

Today:

# Deep Learning

Source : Fei-Fei Li, Jiajun Wu, Ruohan Gao

dall-e-2



“Teddy bears working on new AI research on the moon in the 1980s.”



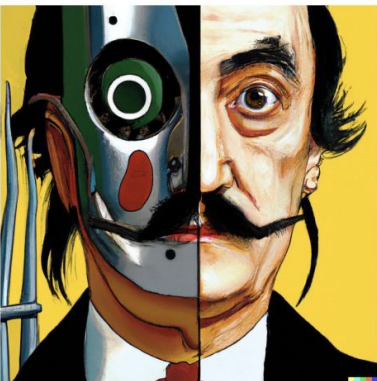
“Rabbits attending a college seminar on human anatomy.”



“A wise cat meditating in the Himalayas searching for enlightenment.”

Image source: Sam Altman, <https://openai.com/dall-e-2/>, <https://twitter.com/sama/status/1511724264629678084>





vibrant portrait painting of Salvador Dalí with a robotic half face



a shiba inu wearing a beret and black turtleneck



a close up of a handpalm with leaves growing from it



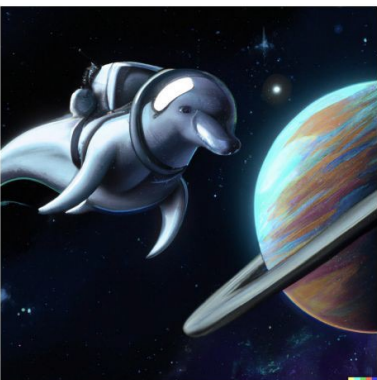
an espresso machine that makes coffee from human souls, artstation



panda mad scientist mixing sparkling chemicals, artstation



a corgi's head depicted as an explosion of a nebula



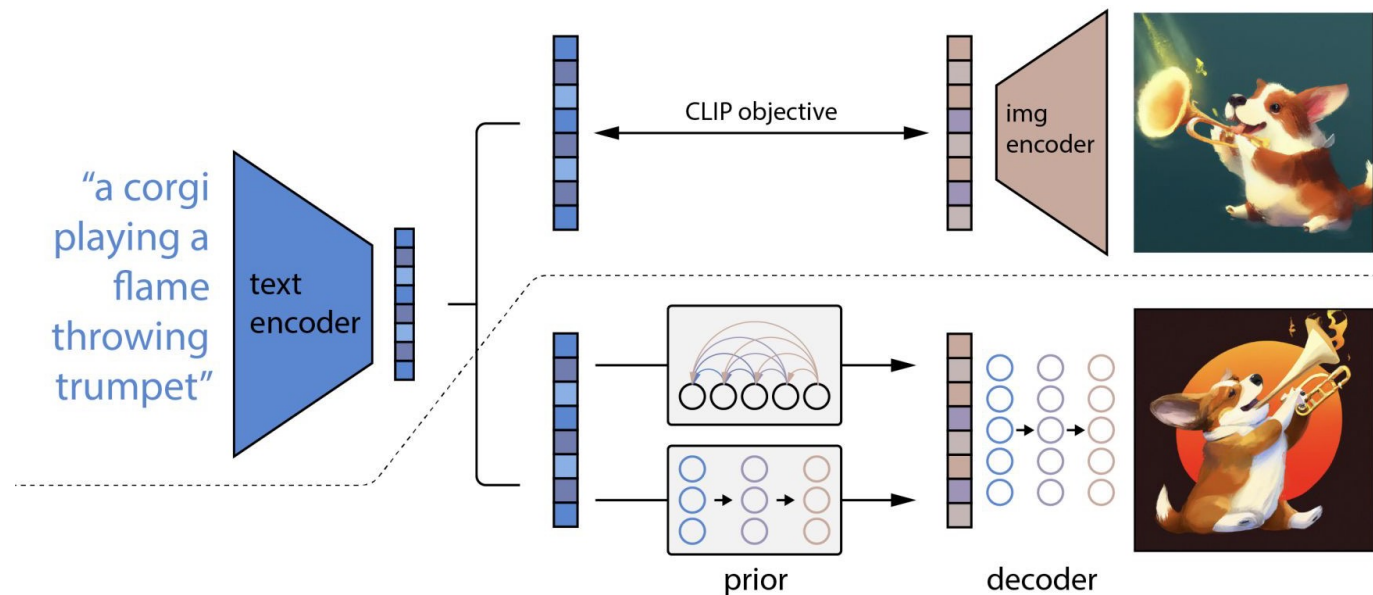
a dolphin in an astronaut suit on saturn, artstation



a propaganda poster depicting a cat dressed as french emperor napoleon holding a piece of cheese



a teddybear on a skateboard in times square



Ramesh et al., Hierarchical Text-Conditional Image Generation with CLIP Latents, 2022.

# Neural Networks

# Neural networks: the original linear classifier

(**Before**) Linear score function:  $f = Wx$

$$x \in \mathbb{R}^D, W \in \mathbb{R}^{C \times D}$$

# Neural networks: 2 layers

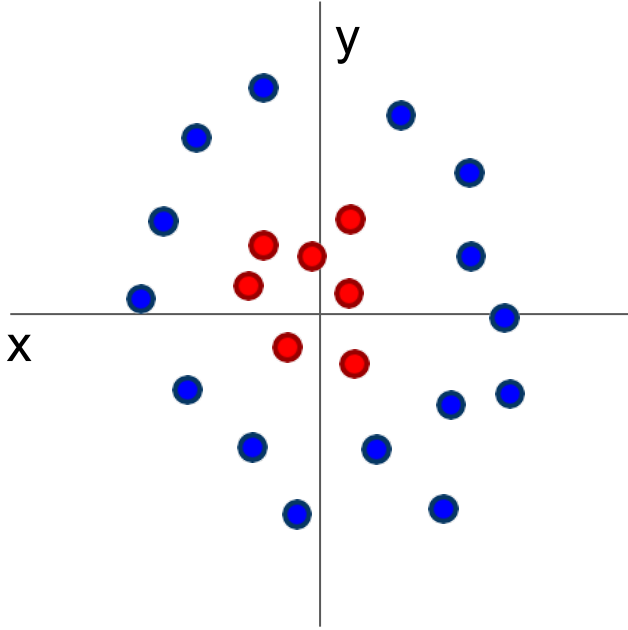
**(Before)** Linear score function:  $f = Wx$

**(Now)** 2-layer Neural Network  $f = W_2 \max(0, W_1 x)$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

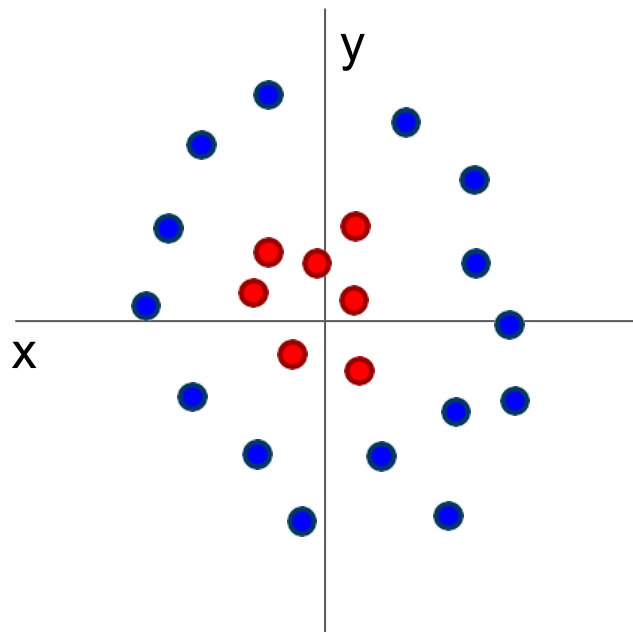
(In practice we will usually add a learnable bias at each layer as well)

# Why do we want non-linearity?



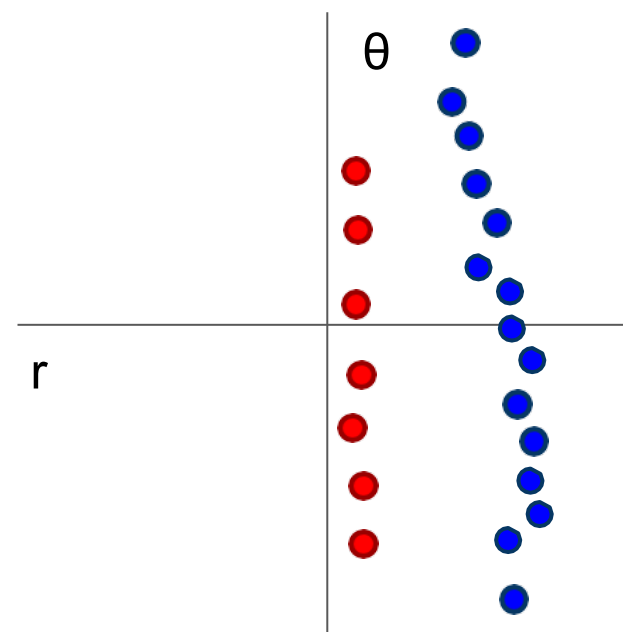
Cannot separate red  
and blue points with  
linear classifier

# Why do we want non-linearity?



Cannot separate red and blue points with linear classifier

$$f(x, y) = (r(x, y), \theta(x, y))$$



After applying feature transform, points can be separated by linear classifier



Neural networks: also called fully connected network

**(Before)** Linear score function:  $f = Wx$

**(Now)** 2-layer Neural Network  $f = W_2 \max(0, W_1 x)$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

“Neural Network” is a very broad term; these are more accurately called “fully-connected networks” or sometimes “multi-layer perceptrons” (MLP)

(In practice we will usually add a learnable bias at each layer as well)

# Neural networks: 3 layers

**(Before)** Linear score function:  $f = Wx$

**(Now)** 2-layer Neural Network  
or 3-layer Neural Network  $f = W_2 \max(0, W_1 x)$

$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$

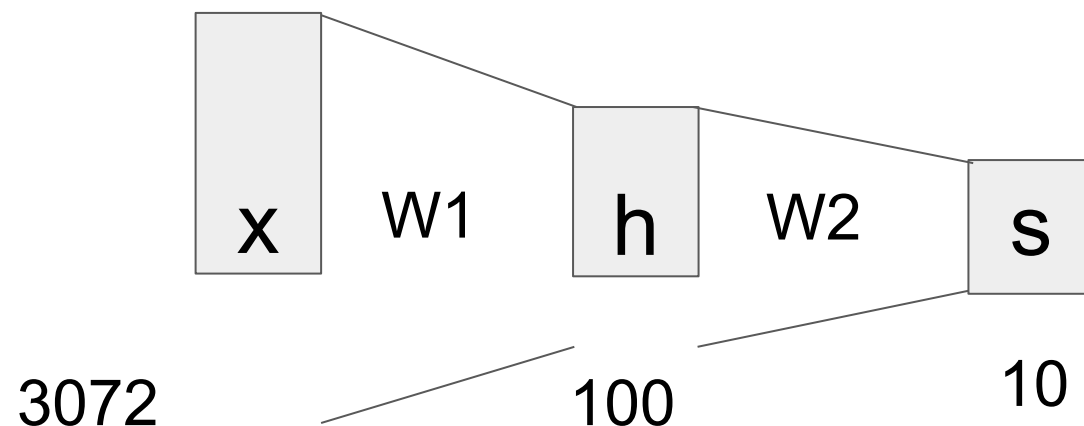
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H_1 \times D}, W_2 \in \mathbb{R}^{H_2 \times H_1}, W_3 \in \mathbb{R}^{C \times H_2}$$

(In practice we will usually add a learnable bias at each layer as well)

# Neural networks: hierarchical computation

(**Before**) Linear score function:  $f = Wx$

(**Now**) 2-layer Neural Network  $f = W_2 \max(0, W_1 x)$

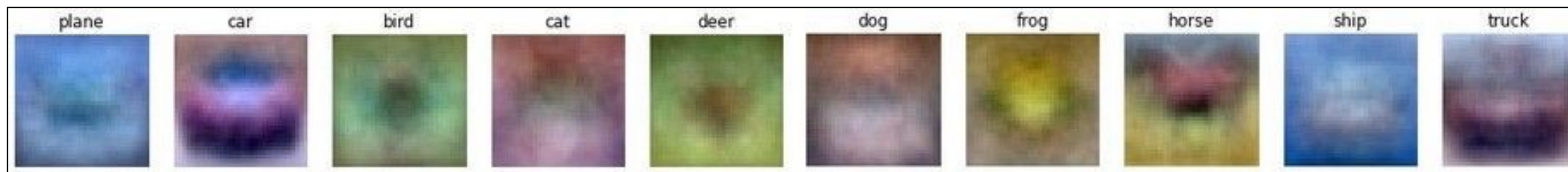
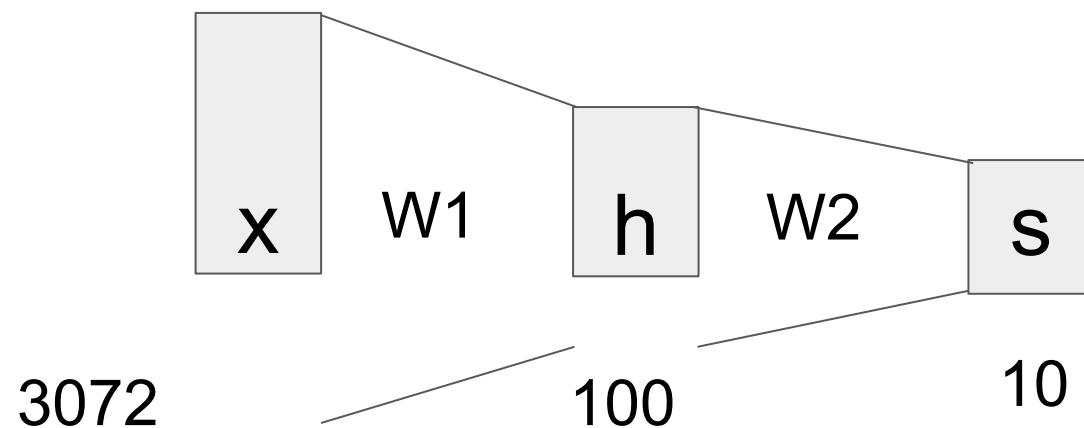


$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

# Neural networks: learning 100s of templates

(**Before**) Linear score function:  $f = Wx$

(**Now**) 2-layer Neural Network  $f = W_2 \max(0, W_1 x)$



Learn 100 templates instead of 10.

Share templates between classes

Neural networks: why is max operator important?

(**Before**) Linear score function:  $f = Wx$

(**Now**) 2-layer Neural Network  $f = W_2 \max(0, W_1 x)$

The function  $\max(0, z)$  is called the **activation function**.

**Q:** What if we try to build a neural network without one?

$$f = W_2 W_1 x$$



Neural networks: why is max operator important?

(**Before**) Linear score function:  $f = Wx$

(**Now**) 2-layer Neural Network  $f = W_2 \max(0, W_1 x)$

The function  $\max(0, z)$  is called the **activation function**.

**Q:** What if we try to build a neural network without one?

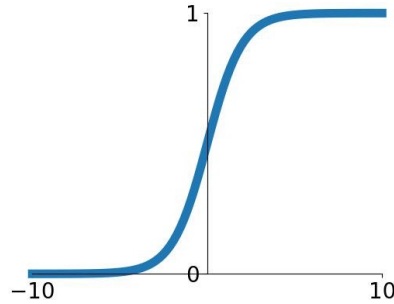
$$f = W_2 W_1 x \quad W_3 = W_2 W_1 \in \mathbb{R}^{C \times H}, f = W_3 x$$

**A:** We end up with a linear classifier again!

# Activation functions

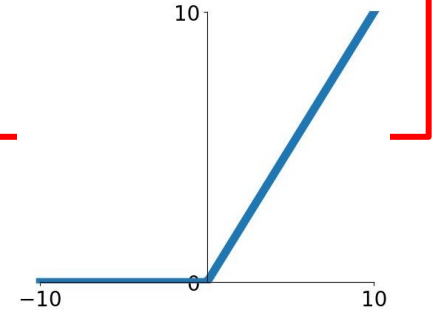
## Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



