# Deep Neural Networks
# Convolutional Networks II

Bhiksha Raj

# Story so far

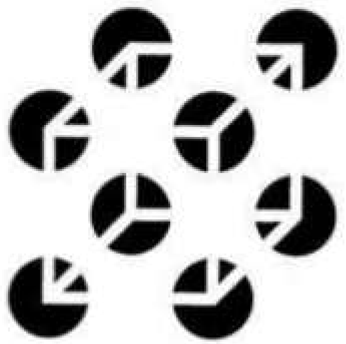
- Pattern classification tasks such as "does this picture contain a cat", or "does this recording include HELLO" are best performed by scanning for the target pattern

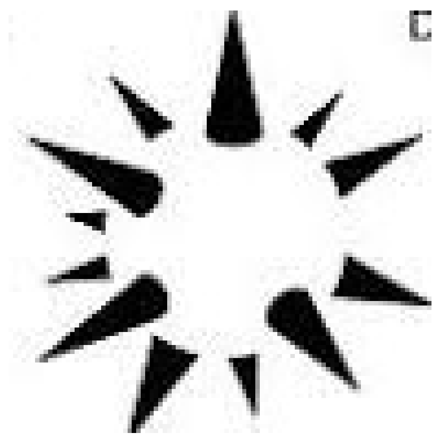
- Scanning an input with a network and combining the outcomes is equivalent to scanning with individual neurons
  - First level neurons scan the input
  - Higher-level neurons scan the "maps" formed by lower-level neurons
  - A final "decision" unit or layer makes the final decision

- Deformations in the input can be handled by "max pooling"

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

## A little history



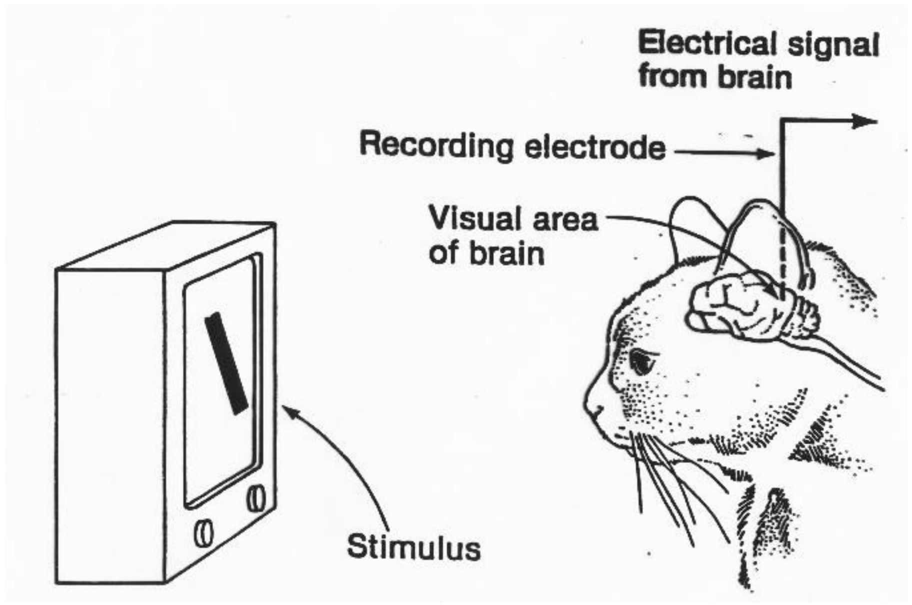


- How do animals see?
  - What is the neural process from eye to recognition?
- Early research:
  - largely based on behavioral studies
    - Study behavioral judgment in response to visual stimulation
    - Visual illusions
  - and gestalt
    - Brain has innate tendency to organize disconnected bits into whole objects
  - But no real understanding of how the brain processed images

## Hubel and Wiesel 1959

- First study on neural correlates of vision.
  - "Receptive Fields in Cat Striate Cortex"
    - "Striate Cortex": Approximately equal to the V1 visual cortex
      - "Striate" – defined by structure, "V1" – functional definition
- 24 cats, anaesthetized, immobilized, on artificial respirators
  - Anaesthetized with truth serum
  - Electrodes into brain
    - Do not report if cats survived experiment, but claim brain tissue was studied

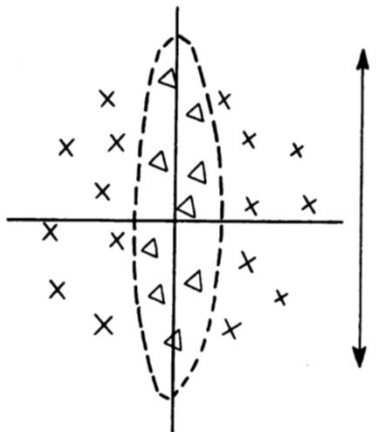
## Hubel and Wiesel 1959

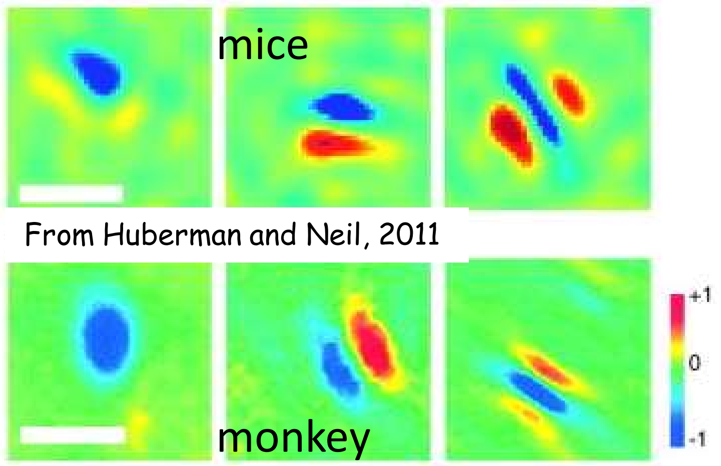


- Light of different wavelengths incident on the retina through fully open (slitted) Iris
  - Defines *immediate* (20ms) response of these cells

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

# Hubel and Wiesel 1959

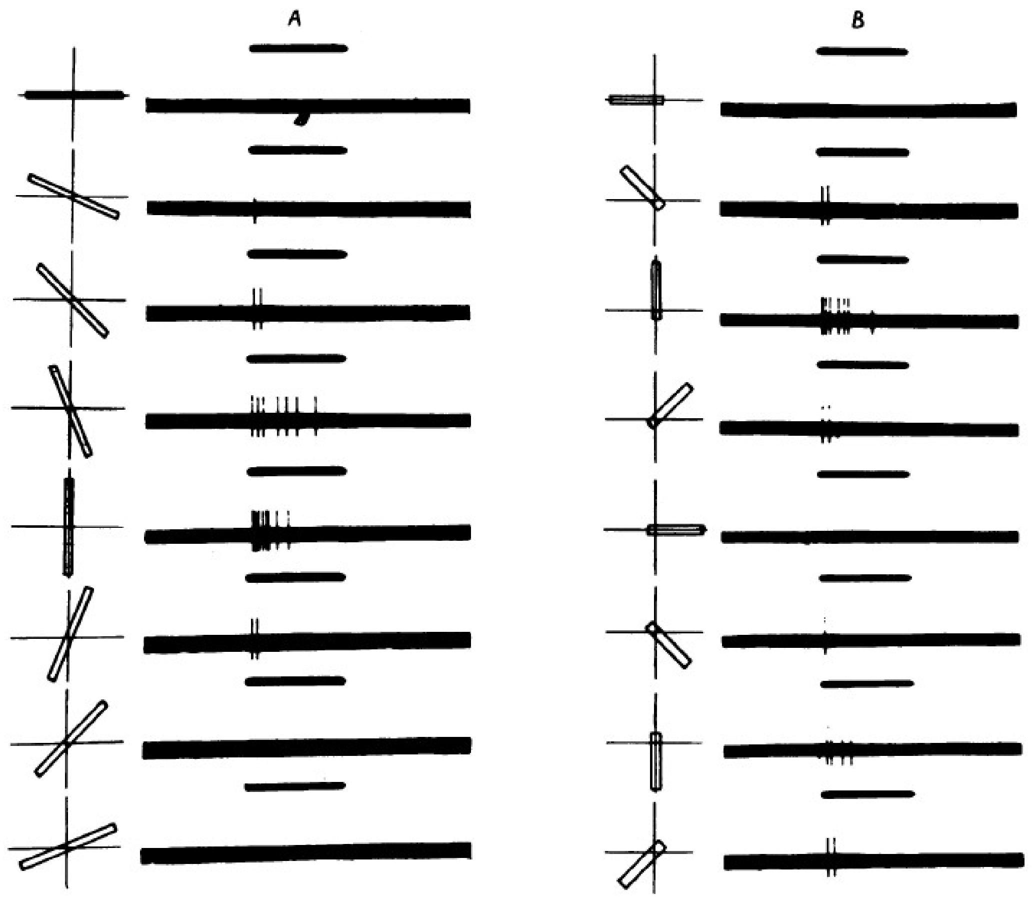

From Hubel and Wiesel



From Huberman and Neil, 2011

- Restricted retinal areas which on illumination influenced the firing of single cortical units were called ***receptive fields***.
  - These fields were usually subdivided into excitatory and inhibitory regions.
- Findings:
  - A light stimulus covering the whole receptive field, or diffuse illumination of the whole retina, was ineffective in driving most units, as excitatory regions cancelled inhibitory regions
    - Light must fall on excitatory regions and NOT fall on inhibitory regions, resulting in clear patterns
  - Receptive fields could be oriented in a vertical, horizontal or oblique manner.
    - Based on the arrangement of excitatory and inhibitory regions within receptive fields.
  - A spot of light gave greater response for some directions of movement than others.

## Hubel and Wiesel 59

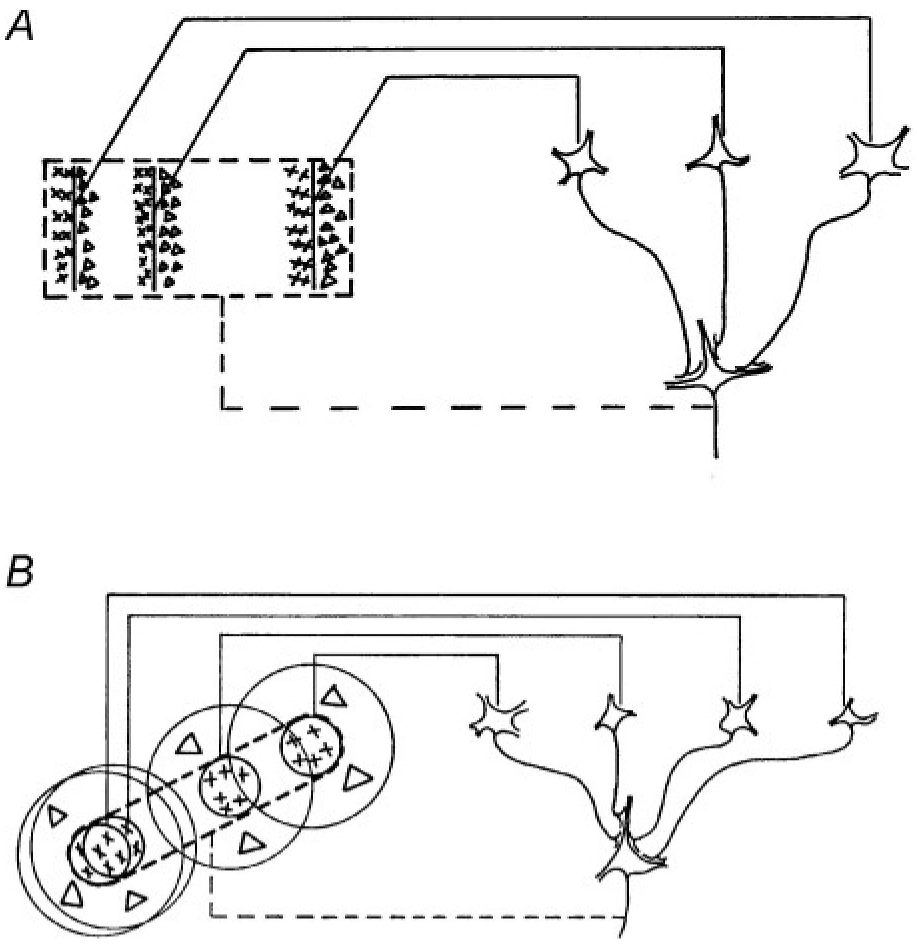


- Response as orientation of input light rotates
  - Note spikes – this neuron is sensitive to vertical bands

## Hubel and Wiesel

- Oriented slits of light were the most effective stimuli for activating striate cortex neurons
- The orientation selectivity resulted from *the previous level of input* because lower level neurons responding to a slit also responded to patterns of spots if they were aligned with the same orientation as the slit.
- In a later paper (Hubel & Wiesel, 1962), they showed that within the striate cortex, two levels of processing could be identified
  - Between neurons referred to as *simple* S-cells and *complex* C-cells.
  - Both types responded to oriented slits of light, but complex cells were not "confused" by spots of light while simple cells could be confused

# Hubel and Wiesel model



Composition of complex receptive fields from simple cells. The C-cell responds to the largest output from a bank of S-cells to achieve oriented response that is robust to distortion

Transform from circular retinal receptive fields to elongated fields for simple cells. The simple cells are susceptible to fuzziness and noise

## Hubel and Wiesel

- Complex C-cells build from similarly oriented simple cells
  - They “finetune” the response of the simple cell

- Show complex buildup – building *more complex patterns* by composing early neural responses
  - Successive transformation through Simple-Complex combination layers

- Demonstrated more and more complex responses in later papers
  - Later experiments were on waking macaque monkeys
    - Too horrible to recall

## Hubel and Wiesel

## Adding insult to injury..

- “However, this model cannot accommodate the color, spatial frequency and many other features to which neurons are tuned.  The exact organization of all these cortical columns within V1 remains a hot topic of current research.”

## Forward to 1980

- Kunihiko Fukushima
- Recognized deficiencies in the Hubel-Wiesel model

Kunihiko Fukushima

- One of the chief problems: Position invariance of input
  – Your grandmother cell fires even if your grandmother moves to a different location in your field of vision

# NeoCognitron

Figures from Fukushima, '80

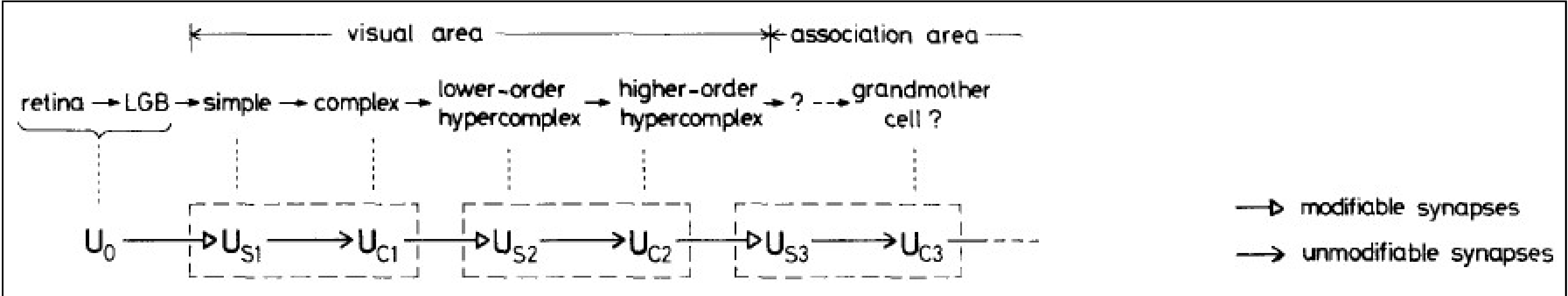


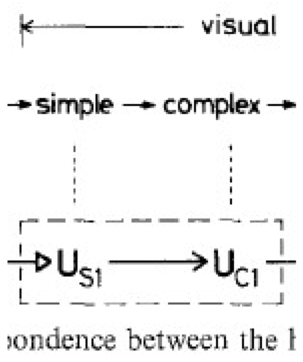
**Fig. 1.** Correspondence between the hierarchy model by Hubel and Wiesel, and the neural network of the neocognitron

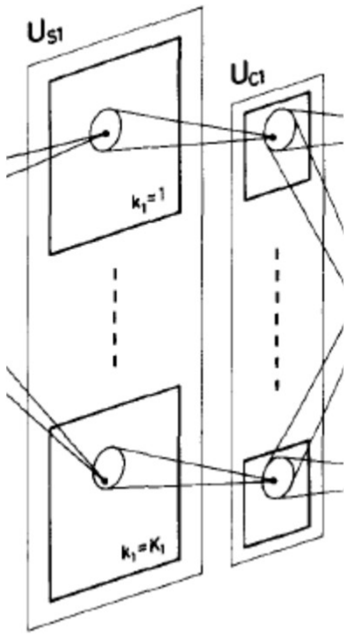
- Visual system consists of a hierarchy of modules, each comprising a layer of "S-cells" followed by a layer of "C-cells"
  - $U_{Sl}$ is the l<sup>th</sup> layer of S cells, $U_{Cl}$ is the l<sup>th</sup> layer of C cells

- Only S-cells are "plastic" (i.e. learnable), C-cells are fixed in their response

- S-cells ***respond*** to the signal in the previous layer
- C-cells ***confirm*** the S-cells' response

# NeoCognitron





Each cell in a plane "looks" at a slightly shifted region of the input to the plane than the adjacent cells in the plane.

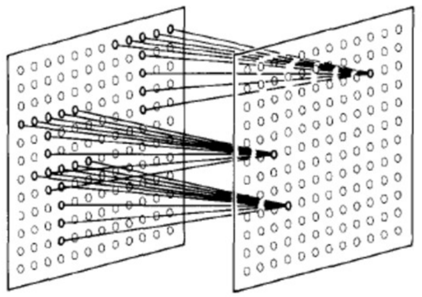
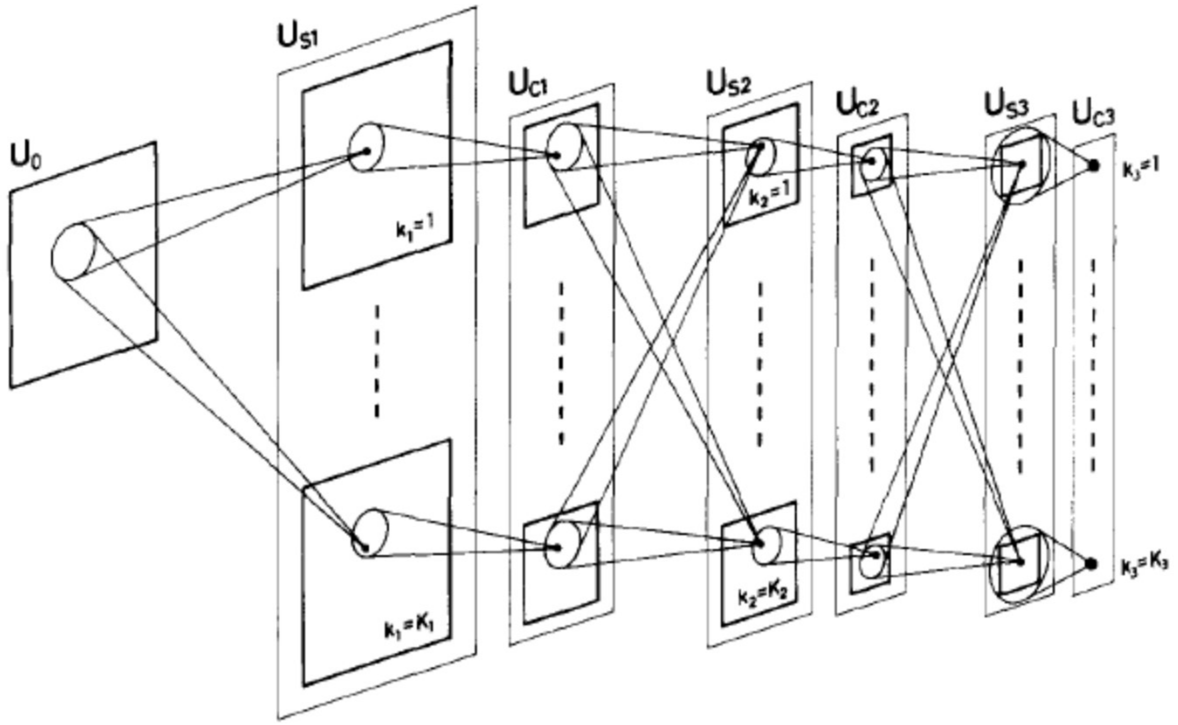


Fig. 3. Illustration showing the input interconnections to the cells within a single cell-plane

- Each simple-complex module includes a layer of S-cells and a layer of C-cells
- S-cells are organized in rectangular groups called S-planes.
  - All the cells within an S-plane have identical learned responses
- C-cells too are organized into rectangular groups called C-planes
  - One C-plane per S-plane
  - All C-cells have identical fixed response
- In Fukushima's original work, each C and S cell "looks" at an elliptical region in the previous plane

# NeoCognitron



- The complete network
- $U_0$ is the retina
- In each subsequent module, the planes of the S layers detect plane-specific patterns in the previous layer (C layer or retina)
- The planes of the C layers “refine” the response of the corresponding planes of the S layers

## Neocognitron

- S cells: RELU like activation

$$u_{Sl}(k_l, \mathbf{n}) = r_l \cdot \varphi\left[\frac{1 + \sum_{k_{l-1}=1}^{K_{l-1}} \sum_{\mathbf{v} \in S_l} a_l(k_{l-1}, \mathbf{v}, k_l) \cdot u_{Cl-1}(k_{l-1}, \mathbf{n}+\mathbf{v})}{1 + \frac{2r_l}{1+r_l} \cdot b_l(k_l) \cdot v_{Cl-1}(\mathbf{n})} - 1\right]$$

  - $\varphi$ is a RELU

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

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

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

## Neocognitron

- S cells: RELU like activation

$$u_{Sl}(k_l, \mathbf{n}) = r_l \cdot \varphi\left[\frac{1 + \sum_{k_{l-1}=1}^{K_{l-1}} \sum_{\mathbf{v} \in S_l} a_l(k_{l-1}, \mathbf{v}, k_l) \cdot u_{Cl-1}(k_{l-1}, \mathbf{n}+\mathbf{v})}{1 + \frac{2r_l}{1+r_l} \cdot b_l(k_l) \cdot v_{Cl-1}(\mathbf{n})} - 1\right]$$

  - $\varphi$ is a RELU

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

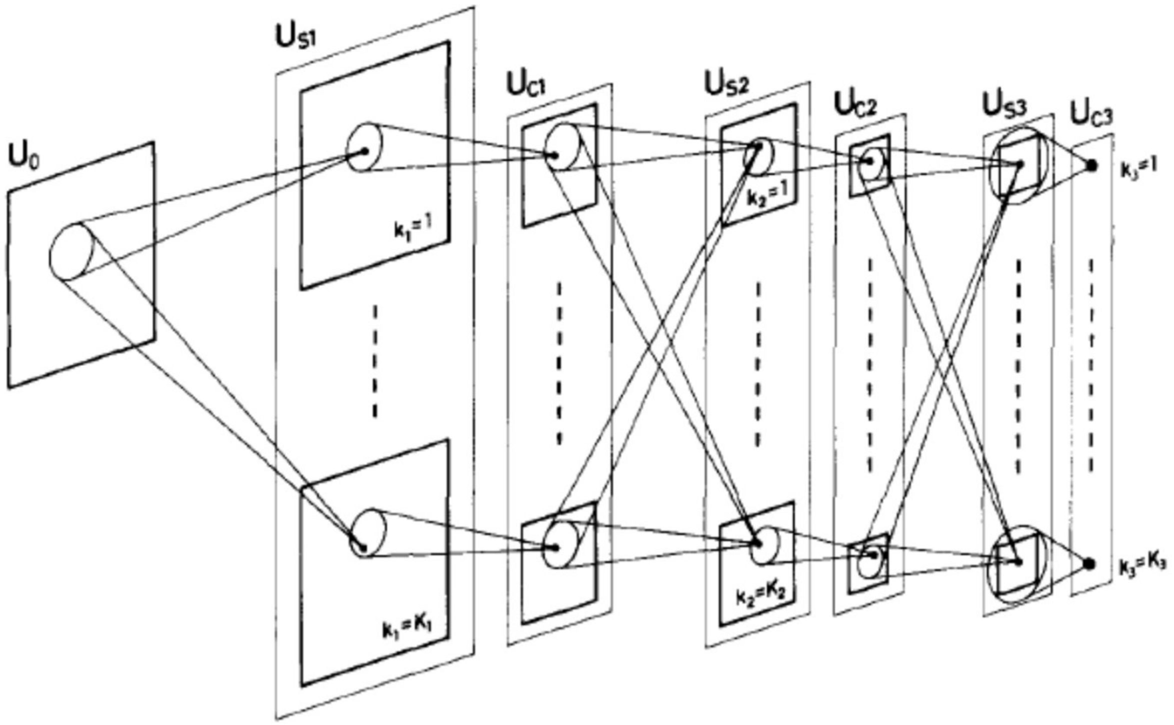
$$u_{Cl}(k_l, \mathbf{n}) = \psi\left[\frac{1 + \sum_{\mathbf{v} \in D_l} d_l(\mathbf{v}) \cdot u_{Sl}(k_l, \mathbf{n}+\mathbf{v})}{1 + v_{Sl}(\mathbf{n})} - 1\right]$$

  -

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

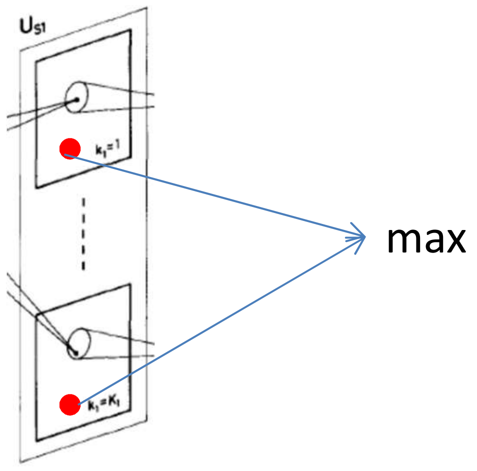
Could simply replace these strange functions with a RELU and a max

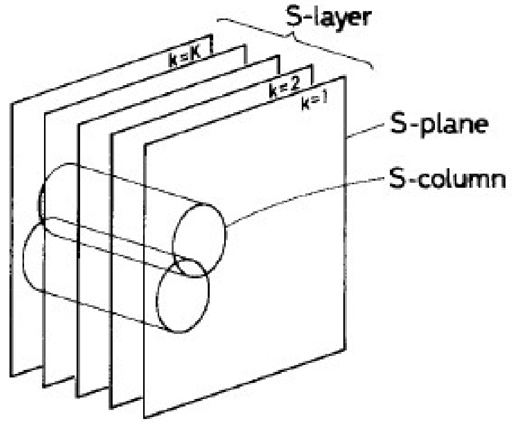
# NeoCognitron

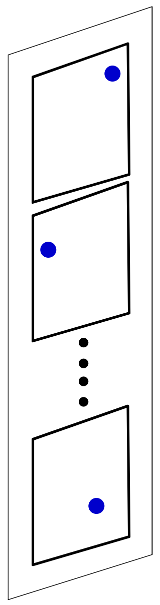


- The deeper the layer, the larger the receptive field of each neuron
  - Cell planes get smaller with layer number
  - Number of planes increases
    - i.e the number of complex pattern detectors increases with layer

# Learning in the neo-cognitron

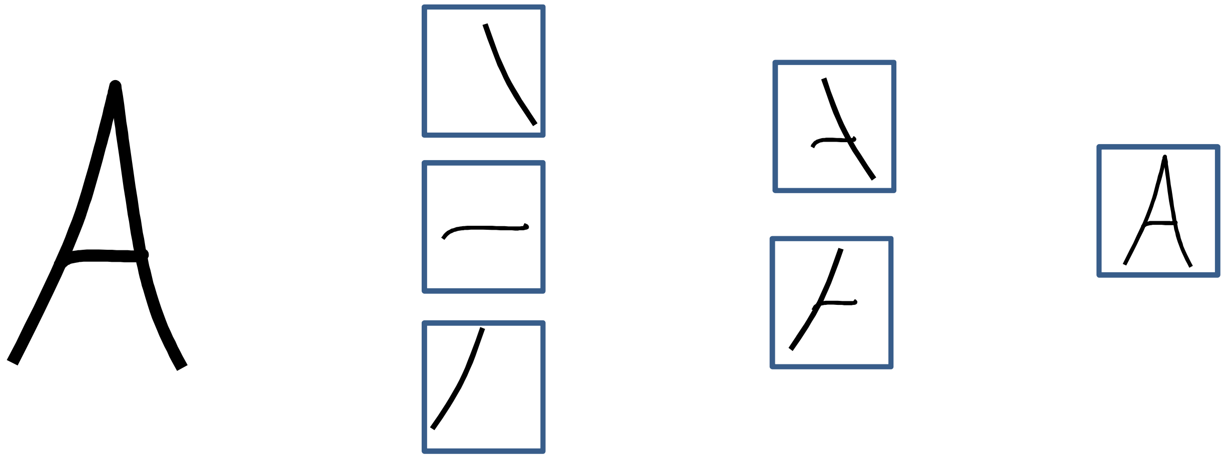






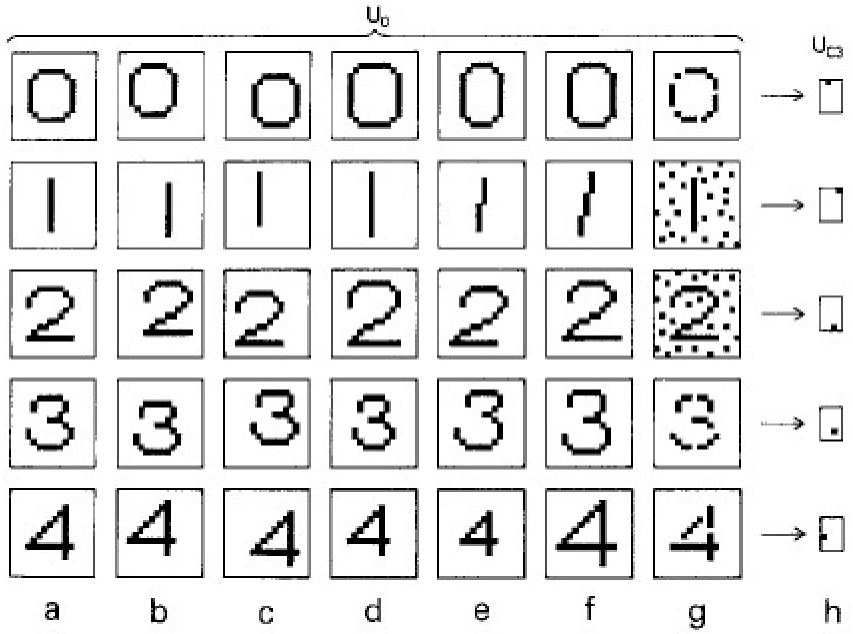
- **Unsupervised learning**
- Randomly initialize S cells, perform Hebbian learning updates in response to input
  - update = product of input and output : $\Delta w_{ij} = x_i y_j$
- Within any layer,  at any position, only the maximum S from all the layers is selected for update
  - Also viewed as max-valued cell from each *S column*
  - Ensures only one of the planes picks up any feature
  - But across all positions, multiple planes will be selected
- If multiple max selections are on the same plane, only the largest is chosen
- Updates are distributed across all cells within the plane

# Learning in the neo-cognitron



- Ensures different planes learn different features
- Any plane learns only one feature
  - E.g. Given many examples of the character “A” the different cell planes in the S-C layers may learn the patterns shown
    - Given other characters, other planes will learn their components
  - Going up the layers goes from local to global receptor fields
- Winner-take-all strategy makes it robust to distortion
- Unsupervised: Effectively clustering

# Neocognitron – finale



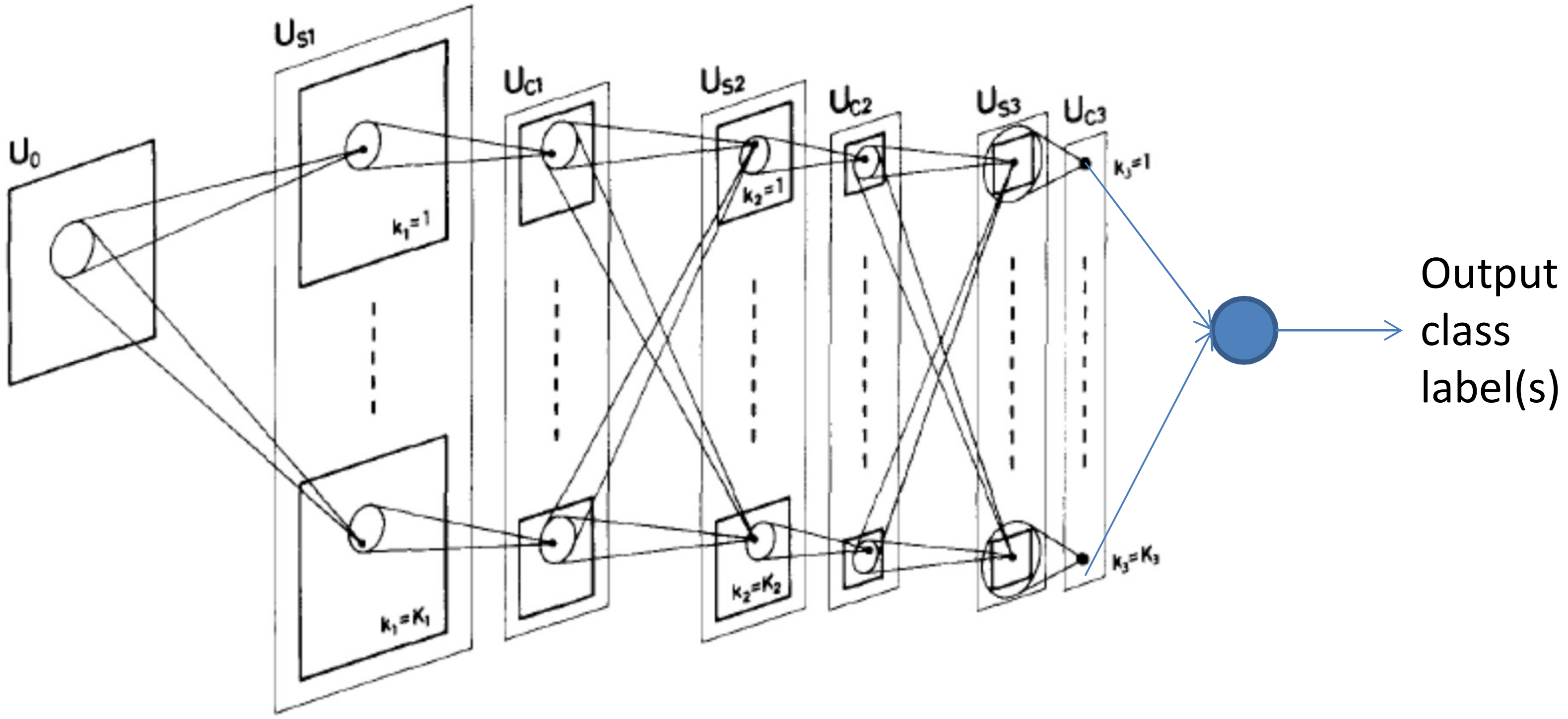
- Fukushima showed it successfully learns to cluster semantic visual concepts
  - E.g. number or characters, even in noise

## Adding Supervision

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

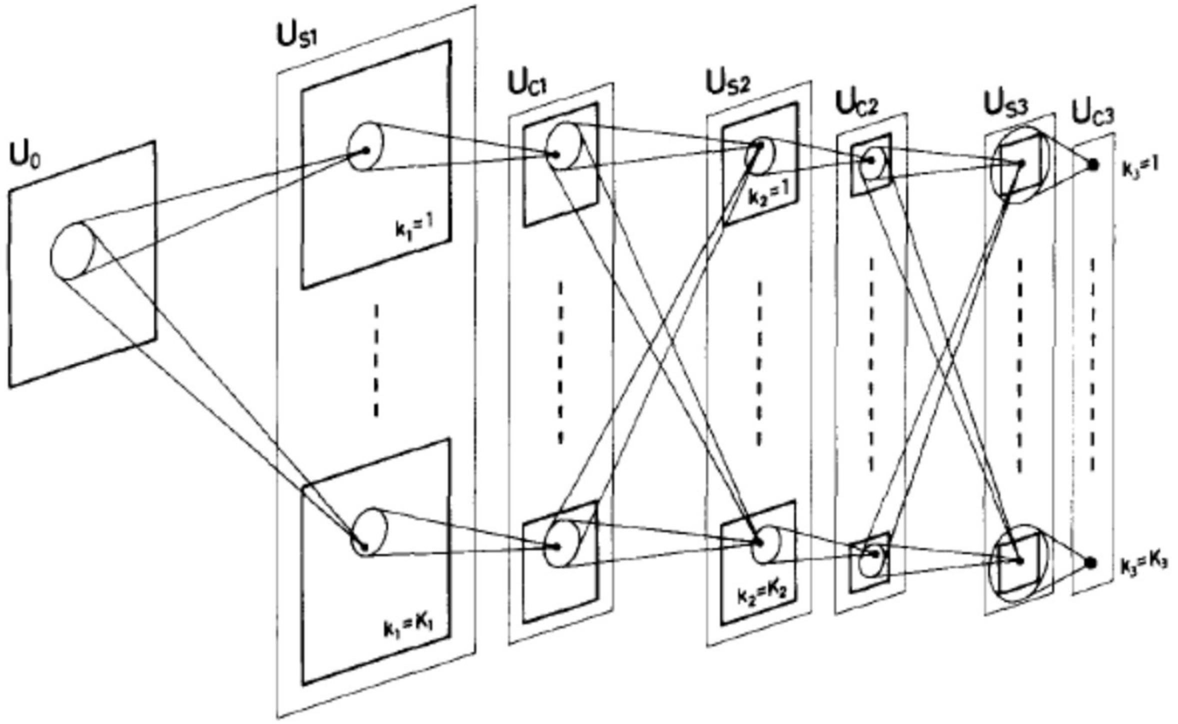
- Convolutional neural networks: LeCun

# Supervising the neocognitron



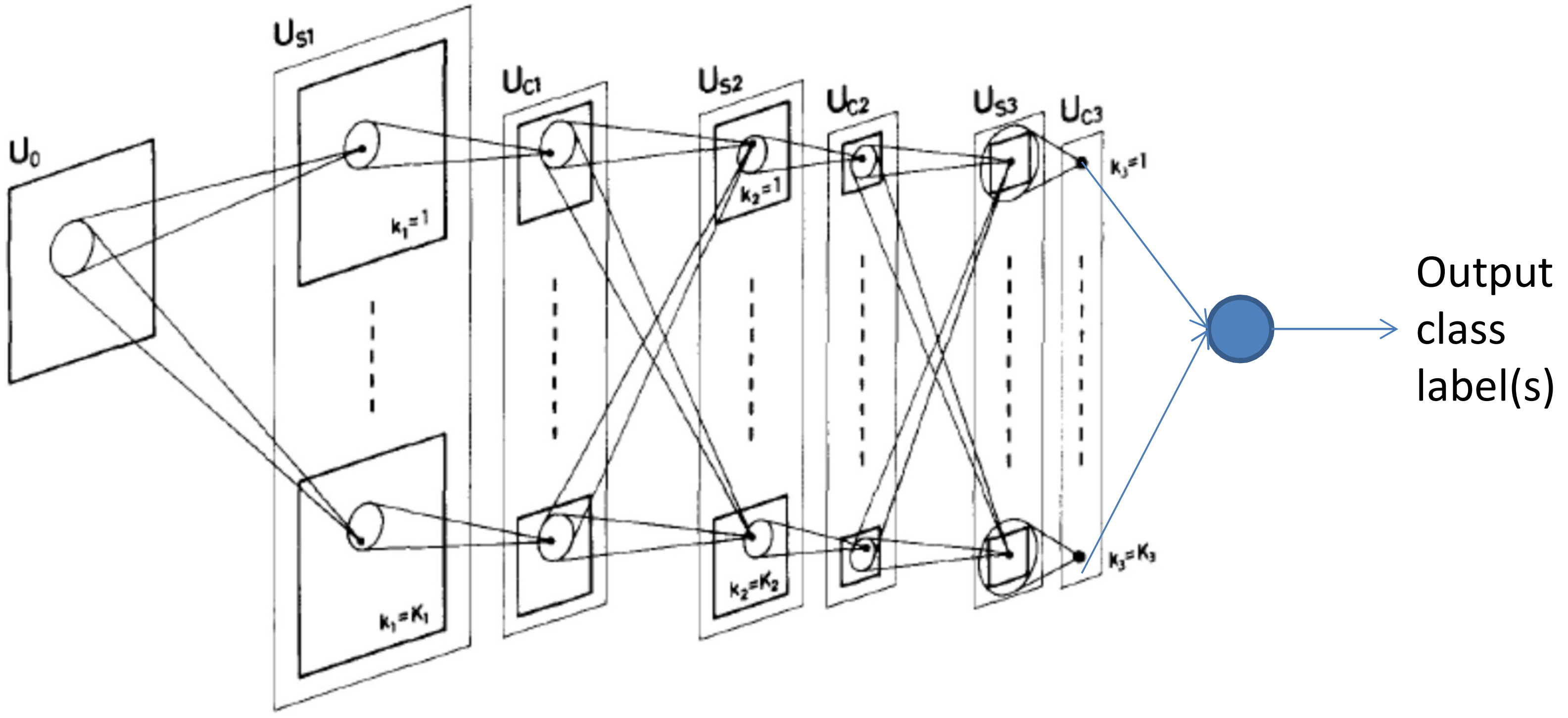
- Add an extra decision layer after the final C layer
  - Produces a class-label output
- We now have a fully feed forward MLP with shared parameters
  - All the S-cells within an S-plane have the same weights
- Simple backpropagation can now train the S-cell weights in every plane of every layer
  -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

## Scanning vs. multiple filters



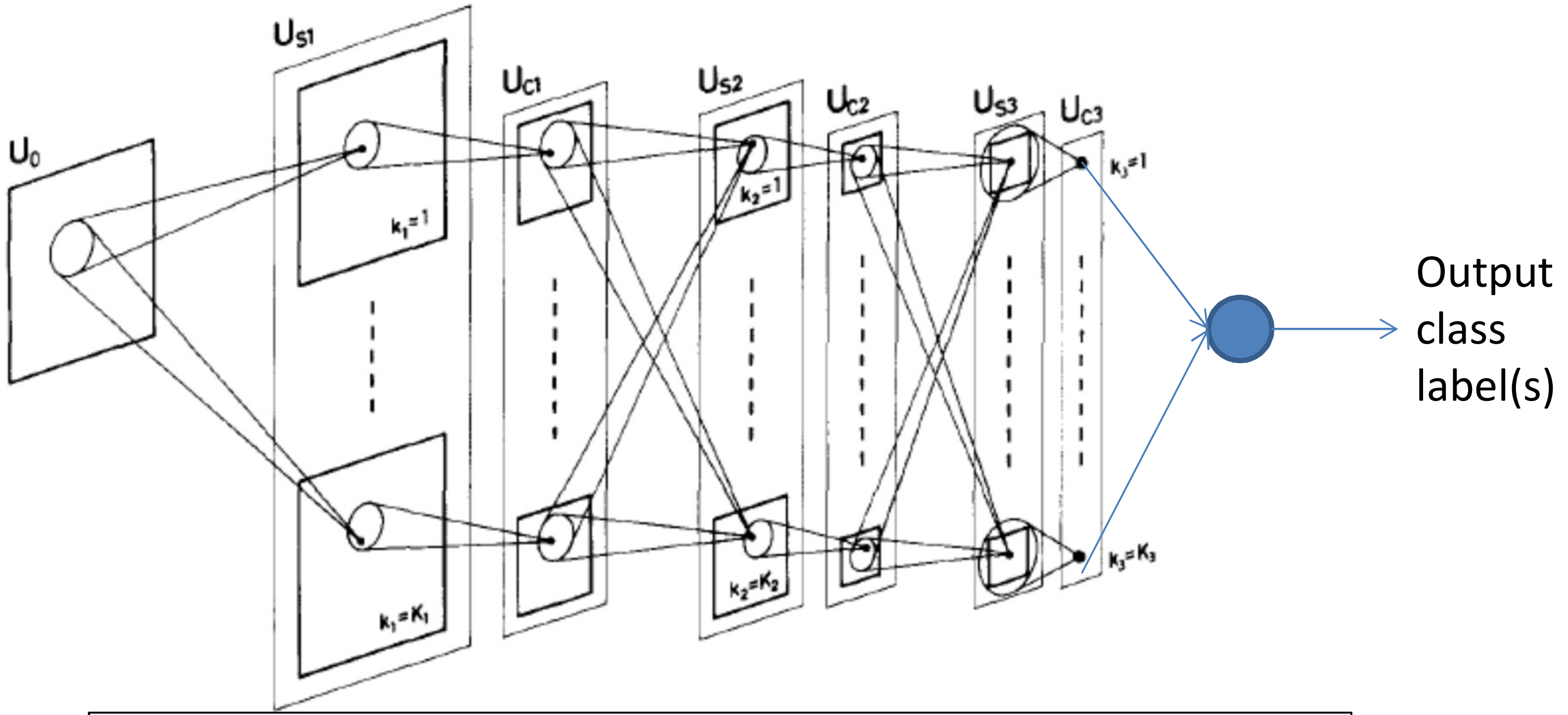
- **Note**: The original Neocognitron actually uses many identical copies of a neuron in each S and C plane

# Supervising the neocognitron



- The Math
  - Assuming *square* receptive fields, rather than elliptical ones
  - Receptive field of S cells in lth layer is $K_l \times K_l$
  - Receptive field of C cells in lth layer is $L_l \times L_l$

# Supervising the neocognitron

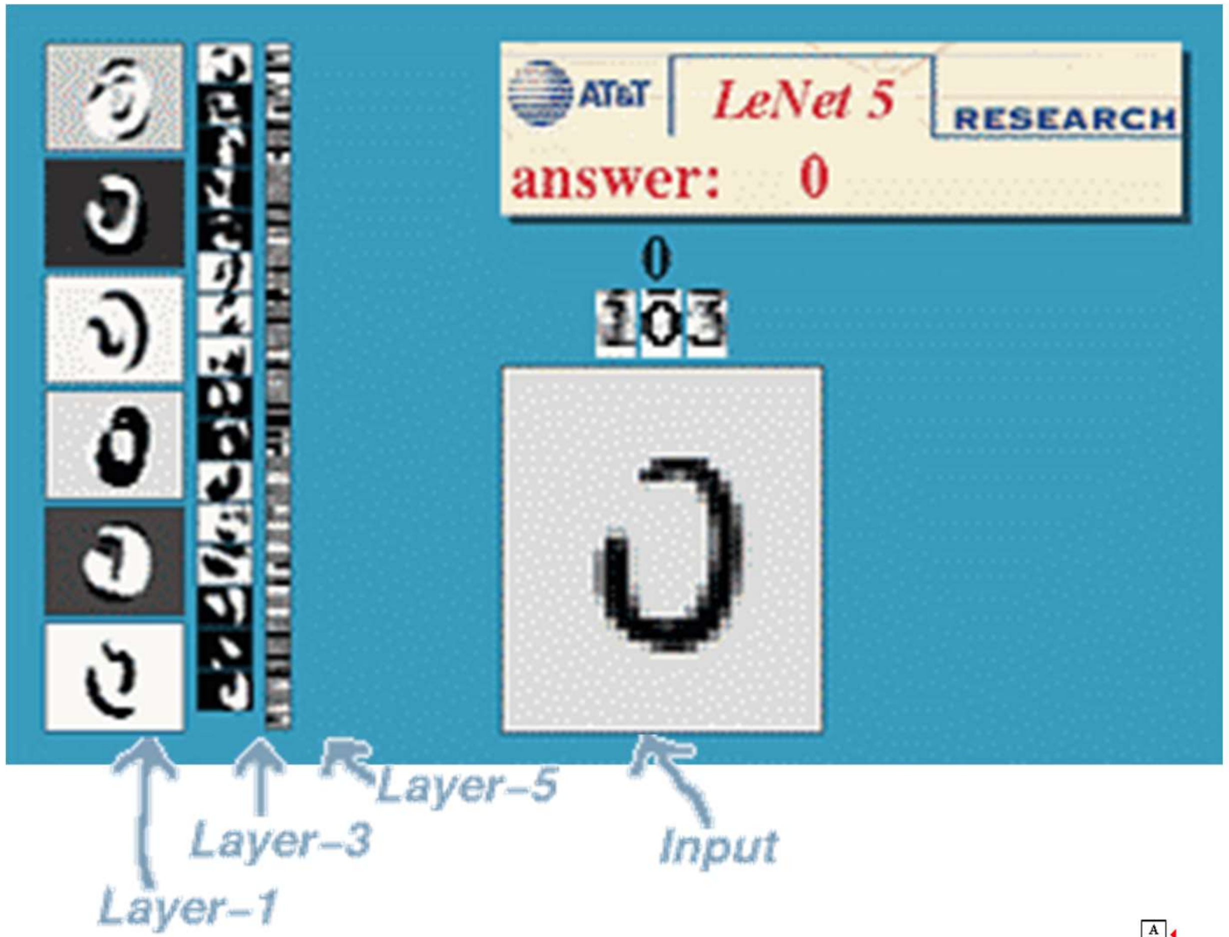


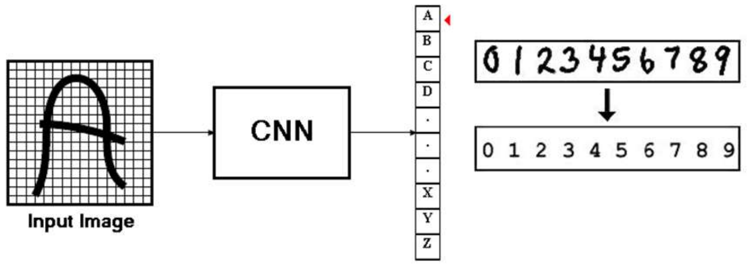
$$\boxed{U_{S,l,n}(i,j) = \sigma\left(\sum_{p}\sum_{k=1}^{K_l}\sum_{l=1}^{K_l} w_{S,l,n}(p,k,l)\,U_{C,l-1,p}(i+l,j+k)\right)}$$

