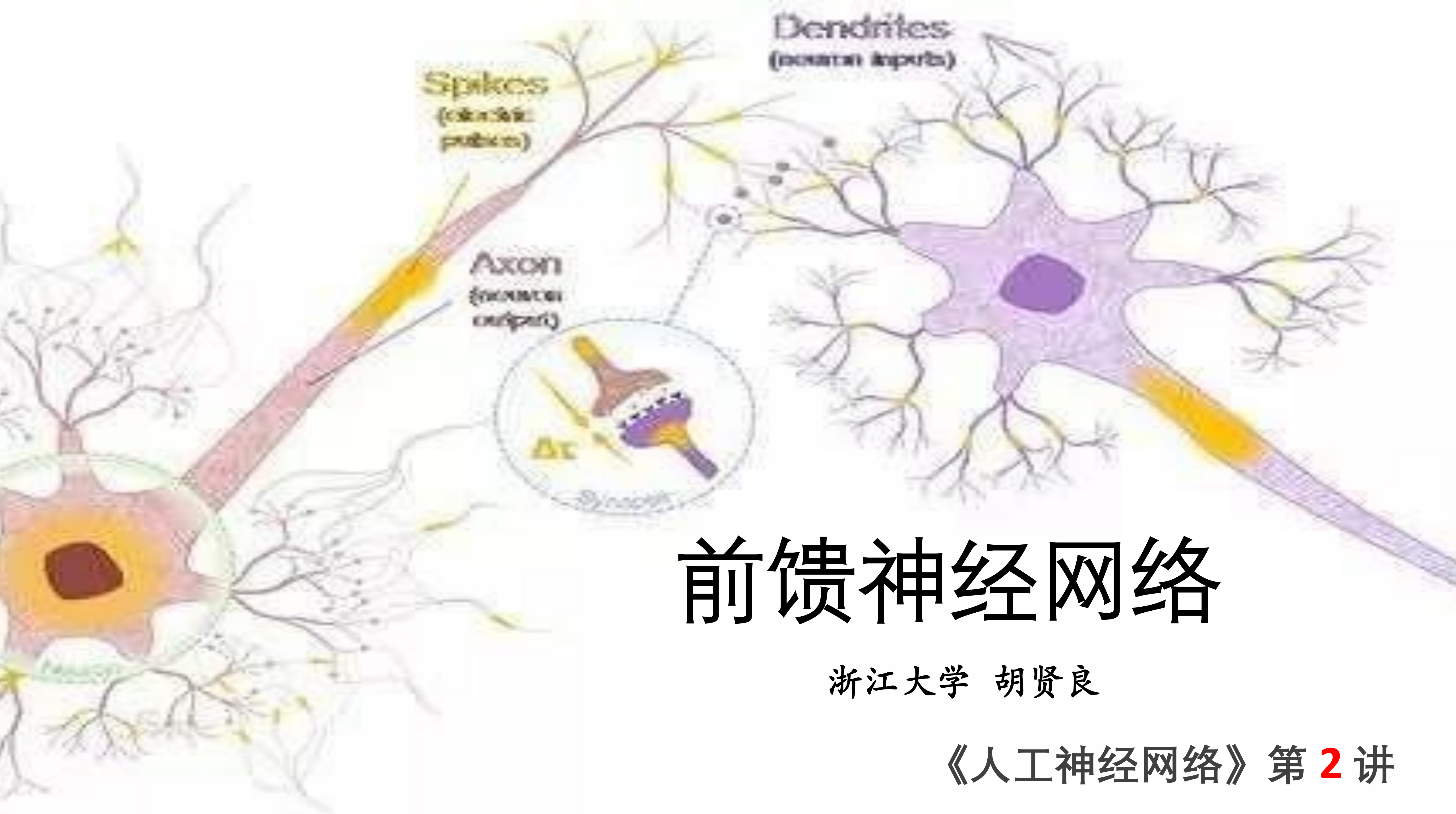


前馈神经网络

浙江大学 胡贤良

《人工神经网络》第 **2** 讲

主要内容

1. 细说回归
2. 前馈神经网络（多层感知机）
3. 计算图（误差反向传播机制）

1. 回归(Regression)

回归问题：解决如预测房价、销售额等输出连续值的问题（ml中与分类问题的区别之处）

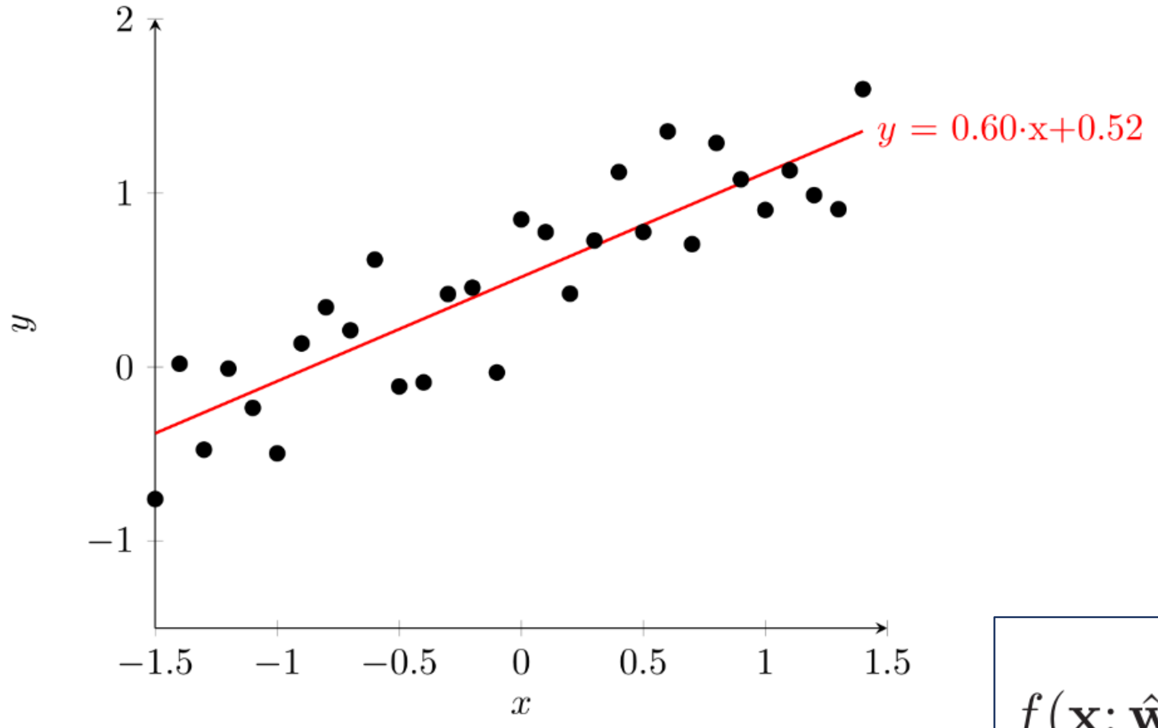
典型：线性回归（参考下一页）、Logistic回归等；高尔顿提出

典型案例：散乱数据点拟合

案例 - 线性回归(Linear Regression)

参看李宏毅的demo!

$$\boxed{f(\mathbf{x}; \mathbf{w}, b) = \mathbf{w}^{\mathrm{T}}\mathbf{x} + b}$$



$$\hat{\mathbf{x}} = \mathbf{x} \oplus 1 \triangleq \begin{bmatrix} \mathbf{x} \\ \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 \\ \vdots \\ x_k \\ 1 \end{bmatrix},$$

$$\hat{\mathbf{w}} = \mathbf{w} \oplus b \triangleq \begin{bmatrix} \mathbf{w} \\ \\ b \end{bmatrix} = \begin{bmatrix} w_1 \\ \vdots \\ w_k \\ b \end{bmatrix},$$

$$\boxed{f(\mathbf{x}; \hat{\mathbf{w}}) = \hat{\mathbf{w}}^{\mathrm{T}}\hat{\mathbf{x}},}$$

训练线性回归模型

- 模型（以截距为0为例）

$$f(\mathbf{x};\mathbf{w}) = \mathbf{w}^{\mathrm{T}}\mathbf{x}$$

- 损失函数

$$\begin{aligned}\mathcal{R}(\mathbf{w}) &= \sum_{n=1}^{N}\mathcal{L}(y^{(n)}, f(\mathbf{x}^{(n)};\mathbf{w}))\\ &= \frac{1}{2}\sum_{n=1}^{N}\left(y^{(n)} - \mathbf{w}^{\mathrm{T}}\mathbf{x}^{(n)}\right)^2\\ &= \frac{1}{2}\|\mathbf{y} - X^{\mathrm{T}}\mathbf{w}\|^2,\end{aligned}$$

- 优化准则:

$$\begin{aligned}\frac{\partial\mathcal{R}(\mathbf{w})}{\partial\mathbf{w}} &= \frac{1}{2}\frac{\partial\,\|\mathbf{y} - X^{\mathrm{T}}\mathbf{w}\|^2}{\partial\mathbf{w}}\\ &= -X(\mathbf{y} - X^{\mathrm{T}}\mathbf{w}),\end{aligned}$$

$$\xrightarrow{\frac{\partial}{\partial\mathbf{w}}\mathcal{R}(\mathbf{w}) = 0}\ \cdots\cdots$$

经验风险最小化（最小二乘法）
结构风险最小化（岭回归）
最大似然估计
最大后验估计

例：一元情形 $\widehat{Y}_i = \widehat{\beta_0} + \widehat{\beta_1}X_i + e_i$

- 代价函数

$$Q = \sum_{i=1}^{n}e_i^2 = \sum_{i=1}^{n}(Y_i - \widehat{Y}_i)^2 = \sum_{i=1}^{n}(Y_i - \widehat{\beta_0} - \widehat{\beta_1}X_i)^2$$

- 极值必要条件（一阶最优性条件）

$$\frac{\partial Q}{\partial\widehat{\beta_0}} = 2\sum_{i=1}^{n}\left(Y_i - \widehat{\beta_0} - \widehat{\beta_1}X_i\right)(-1) = 0$$

$$\frac{\partial Q}{\partial\widehat{\beta_1}} = 2\sum_{i=1}^{n}\left(Y_i - \widehat{\beta_0} - \widehat{\beta_1}X_i\right)(-X_i) = 0$$

- 求得最优解

$$\widehat{\beta_0} = \frac{n\sum X_iY_i - \sum X_i\sum Y_i}{n\sum X_i^2 - (\sum X_i)^2}, \qquad \widehat{\beta_1} = \frac{\sum X_i^2\sum Y_i - \sum X_i\sum X_iY_i}{n\sum X_i^2 - (\sum X_i)^2}$$

代码实现参考： demo_regression.ipynb 前半部分

推广&应用

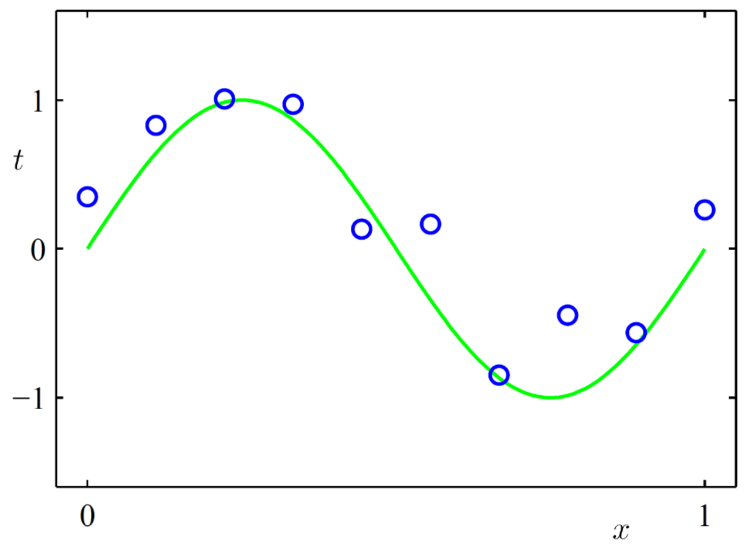
推广:多元线性回归 $p(x)=a_0+a_1x+a_2x^2+......+a_nx^n$

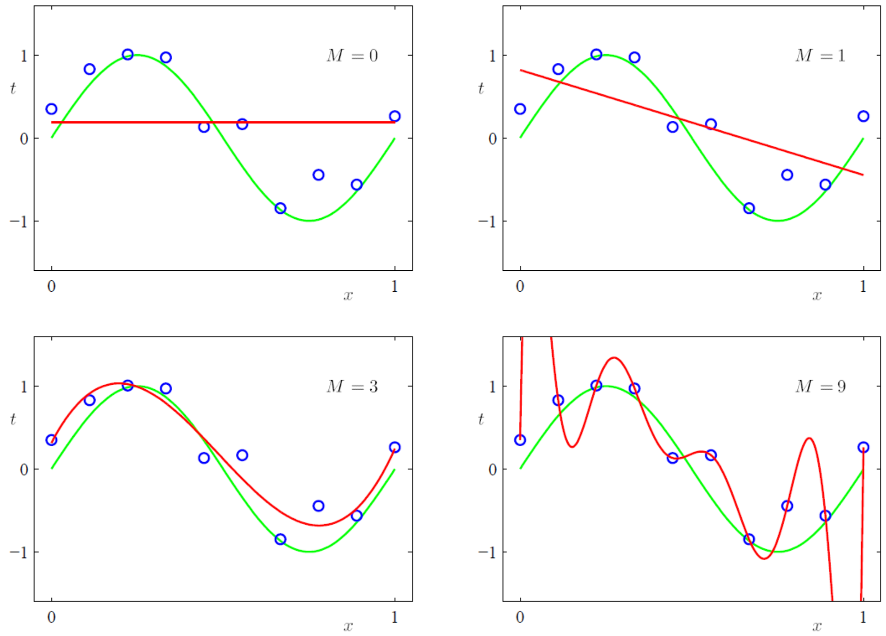
$$\begin{bmatrix} n & \sum_{i=1}^{n} x_i & \cdots & \sum_{i=1}^{n} x_i^k \\ \sum_{i=1}^{n} x_i & \sum_{i=1}^{n} x_i^2 & \cdots & \sum_{i=1}^{n} x_i^{k+1} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^{n} x_i^k & \sum_{i=1}^{n} x_i^{k+1} & \cdots & \sum_{i=1}^{n} x_i^{2k} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_k \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} x_i y_i \\ \vdots \\ \sum_{i=1}^{n} x_i^k y_i \end{bmatrix}.$$

应用：Polynomial Curve Fitting

模型: $y(x,\mathbf{w}) = w_0+w_1x+w_2x^2+\ldots+w_Mx^M$

损失函数: $E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n,\mathbf{w})-t_n\}^2$



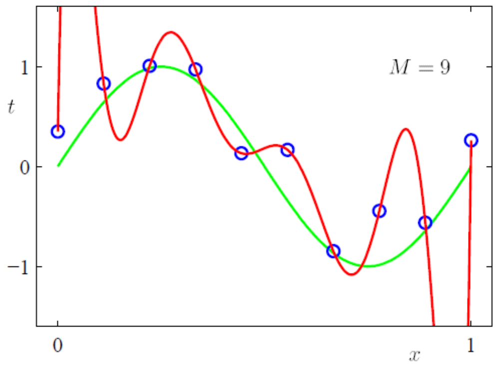


Which Degree of Polynomial?

