Deep Neural Networks
Convolutional Networks IV

Bhiksha Raj

Spring 2022

Outline

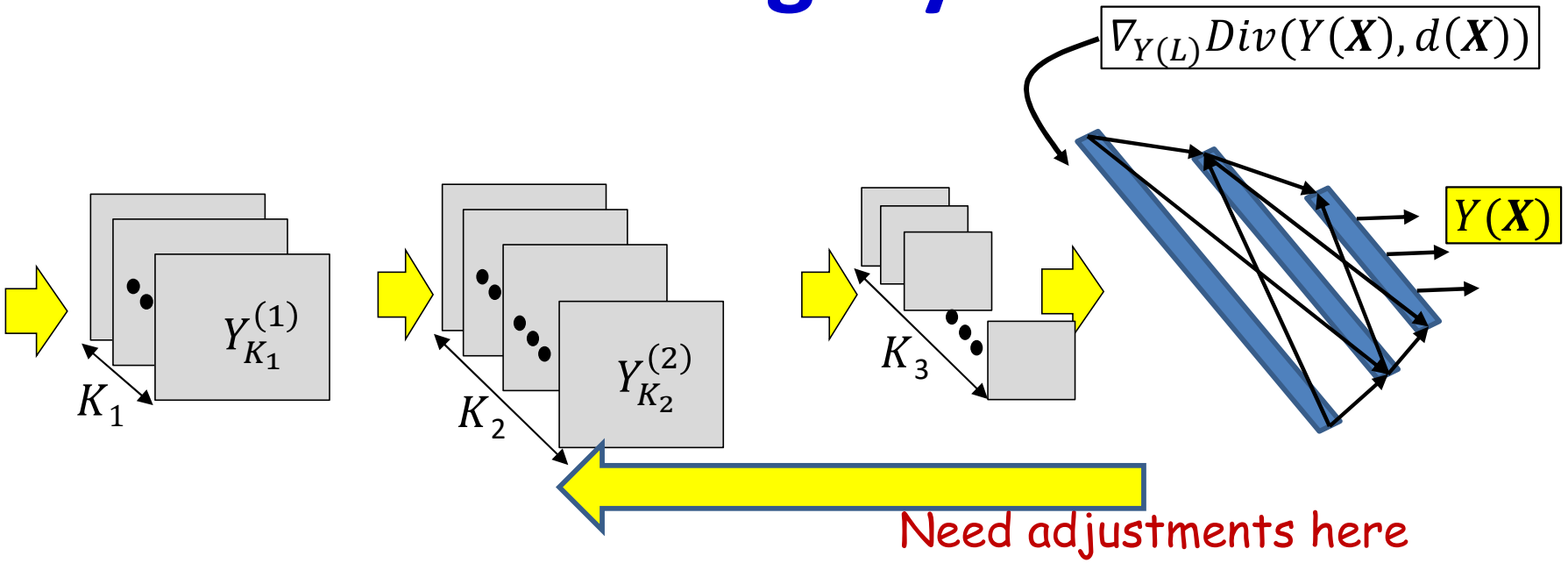
- Quick recap
- Back propagation through a CNN
- Modifications: Transposition, scaling, rotation and deformation invariance
- Segmentation and localization
- Some success stories
- Some advanced architectures
  - Resnet
  - Densenet
  - Transformers and self similarity

Story so far

- Shift-invariant 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 using CNNs (or TDNNs)

- These are “shared parameter” models that can be trained with variations of backprop

Backpropagation: Convolutional and Pooling layers



- For each training instance: First, a forward pass through the net
- Then the backpropagate the derivative of the divergence
- Regular backprop until the first “flat” layer
- Subsequent backpropagation from the flat MLP requires special consideration of
  - The shared computation in the convolution layers
  - The pooling layers

Backpropagation: Convolutional and Pooling layers

- **Required:**
  - **For convolutional layers**:
    - Given the derivatives for the output activation maps $Y(l)$, how to compute the derivatives w.r.t. the affine $Z(l)$ maps
    - Given the derivatives for the affine maps $Z(l)$ How to compute the derivative w.r.t. $Y(l-1)$ and $w(l)$
  - **For pooling layers**:
    - How to compute the derivative w.r.t. input layer $Y(l-1)$ given derivatives w.r.t. pooled output $Y(l)$

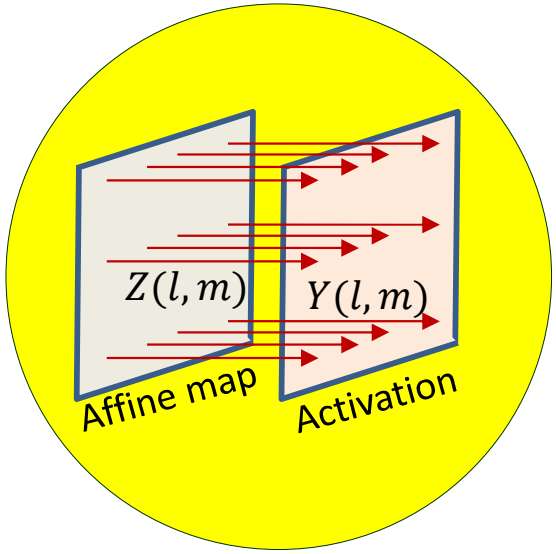
Backpropagation: Convolutional and Pooling layers

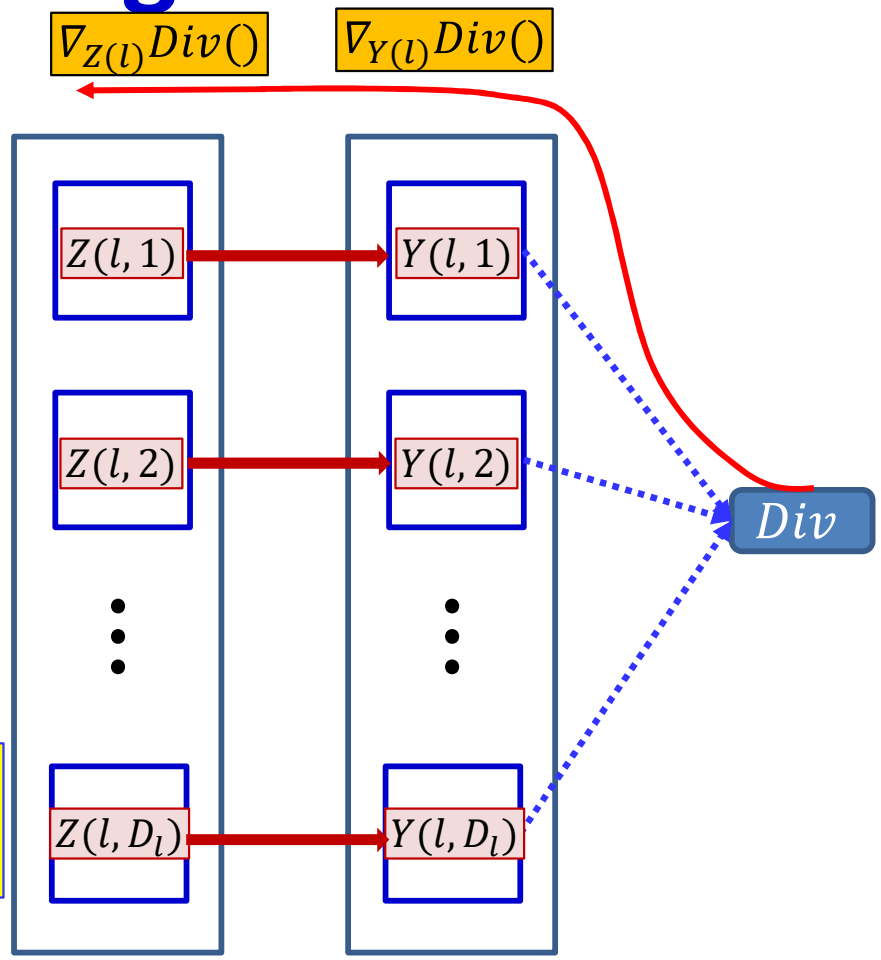
- **Required:**
  - **For convolutional layers**:
    - Given the derivatives for the output activation maps $Y(l)$, how to compute the derivatives w.r.t. the affine $Z(l)$ maps
    - Given the derivatives for the affine maps $Z(l)$ How to compute the derivative w.r.t. $Y(l-1)$ and $w(l)$
  - **For pooling layers**:
    - How to compute the derivative w.r.t. input layer $Y(l-1)$ given derivatives w.r.t. pooled output $Y(l)$

Backpropagation: Convolutional and Pooling layers

- **Required:**
  - **For convolutional layers**:
    - Given the derivatives for the output activation maps $Y(l)$, how to compute the derivatives w.r.t. the affine $Z(l)$ maps
    - Given the derivatives for the affine maps $Z(l)$ How to compute the derivative w.r.t. $Y(l-1)$ and $w(l)$
  - **For pooling layers**:
    - How to compute the derivative w.r.t. input layer $Y(l-1)$ given derivatives w.r.t. pooled output $Y(l)$

Backpropagating through the activation





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

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

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

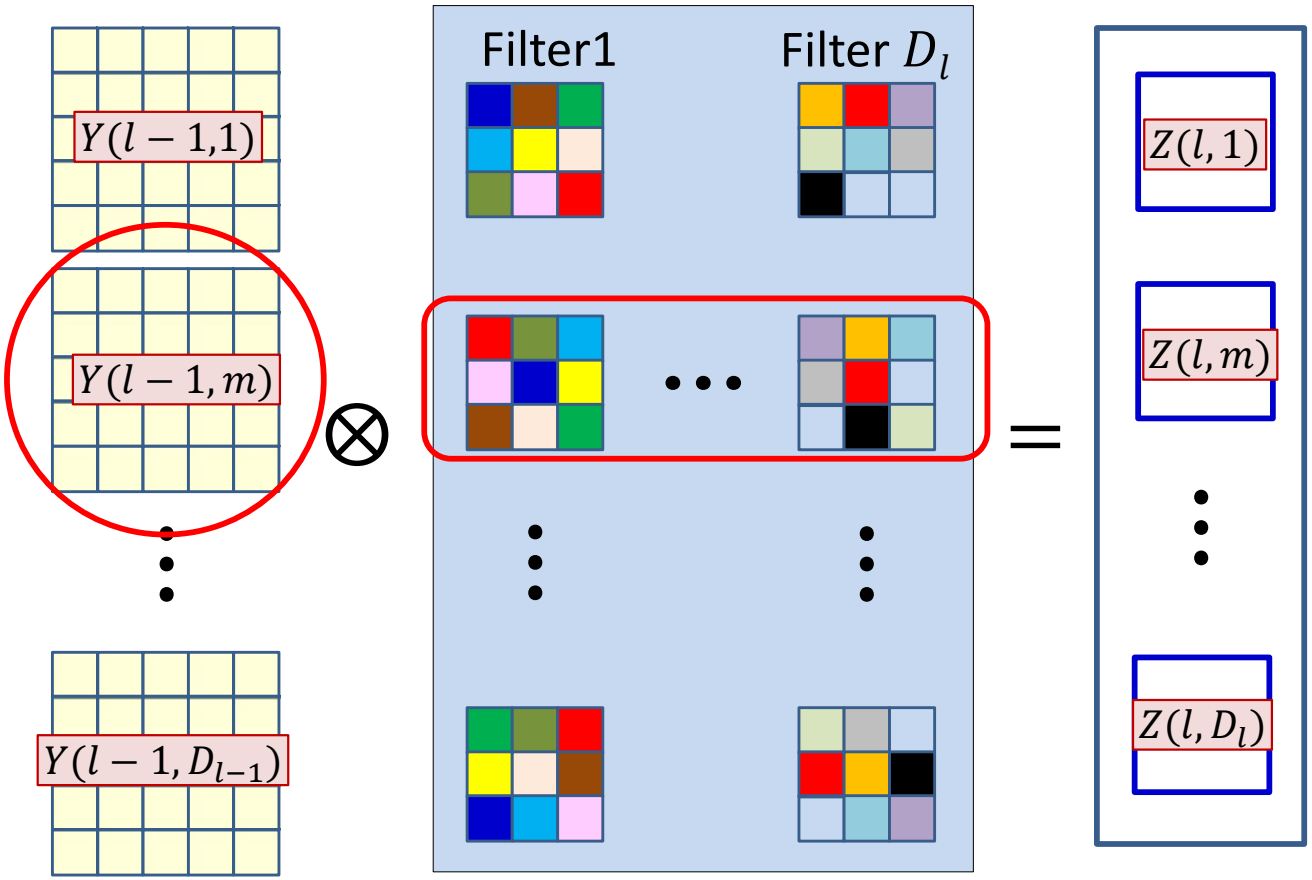
$$\frac{dDiv}{dz(l,m,x,y)} = \frac{dDiv}{d\,y(l,m,x,y)} f'(z(l,m,x,y))$$

  - Simple component-wise computation

Backpropagation: Convolutional and Pooling layers

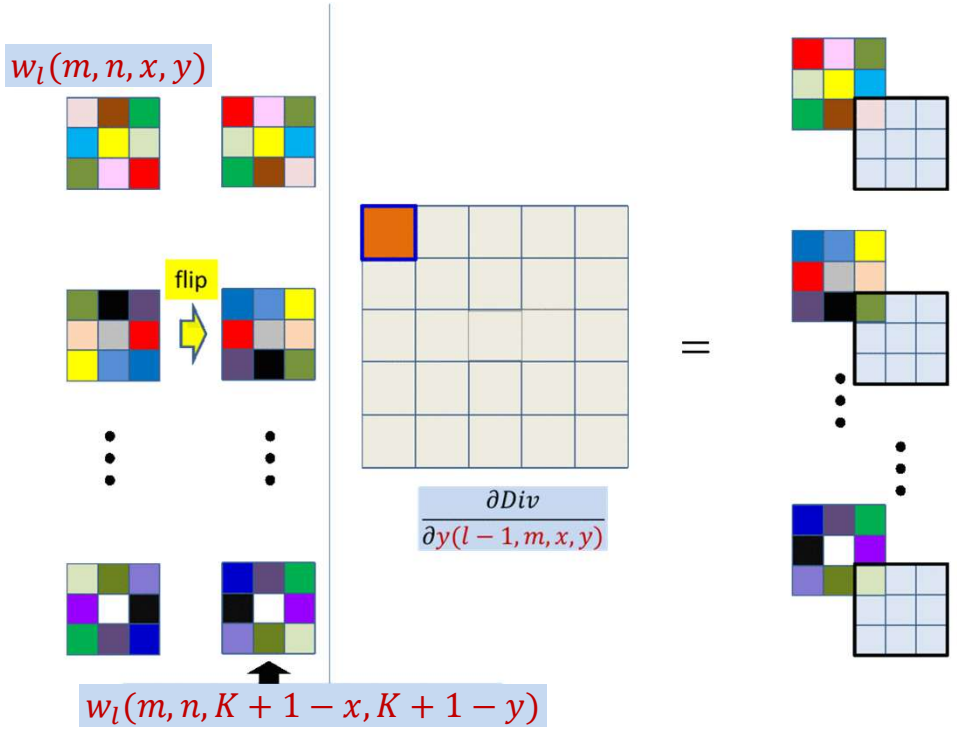
- **Required:**
  - **For convolutional layers**:
    - Given the derivatives for the output activation maps $Y(l)$, how to compute the derivatives w.r.t. the affine $Z(l)$ maps
    - Given the derivatives for the affine maps $Z(l)$ How to compute the derivative w.r.t. $Y(l-1)$ and $w(l)$
  - **For pooling layers**:
    - How to compute the derivative w.r.t. input layer $Y(l-1)$ given derivatives w.r.t. pooled output $Y(l)$

The derivatives for $Y(l-1,m)$



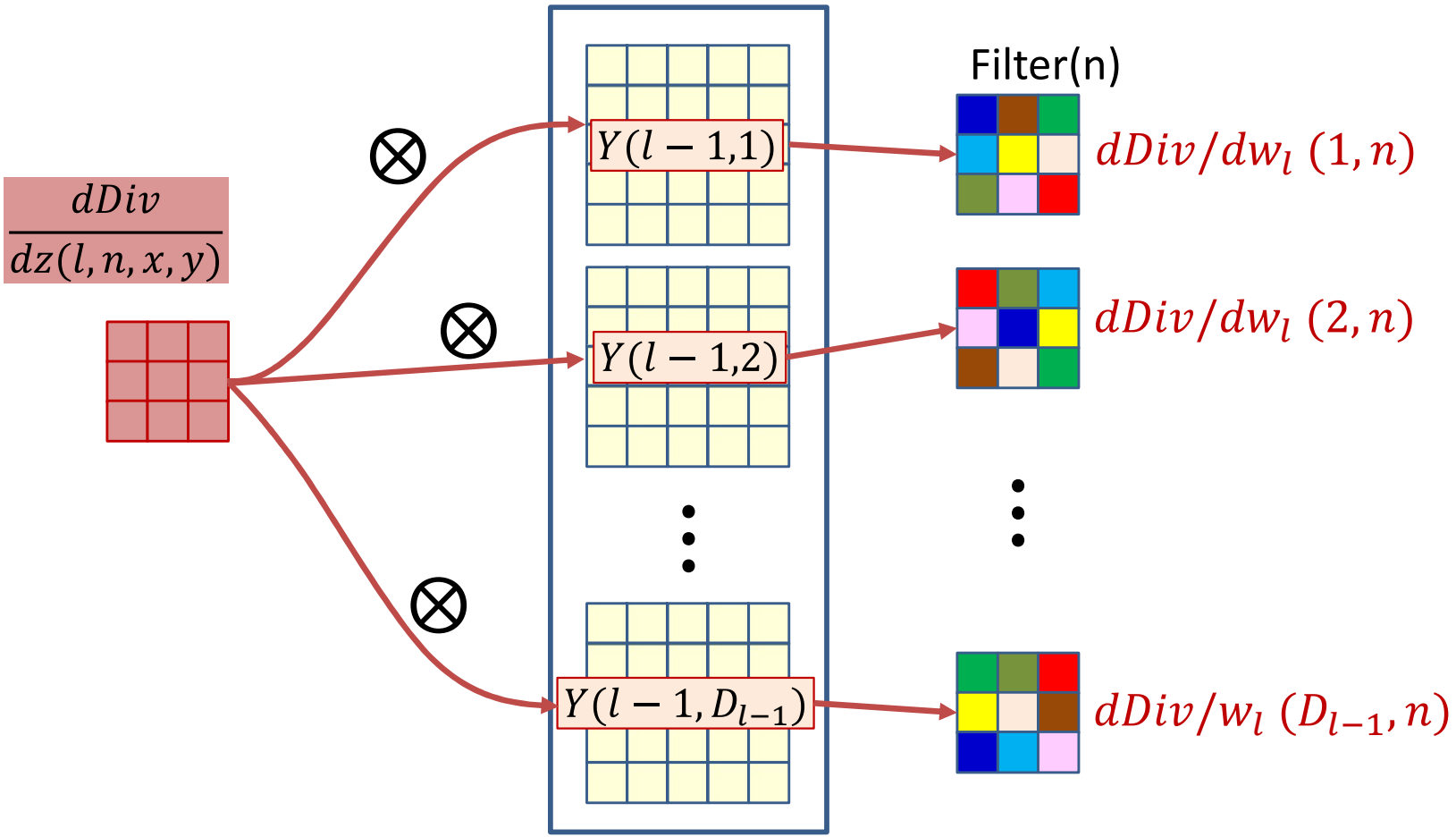
- The D_l affine maps are produced by convolving with D_l filters
- The m^{th} Y map always convolves the m^{th} plane of the filters
- The derivative for the m^{th} Y map will invoke the m^{th} plane of *all* the filters

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



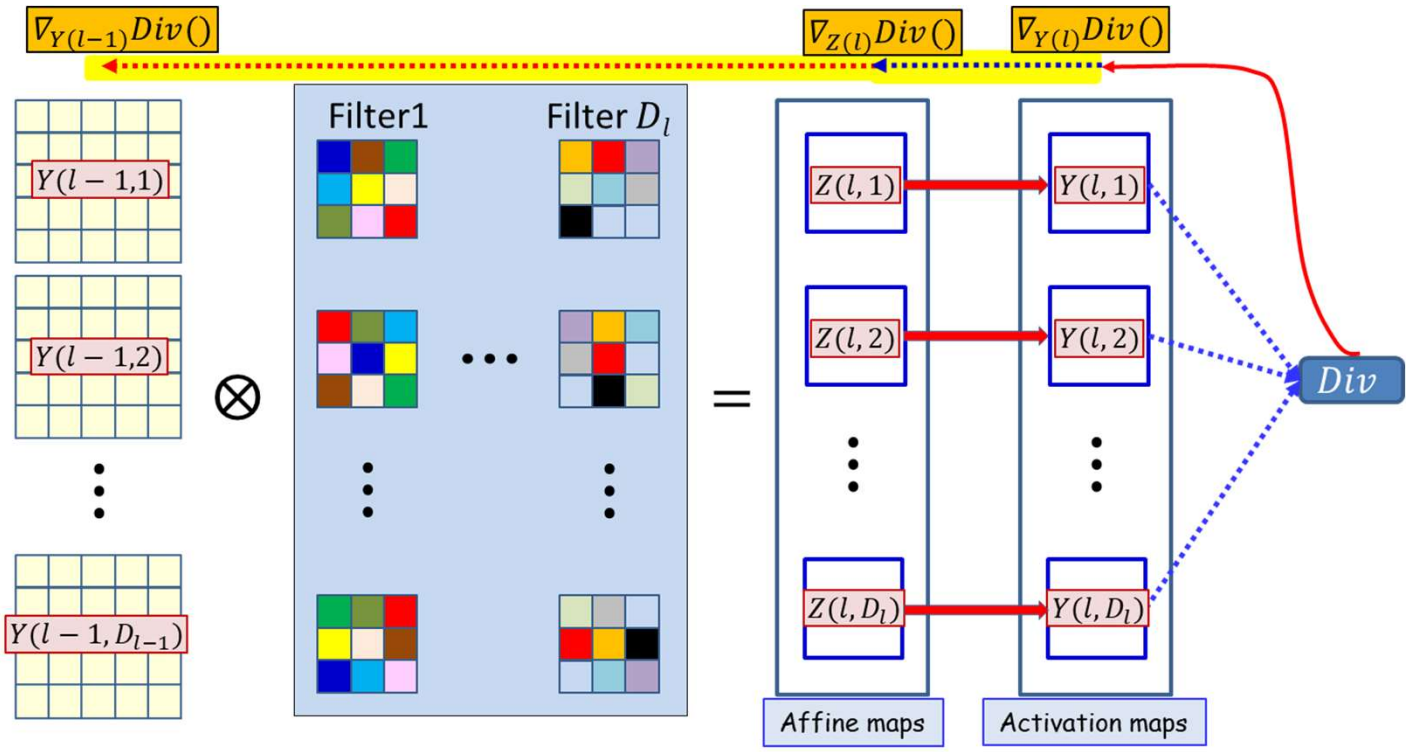
- This is just a convolution of the zero-padded $\frac{\partial Div}{\partial z(l,n,x,y)}$ maps by the transposed and flipped filter
  - After zero padding it first with $K-1$ zeros on every side

The filter derivative



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

Backpropagation: Convolutional layers



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

Backpropagation: Convolutional and Pooling layers

- **Required:**

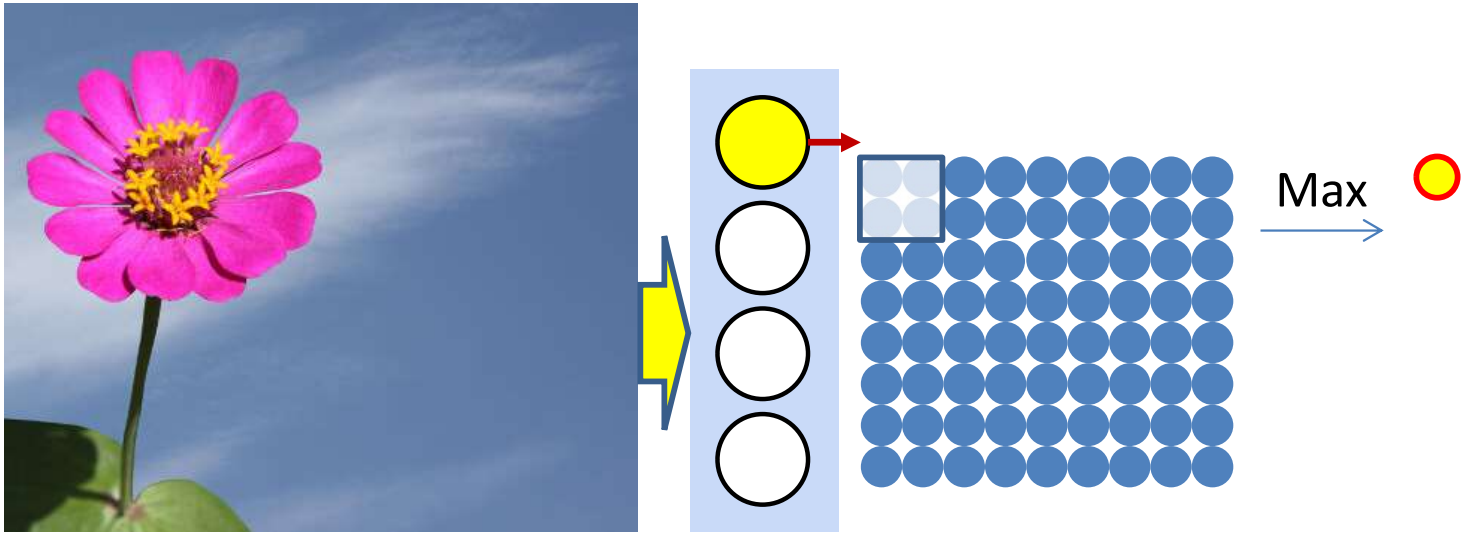
✓ **For convolutional layers**:

- Given the derivatives for the output activation maps $Y(l)$, how to compute the derivatives w.r.t. the affine $Z(l)$ maps
- Given the derivatives for the affine maps $Z(l)$ How to compute the derivative w.r.t. $Y(l-1)$ and $w(l)$

– **For pooling layers**:

