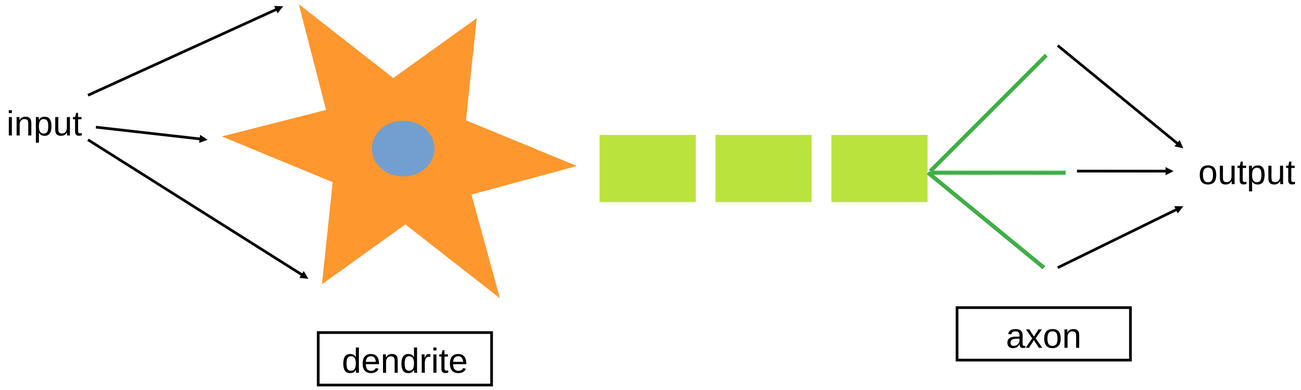
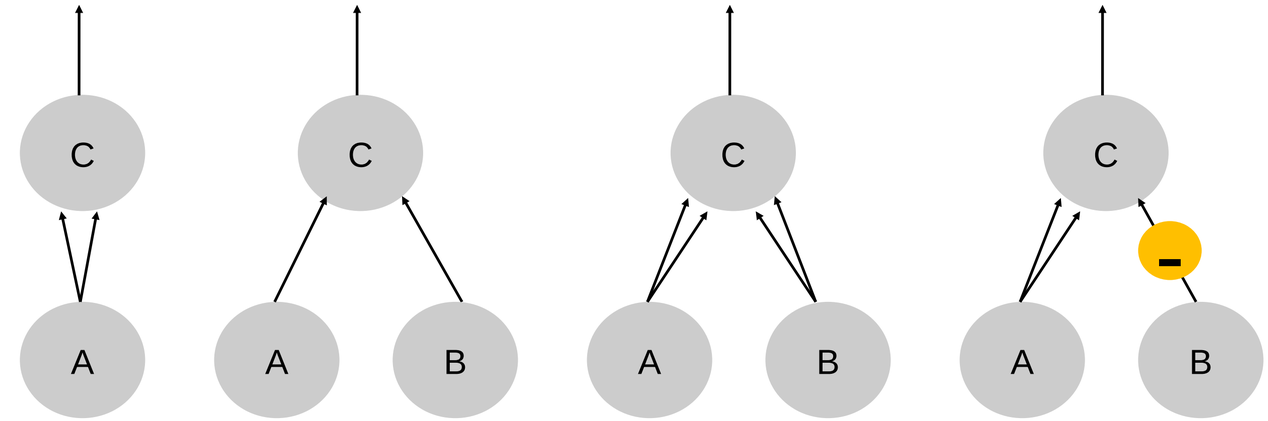
Neural Network

neuron in biology

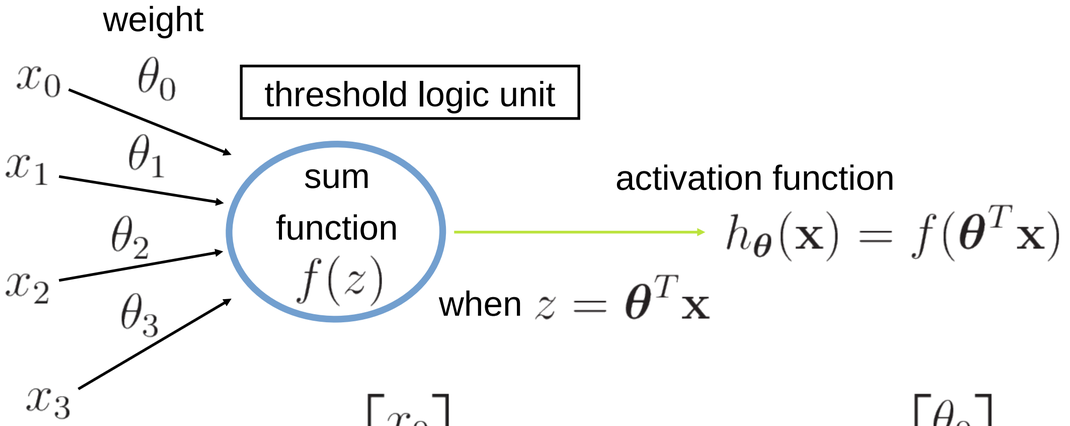


artificial neuron



$C = A$ $C = A \wedge B$ $C = A \vee B$ $C = A \vee \neg B$

neuron model : perceptron



input $\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$ weight parameter $\boldsymbol{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \end{bmatrix}$

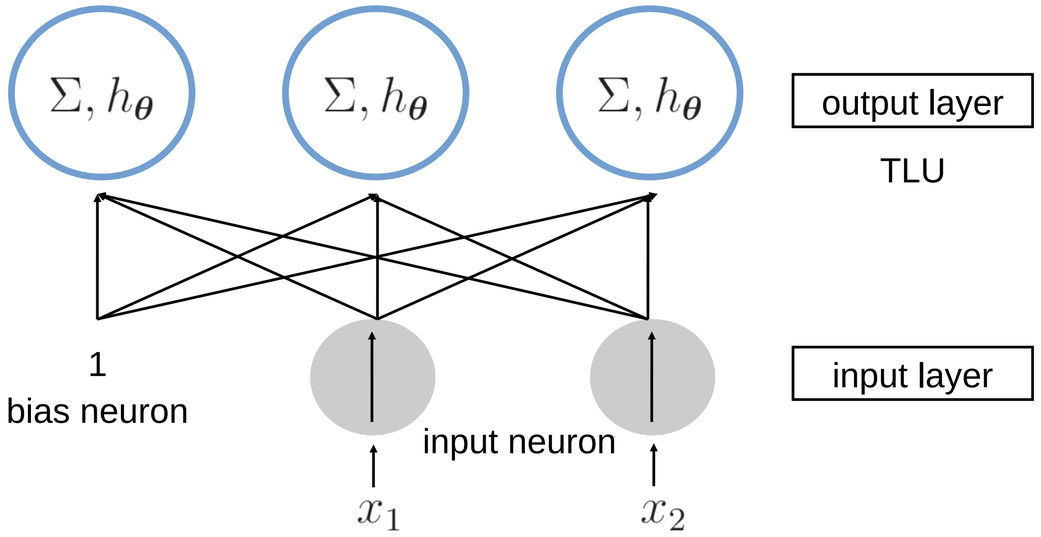
generally use heaviside function or sign function for f(z)

$$heaviside(z) = \begin{cases} 0 & (z < 0) \\ 1 & (z \geq 0) \end{cases}$$

$$sgn(z) = \begin{cases} -1 & (z < 0) \\ 0 & (z = 0) \\ 1 & (z > 0) \end{cases}$$

