Deep Neural Networks
Convolutional Networks IV

Bhiksha Raj

Spring 2021

Outline

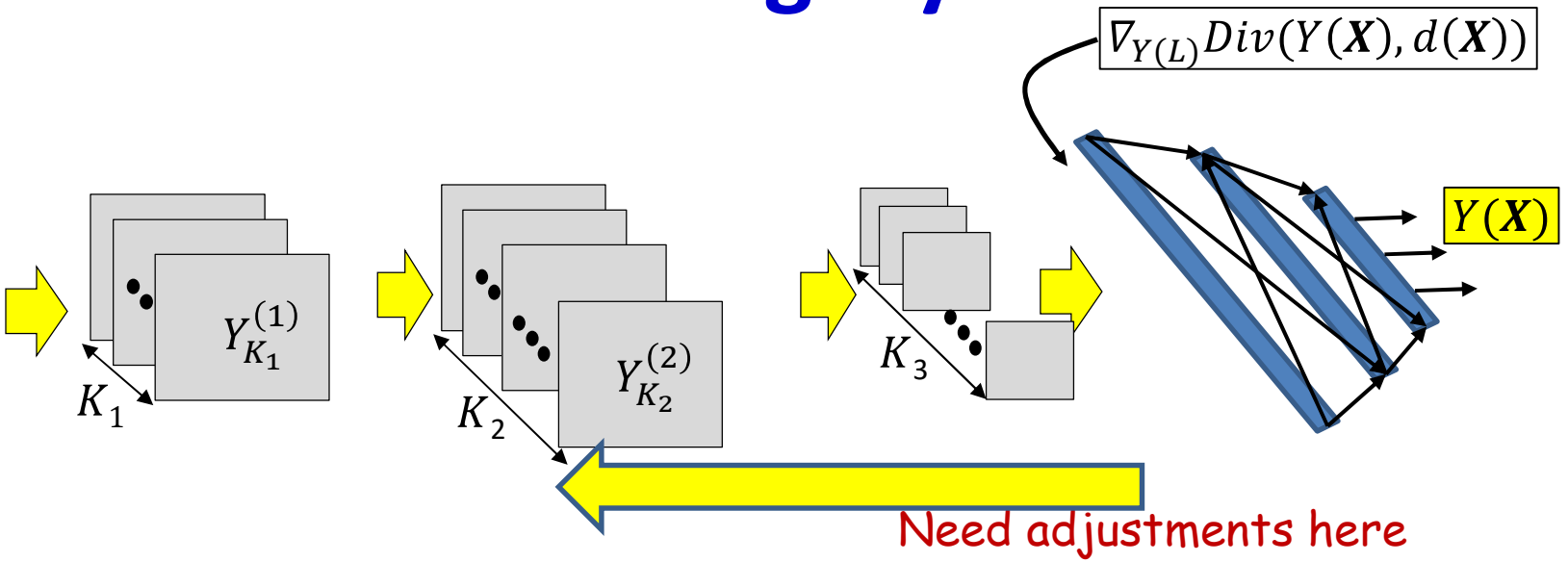
- Quick recap
- Back propagation through a CNN
- Modifications: Transposition, scaling, rotation and deformation invariance
- Segmentation and localization
- Some success stories
- Some advanced architectures
  - Resnet
  - Densenet
  - Transformers and self similarity

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

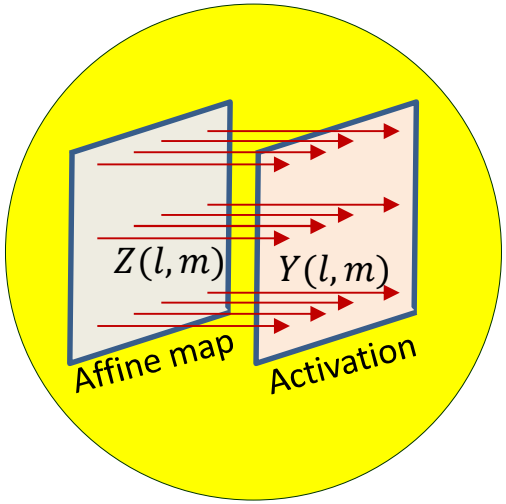
Backpropagation: Convolutional and Pooling layers

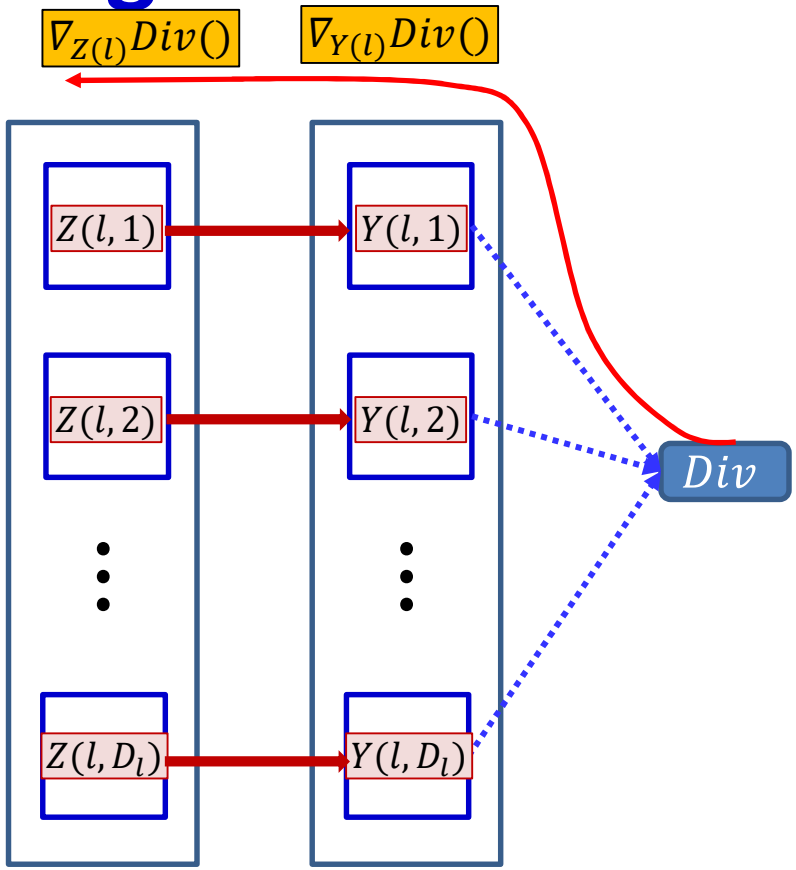
- **Required:**
  - **For convolutional layers**:
    - How to compute the derivatives w.r.t. the affine combination $Z(l)$ maps from the derivatives for the activation output maps $Y(l)$
    - How to compute the derivative w.r.t. $Y(l-1)$ and $w(l)$ given derivatives w.r.t. $Z(l)$
  - **For pooling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

Backpropagation: Convolutional and Pooling layers

- **Required:**
  - **For convolutional layers**:
    - How to compute the derivatives w.r.t. the affine combination $Z(l)$ maps from the derivatives for 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. 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:

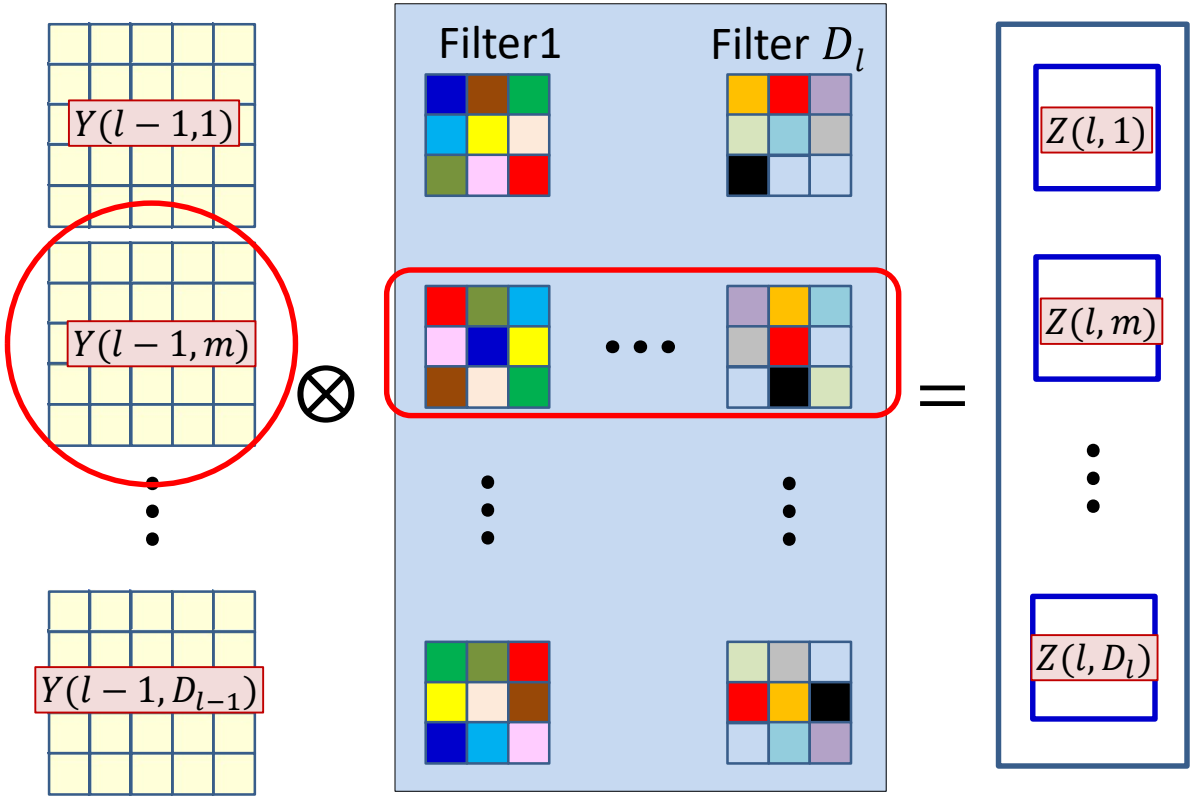
$$\frac{dDiv}{dz(l,m,x,y)} = \frac{dDiv}{d\,y(l,m,x,y)} f'(z(l,m,x,y))$$

  - Simple component-wise computation

Backpropagation: Convolutional and Pooling layers

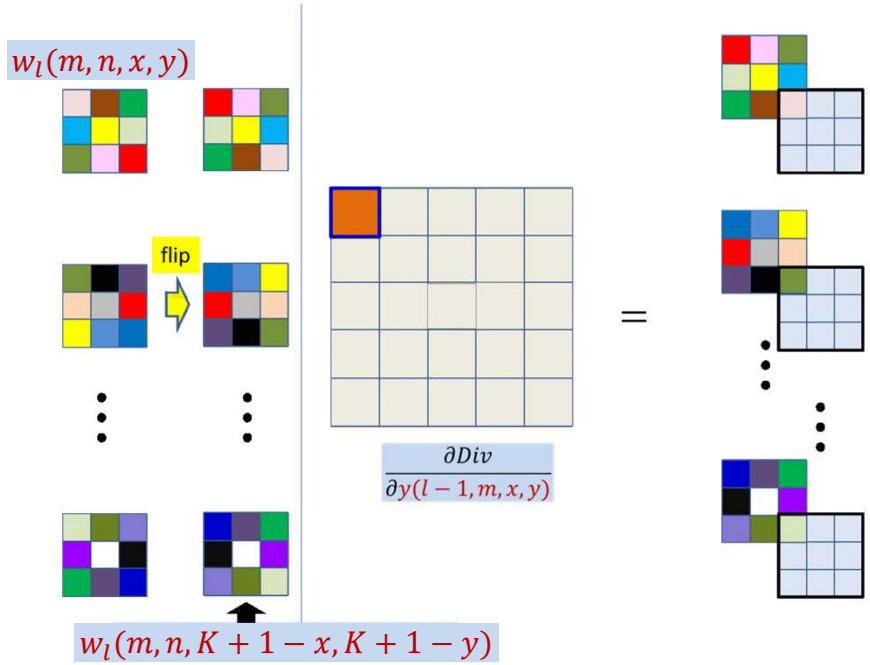
- **Required:**
  - **For convolutional layers**:
    - How to compute the derivatives w.r.t. the affine combination $Z(l)$ maps from the derivatives for the activation output maps $Y(l)$
    - How to compute the derivative w.r.t. $Y(l-1)$ and $w(l)$ given derivatives w.r.t. $Z(l)$
  - **For pooling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

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



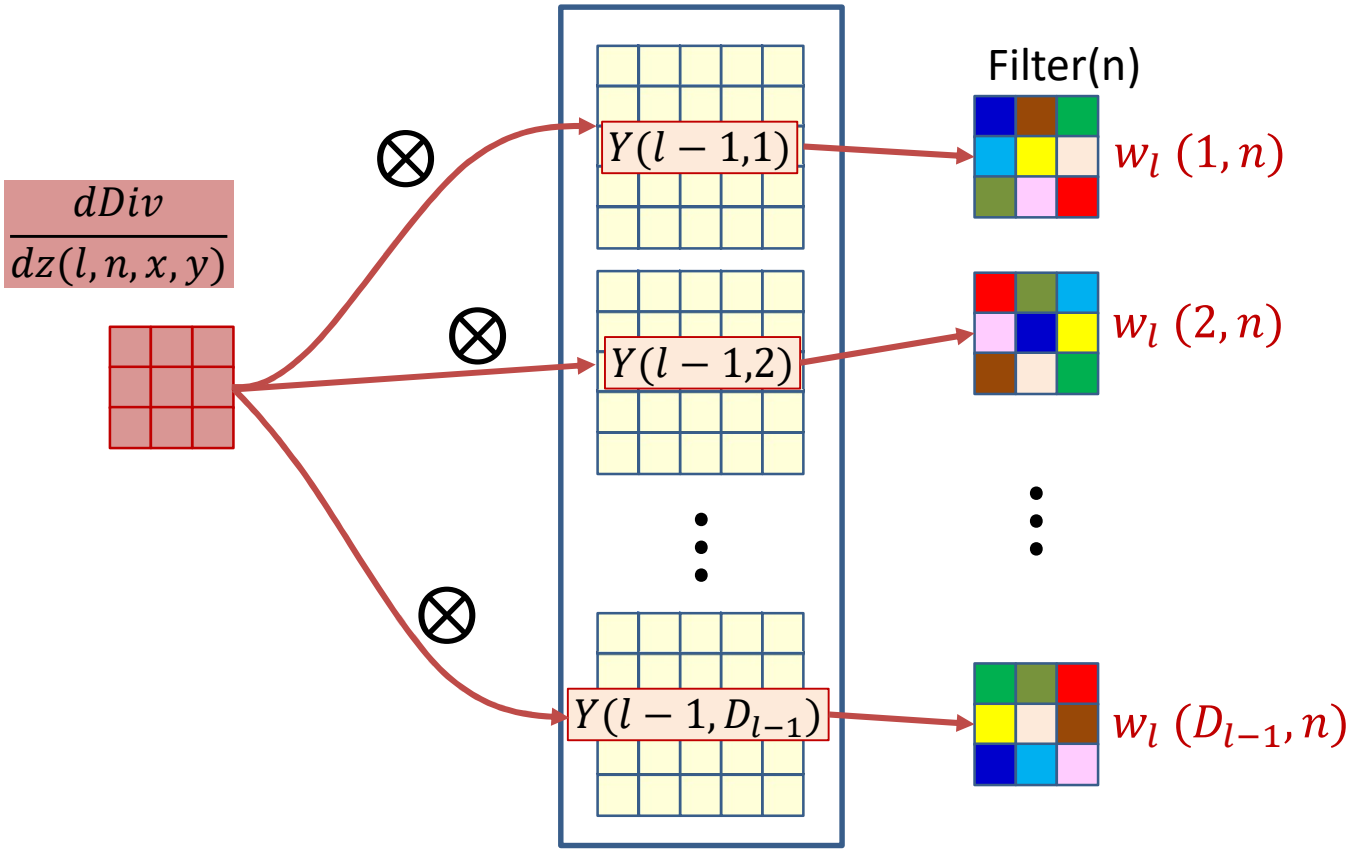
- The D_l affine maps are produced by convolving with D_l filters
- The m^{th} Y map always convolves the m^{th} plane of the filters
- The derivative for the m^{th} Y map will invoke the m^{th} plane of *all* the filters

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



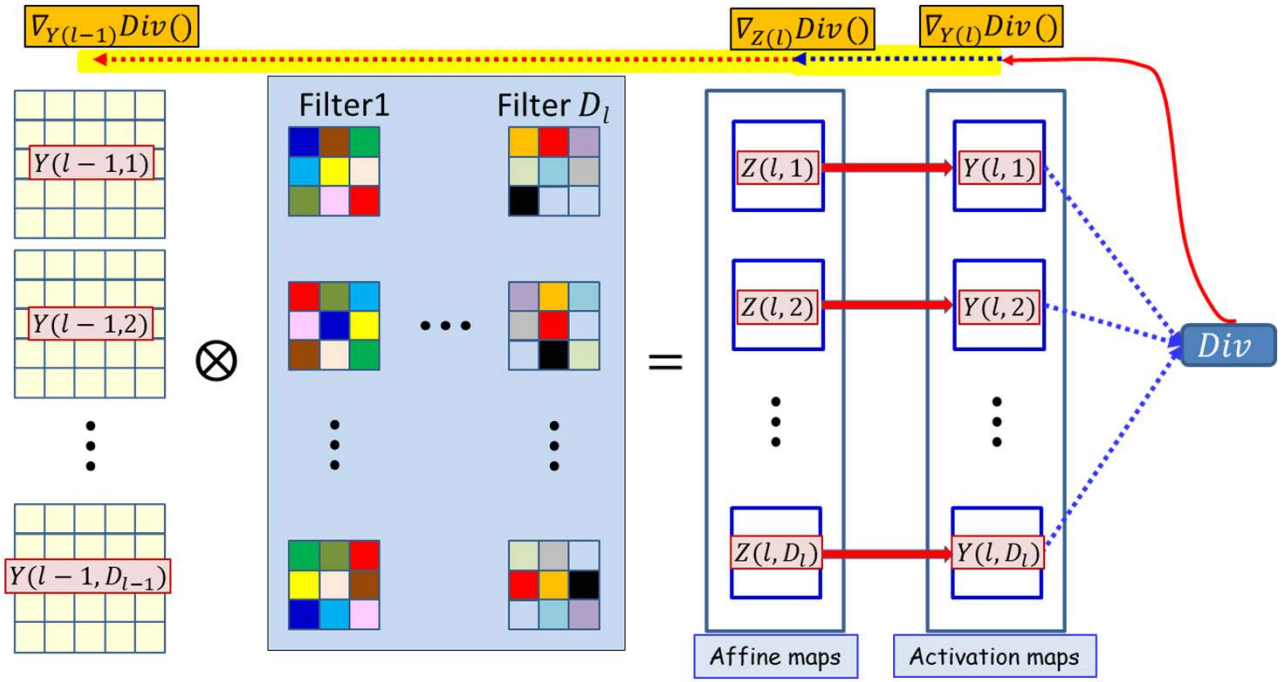
- This is just a convolution of the zero-padded $\frac{\partial Div}{\partial z(l,n,x,y)}$ maps by the transposed and flipped filter
  - After zero padding it first with $K-1$ zeros on every side

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} "plane" 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. $Z(l)$

CNN: Forward

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

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

Switching to 1-based indexing with appropriate adjustments

Backward layer l

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

Complete Backward (no pooling)

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

Complete Backward (no pooling)

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

Multiple ways of recasting this as tensor/ vector operations.

Will not discuss here

Complete Backward (with strides)

