神经网络和深度学习基础理论

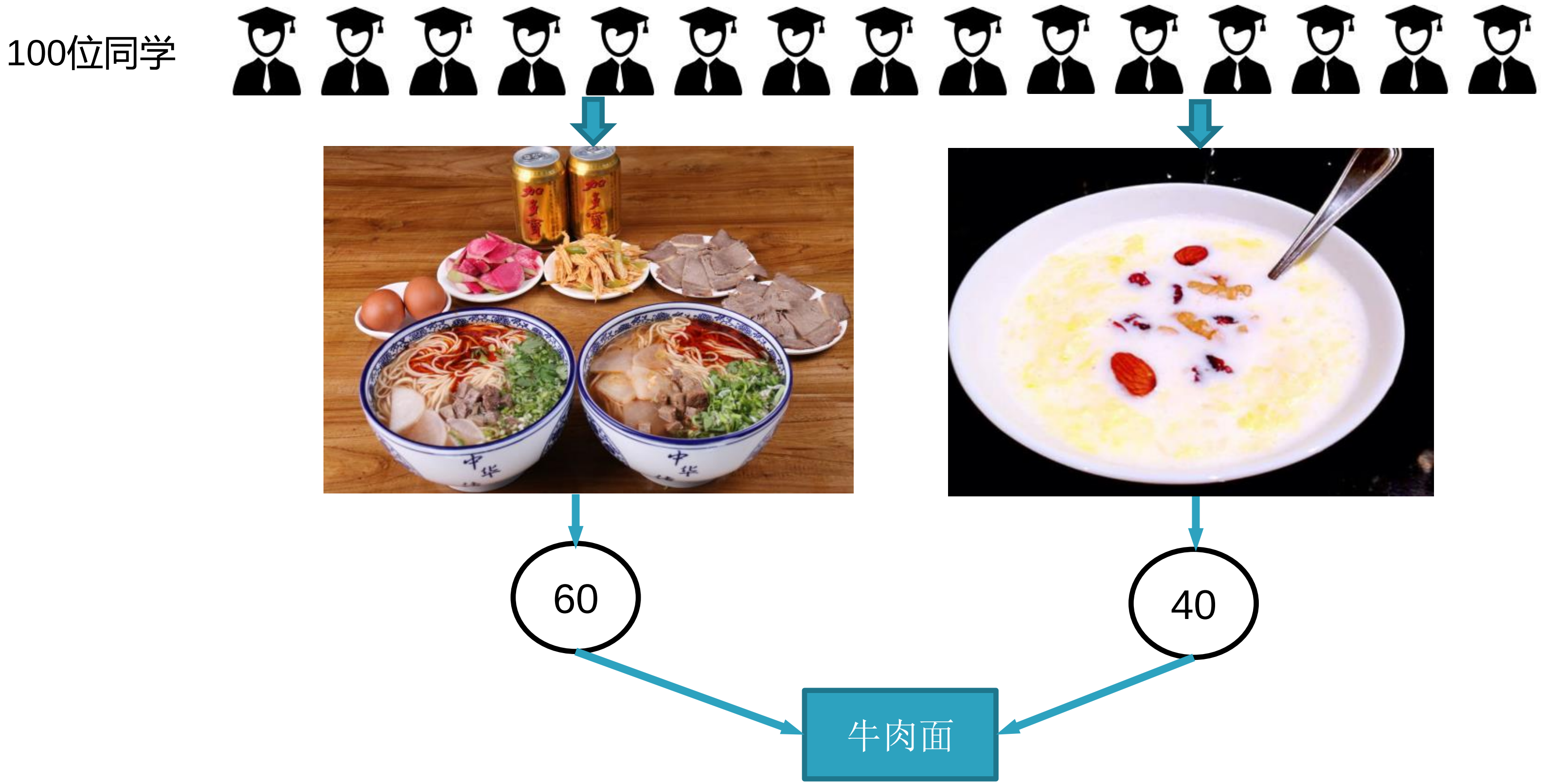
兰州大学 雍宾宾

yongbb@lzu.edu.cn

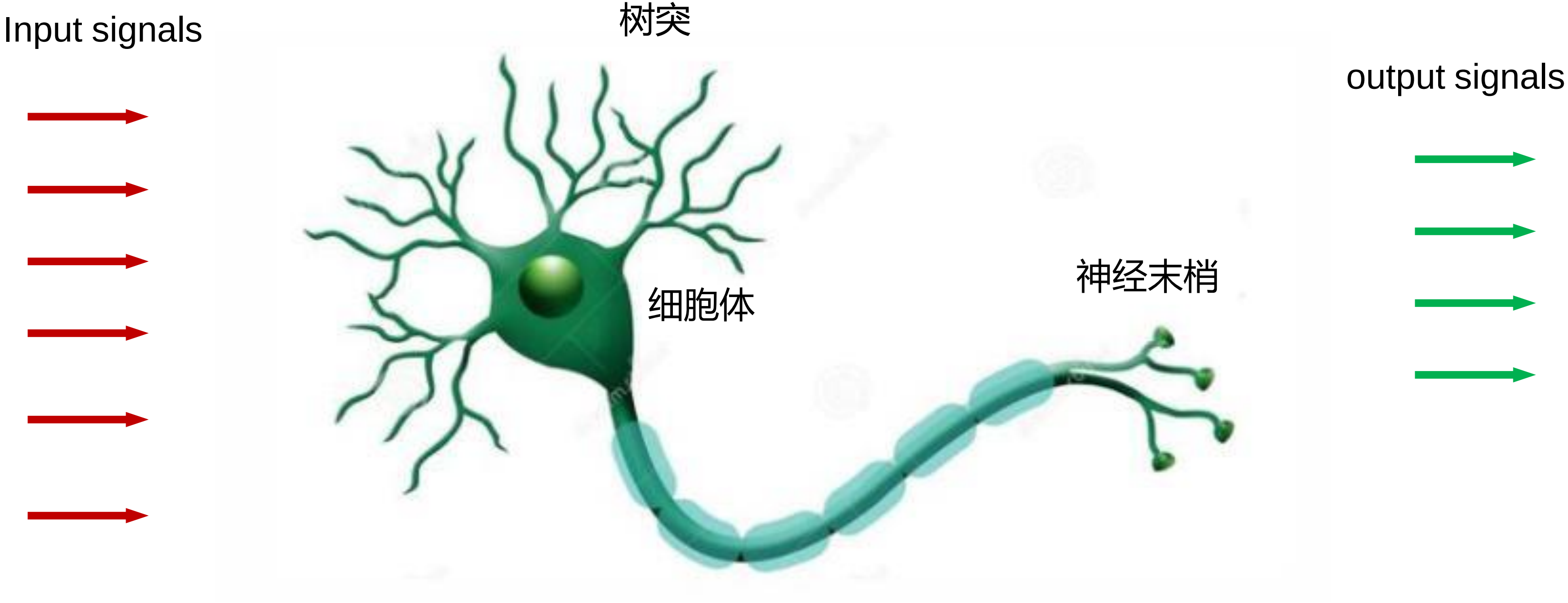
人脑组成

- 860亿个神经元，至少被开发利用了90%
- 每个神经元1万连接(组合连接无限)
- 学习可以增强连接
- 功耗10-23瓦

什么是神经网络?

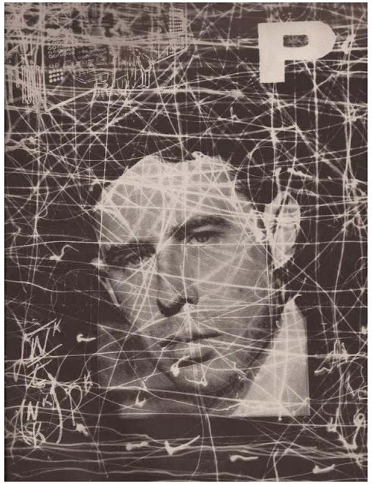


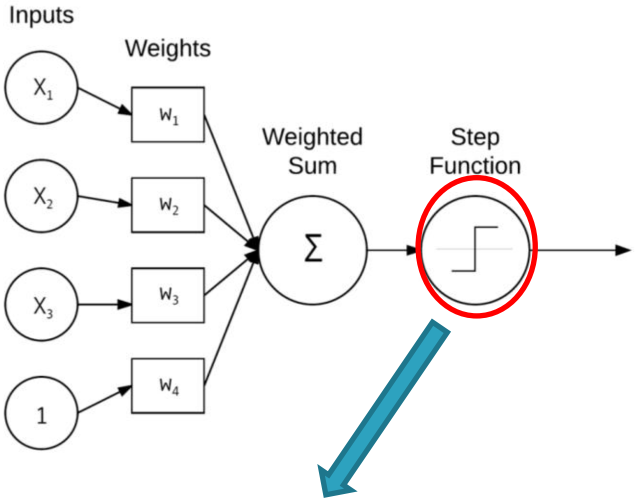
神经元模型

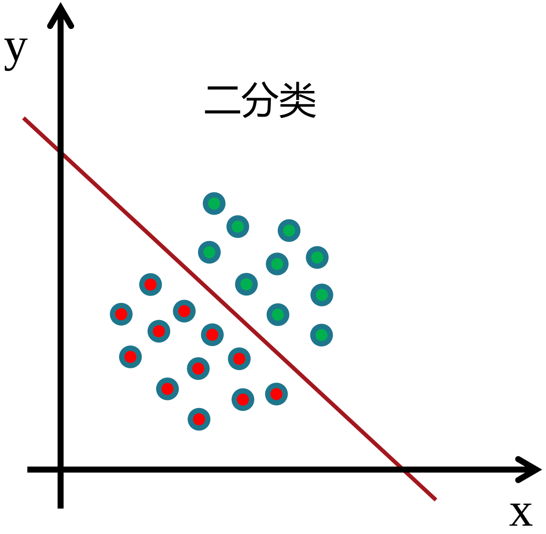


感知机模型-1958年

Frank Rosenblatt

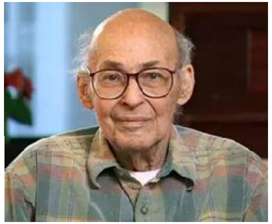






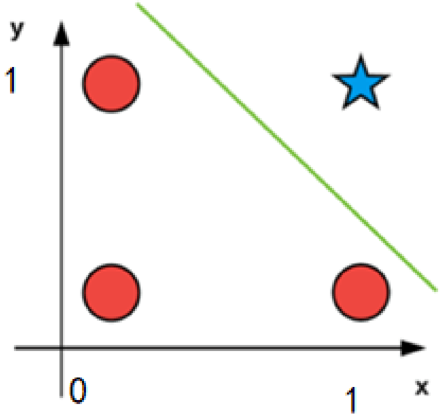
知识点：非线性 $\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$