Neural Network

perceptron structure



how to train perceptron – Hebb's rule

"neurons that activate each other are interconnected"

$$\boldsymbol{\theta}_{i,j}^{new} = \boldsymbol{\theta}_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

$\boldsymbol{\theta}_{i,j}$: connection weight between *i*th input neuron and *j*th output neuron

η : learning rate

y_j : target output of *j*th output neuron for current training instance

$\hat{y}_j$: output of *j*th output neuron for current training instance

x_i : *i*th input value of current training instance

perceptron convergence theorem

"for finite set of linearly separable labeled examples, perceptron learning algorithm converges after finite number of iterations"

i.e.

"after finite number of iterations, algorithm makes a vector that classifies perfectly all examples"

linearly separable datasets X^+, X^-

label $y_i = \pm 1(\mathbf{x}_i \in X^{\pm})$

there is N data in whole dataset $X = X^+ \cup X^-$

input return to first after N data are trained

$$\mathbf{x}(1), \mathbf{x}(2), ..., \mathbf{x}(N), \mathbf{x}(1), ..., \mathbf{x}(N), \mathbf{x}(1), \mathbf{x}(2), ...$$

$\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n, ...$ input data (prediction failed)

initial weight $\mathbf{w}_1 = \mathbf{0}$

weight updating $\mathbf{w}_{n+1} = \mathbf{w}_n + y_n \mathbf{x}_n$

then, $\exists\ n_o$ s.t. $\mathbf{w}_{n_0} = \mathbf{w}_{n_0+1} = \mathbf{w}_{n_0+2} = ...$

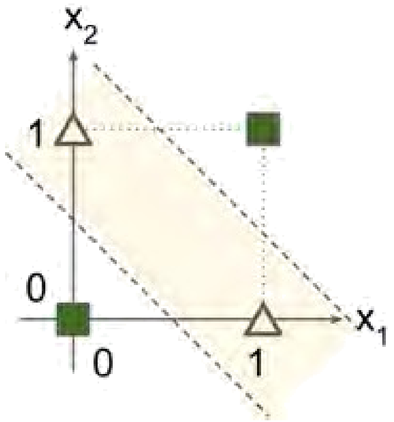
$$\begin{cases} \mathbf{w}_{n_0}^T \mathbf{x} > 0 \text{ if } \mathbf{x} \in X^+ \\ \mathbf{w}_{n_0}^T \mathbf{x} \leq 0 \text{ if } \mathbf{x} \in X^- \end{cases}$$

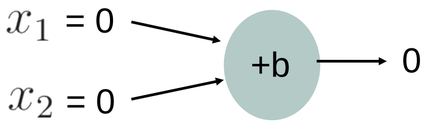
Neural Network

proof

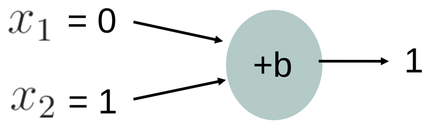
Neural Network

multi-layer perceptron

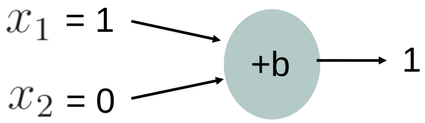




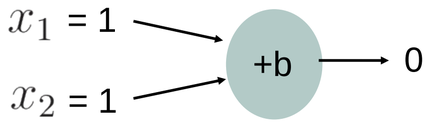
$b < 0.5$



$x_2 + b > 0.5$



$x_1 + b > 0.5$



$x_1 + x_2 + b < 0.5$

XOR problem

x_1	x_2	y
0	0	0
1	0	1
0	1	1
1	1	0

$$b < 0.5$$
$$x_2 + b > 0.5$$
$$x_1 + b > 0.5$$
$$x_1 + x_2 + b < 0.5$$

add x_1 to both sides of second inequality

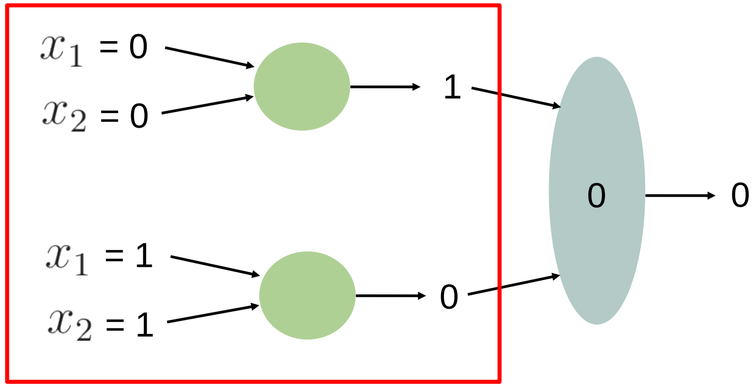
$$x_1 + x_2 + b > 0.5 + x_1$$

according to fourth inequality, $x_1 < 0$

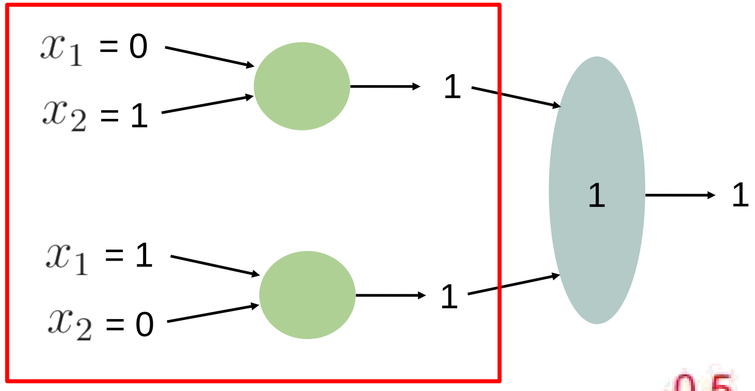
: contradict to first and third inequality

