《计算机视觉》

# 第四章　前馈神经网络

参考资料：[1] 邱锡鹏《神经网络与深度学习》
[2] 计算机视觉课程组资料

# 内容

- 神经网络
  - 神经元
  - 网络结构
- 前馈神经网络
  - 神经网络的直观理解
  - 参数学习
  - 反向传播算法
  - 优化问题

# 神经网络

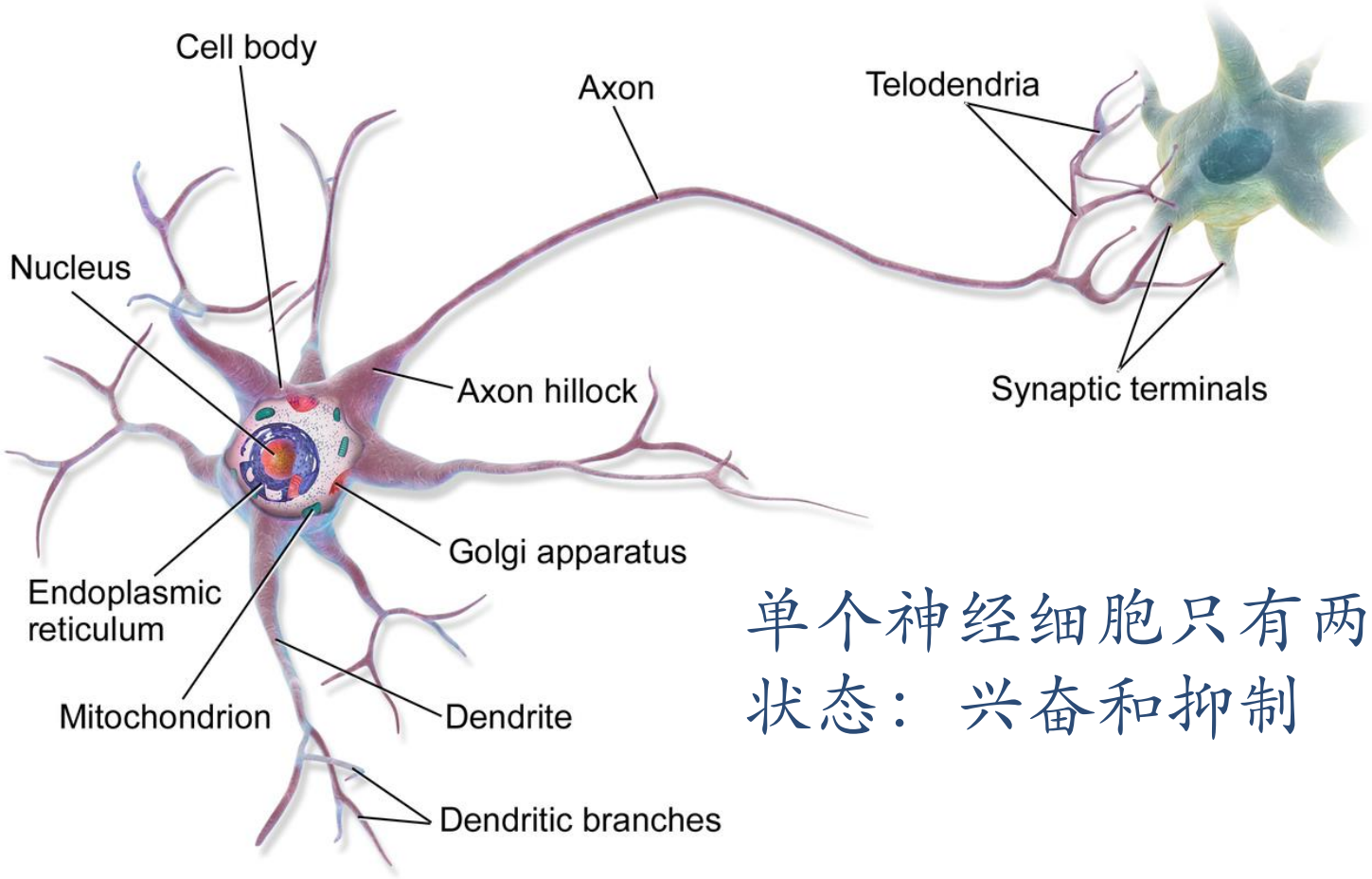
# 神经网络

- 神经网络最早是作为一种主要的连接主义模型。
  - 20世纪80年代后期，最流行的一种连接主义模型是分布式并行处理（Parallel Distributed Processing，PDP）网络，其有3个主要特性：
    - 1）信息表示是分布式的（非局部的）；
    - 2）记忆和知识是存储在单元之间的连接上；
    - 3）通过逐渐改变单元之间的连接强度来学习新的知识。
  - 引入误差反向传播来改进其学习能力之后，神经网络也越来越多地应用在各种机器学习任务上。

# 神经元

# 生物神经元

video: structure of brain



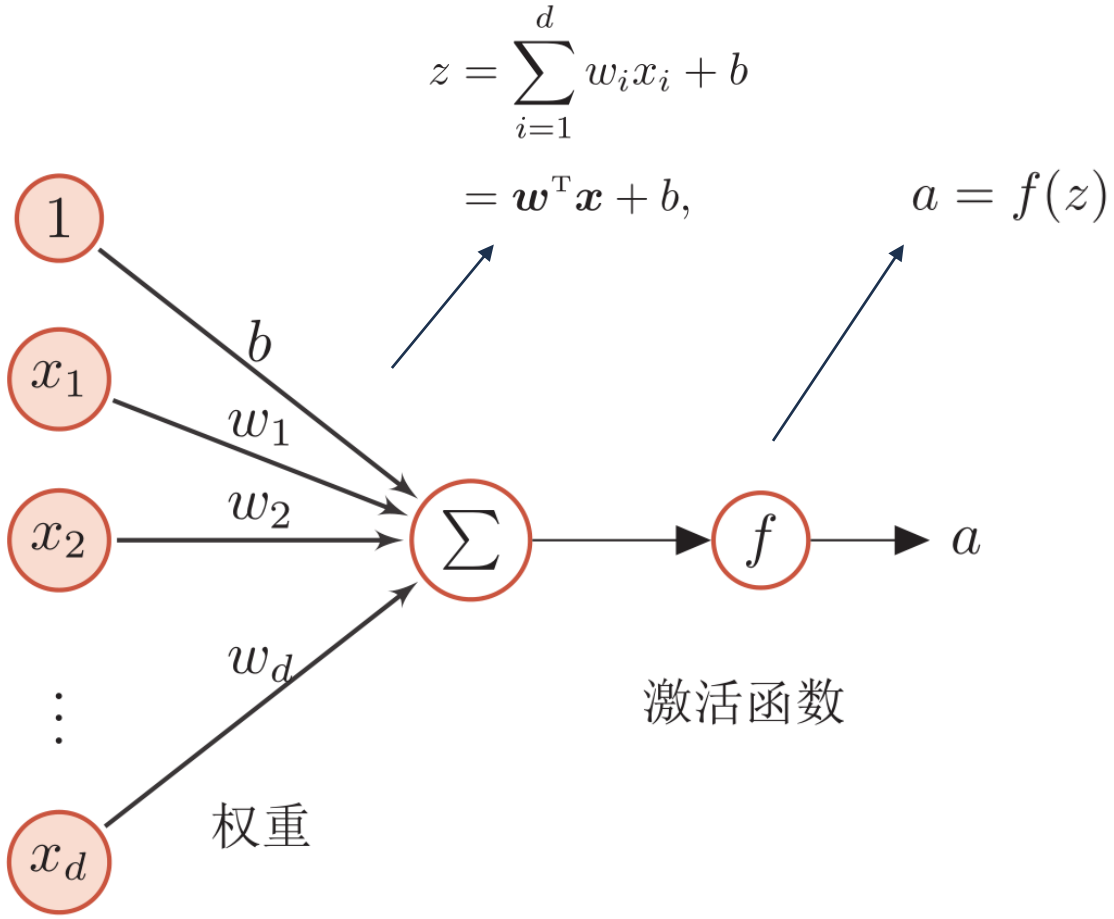

单个神经细胞只有两种状态：兴奋和抑制

1、一个生物神经元通常具有多个树突和一条轴突。树突用来接受信息，轴突用来发送信息。

2、当神经元所获得的输入信号的积累超过某个阈值时，它就处于兴奋状态，产生电脉冲。

3、轴突尾端有许多末梢可以与其他多个神经元的树突产生连接（突触），并将电脉冲信号传递给其它神经元。

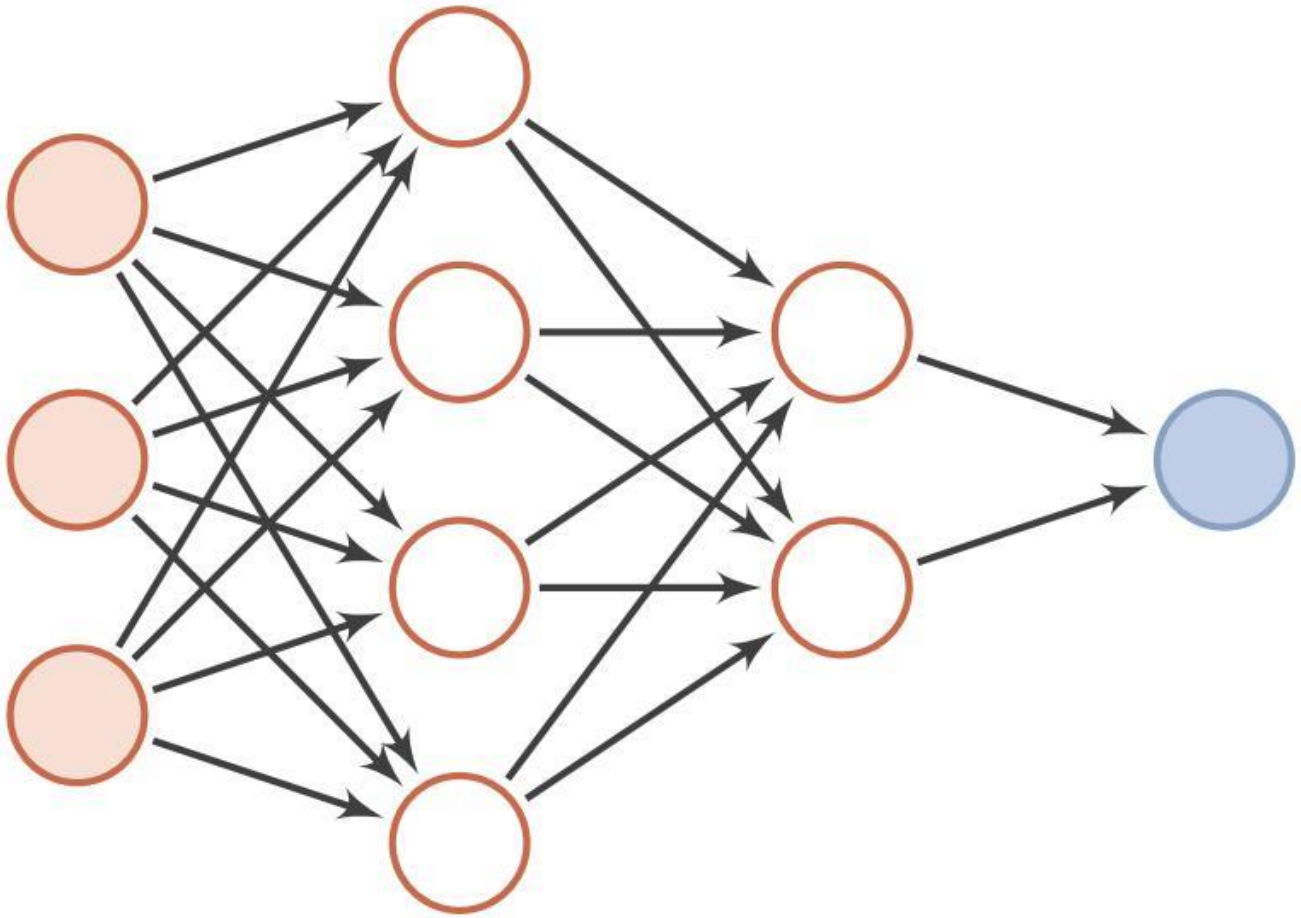
# 人工神经元




- X1，X2，X3 … 对应神经元d个树突接收到的信号
- a对应神经元轴突的输出信号，输出信号可以传输给其他神经元
- ω1X1表示X1通过树突到达细胞体之后的信号
- d个信号的加权和模拟输入信号的积累。
- 激活函数判断细胞积累是否超过某个阈值，模拟神经元的兴奋状态，产生电脉冲输出到轴突

# 神经网络的作用





1、神经网络通过大量相互连接的神经元**模拟人类大脑**处理问题。

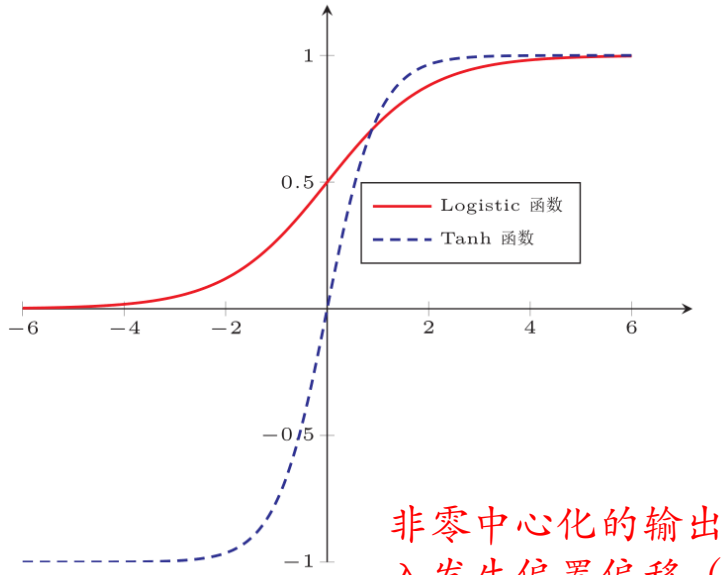
2、单个神经元解决简单问题，多个神经元就能**解决复杂问题**。

3、神经网络具有很强的**表达能力。**

# 常见激活函数

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

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




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sigmoid (logistic)函数的输出恒大于0

# 常见激活函数

$$\text{ReLU}(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$

$$= \max(0, x).$$

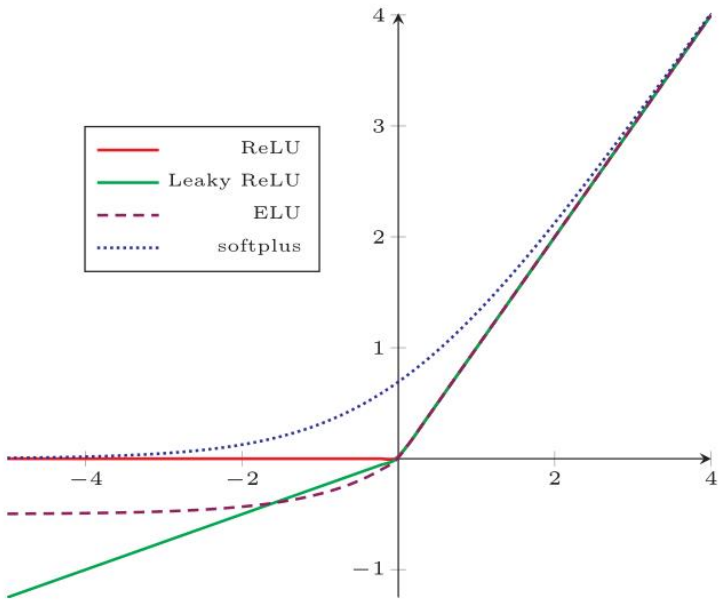
$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \gamma x & \text{if } x \leq 0 \end{cases}$$

$$\text{PReLU}_i(x) = \begin{cases} x & \text{if } x > 0 \\ \gamma_i x & \text{if } x \leq 0 \end{cases}$$

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

$$= \max(0, x) + \min(0, \gamma(\exp(x) - 1))$$

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




- 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

如何选择激活函数?

# 常见激活函数及其导数

| 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 | $f(x) = \max(0, x) + \min\Big(0, \gamma\big(\exp(x) - 1\big)\Big)$ | $f'(x) = I(x > 0) + I(x \leq 0) \cdot \gamma \exp(x)$ |
| SoftPlus 函数 | $f(x) = \log\big(1 + \exp(x)\big)$ | $f'(x) = \frac{1}{1+\exp(-x)}$ |

# 激活函数的性质 (剧透，why?)

- 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
  - ???

