第三讲 全连接网络

《深度学习》

南开大学 人工智能学院

线性模型的不足

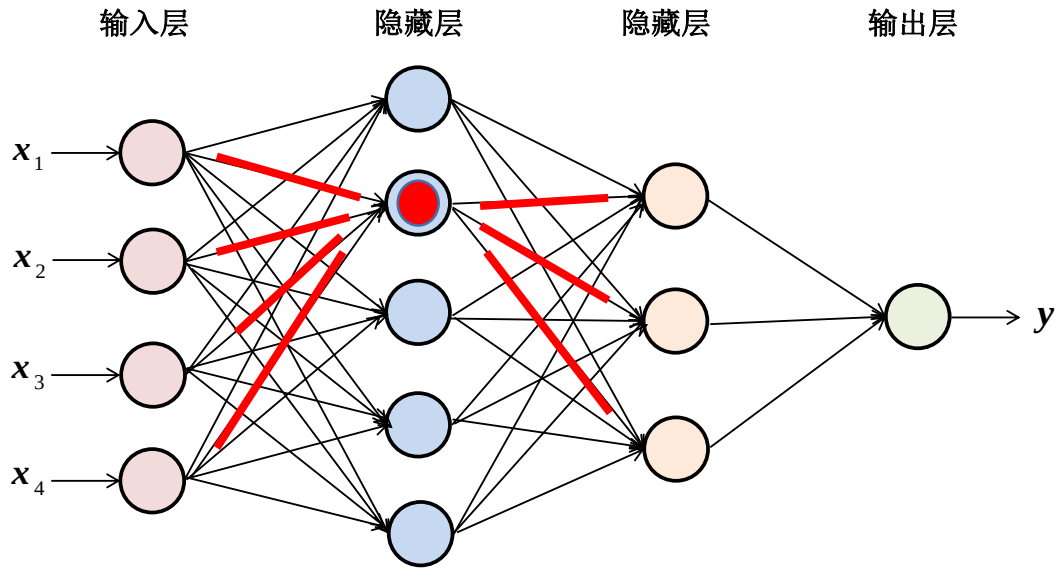
- 线性模型

$$f(w,b) = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$$

- 线性模型的特点
  - 任何特征x_i的增大都会导致模型输出的增大（如果对应的权重w_i为正）或减小（如果对应的权重为负）
- 线性模型的不足：无法建模任意两个输入变量间的相互作用
  - 示例：猫狗图像分类任务
    - 任何像素的重要性都以一种很复杂的方式取决于该像素的上下文（周围像素的值）
    - 线性模型无法建模任意两个输入变量间的相互作用

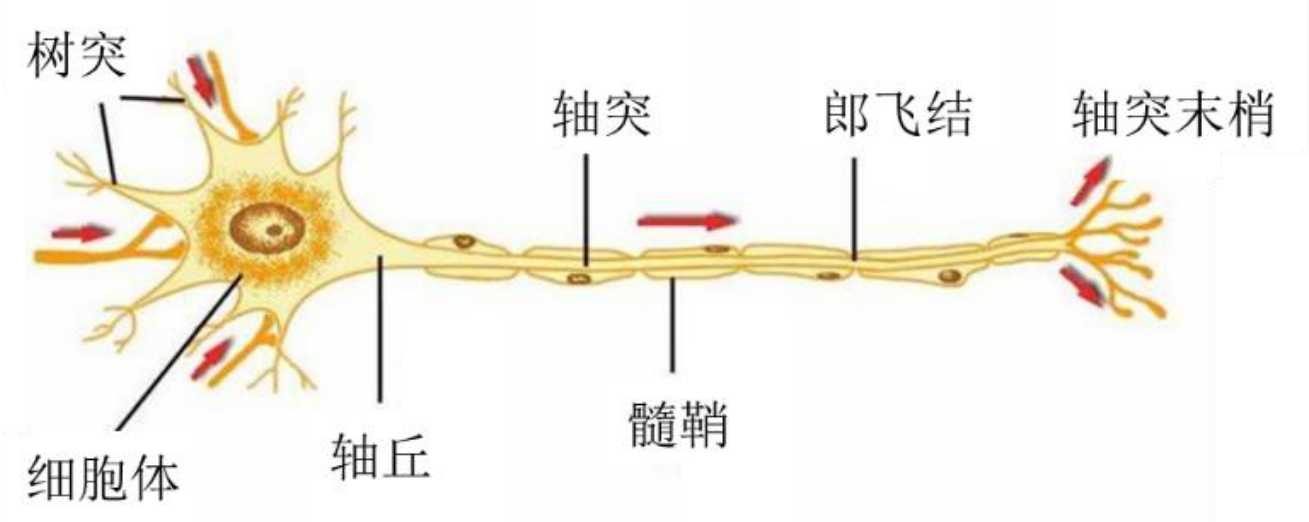
全连接网络

- **全连接网络 /多层感知机** 通过在网络中加入一个或多个隐藏层来克服线性模型的限制
  - 将多个全连接层堆叠在一起，每一层的输出都是后一层的输入，直到生成最后的输出
  - 全连接层：每一个神经元都跟前一层和后一层的所有神经元相连
  - 层数：只统计隐藏层和输出层



3.1.1 生物神经元

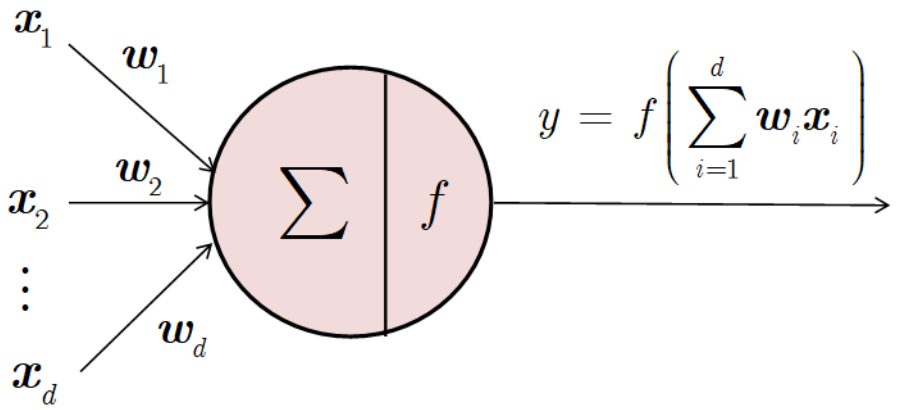
- 每个生物神经元由细胞体和多个延伸分支组成，后者又分为树突和轴突
  - 轴突的长度可能比细胞体长几倍，或者长几万倍，轴突在其末端分裂成许多分支，这些分支的顶端是称为突触的微小结构，与其他神经元的树突或细胞体相连



  - 当生物神经元接收到刺激时，产生电信号并将其沿着轴突传导至突触。在突触处，神经元释放化学物质，这些化学物质影响相邻神经元的电位，从而改变它们的状态。如果某个神经元的电位超过了一个特定的阈值，那么它就会被激活，并向其他神经元发送化学物质，进一步传递信息

3.1.2 人工神经元

- 人工神经元受生物神经元的工作原理启发而提出
  - 人工神经元具有1个或多个输入，模拟了生物神经元接收来自其他神经元的信号
  - 这些输入被加权求和，模拟了细胞体对神经信号的积累
  - 然后，对加权和进行激活函数计算，模拟了细胞体的兴奋或抑制过程
  - 最后，输出激活值传递到下一个人工神经元，模拟了生物神经元通过轴突将信号传递给其他神经元的过程