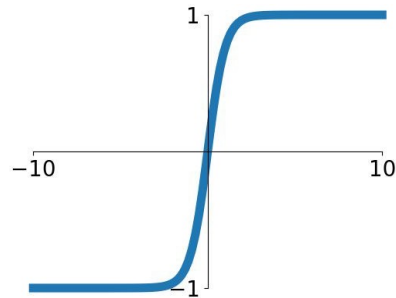
## ReLU

$$\max(0, x)$$

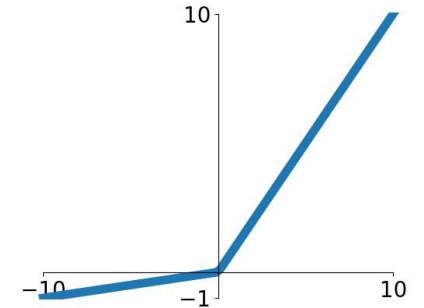


ReLU is a good default choice for most problems

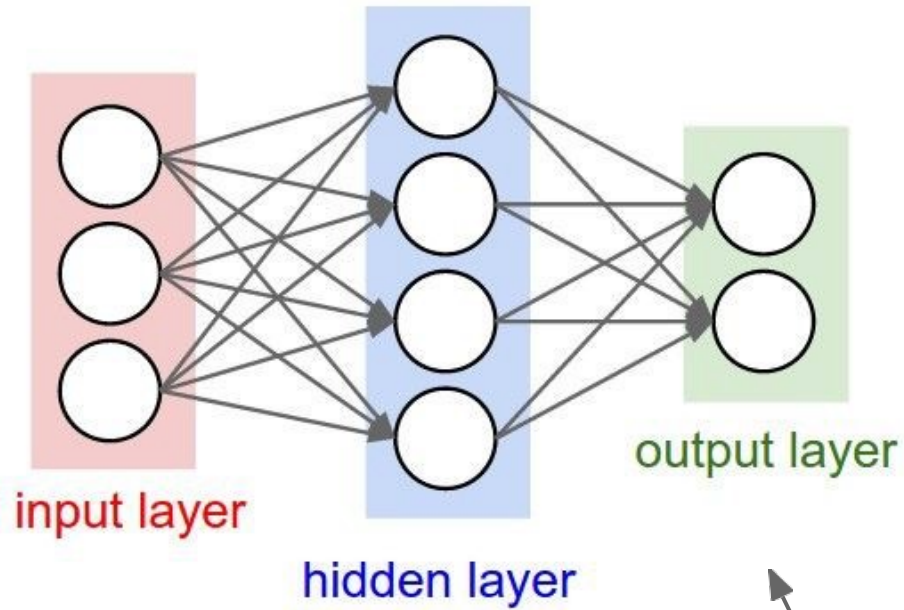
## tanh

$$\tanh(x)$$


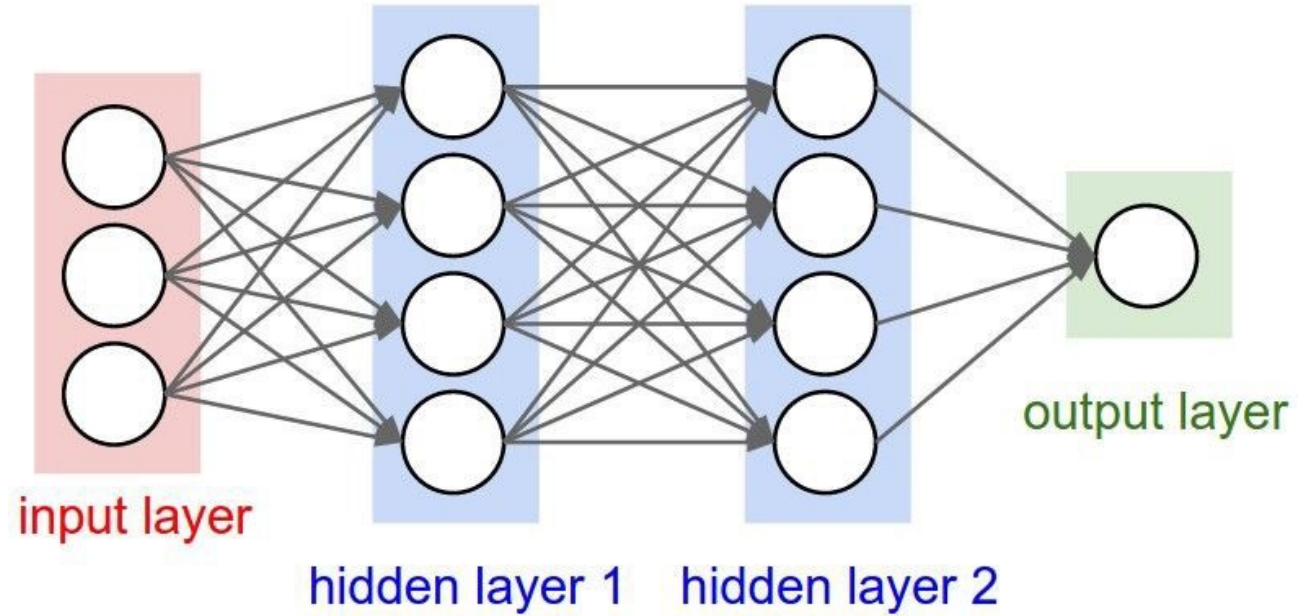
## Leaky ReLU

$$\max(0.1x, x)$$


# Neural networks: Architectures



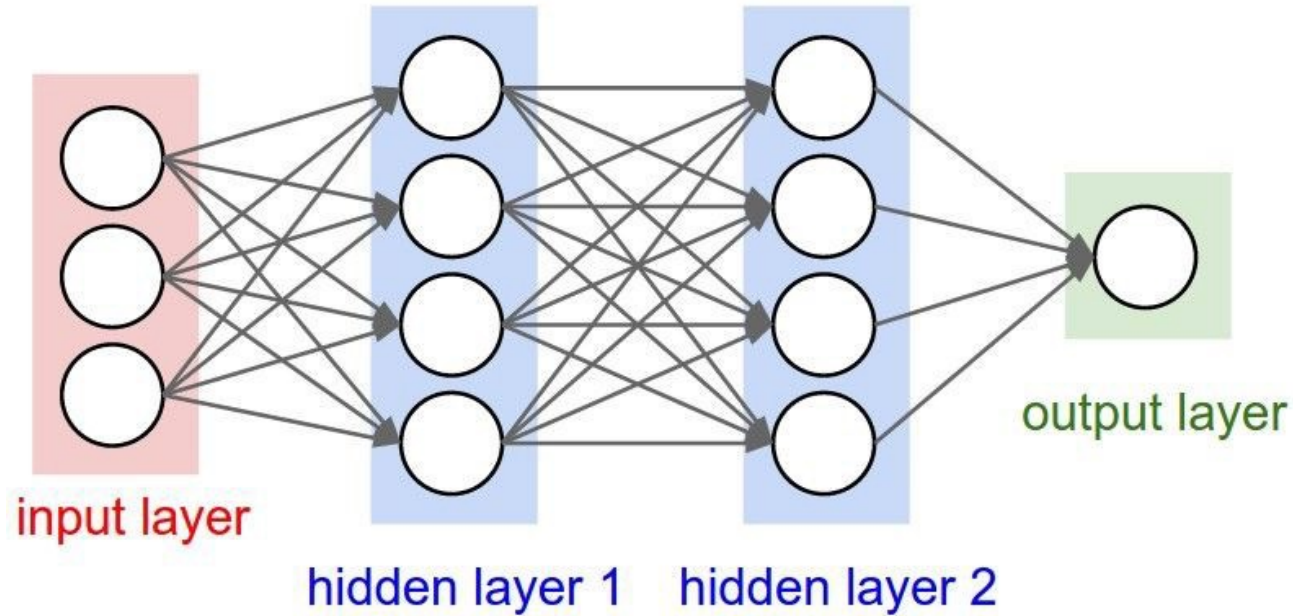
“2-layer Neural Net”, or  
“1-hidden-layer Neural Net”



“3-layer Neural Net”, or  
“2-hidden-layer Neural Net”

**“Fully-connected” layers**

# Example feed-forward computation of a neural network



```
# forward-pass of a 3-layer neural network:  
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)  
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)  
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)  
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)  
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1  import numpy as np
2  from numpy.random import randn
3
4  N, D_in, H, D_out = 64, 1000, 100, 10
5  x, y = randn(N, D_in), randn(N, D_out)
6  w1, w2 = randn(D_in, H), randn(H, D_out)
7
8  for t in range(2000):
9      h = 1 / (1 + np.exp(-x.dot(w1)))
10     y_pred = h.dot(w2)
11     loss = np.square(y_pred - y).sum()
12     print(t, loss)
13
14     grad_y_pred = 2.0 * (y_pred - y)
15     grad_w2 = h.T.dot(grad_y_pred)
16     grad_h = grad_y_pred.dot(w2.T)
17     grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19     w1 -= 1e-4 * grad_w1
20     w2 -= 1e-4 * grad_w2
```



# Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1  import numpy as np
2  from numpy.random import randn
3
4  N, D_in, H, D_out = 64, 1000, 100, 10
5  x, y = randn(N, D_in), randn(N, D_out)
6  w1, w2 = randn(D_in, H), randn(H, D_out)
7
8  for t in range(2000):
9      h = 1 / (1 + np.exp(-x.dot(w1)))
10     y_pred = h.dot(w2)
11     loss = np.square(y_pred - y).sum()
12     print(t, loss)
13
14     grad_y_pred = 2.0 * (y_pred - y)
15     grad_w2 = h.T.dot(grad_y_pred)
16     grad_h = grad_y_pred.dot(w2.T)
17     grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19     w1 -= 1e-4 * grad_w1
20     w2 -= 1e-4 * grad_w2
```

Define the network

Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1  import numpy as np
2  from numpy.random import randn
3
4  N, D_in, H, D_out = 64, 1000, 100, 10
5  x, y = randn(N, D_in), randn(N, D_out)
6  w1, w2 = randn(D_in, H), randn(H, D_out)
7
8  for t in range(2000):
9      h = 1 / (1 + np.exp(-x.dot(w1)))
10     y_pred = h.dot(w2)
11     loss = np.square(y_pred - y).sum()
12     print(t, loss)
13
14     grad_y_pred = 2.0 * (y_pred - y)
15     grad_w2 = h.T.dot(grad_y_pred)
16     grad_h = grad_y_pred.dot(w2.T)
17     grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19     w1 -= 1e-4 * grad_w1
20     w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

Calculate the analytical gradients

Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1  import numpy as np
2  from numpy.random import randn
3
4  N, D_in, H, D_out = 64, 1000, 100, 10
5  x, y = randn(N, D_in), randn(N, D_out)
6  w1, w2 = randn(D_in, H), randn(H, D_out)
7
8  for t in range(2000):
9      h = 1 / (1 + np.exp(-x.dot(w1)))
10     y_pred = h.dot(w2)
11     loss = np.square(y_pred - y).sum()
12     print(t, loss)
13
14     grad_y_pred = 2.0 * (y_pred - y)
15     grad_w2 = h.T.dot(grad_y_pred)
16     grad_h = grad_y_pred.dot(w2.T)
17     grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19     w1 -= 1e-4 * grad_w1
20     w2 -= 1e-4 * grad_w2
```

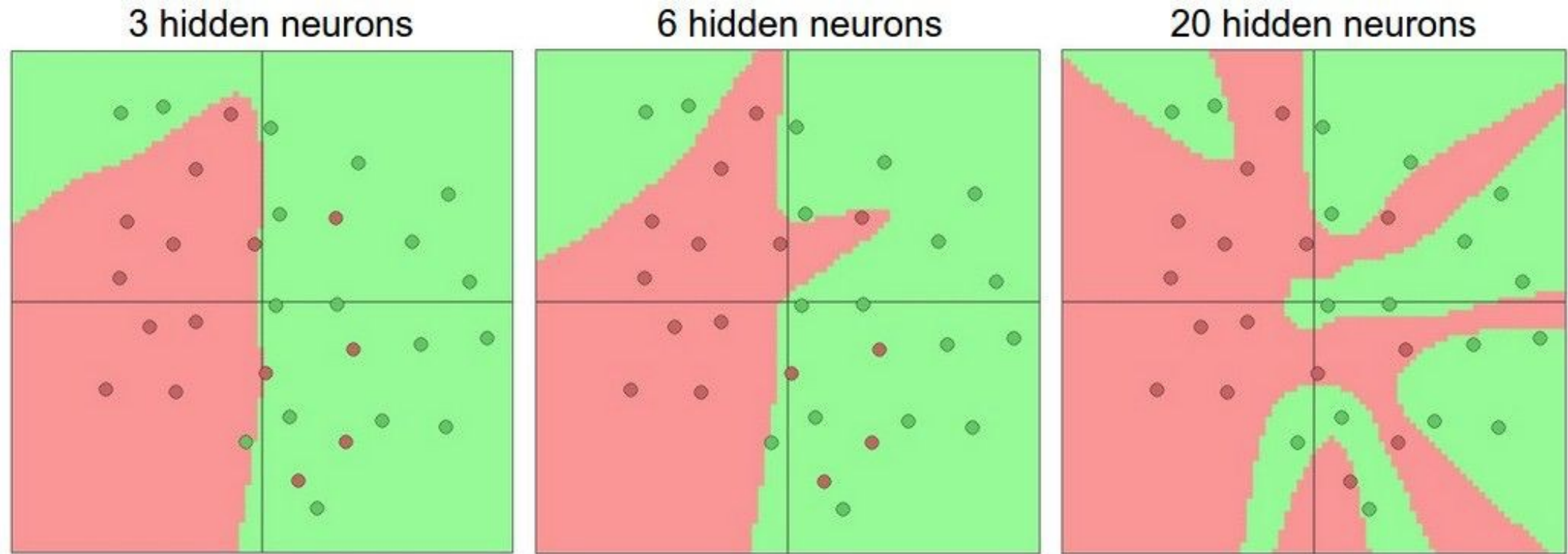
Define the network

Forward pass

Calculate the analytical gradients

Gradient descent

# Setting the number of layers and their sizes



↑  
more neurons = more capacity





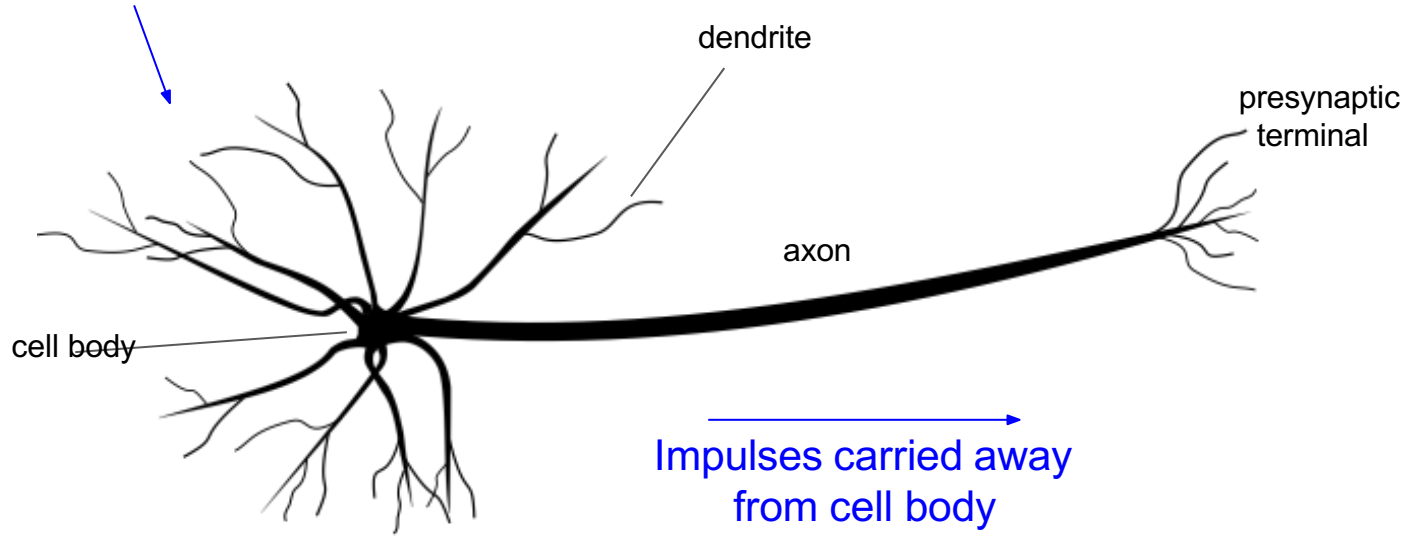
ohan Gao

Lecture 4 - 24

April 07, 2022

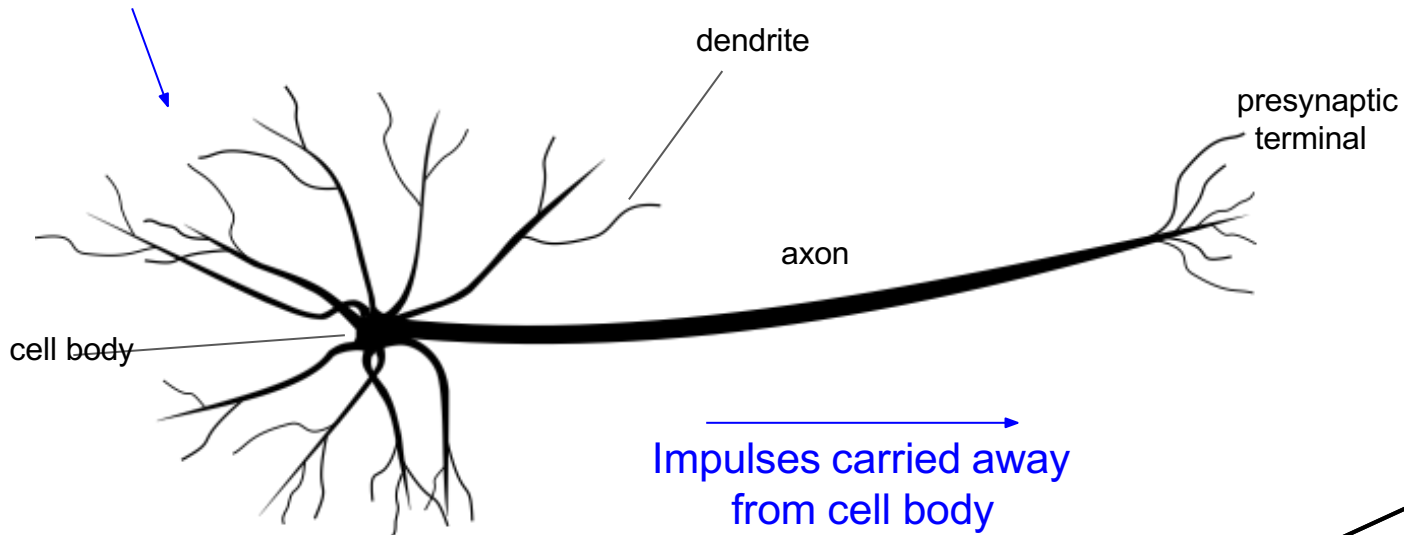
[This image](#) by [Fotis Bobolas](#) is  
licensed under [CC-BY 2.0](#)

Impulses carried toward cell body

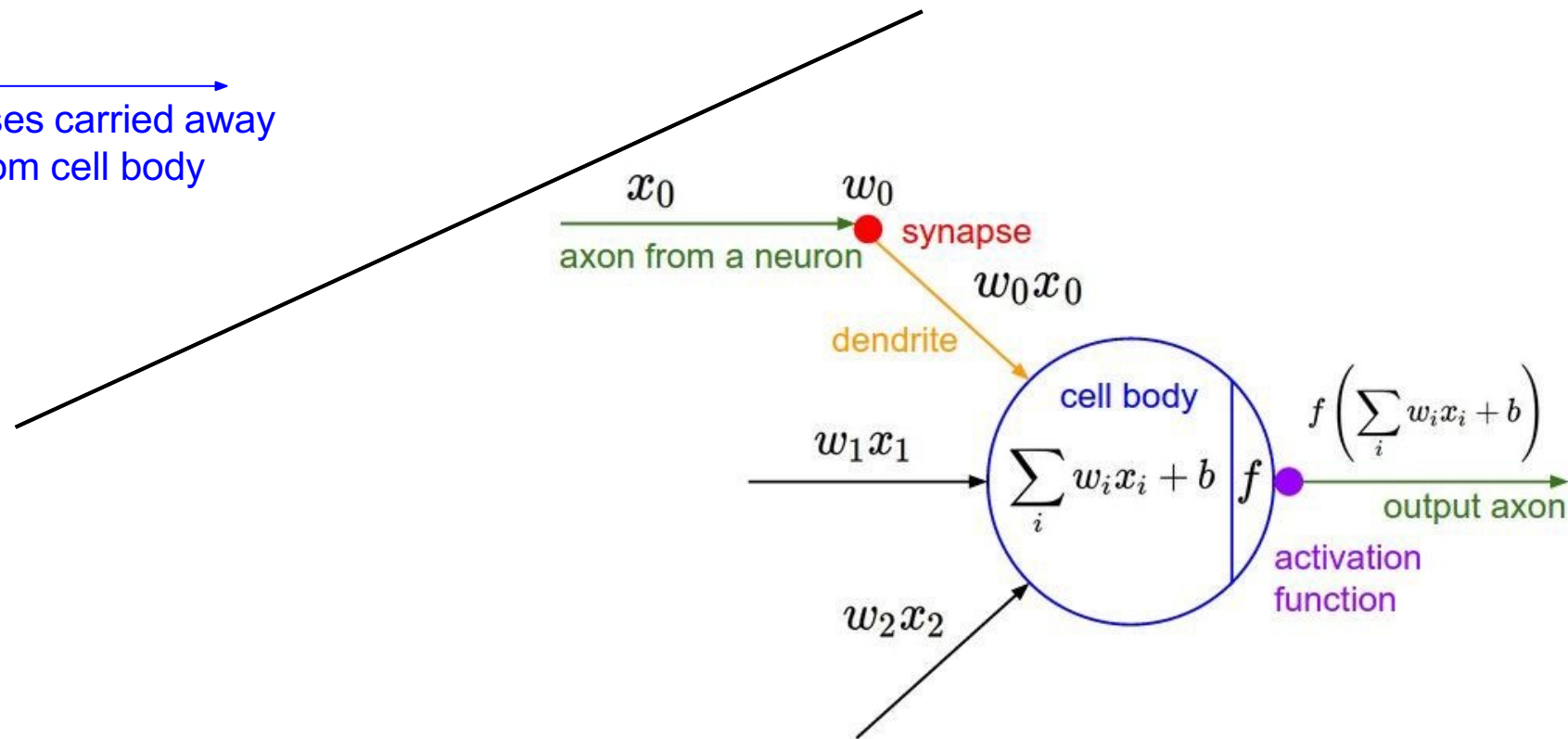


[This image](#) by Felipe Perucho  
is licensed under [CC-BY 3.0](#)

