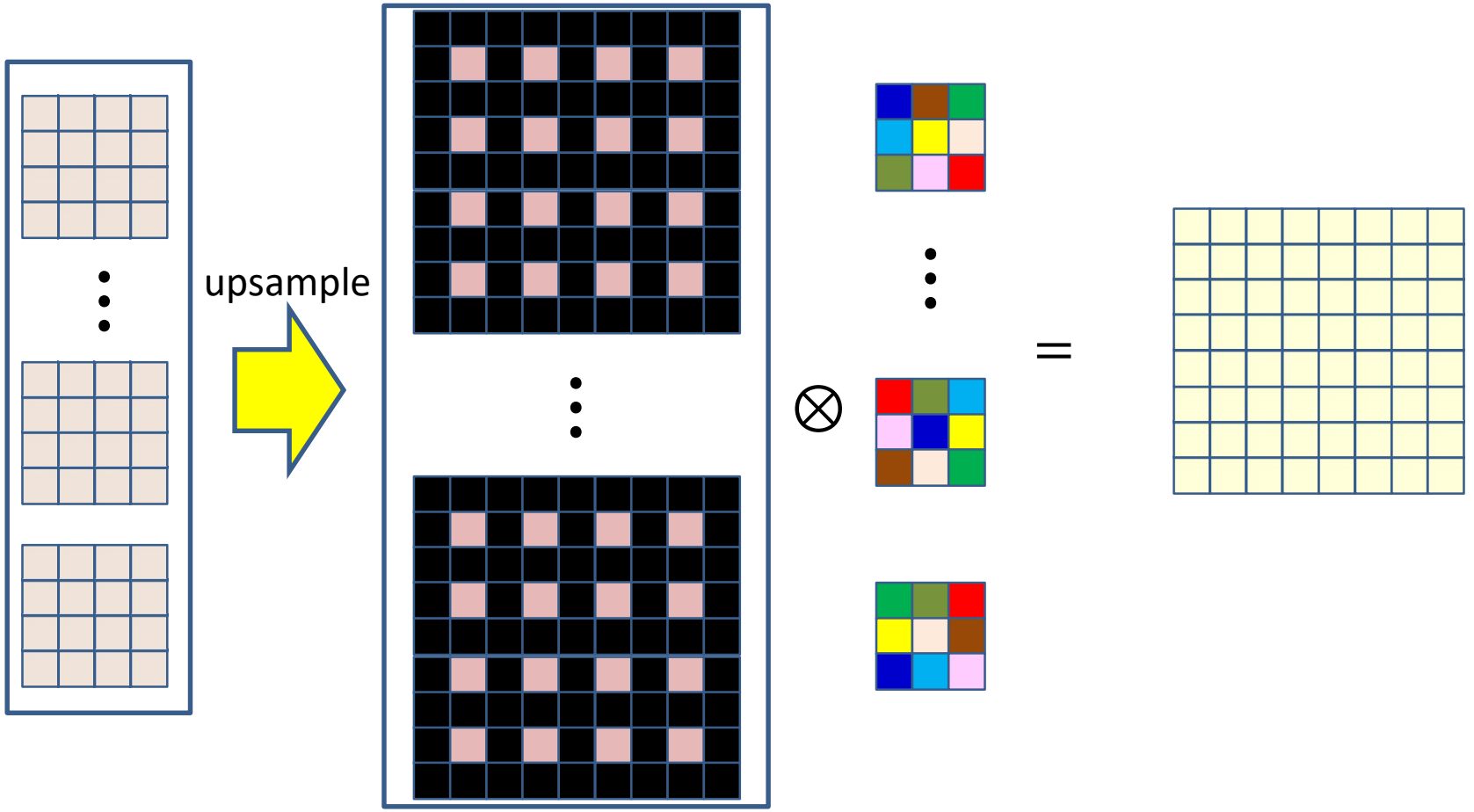
- Signal propagation rules are transposed for expanding layers
- In regular convolution, the affine value Z for a layer "pulls" Y values from the lower layer
  - In vector form
$$Z_l = W_l Y_{l-1}$$
  - The ith neuron:
$$z_l(i) = W_l(i,:)Y_{l-1}$$
  - Invokes the ith row of W_l
- In an upsampling layer the Y values are "pushed" to the upper Z
$$Z_l = \sum_j W_l(:,j)Y_{l-1}(j)$$
  - Invokes the jth column of W_l
  - Or alternately, the jth row of W_l^T
- Expanding operations are sometimes called *transpose* convolutions as a result
  - The primary operation uses the *transpose* of the convolutional filter

In 2-D



- Similar computation

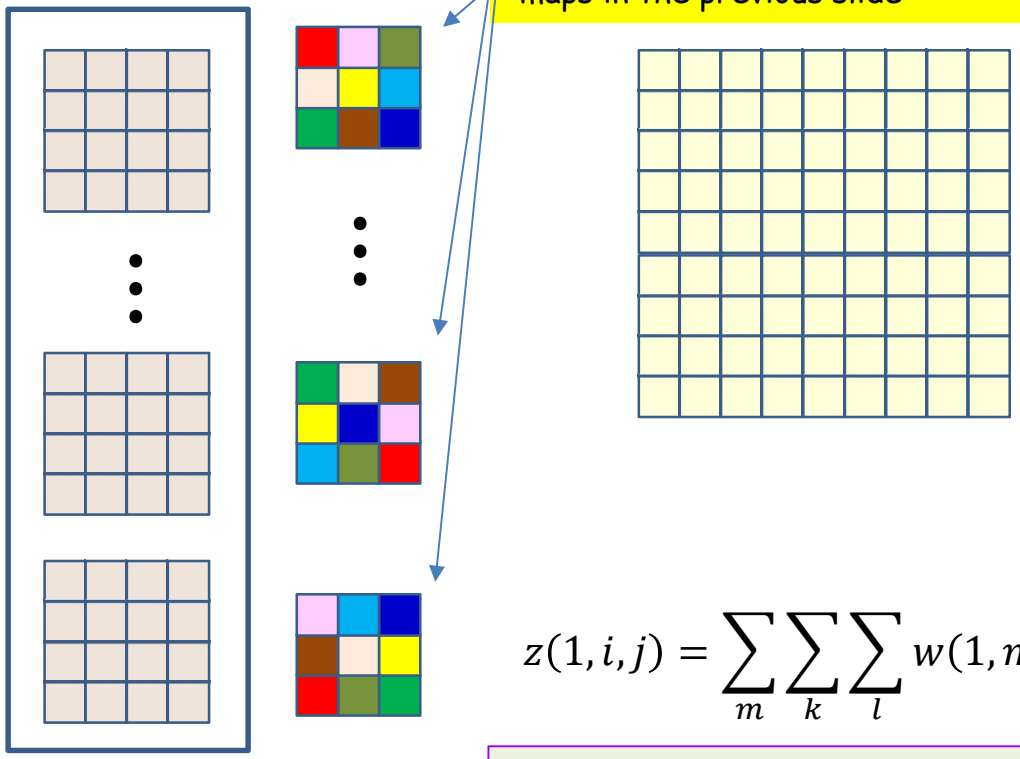
2D expanding convolution



- Upsample the input to the appropriate size by interpolating $b - 1$ zeros between adjacent elements to increase the size of the map by b
- Convolve with the filter with stride 1, to get the final upsampled output
  - Output map size also dependent on size of filter
  - Zero-pad upsampled input maps to ensure the output is exactly the desired size

2D expanding convolution in practice

These filters are "transposed" (flipped across the top-left to bottom-right diagonal) w.r.t. the scanning filters for the upsampled maps in the previous slide

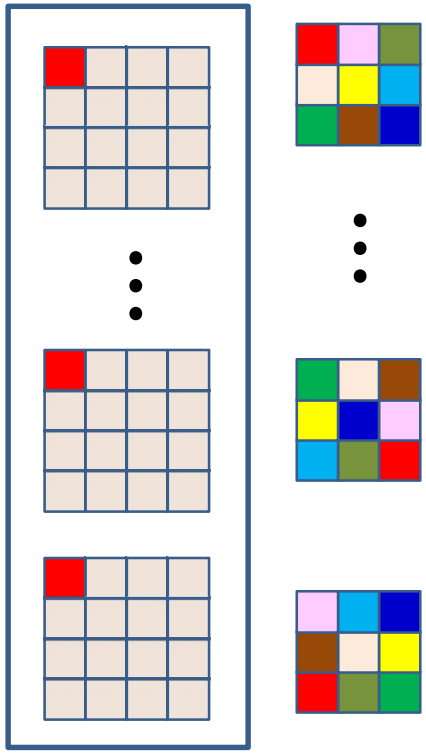


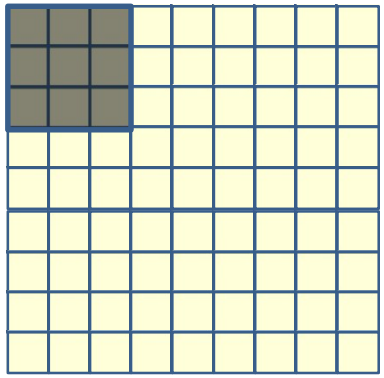
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb)I(m,k,l)$$

b is the "stride"
(scaling factor between the sizes of Z and Y)

- The parameters are filter size and output stride
- Output size is primarily decided by filter stride
  - Edges padded by $K-1$ rows/columns (K is width of filter)
  - Size of new map: $(bH + (K-1)) \times (bW + (K-1))$
  - Adjust filter stride and filter stride, and crop output map to ensure it is the right size

2D expanding convolution in practice



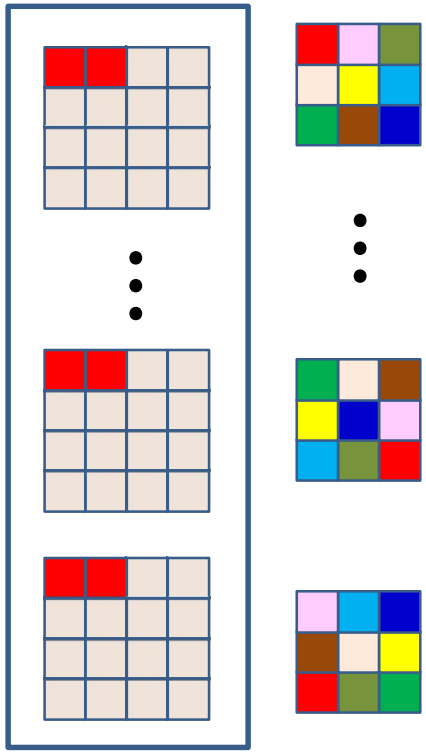


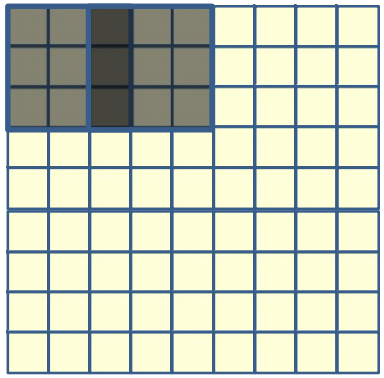
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb)I(m,k,l)$$

b is the "stride"
(scaling factor between the sizes of Z and Y)

- The parameters are filter size and output stride
- Output size is primarily decided by filter stride
  - Edges padded by $K-1$ rows/columns (K is width of filter)
  - Size of new map: $(bH + (K-1))\ x\ (bW + (K-1))$
  - Adjust filter stride and filter stride, and crop output map to ensure it is the right size

2D expanding convolution in practice



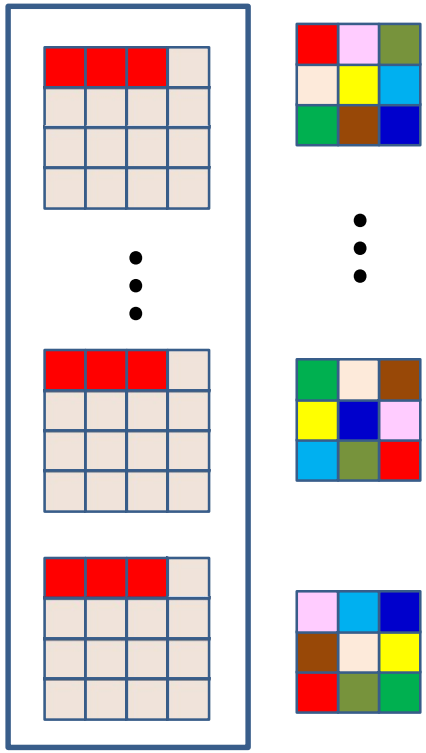


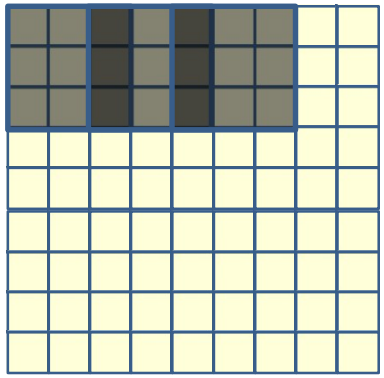
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb) I(m,k,l)$$

b is the "stride"
(scaling factor between the sizes of Z and Y)

- The parameters are filter size and output stride
- Output size is primarily decided by filter stride
  - Edges padded by $K-1$ rows/columns (K is width of filter)
  - Size of new map: $(bH + (K-1))\ x\ (bW + (K-1))$
  - Adjust filter stride and filter stride, and crop output map to ensure it is the right size

2D expanding convolution in practice



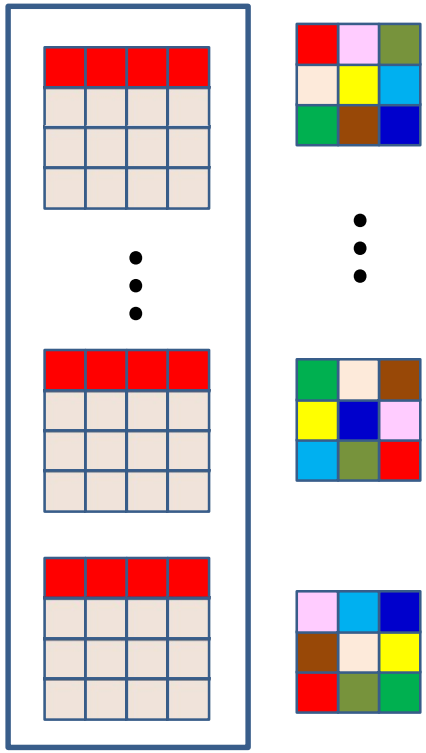


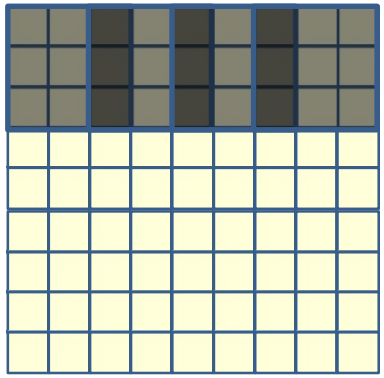
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb) I(m,k,l)$$

b is the "stride"
(scaling factor between the sizes of Z and Y)

