(Deep) Neural Networks
Summary

Yin Li

yin.li@wisc.edu

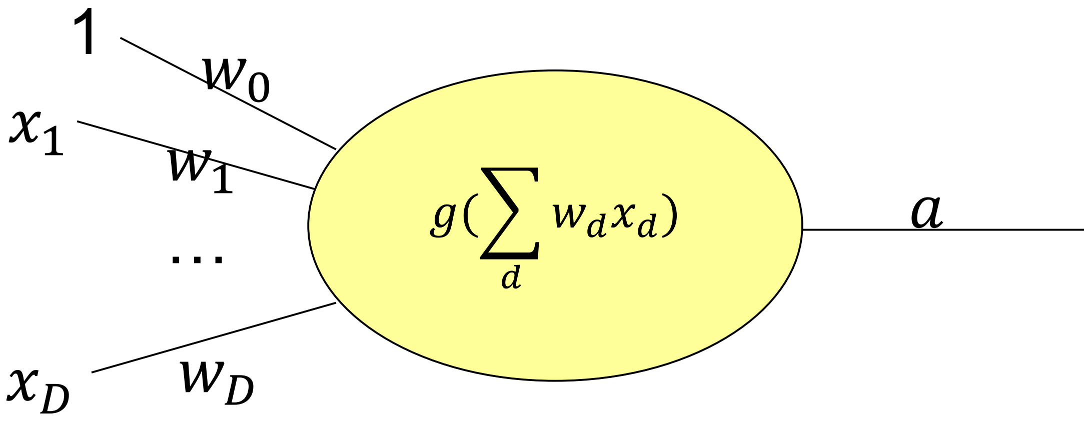
University of Wisconsin, Madison

Key concepts of (deep) neural networks

- Modeling a single neuron
  - Linear / Nonlinear Perception
  - Limited power of a single neuron
- Connecting many neurons
  - Neural networks
- Training of neural networks
  - Loss functions
  - Backpropagation on a computational graph
- Deep neural networks
  - Convolution
  - Activation / pooling
  - Design of deep networks

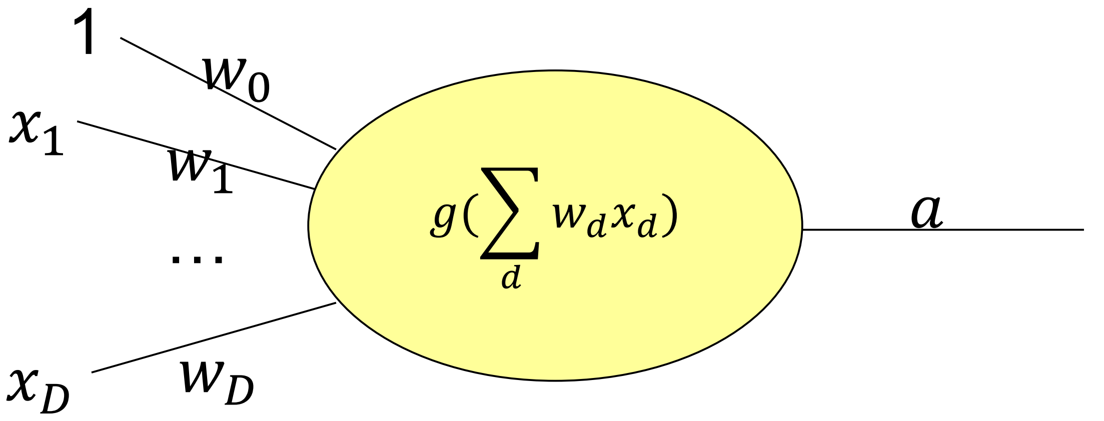
Modeling a single neuron

- Perceptron: $a = g(\sum_d w_d x_d)$
- Activation function g: identity, sigmoid, ReLU

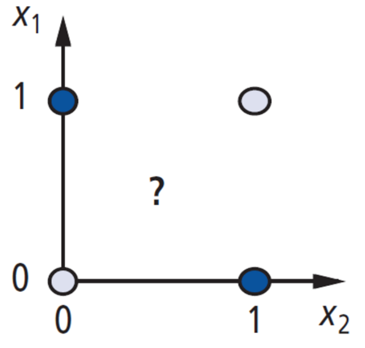


Limited power of one single neuron

- Perceptron: $a = g(\sum_d w_d x_d)$
- Activation function g: identity, sigmoid, ReLU

