神经网络与深度学习

https://nndl.github.io/

前馈神经网络

高 飞 Fei Gao

gaofei@hdu.edu.cn

神经网络

神经网络

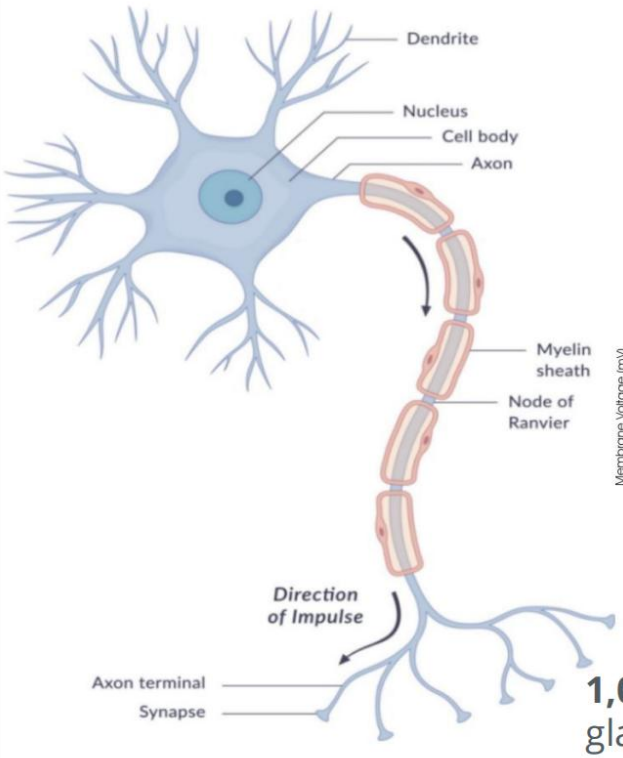
- **分布式并行处理(Parallel Distributed Processing, PDP)网络**
  - 神经网络最早是作为一种主要的连接主义模型。
  - 20世纪80年代后期，最流行的一种连接主义模型
  - 其有3个主要特性：
    - （1）信息表示是分布式的（非局部的）；
    - （2）记忆和知识是存储在单元之间的连接上；
    - （3）通过逐渐改变单元之间的连接强度来学习新的知识。

- **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。**

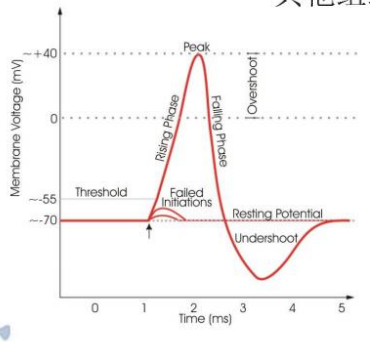
生物神经元

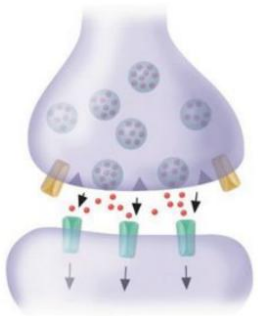
Learning in Nature how does the brain work?

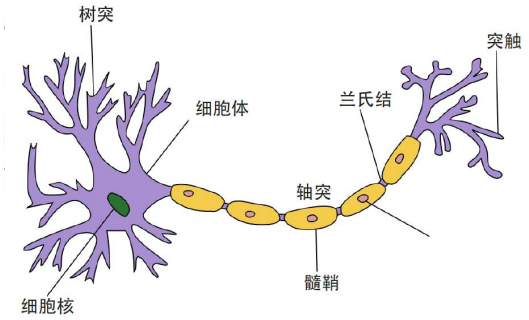
1,000's of inputs (other neurons, sensory



- 细胞体（Soma）中的神经细胞膜上有各种受体和离子通道，胞膜的受体可与相应的化学物质神经递质结合，引起离子通透性及膜内外电位差发生改变，产生相应的生理活动：兴奋或抑制。
- 细胞突起是由细胞体延伸出来的细长部分，又可分为树突和轴突。
  - 树突（Dendrite）可以接受刺激并将兴奋传入细胞体。每个神经元可以有一或多个树突。
  - 轴突 (Axons) 可以把自身的兴奋状态从胞体传送到另一个神经元或其他组织。每个神经元只有一个轴突。



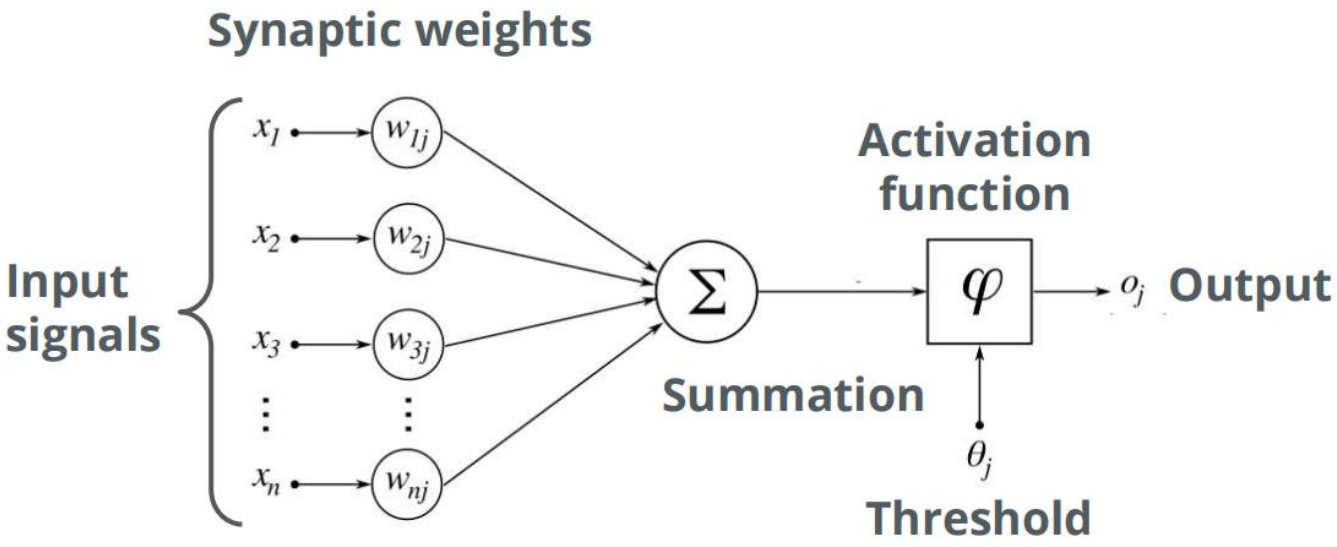




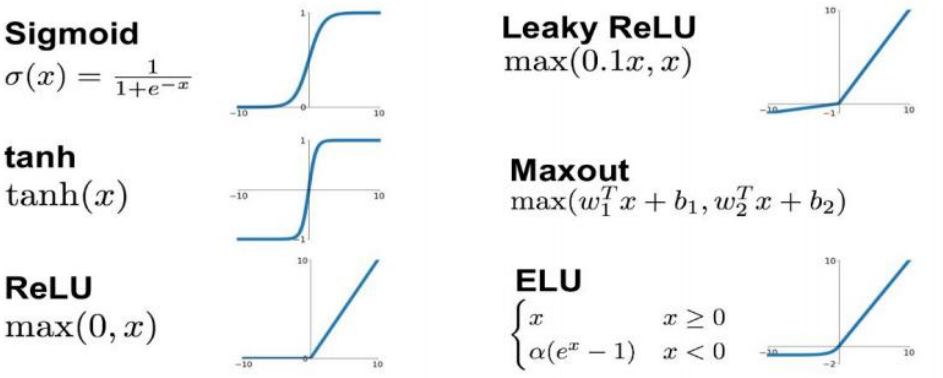
1,000's of output targets (e.g. other neurons, muscle cells, gland cells, blood vessels to release hormones...)

Figure by wetcake (left) and Andrej Kral (right)

人工神经元



$$z = \sum_{i=1}^{d} w_i x_i + b$$
$$= \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x} + b,$$



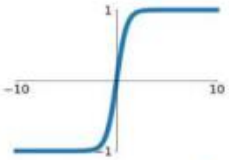
$$a = f(z)$$

激活函数的性质

- **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。

tanh

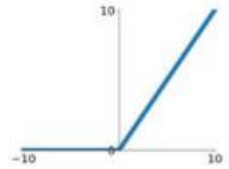
$\tanh(x)$



- **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。

ReLU

$\max(0, x)$



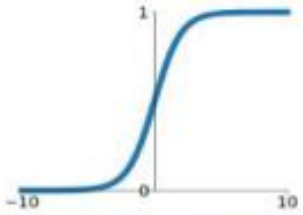
- **激活函数的导函数的值域要在一个合适的区间内**
  - 不能太大也不能太小，否则会影响训练的效率和稳定性。

- **单调递增**
  - ???

