

# **Deep Neural Networks**

# **Convolutional Networks III**

Bhiksha Raj

Spring 2020

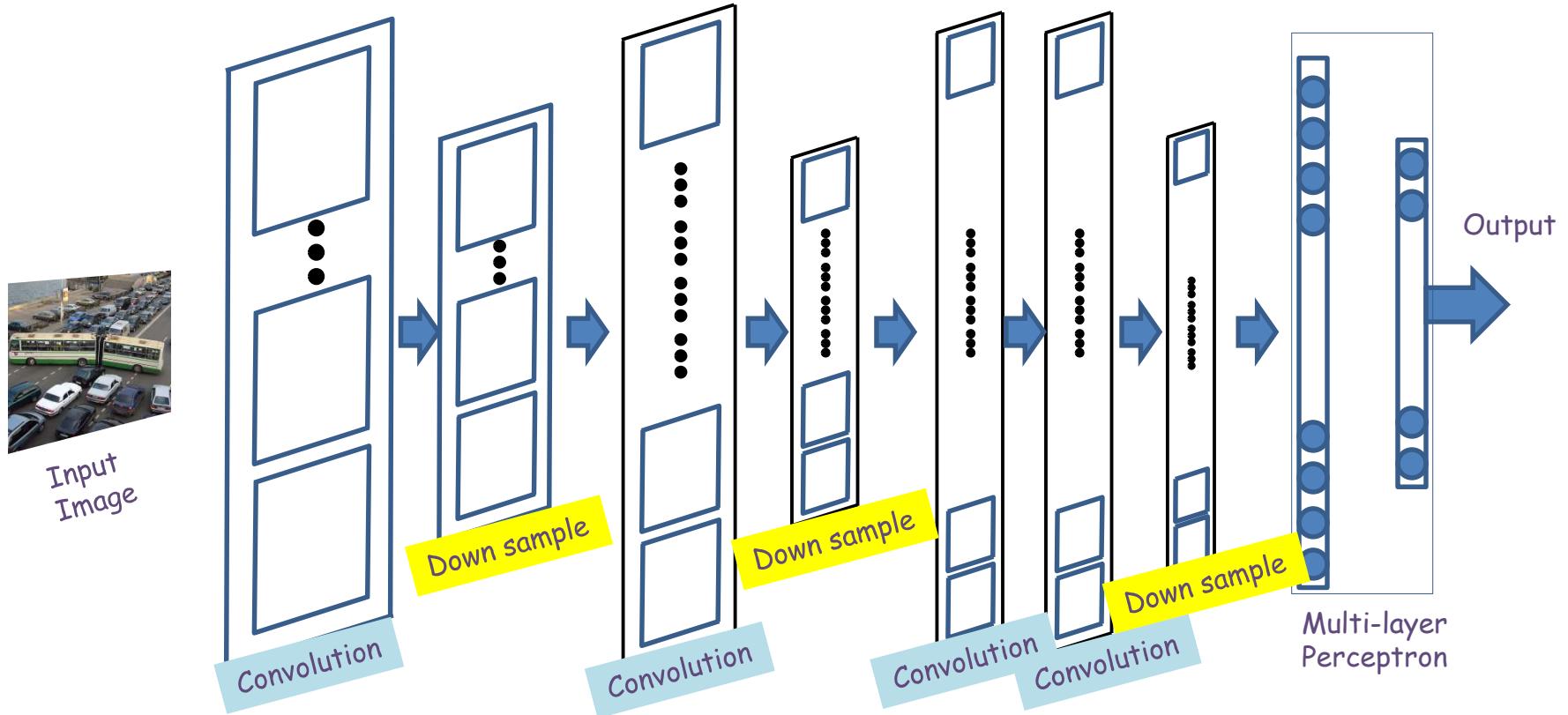
# Outline

- Quick recap
- Back propagation through a CNN
- Modifications: Transposition, scaling, rotation and deformation invariance
- Segmentation and localization
- Some success stories
- Some advanced architectures
  - Resnet
  - Densenet
  - Transformers and self similarity

# Story so far

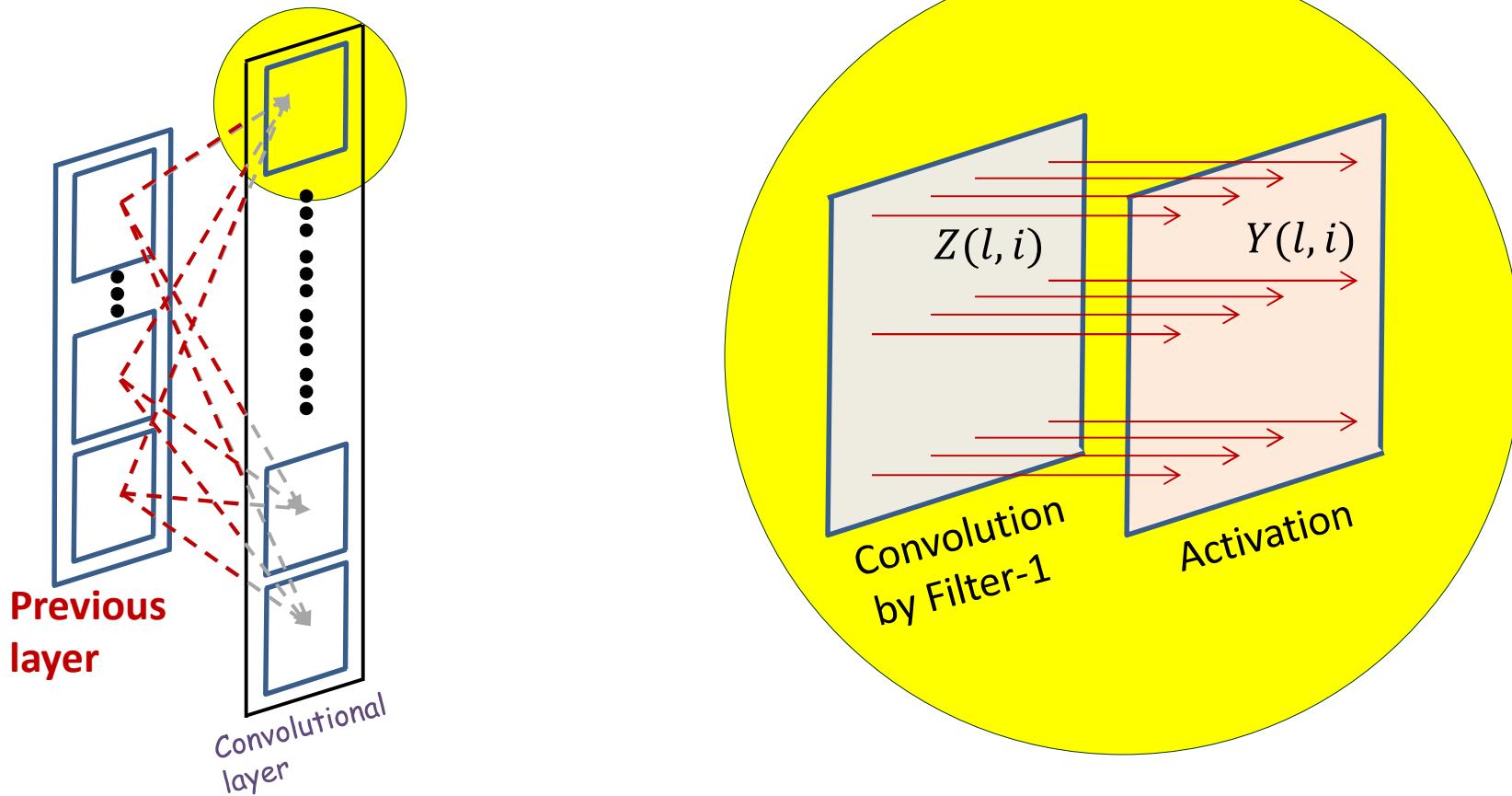
- Pattern classification tasks such as “does this picture contain a cat”, or “does this recording include HELLO” are best performed by scanning for the target pattern
- Scanning an input with a network and combining the outcomes is equivalent to scanning with individual neurons hierarchically
  - First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

# Recap: The general architecture of a convolutional neural network



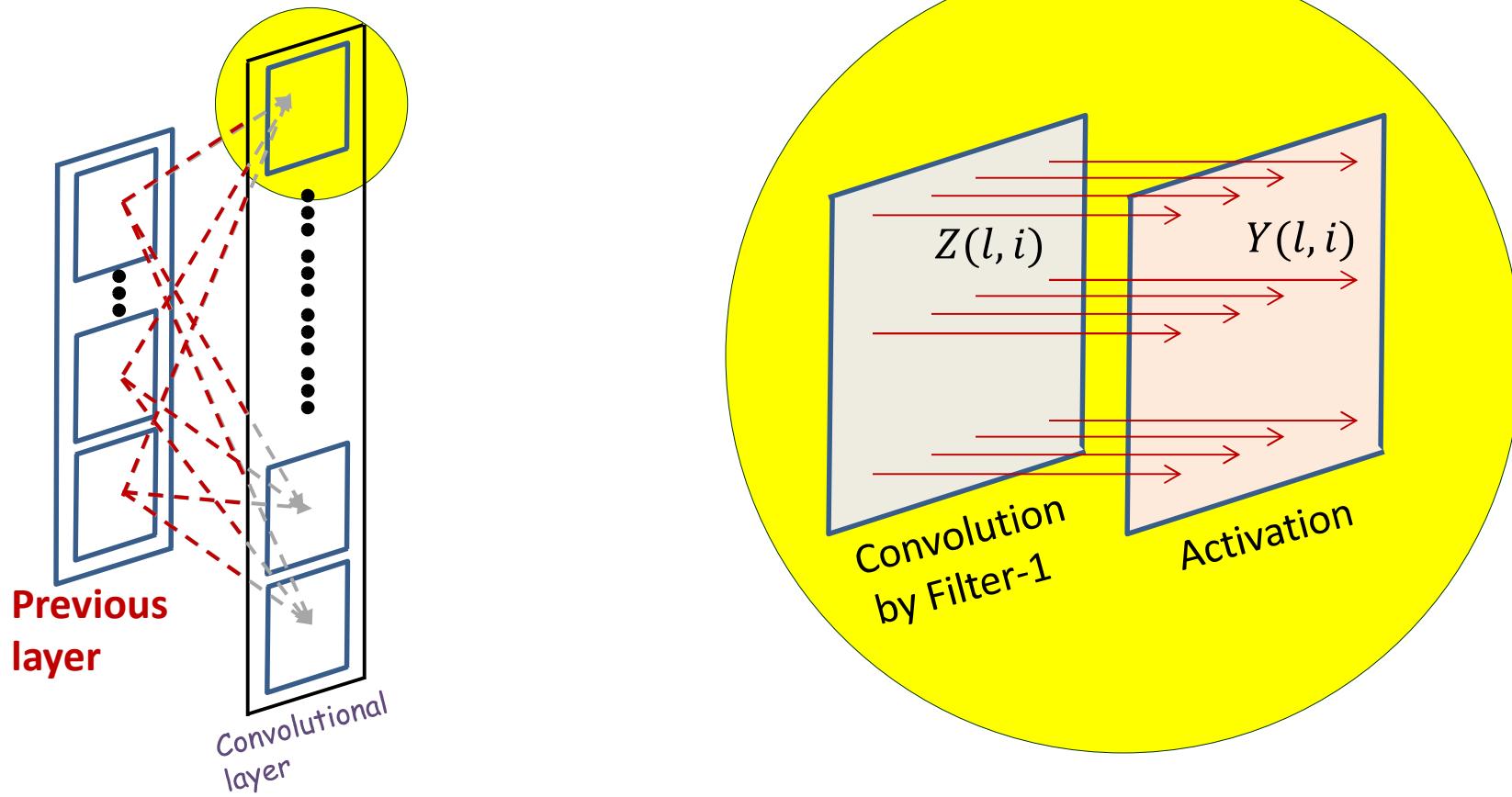
- A convolutional neural network comprises of “convolutional” and optional “downsampling” layers
- Followed by an MLP with one or more layers

# Recap: A convolutional layer



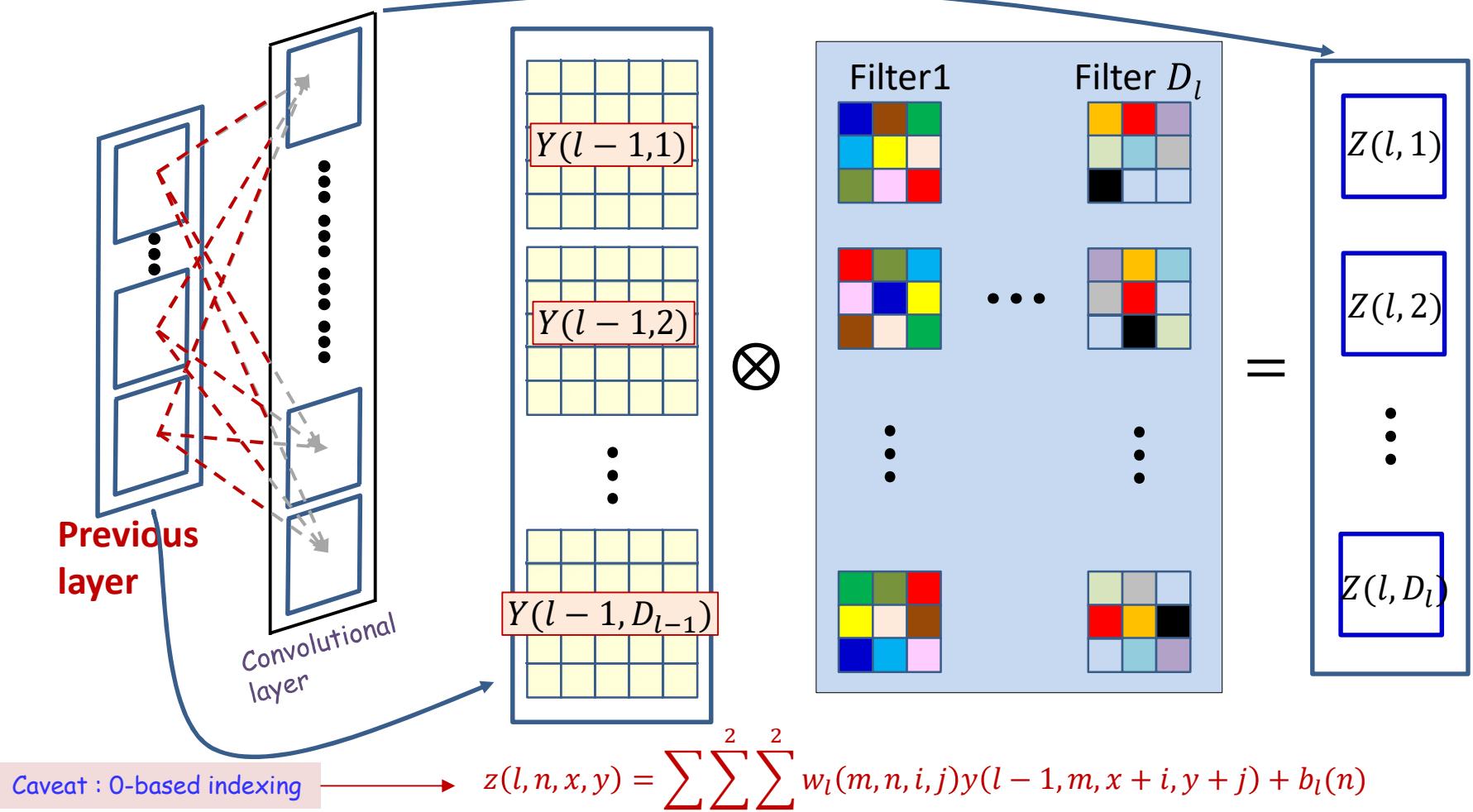
- The computation of each output map has two stages
  - Computing an *affine* map, by *convolution* over maps in the previous layer
    - Each affine map has, associated with it, a **learnable filter**
  - An *activation* that operates *point-wise* on the output of the convolution

# Recap: A convolutional layer



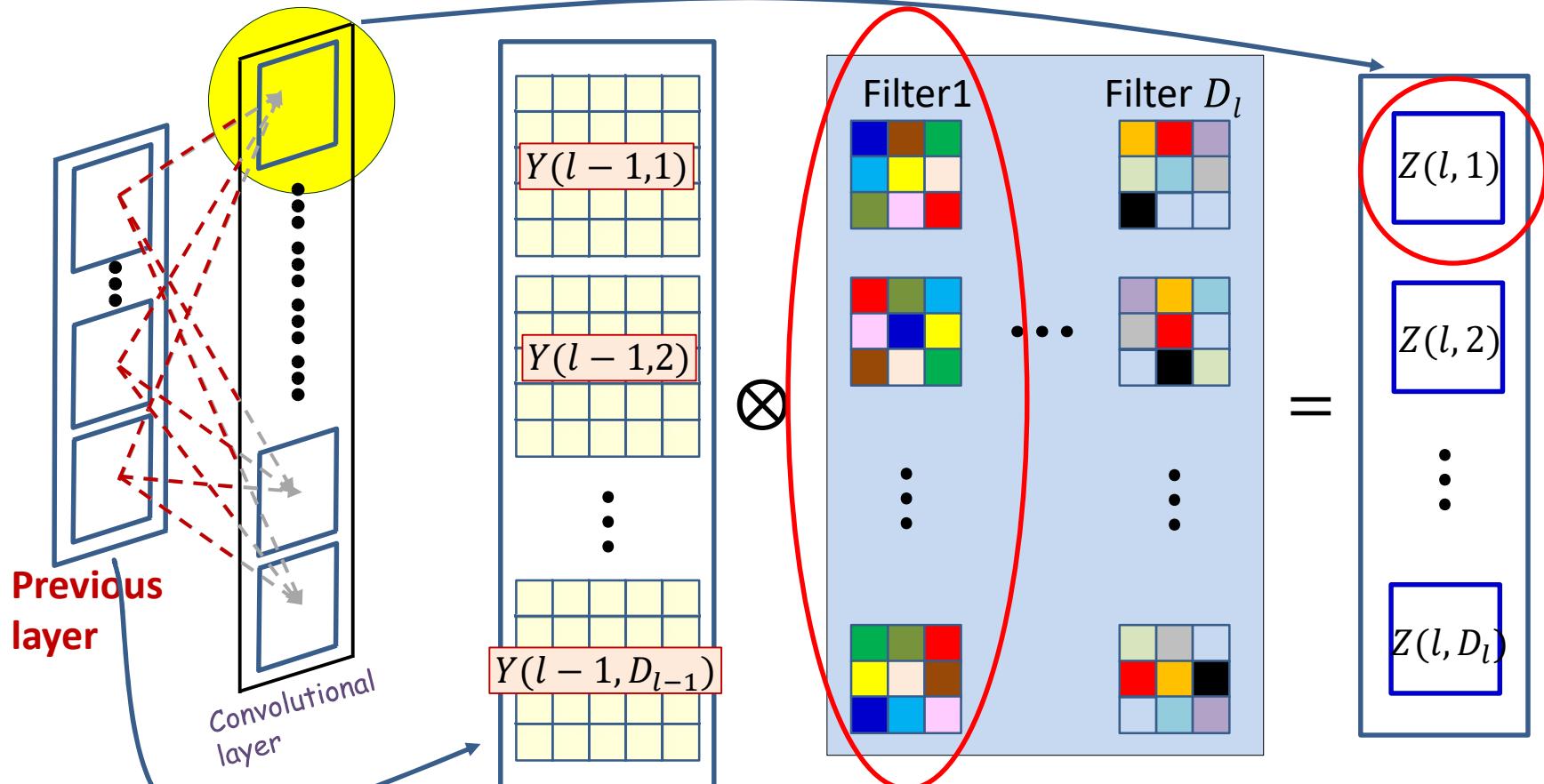
- The computation of each output map has two stages
  - Computing an *affine* map, by *convolution* over maps in the previous layer
    - Each affine map has, associated with it, a **learnable filter**
  - An *activation* that operates *point-wise* on the output of the convolution

# Recap: Convolution



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

# Recap: Convolution

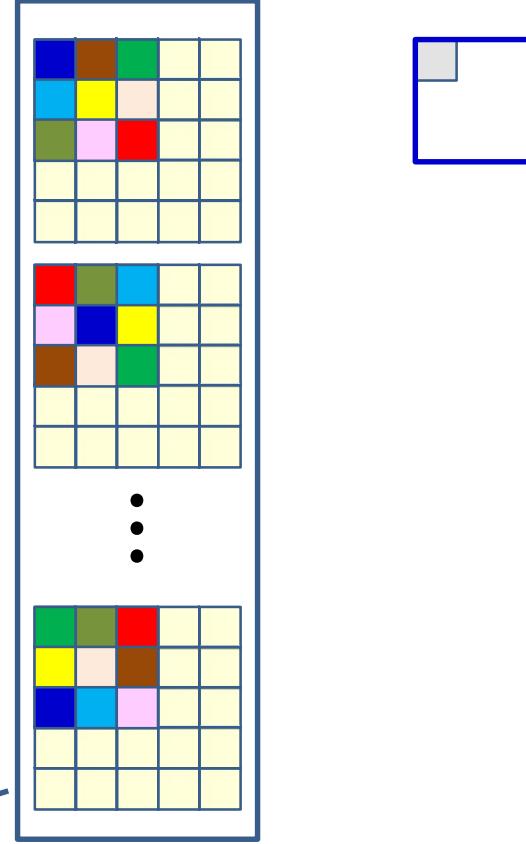
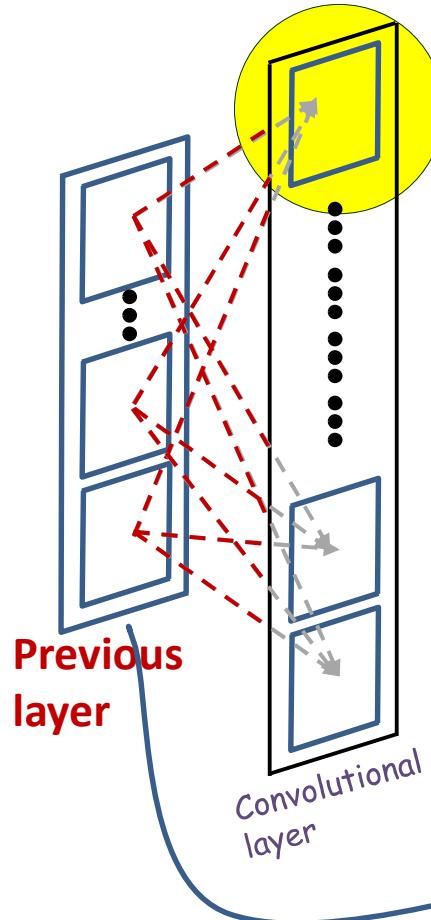


Caveat : 0-based indexing

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

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

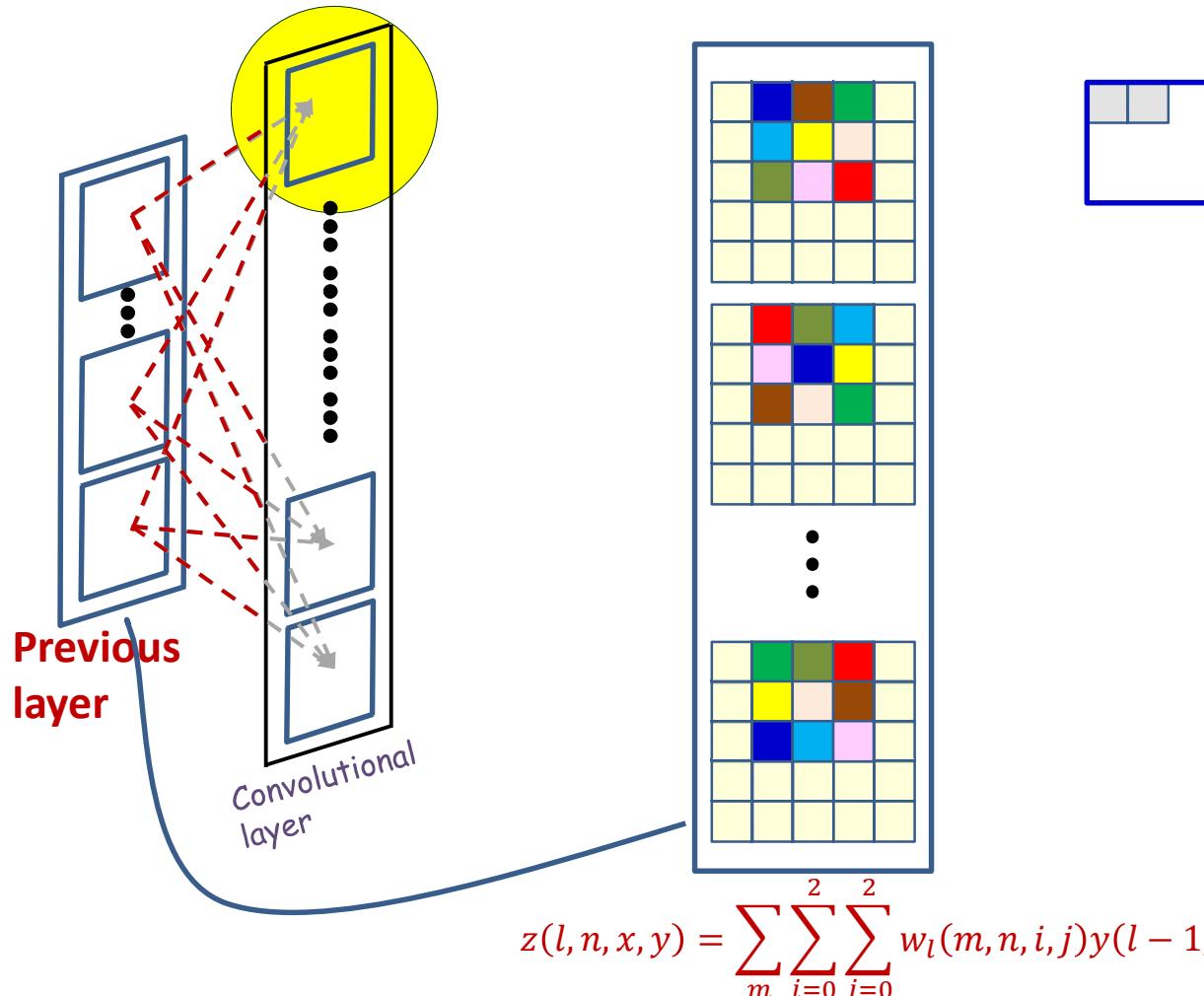
# Recap: Convolution



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

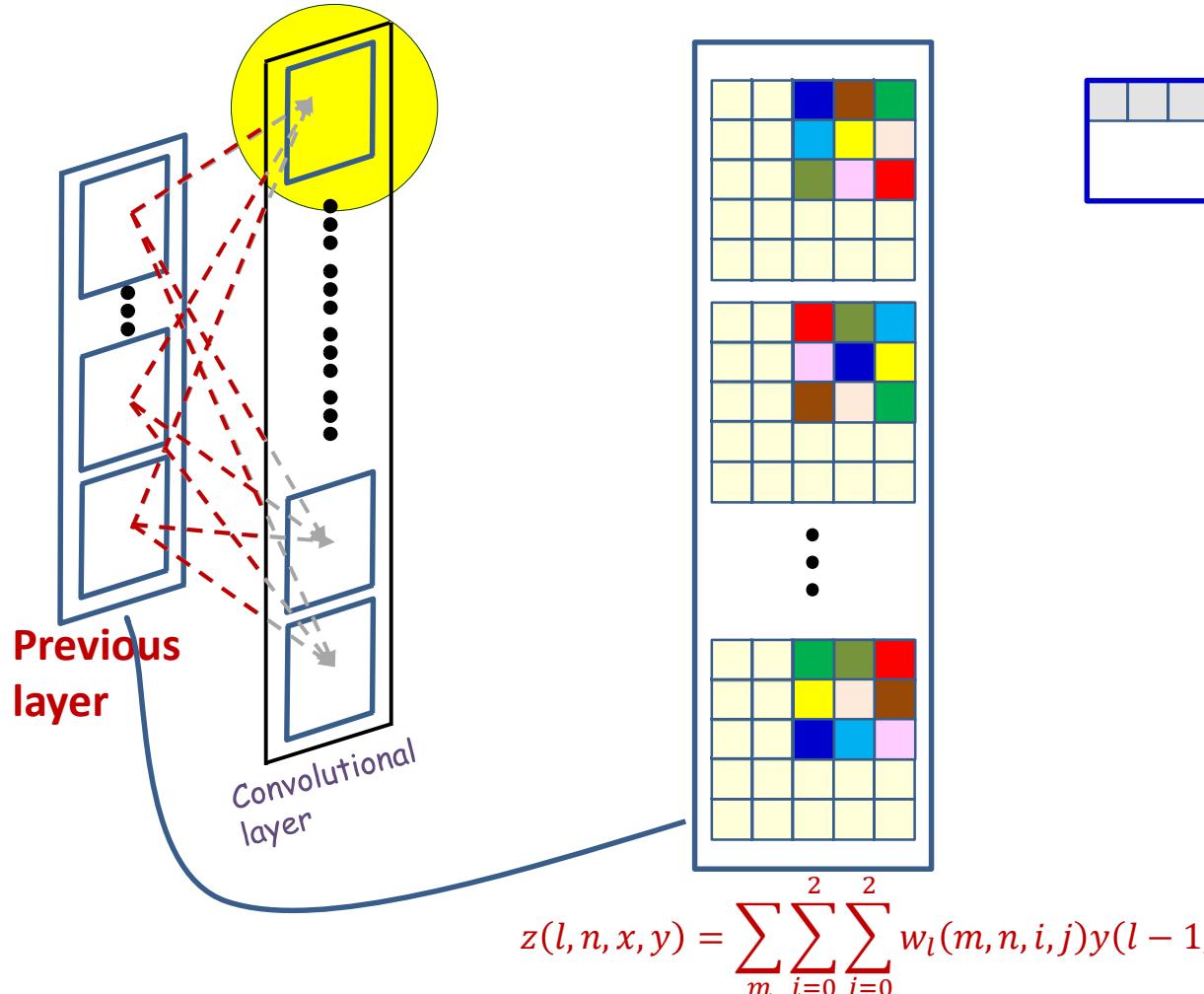
- Each affine output is computed from multiple input maps simultaneously
- There are as many weights (for each output map) as  
*size of the filter x no. of maps in previous layer*

# Recap: Convolution



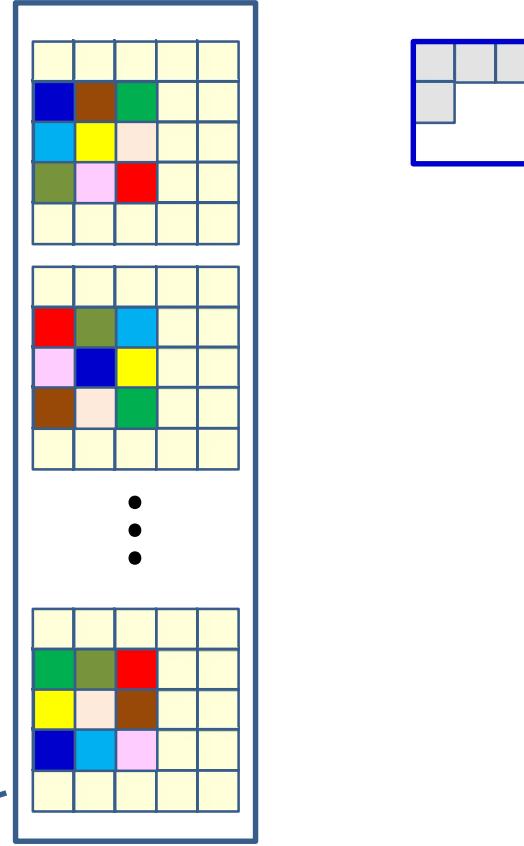
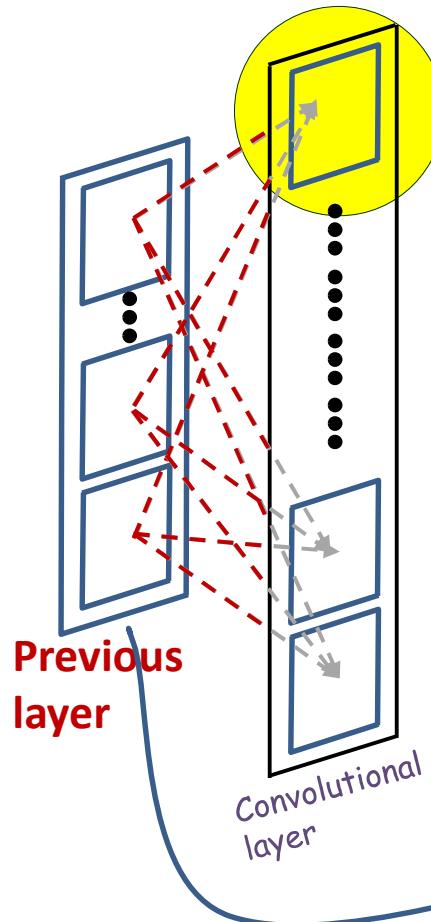
- Each affine output is computed from multiple input maps simultaneously
- There are as many weights (for each output map) as  
*size of the filter x no. of maps in previous layer*

# Recap: Convolution



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

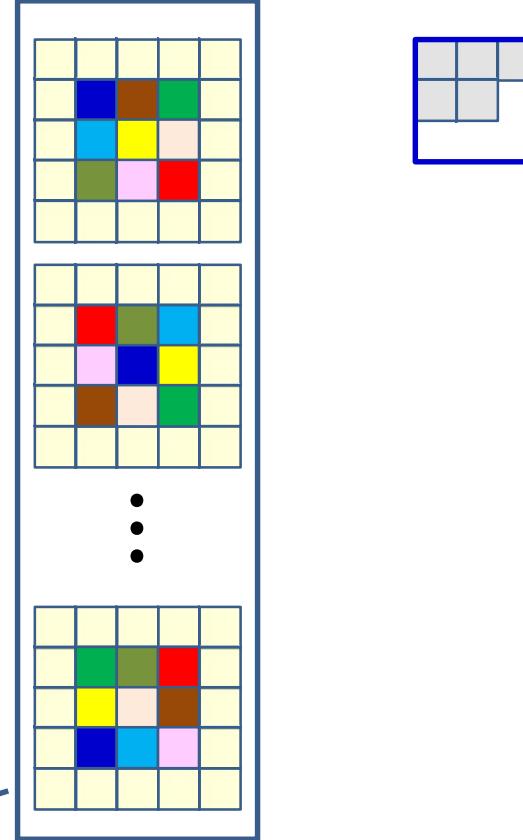
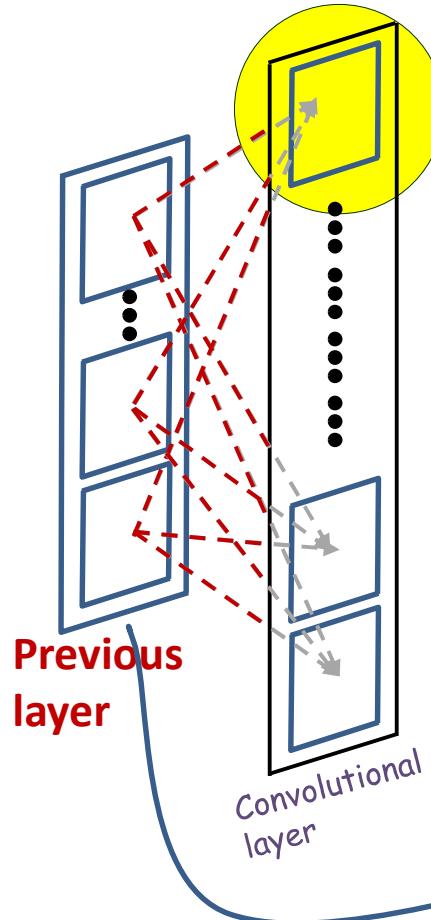
# Recap: Convolution



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

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

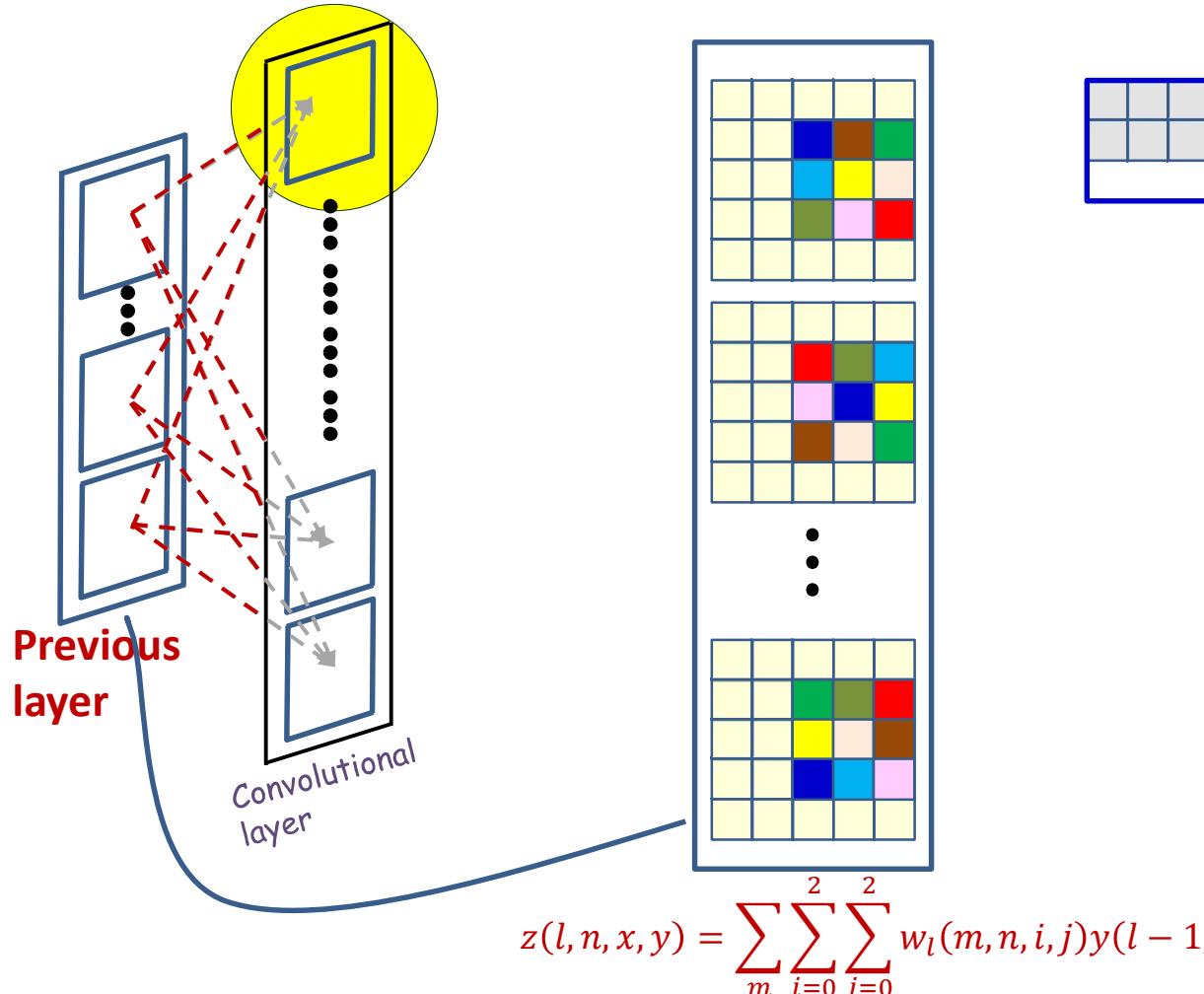
# Recap: Convolution



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

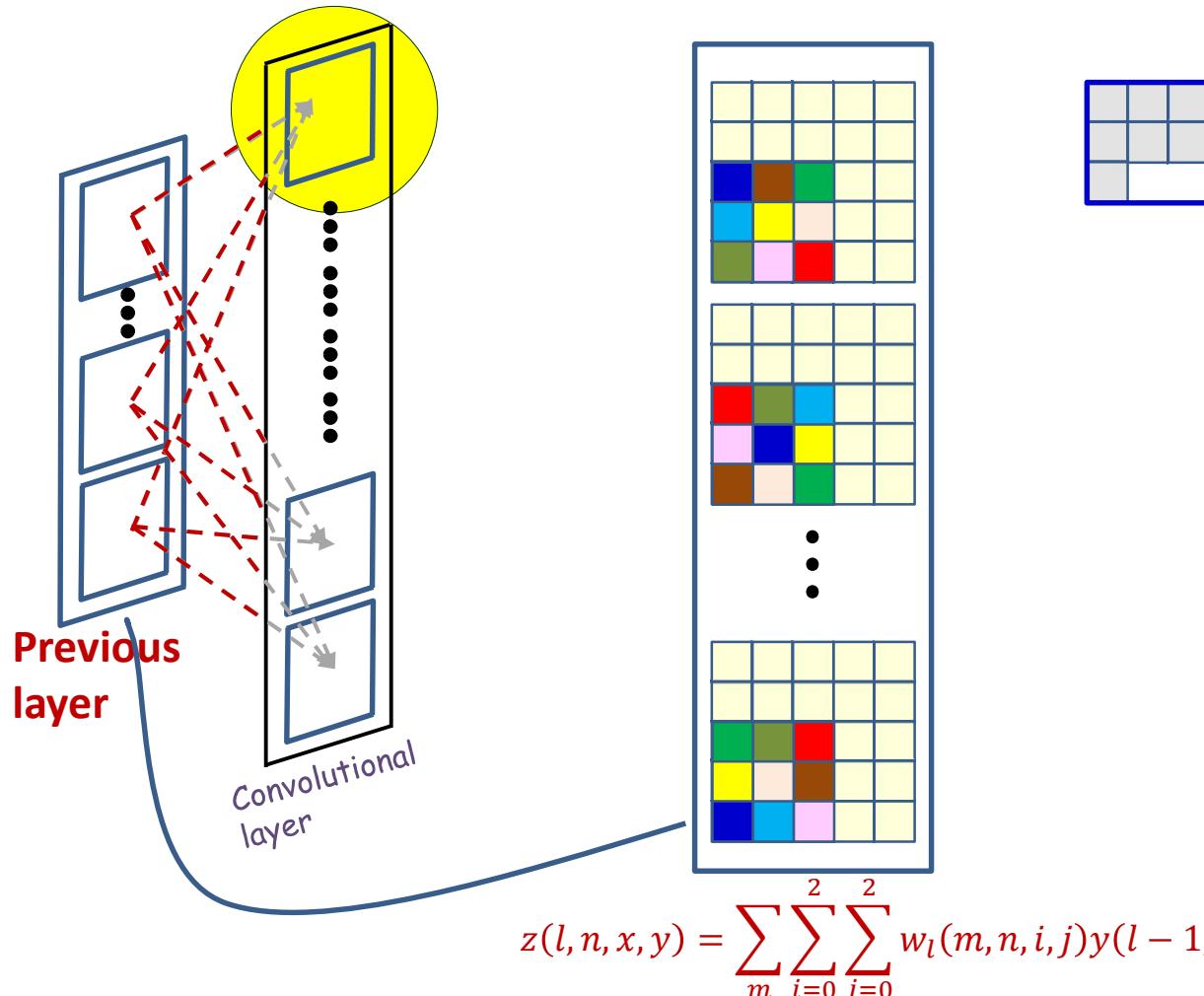
- Each affine output is computed from multiple input maps simultaneously
- There are as many weights (for each output map) as  
*size of the filter x no. of maps in previous layer*

# Recap: Convolution



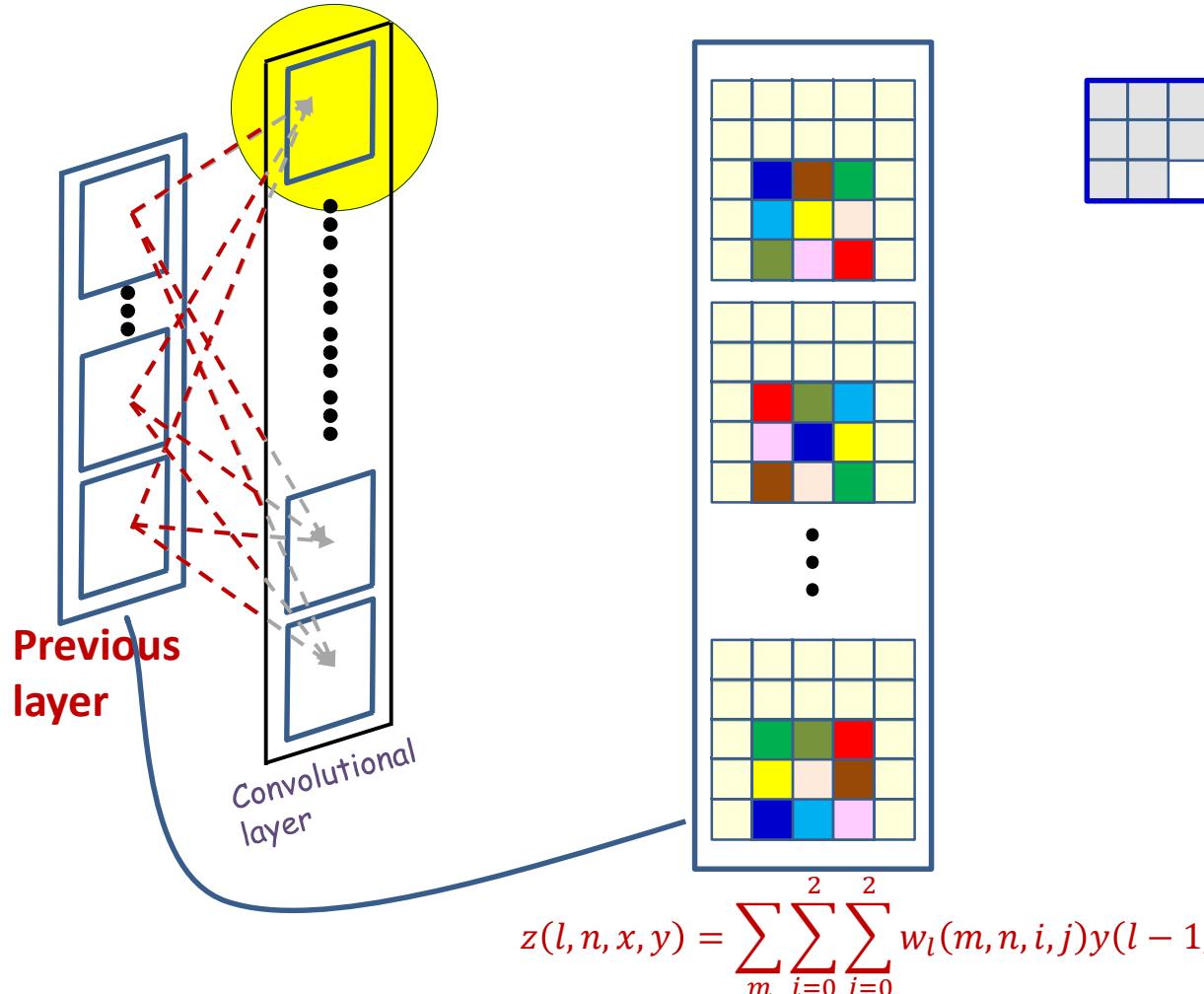
- Each affine output is computed from multiple input maps simultaneously
- There are as many weights (for each output map) as  
*size of the filter x no. of maps in previous layer*

# Recap: Convolution



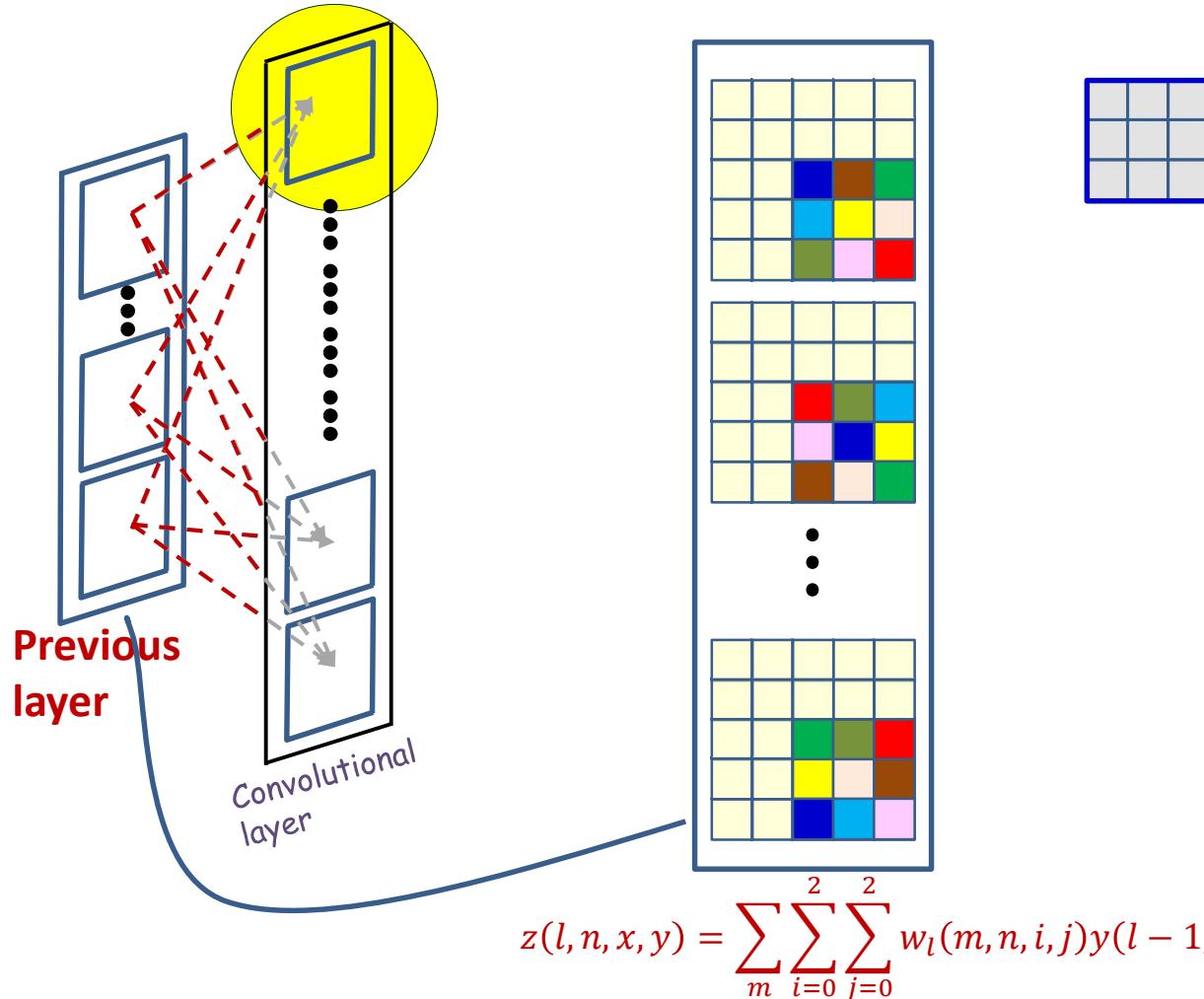
- Each affine output is computed from multiple input maps simultaneously
- There are as many weights (for each output map) as  
*size of the filter x no. of maps in previous layer*

# Recap: Convolution



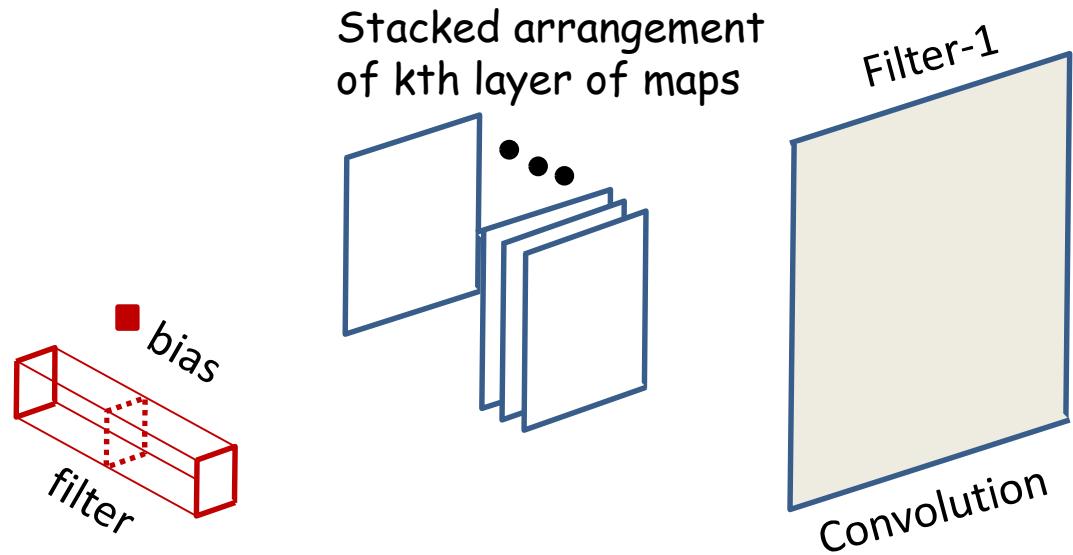
- Each affine output is computed from multiple input maps simultaneously
- There are as many weights (for each output map) as  
*size of the filter x no. of maps in previous layer*

# Recap: Convolution



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

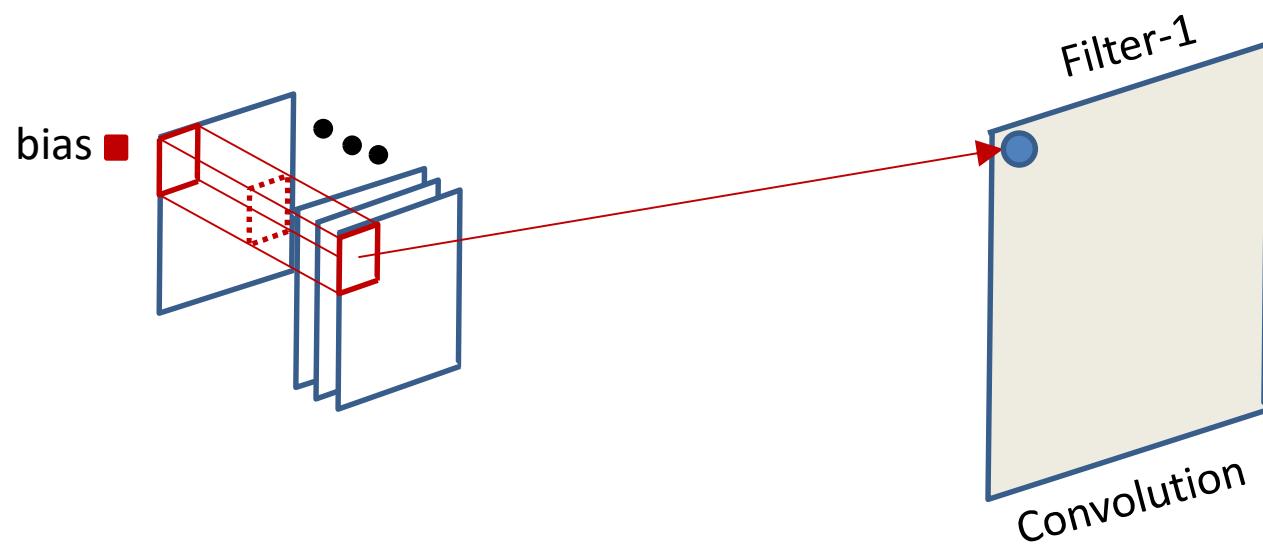
# Recap: A cube visualization



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

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

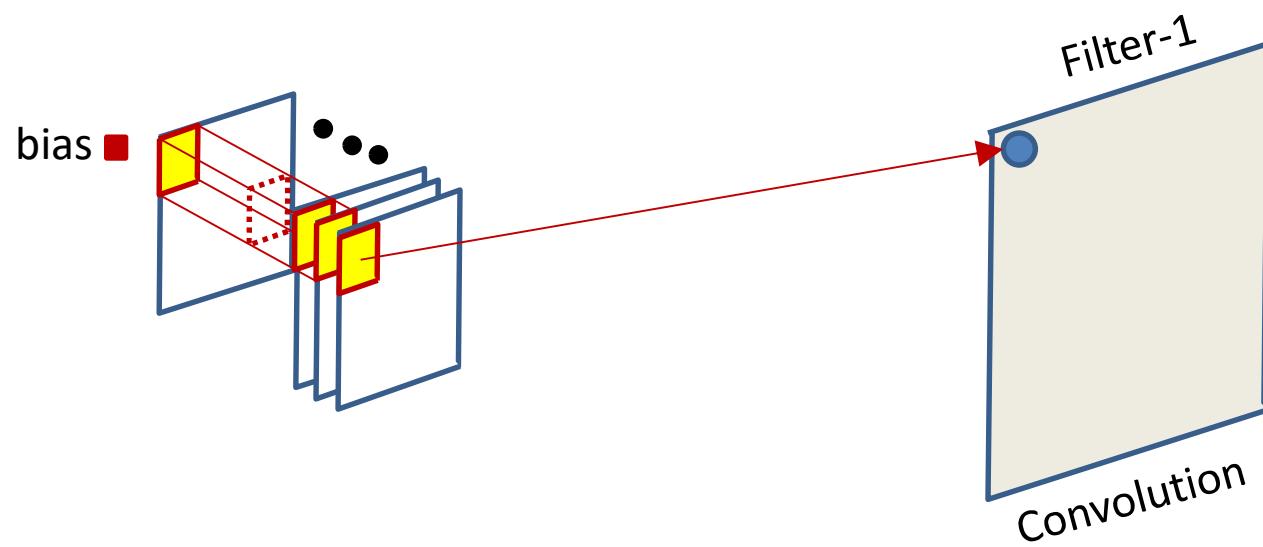
# Recap: A cube visualization



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

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

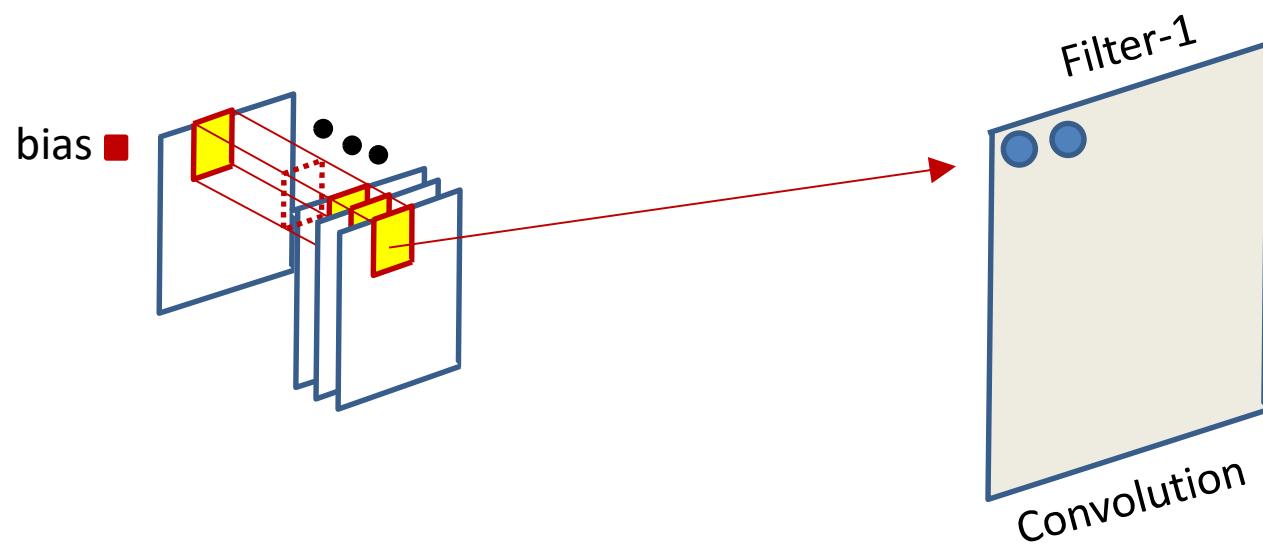
# Recap: A cube visualization



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

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

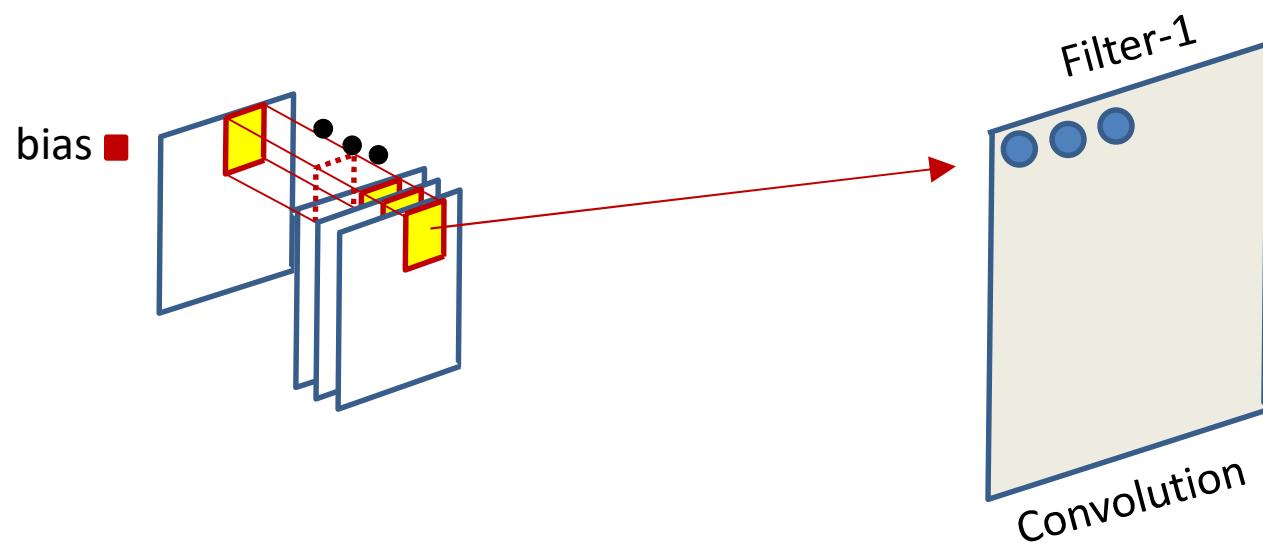
# Recap: A cube visualization



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

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

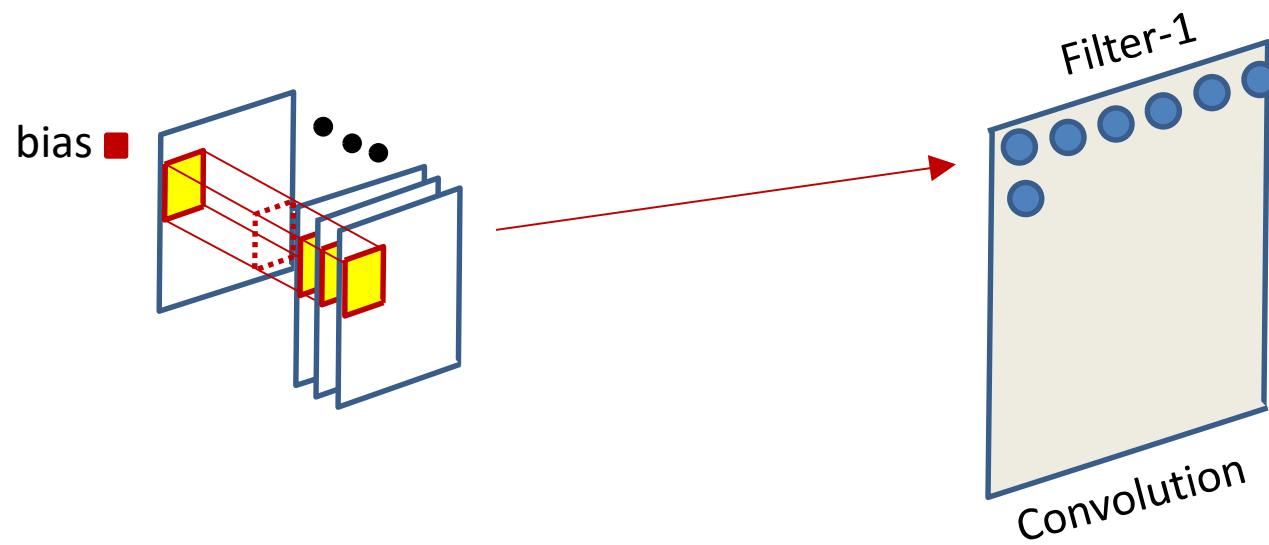
# Recap: A cube visualization



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

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

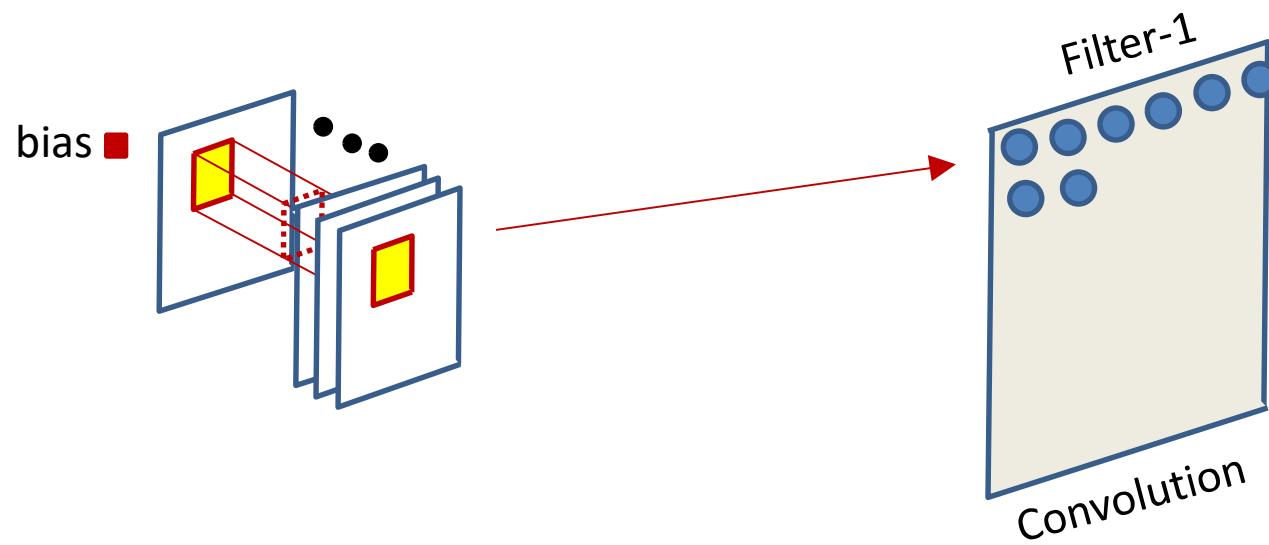
# Recap: A cube visualization



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

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

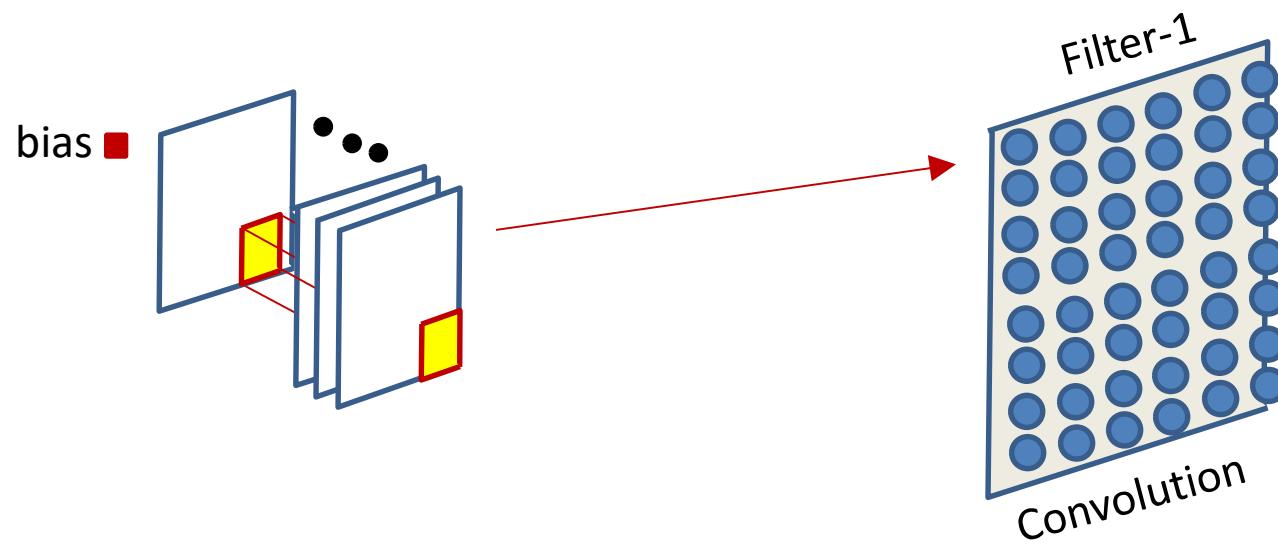
# Recap: A cube visualization



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

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

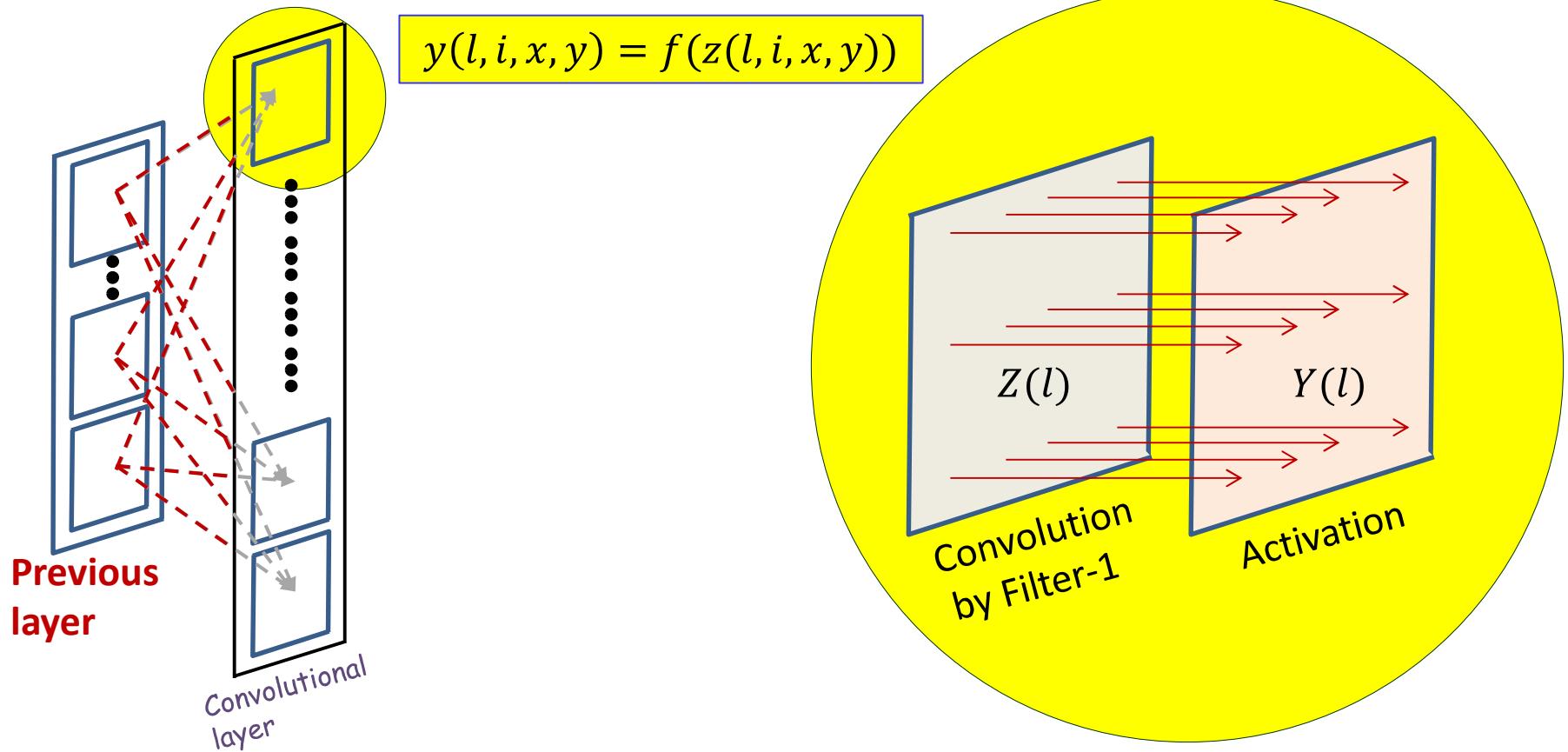
# Recap: A cube visualization



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

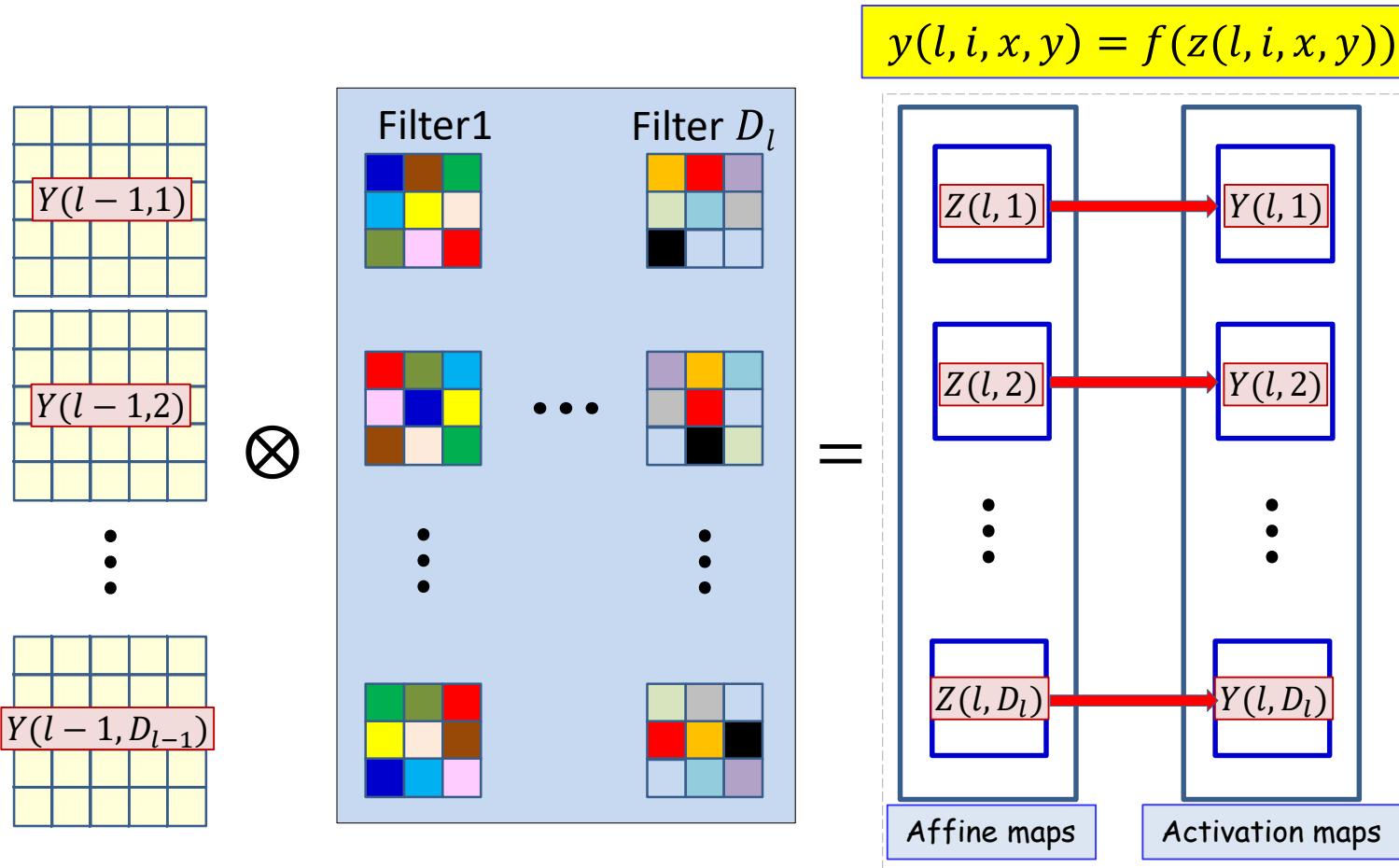
- The computation of the convolutive map at any location *sums* the convolutive outputs *at all planes*

# Recap: A convolutional layer



- The computation of each output map has two stages
  - Computing an *affine* map, 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

# Convolution layer: A more explicit illustration



- Input maps  $Y(l - 1, *)$  are convolved with several filters to generate the affine maps  $Z(l, *)$ 
  - Each filter consists of a set of square patterns of weights, with one set for each map in  $Y(l - 1, *)$
  - We get one affine map per filter
- A *point-wise* activation function  $f(z)$  is applied to each map in  $Z(l, *)$  to produce the activation maps  $Y(l, *)$

# Pseudocode: Vector notation

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

```
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 + b(l, j) #tensor prod.
                Y(l, j, x, y) = activation(z(l, j, x, y))
Y = softmax( {Y(L, :, :, :)} )
```

Pseudocode has 1-based indexing

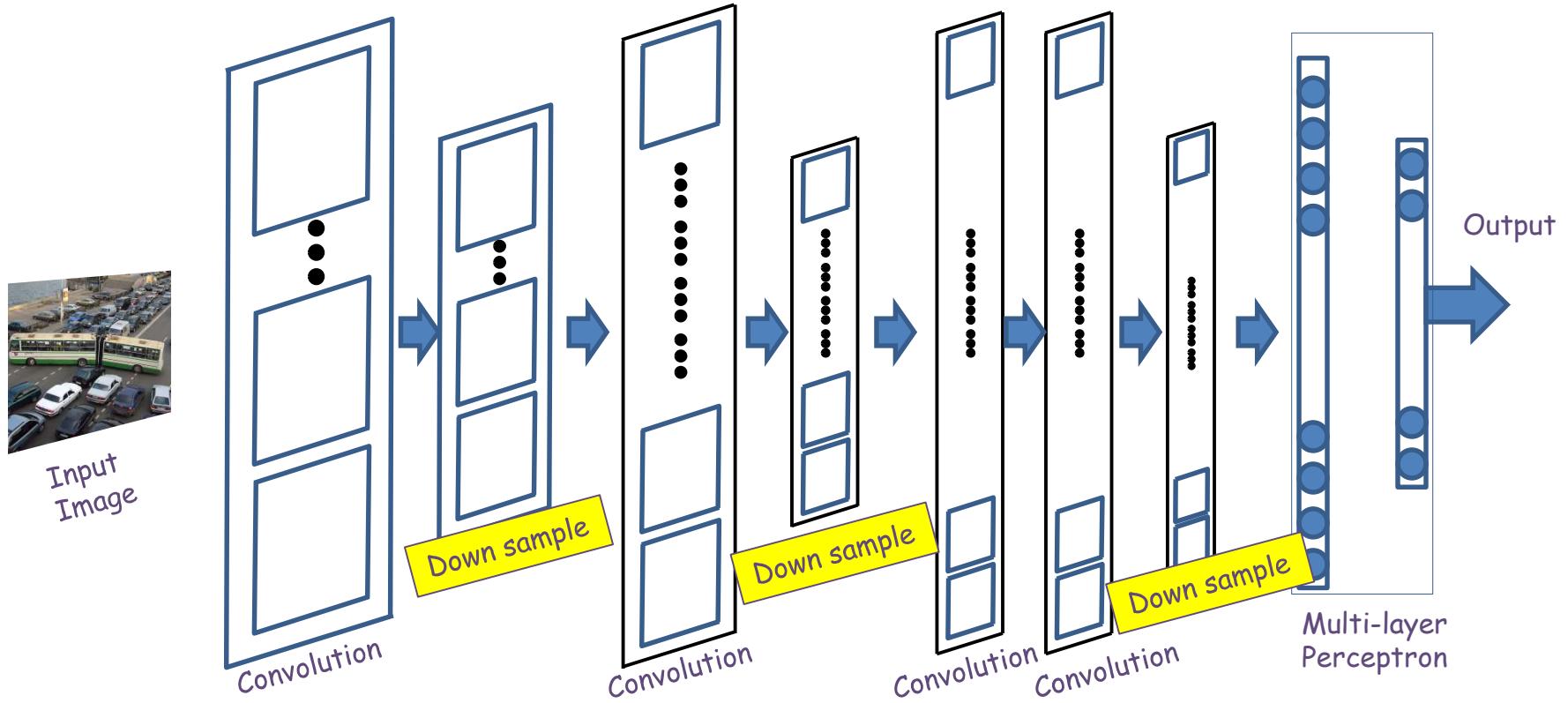
# Pseudocode: Vector notation

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

```
Y(0) = Image
for l = 1:L  # layers operate on vector at (x,y)
    m = 1
    for x = 1:stride:W_{l-1}-K_l+1
        n = 1
        for y = 1:stride: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 + b(l, j) #tensor prod.
                Y(l, j, m, n) = activation(z(l, j, m, n))
            n++
        m++
```

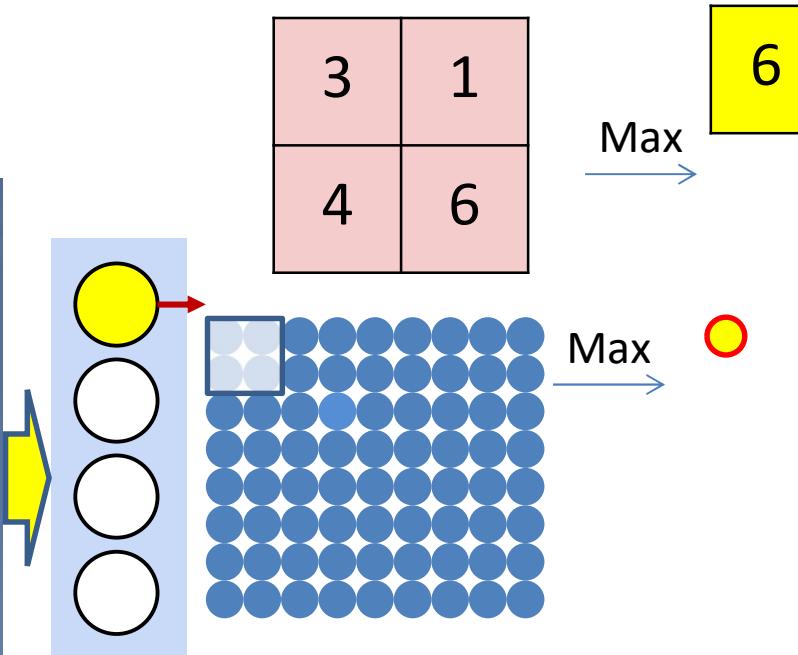
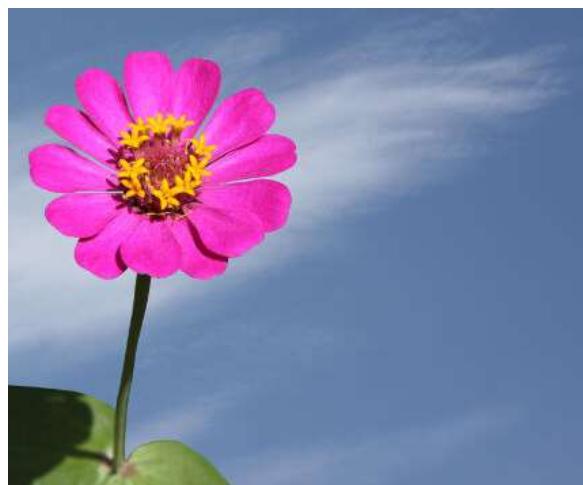
Y = softmax( {Y(L, :, :, :, :)} )

# Downsampling/Pooling



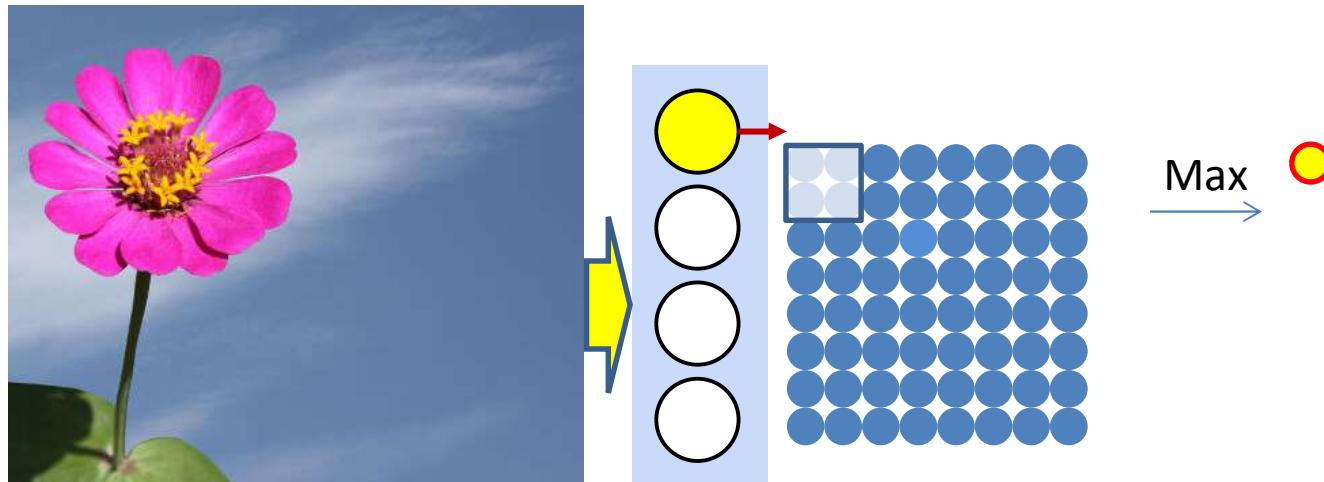
- Convolutional (and activation) layers are followed intermittently by “downsampling” (or “pooling”) layers
  - Often, they alternate with convolution, though this is not necessary

# Recall: Max pooling



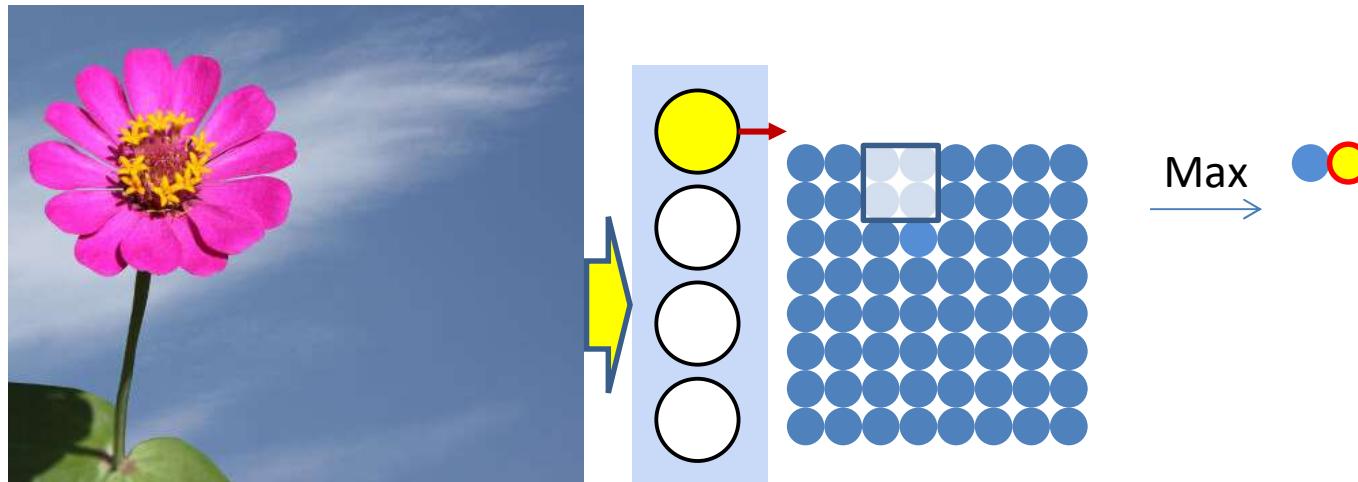
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

# Pooling and downsampling



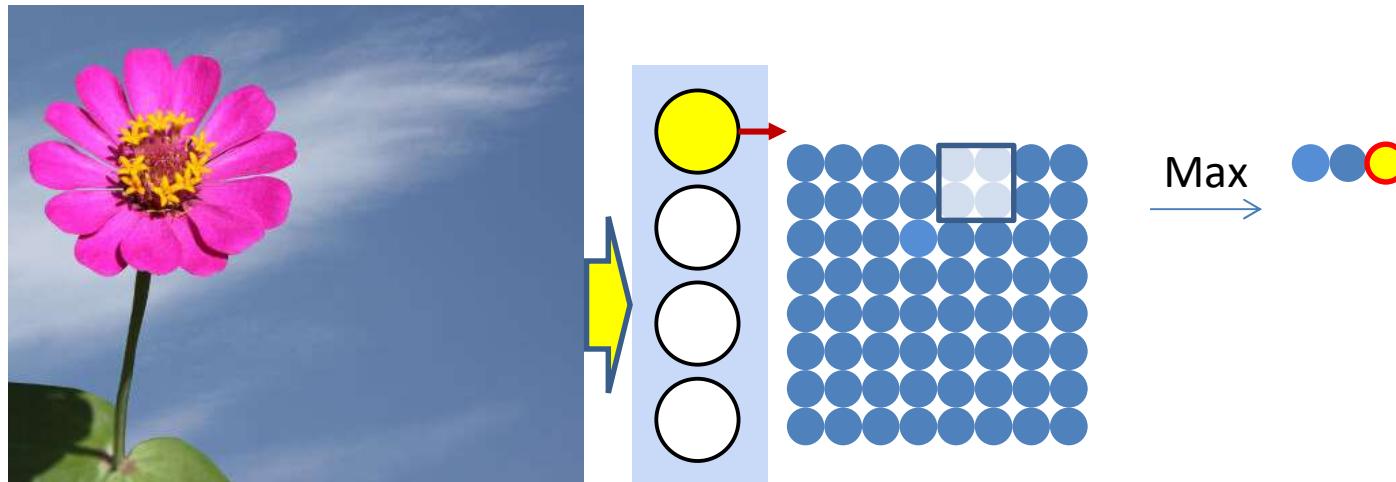
- Pooling is typically performed with strides  $> 1$ 
  - Results in shrinking of the map
  - “Downsampling”

# Pooling and downsampling



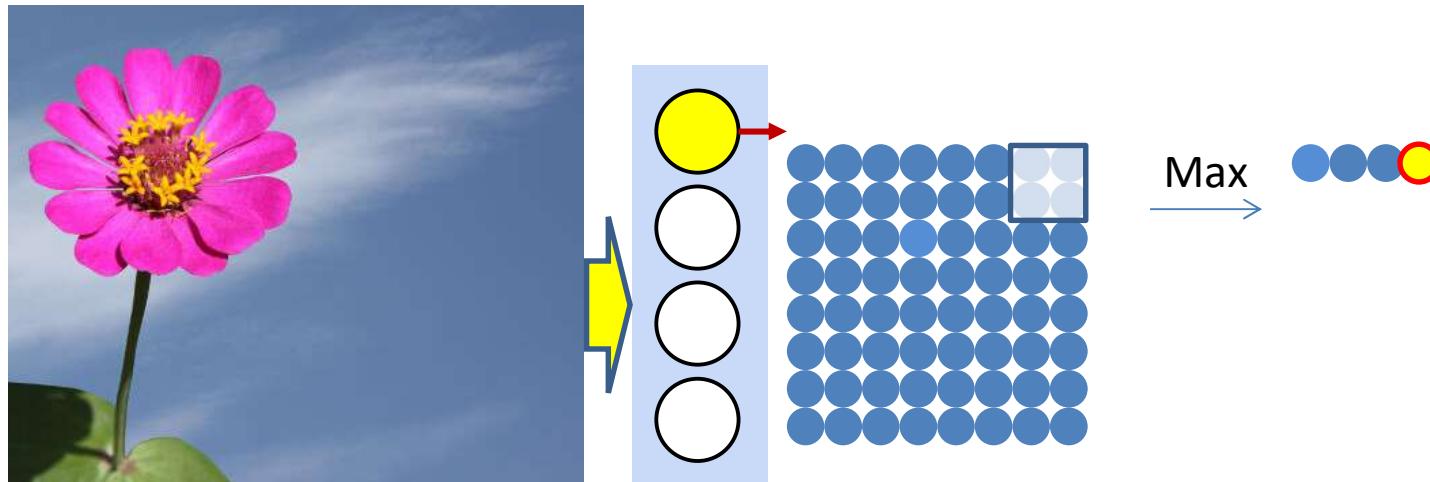
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

# Pooling and downsampling



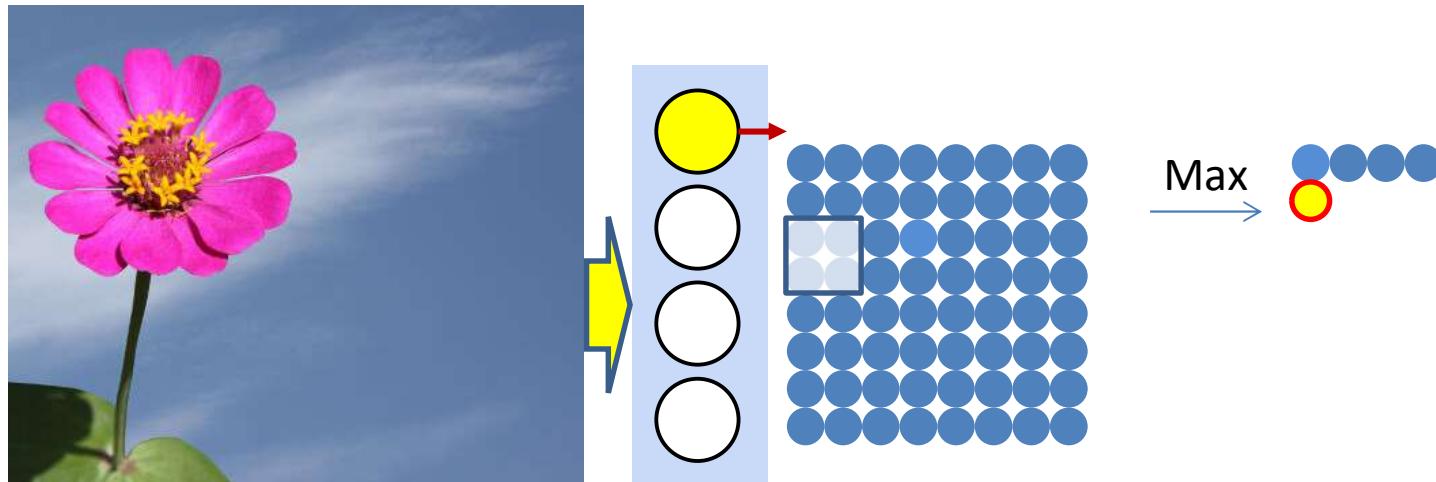
- Pooling is typically performed with strides  $> 1$ 
  - Results in shrinking of the map
  - “Downsampling”

# Pooling and downsampling



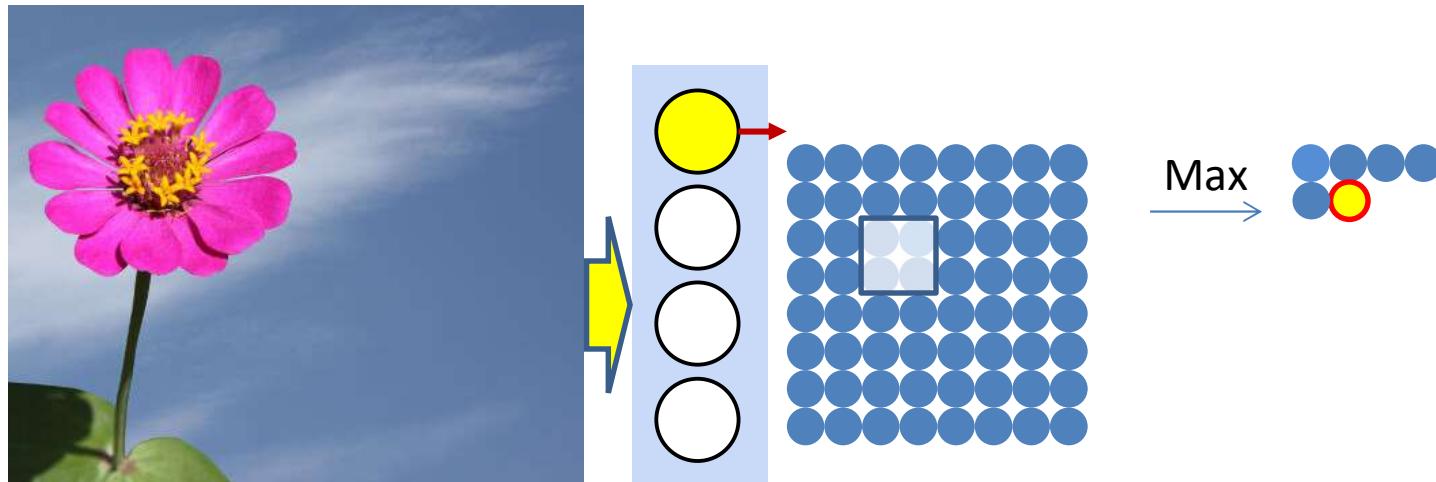
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

# Pooling and downsampling



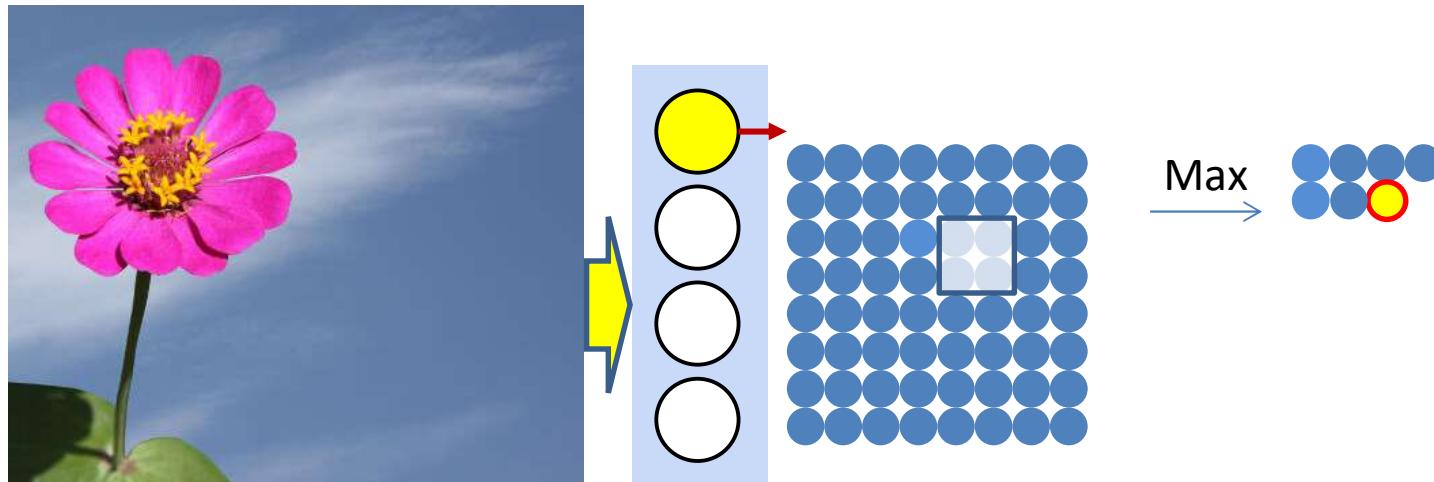
- Pooling is typically performed with strides  $> 1$ 
  - Results in shrinking of the map
  - “Downsampling”