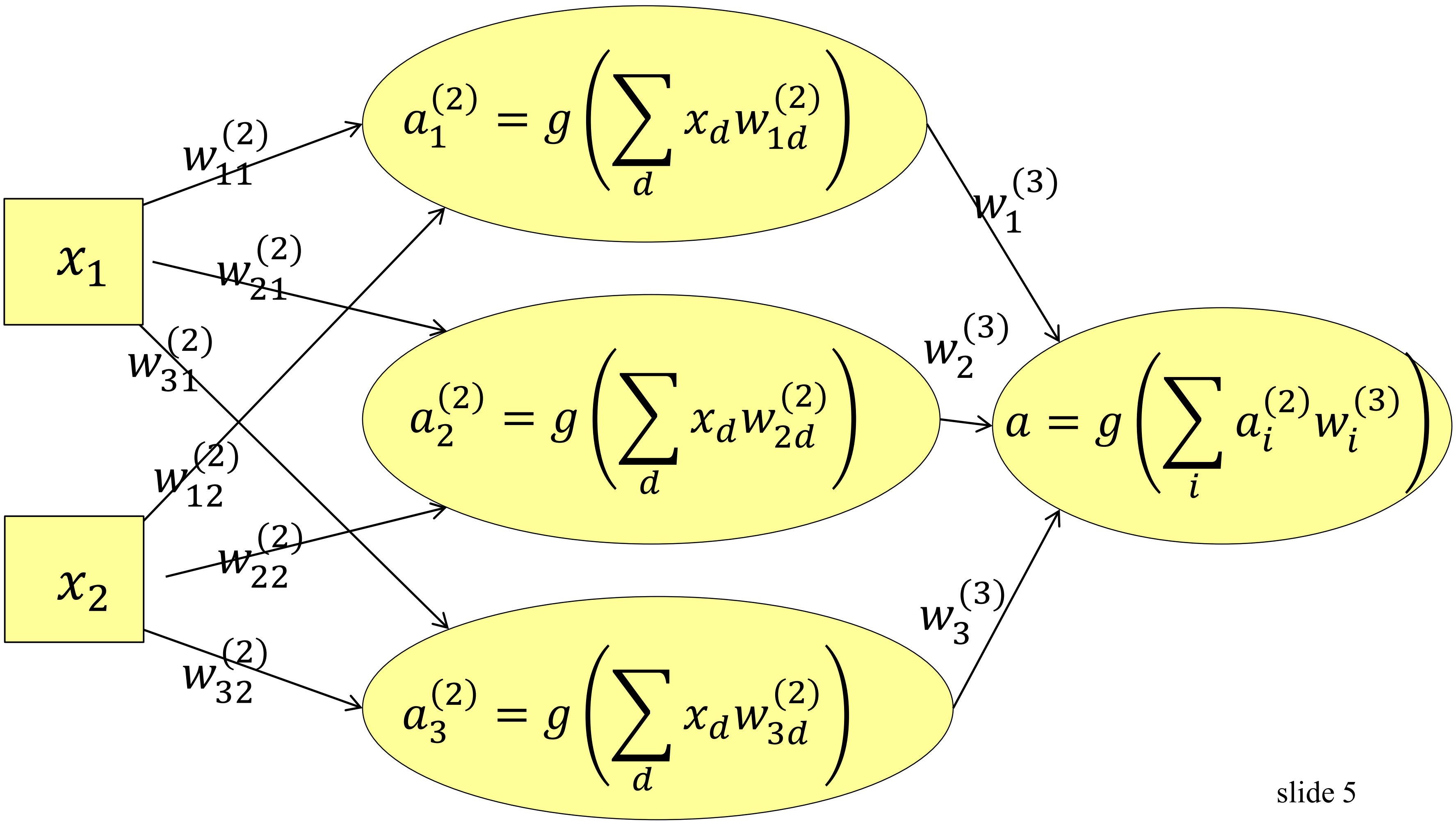


- Decision boundary linear even for nonlinear g
- Can not handle XOR problem



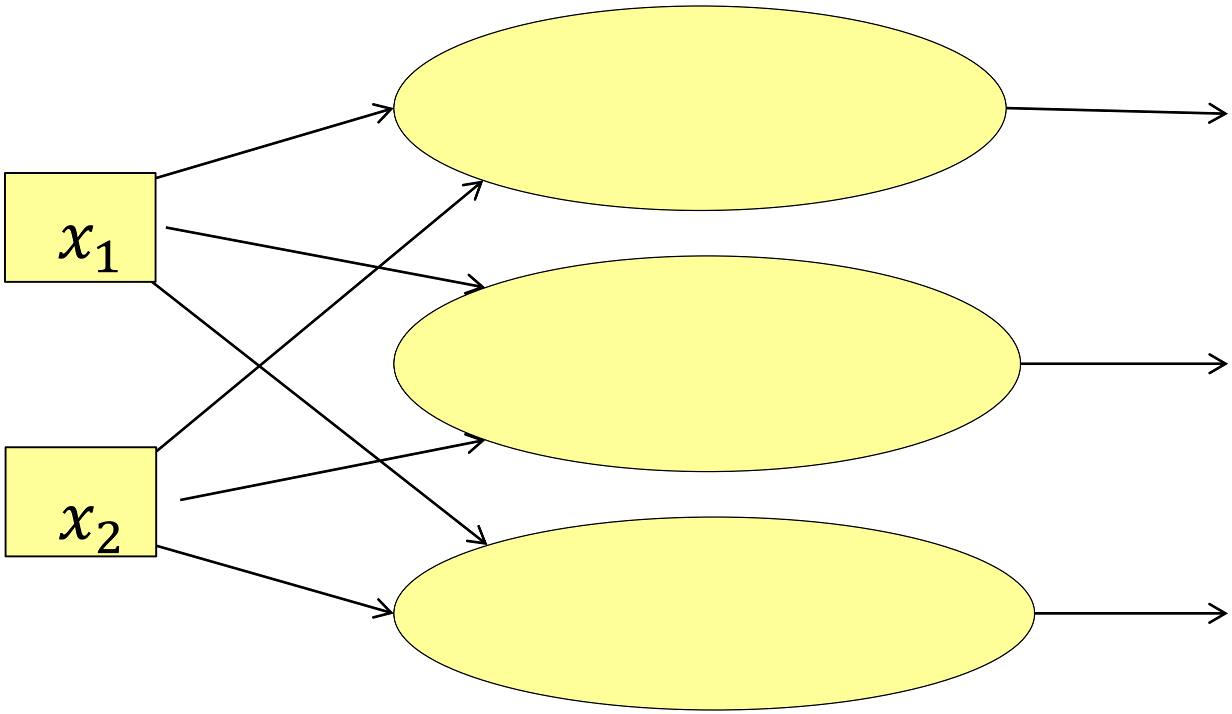
Connecting many neurons: multilayer perceptron

- Standard way to connect Perceptrons
- Example: 1 hidden layer, 1 output layer, depth = 2