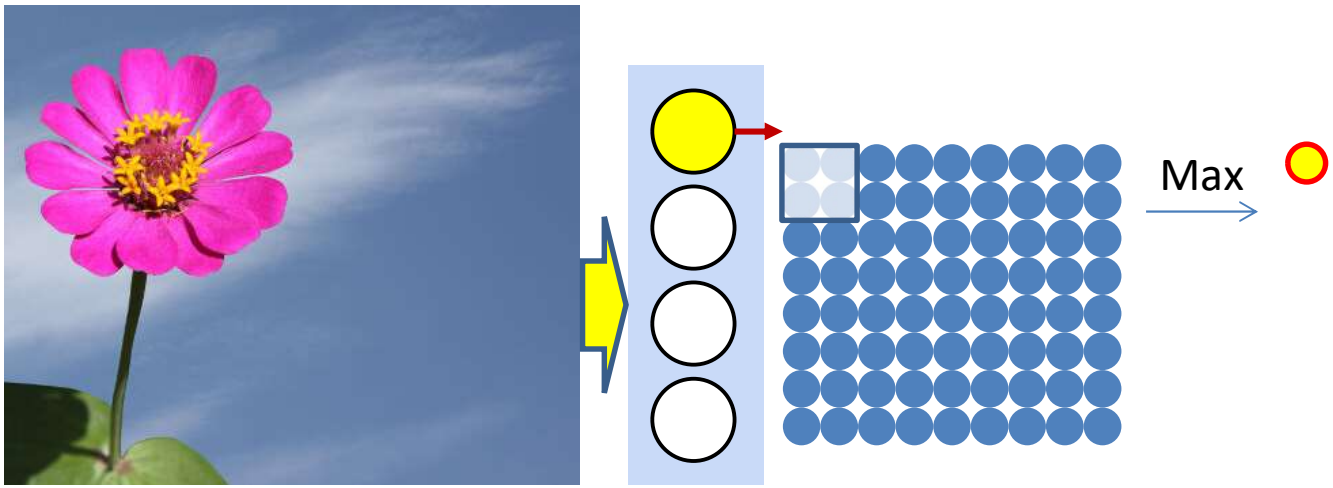
```
dY(L) = dDiv/dY(L)
for l = L:1  # Backward through layers
    dw(l) = zeros(D_l x D_{l-1} x K_l x K_l)
    dY(l-1) = zeros(D_{l-1} x W_{l-1} x H_{l-1})
    for x = W_l:-stride:1
        m = (x-1)stride
        for y = H_l:-stride:1
            n = (y-1)stride
            for j = D_l:-1:1
       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

Backpropagation: Convolutional and Pooling layers

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

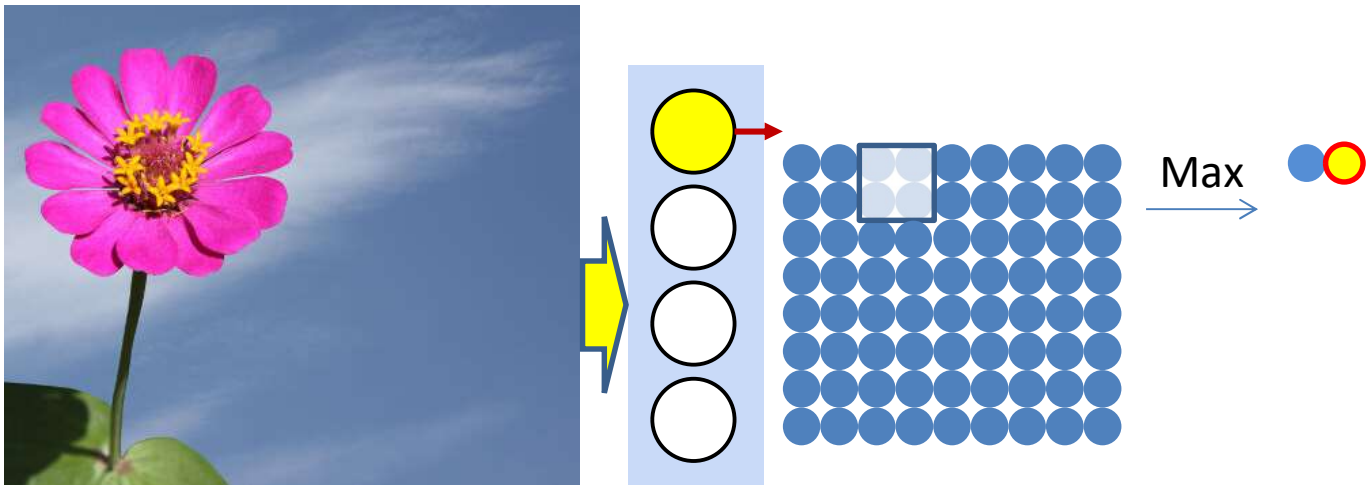
- **Required:**
  - **For convolutional layers**:
    - How to compute the derivatives w.r.t. the affine combination $Z(l)$ maps from the activation output maps $Y(l)$
    - How to compute the derivative w.r.t. $Y(l-1)$ and $w(l)$ given derivatives w.r.t. $Z(l)$
  - **For pooling layers**:
    - How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

Pooling and downsampling



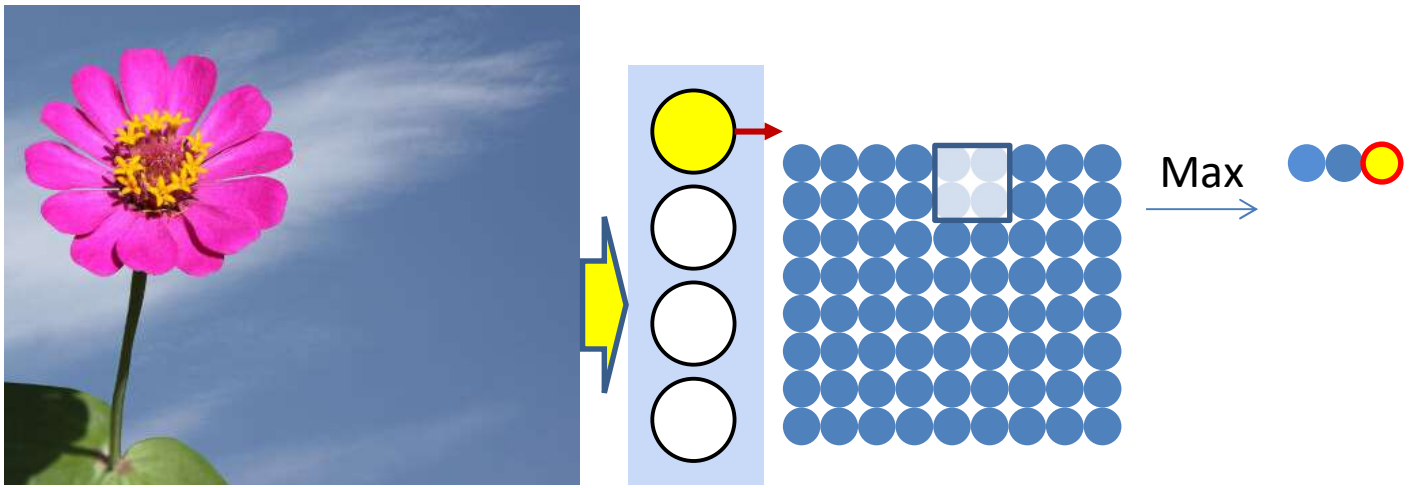
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



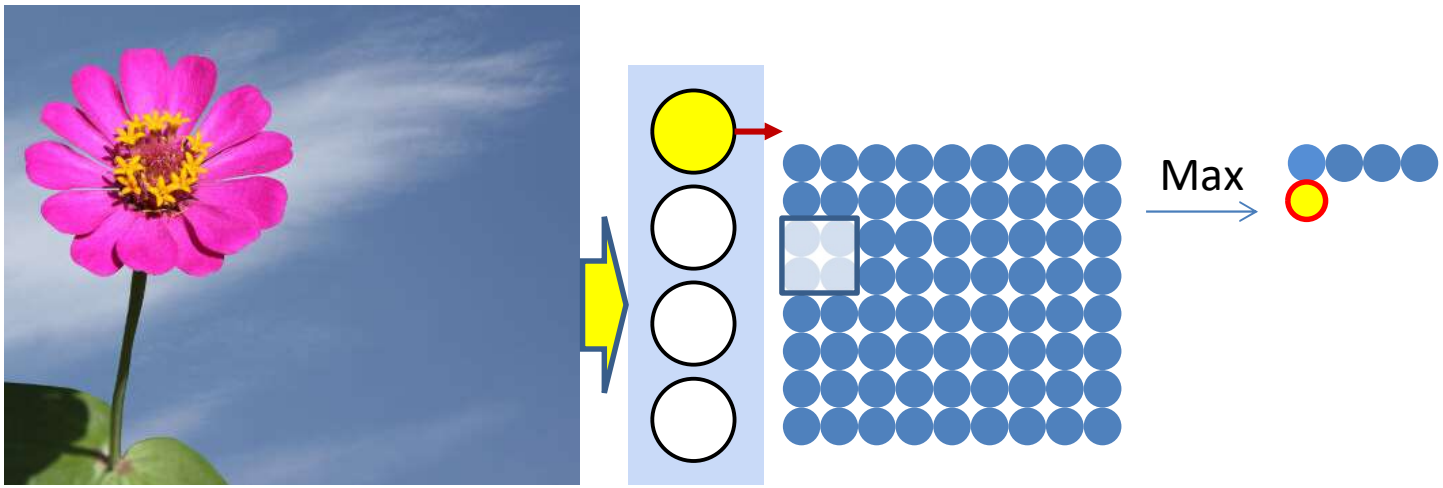
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



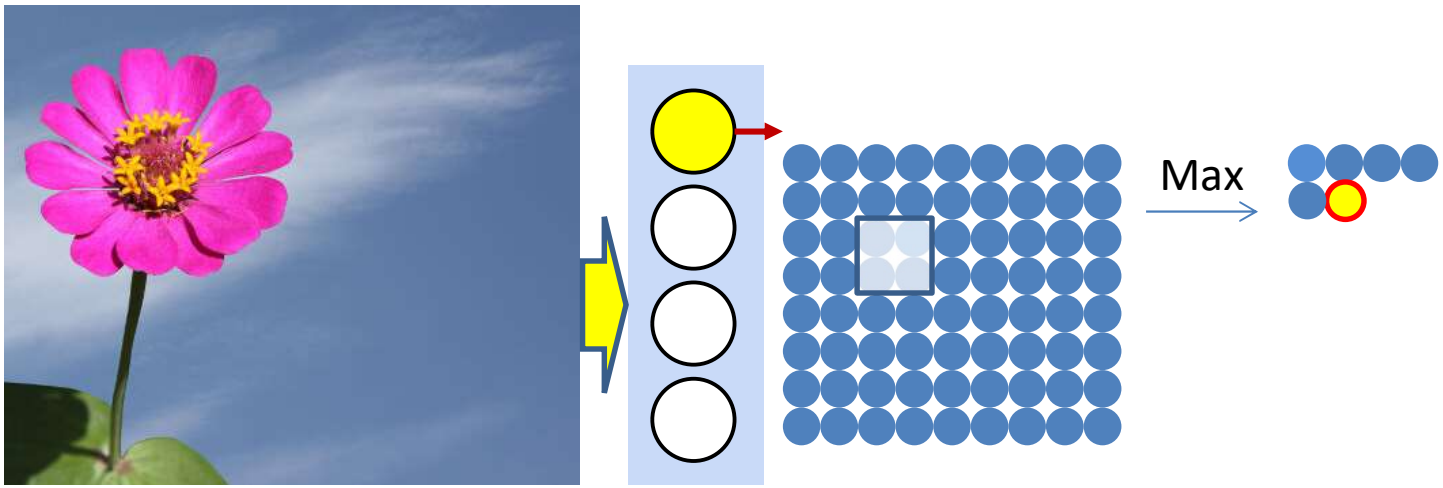
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



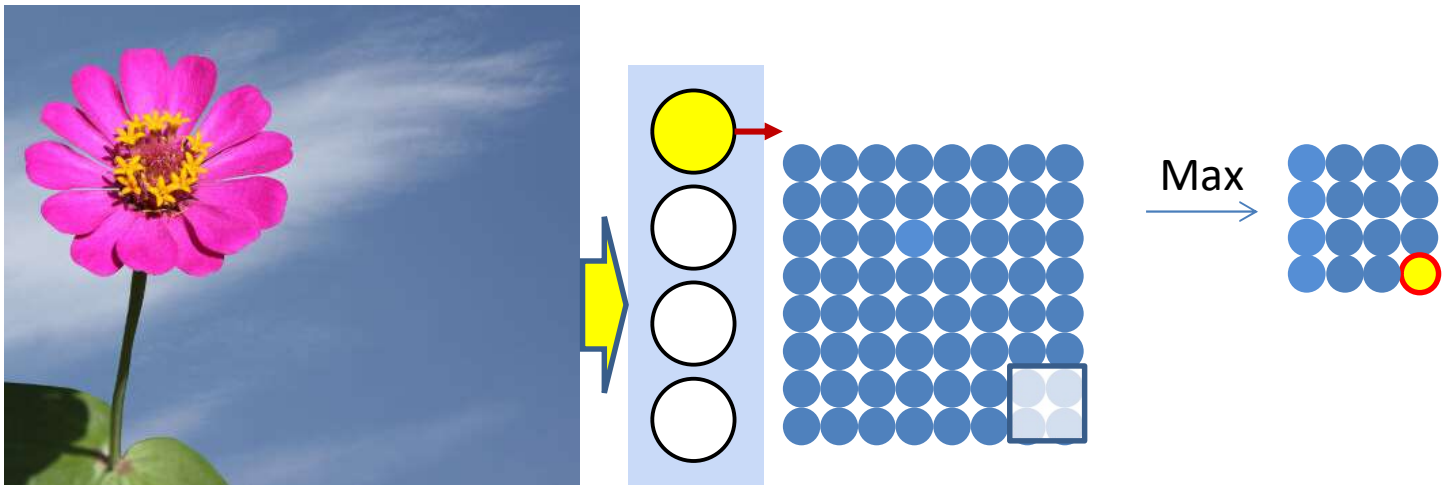
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



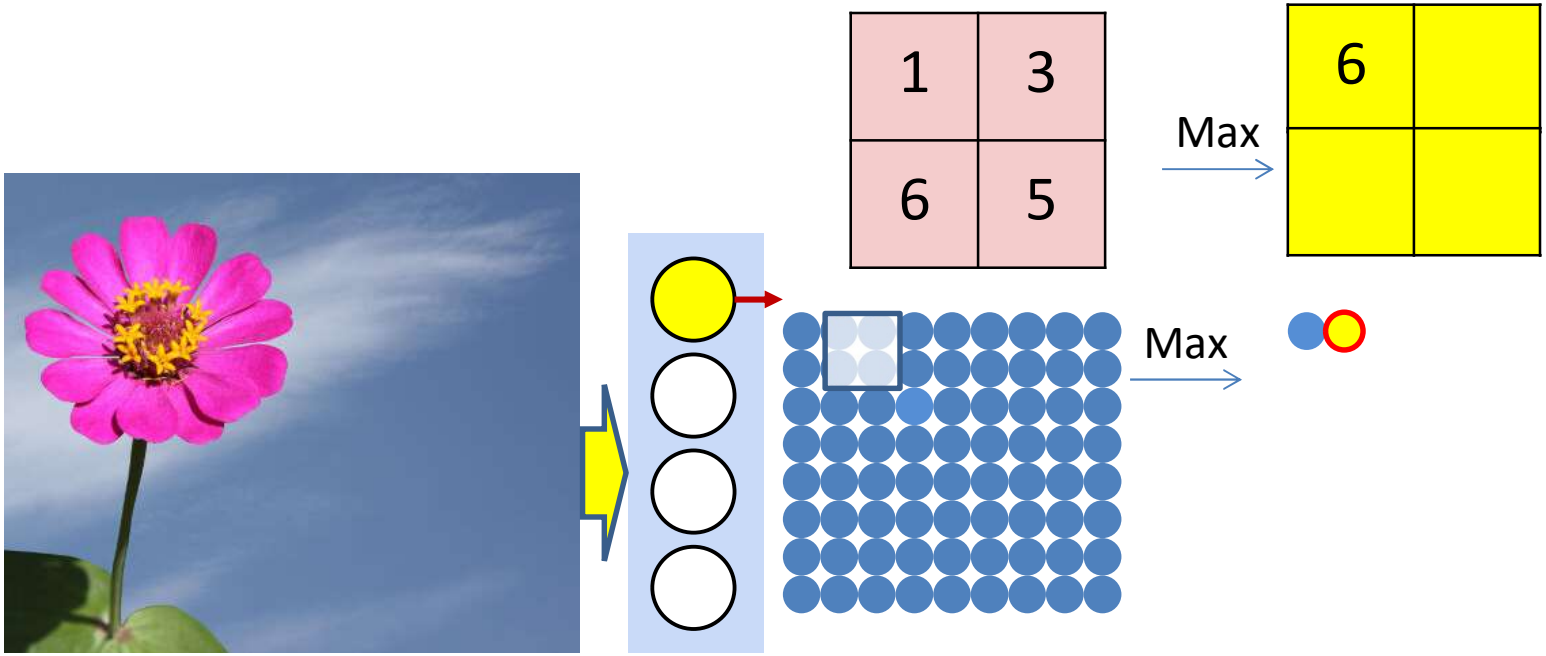
- Pooling is typically performed with strides > 1
  - Results in shrinking of the map
  - “Downsampling”

Pooling and downsampling



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

Max pooling

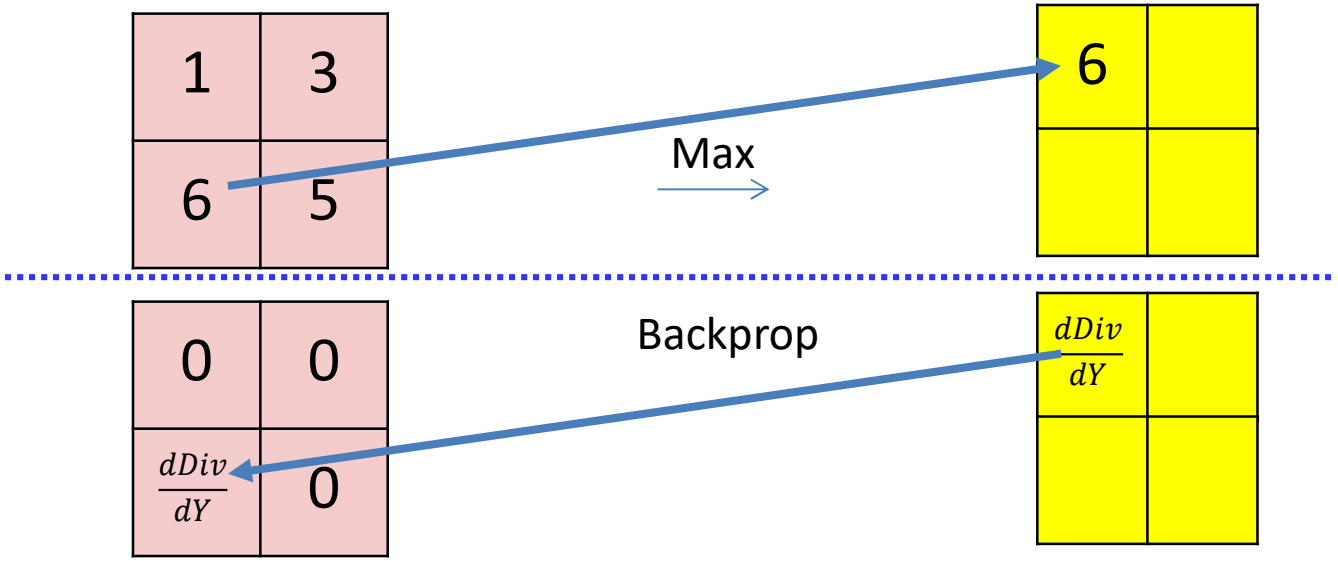


- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

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

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

Derivative of Max pooling



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

- Max pooling selects the largest from a pool of elements

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

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

Max Pooling layer at layer l

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

Max pooling

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

Derivative of max pooling layer at layer l

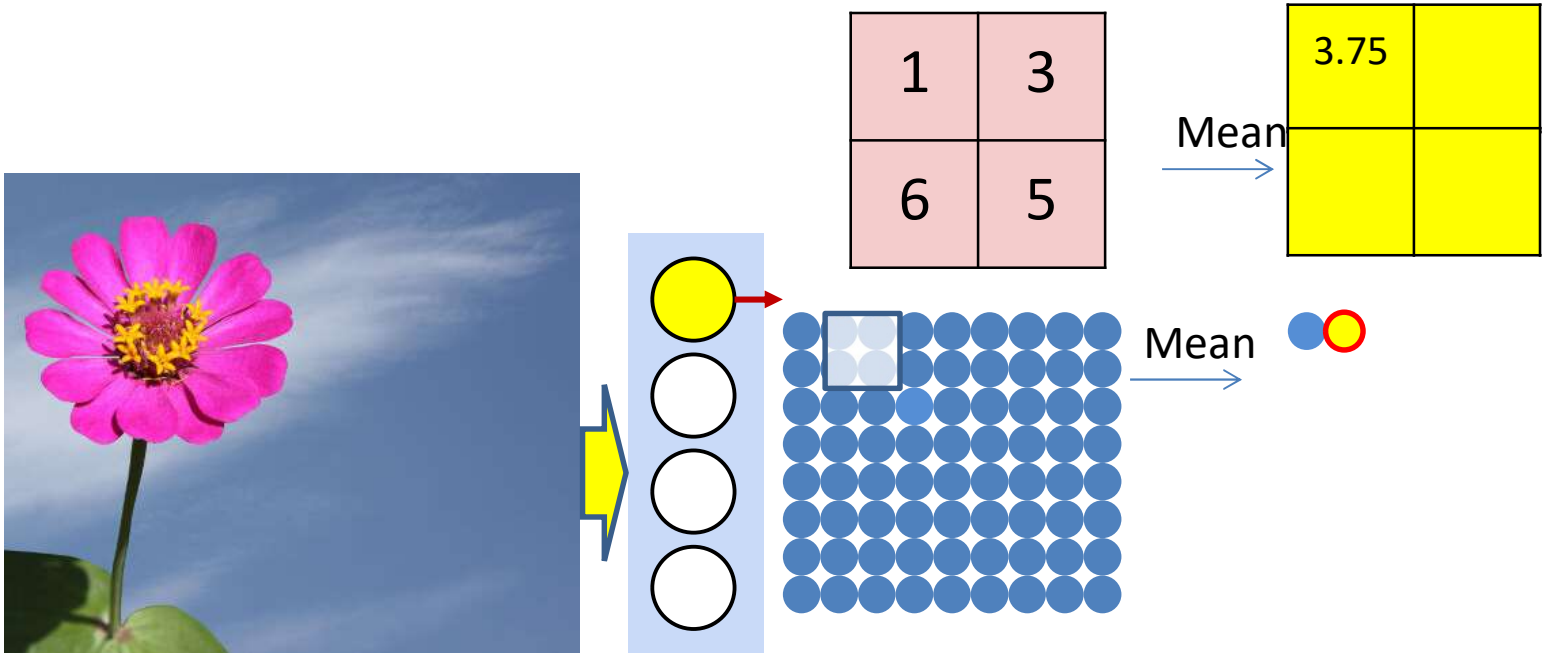
a) Performed separately for every map (j).
*) Not combining multiple maps within a single max operation.
b) Keeping track of location of max

Max pooling

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

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

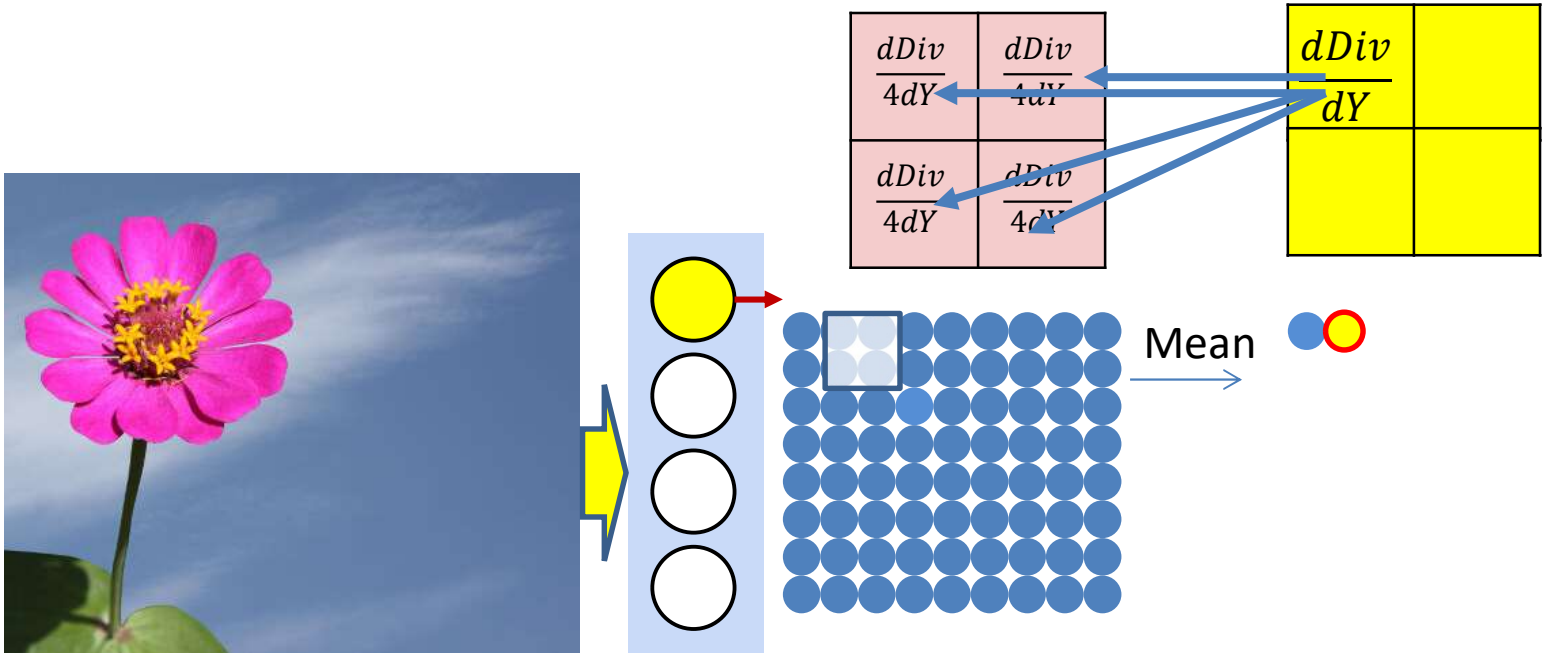
Mean pooling



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

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

Derivative of mean pooling



- The derivative of mean pooling is distributed over the pool

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

Mean Pooling layer at layer l

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

Mean pooling

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