Controlling Overfitting 1: Regularization

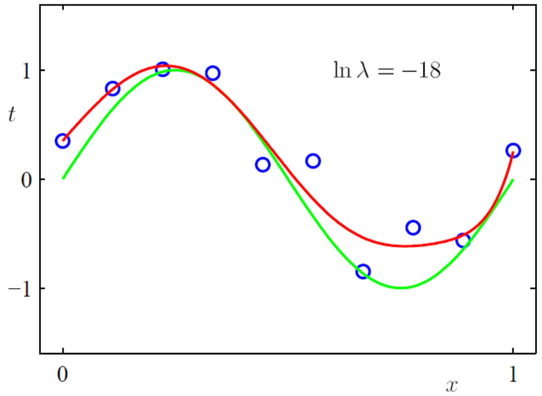
$$\tilde{E}(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(x_n, \boldsymbol{w}) - t_n\}^2 + \frac{\lambda}{2}||\boldsymbol{w}||^2$$

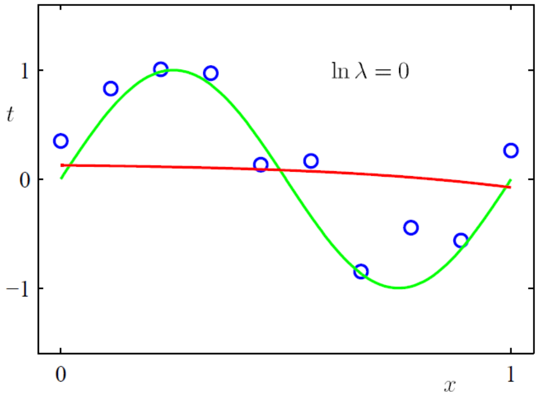
对大的系数进行惩罚



	$M = 0$	$M = 1$	$M = 3$	$M = 9$
$w_0^\star$	0.19	0.82	0.31	0.35
$w_1^\star$		-1.27	7.99	232.37
$w_2^\star$			-25.43	-5321.83
$w_3^\star$			17.37	48568.31
$w_4^\star$				-231639.30
$w_5^\star$				640042.26
$w_6^\star$				-1061800.52
$w_7^\star$				1042400.18
$w_8^\star$				-557682.99
$w_9^\star$				125201.43

As order of polynomial M increases, so do the coefficient magnitudes!





	$\ln \lambda = -\infty$	$\ln \lambda = -18$	$\ln \lambda = 0$
$w_0^\star$	0.35	0.35	0.13
$w_1^\star$	232.37	4.74	-0.05
$w_2^\star$	-5321.83	-0.77	-0.06
$w_3^\star$	48568.31	-31.97	-0.05
$w_4^\star$	-231639.30	-3.89	-0.03
$w_5^\star$	640042.26	55.28	-0.02
$w_6^\star$	-1061800.52	41.32	-0.01
$w_7^\star$	1042400.18	-45.95	-0.00
$w_8^\star$	-557682.99	-91.53	0.00
$w_9^\star$	125201.43	72.68	0.01

线性回归训练 - 2

经验风险最小化（最小二乘法）
结构风险最小化（岭回归）
最大似然估计
最大后验估计

- 损失函数

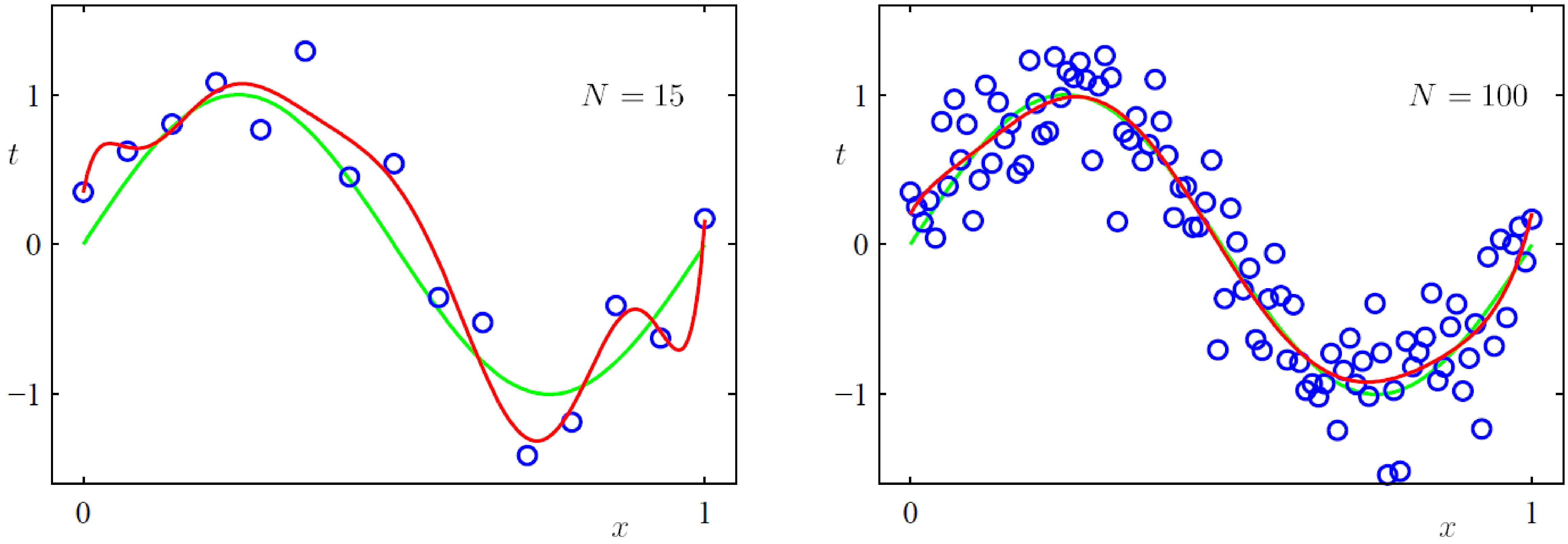
$$\mathcal{R}(\boldsymbol{w}) = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{X}^{\top}\boldsymbol{w}\|^2 + \boxed{\frac{1}{2}\lambda\|\boldsymbol{w}\|^2,}$$

- 优化准则:

$$\frac{\partial}{\partial \mathbf{w}}\mathcal{R}(\mathbf{w}) = 0 \quad \Longrightarrow \quad \boldsymbol{w}^* = (\boldsymbol{X}\boldsymbol{X}^{\top} + \lambda I)^{-1}\boldsymbol{X}\boldsymbol{y},$$

Ridge Regression

Controlling Overfitting 2 : Dataset size

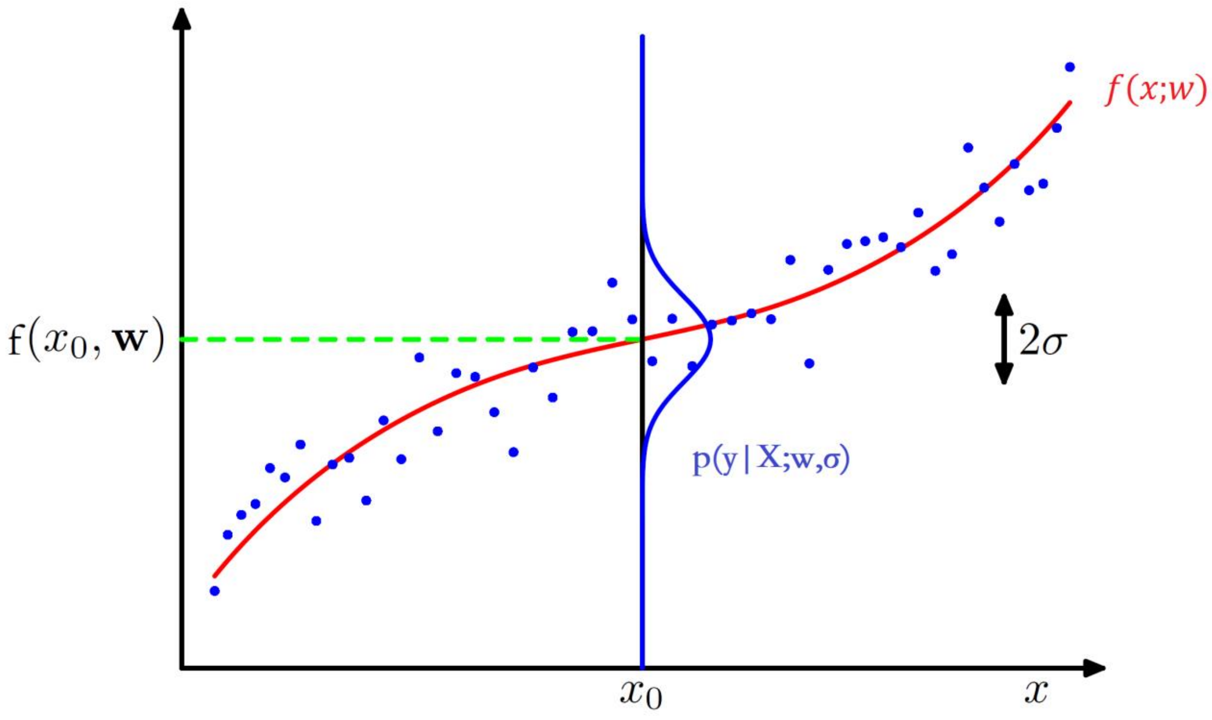


概率角度来看线性回归

设标签y为一个随机变量，服从均值$f(x;w) = w^T x$，

方差σ^2 的高斯分布：

$$p(y|\mathbf{x};\mathbf{w},\sigma) = \mathcal{N}(y;\mathbf{w}^{\mathrm{T}}\mathbf{x},\sigma^2)$$
$$= \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(y-\mathbf{w}^{\mathrm{T}}\mathbf{x})^2}{2\sigma^2}\right).$$



线性回归训练 - 3

经验风险最小化（最小二乘法）
结构风险最小化（岭回归）
最大似然估计
最大后验估计

1. 参数w在训练集D上的似然函数（Likelihood）为

$$p(\mathbf{y}|X;\mathbf{w},\sigma) = \prod_{n=1}^{N} p(y^{(n)}|\mathbf{x}^{(n)};\mathbf{w},\sigma)$$
$$= \prod_{n=1}^{N} \mathcal{N}(y^{(n)};\mathbf{w}^{\mathrm{T}}\mathbf{x}^{(n)},\sigma^2)$$

2. 最大似然估计（Maximum Likelihood Estimate，MLE）

- 找一组参数w，使得似然函数p(y|X;w,σ)最大，即：

$$令 \frac{\partial \log p(\mathbf{y}|X;\mathbf{w},\sigma)}{\partial \mathbf{w}} = 0$$

$$\Downarrow$$

$$\mathbf{w}^{ML} = (XX^{\mathrm{T}})^{-1}X\mathbf{y}.$$

贝叶斯公式：$p(Y|X) = \frac{p(X|Y)p(Y)}{p(X)}$

$$p(w|X) \propto p(X|w)p(w)$$

后验　　似然　　先验

posterior ∝ likelihood × prior

线性回归训练 - 4

经验风险最小化（最小二乘法）
结构风险最小化（岭回归）
最大似然估计
最大后验估计

$$p(\boldsymbol{w}|\boldsymbol{X},\boldsymbol{y};\nu,\sigma)=\frac{p(\boldsymbol{w},\boldsymbol{y}|\boldsymbol{X};\nu,\sigma)}{\sum_{\boldsymbol{w}} p(\boldsymbol{w},\boldsymbol{y}|\boldsymbol{X};\nu,\sigma)}$$
$$\propto \underline{p(\boldsymbol{y}|\boldsymbol{X},\boldsymbol{w};\sigma)}p(\boldsymbol{w};\nu),$$

$$p(\boldsymbol{w};\nu)=\mathcal{N}(\boldsymbol{w};\boldsymbol{0},\nu^2 I)$$

后验
posterior

似然
likelihood

先验
prior

$$\log p(\boldsymbol{w}|\boldsymbol{X},\boldsymbol{y};\nu,\sigma)\propto \log p(\boldsymbol{y}|\boldsymbol{X},\boldsymbol{w};\sigma)+\log p(\boldsymbol{w};\nu)$$
$$\propto -\frac{1}{2\sigma^2}\sum_{n=1}^{N}\underline{\left(y^{(n)}-\boldsymbol{w}^{\top}\boldsymbol{x}^{(n)}\right)}^2-\frac{1}{2\nu^2}\underline{\boldsymbol{w}^{\top}\boldsymbol{w}},$$
$$=-\frac{1}{2\sigma^2}\|\boldsymbol{y}-\boldsymbol{X}^{\top}\boldsymbol{w}\|^2-\frac{1}{2\nu^2}\boldsymbol{w}^{\top}\boldsymbol{w}.$$

等价于正则化系数 $\lambda=\sigma^2/\nu^2$

关于回归的总结

	无先验	引入先验
平方误差	经验风险最小化	结构风险最小化
概率	最大似然估计	最大后验估计

$$\boldsymbol{w}^{ML} = (\boldsymbol{X}\boldsymbol{X}^\top)^{-1}\boldsymbol{X}\boldsymbol{y}$$

$$\boldsymbol{w}^* = (\boldsymbol{X}\boldsymbol{X}^\top + \lambda I)^{-1}\boldsymbol{X}\boldsymbol{y}$$

