# Convolutional Neural Networks

M. Soleymani
Sharif University of Technology
Spring 2023

Most slides have been adopted from Fei Fei Li and colleagues, cs231n, Stanford 2022
and some from Bhiksha Raj, 11-785, CMU

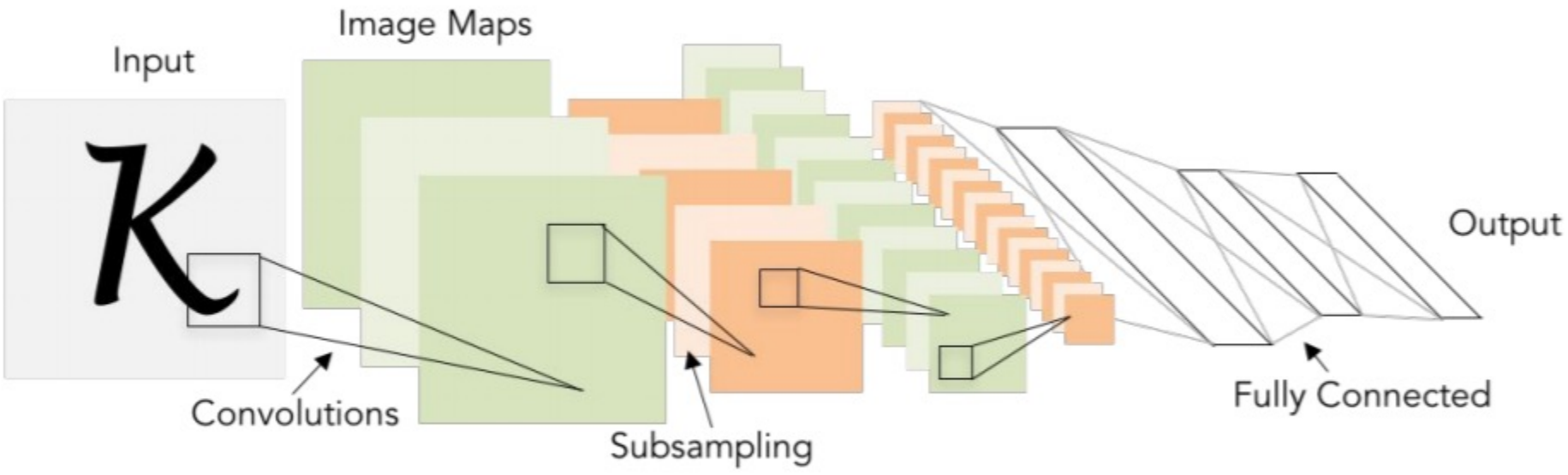
# Convolutional neural nets

- One of *the* most frequently used networks

- Used *everywhere*
  - Not just for image classification
  - Used in speech and audio processing
    - Convnets on *spectrograms*
  - And also for texts, …

# LeNet

[LeCun, Bottou, Bengio, Haffner 1998]




LeNet-5

# AlexNet

[Krizhevsky, Sutskever, Hinton, 2012]

- ImageNet Classification with Deep Convolutional Neural Networks

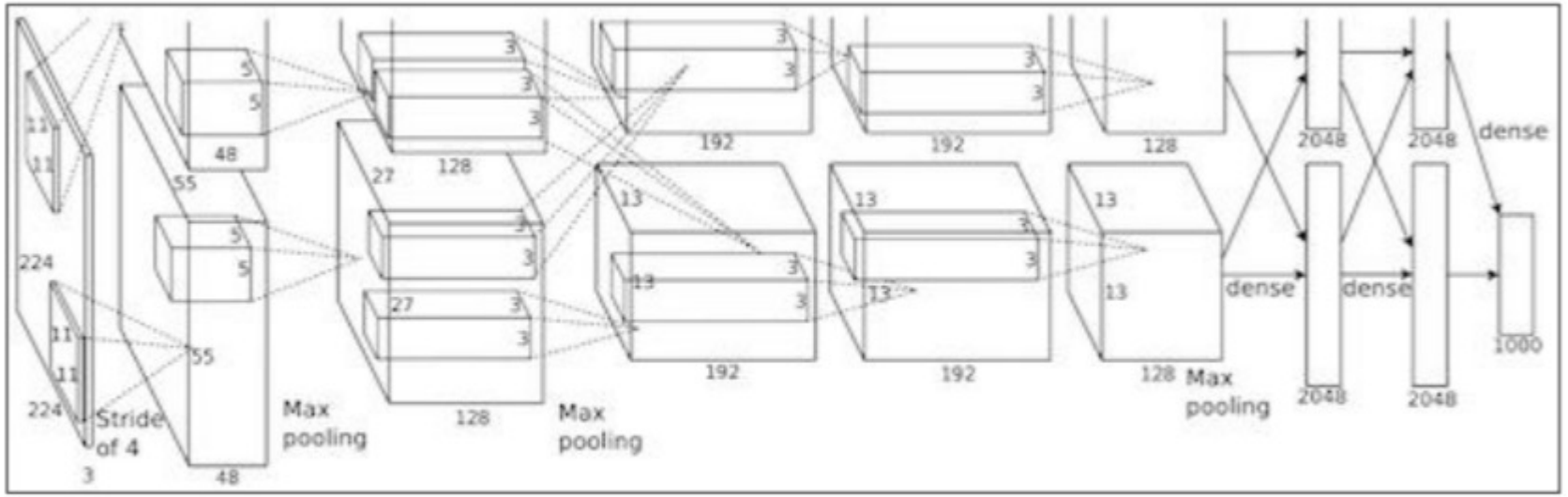
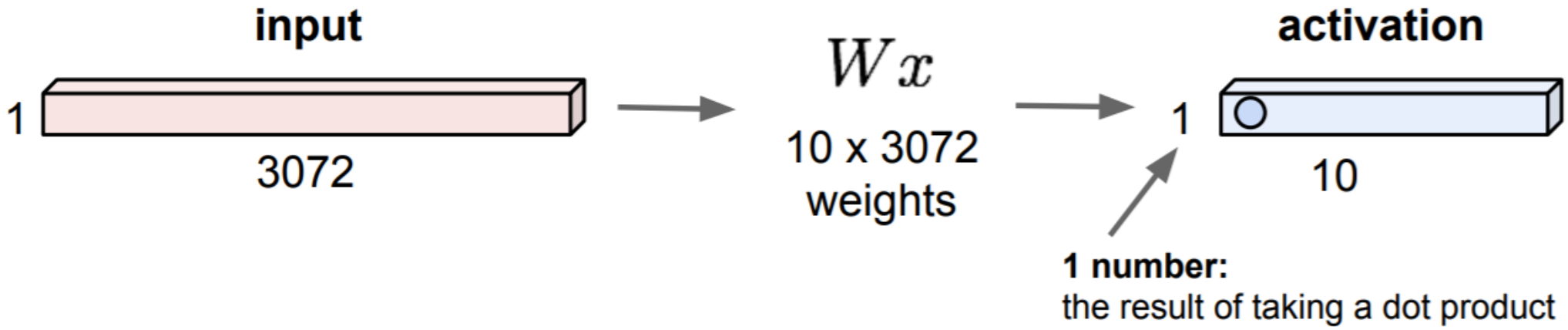



Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# Fully connected layer

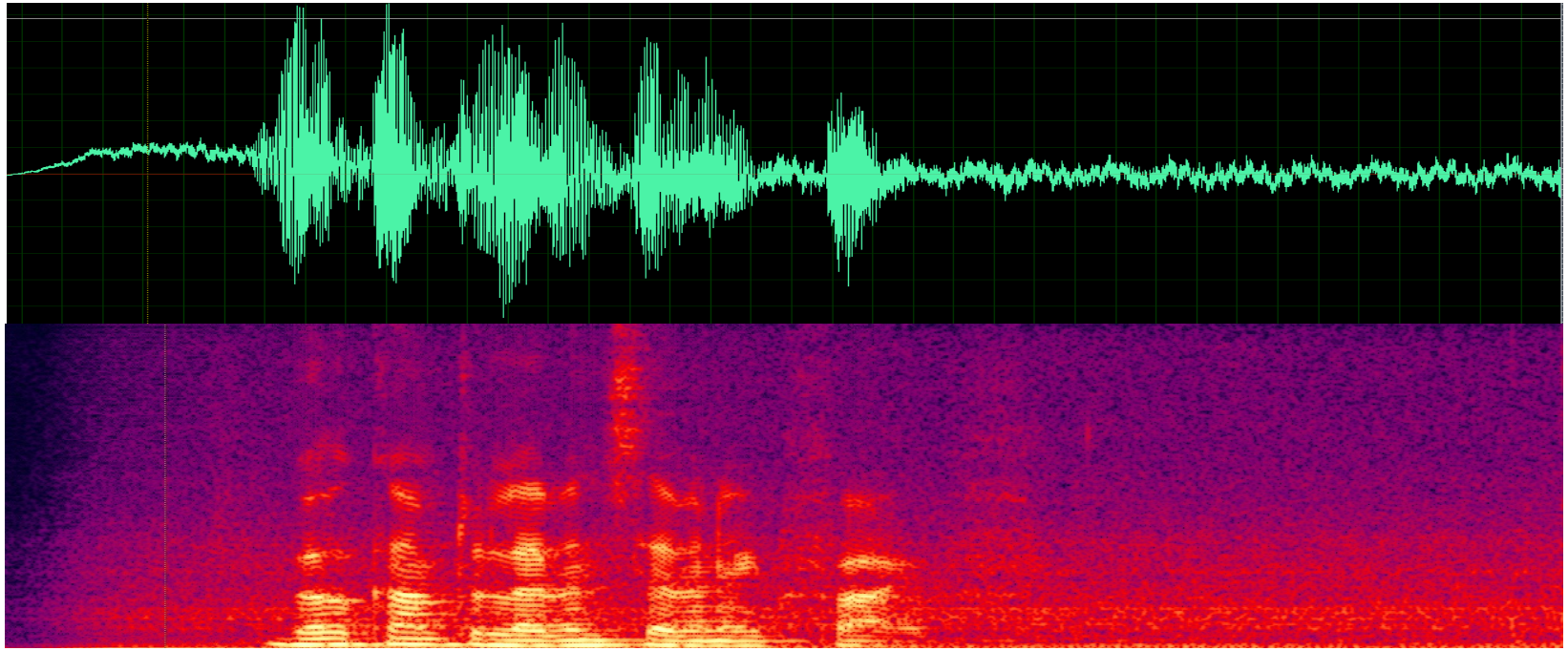
32x32x3 image -> stretch to 3072 x 1



# Fully connected layers

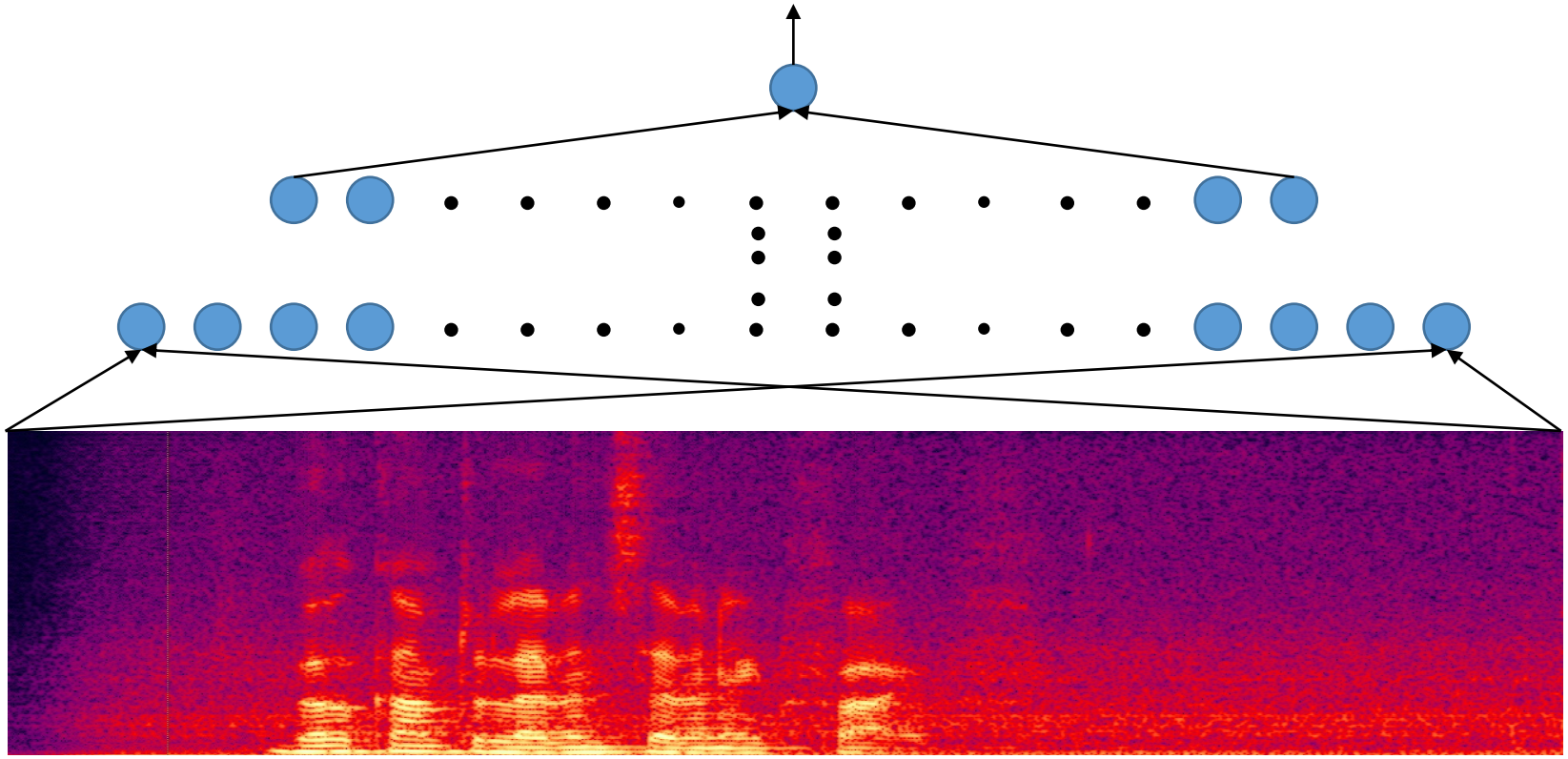
- Neurons in a single layer function completely independently and do not share any connections even for inputs
- To be shift-invariant many samples (in various time or locations) must show to them
- Regular Neural Nets don’t scale well to full images
  - parameters would add up quickly!
  - full connectivity is wasteful and the huge number of parameters would quickly lead to overfitting.

# A problem



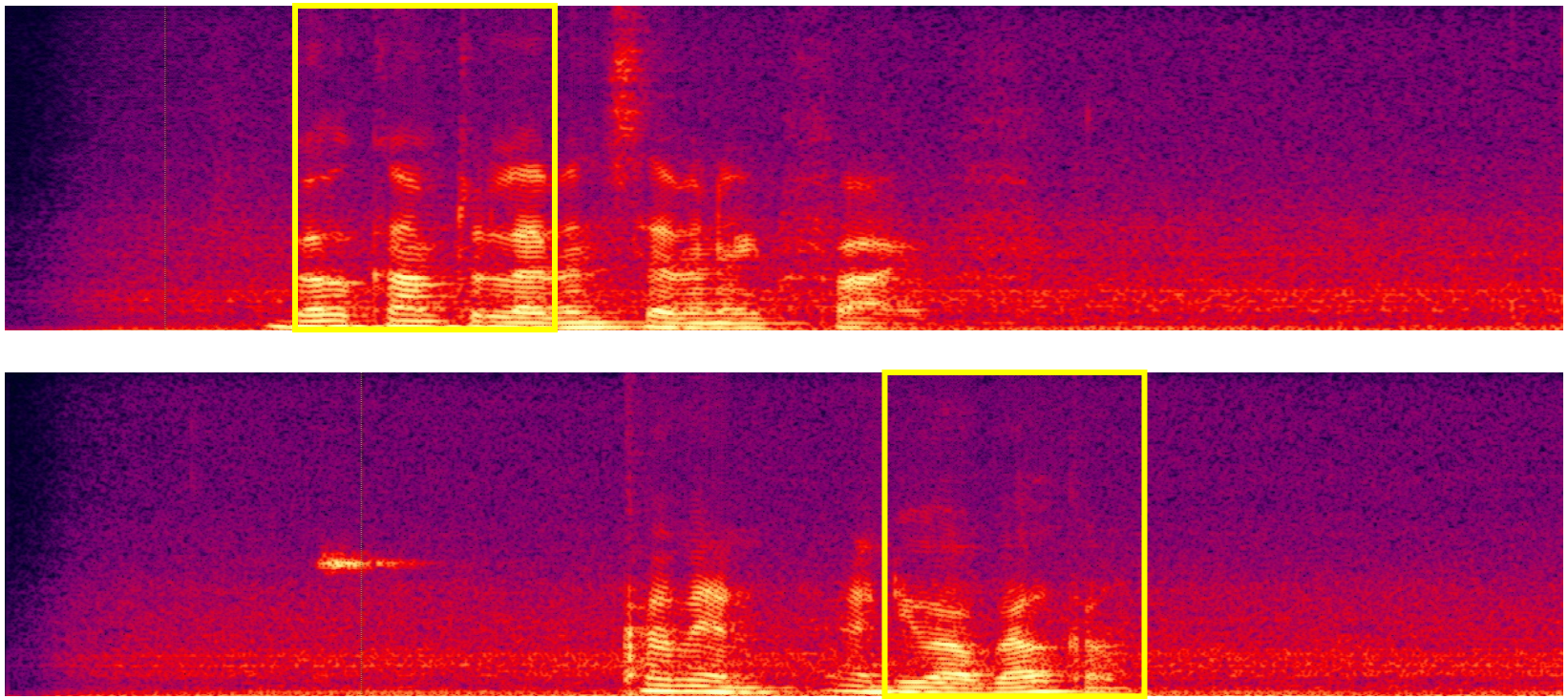
- Does this signal contain the word “Welcome”?
- Compose an MLP for this problem.
  - Assuming all recordings are exactly the same length..

# Finding a Welcome



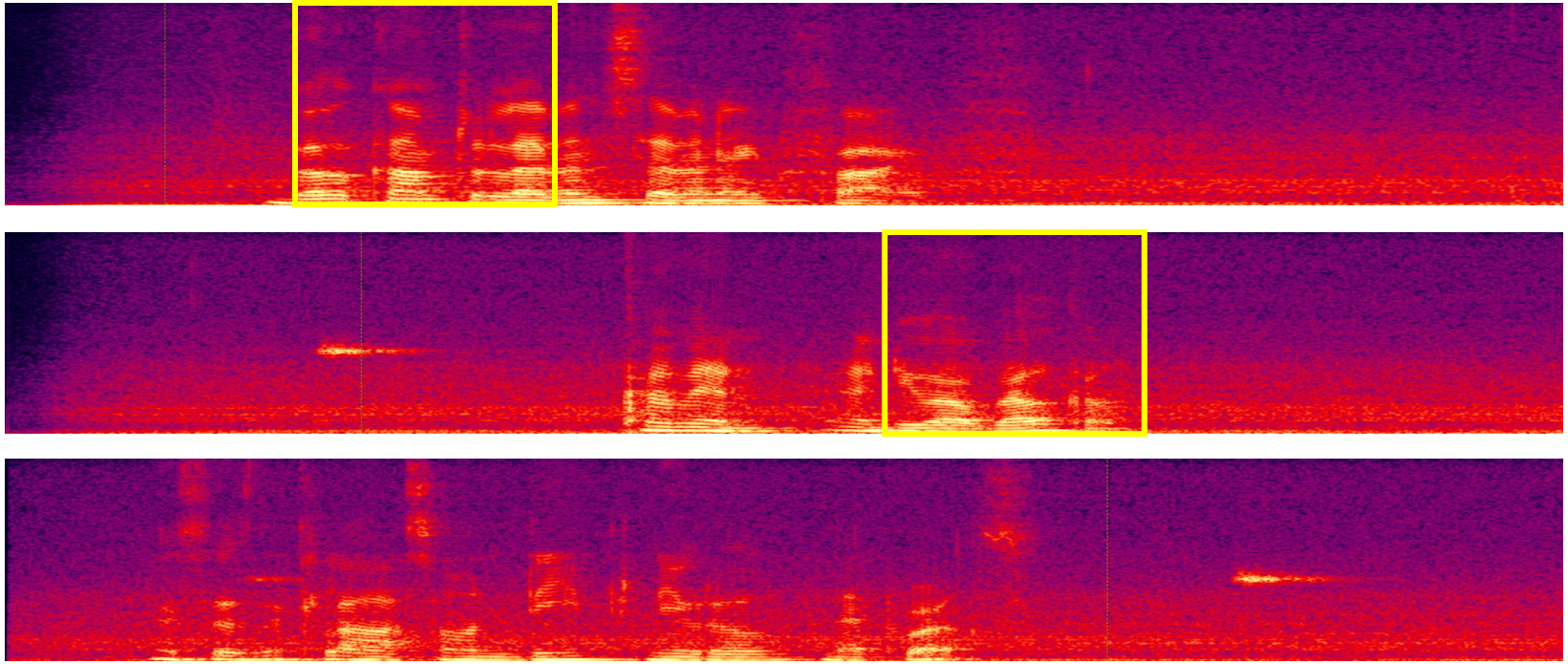
- Trivial solution:  Train an MLP for the entire recording

# Finding a Welcome



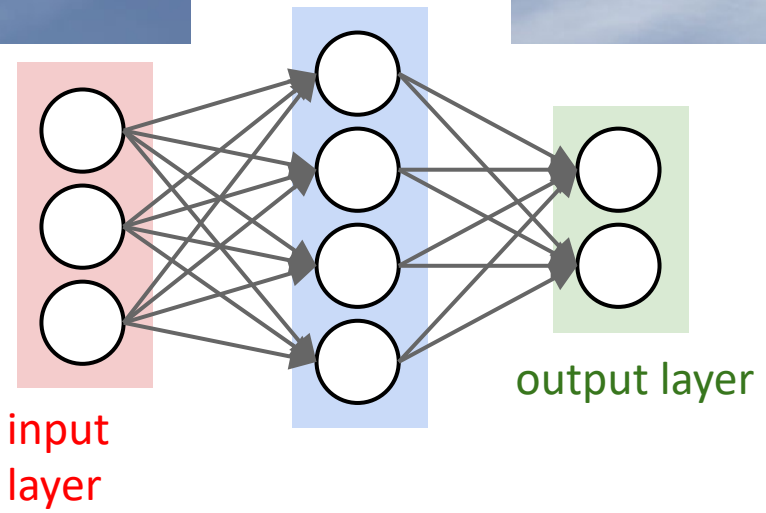
- Problem with trivial solution: Network that finds a “welcome” in the top recording will not find it in the lower one
  - Unless trained with both
  - Will require a very large network and a large amount of training data to cover every case

# Finding a Welcome



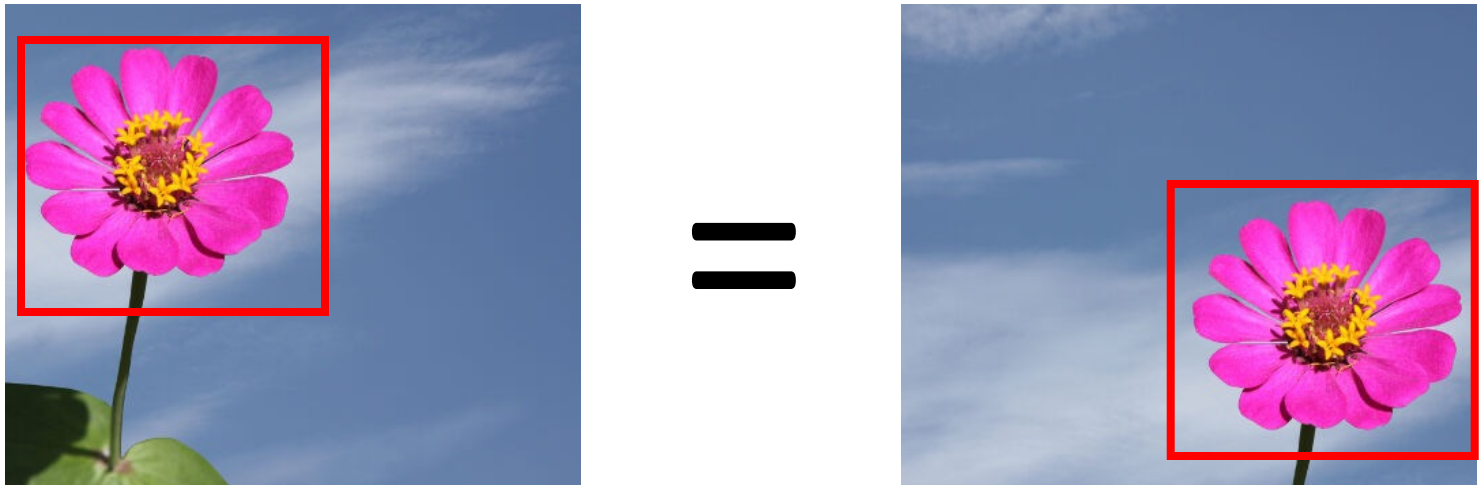
- Need a *simple* network that will fire regardless of the location of "Welcome"
  - and not fire when there is none

# A problem




- Will an MLP that recognizes the left image as a flower also recognize the one on the right as a flower?

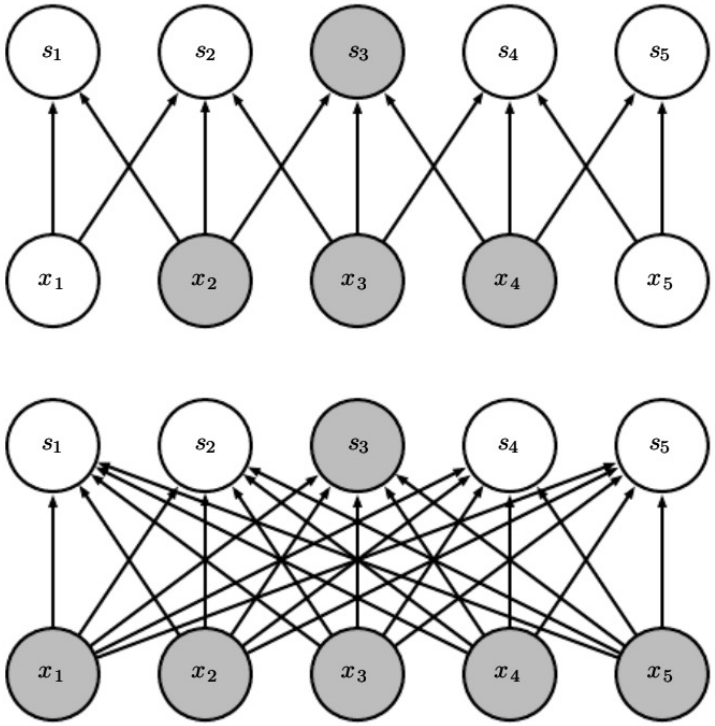
# The need for *shift invariance*



- In many problems the *location* of a pattern is not important
  - Only the presence of the pattern
- Conventional MLPs are sensitive to the location of the pattern
  - Moving it by one component results in an entirely different input that the MLP wont recognize
- Requirement:  Network must be *shift invariant*

# Convolutional layer VS fully connected layer

- **locality**




convolution

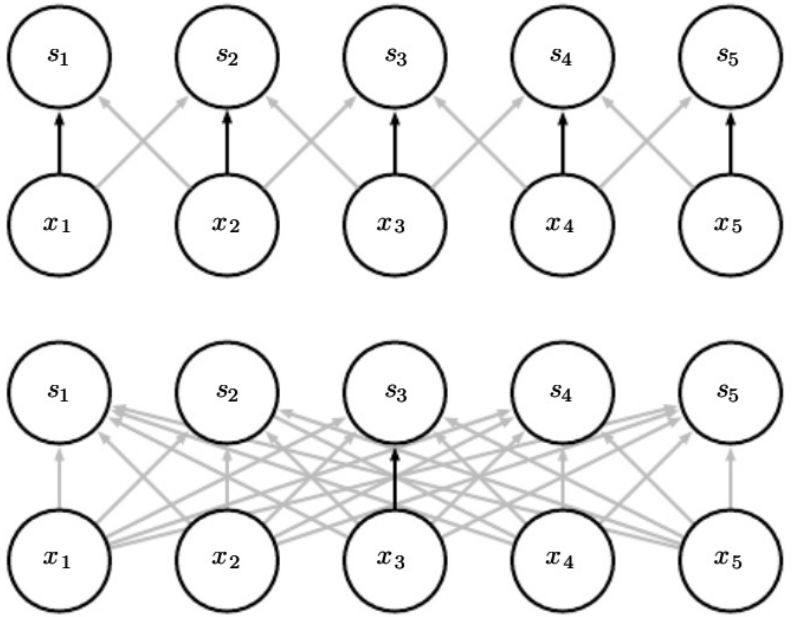
fully connected

[Goodfellow et al. 2016]

# Convolutional layer VS fully connected layer

- **Parameter sharing**

(Black arrows indicate the connections that use a particular parameter in two different models)




[Goodfellow et al. 2016]

# Example

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

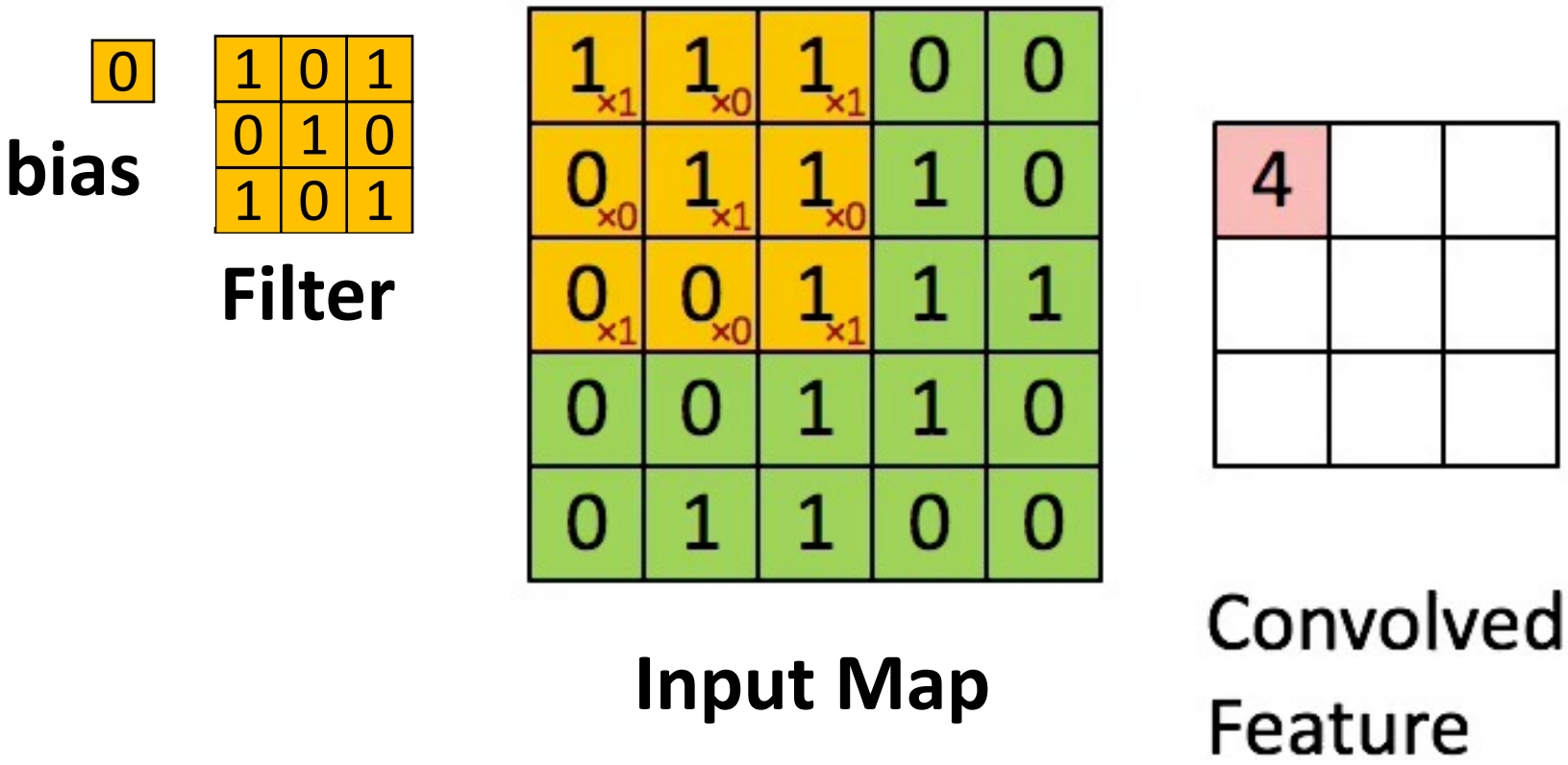
| | | |
|---|---|---|
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 1 |

bias

| |
|---|
| 0 |

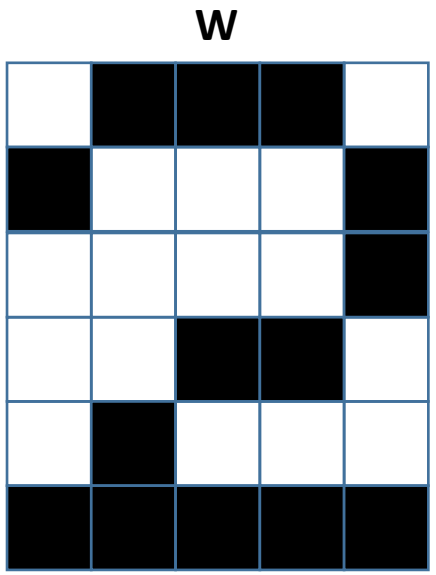
- Scanning an image with a “filter”
  - Note: a filter is really just a perceptron, with weights and a bias

# What is a convolution

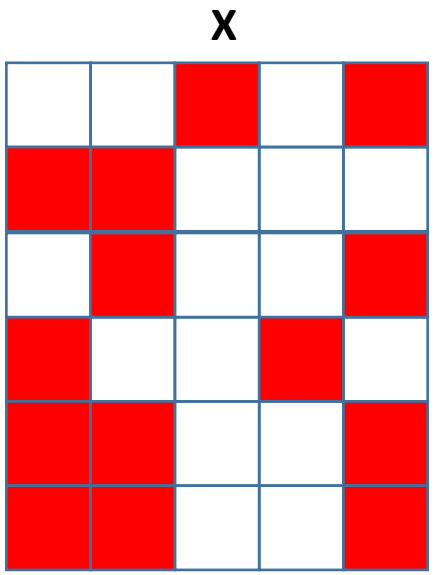



- Scanning an image with a “filter”
  - At each location, the “filter and the underlying map values are multiplied component wise, and the products are added along with the bias

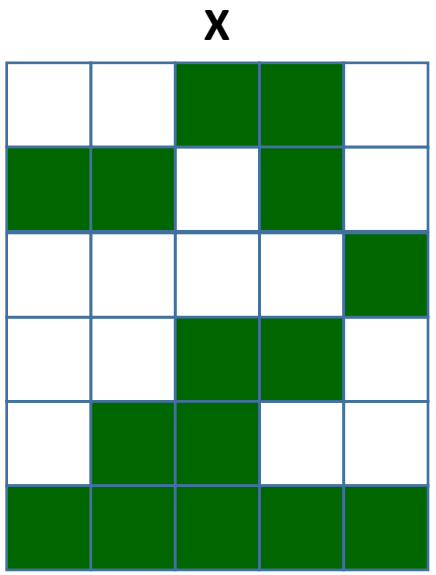
# The weights as a correlation filter




$$o = \begin{cases} 1 \; if \sum_i w_i x_i \geq T \\ 0 \; else \end{cases}$$



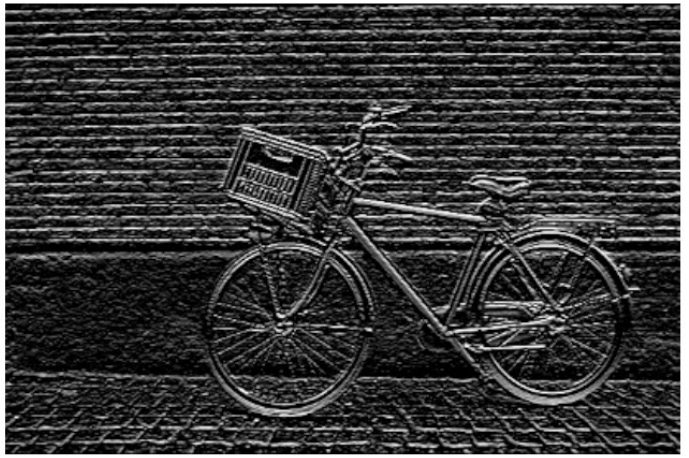

Correlation = 0.57




Correlation = 0.82

- The green pattern looks more like the weights pattern (black) than the red pattern
  - The green pattern is more *correlated* with the weights

# Convolution: Example

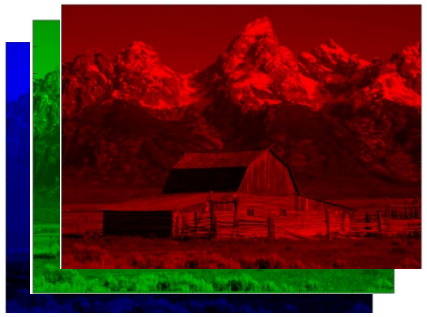


| | | |
|---|---|---|
| −1 | 0 | 1 |
| −1 | 0 | 1 |
| −1 | 0 | 1 |

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

Bishop Deep Learning, 2023

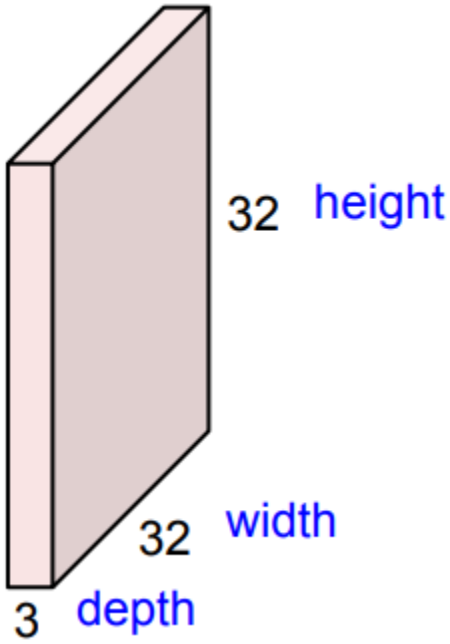
# Input for colored images



- Input: 3 pictures

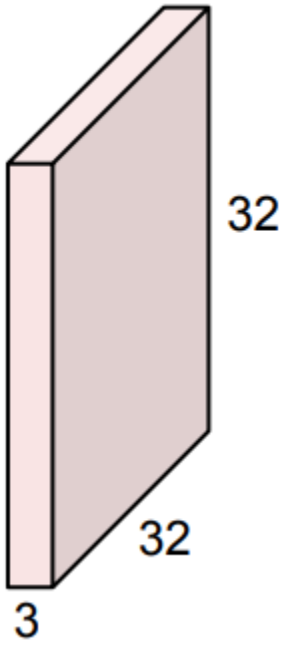
# Convolution Layer

32x32x3 image -> preserve spatial structure

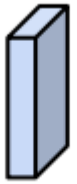


# Convolution Layer

32x32x3 image




5x5x3 filter

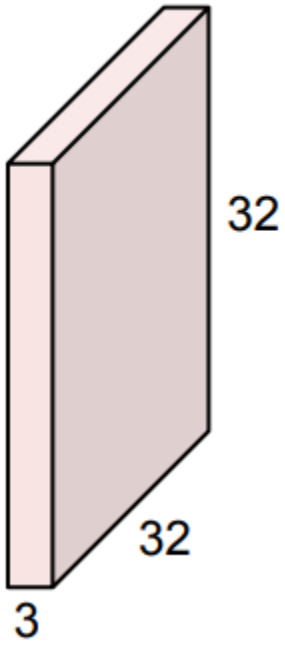


**Convolve** the filter with the image i.e. “slide over the image spatially, computing dot products”

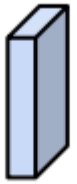
# Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