- The parameters are filter size and output stride
- Output size is primarily decided by filter stride
  - Edges padded by $K-1$ rows/columns (K is width of filter)
  - Size of new map: $(bH + (K-1))\ x\ (bW + (K-1))$
  - Adjust filter stride and filter stride, and crop output map to ensure it is the right size

2D expanding convolution in practice



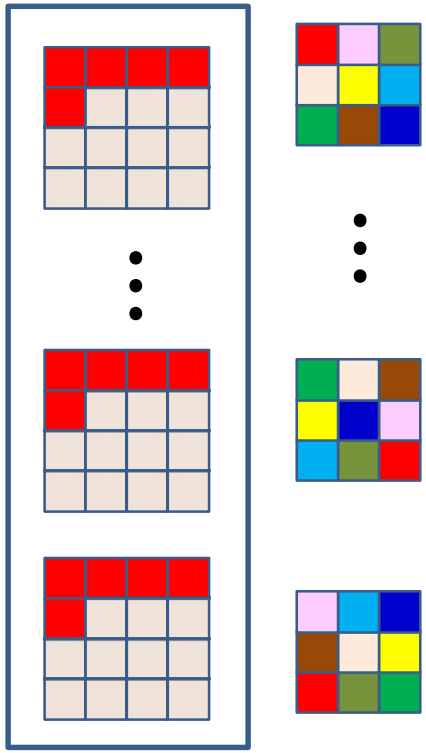


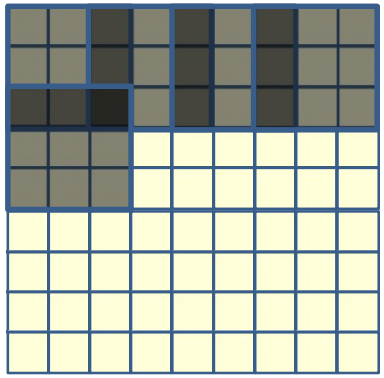
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb)I(m,k,l)$$

b is the "stride"
(scaling factor between the sizes of Z and Y)

- The parameters are filter size and output stride
- Output size is primarily decided by filter stride
  - Edges padded by $K-1$ rows/columns (K is width of filter)
  - Size of new map: $(bH + (K-1))\ x\ (bW + (K-1))$
  - Adjust filter stride and filter stride, and crop output map to ensure it is the right size

2D expanding convolution in practice



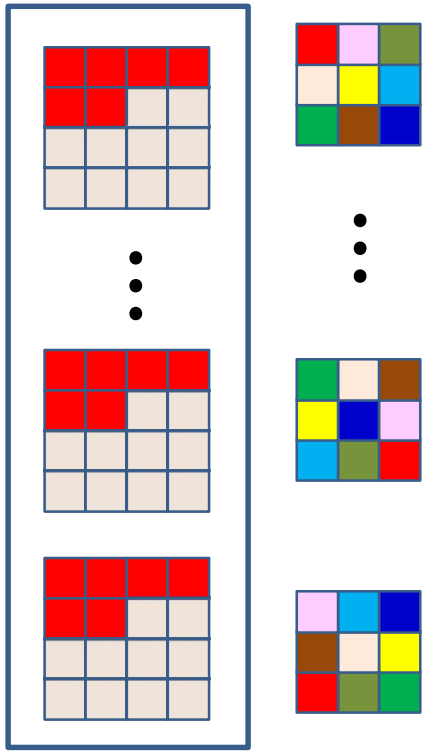


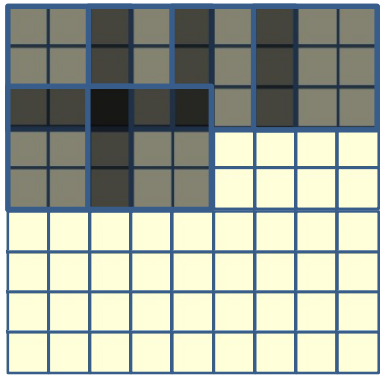
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb) I(m,k,l)$$

b is the "stride"
(scaling factor between the sizes of Z and Y)

- The parameters are filter size and output stride
- Output size is primarily decided by filter stride
  - Edges padded by $K-1$ rows/columns (K is width of filter)
  - Size of new map: $(bH + (K-1))\ x\ (bW + (K-1))$
  - Adjust filter stride and filter stride, and crop output map to ensure it is the right size

2D expanding convolution in practice



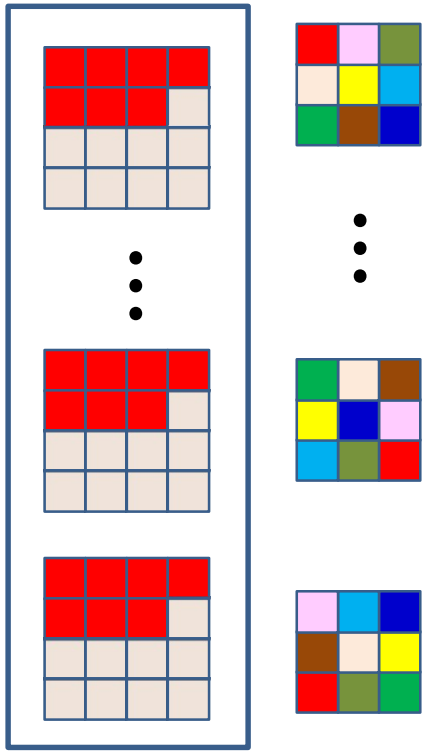


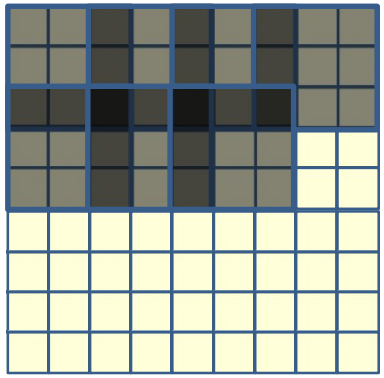
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb) I(m,k,l)$$

b is the "stride"
(scaling factor between the sizes of Z and Y)

- The parameters are filter size and output stride
- Output size is primarily decided by filter stride
  - Edges padded by $K-1$ rows/columns (K is width of filter)
  - Size of new map: $(bH + (K-1))\ x\ (bW + (K-1))$
  - Adjust filter stride and filter stride, and crop output map to ensure it is the right size

2D expanding convolution in practice



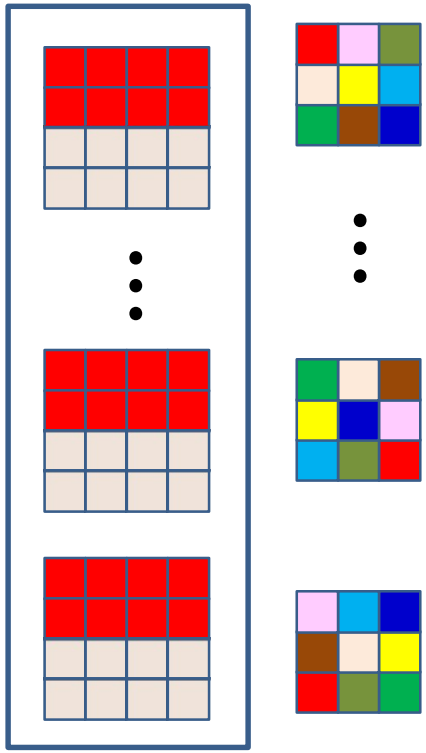


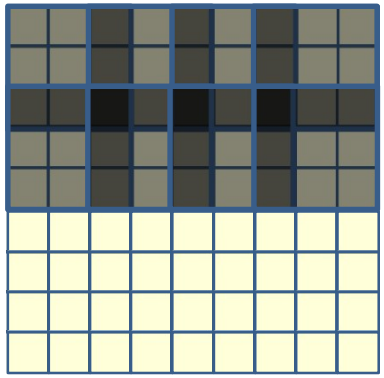
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb) I(m,k,l)$$

b is the "stride"
(scaling factor between the sizes of Z and Y)

- The parameters are filter size and output stride
- Output size is primarily decided by filter stride
  - Edges padded by $K-1$ rows/columns (K is width of filter)
  - Size of new map: $(bH + (K-1))\ x\ (bW + (K-1))$
  - Adjust filter stride and filter stride, and crop output map to ensure it is the right size

2D expanding convolution in practice



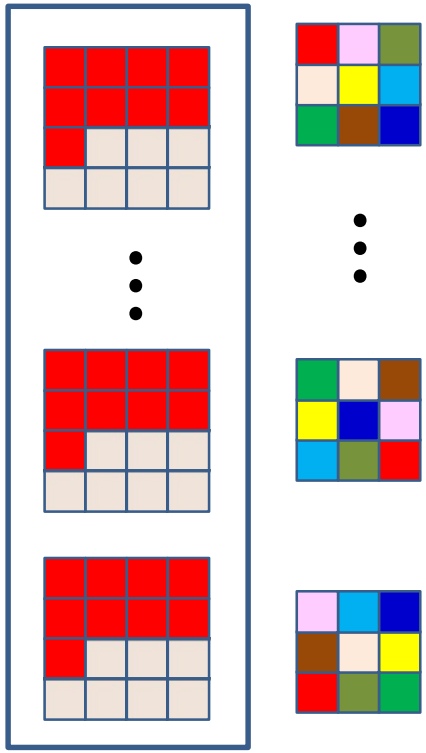


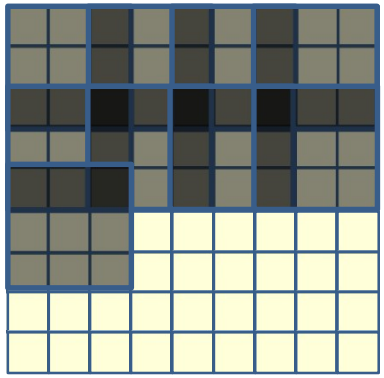
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb) I(m,k,l)$$

b is the "stride"
(scaling factor between the sizes of Z and Y)

- The parameters are filter size and output stride
- Output size is primarily decided by filter stride
  - Edges padded by $K-1$ rows/columns (K is width of filter)
  - Size of new map: $(bH + (K-1))\ x\ (bW + (K-1))$
  - Adjust filter stride and filter stride, and crop output map to ensure it is the right size

2D expanding convolution in practice



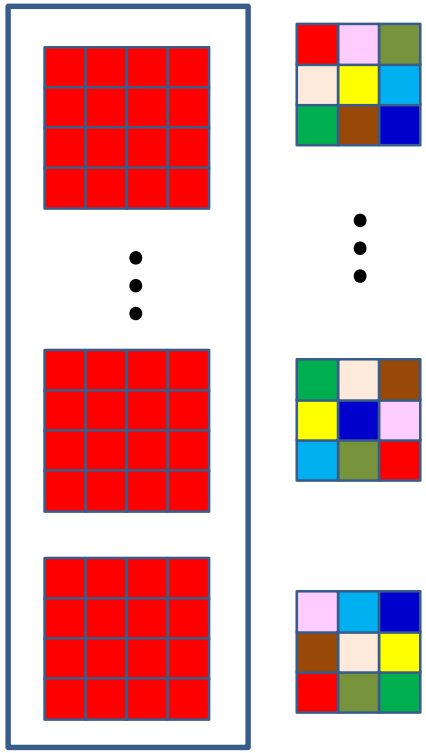


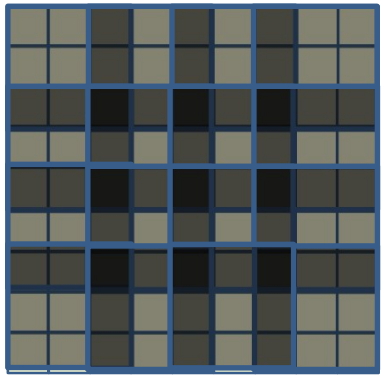
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb) I(m,k,l)$$

b is the "stride"
(scaling factor between the sizes of Z and Y)

- The parameters are filter size and output stride
- Output size is primarily decided by filter stride
  - Edges padded by $K-1$ rows/columns (K is width of filter)
  - Size of new map: $(bH + (K-1))\ x\ (bW + (K-1))$
  - Adjust filter stride and filter stride, and crop output map to ensure it is the right size

2D expanding convolution in practice





$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb) I(m,k,l)$$

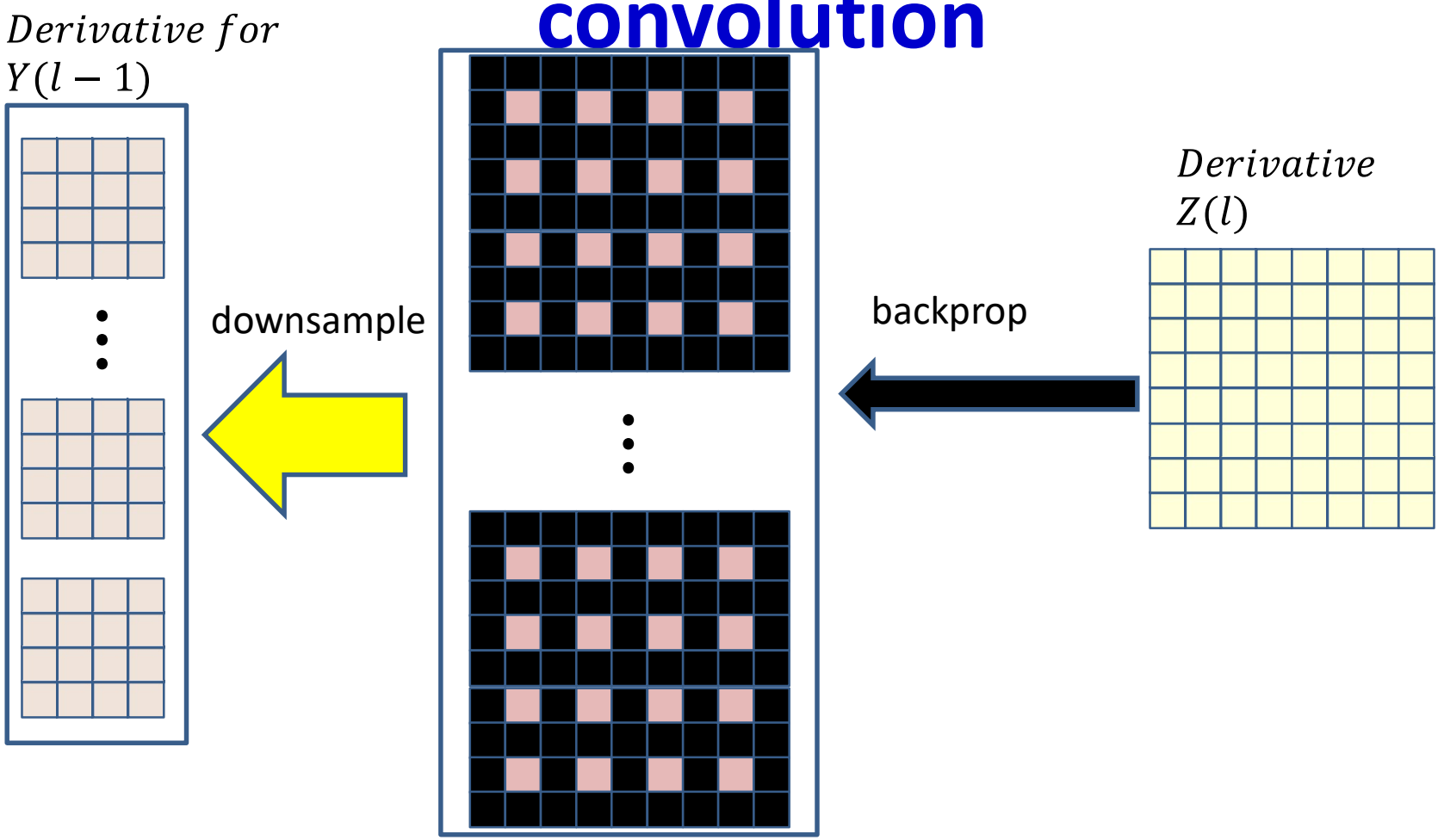
b is the "stride"
(scaling factor between the sizes of Z and Y)