2.前馈神经网络

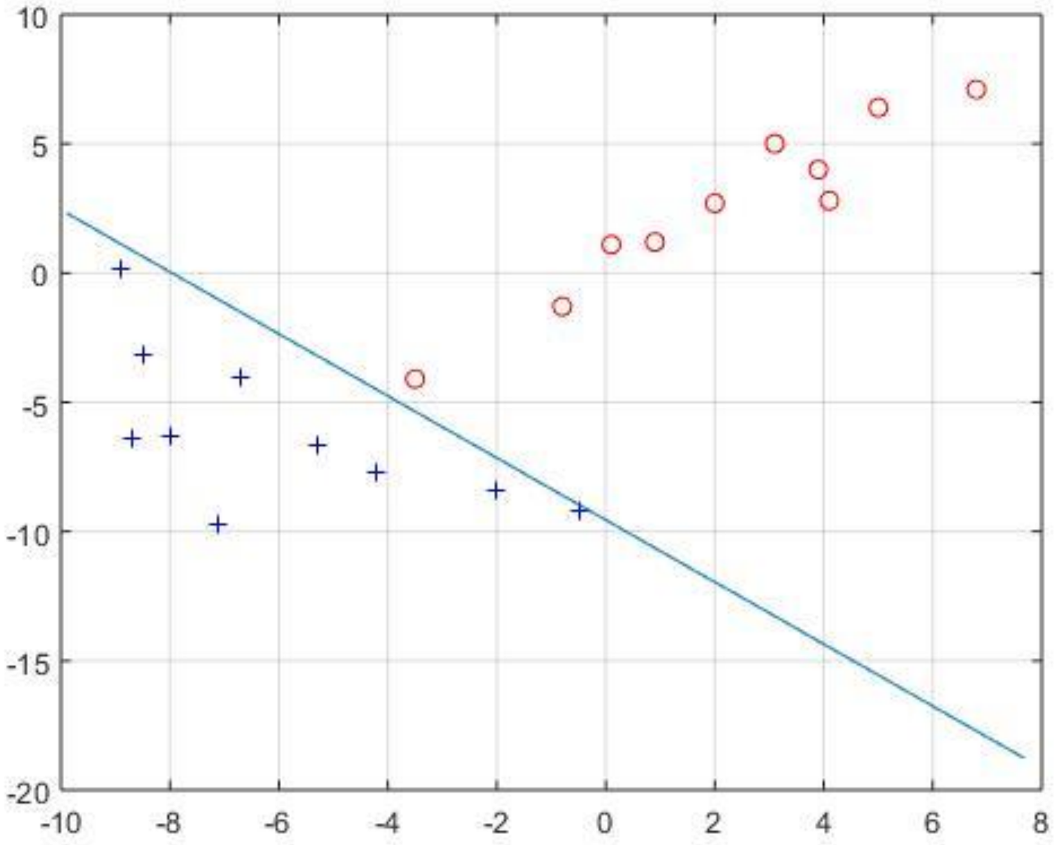
多层感知机

感知机实现样例

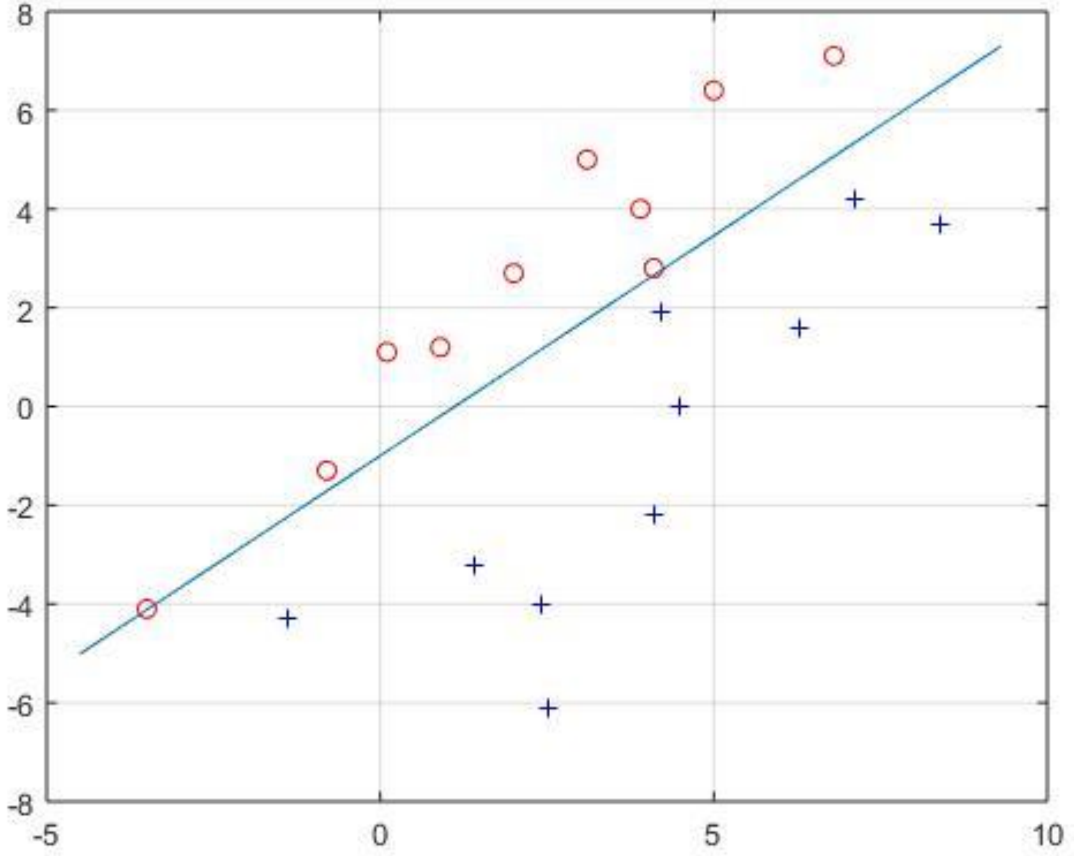
```python
class Perceptron:
 def __init__(self,x,y,a=1):
  self.x = x
    self.y = y
    self.w= np.zeros((x.shape[1],1)) # 初始化权重为0
    self.b = 0
    self. learning_rate = 1    #学习率
    self.numsamples = self.x.shape[0]
    self.numfeatures = self.x.shape[1]
 def sign(self,w,b,x):
  y = np.dot(x,w) + b
    return int(y)
 def update(self,label_i,data_i):
   err = label_i * data_i
     err = tmp.reshape(self.w.shape)
     self.w = self.w + …       # Try It Yourself
     self.b = self.b + ….      #
 def train(self):
    isFind = False
    while not isFind:
      count = 0
      for i in range(self.numsamples):
        tmpY = self.sign(self.w,self.b,self.x[i,:])
        if tmpY*self.y[i] <= 0:    #如果是一个误分类实例点
           print '误分类点:',self.x[i,:],', w=',self.w,', b=',self.b
           count += 1
           self.update(self.y[i], self.x[i,:])
      if count == 0:
        print 'Finally: w = ', self.w, ', b = ', self.b
        isFind = True
   return self.w, self.b
```

```python
import numpy as np
import matplotlib.pyplot as plt
def createdata():  #1、创建数据集
 samples=np.array([[3,-3],[4,-3],[1,1],[1,2]])
 labels=[-1,-1,1,1]
 return samples,labels
class Picture:
 def __init__(self,data, w, b):
  self.b = b
  self.w = w
  plt.figure(1)
  plt.title('Perceptron Learning Algorithm',size=14)
  plt.xlabel('x0-axis',size=14)
  plt.ylabel('x1-axis',size=14)

  xData=np.linspace(0,5,100)
  yData=self.expression(xData)
  plt.plot(xData,yData,color='r',label='sample data')

  plt.scatter(data[0][0],data[0][1],s=50)
  plt.scatter(data[1][0],data[1][1],s=50)
  plt.scatter(data[2][0],data[2][1],s=50,marker='x')
  plt.scatter(data[3][0],data[3][1],s=50,marker='x')
  plt.savefig('2d.png',dpi=75)

 def expression(self,x):
  y=(-self.b-self.w[0]*x)/self.w[1
  return y

 def show(self):
  plt.show()

if __name__ == '__main__':
 samples,labels = createdata()
 myperceptron   = Perceptron(x = samples, y = labels)
 weights,bias   = Perceptron.train()
 Picture        = Picture(samples, weights, bias)
 Picture.show()
```

感知器 – 成功算例

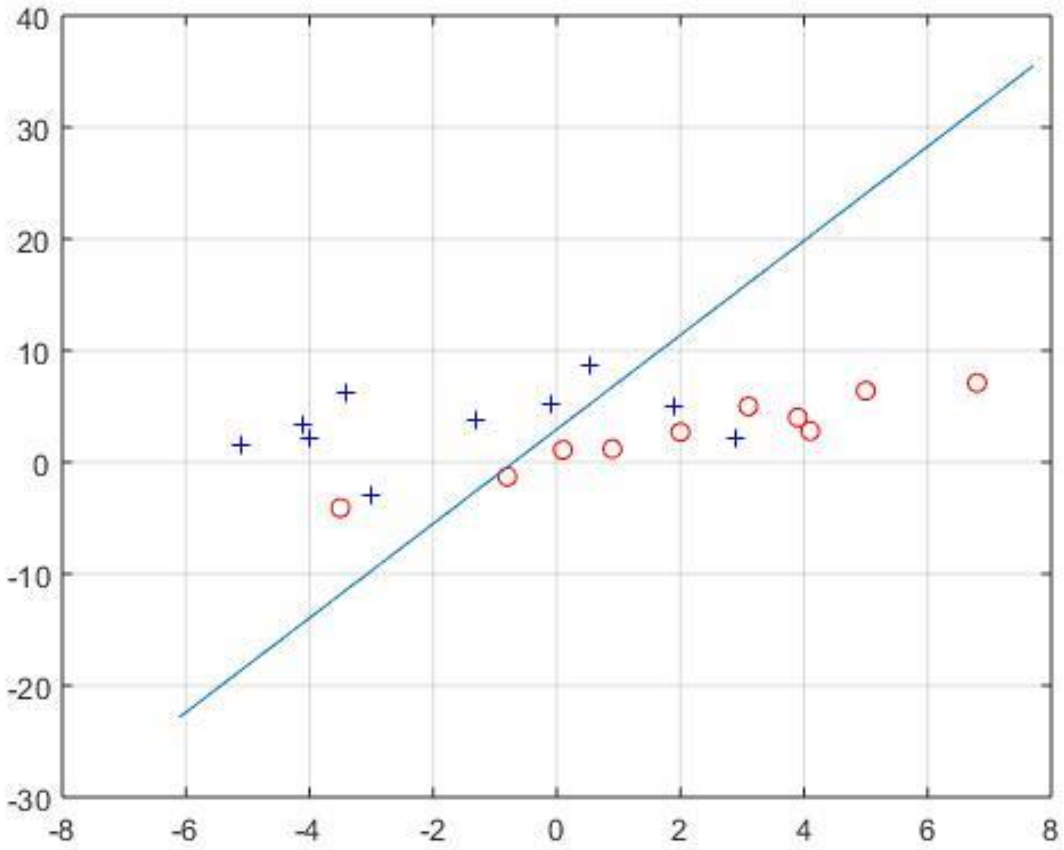


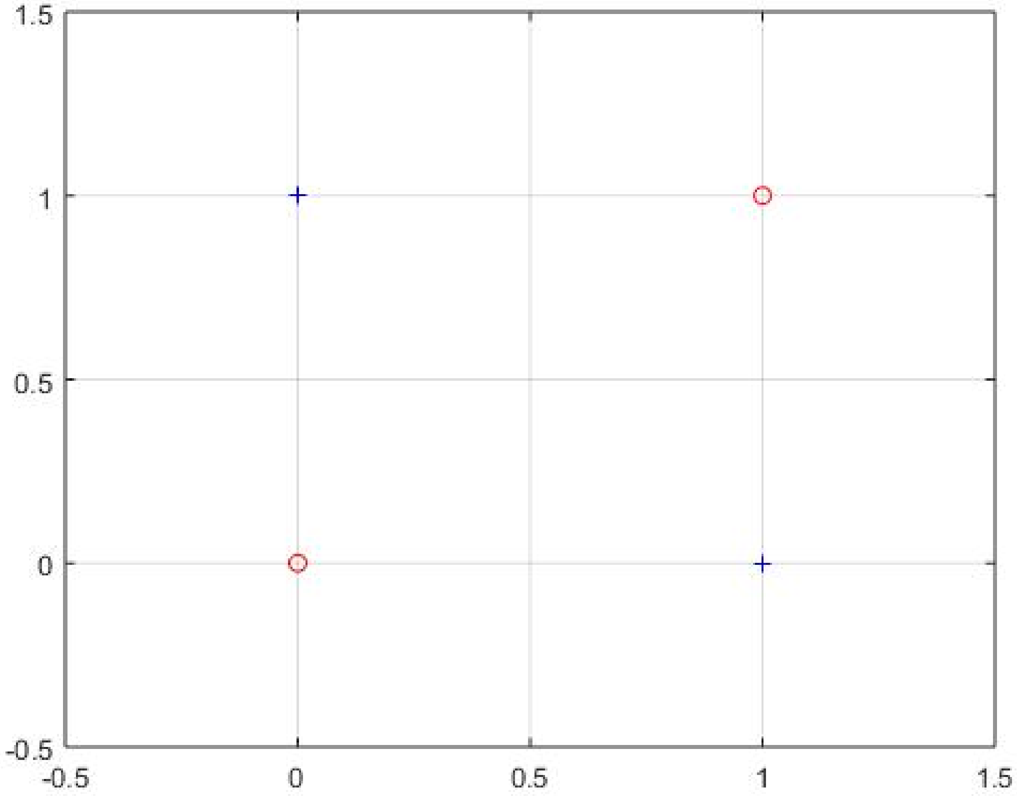
>> Perceptron(xo, xx);
感知器算法收敛时解矢量w为： 43 5.4 4.5
感知器算法收敛步数kt为： 35



>> Perceptron(xo, xx);
感知器算法收敛时解矢量w为： 34 -30.4 34.1
感知器算法收敛步数kt为： 24

感知器 – 失败算例





>> Perceptron(xo, xx);
目标函数在规定的最大迭代次数内无法收敛
感知器算法的解矢量w为： 20　　28.72　　-6.8

>> xo = [0 0;1 1];
>> xx = [1 0;0 1];
>> Perceptron(x0, xx);　**# 注意修改输入数据格式**
目标函数在规定的最大迭代次数内无法收敛
感知器算法的解矢量w为： 0 0 0