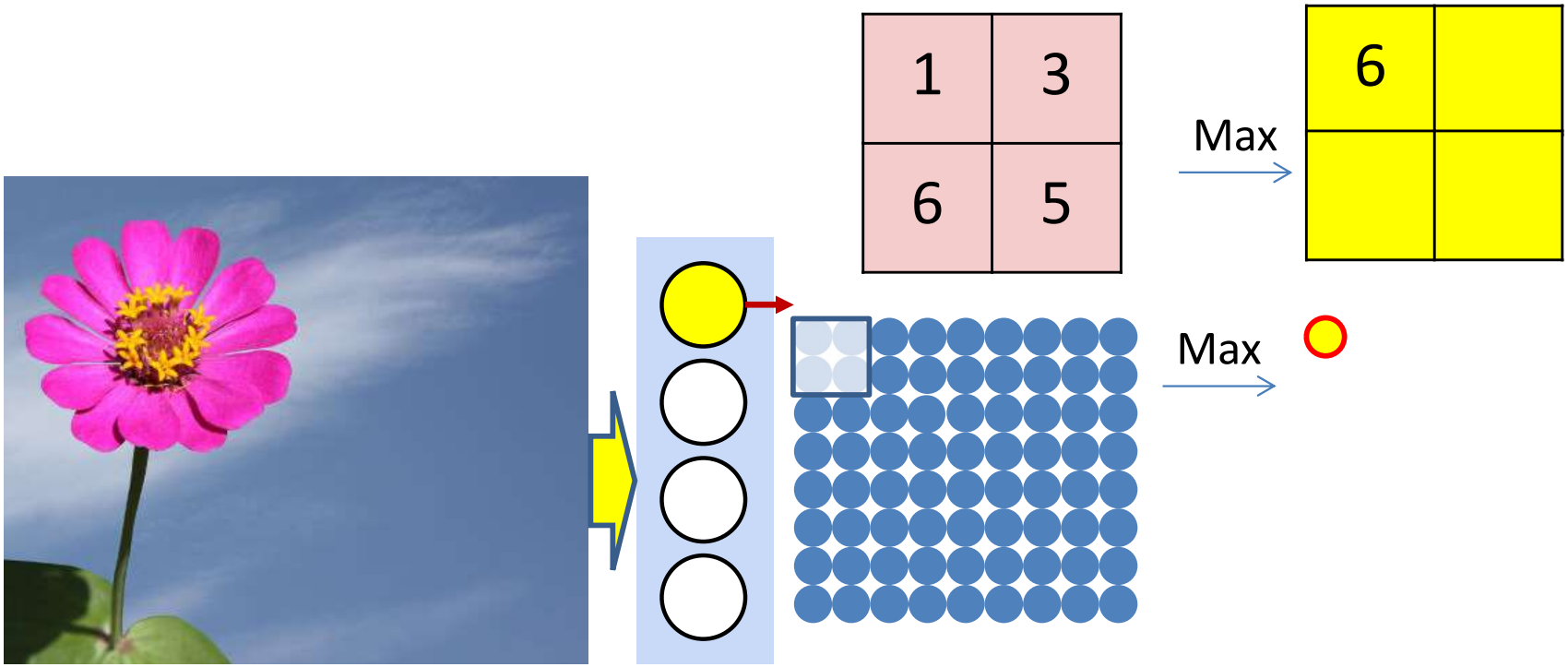
- How to compute the derivative w.r.t. input layer $Y(l-1)$ given derivatives w.r.t. pooled output $Y(l)$

Pooling



- Pooling “pools” groups of values to reduce jitter-sensitivity
  - Scanning with a “pooling” filter
- The most common pooling is “Max” pooling

Max Pooling

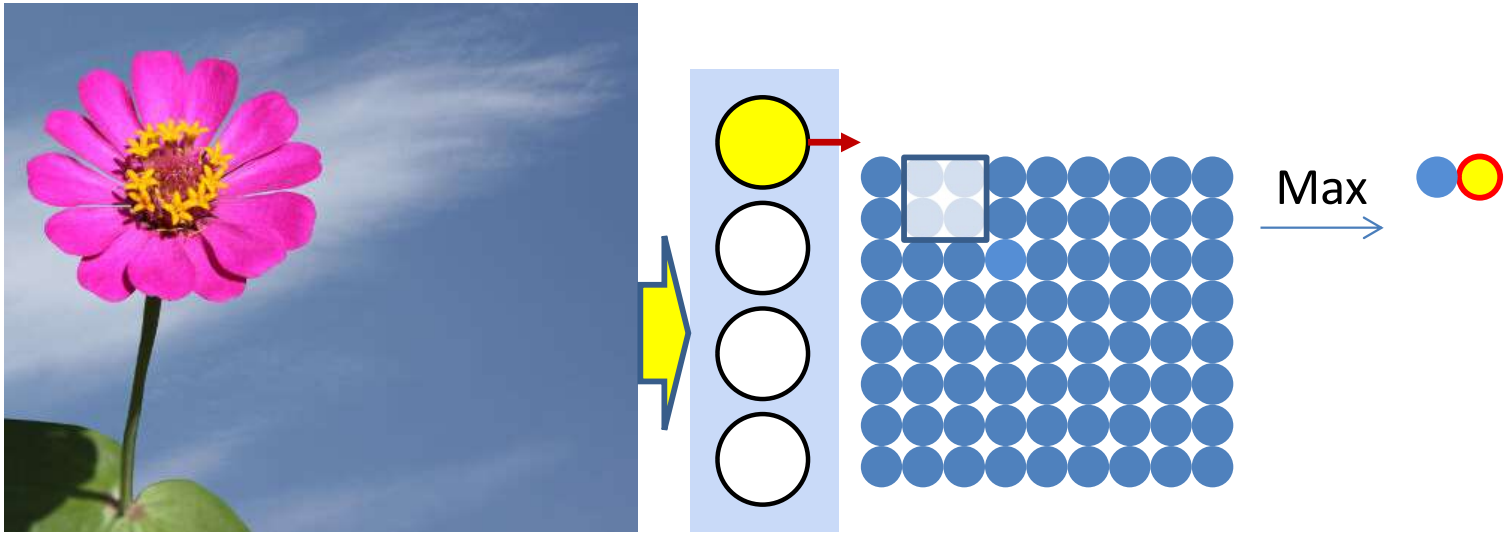


- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

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

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

Max pooling

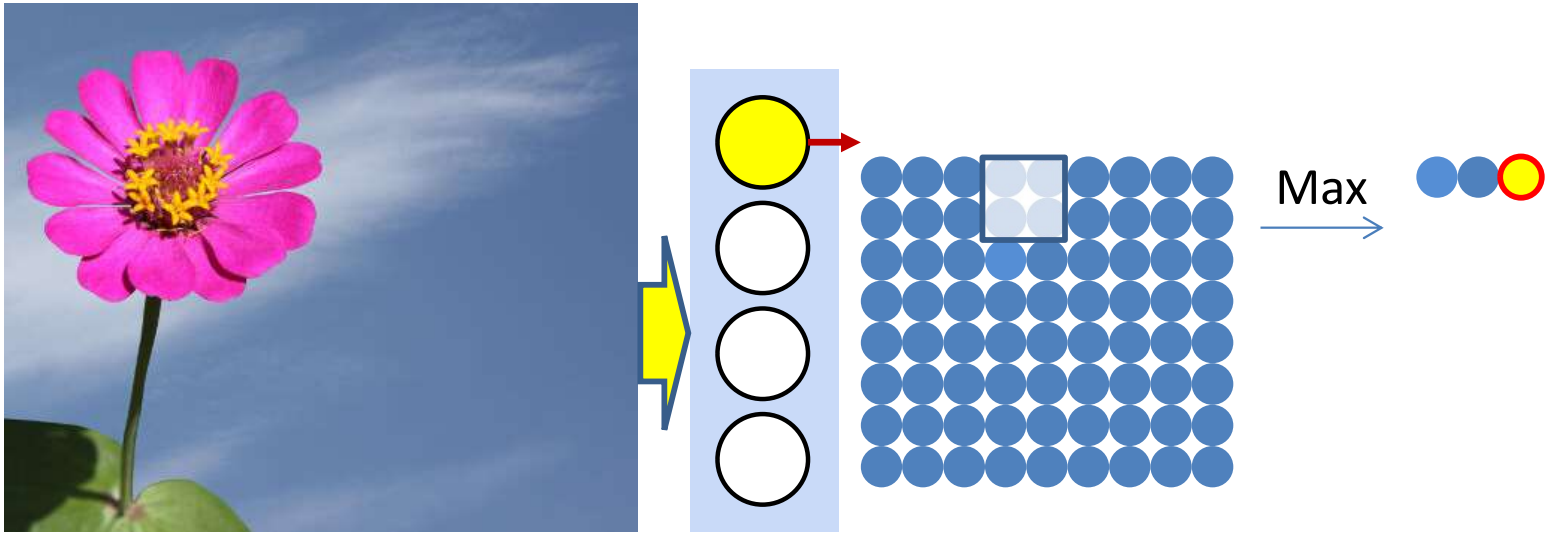


- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

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

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

Max pooling

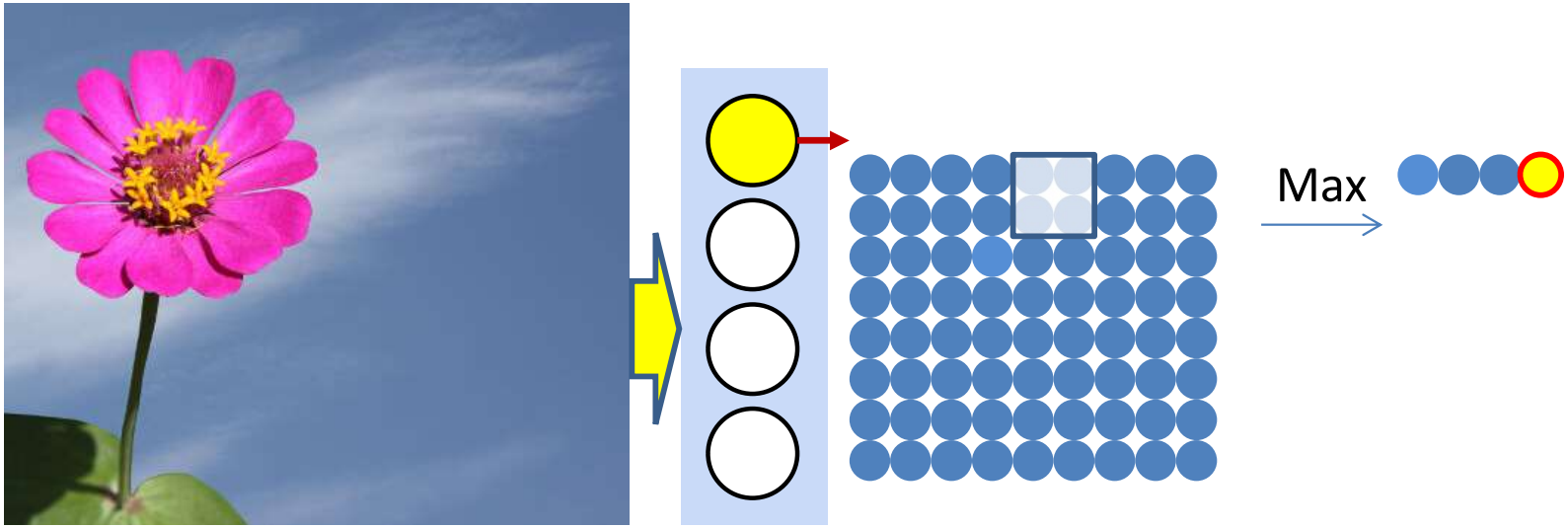


- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

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

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

Max pooling

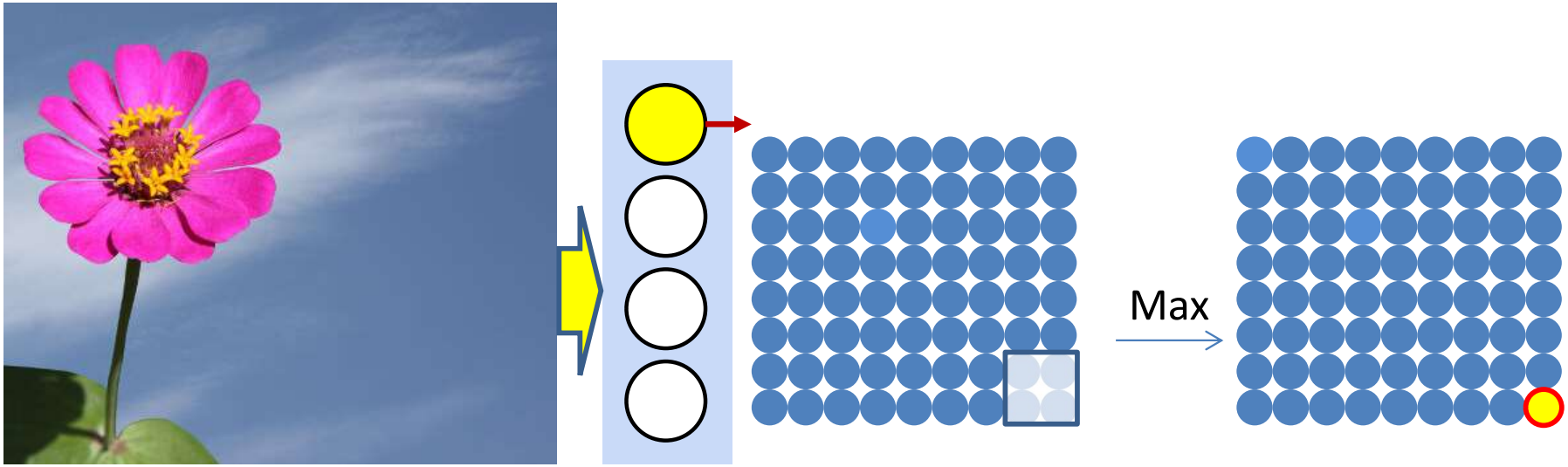


- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

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

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

Max pooling

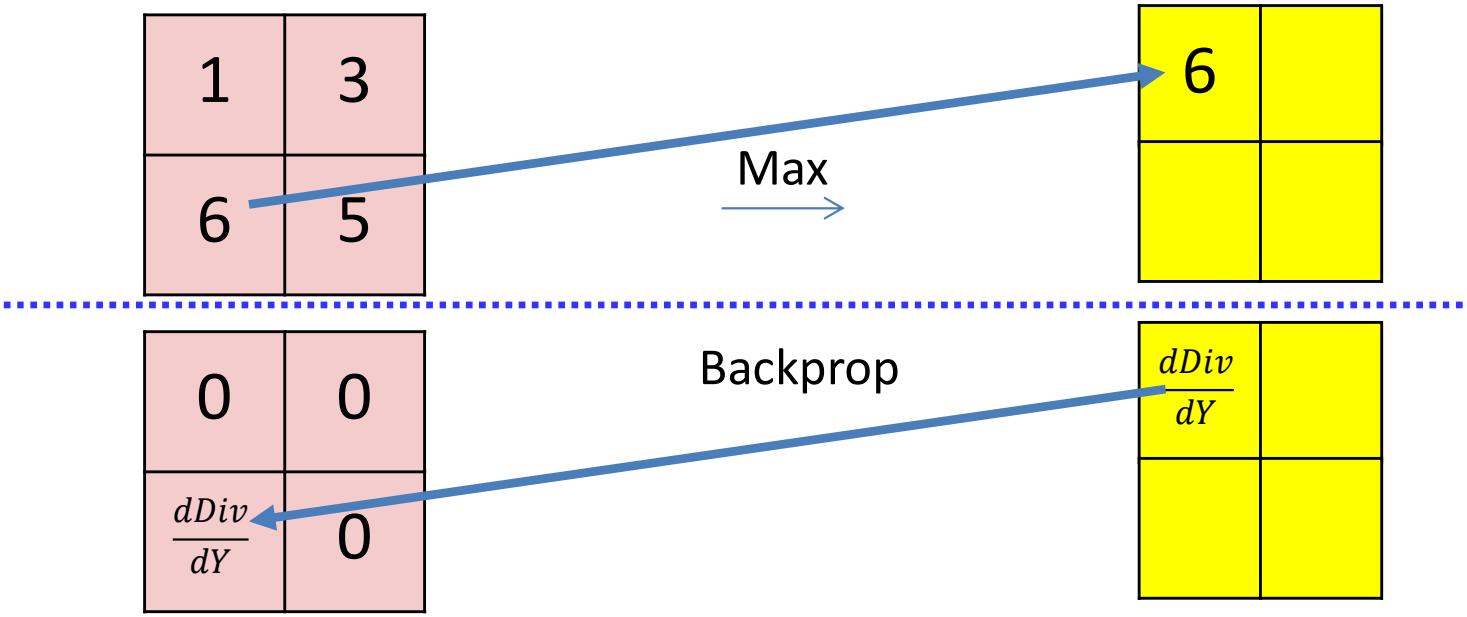


- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

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

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

Derivative of Max pooling



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

- Max pooling selects the largest from a pool of elements

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

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

Max Pooling layer at layer l

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

Max pooling

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

Derivative of max pooling layer at layer l

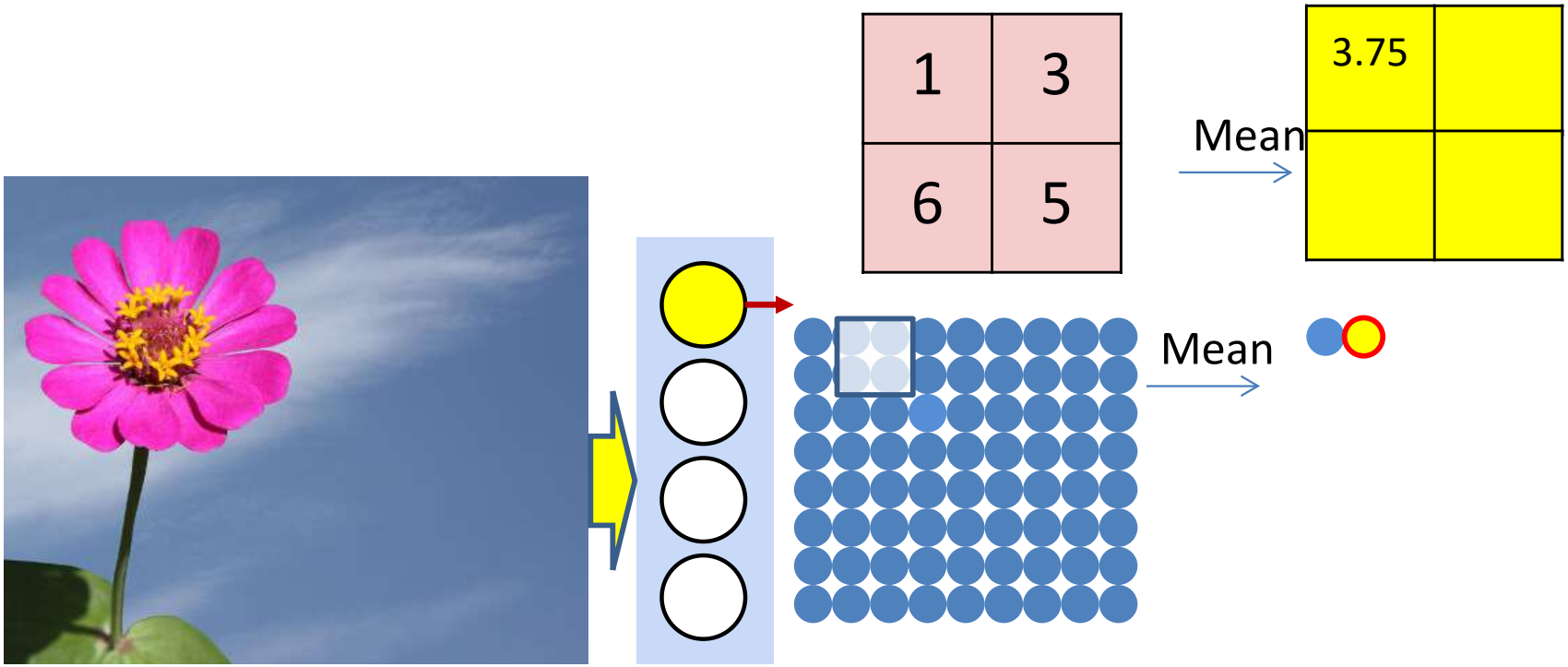
a) Performed separately for every map (j).
*) *Not* combining multiple maps within a single max operation.
b) Keeping track of location of max

