Deep Neural Networks
Convolutional Networks III

Bhiksha Raj

Outline

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

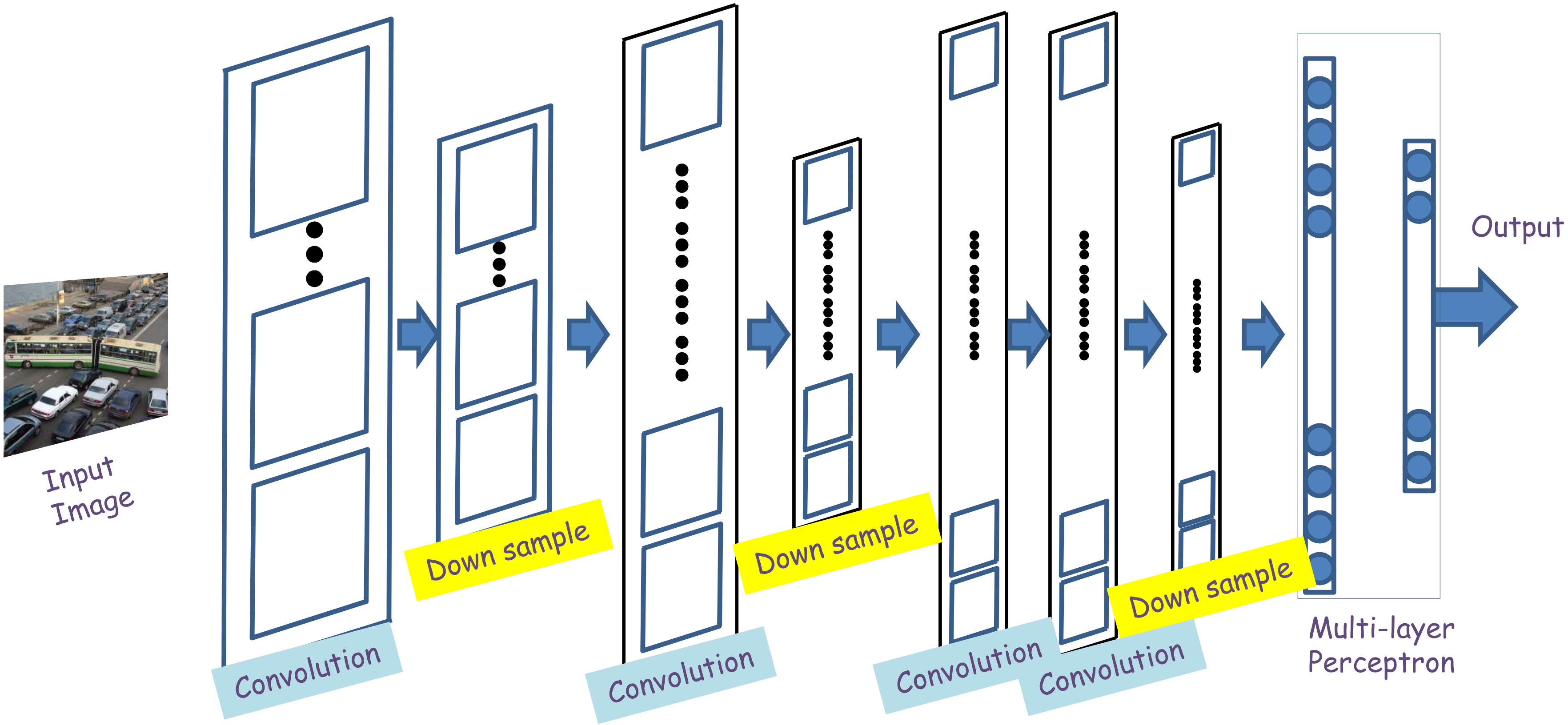
Story so far

- 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”

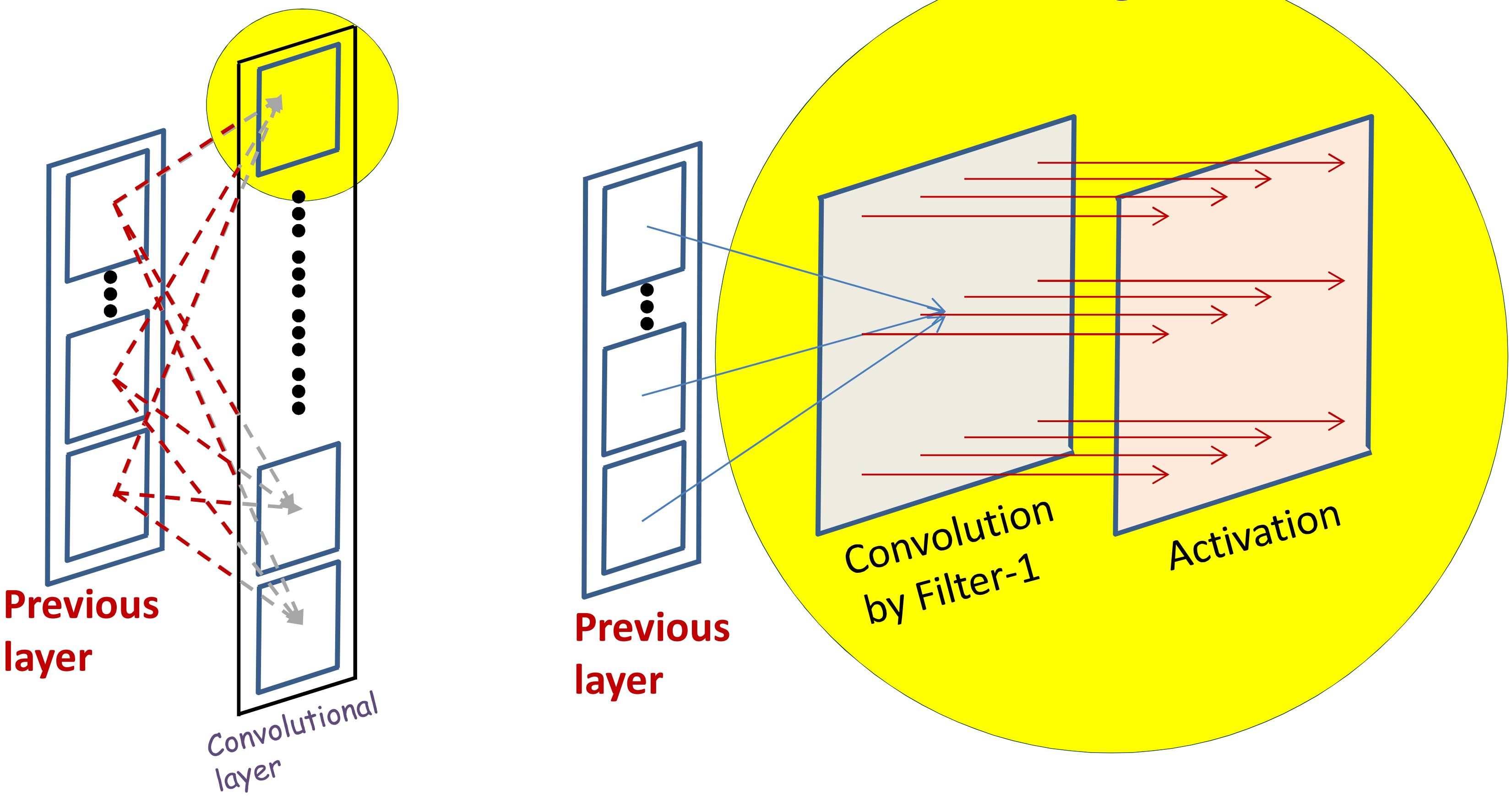
- For 2-D (or higher-dimensional) scans, the structure is called a convnet
- For 1-D scan along time, it is called a Time-delay neural network

The general architecture of a convolutional neural network



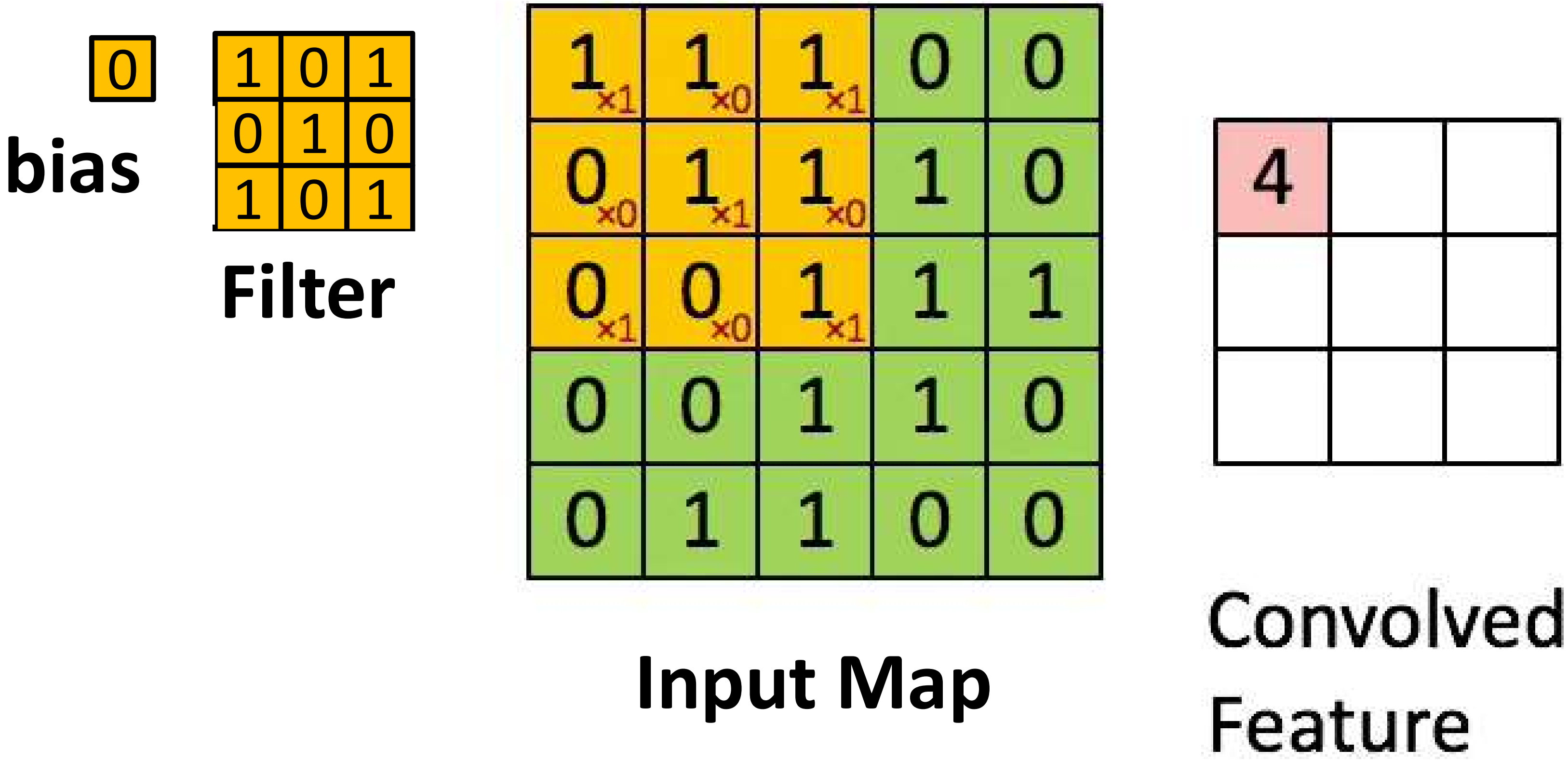
- A convolutional neural network comprises of "convolutional" and optional "downsampling" layers
- Followed by an MLP with one or more layers

A convolutional layer



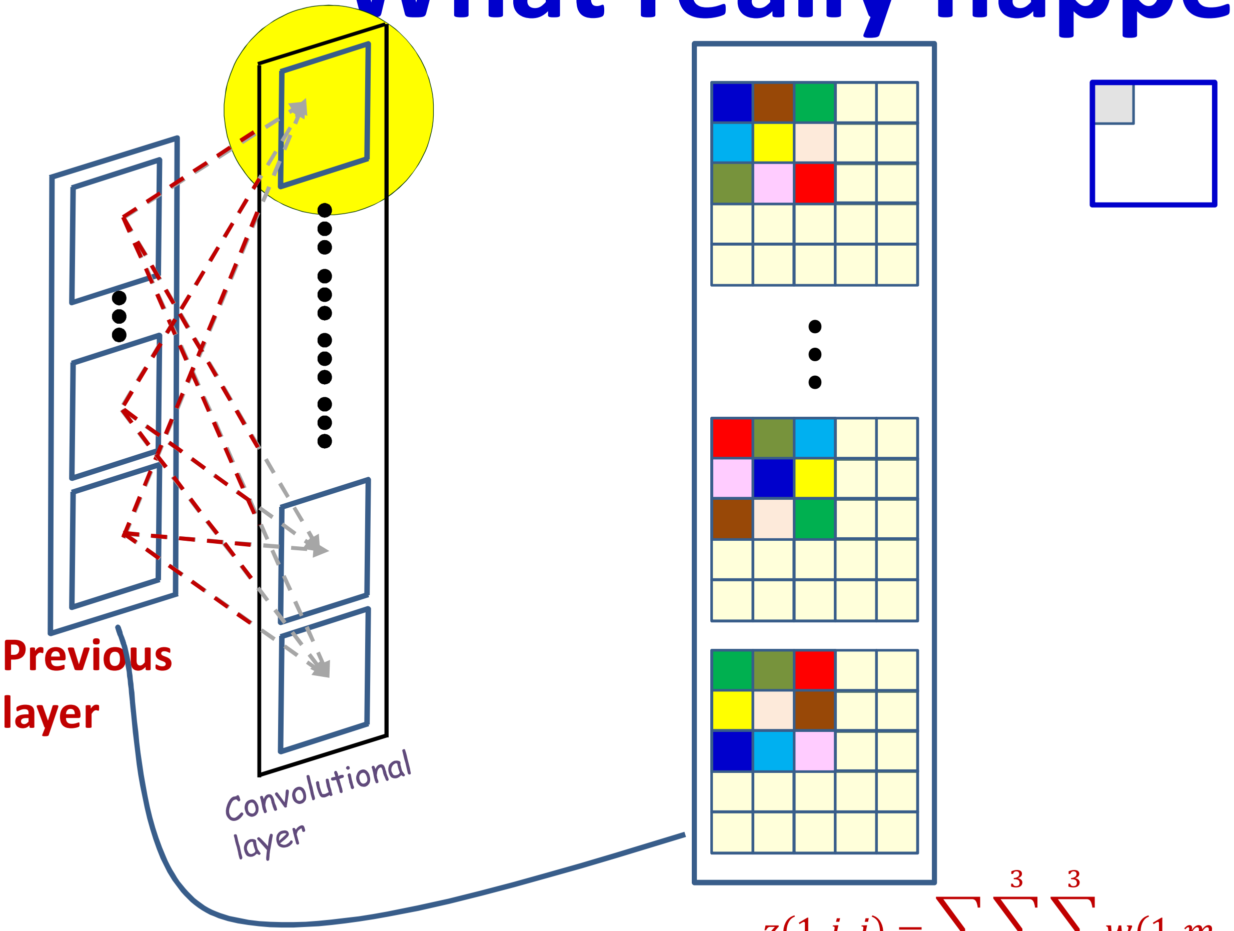
- Each activation map in the convolutional layer has two components
  - A *linear* map, obtained by ***convolution*** over maps in the previous layer
    - Each linear map has, associated with it, a ***learnable filter***
  - An ***activation*** that operates on the output of the convolution

What is a convolution



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

What really happens

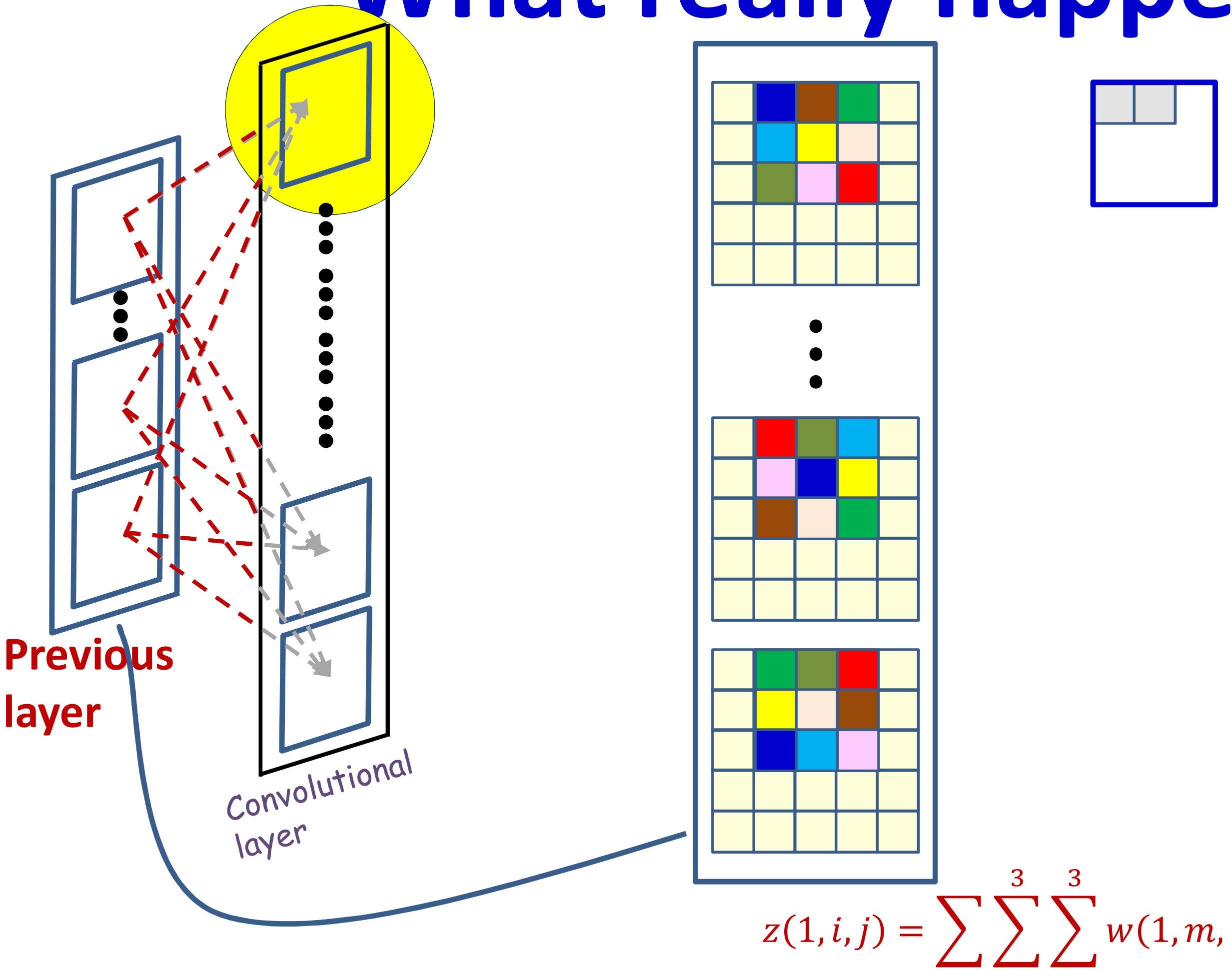


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as

  size of the filter x *no. of maps in previous layer*

What really happens

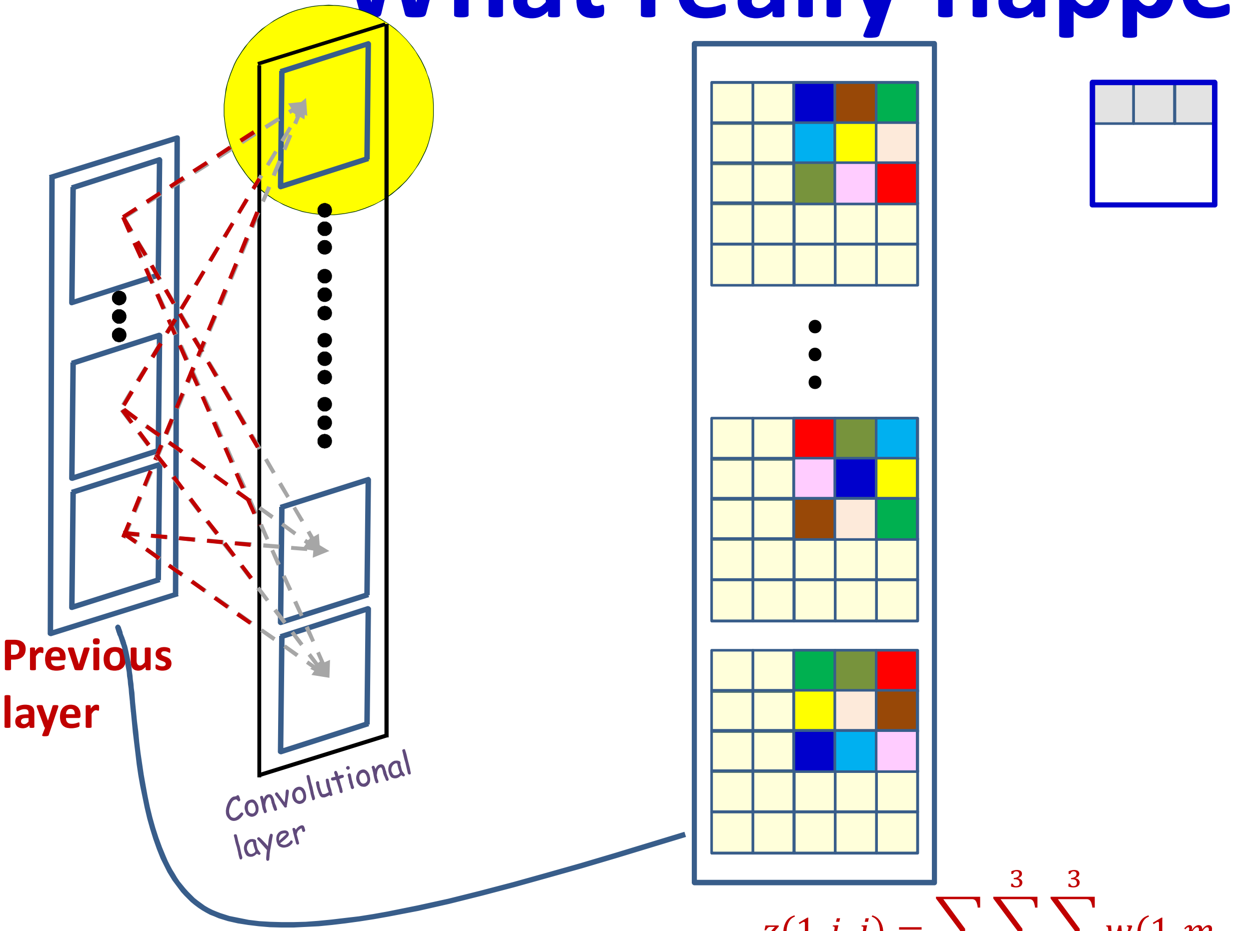


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as

  size of the filter x *no. of maps in previous layer*

What really happens

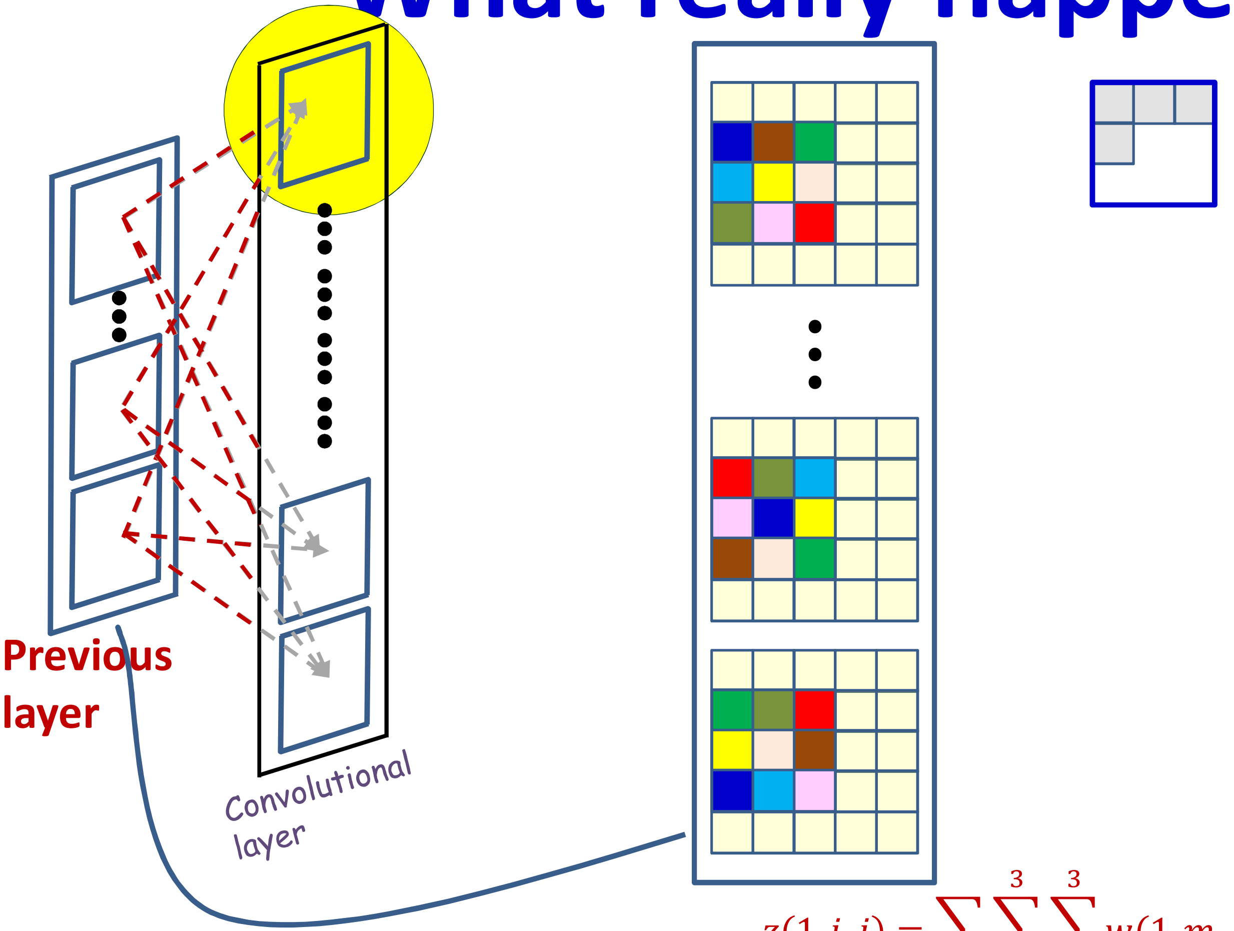


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as

  size of the filter x *no. of maps in previous layer*

What really happens

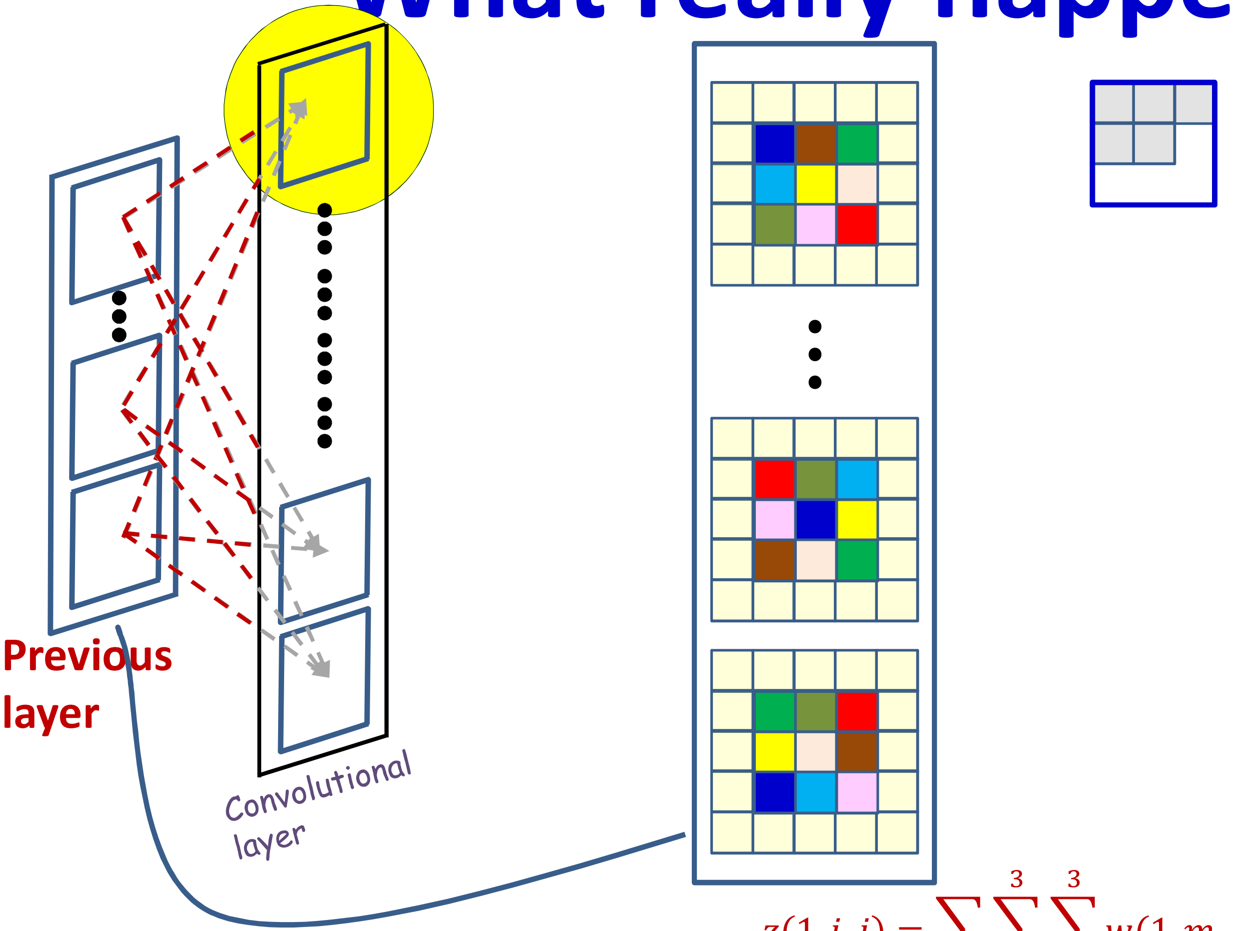


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as

  size of the filter x *no. of maps in previous layer*

What really happens

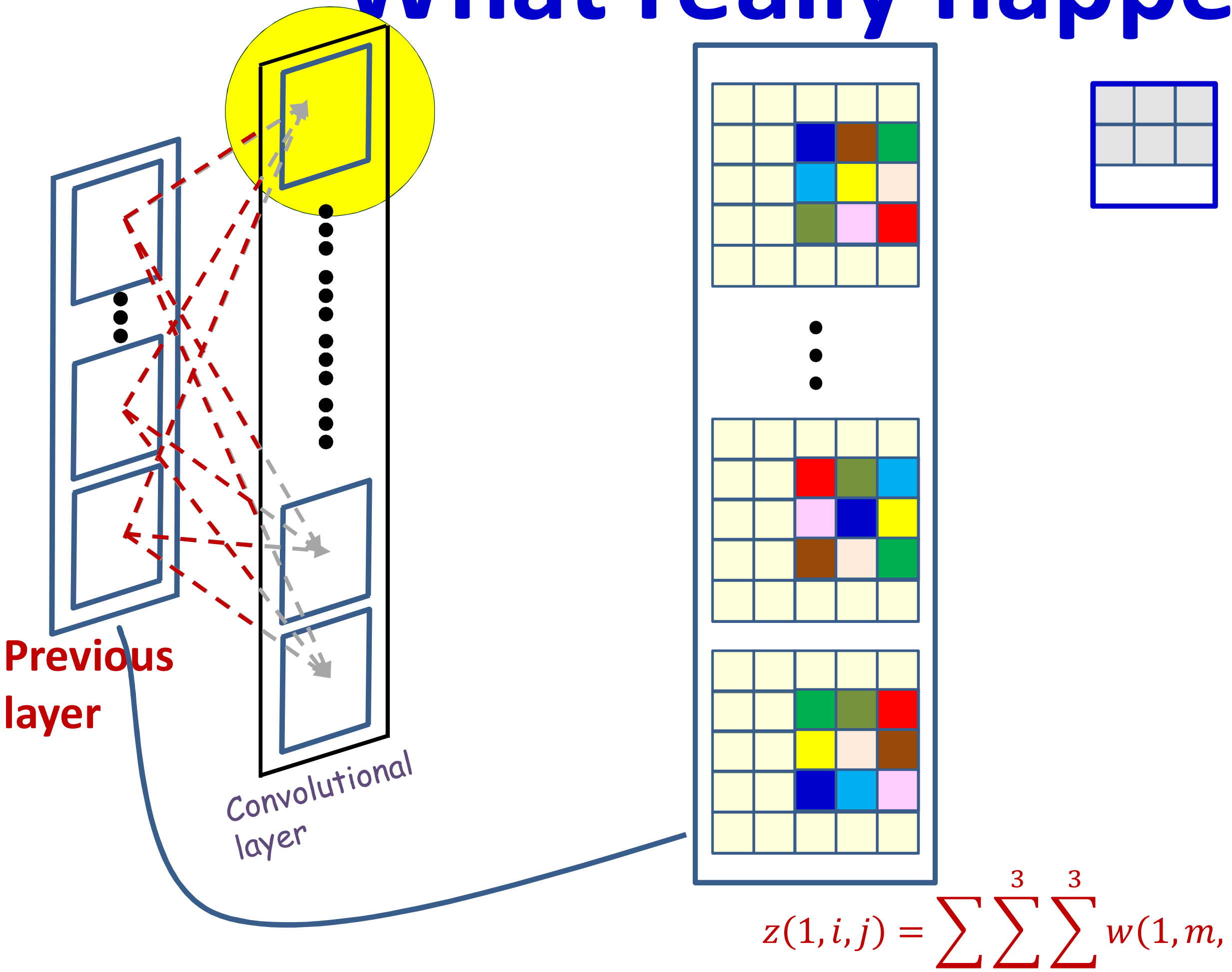


$$z(1,i,j) = \sum_{m}\sum_{k=1}^{3}\sum_{l=1}^{3} w(1,m,k,l)I(m,i+l-1,j+k-1) + b$$

- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as

  size of the filter x *no. of maps in previous layer*

What really happens

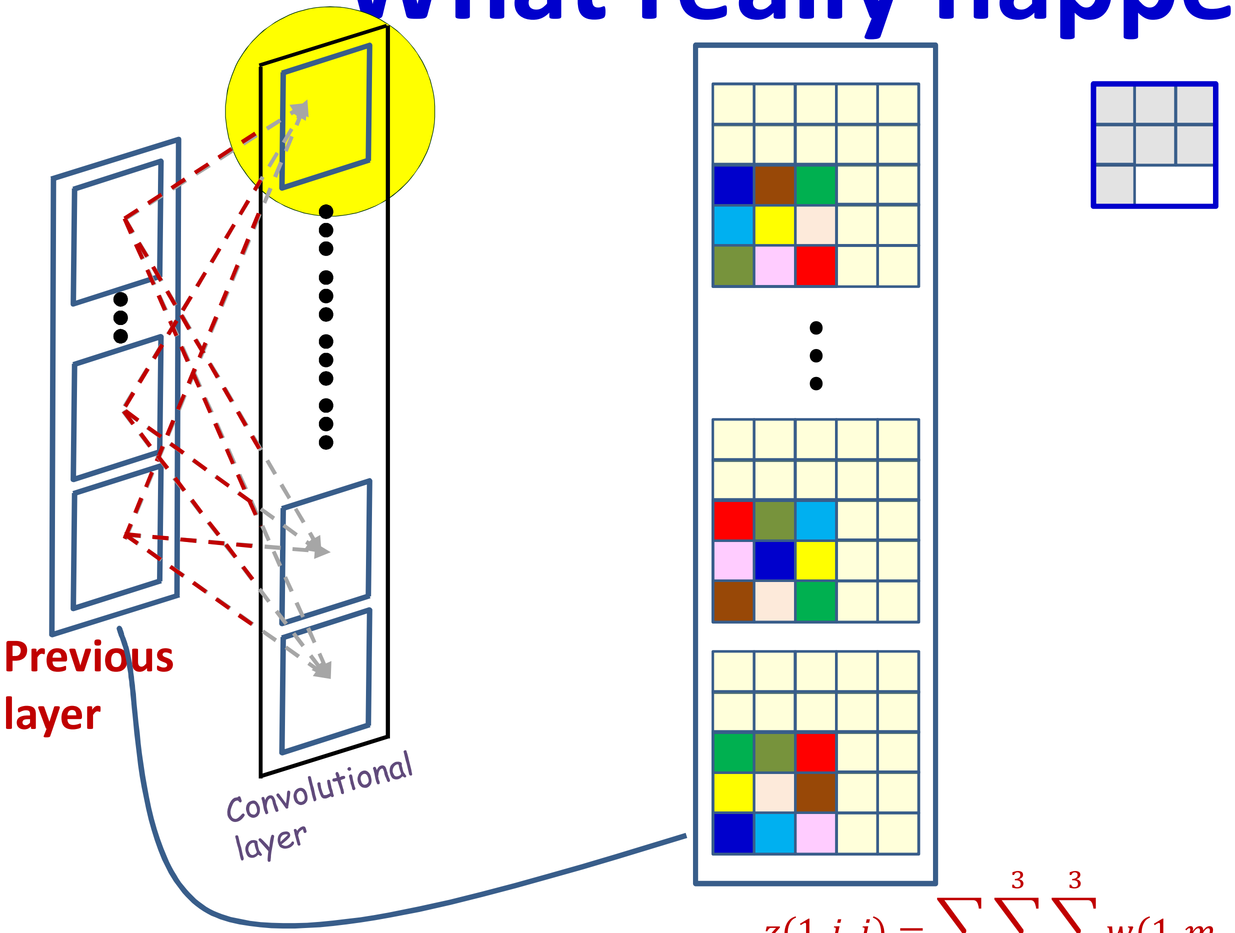


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as

  size of the filter x *no. of maps in previous layer*

What really happens

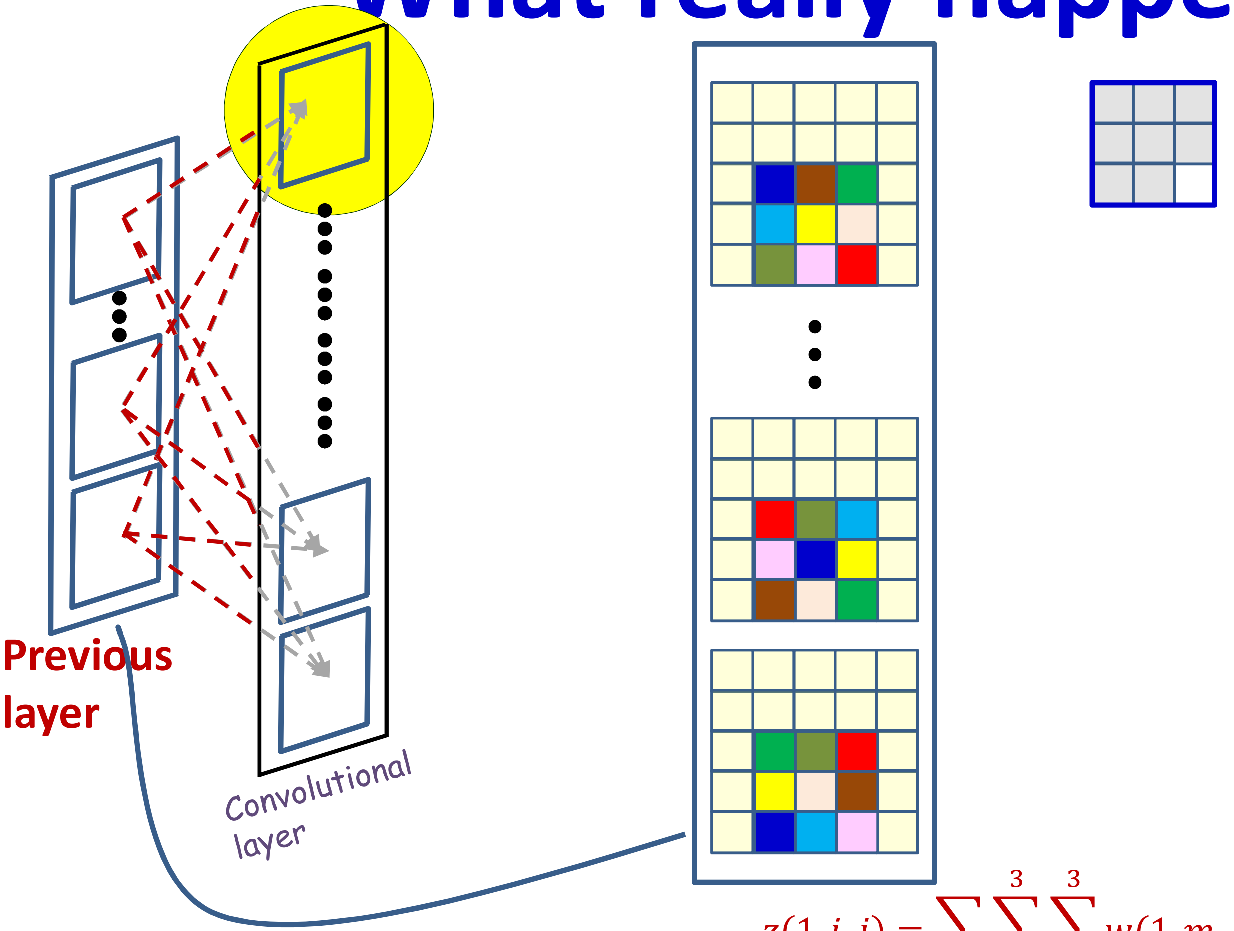


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as

  size of the filter x *no. of maps in previous layer*

What really happens

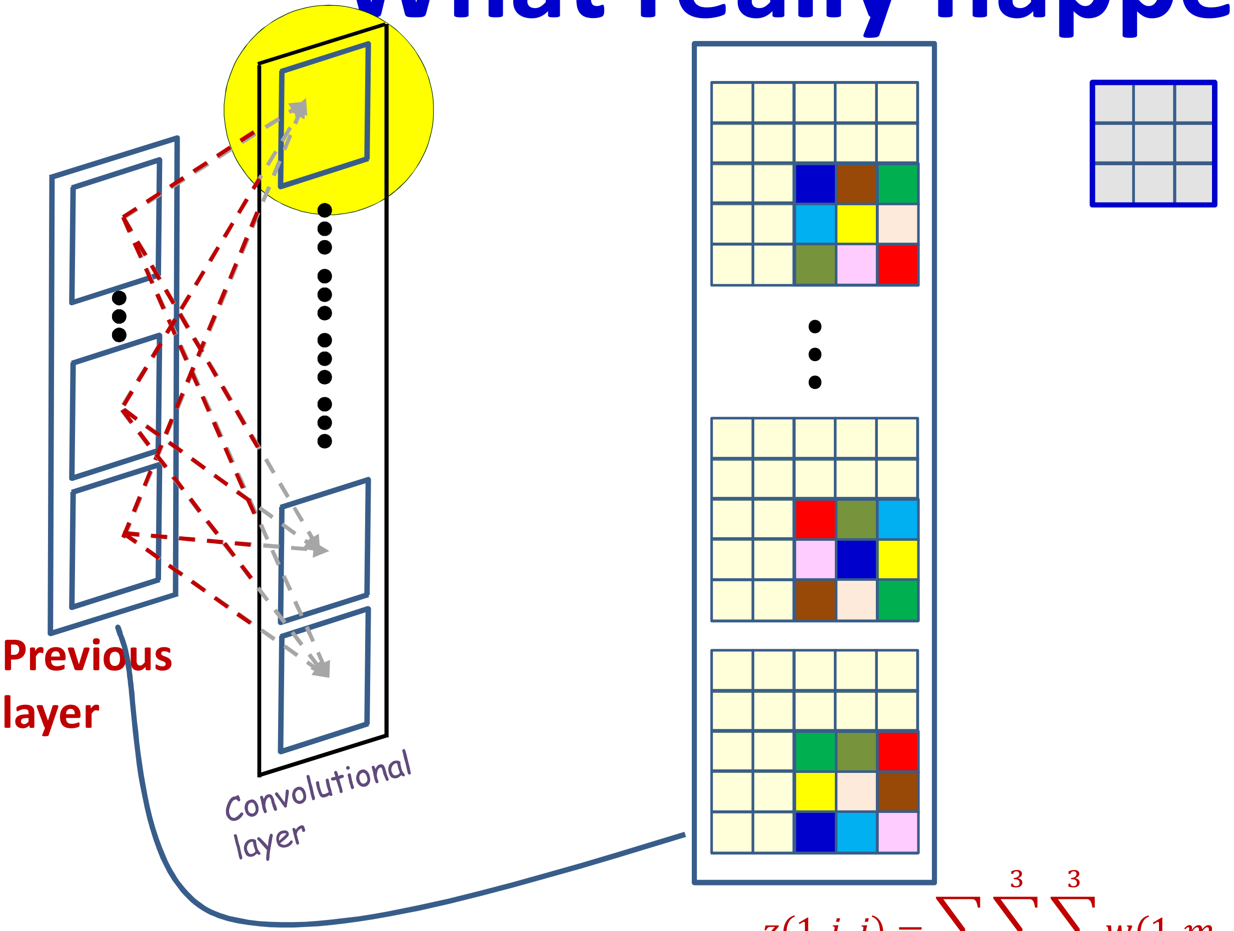


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as

  size of the filter x *no. of maps in previous layer*

What really happens

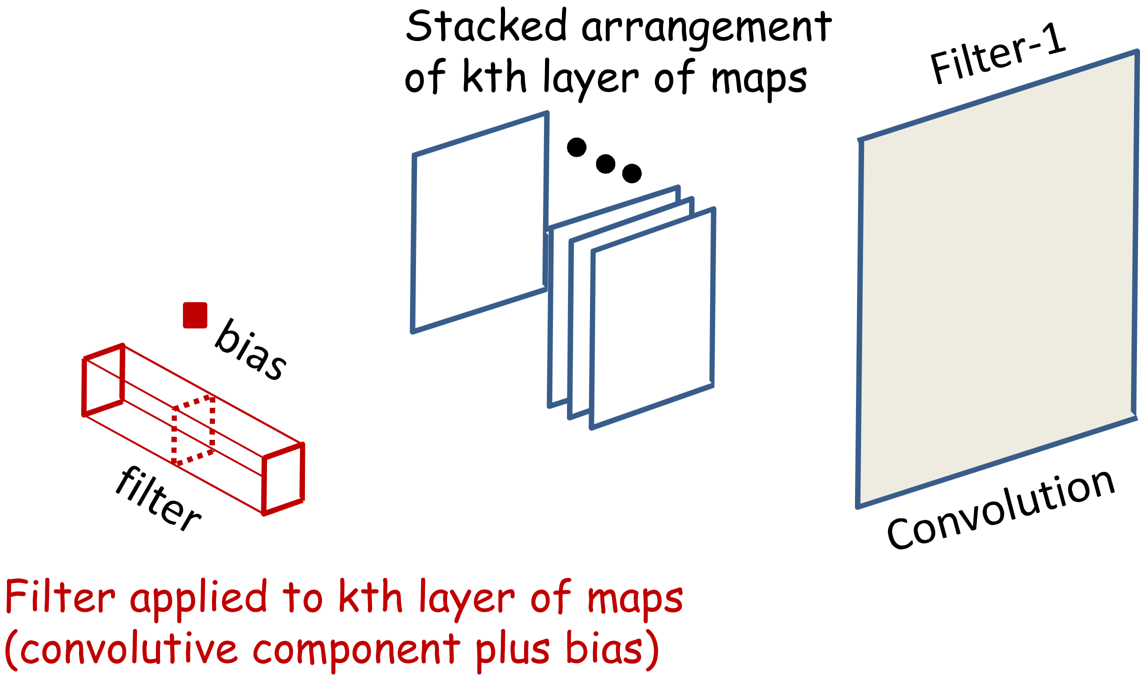


$$z(1,i,j) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(1,m,k,l) I(m, i+l-1, j+k-1) + b$$