收敛性 – 线性可分性

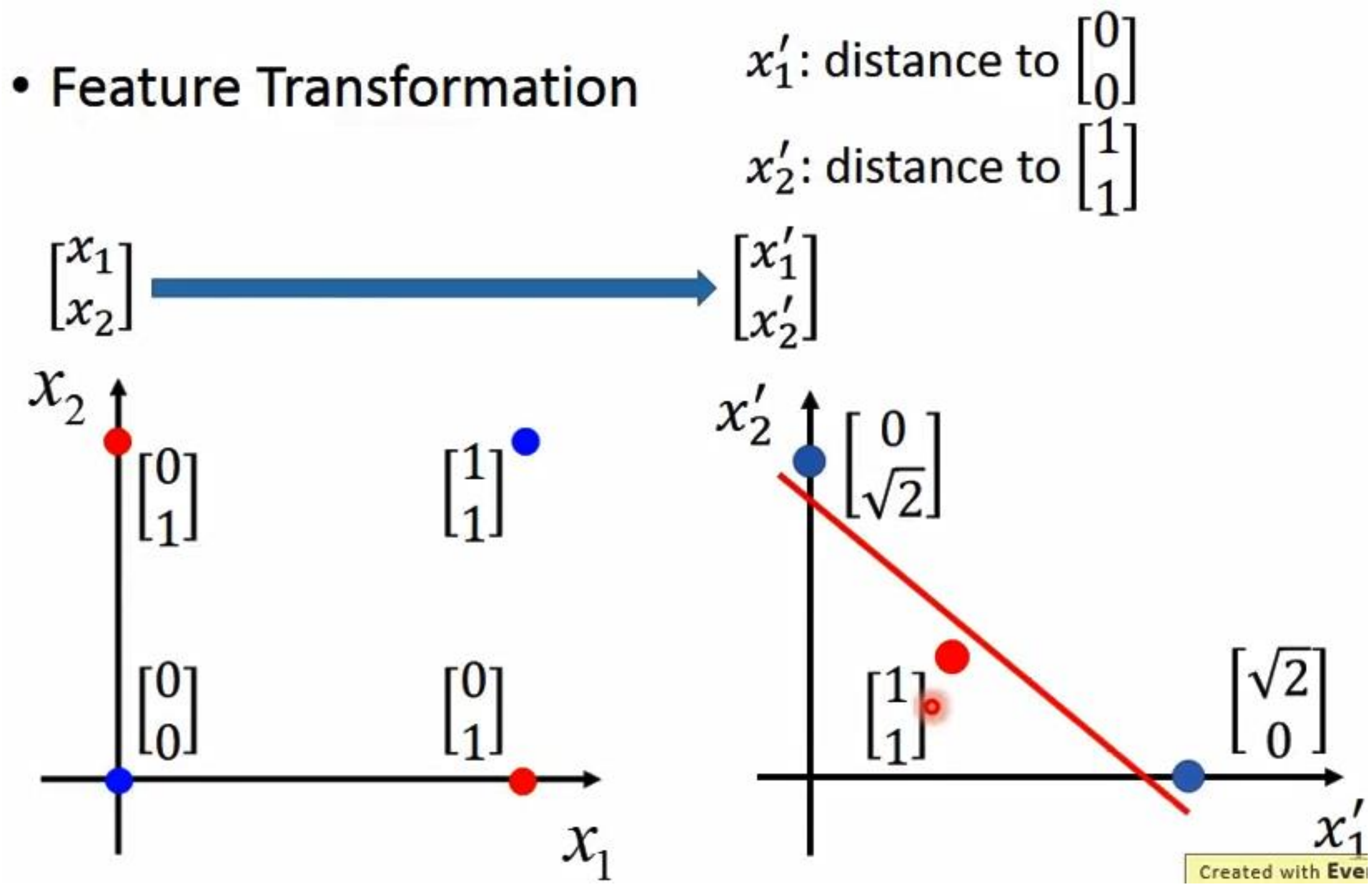
定义 3.1 – 两类线性可分：对于训练集 $\mathcal{D} = \{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^{N}$，如果存在权重向量 $\mathbf{w}^*$，对所有样本都满足 $yf(\mathbf{x}; \mathbf{w}^*) > 0$，那么训练集 $\mathcal{D}$ 是线性可分的。

定理 3.1 – 感知器收敛性：给定一个训练集 $\mathcal{D} = \{(\mathbf{x}^{(n)}, y^{(n)})\}_{n=1}^{N}$，假设 R 是训练集中最大的特征向量的模，