常见激活函数

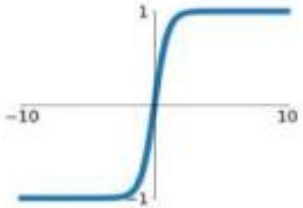
Sigmoid

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



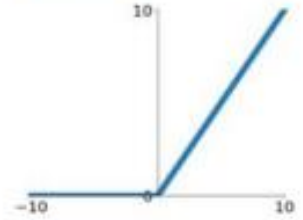
tanh

$$\tanh(x)$$



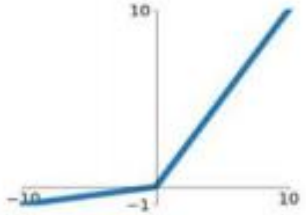
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

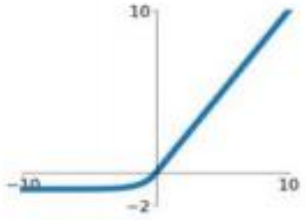


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

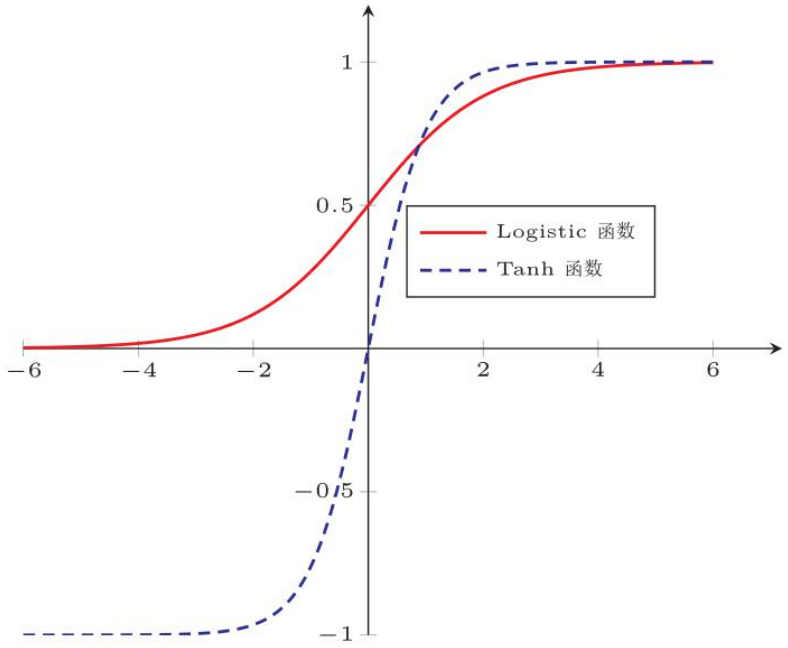


常见激活函数

- **Logistic & Tanh**

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

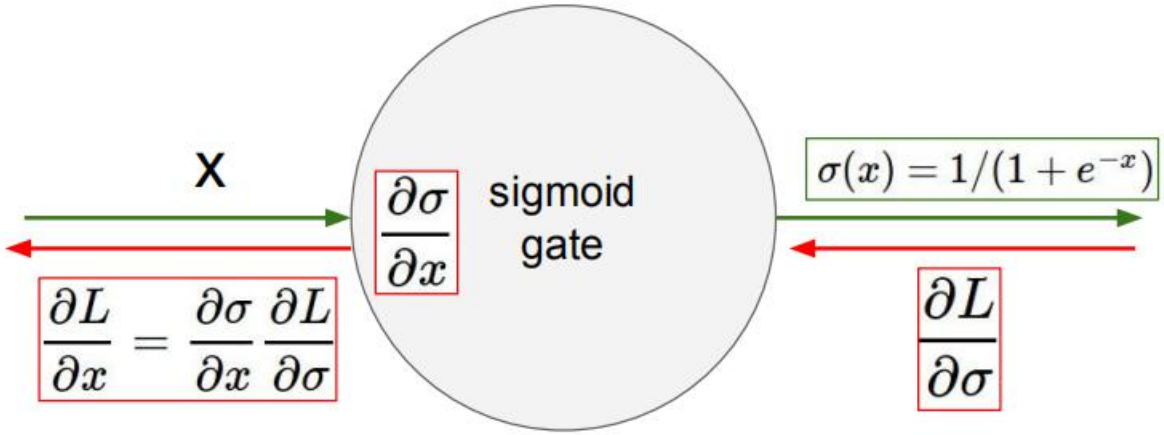
$$\tanh(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$$

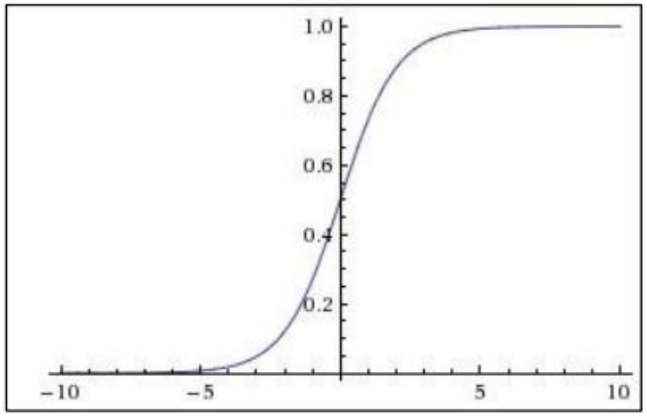


非零中心化的输出会使得其后一层的神经元的输入发生偏置偏移（bias shift），并进一步使得梯度下降的收敛速度变慢。

- **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

常见激活函数





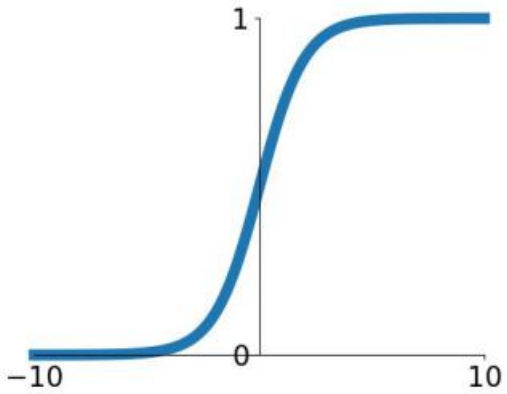
What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

激活函数	函数	导数
Logistic 函数	$f(x) = \frac{1}{1+\exp(-x)}$	$f'(x) = f(x)\big(1 - f(x)\big)$

常见激活函数

Activation Functions

$$\sigma(x) = 1/(1+e^{-x})$$



Sigmoid

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

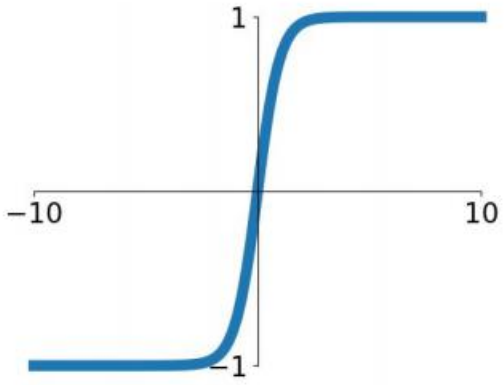
3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

激活函数	函数	导数
Logistic 函数	$f(x) = \frac{1}{1+\exp(-x)}$	$f'(x) = f(x)\big(1 - f(x)\big)$

常见激活函数

Activation Functions



tanh(x)

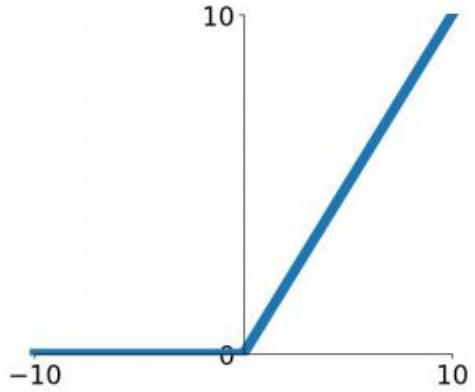
- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

[LeCun et al., 1991]

Tanh 函数 $f(x) = \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)}$ $f'(x) = 1 - f(x)^2$

常见激活函数

Activation Functions



ReLU
(Rectified Linear Unit)

- Computes **f(x) = max(0,x)**

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

[Krizhevsky et al., 2012]