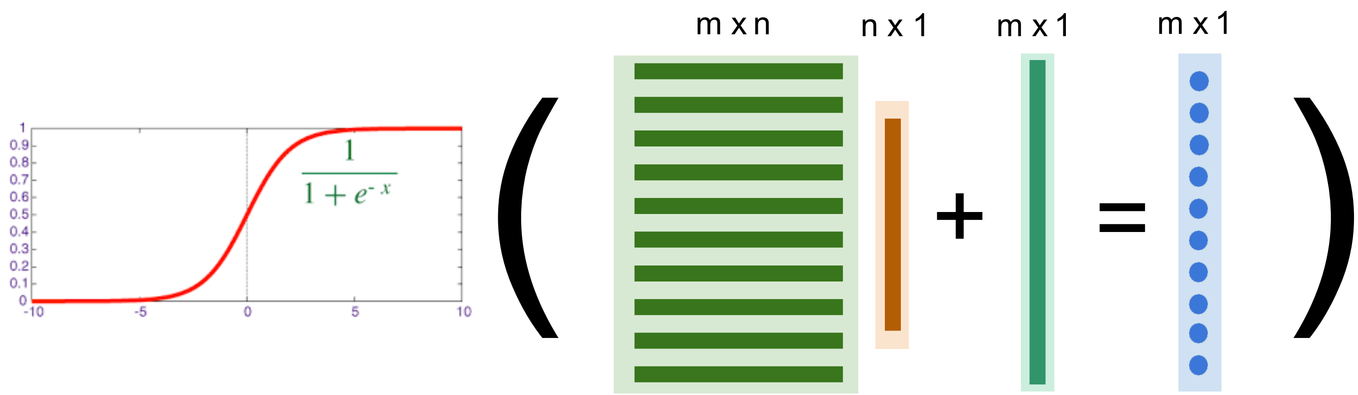
A single layer in neural networks

- $\boldsymbol{a} \quad = g\left(\boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{x} + \boldsymbol{b}\right)$



A single layer in neural networks

- $\boldsymbol{a} \quad = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b})$
- Work for any element-wise activation function g
- Work for any number of neurons
- Map an input $\boldsymbol{x} \in R^n$ to an output $\boldsymbol{a} \in R^m$
- $\boldsymbol{x} \in R^n, \ \boldsymbol{W} \in R^{n \times m}, \ \boldsymbol{b} \in R^m, \ \boldsymbol{a} \in R^m$
- Also called a fully connected layer



Neural Networks

- What type of functions shall we consider for *f?*

Features
+ Decision
$f(x;\ \theta)$

Chair

Proposal: Composing a set of (nonlinear) functions *g*

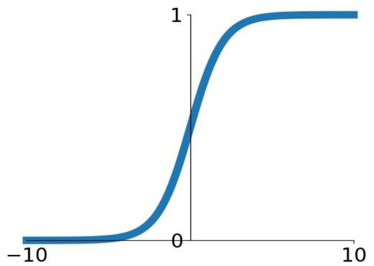
$$f(\boldsymbol{x};\boldsymbol{\theta}) = g_1(\ldots g_{n-1}(g_n(\boldsymbol{x};\boldsymbol{\theta_n}),\boldsymbol{\theta_{n-1}})\ldots,\boldsymbol{\theta_1})$$

Example: $\mathbf{a} = sigmoid\left(\boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{x} + \boldsymbol{b}\right) = g(\boldsymbol{x};\boldsymbol{W},\boldsymbol{b})$

Output normalization: Sigmoid

- Normalize the output into the range of (0,1)
- As a probability distribution for a *binary* variable
- No parameters and differentiable

Sigmoid



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

Output normalization: Softmax

- Normalize a vector such that
  - Each element in the range of (0, 1)
  - All elements sum to 1
- As a probability distribution for a *categorical* variable (e.g., $x = \{1, \ldots K\}$)
- No parameters and differentiable

Softmax

$$softmax(x_k) = \frac{\exp(x_k)}{\sum_j \exp(x_j)}$$

Loss functions

- Classification
  - Cross entropy loss
  - C-way classification problem
  - Often in combination with sigmoid (binary) or softmax (C-way)

$$H(y,p) = -\sum_j y_j \log(p_j)$$

- Regression
  - L2 loss

$$L_2(y,\hat{y}) = \sum_j \left(y_j - \hat{y}_j\right)^2$$

Learning in neural networks

- Define a loss function

$$E = \frac{1}{|D|} \sum_{x \in D} E_x$$

  - x: one training point in the training set D
  - a: the output for the training point x
  - y: the binary label for x
- Optimize all the weights w on all the edges
  - Apparent difficulty: how to update the weights for the hidden units?
  - It turns out to be OK: we can still do gradient descent. The trick you need is the chain rule
  - The algorithm is known as back-propagation

Mini-batch stochastic gradient descent

- Select a learning rate $\alpha > 0$
- Initialize the model parameters (edge weights) $w^{(0)}$
- For $t = 1, 2, \ldots$
  - Randomly sample a subset $\widehat{D}$ from D
  - Compute $\frac{\partial E_x}{\partial w}$ (per sample gradients w.r.t. w) for $\mathrm{x} \in \widehat{D}$ using back-propagation
  - Update the parameters

$$w^{(t)} = w^{(t-1)} - \alpha \frac{1}{|\widehat{D}|} \sum_{x \in \widehat{D}} \frac{\partial E_x}{\partial w}$$