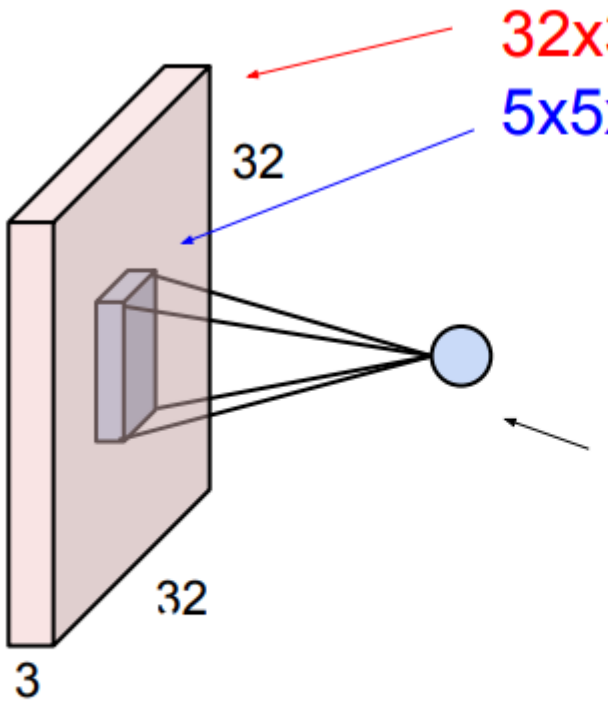



5x5x3 filter



**Convolve** the filter with the image i.e. “slide over the image spatially, computing dot products”

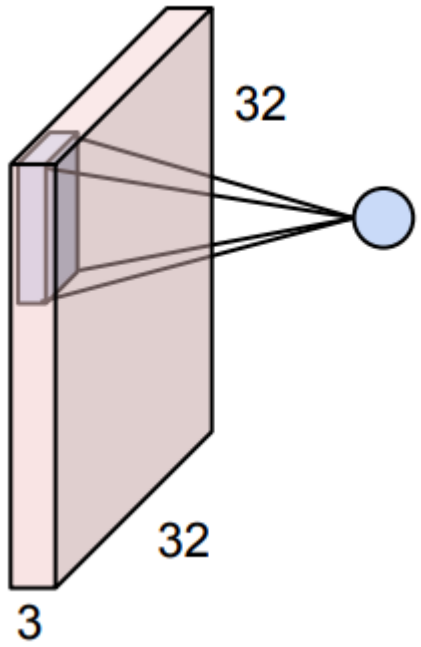
# Convolution Layer



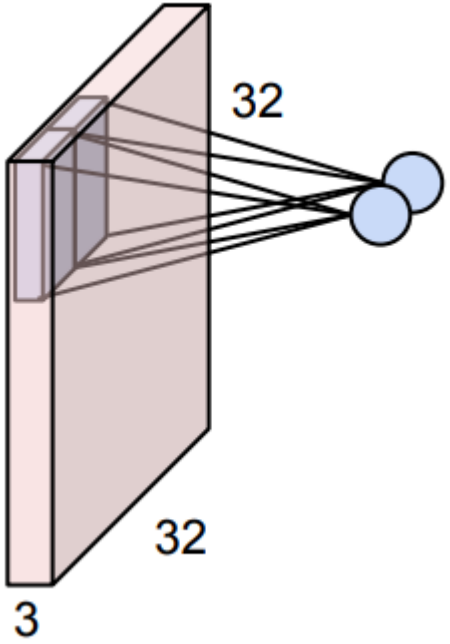

**1 number:**
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

$$w^T x + b$$

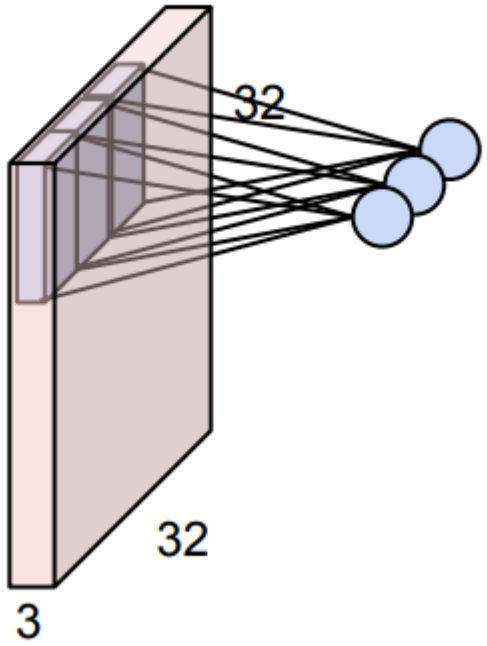
# Convolution Layer



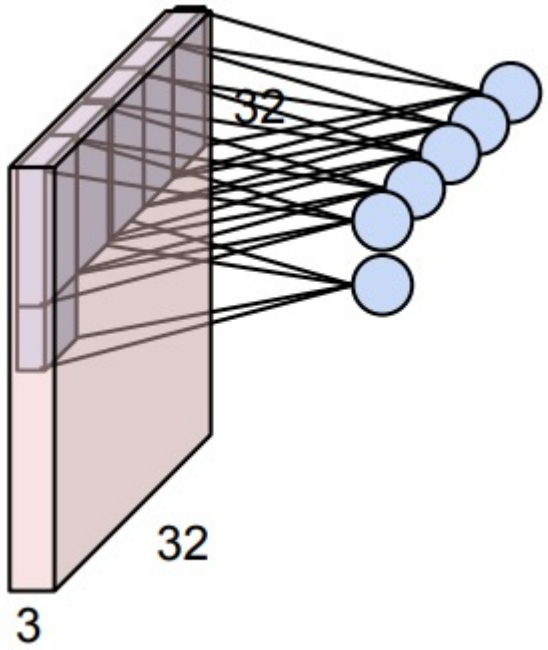
# Convolution Layer



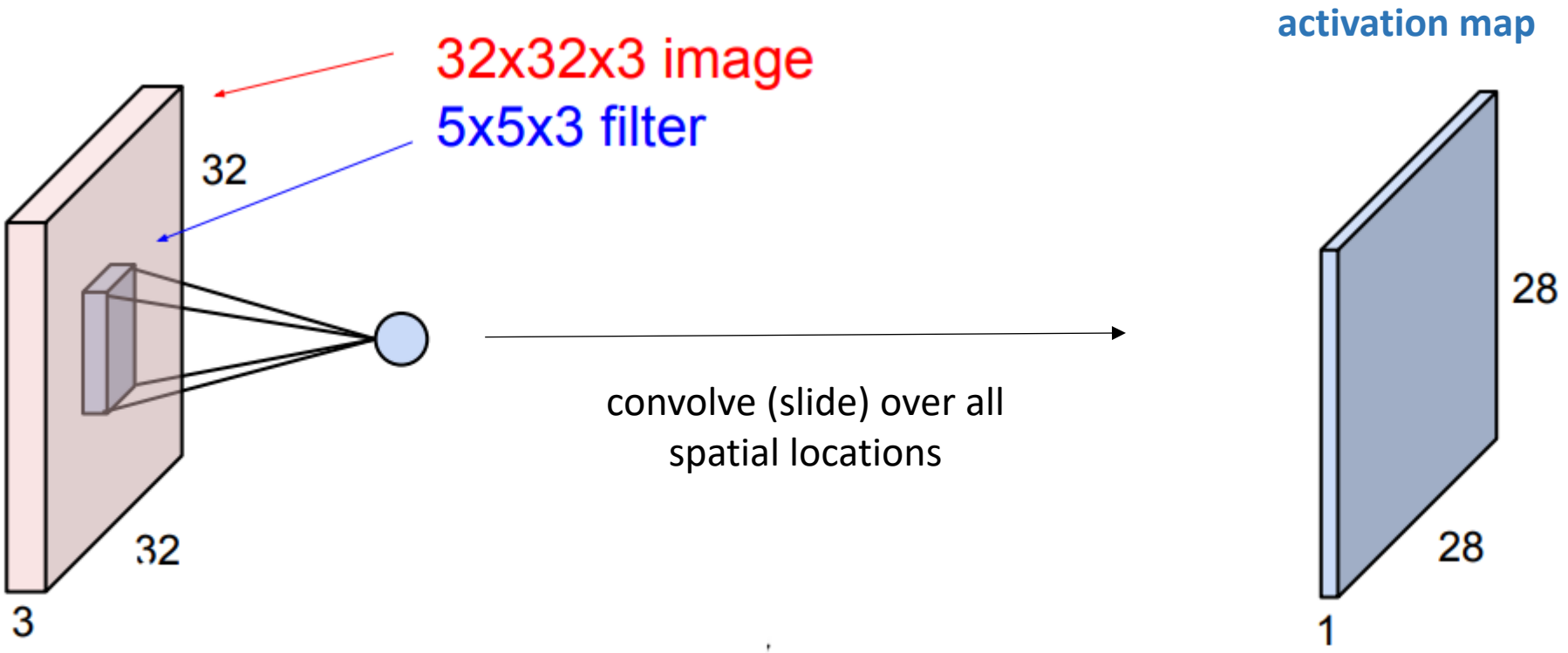
# Convolution Layer



# Convolution Layer

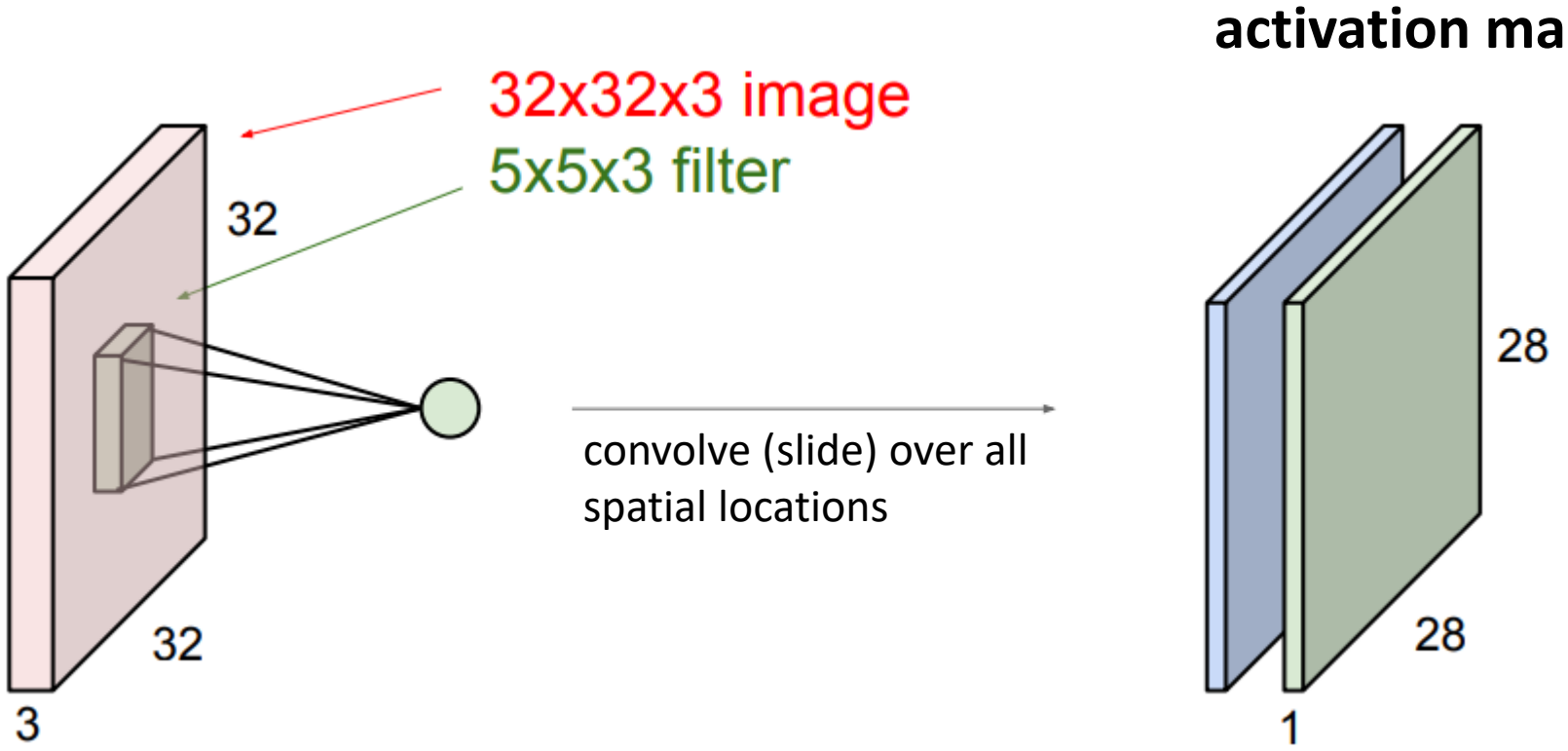


# Convolution Layer

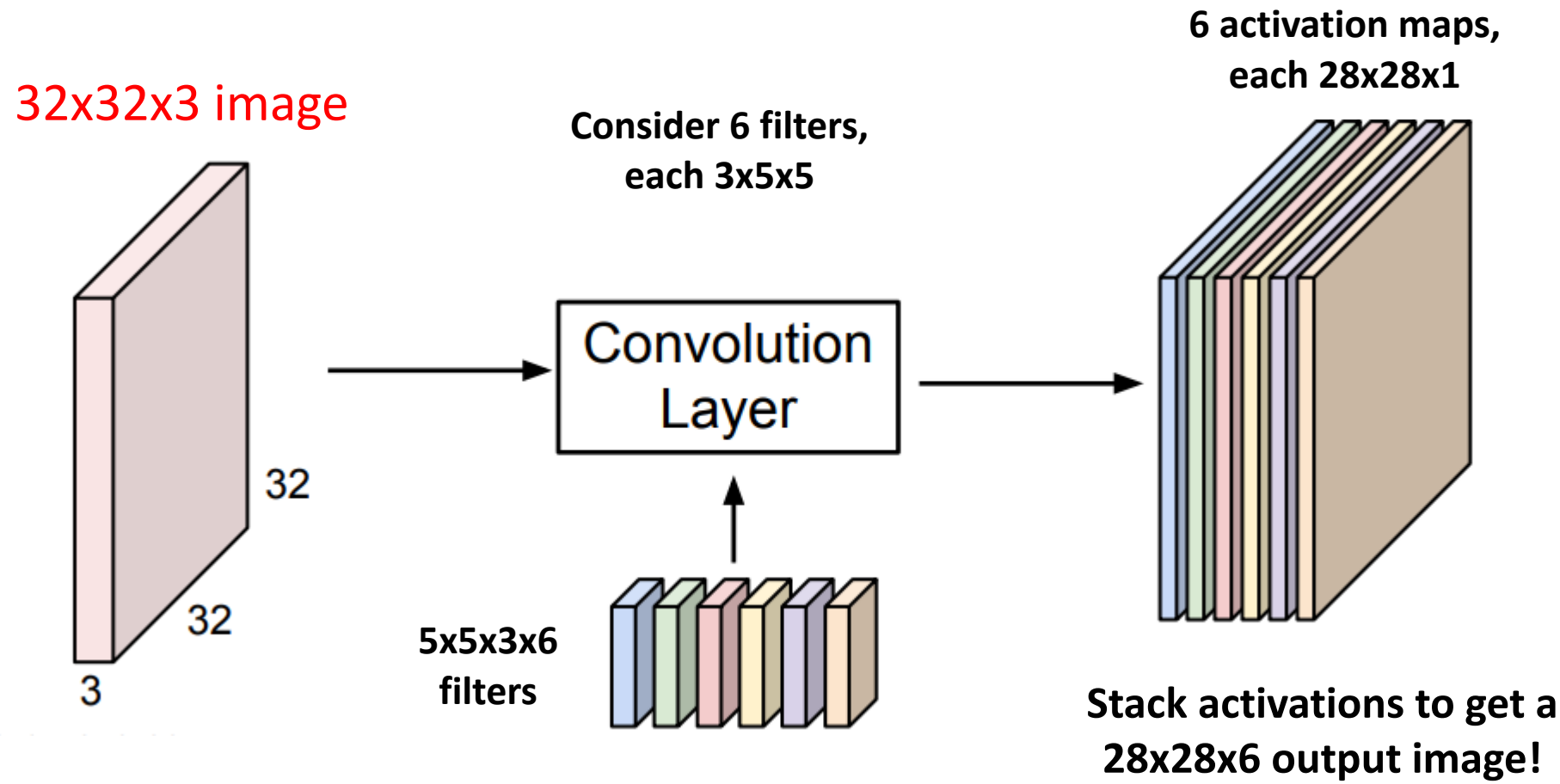


# Convolution Layer

**consider a second, green filter**

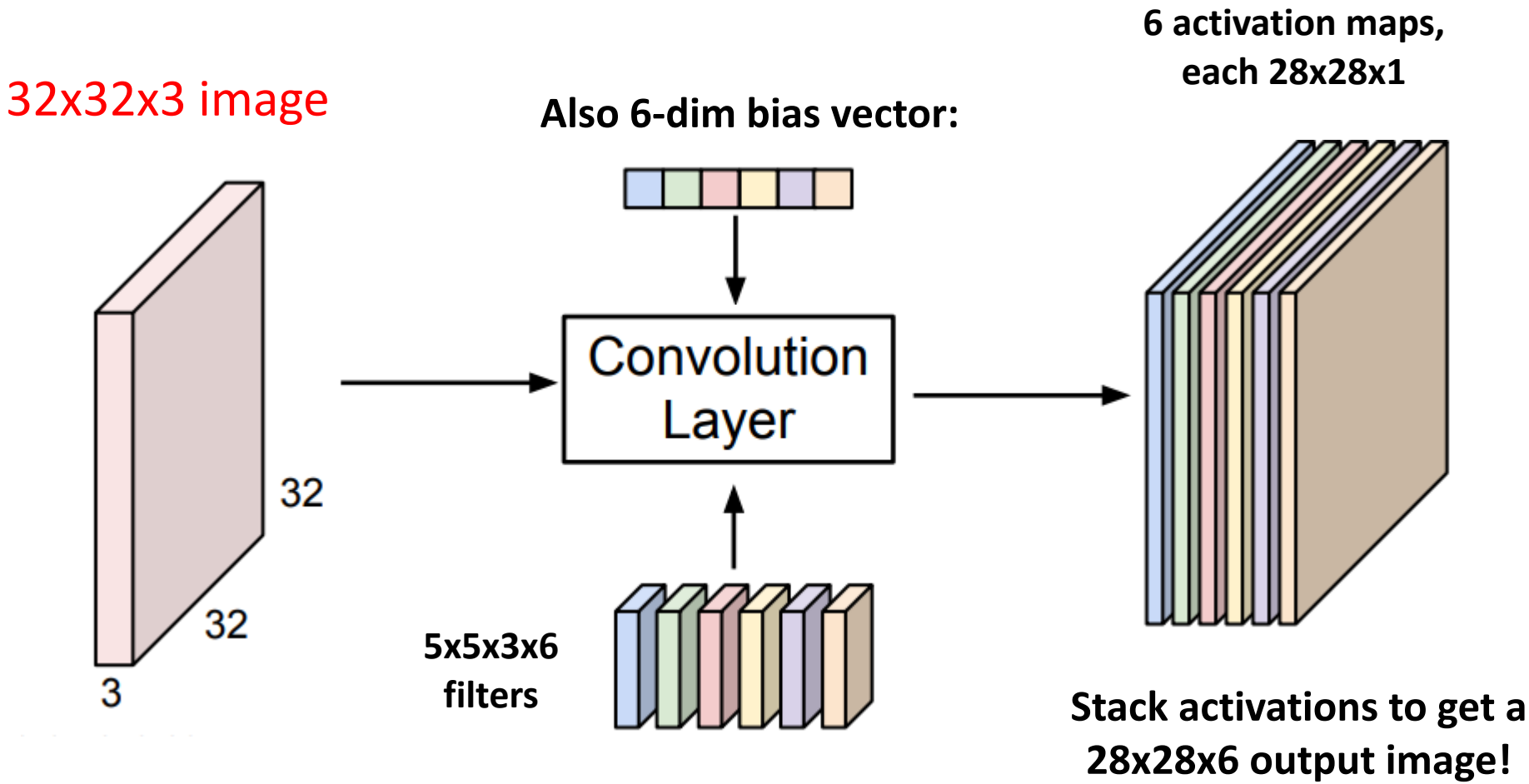


# Convolution Layer

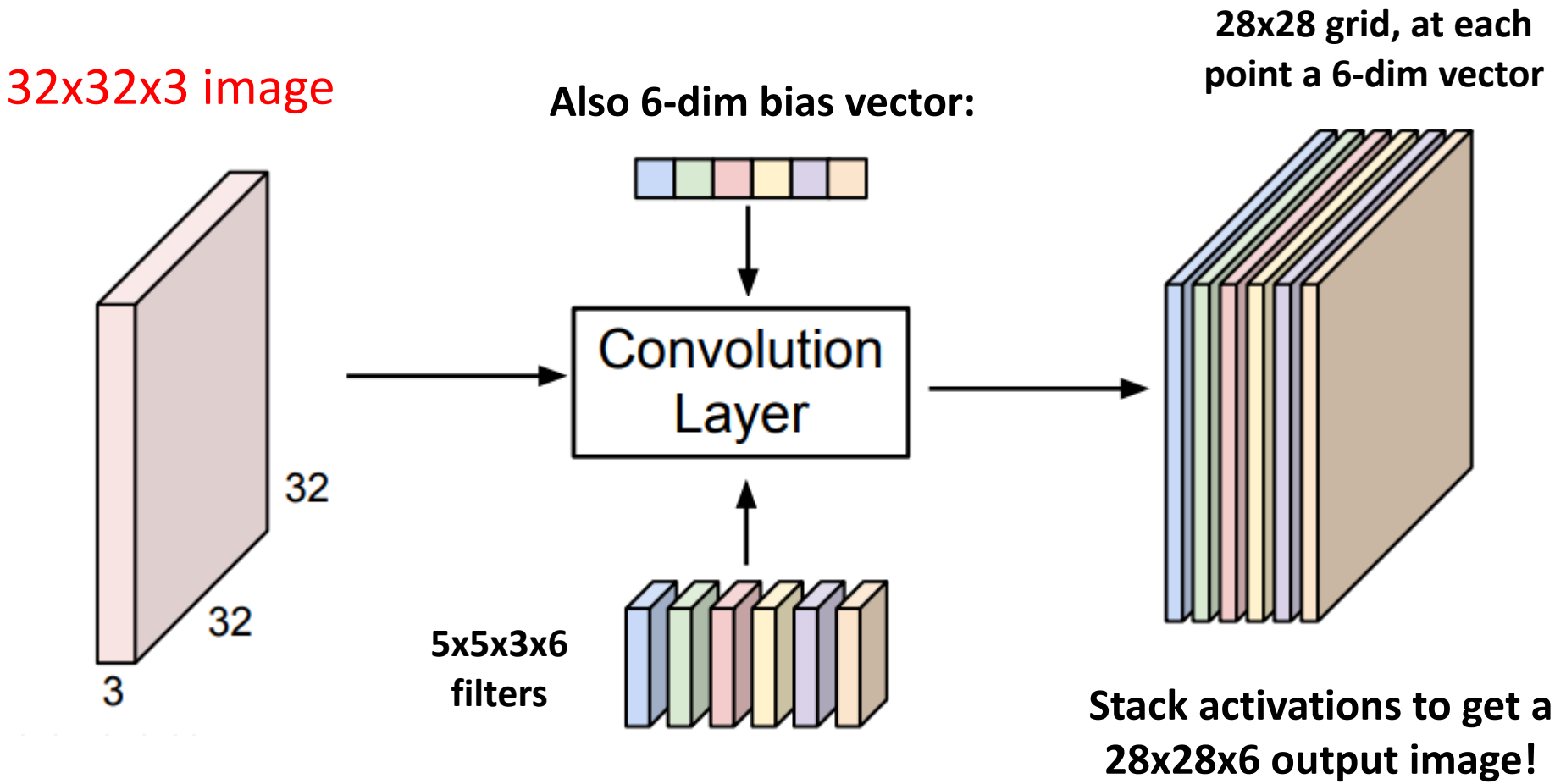



- We stack these up to get a "new image" of size 28x28x6!
  - **depth** of the output volume equals to the number of filters