- The parameters are filter size and output stride
- Output size is primarily decided by filter stride
  - Edges padded by $K-1$ rows/columns (K is width of filter)
  - Size of new map: $(bH + (K-1))\ x\ (bW + (K-1))$
  - Adjust filter stride and filter stride, and crop output map to ensure it is the right size

CNN: Expanding convolution layer l

```
Z(l) = zeros(Dl x ((W-1)b+K_l) x ((H-1)b+K_l)) # b = stride
for j = 1:D_l
  for x = 1:W
    for y = 1:H
      for i = 1:D_l-1
        for x' = 1:K_l
          for y' = 1:K_l
            z(l,j,(x-1)b+x',(y-1)b+y') +=
                        w(l,j,i,x',y')y(l-1,i,x,y)
```

Backprop through expanding convolution



- Backpropagation will give us derivatives for every element of the upsampled map
- Downsample the derivative map by dropping elements corresponding to zeros introduced during upsampling
- Continue backprop from there
- Actually easier in code…

CNN: Expanding convolution layer l

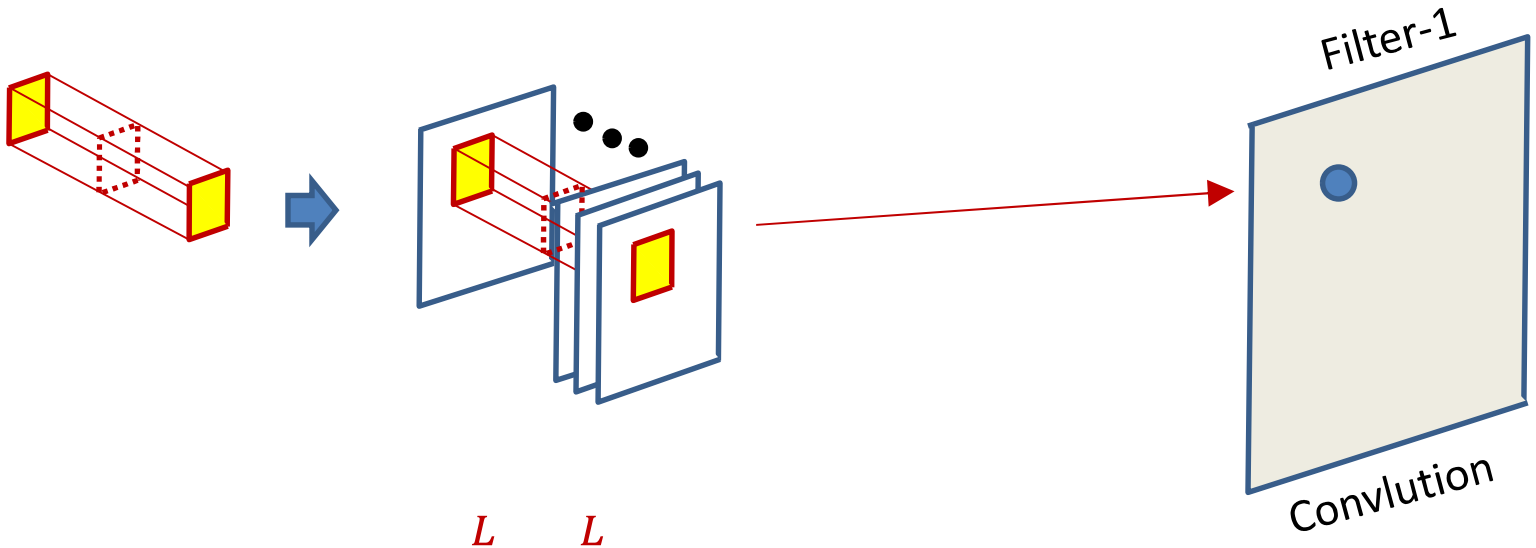
```
Z(l) = zeros(Dl x ((W-1)b+K_l) x ((H-1)b+K_l)) # b = stride
for j = 1:D_l
  for x = 1:W
    for y = 1:H
      for i = 1:D_l-1
        for x' = 1:K_l
          for y' = 1:K_l
            z(l,j,(x-1)b+x',(y-1)b+y') +=
                        w(l,j,i,x',y')y(l-1,i,x,y)
```

We leave the rather trivial issue of how to modify this code to compute the derivatives w.r.t `w` and `y` to you

Invariance

- CNNs are shift invariant
- What about rotation, scale or reflection invariance

Shift-invariance – a different perspective

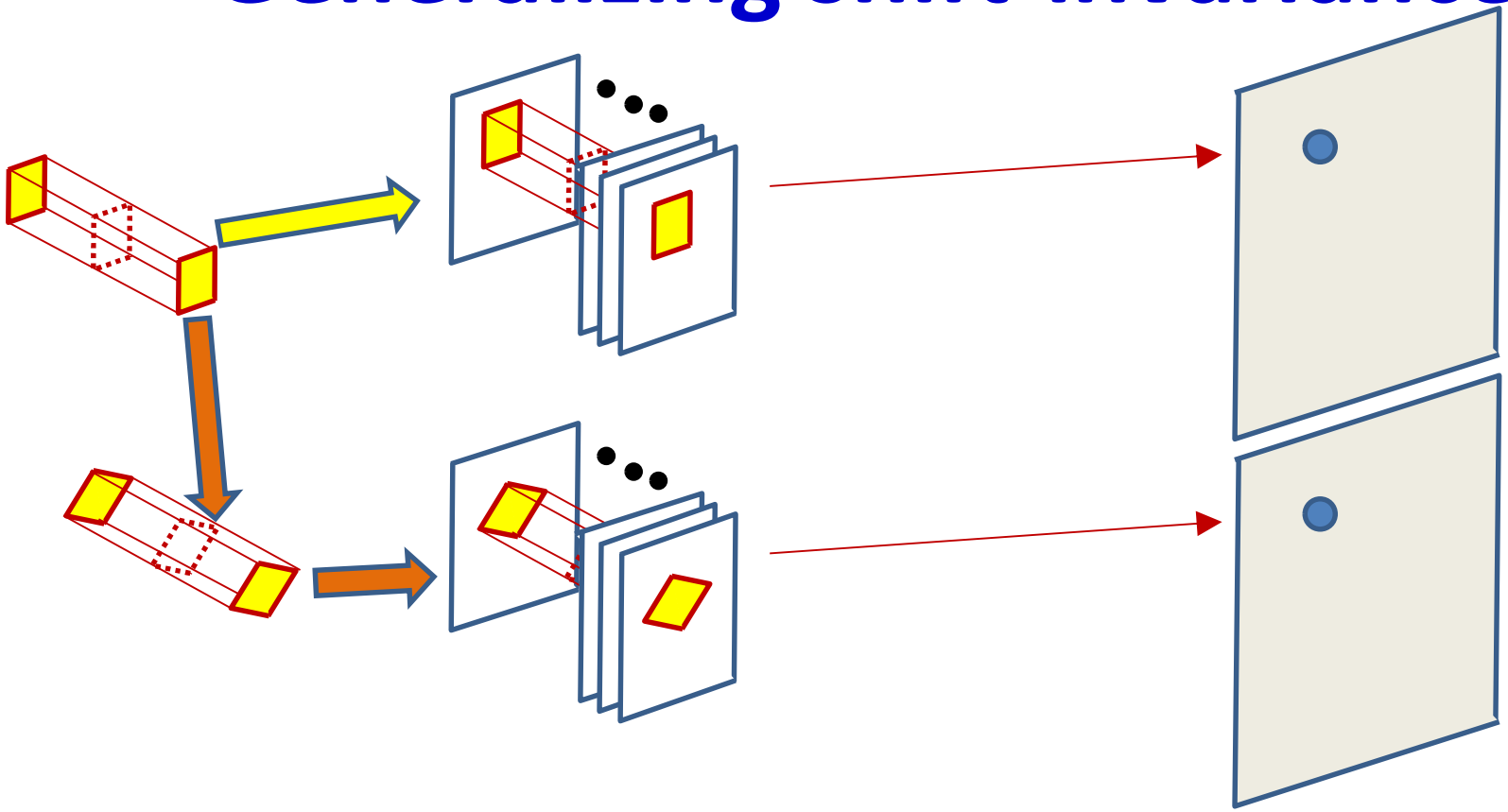


$$z(l,s,i,j) = \sum_{p} \sum_{k=1}^{L} \sum_{m=1}^{L} w(l,s,p,k,m) Y(l-1,p,i+k,j+m)$$

- We can rewrite this as so (tensor inner product)

$$z(s,i,j) = \boldsymbol{Y}.shift(\boldsymbol{w}(\boldsymbol{s}),i,j)$$

Generalizing shift-invariance



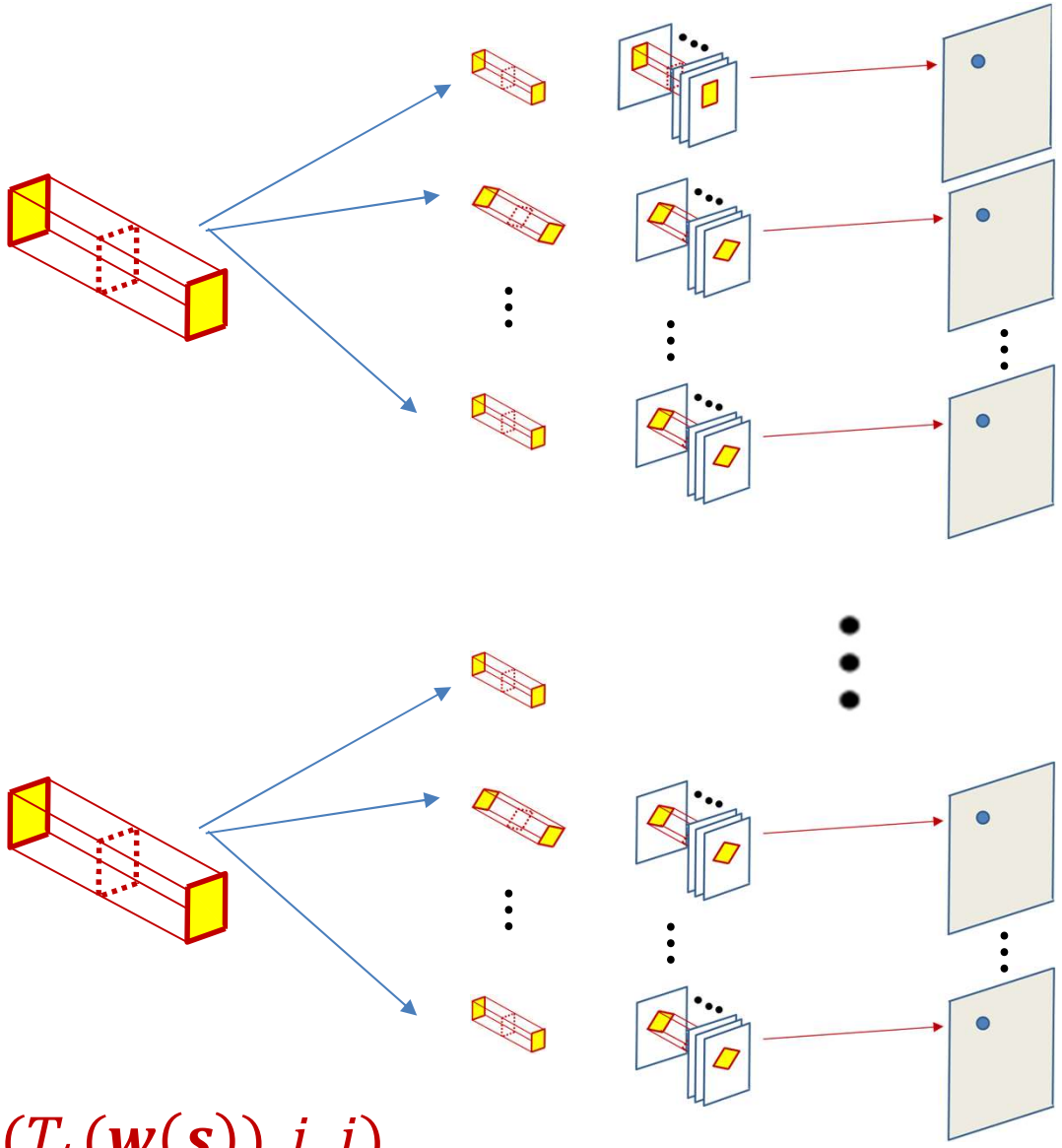
$$z_{regular}(s,i,j) = \boldsymbol{Y}.shift(\boldsymbol{w}(\boldsymbol{s}),i,j)$$

- Also find *rotated by 45 degrees* version of the pattern

$$z_{rot45}(s,i,j) = \boldsymbol{Y}.shift(rotate45(\boldsymbol{w}(\boldsymbol{s})),i,j)$$