感知机-逻辑表示

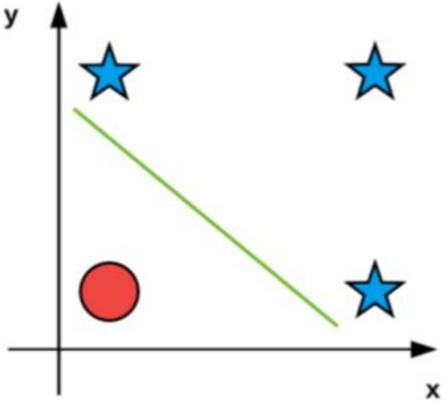


马文 明斯基
《感知机》-1969年

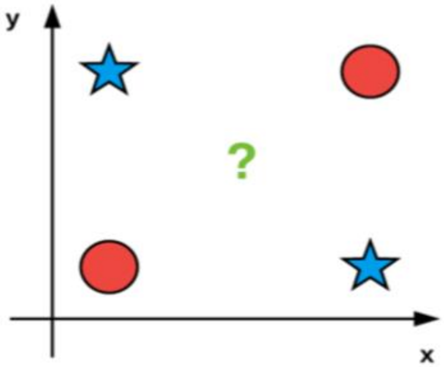
AND



OR



XOR

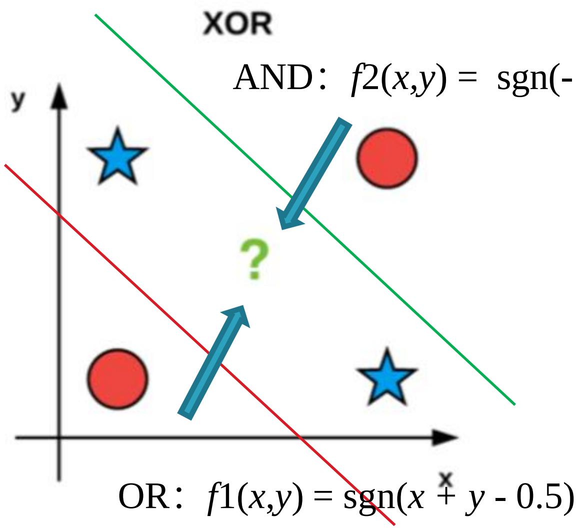


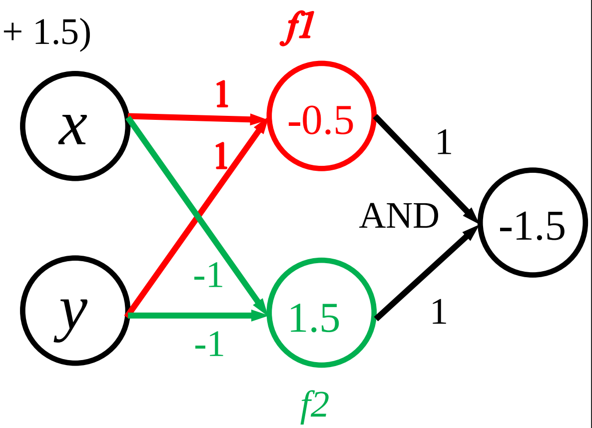
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1

0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1

0 ^ 0 = 0
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0

异或问题-感知机

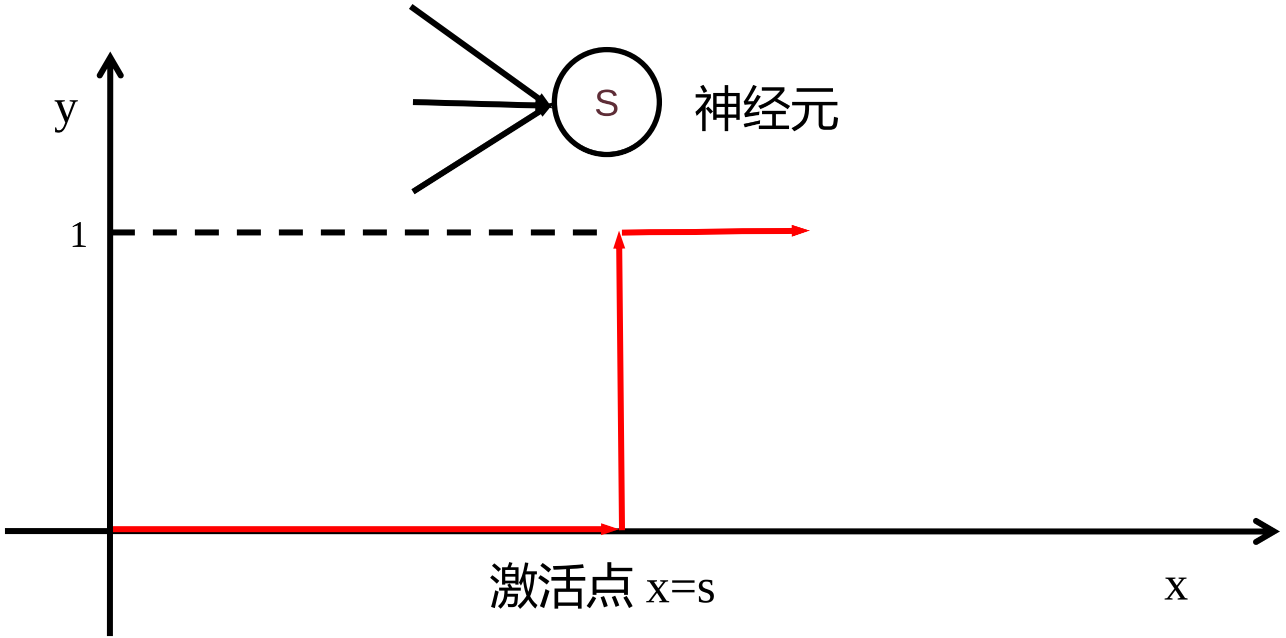




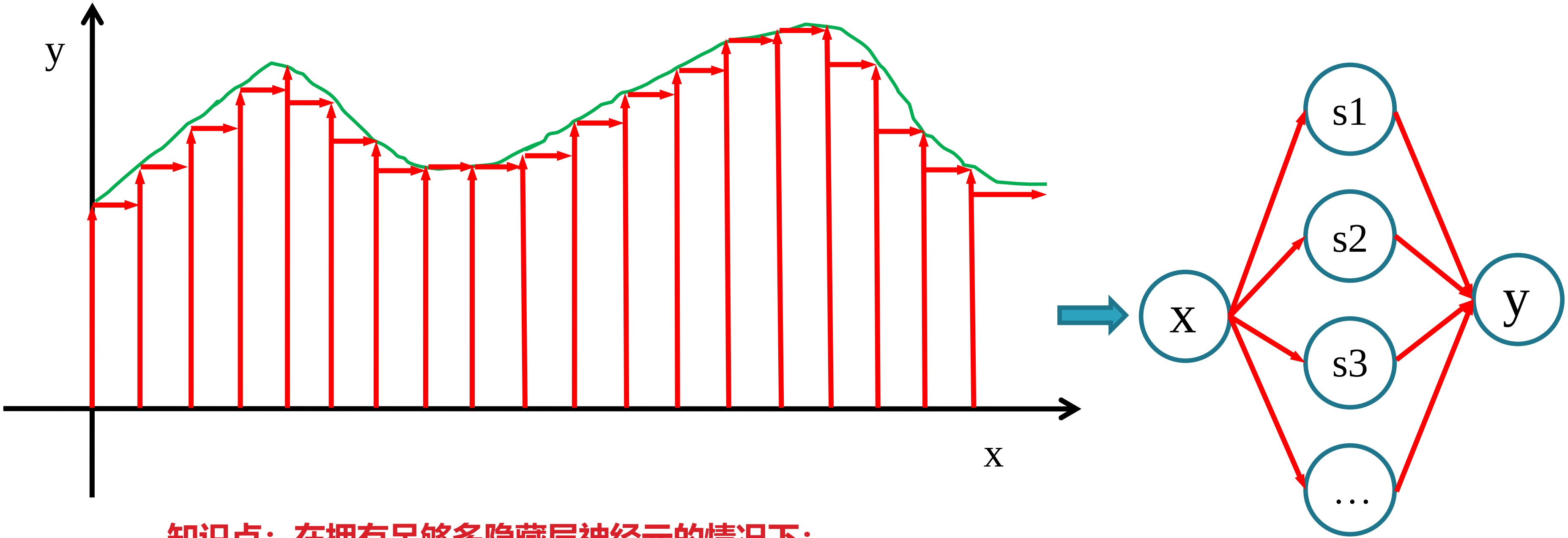

```python
import numpy as np
def step(x):
    return (np.sign(x)+1)/2
def f1(x,y):
    return step(x+y-0.5)
def f2(x,y):
    return step(-x-y+1.5)
def xor(x,y):
    return step(f1(x,y)+f2(x,y)-1.5)

x = np.array([0,0,1,1])
y = np.array([0,1,0,1])
z = xor(x,y)
for i in range(len(x)):
    print("xor(%d,%d)=%d" % (x[i],y[i],z[i]))
```