ReLU $f(x) = \max(0, x)$ $f'(x) = I(x > 0)$

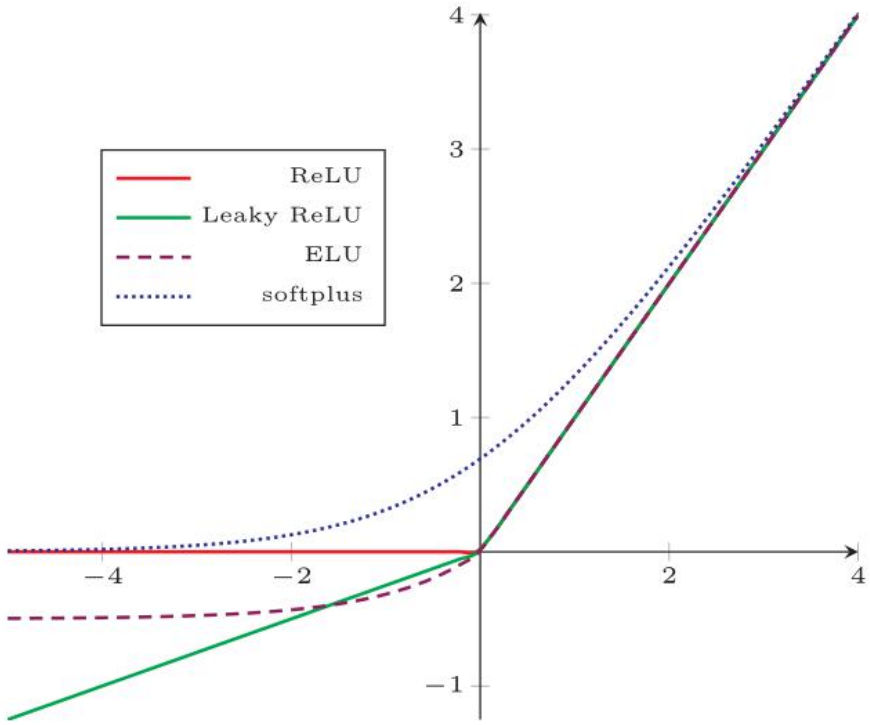
常见激活函数

- **ReLU系列**

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

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

$$f'(x) = I(x > 0)$$

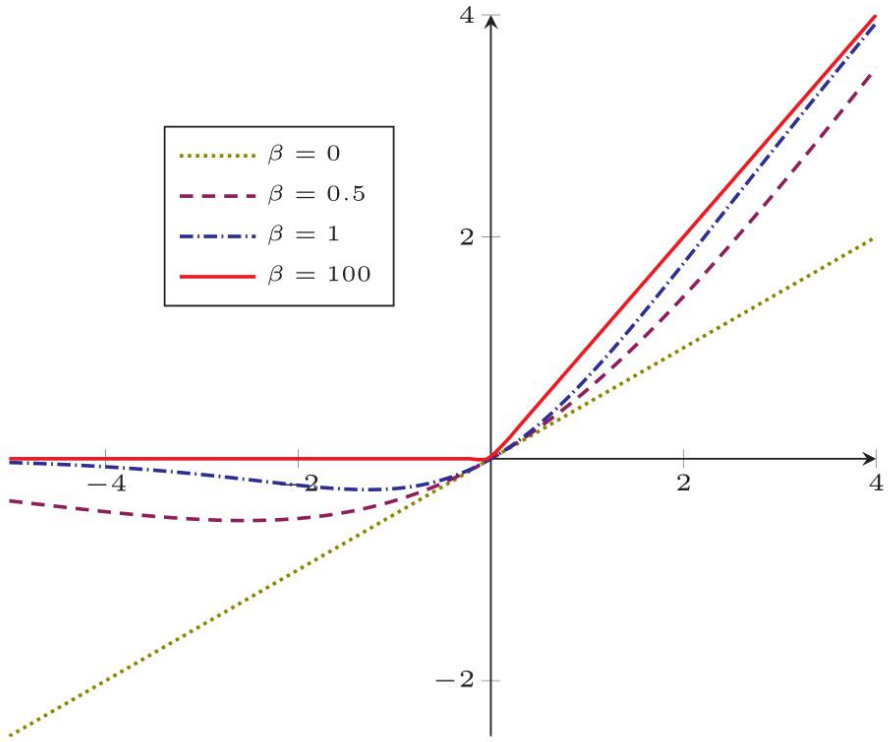


- 计算上更加高效。
- 生物上的解释性
  - 单侧抑制、宽兴奋边界
- 在一定程度上缓解梯度消失问题
- 死亡ReLU问题 (Dying ReLU Problem)