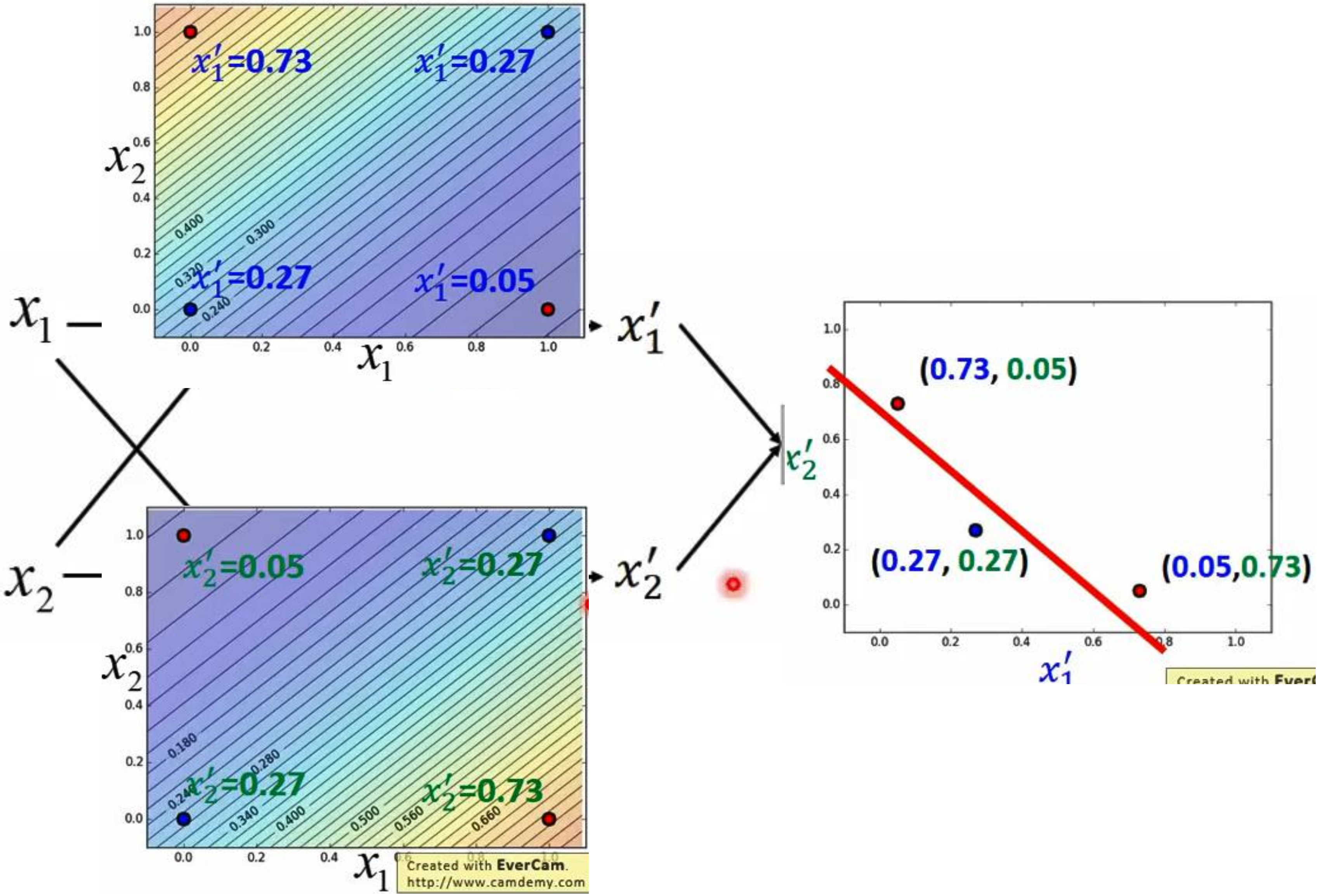
$$R = \max_{n} \|x^{(n)}\|.$$

如果训练集 $\mathcal{D}$ 线性可分，感知器学习算法3.1的权重更新次数不超过 $\frac{R^2}{\gamma^2}$。

再谈XOR – 参考李宏毅讲稿

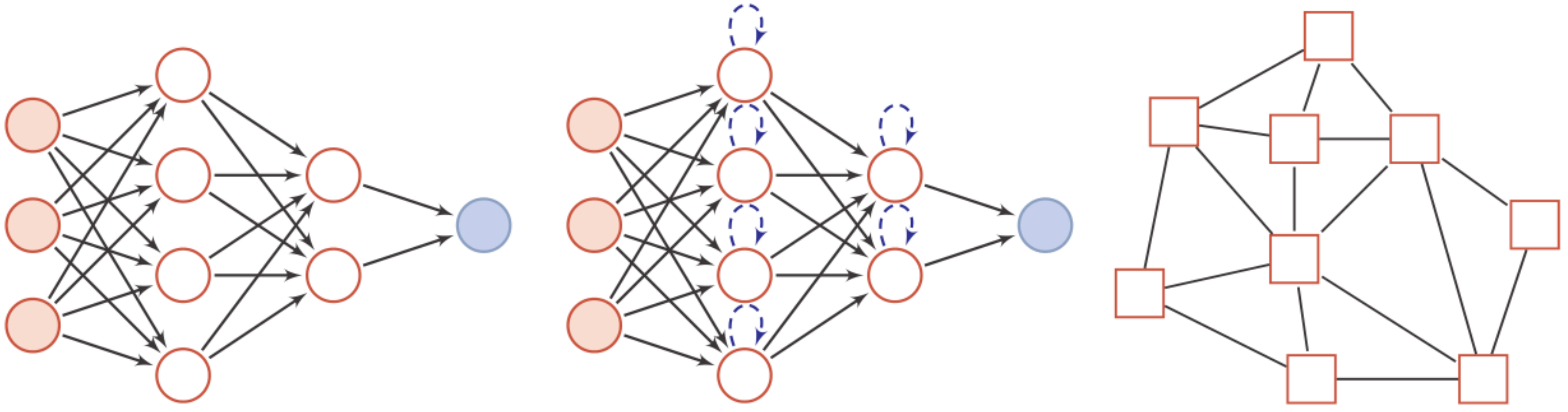


特征变换：堆叠的线性变换！



回顾：人工神经网络

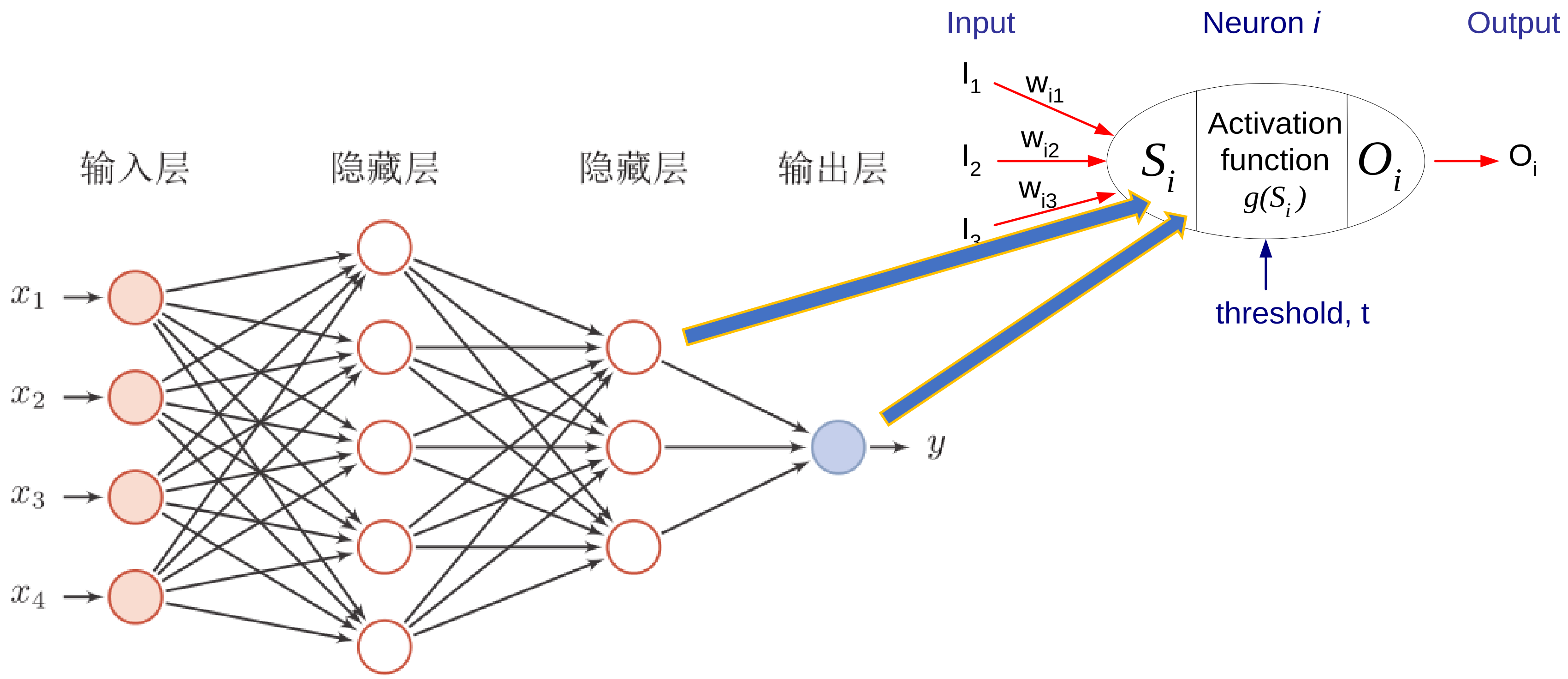
由大量的神经元以及它们之间的有向连接构成



(a) 前馈网络　(b) 记忆网络　(c) 图网络

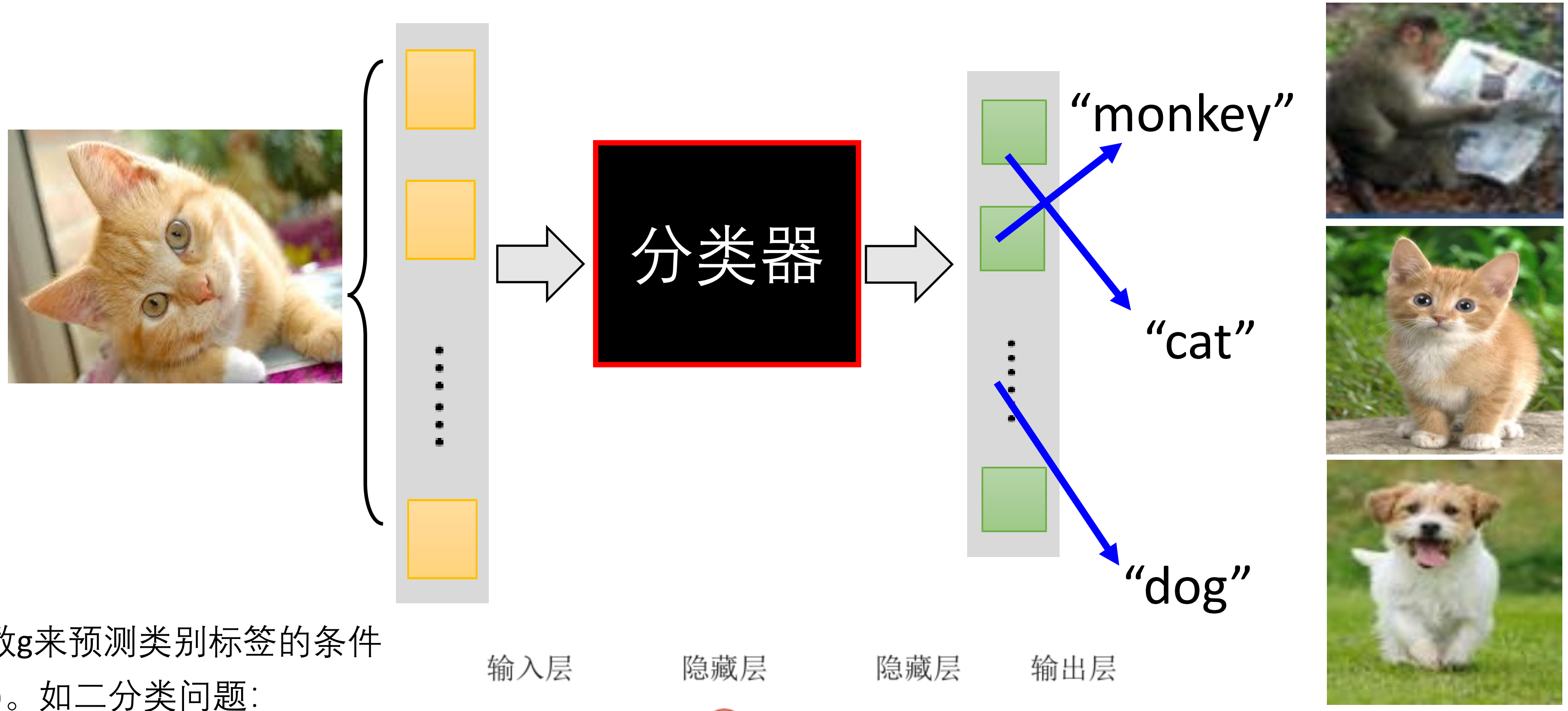
1. 神经元的激活规则：神经元输入到输出之间的映射关系
2. 网络的拓扑结构：不同神经元之间的连接关系
3. 学习算法：通过训练数据来学习神经网络的参数

前馈神经网络(全连接网络、多层感知器)



- 各神经元分属于不同的层，层内无连接，相邻层间神经元两两连接
- 整个网络中无反馈，信号从输入层向输出层单向传播(有向无环图)
- 训练神经网络意味着学习神经元的权值

应用：分类模型

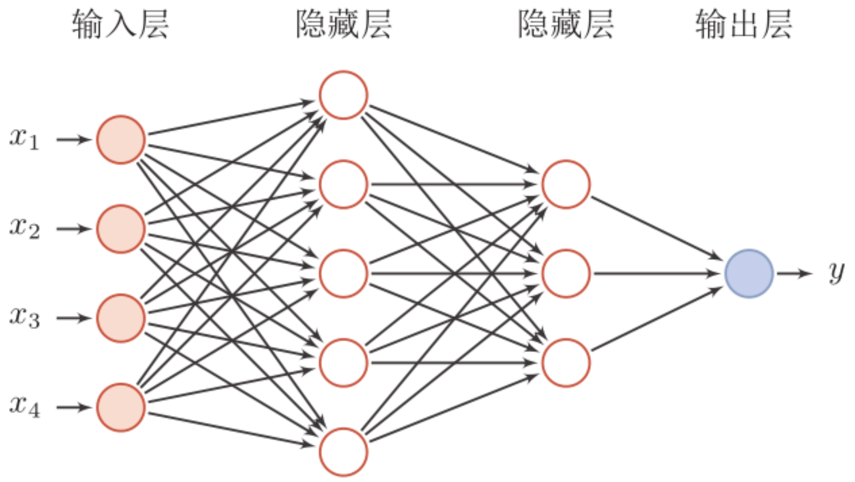


构造非线性函数g来预测类别标签的条件概率$p(y=c|\mathbf{x})$。如二分类问题：

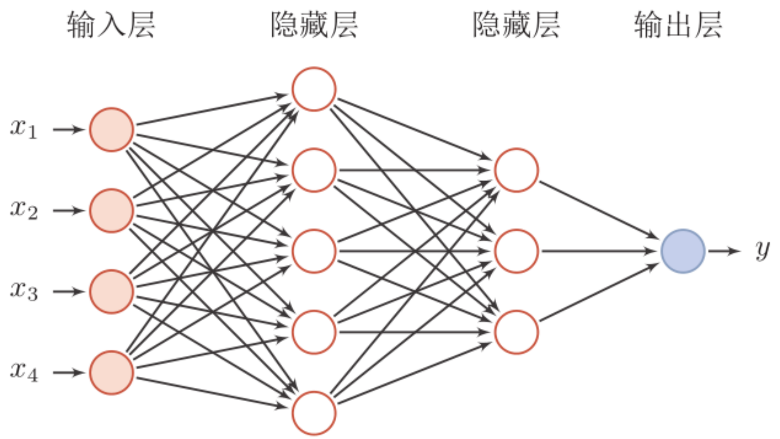
$$P(y=1|\boldsymbol{x}) = g(f(\boldsymbol{x};\boldsymbol{w}))$$

其中，

- 函数f：线性函数
- 函数g：把线性函数的值域从实数区间“挤压”到(0,1)表示概率



计算公式



记号	含义
L	神经网络的层数
M_l	第 l 层神经元的个数
$f_l(\cdot)$	第 l 层神经元的激活函数
$\boldsymbol{W}^{(l)} \in \mathbb{R}^{M_l \times M_{l-1}}$	第 $l-1$ 层到第 l 层的权重矩阵
$\boldsymbol{b}^{(l)} \in \mathbb{R}^{M_l}$	第 $l-1$ 层到第 l 层的偏置
$\boldsymbol{z}^{(l)} \in \mathbb{R}^{M_l}$	第 l 层神经元的净输入（净活性值）
$\boldsymbol{a}^{(l)} \in \mathbb{R}^{M_l}$	第 l 层神经元的输出（活性值）

- 连续并可导（允许少数点上不可导）的非线性函数。
  - 可导的激活函数可以直接利用数值优化的方法来学习网络参数。
- 激活函数及其导函数要尽可能的简单
  - 有利于提高网络计算效率。
- 激活函数的导函数的值域要在一个合适的区间内
  - 不能太大也不能太小，否则会影响训练的效率和稳定性。
- 单调递增

- 前馈神经网络公式表示

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)} \boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)},$$
$$\boldsymbol{a}^{(l)} = f_l(\boldsymbol{z}^{(l)}).$$

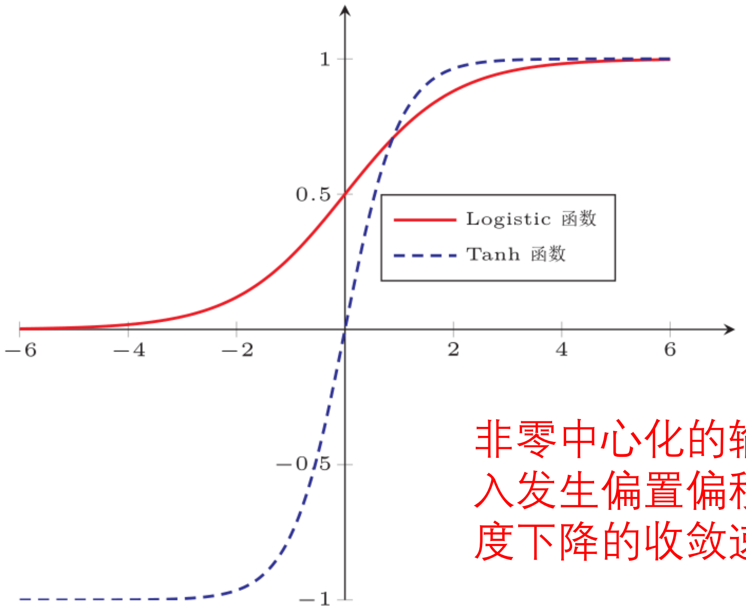
- 前馈计算：

$$\boldsymbol{x} = \boldsymbol{a}^{(0)} \rightarrow \boldsymbol{z}^{(1)} \rightarrow \boldsymbol{a}^{(1)} \rightarrow \boldsymbol{z}^{(2)} \rightarrow \cdots \rightarrow \boldsymbol{a}^{(L-1)} \rightarrow \boldsymbol{z}^{(L)} \rightarrow \boldsymbol{a}^{(L)} = \phi(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b})$$

经典激活函数

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

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

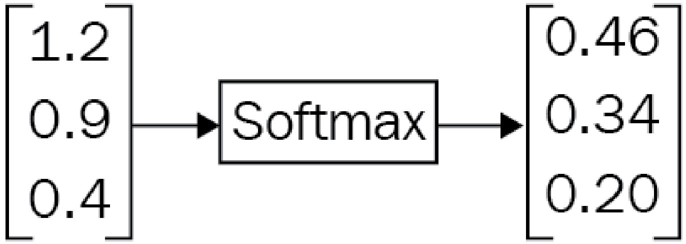


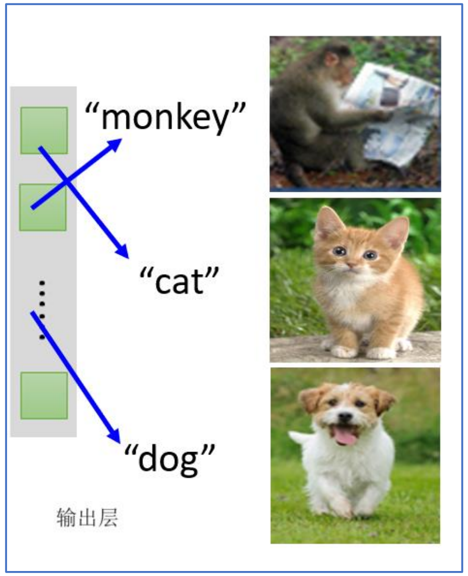
非零中心化的输出会使得其后一层的神经元的输入发生偏置偏移（bias shift），并进一步使得梯度下降的收敛速度变慢。

- 性质:
  - 饱和函数
  - Tanh函数是零中心化的，而logistic函数的输出恒大于0

- 用于多分类的Softmax函数

$$\mathrm{softmax}(x_k) = \frac{\exp(x_k)}{\sum_{i=1}^{K}\exp(x_i)}$$





共性：将数值“挤压”到(0,1)上，便于分类！

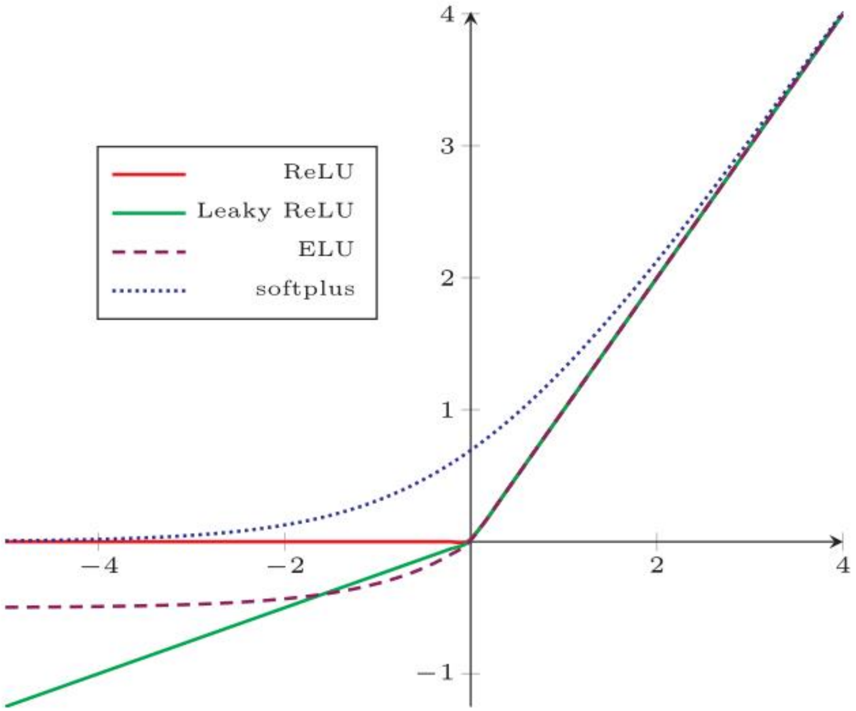
其他常用激活函数

$$\mathrm{ReLU}(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$
$$= \max(0, x).$$

$$\mathrm{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \gamma x & \text{if } x \leq 0 \end{cases}$$
$$= \max(0, x) + \gamma \min(0, x)$$

$$\mathrm{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \gamma(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$
$$= \max(0, x) + \min(0, \gamma(\exp(x) - 1))$$

$$\mathrm{softplus}(x) = \log(1 + \exp(x))$$



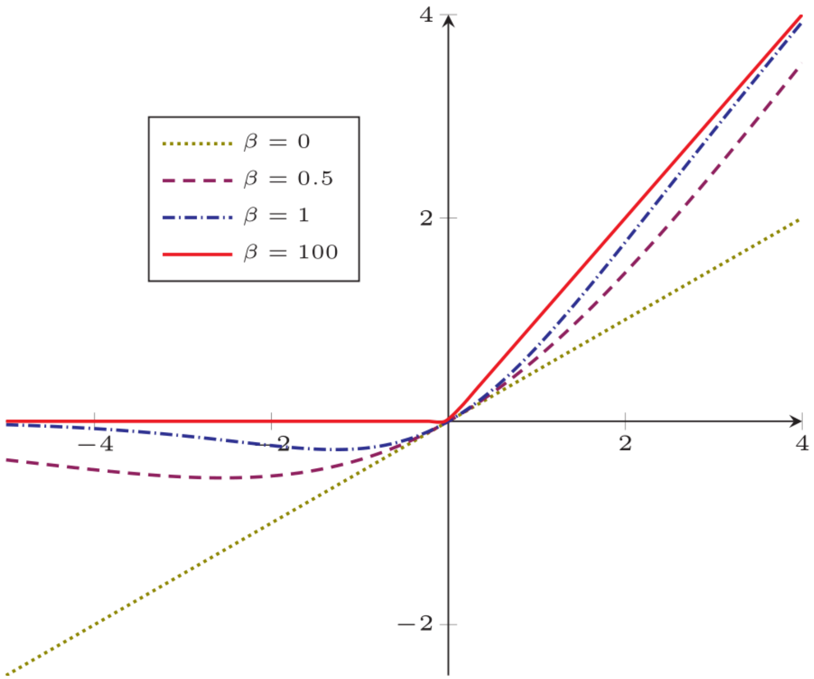
- 计算上更加高效
- 生物学合理性
  - 单侧抑制、宽兴奋边界
- 在一定程度上缓解梯度消失问题