- Each output is computed from multiple maps simultaneously
- There are as many weights (for each output map) as

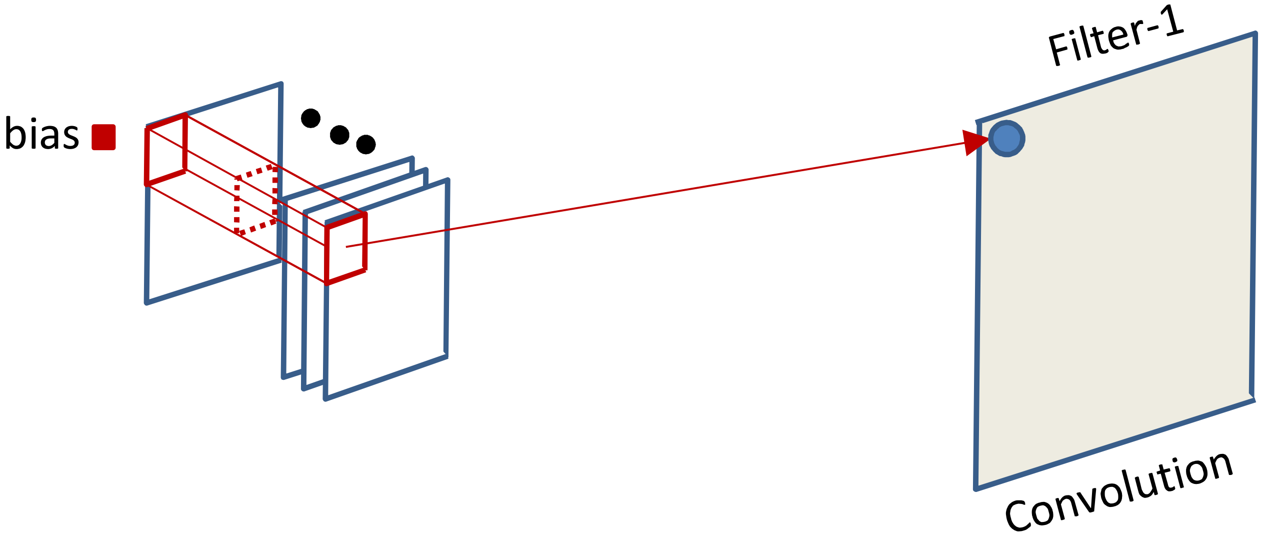
  size of the filter x *no. of maps in previous layer*

A better representation


Stacked arrangement
of kth layer of maps
Filter-1
bias
filter
Convolution
Filter applied to kth layer of maps
(convolutive component plus bias)

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

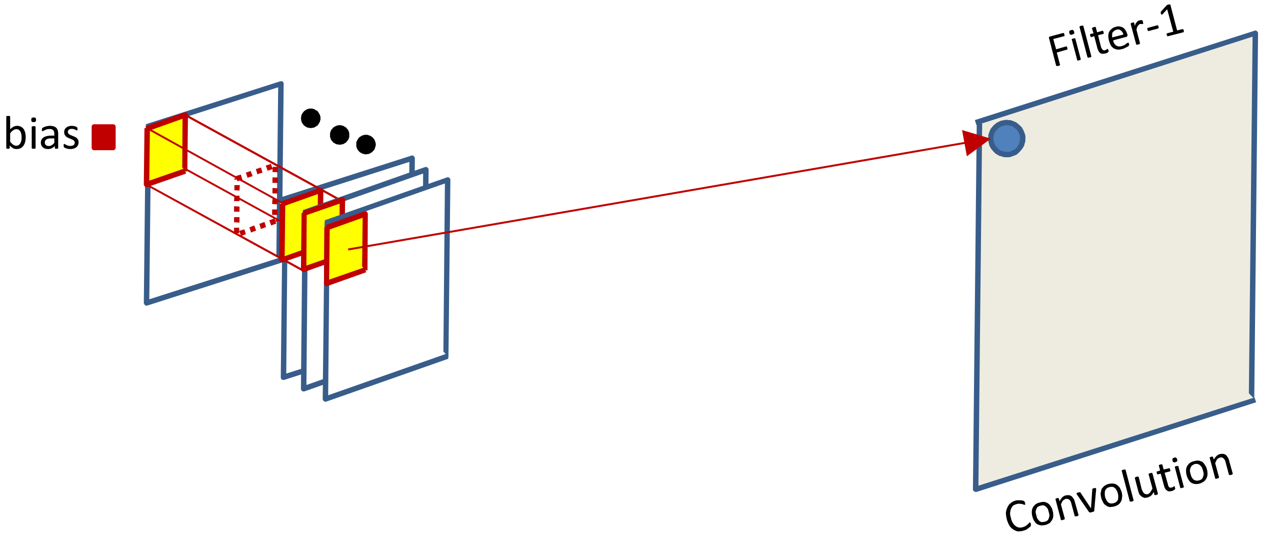
A better representation



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

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

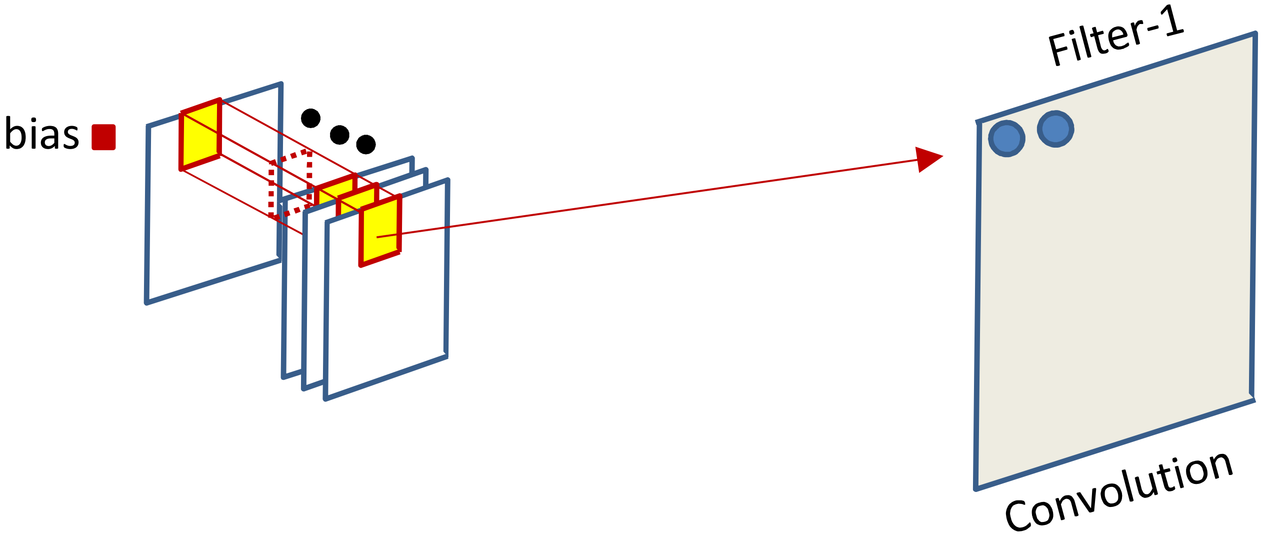
A better representation



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

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

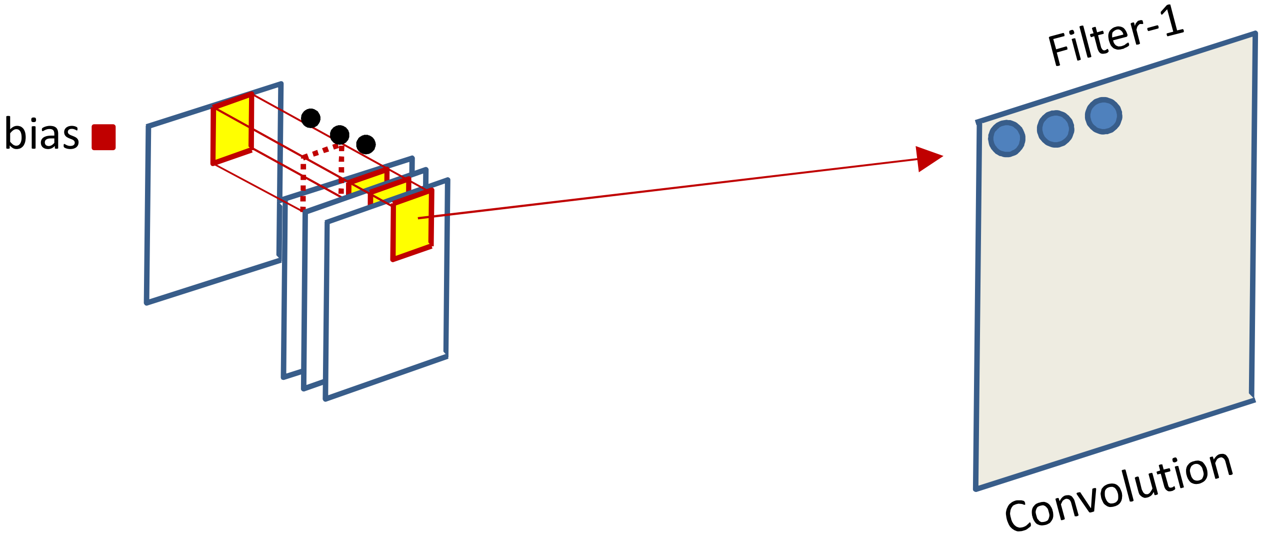
A better representation



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

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

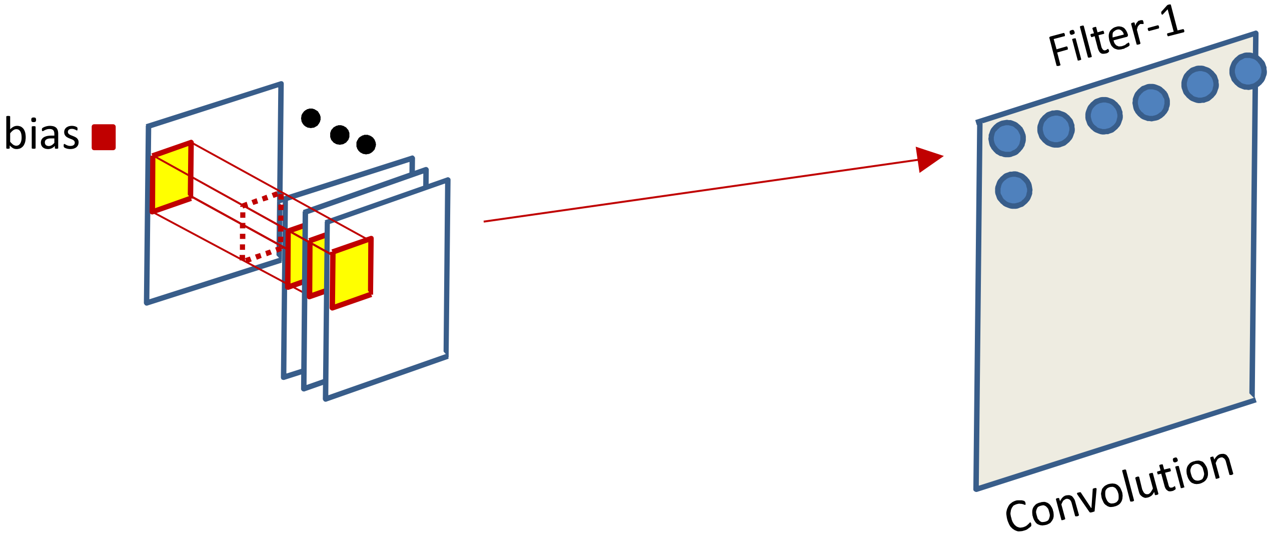
A better representation



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

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

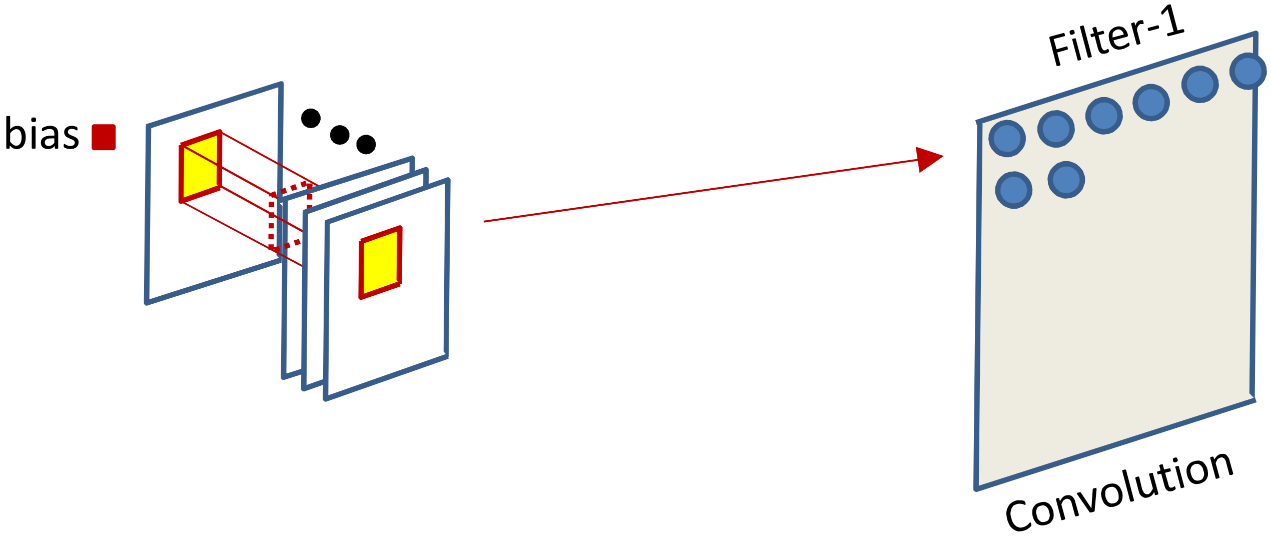
A better representation



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

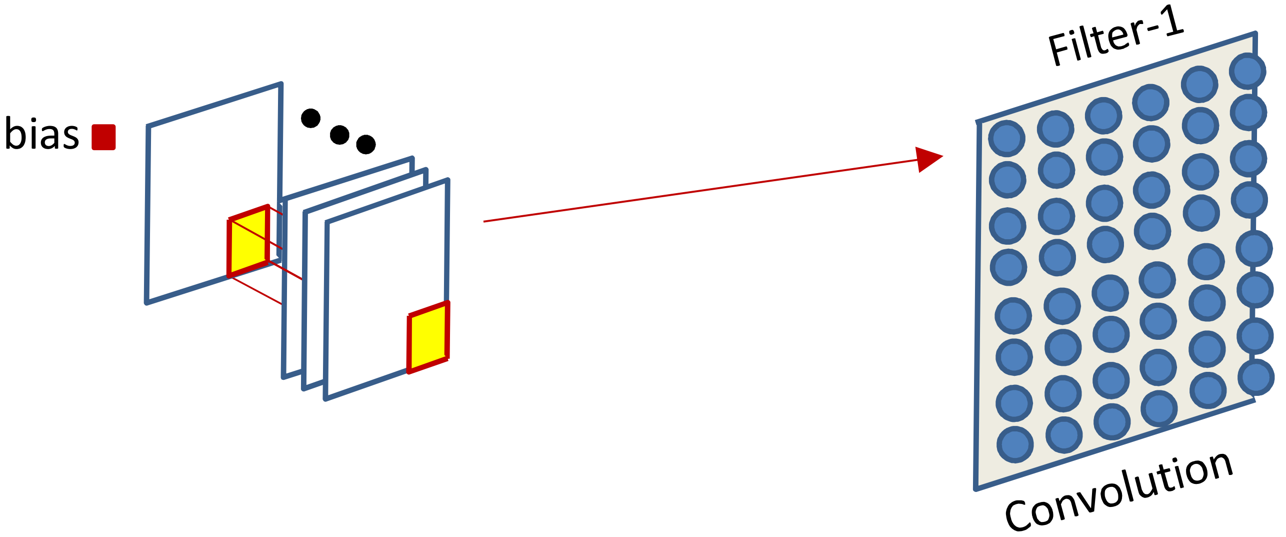
- The computation of the convolutive map at any location *sums* the convolutive outputs *at all planes*

A better representation



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

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



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

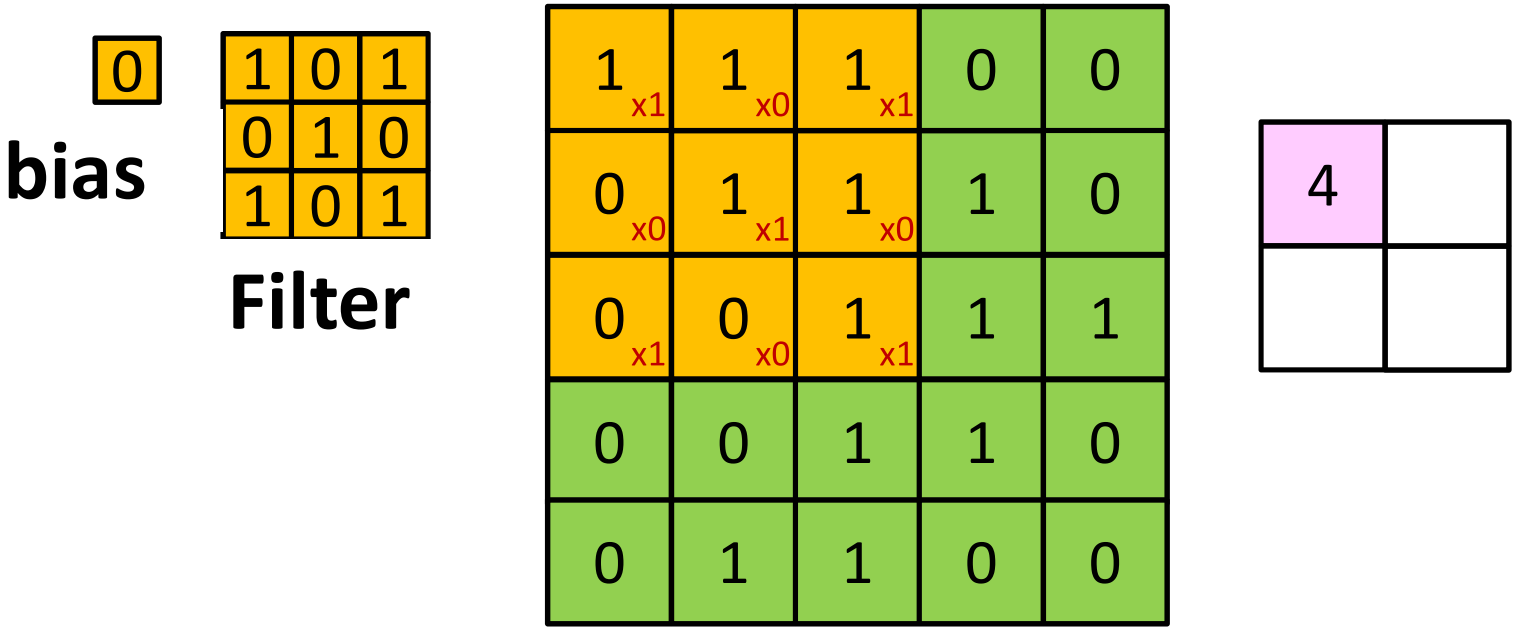
- The computation of the convolutive map at any location *sums* the convolutive outputs *at all planes*

Convolutional neural net: Vector notation

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

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

Convolution can *shrink* a map by using strides greater than 1



0

bias

1	0	1
0	1	0
1	0	1

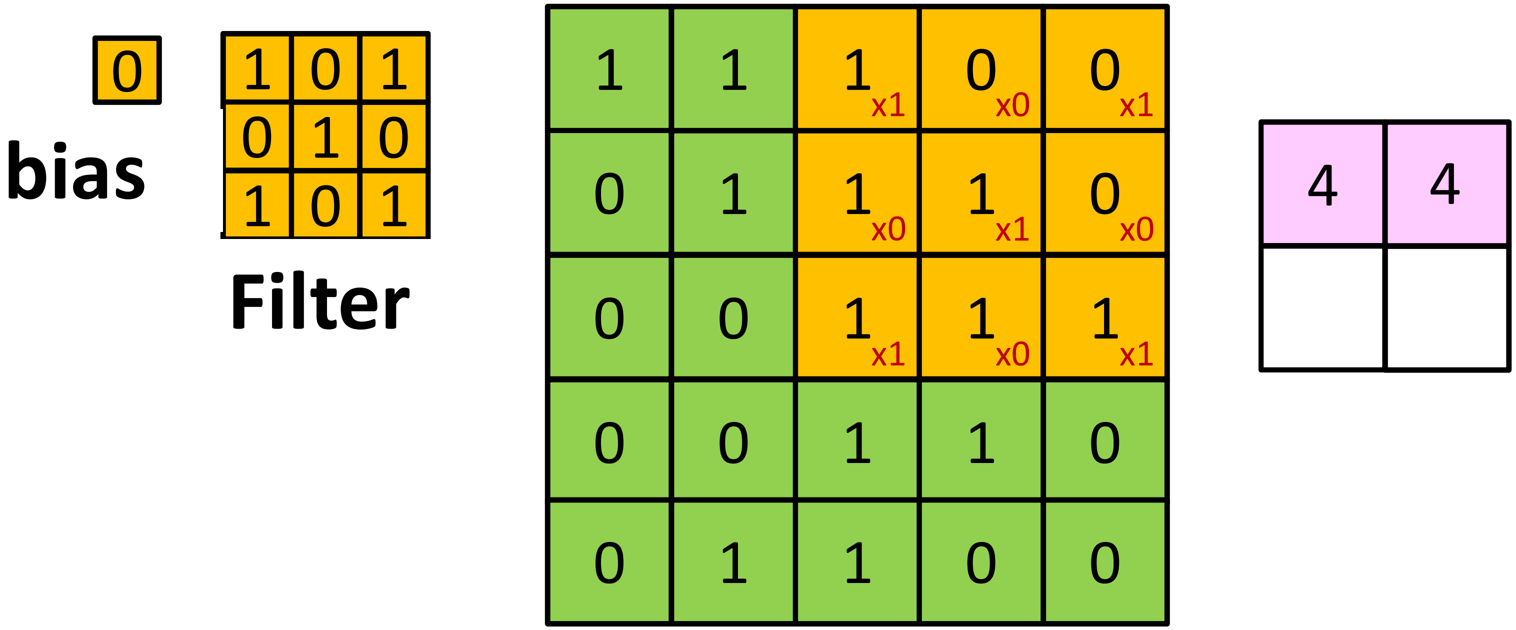
Filter

1 x1	1 x0	1 x1	0	0
0 x0	1 x1	1 x0	1	0
0 x1	0 x0	1 x1	1	1
0	0	1	1	0
0	1	1	0	0

4	

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

Convolution can *shrink* a map by using strides greater than 1



0

bias

1	0	1
0	1	0
1	0	1

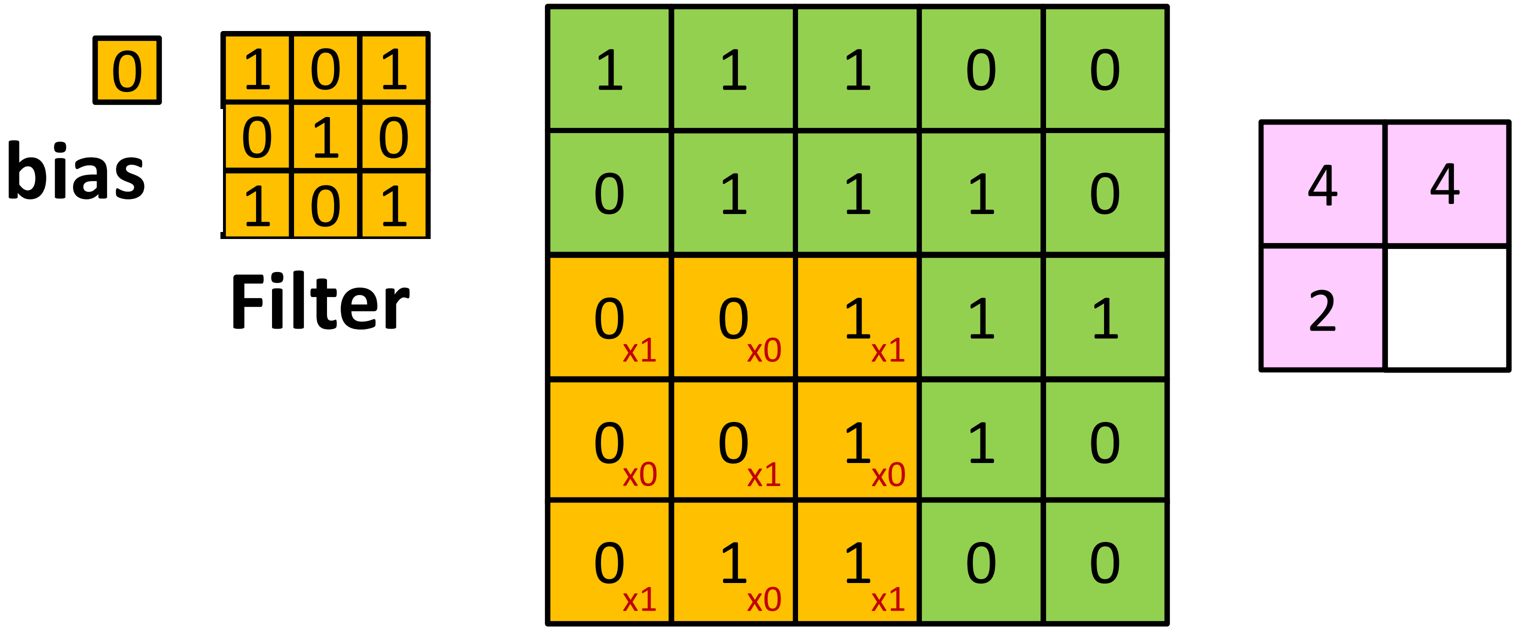
Filter

1	1	1 x1	0 x0	0 x1
0	1	1 x0	1 x1	0 x0
0	0	1 x1	1 x0	1 x1
0	0	1	1	0
0	1	1	0	0

4	4

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

Convolution can *shrink* a map by using strides greater than 1



0

bias

1	0	1
0	1	0
1	0	1

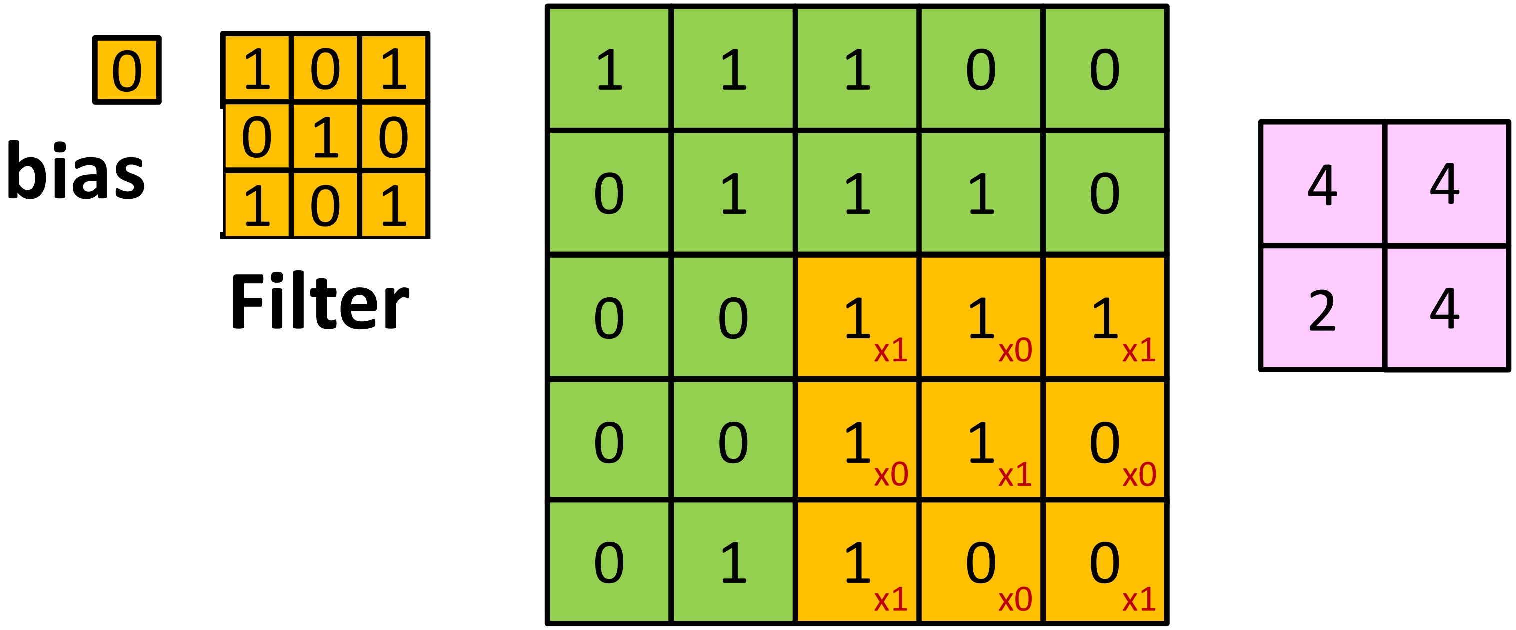
Filter

1	1	1	0	0
0	1	1	1	0
0_{x1}	0_{x0}	1_{x1}	1	1
0_{x0}	0_{x1}	1_{x0}	1	0
0_{x1}	1_{x0}	1_{x1}	0	0

4	4
2	

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

Convolution can *shrink* a map by using strides greater than 1



0

bias

1	0	1
0	1	0
1	0	1

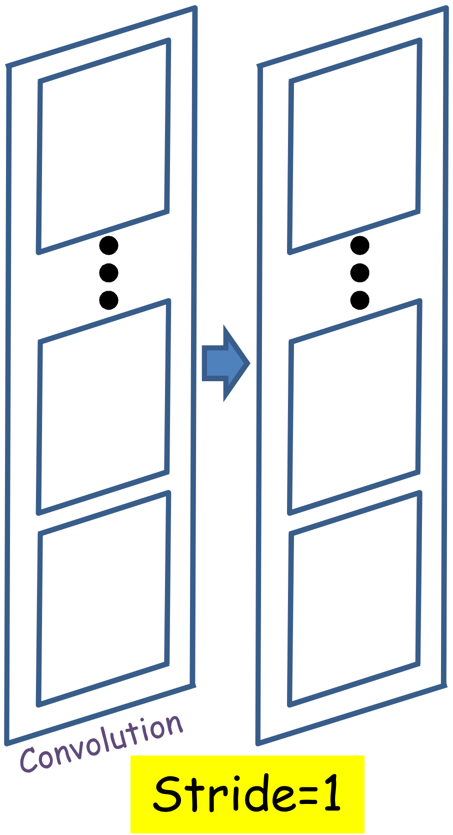
Filter

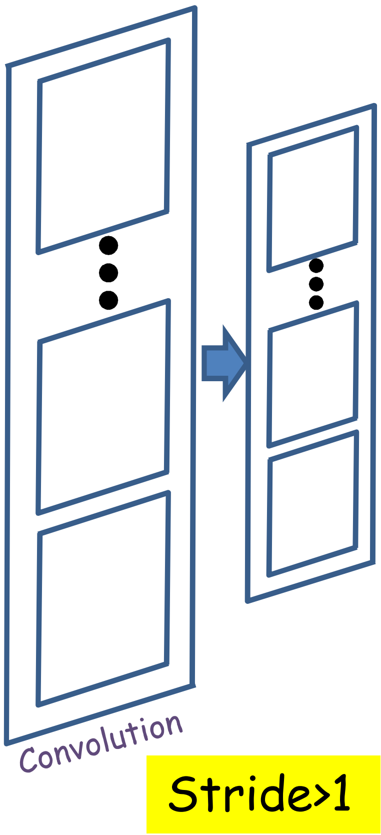
1	1	1	0	0
0	1	1	1	0
0	0	1 x1	1 x0	1 x1
0	0	1 x0	1 x1	0 x0
0	1	1 x1	0 x0	0 x1

4	4
2	4

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

Convolution strides





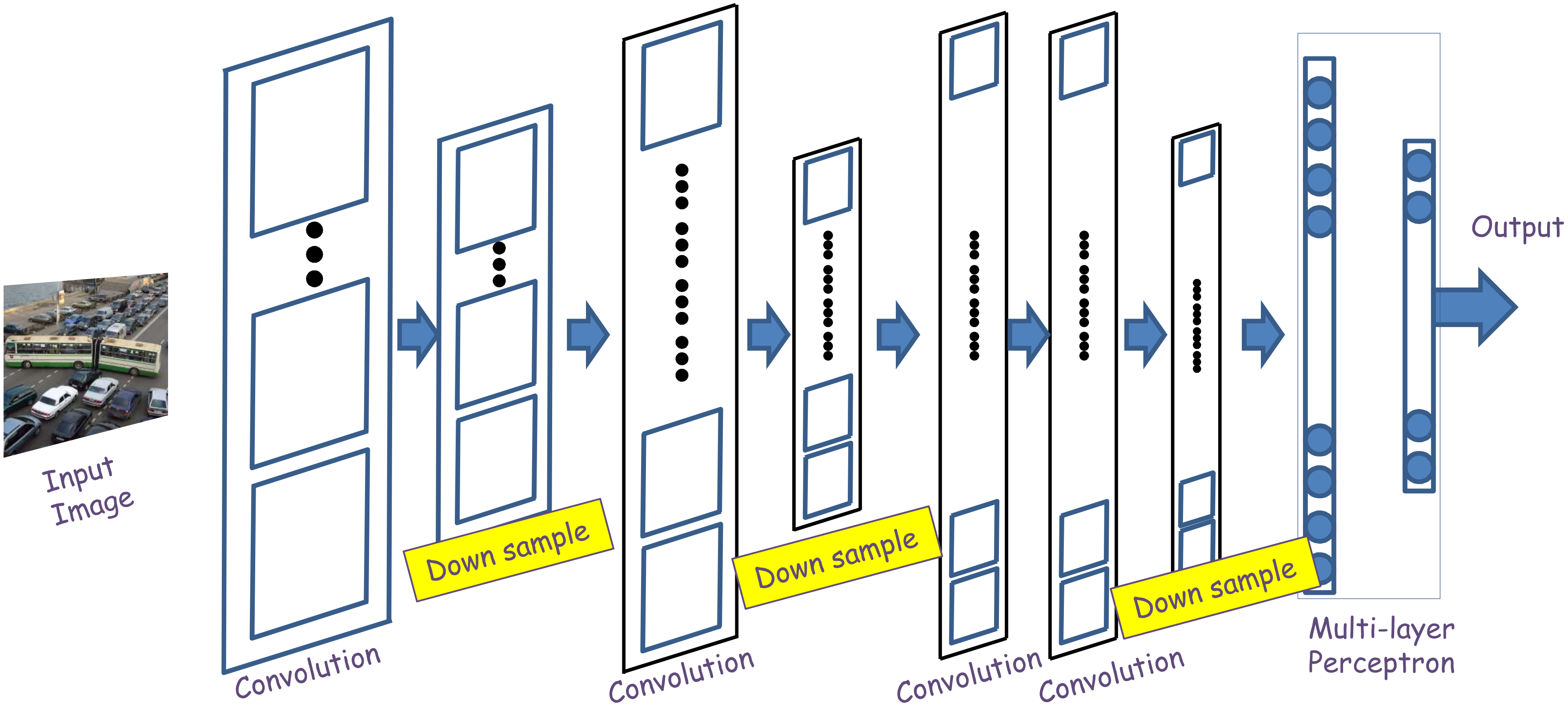
- Convolution with stride 1 → output size same as input size
  - Besides edge effects
- Stride greater than 1 → output size shrinks w.r.t. input