Max pooling

```
dy(:,:,:) = zeros(D_l x W_l x H_l)
for j = 1:D_l
    for x = 1:W_l
        for y = 1:H_l
            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

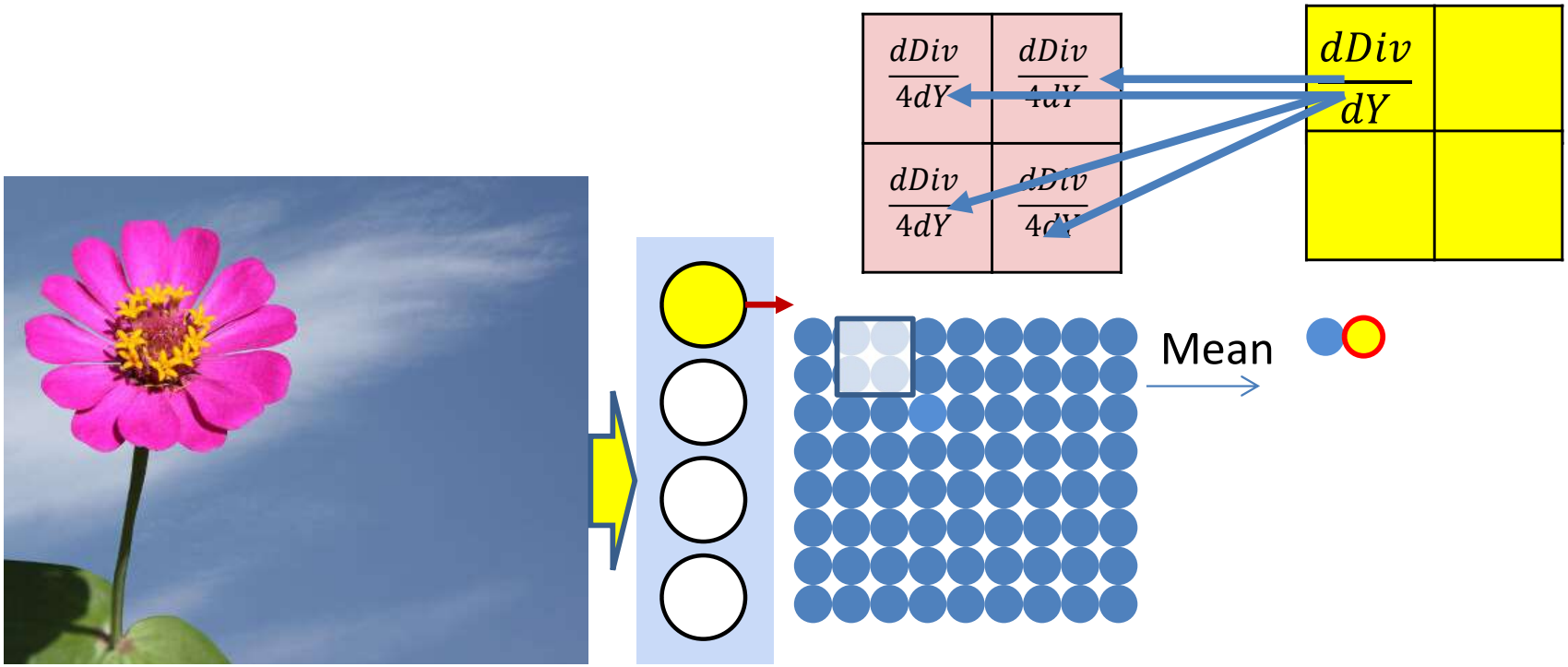
Mean pooling



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

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

Derivative of mean pooling



- The derivative of mean pooling is distributed over the pool

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

Mean Pooling layer at layer l

Mean pooling

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

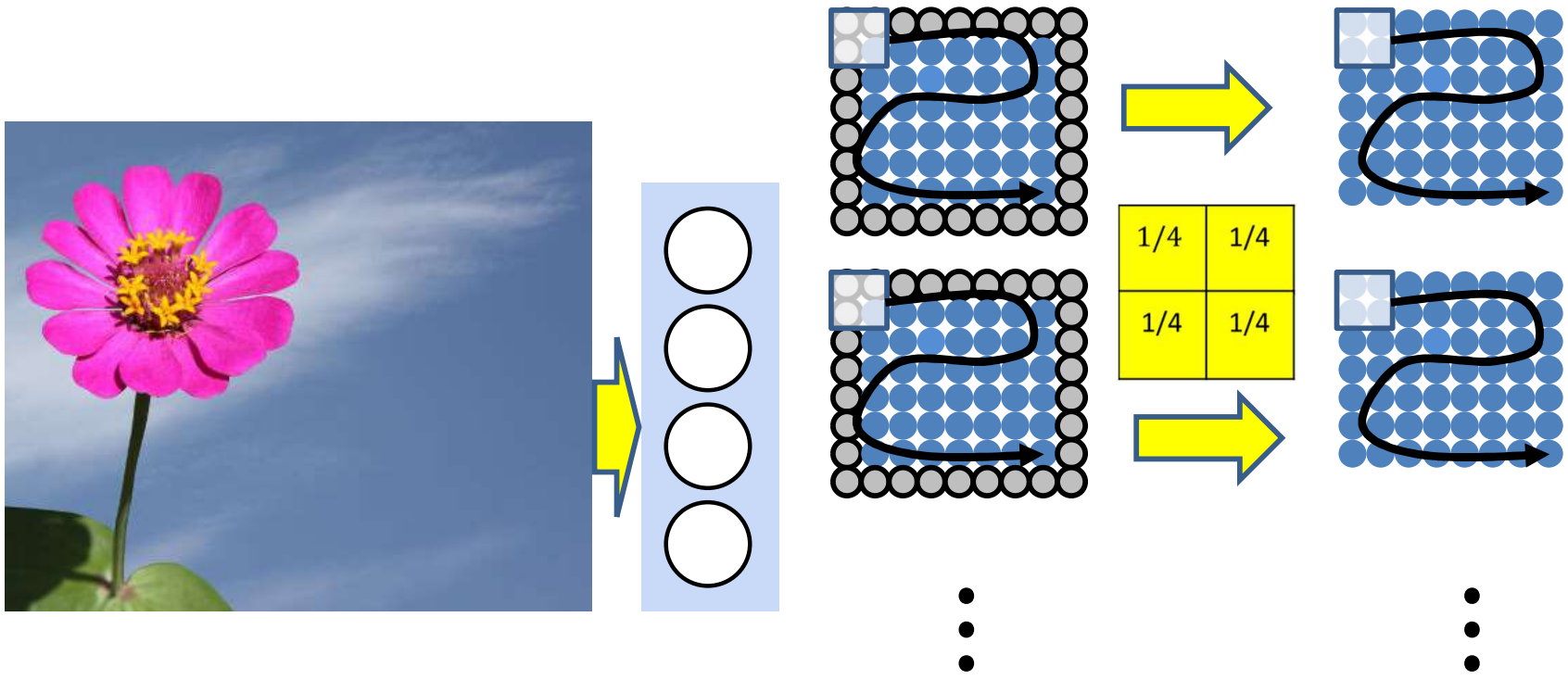
Derivative of mean pooling layer at layer l

Mean pooling

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

"+=" because adjacent windows may overlap

Derivative of mean pooling



- This is actually the *channel-wise* convolution of $dy(l, m)$ by a "uniform" filter
  - After zero-padding on every side by (N-1) rows/columns for an $N \times N$ mean pooling filter
  - All values in the filter are $\frac{1}{K^2}$

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. $Y(l)$

Poll 1

Poll 1

When backpropagating through a MAXpooling layer, derivatives from the pooling output backpropagate only to the position of the largest input within the input pooling window for that output

- **True**
- False

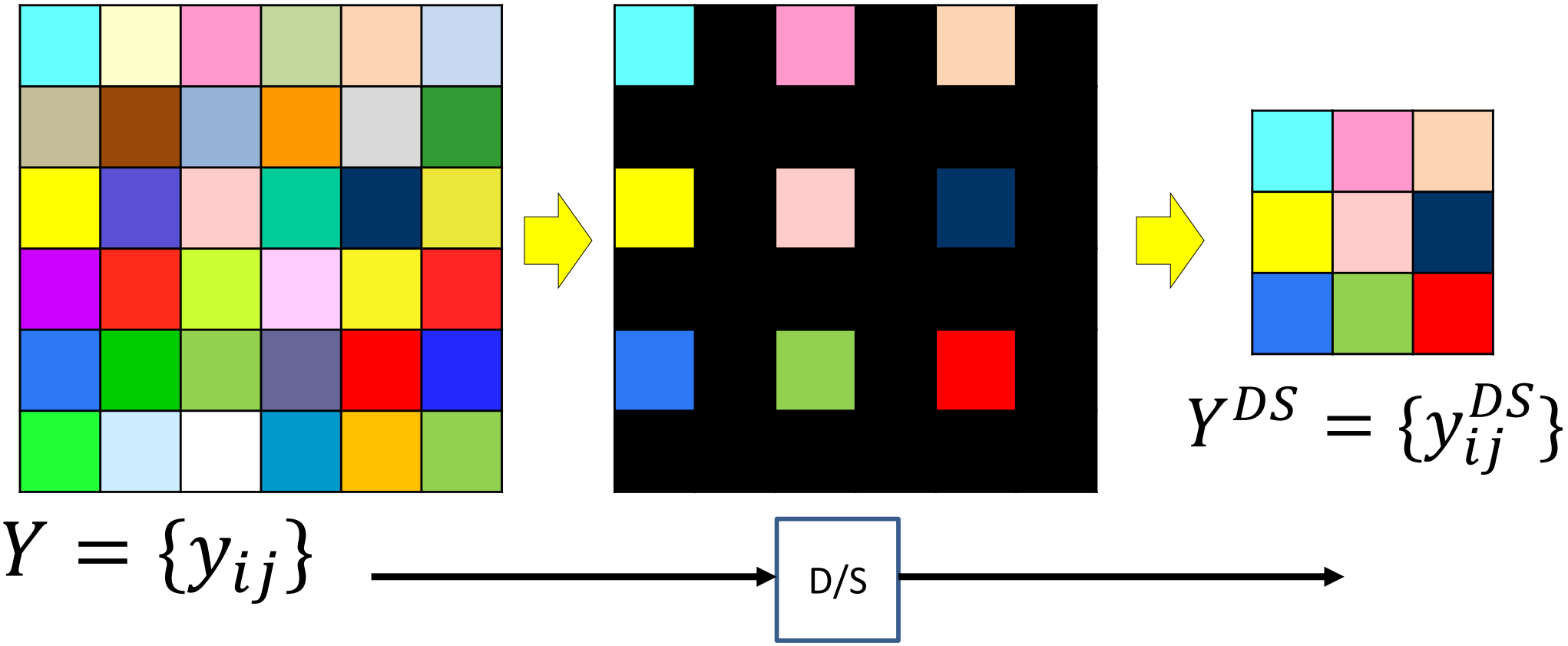
When backpropagating through a meanpooling layer, derivatives from the pooling output are distributed uniformly over the input pooling window for that output

- **True**
- False

Recap

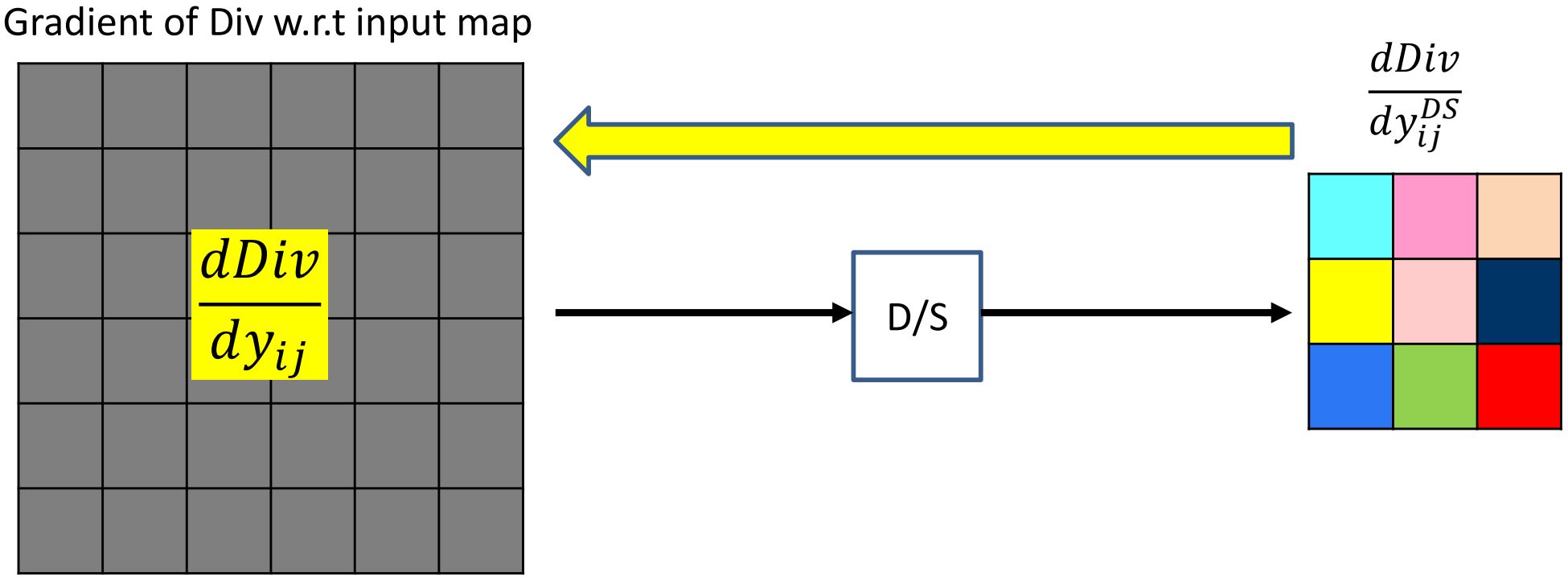
- Upsampling and downsampling layers can increase or decrease the size of the map
- Upsampling followed by convolution can be viewed as convolution with a *fractional* stride
- Convolution followed by downsamping can be viewed as convolution with a stride greater than 1
- How do we backpropagate through upsampling and downsampling layers?

Recap: The Downsampling Layer



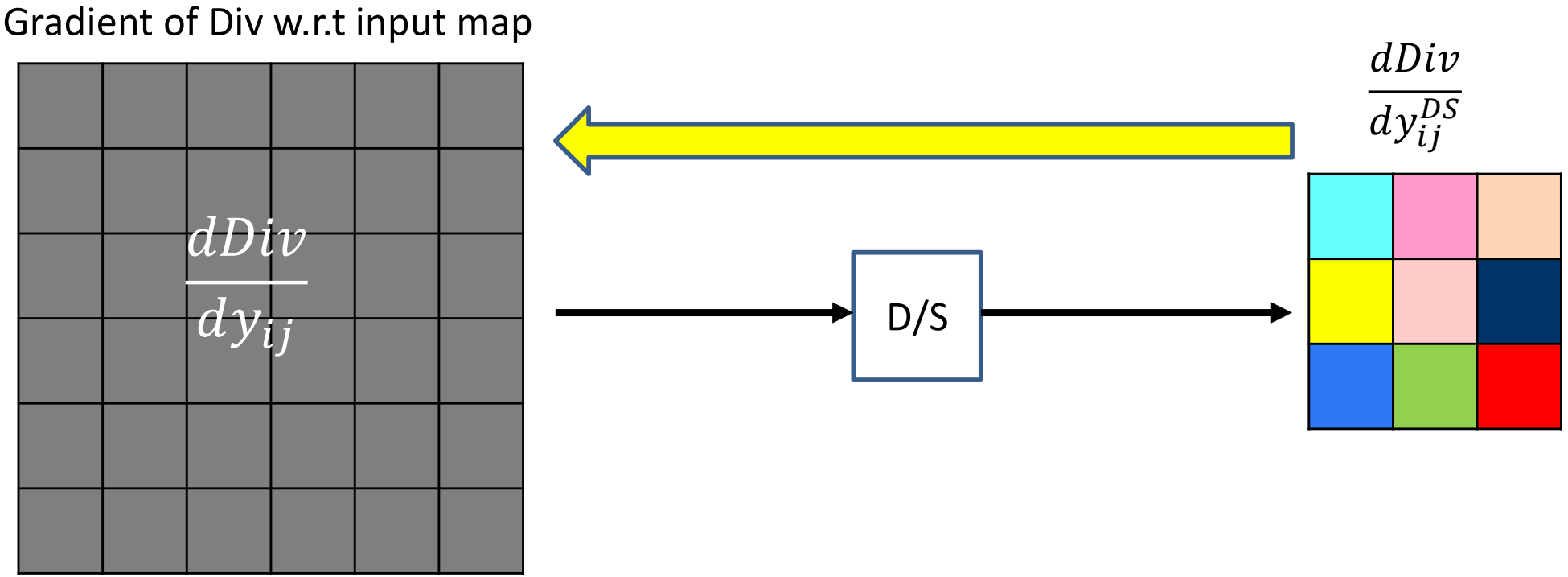
- A *downsampling* layer simply "drops" $S - 1$ of S rows and columns for every map in the layer
  - Effectively reducing the size of the map by factor S in every direction

The derivative size rule



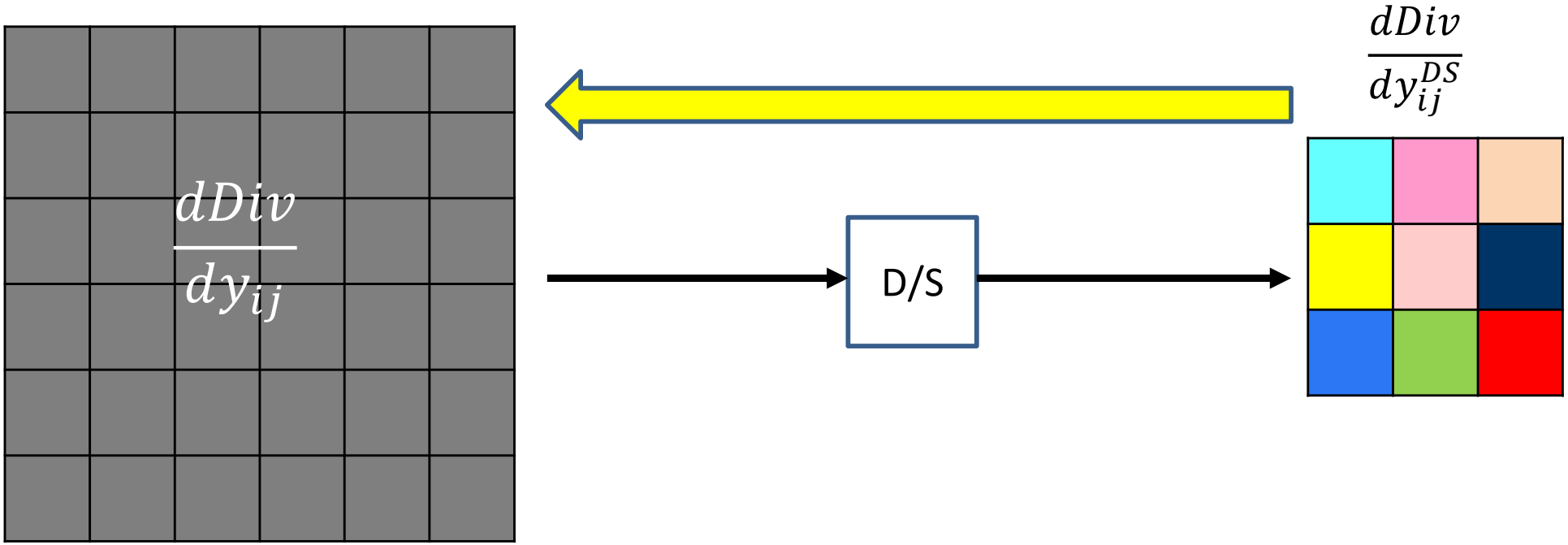
- Important note: the gradient of the divergence with respect to any variable will be the same size as the variable
  - For the input maps of a D/S layer, they will be the same size as the original input maps, regardless of the size of the output

Backprop through D/S layer



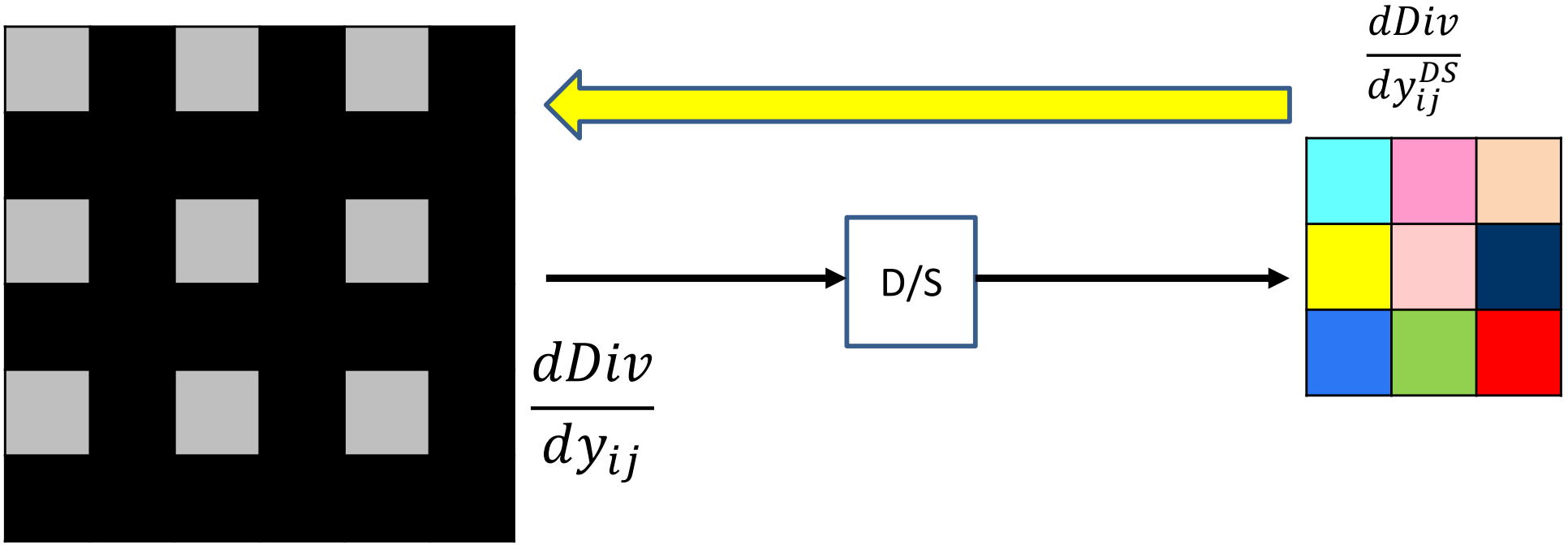
- **Backpropagation**: Given the derivative of the divergence with respect to the elements of the *output* of the downsampling, compute derivatives with respect to every element of the *input* to the down sampling

Backprop through D/S layer



- Step 1: Allocate a map *of the size of the input* that was downsampled
  - This information must be retained, or derived from the known size of the outcome of the computation of previous layers

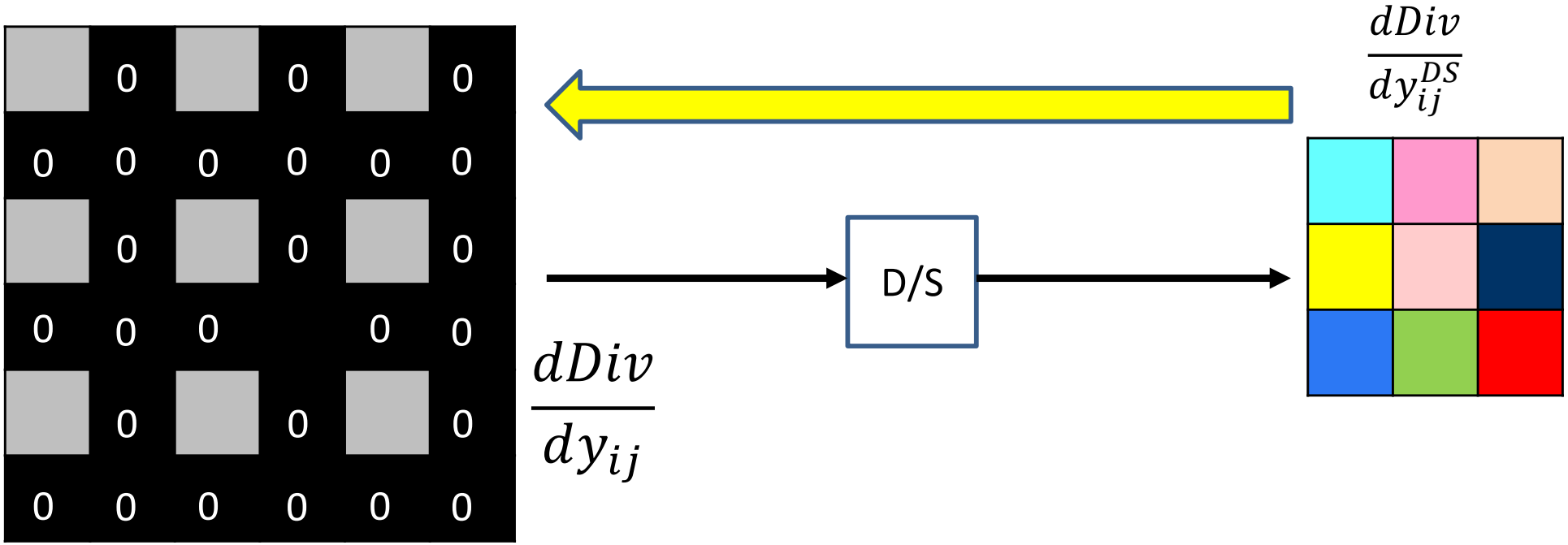
Backprop through the D/S layer



- Step 1: Allocate a map *of the size of the input* that was downsampled
  - This information must be retained, or derived from the known size of the outcome of the computation of previous layers

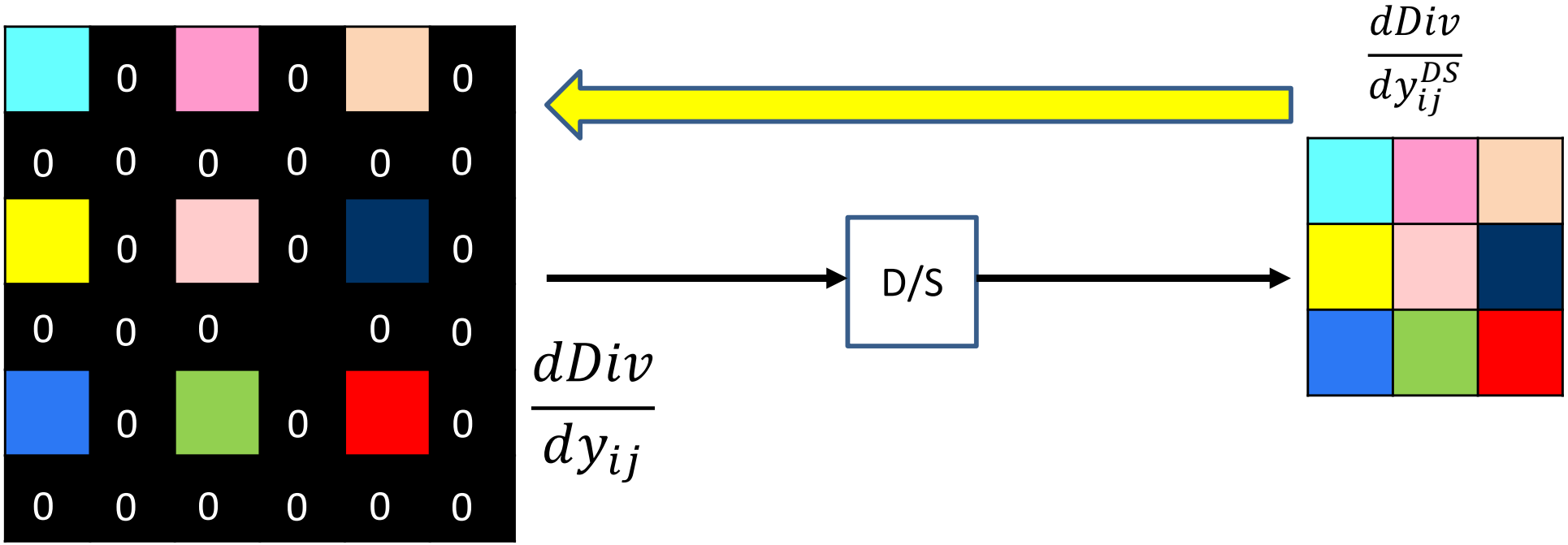
- Step 2: The "deleted" values (blackened) do not affect the output

Backprop through the D/S layer



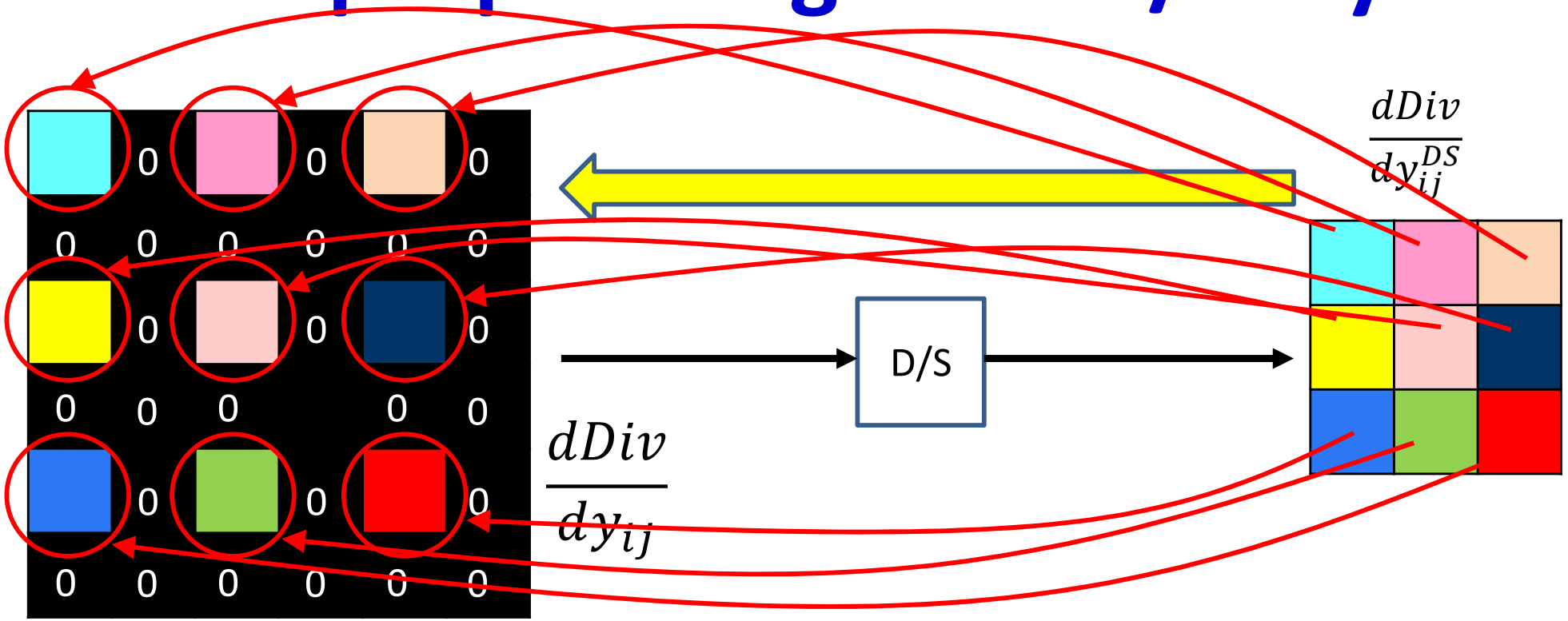
- Step 1: Allocate a map *of the size of the input* that was downsampled
  - This information must be retained, or derived from the known size of the outcome of the computation of previous layers
- Step 2: The "deleted" values (blackened) do not affect the output
  - **The derivative with respect to these elements is 0**

Backprop through the D/S layer



- **Step 3:** The remaining values are identical in the original and downsampled maps in the forward pass
  - The divergence derivatives too will be identical

Backprop through the D/S layer



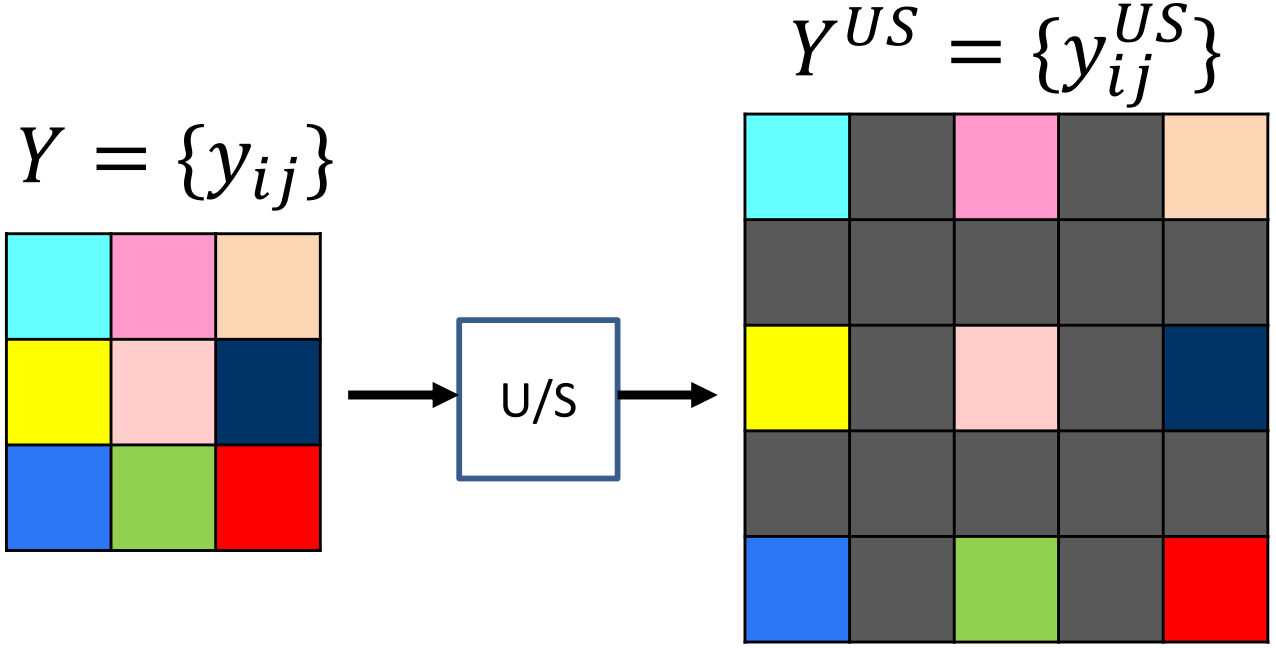
- Step 3: The remaining values are identical in the original and downsampled maps in the forward pass
  - Their derivatives too will be identical

- **Simply copy the derivatives for the output over to the appropriate location of the input**

Backprop through D/S pseudocode

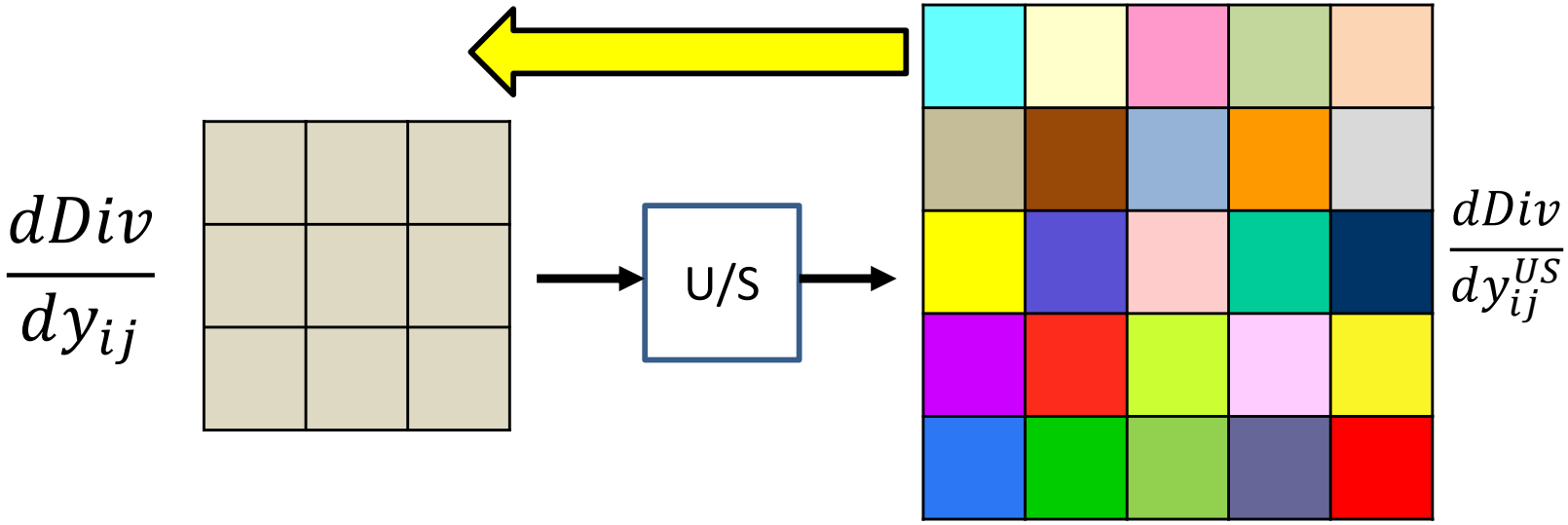
```
# H and W are the height and width of the input
# to the downsampling layer in the forward pass
# S is the stride in the forward pass
# dz contains the divergence derivative for the D/S
#    output z in the forward pass