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

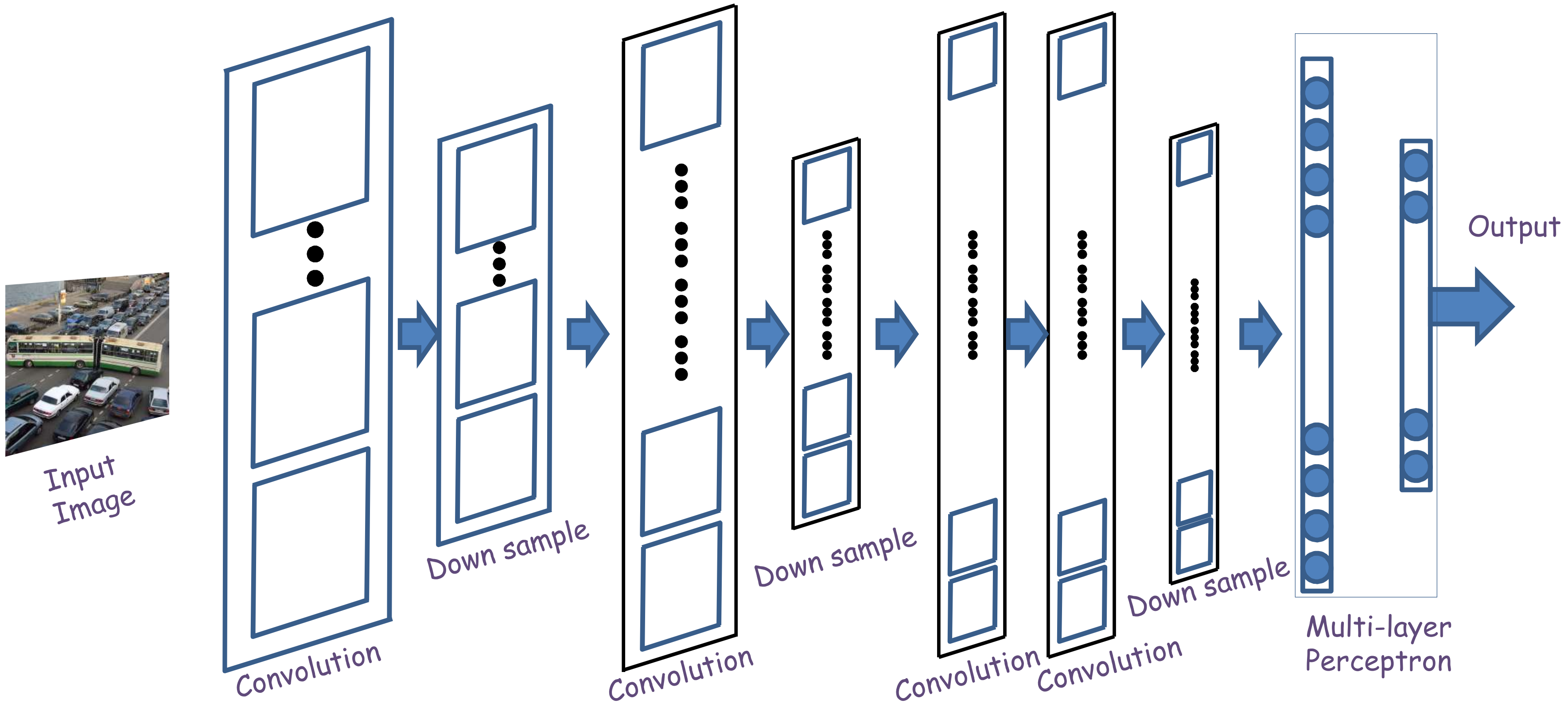
- This is, however, identical to “scanning” (convolving) with a single neuron/filter (what LeNet actually did)

# Convolutional Neural Networks



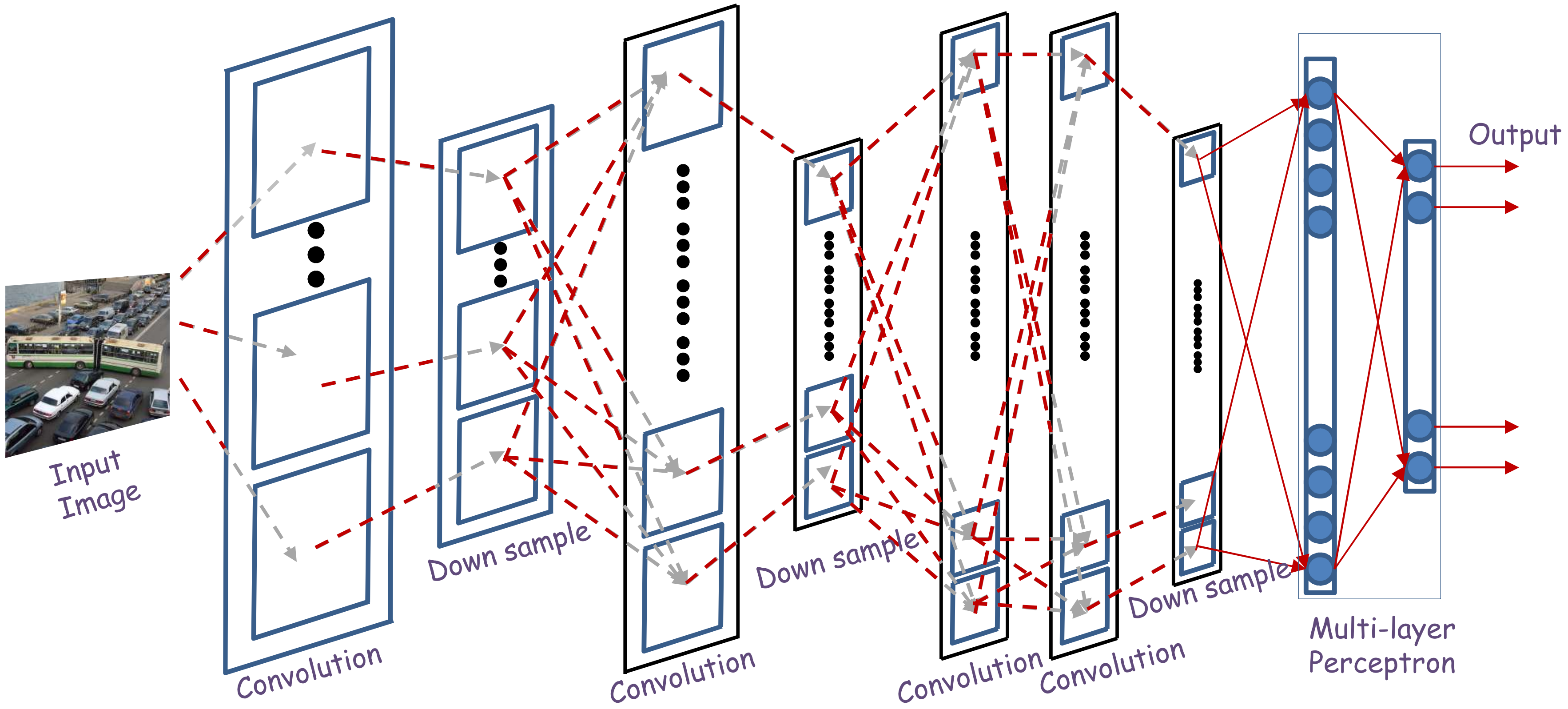


## The general architecture of a convolutional neural network



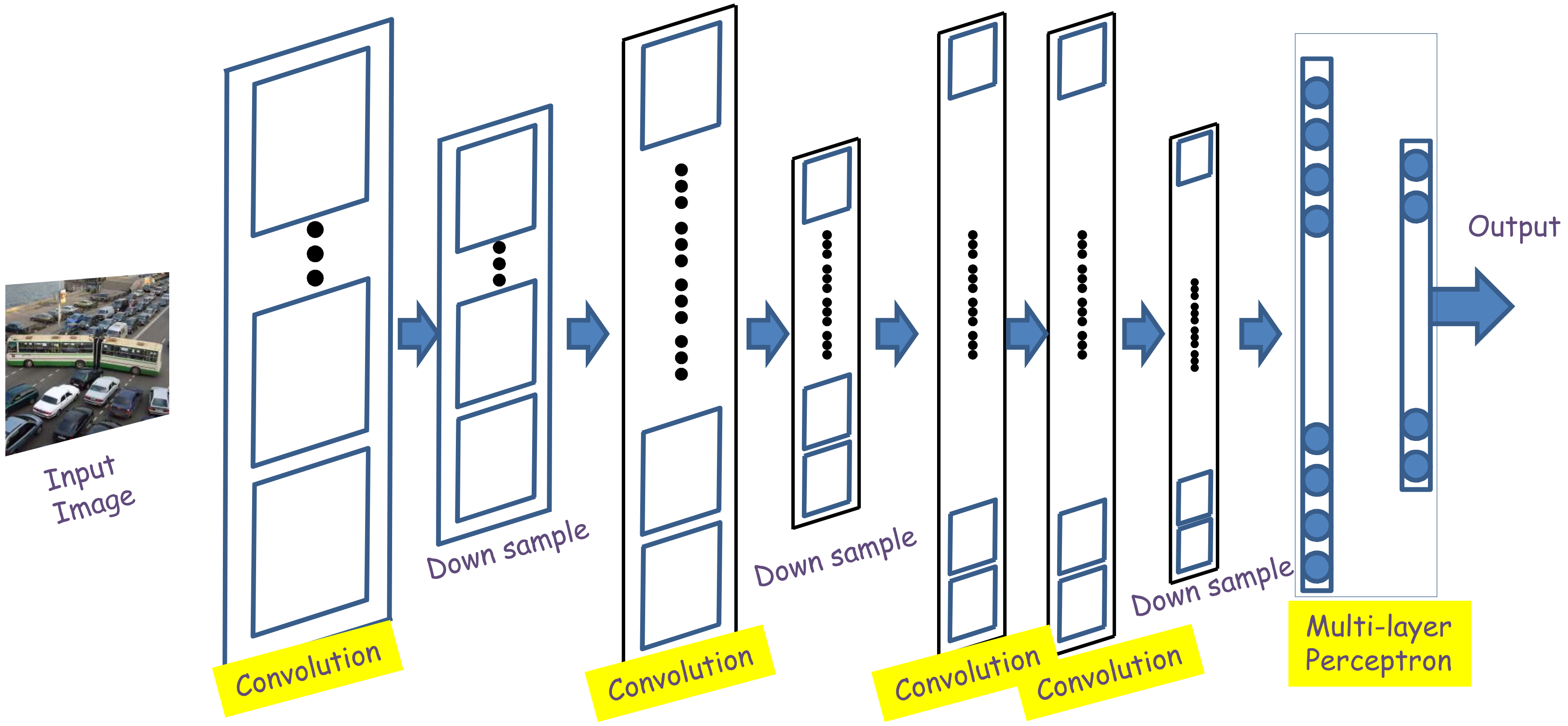
- A convolutional neural network comprises of “convolutional” and “down-sampling” layers
  - The two may occur in any sequence, but typically they alternate
- Followed by an MLP with one or more layers

# The general architecture of a convolutional neural network



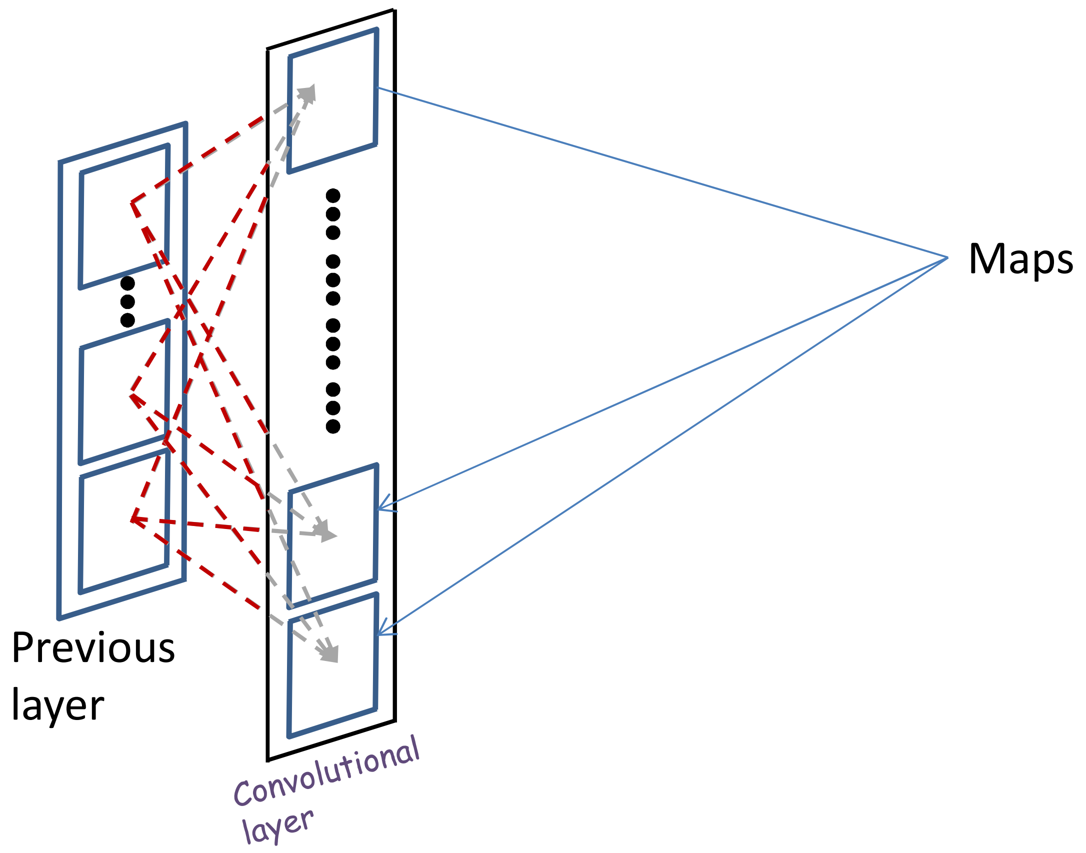
- A convolutional neural network comprises of “convolutional” and “downsampling” layers
  - The two may occur in any sequence, but typically they alternate
- Followed by an MLP with one or more layers

# The general architecture of a convolutional neural network



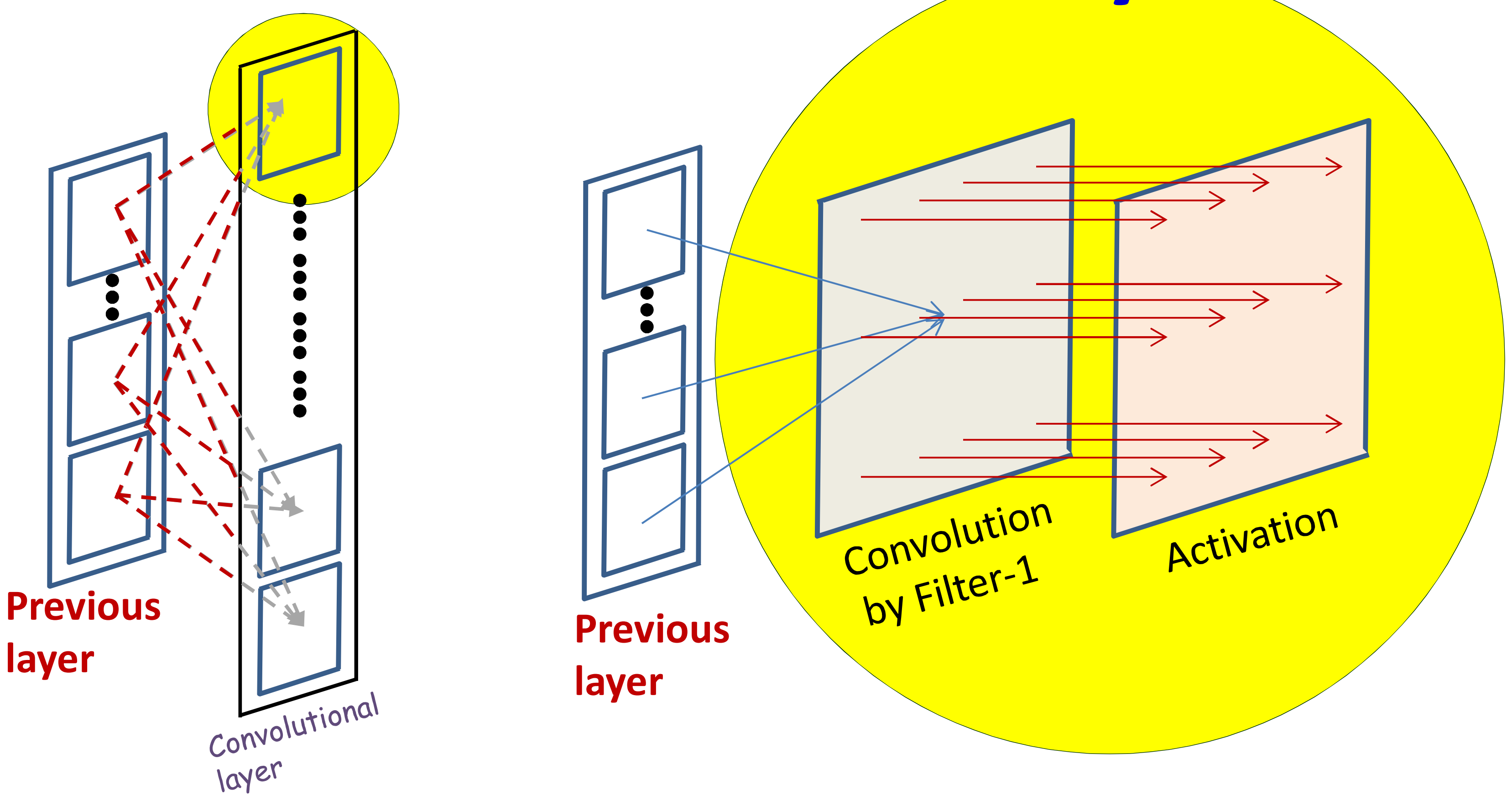
- **Convolutional layers and the MLP are *learnable***
  - Their parameters must be learned from training data for the target classification task
- Down-sampling layers are fixed and generally not learnable

# A convolutional layer



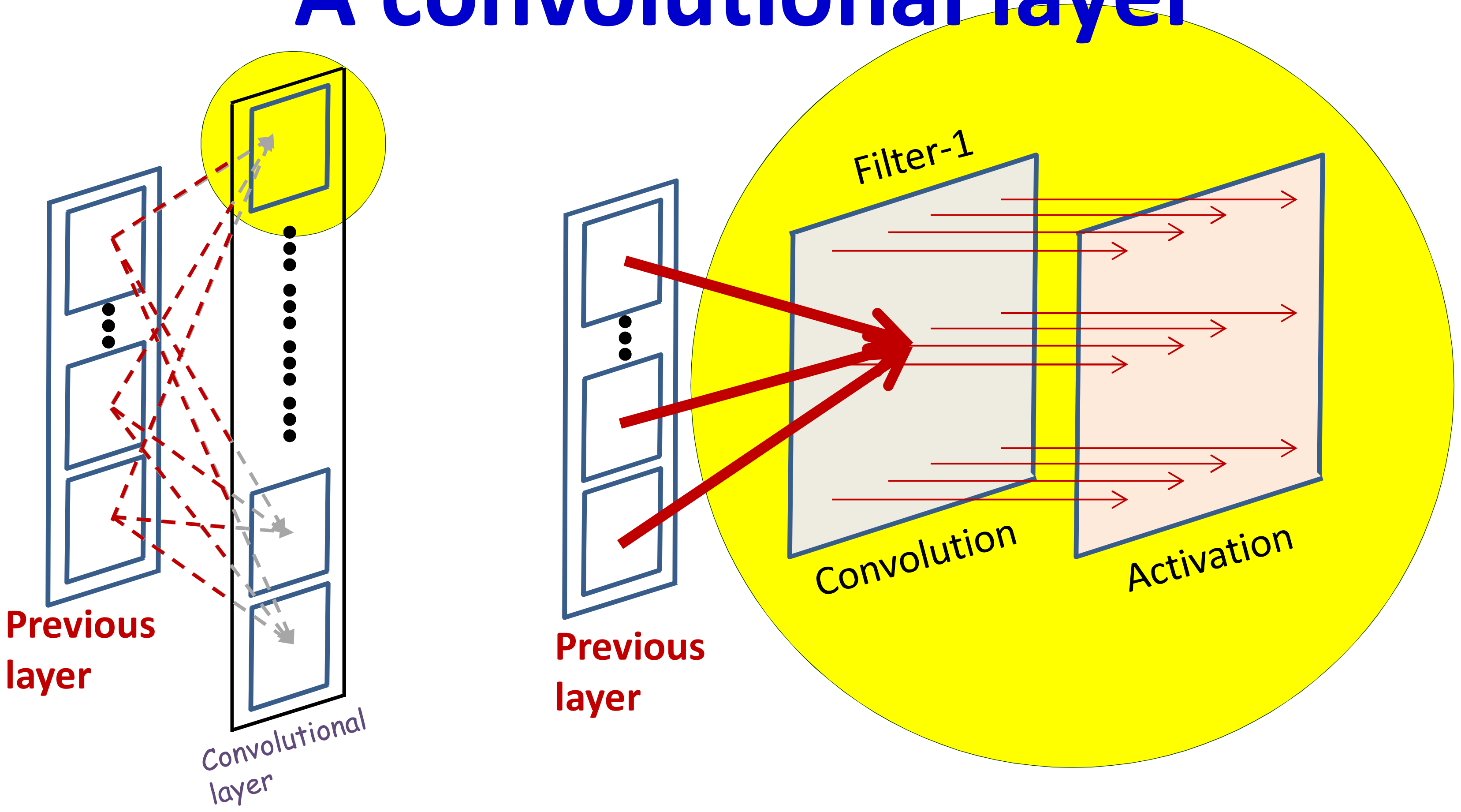
- A convolutional layer comprises of a series of “maps”
  - Corresponding the “S-planes” in the Neocognitron
  - Variously called feature maps or activation maps

# A convolutional layer



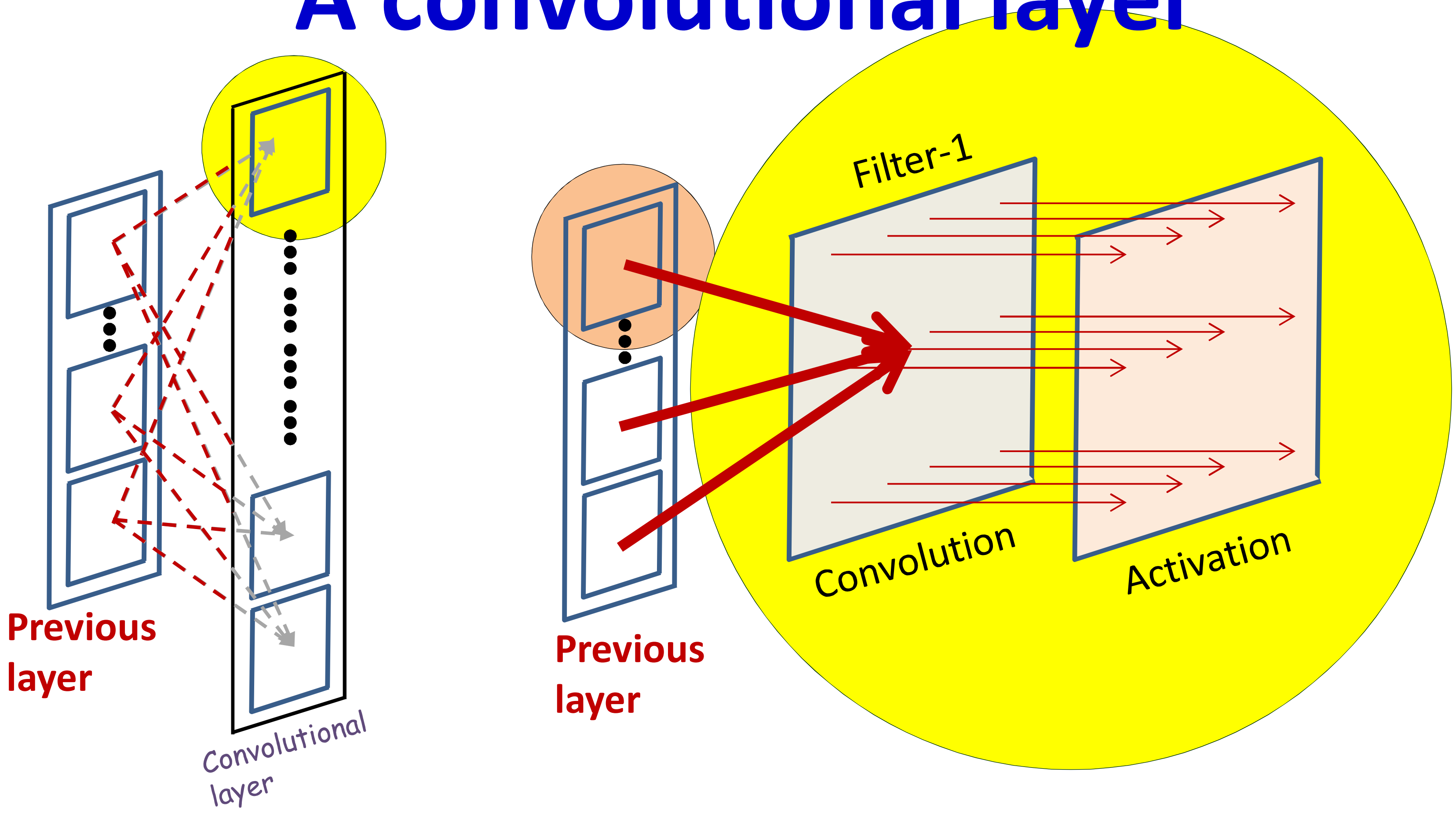
- Each activation map has two components
  - A *linear* map, obtained by *convolution* over maps in the previous layer
    - Each linear map has, associated with it, a ***learnable filter***
  - An *activation* that operates on the output of the convolution

# A convolutional layer



- All the maps in the previous layer contribute to each convolution

# A convolutional layer



- All the maps in the previous layer contribute to each convolution
  - Consider the contribution of a *single* map

# What is a convolution

Example 5x5 image with binary pixels

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |

Example 3x3 filter

| | | |
|---|---|---|
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |

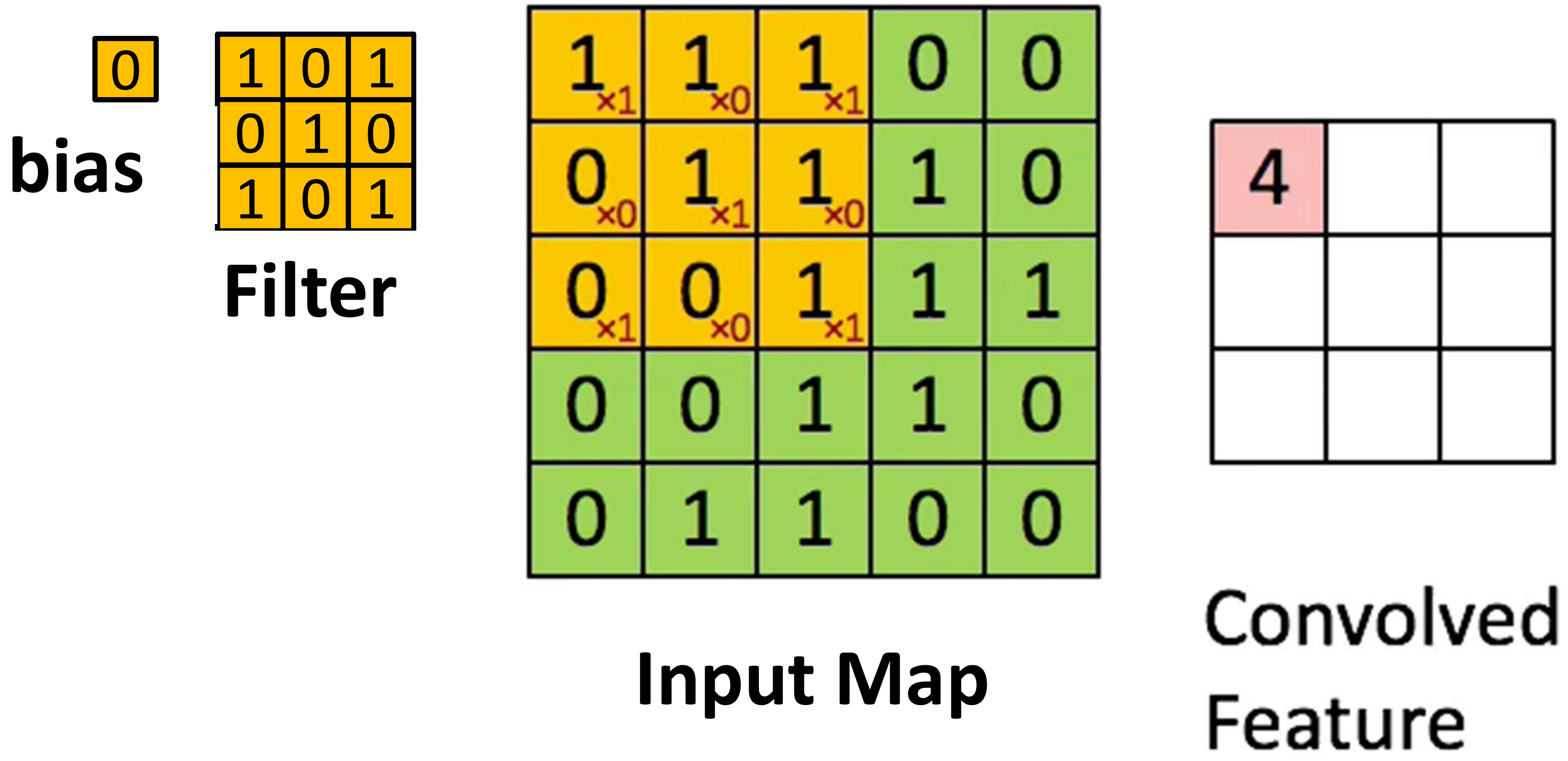
bias

| |
|---|
| 0 |

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

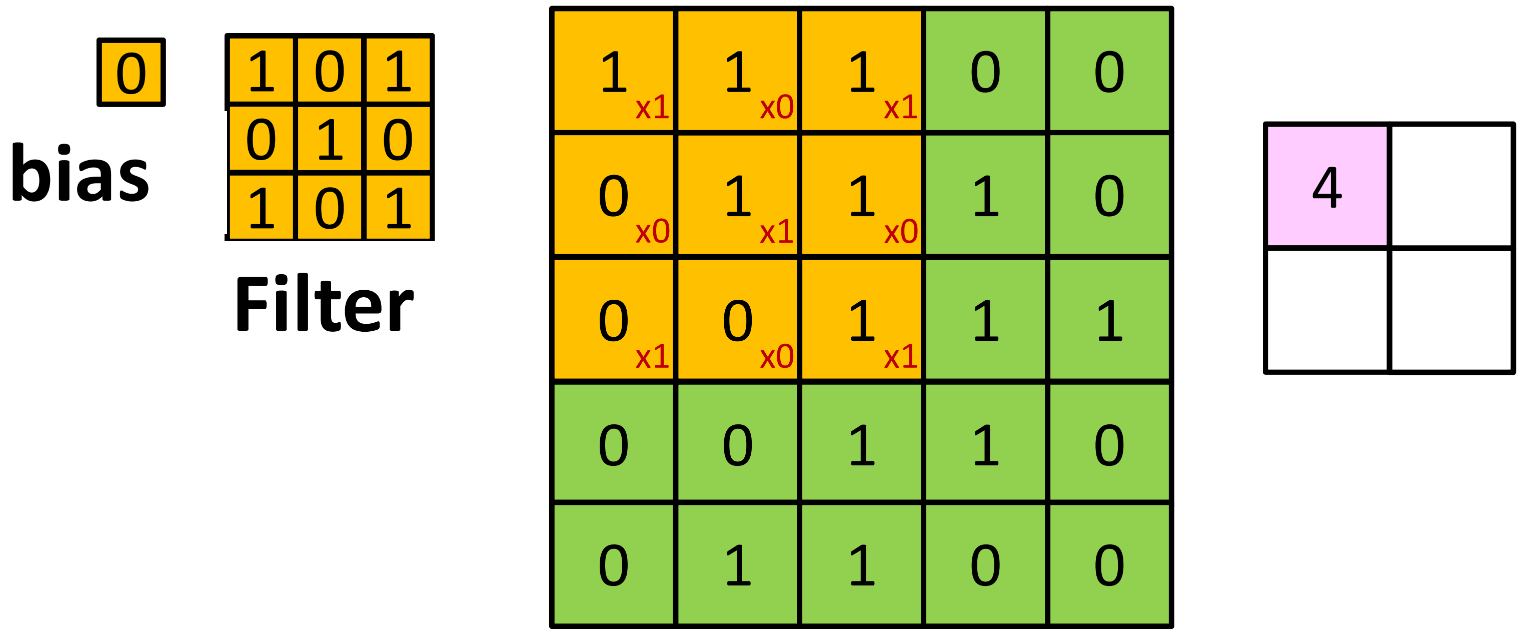
- Scanning an image with a “filter”
  - Note: a filter is really just a perceptron, with weights and a bias

## What is a convolution



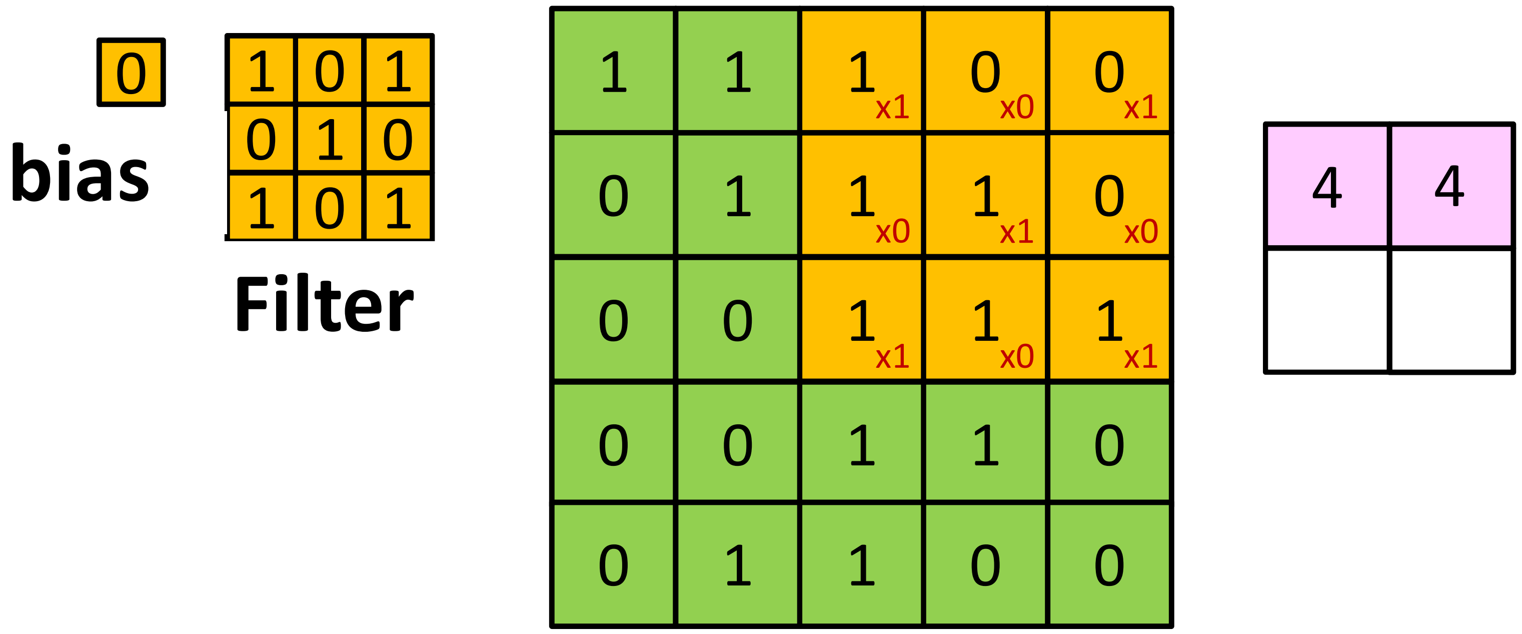
- Scanning an image with a “filter”
  - At each location, the “filter and the underlying map values are multiplied component wise, and the products are added along with the bias

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



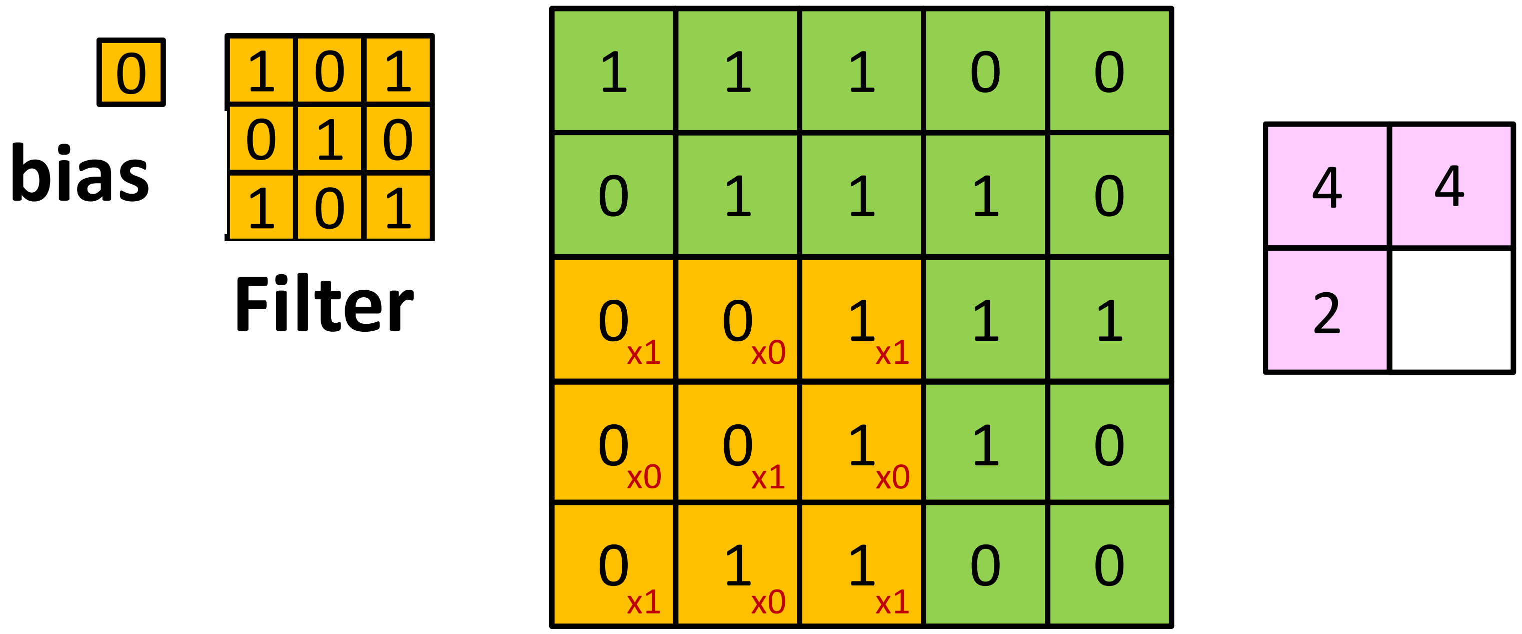
- Scanning an image with a "filter"
  - The filter may proceed by *more* than 1 pixel at a time
  - E.g. with a "stride" of *two* pixels per shift

## The “Stride” between adjacent scanned locations need not be 1



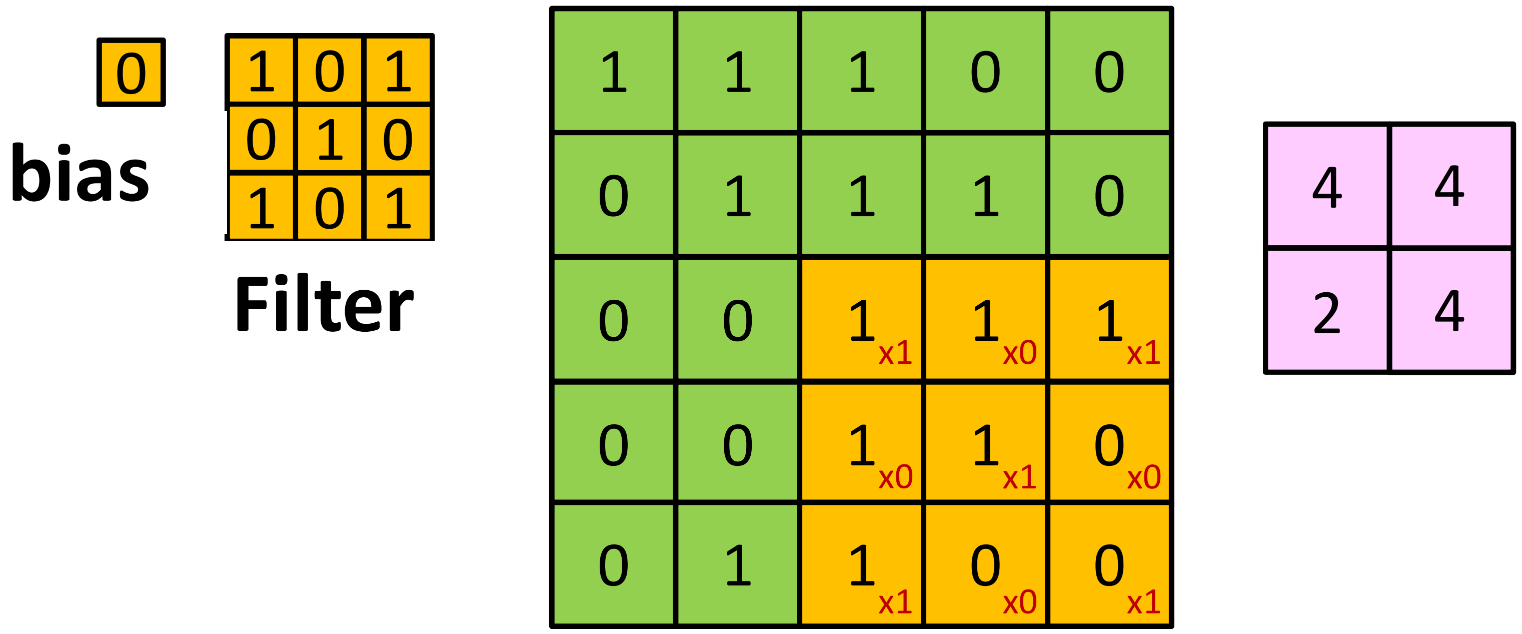
- Scanning an image with a “filter”
  - The filter may proceed by *more* than 1 pixel at a time
  - E.g. with a “hop” of *two* pixels per shift

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



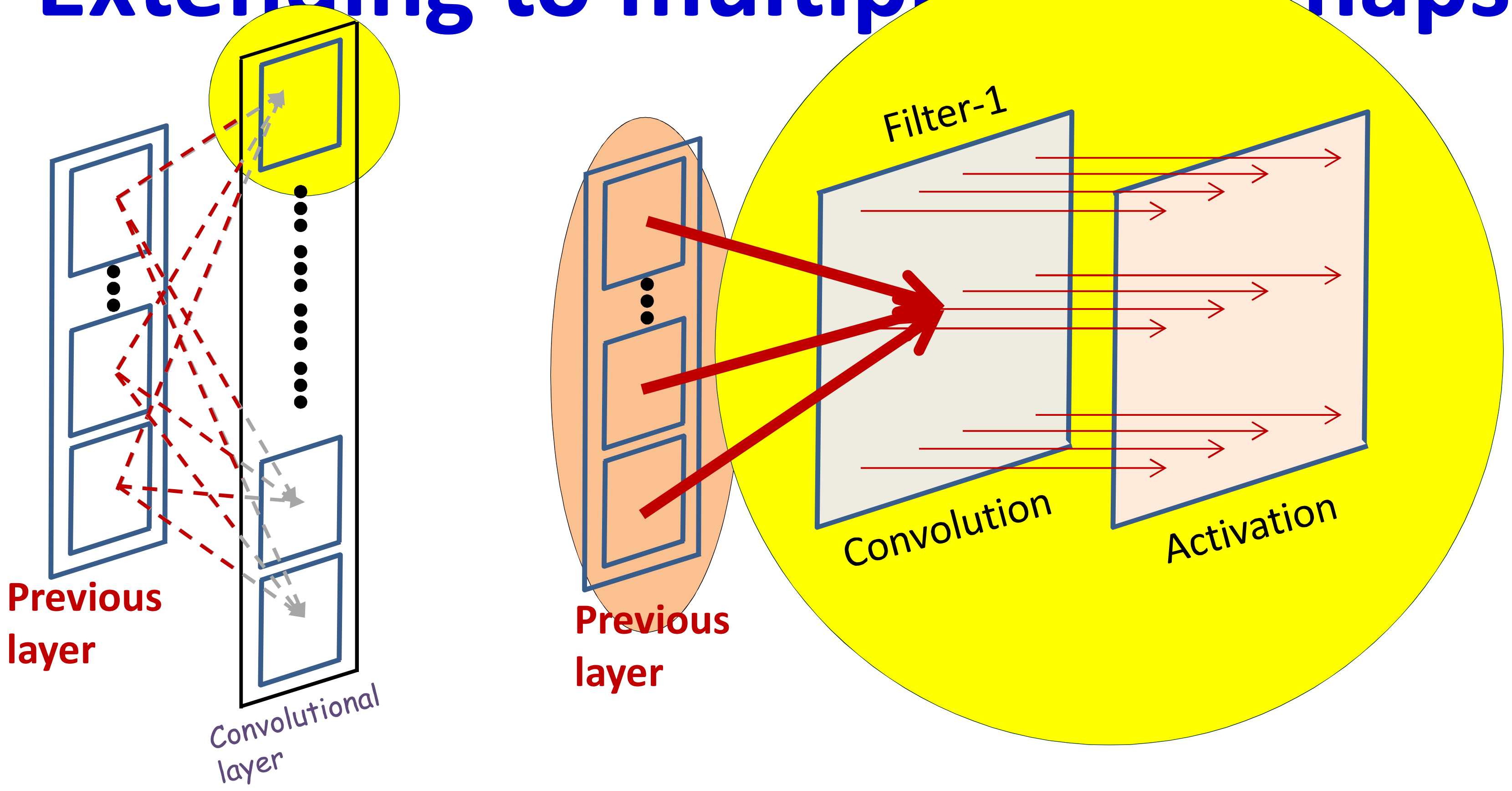
- Scanning an image with a "filter"
  - The filter may proceed by *more* than 1 pixel at a time
  - E.g. with a "hop" of *two* pixels per shift

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



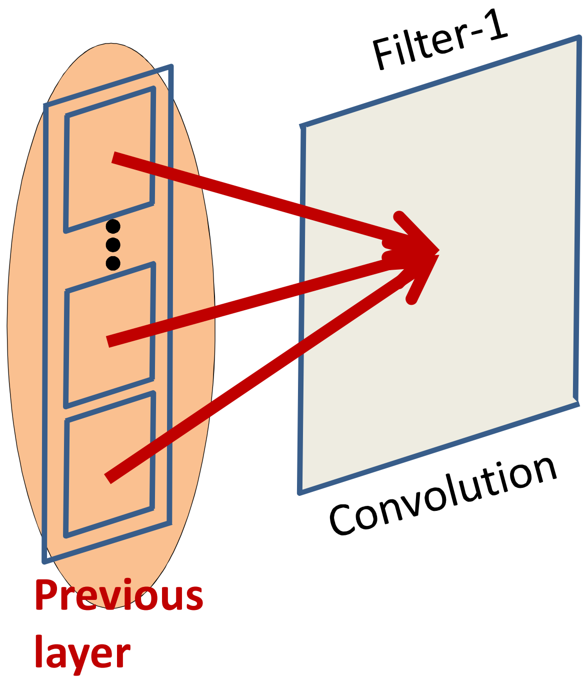
- Scanning an image with a "filter"
  - The filter may proceed by *more* than 1 pixel at a time
  - E.g. with a "hop" of *two* pixels per shift

## Extending to multiple input maps



- We actually compute any individual convolutional map from *all* the maps in the previous layer

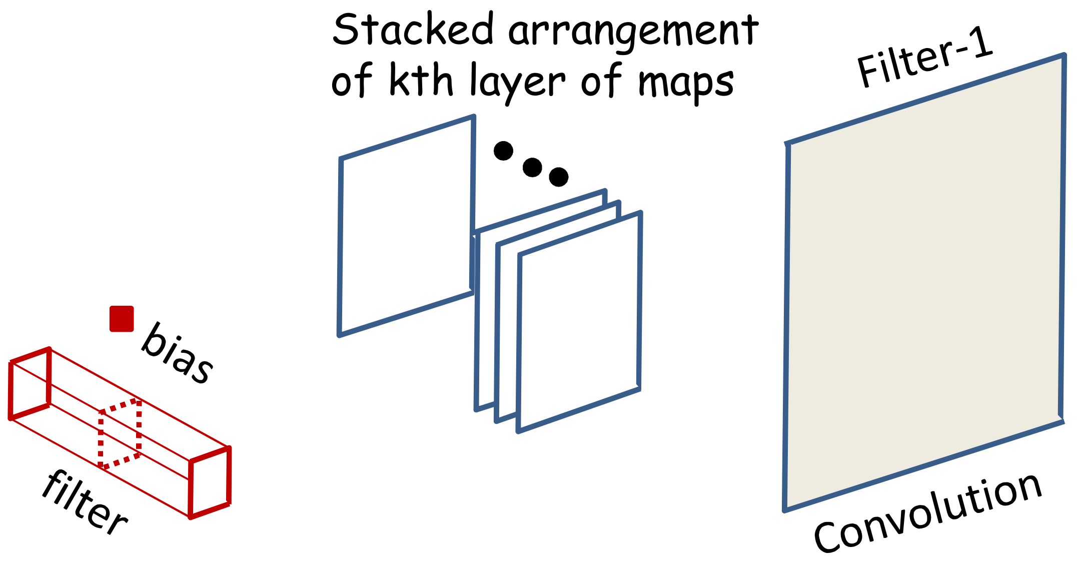
## Extending to multiple input maps



- We actually compute any individual convolutional map from *all* the maps in the previous layer

- The actual processing is better understood if we modify our visualization of all the maps in a layer as vertical arrangement to..