Transform invariance

- More generally each filter produces a set of transformed (and shifted) maps
  - Set of transforms must be enumerated and discrete
  - E.g. discrete set of rotations and scaling, reflections etc.
- The network becomes invariant to all the transforms considered



$$z_{T_t}(s,i,j) = \boldsymbol{Y}.shift(T_t(\boldsymbol{w}(\boldsymbol{s})),i,j)$$

Regular CNN : single layer l

The weight W(l,j)is a 3D D_{l-1}xK_lxK_l tensor

```
for x = 1:W_{l-1}-K_l+1
  for y = 1:H_{l-1}-K_l+1
    for j = 1:D_l
      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))
```

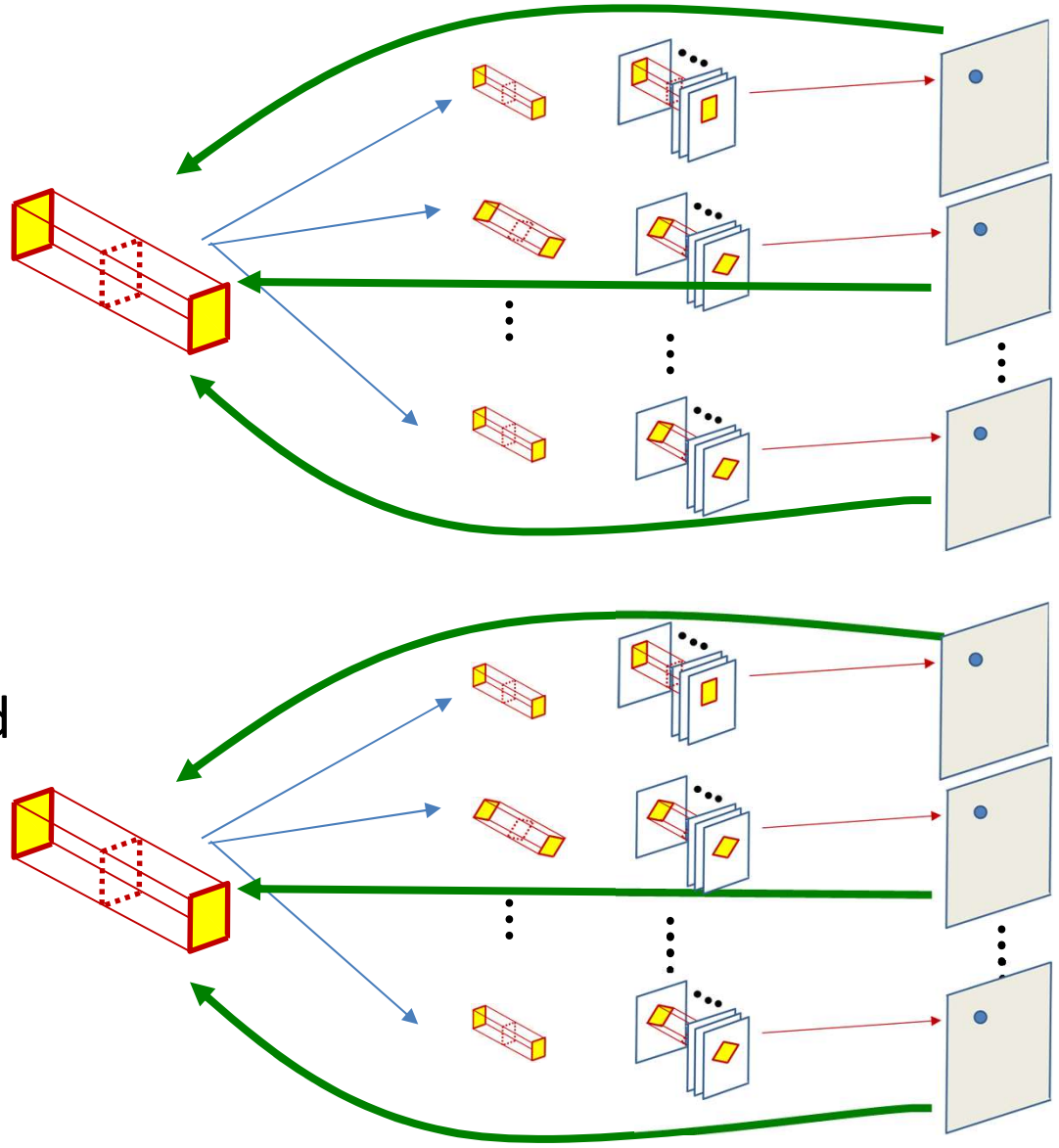
Transform invariance

The weight W(l,j) is a 3D D_{l-1}xK_lxK_l tensor

```
for x = 1:W_{l-1}-K_l+1
  for y = 1:H_{l-1}-K_l+1
    m = 1
    for j = 1:D_l
      for t in {Transforms} # enumerated transforms
        TW = T(W(l,j))
        segment = Y(l-1, :, x:x+K_l-1, y:y+K_l-1)#3D tensor
        z(l,m,x,y) = TW.segment #tensor inner prod.
        Y(l,m,x,y) = activation(z(l,m,x,y))
    m = m + 1
```

BP with transform invariance

- Derivatives flow back through the transforms to update individual filters
  - Need point correspondences between original and transformed filters
  - Left as an exercise



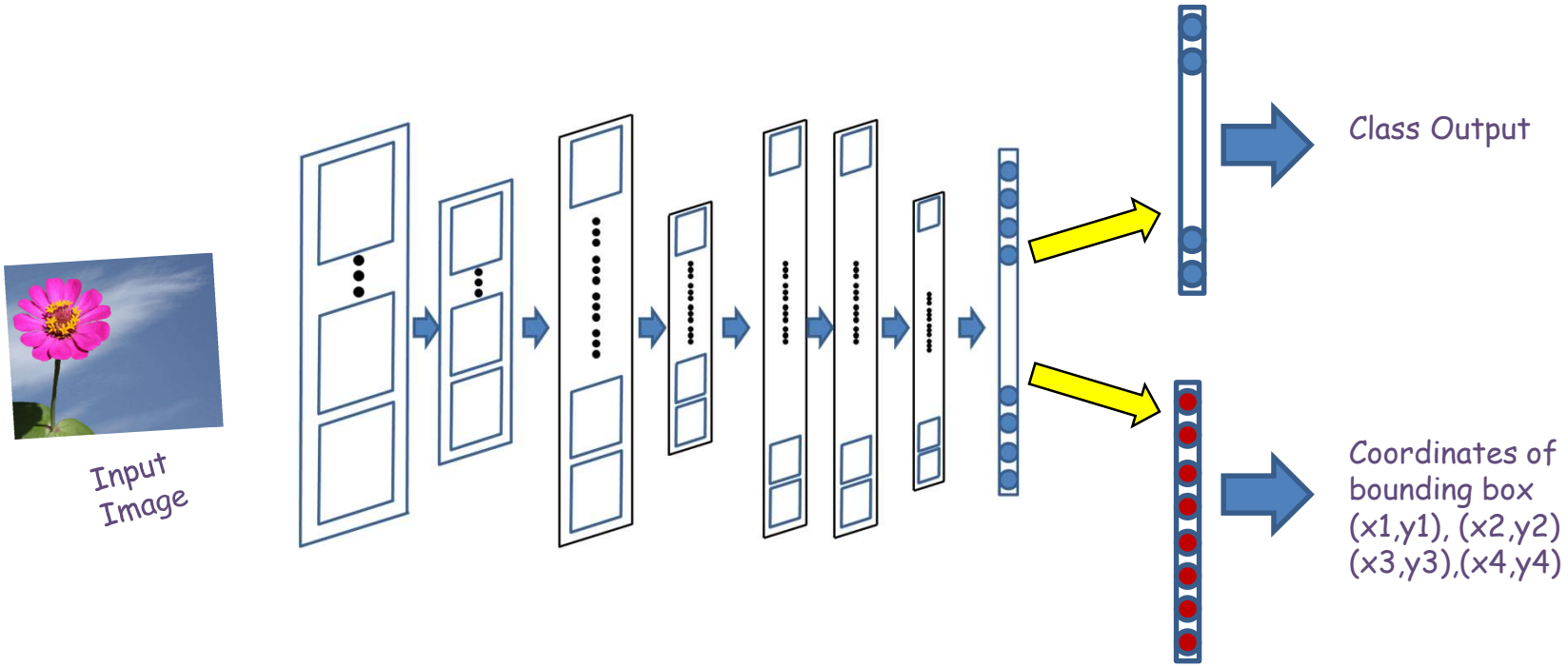
Story so far

- CNNs are shift-invariant neural-network models for shift-invariant pattern detection
  - Are equivalent to scanning with shared-parameter MLPs with distributed representations
- The parameters of the network can be learned through regular back propagation
- Like a regular MLP, individual layers may either increase or decrease the span of the representation learned
- The models can be easily modified to include invariance to other transforms
  - Although these tend to be computationally painful

But what about the exact location?

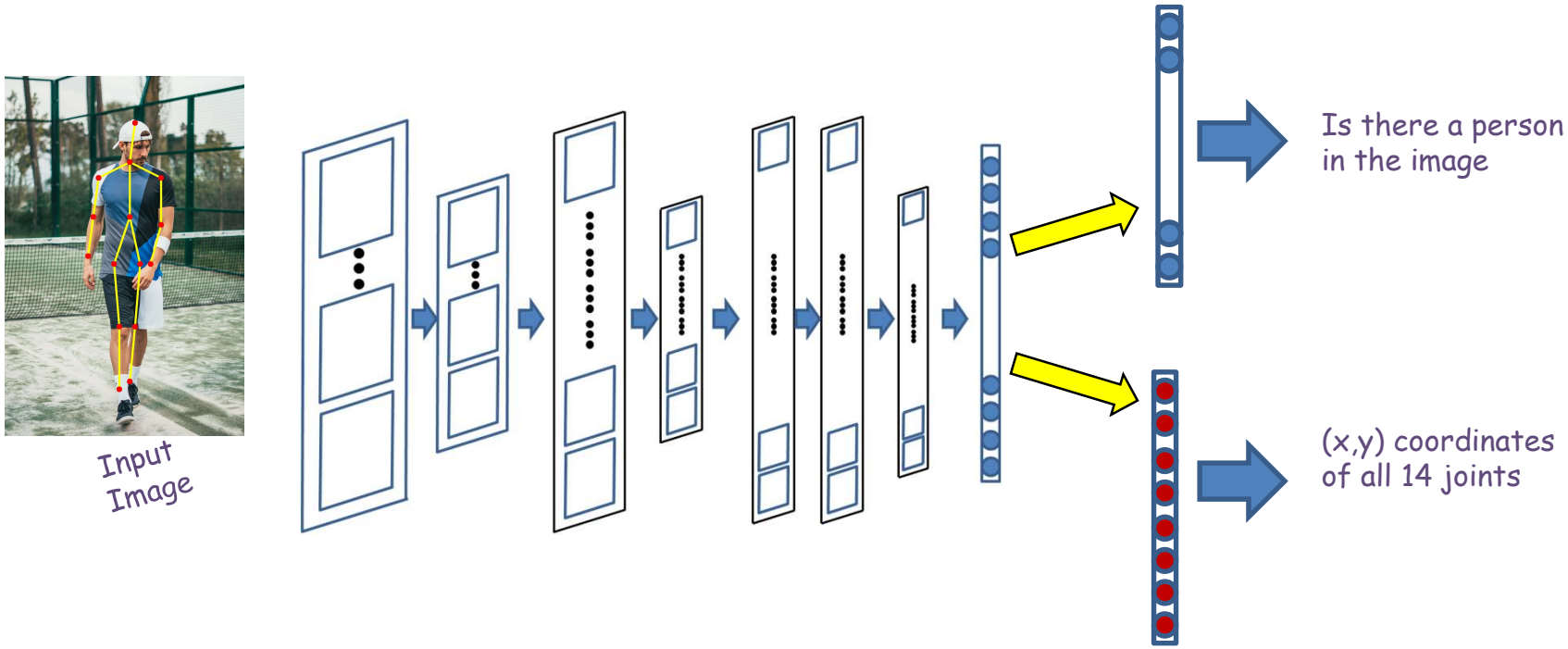
- We began with the desire to identify the picture as containing a flower, regardless of the position of the flower
  - Or more generally the class of object in the picture
- But can we detect the *position* of the main object?

Finding Bounding Boxes



- The flatten layer outputs to two separate output layers
- One predicts the class of the output
- The second predicts the corners of the bounding box of the object (8 coordinates) in all
- The divergence minimized is the sum of the cross-entropy loss of the classifier layer and L2 loss of the bounding-box predictor
  - Multi-task learning

Pose estimation

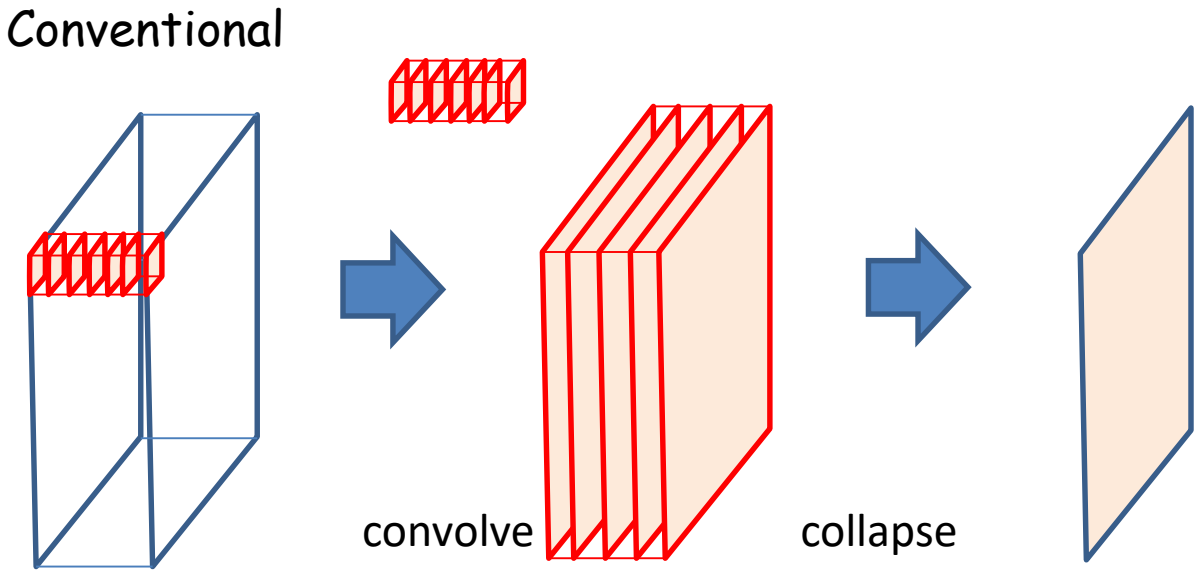


- Can use the same mechanism to predict the joints of a stick model
  - For pose estimation