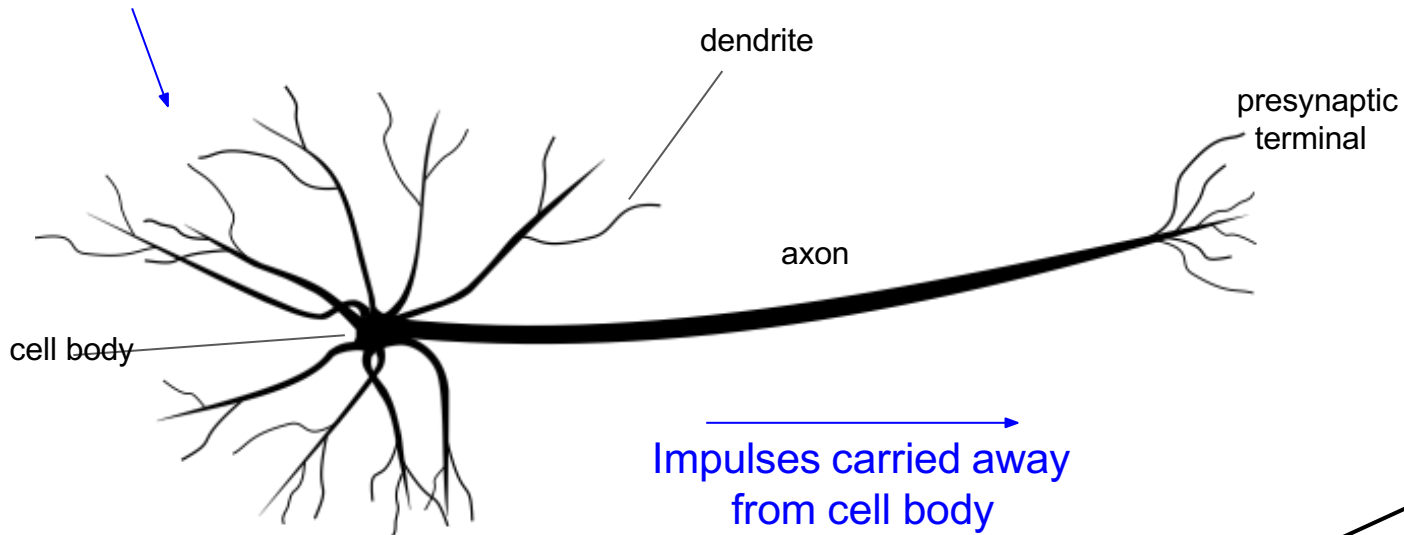
Impulses carried toward cell body



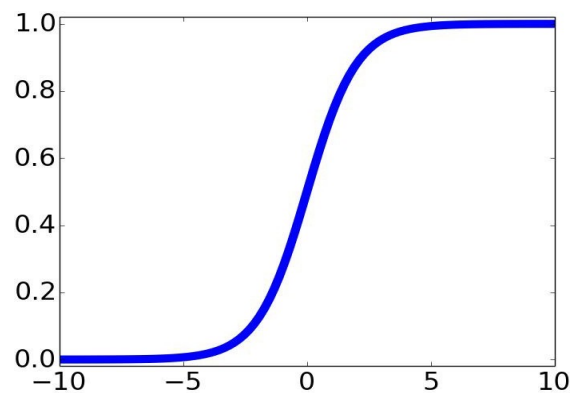
[This image](#) by Felipe Perucho is licensed under [CC-BY 3.0](#)



Impulses carried toward cell body

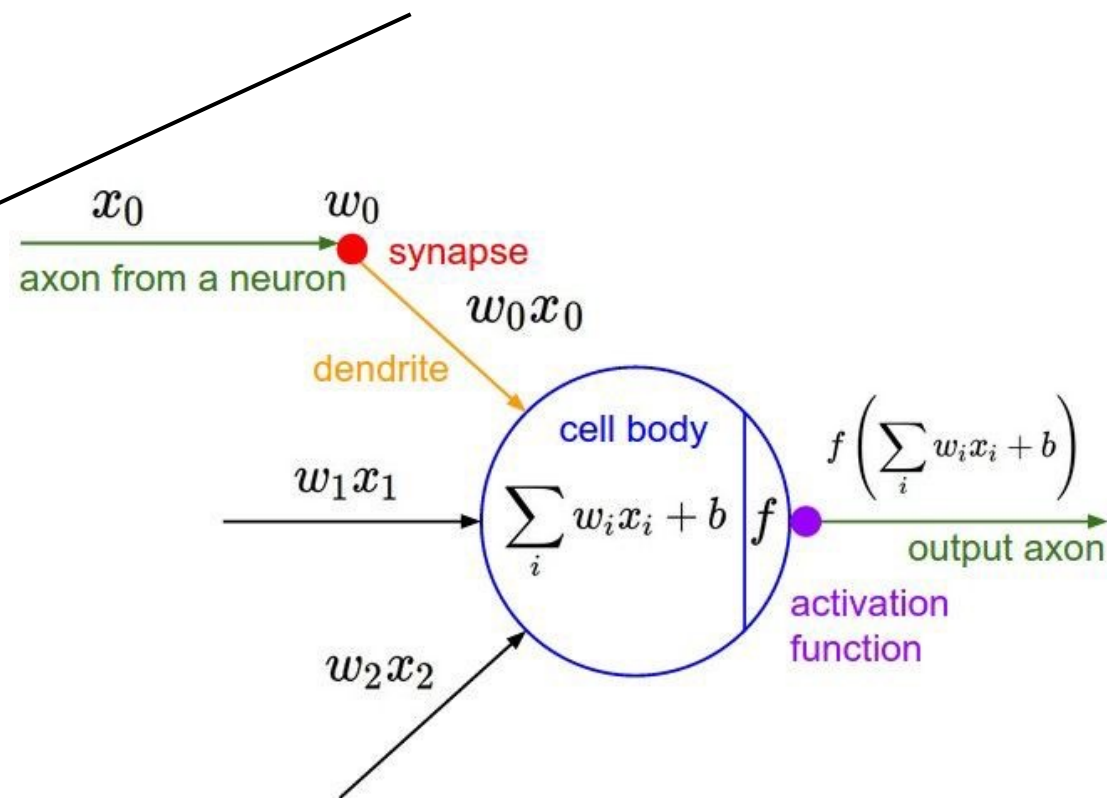


[This image](#) by Felipe Peruchio is licensed under [CC-BY 3.0](#)



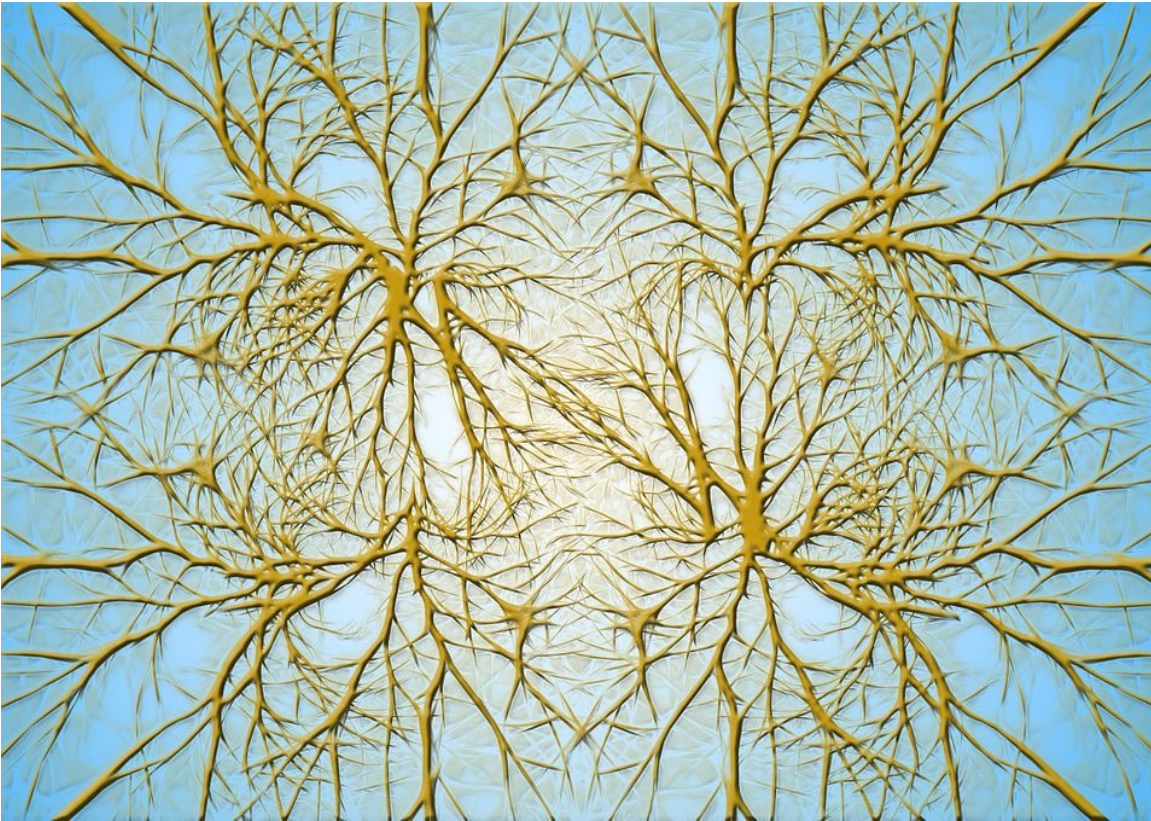
sigmoid activation function

$$\frac{1}{1 + e^{-x}}$$



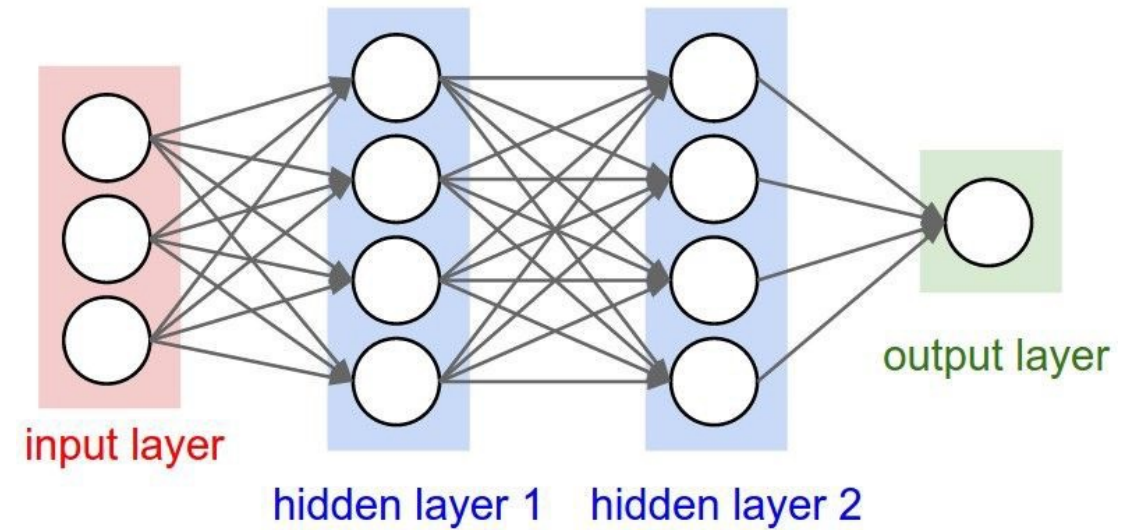


Biological Neurons:  
Complex connectivity patterns



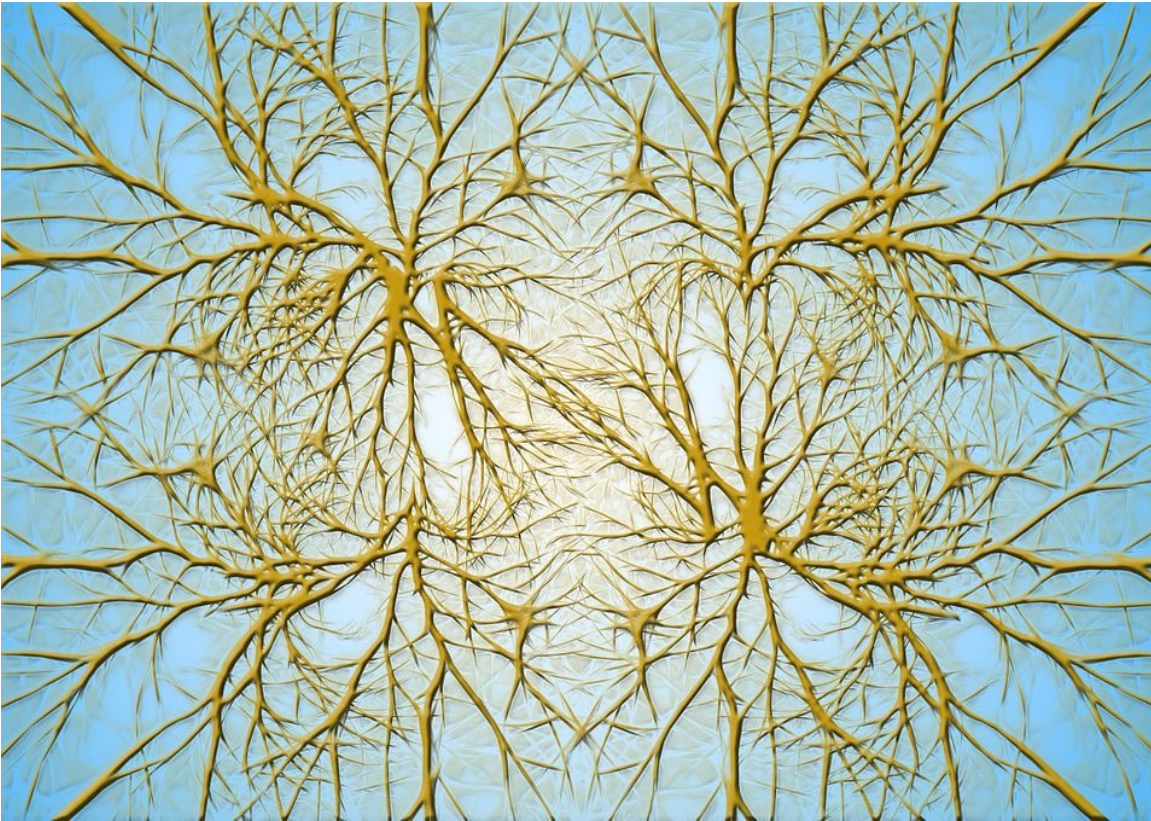
[This image is CC0 Public Domain](#)

Neurons in a neural network:  
Organized into regular layers for  
computational efficiency



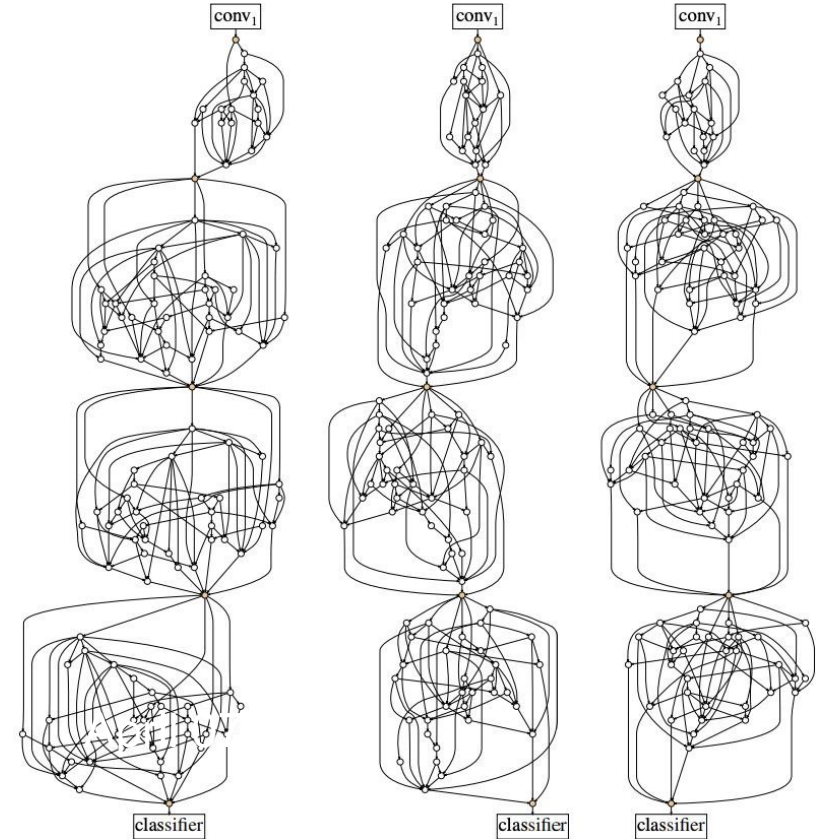


## Biological Neurons: Complex connectivity patterns



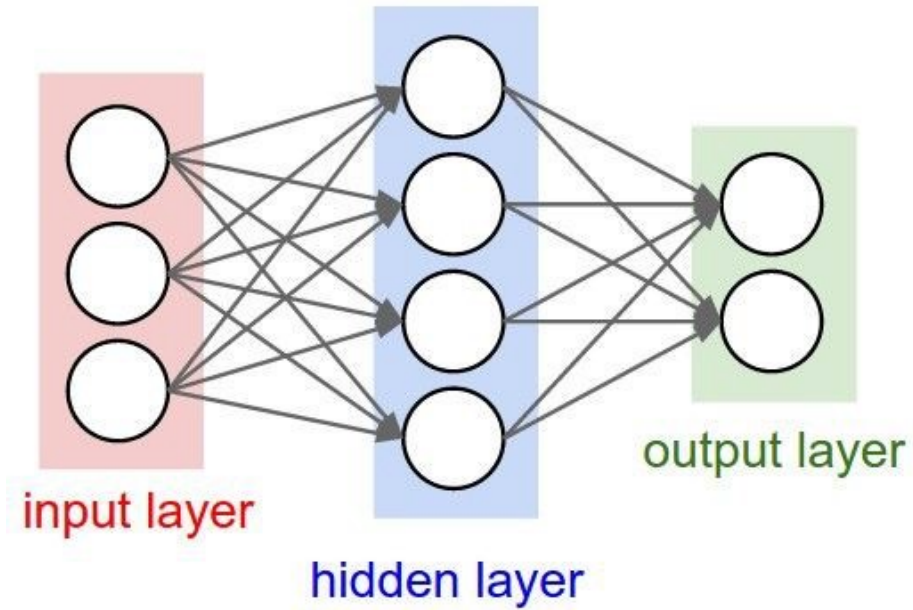
[This image](#) is [CC0 Public Domain](#)