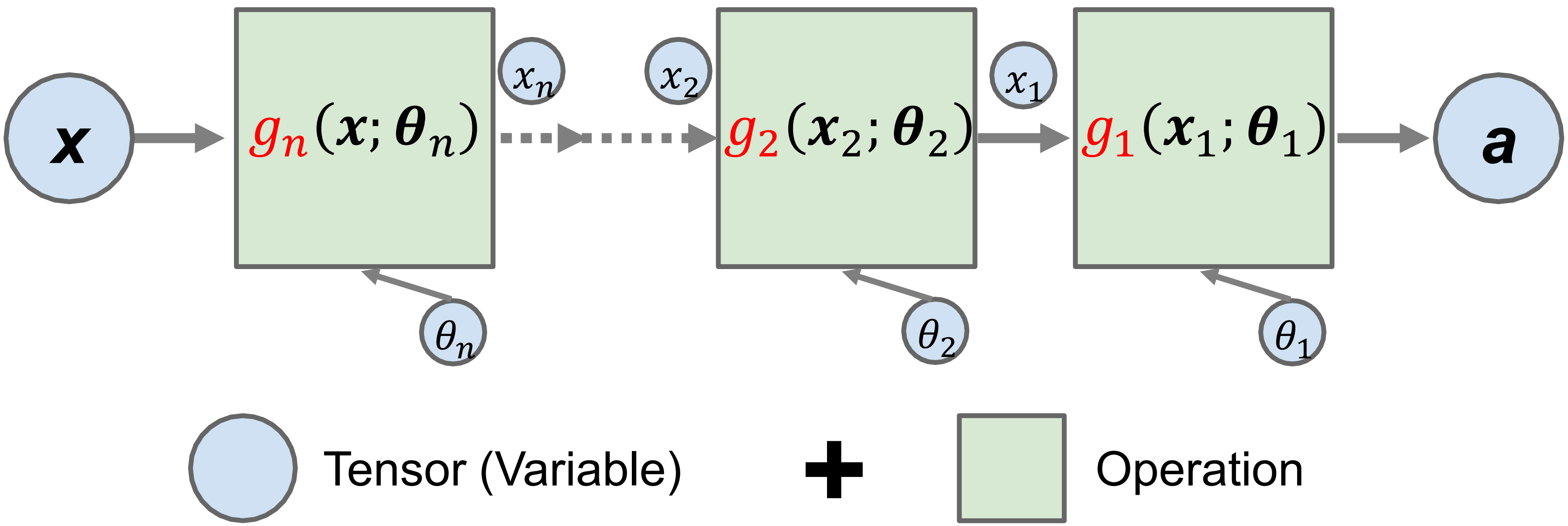
- Repeat until E converges

The key challenge is to compute $\frac{\partial E_x}{\partial w}$!

Neural network as computational graph

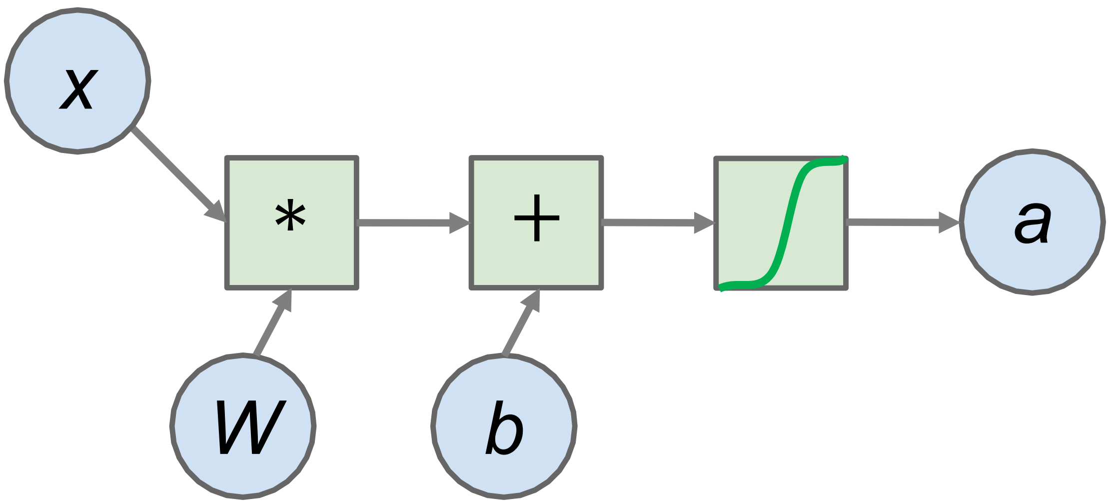
- (Deep) Neural Network:
  Composing a set of (nonlinear) functions *g*

$$f(\boldsymbol{x};\boldsymbol{\theta}) = g_1(\ldots g_{n-1}(g_n(\boldsymbol{x};\boldsymbol{\theta_n}),\boldsymbol{\theta_{n-1}})\ldots,\boldsymbol{\theta_1})$$



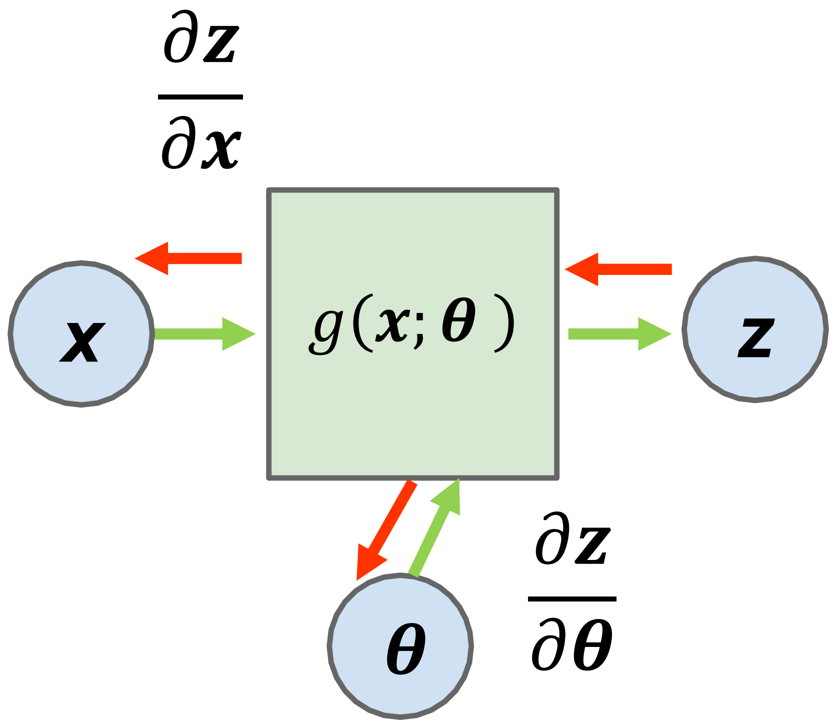
Neural network as computational graph

- $\boldsymbol{a} = sigmoid\left(\boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{x} + \boldsymbol{b}\right)$
- Decompose functions into atomic operations
- Separate data (variables) and computing (operations)
- Known as a computational graph