常见激活函数

- Swish函数

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



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

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：**

① 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。

② 网络的拓扑结构

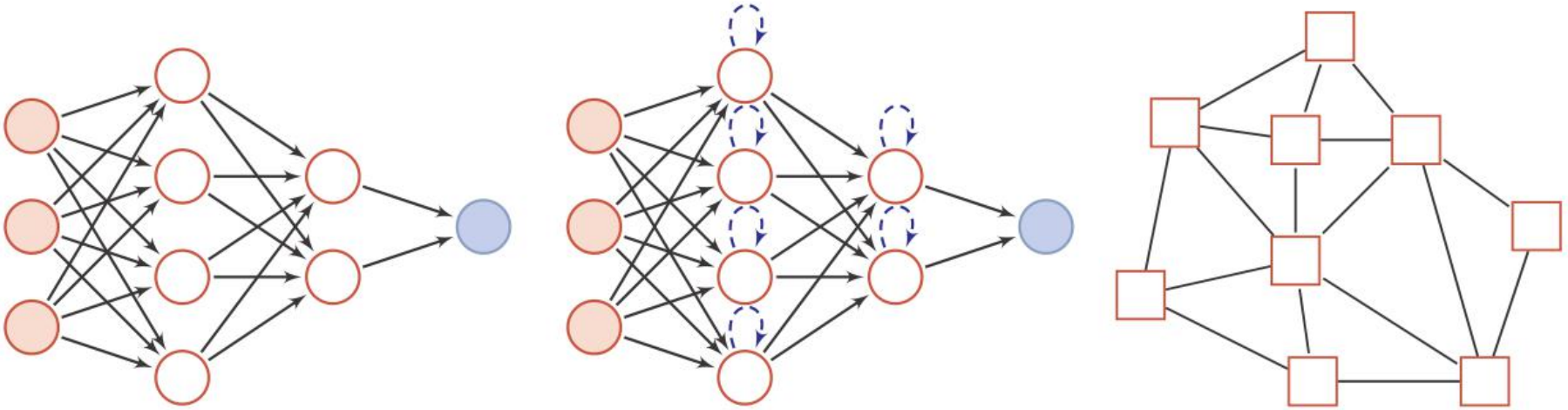
- 不同神经元之间的连接关系。

③ 学习算法

- 通过训练数据来学习神经网络的参数。

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。**



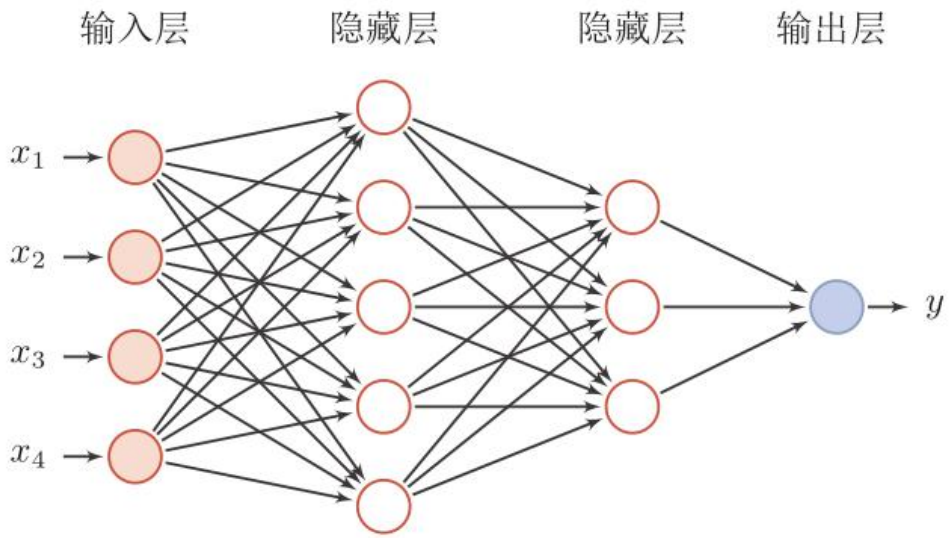
(a) 前馈网络　(b) 记忆网络　(c) 图网络

圆形节点表示一个神经元，方形节点表示一组神经元。

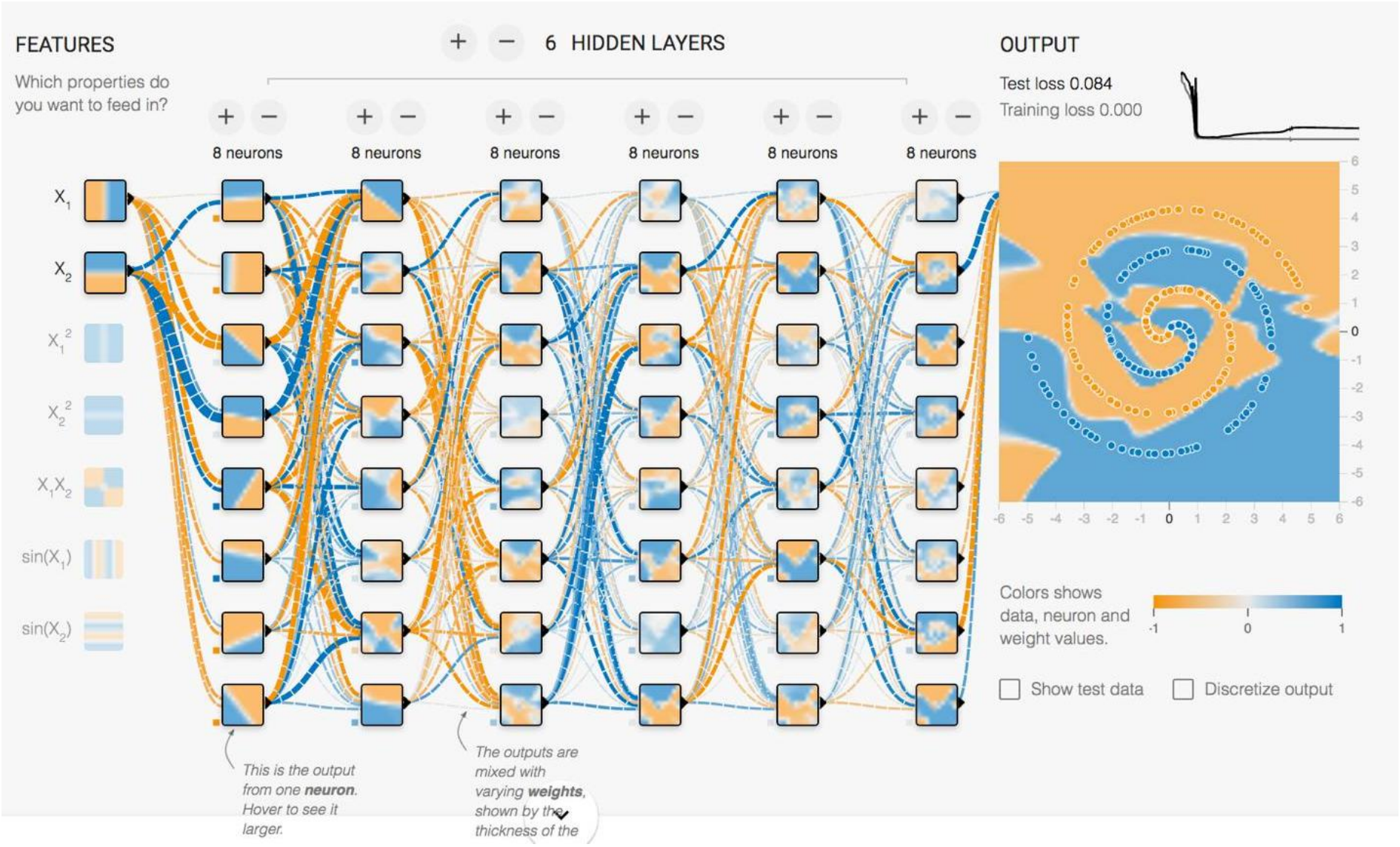
前馈神经网络

网络结构

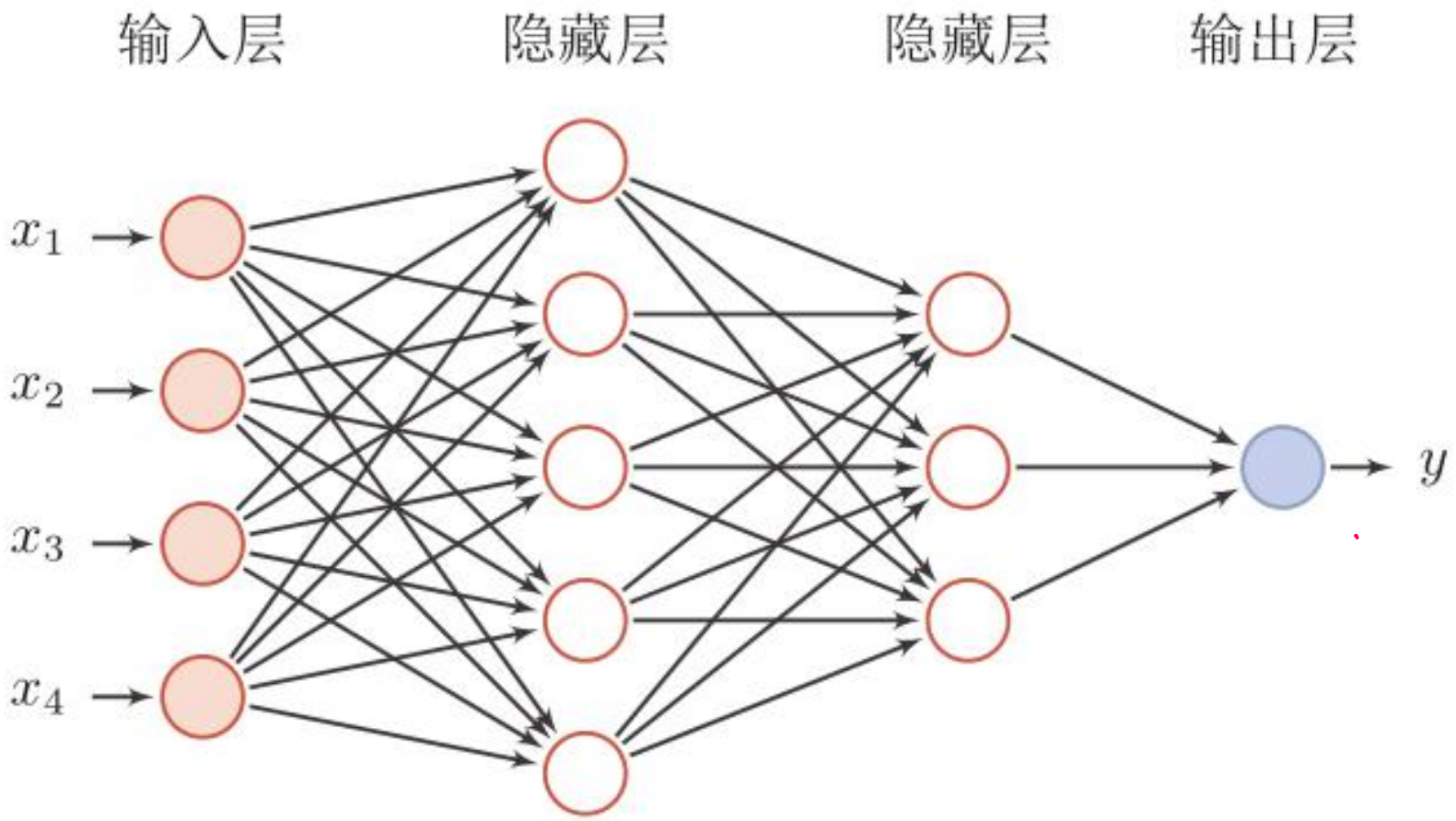
- **前馈神经网络（全连接神经网络、多层感知器）**
  - 各神经元分别属于不同的层，层内无连接。
  - 相邻两层之间的神经元全部两两连接。
  - 整个网络中无反馈，信号从输入层向输出层单向传播，可用一个有向无环图表示。



网络结构

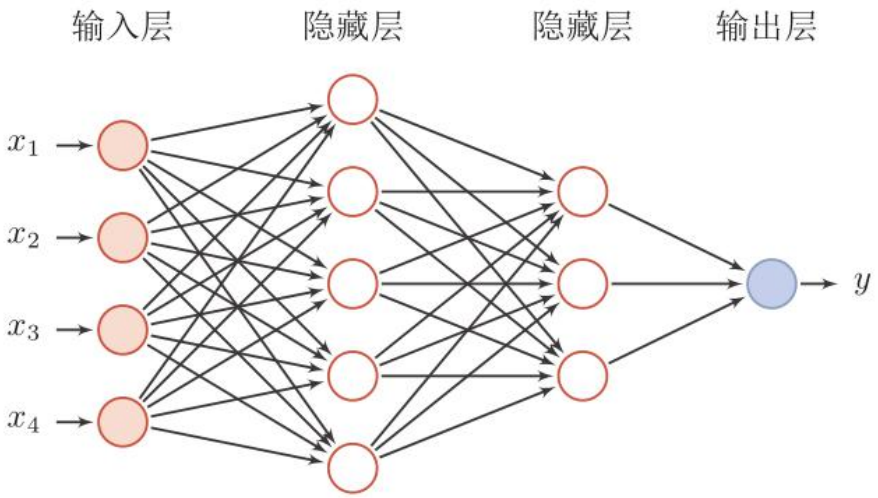

FEATURES
Which properties do you want to feed in?
6 HIDDEN LAYERS
8 neurons
8 neurons
8 neurons
8 neurons
8 neurons
8 neurons
X1
X2
X1²
X2²
X1X2
sin(X1)
sin(X2)
OUTPUT
Test loss 0.084
Training loss 0.000
Colors shows data, neuron and weight values.
-1
0
1
Show test data
Discretize output
This is the output from one neuron. Hover to see it larger.
The outputs are mixed with varying weights, shown by the thickness of the

信息传递过程



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：



- L：表示神经网络的层数；
- n^l：表示第 l 层神经元的个数；
- $f_l(\cdot)$：表示 l 层神经元的激活函数；
- $W^{(l)} \in \mathbb{R}^{n^l \times n^{l-1}}$：表示 $l-1$ 层到第 l 层的权重矩阵；
- $\mathbf{b}^{(l)} \in \mathbb{R}^{n^l}$：表示 $l-1$ 层到第 l 层的偏置；
- $\mathbf{z}^{(l)} \in \mathbb{R}^{n^l}$：表示 l 层神经元的净输入（净活性值）；
- $\mathbf{a}^{(l)} \in \mathbb{R}^{n^l}$：表示 l 层神经元的输出（活性值）。