: can't classify linearly

solving – multi layer perceptron

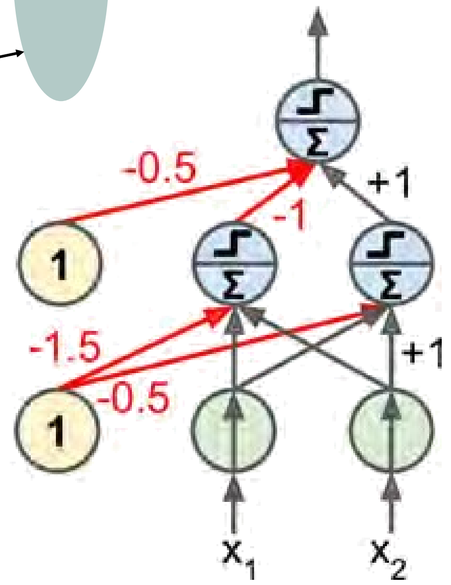


NAND gate



OR gate

using bias



Neural Network

gradient descent

method to find minimum of function

large tangent → x is far from minimum

the larger x, the larger function value (positive tangent)

: move x to - direction

the larger x, the smaller function value (negative tangent)

: move x to + direction

$$x_{i+1} = x_i - \text{distance} \times \text{sign of tangent}$$

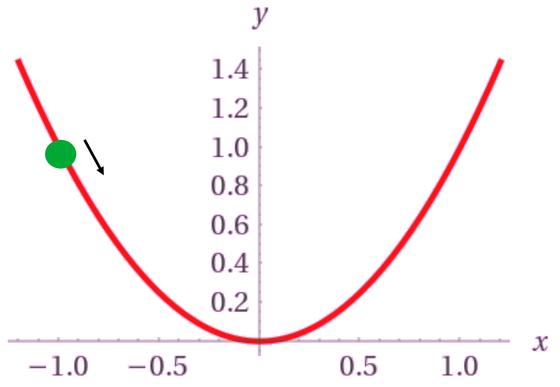
distance value → use factor that is in proportion to gradient

'step size' α

$$x_{i+1} = x_i - \alpha \frac{df}{dx}(x_i)$$

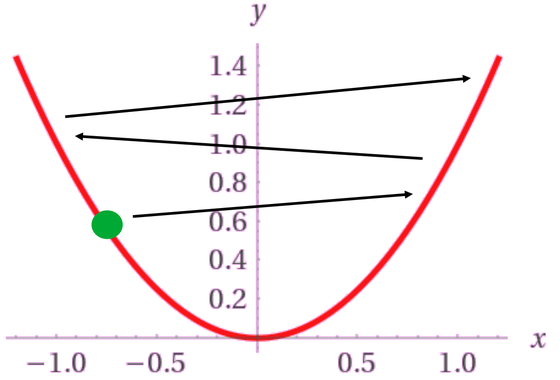
multivariate function : $x_{i+1} = x_i - \alpha \nabla f(x_i)$

problems of gradient descent



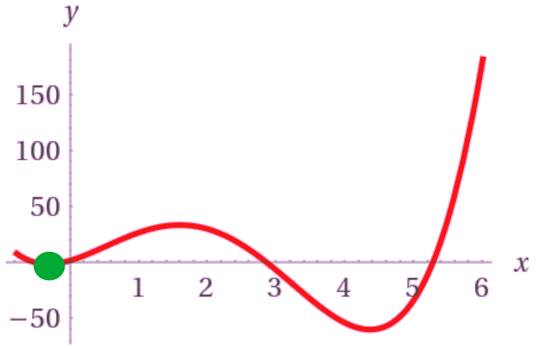
too small step size

can't converge well



too big step size

diverges



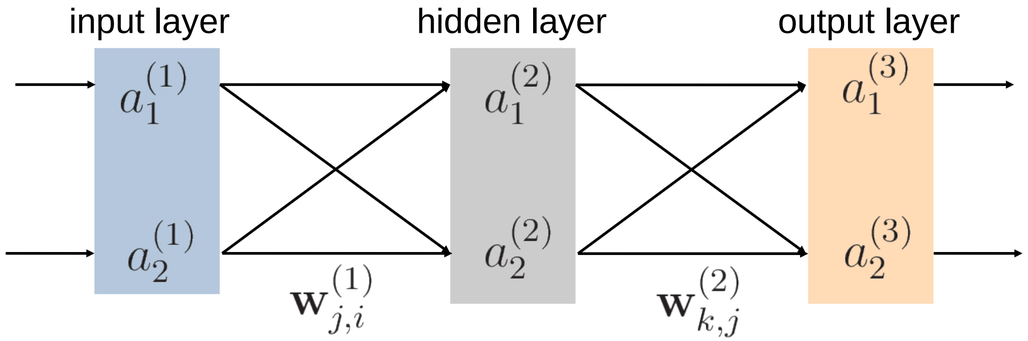
wrong initialization

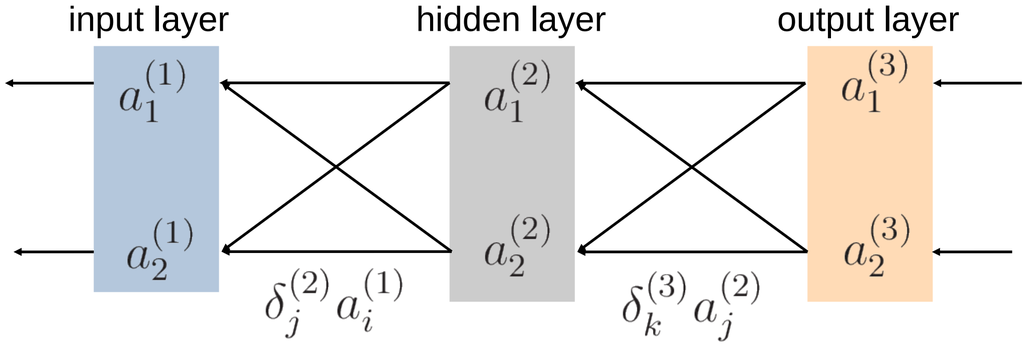
→ caught up in local minimum

Neural Network

back propagation

feedforward





⟶ direction : calculate output and error, make prediction

⟵ direction : calculate contribution for error, adjust weight

need back propagation algorithm

update weight value via loss ftn.