# Pooling and downsampling



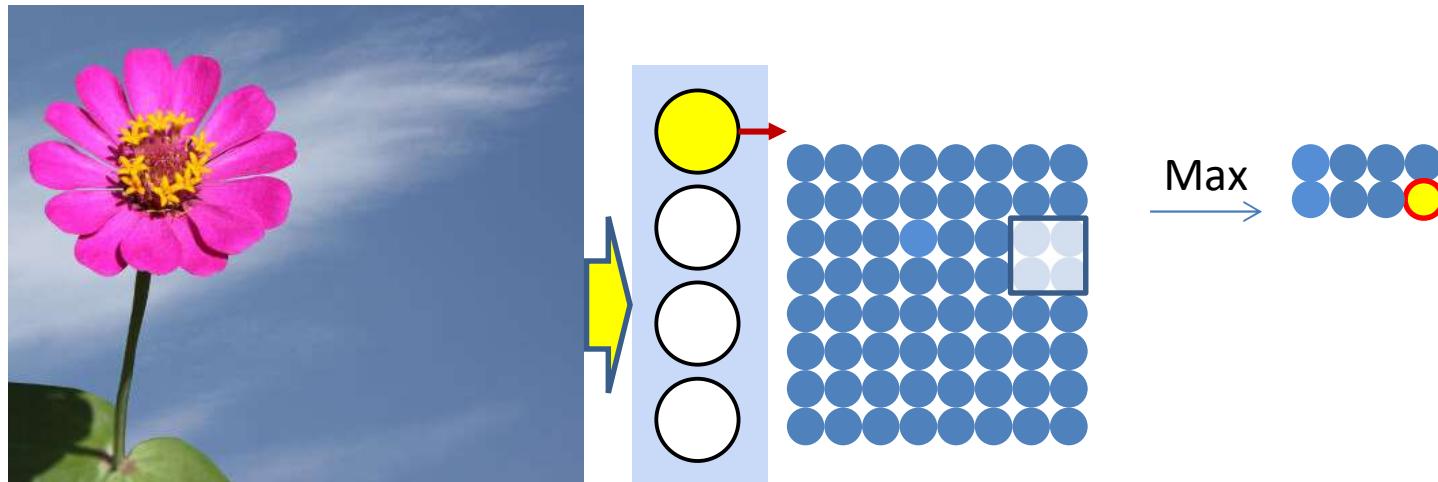
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

# Pooling and downsampling



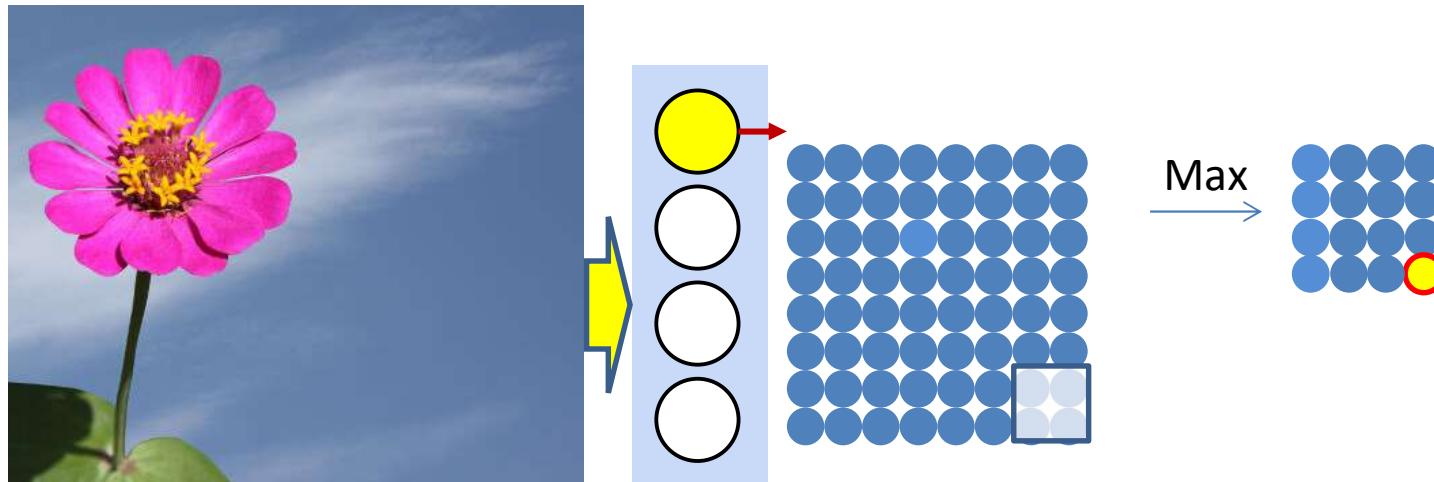
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

# Pooling and downsampling



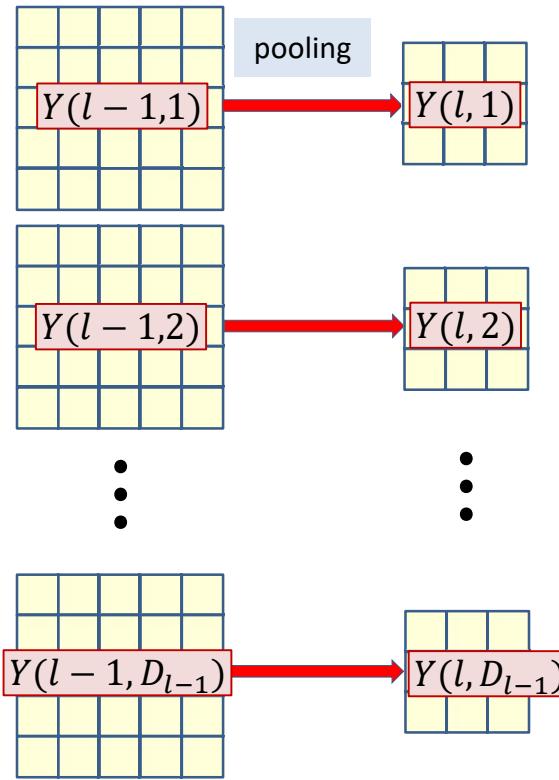
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

# Pooling and downsampling



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

# Recap: Pooling and downsampling layer



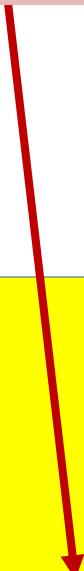
- Input maps  $Y(l - 1, *)$  are operated on individually by pooling operations to produce the pooled maps  $Y(l, *)$ 
  - Pooling is performed with stride  $> 1$  resulting in downsampling
    - Output maps are smaller than input maps

# Recap: Max Pooling layer at layer $l$

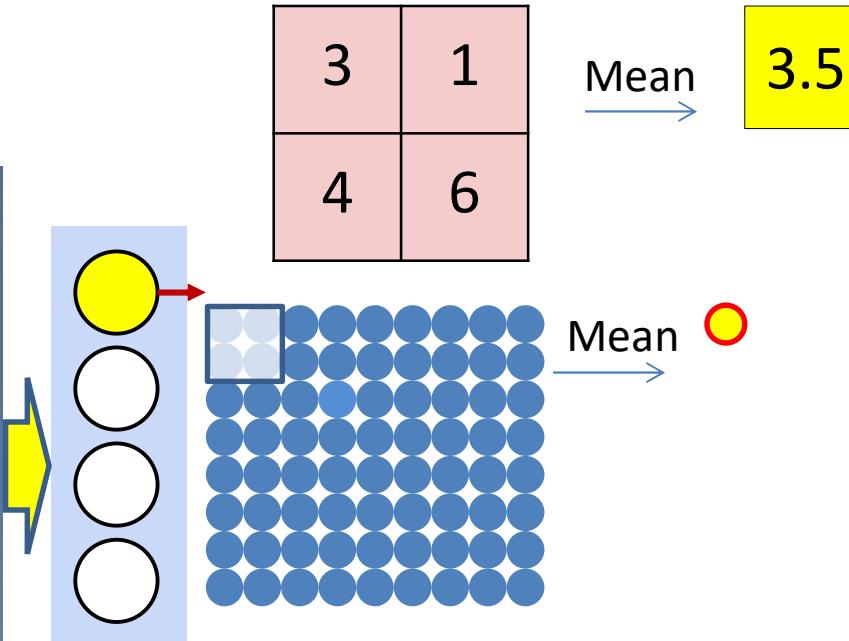
- a) Performed separately for every map ( $j$ ).  
\*) Not combining multiple maps within a single max operation.
- b) Keeping track of location of max

## Max pooling

```
for j = 1:D1
    m = 1
    for x = 1:stride(l):Wl-1-Kl+1
        n = 1
        for y = 1:stride(l):Hl-1-Kl+1
            pidx(l,j,m,n) = maxidx(Y(l-1,j,x:x+Kl-1,y:y+Kl-1))
            u(l,j,m,n) = Y(l-1,j,pidx(l,j,m,n))
            n = n+1
        m = m+1
```



# Recall: Mean pooling



- Mean pooling computes the *mean* of the window of values
  - As opposed to the max of max pooling
- Scanning with strides is otherwise identical to max pooling

# Recap: Mean Pooling layer at layer $l$

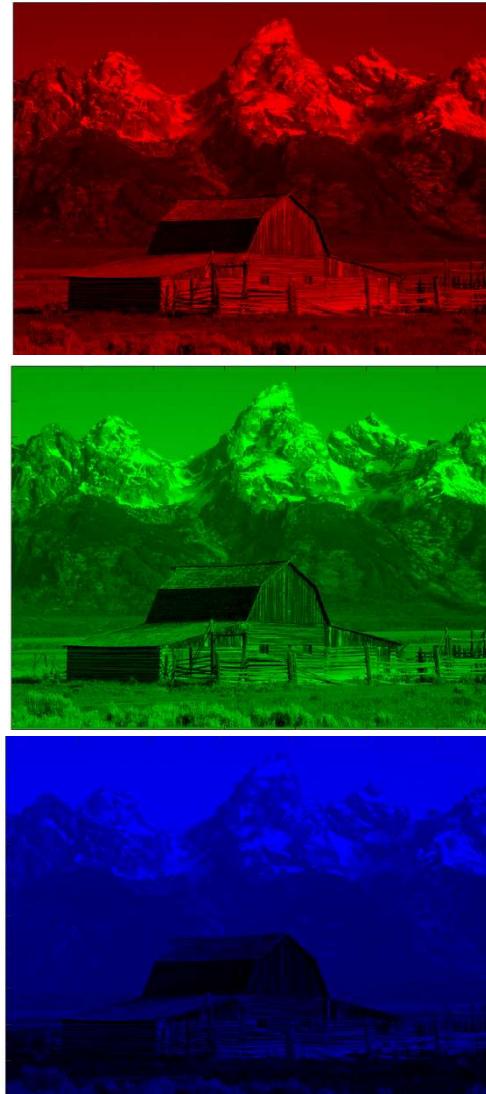
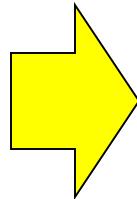
a) Performed separately for every map ( $j$ )

## Mean pooling

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

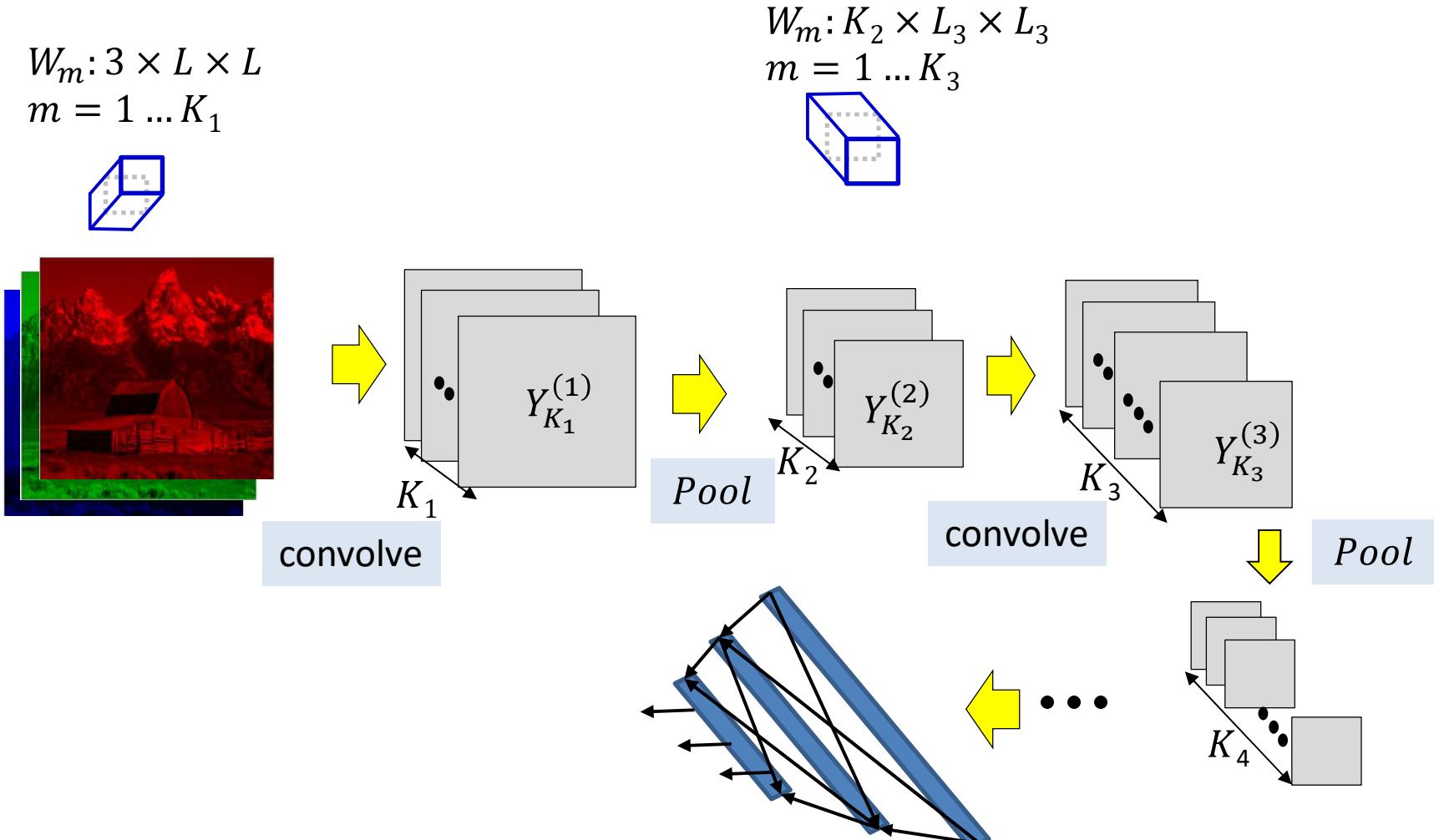


# Recap: A CNN, end-to-end



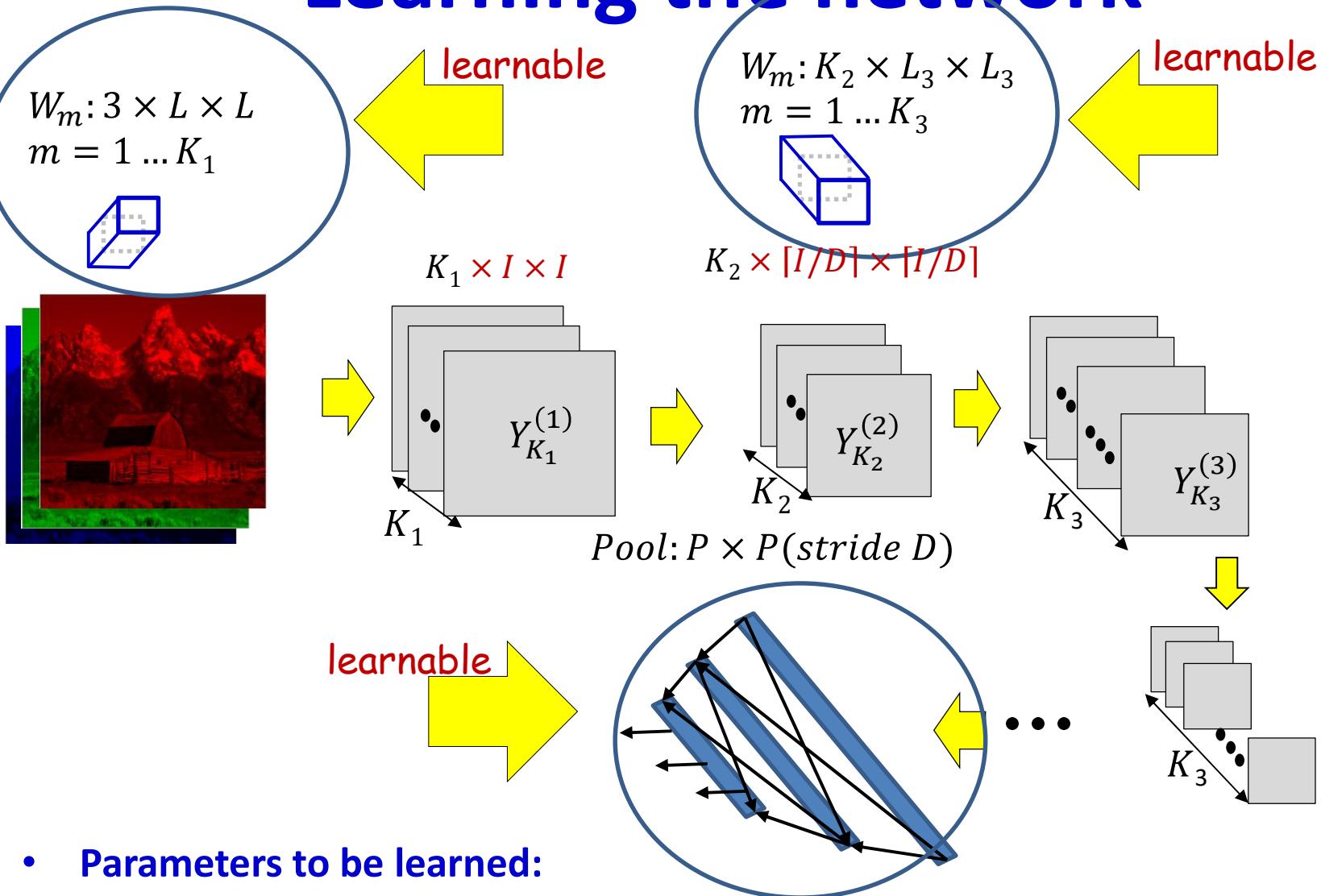
- Typical image classification task
  - Assuming maxpooling..
- Input: RGB images
  - Will assume color to be generic

# Recap: A CNN, end-to-end



- Several convolutional and pooling layers.
- The output of the last layer is “flattened” and passed through an MLP

# Learning the network



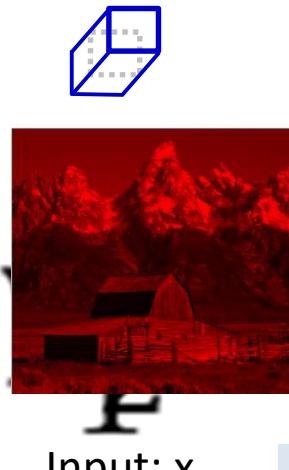
- Parameters to be learned:
  - The weights of the neurons in the final MLP
  - The (weights and biases of the) filters for every *convolutional* layer

# Recap: Learning the CNN

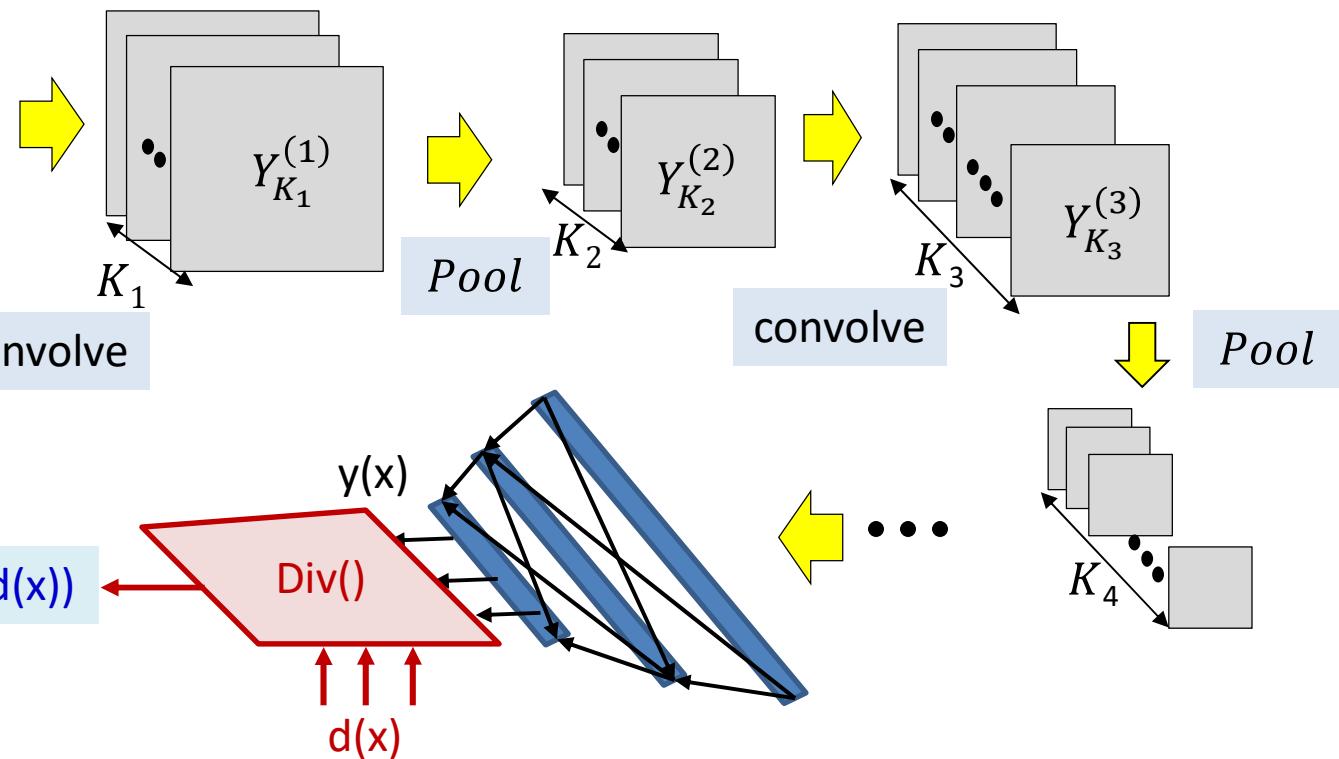
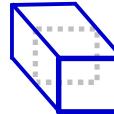
- Training is as in the case of the regular MLP
  - The *only* difference is in the *structure* of the network
- **Training examples of (Image, class) are provided**
- **Define a loss:**
  - Define a divergence between the desired output and true output of the network in response to any input
  - The 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

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



$$W_m: K_2 \times L_3 \times L_3 \\ m = 1 \dots K_3$$



- The loss for a single instance

# Recap: Problem Setup

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

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

- Minimize  $\textit{Loss}$  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:

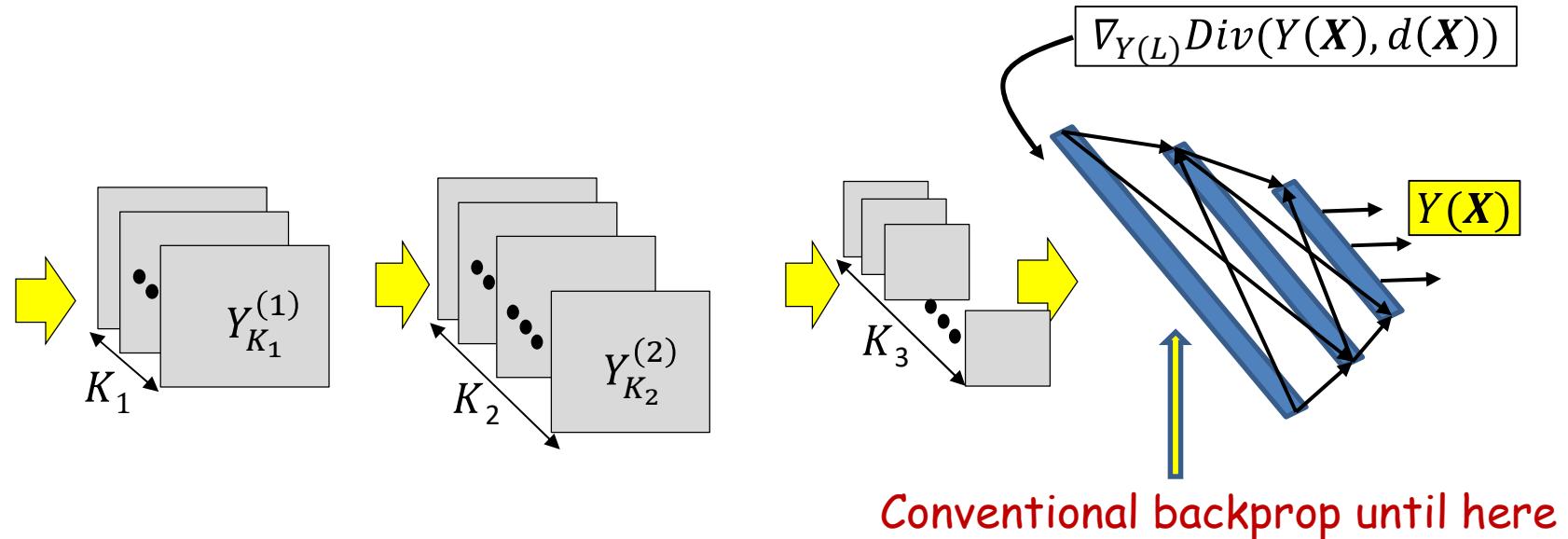
$$Loss = \frac{1}{T} \sum_i Div(Y_i, d_i)$$

- Computing the derivative

Total derivative:

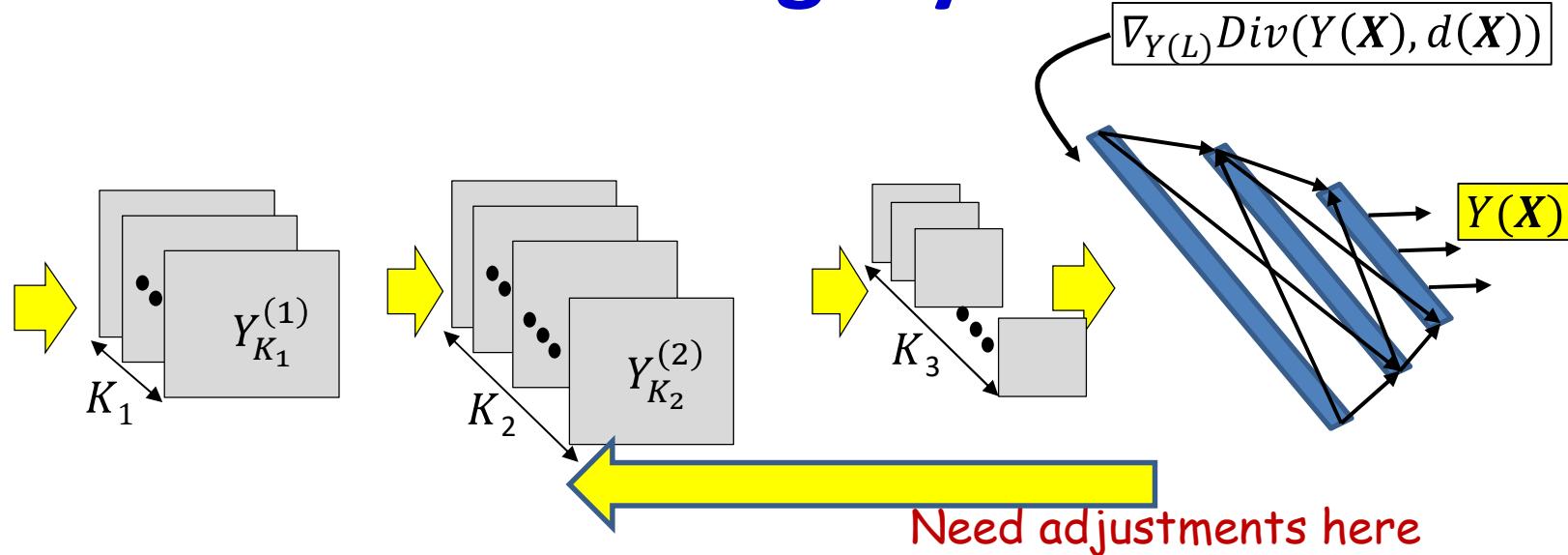
$$\frac{dLoss}{dw} = \frac{1}{T} \sum_i \frac{dDiv(Y_i, d_i)}{dw}$$

# Backpropagation: Final flat layers



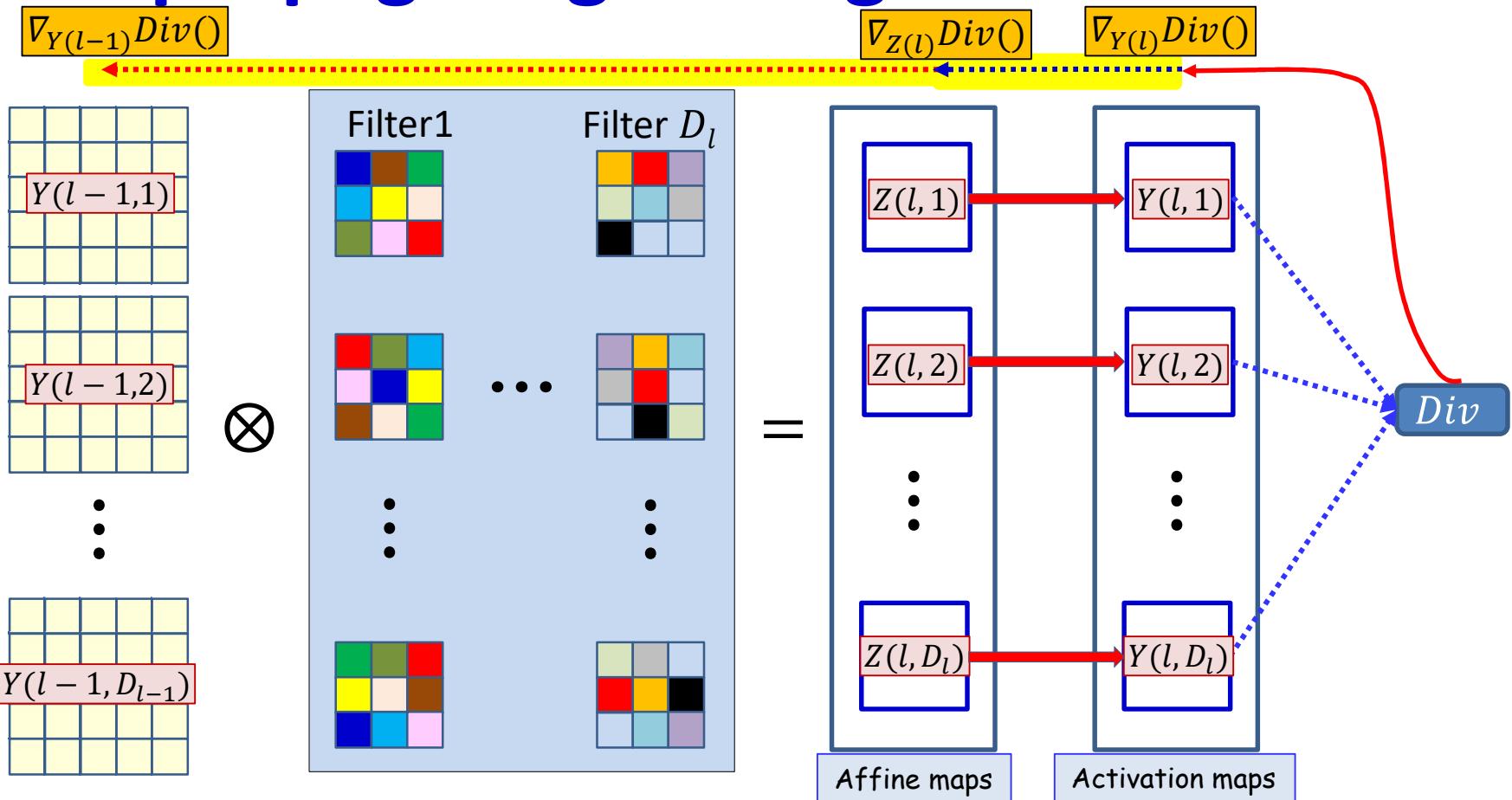
- For each training instance: First, a forward pass through the net
- Then the backpropagation of the derivative of the divergence
- Backpropagation continues in the usual manner until the computation of the derivative of the divergence w.r.t the inputs to the first “flat” layer
  - Important to recall: the first flat layer is only the “unrolling” of the maps from the final convolutional layer

# Backpropagation: Convolutional and Pooling layers



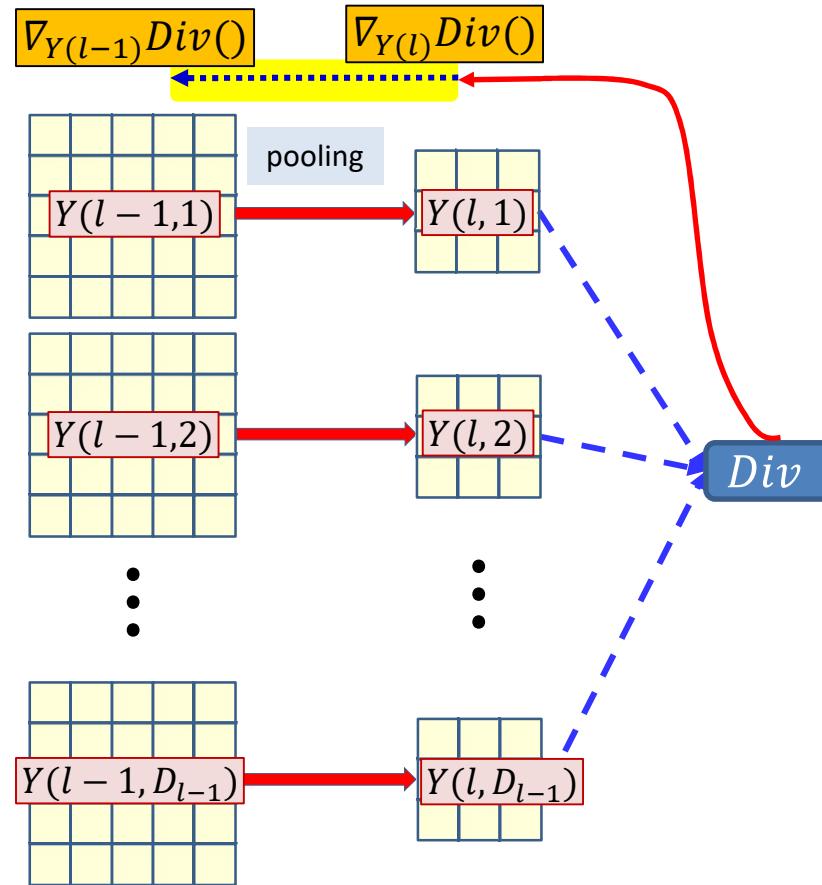
- Backpropagation from the flat MLP requires special consideration of
  - The shared computation in the convolution layers
  - The pooling layers (particularly maxout)

# Backpropagating through the convolution



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

# 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:**
    - 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.  $Y(l)$

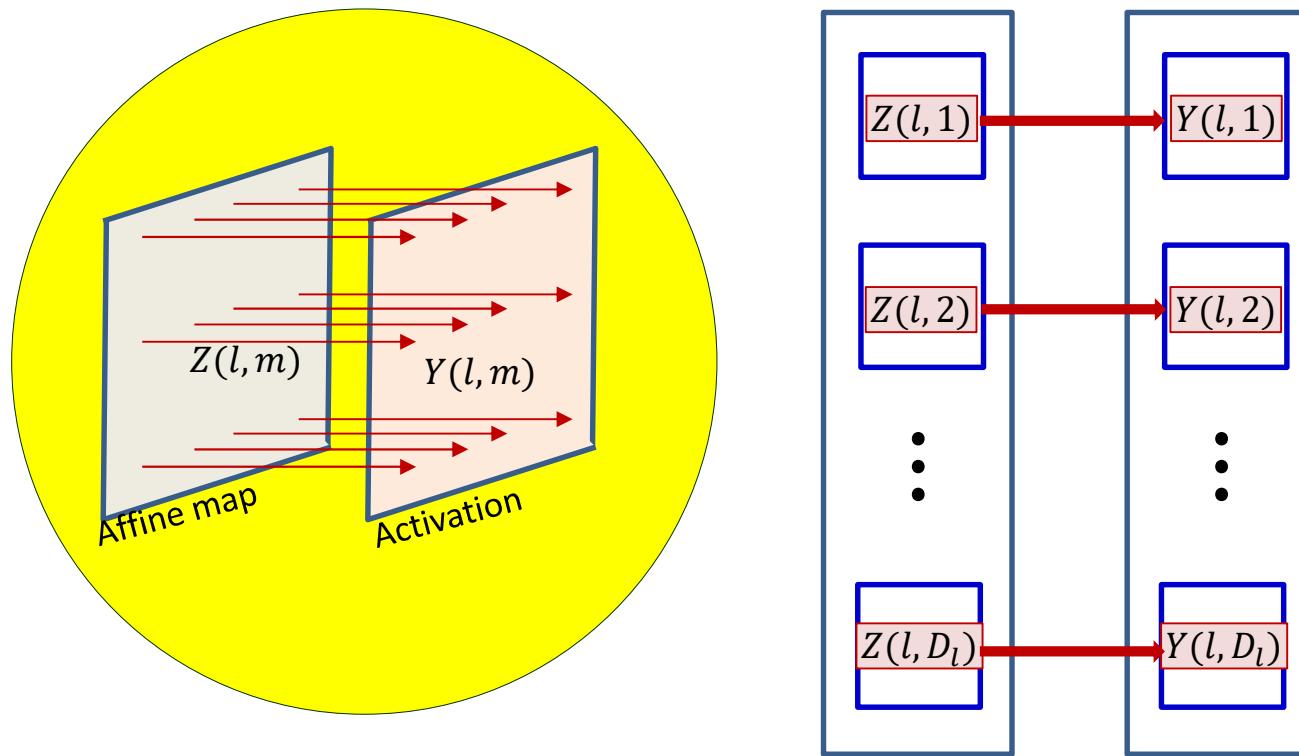
# Backpropagation: Convolutional and Pooling layers

- **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.  $Y(l)$

# 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.  $Y(l)$

# Backpropagating through the activation

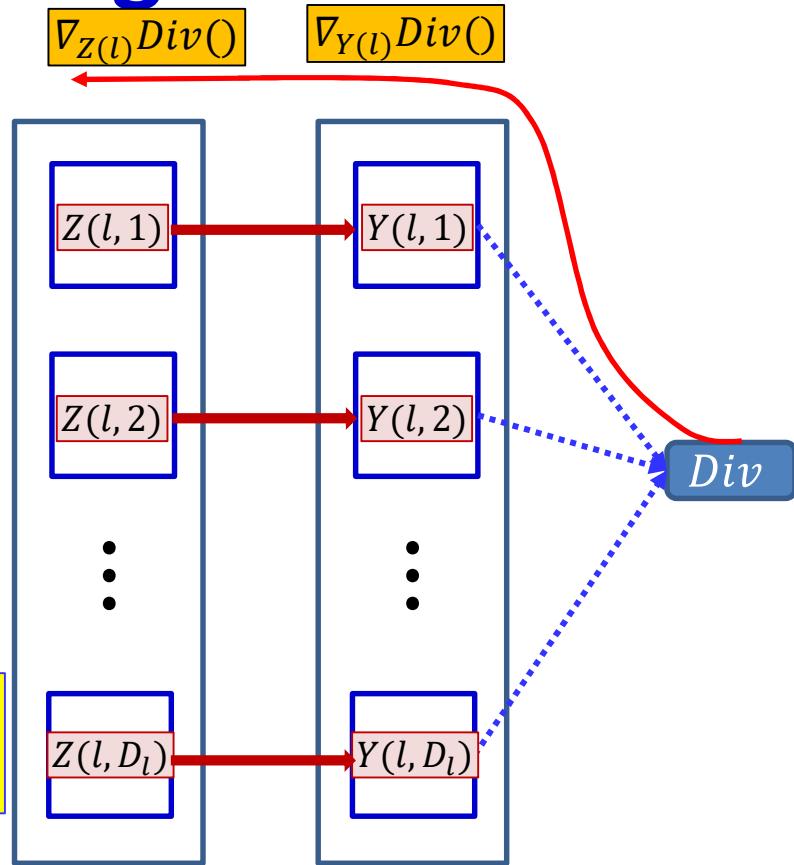
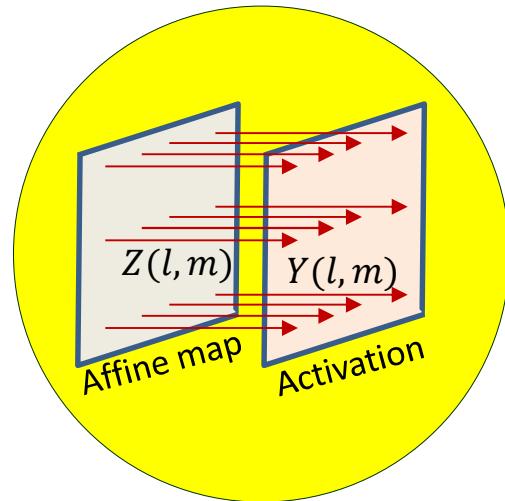


- **Forward computation:** The activation maps are obtained by point-wise application of the activation function to the affine maps

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

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

# Backpropagating through the activation



- **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:

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

- 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:**
    - ✓ 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.  $Y(l)$

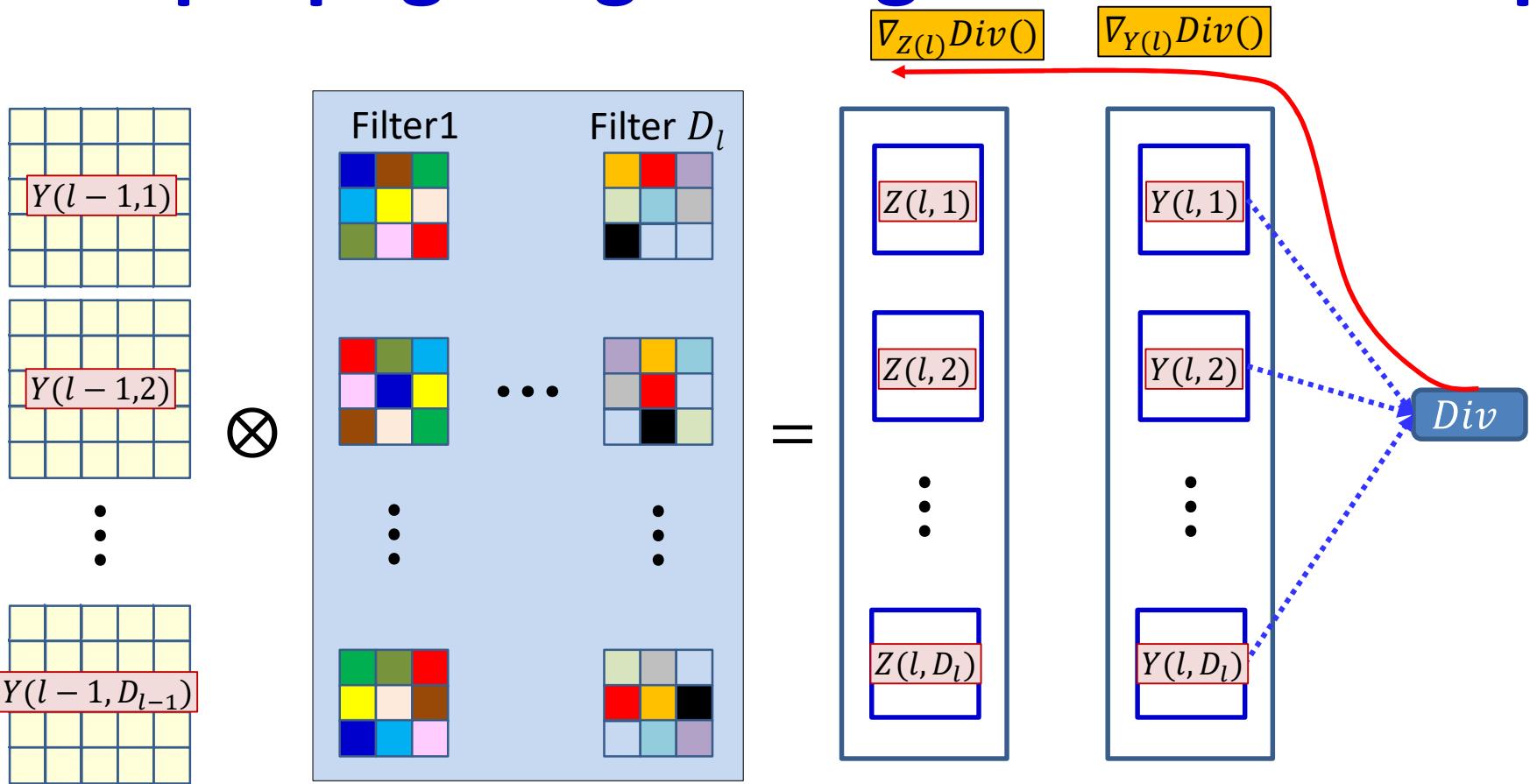
# Backpropagating through affine map

- 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 affine map

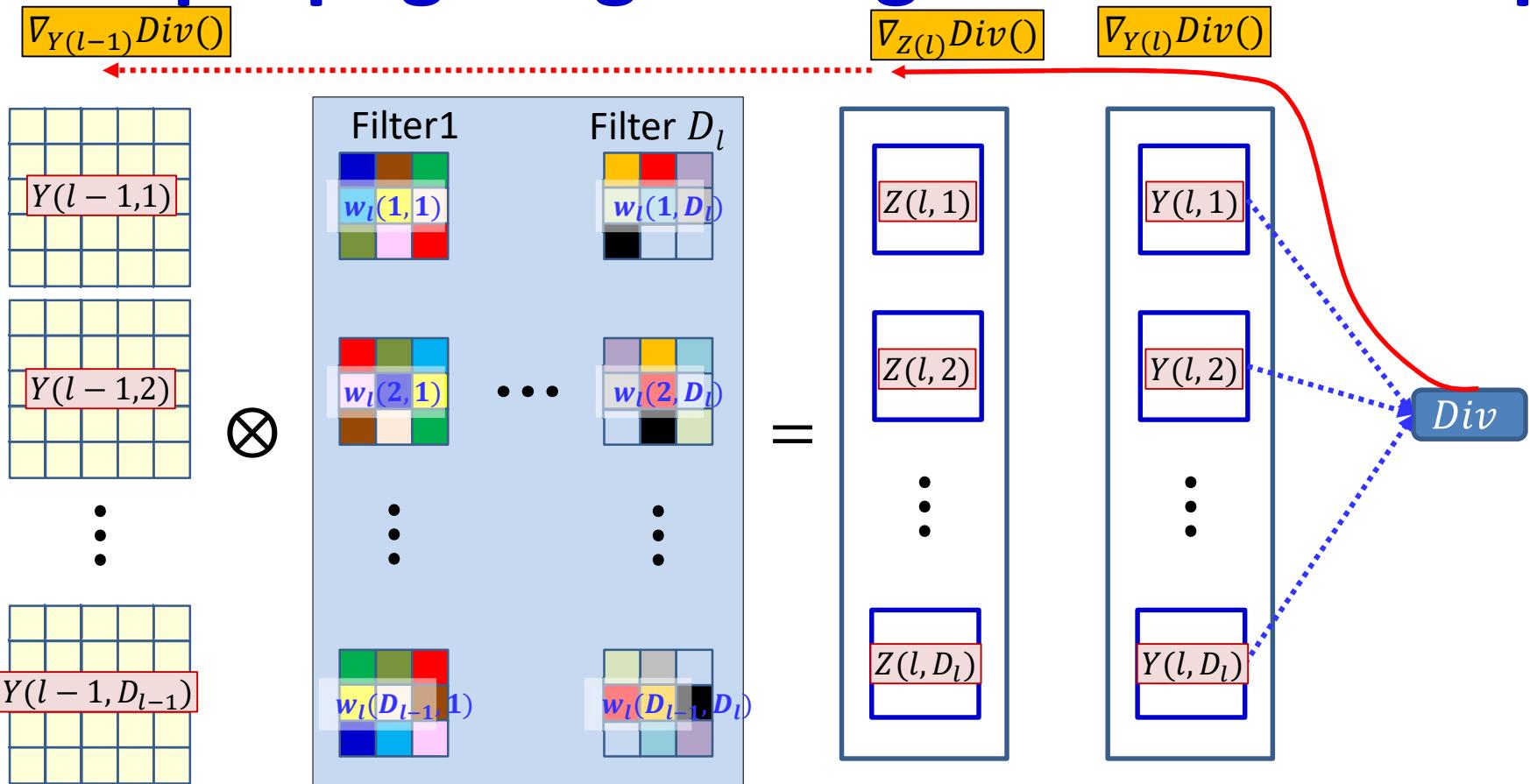
- Forward affine computation:
  - Compute affine maps  $z(l, n, x, y)$  from previous layer maps  $y(l - 1, m, x, y)$  and filters  $w_l(m, n, x, y)$
- Backpropagation: Given  $\frac{dDiv}{dz(l,n,x,y)}$ 
  - Compute derivative w.r.t.  $y(l - 1, m, x, y)$
  - Compute derivative w.r.t.  $w_l(m, n, x, y)$

# Backpropagating through the affine map



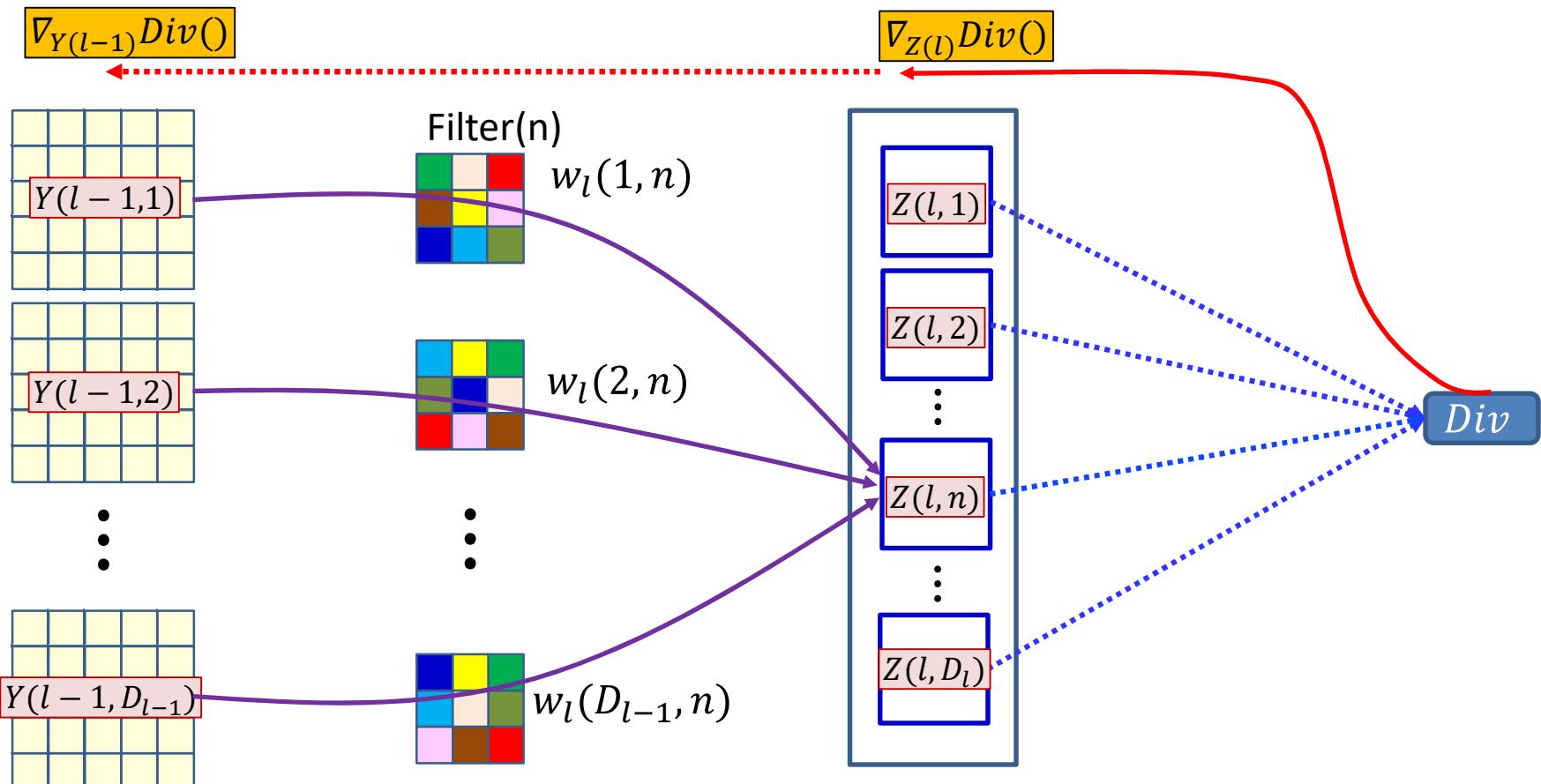
- We already have the derivative w.r.t  $Z(l, *)$ 
  - Having backpropagated it past  $Y(l, *)$

# Backpropagating through the affine map



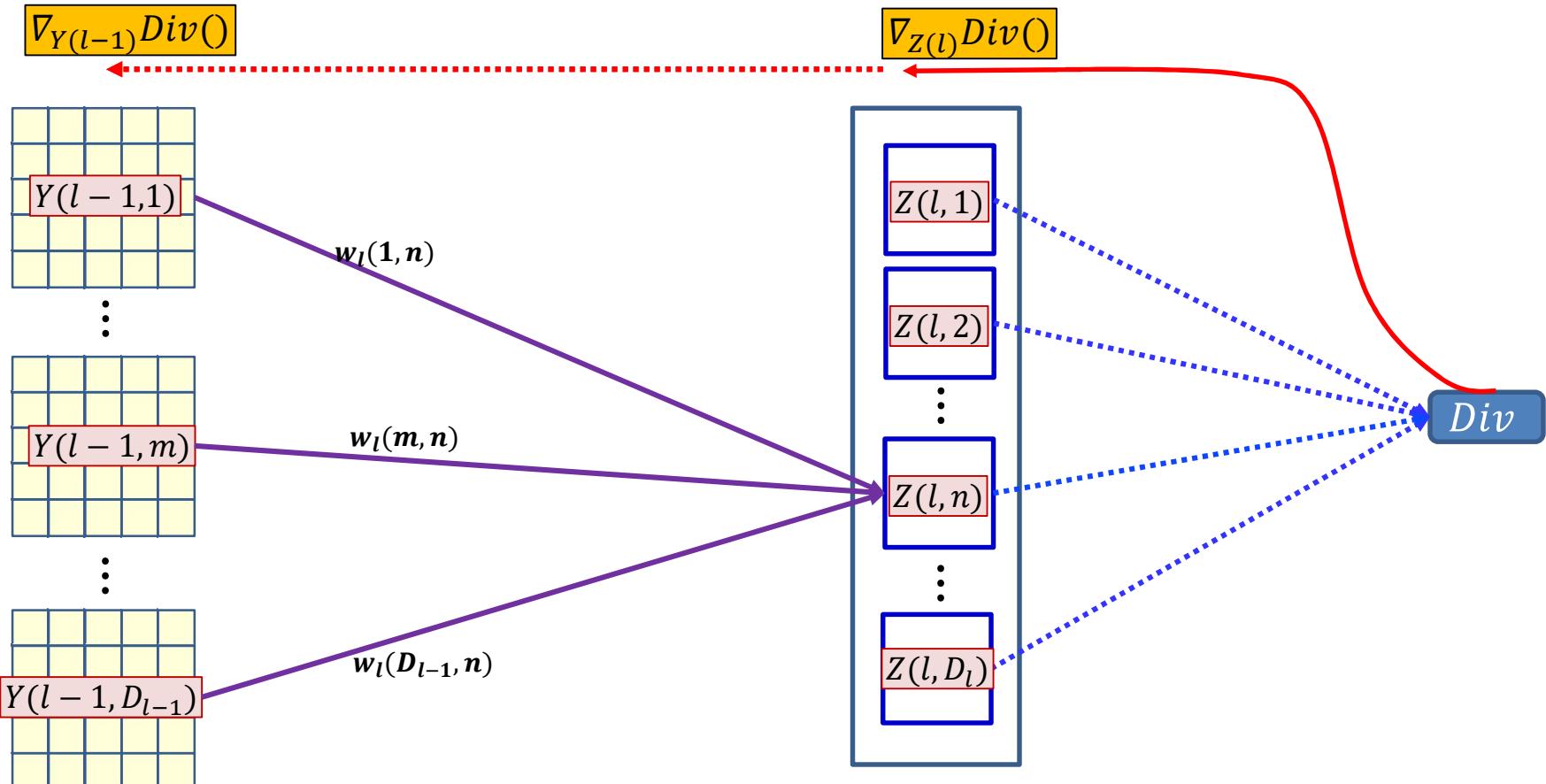
- We already have the derivative w.r.t  $Z(l,*)$ 
  - Having backpropagated it past  $Y(l,*)$
- We must compute the derivative w.r.t  $Y(l - 1,*)$

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



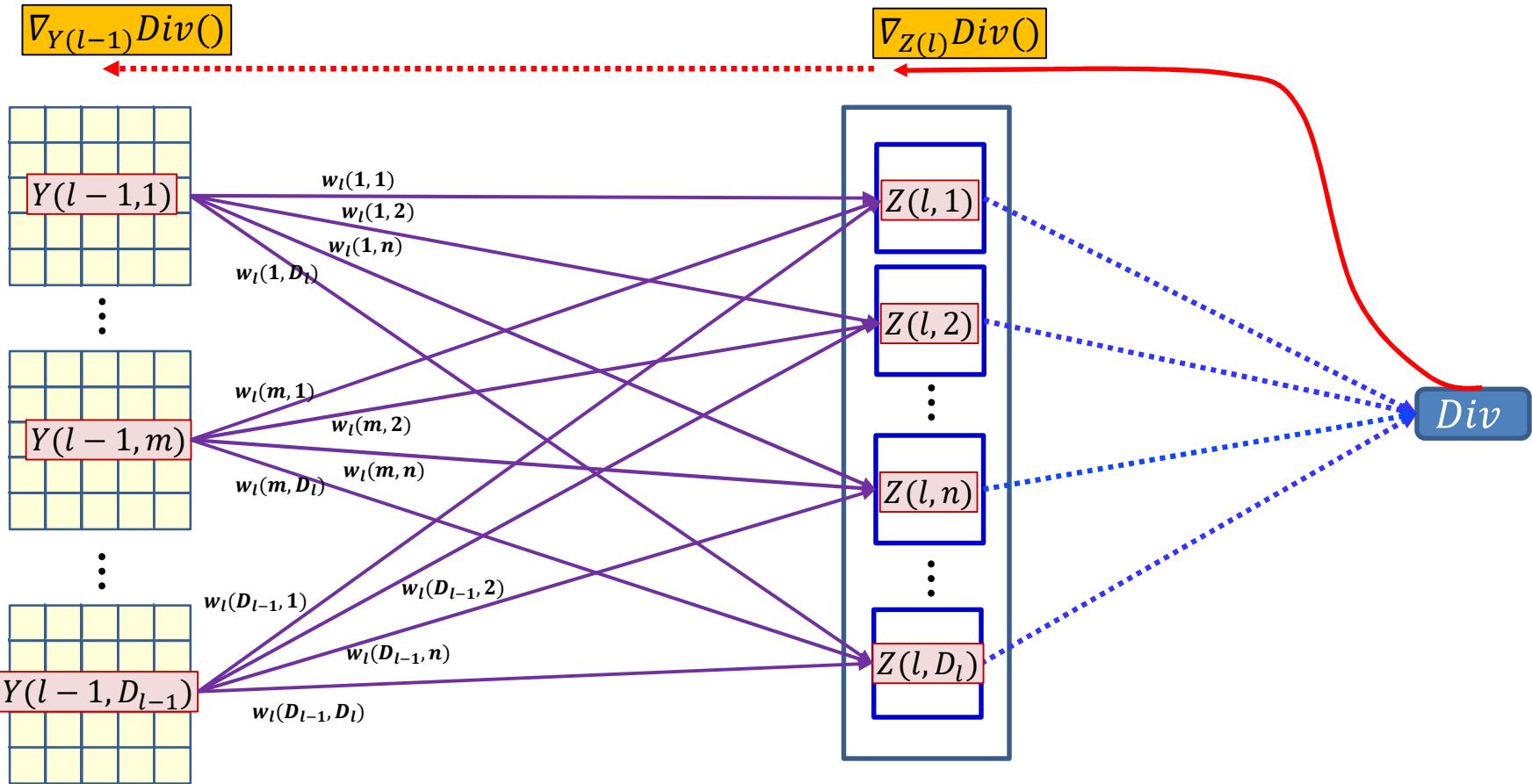
- Each  $Y(l - 1, m)$  map influences  $Z(l, n)$  through the  $m$ th “plane” of the  $n$ th filter  $w_l(m, n)$

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



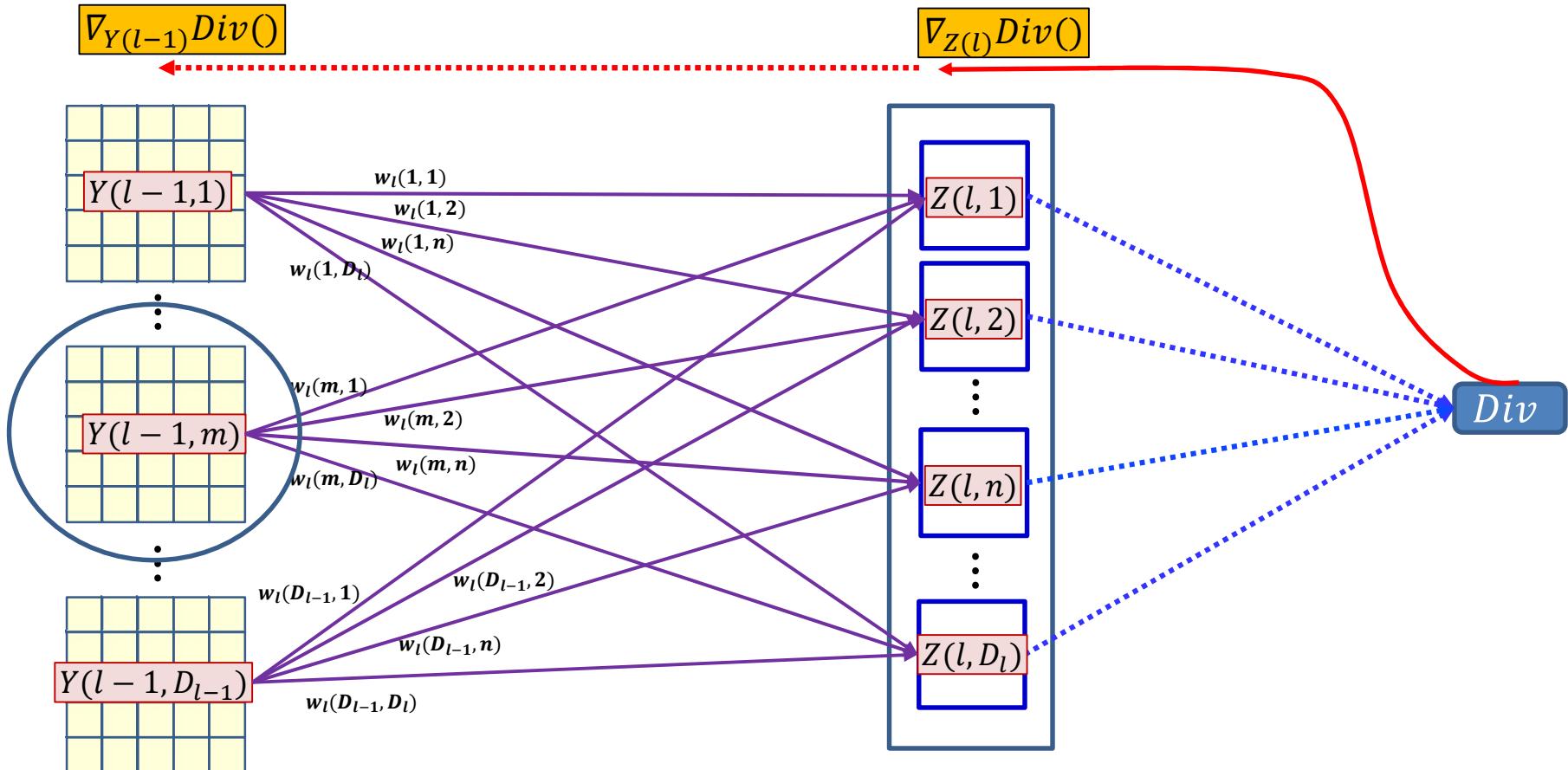
- Each  $Y(l - 1, m)$  map influences  $Z(l, n)$  through the  $m$ th “plane” of the  $n$ th filter  $w_l(m, n)$

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



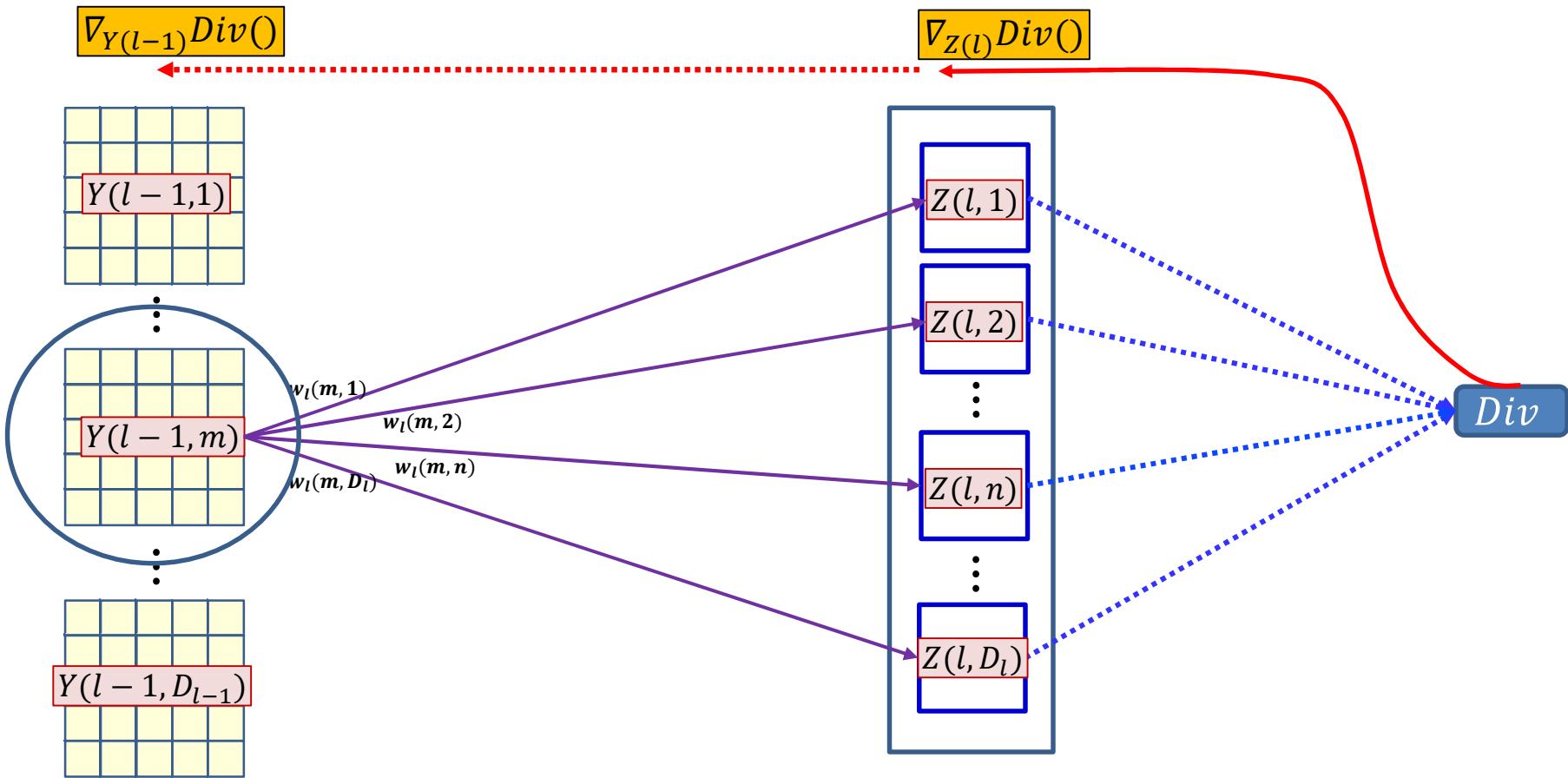
- Each  $Y(l - 1, m)$  map influences  $Z(l, n)$  through the  $m$ th “plane” of the  $n$ th filter  $w_l(m, n)$

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



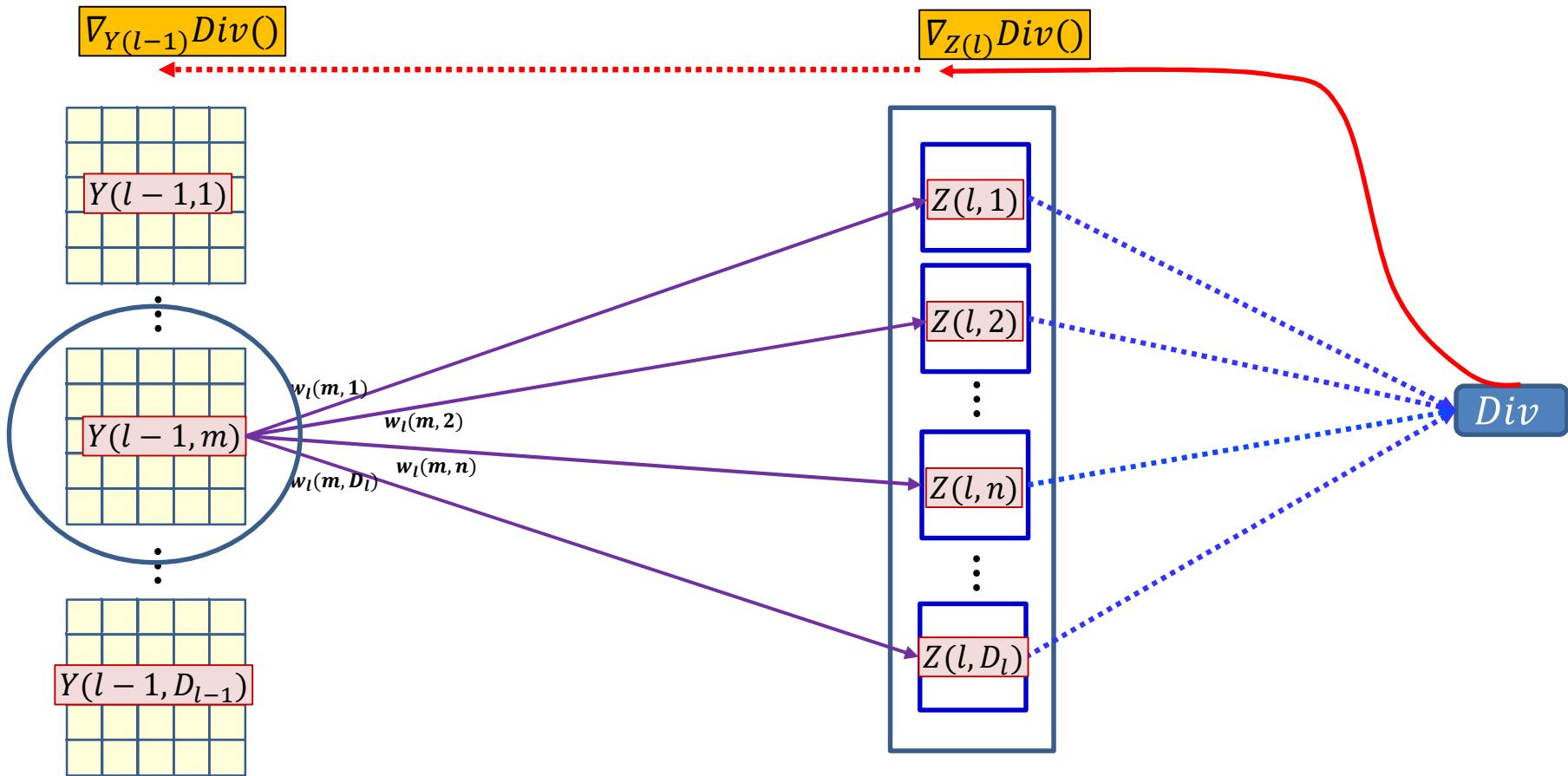
- Each  $Y(l - 1, m)$  map influences  $Z(l, n)$  through the  $m$ th “plane” of the  $n$ th filter  $w_l(m, n)$

# Dependency diagram for a single map



- Each  $Y(l - 1, m)$  map influences  $Z(l, n)$  through the  $m$ th “plane” of the  $n$ th filter  $w_l(m, n)$
- $Y(l - 1, m, *, *)$  influences the divergence through all  $Z(l, n, *, *)$  maps

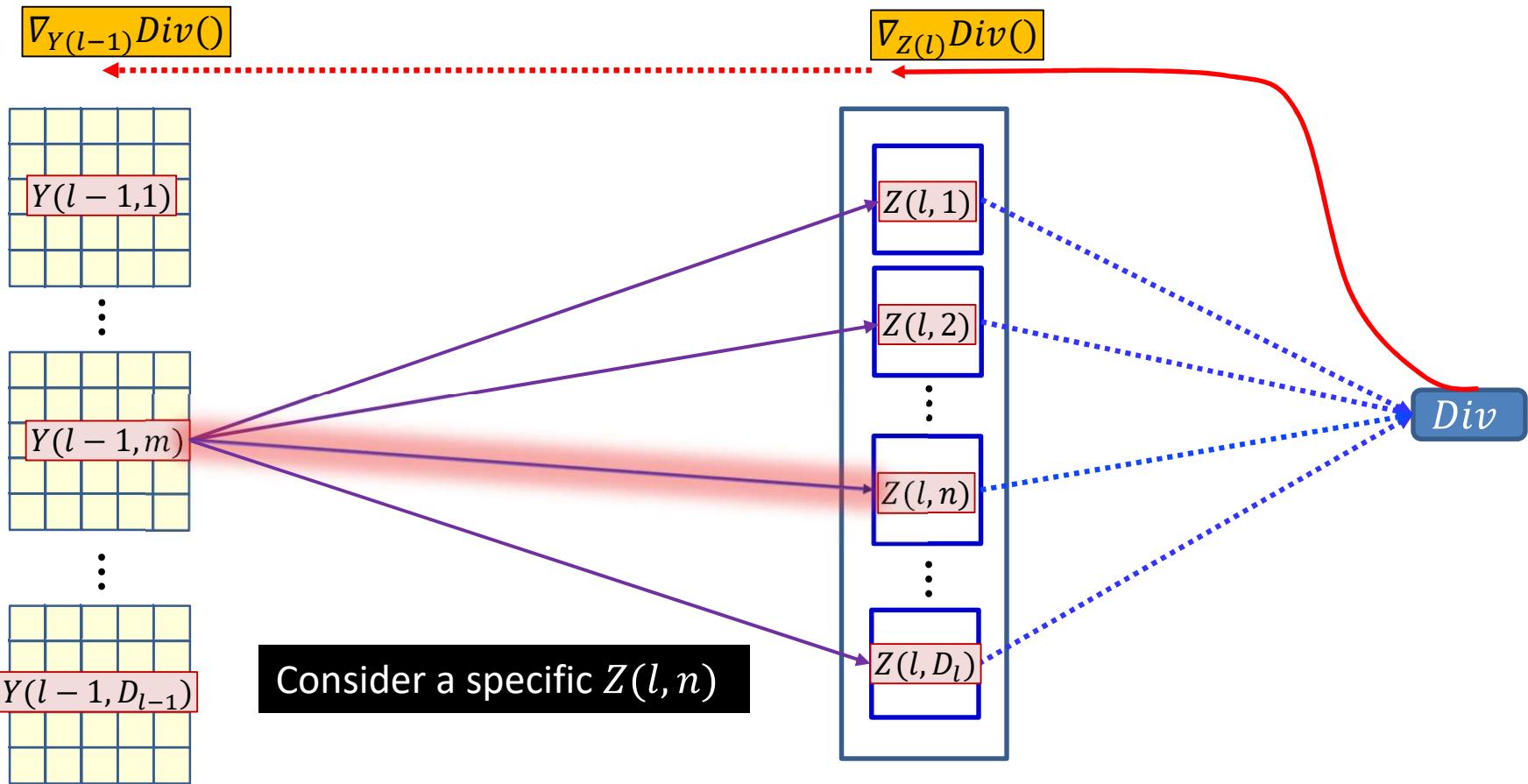
# Dependency diagram for a single map



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

- 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



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

- 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

1 <small>x1</small>	1 <small>x0</small>	1 <small>x1</small>	0	0
0 <small>x0</small>	1 <small>x1</small>	1 <small>x0</small>	1	0
0 <small>x1</small>	0 <small>x0</small>	1 <small>x1</small>	1	1
0	0	1	1	0
0	1	1	0	0

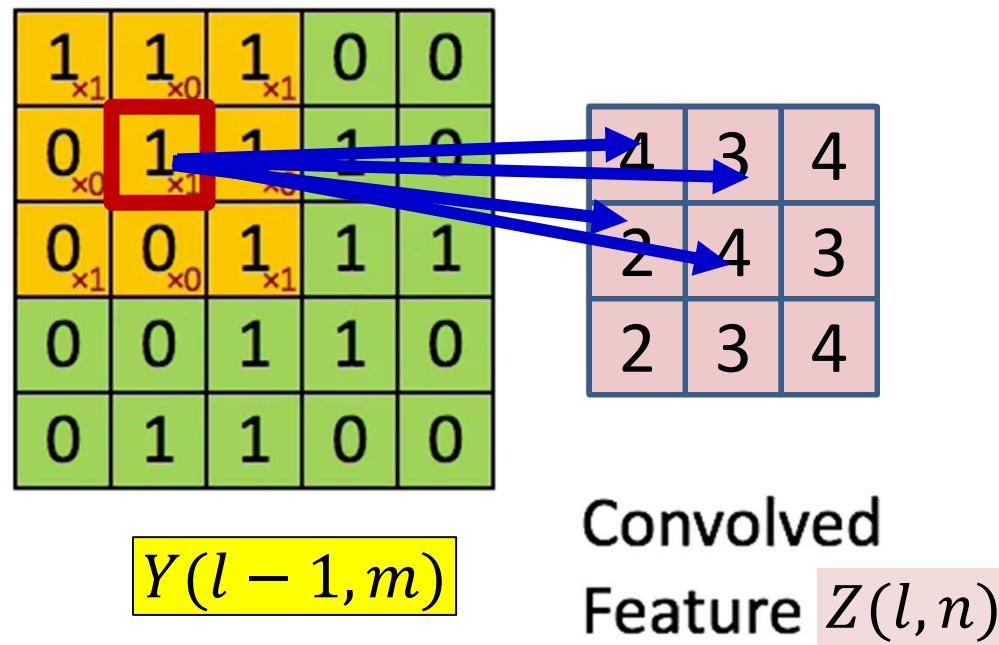
$$Y(l - 1, m)$$

4		

Convolved  
Feature  $Z(l, n)$

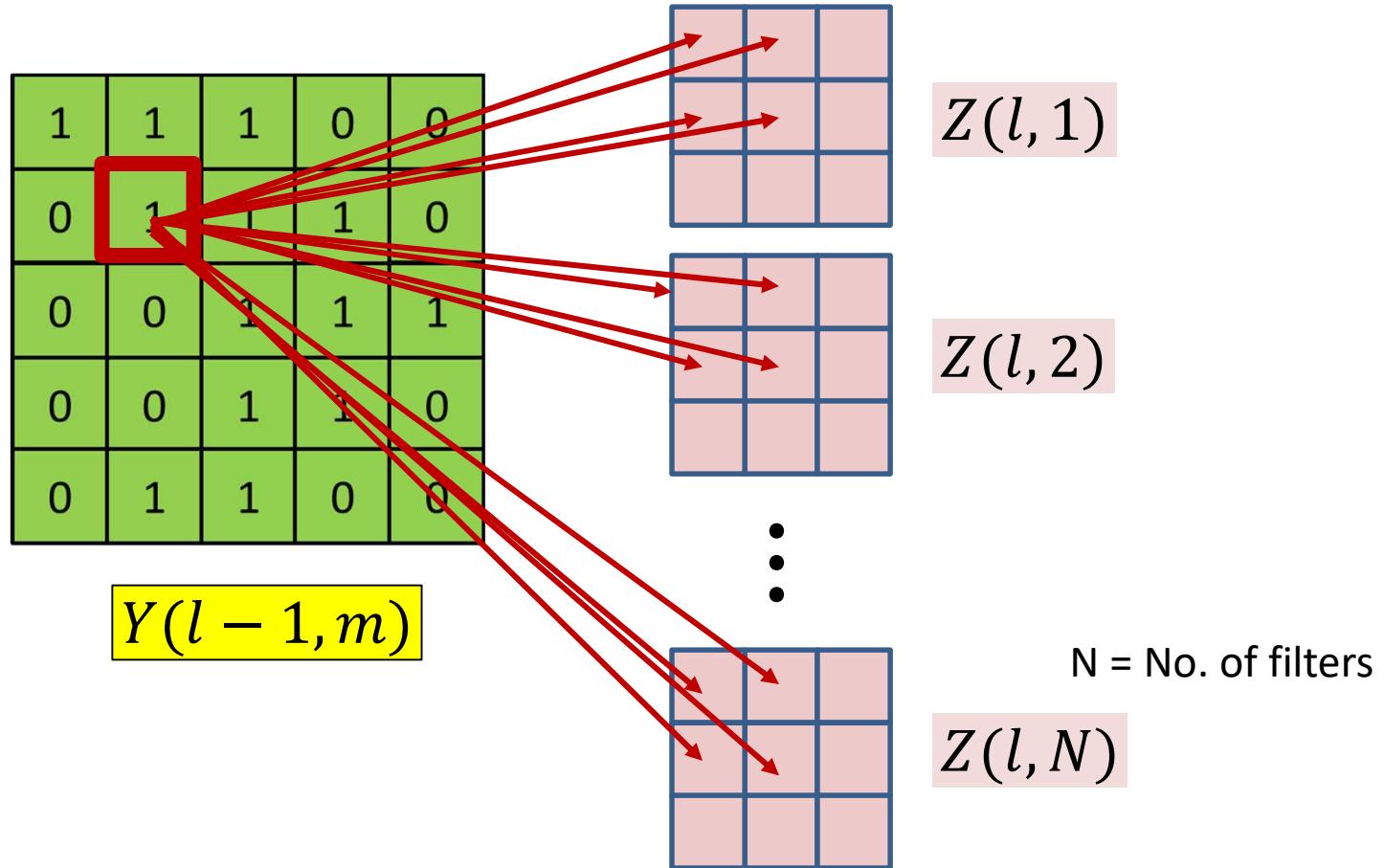
- Each  $Y(l - 1, m, x, y)$  affects several  $z(l, n, x', y')$  terms

# BP: Convolutional layer



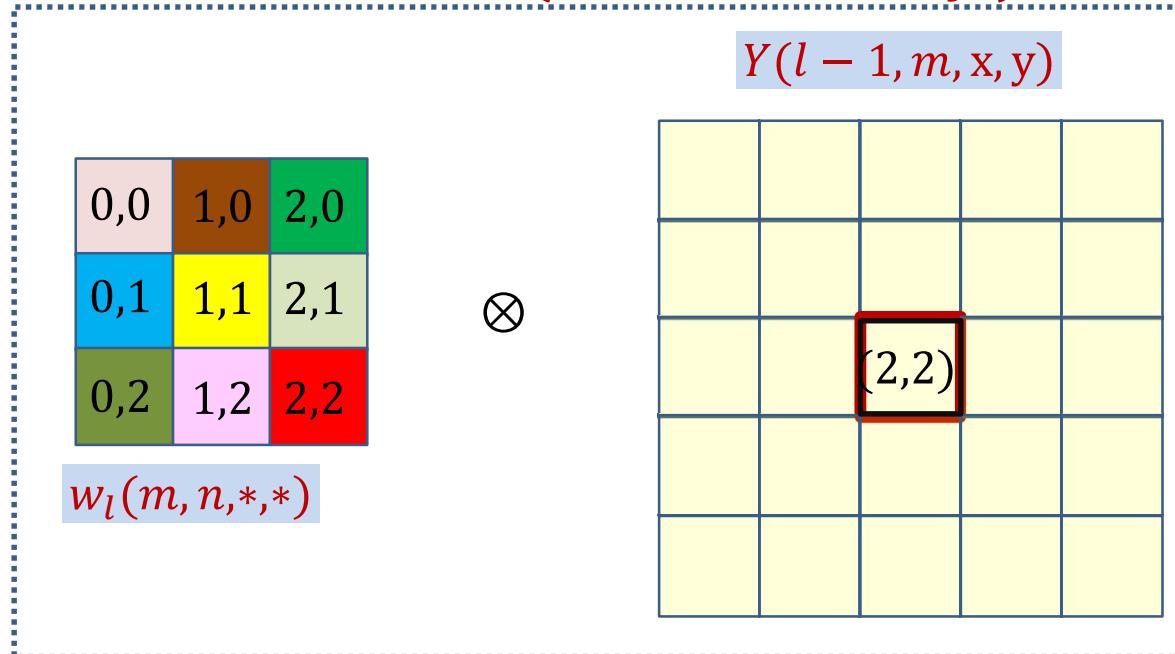
- Each  $Y(l - 1, m, x, y)$  affects several  $z(l, n, x', y')$  terms

# BP: Convolutional layer



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

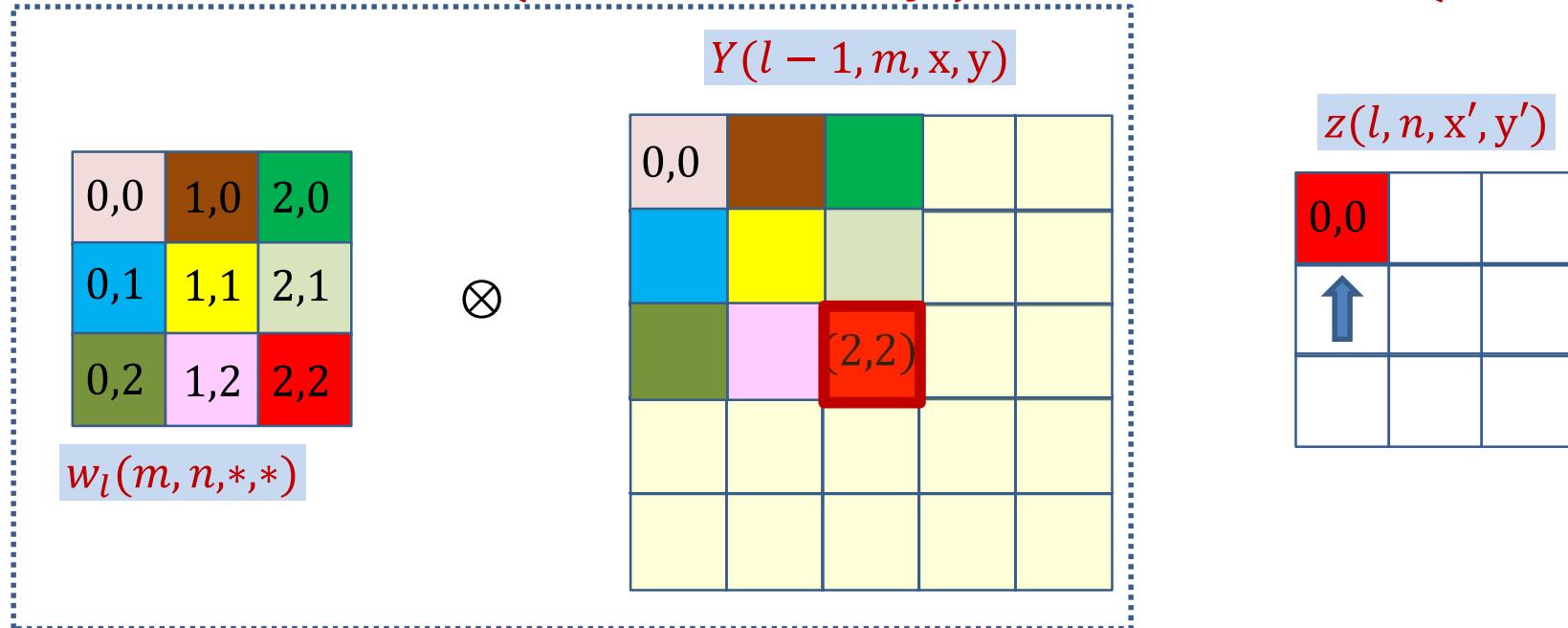
# How a single $Y(l - 1, m, x, y)$ influences $z(l, n, x', y')$