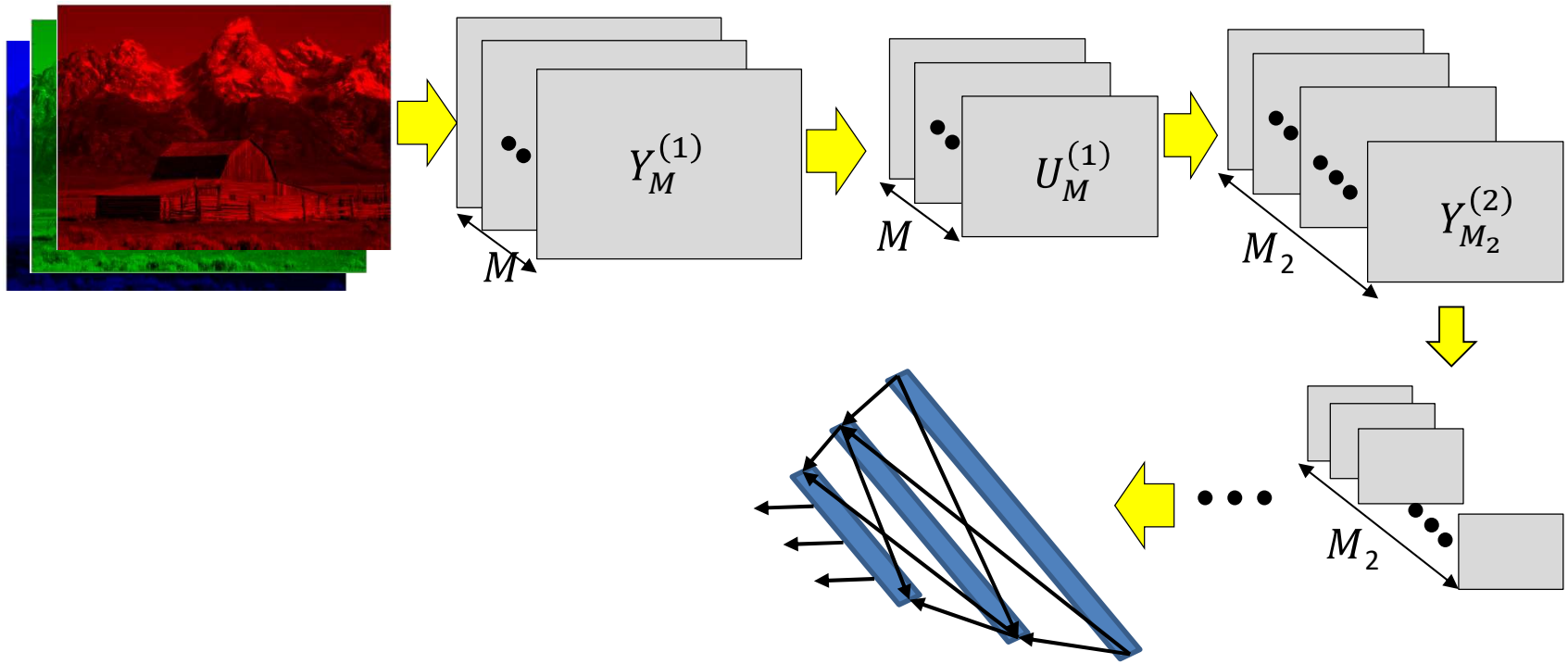
Derivative of mean pooling layer at layer l

Mean pooling

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

"+=" because adjacent windows may overlap

Learning the network

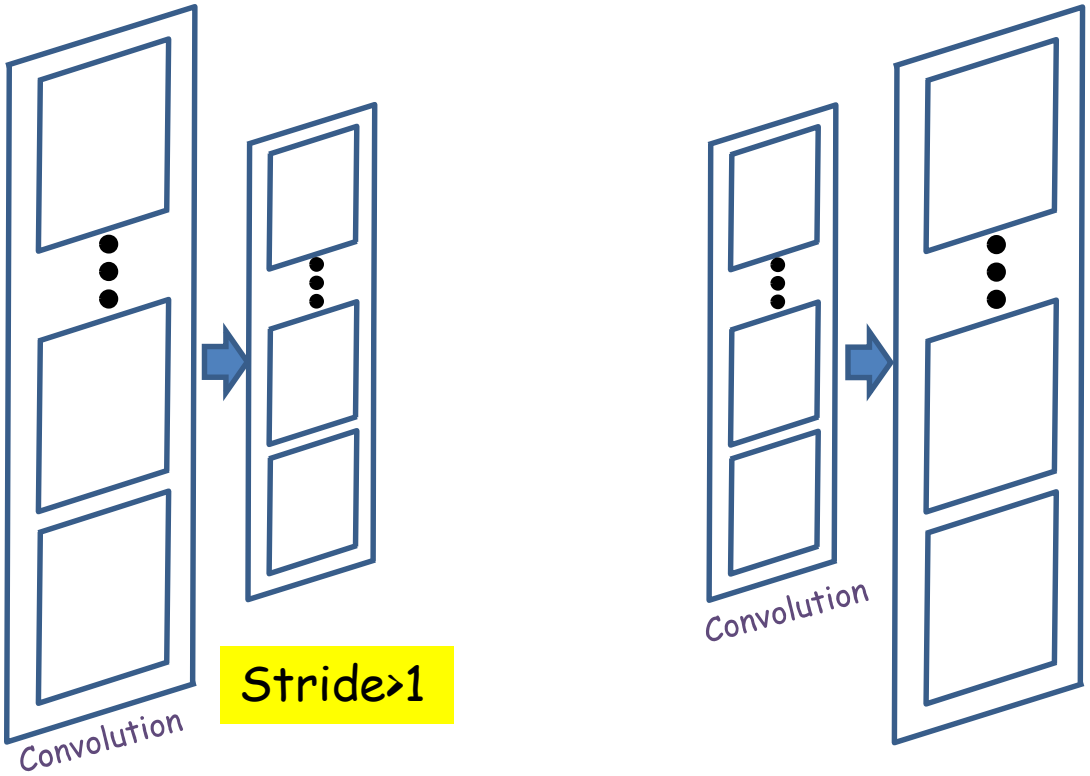


- Have shown the derivative of divergence w.r.t every intermediate output, and every free parameter (filter weights)
- Can now be embedded in gradient descent framework to learn the network

Story so far

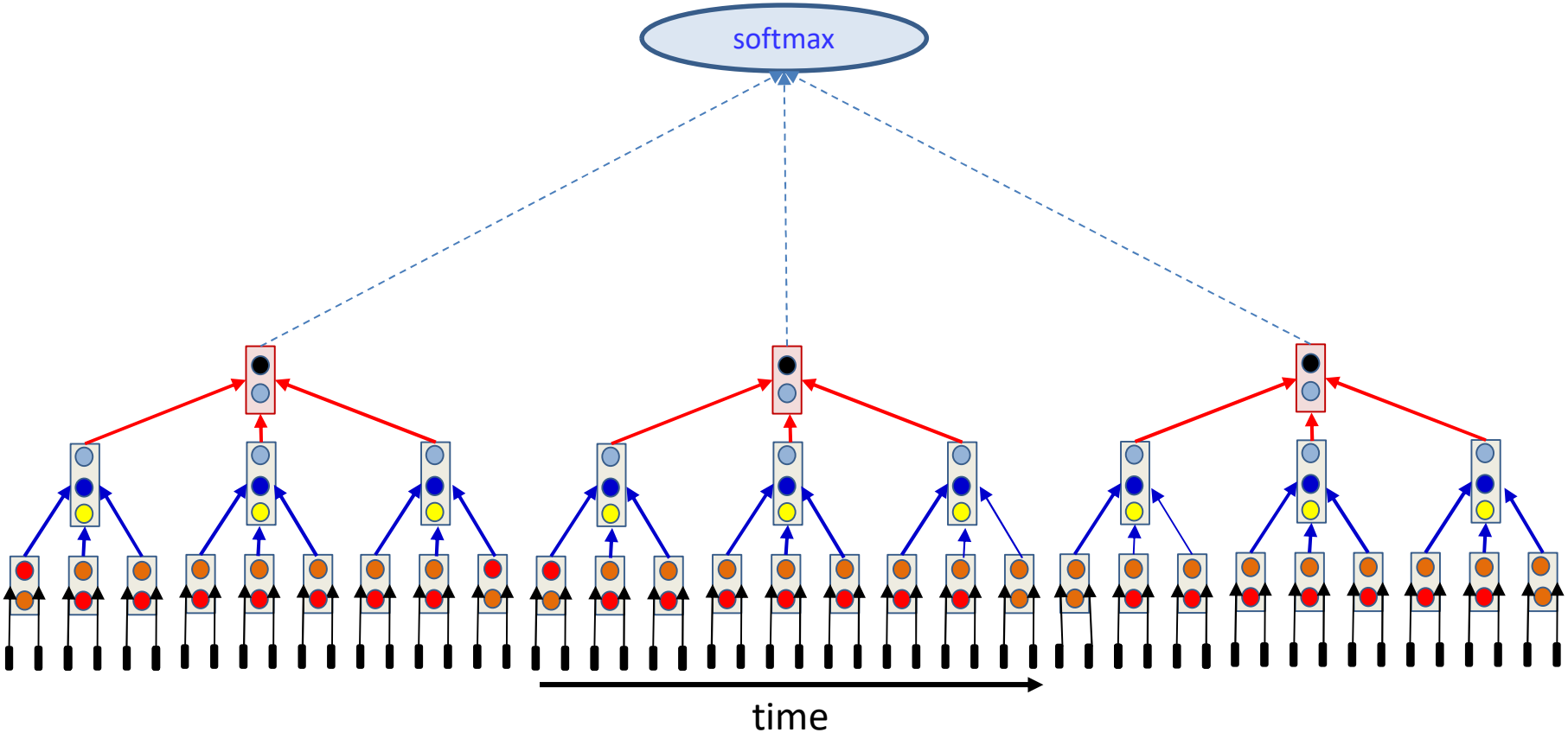
- The convolutional neural network is a supervised version of a computational model of mammalian vision
- It includes
  - Convolutional layers comprising learned filters that scan the outputs of the previous layer
  - Downsampling layers that operate over groups of outputs from the convolutional layer to reduce network size
- The parameters of the network can be learned through regular back propagation
  - Maxpooling layers must propagate derivatives only over the maximum element in each pool
    - Other pooling operators can use regular gradients or subgradients
  - Derivatives must sum over appropriate sets of elements to account for the fact that the network is, in fact, a shared parameter network

An implicit assumption



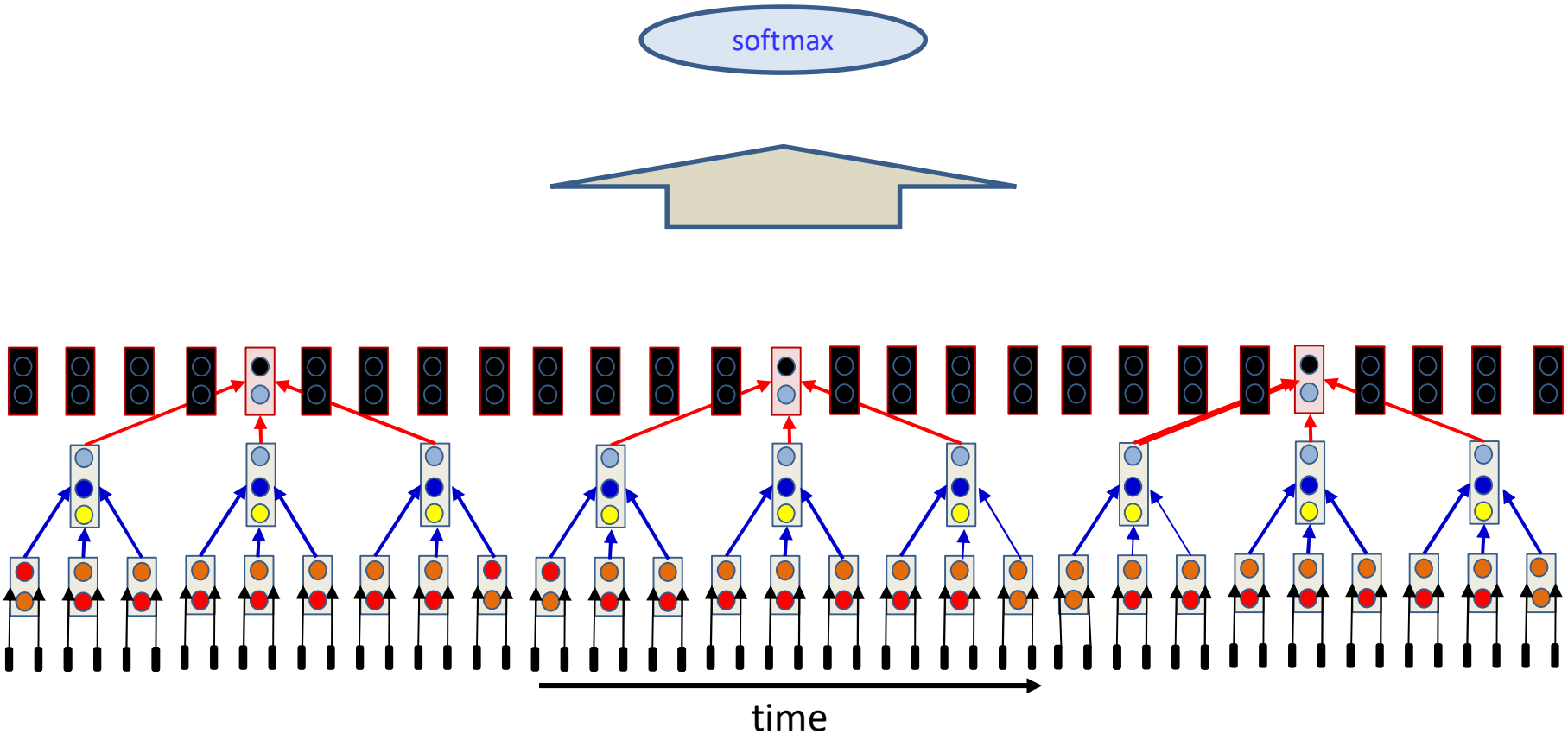
- We've always assumed that subsequent steps *shrink* the size of the maps
- Can subsequent maps *increase* in size?

1-D scans



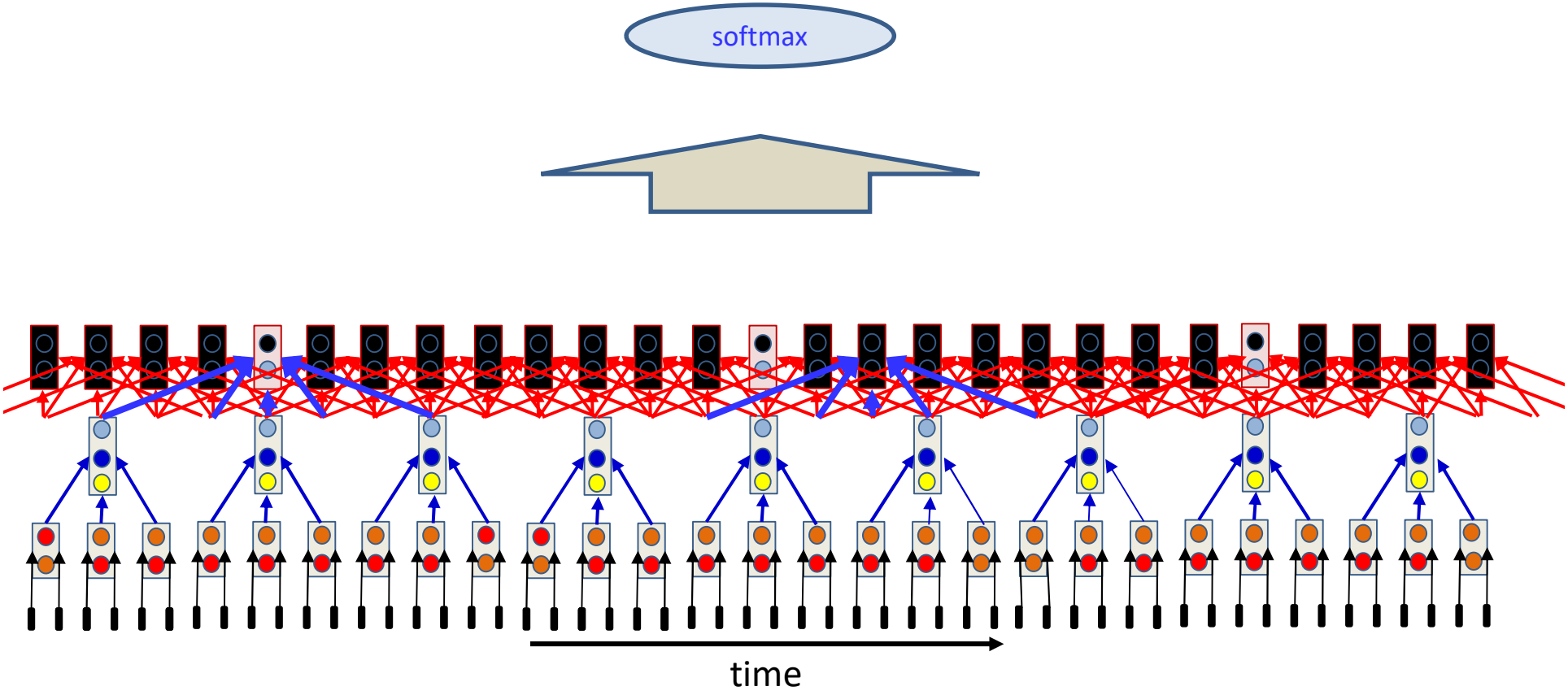
- The number of "bars" in each layer is usually the same or *smaller* than the bars in the previous layer
  - Scanning maintains or reduces the time resolution of the signal at each layer

Upsampling 1-D scans



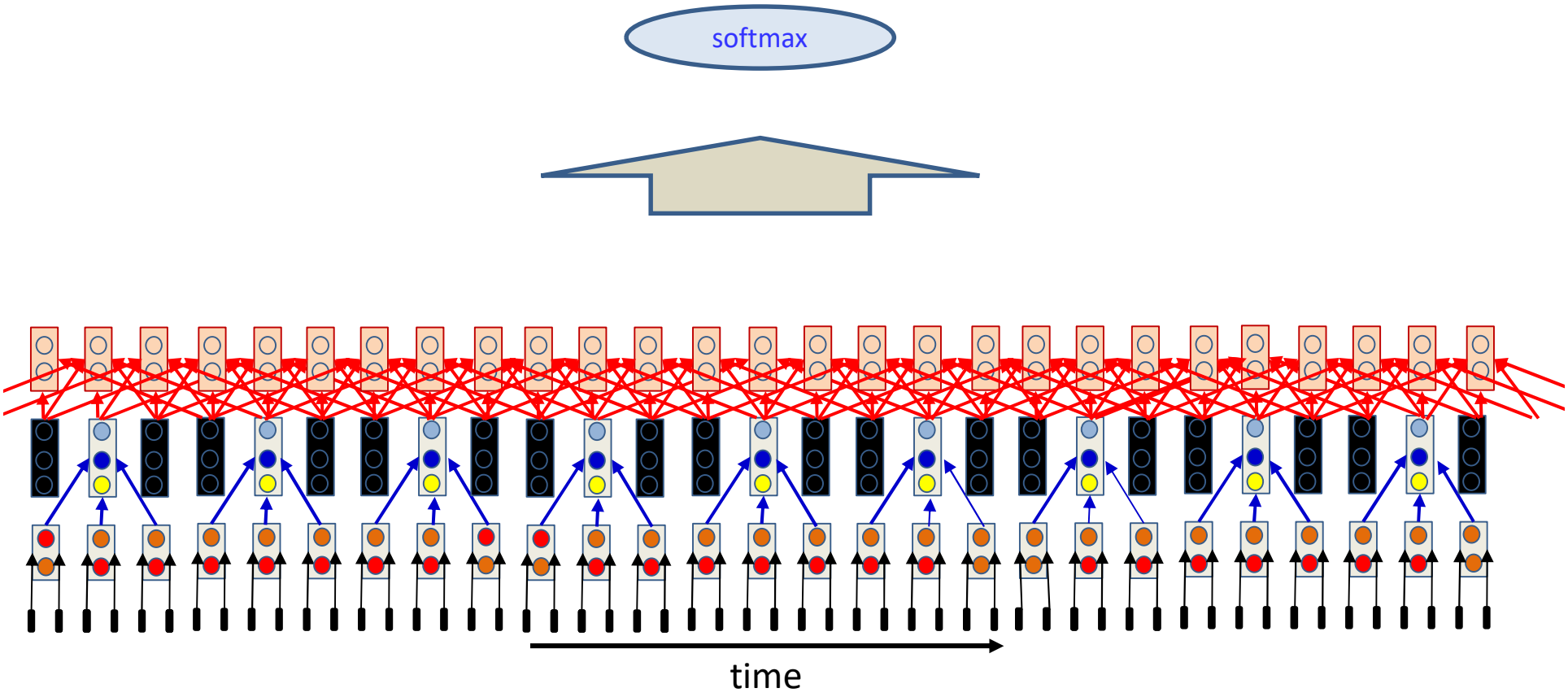
- The number of "bars" in each layer is usually the same or *smaller* than the bars in the previous layer
  - Scanning maintains or reduces the time resolution of the signal at each layer
- What if we want to *increase the* time resolution with layers?

Upsampling 1-D scans



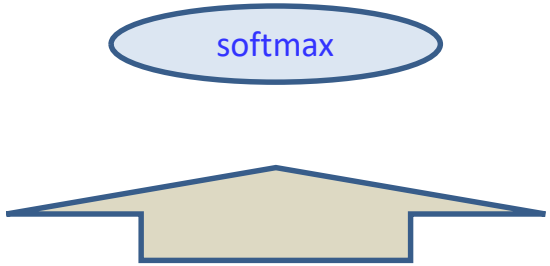
- **Problem:** The values required to compute the intermediate values are missing from the previous layer!

Upsampling 1-D scans

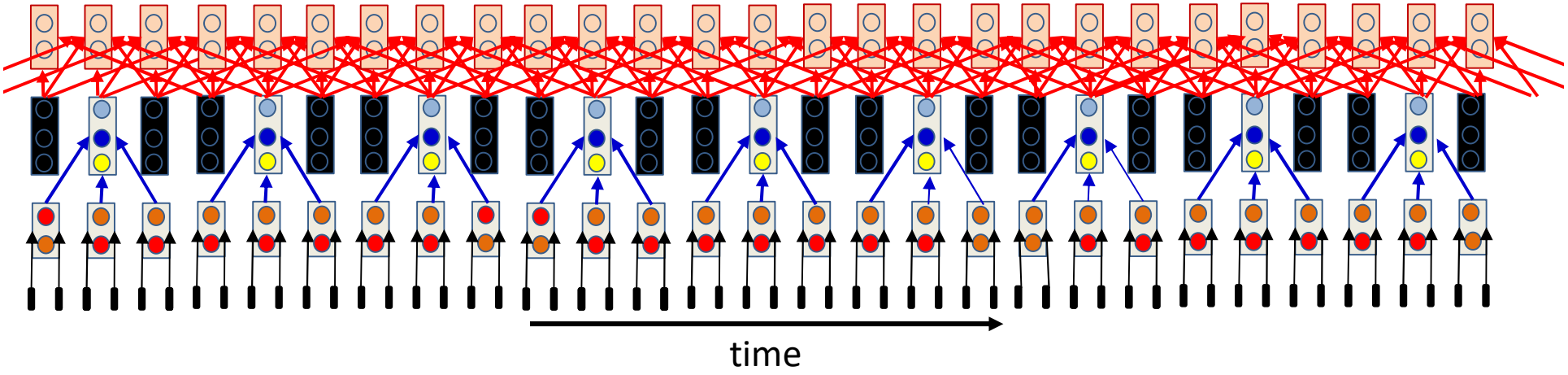


- **Problem:** The values required to compute the intermediate values are missing from the previous layer!
- **Solution:** Synthetically fill in the missing intermediate values of the previous layer
  - With zeros
    - Could also fill them in with linear or spline interpolation of neighbors, but it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

Upsampling 1-D scans

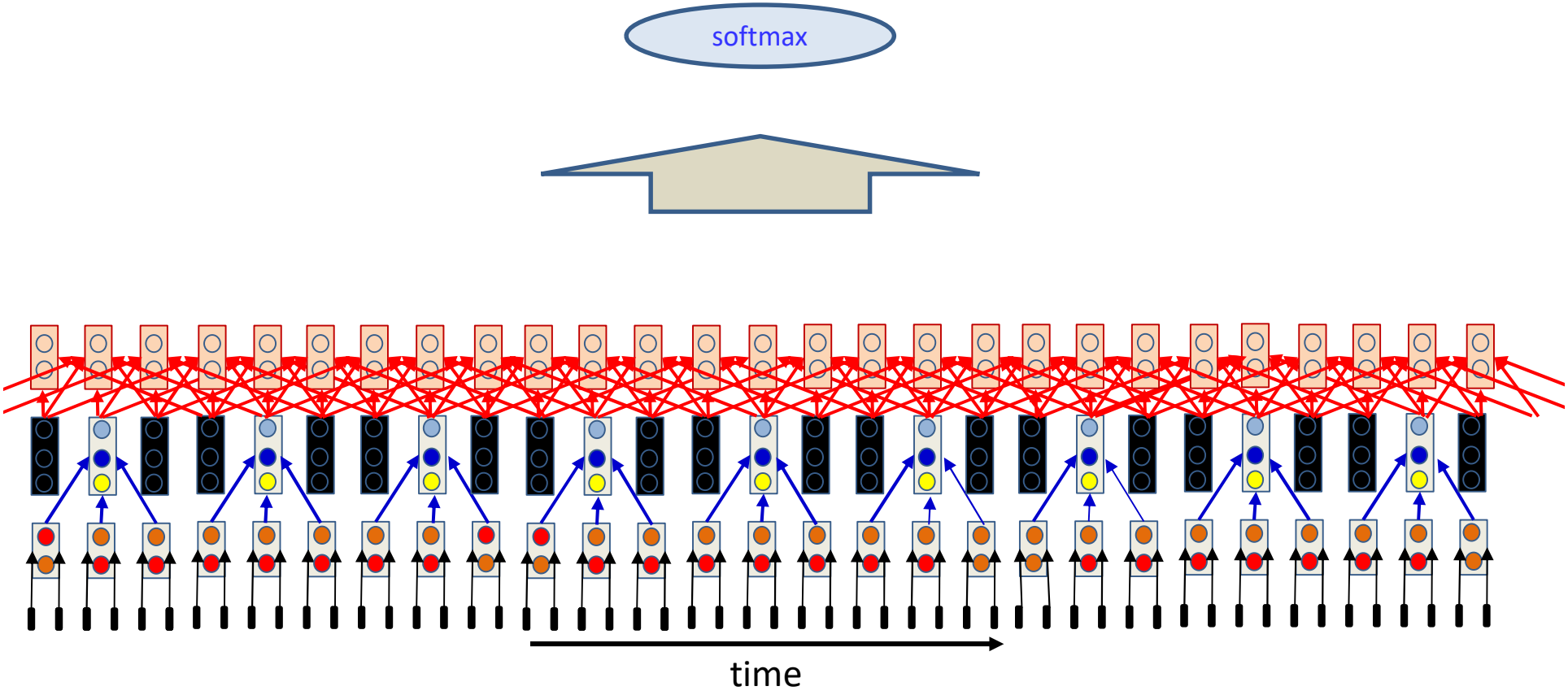


This is exactly analogous to the upsampling performed during backprop when forward convolution uses stride > 1



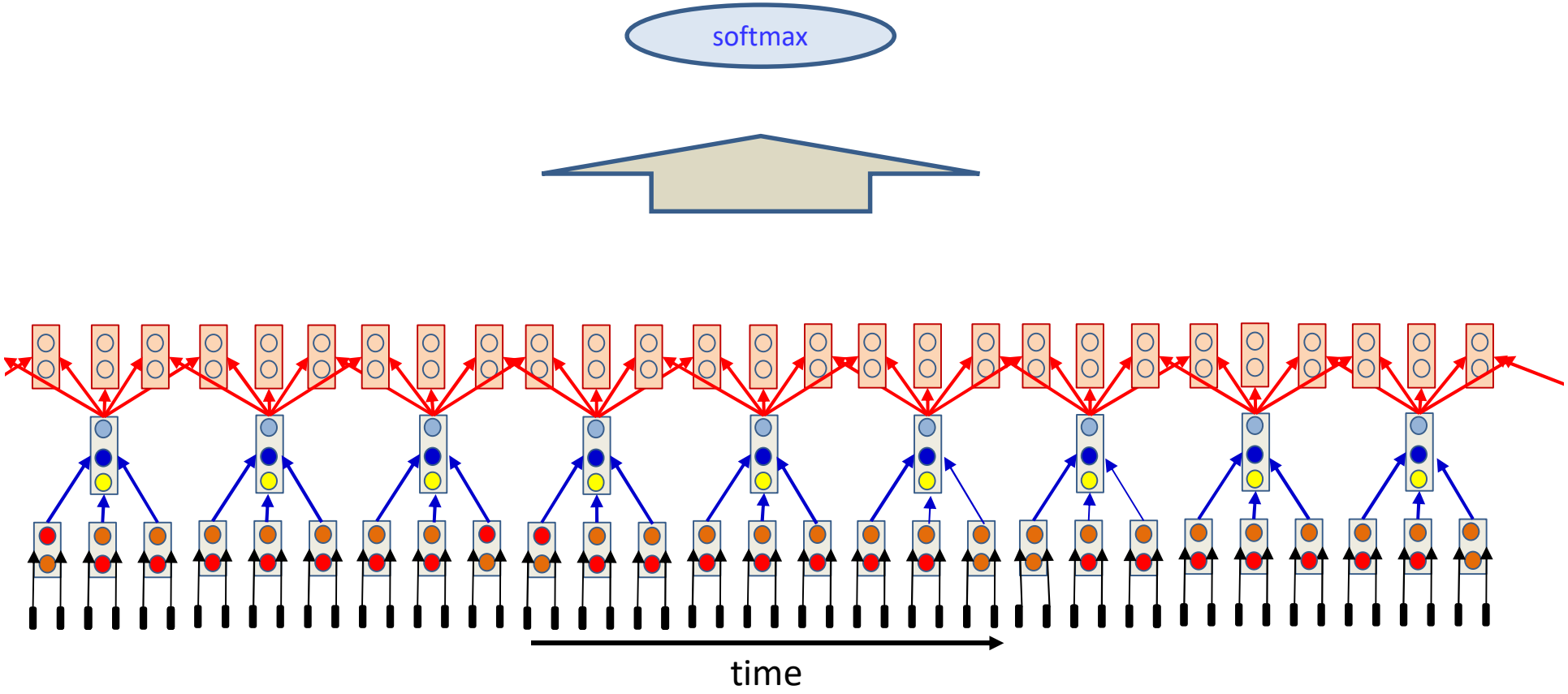
- **Problem:** The values required to compute the intermediate values are missing from the previous layer!
- **Solution:** Synthetically fill in the missing intermediate values of the previous layer