determine weight $\mathbf{w}_{j,i}^{(l)}$ ⟶ minimize loss function

$$J_1 = \frac{1}{2}(a_1^{(3)} - y_1)^2$$

$$J_2 = \frac{1}{2}(a_2^{(3)} - y_2)^2$$

weights update

$$\mathbf{w}_{j,i}^{(l)new} = \mathbf{w}_{j,i}^{(l)} - \delta_j^{(l+1)} a_i^{(l)}$$

$$\delta_j^{(3)} = (a_j^{(3)} - y_i) \times a_j^{(3)}(1 - a_j^{(3)})$$

$$\delta_j^{(2)} = (\delta_1^{(3)} \mathbf{w}_{1,j}^{(2)} + \delta_2^{(3)} \mathbf{w}_{2,j}^{(2)}) \times a_j^{(2)}(1 - a_j^{(2)})$$

procedure to update weight

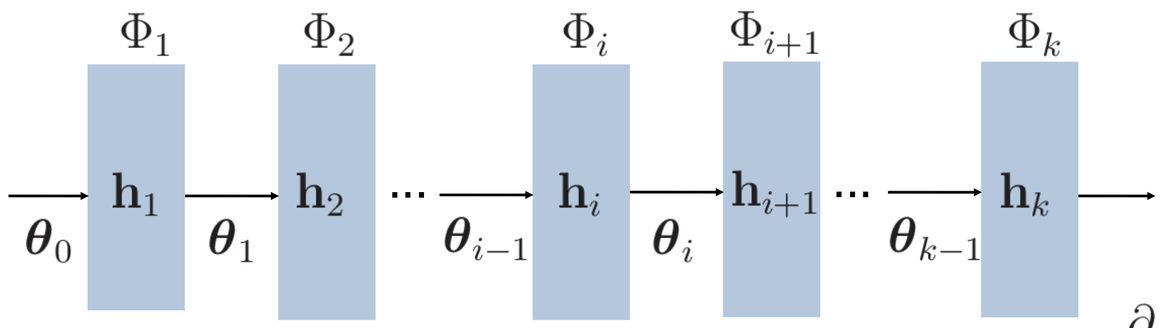
feedforward : caculate output using given weight

subtract differentiated (to weight) error from weight

updated weight

$$\mathbf{w}_j^{new} = \mathbf{w}_j - \eta \frac{\partial J_{total}}{\partial \mathbf{w}_j}$$

Neural Network



hidden layers $\mathbf{h}_j \ (j = 1, 2, ..., k)$

parameters $\boldsymbol{\theta}_j \ (j = 0, 1, ..., k-1)$

minimize loss function $\min_{\boldsymbol{\theta}} L$

using gradient descent $\boldsymbol{\theta}^{new} = \boldsymbol{\theta} - \eta \frac{\partial L}{\partial \theta}$

$$\mathbf{h}_0 = \mathbf{x}$$

$$\mathbf{h}_{i+1} = \Phi_{i+1}(\mathbf{h}_i, \boldsymbol{\theta}_i) \quad \text{for} \quad i = 0, 1, ..., k-1$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_0} = \frac{\partial L}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \mathbf{h}_{k-1}} \cdots \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2} \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1} \frac{\partial \mathbf{h}_1}{\partial \boldsymbol{\theta}_0} \Rightarrow \boldsymbol{\theta}_0^{new} = \boldsymbol{\theta}_0 - \eta \frac{\partial L}{\partial \boldsymbol{\theta}_0}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_1} = \frac{\partial L}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \mathbf{h}_{k-1}} \cdots \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2} \frac{\partial \mathbf{h}_2}{\partial \boldsymbol{\theta}_1} \Rightarrow \boldsymbol{\theta}_1^{new} = \boldsymbol{\theta}_1 - \eta \frac{\partial L}{\partial \boldsymbol{\theta}_1}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_j} = \frac{\partial L}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \mathbf{h}_{k-1}} \cdots \frac{\partial \mathbf{h}_{j+2}}{\partial \mathbf{h}_{j+1}} \frac{\partial \mathbf{h}_{j+1}}{\partial \boldsymbol{\theta}_j} \Rightarrow \boldsymbol{\theta}_j^{new} = \boldsymbol{\theta}_j - \eta \frac{\partial L}{\partial \boldsymbol{\theta}_j}$$

$$\vdots$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_{k-1}} = \frac{\partial L}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \boldsymbol{\theta}_{k-1}} \Rightarrow \boldsymbol{\theta}_{k-1}^{new} = \boldsymbol{\theta}_{k-1} - \eta \frac{\partial L}{\partial \boldsymbol{\theta}_{k-1}}$$

$$\vdots$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_j} = \frac{\partial L}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \mathbf{h}_{k-1}} \cdots \frac{\partial \mathbf{h}_{j+3}}{\partial \mathbf{h}_{j+2}} \frac{\partial \mathbf{h}_{j+2}}{\partial \mathbf{h}_{j+1}} \frac{\partial \mathbf{h}_{j+1}}{\partial \boldsymbol{\theta}_j} \Rightarrow \boldsymbol{\theta}_j^{new} = \boldsymbol{\theta}_j - \eta \frac{\partial L}{\partial \boldsymbol{\theta}_j}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_{j+1}} = \frac{\partial L}{\partial \mathbf{h}_k} \frac{\partial \mathbf{h}_k}{\partial \mathbf{h}_{k-1}} \cdots \frac{\partial \mathbf{h}_{j+3}}{\partial \mathbf{h}_{j+2}} \frac{\partial \mathbf{h}_{j+2}}{\partial \boldsymbol{\theta}_{j+1}} \Rightarrow \boldsymbol{\theta}_{j+1}^{new} = \boldsymbol{\theta}_{j+1} - \eta \frac{\partial L}{\partial \boldsymbol{\theta}_{j+1}}$$

$$\frac{\partial L}{\partial \boldsymbol{\theta}_j} = \frac{\partial L}{\partial \mathbf{h}_{j+1}} \frac{\partial \mathbf{h}_{j+1}}{\partial \boldsymbol{\theta}_j}$$

$$\frac{\partial L}{\partial \mathbf{h}_{j+1}} = \frac{\partial L}{\partial \mathbf{h}_{j+2}} \frac{\partial \mathbf{h}_{j+2}}{\partial \mathbf{h}_{j+1}}$$