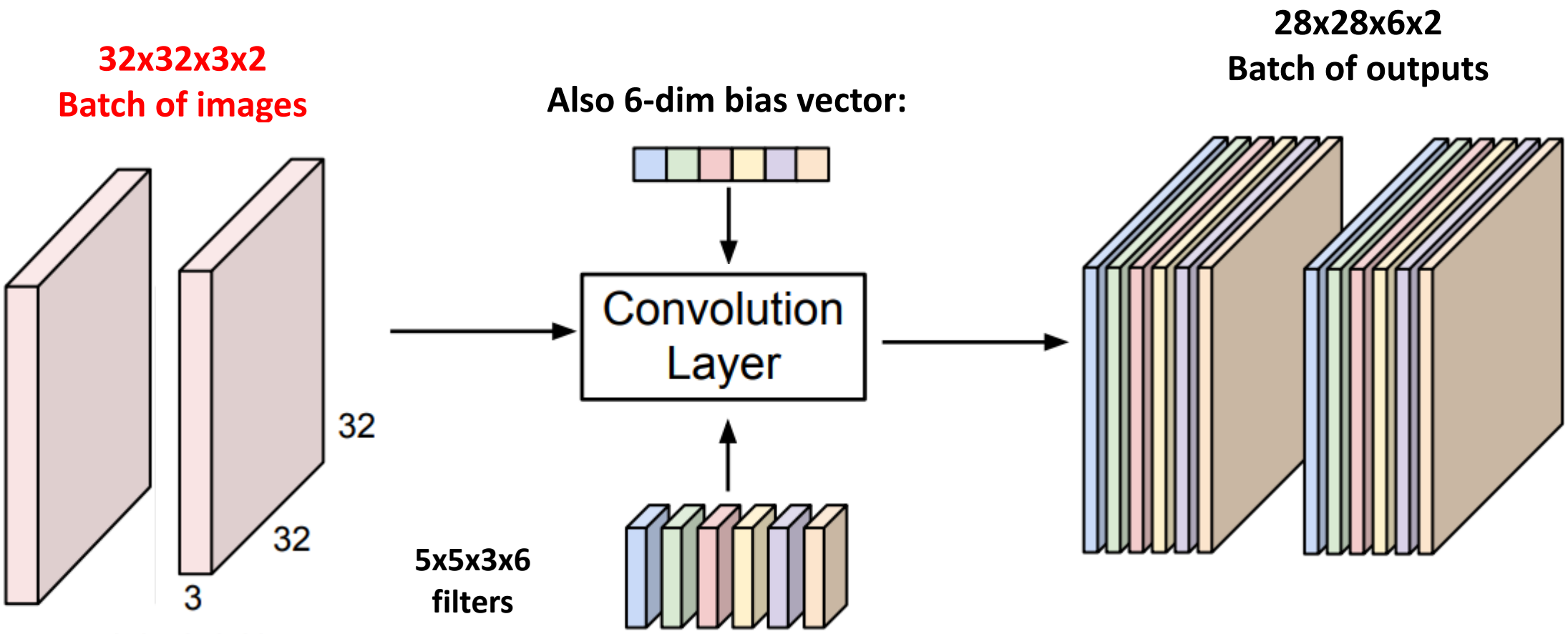
# Convolution Layer



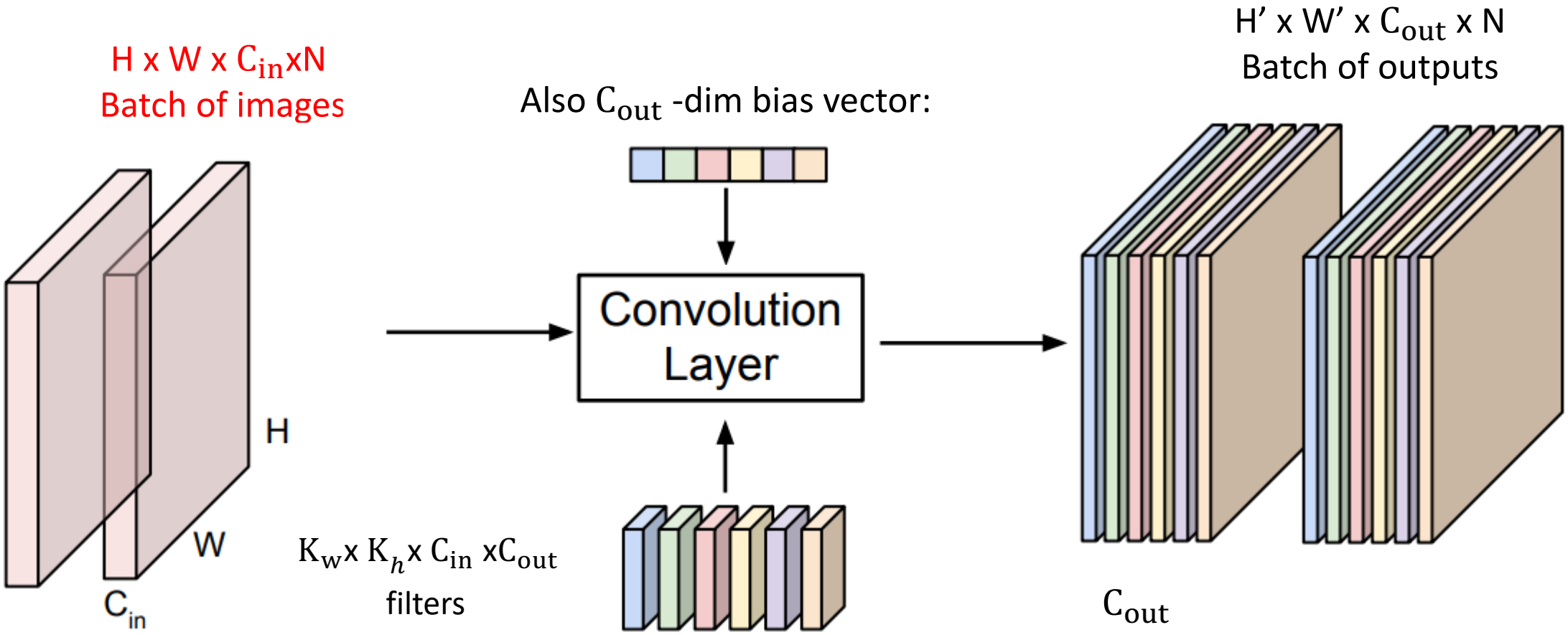
# Convolution Layer



# Convolution Layer

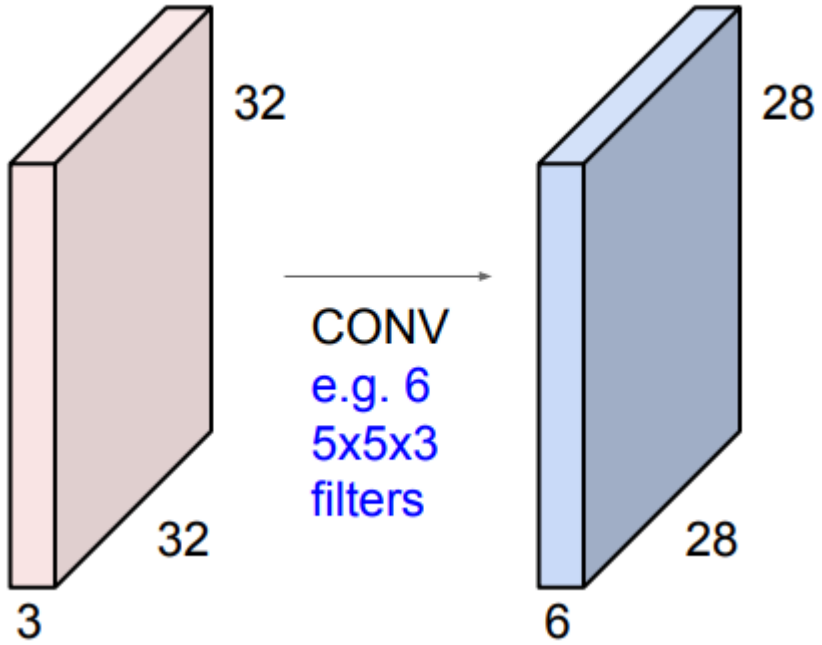


# Convolution Layer



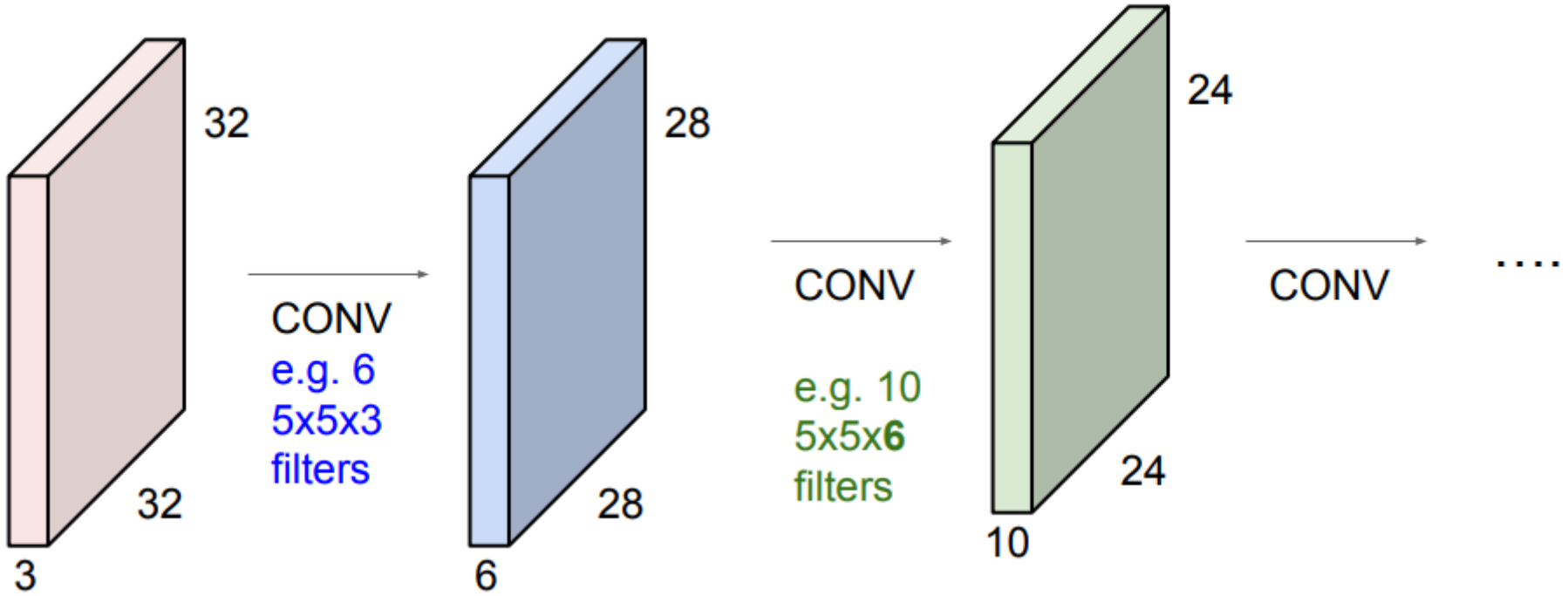
# ConvNet

**Preview**: ConvNet is a sequence of Convolution Layers



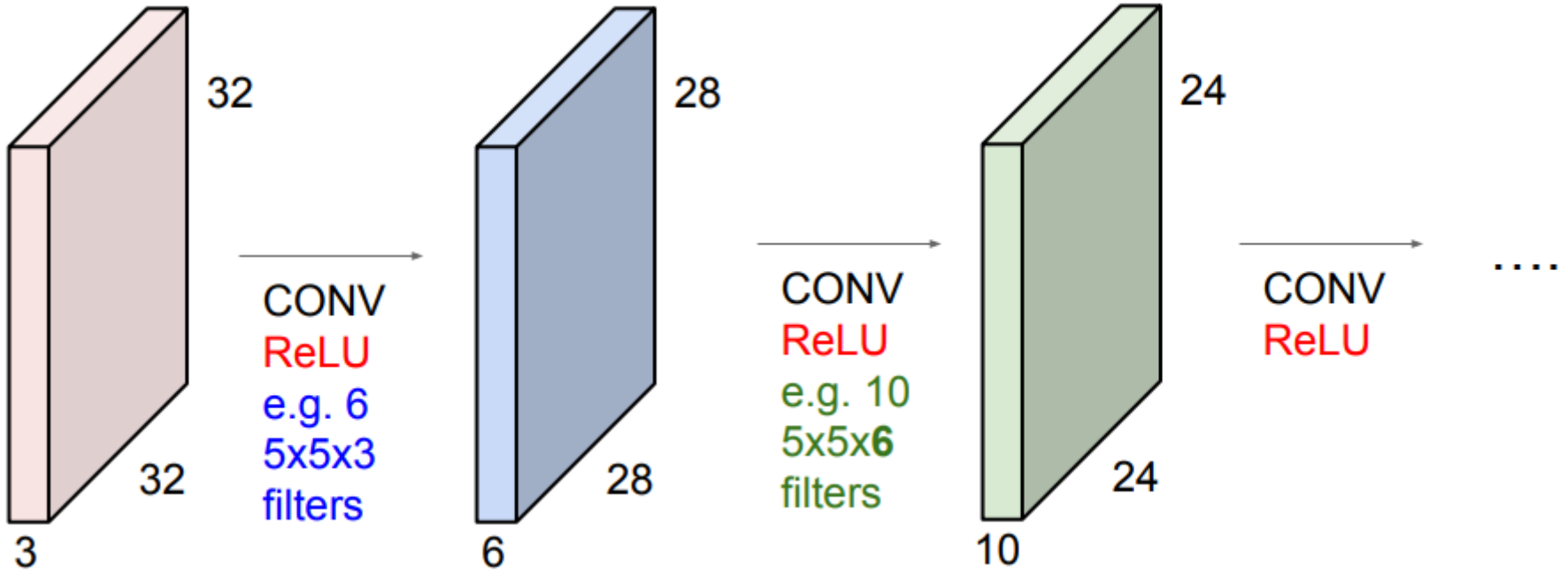
# ConvNet

**Preview**: ConvNet is a sequence of Convolution Layers

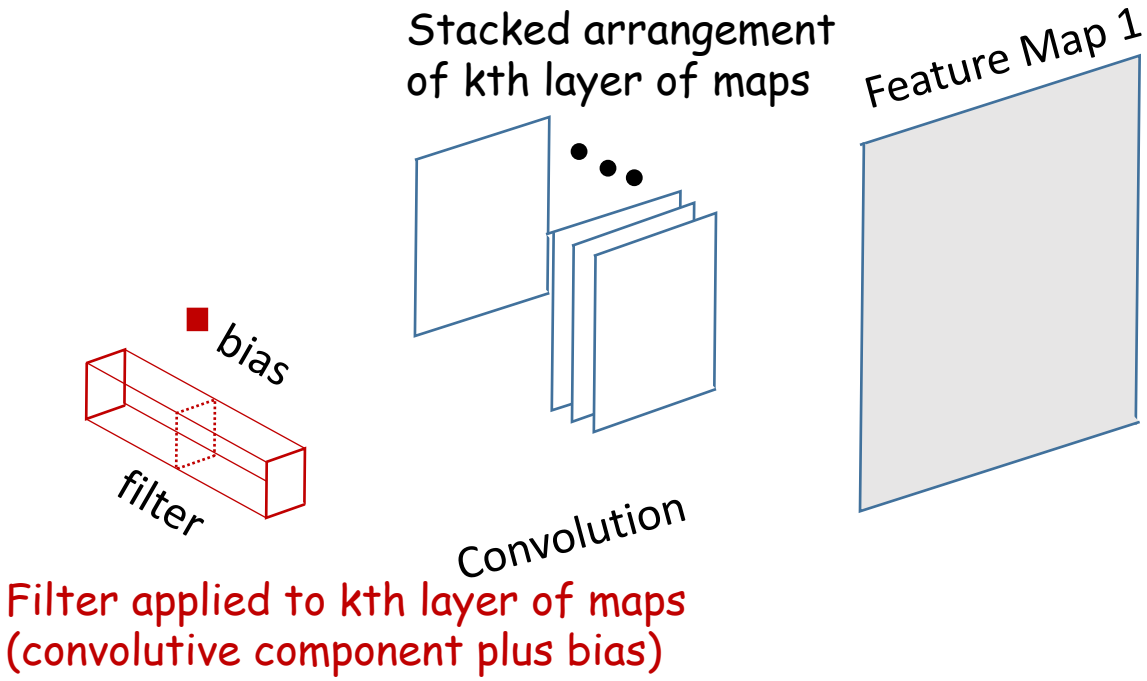


# ConvNet

**Preview**: ConvNet is a sequence of Convolution Layers, interspersed with activation functions

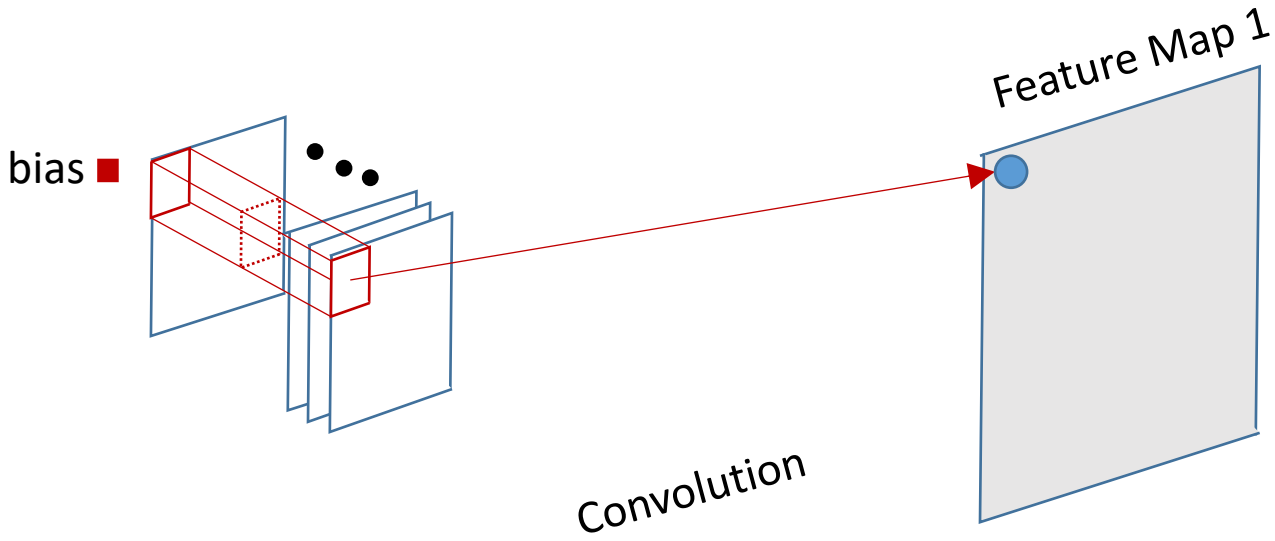


# A different view




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

# A different view

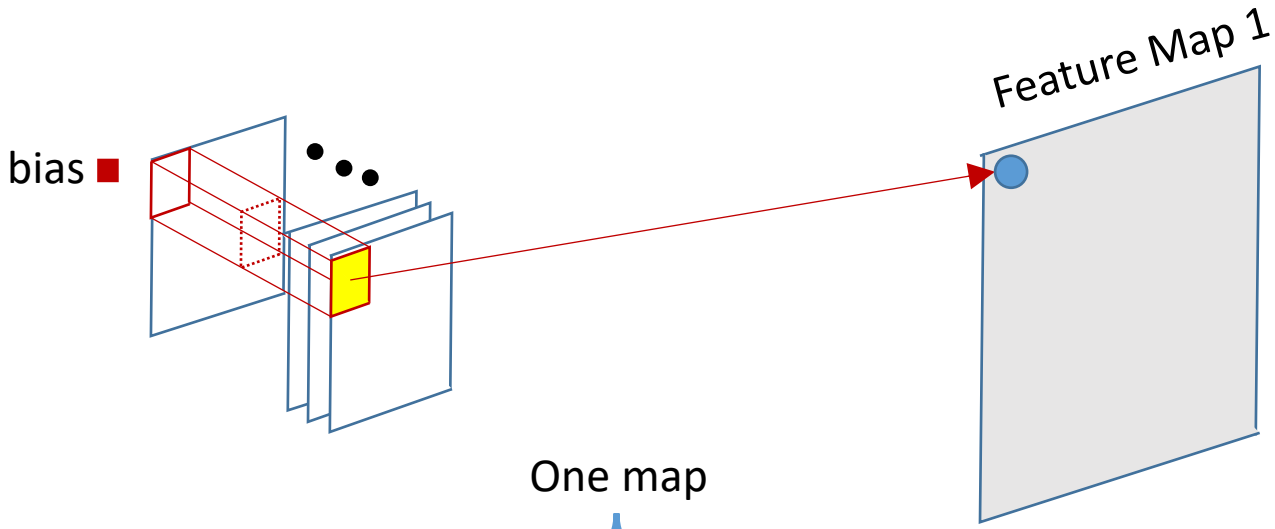



$s = 1$

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

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

# A different view

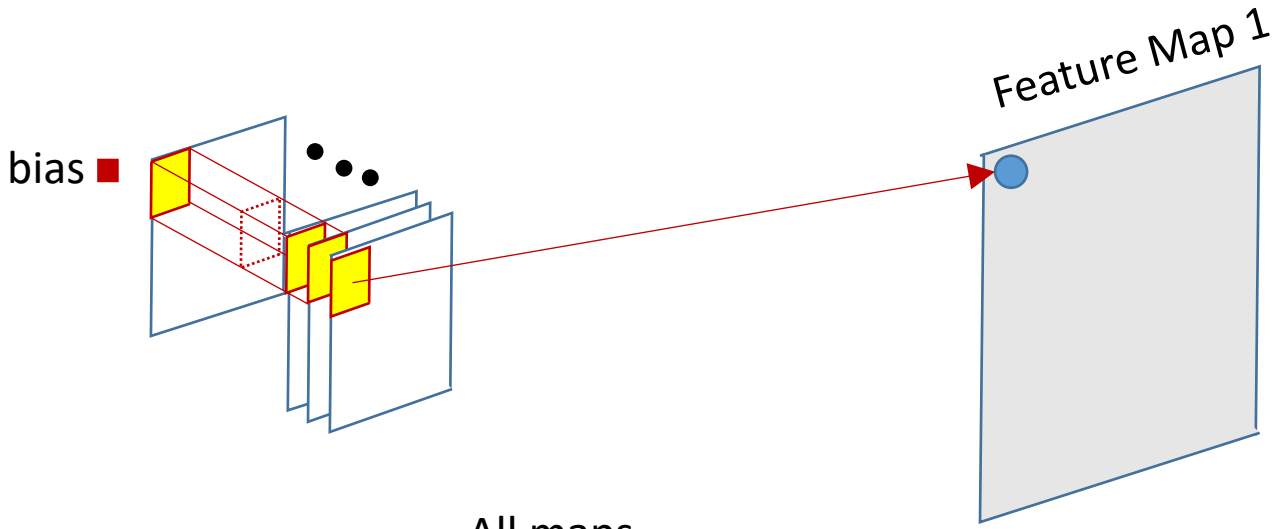



$s = 1$

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

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

# A different view

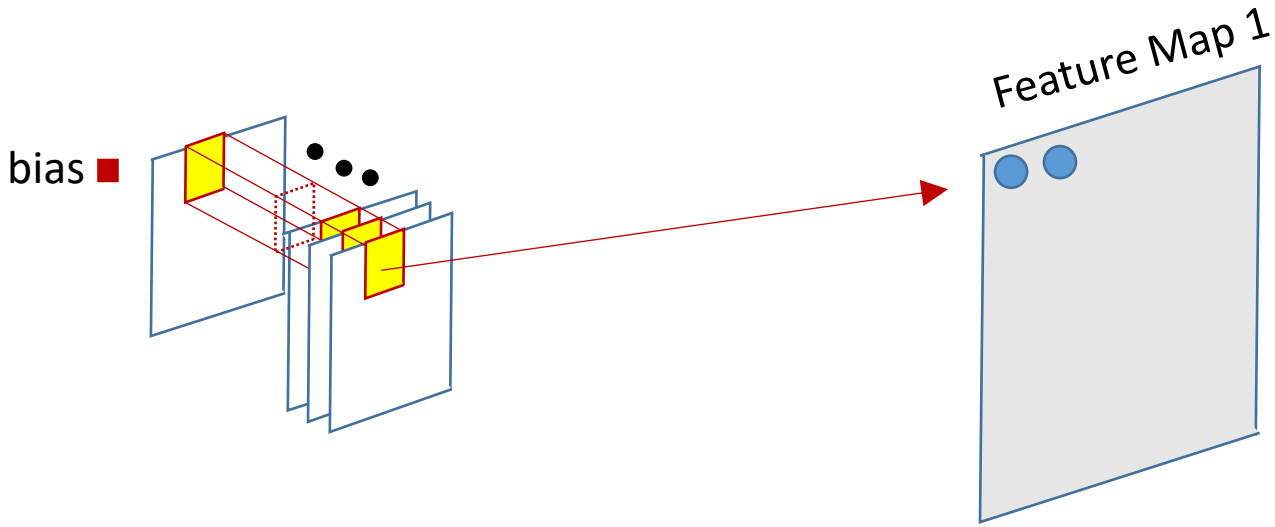



$s = 1$ All maps

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

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

# A different view

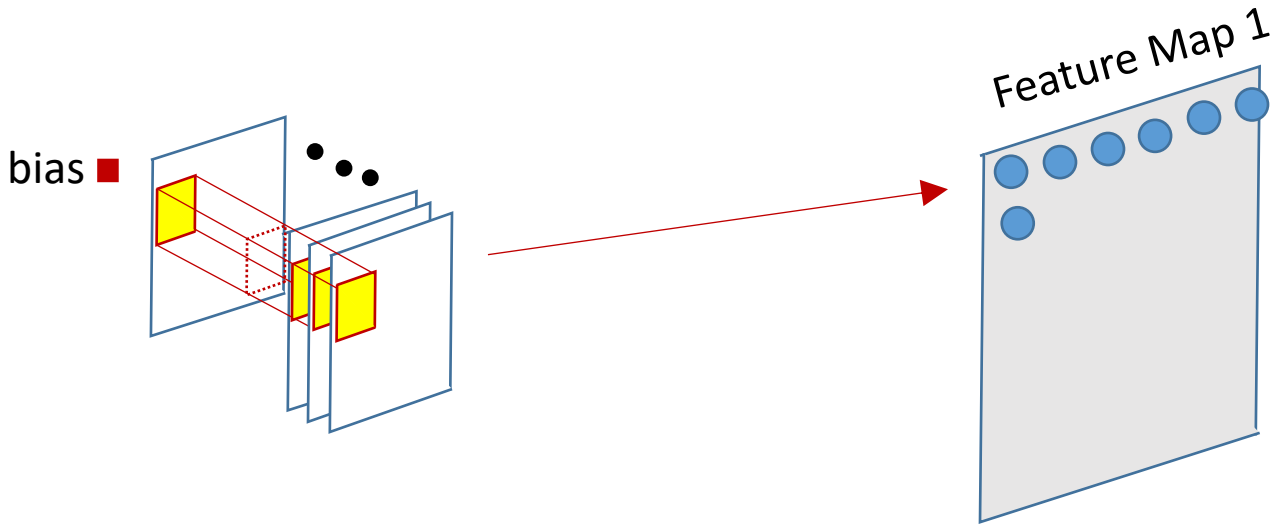



$s = 1$

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

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

# A different view

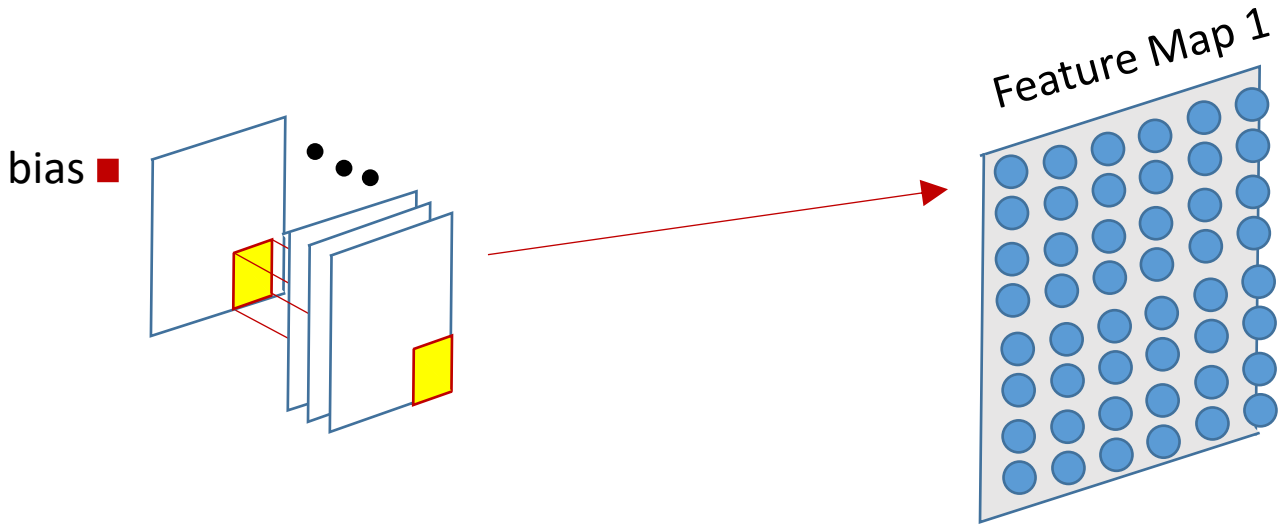



$s = 1$

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

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

# A different view

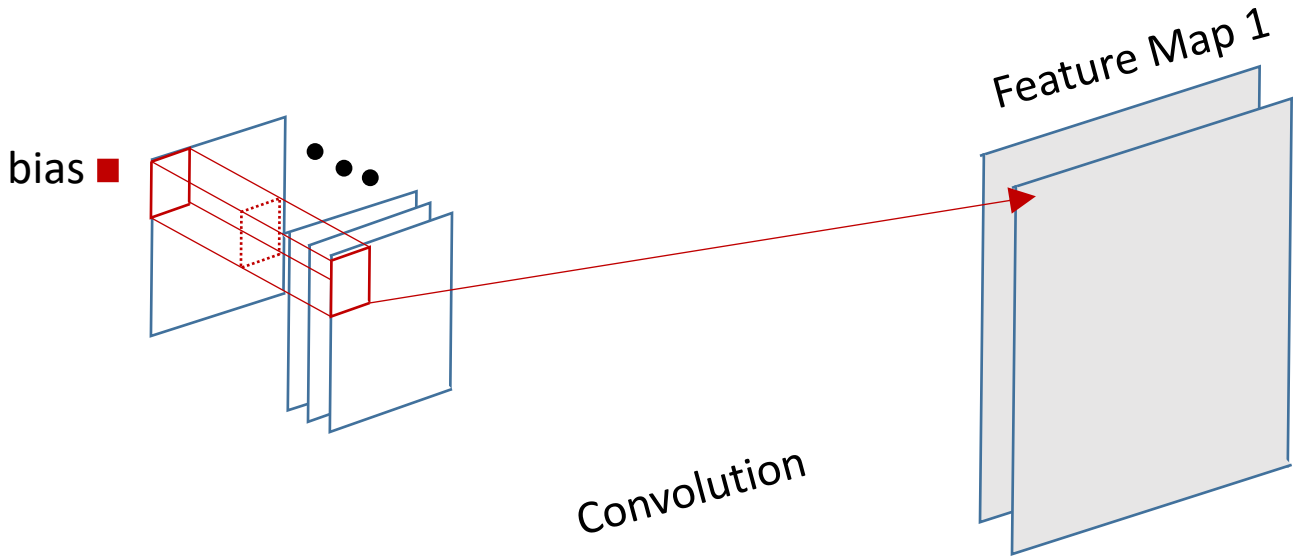



$s = 1$

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

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

# A different view

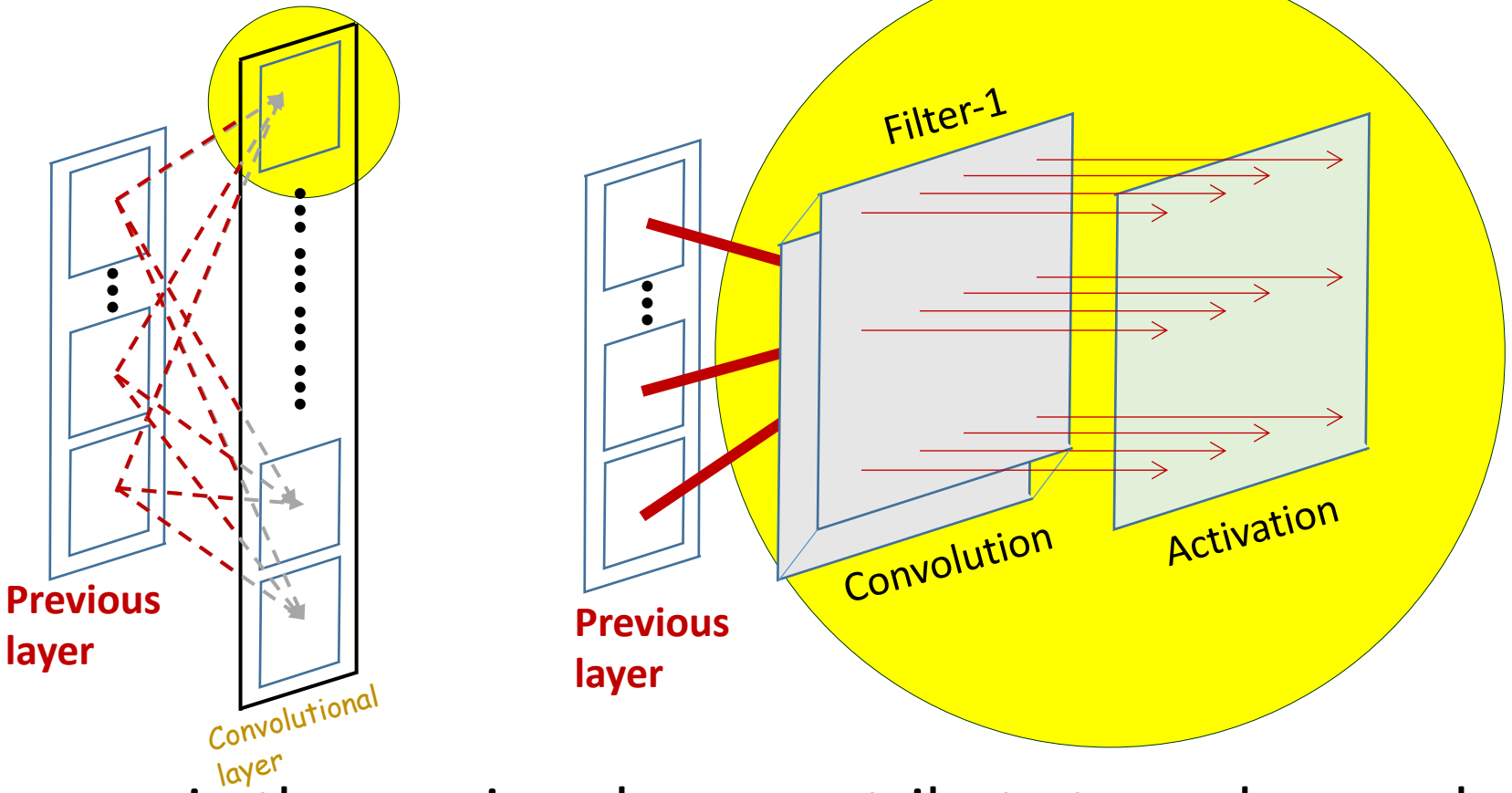



$s = 2$

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

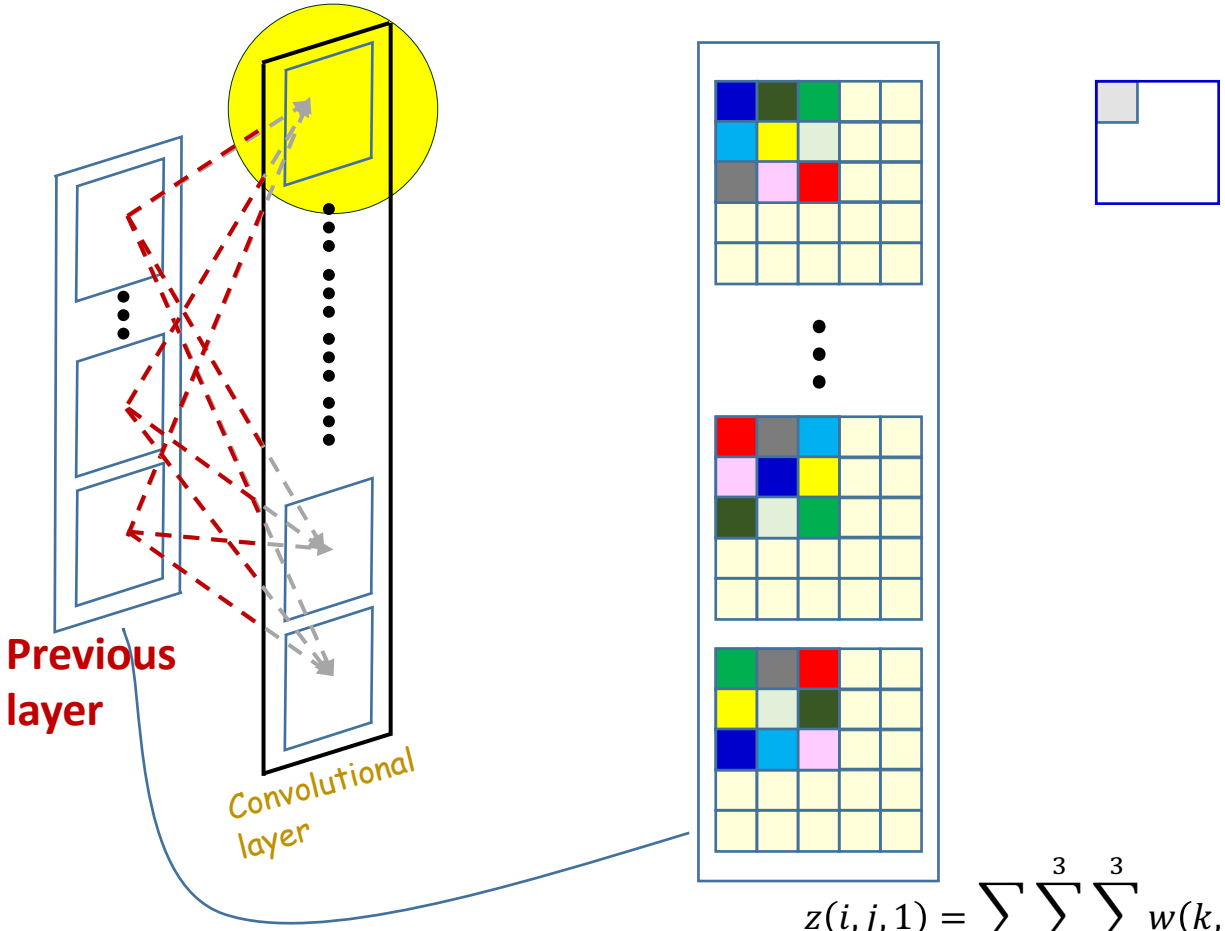
- The computation of the convolutive map at any location *sums* the convolutive outputs *at all planes*

# A convolutional layer




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

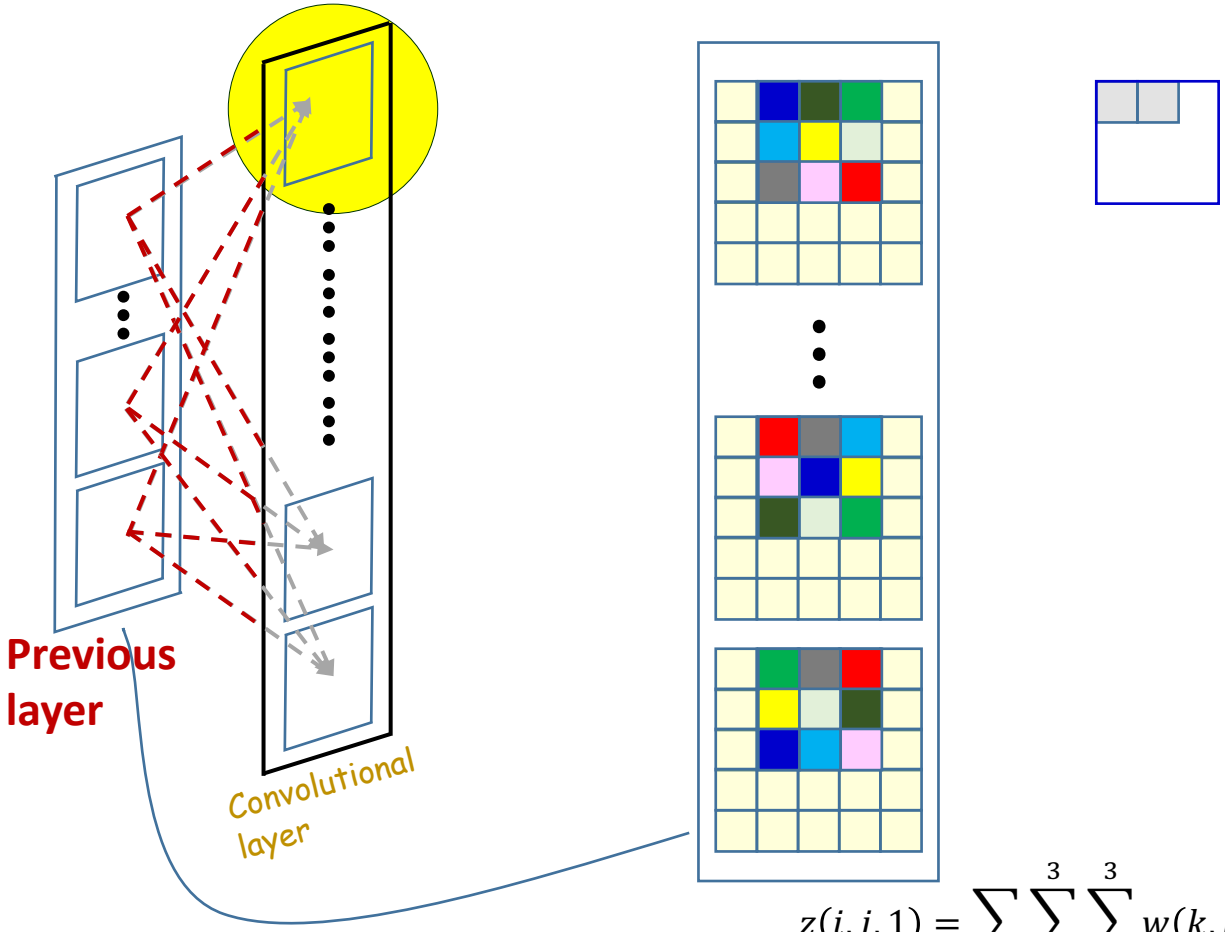
# What really happens




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

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

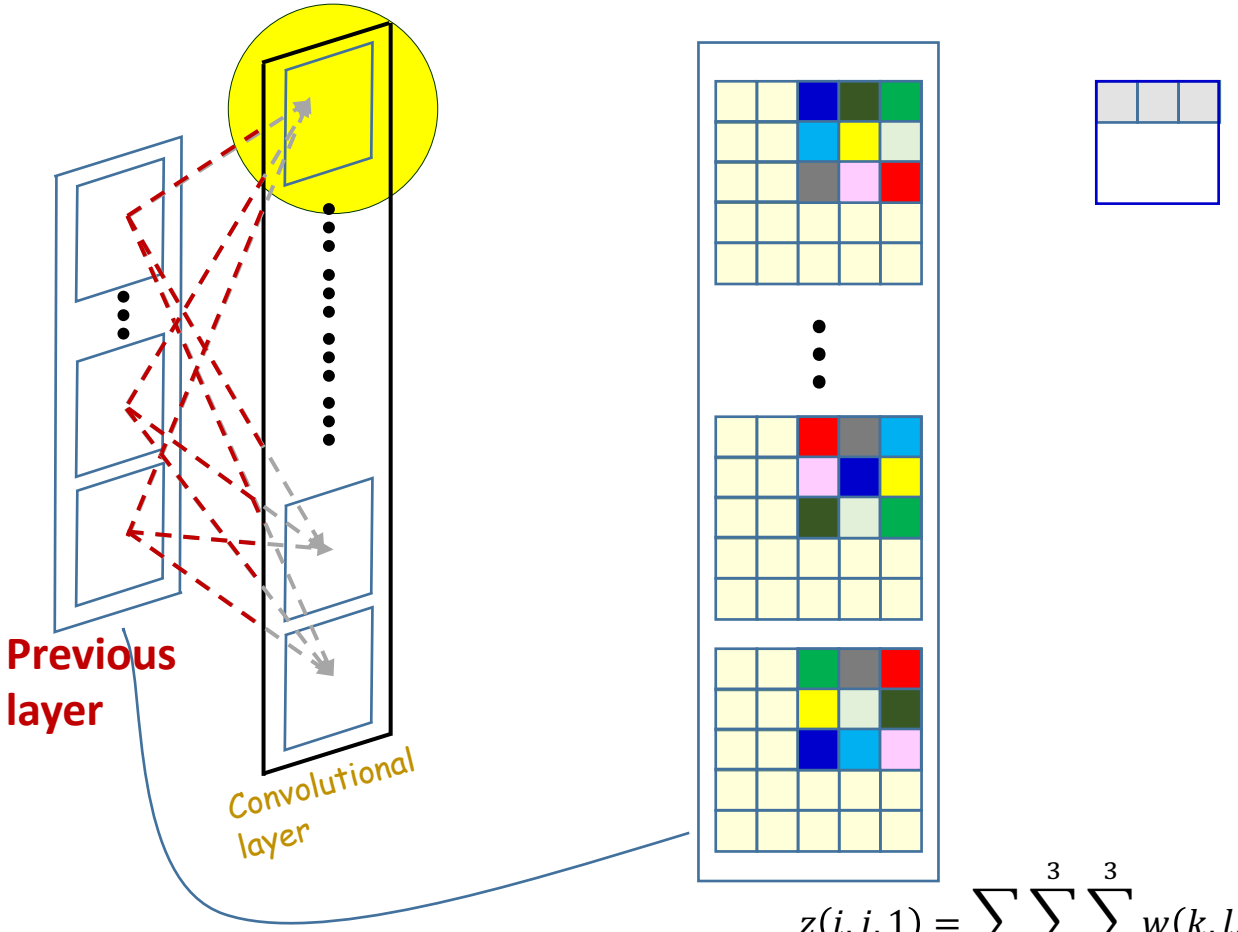
# What really happens




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

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

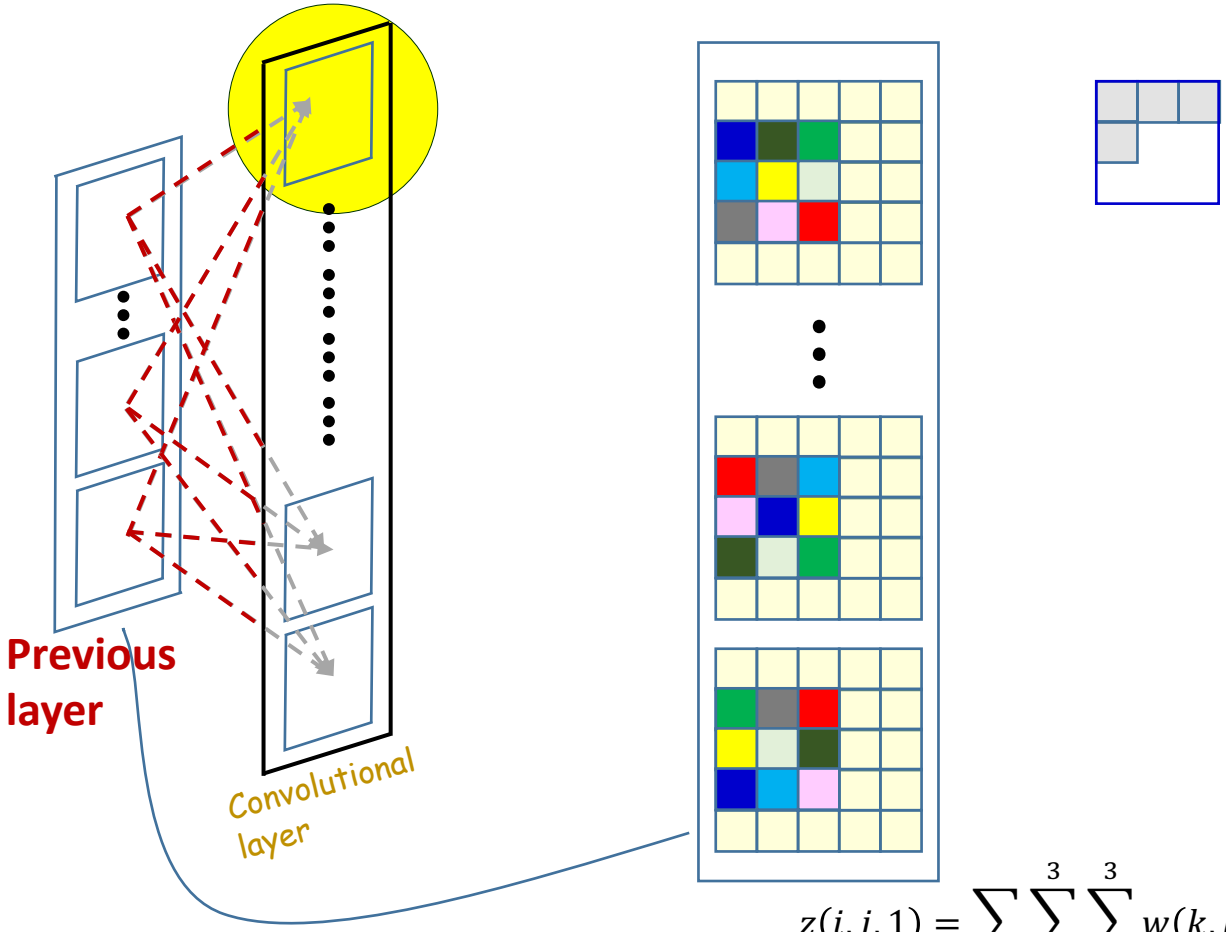
# What really happens




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

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

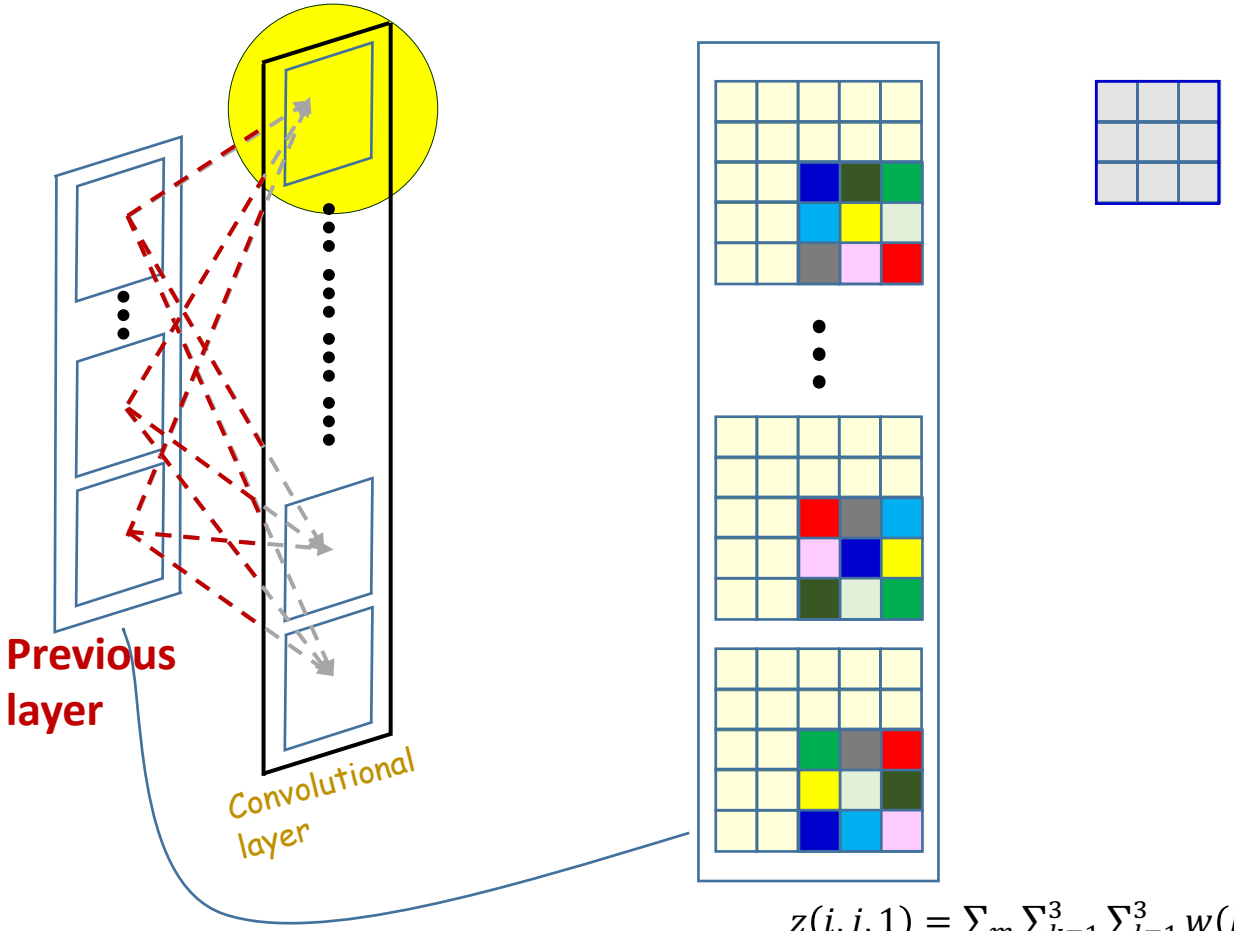
# What really happens




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

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

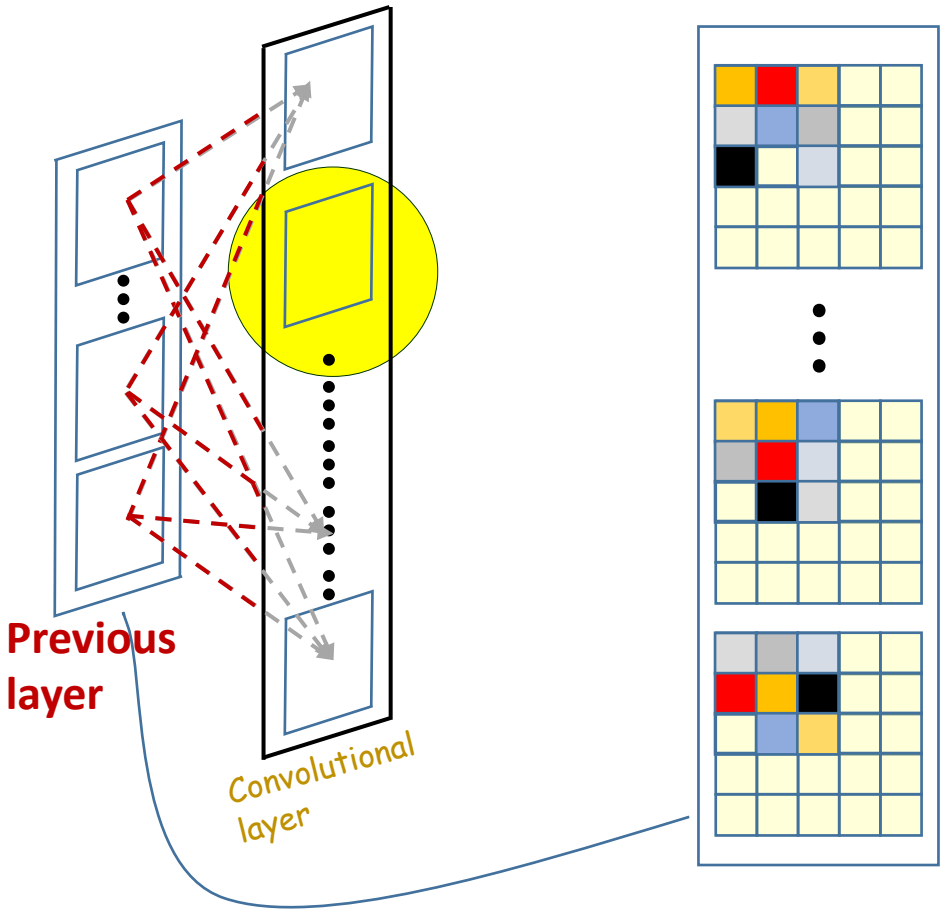
# What really happens



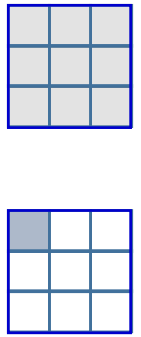

$$z(i,j,1) = \sum_m \sum_{k=1}^{3} \sum_{l=1}^{3} w(k,l,m,1) I(i+l-1, j+k-1, m) + b(1)$$