## But neural networks with random connections can work too!

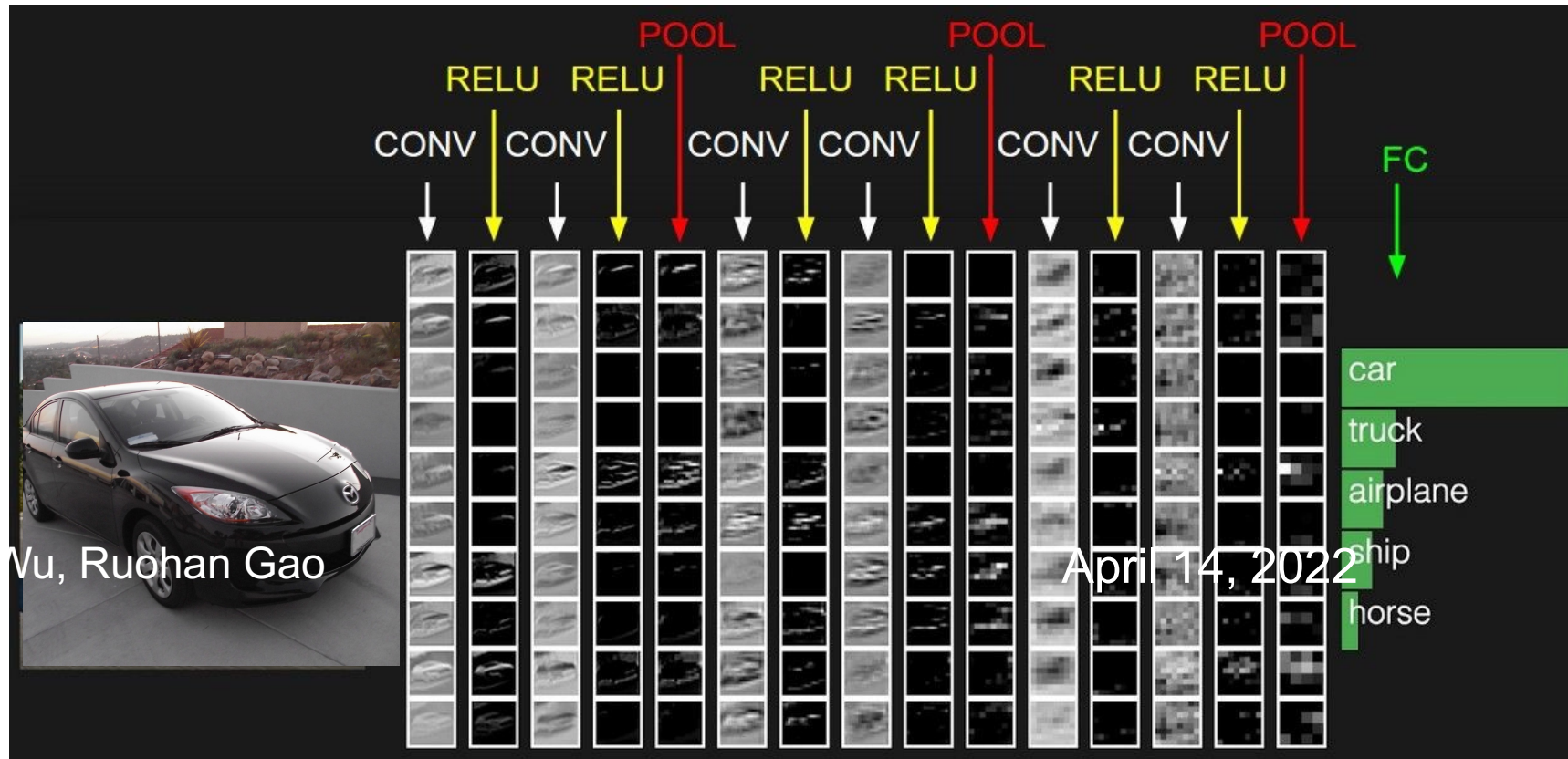


Xie et al, "Exploring Randomly Wired Neural Networks for Image Recognition", arXiv 2019

# Problem for fully-connected networks

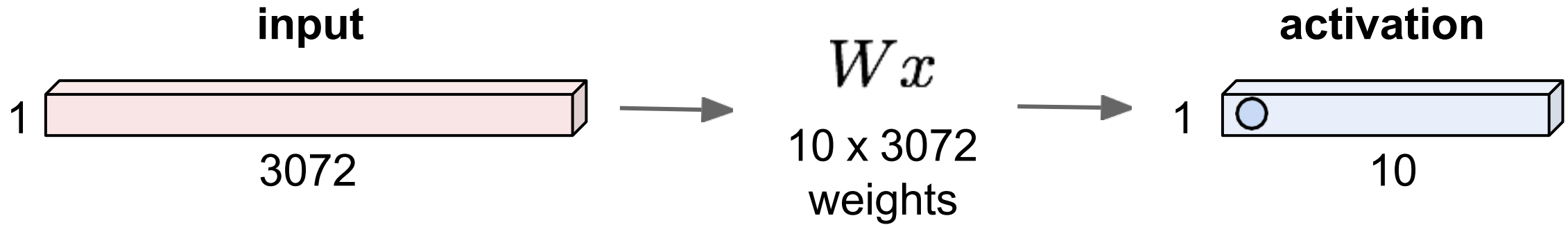


# Convolutional Neural Networks



# Recap: Fully Connected Layer

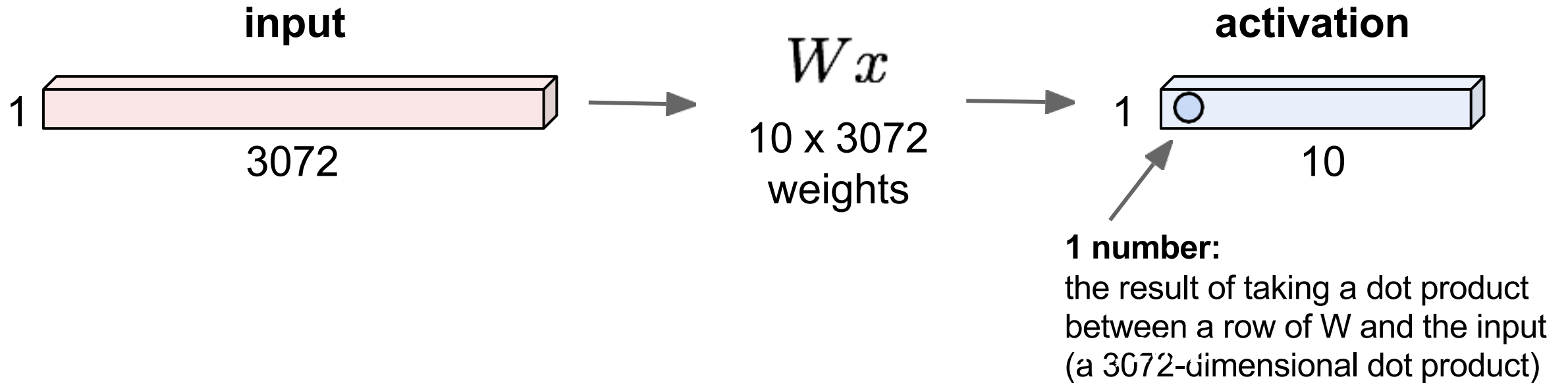
32x32x3 image -> stretch to 3072 x 1





# Fully Connected Layer

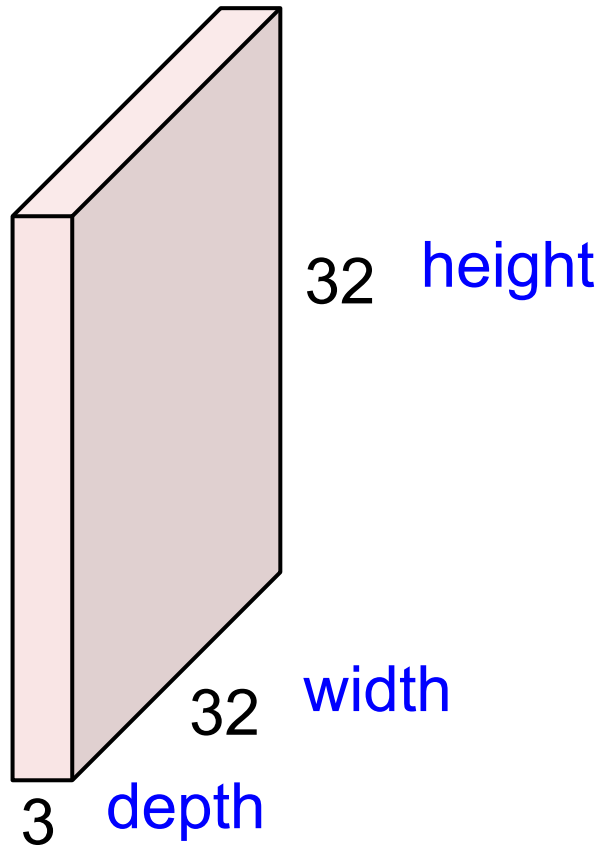
32x32x3 image -> stretch to 3072 x 1





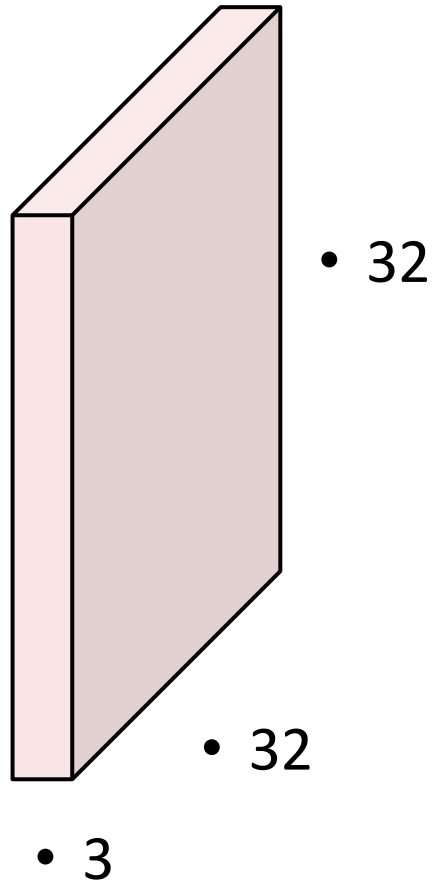
# Convolution Layer

32x32x3 image -> preserve spatial structure



# Convolution Layer

- 32x32x3 image



- 5x5x3 filter

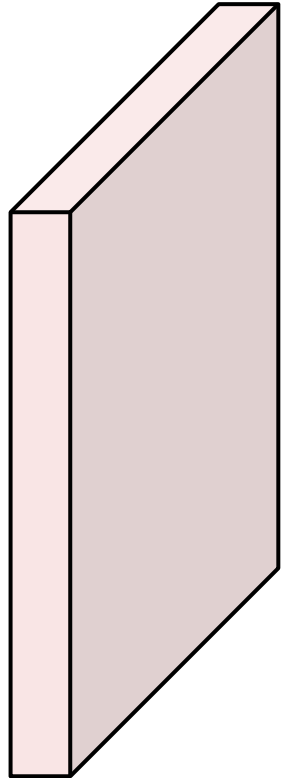


- **Convolve** the filter with the image
- i.e. “slide over the image spatially, computing dot products”

# Convolution Layer

Filters always extend the full depth of the input volume

- 32x32x3 image



• 32

• 32

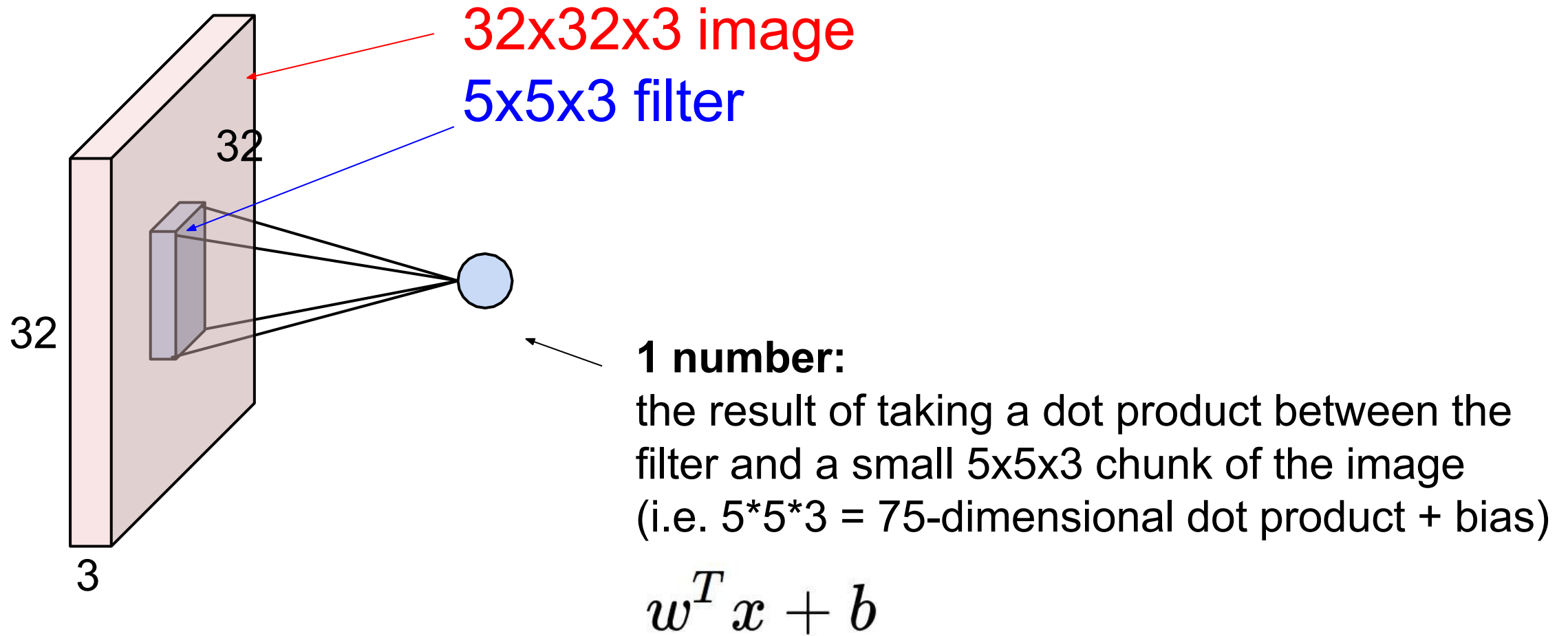
• 3

- 5x5x3 filter

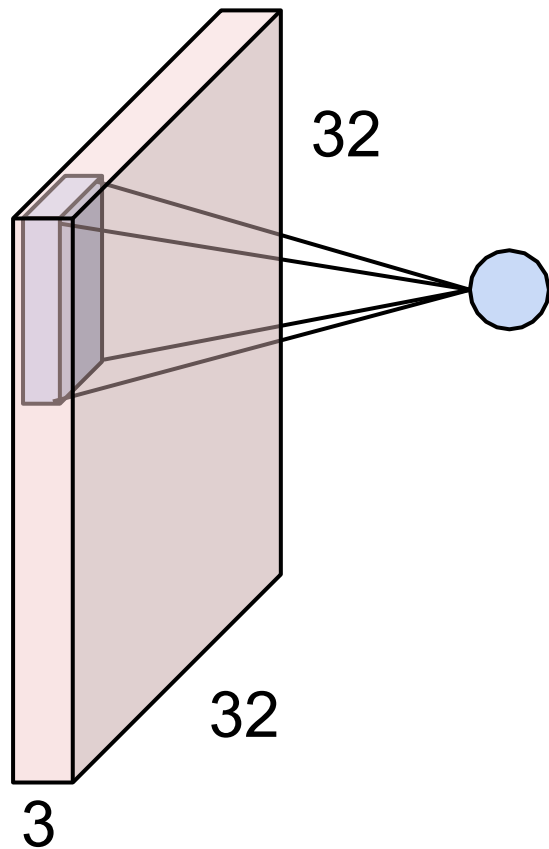


- **Convolve** the filter with the image
- i.e. “slide over the image spatially, computing dot products”