Assuming indexing  
begins at 0

- 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 $z(l, n, x', y')$

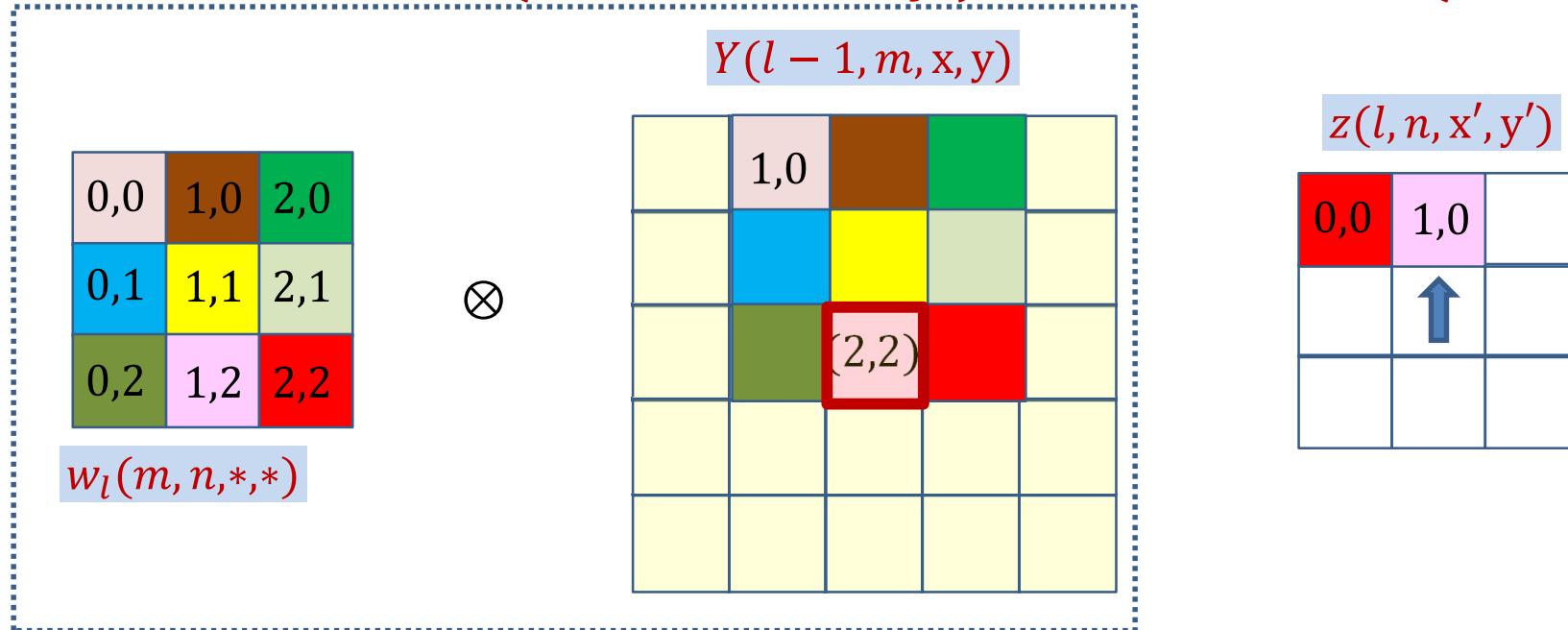


$$z(l, n, 0,0) += Y(l - 1, m, 2,2) w_l(m, n, 2,2)$$

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

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

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

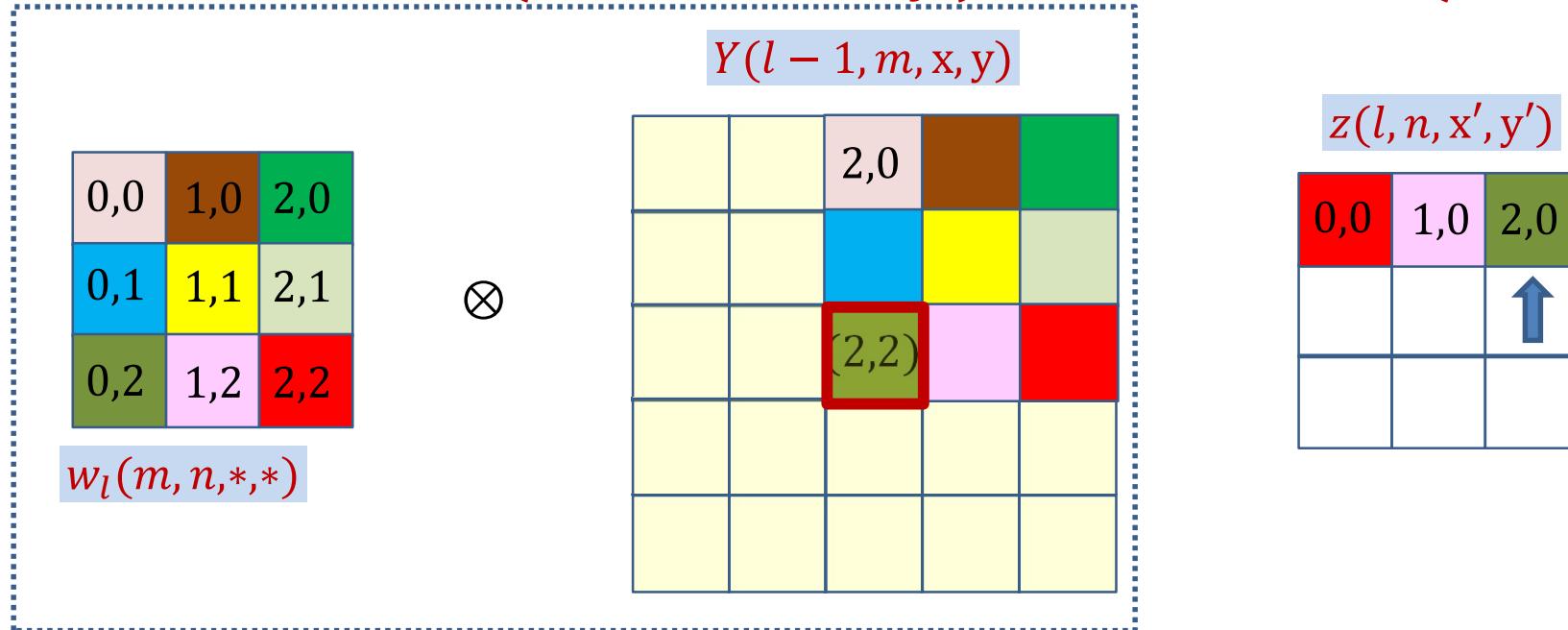


$$z(l, n, 1, 0) += Y(l - 1, m, 2, 2) w_l(m, n, 1, 2)$$

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

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

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

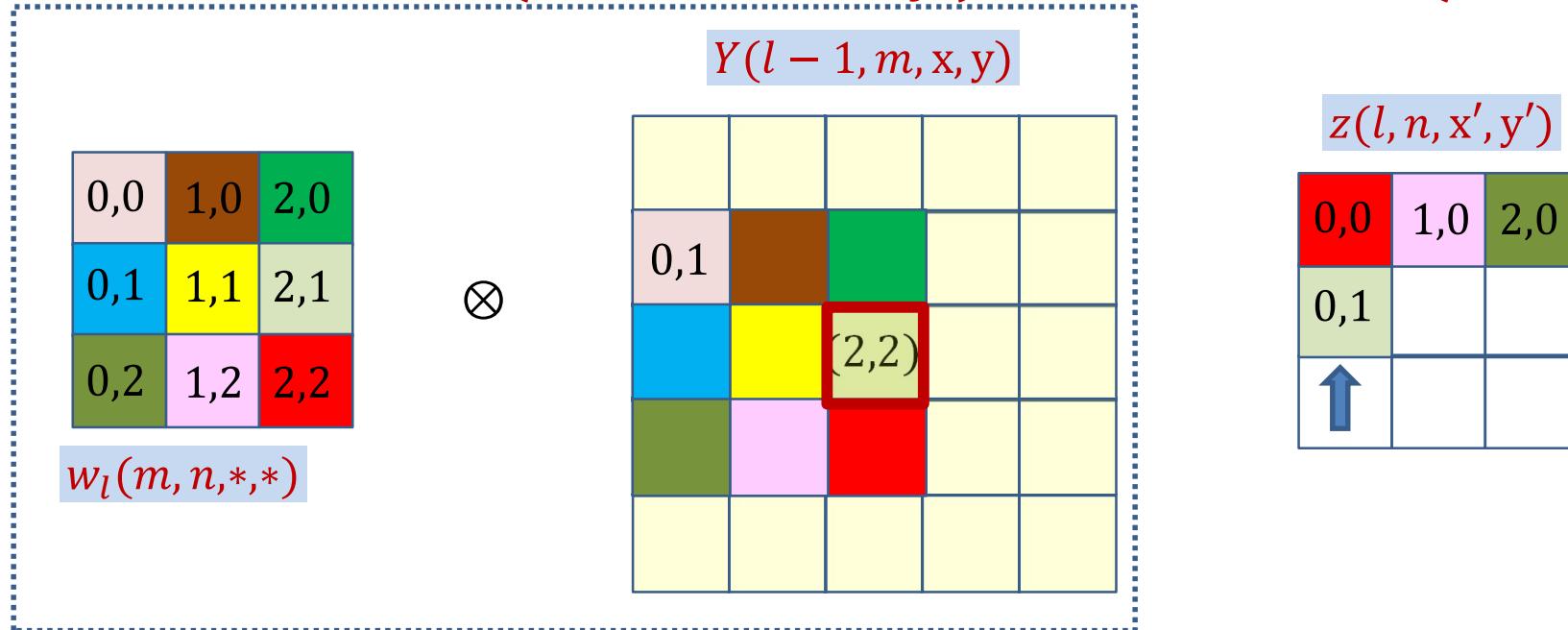


$$z(l, n, 2,0) += Y(l - 1, m, 2,2) w_l(m, n, 0,2)$$

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

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

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

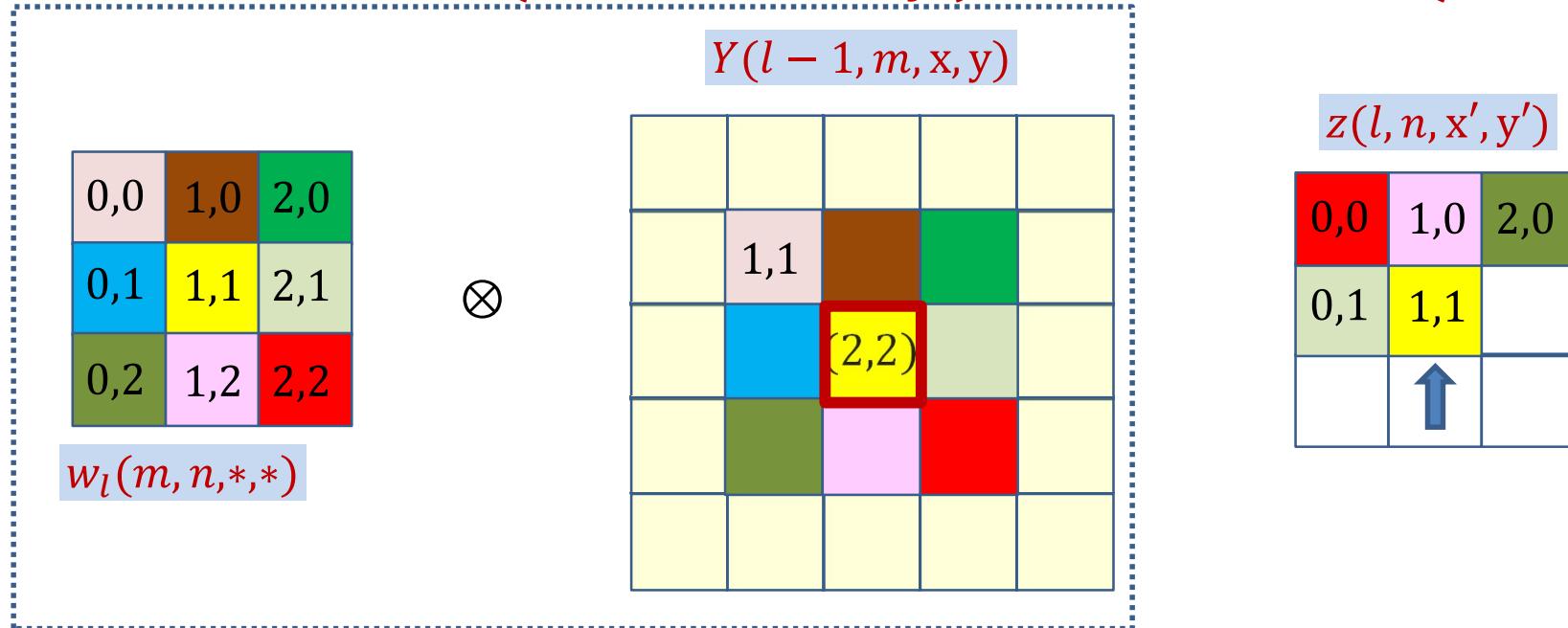


$$z(l, n, 0,1) += Y(l - 1, m, 2,2) w_l(m, n, 2,1)$$

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

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

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

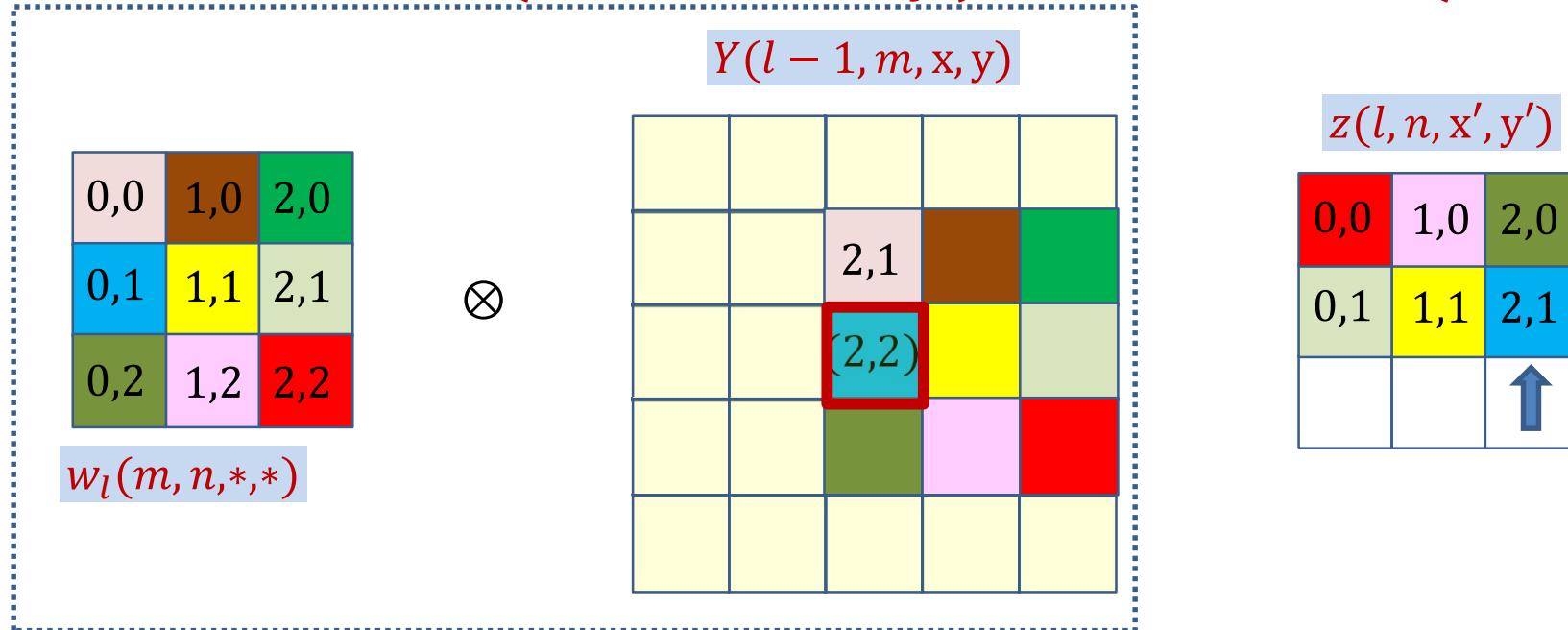


$$z(l, n, 1,1) += Y(l - 1, m, 2,2) w_l(m, n, 1,1)$$

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

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

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

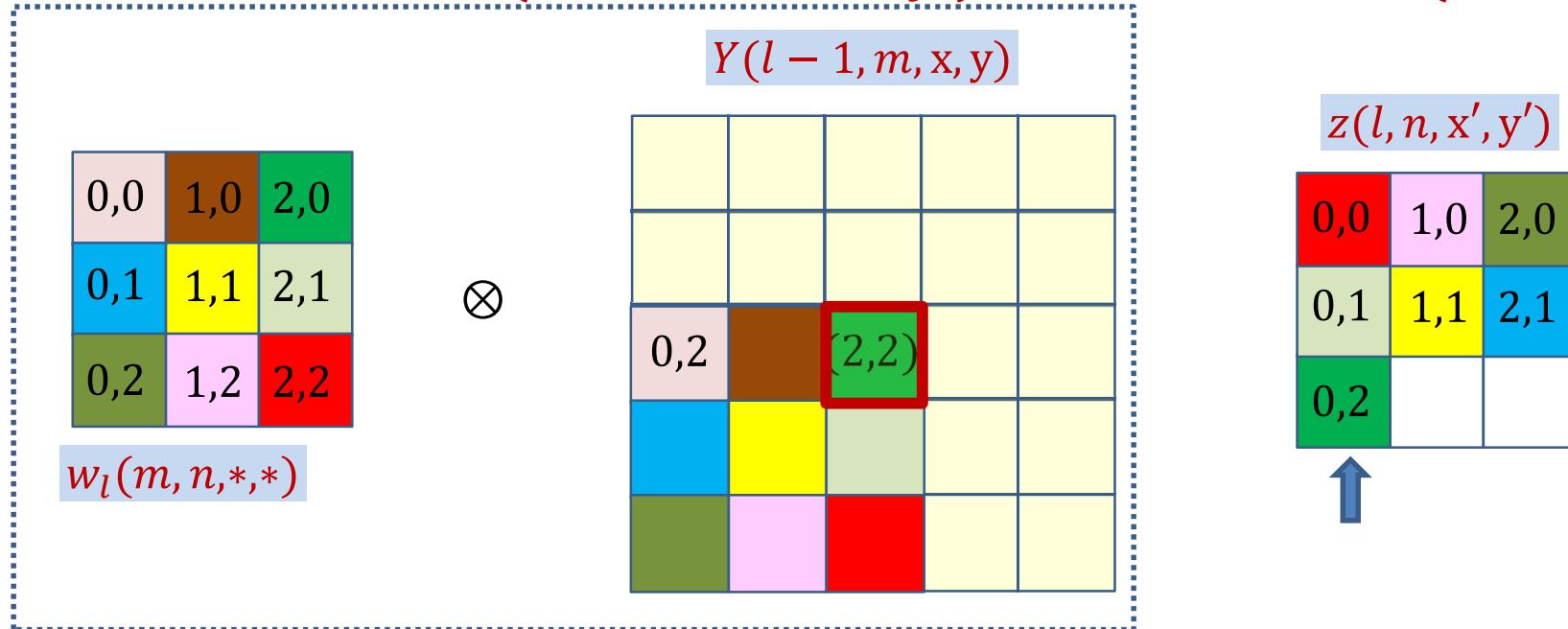


$$z(l, n, 2,1) += Y(l - 1, m, 2,2)w_l(m, n, 0,1)$$

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

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

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

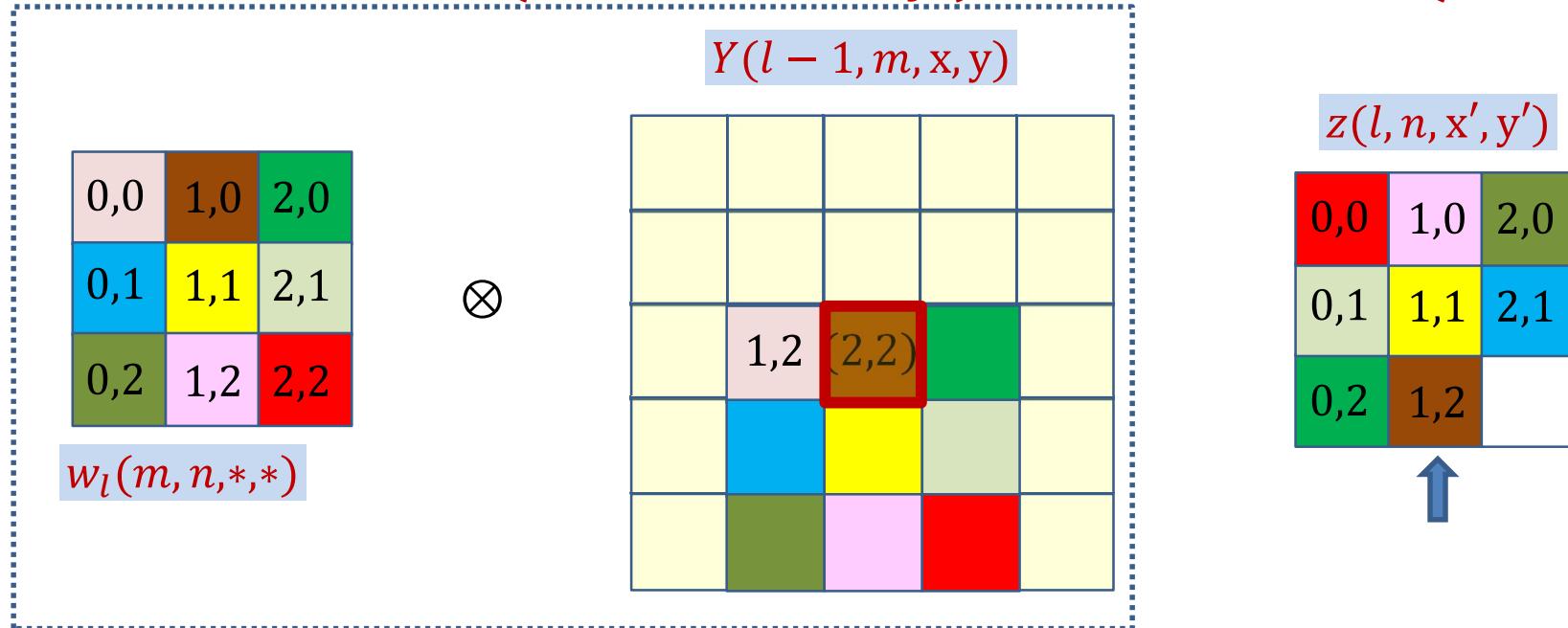


$$z(l, n, 0, 2) += Y(l - 1, m, 2, 2) w_l(m, n, 2, 0)$$

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

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

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

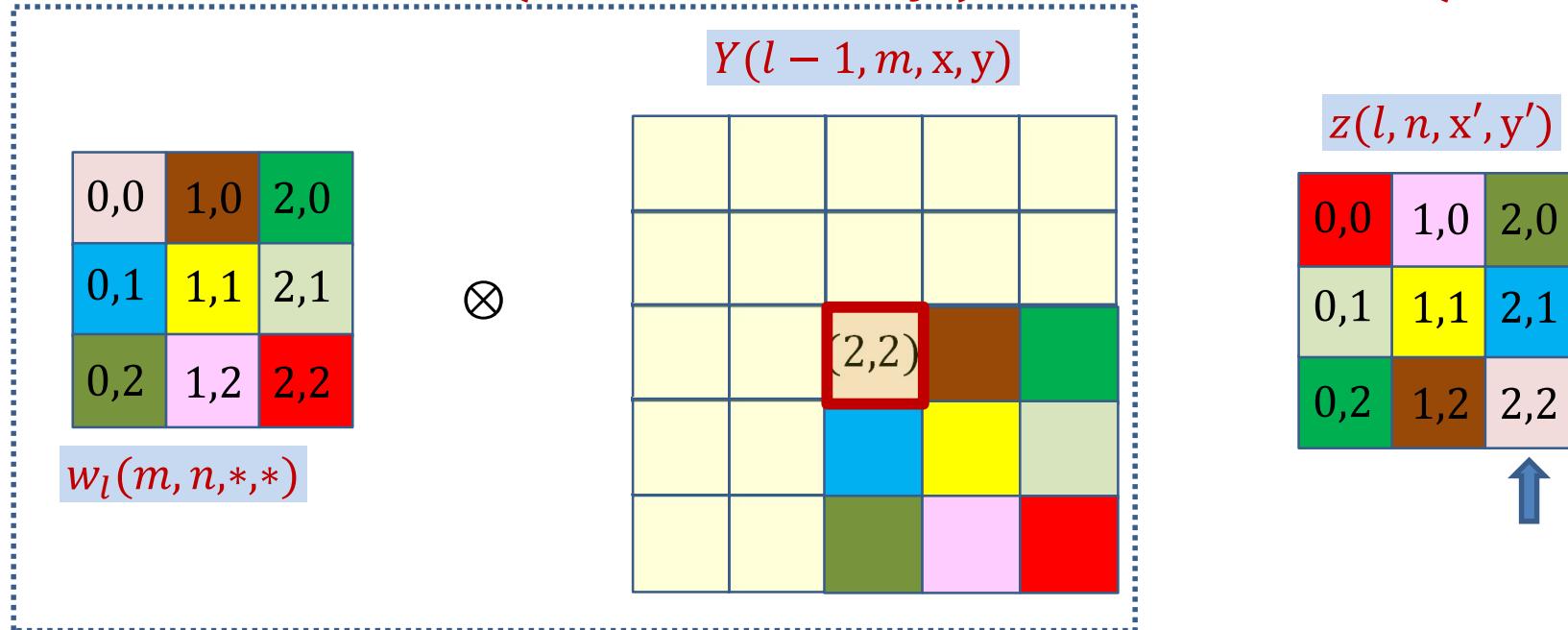


$$z(l, n, 1, 2) += Y(l - 1, m, 2, 2) w_l(m, n, 2, 1)$$

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

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

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

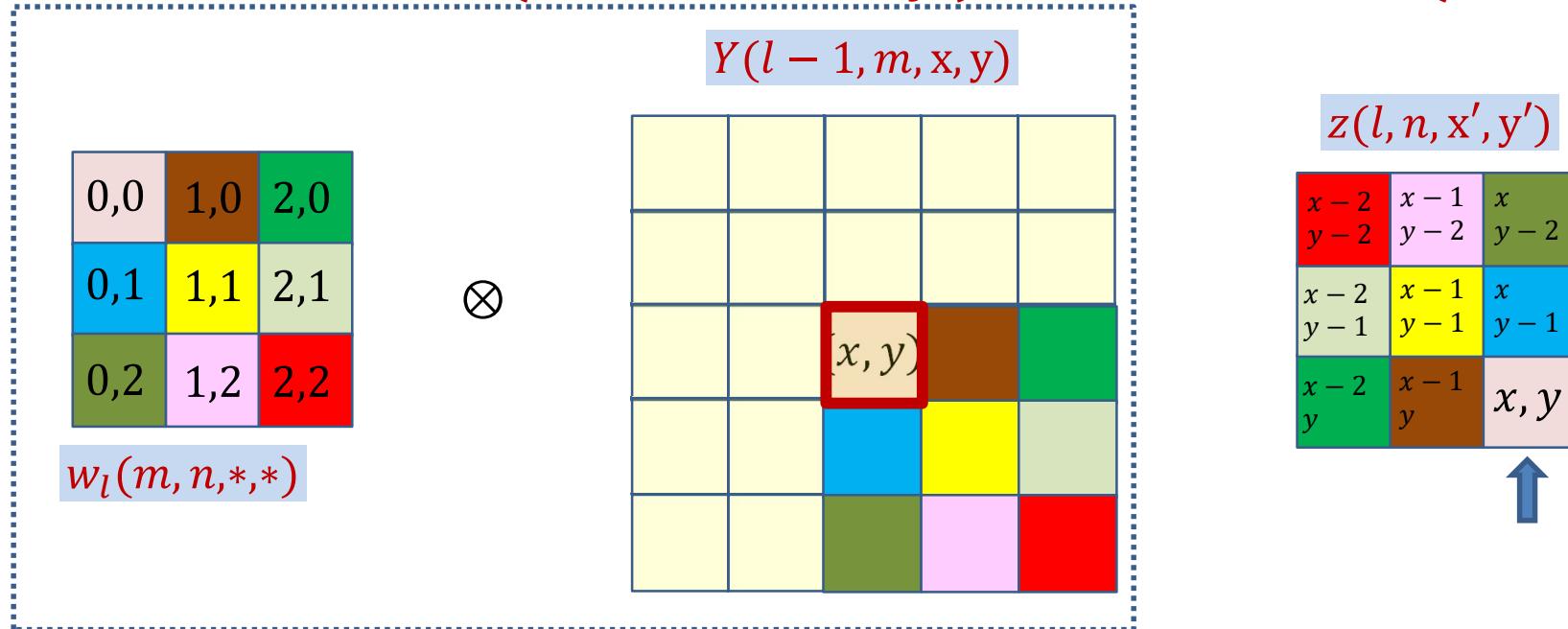


$$z(l, n, 2, 2) += Y(l - 1, m, 2, 2) w_l(m, n, 0, 0)$$

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

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

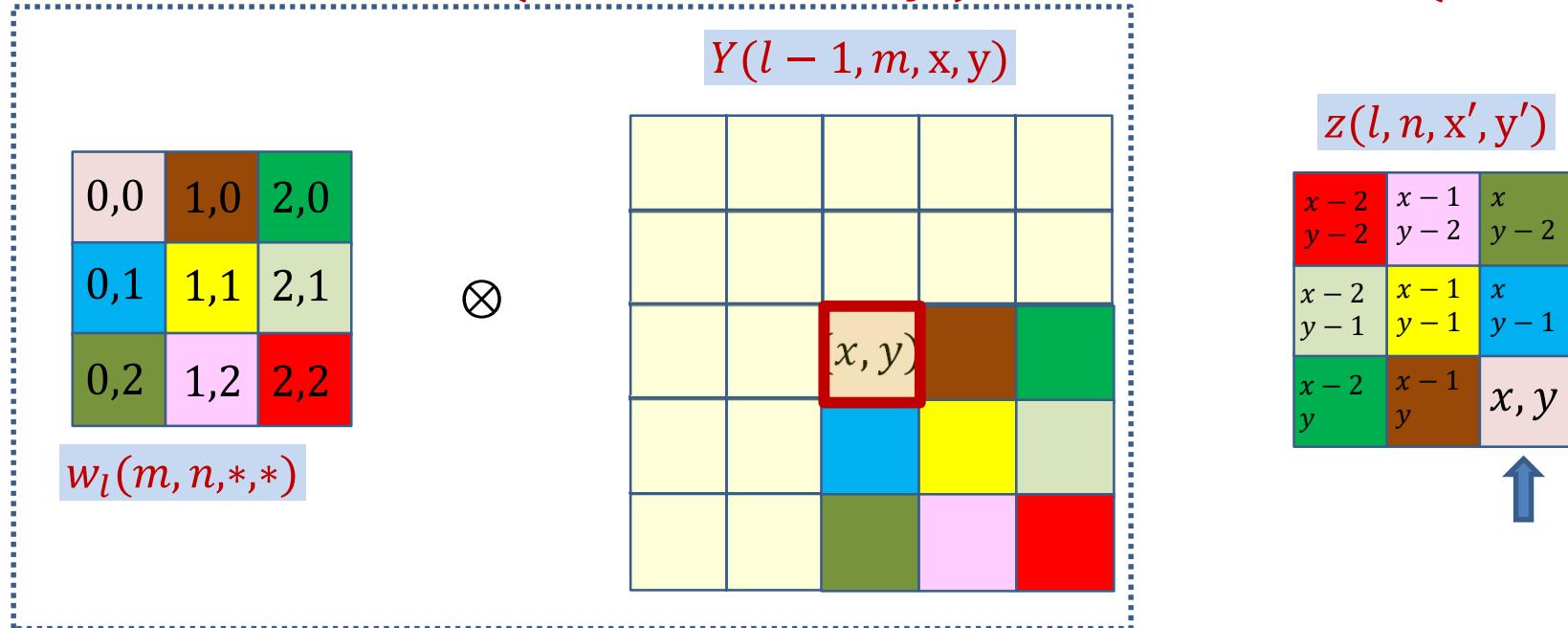
# How a single $Y(l - 1, m, x, y)$ influences $z(l, n, x', y')$



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

- **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 $z(l, n, x', y')$

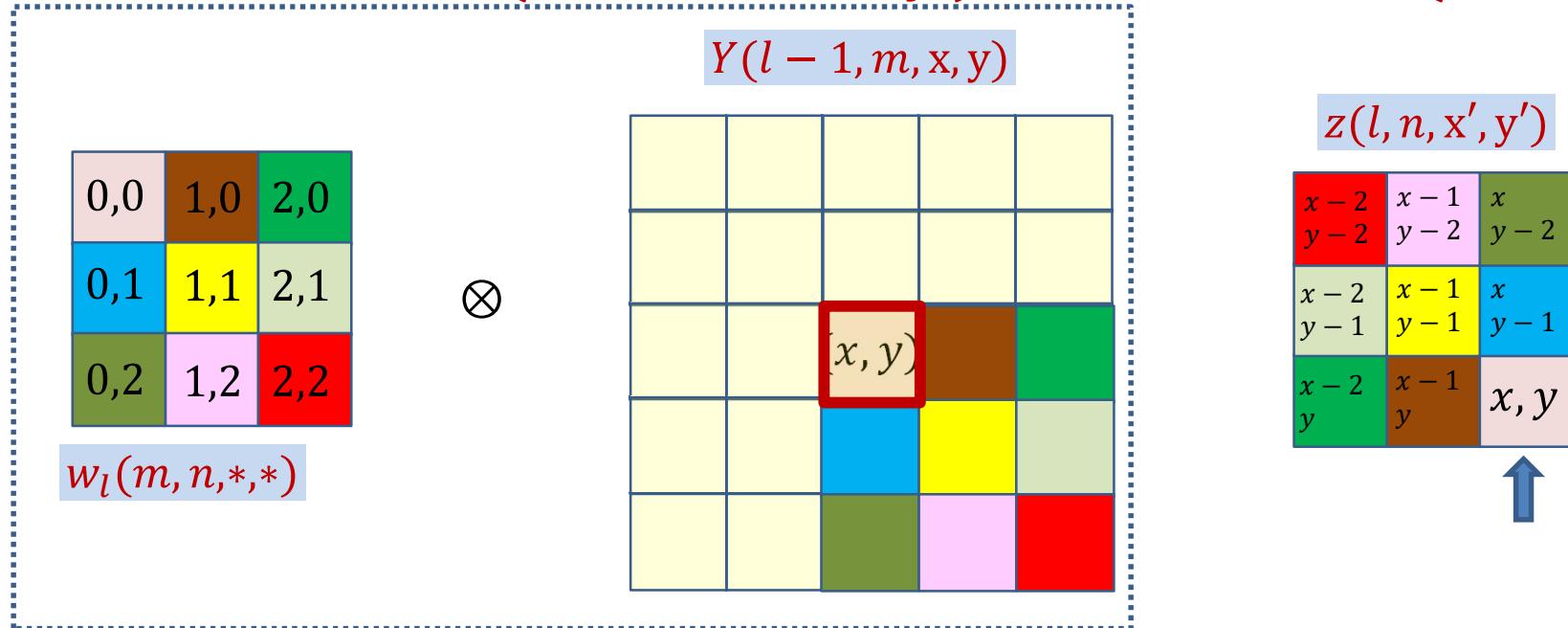


$$z(l, n, x', y') += Y(l - 1, m, x, y) w_l(m, n, x - x', y - y')$$

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

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

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

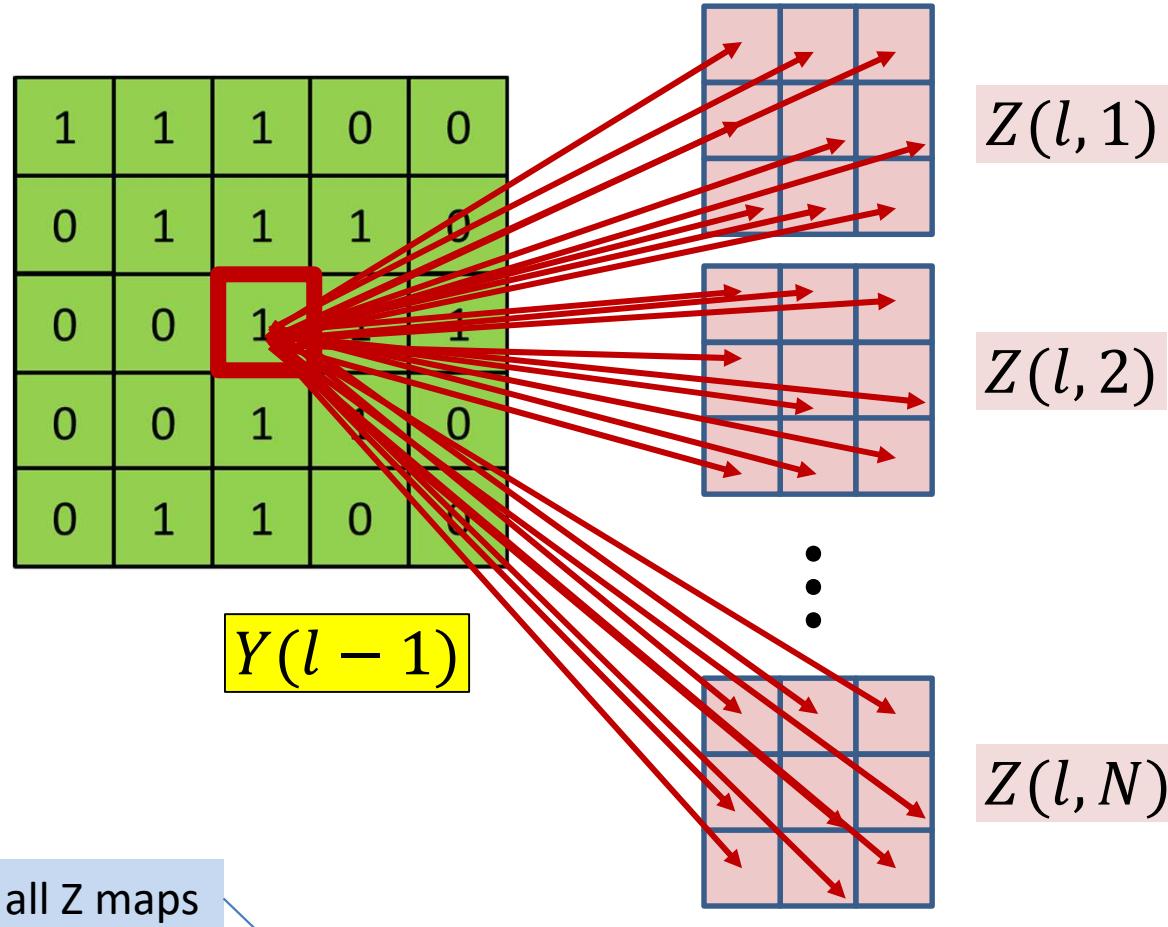


$$z(l, n, x', y') += Y(l - 1, m, x, y) w_l(m, n, x - x', y - y')$$

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

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

# BP: Convolutional layer



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

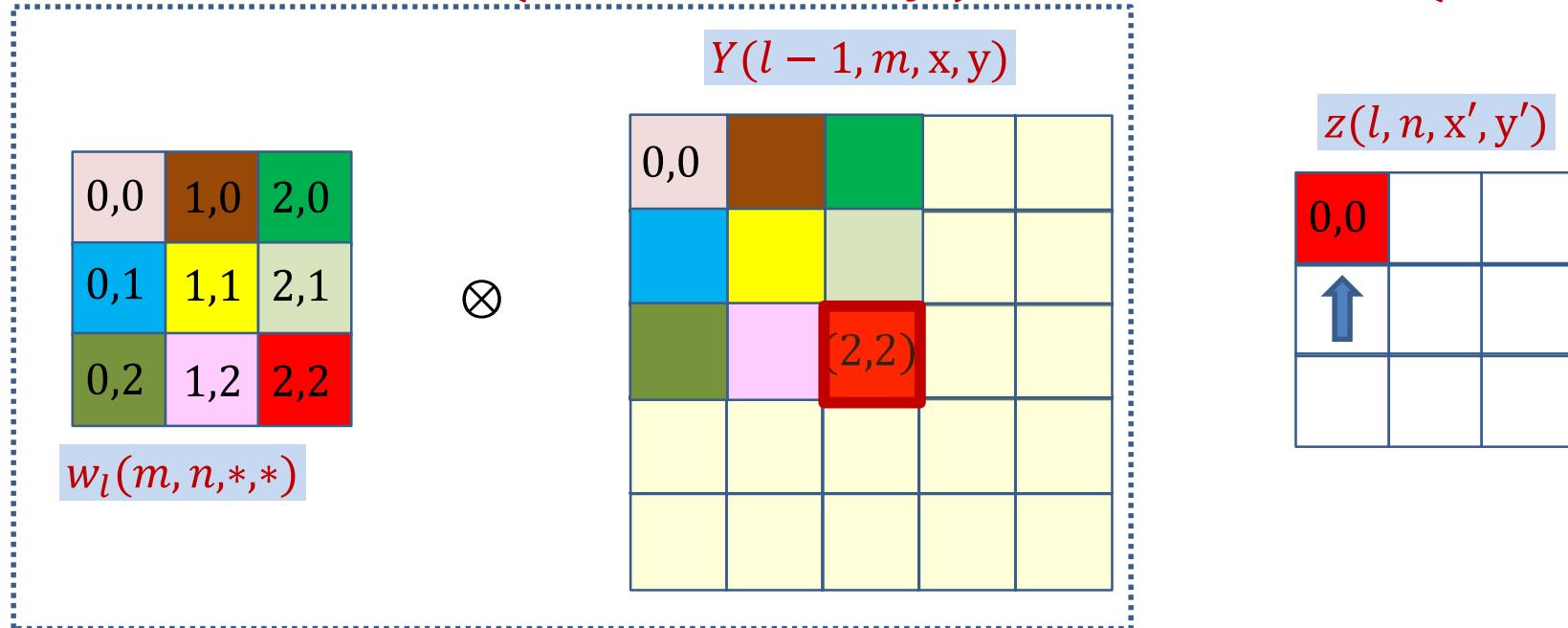
# Computing derivative for $Y(l - 1, m, *, *)$

- 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

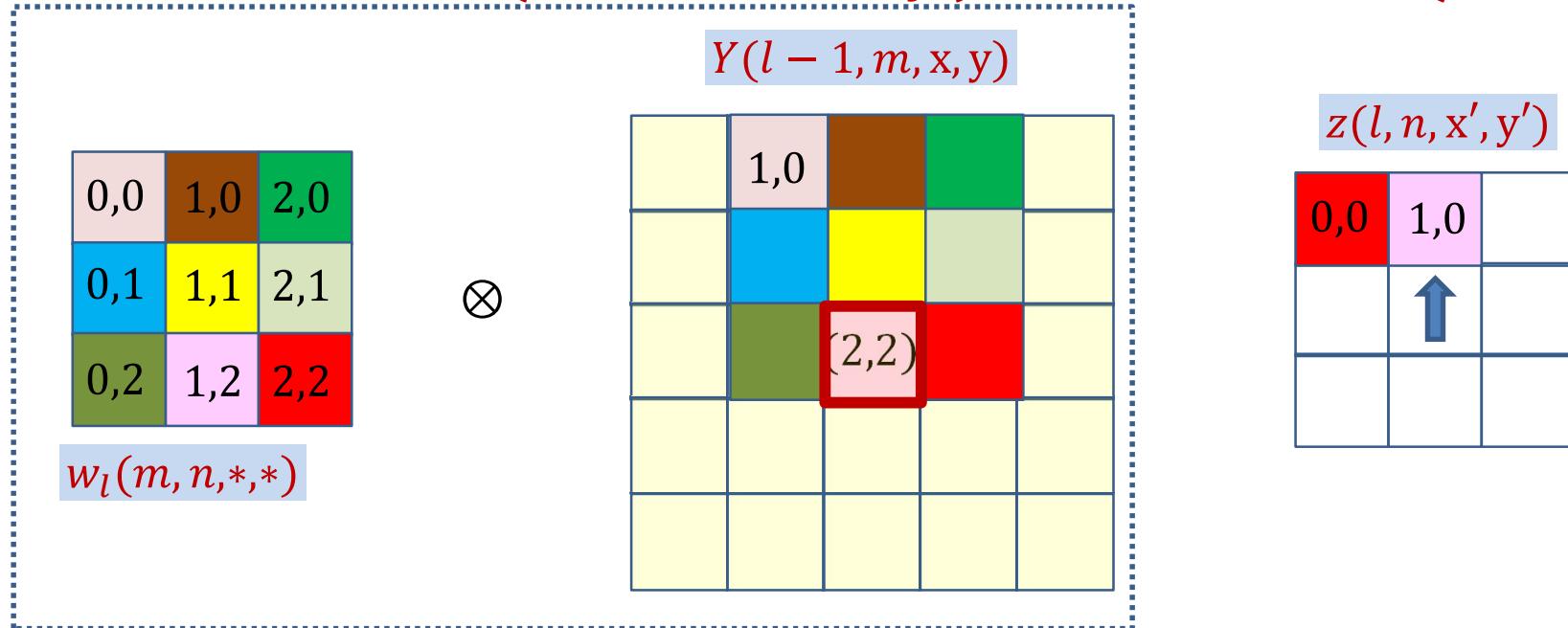
# How a single $Y(l - 1, m, x, y)$ influences $z(l, n, x', y')$



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

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

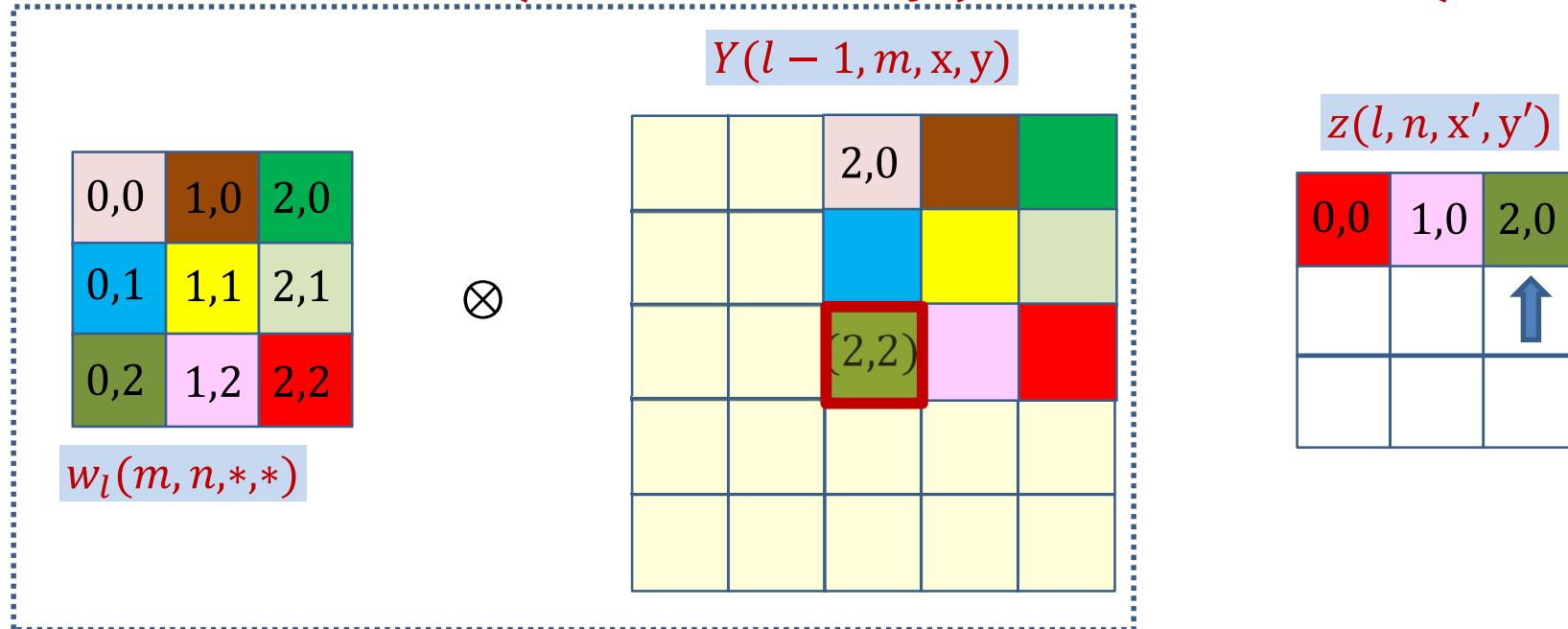
# How a single $Y(l - 1, m, x, y)$ influences $z(l, n, x', y')$



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

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

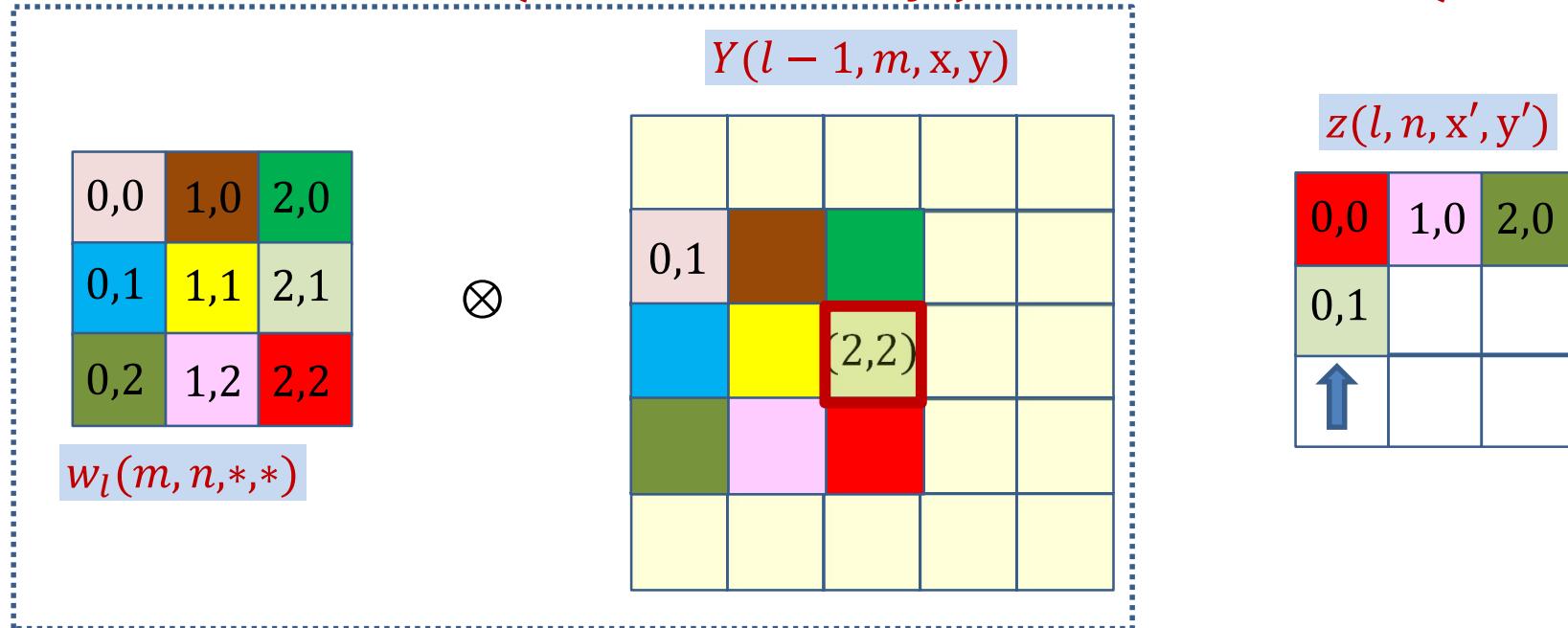
# How a single $Y(l - 1, m, x, y)$ influences $z(l, n, x', y')$



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

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

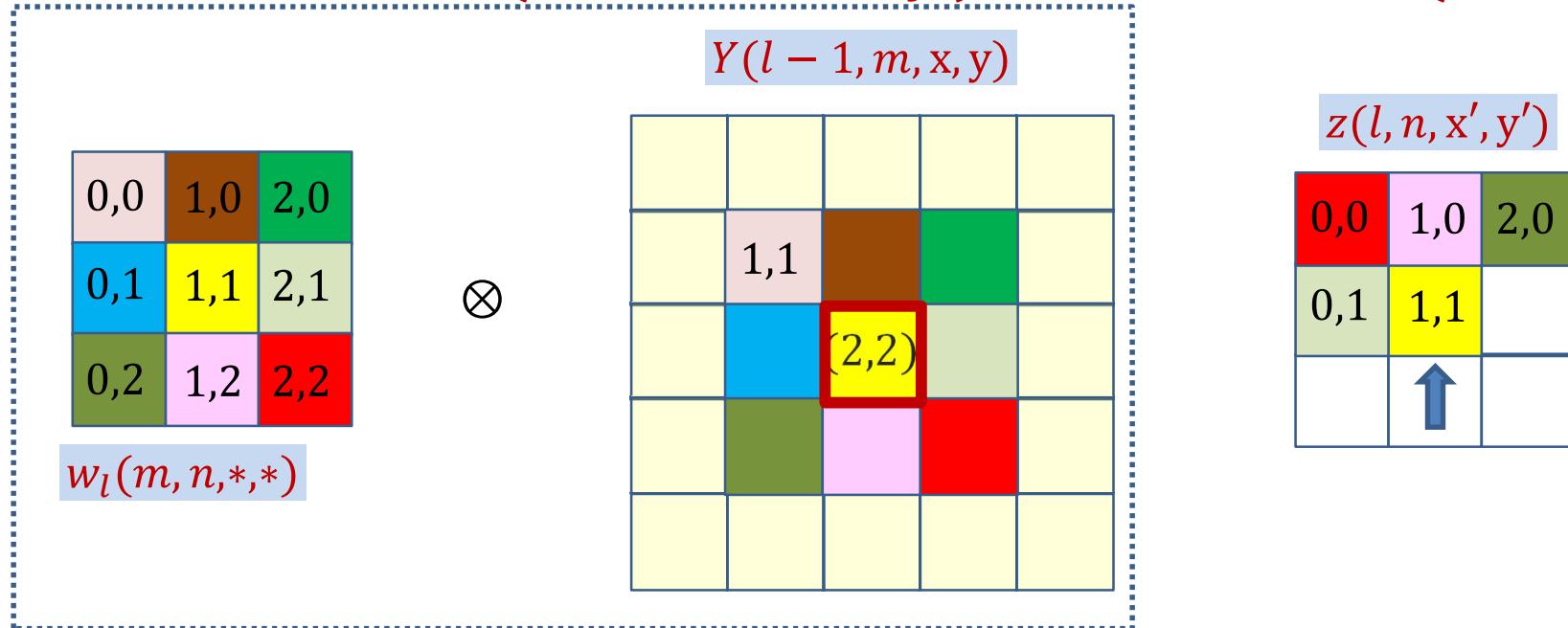
# How a single $Y(l - 1, m, x, y)$ influences $z(l, n, x', y')$



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

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

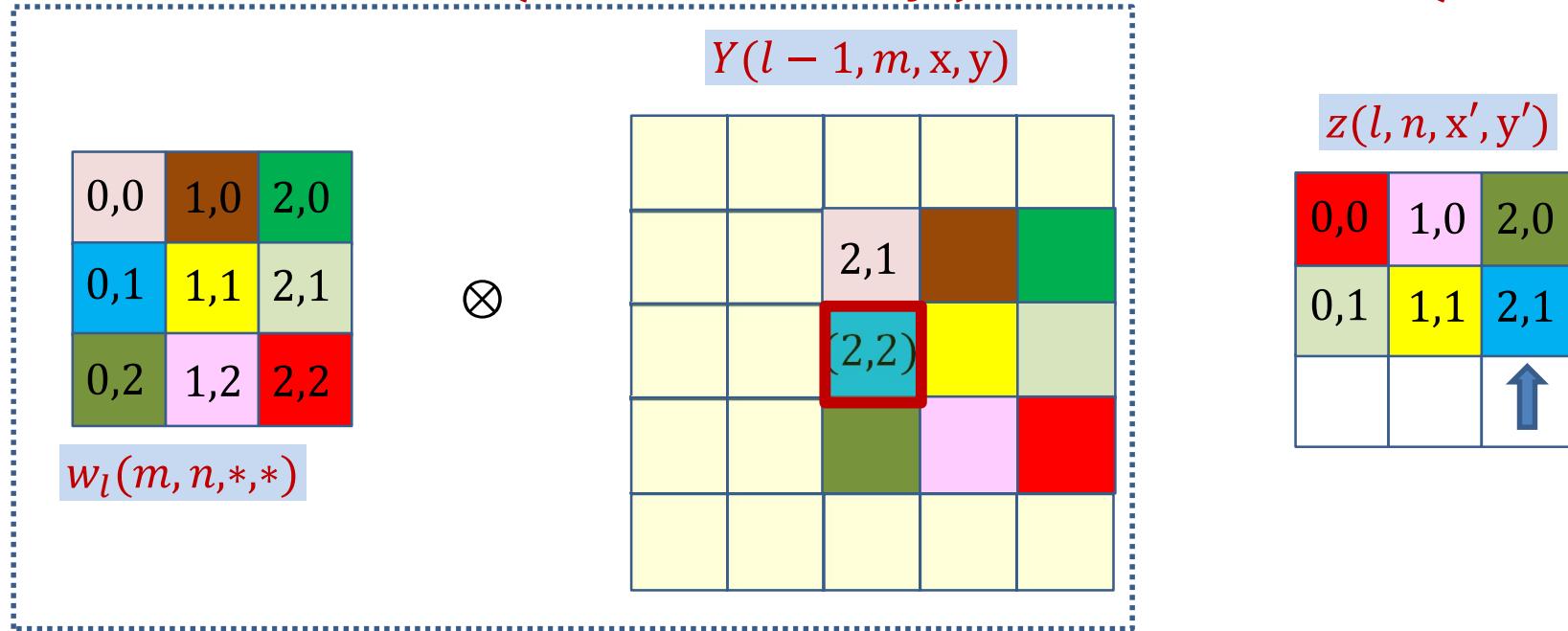
# How a single $Y(l - 1, m, x, y)$ influences $z(l, n, x', y')$



$$z(l, n, 1,1) += Y(l - 1, m, 2,2) w_l(m, n, 1,1)$$

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

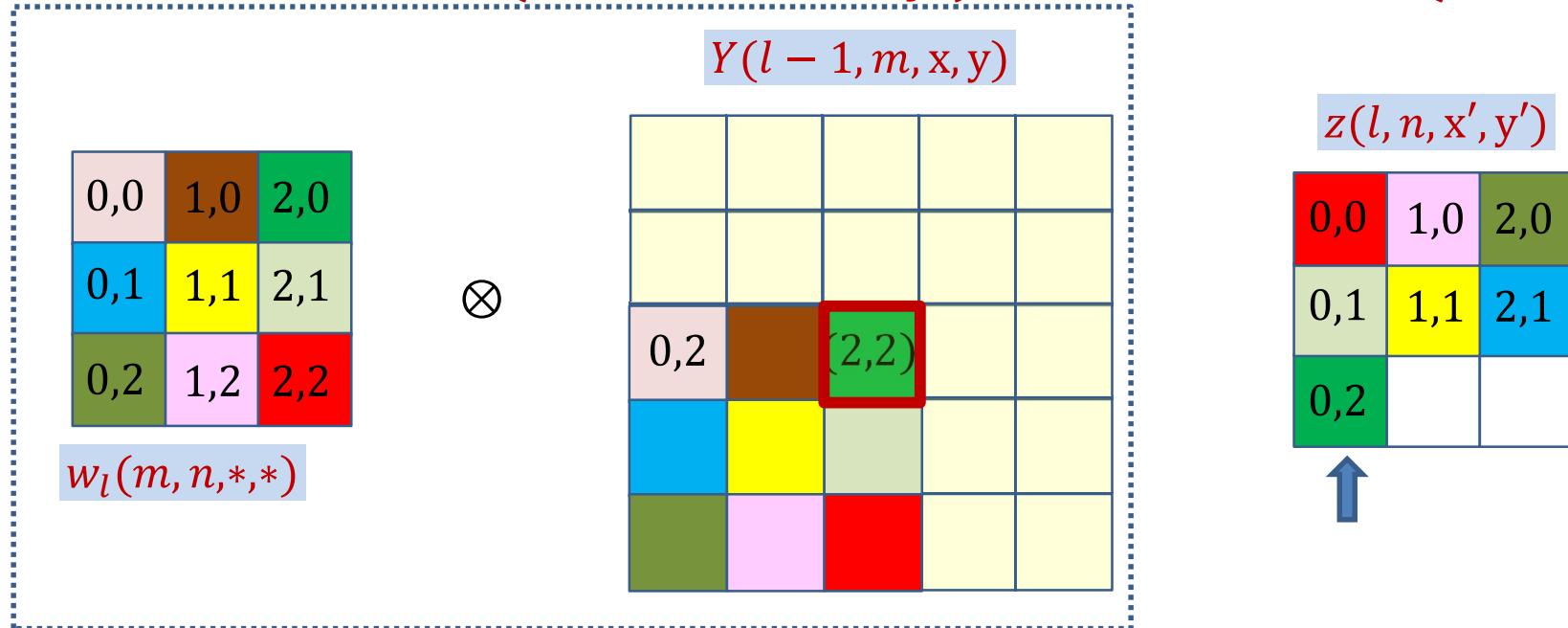
# How a single $Y(l - 1, m, x, y)$ influences $z(l, n, x', y')$



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

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

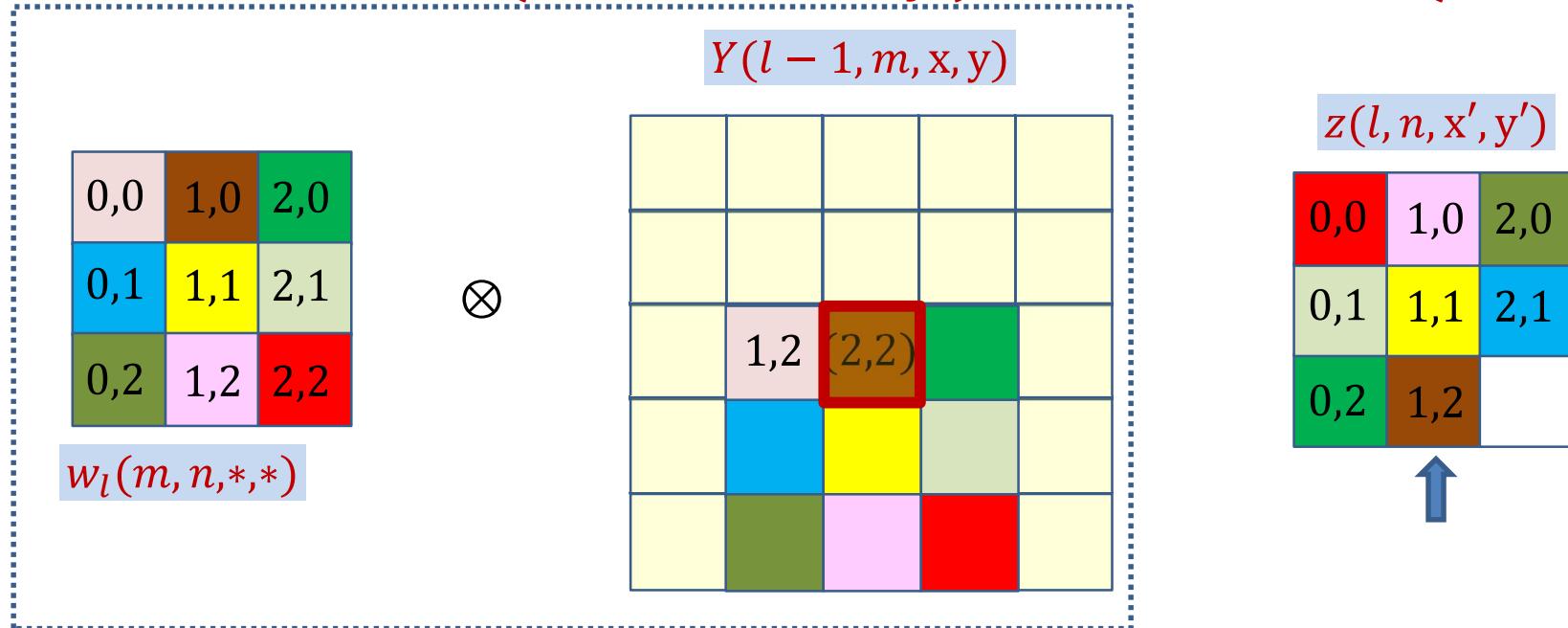
# How a single $Y(l - 1, m, x, y)$ influences $z(l, n, x', y')$



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

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

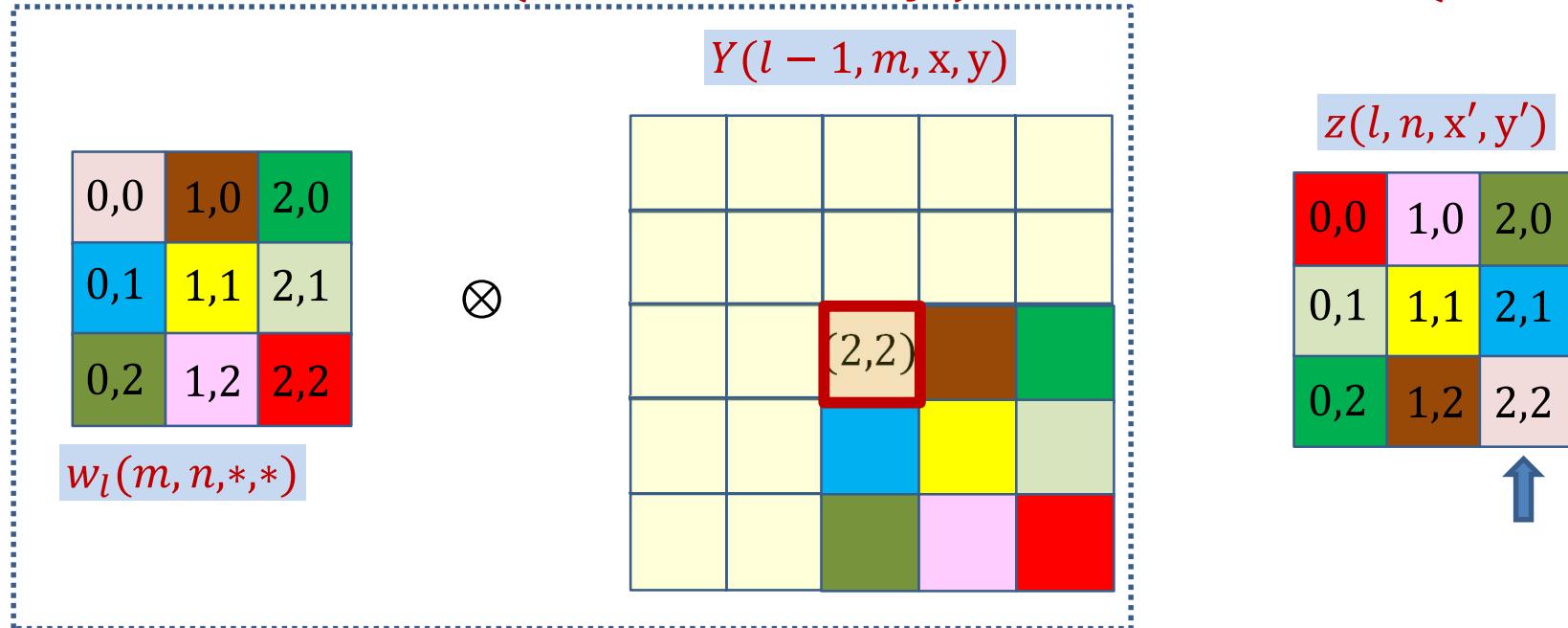
# How a single $Y(l - 1, m, x, y)$ influences $z(l, n, x', y')$



$$z(l, n, 1,2) += Y(l - 1, m, 2,2) w_l(m, n, 2,1)$$

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

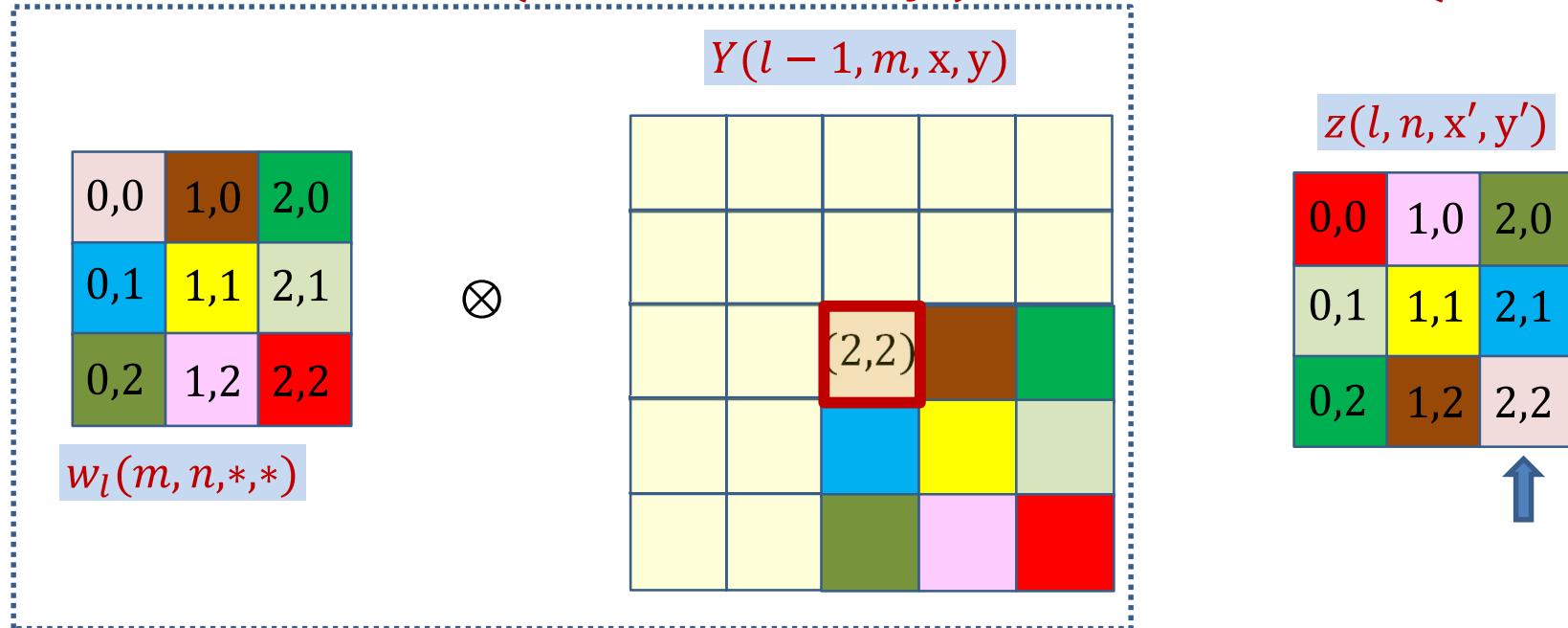
# How a single $Y(l - 1, m, x, y)$ influences $z(l, n, x', y')$



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

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

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

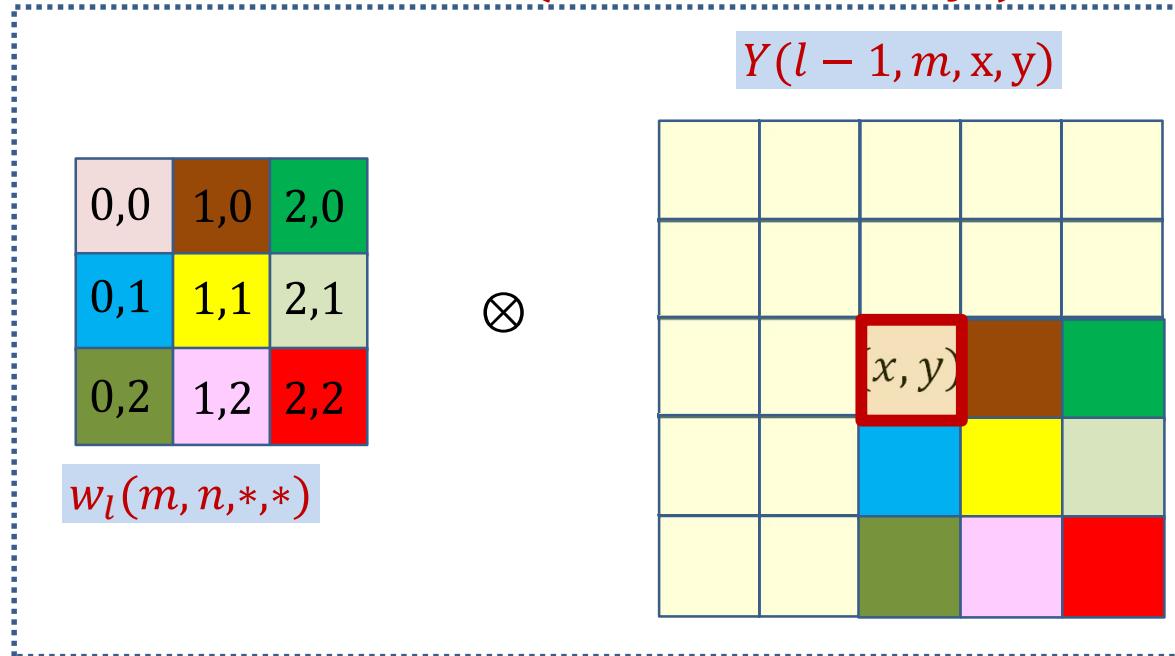


$$z(l, n, x', y') += Y(l - 1, m, 2, 2) w_l(m, n, 2 - x', 2 - y')$$

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

- The derivative at  $Y(l - 1, m, 2, 2)$  is the sum of component-wise product of the filter elements and the elements of the derivative at  $z(l, m, \dots)$

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

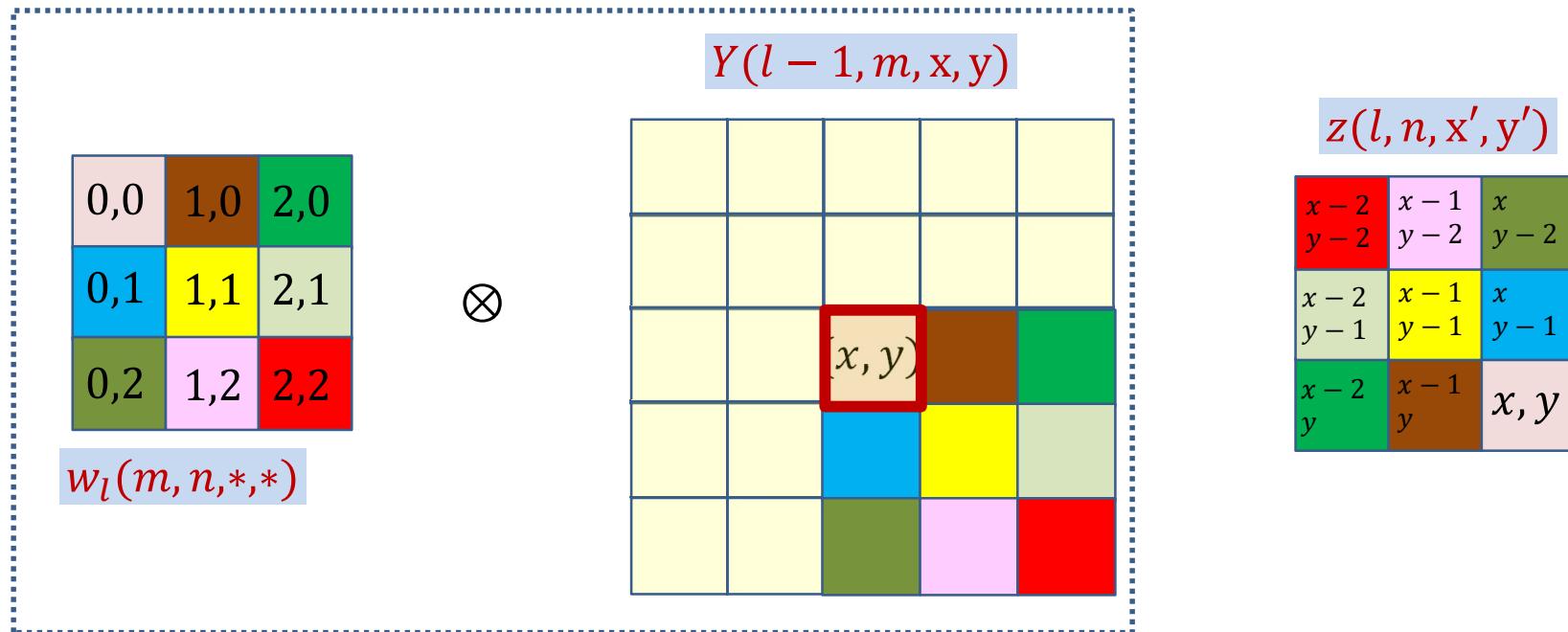


$$z(l, n, x', y') += Y(l - 1, m, x, y) w_l(m, n, x - x', y - y')$$

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

- The derivative at  $Y(l - 1, m, x, y)$  is the sum of component-wise product of the filter elements and the elements of the derivative at  $z(l, n, x', y')$ .

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



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

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

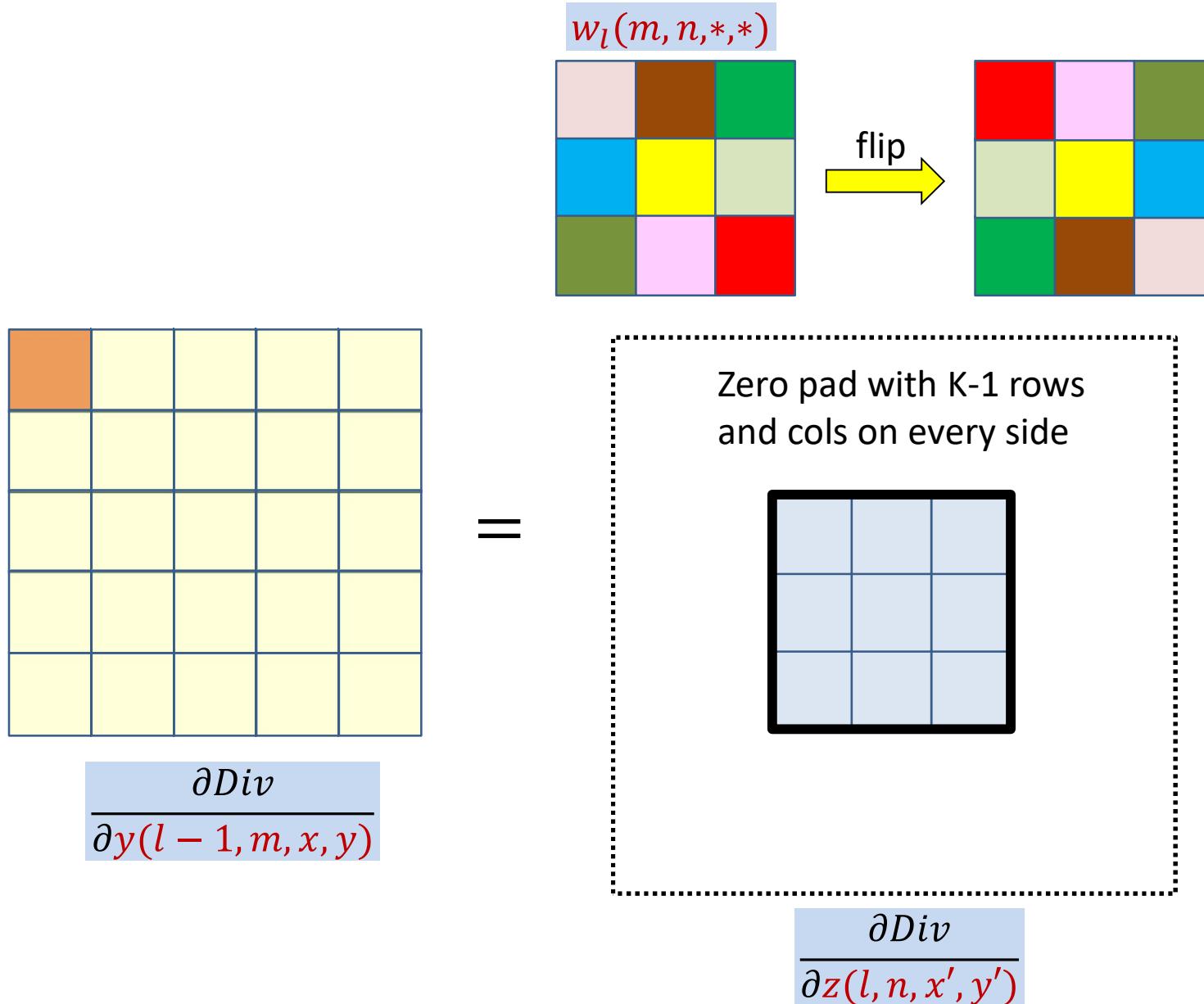
Contribution of the entire  $n$ th affine map  $z(l, n, *, *)$

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

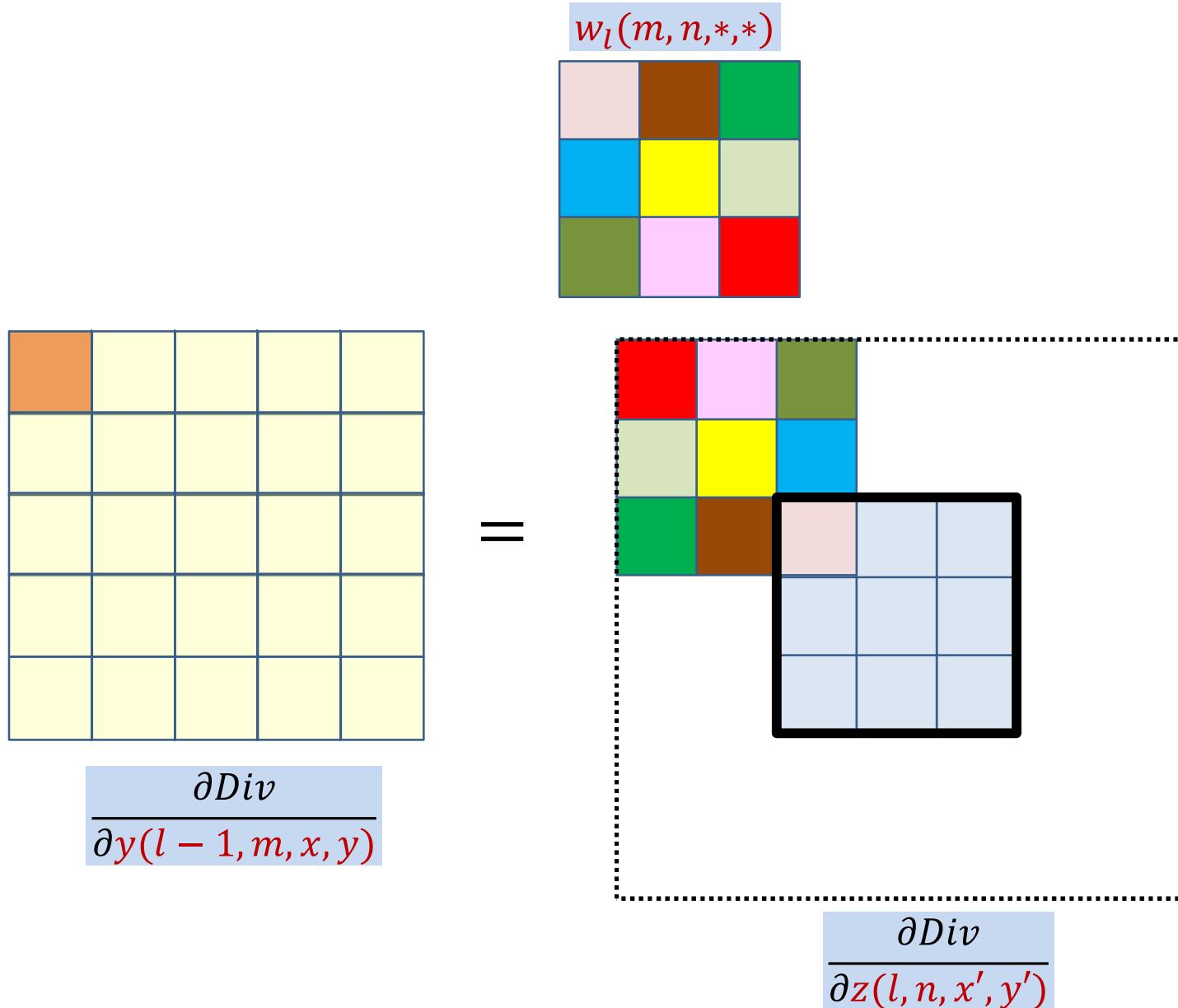
$$\frac{\partial \text{Div}}{\partial y(l - 1, m, x, y)} = w_l(m, n, *, *) \cdot \frac{\partial \text{Div}}{\partial z(l, n, x', y')}$$

The diagram illustrates the computation of a derivative from a single input map. On the left, a large 5x5 grid represents the input map  $Y(l - 1, m)$ . A single orange cell in the top-left corner is highlighted. To its right is an equals sign. Above the equals sign is a 3x3 kernel labeled  $w_l(m, n, *, *)$ , which is itself a 3x3 grid of colored cells: light pink, brown, green, blue, yellow, light green, olive green, pink, and red. To the right of the equals sign is a smaller 3x3 grid with a black border, representing the result of applying the kernel to the input map. The pink cell in the center of the kernel corresponds to the highlighted orange cell in the input map.

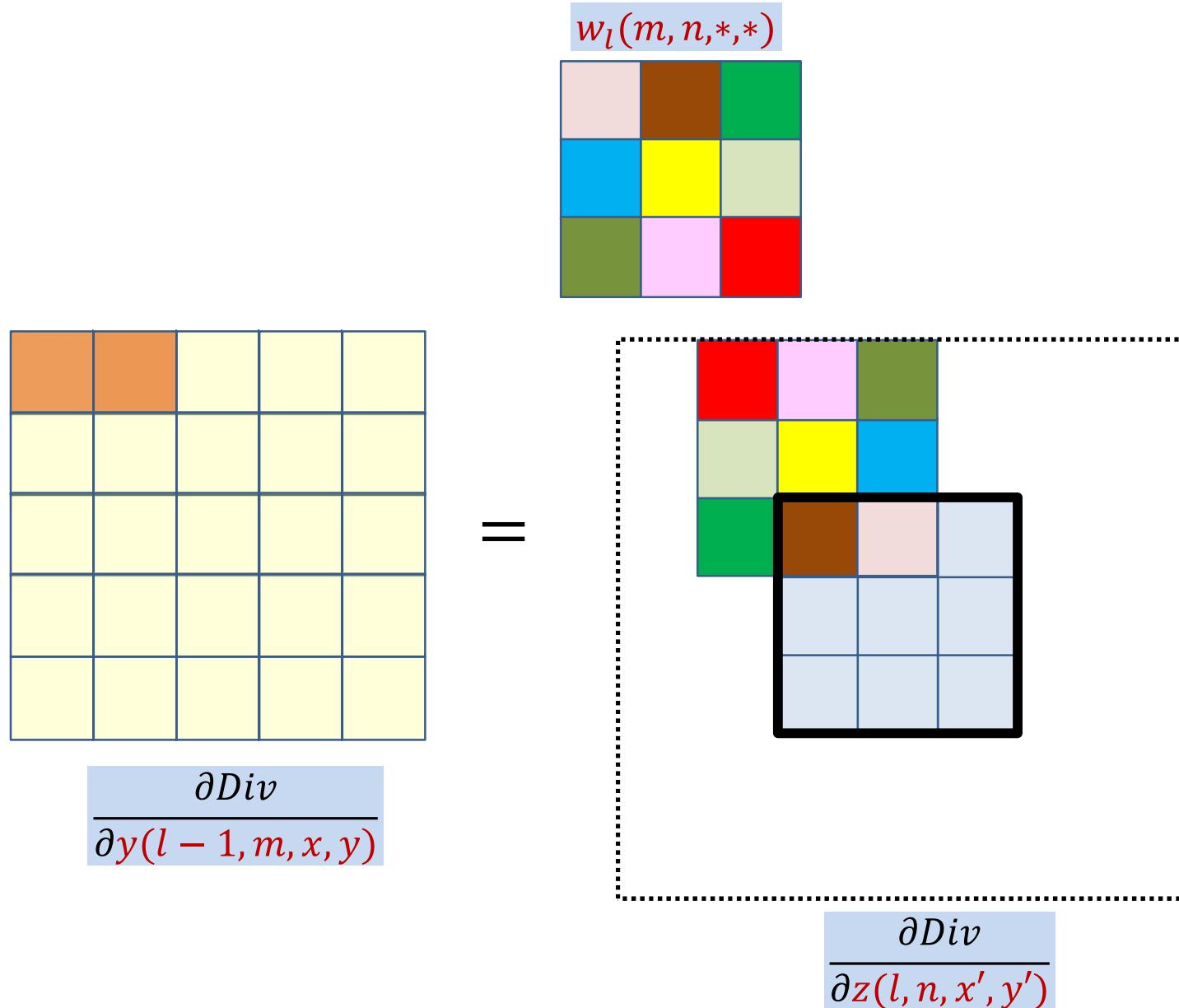
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



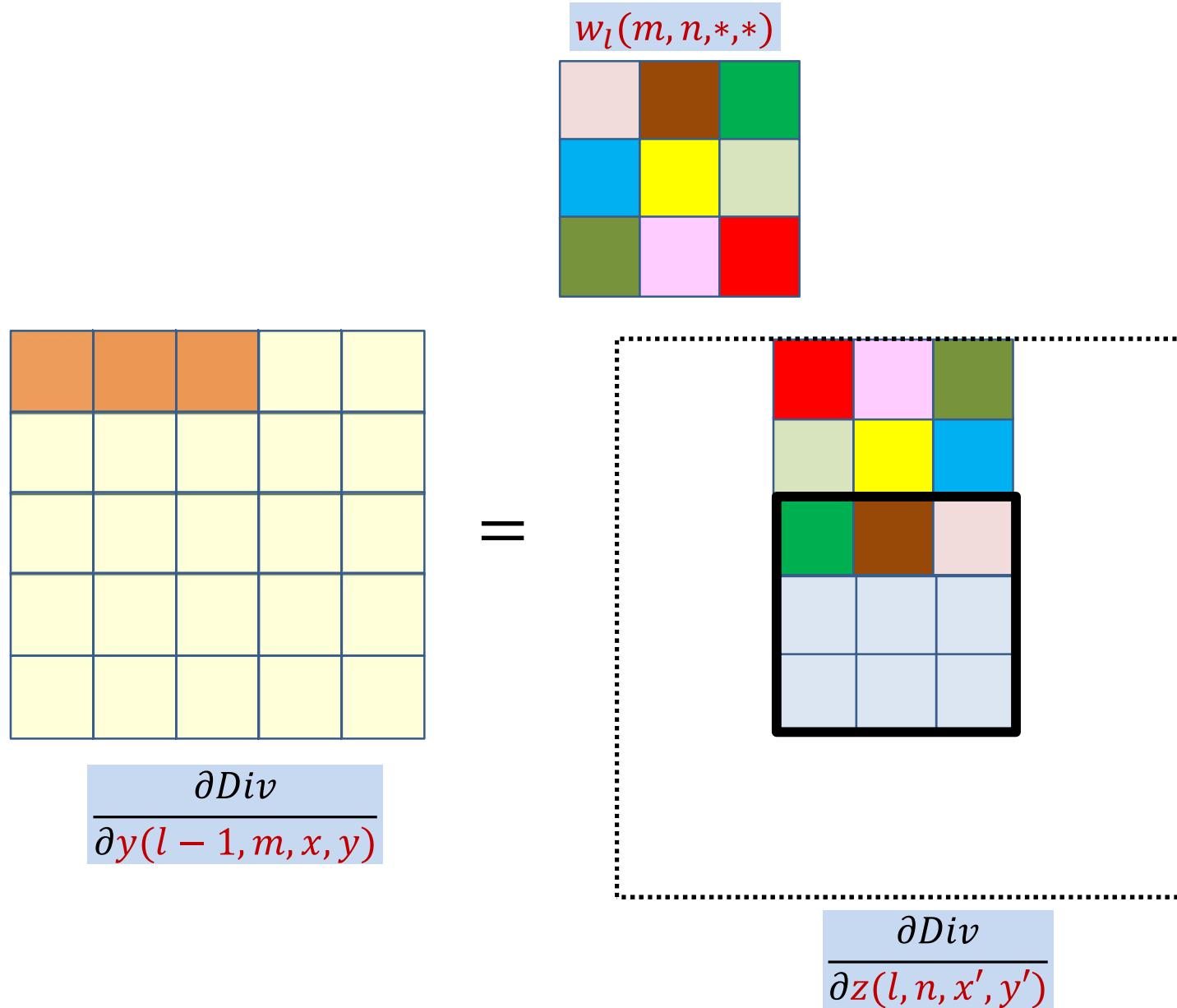
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



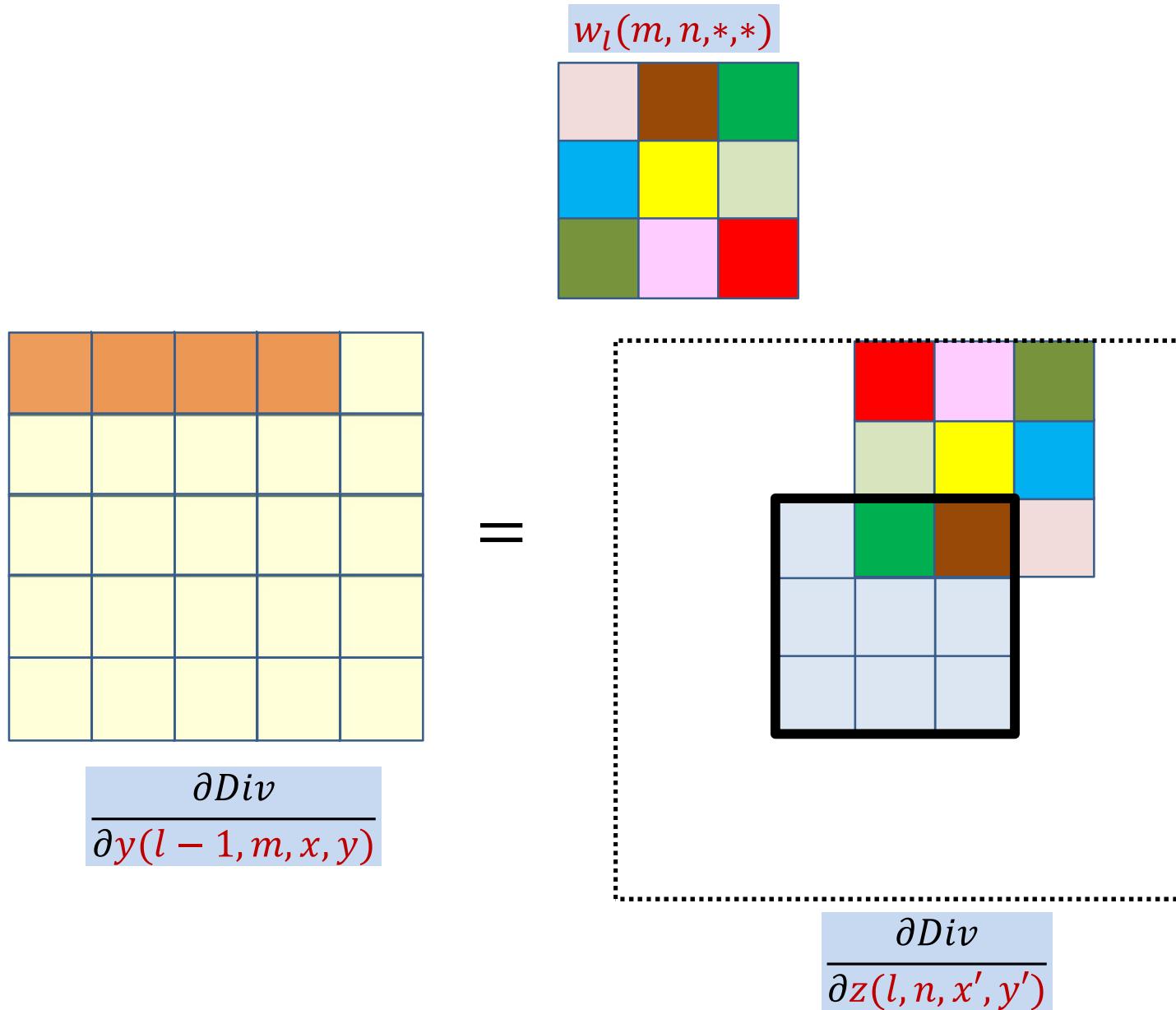
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



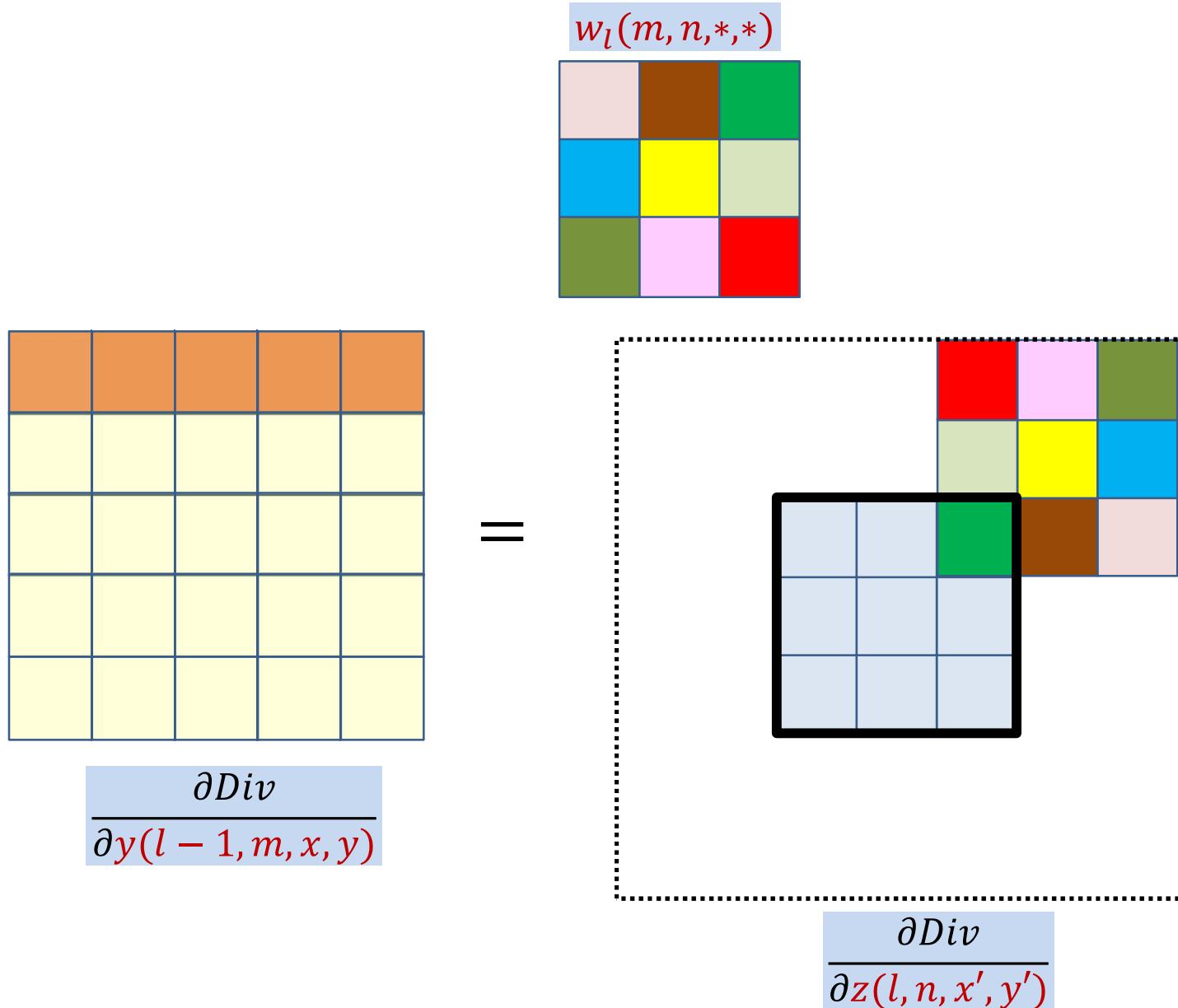
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



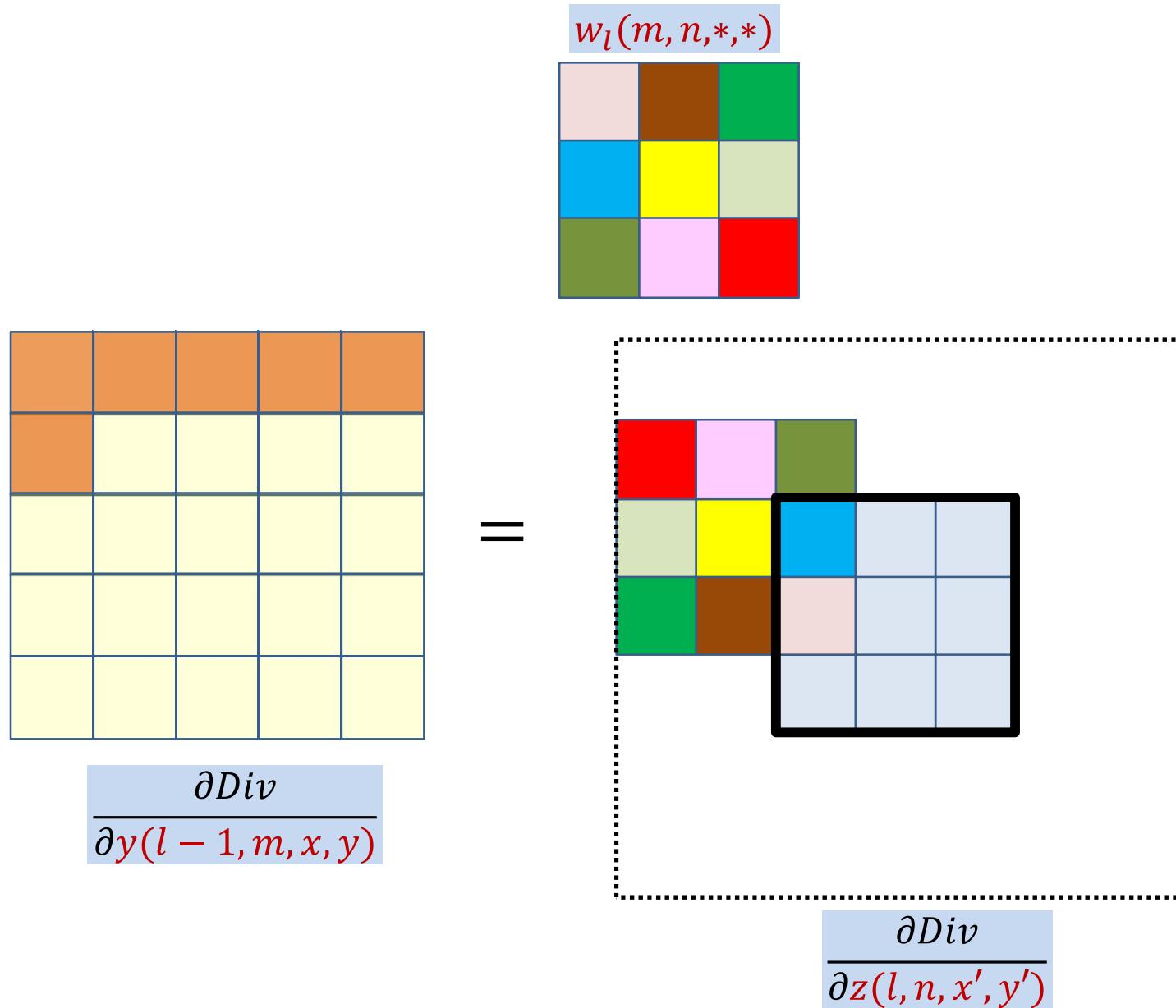
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



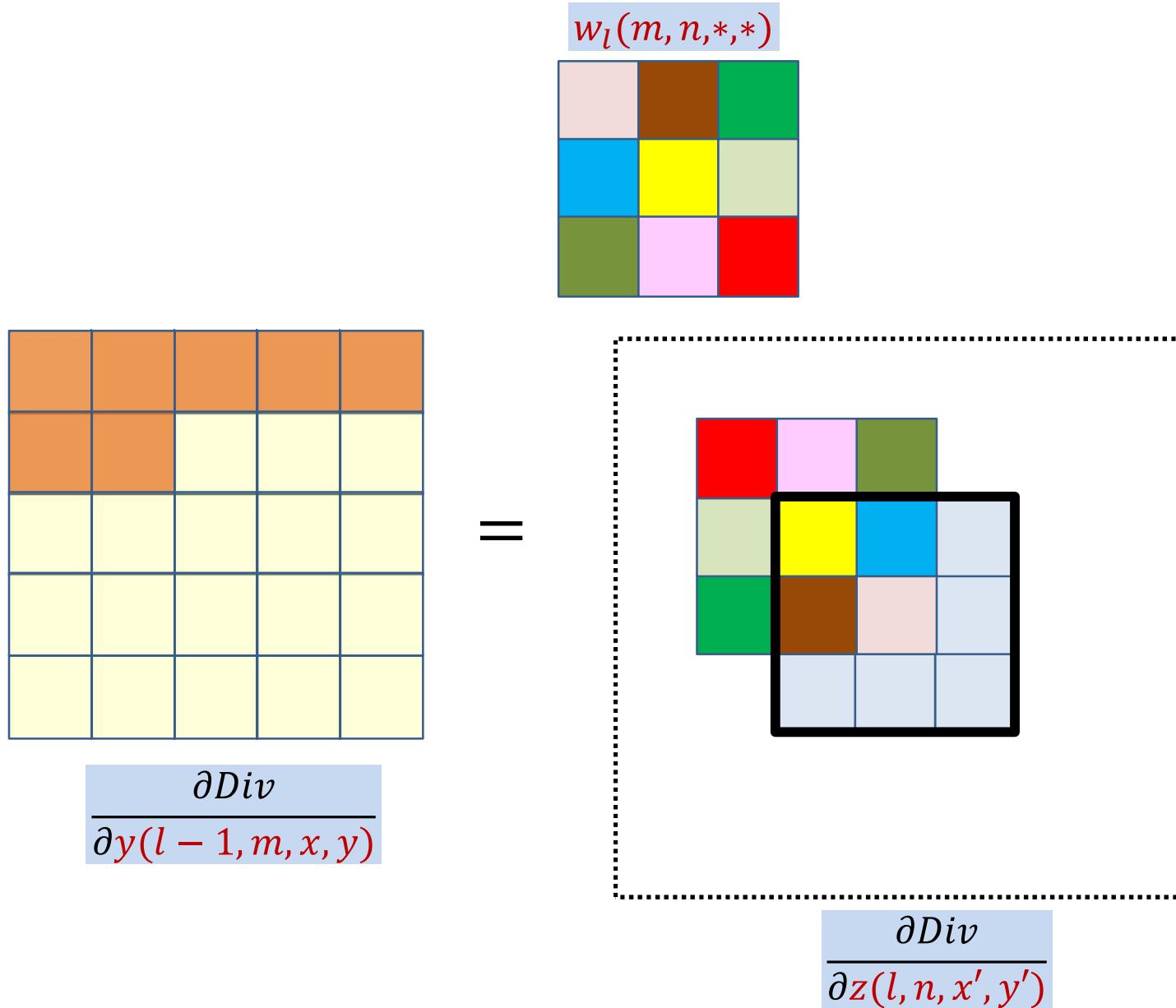
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