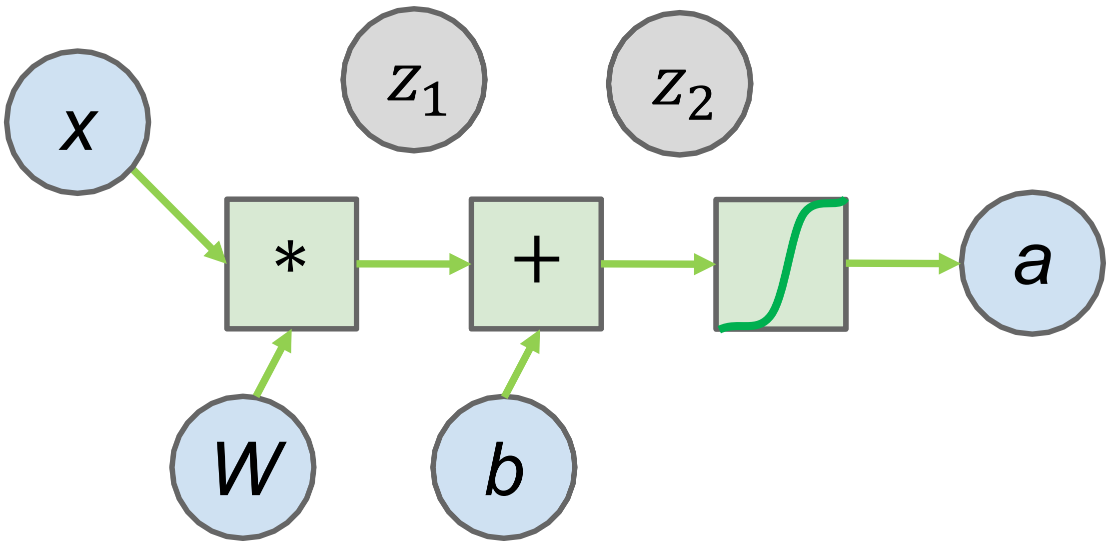
Neural network as computational graph

- Differentiable operations
- Forward / backward



Neural network: forward propagation

- Compute the output of the network

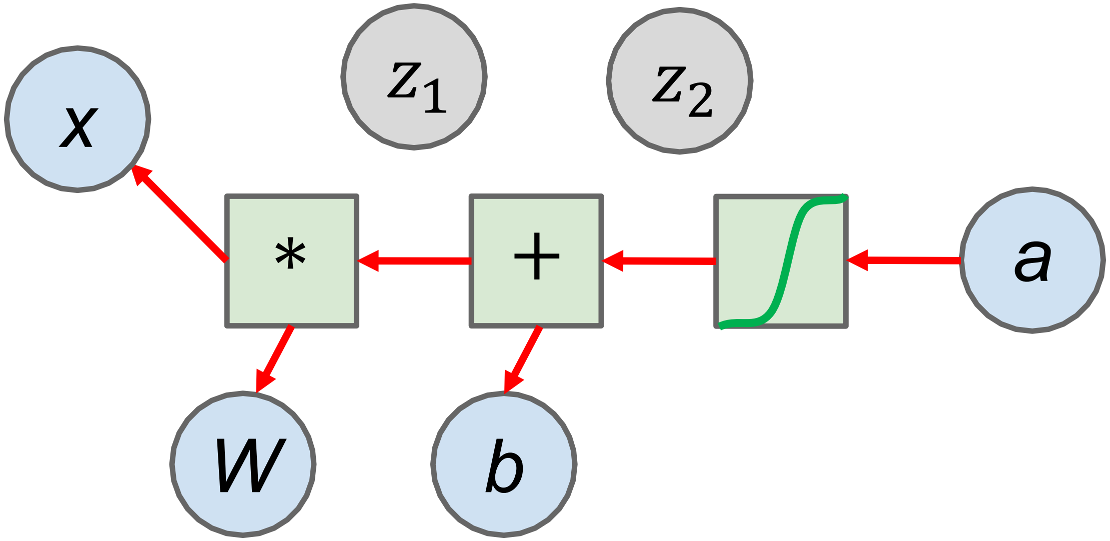


Neural network: backward propagation

- Define a loss function E
- Gradient to a variable =

gradient on the top x gradient from the current operation

$$\frac{\partial E}{\partial \boldsymbol{W}} = \frac{\partial E}{\partial \boldsymbol{z_1}} \frac{\partial \boldsymbol{z_1}}{\partial W}$$



Deep neural networks

- Deep Learning: Composing a set of (nonlinear) functions *g*

$$f(\boldsymbol{x};\boldsymbol{\theta}) = g_1(\ldots g_{n-1}(g_n(\boldsymbol{x};\boldsymbol{\theta_n}),\boldsymbol{\theta_{n-1}})\ldots,\boldsymbol{\theta_1})$$

- Each of the function *g* is represented using a layer of a neural network

- Key element: $\sigma\left(\boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{x}+\boldsymbol{b}\right)$
  - Convolution
  - Activation functions
  - Pooling