function dy = backprop_through_DS(dz, S, H, W)
  #c = number of channels in dz
  dy = zeros(c,H,W)  # preallocate to right size and set to 0
  for i = 1:width(z)
    for j = 1:height(z)
      dy(:,(i-1)S+1, (j-1)S+1) = dz(:,i,j)
  return dy
```

Recap: The Upsampling Layer



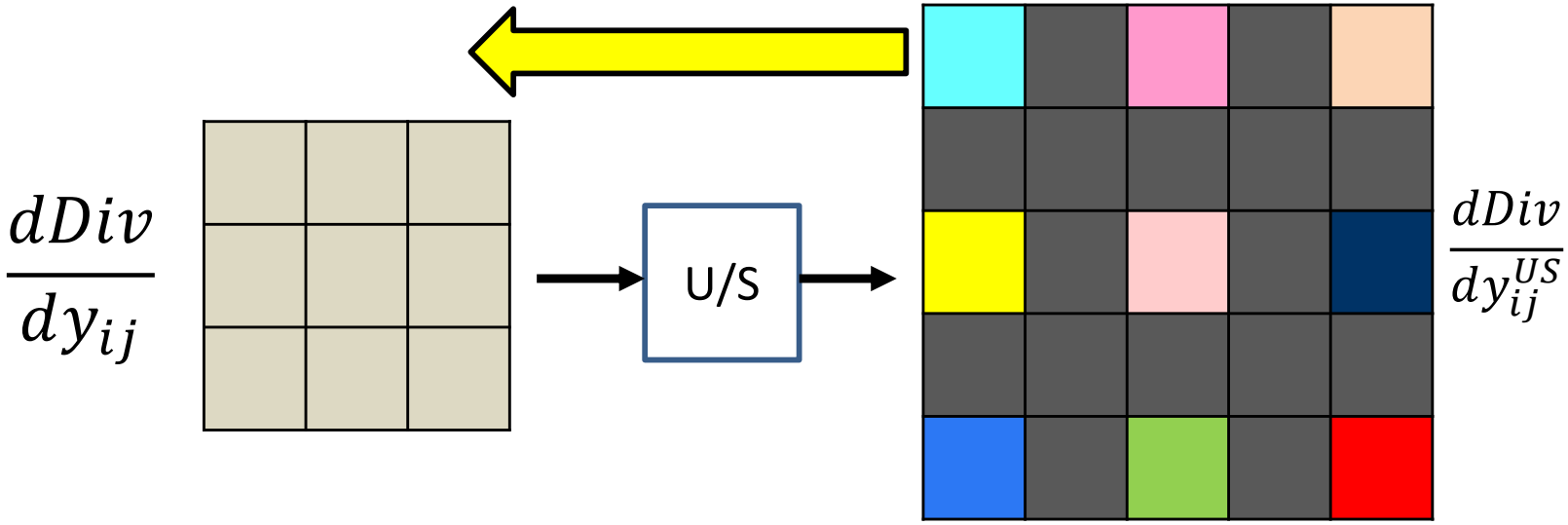
- An *upsampling* (or dilation) layer simply introduces $S - 1$ rows and columns for every map in the layer
  - Effectively *increasing* the size of the map by factor S in every direction
- Used explicitly to increase the map size by a uniform factor

Backprop through the upsampling layer



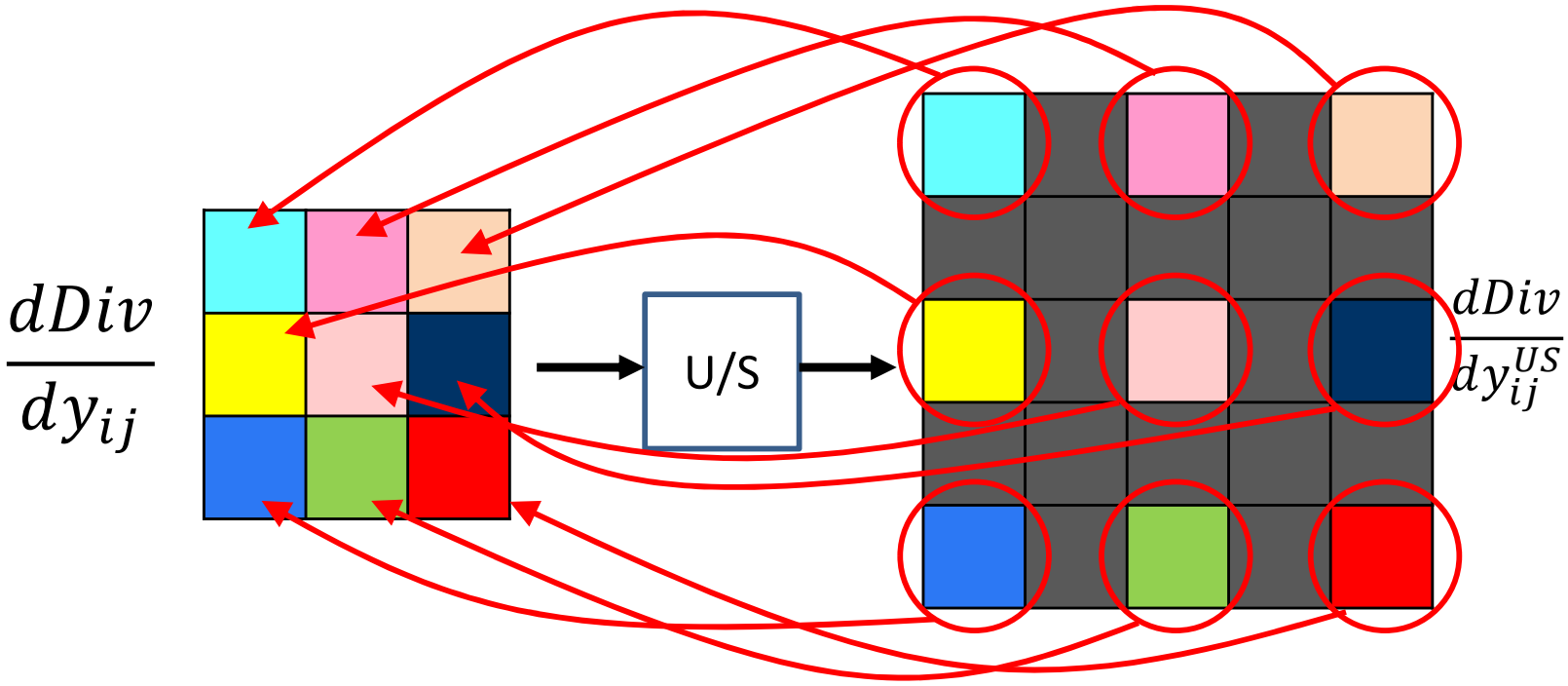
- **Backpropagation**: Given the derivative of the divergence with respect to the elements of the *output* of the upsampling, compute derivatives with respect to every element of the *input* to the upsampling
  - The "map" of these derivatives will be the same size as the input

Backprop through the upsampling layer



- The zero elements introduced during the forward pass in upsampling are *not* functions of the input
  - They are always introduced as 0, regardless of the input
- During backpropagation, they do not influence the derivatives going backward

Backprop through the upsampling layer

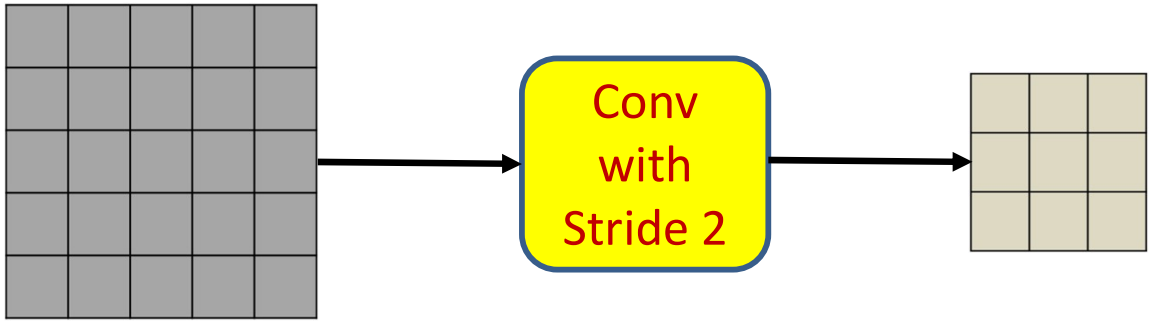


- The remaining elements are identical
  - The derivatives are identical
- Simply copy the derivatives for the “valid” locations over into the derivative for the input

Backprop through U/S pseudocode

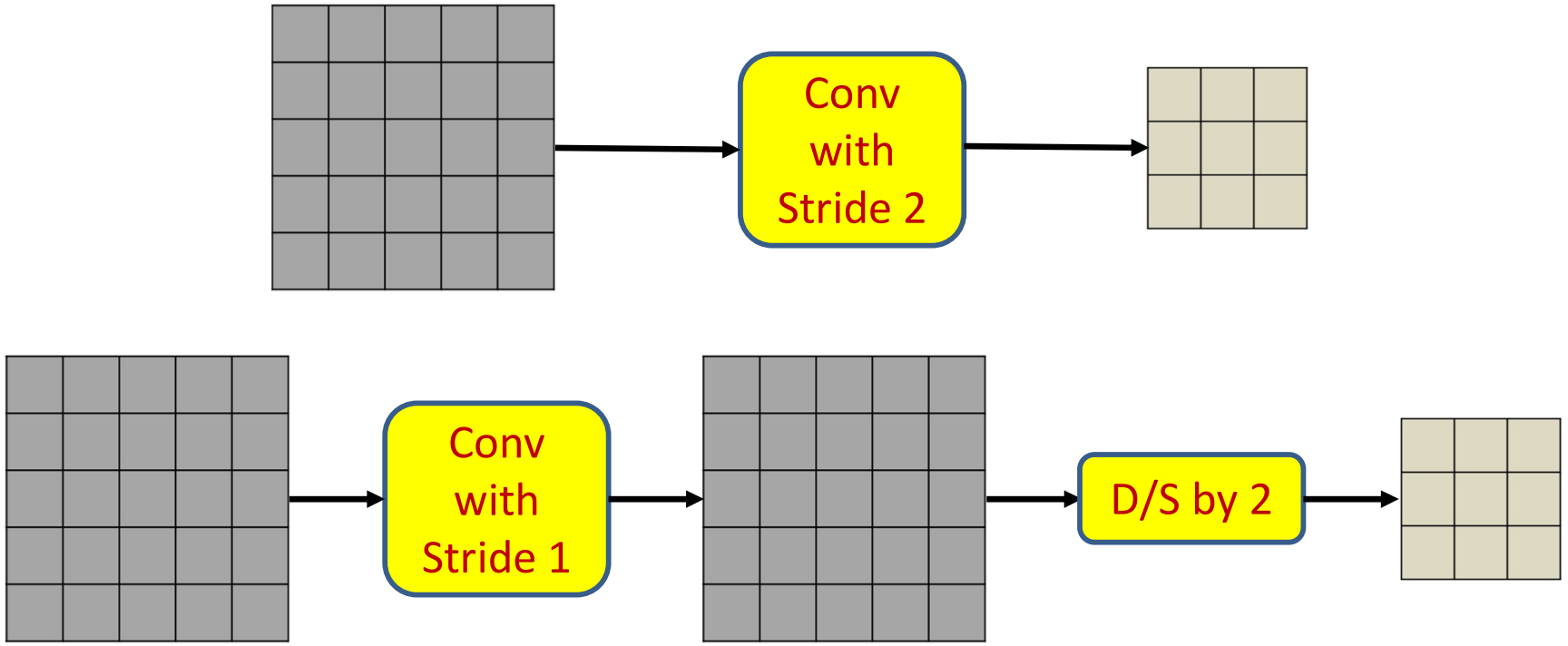
```
# S is the stride in the forward pass
# dz contains the divergence derivative for the U/S
#    output z in the forward pass