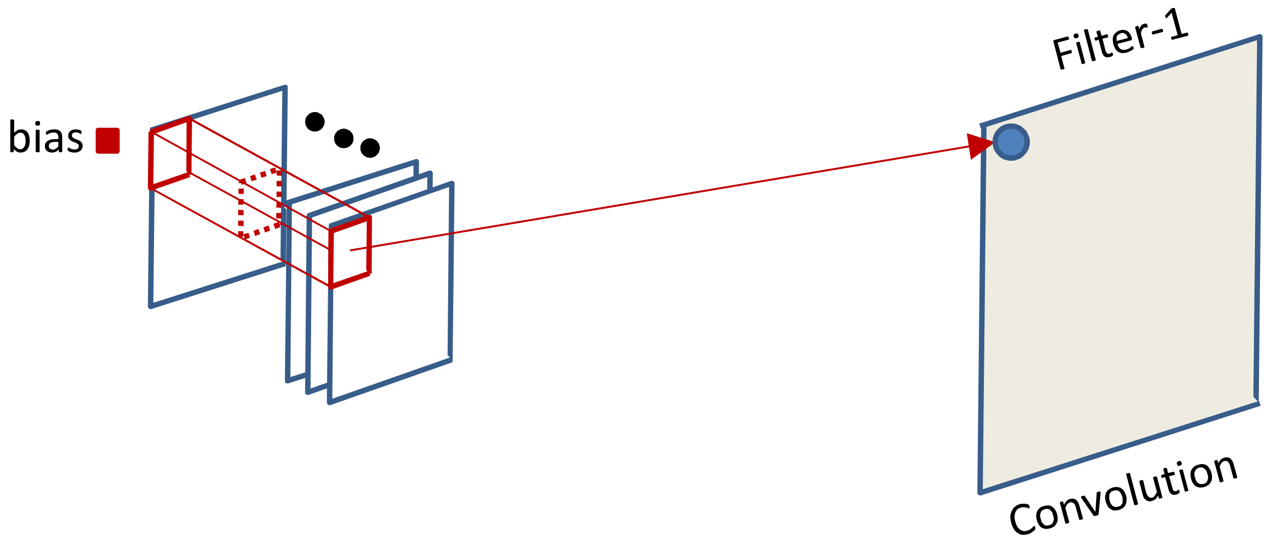
## Extending to multiple input maps



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

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

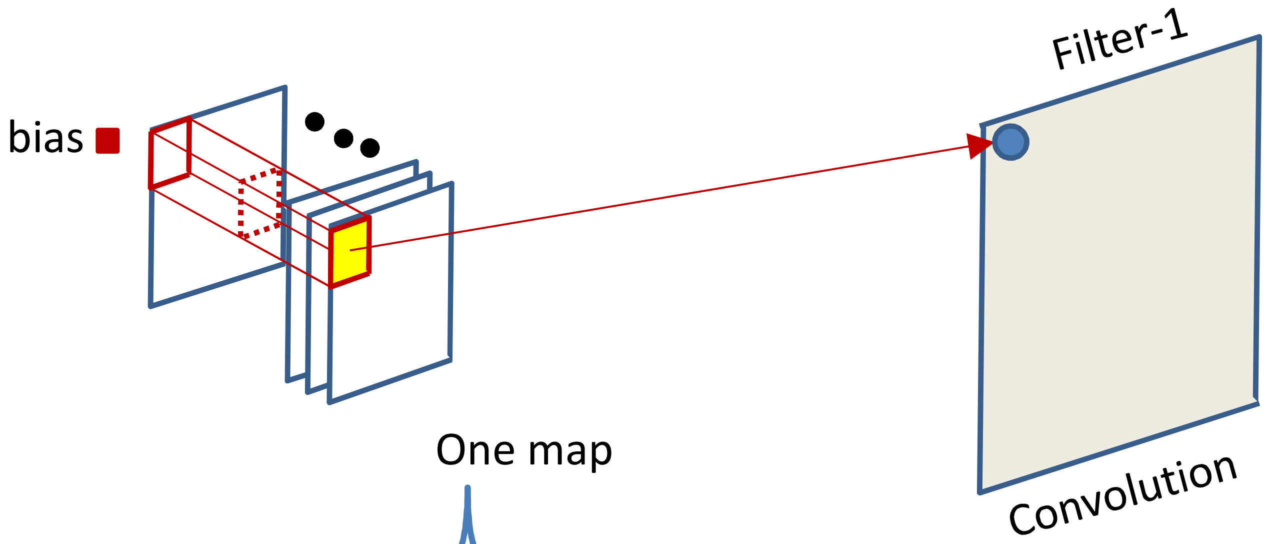
## Extending to multiple input maps



$$z(i,j) = \sum_{p} \sum_{k=1}^{L} \sum_{l=1}^{L} w_{S,l,n}(p,k,l) Y_p(i+l, j+k) + b$$

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

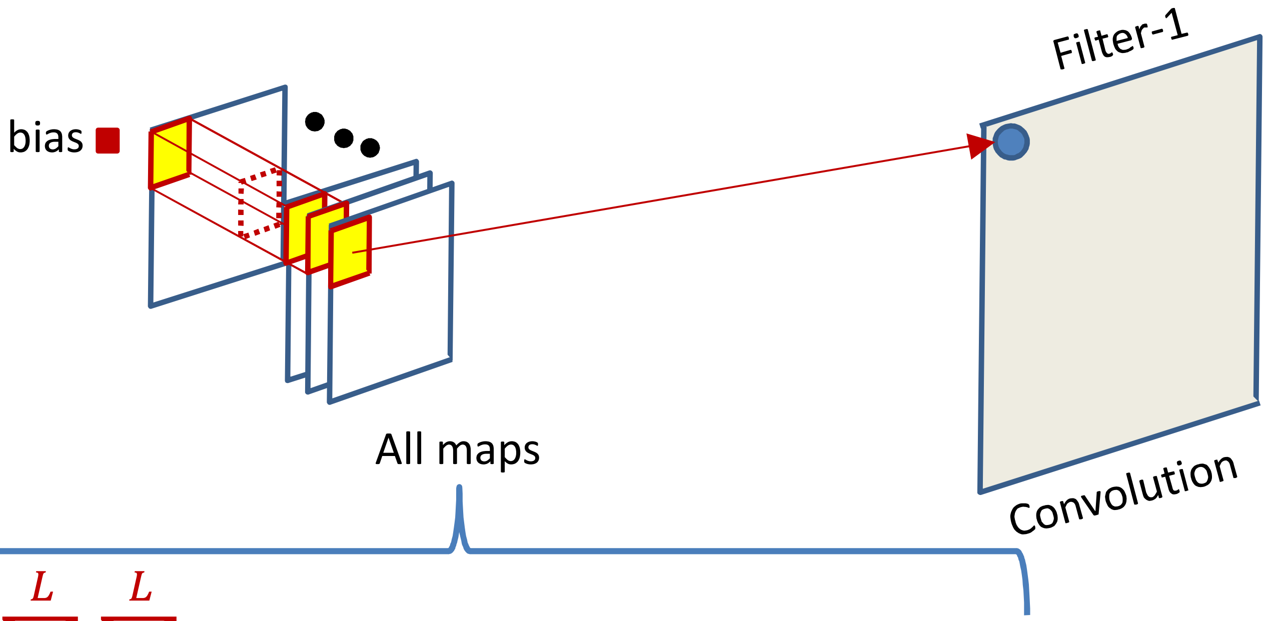
# Extending to multiple input maps



$$z(i,j) = \sum_{p} \overbrace{\sum_{k=1}^{L} \sum_{l=1}^{L} w_{S,l,n}(p,k,l) Y_p(i+l, j+k)}^{\text{One map}} + b$$

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

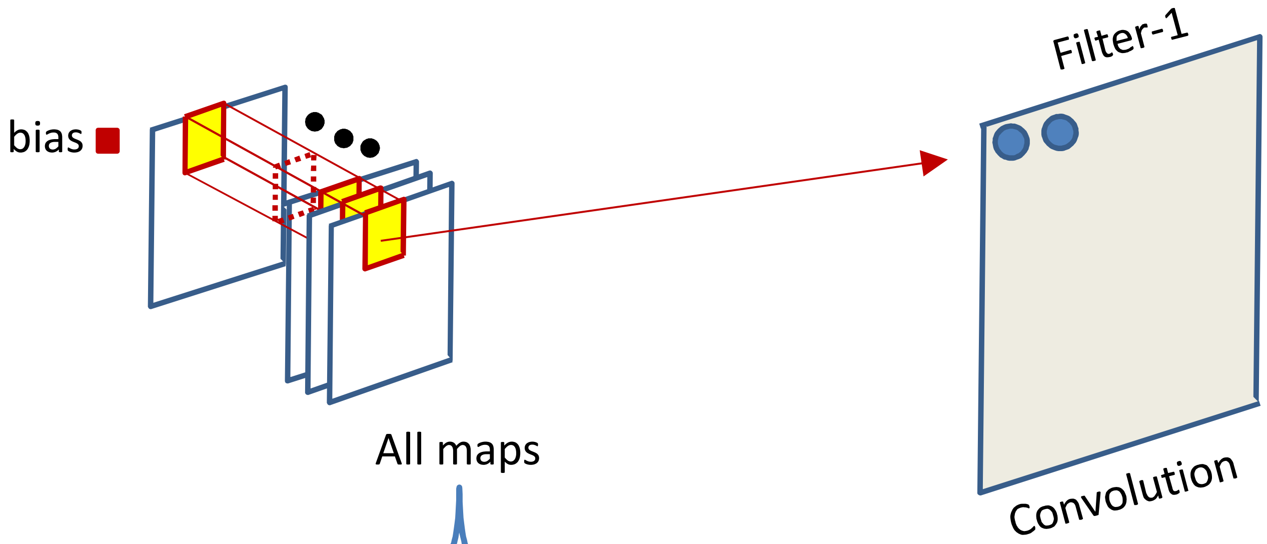
# Extending to multiple input maps



$$z(i,j) = \sum_{p} \sum_{k=1}^{L} \sum_{l=1}^{L} w_{S,l,n}(p,k,l) Y_p(i+l, j+k) + b$$

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

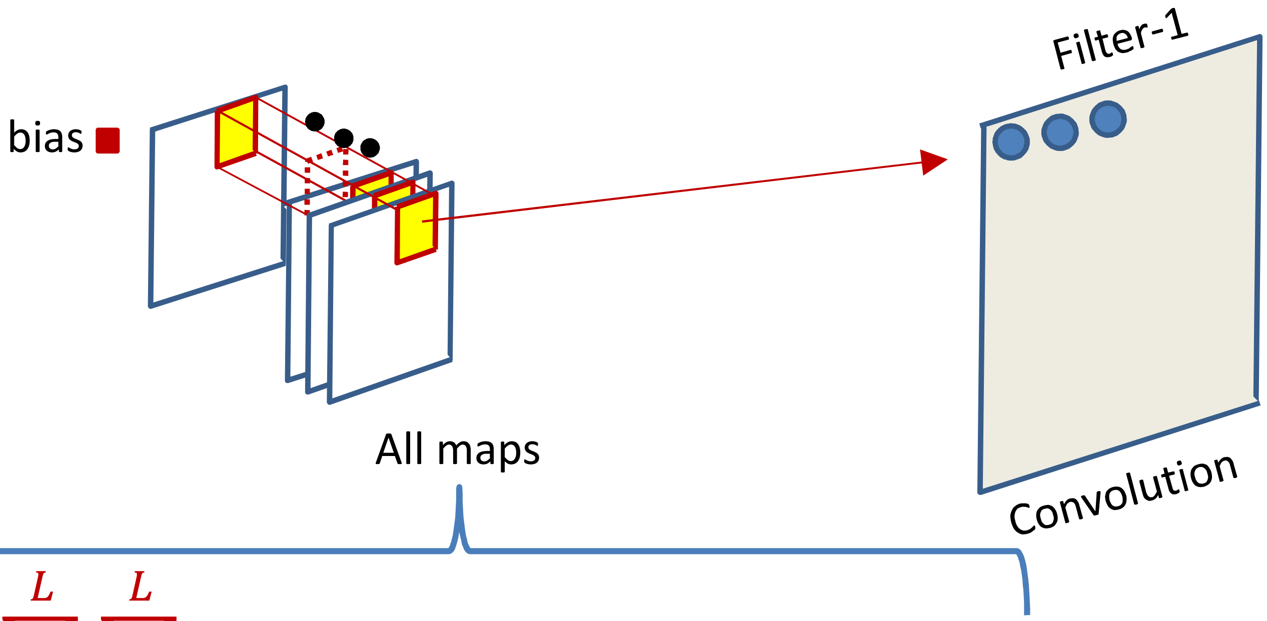
# Extending to multiple input maps



$$z(i,j) = \overbrace{\sum_{p} \sum_{k=1}^{L} \sum_{l=1}^{L} w_{S,l,n}(p,k,l) Y_p(i+l, j+k) + b}^{\text{All maps}}$$

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

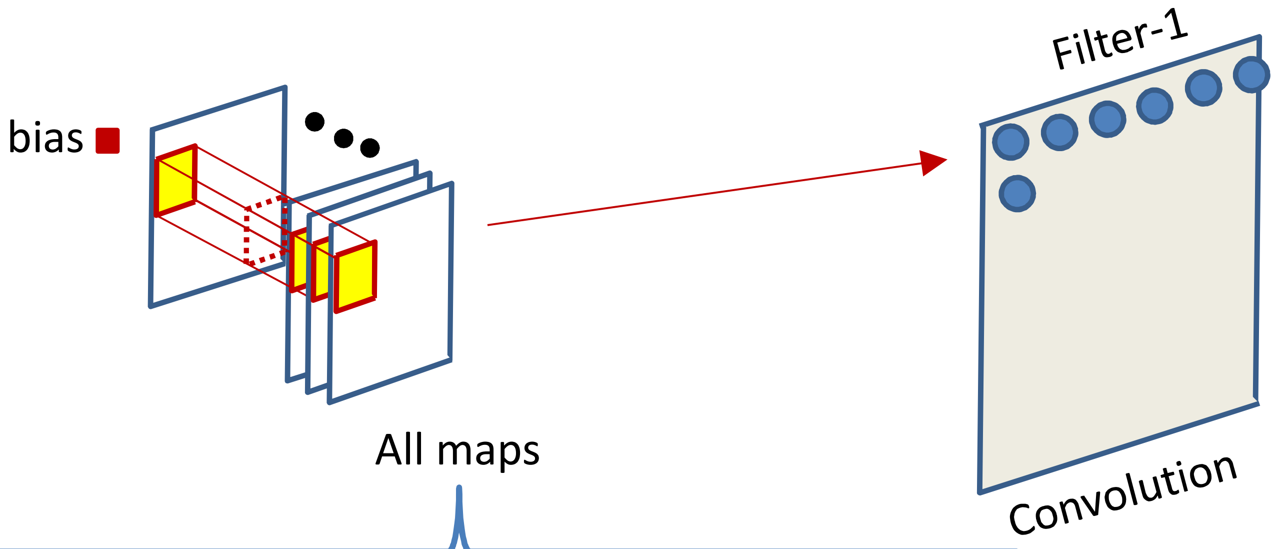
# Extending to multiple input maps



$$z(i,j) = \sum_{p} \sum_{k=1}^{L} \sum_{l=1}^{L} w_{S,l,n}(p,k,l) Y_p(i+l, j+k) + b$$

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

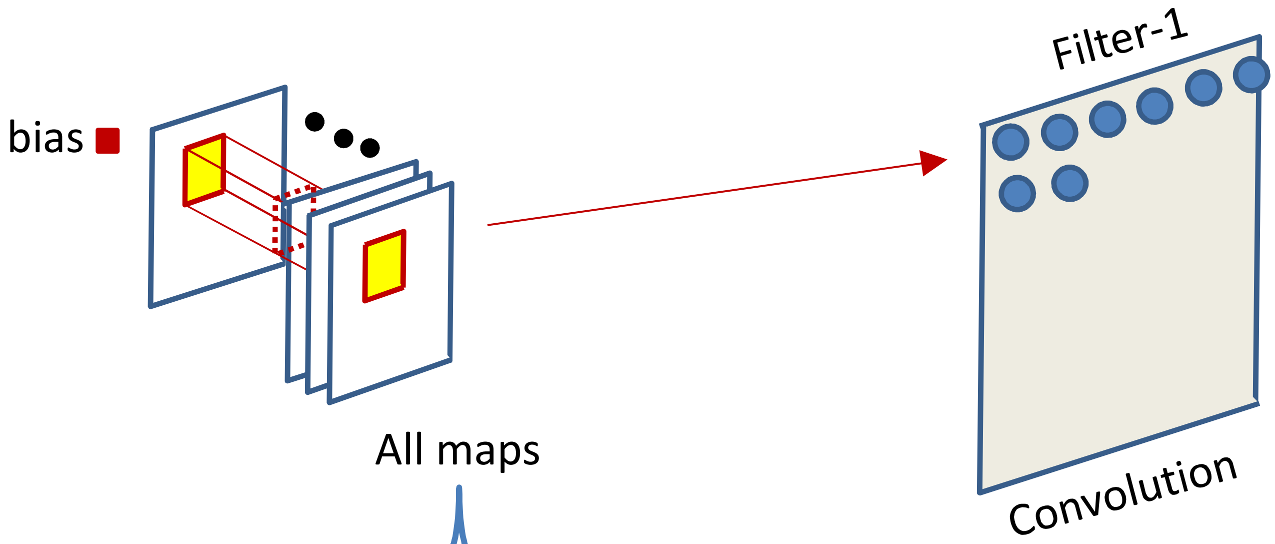
# Extending to multiple input maps



$$z(i,j) = \underbrace{\sum_{p}\sum_{k=1}^{L}\sum_{l=1}^{L} w_{S,l,n}(p,k,l)Y_p(i+l,j+k)}_{\text{All maps}} + b$$

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

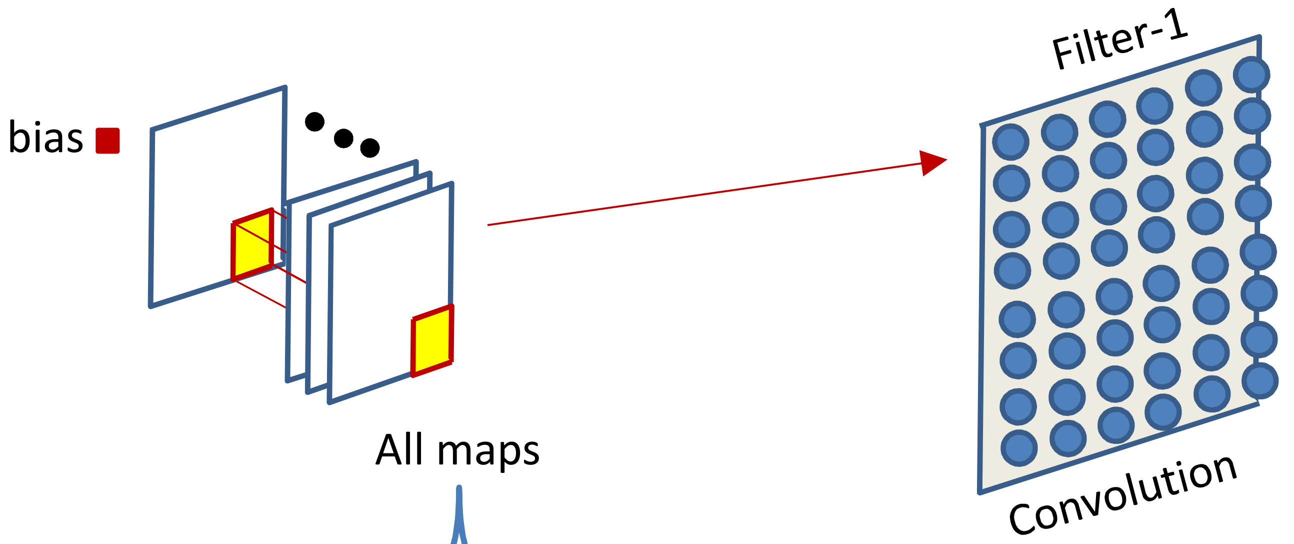
## Extending to multiple input maps



$$z(i,j) = \overbrace{\sum_{p} \sum_{k=1}^{L} \sum_{l=1}^{L} w_{S,l,n}(p,k,l) Y_p(i+l, j+k) + b}^{\text{All maps}}$$

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

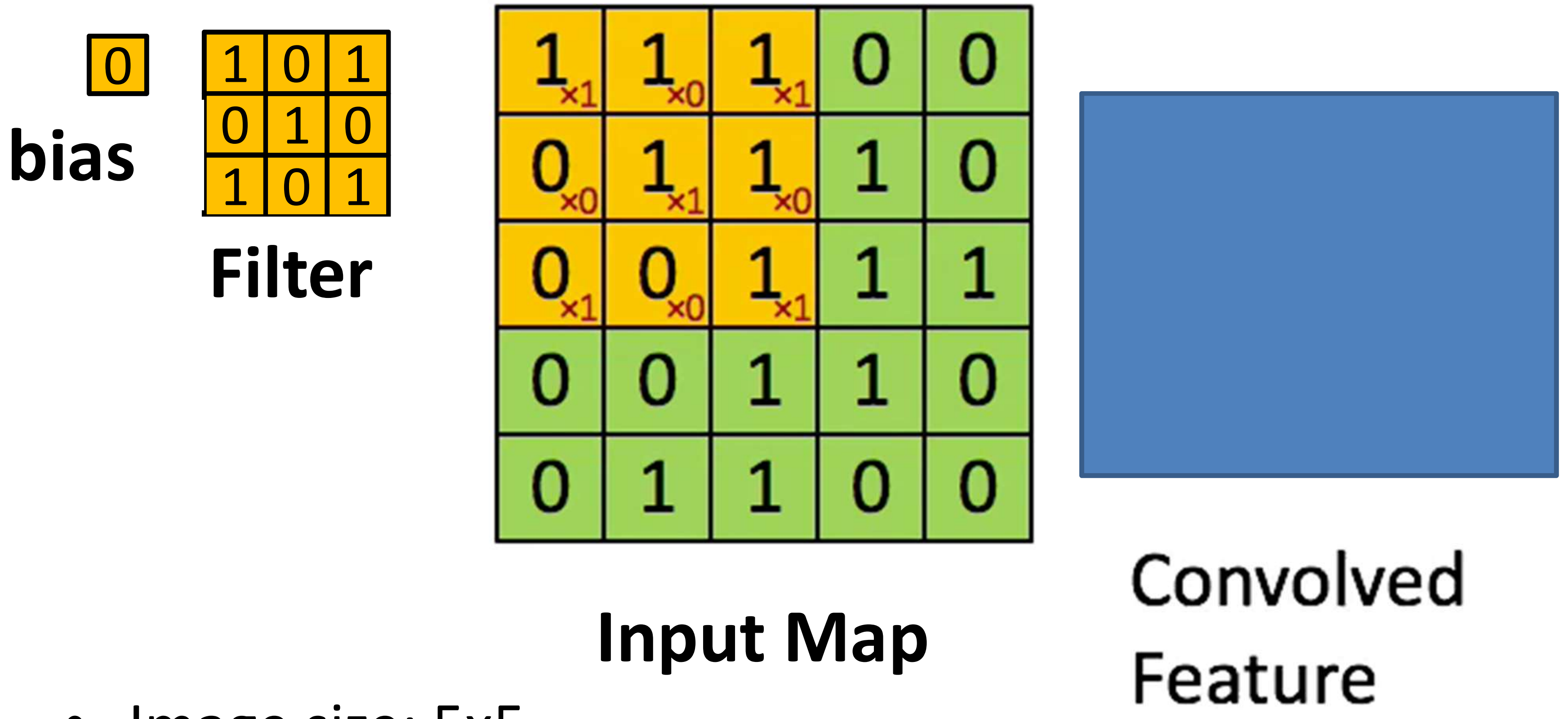
# Extending to multiple input maps



$$z(i,j) = \overbrace{\sum_{p} \sum_{k=1}^{L} \sum_{l=1}^{L} w_{S,l,n}(p,k,l) Y_p(i+l, j+k) + b}^{\text{All maps}}$$

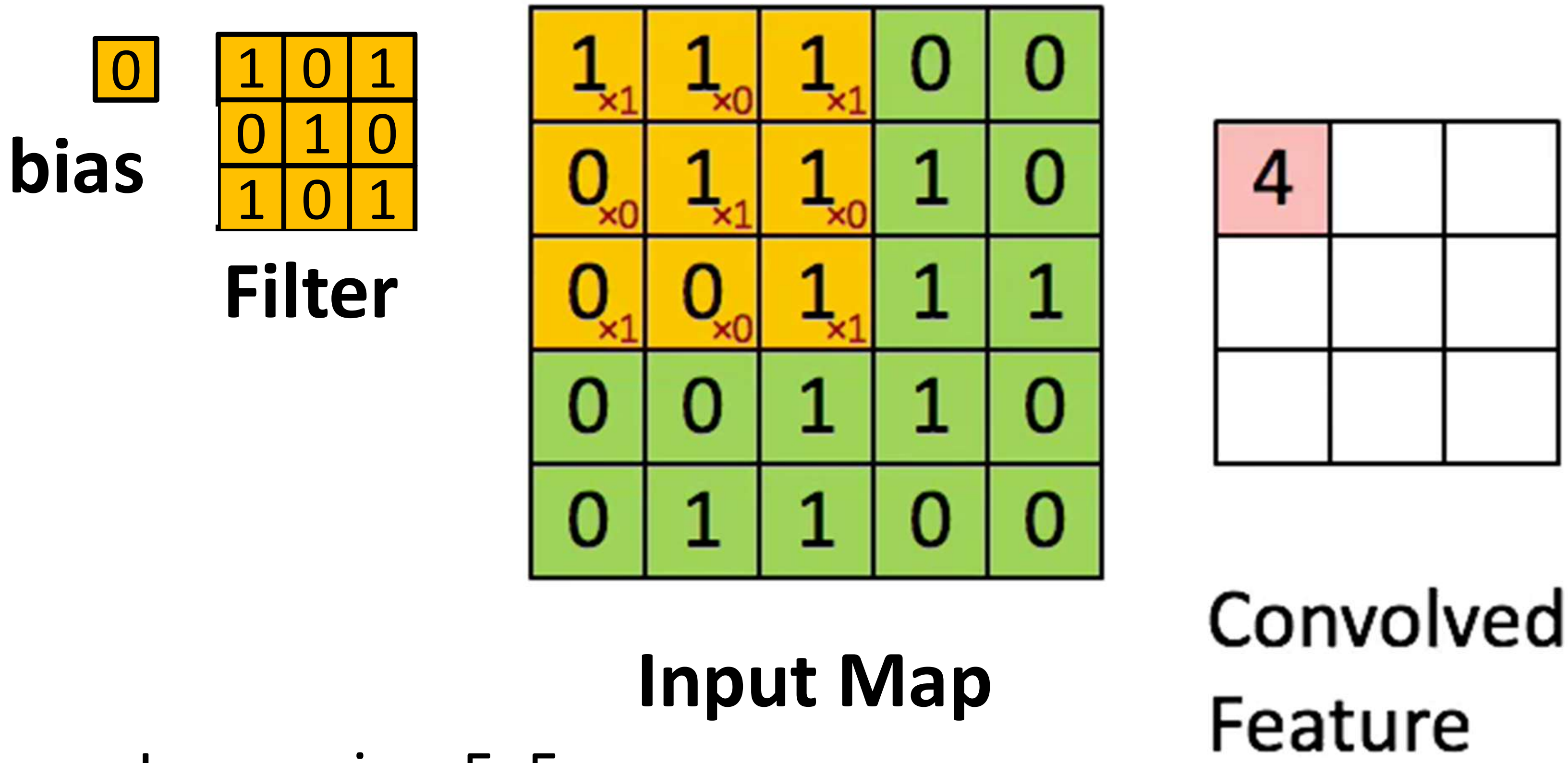
- The computation of the convolutive map at any location *sums* the convolutive outputs *at all planes*

## The size of the convolution



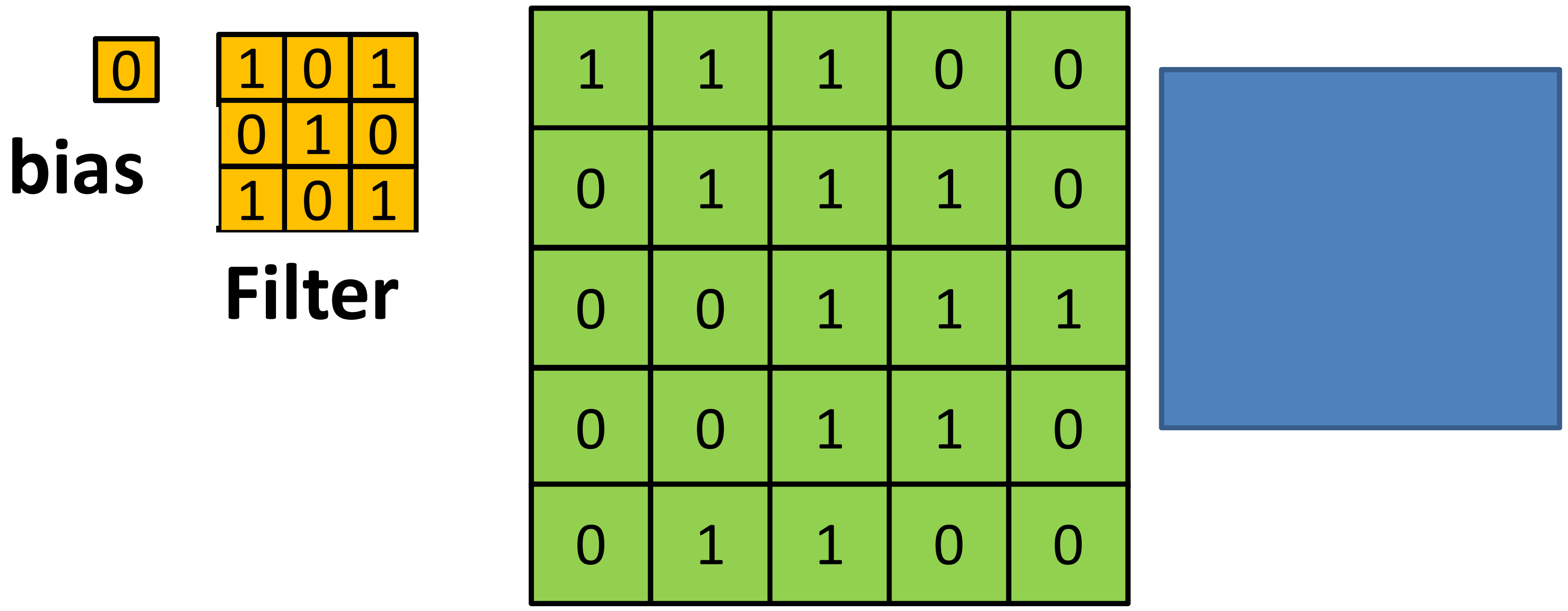
- Image size: 5x5
- Filter: 3x3
- “Stride”: 1
- Output size = ?

## The size of the convolution



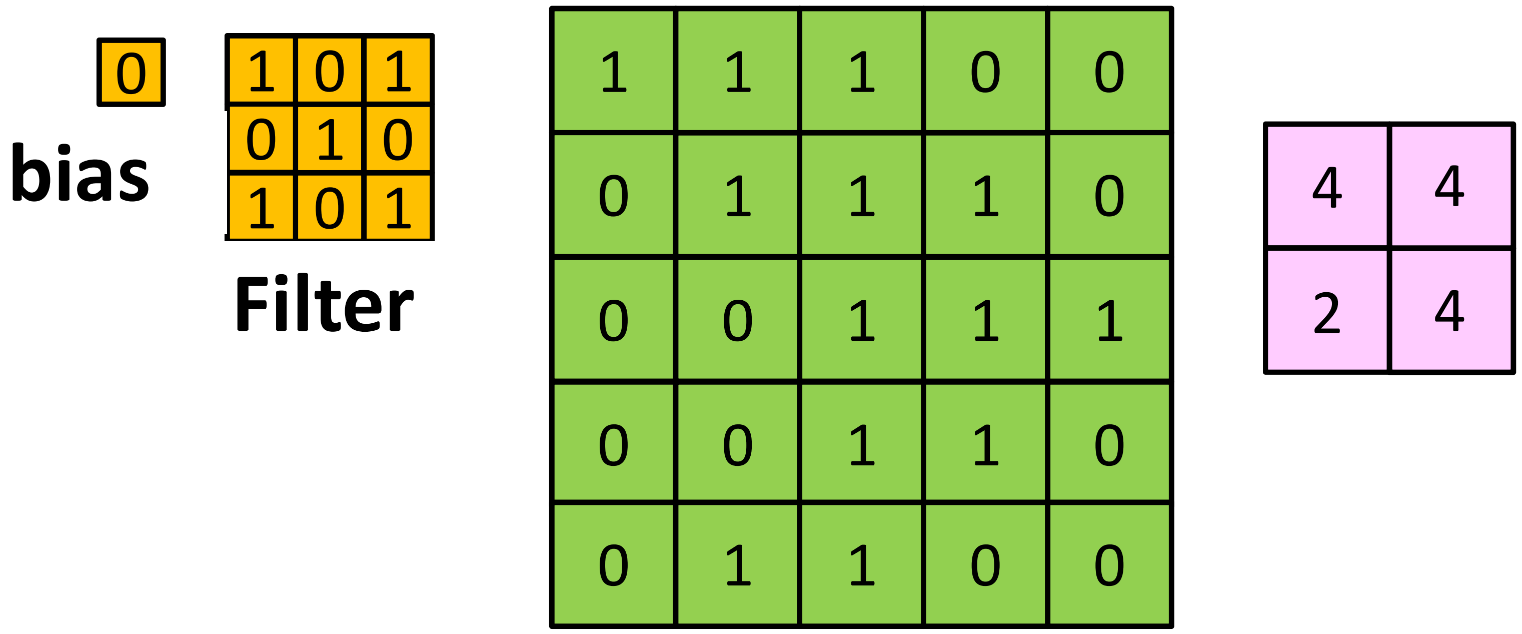
- Image size: 5x5
- Filter: 3x3
- Stride: 1
- Output size = ?

# The size of the convolution



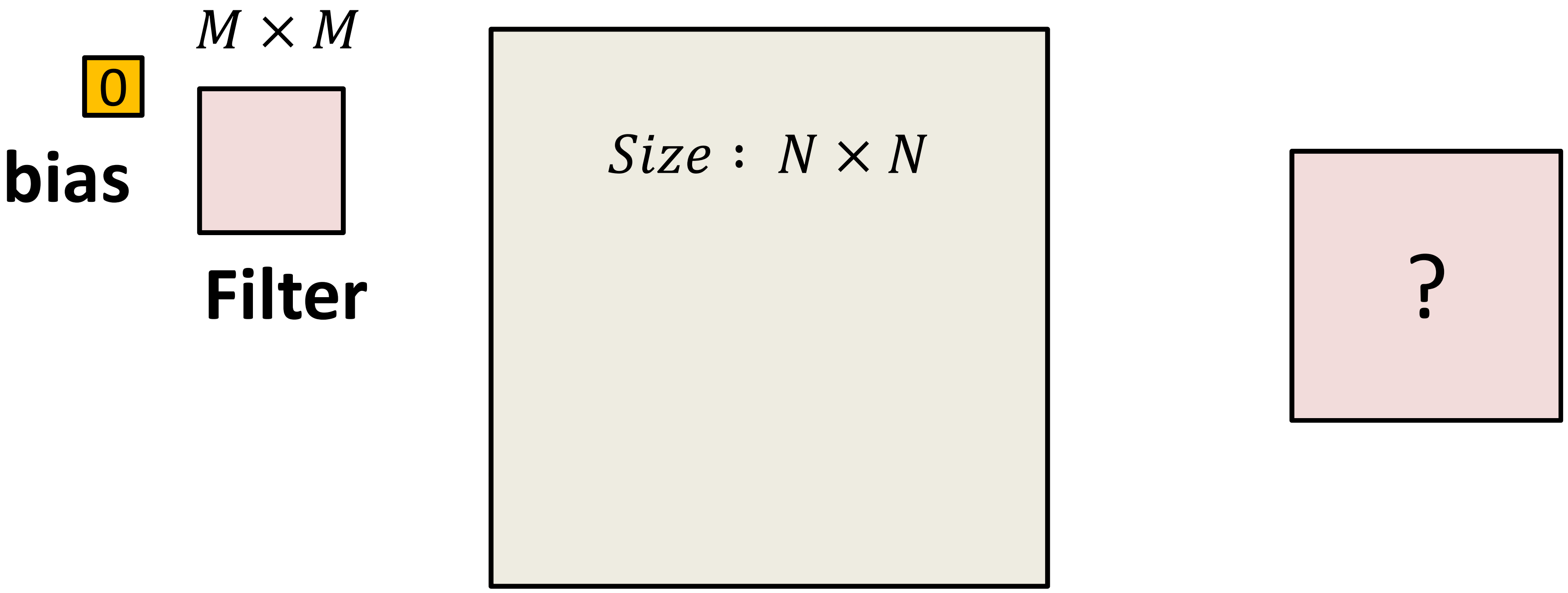
- Image size: 5x5
- Filter: 3x3
- Stride: 2
- Output size = ?

# The size of the convolution



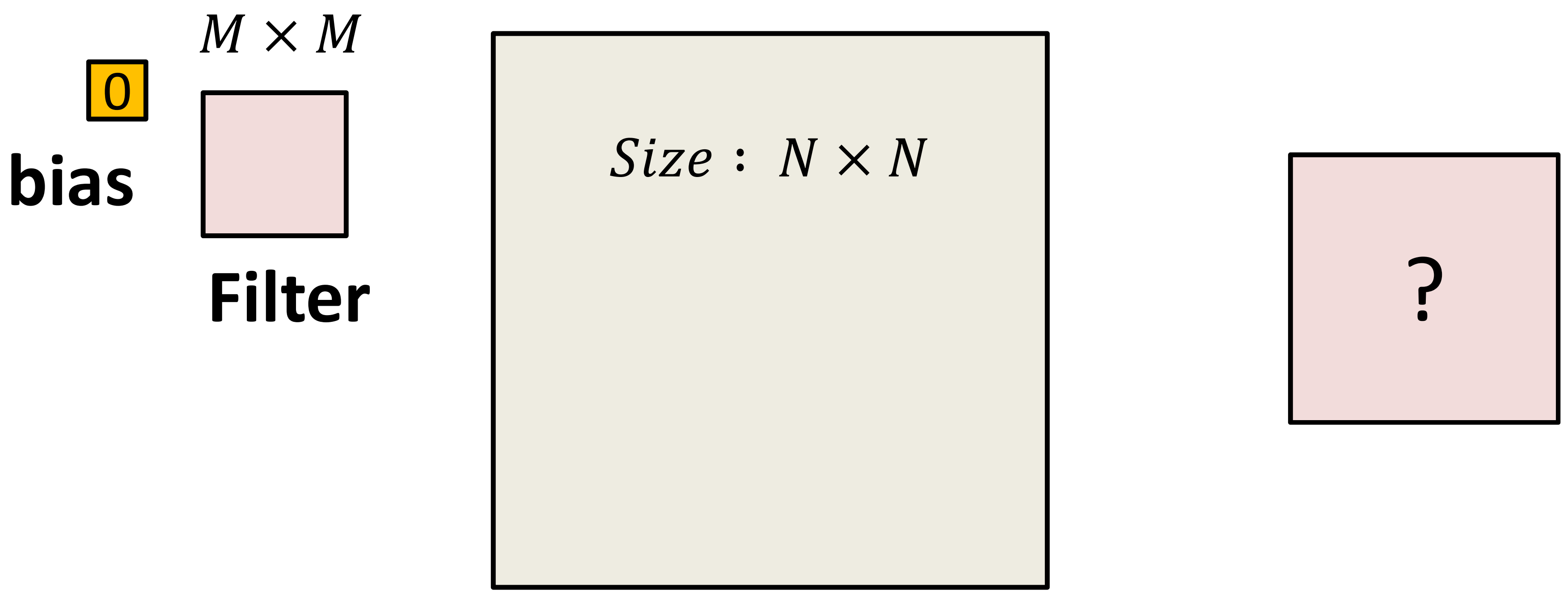
- Image size: 5x5
- Filter: 3x3
- Stride: 2
- Output size = ?

## The size of the convolution



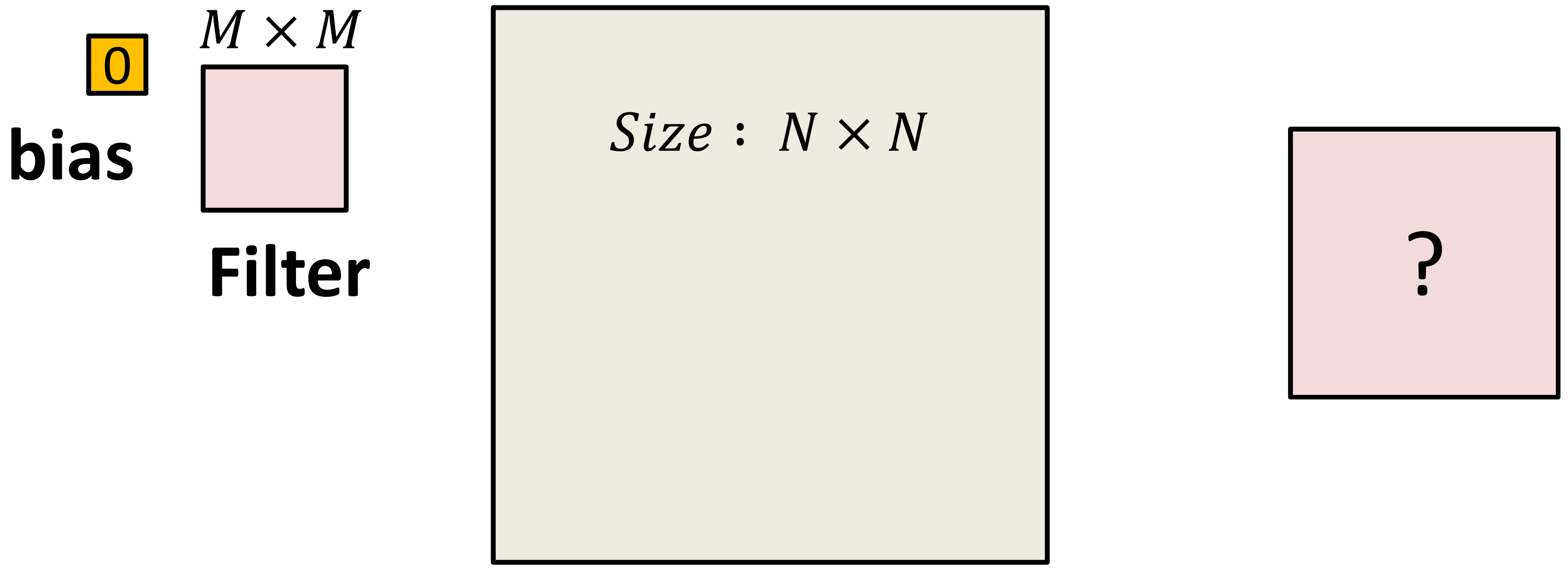
- Image size: $N \times N$
- Filter: $M \times M$
- Stride: 1
- Output size = ?