前馈网络

- 前馈神经网络通过下面公式进行信息传播。

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

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

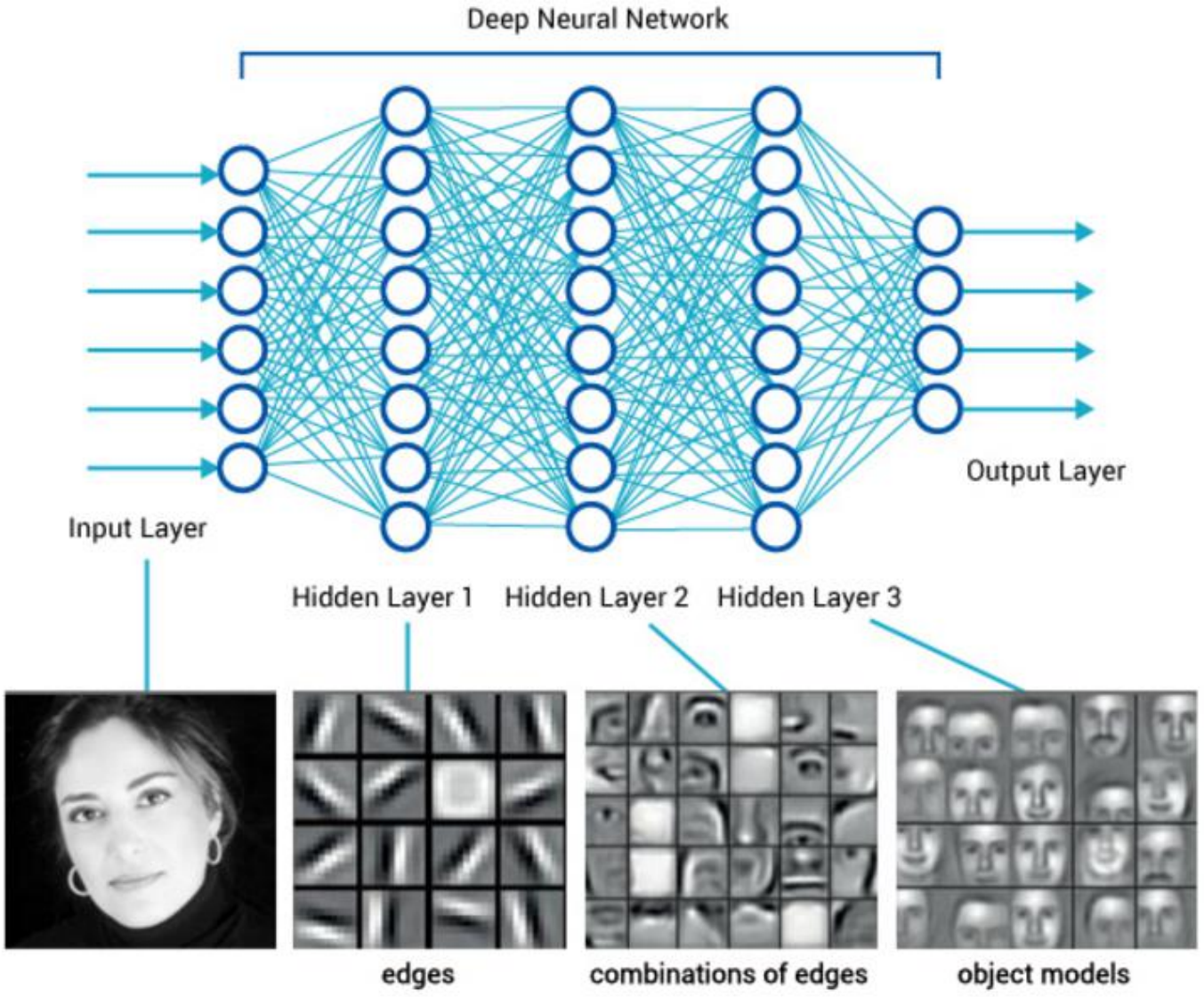
Example: LeNet5

```
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5, padding=2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1   = nn.Linear(16*5*5, 120)
        self.fc2   = nn.Linear(120, 84)
        self.fc3   = nn.Linear(84, 10)

    def forward(self, x):
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), (2, 2))
        x = flatten(x)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

- 前馈计算：

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

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。

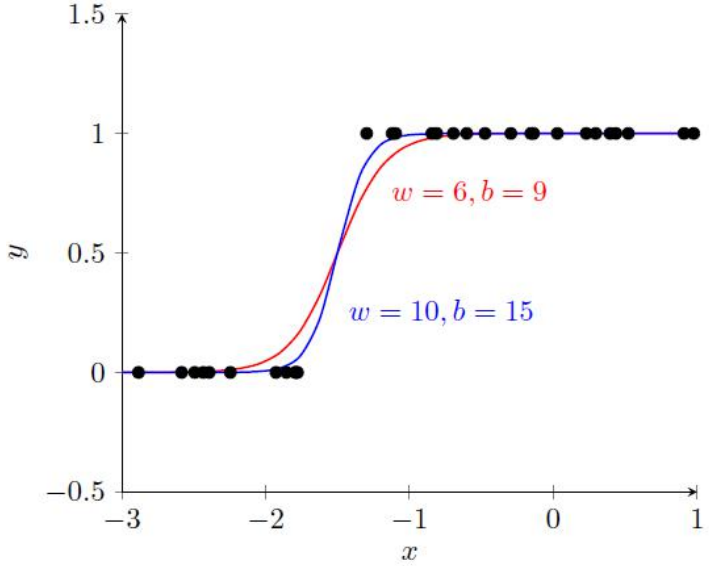
根据通用近似定理，对于具有线性输出层和至少一个使用“挤压”性质的激活函数的隐藏层组成的前馈神经网络，只要其隐藏层神经元的数量足够，它可以以任意的精度来近似任何从一个定义在实数空间中的有界闭集函数。

应用到机器学习

- 神经网络可以作为一个“万能”函数来使用，可以用来进行复杂的特征转换，或逼近一个复杂的条件分布。

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

分类器　　神经网络



- 如果g(•)为logistic回归，那么logistic回归分类器可以看成神经网络的最后一层。

$$p(y = 1|\mathbf{x}) = \sigma(\mathbf{w}^{\mathrm{T}}\mathbf{x}) \triangleq \frac{1}{1 + \exp(-\mathbf{w}^{\mathrm{T}}\mathbf{x})}$$

应用到机器学习

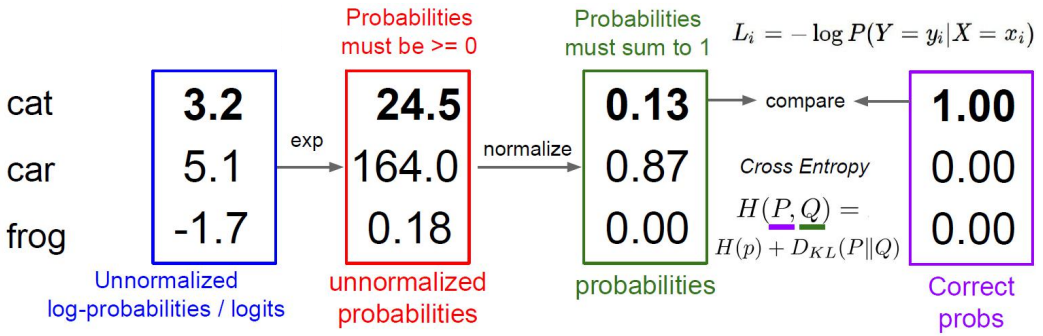
- **对于多类分类问题**
  - 如果使用softmax回归分类器，相当于网络最后一层设置C 个神经元，其输出经过softmax函数进行归一化后可以作为每个类的条件概率。

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z}^{(L)})$$

$$P(y = c|\mathbf{x}) = \text{softmax}(\mathbf{w}_c^{\mathrm{T}}\mathbf{x}) = \frac{\exp(\mathbf{w}_c^{\top}\mathbf{x})}{\sum_{i=1}^{C}\exp(\mathbf{w}_i^{\top}\mathbf{x})}.$$

  - 采用交叉熵损失函数，对于样本(x,y)，其损失函数为

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



参数学习

- 给定训练集为D，将每个样本输入给前馈神经网络，得到网络输出。其在数据集D上的结构化风险函数为：

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

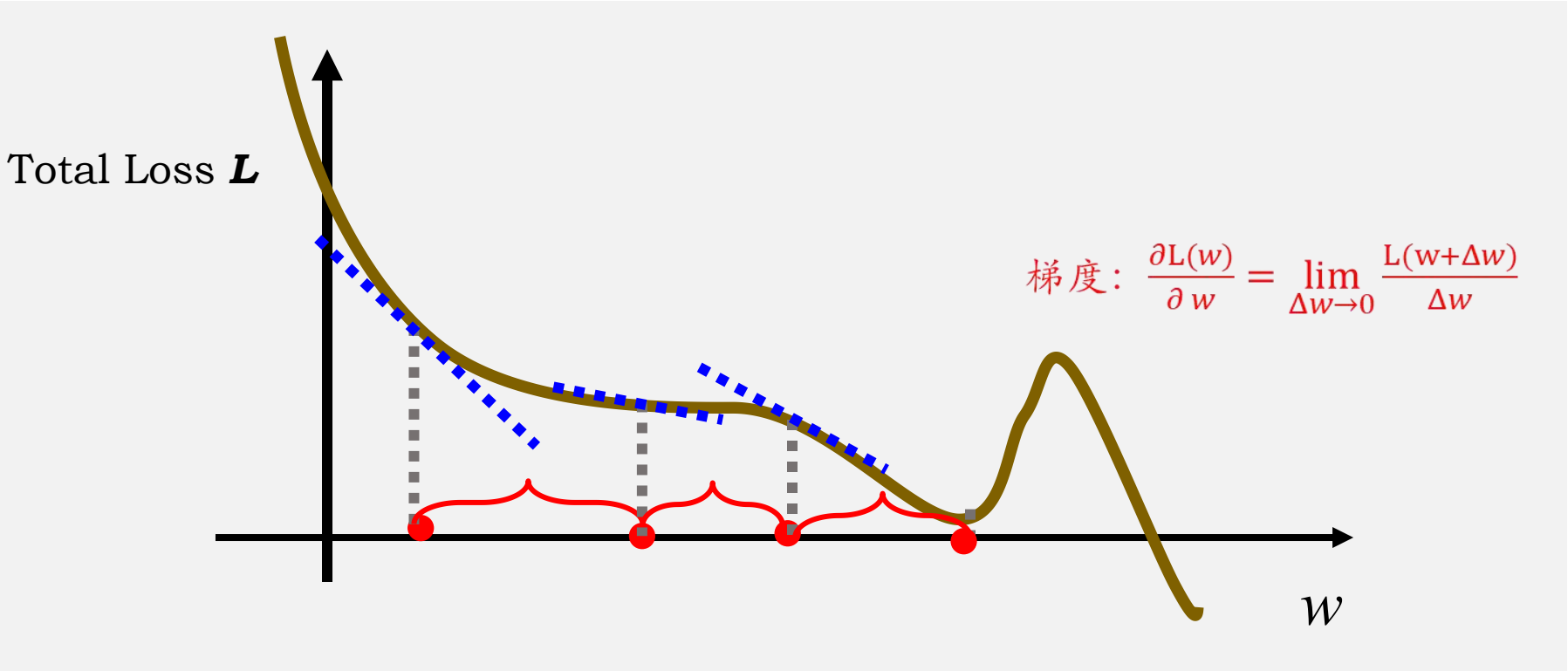
- 梯度下降

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

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

梯度下降

- 初始化网络参数w
- 重复
  - 计算梯度：$\partial L / \partial w$
  - 更新参数：$w \leftarrow w - \lambda \cdot \partial L / \partial w$



如何计算梯度?

- **神经网络为一个复杂的复合函数**
  - 链式法则

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

- **反向传播算法**
  - 根据前馈网络的特点而设计的高效方法

$$\frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial W_{ij}^{(l)}} = \left(\frac{\partial \mathbf{z}^{(l)}}{\partial W_{ij}^{(l)}} \right)^{\mathrm{T}} \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{z}^{(l)}},$$

$$\frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{b}^{(l)}} = \left(\frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}} \right)^{\mathrm{T}} \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{z}^{(l)}}.$$

- **一个更加通用的计算方法**
  - 自动微分（Automatic Differentiation，AD）

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。
- **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（Chain Rule）

- 链式法则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

只有 z_i 是与 $\boldsymbol{w}_{i:}$ 相关的

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

（1）计算偏导数 $\dfrac{\partial \mathbf{z}^{(l)}}{\partial W_{ij}^{(l)}}$

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