知识点：深层网络比浅层网络拥有更强的表达能力。

神经网络的表达能力

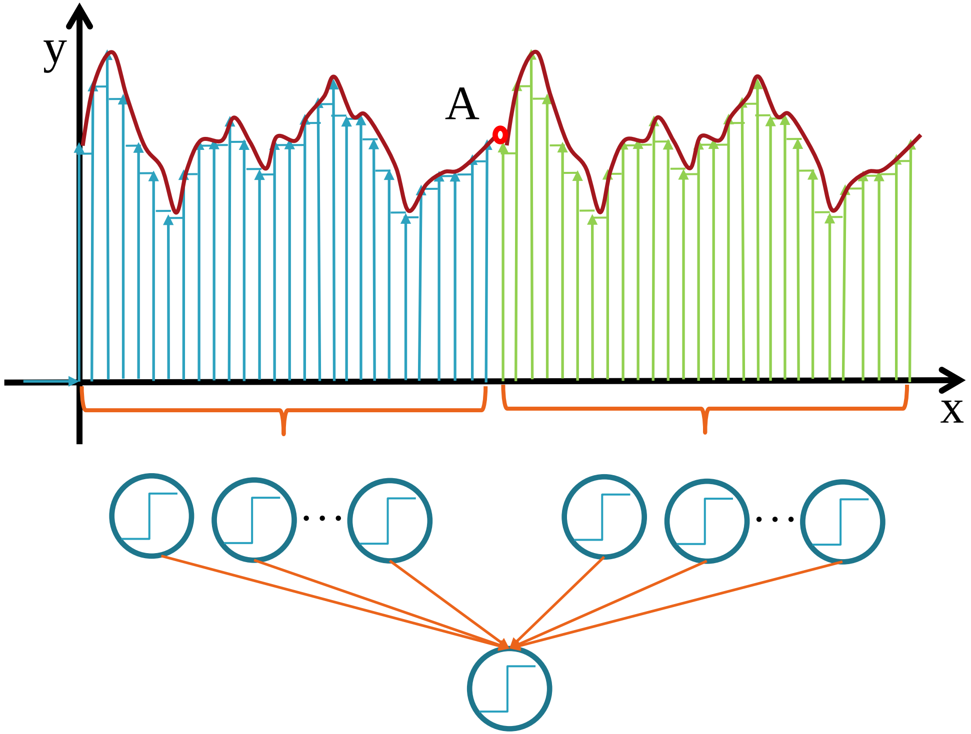


神经网络的表达能力

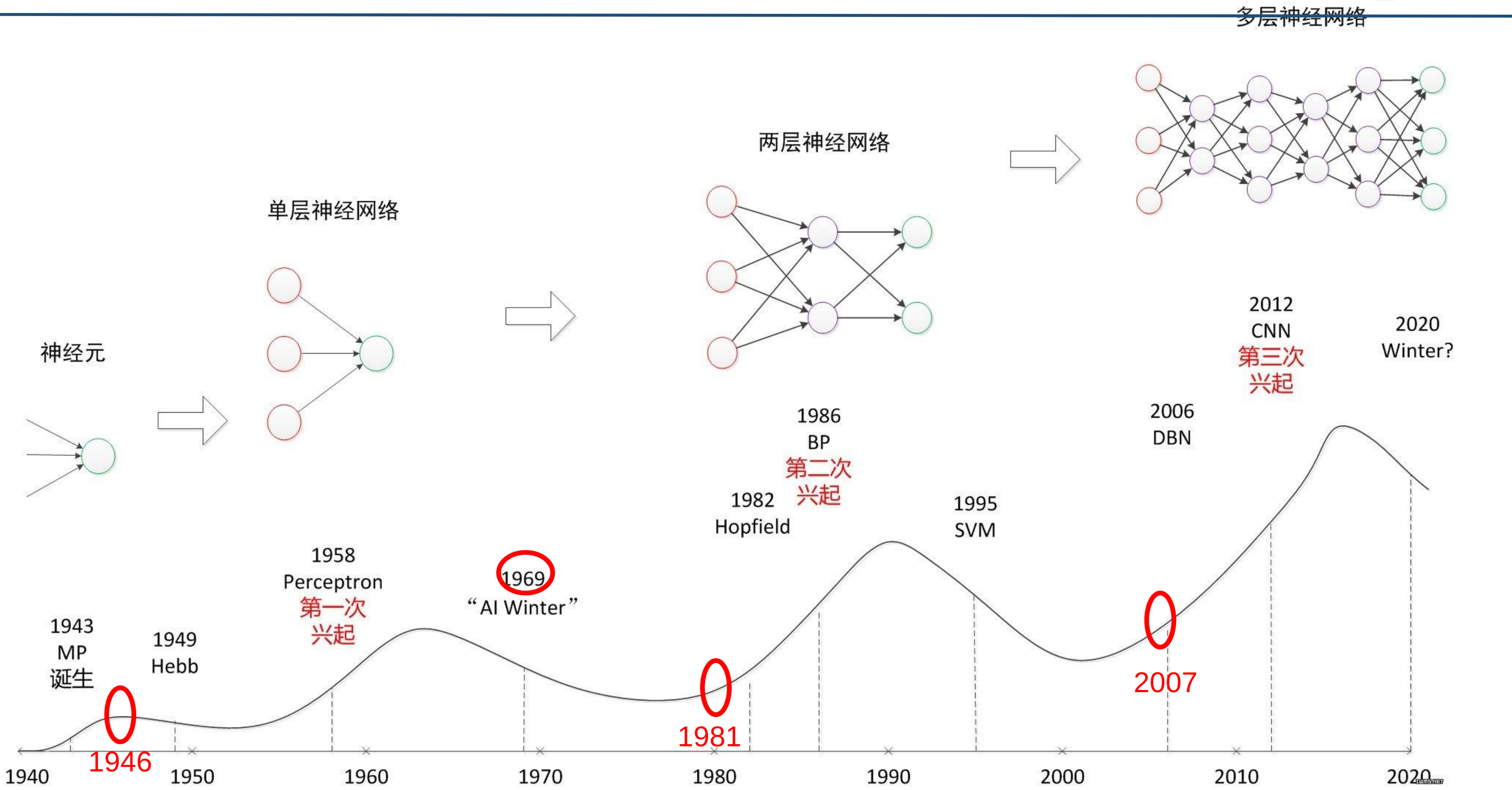


知识点：在拥有足够多隐藏层神经元的情况下：
两层神经网络(单隐含层)可以表示任意连续函数，三层神经网络可以表示任意函数。

神经网络的表达能力



神经网络发展历史



多层神经网络
两层神经网络
单层神经网络
神经元
2012
CNN
第三次
兴起
2020
Winter?
2006
DBN
1986
BP
第二次
兴起
1982
Hopfield
1995
SVM
1958
Perceptron
第一次
兴起
1969
"AI Winter"
1943
MP
诞生
1949
Hebb
1946
1981
2007
1940
1950
1960
1970
1980
1990
2000
2010
2020

知识点总结

- 激活函数非线性
- 深层网络比浅层网络拥有更强的表达能力
- 三层神经网络可以表示任意函数（足够隐藏单元）

- 神经网络与图灵机(现代计算机)等价
- 计算性能的发展带动神经网络的发展
- 深层神经网络可以充分利用计算机的性能

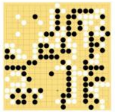
- 语音识别

$f($  $)=$ “你好”

- 图像识别

$f($  $)=$ “9”

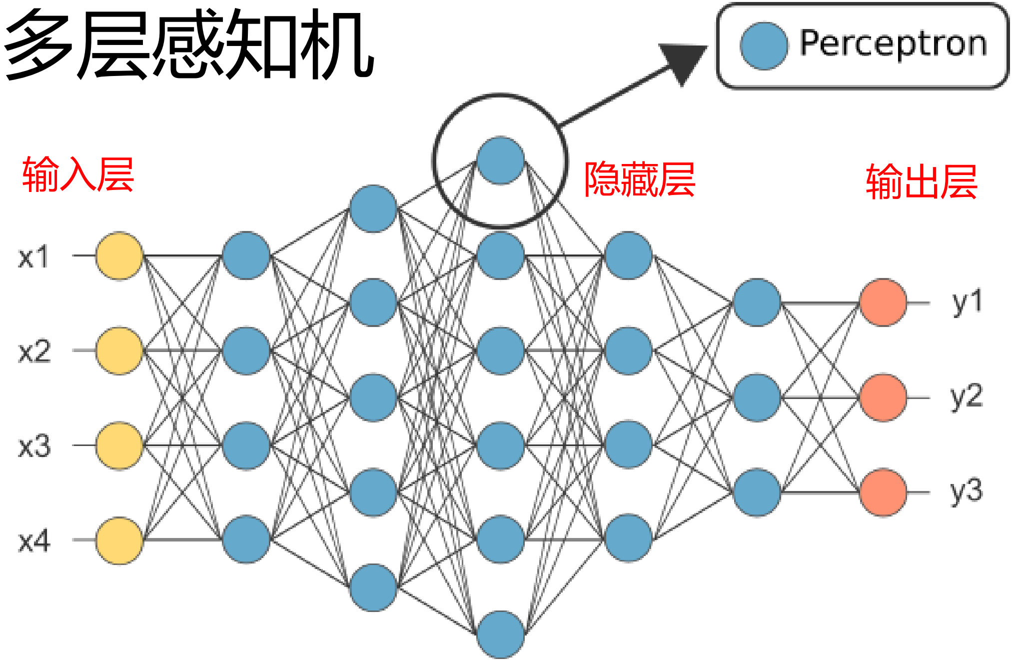
- 围棋

$f($  $)=$ “6-5” （落子位置）

- 机器翻译

$f($ “你好！” $)=$ “Hello!”

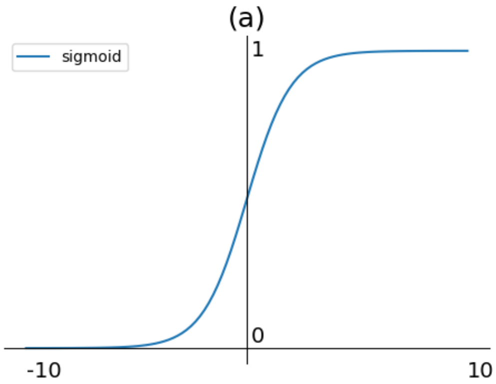
多层感知机

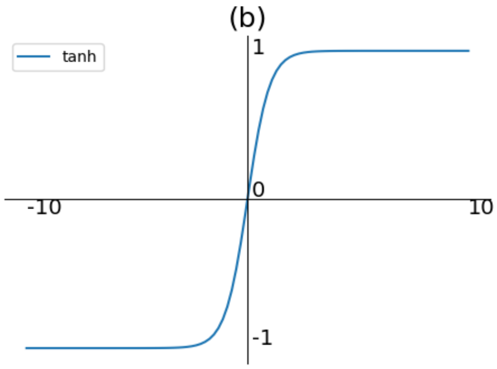


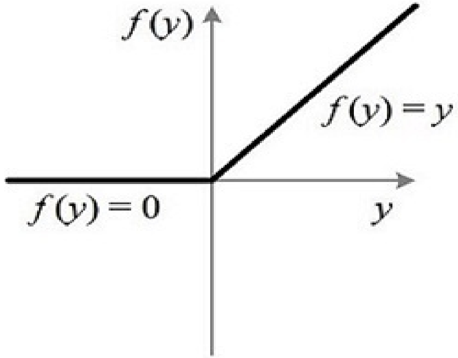
从前有座山，山上有座庙，庙里有个老和尚，还有一个小和尚。有一天，老和尚对小和尚说，从前有座山，山上有座庙，庙里有个老和尚，还有一个小和尚。有一天，老和尚对小和尚说，从前有座山，山上有座庙，庙里有个老和尚，还有一个小和尚。有一天，老和尚对小和尚说……

1. 复合函数 $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$
2. 激活函数
3. 梯度下降-反向传播算法

激活函数







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

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

$$ReLU(x) = \begin{cases} x & if\, x > 0 \\ 0 & if\, x \leq 0 \end{cases}$$

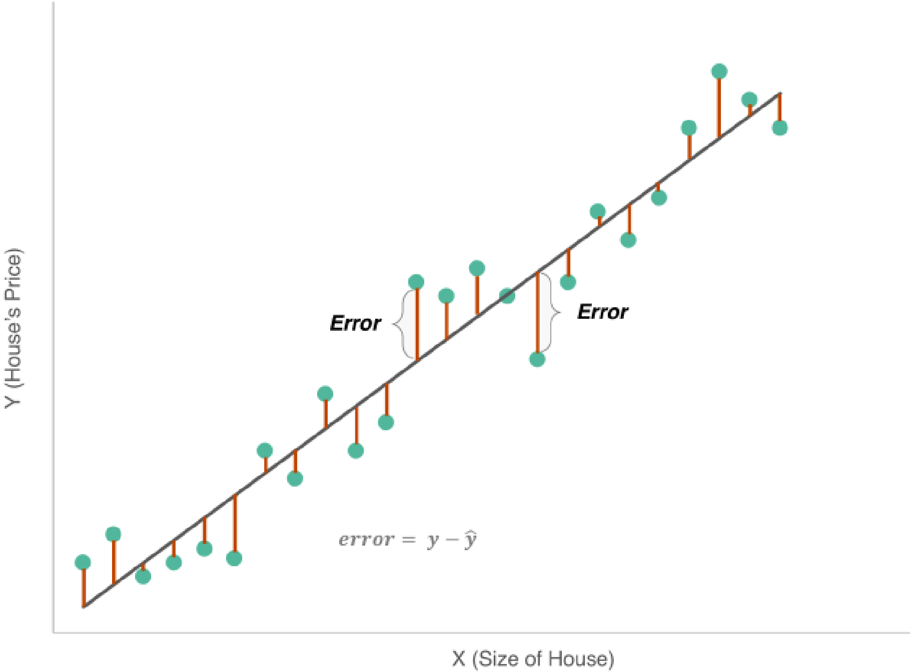
$sigmoid'(x) = sigmoid(x)(1 - sigmoid(x))$

$tanh'(x) = 1 - tanh^2(x)$

```
import numpy as np
def sigmoid(x):
    return 1/(1+np.exp(-x))
```

```
import numpy as np
def tanh(x):
    return (np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
```

监督学习和损失函数(Loss Function)



To minimize $J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$

$$H(x,y) = -\sum_{i=i}^{n} x_i \ln y_i$$

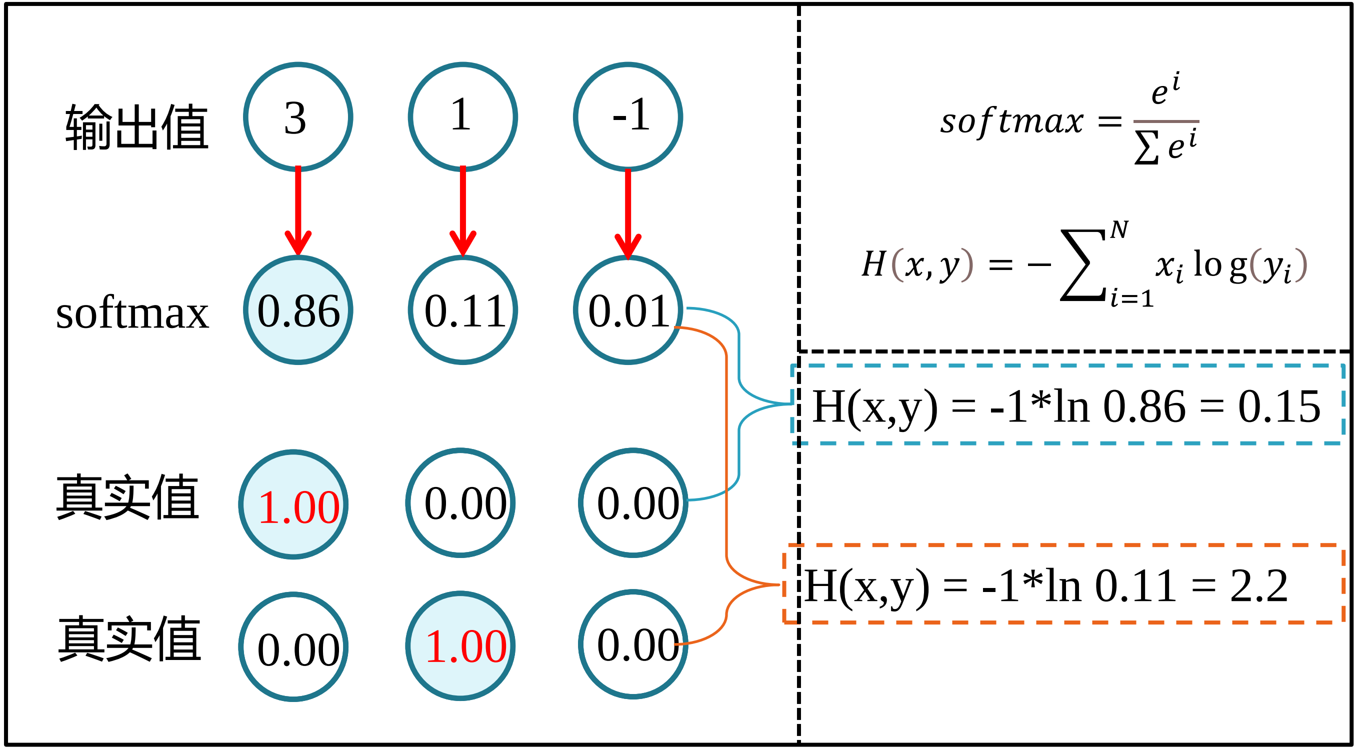
$$S_i = \frac{e^{V_i}}{\sum_j e^{V_j}}$$