## The size of the convolution



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

# The size of the convolution



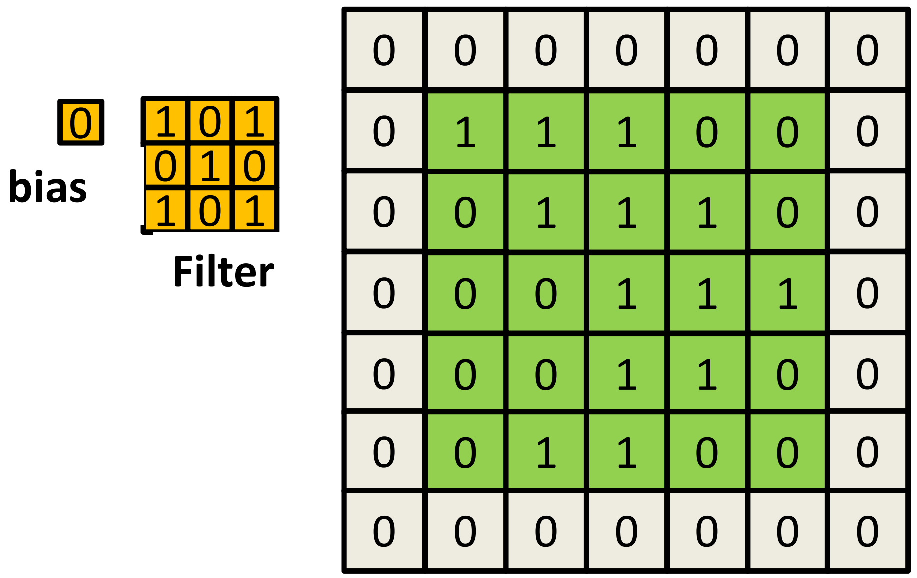
- Image size: $N \times N$
- Filter: $M \times M$
- Stride: $S$
- Output size (each side) = $\lfloor (N - M)/S \rfloor + 1$
  - Assuming you're not allowed to go beyond the edge of the input

## Convolution Size

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

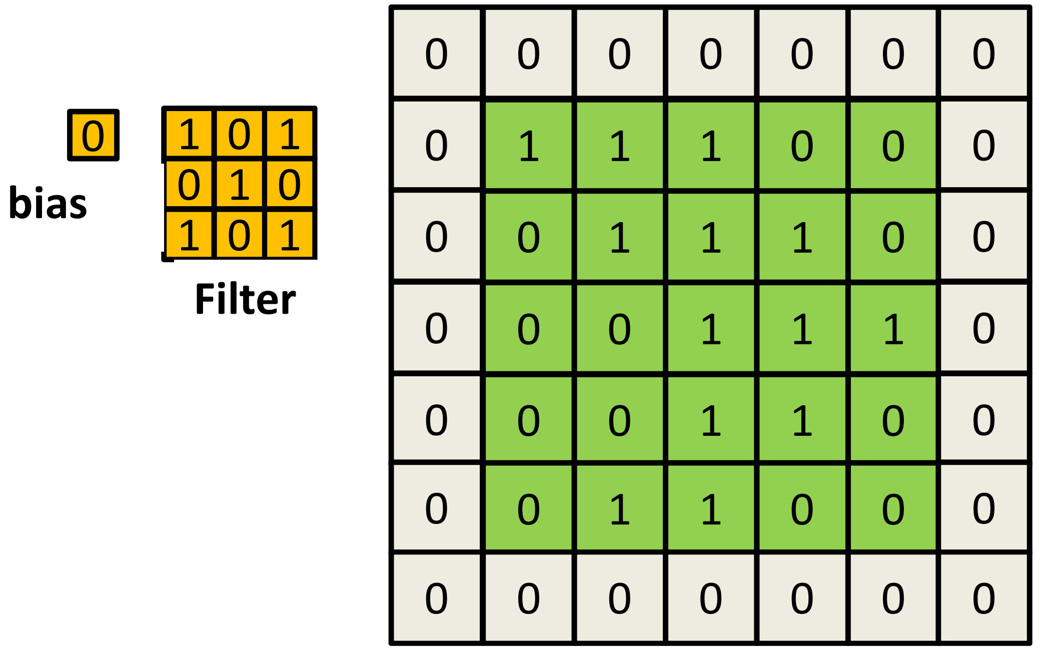
- Results in a reduction in the output size
  - Even if $S = 1$
  - Sometimes not considered acceptable
    - If there's no active downsampling, through max pooling and/or $S > 1$, then the output map should ideally be the same size as the input

# Solution



- Zero-pad the input
  - Pad the input image/map all around
    - Add $P_L$ rows of zeros on the left and $P_R$ rows of zeros on the right
    - Add $P_L$ rows of zeros on the top and $P_L$ rows of zeros at the bottom
  - $P_L$ and $P_R$ chosen such that:
    - $P_L = P_R$ OR $|P_L - P_R| = 1$
    - $P_L + P_R = M-1$
      - **For stride 1, the result of the convolution is the same size as the original image**

# Solution



| |
|---|
| 0 |

| | | |
|---|---|---|
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

- Zero-pad the input
  - Pad the input image/map all around
  - Pad as symmetrically as possible, such that..
  - **For stride 1, the result of the convolution is the same size as the original image**

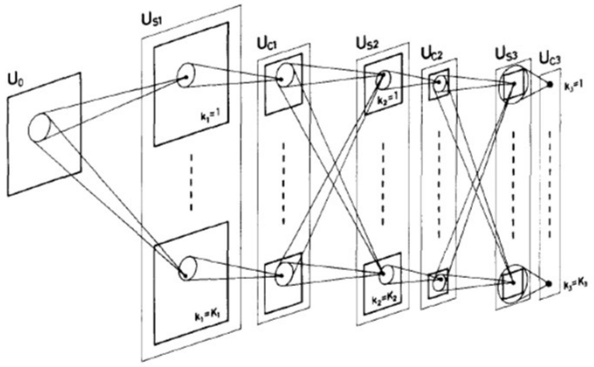
# Zero padding

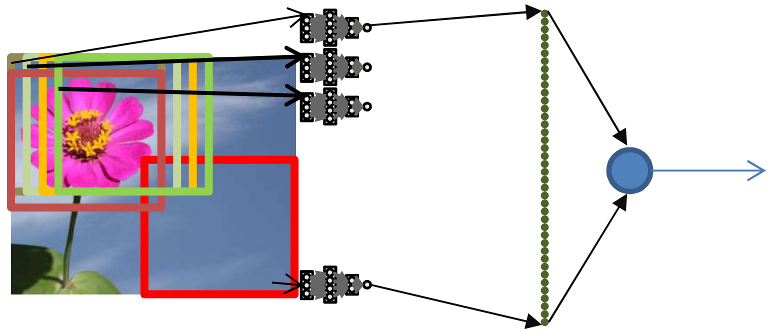
- For an $L$ width filter:
  - Odd $L$ : Pad on both left and right with $(L-1)/2$ columns of zeros
  - Even $L$ :  Pad one side with$L/2$ columns of zeros, and the other with $\frac{L}{2}-1$ columns of zeros
  - The resulting image is width  $N+L-1$
  - The result of the convolution is width $N$

- The top/bottom zero padding follows the same rules to maintain map height after convolution

- For hop size $S > 1$, zero padding is adjusted to ensure that the size of the convolved output is $\lceil N/S \rceil$
  - Achieved by *first* zero padding the image with $S\lceil N/S \rceil - N$ columns/rows of zeros and then applying above rules

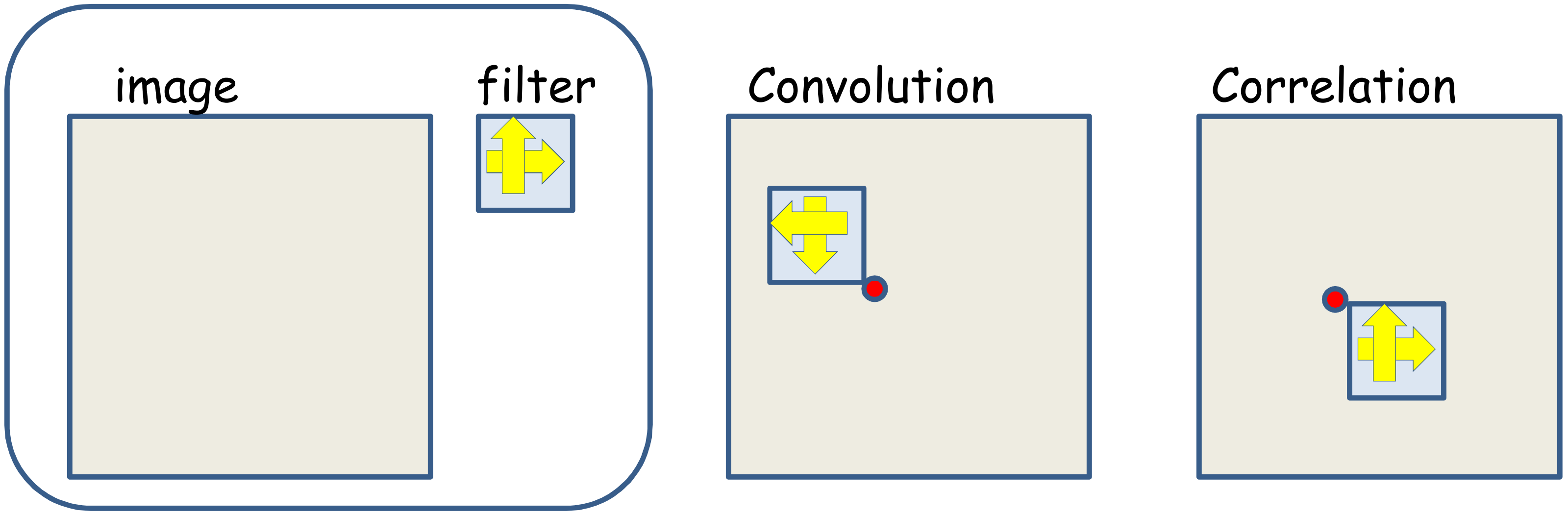
## Why convolution?





- Convolutional neural networks are, in fact, equivalent to *scanning* with an MLP
  - Just run the entire MLP on each block separately, and combine results
    - As opposed to scanning (convolving) the picture with individual neurons/filters
  - Even computationally, the number of operations in both computations is identical
    - The neocognitron in fact views it equivalently to a scan
- So why convolutions?

# Correlation, not Convolution



- The operation performed is technically a correlation, not a convolution
- **Correlation**:

$$y(i,j) = \sum_l \sum_m x(i+l, j+m) w(l,m)$$

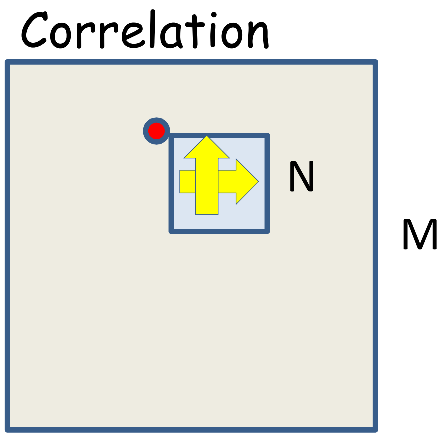
  – Shift the "filter" $w$ to "look" at the input $x$ block *beginning* at $(i,j)$

- **Convolution**:

$$y(i,j) = \sum_l \sum_m x(i-l, j-m) w(l,m)$$

- Effectively "flip" the filter, right to left, top to bottom

## Cost of Correlation

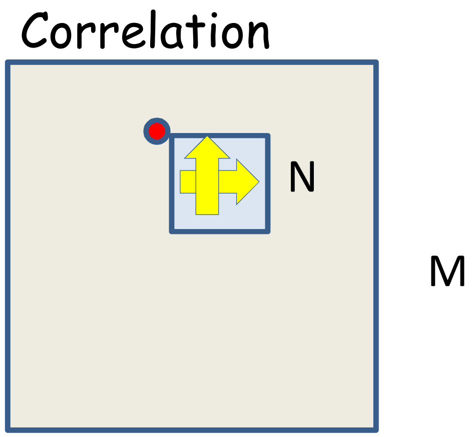


- **Correlation**:

$$y(i,j) = \sum_{l}\sum_{m} x(i+l, j+m)w(l,m)$$

- Cost of scanning an $M \times M$ image with an $N \times N$ filter: $O(M^2N^2)$
  - $N^2$ multiplications at each of $M^2$ positions
    - Not counting boundary effects
  - Expensive, for large filters

## Correlation in Transform Domain

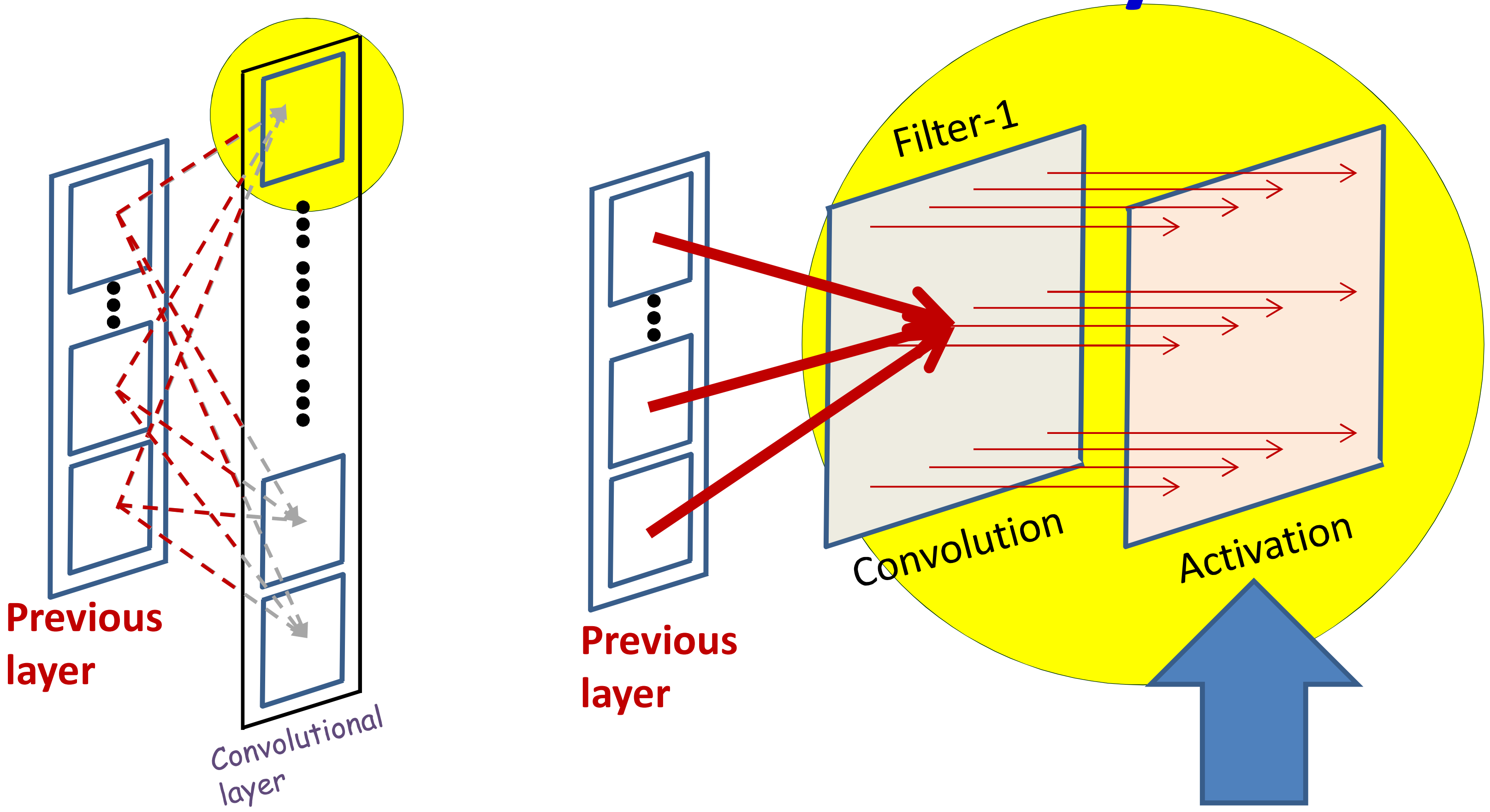


- **Correlation usind DFTs**:

$$\mathrm{Y} = IDFT2\big(DFT2(X) \circ conj(DFT2(W))\big)$$

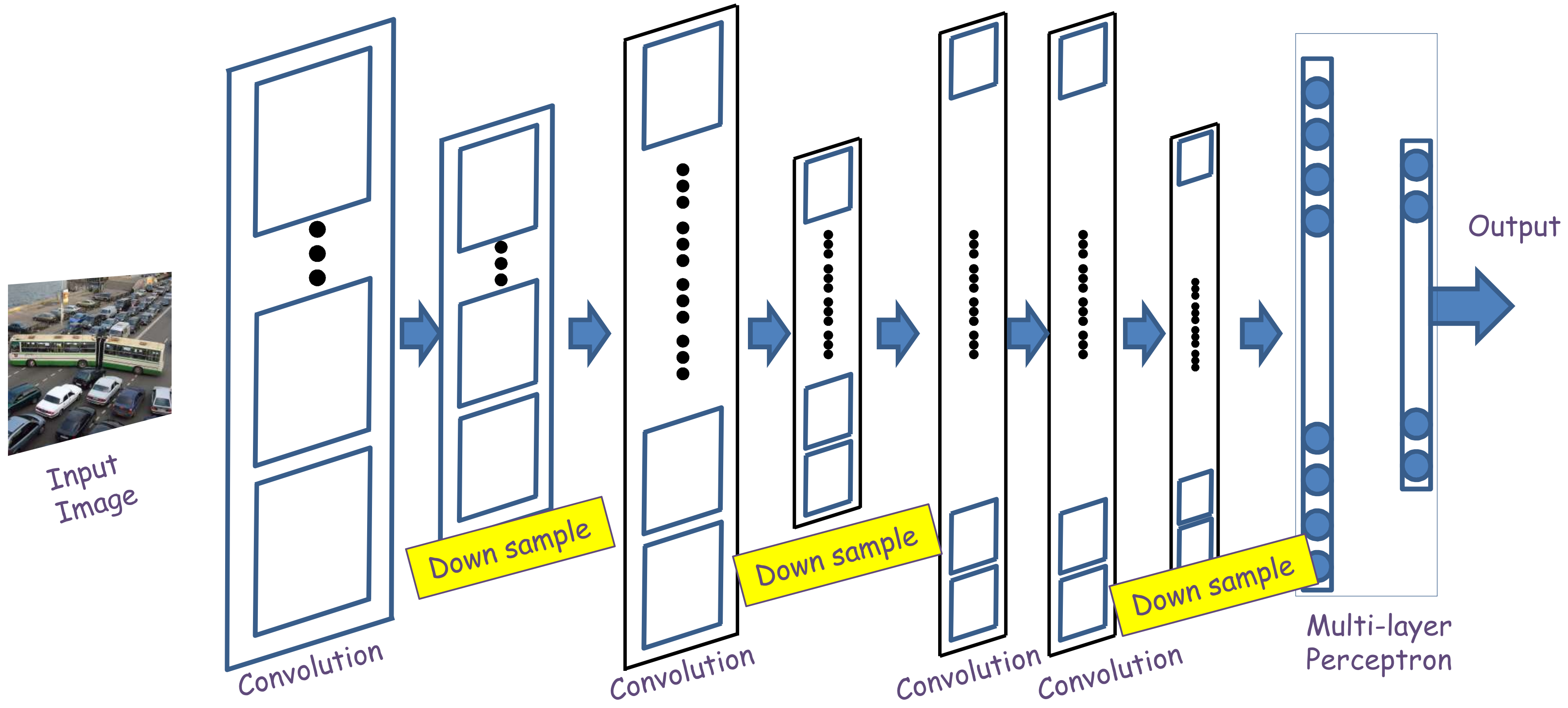
- Cost of doing this using the Fast Fourier Transform to compute the DFTs: $\mathrm{O}\big(M^2 logN\big)$
  - Significant saving for large filters
  - Or if there are many filters

# A convolutional layer



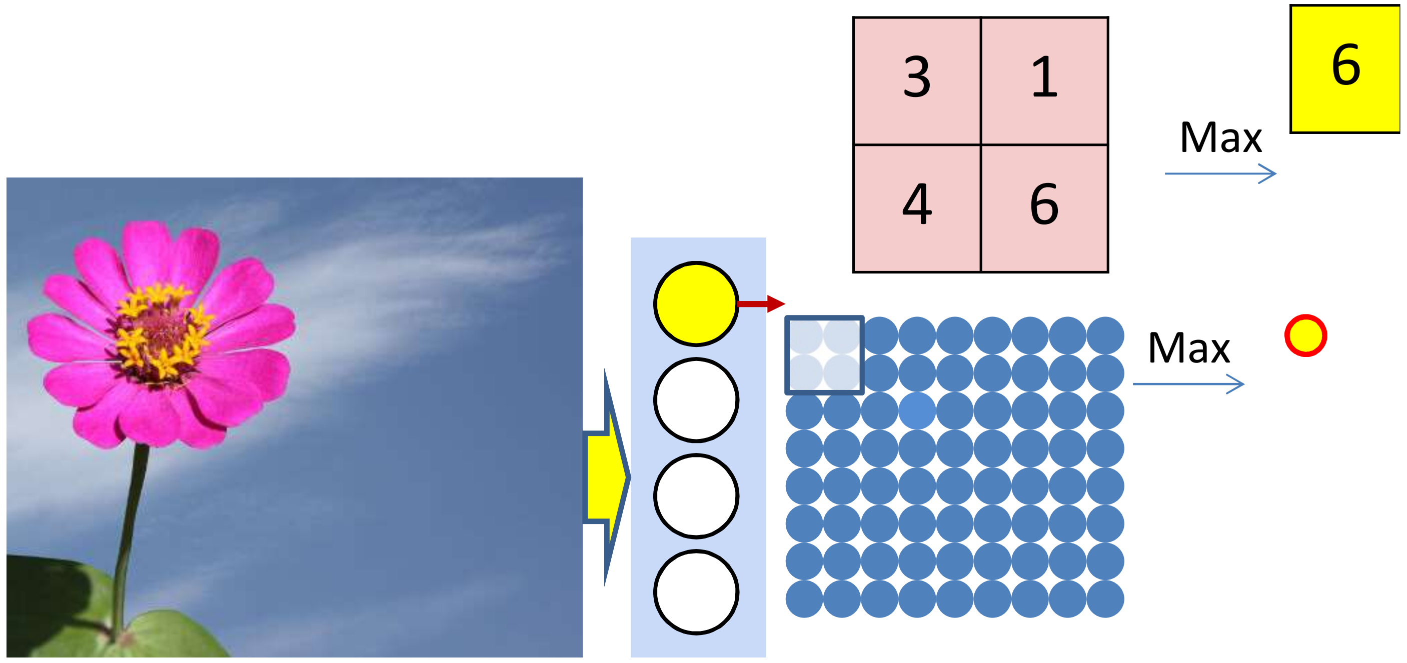
- The convolution operation results in a convolution map
- An *Activation* is finally applied to every entry in the map

# The other component
# Downsampling/Pooling



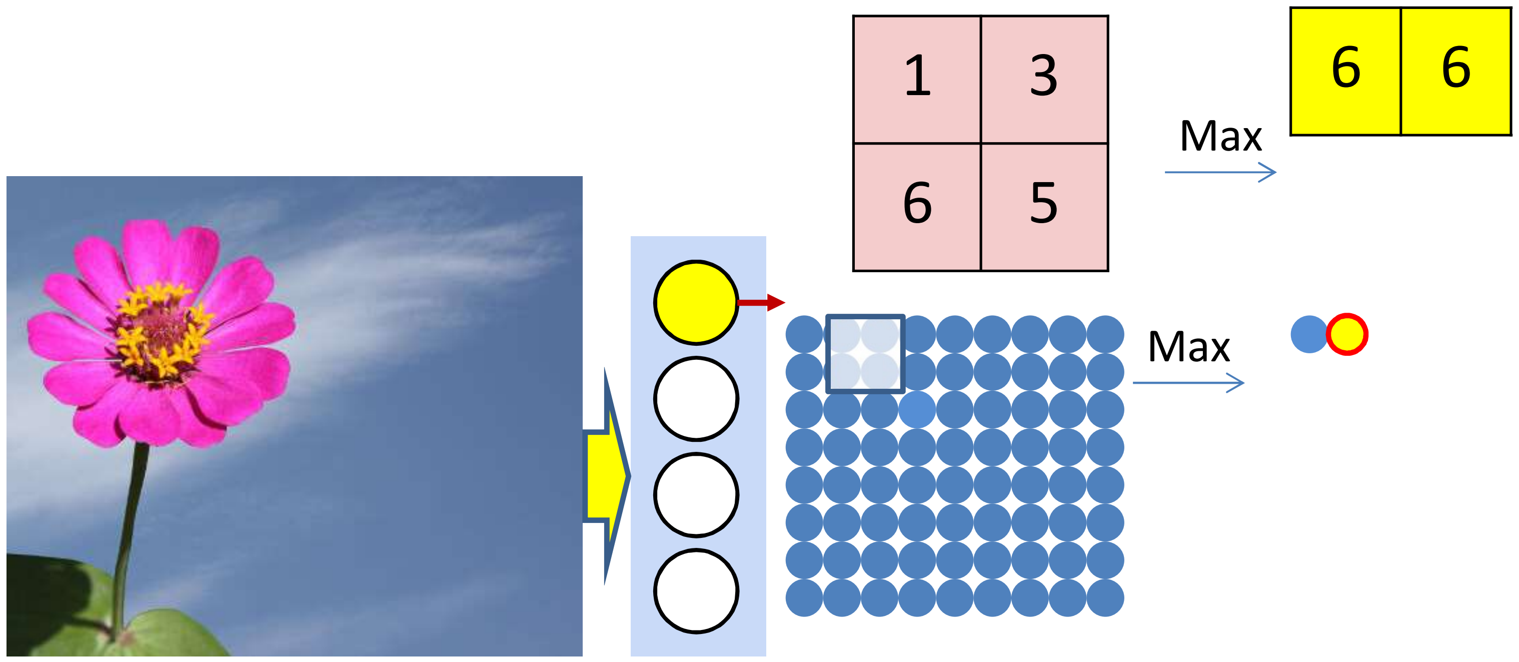
- Convolution (and activation) layers are followed intermittently by "downsampling" (or "pooling") layers
  - Often, they alternate with convolution, though this is not necessary

## Recall: Max pooling



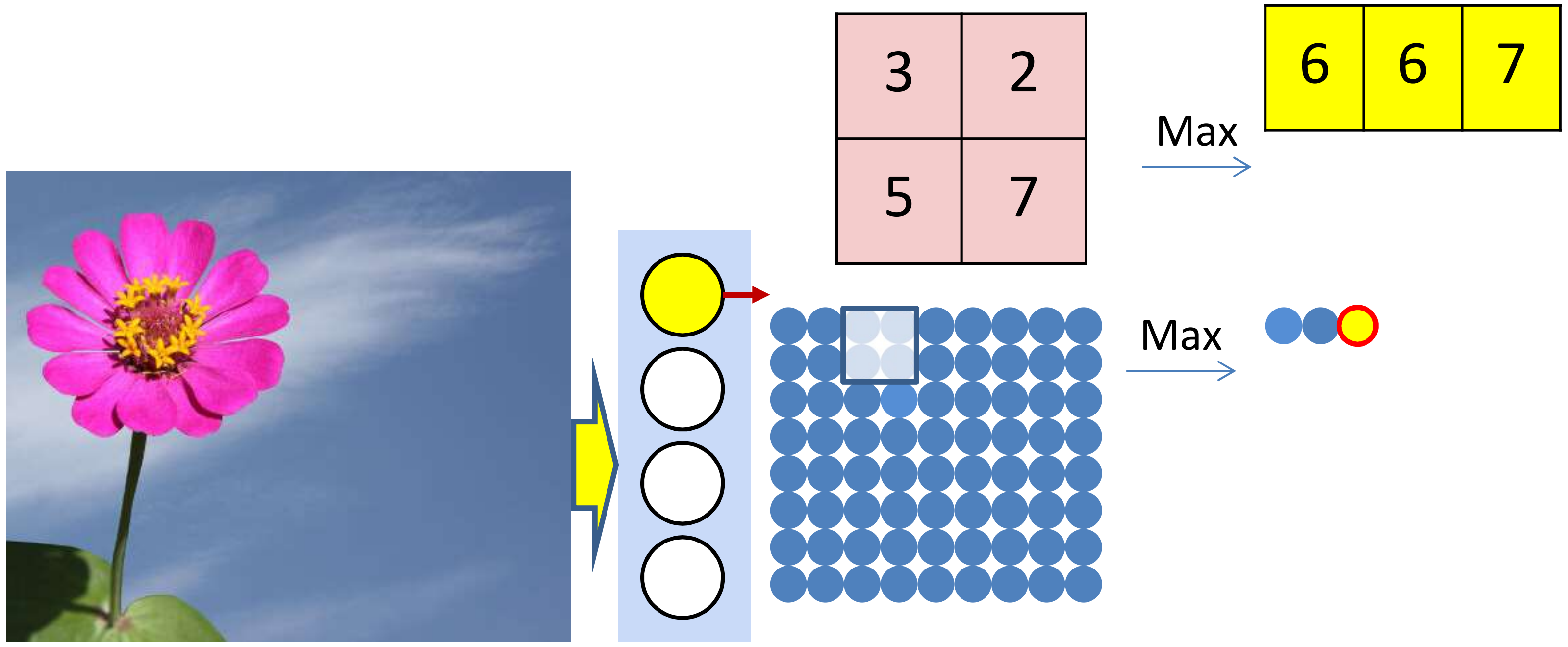
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

## Recall: Max pooling



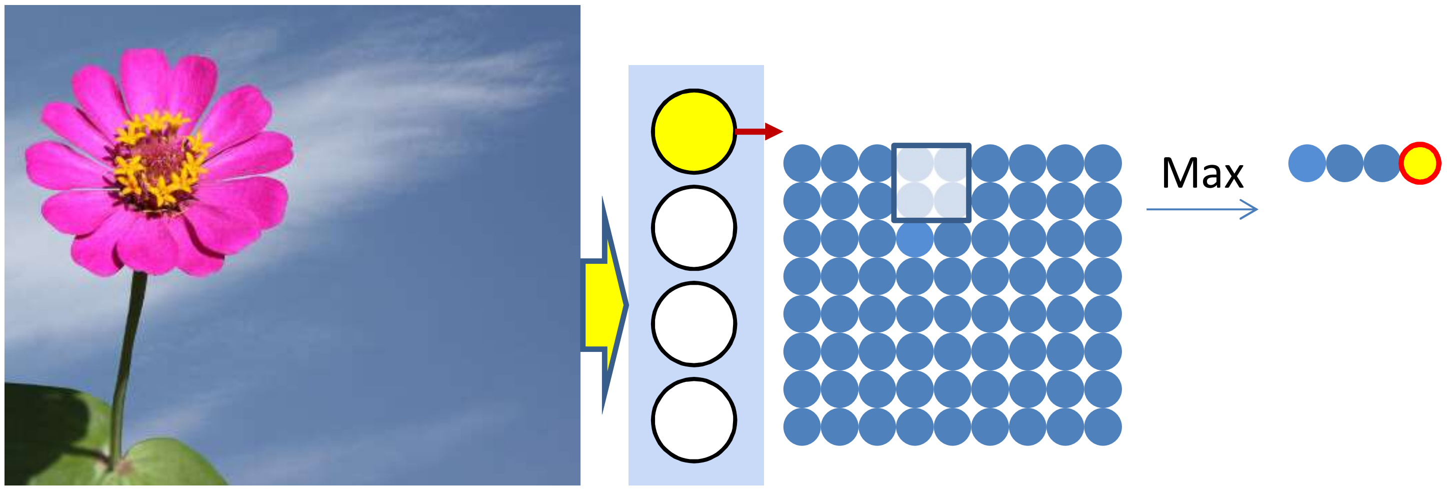
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

# Recall: Max pooling



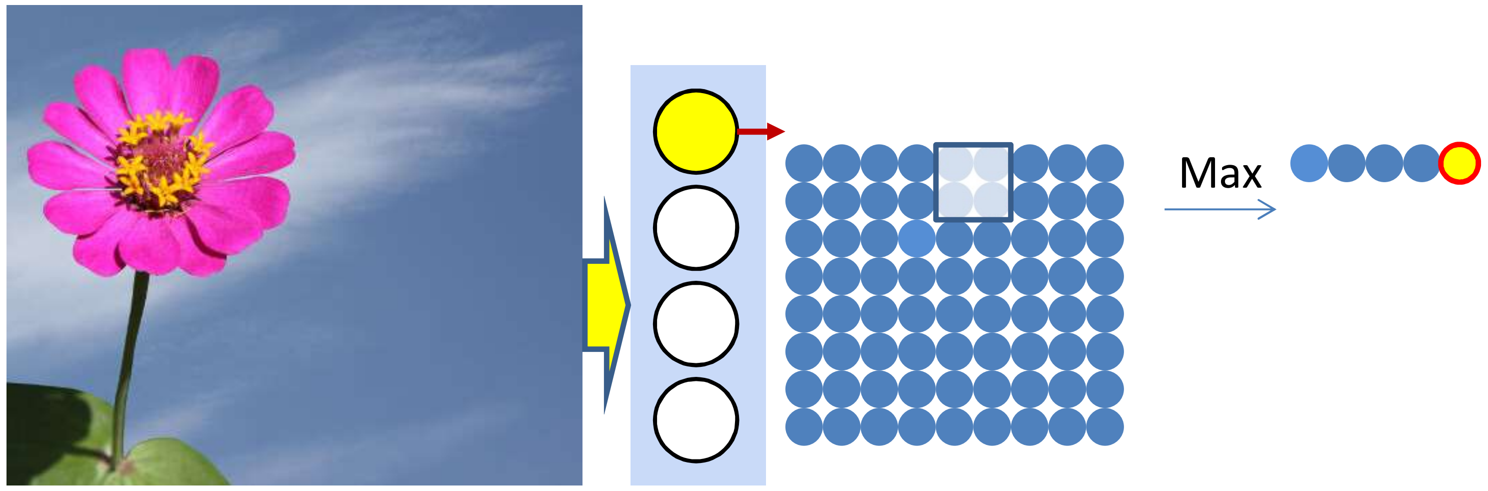
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

## Recall: Max pooling



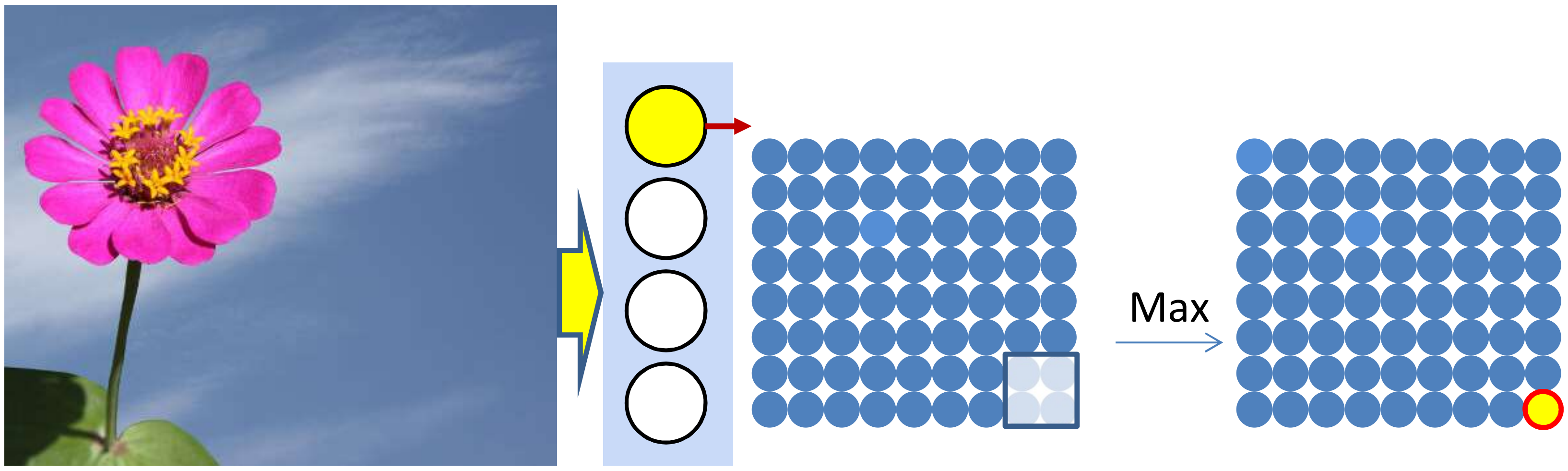
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

## Recall: Max pooling



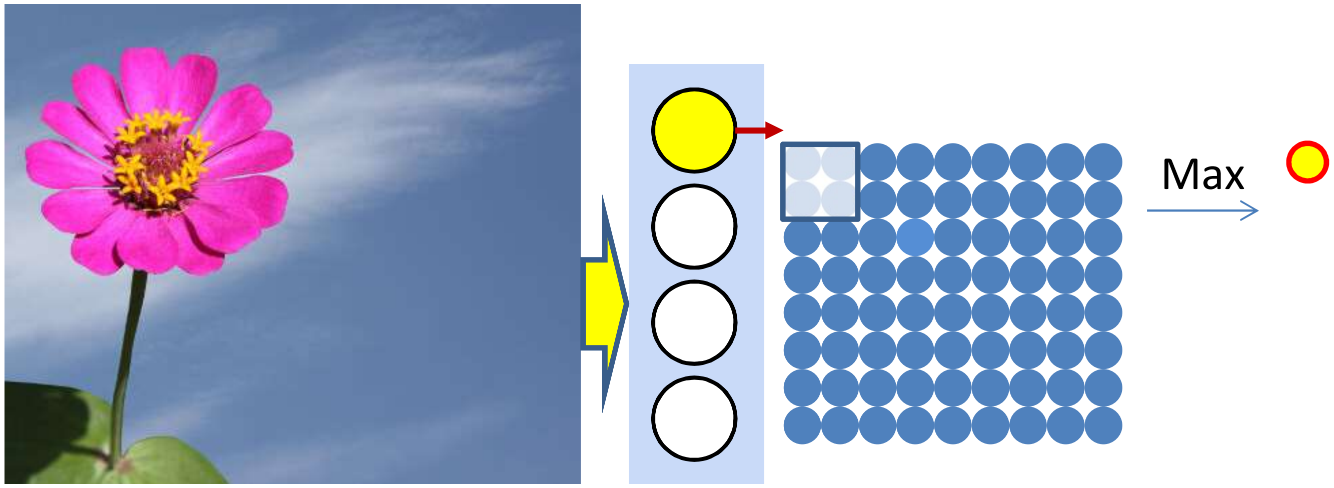
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

## Recall: Max pooling



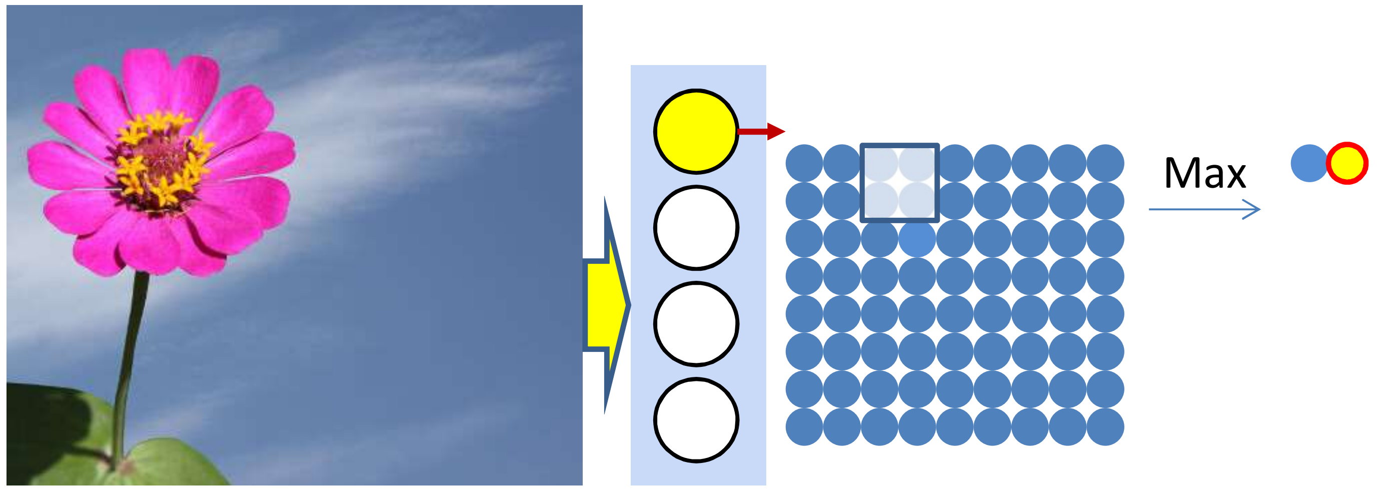
- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input

# “Strides”



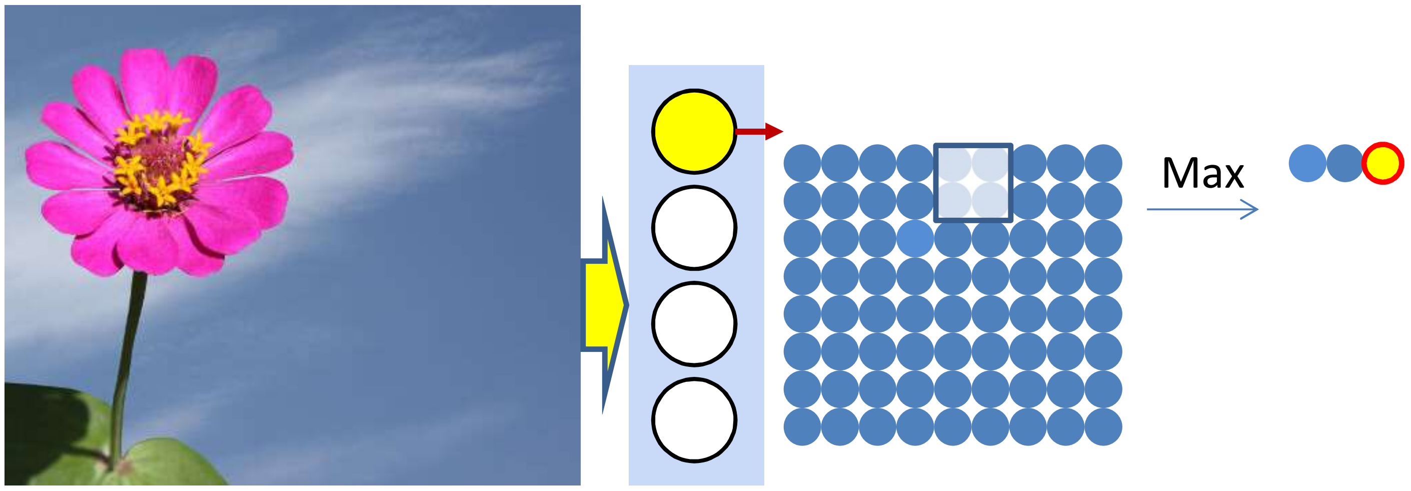
- The “max” operations may “stride” by more than one pixel

# “Strides”



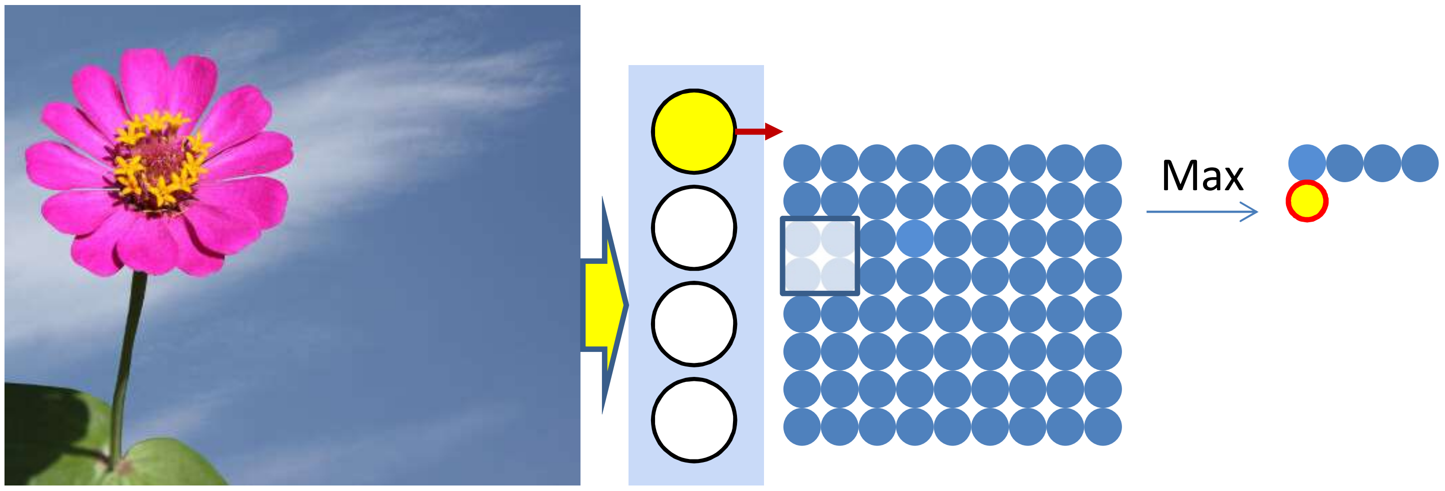
- The “max” operations may “stride” by more than one pixel

# “Strides”



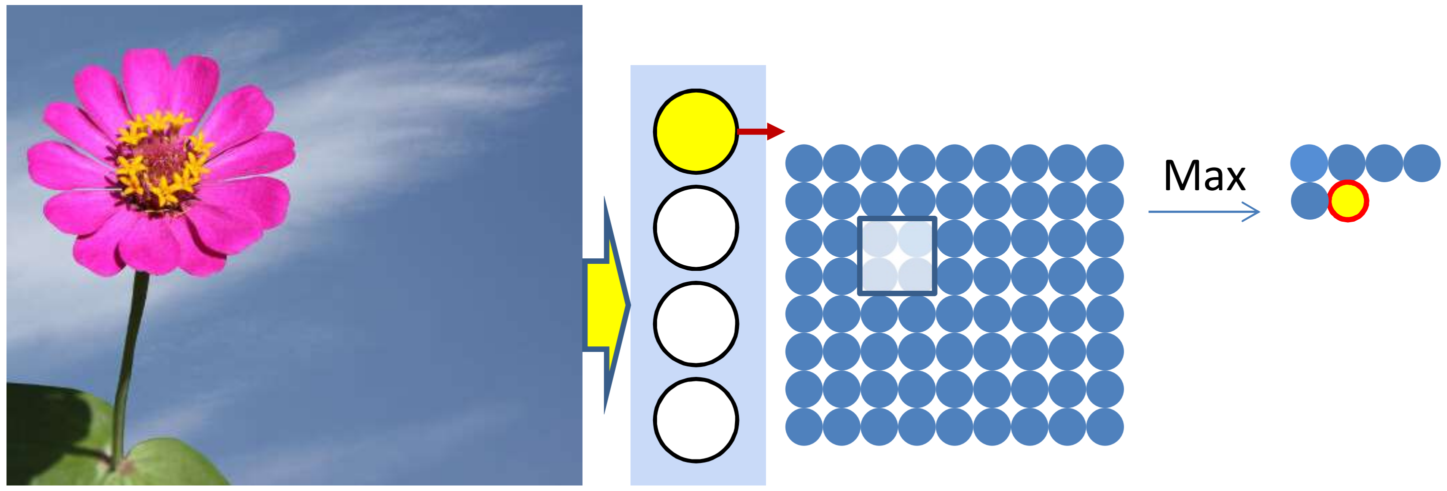
- The “max” operations may “stride” by more than one pixel

## “Strides”



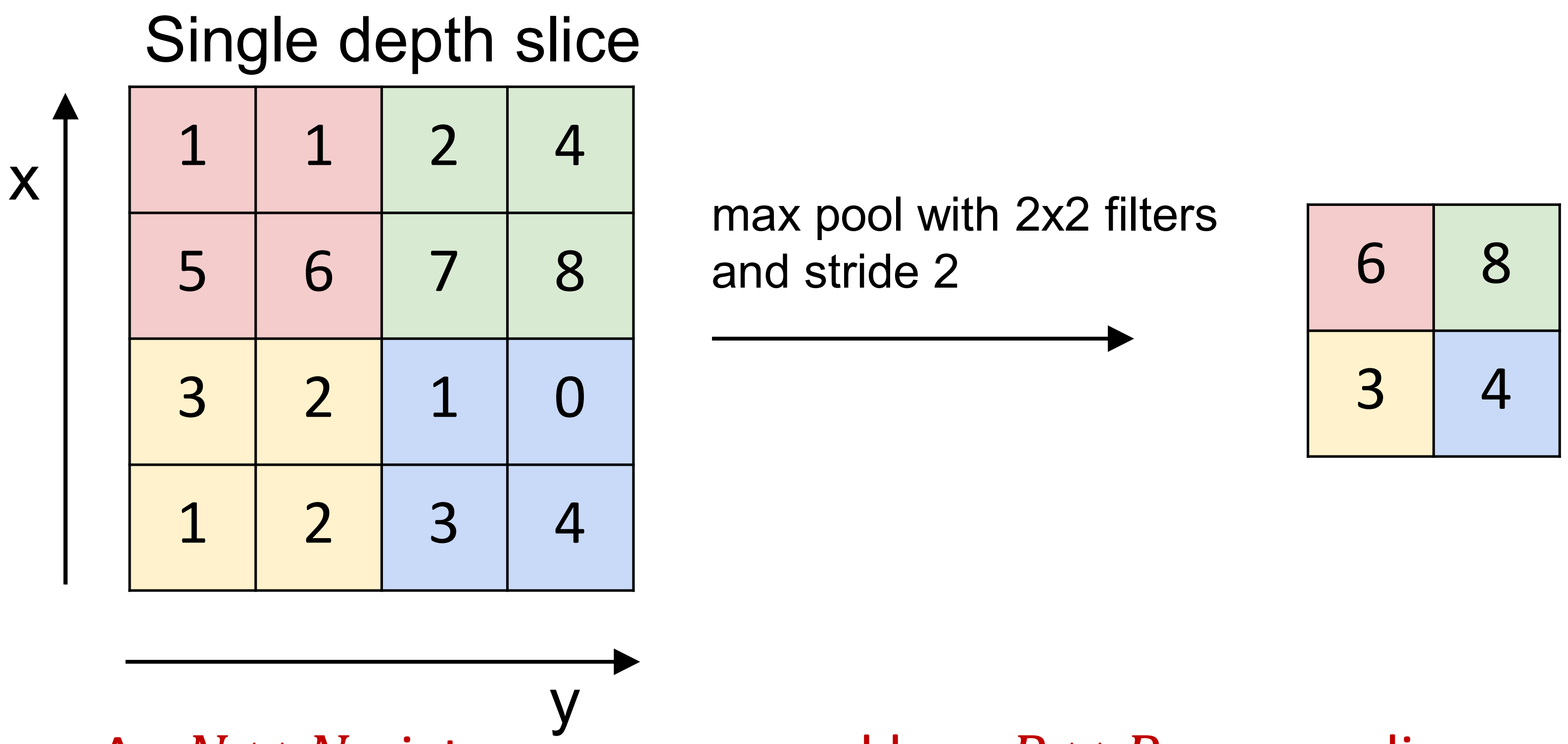
- The “max” operations may “stride” by more than one pixel

## “Strides”



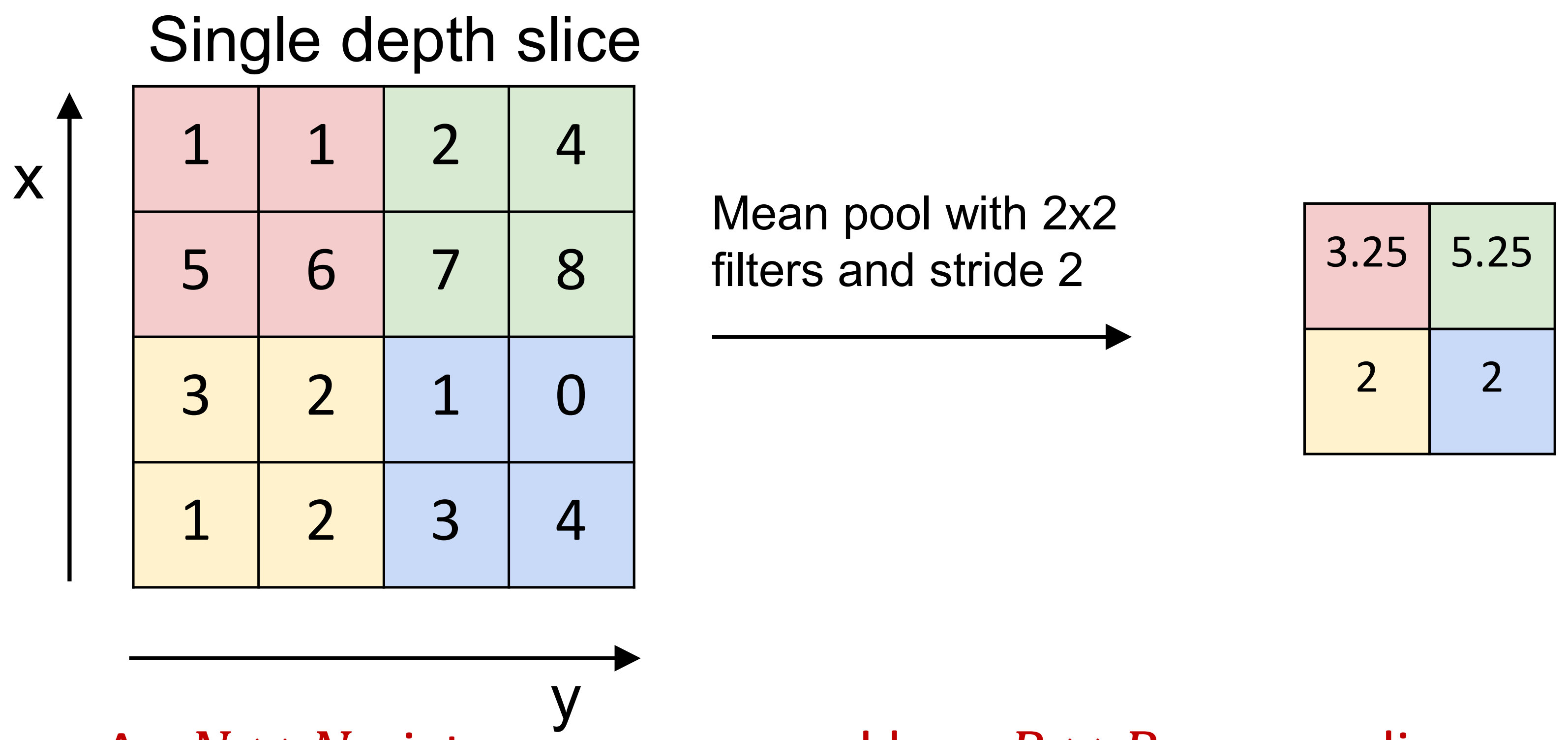
- The “max” operations may “stride” by more than one pixel