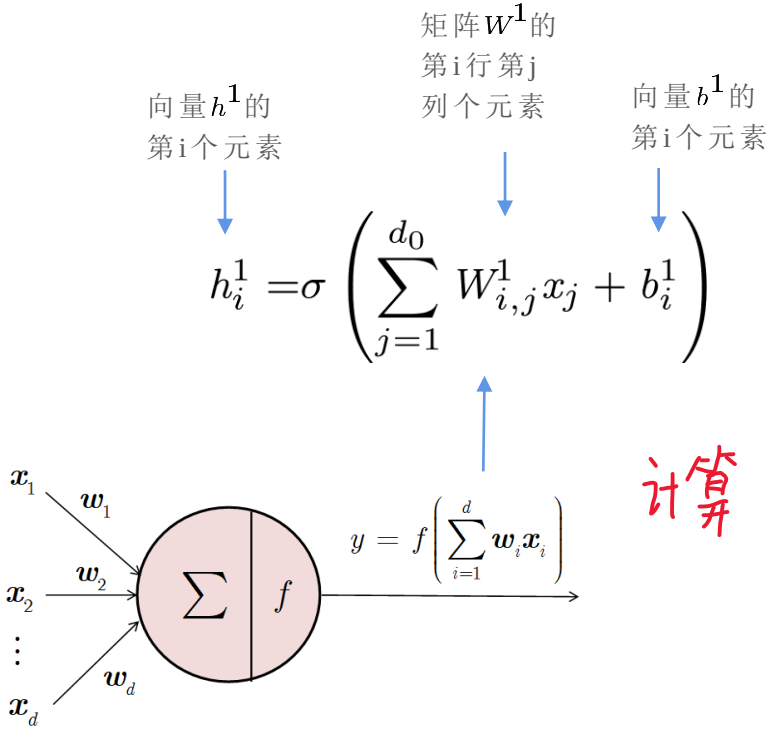
  - 基于这种人工神经元模型的多层神经网络能够模拟和学习复杂的非线性关系
    - 每个神经元都很简单，但多个神经元连接起来就可以处理复杂的问题——连接主义

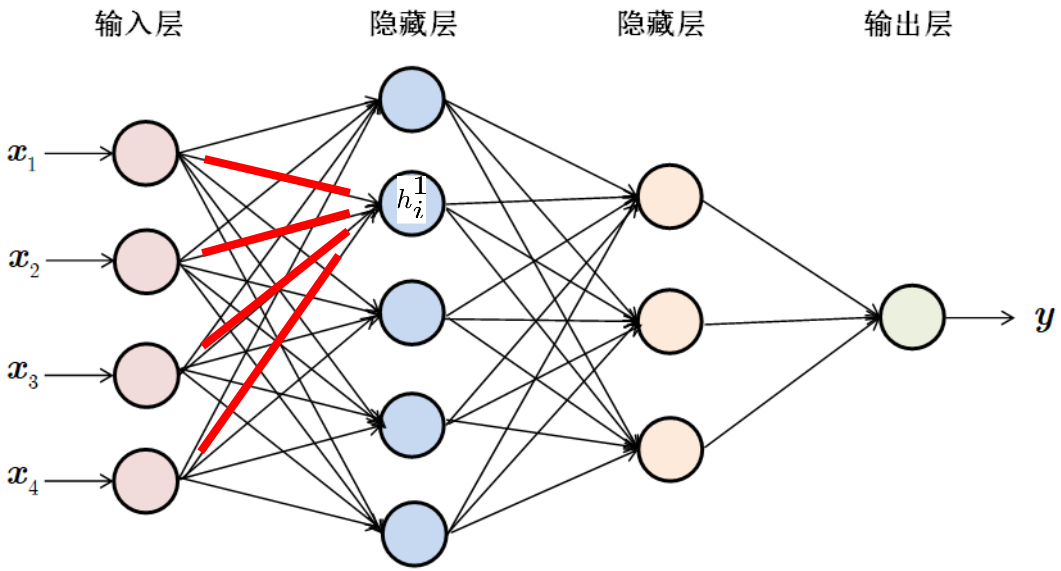
3.1.3 隐藏层

- 隐藏层可以增强网络的表达能力
- 将向量 $x \in \mathbb{R}^{d_0}$ 作为输入，记第一个隐藏层的输出为 $h^1 \in \mathbb{R}^{d_1}$

$$h^1 = \sigma\left(W^1 x + b^1\right)$$

  - 其中 $W^1 \in \mathbb{R}^{d_1 \times d_0}$ 为输入层和第一隐藏层的权重，$b^1 \in \mathbb{R}^{d_1}$ 为第一隐藏层偏置，σ 为激活函数





四条红边的权重：$W_{i,1}^1$, $W_{i,2}^1$, $W_{i,3}^1$, $W_{i,4}^1$,

3.1.3 隐藏层

- 将向量 $x \in \mathbb{R}^{d_0}$ 作为输入，记第一个隐藏层的输出为 $h^1 \in \mathbb{R}^{d_1}$

$$h^1 = \sigma\left(W^1 x + b^1\right)$$

  - 其中 $W^1 \in \mathbb{R}^{d_1 \times d_0}$ 为输入层和第一隐藏层的权重，$b^1 \in \mathbb{R}^{d_1}$ 为第一隐藏层偏置，σ 为激活函数

- 记第二个隐藏层的输出为 $h^2 \in \mathbb{R}^{d_2}$

$$h^2 = \sigma\left(W^2 h^1 + b^2\right)$$

  - 其中 $W^2 \in \mathbb{R}^{d_2 \times d_1}$ 为第一隐藏层和第二隐藏层的权重，$b^2 \in \mathbb{R}^{d_2}$ 为第二隐藏层偏置

- 记第 k 个隐藏层的输出为 $h^k \in \mathbb{R}^{d_k}$

$$h^k = \sigma\left(W^k h^{k-1} + b^k\right)$$

  - 其中 $W^k \in \mathbb{R}^{d_k \times d_{k-1}}$ 为第 k-1 隐藏层和 k 隐藏层的权重，$b^k \in \mathbb{R}^{d_k}$ 为第 k 隐藏层偏置

