



南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Brain Intelligence and Artificial Intelligence

人脑智能与机器智能

Lecture 15 – GD & BP & CNN

Quanying Liu (刘泉影)

SUSTech, BME department

Email: liuqy@sustech.edu.cn

Lecture 15 – GD & BP & CNN & LeNet hands on

- Gradient Descent (GD)
 - What is Gradient Descent?
 - Gradient Descent to train deep NNs → Error Backpropagation
- Error Back-propagation (BP)
 - Backpropagation
 - Backpropagation – forward pass
 - Backpropagation – backward pass
- The Architecture of CNN
 - Convolution
 - Activation function
 - Pooling
 - Flatten
 - FC
- CNN Hands-on (tensorflow), thanks to 曲由之--> next lecture

Gradient Descent

Goal: to solve an optimization problem

$$\boldsymbol{\theta}^* = \operatorname{argmin}_{\boldsymbol{\theta}} L(\boldsymbol{\theta})$$

L : loss function

$\boldsymbol{\theta}$: parameters

Suppose that $\boldsymbol{\theta}$ has two variables $\{\theta_1, \theta_2\}$

Randomly start at $\boldsymbol{\theta}^0 = \begin{bmatrix} \theta_1^0 \\ \theta_2^0 \end{bmatrix}$

$$\boldsymbol{\theta}^1 = \begin{bmatrix} \theta_1^1 \\ \theta_2^1 \end{bmatrix} = \begin{bmatrix} \theta_1^0 \\ \theta_2^0 \end{bmatrix} - \eta \begin{bmatrix} \partial L(\boldsymbol{\theta}^0) / \partial \theta_1 \\ \partial L(\boldsymbol{\theta}^0) / \partial \theta_2 \end{bmatrix} \rightarrow \boldsymbol{\theta}^1 = \boldsymbol{\theta}^0 - \eta \nabla L(\boldsymbol{\theta}^0)$$

$$\boldsymbol{\theta}^2 = \begin{bmatrix} \theta_1^2 \\ \theta_2^2 \end{bmatrix} = \begin{bmatrix} \theta_1^1 \\ \theta_2^1 \end{bmatrix} - \eta \begin{bmatrix} \partial L(\boldsymbol{\theta}^1) / \partial \theta_1 \\ \partial L(\boldsymbol{\theta}^1) / \partial \theta_2 \end{bmatrix} \rightarrow \boldsymbol{\theta}^2 = \boldsymbol{\theta}^1 - \eta \nabla L(\boldsymbol{\theta}^1)$$

Gradient:

$$\nabla L(\boldsymbol{\theta}) = \begin{bmatrix} \partial L(\boldsymbol{\theta}) / \partial \theta_1 \\ \partial L(\boldsymbol{\theta}) / \partial \theta_2 \end{bmatrix}$$

Learning rate: η

.....

until converge to $\boldsymbol{\theta}^*$

An example to calculate gradient of loss function

Loss function: $L(\theta) = \frac{1}{2}(\theta - y)^2$

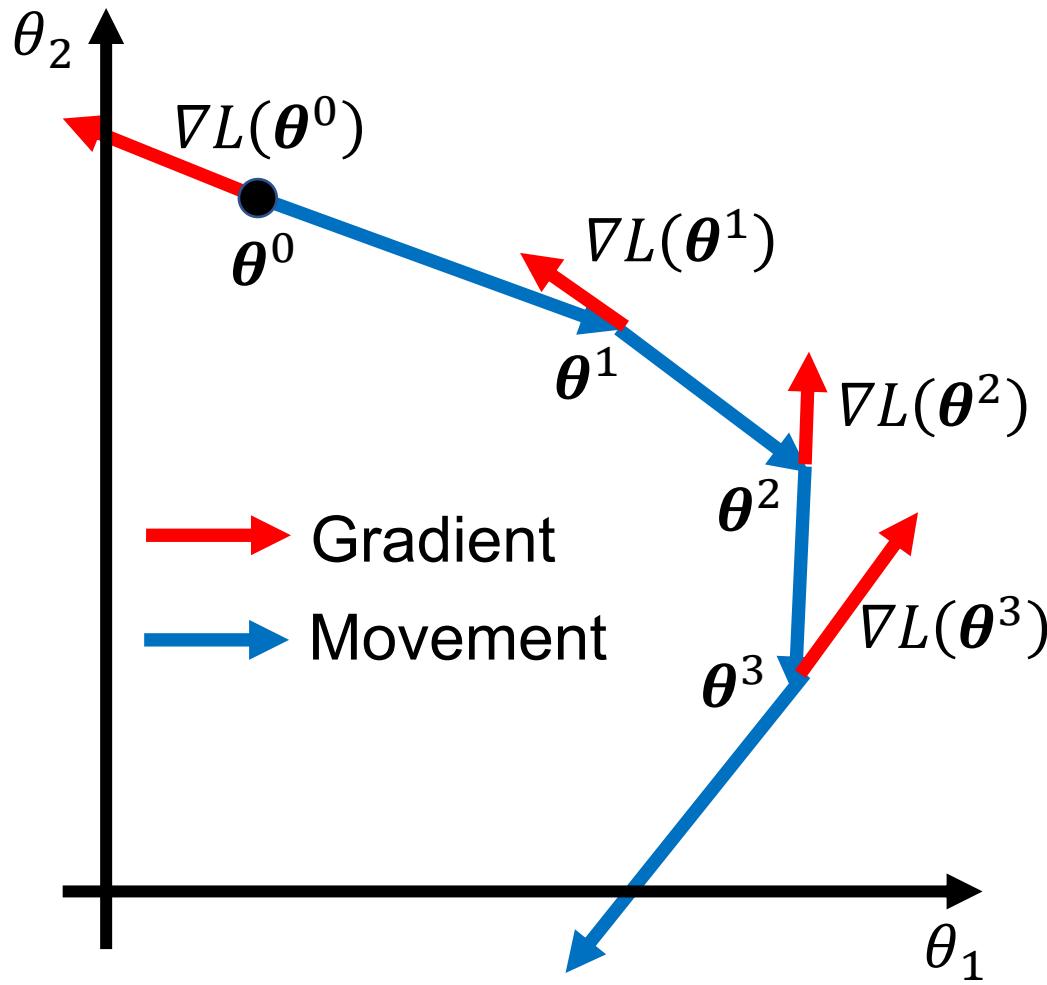
$$L(\theta) = \frac{1}{2} \left(\begin{bmatrix} \theta_1 - y_1 \\ \theta_2 - y_2 \end{bmatrix} \right)^2$$

Gradient:

$$\nabla L(\theta) = \begin{bmatrix} \partial L(\theta)/\partial \theta_1 \\ \partial L(\theta)/\partial \theta_2 \end{bmatrix} = ?$$

$$\nabla L \left(\begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix} \right) = \begin{bmatrix} \theta_1 - y_1 \\ \theta_2 - y_2 \end{bmatrix}$$

Steps for Gradient Descent



Start at position θ^0

Compute gradient at θ^0

Move to $\theta^1 = \theta^0 - \eta \nabla L(\theta^0)$

Compute gradient at θ^1

Move to $\theta^2 = \theta^1 - \eta \nabla L(\theta^1)$

: :

Gradient: derivative of Loss function

$$\nabla L(\theta) = \begin{bmatrix} \partial L(\theta_1)/\partial \theta_1 \\ \partial L(\theta_2)/\partial \theta_2 \end{bmatrix}$$

Gradient Descent to train Neural Networks

Network parameters

$$\theta = \{w_1, w_2, \dots, b_1, b_2, \dots\}$$

Starting
Parameters

$$\theta^0 \longrightarrow \theta^1 \longrightarrow \theta^2 \longrightarrow \dots$$

Compute $\nabla L(\theta^0)$ *Compute $\nabla L(\theta^1)$*

$$\theta^1 = \theta^0 - \eta \nabla L(\theta^0) \quad \theta^2 = \theta^1 - \eta \nabla L(\theta^1)$$

$$\nabla L(\theta) = \begin{bmatrix} \partial L(\theta)/\partial w_1 \\ \partial L(\theta)/\partial w_2 \\ \vdots \\ \partial L(\theta)/\partial b_1 \\ \partial L(\theta)/\partial b_2 \\ \vdots \end{bmatrix}$$

NNs have Millions of parameters

To compute the gradients efficiently in NNs,
we use **backpropagation**.

Recall Calculus: Chain Rule

Case 1

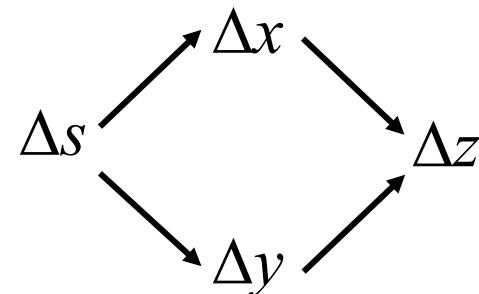
$$y = g(x) \quad z = h(y)$$

$$\Delta x \rightarrow \Delta y \rightarrow \Delta z$$

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$$

Case 2

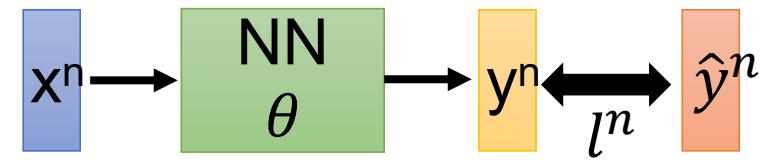
$$x = g(s) \quad y = h(s) \quad z = k(x, y)$$



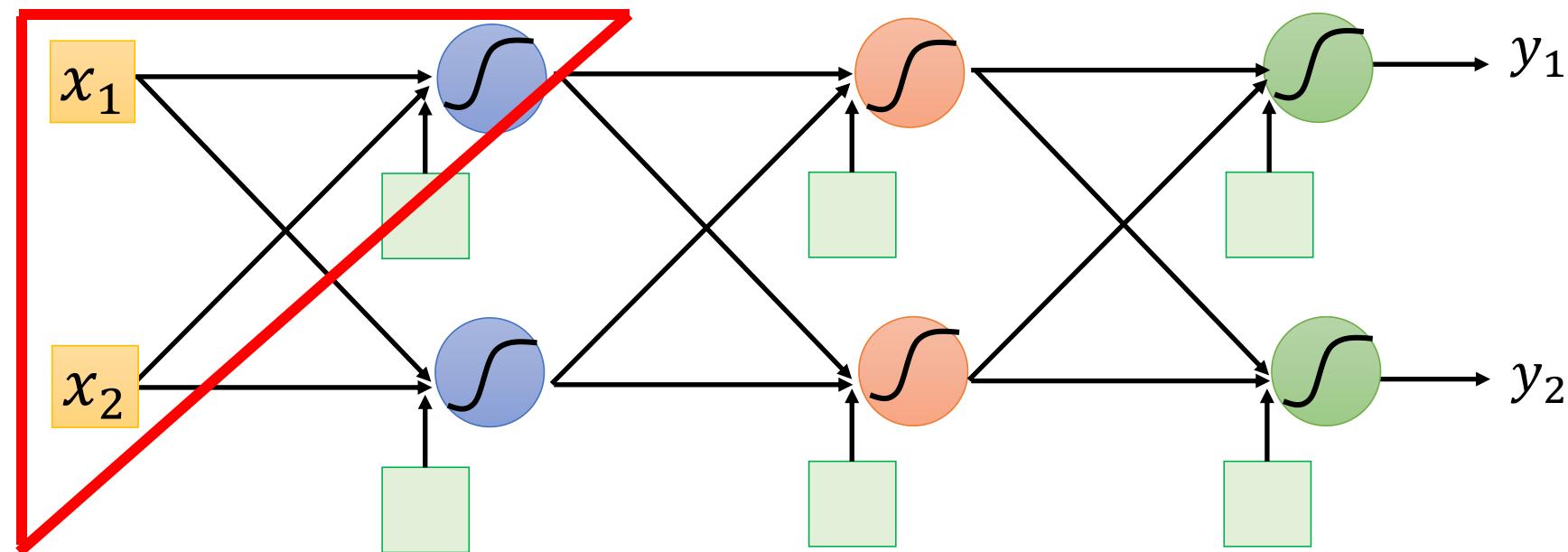
$$\frac{dz}{ds} = \frac{\partial z}{\partial x} \frac{dx}{ds} + \frac{\partial z}{\partial y} \frac{dy}{ds}$$

Error Backpropagation

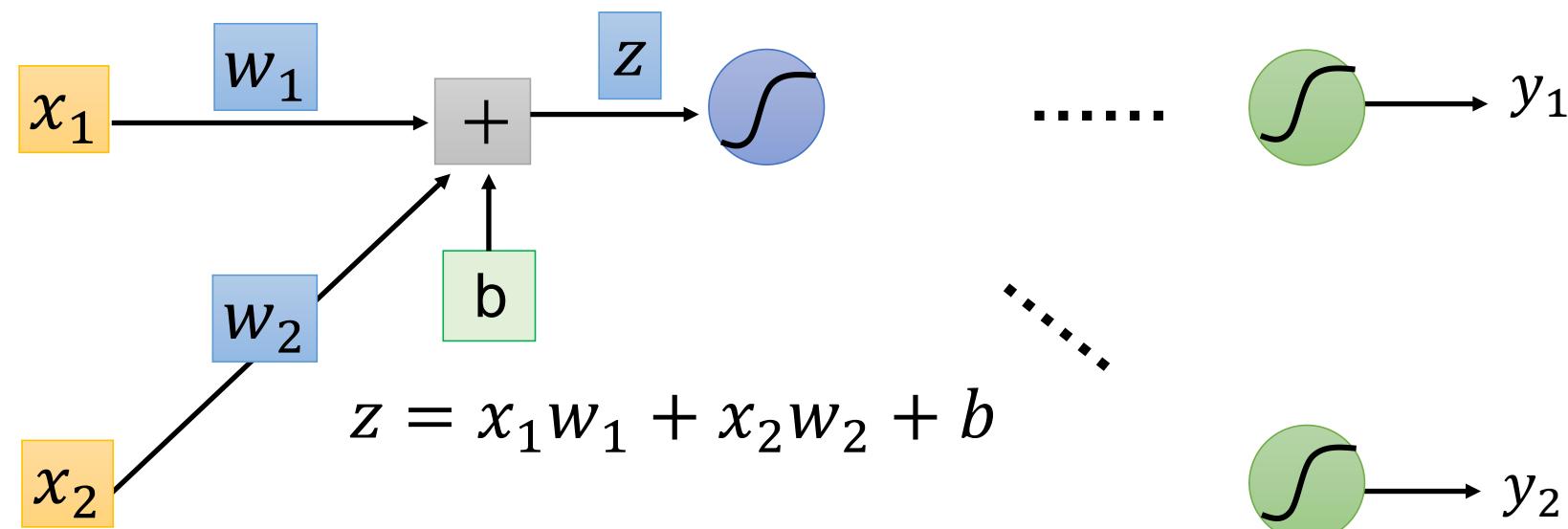
Backpropagation computes how slightly changing each synapse strength (**weight**) would change the network's **error**, using the chain rule.



$$L(\theta) = \sum_{n=1}^N l^n(\theta) \quad \rightarrow \quad \frac{\partial L(\theta)}{\partial w} = \sum_{n=1}^N \frac{\partial l^n(\theta)}{\partial w}$$



BP in fully-connected neural networks (fcNN)



Forward pass:

$\frac{\partial l}{\partial w} = ? \quad \frac{\partial l}{\partial z} \frac{\partial z}{\partial w}$

(Chain rule)

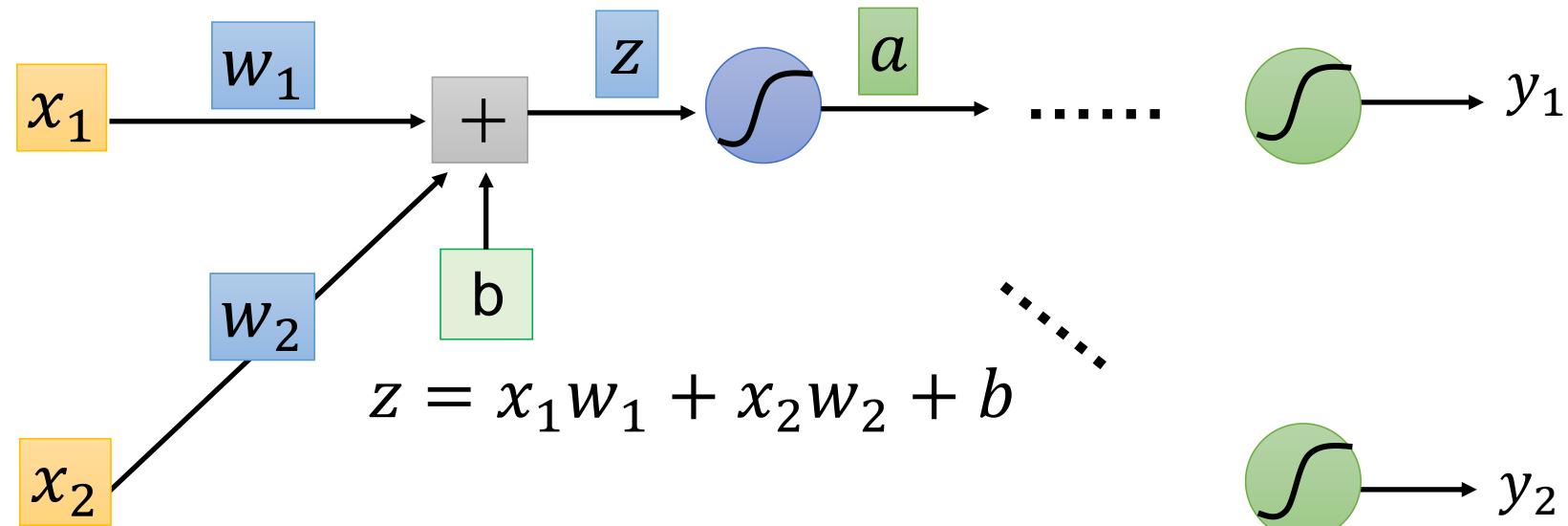
Compute $\frac{\partial z}{\partial w}$ for all parameters

Backward pass:

Compute $\frac{\partial l}{\partial z}$ for all activation function inputs z

Backpropagation – Forward pass

Compute $\partial z / \partial w$ for all parameters

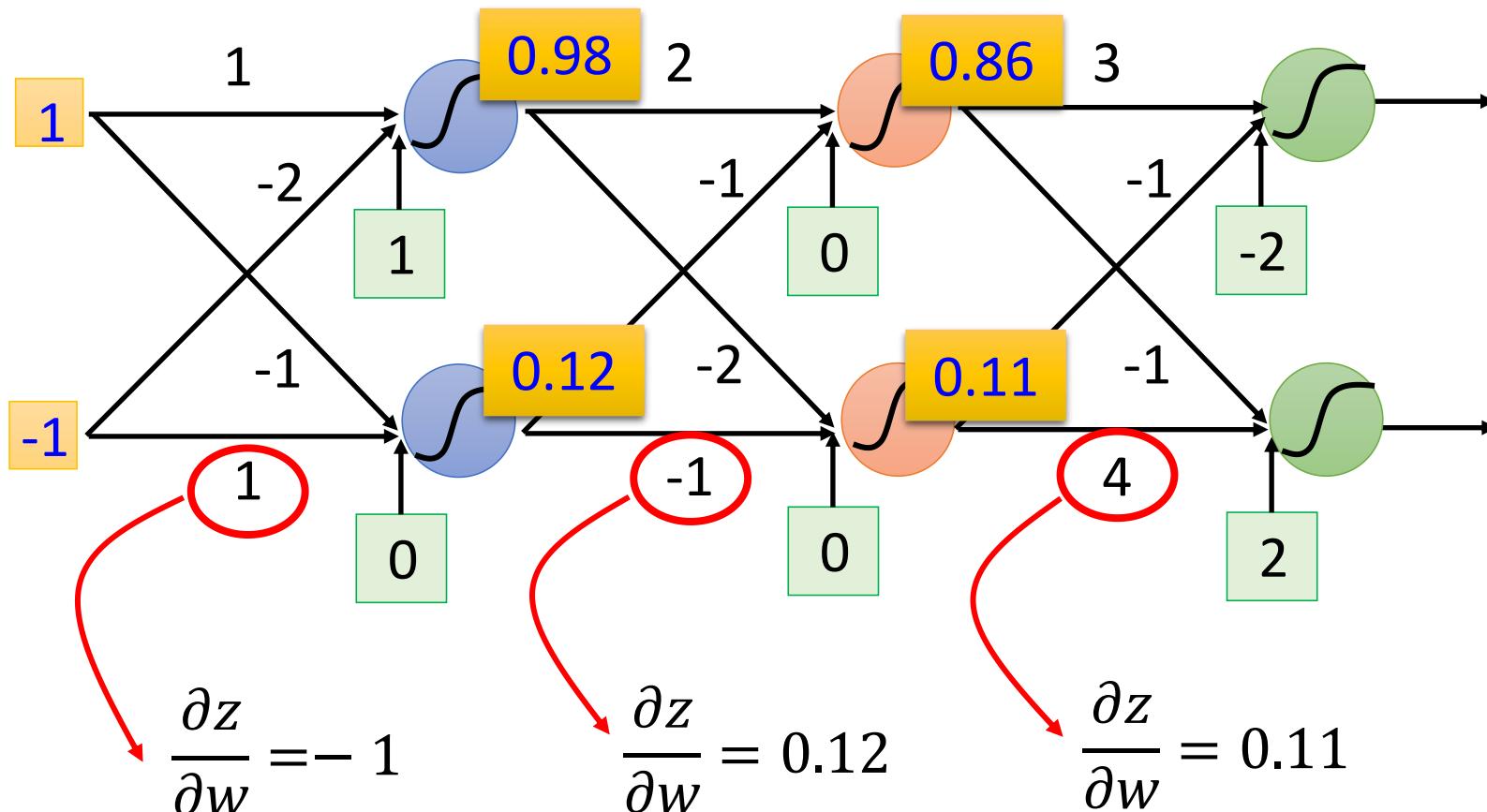


$$\begin{aligned}\partial z / \partial w_1 &= ? \ x_1 \\ \partial z / \partial w_2 &= ? \ x_2\end{aligned}$$