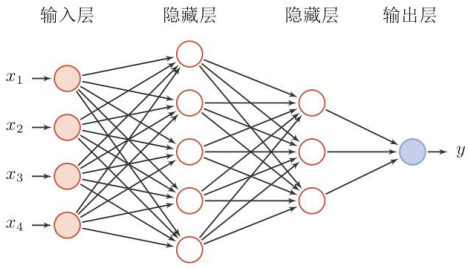
$$\frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial w_{ij}^{(l)}} = \frac{\partial \mathbf{z}^{(l)}}{\partial w_{ij}^{(l)}} \frac{\partial \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{z}^{(l)}}$$

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

（3）计算误差项

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

误差项



（2）计算偏导数 $\dfrac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}}$

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

$m^{(l)} \times m^{(l)}$ 的单位矩阵

计算

误差项 $\delta^{(l)}$ 来表示第 l 层神经元对最终损失的影响，也反映了最终损失对第 l 层神经元的敏感程度。误差项也间接反映了不同神经元对网络能力的贡献程度，从而比较好地解决了“贡献度分配问题”。

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

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

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

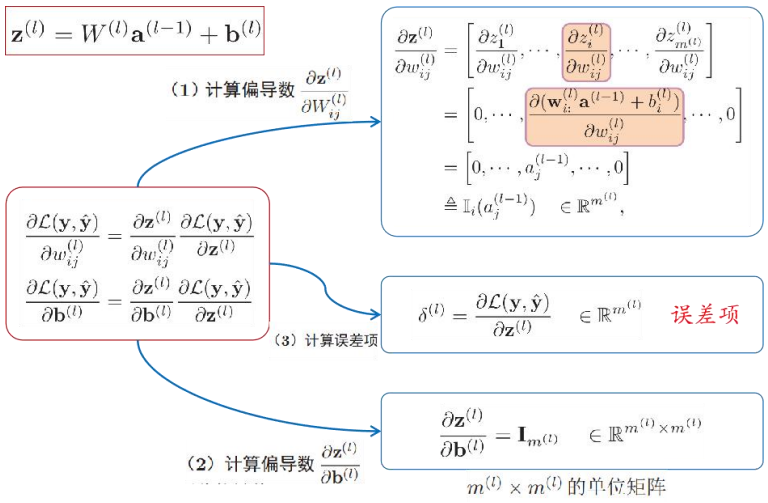
根据 $\mathbf{a}^{(l)} = f_l(\mathbf{z}^{(l)})$，其中 $f_l(\cdot)$ 为按位计算的函数

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

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

反向传播：第 l 层的一个神经元的误差项（或敏感性）是所有与该神经元相连的第 $l+1$ 层的神经元的误差项的权重和。然后，再乘上该神经元激活函数的梯度。

反向传播算法



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

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

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（backpropagation，BP）

1. 前馈计算每一层的净输入 $\mathbf{z}^{(l)}$ 和激活值 $\mathbf{a}^{(l)}$，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。

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

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

算法 4.1: 基于随机梯度下降的反向传播算法

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

随机初始化 $W, \mathbf{b}$；
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 $(\mathbf{x}^{(n)}, y^{(n)})$；
    前馈计算每一层的净输入 $\mathbf{z}^{(l)}$ 和激活值 $\mathbf{a}^{(l)}$，直到最后一层；
    反向传播计算每一层的误差 $\delta^{(l)}$； // 公式 (4.60)
    // 计算每一层参数的导数
    $\forall l, \quad \frac{\partial \mathcal{L}(\mathbf{y}^{(n)}, \hat{\mathbf{y}}^{(n)})}{\partial W^{(l)}} = \delta^{(l)} (\mathbf{a}^{(l-1)})^{\mathrm{T}}$； // 公式 (4.62)
    $\forall l, \quad \frac{\partial \mathcal{L}(\mathbf{y}^{(n)}, \hat{\mathbf{y}}^{(n)})}{\partial \mathbf{b}^{(l)}} = \delta^{(l)}$； // 公式 (4.63)
    // 更新参数
    $W^{(l)} \leftarrow W^{(l)} - \alpha(\delta^{(l)} (\mathbf{a}^{(l-1)})^{\mathrm{T}} + \lambda W^{(l)})$；
    $\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \alpha \delta^{(l)}$；
  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: $W, \mathbf{b}$

自动微分与计算图

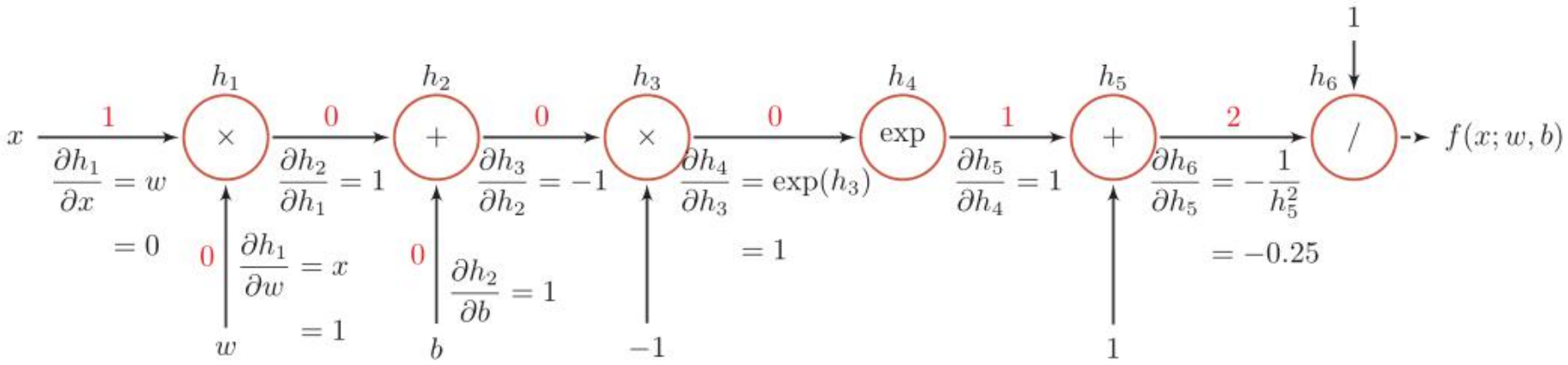
- 自动微分是利用链式法则来自动计算一个复合函数的梯度。

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

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

计算图

计算图（Computational Graph）



当 x = 1,w = 0,b = 0 时，可以得到

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

自动微分

- **前向模式和反向模式**
  - 反向模式和反向传播的计算梯度的方式相同

- **如果函数和参数之间有多条路径，可以将这多条路径上的导数再进行相加，得到最终的梯度。**

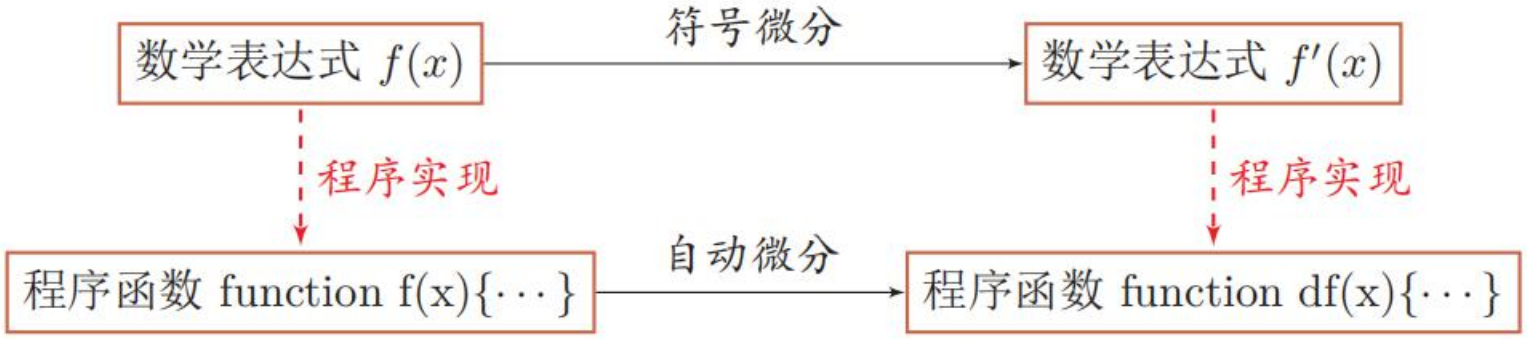
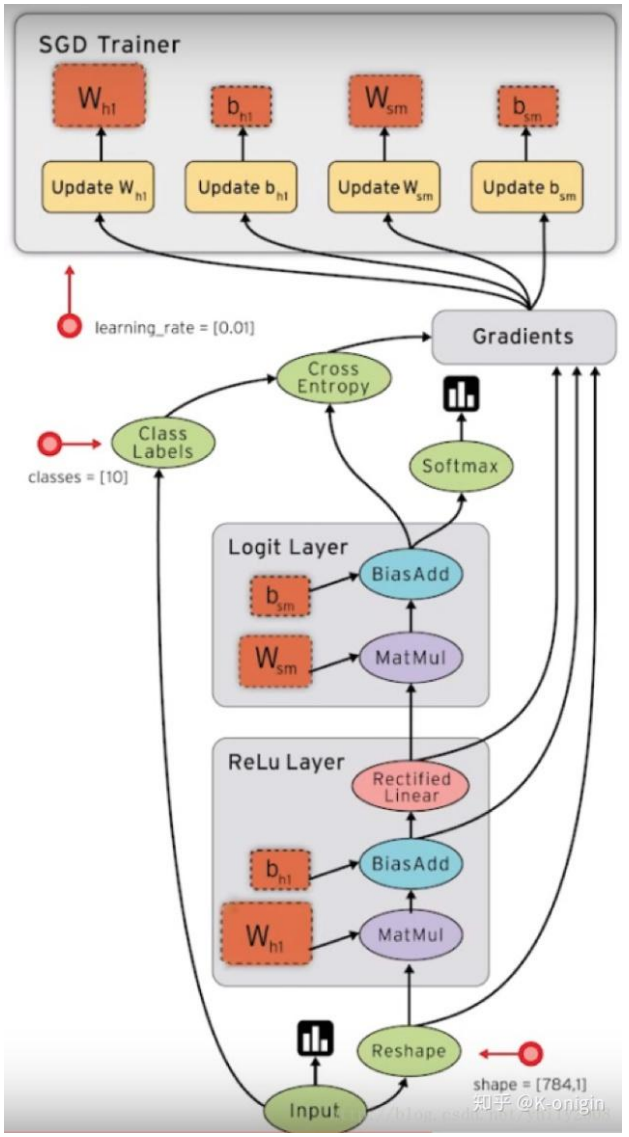


图 4.9 符号微分与自动微分对比

静态计算图和动态计算图

- **静态计算图**
  - 是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。
  - Theano和Tensorflow1.0