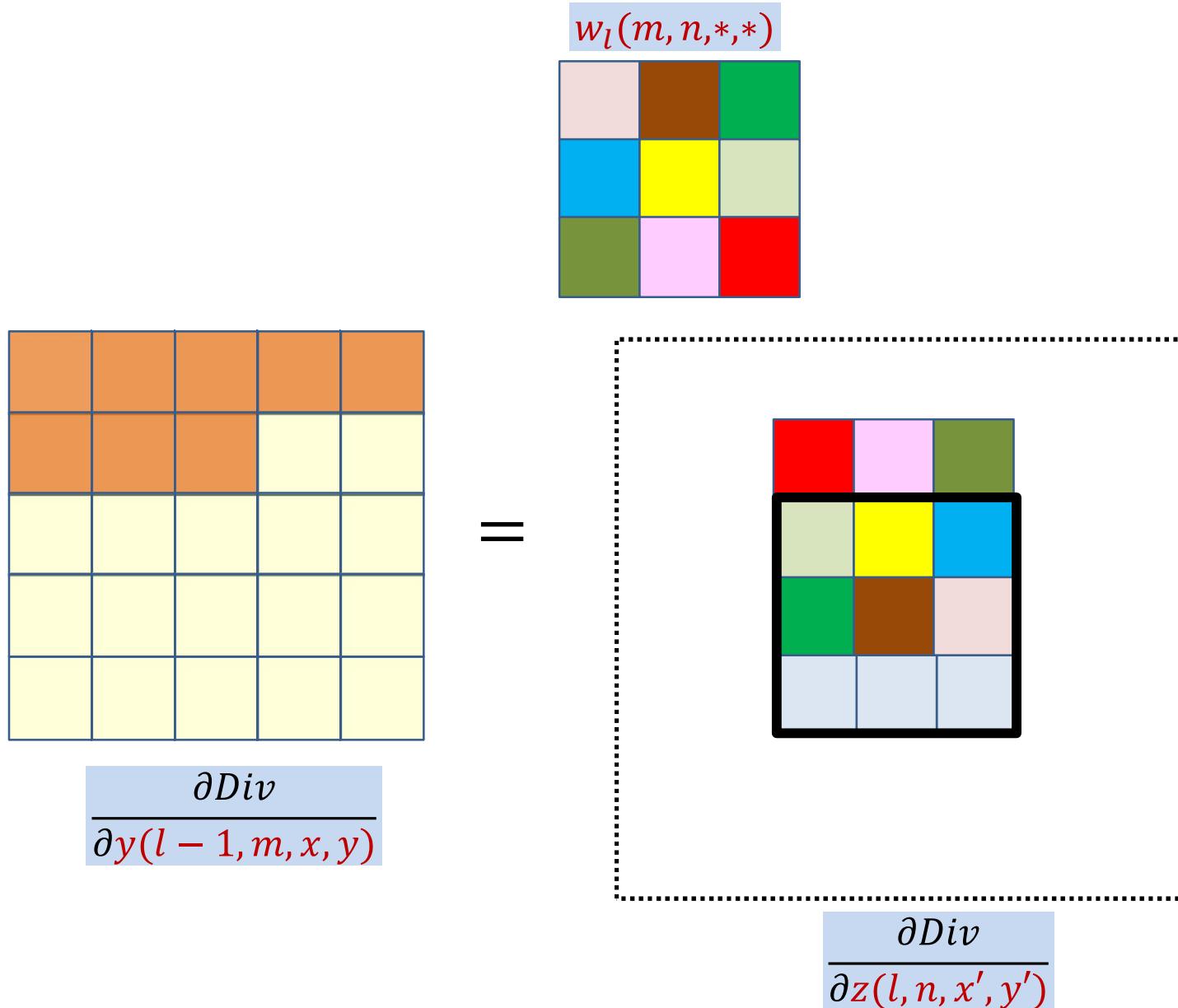
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



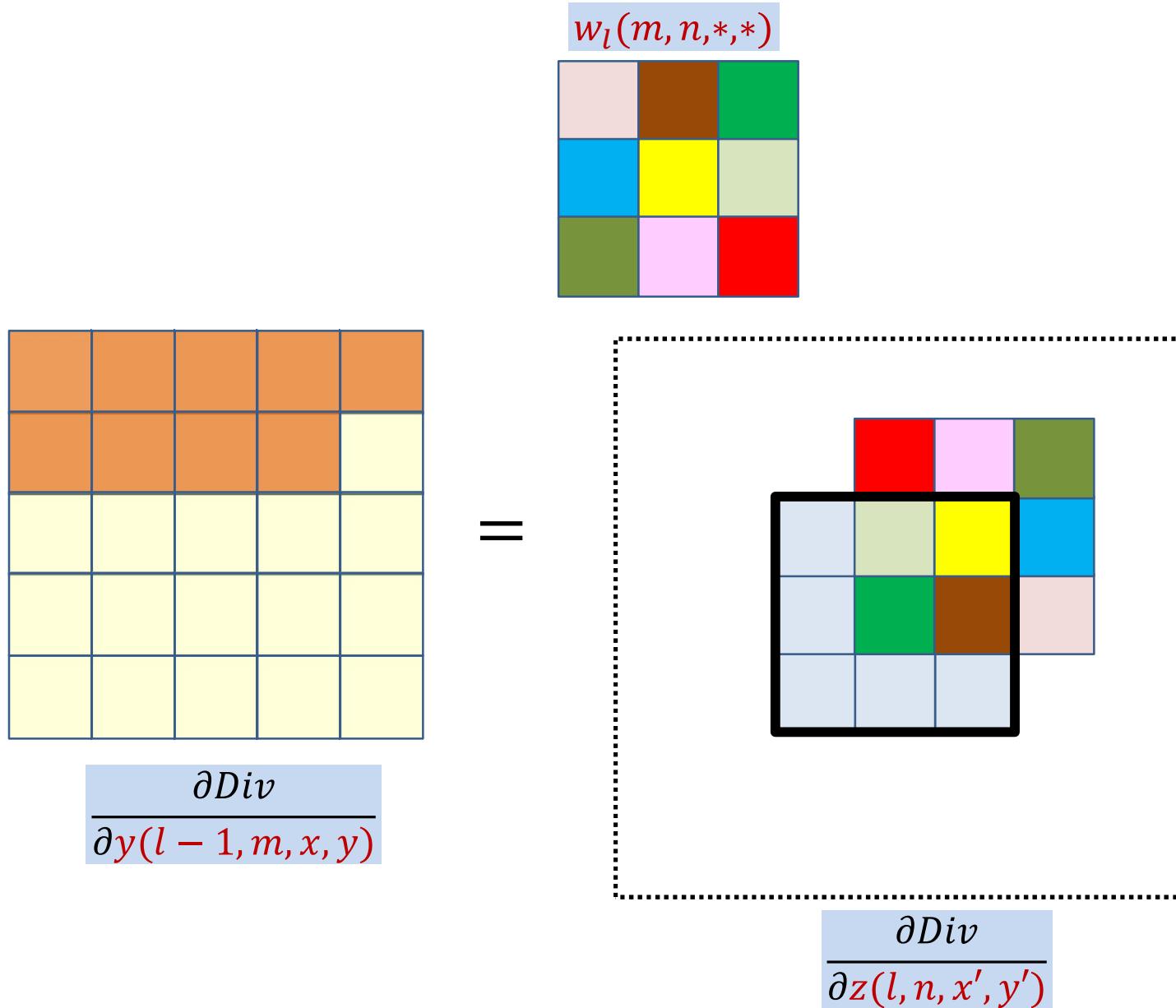
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



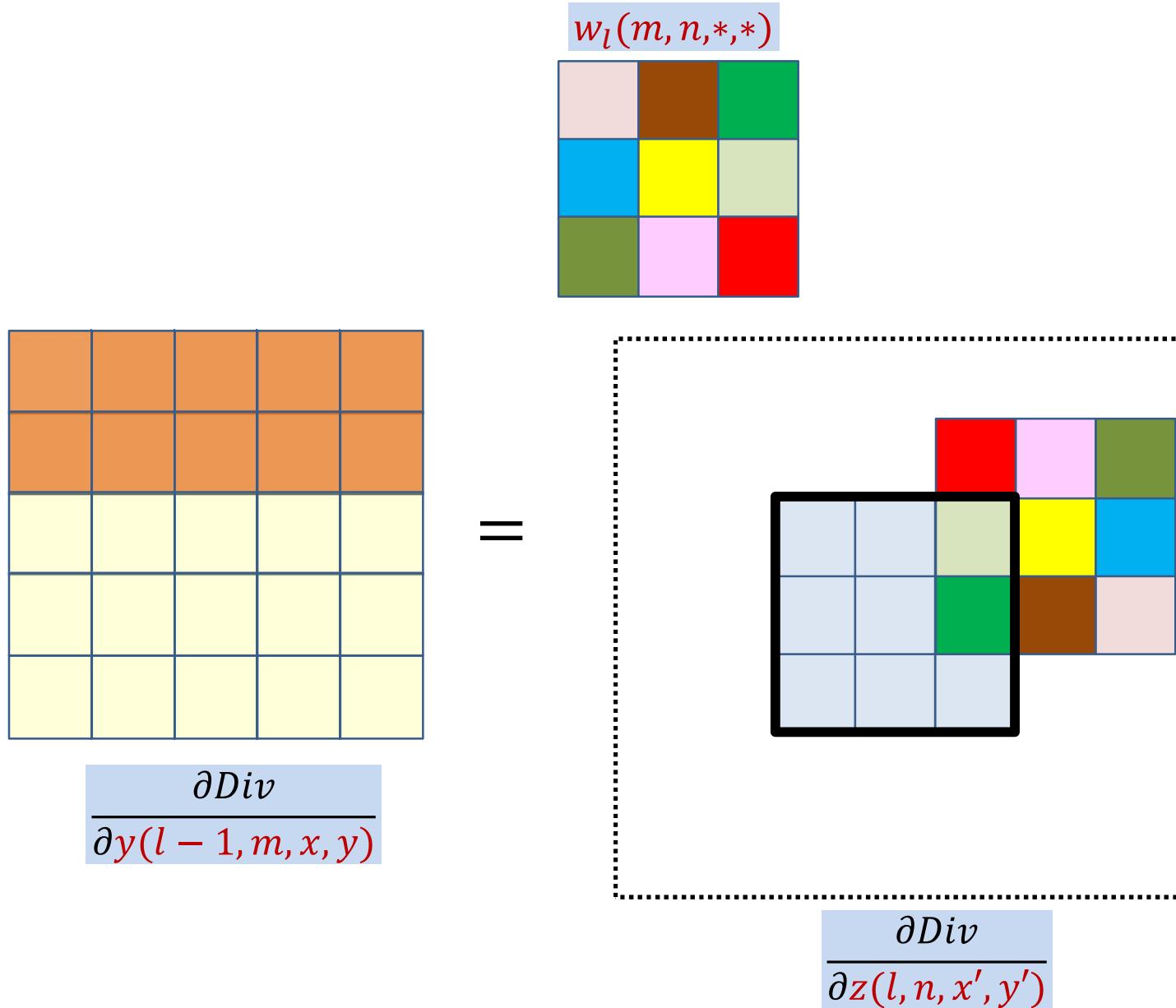
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



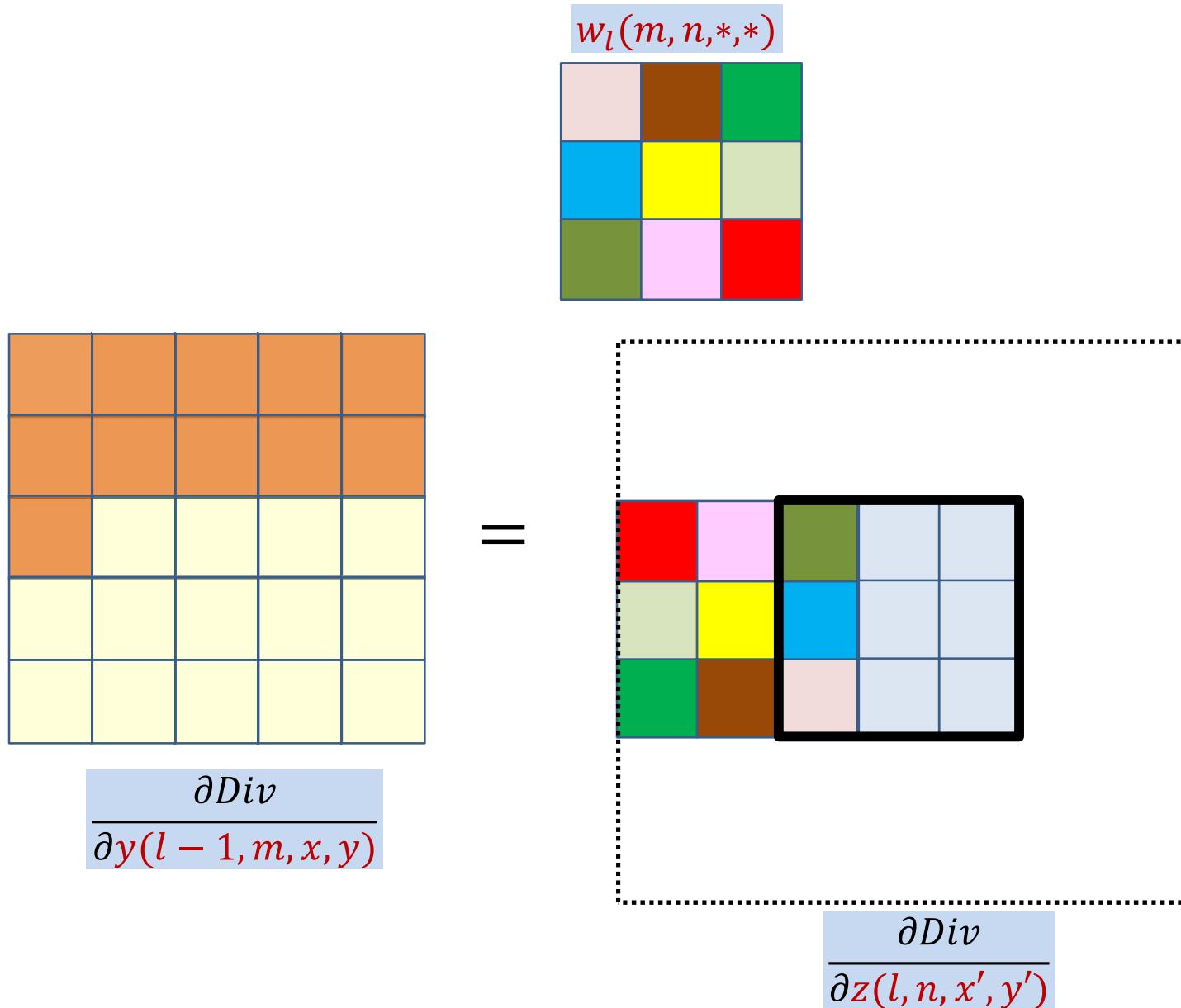
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



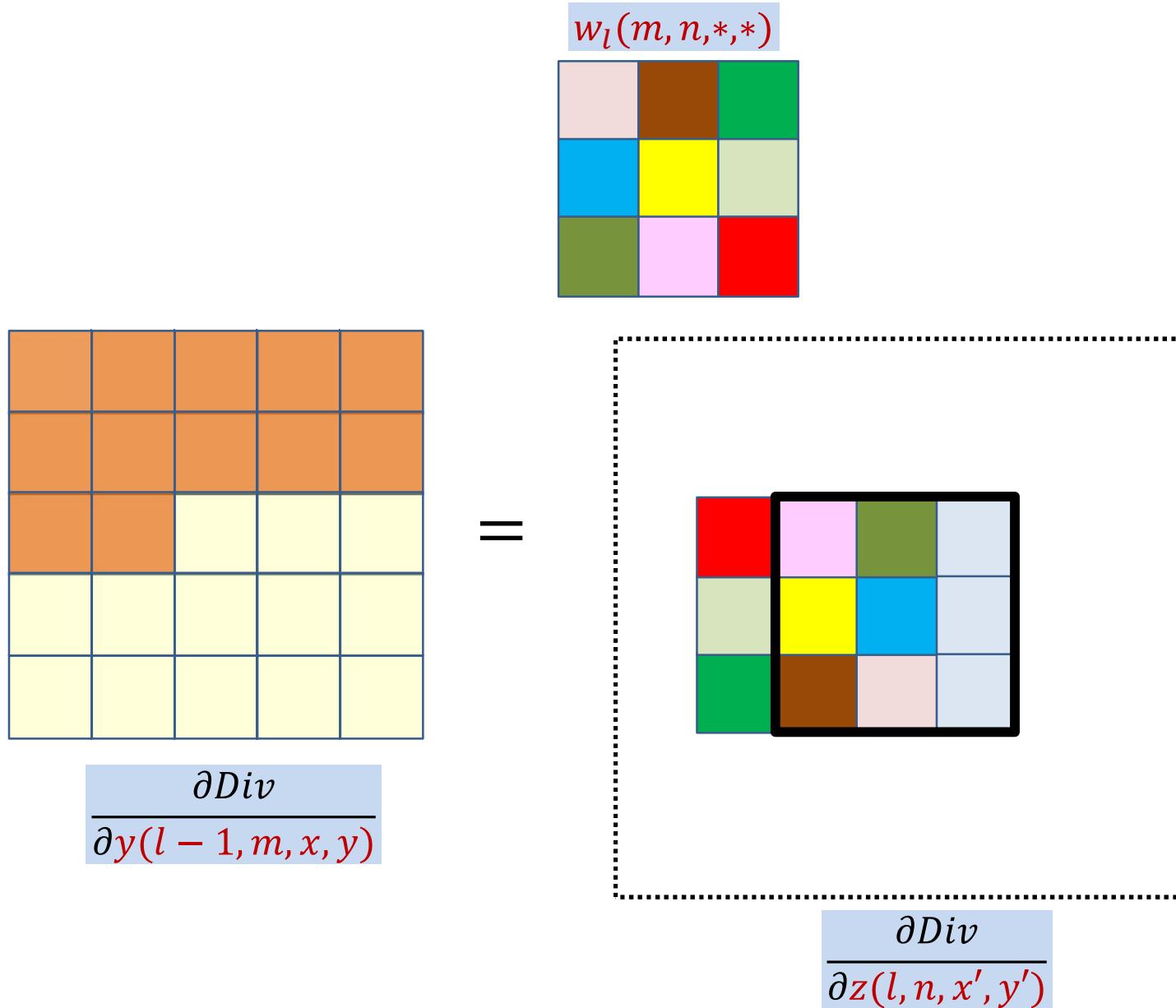
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



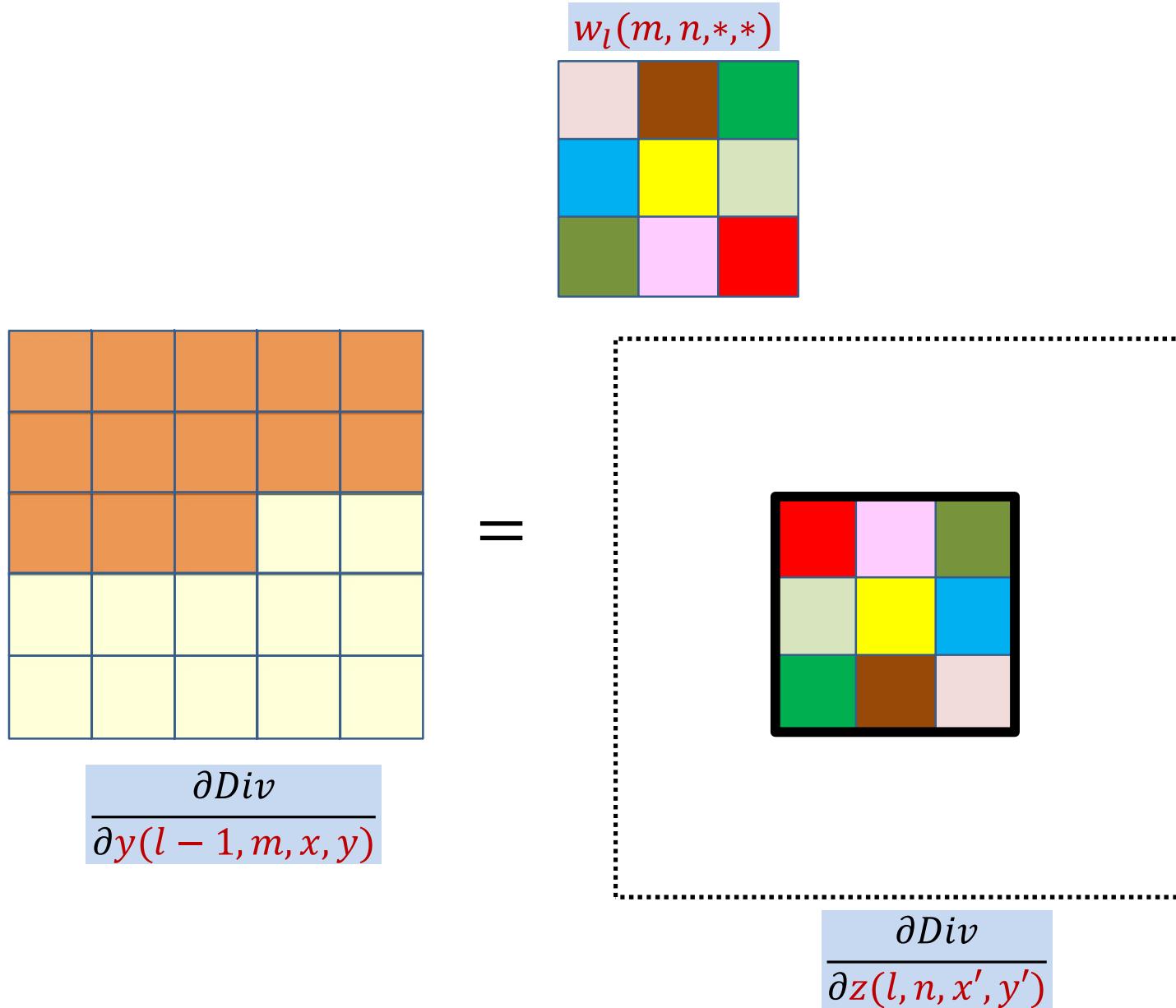
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



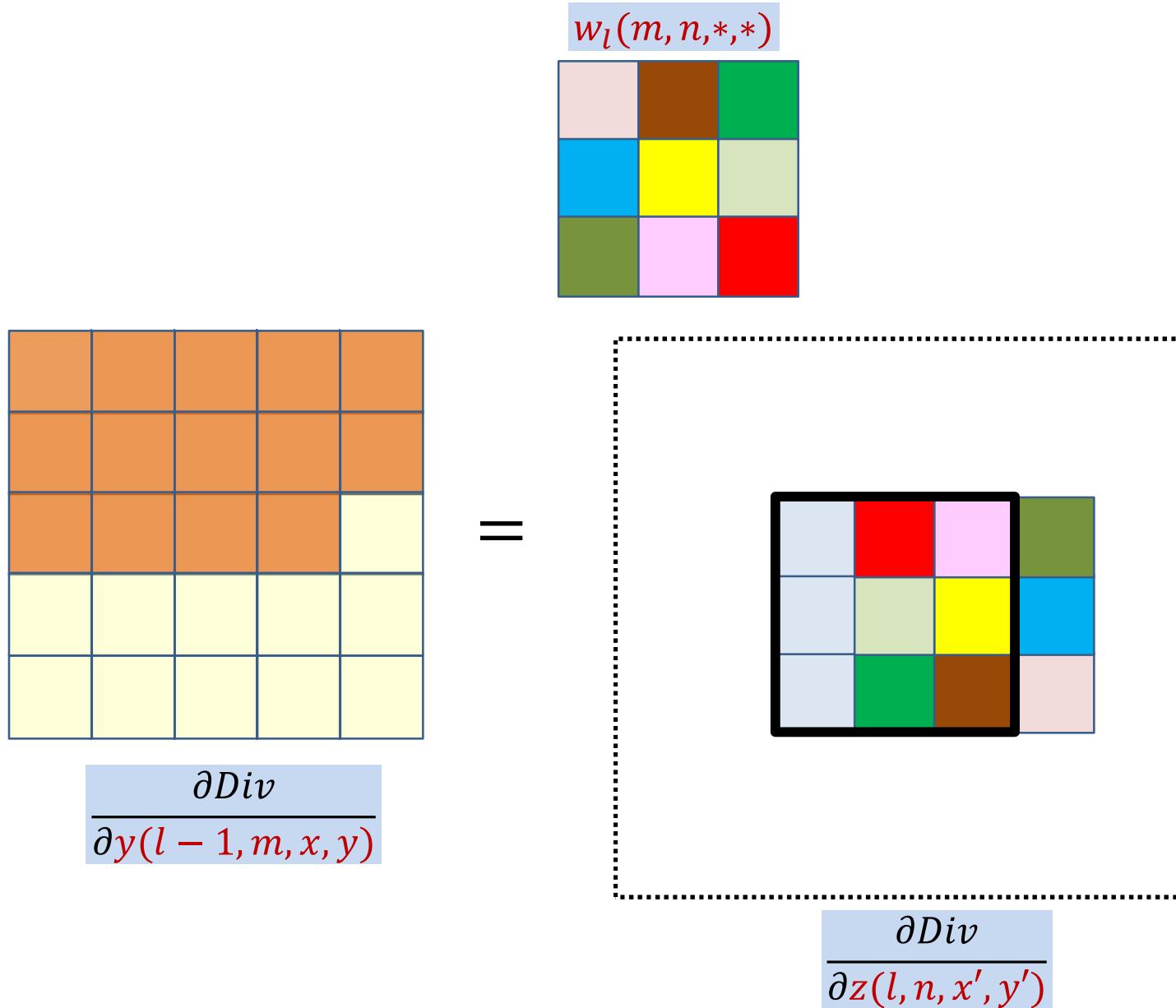
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



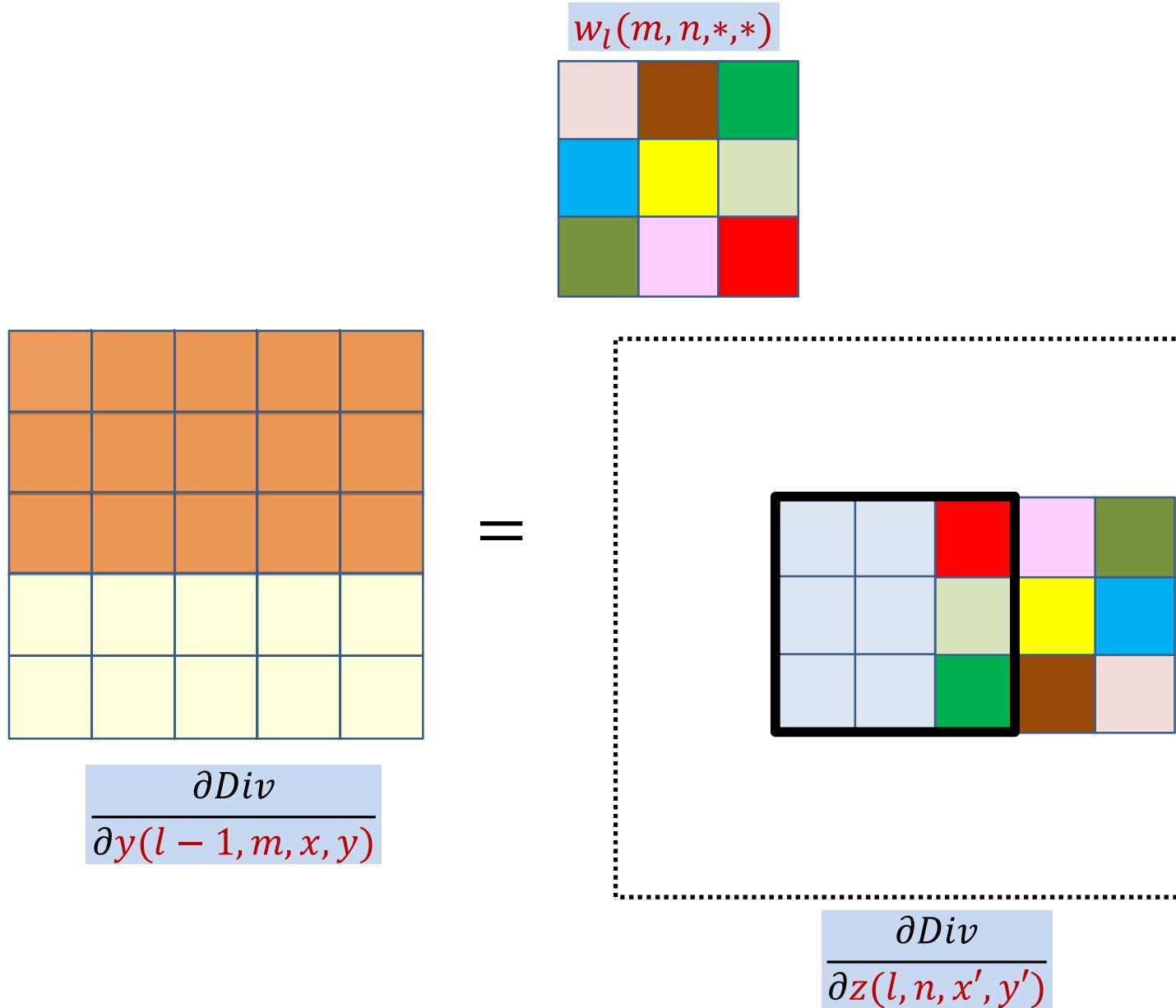
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



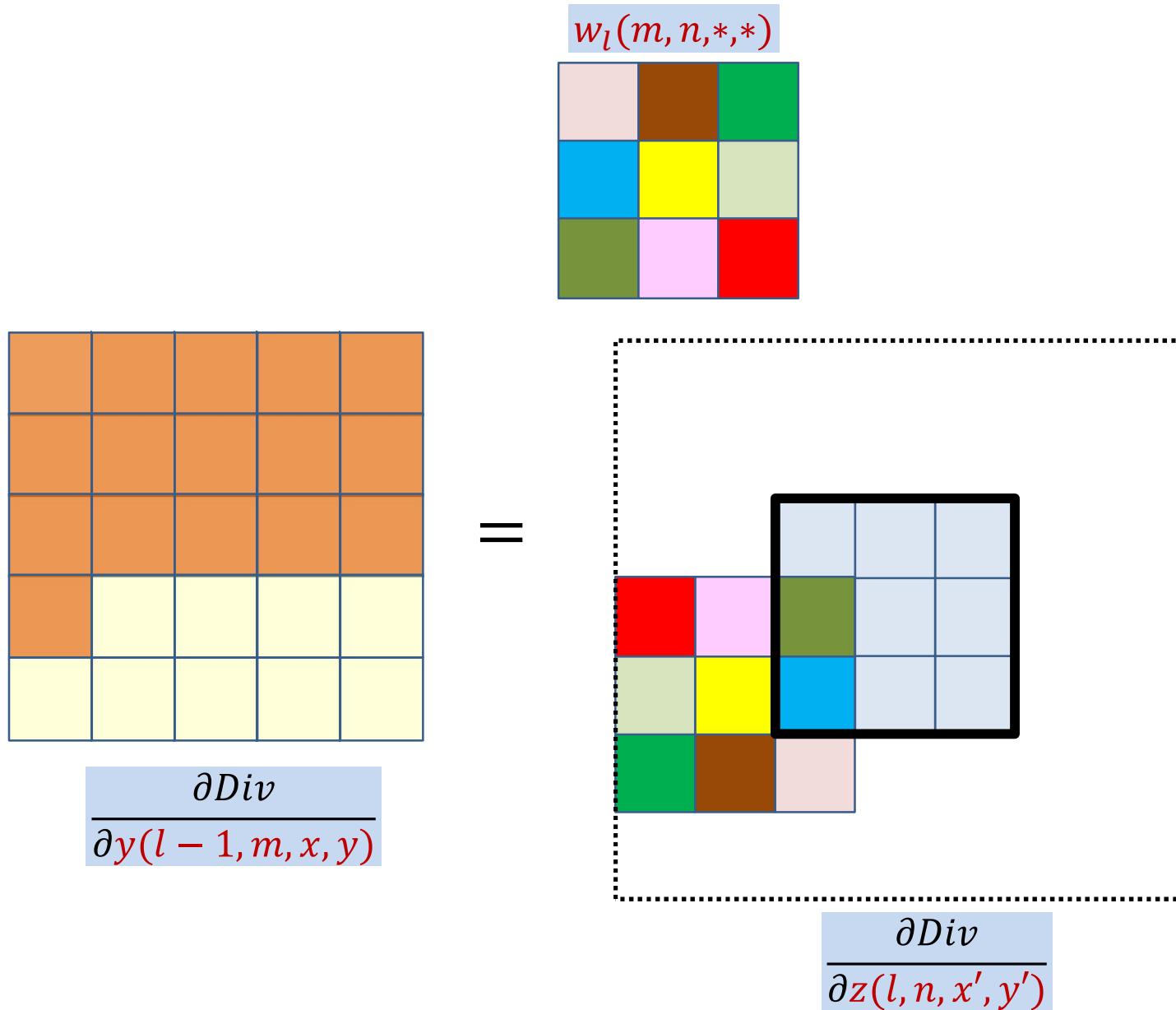
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



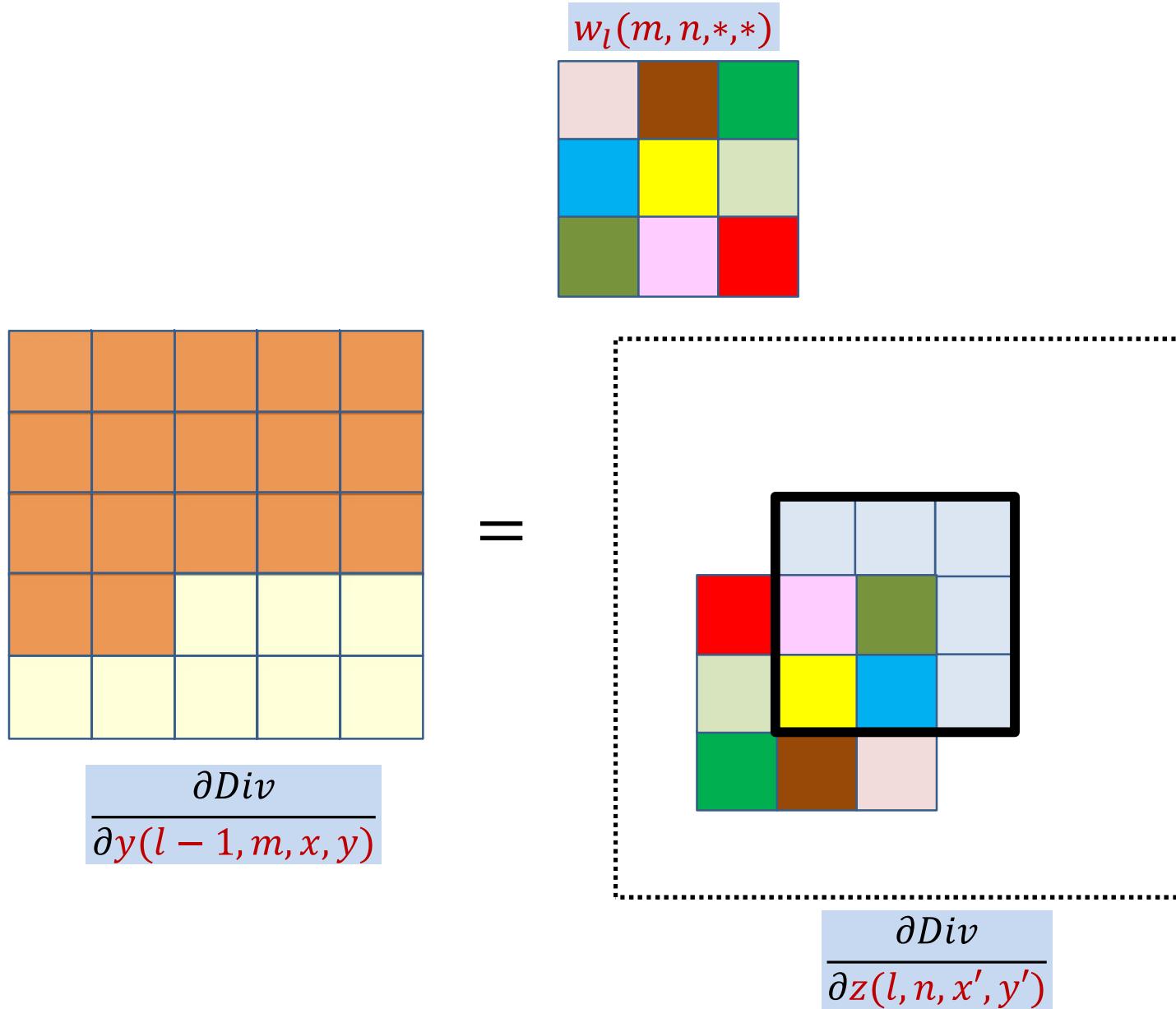
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



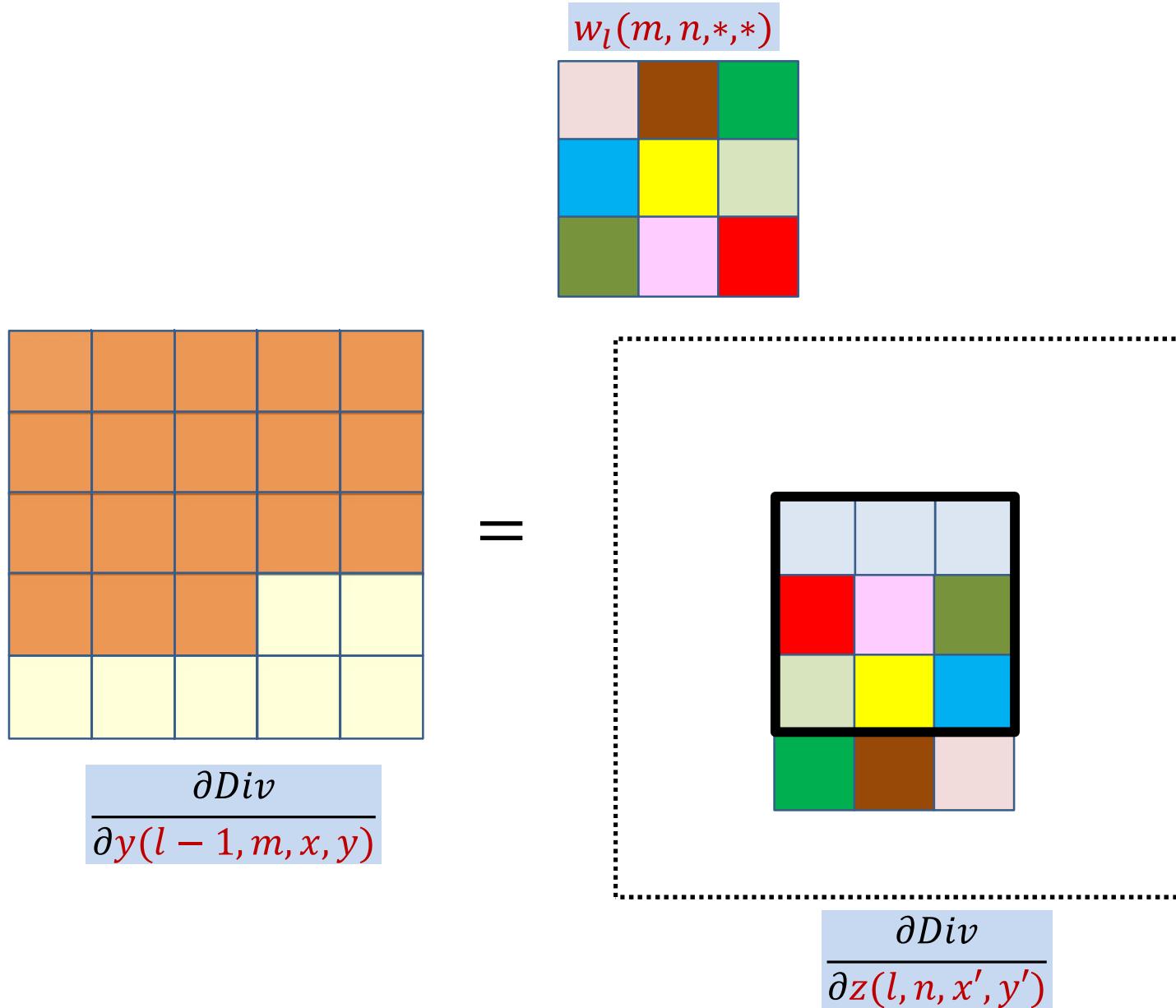
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



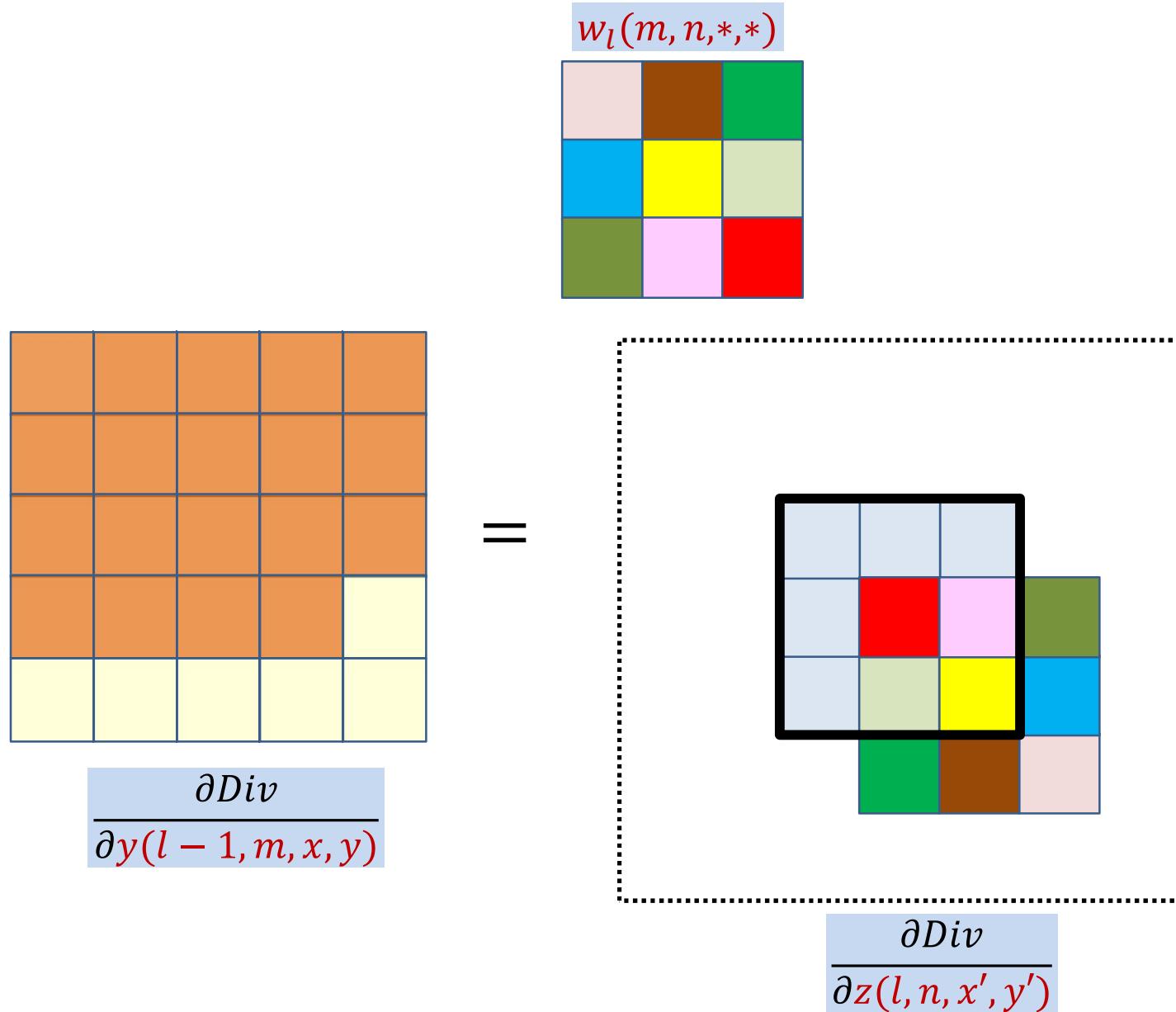
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



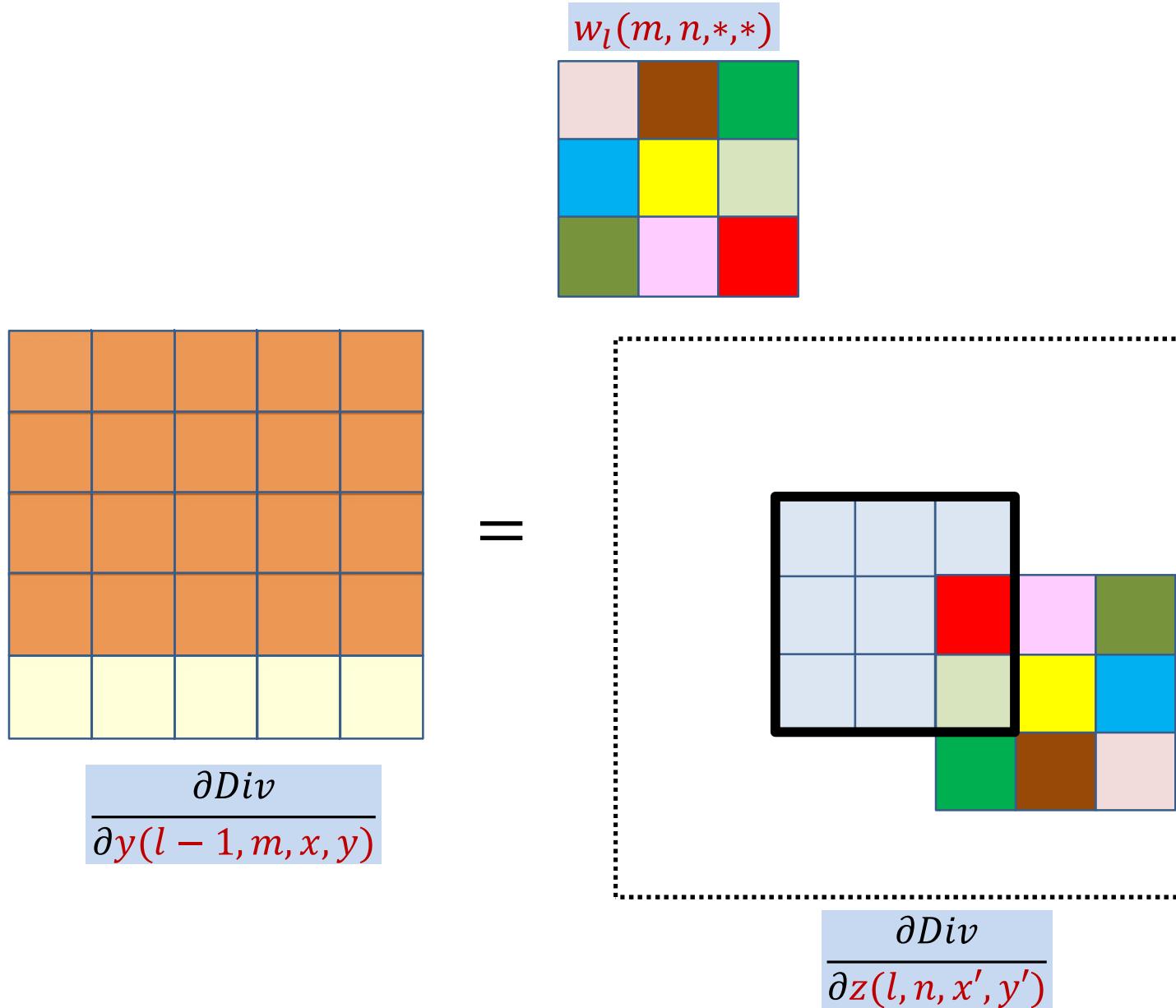
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



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



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



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

$$\frac{\partial \text{Div}}{\partial y(l - 1, m, x, y)} = w_l(m, n, *, *) \cdot \frac{\partial \text{Div}}{\partial z(l, n, x', y')}$$

The diagram illustrates the computation of the derivative of the divergence operator with respect to a specific element of a matrix. On the left, a 5x5 input matrix  $\frac{\partial \text{Div}}{\partial y}$  is shown with alternating orange and yellow cells. A 3x3 mask  $w_l(m, n, *, *)$  is applied to this matrix, centered on the second column and third row. The result is a 3x3 output matrix  $\frac{\partial \text{Div}}{\partial z}$ , where the central cell is highlighted with a black border. The output matrix contains the values from the input matrix that align with the non-zero elements of the mask.

Input Matrix ( $\frac{\partial \text{Div}}{\partial y}$ ):

Orange	Orange	Orange	Orange	Orange
Orange	Orange	Orange	Orange	Orange
Orange	Orange	Orange	Orange	Orange
Orange	Yellow	Yellow	Yellow	Yellow
Orange	Yellow	Yellow	Yellow	Yellow

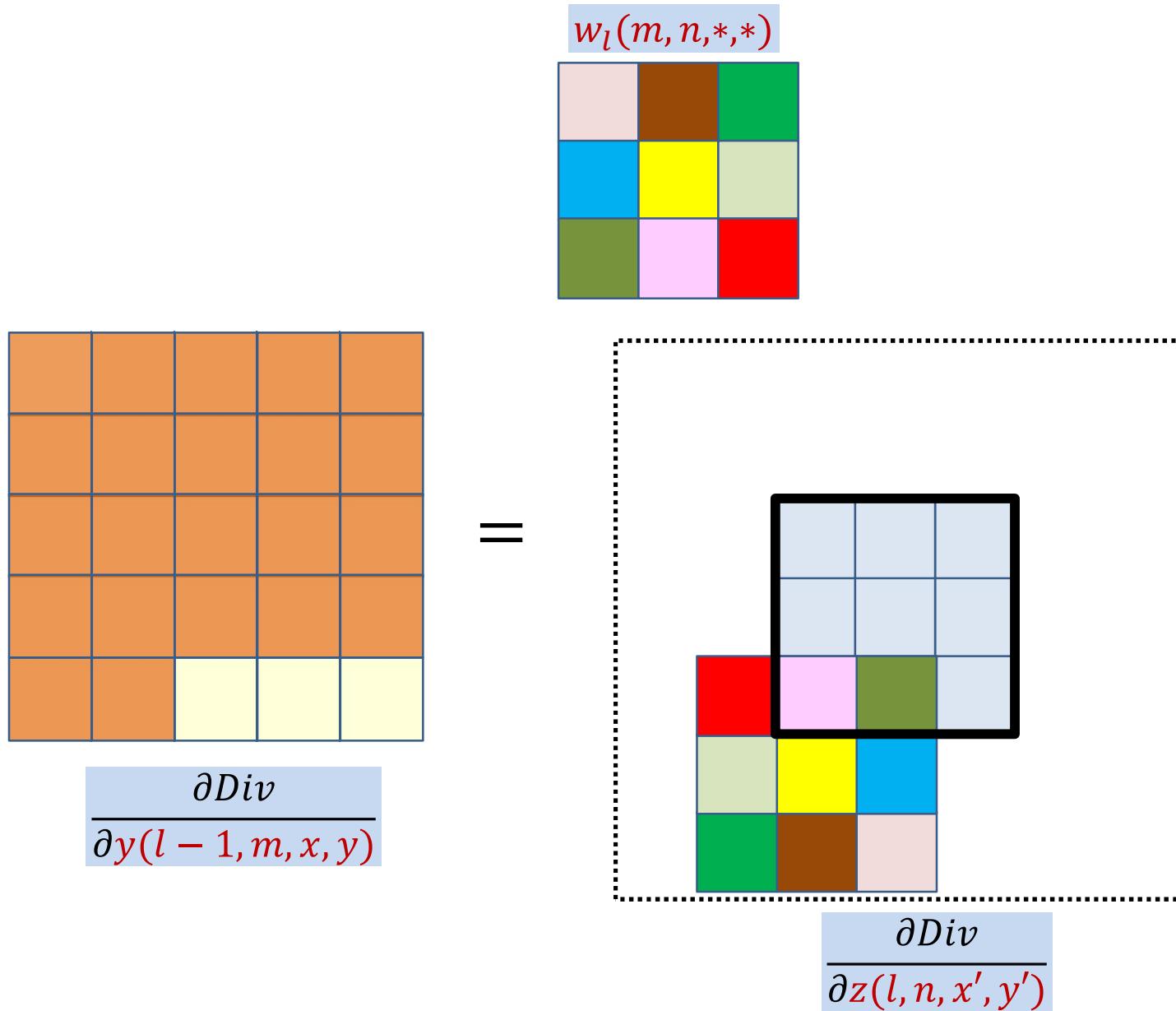
Mask ( $w_l(m, n, *, *)$ ):

Pink	Brown	Green
Blue	Yellow	Light Green
Green	Pink	Red

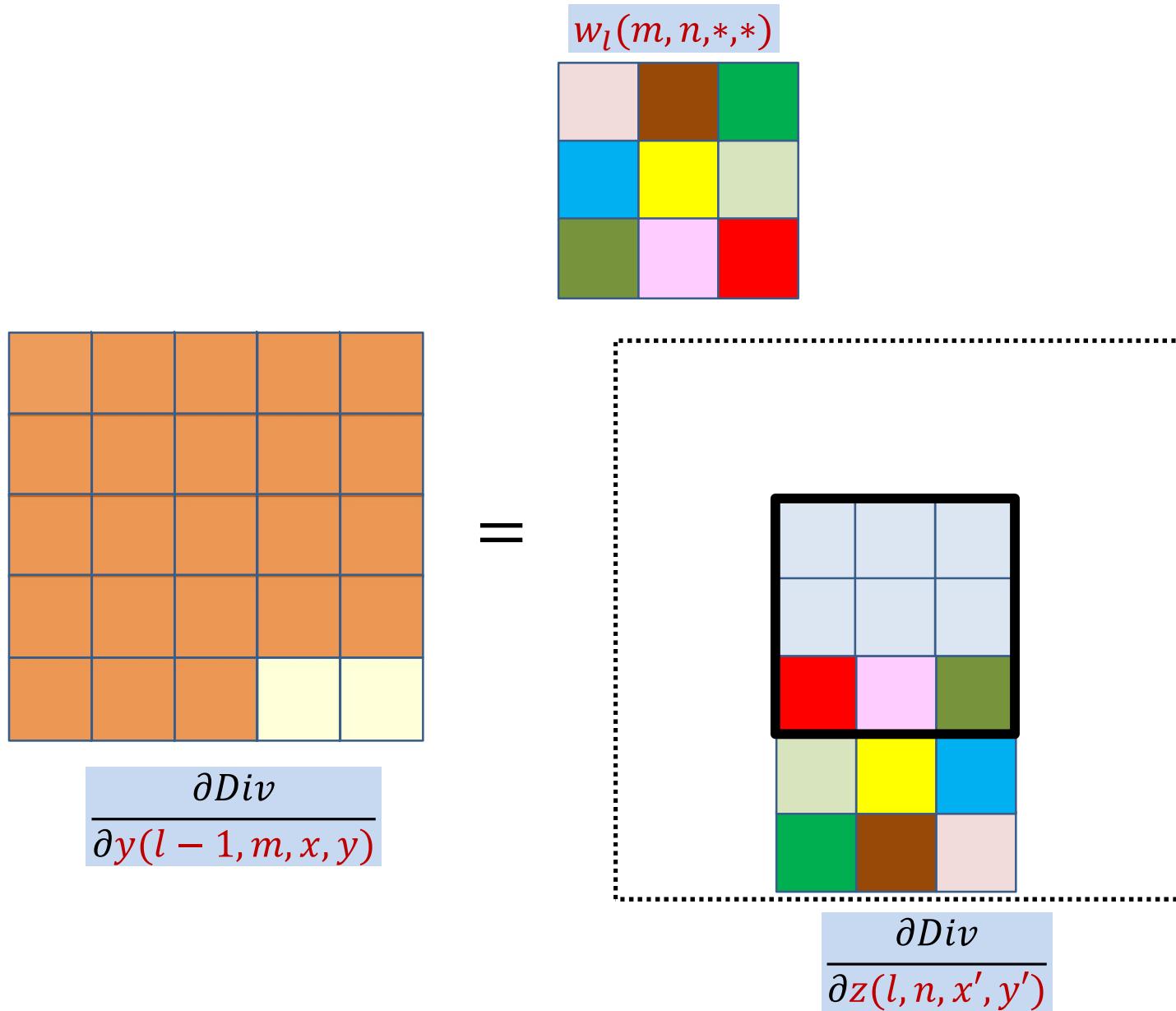
Output Matrix ( $\frac{\partial \text{Div}}{\partial z}$ ):

Red	Pink	Green
Light Green	Yellow	Blue
Green	Brown	Pink

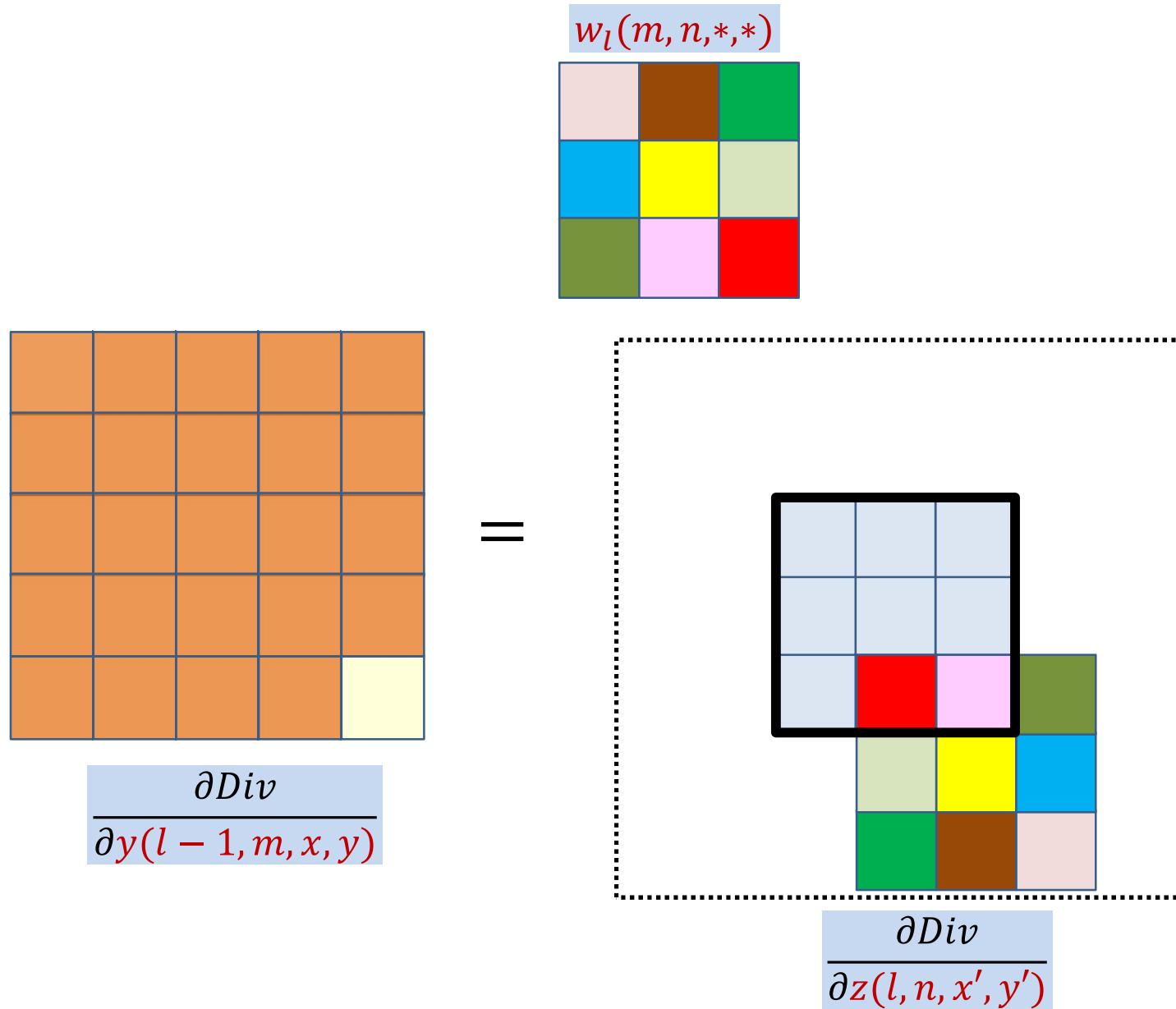
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



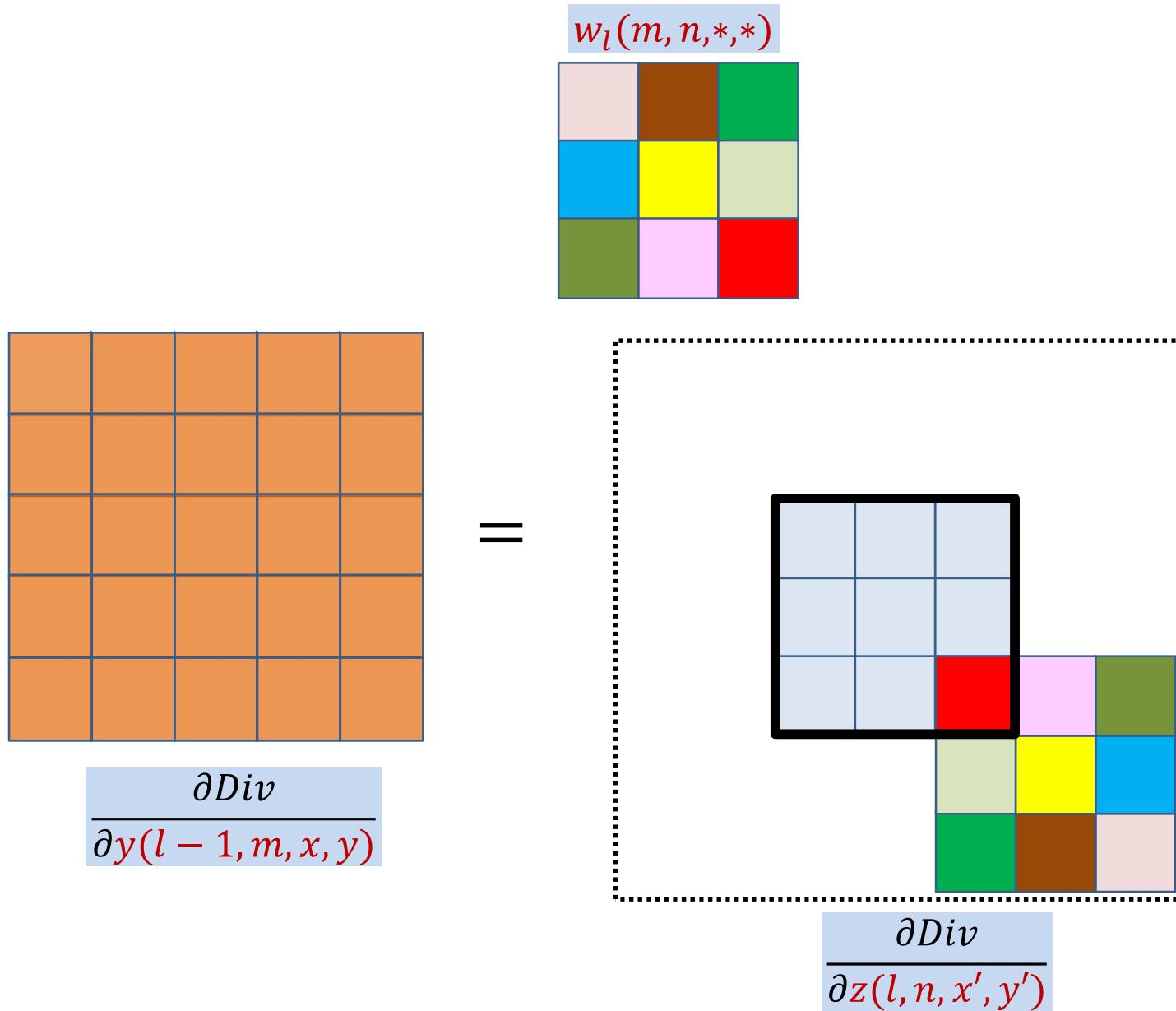
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



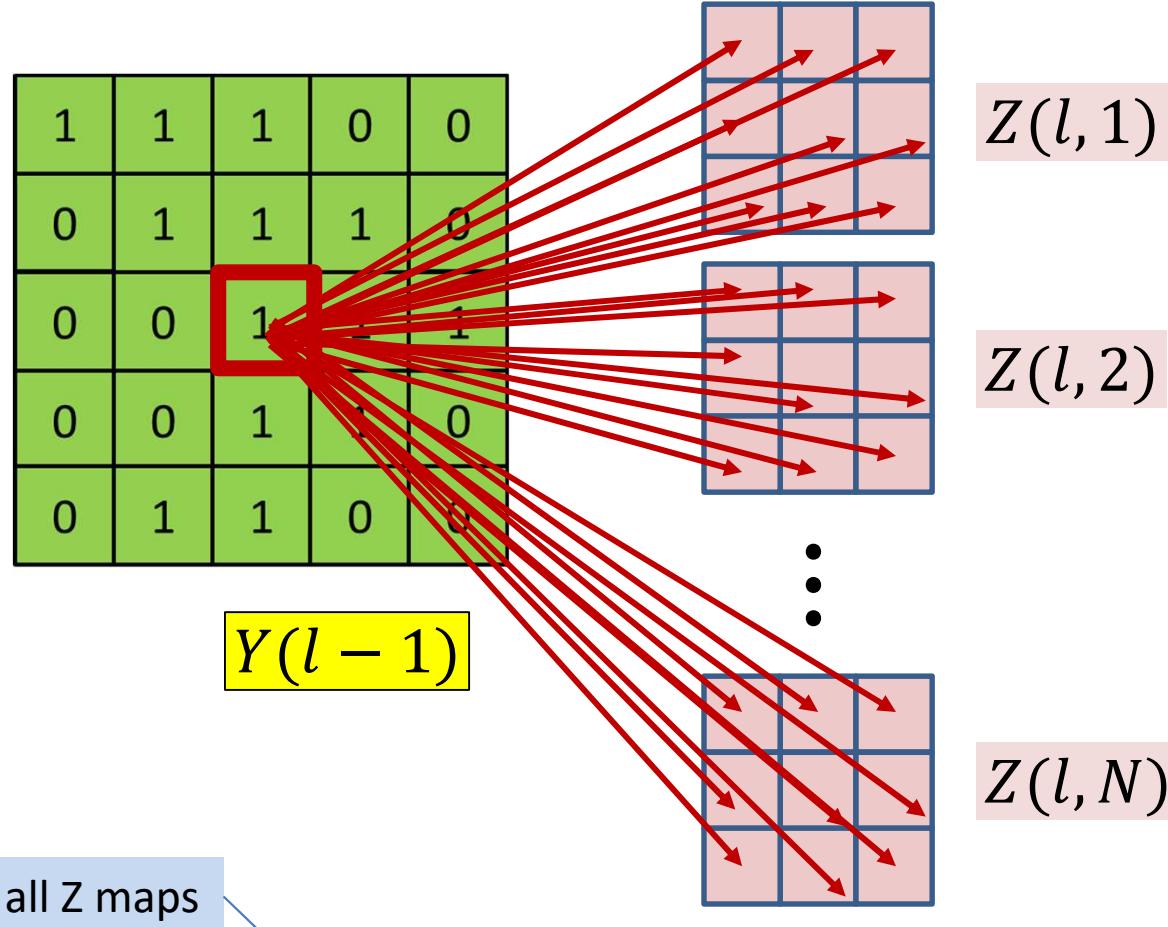
# Derivative at $Y(l - 1, m)$ from a single $Z(l, n)$ map



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

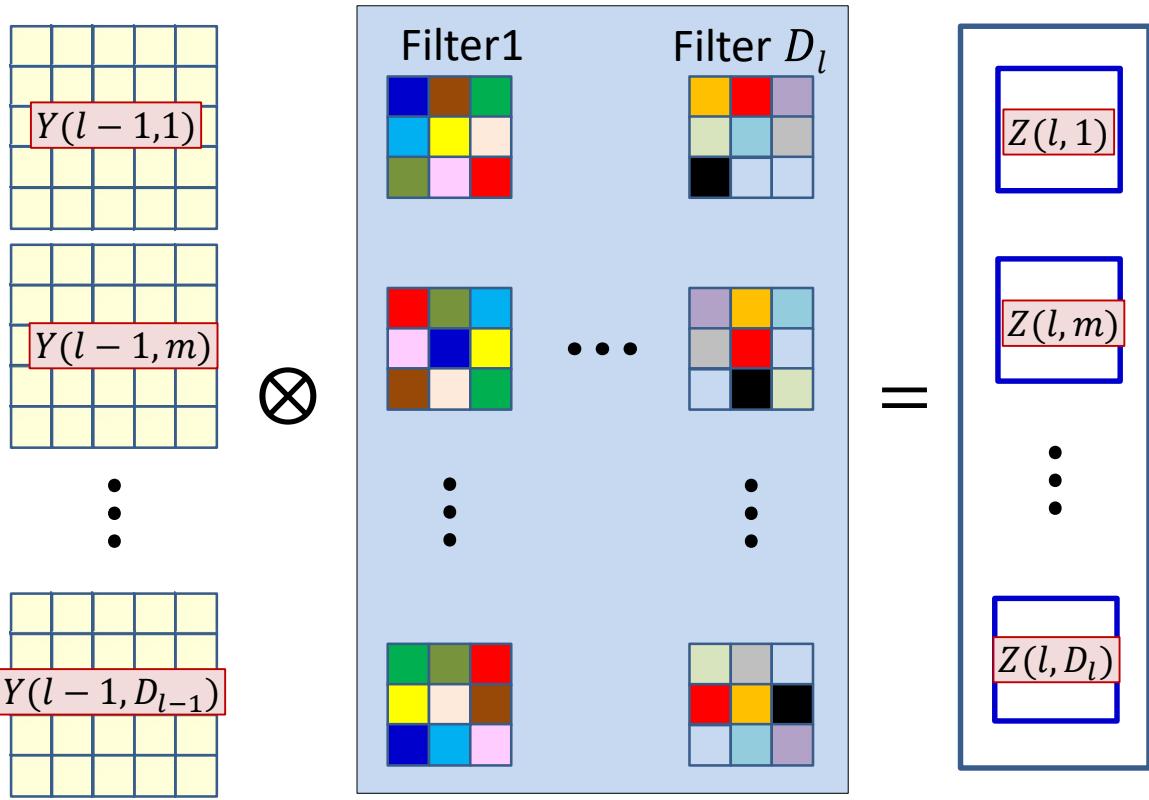


# BP: Convolutional layer



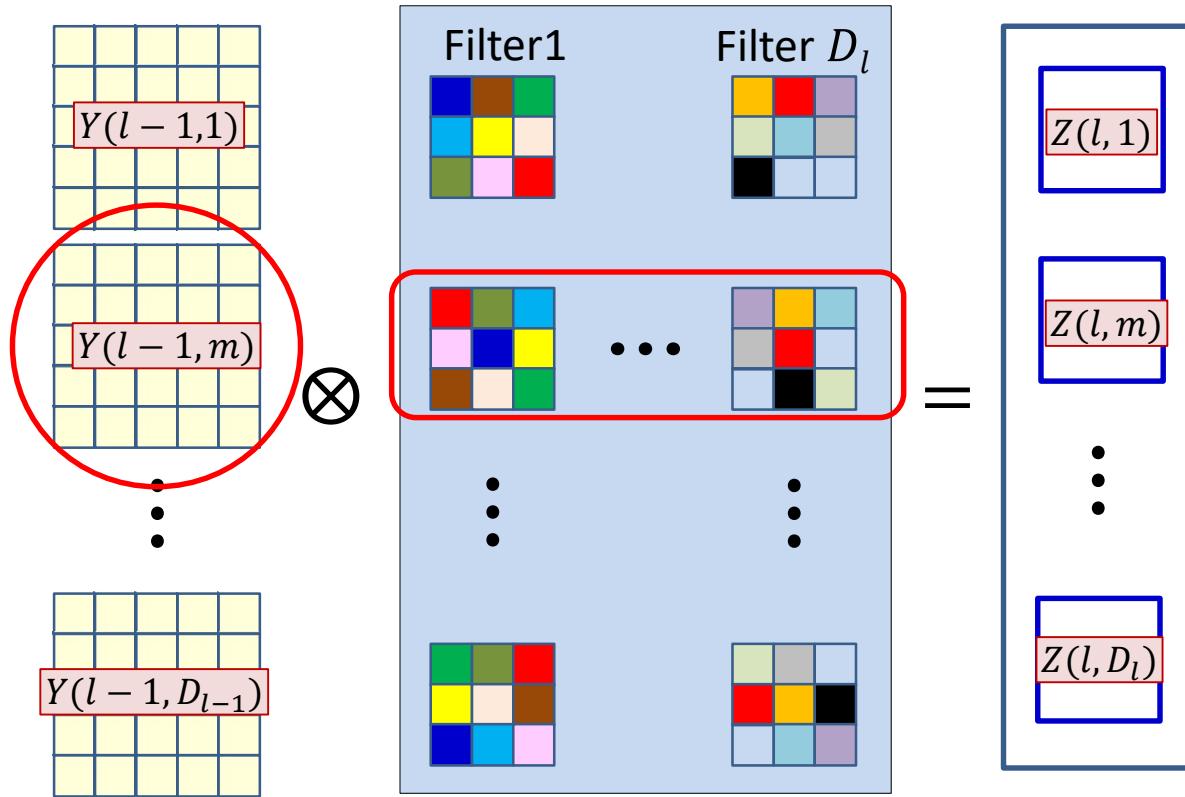
$$\frac{dDiv}{dY(l-1, m, x, y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l, n, x', y')} w_l(m, n, x - x', y - y')$$

# The actual convolutions



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

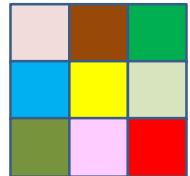
# The actual convolutions



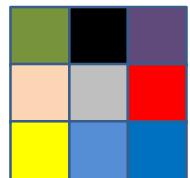
- The  $D_l$  affine maps are produced by convolving with  $D_l$  filters
- The  $m^{\text{th}}$   $Y$  map always convolves the  $m^{\text{th}}$  plane of the filters
- The derivative for the  $m^{\text{th}}$   $Y$  map will invoke the  $m^{\text{th}}$  plane of *all* the filters

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

$n = 1$

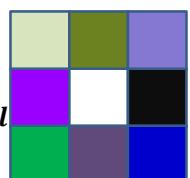


$n = 2$



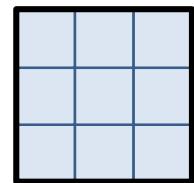
⋮

$n = D_l$

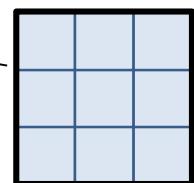


In reality, the derivative at each (x,y) location is obtained from *all* z maps

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

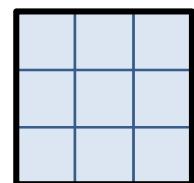


=



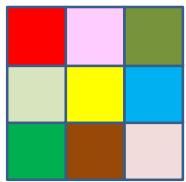
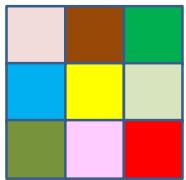
⋮

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

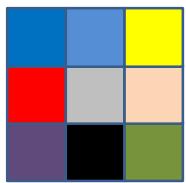
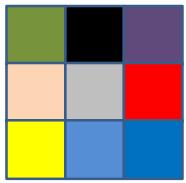


$$w_l(m, n, x, y)$$

$n = 1$



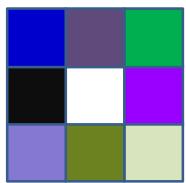
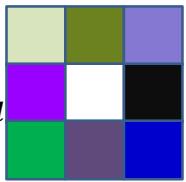
$n = 2$



⋮

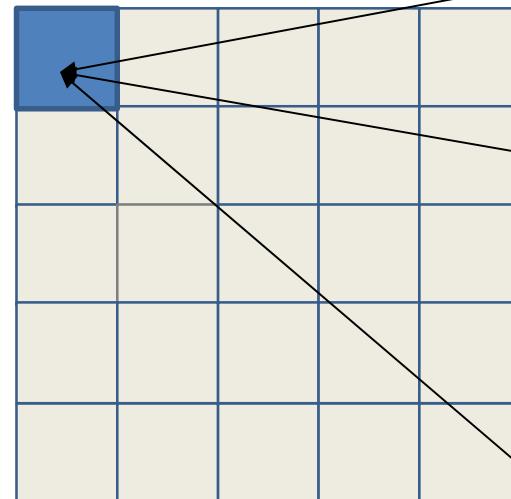
⋮

$n = D_l$

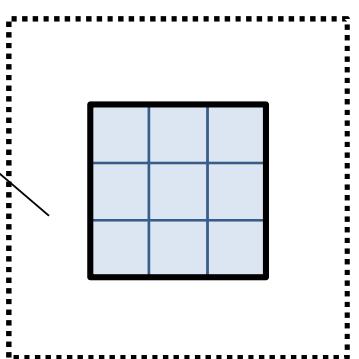
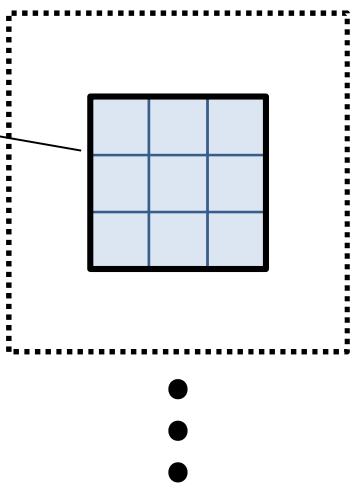
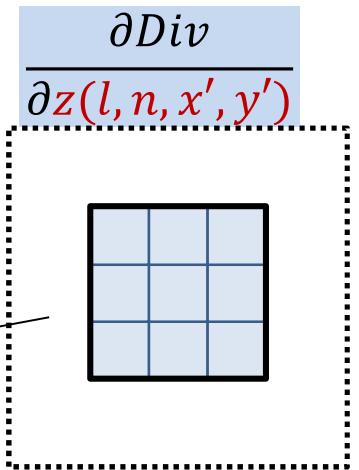


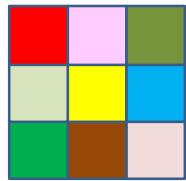
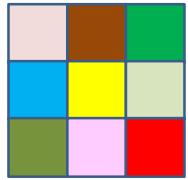
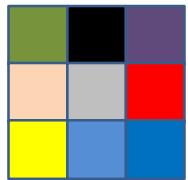
$$w_l(m, n, K + 1 - x, K + 1 - y)$$

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

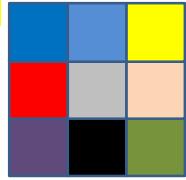
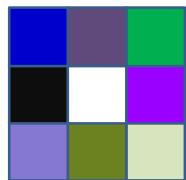
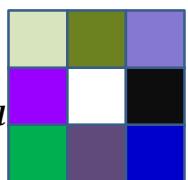


=

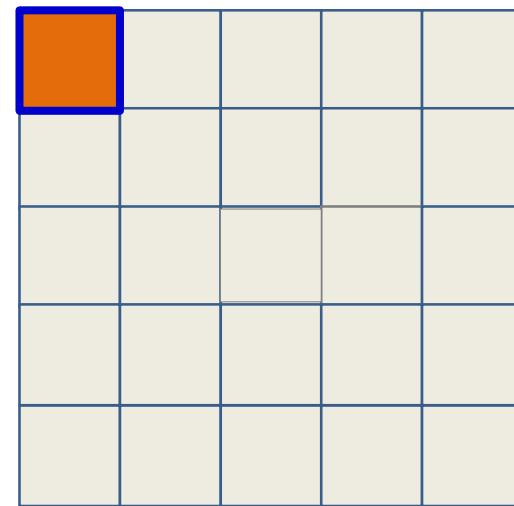


$w_l(m, n, x, y)$  $n = 1$  $n = 2$ 

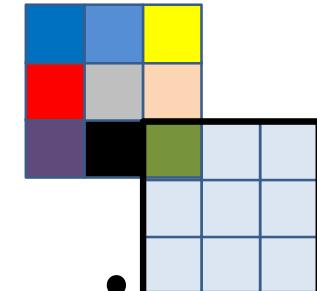
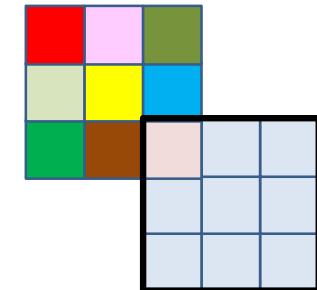
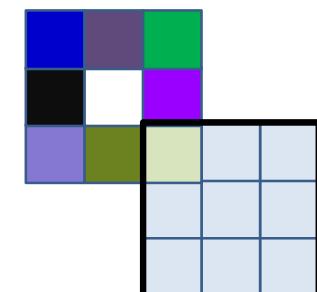
flip

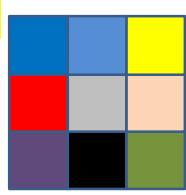
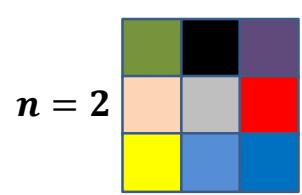
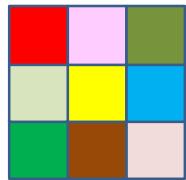
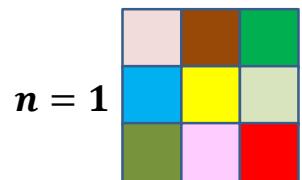
 $\vdots$  $\vdots$  $n = D_l$  $w_l(m, n, K + 1 - x, K + 1 - y)$ 

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



=

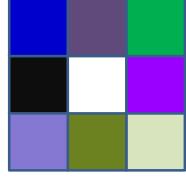
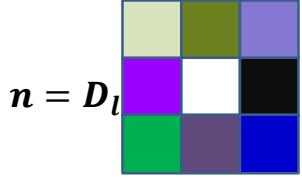
 $\vdots$   
 $\vdots$   
 $\vdots$ 

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

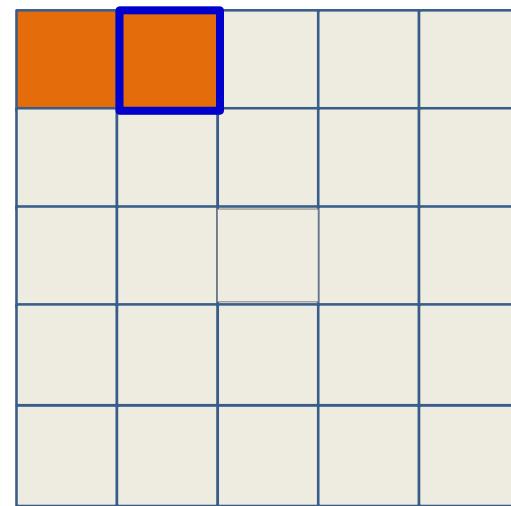
flip

⋮

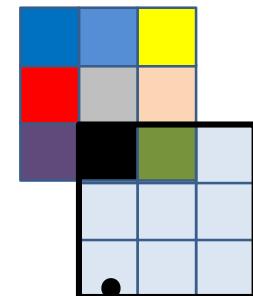
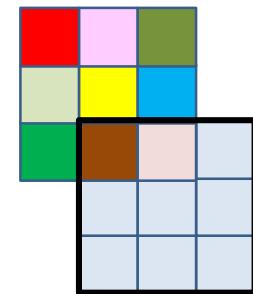
⋮

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

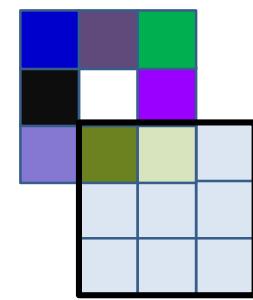
$$\frac{dDiv}{dY(l-1, m, x, y)} = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l, n, x', y')} w_l(m, n, x - x', y - y')$$



=

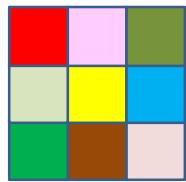
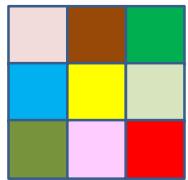


⋮

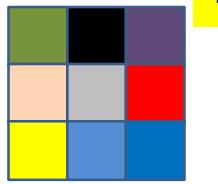


$w_l(m, n, x, y)$

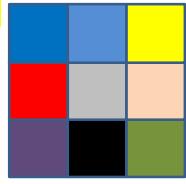
$n = 1$



$n = 2$



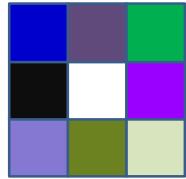
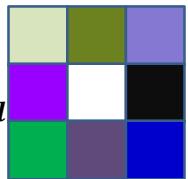
flip



⋮

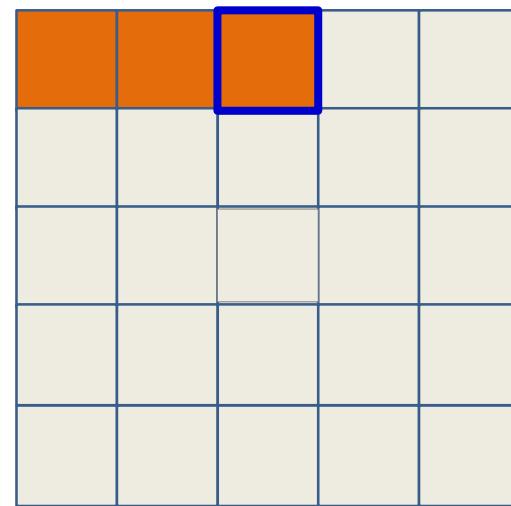
⋮

$n = D_l$

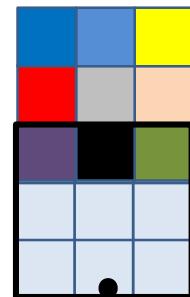
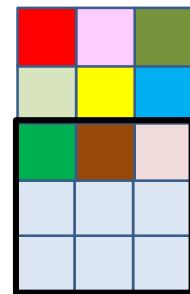


$w_l(m, n, K + 1 - x, K + 1 - y)$

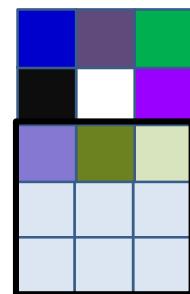
$$\frac{dDiv}{dY(l-1, m, x, y)} = \sum_n \sum_{x', y'} \frac{dDiv}{dz(l, n, x', y')} w_l(m, n, x - x', y - y')$$



=

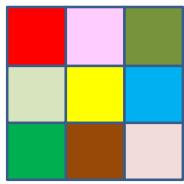
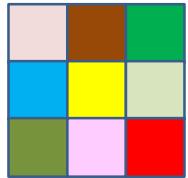


⋮

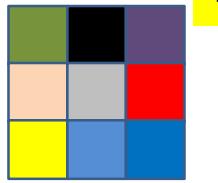


$w_l(m, n, x, y)$

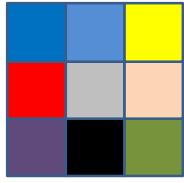
$n = 1$



$n = 2$



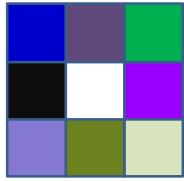
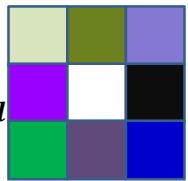
flip



⋮

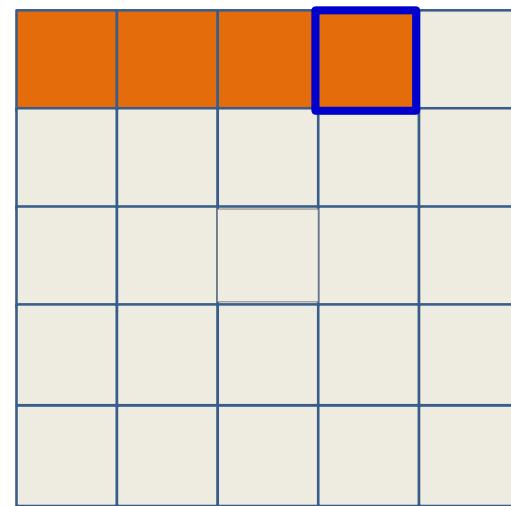
⋮

$n = D_l$

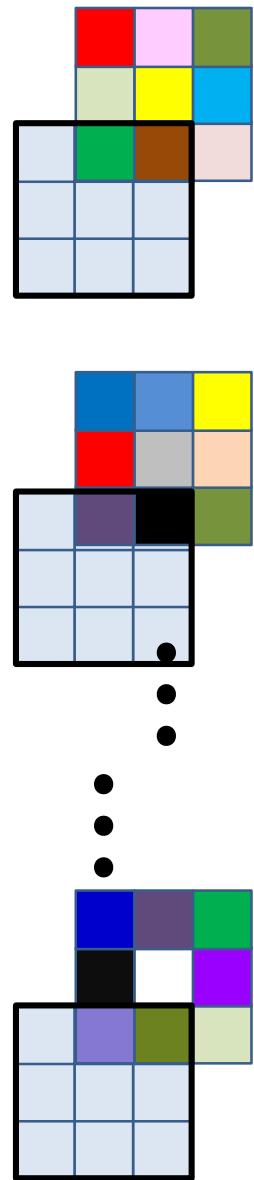


$w_l(m, n, K + 1 - x, K + 1 - y)$

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

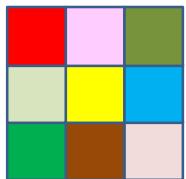
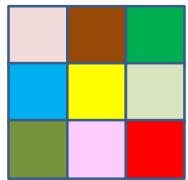


=

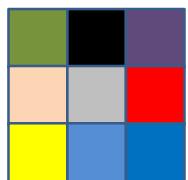


$w_l(m, n, x, y)$

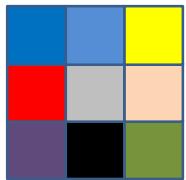
$n = 1$



$n = 2$



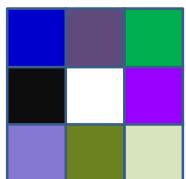
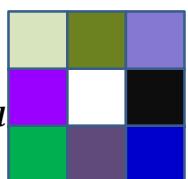
flip



⋮

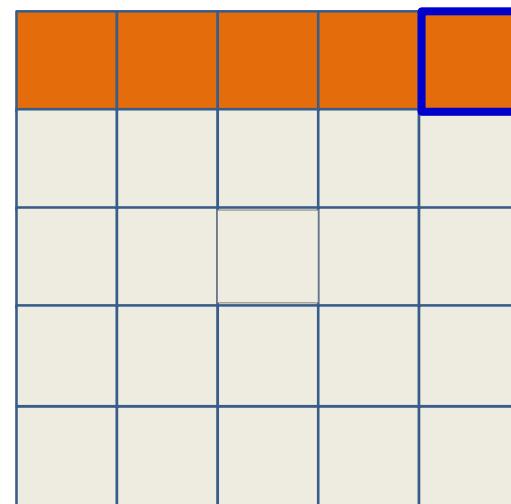
⋮

$n = D_l$

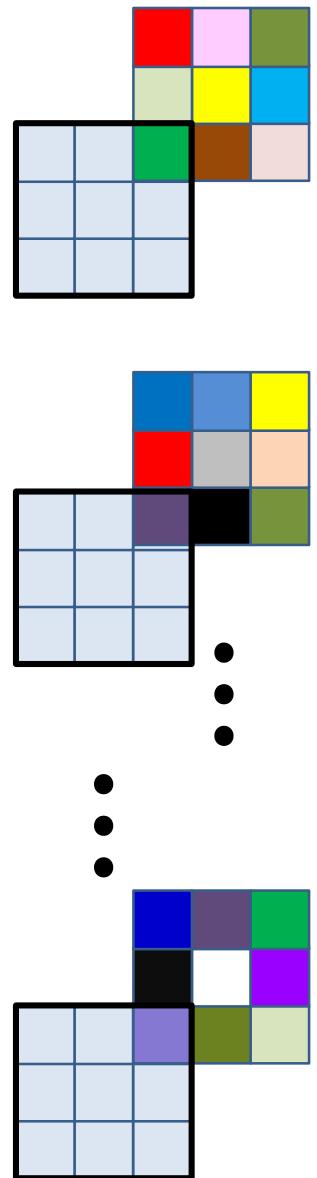


$w_l(m, n, K + 1 - x, K + 1 - y)$

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

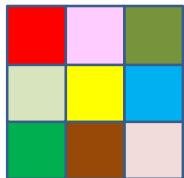
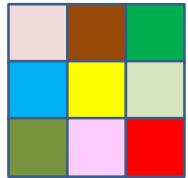


=

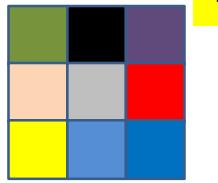


$w_l(m, n, x, y)$

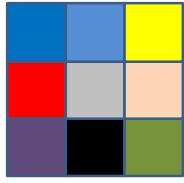
$n = 1$



$n = 2$



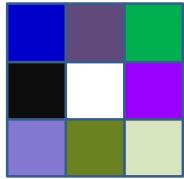
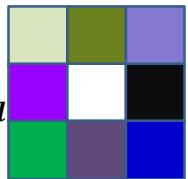
flip



⋮

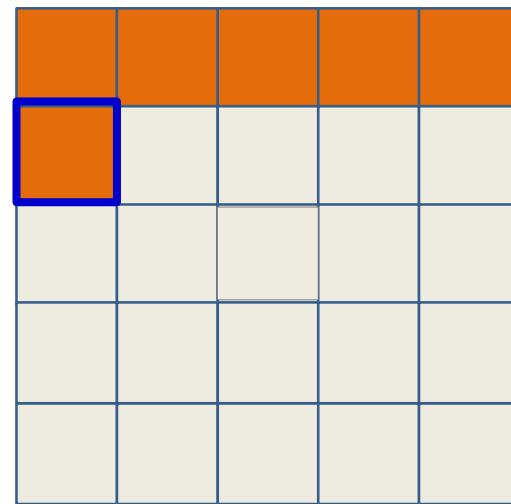
⋮

$n = D_l$

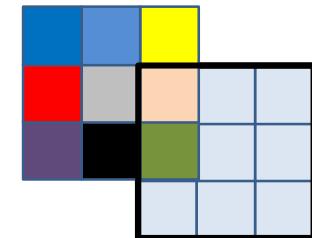
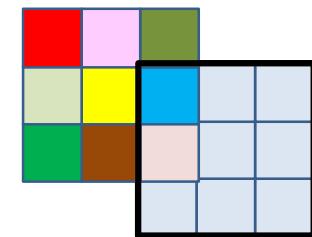


$w_l(m, n, K + 1 - x, K + 1 - y)$

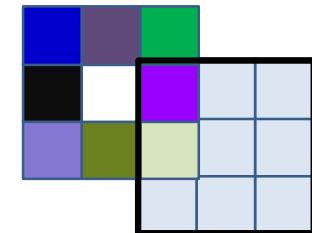
$$\frac{dDiv}{dY(l-1, m, x, y)} = \sum_n \sum_{x', y'} \frac{dDiv}{dz(l, n, x', y')} w_l(m, n, x - x', y - y')$$

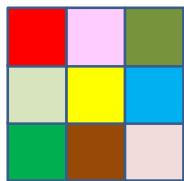
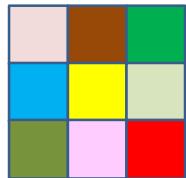
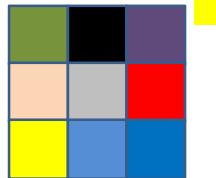


=

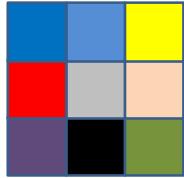


⋮  
⋮  
⋮  
⋮



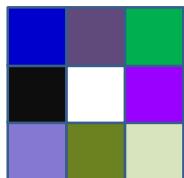
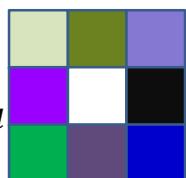
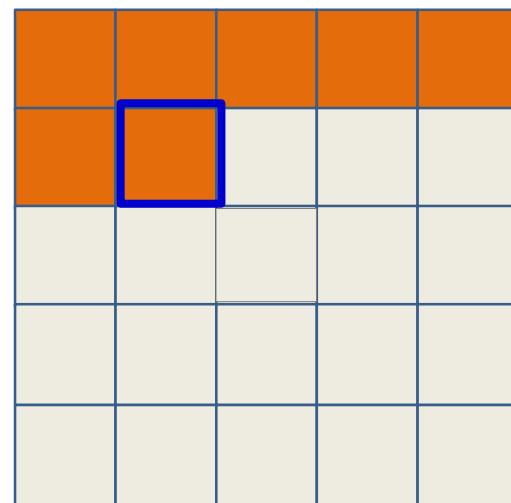
$w_l(m, n, x, y)$  $n = 1$  $n = 2$ 

flip

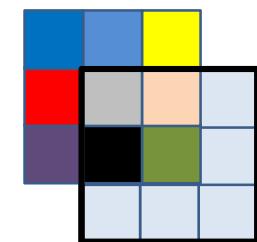
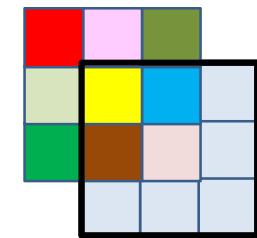


⋮

⋮

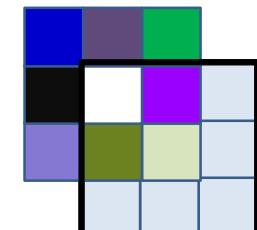
 $n = D_l$  $w_l(m, n, K + 1 - x, K + 1 - y)$ 

=



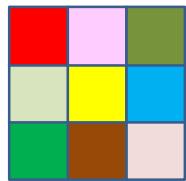
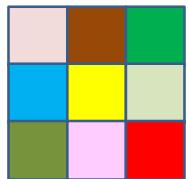
⋮

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

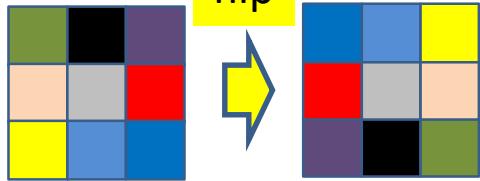


$w_l(m, n, x, y)$

$n = 1$



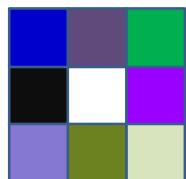
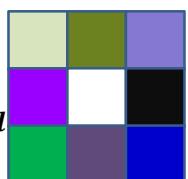
$n = 2$



⋮

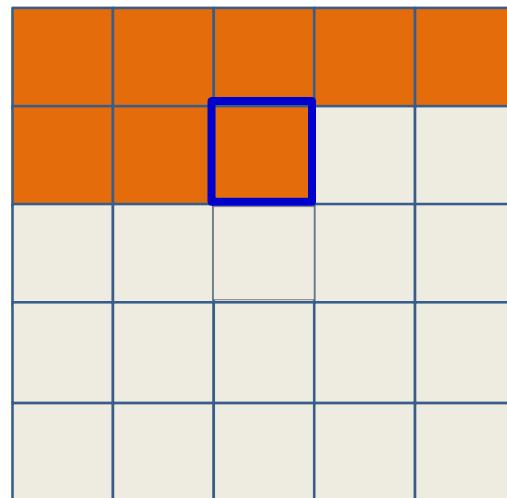
⋮

$n = D_l$

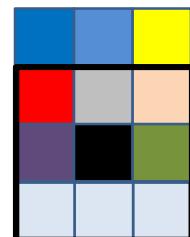
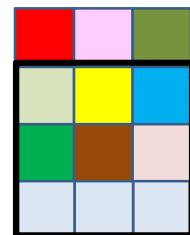


$w_l(m, n, K + 1 - x, K + 1 - y)$

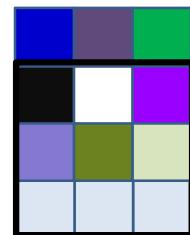
$$\frac{dDiv}{dY(l-1, m, x, y)} = \sum_n \sum_{x', y'} \frac{dDiv}{dz(l, n, x', y')} w_l(m, n, x - x', y - y')$$

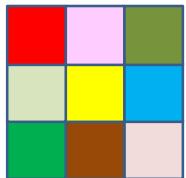
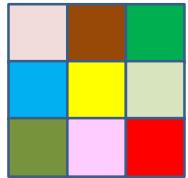
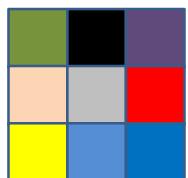


=

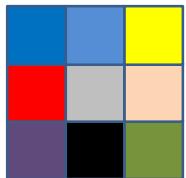
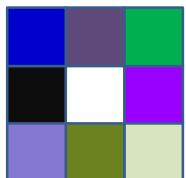
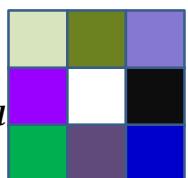