# 人工神经网络

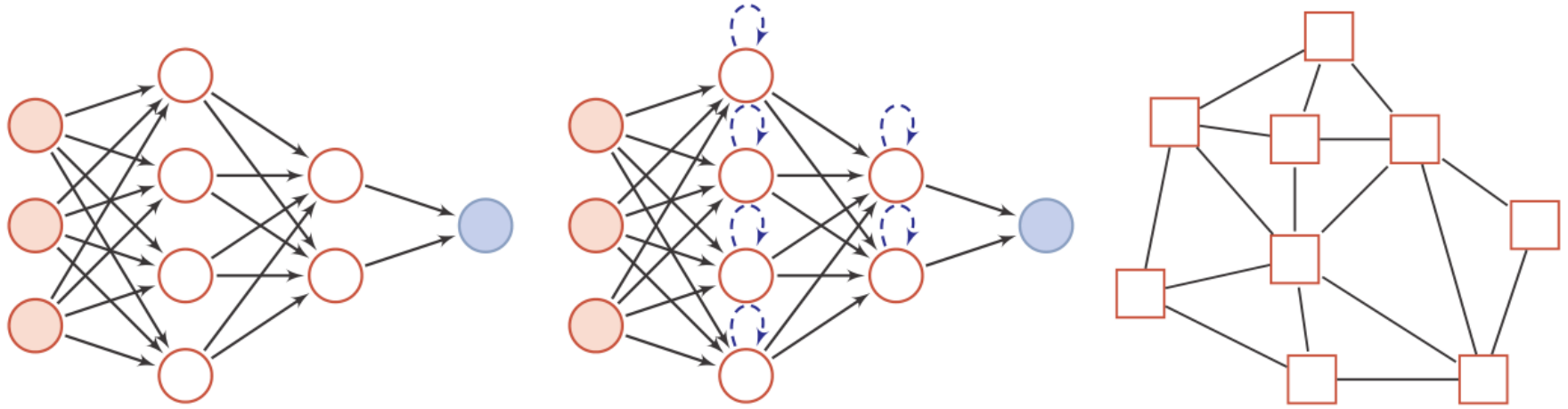
- 人工神经网络主要由大量的神经元以及它们之间的有向连接构成。因此考虑三方面：

- 神经元的激活规则
  - 主要是指神经元输入到输出之间的映射关系，一般为非线性函数。
- 网络的拓扑结构
  - 不同神经元之间的连接关系。
- 学习算法
  - 通过训练数据来学习神经网络的参数。

Why?

# 网络结构

- 人工神经网络由神经元模型构成，这种由许多神经元组成的信息处理网络具有并行分布结构。



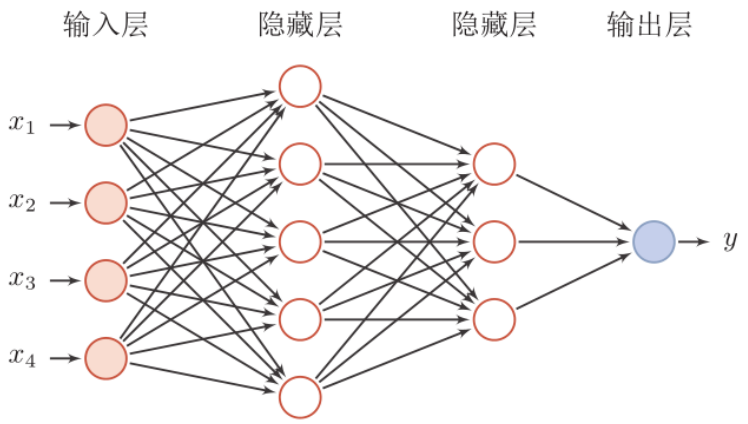
(a) 前馈网络　(b) 记忆网络　(c) 图网络

圆形节点表示一个神经元，方形节点表示一组神经元。

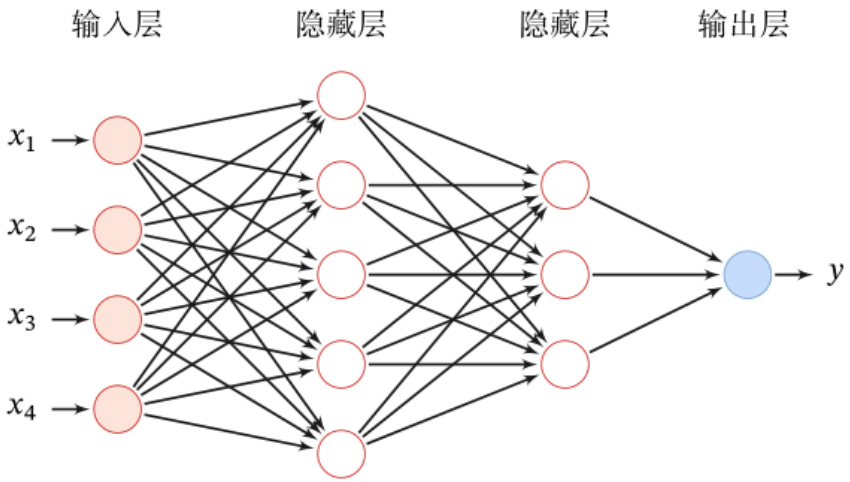
# 前馈神经网络

# 网络结构

- 前馈神经网络（全连接神经网络、多层感知器）
  - 各神经元分别属于不同的层，层内无连接。
  - 相邻两层之间的神经元全部两两连接。
  - 整个网络中无反馈，信号从输入层向输出层单向传播，可用一个有向无环图表示。



# 前馈网络




给定一个前馈神经网络，用下面的记号来描述这样网络：

| 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） |

# 信息传递过程

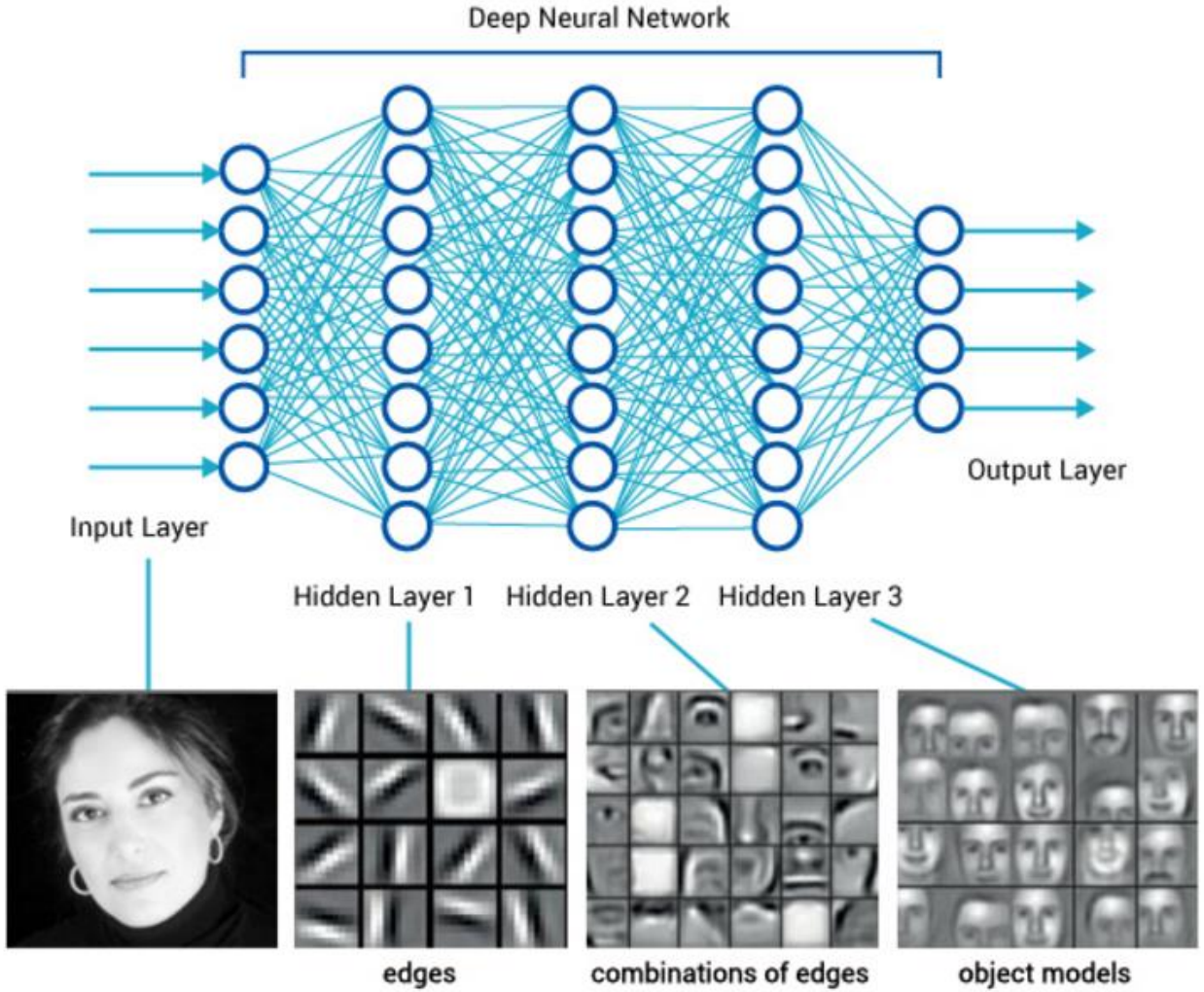
- 前馈神经网络通过下面公式进行信息传播。

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

- 前馈计算：

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

# 深层前馈神经网络


Deep Neural Network
Input Layer
Hidden Layer 1
Hidden Layer 2
Hidden Layer 3
Output Layer
edges
combinations of edges
object models

# 通用近似定理

**定理 4.1 – 通用近似定理（Universal Approximation Theorem）[Cybenko, 1989, Hornik et al., 1989]：** 令 $\varphi(\cdot)$ 是一个非常数、有界、单调递增的连续函数，$\mathcal{I}_d$ 是一个 $d$ 维的单位超立方体 $[0,1]^d$，$C(\mathcal{I}_d)$ 是定义在 $\mathcal{I}_d$ 上的连续函数集合。对于任何一个函数 $f \in C(\mathcal{I}_d)$，存在一个整数 $m$，和一组实数 $v_i, b_i \in \mathbb{R}$ 以及实数向量 $\mathbf{w}_i \in \mathbb{R}^d$，$i = 1, \cdots, m$，以至于我们可以定义函数

$$F(\mathbf{x}) = \sum_{i=1}^{m} v_i \varphi(\mathbf{w}_i^{\mathrm{T}} \mathbf{x} + b_i), \tag{4.33}$$

作为函数 $f$ 的近似实现，即

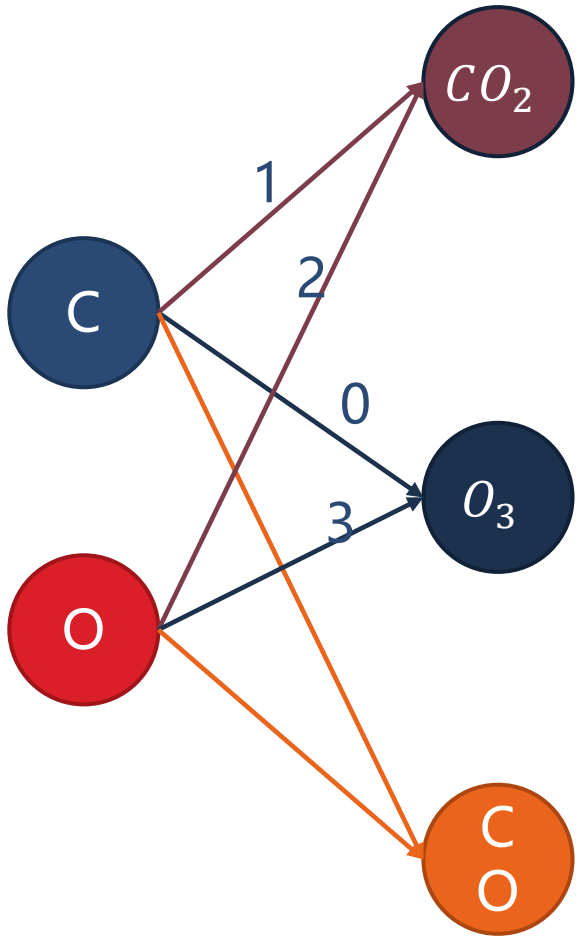
$$|F(\mathbf{x}) - f(\mathbf{x})| < \epsilon, \forall \mathbf{x} \in \mathcal{I}_d. \tag{4.34}$$

其中 $\epsilon > 0$ 是一个很小的正数。

根据通用近似定理，对于具有线性输出层和至少一个使用“挤压”性质的激活函数的隐藏层组成的前馈神经网络，只要其隐藏层神经元的数量足够，它可以以任意的精度来近似任何从一个定义在实数空间中的有界闭集函数。

# 神经网络的直观理解

# 神经网络的理解




假设输入x是二维向量

- 维度一决定C原子数量
- 维度二决定O原子数量

左图是输出y为三维向量的情况，通过改变权重的值可以获得不同的物质,如下就是我们常见的y = wx

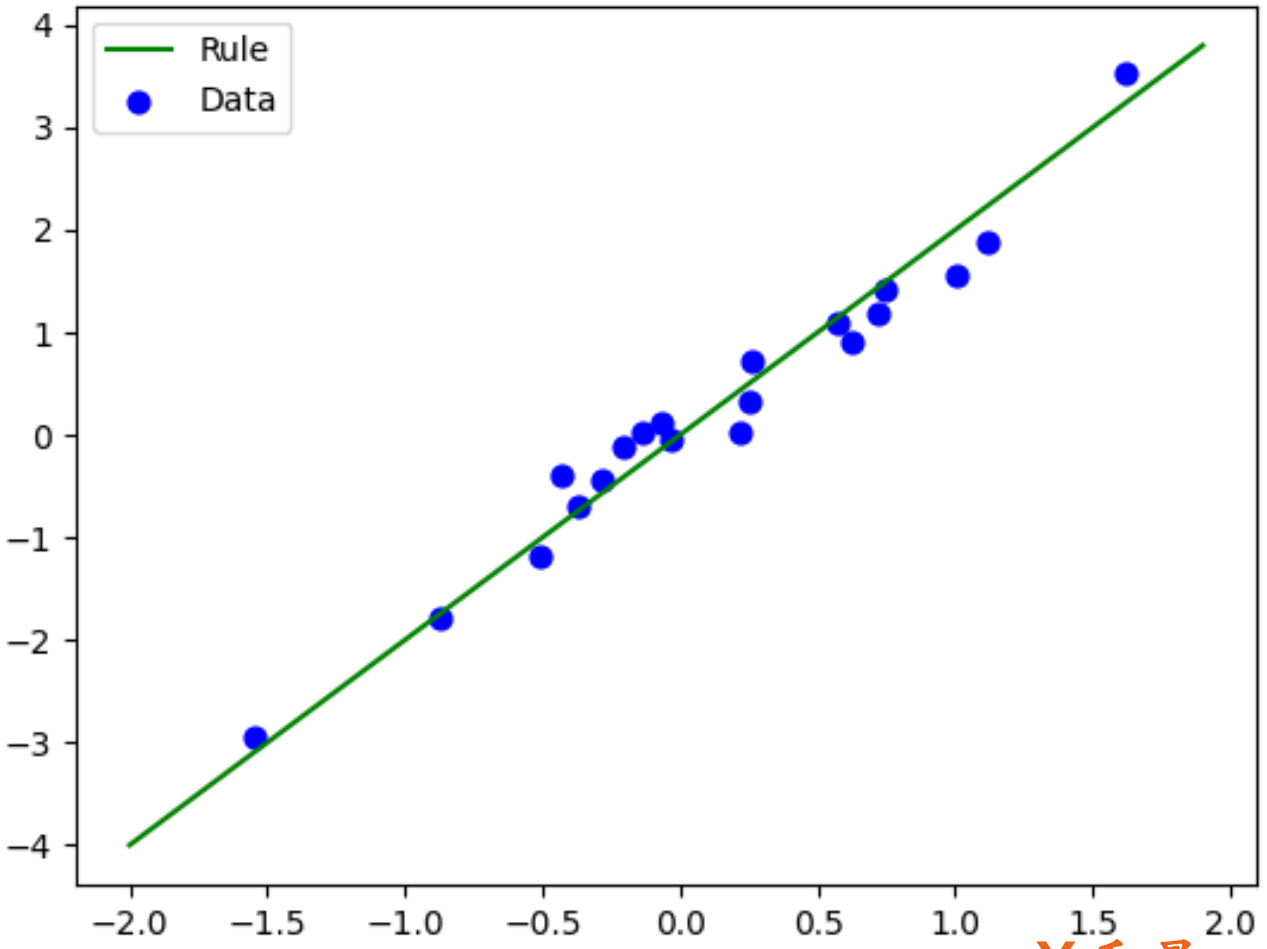
$$\begin{bmatrix} CO_2 \\ O_3 \\ CO \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 0 & 3 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} C \\ O \end{bmatrix} \qquad (1)$$