One-hot编码

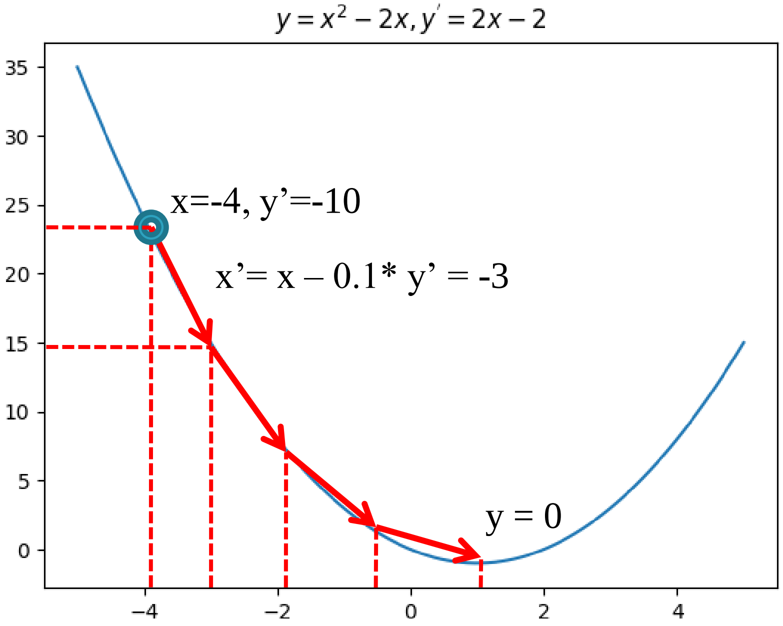
小学 -> [1, 0, 0, 0, 0]
中学 -> [0, 1, 0, 0, 0]
大学 -> [0, 0, 1, 0, 0]
硕士 -> [0, 0, 0, 1, 0]
博士 -> [0, 0, 0, 0, 1]

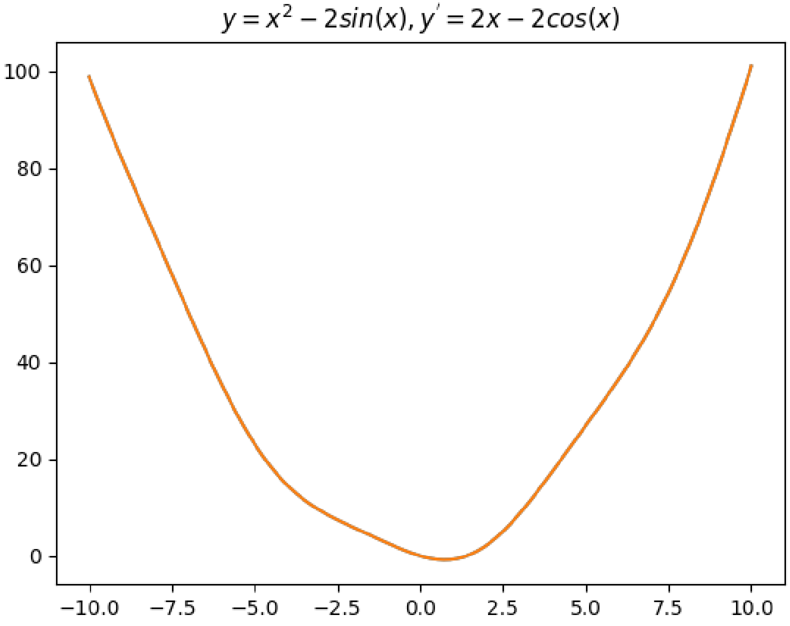
y_0	0
y_1	1
y_2	0
y_3	0
y_4	0
y_5	0
y_6	0
y_7	0
y_8	0
y_9	0

损失函数(Loss Function) - 交叉熵



梯度下降算法(Gradient Descent)

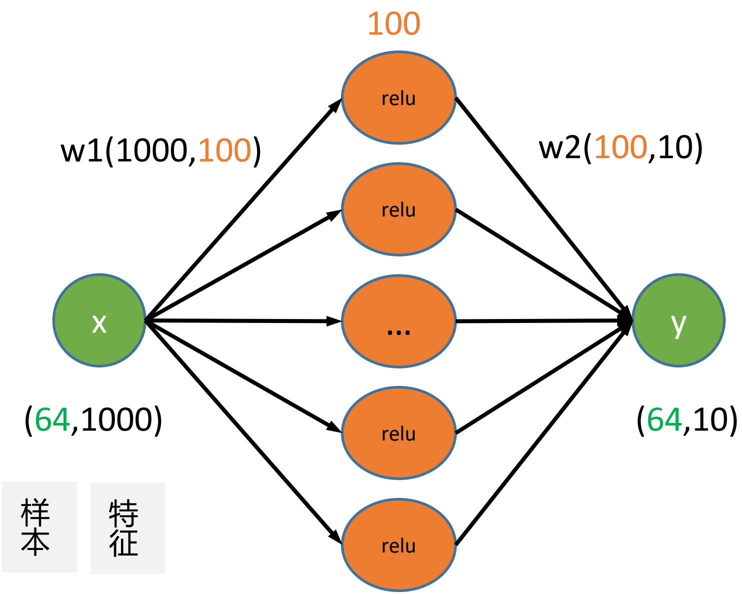




令 y' = 0 ⇨ x=1, y = 0

y' = x – cos(x) = 0, x = ?

前向传播-维度变换




```python
import numpy as np
x = np.random.randn(64, 1000)
y = np.random.randn(64, 10)
w1 = np.random.randn(1000, 100)
w2 = np.random.randn(100, 10)

h = x.dot(w1)   # (64,100)
h_relu = np.maximum(h, 0) # (64,100)
y_preb = h_relu.dot(w2) # (64,10)
```

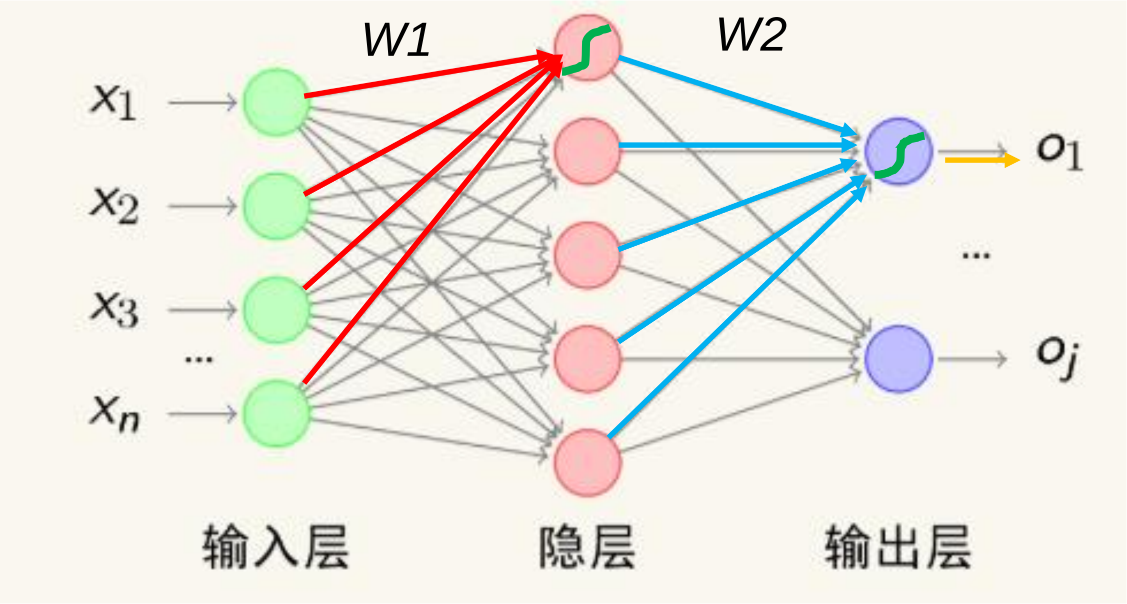
$z = x * w1$

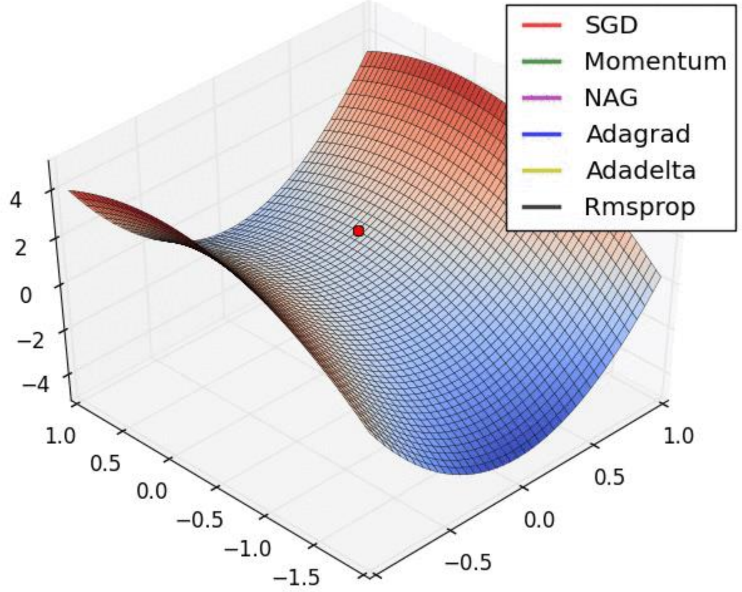
$a = relu(z)$

$y = a * w2$

$loss = (y\_pre - y)^2$

反向传播算法(BP)(链式求导)