## Max Pooling



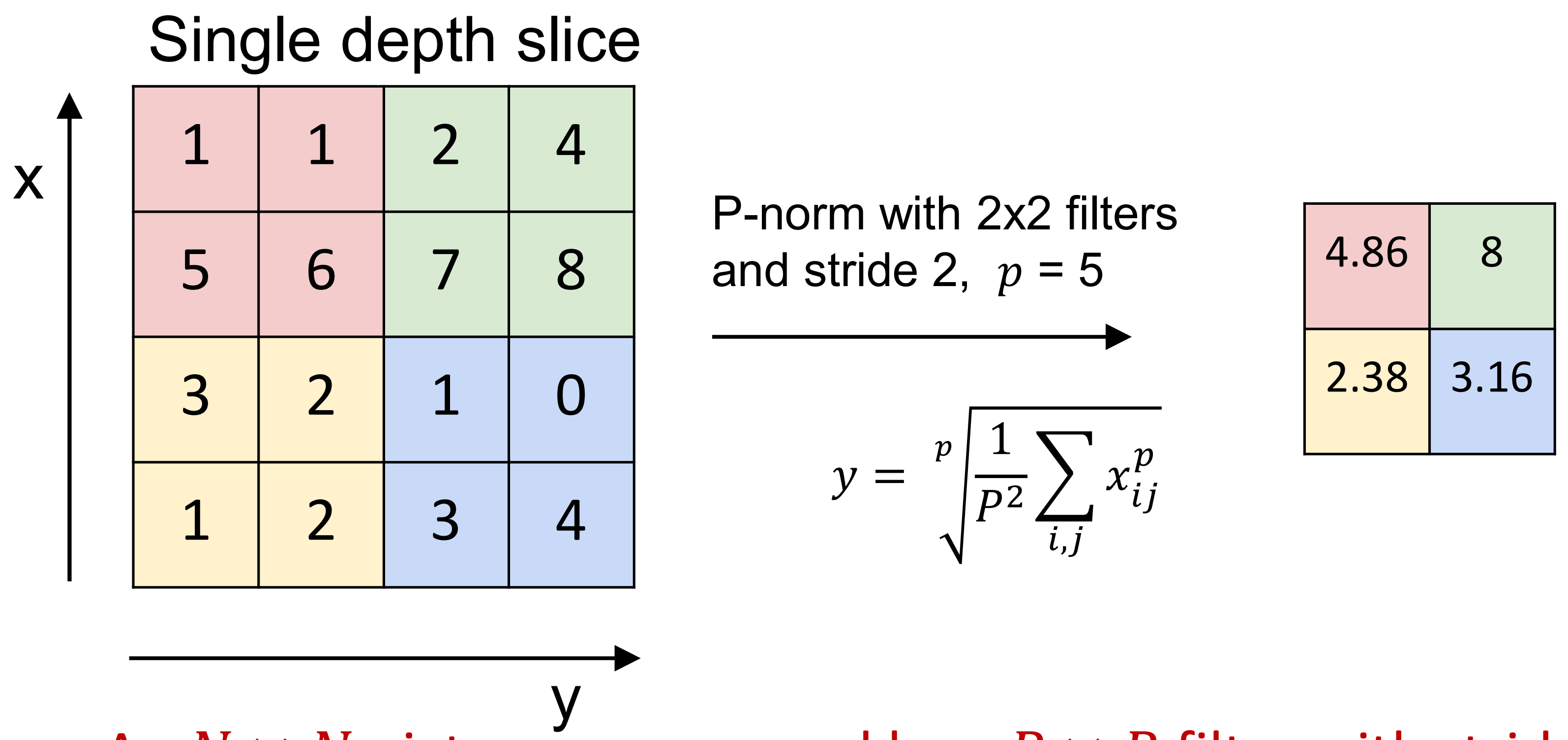
- An $N \times N$ picture compressed by a $P \times P$ maxpooling filter with stride $D$ results in an output map of side $\lceil (N - P)/D \rceil + 1$
  - Typically do not zero pad

# Alternative to Max pooling: Mean Pooling



- An $N \times N$ picture compressed by a $P \times P$ maxpooling filter with stride $D$ results in an output map of side $\lceil (N - P)/D \rceil + 1$
  - Typically do not zero pad

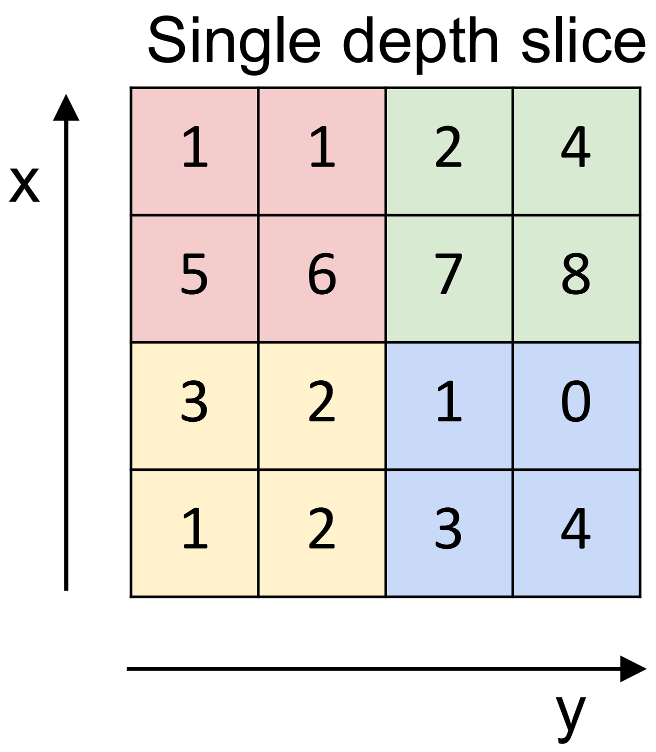
# Alternative to Max pooling: P-norm



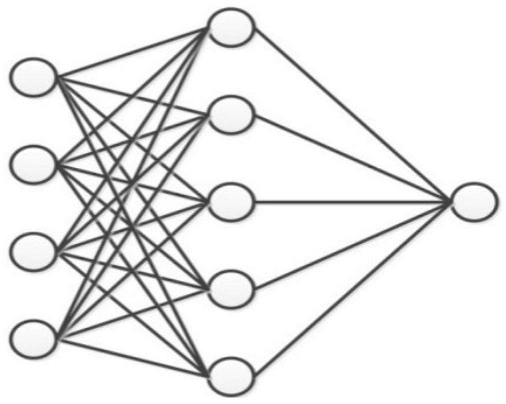
$$y = \sqrt[p]{\frac{1}{P^2}\sum_{i,j} x_{ij}^p}$$

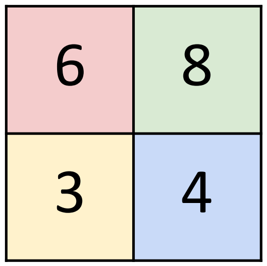
- An $N \times N$ picture compressed by a $P \times P$ filter with stride $D$ results in an output map of side $\lceil (N-P)/D \rceil + 1$

# Other options



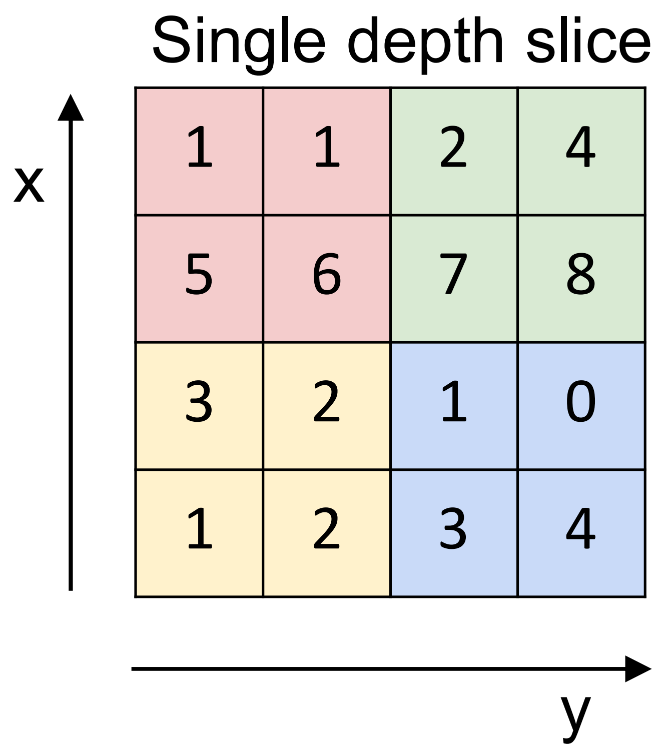
Network applies to each 2x2 block and strides by 2 in this example



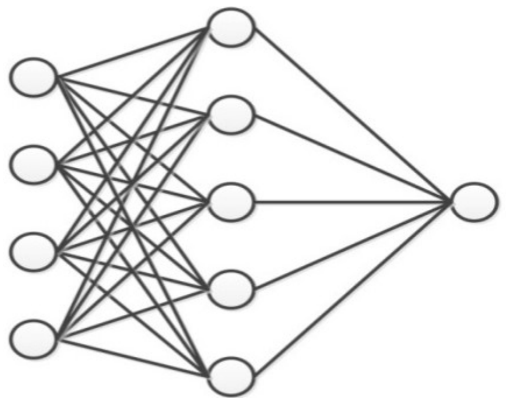


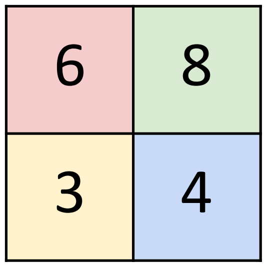
- The pooling may even be a *learned* filter
  - The *same* network is applied on each block
    - (Again, a shared parameter network)

# Other options



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

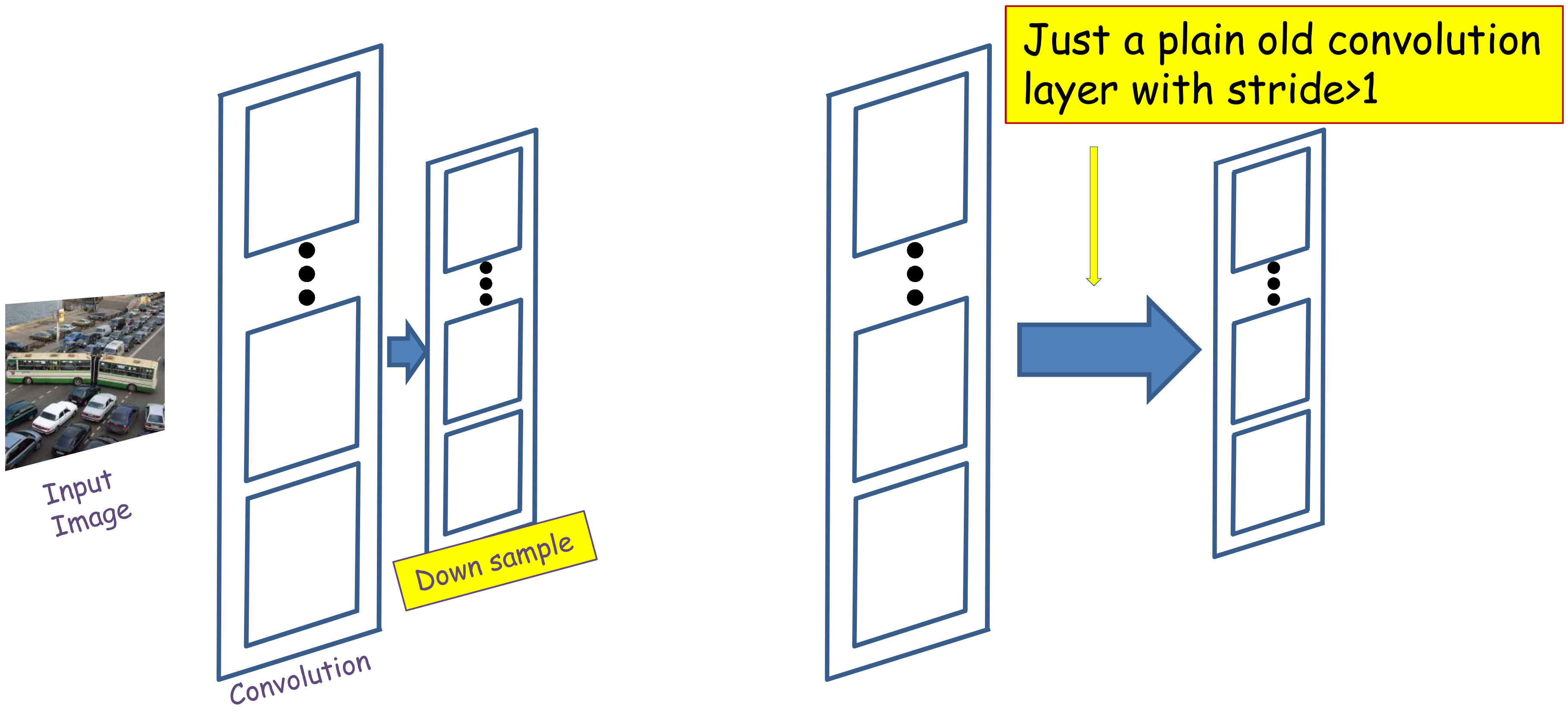




Network in network

- The pooling may even be a *learned* filter
  - The *same* network is applied on each block
    - (Again, a shared parameter network)

# Or even an "all convolutional" net



- Downsampling may even be done by a simple convolution layer with stride larger than 1
  - Replacing the maxpooling layer with a conv layer

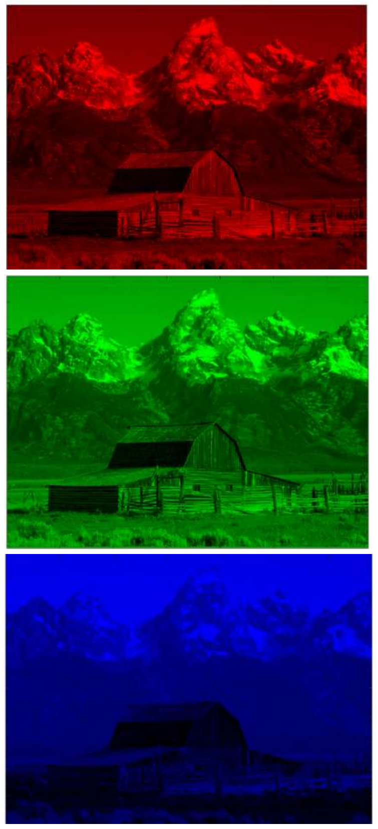
## Setting everything together

- Typical image classification task
  - Assuming maxpooling..

## Convolutional Neural Networks

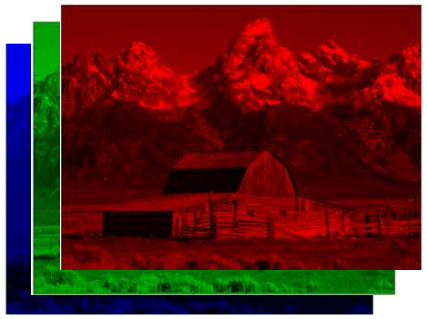
- Input: 1 or 3 images
  - Black and white or color
  - Will assume color to be generic

# Convolutional Neural Networks



- Input: 3 pictures

## Convolutional Neural Networks

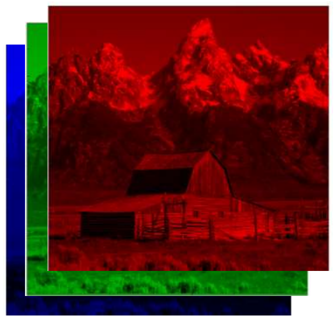


- Input: 3 pictures

## Preprocessing

- Typically works with *square* images
  - Filters are also typically square
- Large networks are a problem
  - Too much detail
  - Will need big networks
- Typically scaled to small sizes, e.g. 32x32 or 128x128

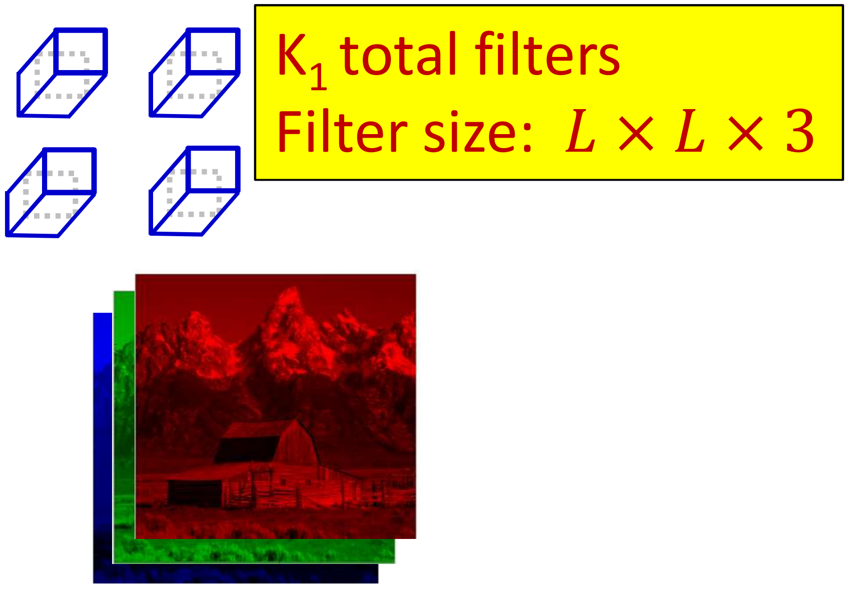
## Convolutional Neural Networks



$I \times I\ image$

- Input: 3 pictures

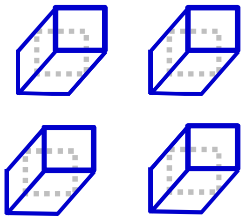
# Convolutional Neural Networks



$I \times I$ *image*

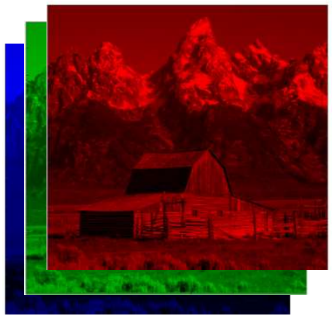
- Input is convolved with a set of $K_1$ filters
  - Typically $K_1$ is a power of 2, e.g. 2, 4, 8, 16, 32,..
  - Filters are typically 5x5, 3x3, or even 1x1

# Convolutional Neural Networks



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

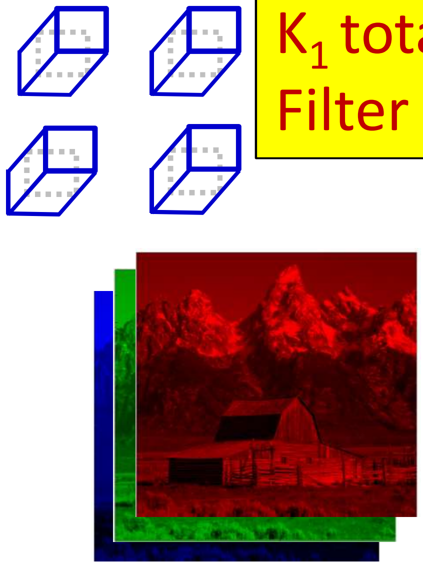
Small enough to capture fine features
(particularly important for scaled-down images)



$I \times I$ image

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

# Convolutional Neural Networks



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

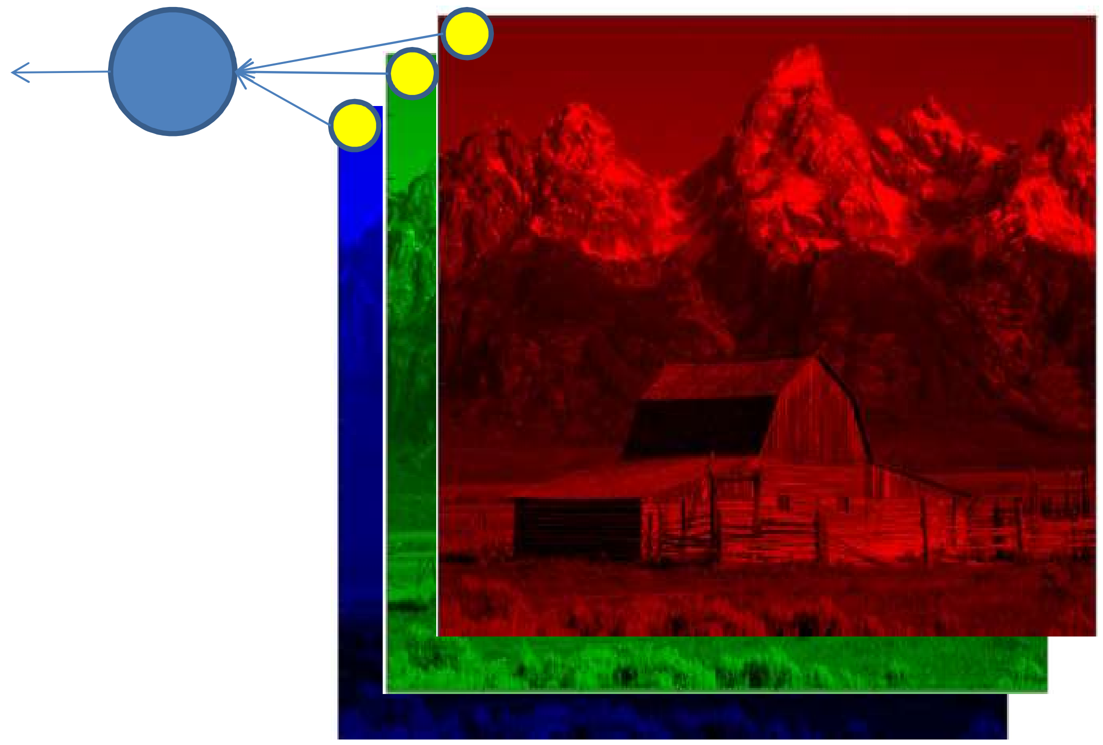
Small enough to capture fine features
(particularly important for scaled-down images)

What on earth is this?

$I \times I$ *image*

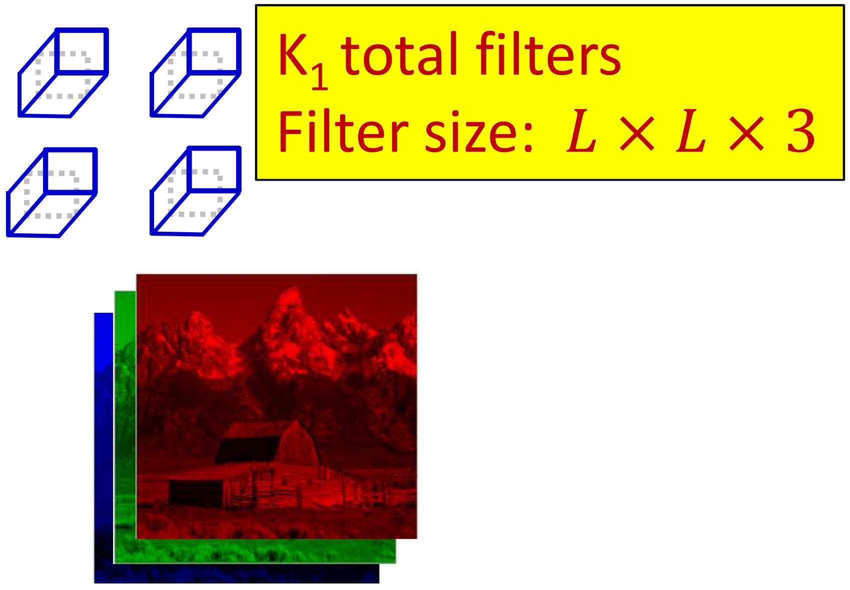
- Input is convolved with a set of $K_1$ filters
  - Typically $K_1$ is a power of 2, e.g. 2, 4, 8, 16, 32,..
  - Filters are typically 5x5, 3x3, or even 1x1

## The 1x1 filter



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

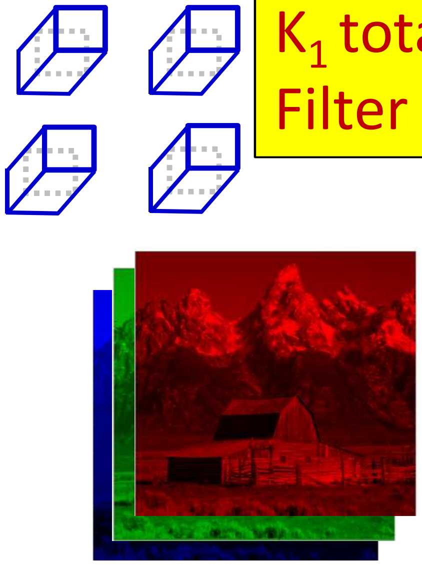
# Convolutional Neural Networks



$I \times I$ *image*

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

# Convolutional Neural Networks



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

$I \times I$ $image$

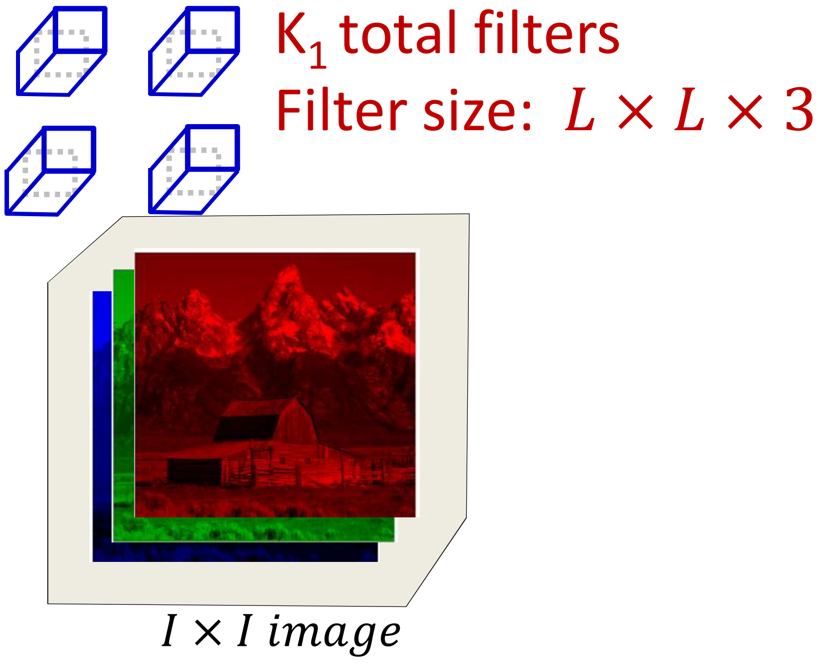
Parameters to choose: $K_1$, $L$ and $S$

1. Number of filters $K_1$
2. Size of filters $L \times L \times 3 + bias$
3. *Stride* of convolution $S$

Total number of parameters: $K_1(3L^2 + 1)$

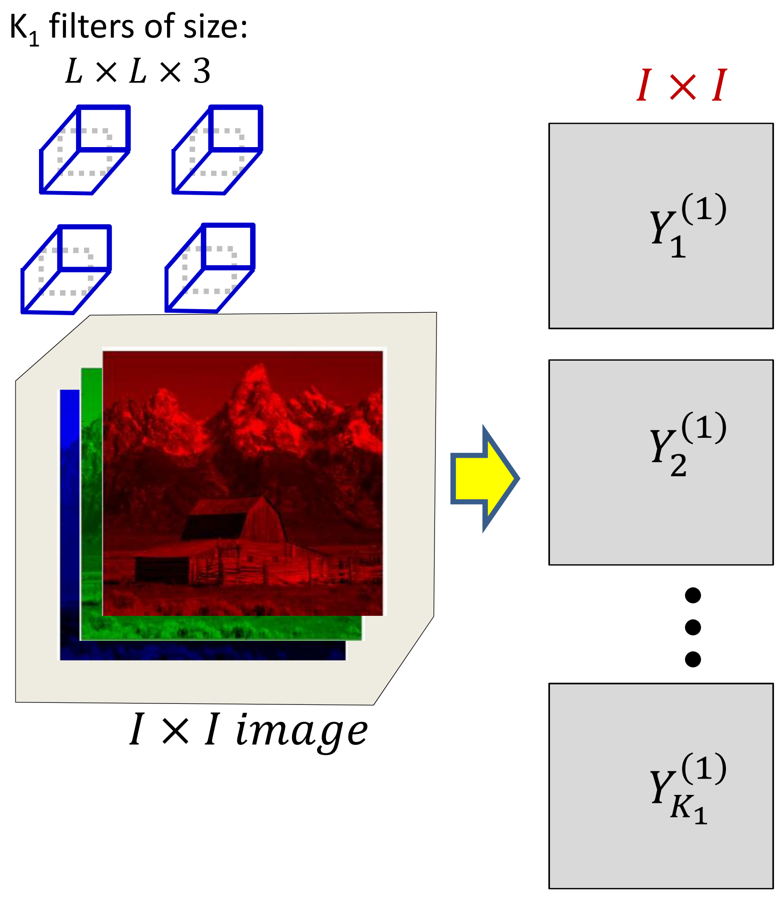
- Input is convolved with a set of $K_1$ filters
  - Typically $K_1$ is a power of 2, e.g. 2, 4, 8, 16, 32,..
  - **Better notation:** Filters are typically 5x5(x3), 3x3(x3), or even 1x1(x3)
  - ***Typical stride***: 1 or 2

# Convolutional Neural Networks



- The input may be zero-padded according to the size of the chosen filters

# Convolutional Neural Networks



The layer includes a convolution operation followed by an activation (typically RELU)

$$z_m^{(1)}(i,j) = \sum_{c \in \{R,G,B\}} \sum_{k=1}^{L} \sum_{l=1}^{L} w_m^{(1)}(c,k,l) I_c(i+k, j+l) + b_m^{(1)}$$

$$Y_m^{(1)}(i,j) = f\left(z_m^{(1)}(i,j)\right)$$

- **First convolutional layer:** Several convolutional filters
  - Filters are “3-D” (third dimension is color)
  - Convolution followed typically by a RELU activation
- Each filter creates a single 2-D output map

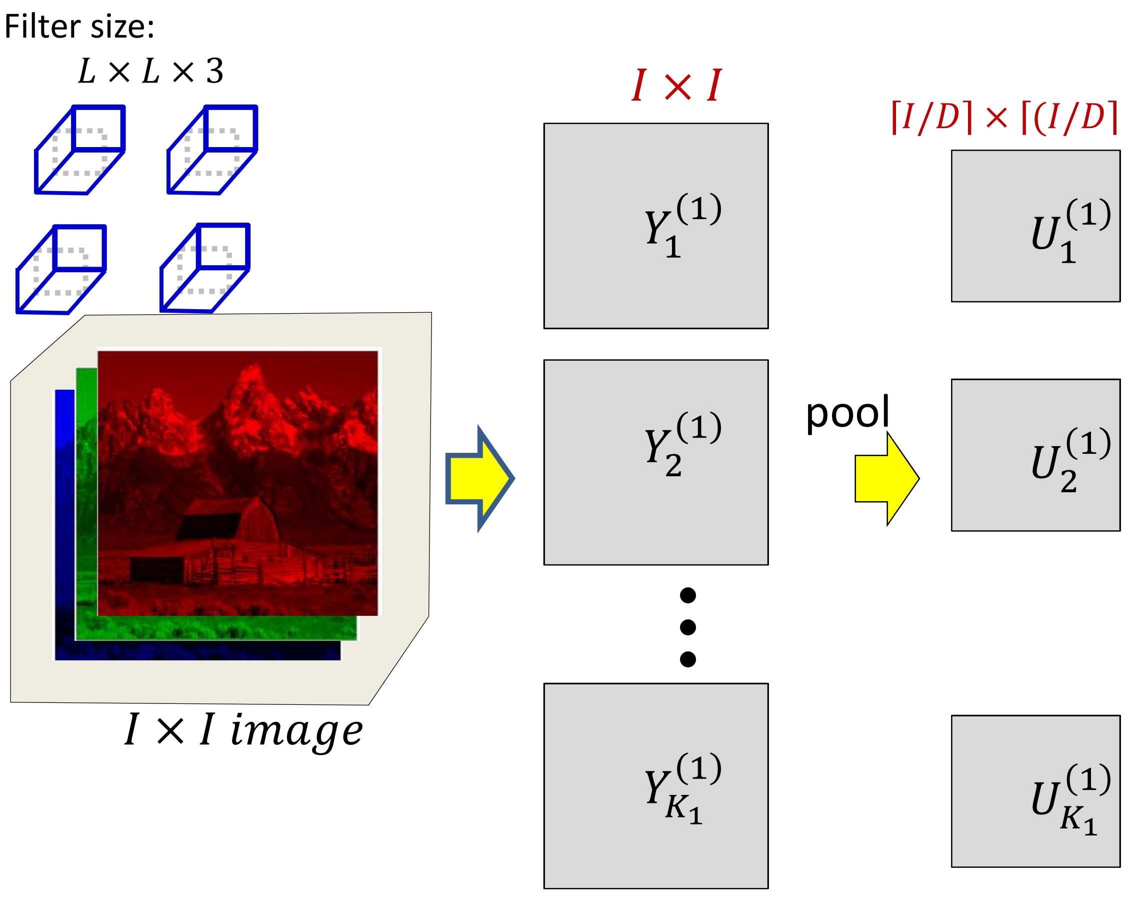
## Learnable parameters in the first convolutional layer

- The first convolutional layer comprises $K_1$ filters, each of size $L \times L \times 3$
  - Spatial span: $L \times L$
  - Depth : 3 (3 colors)

- This represents a total of $K_1\left(3L^2 + 1\right)$ parameters
  - "+ 1" because each filter also has a bias

- All of these parameters must be learned

# Convolutional Neural Networks

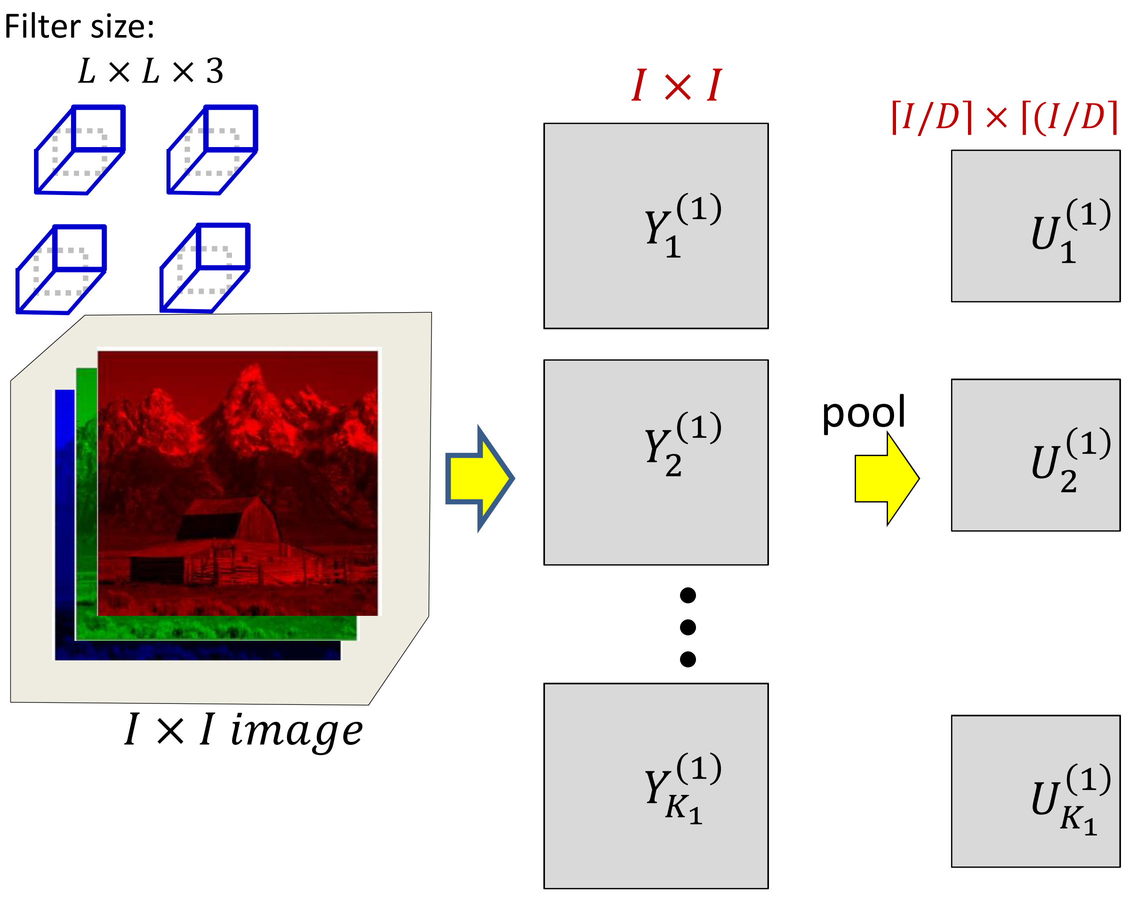


The layer pools PxP blocks of Y into a single value
It employs a stride D between adjacent blocks

$$U_m^{(1)}(i,j) = \max_{\substack{k \in \{(i-1)D+1,\, iD\},\\ l \in \{(j-1)D+1,\, jD\}}} Y_m^{(1)}(k,l)$$

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

# Convolutional Neural Networks



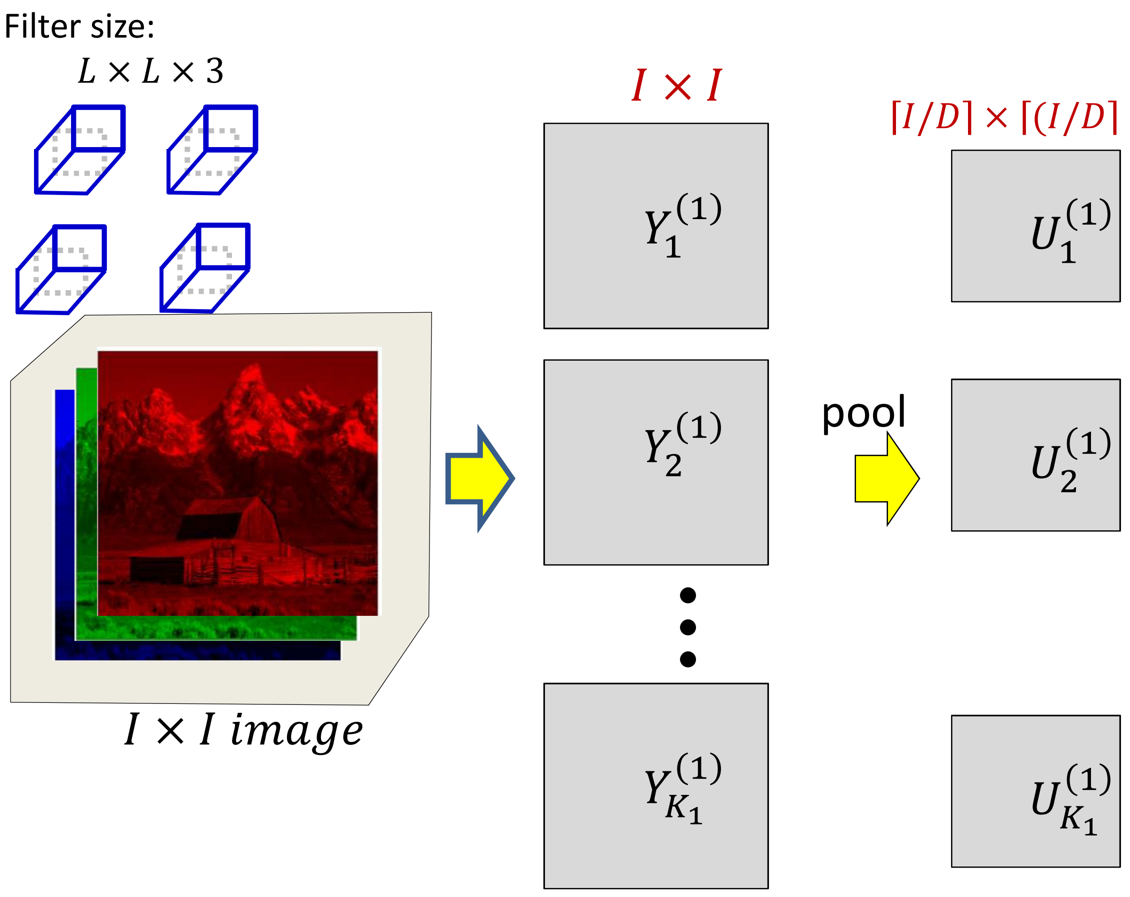
$$U_m^{(1)}(i,j) = \max_{\substack{k \in \{(i-1)D+1,\ iD\},\\ l \in \{(j-1)D+1,\ jD\}}} Y_m^{(1)}(k,l)$$

Parameters to choose:
Size of pooling block $P$
Pooling stride $D$

Choices: Max pooling or mean pooling?
Or learned pooling?