  - With zeros
    - Could also fill them in with linear or spline interpolation of neighbors, but it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

Upsampling 1-D scans



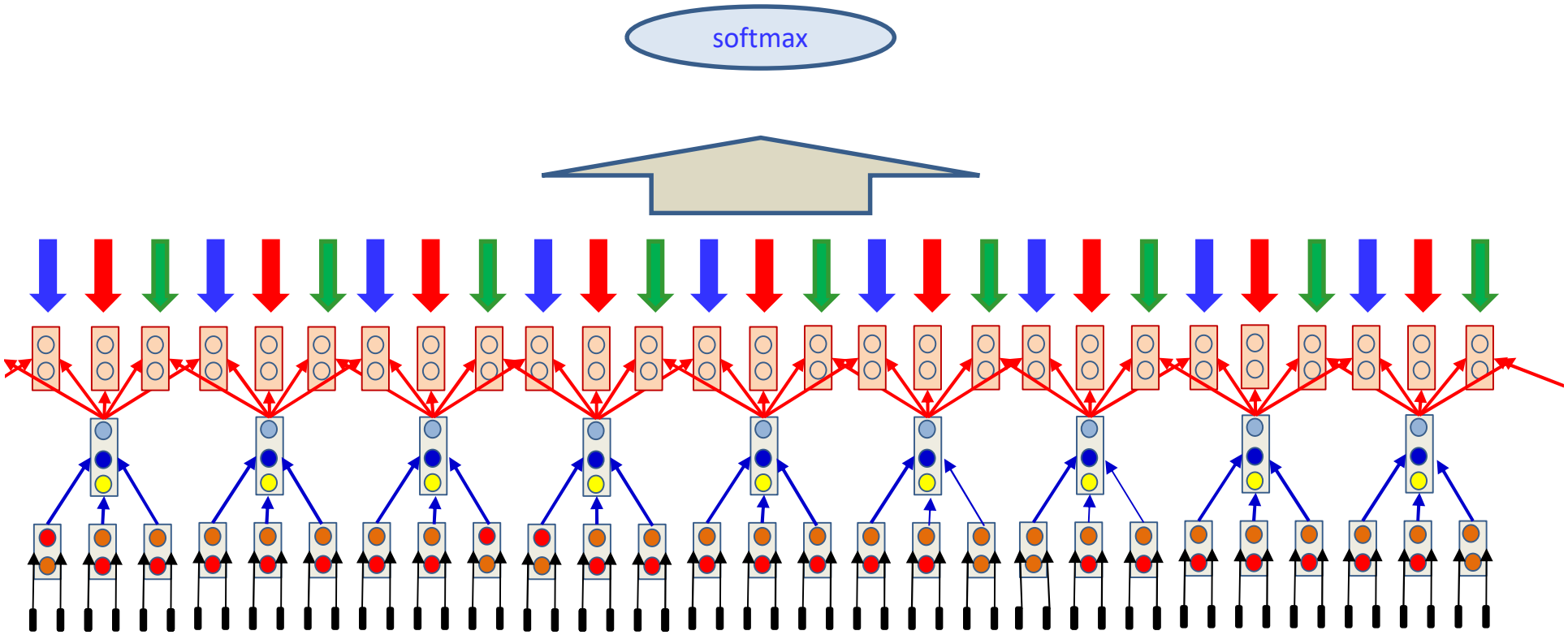
- The 0-valued interpolated inputs do not really provide any input
- They, and their connections can be removed without changing the computation

Upsampling 1-D scans



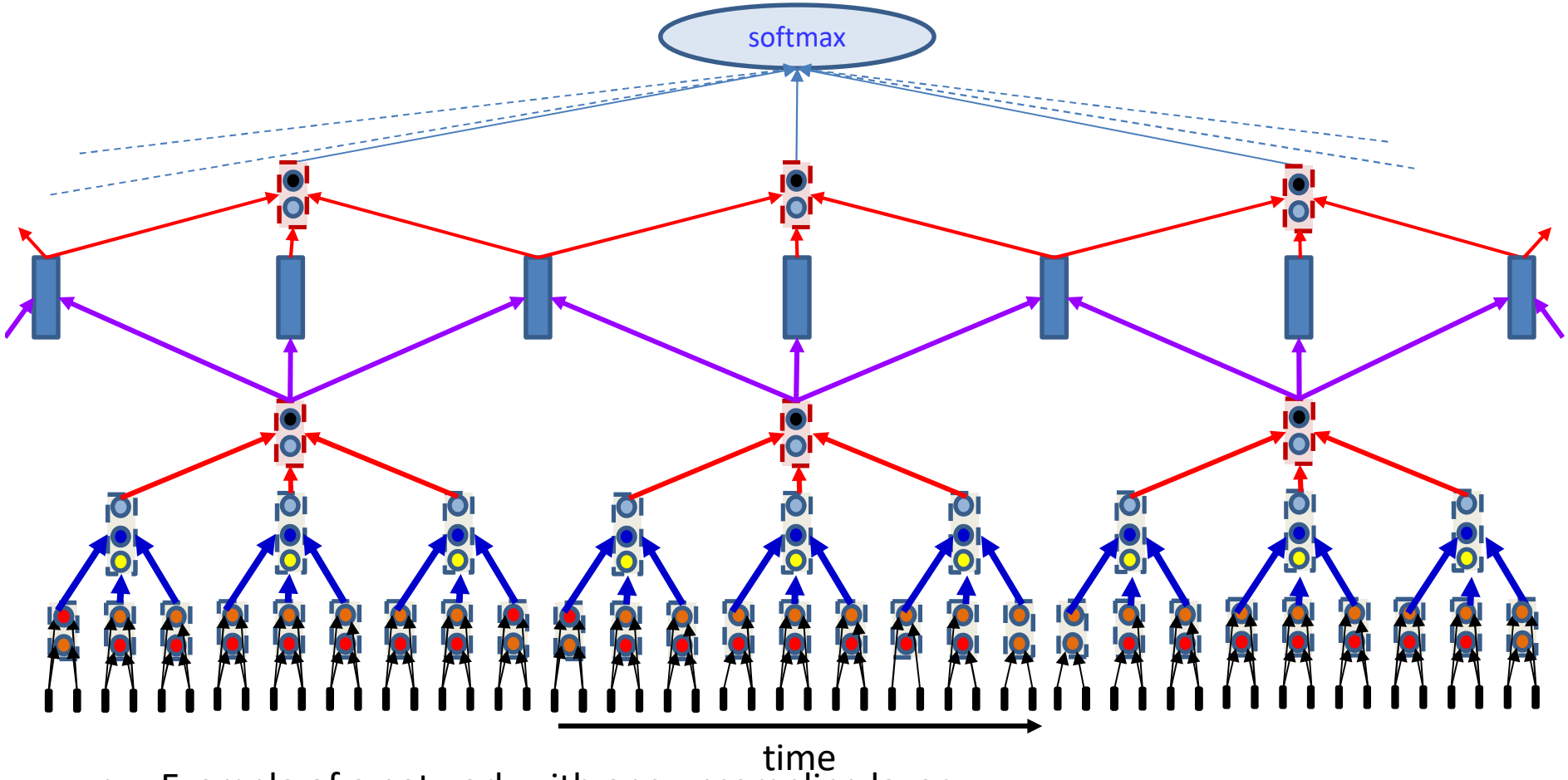
- The 0-valued interpolated inputs do not really provide any input
- They, and their connections can be removed without changing the computation
- *This* is the actual computation performed

Upsampling 1-D scans



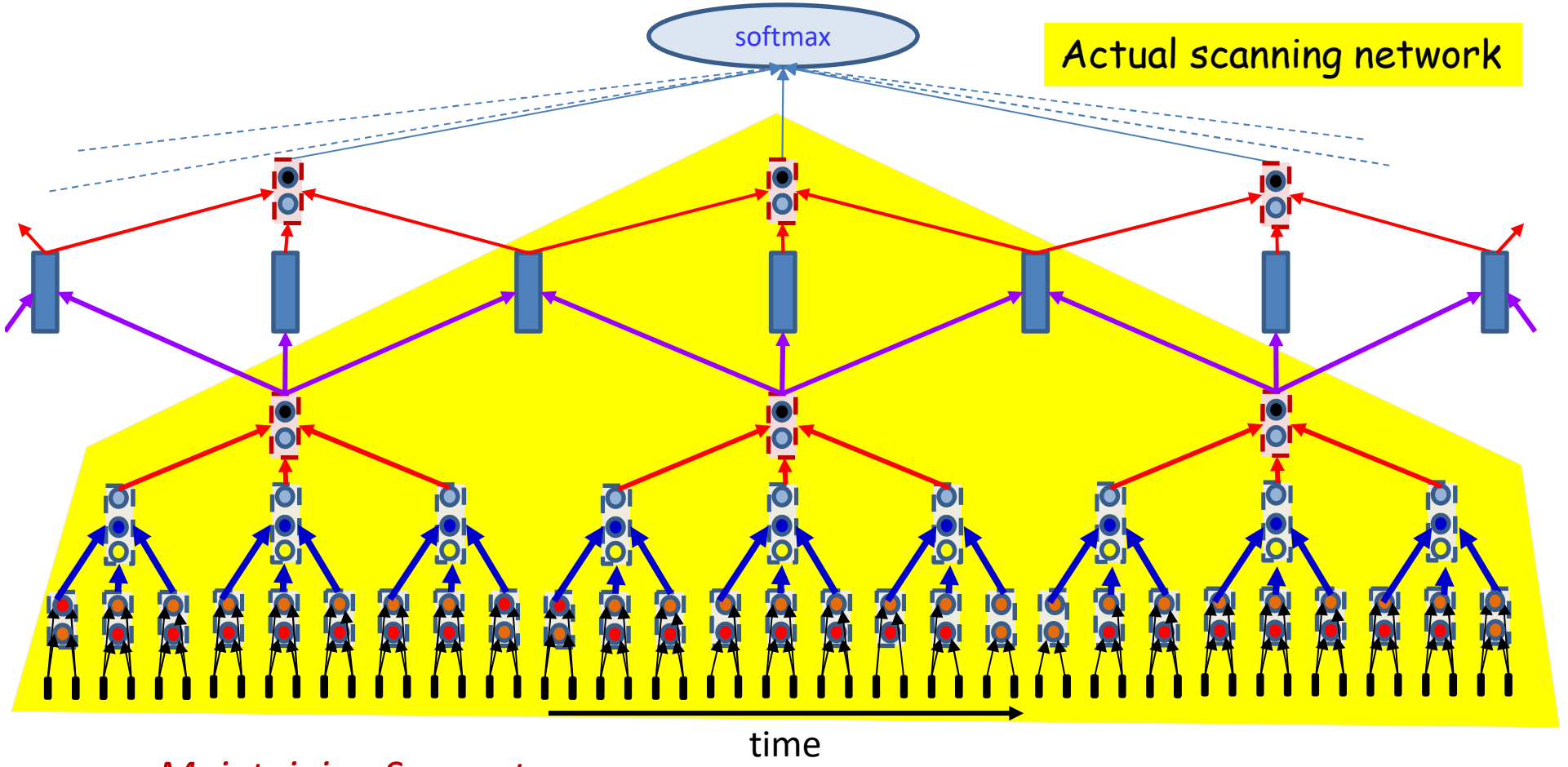
- Key difference from downsampling layers
  - All the “columns” in the regular/downsampling layers are identical
    - Their *incoming* weight patterns are identical
  - The columns in the upsampling layers are *not* identical
    - The *outgoing* weight patterns of the *lower* layer columns are identical

Upsampling as a scanning network



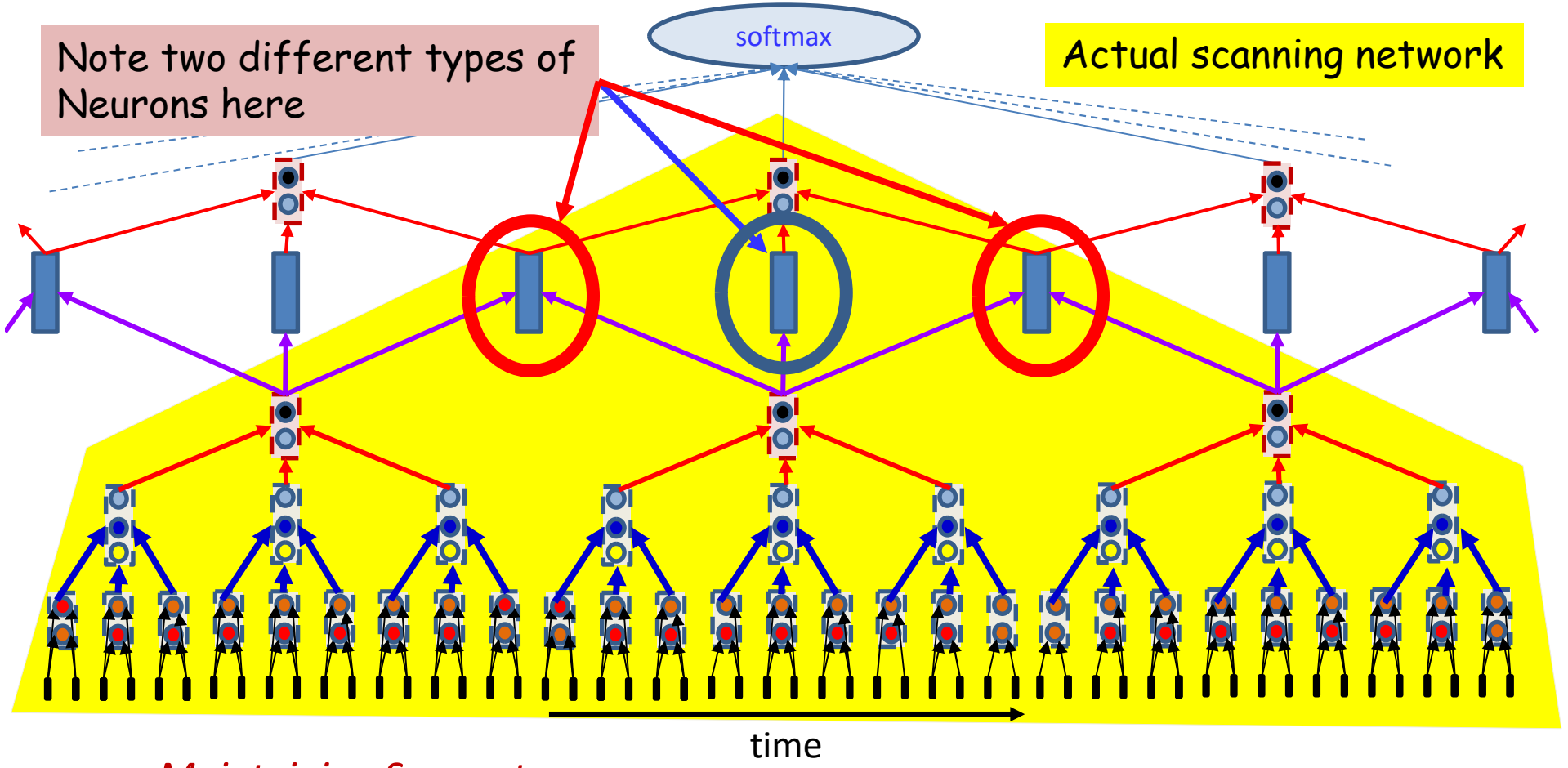
- Example of a network with one upsampling layer
- *Maintaining Symmetry:*
  - Vertical bars in the 4th layer are regularly arranged w.r.t. bars of layer 3
  - The pattern of values of upward weights for each of the three pink (3rd layer) bars is identical

Upsampling as a scanning network



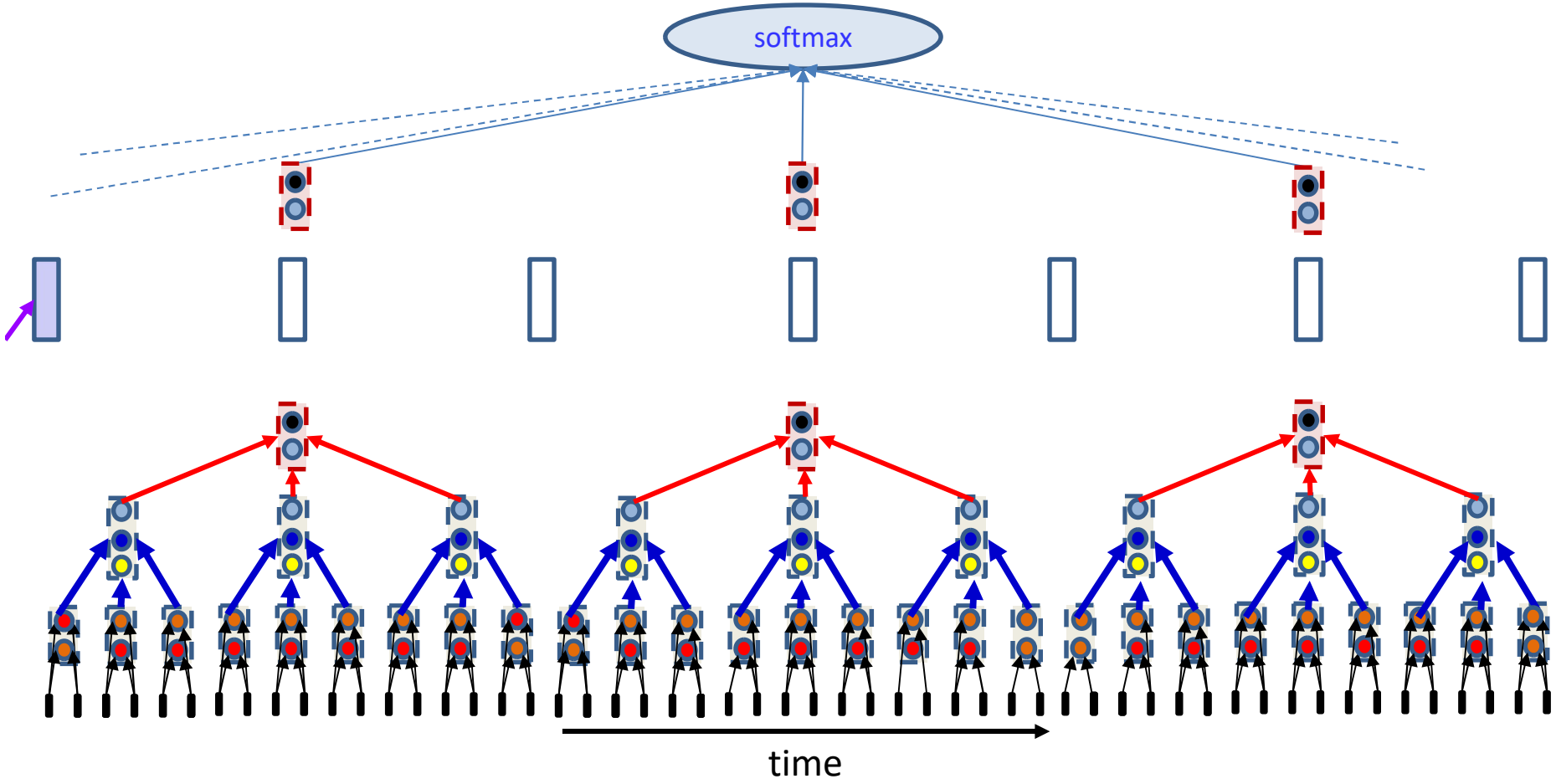
- *Maintaining Symmetry:*
  - Vertical bars in the 4th layer are regularly arranged w.r.t. bars of layer 3
  - The pattern of values of upward weights for each of the three pink (3rd layer) bars is identical

Upsampling as a scanning network



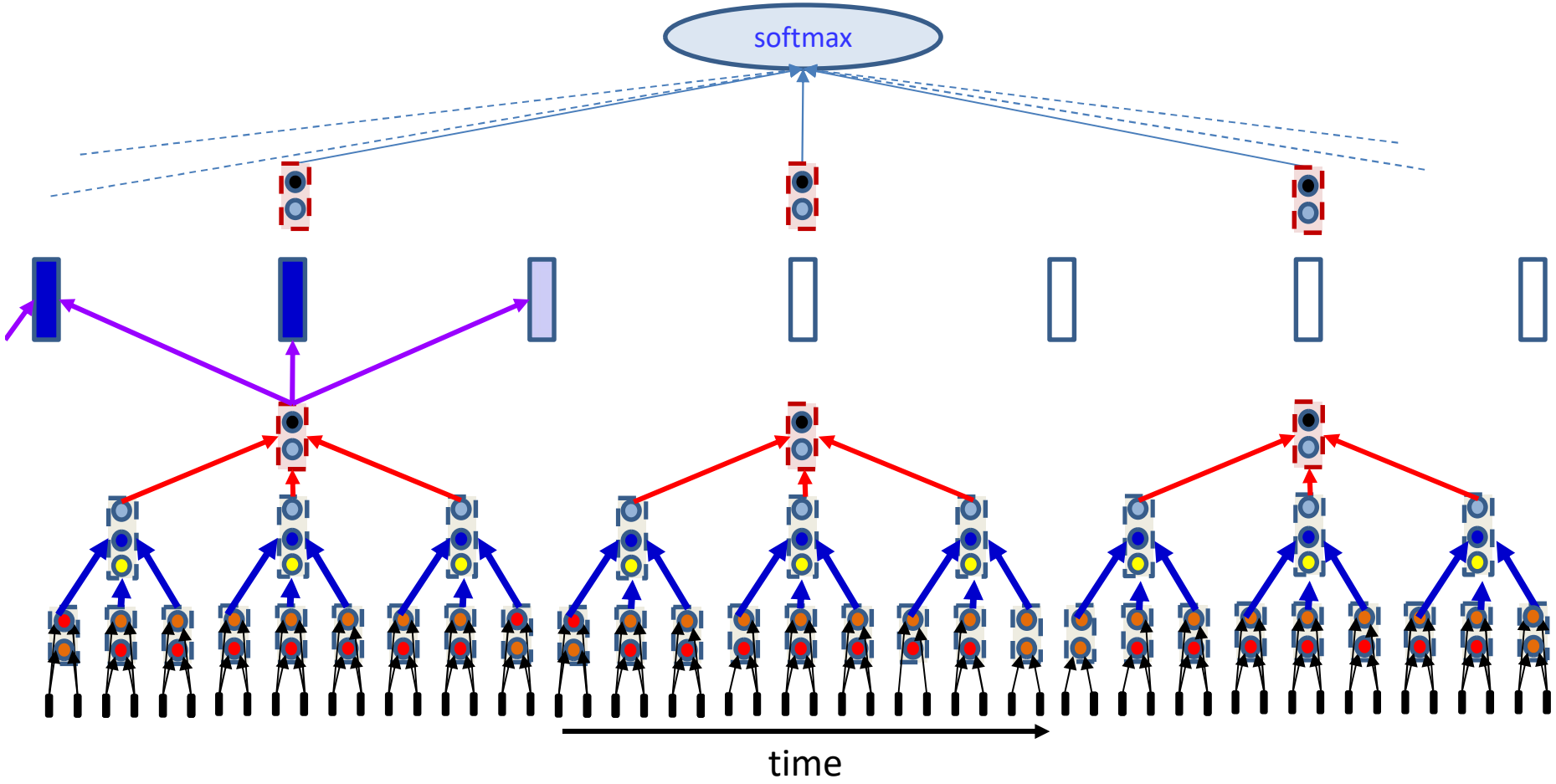
- *Maintaining Symmetry:*
  - Vertical bars in the 4th layer are regularly arranged w.r.t. bars of layer 3
  - The pattern of values of upward weights for each of the three pink (3rd layer) bars is identical

Scanning with increased-res layer



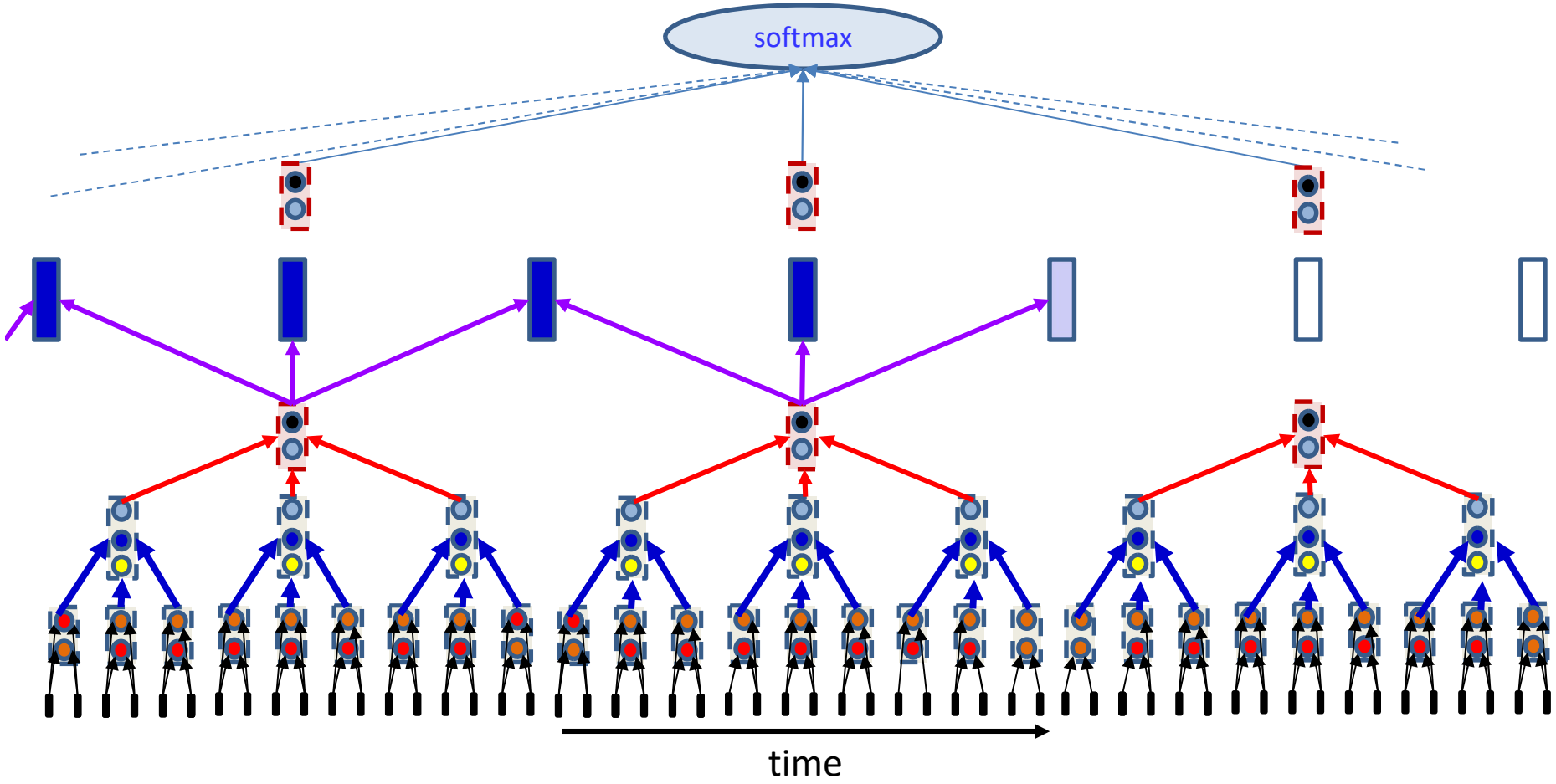
- Flow of info from bottom to top when implemented as a left-to-right scan
  - Note: Arrangement of vertical bars is predetermined by architecture

With layer of increased size



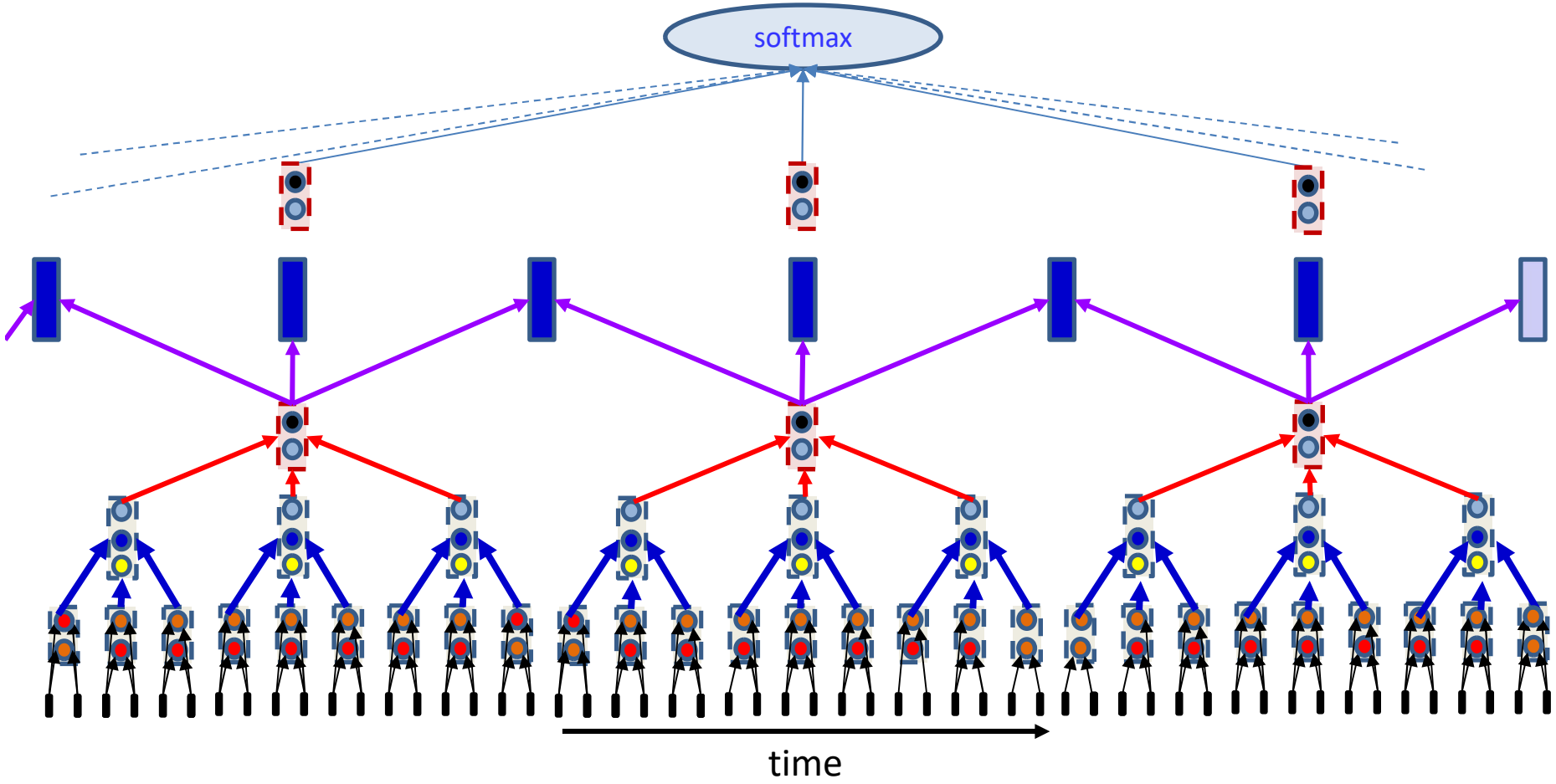
- Flow of info from bottom to top when implemented as a left-to-right scan
  - Note: Arrangement of vertical bars is predetermined by architecture

With layer of increased size



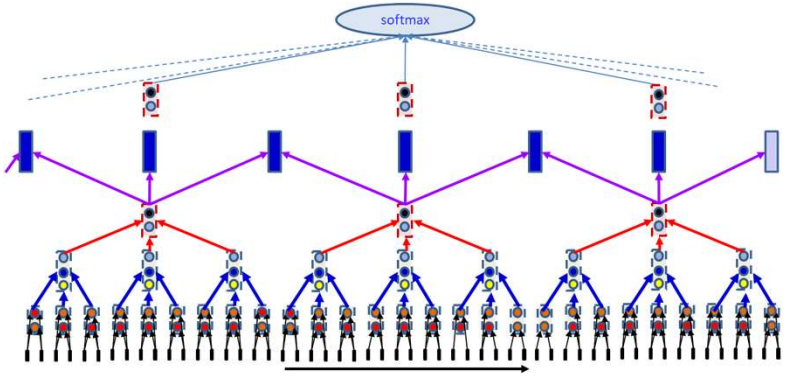
- Flow of info from bottom to top when implemented as a left-to-right scan
  - Note: Arrangement of vertical bars is predetermined by architecture

With layer of increased size



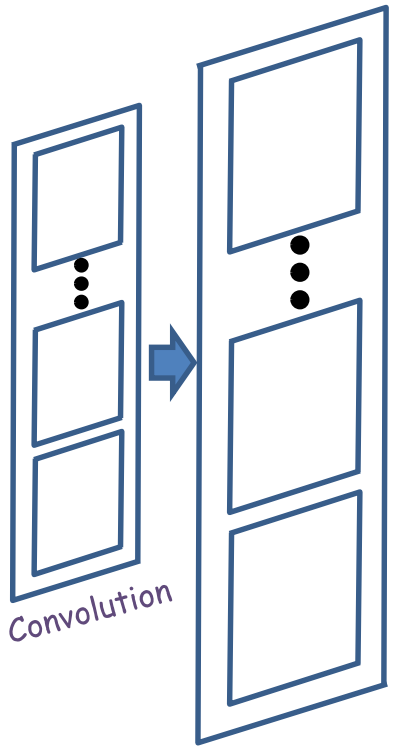
- Flow of info from bottom to top when implemented as a left-to-right scan
  - Note: Arrangement of vertical bars is predetermined by architecture

Transposed convolution



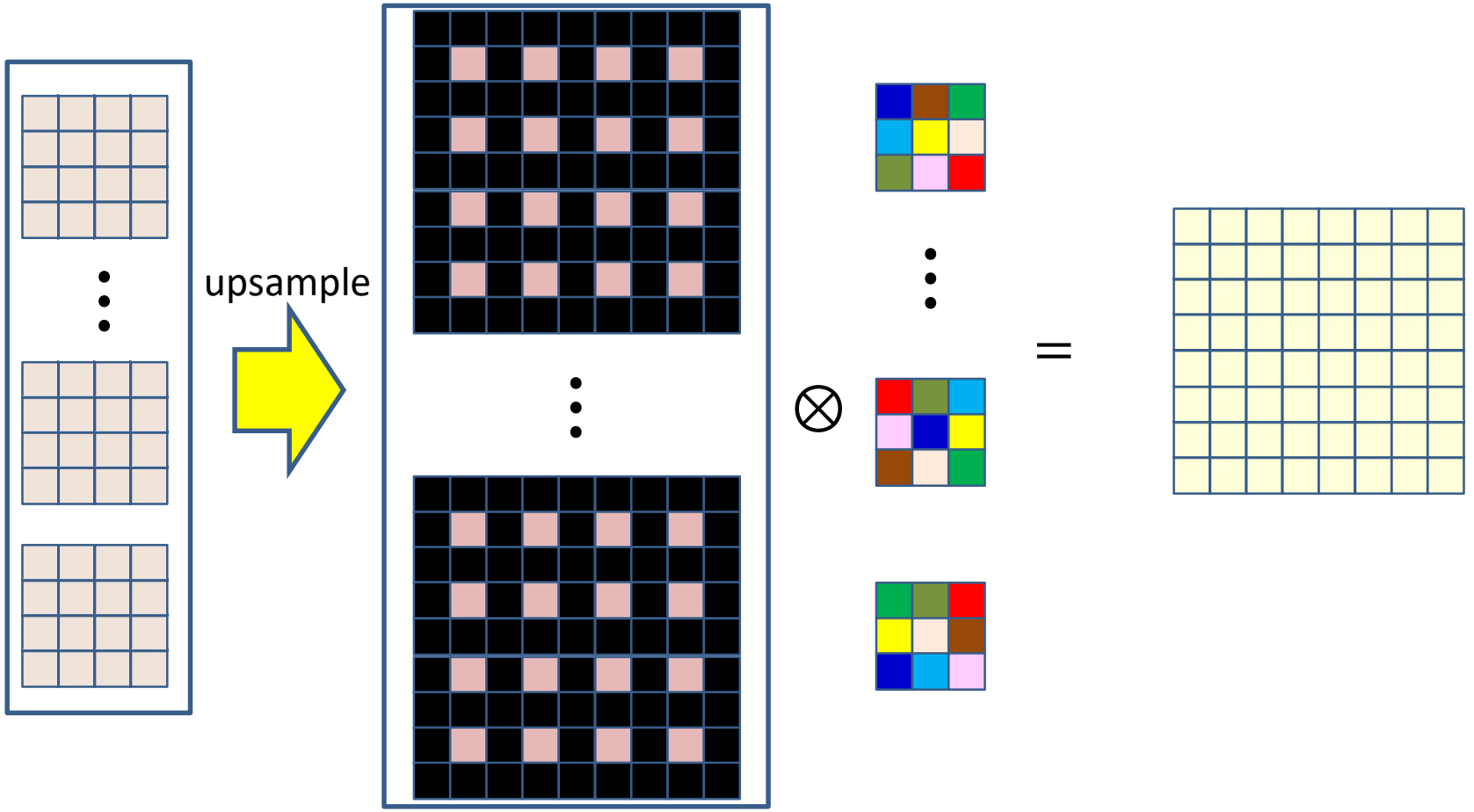
- Signal propagation rules are transposed for expanding layers