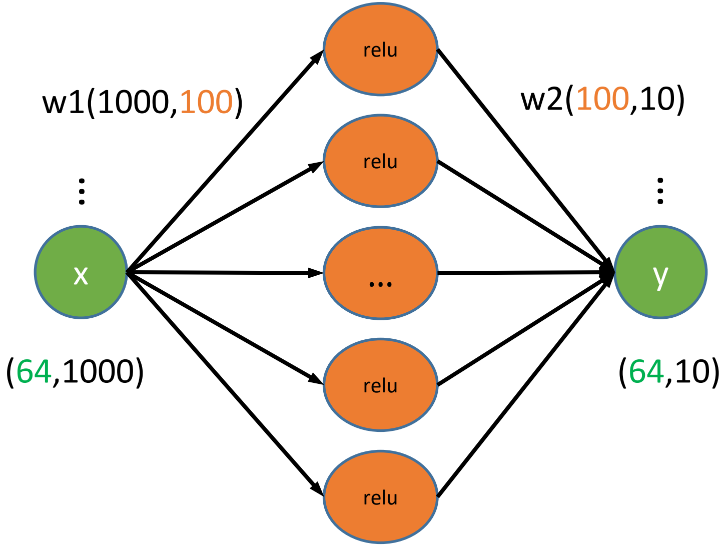


$$w = w + \alpha \frac{\partial L(w,b)}{\partial w}$$

$$b = b + \alpha \frac{\partial L(w,b)}{\partial b}$$

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

BPNN-梯度下降




```python
import numpy as np
x = np.random.randn(64, 1000)
y = np.random.randn(64, 10)
w1 = np.random.randn(1000, 100)
w2 = np.random.randn(100, 10)
learning_rate = 1e-6
for i in range(1000):
    h = x.dot(w1)  # (64,100)
    h_relu = np.maximum(h, 0) # (64,100)
    y_preb = h_relu.dot(w2) # (64,10)
    loss = np.square(y_preb - y).sum()

    grad_w2 = …
    grad_w1 = …
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

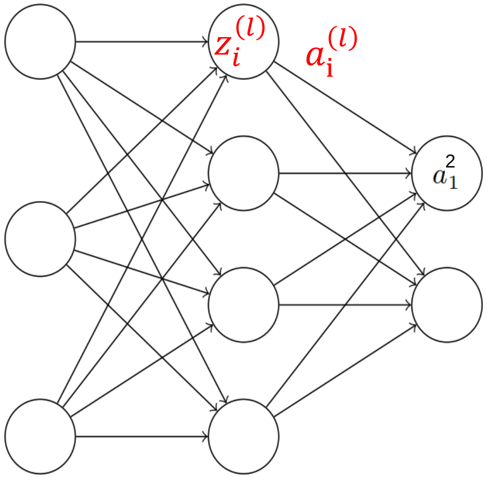
$z = x * w1$

$a = relu(z)$

$y = a * w2$

$loss = (y\_pre - y)^2$

反向传播算法(BP)



$$\frac{\partial c}{\partial a} = \frac{\partial c}{\partial b} \times \frac{\partial b}{\partial a}$$

$$z^{(l)} = w^{(l)} a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = f\left(z^{(l)}\right) = f\left(w^{(l)} a^{(l-1)} + \ b^{(l)}\right)$$

$$J(W,b;x,y) = \frac{1}{2}\left(a^{(L)} - y\right)^2$$

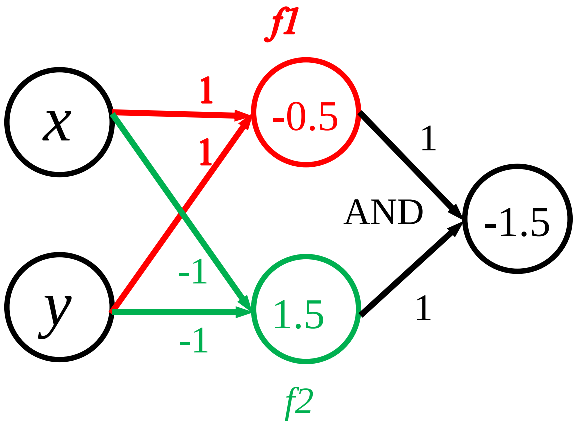
定义残差: $\delta_i^{(l)} = \frac{J(W,b;x,y)}{\partial z_i^{(l)}}$

输出层: $\delta^{(L)} = \frac{\partial J(W,b;x,y)}{\partial z_i^{(L)}} = \left(a^{(L)} - y\right) \cdot \frac{\partial a^{(L)}}{\partial z_i^{(L)}} = \left(a^{(L)} - y\right) \cdot f'\left(z_i^{(L)}\right)$

其它层: $\delta^{(l)} = \frac{\partial J}{\partial Z_{ij}^{(l)}} = \frac{\partial J}{\partial Z_{ij}^{(l+1)}} \frac{\partial Z_{ij}^{(l+1)}}{\partial Z_{ij}^{(l)}} = \left(\left(W^{(l+1)}\right)^T \delta^{(l+1)}\right) \cdot f'\left(z^{(l)}\right)$

梯度: $\frac{\partial J(W,b;x,y)}{\partial w_{ij}^{(l)}} = \frac{\partial J}{\partial Z_{ij}^{(l)}} \frac{\partial Z_{ij}^{(l)}}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} \cdot a_j^{(l-1)}$

BPNN



$f(0,0) = 0$
$f(0,1) = 1$
$f(1,0) = 1$
$f(1,1) = 0$

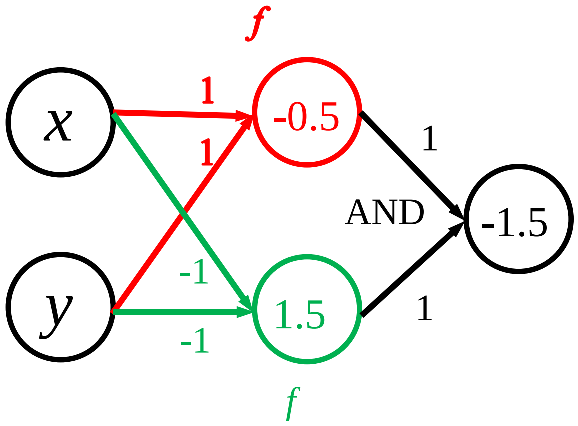
输出层： $\delta^{(L)} = \left(a^{(L)} - y\right)\cdot f'\left(z^{(L)}\right)$

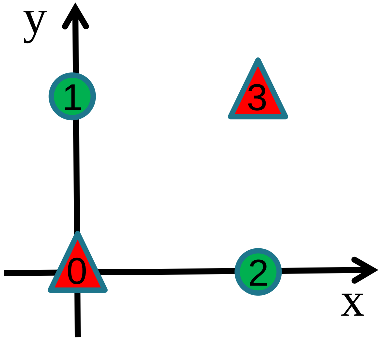
其它层： $\delta^{(l)} = \frac{\partial J}{\partial Z_{ij}^{(l)}} = \frac{\partial J}{\partial Z_{ij}^{(l+1)}}\frac{\partial Z_{ij}^{(l+1)}}{\partial Z_{ij}^{(l)}} = \left(\left(W^{(l+1)}\right)^T\delta^{(l+1)}\right)\cdot f'\left(z^{(l)}\right)$

梯度： $\frac{\partial C}{\partial w_{ij}^{(l)}} = \frac{\partial C}{\partial z_{ij}^{(l)}}\frac{\partial z_{ij}^{(l)}}{\partial w_{ij}^{(l)}} = \delta_i^{(l)}\cdot a_j^{(l-1)}$

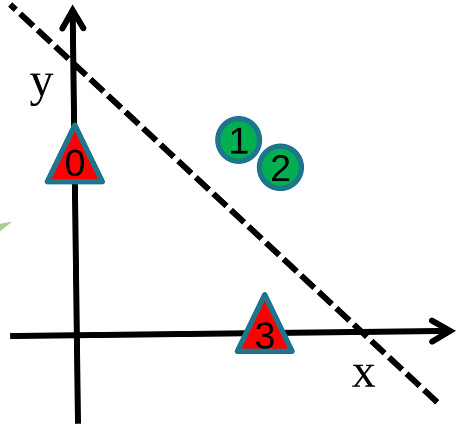
```
import numpy as np

def act(x, deriv=False):
    if(deriv==True):
        return x*(1-x)#x is activated a(z)
    return 1/(1+np.exp(-x))

NH = 10
x = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])
w1 = 2*np.random.random((2,NH))-1
w2 = 2*np.random.random((NH,1))-1

def feedfoward(x):
     a0 = x;
     a1 = act(np.dot(a0,w1))
     a2 = act(np.dot(a1,w2))
     return (a0,a1,a2)

n_epochs = 1000000
for i in range(n_epochs):
     a0,a1,a2 = feedfoward(x)
     l2_delta = (a2 - y)*act(a2, deriv=True)
     l1_delta = l2_delta.dot(w2.T) * act(a1,deriv=True)
     w2 = w2 - a1.T.dot(l2_delta)*0.1
     w1 = w1 - a0.T.dot(l1_delta)*0.1
     if(i % 10000) ==0:
          loss =np.mean(np.abs(y - a2))
          print("epochs %d/%d loss = %f" % (i/1e4+1, n_epochs/1e4, loss))

a0,a1,a2 = feedfoward(x)
print ("xor(",x,") = ",a2)
```

BPNN-非线性变换





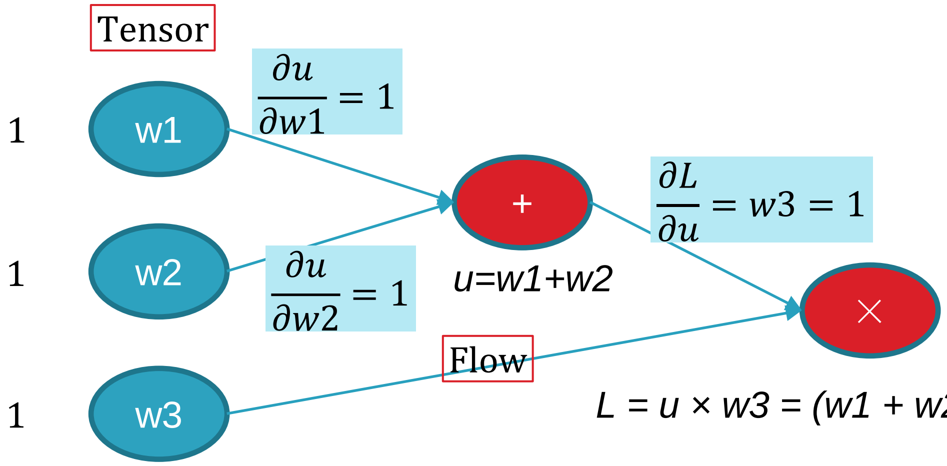
点0 f: [0, 0] - [0,1]
点1 f: [0, 1] - [1,1]
点2 f: [1, 0] - [1,1]
点3 f: [1, 1] - [1,0]



计算图与自动微分

$$\frac{\partial f(\mathbf{x})}{\partial x} \approx \frac{f(\mathbf{x}+h)-f(\mathbf{x})}{h} \qquad \frac{\partial f(\mathbf{x})}{\partial x} \approx \frac{f(\mathbf{x}+h)-f(\mathbf{x}-h)}{2h}$$

链式规则 $\frac{\partial c}{\partial a} = \frac{\partial c}{\partial b} \times \frac{\partial b}{\partial a}$ $\qquad \frac{\partial a_n}{\partial a_1} = \frac{\partial a_n}{\partial a_{n-1}} \cdots \frac{\partial a_3}{\partial a_2} \cdot \frac{\partial a_2}{\partial a_1}$



$\frac{\partial u}{\partial w1} = 1$

$\frac{\partial u}{\partial w2} = 1$

$\frac{\partial L}{\partial u} = w3 = 1$

u=w1+w2

L = u × w3 = (w1 + w2) × w3

$$\frac{\partial L}{\partial w1} = \frac{\partial L}{\partial u} \times \frac{\partial u}{\partial w1} = 1 \times 1 = 1$$

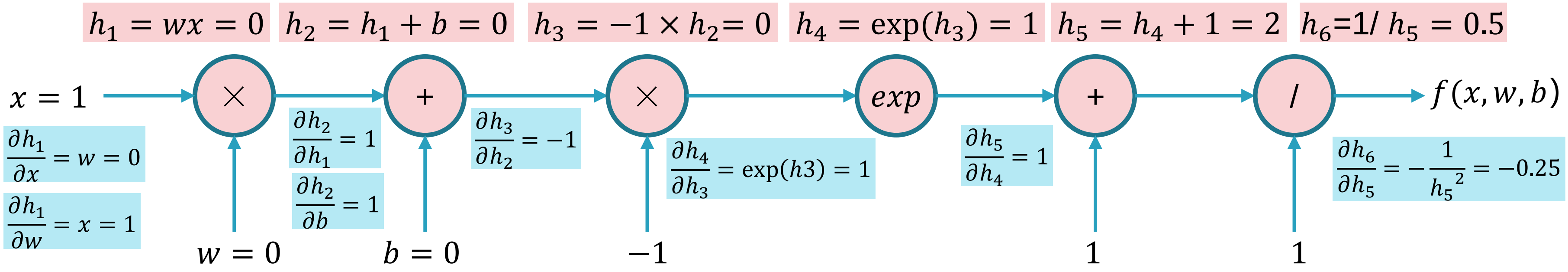
$$\frac{\partial L}{\partial w2} = \frac{\partial L}{\partial u} \times \frac{\partial u}{\partial w2} = 1 \times 1 = 1$$

$$\frac{\partial L}{\partial w3} = u = w1 + w2 = 1 + 1 = 2$$

知识点：计算图可通过链式法则，自动计算微分

逆向自动微分 $f(x,w,b)=\dfrac{1}{exp(-(wx+b))+1}$

当 (x, w, b) = (1, 0, 0) 时计算图如下：



$$\frac{\partial f(x;w,b)}{\partial w} = \frac{\partial f(x;w,b)}{\partial h_6}\frac{\partial h_6}{\partial h_5}\frac{\partial h_5}{\partial h_4}\frac{\partial h_4}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial w}$$

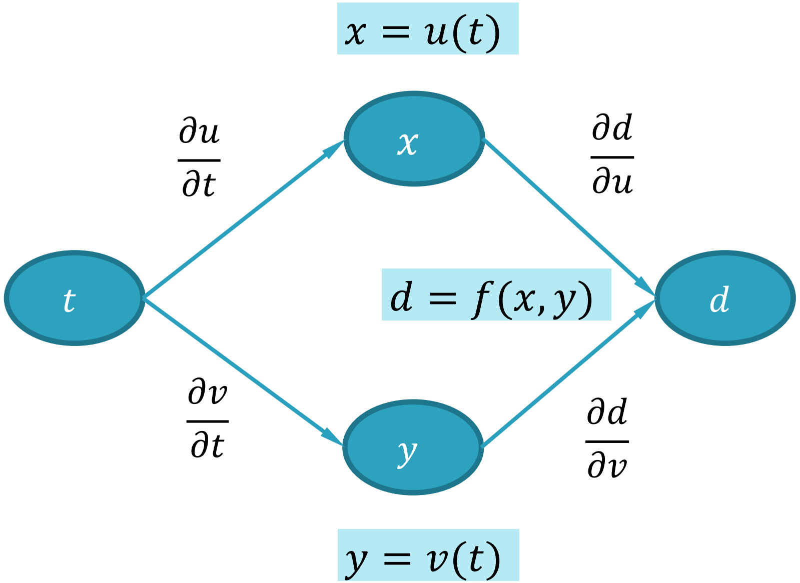
$$\frac{\partial f(x;w,b)}{\partial h_2} = \frac{\partial f(x;w,b)}{\partial h_6}\cdot\frac{\partial h_6}{\partial h_5}\cdot\frac{\partial h_5}{\partial h_4}\cdot\frac{\partial h_4}{\partial h_3}\cdot\frac{\partial h_3}{\partial h_2} = 1 \times -0.25 \times 1 \times 1 \times -1 = 0.25$$

$$\frac{\partial f(x;w,b)}{\partial b} = \boxed{\frac{\partial f(x;w,b)}{\partial h_2}}\cdot\frac{\partial h_2}{\partial b} = 0.25 \times 1 = 0.25$$

计算复用，提高计算效率

知识点：反向传播算法只是自动微分的一种特殊形式

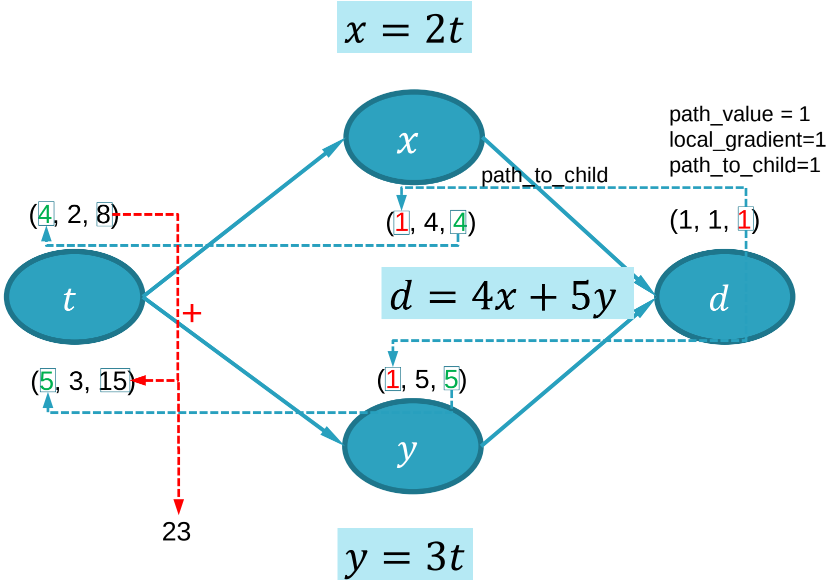
逆向自动微分框架



$$\frac{\partial d}{\partial t} = \frac{\partial d}{\partial u}\frac{\partial u}{\partial t} + \frac{\partial d}{\partial v}\frac{\partial v}{\partial t}$$