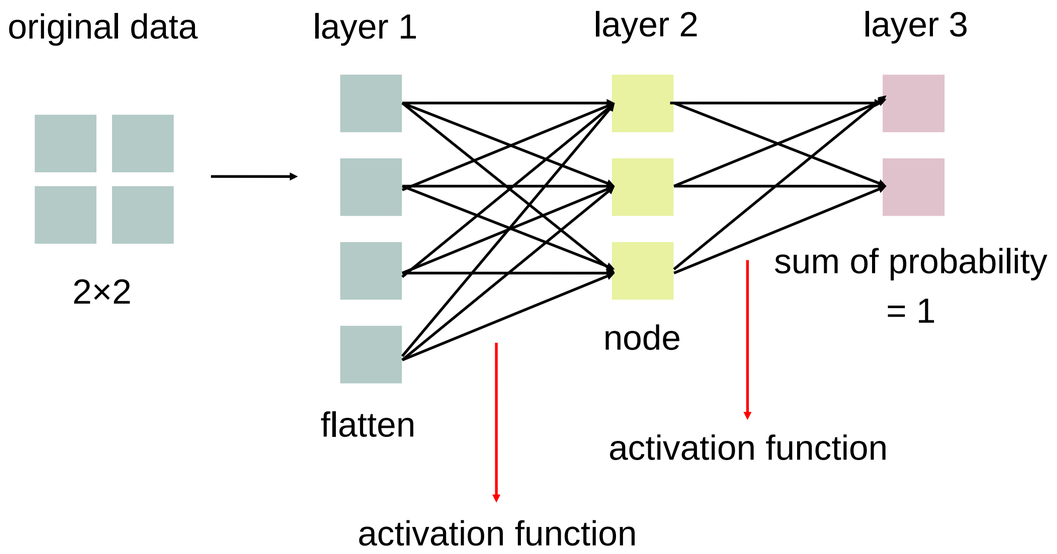
$$\longrightarrow \quad \boldsymbol{\theta}_j^{new} = \boldsymbol{\theta}_j - \eta \frac{\partial L}{\partial \mathbf{h}_{j+1}} \frac{\partial \mathbf{h}_{j+1}}{\partial \boldsymbol{\theta}_j}$$

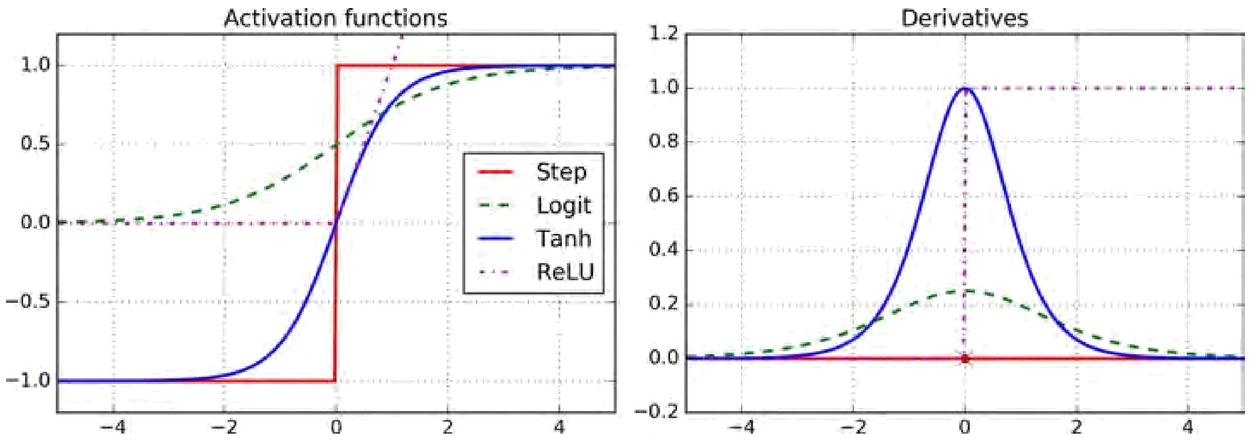
where $\frac{\partial L}{\partial \mathbf{h}_{j+1}} = \frac{\partial L}{\partial \mathbf{h}_{j+2}} \frac{\partial \mathbf{h}_{j+2}}{\partial \mathbf{h}_{j+1}}$

$$j = k-1, k-2, ..., 0$$

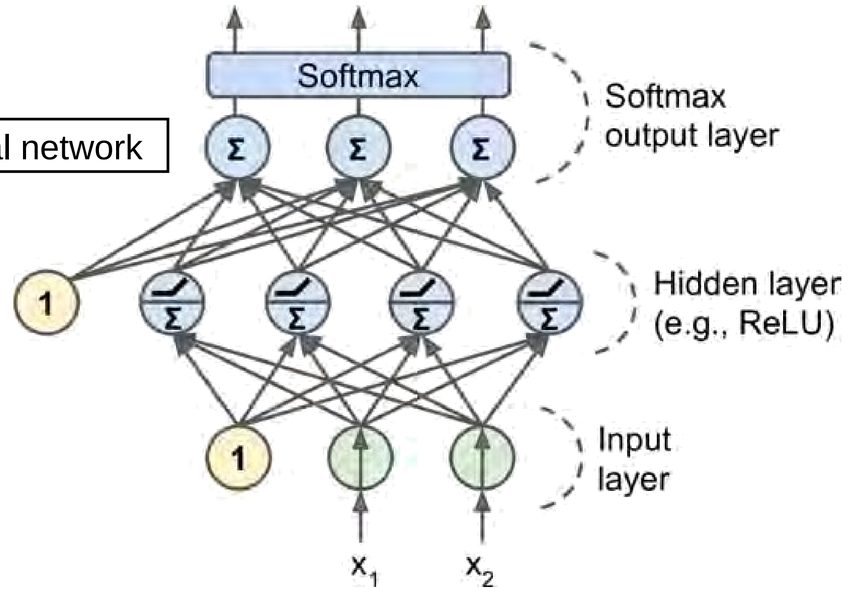
Neural Network

fully connected





feed forward neural network



activation function

- logistic
- hyperbolic tangent
- softmax
- ReLU

$$softmax(x_i) = \frac{exp(x_i)}{\sum_{j=1}^{n} exp(x_j)}$$

$$\mathrm{ReLU}(x) = max(0, x)$$

and
etc.