# Convolution Layer

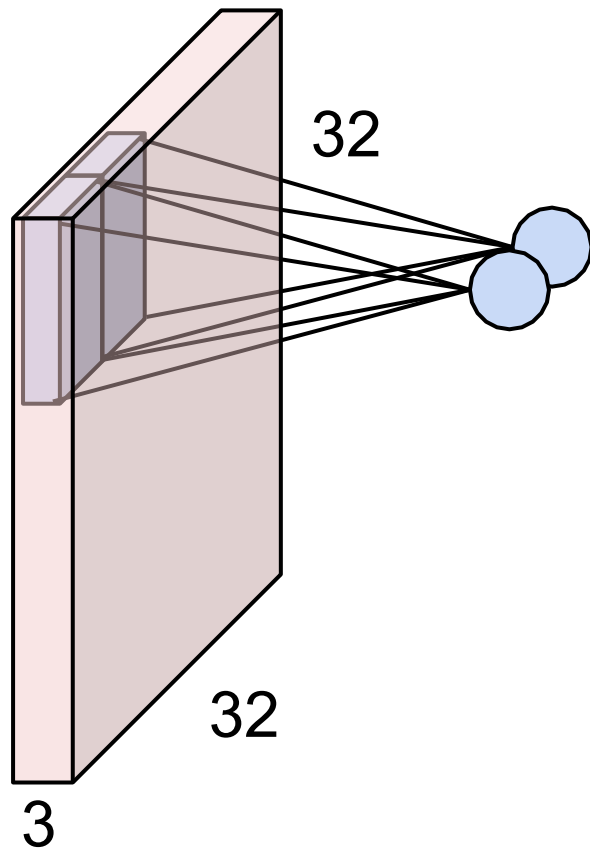


# Convolution Layer

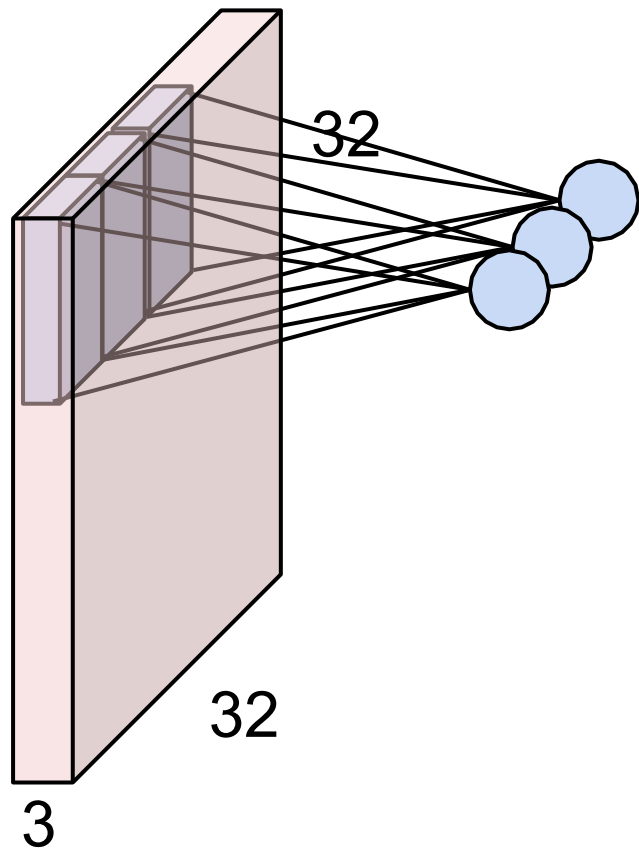




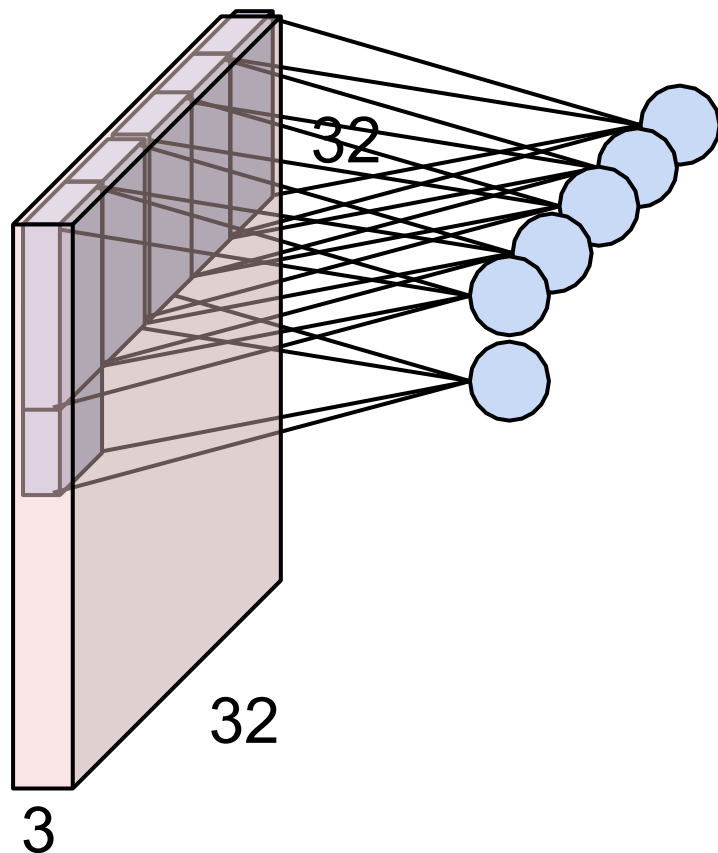
# Convolution Layer



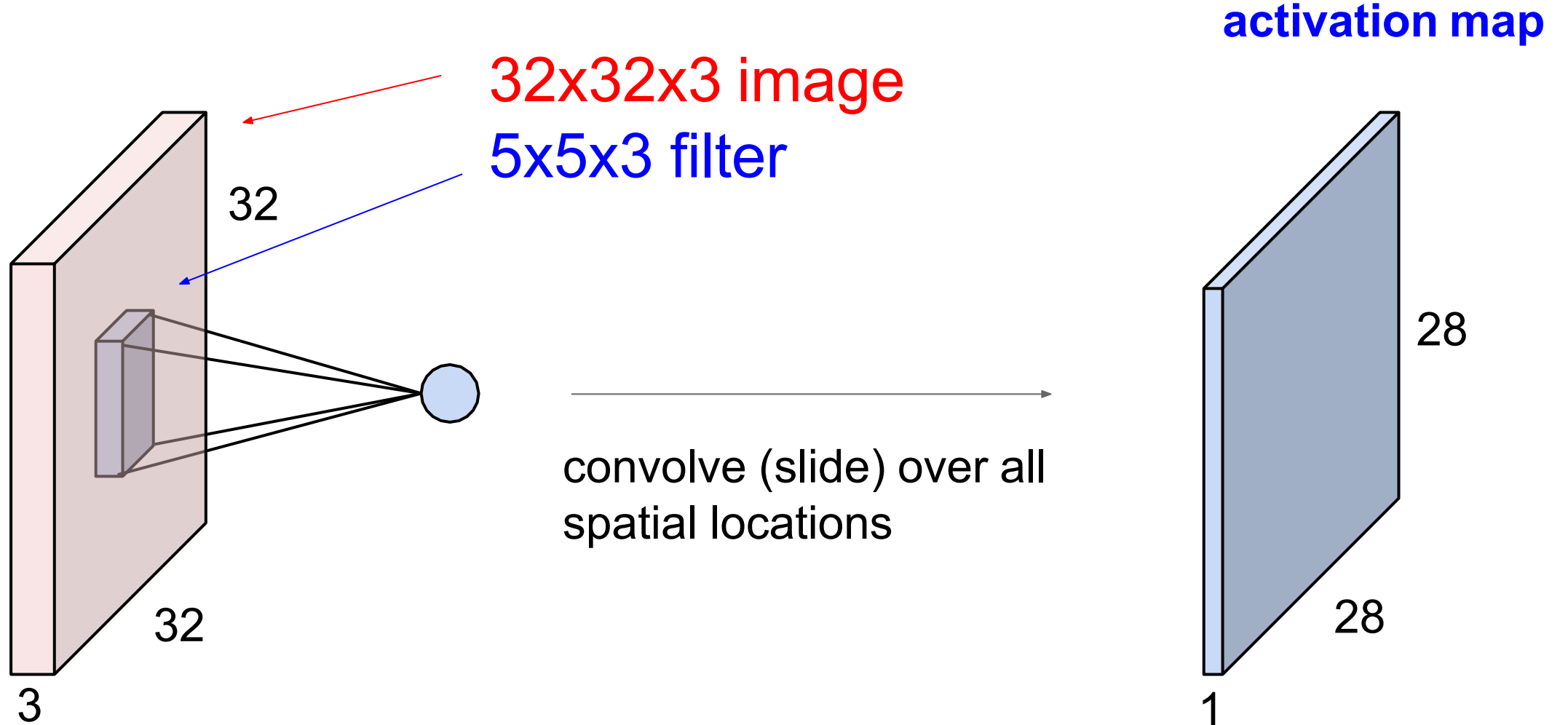
# Convolution Layer



# Convolution Layer



# Convolution Layer



# Convolution Layer

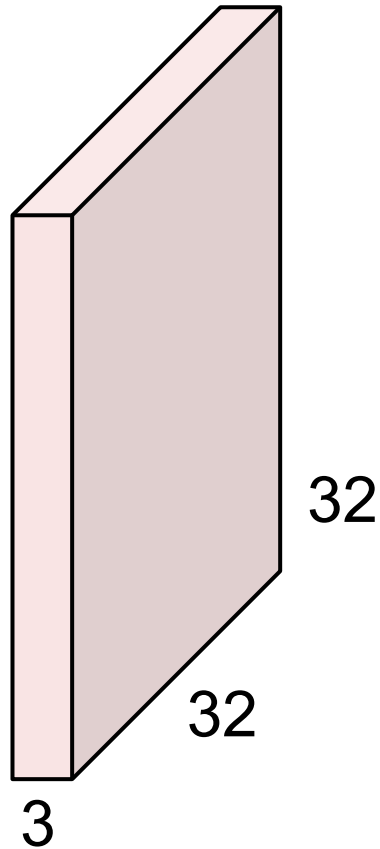
consider a second, **green** filter





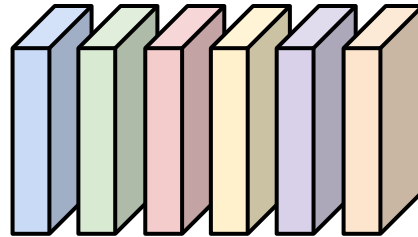
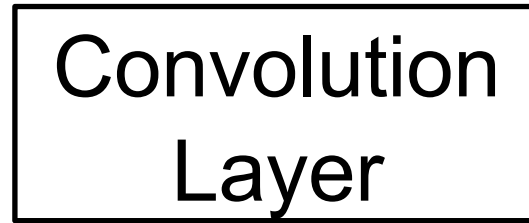
# Convolution Layer

3x32x32 image

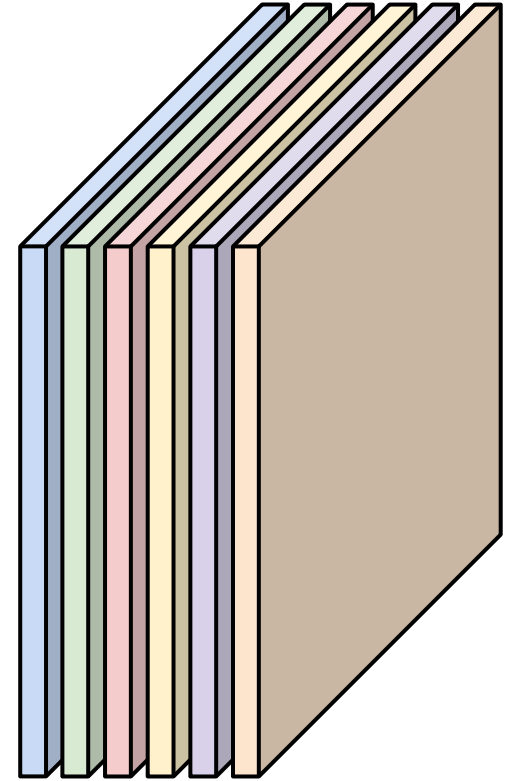


Consider 6 filters,  
each 3x5x5

6x3x5x5  
filters



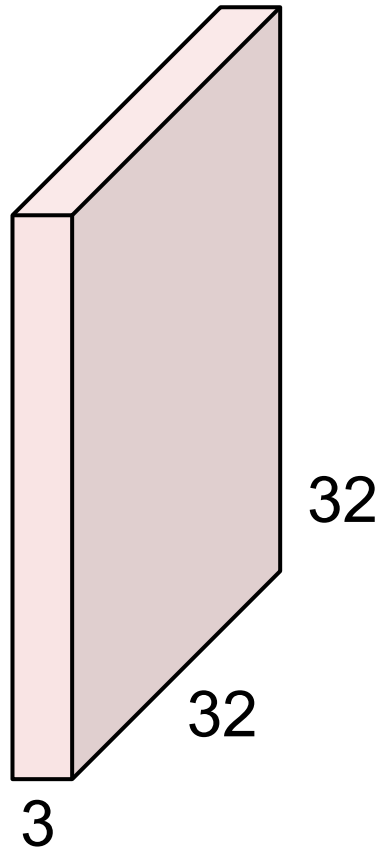
6 activation maps,  
each 1x28x28



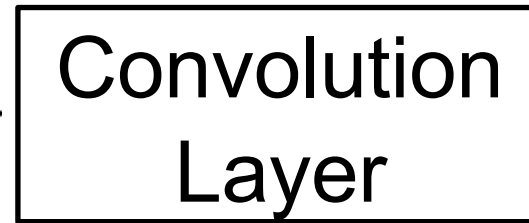
Stack activations to get a  
6x28x28 output image!

# Convolution Layer

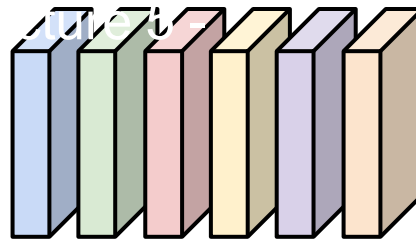
3x32x32 image



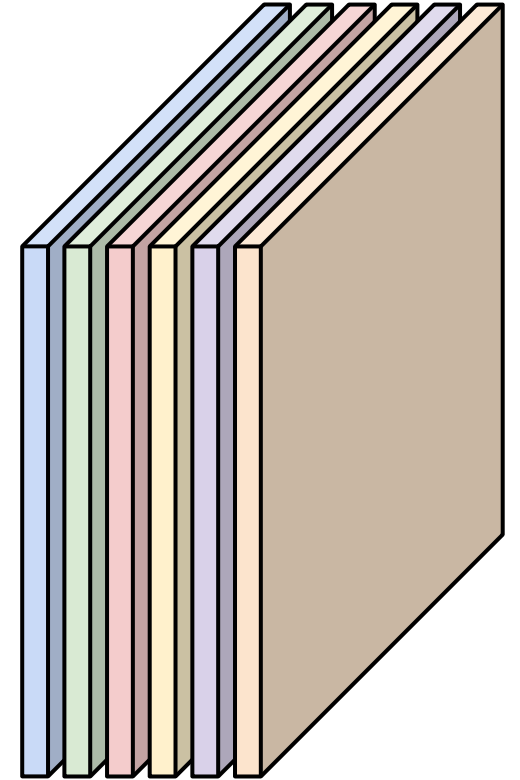
Also 6-dim bias vector:



6x3x5x5  
filters



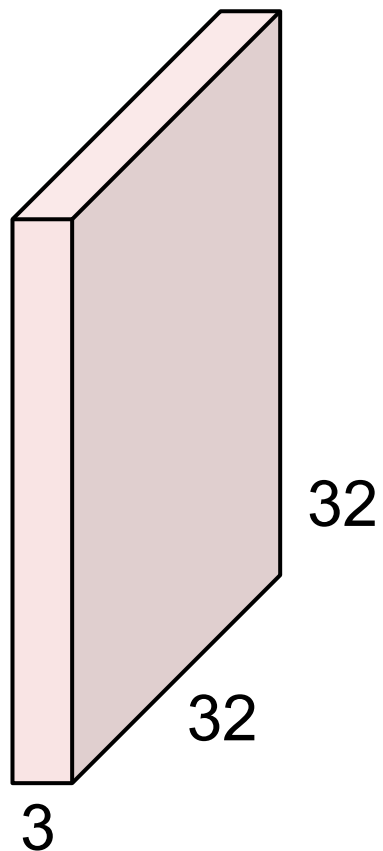
6 activation maps,  
each 1x28x28



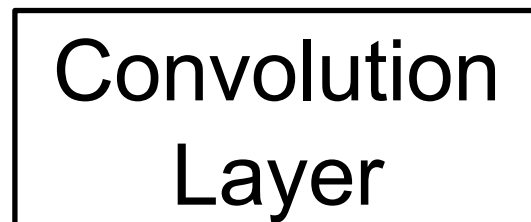
Stack activations to get a  
6x28x28 output image!

# Convolution Layer

3x32x32 image



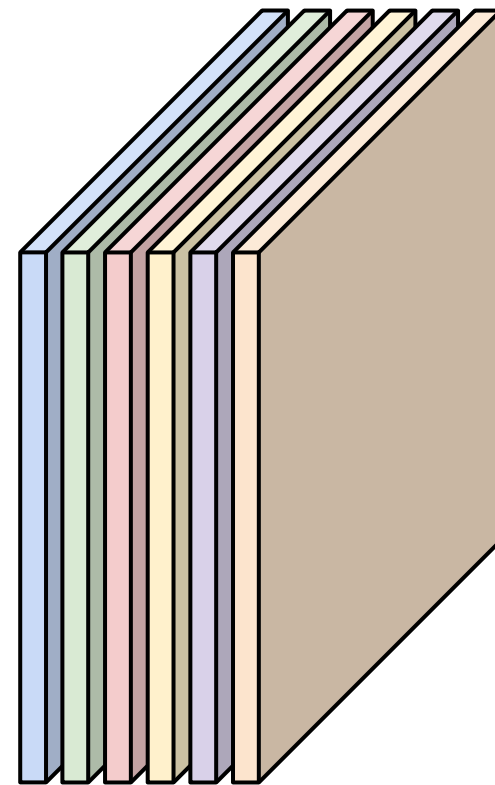
Also 6-dim bias vector:



6x3x5x5  
filters



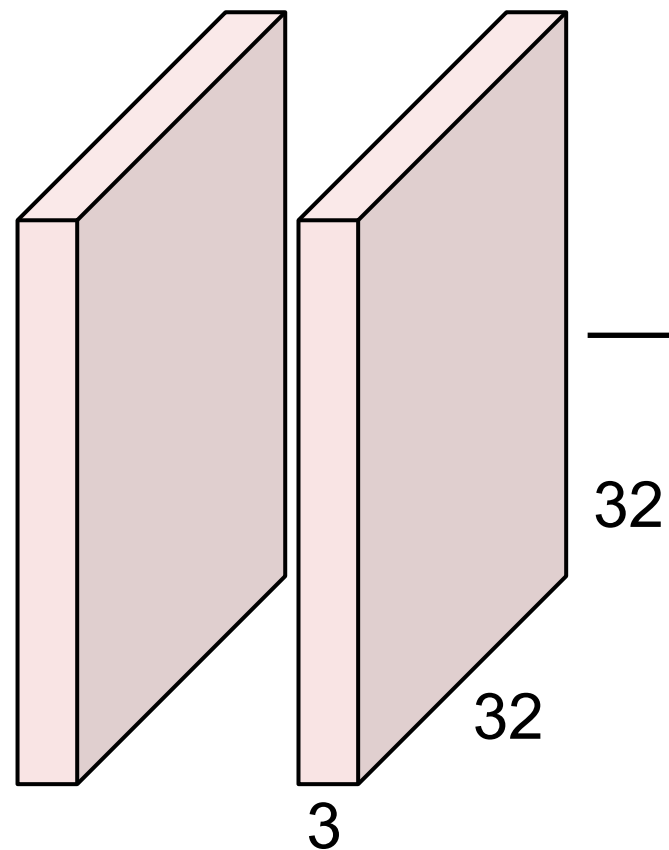
28x28 grid, at each  
point a 6-dim vector



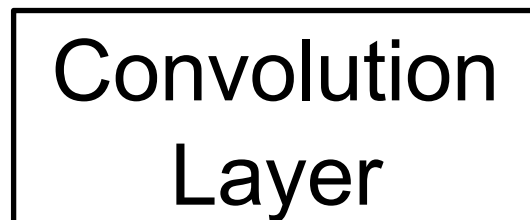
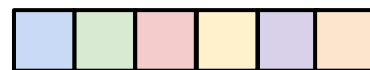
Stack activations to get a  
6x28x28 output image!

# Convolution Layer

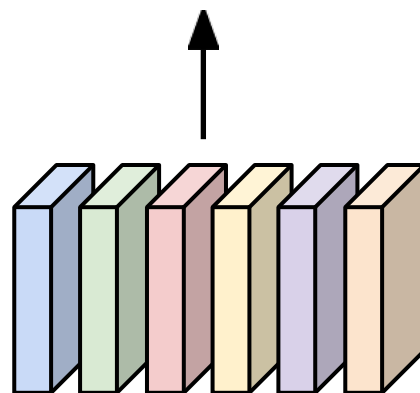
2x3x32x32  
Batch of images



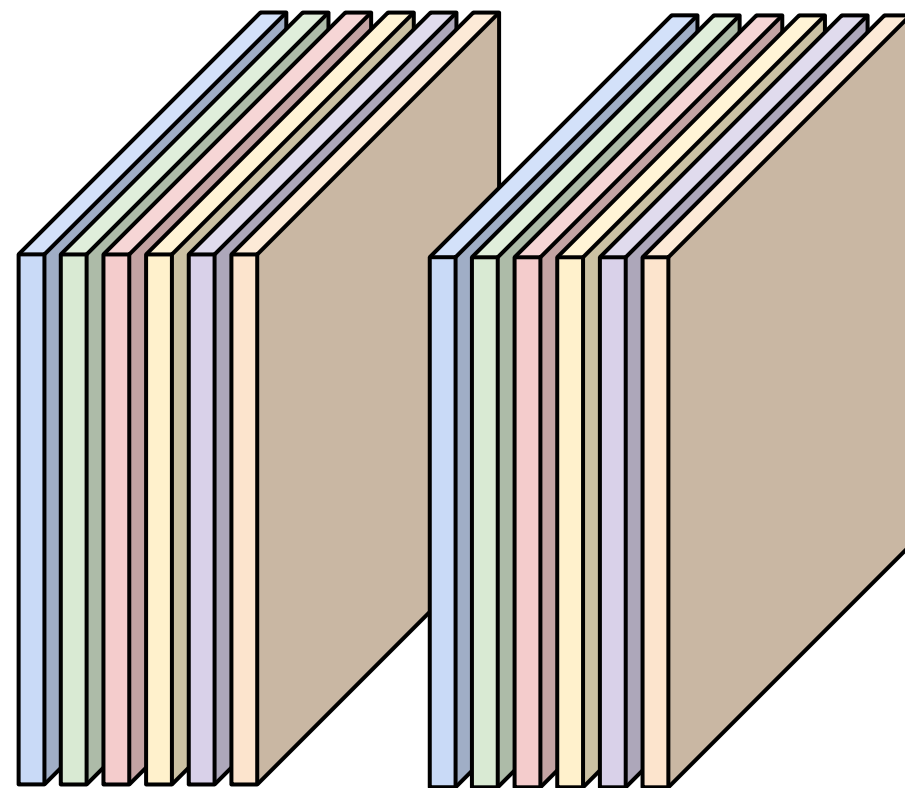
Also 6-dim bias vector:



6x3x5x5  
filters

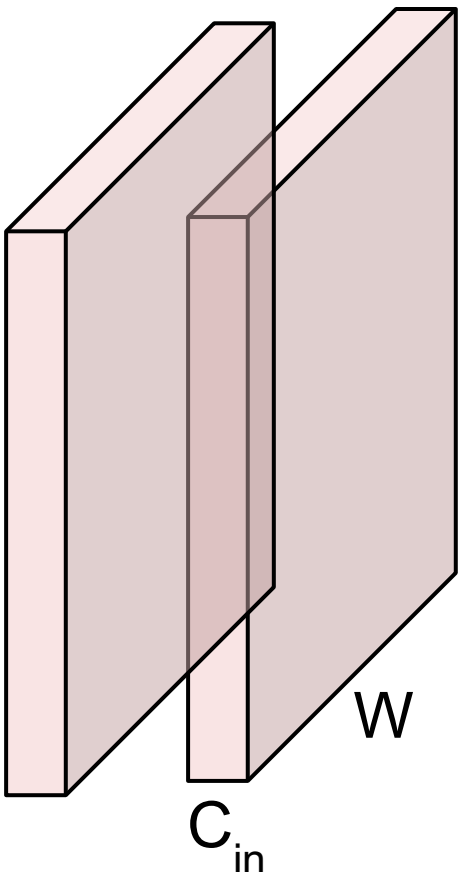


2x6x28x28  
Batch of outputs

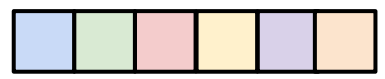


# Convolution Layer

$N \times C_{in} \times H \times W$   
Batch of images

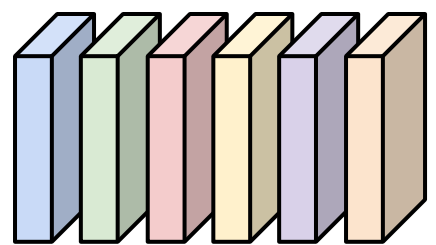


Also  $C_{out}$ -dim bias vector:

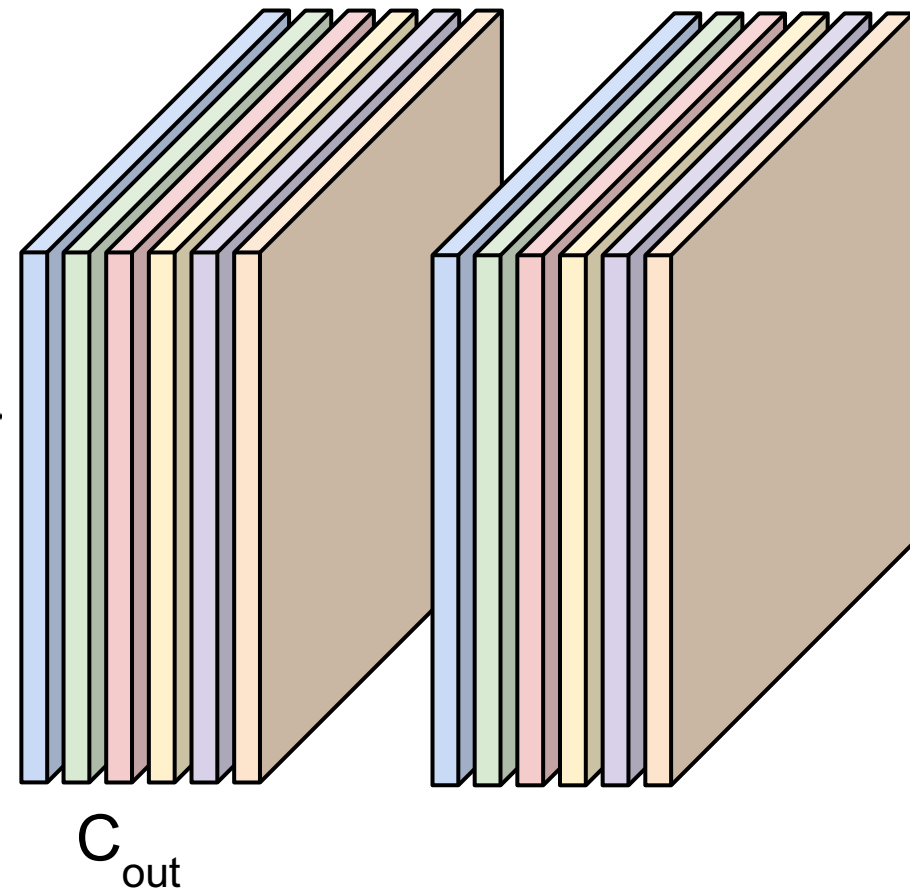


Convolution  
Layer

$C_{out} \times C_{in} \times K_w \times K_h$   
filters



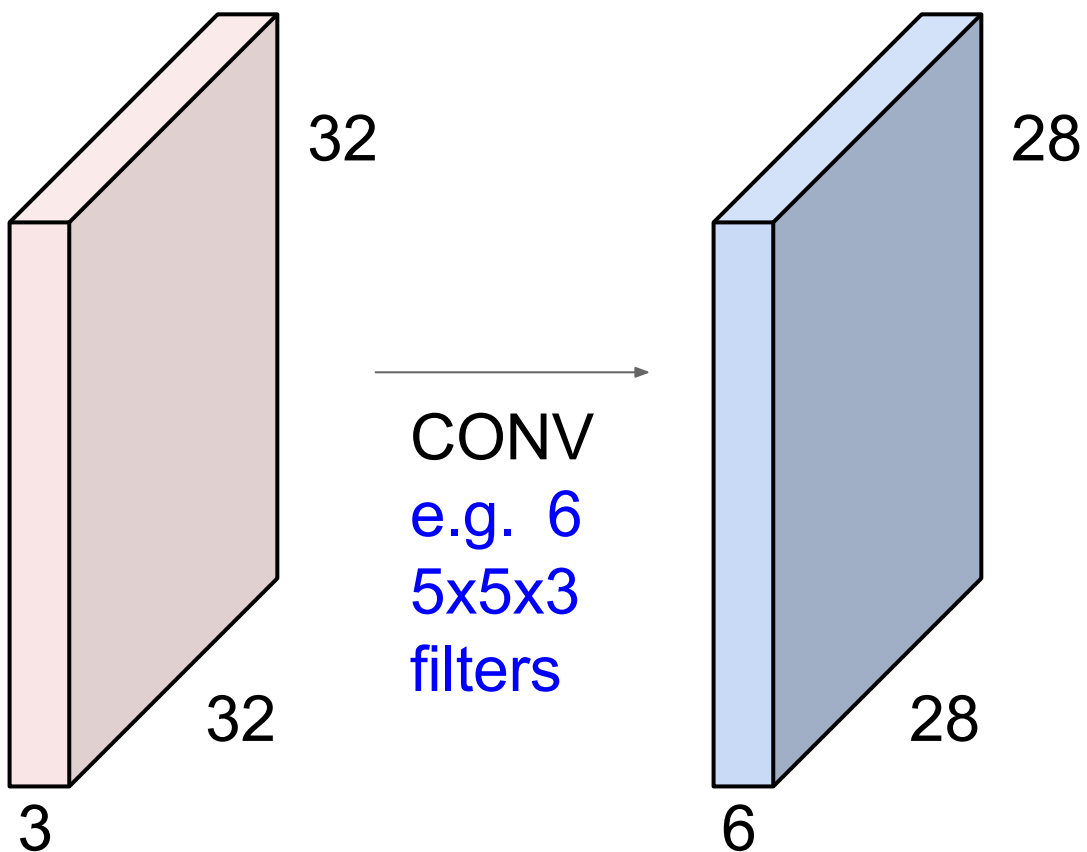
$N \times C_{out} \times H' \times W'$   
Batch of outputs



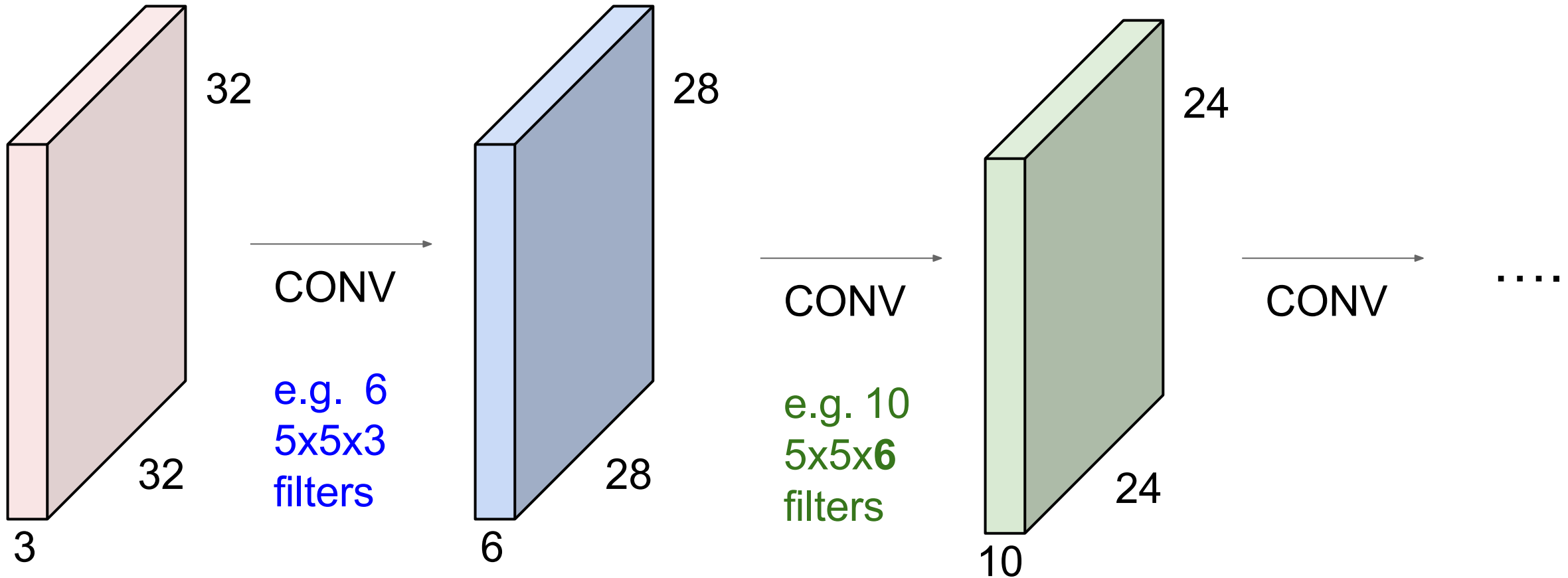
Slide inspiration: Justin Johnson



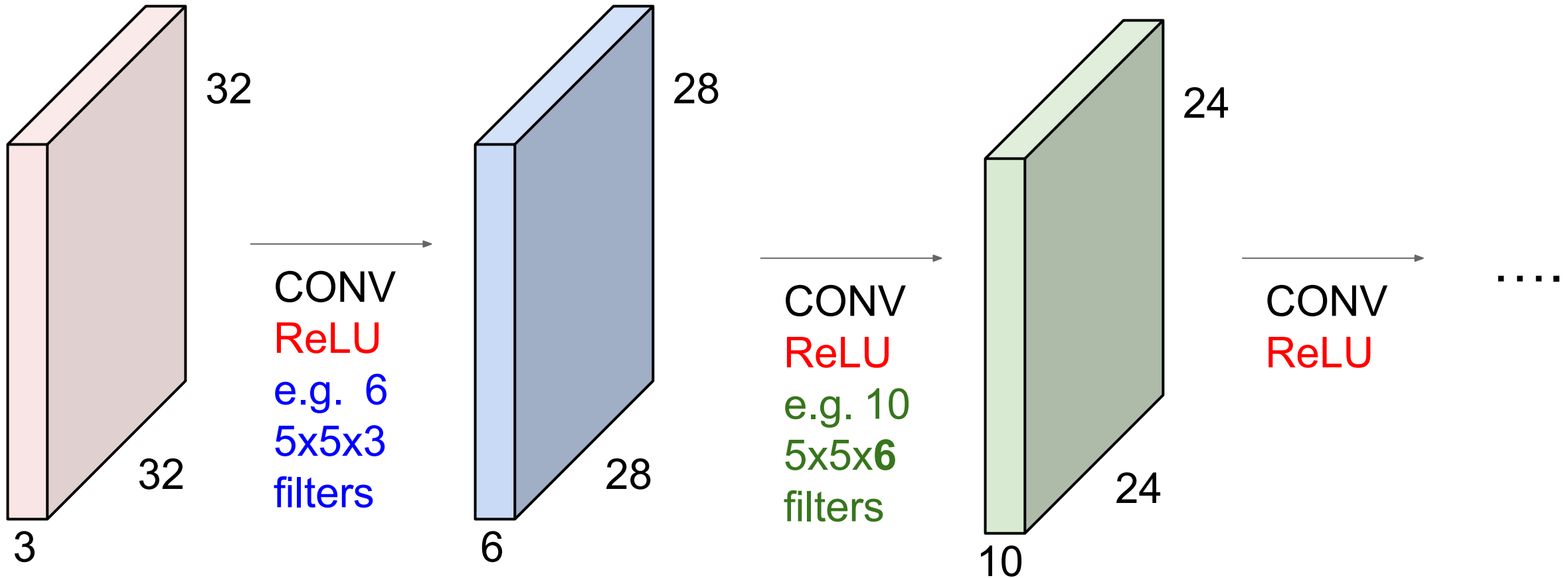
## Preview: ConvNet is a sequence of Convolution Layers



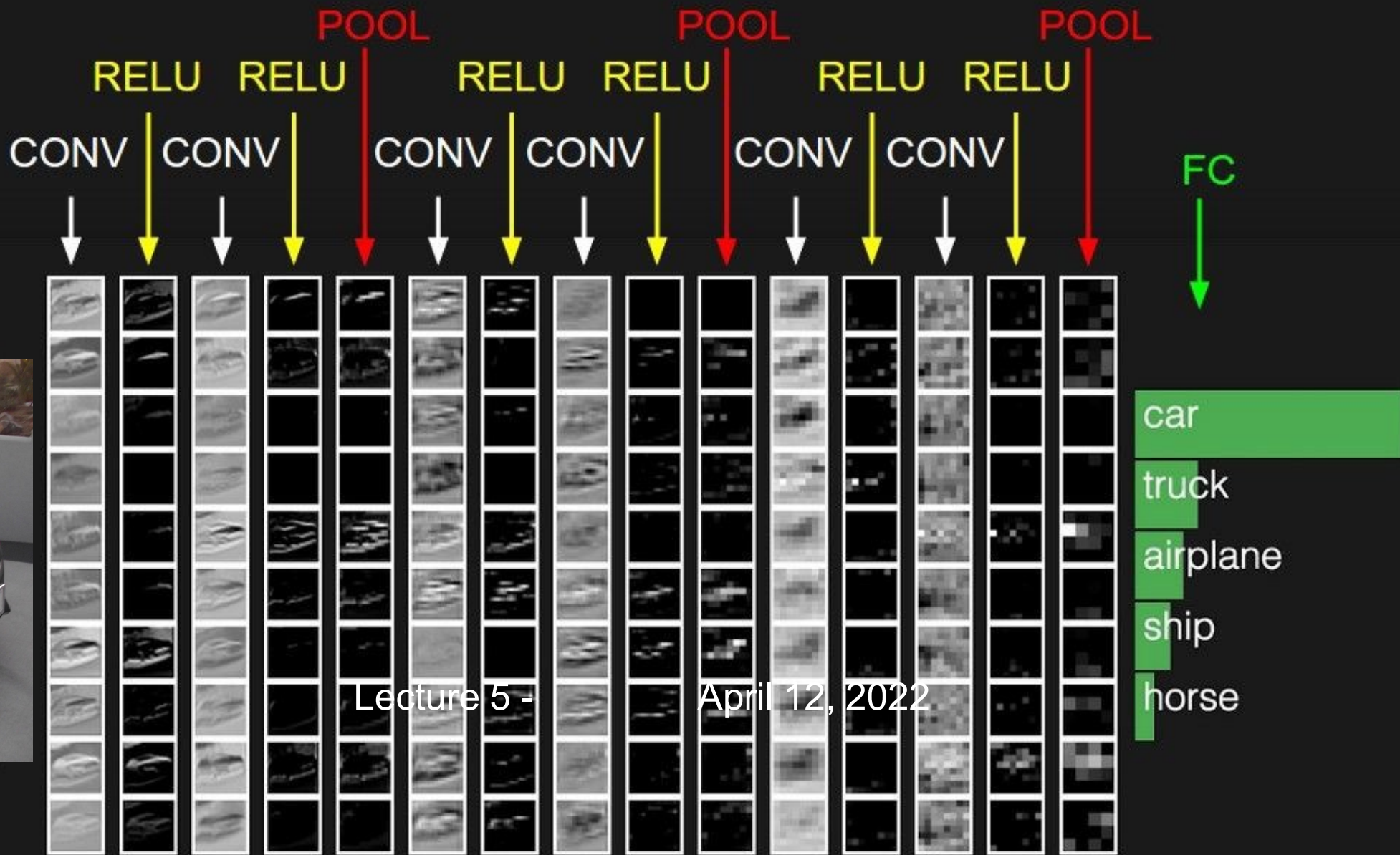
## Preview: ConvNet is a sequence of Convolution Layers



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



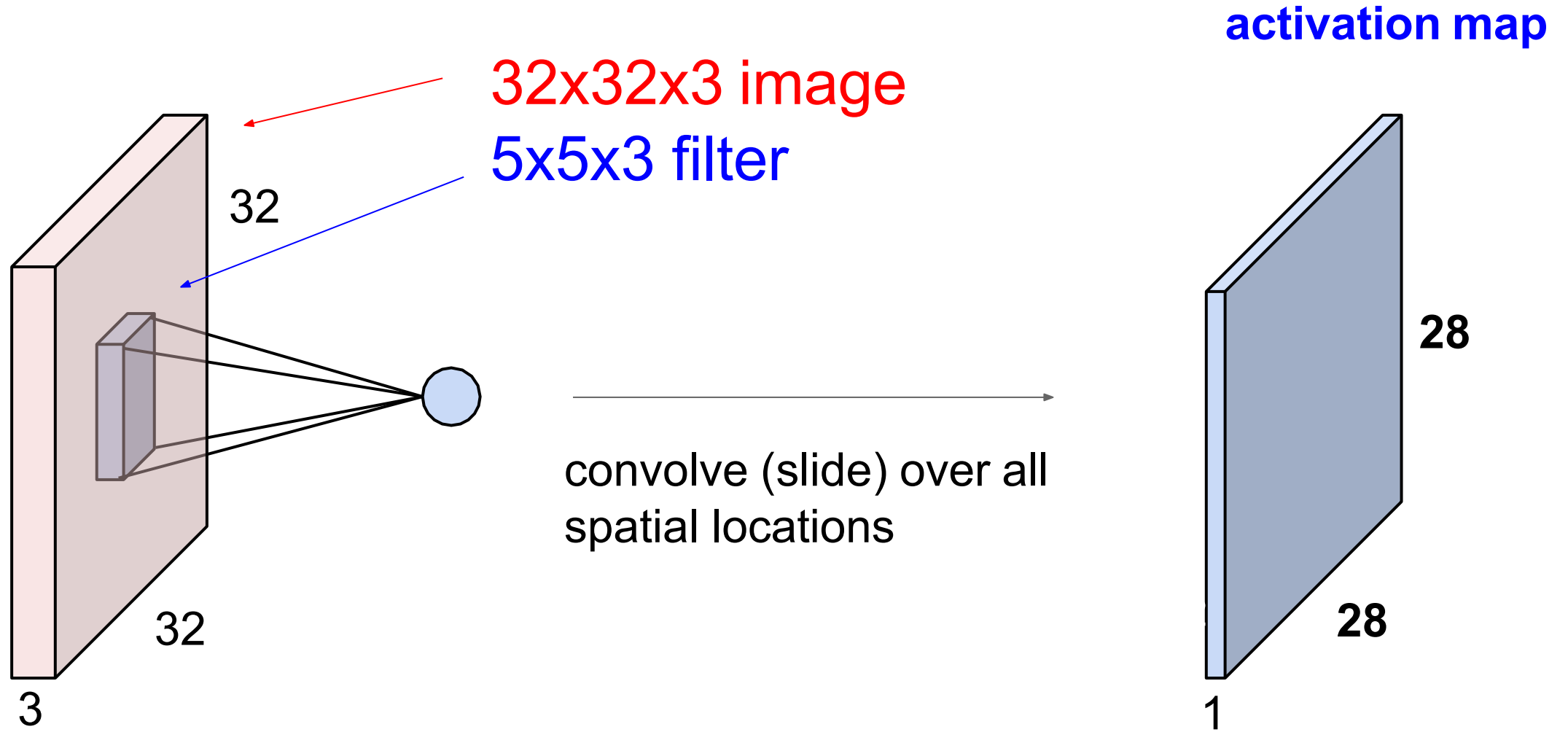
preview:



Lecture 5 -

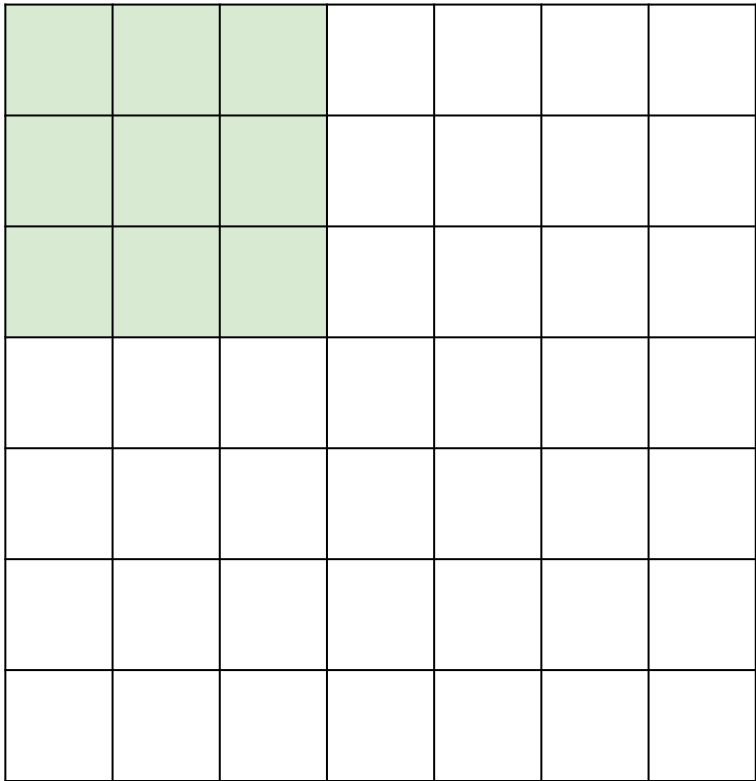
April 12, 2022

A closer look at spatial dimensions:



A closer look at spatial dimensions:

7



7x7 input (spatially)  
assume 3x3 filter

7



A closer look at spatial dimensions:

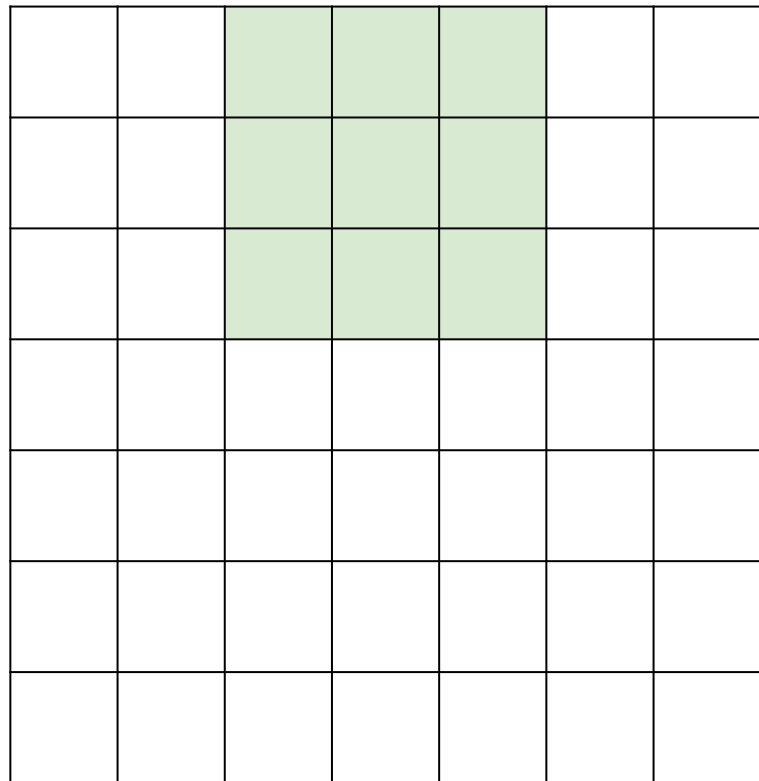
7


7

7x7 input (spatially)  
assume 3x3 filter

A closer look at spatial dimensions:

7

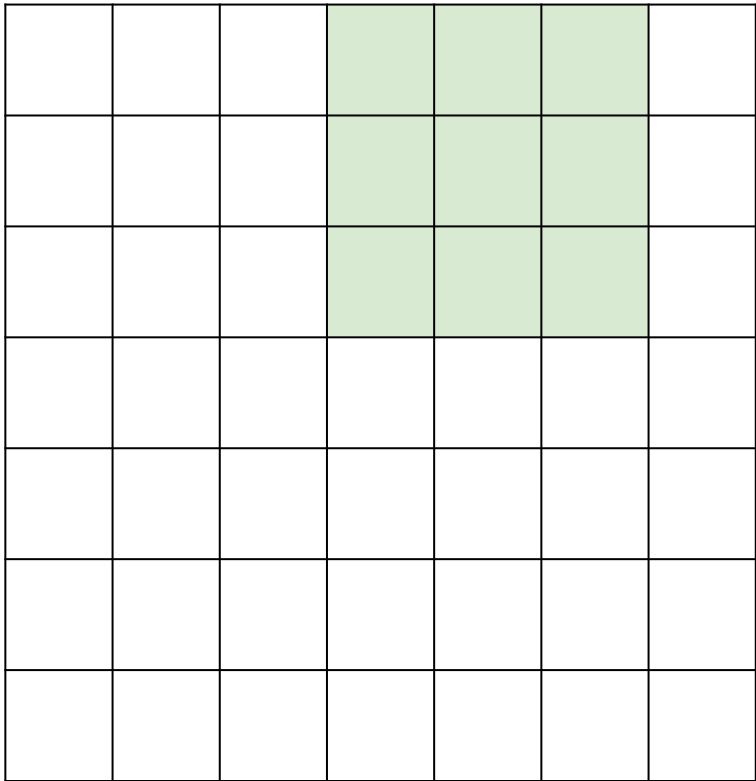


7x7 input (spatially)  
assume 3x3 filter

7

A closer look at spatial dimensions:

7



7x7 input (spatially)  
assume 3x3 filter

7

A closer look at spatial dimensions:

7

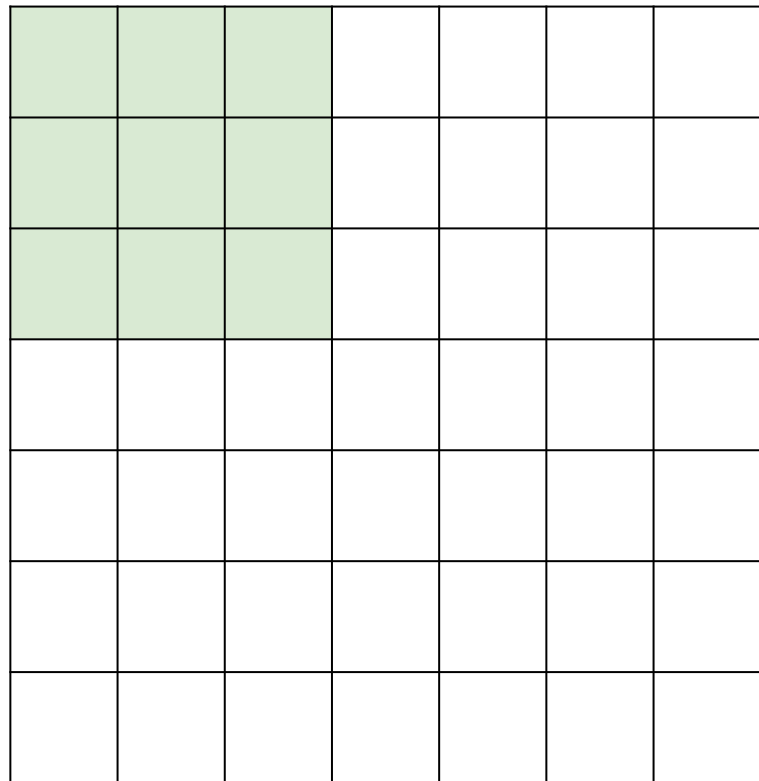

7

7x7 input (spatially)  
assume 3x3 filter

**=> 5x5 output**

A closer look at spatial dimensions:

7

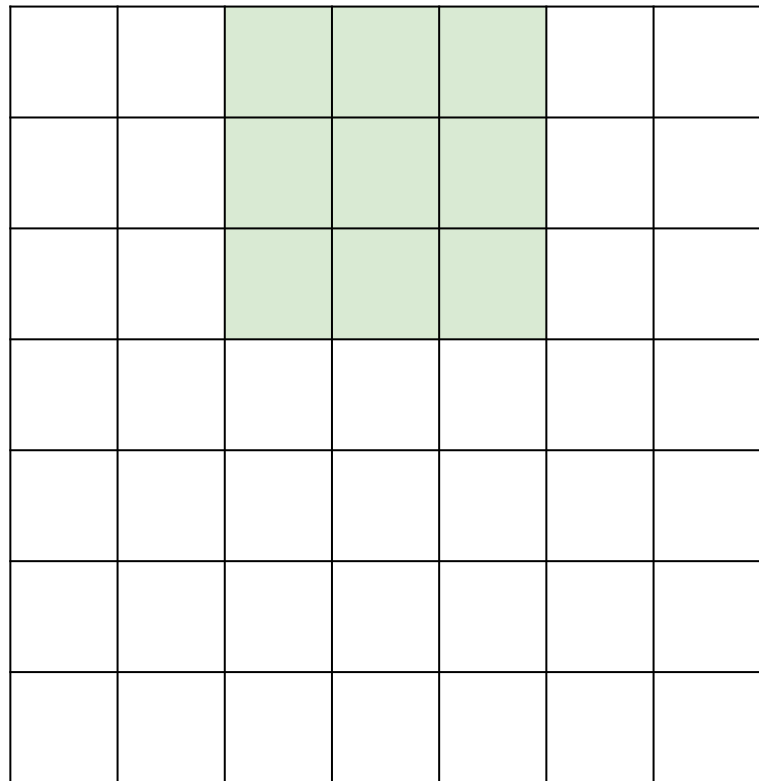


7

7x7 input (spatially)  
assume 3x3 filter  
applied **with stride 2**

A closer look at spatial dimensions:

7



7

7x7 input (spatially)  
assume 3x3 filter  
applied **with stride 2**



A closer look at spatial dimensions:

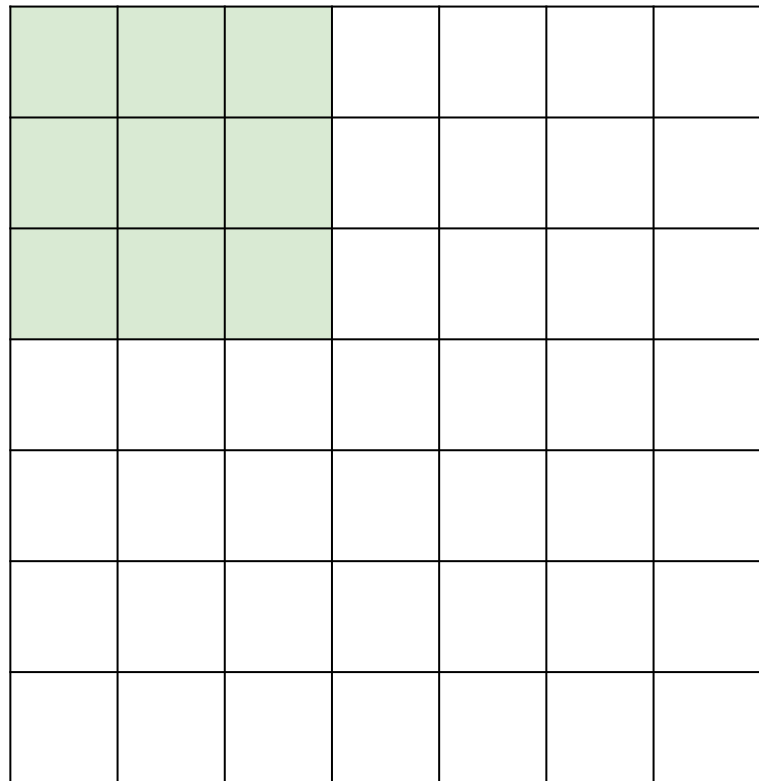
7


7

7x7 input (spatially)  
assume 3x3 filter  
applied **with stride 2**  
**=> 3x3 output!**

A closer look at spatial dimensions:

7

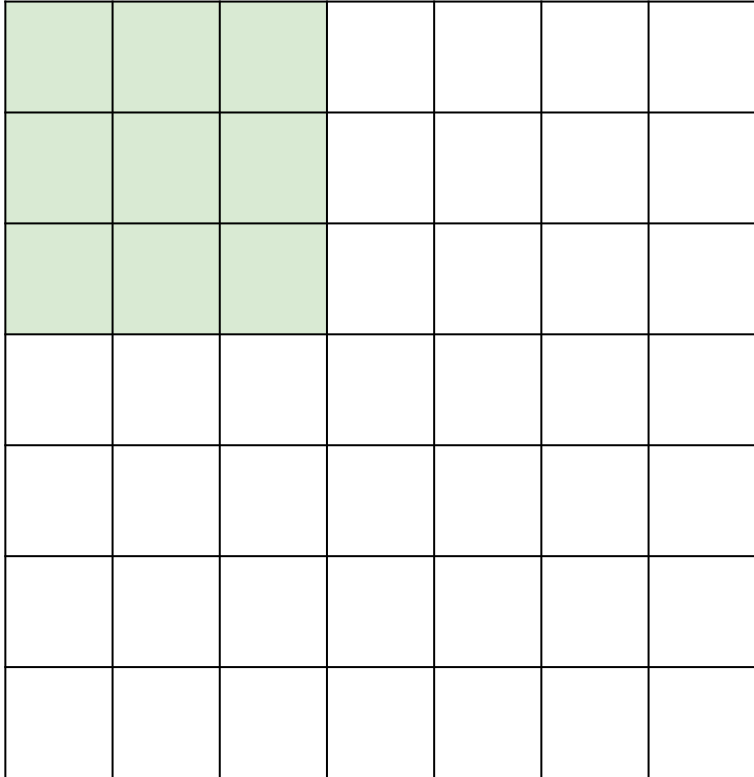


7

7x7 input (spatially)  
assume 3x3 filter  
applied **with stride 3?**

A closer look at spatial dimensions:

7

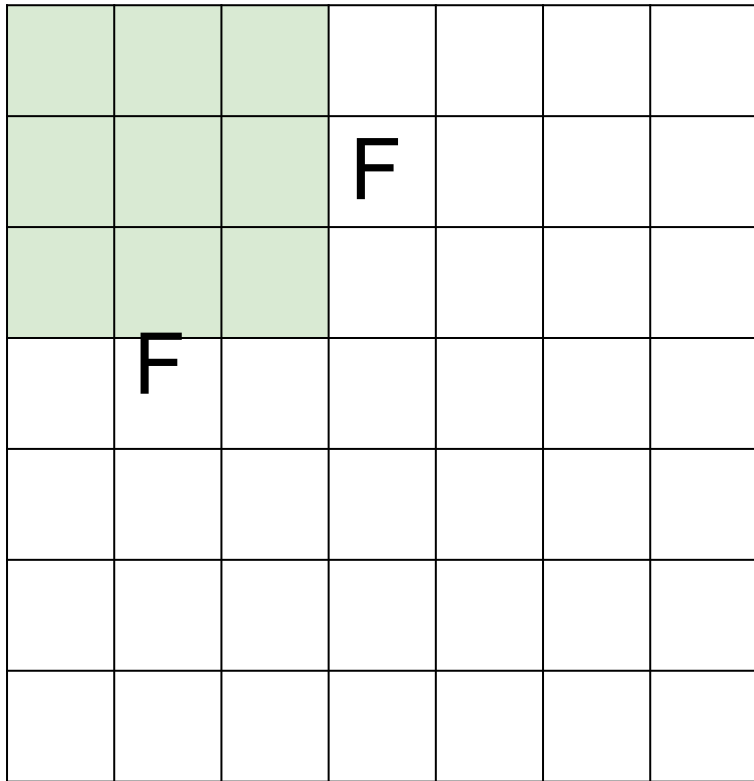


7

7x7 input (spatially)  
assume 3x3 filter  
applied **with stride 3?**

**doesn't fit!**  
cannot apply 3x3 filter on  
7x7 input with stride 3.

N



Output size:

$$(N - F) / \text{stride} + 1$$

e.g.  $N = 7, F = 3$ :

$$\text{stride } 1 \Rightarrow (7 - 3) / 1 + 1 = 5$$

$$\text{stride } 2 \Rightarrow (7 - 3) / 2 + 1 = 3$$

$$\text{stride } 3 \Rightarrow (7 - 3) / 3 + 1 = 2.33 \text{ :}\backslash$$

# In practice: Common to zero pad the border

0	0	0	0	0	0			
0								
0								
0								
0								

e.g. input 7x7

**3x3** filter, applied with **stride 1**

**pad with 1 pixel** border => what is the output?

(recall:)

$$(N - F) / \text{stride} + 1$$

# In practice: Common to zero pad the border

0	0	0	0	0	0			
0								
0								
0								
0								

e.g. input 7x7

**3x3** filter, applied with **stride 1**

**pad with 1 pixel** border => what is the output?

**7x7 output!**

(recall:)

$$(N + 2P - F) / \text{stride} + 1$$



# In practice: Common to zero pad the border

0	0	0	0	0	0			
0								
0								
0								
0								

e.g. input 7x7

**3x3** filter, applied with **stride 1**

**pad with 1 pixel** border => what is the output?

**7x7 output!**

in general, common to see CONV layers with stride 1, filters of size  $F \times F$ , and zero-padding with  $(F-1)/2$ . (will preserve size spatially)

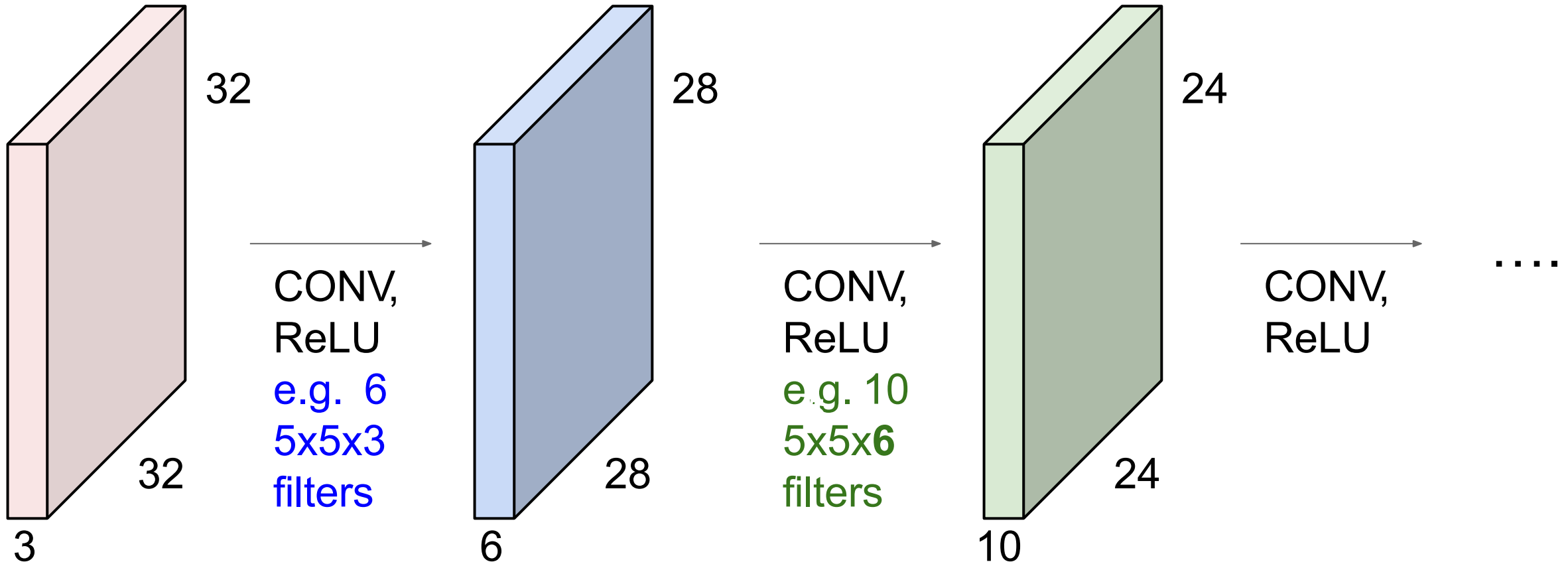
e.g.  $F = 3 \Rightarrow$  zero pad with 1

$F = 5 \Rightarrow$  zero pad with 2

$F = 7 \Rightarrow$  zero pad with 3

## Remember back to...

E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially! (32 -> 28 -> 24 ...). Shrinking too fast is not good, doesn't work well.