- In regular convolution, the affine value Z for a layer "pulls" Y values from the lower layer
  - In vector form
$$Z_l = W_l Y_{l-1}$$
  - The ith neuron:
$$z_l(i) = W_l(i,:)Y_{l-1}$$
  - Invokes the ith row of W_l
- In an upsampling layer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

In 2-D



- Similar computation

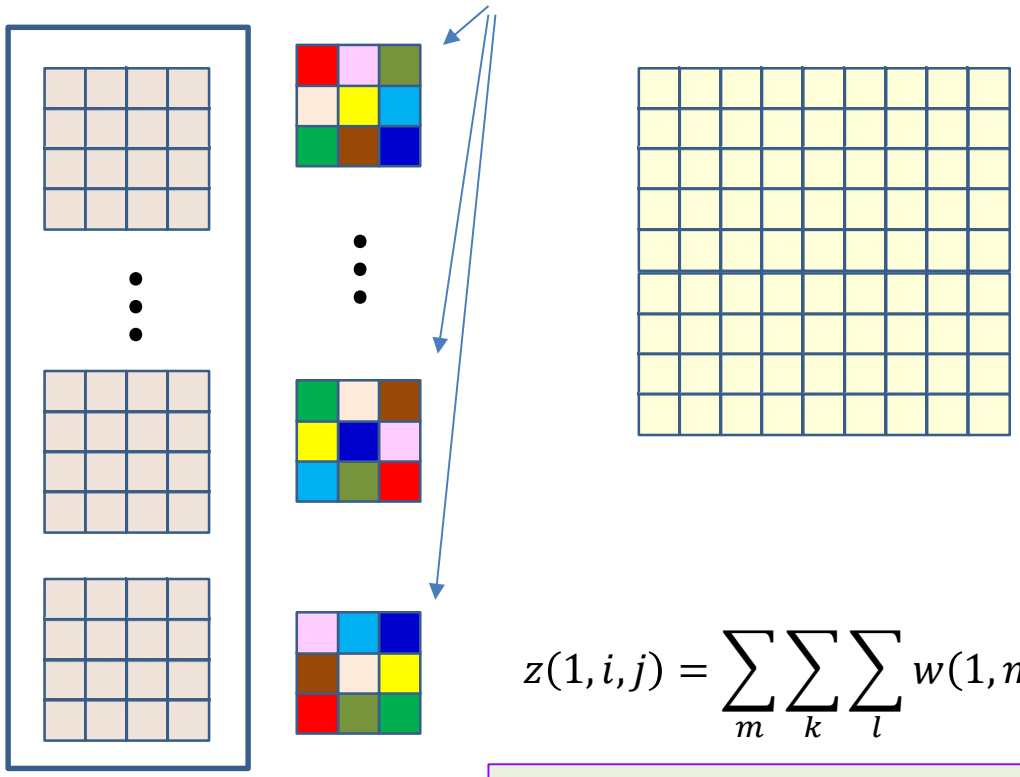
2D expanding convolution



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

2D expanding convolution in practice

These filters are "transposed" (flipped across the diagonal) w.r.t. the scanning filters for the upsampled maps in the previous slide

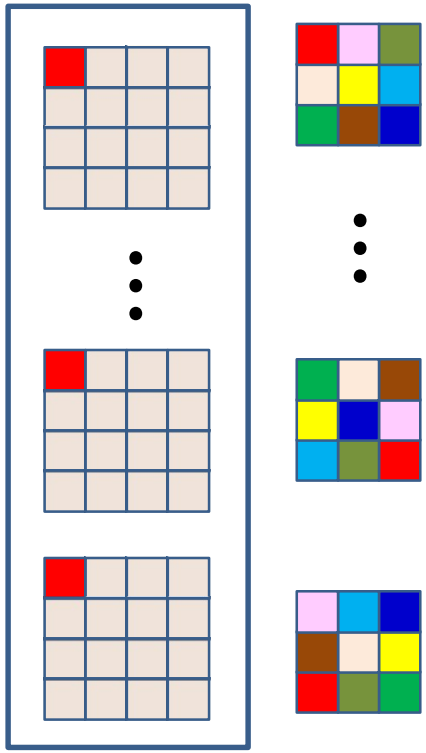


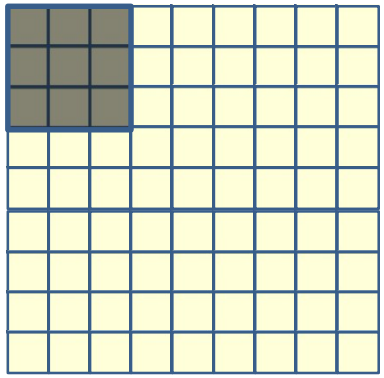
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb)I(m,k,l)$$

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

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

2D expanding convolution in practice



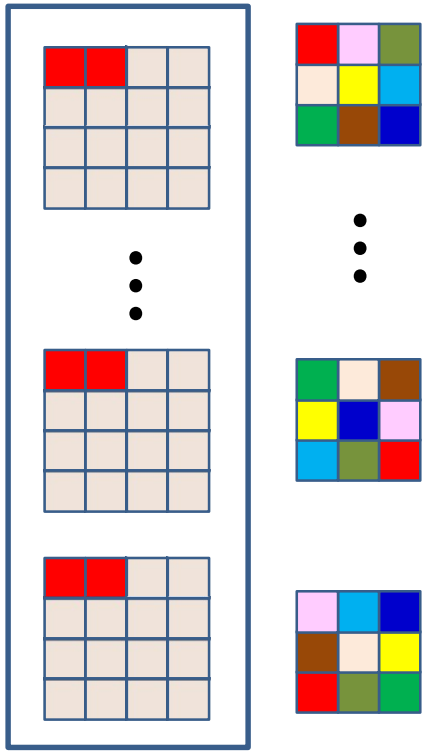


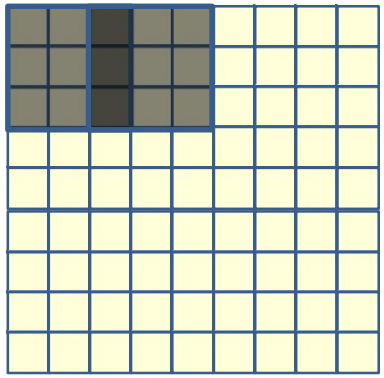
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb) I(m,k,l)$$

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

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

2D expanding convolution in practice



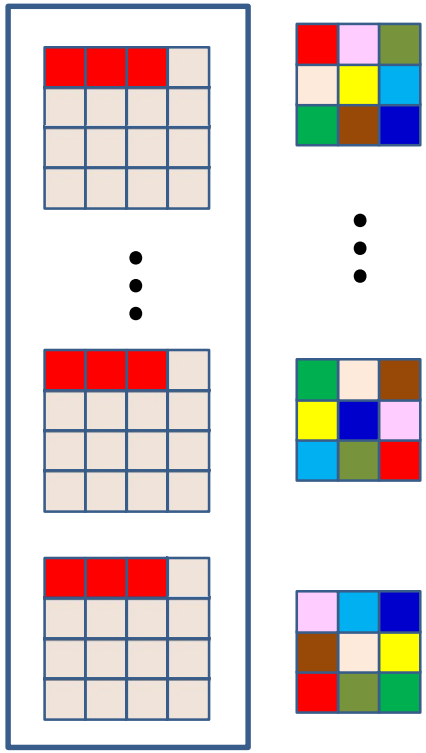


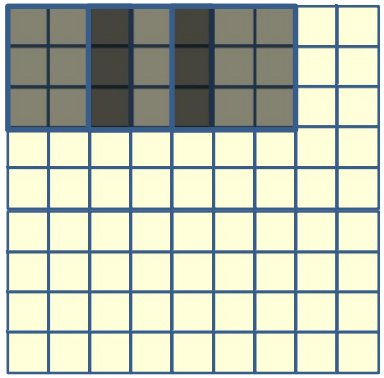
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb)I(m,k,l)$$

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

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

2D expanding convolution in practice



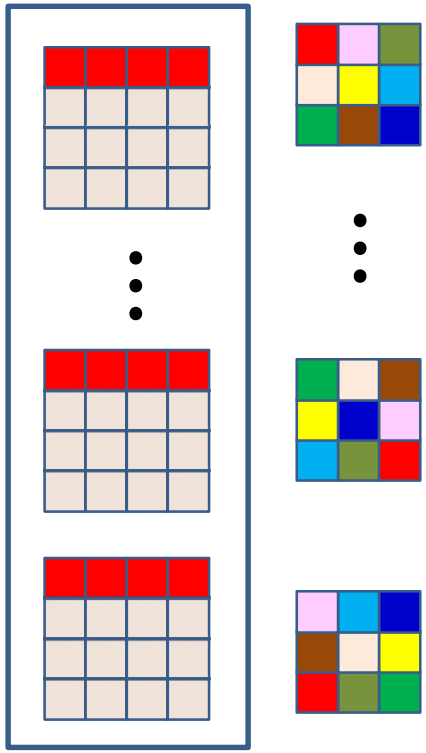


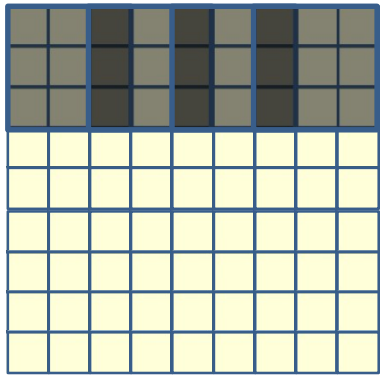
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb) I(m,k,l)$$

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

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

2D expanding convolution in practice



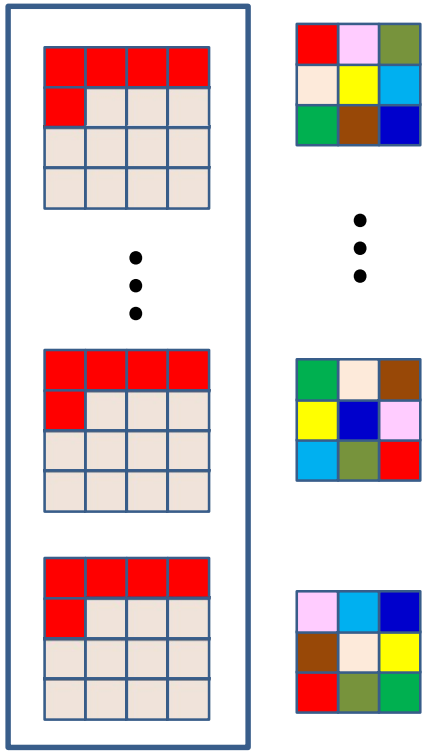


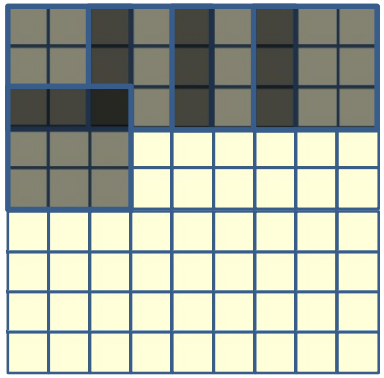
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb)I(m,k,l)$$

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

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

2D expanding convolution in practice



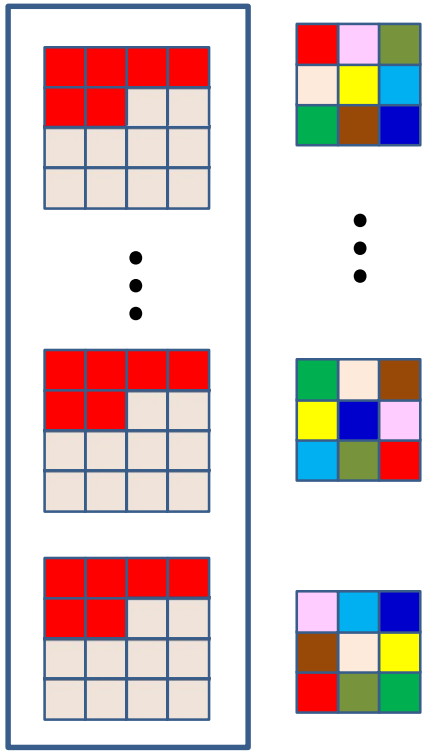


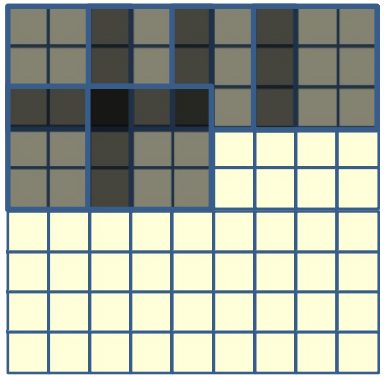
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb) I(m,k,l)$$

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

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

2D expanding convolution in practice



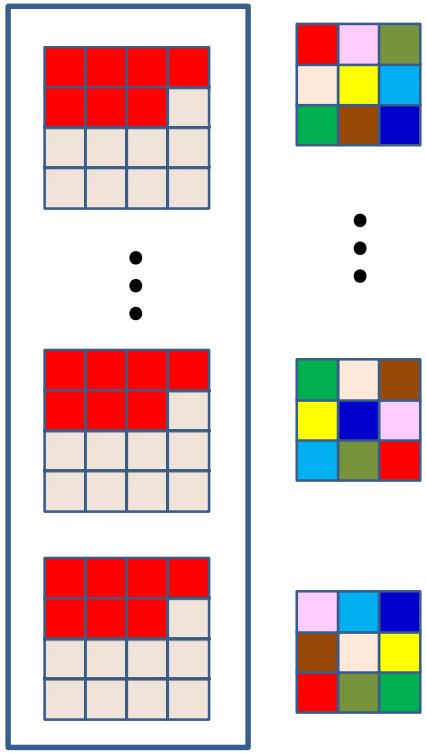


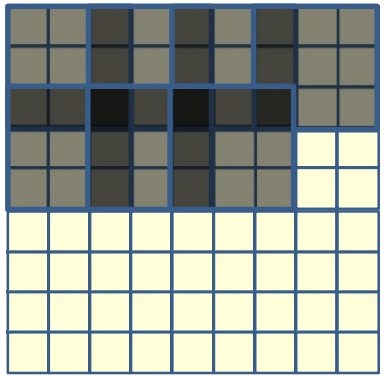
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb)I(m,k,l)$$

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

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

2D expanding convolution in practice



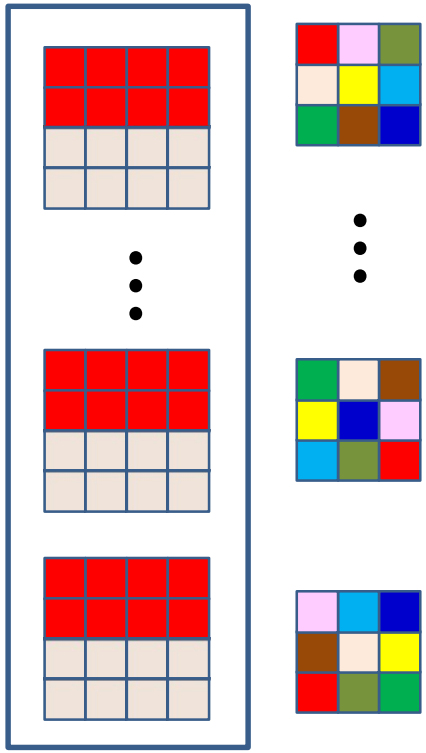


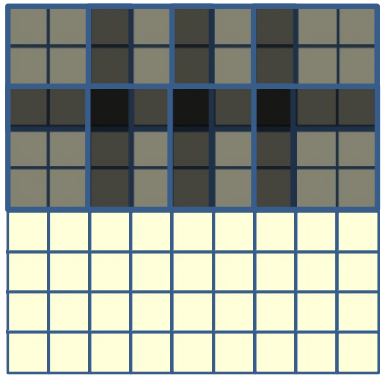
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb) I(m,k,l)$$

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

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