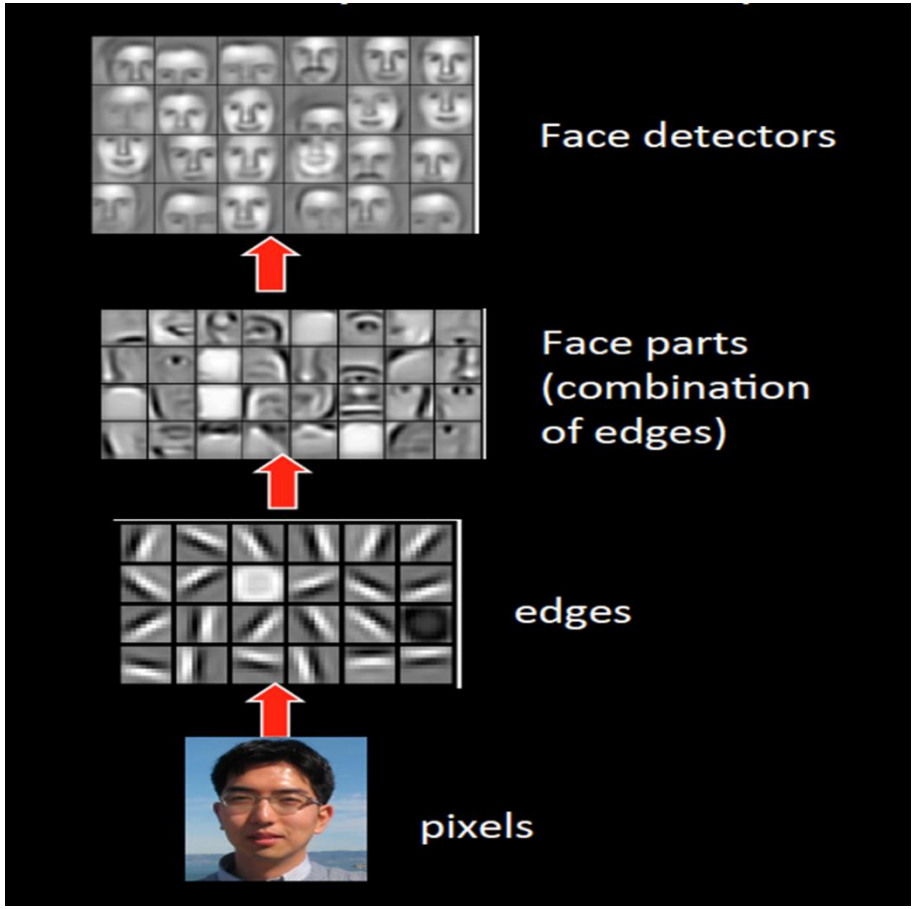
- 记输出为 $y \in \mathbb{R}^q$，并且网络共有有 L 层

$$y = \sigma\left(W^L h^{L-1} + b^L\right)$$

  - 如果为分类任务，一般还要添加一个softmax操作（见3.4.4节）

3.1.3 隐藏层

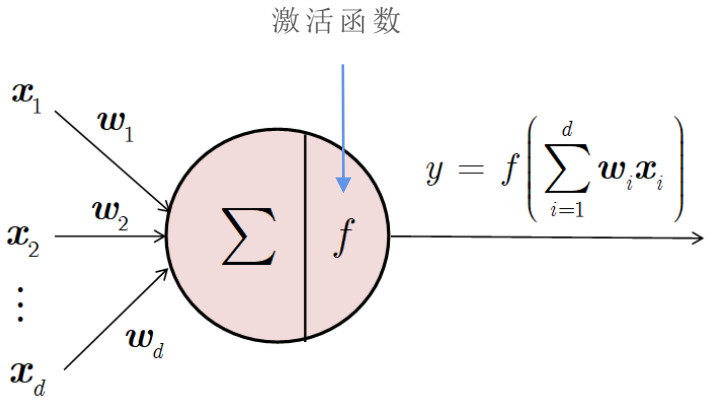
- 隐藏层作用：多个隐藏层可以实现对输入特征的多层次抽象，从而更好地处理数据
- 但目前普遍认为并不是隐藏层越多越好
- 增加隐藏层的数量理论上可以提高特征提取的效果，但会引发两个问题
  - 随着隐藏层数的增加，网络参数也会呈爆炸式增长，增加了训练的计算成本
  - 类似于边际效用递减规律，当隐藏层数量达到一定程度后，再继续增加隐藏层可能导致应用效果的提升越来越不明显，甚至可能引入过拟合问题
  - 因此，在设计神经网络时，需要权衡隐藏层数量与训练难度，并防止过拟合



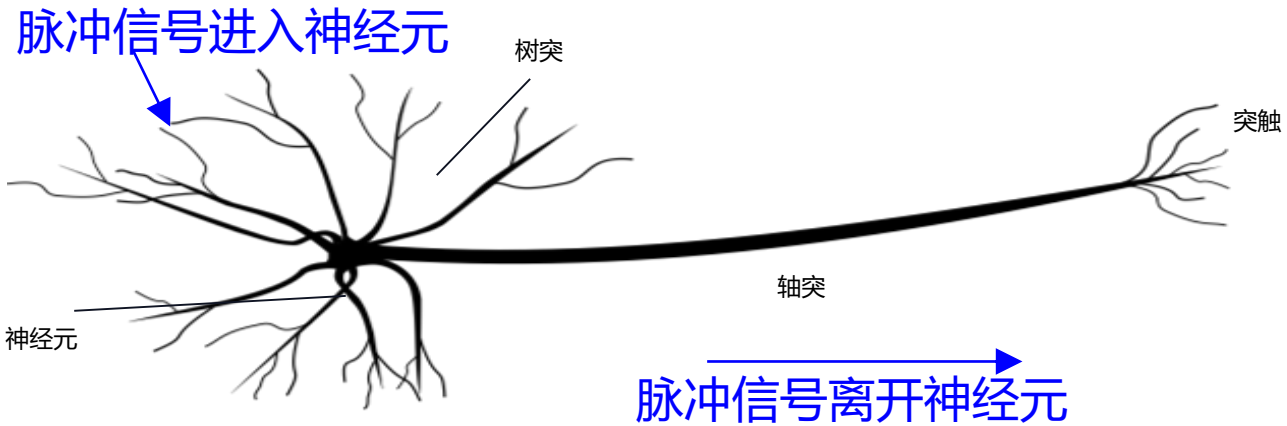
3.2 激活函数

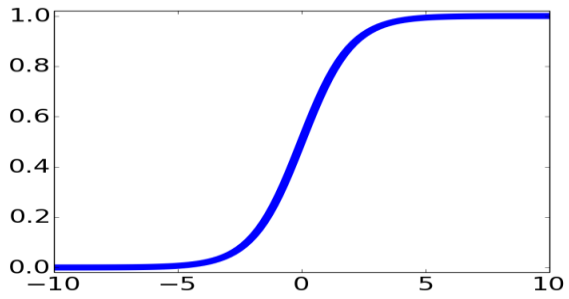
每种激活函数.概念

- **激活函数**是在神经元输入与输出之间的一种函数变换，目的是为了加入非线性因素，增强模型的表达能力
  - 人工神经元中，激活函数是检测某种特定特征的开关
- 激活函数需要具备以下几点性质：
  - 非线性函数
  - 连续并可导（允许少数点上不可导）
    - 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
    - 作为例外，当前常用的GeLU函数并不满足第三条性质



3.2.1 激活函数的由来

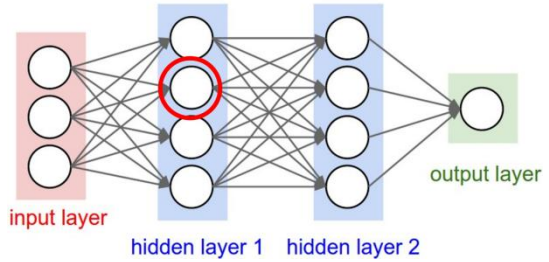


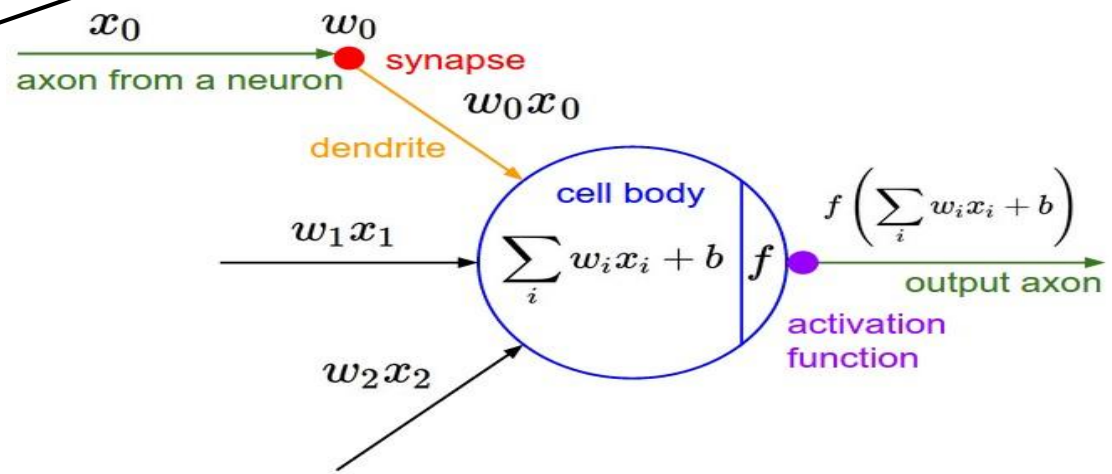


sigmoid 激活函数

$$\frac{1}{1+e^{-x}}$$

激活函数模拟了生物神经元的兴奋或抑制过程





3.2.2 激活函数的必要性

- 考虑只有一个隐藏层的无激活函数的神经网络

$$
\begin{aligned}
h_1 &= W_1 x + b_1 \\
h_2 &= W_2 h_1 + b_2
\end{aligned}
\quad \Longrightarrow \quad
\begin{aligned}
h_2 &= W_2 h_1 + b_2 \\
&= W_2 (W_1 x + b_1) + b_2 \\
&= W_2 W_1 x + (W_2 b_1 + b_2) \\
&= Wx + b
\end{aligned}
$$