function dy = backprop_through_Upsampling (dz, S)
  #c = number of channels in dz
  for i = 1:width(z)
    for j = 1:height(z)
      dy(:,(i-1)S+1, (j-1)S+1) = dz(:,i,j)
  return dy
```

Convolutional layer with stride > 1



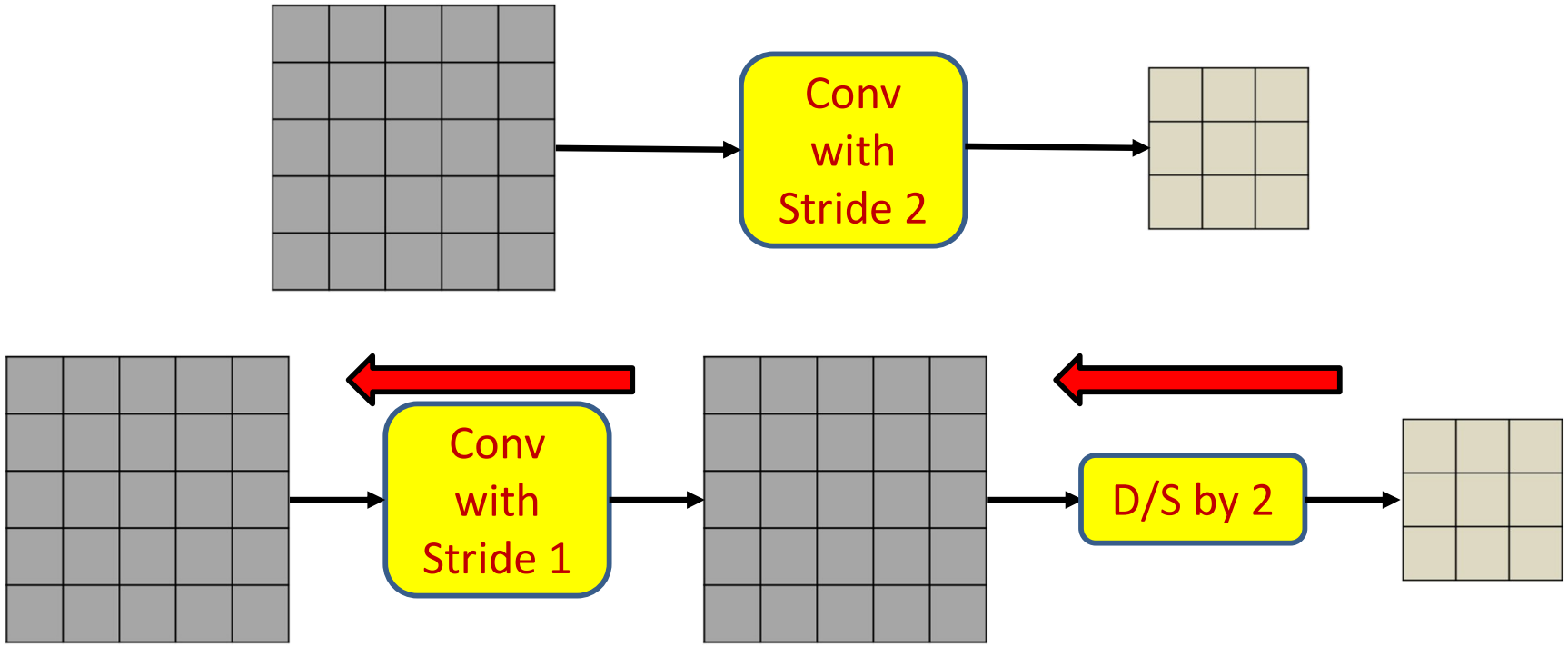
- Convolution is often performed with a stride larger than 1 to result in a smaller output map

Convolutional layer with stride > 1



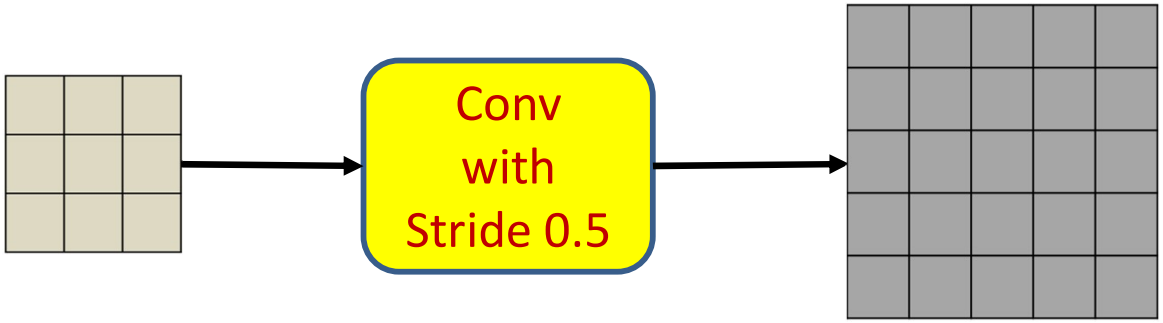
- Convolution is often performed with a stride larger than 1 to result in a smaller output map
- For purposes of backprop, it is easiest to view this as Convolution followed by down sampling

Convolutional layer with stride > 1



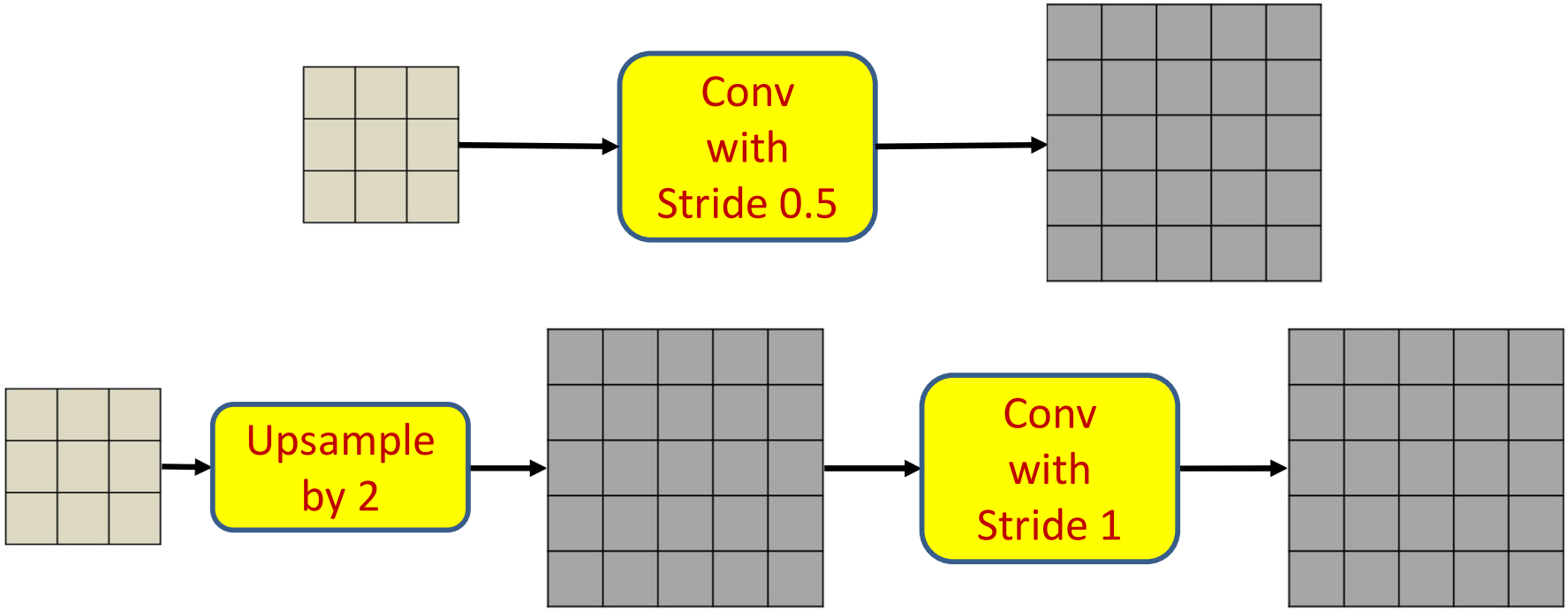
- Convolution is often performed with a stride larger than 1 to result in a smaller output map
- For purposes of backprop, it is easiest to view this as Convolution followed by down sampling
  - Backprop will first propagate derivatives through the D/S layer, and then through the Convolution layer
  - Simpler than trying to modify backprop rules to account for stride in convolution

Convolutional layer with fractional stride



- Convolution is also sometimes performed with a *fractional* stride to result in a larger output map

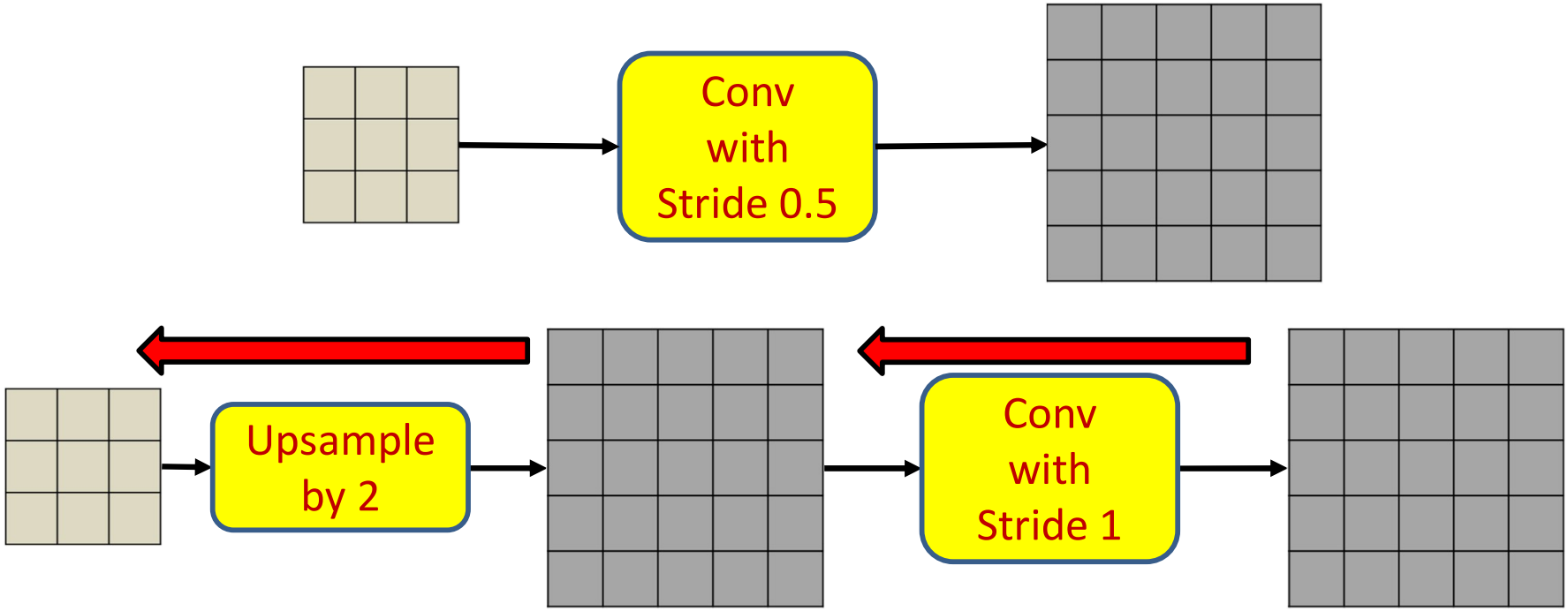
Convolutional layer with fractional stride



- Convolution is also sometimes performed with a *fractional* stride to result in a larger output map

- For purposes of backprop, it is easiest to view this as ***upsampling followed by convolution*.**

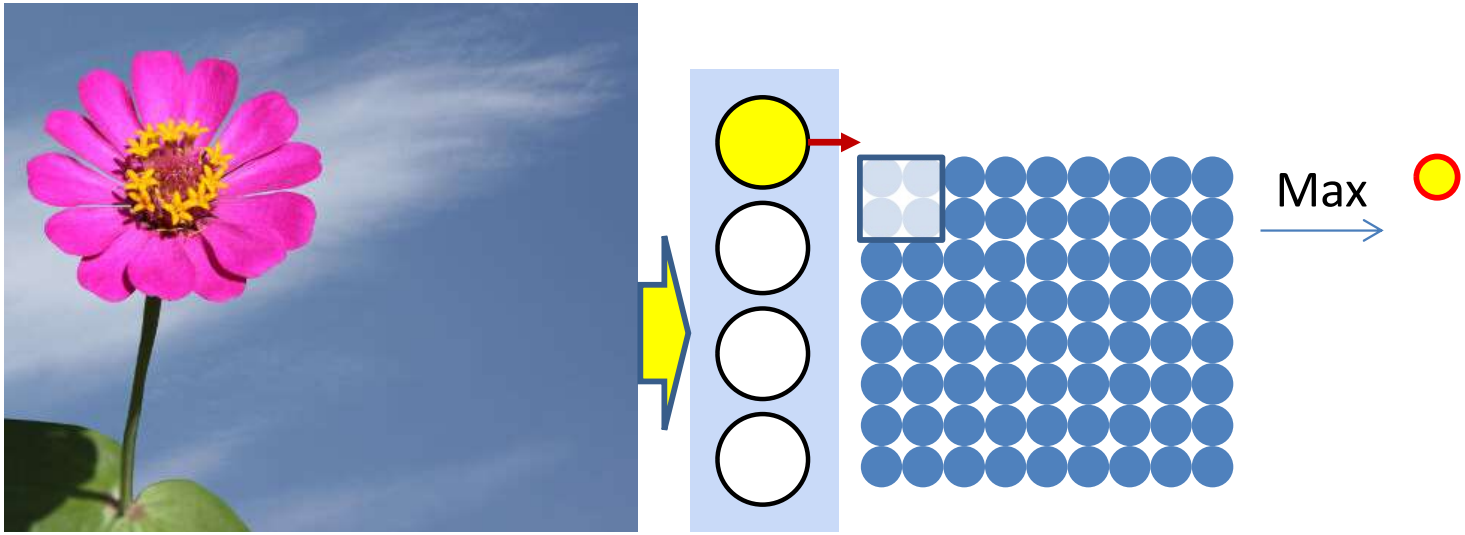
Convolutional layer with fractional stride



- Convolution is also sometimes performed with a *fractional* stride to result in a larger output map

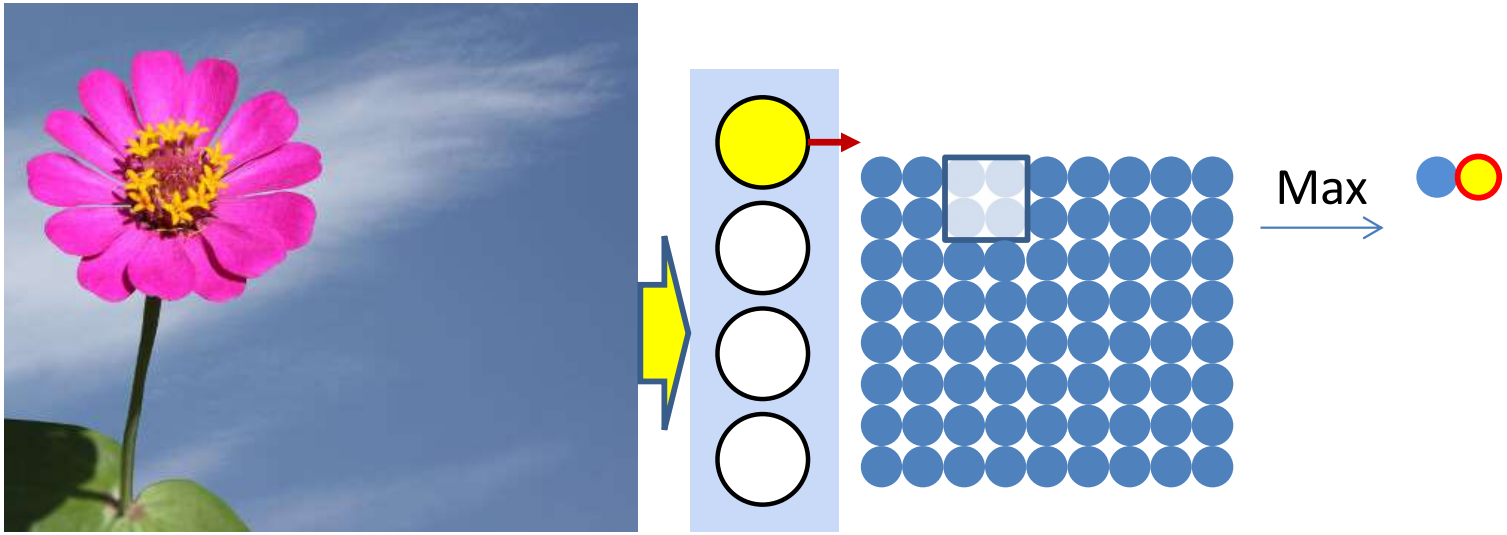
- For purposes of backprop, it is easiest to view this as ***upsampling followed by convolution*.**
  - Backprop will first propagate derivatives through convolution layer, and then the upsampling layer

Pooling and downsampling



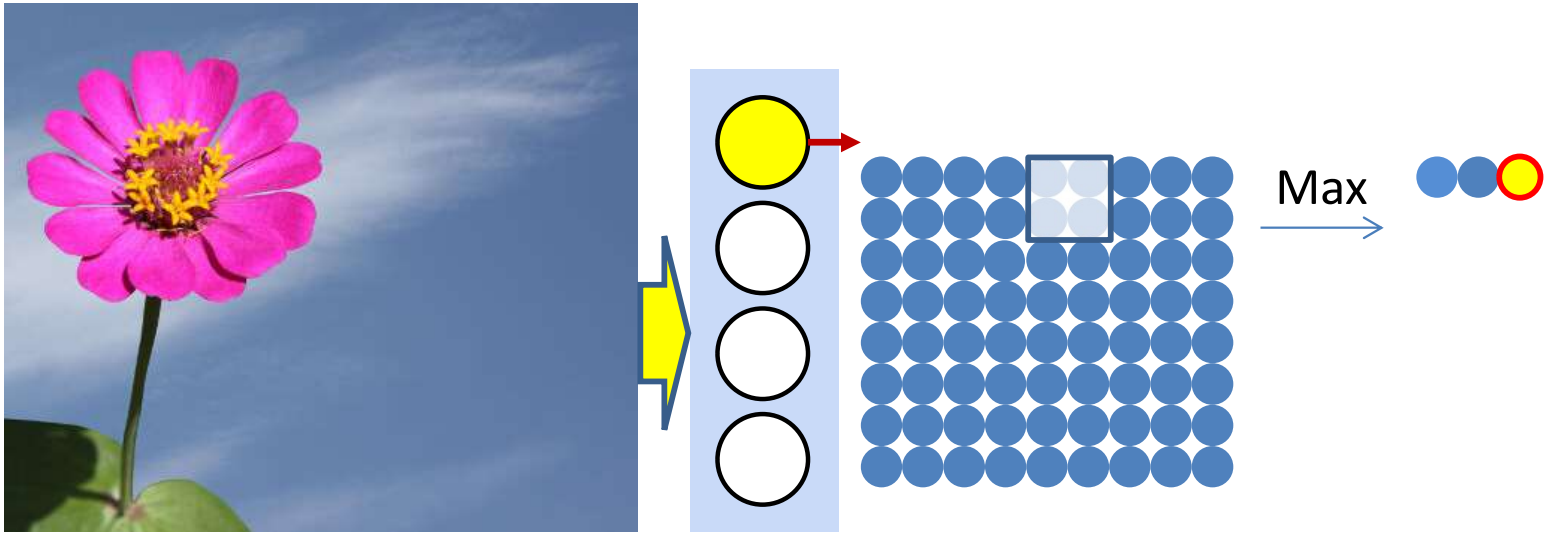
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



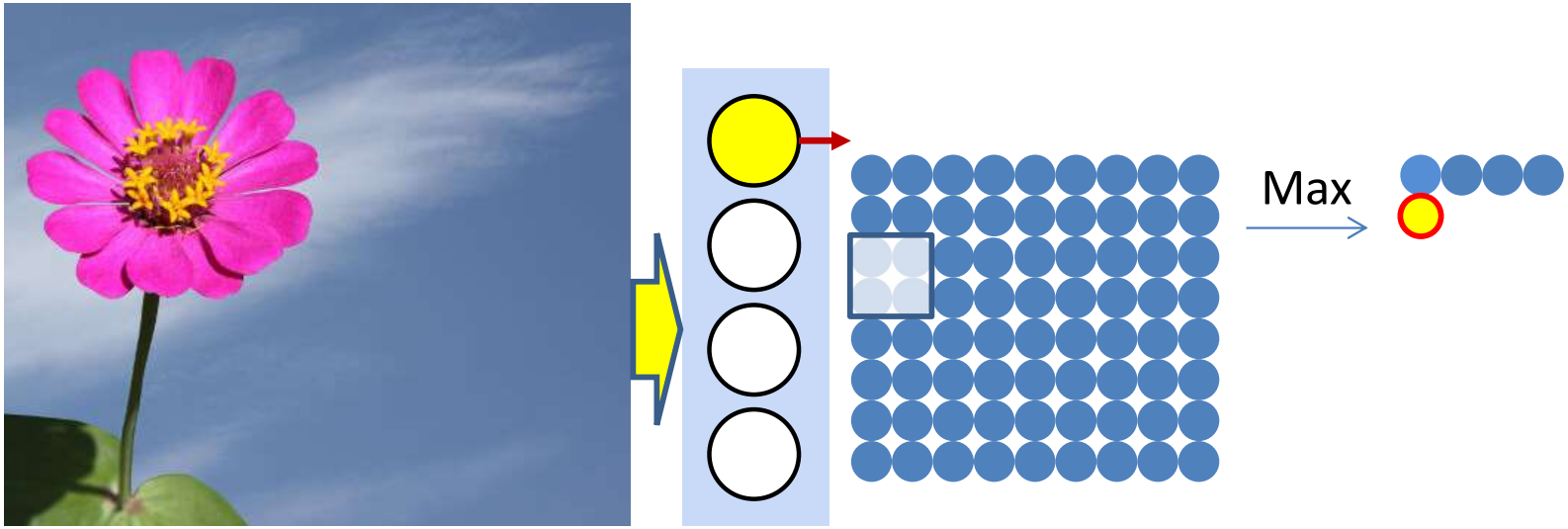
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



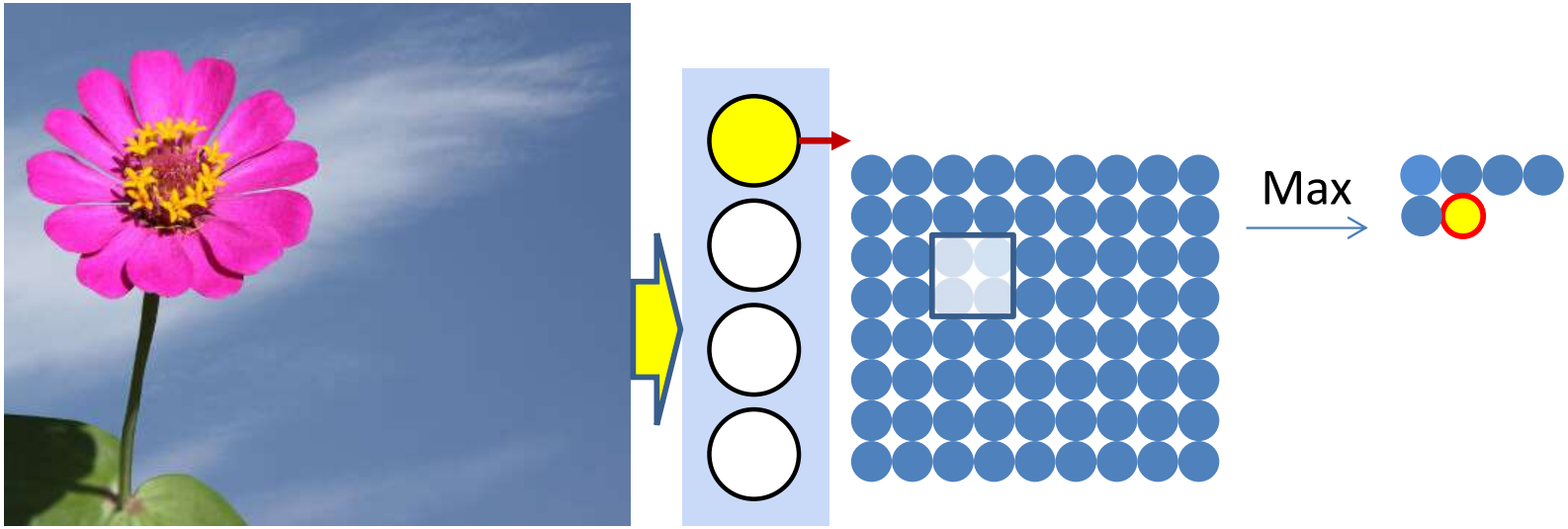
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



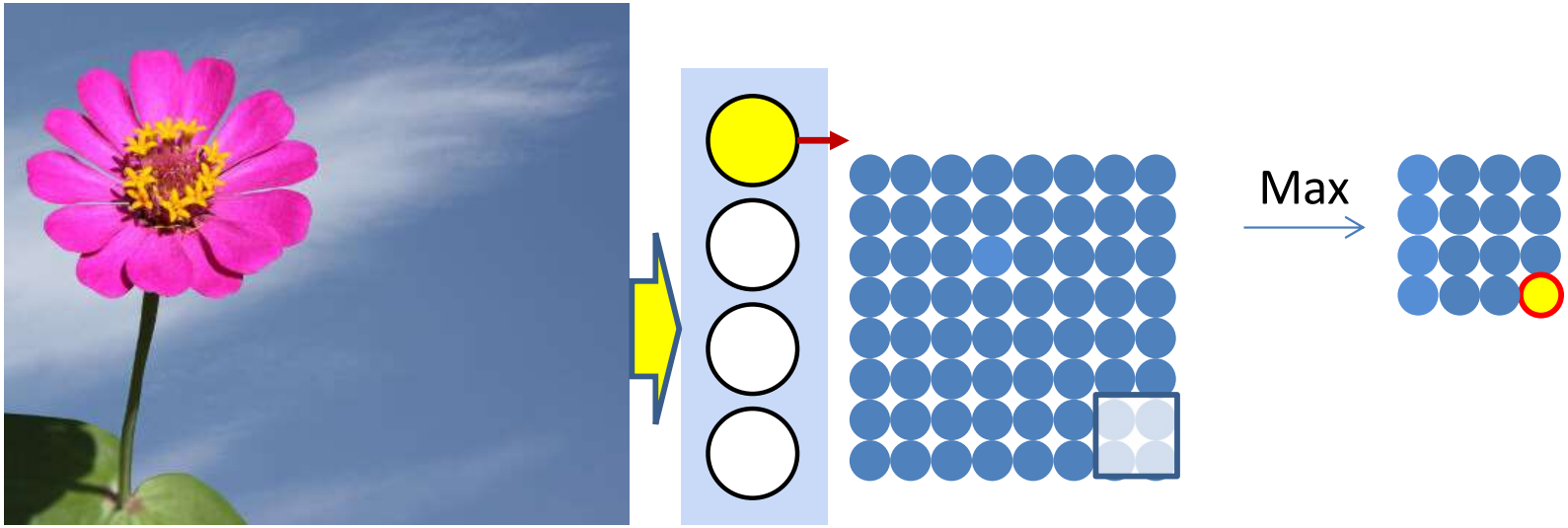
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



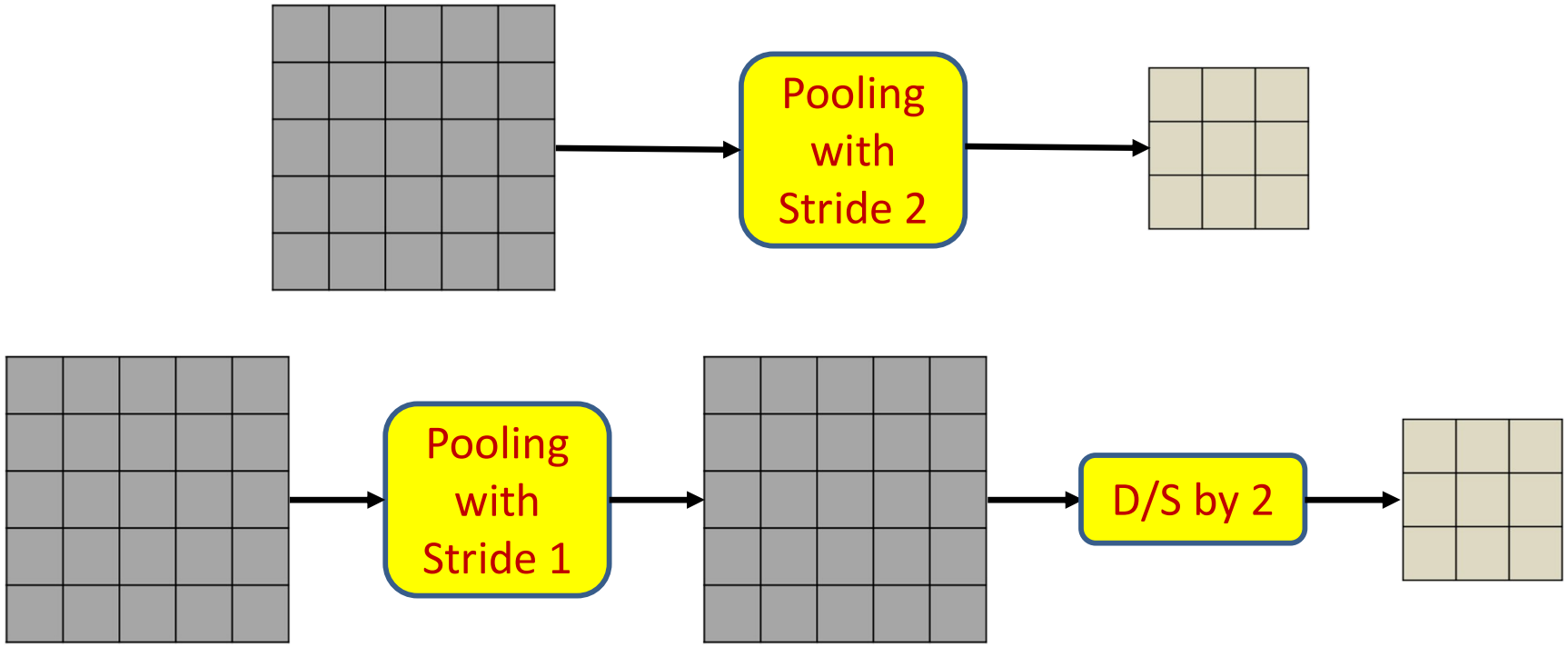
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



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

Pooling layer with stride > 1



- Convolution is often performed with a stride larger than 1 to result in a smaller output map
- For purposes of backprop, it is easiest to view this as Convolution followed by down sampling

Through the eyes of code

- As always, the code is simpler

Convolution: Forward layer *l*

```
Y(0,:,:,:) = Image
for x = 1:W_{l-1}-K_l+1
    for y = 1:H_{l-1}-K_l+1
        for j = 1:D_l
            z(l,j,x,y) = 0
            for i = 1:D_{l-1}
                for x' = 1:K_l
                    for y' = 1:K_l
                        z(l,j,x,y) += w(l,j,i,x',y')
                                      Y(l-1,i,x+x'-1,y+y'-1)
        Y(l,j,x,y) = activation(z(l,j,x,y))
```

Switching to 1-based indexing with appropriate adjustments

Conv Backward layer *l*

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

Convolution forward with stride layer *l*

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

```
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
            z(l,j,m,n) = 0
            for i = 1:D_{l-1}
                for x' = 1:K_l
                    for y' = 1:K_l
                        z(l,j,m,n) += w(l,j,i,x',y')
                                      Y(l-1,i,x+x'-1,y+y'-1)
            Y(l,j,m,n) = activation(z(l,j,m,n))
        n++
    m++
Y = softmax( {Y(L,:,:,:)} )
```

Conv Backward (with strides) at layer l