2D expanding convolution in practice



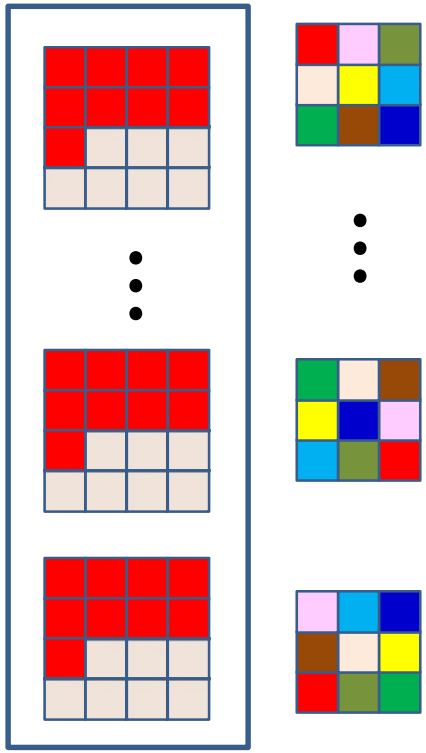


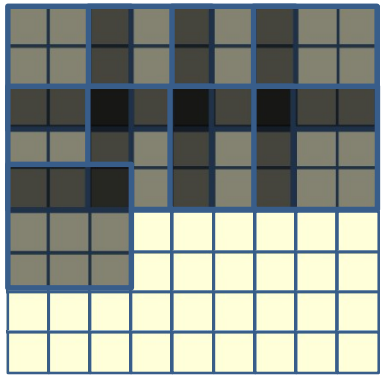
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb)I(m,k,l)$$

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

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

2D expanding convolution in practice



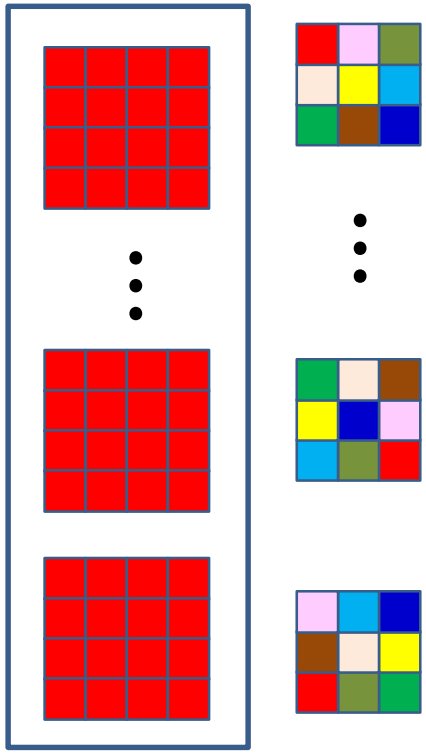


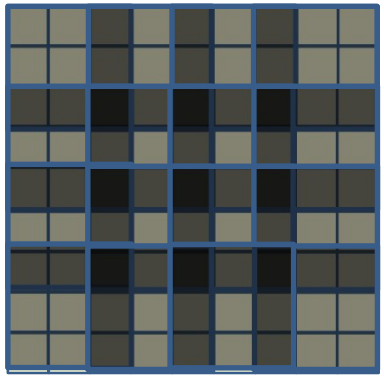
$$z(1,i,j) = \sum_m \sum_k \sum_l w(1,m,i-kb,j-lb) I(m,k,l)$$

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

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

2D expanding convolution in practice





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

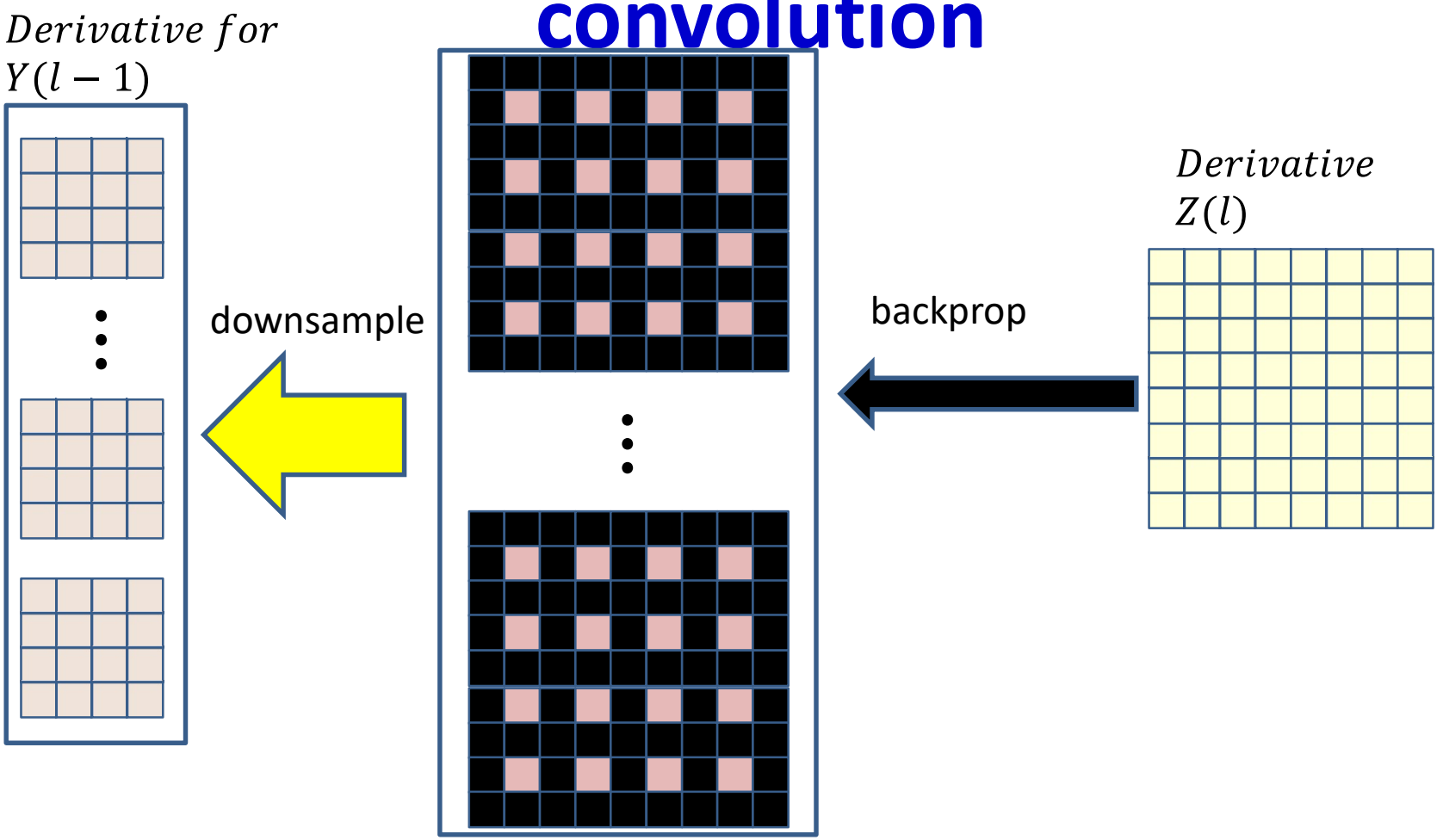
b is the "stride"
(scaling factor between the sizes of Z and Y)

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

CNN: Expanding convolution layer l

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

Backprop through expanding convolution



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

CNN: Expanding convolution layer l

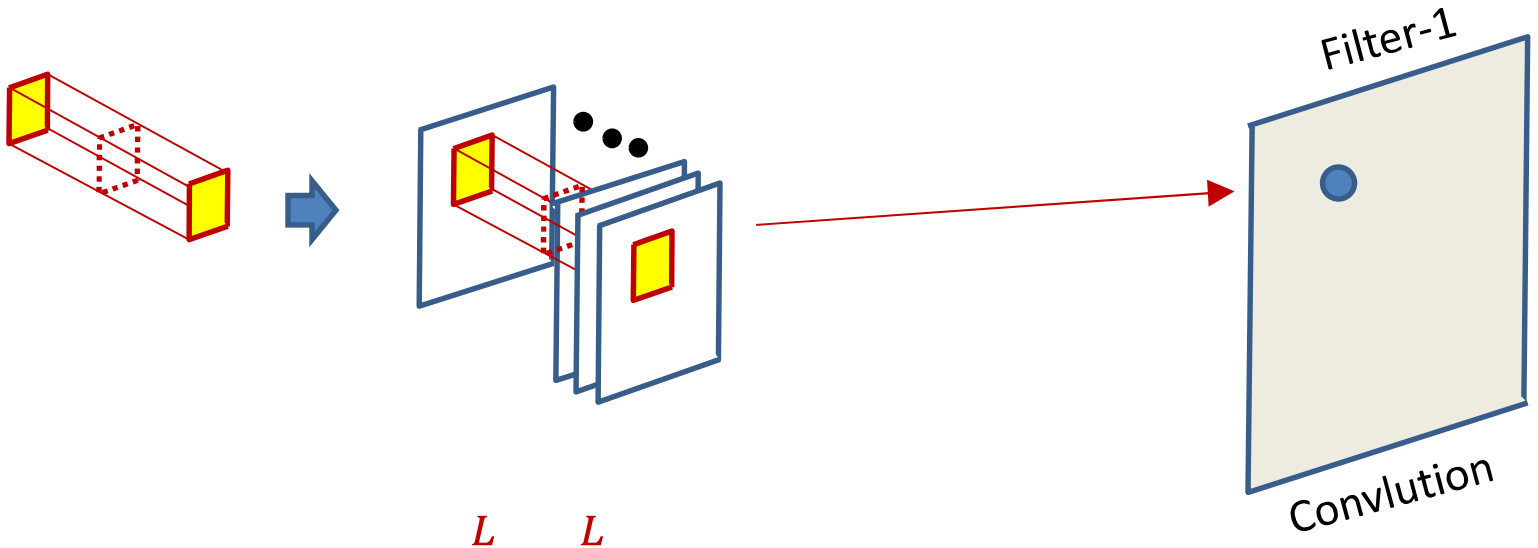
```
Z(l) = zeros(Dl x ((W-1)b+K_l) x ((H-1)b+K_l)) # b = stride
for j = 1:D_l
  for x = 1:W
    for y = 1:H
      for i = 1:D_l-1
        for x' = 1:K_l
          for y' = 1:K_l
            z(l,j,(x-1)b+x',(y-1)b+y') +=
                        w(l,j,i,x',y')y(l-1,i,x,y)
```

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

Invariance

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

Shift-invariance – a different perspective

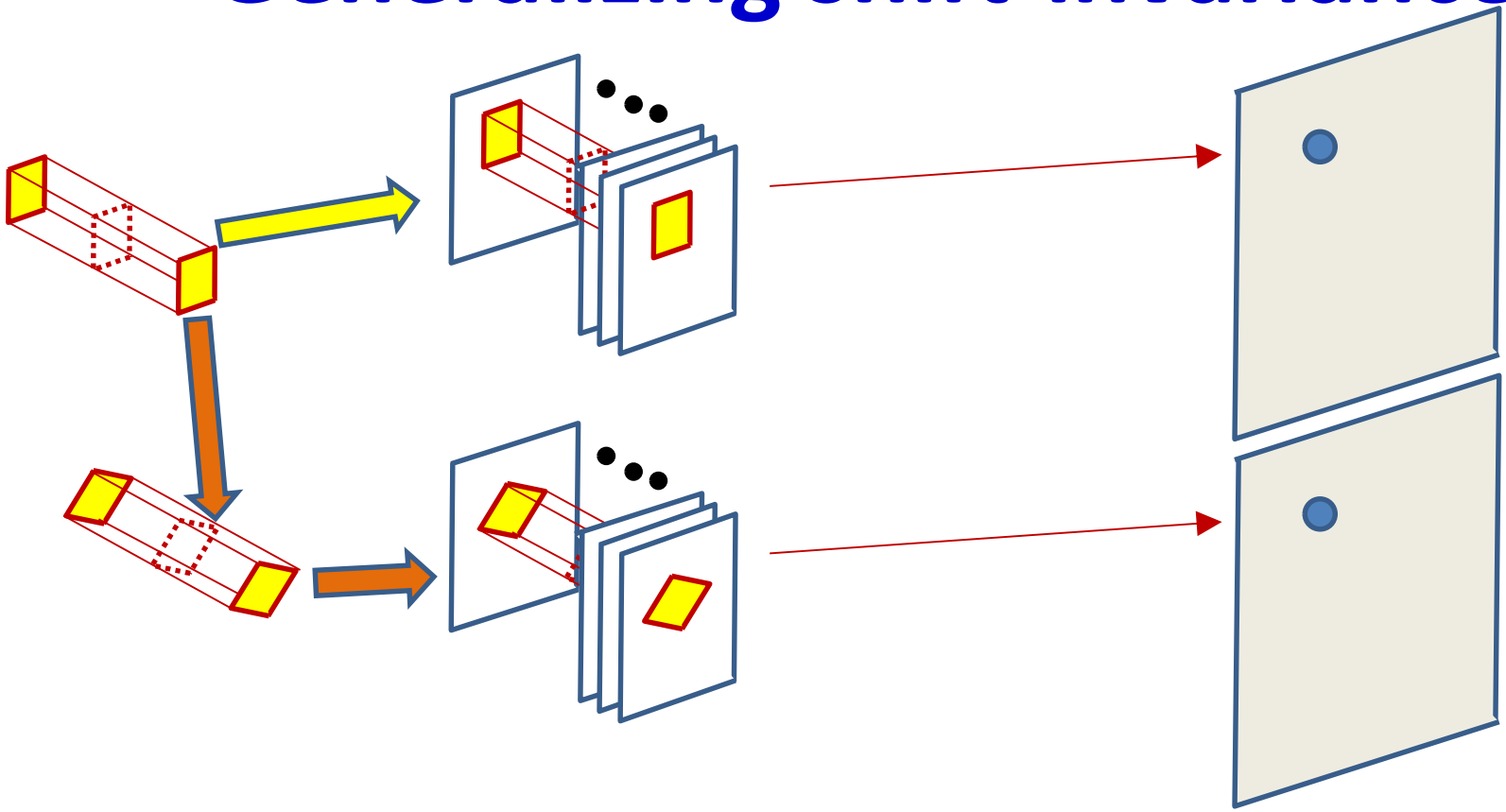


$$z(l,s,i,j) = \sum_{p} \sum_{k=1}^{L} \sum_{m=1}^{L} w(l,s,p,k,m) Y(l-1,p,i+k,j+m)$$

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

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

Generalizing shift-invariance



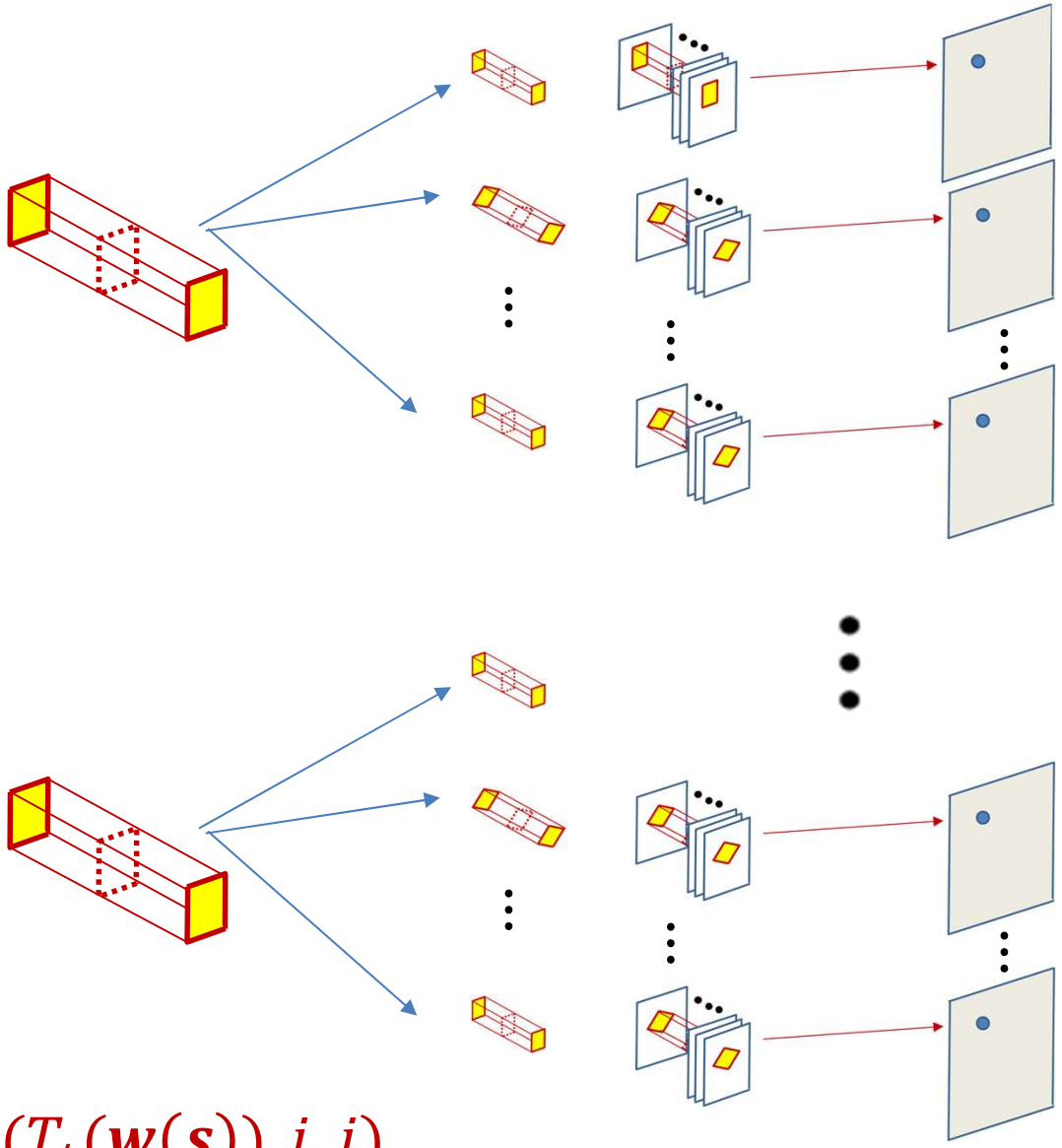
$$z_{regular}(s,i,j) = \boldsymbol{Y}.shift(\boldsymbol{w}(\boldsymbol{s}),i,j)$$

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

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

Transform invariance

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



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

Regular CNN : single layer l

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

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

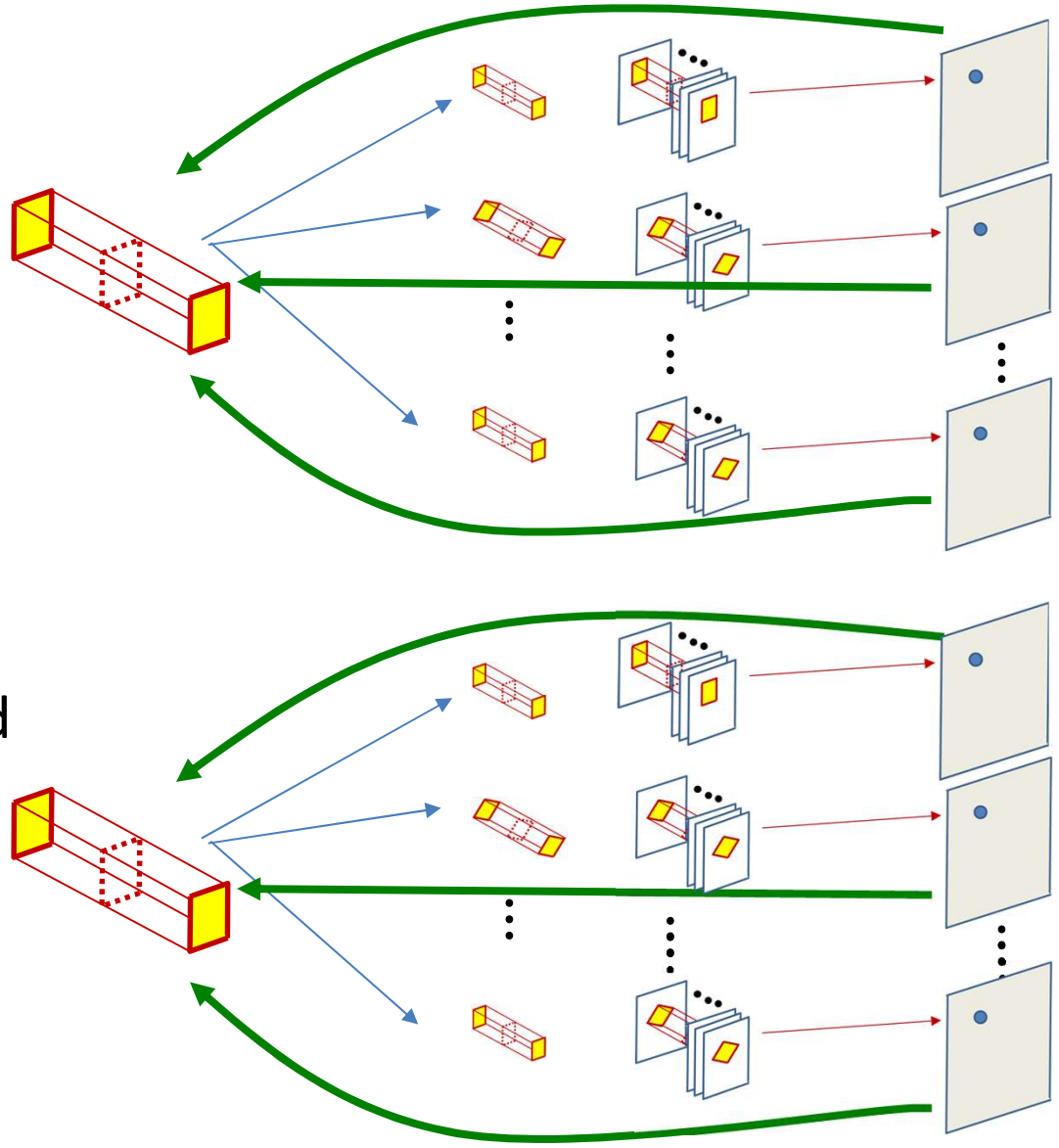
Transform invariance

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

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

BP with transform invariance

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



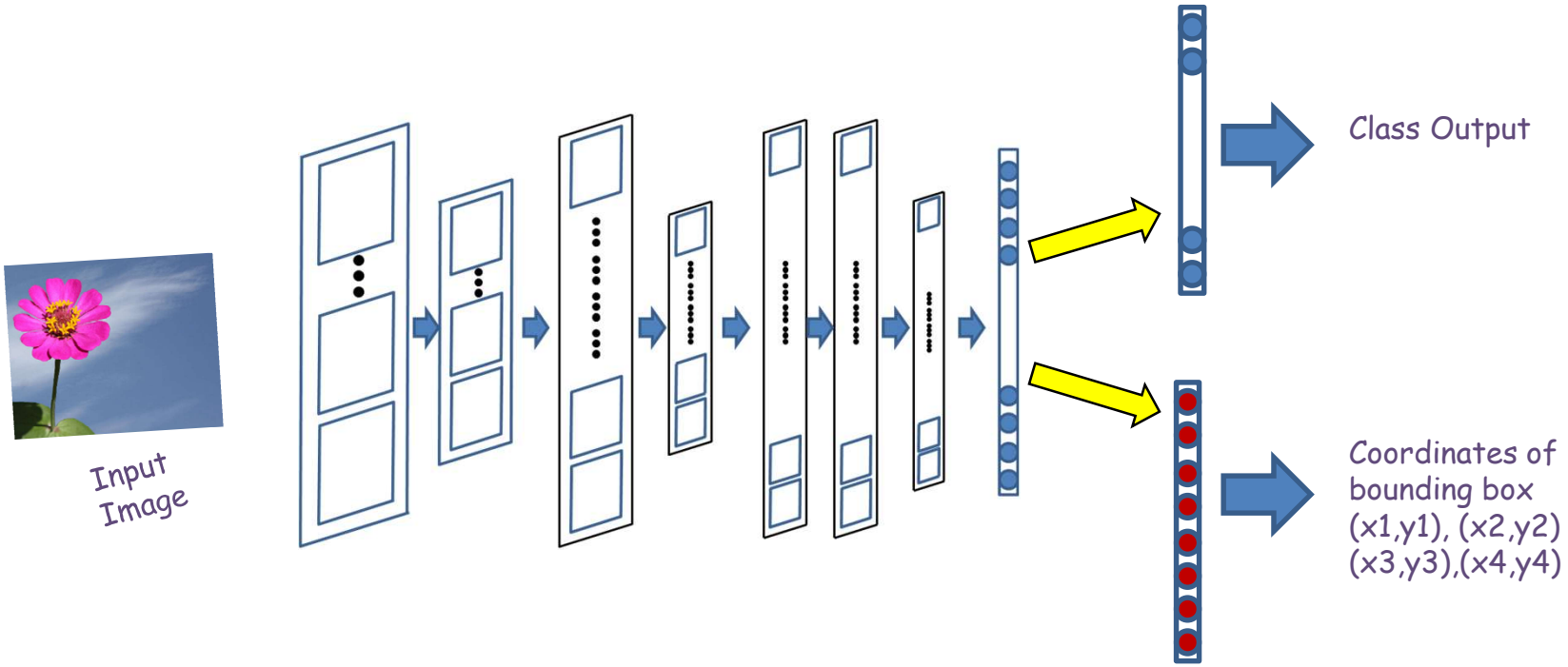
Story so far

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

But what about the exact location?

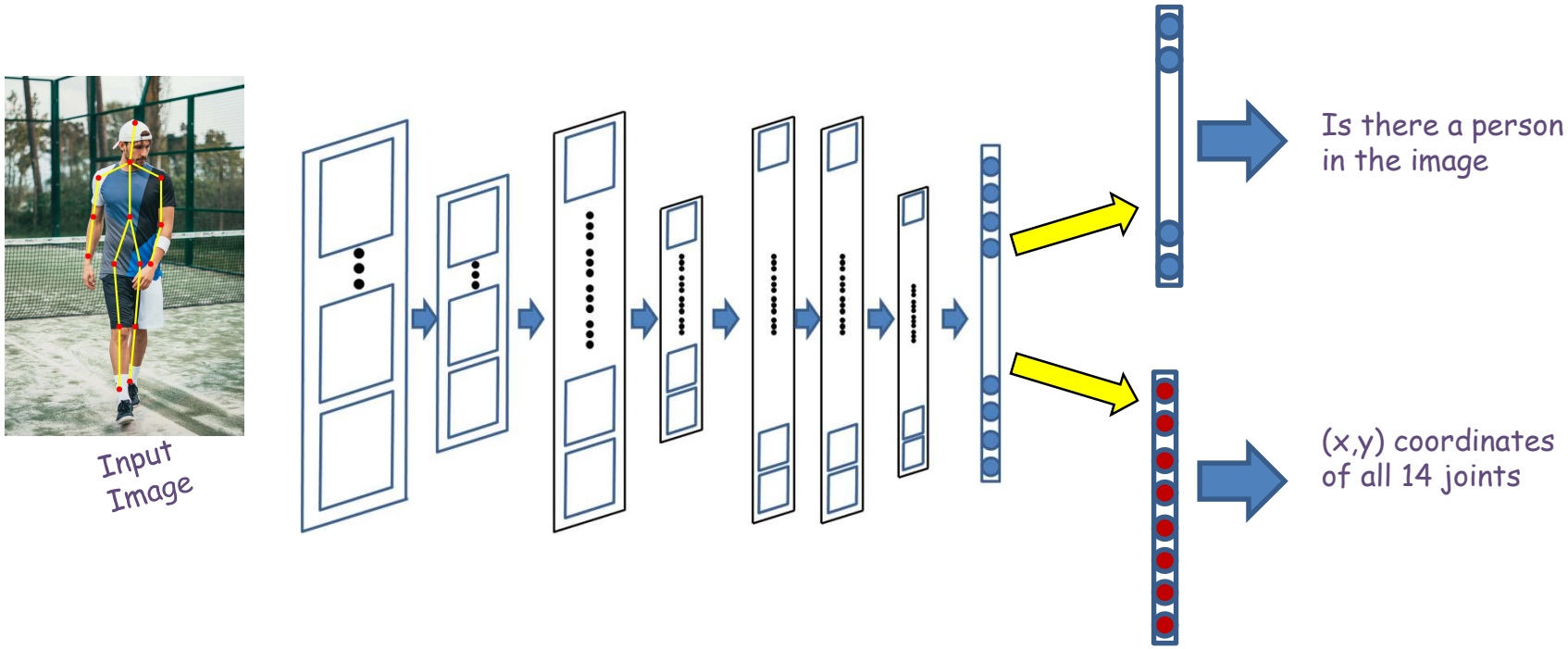
- We began with the desire to identify the picture as containing a flower, regardless of the position of the flower
  - Or more generally the class of object in the picture