其中 $W = W_2 W_1$，$b = W_2 b_1 + b_1$

  - 等价于没有使用隐藏层的简单线性网络
    - 隐藏层没有起作用
    - 线性函数和线性函数的复合仍然是线性函数
- 为了发挥多层架构的潜力，需要在线性变换之后对每个隐藏层单元应用**非线性**的**激活函数**

$$
\begin{aligned}
h_1 &= \sigma(W_1 x + b_1) \\
h_2 &= \sigma(W_2 h_1 + b_2)
\end{aligned}
$$

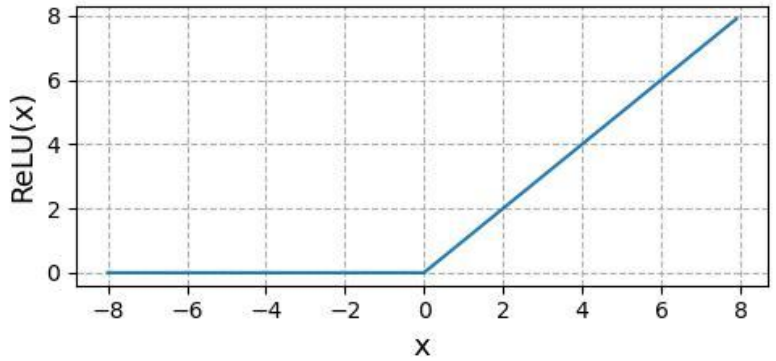
  - 有了激活函数，就不可能再将多层神经网络退化成简单线性模型
  - 激活函数为神经网络加入了非线性因素，增强模型的表达能力

3.2.3 ReLU

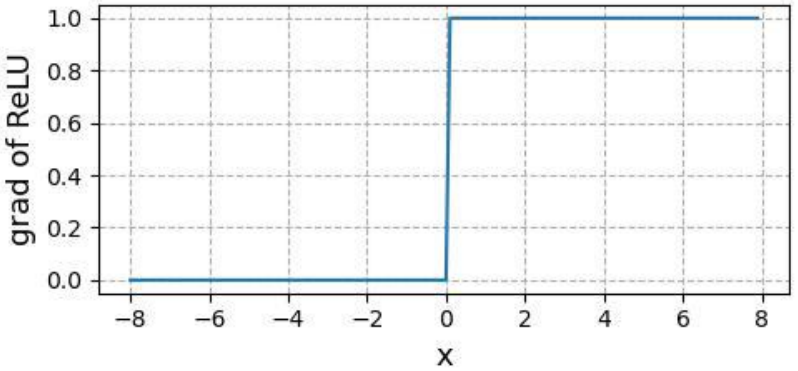
- **修正 / 整流线性单元**（Rectified linear unit，ReLU）
  - 给定输入x，ReLU函数输出该元素与0的最大值:

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

  - ReLU函数通过将激活阈值设为0，仅保留正元素并丢弃负元素
  - 具有两个线性部分的分段线性函数，保留了线性函数的很多良好性质
- 当输入为负时，ReLU函数的导数为0
- 而当输入为正时，ReLU函数的导数为1
- 当输入值精确等于0时，ReLU函数不可导
  - 此时，一般按导数为0处理
  - 在计算机系统中，很少有数值精确等于0
- 反向传播时，ReLU的导数形式要么让信息流消失，要么让信息流完整通过



ReLU函数



ReLU函数导数

3.2.3 ReLU

- 使用ReLU的一个隐藏层

$$h_k = \mathrm{ReLU}\left(W_k h_{k-1} + b_k\right)$$

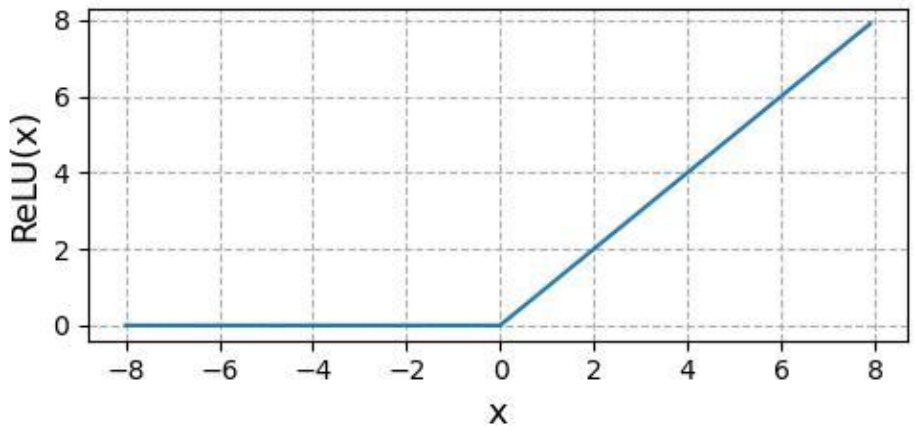
  - 一般将 b_k 初始化为较小的正数，例如0.1，从而使得初始时ReLU对大部分输入呈激活状态，允许通过
- ReLU函数的优点是计算简单，计算速度快
- ReLU的缺点是当输入为负数时，ReLU函数的输出及导数都为零，神经元将不会更新权重，此时产生了“dead neurons”问题

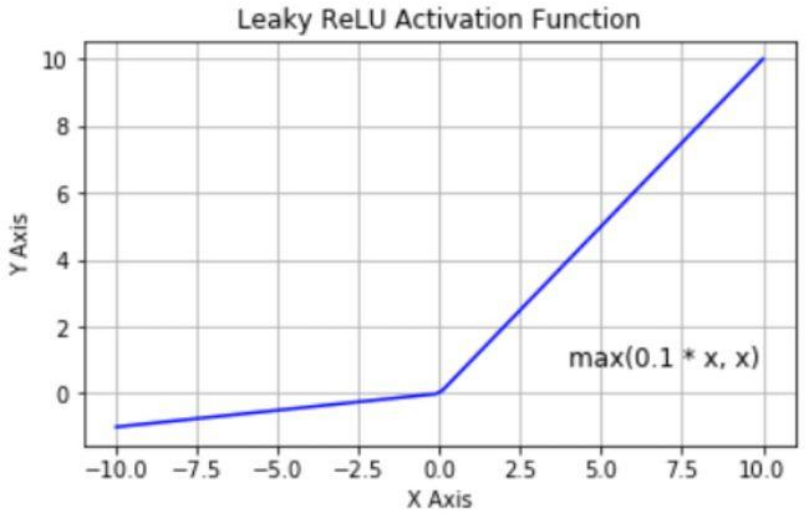
3.2.3 ReLU

- 扩展：Leaky ReLU

$$\mathrm{LReLU}(x) = \max(x, 0) + \alpha \min(x, 0) = \begin{cases} x, & x \geq 0 \\ \alpha x, & x < 0 \end{cases}, \quad \frac{\partial}{\partial x}\mathrm{LReLU} = \begin{cases} 1, & x \geq 0 \\ \alpha, & x < 0 \end{cases}$$

  - 一般取 $\alpha = 0.01$
  - 作用：当输入为负时，也允许通过，只是通过的量很小，解决了ReLU“Dead neurons”的问题
- ReLU及其扩展的设计原则：如果它们的行为更接近线性，那么模型更容易优化





3.2.3 ReLU

- PyTorch API

API: torch.nn.ReLU(inplace=False)

示例：

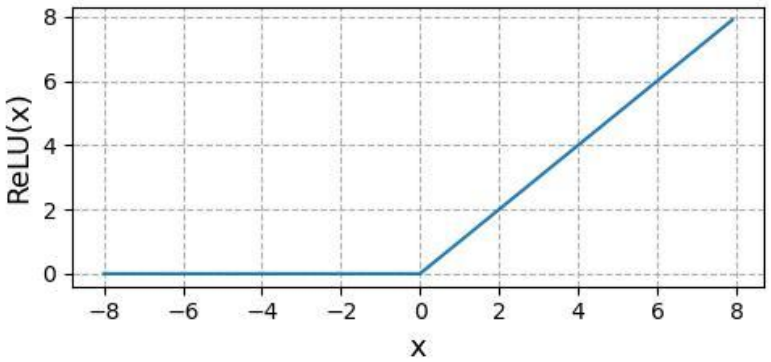
```
import torch
from torch import nn                                    #本节其余示例中将忽略这两行导入语句
m = nn.ReLU()
input = torch.randn(2)
output = m(input)
```