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

# Convolutional Neural Networks

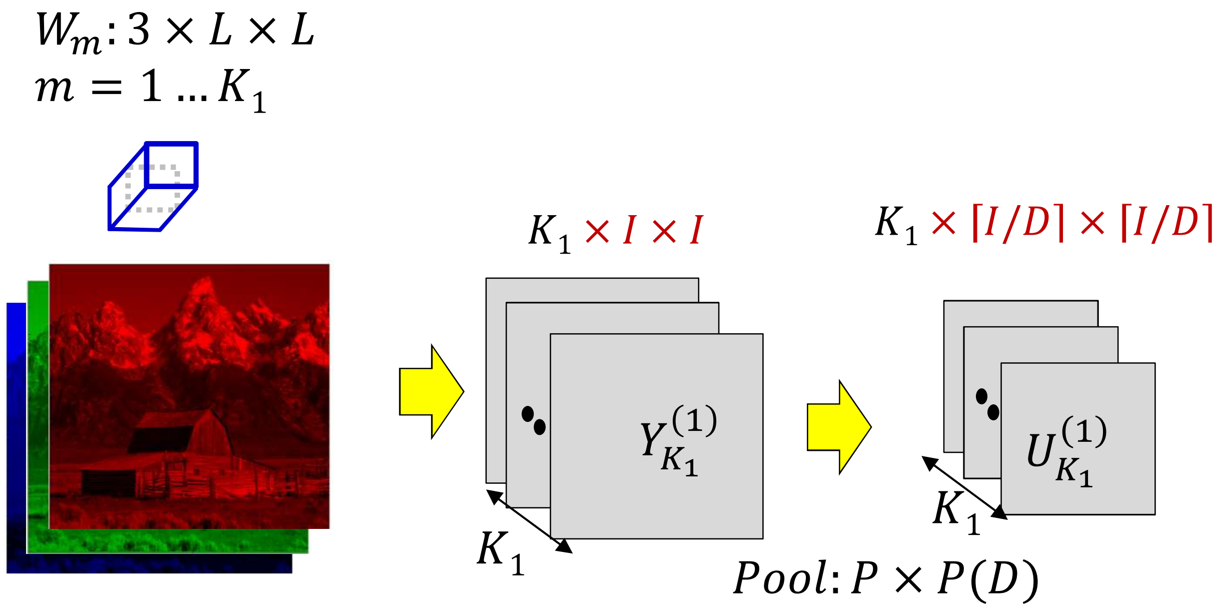


$$P_m^{(1)}(i,j) = \underset{\substack{k\in\{(i-1)D+1,\, iD\},\\ l\in\{(j-1)D+1,\, jD\}}}{\operatorname{argmax}} Y_m^{(1)}(k,l)$$

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

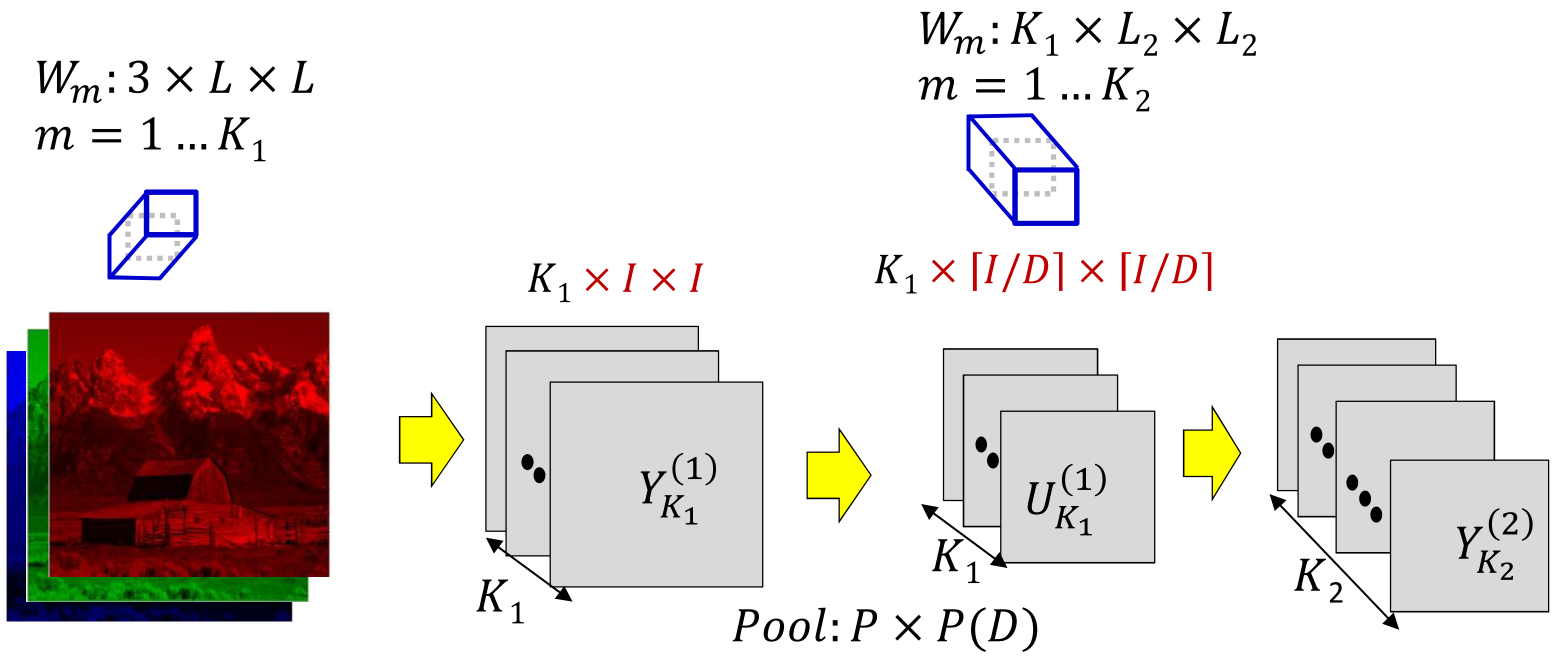
- **First downsampling layer:** From each $P \times P$ block of each map, *pool* down to a single value
  - **For max pooling, during training keep track of which position had the highest value**

# Convolutional Neural Networks



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

# Convolutional Neural Networks

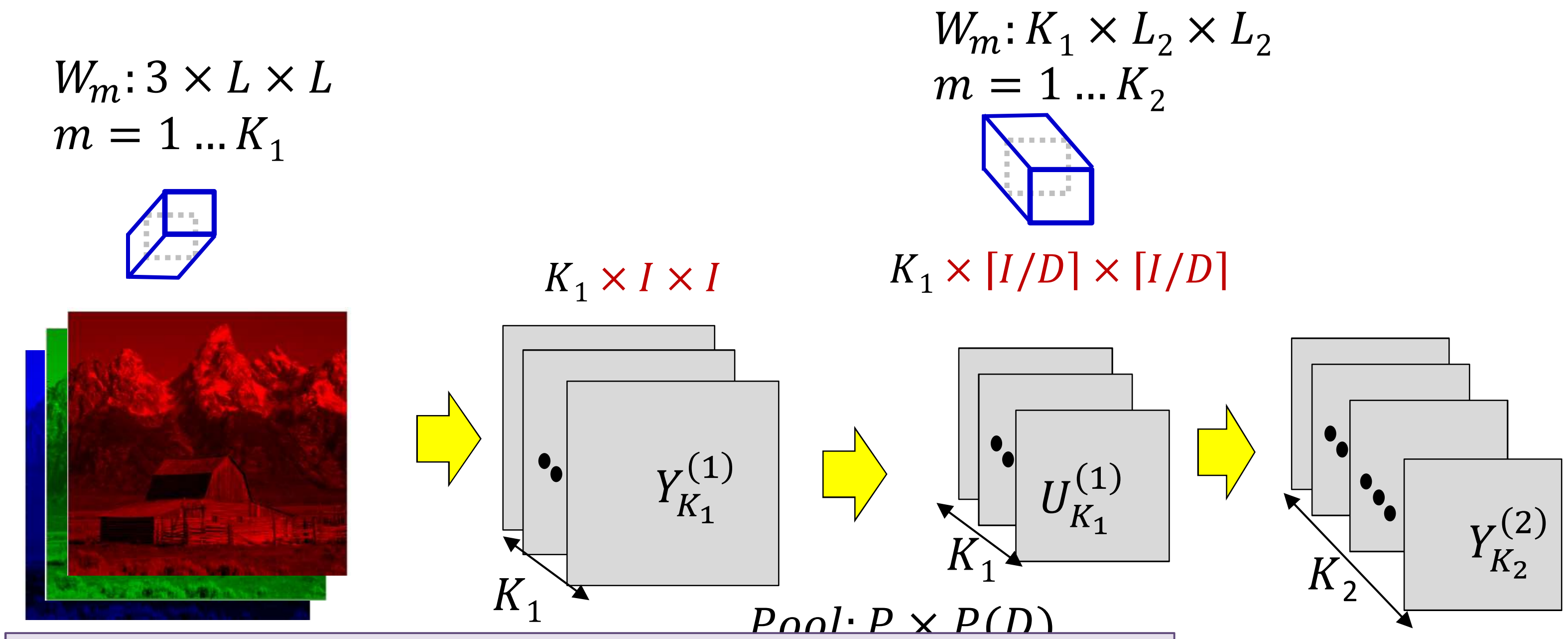


$$z_m^{(n)}(i,j) = \sum_{r=1}^{K_{n-1}} \sum_{k=1}^{L_n} \sum_{l=1}^{L_n} w_m^{(n)}(r,k,l) U_r^{(n-1)}(i+k, j+l) + b_m^{(n)}$$

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

- **Second convolutional layer:** $K_2$ 3-D filters resulting in $K_2$ 2-D maps

# Convolutional Neural Networks



$_m^{(n)}$

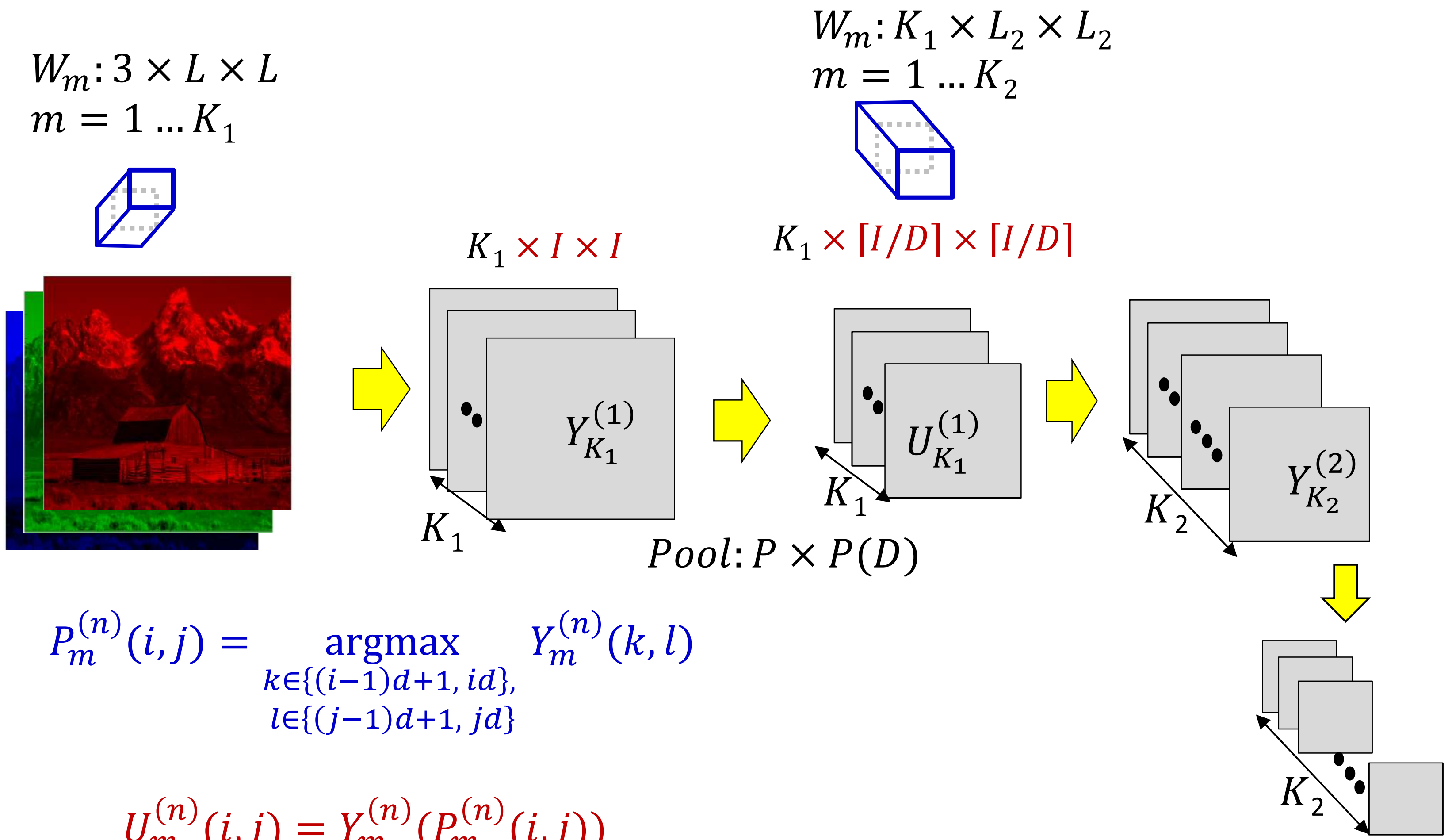
Parameters to choose: $K_2$, $L_2$ and $S_2$

1. Number of filters $K_2$
2. Size of filters $L_2 \times L_2 \times K_1 + bias$
3. *Stride* of convolution $S_2$

Total number of parameters: $K_2(K_1 L_2^2 + 1)$
All these parameters must be learned

ng in $K_2$ 2-D maps

# Convolutional Neural Networks

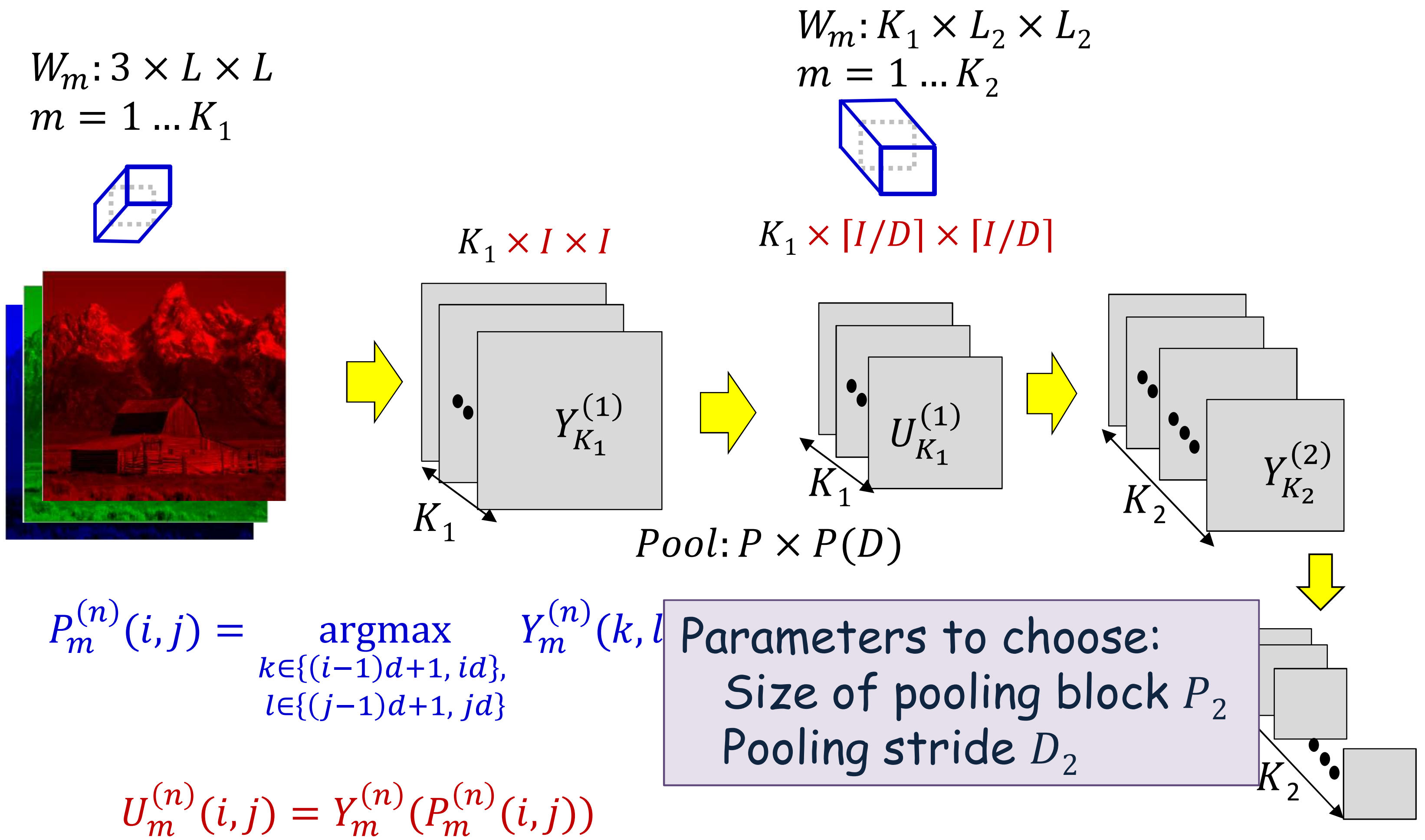


$$P^{(n)}_m(i,j) = \underset{\substack{k\in\{(i-1)d+1,\, id\},\\ l\in\{(j-1)d+1,\, jd\}}}{\operatorname{argmax}} Y^{(n)}_m(k,l)$$

$$U^{(n)}_m(i,j) = Y^{(n)}_m(P^{(n)}_m(i,j))$$

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

# Convolutional Neural Networks



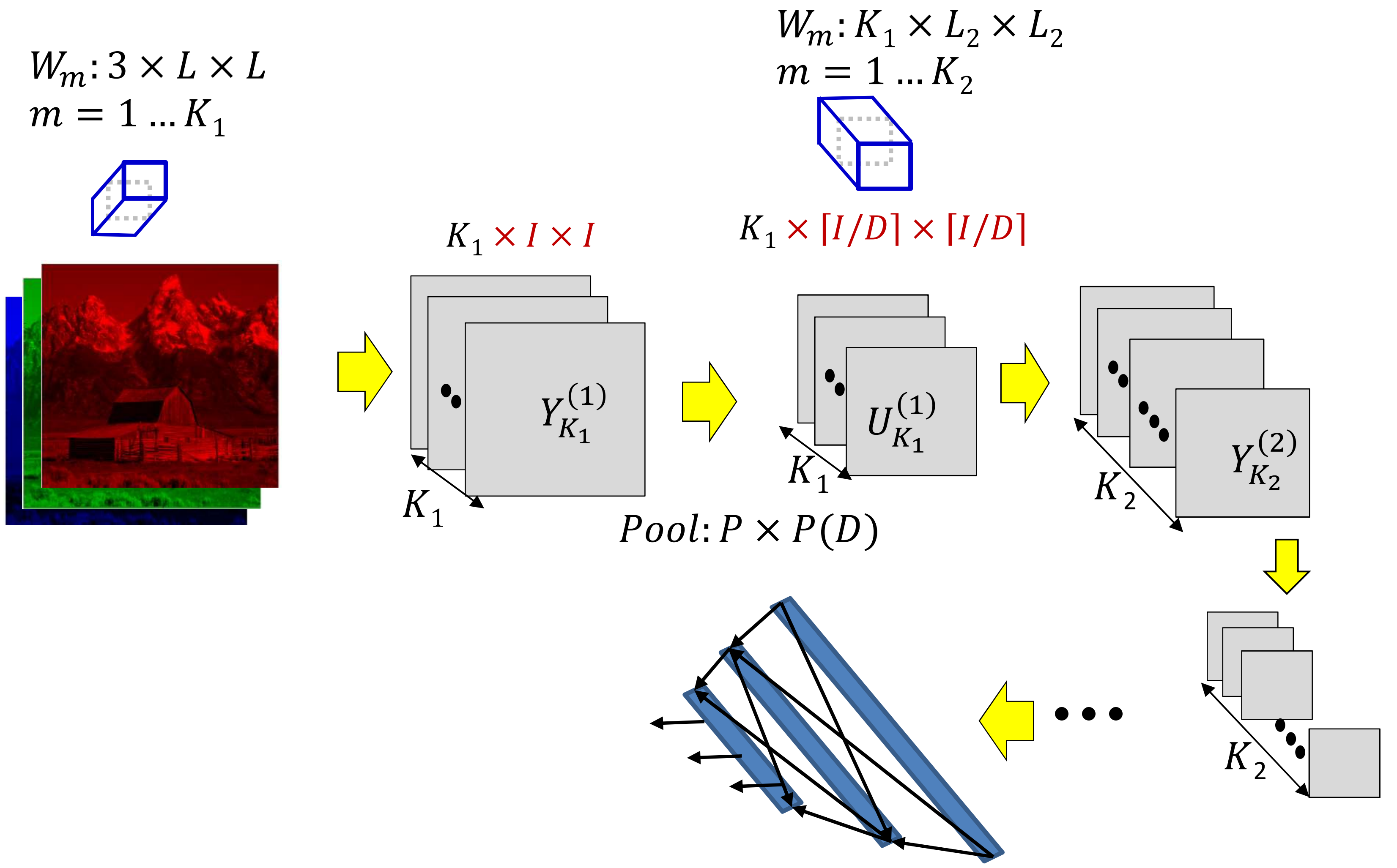
$$P_m^{(n)}(i,j) = \underset{\substack{k \in \{(i-1)d+1,\, id\}, \\ l \in \{(j-1)d+1,\, jd\}}}{\operatorname{argmax}} \; Y_m^{(n)}(k, l$$

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

Parameters to choose:
Size of pooling block $P_2$
Pooling stride $D_2$

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

# Convolutional Neural Networks



- **This continues for several layers until the final convolved output is fed to an MLP**

## The Size of the Layers

- Each convolution layer maintains the size of the image
  - With appropriate zero padding
  - If performed *without* zero padding it will decrease the size of the input

- Each convolution layer may *increase* the *number* of maps from the previous layer

- Each pooling layer with hop $D$ *decreases* the *size* of the maps by a factor of $D$

- Filters within a layer must all be the same size, but sizes may vary with layer
  - Similarly for pooling, $D$ may vary with layer

- In general the number of convolutional filters increases with layers

## Parameters to choose (design choices)

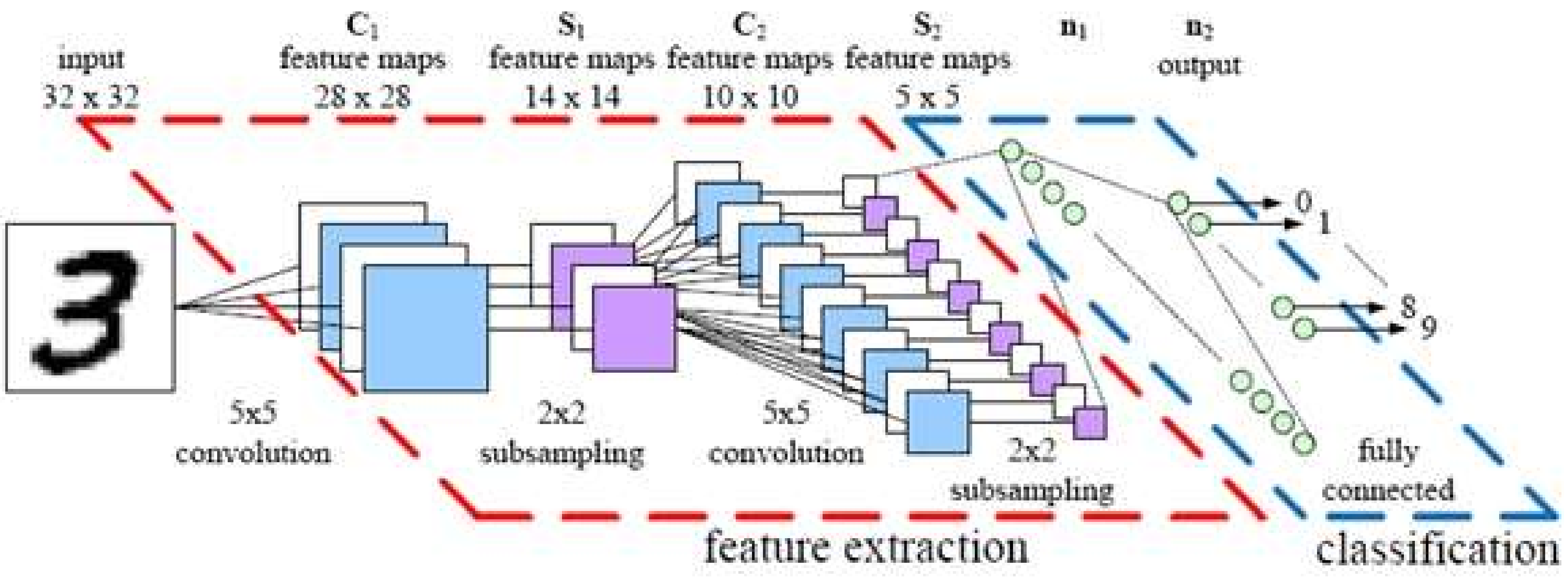
- Number of convolutional and downsampling layers
  - And arrangement (order in which they follow one another)

- For each convolution layer:
  - Number of filters $K_i$
  - Spatial extent of filter $L_i \times L_i$
    - The "depth" of the filter is fixed by the number of filters in the previous layer $K_{i-1}$
  - The stride $S_i$

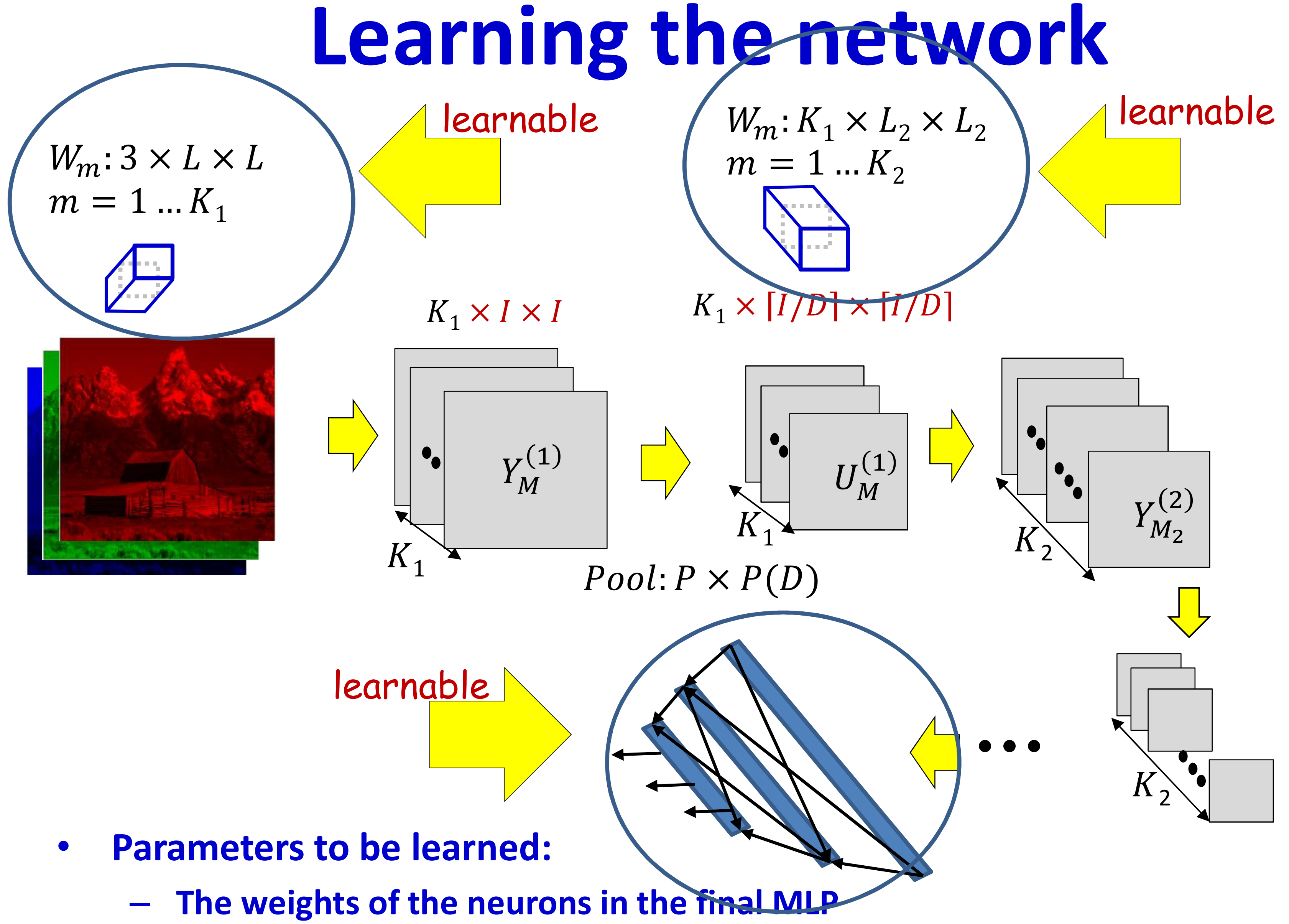
- For each downsampling/pooling layer:
  - Spatial extent of filter $P_i \times P_i$
  - The stride $D_i$

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

## Digit classification



# Learning the network



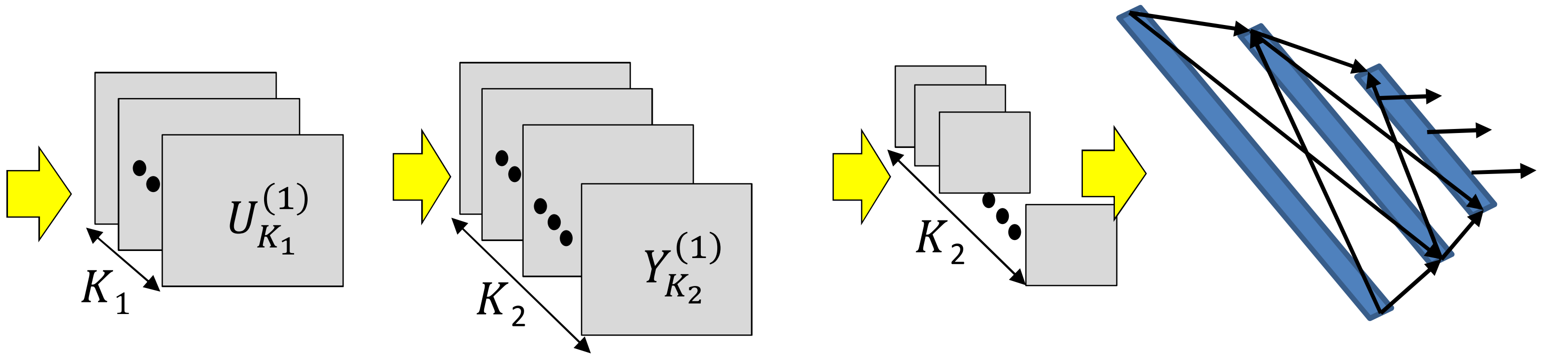
- **Parameters to be learned:**
  - **The weights of the neurons in the final MLP**
  - **The (weights and biases of the) filters for every *convolutional* layer**

# Learning the CNN

- In the final "flat" multi-layer perceptron, all the weights and biases of each of the perceptrons must be learned

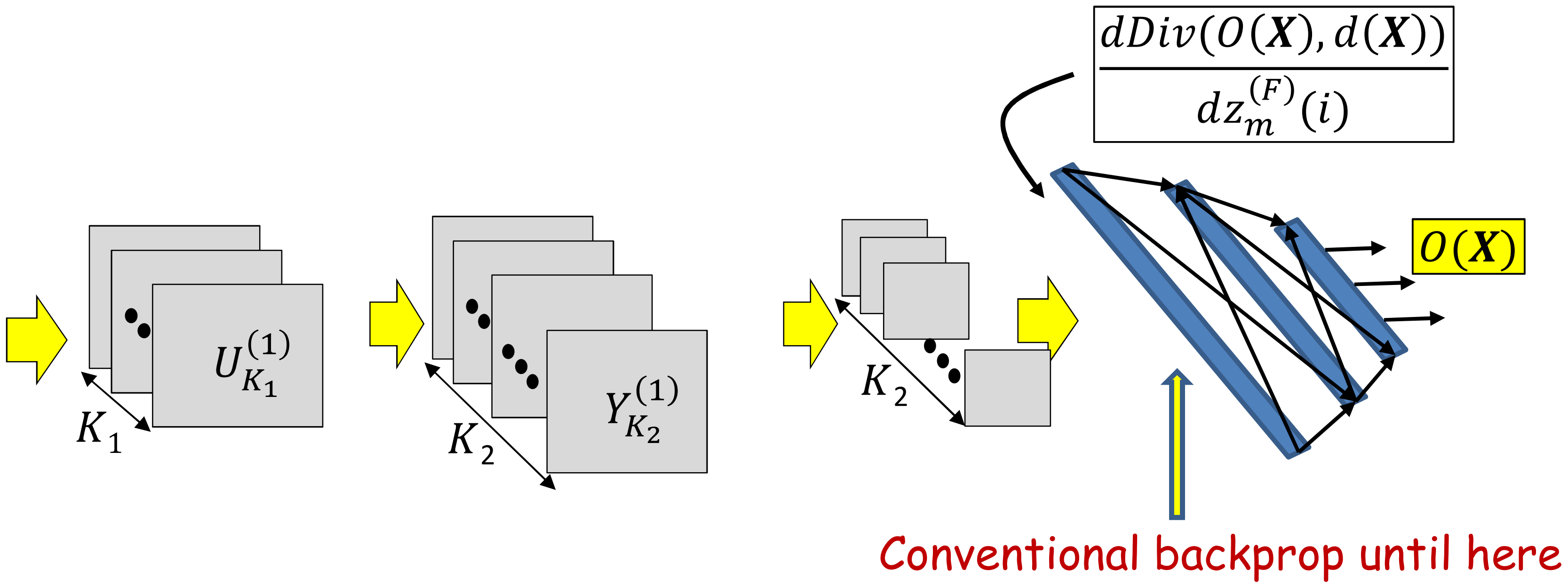
- In the *convolutional layers* the filters must be learned
- Let each layer $J$ have $K_J$ maps
  - $K_0$ is the number of maps (colours) in the input
- Let the filters in the $J^{\text{th}}$ layer be size $L_J \times L_J$
- For the $J^{\text{th}}$ layer we will require $K_J\left(K_{J-1}L_J^2 + 1\right)$ filter parameters
- Total parameters required for the *convolutional* layers: $\sum_{J \in convolutional\ layers} K_J\left(K_{J-1}L_J^2 + 1\right)$

# Training



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

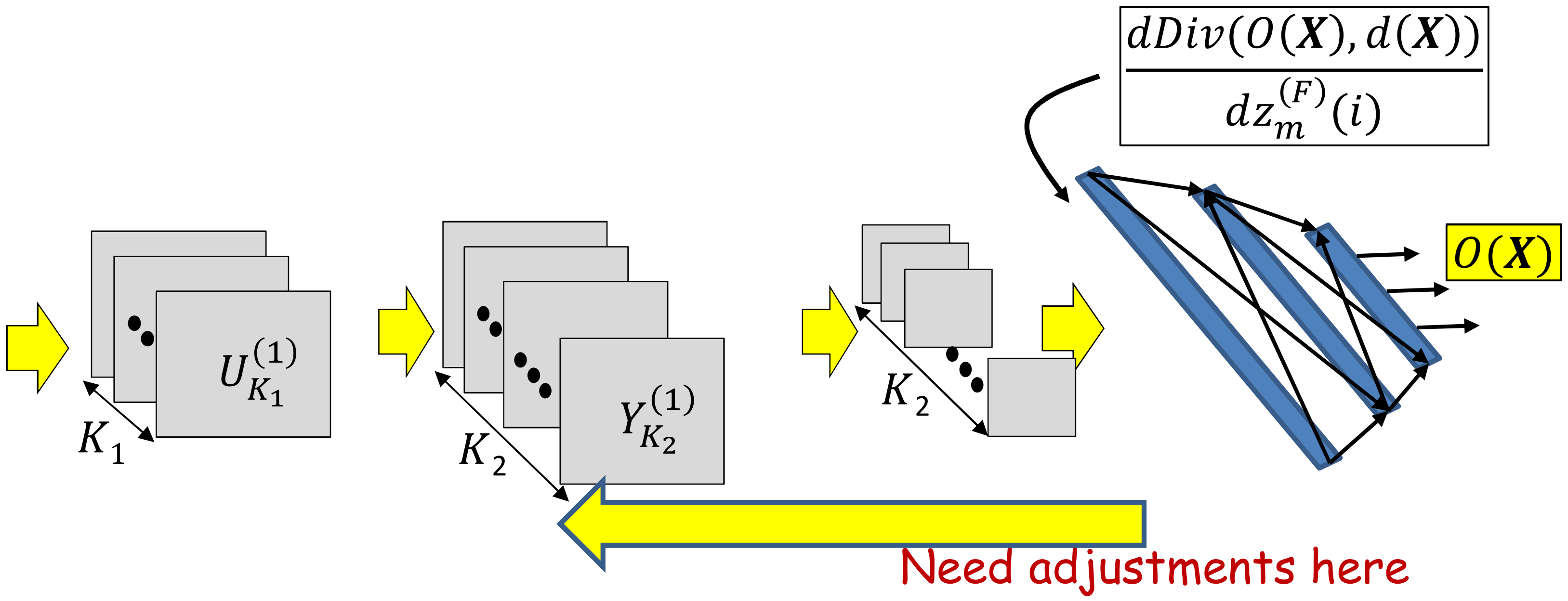
# Backpropagation: Final flat layers



$\frac{dDiv(O(\boldsymbol{X}), d(\boldsymbol{X}))}{dz_m^{(F)}(i)}$
$U_{K_1}^{(1)}$
$K_1$
$Y_{K_2}^{(1)}$
$K_2$
$K_2$
$O(\boldsymbol{X})$
Conventional backprop until here

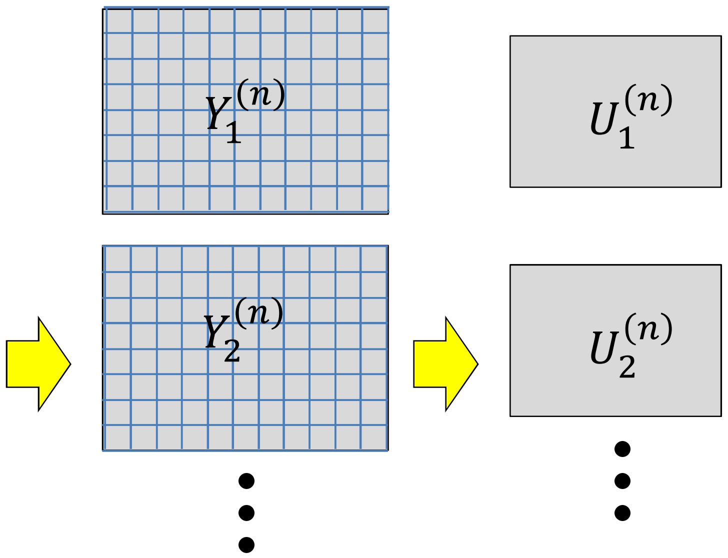
- Backpropagation continues in the usual manner until the computation of the derivative of the divergence w.r.t the inputs to the first "flat" layer
  - Important to recall: the first flat layer is only the "unrolling" of the maps from the final convolutional layer

# Backpropagation: Final flat layers



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

# Backpropagation: Maxout layers

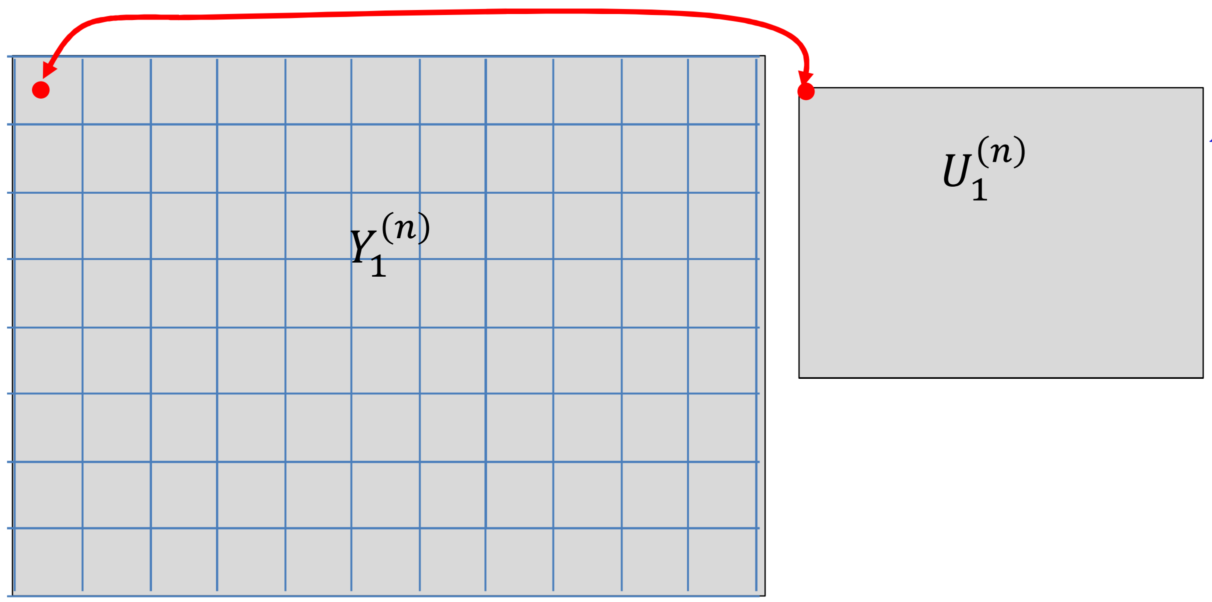


$$P_m^{(n)}(i,j) = \underset{\substack{k\in\{(i-1)d+1,\, id\},\\ l\in\{(j-1)d+1,\, jd\}}}{\operatorname{argmax}} Y_m^{(n)}(k,l)$$

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

- The derivative w.r.t $U_m^{(n)}(i,j)$ can be computed via backprop
- But this cannot be propagated backwards to compute the derivative w.r.t. $Y_m^{(n)}(k,l)$
- **Max and argmax are not differentiable**

# Backpropagation: Maxout layers



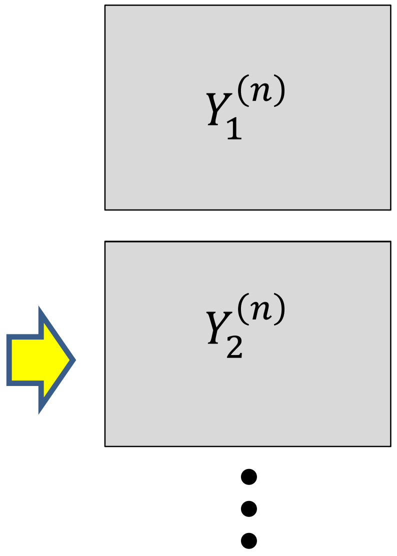
$$P_m^{(n)}(i,j) = \underset{\substack{k\in\{(i-1)d+1,\, id\},\\ l\in\{(j-1)d+1,\, jd\}}}{\operatorname{argmax}} Y_m^{(n)}(k,l)$$

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

$$\boxed{\frac{dDiv(O(\boldsymbol{X}),d(\boldsymbol{X}))}{dY_m^{(n)}(k,l)} = \begin{cases} \dfrac{dDiv(O(\boldsymbol{X}),d(\boldsymbol{X}))}{dU_m^{(n)}(i,j)} & if\ (k,l) = P_m^{(n)}(i,j) \\ 0\ otherwise \end{cases}}$$

- Approximation: Derivative w.r.t the $Y$ terms that did not contribute to the maxout map is 0

## Backpropagation: Weights



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

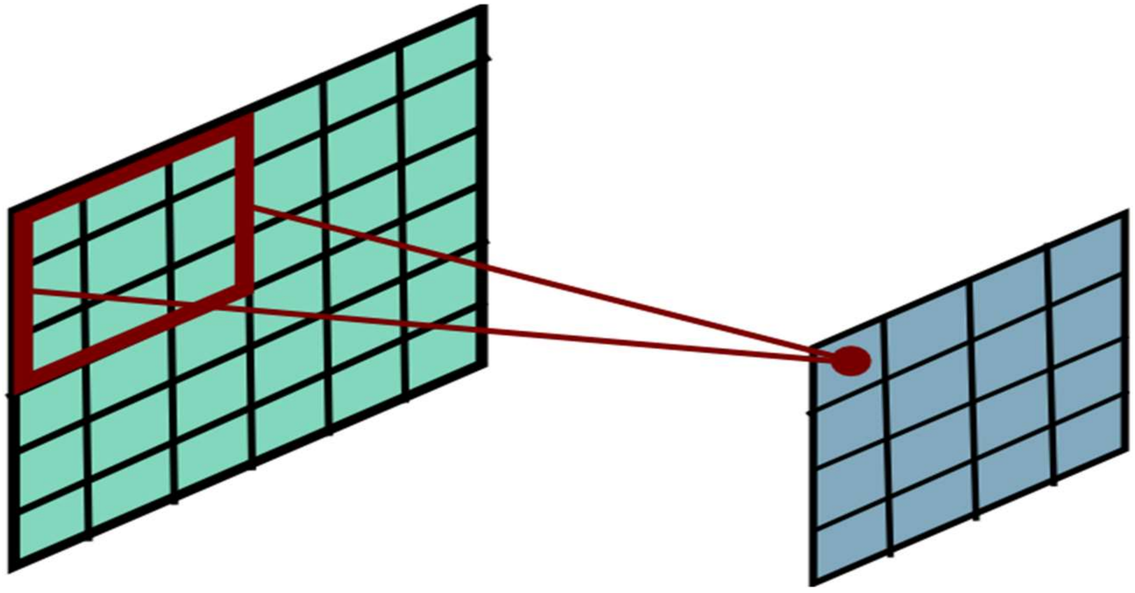
$$Y_m^{(n)}(i,j) = f\left(z_m^{(n)}(i,j)\right)$$

- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative of the weight**
  - **Shared parameter updates**
    - **Look at slides..**

# Convolutional Layer

$$z(i,j) = \sum_{k=1}^{K} \sum_{l=1}^{L} w(k,l) I(i+l, j+k)$$

$$y(i,j) = f(z(i,j))$$



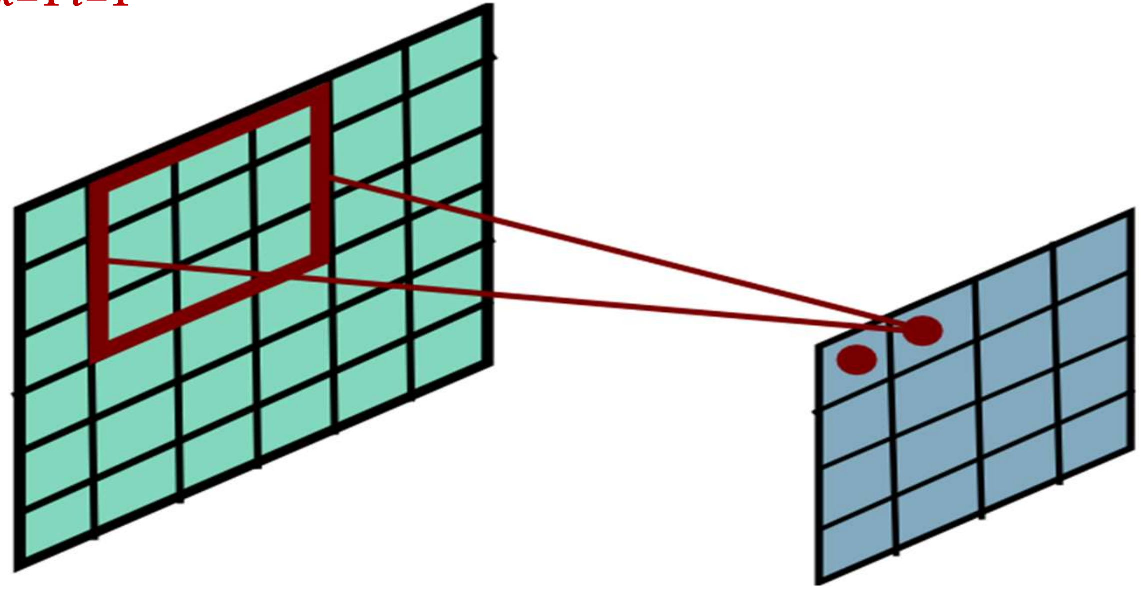
- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative of the weight**

Ranzato

# Convolutional Layer

$$z(i,j) = \sum_{k=1}^{K} \sum_{l=1}^{L} w(k,l) I(i+l, j+k)$$

$$y(i,j) = f(z(i,j))$$

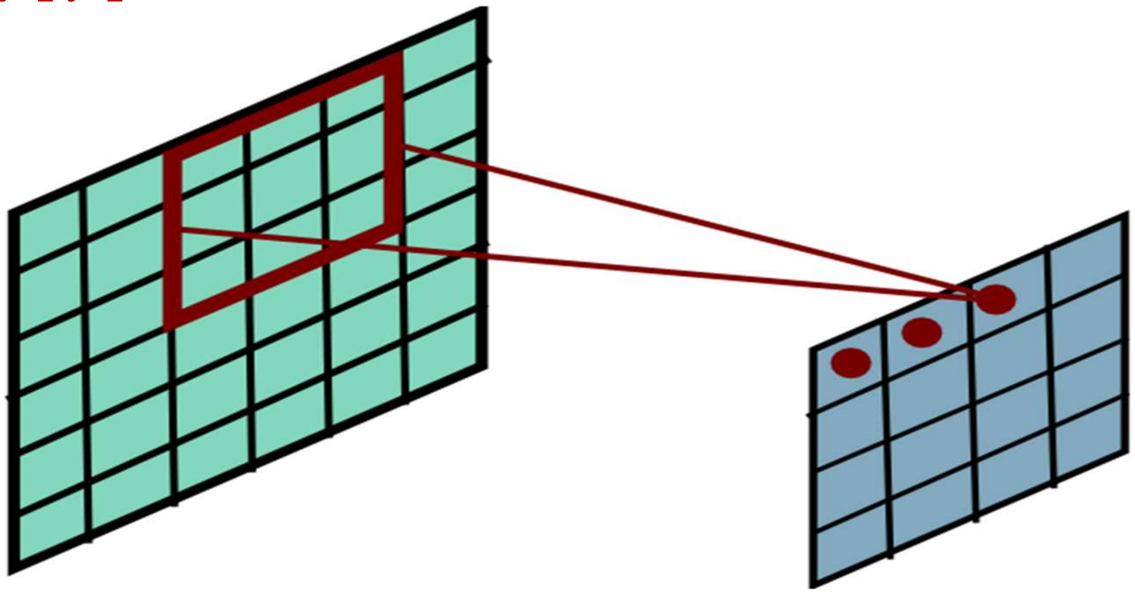


- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative of the weight**

Ranzato

## Convolutional Layer

$$z(i,j) = \sum_{k=1}^{K}\sum_{l=1}^{L} w(k,l) I(i+l, j+k) \qquad y(i,j) = f(z(i,j))$$



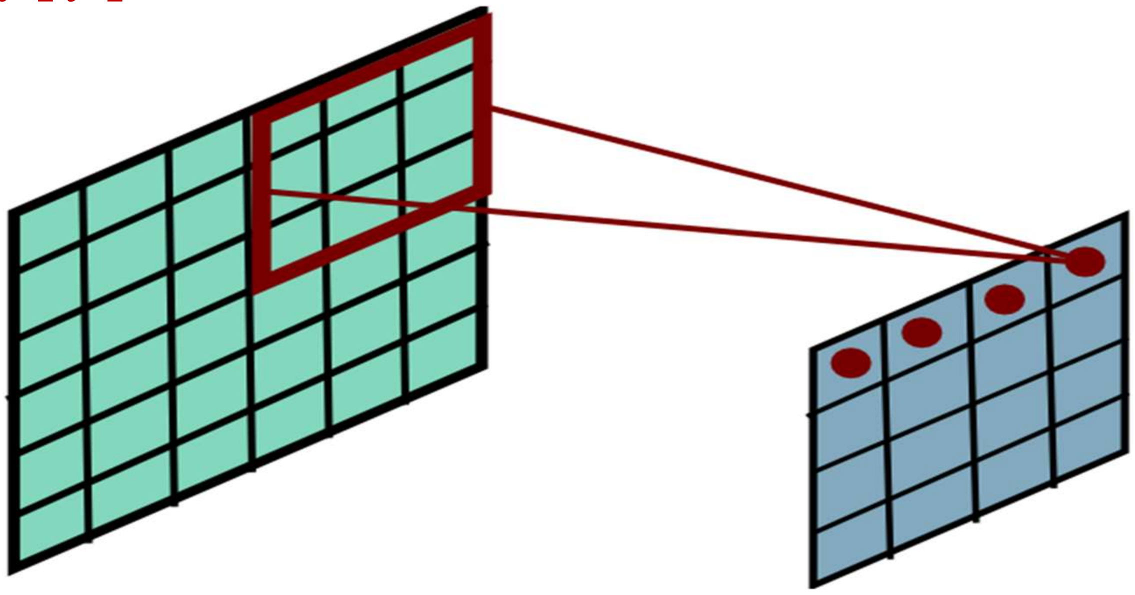
- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative of the weight**

Ranzato

# Convolutional Layer

$$z(i,j) = \sum_{k=1}^{K}\sum_{l=1}^{L} w(k,l) I(i+l, j+k)$$

$$y(i,j) = f(z(i,j))$$

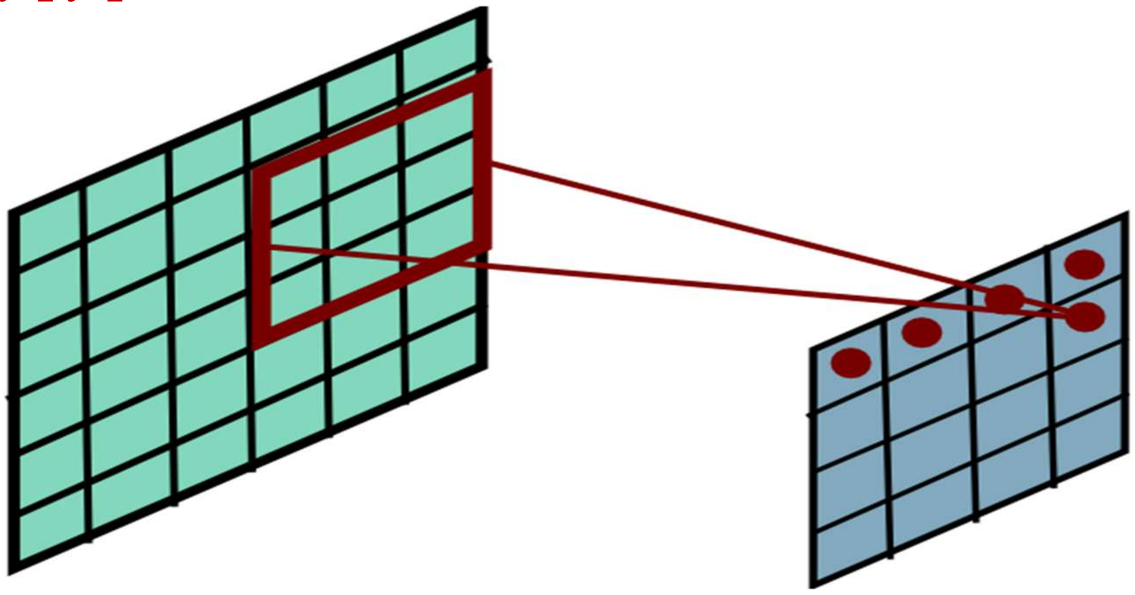


- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative of the weight**

Ranzato

# Convolutional Layer

$$z(i,j) = \sum_{k=1}^{K} \sum_{l=1}^{L} w(k,l) I(i+l, j+k) \qquad y(i,j) = f(z(i,j))$$