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

# What really happens



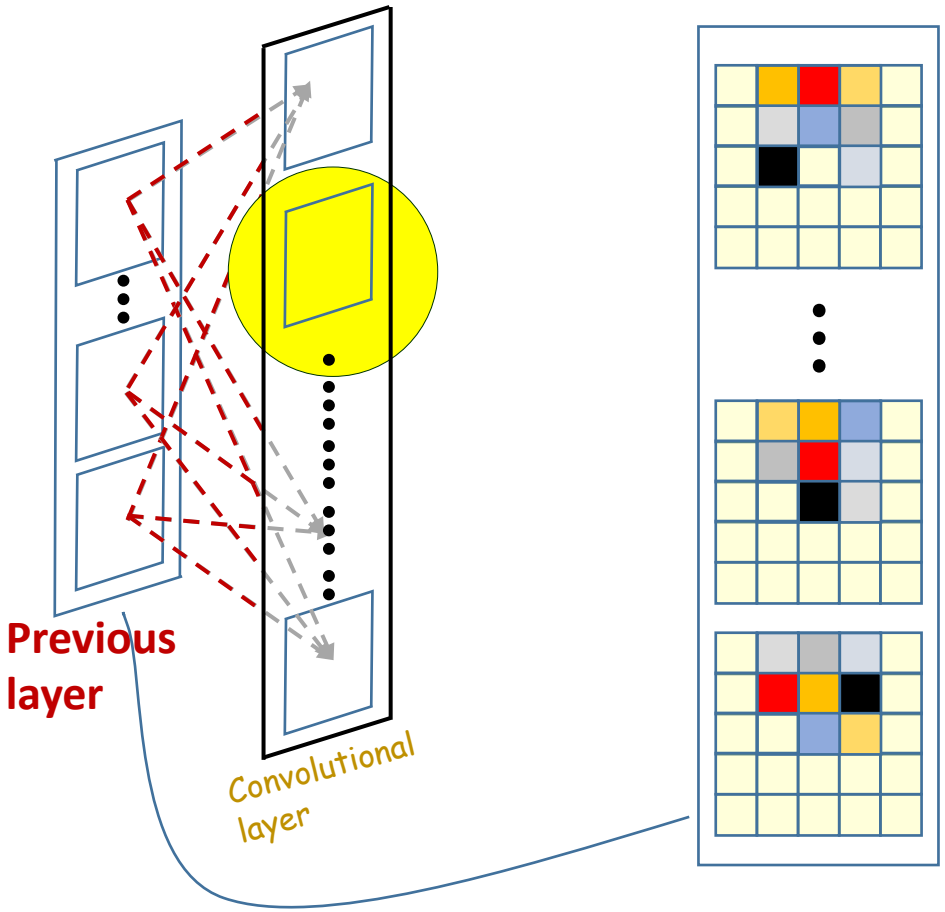


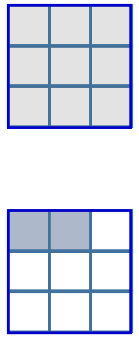
$$z(i,j,2) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(k,l,m,2) I(i+l-1, j+k-1, m) + b \; (2)$$

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

# What really happens



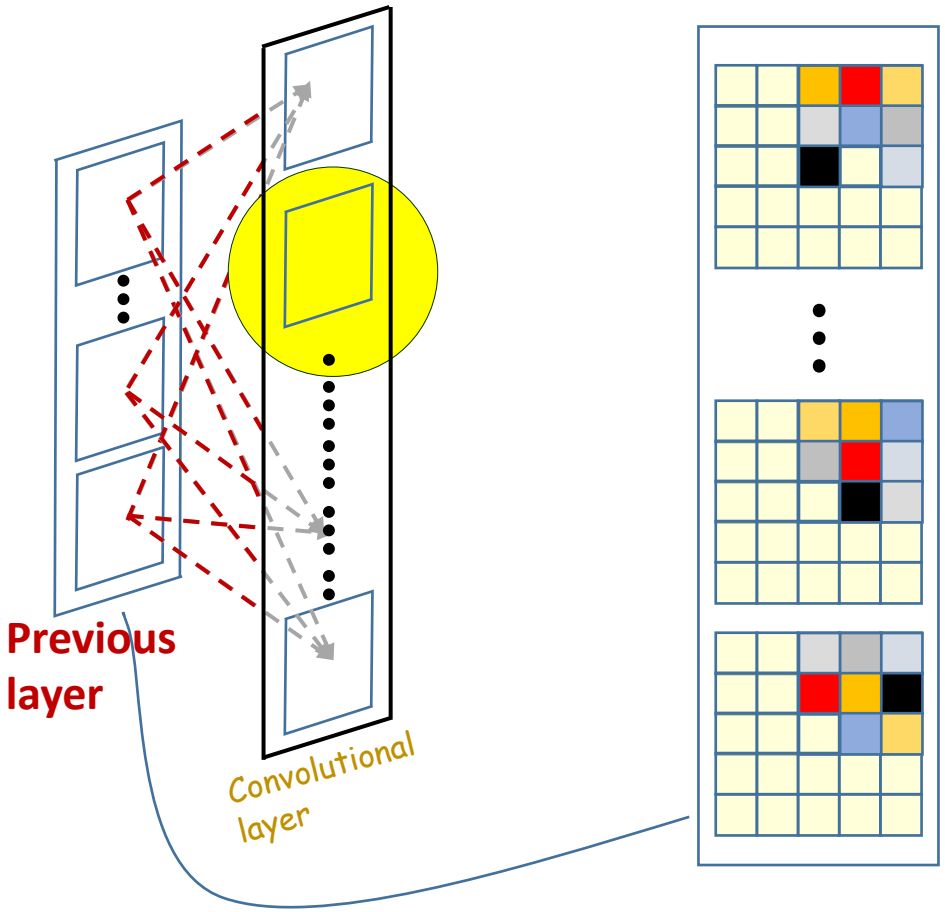


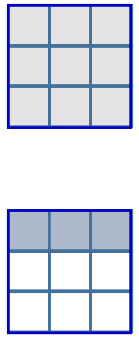
$$z(i,j,2) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(k,l,m,2) I(i+l-1, j+k-1, m) + b \; (2)$$

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

# What really happens



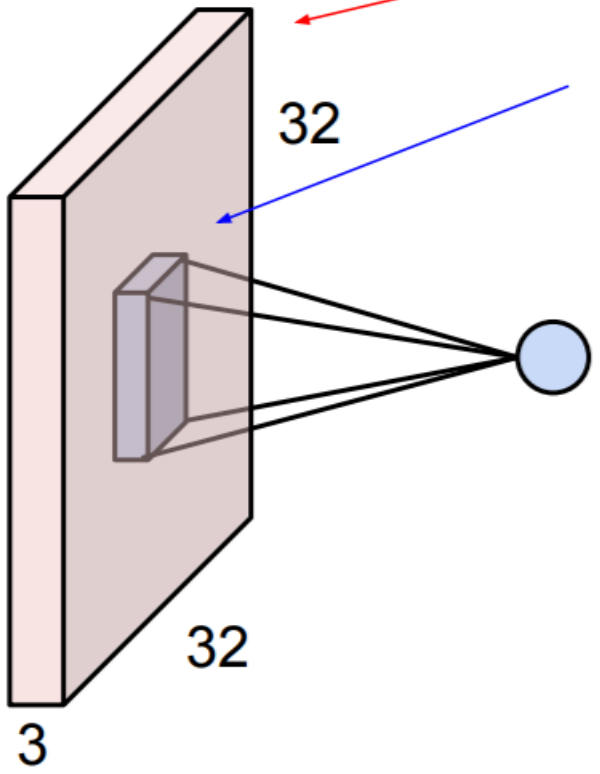


$$z(i,j,2) = \sum_{m} \sum_{k=1}^{3} \sum_{l=1}^{3} w(k,l,m,2) I(i+l-1, j+k-1, m) + b \; (2)$$

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

# Convolutional layer: neural view



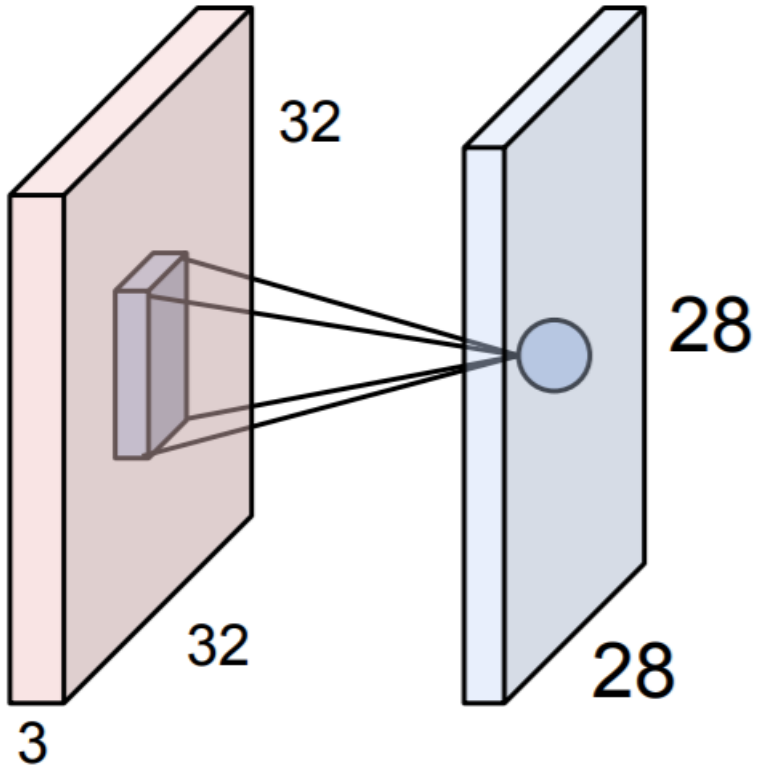

32x32x3 image

5x5x3 filter

It's just a neuron with local connectivity...

**1 number:**
the result of taking a dot product between the filter and this part of the image
(i.e. 5*5*3 = 75-dimensional dot product)

# Convolutional layer: neural view



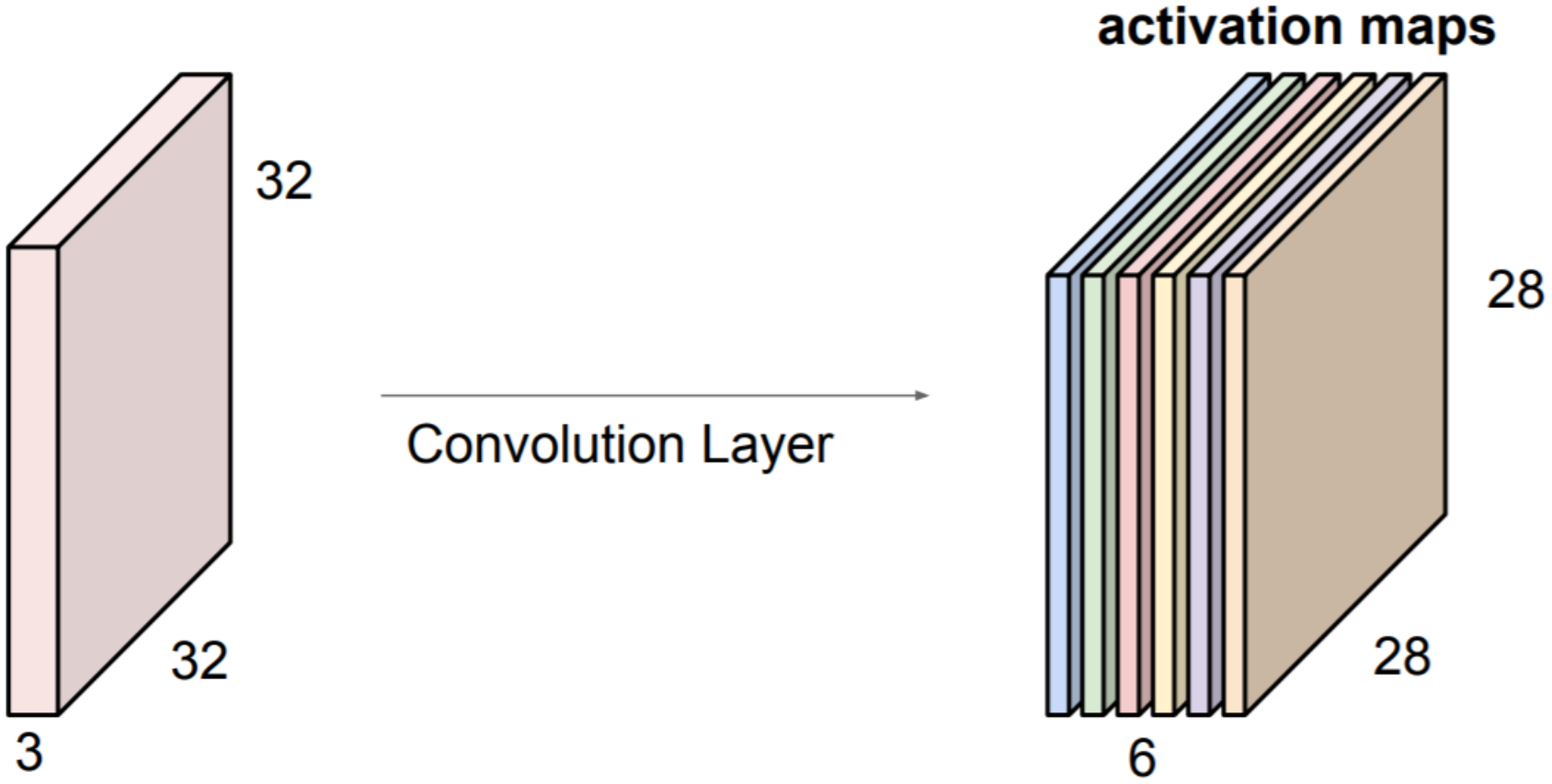

An activation map is a 28x28 sheet of neuron outputs:

1. Each is connected to a small region in the input
2. All of them share parameters “5x5x3”

“5x5 filter” => “5x5 receptive field for each neuron”

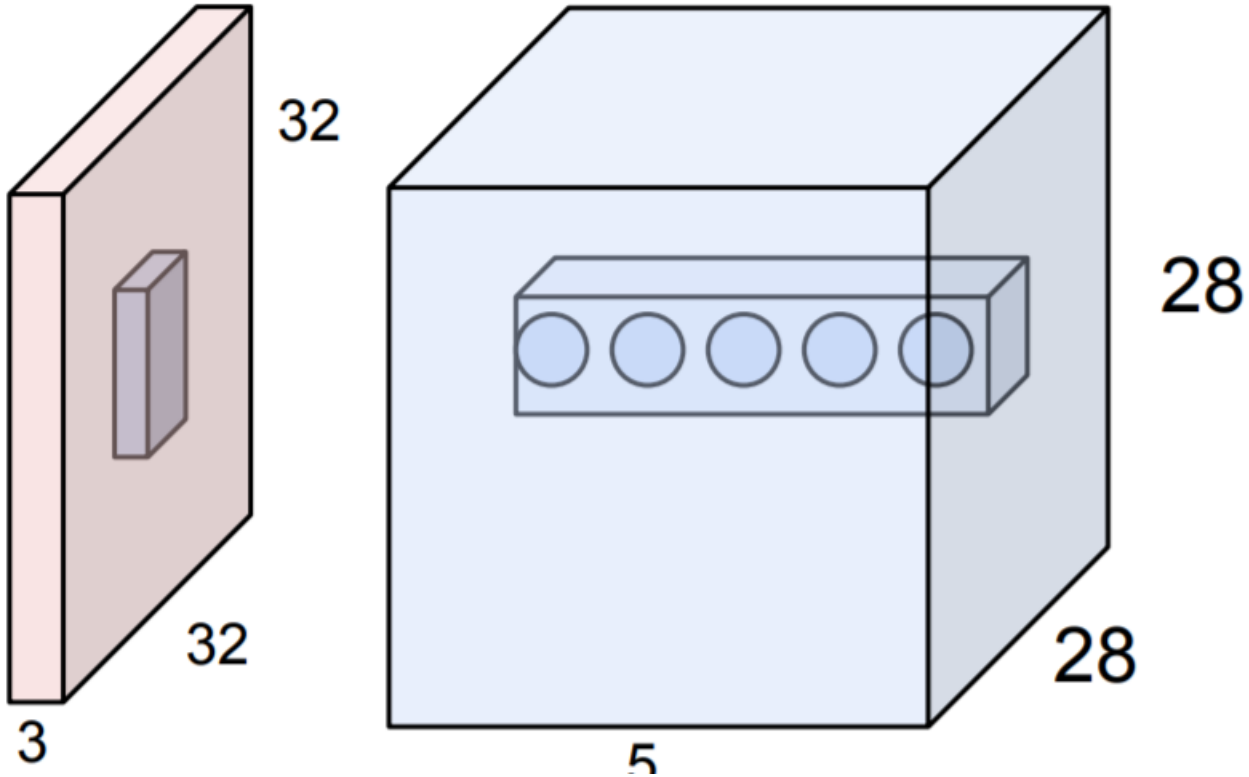
# Convolutional layer: neural view

- If we had 6 “5x5 filters”, we’ll get 6 separate activation maps:




There will be 6 different neurons all looking at the same region in the input volume
constrain the neurons in each depth slice to use the same weights and bias

# Convolutional layer: neural view



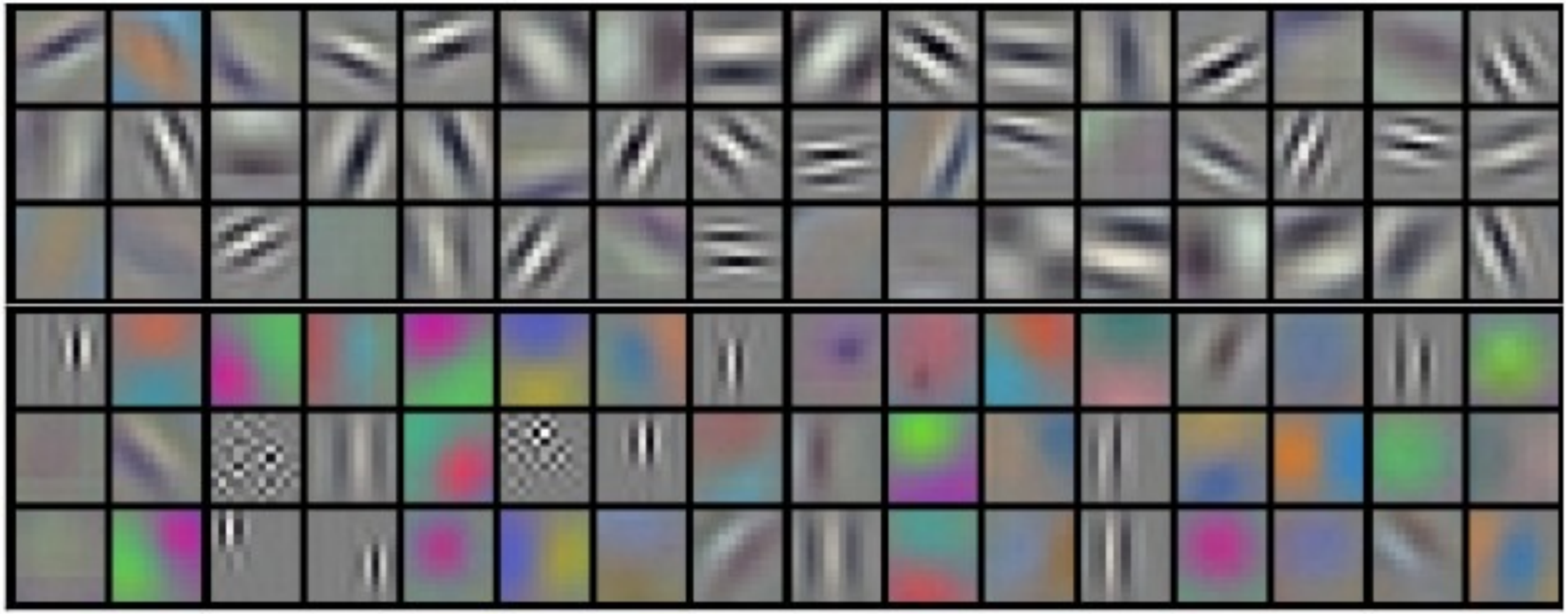

set of neurons that are all looking at the same region of the input as a **depth column**

E.g. with 5 filters,
CONV layer consists of neurons arranged in a 3D grid (28x28x5)

There will be 5 different neurons all looking at the same region in the input volume

# Alexnet: the first layer filters

- filters learned by Krizhevsky et al.
  - Each of the 96 filters shown here is of size [11x11x3]
  - and each one is shared by the 55*55 neurons in one depth slice



one filter =>
one activation map

Activations:

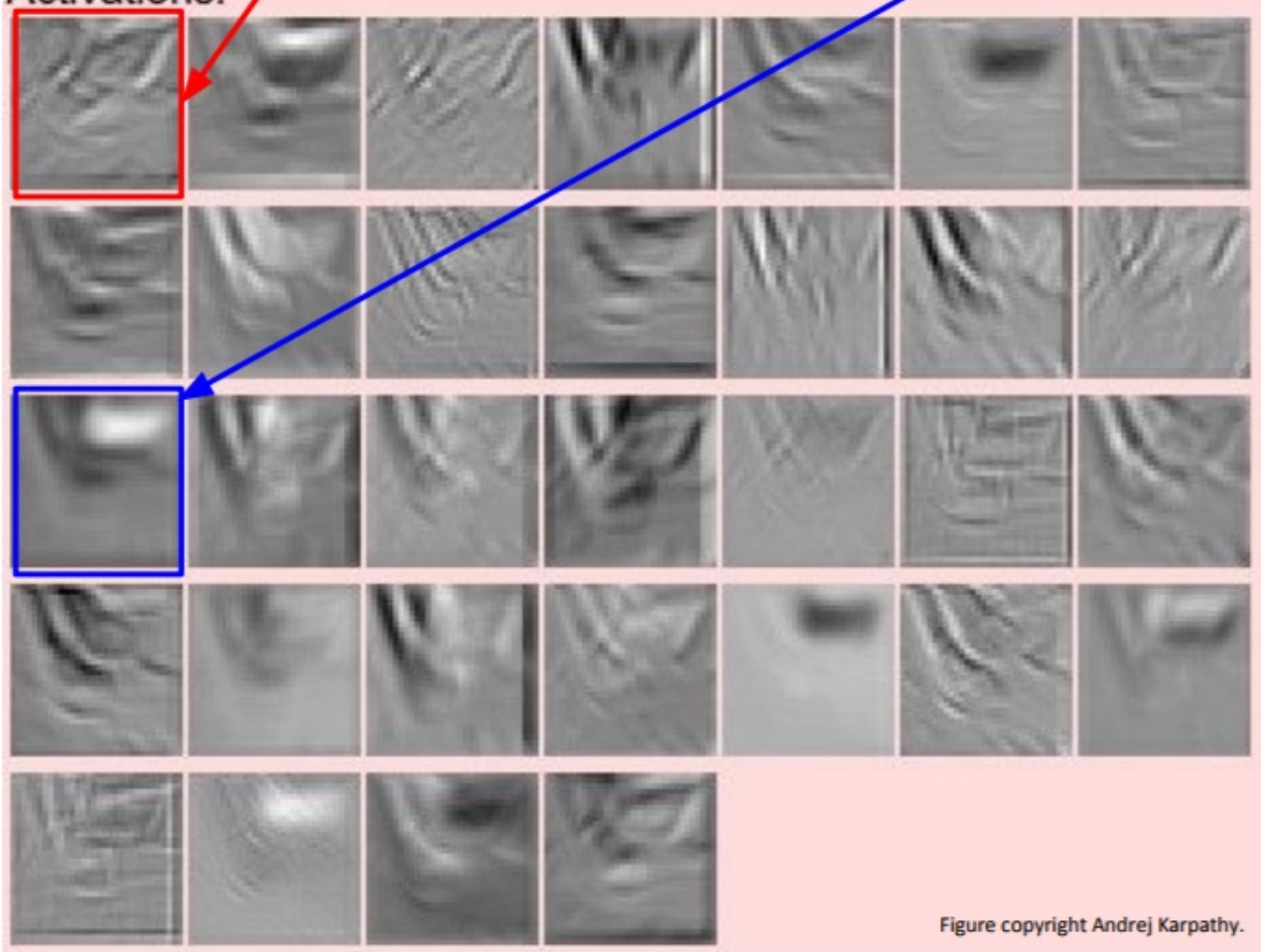


Figure copyright Andrej Karpathy.

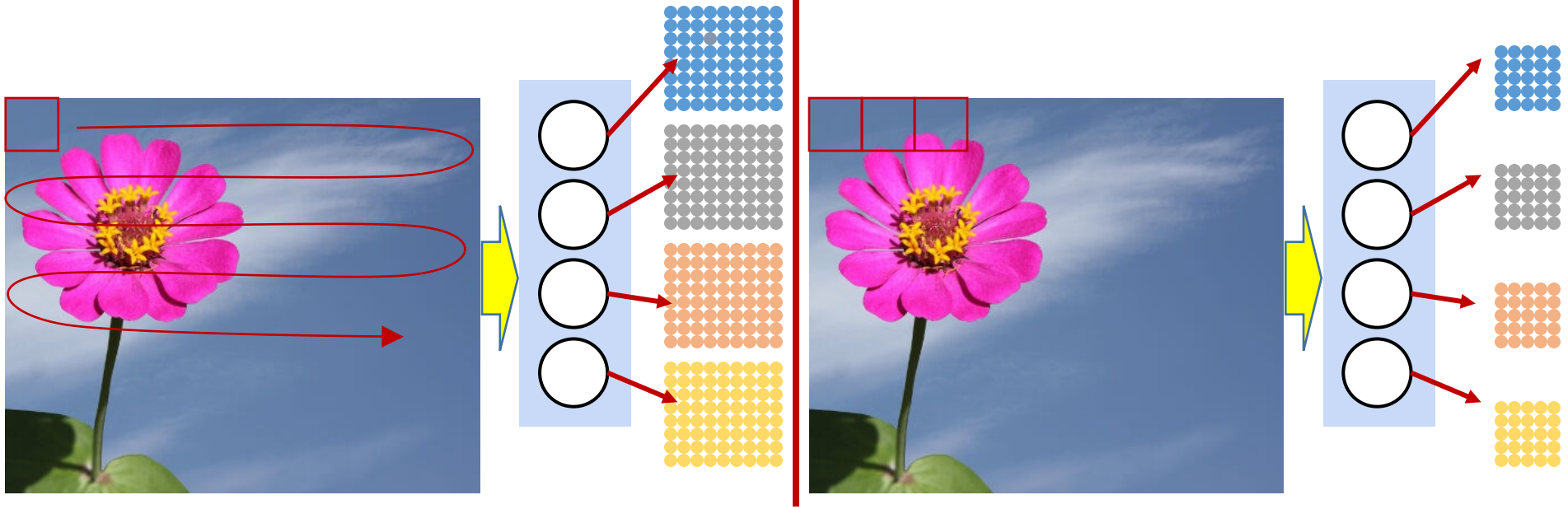
example 5x5 filters
(32 total)

We call the layer convolutional because it is related to convolution of two signals:

$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1, y-n_2]$$

elementwise multiplication and sum of a filter and the signal (image)

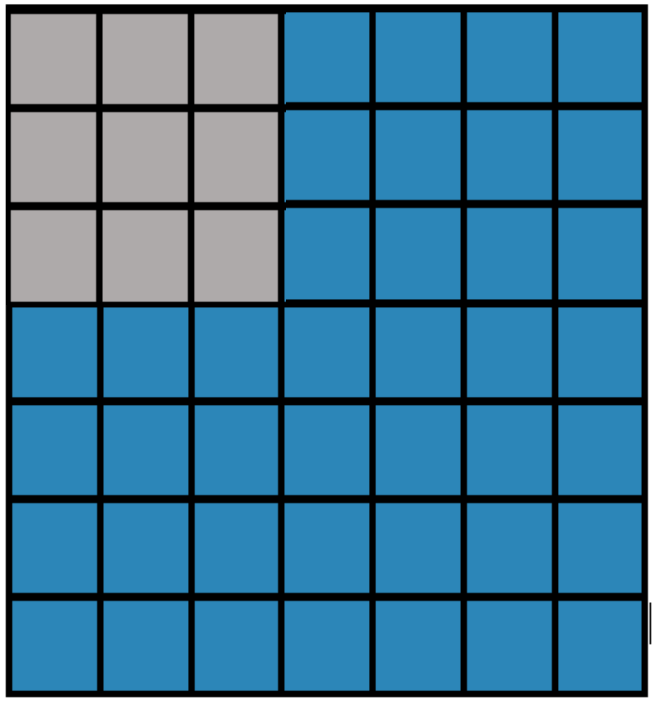
# Convolutional “Stride”



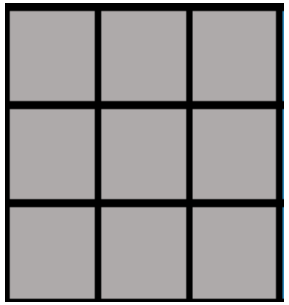
- The scans of the individual “filters” may advance by more than one pixel at a time
  - The “stride” may be greater than 1
  - Effectively increasing the granularity of the scan
    - Saves computation, sometimes at the risk of losing information
- This will result in a reduction of the size of the resulting maps
  - They will shrink by a factor equal to the stride
- This can happen at any layer