3.2.4 Softplus

- ReLU：$x=0$ 处不光滑
- Softplus：ReLU的平滑近似

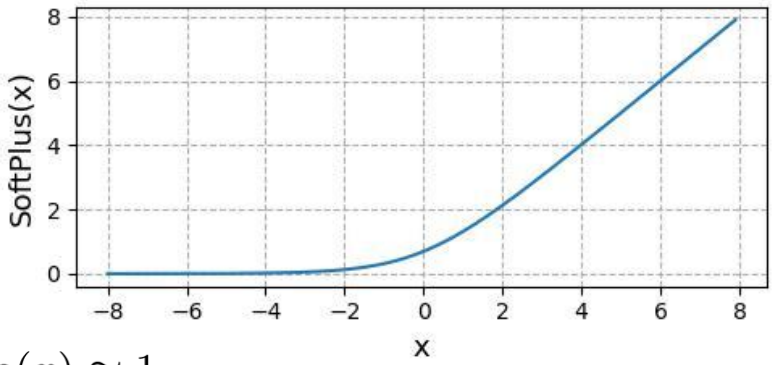
$$\text{softplus}(x)=\log(1+\exp(x))$$

$$1+\exp(x)\approx\exp(x)$$
$$\log\exp(x)=x$$



ReLUs函数

$$1+\exp(x)\approx 1$$
$$\log 1=0$$



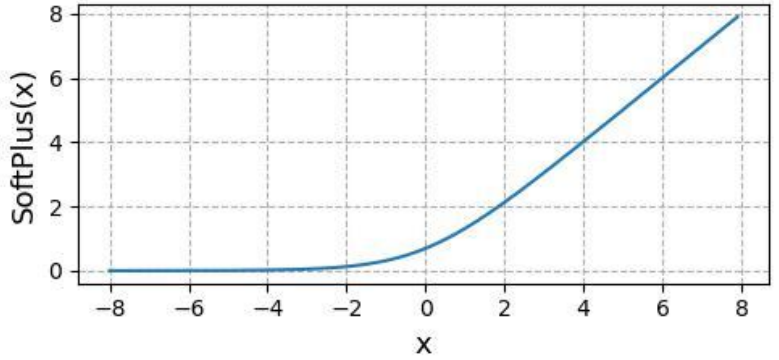
Softplus函数

3.2.4 Softplus

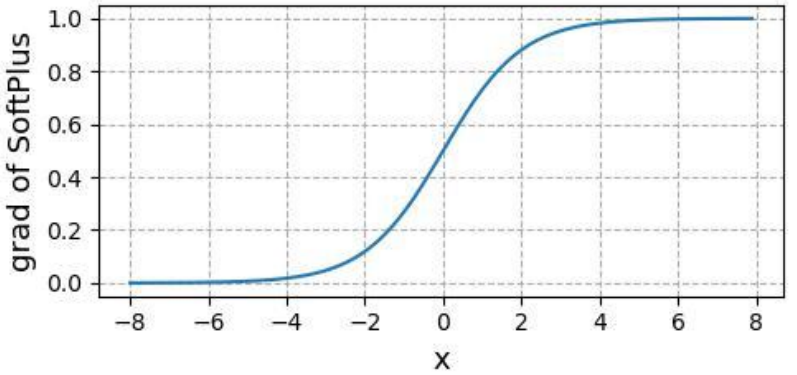
- Softplus处处可导

$$\frac{\partial}{\partial x}\text{softplus}(x) = \frac{\exp(x)}{1+\exp(x)} = \frac{1}{1+\exp(-x)}$$

  - 当 x 是很大的正数时，softplus(x)接近 x，其导数接近1
  - 当 x 是很小的负数时，softplus(x)接近 0，其导数接近0
  - 当 x 等于 0 时， softplus(x)=log2，其导数为1/2
    - ReLU在 0 处不可导
- 与ReLU相比，SoftPlus具有平滑性，但SoftPlus涉及指数运算，可能导致数值计算上的不稳定性，实际效果可能不如ReLU函数



Softplus函数



Softplus函数导数

3.2.4 Softplus

- PyTorch API

API: torch.nn.Softplus(beta=1.0, threshold=20.0)

参数：beta：softplus 的广义实现 $\frac{1}{\beta}\log\left(1+\exp(\beta x)\right)$ 中的 β，默认取 1；

threshold：为了数值稳定性，当 $\beta x \geq$ threshold 时，softplus 退化为线性函数

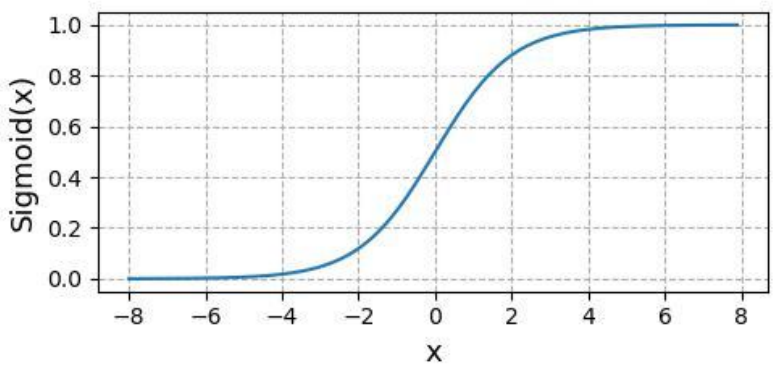
示例：
```
m = nn.Softplus()
input = torch.randn(2)
output = m(input)
```

3.2.5 Sigmoid

- 将范围（-inf,inf）上的任意输入压缩到区间（0,1）上

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

  - 生物神经元中阈值函数：输入高于某个阈值时取1，低于阈值时取0
  - Sigmoid函数是0-1阈值函数的一个常用近似
  - Sigmoid函数适用于将预测概率作为输出的模型（尤其是二分类）或需要门控机制的模型中（如LSTM）



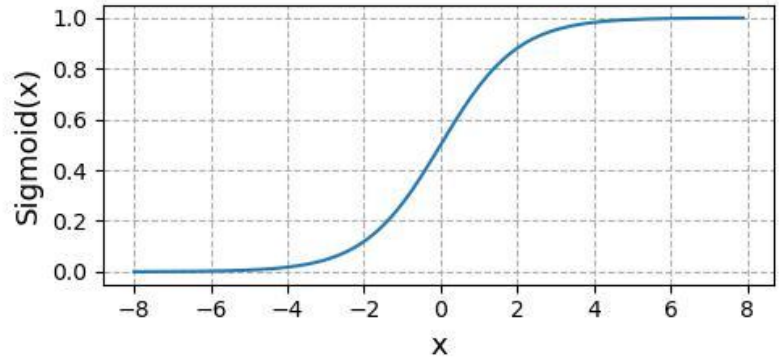
Sigmoid函数

3.2.5 Sigmoid

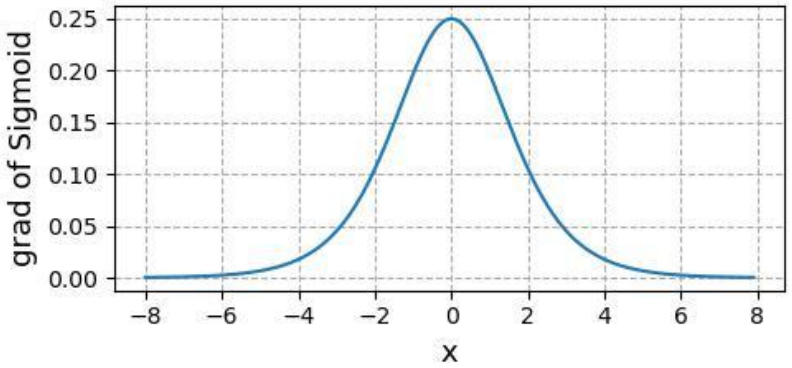
- Sigmoid函数的导数

$$\frac{d}{dx}\,\text{sigmoid}(x) = \frac{\exp(-x)}{(1+\exp(-x))^2} = \text{sigmoid}(x)\,(1-\text{sigmoid}(x))$$