```python
from collections import defaultdict
class Variable:
    def __init__(self, value, local_gradients=[]):
        self.value = value
        self.local_gradients = local_gradients
    def __add__(self, other):
        return add(self, other)

def add(a, b):
    value = a.value + b.value
    local_gradients = ((a, 1), (b, 1))
    return Variable(value, local_gradients)

def get_gradients(variable):
    gradients = defaultdict(lambda: 0)
    def compute_gradients(variable, path_value):
        for child_variable, local_gradient in variable.local_gradients:
            value_of_path_to_child = path_value * local_gradient
            gradients[child_variable] += value_of_path_to_child
            compute_gradients(child_variable, value_of_path_to_child)

    compute_gradients(variable, path_value=1)
    return gradients

x = Variable(1)
b = Variable(0)
y = x+b+b
gradients = get_gradients(y)
print('dy/dx=%f, dy/db=%f' % (gradients[x], gradients[b]))
```

逆向自动微分框架

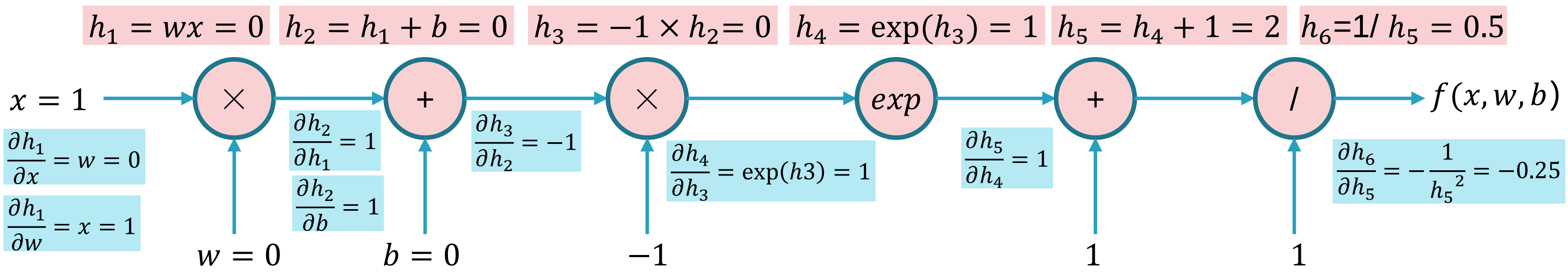


d = 4x+5y=4*(2t)+5*(3t)=23t

```python
def add(a, b):
    value = a.value + b.value
    local_gradients = ((a, 1), (b, 1))
    return Variable(value, local_gradients)
def get_gradients(variable):
    gradients = defaultdict(lambda: 0)
    def compute_gradients(variable, path_value):
        for child_variable, local_gradient in variable.local_gradients:
            value_of_path_to_child = path_value * local_gradient
            gradients[child_variable] += value_of_path_to_child
            compute_gradients(child_variable, value_of_path_to_child)

    compute_gradients(variable, path_value=1)
    return gradients
```

逆向自动微分 $f(x,w,b)=\dfrac{1}{exp(-(wx+b))+1}$

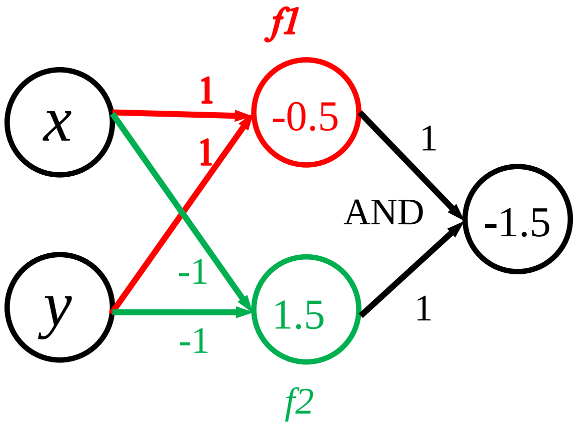
当 (x, w, b) = (1, 0, 0) 时计算图如下：




```
def sigmoid(z):
    ONE = Variable(1)
    return ONE / (ONE + exp(-z))

x = Variable(1)
w = Variable(0)
b = Variable(0)
Y = Variable(1)
y = sigmoid(w * x + b)
gradients = get_gradients(y)
print('dy/dw=%f, dy/db=%f' % (gradients[w],
gradients[b]))
```

```
for i in range(10):
    y = sigmoid(w * x + b)
    C = (y - Y) * (y - Y)
    print('Cost=%f, y=%f' % (C.value, y.value))
    gradients = get_gradients(C)
    w.value = w.value - 0.1 * gradients[w]
    b.value = b.value - 0.1 * gradients[b]
```

自动微分XOR

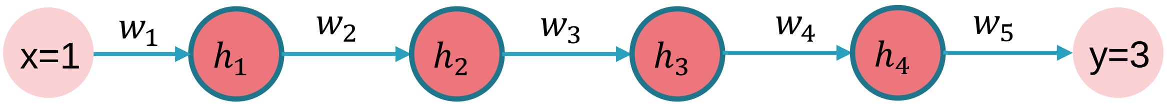


$f(0,0) = 0$
$f(0,1) = 1$
$f(1,0) = 1$
$f(1,1) = 0$

```
def sigmoid(z):
    ONE = Variable(1)
    return ONE/(ONE + exp(-z))

w = [Variable(np.random.randn()) for i in range(9)]
def f(x1, x2):
    x1 = Variable(x1)
    x2 = Variable(x2)
    h1 = sigmoid(x1*w[0]+x2*w[1]+w[2])
    h2 =sigmoid(x1*w[3]+x2*w[4]+w[5])
    ho =sigmoid(w[6]*h1+w[7]*h2+w[8])
    return ho

for i in range(20000):
    C = (f(0, 0) - Variable(0)) * (f(0, 0) - Variable(0))
    C = (f(0, 1) - Variable(1)) * (f(0, 1) - Variable(1)) + C
    C = (f(1, 0) - Variable(1)) * (f(1, 0) - Variable(1)) + C
    C = (f(1, 1) - Variable(0)) * (f(1, 1) - Variable(0)) + C
    print('Epoch %d, Cost=%f' % (i,C.value))
    gradients = get_gradients(C)
    for j in range(len(w)):
        w[j].value = w[j].value - 0.1 * gradients[w[j]]