Why don't use step function?

step function has no gradient : can't apply gradient descent

Necessity of CNN

problems of fully-connected

MNIST dataset from chap 10

28×28 pixels $x \in \mathbb{R}^{784}$

original data

3-dimensional

(width,height,channel) = (28,28,1)

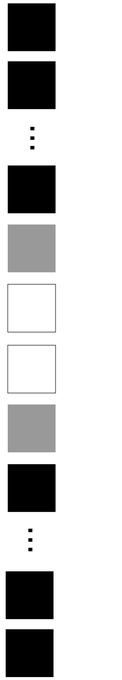
flatten

input layer

1-dimensional

28×28×1 = 784

28×28 pixels

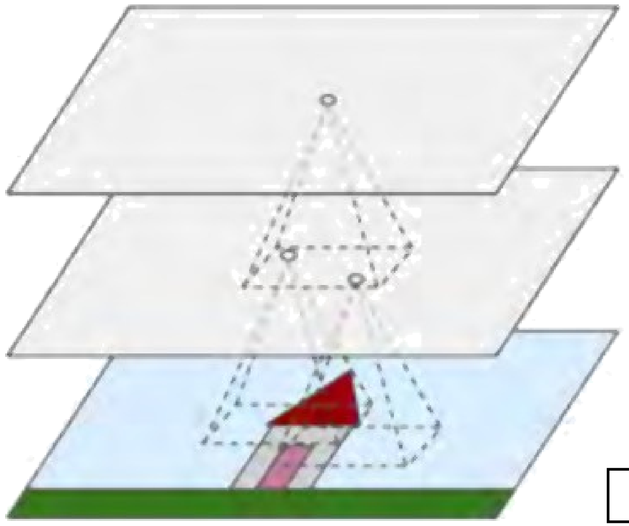


784 pixels

ignore shape of data

vanish spatial structure

convolutional layer



input layer 2

input layer 1

3-dimensional data (image)

maintain shape of input data

preserve spatial structure

CNN

layers of CNN

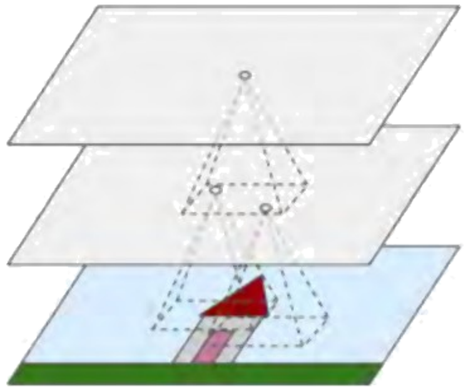
fully connected

affine | ReLU | affine | ReLU | affine | softmax

CNN

conv | ReLU | pooling | conv | ReLU | pooling

convolutional layer

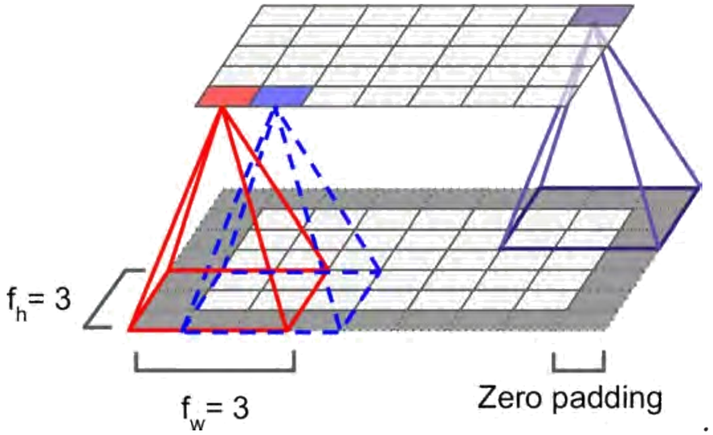


neuron of convolutional layers

: only connect to receptive field pixels

forward convolutional layer ⟶ low level features

backward convolutional layer ⟶ high level features



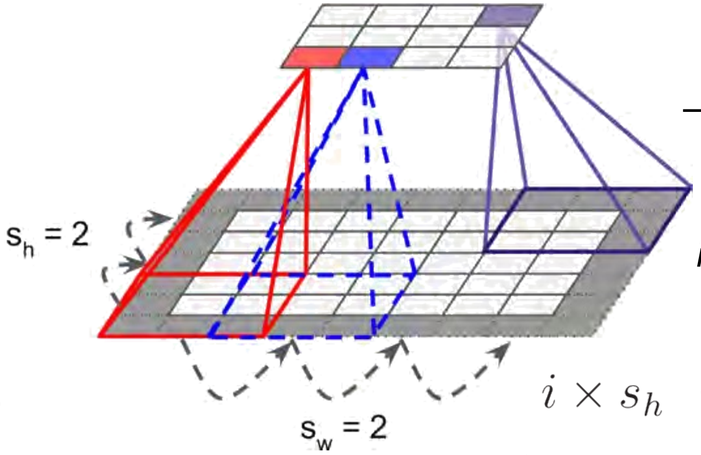
*i*th row, *j*th column neuron ⟶ i to $i + f_h - 1$, j to $j + f_w - 1$

spatial size of data become small when pass each conv layer

⟶ edge data are lost

zero padding : 0 cells around input data

adjust spatial size of output data



large input layer ⟶ small input layer

stride

*i*th row, *j*th column neuron

$i \times s_h$ to $i \times s_h + f_h - 1$

$j \times s_w$ to $j \times s_w + f_w - 1$

CNN

cross-correlation

in CNN, convolutional layer uses cross-correlation

reason why don't use convolution

convolutional layer

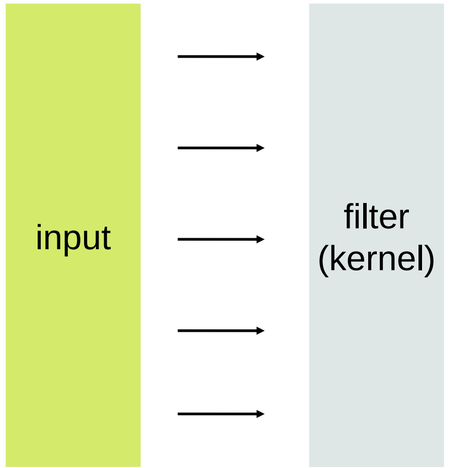
definition of convolution (in mathematics)

"integral of two functions' product after one is reversed and shifted"