- 我们希望通过网络能从[C,O]转变到$[CO_2, O_3, CO]$，那么**网络的学习过程就是将w的数值尽可能接近(1)的过程。**
- 如果在其后面再加一层，就表示可以通过**组合**$[CO_2, O_3, CO]$从而形成更高层次的物质

每层神经网络的物理理解：**通过现有的不同物质的组合形成新物质**。

# 案例-----任务A一元线性回归

Y 总价：1维




X重量：1维

已知一部分西瓜的重量x与其对应的总价y的关系，考虑称重误差，想要知道任意重量的总价。

# 案例-----任务B手写体数字识别

X 图像: 784 维

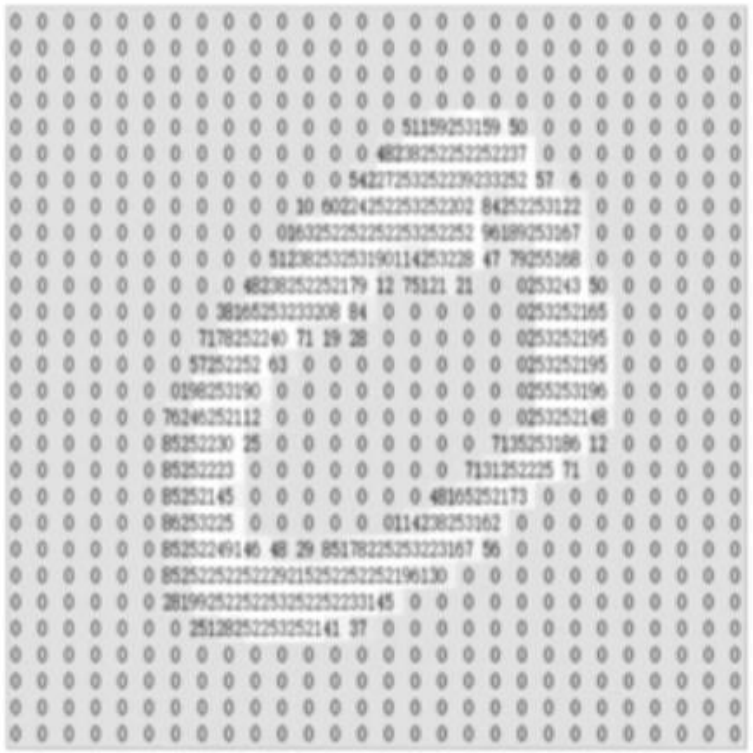
Y 数字: 10 维

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

求手写体数字图像x，与各数字可能性y的关系，想要知道任意图像是什么数字

# 案例-----任务B手写体数字识别

$X$ 图像: 784 维



$Y$ 数字: 10 维

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

图像，在计算机中也只是一堆数字，控制着每个像素的颜色，黑白模式时，255表示最亮，0表示最暗

这里每个图像都有28x28=784个像素，可以伸展成一个784维的向量，而输出是维度为10的向量，每个位置的数值都对应着一种可能性

# 数据准备

## 任务A：总样本：100 训练集/测试集：8:2

任务A 一元回归

```
#导入tensorflow+numpy
import tensorflow as tf
import numpy as np

#即显模式
tf.enable_eager_execution()
#生成数据
n_example = 100
X = tf.random_normal([n_example])
#称重误差
noise = tf.random_uniform([n_example],-0.5,0.5)
Y = X * 2 + noise
#训练集
train_x = X[:80]
train_y = Y[:80]
#测试集
test_x = X[80:]
test_y = Y[80:]
```

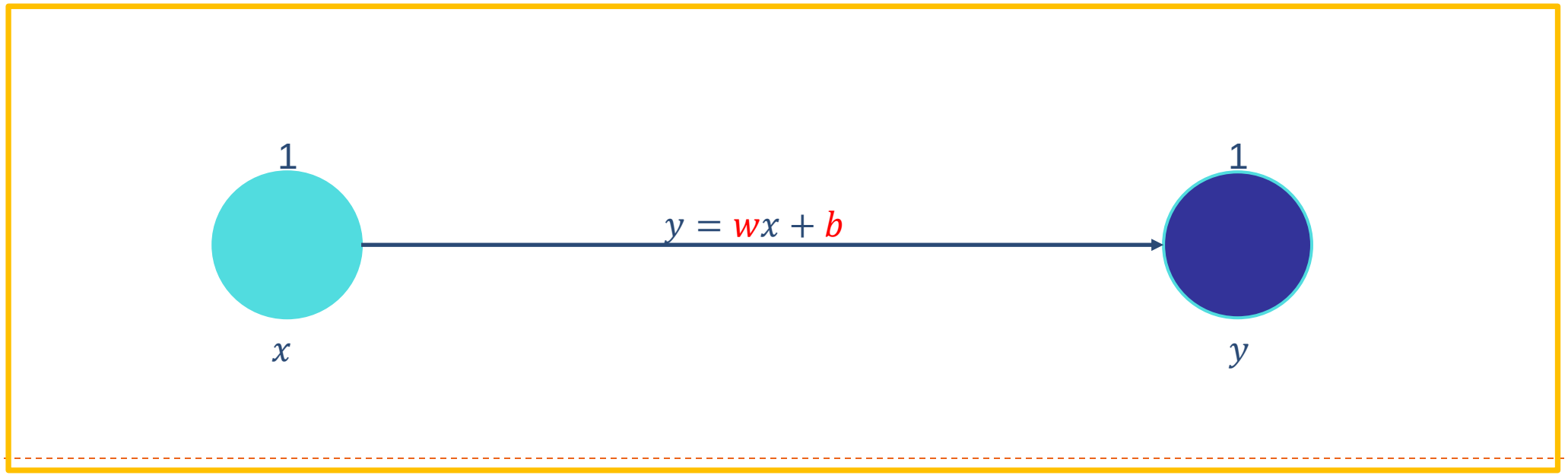
## 任务B:总样本:70000 训练集/测试集6:1

任务B 手写体数字识别

```
#导入tensorflow + numpy
import tensorflow as tf
import numpy as np
#即显模式
tf.enable_eager_execution()
#导入数据
train,test = tf.keras.datasets.mnist.load_data()
(train_x,train_y),(test_x,test_y) = train,test
#形状和类型
train_x = train_x.reshape([-1,784]).astype('float32')
test_x = test_x.reshape([-1,784]).astype('float32')
train_y = train_y.astype('int32')
test_y = test_y.astype('int32')
#训练、测试样本量
print('训练集%s, %s' %(train_x.shape,train_y.shape))
print('测试集%s, %s' %(test_x.shape,test_y.shape))
# 训练集(60000, 784), (60000,)
# 测试集(10000, 784), (10000,)
```

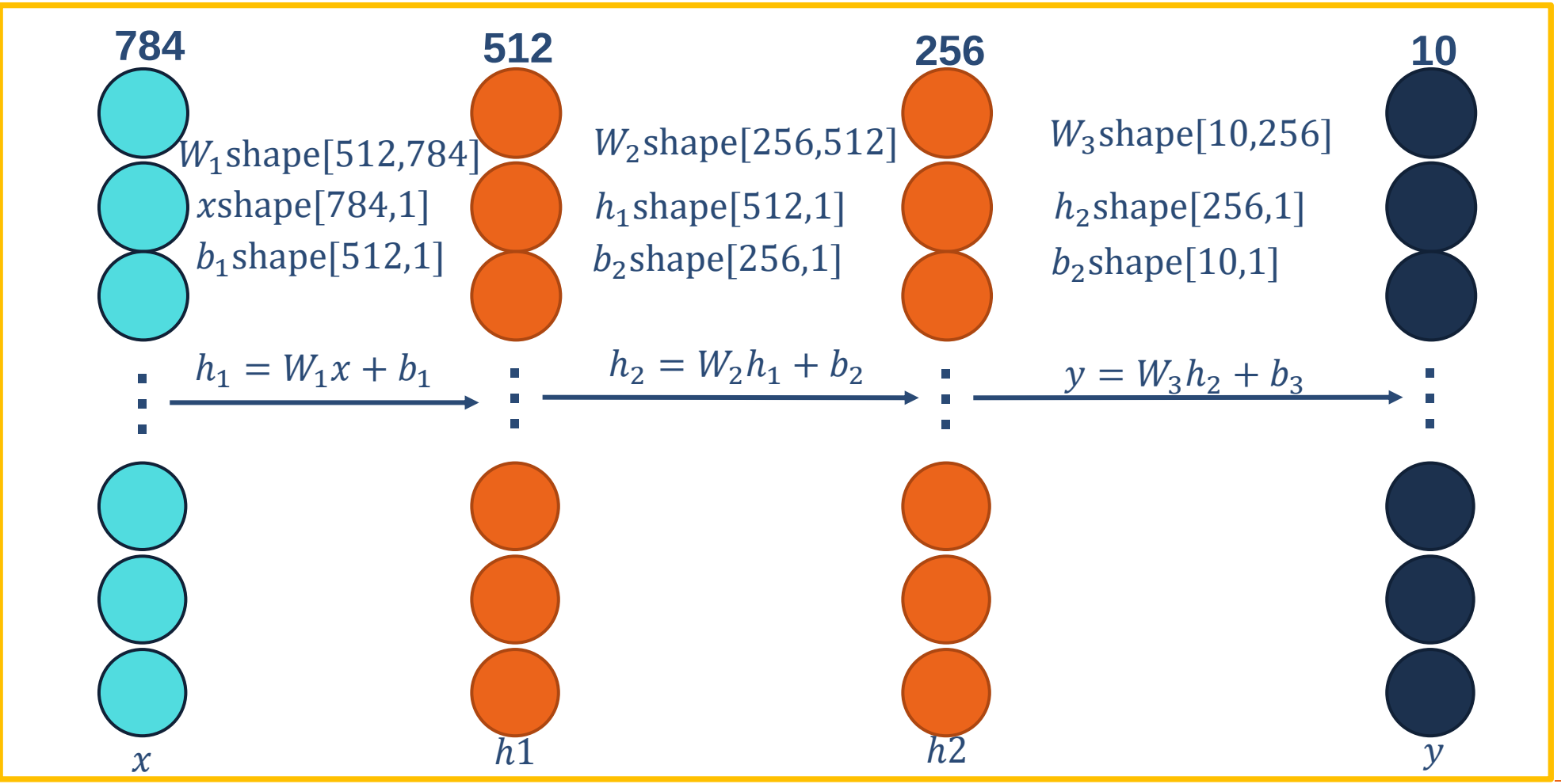
# 案例-----一元线性回归

输入和输出的关系是由模型来决定的，整理好数据的下一步就是选择适当的模型，现实中我们并不知道输入和输出的关系是什么，只能凭经验猜测，所以西瓜总价任务用$y = wx + b$线性模型直接从x得到y，要学习的参数是ω和b



# 任务分析

任务B 手写体数字识别

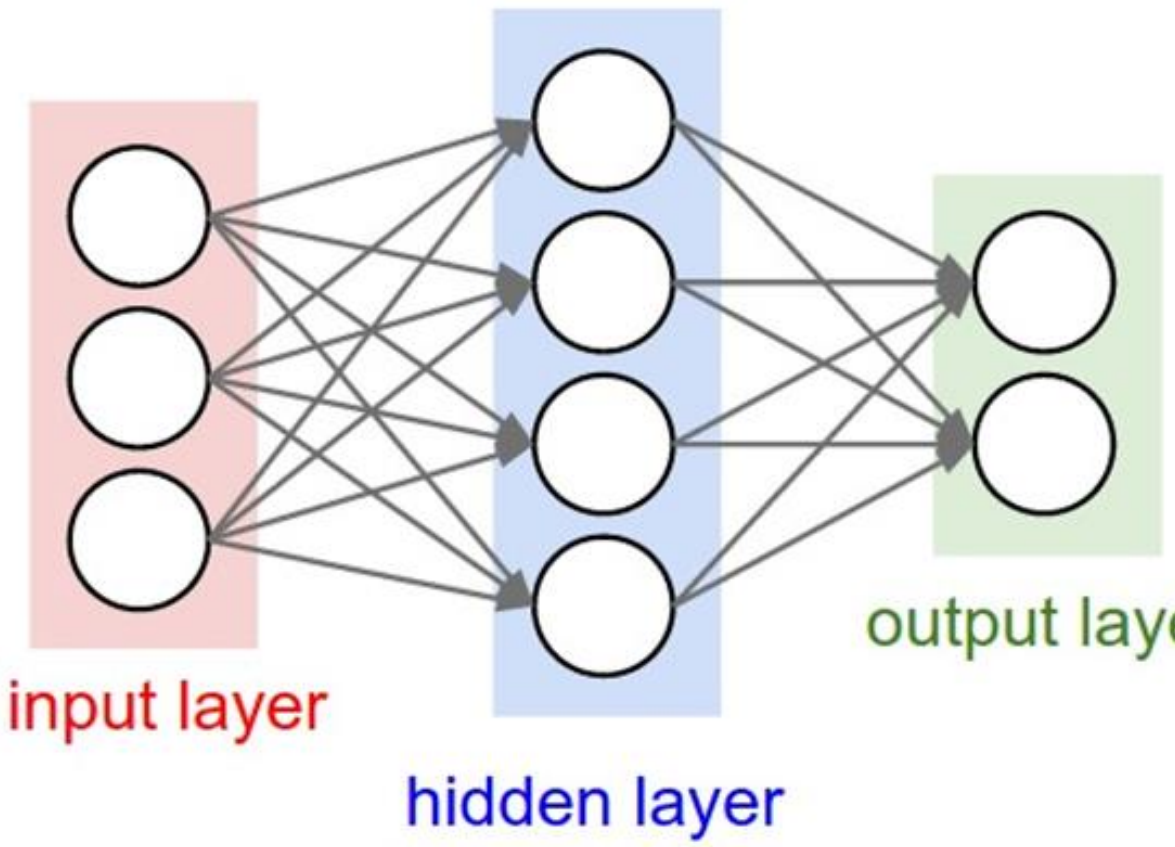



对于数字识别任务，**直接从x到y是很困难的。**

**思考：**

此时我们在线性模型前多增加两个线性模型，或者多增加100个线性模型，会有什么效果?