print("f(0,0)=%f" % f(0,0).value)
print("f(0,1)=%f" % f(0,1).value)
print("f(1,0)=%f" % f(1,0).value)
print("f(1,1)=%f" % f(1,1).value)
```

BP(梯度消失, Vanishing Gradient Problem)

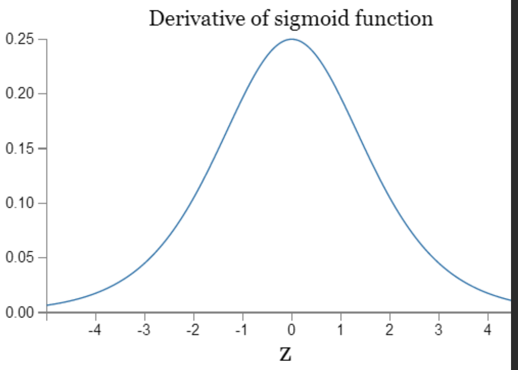
四个隐藏层



$$y = w_5 h_4 = w_5 f(w_4 h_3) = w_5 f(w_4 f(w_3 h_2)) = w_5 f(w_4 f(w_3 f(w_2 h_1))) = w_5 f(w_4 f(w_3 f(w_2 f(w_1 x))))$$

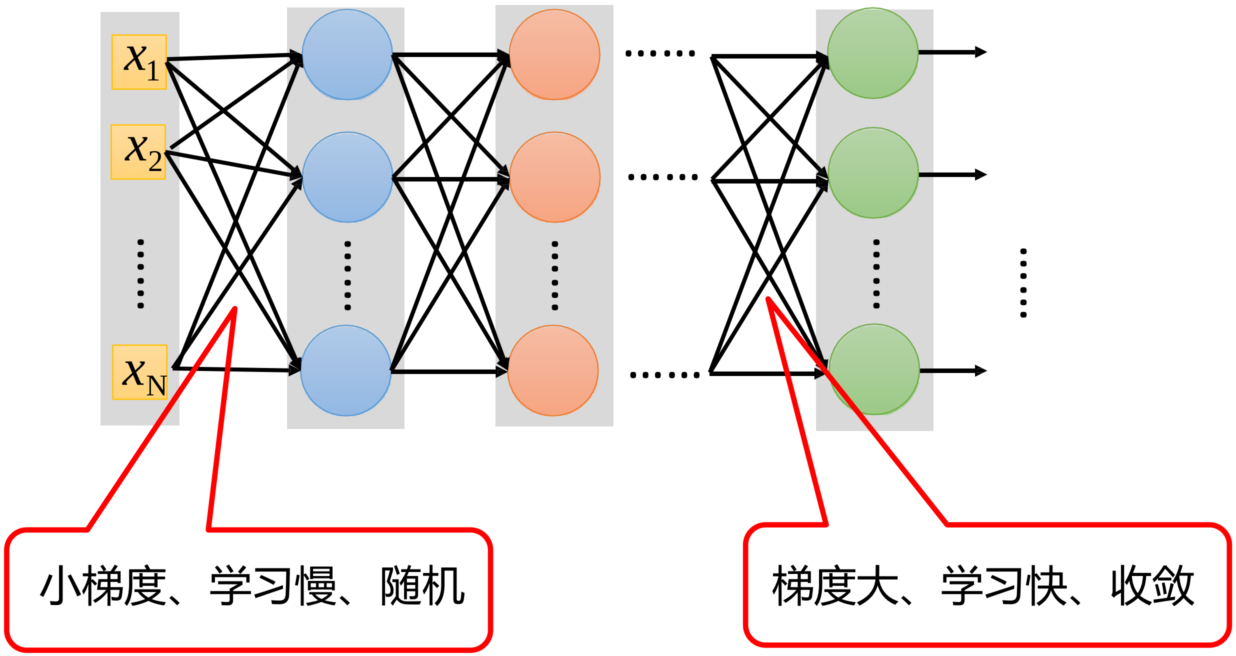
$$\frac{\partial C}{\partial w_1} = \frac{\partial \frac{1}{2}(y-3)^2}{\partial w_1} = (y-3)\frac{\partial y}{\partial w_1} = (y-3)w_5\frac{\partial h_4}{\partial w_1} = (y-3)w_5 w_4 w_3 w_2 \boxed{f'(h_4) f'(h_3)\, f'(h_2)\, f'(h_1)}$$

$$\begin{aligned} f'(z) &= \left(\frac{1}{1+e^{-z}}\right)' \\ &= \frac{e^{-z}}{(1+e^{-z})^2} \\ &= \frac{1+e^{-z}-1}{(1+e^{-z})^2} \\ &= \frac{1}{(1+e^{-z})}\left(1-\frac{1}{(1+e^{-z})}\right) \\ &= f(z)(1-f(z)) \end{aligned}$$

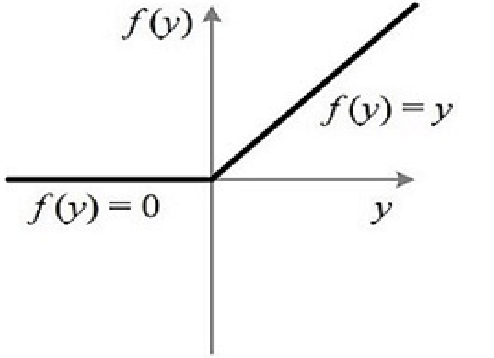