# Convolutional filter: stride

**Stride = 2**



7x7 input

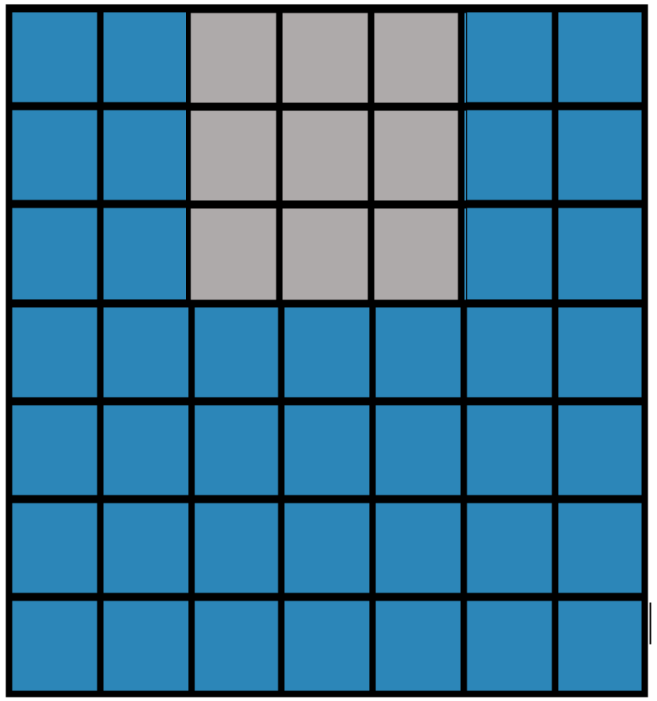


3x3 filter

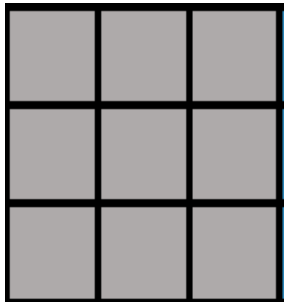
# Convolutional filter: stride

**Stride = 2**

filters jump 2 pixels at a time as we slide them around



7x7 input

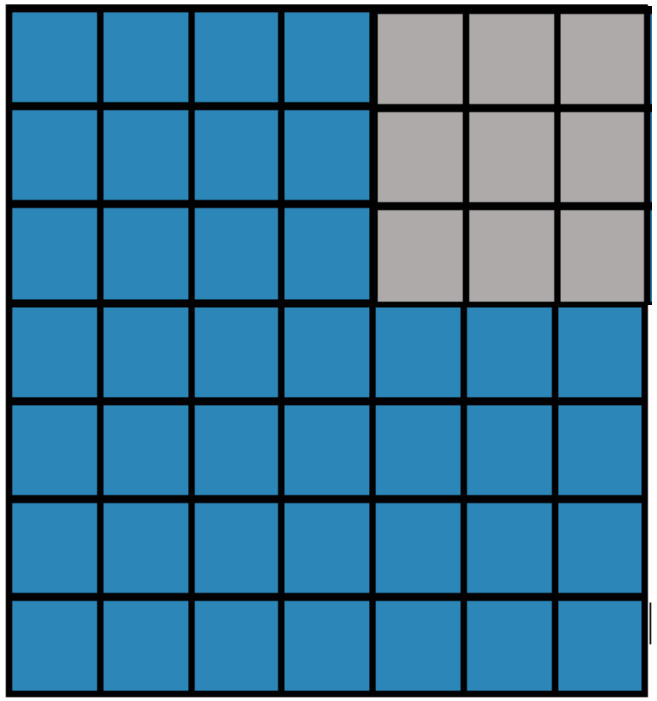


3x3 filter

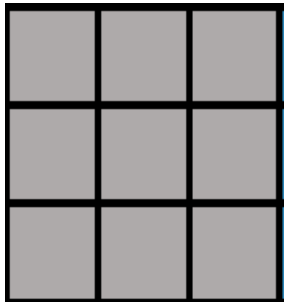
# Convolutional filter: stride

**Stride = 2**

filters jump 2 pixels at a time as we slide them around



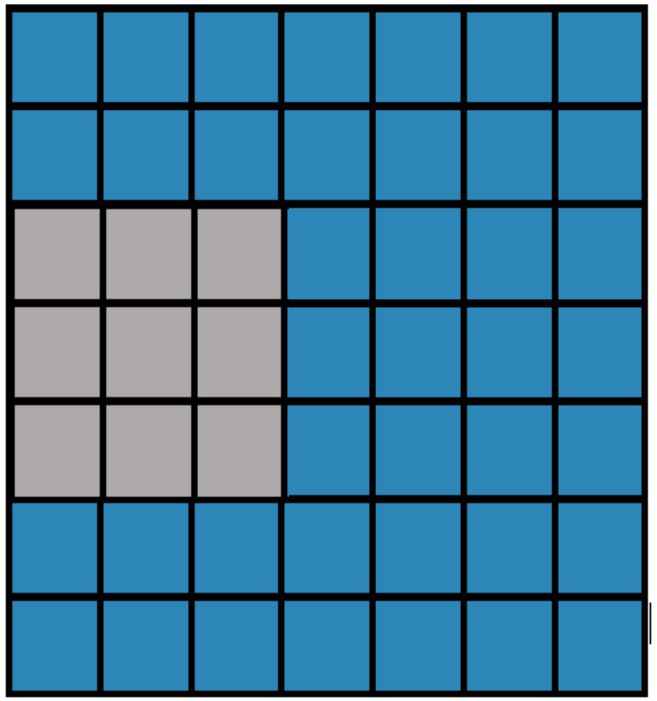
7x7 input



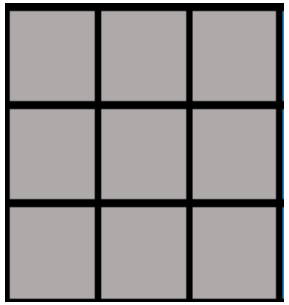
3x3 filter

# Convolutional filter: stride

**Stride = 2**



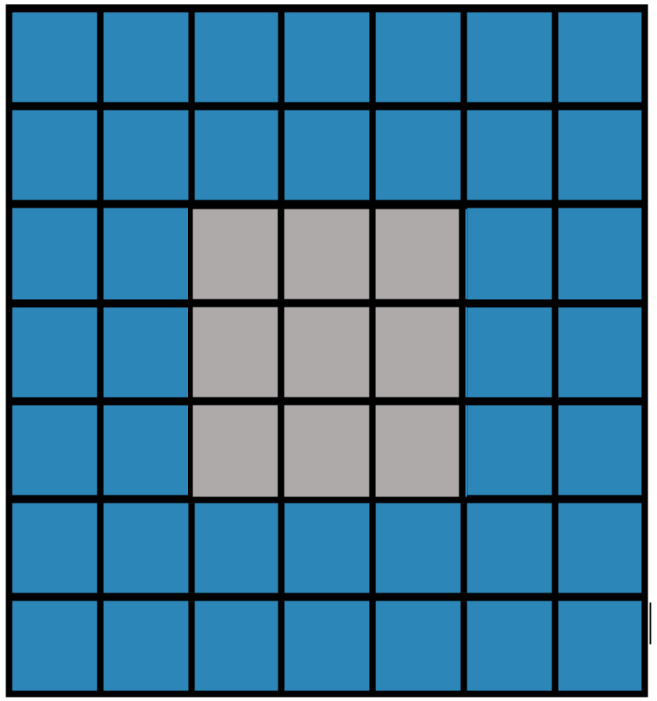
7x7 input



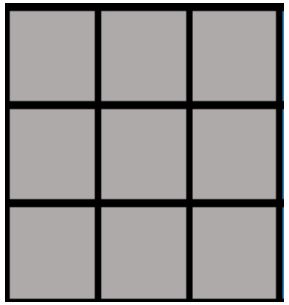
3x3 filter

# Convolutional filter: stride

**Stride = 2**



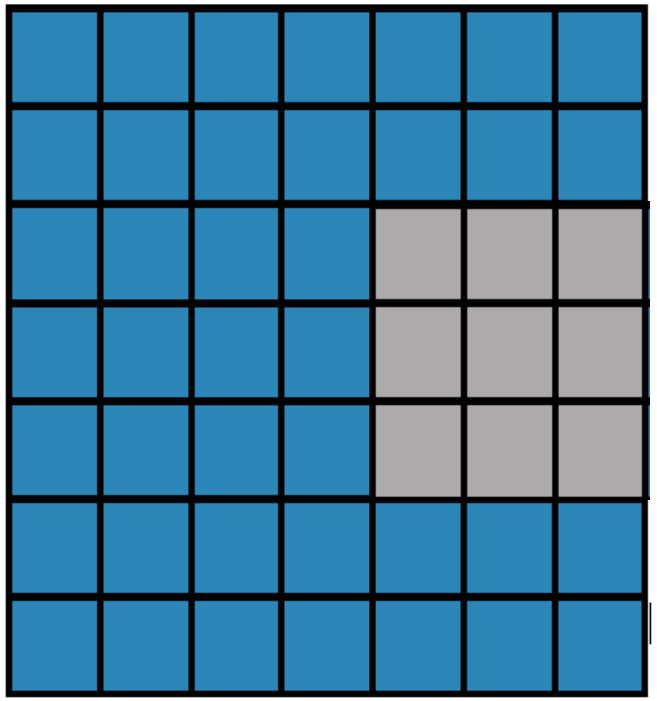
7x7 input



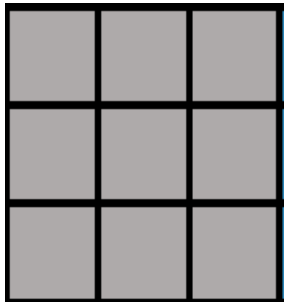
3x3 filter

# Convolutional filter: stride

**Stride = 2**



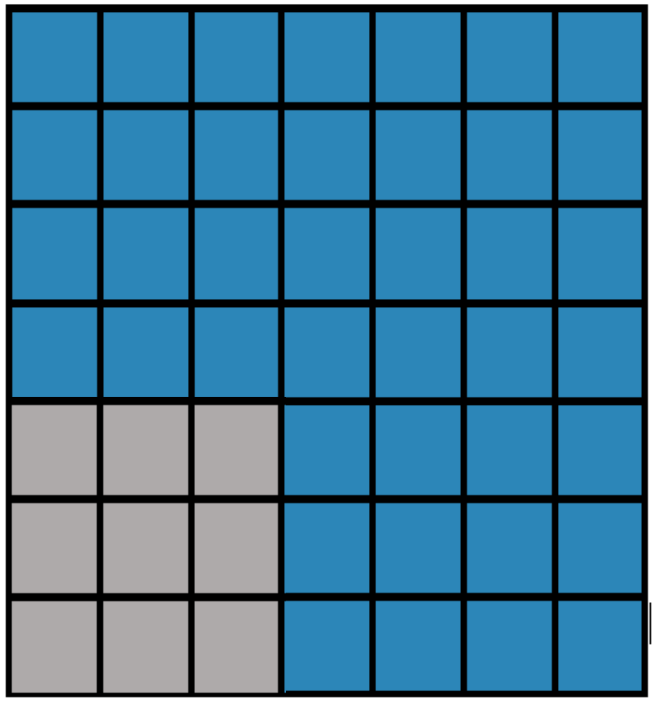
7x7 input



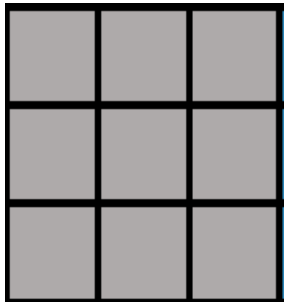
3x3 filter

# Convolutional filter: stride

**Stride = 2**



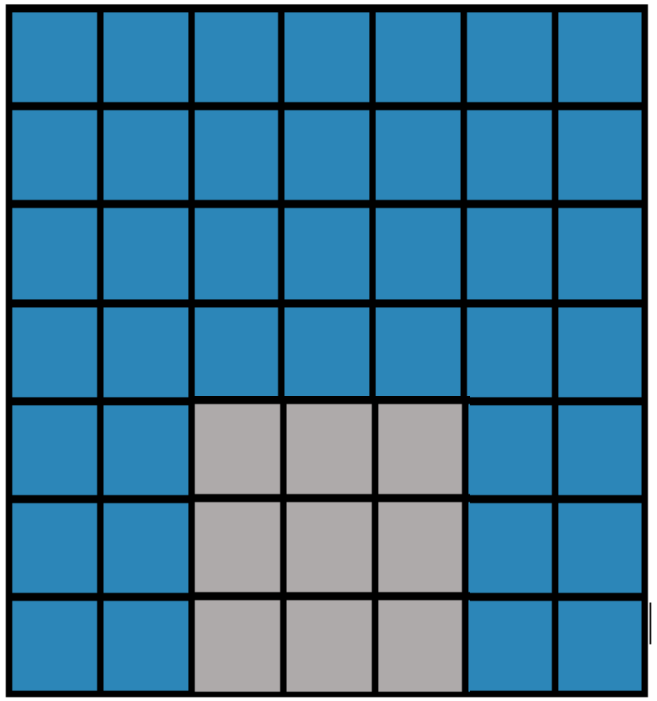
7x7 input



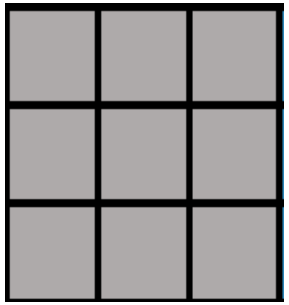
3x3 filter

# Convolutional filter: stride

**Stride = 2**



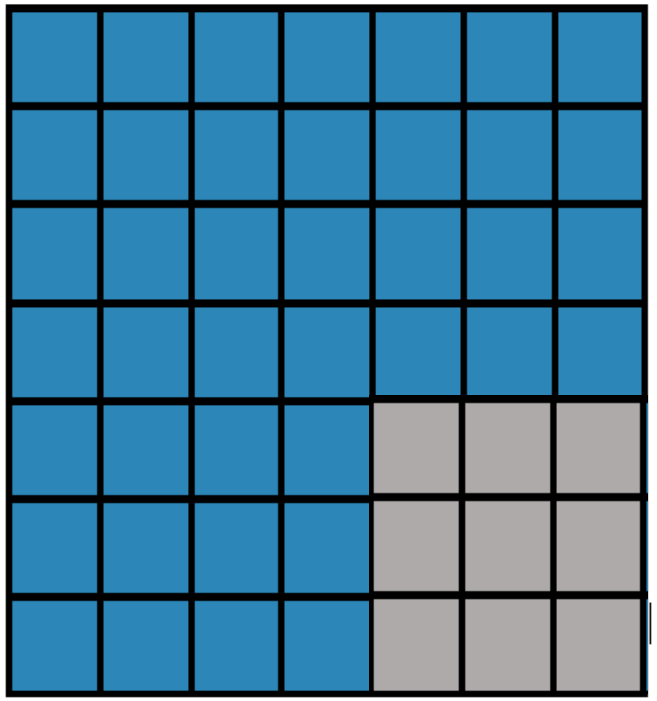
7x7 input



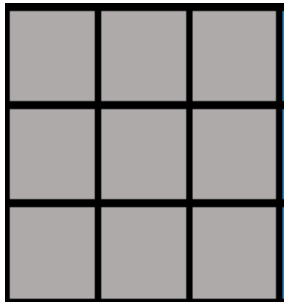
3x3 filter

# Convolutional filter: stride

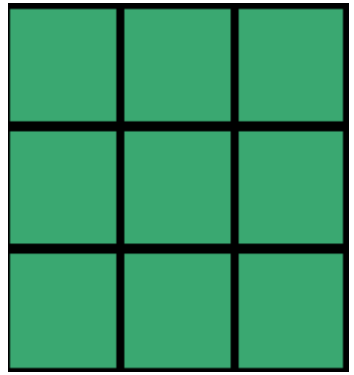
**Stride = 2**



7x7 input

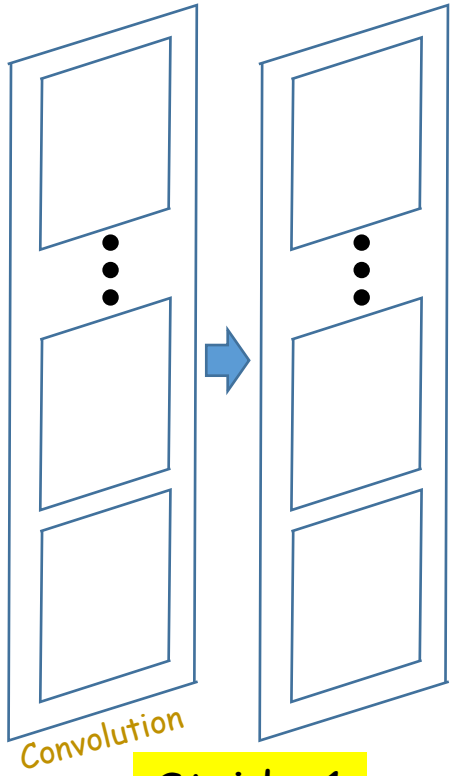


3x3 filter

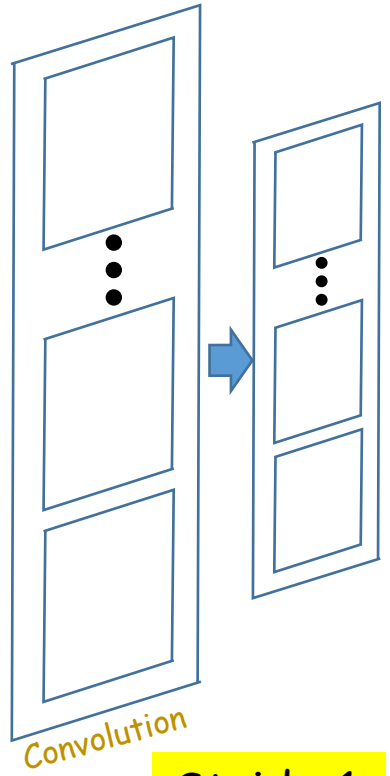


3x3 output

# Convolution strides




Stride=1

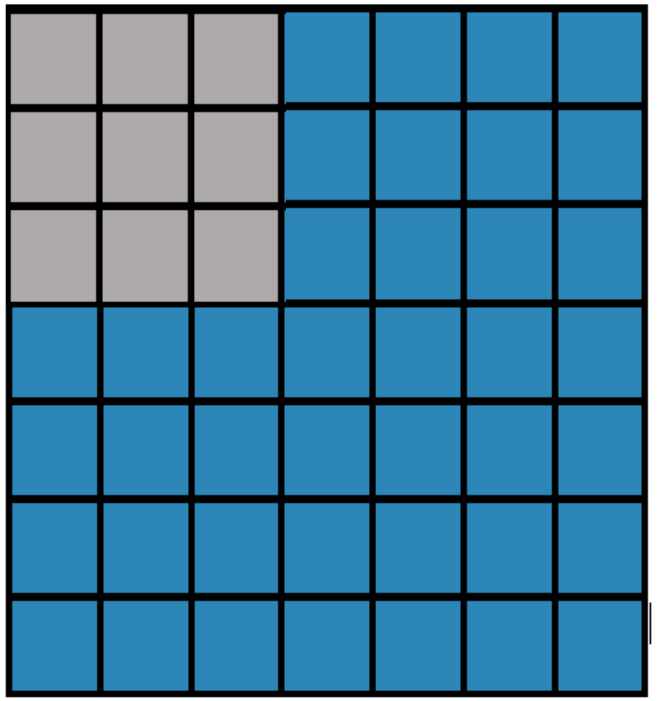



Stride>1

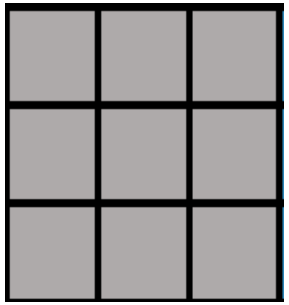
- Convolution with stride 1 → output size same as input size
  - Besides edge effects
- Stride greater than 1 → output size shrinks w.r.t. input

# Convolutional filter: stride

**Stride = 3**



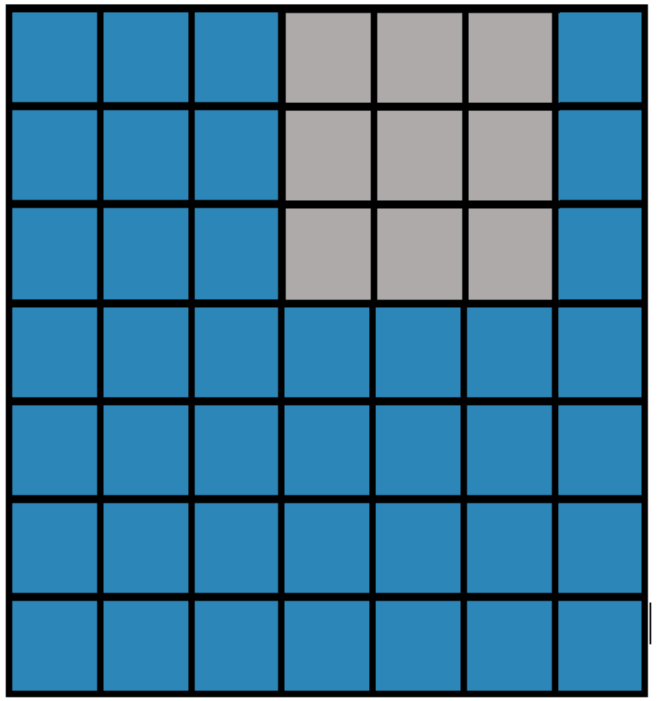
7x7 input



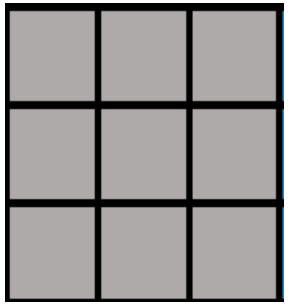
3x3 filter

# Convolutional filter: stride

**Stride = 3**



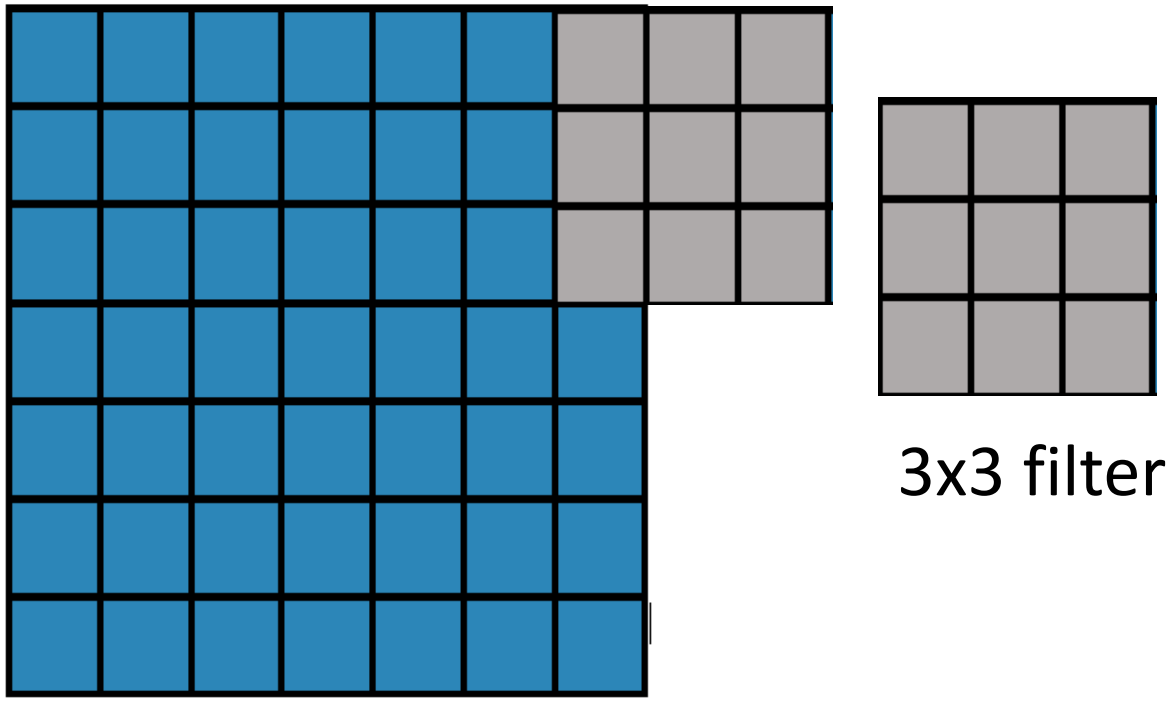
7x7 input



3x3 filter

# Convolutional filter: stride

**Stride = 3**

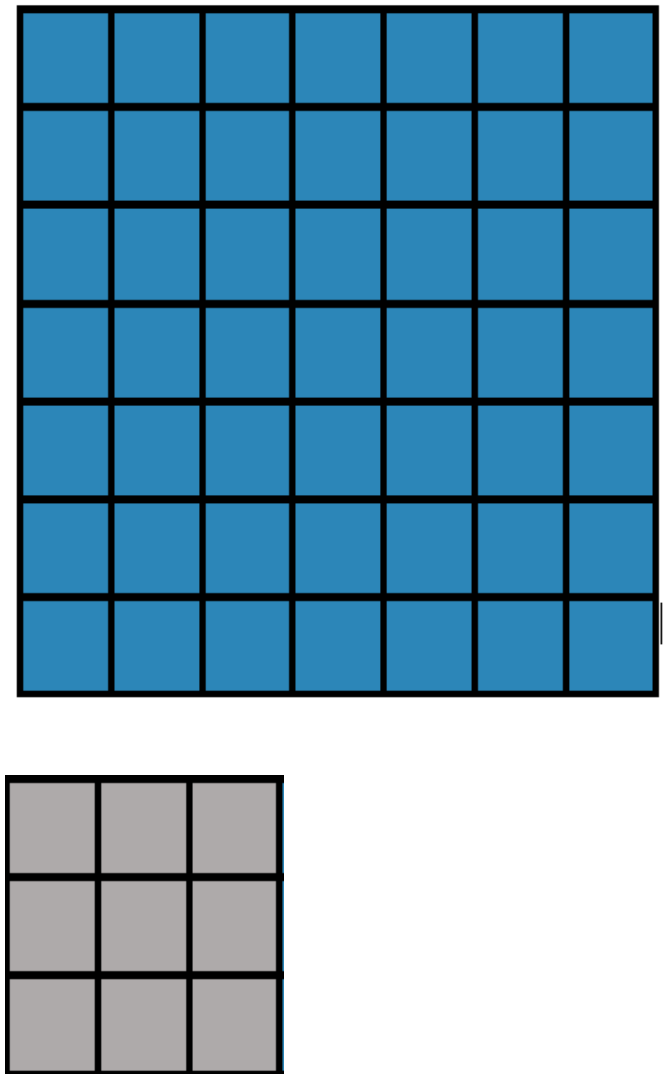


3x3 filter

7x7 input

cannot apply 3x3 filter on 7x7 input with stride 3.

# Output size




Output size:

**(N - F) / stride + 1**

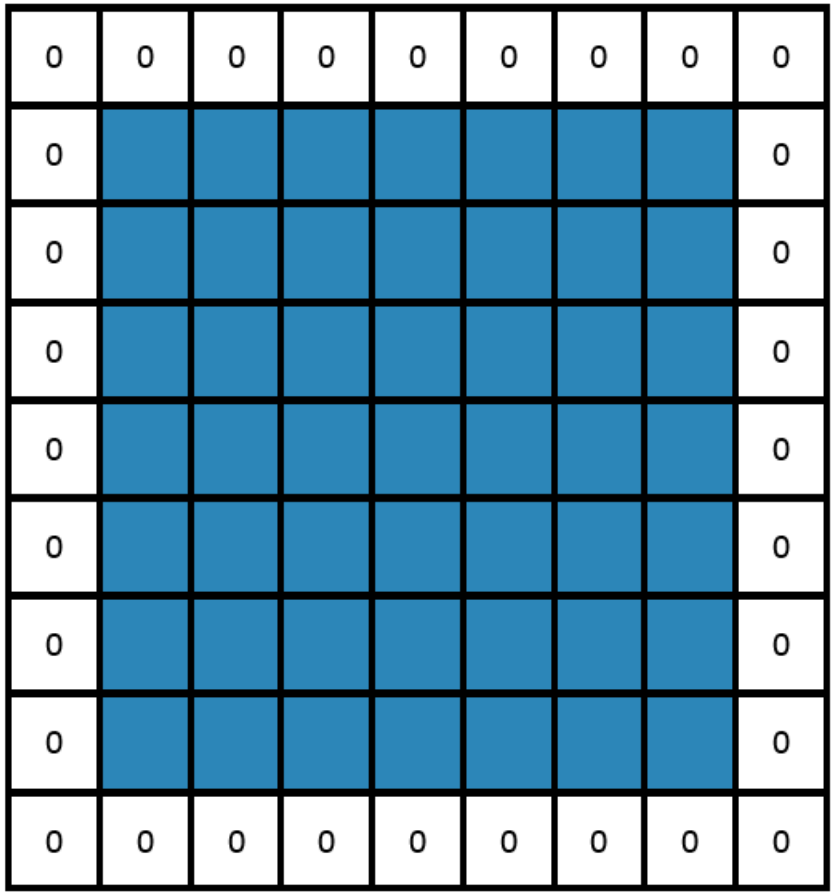
Example:

N = 7, F = 3: stride 1 => (7 - 3)/1 + 1 = 5

stride 2 => (7 - 3)/2 + 1 = 3

stride 3 => (7 - 3)/3 + 1 = 2.33 :\

# In practice: Common to zero pad the border

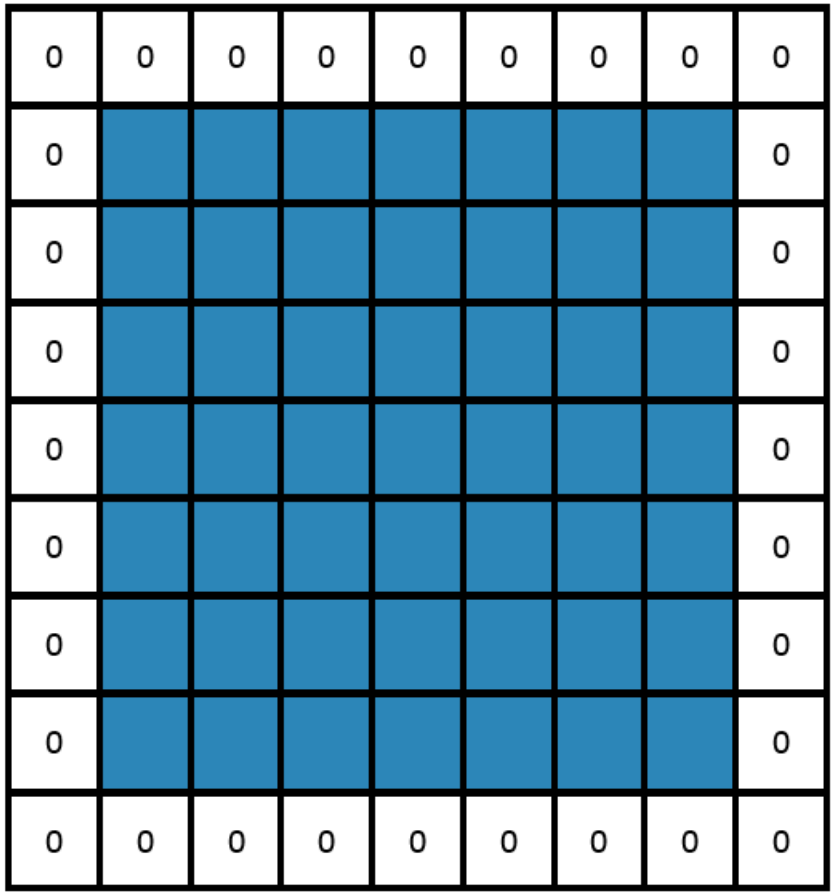



input 7x7
Filter 3x3
stride 1
zero pad with 1 pixel border

=> output= 7x7

Output size:
**(N+2P - F) / stride + 1**

# In practice: Common to zero pad the border




**Common in practice:**

filters FxF

stride 1

zero-padding with (F-1)/2

=> will preserve size

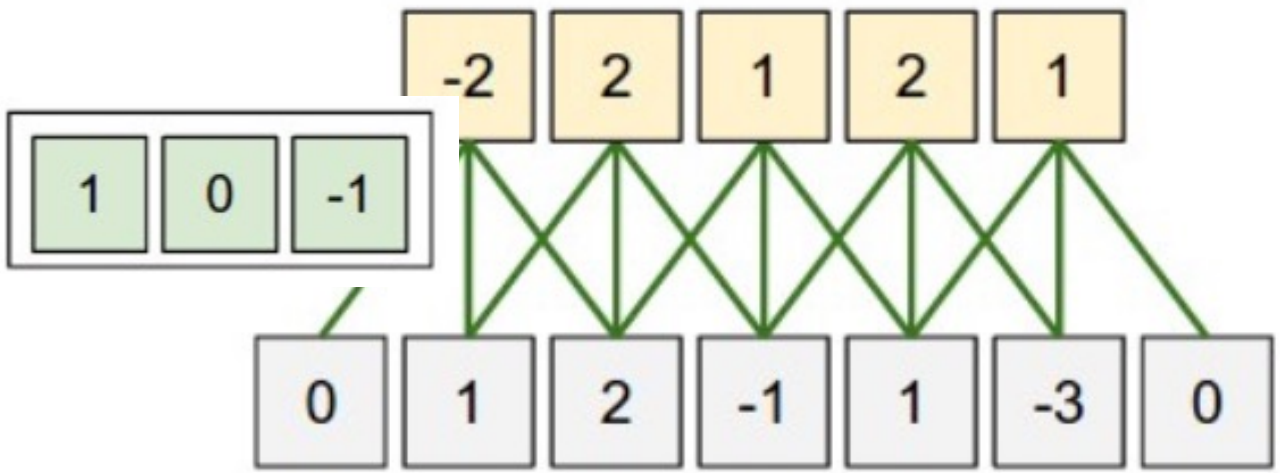
e.g. F = 3 => zero pad with 1

F = 5 => zero pad with 2

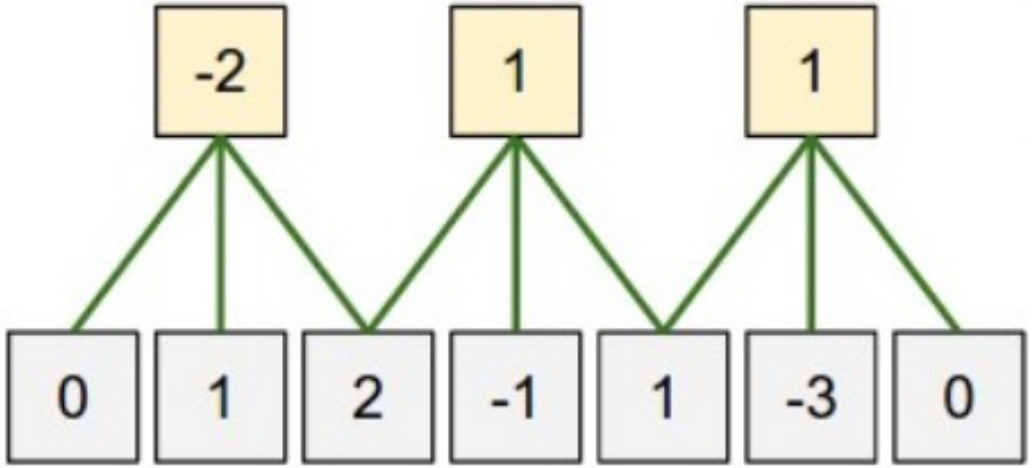
F = 7 => zero pad with 3

zero padding allows us to control the spatial size of the output volumes

# 1D example




N = 5
F = 3
P = 1
S = 1
Output = (5 - 3 + 2)/1+1 = 5.




N = 5
F = 3
P = 1
S = 2
Output = (5 - 3 + 2)/2+1 = 3.

# Zero padding

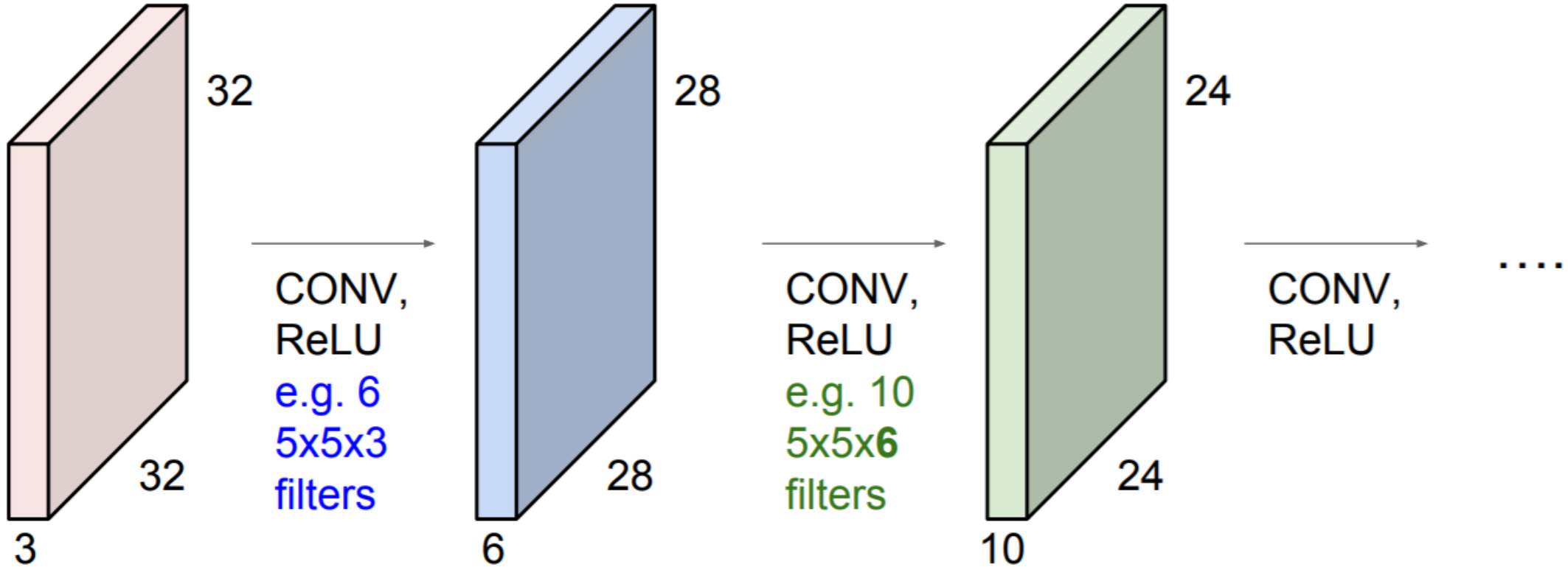
- Not only for stride > 1
- Also to prevent reduction in the output size even if $S = 1$
  - Sometimes not considered acceptable
    - We may want the output map ideally to be the same size as the input
- For hop size $S > 1$, zero padding is adjusted to ensure that the size of the convolved output is $\lceil N/S \rceil$
  - Achieved by *first* zero padding the image with $S\lceil N/S \rceil - N$ columns/rows of zeros and then applying above rules

# Zero padding

- For an $F$ width filter:
  - Odd $F$ : Pad on both left and right with $(F-1)/2$ columns of zeros
  - Even $F$ : Pad one side with $F/2$ columns of zeros, and the other with $\frac{F}{2}-1$ columns of zeros
  - The resulting image is width $N+F-1$
  - The result of the convolution (with S=1) is width $N$
- The top/bottom zero padding follows the same rules to maintain map height after convolution

# We want to maintain the input size

- (32 -> 28 -> 24 ...).
- Shrinking too fast is not good, it may not work well.



# Example

- Input: 32x32x3
- Filters: 10  5x5x3 filters
- Stride: 1
- Pad: 2

- Output size: ?

# Example

- Input: 32x32x3
- Filters: 10  5x5x3 filters
- Stride: 1
- Pad: 2

- Output size: 32x32x10

# Example

- Input: 32x32x3
- Filters: 10  5x5x3 filters
- Stride: 1
- Pad: 2

- Number of parameters in this layer?

# Example

- Input: 32x32x3
- Filters: 10  5x5x3 filters
- Stride: 1
- Pad: 2

- Number of parameters in this layer?
  - each filter has 5*5*3 + 1 = 76 params (+1 for bias)
  - => 76*10 = 760

# Parameter Setting

- Accepts a volume of size $W_1 \times H_1 \times D_1$

- Requires four hyper-parameters:
  - Number of filters $K$
  - Their spatial extent $F$
  - The stride $S$
  - The amount of zero padding $P$

- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = \frac{(W_1 - F + 2P)}{S} + 1$
  - $H_2 = \frac{(H_1 - F + 2P)}{S} + 1$
  - $D_2 = K$

Common settings:

K = powers of 2 (e.g. 32, 64, …)

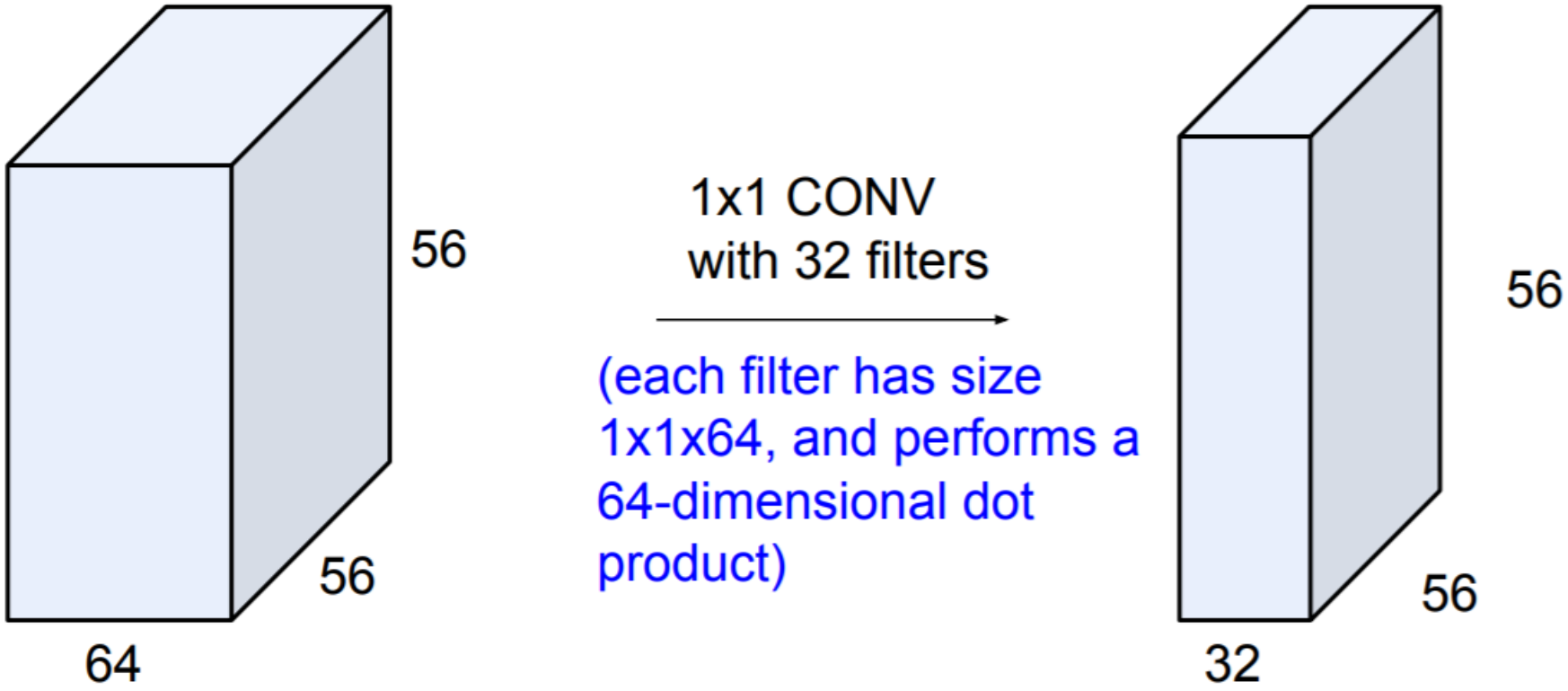
F = 3, S = 1, P=1

F = 5, S = 1, P=2

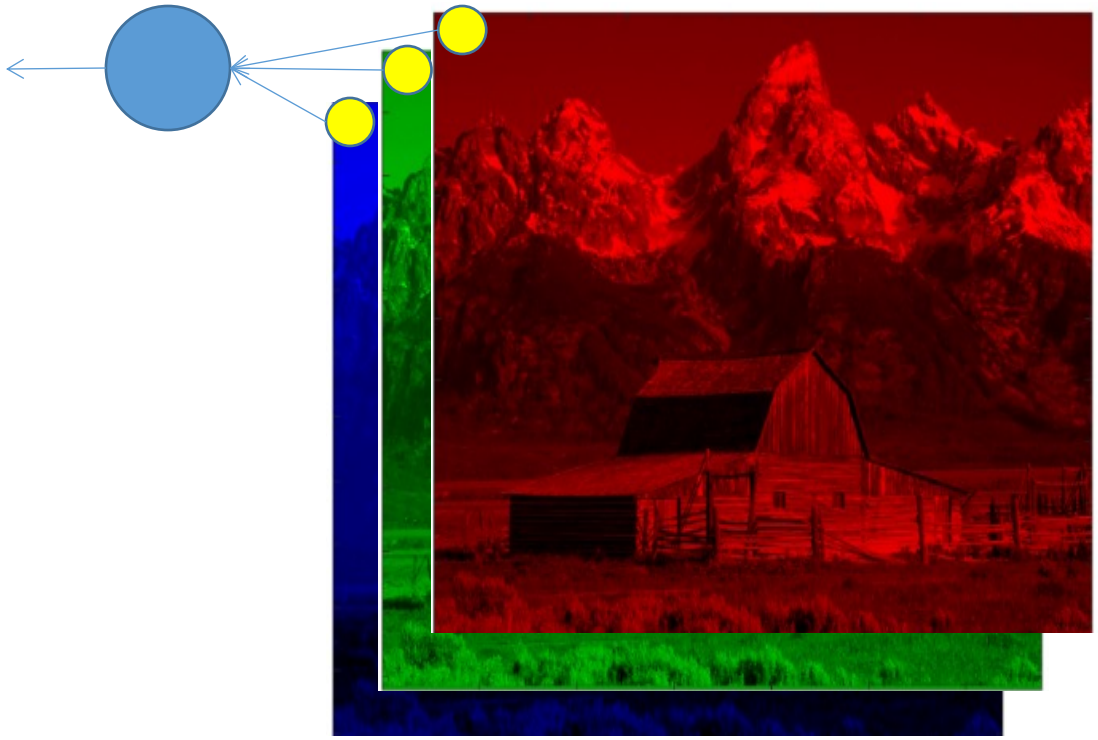
F = 5, S = 2, P = ? (whatever fits)

F = 1, S = 1, P = 0

# The 1x1 filter



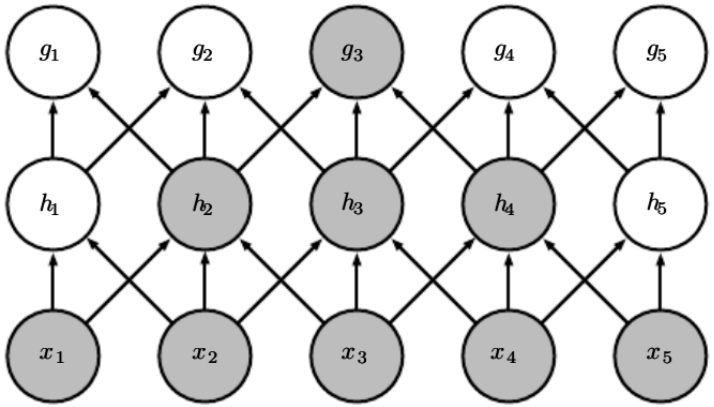
# The 1x1 filter



- A 1x1 filter is simply a neuron 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

# Receptive Field

How big of a region in the input does a neuron on the second conv-layer see?