The value of the **input** connected by the weight

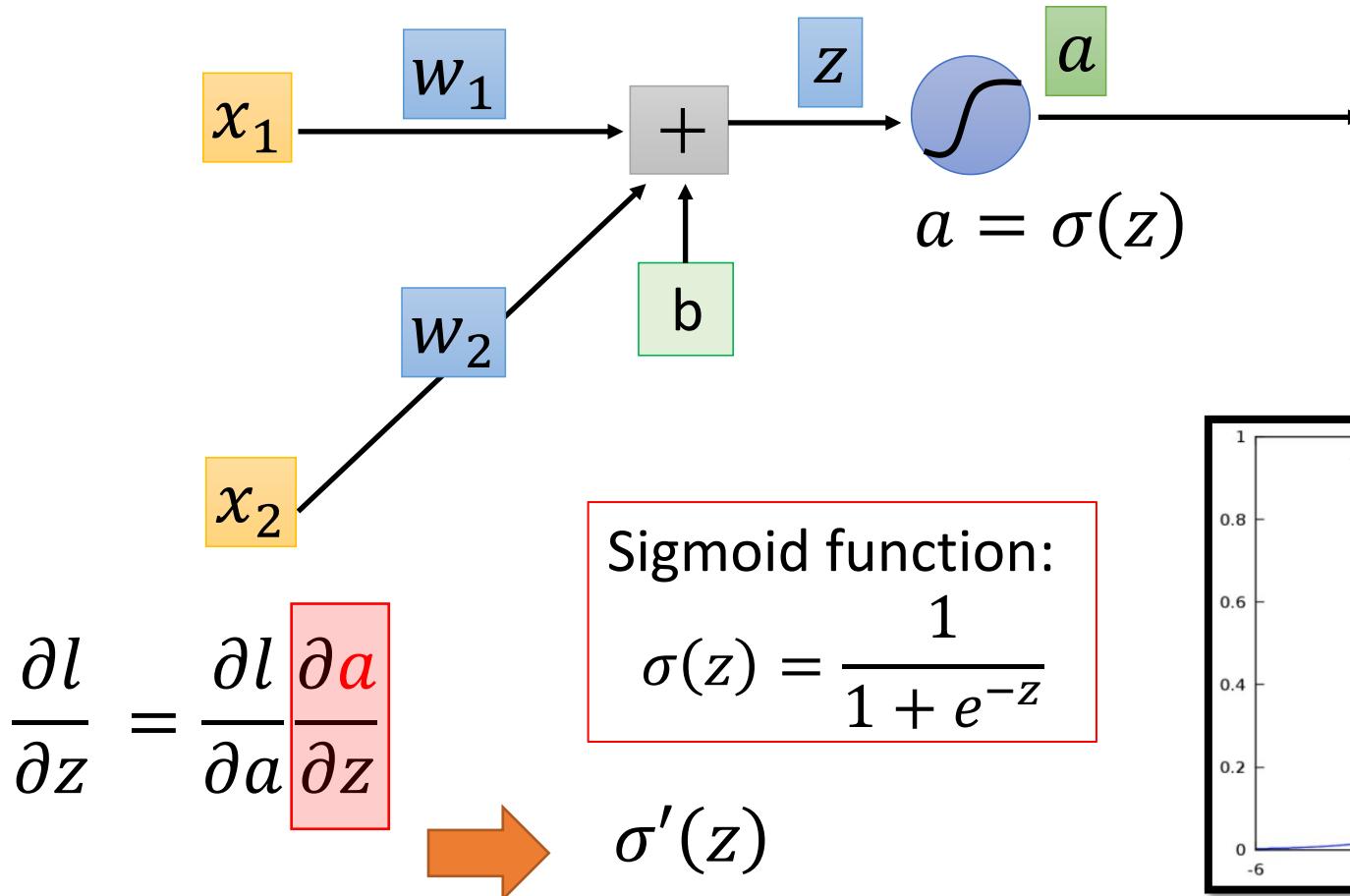
Backpropagation – Forward pass

Compute $\partial z / \partial w$ for all parameters



Backpropagation – Backward pass

Compute $\partial a / \partial z$ for all activation function inputs z

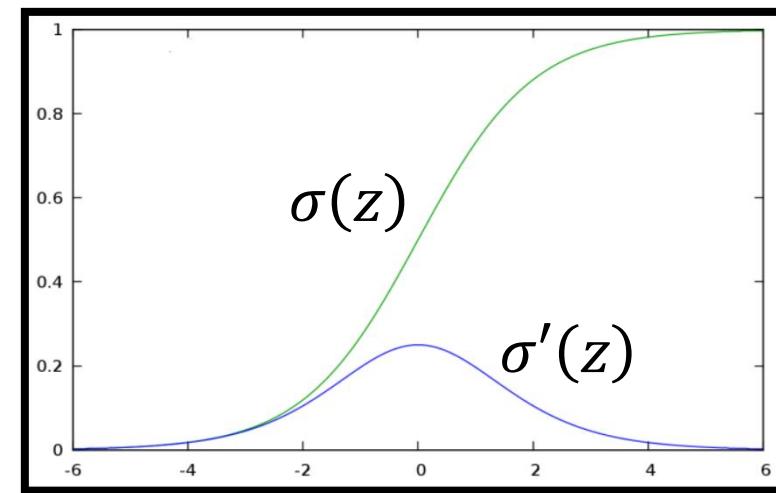
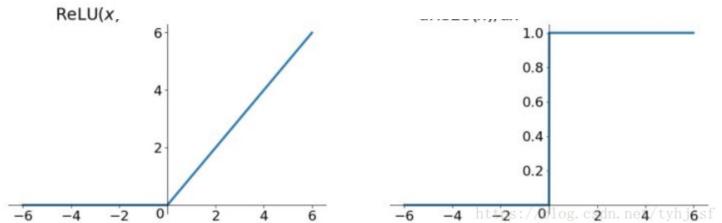


Sigmoid function:

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

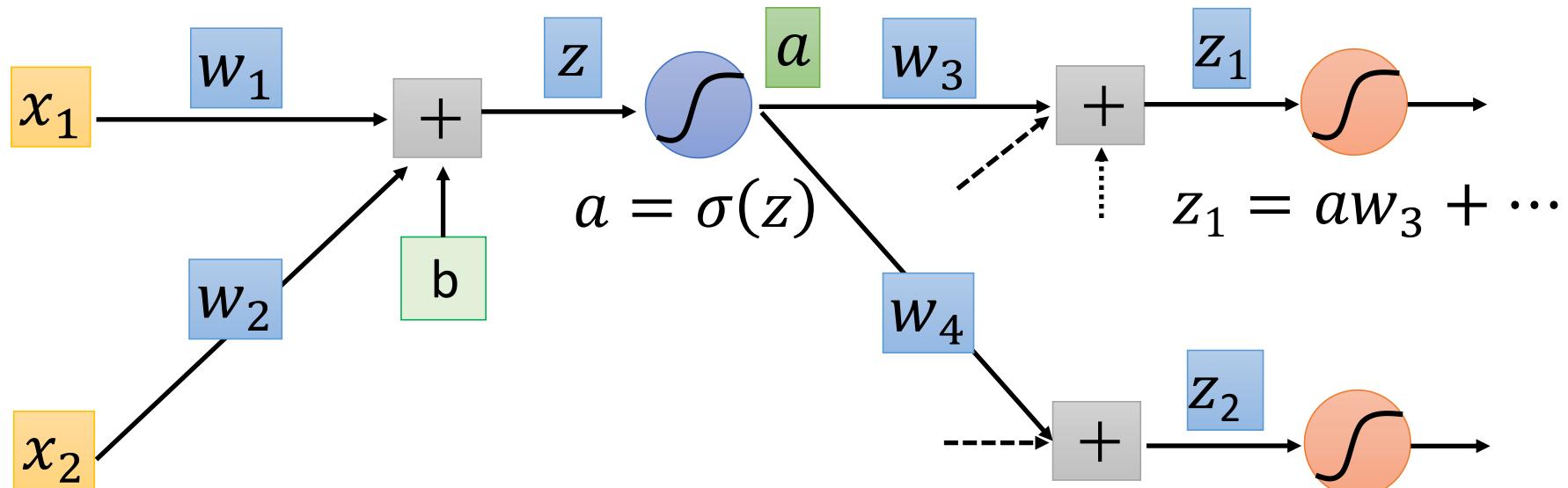
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\text{Relu} = \max(0, x)$$



Backpropagation – Backward pass

Compute $\partial a / \partial z$ for all activation function inputs z



$$\frac{\partial l}{\partial z} = \frac{\partial l}{\partial a} \frac{\partial a}{\partial z}$$

$\sigma'(z)$

$$\frac{\partial l}{\partial a} = \frac{\partial l}{\partial z_1} \frac{\partial z_1}{\partial a} + \frac{\partial l}{\partial z_2} \frac{\partial z_2}{\partial a}$$

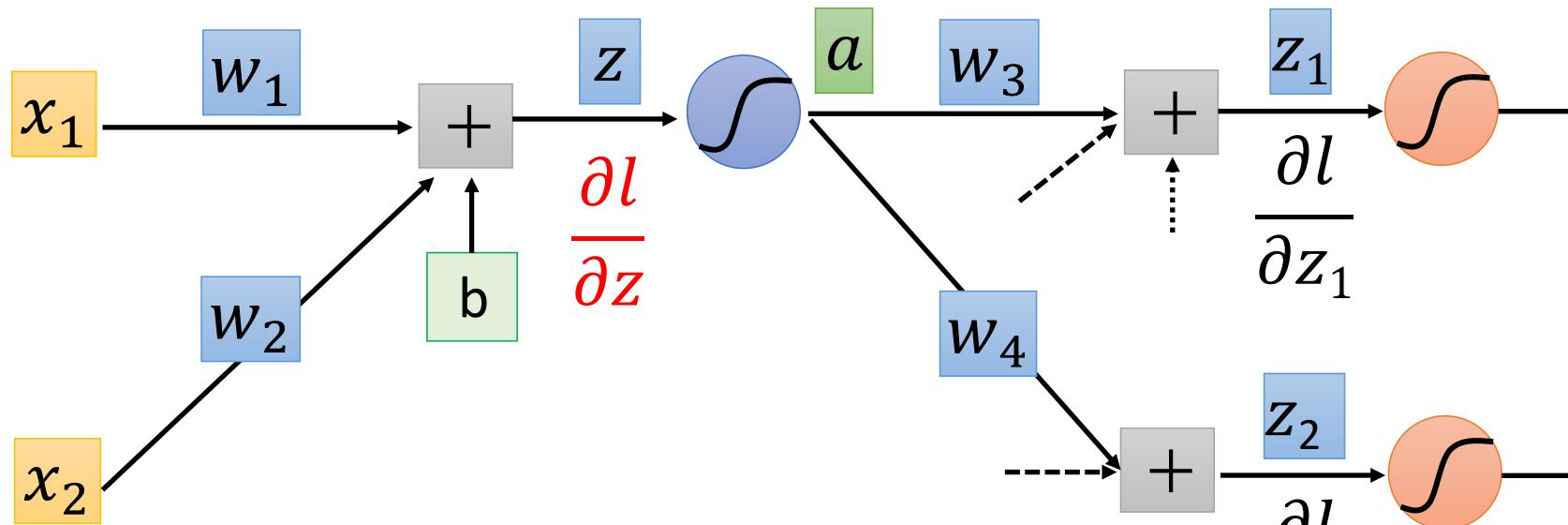
(Chain rule)

? w_3 ? w_4

Assumed it's known

Backpropagation – Backward pass

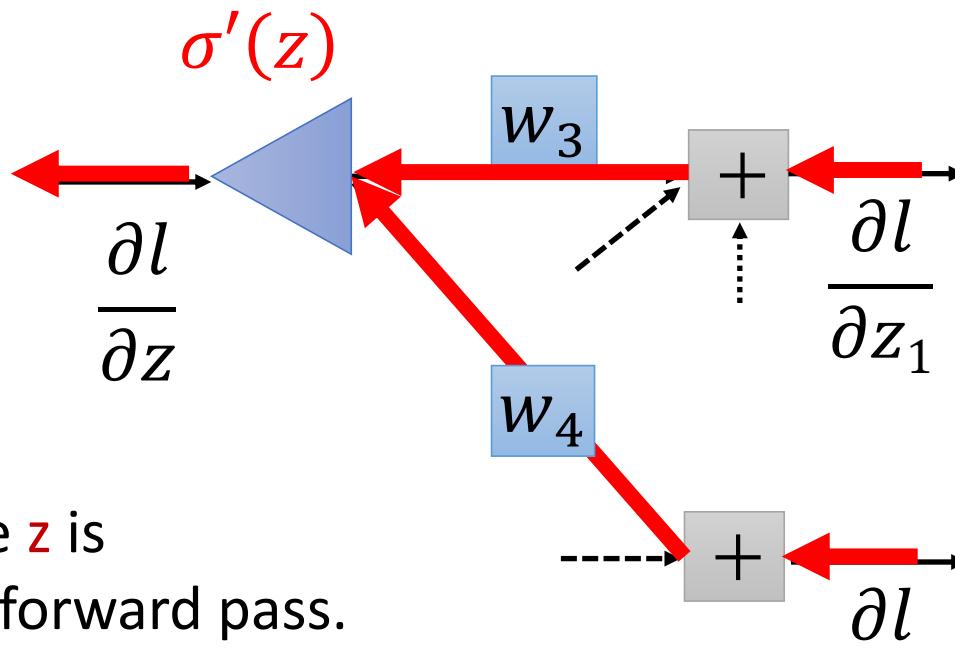
Compute $\frac{\partial l}{\partial z}$ for all activation function inputs z



$$\frac{\partial l}{\partial z} = \sigma'(z) \left[w_3 \frac{\partial l}{\partial z_1} + w_4 \frac{\partial l}{\partial z_2} \right]$$

Backpropagation – Backward pass

Compute $\frac{\partial l}{\partial z}$ for all activation function inputs z

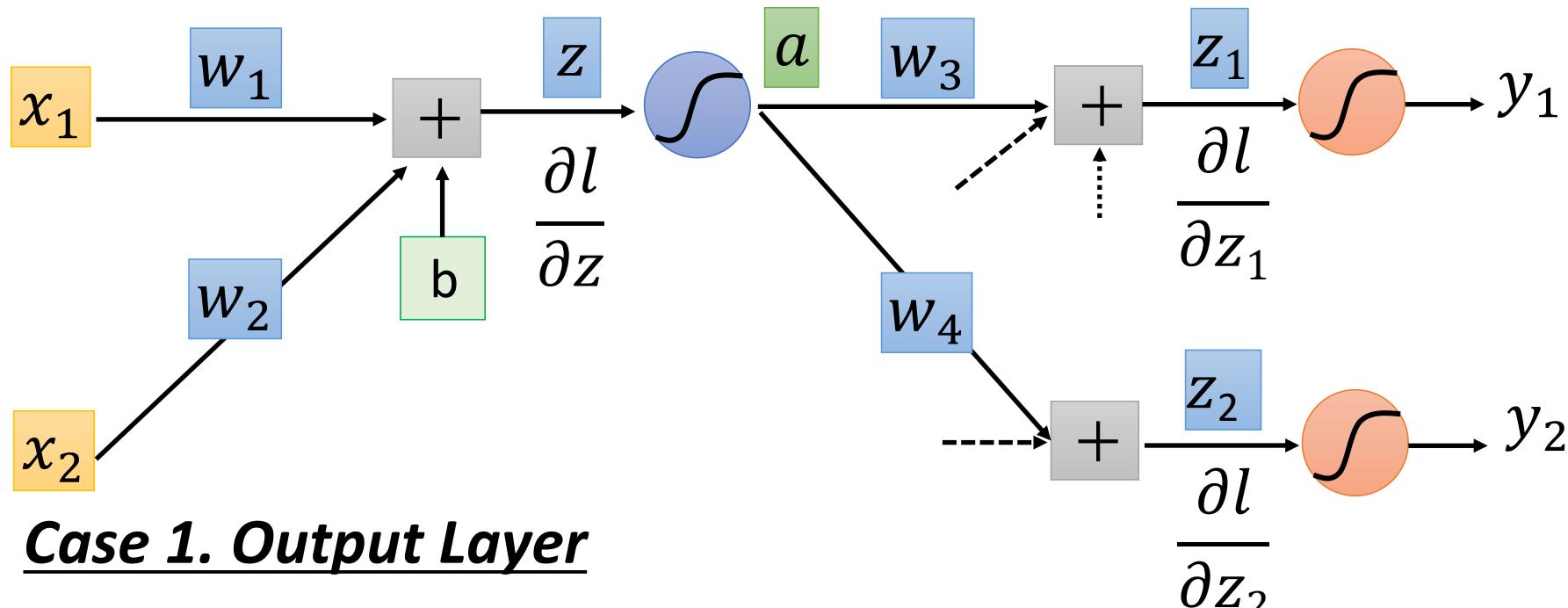


$\sigma'(z)$ is a constant because z is already determined in the forward pass.

$$\frac{\partial l}{\partial z} = \sigma'(z) \left[w_3 \frac{\partial l}{\partial z_1} + w_4 \frac{\partial l}{\partial z_2} \right]$$

Backpropagation – Backward pass

Compute $\frac{\partial l}{\partial z}$ for all activation function inputs z



Case 1. Output Layer

$$\frac{\partial l}{\partial z_1} = \frac{\partial l}{\partial y_1} \frac{\partial y_1}{\partial z_1}$$

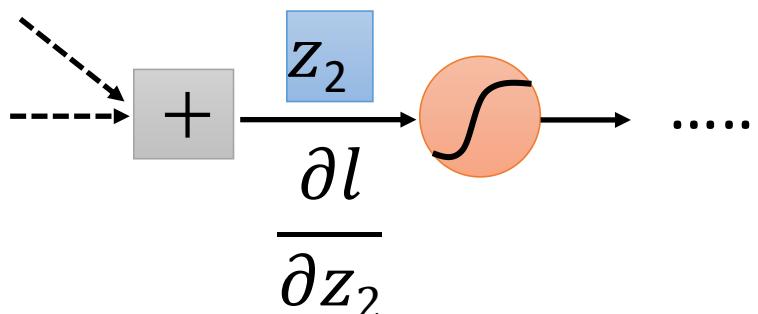
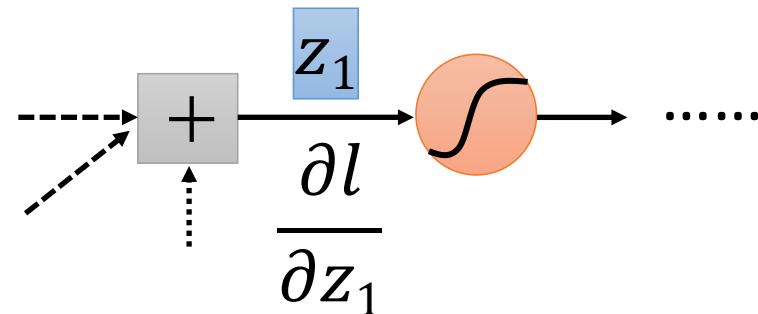
$$\frac{\partial l}{\partial z_2} = \frac{\partial l}{\partial y_2} \frac{\partial y_2}{\partial z_2}$$

Done!