Model variations

- *Very deep* networks
  - *100* or more layers in MLP
  - Formalism called “Resnet”
    - You will encounter this in your HWs
- *“Depth-wise” convolutions*
  - Instead of multiple independent filters with independent parameters, use common layer-wise weights and combine the layers differently for each filter

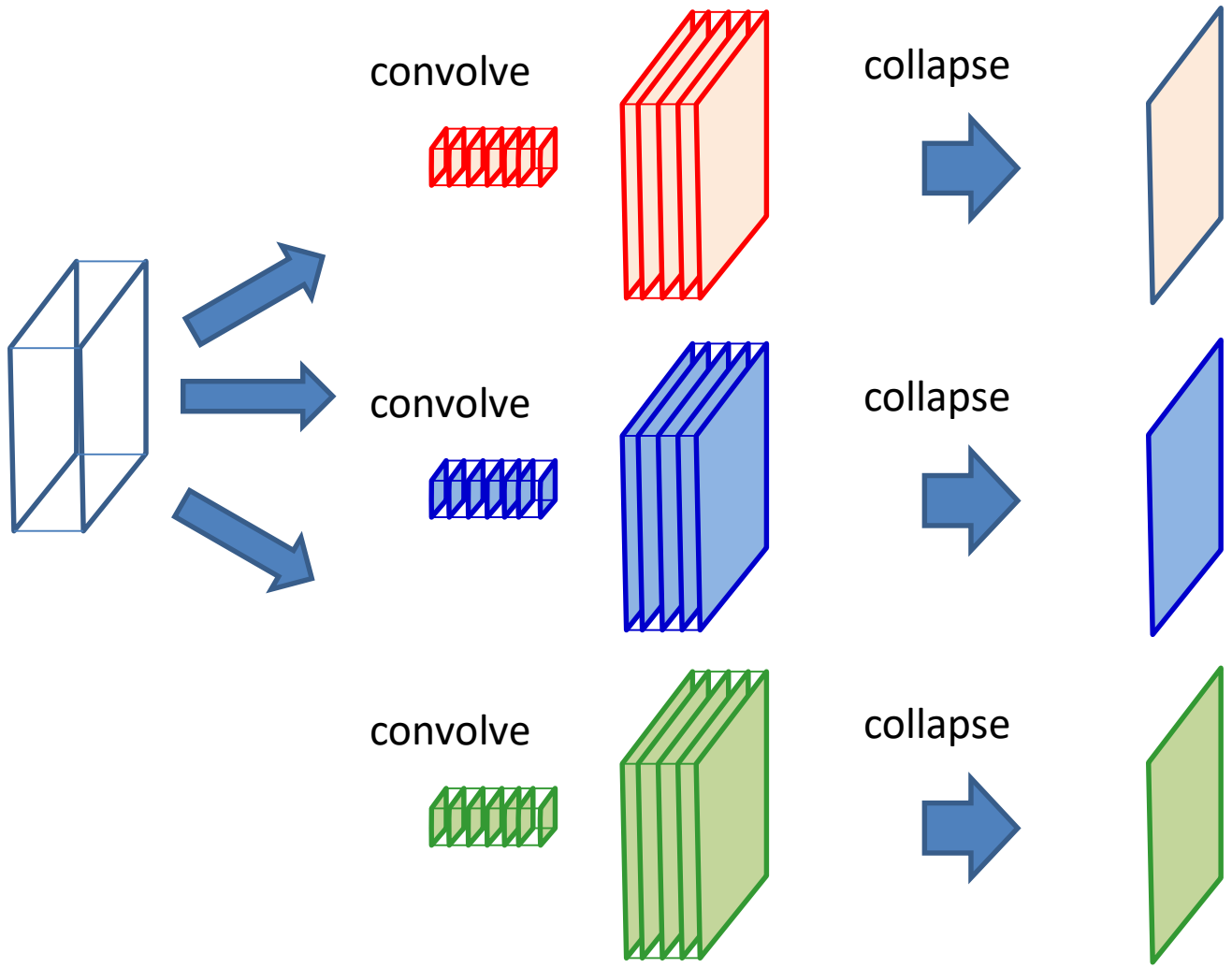
Conventional convolutions



- Alternate view of conventional convolution:

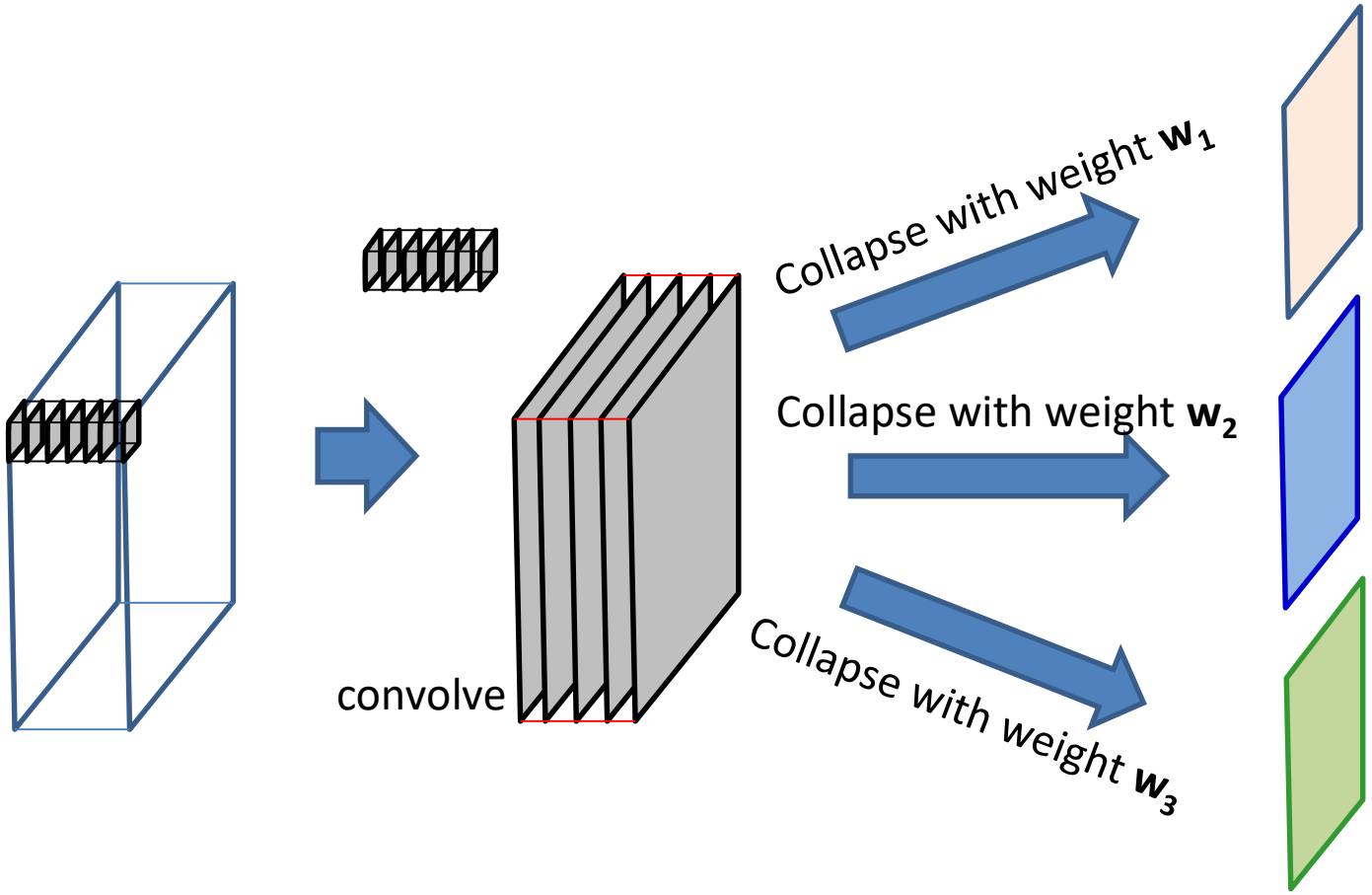
- E*ach layer of each filter* scans its corresponding map to produce a convolved map
- N input channels will require a filter with N layers
- The independent convolutions of each layer of the filter result in N convolved maps
- The N convolved maps are *added together* to produce the final output map (or channel) for that filter

Conventional convolutions



- This is done separately for each of the M filters producing M output maps (channels)

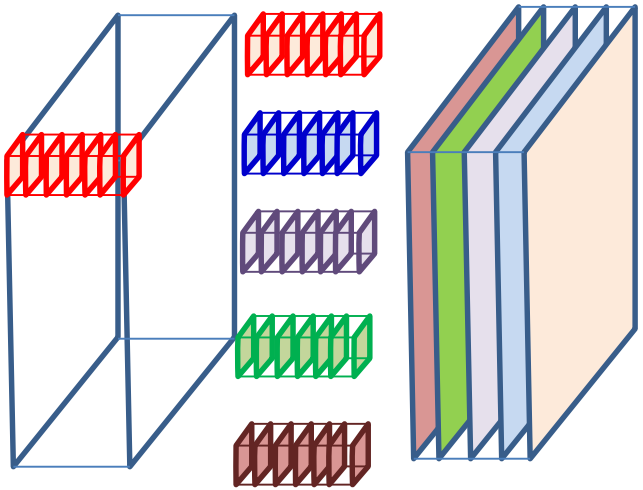
Depth-wise convolution



- In *depth-wise convolution* the convolution step is performed only once
- The simple summation is replaced by a *weighted* sum across channels
  - Different weights (for summation) produce different output channels

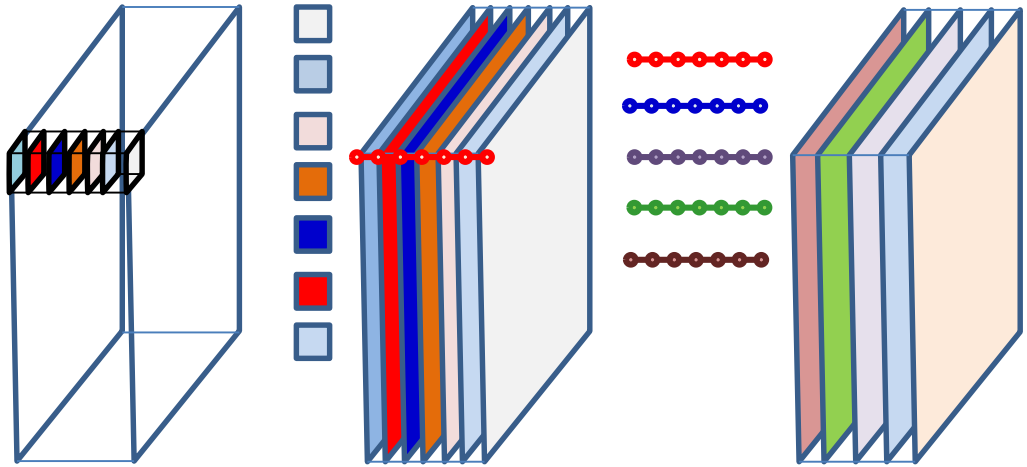
Conventional vs. depth-wise convolution

Conventional



- M input channels, N output channels:
- N independent MxKxK **3D** filters, which span all M input channels
- Each filter produces one output channel
- Total NMK^2 parameters

Depth-wise

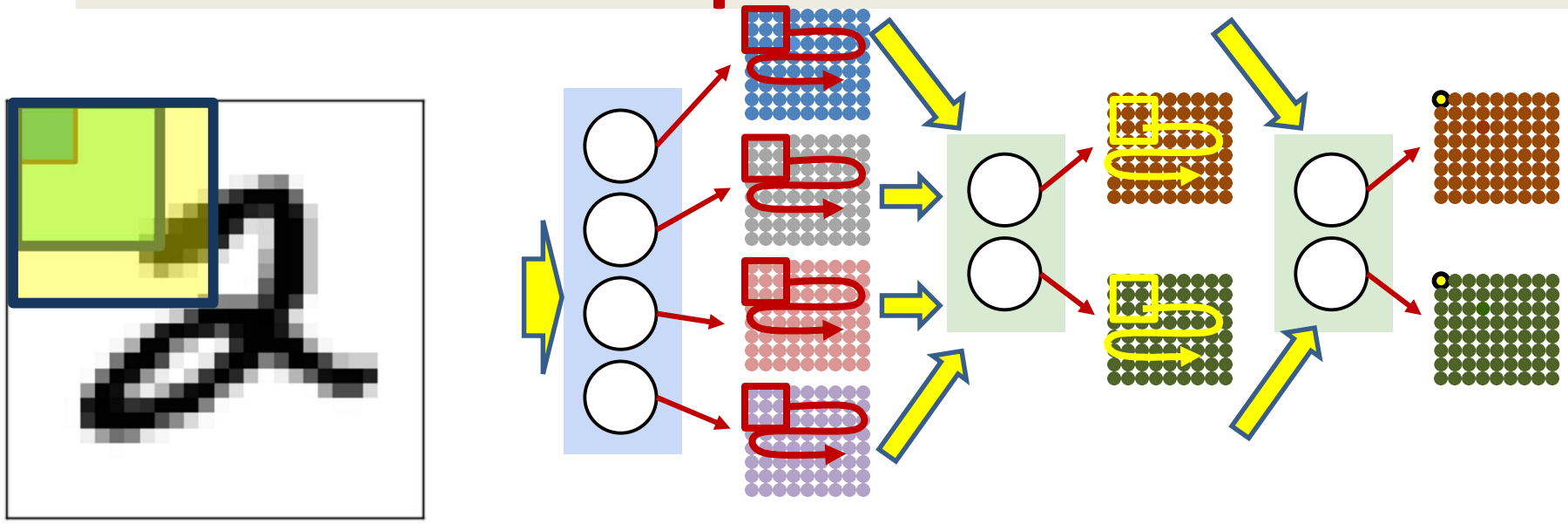


- M input channels, N output channels in 2 stages:
- Stage 1:
  - M independent KxK **2D** filters, one per input channel
  - Each filter applies to only one input channel
  - No. of output channels = no. of input channels
- Stage 2:
  - N Mx1x1 1D filters
  - Each applies to *one* 2D location across all M input channels
- Total $NM + MK^2$ parameters