Convolutional neural net: Vector notation

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

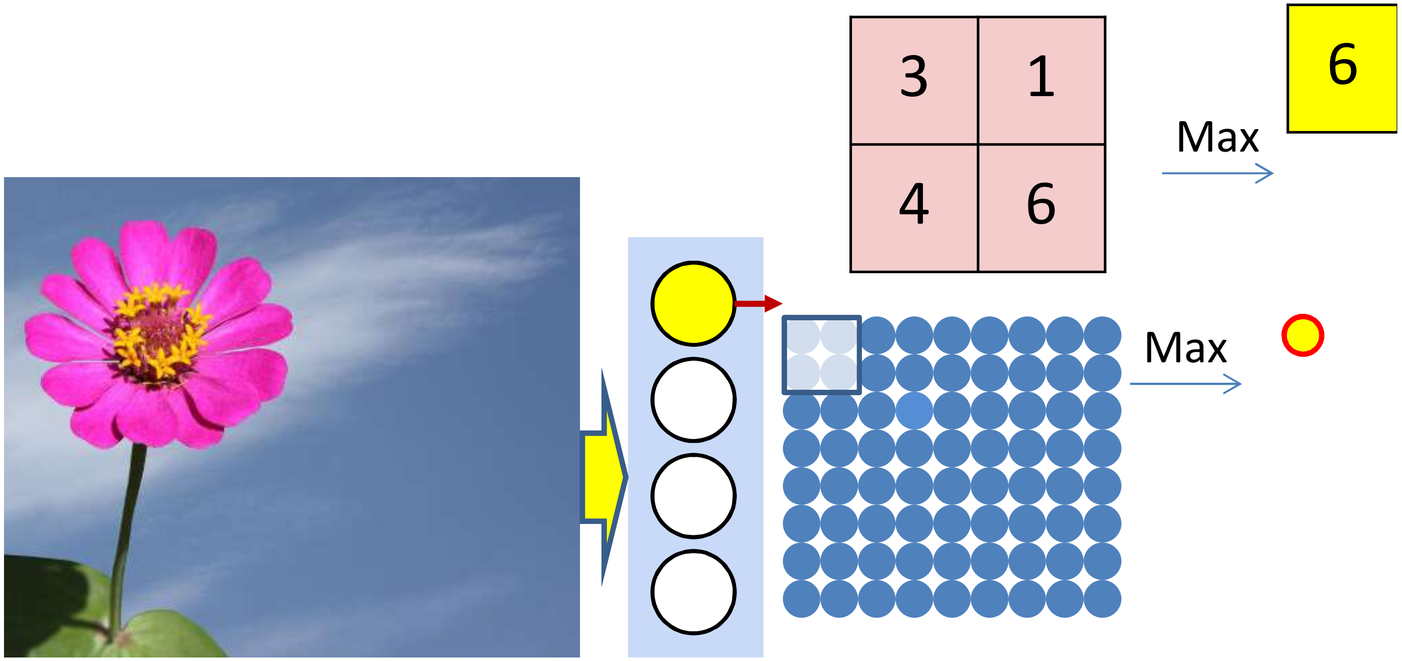
```
Y(0) = Image
for l = 1:L  # layers operate on vector at (x,y)
  for j = 1:$D_l$
      m = 1
      for x = 1:stride:$W_{l-1}-K_l+1$
         n = 1
         for y = 1:stride:$H_{l-1}-K_l+1$
           segment = Y(l-1, :, x:x+$K_l$-1, y:y+$K_l$-1) #3D tensor
           z(l,j,m,n) = W(l,j).segment #tensor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

The other method for shrinking the maps: Downsampling/Pooling



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

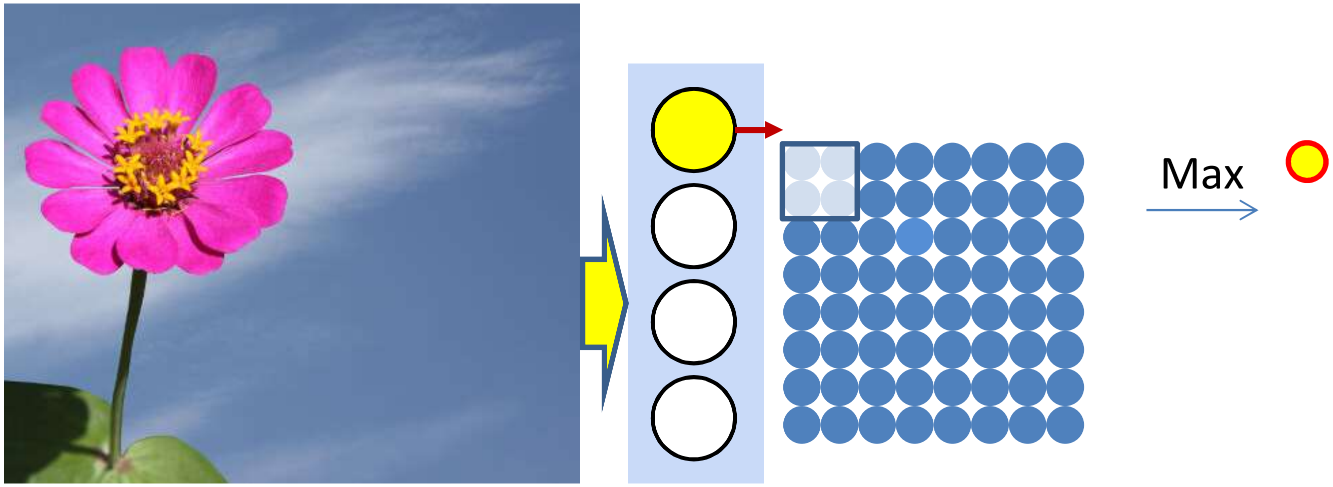
Recall: Max pooling



3	1
4	6

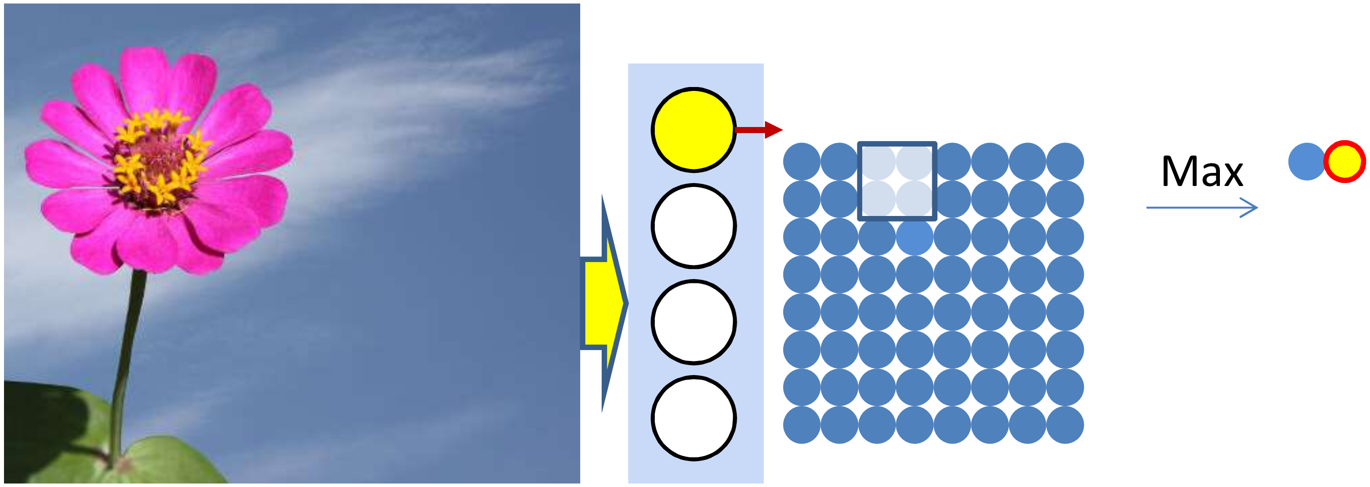
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

Pooling and downsampling



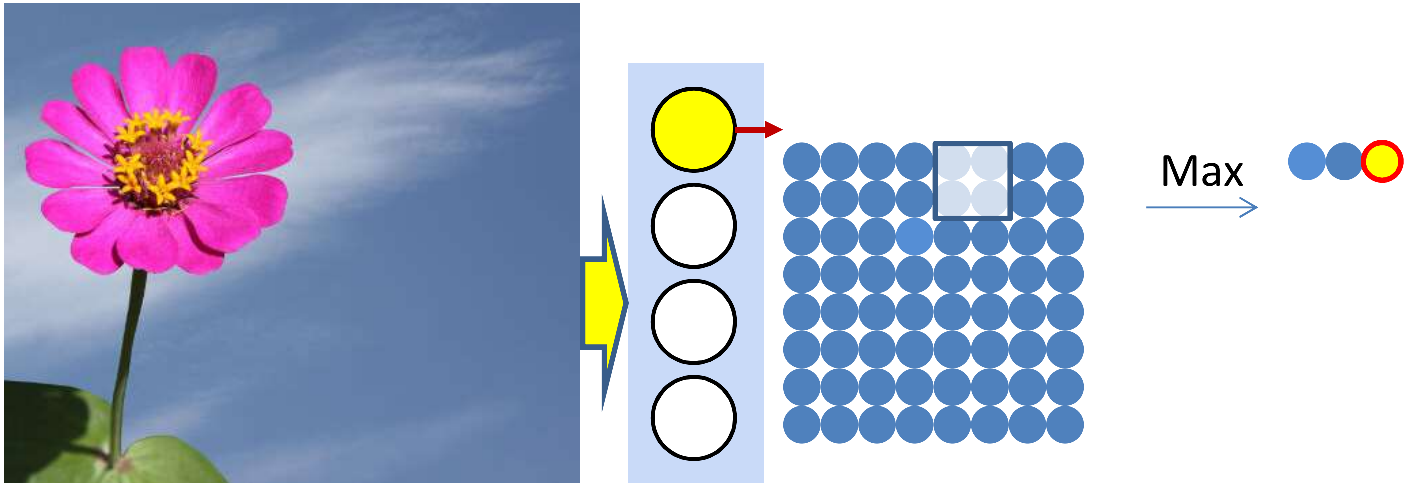
- Pooling is typically performed with strides > 1
  - – Results in shrinking of the map
  - – “Downsampling”

Pooling and downsampling



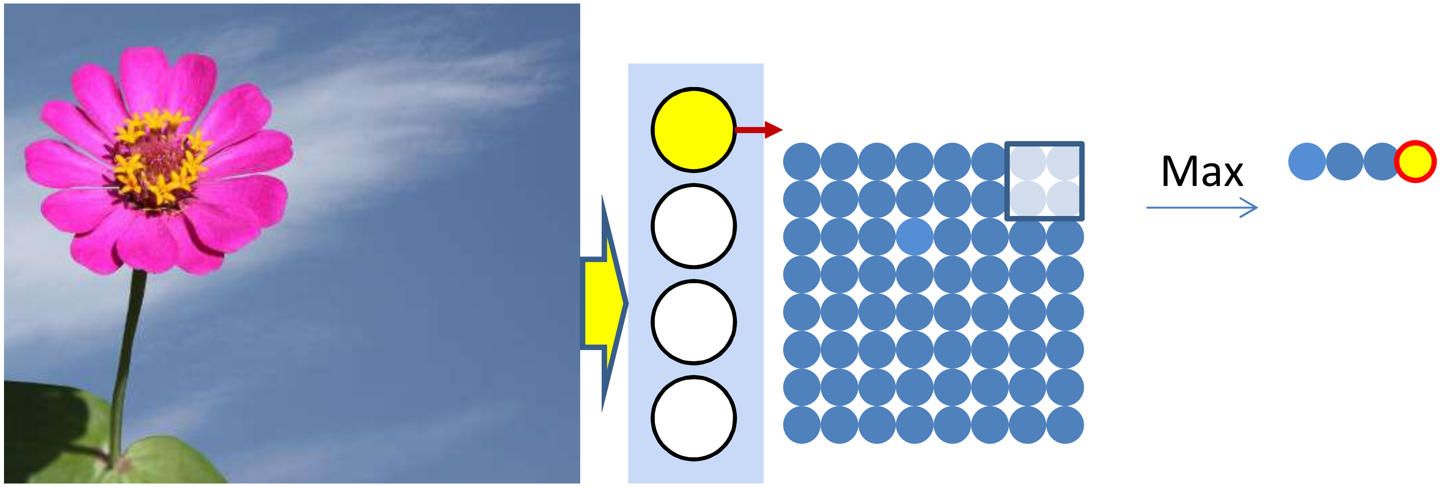
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



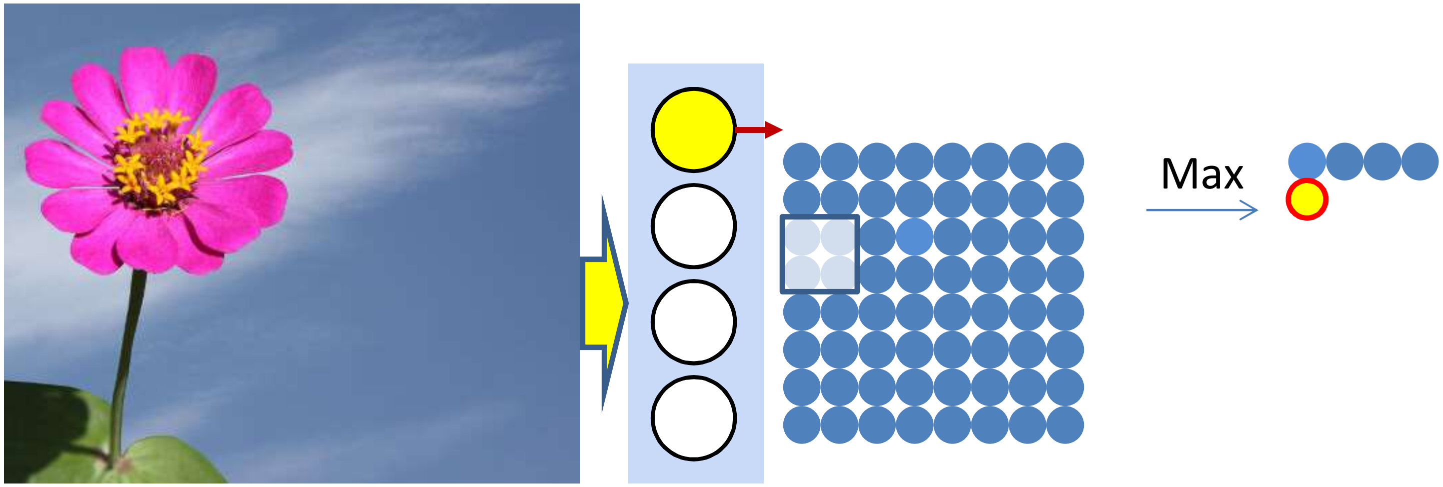
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



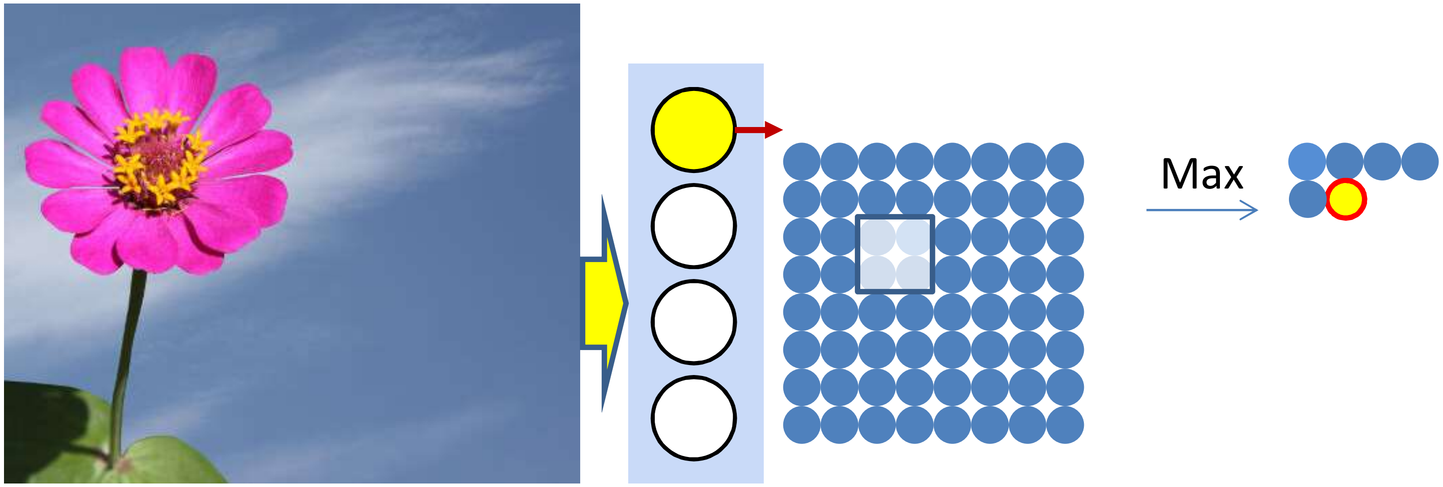
- Pooling is typically performed with strides > 1
  - – Results in shrinking of the map
  - – “Downsampling”

Pooling and downsampling



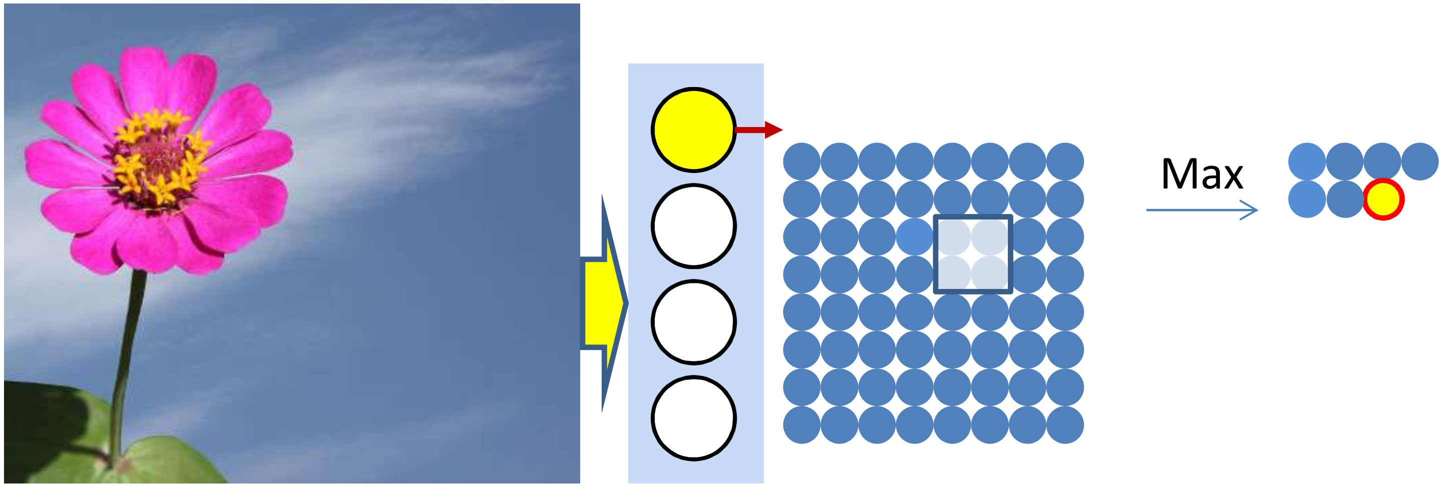
- Pooling is typically performed with strides > 1
  - – Results in shrinking of the map
  - – "Downsampling"

Pooling and downsampling



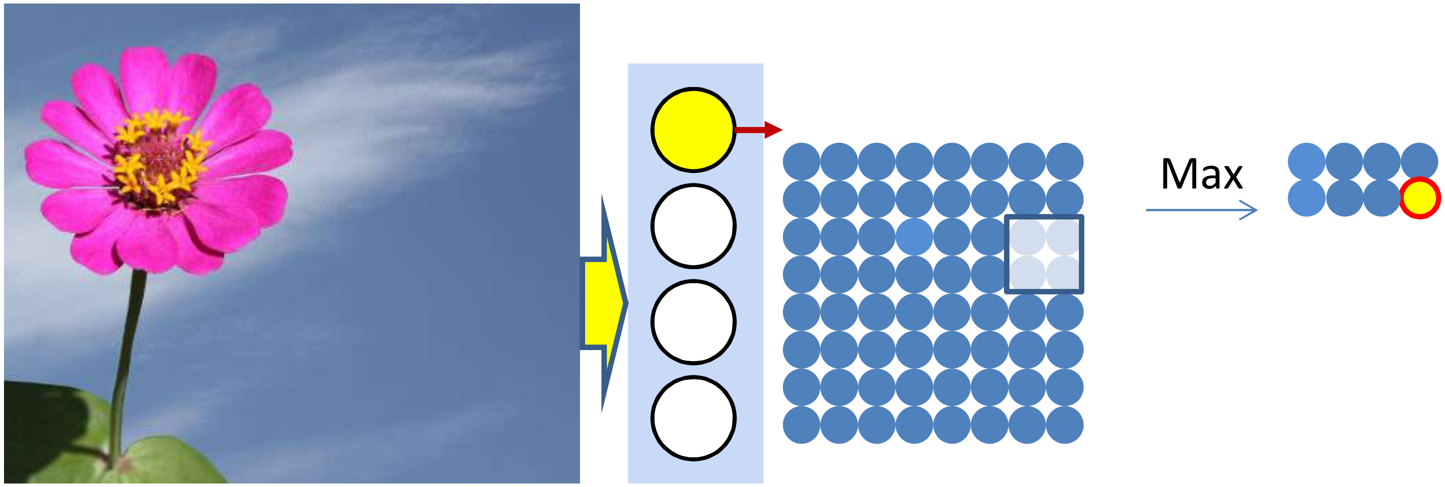
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



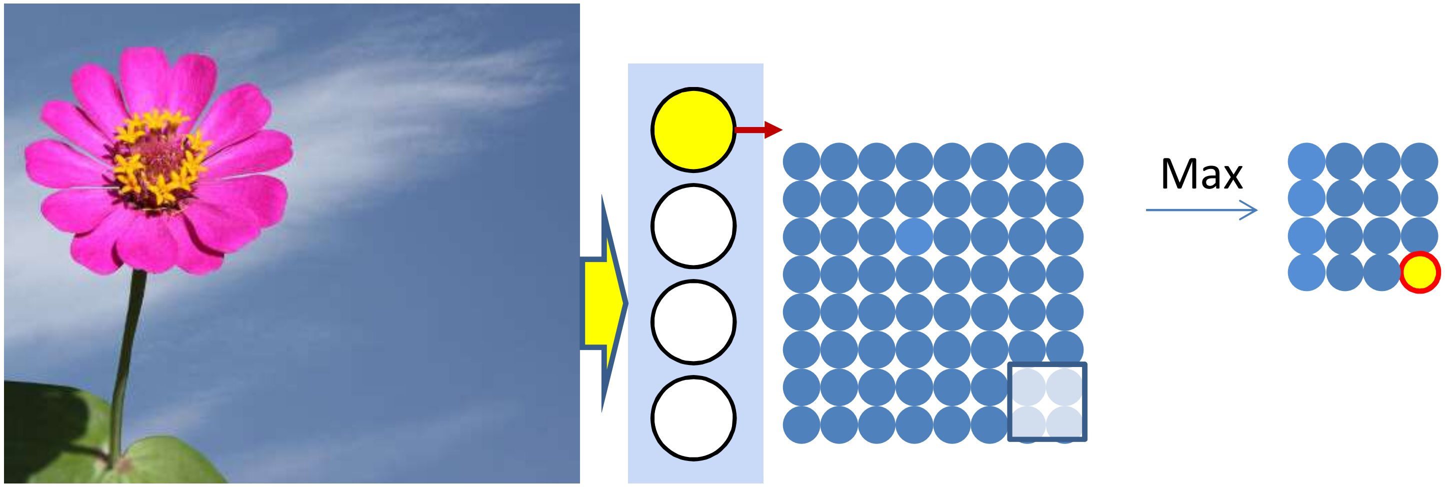
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

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”

Max Pooling layer at layer *l*

a) Performed separately for every map (j).

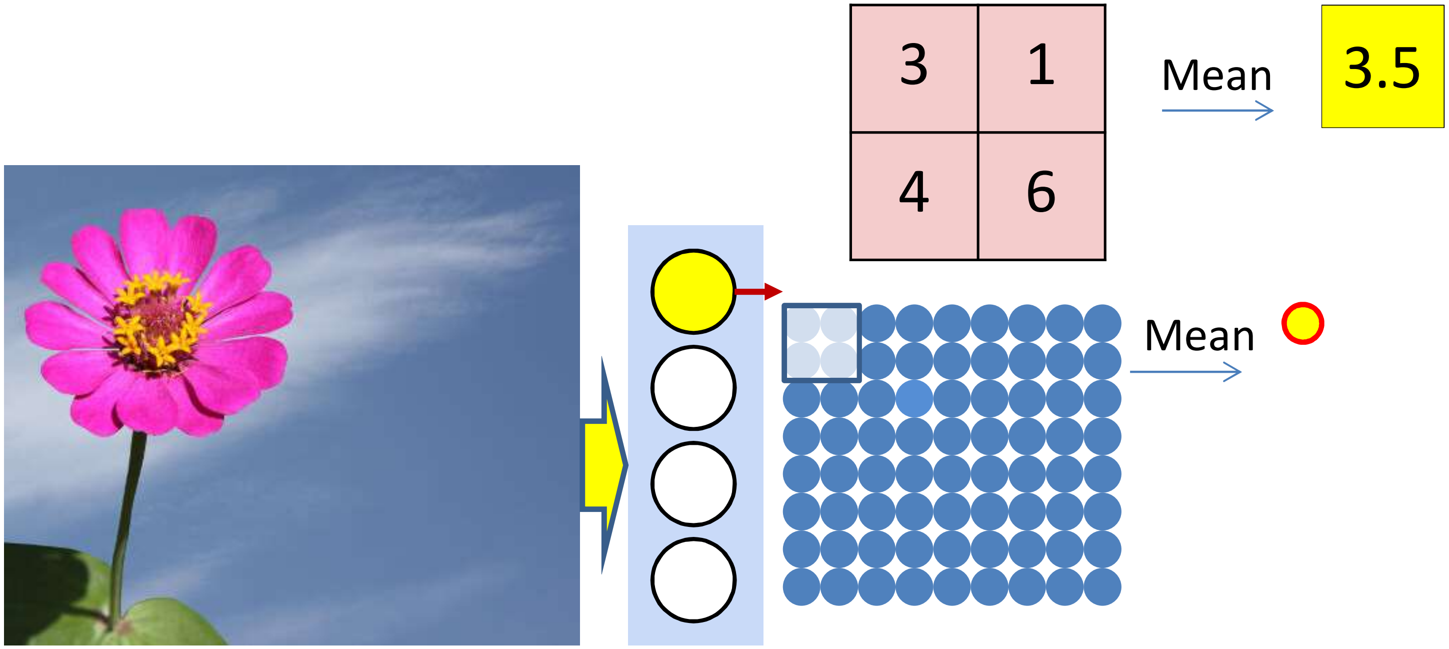
**) Not combining multiple maps within a single max operation.*

b) Keeping track of location of max

Max pooling

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

Max Pooling layer at layer *l*

a) Performed separately for every map (j)

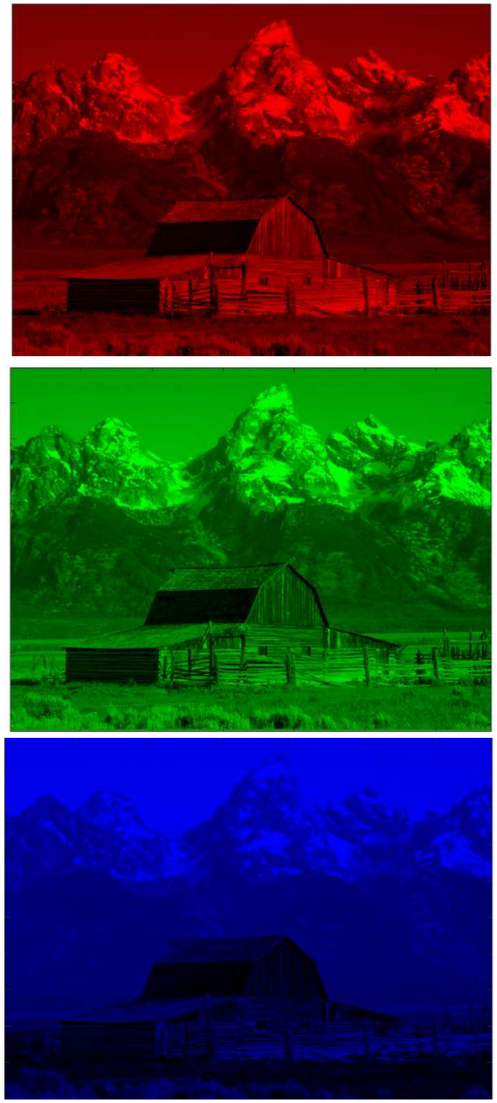
Max pooling

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

Setting everything together

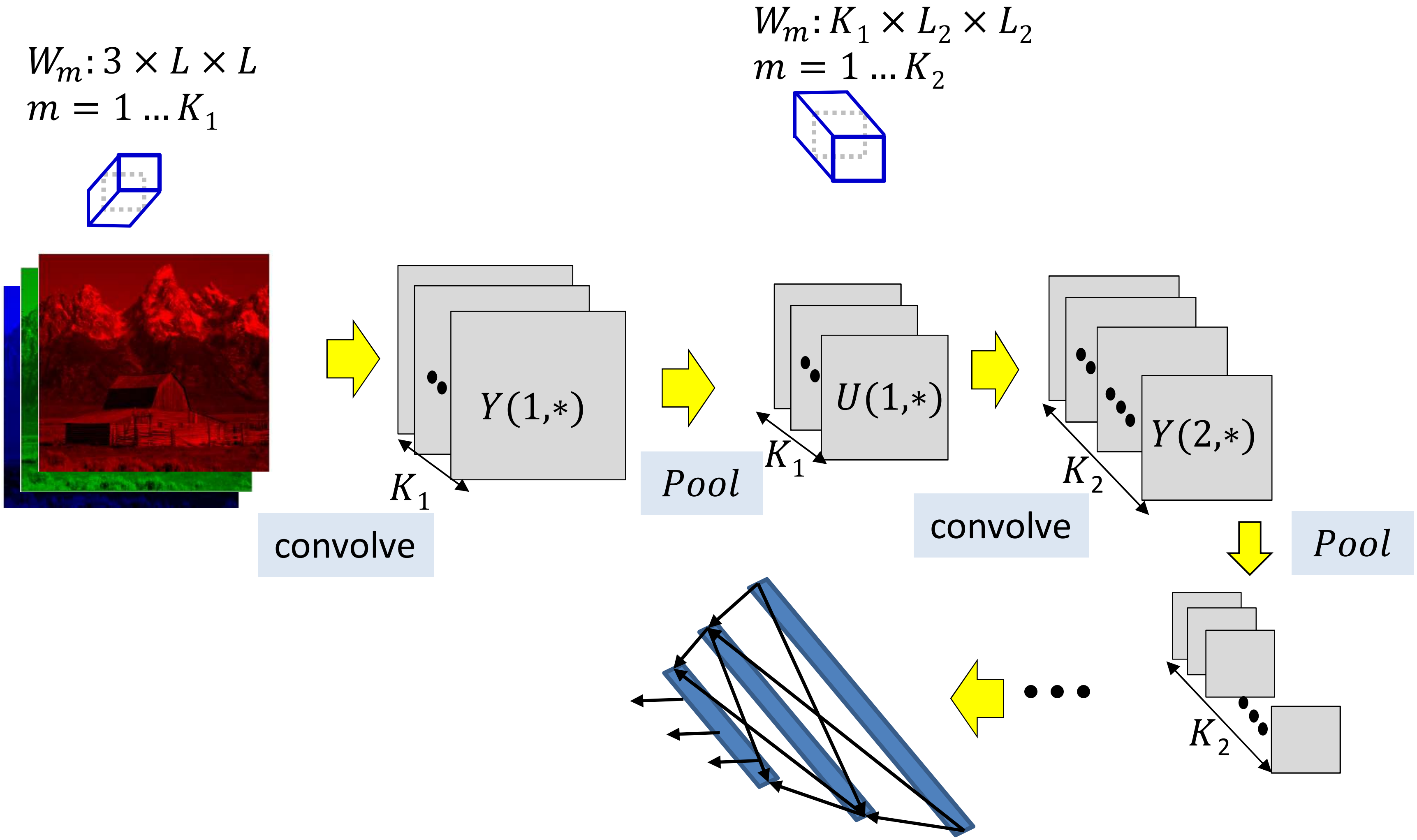
- Typical image classification task
  - Assuming maxpooling..

Convolutional Neural Networks



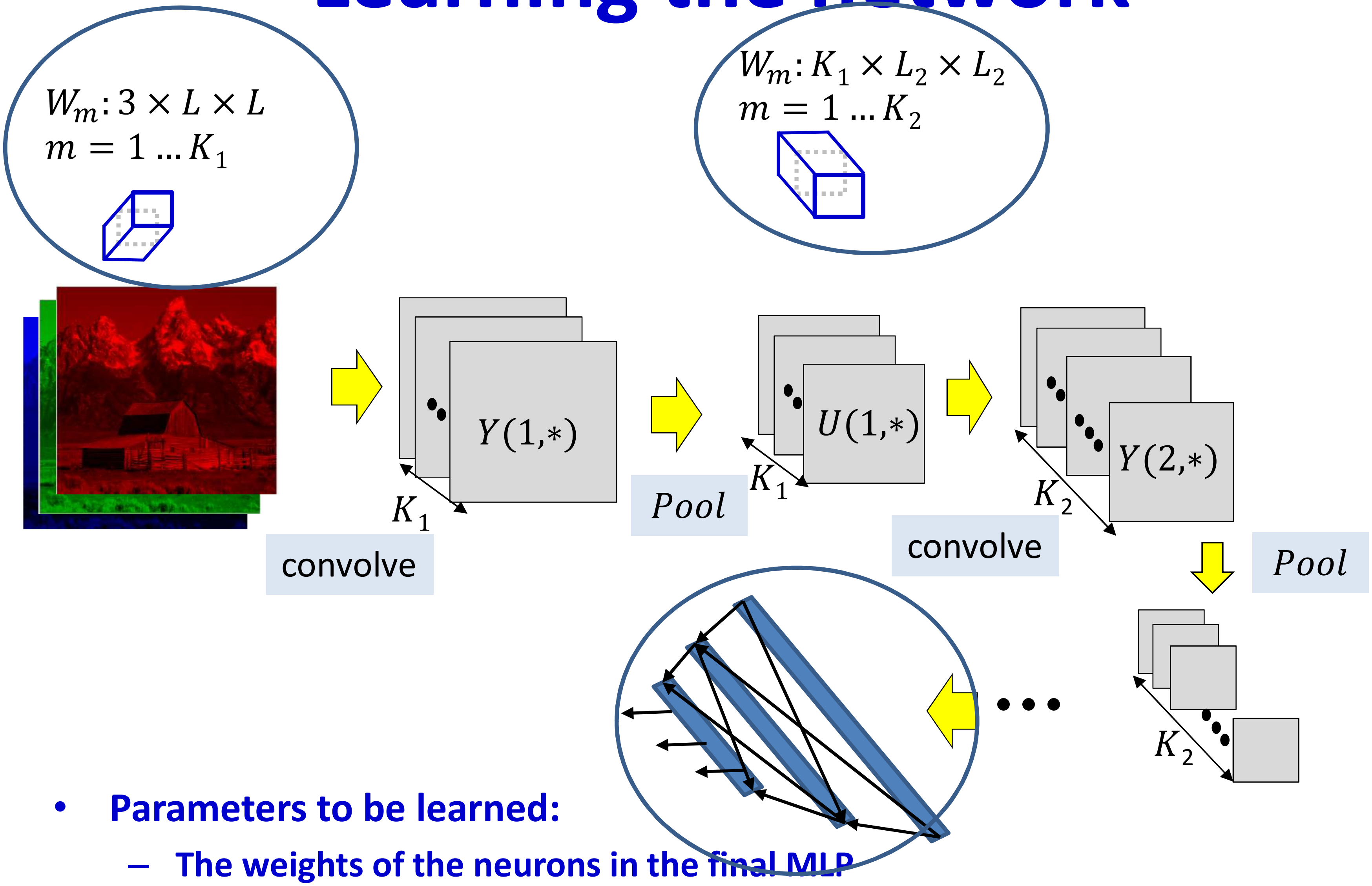
- Input: 1 or 3 images
  - Black and white or color
  - Will assume color to be generic

Convolutional Neural Networks



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

Learning the network



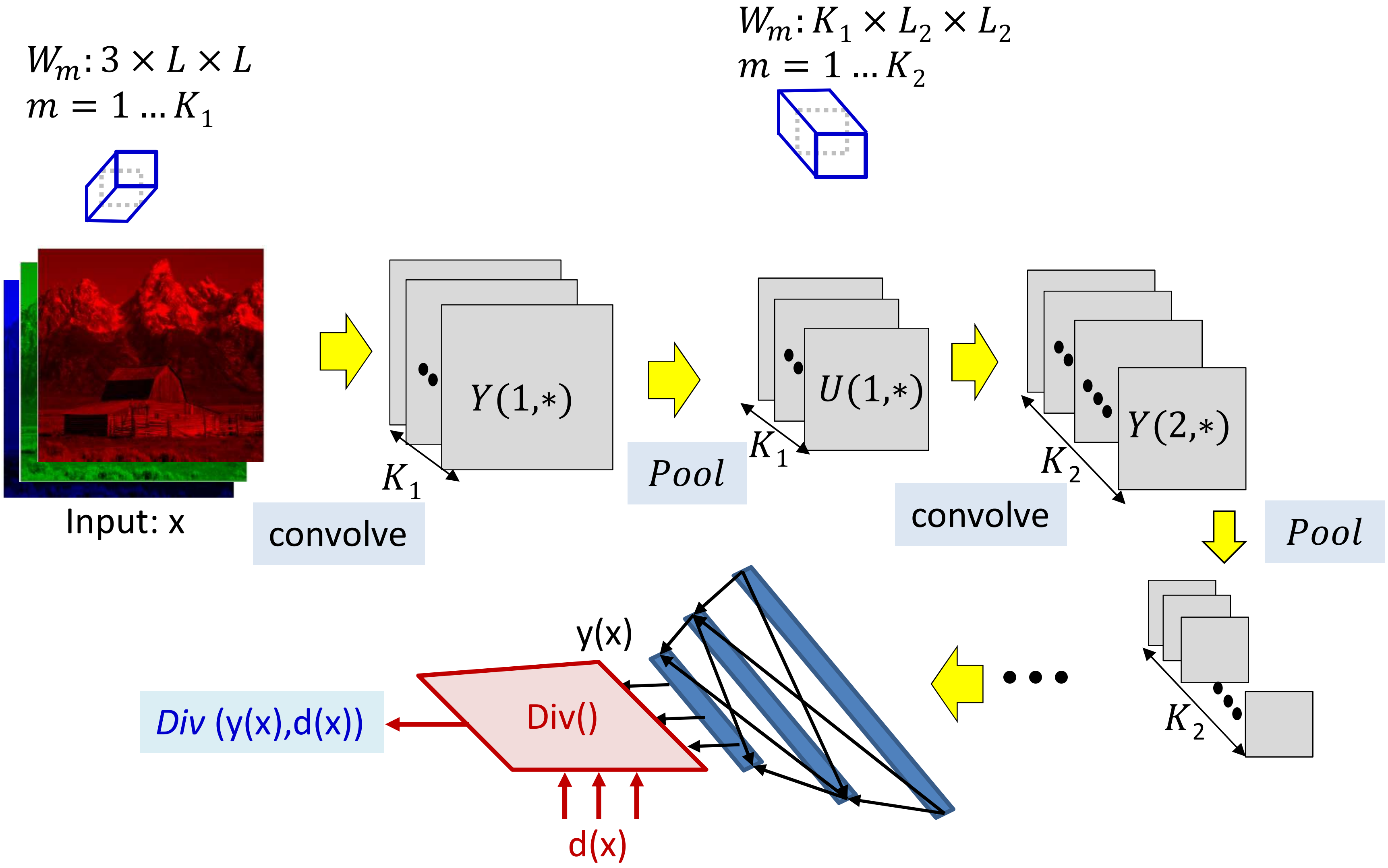
- **Parameters to be learned:**
  - **The weights of the neurons in the final MLP**
  - **The (weights and biases of the) filters for every *convolutional* layer**

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 divergence between the desired output and true output of the network in response to any input
- **Network parameters are trained through variants of gradient descent**
- **Gradients are computed through backpropagation**

Defining the loss



- The loss for a single instance

Problem Setup

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

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

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

Training CNNs through Gradient Descent

Total training error:

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

Assuming the bias is also represented as a weight

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

Training CNNs through Gradient Descent

Total training error:

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

Assuming the bias is also represented as a weight

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

The derivative

Total training error:

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

- Computing the derivative

Total derivative:

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

The derivative

Total training error:

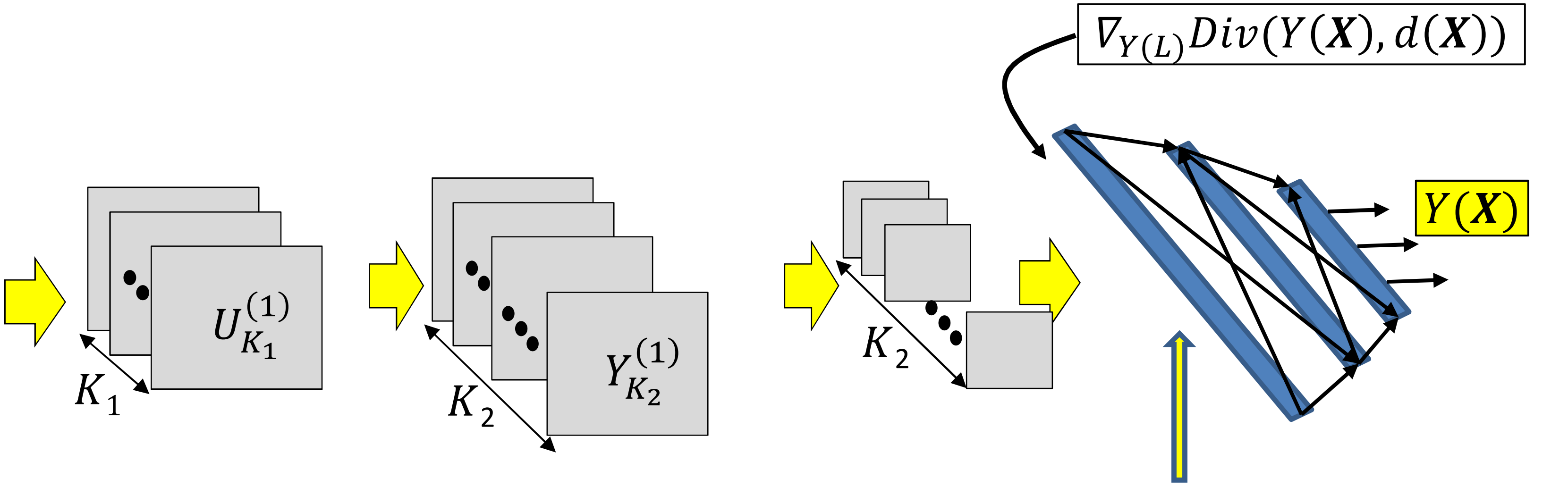
$$Err = \frac{1}{T}\sum_{i} Div(Y_i, d_i)$$

- Computing the derivative

Total derivative:

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

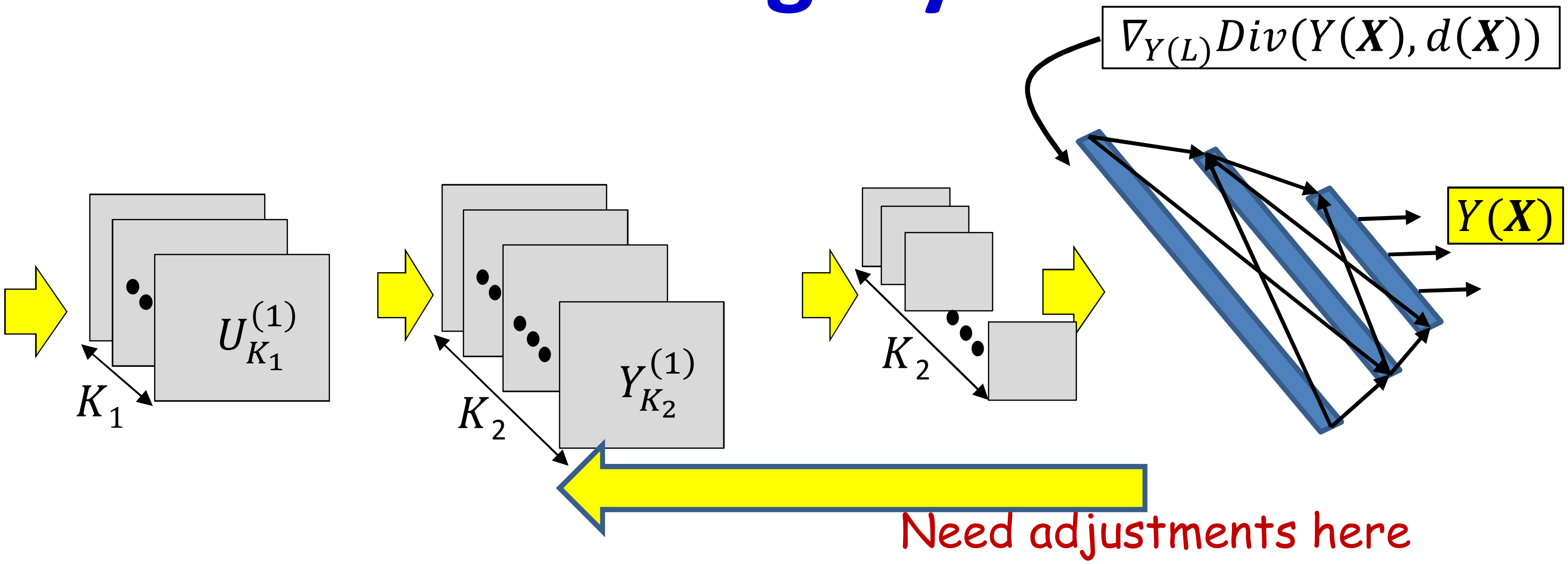
Backpropagation: Final flat layers



Conventional backprop until here

- 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

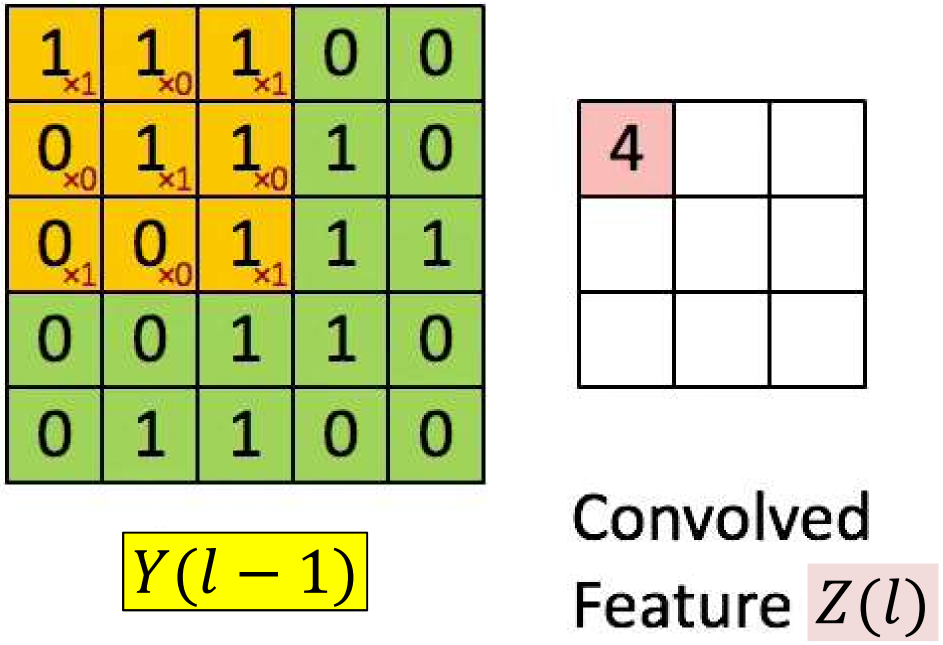
Backpropagation: Convlutional and Pooling layers



Need adjustments here

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

BP: Convolutional layer



$1_{\times 1}$	$1_{\times 0}$	$1_{\times 1}$	0	0
$0_{\times 0}$	$1_{\times 1}$	$1_{\times 0}$	1	0