Backpropagation – Backward pass

Compute $\partial l / \partial z$ for all activation function inputs z

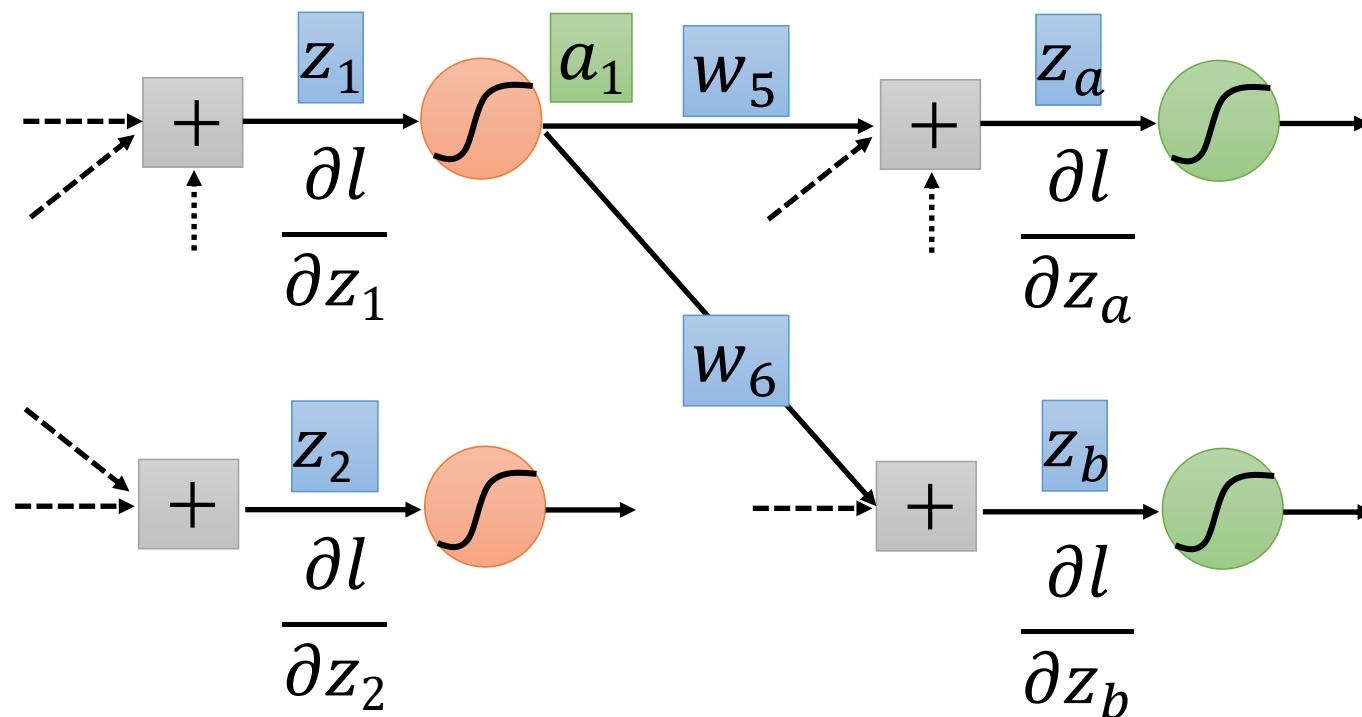
Case 2. Not Output Layer



Backpropagation – Backward pass

Compute $\partial \sigma / \partial z$ for all activation function inputs z

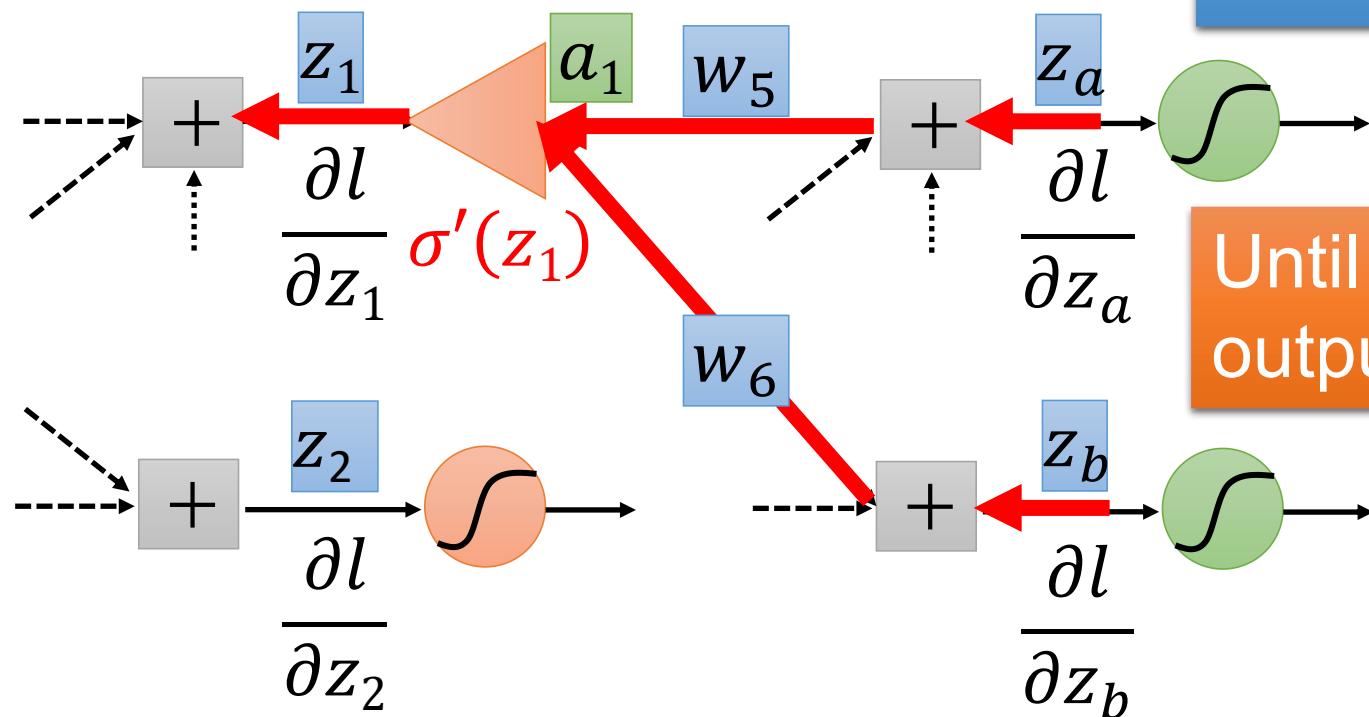
Case 2. Not Output Layer



Backpropagation – Backward pass

Compute $\frac{\partial l}{\partial z}$ for all activation function inputs z

Case 2. Not Output Layer



Compute $\frac{\partial l}{\partial z}$
recursively

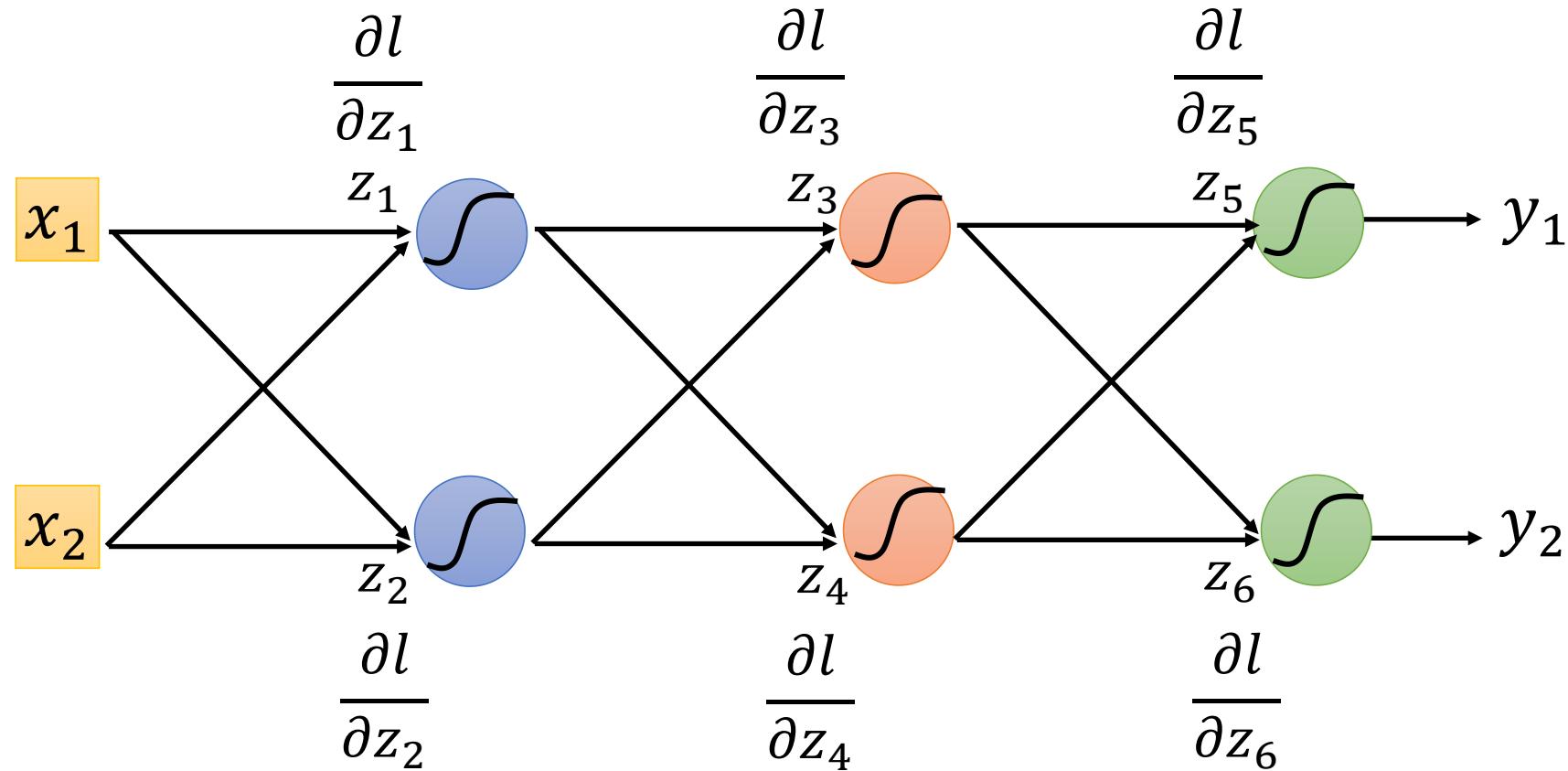
Until we reach the
output layer

Backpropagation – Backward pass

Compute $\partial l / \partial z$ for all activation function inputs z

Compute $\partial l / \partial z$ from the output layer

$$\frac{\partial l}{\partial z} = \sigma'(z) \left[w_3 \frac{\partial l}{\partial z_1} + w_4 \frac{\partial l}{\partial z_2} \right]$$

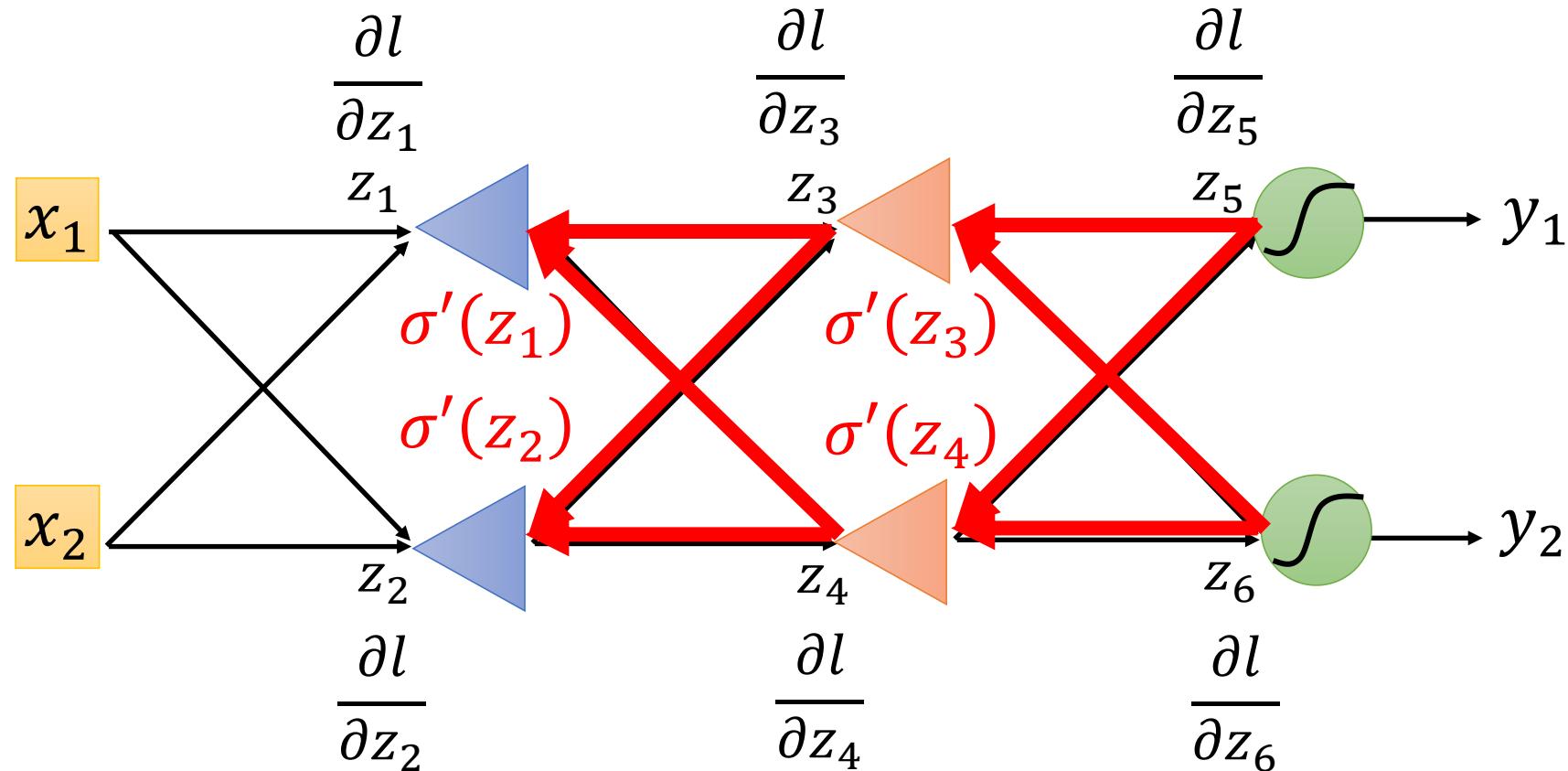


Backpropagation – Backward pass

Compute $\partial l / \partial z$ for all activation function inputs z

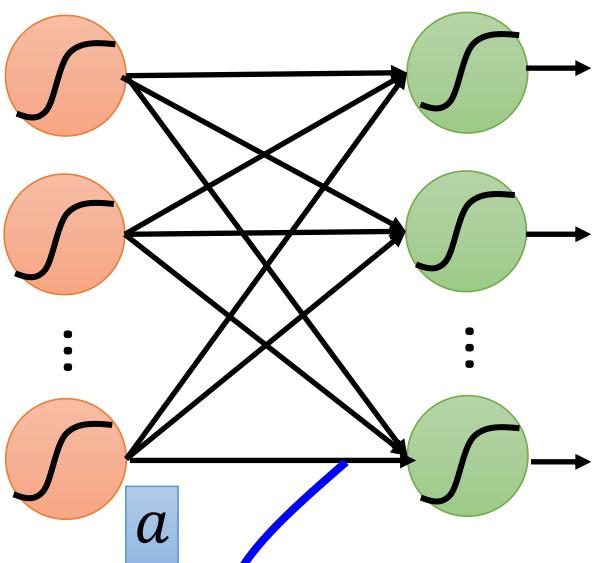
Compute $\partial l / \partial z$ from the output layer

$$\frac{\partial l}{\partial z} = \sigma'(z) \left[w_3 \frac{\partial l}{\partial z_1} + w_4 \frac{\partial l}{\partial z_2} \right]$$



Backpropagation – Overview

Forward Pass

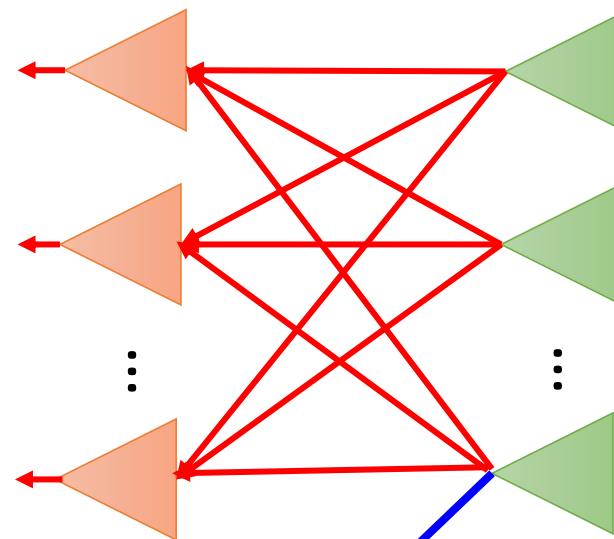


Initiate parameters all w^0
Go forward to get all a^0

$$\frac{\partial z}{\partial w_{i,j}^0} = a_{i,j}^0$$

$$\frac{\partial z}{\partial w} = a$$

Backward Pass



$$X \quad \frac{\partial l}{\partial z} = \frac{\partial l}{\partial w}$$

for all w

$$\frac{\partial l}{\partial z_{i,j}} = \sigma'(z_{i,j}) \left[\sum w_{i,j+1} \frac{\partial l}{\partial z_{i,j+1}} \right]$$

$$\frac{\partial l}{\partial z_{1,n}} = \frac{\partial l}{\partial y_{1,n}} \frac{\partial y_{1,n}}{\partial z_{1,n}}$$