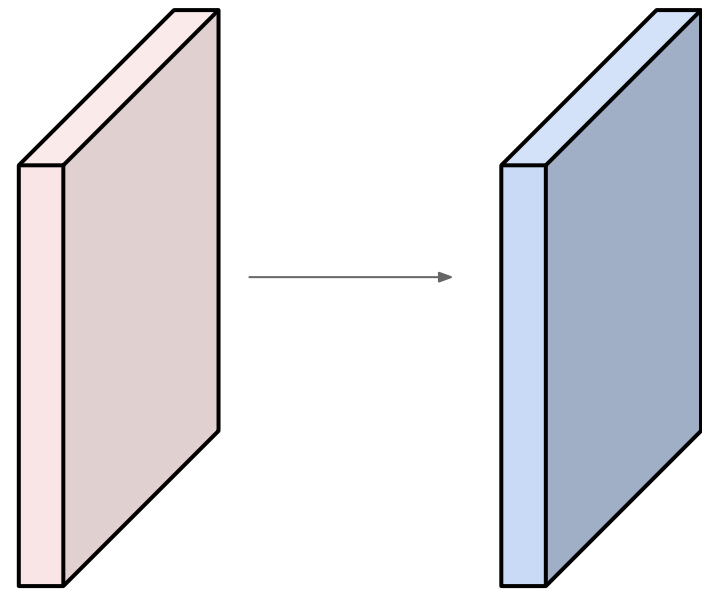


Examples time:

Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

Output volume size: ?



Examples time:

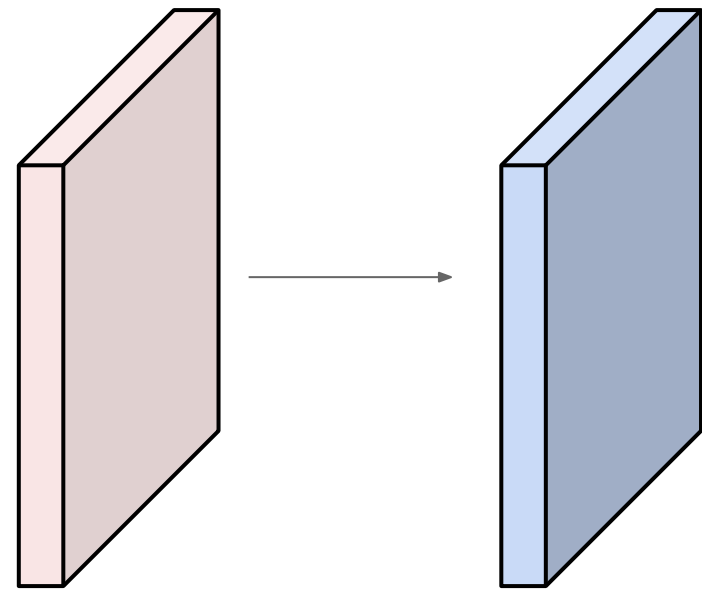
Input volume: **32x32x3**

**10** **5x5** filters with stride **1**, pad **2**

Output volume size:

$(32 + 2 * 2 - 5) / 1 + 1 = 32$  spatially, so

**32x32x10**

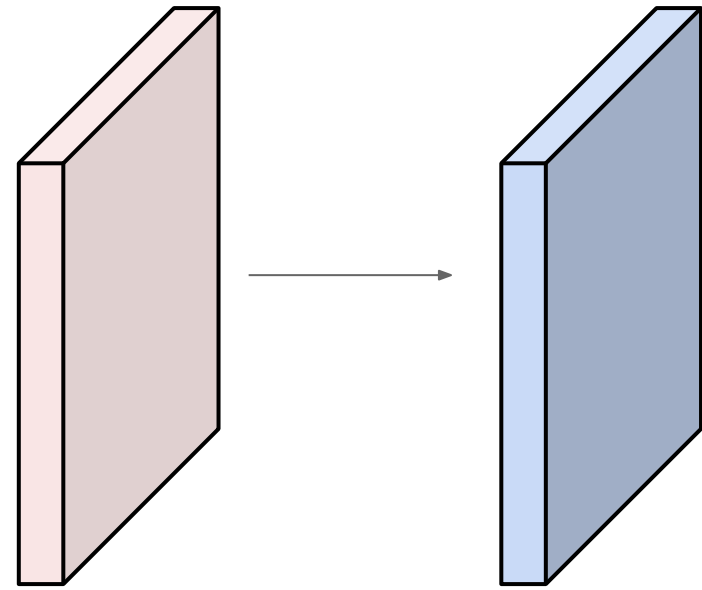


Examples time:

Input volume: **32x32x3**

10 5x5 filters with stride 1, pad 2

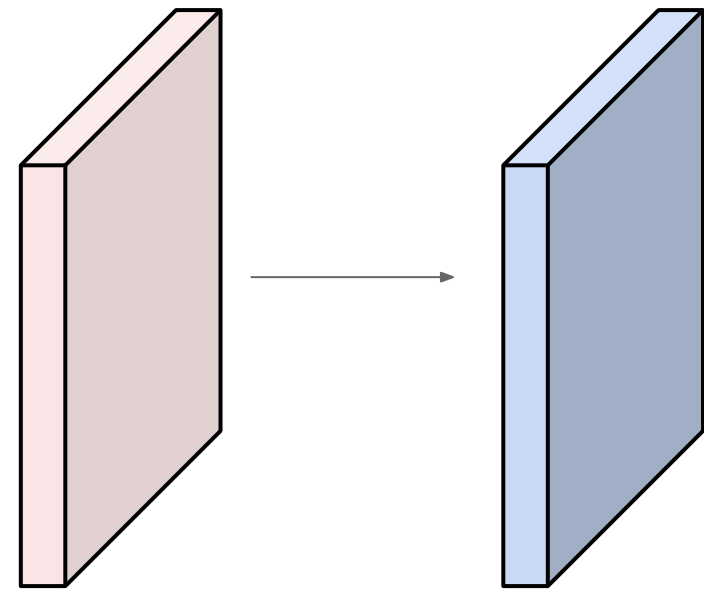
Number of parameters in this layer?



Examples time:

Input volume: **32x32x3**

**10** **5x5** filters with stride 1, pad 2



Number of parameters in this layer?

each filter has  $5*5*3 + 1 = 76$  params (+1 for bias)

=>  $76*10 = 760$

# Convolution layer: summary

Let's assume input is  $W_1 \times H_1 \times C$

Conv layer needs 4 hyperparameters:

- Number of filters **K**
- The filter size **F**
- The stride **S**
- The zero padding **P**

This will produce an output of  $W_2 \times H_2 \times K$

where:

- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$

Number of parameters:  $F^2CK$  and  $K$  biases



# Convolution layer: summary

## Common settings:

Let's assume input is  $W_1 \times H_1 \times C$

Conv layer needs 4 hyperparameters:

- Number of filters **K**
- The filter size **F**
- The stride **S**
- The zero padding **P**

**K** = (powers of 2, e.g. 32, 64, 128, 512)

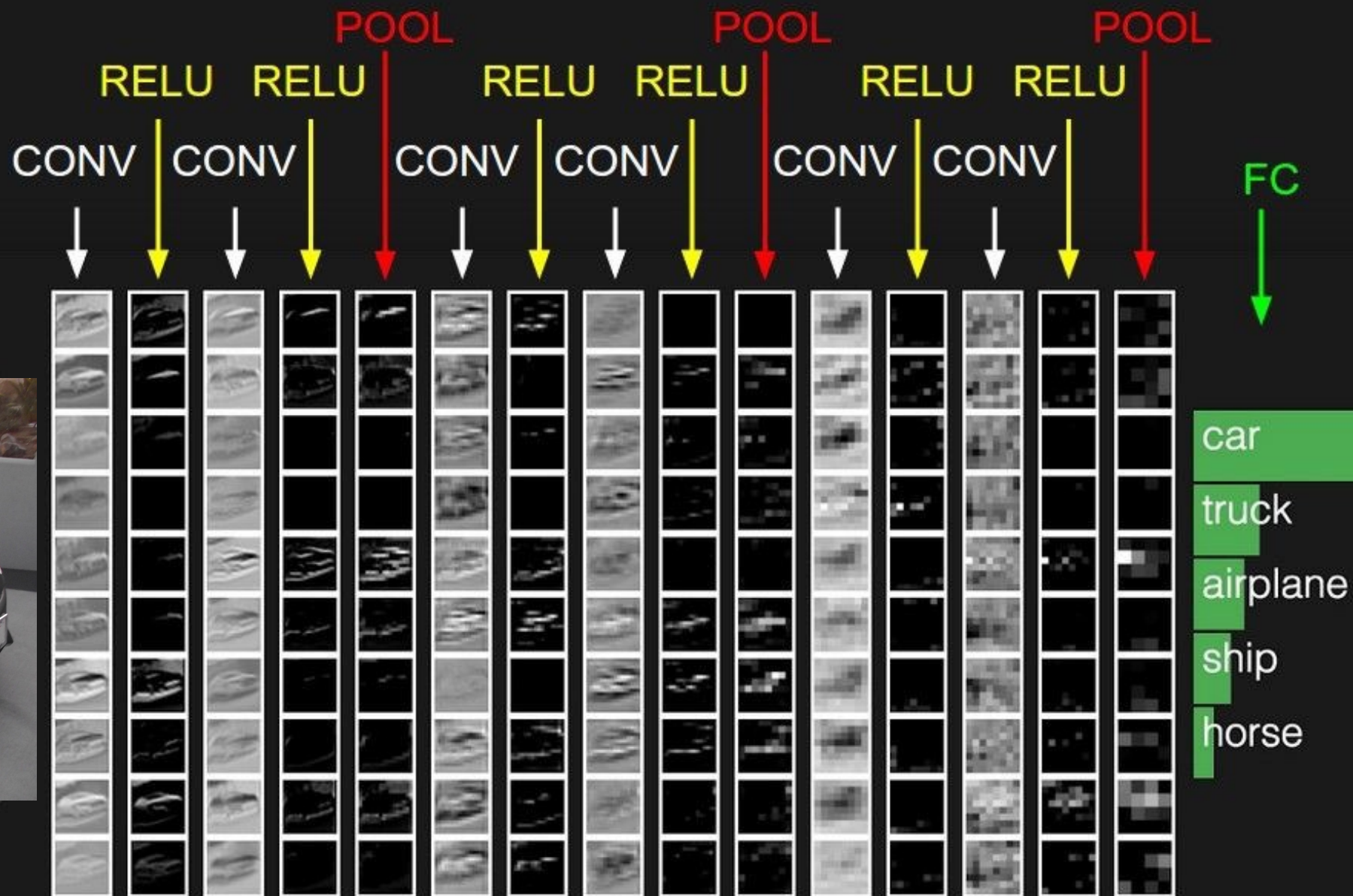
- **F** = 3, **S** = 1, **P** = 1
- **F** = 5, **S** = 1, **P** = 2
- **F** = 5, **S** = 2, **P** = ? (whatever fits)
- **F** = 1, **S** = 1, **P** = 0

This will produce an output of  $W_2 \times H_2 \times K$

where:

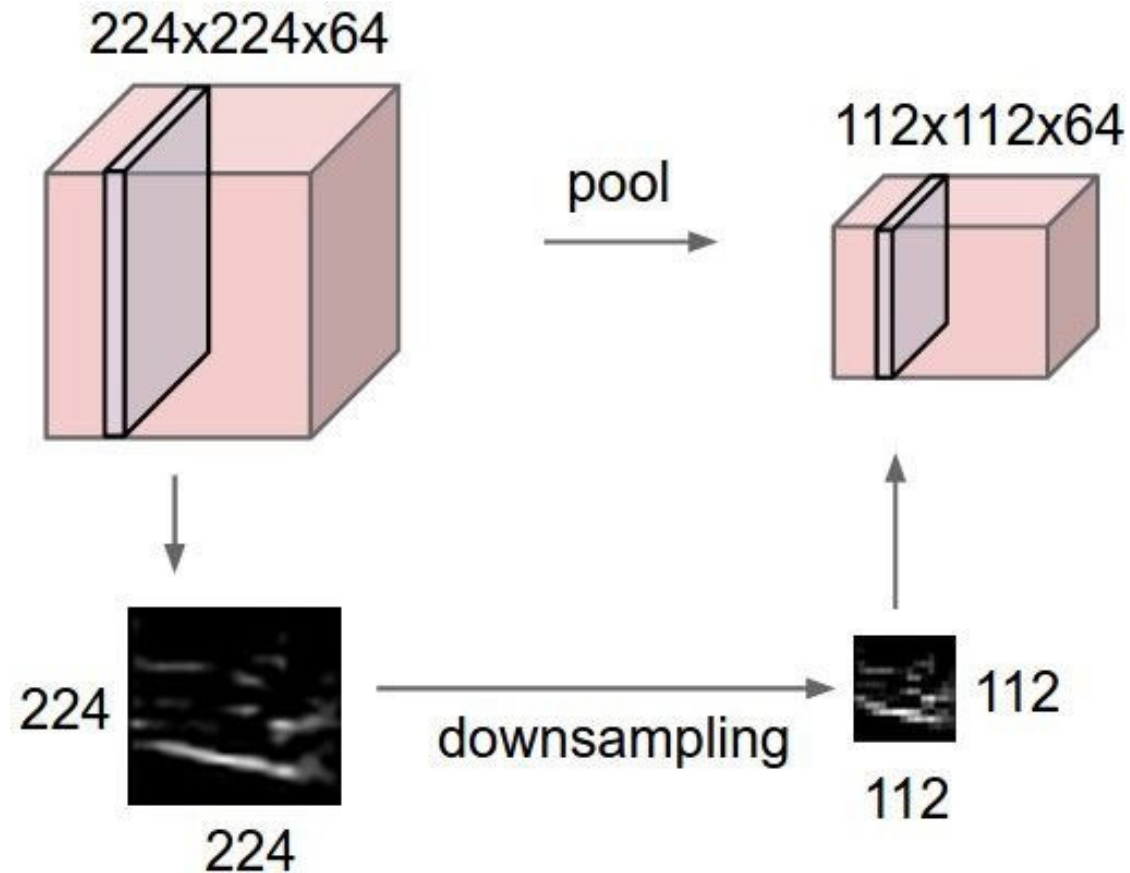
- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$

Number of parameters:  $F^2CK$  and  $K$  biases



# Pooling layer

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



# MAX POOLING

Single depth slice

x ↑

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

→ y

max pool with 2x2 filters  
and stride 2



6	8
3	4

# MAX POOLING

Single depth slice

x ↑

1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

→ y

max pool with 2x2 filters  
and stride 2



6	8
3	4

- No learnable parameters
- introduces spatial invariance

## Pooling layer: summary

Let's assume input is  $W_1 \times H_1 \times C$

Conv layer needs 2 hyperparameters:

- The spatial extent **F**
- The stride **S**

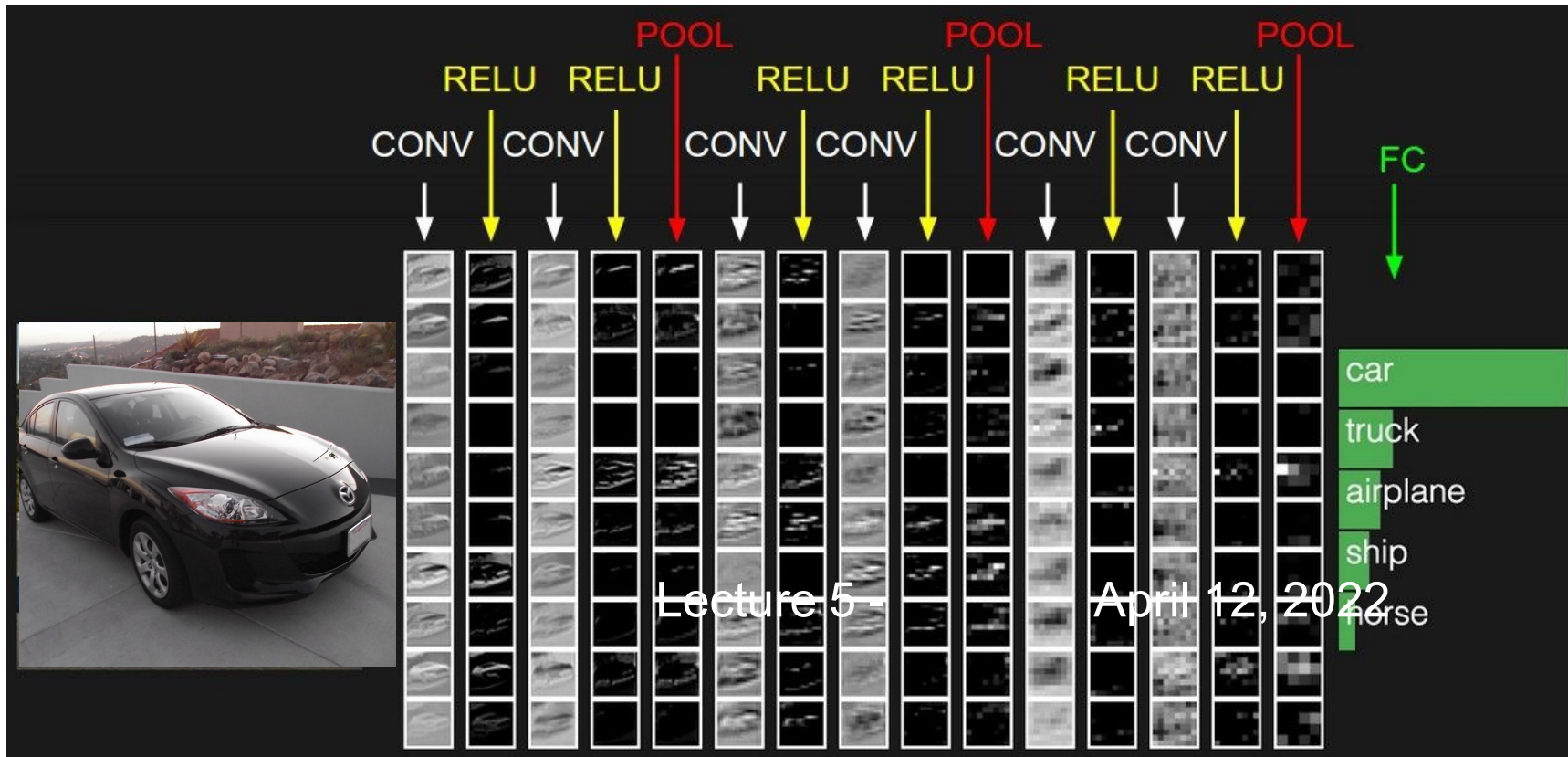
This will produce an output of  $W_2 \times H_2 \times C$  where:

- $W_2 = (W_1 - F) / S + 1$
- $H_2 = (H_1 - F) / S + 1$

Number of parameters: 0

# Fully Connected Layer (FC layer)

- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks



# Summary

- ConvNets stack CONV, POOL, FC layers
- Trend towards smaller filters and deeper architectures
- Trend towards getting rid of POOL/FC layers (just CONV)
- Historically architectures looked like  
 **$[(\text{CONV-RELU})^N\text{-POOL?}]^M\text{-(FC-RELU)}^K, \text{SOFTMAX}$**   
where N is usually up to ~5, M is large,  $0 \leq K \leq 2$ .
- But recent advances such as ResNet/GoogLeNet have challenged this paradigm