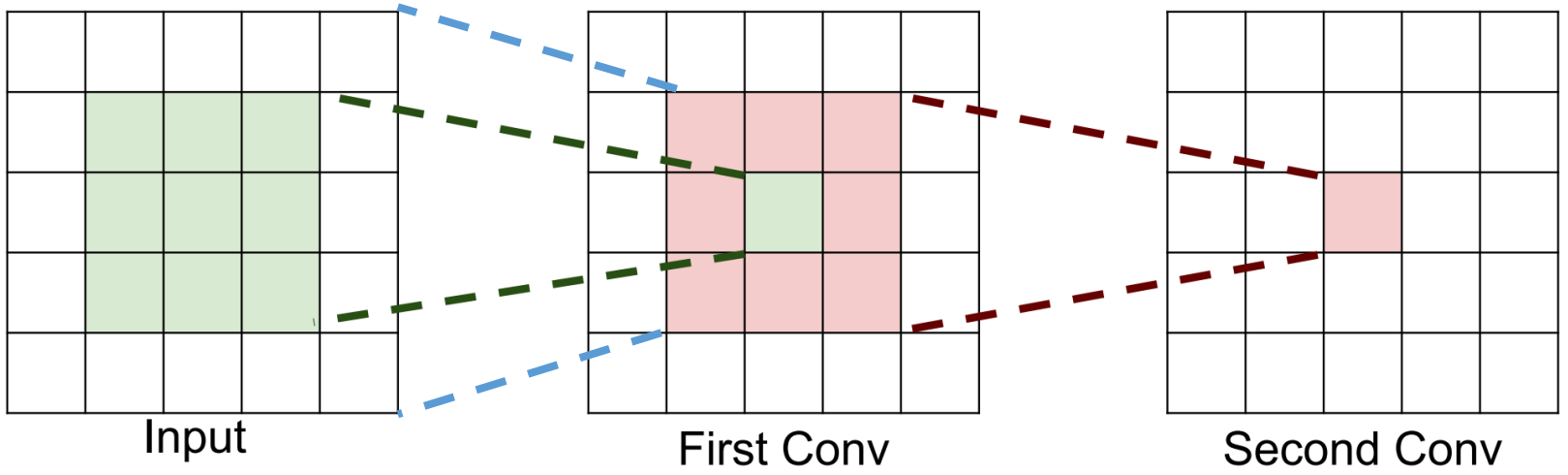

units in the deeper layers can be indirectly connected to all or most of the input image.

[Goodfellow et al. 2016]

# Receptive Field

How big of a region in the input does a neuron on the second conv-layer see?

Two 3x3 filters together perform like one 5x5 filter (same receptive field)



# Power of Small Filters

Suppose input is H x W x C and we use convolutions with C filters to preserve depth (stride 1, padding to preserve H, W)

one CONV with 7 x 7 filters

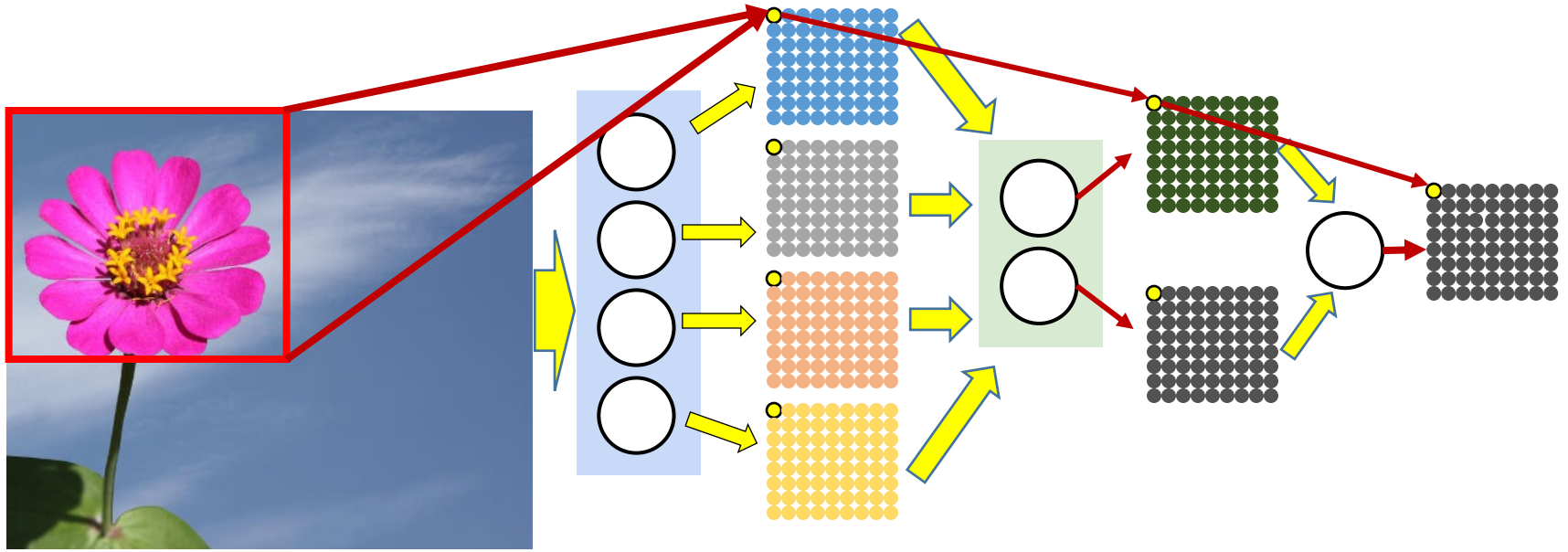
Number of weights:
= C x (7 x 7 x C) = **49 $C^2$**

three CONV with 3 x 3 filters

Number of weights:
= 3 x C x (3 x 3 x C) = **27 $C^2$**

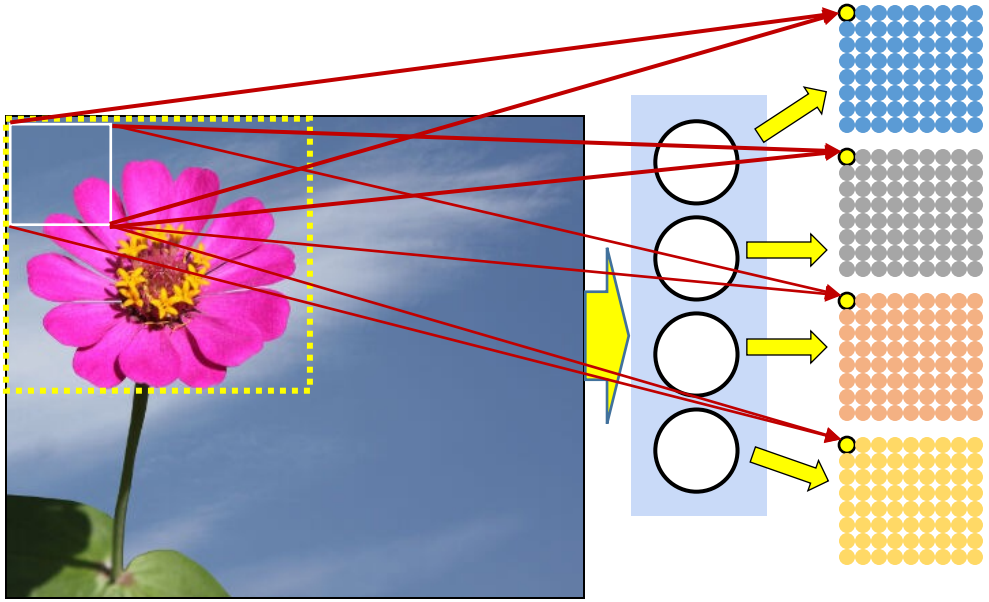
Fewer parameters, more nonlinearity = GOOD

# The behavior of the layers



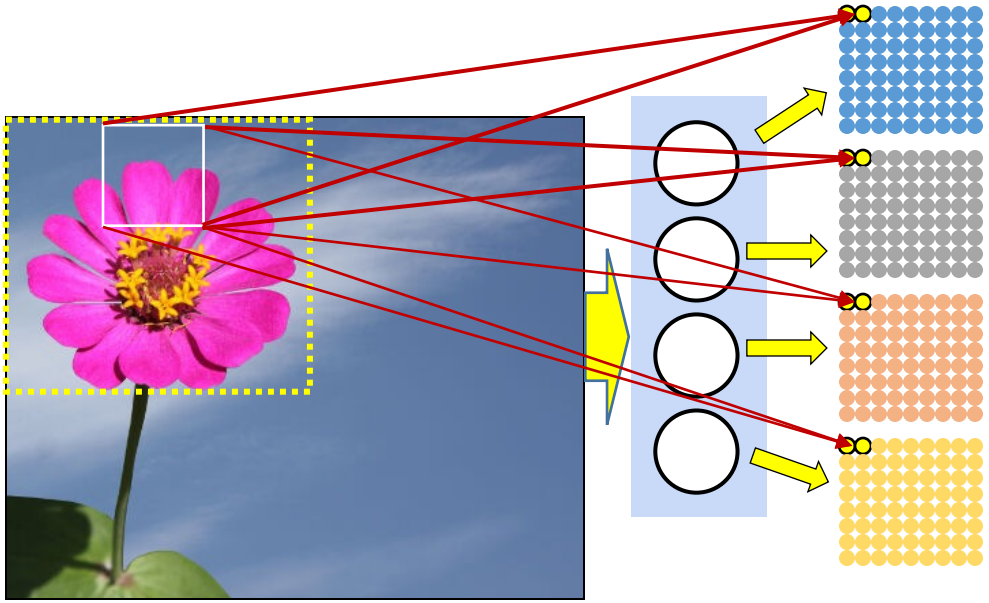
- The first layer neurons "look" at the entire "block" to extract block-level features
  - Subsequent layers only perform classification over these block-level features
- **The first layer neurons is responsible for evaluating the entire *block* of pixels**
  - **Subsequent layers only look at a *single* pixel in their input maps**

# Distributing the scan



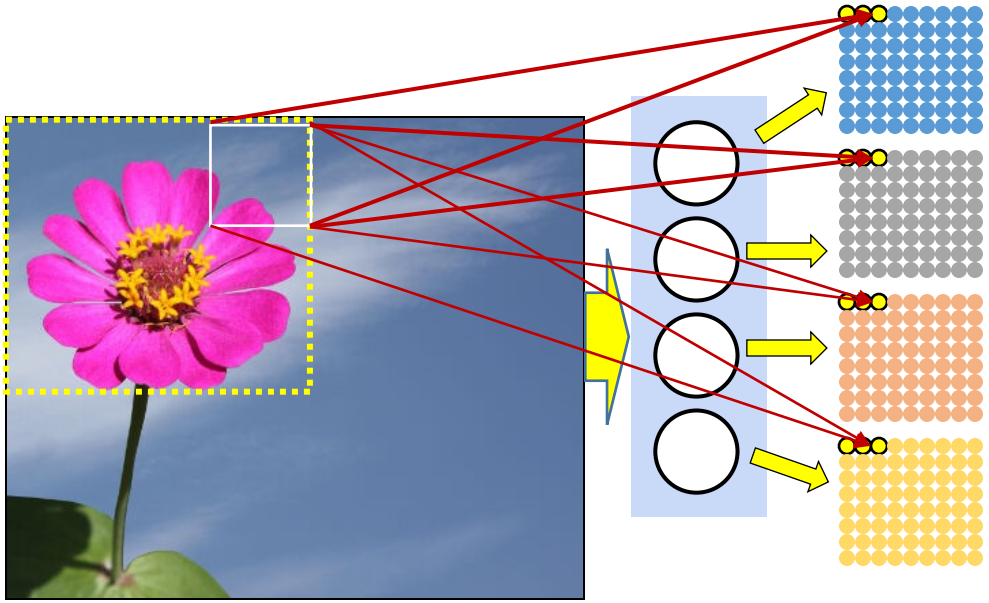
- We can distribute the pattern matching over two layers and still achieve the same block analysis at the second layer
  - The first layer evaluates smaller blocks of pixels

# Distributing the scan



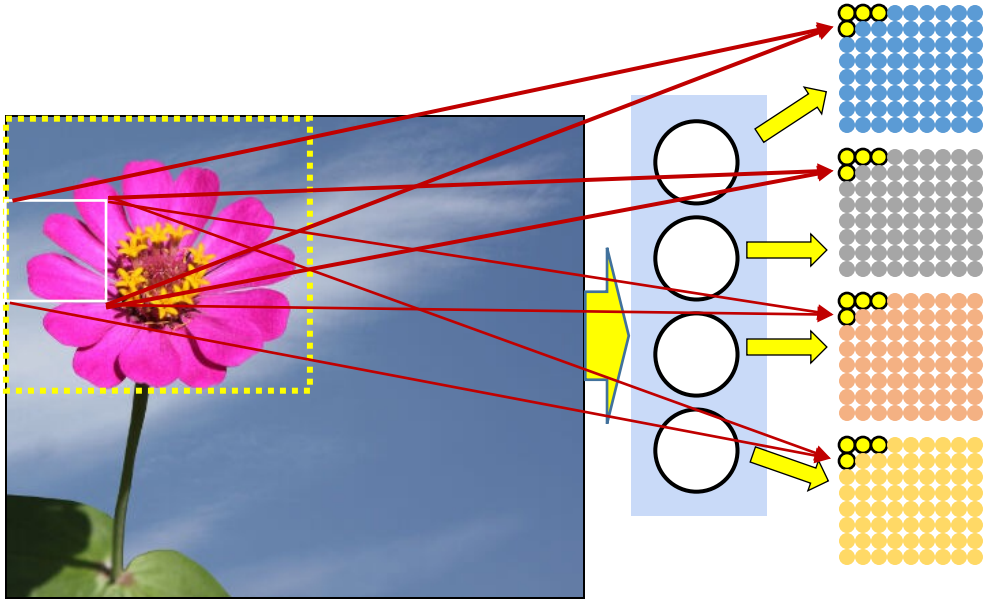
- We can distribute the pattern matching over two layers and still achieve the same block analysis at the second layer
  - The first layer evaluates smaller blocks of pixels

# Distributing the scan



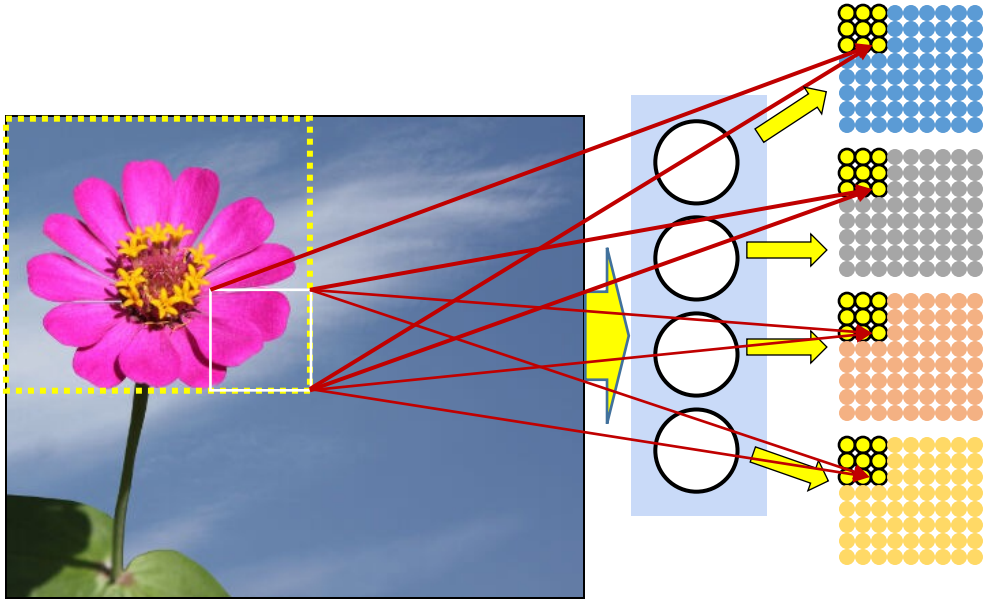
- We can distribute the pattern matching over two layers and still achieve the same block analysis at the second layer
  - The first layer evaluates smaller blocks of pixels

# Distributing the scan



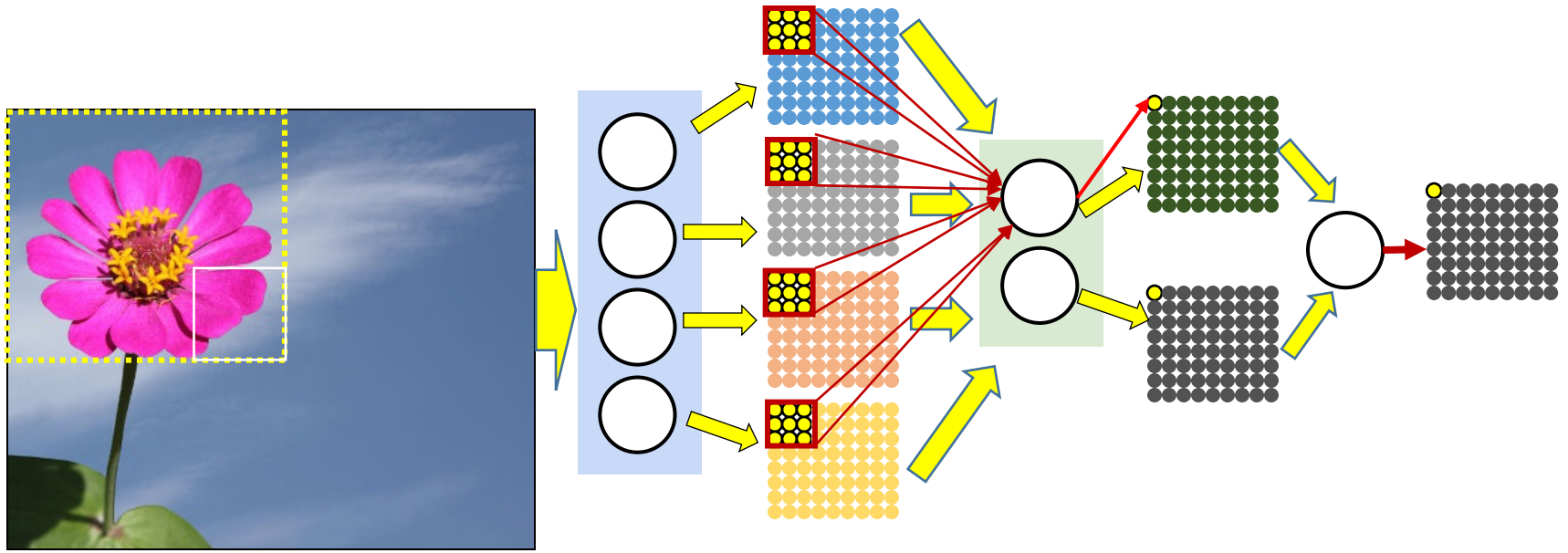
- We can distribute the pattern matching over two layers and still achieve the same block analysis at the second layer
  - The first layer evaluates smaller blocks of pixels

# Distributing the scan



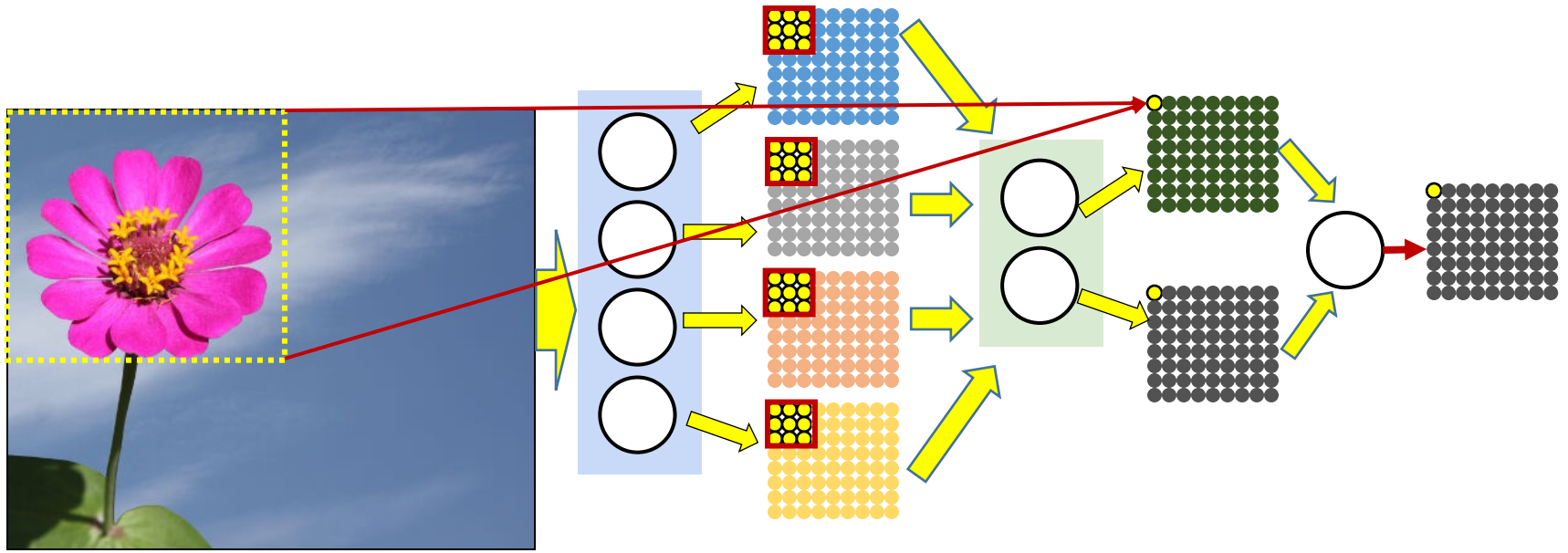
- We can distribute the pattern matching over two layers and still achieve the same block analysis at the second layer
  - The first layer evaluates smaller blocks of pixels

# Distributing the scan



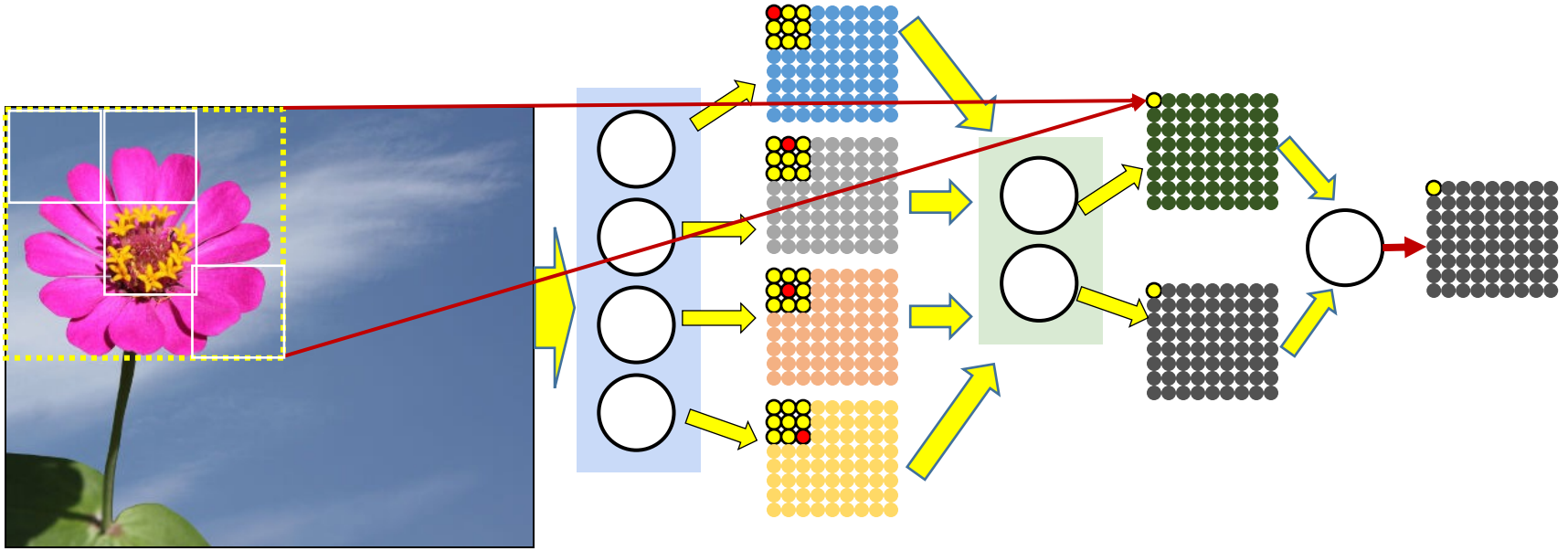
- We can distribute the pattern matching over two layers and still achieve the same block analysis at the second layer
  - The first layer evaluates smaller blocks of pixels
  - The next layer evaluates blocks of outputs from the first layer

# Distributing the scan



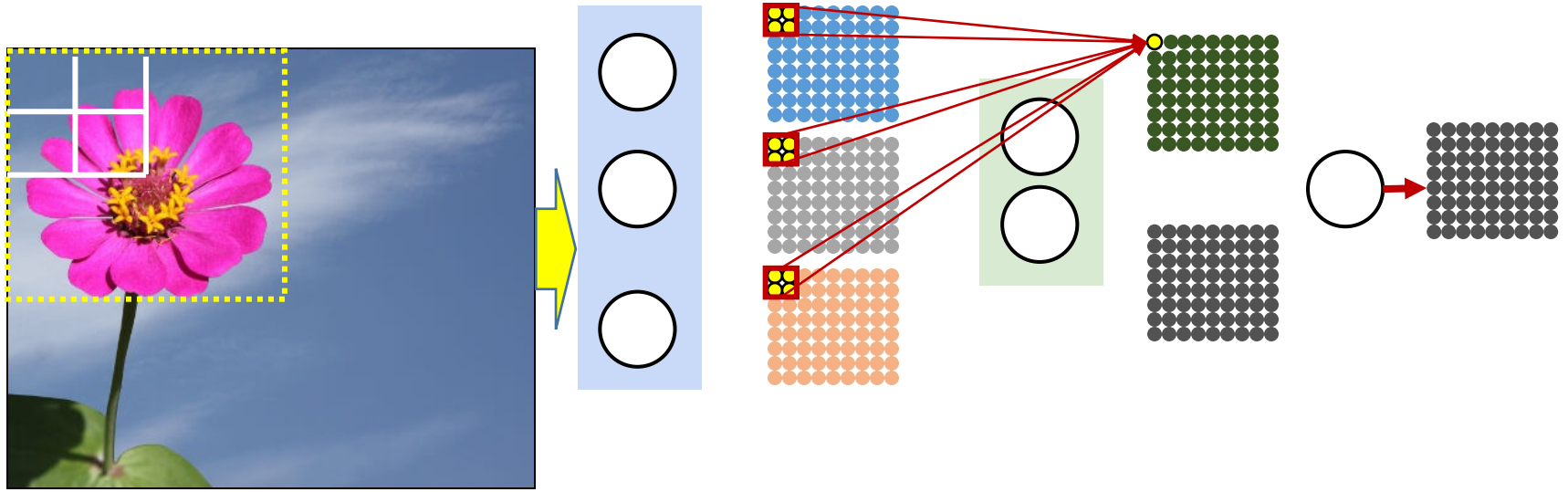
- We can distribute the pattern matching over two layers and still achieve the same block analysis at the second layer
  - The first layer evaluates smaller blocks of pixels
  - The next layer evaluates blocks of outputs from the first layer
  - This effectively evaluates the larger block of the original image

# Distributing the scan



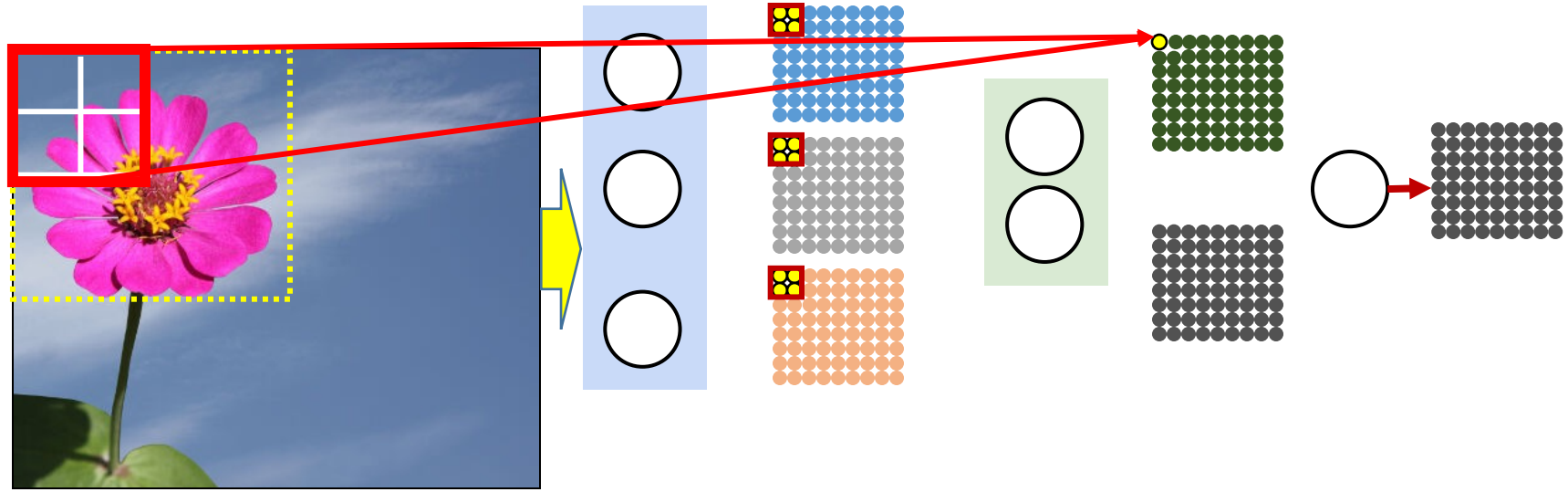
- The higher layer implicitly learns the *arrangement* of sub patterns that represents the larger pattern (the flower in this case)

# This logic can be recursed



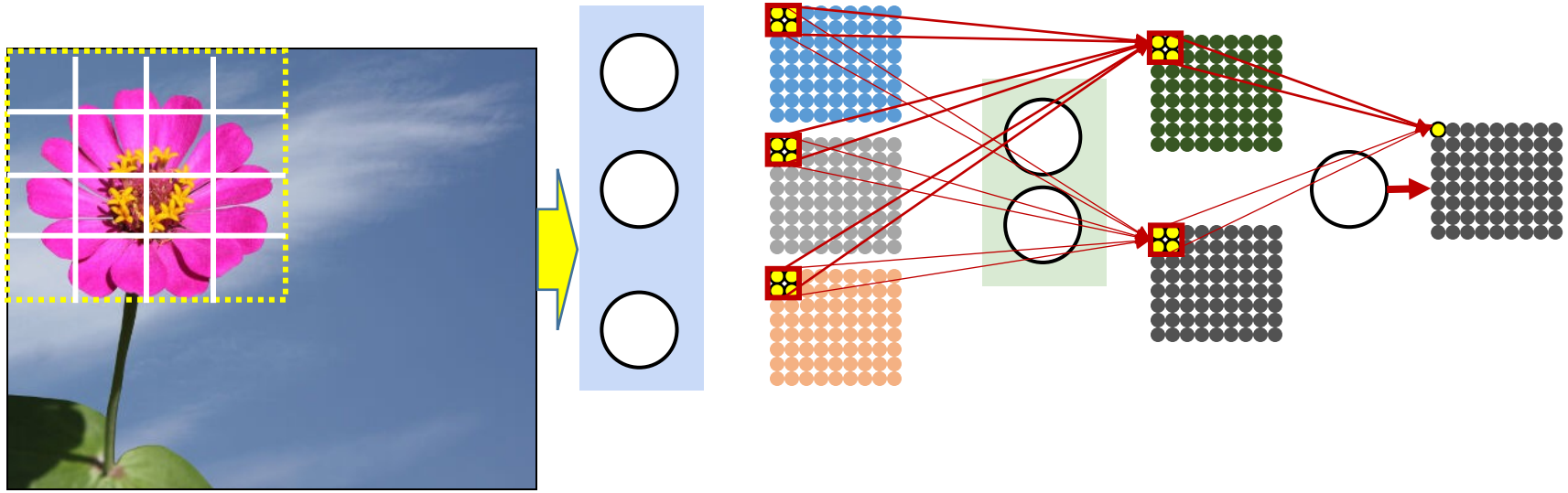
- Building the pattern over 3 layers

# This logic can be recursed



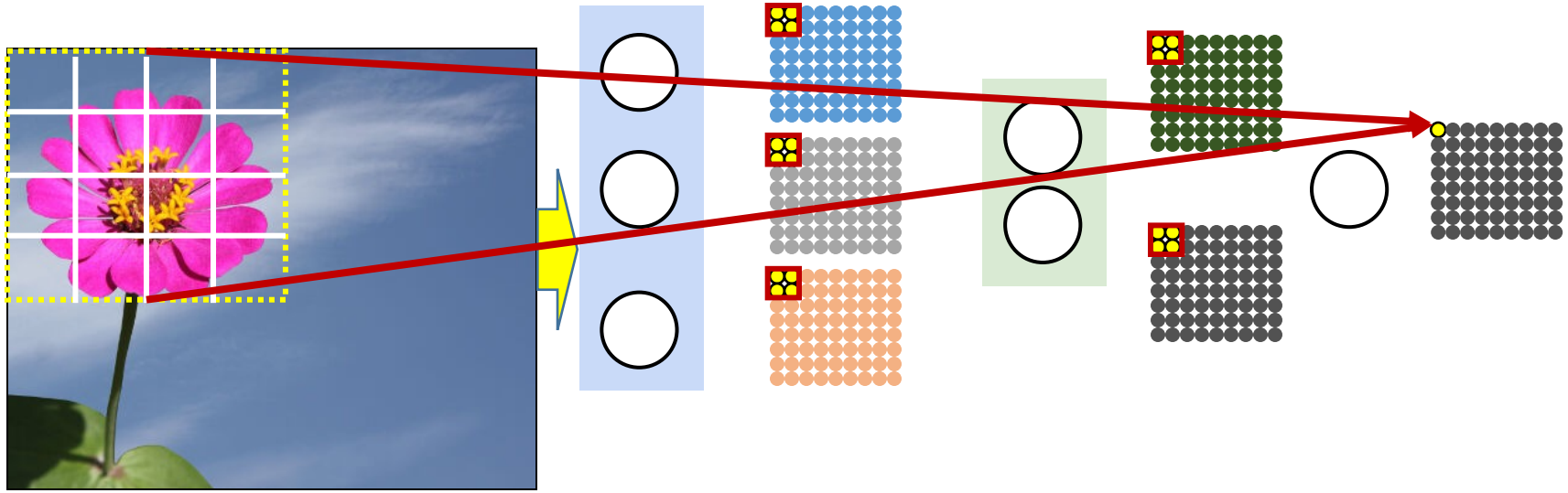
- Building the pattern over 3 layers

# This logic can be recursed



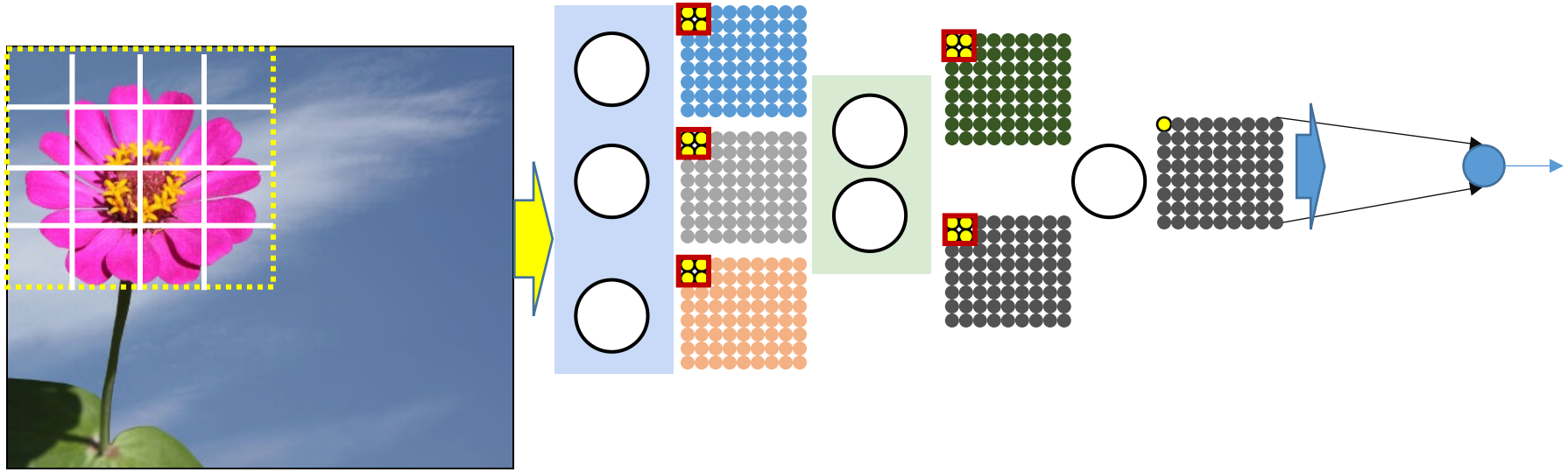
- Building the pattern over 3 layers

# This logic can be recursed



- Building the pattern over 3 layers

# Does the picture have a flower



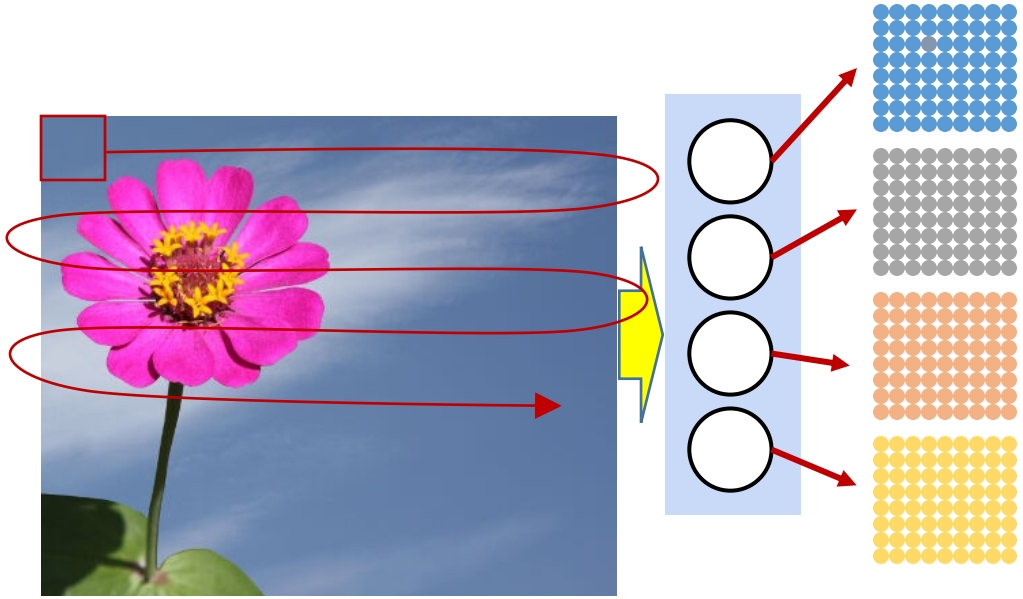
- Building the pattern over 3 layers
- The final classification for the entire image views the outputs from all locations, as seen in the final map

# Why distribute?

- Distribution forces *localized* patterns in lower layers
  - More generalizable

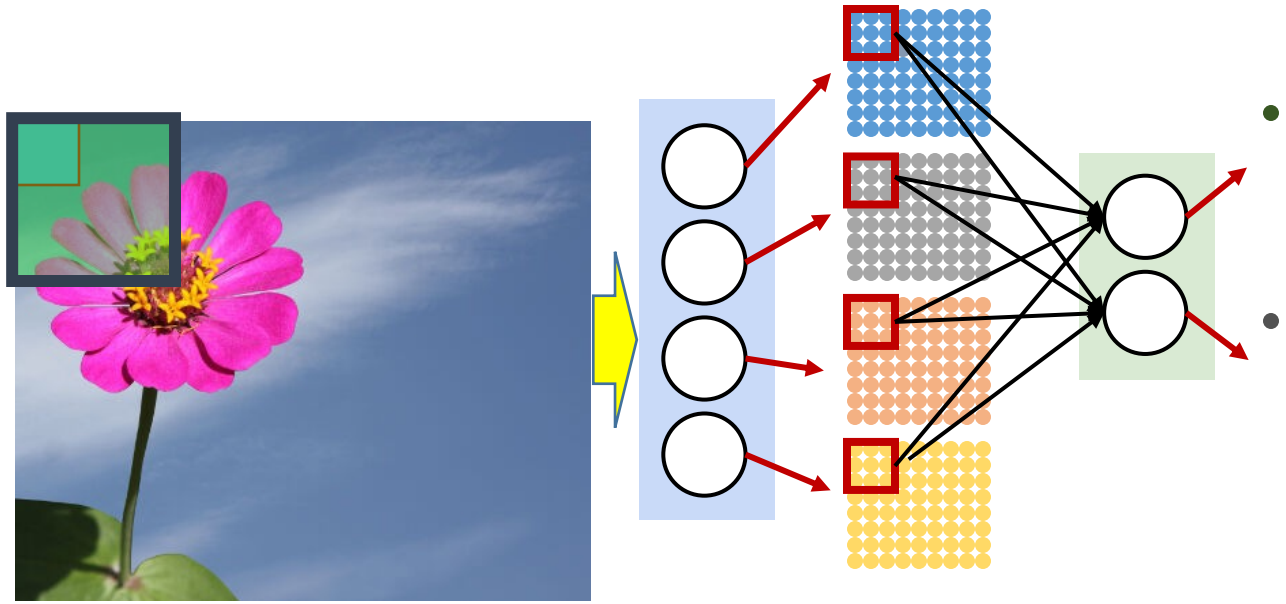
- Number of parameters…
  - Large (sometimes order of magnitude) reduction in parameters
    - Gains increase as we increase the depth over which the blocks are distributed