```
import tensorflow as tf

#定义计算图
g = tf.Graph()
with g.as_default():
    #placeholder为占位符，执行会话时候指定填充对象
    x = tf.placeholder(name='x', shape=[], dtype=tf.string)
    y = tf.placeholder(name='y', shape=[], dtype=tf.string)
    z = tf.string_join([x,y],name = 'join',separator=' ')

#执行计算图
with tf.Session(graph = g) as sess:
    print(sess.run(fetches = z,feed_dict={}))
```

静态计算图和动态计算图

- **动态计算图**
  - 是在程序运行时动态构建。两种构建方式各有优缺点。
  - DyNet，Chainer，PyTorch，TensorFlow2.0

A graph is created on the fly


```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)
```

静态计算图和动态计算图

- **动态计算图**
  - 是在程序运行时动态构建。两种构建方式各有优缺点。
  - DyNet，Chainer，PyTorch，TensorFlow2.0

In [3]:
```
# 动态计算图在每个算子处都进行构建，构建后立即执行

x = tf.constant("hello")
y = tf.constant("world")
z = tf.strings.join([x,y],separator=" ")

tf.print(z)
```

```
hello world
```

In [4]:
```
# 可以将动态计算图代码的输入和输出关系封装成函数

def strjoin(x,y):
    z =  tf.strings.join([x,y],separator = " ")
    tf.print(z)
    return z

result = strjoin(tf.constant("hello"),tf.constant("world"))
print(result)
```

```
hello world
tf.Tensor(b'hello world', shape=(), dtype=string)
```

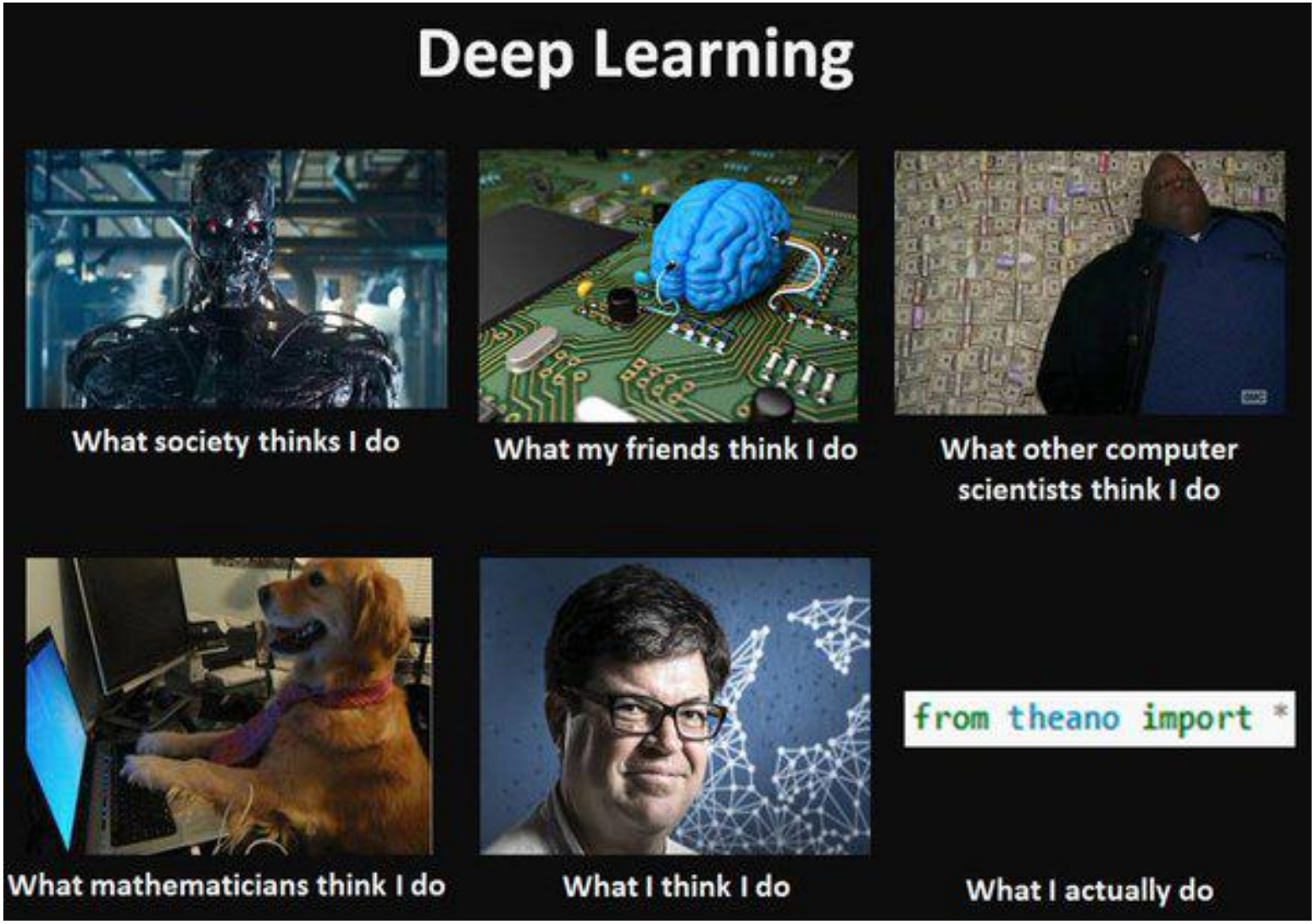
静态计算图和动态计算图

- **静态计算图**
  - 是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。
  - Theano和Tensorflow

- **动态计算图**
  - 是在程序运行时动态构建。两种构建方式各有优缺点。
  - DyNet，Chainer和PyTorch

- **静态 vs. 动态**
  - 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。

如何实现?

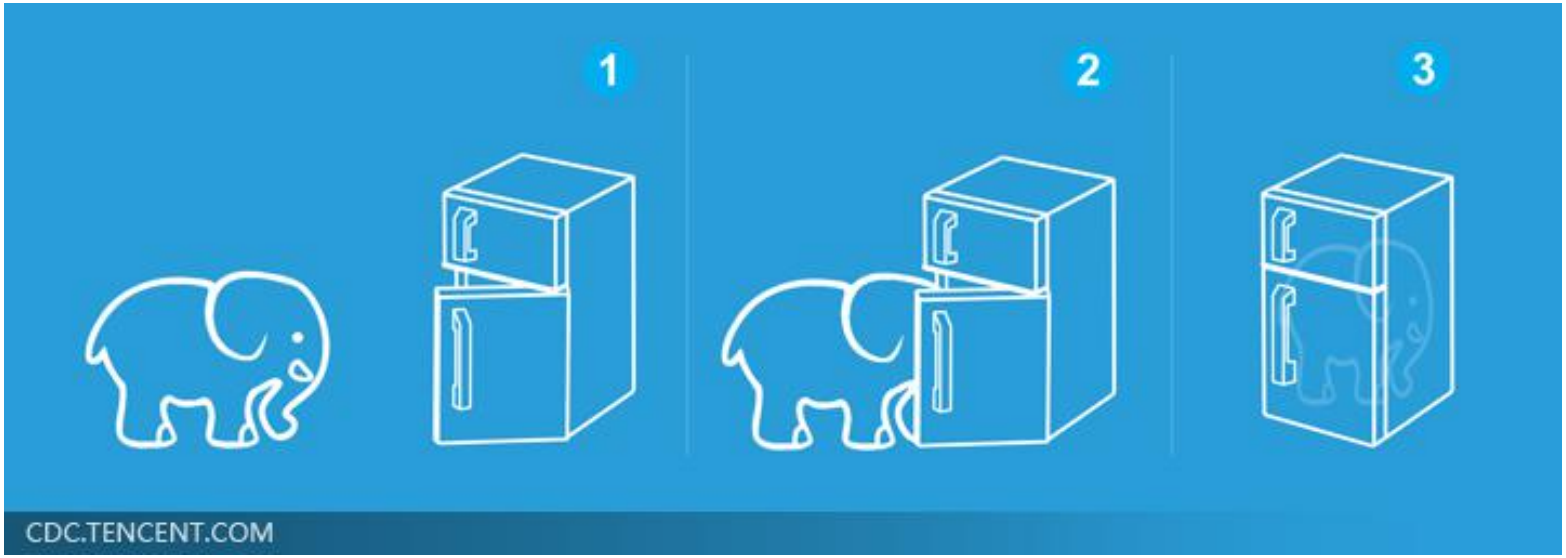

Deep Learning
What society thinks I do
What my friends think I do
What other computer scientists think I do
What mathematicians think I do
What I think I do
from theano import *
What I actually do

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)
```