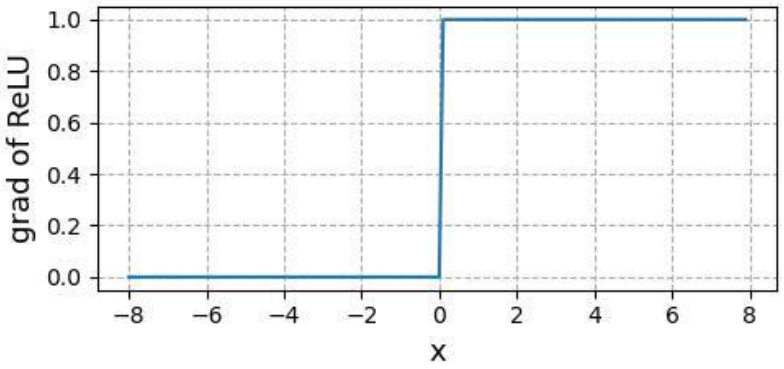
  - 当sigmoid输出接近1或0时，导数接近0
  - 对于大部分输入，sigmoid的导数接近为0
  - 当输入为0时，Sigmoid函数的导数取最大值0.25
  - 反向传播时，sigmoid的导数形式让大部分信息流消失
  - **Sigmoid导致梯度消失**
  - 与SoftPlus函数一样，Sigmoid函数涉及指数运算，计算速度较慢
- **ReLU减轻了sigmoid的梯度消失问题**
  - 当输入为负时，ReLU函数的导数为0
  - 当输入为正时，ReLU函数的导数为1



Sigmoid函数



Sigmoid函数导数



ReLU函数导数

3.2.5 Sigmoid

- PyTorch API

```
API: torch.nn.Sigmoid(*args, **kwargs)
示例：  m = nn.Sigmoid()
        input = torch.randn(2)
        output = m(input)
```

3.2.6 tanh

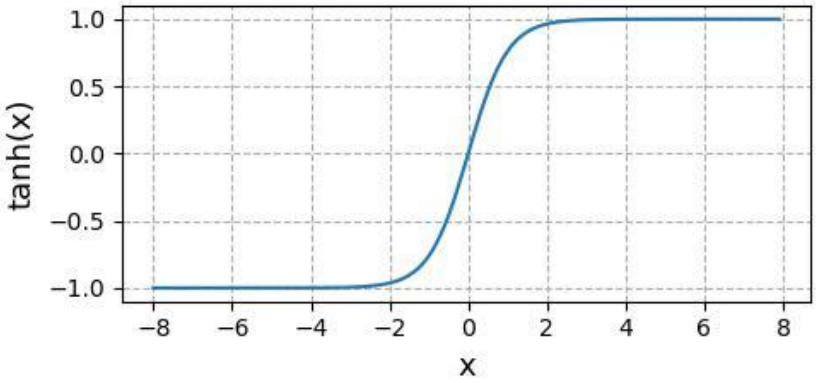
- **tanh（双曲正切函数）**
  - 将范围（-inf,inf）上的任意输入压缩到区间（-1,1）上

$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$

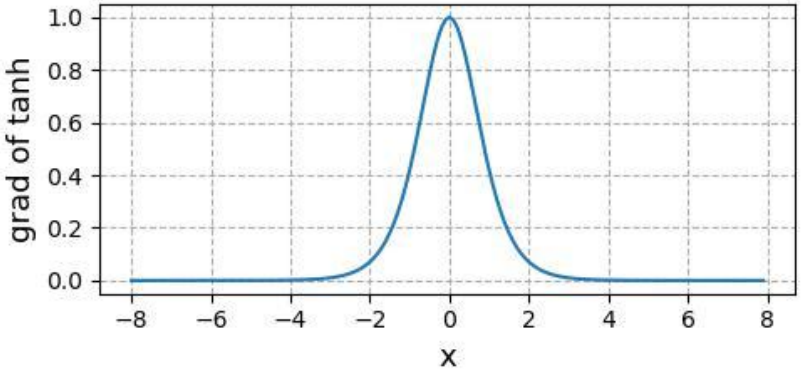
  - 关于坐标系原点中心对称
- tanh函数的导数

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x)$$

  - 当tanh输出接近1或-1时，导数接近0
  - 对于大部分输入，tanh的导数接近为0
  - 反向传播时，tanh的导数形式让大部分信息流消失
  - **tanh导致梯度消失**



tanh函数



tanh函数导数

3.2.6 tanh

- PyTorch API

```
API: torch.nn.Tanh(*args, **kwargs)
示例：  m = nn.Tanh()
        input = torch.randn(2)
        output = m(input)
```

3.2.7 GeLU

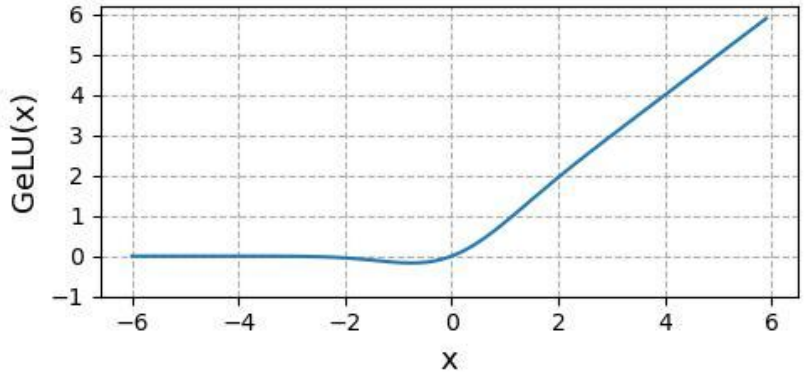
- **GeLU（高斯误差线性单元）**
  - 基于高斯误差函数，常用于GPT、BERT等大语言模型

$$\mathrm{GeLU}(x) = x\Phi(x)$$

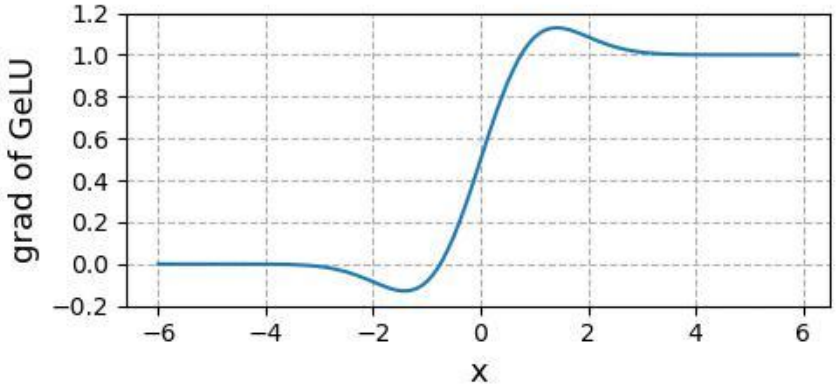
  - $\Phi(x)$是x的高斯正态分布的累计函数
  - GeLU是一个非初等函数，形式复杂
  - 在代码实现中，GeLU可以近似表示为

$$\mathrm{GeLU}(x) = 0.5x\left(1 + \tanh\left[\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)\right]\right)$$

  - 对于较大的输入$x > 0$，GeLU函数近似于线性输出
  - 对于较小的输入$x < 0$，GeLU函数的输出接近0
  - 当输入接近于0时，GeLU函数是连续非线性的
  - GeLU也可以看做是ReLU的一种平滑近似
  - 虽然GeLU的计算比ReLU复杂，但现代硬件（如GPU）对其有良好的支持