```
dw(l) = zeros(D_l x D_{l-1} x K_l x K_l)
dY(l-1) = zeros(D_{l-1} x W_{l-1} x H_{l-1})
for x = W_l:downto:1
    m = (x-1)stride
    for y = H_l:downto:1
        n = (y-1)stride
        for j = D_l:downto:1
            dz(l,j,x,y) = dY(l,j,x,y).f'(z(l,j,x,y))
            for i = D_{l-1}:downto:1
                for x' = K_l:downto:1
                    for y' = K_l:downto:1
                        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')
```

Max Pooling layer at layer l with a stride

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

Max pooling

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

Derivative of max pooling layer at layer l

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

Max pooling

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

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

Mean Pooling layer at layer l with a stride

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

Mean pooling

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

Derivative of mean pooling layer at layer l with a stride

Mean pooling

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

"+=" because adjacent windows may overlap

Poll 2

Poll 2

The backward pass of an upsampling layer is downsampling?

- **True**
- False

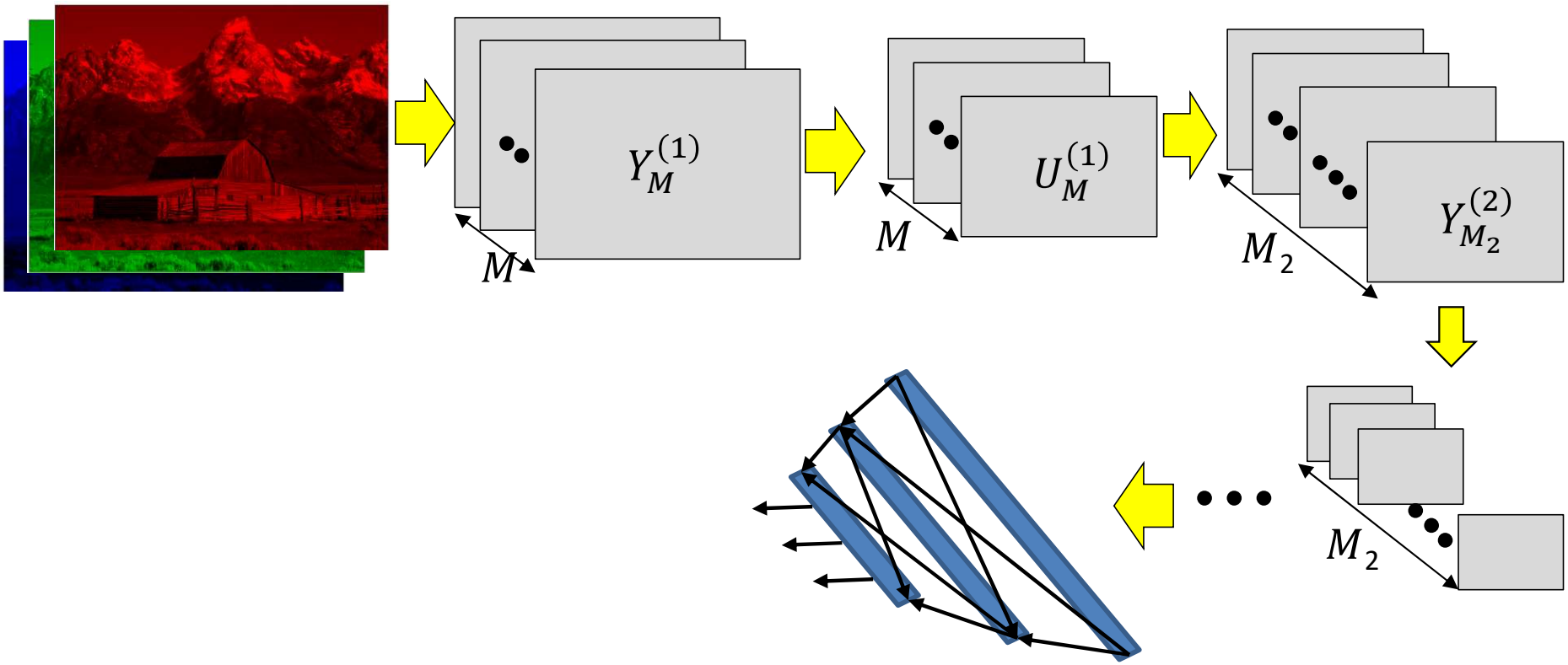
The backward pass of a downsampling layer is upsampling?

- **True**
- False

We can simply use an upsampling layer as the backward pass of downsampling and vice versa

- True
- **False**

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

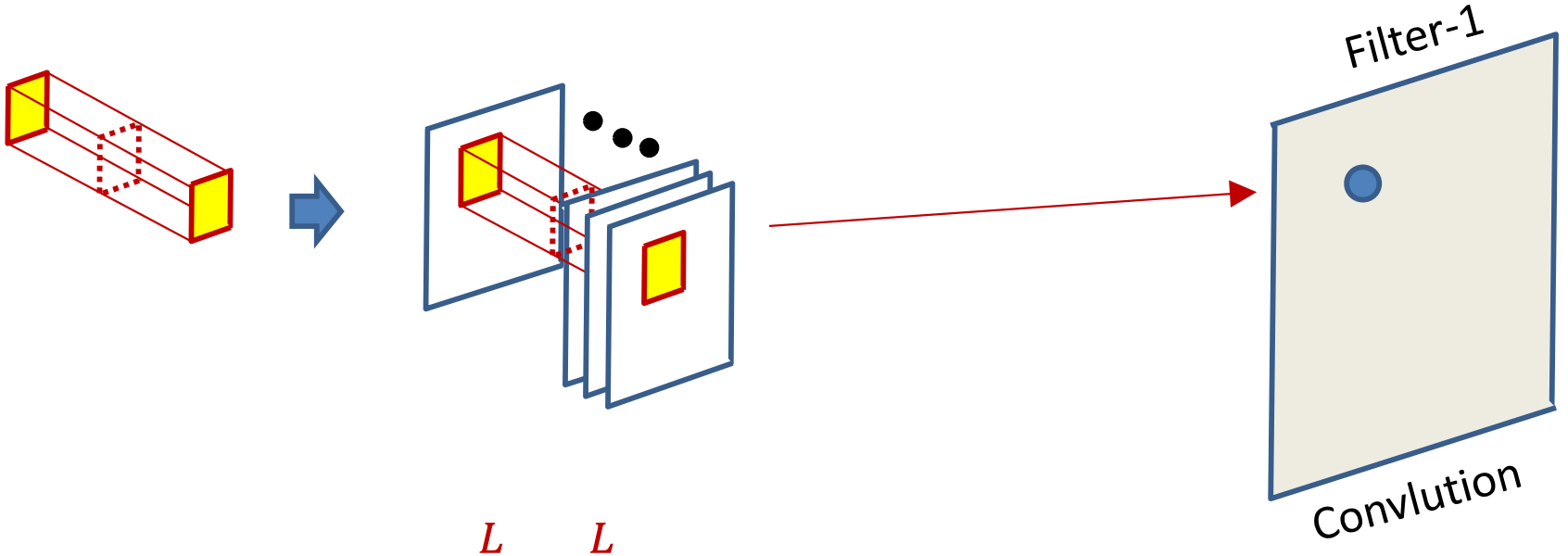
- The convolutional neural network is a supervised version of a computational model of mammalian vision
- It includes
  - Convolutional layers comprising learned filters that scan the outputs of the previous layer
  - Downsampling layers that operate over groups of outputs from the convolutional layer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

Invariance

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

Shift-invariance – a different perspective

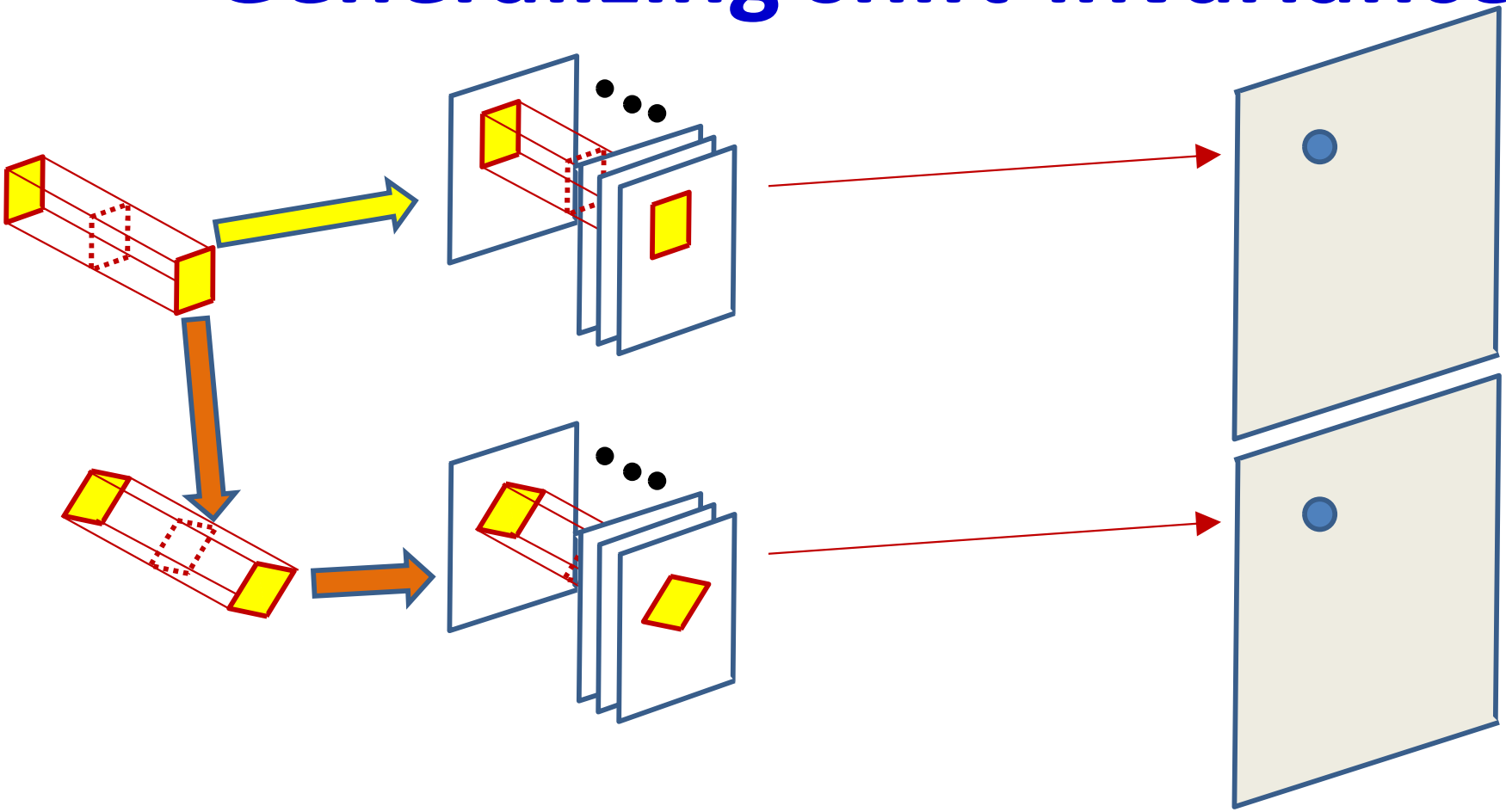


$$z(l,s,i,j) = \sum_{p}\sum_{k=1}^{L}\sum_{m=1}^{L} w(l,s,p,k,m)Y(l-1,p,i+k,j+m)$$

- 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



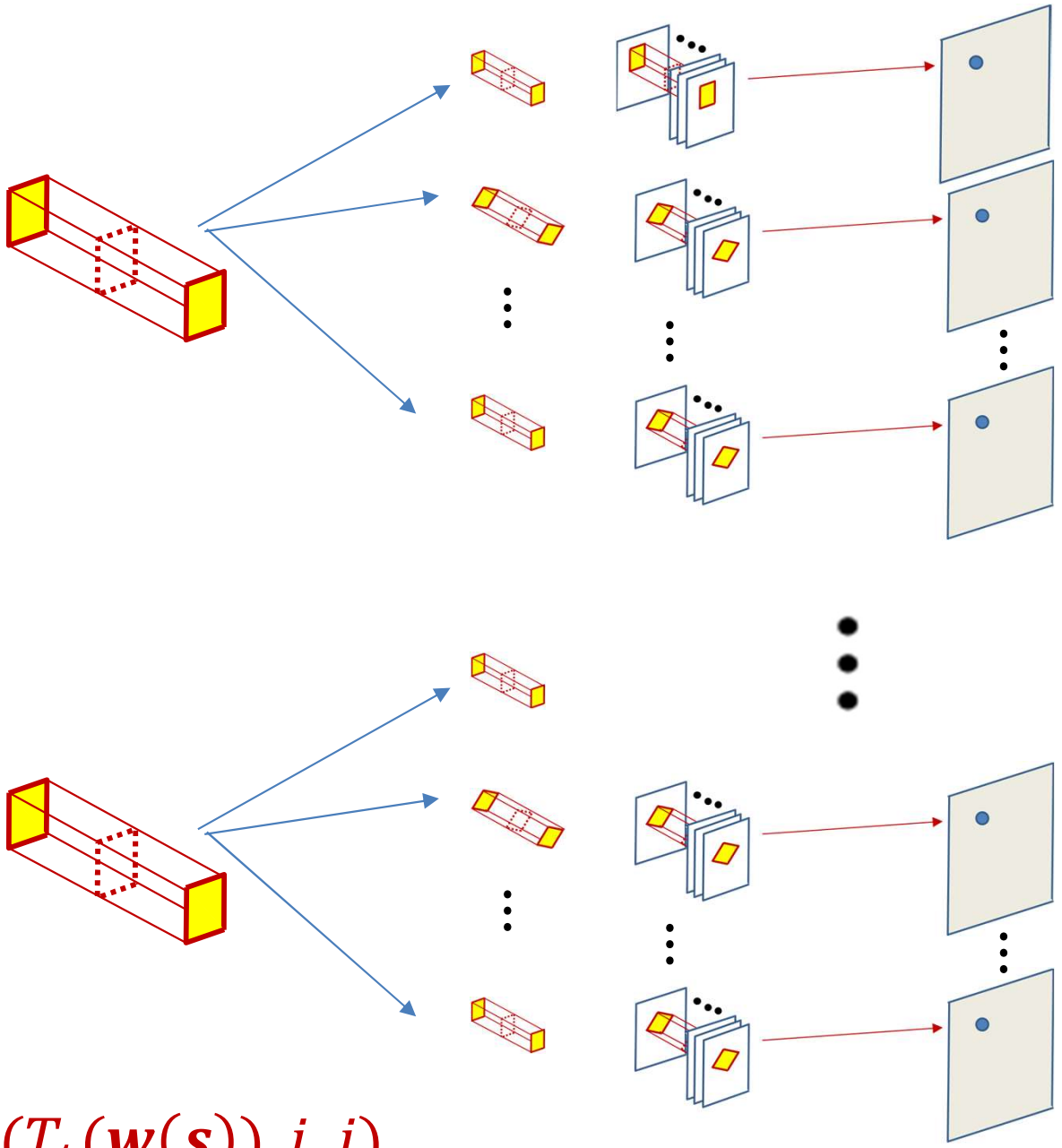
$$z_{regular}(s,i,j) = \boldsymbol{Y}.shift(\boldsymbol{w}(\boldsymbol{s}),i,j)$$

- 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 $D_{l-1} \times K_l \times K_l$ tensor

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

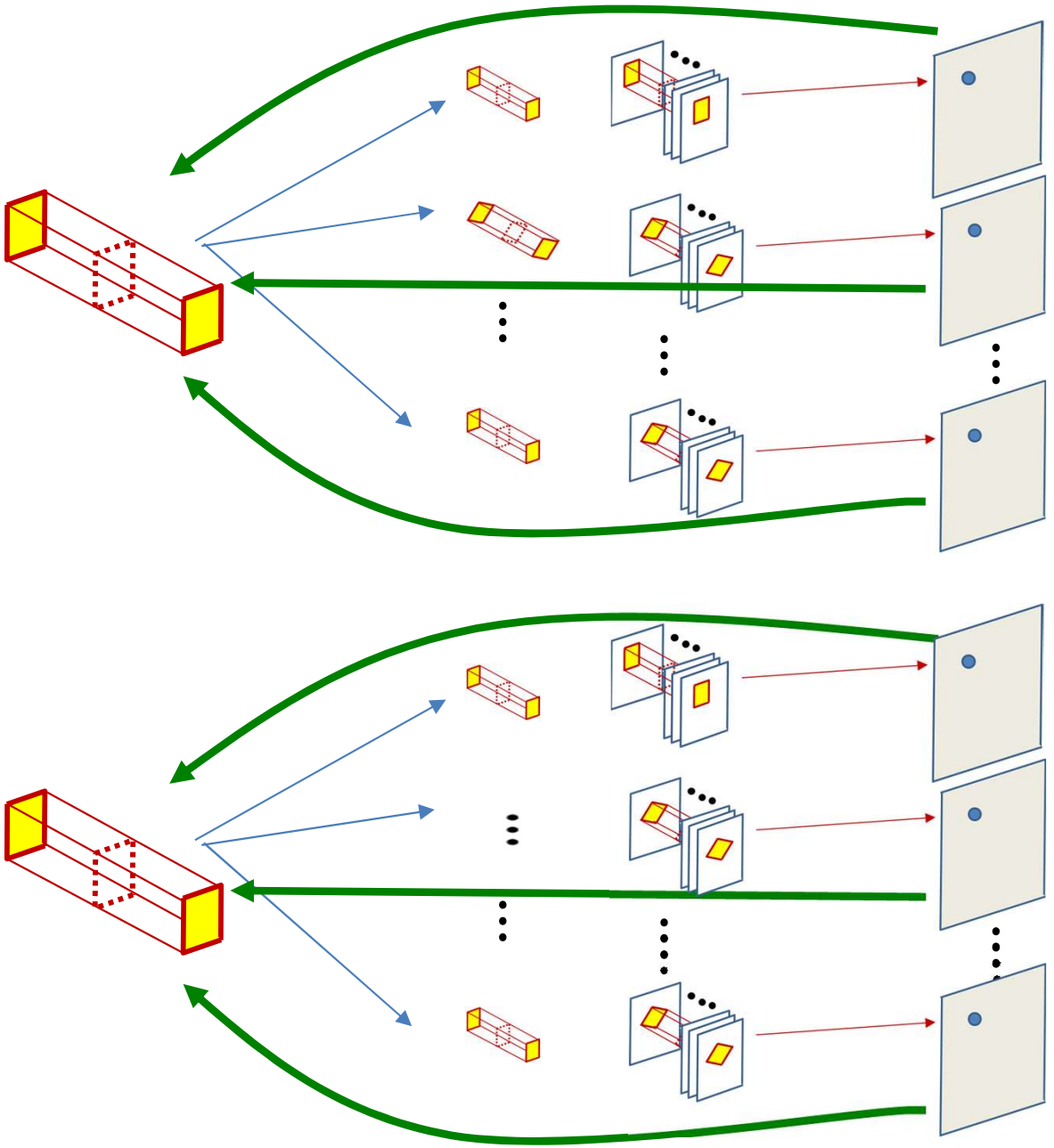
Transform invariance

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

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

BP with transform invariance

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



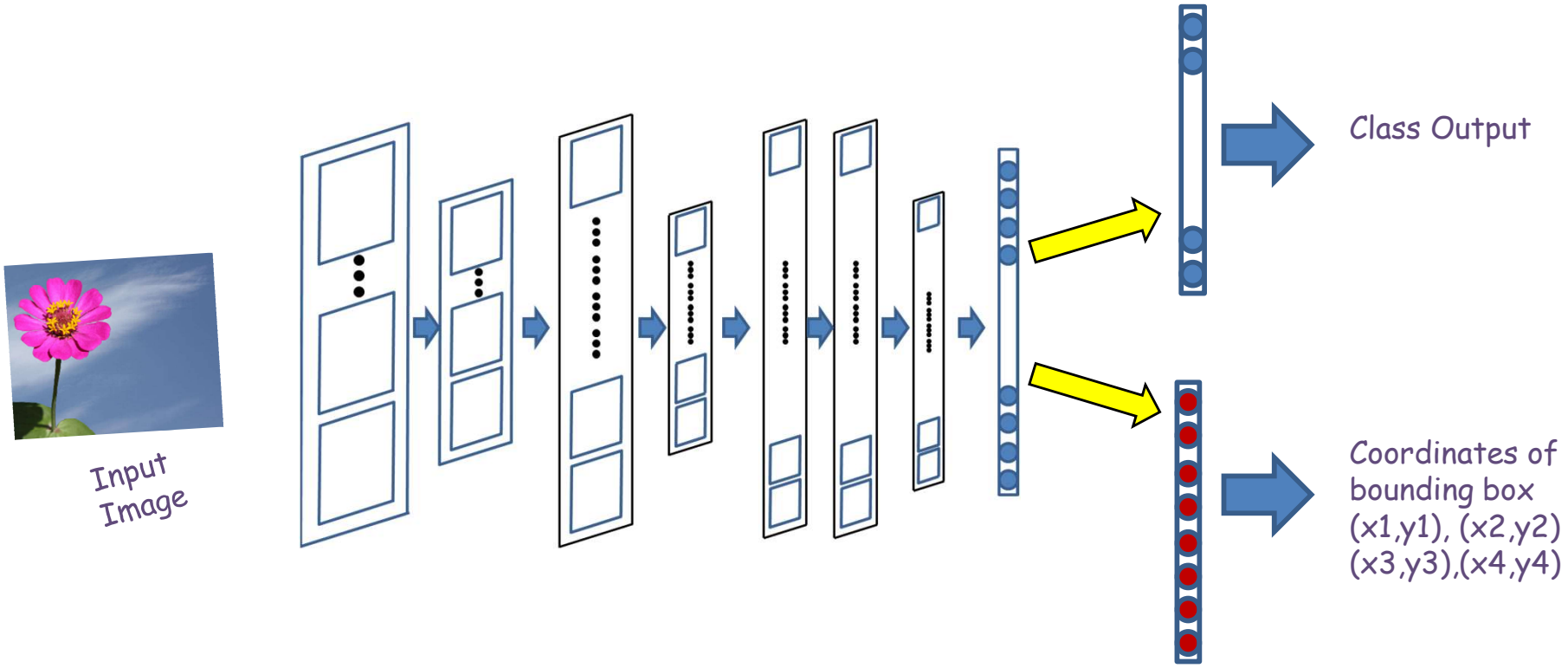
Story so far

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

But what about the exact location?

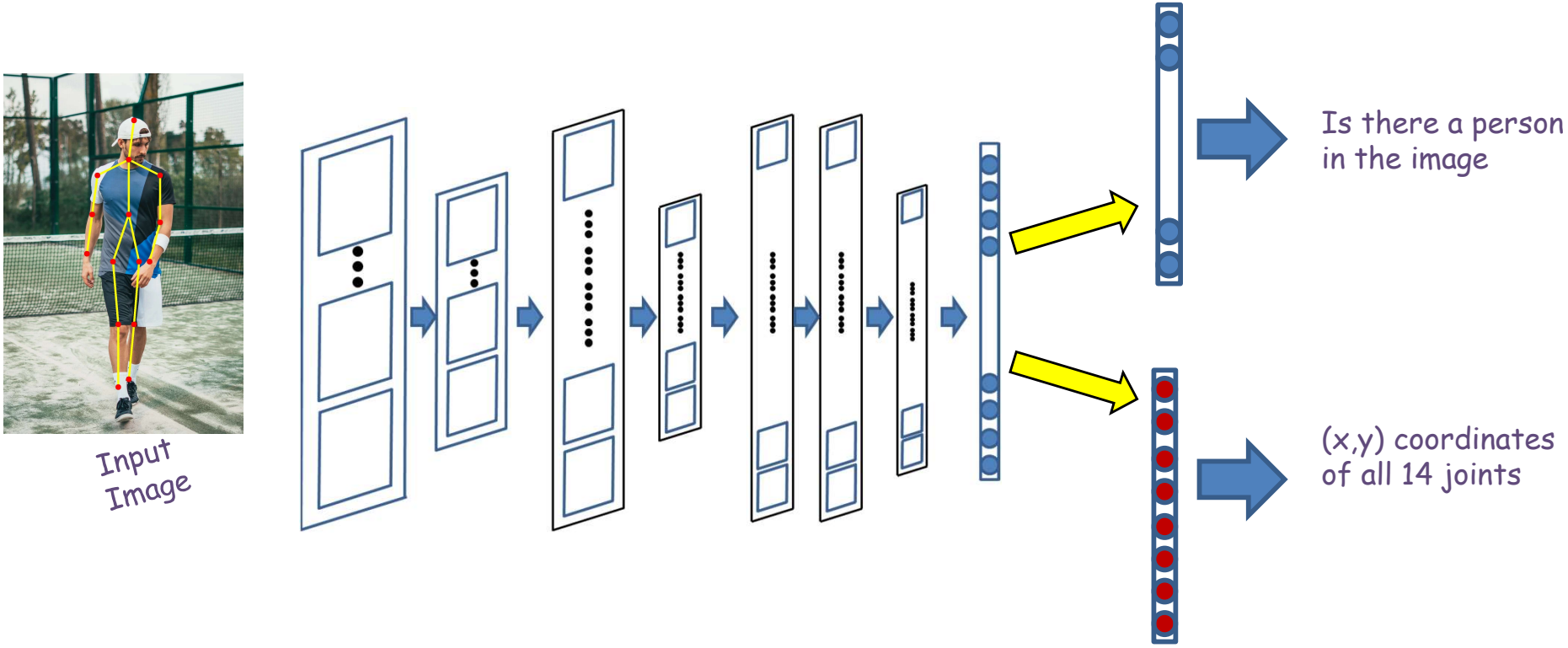
- We began with the desire to identify the picture as containing a flower, regardless of the position of the flower
  - Or more generally the class of object in the picture
- But can we detect the *position* of the main object?

Finding Bounding Boxes



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

Pose estimation



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

Poll 3

Poll 3