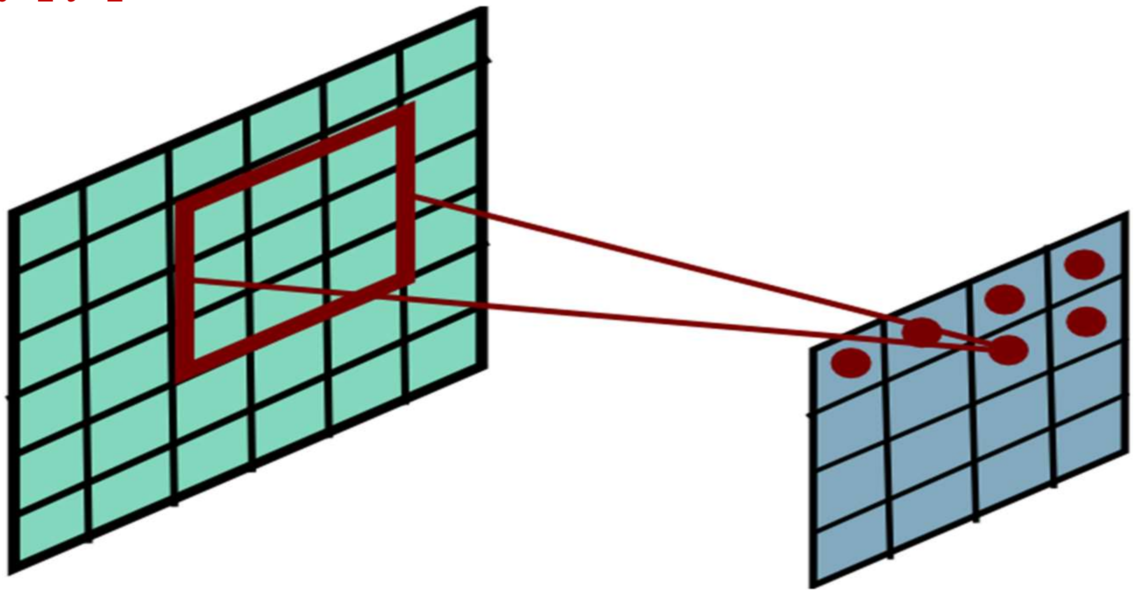
- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative of the weight**

**Ranzato**

# Convolutional Layer

$$\boldsymbol{z(i,j) = \sum_{k=1}^{K}\sum_{l=1}^{L} w(k,l) I(i+l, j+k)}$$

$$\boldsymbol{y(i,j) = f(z(i,j))}$$

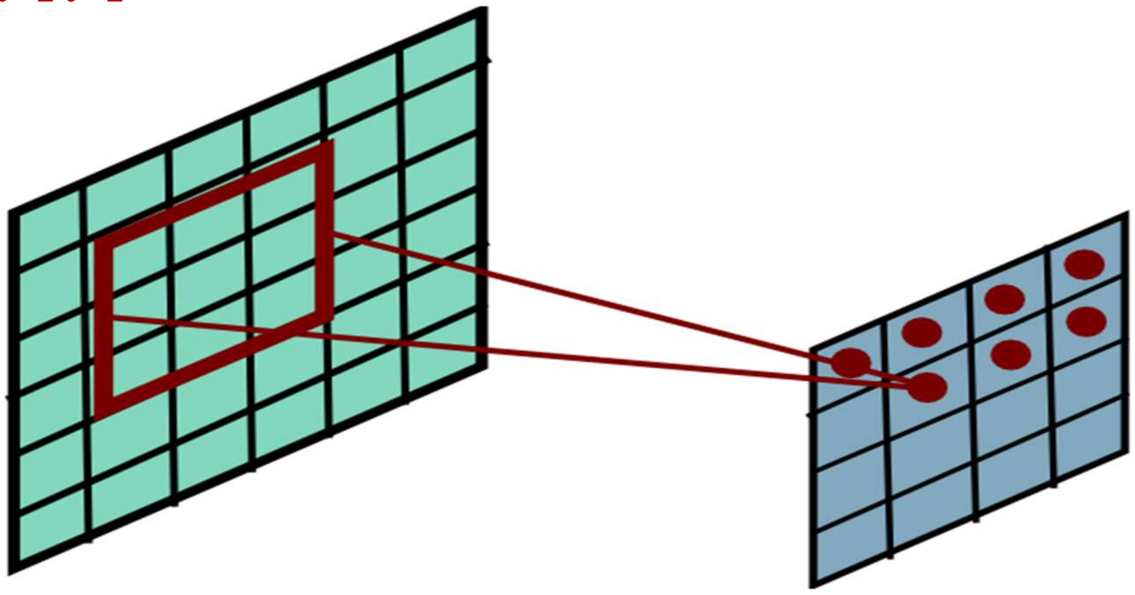


- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative of the weight**

**Ranzato**

# Convolutional Layer

$$z(i,j) = \sum_{k=1}^{K}\sum_{l=1}^{L} w(k,l) I(i+l, j+k) \qquad y(i,j) = f(z(i,j))$$

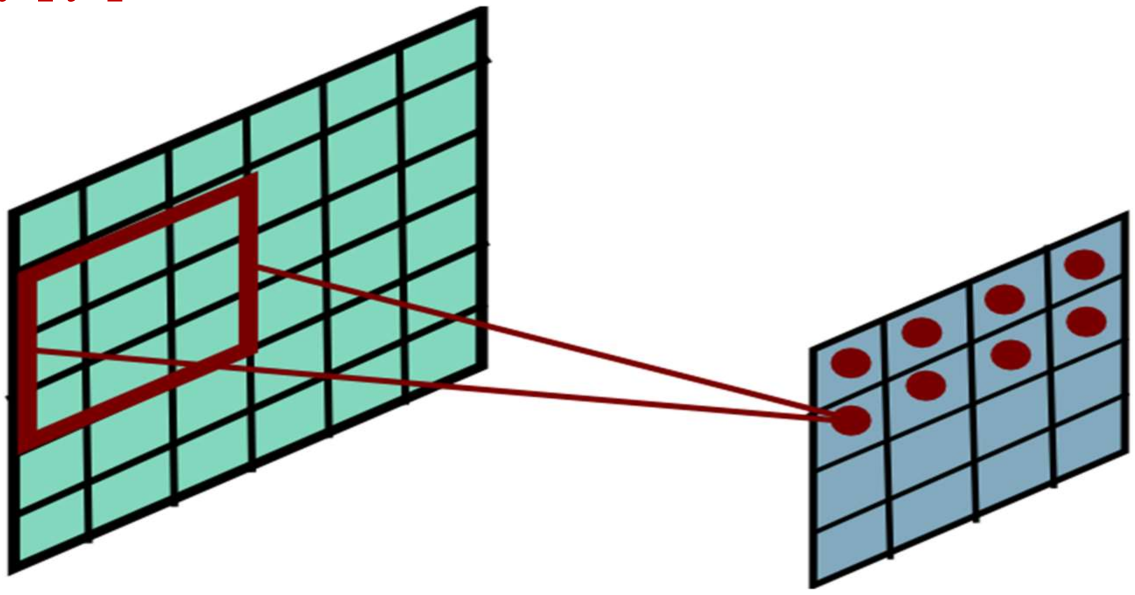


- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative of the weight**

Ranzato

## Convolutional Layer

$$z(i,j) = \sum_{k=1}^{K} \sum_{l=1}^{L} w(k,l) I(i+l, j+k) \qquad y(i,j) = f(z(i,j))$$

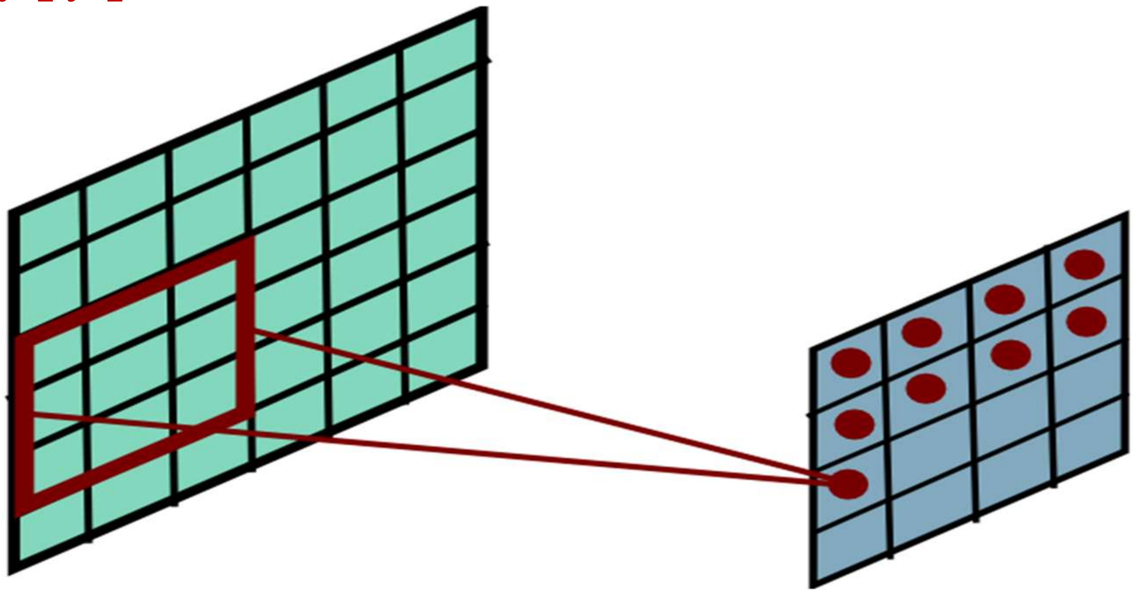


- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative of the weight**

**Ranzato**

# Convolutional Layer

$$z(i,j) = \sum_{k=1}^{K} \sum_{l=1}^{L} w(k,l) I(i+l, j+k) \qquad y(i,j) = f(z(i,j))$$

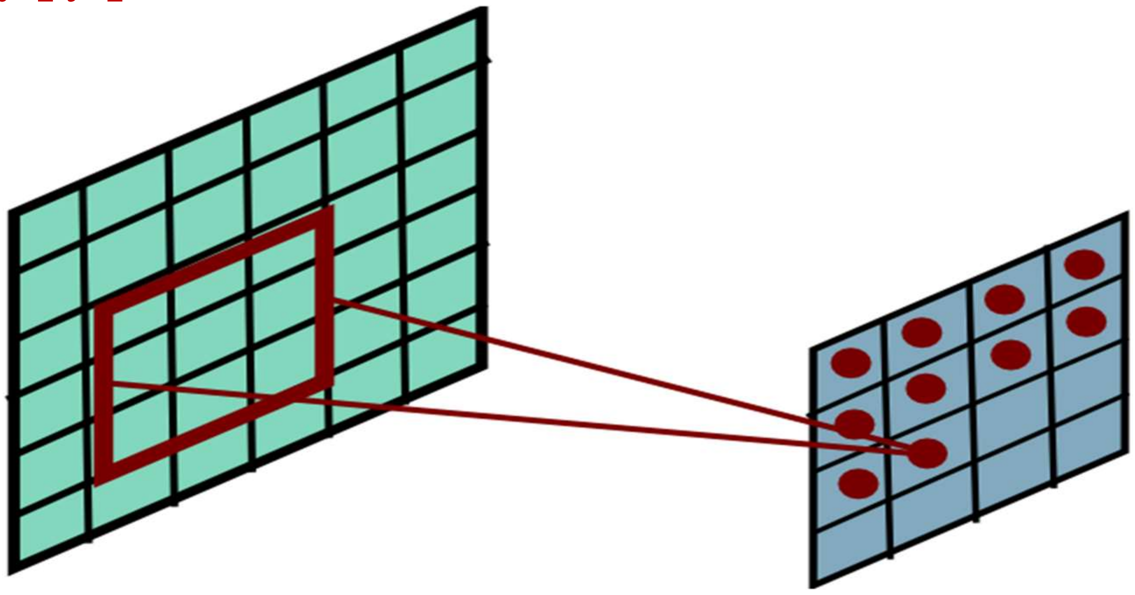


- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative of the weight**

Ranzato

# Convolutional Layer

$$z(i,j) = \sum_{k=1}^{K} \sum_{l=1}^{L} w(k,l) I(i+l, j+k) \qquad y(i,j) = f(z(i,j))$$

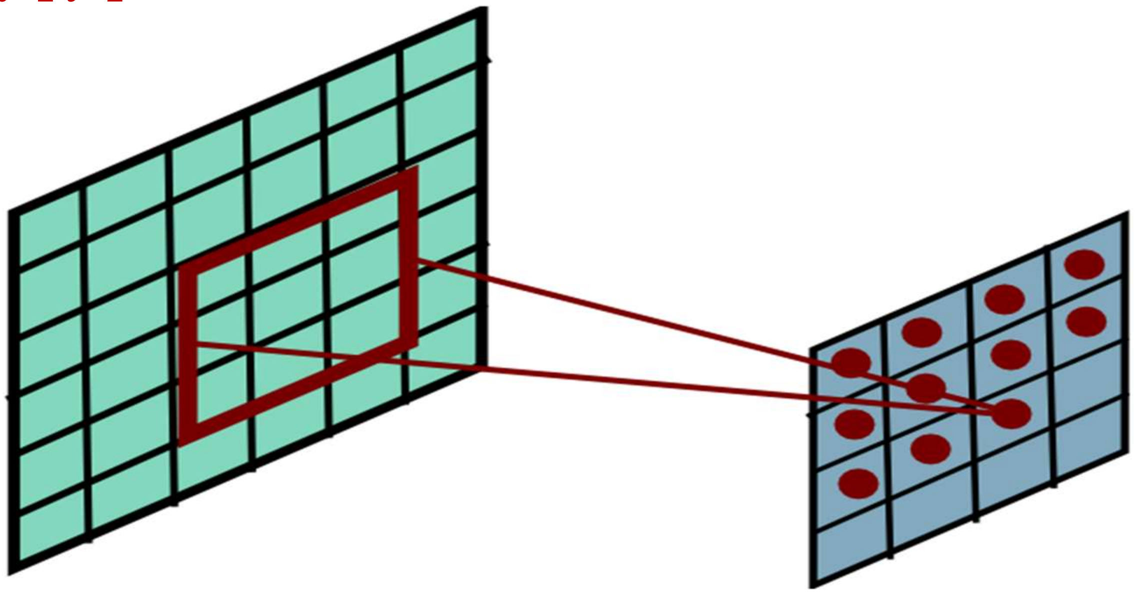


- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative of the weight**

Ranzato

# Convolutional Layer

$$z(i,j) = \sum_{k=1}^{K} \sum_{l=1}^{L} w(k,l) I(i+l, j+k) \qquad y(i,j) = f(z(i,j))$$

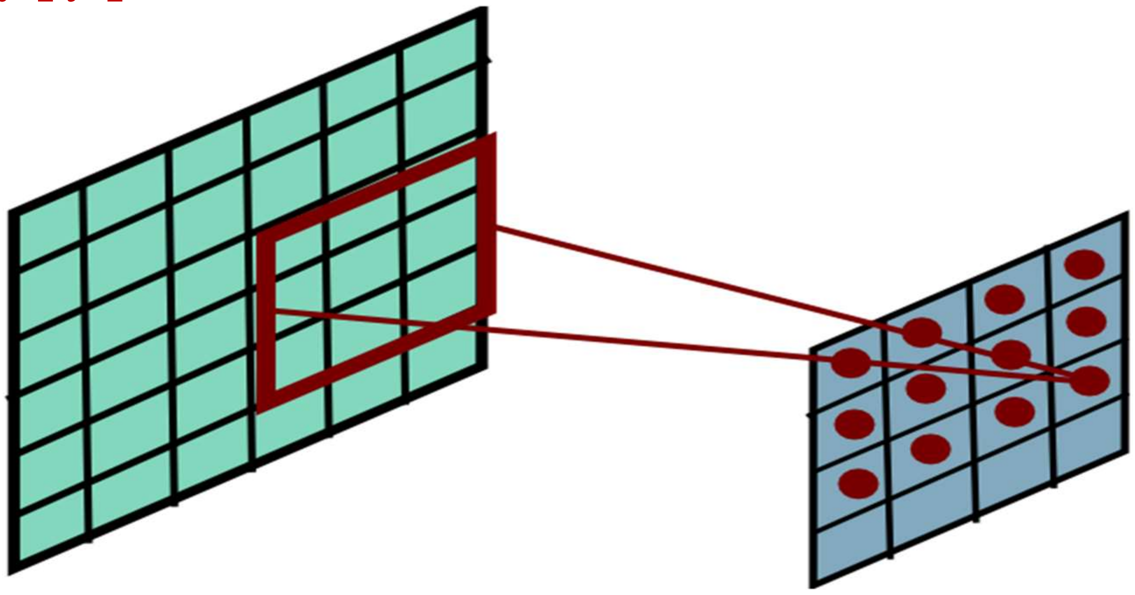


- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative of the weight**

Ranzato

## Convolutional Layer

$$z(i,j) = \sum_{k=1}^{K} \sum_{l=1}^{L} w(k,l) I(i+l, j+k) \qquad y(i,j) = f(z(i,j))$$

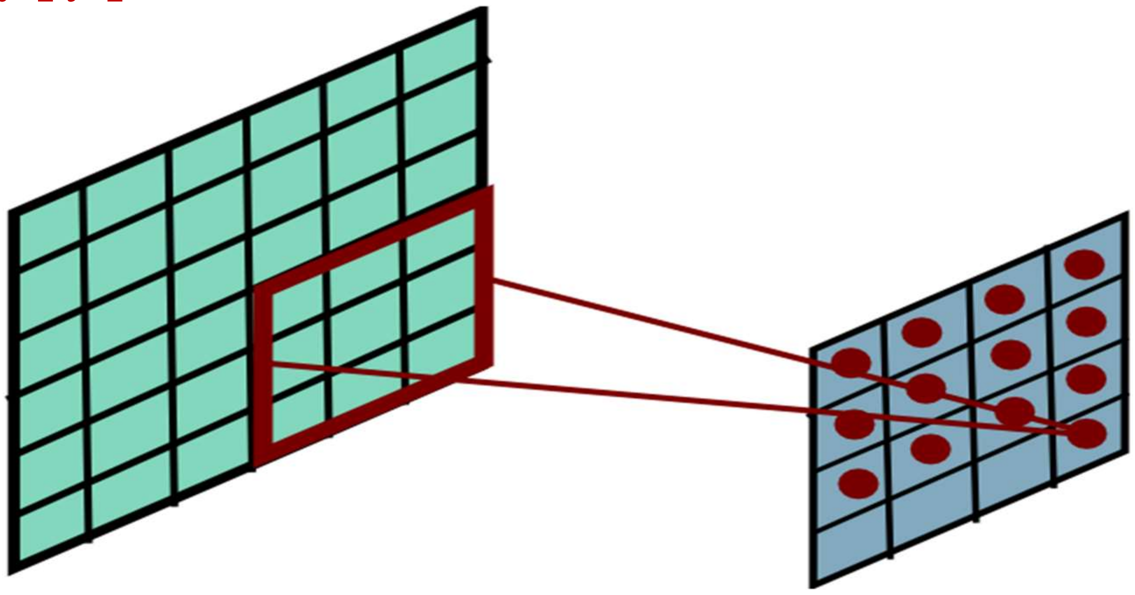


- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative of the weight**

Ranzato

# Convolutional Layer

$$z(i,j) = \sum_{k=1}^{K}\sum_{l=1}^{L} w(k,l)I(i+l,j+k) \qquad y(i,j) = f(z(i,j))$$

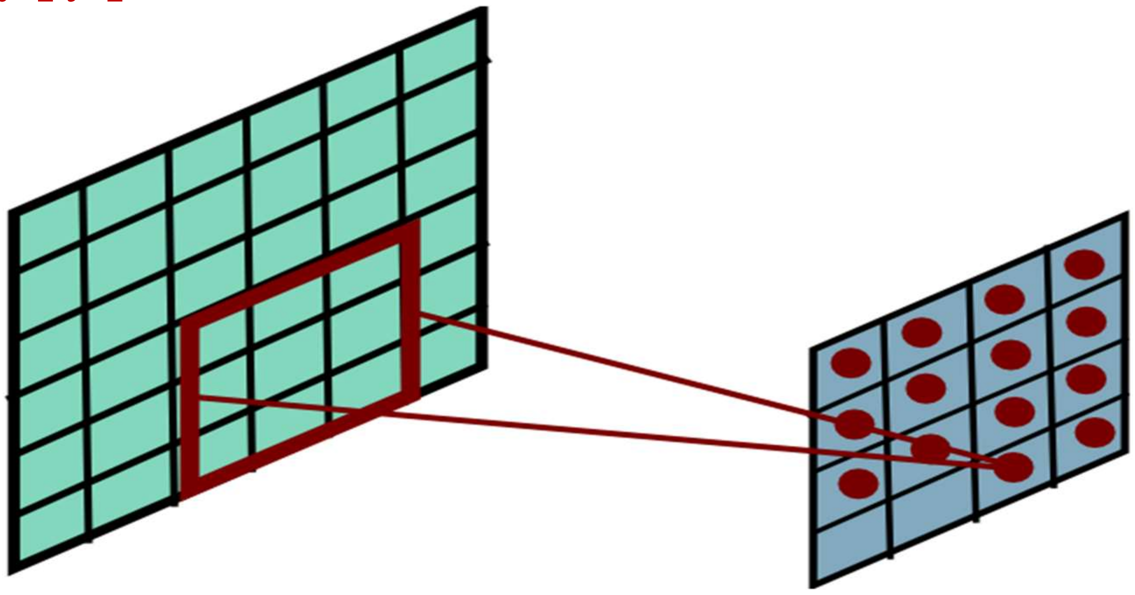


- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative of the weight**

Ranzato

# Convolutional Layer

$$z(i,j) = \sum_{k=1}^{K} \sum_{l=1}^{L} w(k,l) I(i+l, j+k) \qquad y(i,j) = f(z(i,j))$$

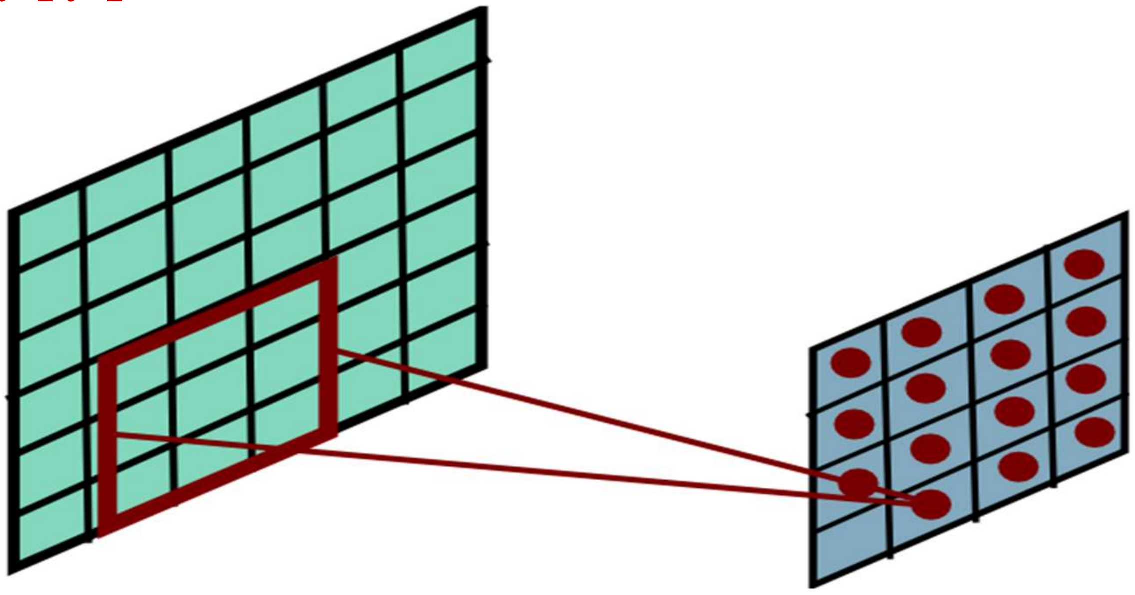


- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative of the weight**

Ranzato

# Convolutional Layer

$$z(i,j) = \sum_{k=1}^{K} \sum_{l=1}^{L} w(k,l) I(i+l, j+k) \qquad y(i,j) = f(z(i,j))$$



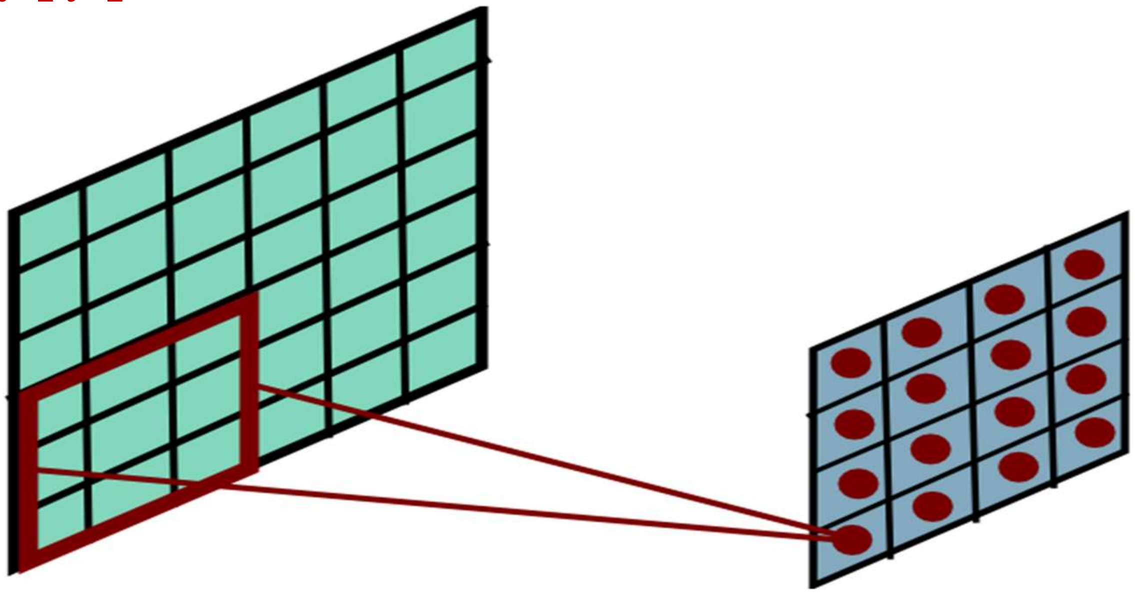
- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative of the weight**

Ranzato

# Convolutional Layer

$$z(i,j) = \sum_{k=1}^{K}\sum_{l=1}^{L} w(k,l)I(i+l, j+k)$$

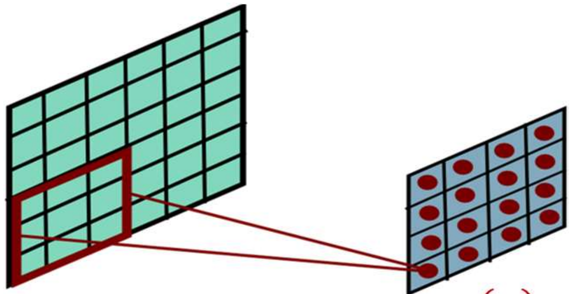
$$y(i,j) = f(z(i,j))$$



- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative of the weight**

Ranzato

# Backpropagation: Weights

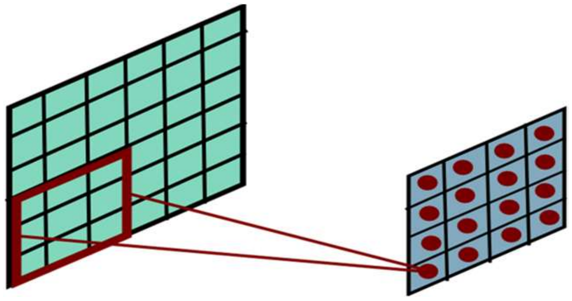


$$z_m^{(n)}(i,j) = \sum_{r=1}^{M_{n-1}} \sum_{k=1}^{L_n} \sum_{l=1}^{L_n} w_m^{(n)}(r,k,l) U_r^{(n-1)}(i+k, j+l) + b_m^{(n)}$$

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

- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative of the weight**

# Backpropagation: Weights



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

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

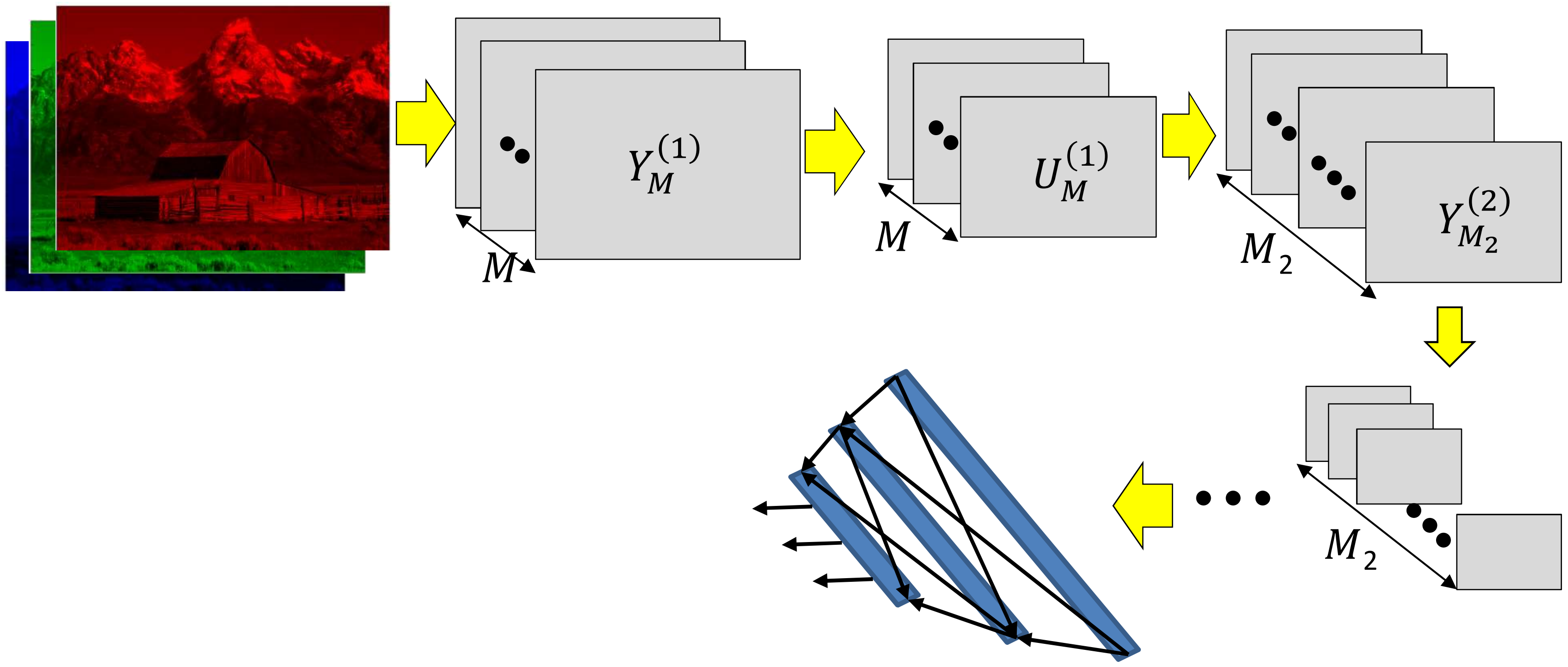
$$\frac{dDiv(O(\boldsymbol{X}), d(\boldsymbol{X}))}{dw_m^{(n)}(r,k,l)} = \sum_{i,j} \frac{dz_m^{(n)}(i,j)}{dw_m^{(n)}(r,k,l)} \frac{dY_m^{(n)}(i,j)}{dz_m^{(n)}(i,j)} \frac{dDiv(O(\boldsymbol{X}), d(\boldsymbol{X}))}{dY_m^{(n)}(i,j)}$$

$$\boxed{\frac{dDiv(O(\boldsymbol{X}), d(\boldsymbol{X}))}{dw_m^{(n)}(r,k,l)} = \sum_{i,j} U_r^{(n-1)}(i+k, j+l) f'\left(z_m^{(n)}(i,j)\right) \frac{dDiv(O(\boldsymbol{X}), d(\boldsymbol{X}))}{dY_m^{(n)}(i,j)}}$$

$$\boxed{\frac{dDiv(O(\boldsymbol{X}), d(\boldsymbol{X}))}{dU_r^{(n-1)}(k,l)} = \sum_{m} \sum_{i,j} w_m^{(n)}(r,i,j) f'\left(z_m^{(n)}(k-i, l-j)\right) \frac{dDiv(O(\boldsymbol{X}), d(\boldsymbol{X}))}{dY_m^{(n)}(k-i, l-j)}}$$

- Note: each weight contributes to *every* position in the map at the output of the convolutional layer
- **Every position will contribute to the derivative w.r.t that weight**
- Each input U also contributes to many positions on each of the maps
- All of them will contribute to the derivative w.r.t that U

## 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

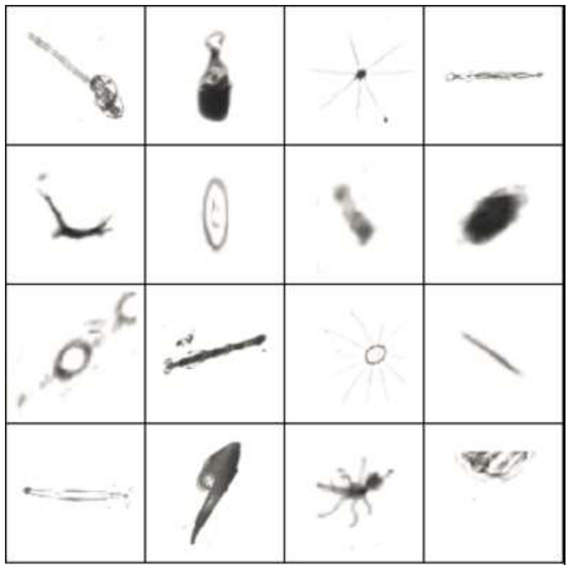
## Training Issues

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

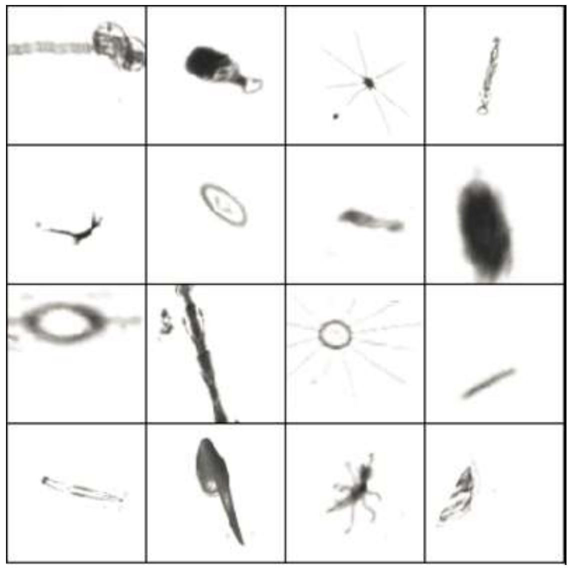
- The number of parameters can quickly become very large
- Insufficient training data to train well
  - Solution: Data augmentation

## Data Augmentation

Original data



Augmented data



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

## Other tricks

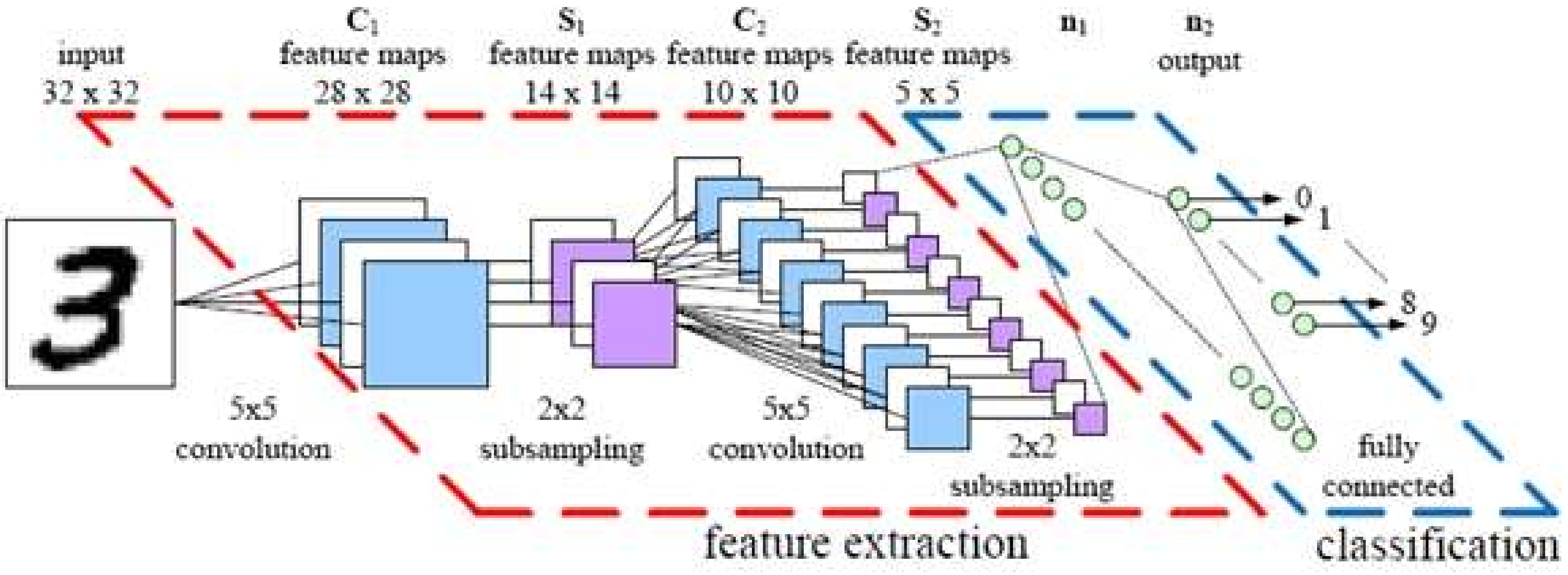
- *Very deep* networks
  - *100* or more layers in MLP
  - Formalism called “Resnet”

## Convolutional neural nets

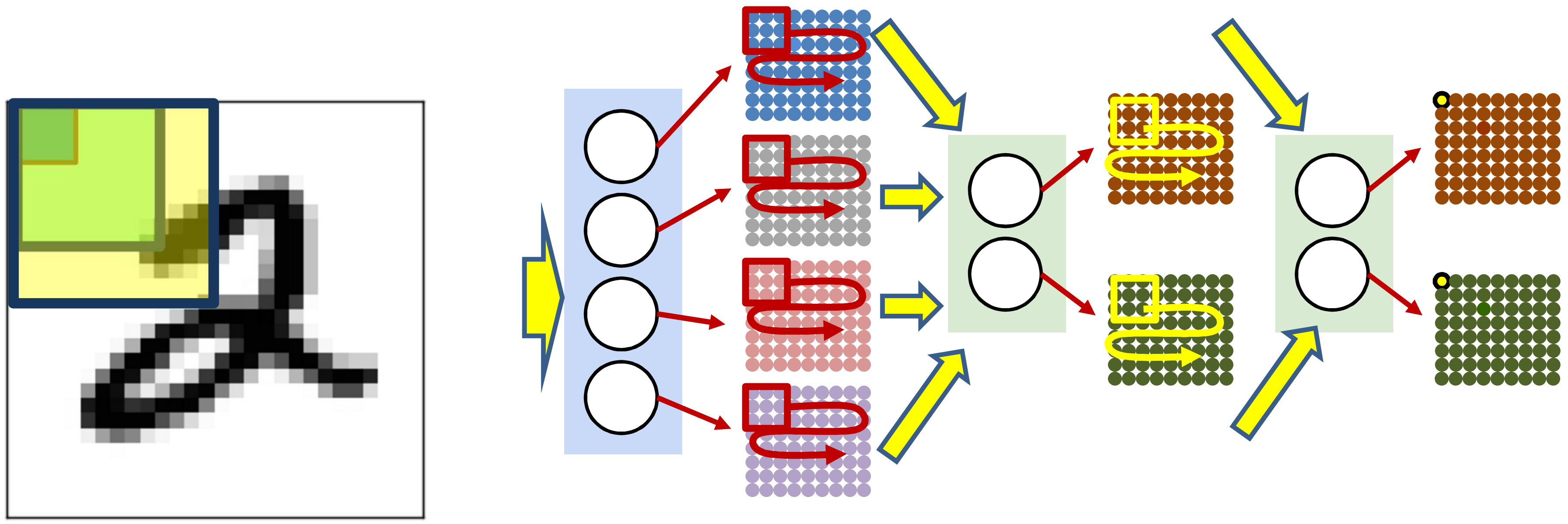
- One of *the* most frequently used nnet formalism today