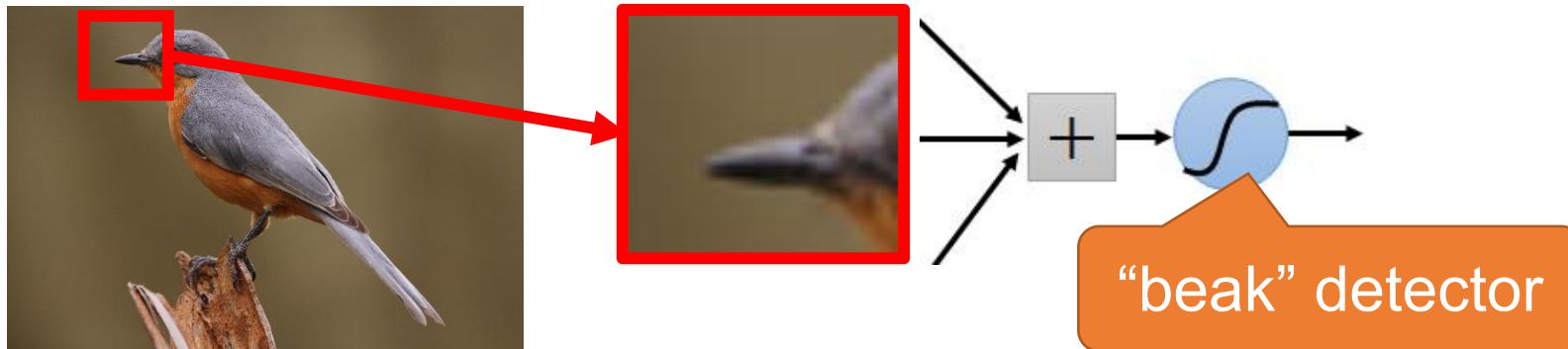
The architecture of CNN

Why CNN for Image process

1. Some patterns are much smaller than the whole image

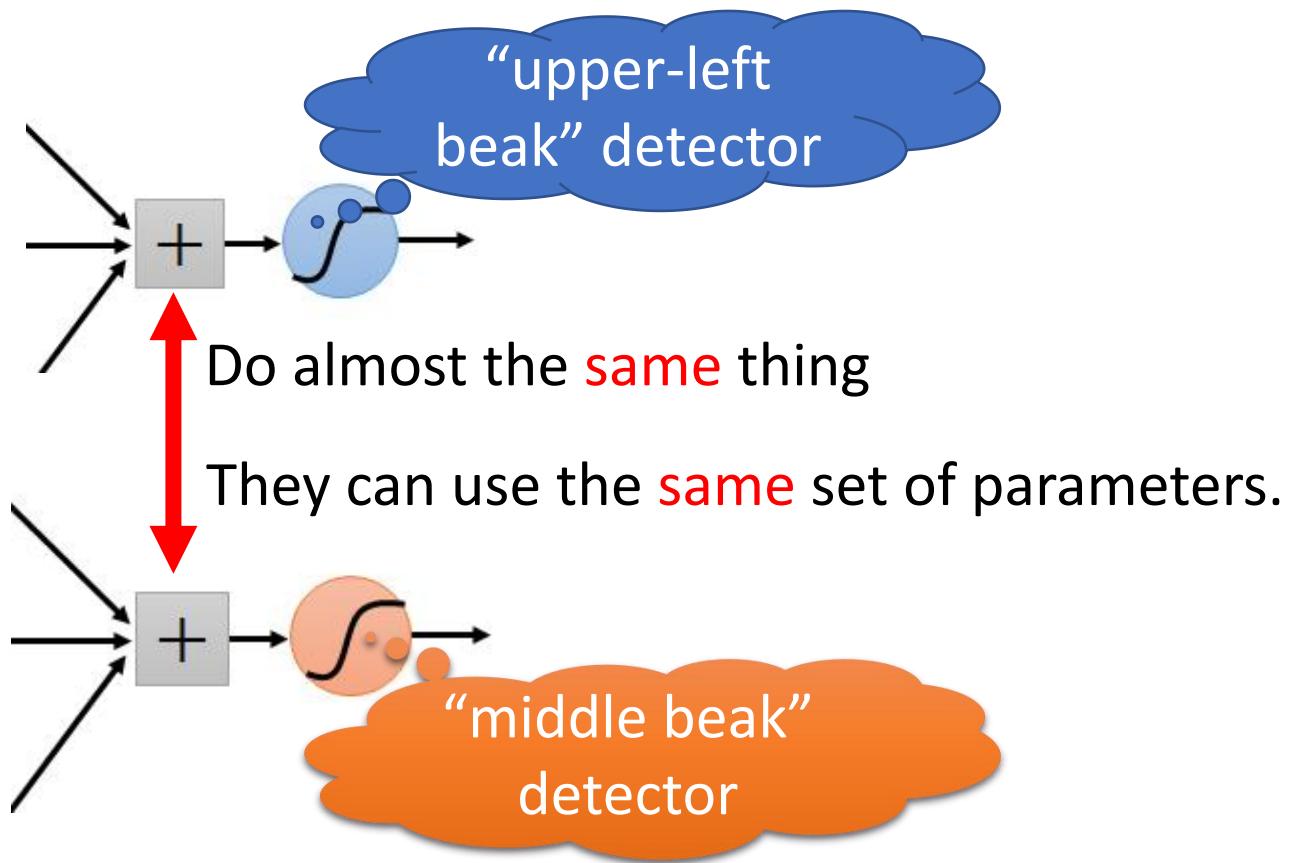
A neuron does **not** have to see the whole image to discover the pattern.

Connecting to small region with **less** parameters



Why CNN for Image process

2. The same patterns appear in different regions.



Why CNN for Image process

3. Subsampling the pixels will not change the object

bird



subsampling

bird

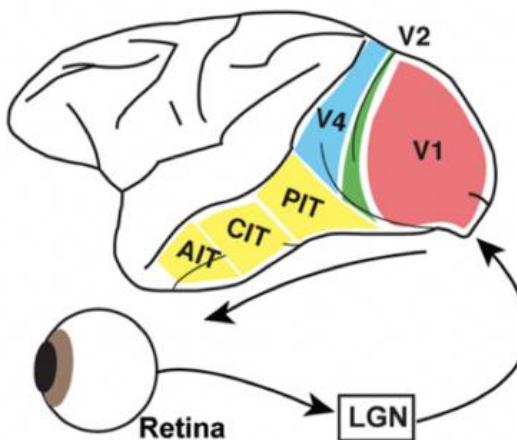
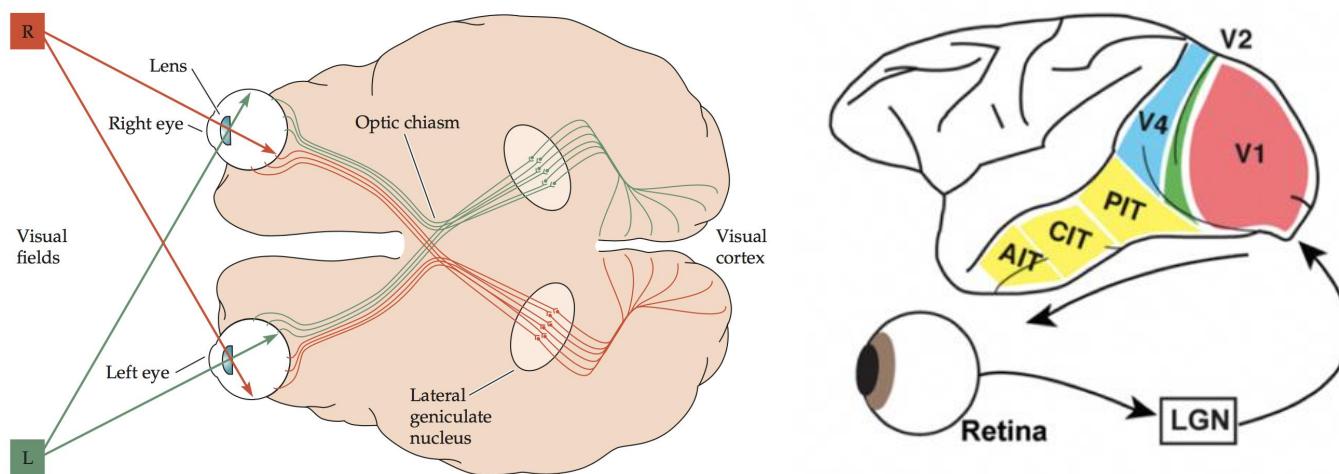


We can subsample the pixels to make image smaller

→ Less parameters for the network to process the image

Why CNN for Image process

4. Layered, hierarchical process in visual system

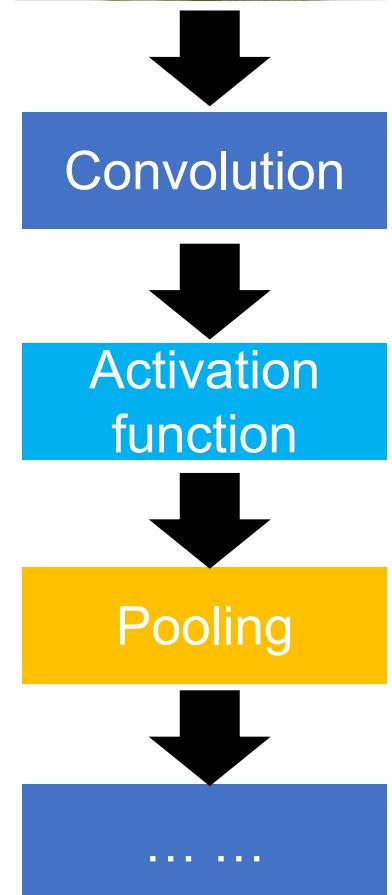
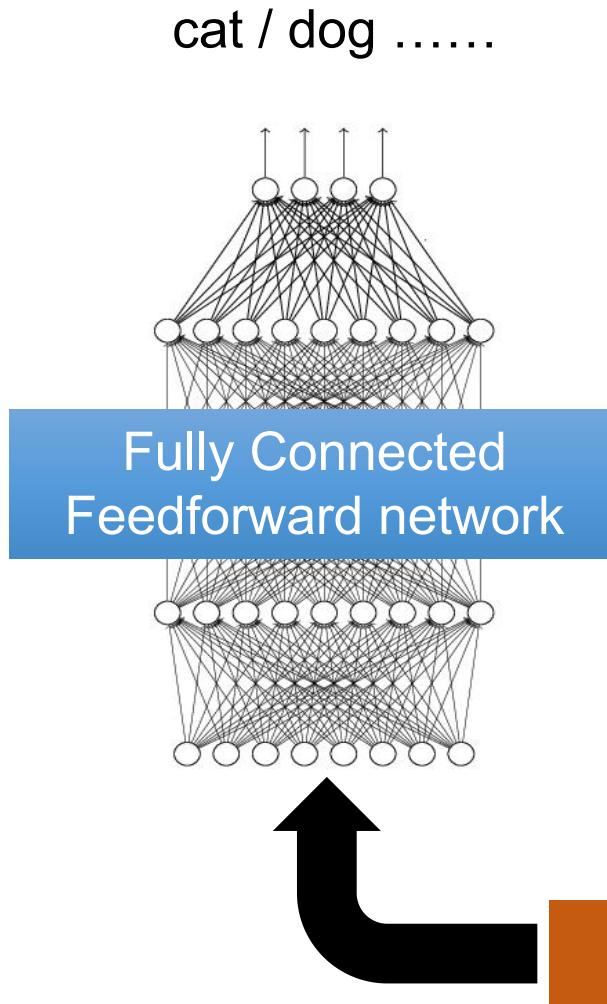


Rods, cones
Interneurons
Ganglion cells
Optic fiber
LGN
V1
Ventral / dorsal pathways

Along the Visual Pathway, feature extraction from simple to complex.

→ **Automatically** learn the hidden features in the image

The Architecture of CNN



These blocks can repeat many times.

The Architecture of CNN

Property 1

- Some patterns are much smaller than the whole image

Property 2

- The same patterns appear in different regions.

Property 3

- Subsampling the pixels will not change the object



Convolution

Activation
function

Pooling

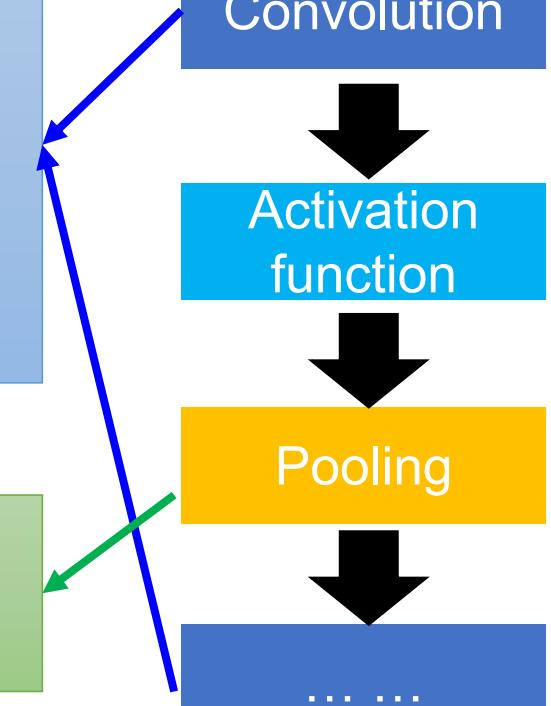
.....

Flatten

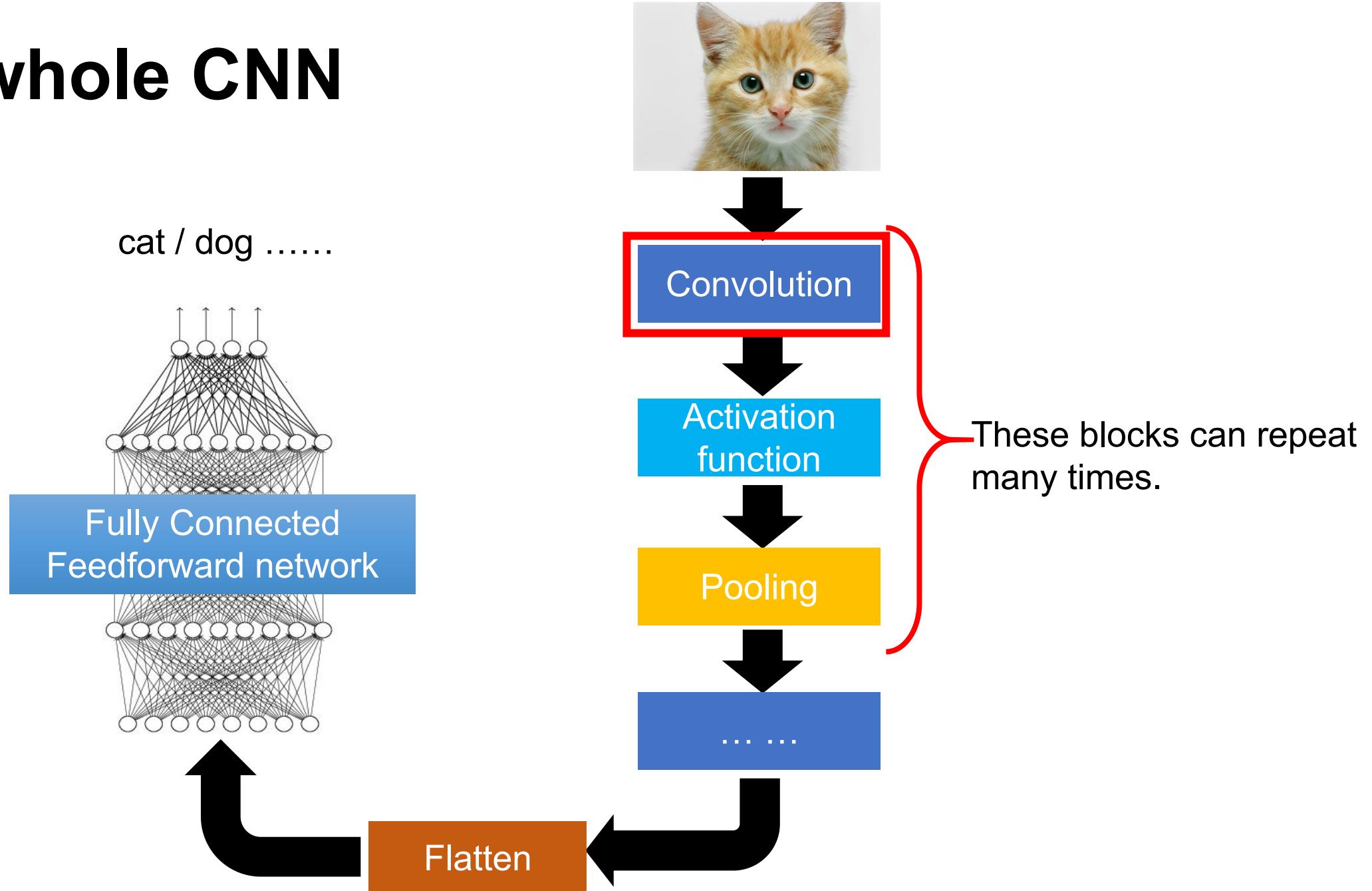
Property 4

Hierarchical architecture

These blocks can repeat many times.



The whole CNN



CNN – Convolution

These are **network parameters** to be learned.

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

Property 1

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

Matrix W1

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

Matrix W2

⋮

Each filter detects a small pattern (3 x 3).

CNN – Convolution

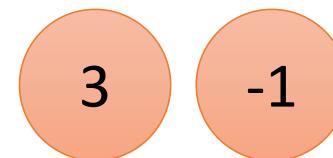
stride=1

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1



CNN – Convolution

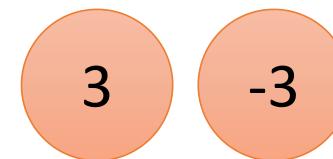
If stride=2

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

1	-1	-1
-1	1	-1
-1	-1	1

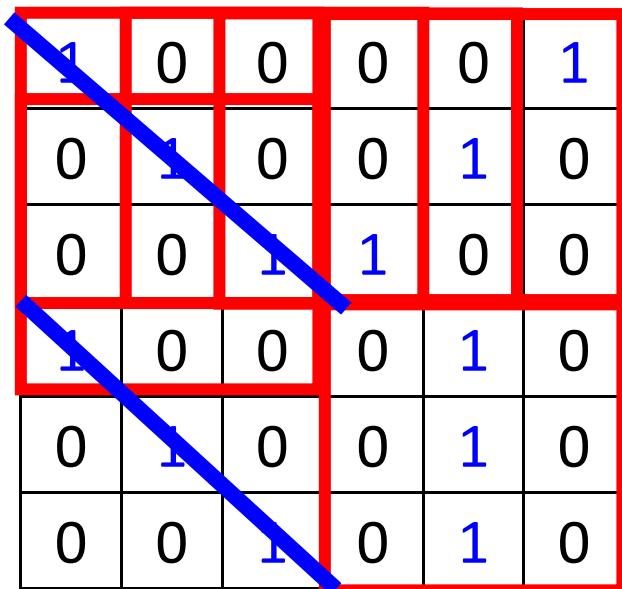
Filter 1



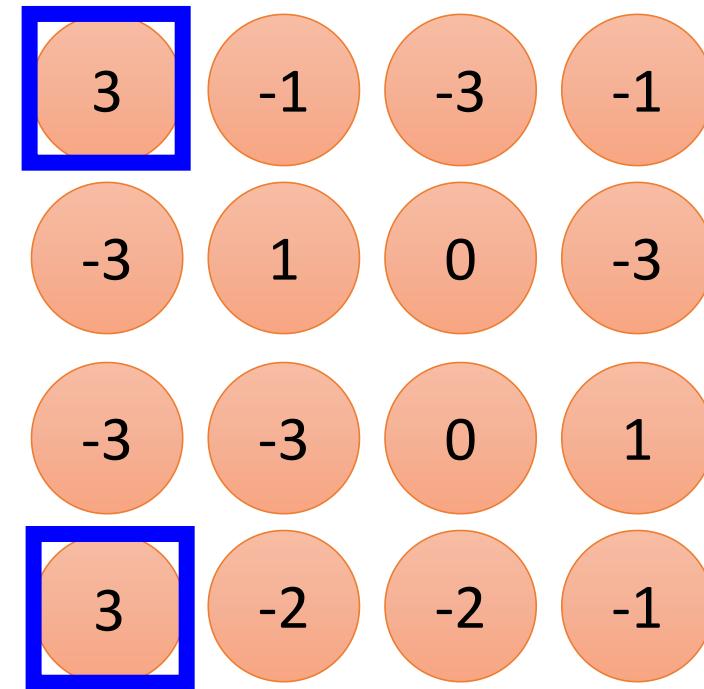
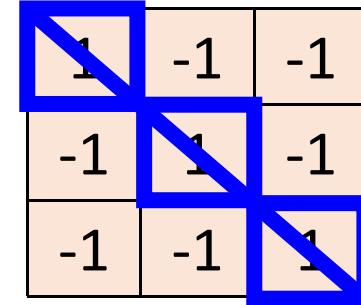
We set stride=1 below

CNN – Convolution

stride=1



6 x 6 image



CNN – Convolution

stride=1

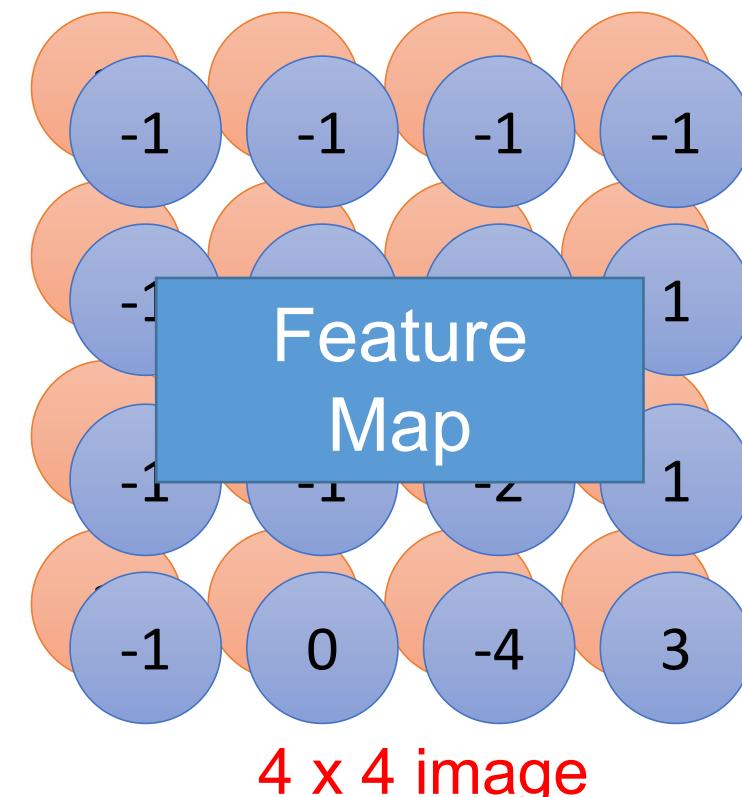
1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image