To find the position of an object using a CNN, we need multiple output layers after the final convolution, one to identify the class and another to predict the position of the object

- **True**
- False

CNNs are invariant to the position, but not the orientation or scale of the target pattern

- **True**
- False

To make them invariant to a transform, transformed versions of every filter must be included in the model, for every transform considered

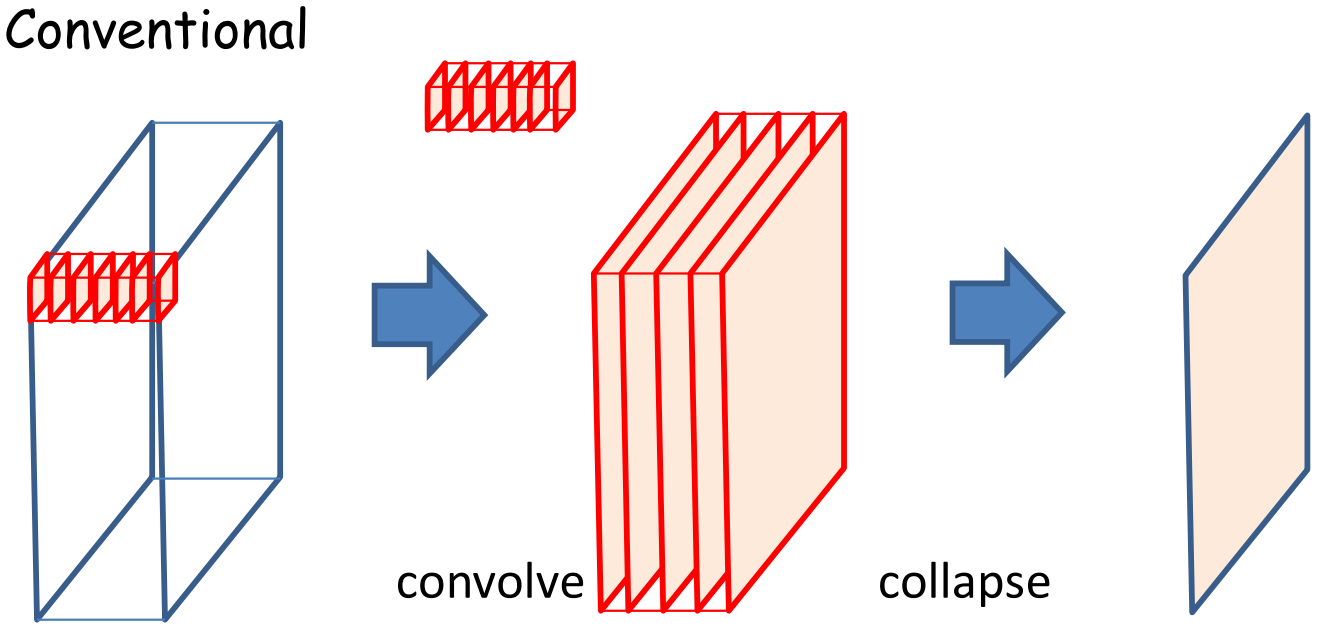
- **True**
- False

Model variations

- *Very deep* networks
  - *100* or more layers in MLP
  - Formalism called “Resnet”
    - You will encounter this in your HWs

- *“Depth-wise” convolutions*
  - Instead of multiple independent filters with independent parameters, use common layer-wise weights and combine the layers differently for each filter

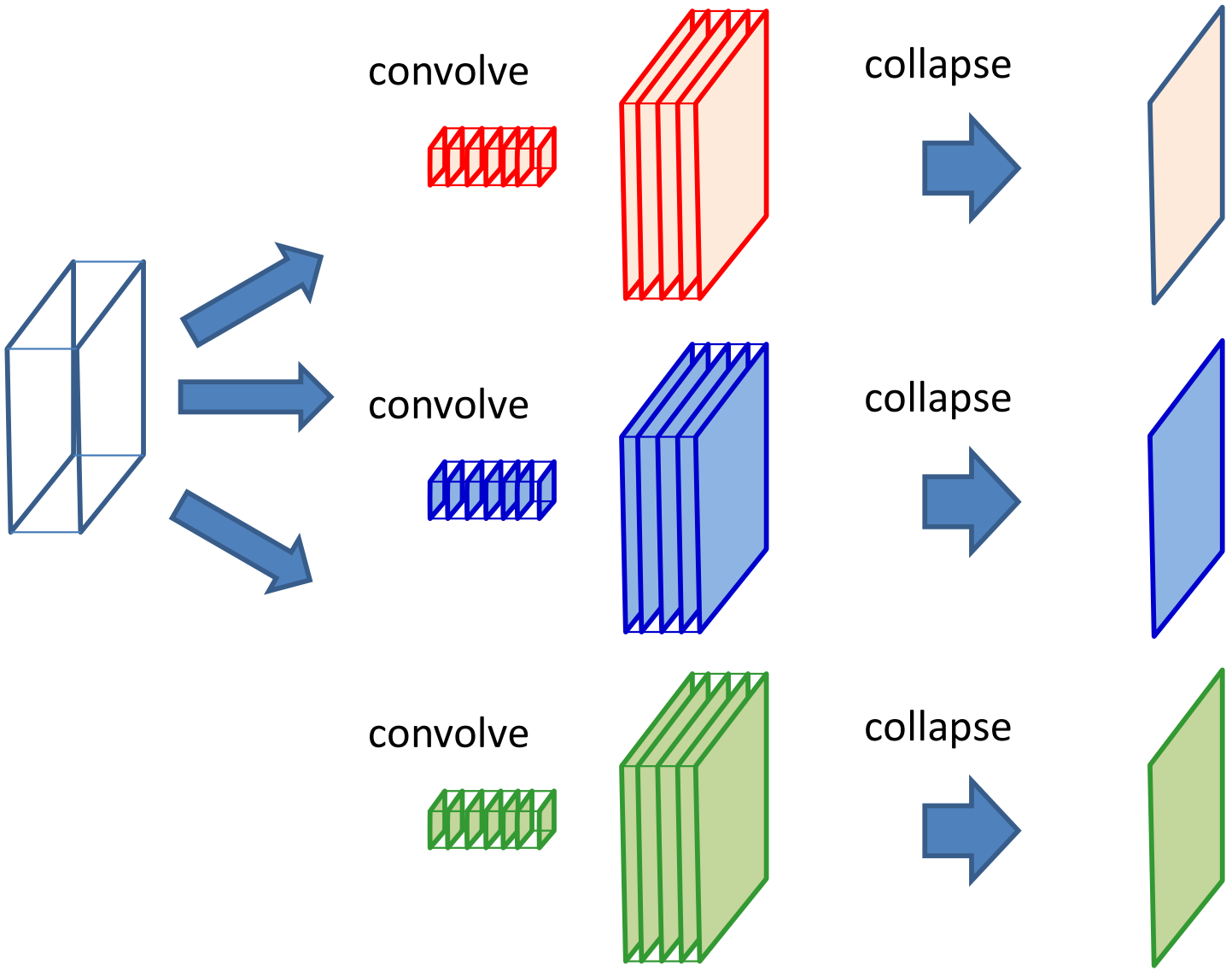
Conventional convolutions



- Alternate view of conventional convolution:

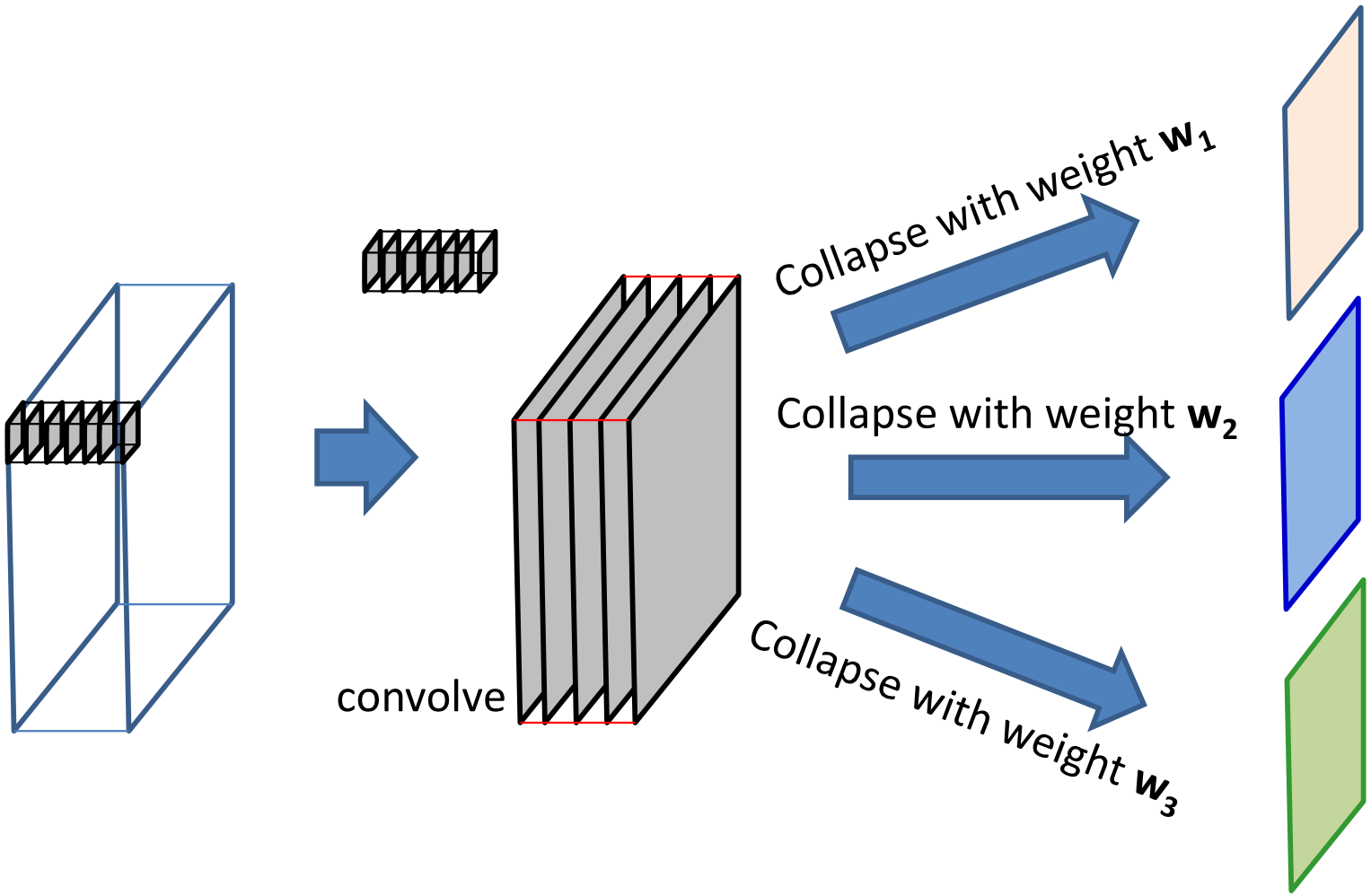
- E*ach layer of each filter* scans its corresponding map to produce a convolved map
- N input channels will require a filter with N layers
- The independent convolutions of each layer of the filter result in N convolved maps
- The N convolved maps are *added together* to produce the final output map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

Conventional convolutions



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

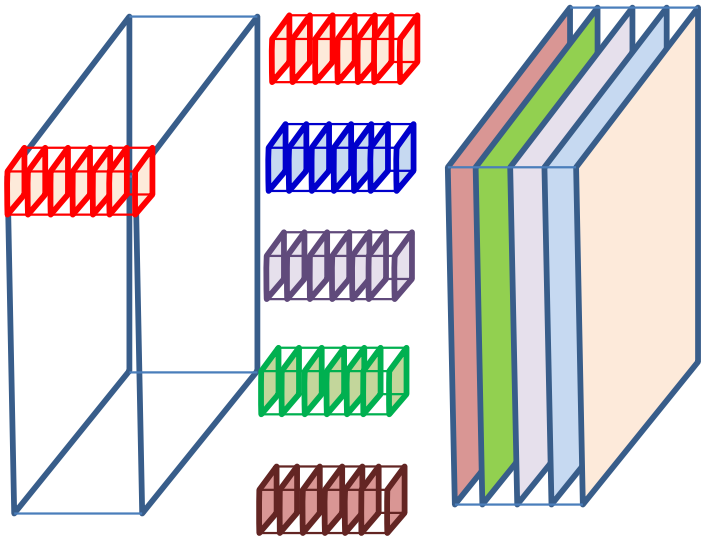
Depth-wise convolution



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

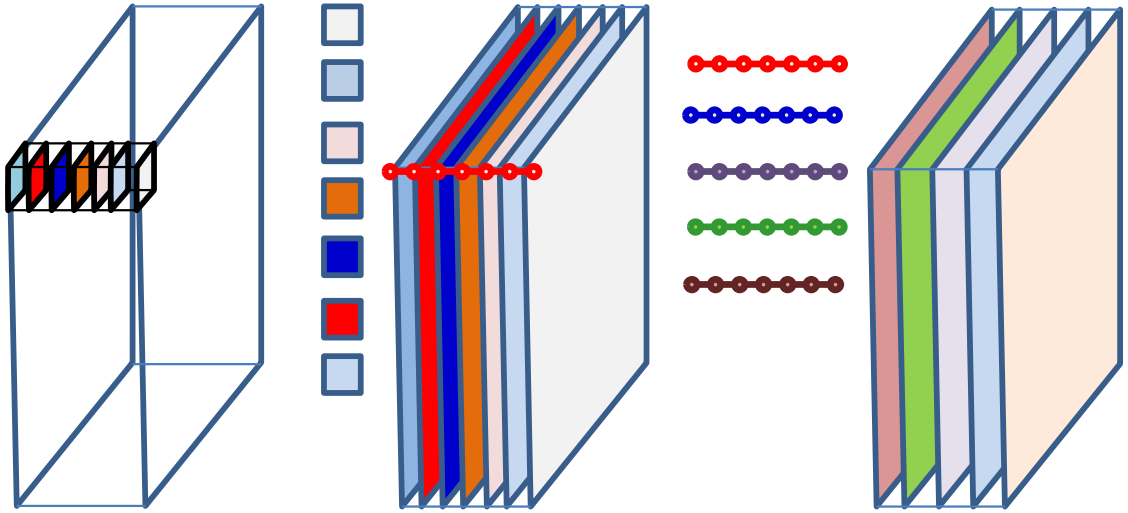
Conventional vs. depth-wise convolution

Conventional



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

Depth-wise



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

Poll 4

Poll 4

Filters in depth-wise convolutions convolve all the input channels simultaneously and sum the result

- True
- **False**

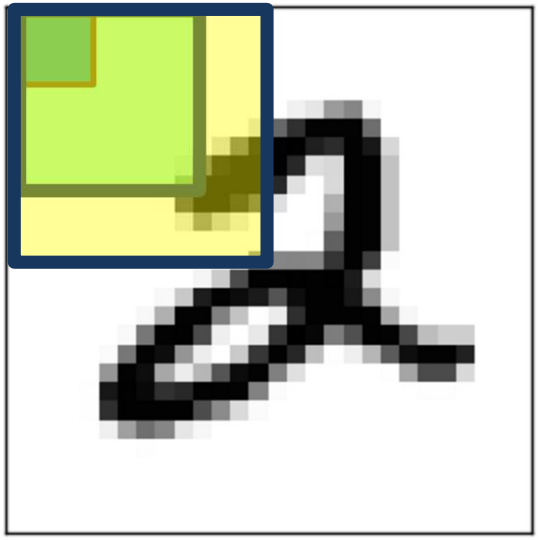
Depthwise convolutions require far fewer parameters and computation than regular convolutions

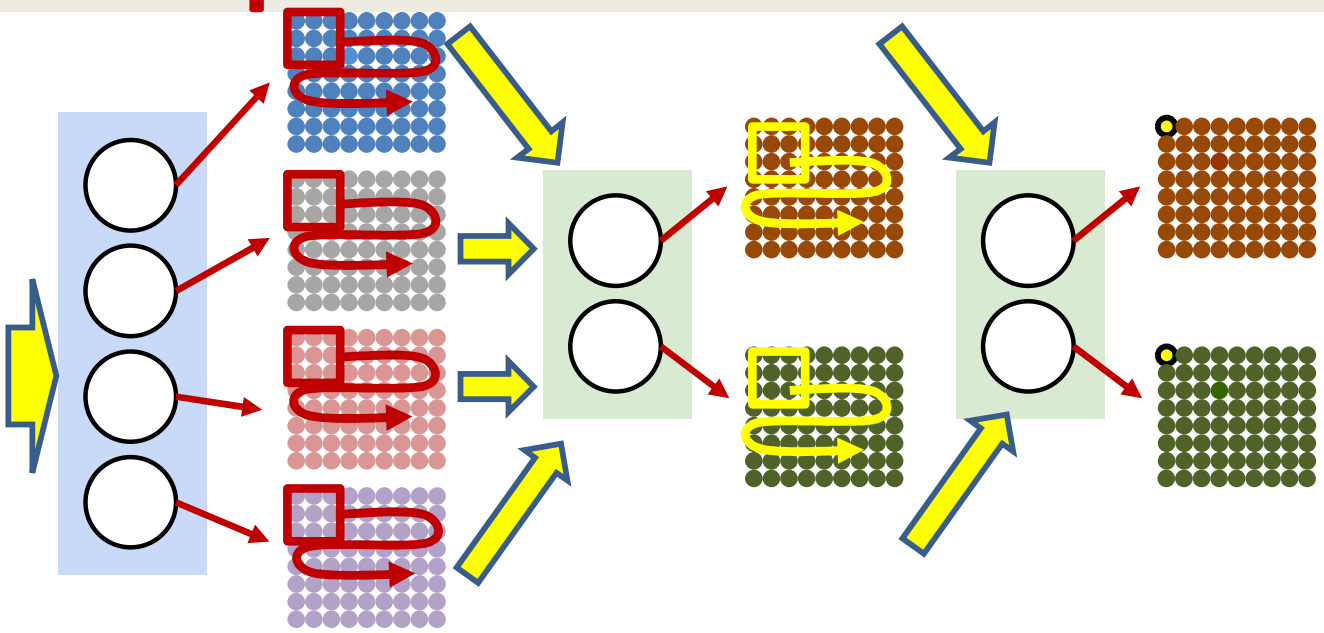
- **True**
- Flase

Story so far

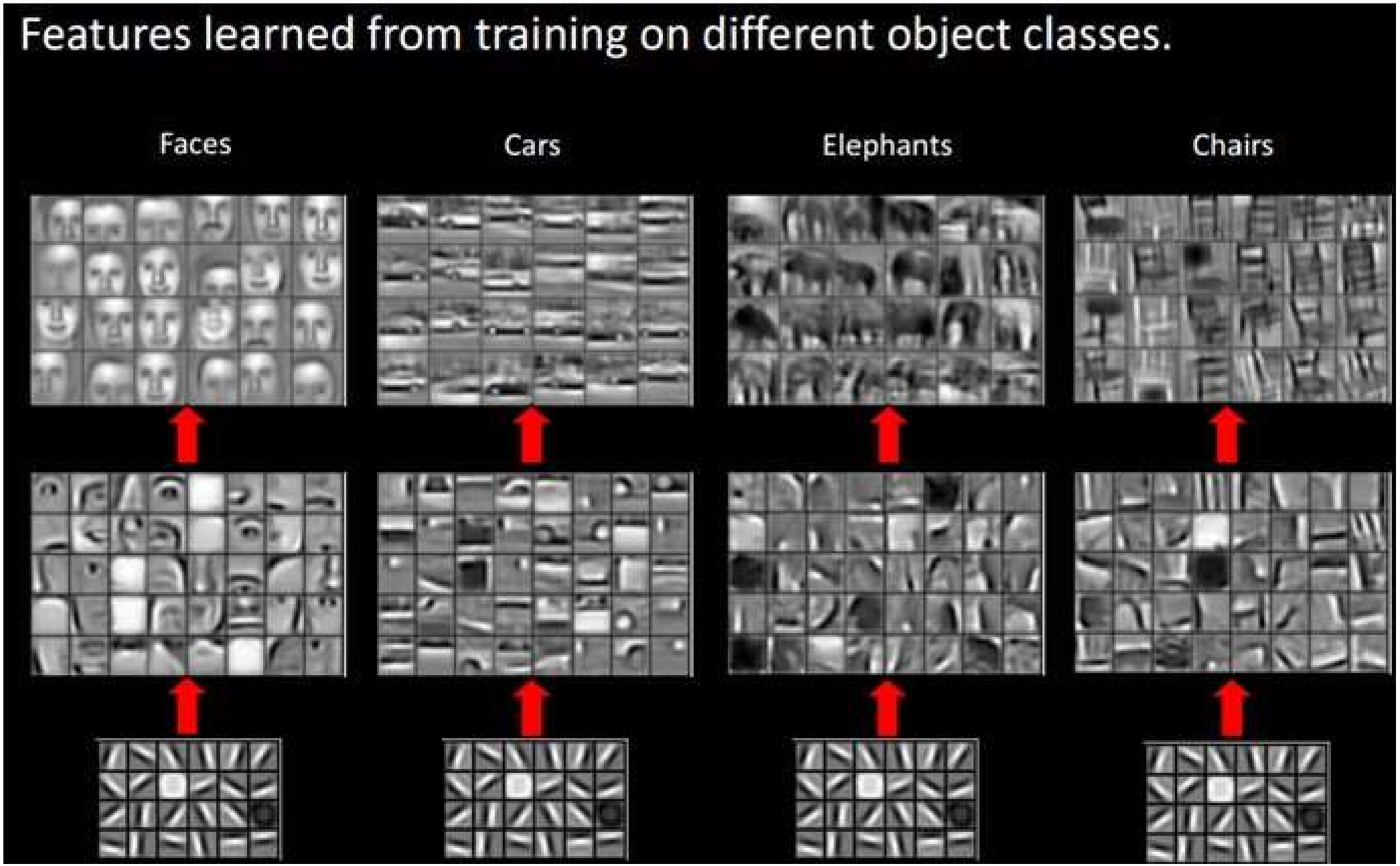
- CNNs are shift-invariant neural-network models for shift-invariant pattern detection
  - Are equivalent to scanning with shared-parameter MLPs with distributed representations
- The parameters of the network can be learned through regular back propagation
- Like a regular MLP, individual layers may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

What do the filters learn? Receptive fields





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

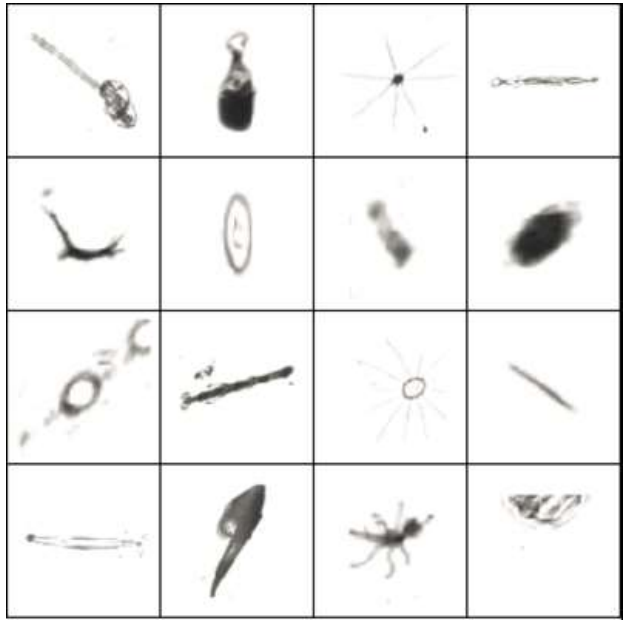

Features learned from training on different object classes.
Faces
Cars
Elephants
Chairs

Training Issues

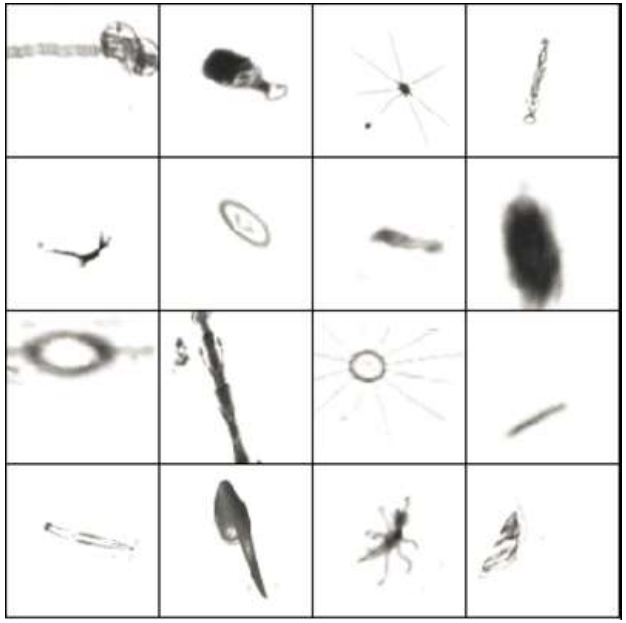
- Standard convergence issues
  - Solution: Adam or other momentum-style algorithms
  - Other tricks such as batch normalization
- The number of parameters can quickly become very large
- Insufficient training data to train well
  - Solution: Data augmentation

Data Augmentation

Original data



Augmented data



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

Convolutional neural nets

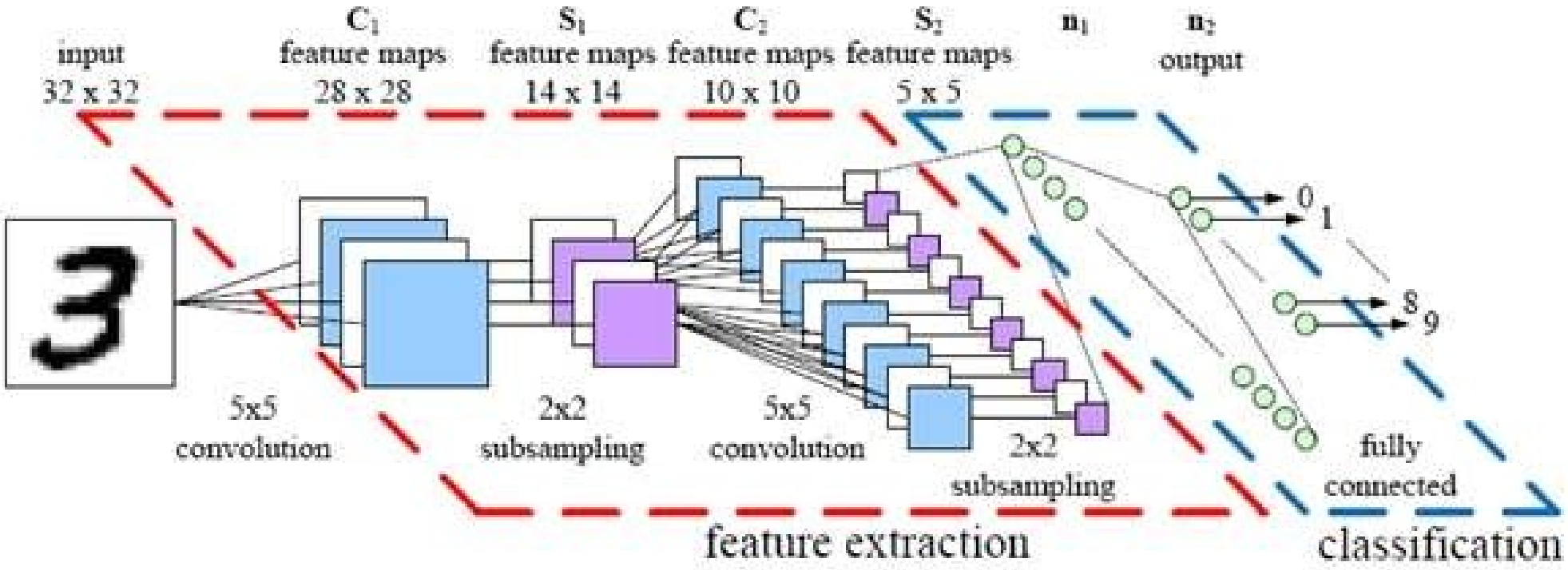
- One of *the* most frequently used nnet formalism today

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

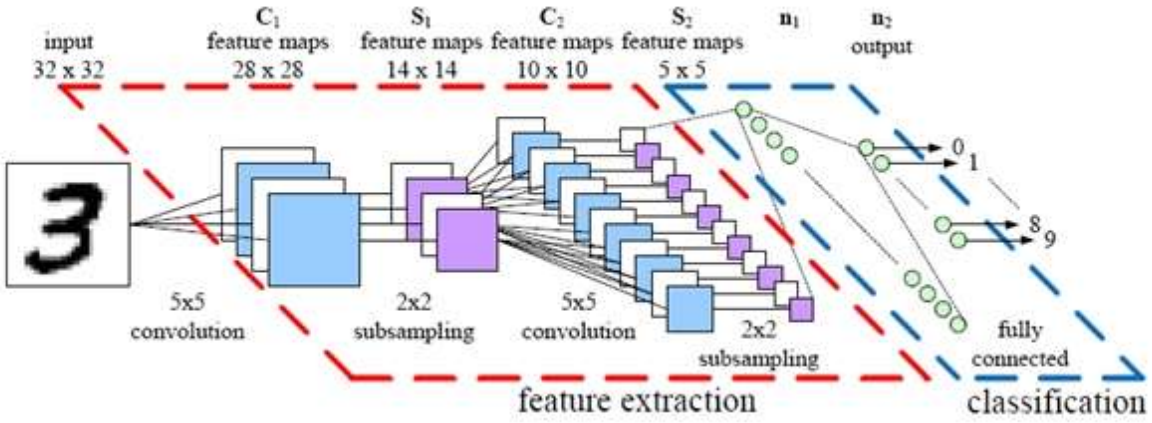
Nice visual example