Convolution

- Given array u_t and w_t, their convolution is a function s_t

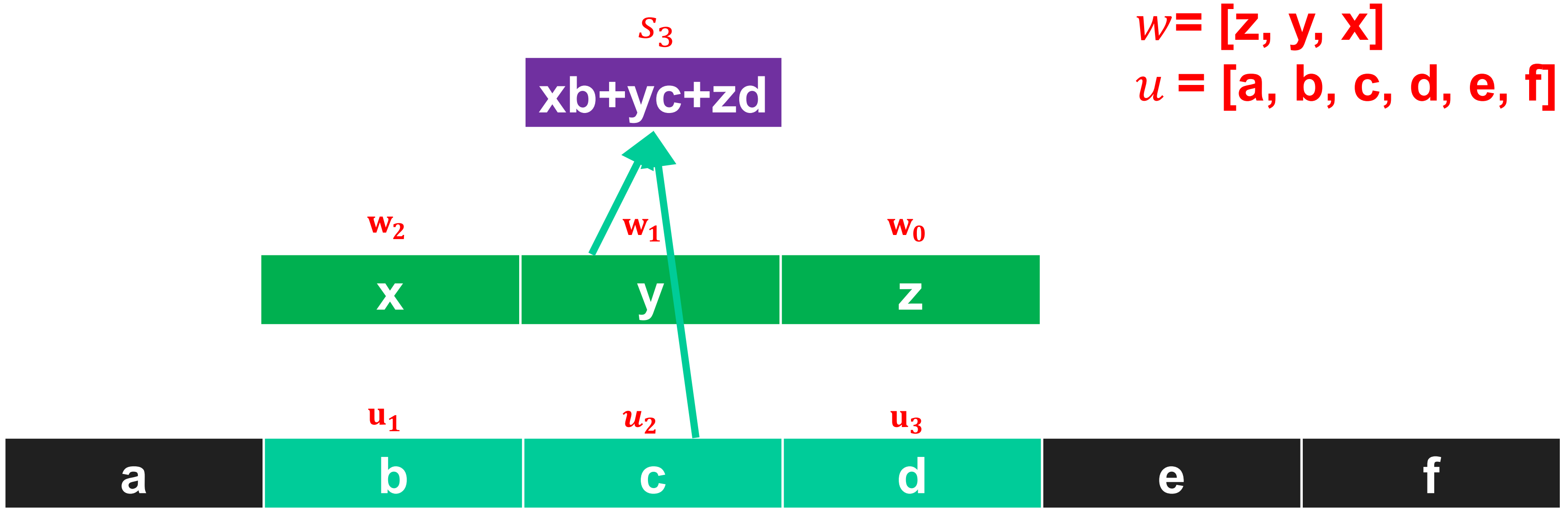
$$s_t = \sum_{a=-\infty}^{+\infty} u_a w_{t-a}$$

- Written as

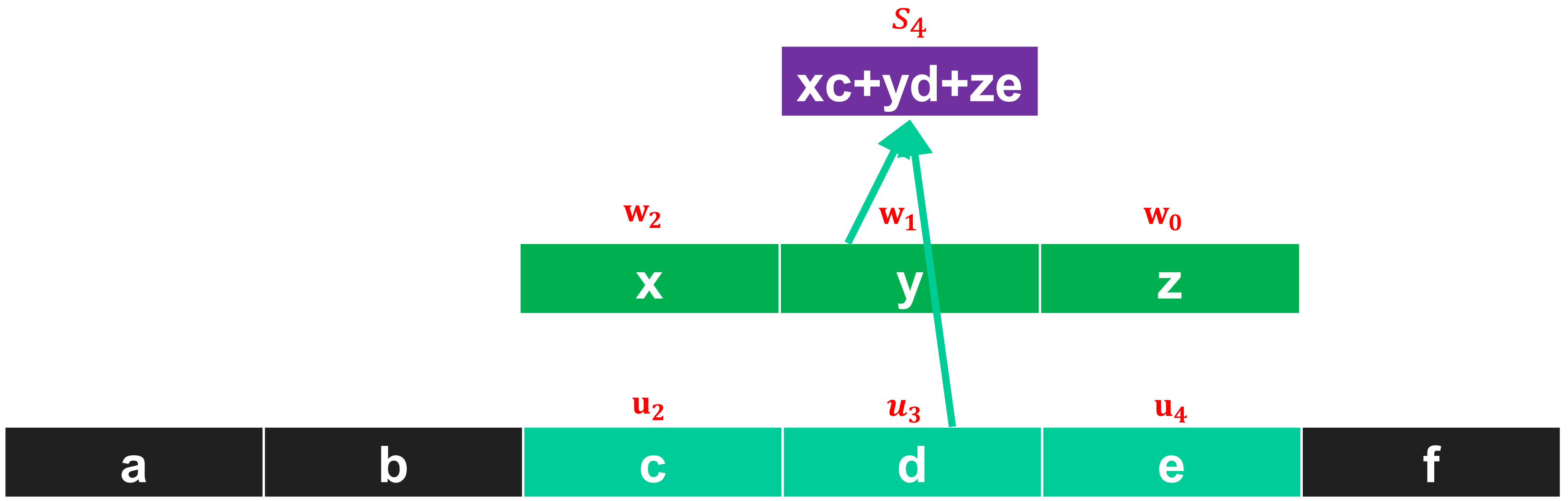
$$s = (u * w) \quad \text{or} \quad s_t = (u * w)_t$$

- When u_t or w_t is not defined, assumed to be 0
- Multiply w_t to every sliding window of u_t and sum up

Convolution



Convolution



Convolution

- A linear operation
- Can be written as matrix vector product

w= [z, y, x], u = [a, b, c, d, e, f]