死亡ReLU问题（Dying ReLU Problem）

更复杂的激活函数示例

1. Swish函数

$$\text{swish}(x) = x\sigma(\beta x)$$



- 2. 高斯误差线性单元（Gaussian Error Linear Unit，GELU）

$$\text{GELU}(x) = xP(X \leq x)$$

  - 其中P(X ≤ x)是高斯分布N(μ,σ 2)的累积分布函数，其中μ,σ为超参数，一般设μ = 0,σ = 1即可
  - 由于高斯分布的累积分布函数为S型函数，因此GELU可以用Tanh函数或Logistic函数来近似

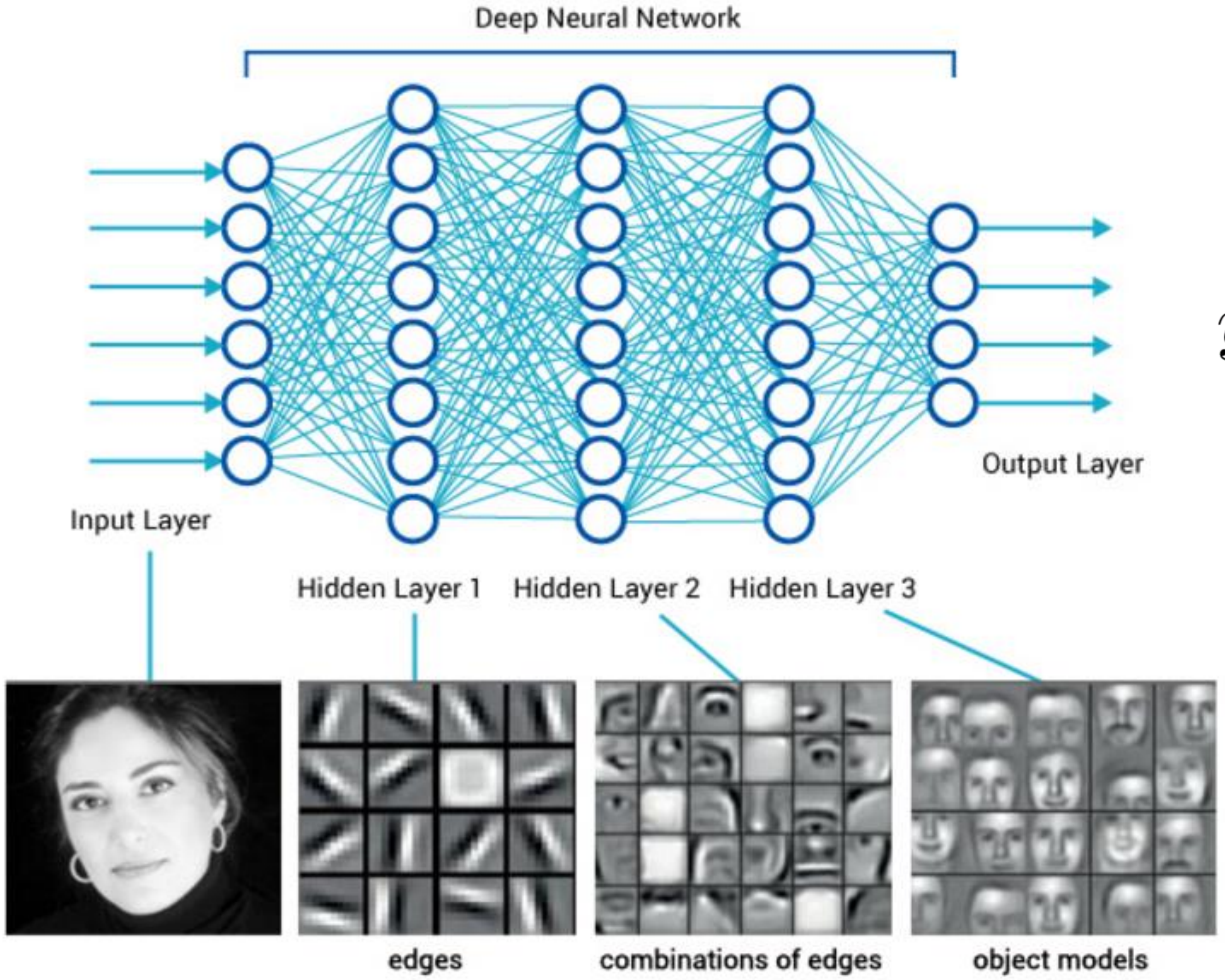
$$\text{GELU}(x) \approx 0.5x\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)\right)\right)$$

或 $$\text{GELU}(x) \approx x\sigma(1.702x).$$

激活函数的导数

激活函数	函数	导数
Logistic 函数	$f(x) = \frac{1}{1+\exp(-x)}$	$f'(x) = f(x)\big(1 - f(x)\big)$
Tanh 函数	$f(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$	$f'(x) = 1 - f(x)^2$
ReLU 函数	$f(x) = \max(0, x)$	$f'(x) = I(x > 0)$
ELU 函数	$f(x) = \max(0, x) + \min\big(0, \gamma(\exp(x) - 1)\big)$	$f'(x) = I(x > 0) + I(x \leq 0) \cdot \gamma \exp(x)$
SoftPlus 函数	$f(x) = \log\big(1 + \exp(x)\big)$	$f'(x) = \frac{1}{1+\exp(-x)}$

深层前馈网络



神经网络

$$\hat{y} = g(\varphi(\mathbf{x}), \theta)$$

分类器

3. 计算图：误差反向传播

谈谈神经网络的训练

参考demo_02_02-ann_in_numpy.ipynb

如何学习/训练?

- 回忆分类模型，目的是极小化损失函数

$$\mathcal{R}(W,\mathbf{b}) = \frac{1}{N}\sum_{n=1}^{N}\mathcal{L}(\boxed{\mathbf{y}^{(n)}},\hat{\mathbf{y}}^{(n)}) + \frac{1}{2}\lambda\|W\|_F^2$$

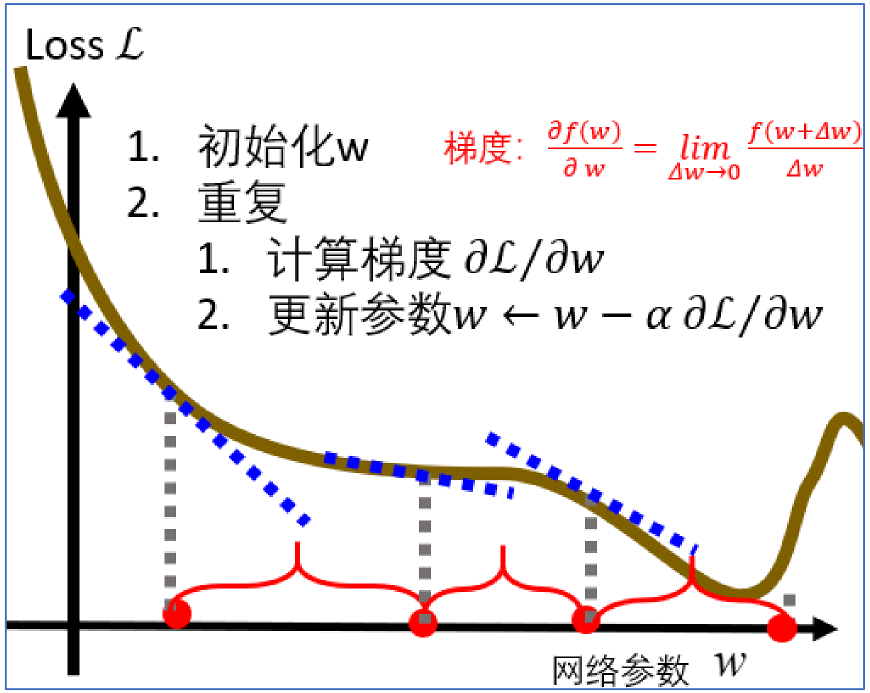
- 运用梯度下降法，我们需要计算

$$W^{(l)} \leftarrow W^{(l)} - \alpha\frac{\partial\mathcal{R}(W,\mathbf{b})}{\partial W^{(l)}} \qquad \mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \alpha\frac{\partial\mathcal{R}(W,\mathbf{b})}{\partial\mathbf{b}^{(l)}}$$

- 神经网络为一个复杂的复合函数
  - 链式法则

$$y = f^5(f^4(f^3(f^2(f^1(x))))) \rightarrow \frac{\partial y}{\partial x} = \frac{\partial f^1}{\partial x}\frac{\partial f^2}{\partial f^1}\frac{\partial f^3}{\partial f^2}\frac{\partial f^4}{\partial f^3}\frac{\partial f^5}{\partial f^4}$$

- 反向传播算法
  - 根据前馈网络的特点而设计的高效方法



回忆：链式法则

链式法则（Chain Rule）是在微积分中求复合函数导数的一种常用方法。

（1）若 $x \in \mathbb{R}, \boldsymbol{u} = u(x) \in \mathbb{R}^s, \boldsymbol{g} = g(\boldsymbol{u}) \in \mathbb{R}^t$，则

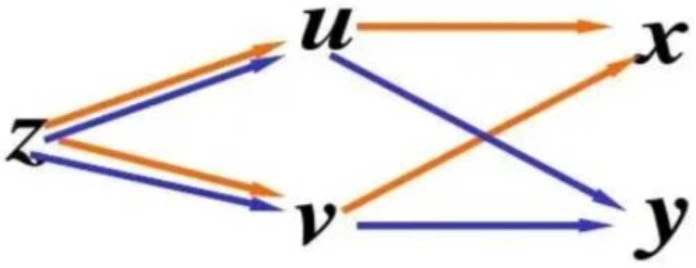
$$\frac{\partial \boldsymbol{g}}{\partial x} = \frac{\partial \boldsymbol{u}}{\partial x}\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{u}} \quad \in \mathbb{R}^{1\times t}.$$

（2）若 $\boldsymbol{x} \in \mathbb{R}^p, \boldsymbol{y} = g(\boldsymbol{x}) \in \mathbb{R}^s, \boldsymbol{z} = f(\boldsymbol{y}) \in \mathbb{R}^t$，则

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}} \quad \in \mathbb{R}^{p\times t}.$$

（3）若 $X \in \mathbb{R}^{p\times q}$ 为矩阵，$\boldsymbol{y} = g(X) \in \mathbb{R}^s, z = f(\boldsymbol{y}) \in \mathbb{R}$，则

$$\frac{\partial z}{\partial X_{ij}} = \frac{\partial \boldsymbol{y}}{\partial X_{ij}}\frac{\partial z}{\partial \boldsymbol{y}} \quad \in \mathbb{R}.$$



$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial u}\cdot\frac{\partial u}{\partial x} + \frac{\partial z}{\partial v}\cdot\frac{\partial v}{\partial x},$$

$$\frac{\partial z}{\partial y} = \frac{\partial z}{\partial u}\cdot\frac{\partial u}{\partial y} + \frac{\partial z}{\partial v}\cdot\frac{\partial v}{\partial y}.$$

反向传播的概念

$$\boldsymbol{x} = \boldsymbol{a}^{(0)} \rightarrow \boldsymbol{z}^{(1)} \rightarrow \boldsymbol{a}^{(1)} \rightarrow \boldsymbol{z}^{(2)} \rightarrow \cdots \rightarrow \boldsymbol{a}^{(L-1)} \rightarrow \boldsymbol{z}^{(L)} \rightarrow \boldsymbol{a}^{(L)} = \phi(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b}))$$

$$\boldsymbol{z}^{(l)} = \boldsymbol{W}^{(l)} \boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)},$$

$$\boldsymbol{a}^{(l)} = f_l(\boldsymbol{z}^{(l)}).$$

在计算出上面三个偏导数之后，公式 (4.49) 可以写为

$$\frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial w_{ij}^{(l)}} = \mathbb{I}_i(a_j^{(l-1)})\delta^{(l)} = \delta_i^{(l)} a_j^{(l-1)}.$$

进一步，$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$ 关于第 l 层权重 $W^{(l)}$ 的梯度为

$$\frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial W^{(l)}} = \delta^{(l)} (\mathbf{a}^{(l-1)})^{\mathrm{T}}.$$

同理，$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$ 关于第 l 层偏置 $\mathbf{b}^{(l)}$ 的梯度为

$$\frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{b}^{(l)}} = \delta^{(l)}.$$

反向传播计算

$$\boxed{\mathbf{z}^{(l)} = W^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}}$$

$$\frac{\partial \mathbf{z}^{(l)}}{\partial w_{ij}^{(l)}} = \left[\frac{\partial z_1^{(l)}}{\partial w_{ij}^{(l)}}, \cdots, \boxed{\frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}}}, \cdots, \frac{\partial z_{m^{(l)}}^{(l)}}{\partial w_{ij}^{(l)}}\right]$$
$$= \left[0, \cdots, \boxed{\frac{\partial(\mathbf{w}_{i:}^{(l)}\mathbf{a}^{(l-1)} + b_i^{(l)})}{\partial w_{ij}^{(l)}}}, \cdots, 0\right]$$
$$= \left[0, \cdots, a_j^{(l-1)}, \cdots, 0\right]$$
$$\triangleq \mathbb{I}_i(a_j^{(l-1)}) \quad \in \mathbb{R}^{m^{(l)}},$$

$$\frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial w_{ij}^{(l)}} = \frac{\partial \mathbf{z}^{(l)}}{\partial w_{ij}^{(l)}} \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{z}^{(l)}}$$

$$\frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{b}^{(l)}} = \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}} \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{z}^{(l)}}$$

$$\delta^{(l)} = \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{z}^{(l)}} \quad \in \mathbb{R}^{m^{(l)}}$$

误差项简记

$$\frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}} = \mathbf{I}_{m^{(l)}} \quad \in \mathbb{R}^{m^{(l)} \times m^{(l)}}$$

计算细节

$$\delta^{(l)} = \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{z}^{(l)}} \quad \in \mathbb{R}^{m^{(l)}}$$

$$\boxed{\mathbf{a}^{(l)} = f_l(\mathbf{z}^{(l)})}$$

$$\boxed{\mathbf{z}^{(l+1)} = W^{(l+1)}\mathbf{a}^{(l)} + \mathbf{b}^{(l+1)}}$$

$$\frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} = \frac{\partial f_l(\mathbf{z}^{(l)})}{\partial \mathbf{z}^{(l)}} = \mathrm{diag}(f_l'(\mathbf{z}^{(l)}))$$