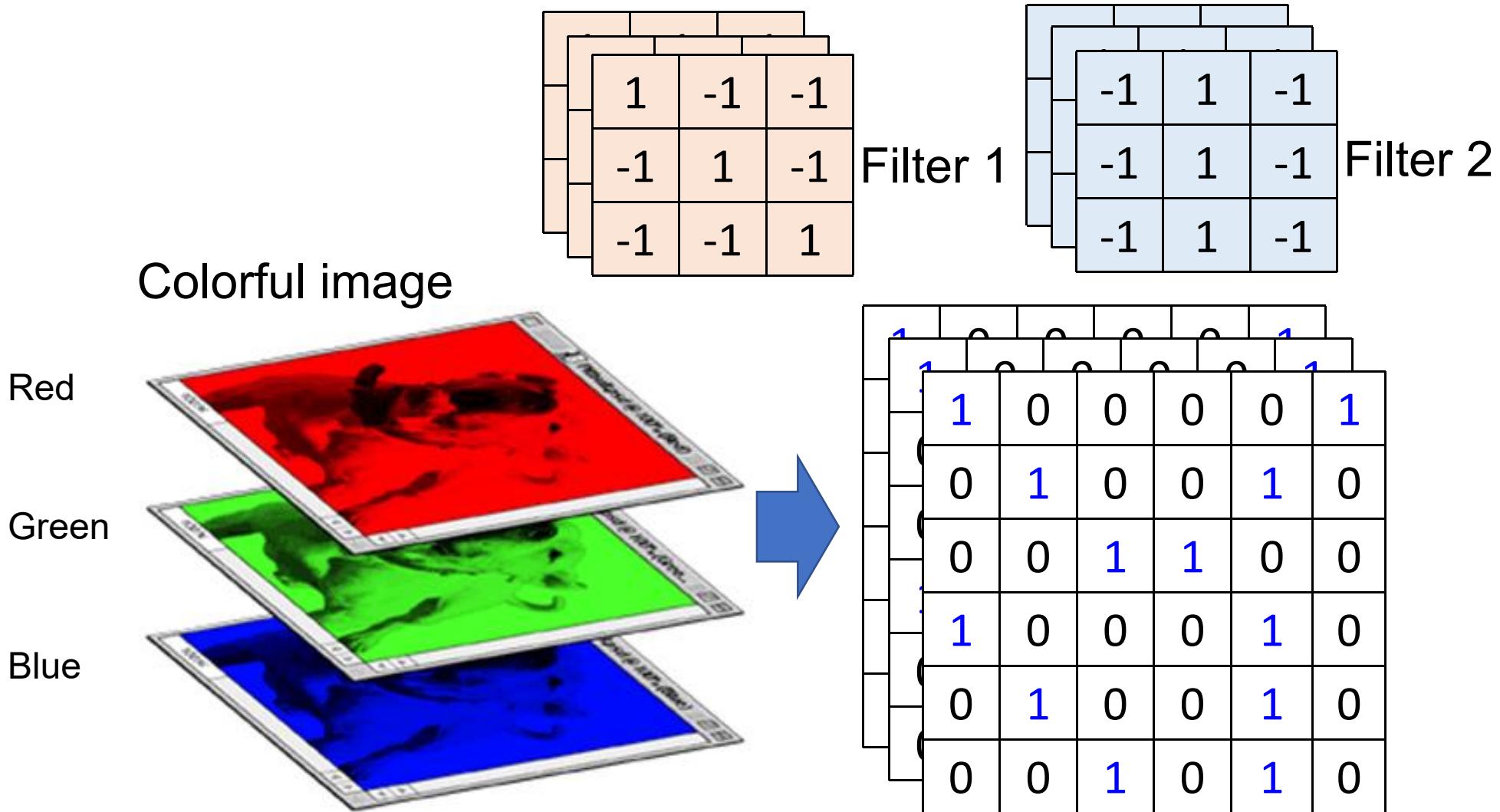
-1	1	-1
-1	1	-1
-1	1	-1

Filter 2

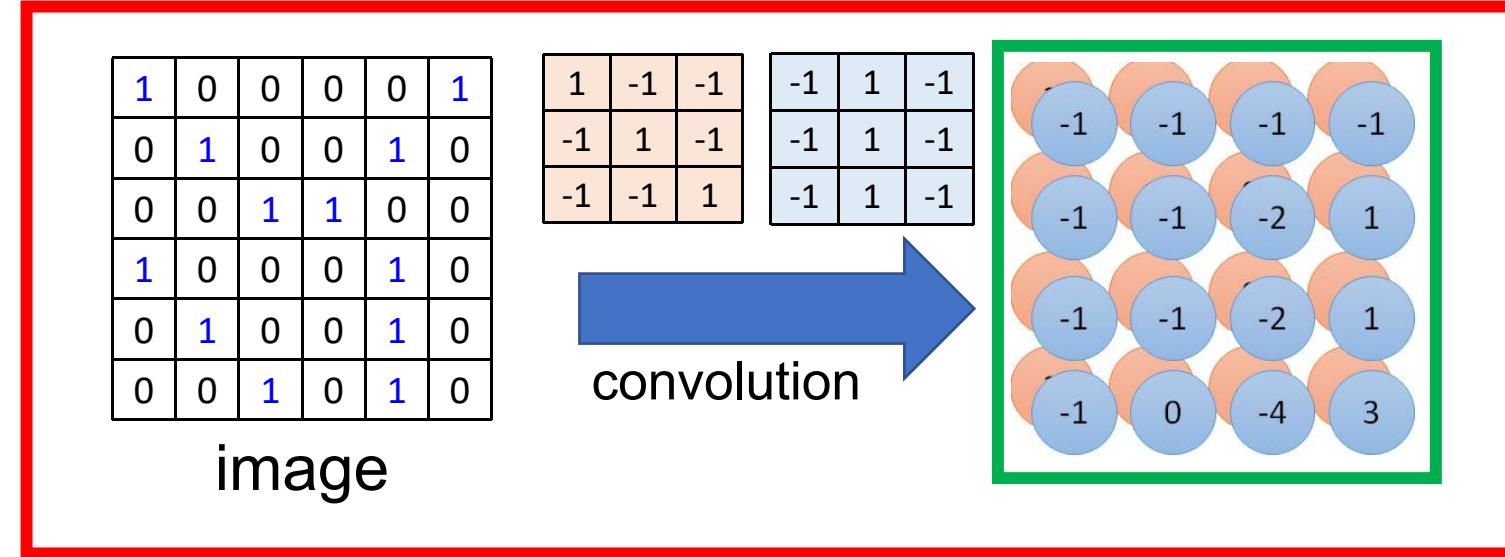
Do the same process
for every filter



CNN – Colorful image



Convolution v.s. Fully Connected

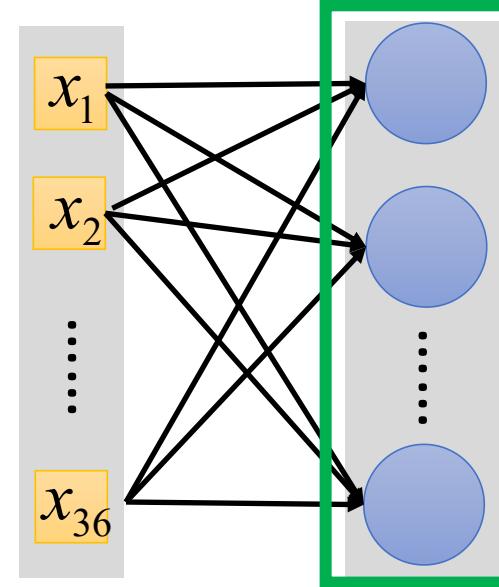


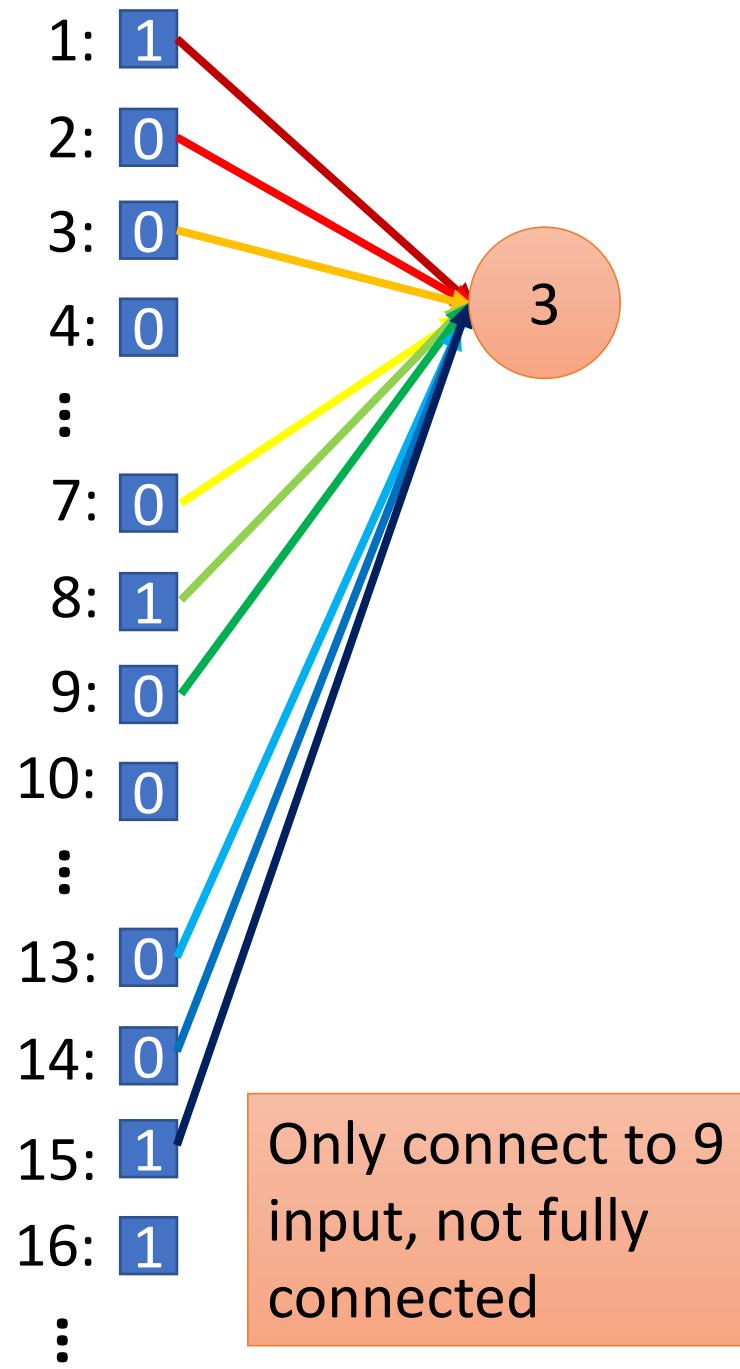
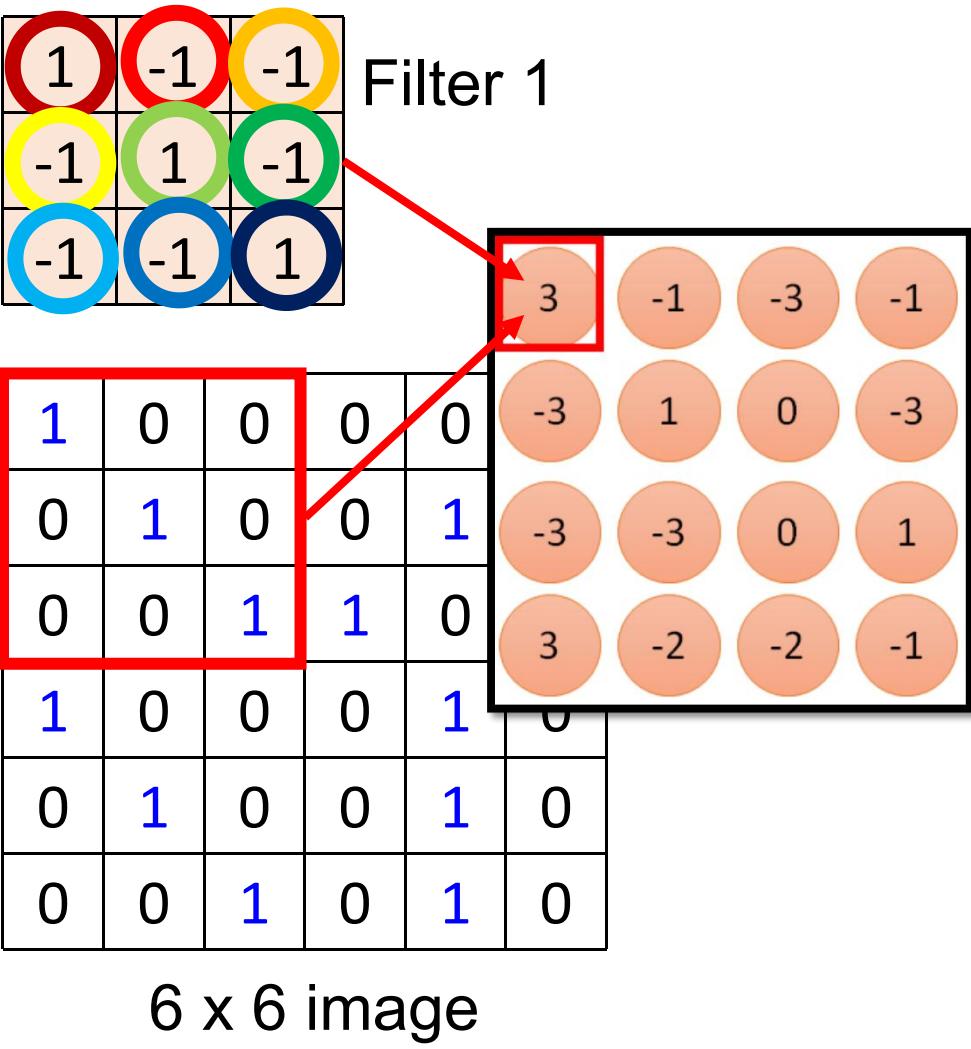
image