$0_{\times 1}$	$0_{\times 0}$	$1_{\times 1}$	1	1
0	0	1	1	0
0	1	1	0	0

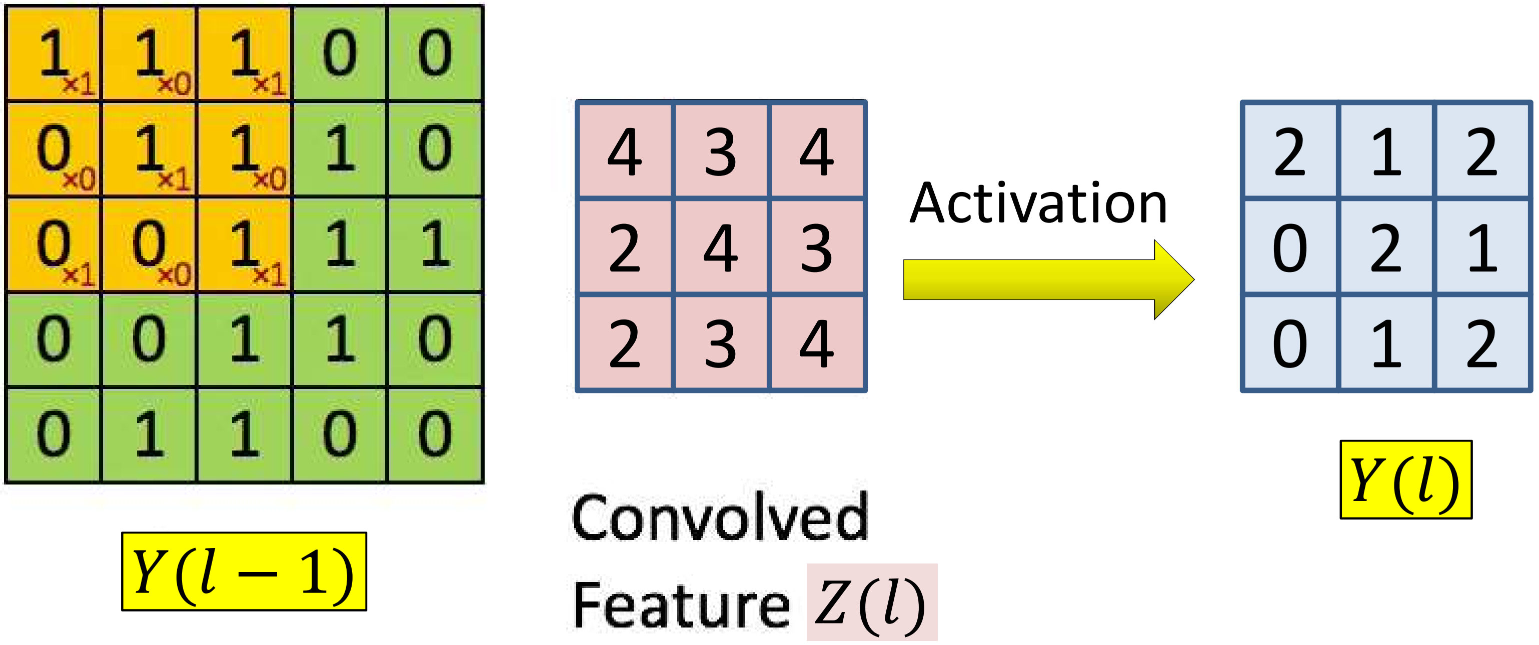
$Y(l-1)$

4		

Convolved Feature $Z(l)$

- For every l^{th} layer filter, each position in the map in the $l-1^{\text{th}}$ layer affects several positions in the map of the l^{th} layer

BP: Convolutional layer



$1_{\times 1}$	$1_{\times 0}$	$1_{\times 1}$	0	0
$0_{\times 0}$	$1_{\times 1}$	$1_{\times 0}$	1	0
$0_{\times 1}$	$0_{\times 0}$	$1_{\times 1}$	1	1
0	0	1	1	0
0	1	1	0	0

$Y(l-1)$

4	3	4
2	4	3
2	3	4

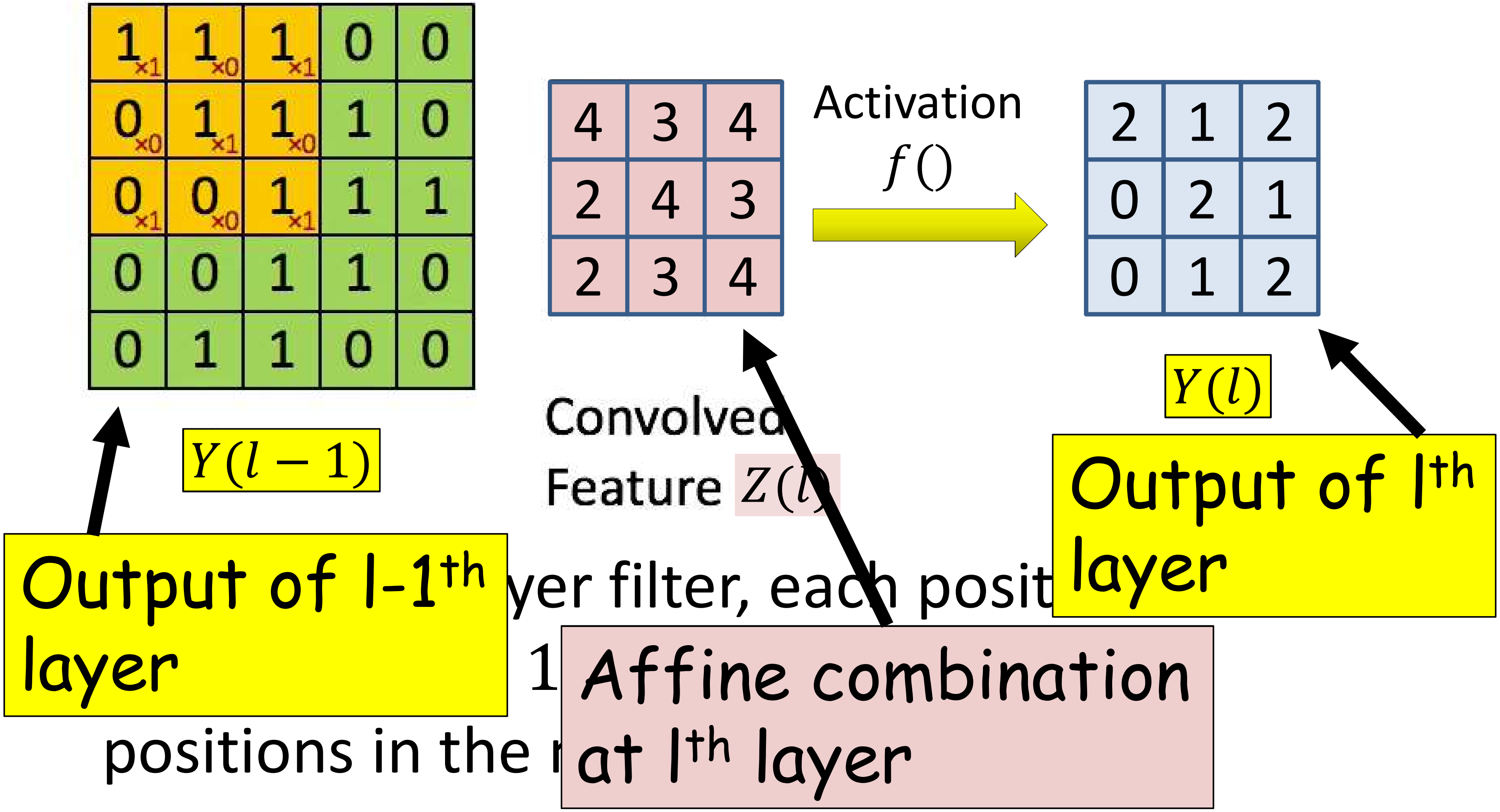
Convolved Feature $Z(l)$

2	1	2
0	2	1
0	1	2

$Y(l)$

- For every l^{th} layer filter, each position in the map in the $l-1^{\text{th}}$ layer affects several positions in the map of the l^{th} layer

BP: Convolutional layer



1 ×1	1 ×0	1 ×1	0	0
0 ×0	1 ×1	1 ×0	1	0
0 ×1	0 ×0	1 ×1	1	1
0	0	1	1	0
0	1	1	0	0

$Y(l-1)$

4	3	4
2	4	3
2	3	4

Convolved Feature $Z(l)$

Activation $f()$

2	1	2
0	2	1
0	1	2

$Y(l)$

Output of l-1th layer

Output of lth layer

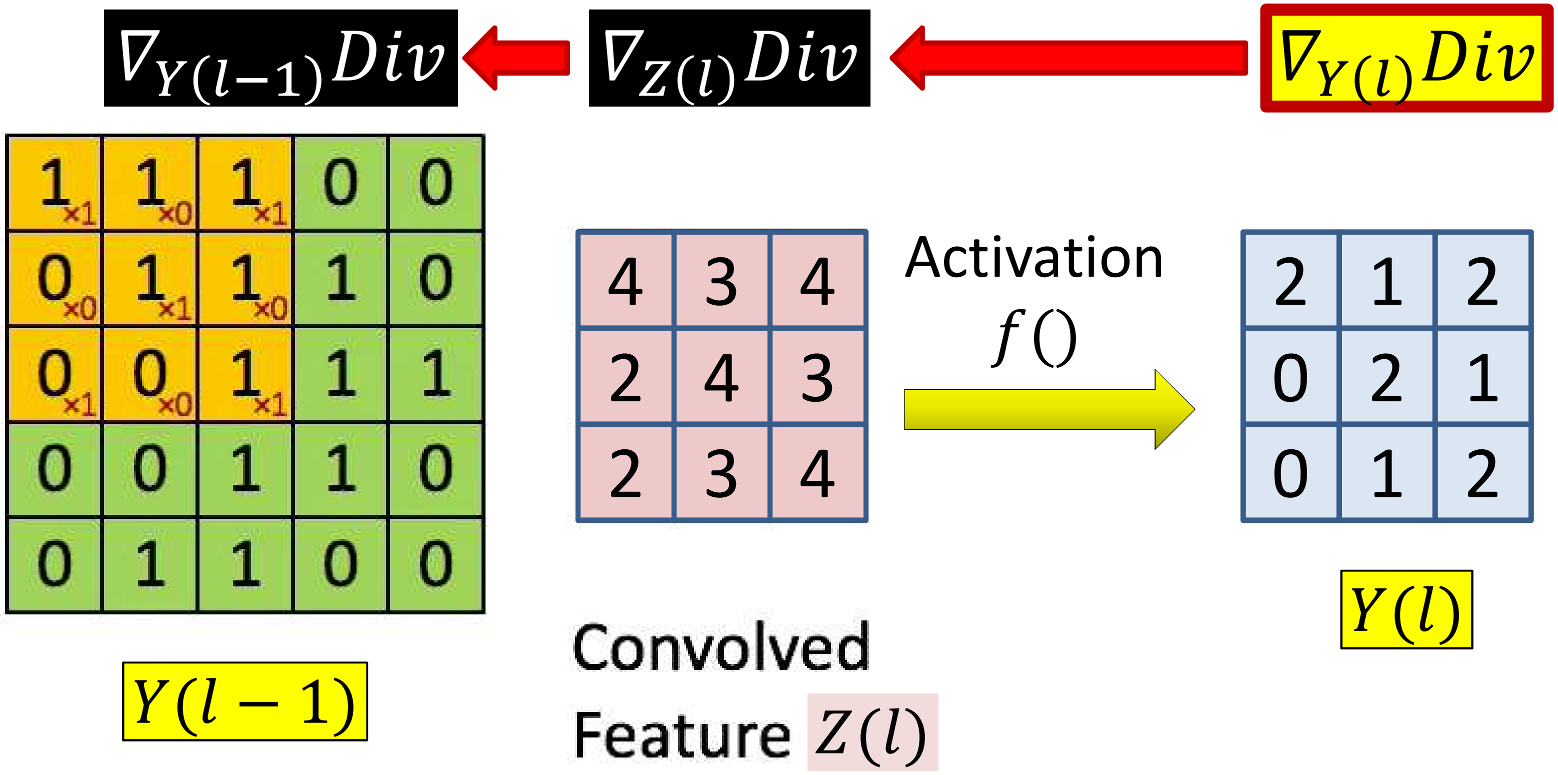
Affine combination at lth layer

yer filter, each posit

1

positions in the r

BP: Convolutional layer



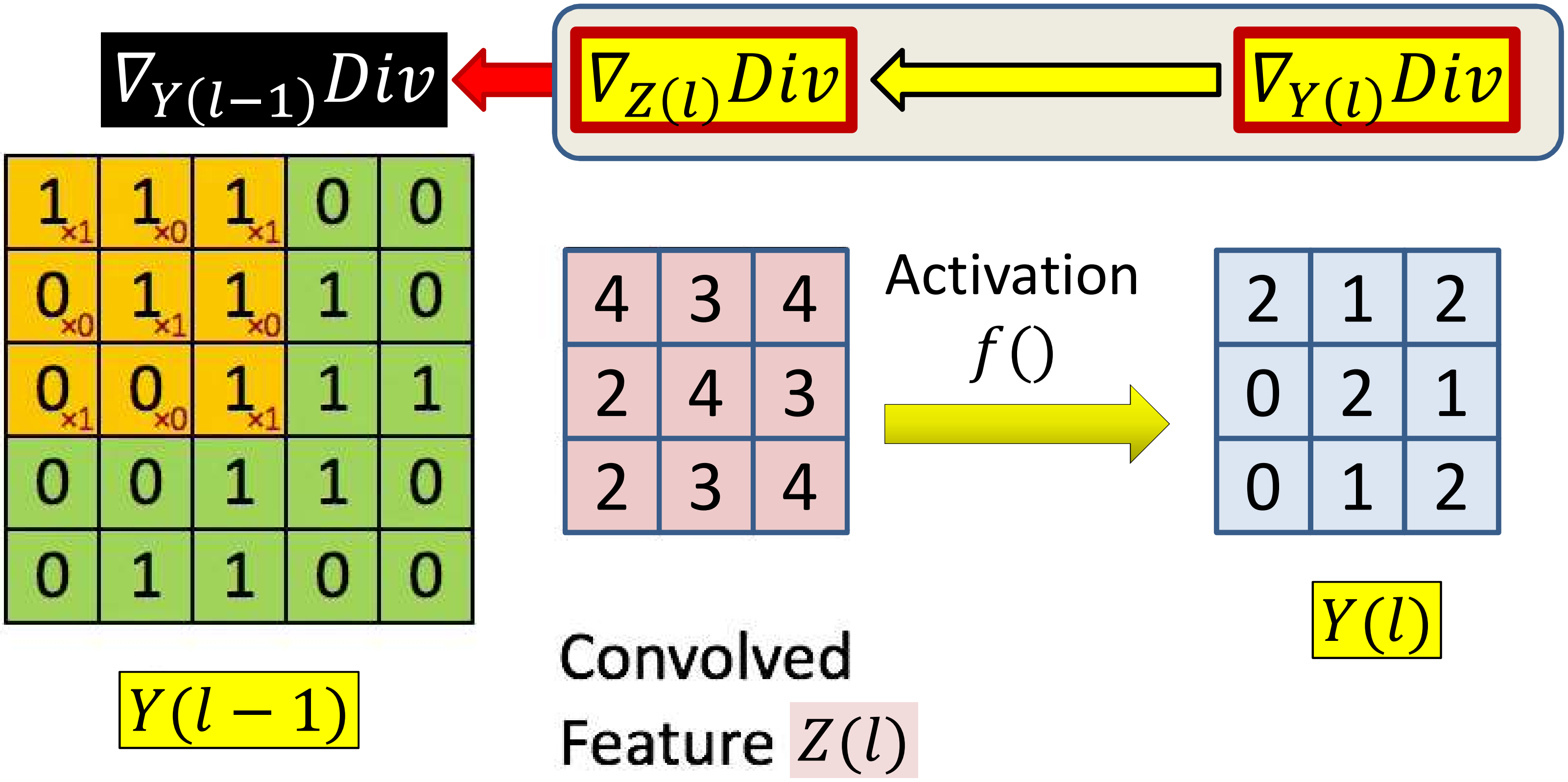
$1_{\times1}$	$1_{\times0}$	$1_{\times1}$	0	0
$0_{\times0}$	$1_{\times1}$	$1_{\times0}$	1	0
$0_{\times1}$	$0_{\times0}$	$1_{\times1}$	1	1
0	0	1	1	0
0	1	1	0	0

4	3	4
2	4	3
2	3	4

2	1	2
0	2	1
0	1	2

- Assuming $\nabla_{Y(l)}Div$ is available
  - Remember – it is available for the Lth layer already from the flat MLP
- Must compute $\nabla_{Z(l)}Div$ and $\nabla_{Y(l-1)}Div$

BP: Convolutional layer



$\nabla_{Y(l-1)}Div$

$\nabla_{Z(l)}Div$

$\nabla_{Y(l)}Div$

$1_{\times 1}$	$1_{\times 0}$	$1_{\times 1}$	0	0
$0_{\times 0}$	$1_{\times 1}$	$1_{\times 0}$	1	0
$0_{\times 1}$	$0_{\times 0}$	$1_{\times 1}$	1	1
0	0	1	1	0
0	1	1	0	0

$Y(l-1)$

4	3	4
2	4	3
2	3	4

Convolved Feature $Z(l)$

Activation $f()$

2	1	2
0	2	1
0	1	2

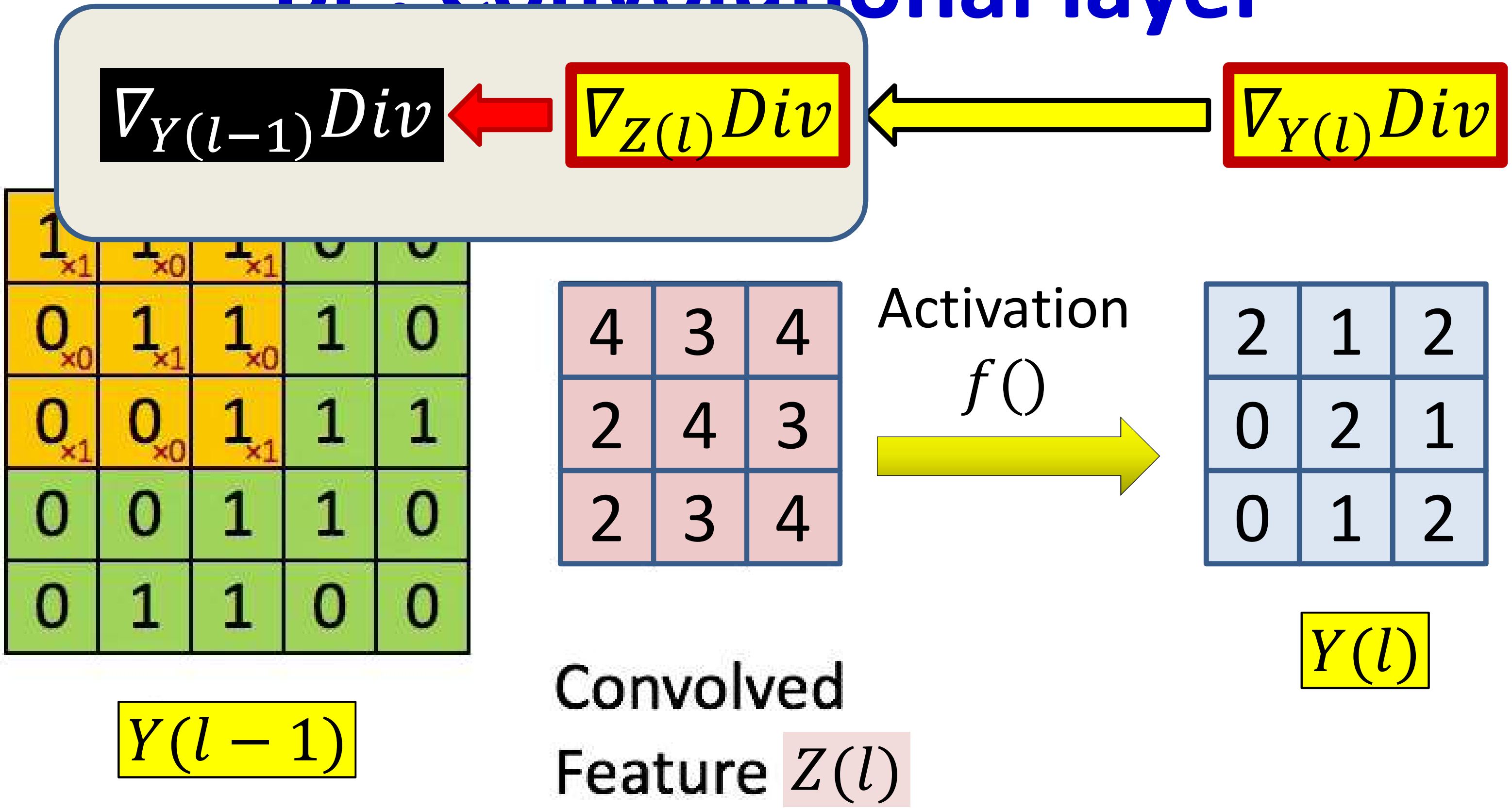
$Y(l)$

- Computing $\nabla_{Z(l)}Div$

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

- Simple component-wise computation

BP: Convolutional layer



$\nabla_{Y(l-1)}Div$ ← $\nabla_{Z(l)}Div$ ← $\nabla_{Y(l)}Div$

1 ×1	1 ×0	1 ×1	0	0
0 ×0	1 ×1	1 ×0	1	0
0 ×1	0 ×0	1 ×1	1	1
0	0	1	1	0
0	1	1	0	0

$Y(l-1)$

4	3	4
2	4	3
2	3	4

Convolved Feature $Z(l)$

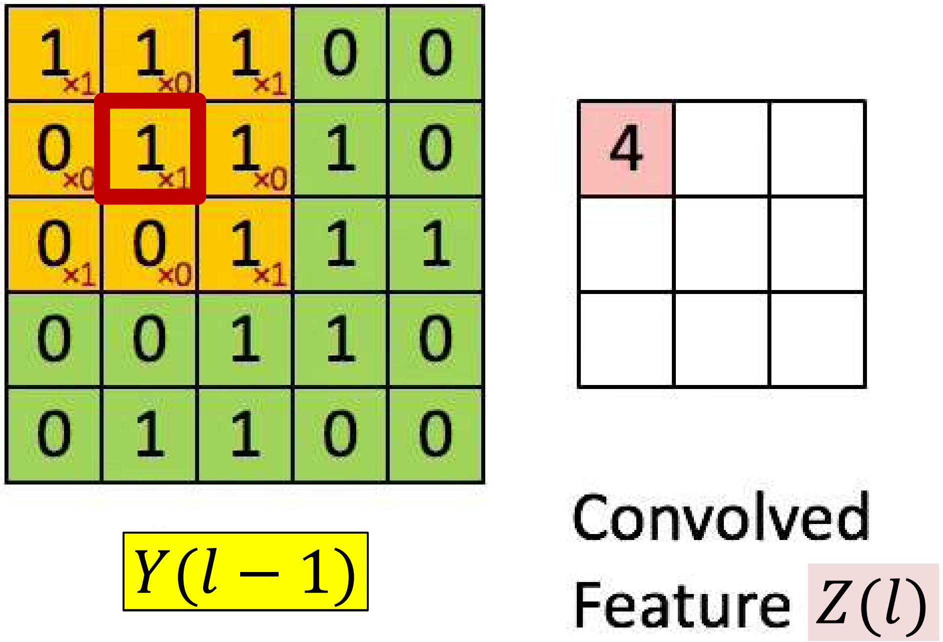
Activation $f()$

2	1	2
0	2	1
0	1	2

$Y(l)$

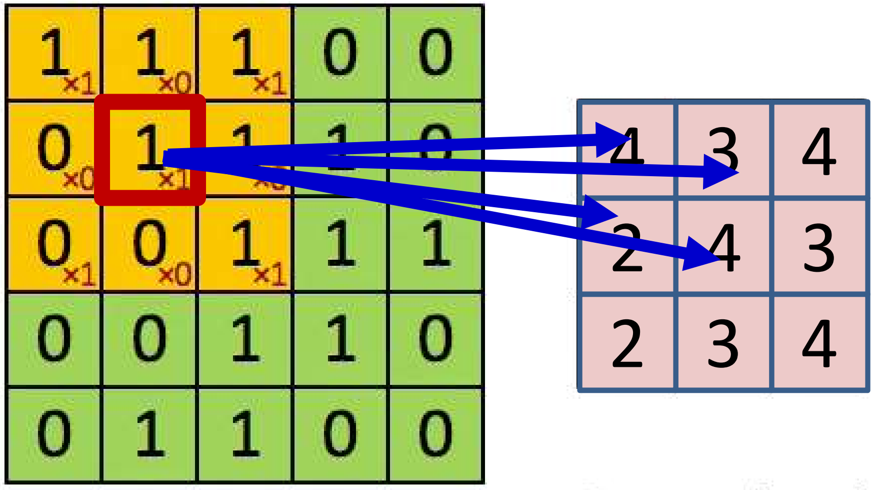
- Computing $\nabla_{Y(l-1)}Div$ and $\nabla_{W(l)}Div$
- Each $y(l-1, m, x, y)$ affects several $z(l, *, x, y)$ terms
  - All of them contribute to the derivative w.r.t. $y(l-1, m, x, y)$

BP: Convolutional layer



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

BP: Convolutional layer



$1_{\times 1}$	$1_{\times 0}$	$1_{\times 1}$	0	0
$0_{\times 0}$	$1_{\times 1}$	$1_{\times 0}$	1	0
$0_{\times 1}$	$0_{\times 0}$	$1_{\times 1}$	1	1
0	0	1	1	0
0	1	1	0	0

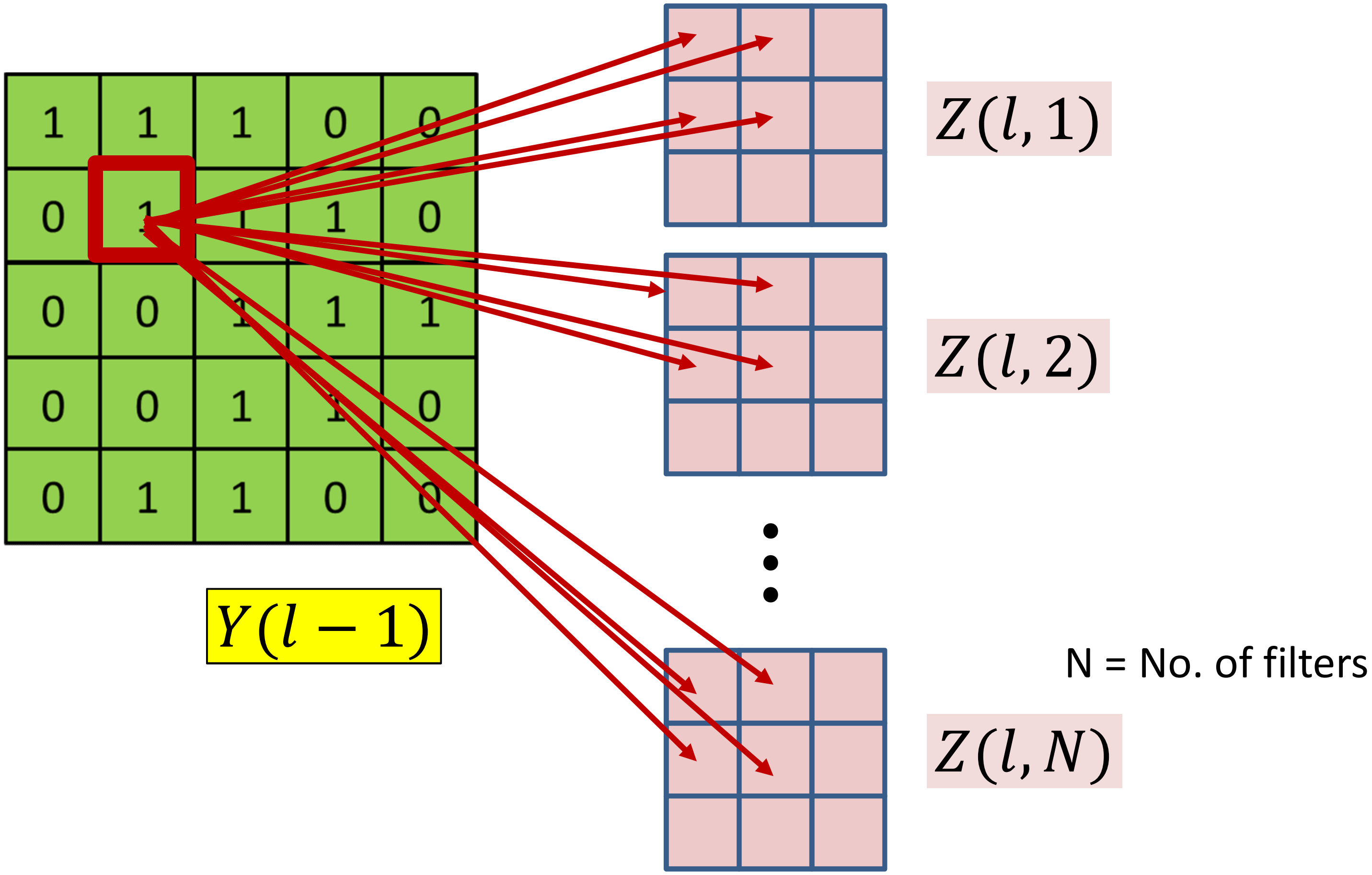
$Y(l-1)$

4	3	4
2	4	3
2	3	4

Convolved
Feature $Z(l)$

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

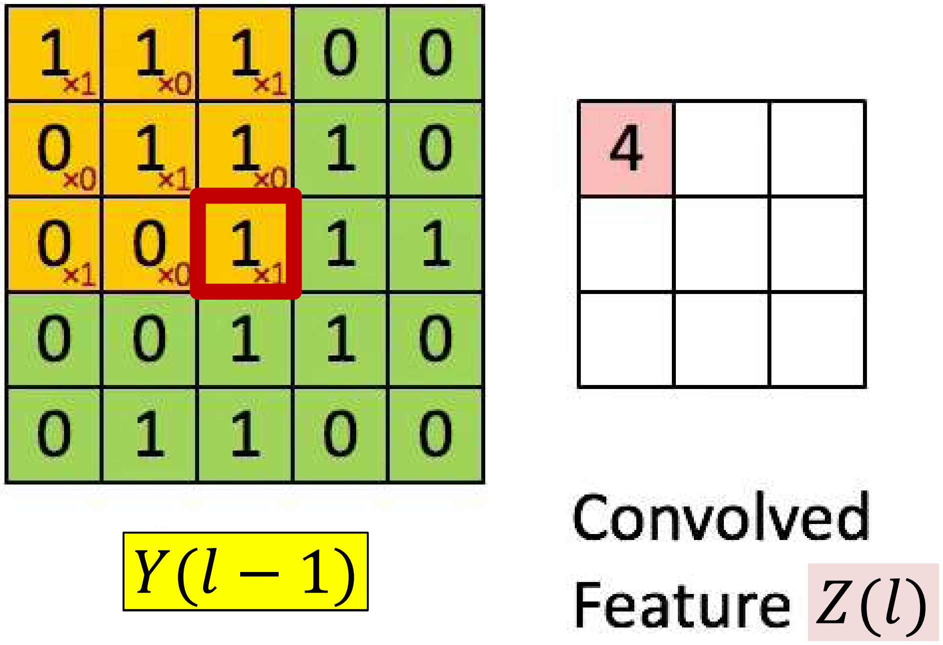
BP: Convolutional layer



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

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

BP: Convolutional layer



1 ×1	1 ×0	1 ×1	0	0
0 ×0	1 ×1	1 ×0	1	0
0 ×1	0 ×0	1 ×1	1	1
0	0	1	1	0
0	1	1	0	0

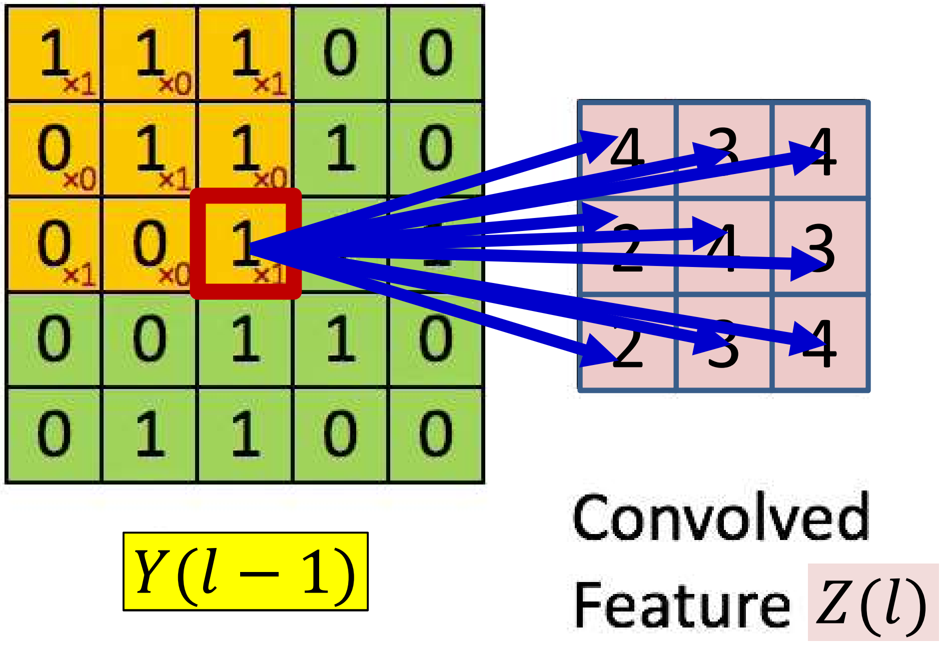
$Y(l-1)$

4		

Convolved Feature $Z(l)$

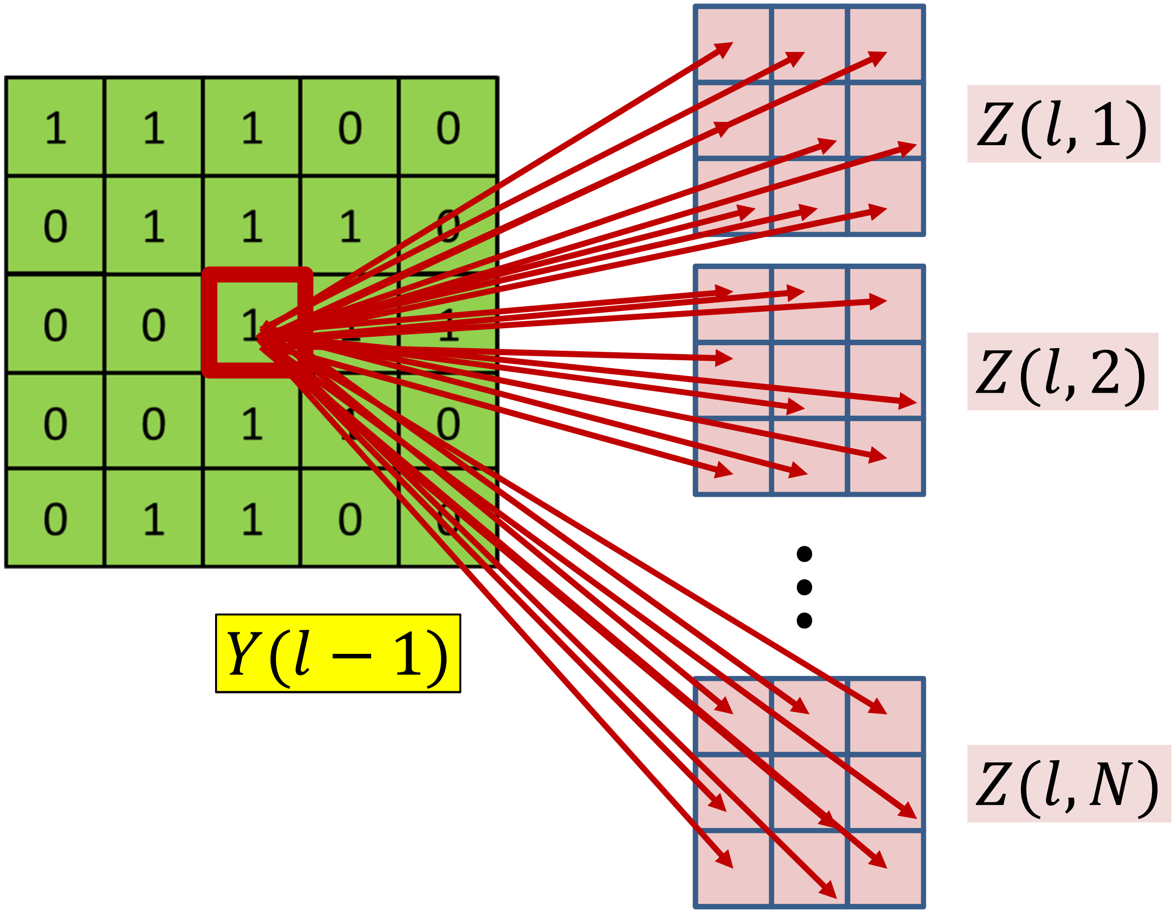
- For every l^{th} layer filter, each $y(l-1, m, x, y)$ affects several $z(l, m, x, y)$ terms

BP: Convolutional layer



- For every l^{th} layer filter, each $y(l-1,m,x,y)$ affects several $z(l,m,x,y)$ terms

BP: Convolutional layer



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

- Each $y(l-1,m,x,y)$ affects several $z(l,n,x,y)$ terms for every n
  - ***Affects terms in all l th layer Z maps***
  - ***All of them contribute to the derivative of the divergence w.r.t. $y(l-1,m,x,y)$***
  - ***All of them contribute to the derivatives w.r.t filter weights***

CNN: Forward

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

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

Backward layer *l*

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

Backward layer *l*

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

Multiple ways of recasting this as tensor/ vector operations.

Will not discuss here

Backward (no pooling)

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

Backward (no pooling)

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

Multiple ways of recasting this as tensor/ vector operations.

Will not discuss here

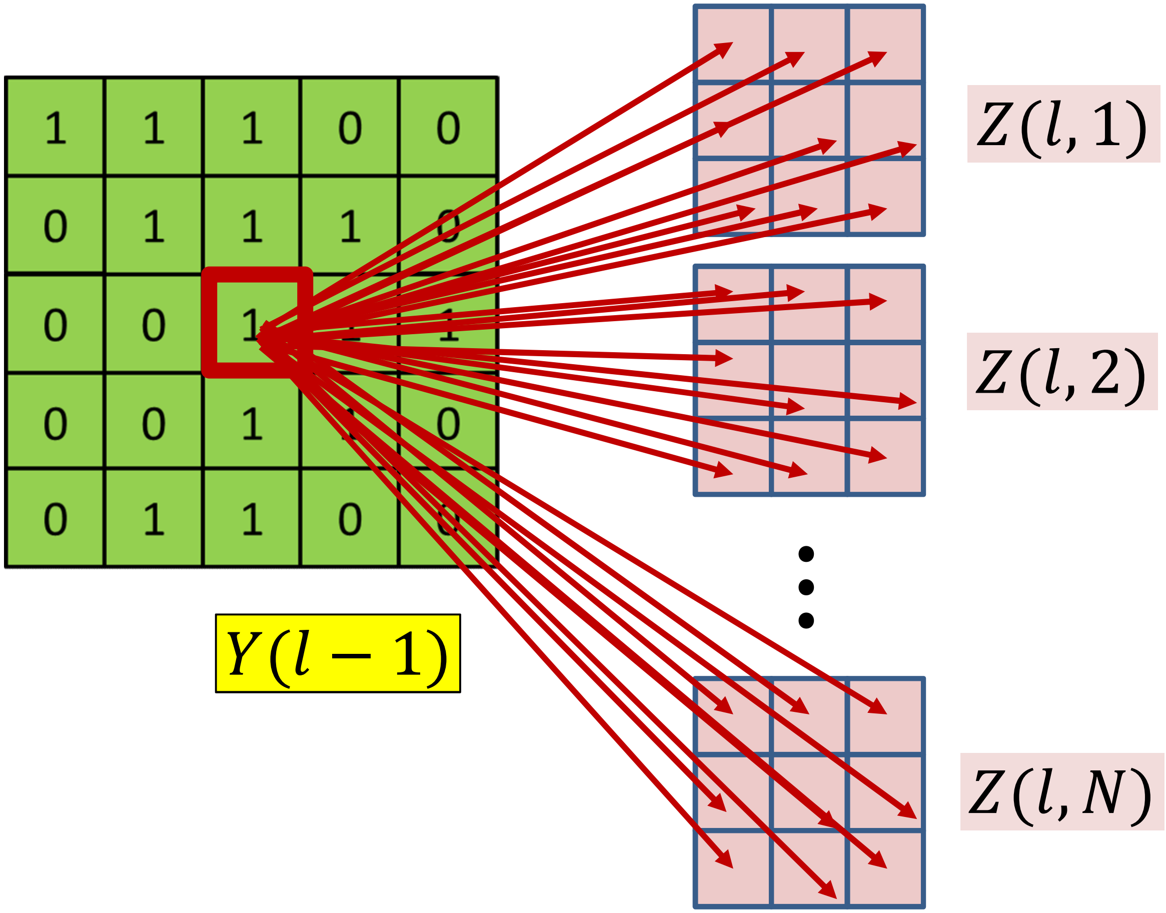
Backward (with strides)