```
w1 = Variable(1)
x = Variable(1)
Y = Variable(1)
y = w1*x
for i in range(4):
    w = Variable(1)
    y = w*sigmoid(w*y)
C = (Y-y)*(Y-y)
gradients = get_gradients(C)
print('dC/dw1=%f' % gradients[w1])
```

梯度消失, Vanishing Gradient Problem

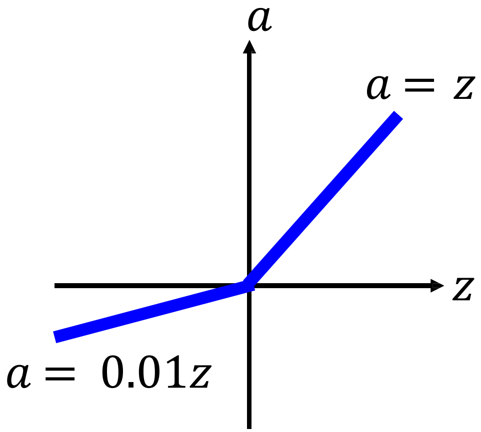


激活函数选取

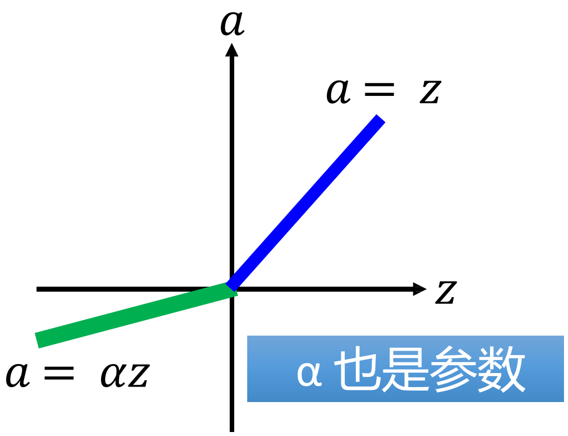


$$ReLU(x) = \begin{cases} x & if x > 0 \\ 0 & if x \le 0 \end{cases}$$

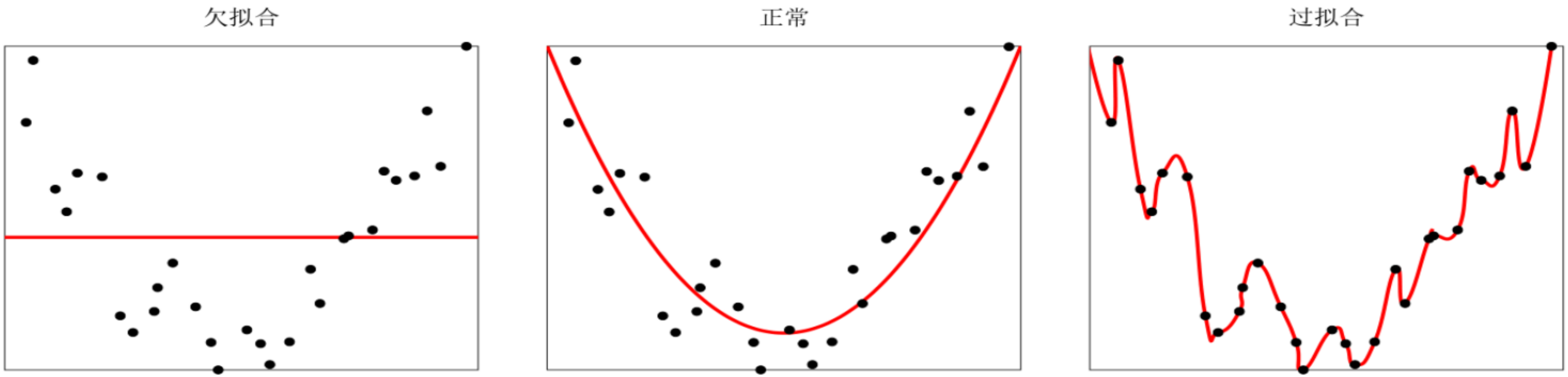
$Leaky\ \ ReLU$



带参数的 $ReLU$

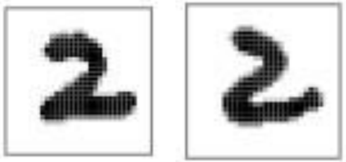


经验风险最小化与过拟合

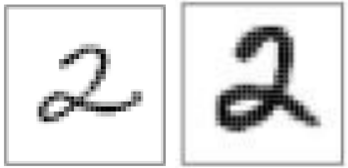


过拟合的原因

Training Data:



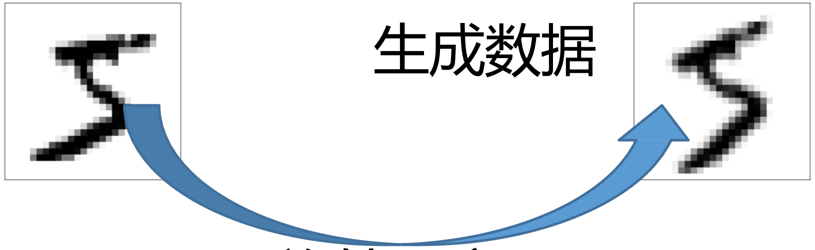
Testing Data:



学习目标由训练数据定义，与测试数据无关

原始数据

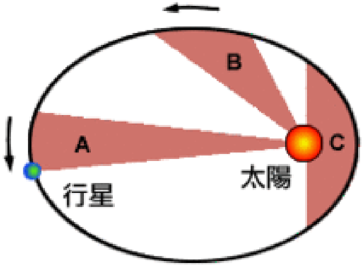
生成数据



旋转15度

过拟合的原因

第谷 -- 数据



开普勒三定律

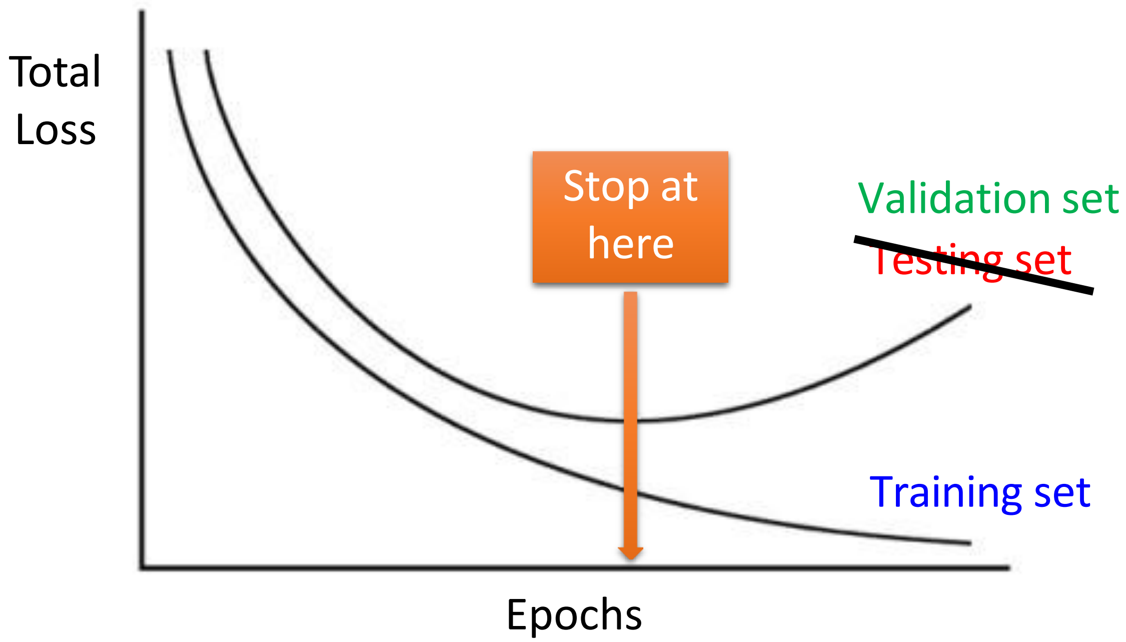
开普勒 – 模型

- 数据问题：数据量少，数据误差较大
- 模型问题： 模型太复杂 (奥卡姆剃刀原理)
- 数据比模型更重要

- 早停策略
- 采集更多样本数据
- 正则化方法

- 研究模型结构和优化方法

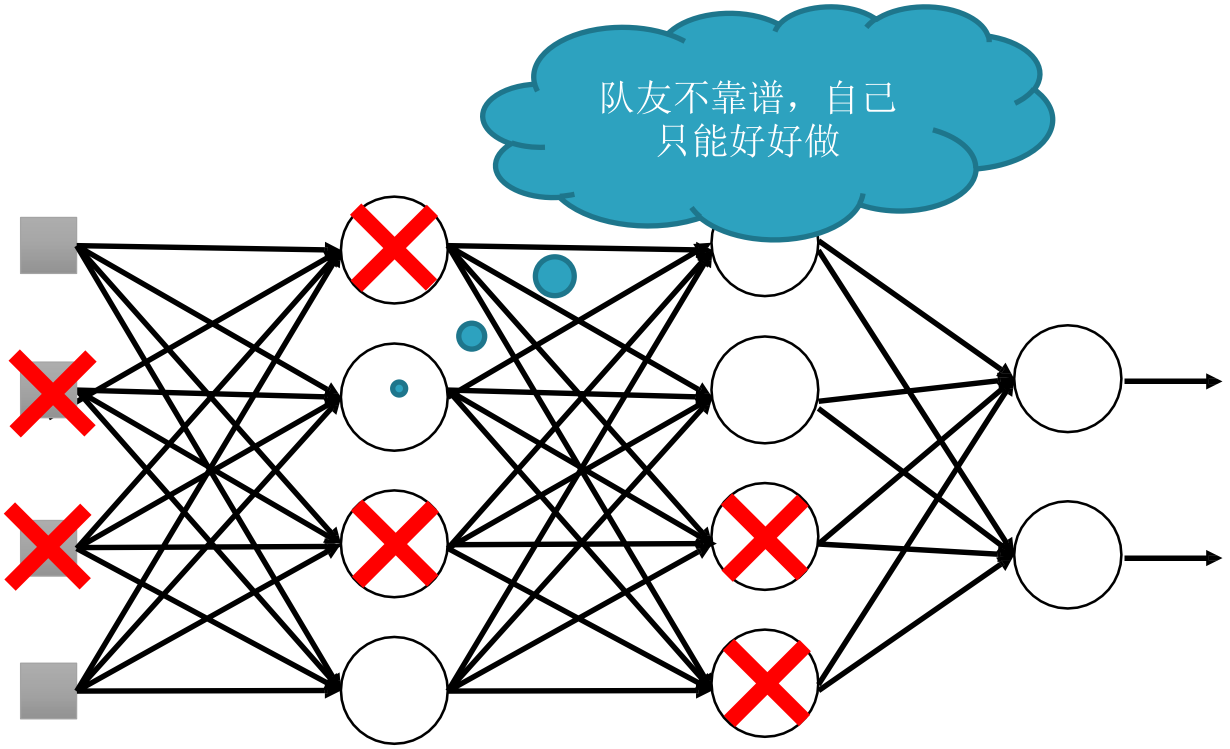
Early Stopping



Keras: http://keras.io/getting-started/faq/#how-can-i-interrupt-training-when-the-validation-loss-isnt-decreasing-anymore

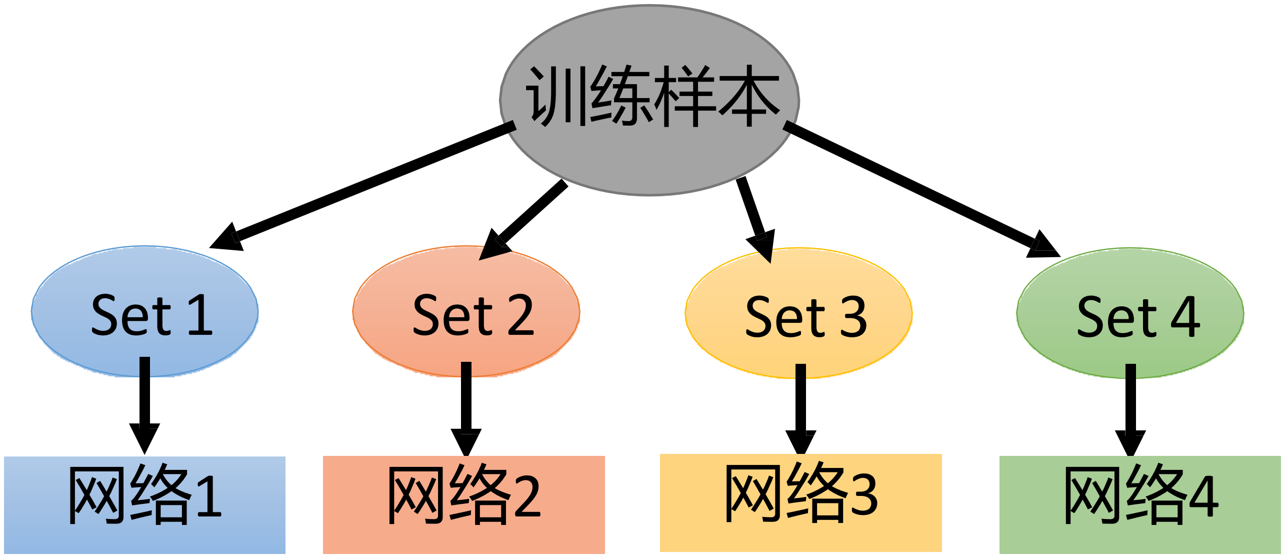
Dropout

Training:

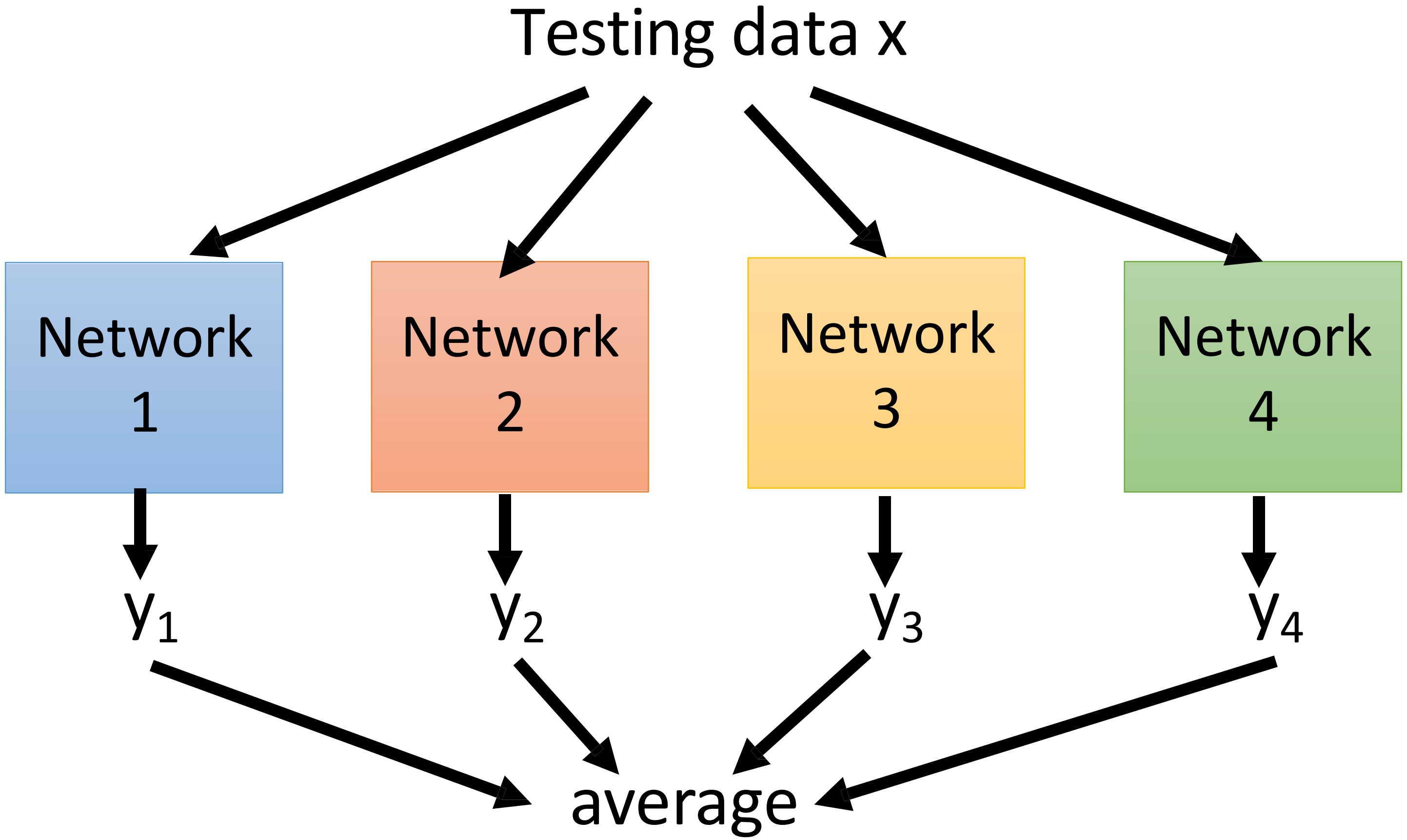


- **Each time before updating the parameters**
  - Each neuron has p% to dropout

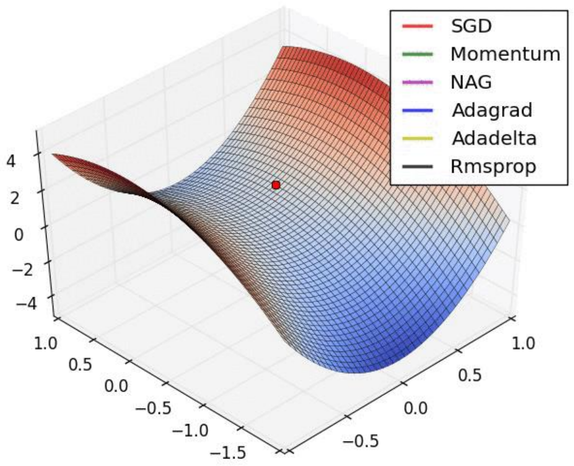
Dropout-模型集成Ensemble



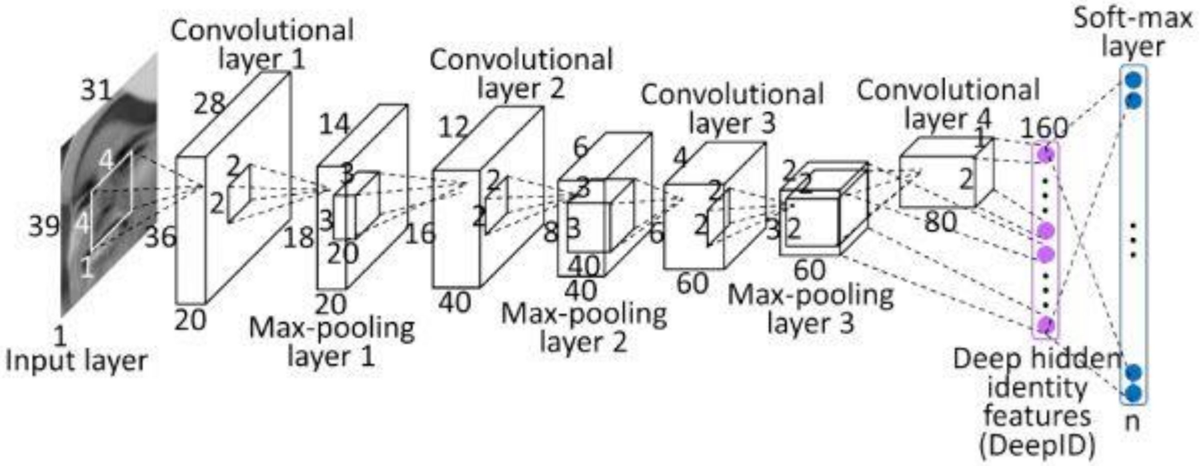
Dropout-模型集成Ensemble



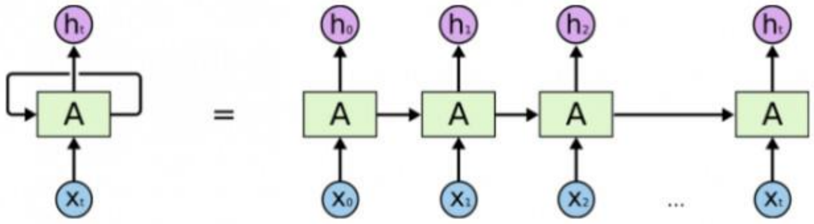
研究方向



SGD
Momentum
NAG
Adagrad
Adadelta
Rmsprop



Input layer
Convolutional layer 1
Max-pooling layer 1
Convolutional layer 2
Max-pooling layer 2
Convolutional layer 3
Max-pooling layer 3
Convolutional layer 4
Deep hidden identity features (DeepID)
Soft-max layer



A
=

Thank you! Questions?