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

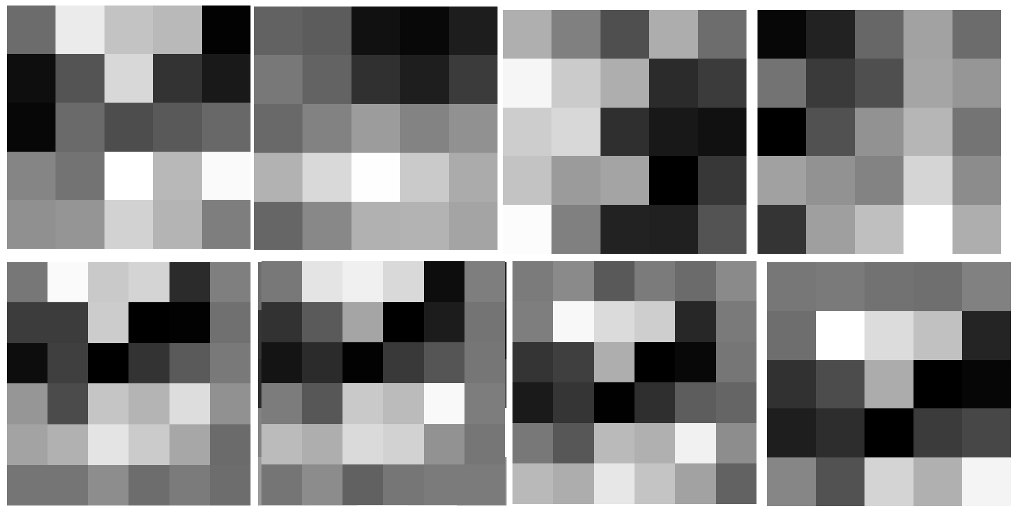
## Digit classification

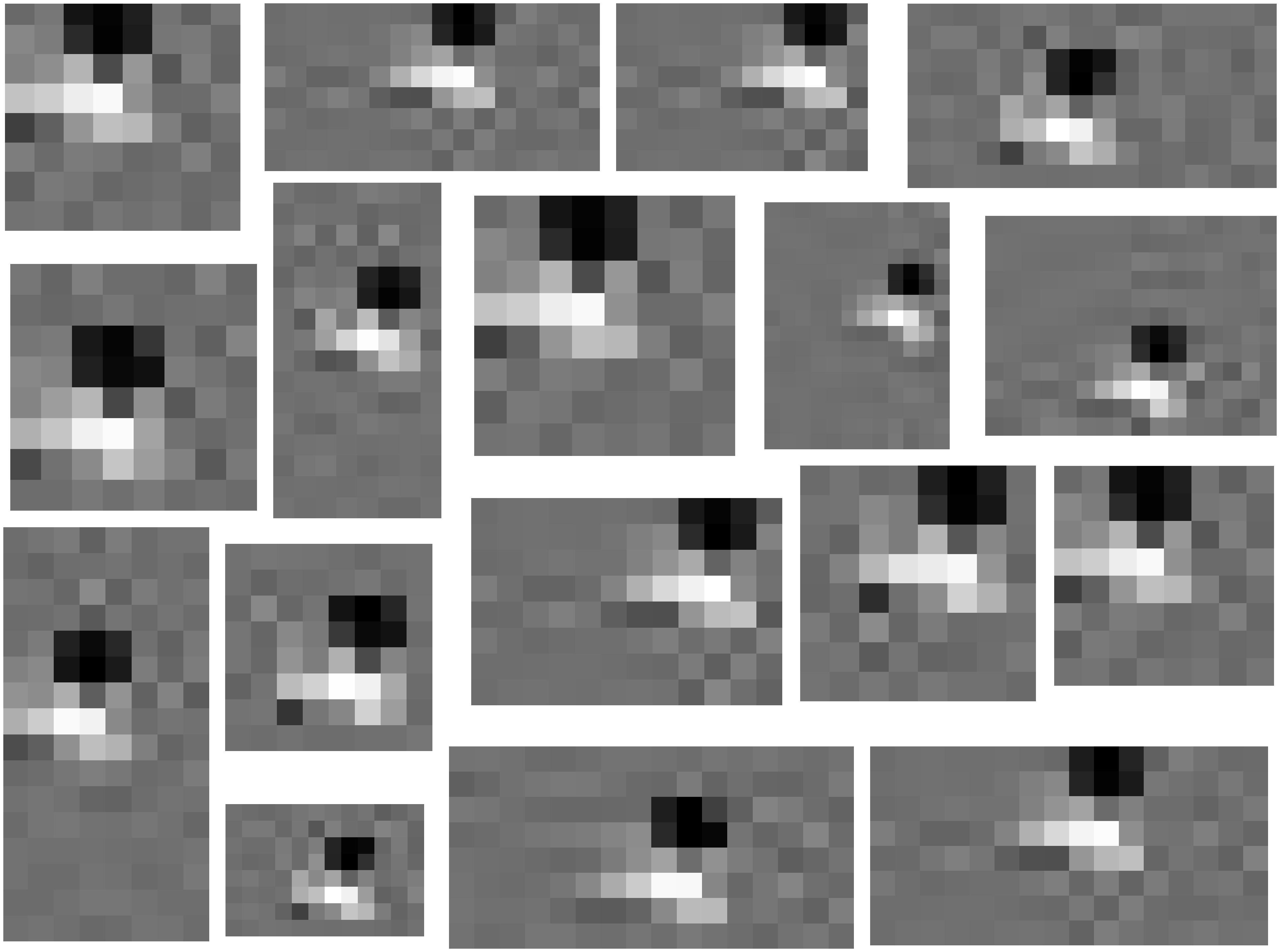


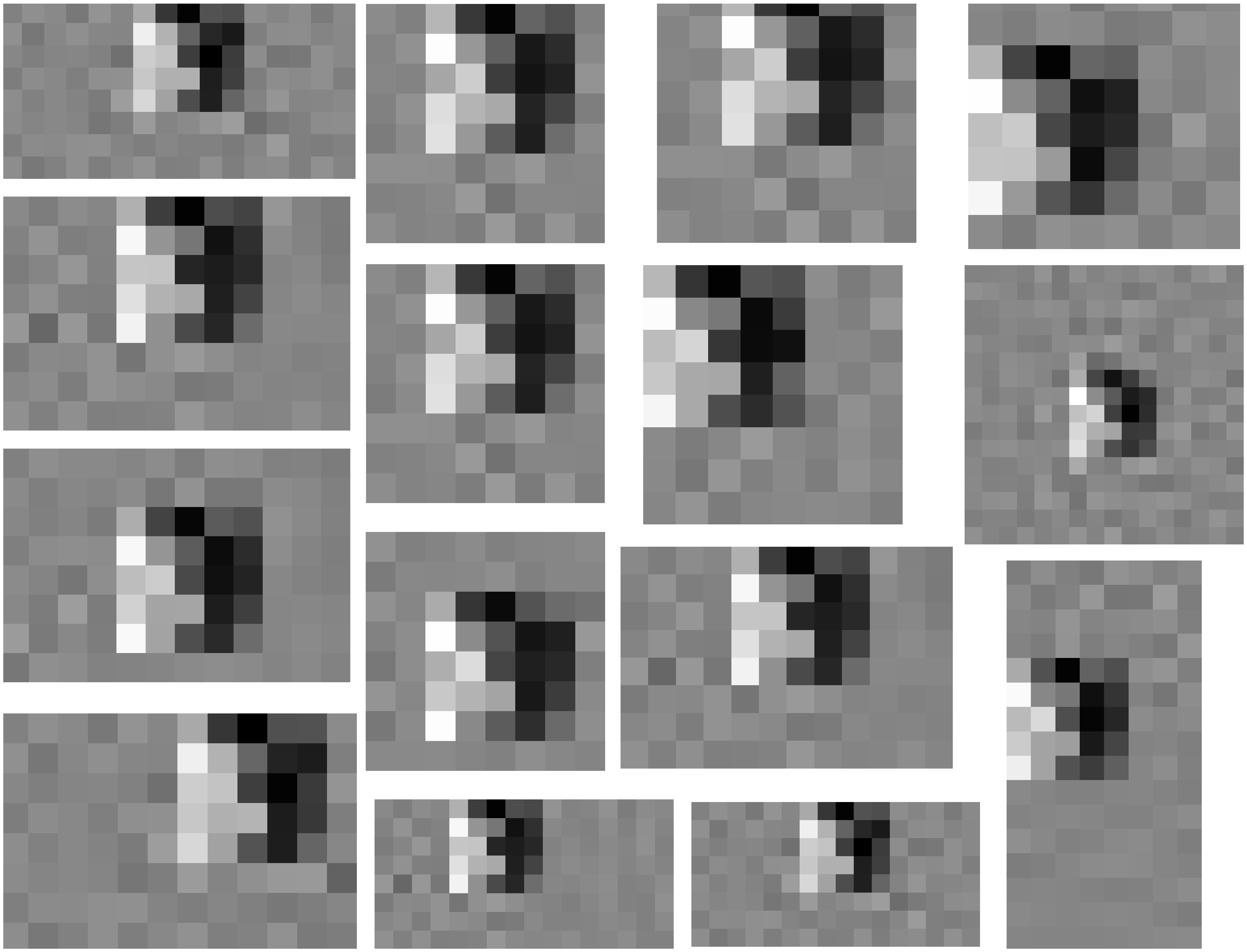
# Receptive fields

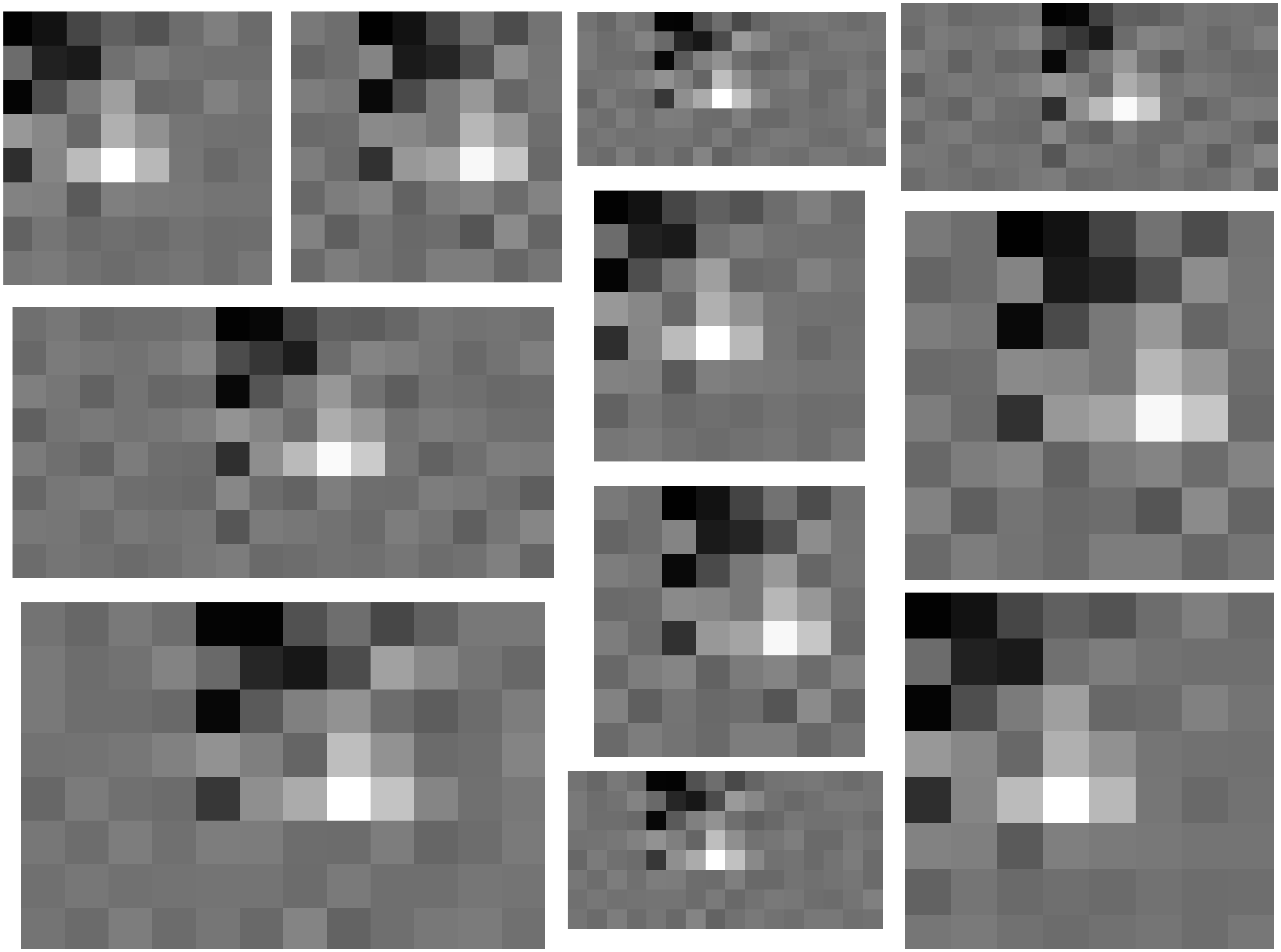


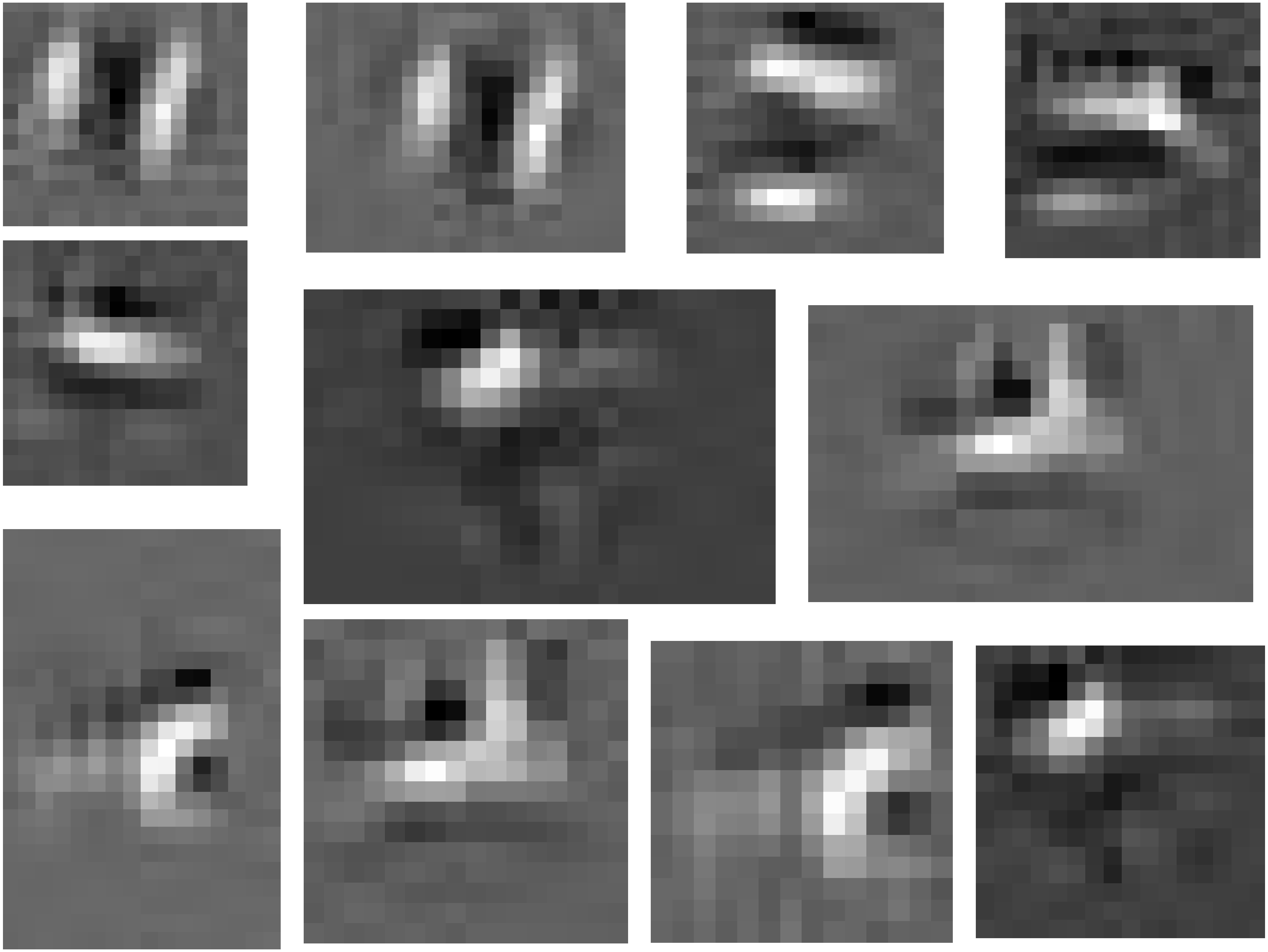
- The pattern in the *input* image that each neuron sees is its "Receptive Field"
- The receptive field for a first layer neurons is simply its arrangement of weights
- For the higher level neurons, the actual receptive field is not immediately obvious and must be *calculated*
  - What patterns in the 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

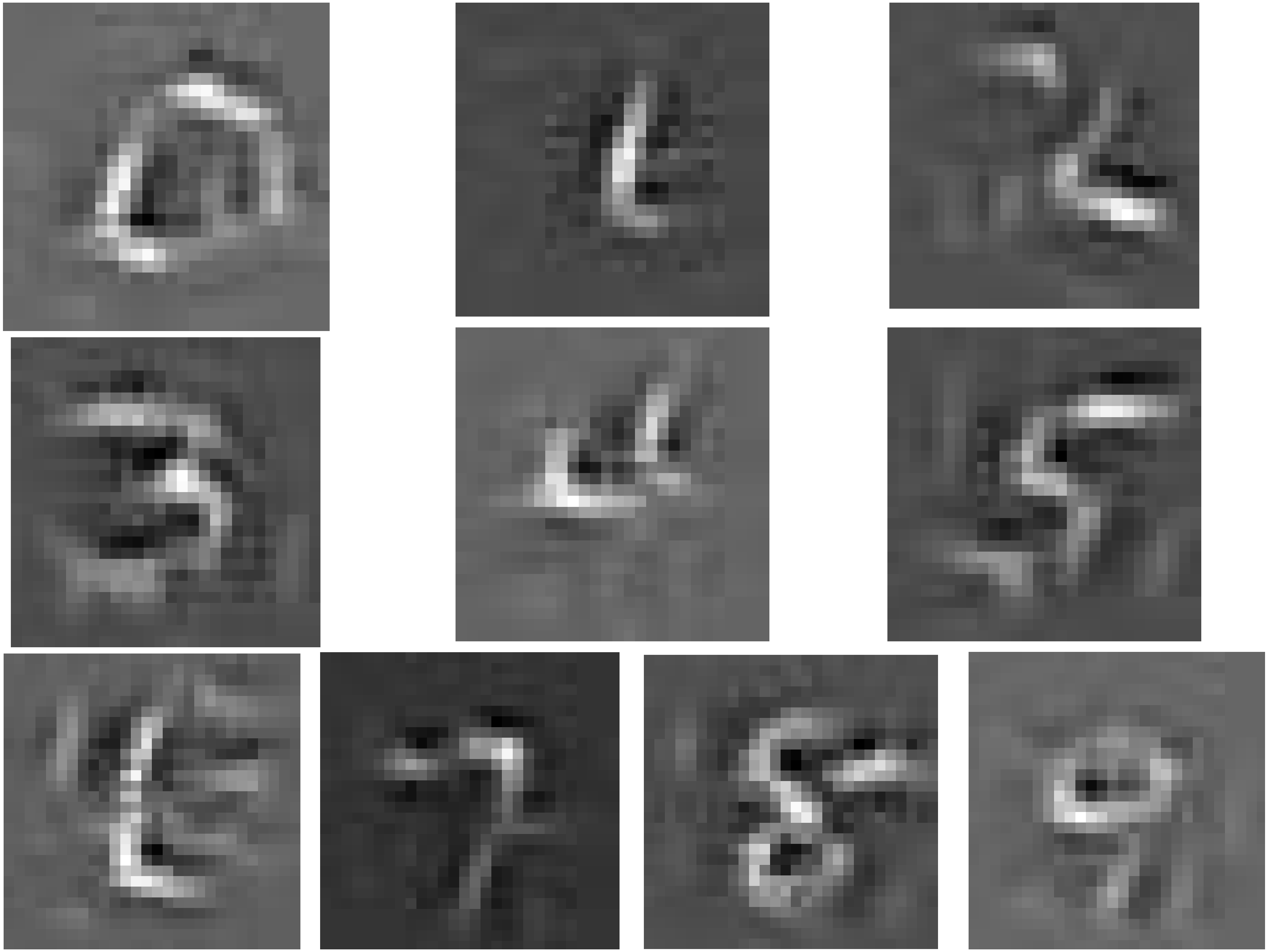




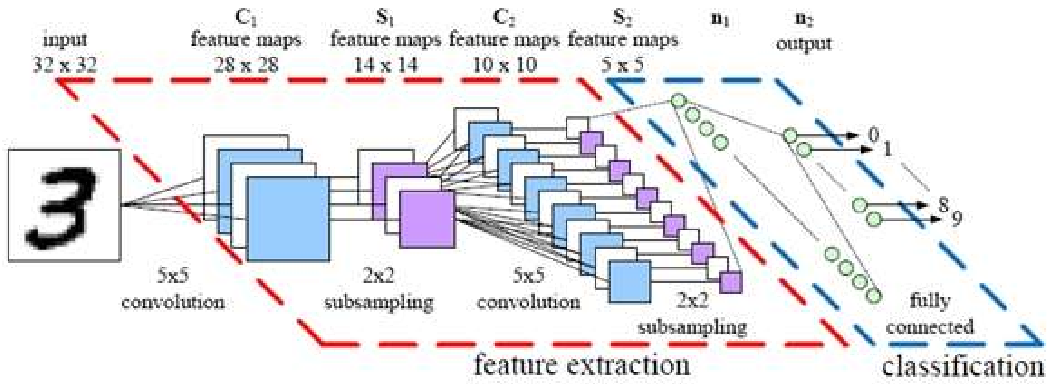








# Le-net 5

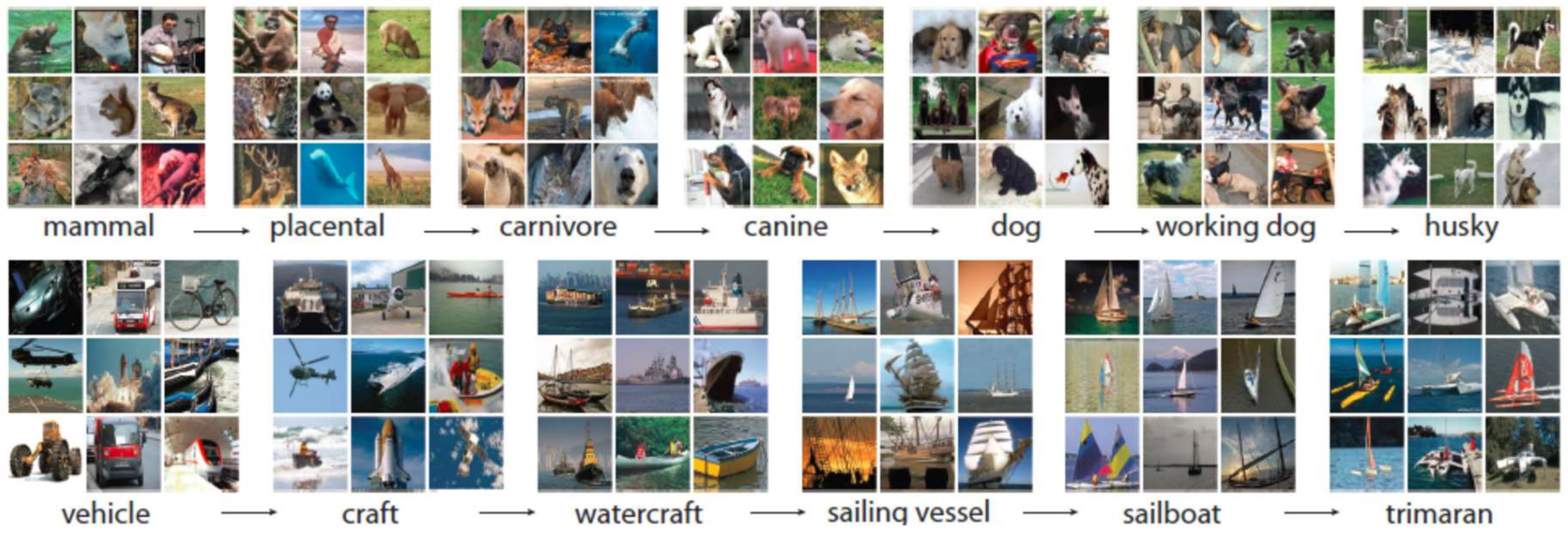


- Digit recognition on MNIST (32x32 images)
  - **Conv1:** 6 5x5 filters in first conv layer (no zero pad), stride 1
    - Result: 6 28x28 maps
  - **Pool1:** 2x2 max pooling, stride 2
    - Result:  6 14x14 maps
  - **Conv2:** 16 5x5 filters in second conv layer, stride 1, no zero pad
    - Result: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

## Nice visual example

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

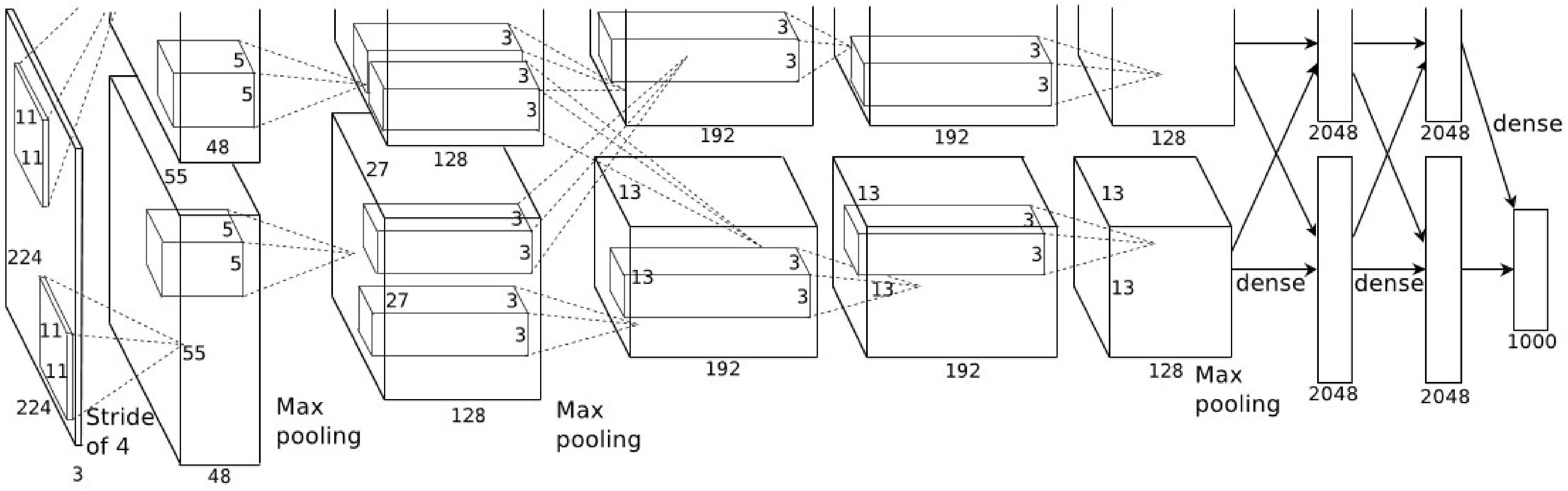
# The imagenet task


mammal → placental → carnivore → canine → dog → working dog → husky
vehicle → craft → watercraft → sailing vessel → sailboat → trimaran

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

## AlexNet

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



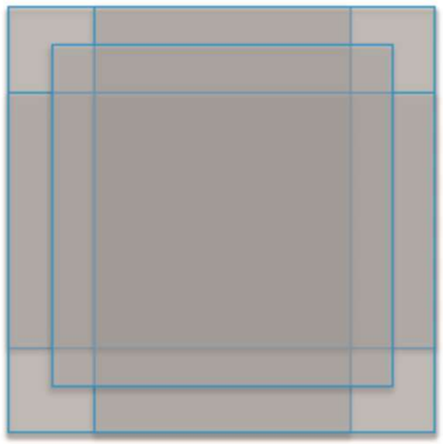
Krizhevsky, A., Sutskever, I. and Hinton, G. E. “ImageNet Classification with Deep Convolutional Neural Networks” NIPS 2012: Neural Information Processing Systems, Lake Tahoe, Nevada

# Krizhevsky et. al.

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

## Alexnet: Total parameters

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


10 patches

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

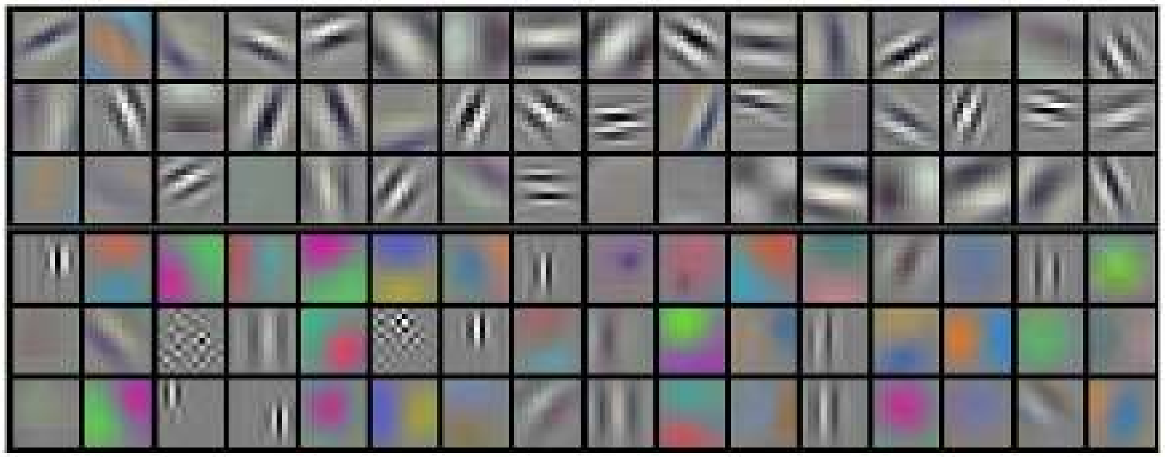
# Learning magic in Alexnet

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

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

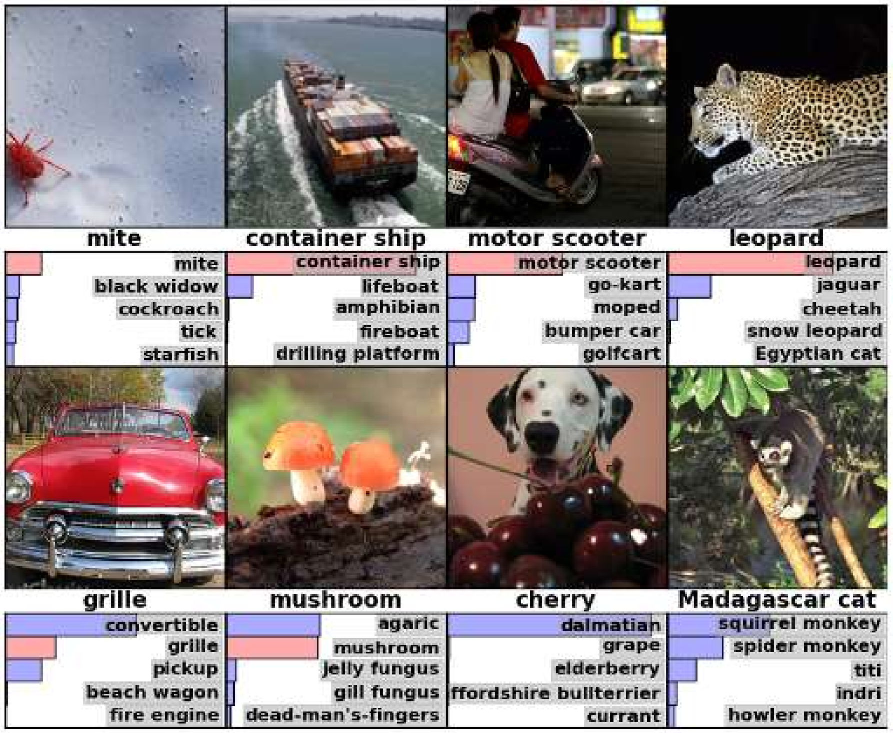
# ImageNet

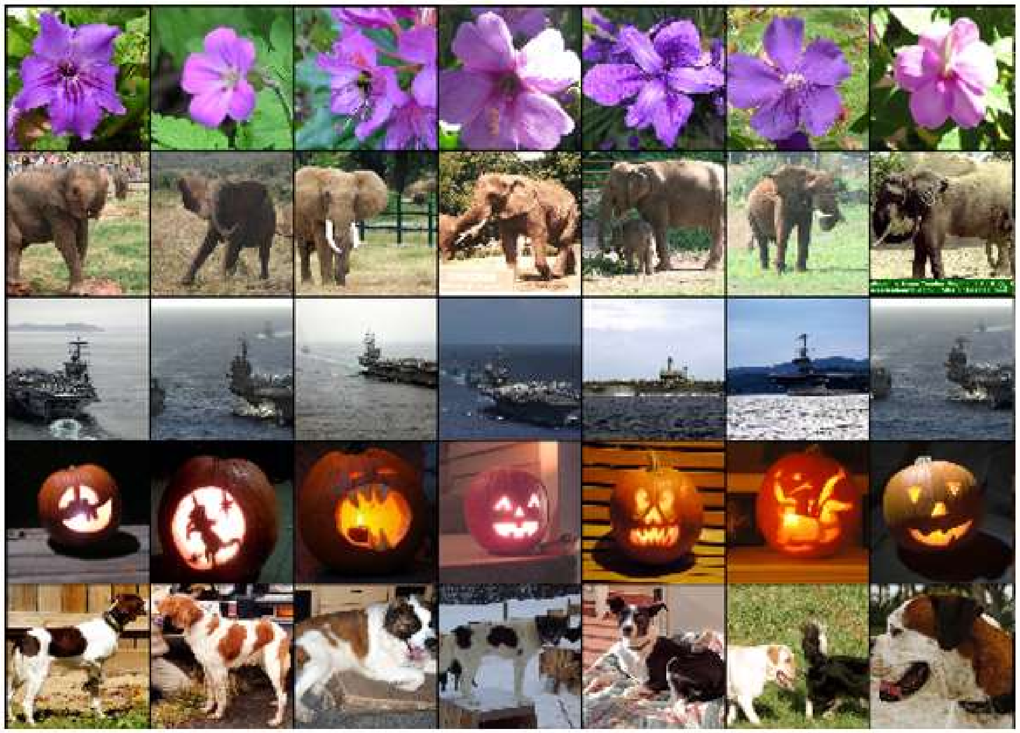
Figure 3: 96 convolutional kernels of size 11×11×3 learned by the first convolutional layer on the 224×224×3 input images. The top 48 kernels were learned on GPU 1 while the bottom 48 kernels were learned on GPU 2. See Section 6.1 for details.



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

# The net actually *learns* features!



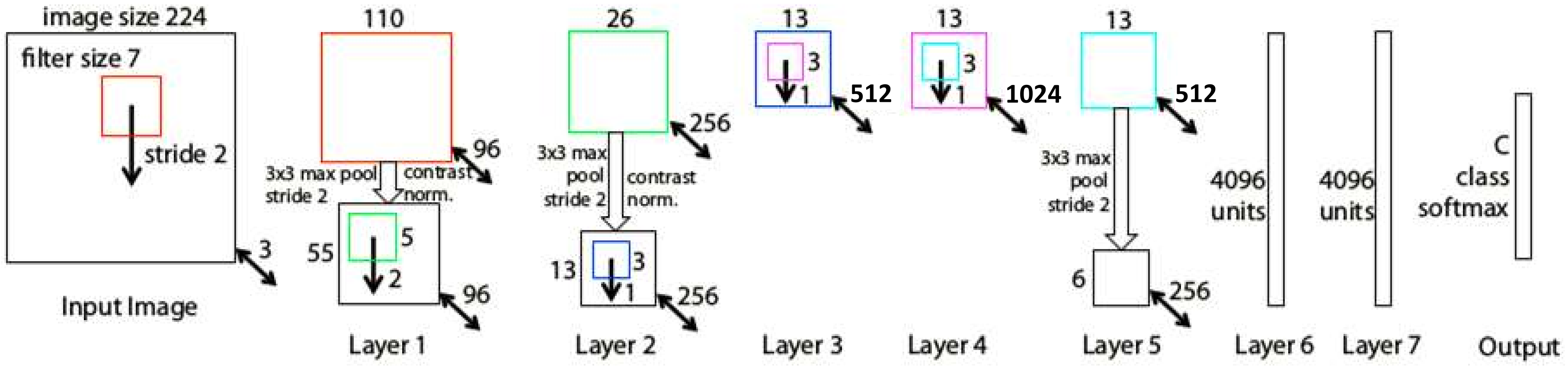


Eight ILSVRC-2010 test images and the five labels considered most probable by our model. The correct label is written under each image, and the probability assigned to the correct label is also shown with a red bar (if it happens to be in the top 5).

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

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

# ZFNet



ZF Net Architecture

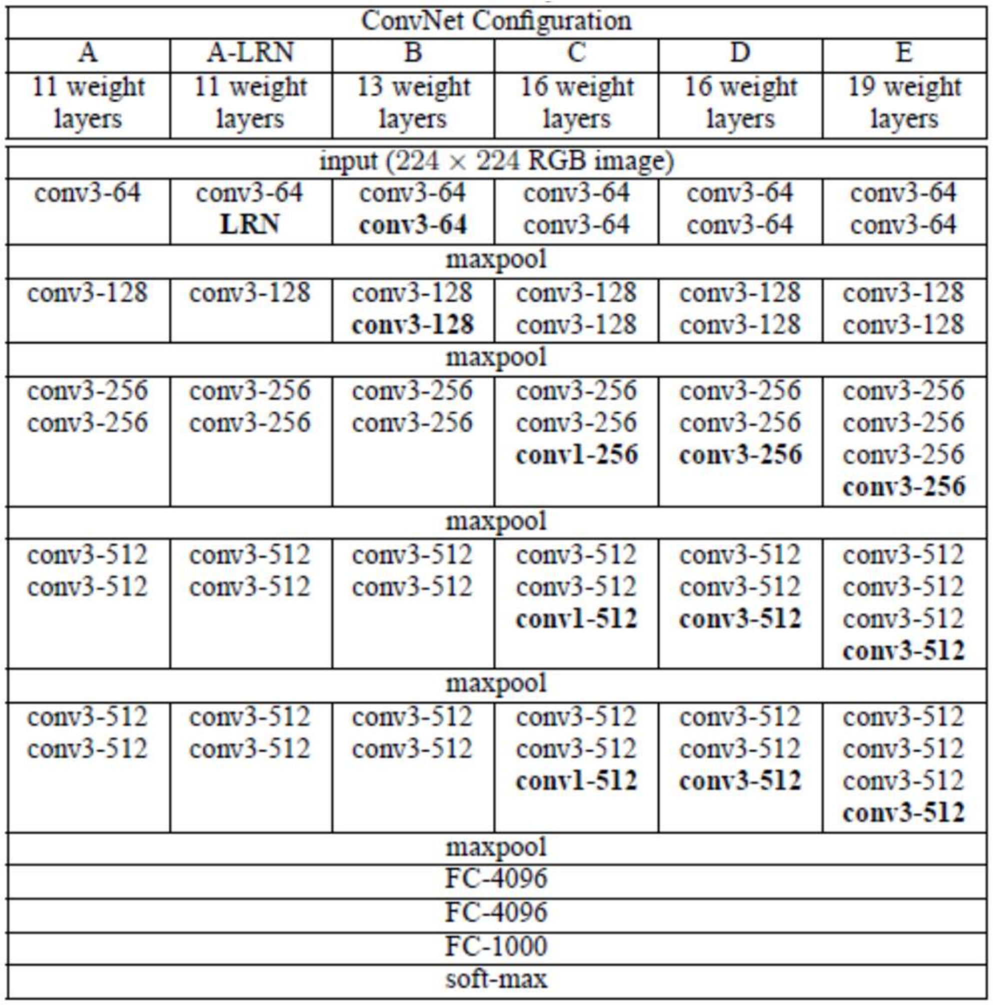
- Zeiler and Fergus 2013
- Same as Alexnet except:
  - 7x7 input-layer filters with stride 2
  - 3 conv layers are 512, 1024, 512
  - Error went down from 15.4% → 14.8%
    - Combining multiple models as before

# VGGNet

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

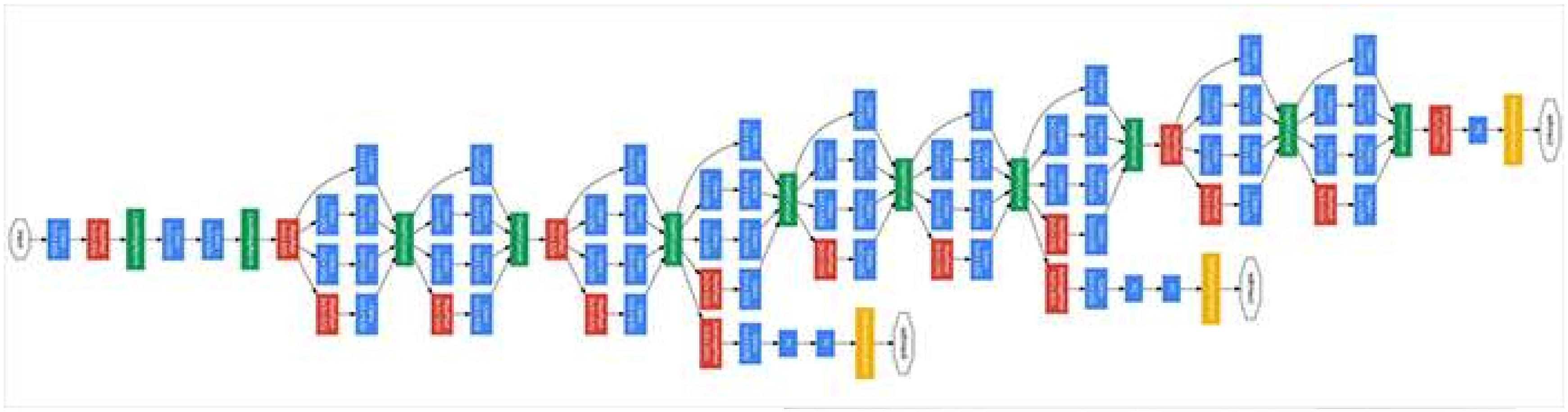
- Tried a large number of architectures.
- Finally obtained 7.3% top-5 error using 13 conv layers and 3 FC layers
  - Combining 7 classifiers
  - Subsequent to paper, reduced error to 6.8% using only two classifiers
- Final arch: 64 conv, 64 conv,
  64 pool,
  128 conv, 128 conv,
  128 pool,
  256 conv,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

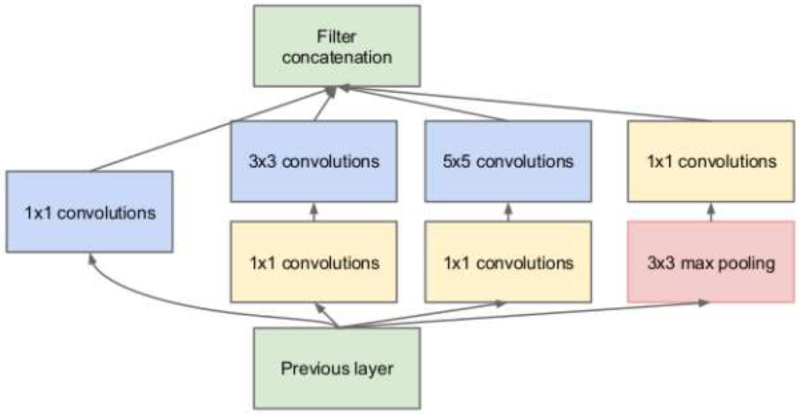
Madness!

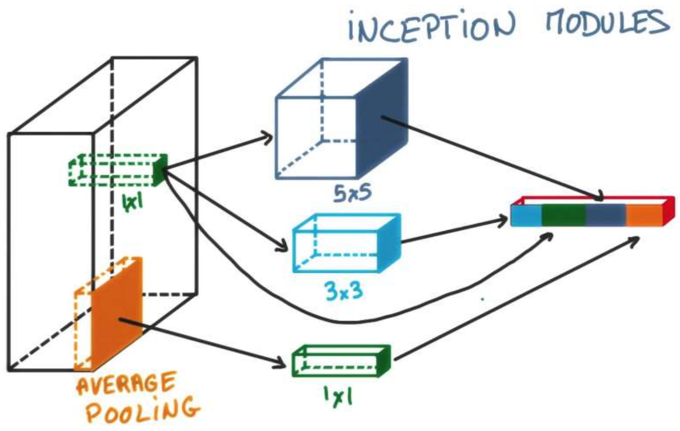


| ConvNet Configuration | | | | | |
|---|---|---|---|---|---|
| A | A-LRN | B | C | D | E |
| 11 weight layers | 11 weight layers | 13 weight layers | 16 weight layers | 16 weight layers | 19 weight layers |
| input (224 × 224 RGB image) | | | | | |
| conv3-64 | conv3-64<br>**LRN** | conv3-64<br>**conv3-64**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

# Googlenet: Inception

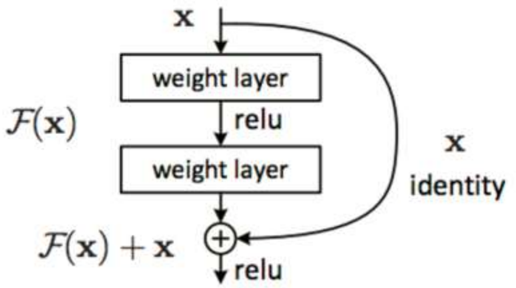






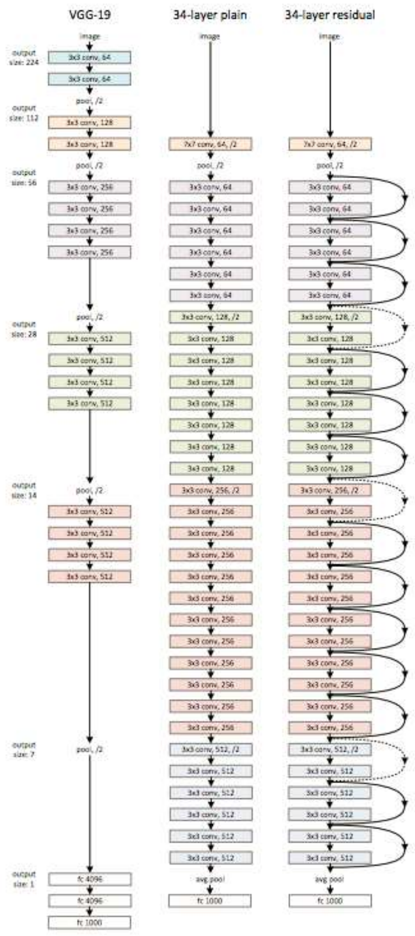
- Multiple filter sizes simultaneously
- Details irrelevant;  error → 6.7%
  – Using only 5 million parameters, thanks to average pooling

# Imagenet



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

Figure 2. Residual learning: a building block.



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

# Resnet details for the curious..

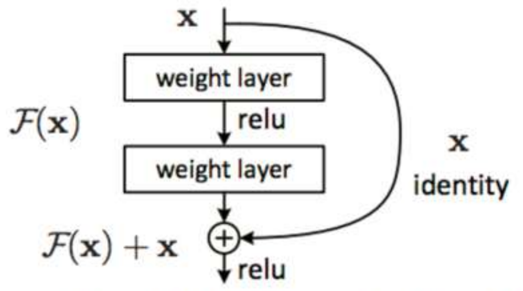
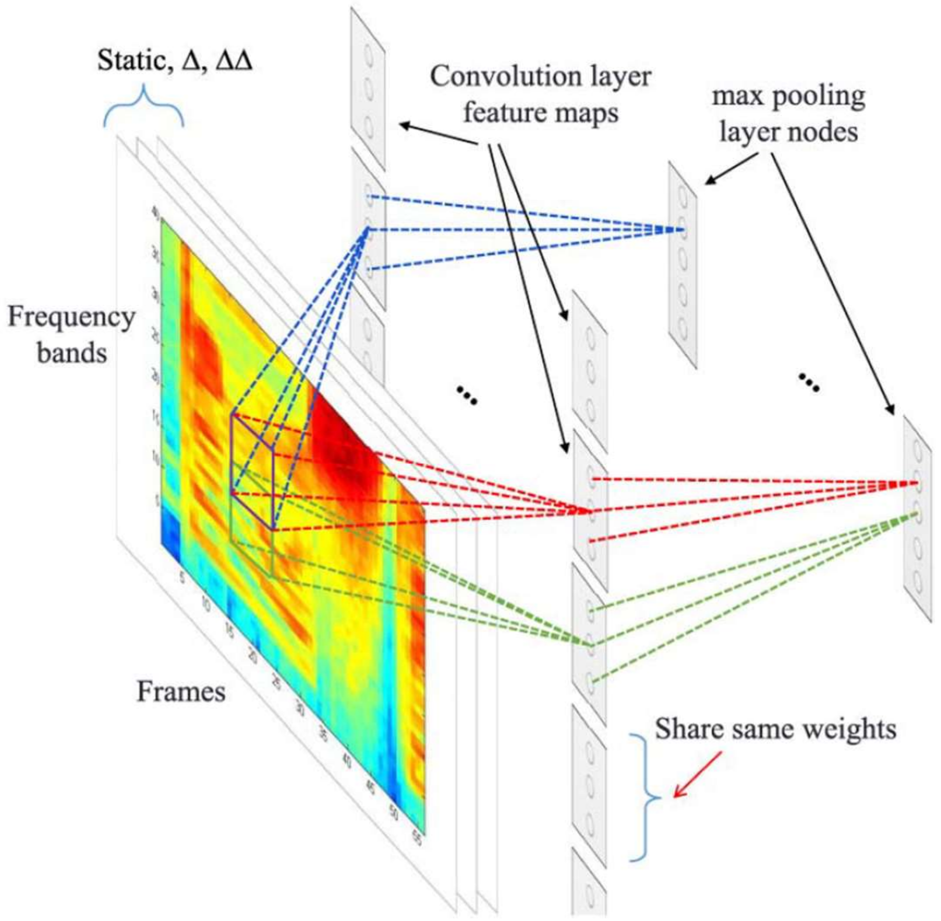

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

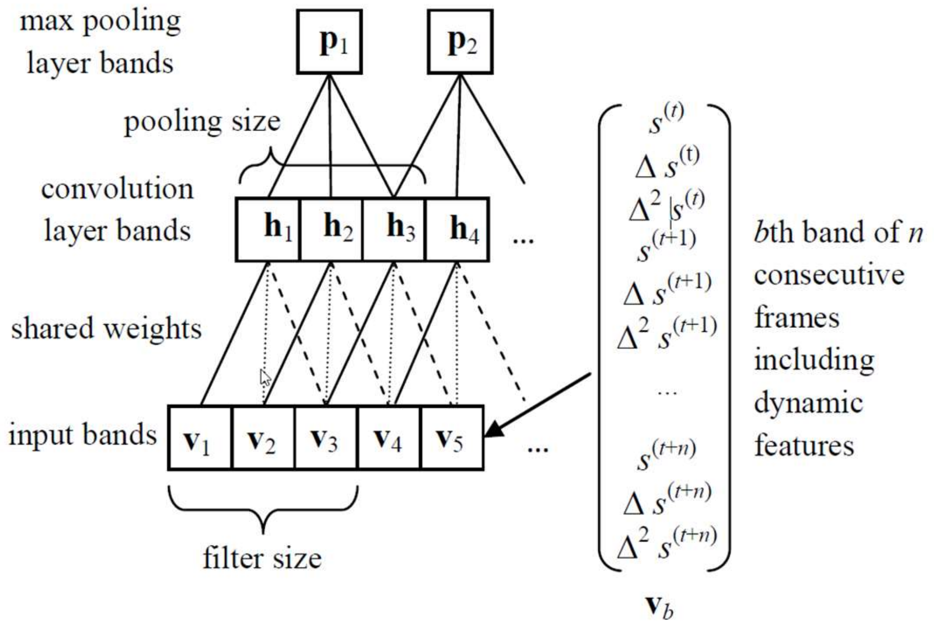
Figure 2. Residual learning: a building block.

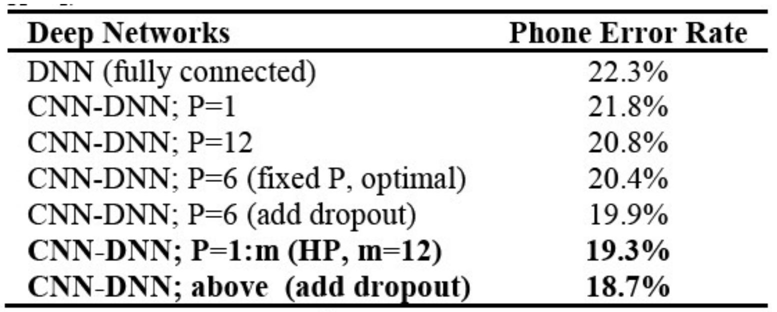
- Last layer before addition must have the same number of filters as the input to the module
- Batch normalization after each convolution
- SGD + momentum (0.9)
- Learning rate 0.1, divide by 10 (batch norm lets you use larger learning rate)
- Mini batch 256
- Weight decay 1e-5
- ***No pooling in Resnet***

# CNN for Automatic Speech Recognition

- Convolution over frequencies
- Convolution over time





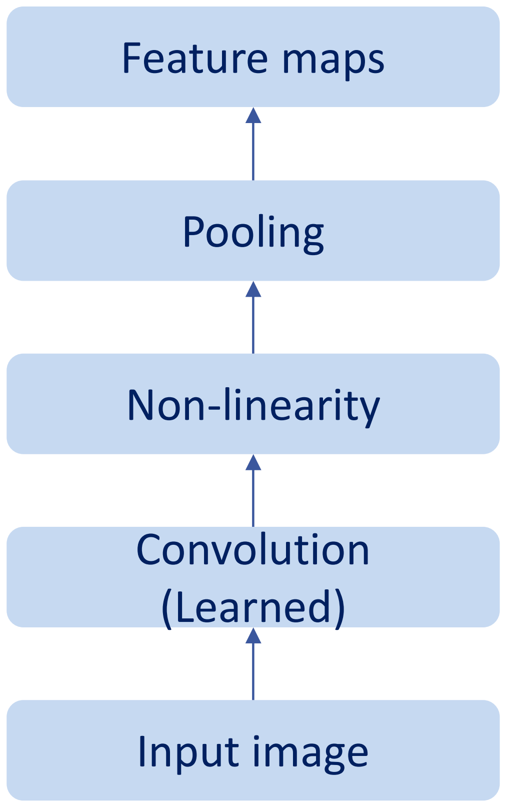


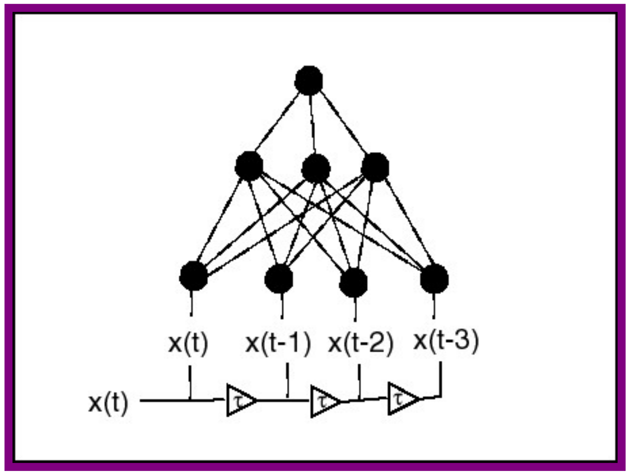
| Deep Networks | Phone Error Rate |
|---|---|
| DNN (fully connected) | 22.3% |
| CNN-DNN; P=1 | 21.8% |
| CNN-DNN; P=12 | 20.8% |
| CNN-DNN; P=6 (fixed P, optimal) | 20.4% |
| CNN-DNN; P=6 (add dropout) | 19.9% |
| **CNN-DNN; P=1:m (HP, m=12)** | **19.3%** |
| **CNN-DNN; above (add dropout)** | **18.7%**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

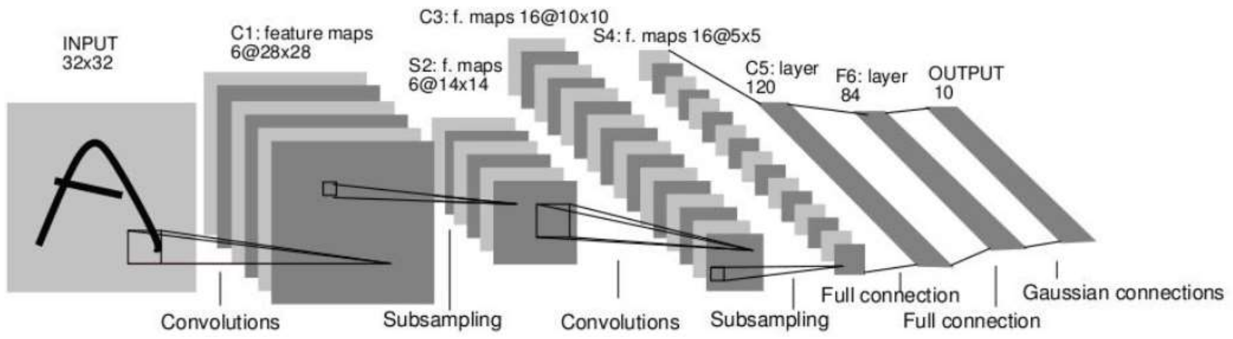
***Table 1:*** *TIMIT core test set phone recognition error rate comparisons.*

# 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 <u>back-propagating</u> error
- Convolution over time


Feature maps
Pooling
Non-linearity
Convolution (Learned)
Input image


x(t)
x(t-1)
x(t-2)
x(t-3)
x(t)
τ
τ
τ


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