```
dw(l) = zeros($D_l$x$D_{l-1}$x$K_l$x$K_l$)
dY(l-1) = zeros($D_{l-1}$x$W_{l-1}$x$H_{l-1}$)
for j = 1:$D_l$
   for x = 1:$W_l$
      m = (x-1)stride
      for y = 1:$H_l$
         n = (y-1)stride
         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'-1,n+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,m+x'-1,n+y'-1)
```

Backward (with strides)

```
dY(L) = dDiv/dY(L)
for l = L:1  # Backward through layers
    dw(l) = zeros($D_l$x$D_{l-1}$x$K_l$x$K_l$)
    dY(l-1) = zeros($D_{l-1}$x$W_{l-1}$x$H_{l-1}$)
    for j = 1:$D_l$
        for x = 1:stride:$W_l$
            m = (x-1)stride
            for y = 1:stride: $H_l$
                n = (y-1)stride
                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’)
```

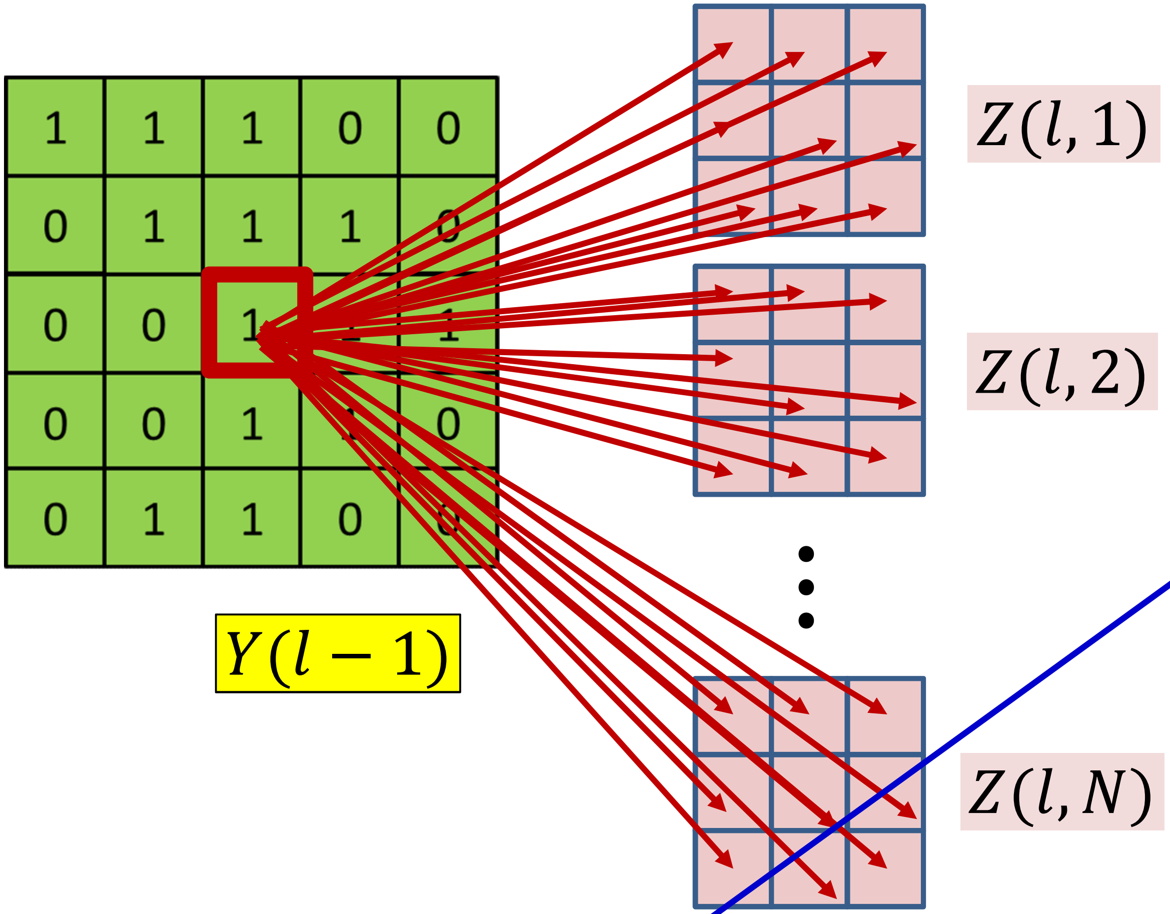
Derivative w.r.t *y*: the math



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

$$\frac{\partial Div}{\partial y(l-1,m,x,y)} = \sum_{n} \sum_{x',y'} w(l,n,m,x',y') \frac{\partial Div}{\partial z(l,n,x-x'+1,y-y'+1)}$$

Derivative w.r.t *y*

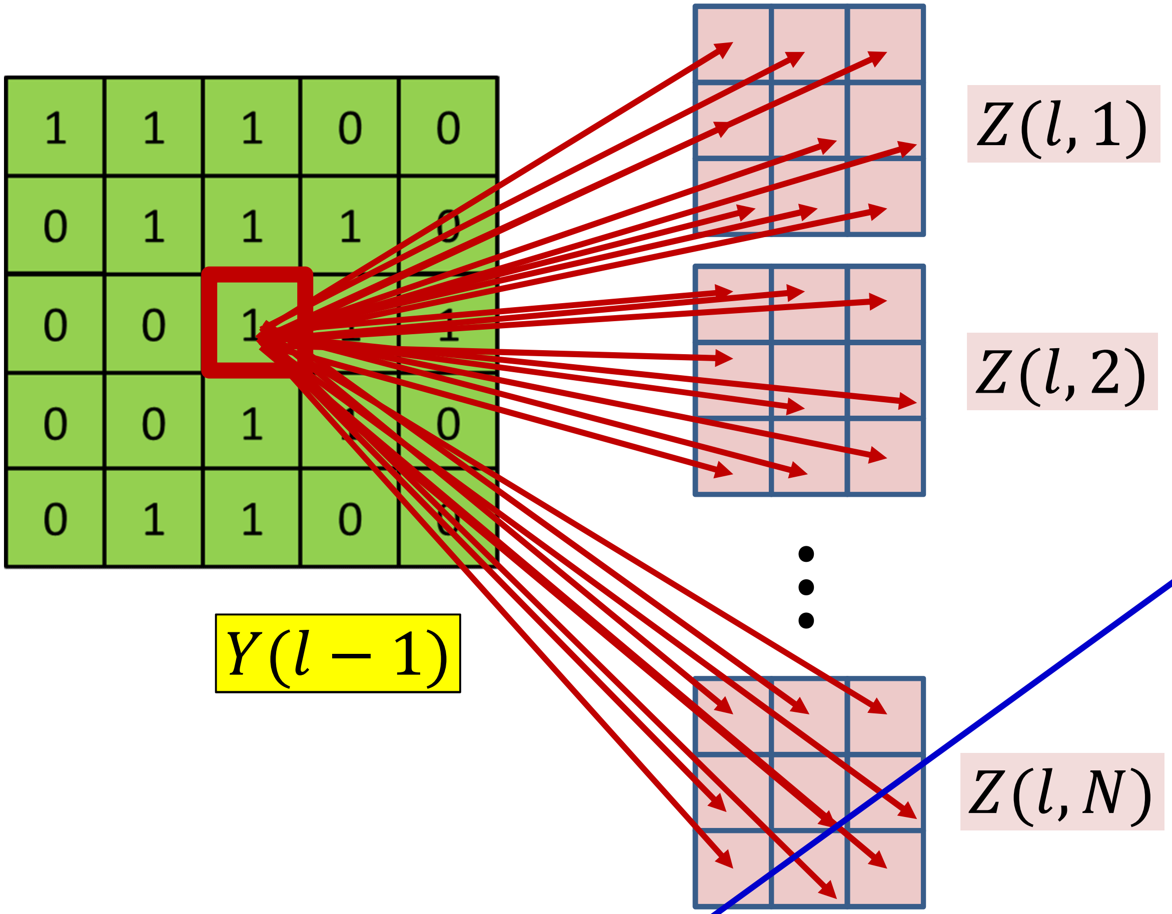


1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Summing over all Z maps

$$\frac{\partial Div}{\partial y(l-1,m,x,y)} = \sum_{n} \sum_{x',y'} w(l,n,m,x',y') \frac{\partial Div}{\partial z(l,n,x-x'+1,y-y'+1)}$$

Derivative w.r.t *y*

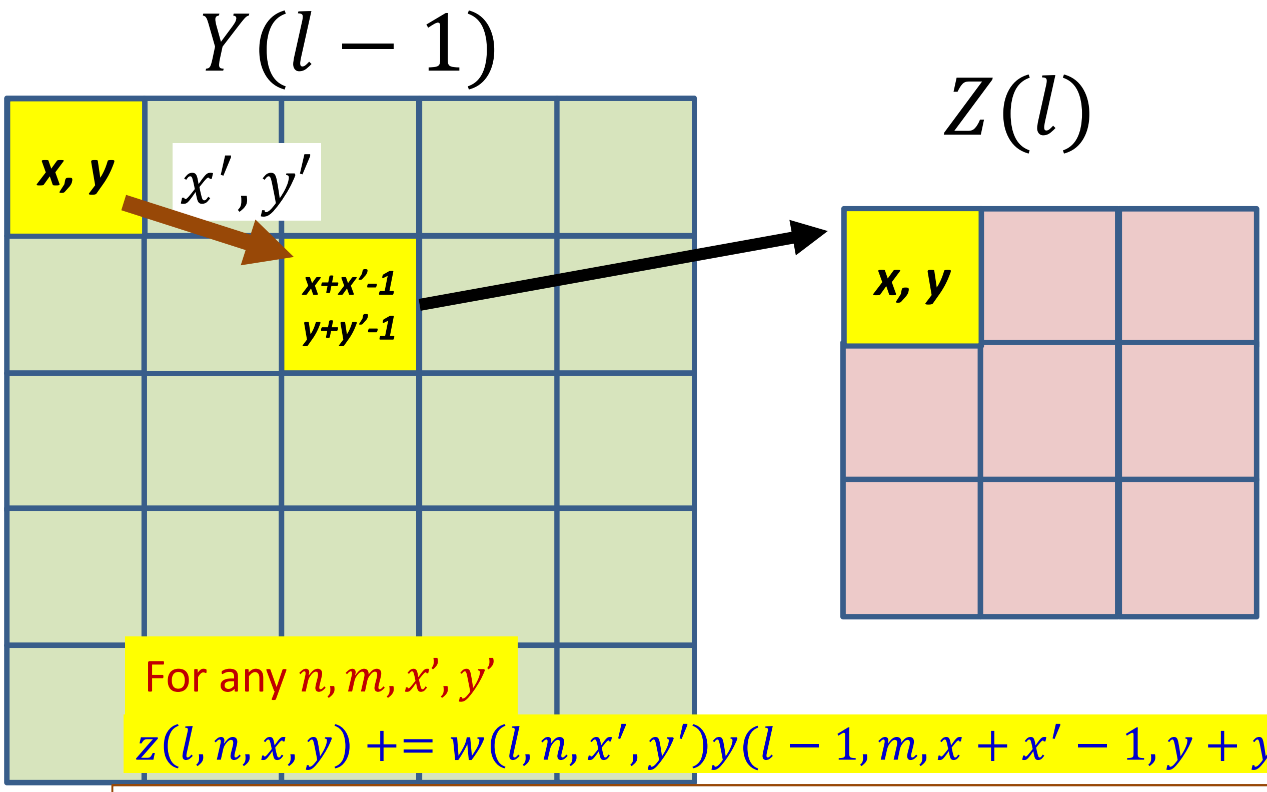


Summing over all Z maps

Why x-x'+1?

$$\frac{\partial Div}{\partial y(l-1,m,x,y)} = \sum_{n} \sum_{x',y'} w(l,n,m,x',y') \frac{\partial Div}{\partial z(l,n,x-x'+1,y-y'+1)}$$

Convolution Forward : why $x - x'$



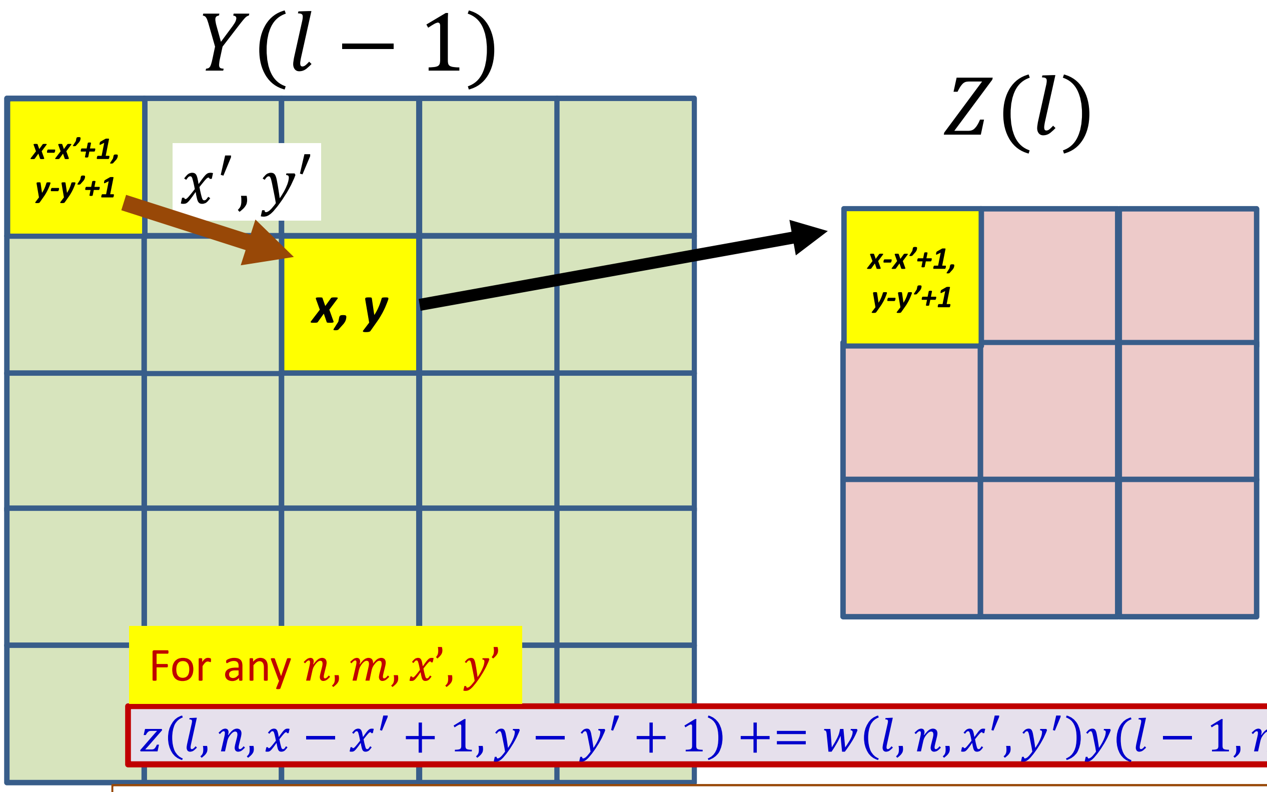
For any n, m, x', y'

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

$y(x+x', y+y')$ affects $z(x,y)$ through $w(x', y')$

The shift from (x,y) to $(x+x', y+y')$ = (x', y')

Convolution Forward : why $x - x'$



Note change in indices. Compare to previous slide (and figs to left)

For any n, m, x', y'

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

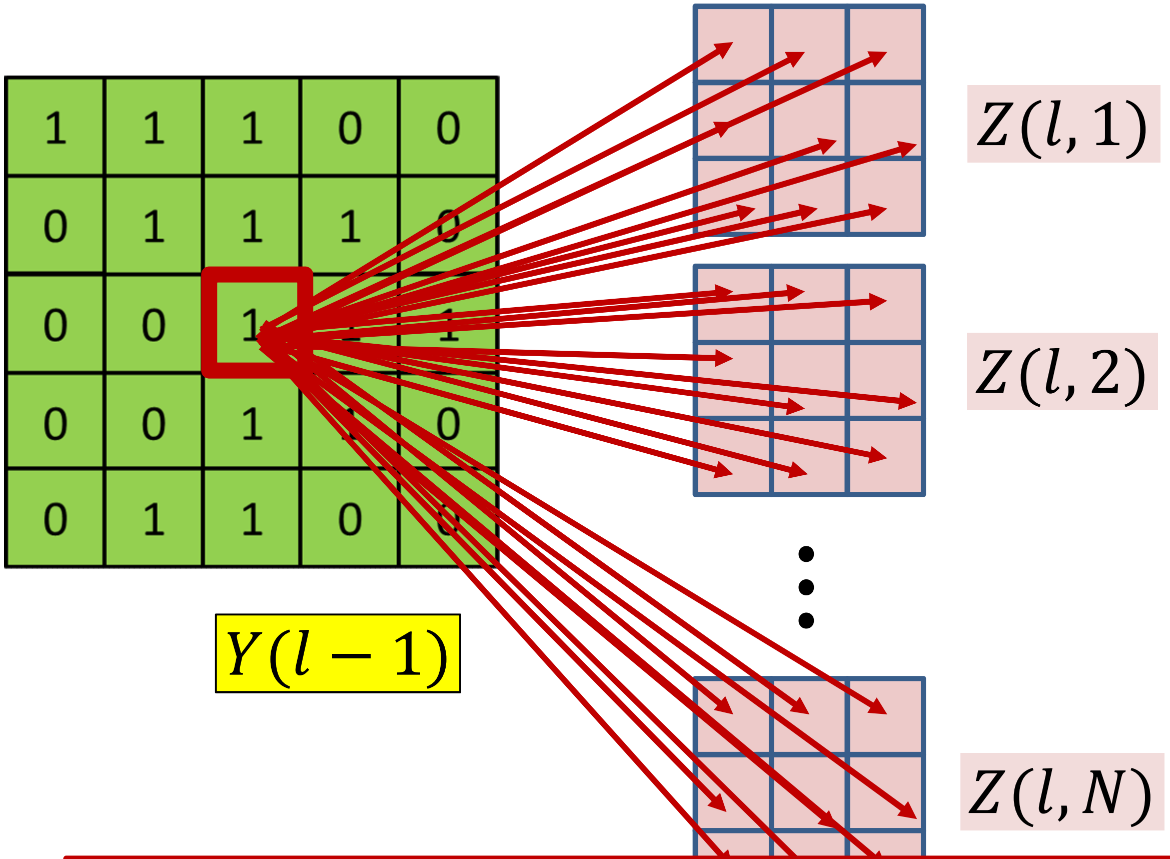
$y(x + x', y + y')$ affects $z(x, y)$ through $w(x', y')$

The shift from (x, y) to $(x + x', y + y')$ = (x', y')

$y(x, y)$ affects $z(x - x', y - y')$ through $w(x', y')$

The shift from $(x - x', y - y')$ to (x, y) = (x', y')

Derivative w.r.t *y*

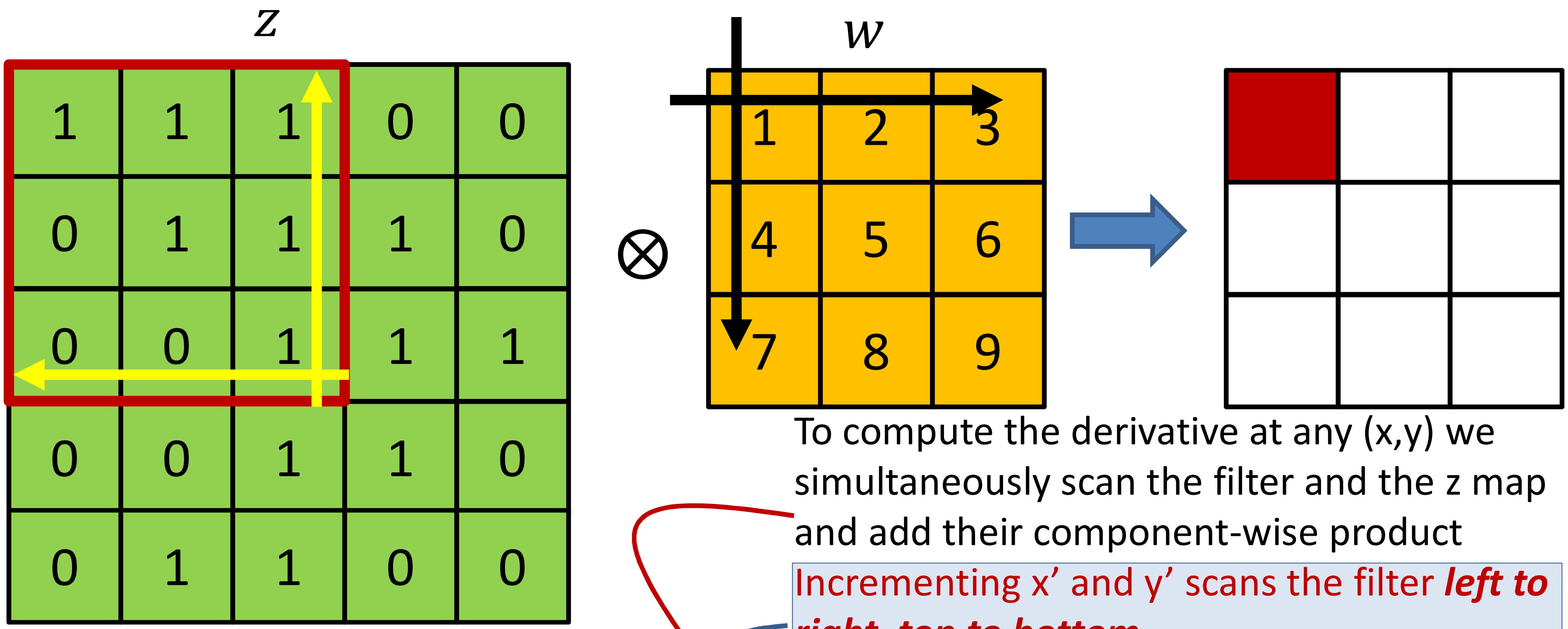


1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

$$z(l,n,x-x'+1,y-y'+1) \mathrel{+}= w(l,n,x',y')y(l-1,m,x,y) \quad \text{for all } n, (x',y')$$

$$\frac{\partial Div}{\partial y(l-1,m,x,y)} = \sum_{n} \sum_{x',y'} w(l,n,m,x',y') \frac{\partial Div}{\partial z(l,n,x-x'+1,y-y'+1)}$$

Derivative w.r.t *y*: the direction of the scan



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

1	2	3
4	5	6
7	8	9

To compute the derivative at any (x,y) we simultaneously scan the filter and the z map and add their component-wise product

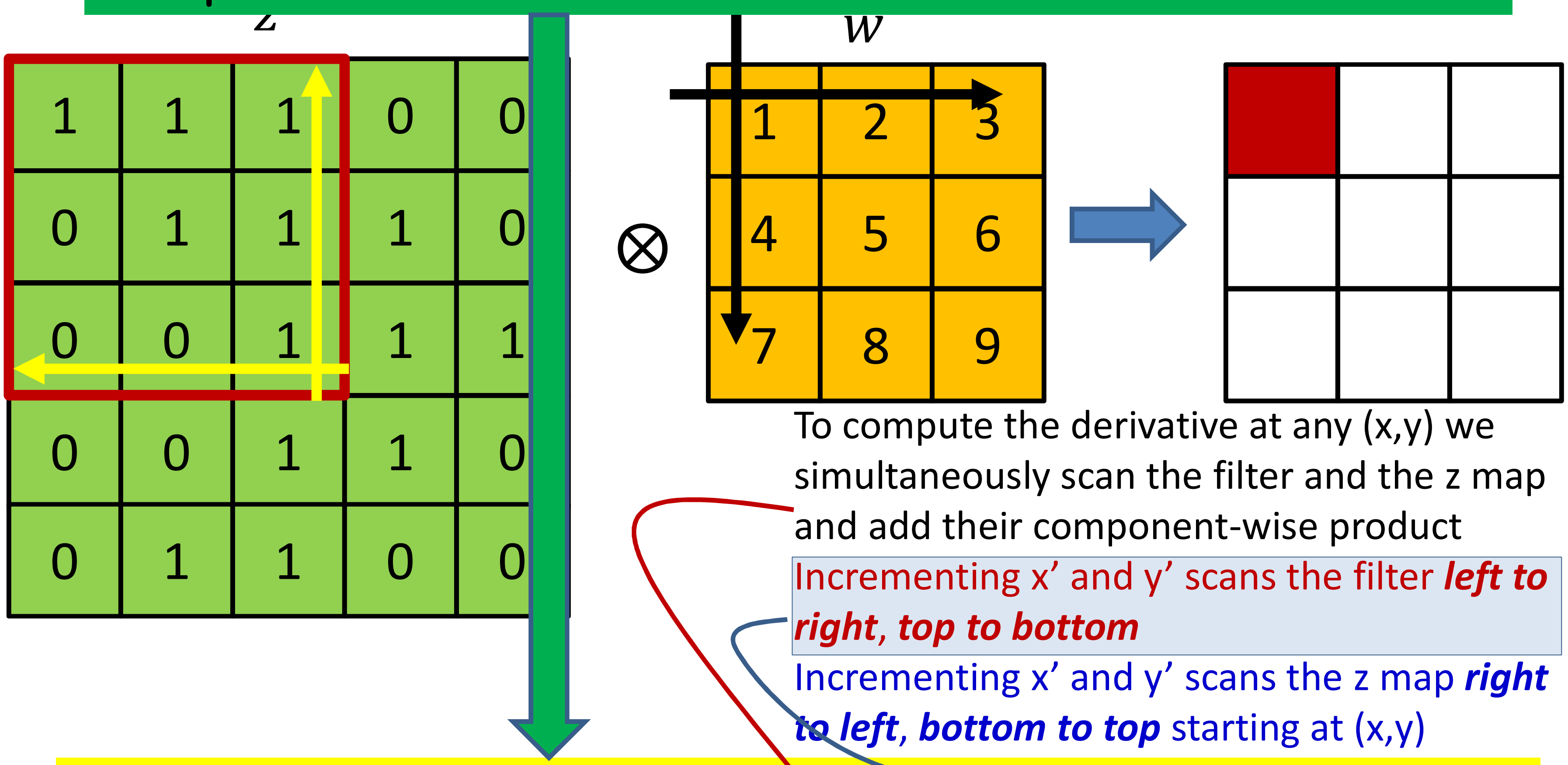
Incrementing x’ and y’ scans the filter ***left to right***, ***top to bottom***

Incrementing x’ and y’ scans the z map ***right to left***, ***bottom to top*** starting at (x,y)

$$\frac{\partial Div}{\partial y(l-1,m,x,y)} = \sum_{n} \sum_{x',y'} w(l,n,m,x',y') \frac{\partial Div}{\partial z(l,n,x-x'+1,y-y'+1)}$$

Derivative w.r.t *y*: the direction of the scan

Flipping the order of the scan will not change the computed derivative



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

1	2	3
4	5	6
7	8	9

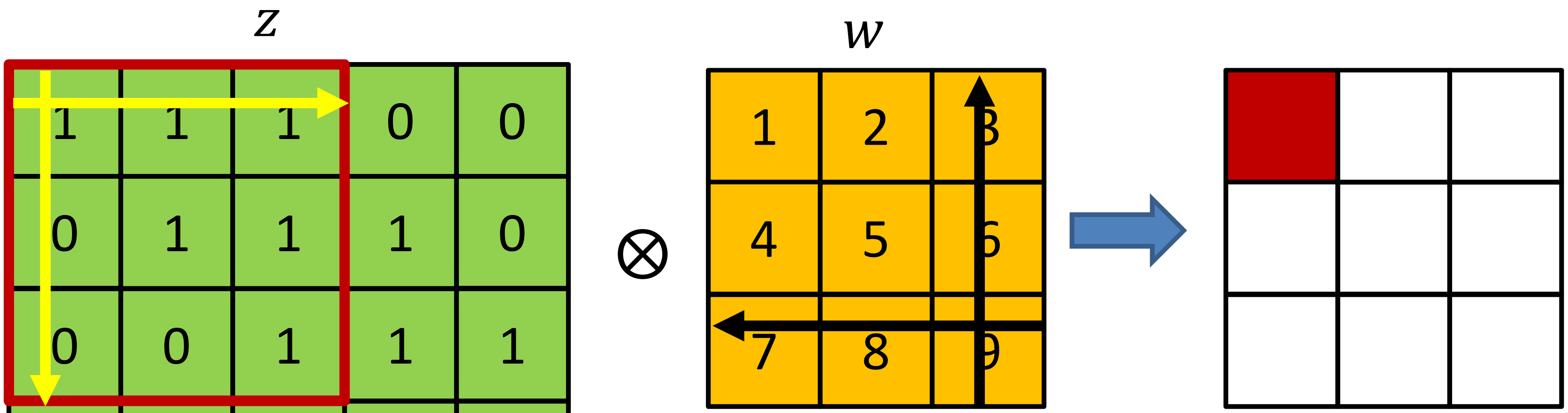
To compute the derivative at any (x,y) we simultaneously scan the filter and the z map and add their component-wise product

Incrementing x’ and y’ scans the filter ***left to right***, ***top to bottom***

Incrementing x’ and y’ scans the z map ***right to left***, ***bottom to top*** starting at (x,y)

$$\frac{\partial Div}{\partial y(l-1,m,x,y)} = \sum_{n} \sum_{x',y'} w(l,n,m,x',y') \frac{\partial Div}{\partial z(l,n,x-x'+1,y-y'+1)}$$

Derivative w.r.t *y*: the direction of the scan



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

1	2	3
4	5	6
7	8	9

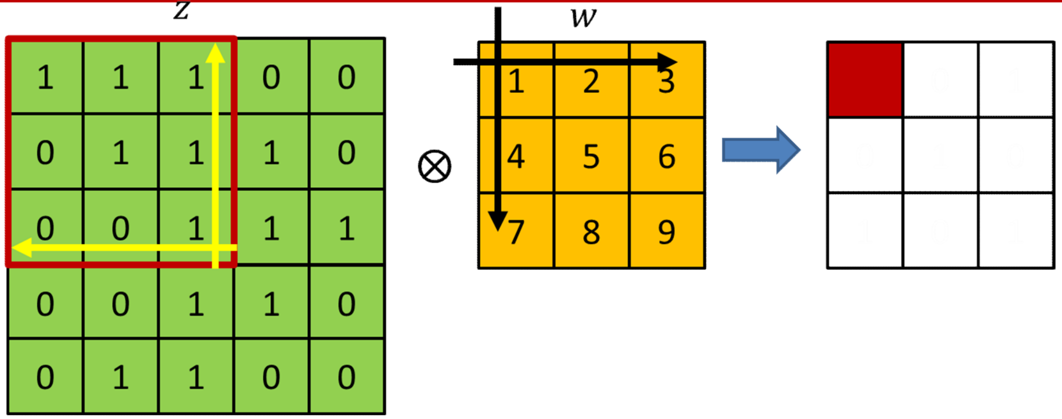
Reversed order of indexing to compute the derivative at any (x,y)

Incrementing x' and y' scans the filter ***right to left*, *bottom to top***

Incrementing x' and y' scans the z map ***left to right*, *top to bottom*** starting at (K,K) (K is the filter width)

$$\frac{\partial Div}{\partial y(l-1,m,x,y)} = \sum_{n} \sum_{x',y'} w(l,n,m,K-x'+1,K-y'+1) \frac{\partial Div}{\partial z(l,n,x+x'-K,y+y'-K)}$$

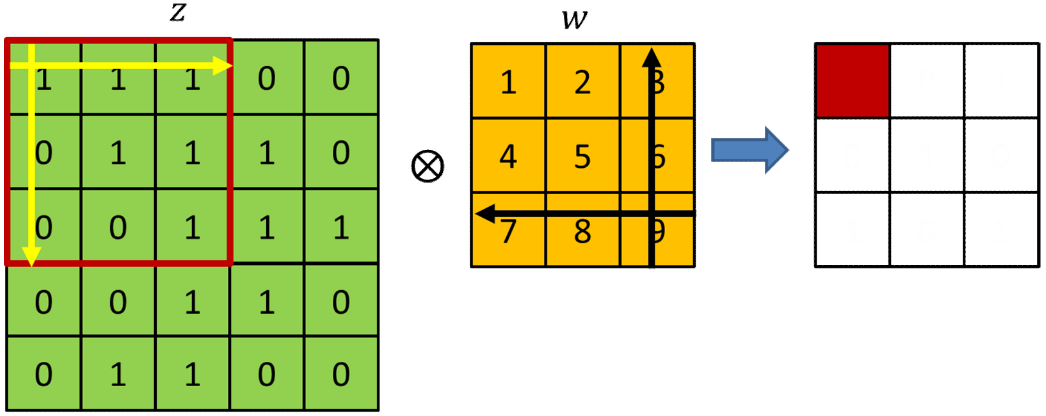
Comparing the two scans: Hint – output is identical



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

1	2	3
4	5	6
7	8	9

$$\frac{\partial Div}{\partial y(l-1,m,x,y)} = \sum_{n} \sum_{x',y'} w(l,n,m,x',y') \frac{\partial Div}{\partial z(l,n,x-x'+1,y-y'+1)}$$



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

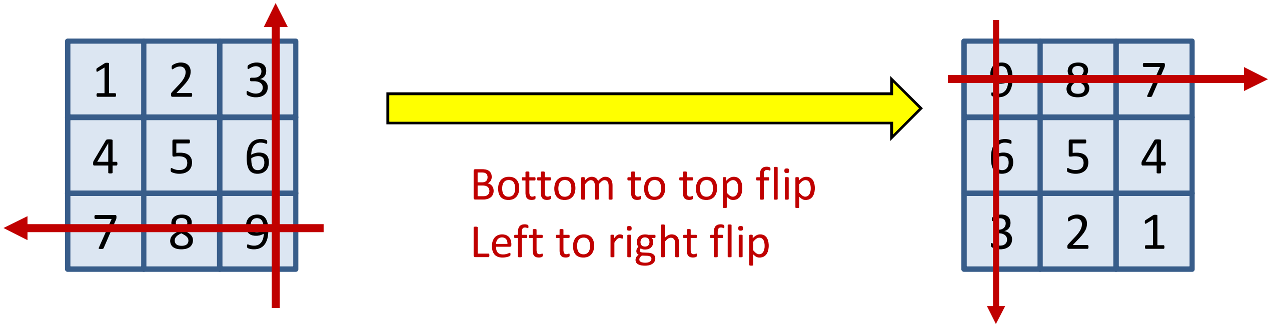
1	2	3
4	5	6
7	8	9

$$\frac{\partial Div}{\partial y(l-1,m,x,y)} = \sum_{n} \sum_{x',y'} w(l,n,m,K-x'+1,K-y'+1) \frac{\partial Div}{\partial z(l,n,x+x'-K,y+y'-K)}$$

Derivative w.r.t *y*

$w(l, n, m, x', y')$

$w(l, n, m, K - x' + 1, K - y' + 1)$



1	2	3
4	5	6
7	8	9

9	8	7
6	5	4
3	2	1

Bottom to top flip
Left to right flip

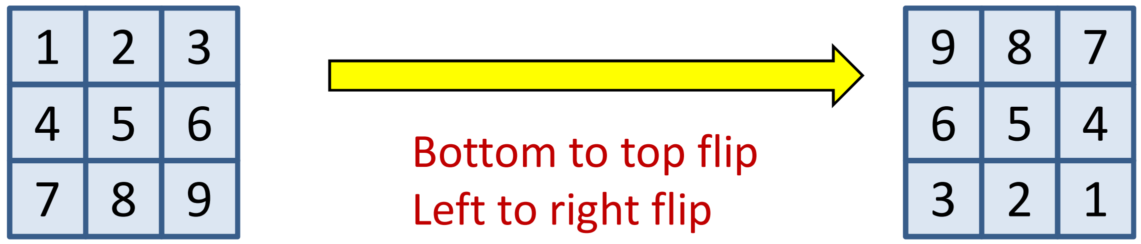
Scanning the filter right to left, bottom to top is the same as

Flipping the filter bottom to top, right to left
and then scanning left to right, top to bottom

$$\frac{\partial Div}{\partial y(l-1, m, x, y)} = \sum_{n} \sum_{x', y'} w(l, n, m, K - x' + 1, K - y' + 1) \frac{\partial Div}{\partial z(l, n, x + x' - K, y + y' - K)}$$

Derivative w.r.t *y*

$w(l, n, m, x', y')$ $\quad\quad$ $w(l, n, m, K - x' + 1, K - y' + 1)$



1	2	3
4	5	6
7	8	9

Bottom to top flip

Left to right flip

9	8	7
6	5	4
3	2	1

Define Flipping the fiter left-right and top-bottom

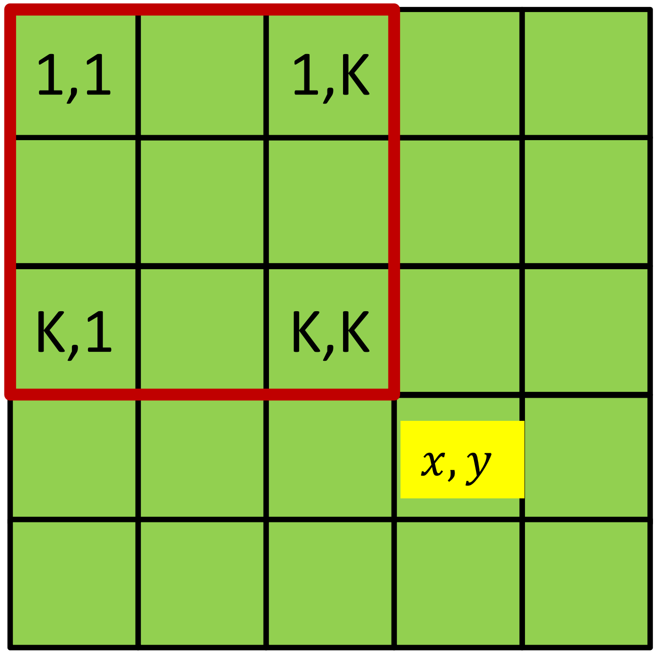
$$\widehat{w}(l, n, m, x', y') = w(l, n, m, K - x' + 1, K - y' + 1)$$

$$\frac{\partial Div}{\partial y(l-1, m, x, y)} = \sum_{n} \sum_{x', y'} w(l, n, m, K - x' + 1, K - y' + 1) \frac{\partial Div}{\partial z(l, n, x + x' - K, y + y' - K)}$$

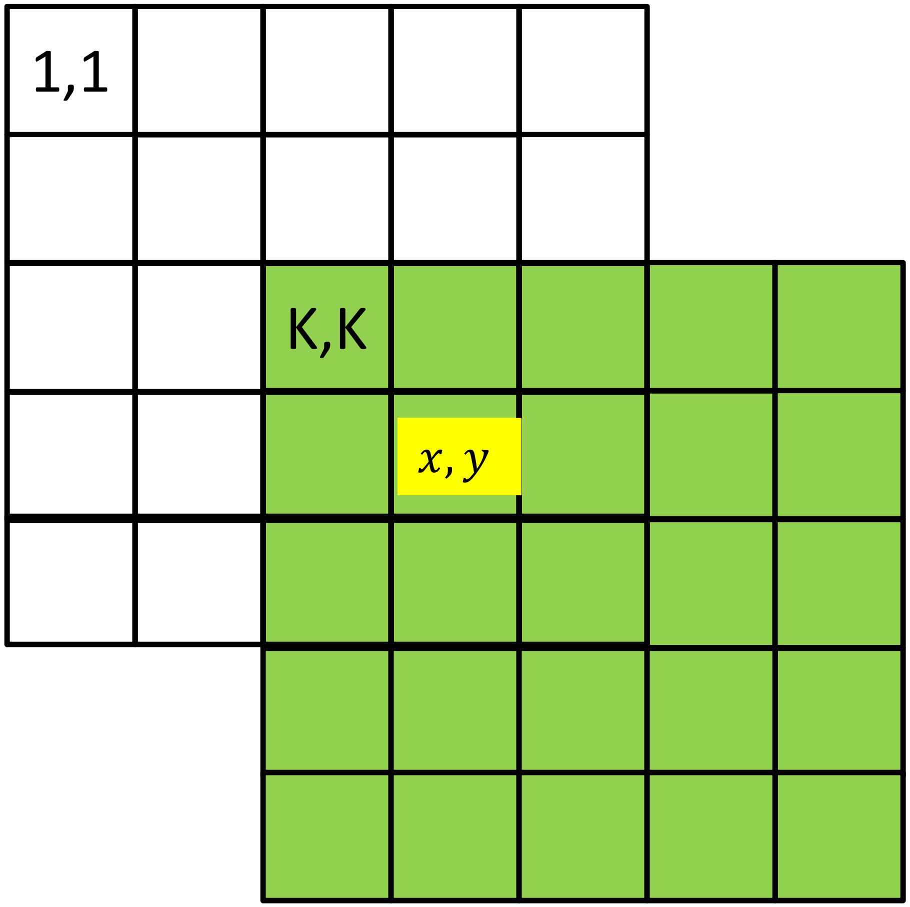
$$\frac{\partial Div}{\partial y(l-1, m, x, y)} = \sum_{n} \sum_{x', y'} \widehat{w}(l, n, m, x', y') \frac{\partial Div}{\partial z(l, n, x + x' - K, y + y' - K)}$$

Derivative w.r.t *y*

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



$z(l, n, x-(K-1), y-(K-1))$



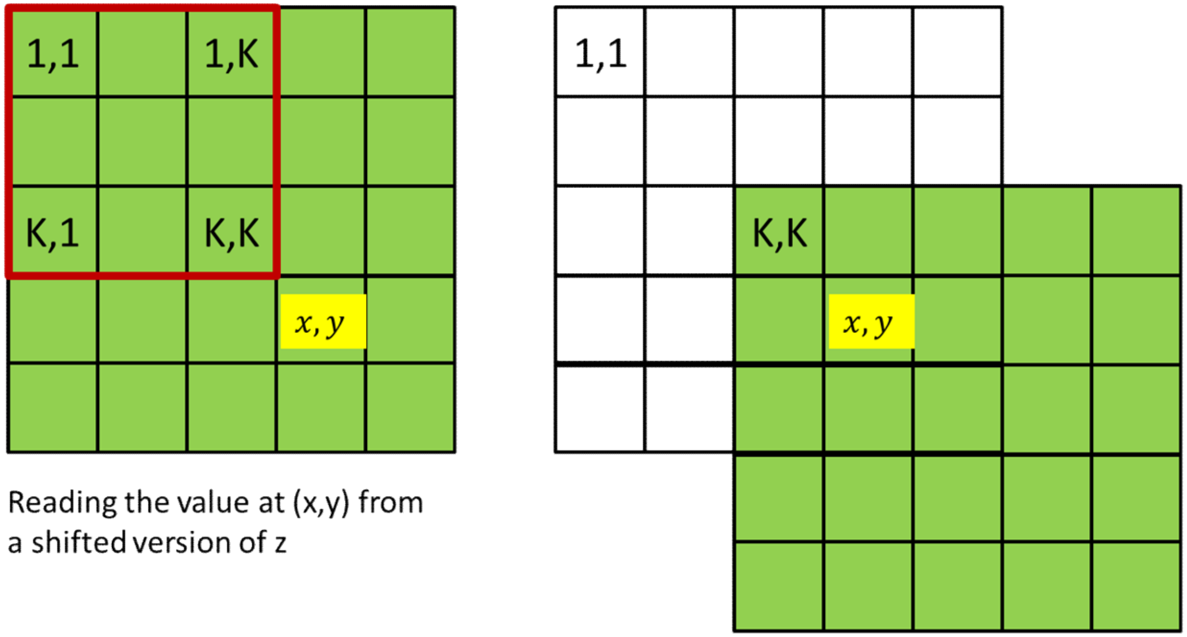
Reading the value at (x,y) from
a shifted version of z

$$\frac{\partial Div}{\partial y(l-1, m, x, y)} = \sum_{n} \sum_{x', y'} \hat{w}(l, n, m, x', y') \frac{\partial Div}{\partial z(l, n, x+x'-K, y+y'-K)}$$

Derivative w.r.t y

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

$z(l,n,x-(K-1),y-(K-1))$



Reading the value at (x,y) from a shifted version of z

$$\frac{\partial Div}{\partial y(l-1,m,x,y)} = \sum_{n}\sum_{x',y'} \widehat{w}(l,n,m,x',y') \frac{\partial Div}{\partial z(l,n,x+x'-K,y+y'-K)}$$

Shifting down and right by K-1, such that 1,1 becomes K,K

$$z_{shift}(l,n,m,x,y) = z(l,n,x-K+1,y-K+1)$$

$$\frac{\partial Div}{\partial y(l-1,m,x,y)} = \sum_{n}\sum_{x',y'} \widehat{w}(l,n,m,x',y') \frac{\partial Div}{\partial z_{shift}(l,n,x+x'-1,y+y'-1)}$$

Derivative w.r.t y

$$\frac{\partial Div}{\partial y(l-1,m,x,y)} = \sum_{n}\sum_{x',y'} w(l,n,m,K-x'+1,K-y'+1)\,\frac{\partial Div}{\partial z(l,n,x+x'-K,y+y'-K)}$$

Define

$$\hat{w}(l,n,m,x',y') = w(l,n,m,K-x'+1,K-y'+1)$$

$$z_{shift}(l,n,m,x,y) = z(l,n,x-K+1,y-K+1)$$

$$\frac{\partial Div}{\partial y(l-1,m,x,y)} = \sum_{n}\sum_{x',y'} \hat{w}(l,n,m,x',y')\,\frac{\partial Div}{\partial z_{shift}(l,n,x+x'-1,y+y'-1)}$$

Derivative w.r.t *y*

Define

$$\hat{w}(l,n,m,x',y') = w(l,n,m,K-x'+1,K-y'+1)$$

$$z_{shift}(l,n,m,x,y) = z(l,n,x-K+1,y-K+1)$$

Regular convolution running on shifted derivative maps using flipped filter

$$\frac{\partial Div}{\partial y(l-1,m,x,y)} = \sum_{n} \sum_{x',y'} \hat{w}(l,n,m,x',y') \frac{\partial Div}{\partial z_{shift}(l,n,x+x'-1,y+y'-1)}$$

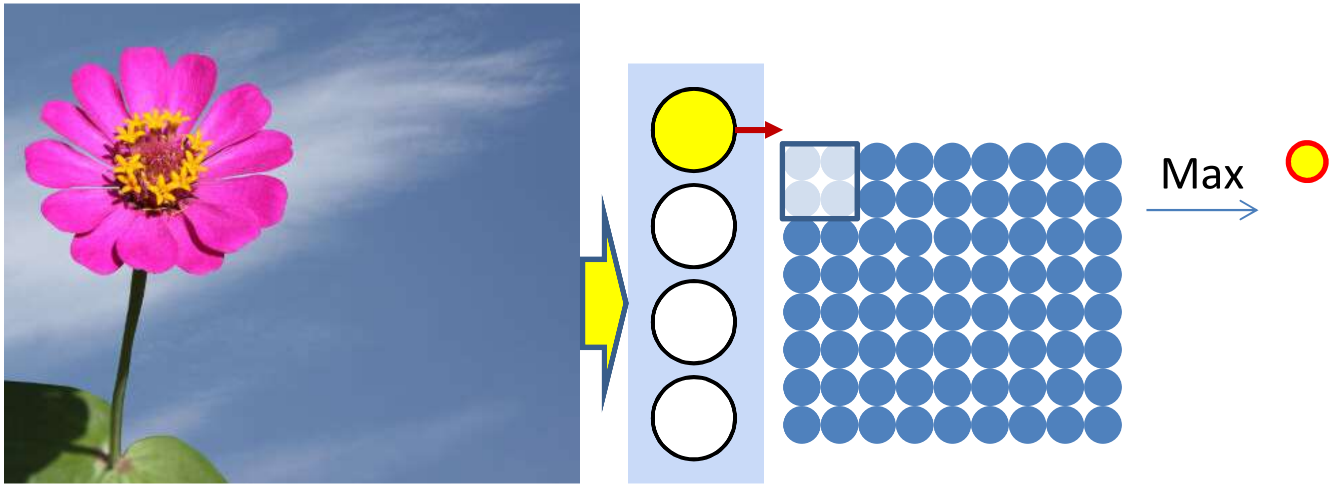
Derivatives for a single layer *l*: Vector notation

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

dzshift = zeros(D_lx(H_l+K_l-1)x(W_l+K_l-1)) #pad for -ve indices
for j = 1:D_l
    Wflip(j,:,:) = flipLeftRight(flipUpDown(W(l,j,:,:)))
    dzshift(j,K_l:end,K_l:end) = dz(l,j,:,:) # move idx 1->K_l
end
```

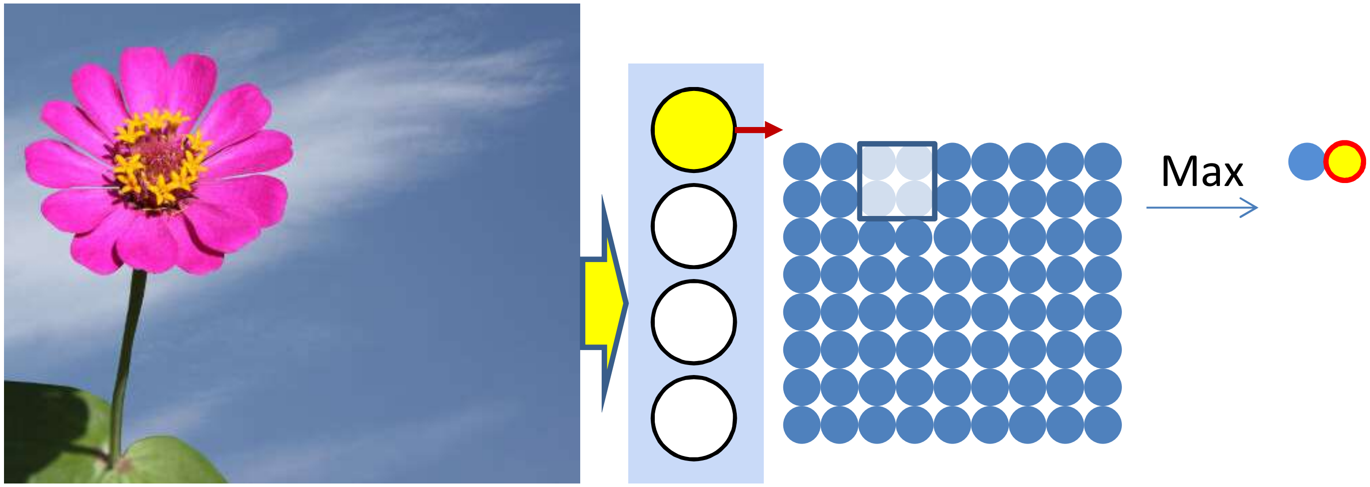
```
for j = 1:D_l
  for x = 1:W_{l-1}
    for y = 1:H_{l-1}
      segment = dzshift(:, x:x+K_l-1, y:y+K_l-1) #3D tensor
      dy(l-1,j,x,y) = Wflip.segment #tensor inner prod.
```

Pooling and downsampling



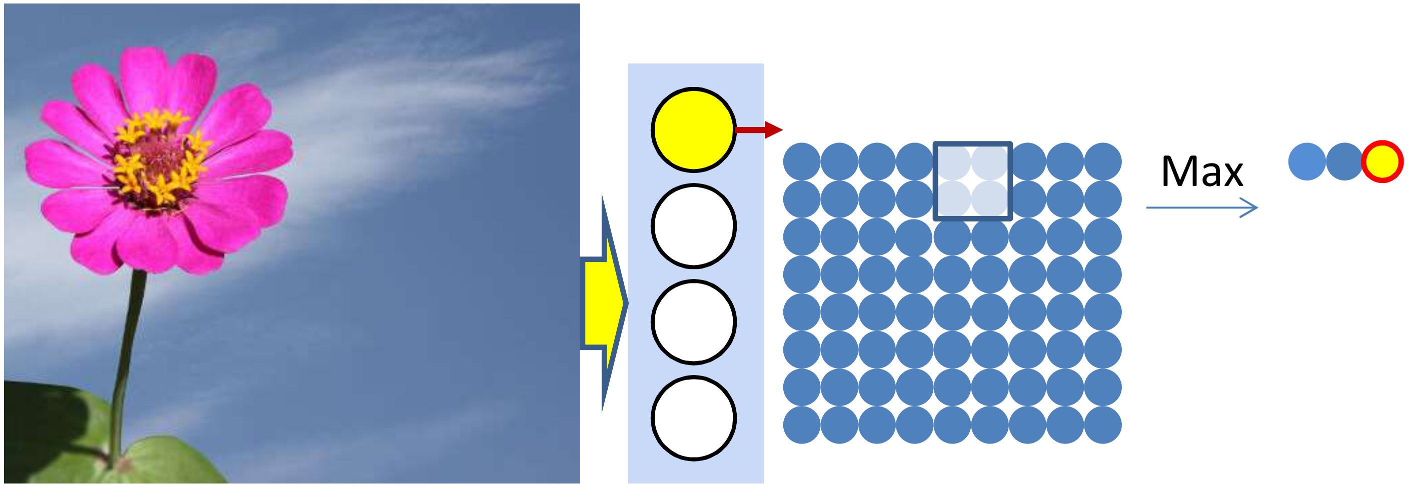
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



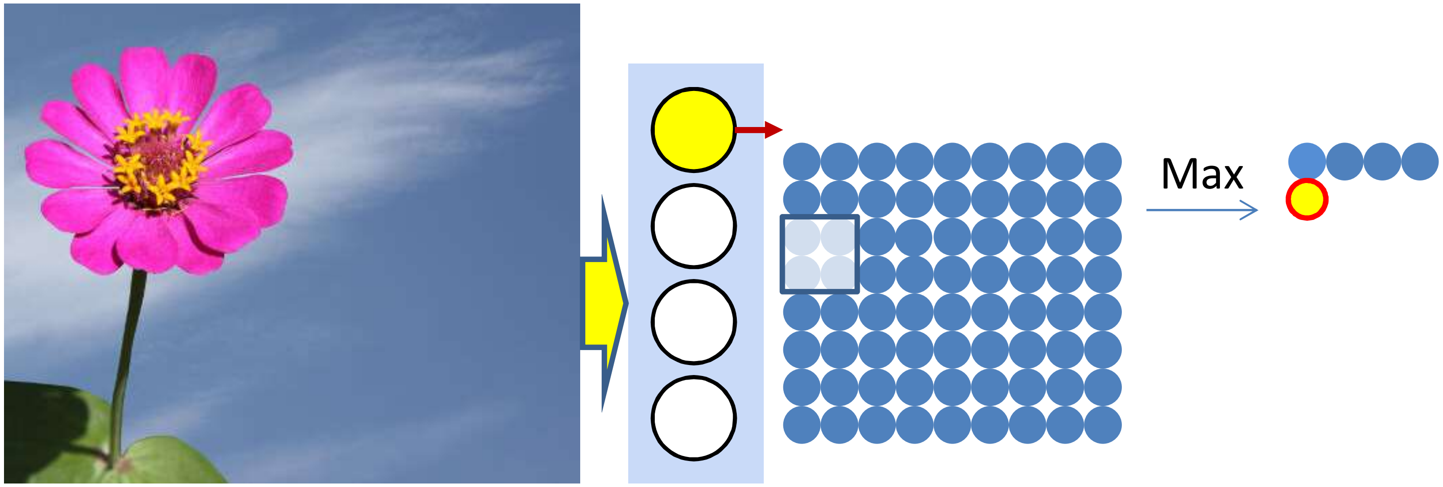
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



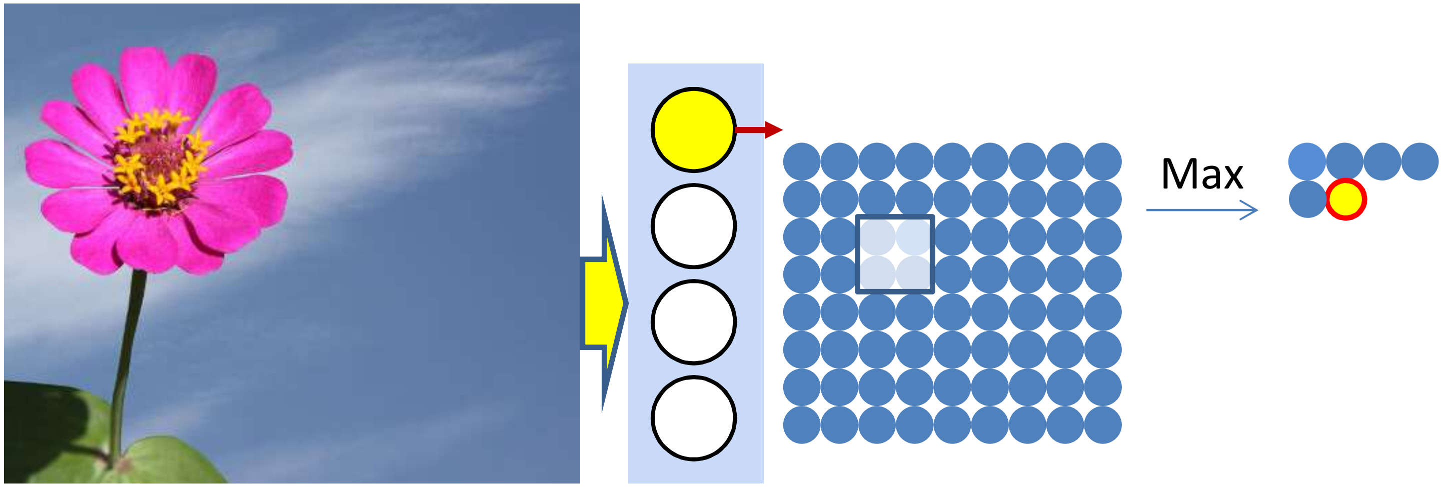
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



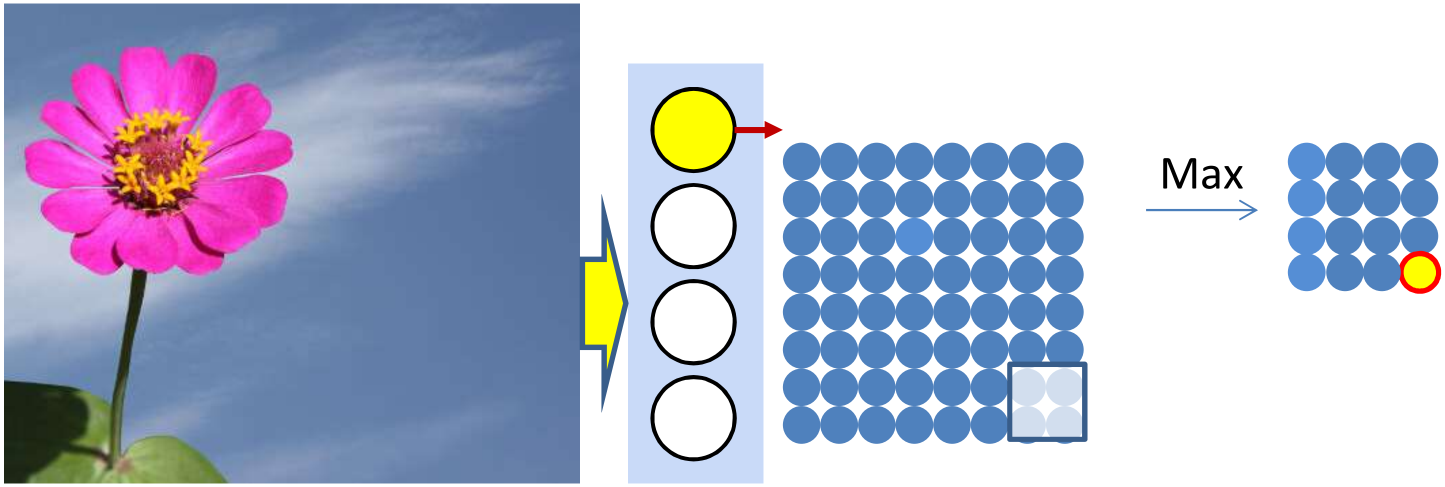
- Pooling is typically performed with strides > 1
  - – Results in shrinking of the map
  - – “Downsampling”

Pooling and downsampling



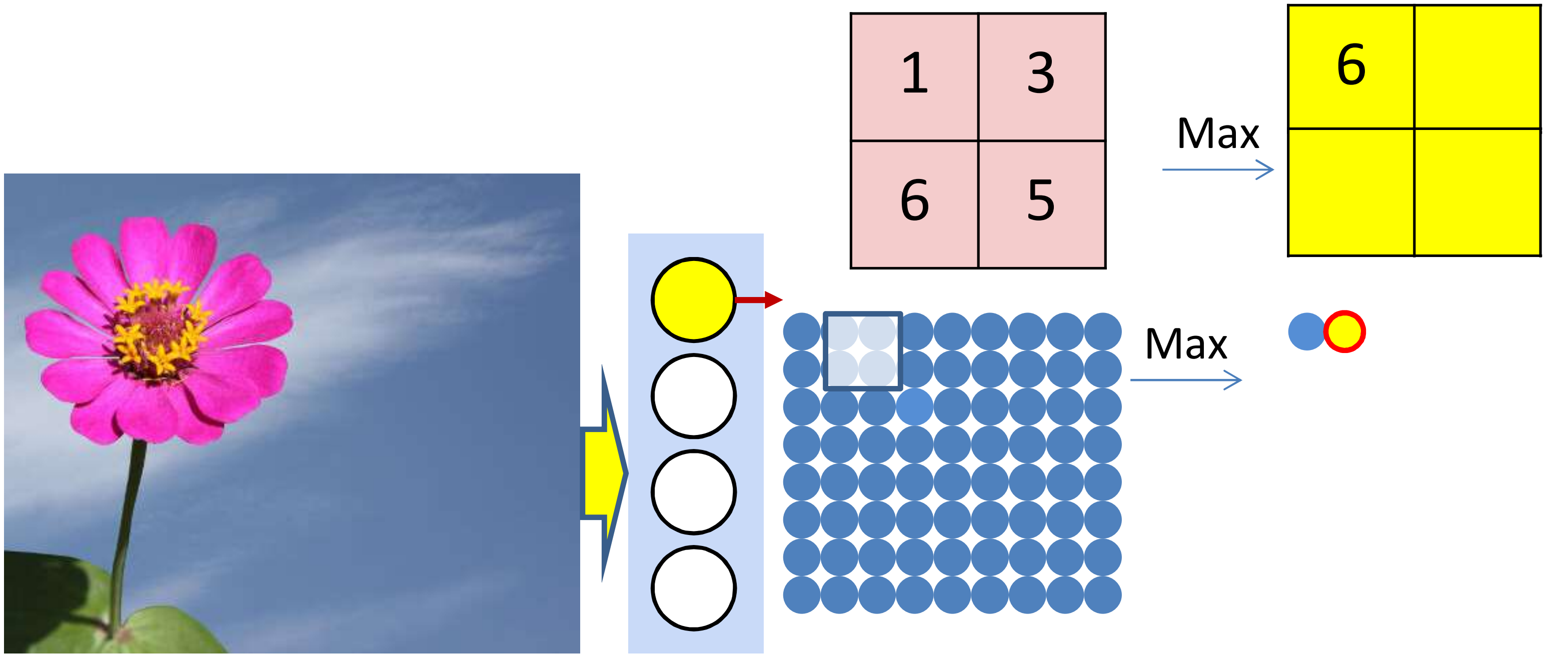
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



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

Max pooling

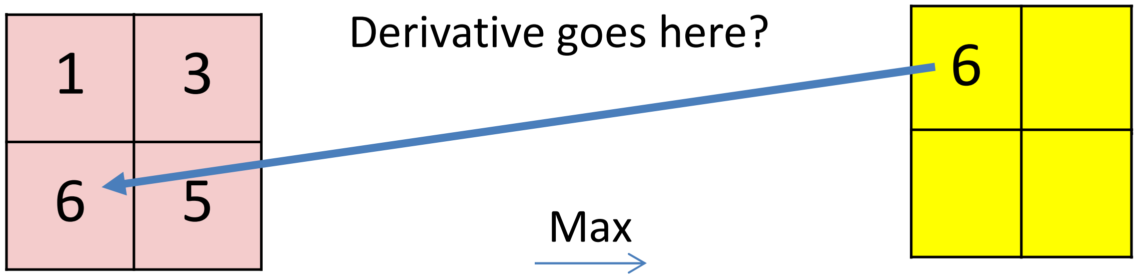


- Max pooling selects the largest from a pool of elements
- Pooling is performed by "scanning" the input

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

$$U(l,m,i,j) = Y(l,m,P(l,m,i,j))$$

Derivative of Max pooling



$$\boxed{\frac{dDiv}{dY(l,m,k,l)} = \begin{cases} \dfrac{dDiv}{dU(l,m,i,j)} & if\ (k,l) = P(l,m,i,j) \\ & 0\ otherwise \end{cases}}$$

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

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

$$U(l,m,i,j) = Y(l,m,P(l,m,i,j))$$

Max Pooling layer at layer *l*

a) Performed separately for every map (j).