# 神经网络的表达能力---------激活函数




## 我们先考虑不使用激活函数的情况

假设H1为隐藏层神经元，X1为输入,y1为输出，则：

$$H1 = \omega 1X1 + b1$$

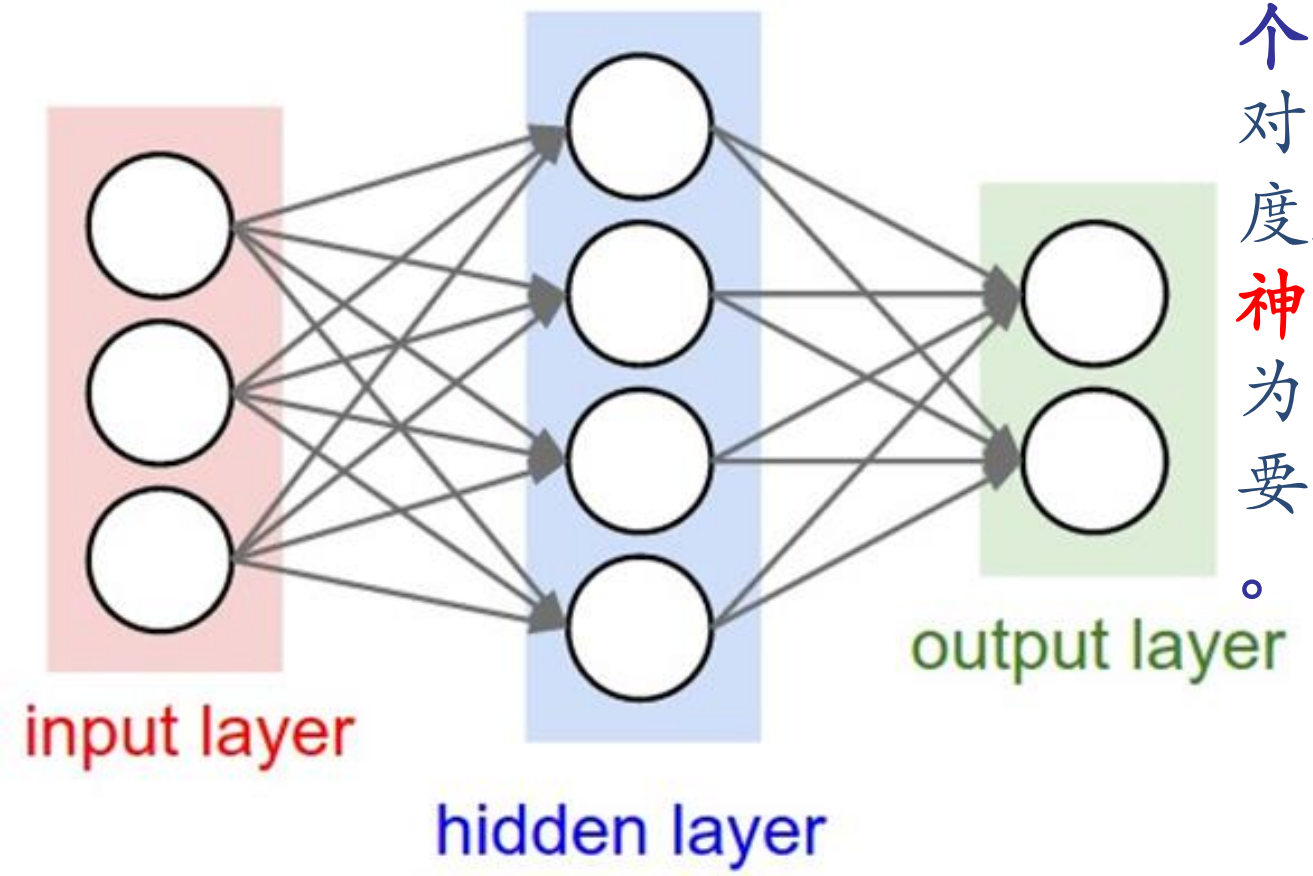
$$y1 = \omega 2H1 + b2$$

即：

$$y1 = \omega 2(\omega 1X1+b1) + b2$$

令$\omega 1X1+b1 = X2$,则 $y1 = w2X2 + b2$

# 神经网络的表达能力---------激活函数




一系列线性方程的运算最终都可以用一个线性方程表示。

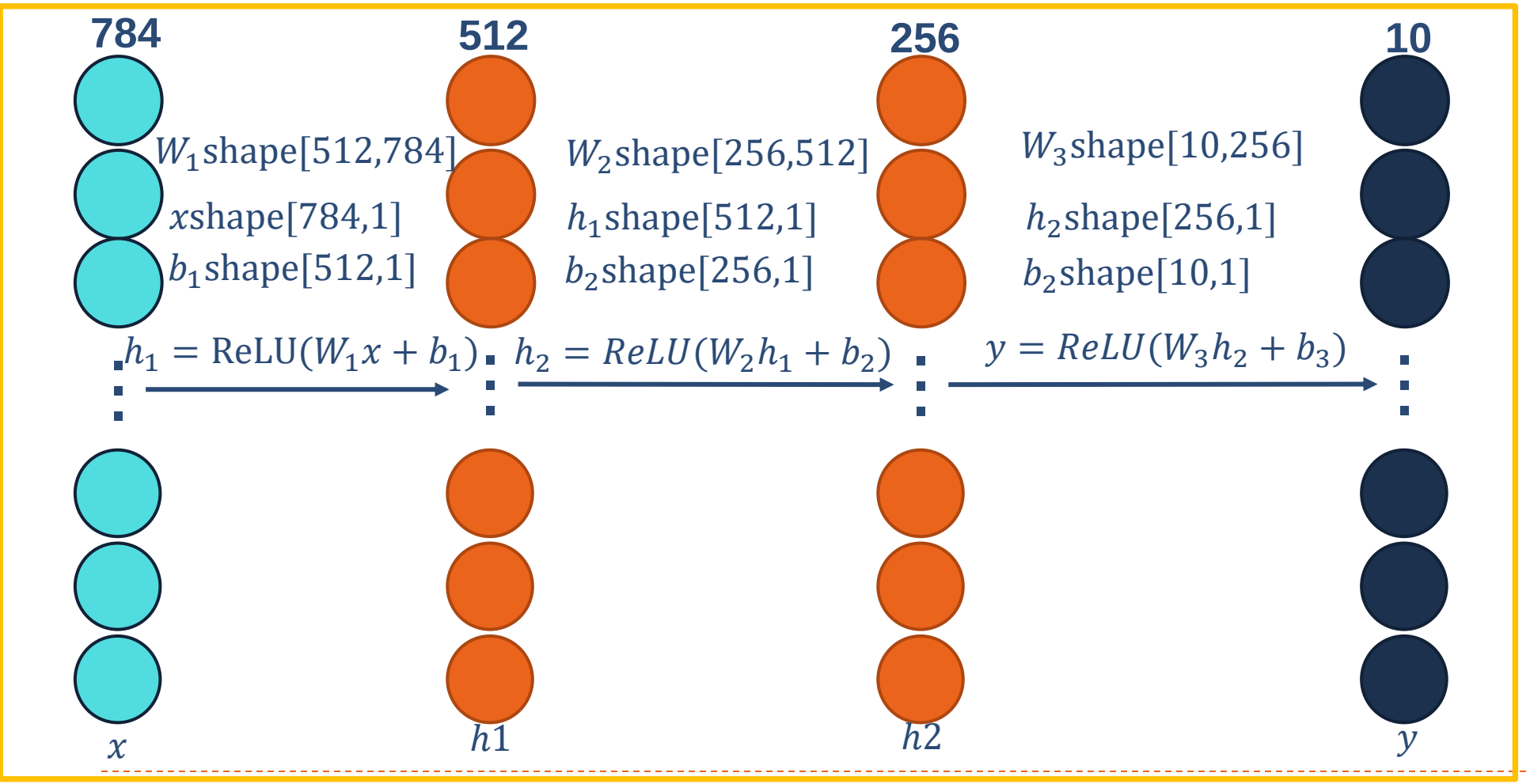
对于两次神经网络是这样，就算网络深度加到100层，也依然是这样。这样的话**神经网络就失去了意义**。

为了更好的增强网络的表达能力，我们要为网络添加**非线性功能（激活函数）**。

三层神经网络（输入，全连接，输出）可以逼近任意连续函数

# 任务分析

任务B 手写体数字识别




我们在线性模型基础上，增加若干个线性和非线性**(激活函数)**的组合**(隐藏层)**,隐藏层维度和层数可任意选择，在一定深度下，越多拟合能力越强，最后将维度调整到输出维度的部分为**输出层**

# 模型构建

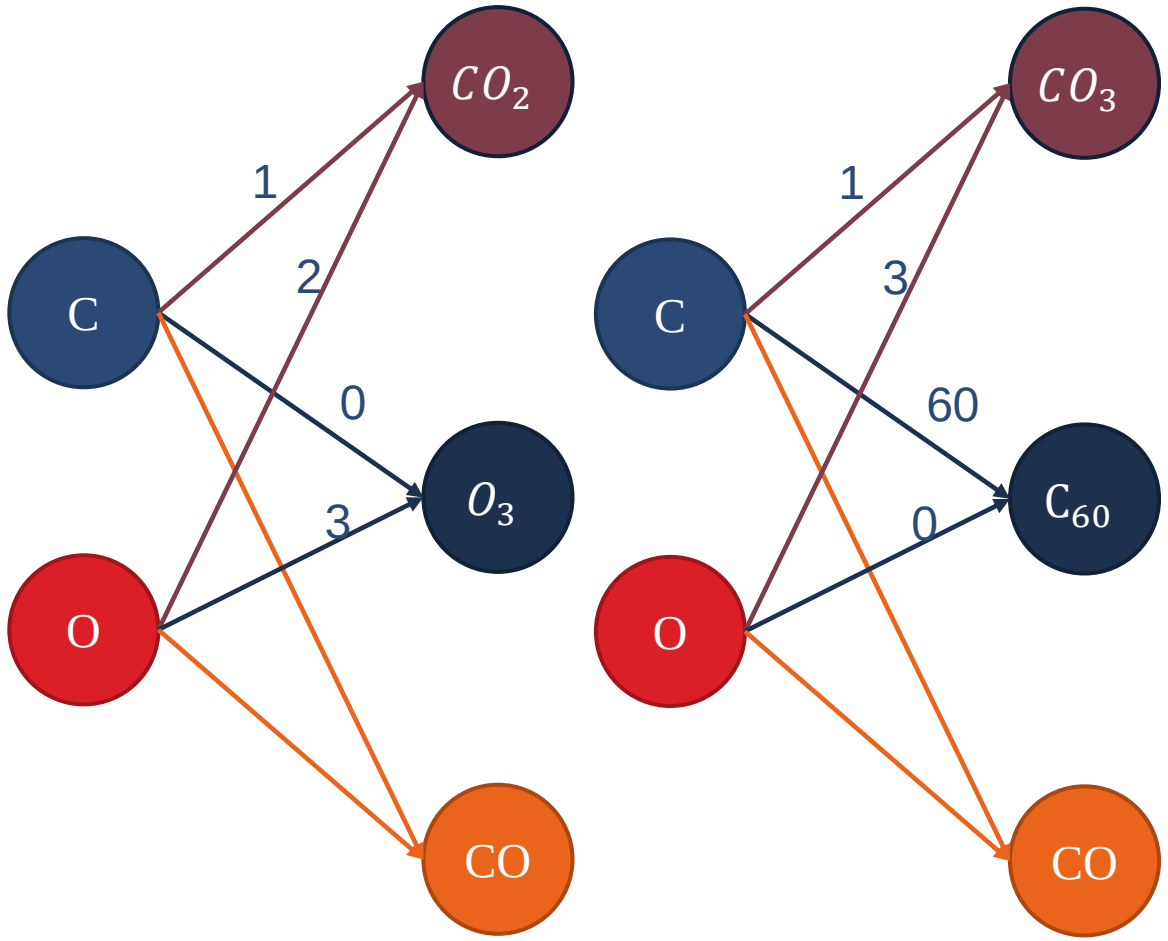
## 任务A 一元回归模型

```
#定义模型类
class Model(object):
    def __init__(self):
        #参数初始化
        self.w = tf.Variable(1.0)
        self.b = tf.Variable(1.0)
    def __call__(self,x):
        #正向传递
        y = self.w * x + self.b
        return y
#实例化模型
model = Model()
```

## 任务B 手写体数字识别模型

```
#定义模型类
class Model(object):
    #初始化方法
    def inits(self,shape):
        return tf.random_uniform(shape,
                                 minval = -np.sqrt(5) * np.sqrt(1.0/shape[0]),
                                 maxval = np.sqrt(5) * np.sqrt(1.0/shape[0]))
    def __init__(self):
        #参数初始化
        self.W1 = tf.Variable(self.inits([784,512]))
        self.b1 = tf.Variable(self.inits([512]))
        self.W2 = tf.Variable(self.inits([512,256]))
        self.b2 = tf.Variable(self.inits([256]))
        self.W3 = tf.Variable(self.inits([256,10]))
        self.b3 = tf.Variable(self.inits([10]))
    def __call__(self, x):
        #正向传递
        h1 =tf.nn.relu(tf.matmul(x,self.W1) + self.b1)
        h2 = tf.nn.relu(tf.matmul(h1,self.W2) + self.b2)
        y = tf.nn.relu(tf.matmul(h2,self.W3) + self.b3)
        return y
#实例模型
model = Model()
```

# 神经网络的训练




三层神经网络可以表达任意连续函数
所以对于我们要求解的大部分问题，我们的目标就是让网络的输出尽可能接近想要预测的值，从而让神经网络更好的拟合对应问题
总而言之，**训练就是要调整权重矩阵ω，让当前网络的预测值接近我们真正想要的目标值**

我们，希望通过网络能从[C,O]转变到$[CO_2, O_3, CO]$,也就是(1)中的情况，这是我们的理想状态，此时的权重矩阵能让神经网络完美拟合我们的问题

当权重矩阵如(2)所示时，我们就需要**调整权重矩阵ω:**

要得到$O_3$而不是$C_{60}$
- 可以将C原子的数量降低，氧原子数量上升

要得到$CO_2$而不是$CO_3$
- 可以将O原子数量上升，C原子不变

# 参数学习

# 神经网络的损失函数

训练过程就是，调整权重矩阵ω，让当前网络的预测值接近我们真正想要的目标值，不妨将**网络的预测值与目标值的差距**称为**损失值**
所以网络训练的最终目的就是**调整权重矩阵ω让损失值loss尽可能小**
那么如何知道哪些权重需要更改？是调高还是调低？
我们需要一个**度量任意某个权重ω好坏的方法**：当发现ω不利于损失值下降时，将ω调小，当发现ω有利于损失值下降时将ω调高
也就是说要找到一个从权重ω输入，输出损失值loss的评估函数，这个函数称为**损失函数**
对于回归任务使用的损失中最常见的是

- **L1 Loss--**绝对值损失Mean Absolute Error(MAE)
- **L2 Loss--**均方差损失Mean Squared Error (MSE)

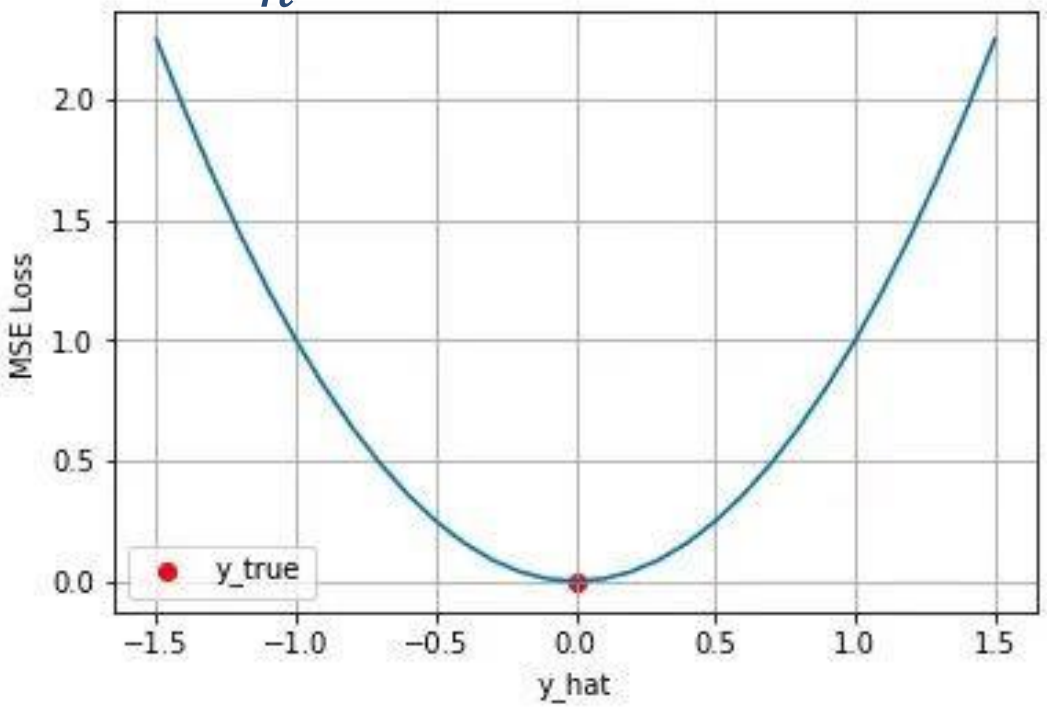
对于分类任务使用的损失中最常见的是

- **交叉熵损失**

# 神经网络的回归损失函数---------平方损失函数L2 Loss

均方差损失Mean Squared Error (MSE)，是机器学习、深度学习回归任务中最常用的一种损失函数，也称为 L2 Loss，对于任务A，均方差可以写成

$$loss = \frac{1}{n}[(pre_1 - y_1)^2 + (pre_2 - y_2)^2 + \cdots + (pre_n - y_n)^2]$$