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

Digit classification

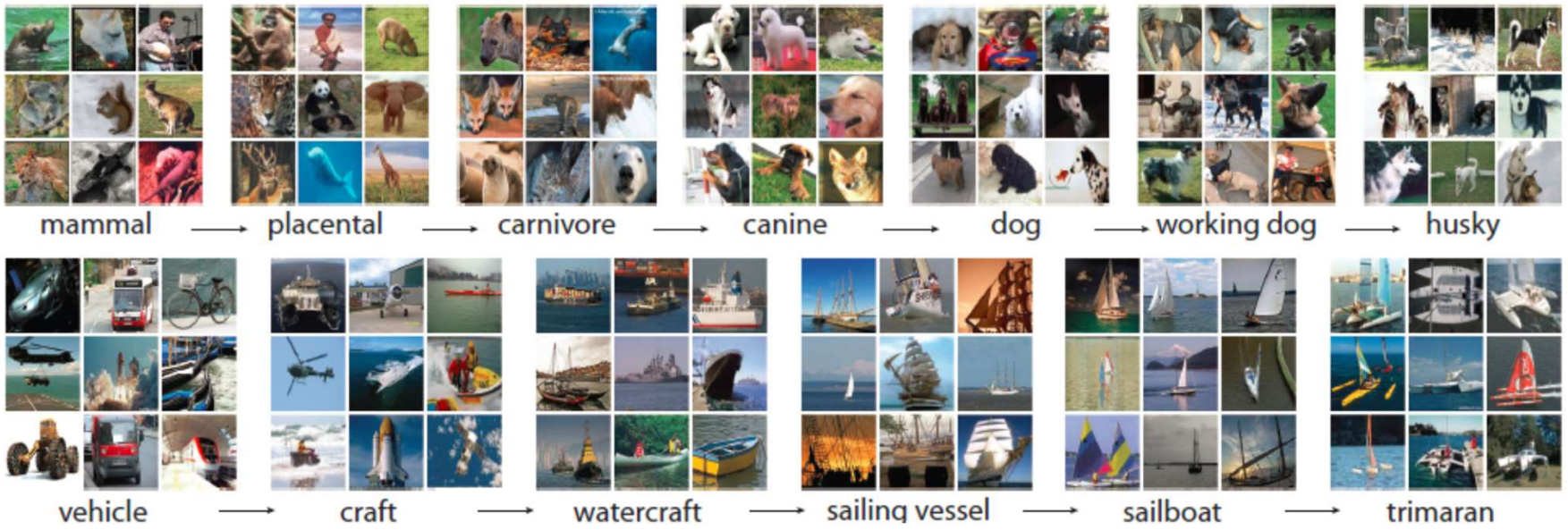


Le-net 5



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

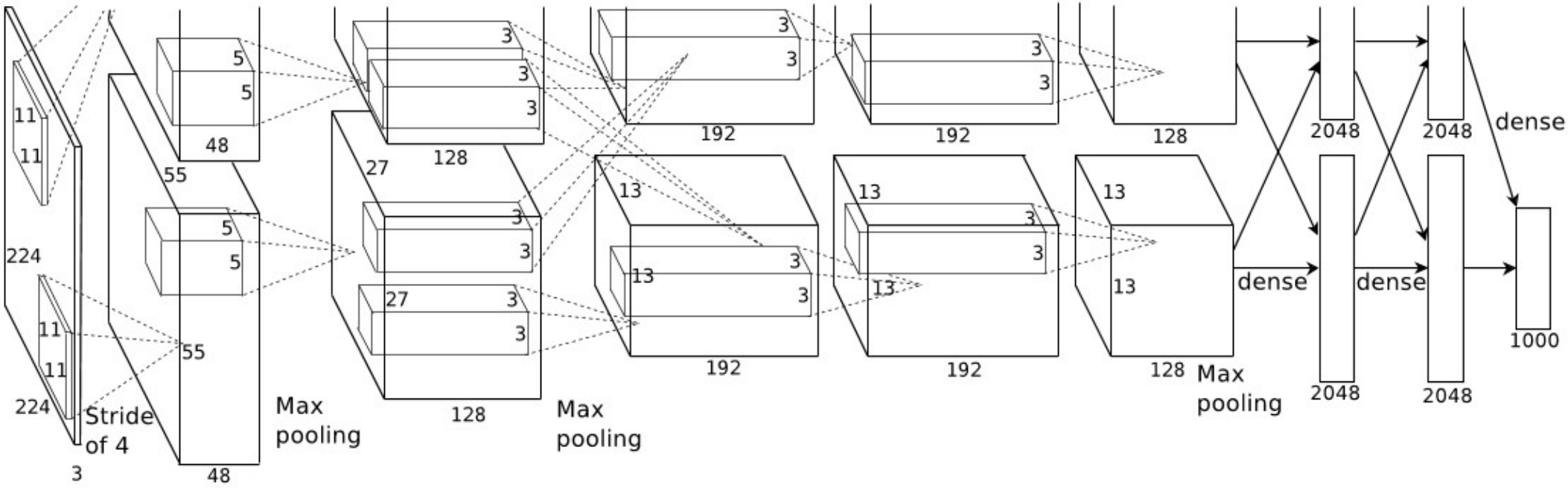
The imagenet task



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

AlexNet

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



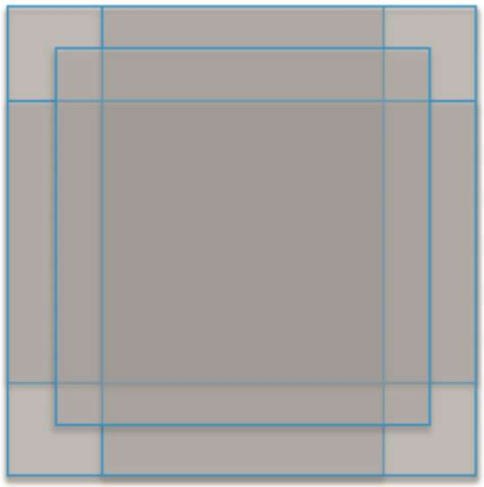
Krizhevsky, A., Sutskever, I. and Hinton, G. E. "ImageNet Classification with Deep Convolutional Neural Networks" NIPS 2012: Neural Information Processing Systems, Lake Tahoe, Nevada

Krizhevsky et. al.

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

Alexnet: Total parameters

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



10 patches

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

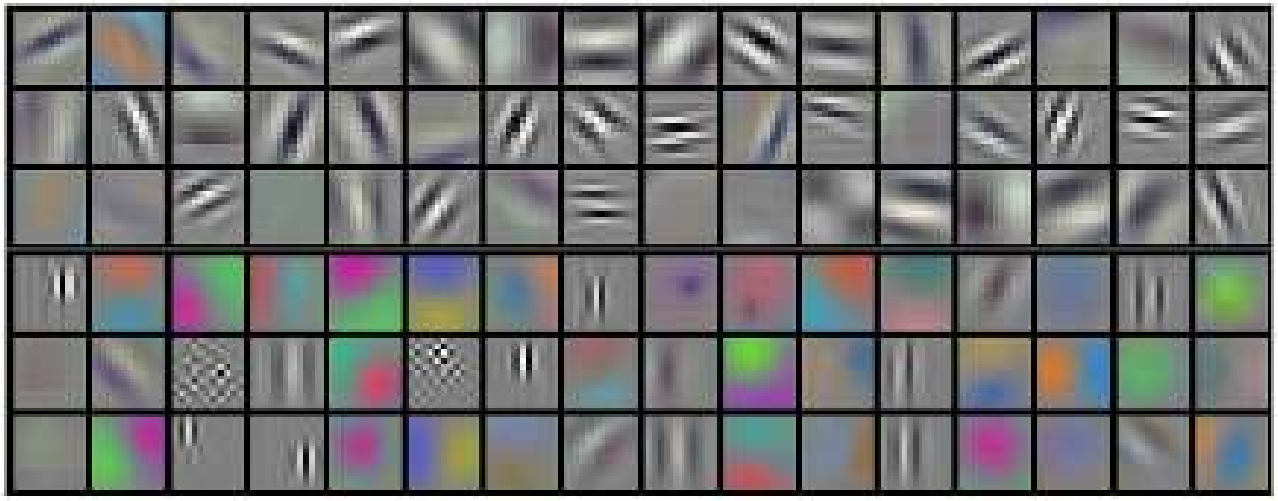
Learning magic in Alexnet

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

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

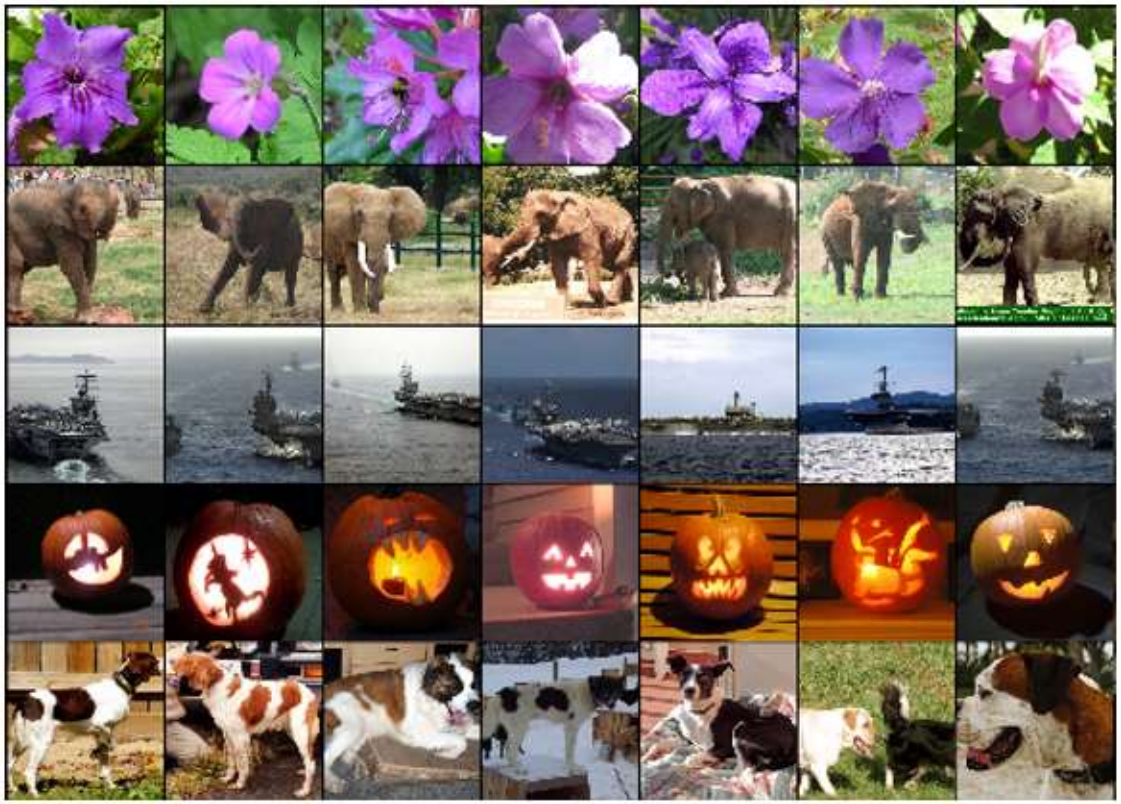
ImageNet

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



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

The net actually *learns* features!

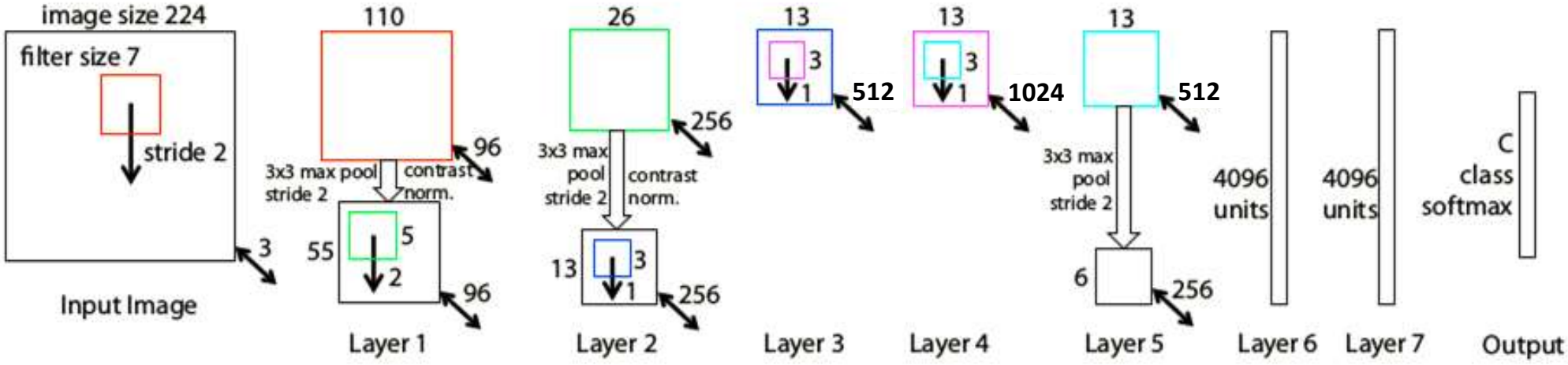


Eight ILSVRC-2010 test images and the five labels considered most probable by our model. The correct label is written under each image, and the probability assigned to the correct label is also shown with a red bar (if it happens to be in the top 5).

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

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

ZFNet



ZF Net Architecture

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

VGGNet

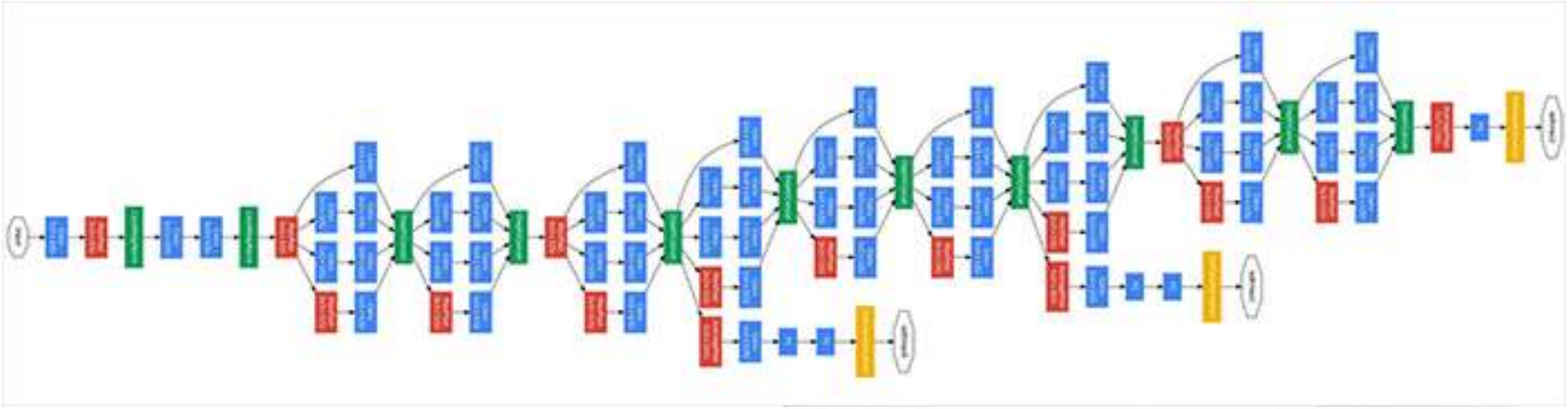
- Simonyan and Zisserman, 2014
- *Only* used 3x3 filters, stride 1, pad 1
- *Only* used 2x2 pooling filters, stride 2

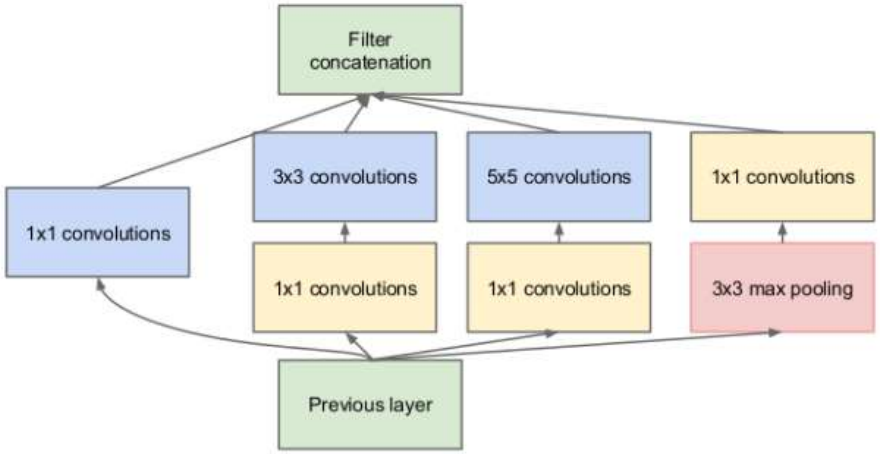
- Tried a large number of architectures.
- Finally obtained 7.3% top-5 error using 13 conv layers and 3 FC layers
  - Combining 7 classifiers
  - Subsequent to paper, reduced error to 6.8% using only two classifiers
- Final arch: 64 conv, 64 conv,
  64 pool,
  128 conv, 128 conv,
  128 pool,
  256 conv,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

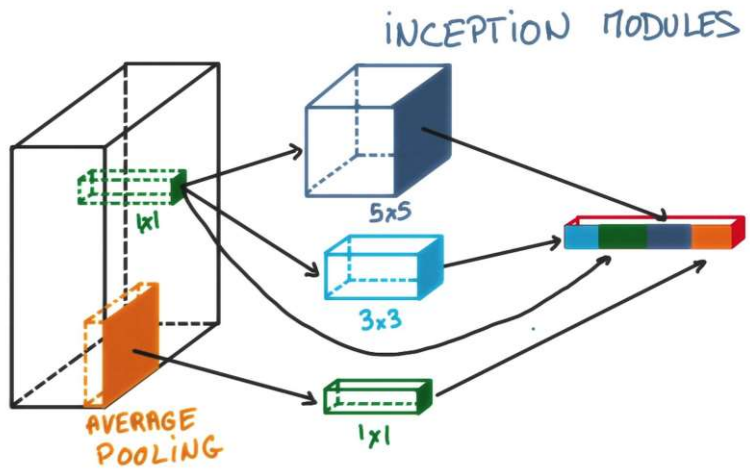
ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128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

Madness!

Googlenet: Inception







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

Resnet

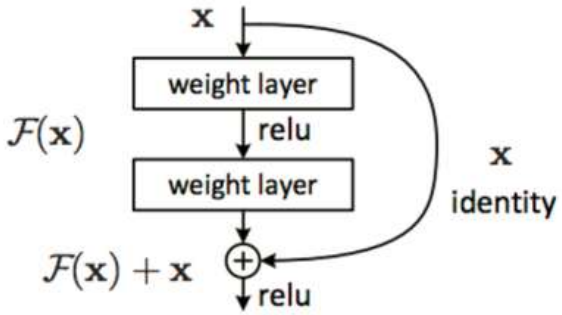
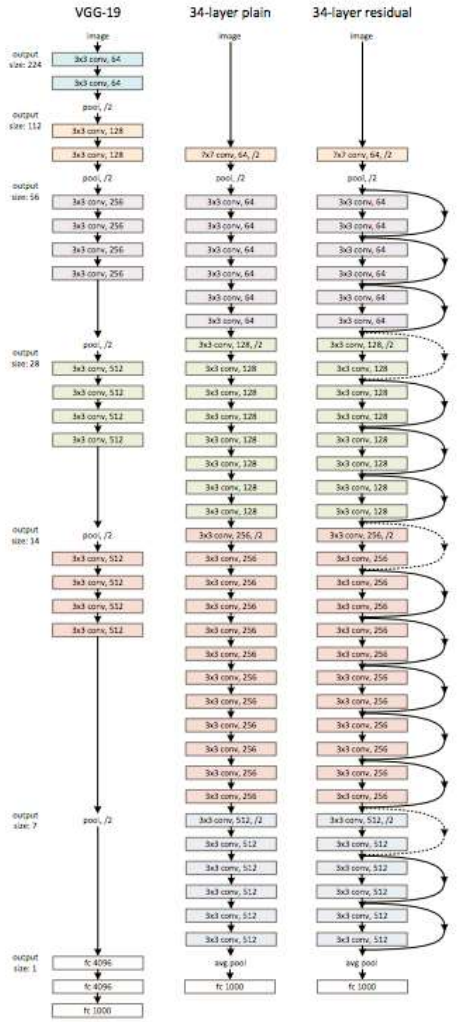


Figure 2. Residual learning: a building block.



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

Resnet details for the curious..

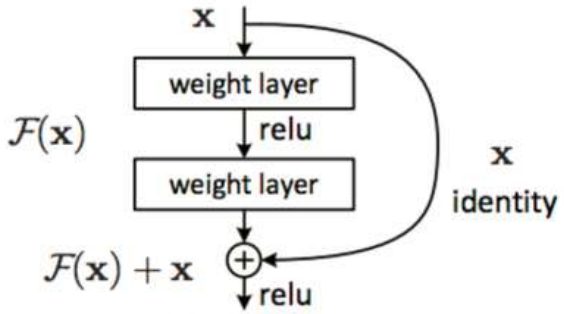
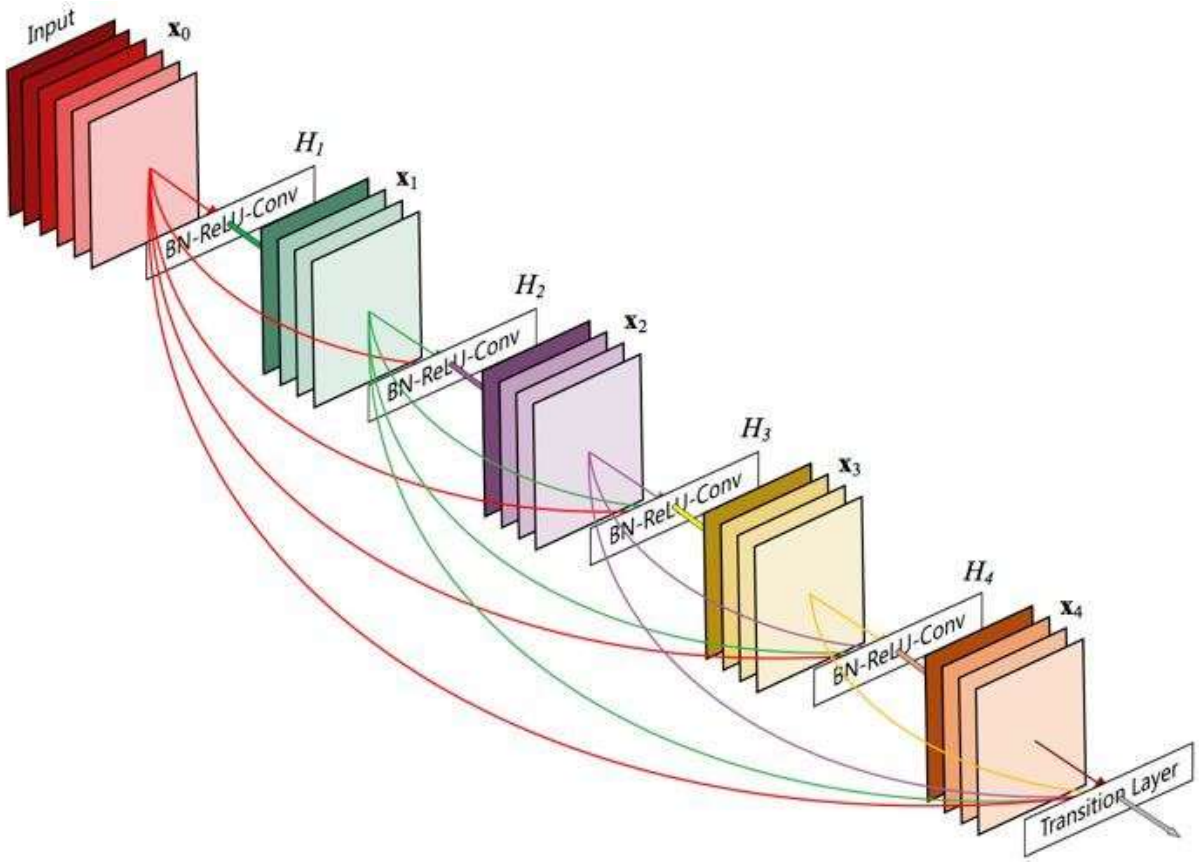


Figure 2. Residual learning: a building block.

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

Densenet



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

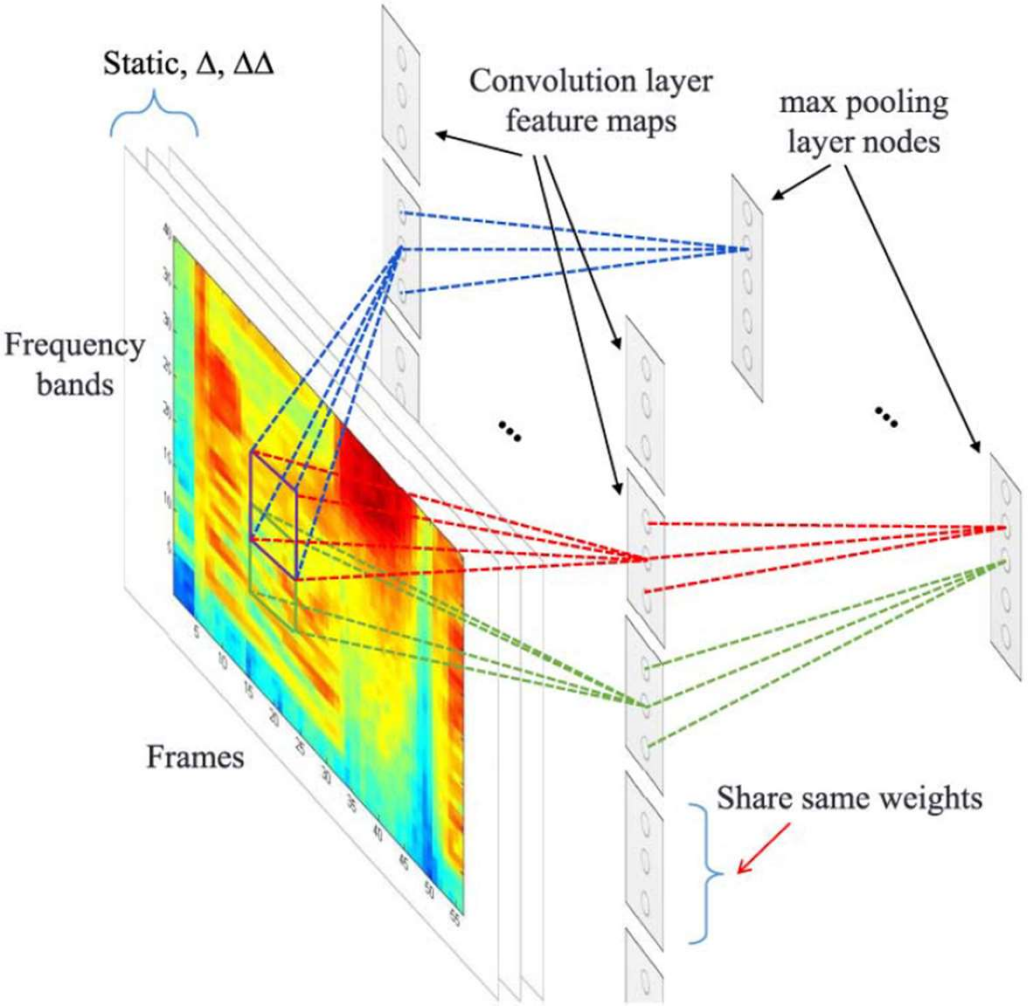
Many many more architectures

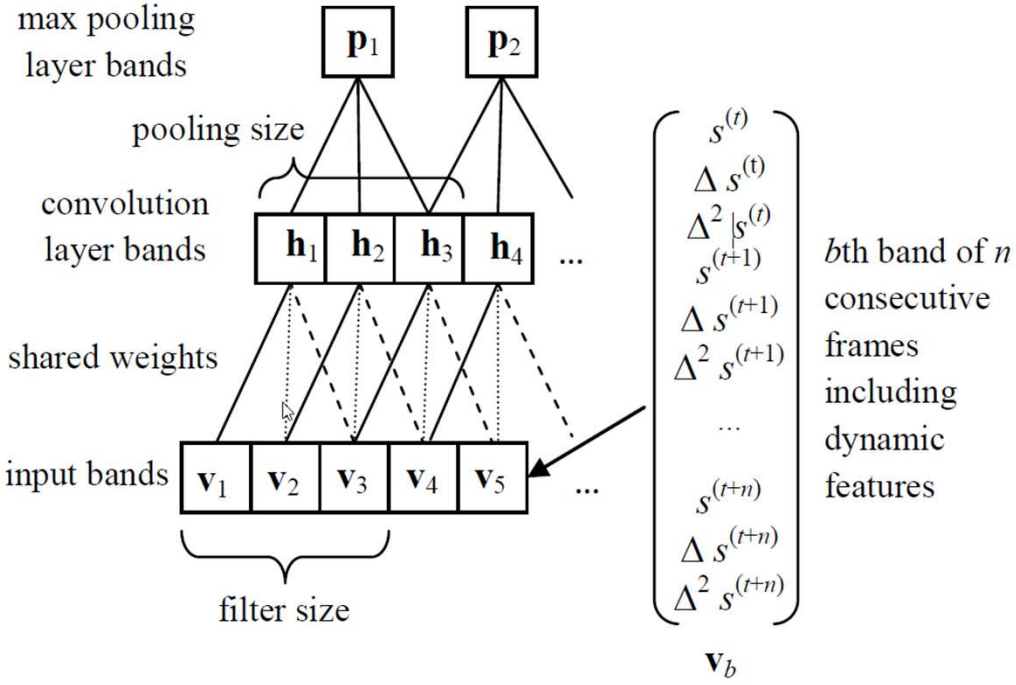
- Daily updates on arxiv..

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

CNN for Automatic Speech Recognition

- Convolution over frequencies
- Convolution over time





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

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

CNN-Recap

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