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

b) Keeping track of location of max

Max pooling

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

Derivative of max pooling layer at layer *l*

a) Performed separately for every map (j).

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

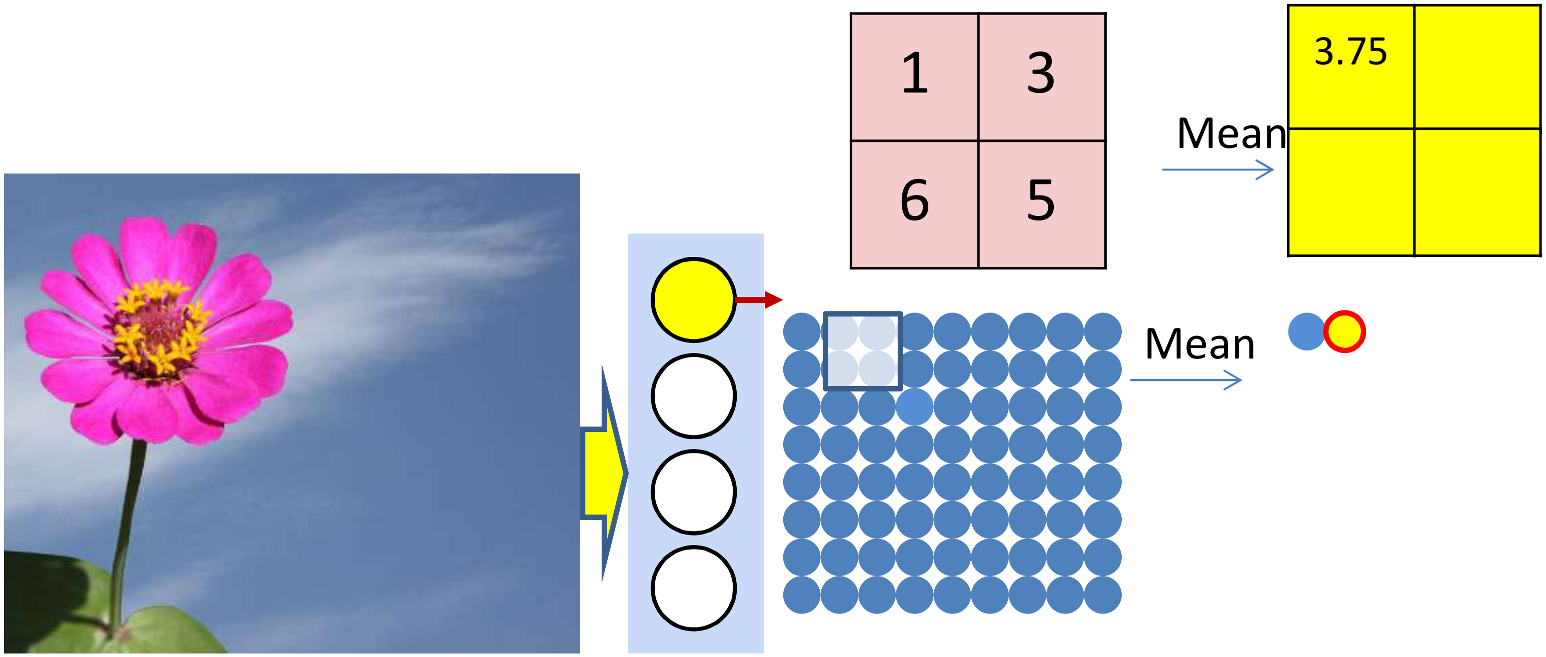
b) Keeping track of location of max

Max pooling

```
dy(:,:,:) = zeros($D_l$ x $W_l$ x $H_l$)
for j = 1:$D_l$
    for x = 1:$W_{l\_downsampled}$
        for y = 1:$H_{l\_downsampled}$
            dy(l,j,pidx(l,j,x,y)) += u(l,j,x,y)
```

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

Mean pooling



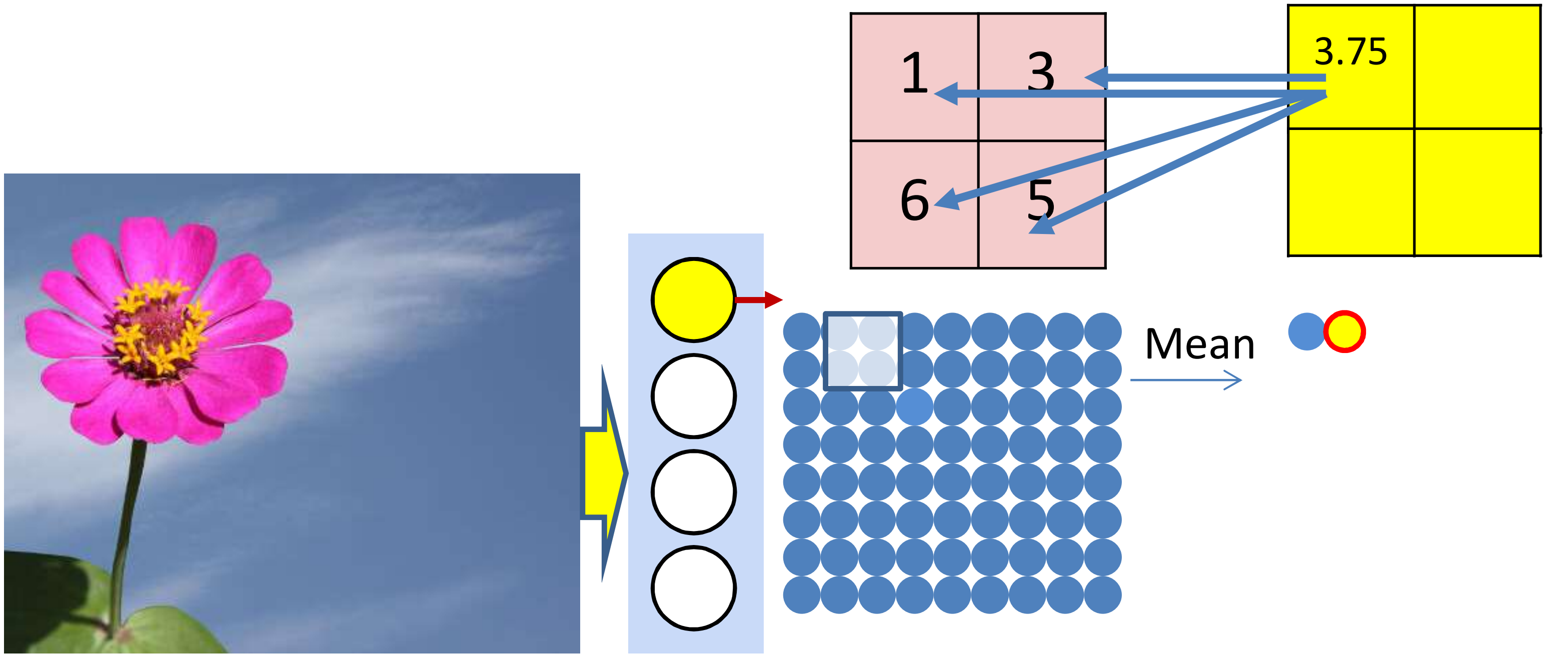
1	3
6	5

3.75	

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

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

Derivative of mean pooling



- The derivative of mean pooling is distributed over the pool

$$\begin{matrix} k \in \left\{(i-1)d+1, (i-1)d+K_{lpool}\right\}, \\ n \in \left\{(j-1)d+1, (j-1)d+K_{lpool}\right\} \end{matrix} \; dy(l,m,k,n) = \frac{1}{K_{lpool}^{2}} du(l,m,k,n)$$

Mean 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

Mean pooling

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

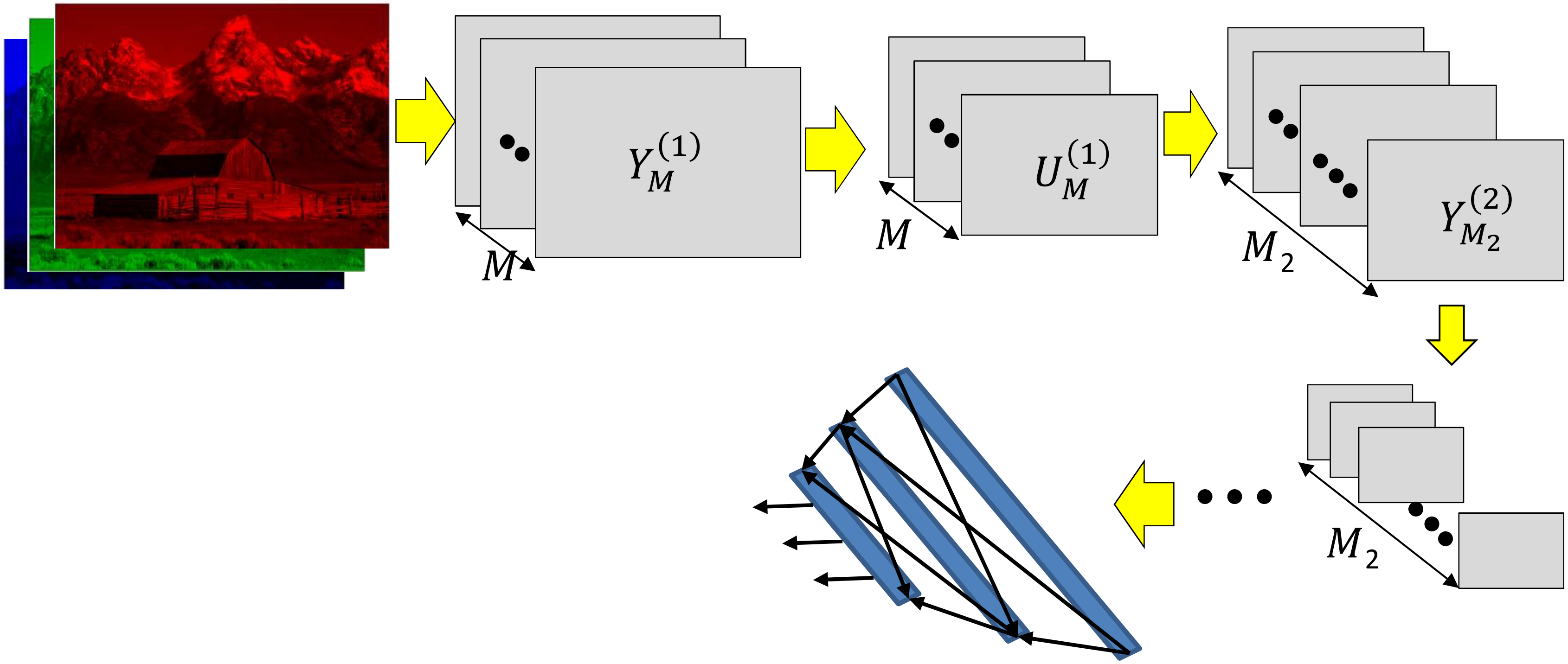
Derivative of mean pooling layer at layer *l*

Mean pooling

```
dy(:,:,:) = zeros($D_l$ x $W_l$ x $H_l$)
for j = 1:$D_l$
   m = 1
   for x = 1:$W_{l\_downsampled}$
      n = (x-1)stride
      for y = 1:$H_{l\_downsampled}$
         m = (y-1)stride
         for i = 1:$K_{lpool}$
            for j = 1:$K_{lpool}$
               dy(l,j,pidx(l,j,n+i,m+j)) +=
                                        (1/$K^2_{lpool}$)u(l,j,x,y)
```

“+=“ because adjacent windows may overlap

Learning the network



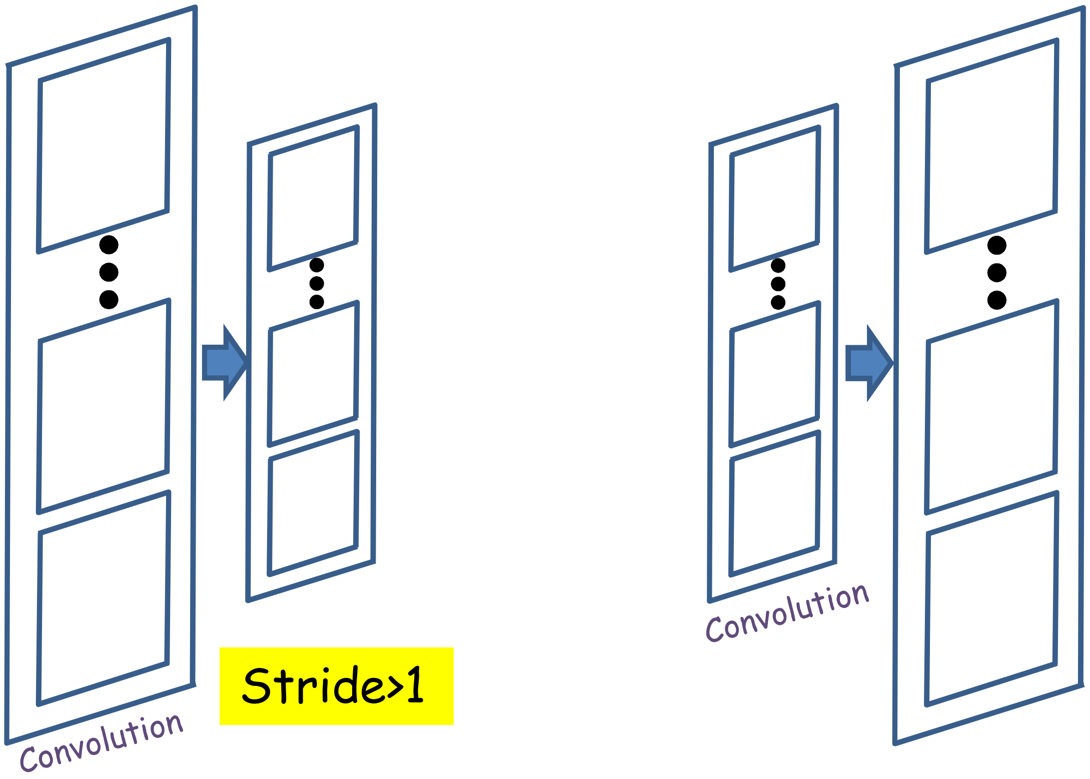
- Have shown the derivative of divergence w.r.t every intermediate output, and every free parameter (filter weights)
- Can now be embedded in gradient descent framework to learn the network

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

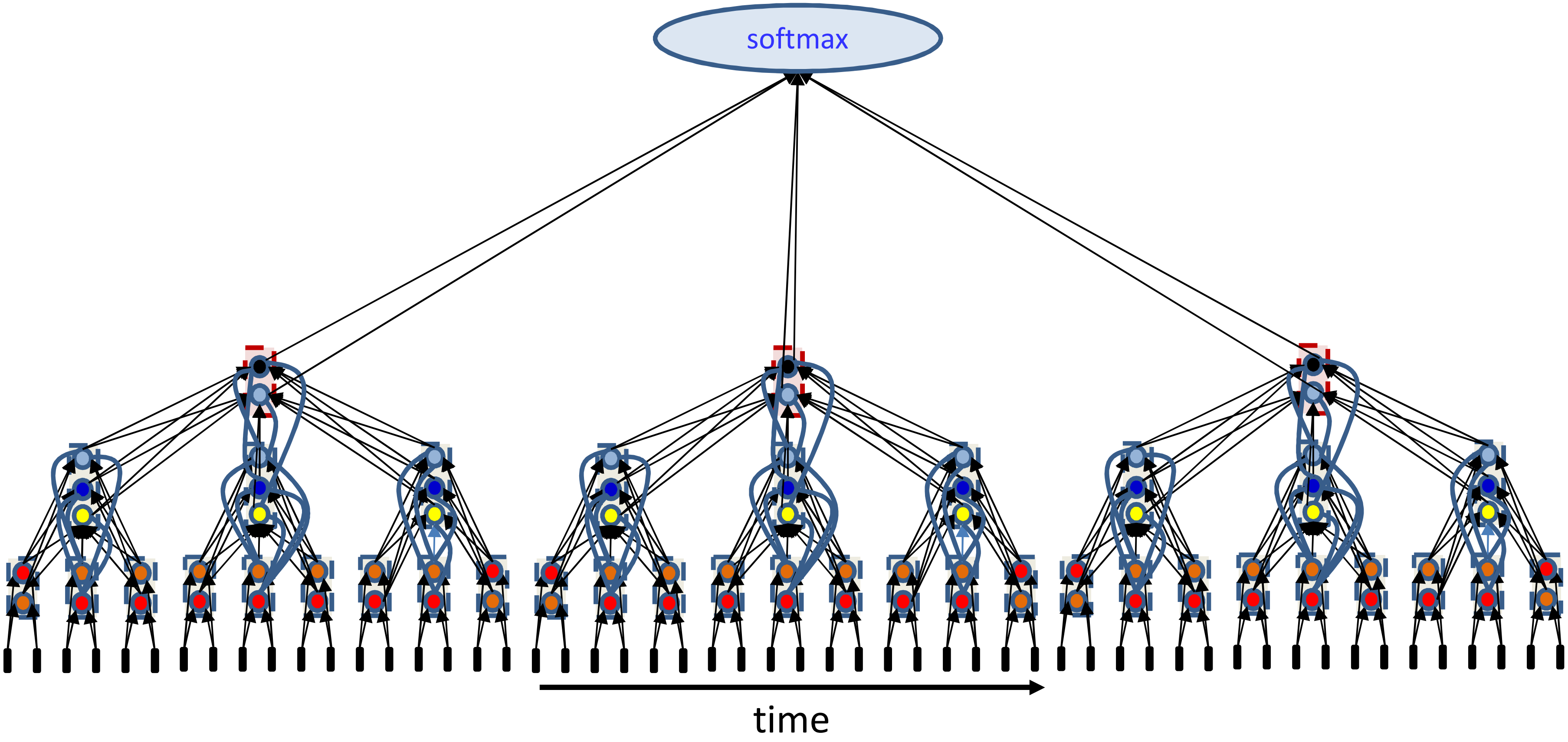
- The parameters of the network can be learned through regular back propagation
  - Maxpooling layers must propagate derivatives only over the maximum element in each pool
    - Other pooling operators can use regular gradients or subgradients
  - Derivatives must sum over appropriate sets of elements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

An implicit assumption



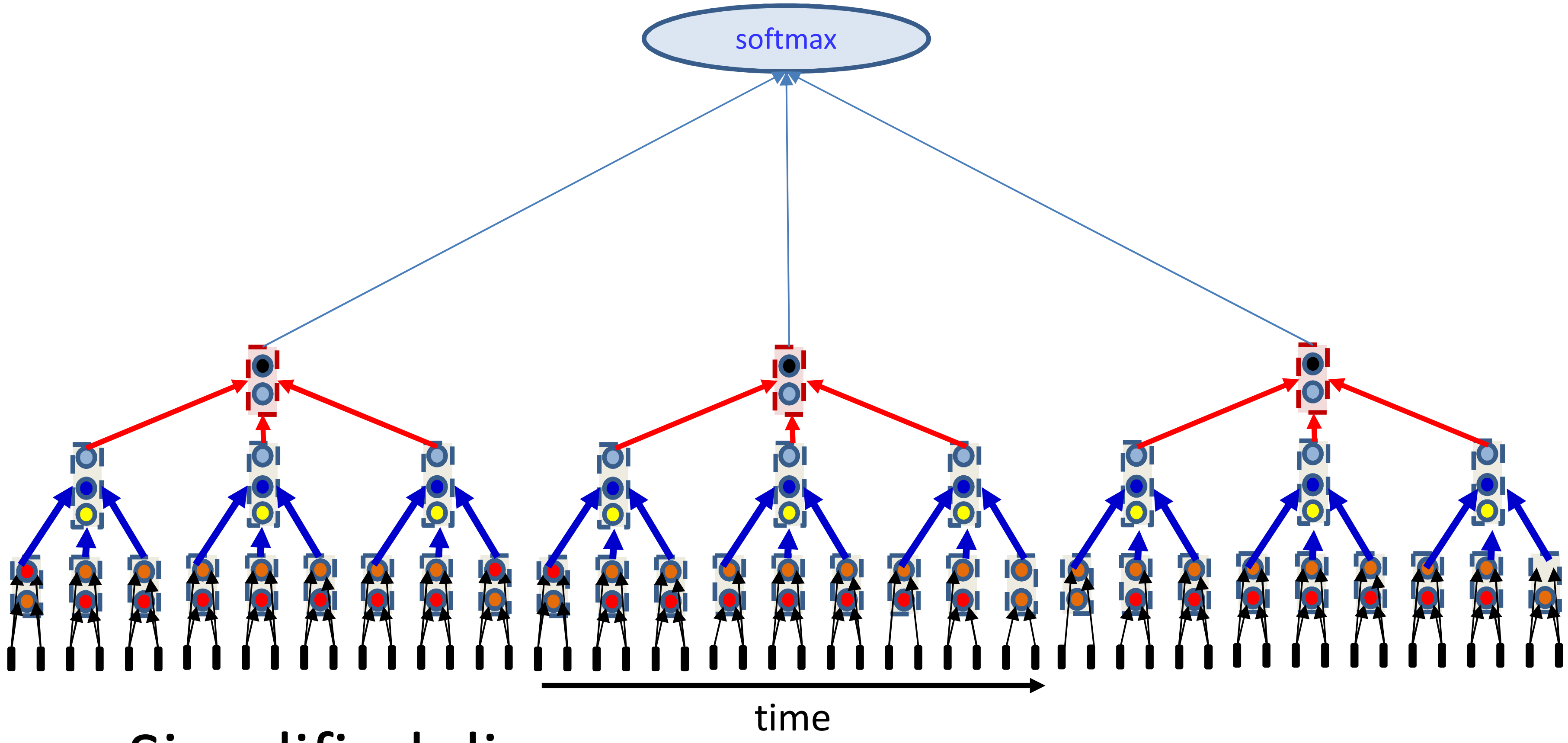
- We’ve always assumed that subsequent steps *shrink* the size of the maps
- Can subsequent maps *increase* in size

Recall this 1-D figure



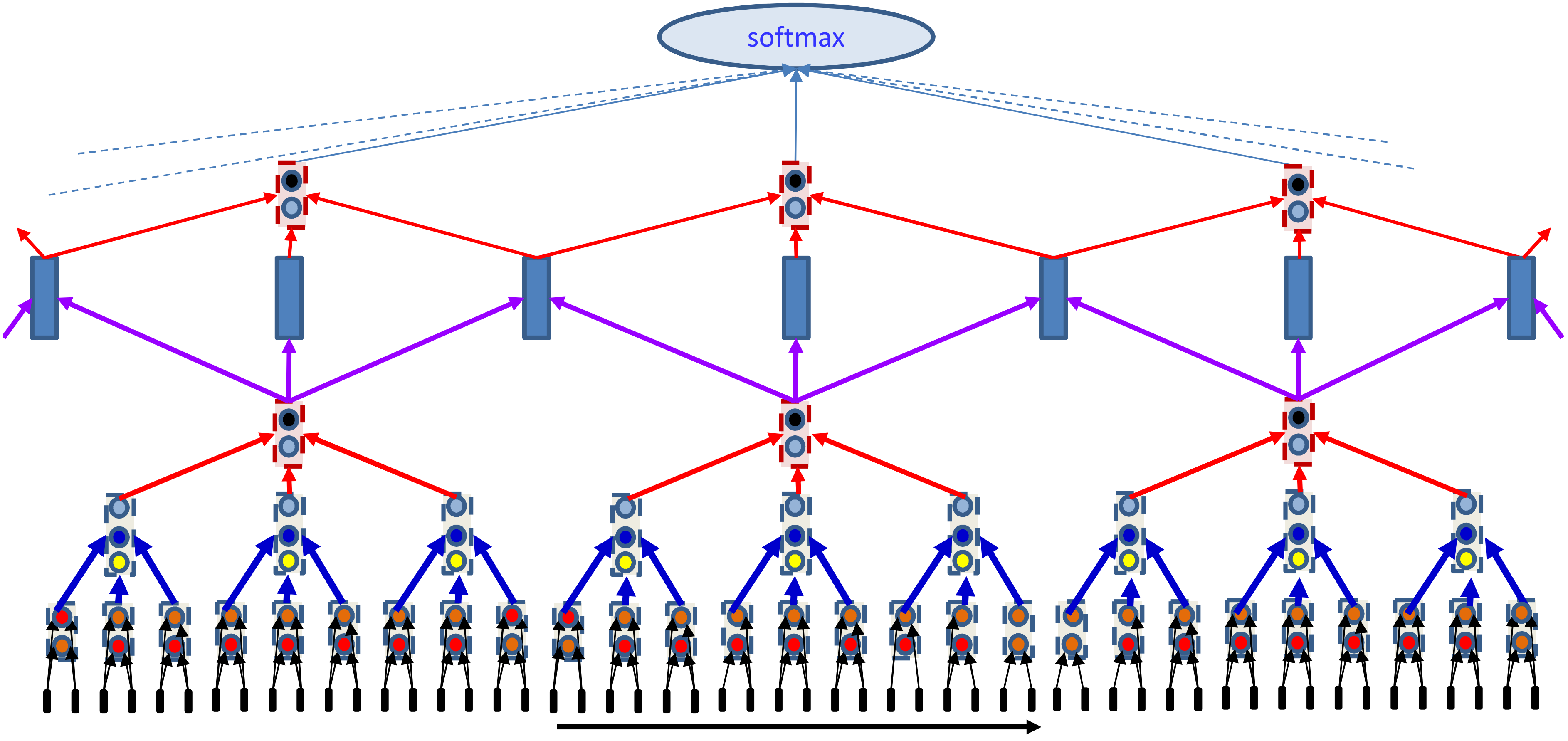
- We’ve seen this before.. where??

Recall this 1-D figure



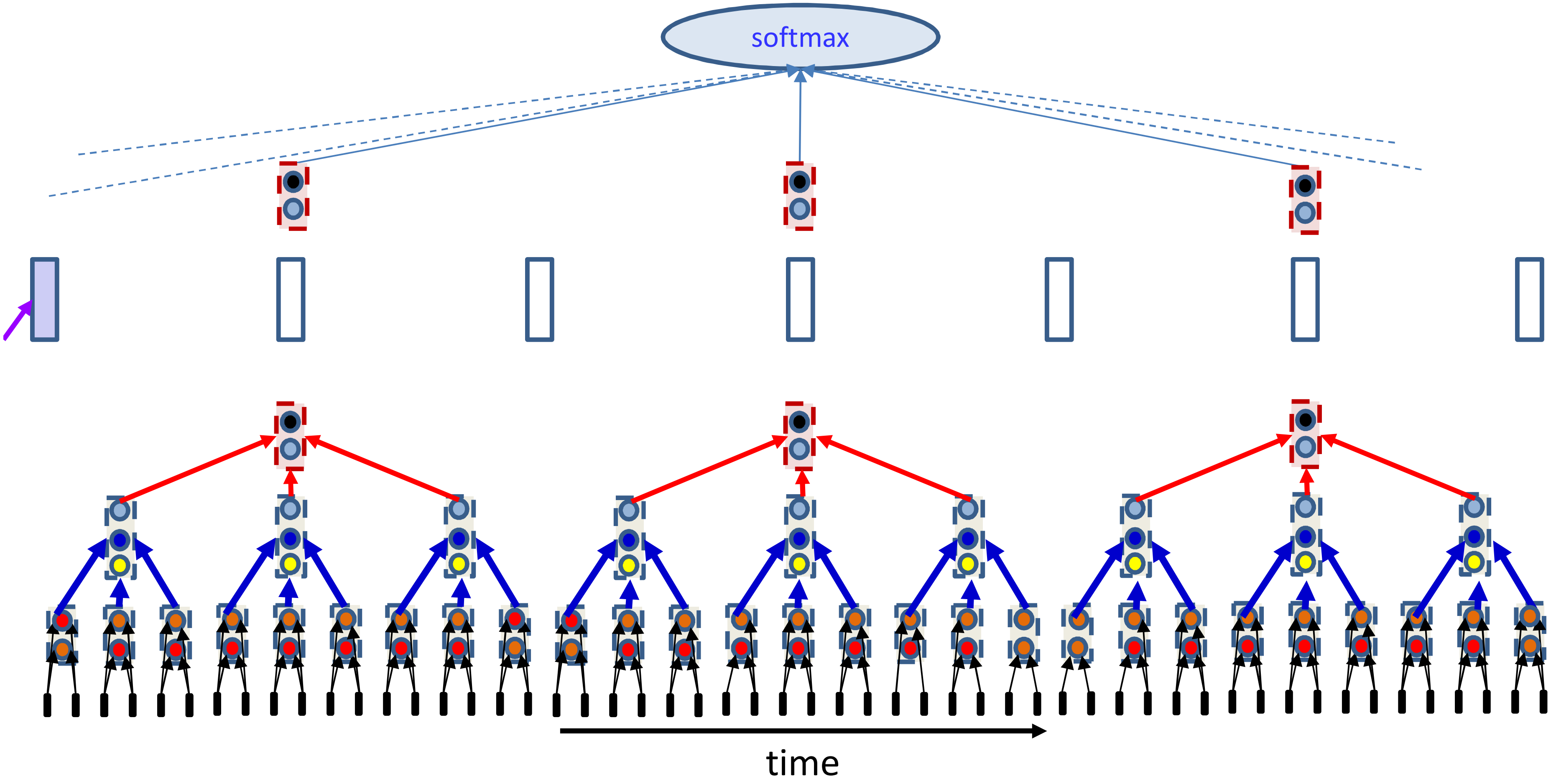
- Simplified diagram

With layer of increased size



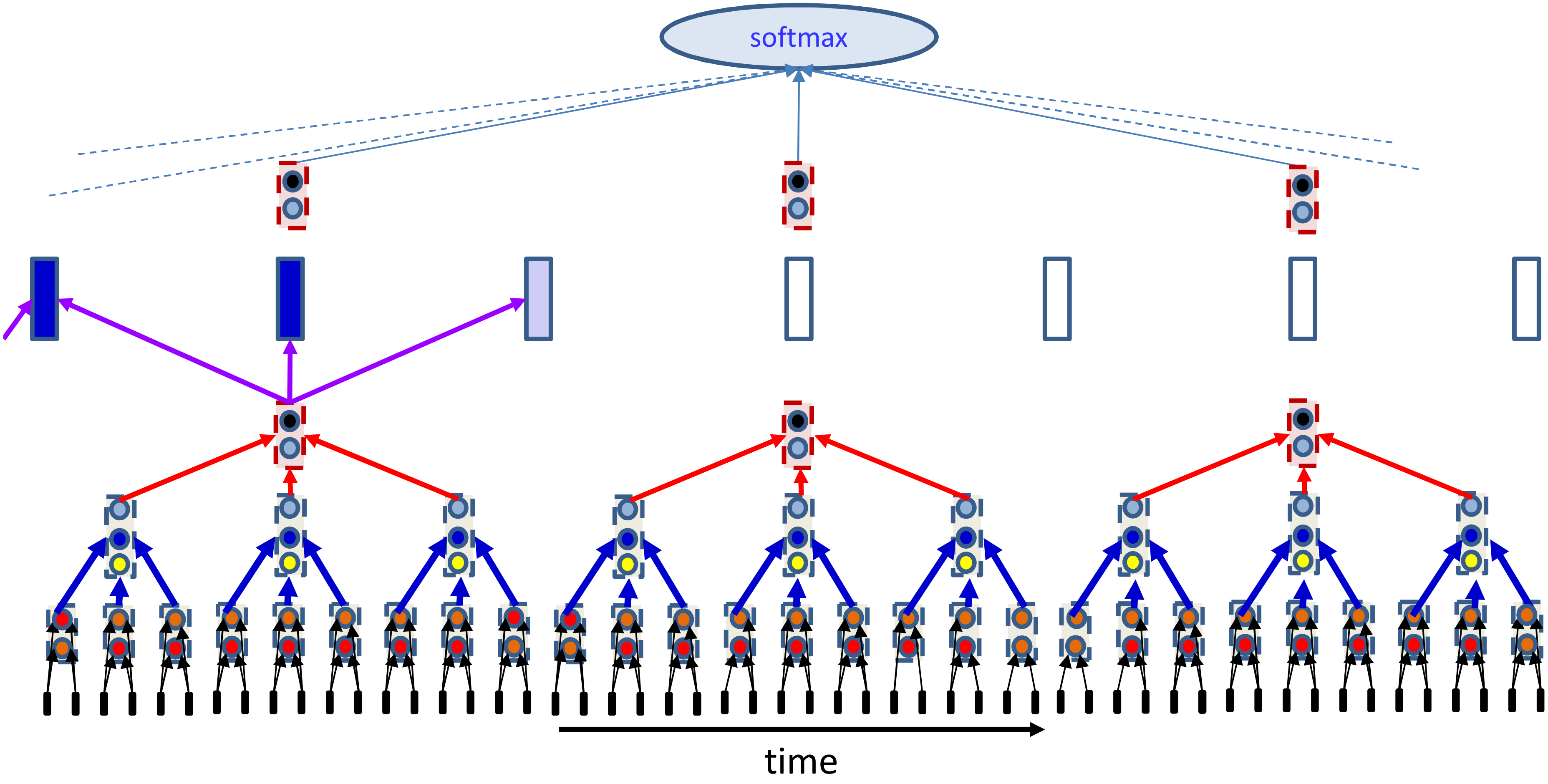
- *Maintaining Symmetry:*
  - Vertical bars in the 4th layer are regularly arranged w.r.t. bars of layer 3
  - The pattern of values of upward weights for each of the three pink (3rd layer) bars is identical

With layer of increased size



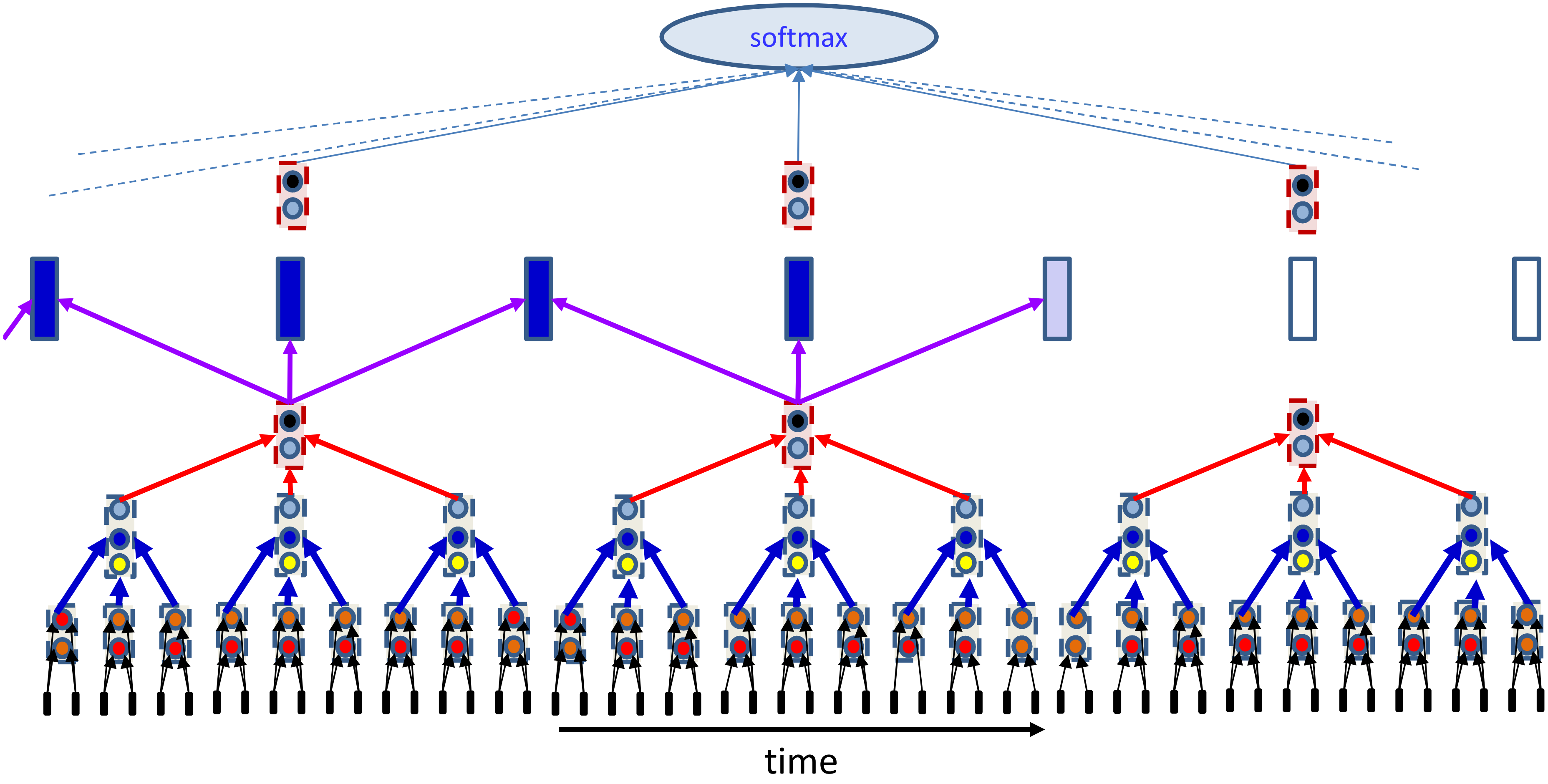
- Flow of info from bottom to top when implemented as a left-to-right scan
    - Note: Arrangement of vertical bars is predetermined by architecture

With layer of increased size



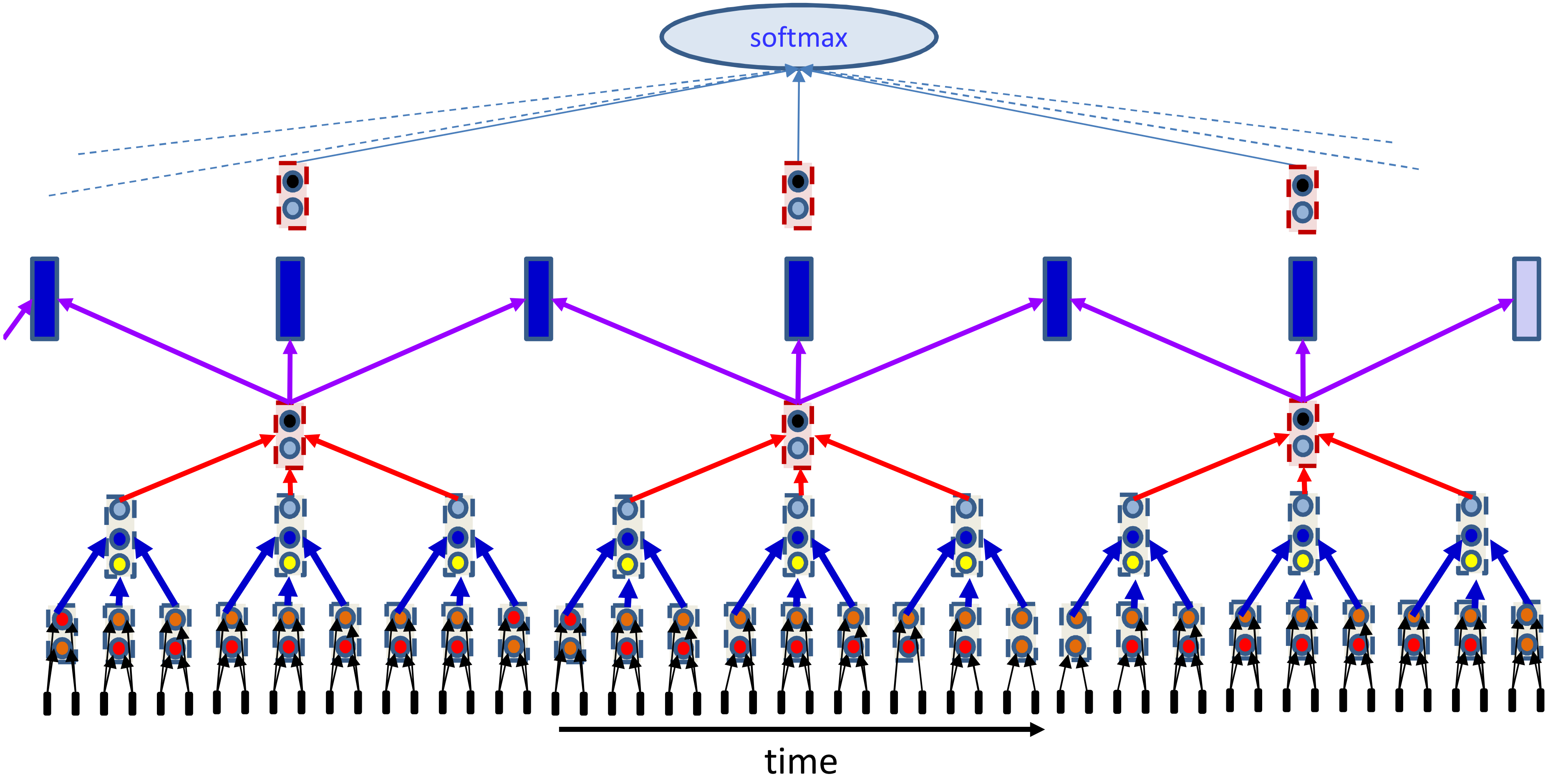
- Flow of info from bottom to top when implemented as a left-to-right scan
  - Note: Arrangement of vertical bars is predetermined by architecture

With layer of increased size



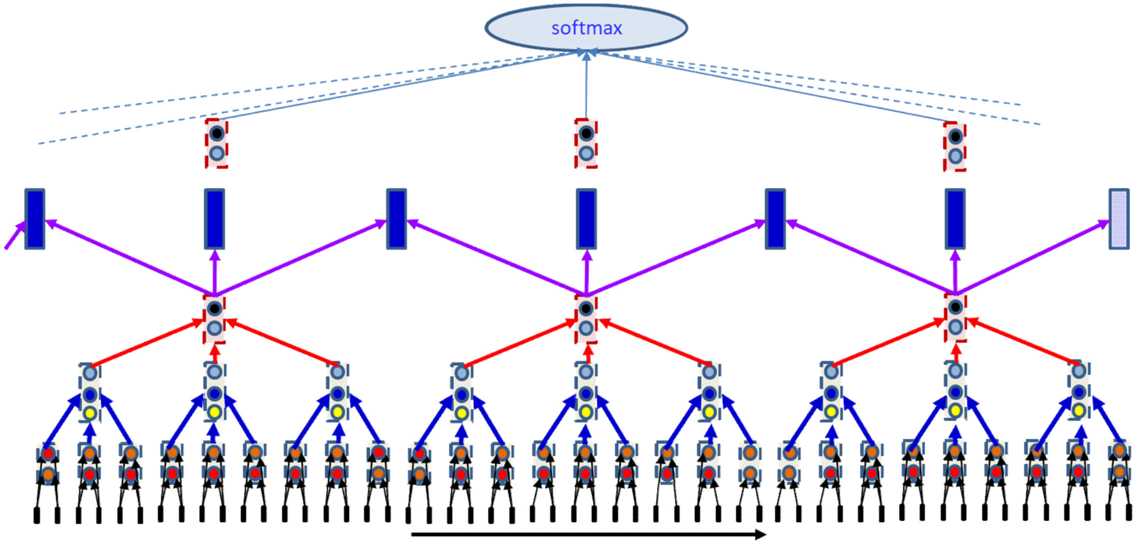
- Flow of info from bottom to top when implemented as a left-to-right scan
  - Note: Arrangement of vertical bars is predetermined by architecture

With layer of increased size



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

“Transposed Convolution”



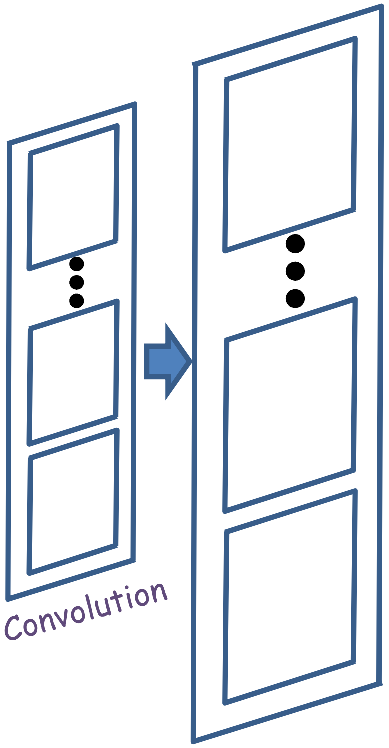
- Connection rules are transposed for expanding layers
  - In shrinking layers, the pattern of *incoming weights* is identical for each bar
  - In expanding layers, the pattern of *outgoing (upward) weights* is identical for each bar