⋮

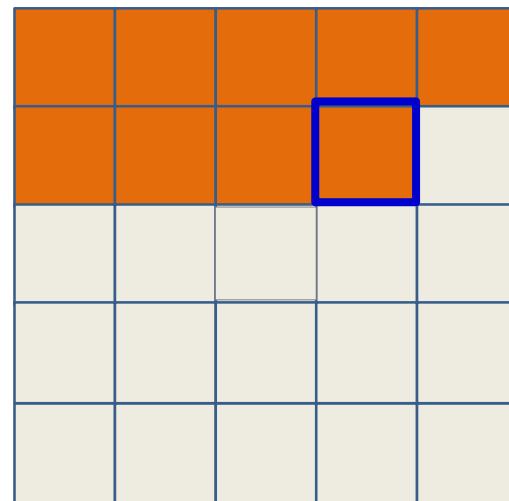


$w_l(m, n, x, y)$  $n = 1$  $n = 2$ 

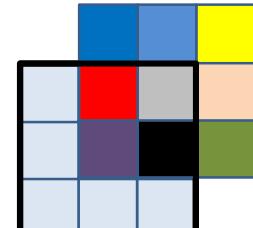
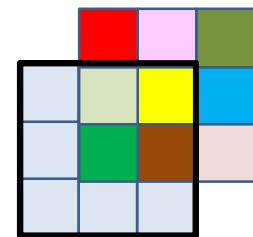
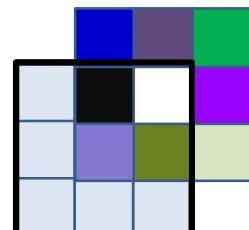
flip

 $\vdots$  $\vdots$  $n = D_l$  $w_l(m, n, K + 1 - x, K + 1 - y)$ 

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

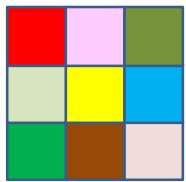
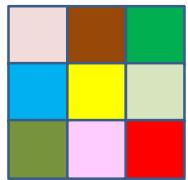


=

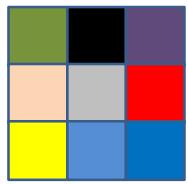
 $\vdots$ 

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

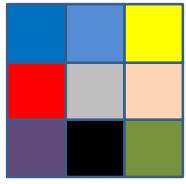
$n = 1$



$n = 2$



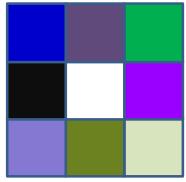
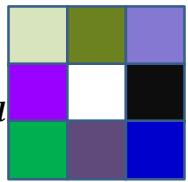
flip



⋮

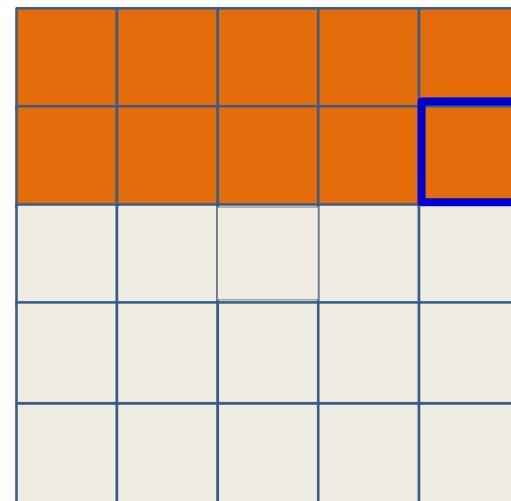
⋮

$n = D_l$

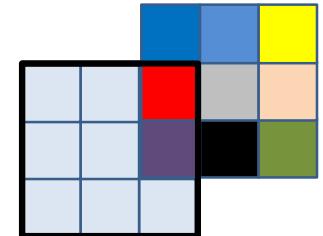
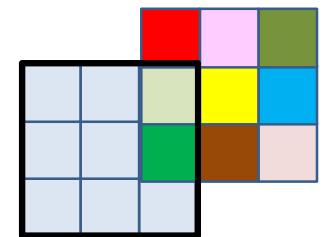


$w_l(m, n, K + 1 - x, K + 1 - y)$

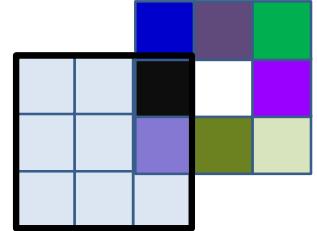
$$\frac{dDiv}{dY(l-1, m, x, y)} = \sum_n \sum_{x', y'} \frac{dDiv}{dz(l, n, x', y')} w_l(m, n, x - x', y - y')$$



=

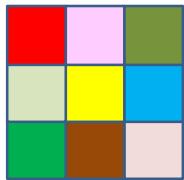
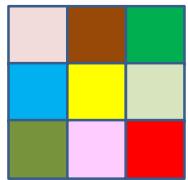


⋮

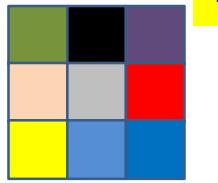


$w_l(m, n, x, y)$

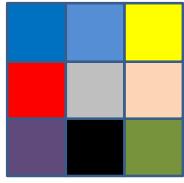
$n = 1$



$n = 2$



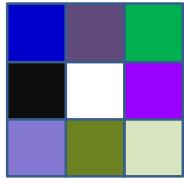
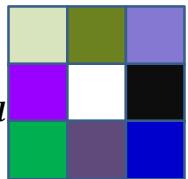
flip



⋮

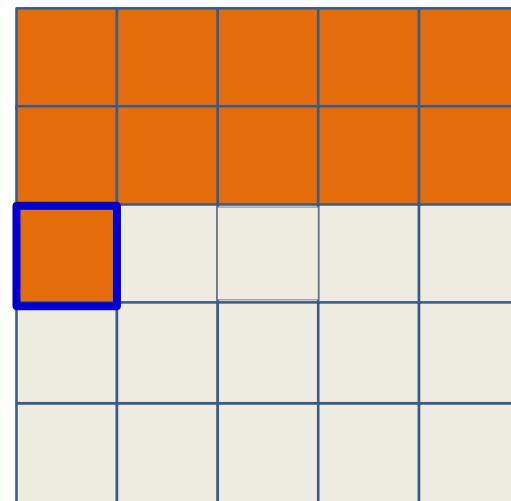
⋮

$n = D_l$

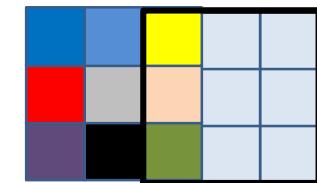
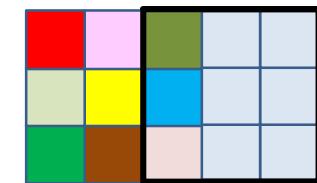


$w_l(m, n, K + 1 - x, K + 1 - y)$

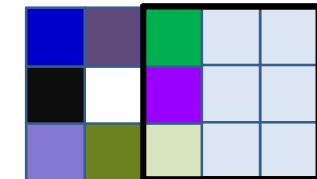
$$\frac{dDiv}{dY(l-1, m, x, y)} = \sum_n \sum_{x', y'} \frac{dDiv}{dz(l, n, x', y')} w_l(m, n, x - x', y - y')$$



=

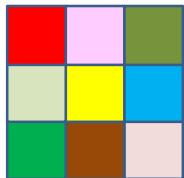
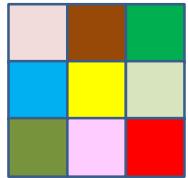


⋮

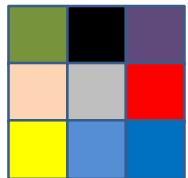


$w_l(m, n, x, y)$

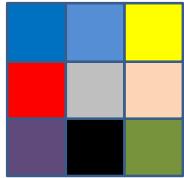
$n = 1$



$n = 2$



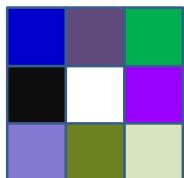
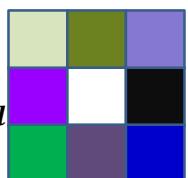
flip



⋮

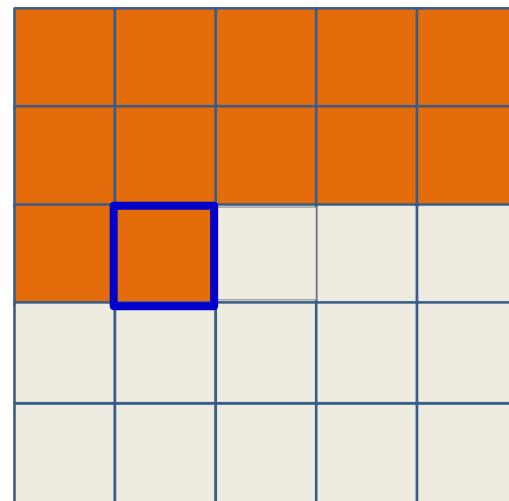
⋮

$n = D_l$

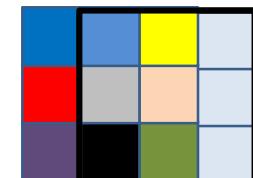
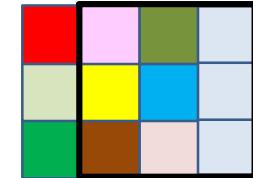


$w_l(m, n, K + 1 - x, K + 1 - y)$

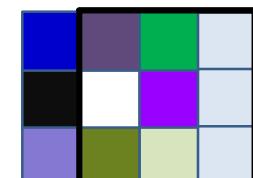
$$dY(l-1, m, x, y) = \sum_n \sum_{x', y'} \frac{dDiv}{dz(l, n, x', y')} w_l(m, n, x - x', y - y')$$

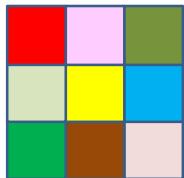
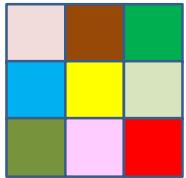
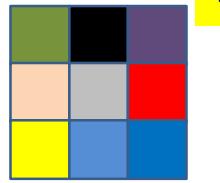


=

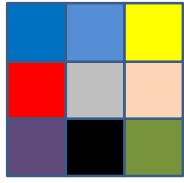
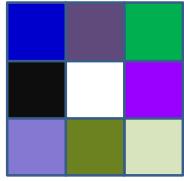
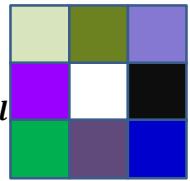


⋮

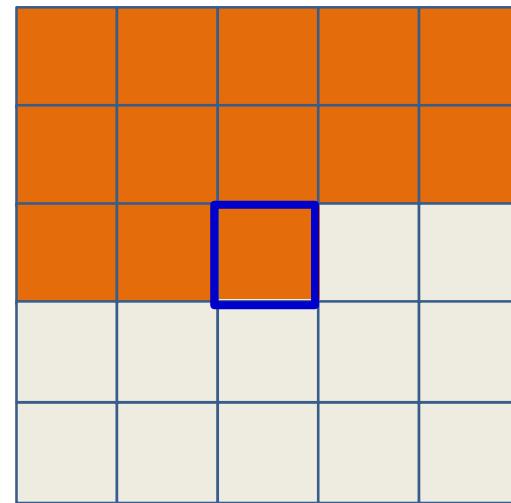


$w_l(m, n, x, y)$  $n = 1$  $n = 2$ 

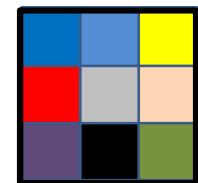
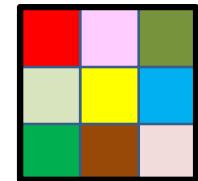
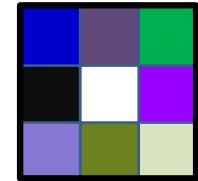
flip

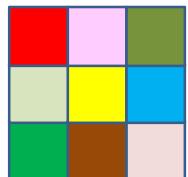
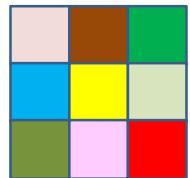
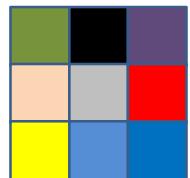
 $\vdots$  $\vdots$  $n = D_l$  $w_l(m, n, K + 1 - x, K + 1 - y)$ 

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

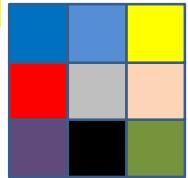
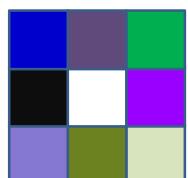
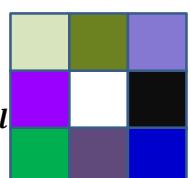


=

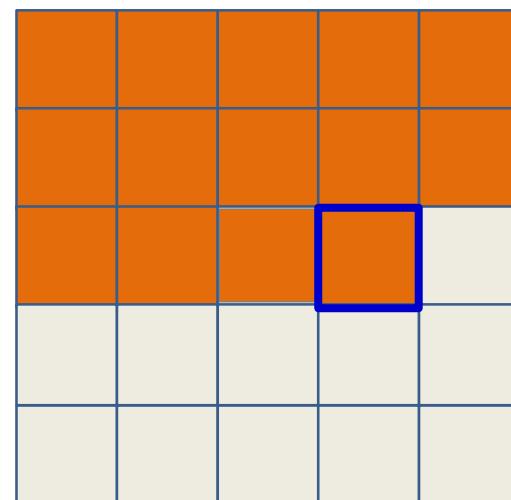
 $\vdots$  $\vdots$ 

$w_l(m, n, x, y)$  $n = 1$  $n = 2$ 

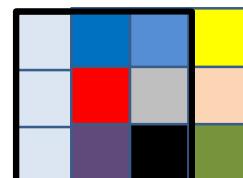
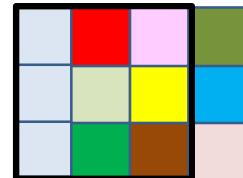
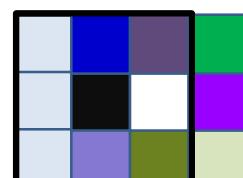
flip

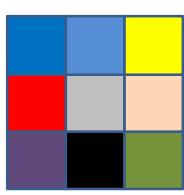
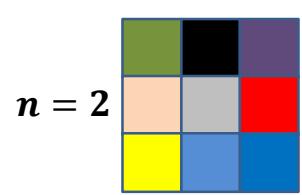
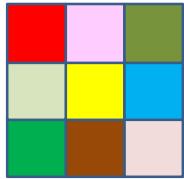
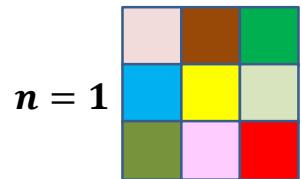
 $\vdots$  $\vdots$  $n = D_l$  $w_l(m, n, K + 1 - x, K + 1 - y)$ 

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



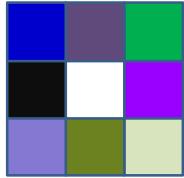
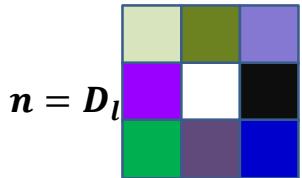
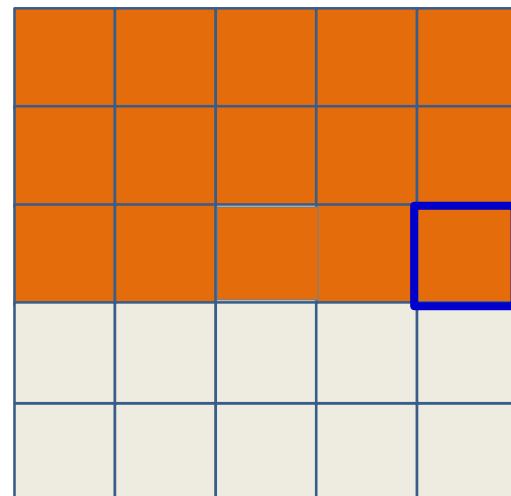
=

 $\vdots$ 

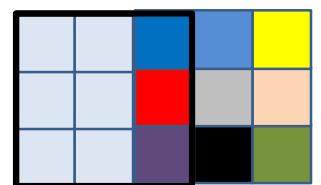
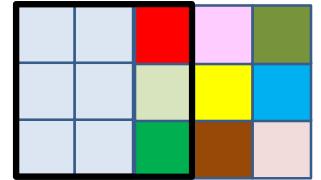
$w_l(m, n, x, y)$ 

⋮

⋮

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

=

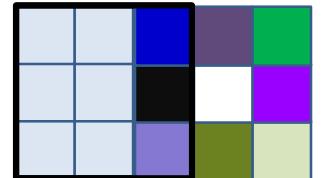


⋮

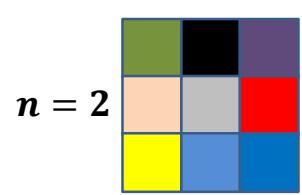
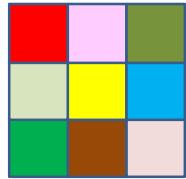
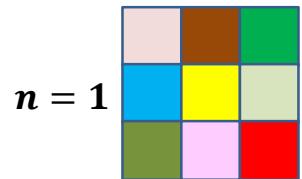
⋮

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

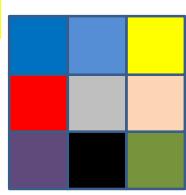
$$d\text{Div} = \sum_n \sum_{x', y'} \frac{d\text{Div}}{dz(l, n, x', y')} w_l(m, n, x - x', y - y')$$



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

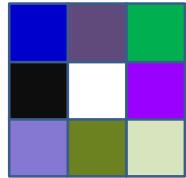
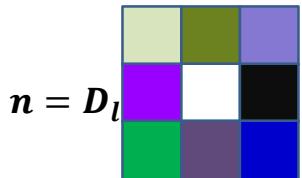


flip



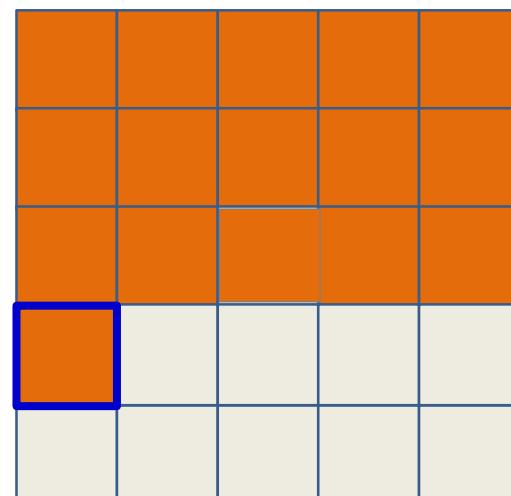
⋮

⋮

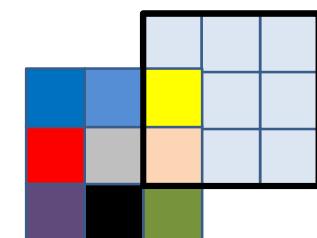
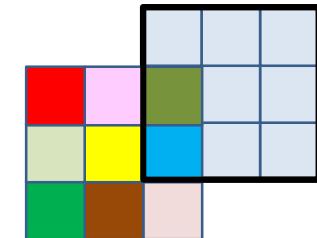


$w_l(m, n, K + 1 - x, K + 1 - y)$

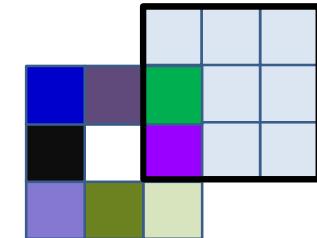
$$\frac{dDiv}{\partial y(l-1, m, x, y)} = \sum_n \sum_{x', y'} \frac{dDiv}{dz(l, n, x', y')} w_l(m, n, x - x', y - y')$$

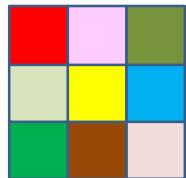
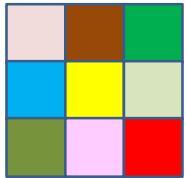
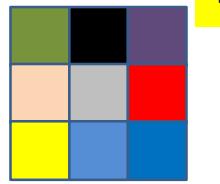


=

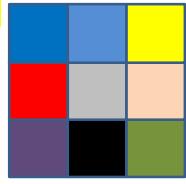
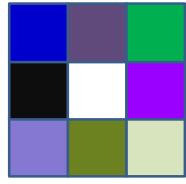
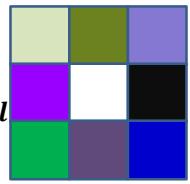


⋮  
⋮  
⋮

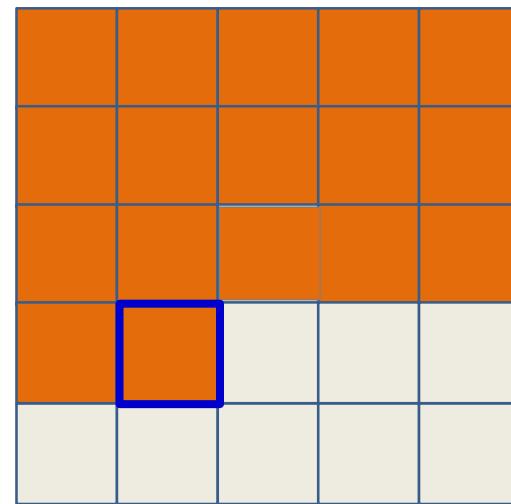


$w_l(m, n, x, y)$  $n = 1$  $n = 2$ 

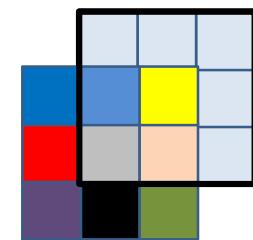
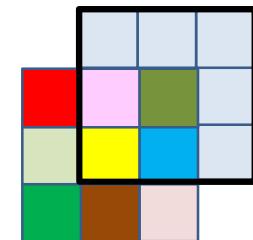
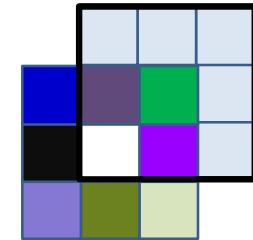
flip

 $\vdots$  $\vdots$  $n = D_l$  $w_l(m, n, K + 1 - x, K + 1 - y)$ 

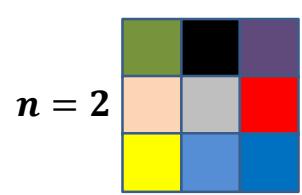
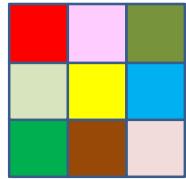
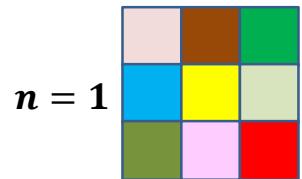
$$\frac{dDiv}{dy(l-1, m, x, y)} = \sum_n \sum_{x', y'} \frac{dDiv}{dz(l, n, x', y')} w_l(m, n, x - x', y - y')$$



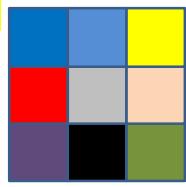
=

 $\vdots$ 

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

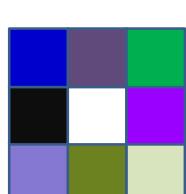
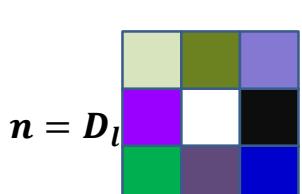


flip

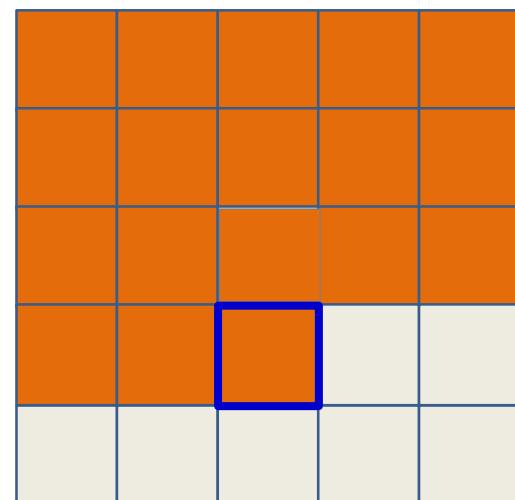


⋮

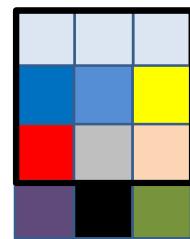
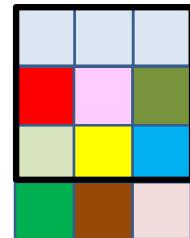
⋮



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

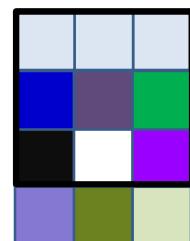


=

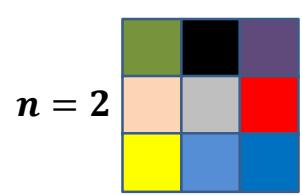
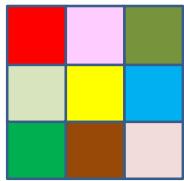
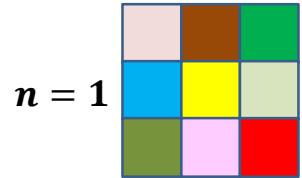


⋮

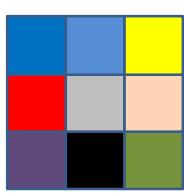
$$dY(l-1, m, x, y) = \sum_n \sum_{x', y'} \frac{dDiv}{dz(l, n, x', y')} w_l(m, n, x - x', y - y')$$



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

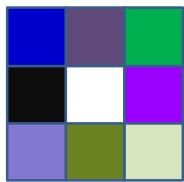
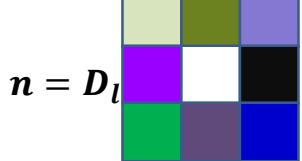


flip



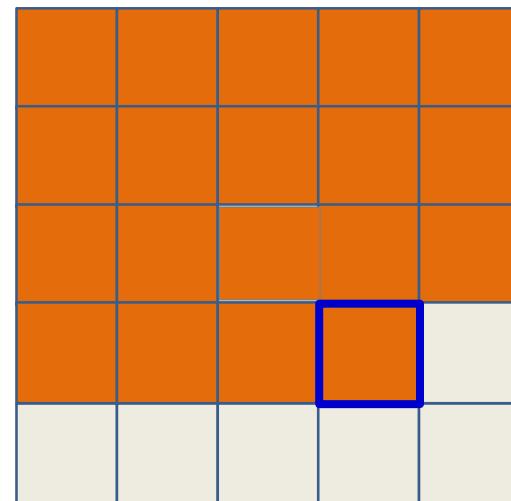
⋮

⋮

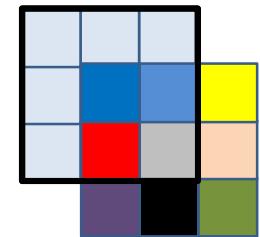
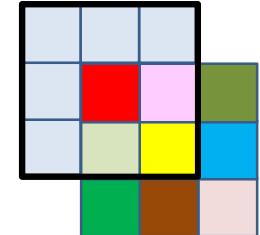


$w_l(m, n, K + 1 - x, K + 1 - y)$

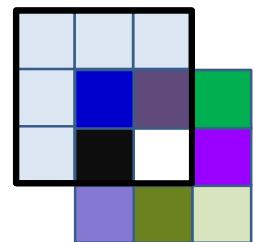
$$\frac{dDiv}{dY(l-1, m, x, y)} = \sum_n \sum_{x', y'} \frac{dDiv}{dz(l, n, x', y')} w_l(m, n, x - x', y - y')$$



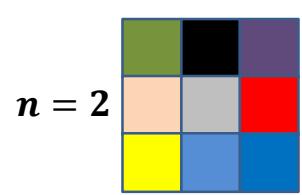
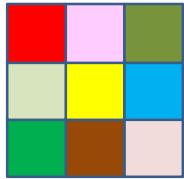
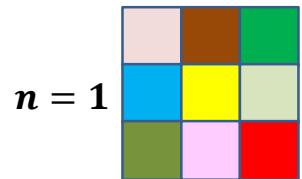
=



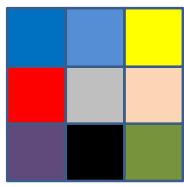
⋮



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

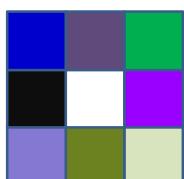
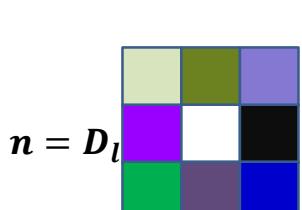


flip

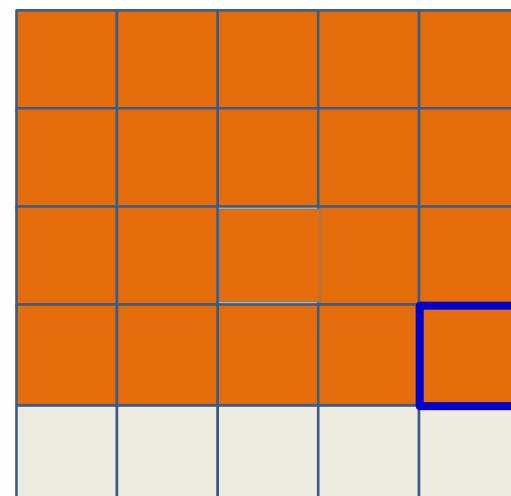


⋮

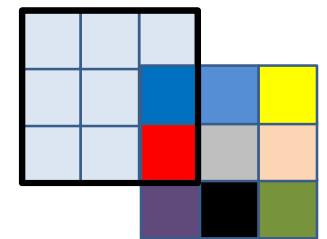
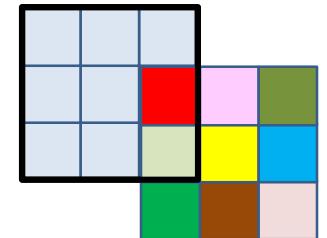
⋮



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

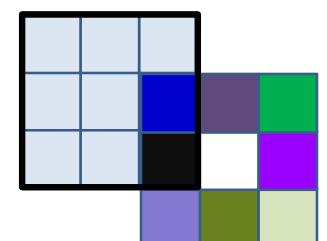


=

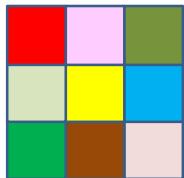
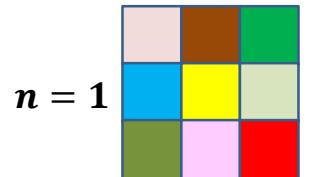


⋮  
⋮  
⋮  
⋮

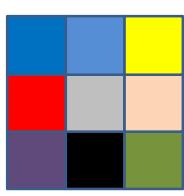
$$dDiv = \sum_n \sum_{x',y'} \frac{dDiv}{dz(l, n, x', y')} w_l(m, n, x - x', y - y')$$



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

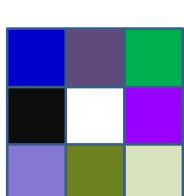
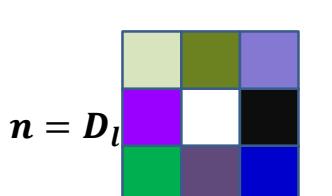


flip

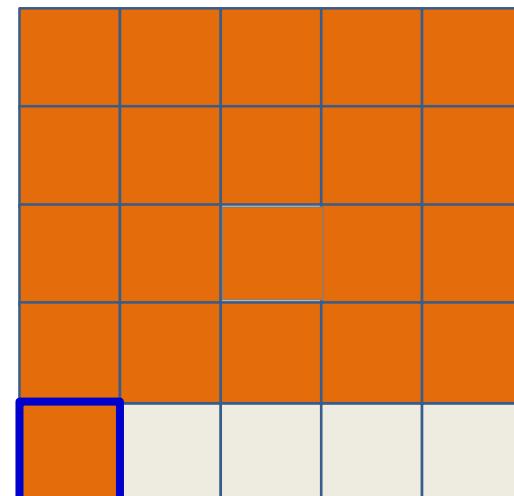


⋮

⋮



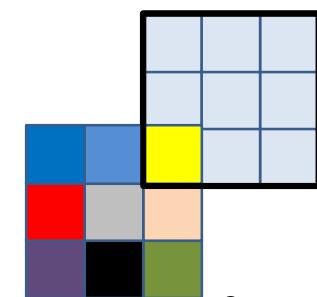
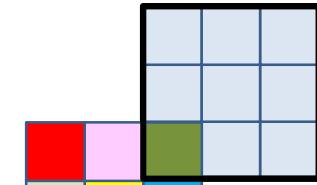
$w_l(m, n, K + 1 - x, K + 1 - y)$



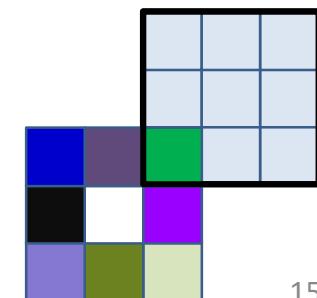
$$\frac{\partial \text{Div}}{\partial y(l-1, m, x, y)}$$

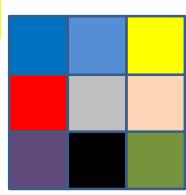
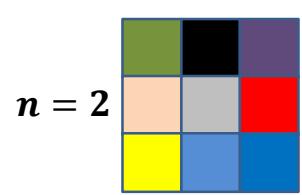
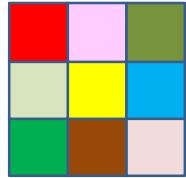
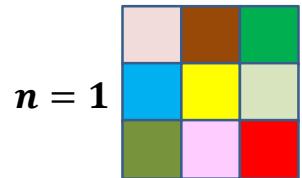
$$\frac{d\text{Div}}{dY(l-1, m, x, y)} = \sum_n \sum_{x', y'} \frac{d\text{Div}}{dz(l, n, x', y')} w_l(m, n, x - x', y - y')$$

=



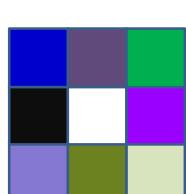
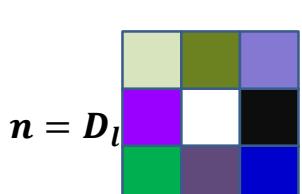
⋮



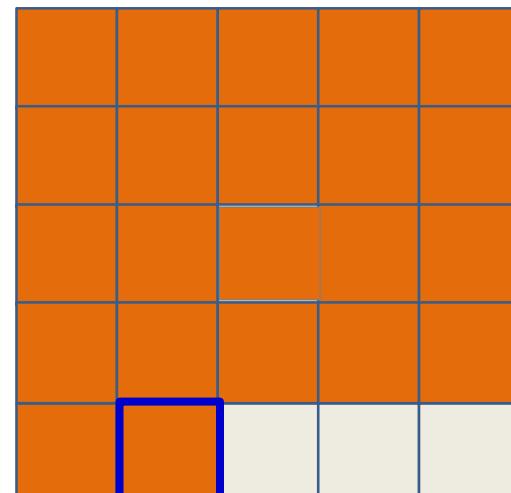
$w_l(m, n, x, y)$ 

⋮

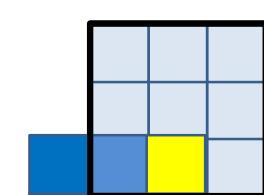
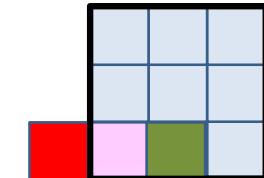
⋮



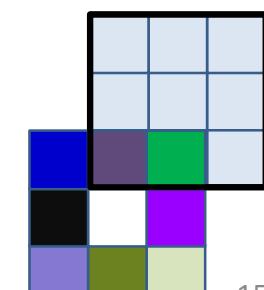
$w_l(m, n, K + 1 - x, K + 1 - y)$



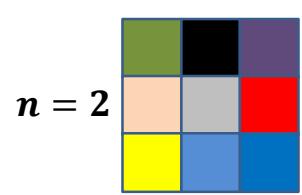
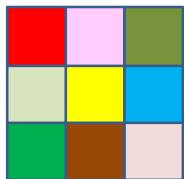
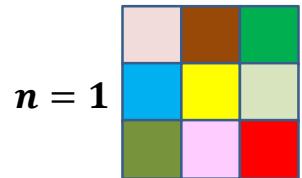
=



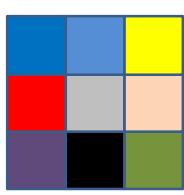
⋮  
⋮  
⋮



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

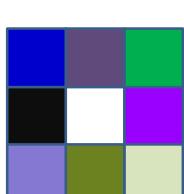
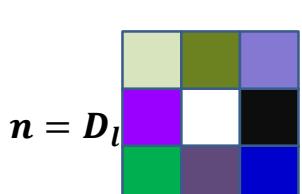
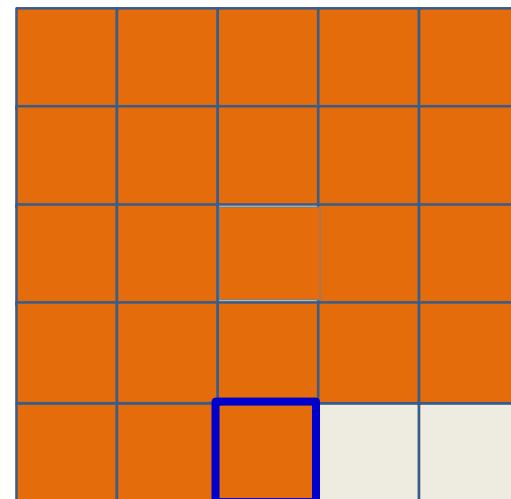
$w_l(m, n, x, y)$ 

flip

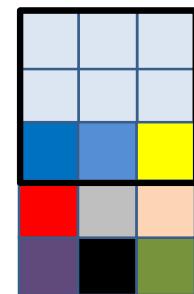
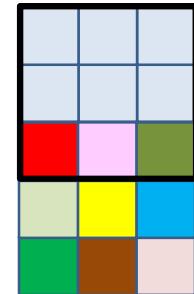


⋮

⋮

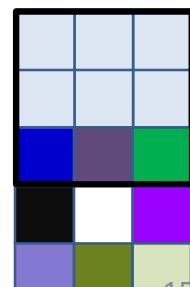
 $w_l(m, n, K + 1 - x, K + 1 - y)$ 

=



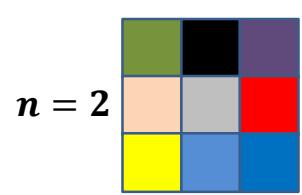
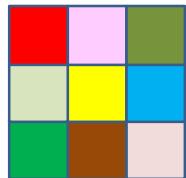
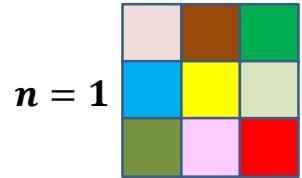
⋮

⋮

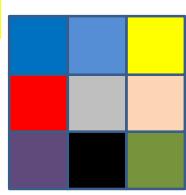


$$\frac{dDiv}{dy(l-1, m, x, y)} = \sum_n \sum_{x', y'} \frac{dDiv}{dz(l, n, x', y')} w_l(m, n, x - x', y - y')$$

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

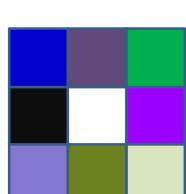
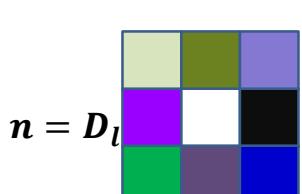


flip

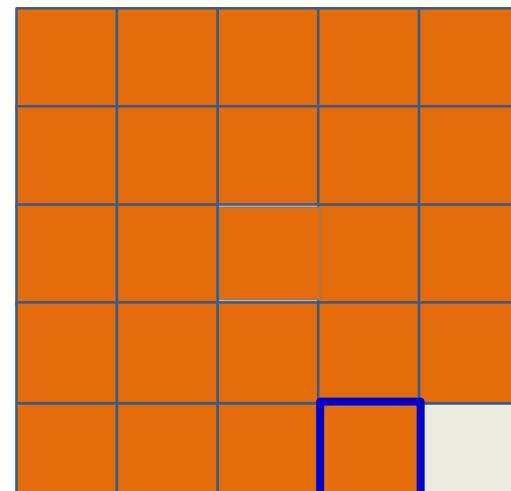


⋮

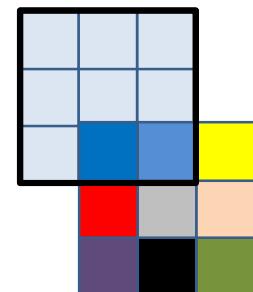
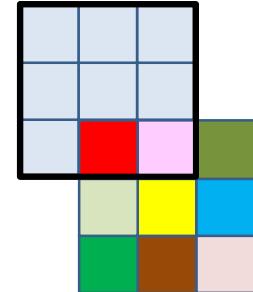
⋮



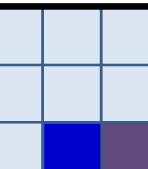
$w_l(m, n, K + 1 - x, K + 1 - y)$



=



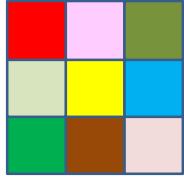
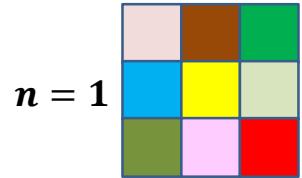
⋮



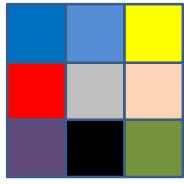
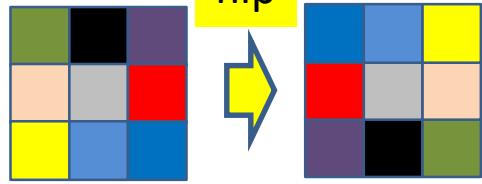
159

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

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



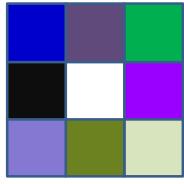
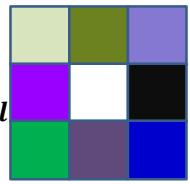
$n = 2$



⋮

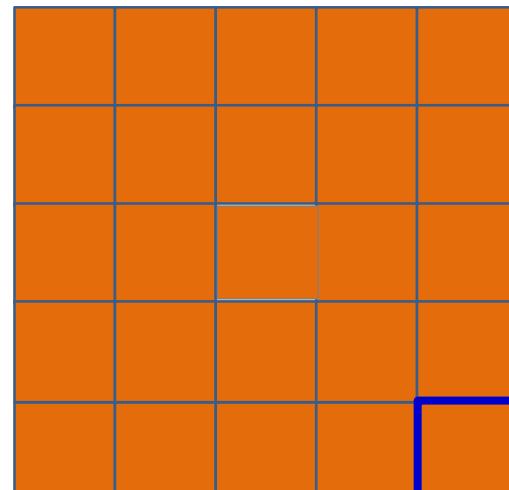
⋮

$n = D_l$

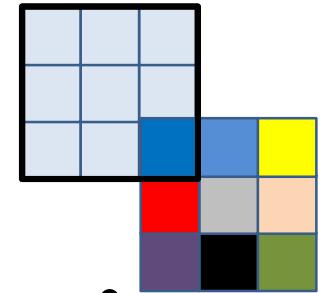
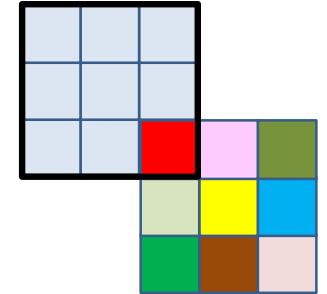


$w_l(m, n, K + 1 - x, K + 1 - y)$

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

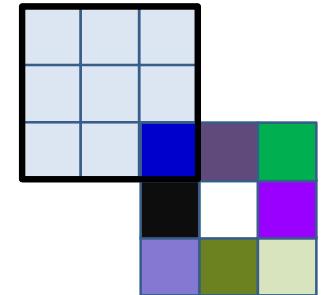


=

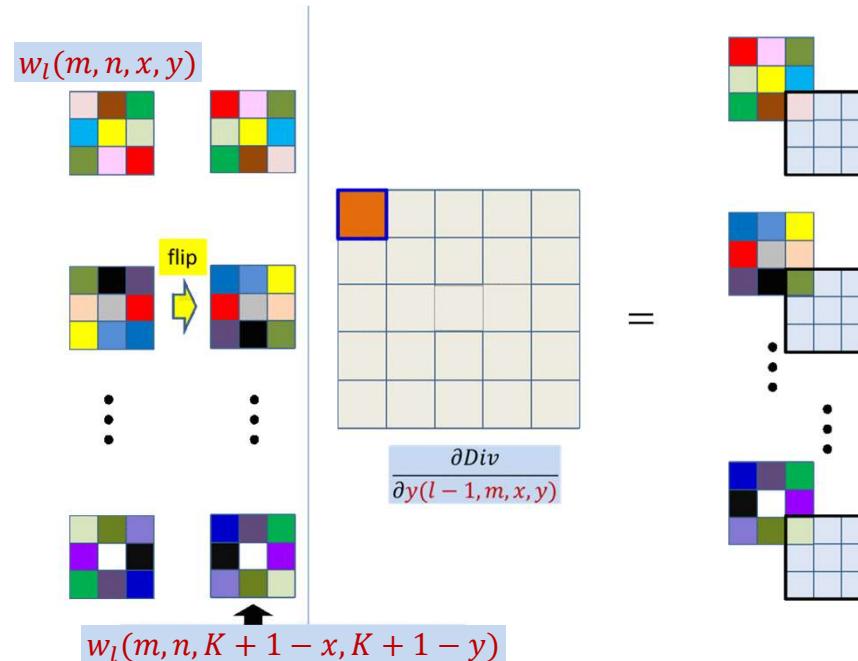


⋮

⋮

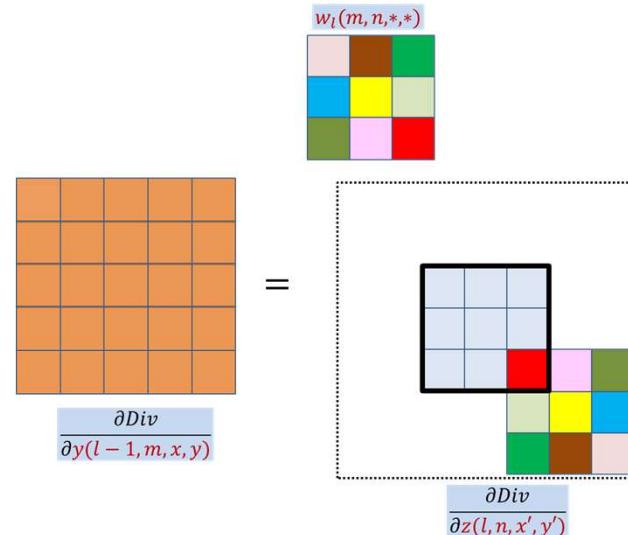


# Computing the derivative for $Y(l - 1, m)$



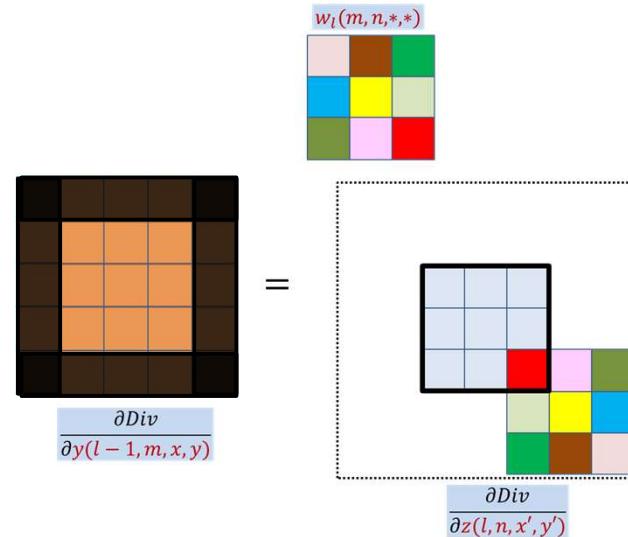
- This is just a convolution of the zero-padded maps by the transposed and flipped filter
  - After zero padding it first with  $K - 1$  zeros on every side

# The size of the Y-derivative map



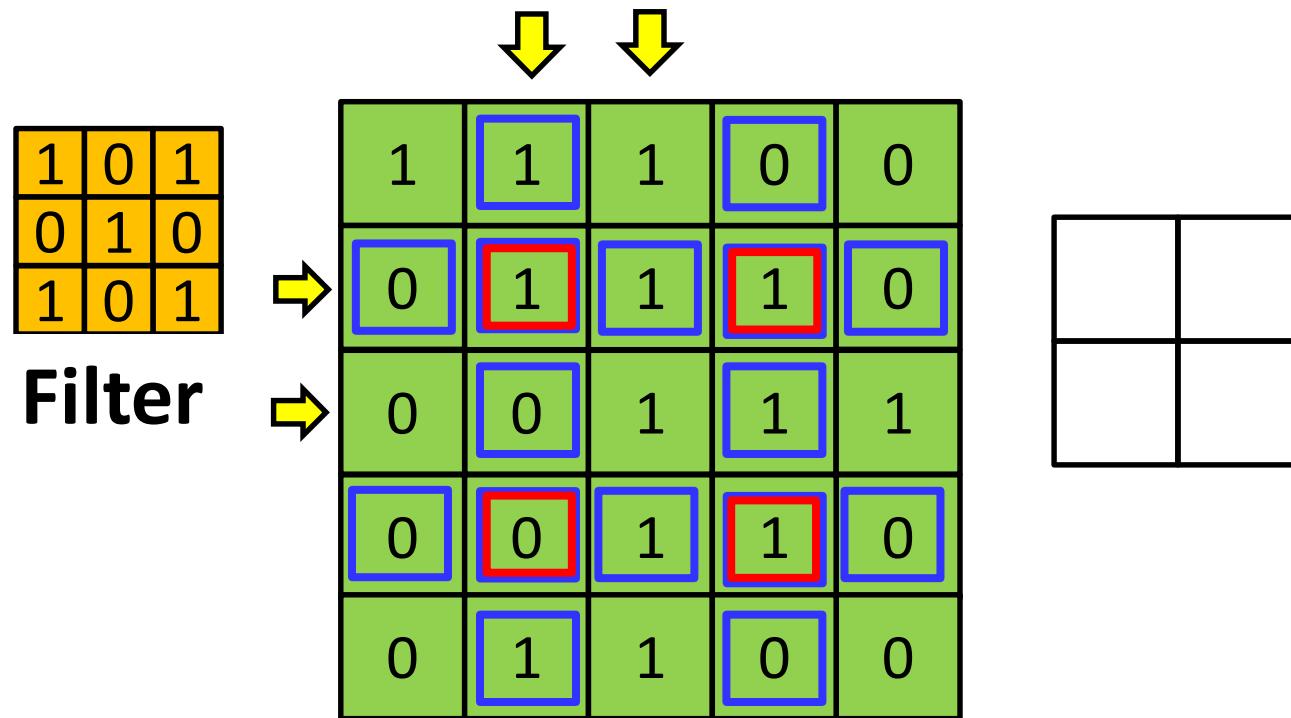
- We continue to compute elements for the derivative  $Y$  map as long as the (flipped) filter has at least one element in the (unpadded) derivative Zmap
  - I.e. so long as the  $Y$  derivative is non-zero
- The size of the  $Y$  derivative map will be  $(H + K - 1) \times (W + K - 1)$ 
  - $H$  and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



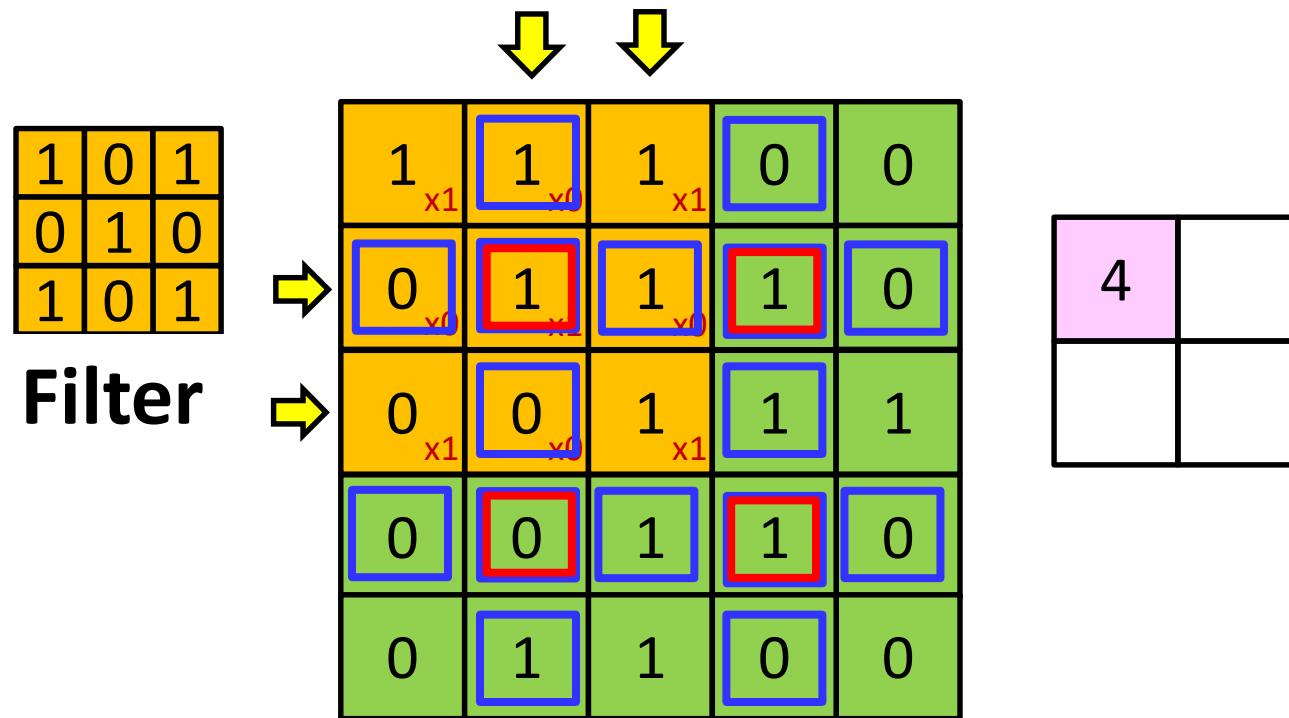
- If the  $Y$  map was zero-padded in the forward pass, the derivative map will be the size of the *zero-padded* map
  - The zero padding regions must be deleted before further backprop

# When the stride is more than 1?



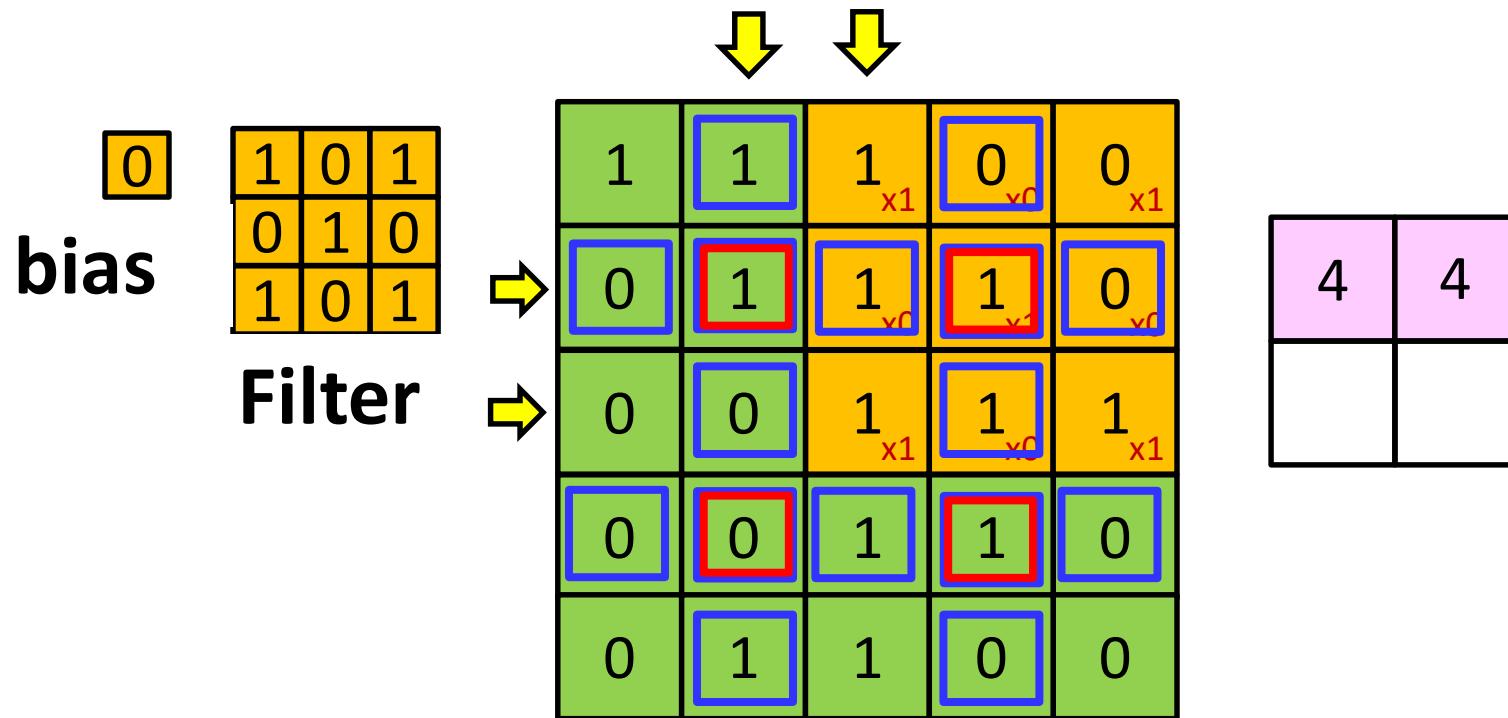
- When the stride is greater than 1, some positions of  $Y(l - 1, m)$  contribute to more locations on the  $Z(l, n)$  maps than others
  - With a stride of 2, the boxed-in-blue  $Y(l - 1, m)$  locations contribute to half as many  $Z(l, n)$  locations as the unboxed locations
  - The double-boxed (blue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



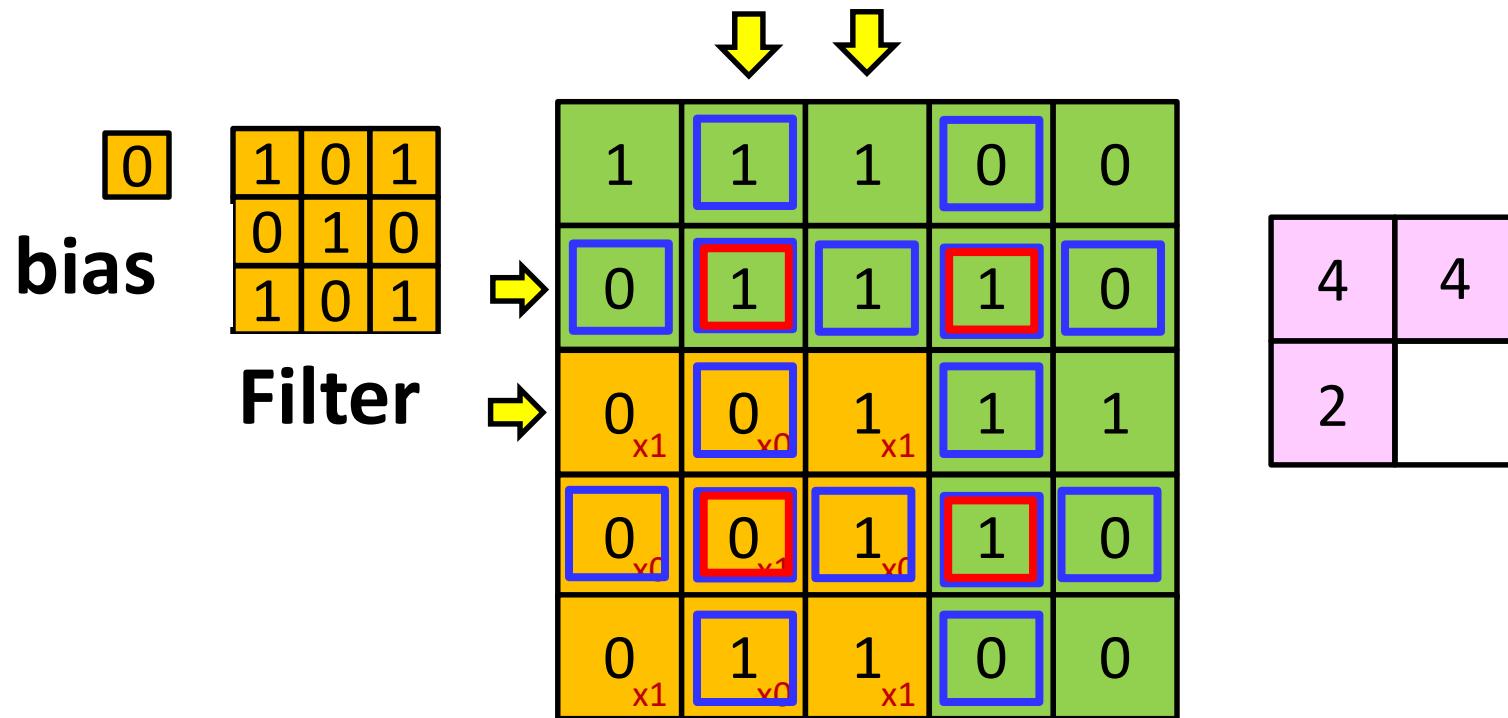
- When the stride is greater than 1, some positions of  $Y(l - 1, m)$  contribute to more locations on the  $Z(l, n)$  maps than others
  - With a stride of 2, the boxed-in-blue  $Y(l - 1, m)$  locations contribute to half as many  $Z(l, n)$  locations as the unboxed locations
  - The double-boxed (blue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



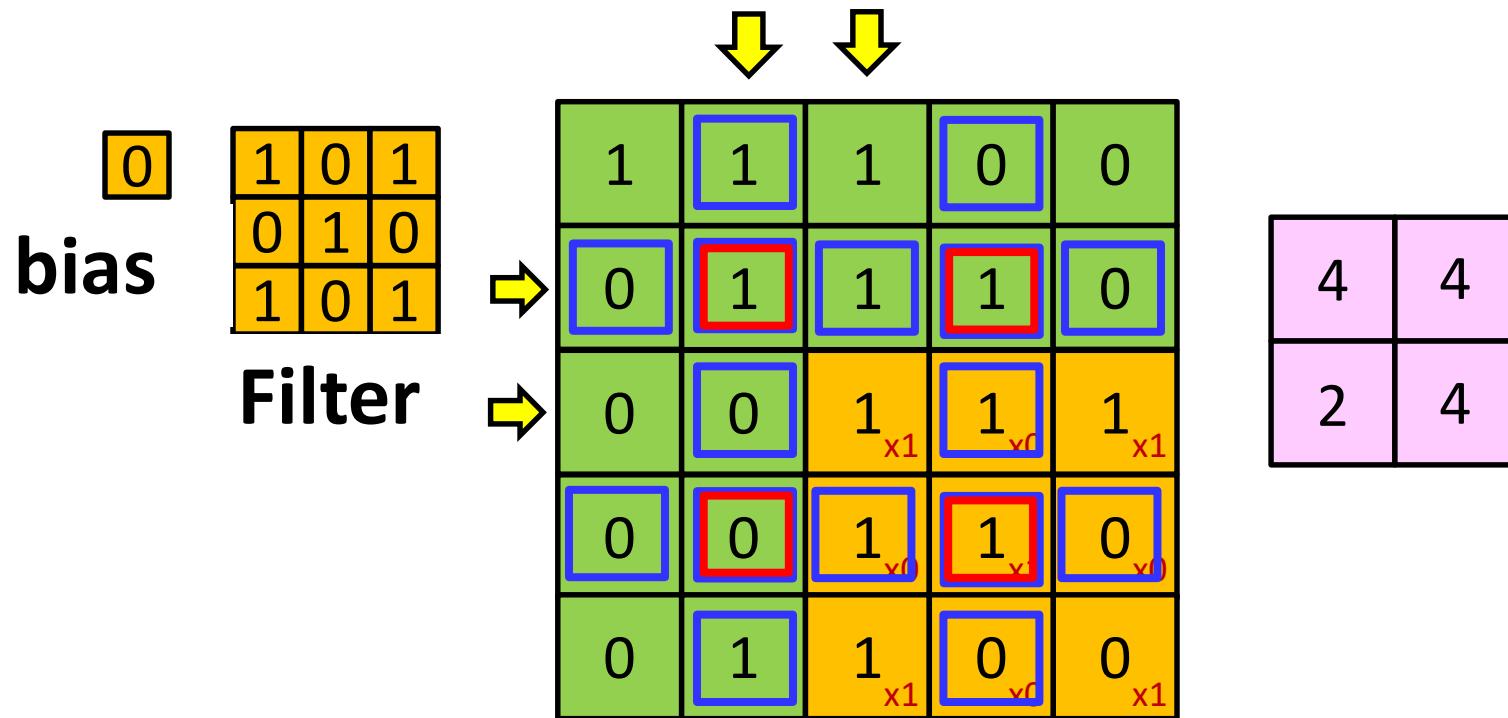
- When the stride is greater than 1, some positions of  $Y(l - 1, m)$  contribute to more locations on the  $Z(l, n)$  maps than others
  - With a stride of 2, the boxed-in-blue  $Y(l - 1, m)$  locations contribute to half as many  $Z(l, n)$  locations as the unboxed locations
  - The double-boxed (blue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



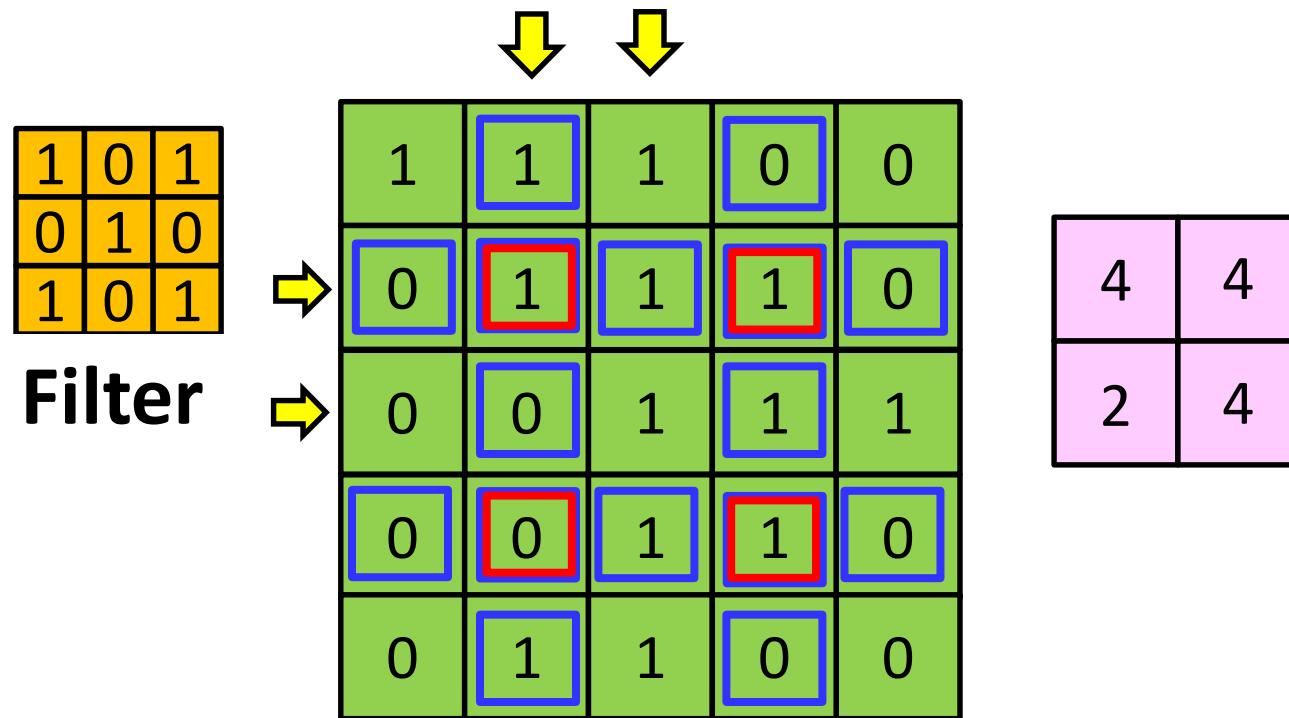
- When the stride is greater than 1, some positions of  $Y(l - 1, m)$  contribute to more locations on the  $Z(l, n)$  maps than others
  - With a stride of 2, the boxed-in-blue  $Y(l - 1, m)$  locations contribute to half as many  $Z(l, n)$  locations as the unboxed locations
  - The double-boxed (blue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



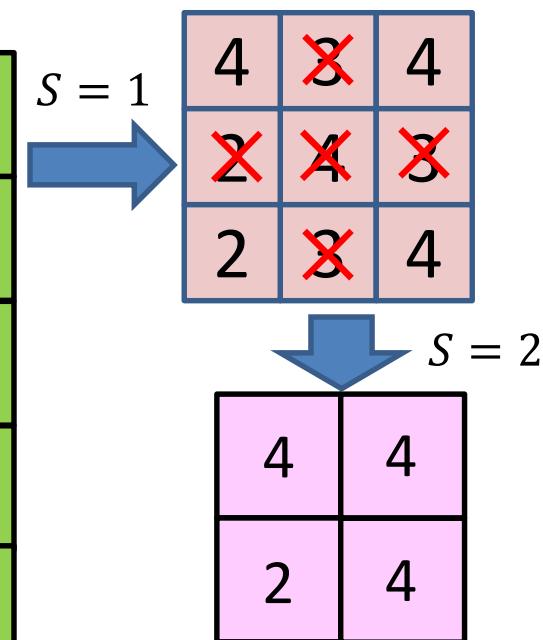
- We must make adjustments for when the stride is greater than 1.

# Stride greater than 1

1	0	1
0	1	0
1	0	1

**Filter**

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



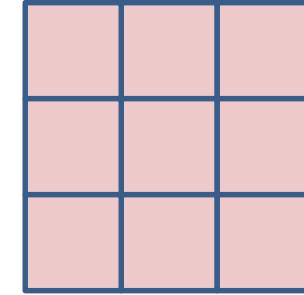
- **Observation:** Convolving with a stride  $S$  greater than 1 is the same as convolving with stride 1 and “dropping”  $S - 1$  out of every  $S$  rows, and  $S - 1$  of every  $S$  columns
  - **Downsampling by  $S$**
  - E.g. for stride 2, it is the same as convolving with stride 1 and dropping every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

# Derivatives with Stride greater than 1

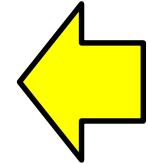
1	0	1
0	1	0
1	0	1

Filter

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



$\frac{dDiv}{dz(0,0)}$	$\frac{dDiv}{dz(1,0)}$
$\frac{dDiv}{dz(0,1)}$	$\frac{dDiv}{dz(1,1)}$



- **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

1	0	1
0	1	0
1	0	1

Filter

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

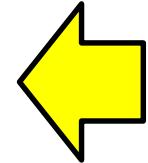
$dDiv$		$dDiv$
$dz(0,0)$		$dz(1,0)$
$dDiv$		$dDiv$
$dz(0,1)$		$dz(1,1)$

$\uparrow S = 2$

$dDiv$	$dDiv$
$dz(0,0)$	$dz(1,0)$

$dDiv$	$dDiv$
$dz(0,1)$	$dz(1,1)$



- **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

1	0	1
0	1	0
1	0	1

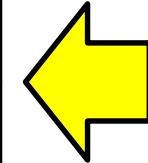
Filter

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

$dDiv$ $dz(0,0)$	0	$dDiv$ $dz(1,0)$
0	0	0
$dDiv$ $dz(0,1)$	0	$dDiv$ $dz(1,1)$

$\uparrow S = 2$

$dDiv$ $dz(0,0)$	$dDiv$ $dz(1,0)$
$dDiv$ $dz(0,1)$	$dDiv$ $dz(1,1)$



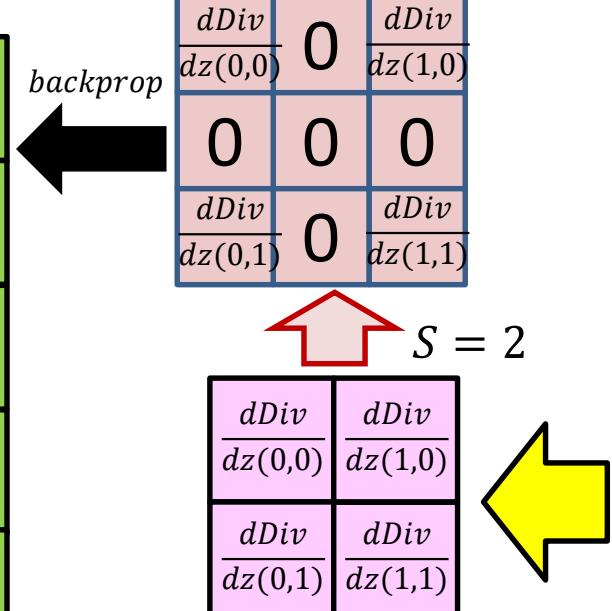
- **Derivatives:** Backprop gives us the derivatives of the divergence with respect to the elements of the *downsampled* (strided)  $Z$  map
- We can place these values back into their original locations of the full-sized  $Z$  map
- The remaining entries of the  $Z$  map do not affect the divergence
  - Since they get dropped out
- The derivative of the divergence w.r.t. these values is 0

# Computing derivatives with Stride > 1

1	0	1
0	1	0
1	0	1

**Filter**

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



- **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

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

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

- 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}xK_lxK_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: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