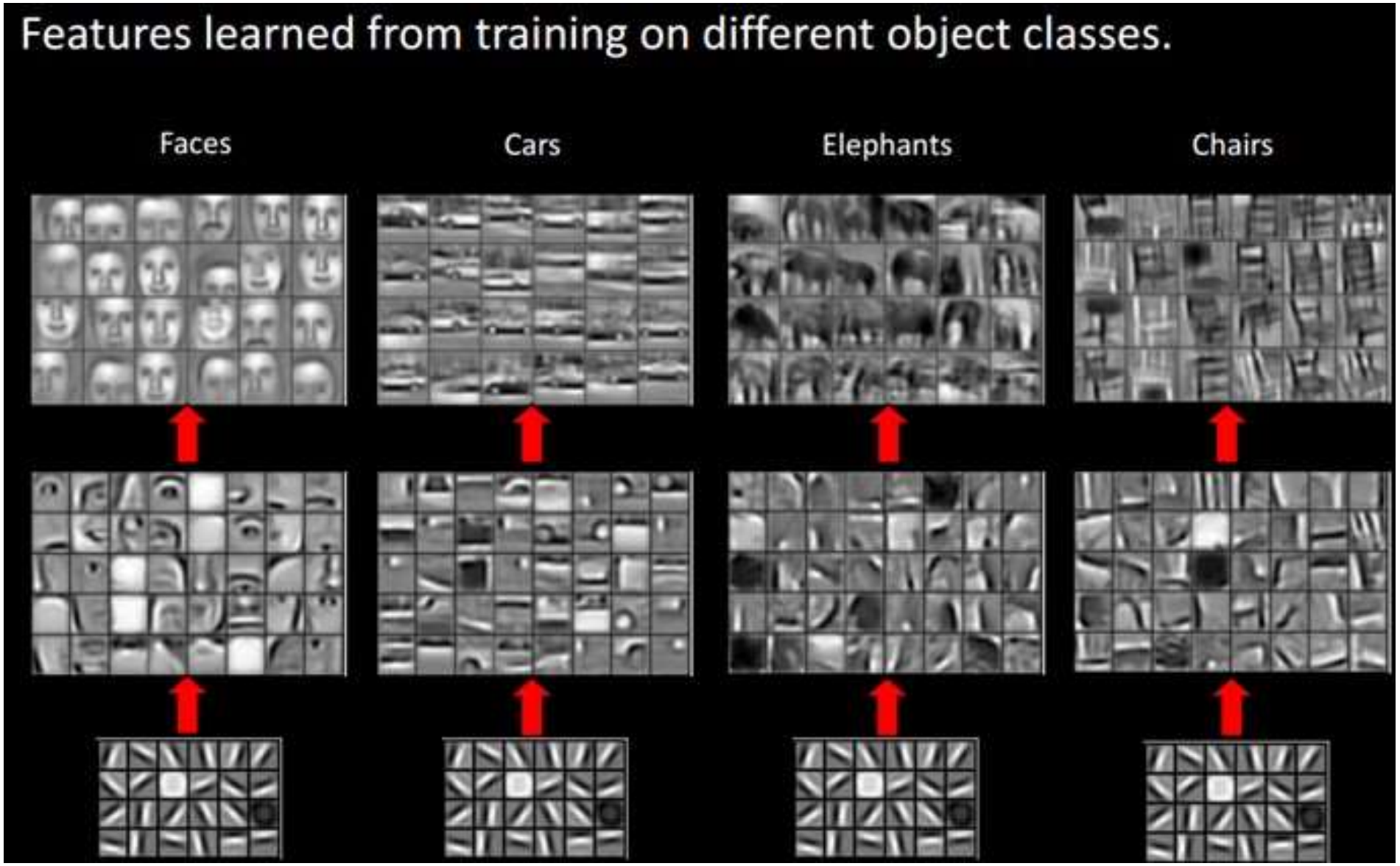
Story so far

- CNNs are shift-invariant neural-network models for shift-invariant pattern detection
  - Are equivalent to scanning with shared-parameter MLPs with distributed representations
- The parameters of the network can be learned through regular back propagation
- Like a regular MLP, individual layers may either increase or decrease the span of the representation learned
- The models can be easily modified to include invariance to other transforms
  - Although these tend to be computationally painful
- Can also make predictions related to the position and arrangement of target object through multi-task learning
- Several variations on the basic model exist to obtain greater parameter efficiency, better ability to compute derivatives, etc.

What do the filters learn? Receptive fields



- The pattern in the *input* image that each neuron sees is its "Receptive Field"
- The receptive field for a first layer neurons is simply its arrangement of weights
- For the higher level neurons, the actual receptive field is not immediately obvious and must be *calculated*
  - What patterns in the input do the neurons actually respond to?
  - We estimate it by setting the output of the neuron to 1, and learning the *input* by backpropagation

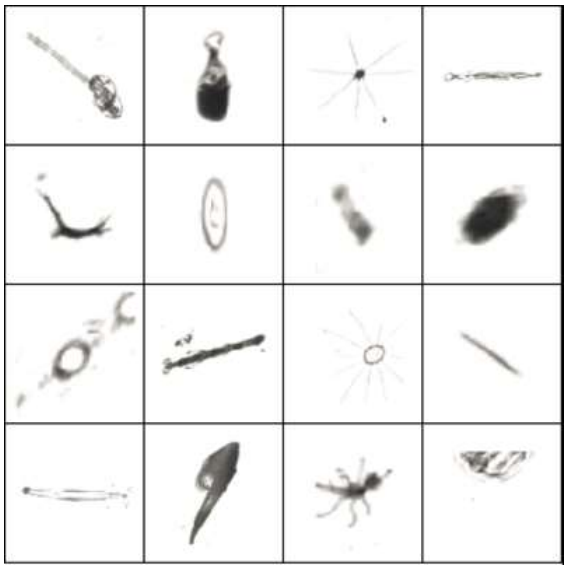

Features learned from training on different object classes.
Faces
Cars
Elephants
Chairs

Training Issues

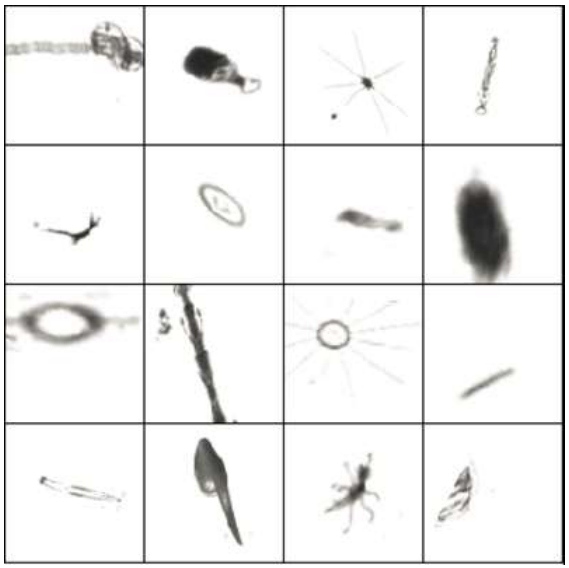
- Standard convergence issues
  - Solution: Adam or other momentum-style algorithms
  - Other tricks such as batch normalization
- The number of parameters can quickly become very large
- Insufficient training data to train well
  - Solution: Data augmentation

Data Augmentation

Original data



Augmented data



- rotation: uniformly chosen random angle between 0° and 360°
- translation: random translation between -10 and 10 pixels
- rescaling: random scaling with scale factor between 1/1.6 and 1.6 (log-uniform)
- flipping: yes or no (bernoulli)
- shearing: random shearing with angle between -20° and 20°
- stretching: random stretching with stretch factor between 1/1.3 and 1.3 (log-uniform)

Convolutional neural nets

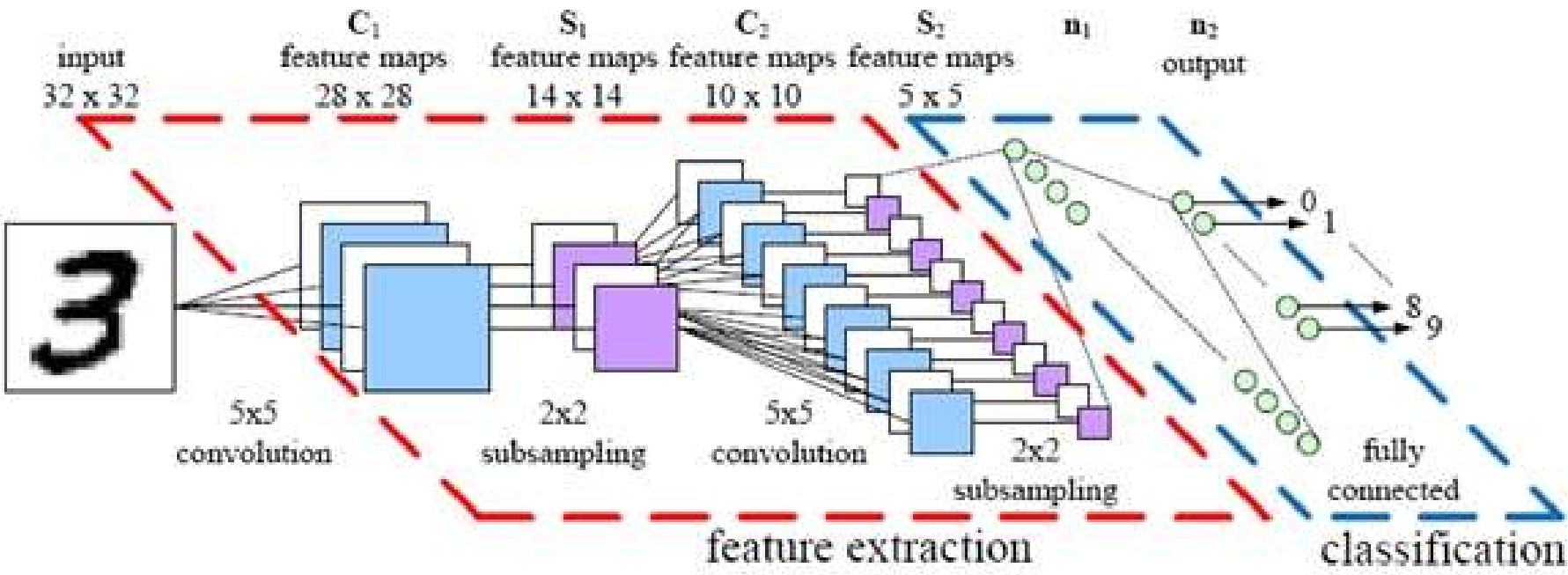
- One of *the* most frequently used nnet formalism today

- Used *everywhere*
  - Not just for image classification
  - Used in speech and audio processing
    - Convnets on *spectrograms*
  - Used in text processing

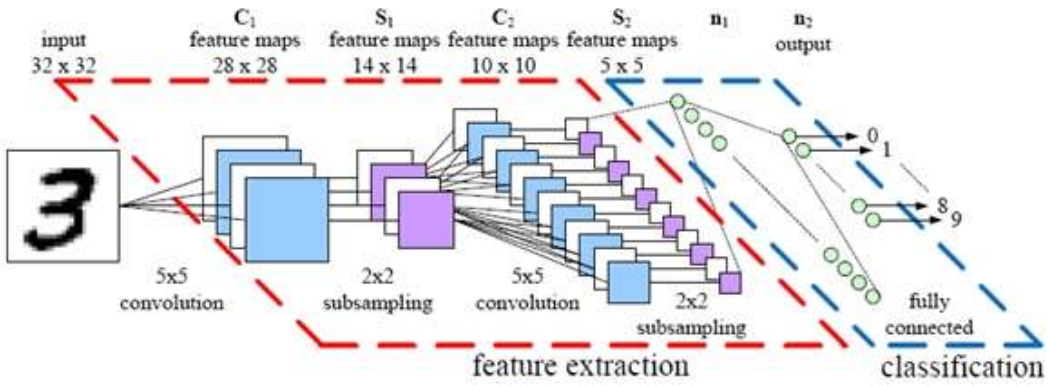
Nice visual example

- http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

Digit classification

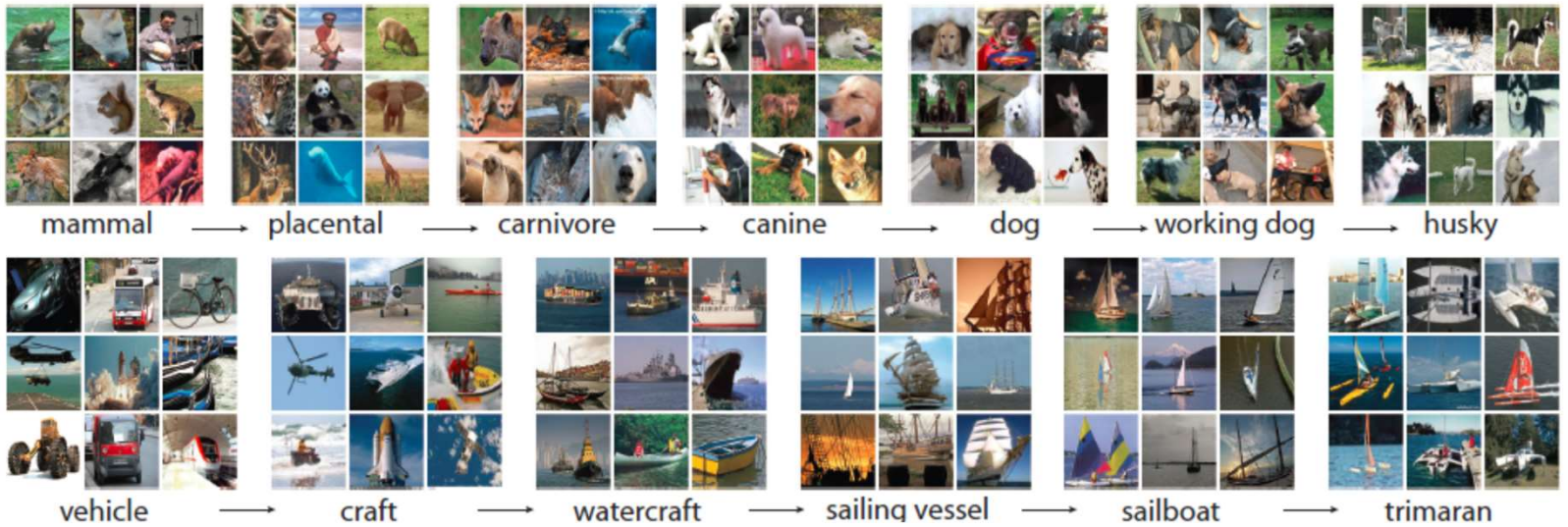


Le-net 5



- Digit recognition on MNIST (32x32 images)
  - **Conv1:** 6 5x5 filters in first conv layer (no zero pad), stride 1
    - Result: 6 28x28 maps
  - **Pool1:** 2x2 max pooling, stride 2
    - Result: 6 14x14 maps
  - **Conv2:** 16 5x5 filters in second conv layer, stride 1, no zero pad
    - Result: 16 10x10 maps
  - **Pool2:** 2x2 max pooling with stride 2 for second conv layer
    - Result 16 5x5 maps (400 values in all)
  - **FC:** Final MLP: 3 layers
    - 120 neurons, 84 neurons, and finally 10 output neurons