- When thought of as an MLP, can write

$$Z_l = W_l Y_{l-1}$$

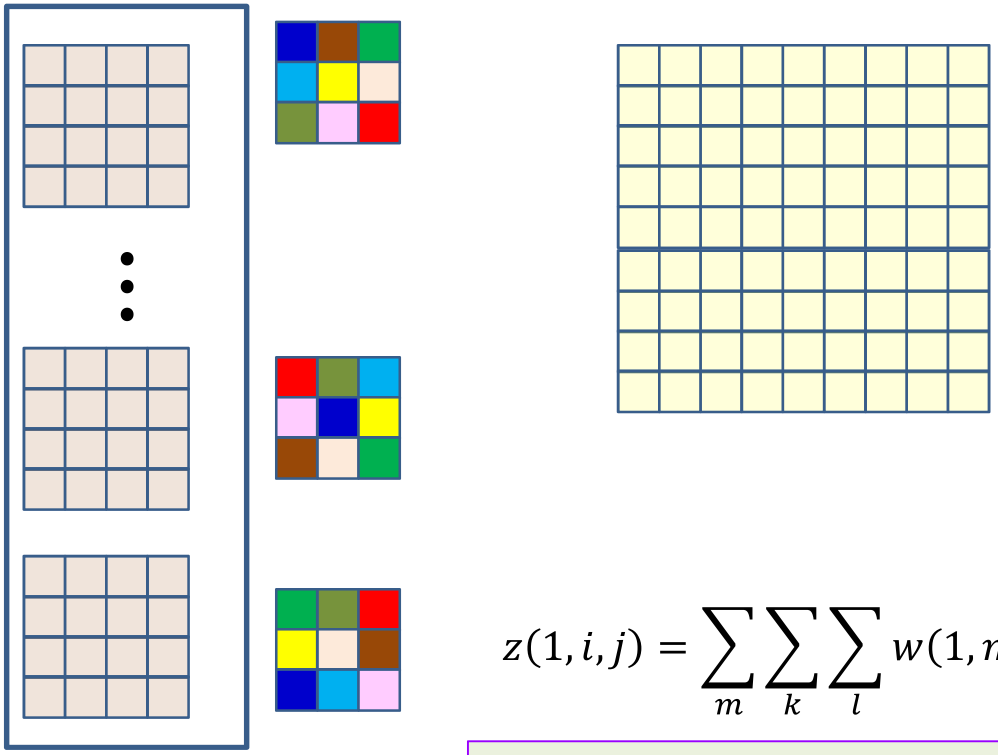
- W_l is broader than tall for a shrinking layer
- W_l is taller than broad for an expanding layer
  - Sometimes viewed as the transpose of a broad matrix
- Leading to terminology “transpose convolution”

In 2-D



- Similar computation

2D expanding convolution

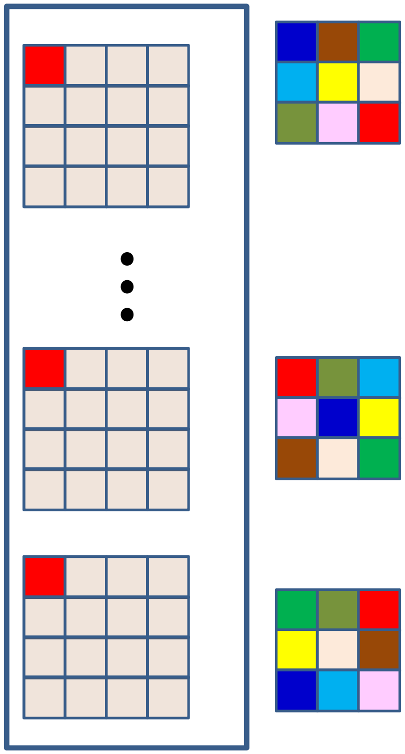


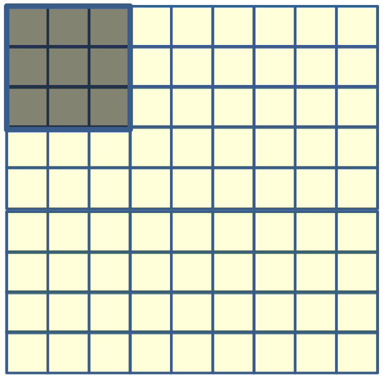
$$z(1,i,j) = \sum_{m} \sum_{k} \sum_{l} w(1,m,i-kb,j-lb)I(m,k,l)$$

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

- Output size is typically an integer multiple of input
    - +1 if filter width is odd
  - Easier to determine assignment of output to input

2D expanding convolution



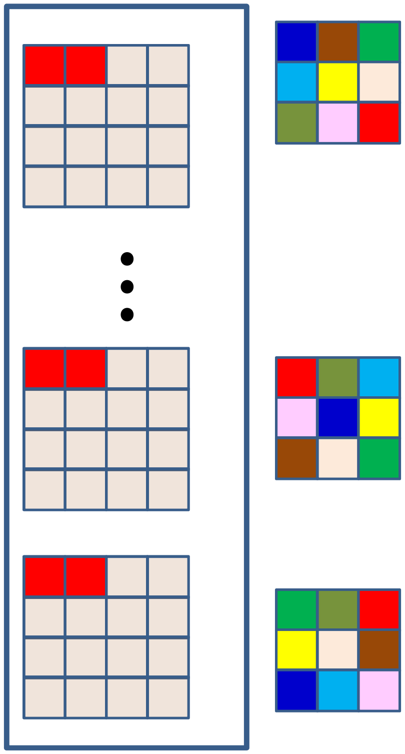


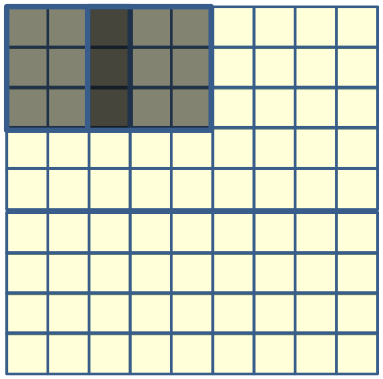
$$z(1,i,j) = \sum_{m} \sum_{k} \sum_{l} w(1,m,i-kb,j-lb)I(m,k,l)$$

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

- Output size is typically an integer multiple of input
    - +1 if filter width is odd
  - Easier to determine assignment of output to input

2D expanding convolution



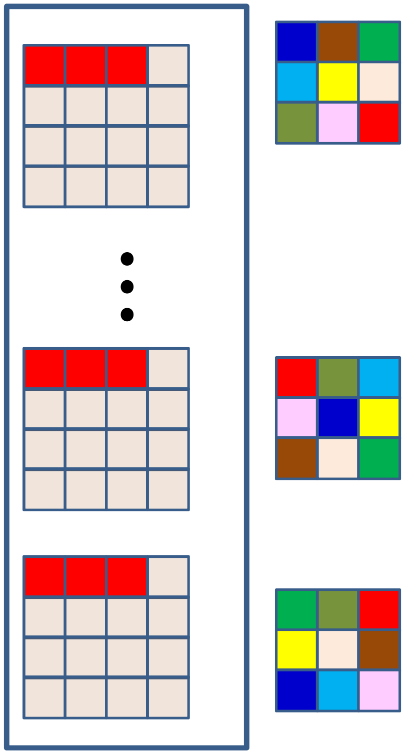


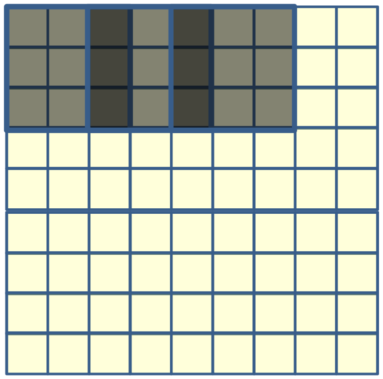
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb)I(m,k,l)$$

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

- Output size is typically an integer multiple of input
  - +1 if filter width is odd
  - – Easier to determine assignment of output to input

2D expanding convolution



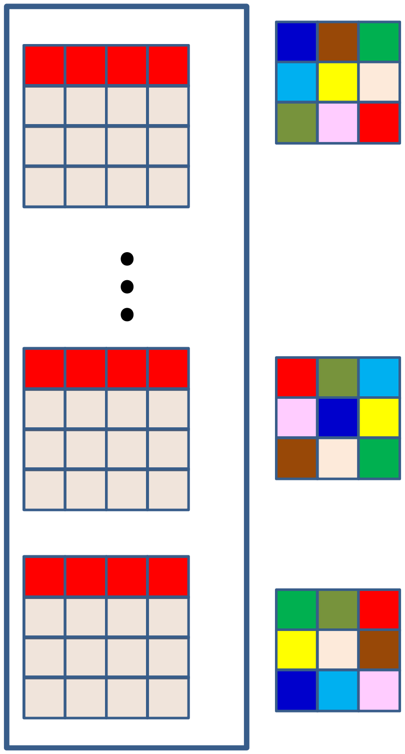


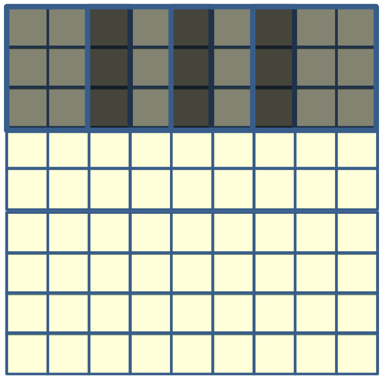
$$z(1,i,j) = \sum_{m} \sum_{k} \sum_{l} w(1,m,i-kb,j-lb)I(m,k,l)$$

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

- Output size is typically an integer multiple of input
    - +1 if filter width is odd
  - Easier to determine assignment of output to input

2D expanding convolution



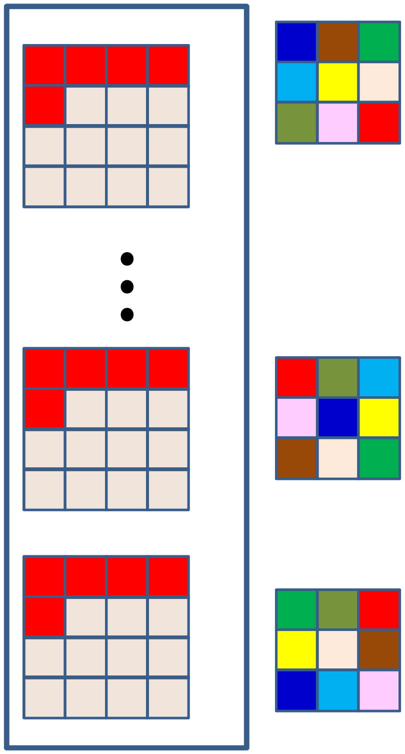


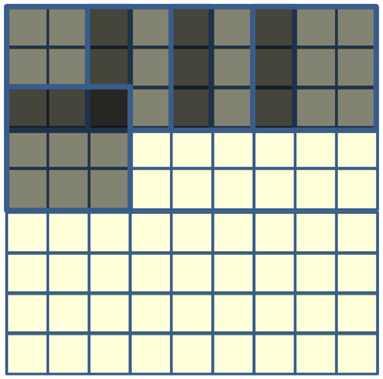
$$z(1,i,j) = \sum_{m} \sum_{k} \sum_{l} w(1,m,i-kb,j-lb)I(m,k,l)$$

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

- Output size is typically an integer multiple of input
  - +1 if filter width is odd
  - Easier to determine assignment of output to input

2D expanding convolution



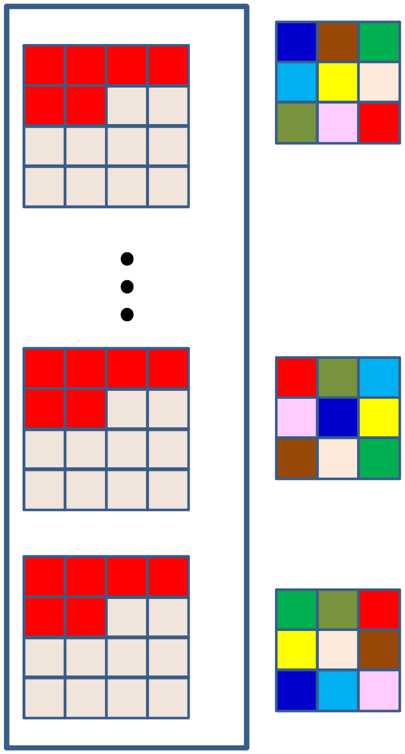


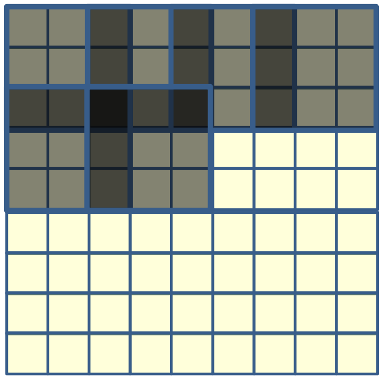
$$z(1,i,j) = \sum_{m} \sum_{k} \sum_{l} w(1,m,i-kb,j-lb)I(m,k,l)$$

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

- Output size is typically an integer multiple of input
    - +1 if filter width is odd
    - Easier to determine assignment of output to input

2D expanding convolution



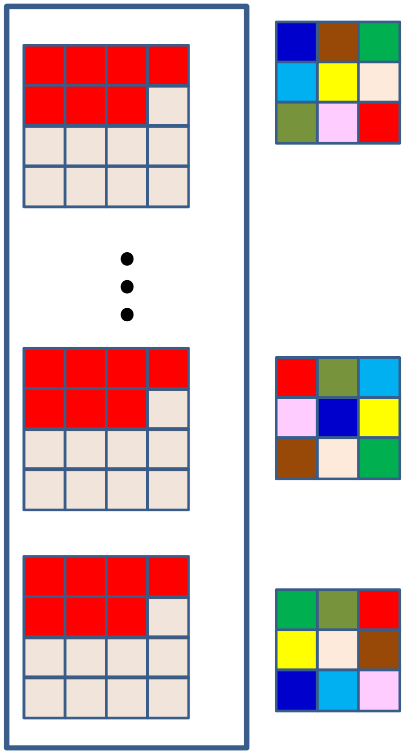


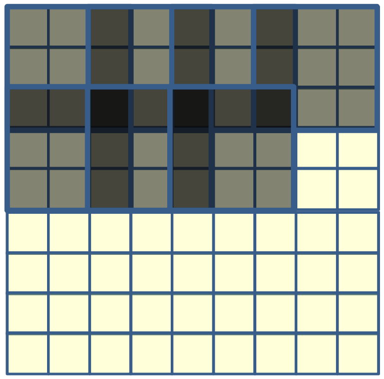
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb)I(m,k,l)$$

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

- Output size is typically an integer multiple of input
  - +1 if filter width is odd
  - – Easier to determine assignment of output to input

2D expanding convolution



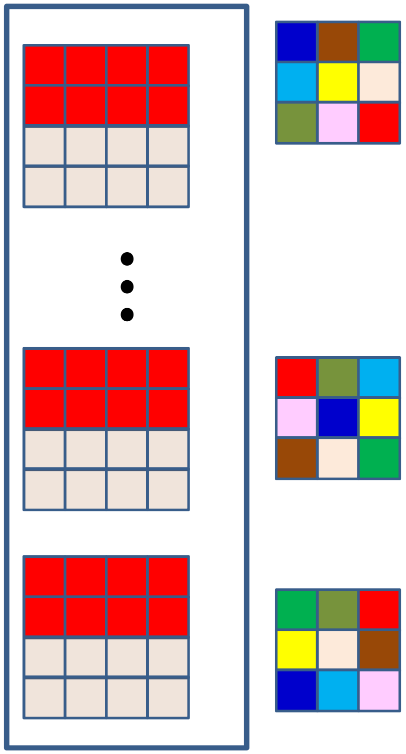


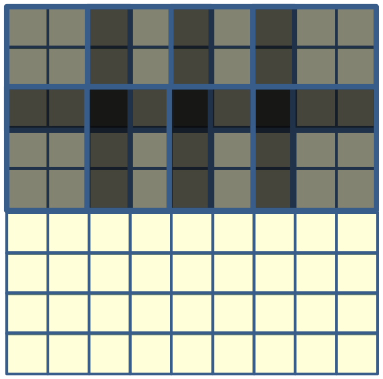
$$z(1,i,j) = \sum_{m}\sum_{k}\sum_{l} w(1,m,i-kb,j-lb)I(m,k,l)$$

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

- Output size is typically an integer multiple of input
    - +1 if filter width is odd
  - Easier to determine assignment of output to input

2D expanding convolution



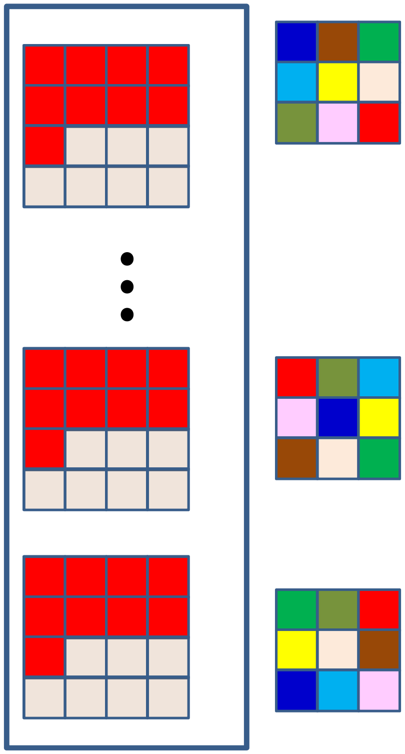


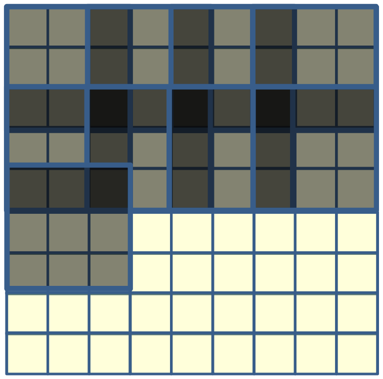
$$z(1,i,j) = \sum_{m} \sum_{k} \sum_{l} w(1,m,i-kb,j-lb) I(m,k,l)$$

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

- Output size is typically an integer multiple of input
  - +1 if filter width is odd
  - Easier to determine assignment of output to input

2D expanding convolution



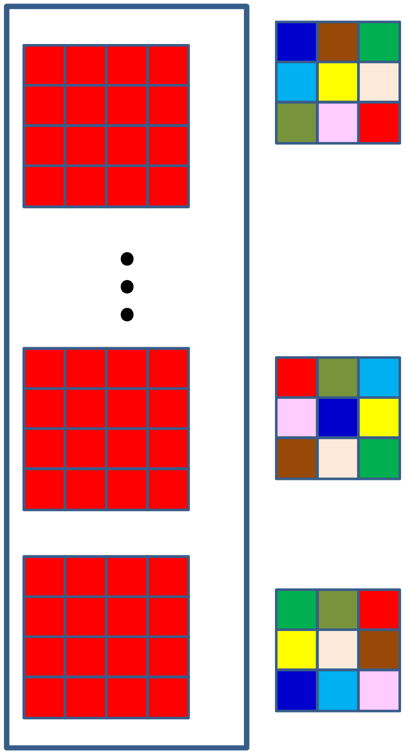


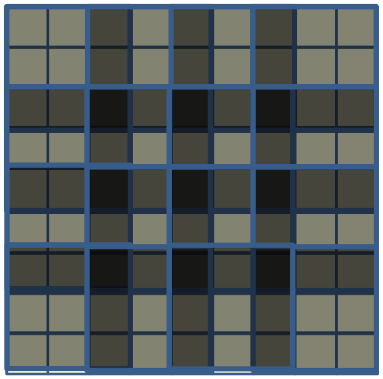
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb)I(m,k,l)$$

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

- Output size is typically an integer multiple of input
  - +1 if filter width is odd
  - Easier to determine assignment of output to input

2D expanding convolution





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

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

- Output size is typically an integer multiple of input
    - +1 if filter width is odd
  - – Easier to determine assignment of output to input

CNN: Expanding convolution layer l

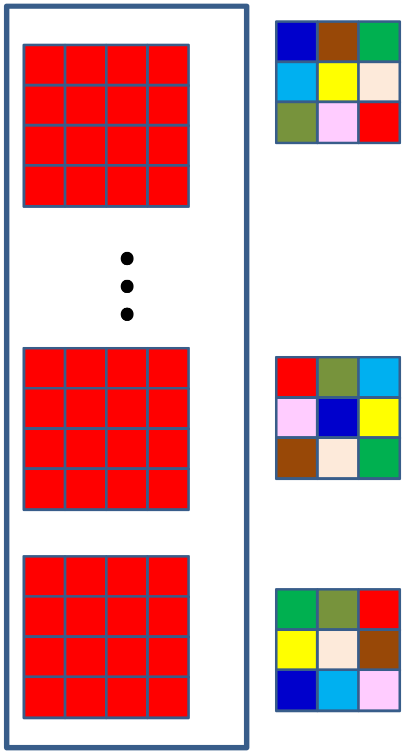
```
Z(l) = zeros(Dl x (Wb+$K_l$) x (Hb+$K_l$)) # b = stride
for j = 1:$D_l$
  for x = 1:W
    for y = 1:H
      for i = 1:$D_{l-1}$
        for x' = 1:$K_l$
          for y' = 1:$K_l$
            z(l,j,(x-1)b+x',(y-1)b+y') +=
                      w(l,j,i,x',y')y(l-1,i,x,y)
```

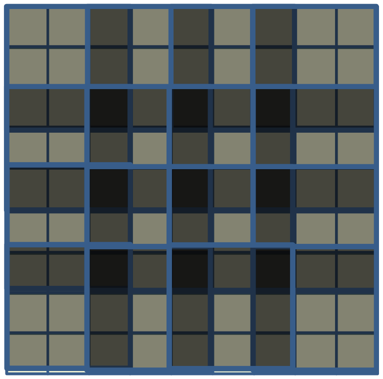
CNN: Expanding convolution layer l

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

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

2D expanding convolution





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

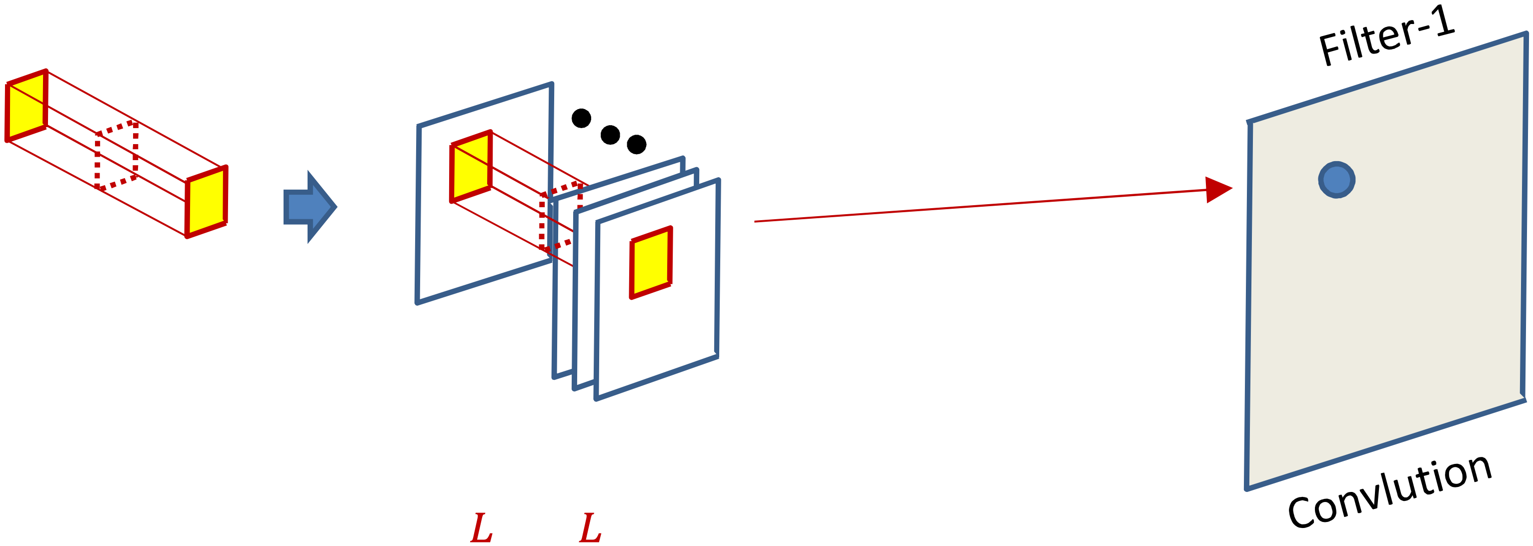
b is the "stride"
(scaling factor between the sizes of Z and Y)

- Also called *transpose convolution*
  - If you recast the CNN as a shared-parameter MLP, expanding layers have weight matrices that are taller than wide
- Also called "deconvolution"
  - Strictly speaking, abuse of terminology

Invariance

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

Shift-invariance – a different perspective

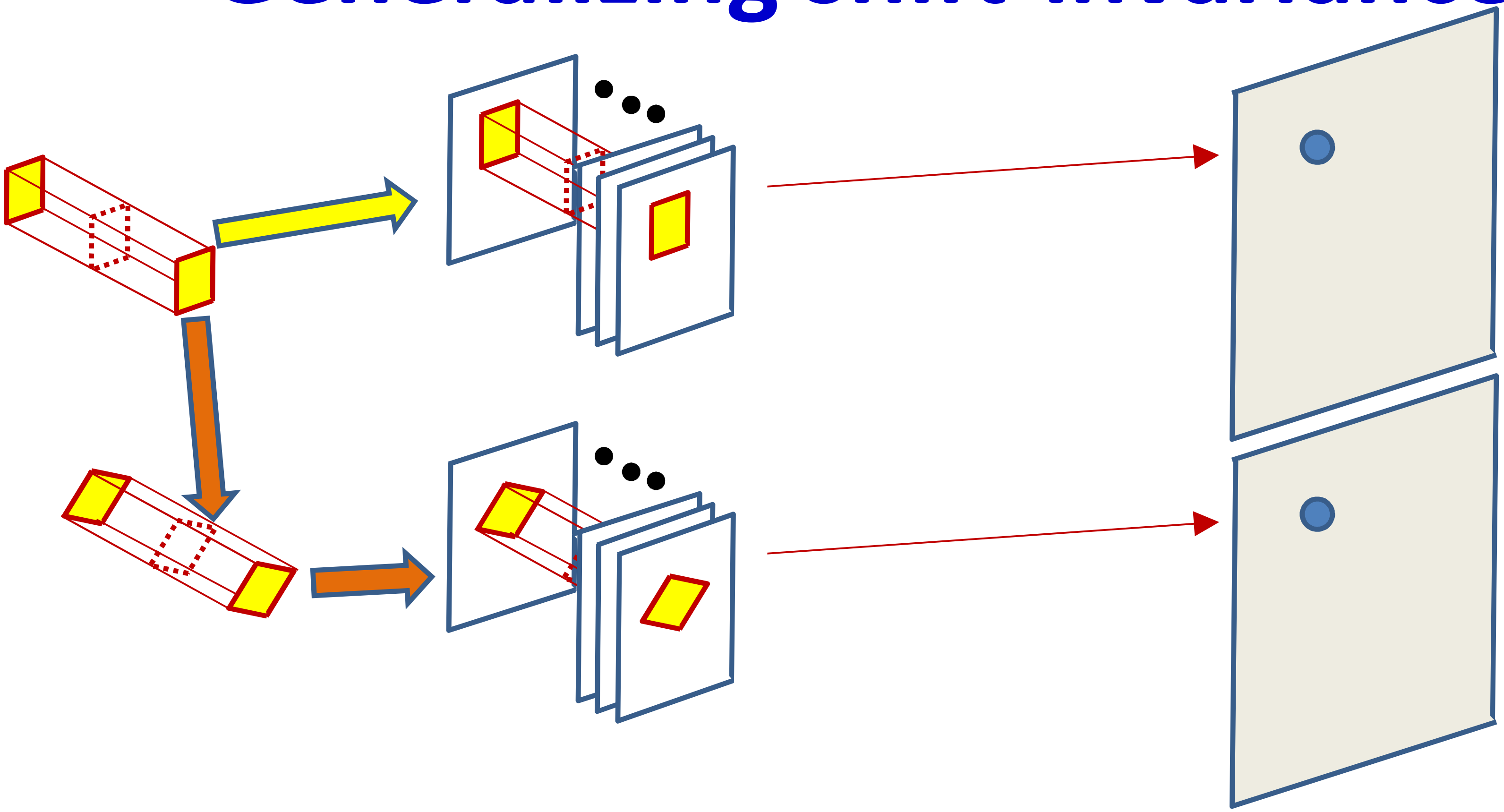


$$z(l,s,i,j) = \sum_{p} \sum_{k=1}^{L} \sum_{m=1}^{L} w(l,s,p,k,m) Y(l-1,p,i+k,j+m)$$

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

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

Generalizing shift-invariance



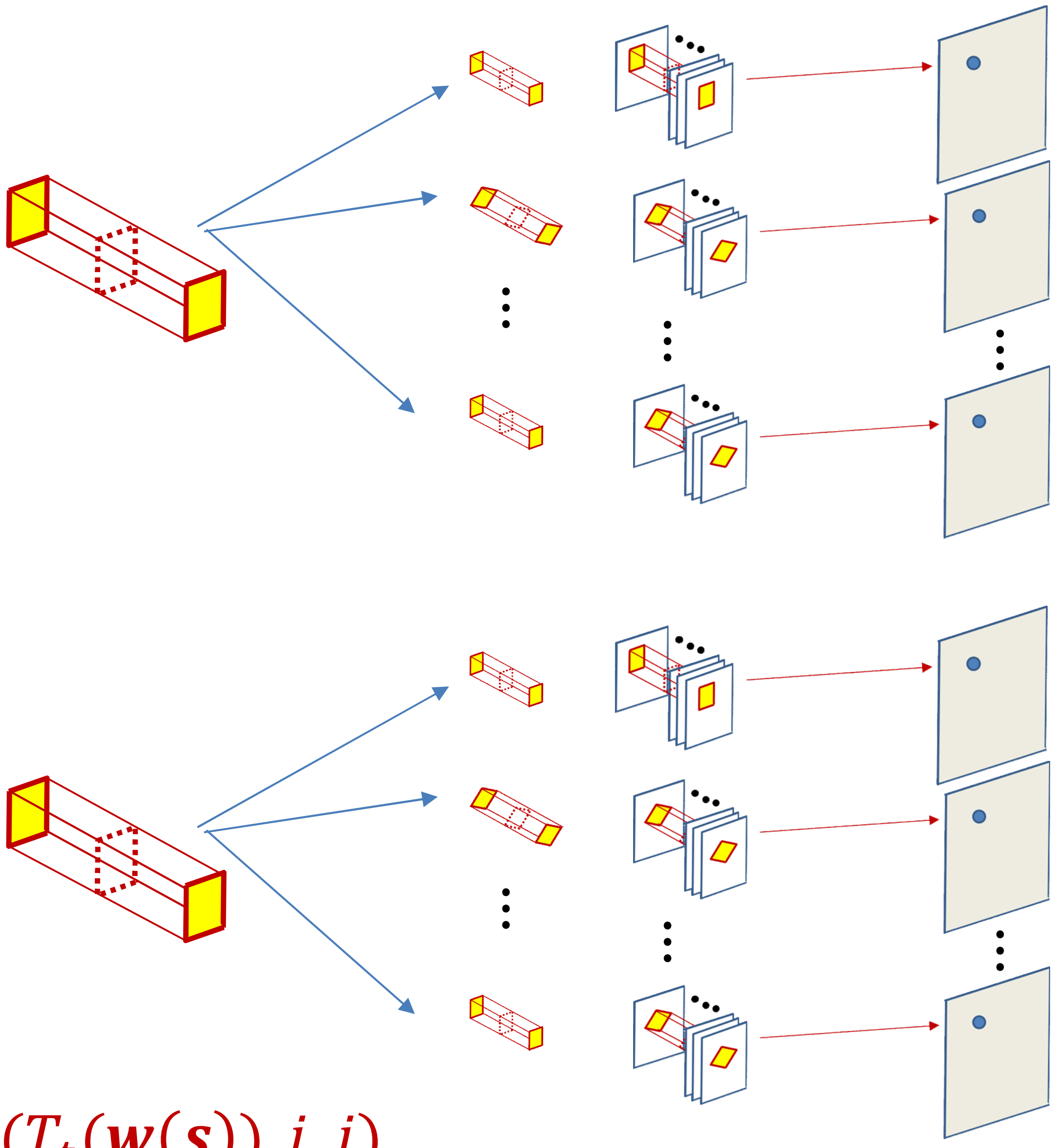
$$z_{regular}(s,i,j) = \boldsymbol{Y}.shift(\boldsymbol{w}(\boldsymbol{s}),i,j)$$

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

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

Transform invariance

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



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

Regular CNN : single layer *l*

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

```
for j = 1:$D_l$
  for x = 1:$W_{l-1}-K_l+1$
    for y = 1:$H_{l-1}-K_l+1$
      segment = Y(l-1, :, x:x+$K_l$-1, y:y+$K_l$-1) #3D tensor
      z(l,j,x,y) = W(l,j).segment #tensor inner prod.
      Y(l,j,x,y) = activation(z(l,j,x,y))
```

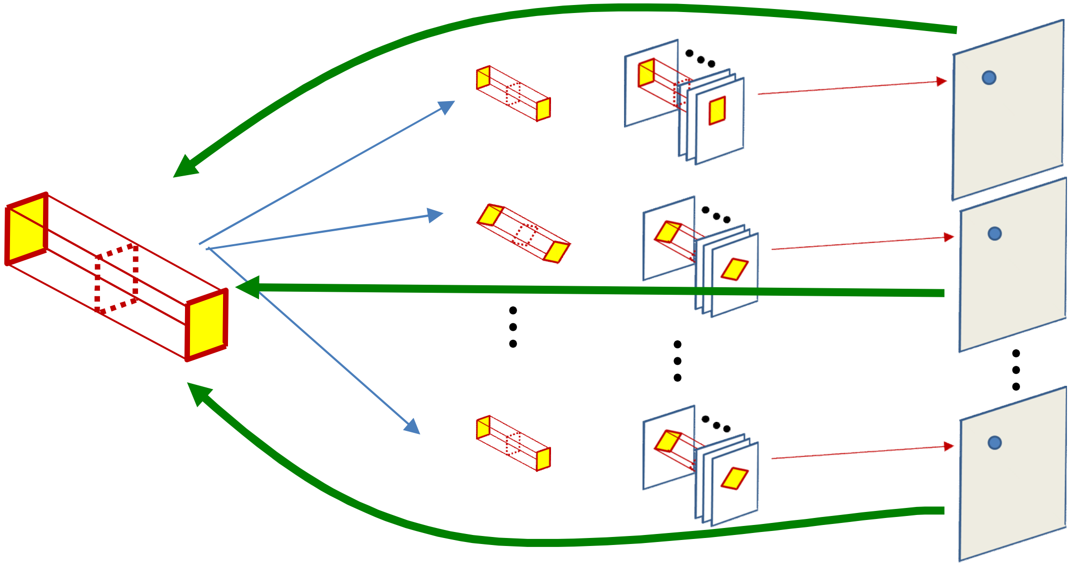
Transform invariance

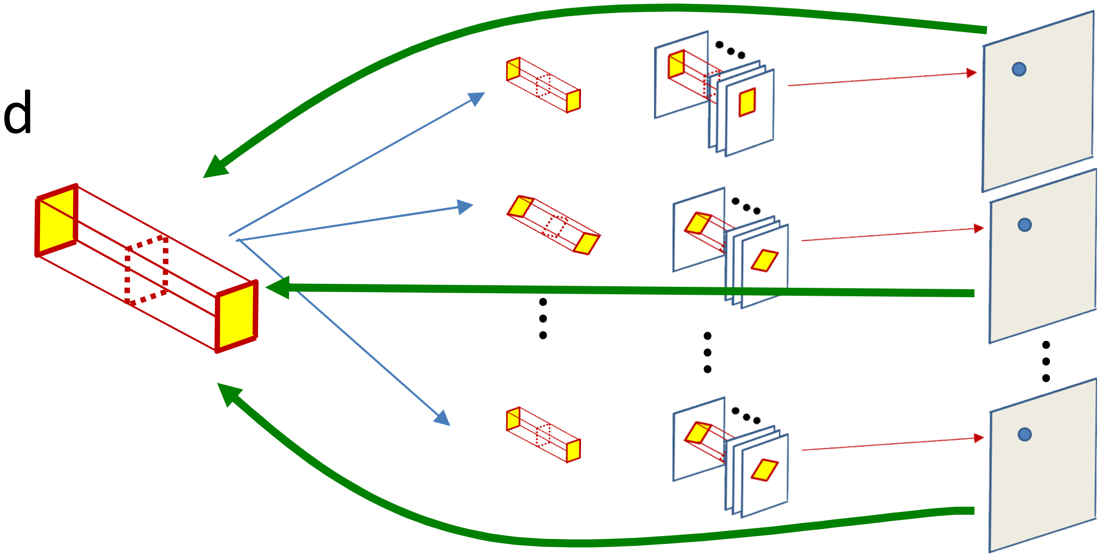
The weight W(l,j)is a 3D D_{l-1}xK_lxK_l tensor

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

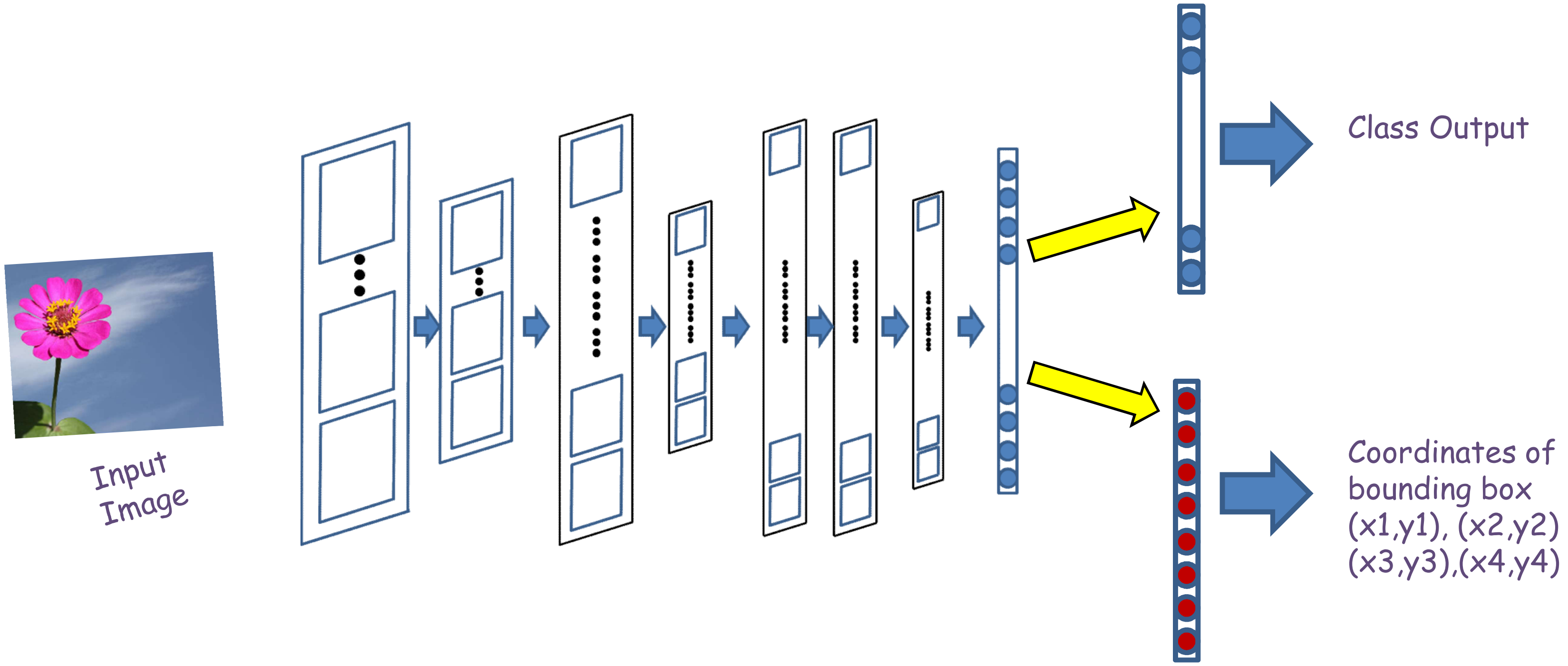
- The models can be easily modified to include invariance to other transforms
  - Although these tend to be computationally painful

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

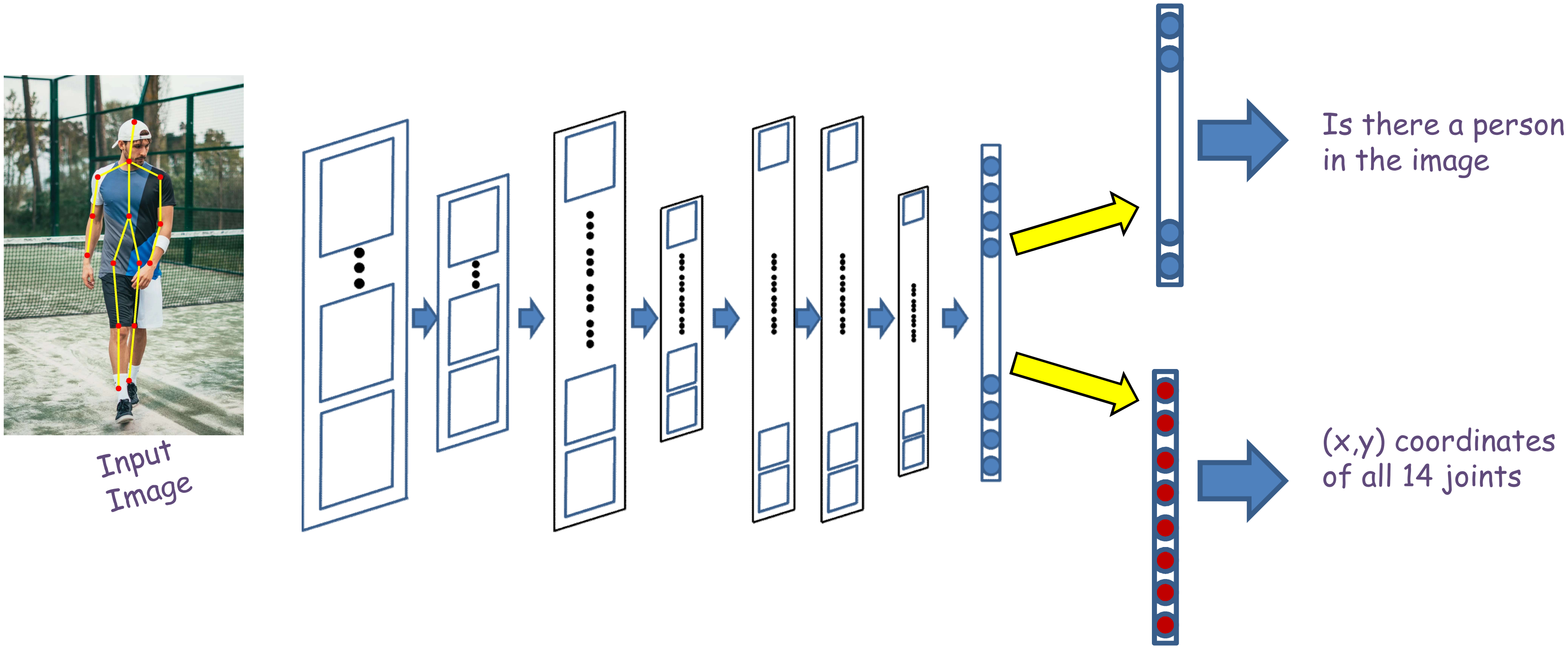
- But can we detect the *position* of the main object?

Finding Bounding Boxes


Input Image
Class Output
Coordinates of bounding box (x1,y1), (x2,y2) (x3,y3),(x4,y4)

- 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


Input Image
Is there a person in the image
(x,y) coordinates of all 14 joints

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

Story so far

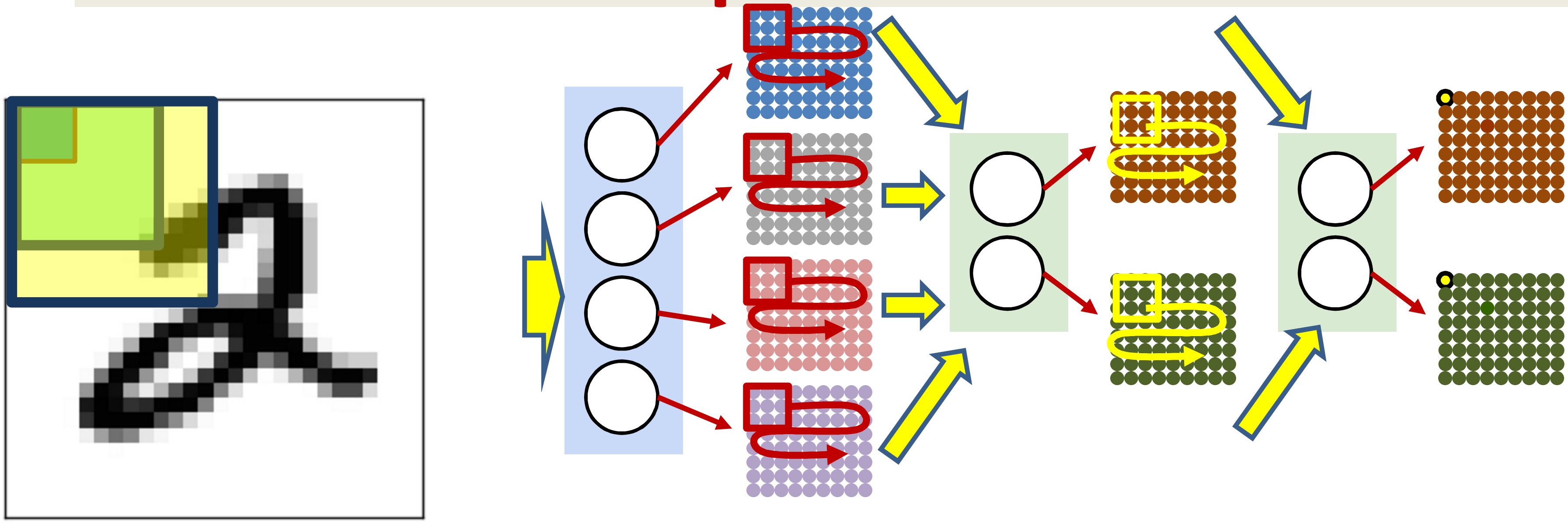
- CNNs are shift-invariant neural-network models for shift-invariant pattern detection
  - Are equivalent to scanning with shared-parameter MLPs with distributed representations

- 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

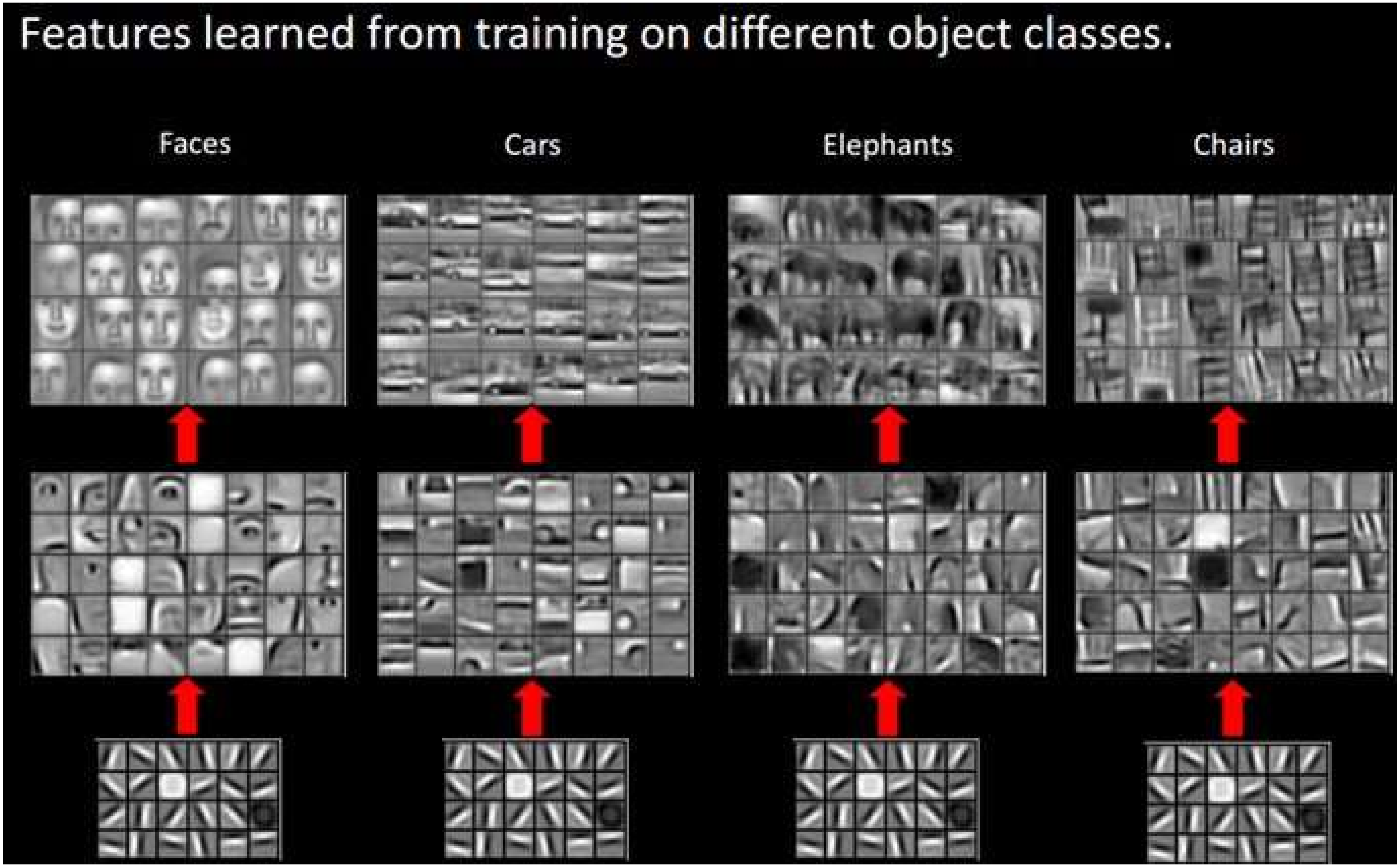
- The models can be easily modified to include invariance to other transforms
  - Although these tend to be computationally painful

- Can also make predictions related to the position and arrangement of target object through multi-task learning

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.