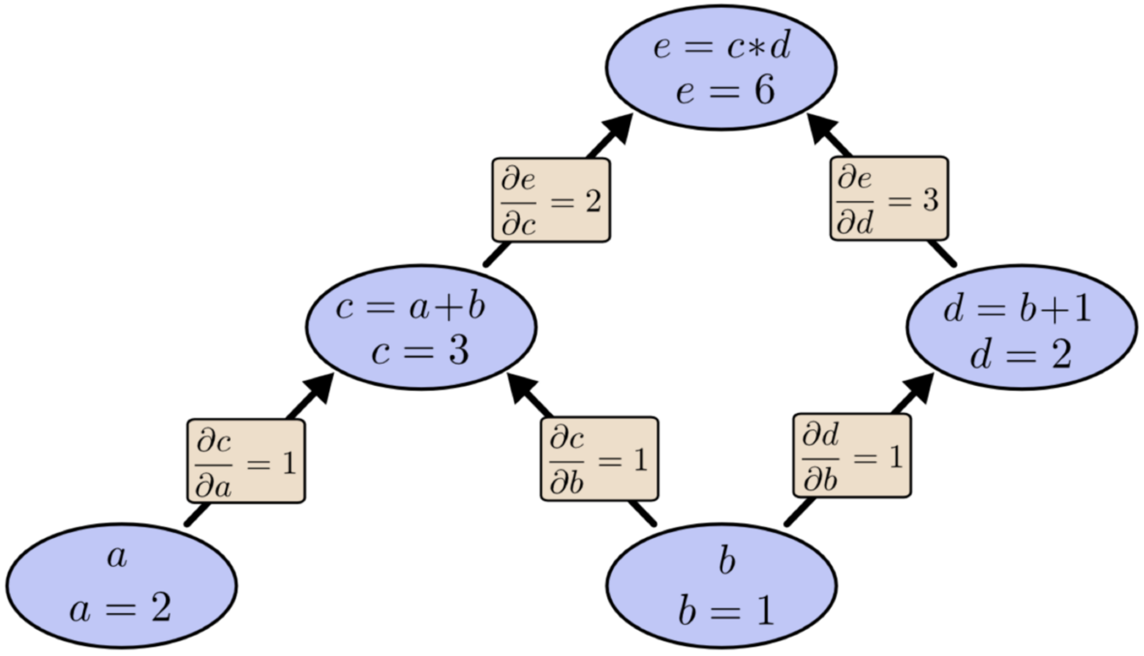
$$\frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} = (W^{(l+1)})^{\mathrm{T}}$$

$$\begin{aligned}
\delta^{(l)} &\triangleq \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{z}^{(l)}} \\
&= \frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} \cdot \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} \cdot \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{z}^{(l+1)}} \\
&= \mathrm{diag}(f_l'(\mathbf{z}^{(l)})) \cdot (W^{(l+1)})^{\mathrm{T}} \cdot \delta^{(l+1)} \\
&= f_l'(\mathbf{z}^{(l)}) \odot \left((W^{(l+1)})^{\mathrm{T}} \delta^{(l+1)}\right),
\end{aligned}$$

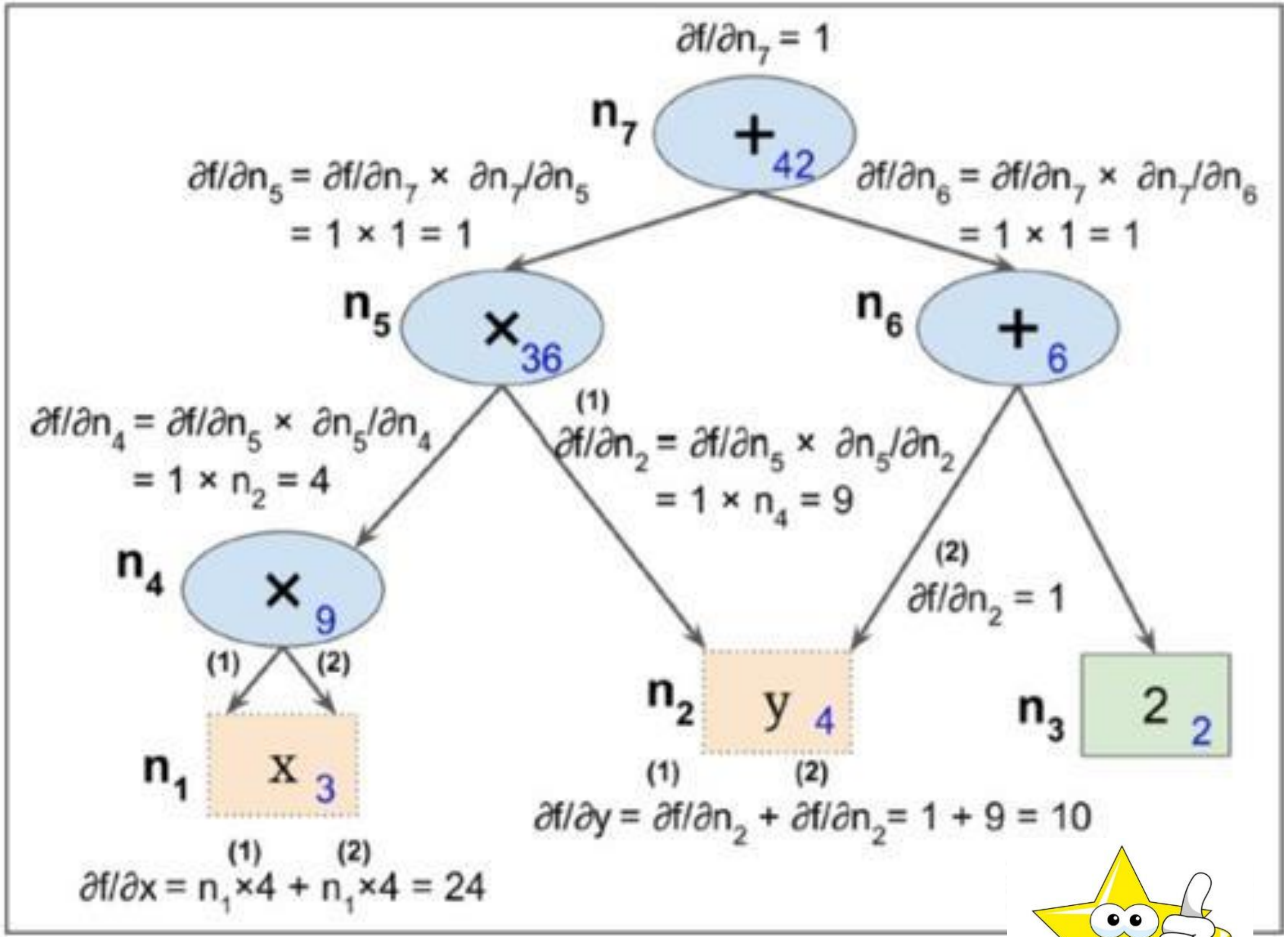
计算图 - 前向传播与反向传播

基本思想：每一步迭代包括

- 前向传播：使用前一次迭代所得到的权值计算网络中每一个神经元的输出，即先计算第k层神经元的输出，再计算第k+1层的输出
- 后向传播：先更新k+1层的梯度/权值，再更新第k层的梯度/权值

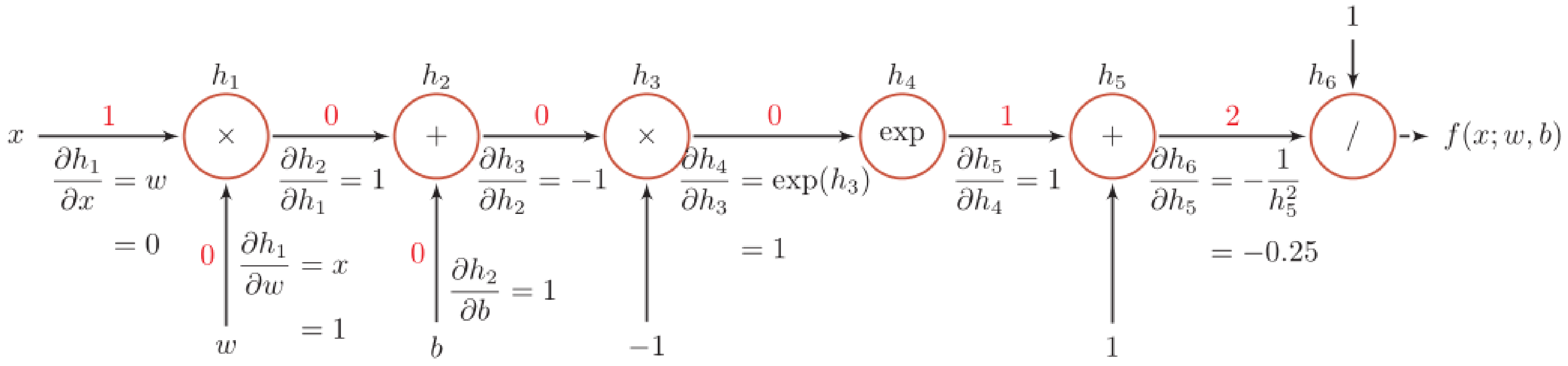


实例1: $\boldsymbol{f(x,y)=x^2y+(y+2)}$



实例2

$$f(x;w,b) = \frac{1}{\exp(-(wx+b))+1}.$$



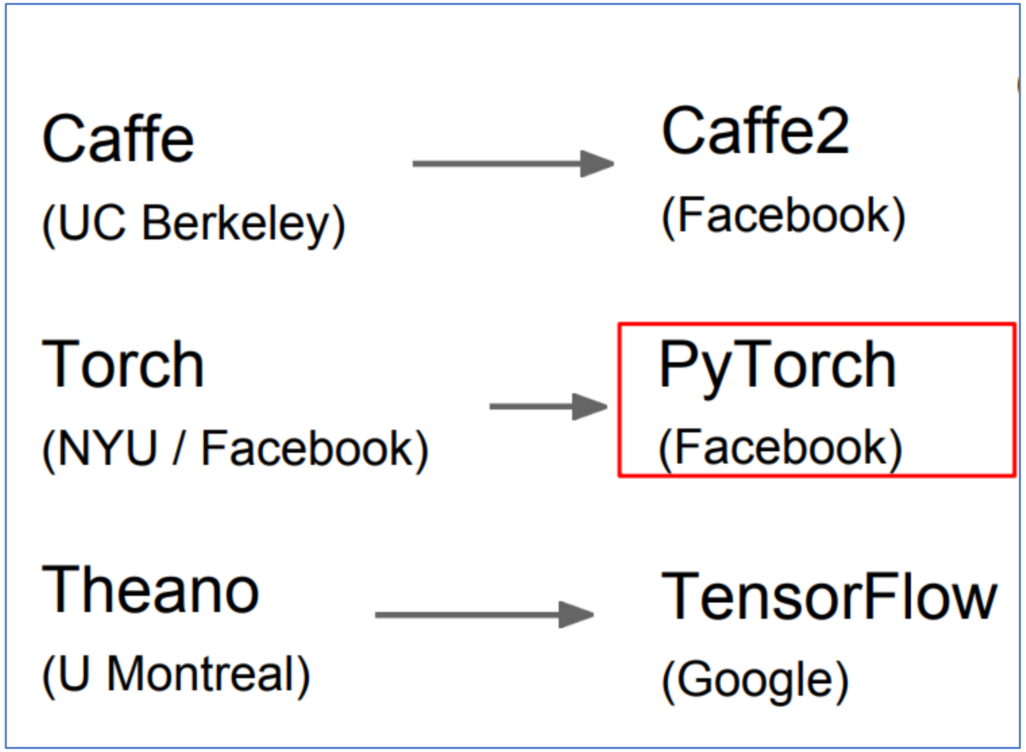
函数	导数	
$h_1 = x \times w$	$\frac{\partial h_1}{\partial w} = x$	$\frac{\partial h_1}{\partial x} = w$
$h_2 = h_1 + b$	$\frac{\partial h_2}{\partial h_1} = 1$	$\frac{\partial h_2}{\partial b} = 1$
$h_3 = h_2 \times -1$	$\frac{\partial h_3}{\partial h_2} = -1$	
$h_4 = \exp(h_3)$	$\frac{\partial h_4}{\partial h_3} = \exp(h_3)$	
$h_5 = h_4 + 1$	$\frac{\partial h_5}{\partial h_4} = 1$	
$h_6 = 1/h_5$	$\frac{\partial h_6}{\partial h_5} = -\frac{1}{h_5^2}$	

当x = 1,w = 0,b = 0时， 可得：

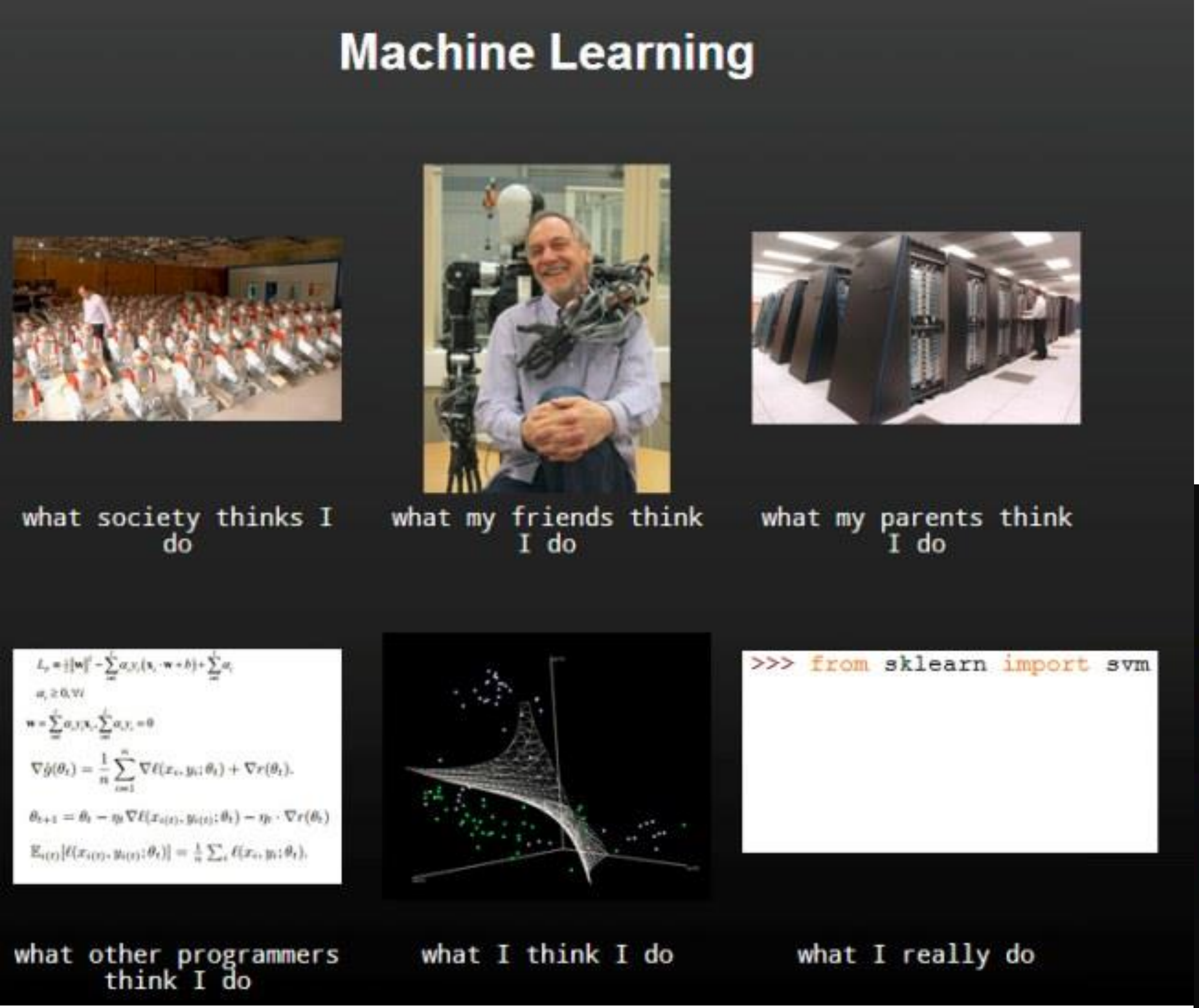
$$\begin{aligned}
&\frac{\partial f(x;w,b)}{\partial w}\Big|_{x=1,w=0,b=0} \\
&= \frac{\partial f(x;w,b)}{\partial h_6}\frac{\partial h_6}{\partial h_5}\frac{\partial h_5}{\partial h_4}\frac{\partial h_4}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial w} \\
&= 1 \times -0.25 \times 1 \times 1 \times -1 \times 1 \times 1 \\
&= 0.25.
\end{aligned}$$