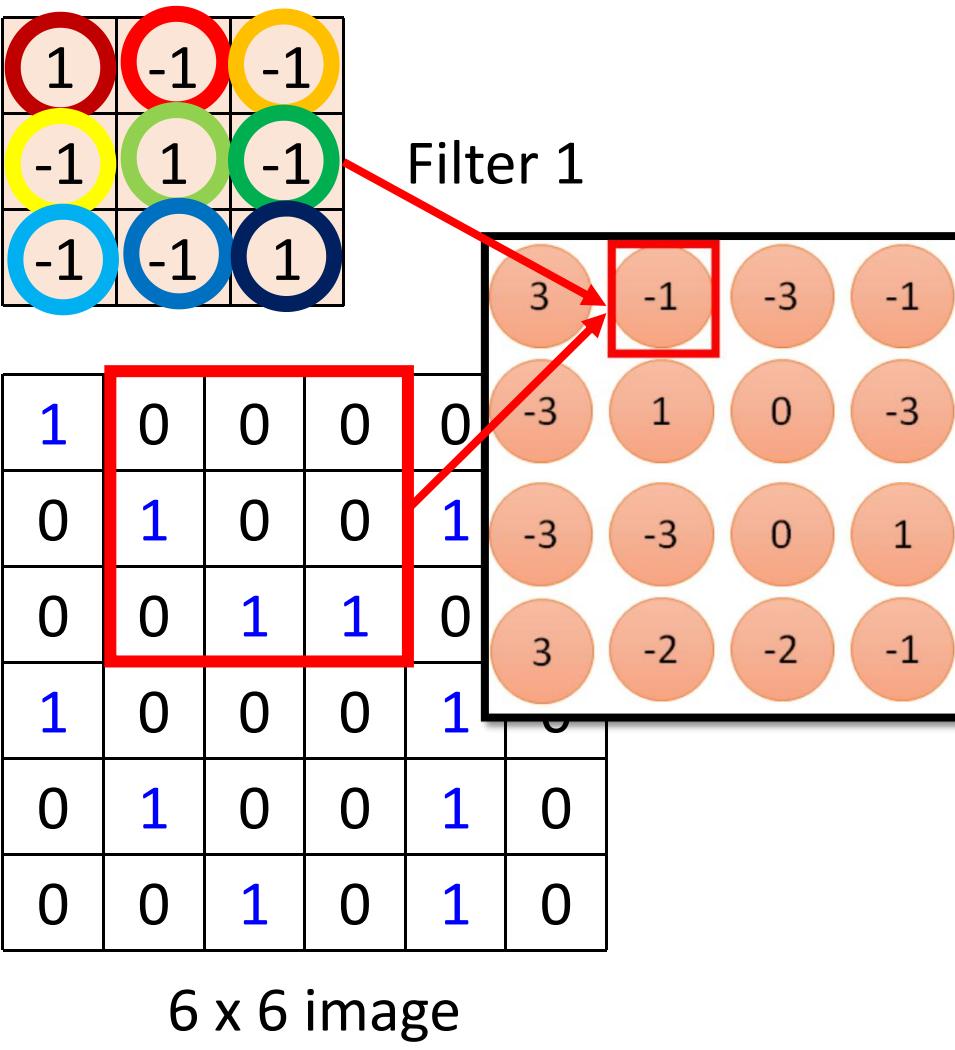
M^*9

Fully-connected

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0



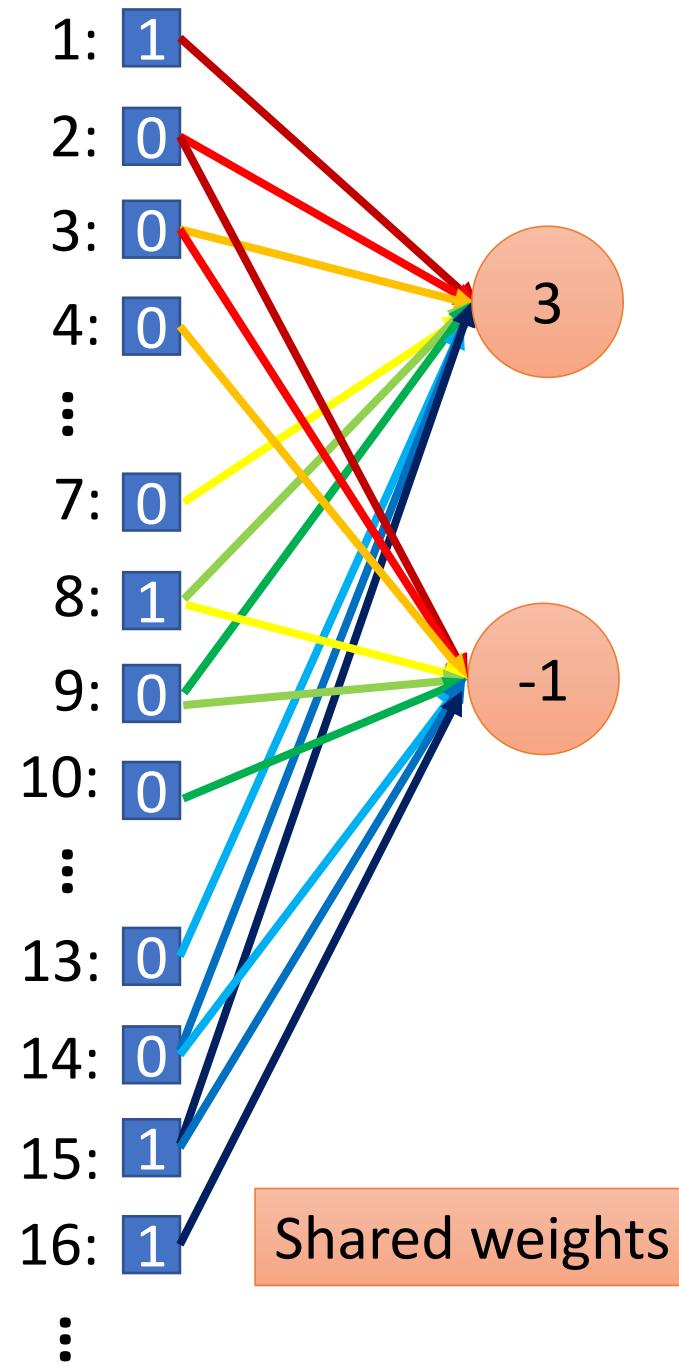




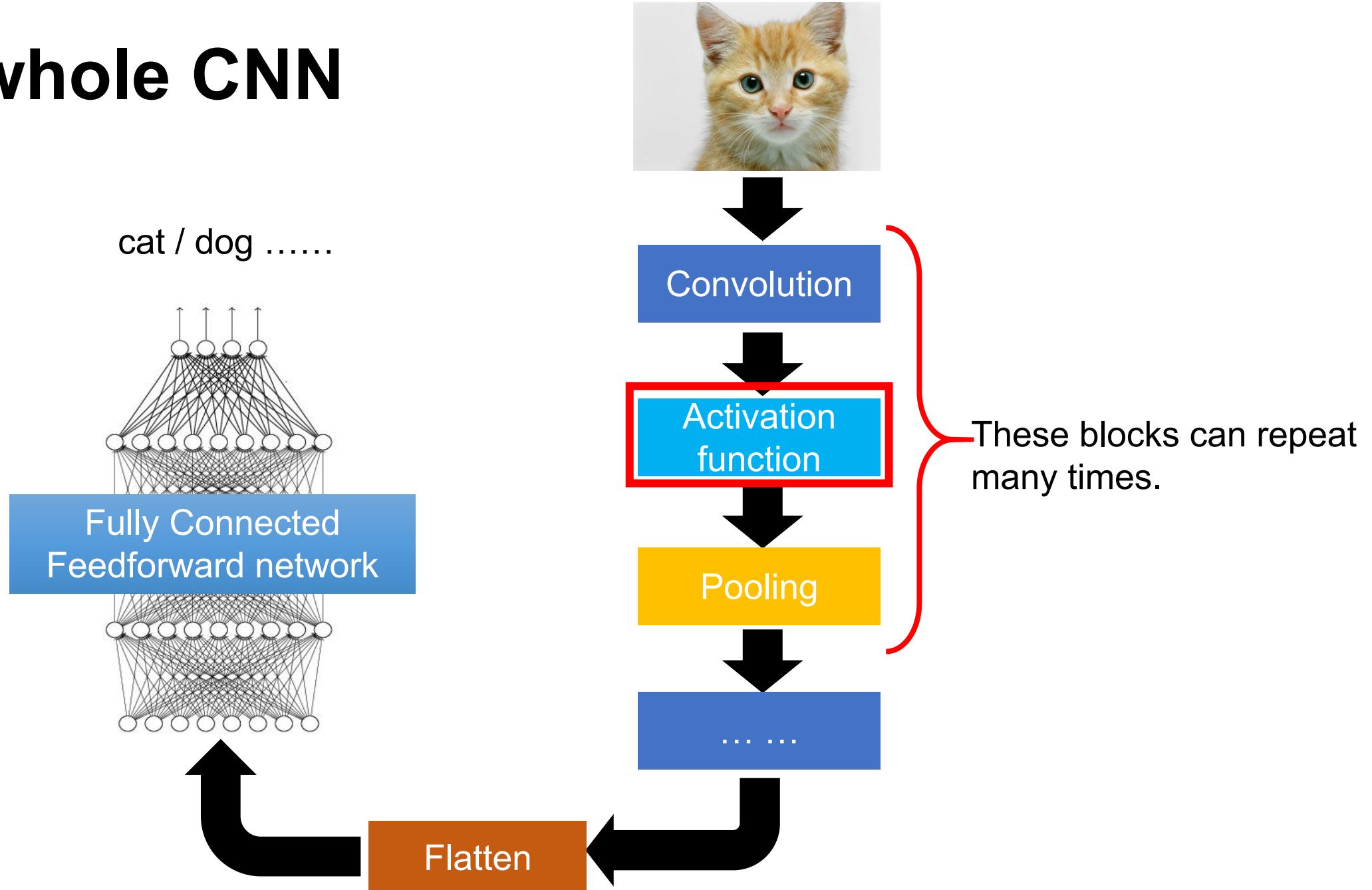
6×6 image

Less parameters!

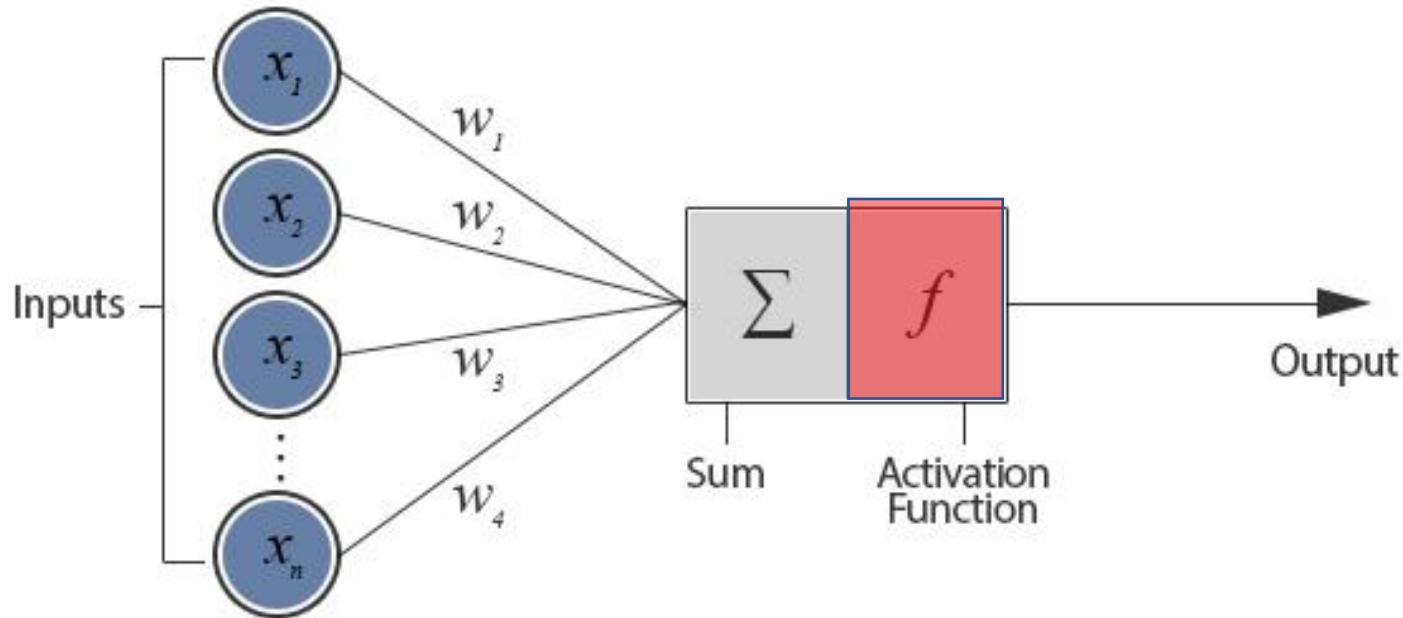
Even less parameters!



The whole CNN



Activation function



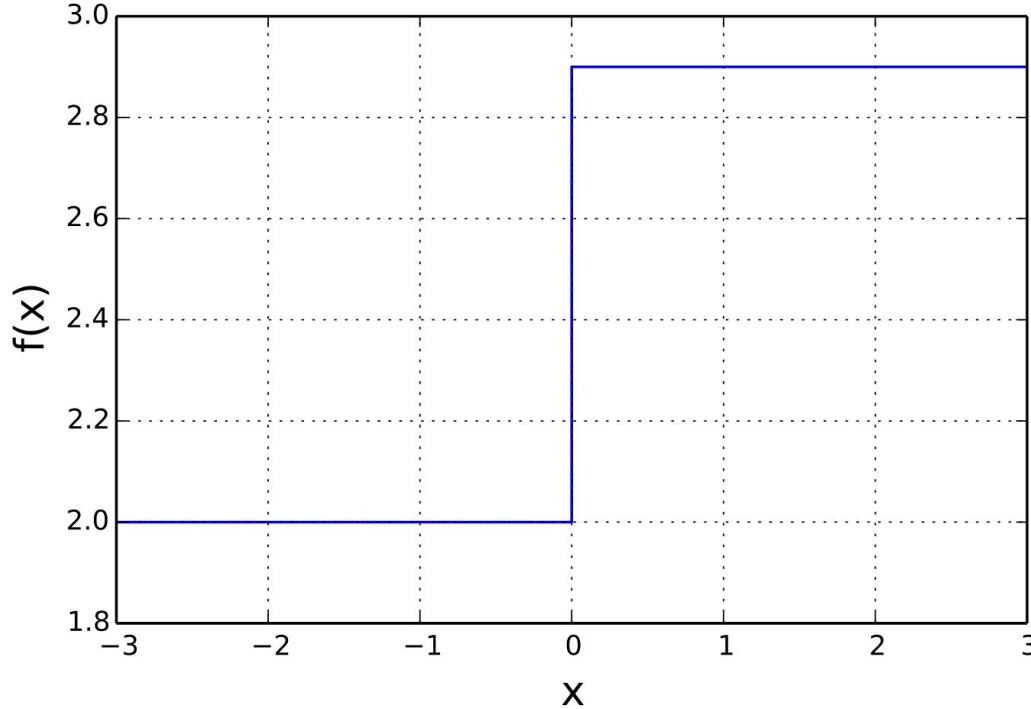
Some Activation Functions

1. Binary Step Function
2. Sigmoid / Logistic
3. TanH / Hyperbolic Tangent
4. Softmax
5. ReLU (Rectified Linear Unit)
6. Leaky ReLU
7. Parametric ReLU
8. ...

Activation function decides **whether a neuron should be activated or not**, by calculating weighted sum and further adding bias with it.

The purpose of the activation function is to introduce **non-linearity** into the output of a neuron.

Activation function – Binary step function



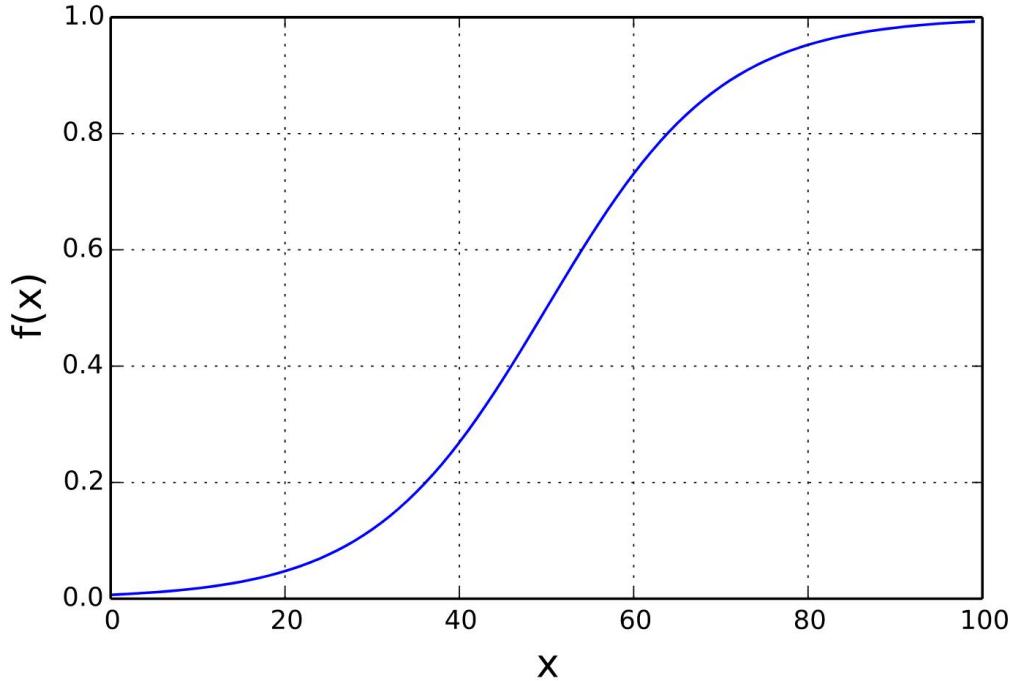
$$a_j^i = f(z_j^i) = \begin{cases} 0 & \text{if } z_j^i < \text{threshold} \\ 1 & \text{if } z_j^i > \text{threshold} \end{cases}$$

A binary step function is generally used in the **Perceptron** linear classifier.

This activation function is useful when the input pattern can only belong to one or two groups, that is, **binary classification**.

The problem with a step function is that **it does not allow multi-value outputs** —for example, it cannot support classifying the inputs into one of several categories.

Activation function – Sigmoid / logistic



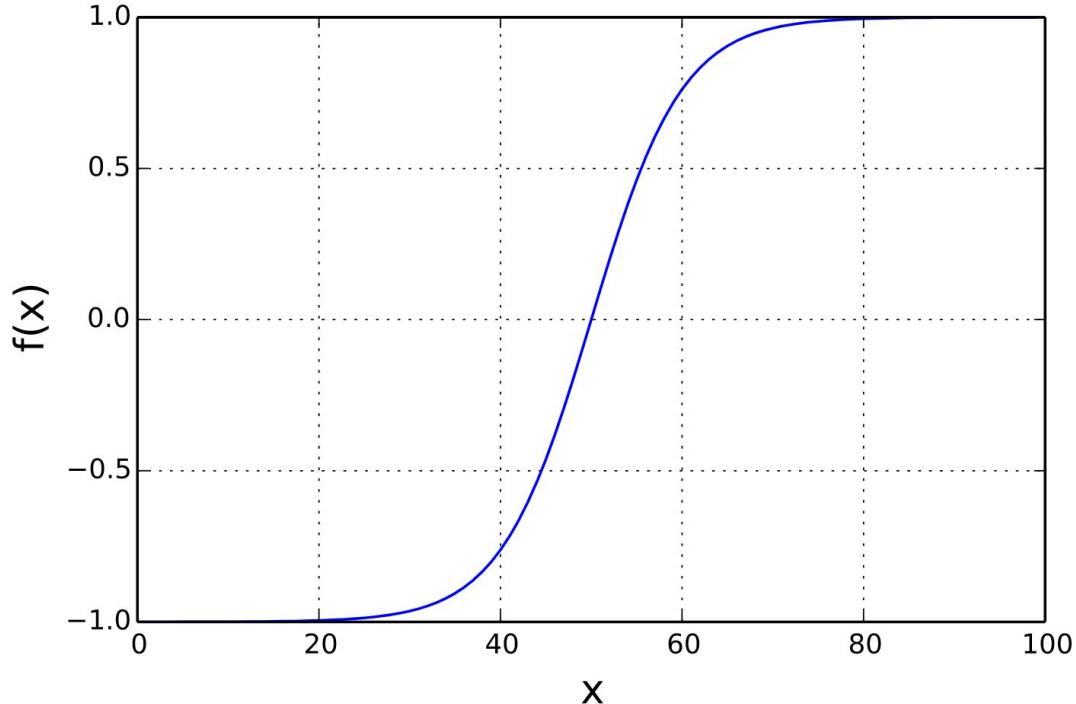
$$a_j^i = f(x_j^i) = \frac{1}{1 + \exp(-x_j^i)}$$

The **sigmoid** or **logistic** activation function maps the input values in the range **(0,1)**, which is essentially their probability of belonging to a class.

It is mostly used for **binary-class classification**.

However, it suffers from the vanishing gradient problem. Also, its output is **not zero-centered**, which causes difficulties during the optimization step. It also has a low convergence rate.

Activation function – tanh



$$a_j^i = f(x_j^i) = \tanh(x_j^i)$$

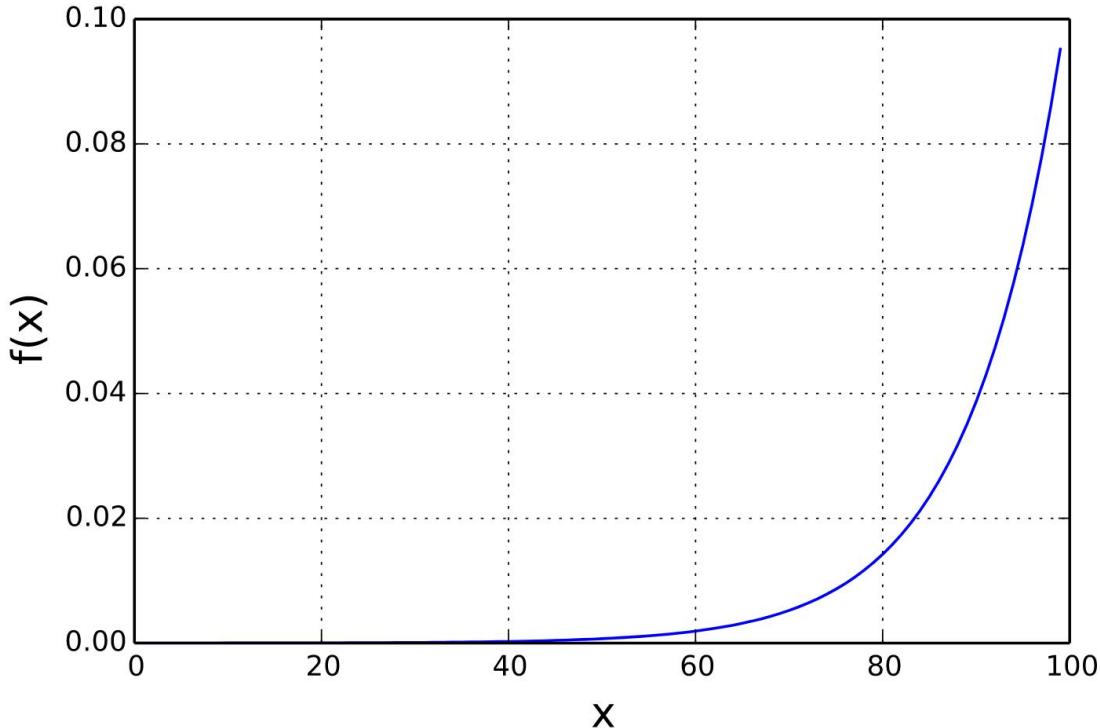
$$\tanh(x) = 2\sigma(2x) - 1$$

The **tanh** non-linearity compresses the input in the range **(-1, 1)**.

It provides an output which is **zero-centered**.

The gradients for tanh are steeper than sigmoid, but it suffers from the [vanishing gradient problem](#).

Activation function – softmax



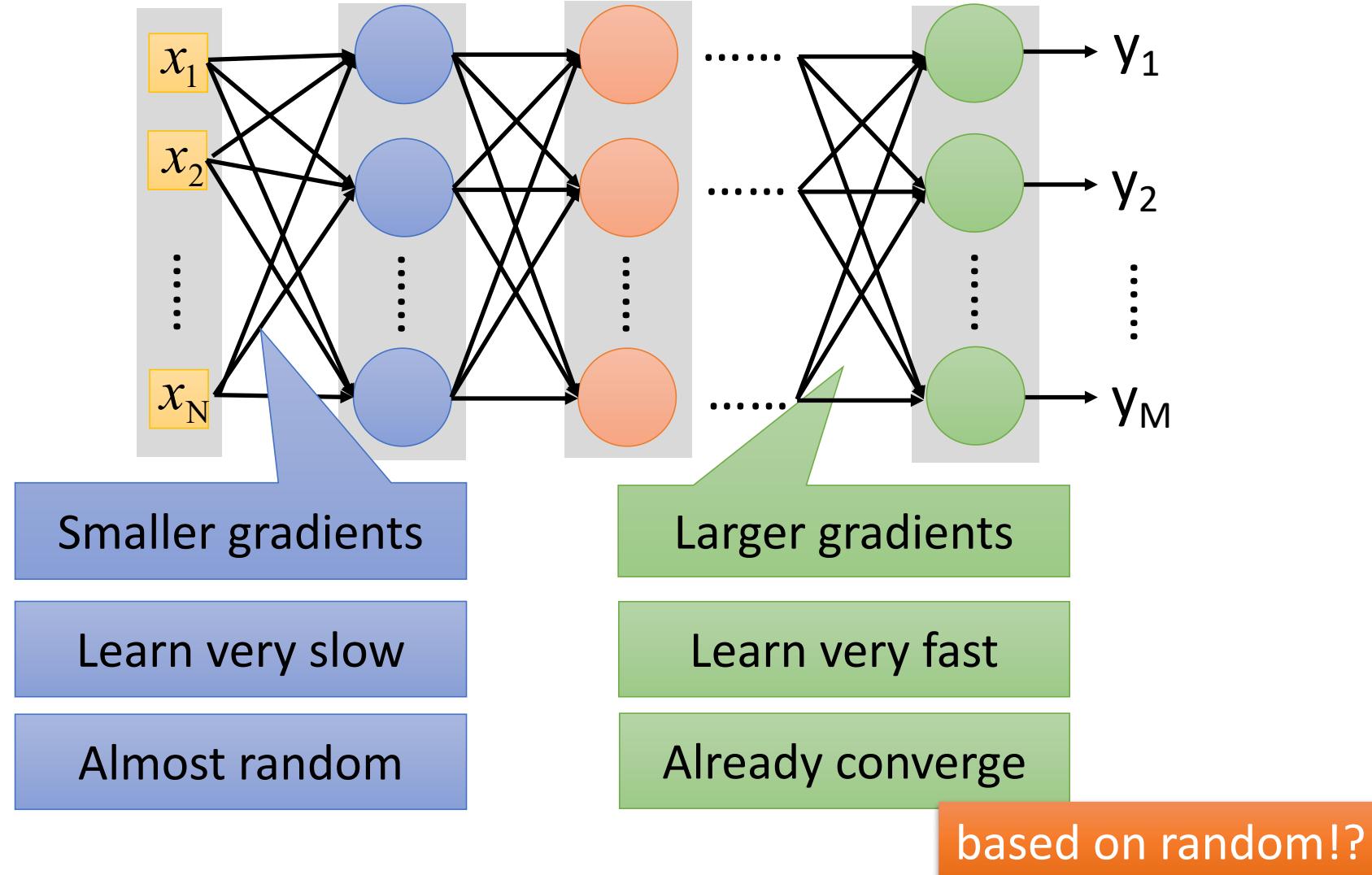
$$a_j^i = f(x_j^i) = \frac{\exp(z_j^i)}{\sum_k \exp(z_k^i)}$$

The **softmax** function gives us the probabilities that any of the classes are true. It produces values in the range $(0,1)$. Its resulting values always **sum to 1**.