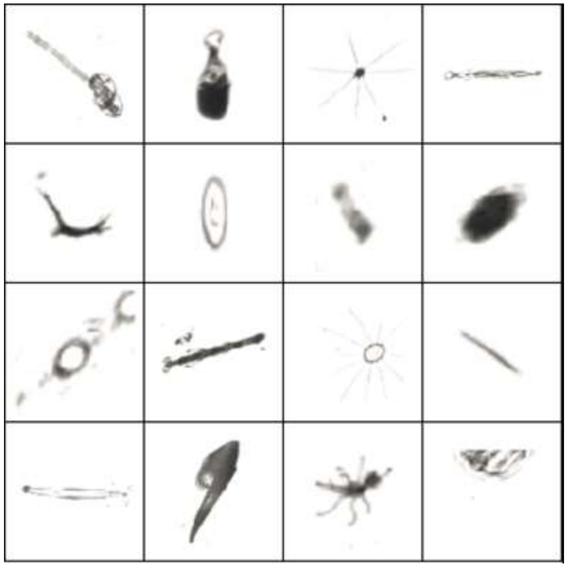
Training Issues

- Standard convergence issues
  - Solution: Adam or other momentum-style algorithms
  - Other tricks such as batch normalization

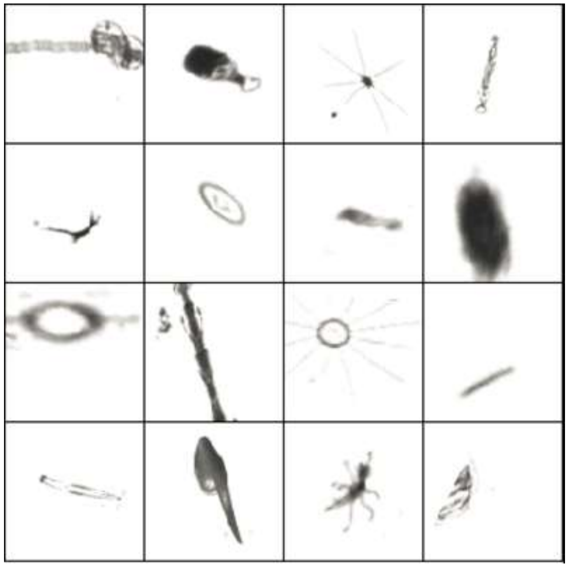
- The number of parameters can quickly become very large
- Insufficient training data to train well
  - Solution: Data augmentation

Data Augmentation

Original data



Augmented data



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

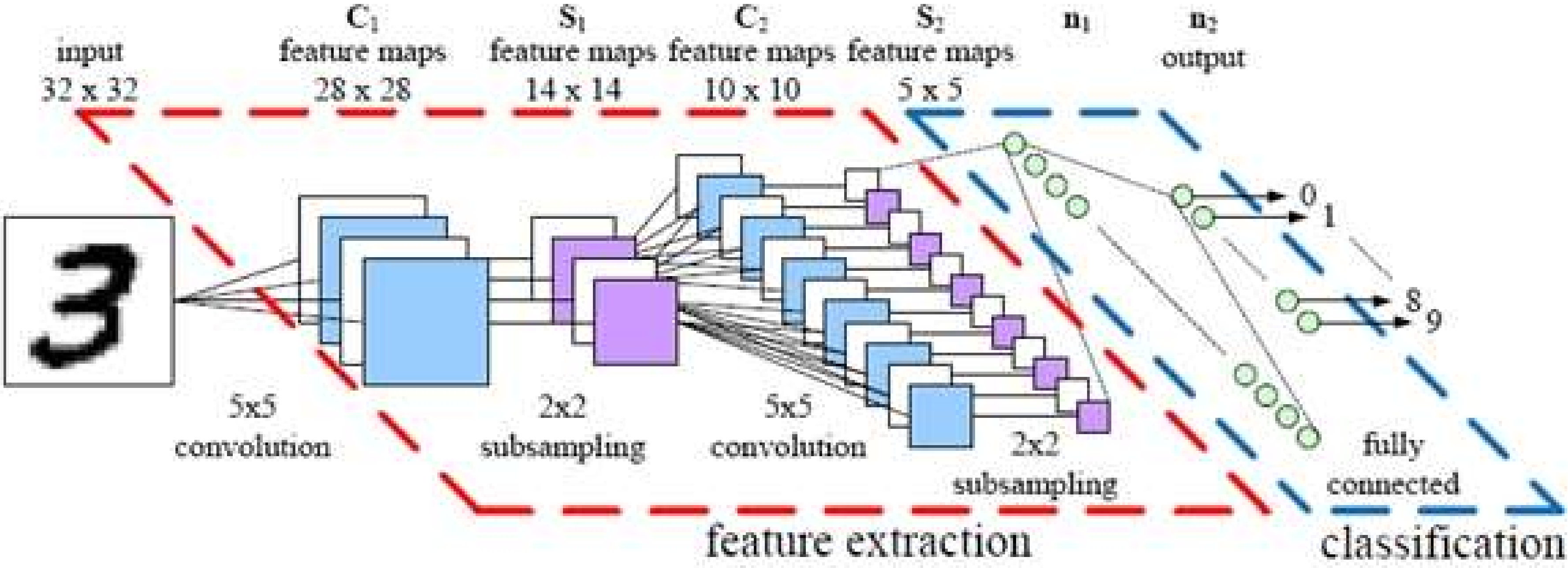
Other tricks

- *Very deep* networks
  - *100* or more layers in MLP
  - Formalism called “Resnet”

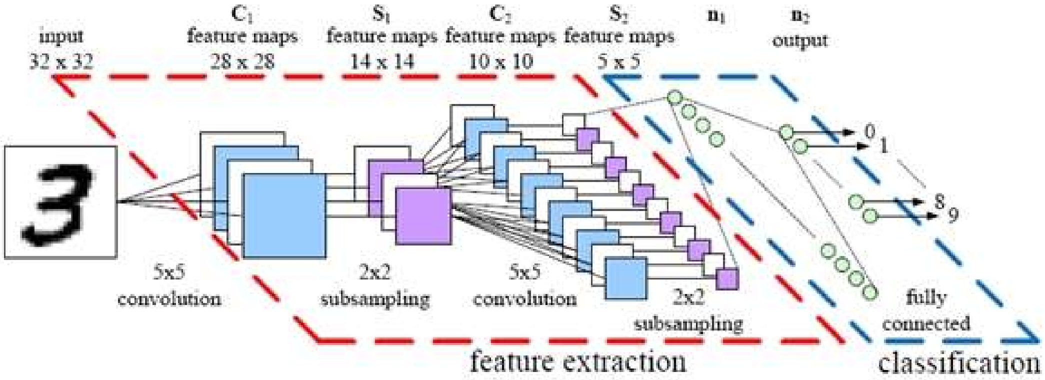
Convolutional neural nets

- 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*

Digit classification



Le-net 5

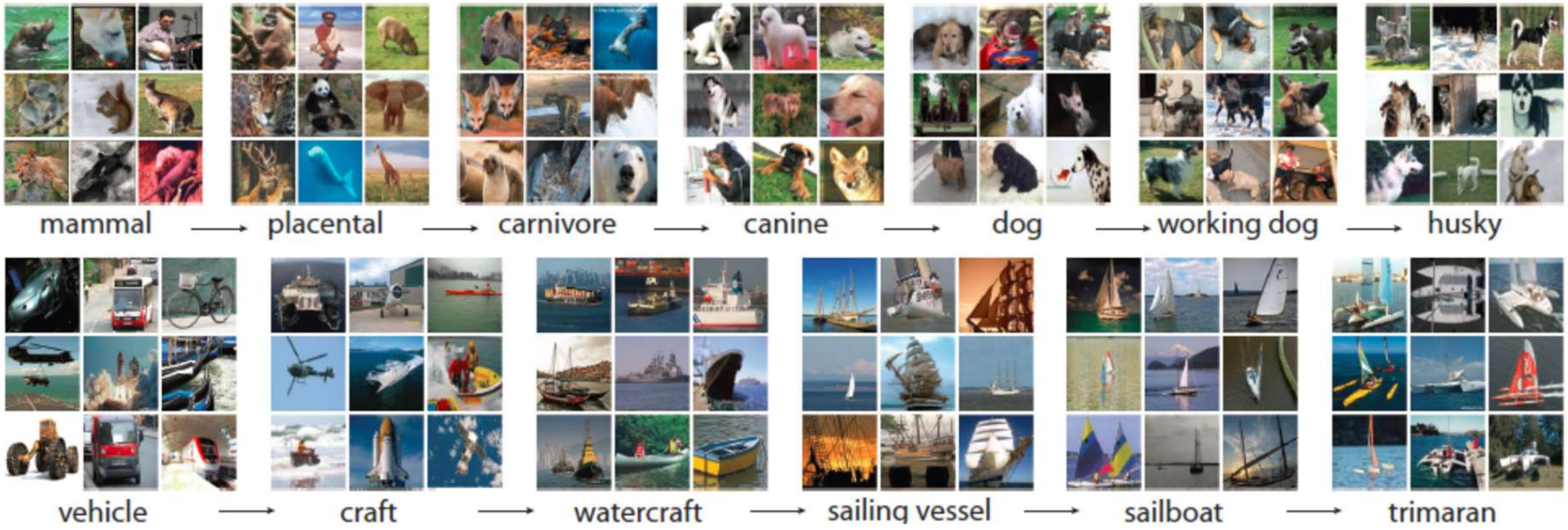


- Digit recognition on MNIST (32x32 images)
  - **Conv1:** 6 5x5 filters in first conv layer (no zero pad), stride 1
    - Result: 6 28x28 maps
  - **Pool1:** 2x2 max pooling, stride 2
    - Result: 6 14x14 maps
  - **Conv2:** 16 5x5 filters in second conv layer, stride 1, no zero pad
    - Result: 16 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

Nice visual example

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

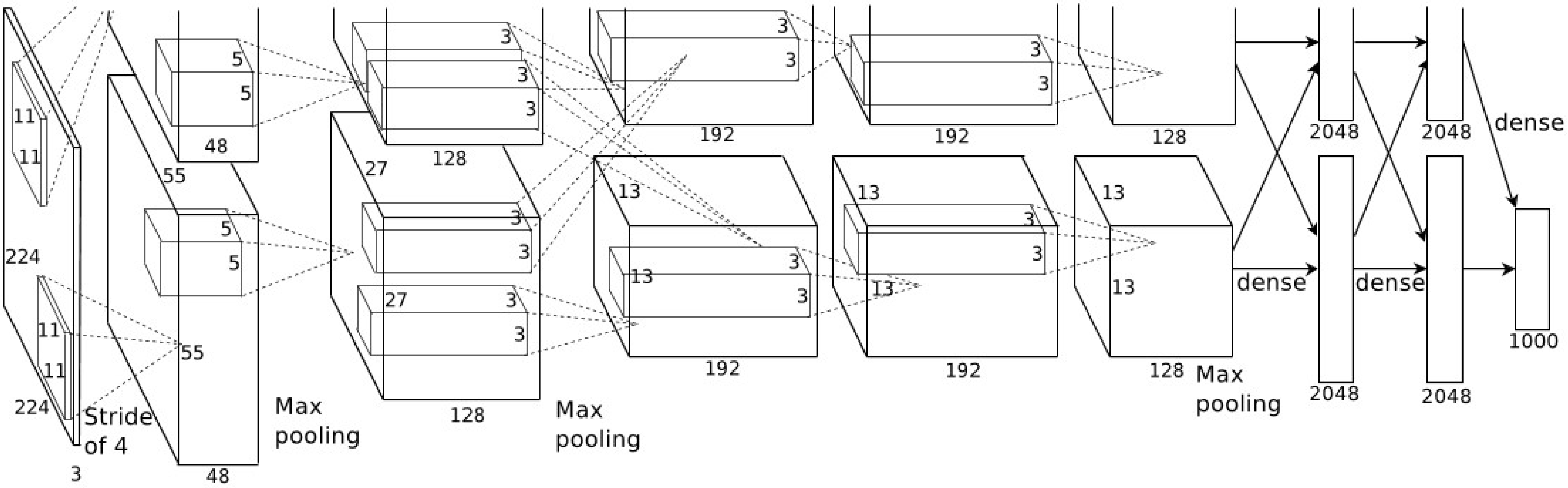
The imagenet task



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

AlexNet

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



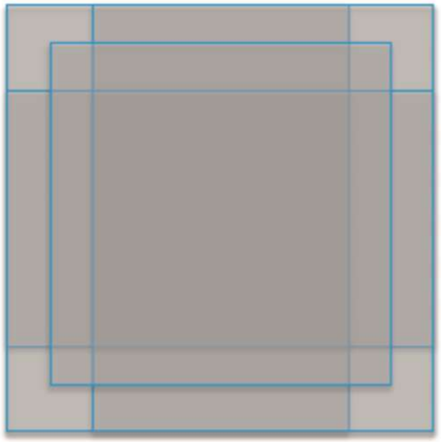
Krizhevsky, A., Sutskever, I. and Hinton, G. E. “ImageNet Classification with Deep Convolutional Neural Networks” NIPS 2012: Neural Information Processing Systems, Lake Tahoe, Nevada

Krizhevsky et. al.

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

Alexnet: Total parameters

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



10 patches

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

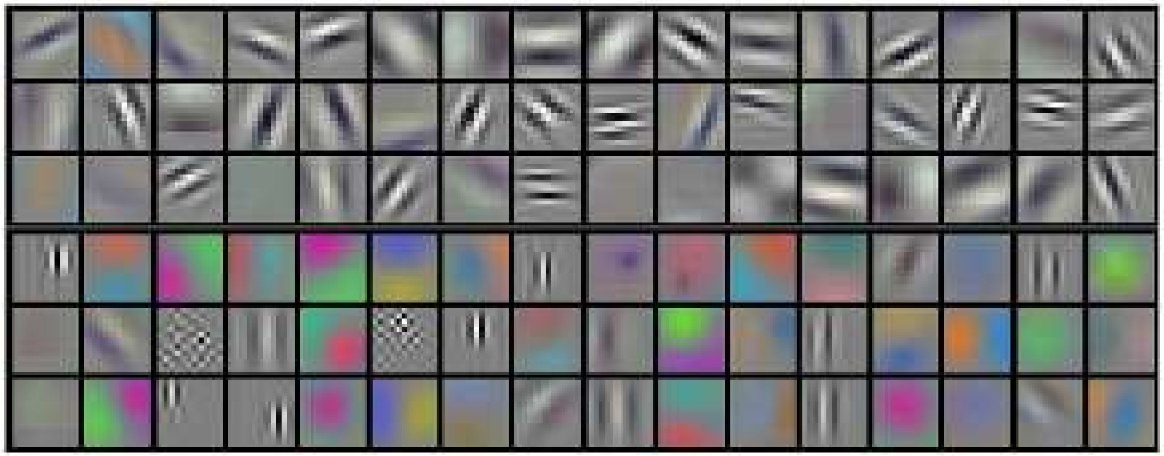
Learning magic in Alexnet

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

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

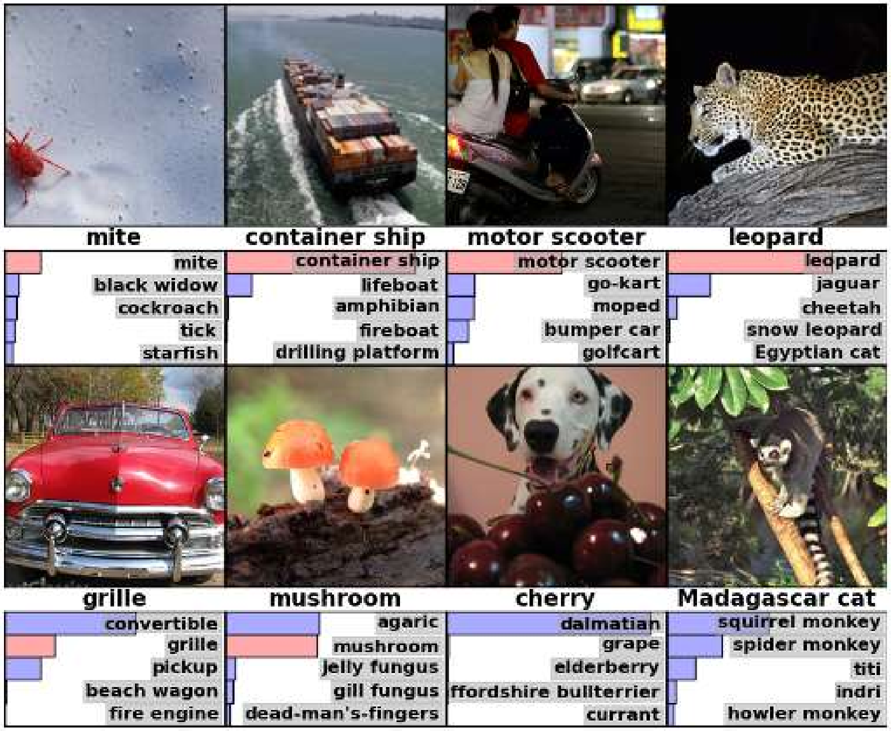
ImageNet

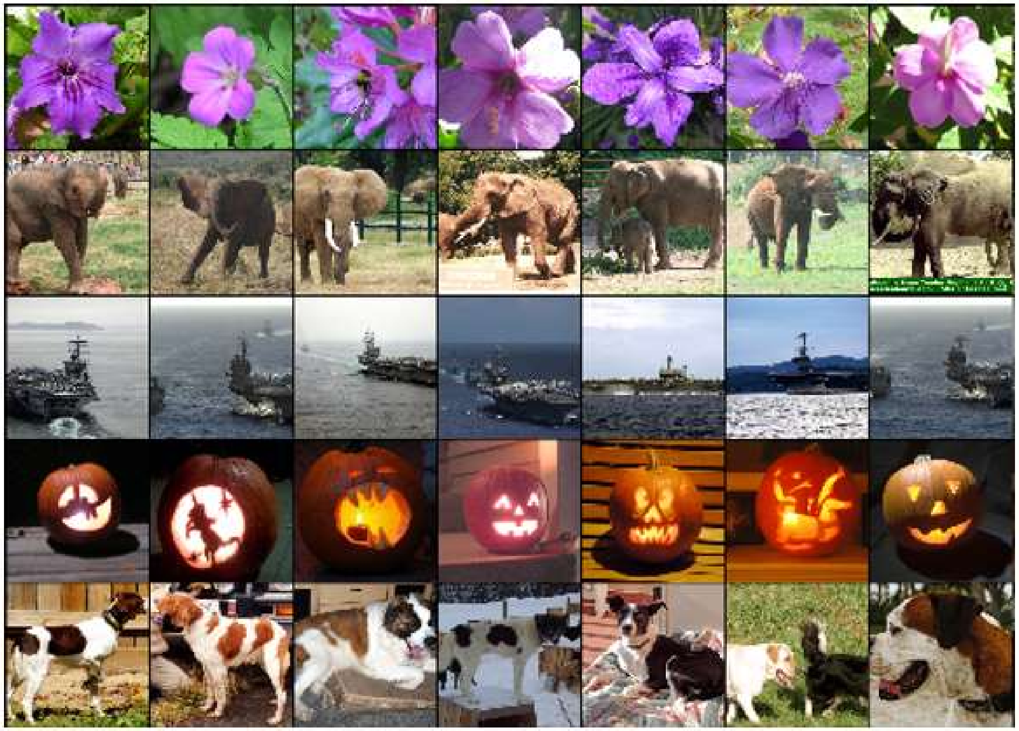
Figure 3: 96 convolutional kernels of size 11×11×3 learned by the first convolutional layer on the 224×224×3 input images. The top 48 kernels were learned on GPU 1 while the bottom 48 kernels were learned on GPU 2. See Section 6.1 for details.



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

The net actually *learns* features!



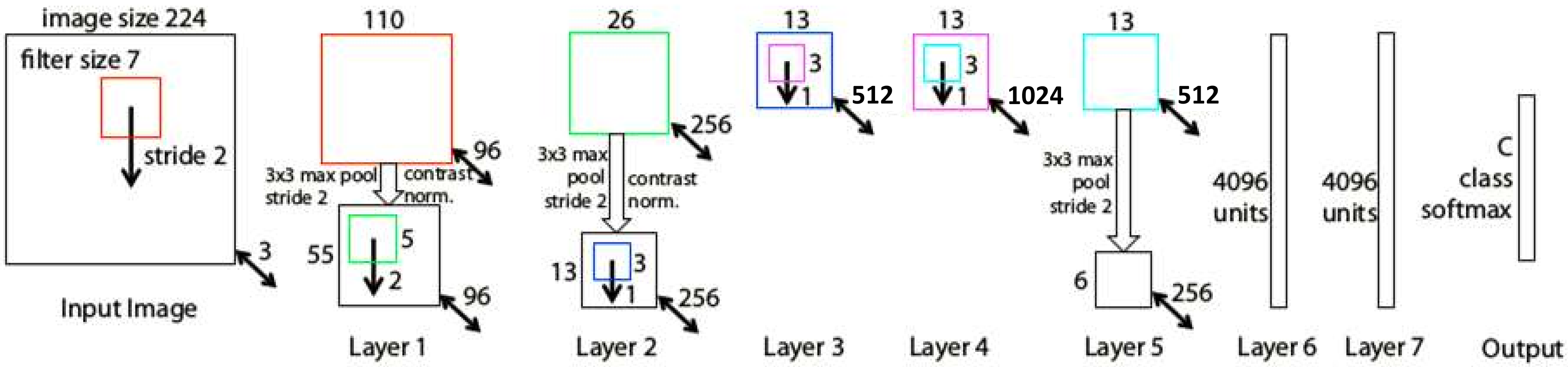


Eight ILSVRC-2010 test images and the five labels considered most probable by our model. The correct label is written under each image, and the probability assigned to the correct label is also shown with a red bar (if it happens to be in the top 5).

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

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

ZFNet



ZF Net Architecture

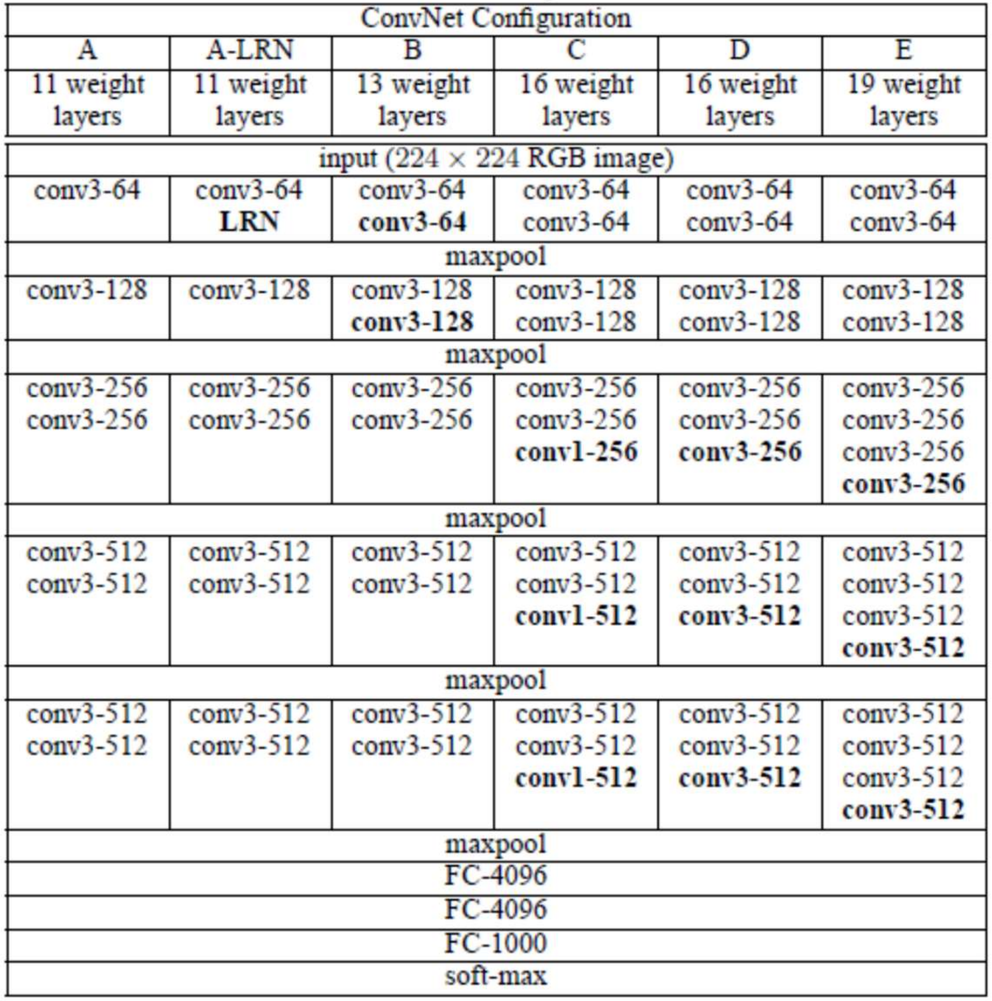
- Zeiler and Fergus 2013
- Same as Alexnet except:
  - 7x7 input-layer filters with stride 2
  - 3 conv layers are 512, 1024, 512
  - Error went down from 15.4% → 14.8%
    - Combining multiple models as before

VGGNet

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

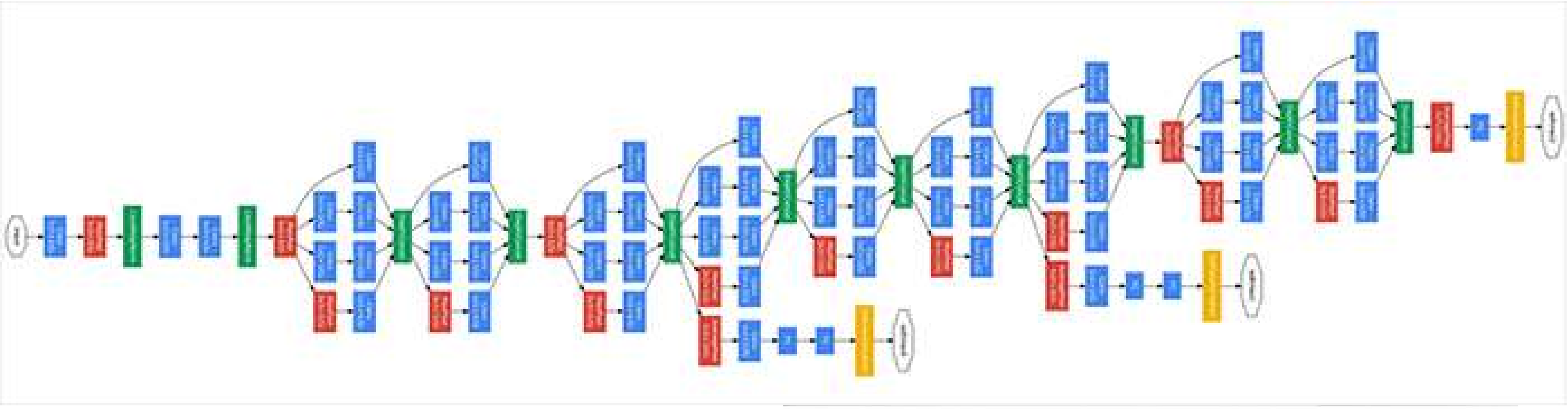
- Tried a large number of architectures.
- Finally obtained 7.3% top-5 error using 13 conv layers and 3 FC layers
  - Combining 7 classifiers
  - Subsequent to paper, reduced error to 6.8% using only two classifiers
- Final arch: 64 conv, 64 conv,
  64 pool,
  128 conv, 128 conv,
  128 pool,
  256 conv,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

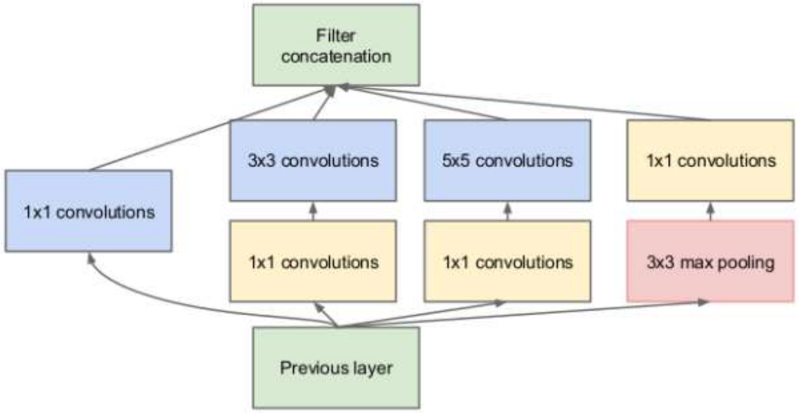
Madness!

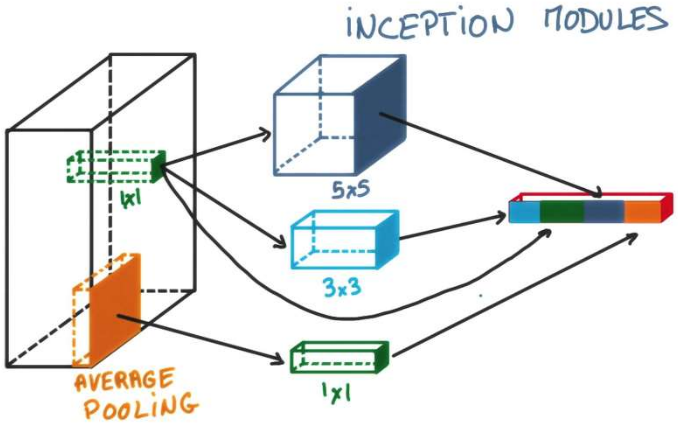


ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128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

Googlenet: Inception







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

Imagenet

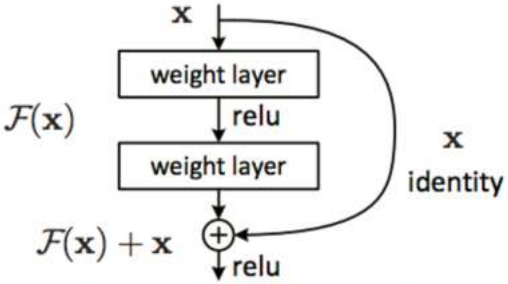
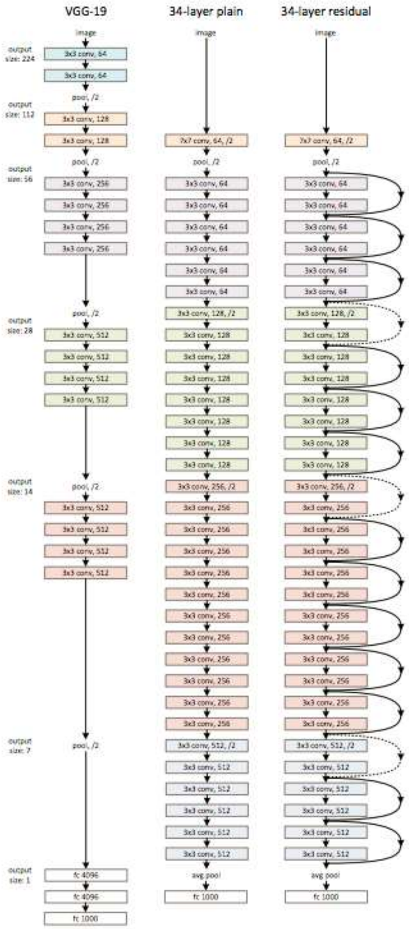

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

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..

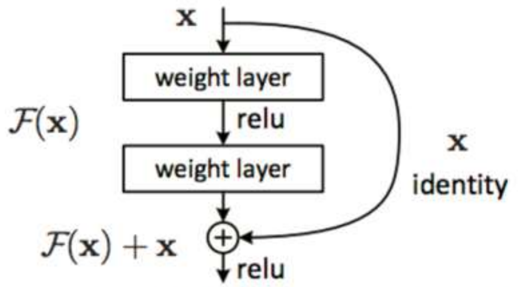
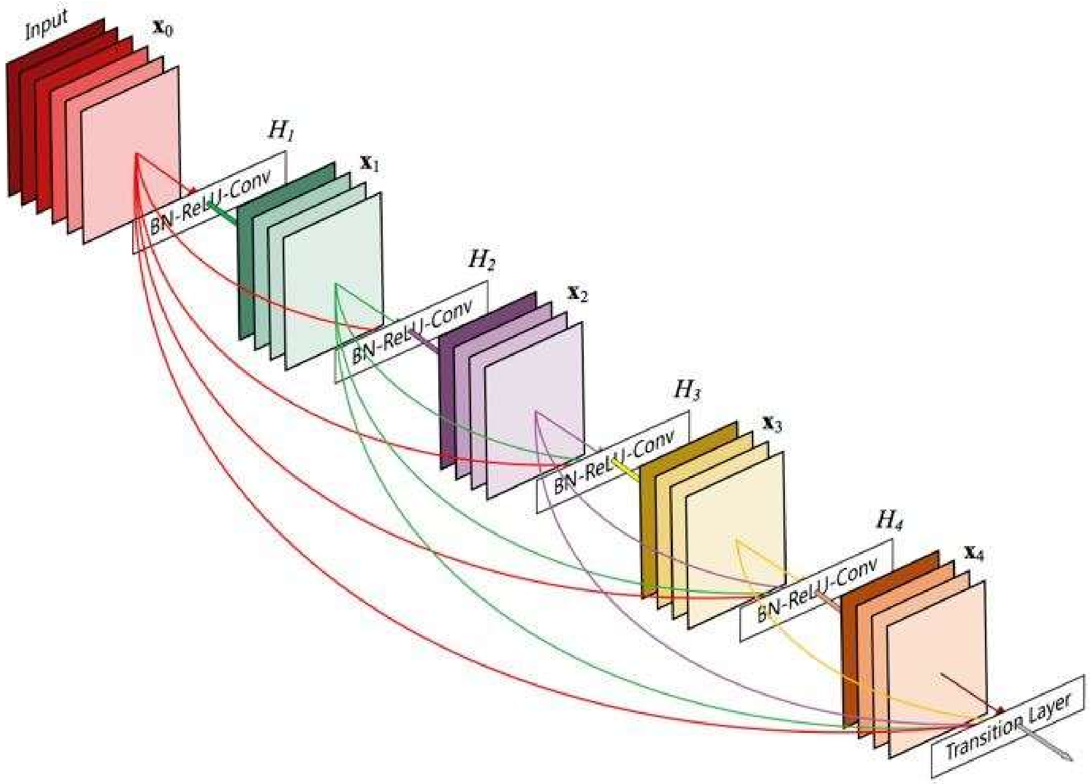

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

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

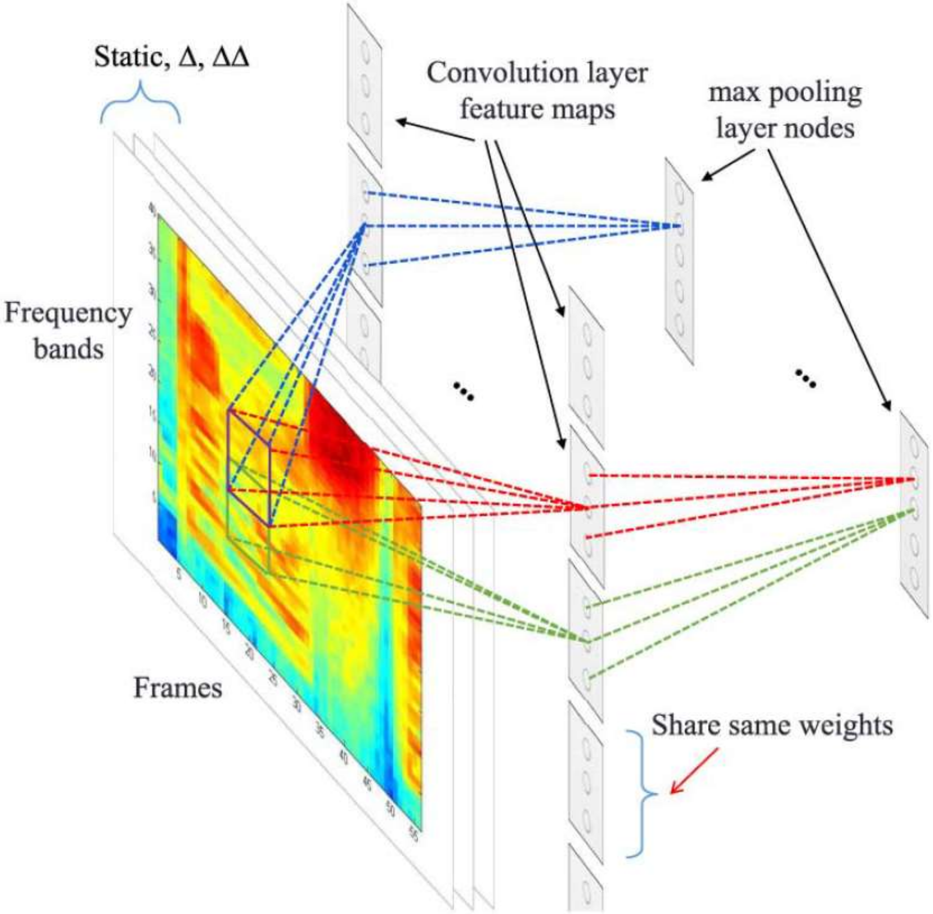
Densenet

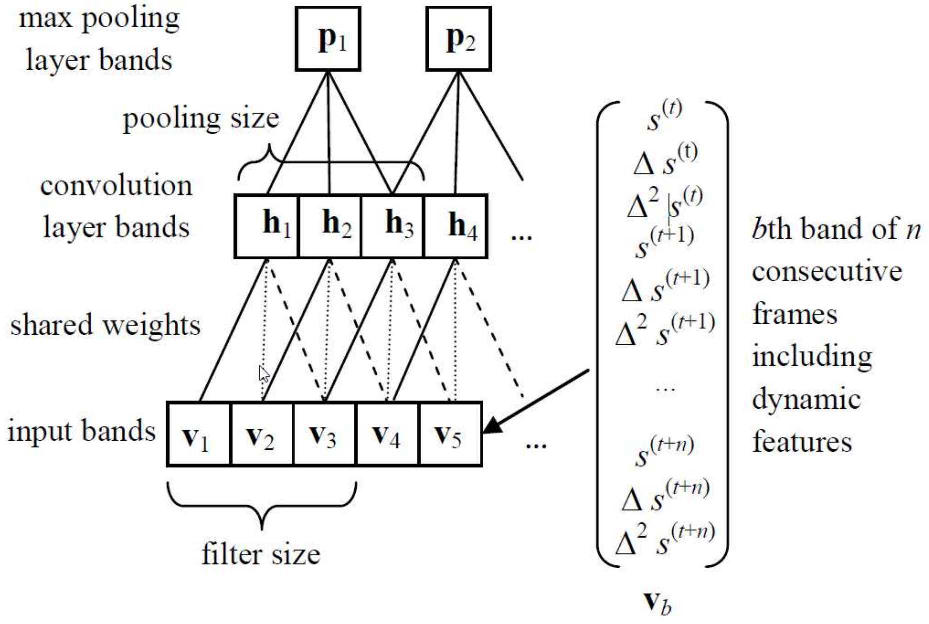


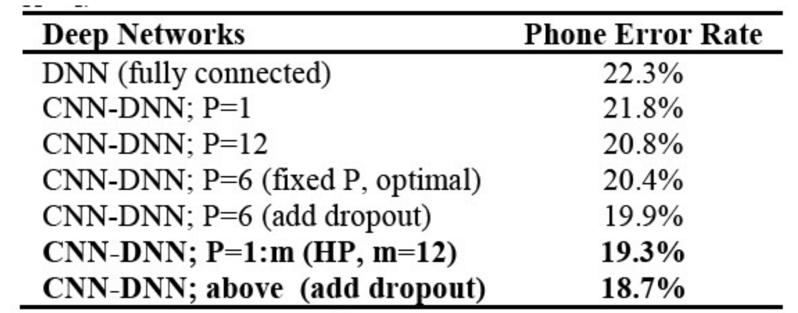
- All convolutional
- Each layer looks at the union of maps from all previous layers
  - Instead of just the set of maps from the immediately previous layer
- Was state of the art before I went for coffee one day
  - Wasn’t when I got back..

CNN for Automatic Speech Recognition

- Convolution over frequencies
- Convolution over time





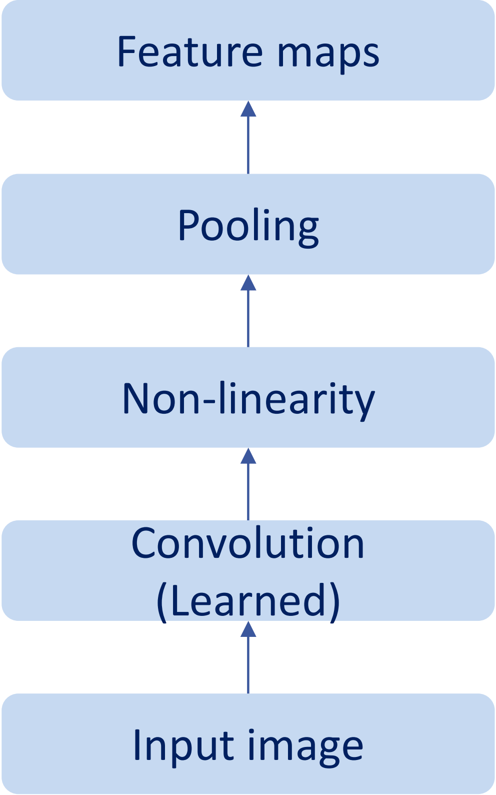


Deep Networks	Phone Error Rate
DNN (fully connected)	22.3%
CNN-DNN; P=1	21.8%
CNN-DNN; P=12	20.8%
CNN-DNN; P=6 (fixed P, optimal)	20.4%
CNN-DNN; P=6 (add dropout)	19.9%
CNN-DNN; P=1:m (HP, m=12)	**19.3%**
CNN-DNN; above (add dropout)	**18.7%**

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

CNN-Recap

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



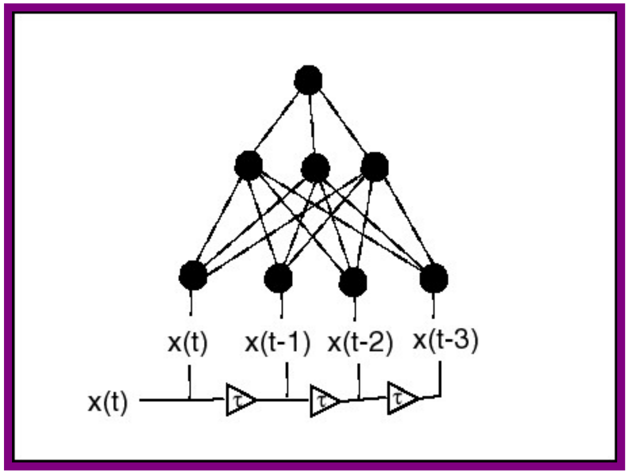
Feature maps

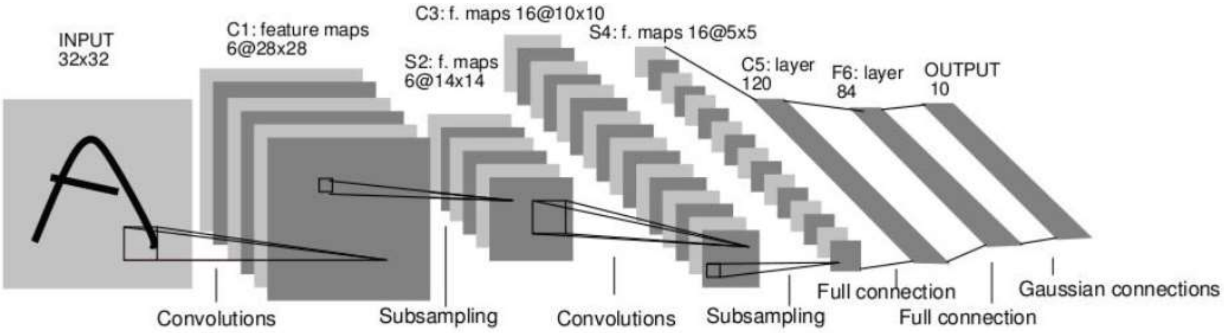
Pooling

Non-linearity

Convolution (Learned)

Input image


x(t)
x(t-1)
x(t-2)
x(t-3)
x(t)
τ
τ
τ


INPUT 32x32
C1: feature maps 6@28x28
S2: f. maps 6@14x14
C3: f. maps 16@10x10
S4: f. maps 16@5x5
C5: layer 120
F6: layer 84
OUTPUT 10
Convolutions
Subsampling
Convolutions
Subsampling
Full connection
Full connection
Gaussian connections