- But can we detect the *position* of the main object?

Finding Bounding Boxes



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

Pose estimation

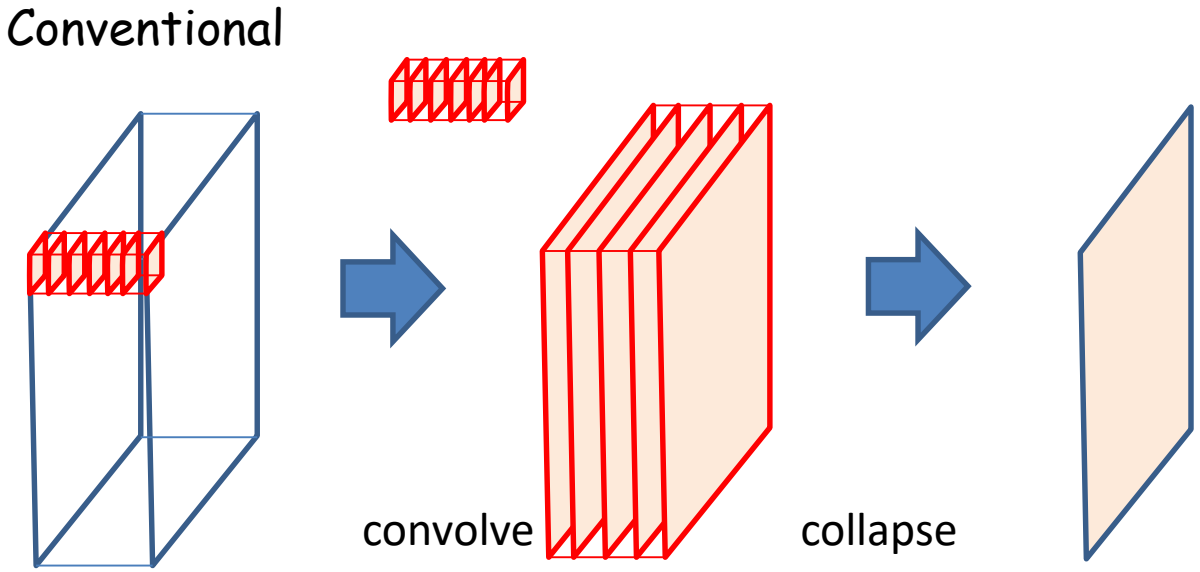


- Can use the same mechanism to predict the joints of a stick model
  - For pose estimation

Model variations

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

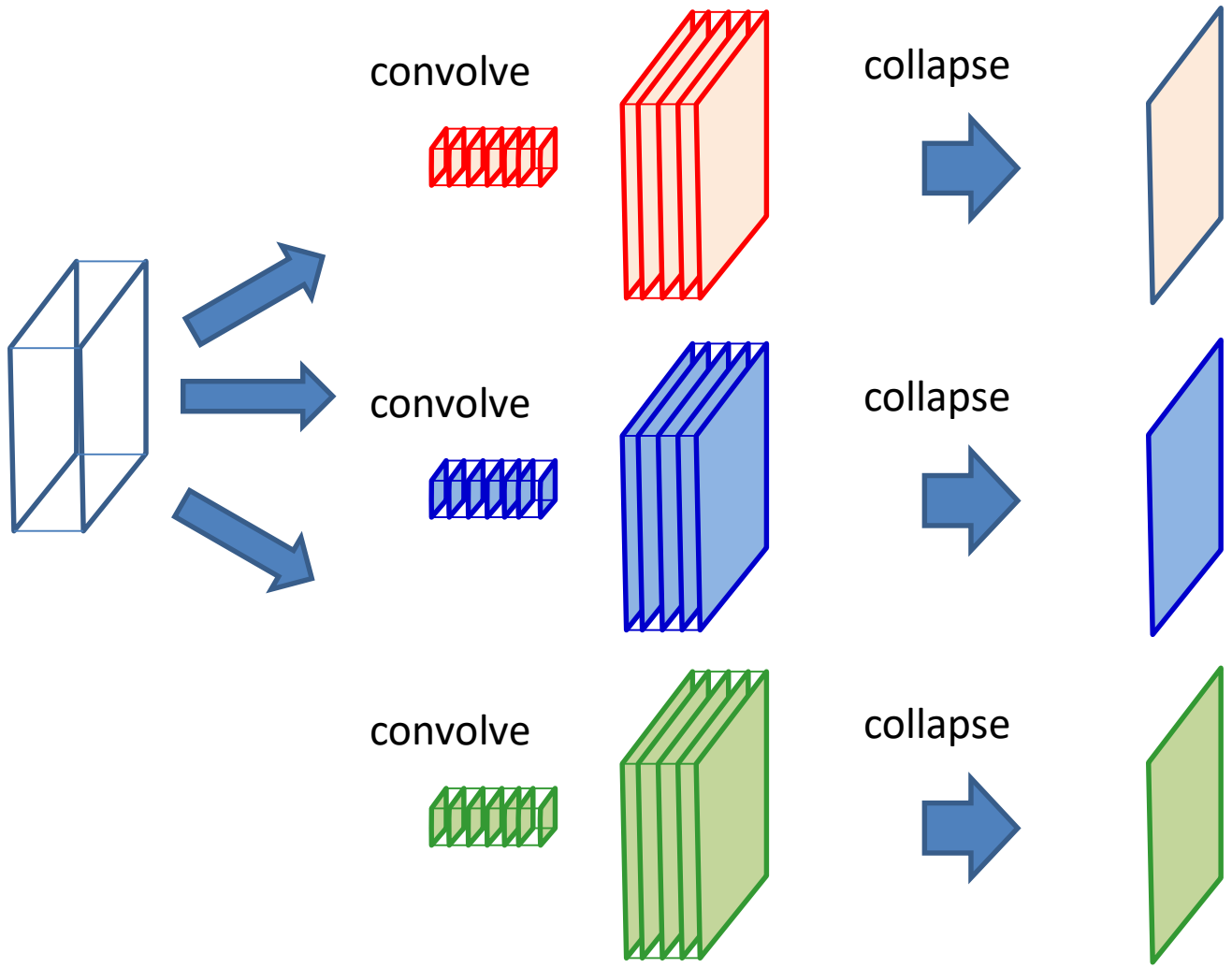
Conventional convolutions



- Alternate view of conventional convolution:

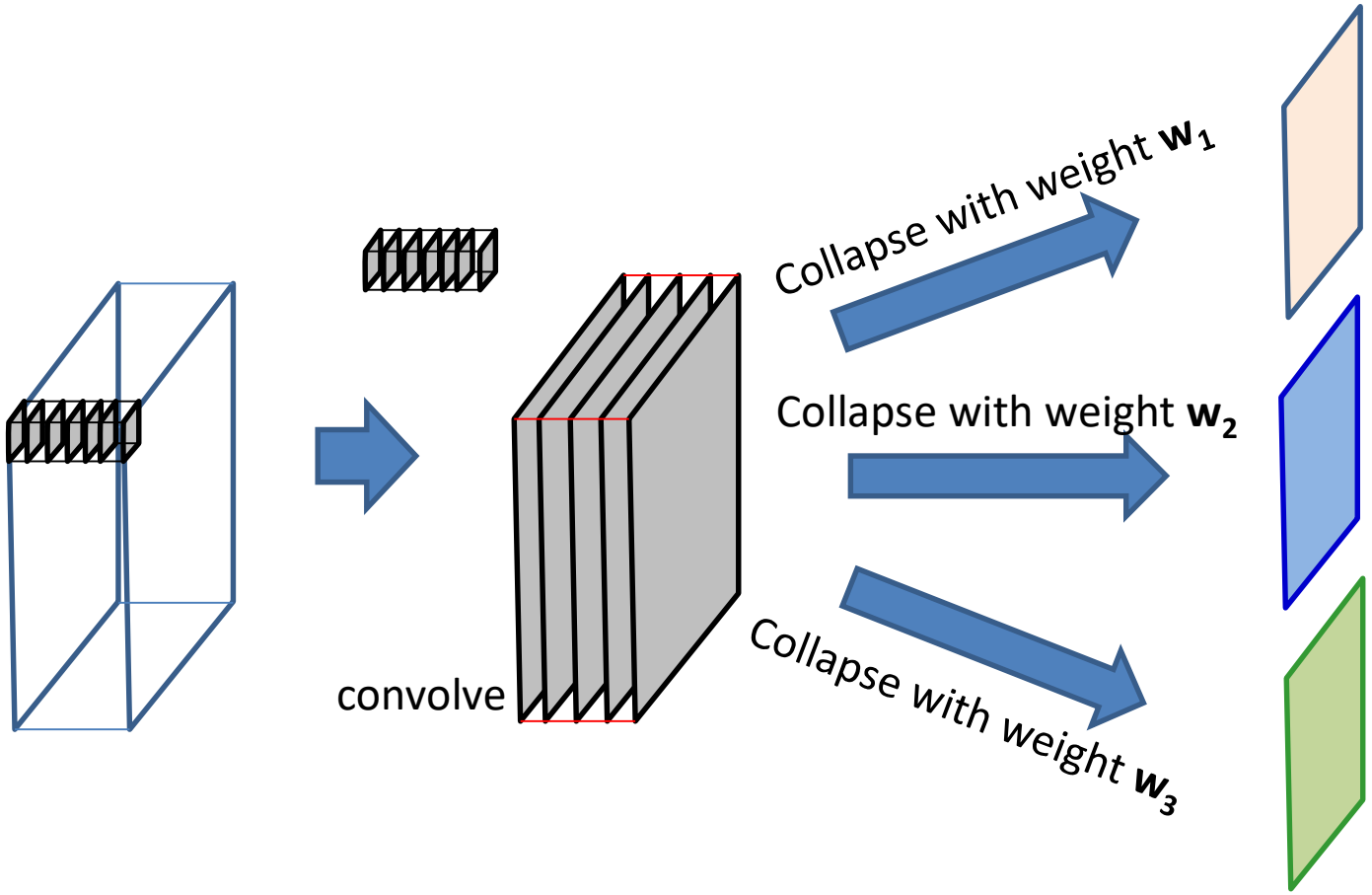
- E*ach layer of each filter* scans its corresponding map to produce a convolved map
- N input channels will require a filter with N layers
- The independent convolutions of each layer of the filter result in N convolved maps
- The N convolved maps are *added together* to produce the final output map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

Conventional convolutions



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

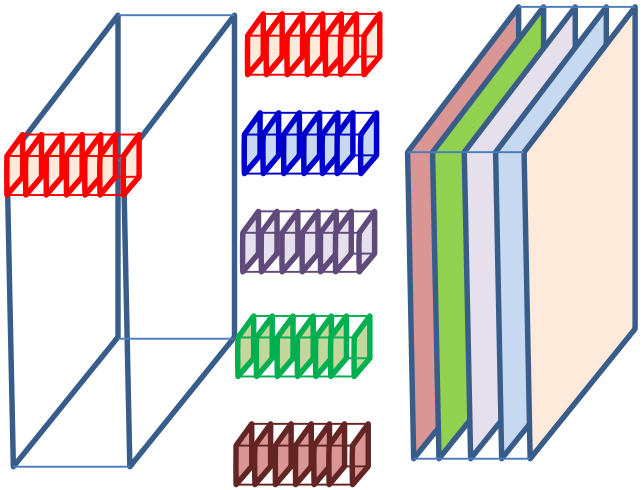
Depth-wise convolution



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

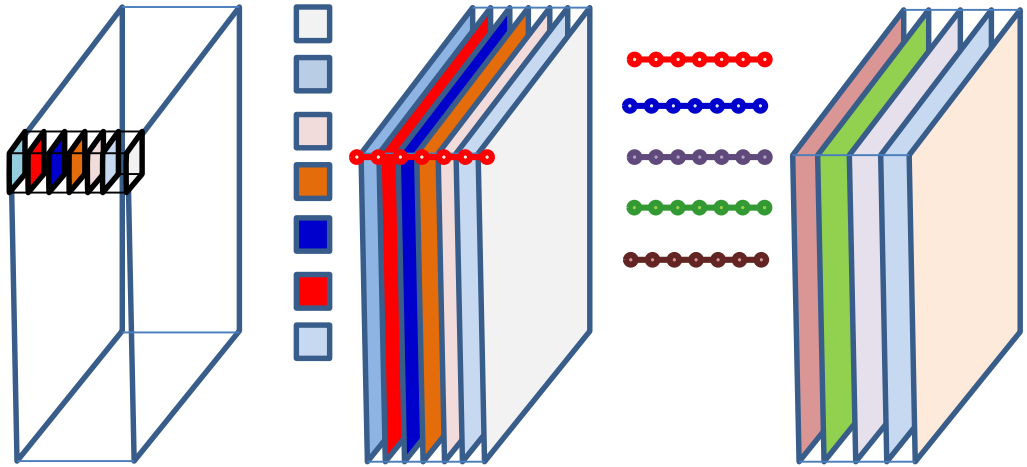
Conventional vs. depth-wise convolution

Conventional



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

Depth-wise

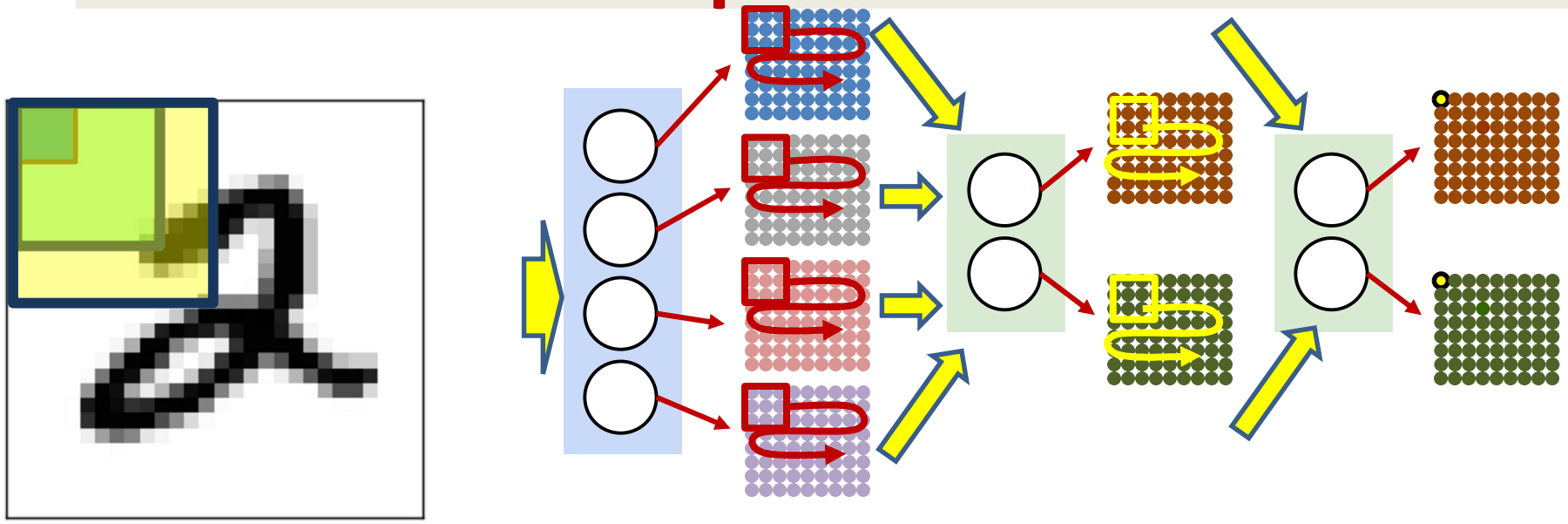


- M input channels, N output channels in 2 stages:
- Stage 1:
  - M independent KxK **2D** filters, one per input channel
  - Each filter applies to only one input channel
  - No. of output channels = no. of input channels
- Stage 2:
  - N Mx1x1 1D filters
  - Each applies to *one* 2D location across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

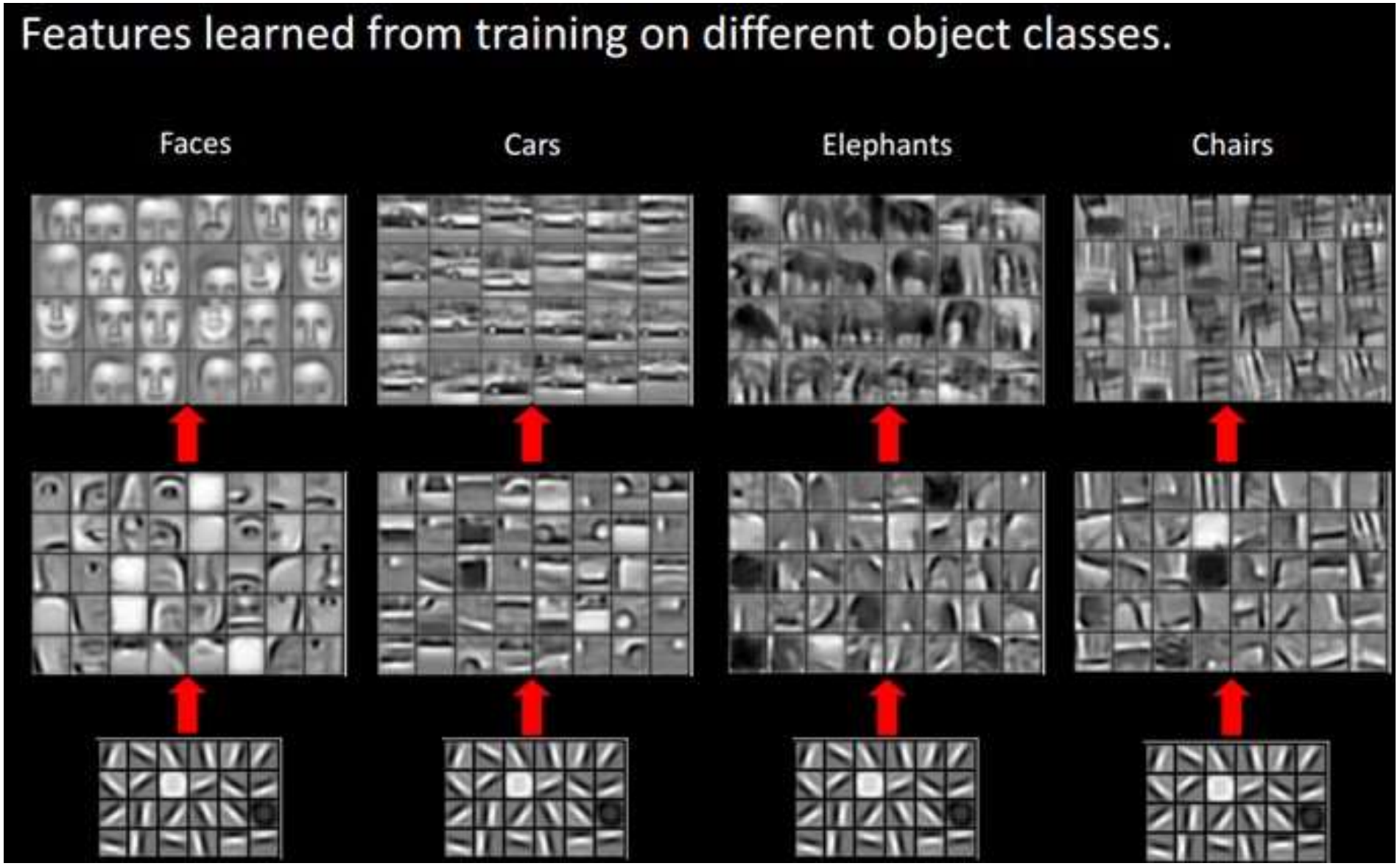
Story so far

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

What do the filters learn? Receptive fields



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

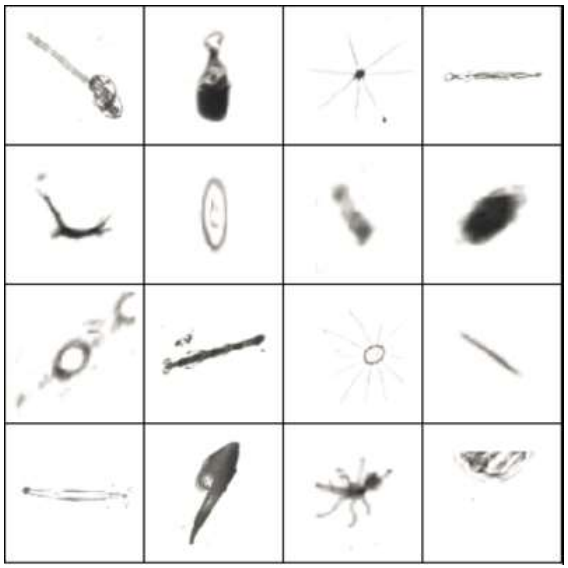

Features learned from training on different object classes.
Faces
Cars
Elephants
Chairs

Training Issues

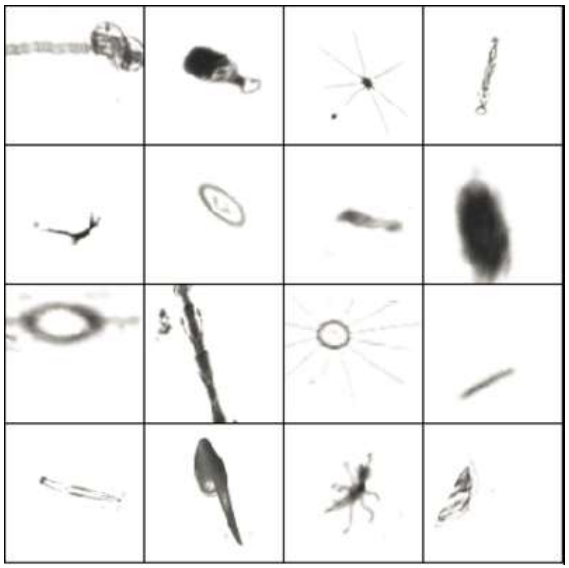
- Standard convergence issues
  - Solution: Adam or other momentum-style algorithms
  - Other tricks such as batch normalization
- The number of parameters can quickly become very large
- Insufficient training data to train well
  - Solution: Data augmentation

Data Augmentation

Original data



Augmented data



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

Convolutional neural nets

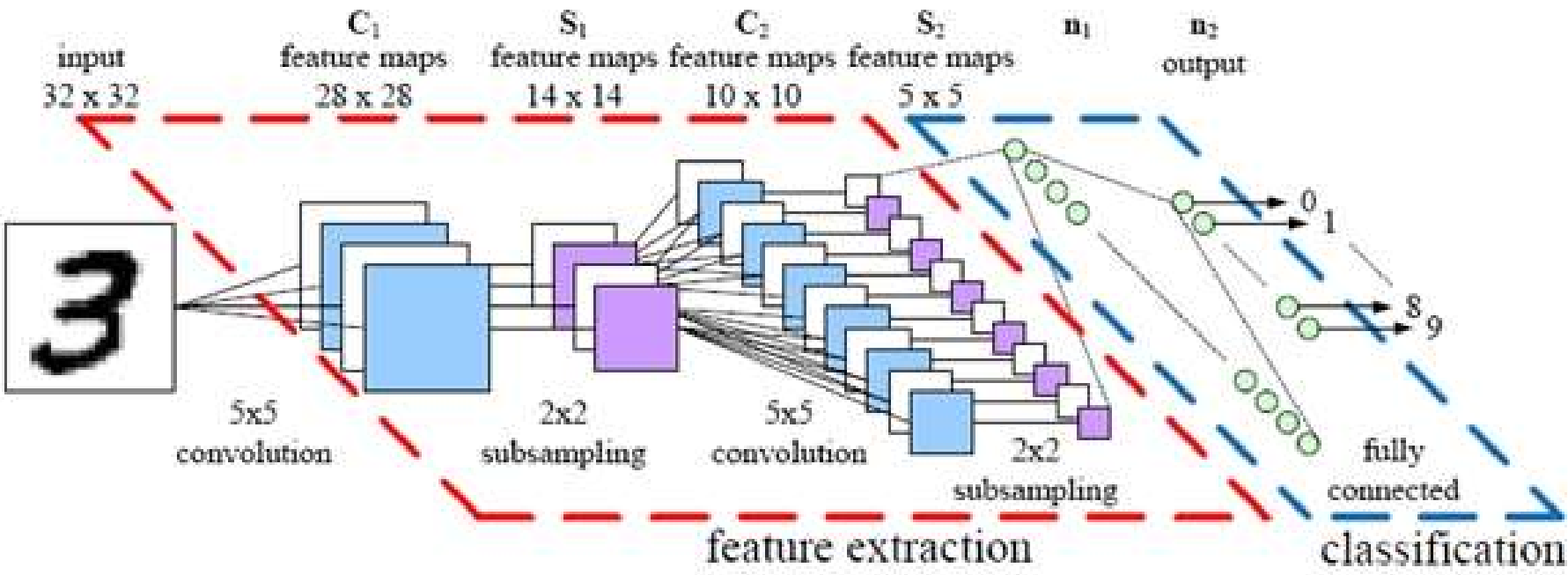
- One of *the* most frequently used nnet formalism today

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

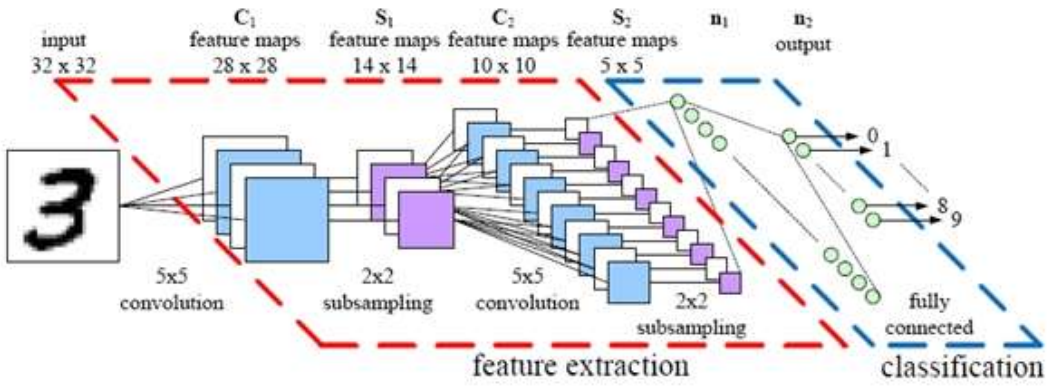
Nice visual example

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

Digit classification

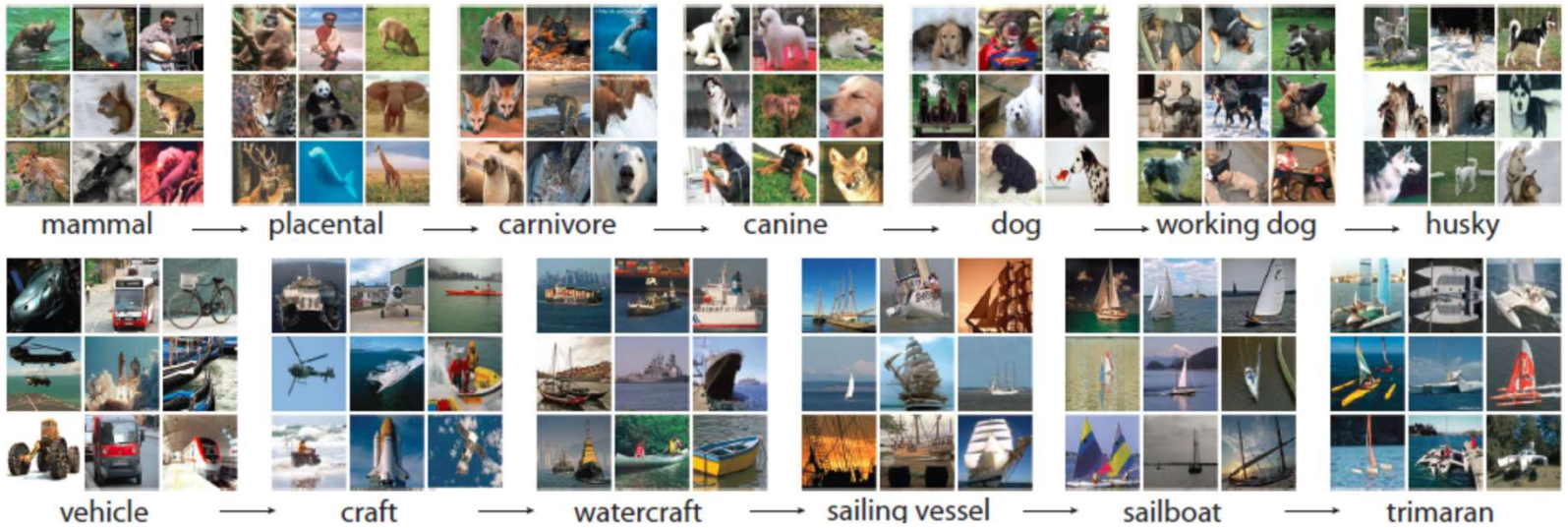


Le-net 5



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

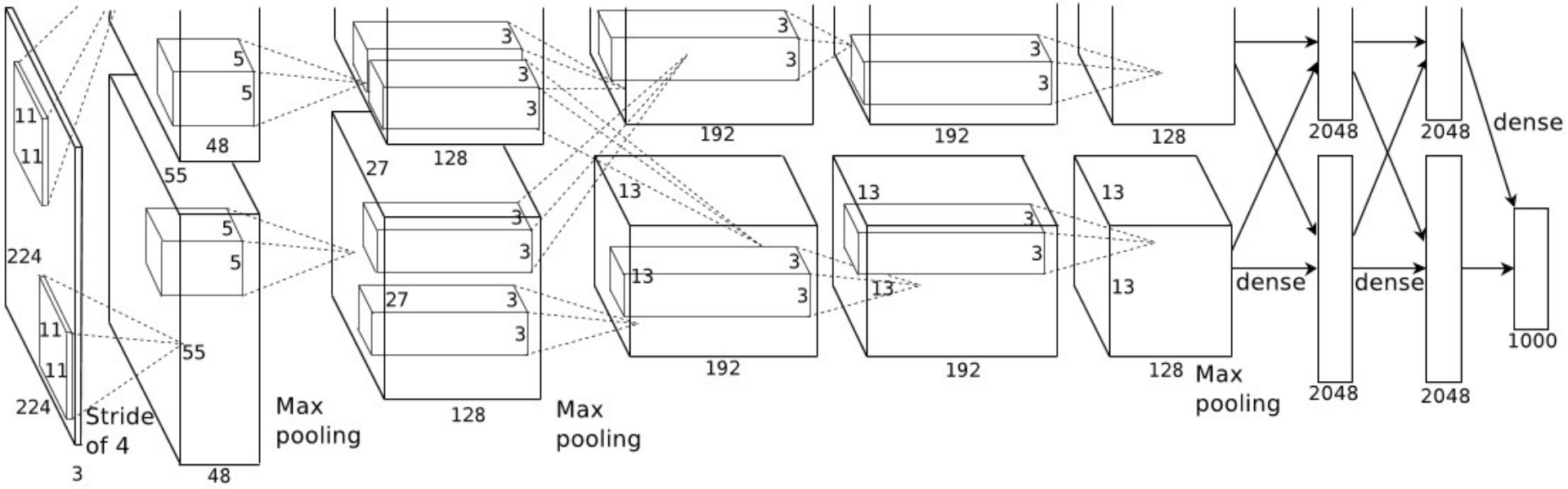
The imagenet task



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

AlexNet