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t - \tau) d\tau$$

2-dimensional (image : height h, width w)

$$(f * g)(i, j) = \sum_{x=0}^{h-1} \sum_{y=0}^{w-1} f(x, y) g(i - x, j - y)$$



convolution : filter have to be reversed

cross-correlation : don't have to reverse

cross-correlation

similarly,

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t + \tau) d\tau$$

$$(f * g)(i, j) = \sum_{x=0}^{h-1} \sum_{y=0}^{w-1} f(x, y) g(i + x, j + y)$$

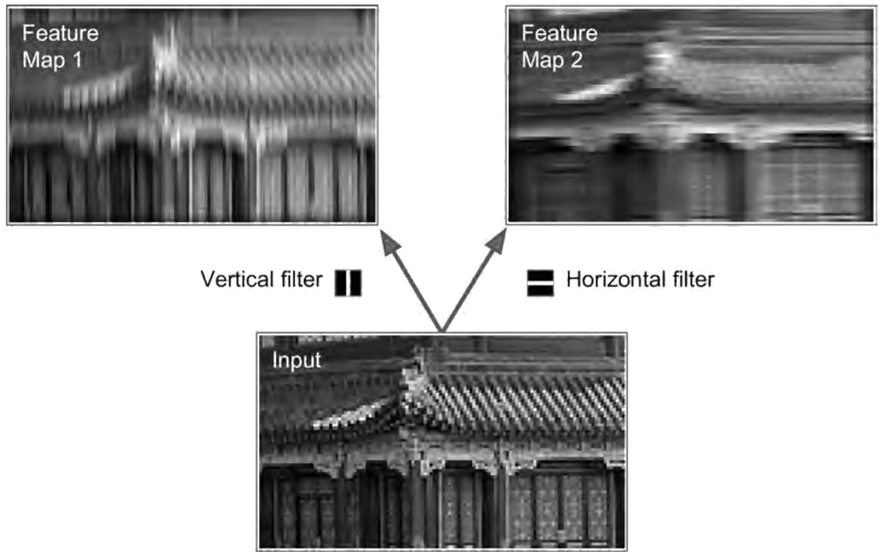
reverse or not reverse, we can get same result

⟶ there's no reason to insist convolution

customarily, we call this convolutional layer

CNN

filter



1	2	3	0
0	1	2	3
3	0	1	2
2	3	0	1

input data

2	0	1
0	1	2
1	0	2

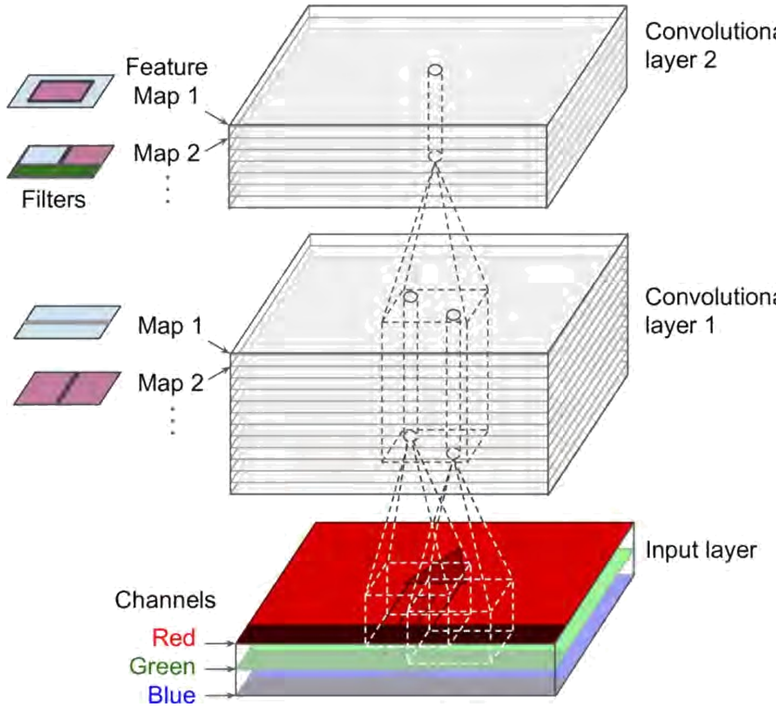
filter (kernel)

elements : weight parameter

15	16
6	15

feature map

multiple feature map



$z_{i,j,k}$: neuron output in feature map k

s_h, s_w : strides

f_h, f_w : receptive field height and width

$f_{n'}$: previous layer feature maps

$x_{i',j',k'}$: previous layer neuron output

b_k : feature map bias term

$\boldsymbol{\theta}_{u,v,k',k}$: connection weight

neuron output on conv layer $z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} \boldsymbol{\theta}_{u,v,k',k}^{T} x_{i',j',k'}$

$$\begin{cases} i' = i \times s_h + u \\ j' = j \times s_w + v \end{cases}$$

CNN

pooling layer

why we use pooling layer

make subsample of input image

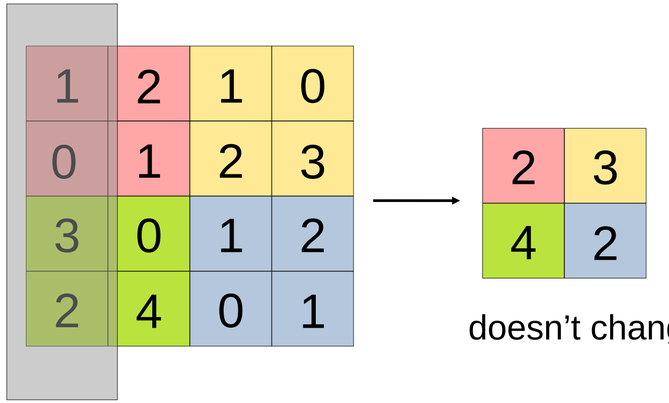
minimize → calculation, memory usage, number of parameters