GeLU函数



GeLU函数导数

3.2.7 GeLU

- PyTorch API

API: torch.nn.functional.gelu(input, approximate='none')
参数： Appxoximate：当为 none 时，不使用近似计算，当为 tanh 时，使用近似计算。
示例：
```
m = nn.GELU()
input = torch.randn(2)
output = m(input)
```

3.2.8 其他激活函数

- PyTorch也实现了nn.ELU、nn. Hardshrink、nn. Hardsigmoid、nn. Hardtanh、nn. Hardswish、nn. LogSigmoid、nn. PReLU、nn. ReLU6、nn. RReLU、nn. SELU、nn. CELU、nn. SiLU、nn. Mish、nn. Softshrink、nn. Softsign、nn. Tanhshrink、nn. Threshold、nn. GLU等激活函数
- PyTorch官方文档https://pytorch.org/docs/stable/nn.html中的Non-linear Activations部分

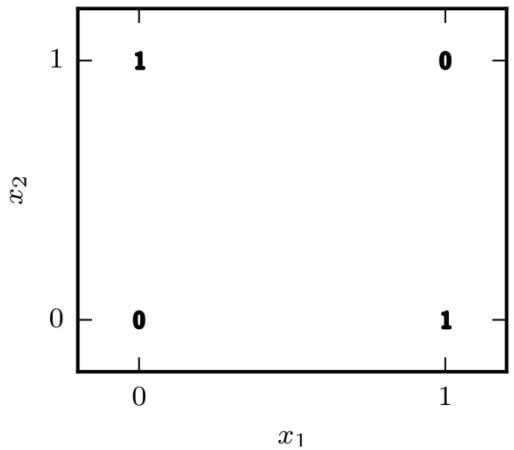
3.2.9 小结

- 一般很少使用 sigmoid
  - 梯度消失
  - 作为例外，LSTM中使用了sigmoid
- 如果一定要使用 sigmoid，可以考虑 tanh
- 默认选择 ReLU
- 尝试 ReLU 的若干扩展，可能会有提升
- GeLU在当前的大语言模型中被使用

3.3 多层的必要性：解决异或问题

- XOR 函数（“异或”逻辑）是两个二进制值 x1 和 x2 的运算

$$\mathrm{XOR}(x) = \begin{cases} 0, & x = (1,1) \\ 1, & x = (1,0) \\ 1, & x = (0,1) \\ 0, & x = (0,0) \end{cases}$$



- 马文·闵斯基指出单个神经元无法学习异或问题
- 多层网络是否可以学习异或问题？
  - 希望设计一个网络，在这四个点X={[0, 0],[0, 1],[1, 0],[1, 1]}上表现正确
  - 在这个简单的例子中，不考虑统计泛化，唯一的挑战是拟合训练集

3.3 多层的必要性：解决异或问题

- 输入层：$x = (x_1, x_2)$
- 隐藏层包含两个神经元，使用ReLU激活函数

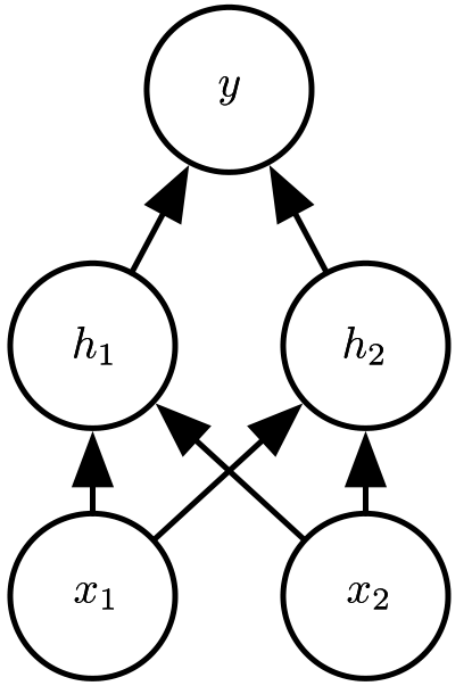
$$h = \max(0, Wx + c)$$

- 输出层包含一个神经元

$$y = w^T h$$

- 最终，整个网络表示为

$$f = w^T \max(0, Wx + c)$$



3.3 多层的必要性：解决异或问题

- 给出 XOR 问题的一个解

$$\boldsymbol{W}=\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix} \qquad \boldsymbol{c}=\begin{bmatrix}0\\ -1\end{bmatrix} \qquad \boldsymbol{w}=\begin{bmatrix}1\\ -2\end{bmatrix}$$

  - 回顾

$$f = w^T \max(0, Wx + c)$$

- 输入4个样本的矩阵表示为 $\boldsymbol{X}=\begin{bmatrix}0 & 0\\ 0 & 1\\ 1 & 0\\ 1 & 1\end{bmatrix}$

- 将输入矩阵乘以第一层的权重矩阵

$$\boldsymbol{XW}=\begin{bmatrix}0 & 0\\ 1 & 1\\ 1 & 1\\ 2 & 2\end{bmatrix}$$

- 加上偏置向量 c，得到 $\begin{bmatrix}0 & -1\\ 1 & 0\\ 1 & 0\\ 2 & 1\end{bmatrix}$

- 使用ReLU，得到 $\begin{bmatrix}0 & 0\\ 1 & 0\\ 1 & 0\\ 2 & 1\end{bmatrix}$

- 最后乘以一个权重向量 w，得到 $\begin{bmatrix}0\\ 1\\ 1\\ 0\end{bmatrix}$

可以解决异或问题！

3.3 多层的必要性：解决异或问题

- 为求解异或问题训练一个全连接神经网络，该网络包含一个输入层、一个隐藏层和一个输出层：

$$\boldsymbol{h} = \mathrm{ReLU}(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}),$$
$$y = \boldsymbol{w}^T\boldsymbol{h} + b$$

  - $\boldsymbol{x} \in \mathbb{R}^2$，$\boldsymbol{W} \in \mathbb{R}^{5\times 2}$，$\boldsymbol{b} \in \mathbb{R}^5$，$\boldsymbol{h} \in \mathbb{R}^5$，$\boldsymbol{w} \in \mathbb{R}^5$，$b \in \mathbb{R}$，$y \in \mathbb{R}$
  - 隐藏层使用ReLU作为激活函数，输出层不使用激活函数
  - 使用均方误差损失作为损失函数
  - 使用随机梯度法（Stochastic Gradient Descent，SGD）训练网络
- 训练结果：