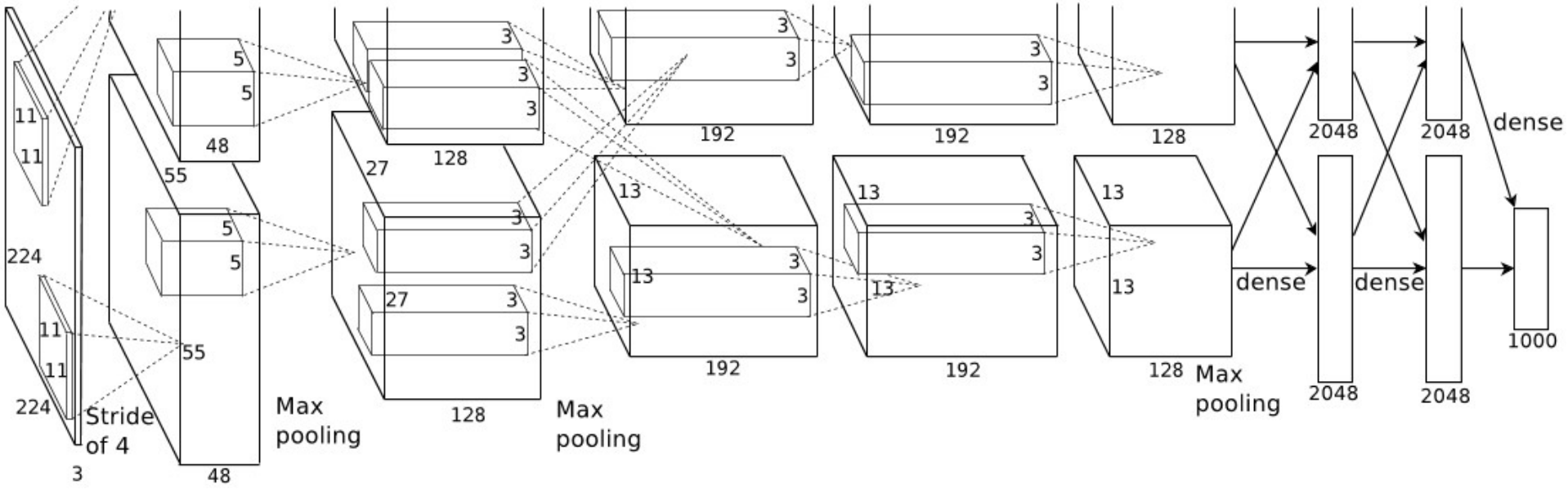
The imagenet task



- **Imagenet Large Scale Visual Recognition Challenge (ILSVRC)**
- http://www.image-net.org/challenges/LSVRC/
- Actual dataset: Many million images, thousands of categories
- For the evaluations that follow:
  - 1.2 million pictures
  - 1000 categories

AlexNet

- 1.2 million high-resolution images from ImageNet LSVRC-2010 contest
- 1000 different classes (softmax layer)
- NN configuration
  - NN contains 60 million parameters and 650,000 neurons,
  - 5 convolutional layers, some of which are followed by max-pooling layers
  - 3 fully-connected layers



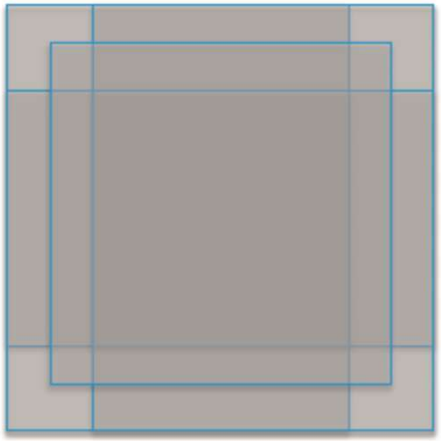
Krizhevsky, A., Sutskever, I. and Hinton, G. E. "ImageNet Classification with Deep Convolutional Neural Networks" NIPS 2012: Neural Information Processing Systems, Lake Tahoe, Nevada

Krizhevsky et. al.

- Input: 227x227x3 images
- Conv1: 96 11x11 filters, stride 4, no zeropad
- Pool1: 3x3 filters, stride 2
- “Normalization” layer [Unnecessary]
- Conv2: 256 5x5 filters, stride 2, zero pad
- Pool2: 3x3, stride 2
- Normalization layer [Unnecessary]
- Conv3: 384 3x3, stride 1, zeropad
- Conv4: 384 3x3, stride 1, zeropad
- Conv5: 256 3x3, stride 1, zeropad
- Pool3: 3x3, stride 2
- FC: 3 layers,
  - 4096 neurons, 4096 neurons, 1000 output neurons

Alexnet: Total parameters

- 650K neurons
- 60M parameters
- 630M connections



10 patches

- Testing: Multi-crop
  - Classify different shifts of the image and vote over the lot!

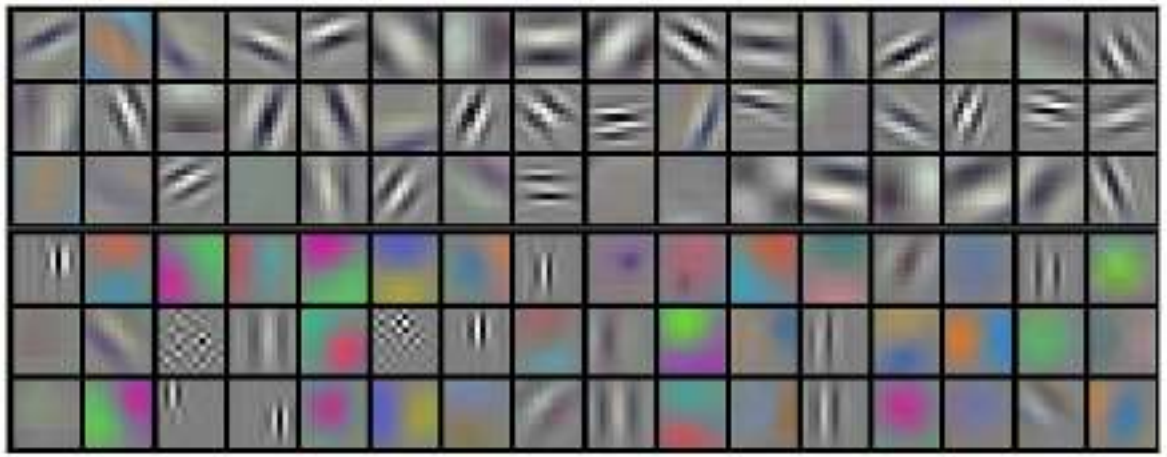
Learning magic in Alexnet

- **Activations were RELU**
  - Made a large difference in convergence
- "Dropout" – 0.5 (in FC layers only)
- *Large amount of data augmentation*
- SGD with mini batch size 128
- Momentum, with momentum factor 0.9
- L2 weight decay 5e-4
- Learning rate: 0.01, decreased by 10 every time validation accuracy plateaus
- Evaluated using: Validation accuracy

- **Final top-5 error: 18.2% with a single net, 15.4% using an ensemble of 7 networks**
  - **Lowest prior error using conventional classifiers: > 25%**

ImageNet

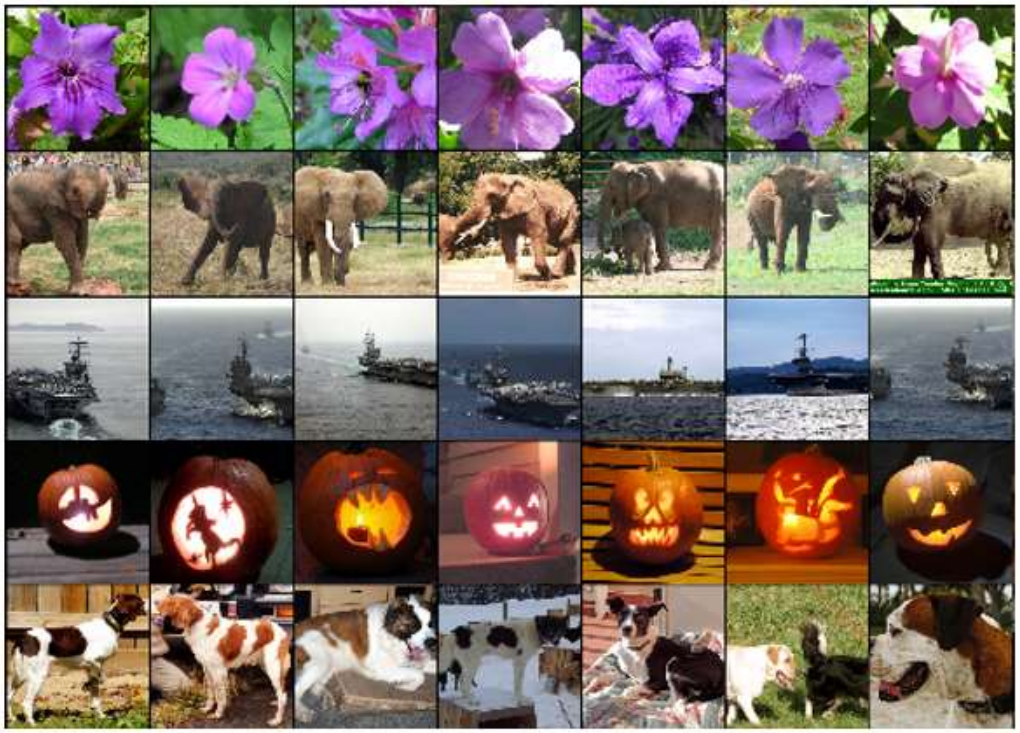
Figure 3: 96 convolutional kernels of size 11×11×3 learned by the first convolutional layer on the 224×224×3 input images. The top 48 kernels were learned on GPU 1 while the bottom 48 kernels were learned on GPU 2. See Section 6.1 for details.



Krizhevsky, A., Sutskever, I. and Hinton, G. E. "ImageNet Classification with Deep Convolutional Neural Networks" NIPS 2012: Neural Information Processing Systems, Lake Tahoe, Nevada

The net actually *learns* features!

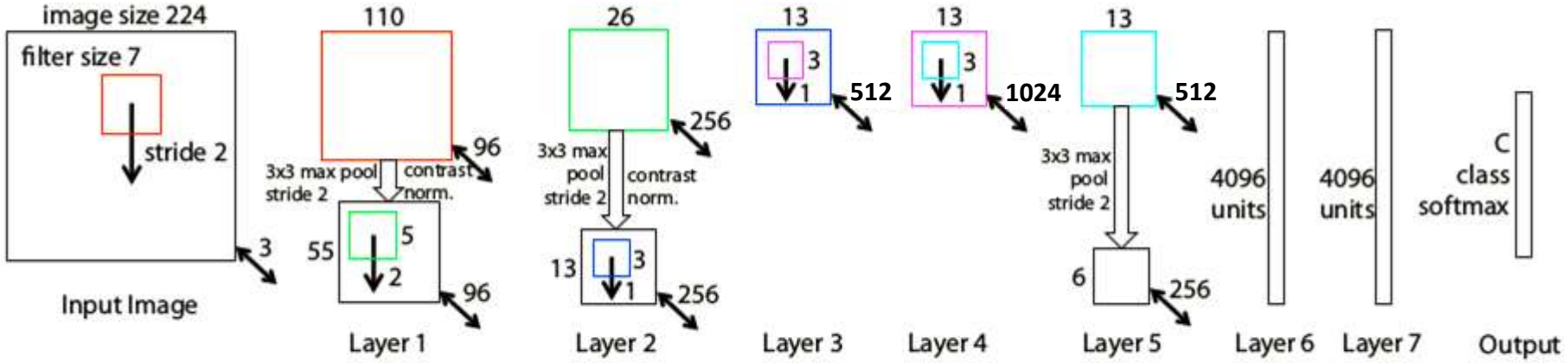
Eight ILSVRC-2010 test images and the five labels considered most probable by our model. The correct label is written under each image, and the probability assigned to the correct label is also shown with a red bar (if it happens to be in the top 5).



Five ILSVRC-2010 test images in the first column. The remaining columns show the six training images that produce feature vectors in the last hidden layer with the smallest Euclidean distance from the feature vector for the test image.

Krizhevsky, A., Sutskever, I. and Hinton, G. E. "ImageNet Classification with Deep Convolutional Neural Networks" NIPS 2012: Neural Information Processing Systems, Lake Tahoe, Nevada

ZFNet



ZF Net Architecture

- Zeiler and Fergus 2013
- Same as Alexnet except:
  - 7x7 input-layer filters with stride 2
  - 3 conv layers are 512, 1024, 512
  - Error went down from 15.4% → 14.8%
    - Combining multiple models as before

VGGNet

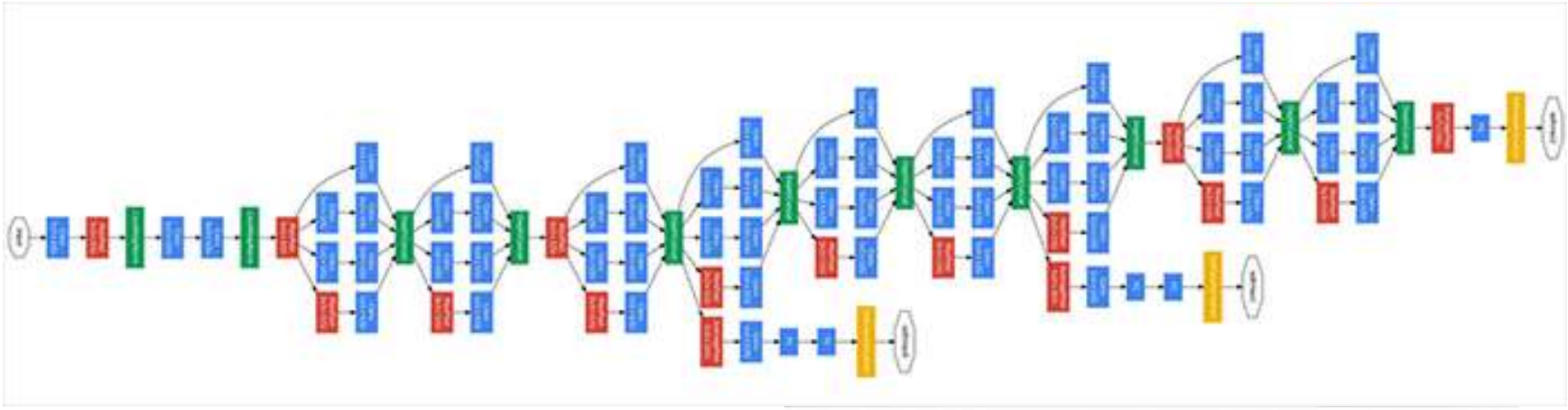
- Simonyan and Zisserman, 2014
- *Only* used 3x3 filters, stride 1, pad 1
- *Only* used 2x2 pooling filters, stride 2

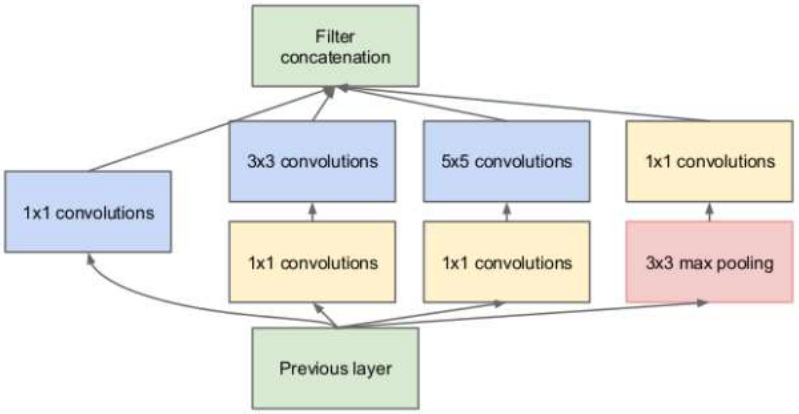
- Tried a large number of architectures.
- Finally obtained 7.3% top-5 error using 13 conv layers and 3 FC layers
  - Combining 7 classifiers
  - Subsequent to paper, reduced error to 6.8% using only two classifiers
- Final arch: 64 conv, 64 conv, 64 pool, 128 conv, 128 conv, 128 pool, 256 conv, 256 conv, 256 conv, 256 pool, 512 conv, 512 conv, 512 conv, 512 pool, 512 conv, 512 conv, 512 conv, 512 pool, FC with 4096, 4096, 1000
- ~140 million parameters in all!