# Building up patterns



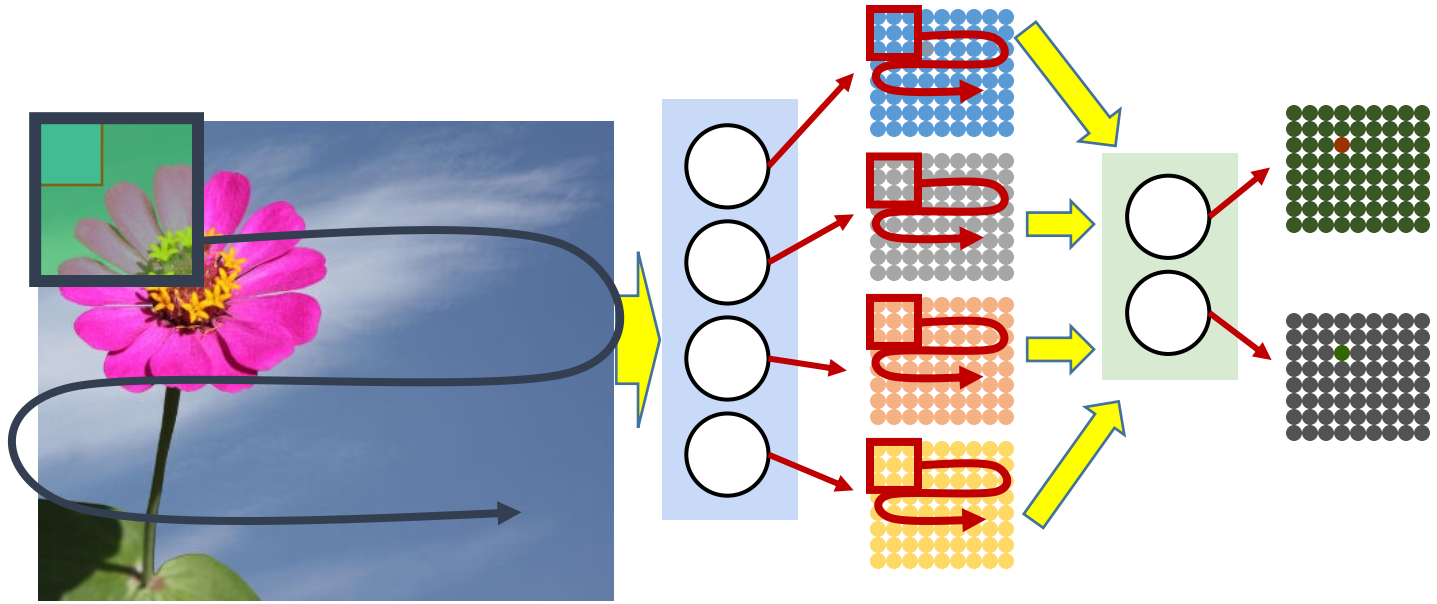
- The first layer looks at small *sub* regions of the main image
  - Sufficient to detect, say, petals

# Building up patterns



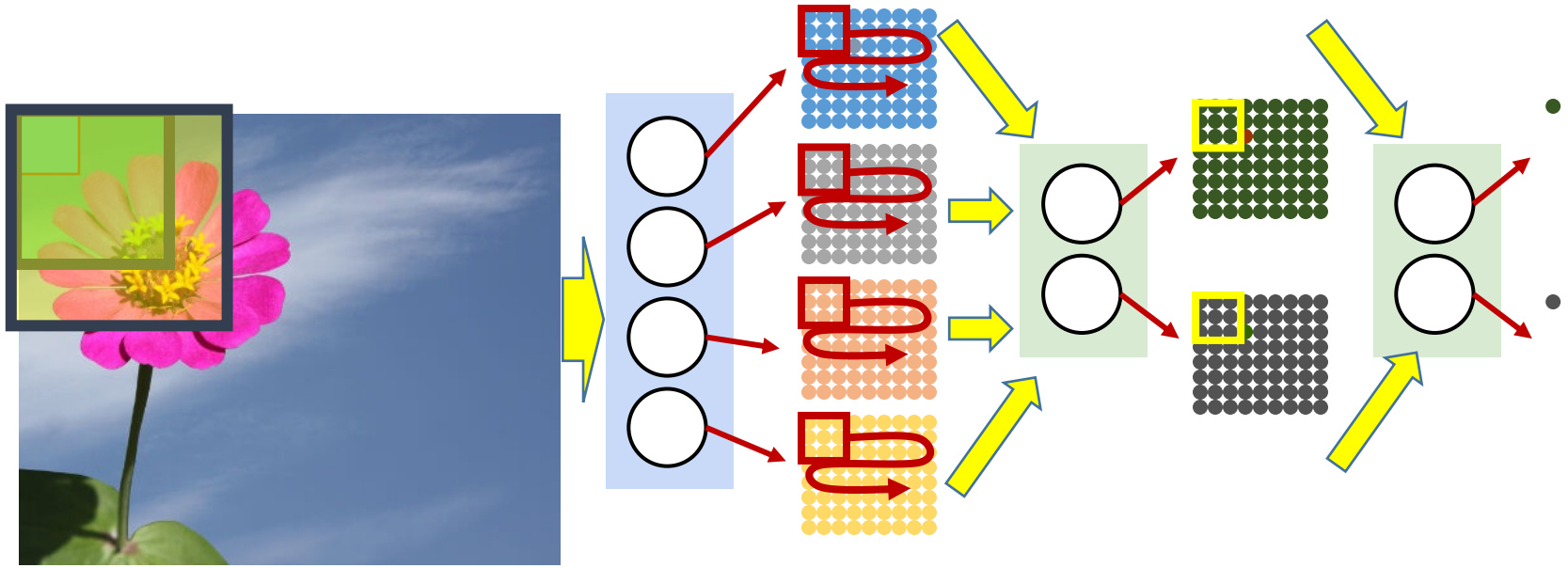
- The first layer looks at *sub* regions of the main image
  - Sufficient to detect, say, petals
- The second layer looks at *regions* of the output of the first layer
  - To put the petals together into a flower
  - This corresponds to looking at a larger region of the original input image

# Building up patterns



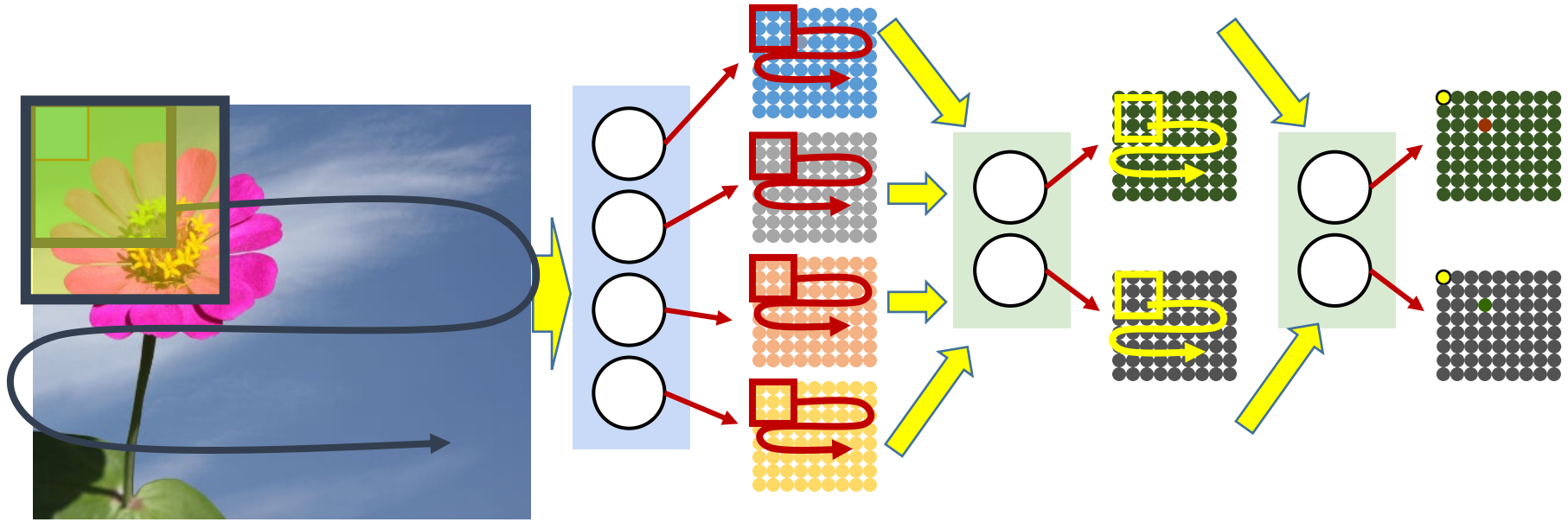
- The first layer looks at *sub* regions of the main image
  - Sufficient to detect, say, petals
- The second layer looks at *regions* of the output of the first layer
  - To put the petals together into a flower
  - This corresponds to looking at a larger region of the original input image

# Building up patterns



- The first layer looks at *sub* regions of the main image
  - Sufficient to detect, say, petals
- The second layer looks at *regions* of the output of the first layer
  - To put the petals together into a flower
  - This corresponds to looking at a larger region of the original input image
- We may have any number of layers in this fashion

# Building up patterns

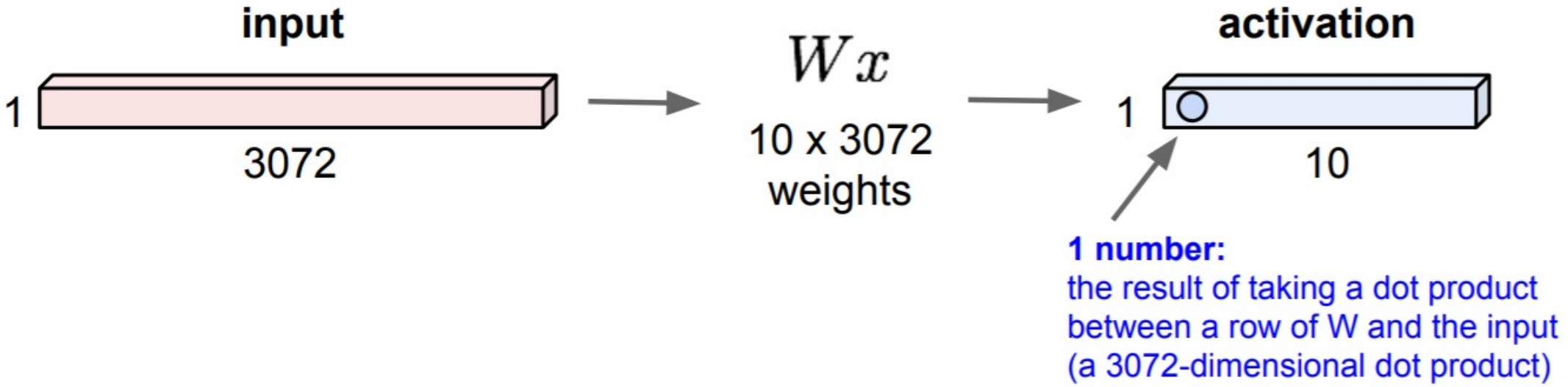



- The first layer looks at *sub* regions of the main image
  - Sufficient to detect, say, petals
- The second layer looks at *regions* of the output of the first layer
  - To put the petals together into a flower
  - This corresponds to looking at a larger region of the original input image
- We may have any number of layers in this fashion

# Fully connected layer

32x32x3 image -> stretch to 3072 x 1

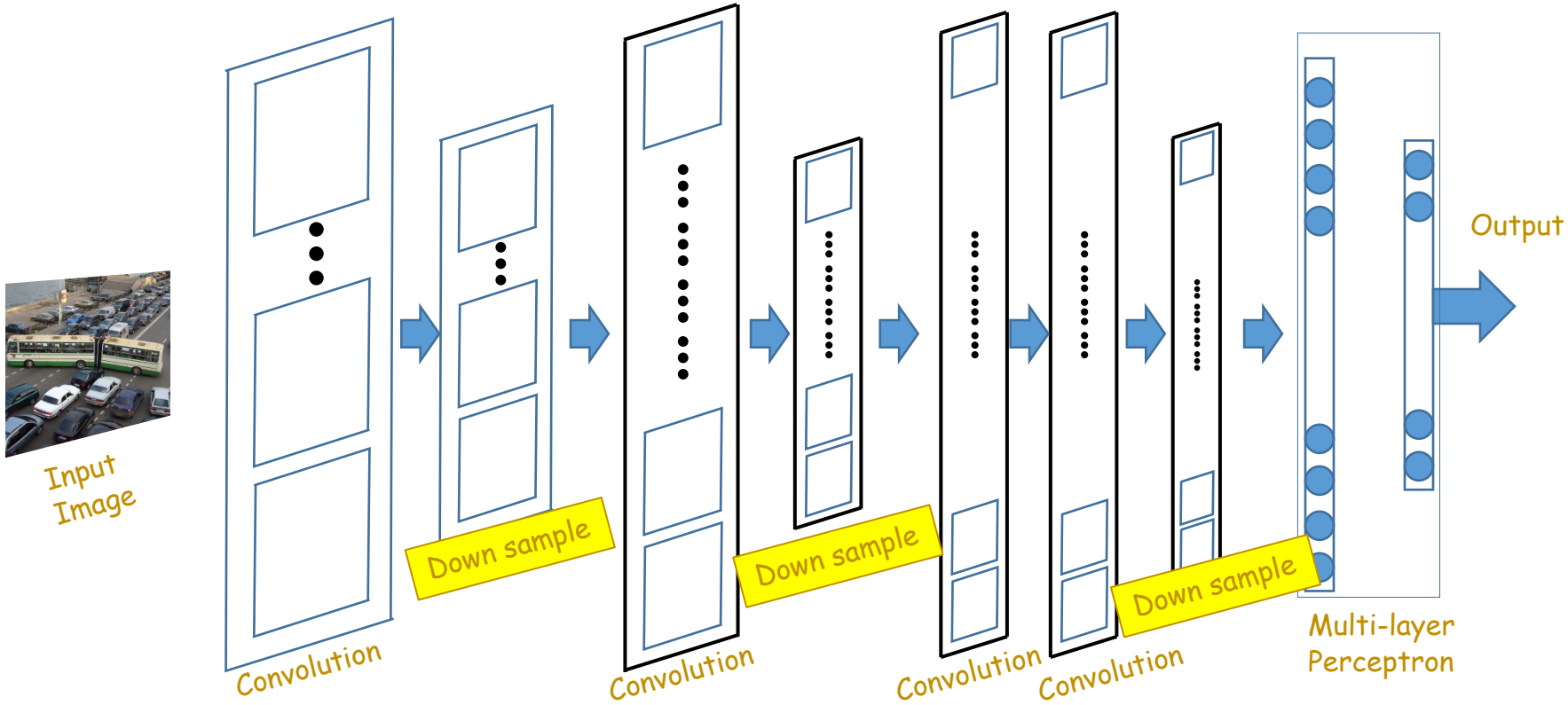
Each neuron looks at the full input volume



# Layers used to build ConvNets

- Three main types of layers
  - **Convolutional Layer**
    - output of neurons are connected to local regions in the input
    - applying the same filter on the whole image
    - CONV layer's parameters consist of a set of learnable filters.
  - **Pooling Layer**
    - perform a downsampling operation along the spatial dimensions
  - **Fully-Connected Layer**

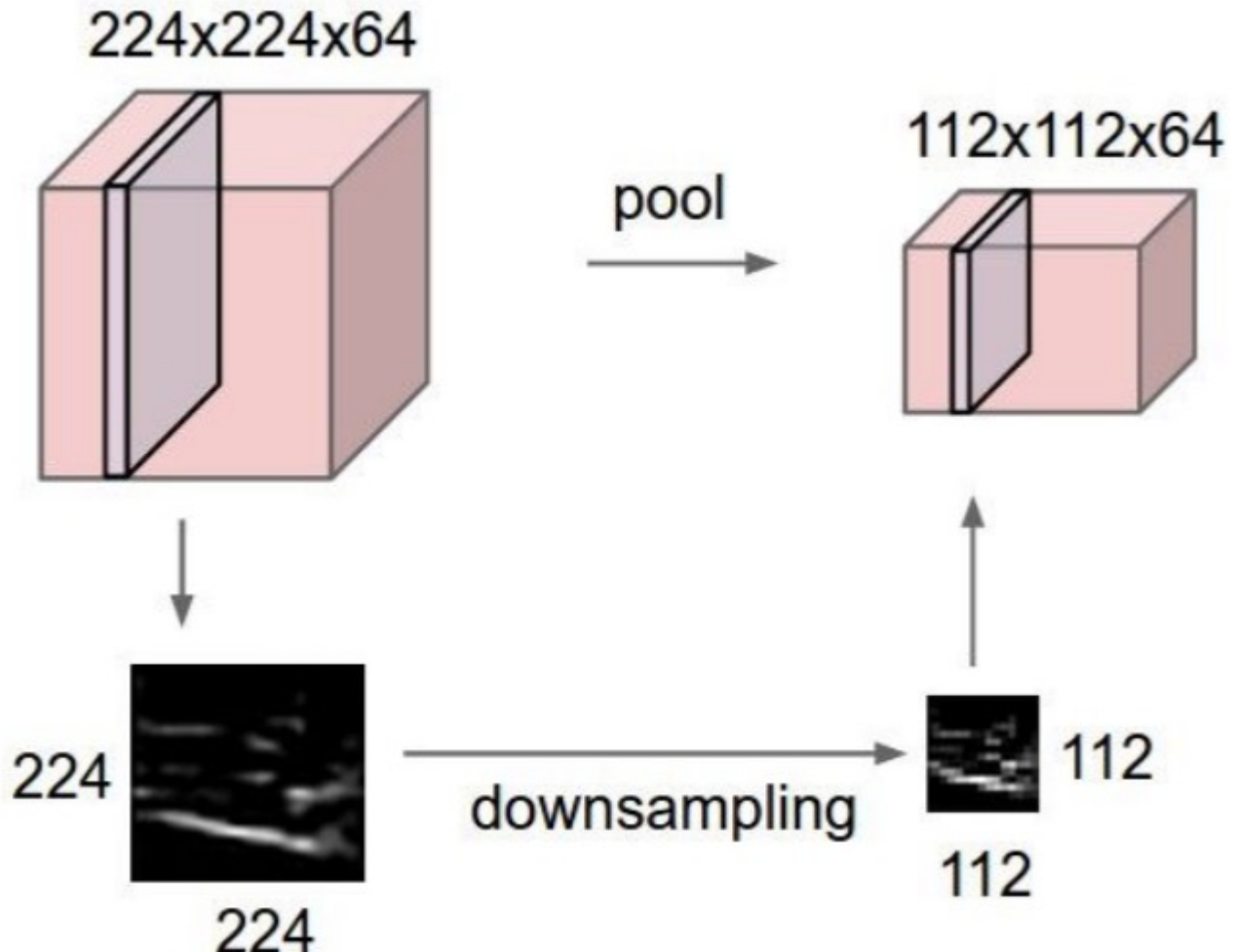
# The other component Downsampling/Pooling



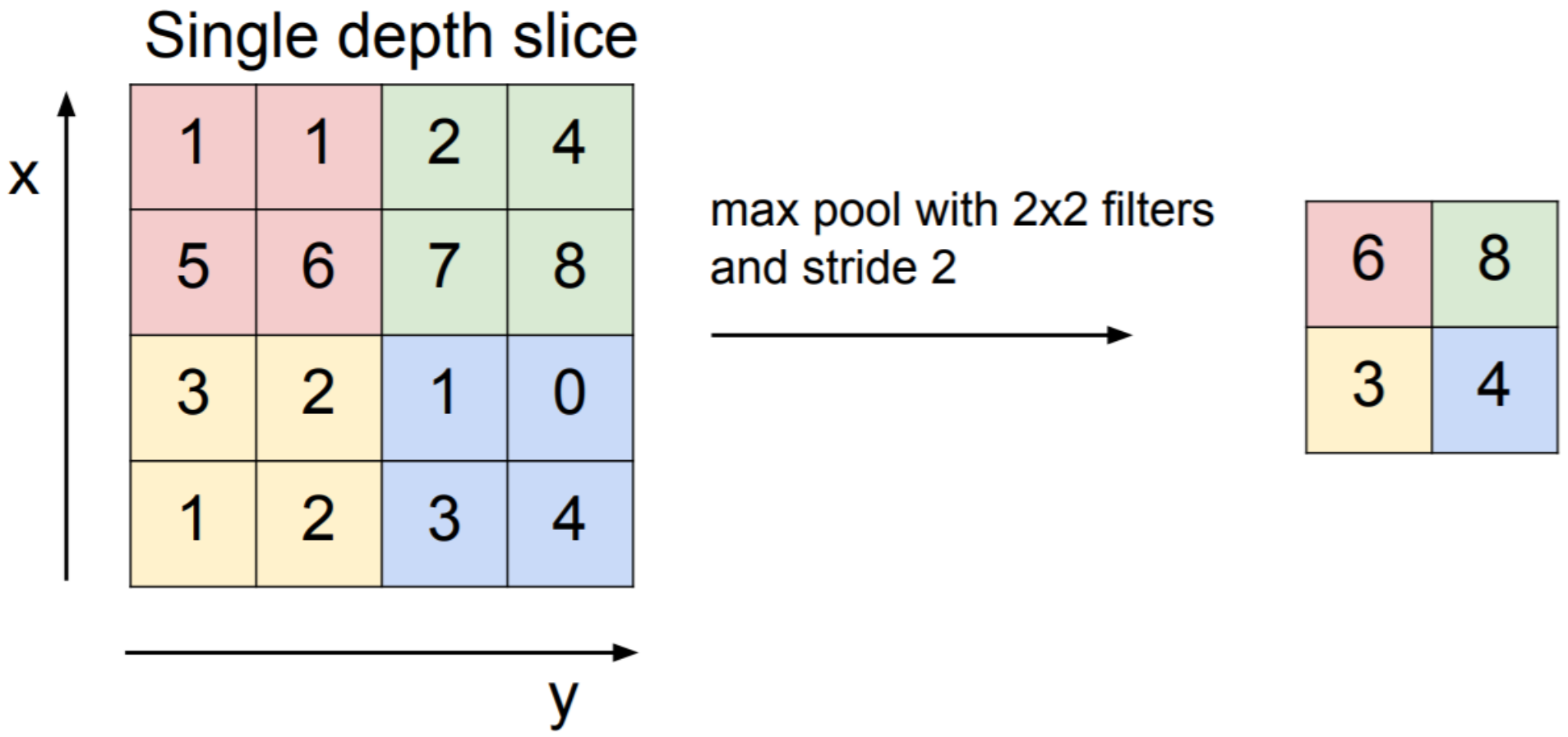

- Convolution (and activation) layers are followed intermittently by "downsampling" (or "pooling") layers
  - Often, they alternate with convolution, though this is not necessary

# Pooling layer

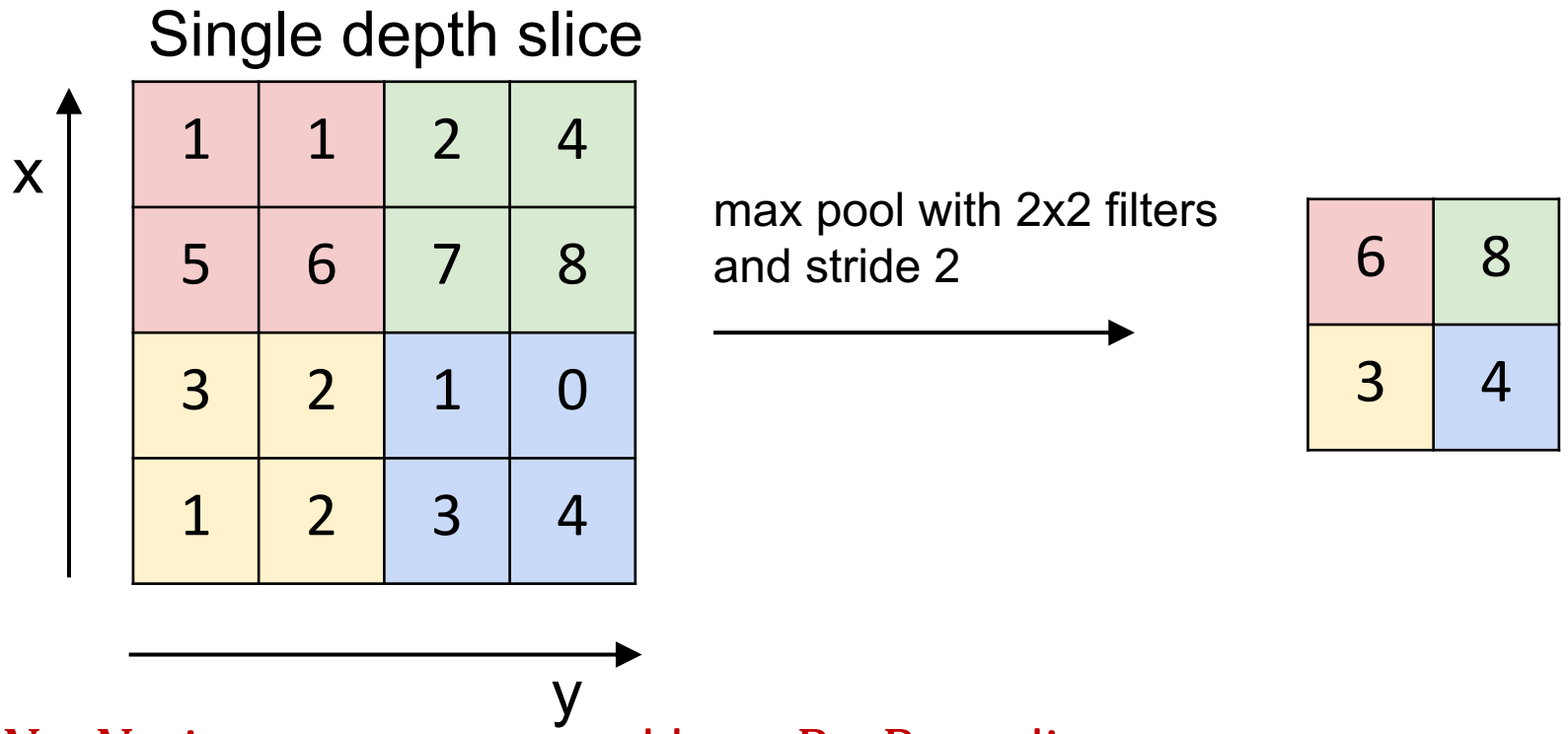
- makes the representations smaller and more manageable
- operates over each activation map independently:



# MAX pooling

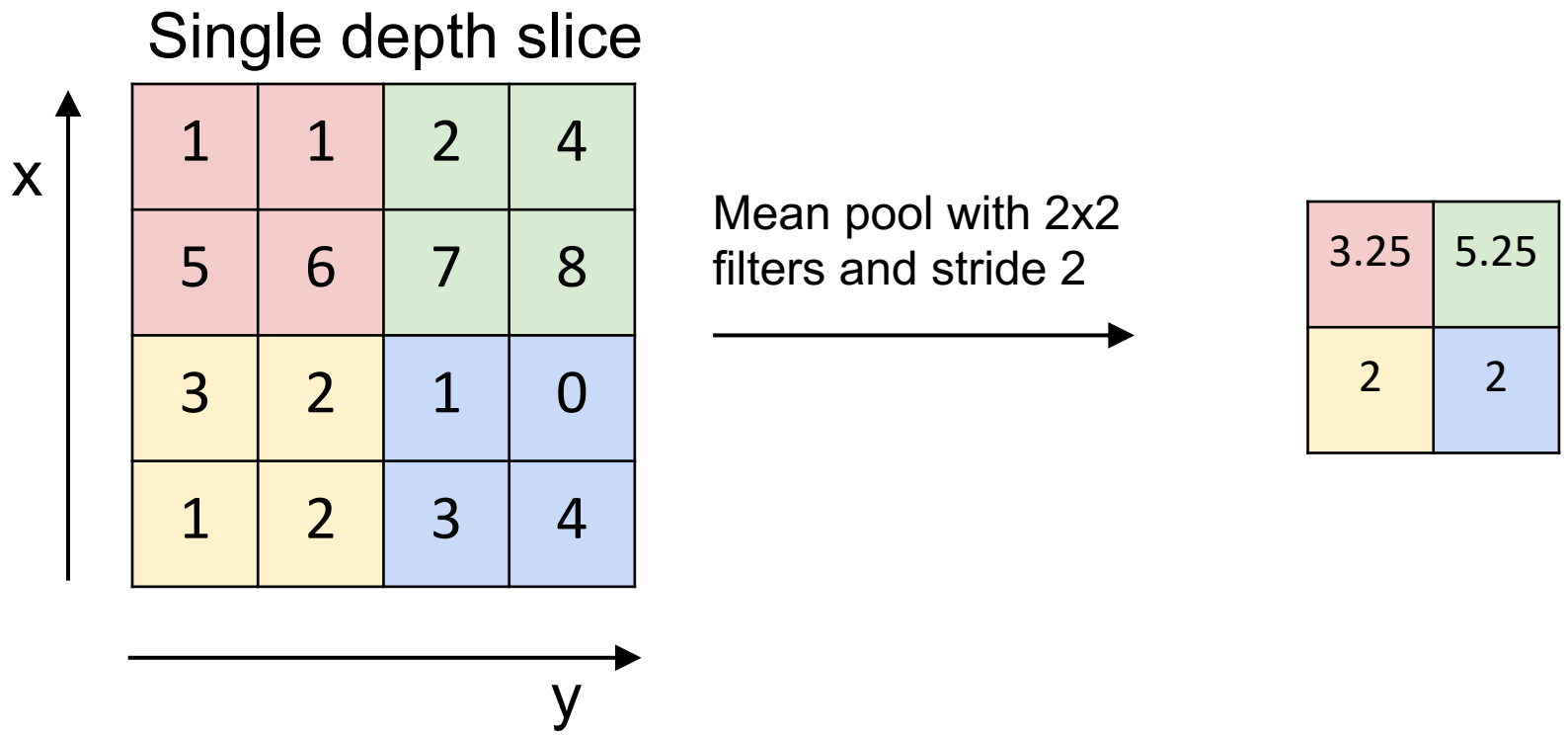

Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4

# Pooling : Size of output



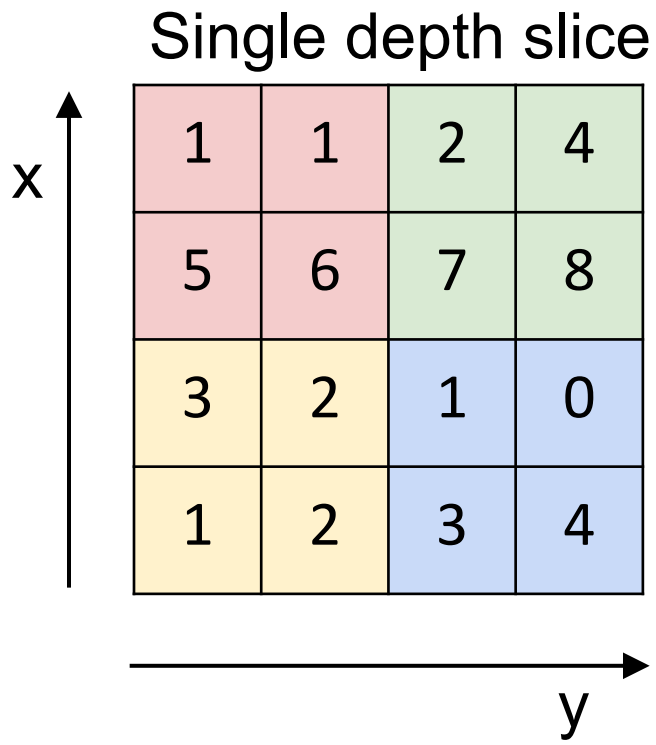

- An $N \times N$ picture compressed by a $P \times P$ pooling filter with stride $S$ results in an output map of side $\lceil (N - P)/S \rceil + 1$
- Typically do not zero pad

# Alternative to Max pooling: Mean pooling




- Compute the mean of the pool, instead of the max

# Alternative to Max pooling: P-norm



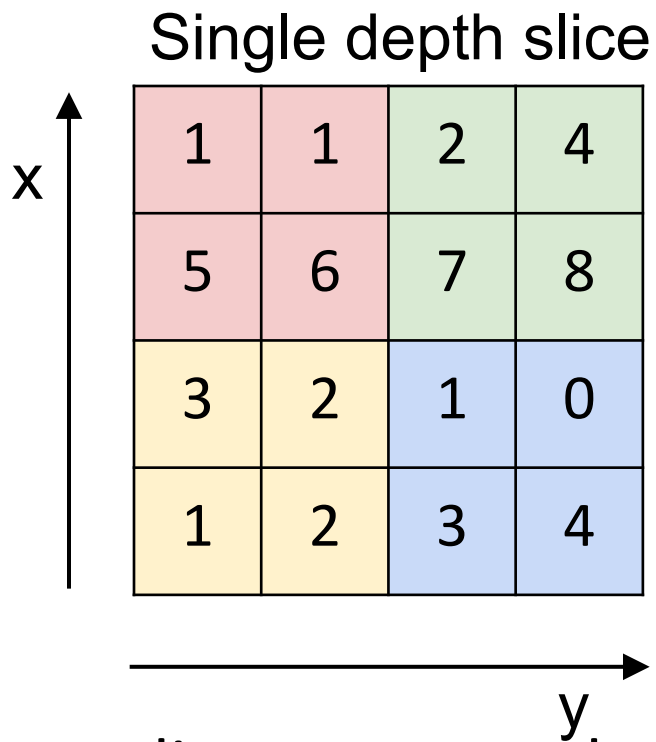

P-norm with 2x2 filters and stride 2, $P = 5$

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

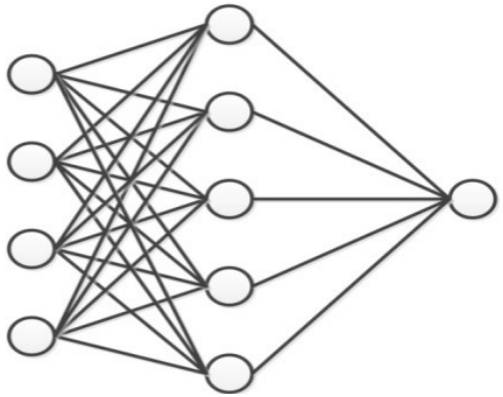
| | |
|---|---|
| 4.86 | 8 |
| 2.38 | 3.16 |

- Compute a p-norm of the pool

# Other options




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



| | |
|---|---|
| 6 | 8 |
| 3 | 4 |

Network in network

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

# Pooling

- reduce the spatial size of the representation
  - to reduce the amount of parameters and computation in the network
  - to control overfitting
- operates independently on every depth slice of the input and resizes it spatially, using the pooling (e.g. MAX) operation.