y	z				
x	y	z			
	x	y	z		
		x	y	z	
			x	y	z
				x	y

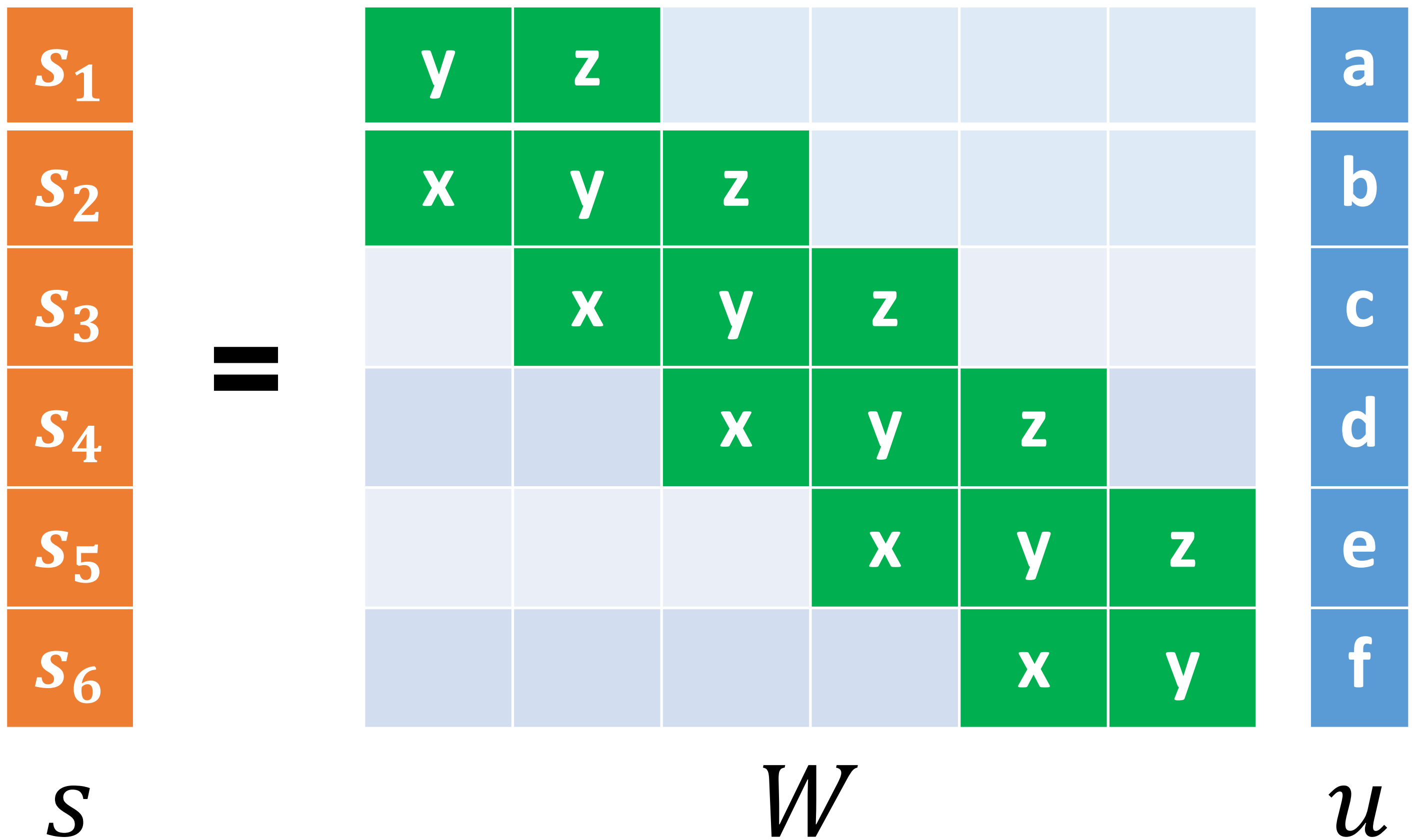
a
b
c
d
e
f

Gradient of convolution

w= [z, y, x] u = [a, b, c, d, e, f] $s = u * w$

$$s = u * w$$
$$= Wu$$

$$\frac{\partial s}{\partial u} = W$$



Gradient of convolution

w= [z, y, x] u = [a, b, c, d, e, f] $s = w * u$

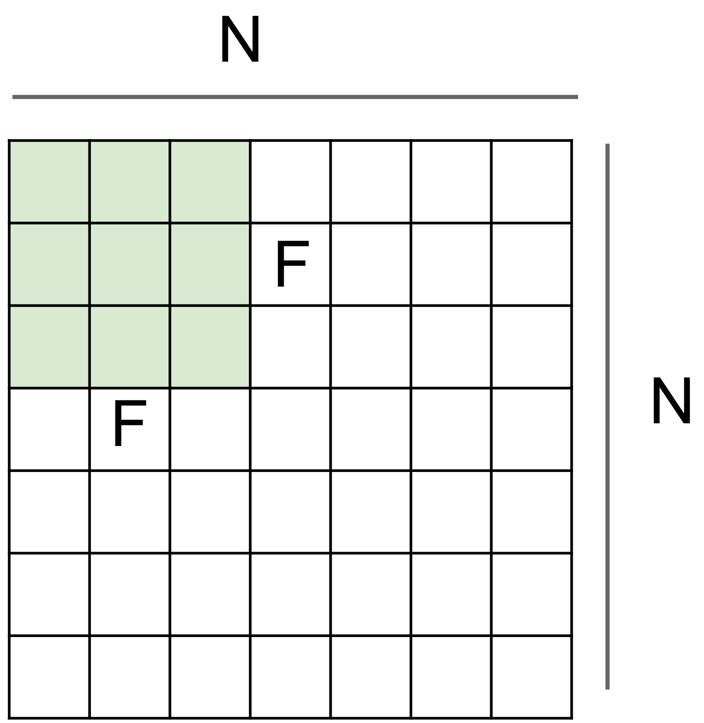
$$\frac{\partial s}{\partial w} = U$$

$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \end{bmatrix} = \begin{bmatrix} b & a & \\ c & b & a \\ d & c & b \\ e & d & c \\ f & e & d \\ & f & e \end{bmatrix} \begin{bmatrix} z \\ y \\ x \end{bmatrix}$$

s U w

Convolution with stride

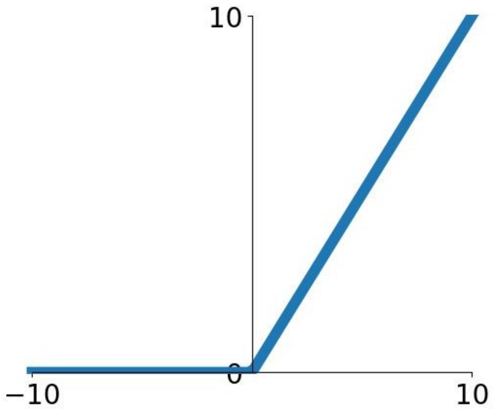
- Stride: the step size of the sliding window



Valid Output size:
(N - F) // stride + 1

e.g. 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

Activation function: ReLU



ReLU

(Rectified Linear Unit)

f(x) = max(0, x)

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid in practice
- Differentiable? Yes, if we fix $f'(0)$

Pooling

- Summarizing the input
- Max / average pooling: output the max / average of the input