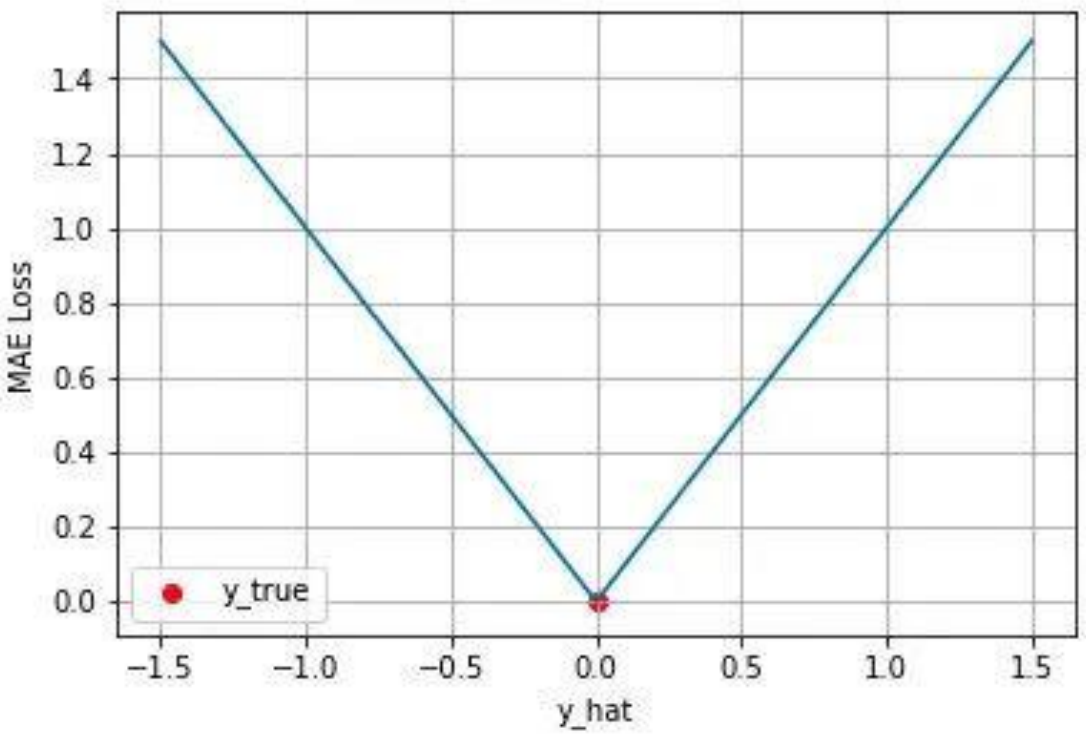



- MSE 损失的最小值为 0（当预测等于真实值时），最大值为无穷大。
- 随着预测与真实值绝对误差$|pred - y|$的增加，MSE 损失呈平方增长

# 神经网络的回归损失函数---------绝对值损失L1 Loss

绝对值损失Mean Absolute Error(MAE)，是另一类常用的损失函数，也称为 L1 Loss，对于任务A，绝对值损失可以写成

$$loss = \frac{1}{n}[|pre_1 - y_1| + |pre_2 - y_2| + \cdots + |pre_n - y_n|]$$




- MAE 损失的最小值为 0（当预测等于真实值时），最大值为无穷大。
- 随着预测与真实值绝对误差$|pred - y|$的增加，MAE 损失呈线性增长

# 神经网络的分类损失函数

- 对于多分类问题
  - 如果使用Softmax回归分类器，相当于网络最后一层设置C 个神经元，其输出经过Softmax函数进行归一化后可以作为每个类的条件概率。

二分类
Sigmoid
(Logistic)

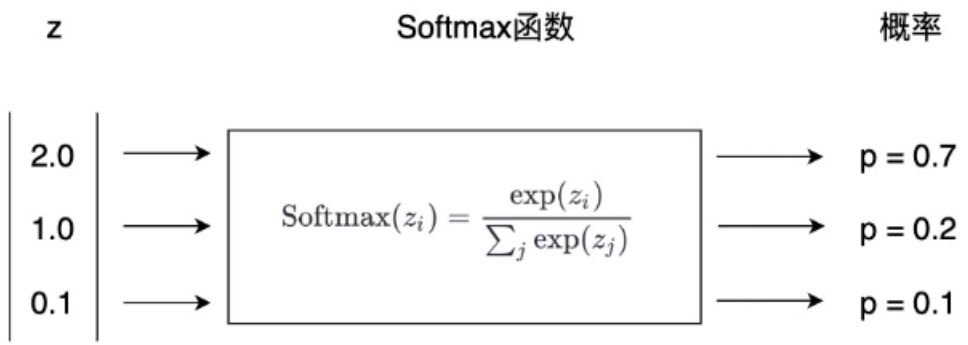
$$g(z) = \frac{1}{1+e^{-z}}$$

多分类
Softmax

$$\text{Softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \qquad \hat{\mathbf{y}} = \text{softmax}(\mathbf{z}^{(L)})$$

- 采用交叉熵损失函数，对于样本(x,y)，其损失函数为

$$\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}}) = -\mathbf{y}^{\mathrm{T}} \log \hat{\mathbf{y}}$$



# 模型构建

**任务A 一元回归模型损失函数定义**

```
#定义均方差误差
def loss(prediction,label):
    loss = tf.reduce_mean(tf.square(prediction - label))
    return loss
```

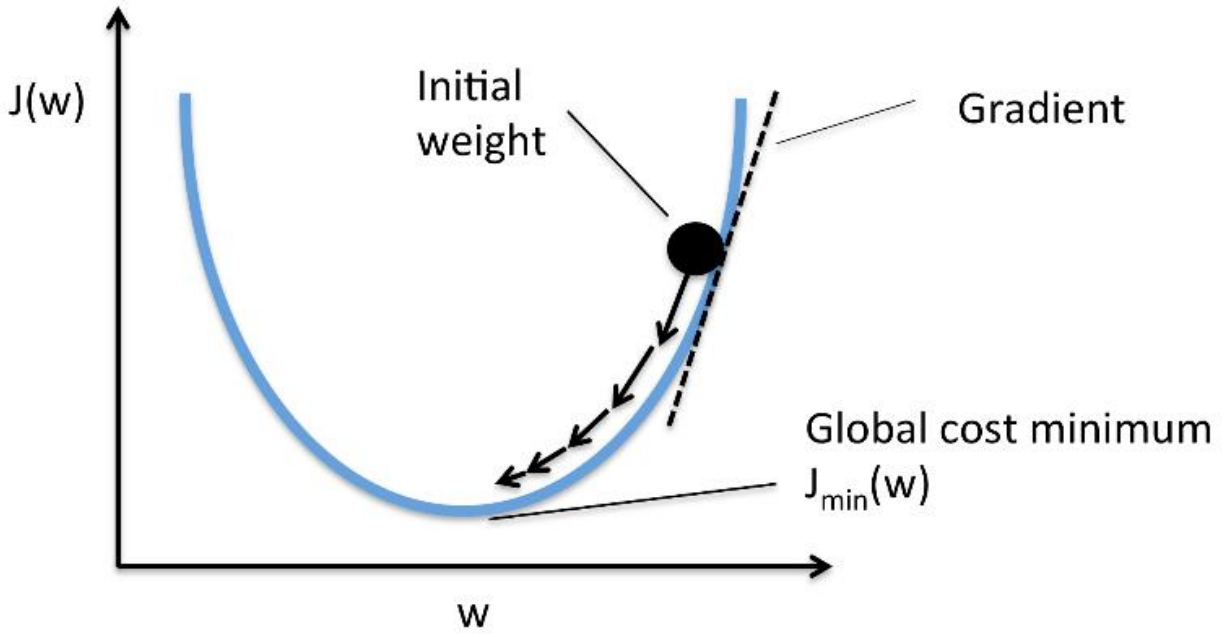
**任务B 手写体数字识别模型损失函数定义**

```
#定义交叉熵误差
def loss(logits,label):
    loss = tf.losses.sparse_softmax_cross_entropy(labels = label,logits =
logits)
    return loss
def accuracy(y_pred, y_true): #定义准确率计算方式
    correct_prediction = tf.equal(tf.argmax(y_pred, 1), y_true)
    return tf.reduce_mean(tf.cast(correct_prediction, tf.float32), axis=-1)
```

# 损失函数的优化

想象人处于山谷中，看不到路线，如何到达谷底?

答案很简单：根据脚上的感觉，感知哪个方向的地面是向下倾斜的，**朝着地面向下倾斜的方向**走大概率能走到谷底



# 损失函数的优化----------梯度下降

在一元条件下,斜率就是

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

在多元条件下生成的偏导数组成的向量就是梯度

Eg:$f(x, y, z) = ax + by + cz + d$，则梯度为$<\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}> = < a, b, c >$

- 在单变量的函数中，梯度其实就是函数的微分，代表着函数在某个给定点的切线的斜率
- 在多变量函数中，梯度是一个向量，向量有方向，梯度的方向就指出了函数在给定点的上升最快的方向

**梯度的方向是函数在给定点上升最快的方向，那么梯度的反方向就是函数在给定点下降最快的方向。所以我们只要沿着梯度的反方向一直走，就能走到局部的最低点！**

# 参数学习

- 给定训练集为$D = \{(\boldsymbol{x}^{(n)}, y^{(n)})\}_{n=1}^{N}$，将每个样本$\boldsymbol{x}^{(n)}$输入给前馈神经网络，得到网络输出为$\hat{y}^{(n)}$，其在数据集D上的结构化风险函数为：

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

- 梯度下降

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

$$= \boldsymbol{W}^{(l)} - \alpha\left(\frac{1}{N}\sum_{n=1}^{N}\left(\frac{\partial\mathcal{L}(\boldsymbol{y}^{(n)}, \hat{\boldsymbol{y}}^{(n)})}{\partial\boldsymbol{W}^{(l)}}\right) + \lambda\boldsymbol{W}^{(l)}\right),$$

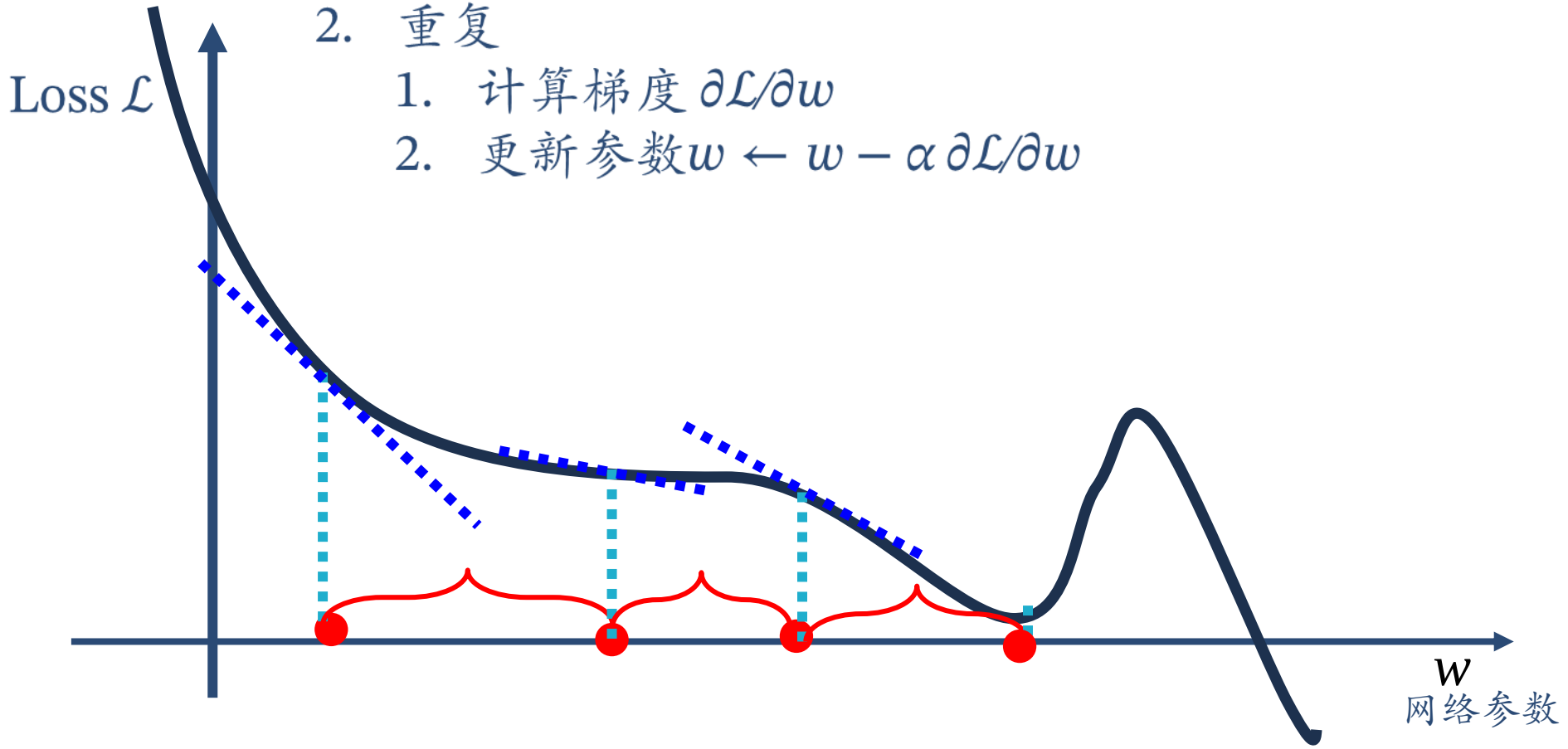
$$\boldsymbol{b}^{(l)} \leftarrow \boldsymbol{b}^{(l)} - \alpha\frac{\partial\mathcal{R}(\boldsymbol{W}, \boldsymbol{b})}{\partial\boldsymbol{b}^{(l)}}$$

$$= \boldsymbol{b}^{(l)} - \alpha\left(\frac{1}{N}\sum_{n=1}^{N}\frac{\partial\mathcal{L}(\boldsymbol{y}^{(n)}, \hat{\boldsymbol{y}}^{(n)})}{\partial\boldsymbol{b}^{(l)}}\right),$$

# 梯度下降

1. 初始化w
2. 重复
   1. 计算梯度 $\partial \mathcal{L}/\partial w$
   2. 更新参数 $w \leftarrow w - \alpha \partial \mathcal{L}/\partial w$

梯度：$\frac{\partial f(w)}{\partial w} = \lim_{\Delta w \to 0} \frac{f(w+\Delta w)}{\Delta w}$



# 损失函数的优化----------随机梯度下降

## 梯度下降弊端

$$L(w) = \frac{1}{N}\sum_{i=1}^{N} L_i(x_i, y_i, W)$$

损失函数可以计算分类器在训练样本中的每一步表现
我们可以进一步设定**数据集的总误差 = 整个训练集误差的平均值**
**实际上训练集的样本一般都很大**，比如如果使用ImageNet数据集，样本数量超过130万，一次性计算整个数据集样本的损失成本非常高。

对于整体样本的损失函数梯度值，相当于每个单项误差梯度值的总和，
如果我们计算完整个训练集的总误差再根据梯度值进行更新权重，无疑会**非常非常慢**
我们可以从一次更新多少样本划分优化方式

# 损失函数的优化

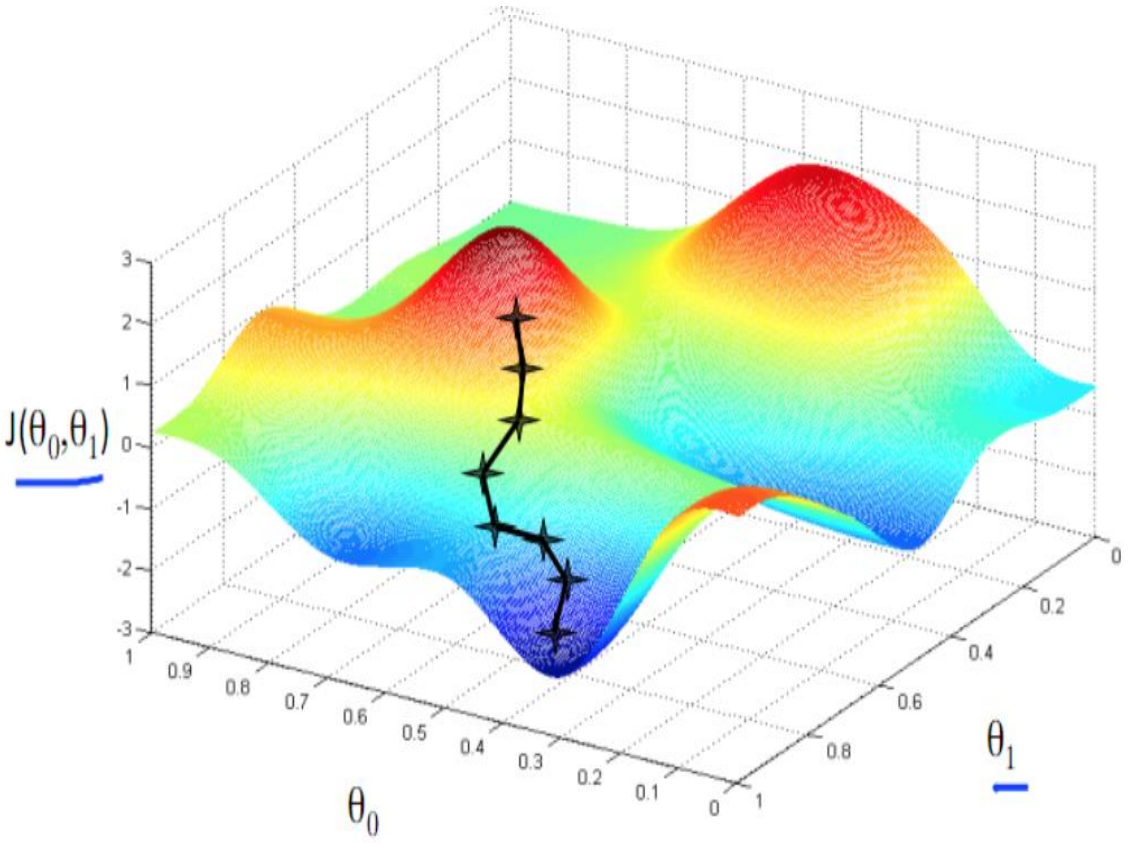
- **梯度下降**：计算完所有的样本损失后，根据梯度值更新权重。也就是所有的样本都有贡献，都参与调整权重，其计算得到的是一个标准梯度。因而理论上来说一次更新的幅度是比较大的。如果样本不多的情况下，这样收敛的速度会更快。
- **随机梯度下降**：随机也就是说用样本中的一个例子来近似所有的样本，来调整权重，因而随机梯度下降是会带来一定的问题，因为计算得到的并不是准确的一个梯度，容易陷入到局部最优解中
- **批量梯度下降**：批量的梯度下降就是一种折中的方法，他用了一些小样本来近似全部的，其本质就是随机指定一个例子替代样本不太准，那比如用30个50个样本那就比随机的要准不少了，而且批量的话还是非常可以反映样本的一个分布情况的。