# Pooling

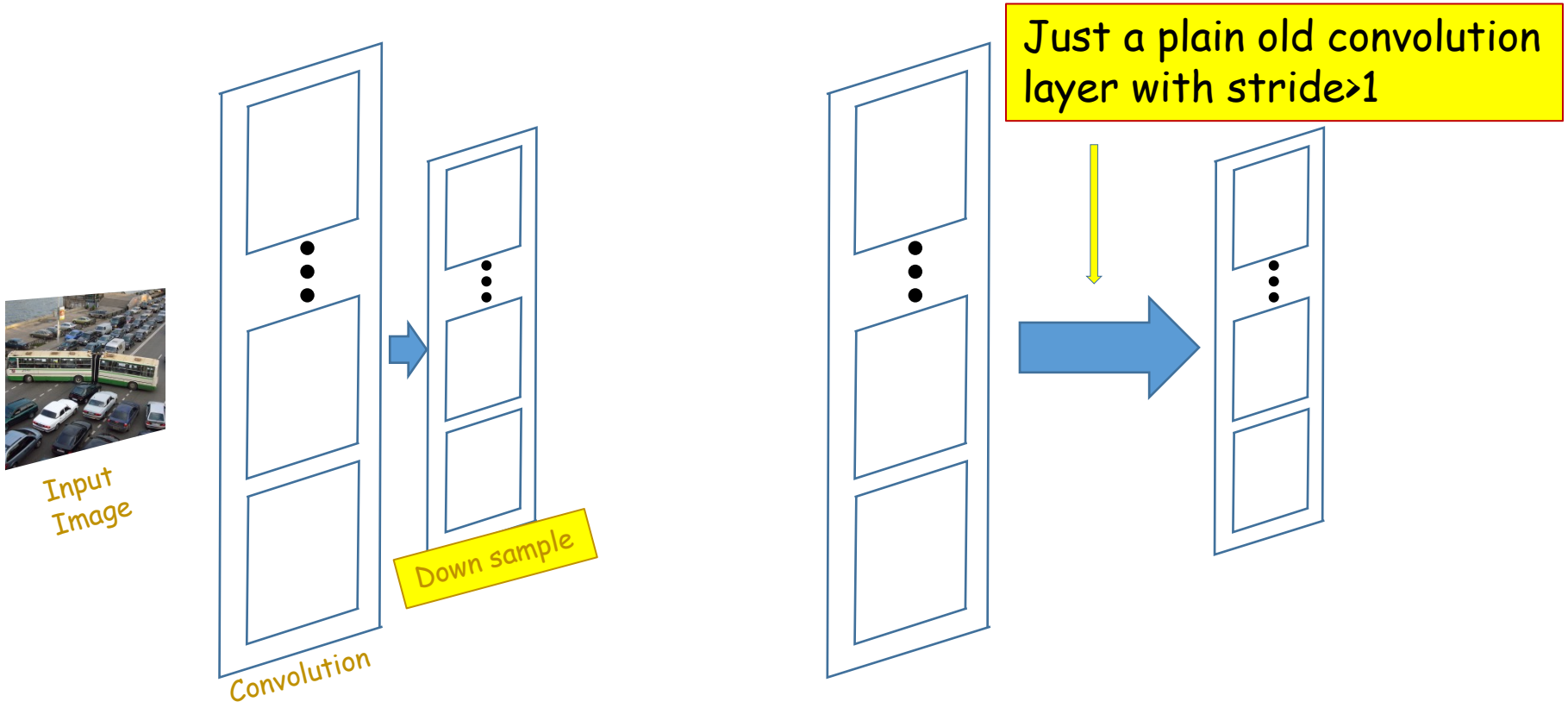
- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires two hyperparameters:
  - their spatial extent $F$,
  - the stride $S$,
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F)/S + 1$
  - $H_2 = (H_1 - F)/S + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for Pooling layers

Common settings:

F = 2, S = 2

F = 3, S = 2

# 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

# Story so far

- CNN includes
  - Convolutional layers comprising learned filters that scan the outputs of the previous layer
  - Downsampling layers that vote over groups of outputs from the convolutional layer

- Downsampling layers may perform max, p-norms, or be learned downsampling networks

- Regular convolutional layers with stride > 1 also perform downsampling
  - Eliminating the need for explicit downsampling layers

# 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 $S$ *decreases* the *size* of the maps by a factor of $S$
- In general the number of convolutional filters increases with layers

# Parameters to choose (design choices)

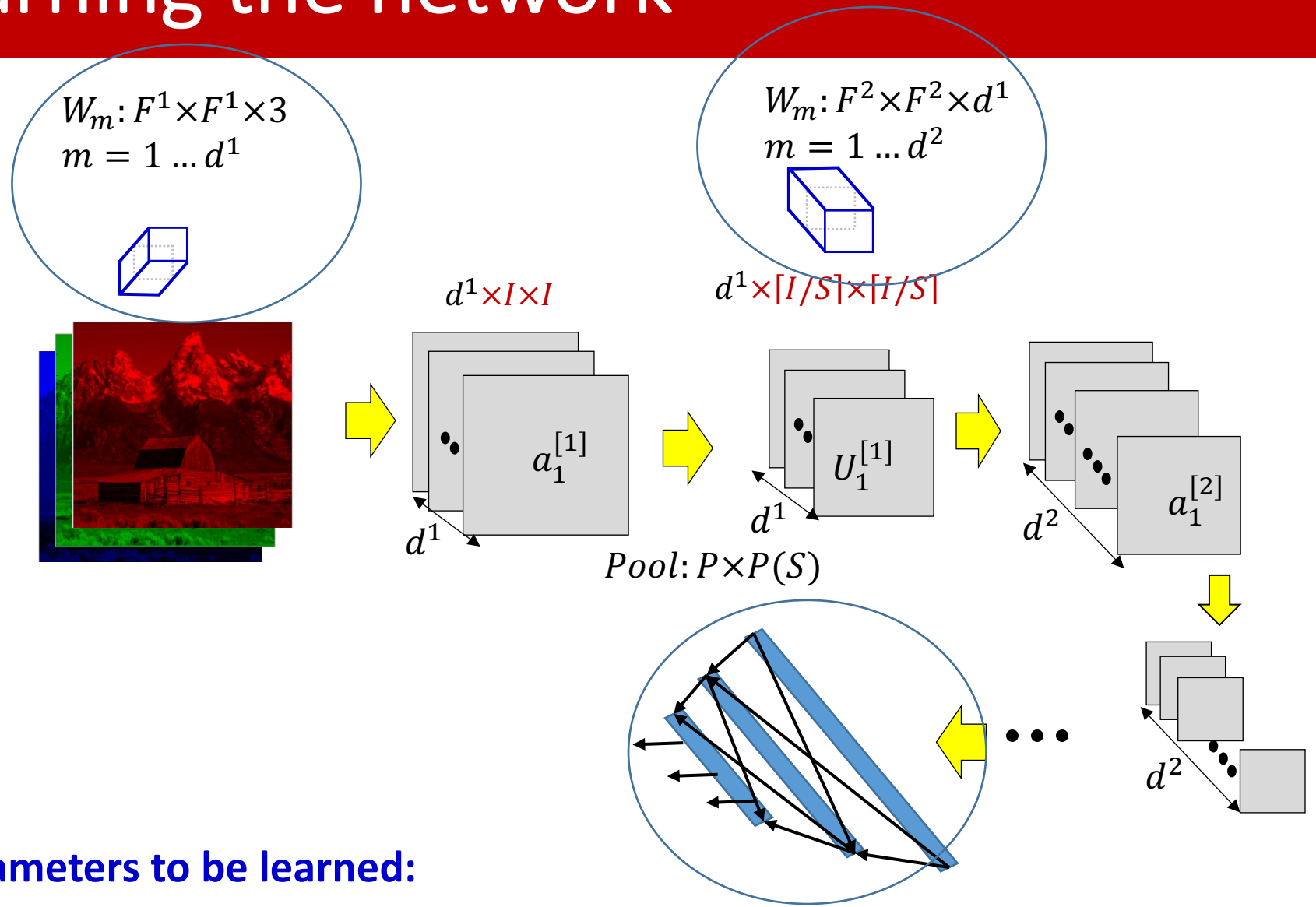
- Number of convolutional and downsampling layers
  - And arrangement (order in which they follow one another)

- For each convolution layer:
  - Number of filters $d^l$
  - Spatial extent of filter $F^l \times F^l$
    - The "depth" of the filter is fixed by the number of filters in the previous layer $d^{l-1}$
  - The stride $S^l$

- For each downsampling/pooling layer:
  - Spatial extent of filter $P^l \times P^l$
  - The stride $S^l$

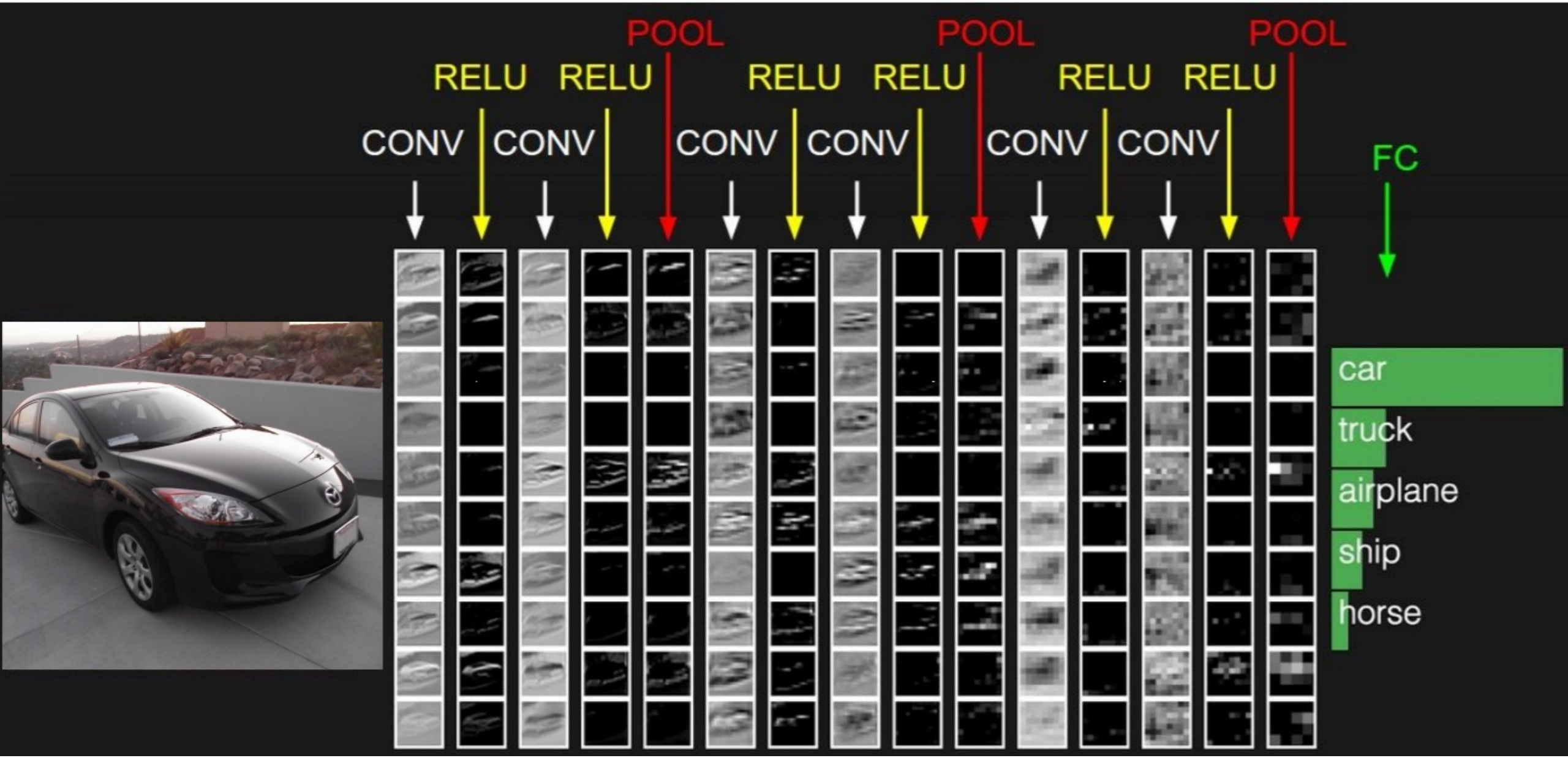
- For the final MLP:
  - Number of layers, and number of neurons in each layer

# 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**

preview:


POOL
POOL
POOL
RELU
RELU
RELU
RELU
RELU
RELU
CONV
CONV
CONV
CONV
CONV
CONV
FC
car
truck
airplane
ship
horse

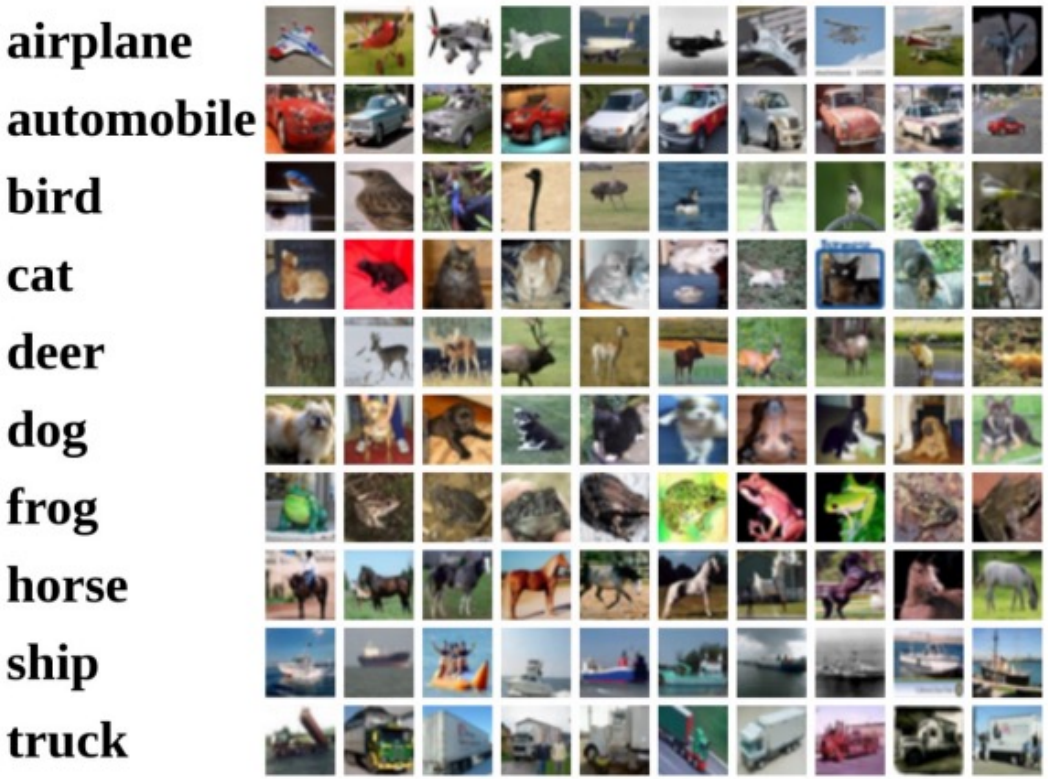
# Typical architecture for CNNs

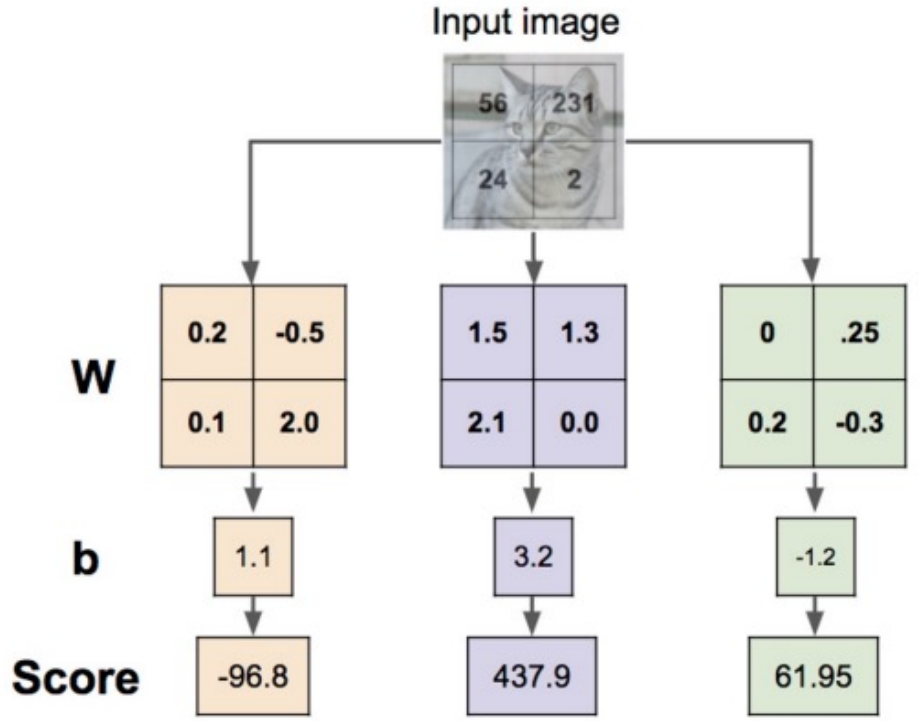
- [(CONV-RELU)*N – POOL?]*M – (FC-RELU)*K – SOFTMAX
  - Where N is usually up to ~5
  - M is large
  - 0 <= K <= 2

But recent advances such as ResNet and GoogleNet challenge this paradigm!

[Goodfellow et al. 2016]

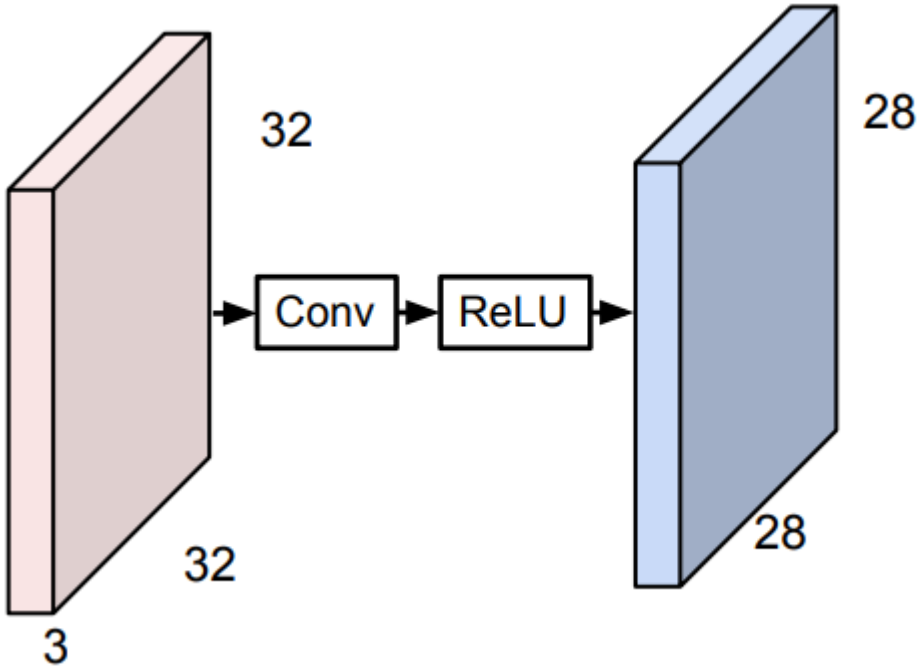
# Interpreting a Linear Classifier: Visual Viewpoint




# convolutional filters

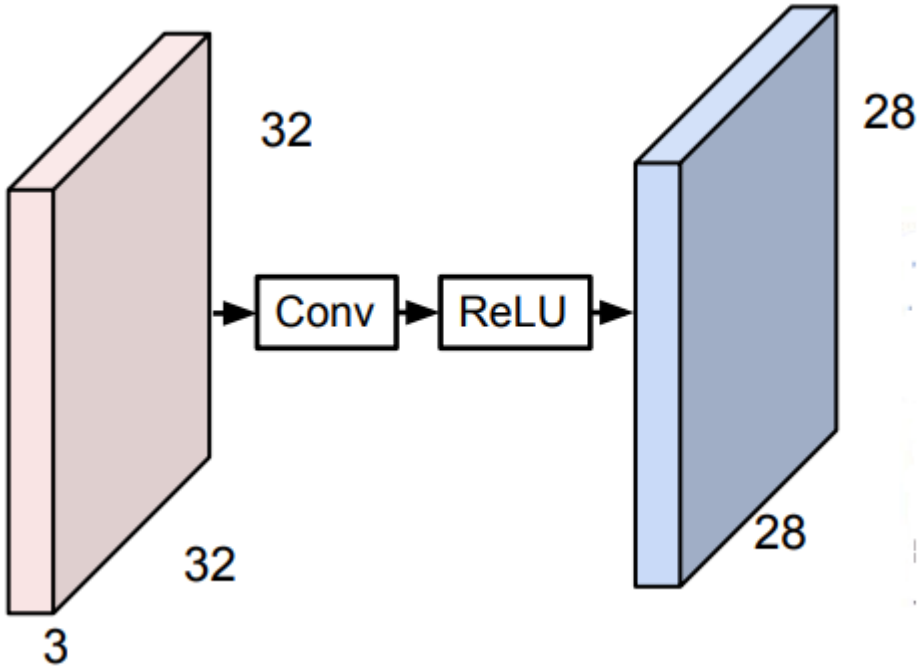
**Preview**: What do convolutional filters learn?




**Linear classifier**: One template per class

# convolutional filters

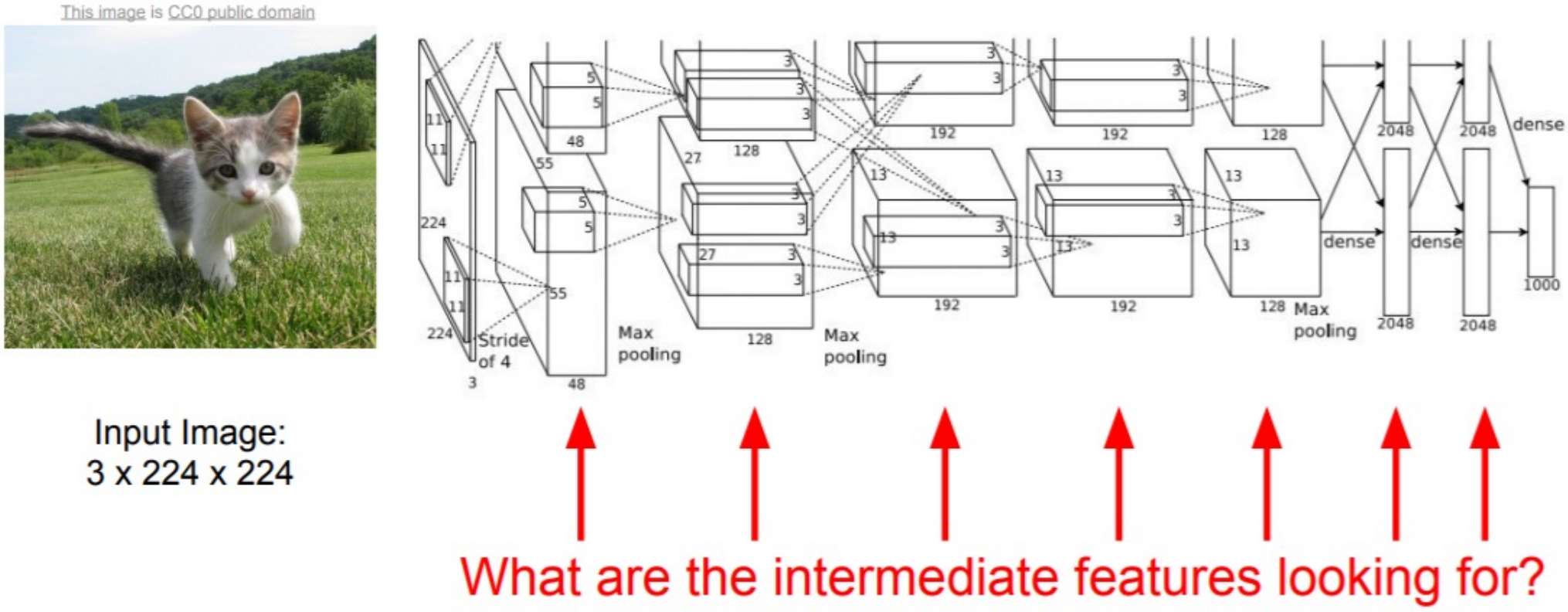
**Preview**: What do convolutional filters learn?




MLP: Bank of whole-image templates

# Interpretation for CNN layers

- Visualizing features to gain intuition about




Krizhevsky et al, "ImageNet Classification with Deep Convolutional Neural Networks", NIPS 2012.
Figure reproduced with permission.

# Visualize the filters/kernels (raw weights)

- We can visualize filters at higher layers, but not that interesting

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

layer 1 weights
16 x 3 x 7 x 7

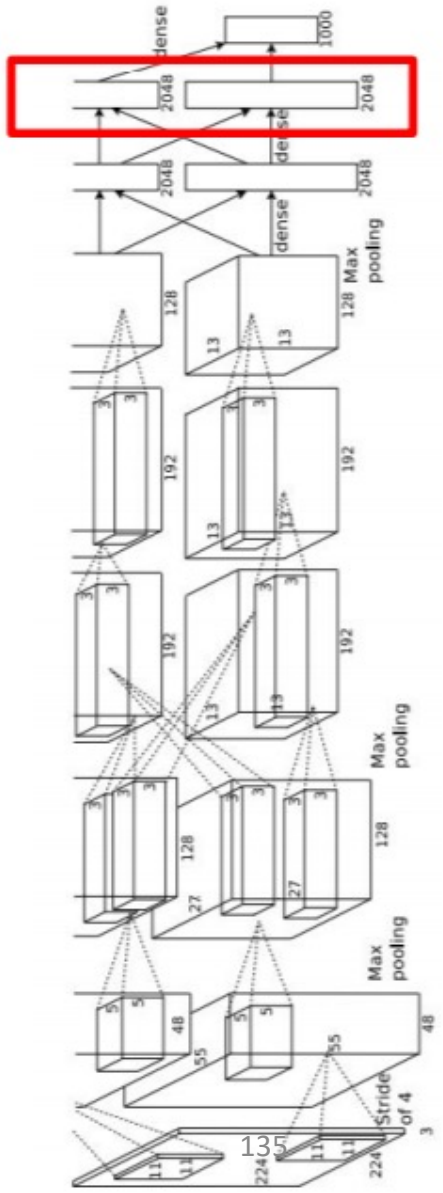

layer 2 weights
20 x 16 x 7 x 7


layer 3 weights
20 x 20 x 7 x 7

# Last layer

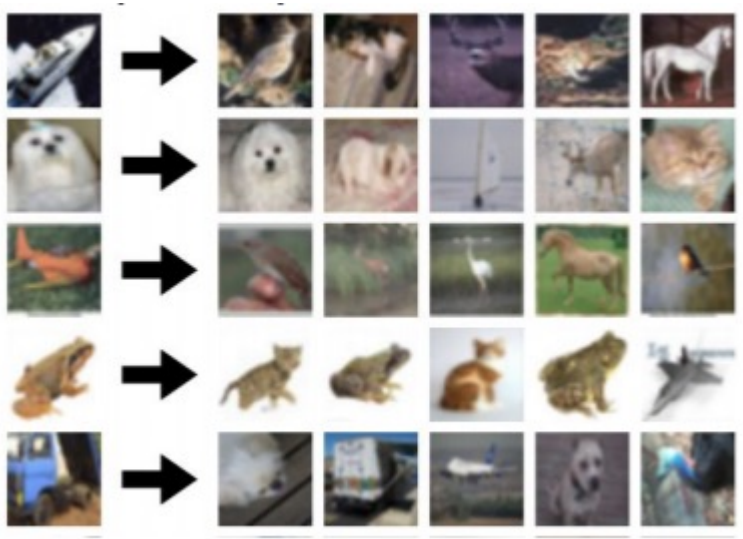
- 4096-dimensional feature vector for an image (layer immediately before the classifier)

- Run the network on many images, collect the feature vectors

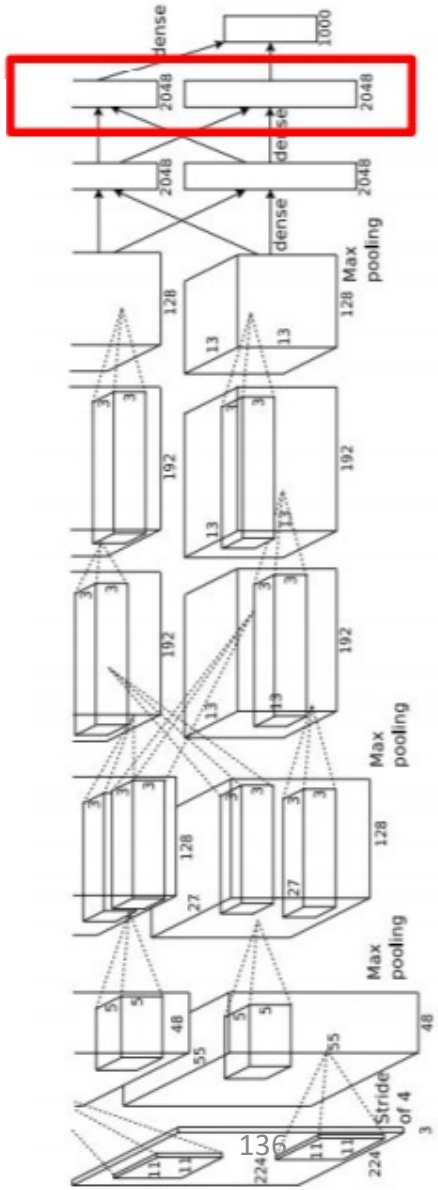


# Last layer: Nearest neighbors

Nearest neighbors in pixel space



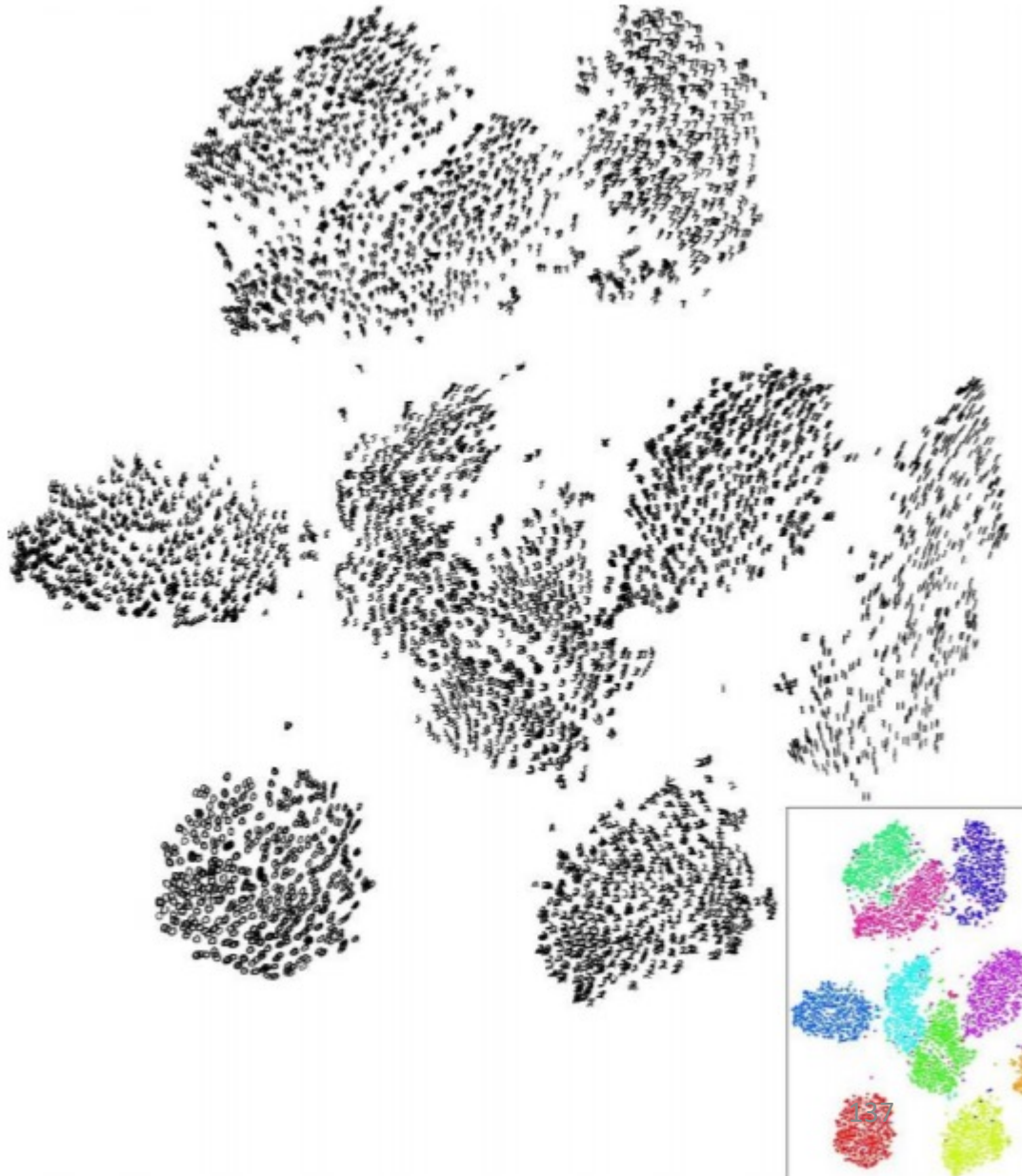
Test image L2 Nearest neighbors in feature space




Krizhevsky et al, "ImageNet Classification with Deep Convolutional Neural Networks", NIPS 2012.
Figures reproduced with permission.

# Last Layer: Dimensionality Reduction

- Visualize the "space" of FC7 feature vectors by reducing dimensionality of vectors from 4096 to 2 dimensions:
  - Simple algorithm: Principle Component Analysis (PCA)
  - More complex: t-SNE



Van der Maaten and Hinton, "Visualizing Data using t-SNE", JMLR 2008
Figure copyright Laurens van der Maaten and Geoff Hinton, 2008. Reproduced with permission.

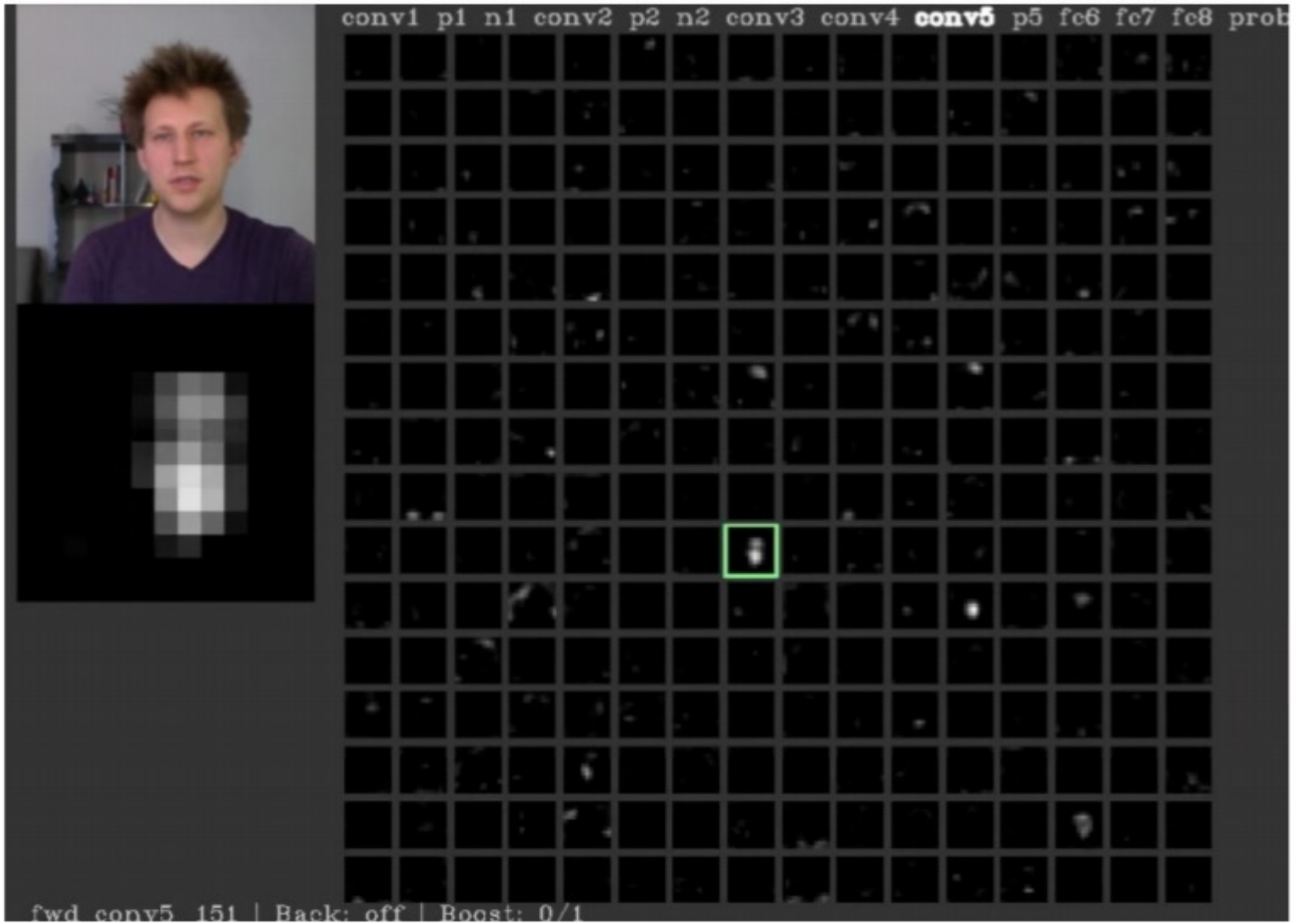
# Last Layer: Dimensionality Reduction

Van der Maaten and Hinton, "Visualizing Data using t-SNE", JMLR 2008
Krizhevsky et al, "ImageNet Classification with Deep Convolutional Neural Networks", NIPS 2012.
Figure reproduced with permission.

See high-resolution versions at
http://cs.stanford.edu/people/karpathy/cnnembed/

# Visualizing Activations

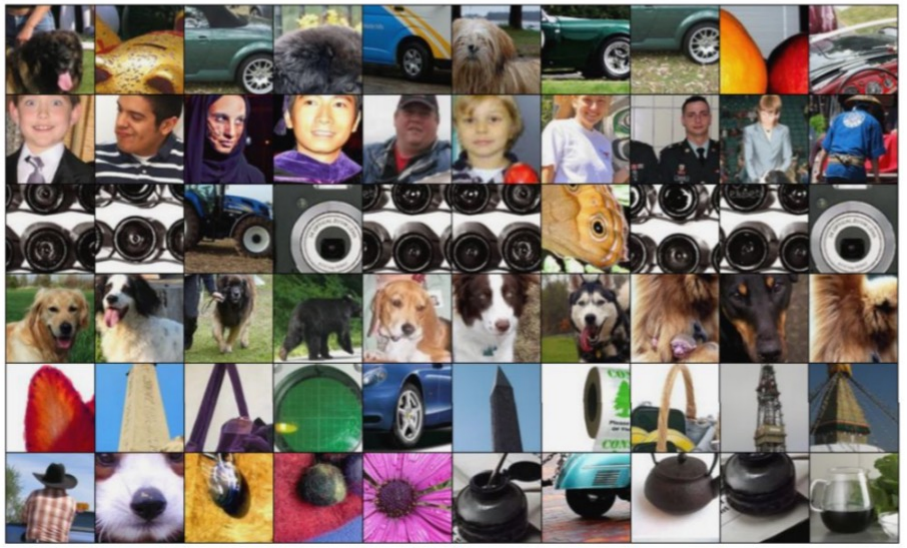
- conv5 feature map is 128x13x13;
  - visualize as 128 13x13 grayscale images

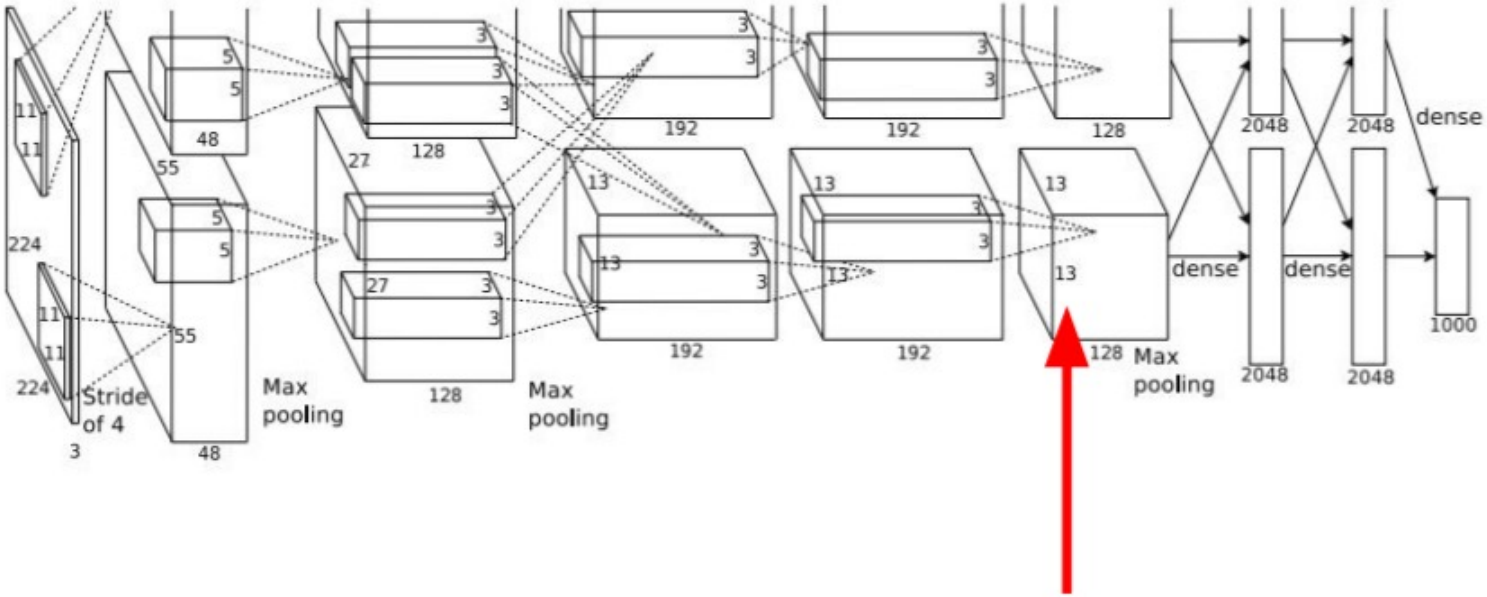



Yosinski et al, "Understanding Neural Networks Through Deep Visualization", ICML DL Workshop 2014.
Figure copyright Jason Yosinski, 2014. Reproduced with permission.

# Maximally Activating Patches

- Pick a layer and a channel;
  - e.g. conv5 is 128 x 13 x 13, pick channel 17/128
  - Run many images through the network, record values of chosen channel
  - Visualize image patches that correspond to maximal activations






Springenberg et al, "Striving for Simplicity: The All Convolutional Net", ICLR Workshop 2015
Figure copyright Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, Martin Riedmiller, 2015; reproduced with permission.

# Demo

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

# Summary

- ConvNets stack CONV,POOL,FC layers
- Trend towards smaller filters and deeper architectures
- Trend towards getting rid of POOL/FC layers (just CONV)

# Conslusion

- Highly structured nature of image data
- Local correlations are important
- CNNs have local and shared connections
- CNNs make strong inductive biases
- To exploit the two-dimensional structure of image data to create inductive biases, we can use four interrelated concepts:
  - Hierarchy
  - Locality
  - Equivariance
  - Invariance ??

# Resources

- Bishop Deep Learning, Chapter 10.
- Please see the following note:
  - http://cs231n.github.io/convolutional-networks/