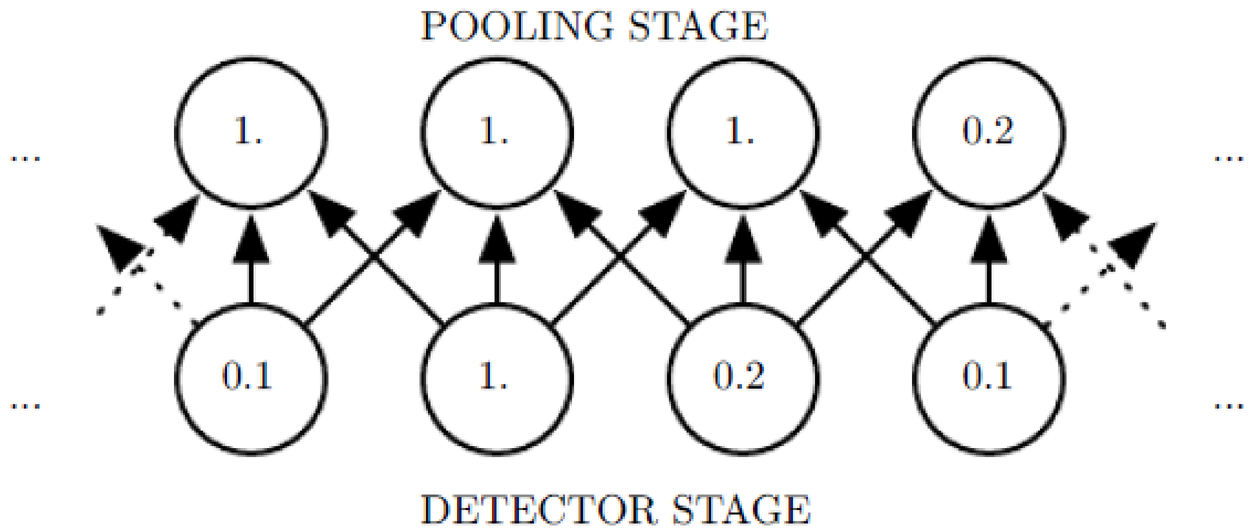
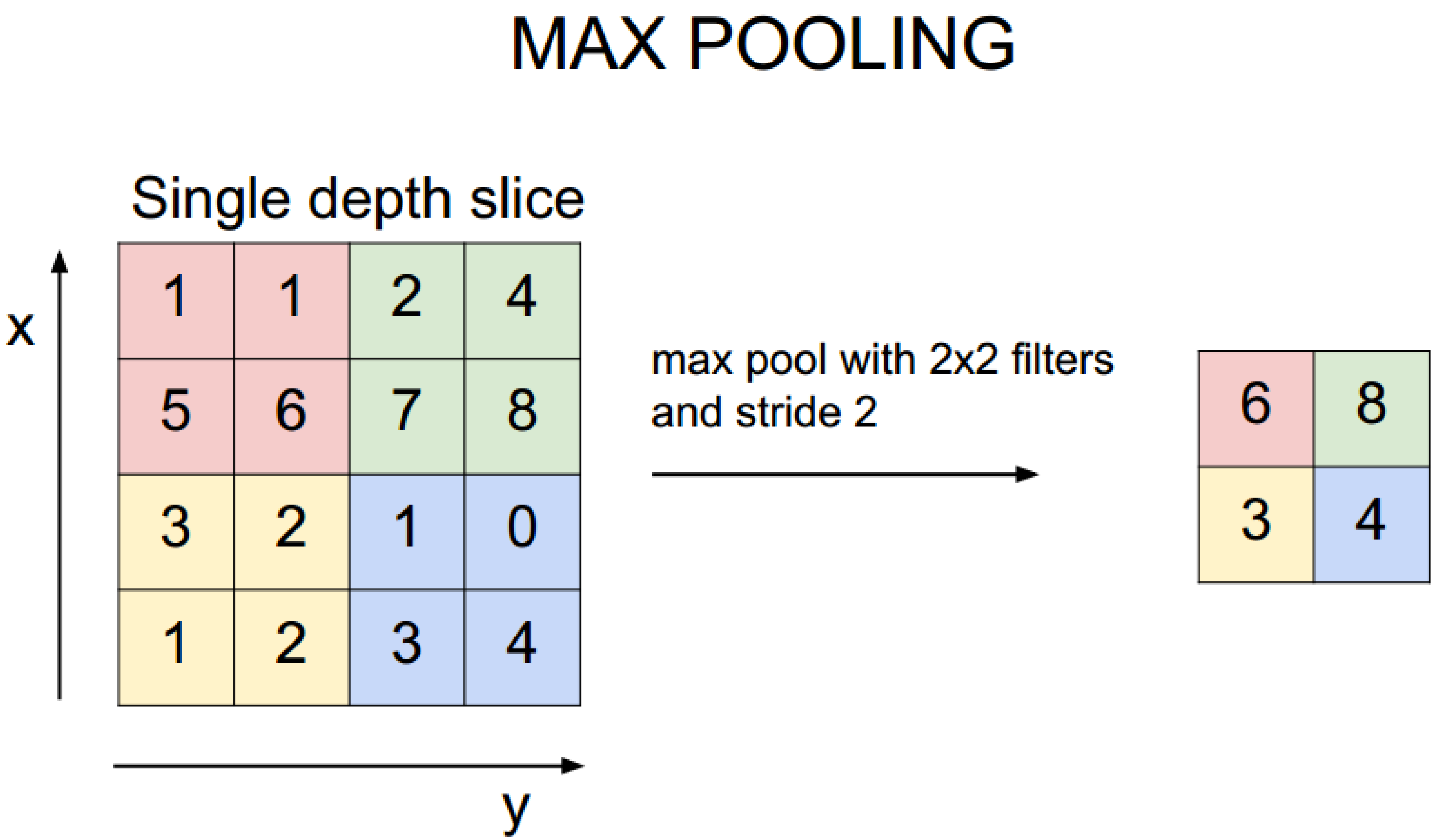


Figure from *Deep Learning,* by Goodfellow, Bengio, and Courville

Pooling operation



Deep neural networks: putting things together

- [[Conv + ReLU] x n + Pooling] x m
- A few fully connected (FC) layers at the end
- Output normalization + Loss function
- Training: mini-batch stochastic gradient descent
- Inference: use the (normalized) outputs

Deep neural networks: putting things together

- AlexNet: make it deep!
- VGGNet: smaller kernels + more layers
- GoogLeNet: multiple parallel branches
- ResNet: add skip connections