深度学习的三个步骤

Deep Learning is so simple ……



优化问题

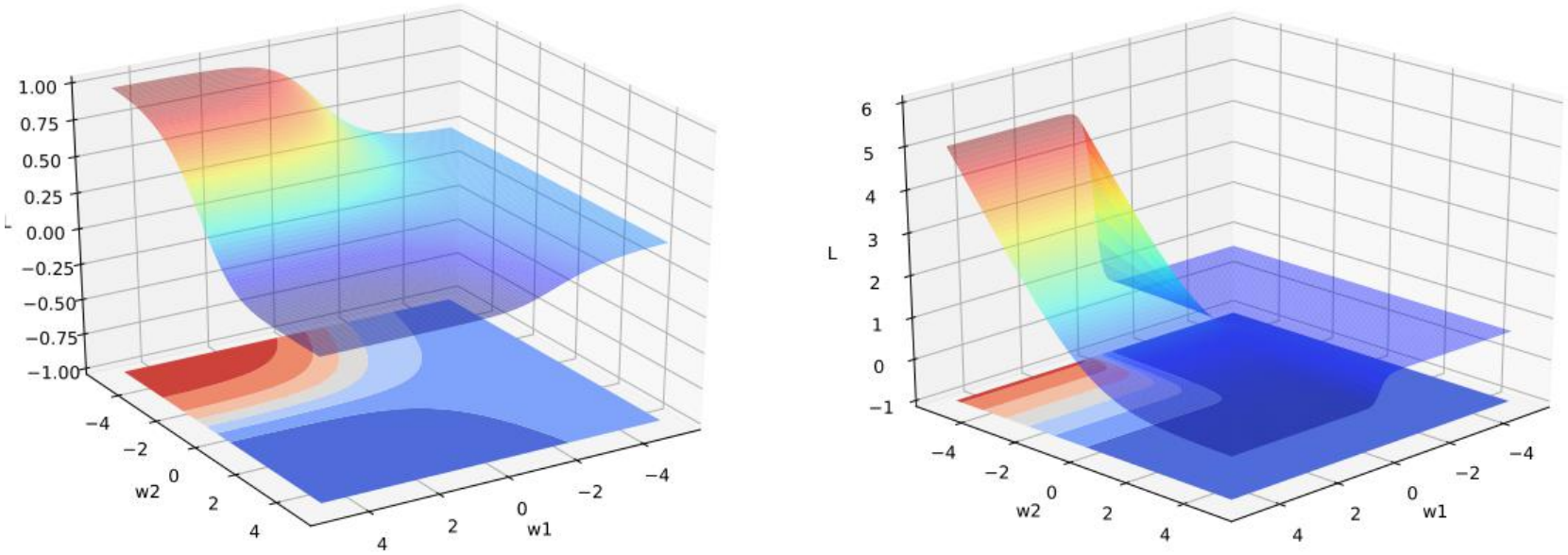
- **难点**
  - 参数过多，影响训练
  - 非凸优化问题：即存在局部最优而非全局最优解，影响迭代
  - 梯度消失问题，下层参数比较难调
  - 参数解释起来比较困难

梯度消失问题在过去的二三十年里一直没有有效地解决，是阻碍神经网络发展的重要原因之一。

- **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

- 非凸优化问题



(a) 平方误差损失　　(b) 交叉熵损失

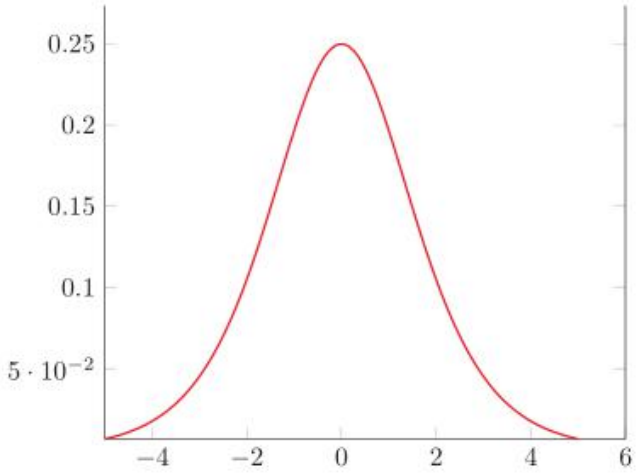
图 4.9 神经网络 $y = \sigma(w_2\sigma(w_1x))$ 的损失函数

优化问题

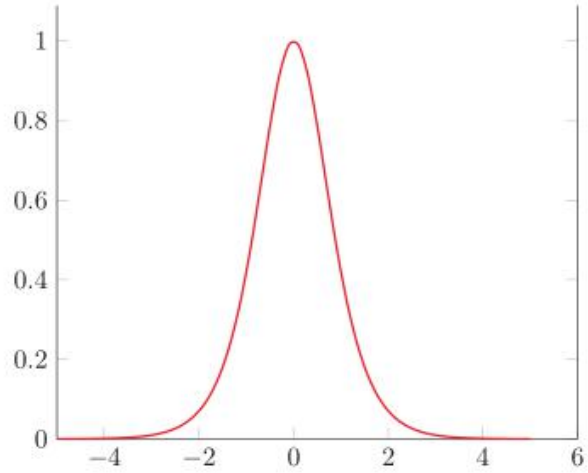
梯度消失问题（Vanishing Gradient Problem）

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

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



(a) logistic 函数的导数



(b) tanh 函数的导数

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

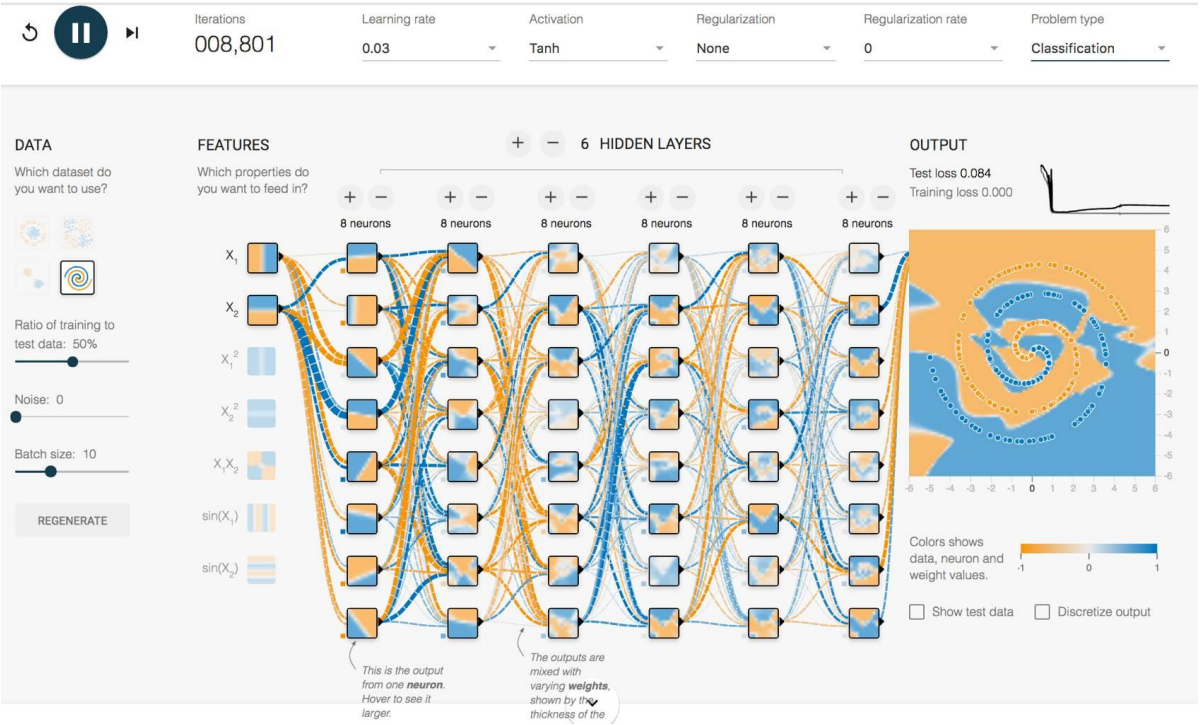
优化问题

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

表 4.2 常见激活函数及其导数

课后练习

- **知识点**
  - 激活函数
  - 误差反向传播
  - 自动微分与计算图
  - TensorFlow游乐场



http://playground.tensorflow.org

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

神经网络与深度学习

https://nndl.github.io/