reduce size of image : location invariance

There is no weight in pooling layer

pooling as an invariant

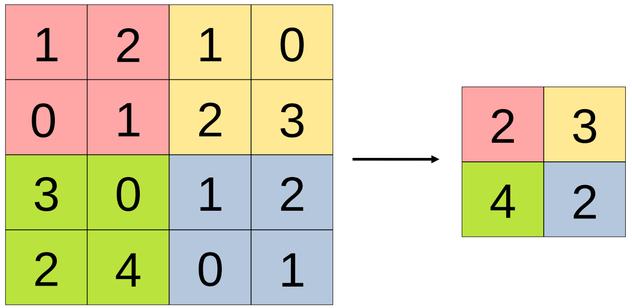
change location of data



doesn't change easily

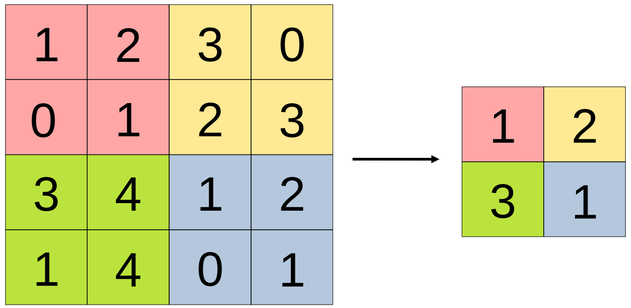
→ location invariance

max pooling



stride : 2

average pooling



stride : 2

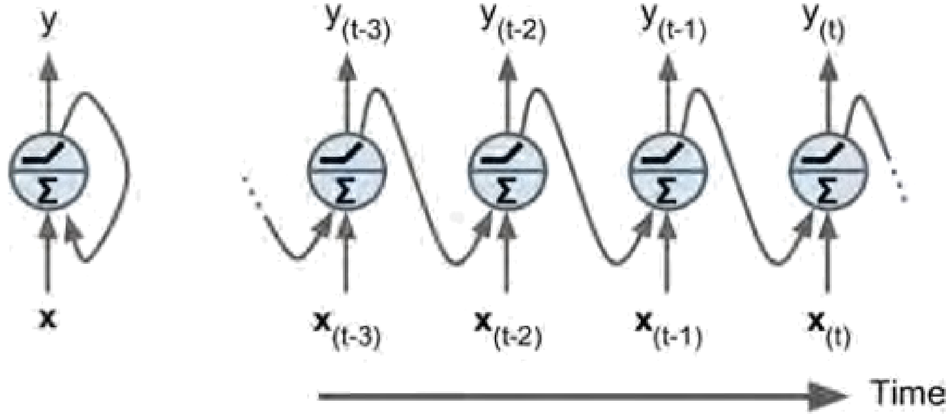
image classification – generally use max pooling

features of pooling layer

- there's no parameter to train
- number of input data channel is same as output data channel
- change of input data affect little to pooling layer

RNN

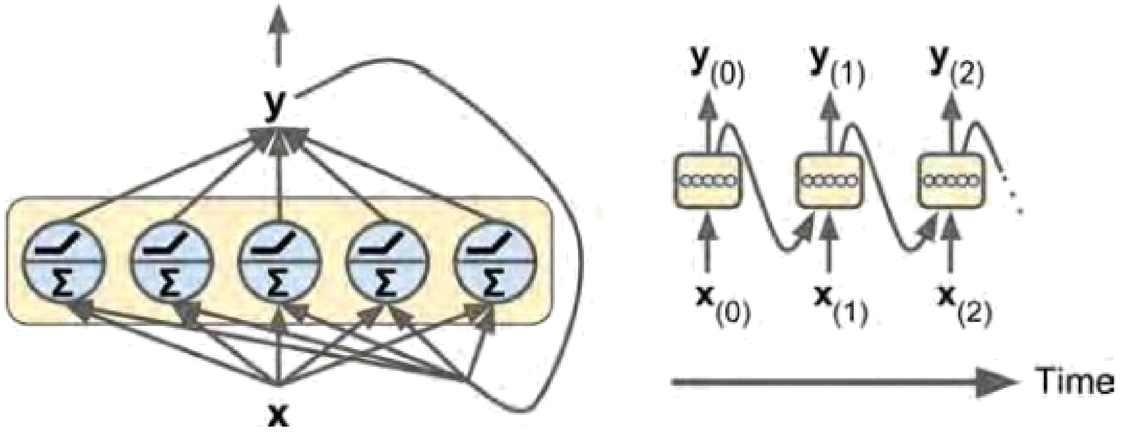
recurrent neurons



network unrolling through time

each time step t : recurrent neuron receives **x** and previous output y as input

layer of recurrent neurons



multiple neurons : input and output are both vector

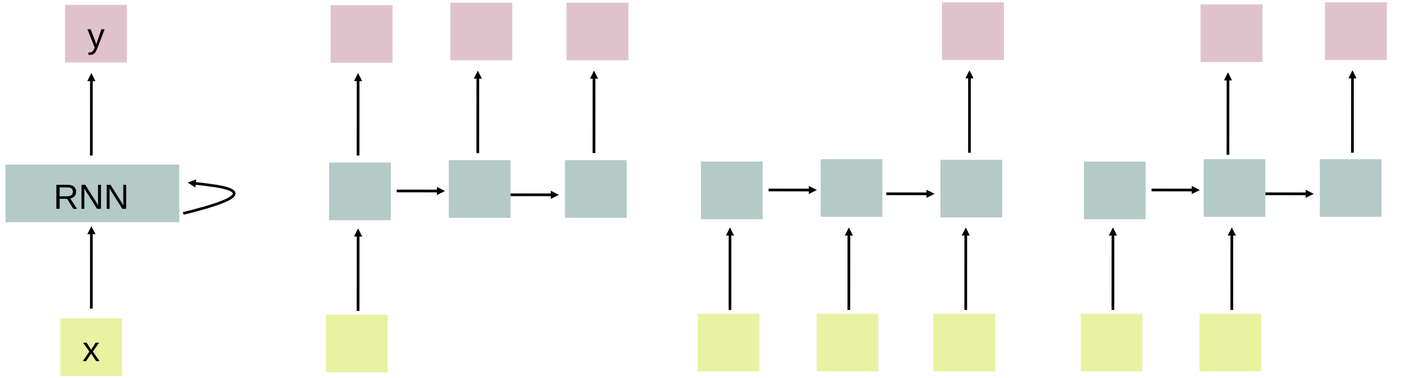
$\mathbf{w}_x$: weight vector for input **x**

$\mathbf{w}_y$: weight vector for output **y**

$$\mathbf{y}_{(t)} = \phi\left(\mathbf{W}_x^T\mathbf{x}_{(t)} + \mathbf{W}_y^T\mathbf{y}_{(t-1)} + \mathbf{b}\right)$$

φ is activation function

generally use nonlinear function (such as tanh, ...)



basic

one to many

e.g. picture - words

many to one

e.g. words - number

many to many

e.g. words - words

why?

if we use linear function $y = f(x) = ax$ for activation function,

after 3 layers : $y = f(f(f(x))) = a^3x = cx$

same result with no hidden layer

RNN

output of recurrent neurons layer

$$\mathbf{Y}_{(t)} = \phi\left(\mathbf{X}_{(t)}\mathbf{W}_x + \mathbf{Y}_{(t-1)}\mathbf{W}_y + \mathbf{b}\right)$$
$$= \phi\left(\left[\mathbf{X}_{(t)}\,\mathbf{Y}_{(t-1)}\right]\mathbf{W} + \mathbf{b}\right)$$

$\mathbf{Y}_{(t)}$: m×n matrix containing output

$\mathbf{X}_{(t)}$: m×n matrix containing input

$\mathbf{W}_x$: n×n matrix containing connection weights for input of current time step

$\mathbf{W}_y$: n×n matrix containing connection weights for output of previous time step

$\mathbf{b}$: n size vector containing each neuron’s bias

Deep Neural Network

main problem

high resolution image classification

: we need a lot of layer

with thousands of connection

vanishing gradient problem

(exploding gradient)

training is slowed down

overfitting to training set