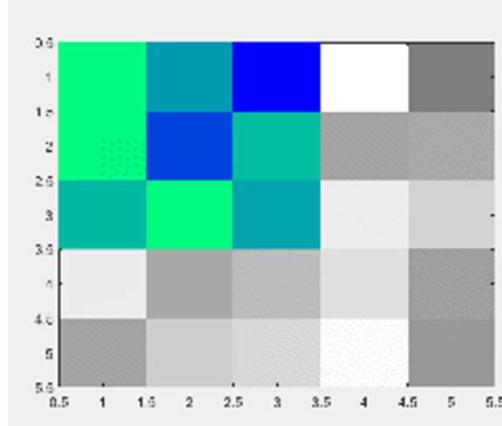
```
# Upsample dz to the size it would be if stride was 1
function upsample(dz, S, W, H, K)
    if (S > 1)    #Insert S-1 zeros between samples
        Hup = H - K + 1
        Wup = W - K + 1
        dzup = zeros(Wup, Hup)
        for x = 1:H
            for y = 1:W
                dzup((x-1)S+1, (y-1)S+1)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

# Backpropagating through affine map

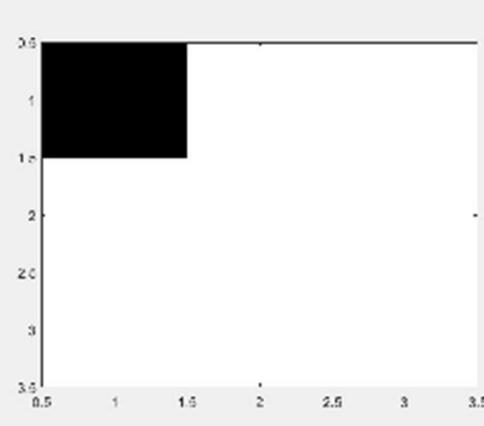
- Forward affine computation:
  - Compute affine maps  $z(l, n, x, y)$  from previous layer maps  $y(l - 1, m, x, y)$  and filters  $w_l(m, n, x, y)$
- Backpropagation: Given  $\frac{dDiv}{dz(l,n,x,y)}$ 
  - ✓ Compute derivative w.r.t.  $y(l - 1, m, x, y)$ 
    - Compute derivative w.r.t.  $w_l(m, n, x, y)$

# The derivatives for the weights

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



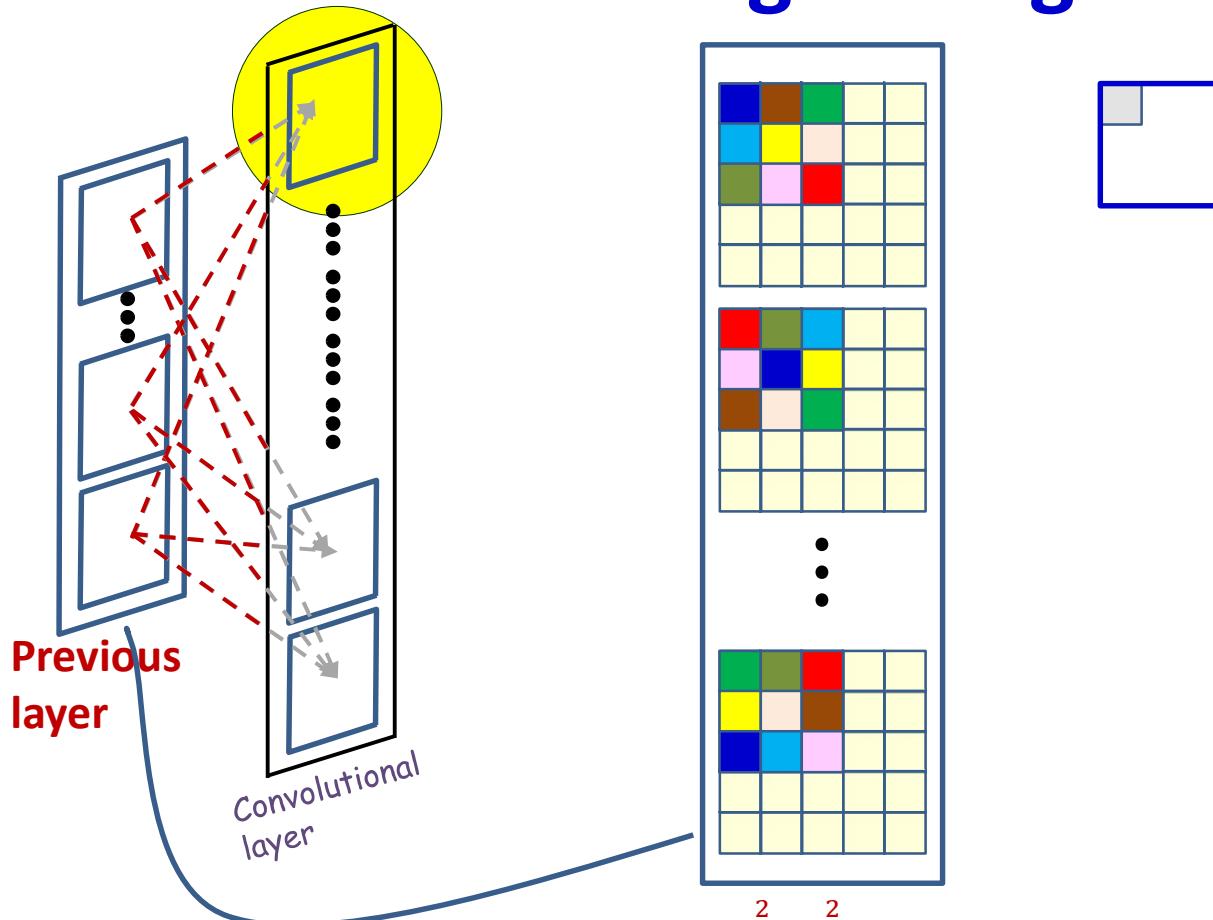
$Z(l, n)$



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

- 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.  $w_l(m, n, 1, 2)$ )

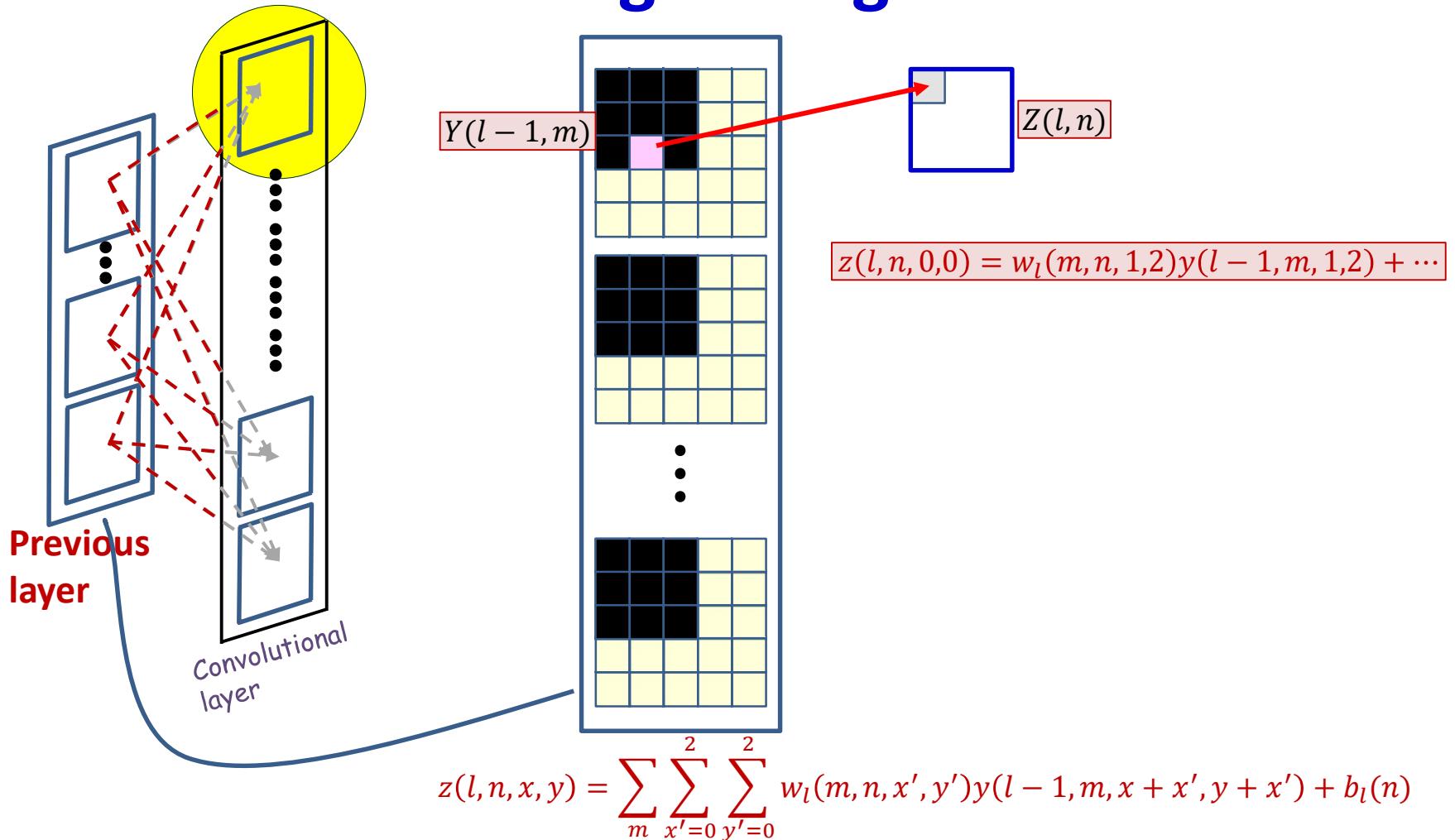
# Convolution: the contribution of a single weight



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

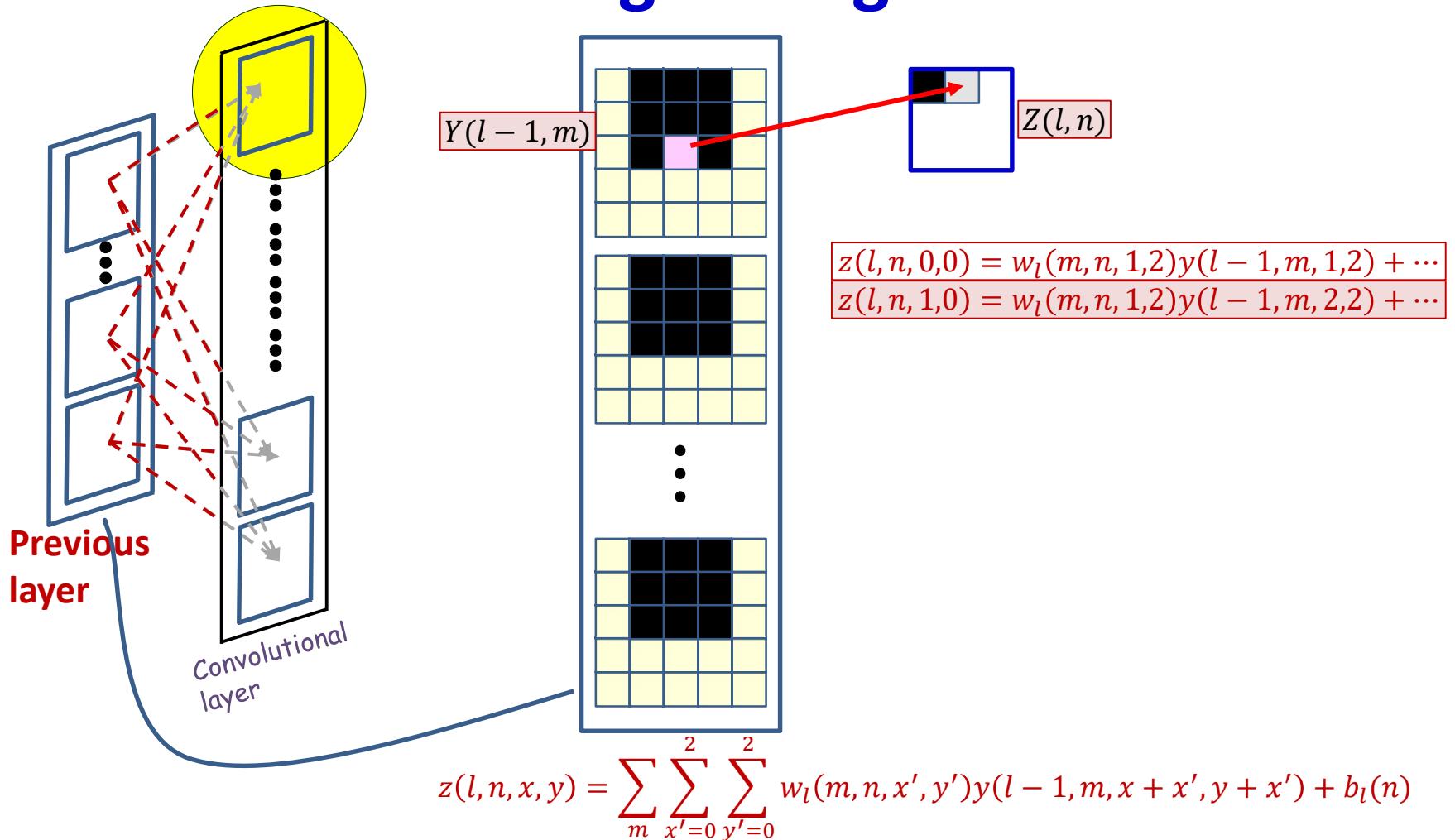
- Each affine output is computed from multiple input maps simultaneously
- Each **weight**  $w_l(m, n, i, j)$  affects several  $z(l, n, x, y)$

# Convolution: the contribution of a single weight



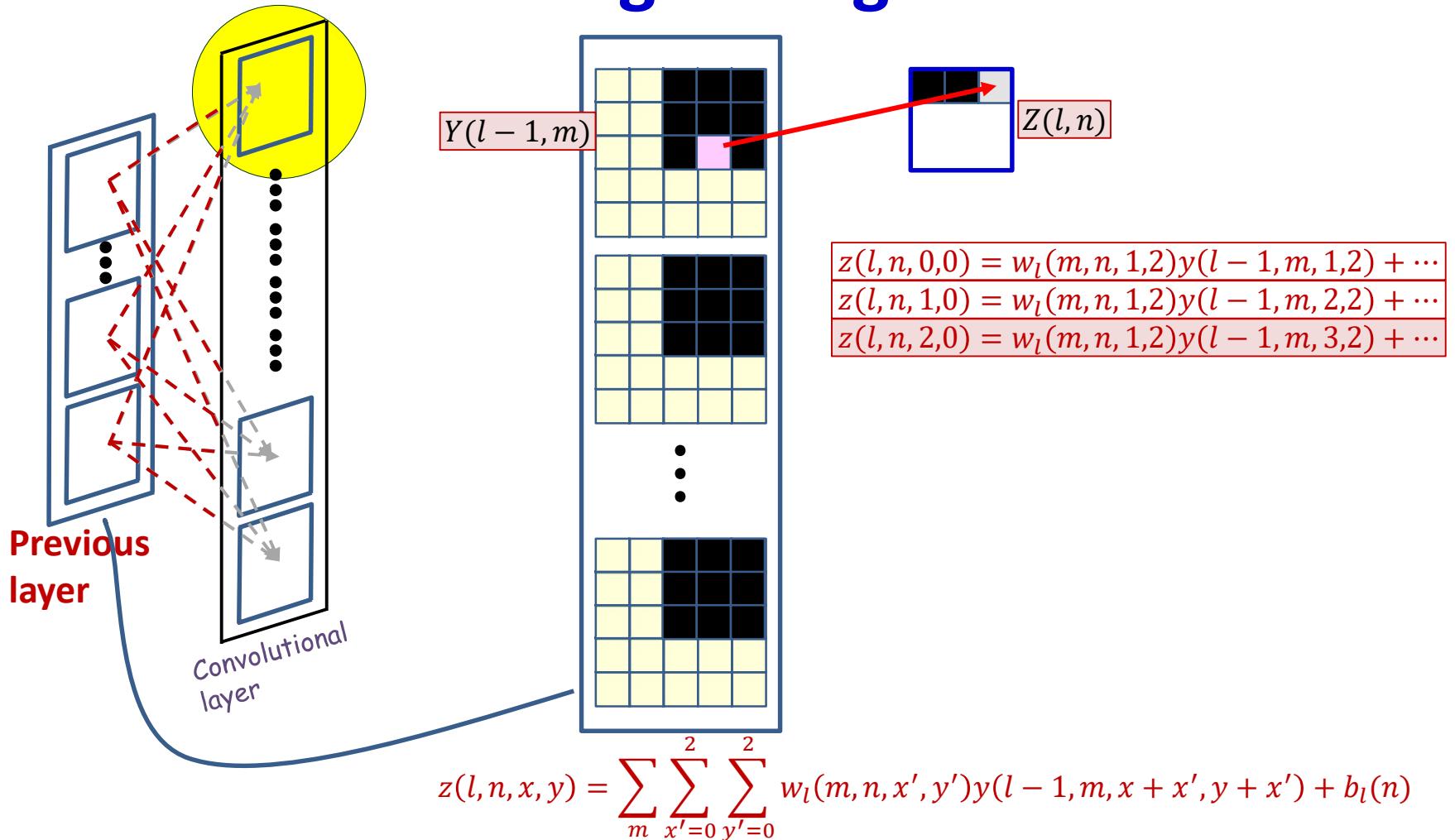
- Each weight  $w_l(m, n, i, j)$  affects several  $z(l, n, x, y)$ 
  - Consider the contribution of one filter components: e.g.  $w_l(m, n, 1, 2)$

# Convolution: the contribution of a single weight



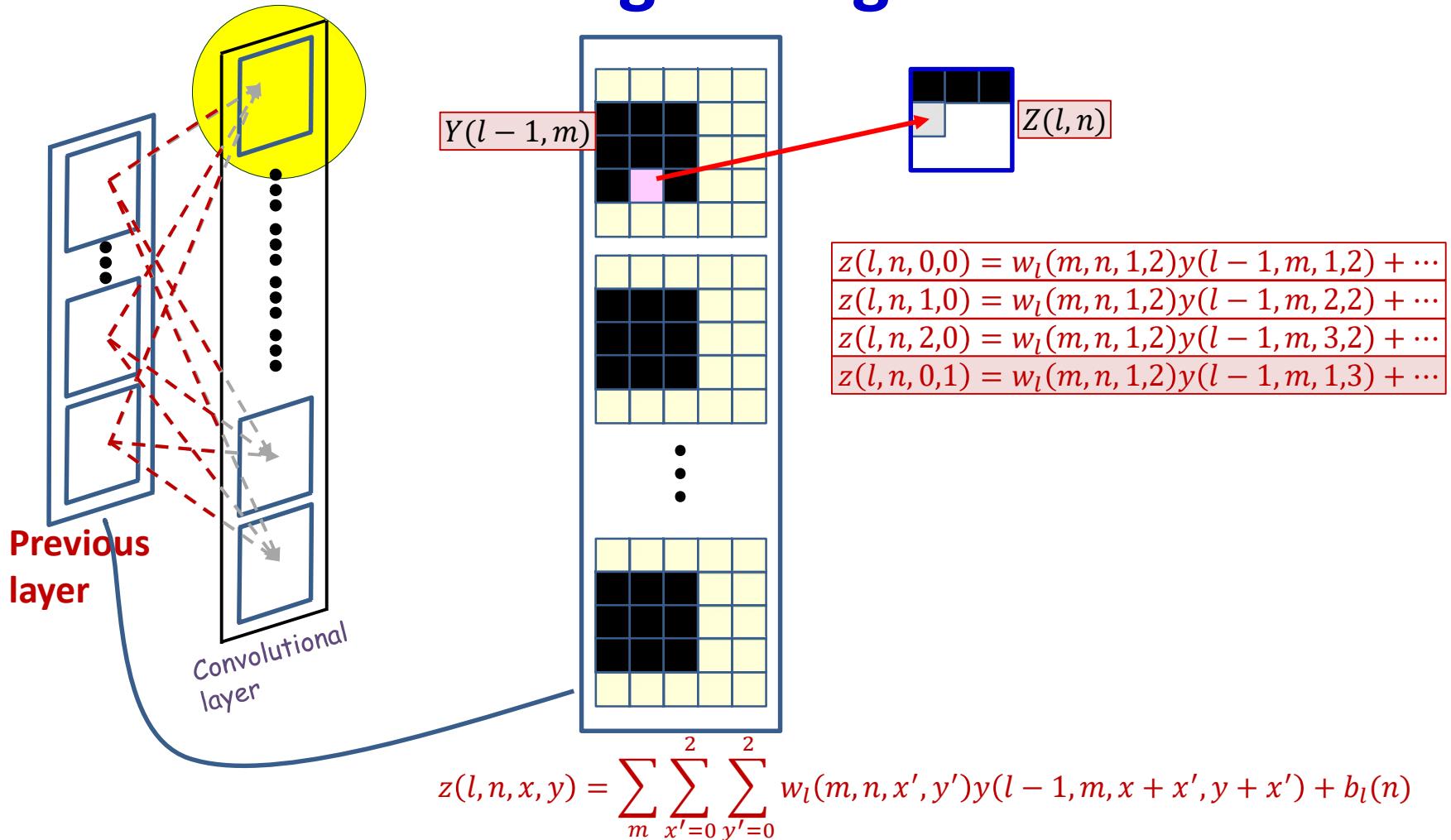
- Each weight  $w_l(m, n, i, j)$  affects several  $z(l, n, x, y)$ 
  - Consider the contribution of one filter components: e.g.  $w_l(m, n, 1, 2)$

# Convolution: the contribution of a single weight



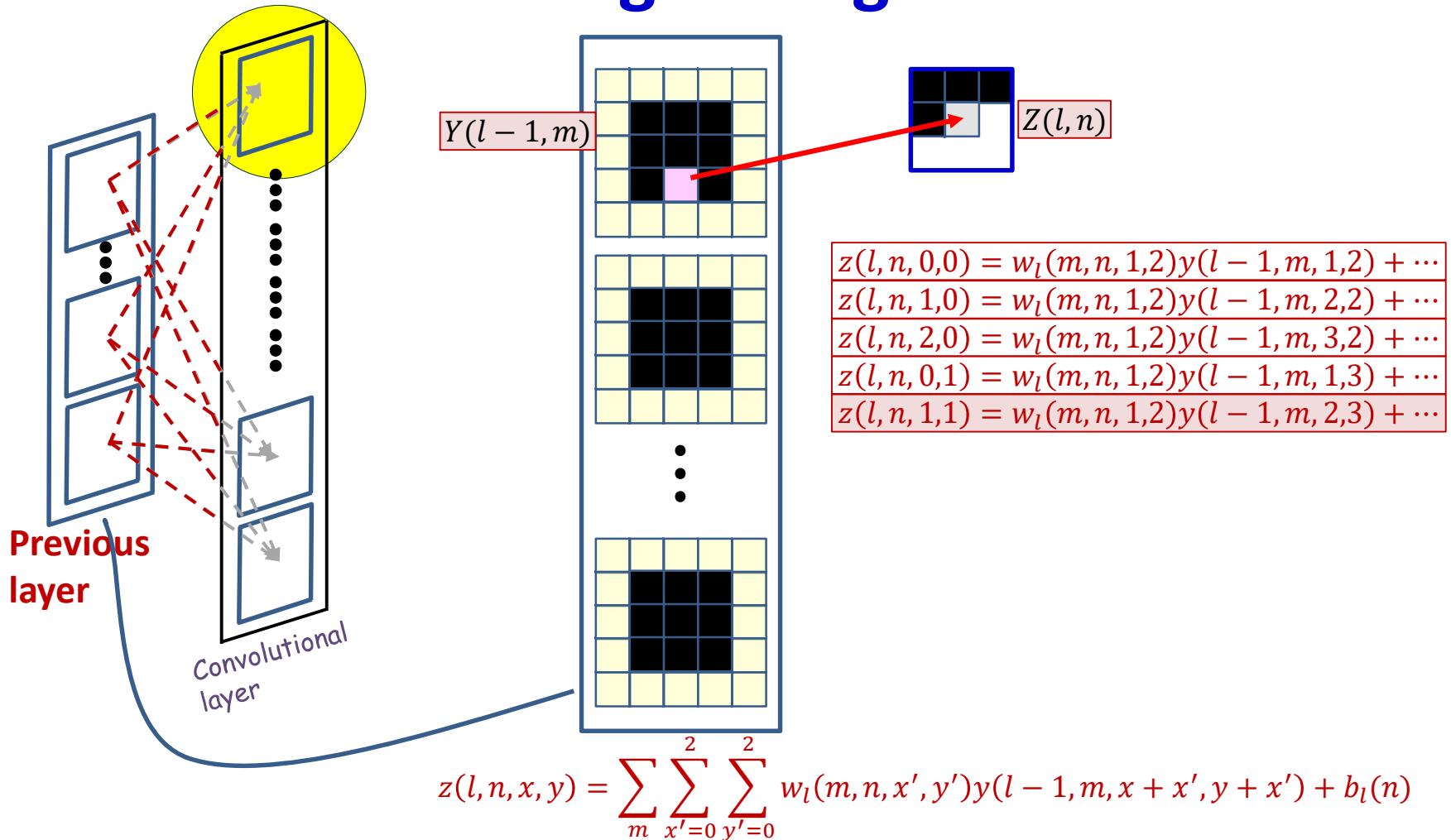
- Each weight  $w_l(m, n, i, j)$  affects several  $z(l, n, x, y)$ 
  - Consider the contribution of one filter components: e.g.  $w_l(m, n, 1, 2)$

# Convolution: the contribution of a single weight



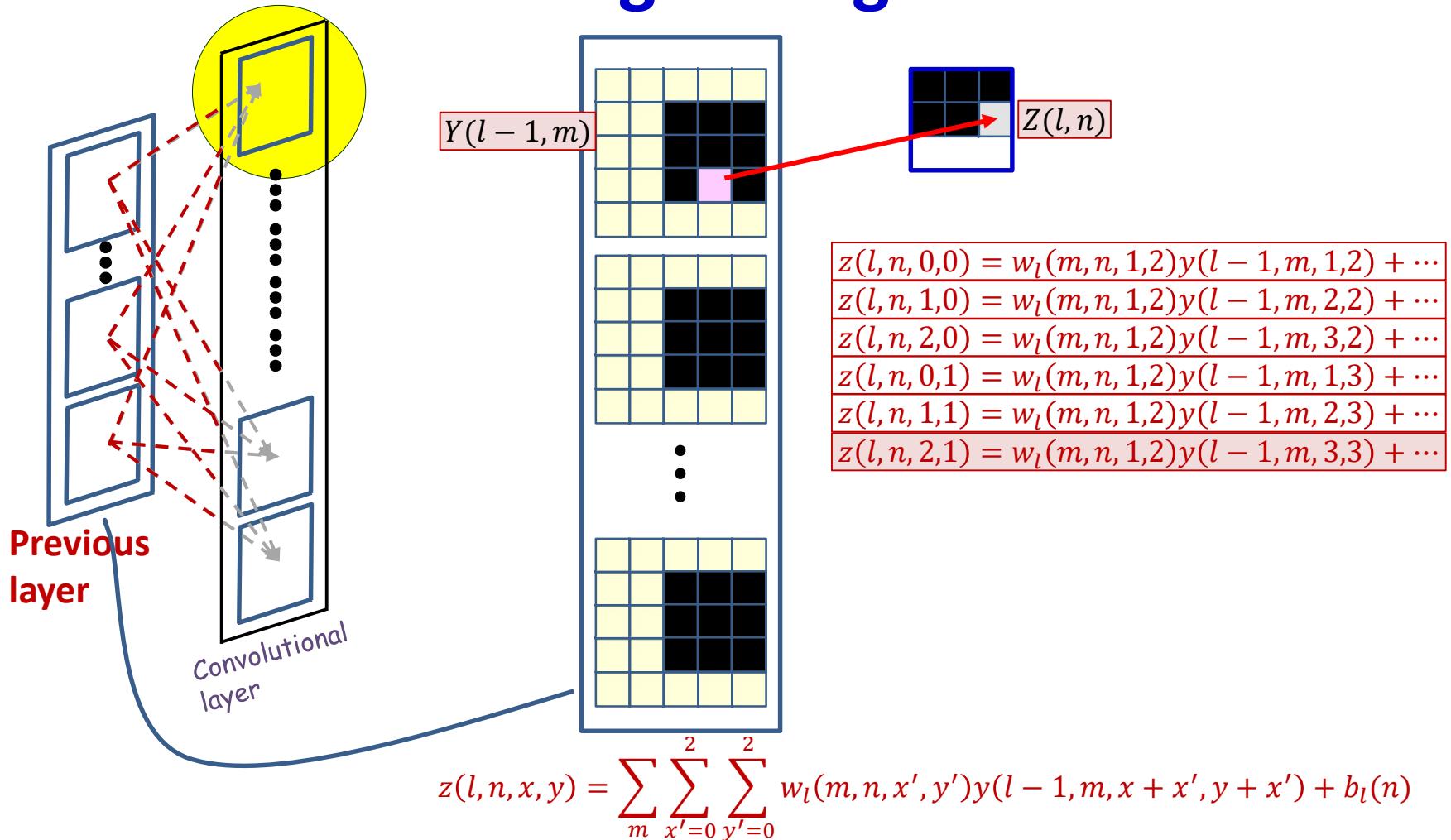
- Each weight  $w_l(m, n, i, j)$  affects several  $z(l, n, x, y)$ 
  - Consider the contribution of one filter components: e.g.  $w_l(m, n, 1, 2)$

# Convolution: the contribution of a single weight



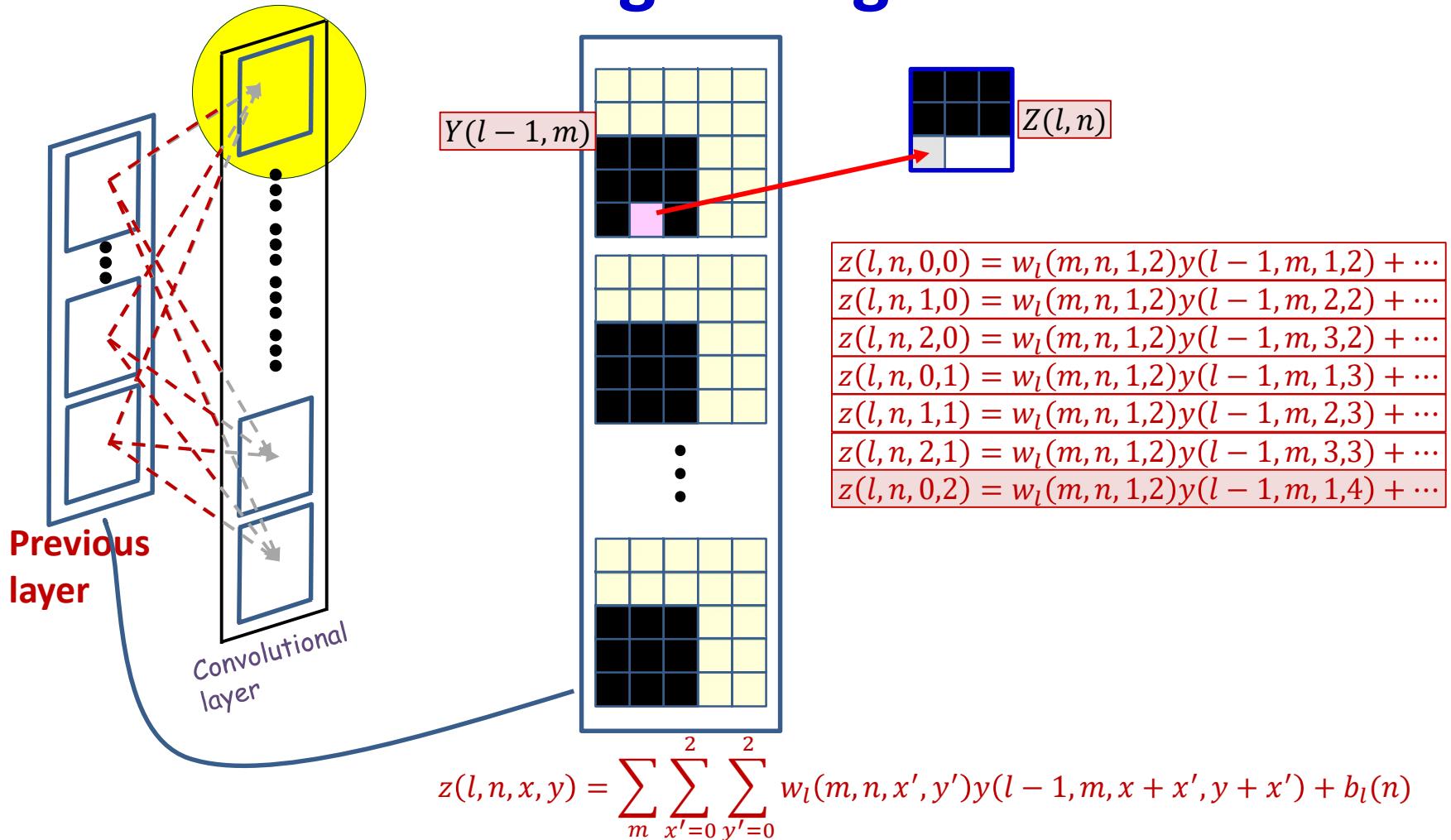
- Each weight  $w_l(m, n, i, j)$  affects several  $z(l, n, x, y)$ 
  - Consider the contribution of one filter components: e.g.  $w_l(m, n, 1, 2)$

# Convolution: the contribution of a single weight



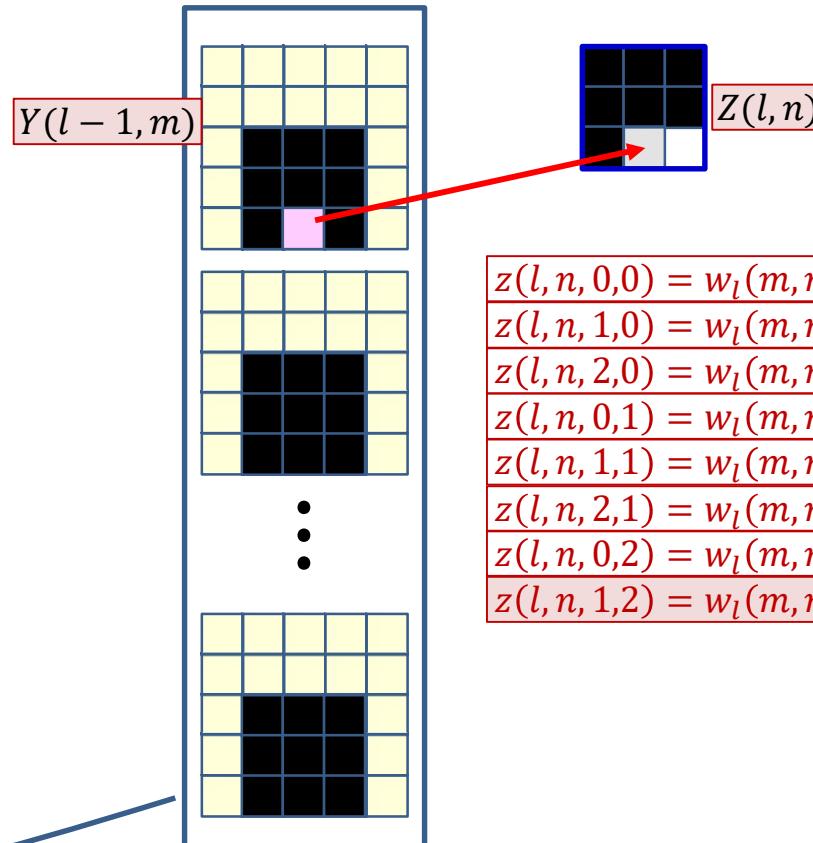
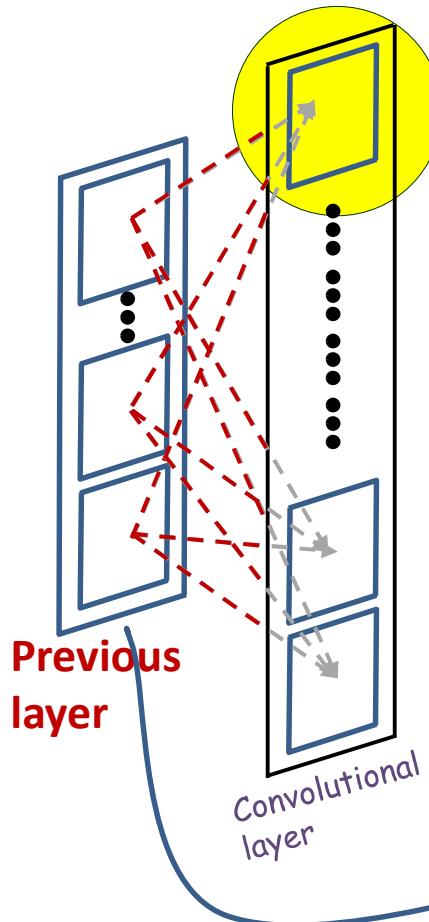
- Each weight  $w_l(m, n, i, j)$  affects several  $z(l, n, x, y)$ 
  - Consider the contribution of one filter components: e.g.  $w_l(m, n, 1, 2)$

# Convolution: the contribution of a single weight



- 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

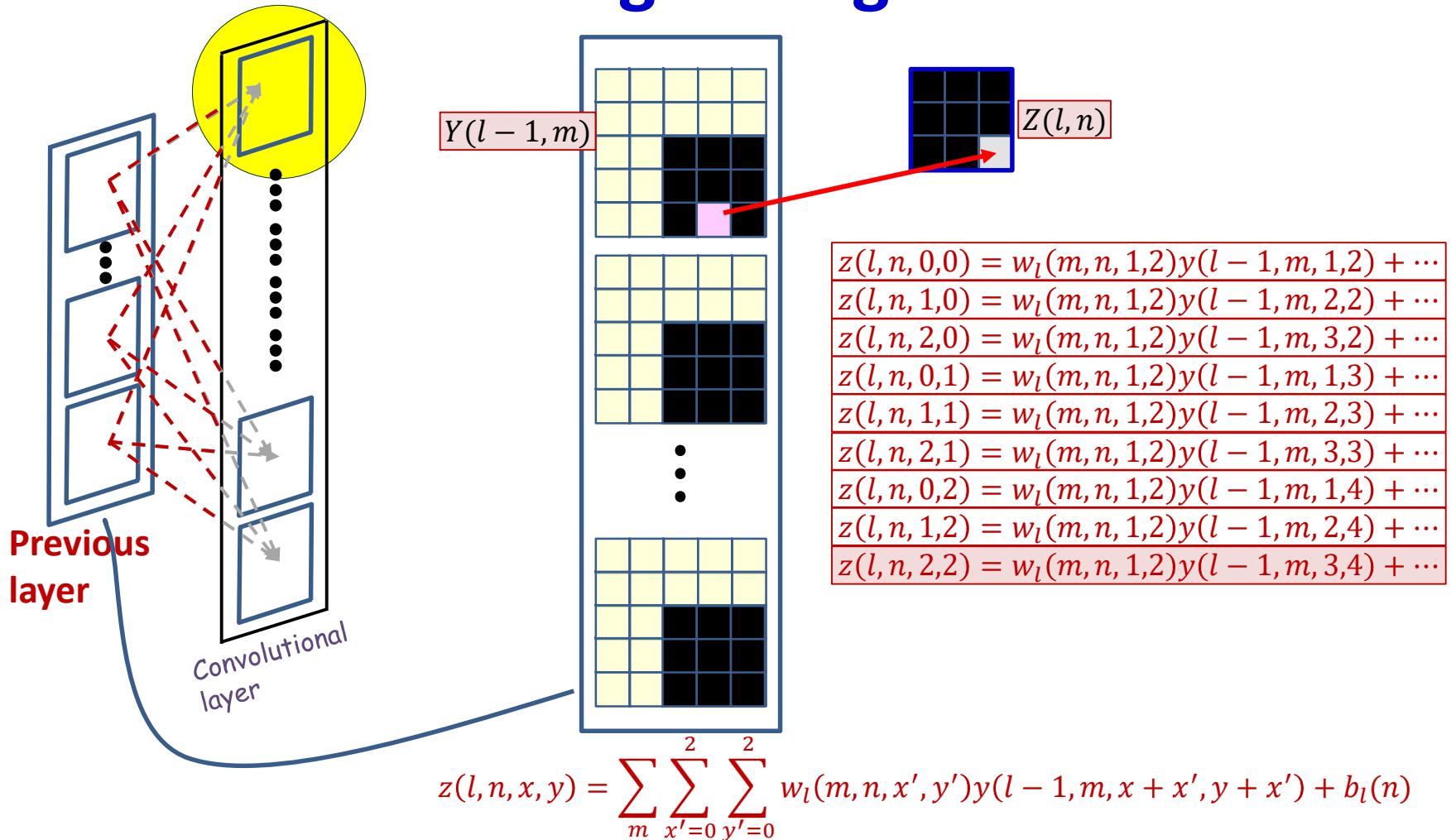


$$\begin{aligned}
 z(l, n, 0, 0) &= w_l(m, n, 1, 2)y(l - 1, m, 1, 2) + \dots \\
 z(l, n, 1, 0) &= w_l(m, n, 1, 2)y(l - 1, m, 2, 2) + \dots \\
 z(l, n, 2, 0) &= w_l(m, n, 1, 2)y(l - 1, m, 3, 2) + \dots \\
 z(l, n, 0, 1) &= w_l(m, n, 1, 2)y(l - 1, m, 1, 3) + \dots \\
 z(l, n, 1, 1) &= w_l(m, n, 1, 2)y(l - 1, m, 2, 3) + \dots \\
 z(l, n, 2, 1) &= w_l(m, n, 1, 2)y(l - 1, m, 3, 3) + \dots \\
 z(l, n, 0, 2) &= w_l(m, n, 1, 2)y(l - 1, m, 1, 4) + \dots \\
 z(l, n, 1, 2) &= w_l(m, n, 1, 2)y(l - 1, m, 2, 4) + \dots
 \end{aligned}$$

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

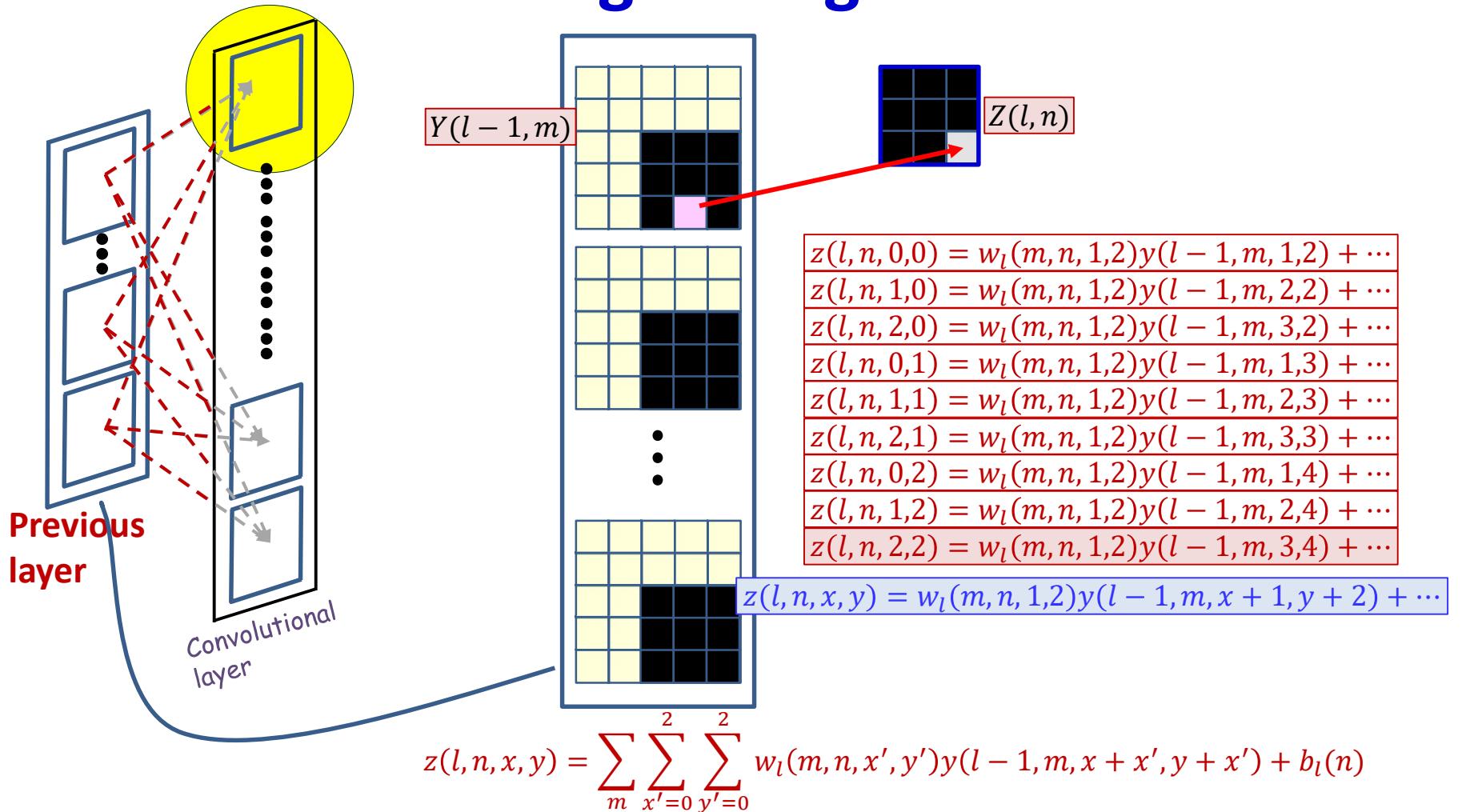
- Each weight  $w_l(m, n, i, j)$  affects several  $z(l, n, x, y)$ 
  - Consider the contribution of one filter components: e.g.  $w_l(m, n, 1, 2)$

# Convolution: the contribution of a single weight



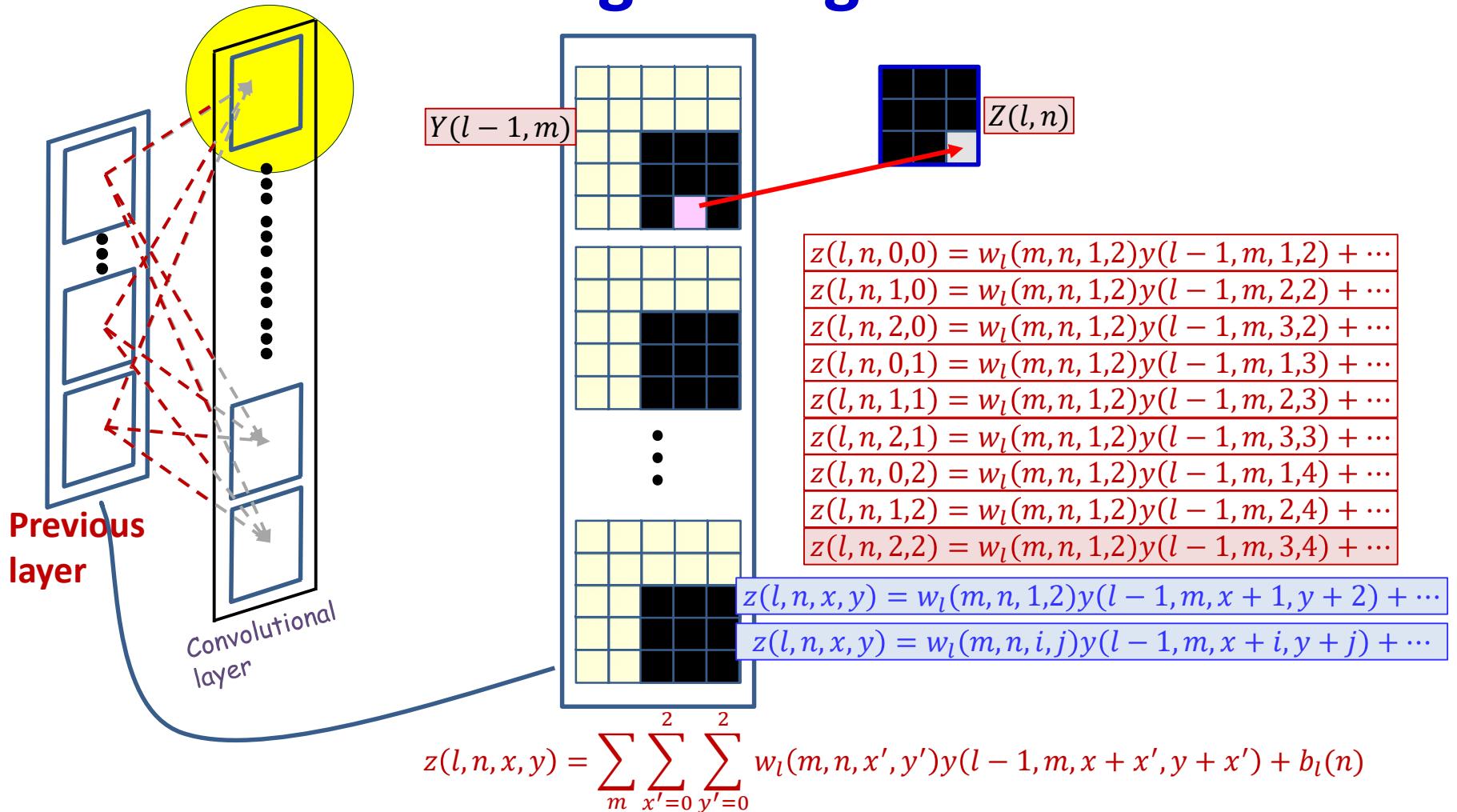
- Each weight  $w_l(m, n, i, j)$  affects several  $z(l, n, x, y)$ 
  - Consider the contribution of one filter components: e.g.  $w_l(m, n, 1, 2)$

# Convolution: the contribution of a single weight



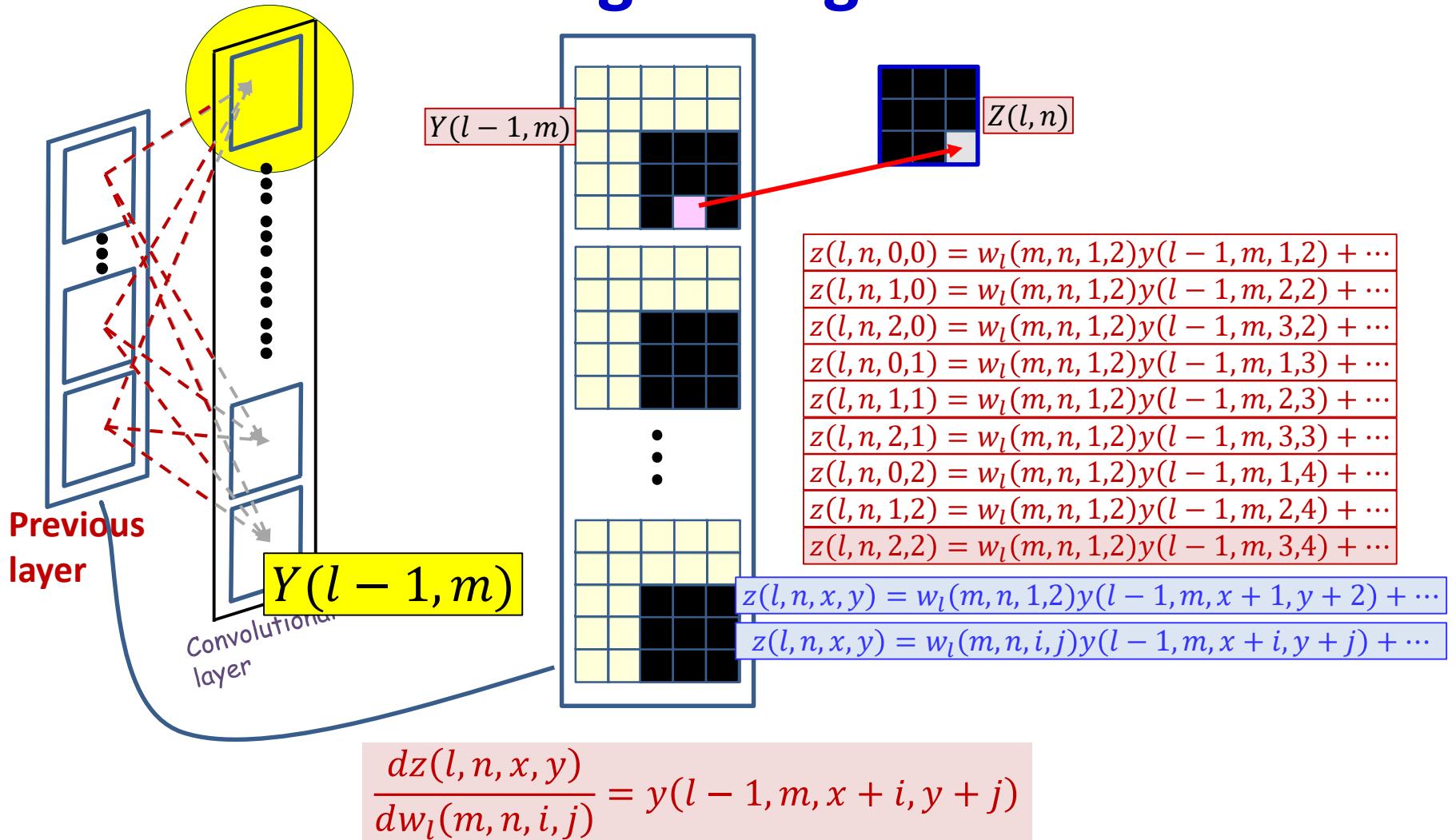
- Each weight  $w_l(m, n, i, j)$  affects several  $z(l, n, x, y)$ 
  - Consider the contribution of one filter components: e.g.  $w_l(m, n, 1,2)$

# Convolution: the contribution of a single weight

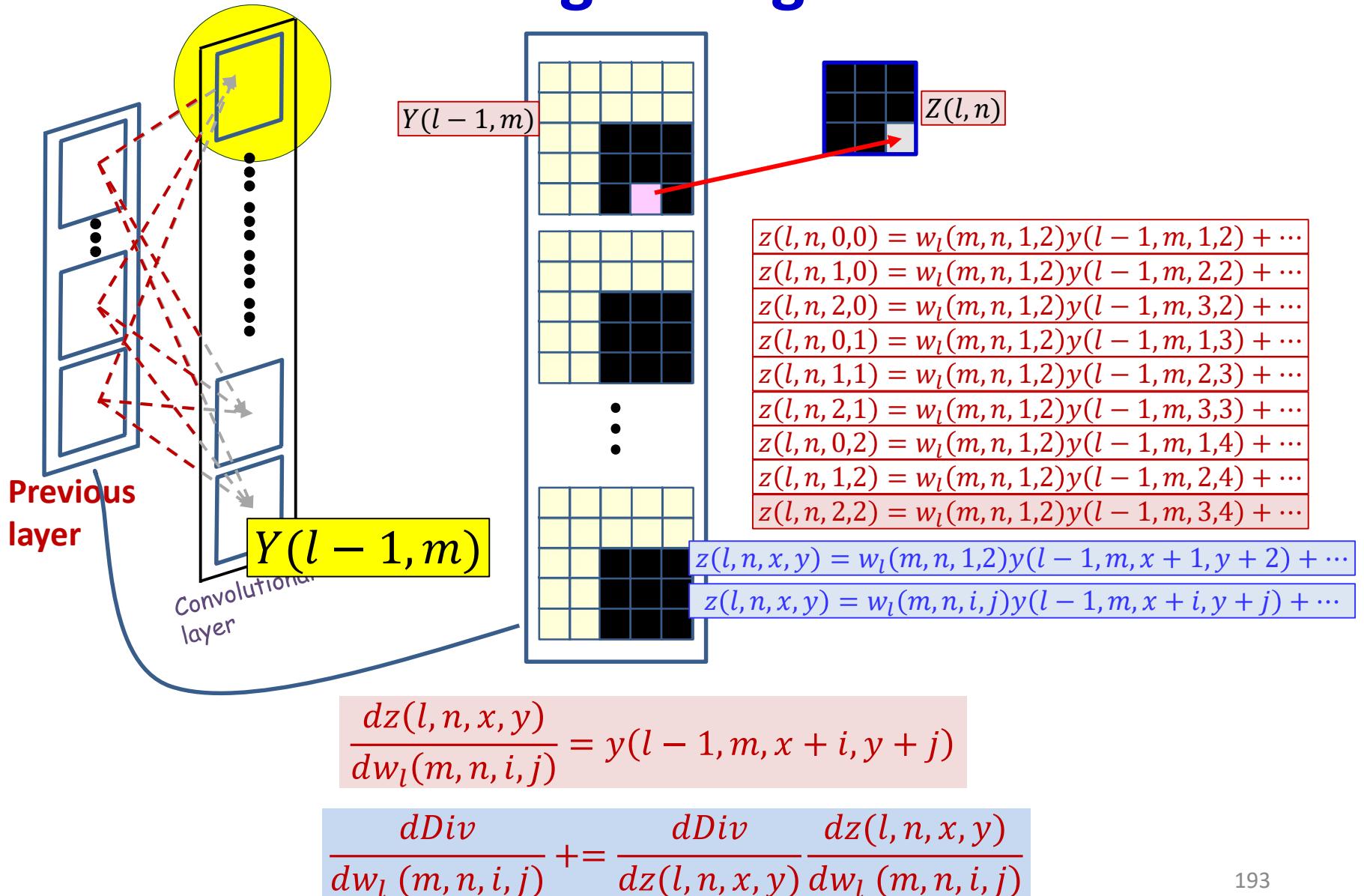


- Each weight  $w_l(m, n, i, j)$  affects several  $z(l, n, x, y)$ 
  - Consider the contribution of one filter components: e.g.  $w_l(m, n, 1,2)_{\text{91}}$

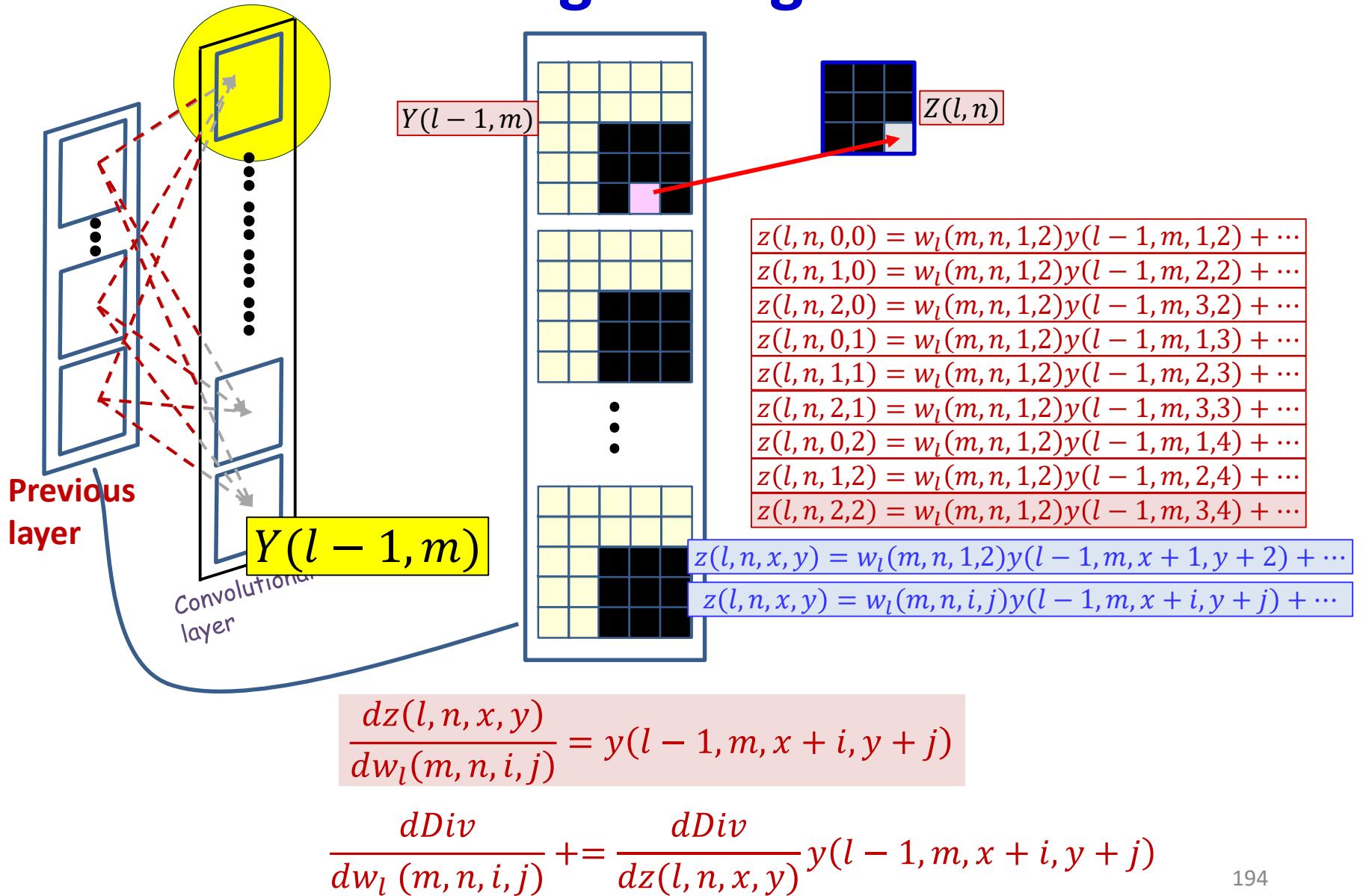
# Convolution: the contribution of a single weight



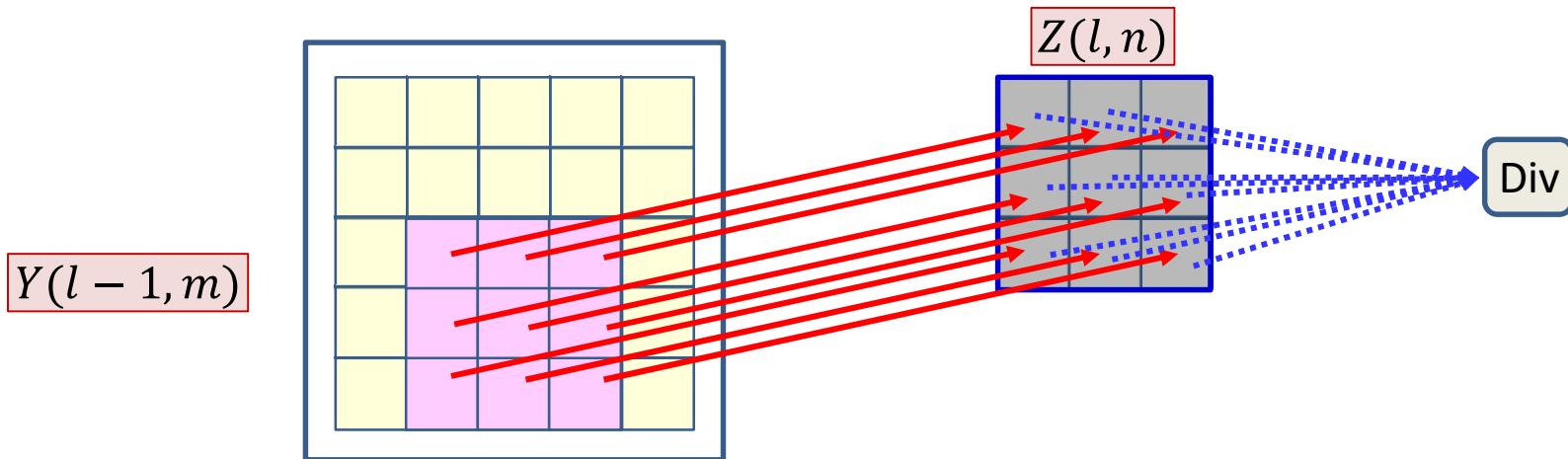
# Convolution: the contribution of a single weight



# Convolution: the contribution of a single weight



# The derivative for a single weight



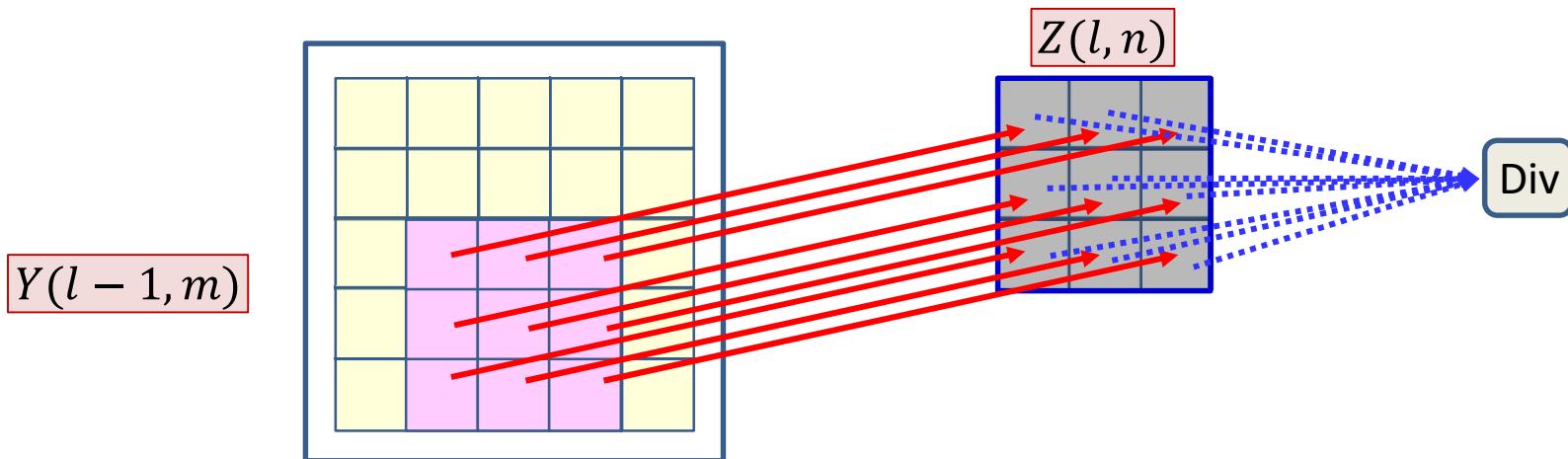
- Each filter component  $w_l(m, n, i, j)$  affects several  $z(l, n, x, y)$ 
  - The derivative of each  $z(l, n, x, y)$  w.r.t.  $w_l(m, n, i, j)$  is given by

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

- 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

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

# The derivative for a single weight



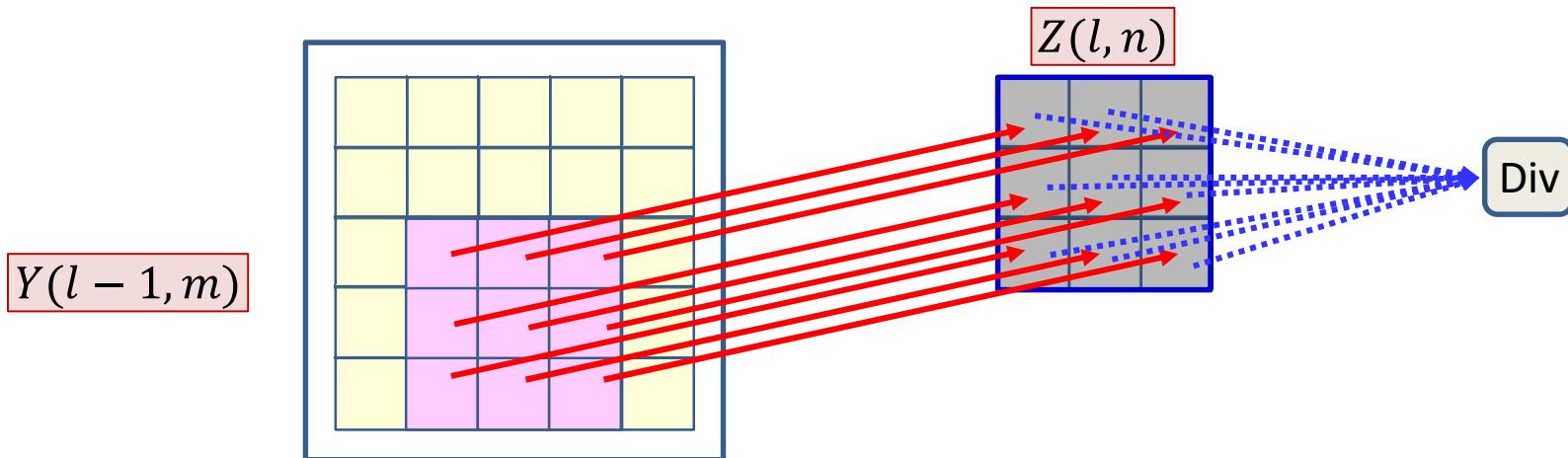
- Each filter component  $w_l(m, n, i, j)$  affects several  $z(l, n, x, y)$ 
  - The derivative of each  $z(l, n, x, y)$  w.r.t.  $w_l(m, n, i, j)$  is given by

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

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

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

# The derivative for a single weight



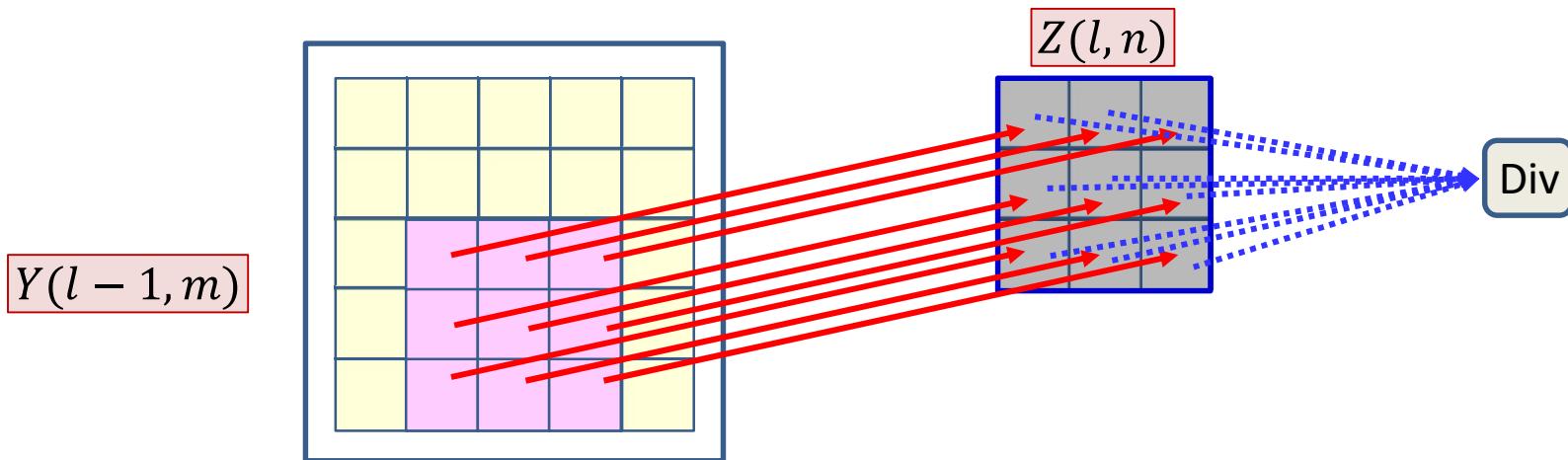
- Each filter component  $w_l(m, n, i, j)$  affects several  $z(l, n, x, y)$ 
  - The derivative of each  $z(l, n, x, y)$  w.r.t.  $w_l(m, n, i, j)$  is given by

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

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

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

# 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

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

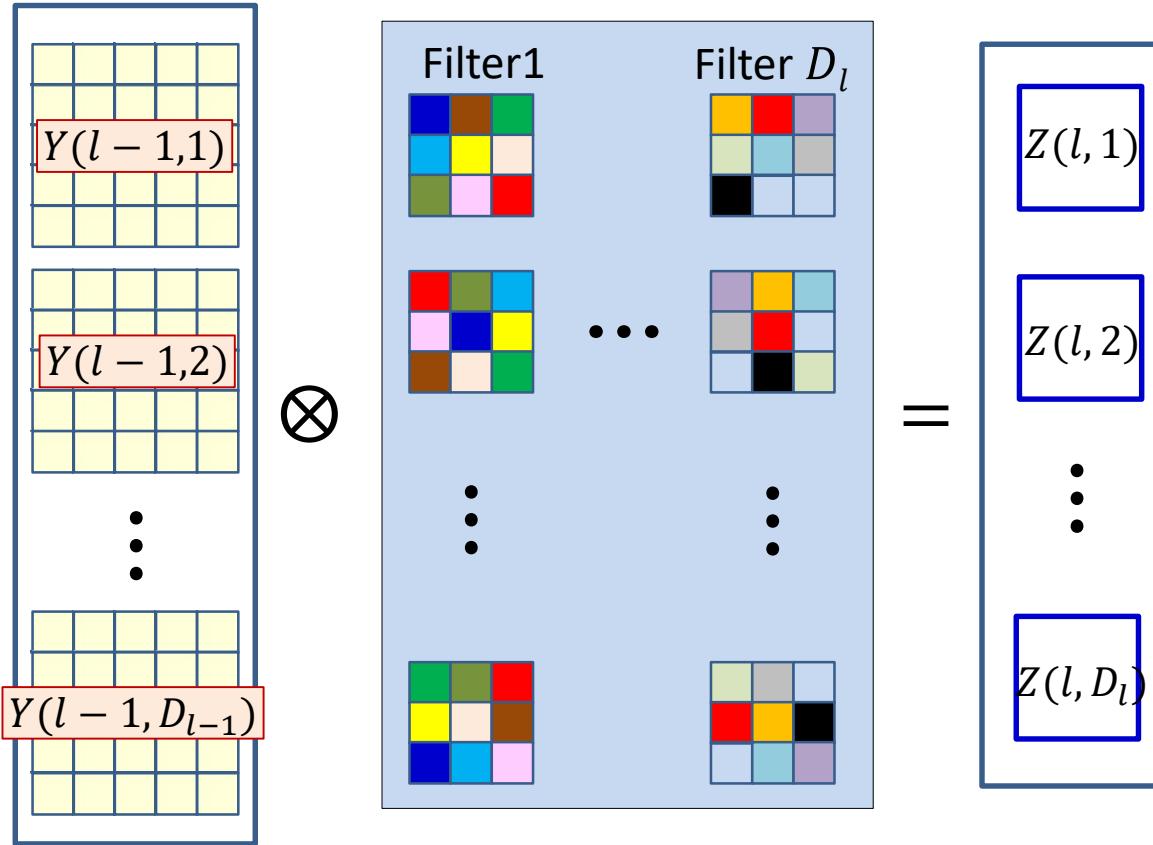
- The derivatives for all components of all filters can be computed directly from the above formula

- 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?

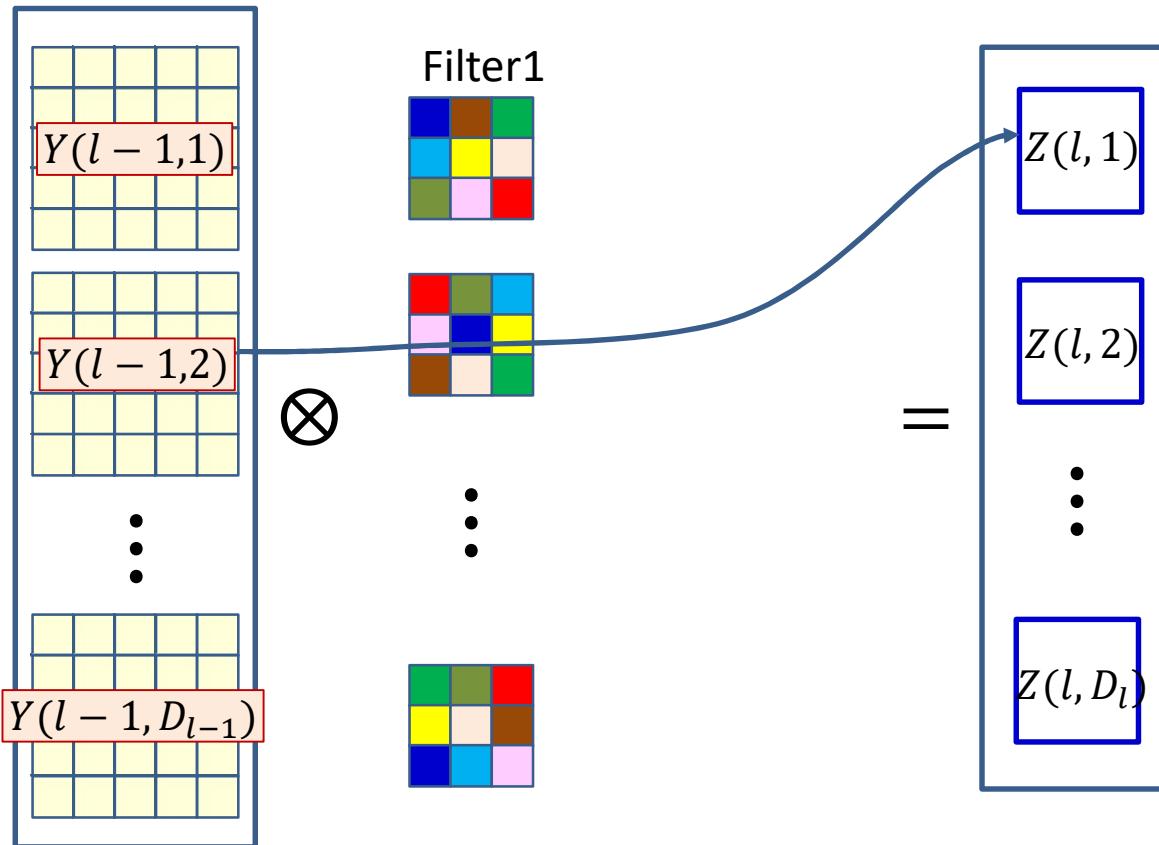
# Recap: Convolution



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

- Forward computation: Each filter produces an affine map

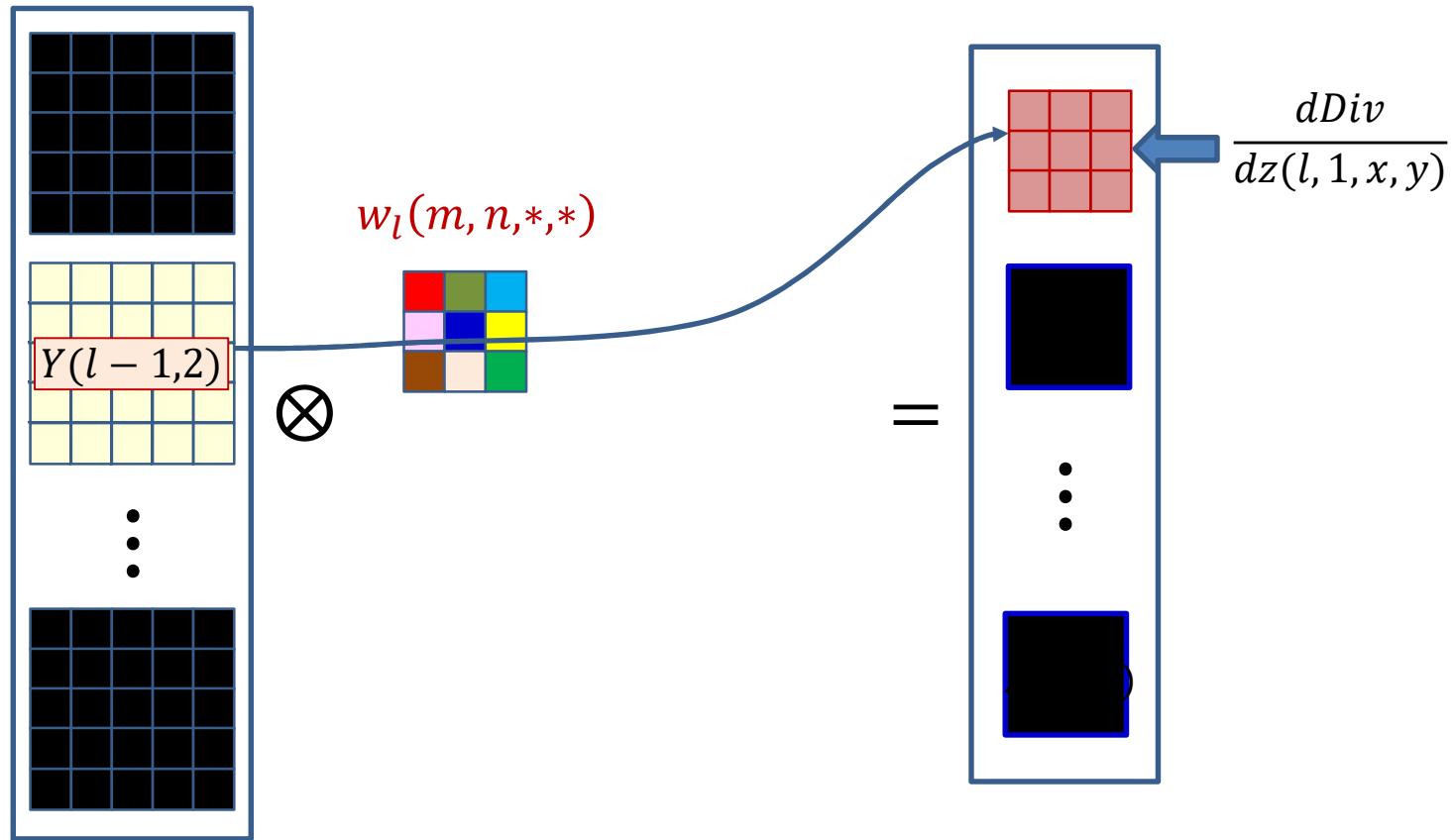
# Recap: Convolution



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

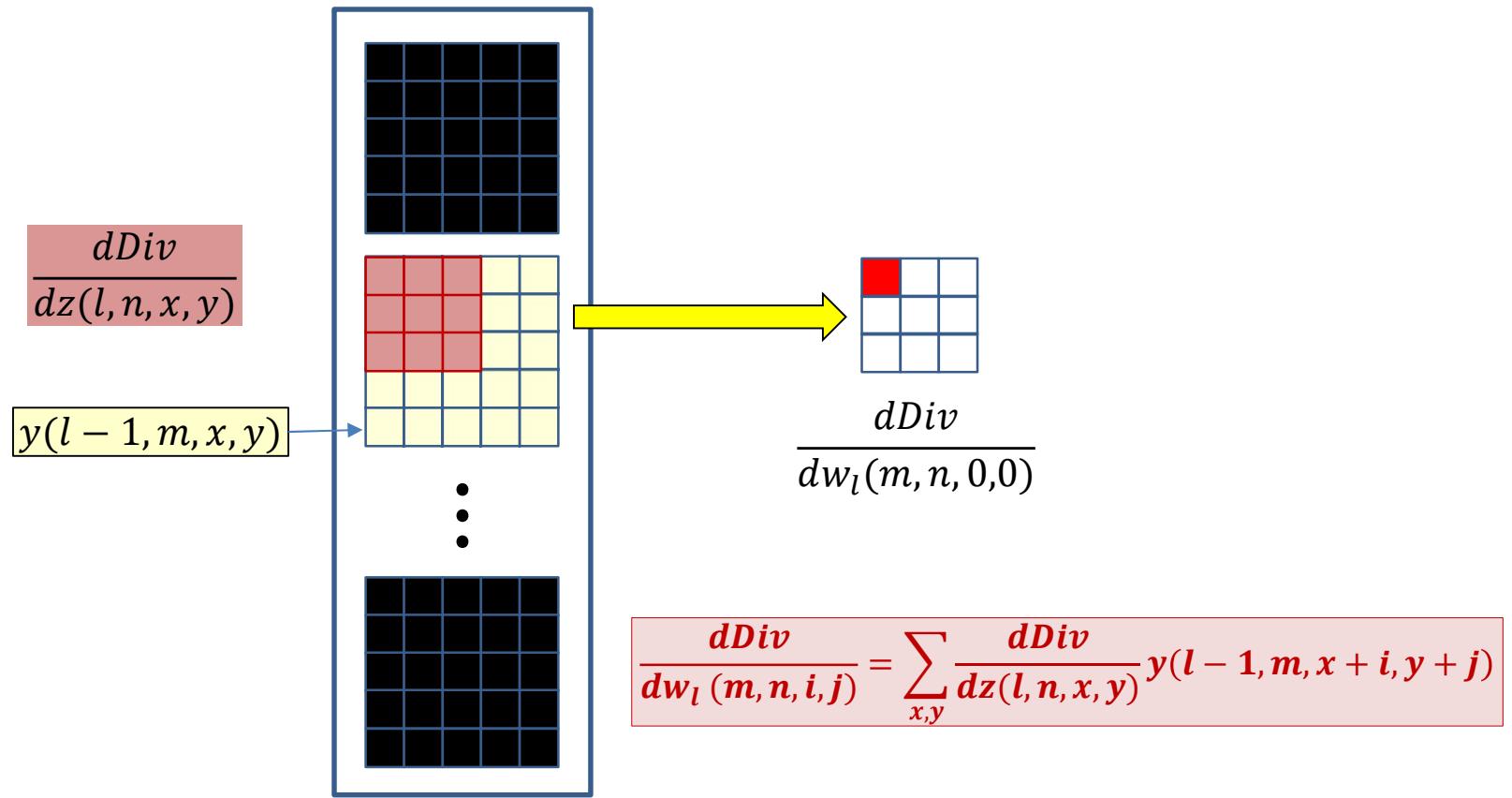
- $Y(l-1, m)$  influences  $Z(l, n)$  through  $w_l(m, n)$

# The filter derivative



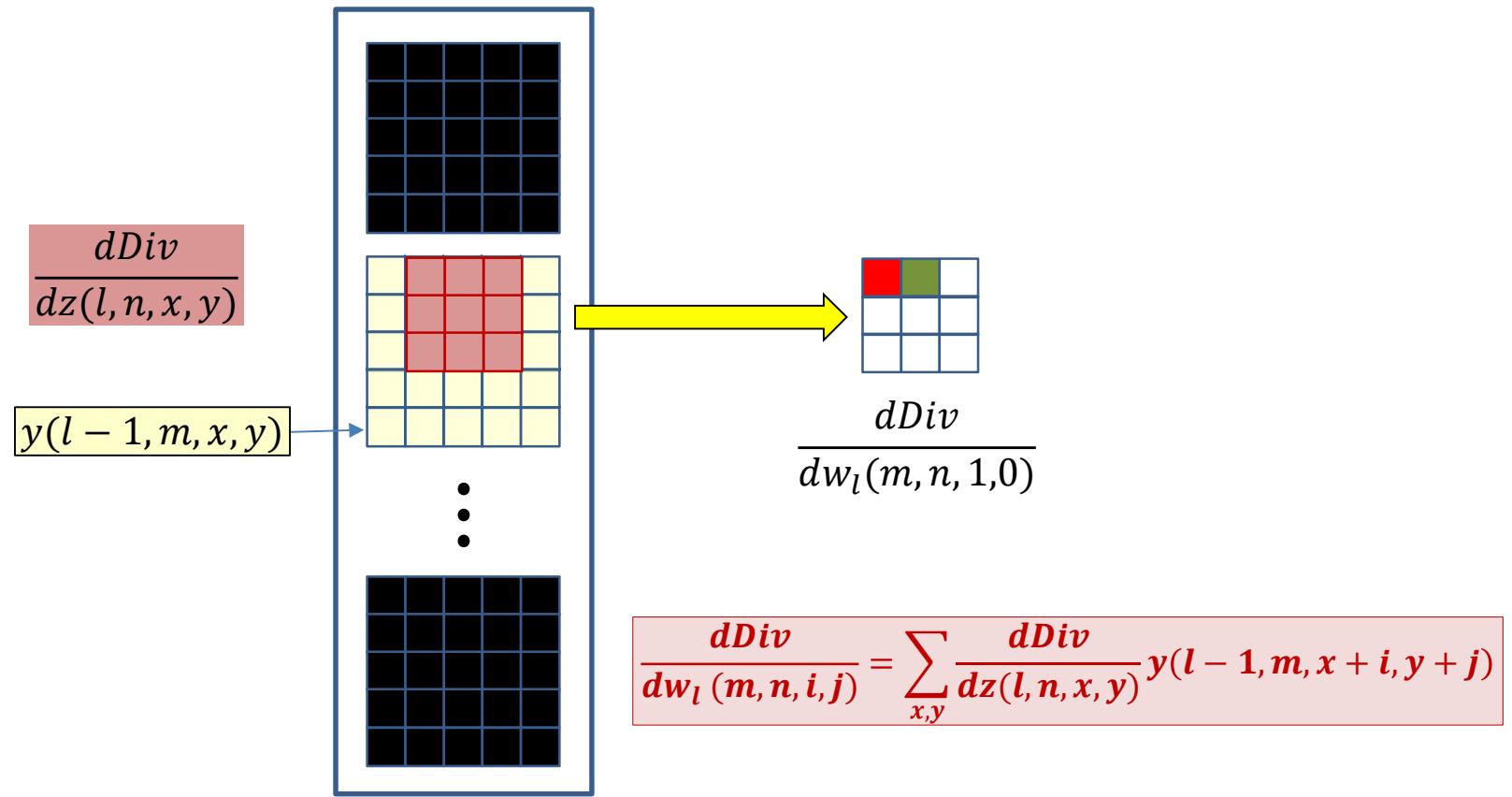
- The derivatives of the divergence w.r.t. every element of  $Z(l, n)$  is known
  - Must use them to compute the derivative for  $w_l(m, n, *, *)^{202}$

# The filter derivative



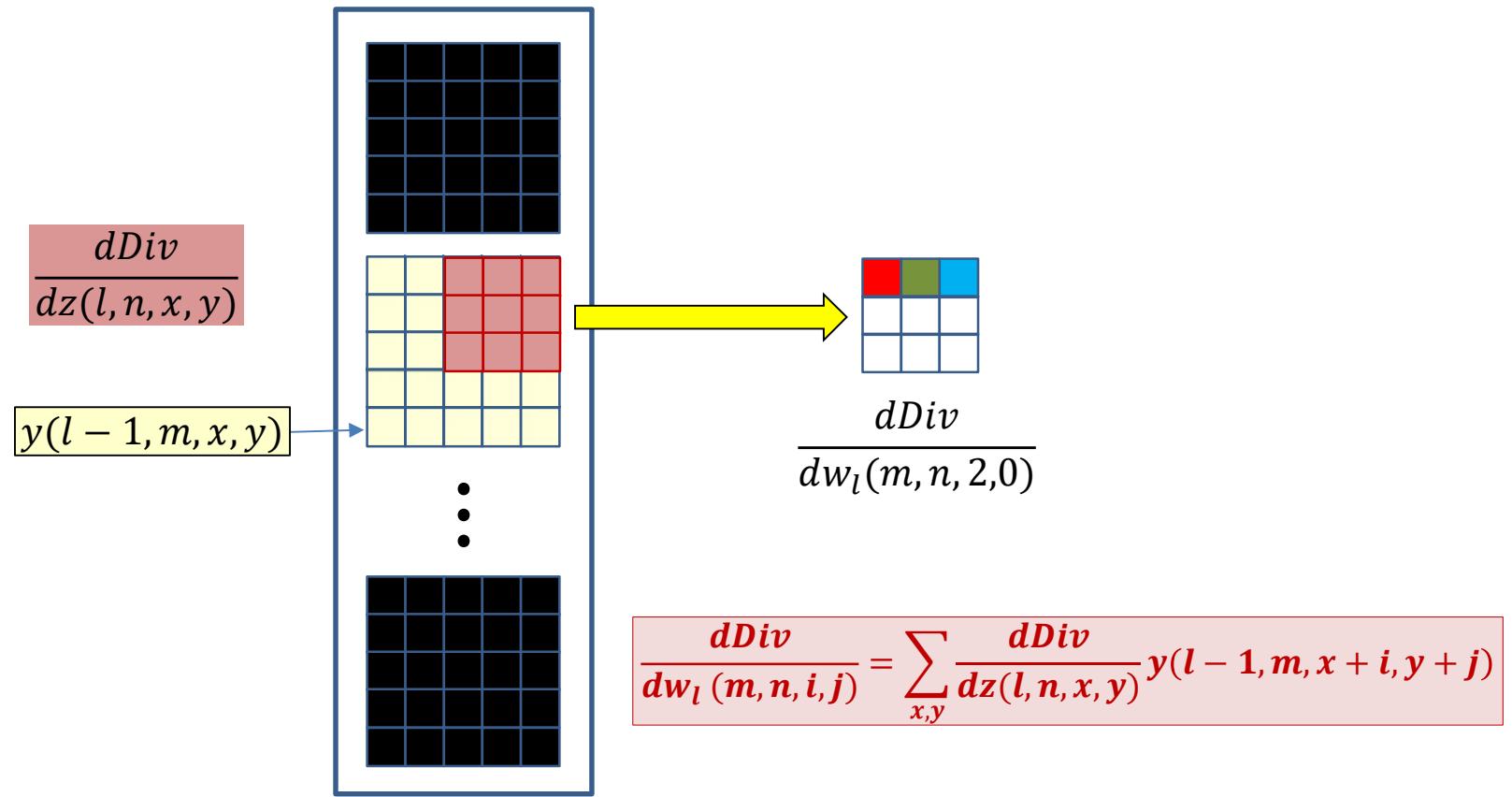
- The derivatives of the divergence w.r.t. every element of  $Z(l, n)$  is known
  - Must use them to compute the derivative for  $w_l(m, n, *, *)^{203}$

# The filter derivative



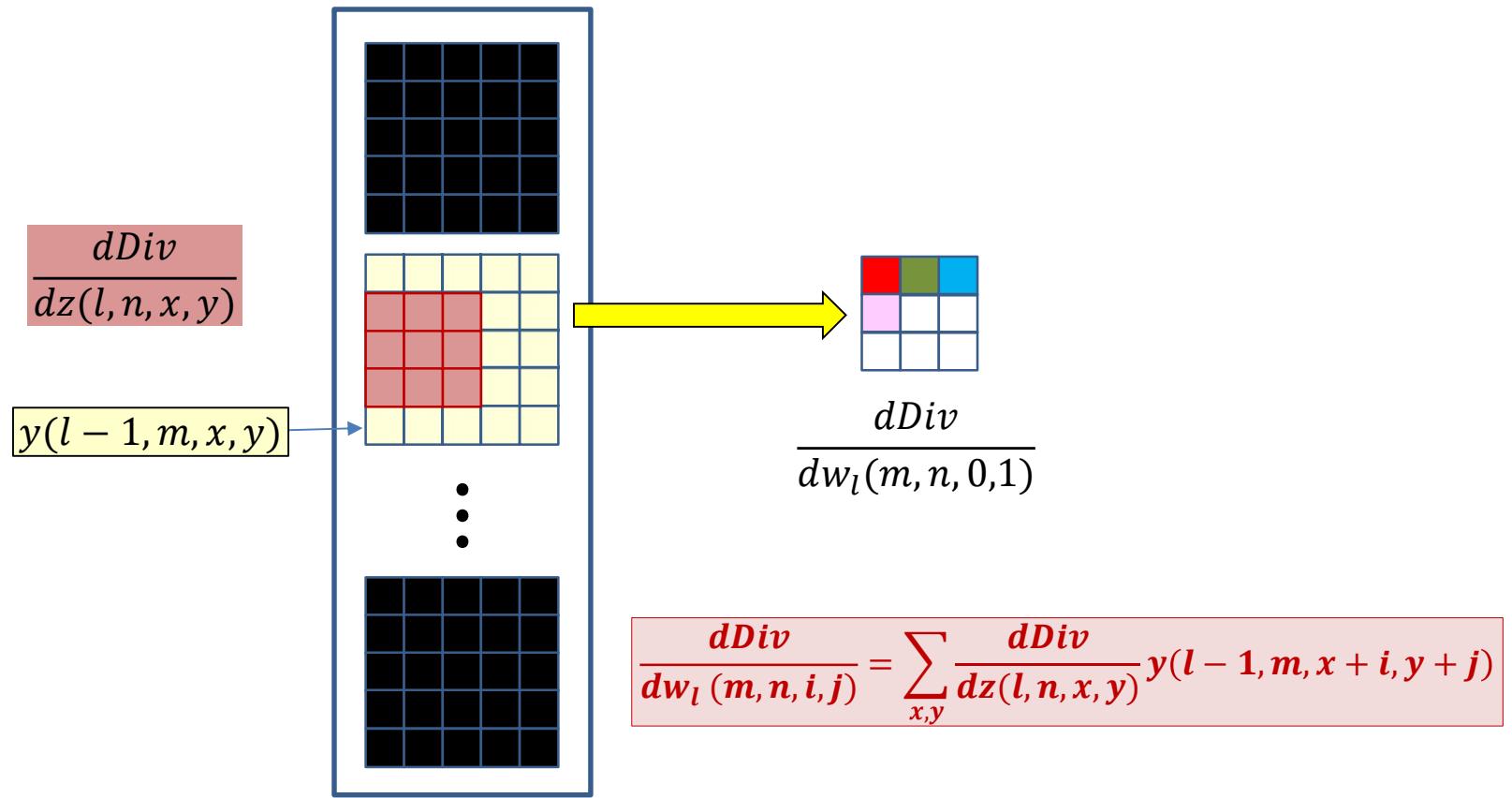
- The derivatives of the divergence w.r.t. every element of  $Z(l, n)$  is known
  - Must use them to compute the derivative for  $w_l(m, n, *, *)^{204}$

# The filter derivative



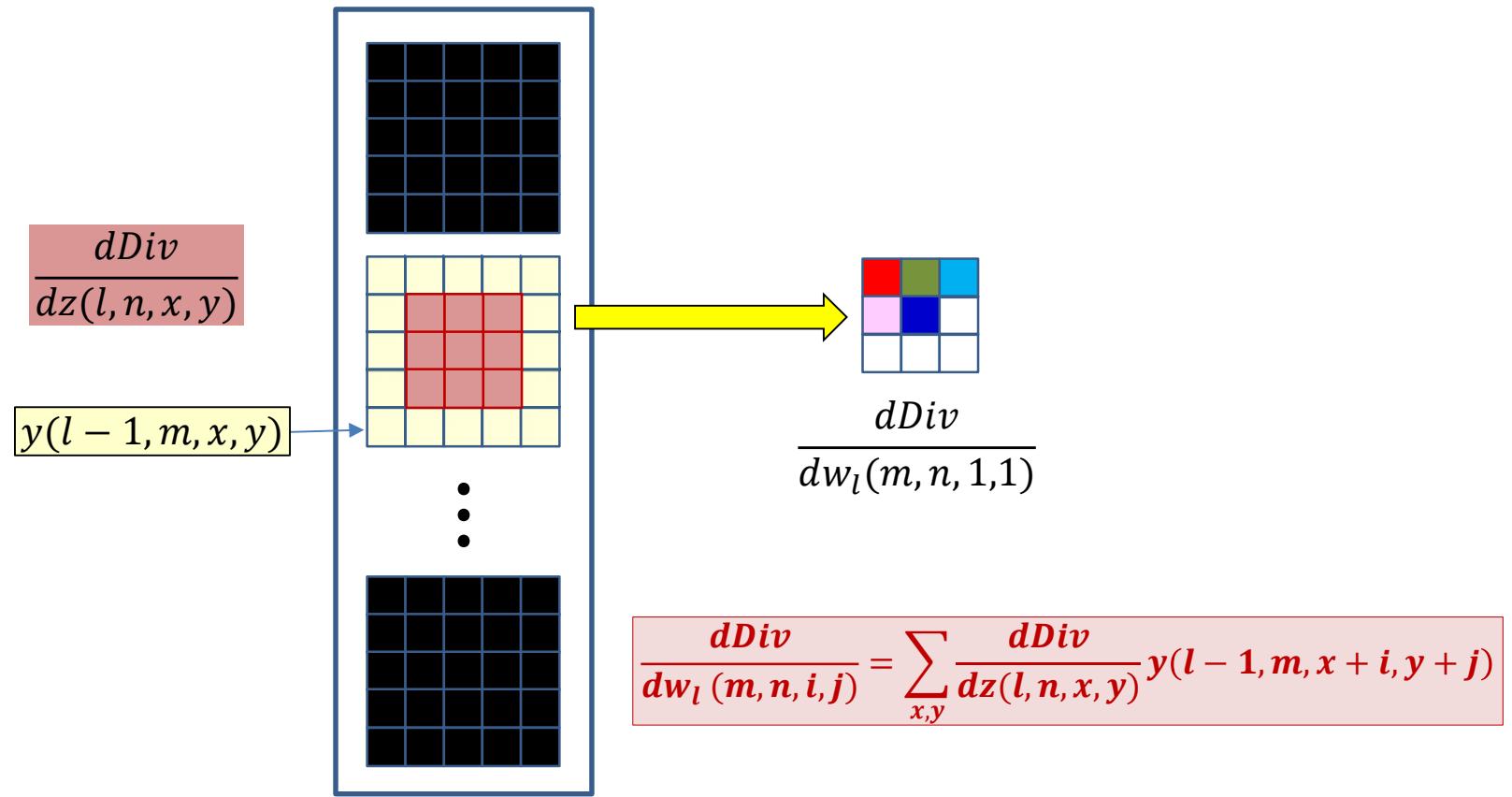
- The derivatives of the divergence w.r.t. every element of  $Z(l, n)$  is known
  - Must use them to compute the derivative for  $w_l(m, n, *, *)^{205}$

# The filter derivative



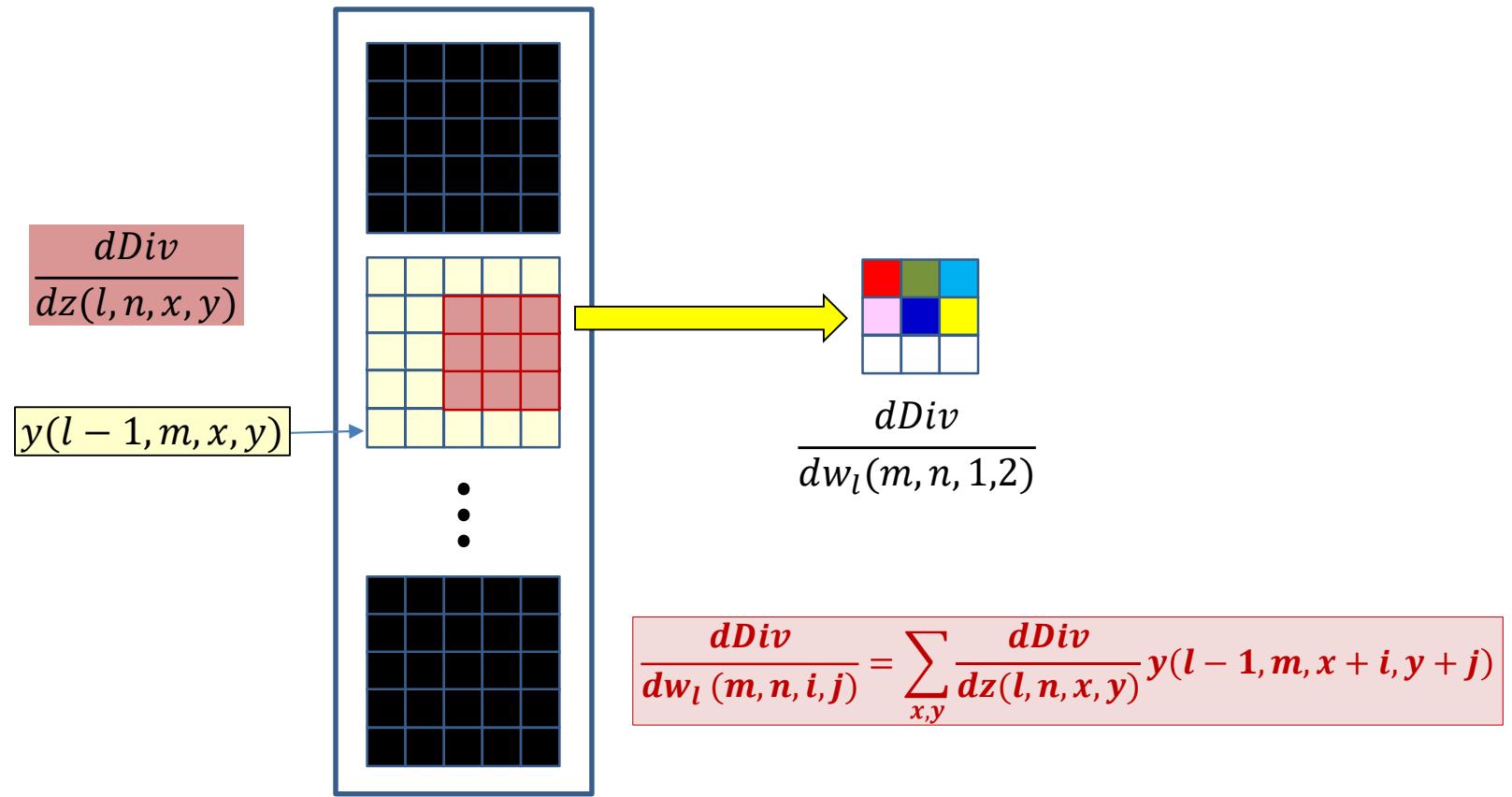
- The derivatives of the divergence w.r.t. every element of  $Z(l, n)$  is known
  - Must use them to compute the derivative for  $w_l(m, n, *, *)^{206}$

# The filter derivative



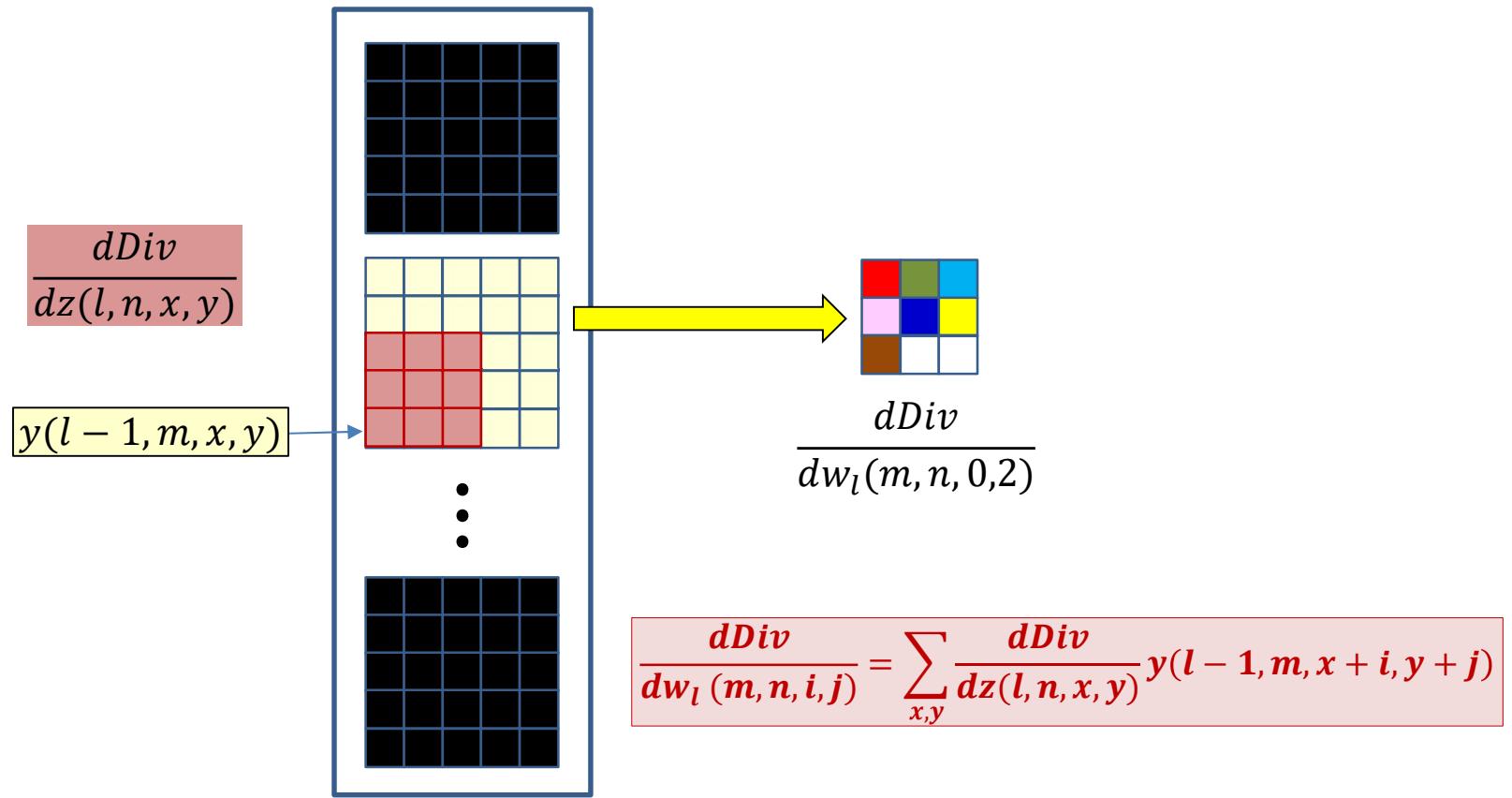
- The derivatives of the divergence w.r.t. every element of  $Z(l, n)$  is known
  - Must use them to compute the derivative for  $w_l(m, n, *, *)^{207}$

# The filter derivative



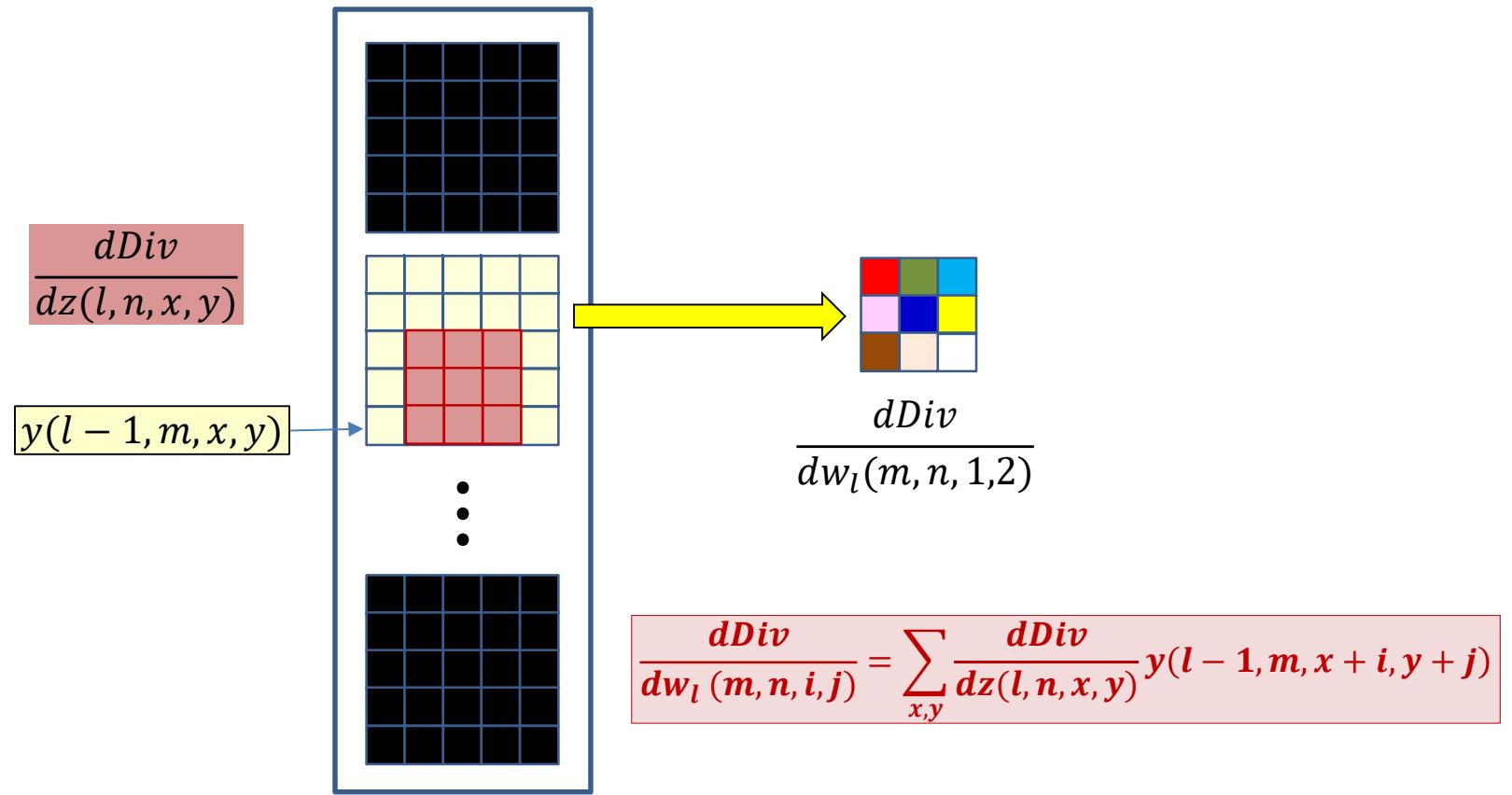
- The derivatives of the divergence w.r.t. every element of  $Z(l, n)$  is known
  - Must use them to compute the derivative for  $w_l(m, n, *, *)^{208}$

# The filter derivative



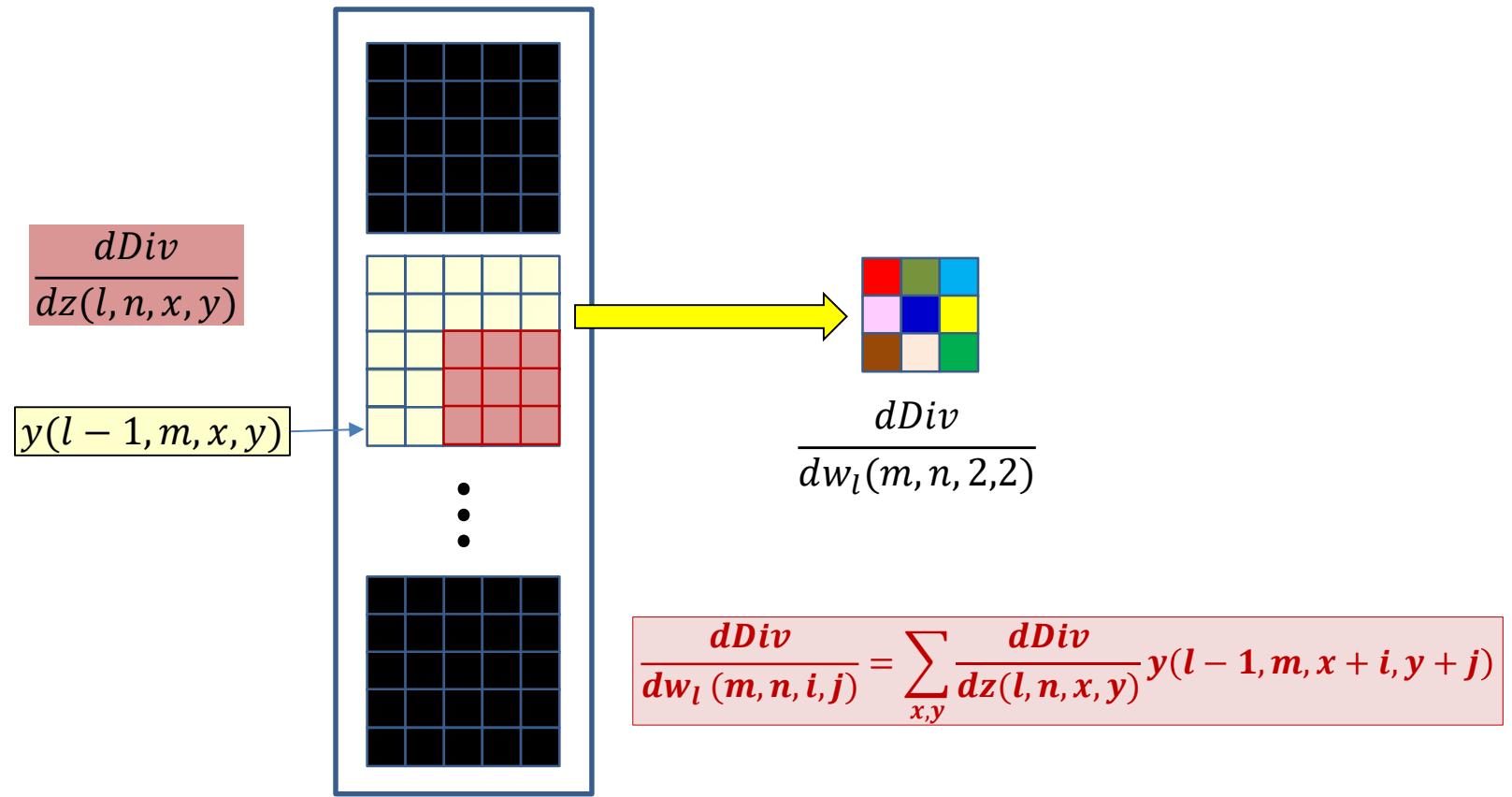
- The derivatives of the divergence w.r.t. every element of  $Z(l, n)$  is known
  - Must use them to compute the derivative for  $w_l(m, n, *, *)^{209}$

# The filter derivative



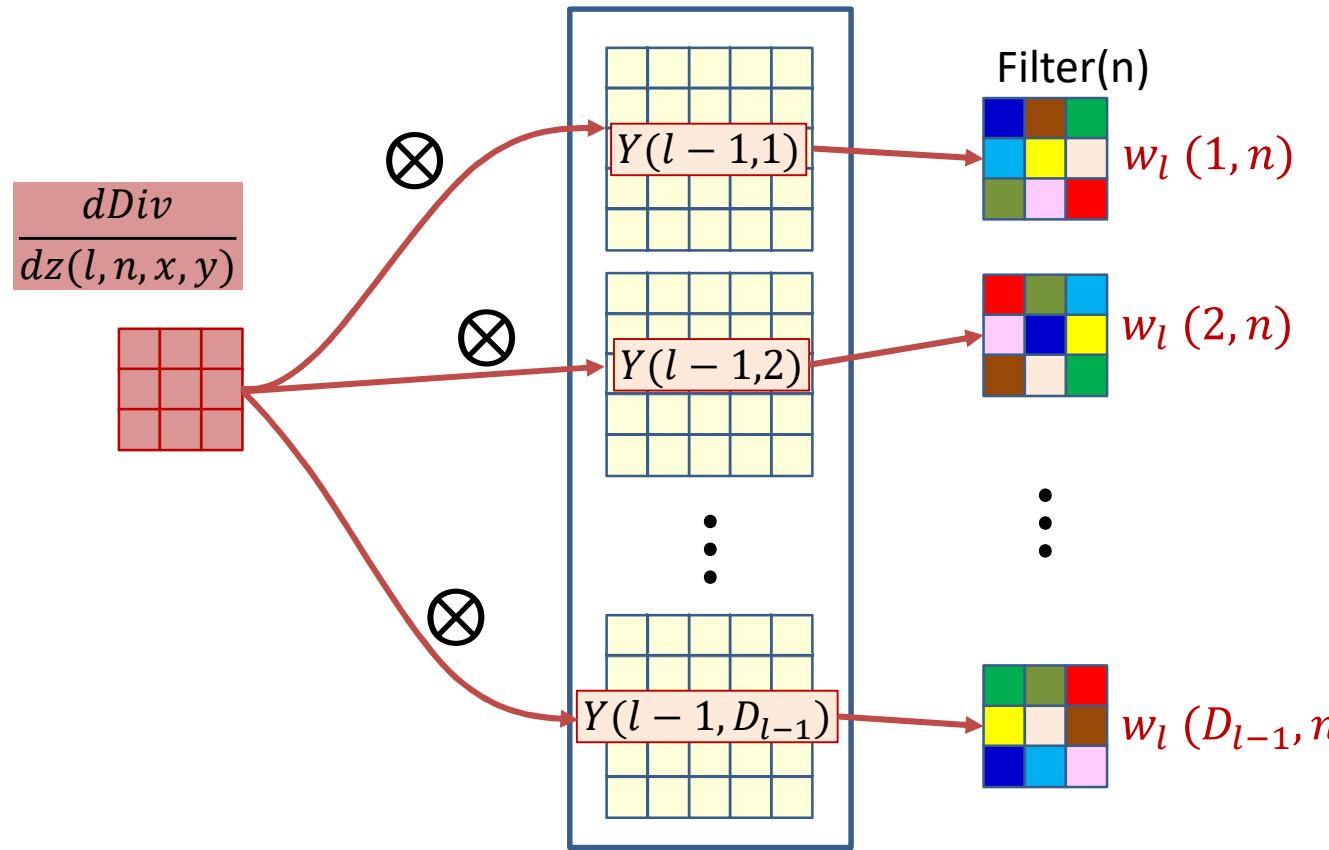
- The derivatives of the divergence w.r.t. every element of  $Z(l, n)$  is known
  - Must use them to compute the derivative for  $w_l(m, n, *, *)^{110}$

# The filter derivative



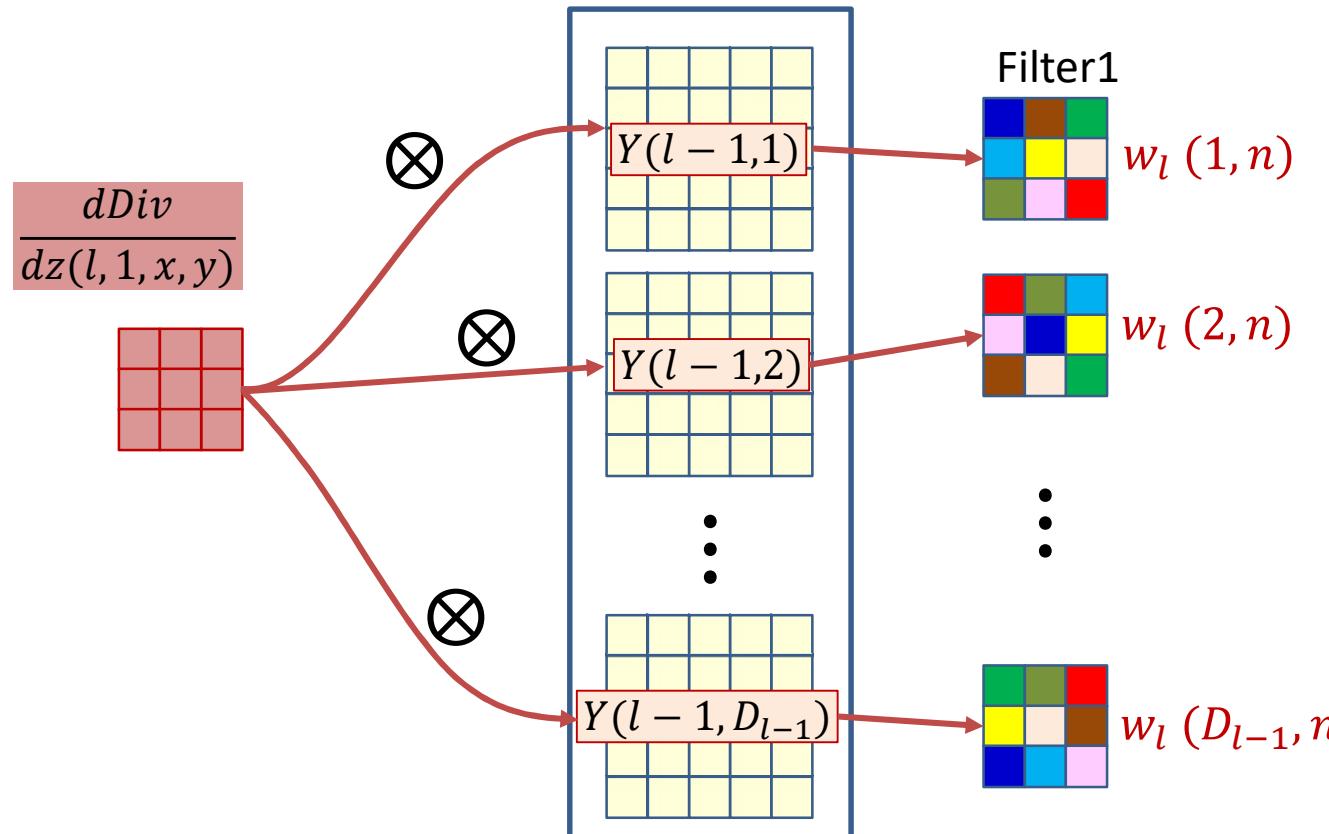
- The derivatives of the divergence w.r.t. every element of  $Z(l, n)$  is known
  - Must use them to compute the derivative for  $w_l(m, n, *, *)^{111}$

# The filter derivative



- The derivative of the  $n^{\text{th}}$  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  $m^{\text{th}}$  “plane” of the  $n^{\text{th}}$  filter

# The filter derivative



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

$\frac{d\text{Div}}{dw_l(m, n, i, j)}$  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

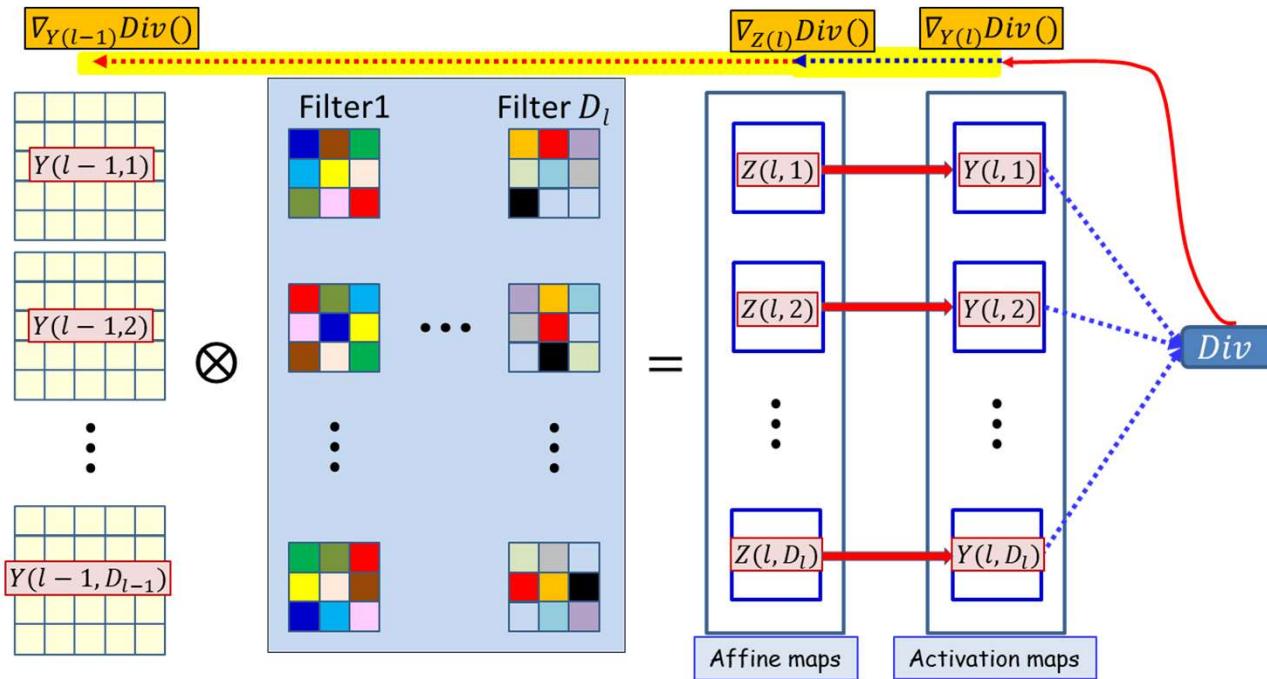
# Derivatives for the filters at layer $l$ :

## Vector notation