**批量梯度下降是深度学习中最常见的下降类型**

# 损失函数的优化----------批量梯度下降

它并非计算整个训练集的误差和梯度值，而是在每一次迭代中，选取一小部分训练样本，也叫小批量(minibatch),按惯例这里都去2的n次幂，常用数字32、64、128。然后我们用minibatch来估算误差总和以及实际梯度。这是随机的，因为你可以把它当做对真实数值期望的一种蒙特卡洛估计

下面这个网站可以帮助你更好理解用梯度下降训练线性分类器究竟是怎么回事
http://vision.stanford.edu/teaching/cs231n-demos/linear-classify/



# 模型训练

任务A 一元回归模型训练

```
import random
#定义反向传递
def train(model,x,y,learning_rate,batch_size,epoch):
    for e in range(epoch): #次数
        tmp = list(zip(x, y)) #洗牌
        random.shuffle(tmp)
        x,y = zip(*tmp)
        for b in range(0,len(x),batch_size): #批量
            with tf.GradientTape() as tape: #梯度
                loss_value = loss(model(x[b:b + batch_size]),y[b:b+batch_size])
                dW,db = tape.gradient(loss_value,[model.w,model.b])
            #更新参数
            model.w.assign_sub(dW * learning_rate)
            model.b.assign_sub(dW * learning_rate)
        print("Training loss为"+str(loss_value.numpy()))
#学习
train(model,train_x,train_y,learning_rate=0.01,batch_size = 2,epoch=10)
#评估
test_p = model(test_x)

print("Final Test loss: %s" %loss(test_p,test_y).numpy())
```

# 模型训练

任务B 手写体数字识别模型训练

```
import random
def train(model,x,y,learning_rate,batch_size,epoch):#定义反向传递
    for e in range(epoch): #次数
        tmp = list(zip(x, y))
        random.shuffle(tmp)
        x,y = zip(*tmp)
        for b in range(0,len(x),batch_size): #批量
            with tf.GradientTape() as tape: #梯度
                loss_value = loss(model(x[b:b+batch_size]),y[b:b+batch_size])
                dW1,db1,dW2,db2,dW3,db3 = tape.gradient(loss_value,
                                          [model.W1,model.b1,model.W2,model.b2,model.W3,model.b3])
            model.W1.assign_sub(dW1 * learning_rate) #更新参数
            model.b1.assign_sub(db1 * learning_rate)
            model.W2.assign_sub(dW2 * learning_rate)
            model.b2.assign_sub(db2 * learning_rate)
            model.W3.assign_sub(dW3 * learning_rate)
            model.b3.assign_sub(db3 * learning_rate)
        print("epoch :"+str(e)," loss:"+str(loss_value.numpy())," test accuary:"+
                                                    str(accuracy(model(test_x),test_y).numpy()))
train(model,train_x,train_y,learning_rate=0.001,batch_size =128,epoch=10) #学习
```

# 如何计算梯度?

- 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

# 反向传播算法

# 矩阵微积分

- 矩阵微积分（Matrix Calculus）是多元微积分的一种表达方式，即使用矩阵和向量来表示因变量每个成分关于自变量每个成分的偏导数。
- 分母布局
  - 标量关于向量的偏导数

$$\frac{\partial y}{\partial \boldsymbol{x}} = [\frac{\partial y}{\partial x_1}, \cdots, \frac{\partial y}{\partial x_p}]^{\mathrm{T}}$$

  - 向量关于向量的偏导数

$$\frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_q}{\partial x_1} \\ \vdots & \vdots & \vdots \\ \frac{\partial y_1}{\partial x_p} & \cdots & \frac{\partial y_q}{\partial x_p} \end{bmatrix} \in \mathbb{R}^{p \times q}$$

# 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

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

# 反向传播算法

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

$$\frac{\partial \mathbf{z}^{(l)}}{\partial w_{ij}^{(l)}} = \left[\frac{\partial z_1^{(l)}}{\partial w_{ij}^{(l)}}, \cdots, \frac{\partial z_i^{(l)}}{\partial w_{ij}^{(l)}}, \cdots, \frac{\partial z_{m^{(l)}}^{(l)}}{\partial w_{ij}^{(l)}}\right]$$

$$= \left[0, \cdots, \frac{\partial(\mathbf{w}_{i:}^{(l)}\mathbf{a}^{(l-1)} + b_i^{(l)})}{\partial w_{ij}^{(l)}}, \cdots, 0\right]$$

$$= \left[0, \cdots, a_j^{(l-1)}, \cdots, 0\right]$$

$$\triangleq \mathbb{I}_i(a_j^{(l-1)}) \quad \in \mathbb{R}^{m^{(l)}},$$

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

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

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

误差项

反映了最终损失对第$l$层神经元的敏感程度

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

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

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

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

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

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

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

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

$$= \frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} \cdot \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} \cdot \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{z}^{(l+1)}}$$

$$= \mathrm{diag}(f_l'(\mathbf{z}^{(l)})) \cdot (W^{(l+1)})^{\mathrm{T}} \cdot \delta^{(l+1)}$$

$$= f_l'(\mathbf{z}^{(l)}) \odot \big((W^{(l+1)})^{\mathrm{T}}\delta^{(l+1)}\big),$$

# 反向传播算法

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

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

# 反向传播算法

在计算出每一层的误差项之后，我们就可以得到每一层参数的梯度. 因此，使用误差反向传播算法的前馈神经网络训练过程可以分为以下三步：

（1） 前馈计算每一层的净输入 $\boldsymbol{z}^{(l)}$ 和激活值 $\boldsymbol{a}^{(l)}$，直到最后一层；

（2） 反向传播计算每一层的误差项 $\delta^{(l)}$；

（3） 计算每一层参数的偏导数，并更新参数.

**算法 4.1　使用反向传播算法的随机梯度下降训练过程**

**输入:** 训练集 $\mathcal{D}=\{(\boldsymbol{x}^{(n)},y^{(n)})\}_{n=1}^{N}$, 验证集 $\mathcal{V}$, 学习率 $\alpha$, 正则化系数 $\lambda$, 网络层数 $L$, 神经元数量 $M_l$, $1\leq l\leq L$.

随机初始化 $\boldsymbol{W},\boldsymbol{b}$;

**repeat**

　对训练集 $\mathcal{D}$ 中的样本随机重排序;

　**for** $n=1\cdots N$ **do**

　　从训练集 $\mathcal{D}$ 中选取样本 $(\boldsymbol{x}^{(n)},y^{(n)})$;

　　前馈计算每一层的净输入 $\boldsymbol{z}^{(l)}$ 和激活值 $\boldsymbol{a}^{(l)}$, 直到最后一层;

　　反向传播计算每一层的误差 $\delta^{(l)}$; // 公式 (4.63)

　　// 计算每一层参数的导数

　　$\forall l,\quad \frac{\partial \mathcal{L}(\boldsymbol{y}^{(n)},\hat{\boldsymbol{y}}^{(n)})}{\partial \boldsymbol{W}^{(l)}}=\delta^{(l)}(\boldsymbol{a}^{(l-1)})^{\mathrm{T}}$; // 公式 (4.68)

　　$\forall l,\quad \frac{\partial \mathcal{L}(\boldsymbol{y}^{(n)},\hat{\boldsymbol{y}}^{(n)})}{\partial \boldsymbol{b}^{(l)}}=\delta^{(l)}$; // 公式 (4.69)

　　// 更新参数

　　$\boldsymbol{W}^{(l)}\leftarrow \boldsymbol{W}^{(l)}-\alpha(\delta^{(l)}(\boldsymbol{a}^{(l-1)})^{\mathrm{T}}+\lambda \boldsymbol{W}^{(l)})$;

　　$\boldsymbol{b}^{(l)}\leftarrow \boldsymbol{b}^{(l)}-\alpha\delta^{(l)}$;

　**end**

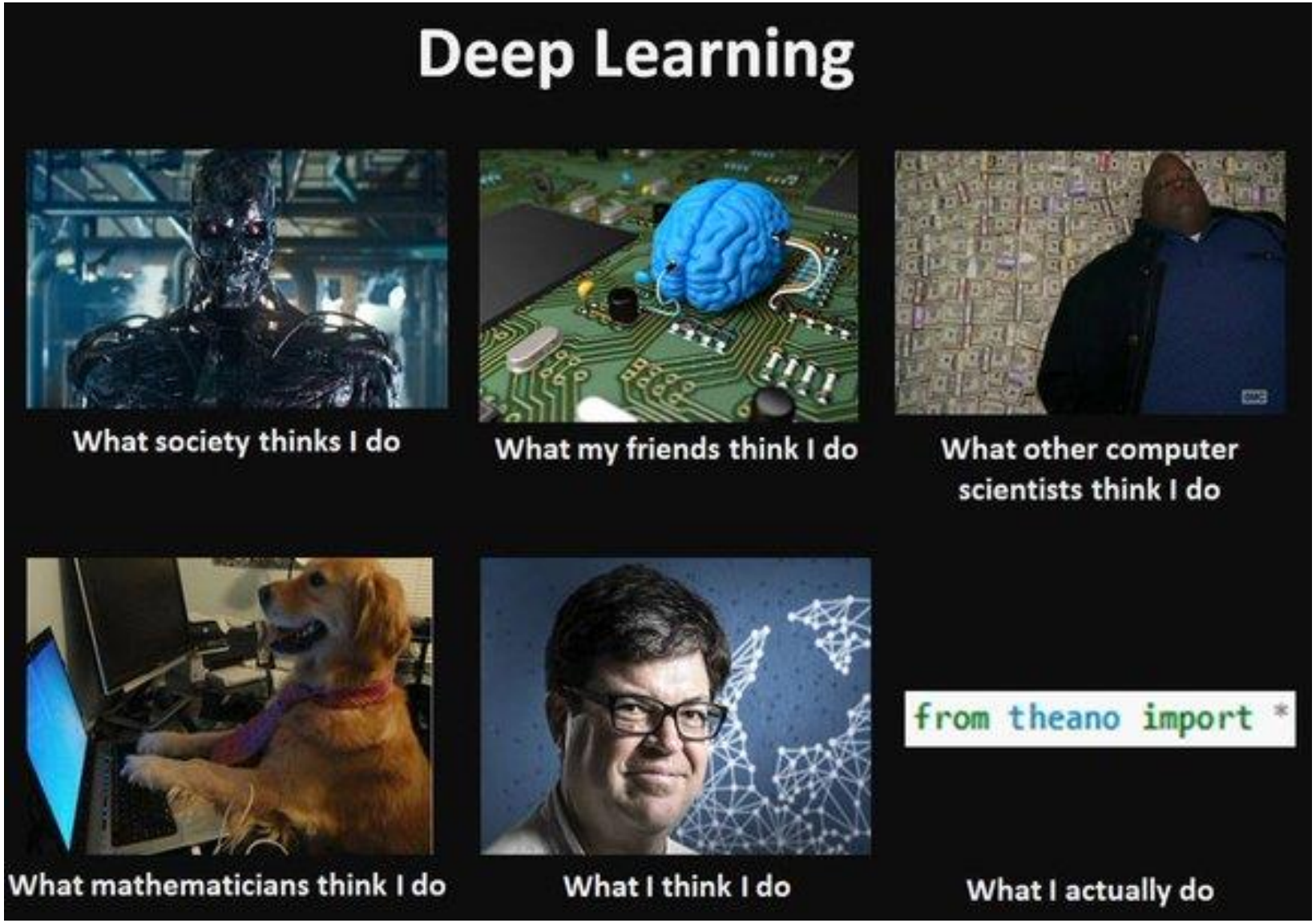
**until** 神经网络模型在验证集 $\mathcal{V}$ 上的错误率不再下降;

**输出:** $\boldsymbol{W},\boldsymbol{b}$

# 反向传播算法 (自动微分的反向模式)

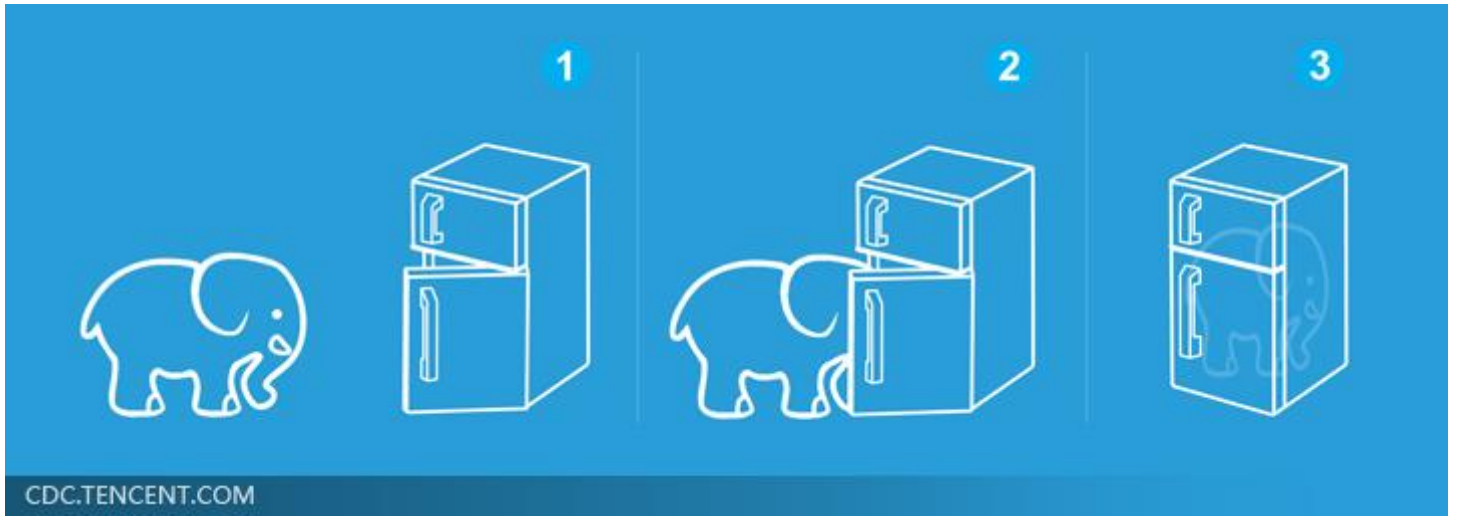
- 前馈神经网络的训练过程可以分为以下三步
  - 前向计算每一层的状态和激活值，直到最后一层
  - 反向计算每一层的参数的偏导数
  - 更新参数

# 如何实现?


Deep Learning
What society thinks I do
What my friends think I do
What other computer scientists think I do
What mathematicians think I do
What I think I do
from theano import *
What I actually do

# 深度学习的三个步骤


Deep Learning is so simple ……



# Getting started: 30 seconds to Keras

```
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras.optimizers import SGD

model = Sequential()
model.add(Dense(output_dim=64, input_dim=100))
model.add(Activation("relu"))
model.add(Dense(output_dim=10))
model.add(Activation("softmax"))

model.compile(loss='categorical_crossentropy',
   optimizer='sgd', metrics=['accuracy'])

model.fit(X_train, Y_train, nb_epoch=5, batch_size=32)

loss = model.evaluate(X_test, Y_test, batch_size=32)
```

# 优化问题

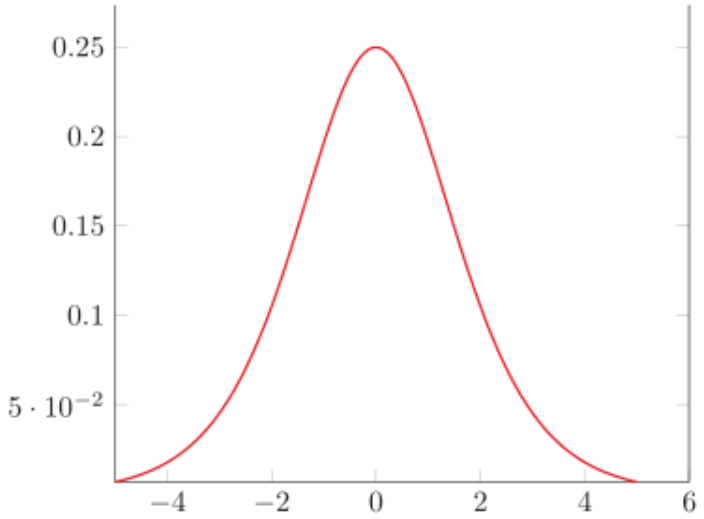
# 优化问题

- 难点
  - 参数过多，影响训练
  - 非凸优化问题：即存在局部最优而非全局最优解，影响迭代
  - 梯度消失问题，下层参数比较难调
  - 参数解释起来比较困难
- 需求
  - 计算资源要大
  - 数据要多
  - 算法效率要好：即收敛快