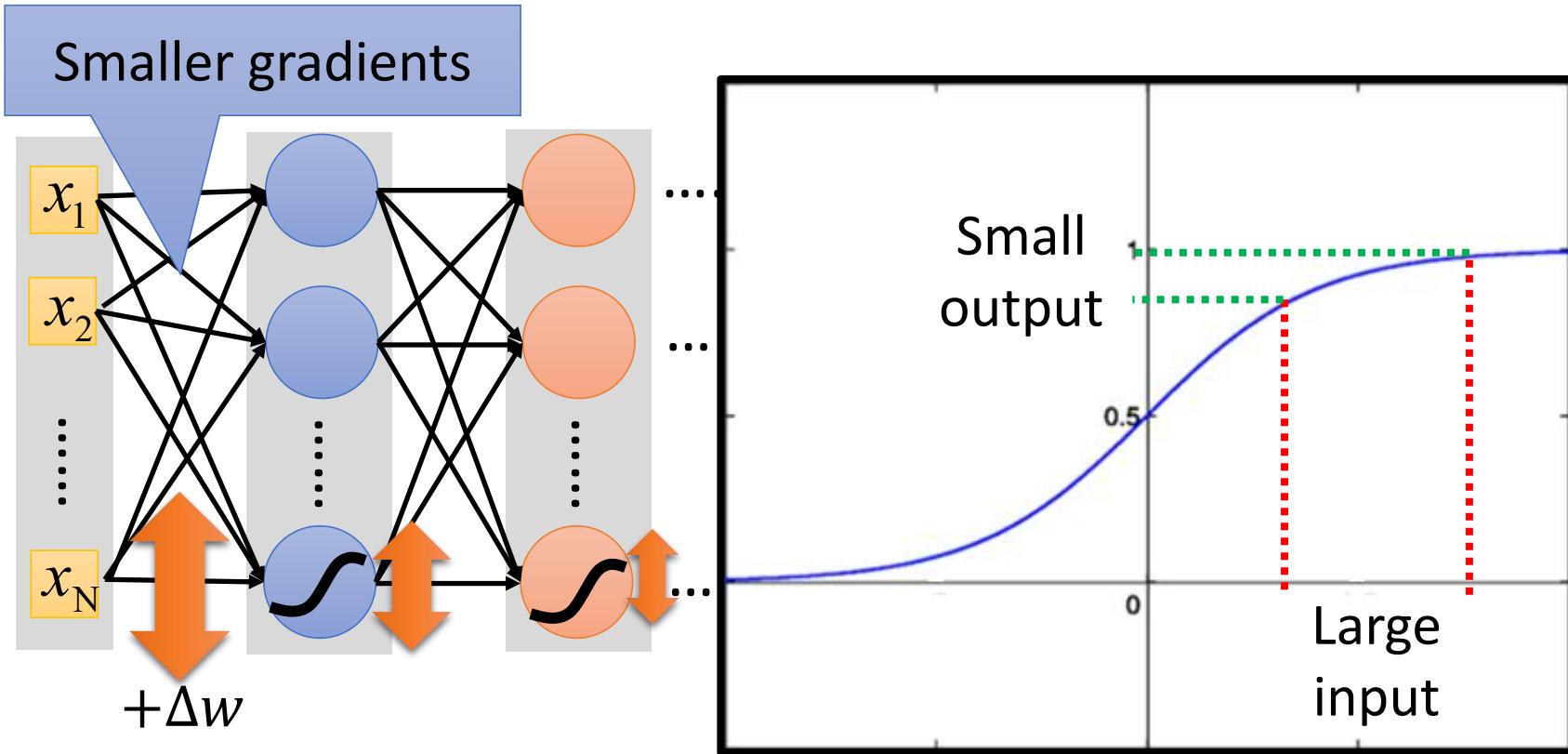
It **highlights** the largest value and tries to **suppress** values which are below the maximum value.

This function is widely used in **multiple classification / logistic regression models**.

Vanishing Gradient Problem



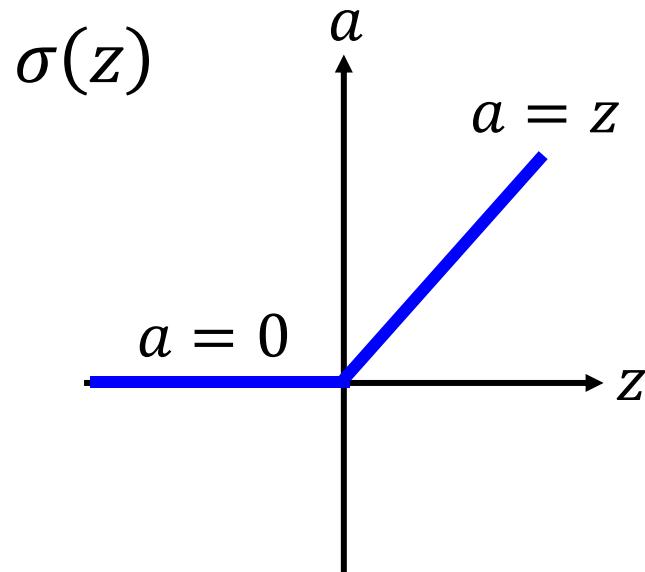
Vanishing Gradient Problem



Intuitive way to compute the derivatives ...

$$\frac{\partial l}{\partial w} = ? \frac{\Delta l}{\Delta w}$$

Activation function – ReLU



[Xavier Glorot, AISTATS'11]

[Andrew L. Maas, ICML'13]

[Kaiming He, arXiv'15]

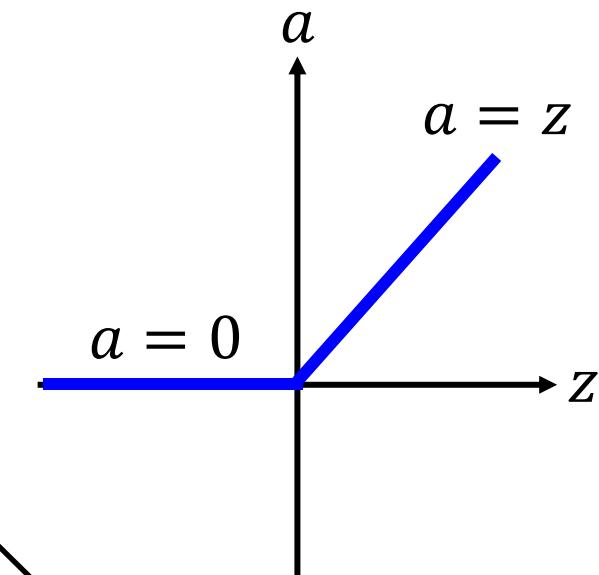
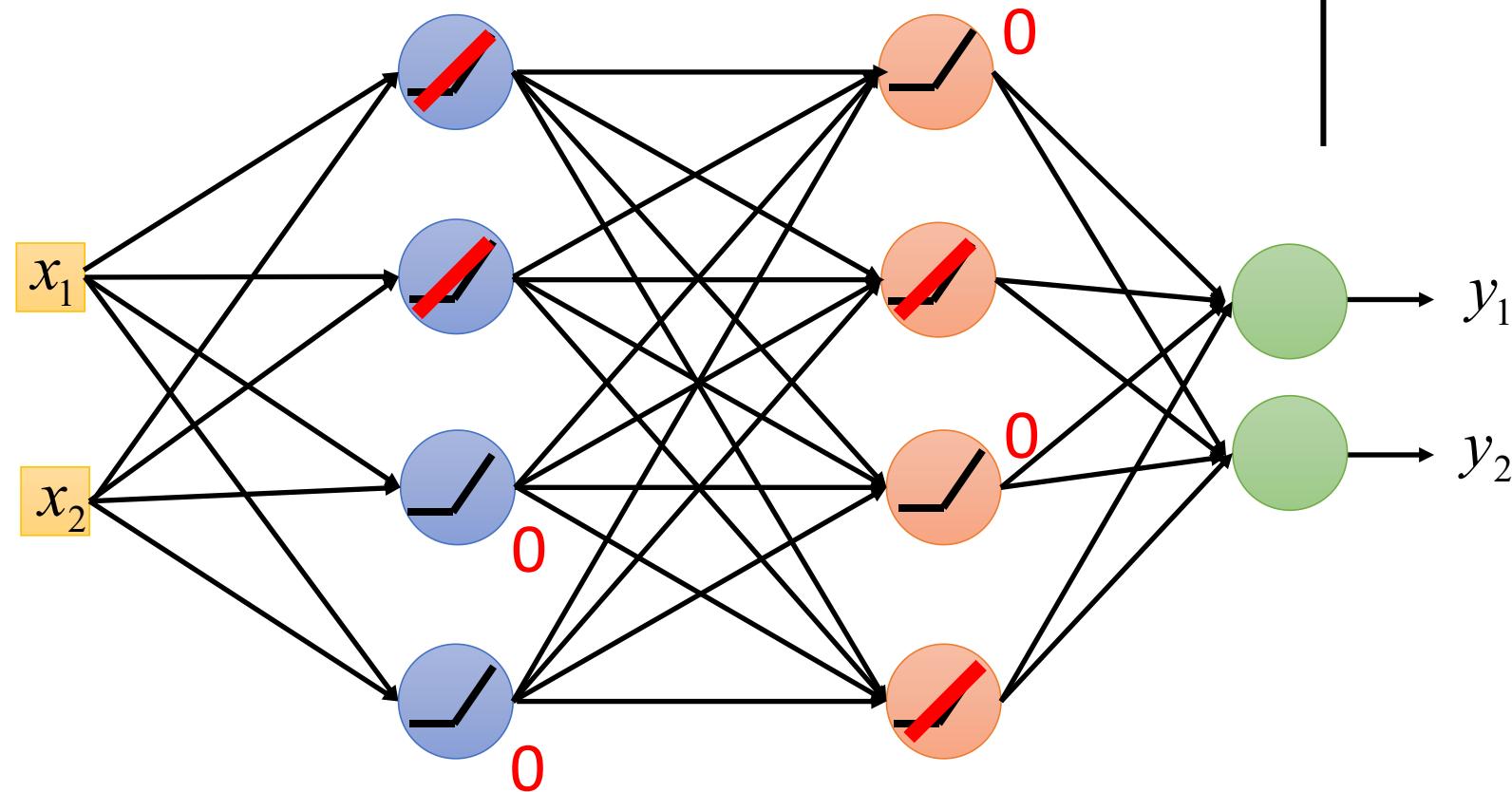
$$a_j^i = f(x_j^i) = \max(0, x_j^i)$$

Reason:

1. Fast to compute
2. Biological reason?
4. Vanishing gradient problem

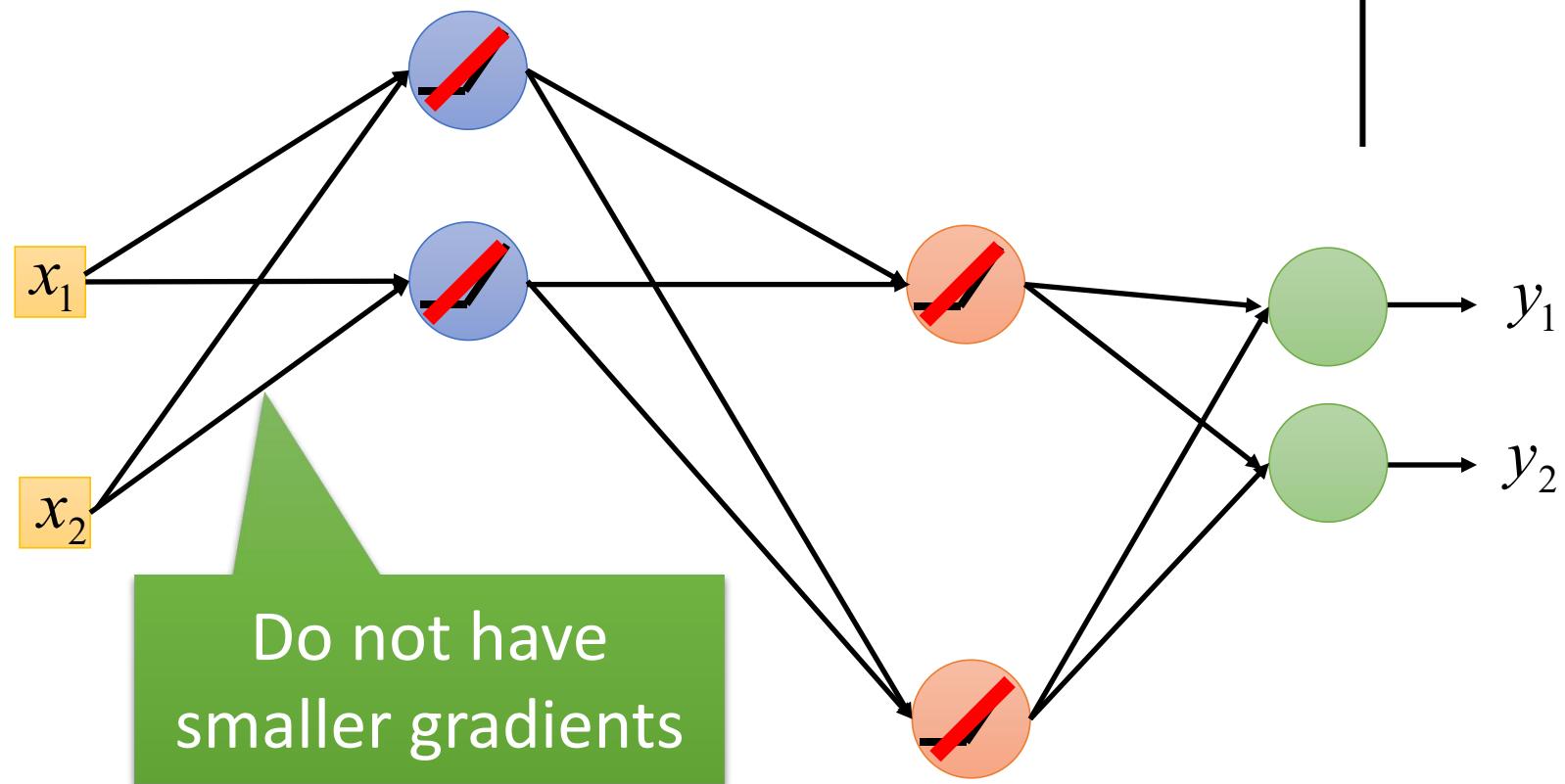
Rectified Linear Unit (ReLU)

ReLU



ReLU

A Thinner linear network



“Dying ReLU” problem

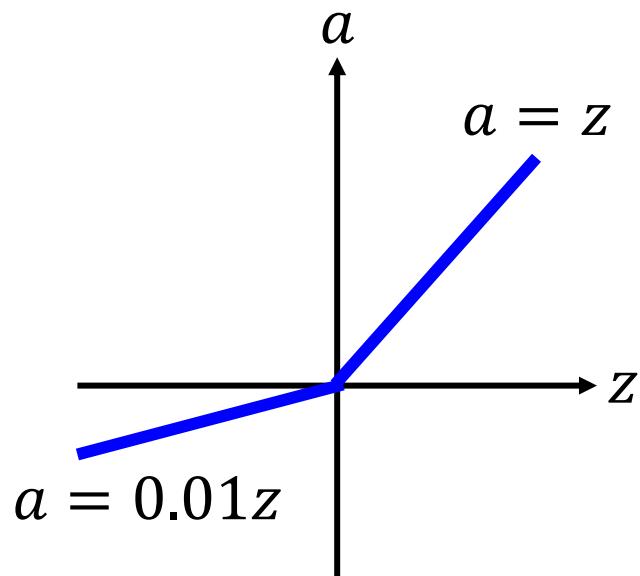
Being **non-differentiable at 0**, ReLU neurons have the tendency to become **inactive** for all inputs, that is, they tend to die out.

This can be caused by **high** learning rates, and can thus reduce the model’s learning capacity.

This is commonly referred to as the **“Dying ReLU” problem**.