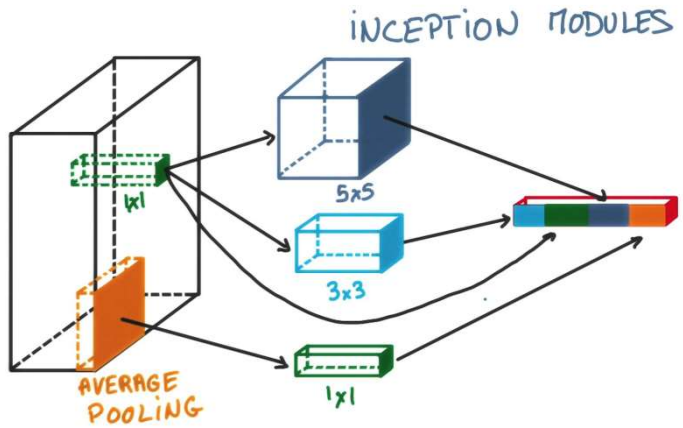
ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Madness!

Googlenet: Inception







- Multiple filter sizes simultaneously
- Details irrelevant; error → 6.7%
  - Using only 5 million parameters, thanks to average pooling

Imagenet

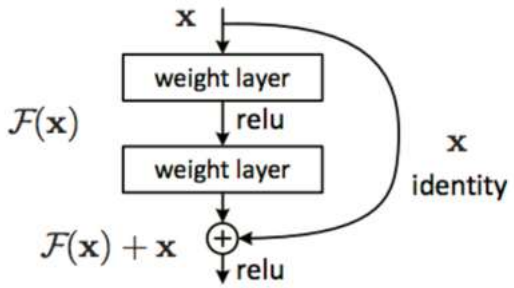
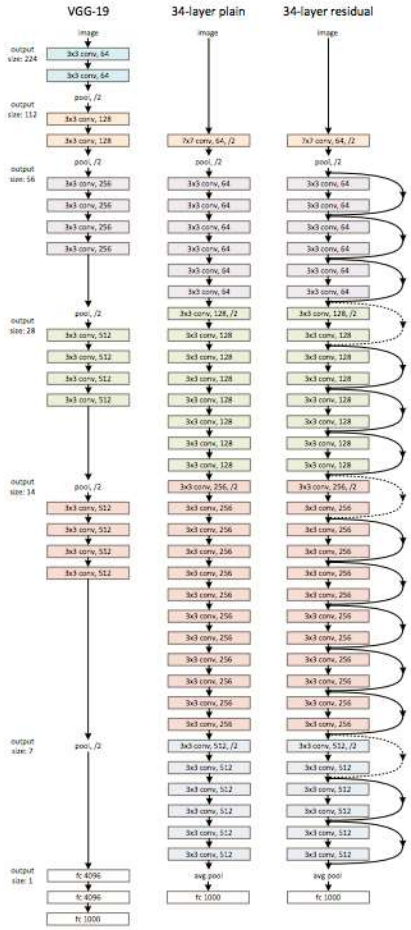


Figure 2. Residual learning: a building block.



- Resnet: 2015
  - Current top-5 error: < 3.5%
  - Over 150 layers, with “skip” connections..

Resnet details for the curious..

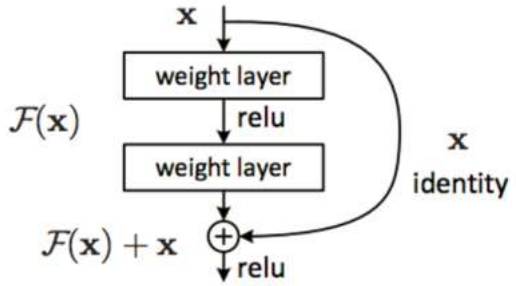
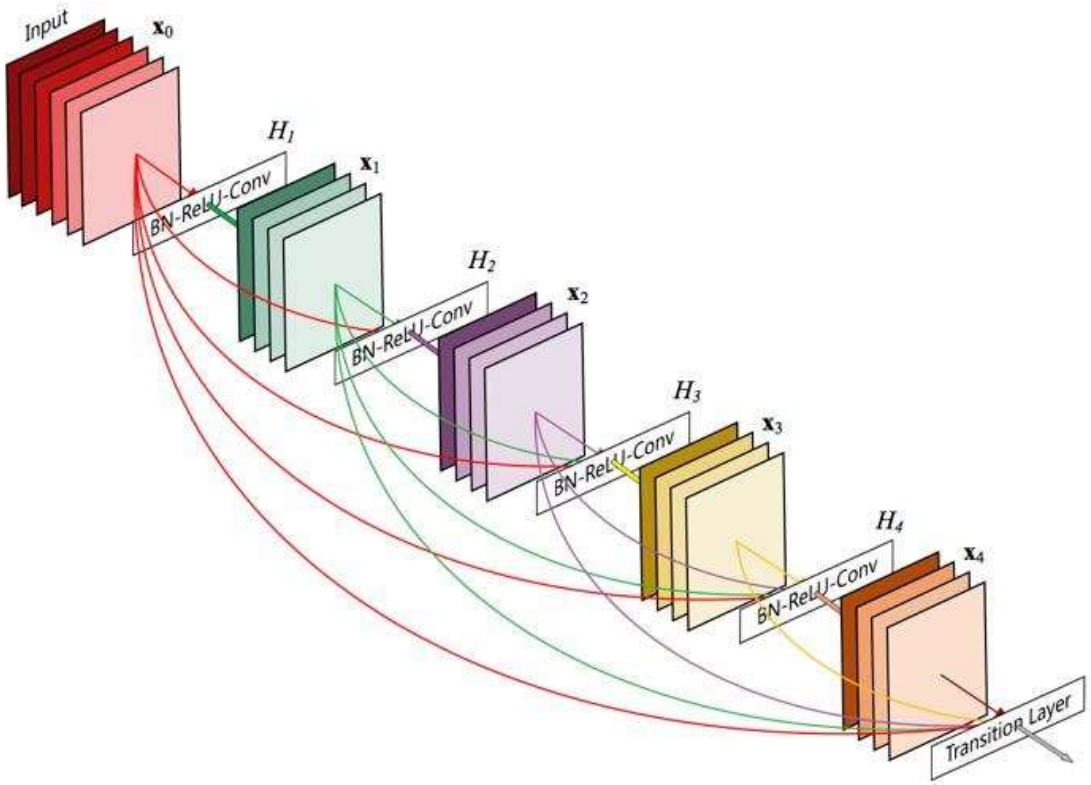


Figure 2. Residual learning: a building block.

- Last layer before addition must have the same number of filters as the input to the module
- Batch normalization after each convolution
- SGD + momentum (0.9)
- Learning rate 0.1, divide by 10 (batch norm lets you use larger learning rate)
- Mini batch 256
- Weight decay 1e-5

Densenet



- All convolutional
- Each layer looks at the union of maps from all previous layers
  - Instead of just the set of maps from the immediately previous layer
- Was state of the art before I went for coffee one day
  - Wasn't when I got back..

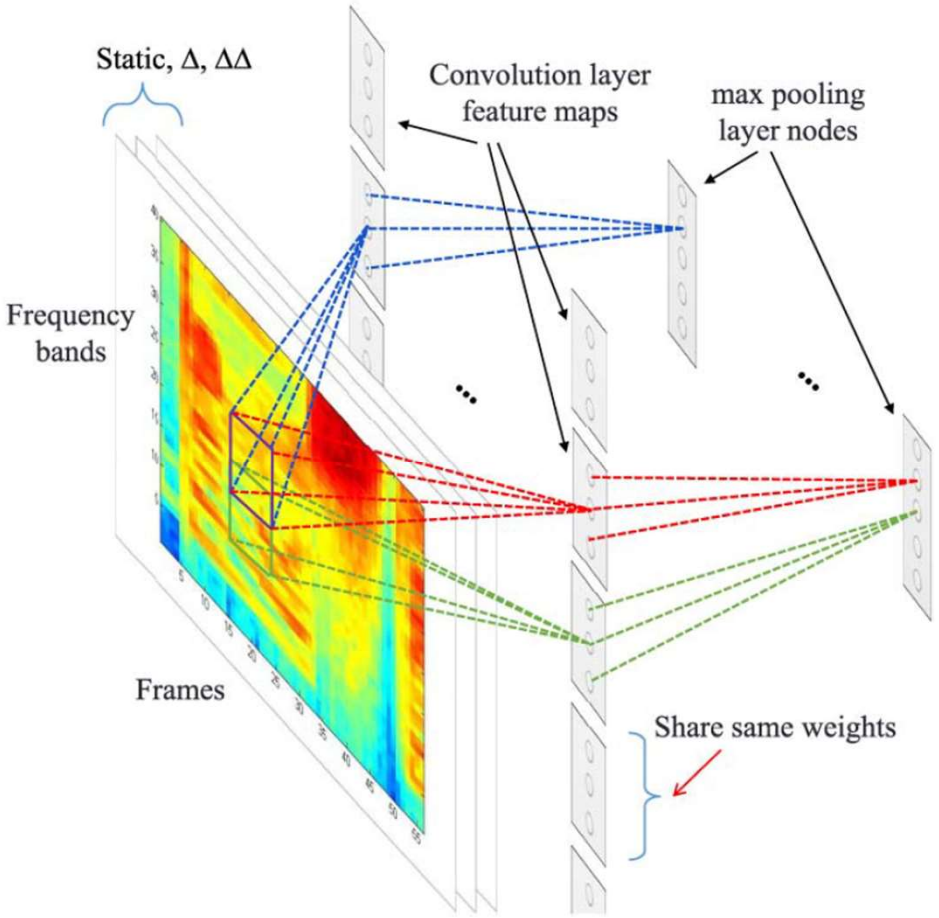
Many many more architectures

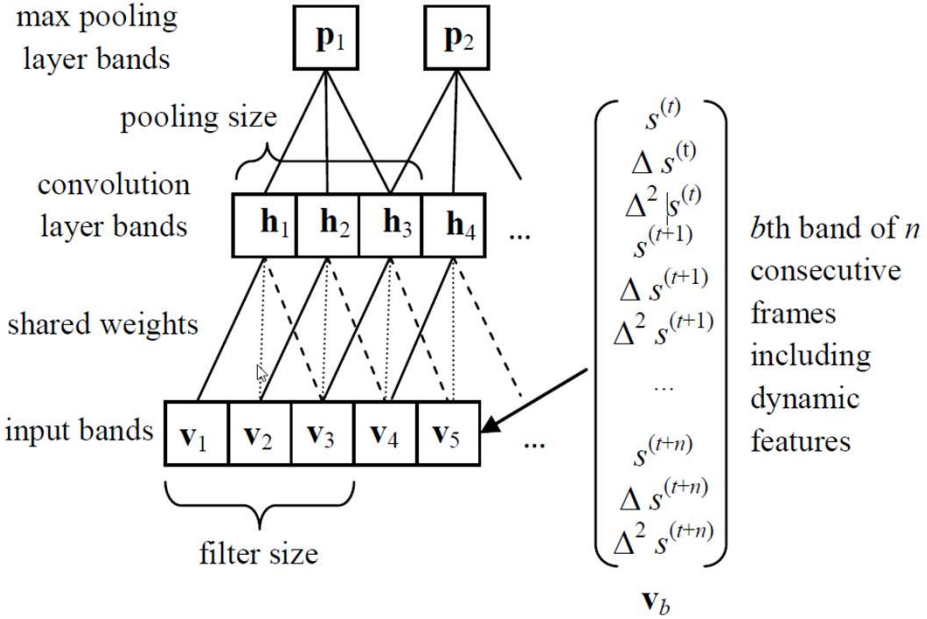
- Daily updates on arxiv..

- Many more applications
  - CNNs for speech recognition
  - CNNs for language processing!
  - More on these later..

CNN for Automatic Speech Recognition

- Convolution over frequencies
- Convolution over time





Deep Networks	Phone Error Rate
DNN (fully connected)	22.3%
CNN-DNN; P=1	21.8%
CNN-DNN; P=12	20.8%
CNN-DNN; P=6 (fixed P, optimal)	20.4%
CNN-DNN; P=6 (add dropout)	19.9%
CNN-DNN; P=1:m (HP, m=12)	**19.3%**
CNN-DNN; above (add dropout)	**18.7%**

Table 1: *TIMIT core test set phone recognition error rate comparisons.*

CNN-Recap

- Neural network with specialized connectivity structure
- Feed-forward:
  - Convolve input
  - Non-linearity (rectified linear)
  - Pooling (local max)
- Supervised training
- Train convolutional filters by back-propagating error
- Convolution over time