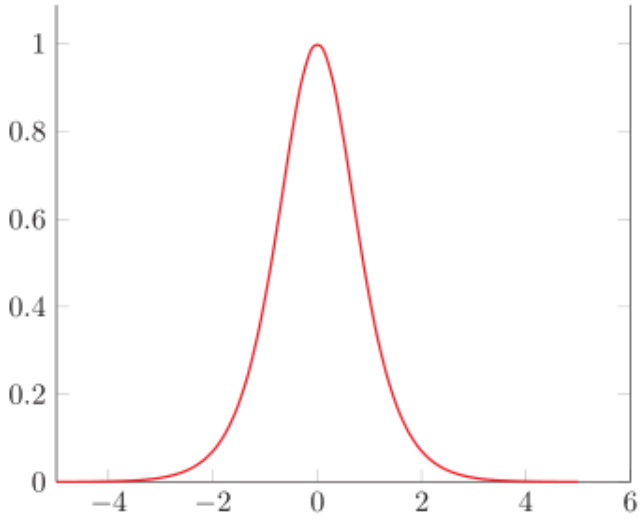
# 优化问题

- 梯度消失问题（Vanishing Gradient Problem）

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




(a) logistic 函数的导数



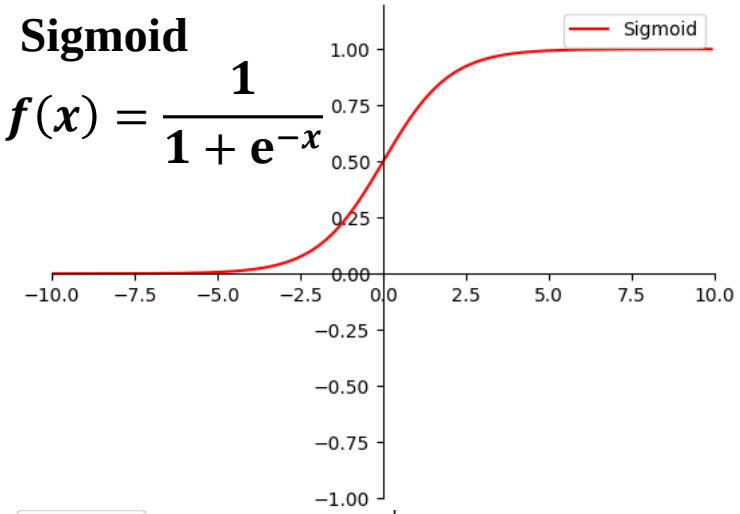

(b) tanh 函数的导数

# 常见激活函数



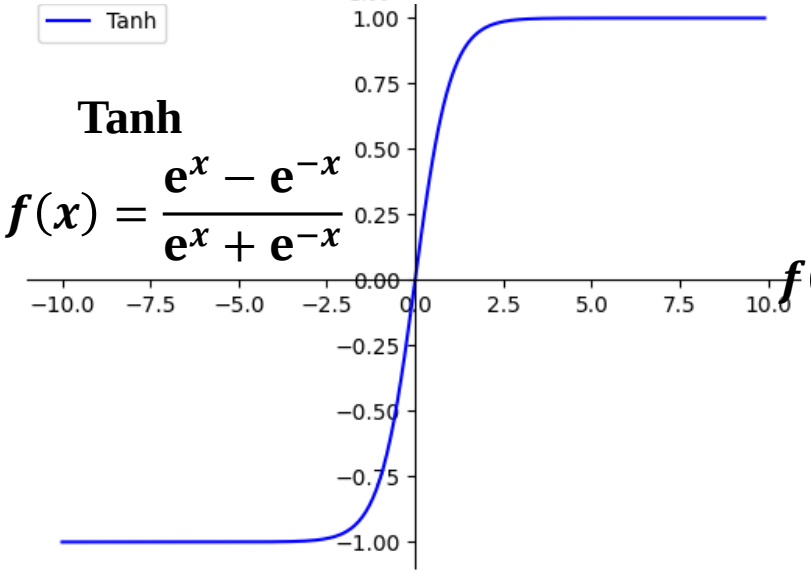
**Sigmoid**

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



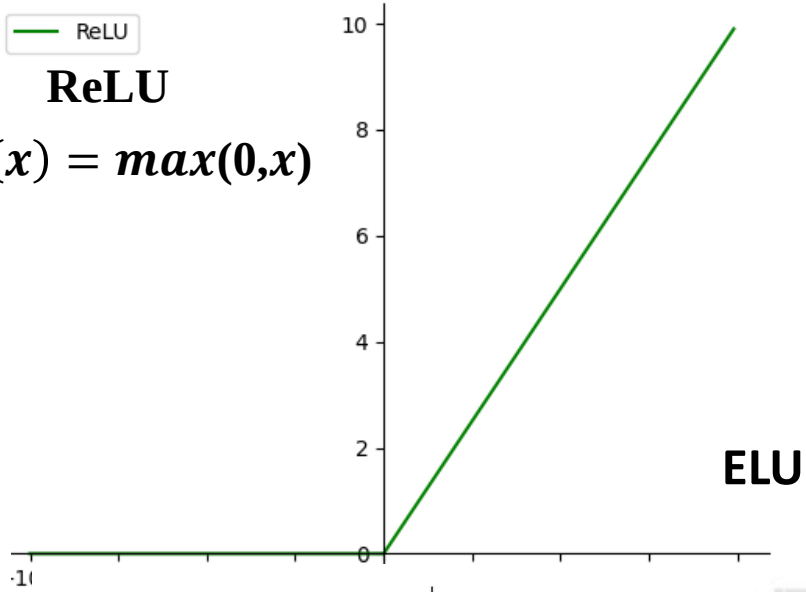

**Tanh**

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



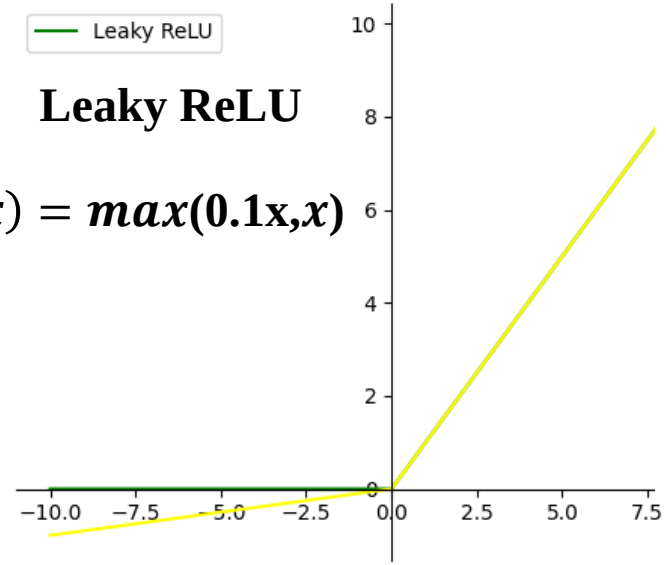

**ReLU**

$$f(x) = max(0,x)$$




**Leaky ReLU**

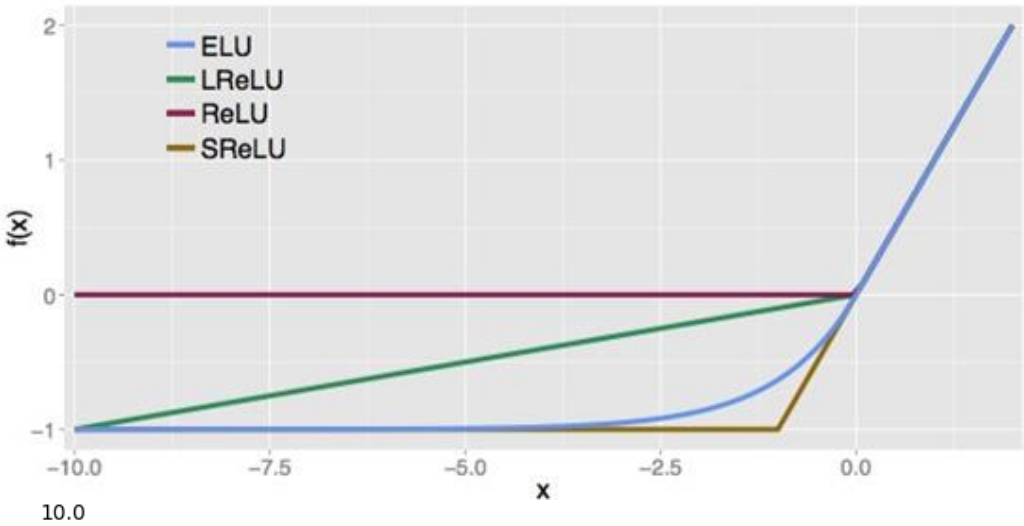
$$f(x) = max(0.1x,x)$$




**Maxout** $max(\omega_1^T x + b1, \omega_2^T x + b2)$

**ELU**

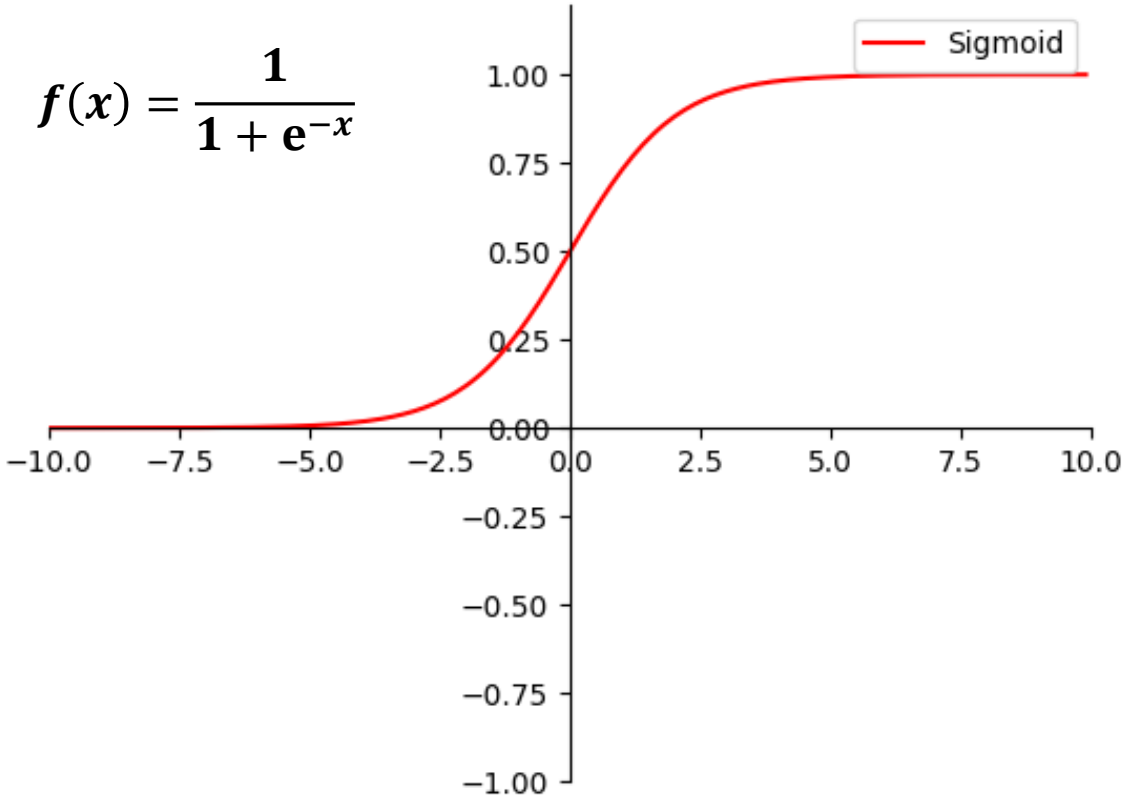
$$f(x) = \begin{cases} x & if\ x > 0 \\ \propto (\exp(x) - 1 & if x \leq 0 \end{cases}$$



# 激活函数------Sigmoid



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




**Sigmoid**

- 将输入值压缩到(0,1)区间内
- 曾经非常流行，因为sigmoid在某种意义上被看做是一种神经元放电率(神经元处于激活状态放电的概率)

**3个缺点**：

1、当输入处于区间$(-\infty,-10)$或$(10,+\infty)$，求出的y处于sigmoid的平滑区域，这些区域会使**梯度消失**，不利于反向传播

2、输出全为正数，**梯度更新效率非常低。简单来说，由于输出同号，在梯度下降过程每个参数只能选择向同一方向更新（即同时增大或者同时减小）**

3、exp()的**计算代价**有点高

# 激活函数------Sigmoid



## 激活函数输出均为非负数有什么坏处?

随意圈出神经网络中的一个单元如图所示,
该单元的左半部分是对输入的线性变换即

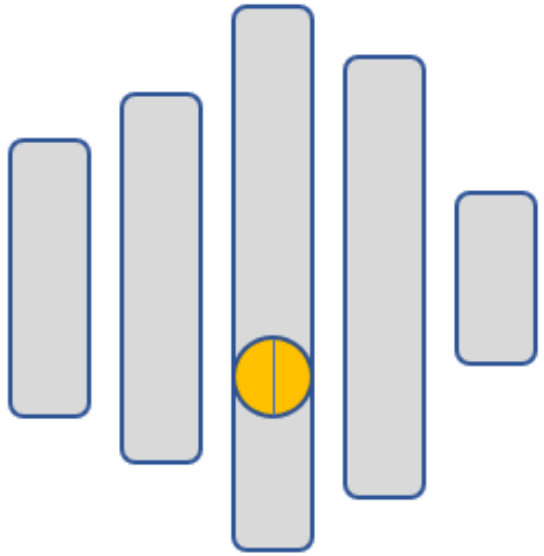
$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

若激活函数为$f(x)$
则根据链式求导法则

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial z}\frac{\partial z}{\partial w_i} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial z}x_i$$

$x_i$为激活函数的输出，如果为sigmoid激活函数，则其必为非负数

即$\frac{\partial L}{\partial w_1}$，$\frac{\partial L}{\partial w_2}$，…，$\frac{\partial L}{\partial w_n}$同号