```
# The weight  $W(l, j)$  is a 3D  $D_{l-1} \times K_l \times K_l$ 
# 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:Dl
  for x = 1:Kl
    for y = 1:Kl
      for m = 1:Dl-1
        dw(l,m,n,x,y) = dz(l,n,:,:,:) . #dot product
                                         y(l-1,m,x:x+Kl-1,y:y+Kl-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:Dl
                z(l,j,x,y) = 0
                for i = 1:Dl-1
                    for x' = 1:Kl
                        for y' = 1:Kl
                            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))
```

Switching to 1-based  
indexing with appropriate  
adjustments

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

# Backward layer $l$

```
dw(l) = zeros(DlxDl-1xKlxKl)
dY(l-1) = zeros(Dl-1xWl-1xHl-1)
for x = 1:Wl-1-Kl+1
    for y = 1:Hl-1-Kl+1
        for j = 1:Dl
            dz(l,j,x,y) = dY(l,j,x,y).f'(z(l,j,x,y))
            for i = 1:Dl-1
                for x' = 1:Kl
                    for y' = 1:Kl
                        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(DlxDl-1xKlxKl)
    dY(l-1) = zeros(Dl-1xWl-1xHl-1)
    for x = 1:Wl-1-Kl+1
        for y = 1:Hl-1-Kl+1
            for j = 1:Dl
                dz(l,j,x,y) = dY(l,j,x,y).f'(z(l,j,x,y))
                for i = 1:Dl-1
                    for x' = 1:Kl
                        for y' = 1:Kl
                            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(DlxDl-1xKlxKl)
```

```
dY(l-1) = zeros(Dl-1 x Wl-1 x Hl-1)
```

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

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

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

```
            dz(l,j,x,y) = dY(l,j,x,y) . f'(z(l,j,x,y))
```

```
            for i = 1:Dl-1
```

```
                for x' = 1:Kl
```

```
                    for y' = 1:Kl
```

```
                        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

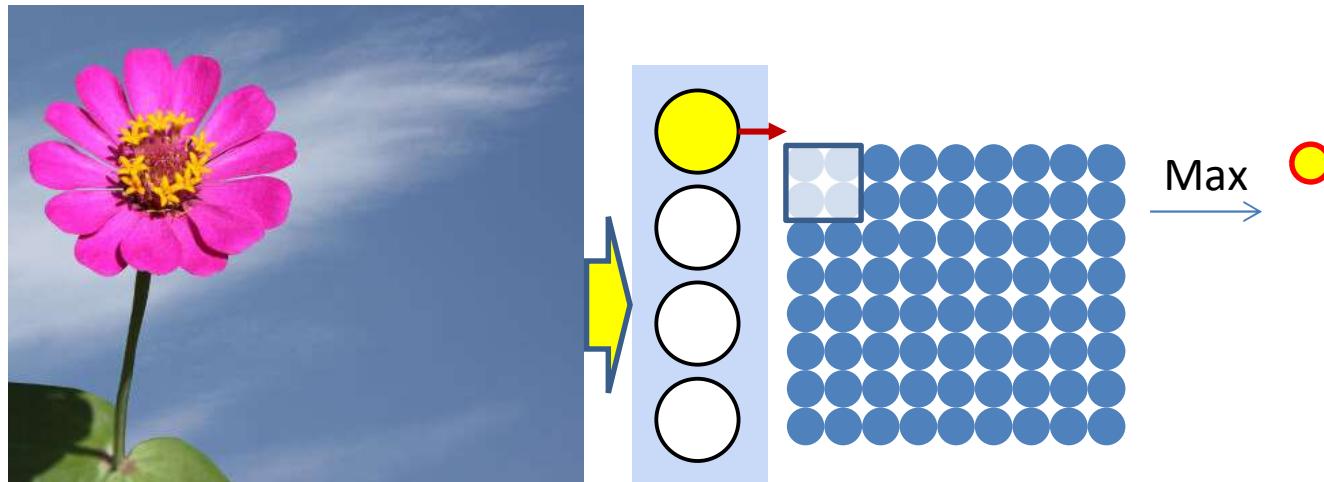
# Complete Backward (with strides)

```
dY(L) = dDiv/dY(L)
for l = L:1 # Backward through layers
    dw(l) = zeros(D_lxD_{l-1}xK_lxK_l)
    dY(l-1) = zeros(D_{l-1}xW_{l-1}xH_{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

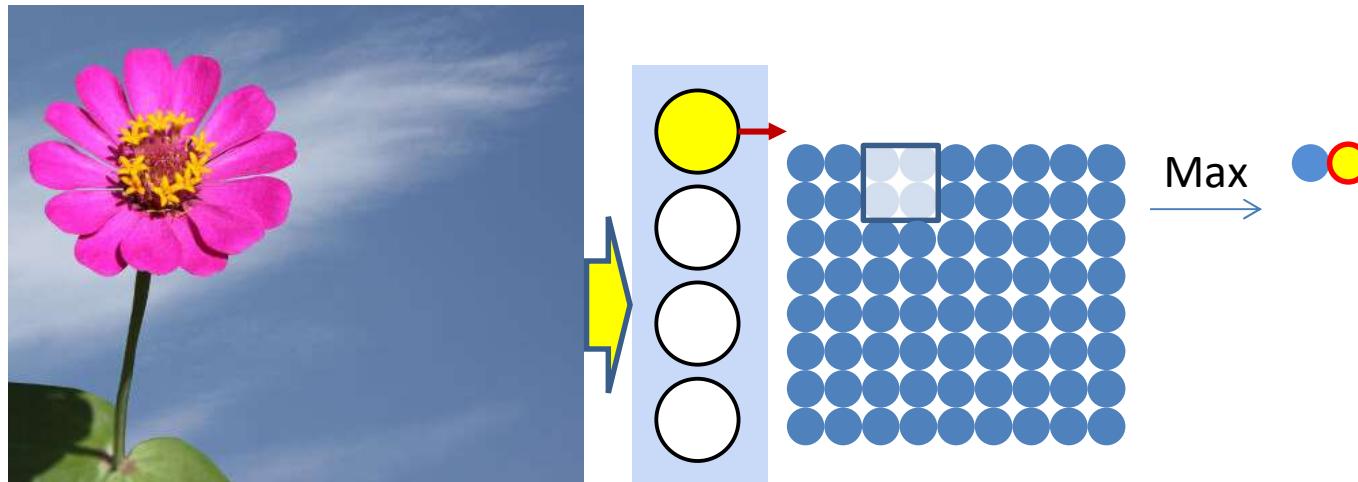
- **Assumption:** We already have the derivatives w.r.t. the elements of the maps output by the final convolutional (or pooling) layer
  - Obtained as a result of backpropagating through the flat MLP
- **Required:**
  - **For convolutional layers:**
    - How to compute the derivatives w.r.t. the affine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



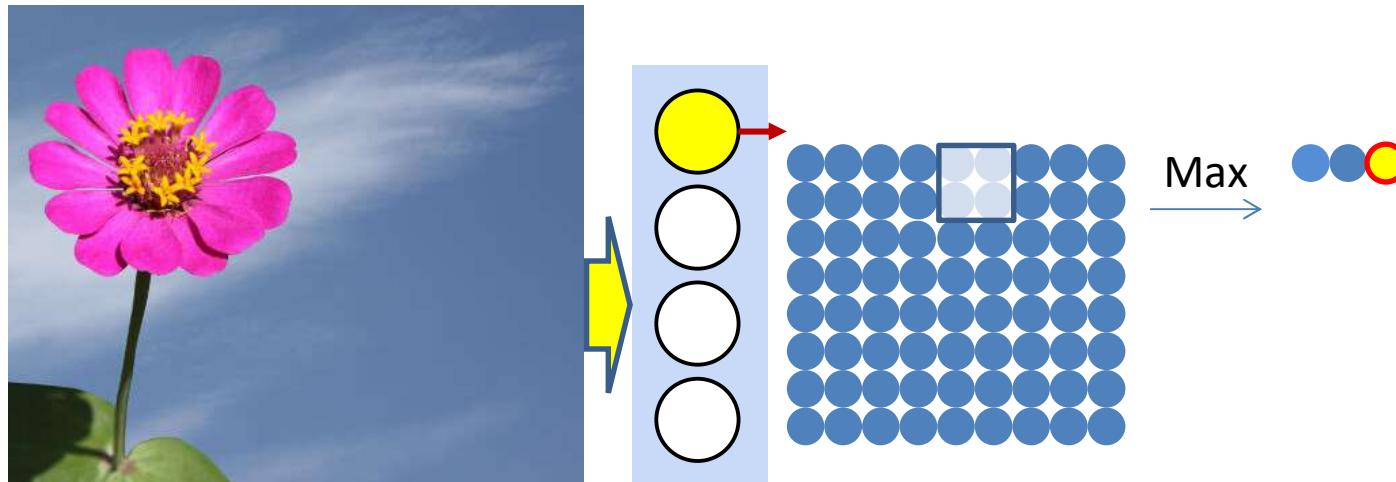
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

# Pooling and downsampling



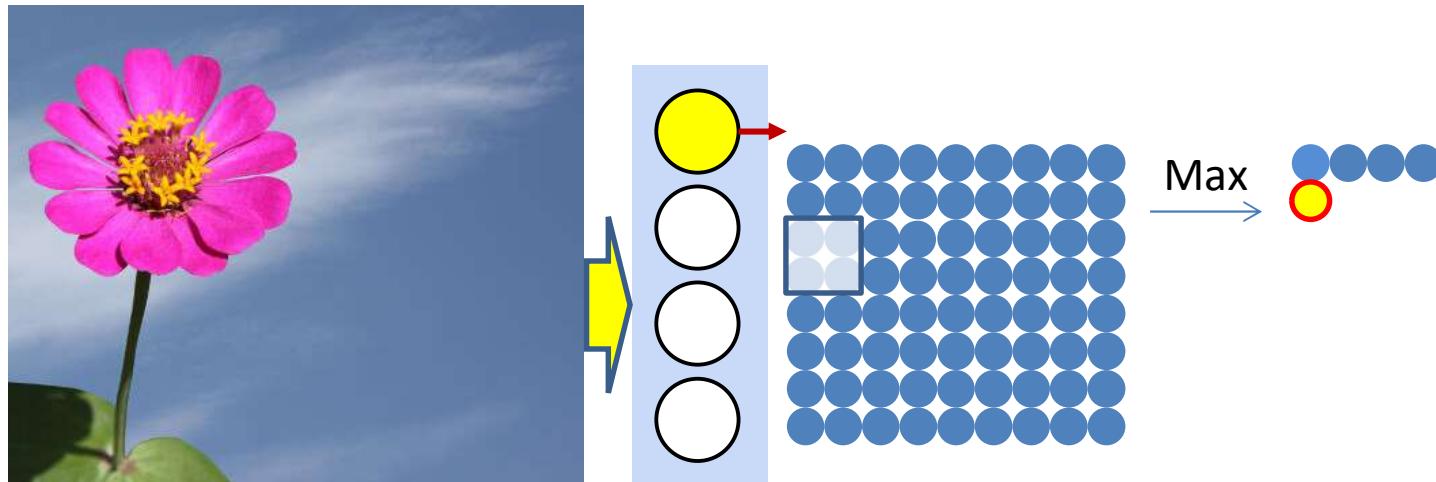
- Pooling is typically performed with strides  $> 1$ 
  - Results in shrinking of the map
  - “Downsampling”

# Pooling and downsampling



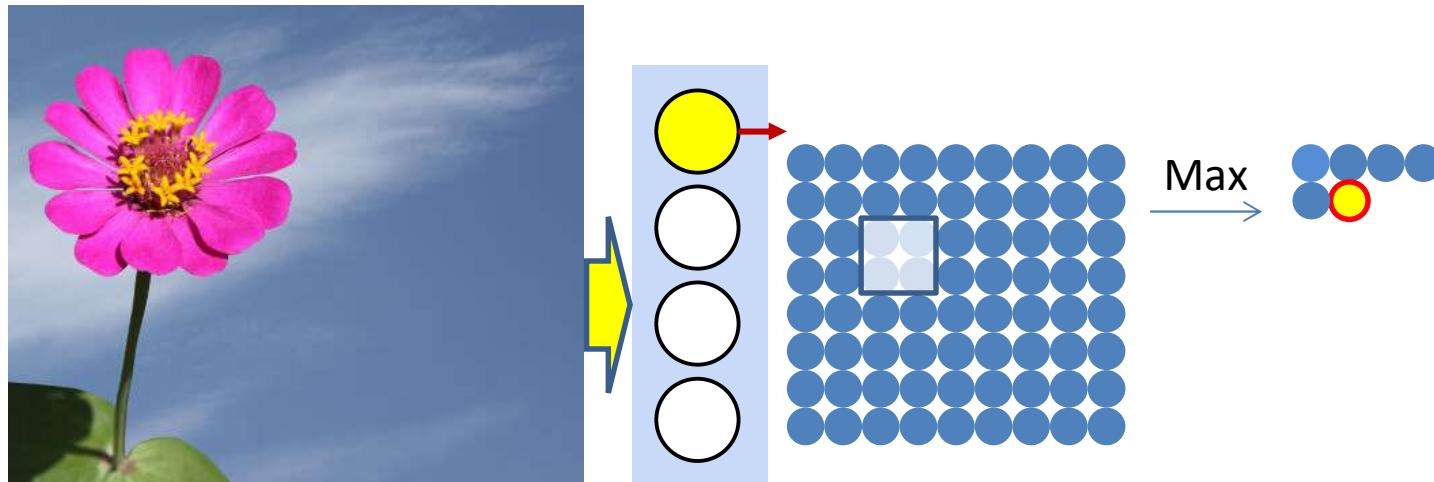
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

# Pooling and downsampling



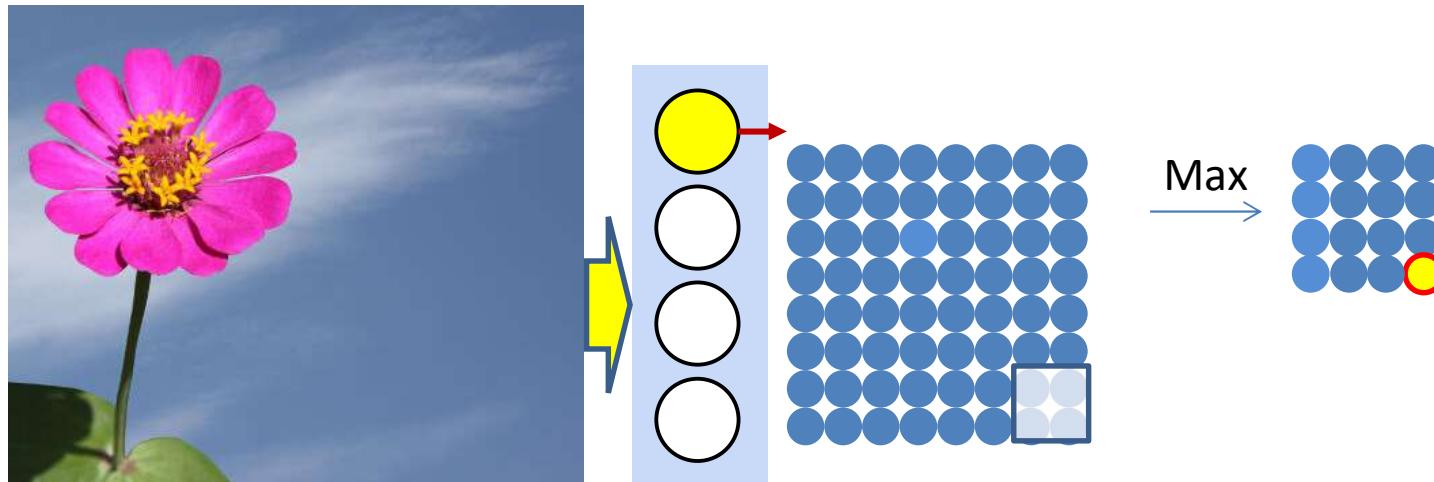
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

# Pooling and downsampling



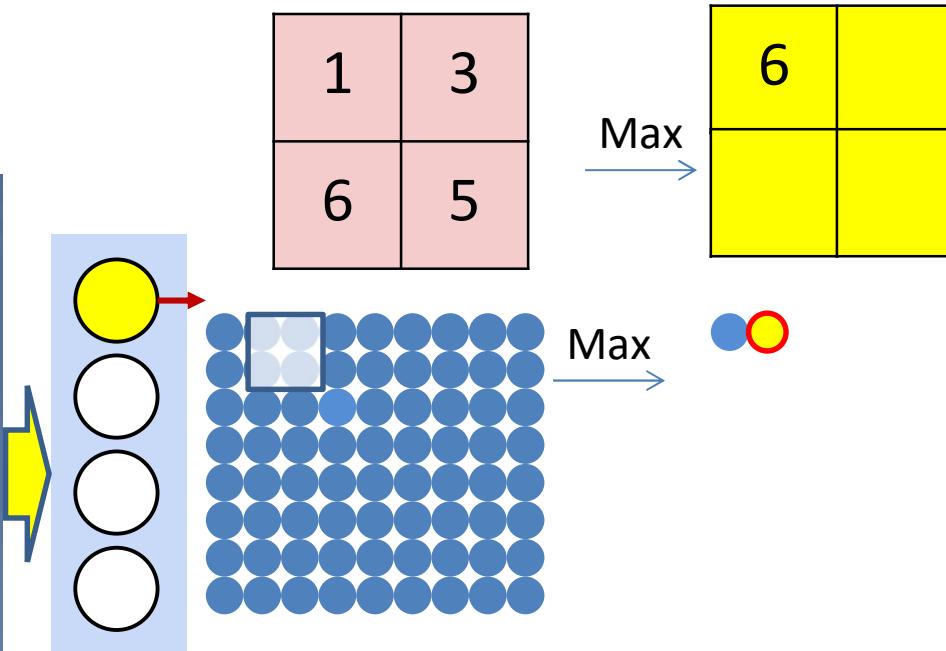
- Pooling is typically performed with strides  $> 1$ 
  - Results in shrinking of the map
  - “Downsampling”

# Pooling and downsampling



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

# Max pooling

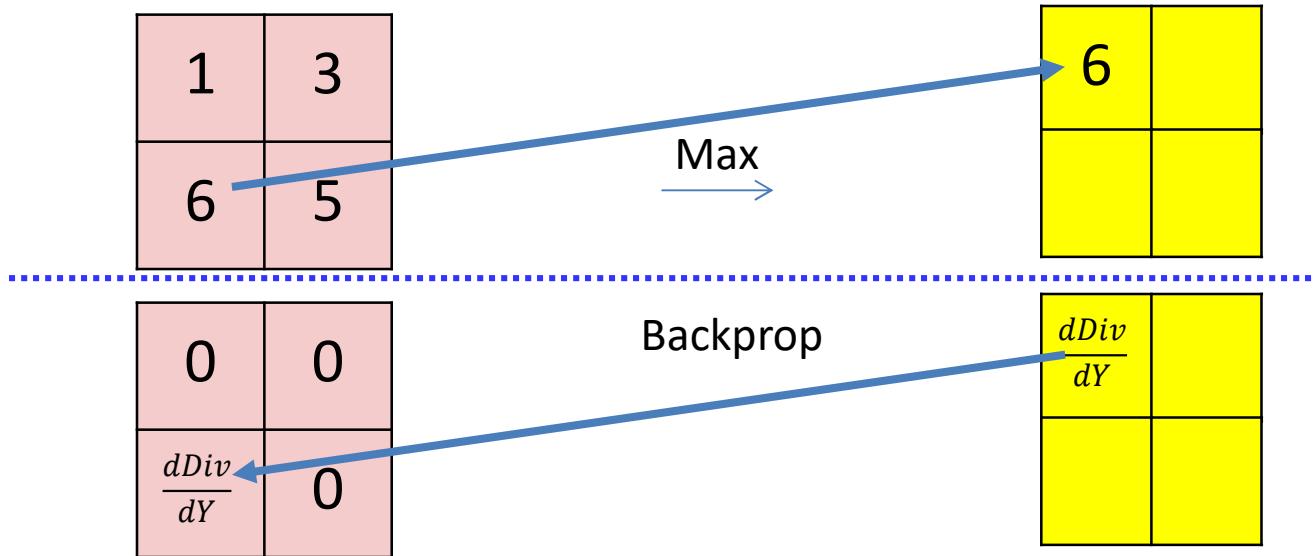


- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

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

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

# Derivative of Max pooling



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

- Max pooling selects the largest from a pool of elements

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

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

# 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:D1
    m = 1
    for x = 1:stride(l):Wl-1-Kl+1
        n = 1
        for y = 1:stride(l):Hl-1-Kl+1
            pidx(l,j,m,n) = maxidx(y(l-1,j,x:x+Kl-1,y:y+Kl-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

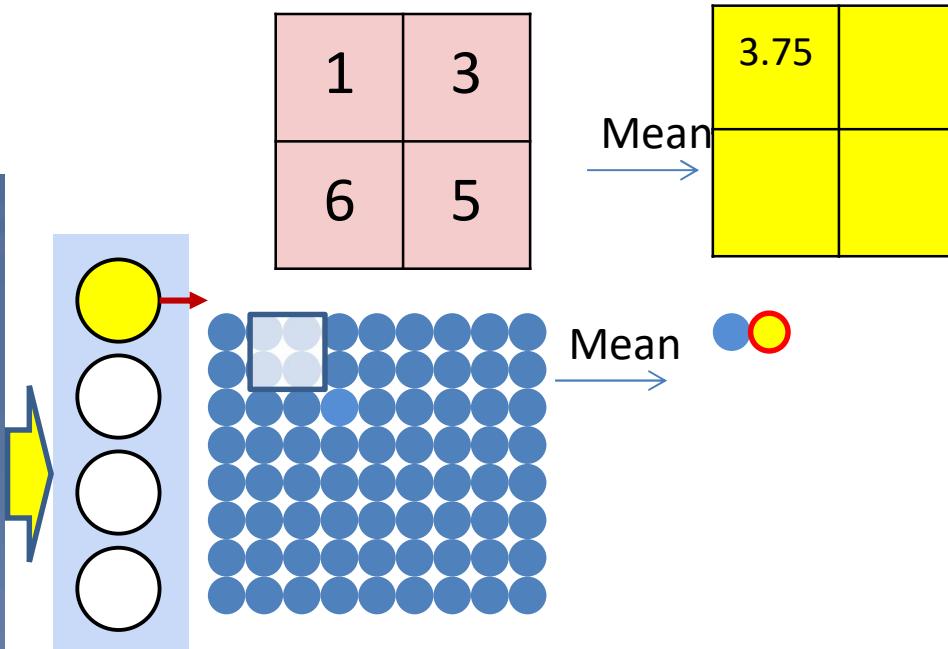


## Max pooling

```
dy (:,:, :) = zeros (D1 x W1 x H1)
for j = 1:D1
    for x = 1:W1_downsampled
        for y = 1:H1_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

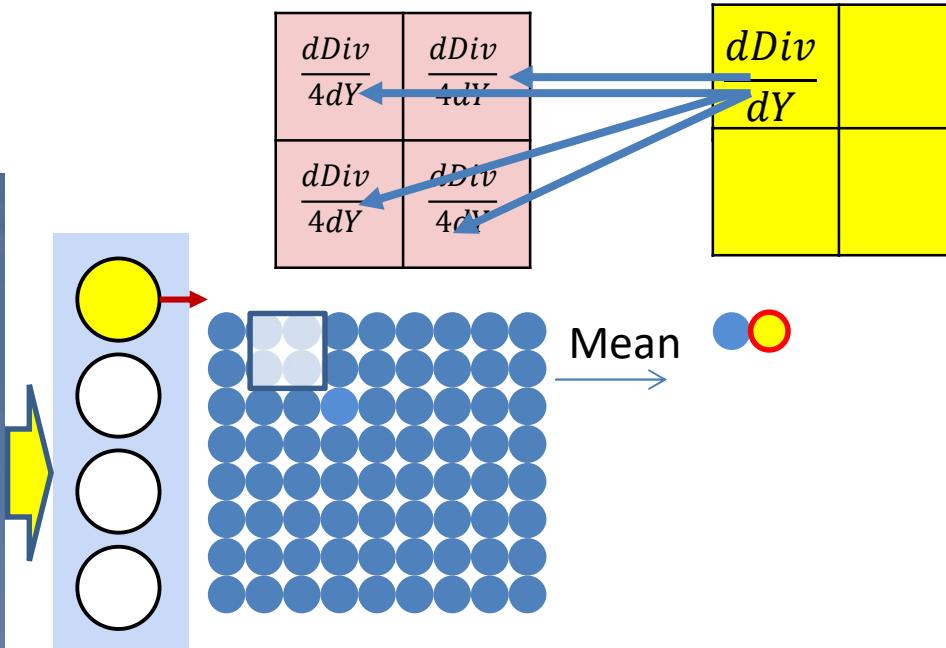
# Mean pooling



- 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

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

# 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:D1 #Over the maps
    m = 1
    for x = 1:stride(l):Wl-1-K1+1 #K1 = pooling kernel size
        n = 1
        for y = 1:stride(l):Hl-1-K1+1
            y(l,j,m,n) = mean(y(l-1,j,x:x+K1-1,y:y+K1-1))
            n = n+1
        m = m+1
```

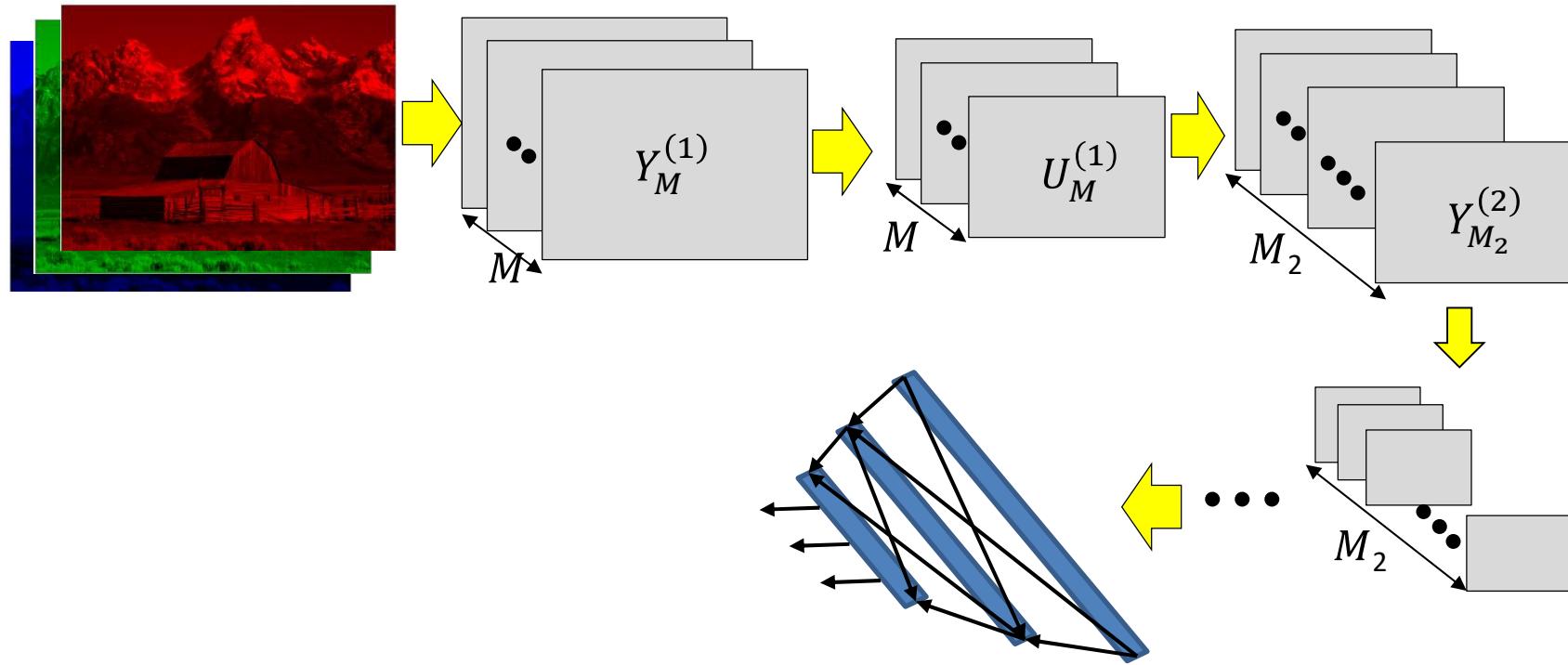
# Derivative of mean pooling layer at layer $l$

## Mean pooling

```
dy (:,:, :, :) = zeros (D1 x W1 x H1)
for j = 1:D1
    for x = 1:W1_downsampled
        n = (x-1)*stride
        for y = 1:H1_downsampled
            m = (y-1)*stride
            for i = 1:Klpool
                for j = 1:Klpool
                    dy (l-1, j, p, n+i, m+j) += (1/K2lpool) y (l, j, x, y)
```

“+=” because adjacent windows may overlap

# Learning the network

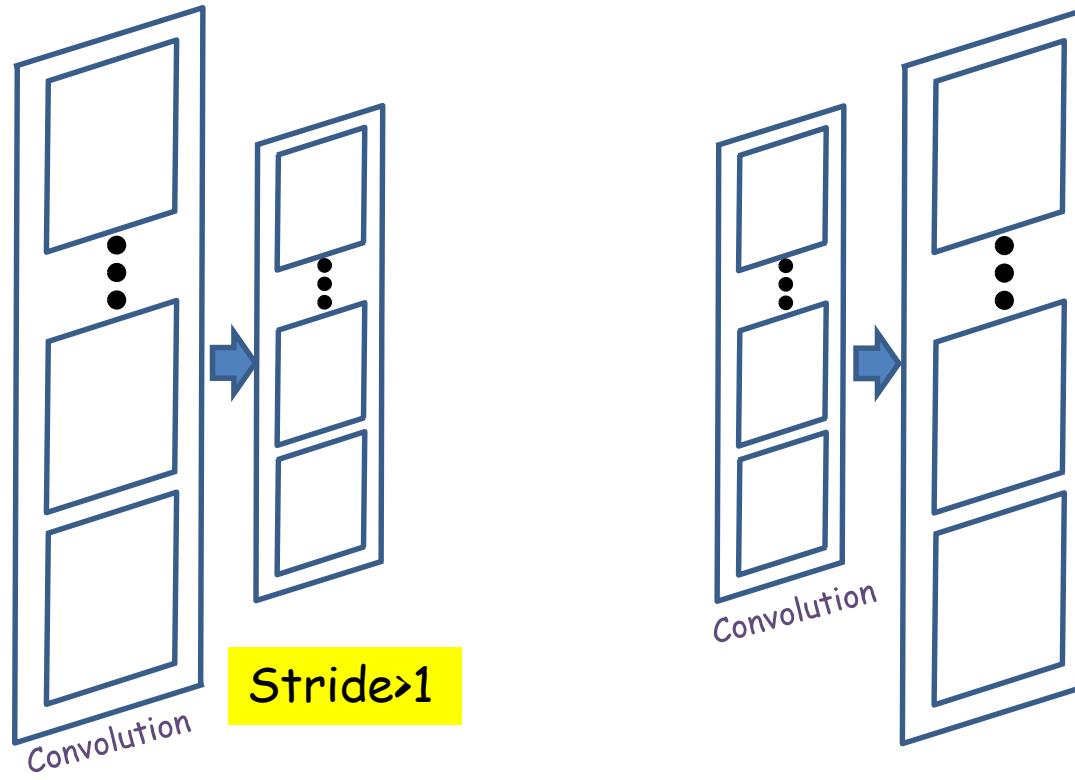


- Have shown the derivative of divergence w.r.t every intermediate output, and every free parameter (filter weights)
- Can now be embedded in gradient descent framework to learn the network

# Story so far

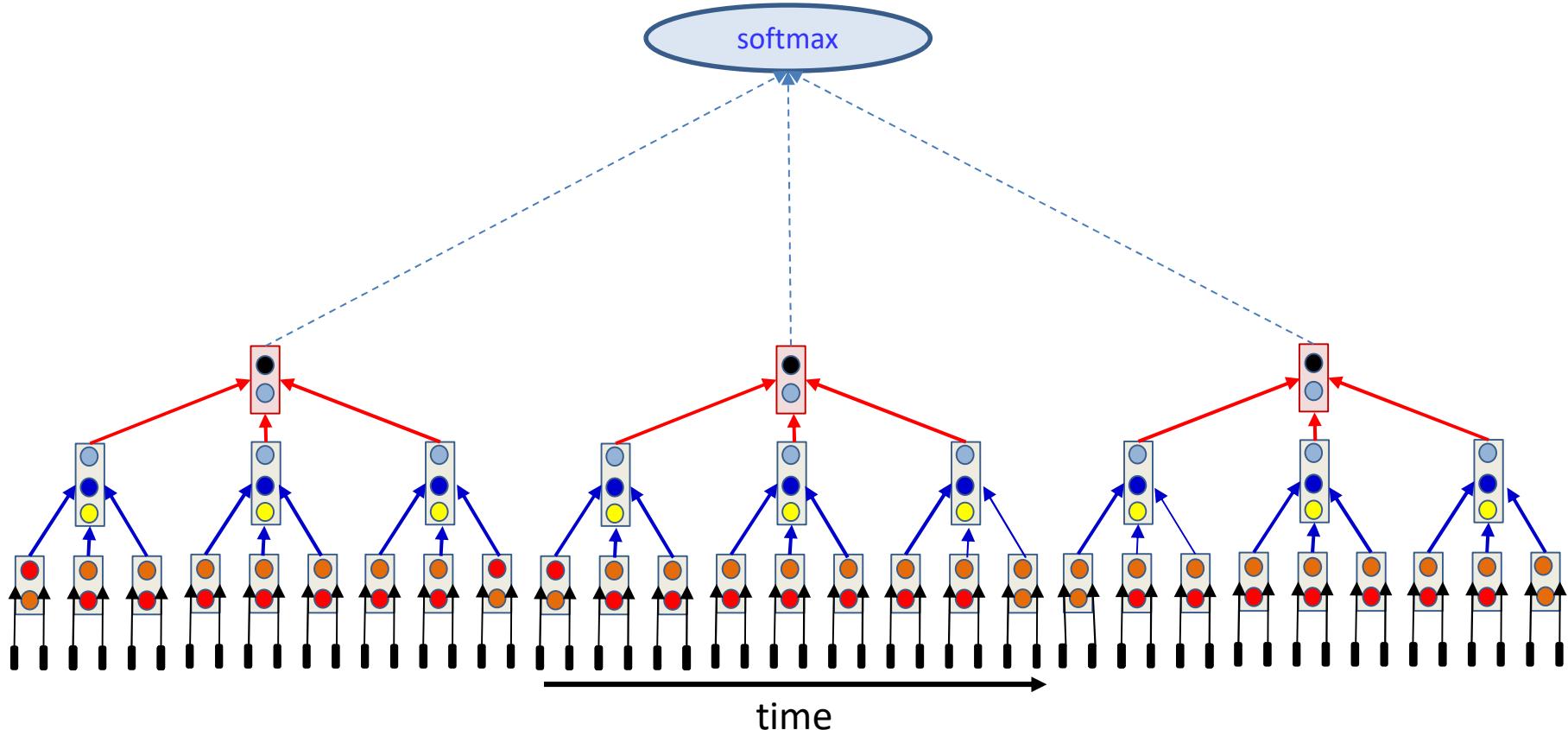
- The convolutional neural network is a supervised version of a computational model of mammalian vision
- It includes
  - Convolutional layers comprising learned filters that scan the outputs of the previous layer
  - Downsampling layers that operate over groups of outputs from the convolutional layer 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



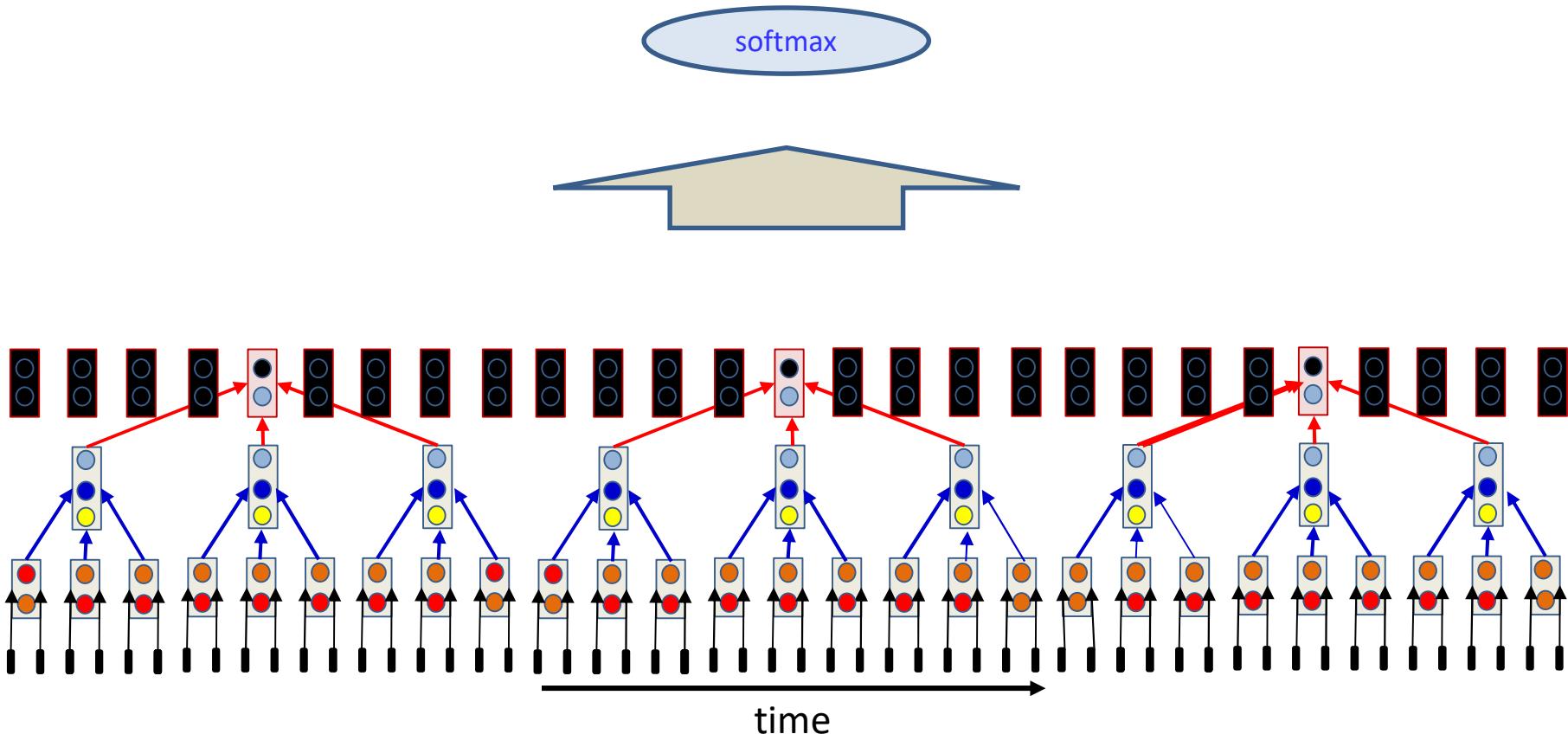
- We've always assumed that subsequent steps *shrink* the size of the maps
- Can subsequent maps *increase* in size?

# 1-D scans



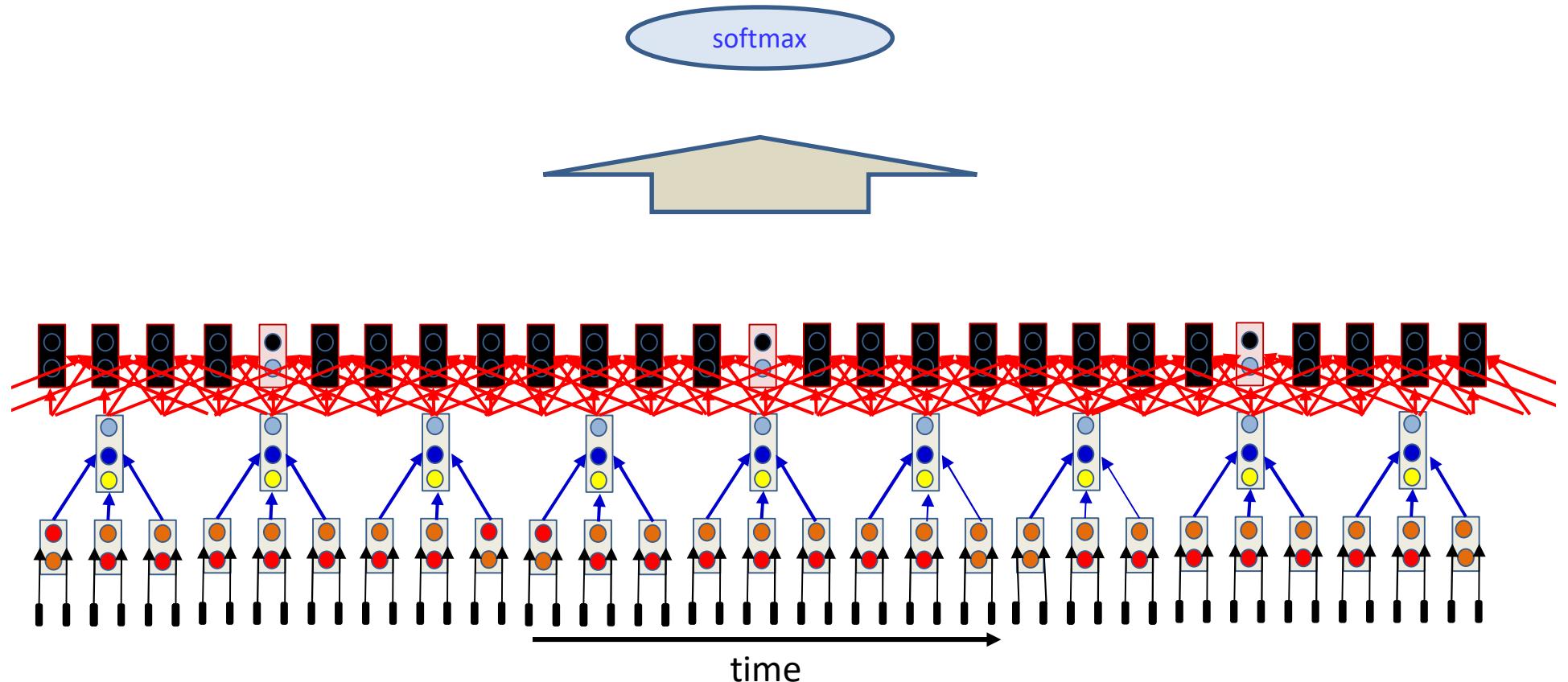
- The number of “bars” in each layer is usually the same or *smaller* than the bars in the previous layer
  - Scanning maintains or reduces the time resolution of the signal at each layer

# Upsampling 1-D scans



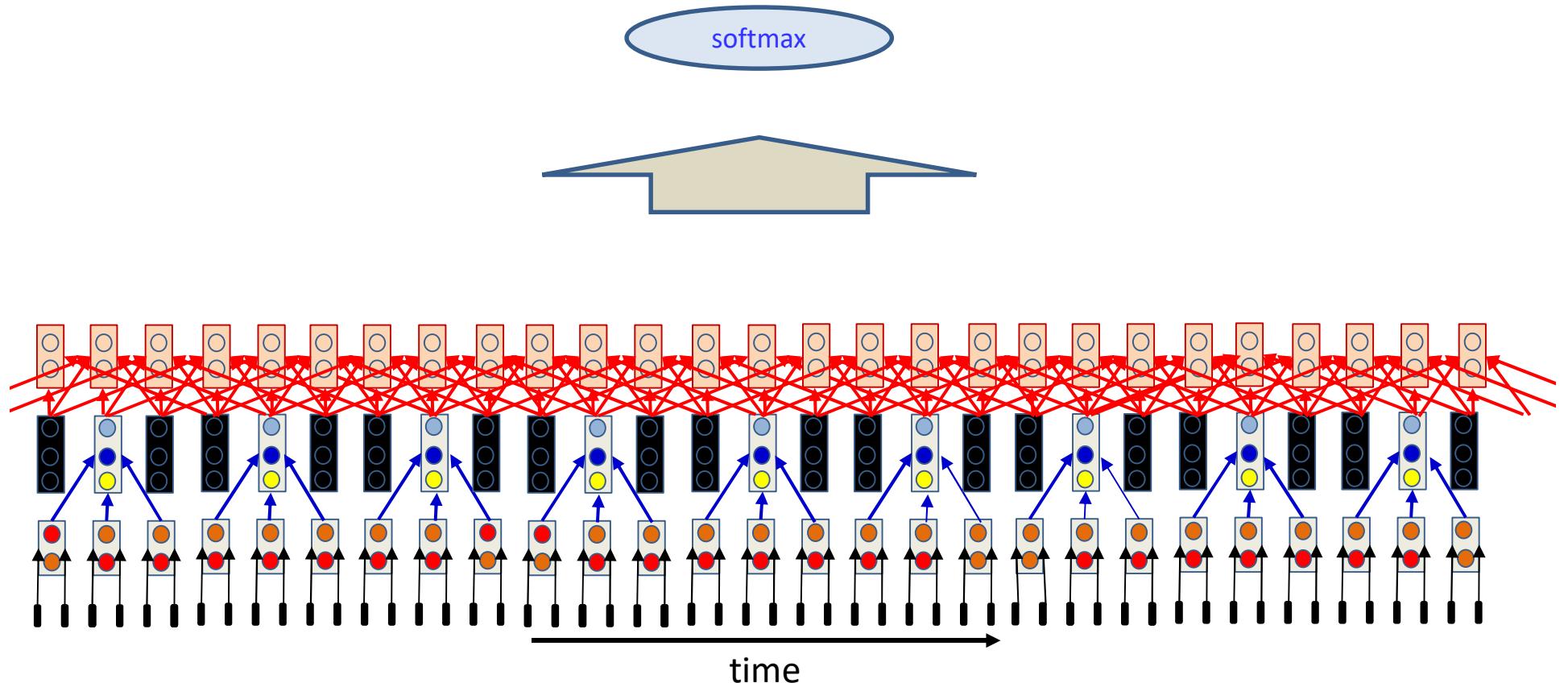
- The number of “bars” in each layer is usually the same or *smaller* than the bars in the previous layer
  - Scanning maintains or reduces the time resolution of the signal at each layer
- What if we want to *increase* the time resolution with layers?

# Upsampling 1-D scans



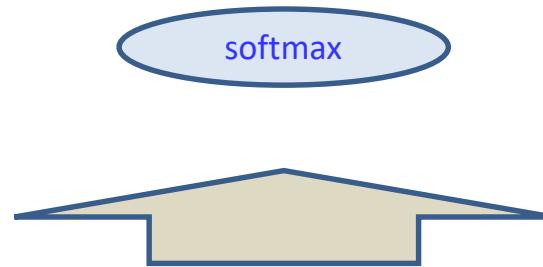
- **Problem:** The values required to compute the intermediate values are missing from the previous layer!

# Upsampling 1-D scans

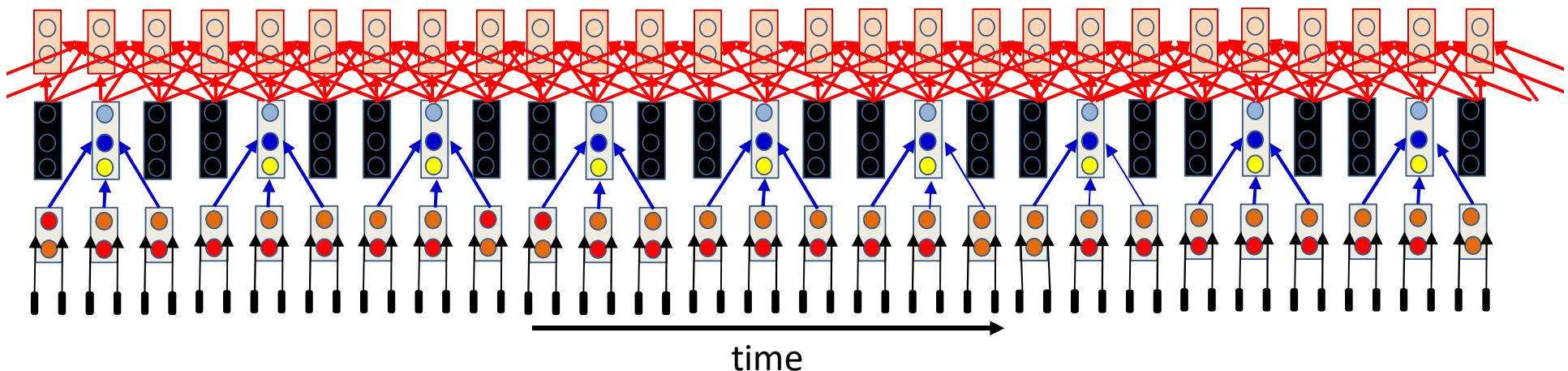


- **Problem:** The values required to compute the intermediate values are missing from the previous layer!
- **Solution:** Synthetically fill in the missing intermediate values of the previous layer
  - With zeros
    - Could also fill them in with linear or spline interpolation of neighbors, but it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

# Upsampling 1-D scans

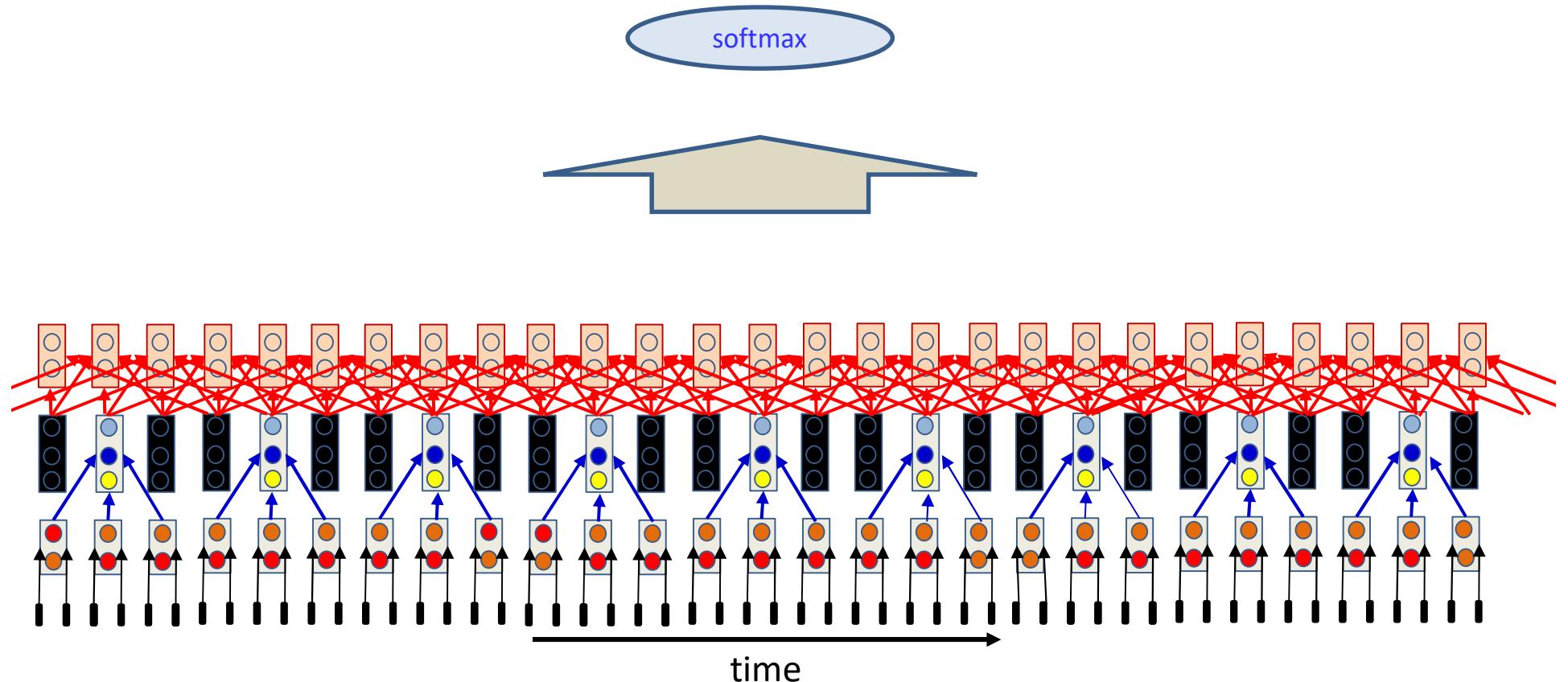


This is exactly analogous to the upsampling performed during backprop when forward convolution uses stride > 1



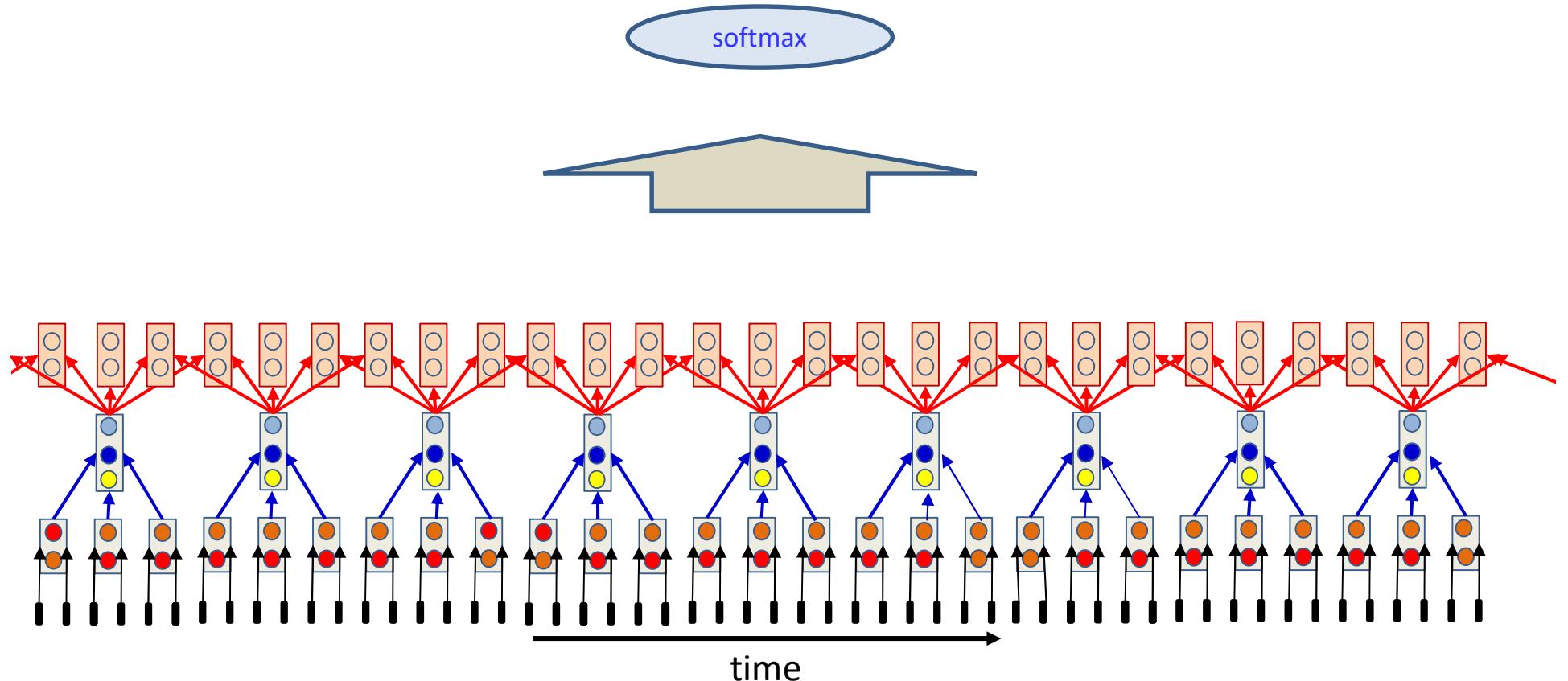
- **Problem:** The values required to compute the intermediate values are missing from the previous layer!
- **Solution:** Synthetically fill in the missing intermediate values of the previous layer
  - With zeros
    - Could also fill them in with linear or spline interpolation of neighbors, but it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

# Upsampling 1-D scans



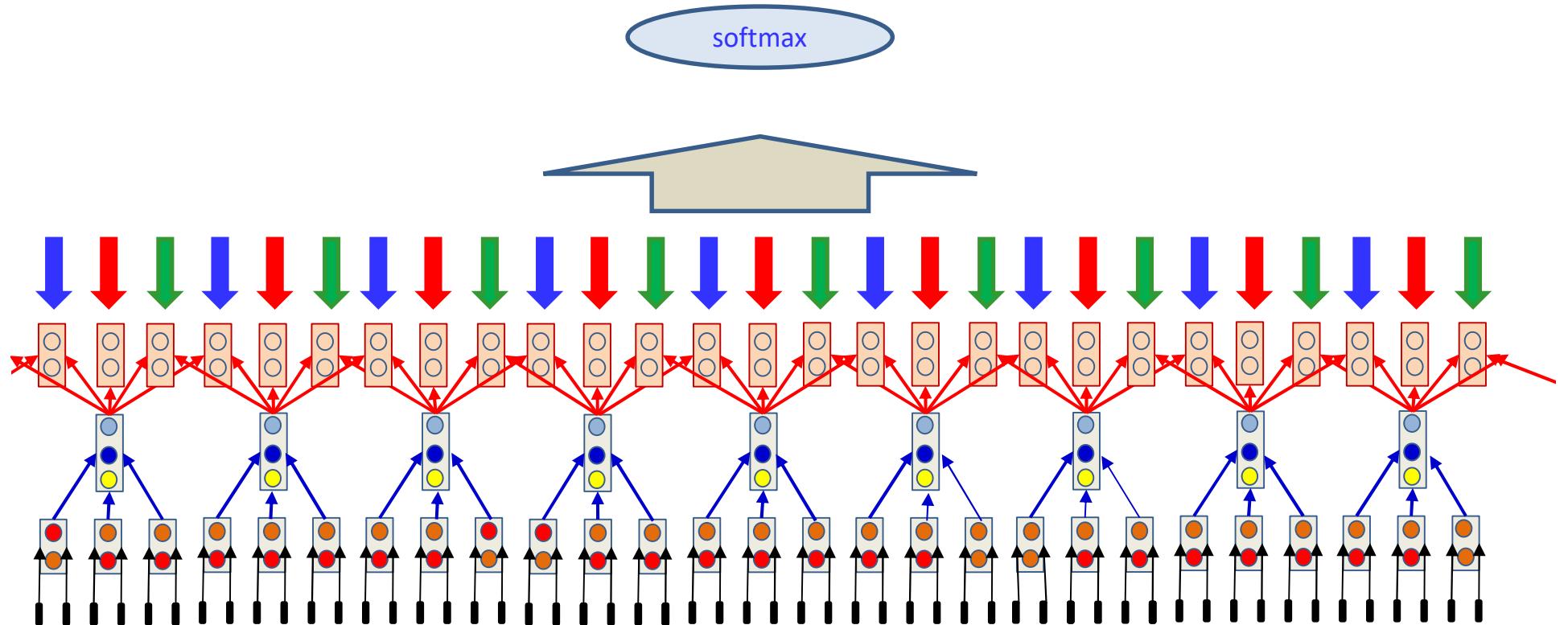
- The 0-valued interpolated inputs do not really provide any input
- They, and their connections can be removed without changing the computation

# Upsampling 1-D scans



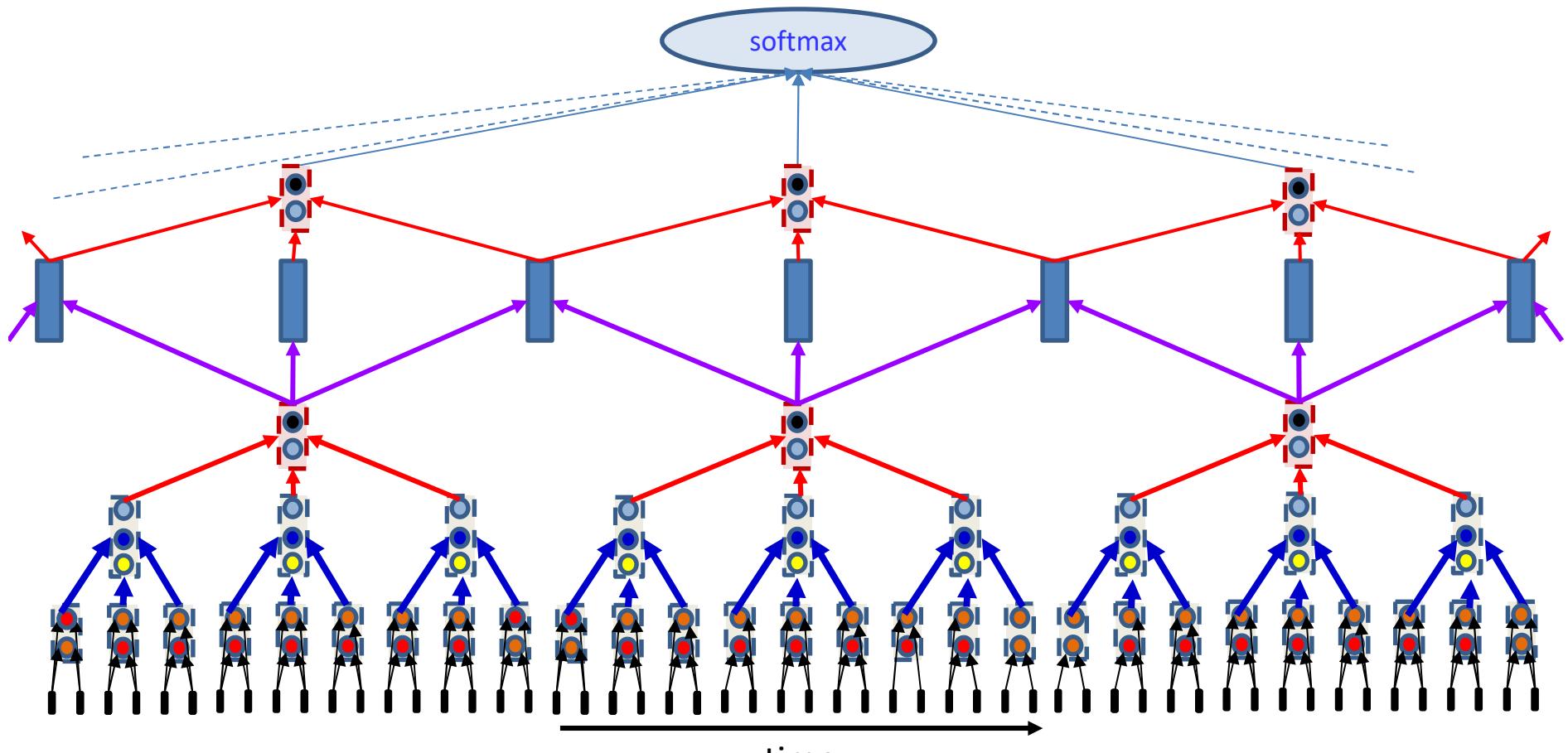
- The 0-valued interpolated inputs do not really provide any input
- They, and their connections can be removed without changing the computation
- *This is the actual computation performed*

# Upsampling 1-D scans



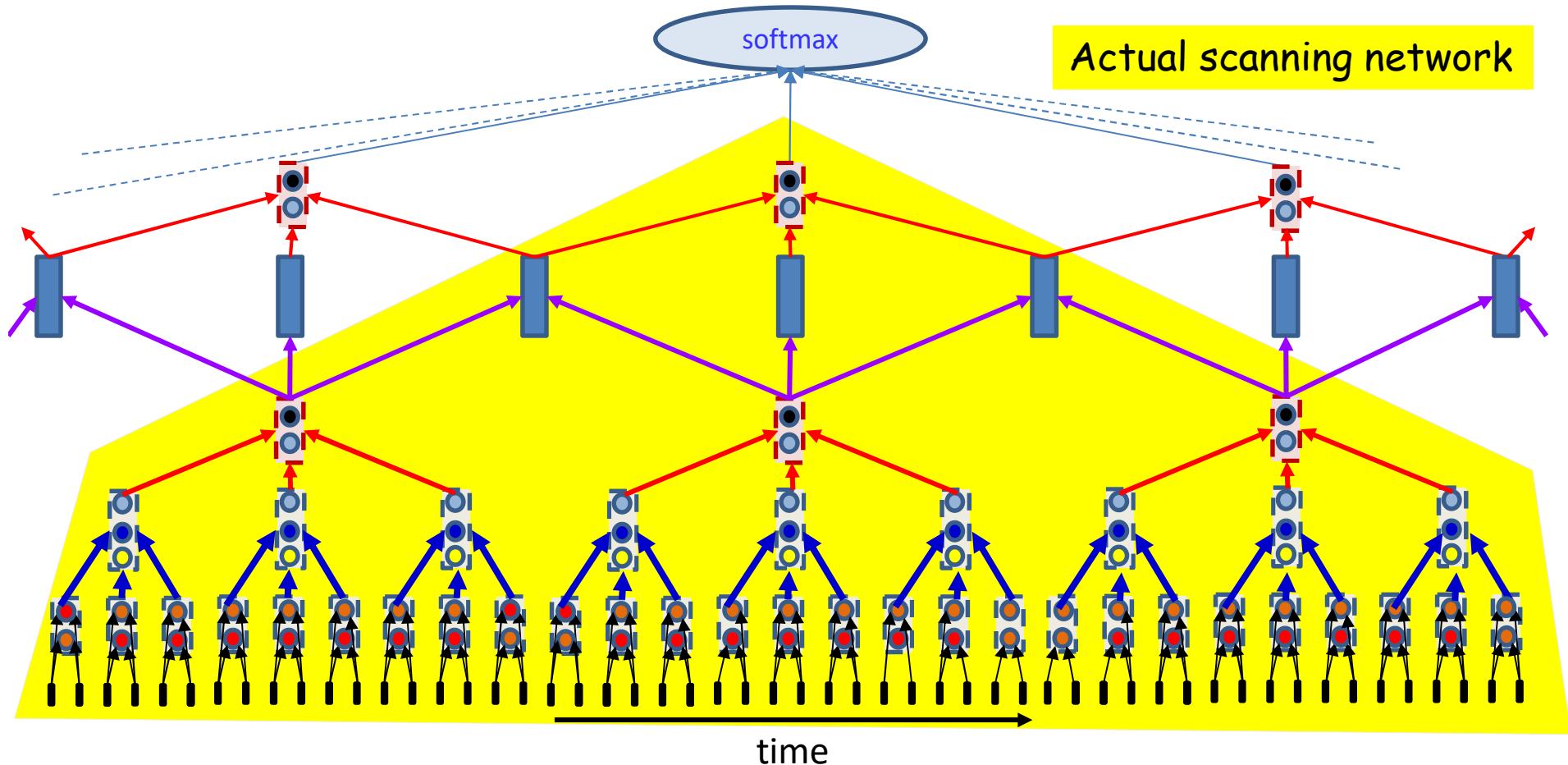
- Key difference from downsampling layers
  - All the “columns” in the regular/downsampling layers are identical
    - Their *incoming* weight patterns are identical
  - The columns in the upsampling layers are *not* identical
    - The *outgoing* weight patterns of the *lower* layer columns are identical

# Upsampling as a scanning network



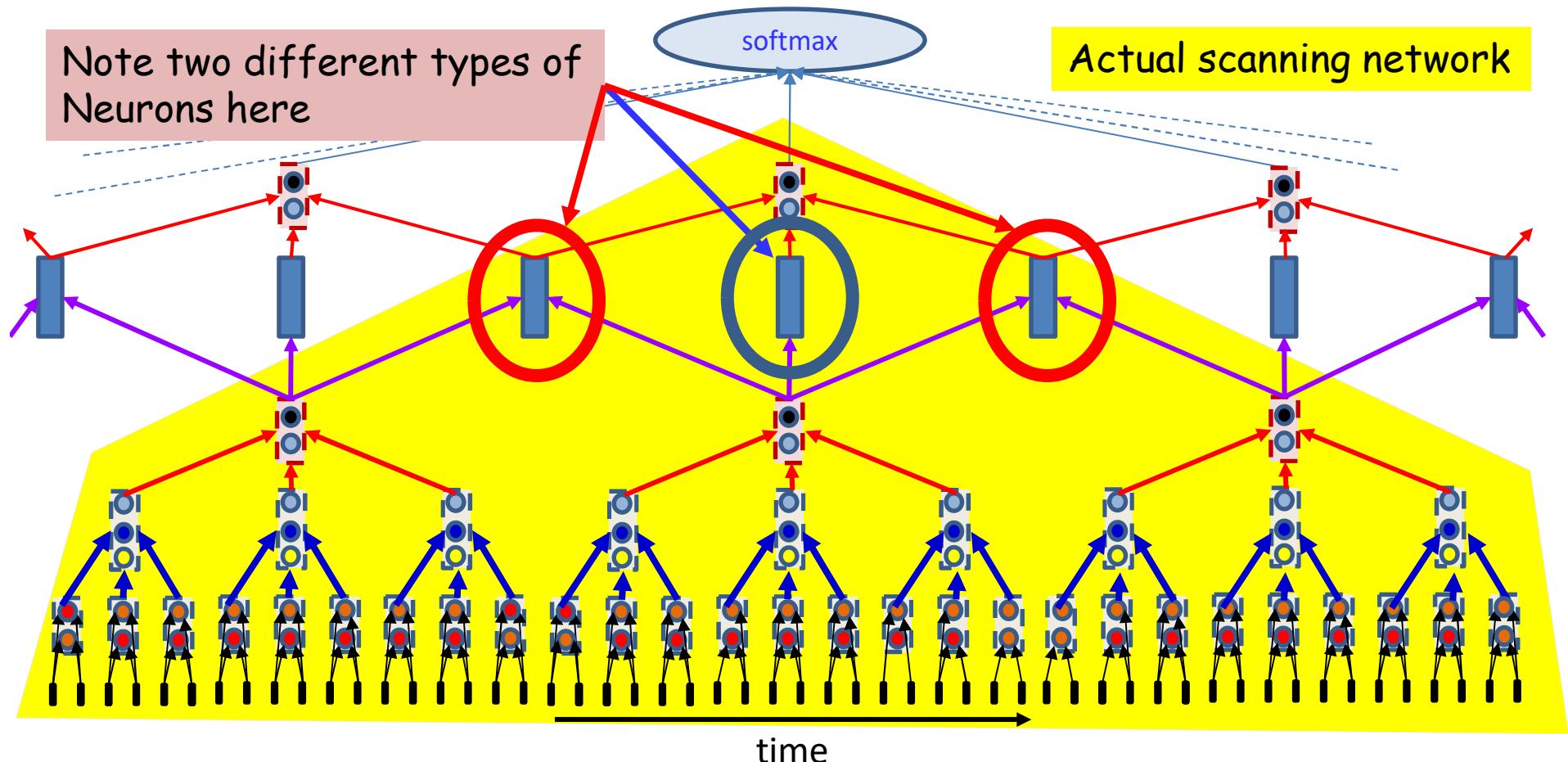
- Example of a network with one upsampling layer
- *Maintaining Symmetry:*
  - Vertical bars in the 4<sup>th</sup> layer are regularly arranged w.r.t. bars of layer 3
  - The pattern of values of upward weights for each of the three pink (3<sup>rd</sup> layer) bars is identical

# Upsampling as a scanning network



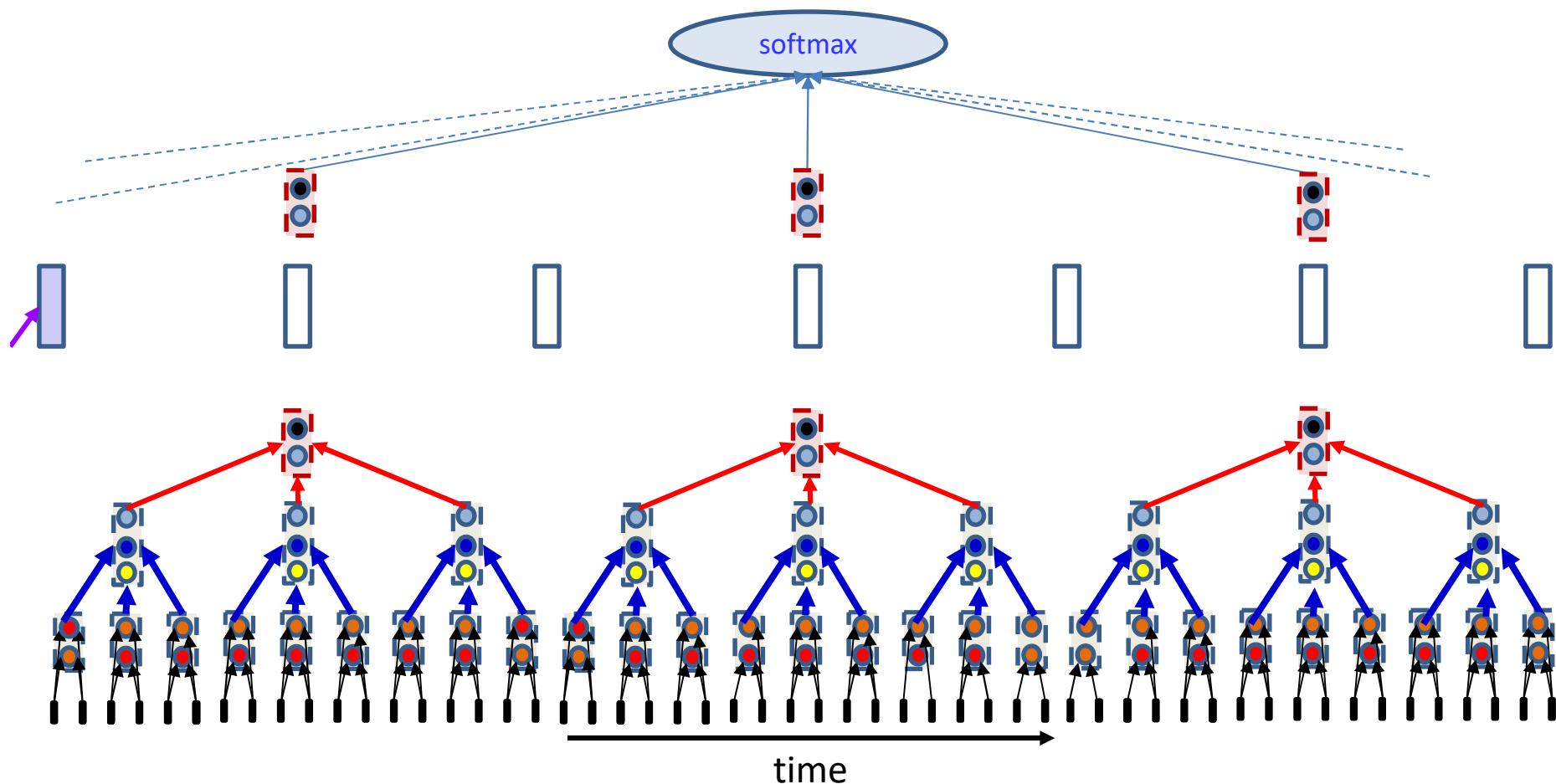
- *Maintaining Symmetry:*
  - Vertical bars in the 4<sup>th</sup> layer are regularly arranged w.r.t. bars of layer 3
  - The pattern of values of upward weights for each of the three pink (3<sup>rd</sup> layer) bars is identical

# Upsampling as a scanning network



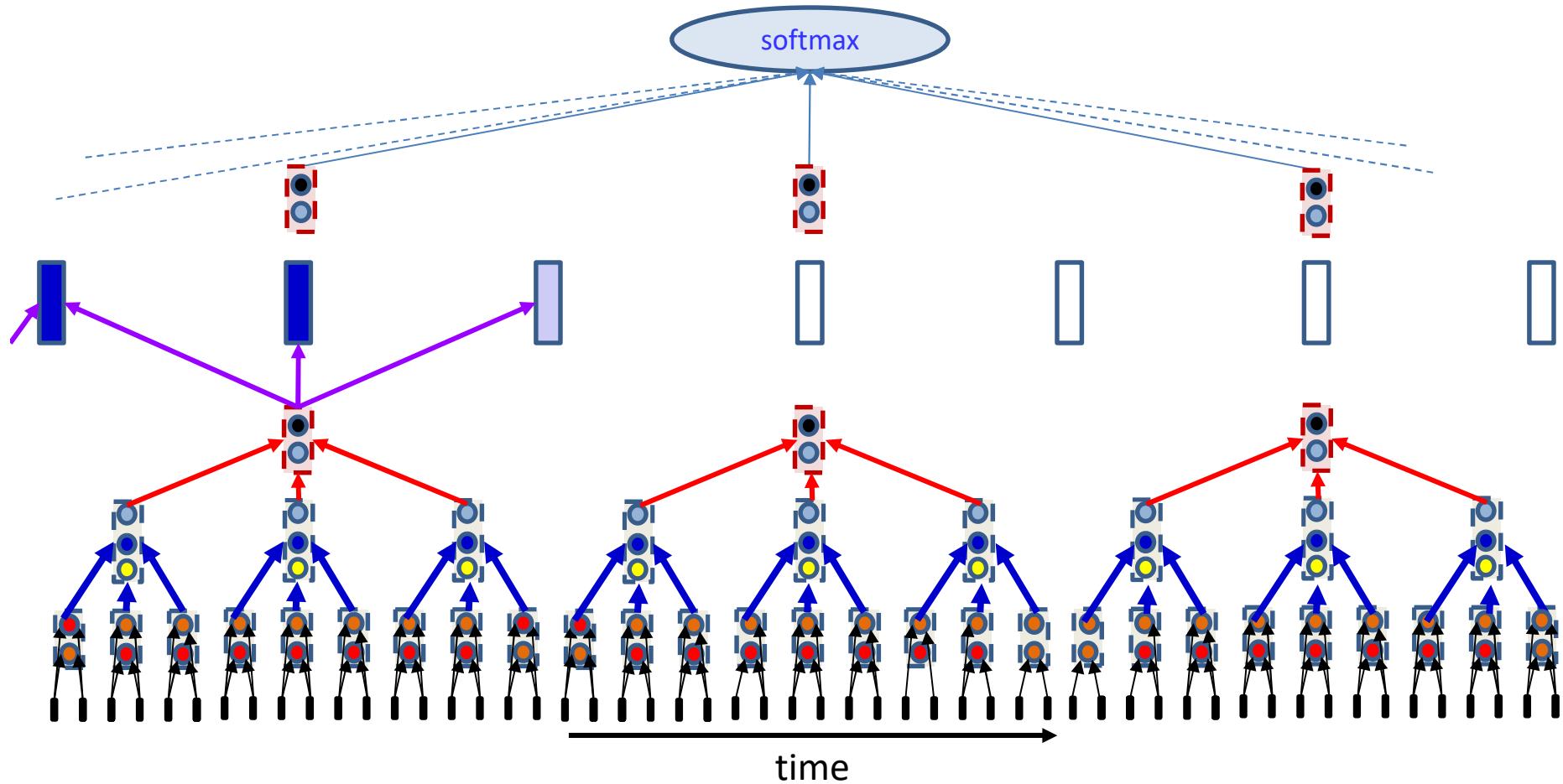
- *Maintaining Symmetry:*
  - Vertical bars in the 4<sup>th</sup> layer are regularly arranged w.r.t. bars of layer 3
  - The pattern of values of upward weights for each of the three pink (3<sup>rd</sup> layer) bars is identical

# Scanning with increased-res layer



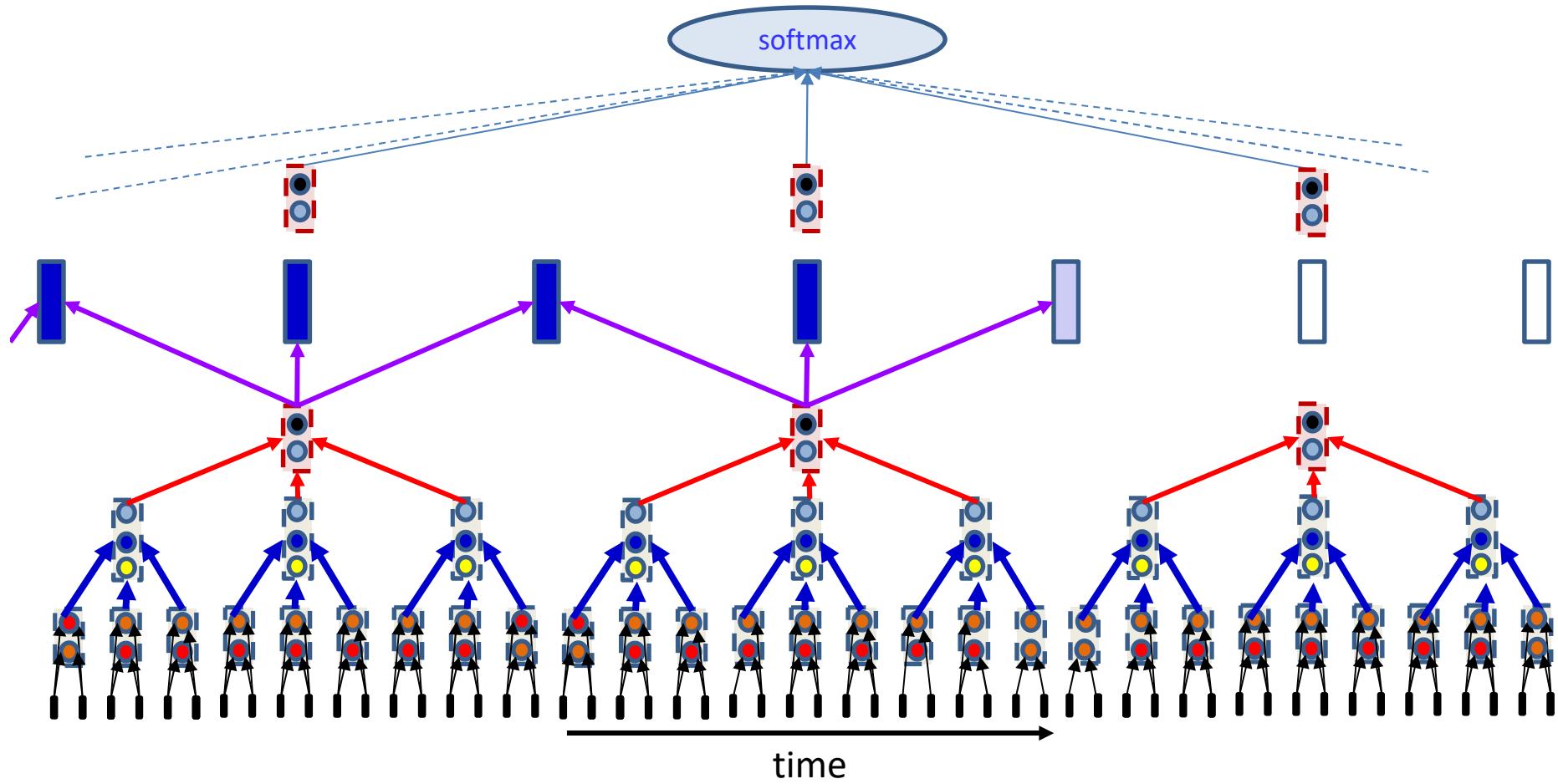
- Flow of info from bottom to top when implemented as a left-to-right scan
  - Note: Arrangement of vertical bars is predetermined by architecture<sup>250</sup>

# With layer of increased size



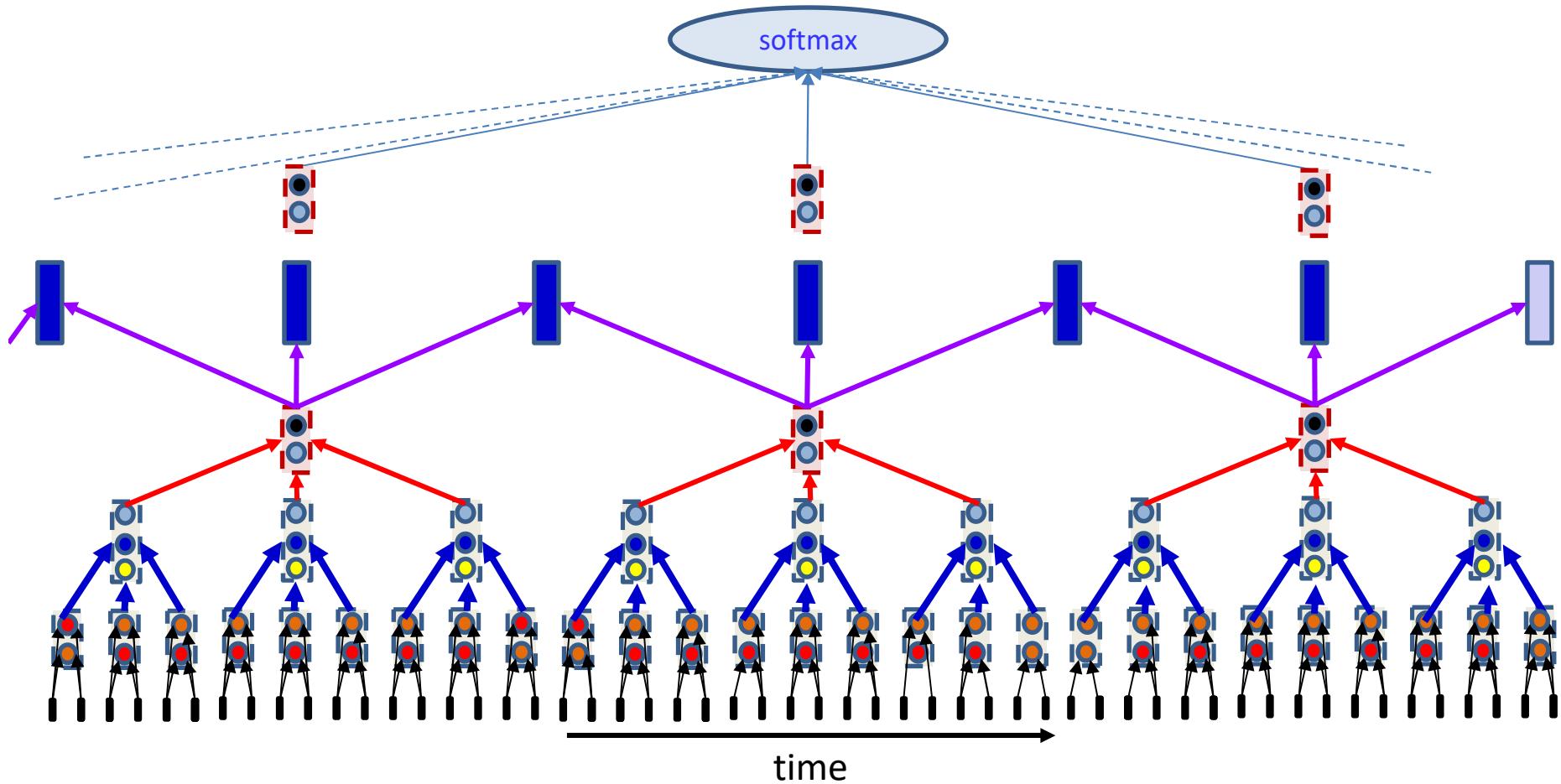
- Flow of info from bottom to top when implemented as a left-to-right scan
  - Note: Arrangement of vertical bars is predetermined by architecture<sup>251</sup>

# With layer of increased size



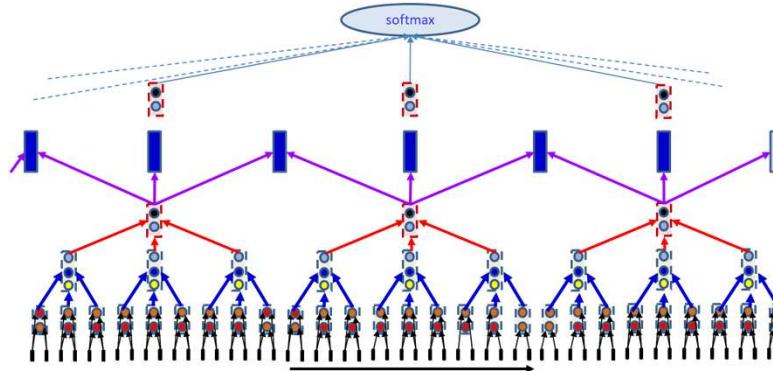
- Flow of info from bottom to top when implemented as a left-to-right scan
  - Note: Arrangement of vertical bars is predetermined by architecture<sup>252</sup>

# 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<sup>253</sup>

# Transposed convolution

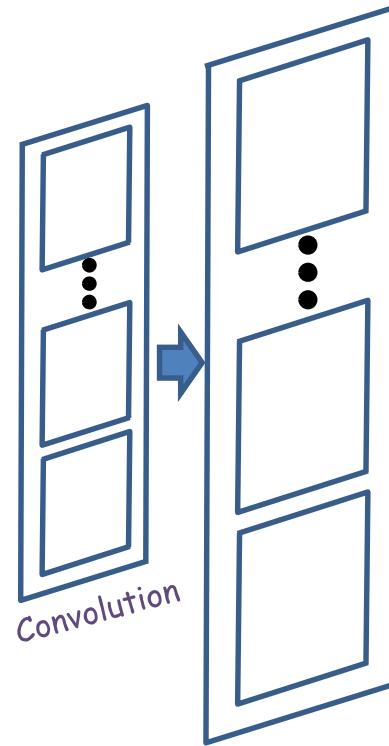


- Signal propagation rules are transposed for expanding layers
- In regular convolution, the affine value  $Z$  for a layer “pulls”  $Y$  values from the lower layer
  - In vector form
  - The  $i$ th neuron:
- In an upsampling layer the  $Y$  values are “pushed” to the upper  $Z$

$$Z_l = \sum_j W_l(:, j) Y_{l-1}(j)$$

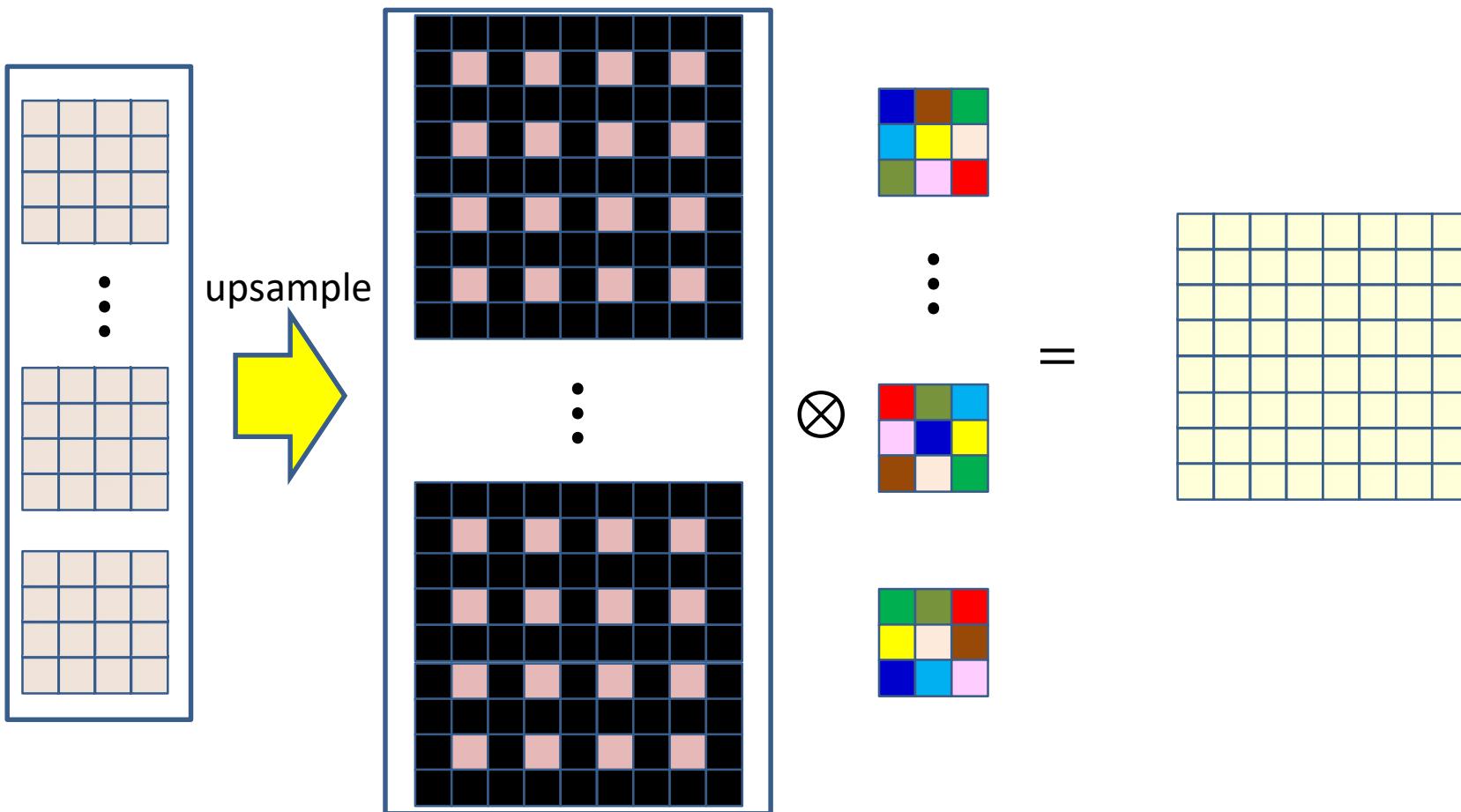
- Invokes the  $j$ th column of  $W_l$
- Or alternately, the  $j$ th row of  $W_l^T$
- Expanding operations are sometimes called *transpose* convolutions as a result
  - The primary operation uses the *transpose* of the convolutional filter

# In 2-D



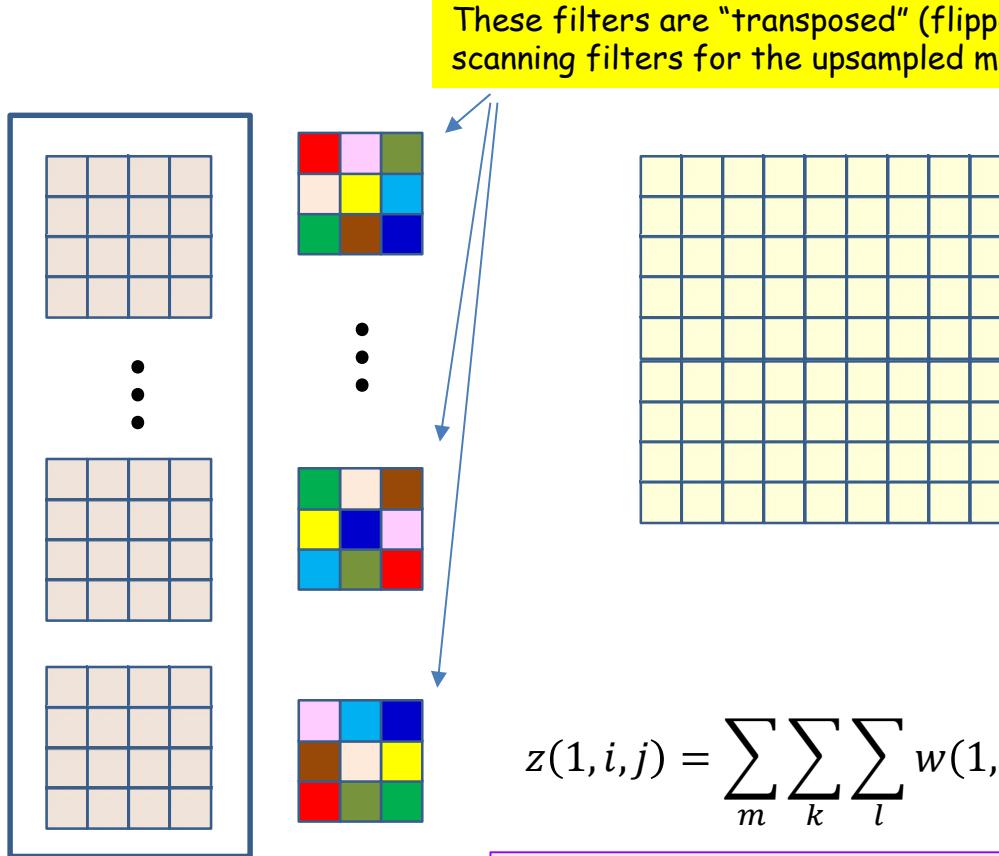
- Similar computation

# 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

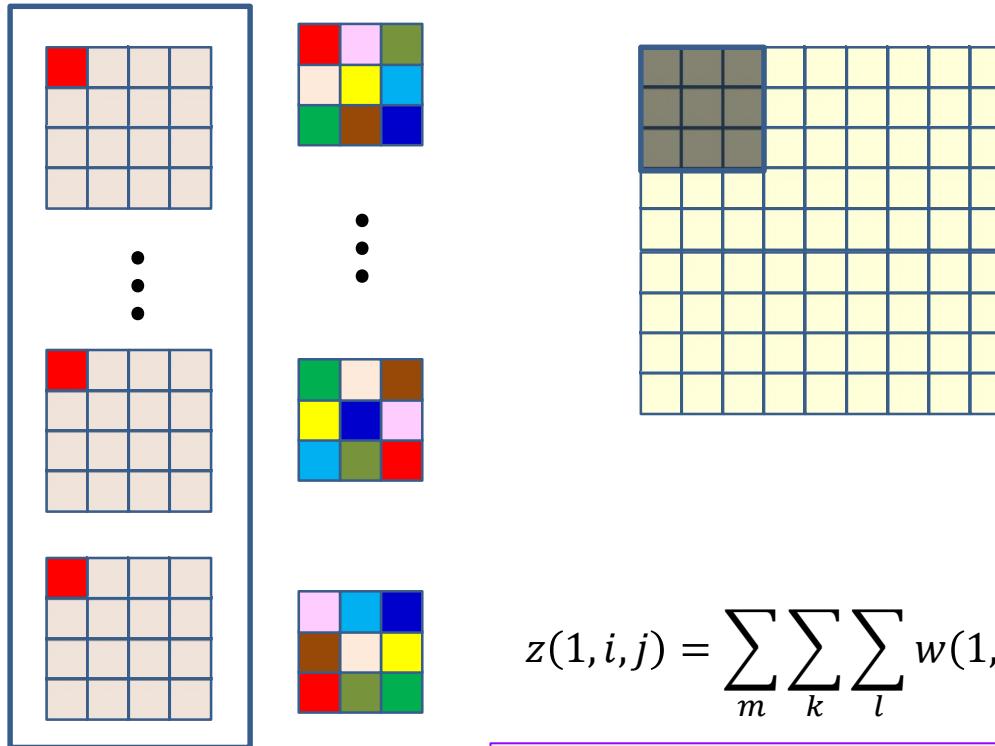


$$z(1, i, j) = \sum_m \sum_k \sum_l w(1, m, i - kb, j - lb) I(m, k, l)$$

*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

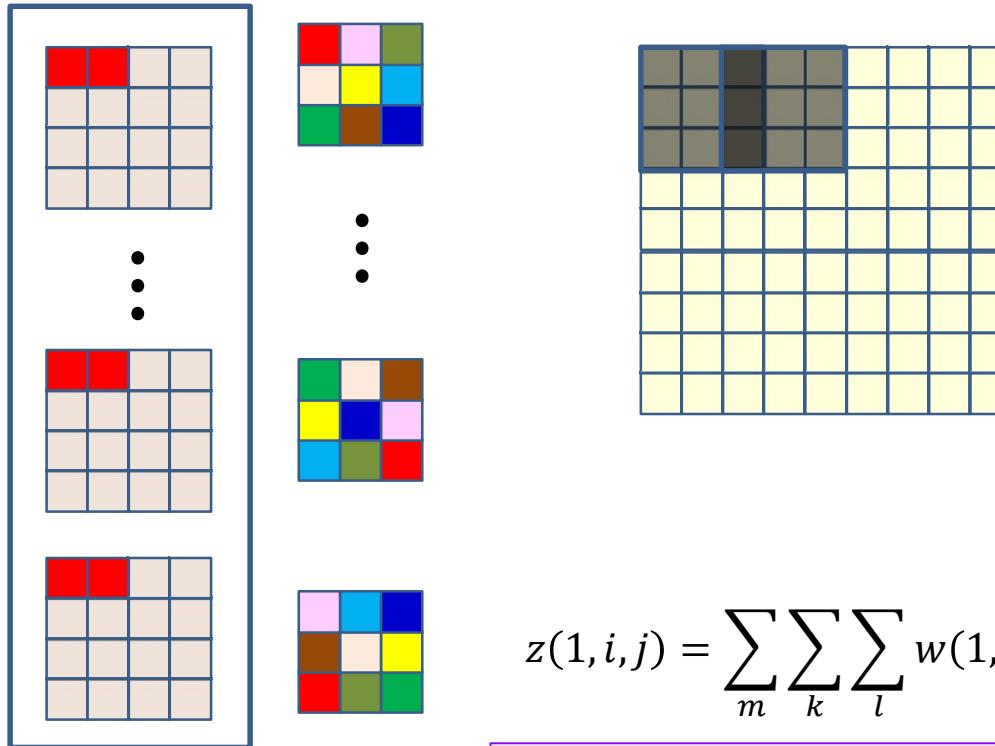


$$z(1, i, j) = \sum_m \sum_k \sum_l w(1, m, i - kb, j - lb) I(m, k, l)$$

*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:  $(bH + (K - 1)) \times (bW + (K - 1))$
  - Adjust filter stride and filter stride, and crop output map to ensure it is the right size

# 2D expanding convolution in practice

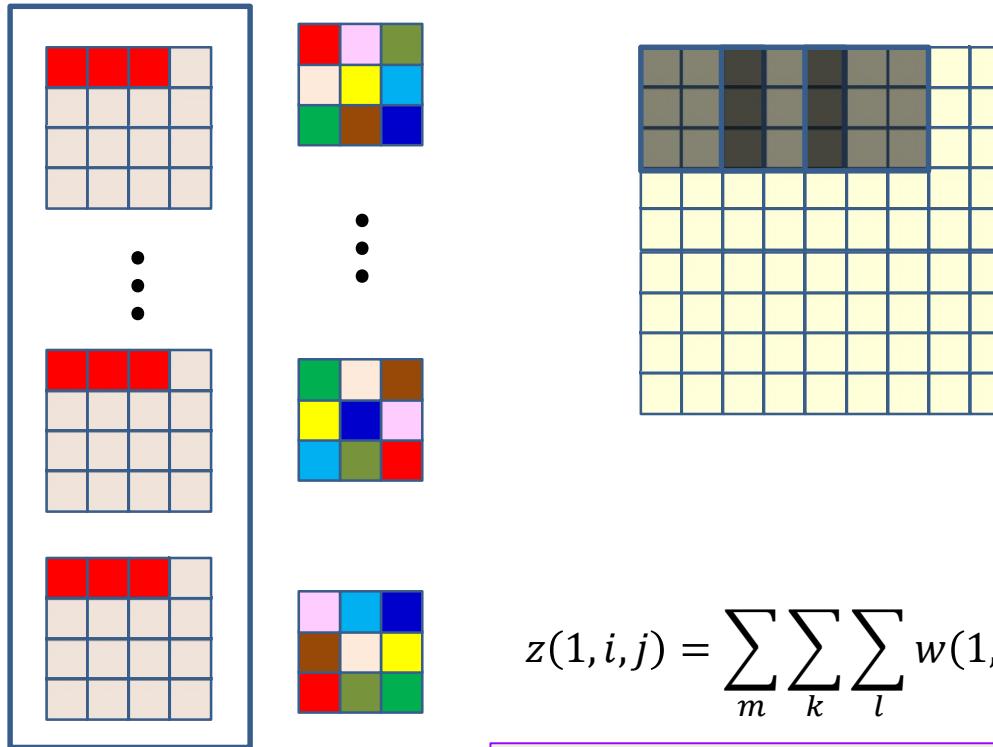


$$z(1, i, j) = \sum_m \sum_k \sum_l w(1, m, i - kb, j - lb) I(m, k, l)$$

*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:  $(bH + (K - 1)) \times (bW + (K - 1))$
  - Adjust filter stride and filter stride, and crop output map to ensure it is the right size

# 2D expanding convolution in practice

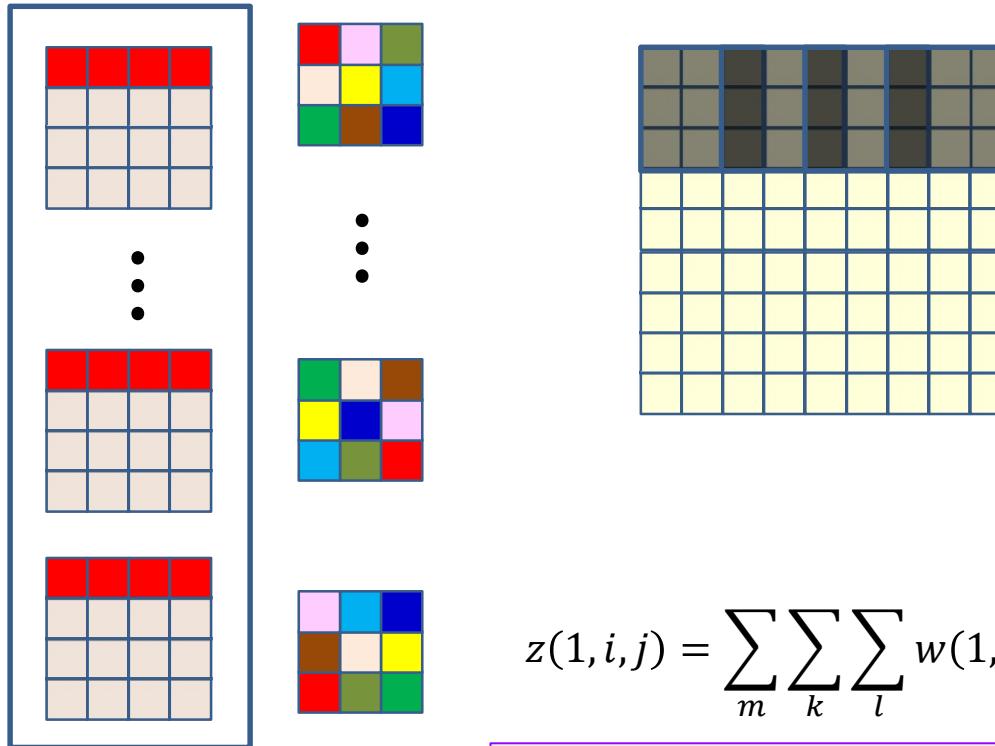


$$z(1, i, j) = \sum_m \sum_k \sum_l w(1, m, i - kb, j - lb) I(m, k, l)$$

*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:  $(bH + (K - 1)) \times (bW + (K - 1))$
  - Adjust filter stride and filter stride, and crop output map to ensure it is the right size

# 2D expanding convolution in practice

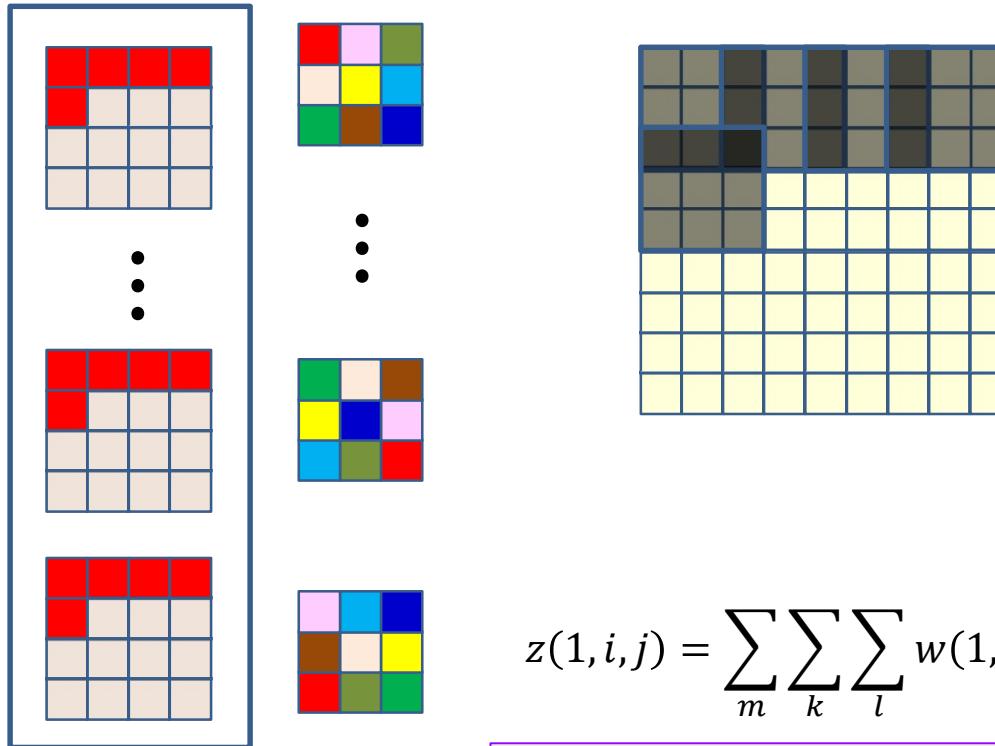


$$z(1, i, j) = \sum_m \sum_k \sum_l w(1, m, i - kb, j - lb) I(m, k, l)$$

*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:  $(bH + (K - 1)) \times (bW + (K - 1))$
  - Adjust filter stride and filter stride, and crop output map to ensure it is the right size

# 2D expanding convolution in practice

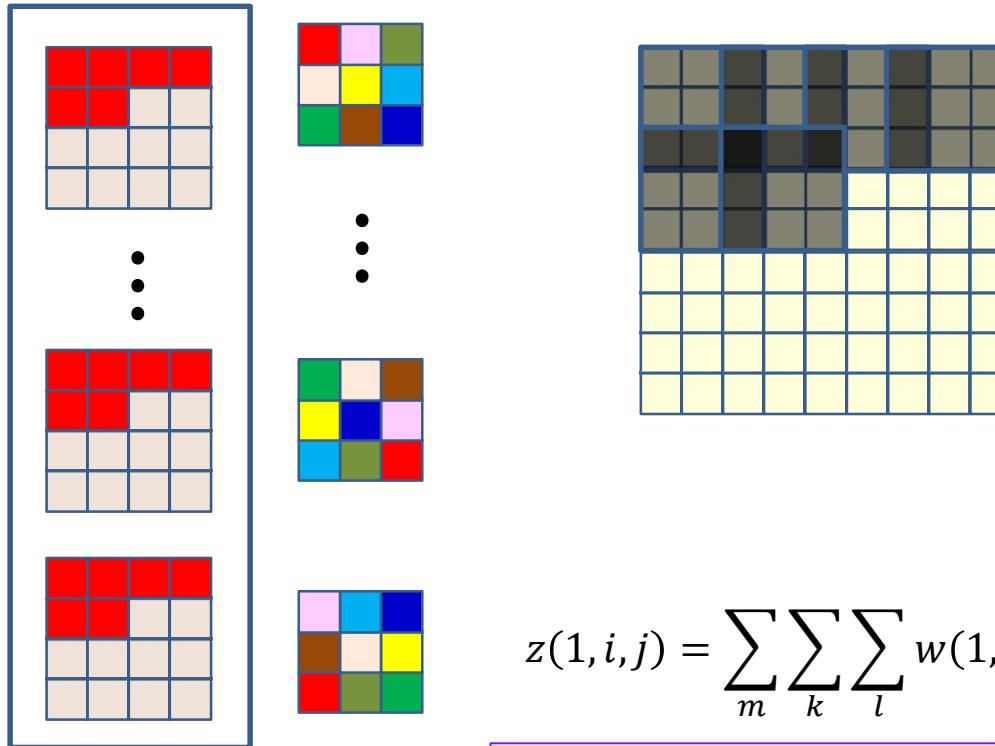


$$z(1, i, j) = \sum_m \sum_k \sum_l w(1, m, i - kb, j - lb) I(m, k, l)$$

*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:  $(bH + (K - 1)) \times (bW + (K - 1))$
  - Adjust filter stride and filter stride, and crop output map to ensure it is the right size

# 2D expanding convolution in practice

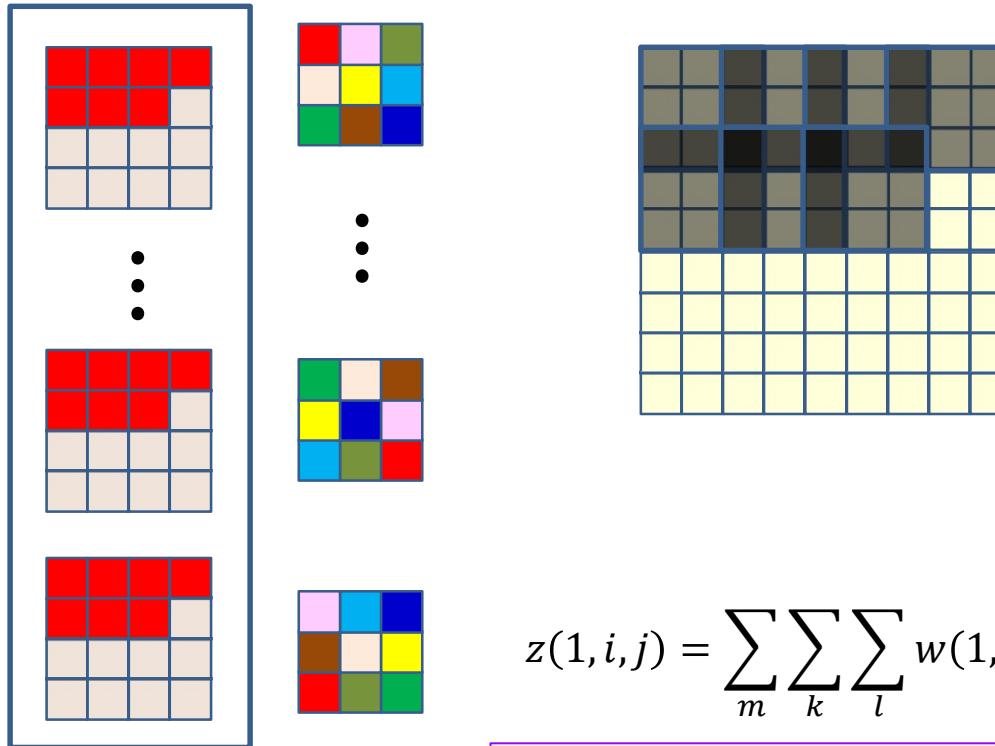


$$z(1, i, j) = \sum_m \sum_k \sum_l w(1, m, i - kb, j - lb) I(m, k, l)$$

*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:  $(bH + (K - 1)) \times (bW + (K - 1))$
  - Adjust filter stride and filter stride, and crop output map to ensure it is the right size

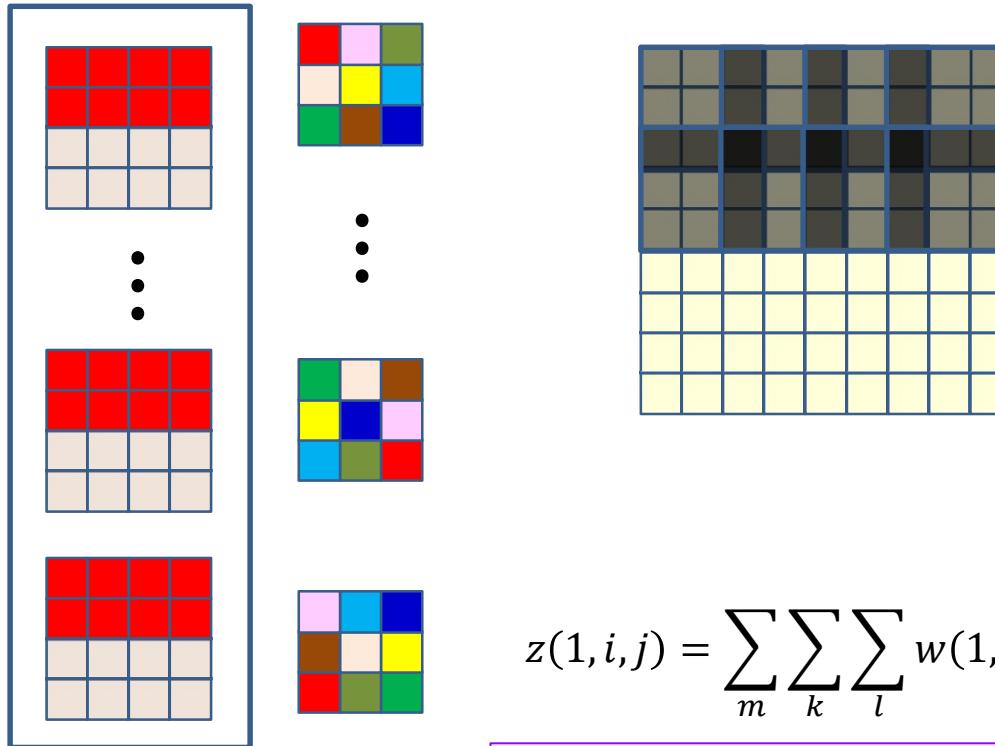
# 2D expanding convolution in practice



*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

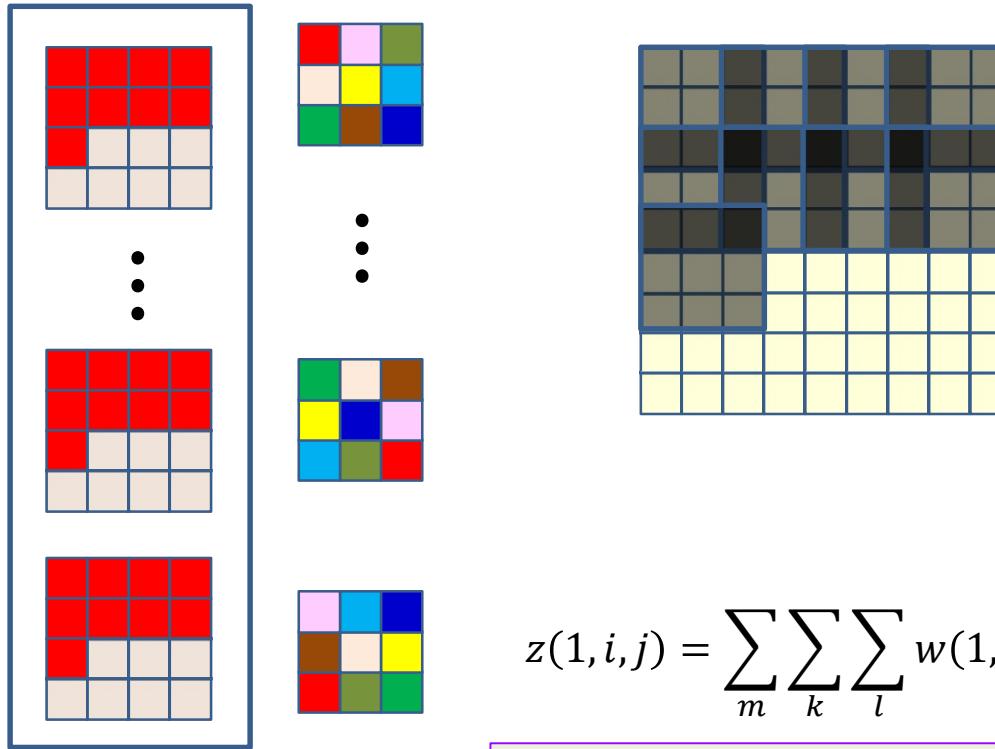


$$z(1, i, j) = \sum_m \sum_k \sum_l w(1, m, i - kb, j - lb) I(m, k, l)$$

*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:  $(bH + (K - 1)) \times (bW + (K - 1))$
  - Adjust filter stride and filter stride, and crop output map to ensure it is the right size

# 2D expanding convolution in practice

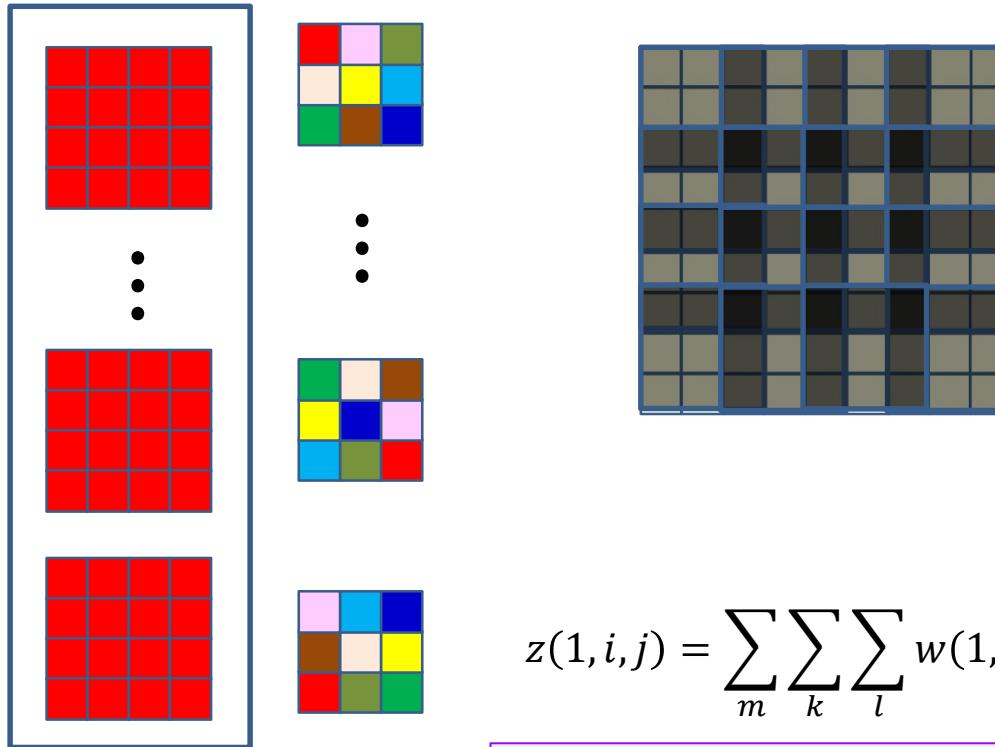


$$z(1, i, j) = \sum_m \sum_k \sum_l w(1, m, i - kb, j - lb) I(m, k, l)$$

*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:  $(bH + (K - 1)) \times (bW + (K - 1))$
  - 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)

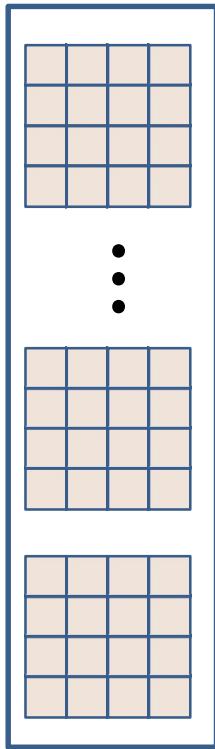
- The parameters are filter size and output stride
- Output size is primarily decided by filter stride
  - Edges padded by  $K - 1$  rows/columns ( $K$  is width of filter)
  - Size of new map:  $(bH + (K - 1)) \times (bW + (K - 1))$
  - Adjust filter stride and filter stride, and crop output map to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

# CNN: Expanding convolution layer $l$

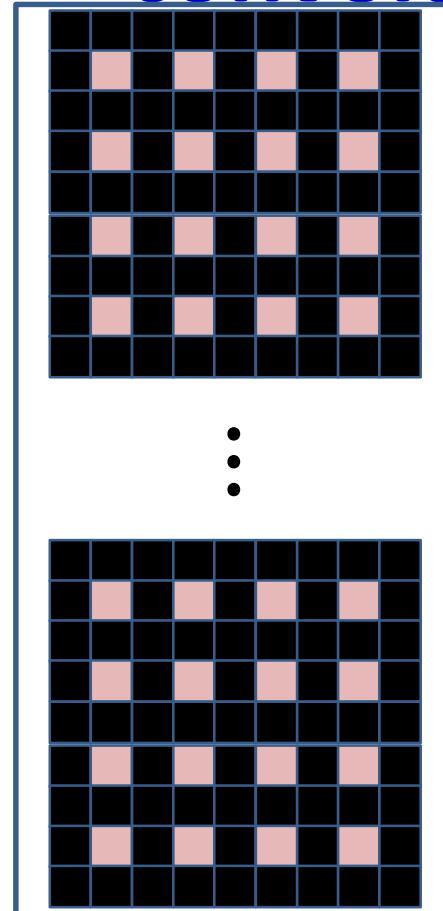
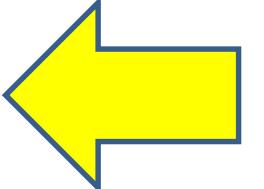
```
Z(l) = zeros(Dl x ((W-1)b+Kl) x ((H-1)b+Kl)) # b = stride
for j = 1:Dl
    for x = 1:W
        for y = 1:H
            for i = 1:Dl-1
                for x' = 1:Kl
                    for y' = 1:Kl
                        z(l,j,(x-1)b+x', (y-1)b+y') +=
                            w(l,j,i,x',y') y(l-1,i,x,y)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