计算图:静态 V.S. 动态

- 静态计算图：在编译时构建计算图，计算图构建好之后在程序运行时不能改变。
- 动态计算图：在程序运行时动态构建。
- 两种构建方式各有优缺点
  - 静态计算图在构建时可以进行优化，并行能力强，但灵活性比较差低。
  - 动态计算图则不容易优化，当不同输入的网络结构不一致时，难以并行计算，但是灵活性比较高。



Show me your code…

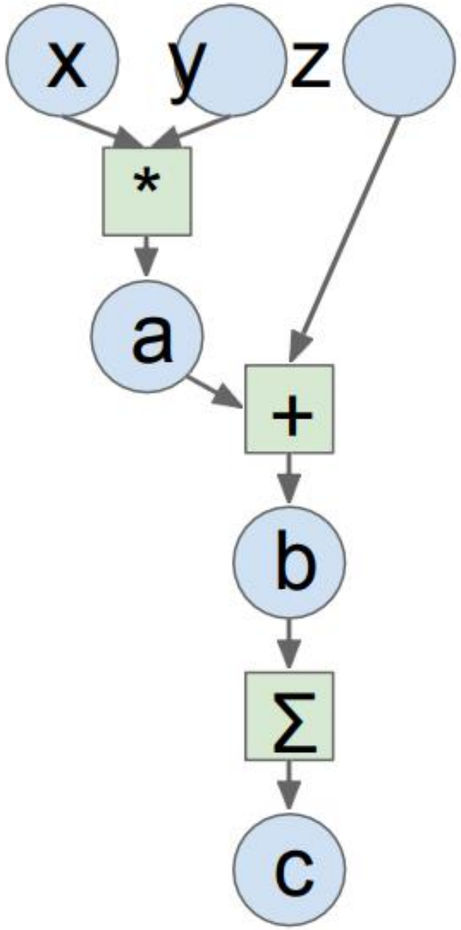


计算图代码赏析

Numpy

```python
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)
```

```python
grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



PyTorch

```python
import torch

N, D = 3, 4
x = torch.randn(N, D)
y = torch.randn(N, D)
z = torch.randn(N, D)

a = x * y
b = a + z
c = torch.sum(b)
```

Looks exactly like numpy!

1. 定义网络结构

demo_mnist_ann.py

```
class NeuralNetwork:
    def __init__(self, layers, activation,
opt_alg):
        """
        layers: layers of NNs
        Activation:activation function
        opt_alg:optimization algorithm
        """
        if activation == 'tanh':
            self.activation = tanh
            self.activation_deriv = tanh_deriv
        elif activation == 'sigmoid':
            self.activation = sigmoid
            self.activation_deriv = sigmoid_deriv
        elif activation == 'RELU':
            self.activation = RELU
            self.activation_deriv = RELU_deriv
        elif activation == 'RELU3':
            self.activation_deriv = RELU3_deriv
```

```
        if opt_alg == 'GD':
            self.opt = self.GD
        elif opt_alg == 'SGD':
            self.opt = self.SGD
        #initial parameters of ADAM
            self.mw = []
            self.mtheta = []
            self.vw = []
            self.vtheta = []
            for i in range(len(layers)-1):
                self.mw.append(np.zeros((layers[i+1], layers[i])))
                self.mtheta.append(np.zeros(layers[i+1]))
                self.mtheta[i] = np.mat(self.mtheta[i]).T
            for i in range(len(layers)-1):
                self.vw.append(np.zeros((layers[i+1], layers[i])))
                self.vtheta.append(np.zeros(layers[i+1]))
                self.vtheta[i] = np.mat(self.vtheta[i]).T

        #initial weights and thetas
        self.weights = []
        self.thetas = []
        #range in -1 to 1
        for i in range(len(layers)-1):
            self.weights.append(
                        2*(np.random.rand(layers[i+1], layers[i]))-
1)

self.thetas.append(2*(np.random.random(layers[i+1]))-1)
            self.thetas[i] = np.mat(self.thetas[i]).T
        self.layers = layers
```

2. 前向\反向传播

```
def propagation(self, x, k):
    “”“
    x:input
    return: output
    """
    temp = x
    for i in range(len(self.weights)):
        temp =self.activation(
         np.dot(self.weights[i], temp) + self.thetas[i]
         )
    z = k * temp
    return z

def backpropagation(self, x, error):
    """
    Compute the back propagation
    x:input
    return: back propagation
    K:output of each layer
    Delta:the propagation of each layer
    """
    n_w = len(self.weights)
    z = []
    K = []
    dweights = []
    dthetas = []
    delta = []
    for i in range(n_w):
        if i == 0:
            z.append(np.dot(self.weights[i], x) + self.thetas[i])
            K.append(x)
            delta.append(x)
        else:
            z.append(
                np.dot(self.weights[i], self.activation(z[i-1]))
                    + self.thetas[i])
            K.append(self.activation(z[i-1]))
            delta.append(self.activation(z[i-1]))

    for i in range(n_w-1, -1, -1):
        if i == n_w-1:
            delta[i] = np.multiply(error, self.activation_deriv(z[i]))
        else:
            delta[i] = np.multiply(
        np.dot(self.weights[i+1].T, delta[i+1]) ,
                    self.activation_deriv(z[i]) )
    for i in range(n_w):
        dweights.append(np.dot(delta[i], K[i].T))
        dthetas.append(delta[i])
    return dweights, dthetas

## 自定义训练算法 GD and SGD
def GD(self, X, Y, k, learning_rate, epochs):
    ### Try It Yourself
    for i in range(len(self.weights)):
        self.weights[i] +=  ###
        self.thetas[i] +=  ###
    return something
```

3. 数值结果 GD V.S. SGD

```
perf: 1151.717221082434 epochs: 9967 predict_true: 3456 precision: 0.6912
perf: 1151.7520690405495 epochs: 9968 predict_true: 3457 precision: 0.6914
perf: 1151.7268360006374 epochs: 9969 predict_true: 3457 precision: 0.6914
perf: 1151.737901898233 epochs: 9970 predict_true: 3457 precision: 0.6914
perf: 1151.7183555250201 epochs: 9971 predict_true: 3457 precision: 0.6914
perf: 1151.7078286233686 epochs: 9972 predict_true: 3457 precision: 0.6914
perf: 1151.7455567021823 epochs: 9973 predict_true: 3456 precision: 0.6912
perf: 1151.7292076041917 epochs: 9974 predict_true: 3456 precision: 0.6912
perf: 1151.7282099598415 epochs: 9975 predict_true: 3457 precision: 0.6914
perf: 1151.719959860039 epochs: 9976 predict_true: 3457 precision: 0.6914
perf: 1151.6755991201467 epochs: 9977 predict_true: 3457 precision: 0.6914
perf: 1151.6803086769437 epochs: 9978 predict_true: 3457 precision: 0.6914
perf: 1151.65095034867 epochs: 9979 predict_true: 3457 precision: 0.6914
perf: 1151.6722665448053 epochs: 9980 predict_true: 3457 precision: 0.6914
perf: 1151.6486455708905 epochs: 9981 predict_true: 3457 precision: 0.6914
perf: 1151.6618234982925 epochs: 9982 predict_true: 3457 precision: 0.6914
perf: 1151.663934855332 epochs: 9983 predict_true: 3458 precision: 0.6916
perf: 1151.6594579574469 epochs: 9984 predict_true: 3459 precision: 0.6918
perf: 1151.656087433337 epochs: 9985 predict_true: 3459 precision: 0.6918
perf: 1151.6402211049058 epochs: 9986 predict_true: 3459 precision: 0.6918
perf: 1151.6025620124294 epochs: 9987 predict_true: 3459 precision: 0.6918
perf: 1151.6574820039716 epochs: 9988 predict_true: 3459 precision: 0.6918
perf: 1151.670972740334 epochs: 9989 predict_true: 3459 precision: 0.6918
perf: 1151.599820279128 epochs: 9990 predict_true: 3459 precision: 0.6918
perf: 1151.5874068696567 epochs: 9991 predict_true: 3459 precision: 0.6918
perf: 1151.573963689703 epochs: 9992 predict_true: 3459 precision: 0.6918
perf: 1151.5816908132958 epochs: 9993 predict_true: 3460 precision: 0.692
perf: 1151.626513247648 epochs: 9994 predict_true: 3460 precision: 0.692
perf: 1151.564730155813 epochs: 9995 predict_true: 3460 precision: 0.692
perf: 1151.5427160377014 epochs: 9996 predict_true: 3459 precision: 0.6918
perf: 1151.5245993468843 epochs: 9997 predict_true: 3459 precision: 0.6918
perf: 1151.5361285780034 epochs: 9998 predict_true: 3459 precision: 0.6918
perf: 1151.4948603403739 epochs: 9999 predict_true: 3459 precision: 0.6918
```

```
perf: 7.341692123196259 epochs: 9966 predict_true: 3955 precision: 0.791
perf: 8.223755231679604 epochs: 9967 predict_true: 3957 precision: 0.7914
perf: 9.579641652290364 epochs: 9968 predict_true: 3957 precision: 0.7914
perf: 6.114119047257799 epochs: 9969 predict_true: 3955 precision: 0.791
perf: 8.314916494454025 epochs: 9970 predict_true: 3958 precision: 0.7916
perf: 8.550272513617722 epochs: 9971 predict_true: 3959 precision: 0.7918
perf: 7.878488553364032 epochs: 9972 predict_true: 3961 precision: 0.7922
perf: 9.544017749534385 epochs: 9973 predict_true: 3956 precision: 0.7912
perf: 6.981533952372505 epochs: 9974 predict_true: 3960 precision: 0.792
perf: 8.828661327789305 epochs: 9975 predict_true: 3954 precision: 0.7908
perf: 11.470910347928829 epochs: 9976 predict_true: 3950 precision: 0.79
perf: 7.2590667818852195 epochs: 9977 predict_true: 3946 precision: 0.7892
perf: 6.3452150416625175 epochs: 9978 predict_true: 3945 precision: 0.789
perf: 6.883654619156194 epochs: 9979 predict_true: 3943 precision: 0.7886
perf: 7.838274253272724 epochs: 9980 predict_true: 3952 precision: 0.7904
perf: 7.500713027480227 epochs: 9981 predict_true: 3951 precision: 0.7902
perf: 6.988023567477538 epochs: 9982 predict_true: 3948 precision: 0.7896
perf: 8.715132384583043 epochs: 9983 predict_true: 3953 precision: 0.7906
perf: 7.75041410303785 epochs: 9984 predict_true: 3954 precision: 0.7908
perf: 9.042222521378422 epochs: 9985 predict_true: 3947 precision: 0.7894
perf: 8.141650316270624 epochs: 9986 predict_true: 3949 precision: 0.7898
perf: 7.955961704505201 epochs: 9987 predict_true: 3946 precision: 0.7892
perf: 11.584425705382154 epochs: 9988 predict_true: 3955 precision: 0.791
perf: 8.60868174342482 epochs: 9989 predict_true: 3956 precision: 0.7912
perf: 8.22857393115915 epochs: 9990 predict_true: 3945 precision: 0.789
perf: 7.79985004179325 epochs: 9991 predict_true: 3946 precision: 0.7892
perf: 11.731010138358858 epochs: 9992 predict_true: 3949 precision: 0.7898
perf: 5.9108459608978485 epochs: 9993 predict_true: 3951 precision: 0.7902
perf: 7.4783986805230676 epochs: 9994 predict_true: 3953 precision: 0.7906
perf: 8.148068809418668 epochs: 9995 predict_true: 3952 precision: 0.7904
perf: 8.688817693198848 epochs: 9996 predict_true: 3947 precision: 0.7894
perf: 9.46231089903901 epochs: 9997 predict_true: 3946 precision: 0.7892
perf: 9.76619232814428 epochs: 9998 predict_true: 3947 precision: 0.7894
perf: 8.529349238501243 epochs: 9999 predict_true: 3944 precision: 0.7888
```

Exercise 1

1. 完善感知器的实现，构造并验证类似本讲稿P19-20四个算例的结果

2. 请简述反向传播算法的计算图表示方法，参考demo_mnist_ann.py，

   Build ANN From Scratch：

   ① 做一做FashionMNIST数据集或其他感兴趣标准数据集的分类问题：
   - 输入：28x28的图片灰度值矩阵，转换为764x1的向量；
   - 输出：十维单位向量，1的位置表示分类的数字
   - 训练集：5000张带标记的28x28的图片以及相应的数字标记；
   - 测试集：另外1000张28x28的数字图片

   ② 尝试不同的网络参数：
   - 三层全连接网络[784,28,10] 或更多（依赖于电脑性能）
   - 尝试用不同的激活函数：sigmoid函数、… …
   - 尝试用不同的优化算法：GD，SGD，**ADAM**

   ③ 否可以继续改善程序性能，给出你的尝试历史