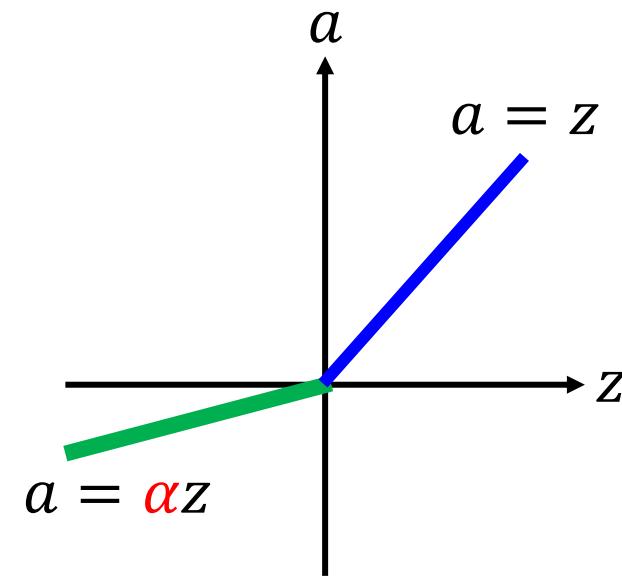
ReLU - variants

Leaky ReLU



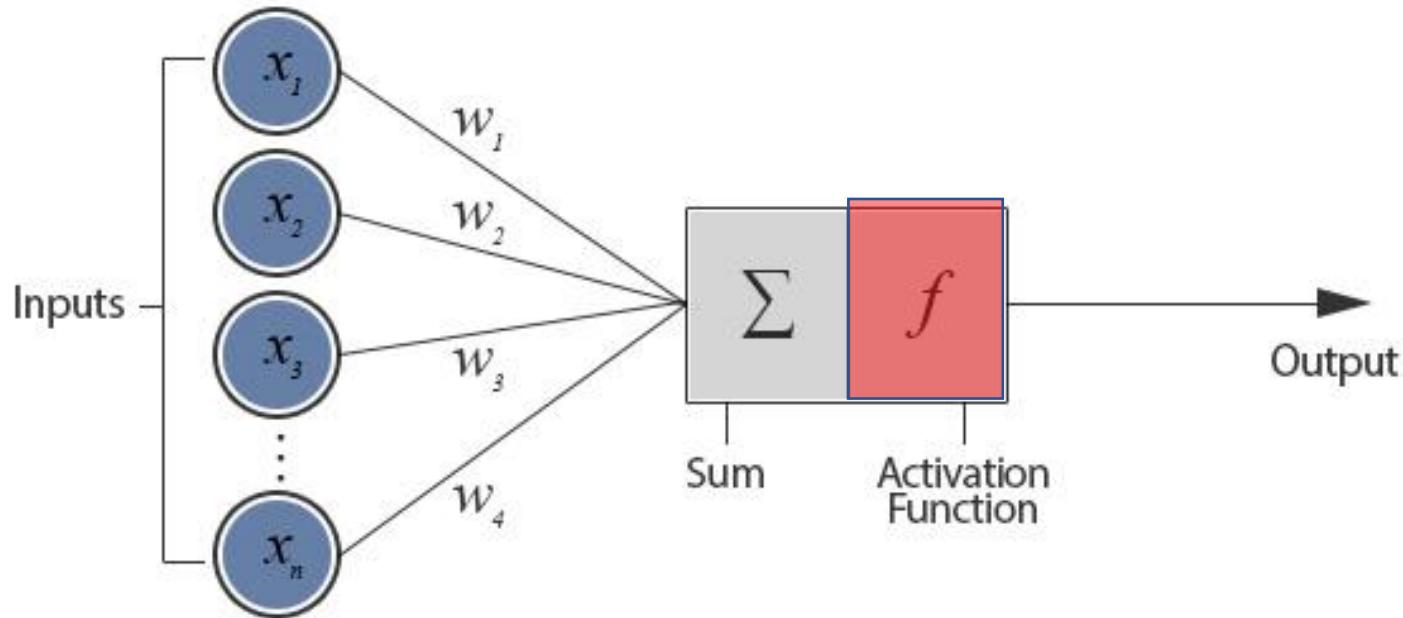
$$a_j^i = f(x_j^i) = \max(0.01x_j^i, x_j^i)$$

Parametric ReLU



α can be free parameter.
Be also learned by gradient descent

Activation function

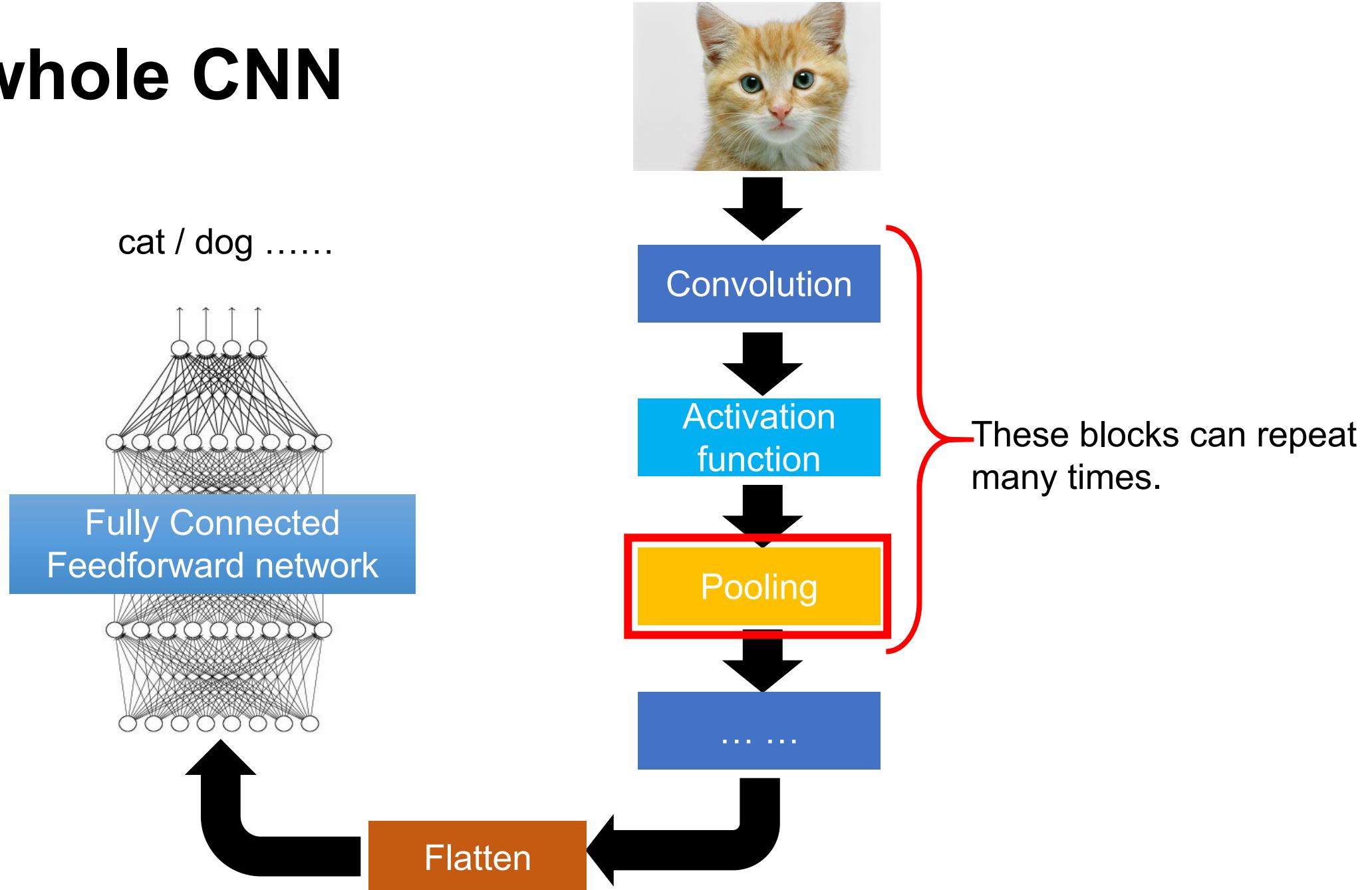


Some Activation Functions

1. Binary Step Function
2. Sigmoid / Logistic
3. TanH / Hyperbolic Tangent
4. Softmax
5. ReLU (Rectified Linear Unit)
6. Leaky ReLU
7. Parametric ReLU
8. ...

A review on the activation functions and their biological plausibility?

The whole CNN



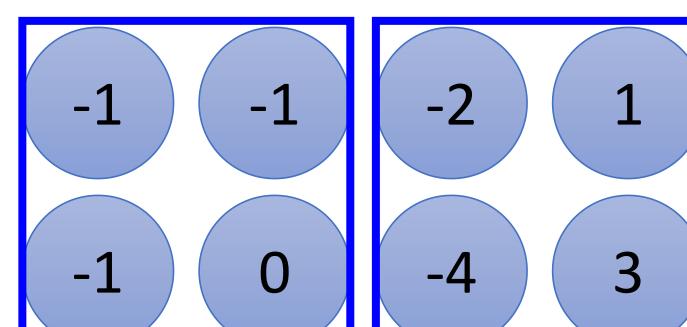
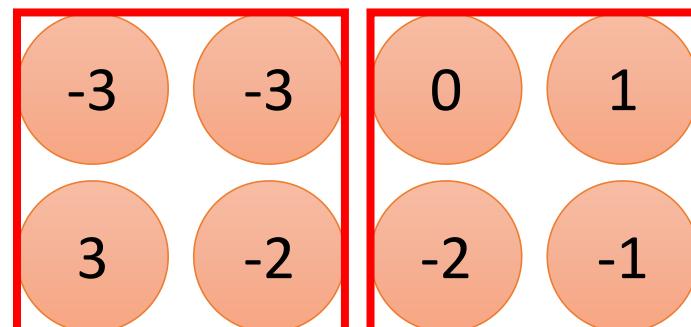
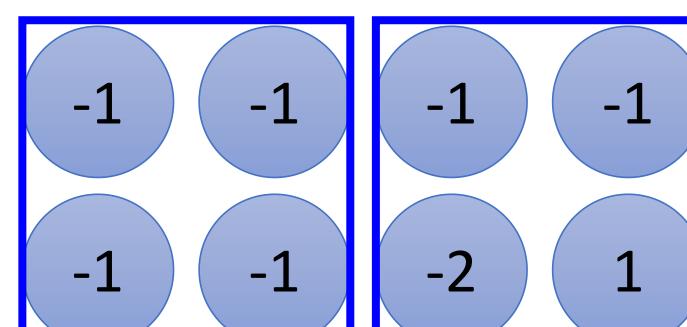
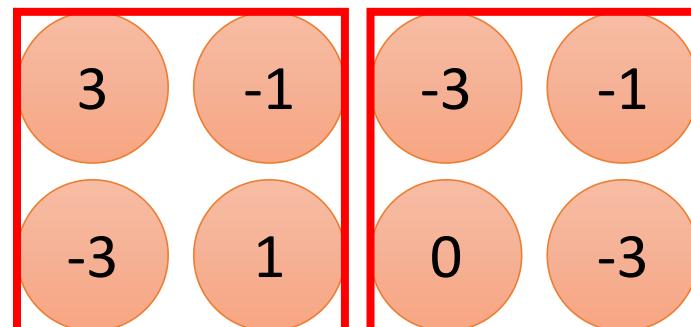
CNN – Max Pooling

1	-1	-1
-1	1	-1
-1	-1	1

Filter 1

-1	1	-1
-1	1	-1
-1	1	-1

Filter 2



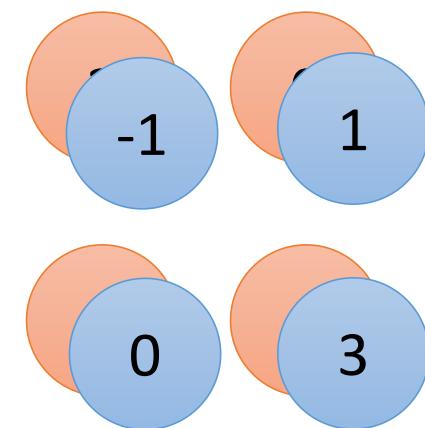
CNN – Max Pooling

1	0	0	0	0	1
0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	0
0	1	0	0	1	0
0	0	1	0	1	0

6 x 6 image



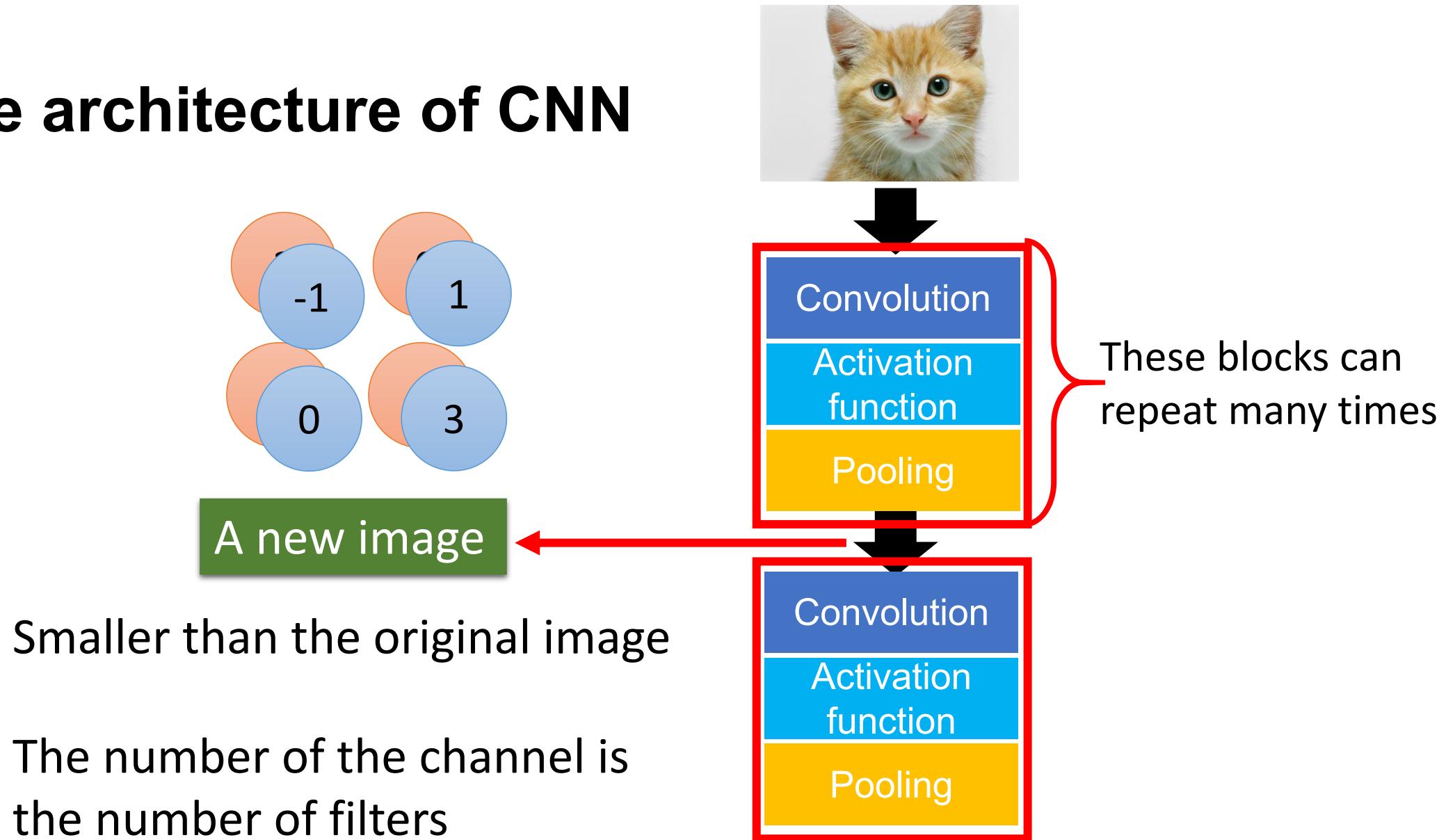
New image
but smaller



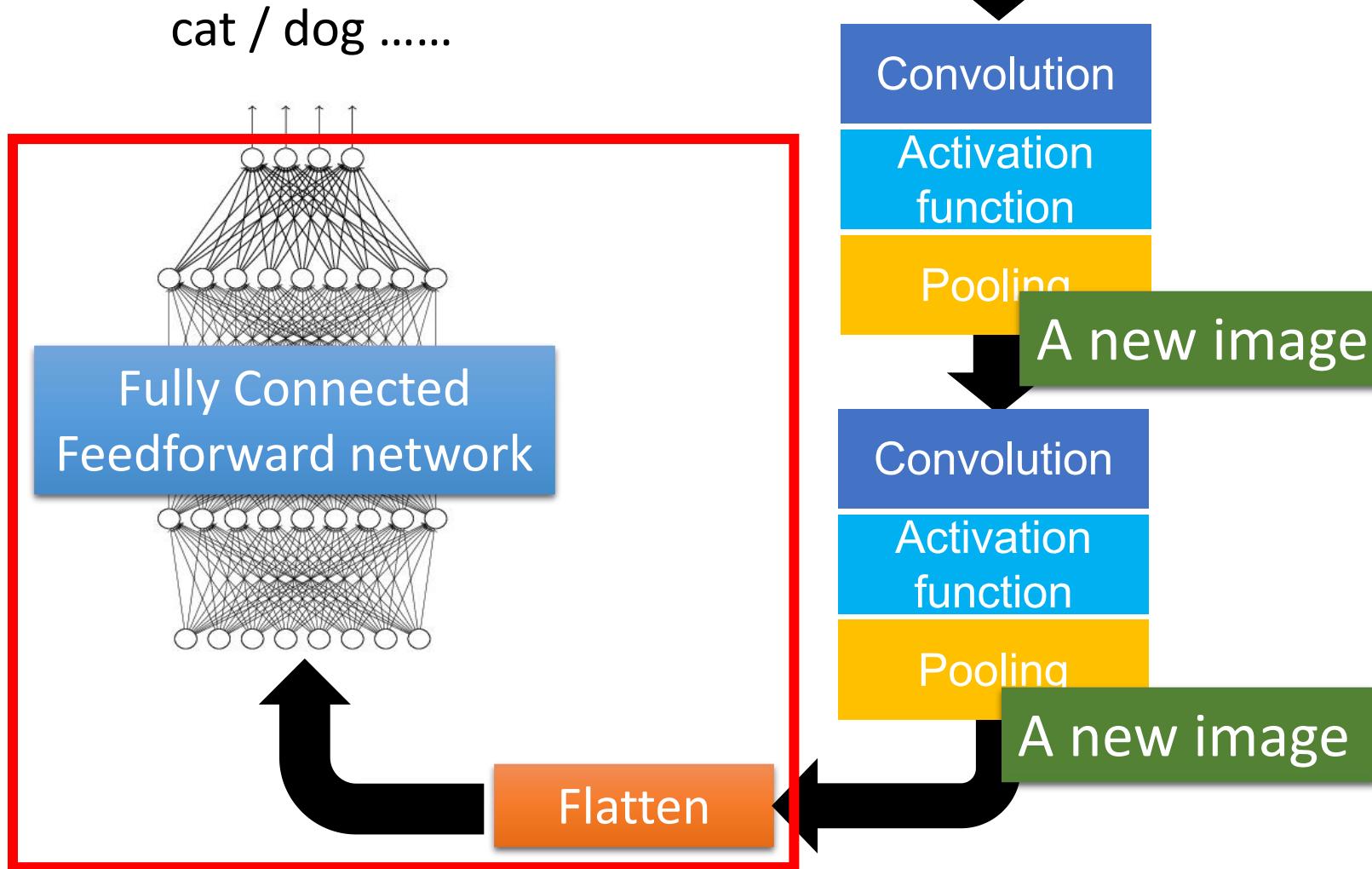
2 x 2 image

Each filter
is a channel

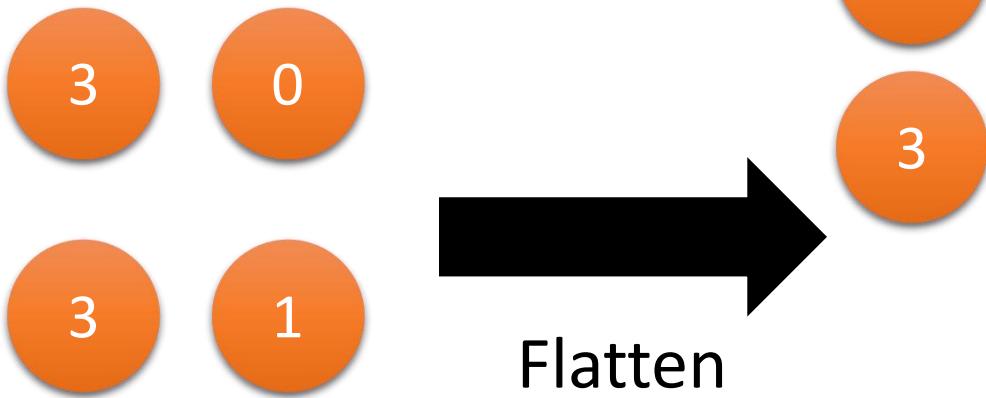
The architecture of CNN



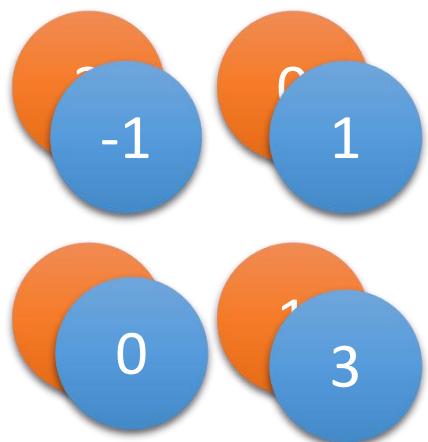
The architecture of CNN



Flatten



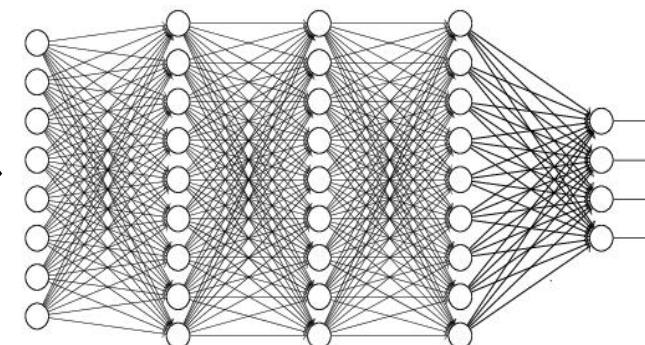
Flatten



Flatten



FC



Fully Connected
Feedforward network

Summary of Lecture 15 – GD & BP & CNN

- Gradient Descent (GD)
 - What is Gradient Descent?
 - Gradient Descent to train deep NNs → Error Backpropagation
- Error Back-propagation (BP)
 - Backpropagation
 - Backpropagation – forward pass
 - Backpropagation – backward pass
- The Architecture of CNN
 - Convolution
 - Activation function
 - Pooling
 - Flatten
 - FC
- CNN Hands-on (tensorflow) → next lecture