# Backprop through expanding convolution

Derivative for  
 $Y(l - 1)$



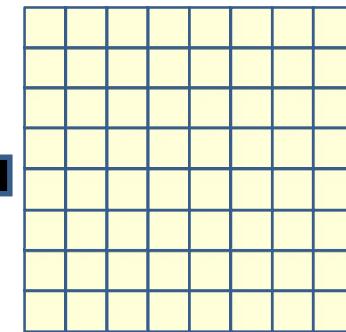
downsample



backprop



Derivative  
 $Z(l)$



- 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+Kl) x ((H-1)b+Kl)) # b = stride
for j = 1:Dl
    for x = 1:W
        for y = 1:H
            for i = 1:Dl-1
                for x' = 1:Kl
                    for y' = 1:Kl
                        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

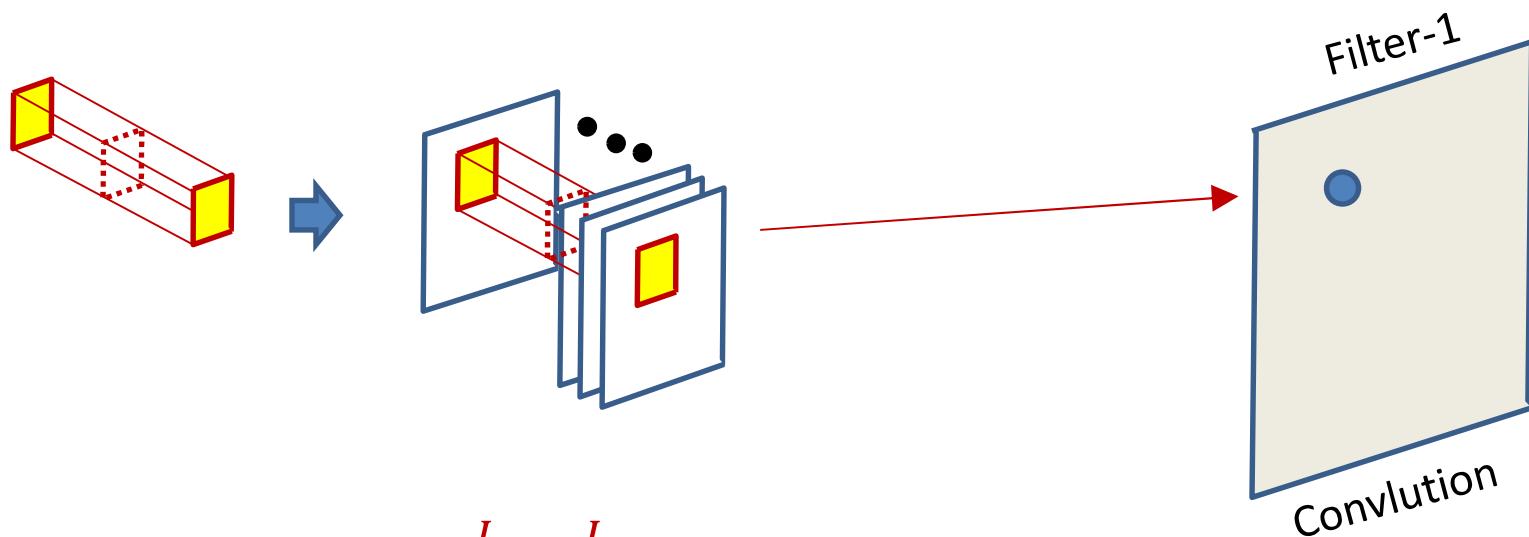
# Invariance



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



# Shift-invariance – a different perspective

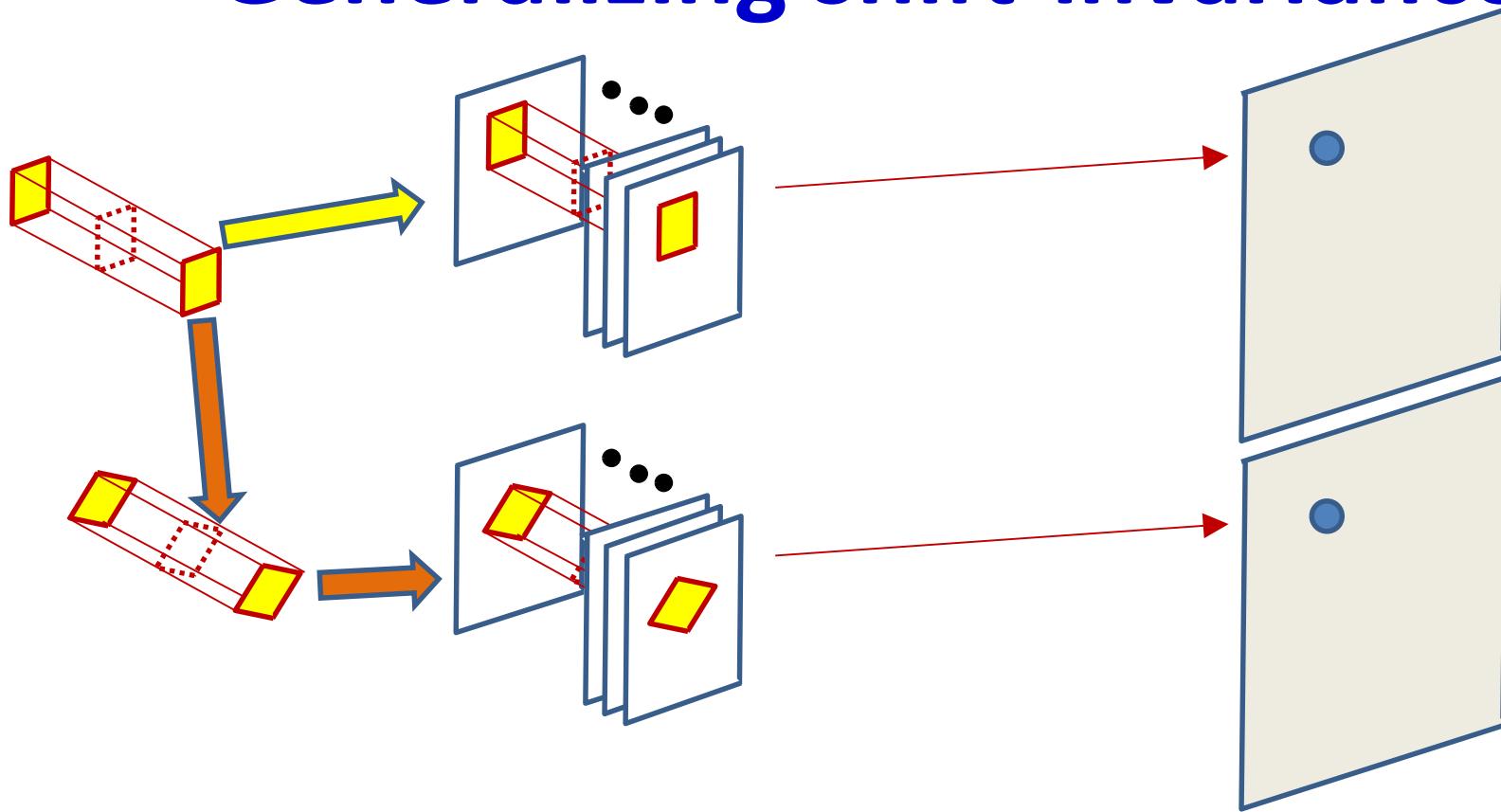


$$z(l, s, i, j) = \sum_p \sum_{k=1}^L \sum_{m=1}^L w(l, s, p, k, m) Y(l - 1, p, i + k, j + m)$$

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

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

# Generalizing shift-invariance



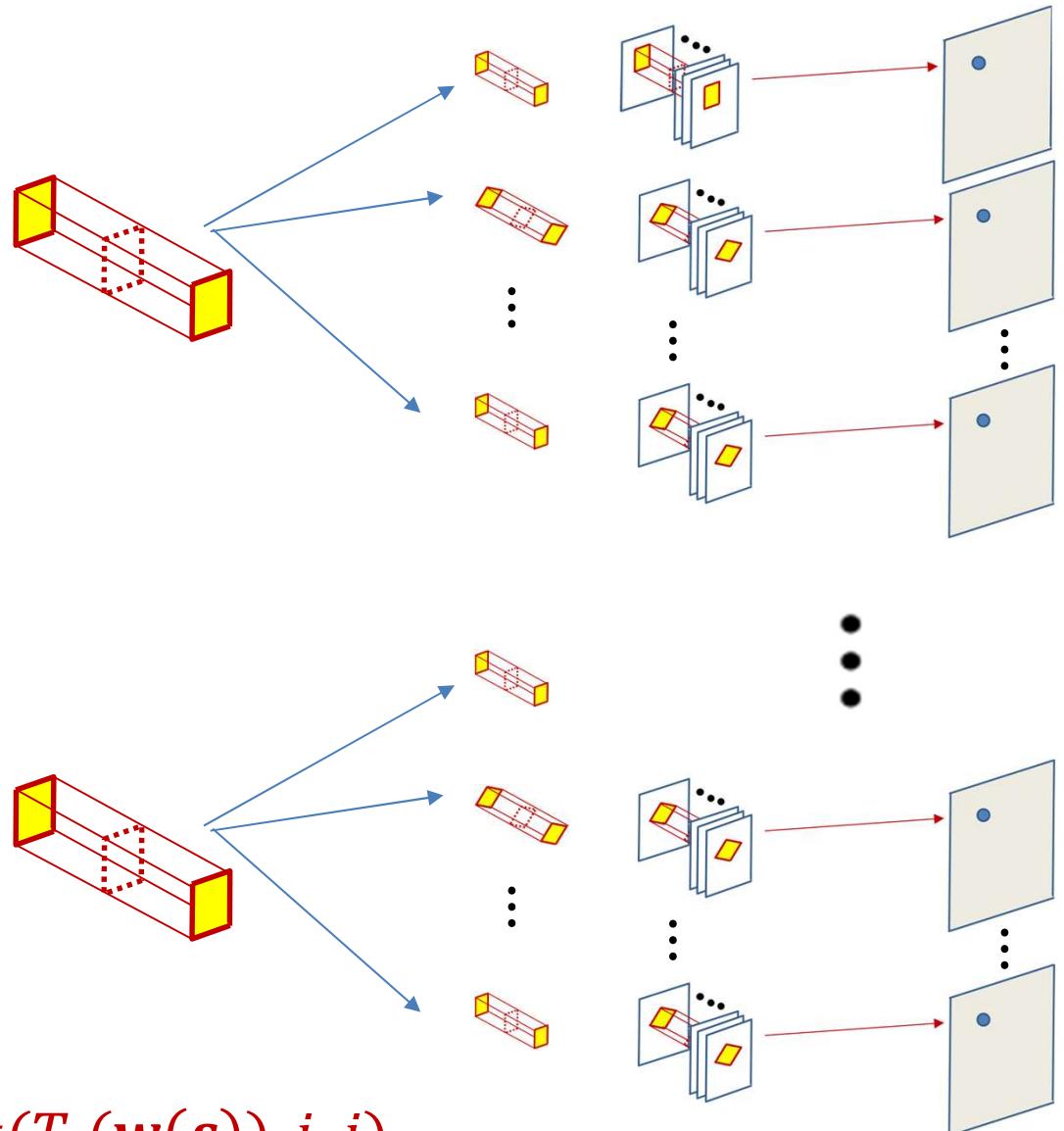
$$z_{regular}(s, i, j) = Y.\text{shift}(\mathbf{w}(s), i, j)$$

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

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

# 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



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

# Regular CNN : single layer $l$

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

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

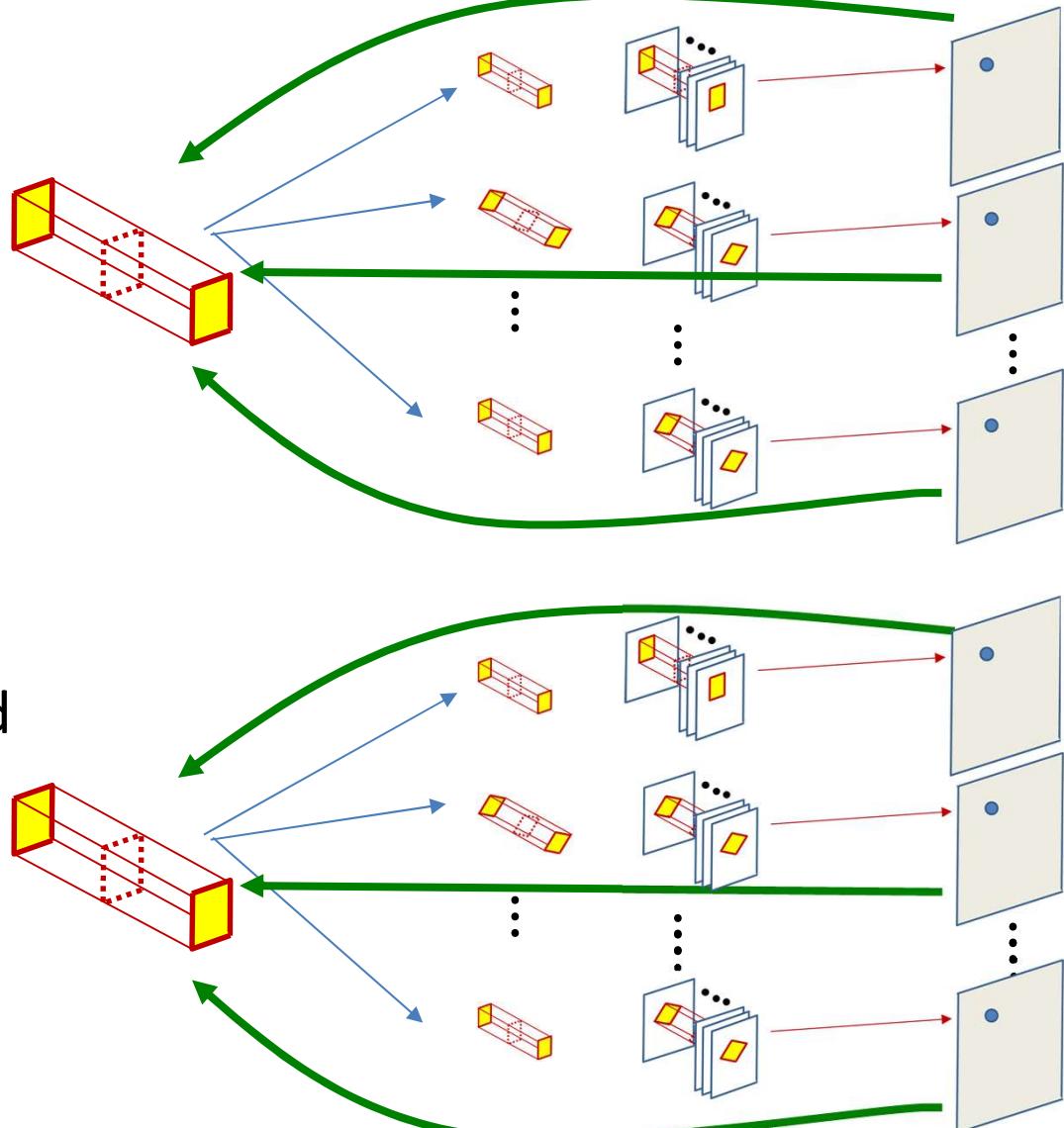
# Transform invariance

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

```
for x = 1:Wl-1-Kl+1
    for y = 1:Hl-1-Kl+1
        m = 1
        for j = 1:Dl
            for t in {Transforms} # enumerated transforms
                TW = T(W(l, j))
                segment = Y(l-1, :, x:x+Kl-1, y:y+Kl-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

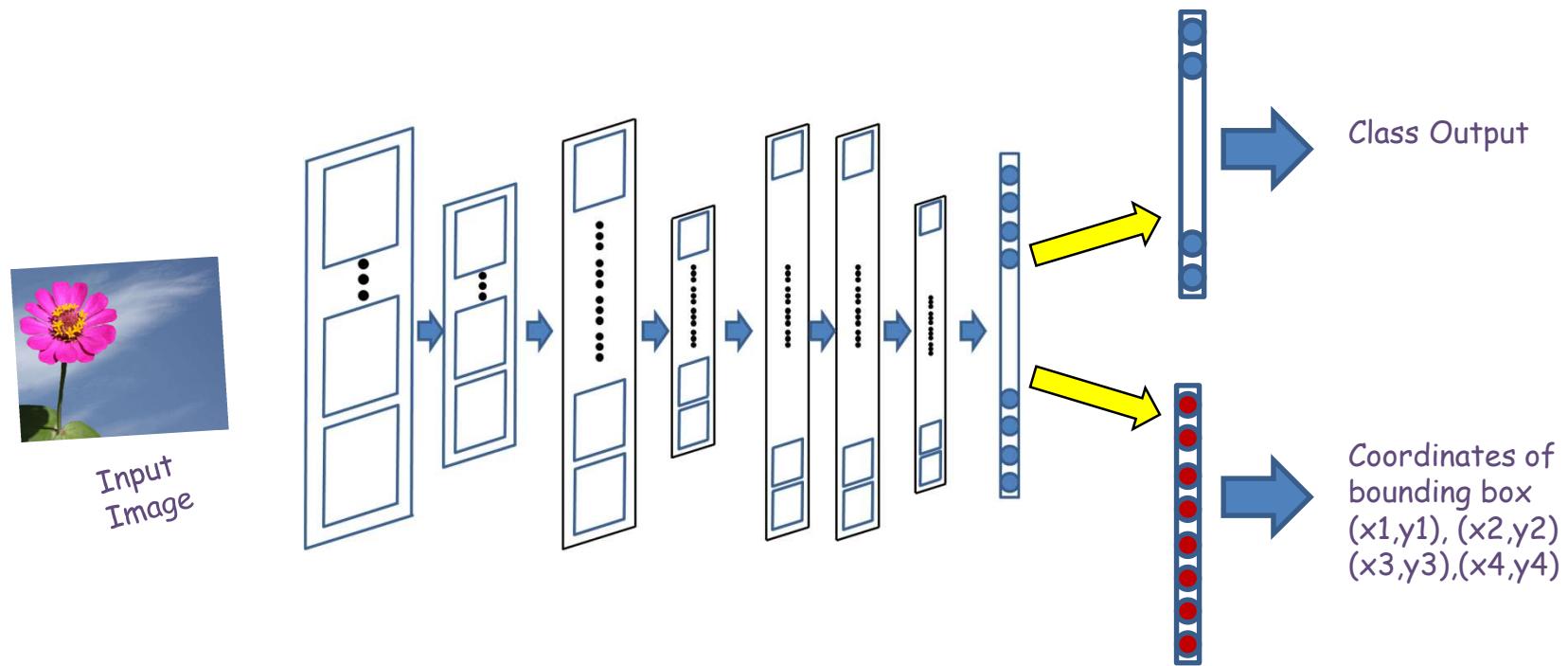
- CNNs are shift-invariant neural-network models for shift-invariant pattern detection
  - Are equivalent to scanning with shared-parameter MLPs with distributed representations
- The parameters of the network can be learned through regular back propagation
- Like a regular MLP, individual layers may 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?



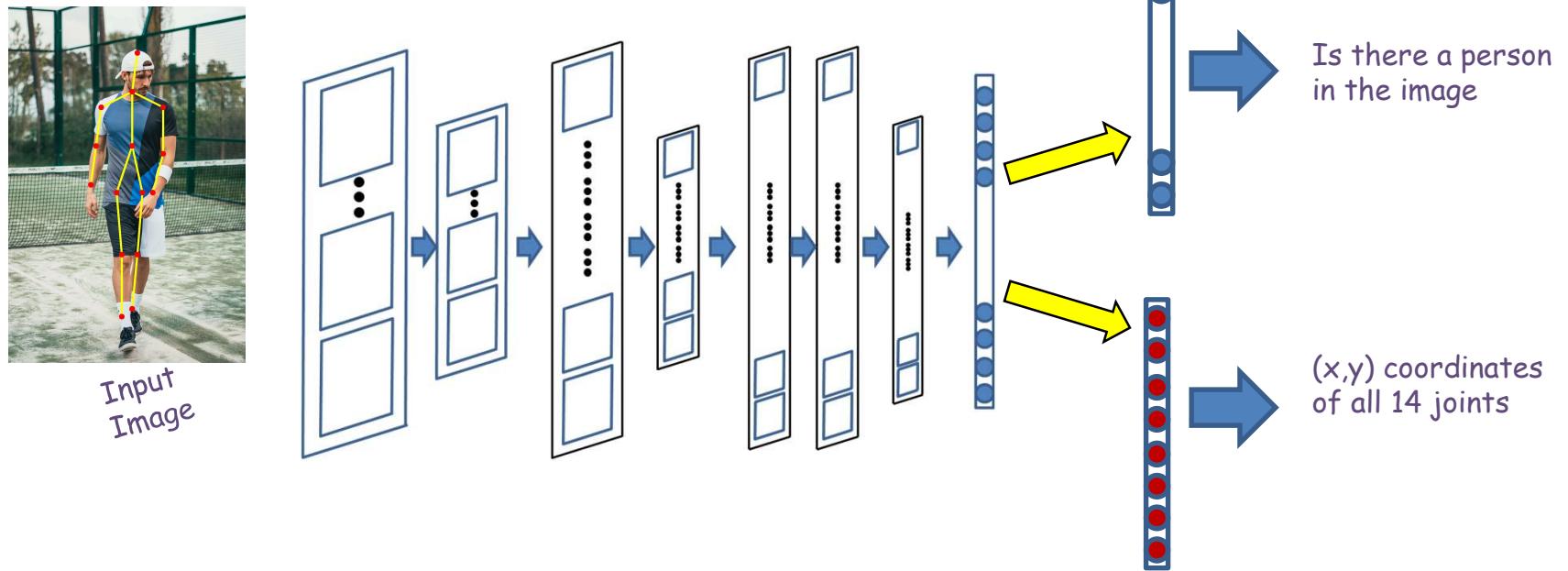
- We began with the desire to identify the picture as containing a flower, regardless of the position of the flower
  - Or more generally the class of object in the picture
- But can we detect the *position* of the main object?

# 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



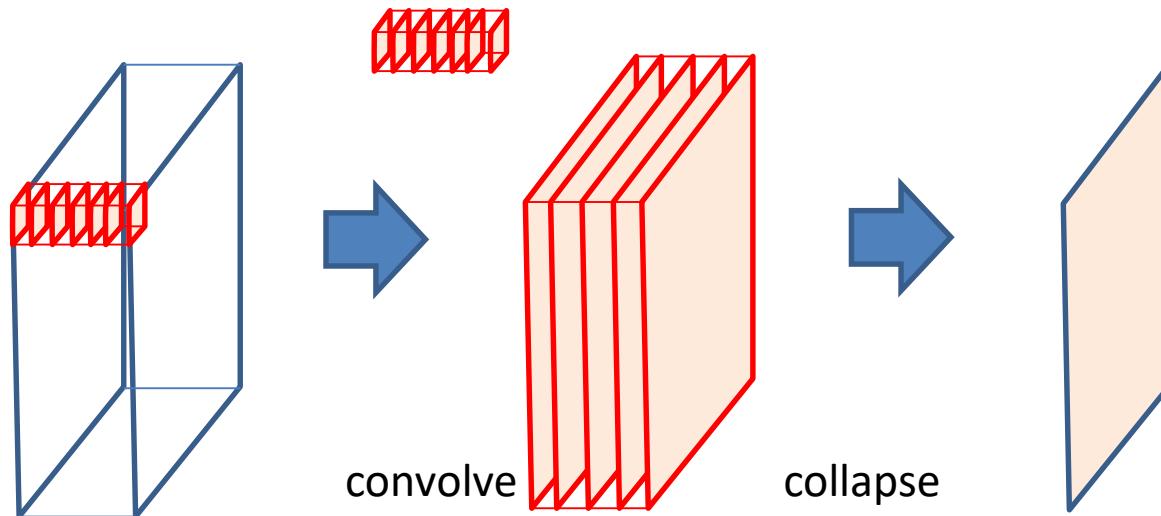
- Can use the same mechanism to predict the joints of a stick model
  - For pose estimation

# 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

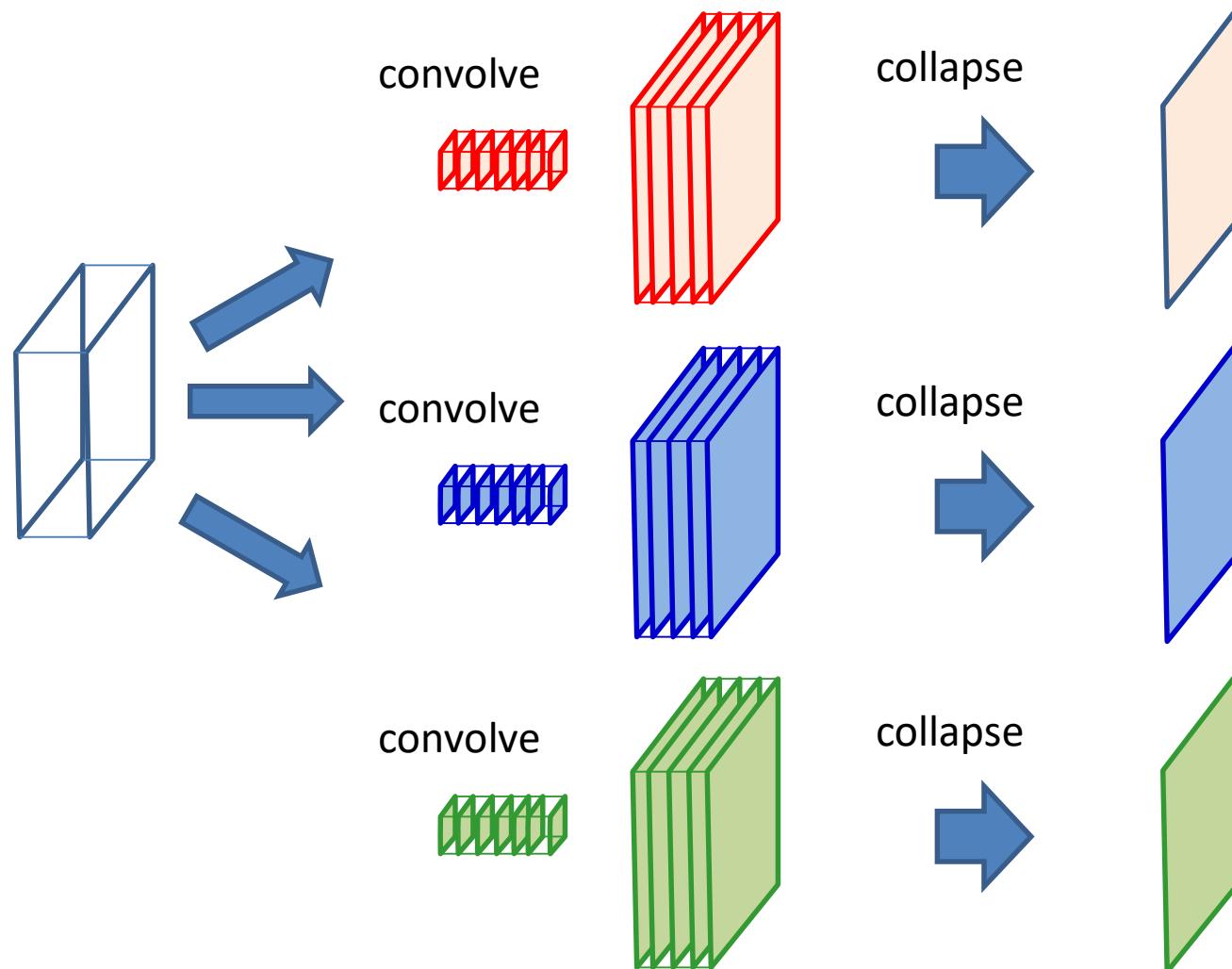
# Conventional convolutions

Conventional



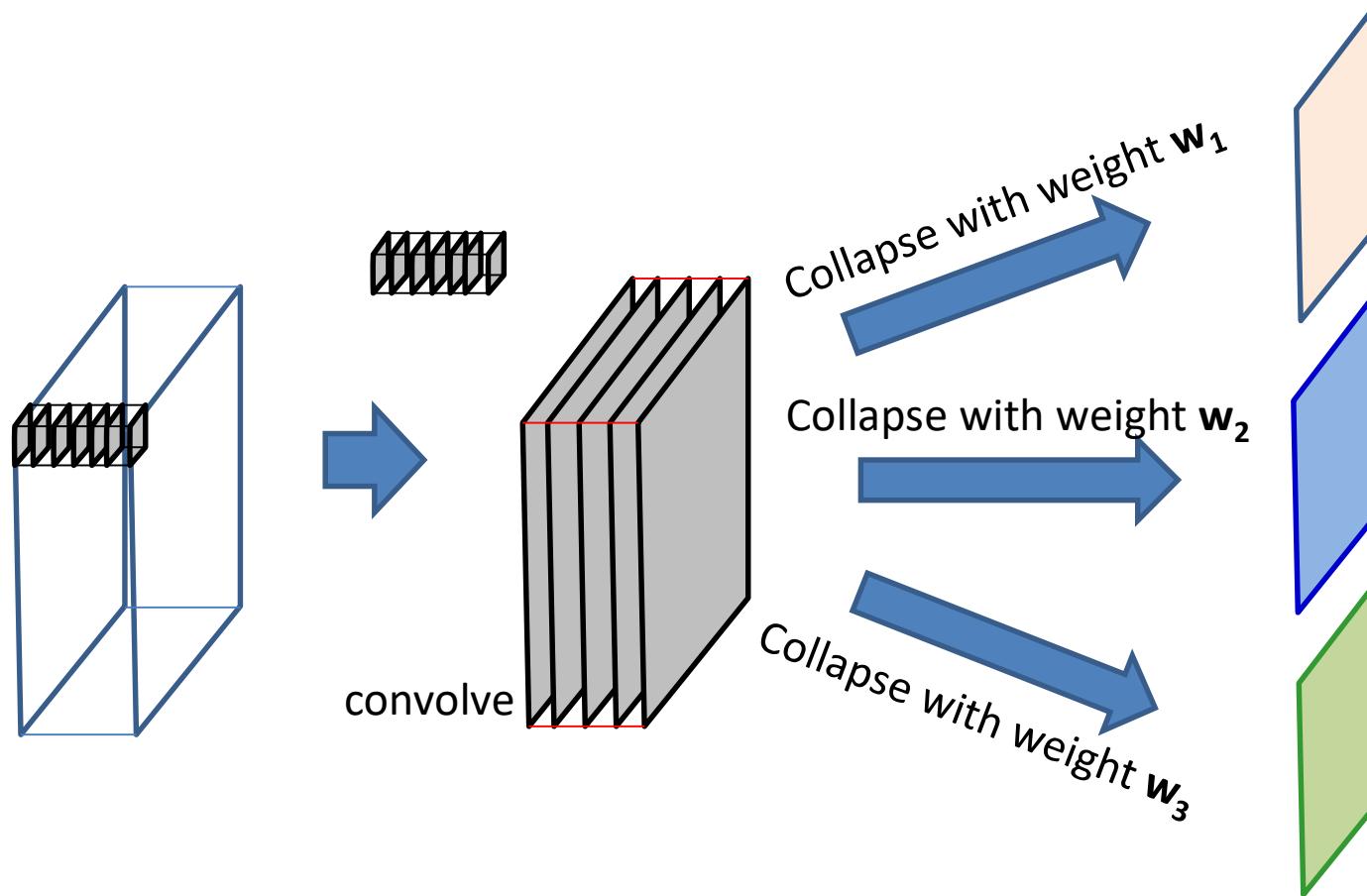
- Alternate view of conventional convolution:
- *Each layer of each filter* scans its corresponding map to produce a convolved map
- N input channels will require a filter with N layers
- The independent convolutions of each layer of the filter result in N convolved maps
- The N convolved maps are *added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

# Conventional convolutions



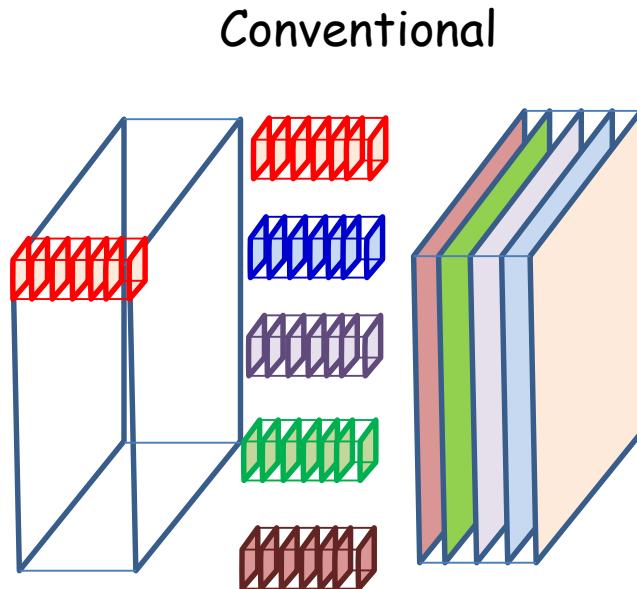
- This is done separately for each of the M filters producing M output maps (channels)

# Depth-wise convolution

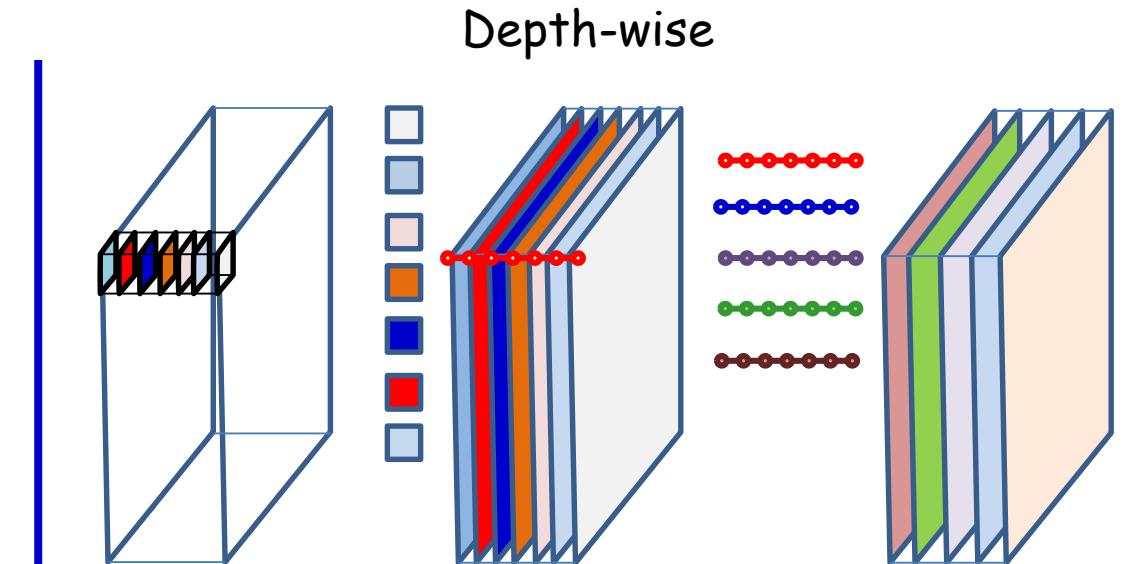


- In *depth-wise convolution* the convolution step is performed only once
- The simple summation is replaced by a *weighted* sum across channels
  - Different weights (for summation) produce different output channels

# Conventional vs. depth-wise convolution



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



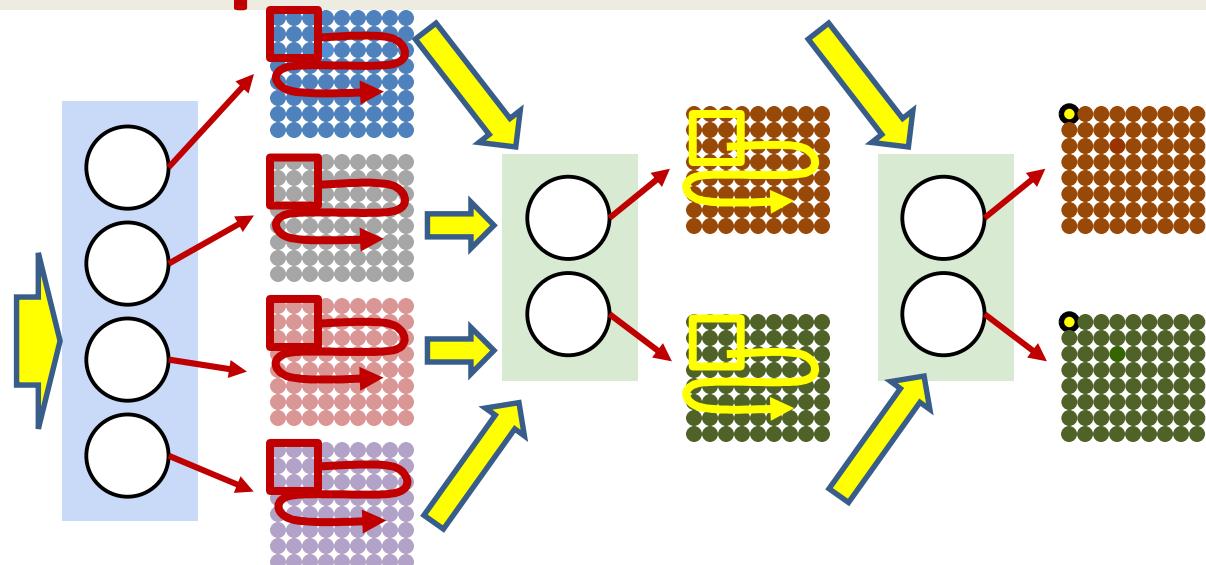
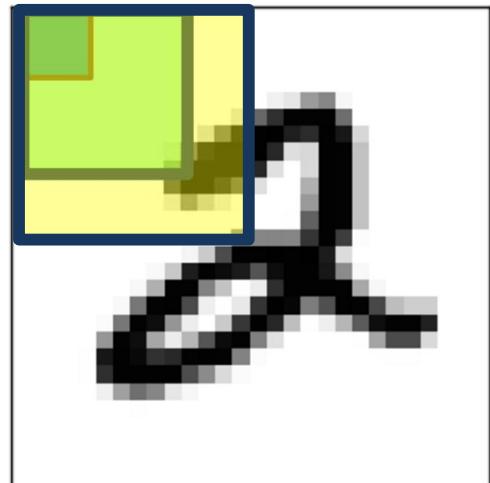
- M input channels, N output channels in 2 stages:
- Stage 1:
  - M independent  $K \times K$  **2D** filters, one per input channel
  - Each filter applies to only one input channel
  - No. of output channels = no. of input channels
- Stage 2:
  - N  $M \times 1 \times 1$  **1D** filters
  - Each applies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

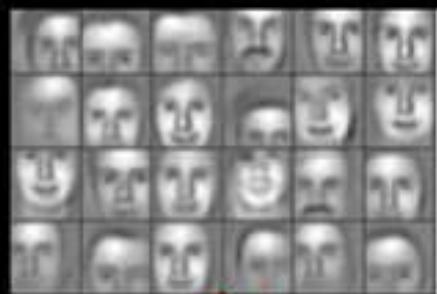
## Receptive fields



- 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.

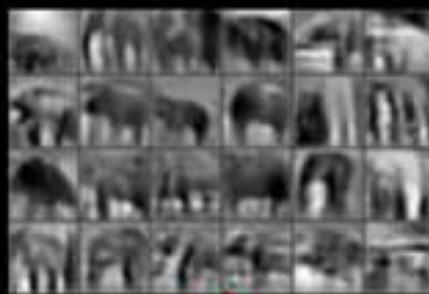
Faces



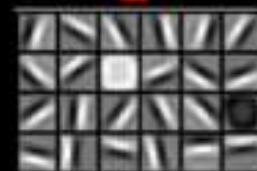
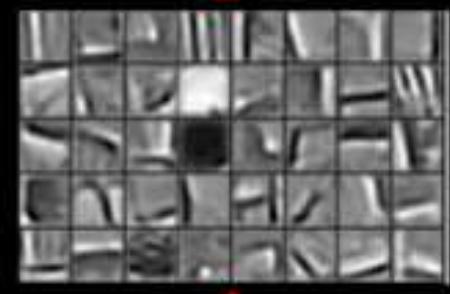
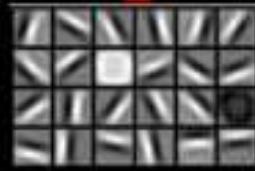
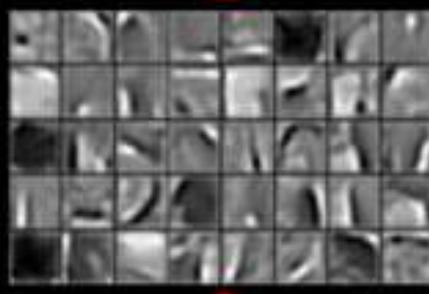
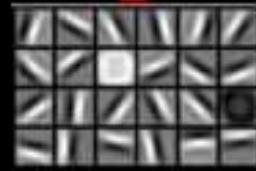
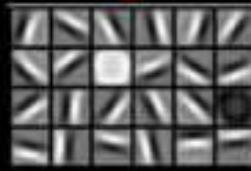
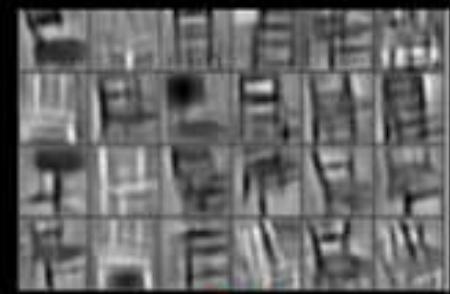
Cars



Elephants



Chairs

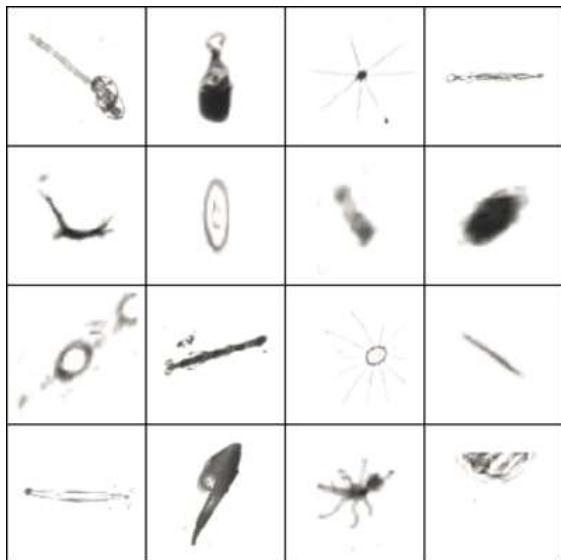


# Training Issues

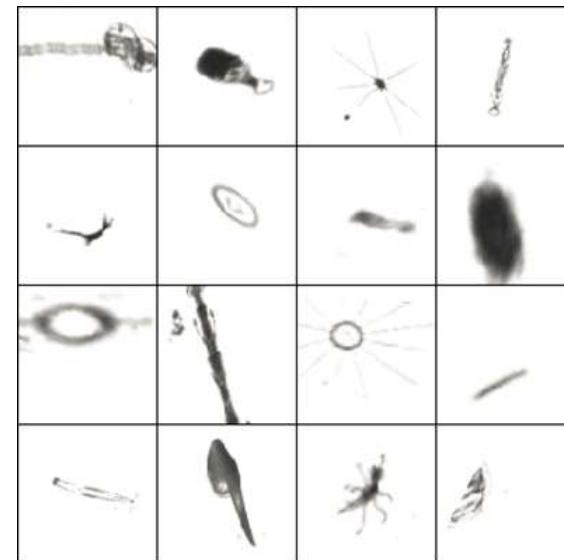
- Standard convergence issues
  - Solution: Adam or other momentum-style algorithms
  - Other tricks such as batch normalization
- The number of parameters can quickly become very large
- Insufficient training data to train well
  - Solution: Data augmentation

# Data Augmentation

Original data



Augmented data



- rotation: uniformly chosen random angle between  $0^\circ$  and  $360^\circ$
- 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  $-20^\circ$  and  $20^\circ$
- stretching: random stretching with stretch factor between 1/1.3 and 1.3 (log-uniform)

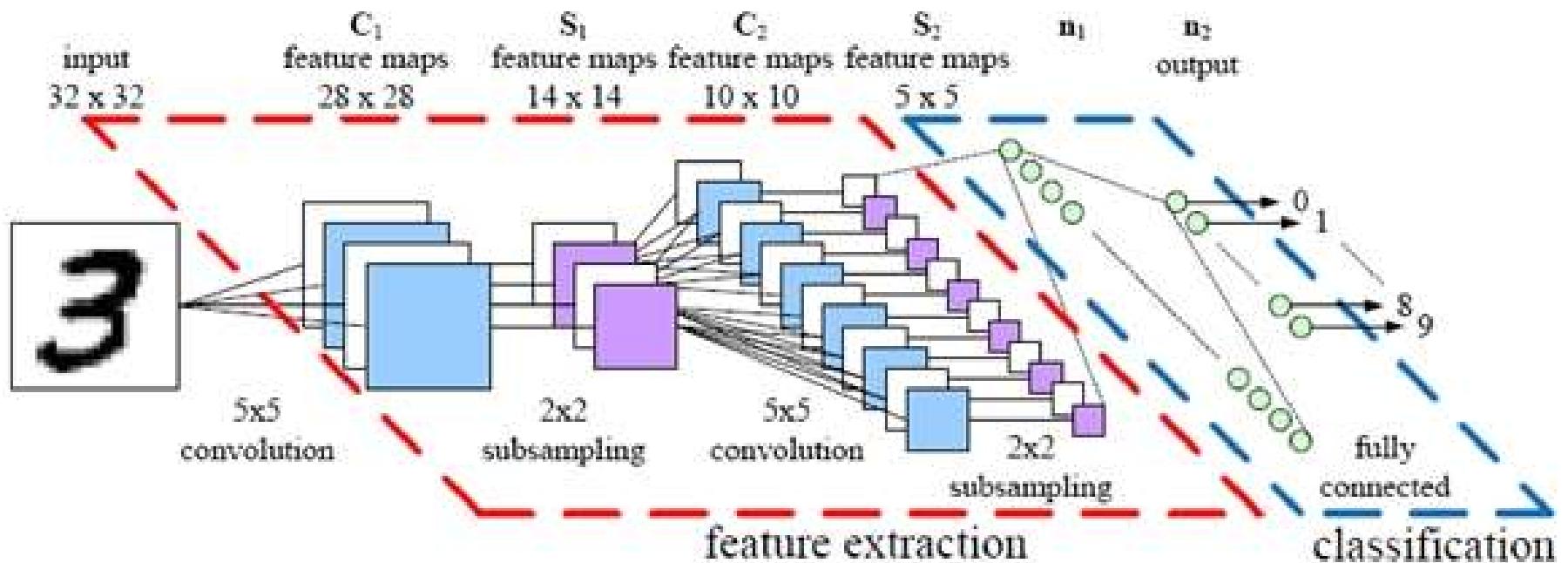
# Convolutional neural nets

- 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

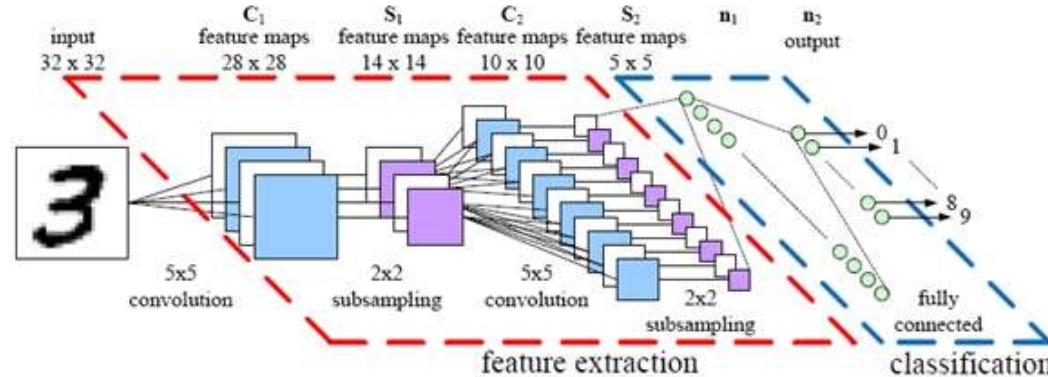
# Nice visual example

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

# Digit classification

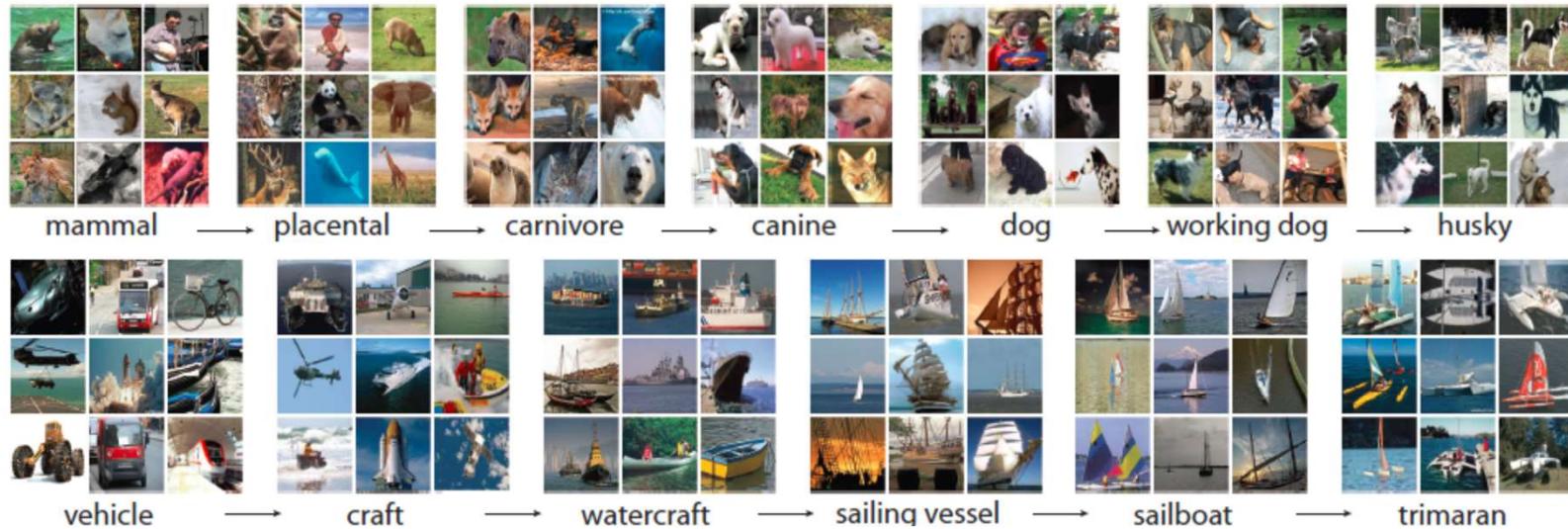


# Le-net 5



- 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

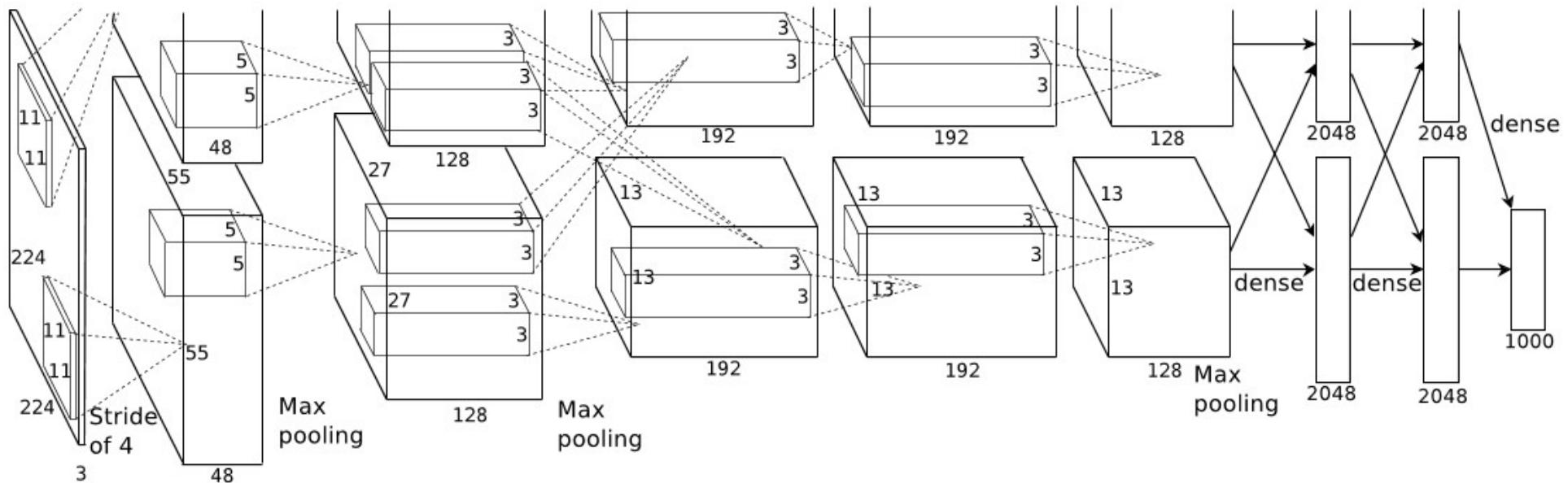
# The imangenet task



- **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



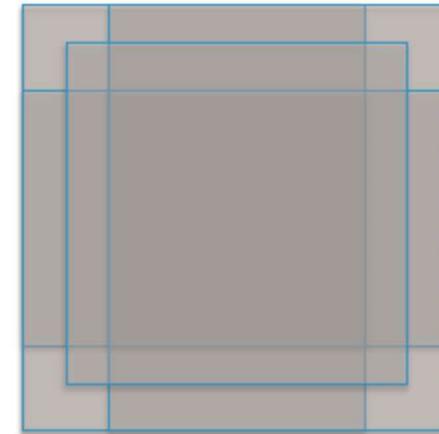
Krizhevsky, A., Sutskever, I. and Hinton, G. E. "ImageNet Classification with Deep Convolutional Neural Networks" NIPS 2012: Neural Information Processing Systems, Lake Tahoe, Nevada

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



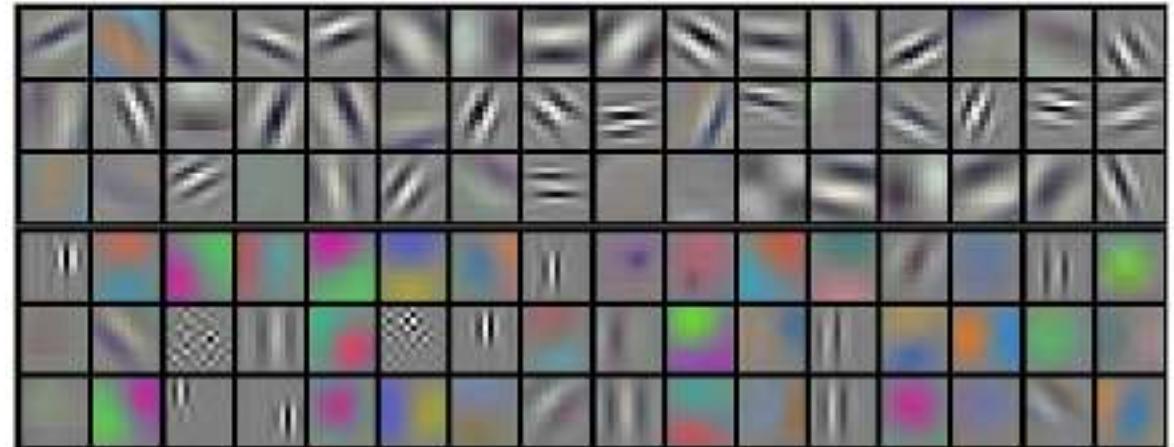
10 patches

# 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  $11 \times 11 \times 3$  learned by the first convolutional layer on the  $224 \times 224 \times 3$  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!



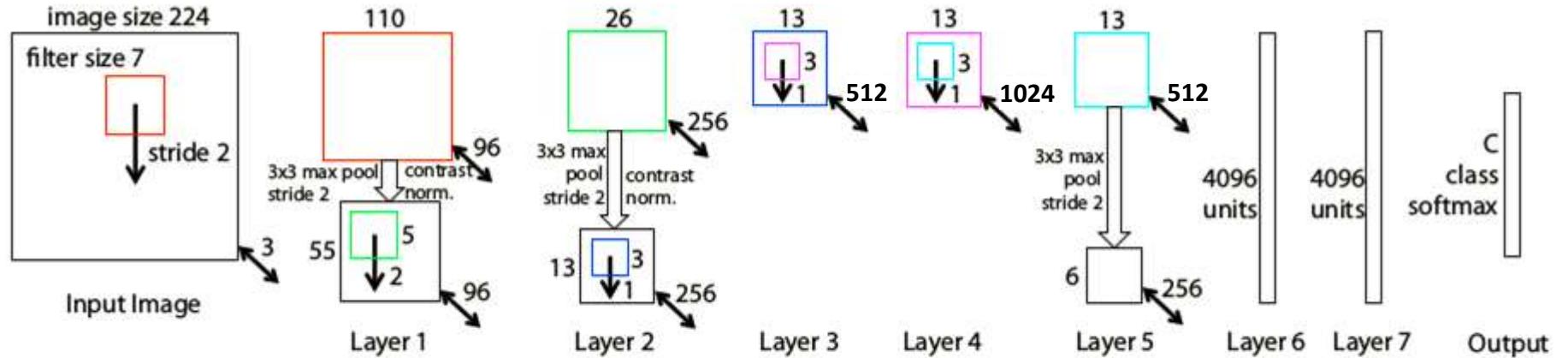
Eight ILSVRC-2010 test images and the five labels considered most probable by our model. The correct label is written under each image, and the probability assigned to the correct label is also shown with a red bar (if it happens to be in the top 5).

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



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.

# 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

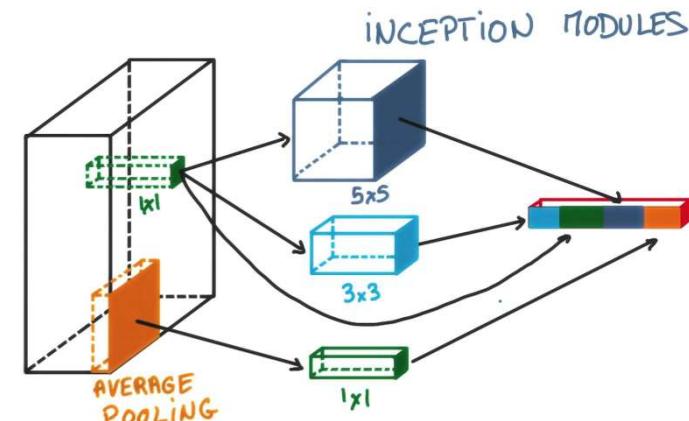
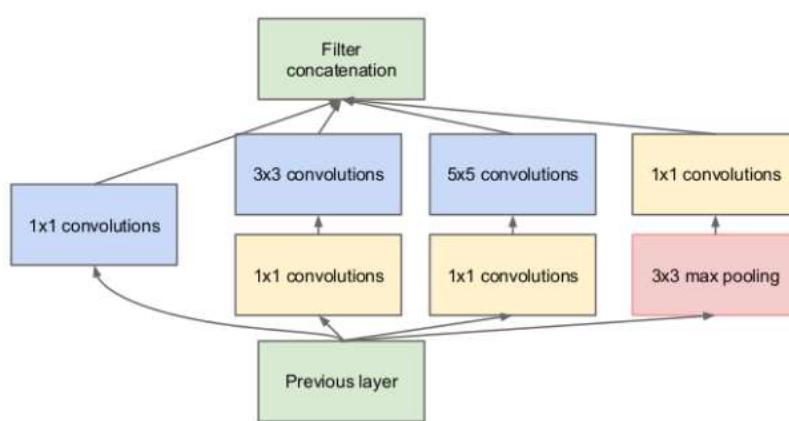
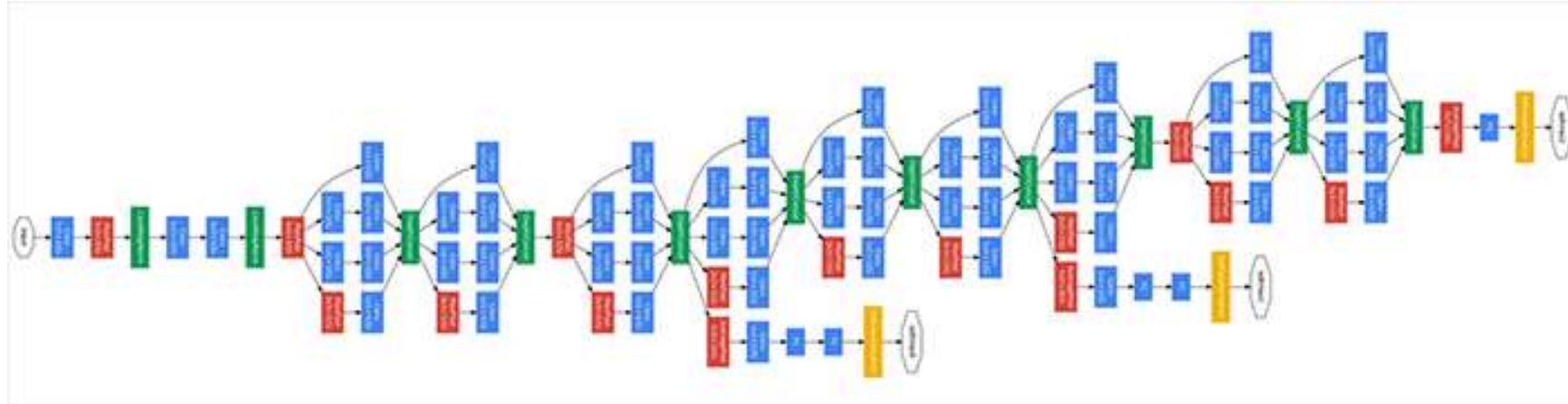
- Simonyan and Zisserman, 2014
- Only used 3x3 filters, stride 1, pad 1
- Only used 2x2 pooling filters, stride 2
- Tried a large number of architectures.
- Finally obtained **7.3% top-5 error** using 13 conv layers and 3 FC layers
  - Combining 7 classifiers
  - Subsequent to paper, reduced error to 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-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

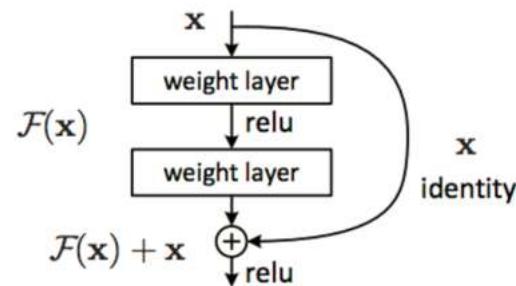
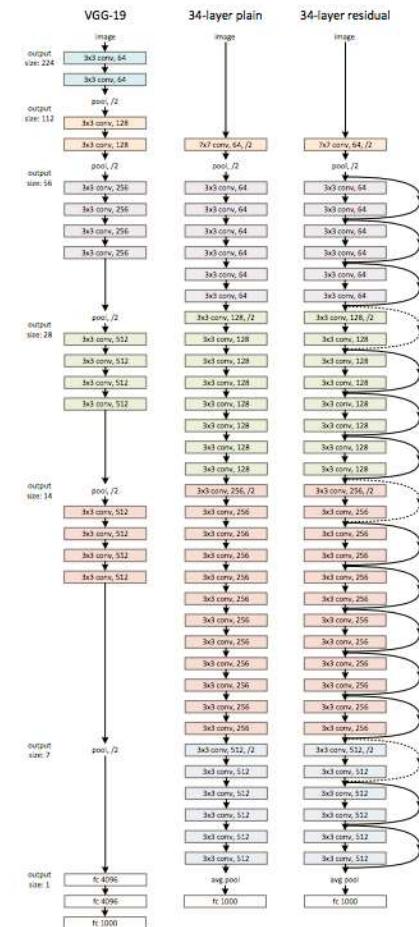


Figure 2. Residual learning: a building block.



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

# Resnet details for the curious..

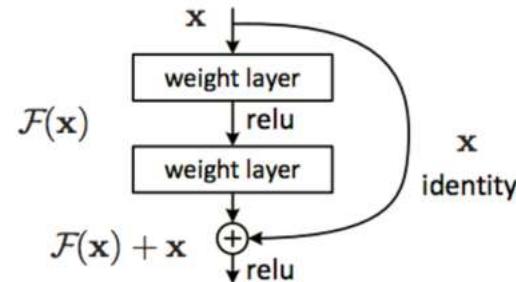
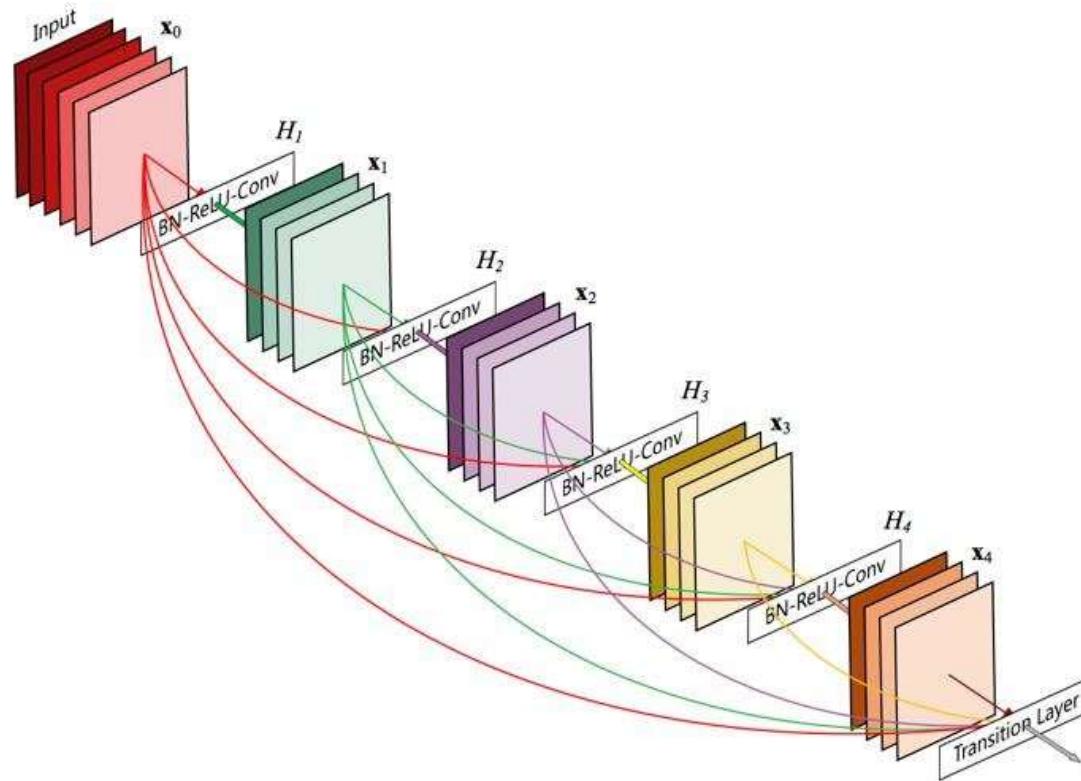


Figure 2. Residual learning: a building block.

- 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



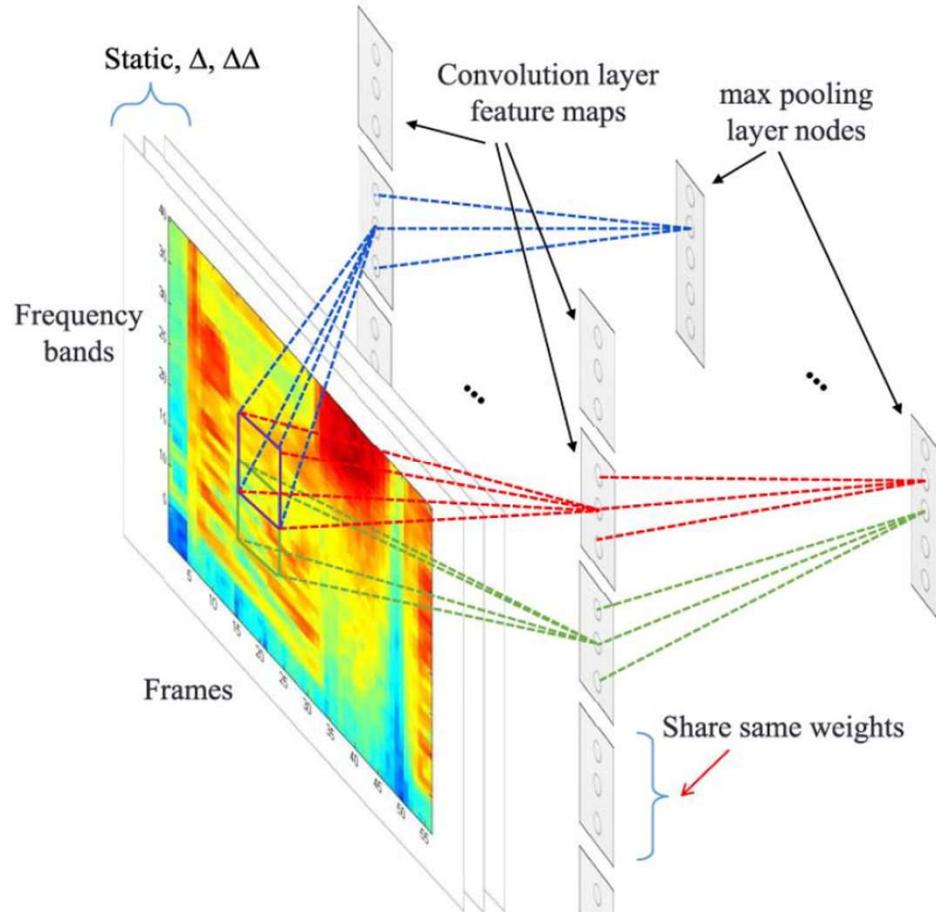
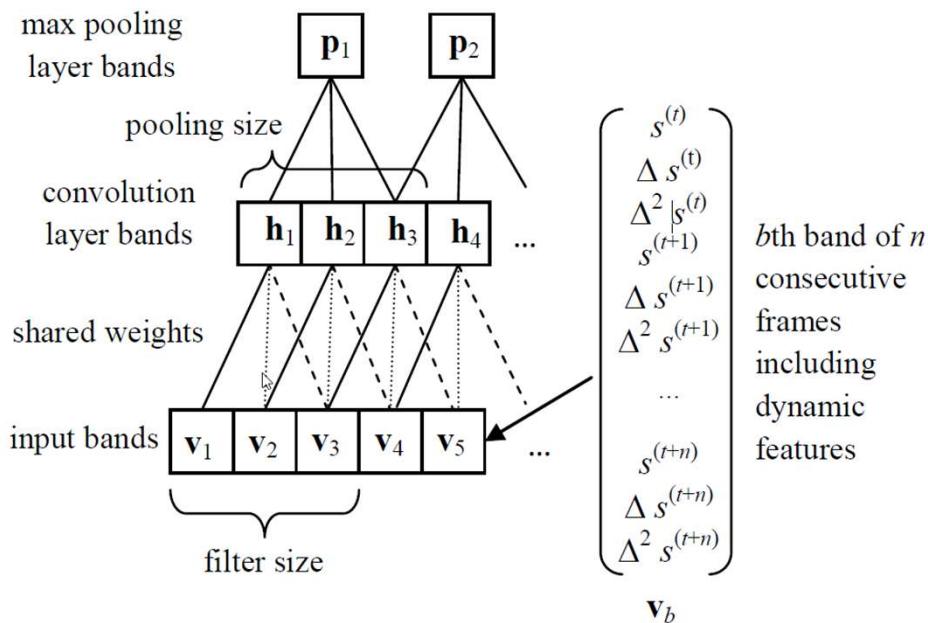
- All convolutional
- Each layer looks at the union of maps from all previous layers
  - Instead of just the set of maps from the immediately previous layer
- Was state of the art before I went for coffee one day
  - Wasn't when I got back..

# 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%
<b>CNN-DNN; P=1:m (HP, m=12)</b>	<b>19.3%</b>
<b>CNN-DNN; above (add dropout)</b>	<b>18.7%</b>

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