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



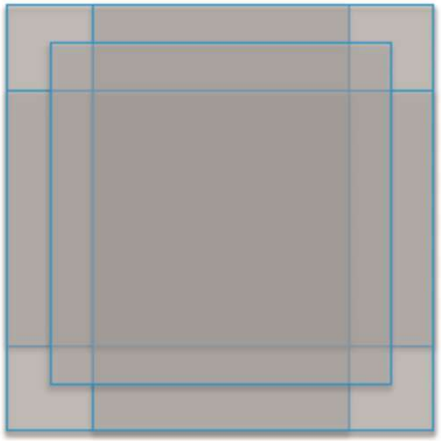
Krizhevsky, A., Sutskever, I. and Hinton, G. E. "ImageNet Classification with Deep Convolutional Neural Networks" NIPS 2012: Neural Information Processing Systems, Lake Tahoe, Nevada

Krizhevsky et. al.

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

Alexnet: Total parameters

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



10 patches

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

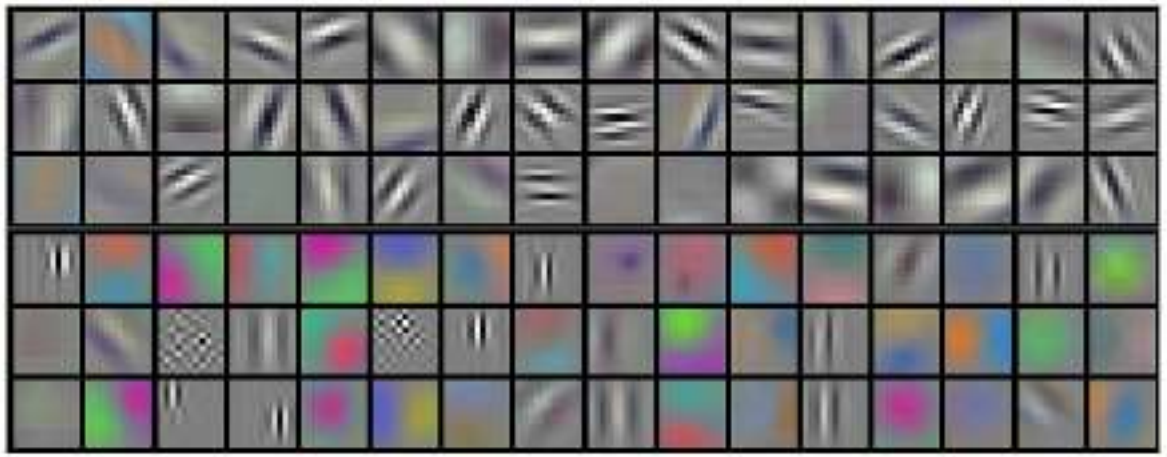
Learning magic in Alexnet

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

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

ImageNet

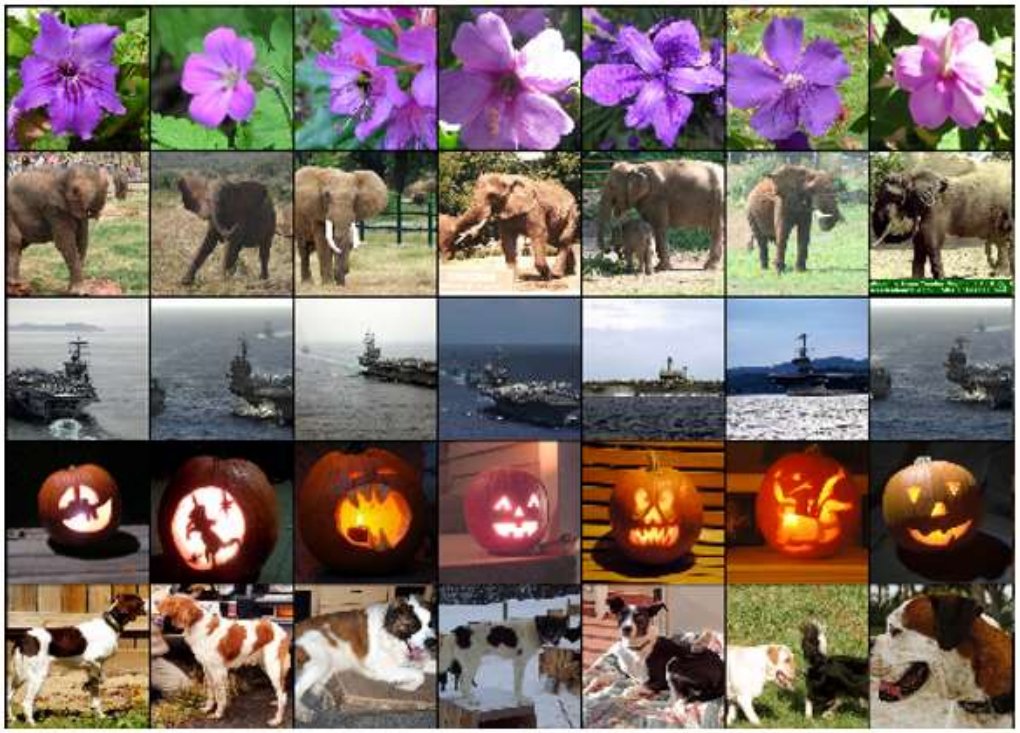
Figure 3: 96 convolutional kernels of size 11×11×3 learned by the first convolutional layer on the 224×224×3 input images. The top 48 kernels were learned on GPU 1 while the bottom 48 kernels were learned on GPU 2. See Section 6.1 for details.



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

The net actually *learns* features!

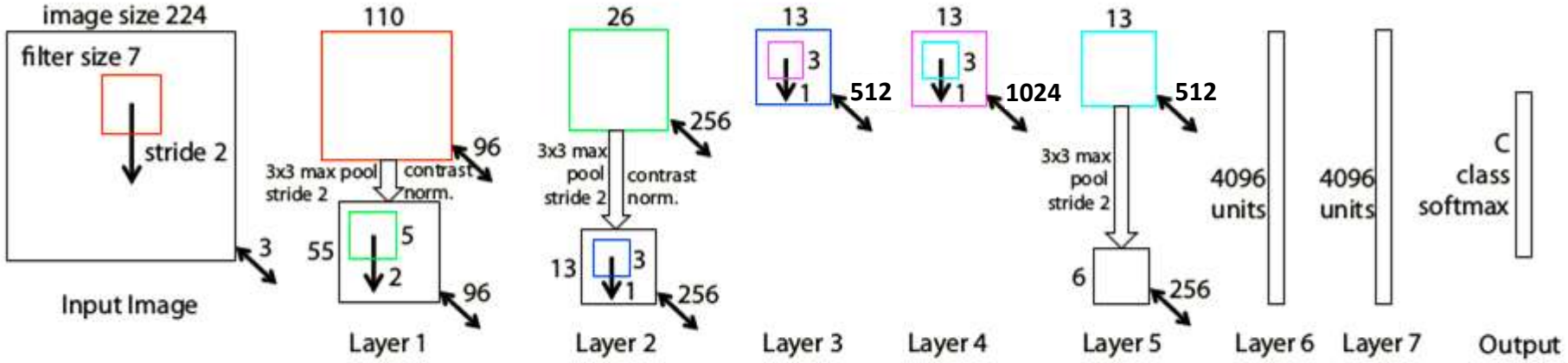
Eight ILSVRC-2010 test images and the five labels considered most probable by our model. The correct label is written under each image, and the probability assigned to the correct label is also shown with a red bar (if it happens to be in the top 5).



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

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

ZFNet



ZF Net Architecture

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

VGGNet

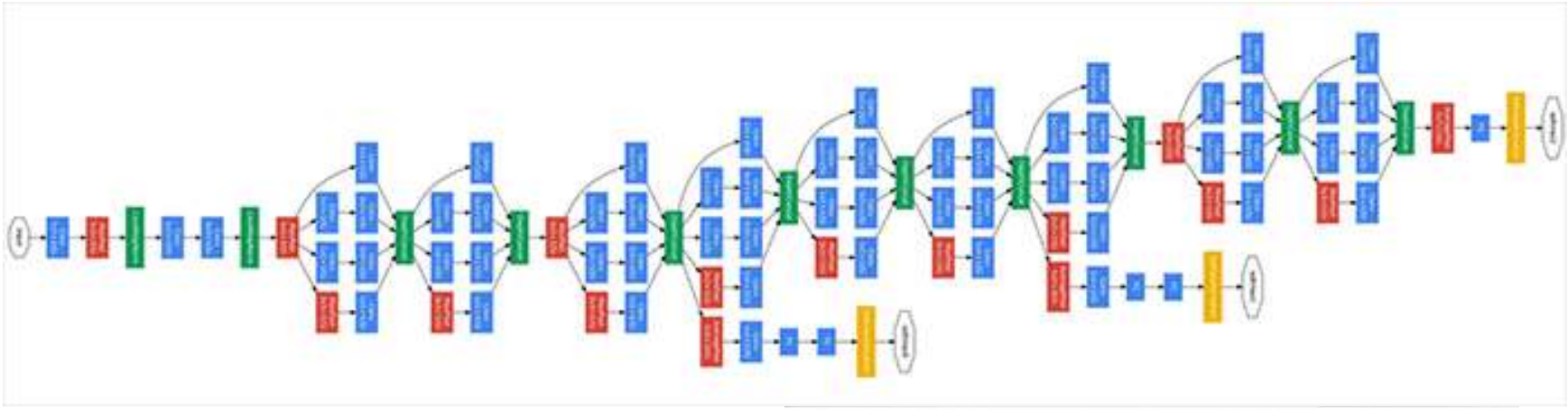
- Simonyan and Zisserman, 2014
- *Only* used 3x3 filters, stride 1, pad 1
- *Only* used 2x2 pooling filters, stride 2

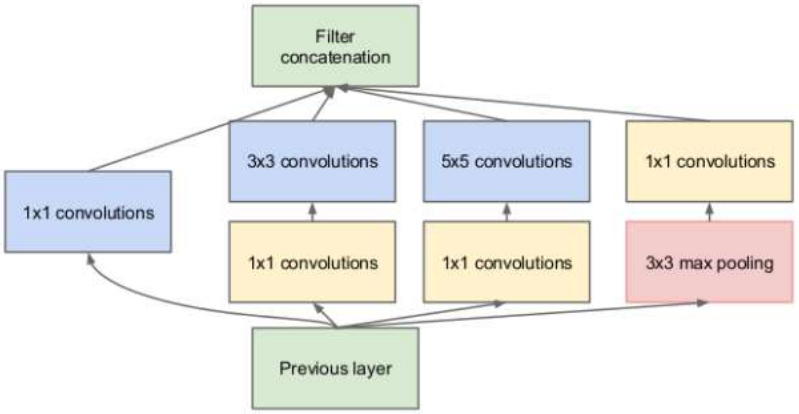
- Tried a large number of architectures.
- Finally obtained 7.3% top-5 error using 13 conv layers and 3 FC layers
  - Combining 7 classifiers
  - Subsequent to paper, reduced error to 6.8% using only two classifiers
- Final arch: 64 conv, 64 conv, 64 pool, 128 conv, 128 conv, 128 pool, 256 conv, 256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

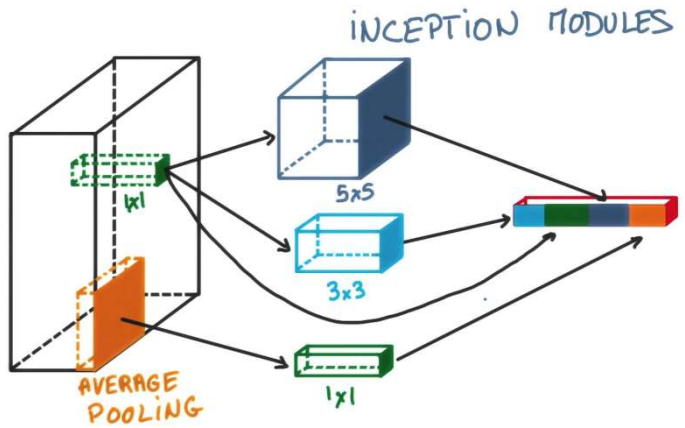
ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128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

Madness!

Googlenet: Inception







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

Resnet

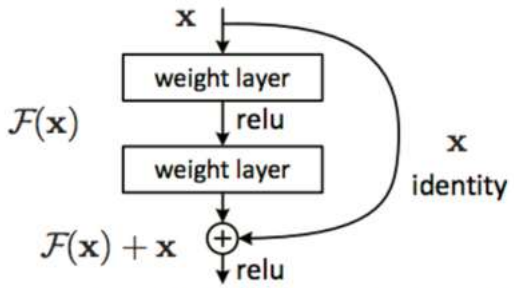
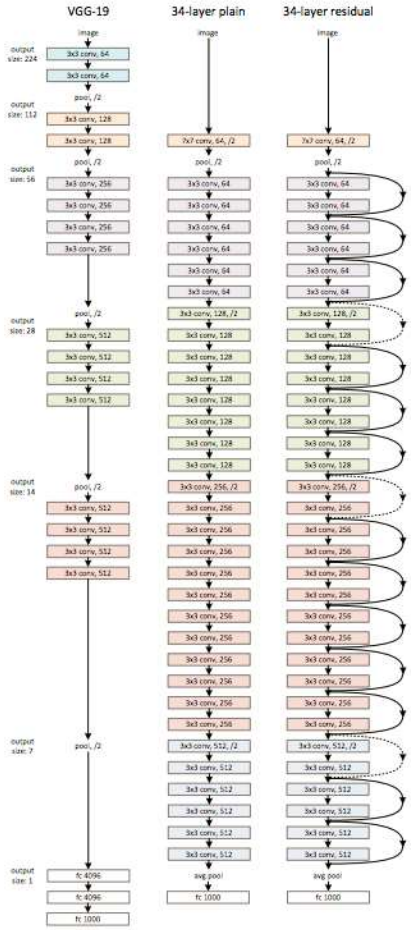


Figure 2. Residual learning: a building block.



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

Resnet details for the curious..

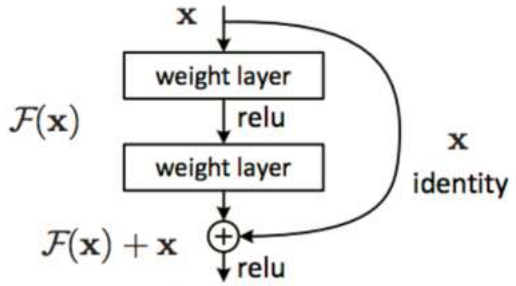
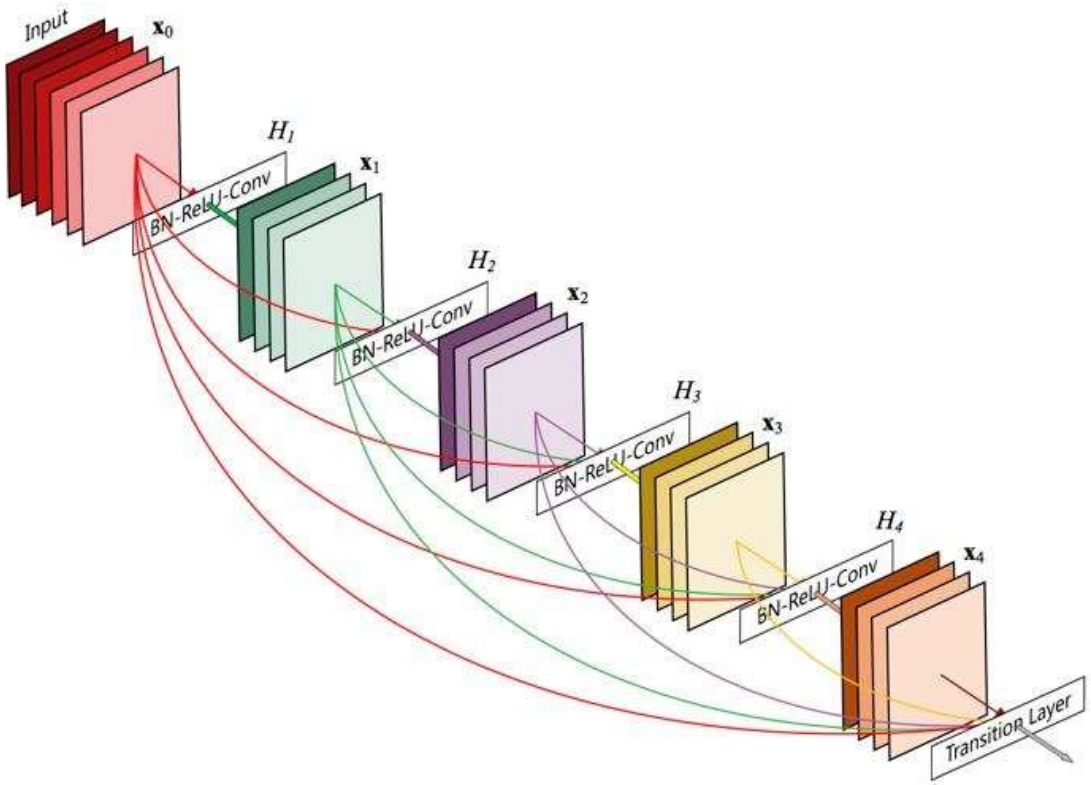


Figure 2. Residual learning: a building block.

- 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..

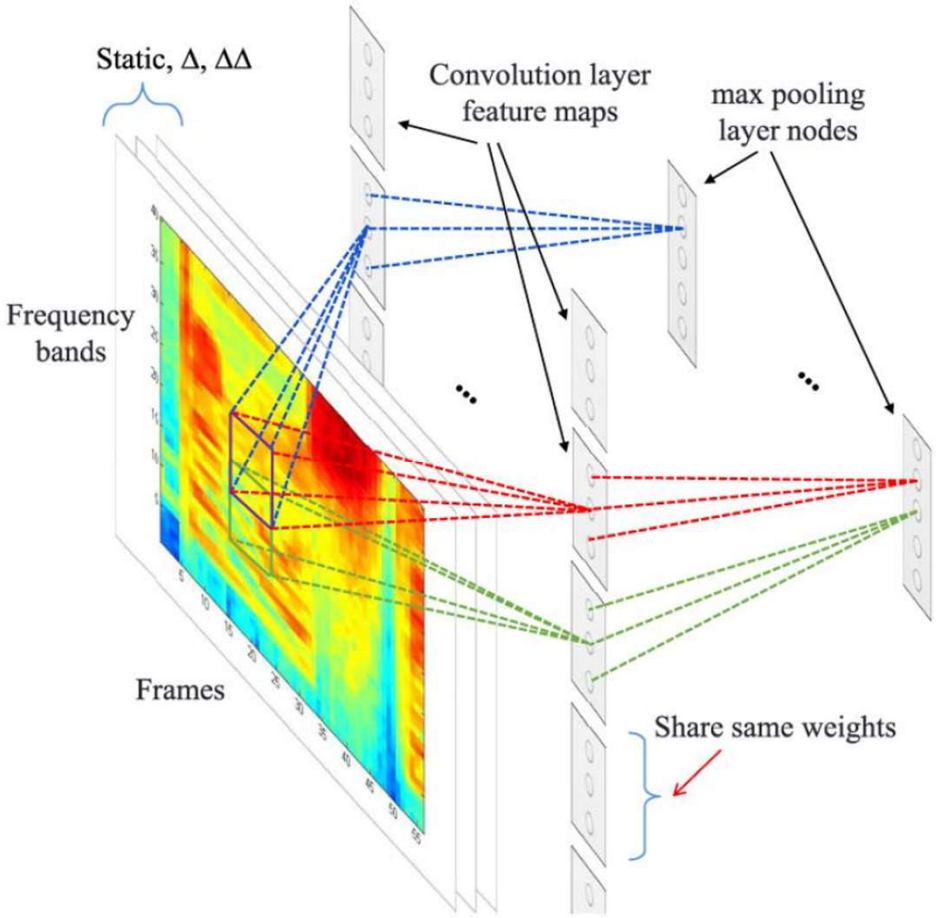
Many many more architectures

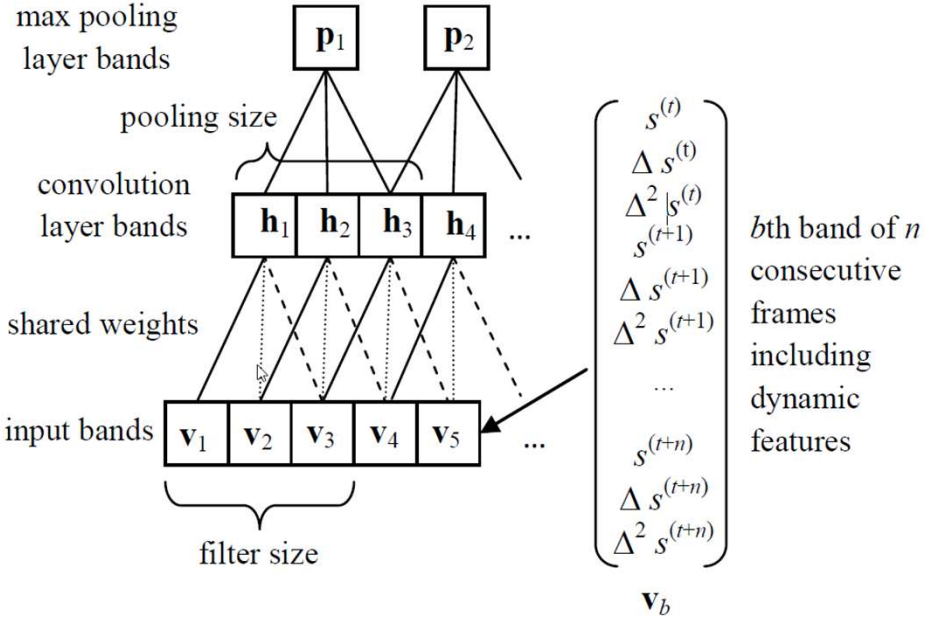
- Daily updates on arxiv..

- 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