# 激活函数------Sigmoid



激活函数输出均为非负数有什么坏处？

参数更新公式如下，显然，我们只能**同时增加参数值，或者同时减小参数值**

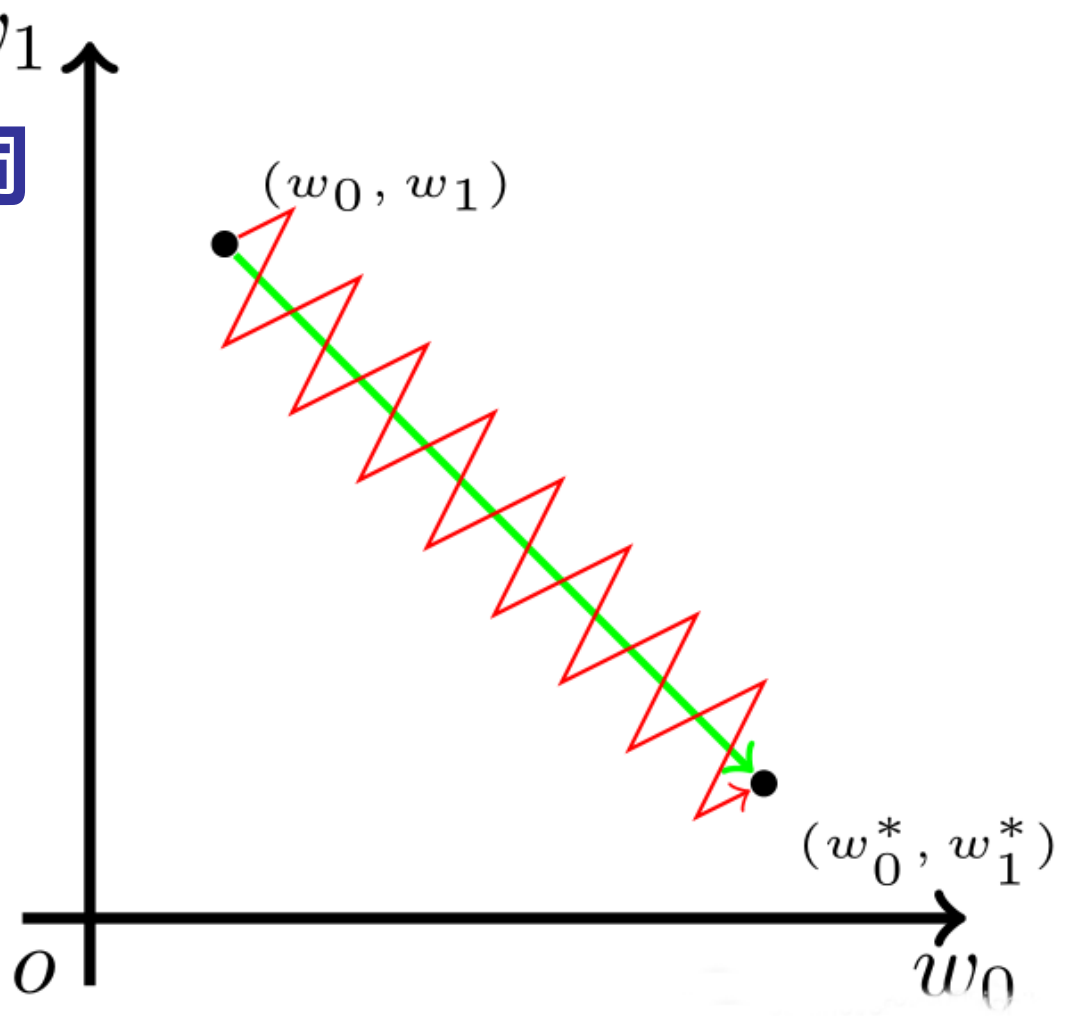
$$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1}$$

$$w_2 \leftarrow w_2 - \eta \frac{\partial L}{\partial w_2}$$

$$\vdots$$

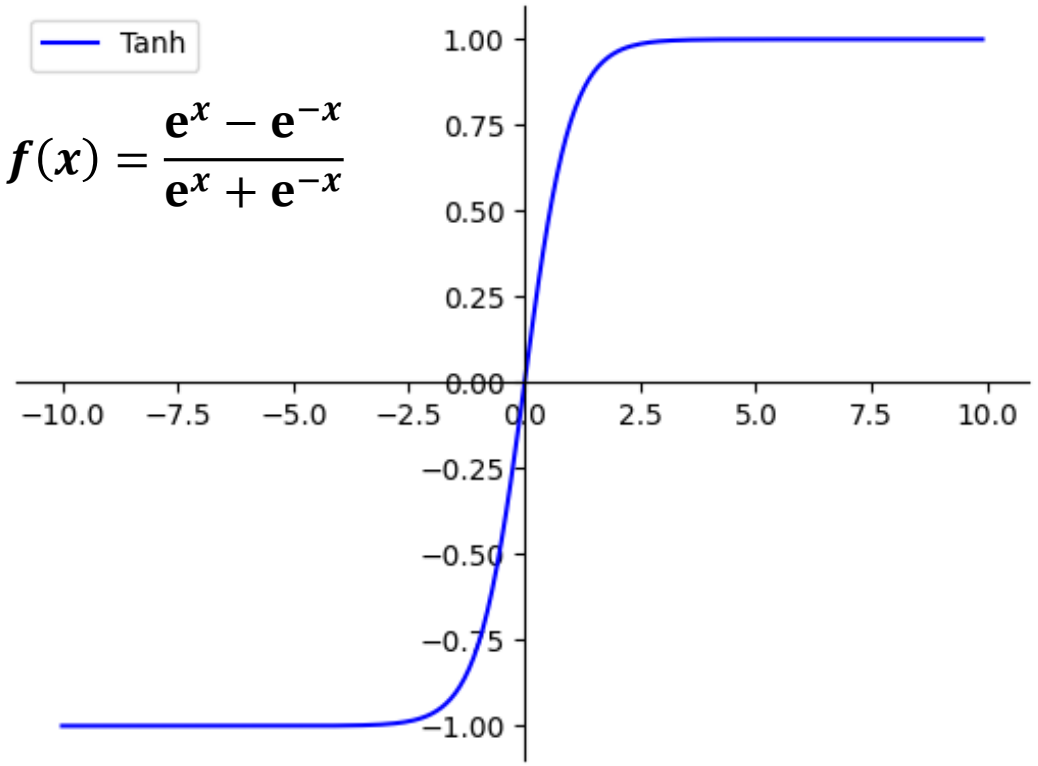
$$w_n \leftarrow w_n - \eta \frac{\partial L}{\partial w_n}$$

只能像图中红线一样以z字形下降逼近最优解，这无疑会增加迭代次数，降低收敛次数



# 激活函数------Tanh



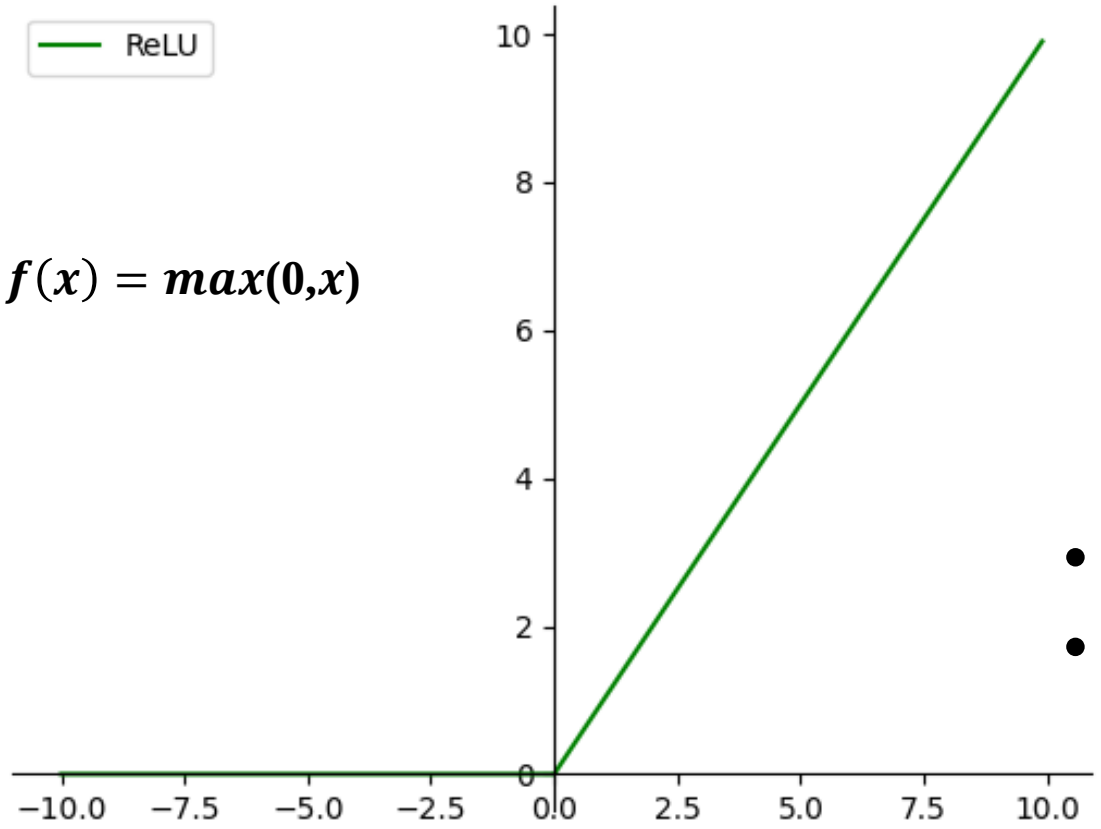



Tanh

- 将输入值压缩到(-1,1)区间内
- 输出以0为中心
- 当输入处于平滑区间时，仍然会出现**梯度消失**的情况

# 激活函数------ReLU






$f(x) = max(0,x)$

**ReLU**

- 在输入为正数的区间，都**不存在饱和**现象(梯度趋向于0)
- **计算成本很小**
- 同等情况下的**收敛速度**，大概是sigmoid/tanh的6倍

- 输出均为非负数，**梯度更新效率非常低**
- 当输入异常或学习率过高时，很可能发生**神经元死亡**

# 激活函数------ReLU



## 神经元死亡原因

$$w_i = w_i - lr\frac{\partial L}{\partial w_i} = w_i - lr\frac{\partial L}{\partial f}\frac{\partial f}{\partial z}x_i$$

权重更新的方式如上所示，假如输入异常大，或者学习率调高了就会导致权重一下子更新过多，当$w_i < lr\frac{\partial L}{\partial w_i}$时，新的权重$w_i < 0$，这样很有可能导致$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n < 0$

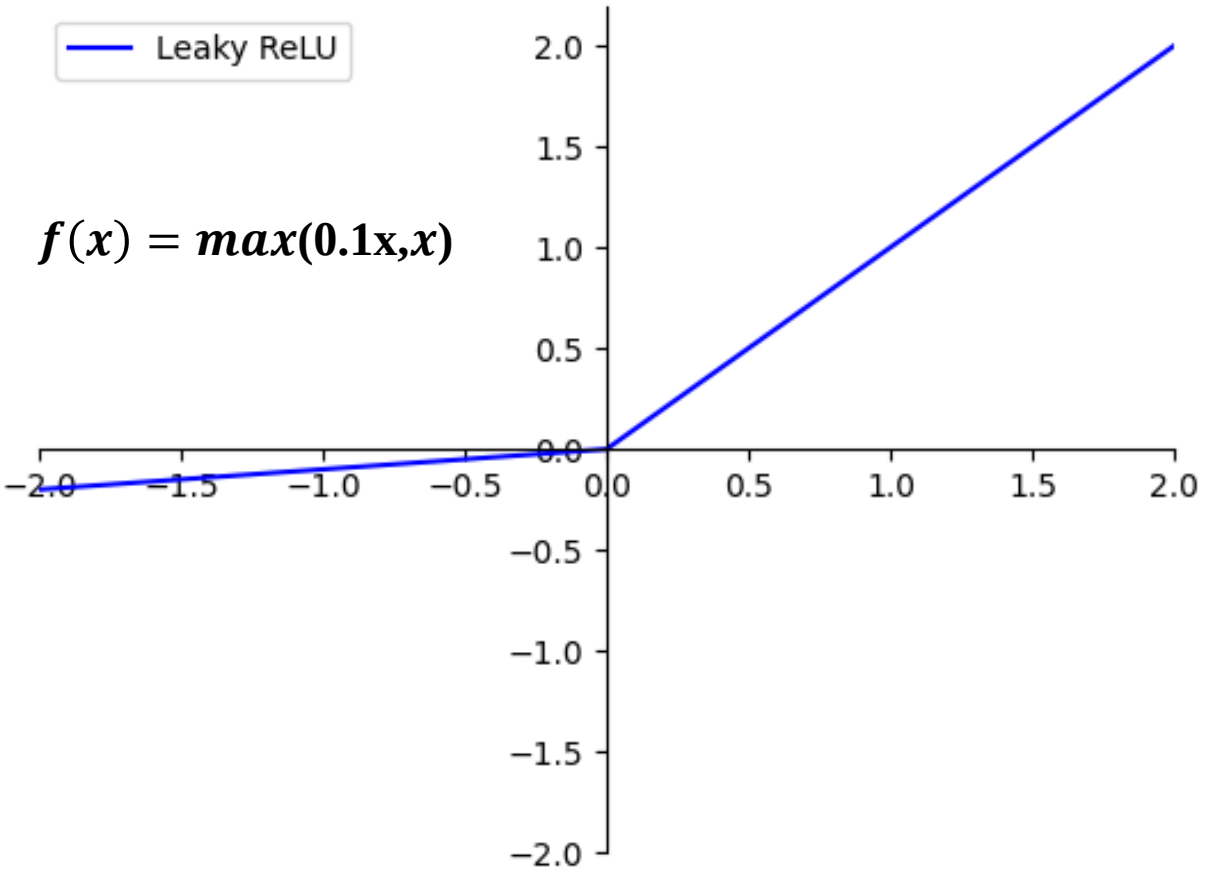
ReLU = max(0,x)，当z <0时，ReLU = 0为常数，此时权重更新公式为

$$w_i = w_i - lr\frac{\partial L}{\partial f}\frac{\partial f}{\partial z}x_i = w_i - lr\frac{\partial L}{\partial f} \times 0 \times x_i = w_i$$

**即权重不会再发生改变，因此称为神经元死亡**

# 激活函数------Leaky ReLU






**Leaky ReLU**

- 无论在正区间还是负区间都**不会饱和**
- 计算依然十分**简单且高效**
- 同等情况下的**收敛速度**，大概是sigmoid/tanh的6倍
- **不会出现神经元死亡**的情况

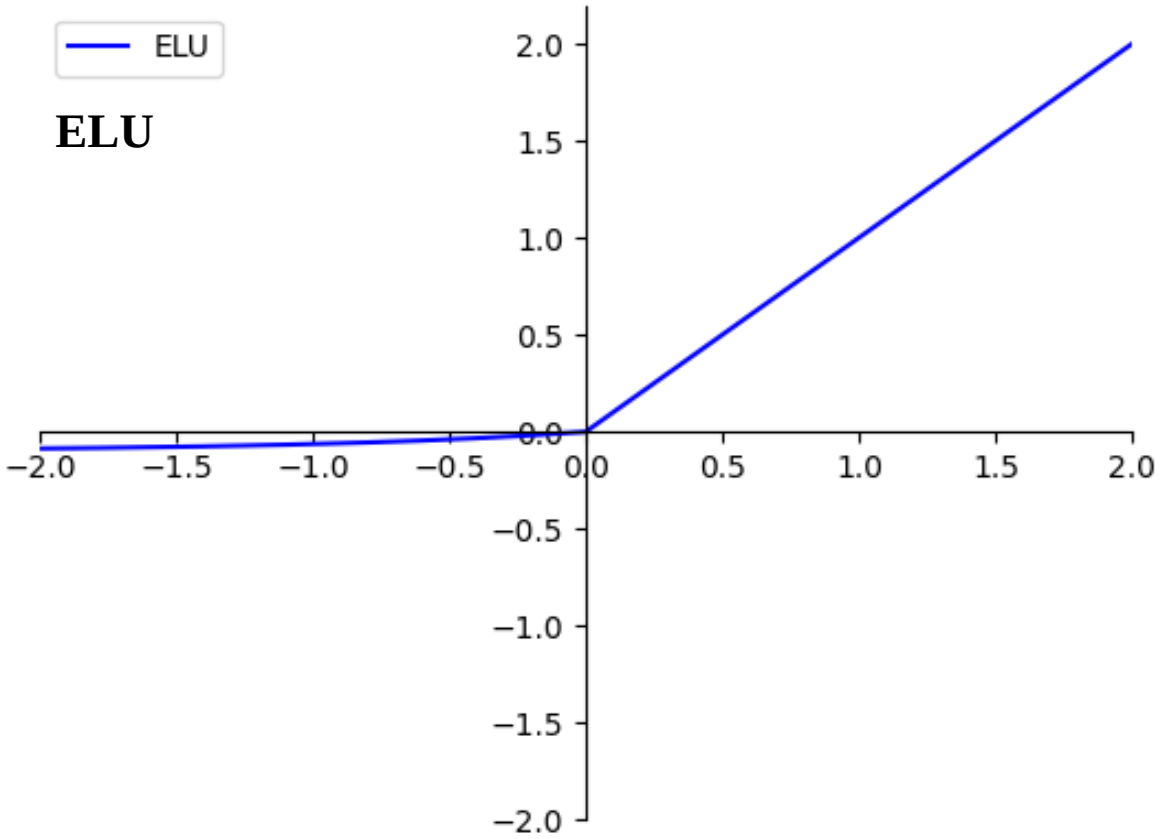
**Parametric Rectifier(PReLU)**

$$f(x) = \max(\propto x, x)$$

Leaky ReLU的一般化，alph可以当做一个可以反向传播和学习的参数

# 激活函数------ELU






$$f(x) = \begin{cases} x & if\ x > 0 \\ \propto (\exp(x) - 1) & if\ x \leq 0 \end{cases}$$

- 继承了ReLU所有的优点
- 输出更接近均值为0的分布
- 相比于LeakyReLU，它在负区间更容易出现饱和

## SoftMax激活函数

$$Softmax(x_1) = \frac{\exp(x_1)}{\sum_{c=1}^{C} \exp(x_c)}$$

- 常用于多分类问题之中,配合交叉熵计算损失,在交叉熵损失中出现过，此处不再赘述

# 课后练习

- 编程练习1
  - 使用Numpy实现前馈神经网络
  - chap4_ simple neural network

https://github.com/nndl/exercise/tree/master/chap4_%20simple%20neural%20network

- 编程练习2*
  - 理论和实验证明，一个两层的ReLU网络可以模拟任何有界闭集函数
  - chap4_ simple neural network

https://github.com/nndl/exercise/tree/master/chap4_ simple neural network

谢 谢!