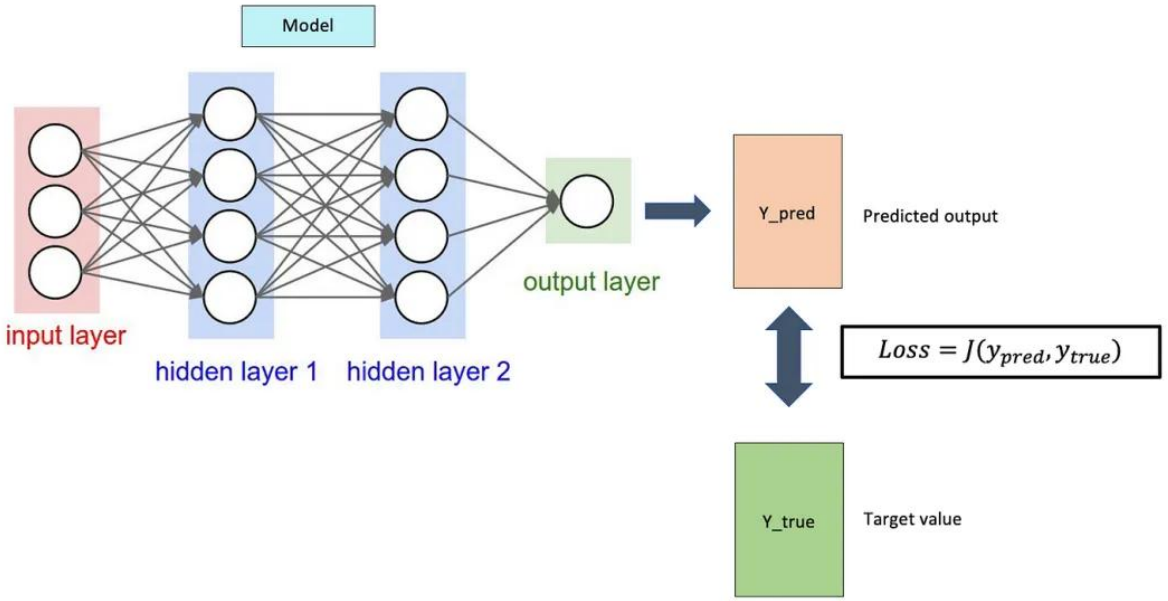
$$\boldsymbol{W} = \begin{bmatrix} 1.0728 & 1.0728 \\ 0.8363 & 1.0607 \\ -0.6455 & 0.0093 \\ 0.4936 & -0.6783 \\ 0.6034 & -0.4383 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} -1.0728 \\ 2.8014 \times 10^{-6} \\ -3.7686 \times 10^{-2} \\ 1.8394 \times 10^{-1} \\ -6.7461 \times 10^{-1} \end{bmatrix}, \quad \boldsymbol{w} = \begin{bmatrix} -1.7115 \\ 1.0000 \\ -0.3295 \\ 0.3312 \\ 0.3853 \end{bmatrix}, \quad b = -0.0608$$

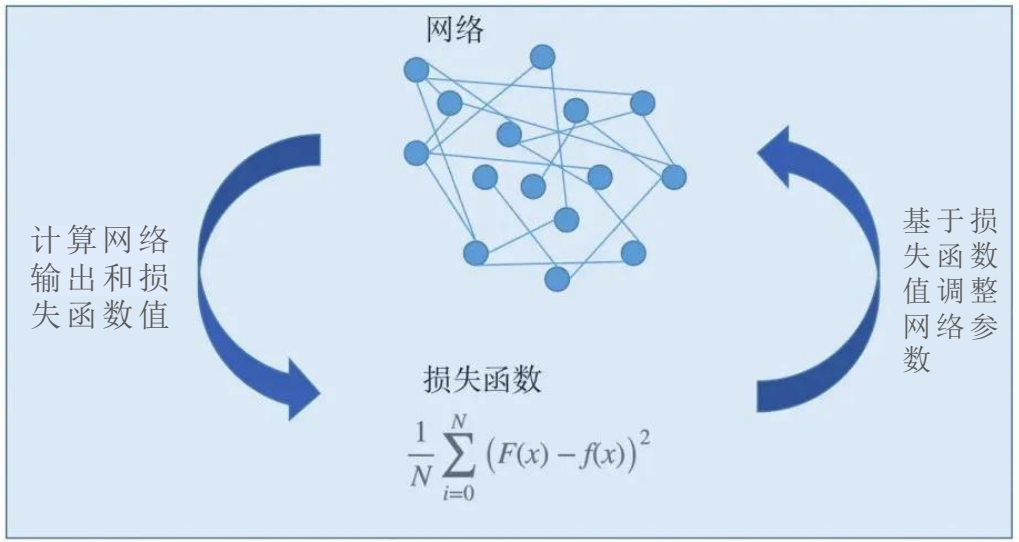
3.3 万能逼近定理

- 一个具有线性输出层和“挤压”性质的激活函数（例如sigmoid）的隐藏层的两层前馈神经网络，只要给予网络足够数量的隐藏单元（即网络足够宽），它可以以任意的精度来近似从一个有限维空间到另一个有限维空间的 任何Borel 可测函数
  - 定义在R^d空间有界闭集上的任意连续函数是 Borel 可测的
  - 前馈网络的导数也可以任意好地来近似函数的导数
  - 万能近似定理已经被证明对于更广泛类别的激活函数也是适用的，包括ReLU
- 通用近似定理意味着无论我们试图学习什么函数，只要有一个足够宽的前馈神经网络，一定能够表示这个函数
  - 然而，我们不能保证训练算法能够学到这个网络
  - 在最坏情况下，可能需要指数数量的隐藏单元
- 具有单隐藏层的前馈网络足以表示任何函数，但是网络层可能大得不可实现，并且可能无法训练出
- 近几年的工作表明浅层网络需要指数量级的隐藏层节点才能表示的问题，通过增加网络深度，可使网络所需节点数降低为线性级

3.4 损失函数

- 损失函数（loss function）用来衡量模型的预测值与样本真实值的偏差，可以反映预测的效果，对模型的学习具有指导作用
- 为什么需要损失函数？在训练过程中，通过调整模型参数来最小化损失函数的值，提高预测的准确性





3.4.1 均方误差损失

- 均方误差损失函数（Mean Square Error，MSE）通过计算模型预测值$\hat{y}$与真实值y之间的平方差来衡量误差，也称为平方L_2损失

$$L_2 \text{ loss} = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

  - $\hat{y}, y_i$为第i个样本的预测值和真实值，n为样本数
  - MSE在任何地方都是可微的，从而可以在训练过程中使用基于梯度的优化方法

$$\frac{\partial L_2 \text{ loss}}{\partial \hat{y}_i} = \frac{2}{n}(\hat{y}_i - y_i)$$

  - MSE对误差较大的异常值较为敏感
    - 一个很大的数平方之后会进一步放大

$$\frac{1}{4}\left(0.01^2 + 0.03^2 + 0.02^2 + 0.01^2\right) = 0.000375$$

$$\frac{1}{4}\left(0.01^2 + 0.03^2 + 0.02^2 + 10^2\right) = 25.00035$$

3.4.1 均方误差损失

- PyTorch API

API: torch.nn.MSELoss(size_average=None, reduce=None, reduction='mean')

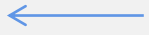
参数： size_average、reduce：一般使用默认设置；

reduction：当为 none 时，返回向量（或张量）$\begin{bmatrix} (\boldsymbol{y}_1 - \hat{\boldsymbol{y}}_1)^2 \\ \vdots \\ (\boldsymbol{y}_n - \hat{\boldsymbol{y}}_n)^2 \end{bmatrix}$；当为 mean 时，返回 $\frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i)^2$；

当为 sum 时，返回 $\sum_{i=1}^{n}(\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i)^2$，默认为 mean。

示例：
```
loss = nn.MSELoss()
input = torch.randn(3, 4, requires_grad=True)
target = torch.randn(3, 4)
output = loss(input, target)
```


可以想象为模型输出


可以想象为样本标签

3.4.2 平均绝对误差损失

- 平均绝对误差损失函数 (Mean Absolute Error, MAE) 通过计算模型预测值$\hat{y}$与真实值y之间的差的绝对值来衡量误差，也称为L_1损失

$$L_1 \text{ loss} = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i|$$

  - 相比于MSE，MAE对异常值不太敏感

$$\frac{1}{4}\left(0.01^2 + 0.03^2 + 0.02^2 + 0.01^2\right) = 0.000375$$

$$\frac{1}{4}\left(0.01^2 + 0.03^2 + 0.02^2 + 10^2\right) = 25.00035$$

相差66668倍

$$\frac{1}{4}(0.01 + 0.03 + 0.02 + 0.01) = 0.0175$$

$$\frac{1}{4}(0.01 + 0.03 + 0.02 + 10) = 2.515$$

相差144倍

  - 但MAE在0点处是不可微的，增加了训练难度

3.4.2 平均绝对误差损失

- PyTorch API

API: torch.nn.L1Loss(size_average=None, reduce=None, reduction='mean')

参数：size_average、reduce：一般使用默认设置；

reduction：当为 none 时，返回向量（或张量）$\begin{bmatrix} \boldsymbol{y}_1 - \hat{\boldsymbol{y}}_1 \\ \vdots \\ \boldsymbol{y}_n - \hat{\boldsymbol{y}}_n \end{bmatrix}$；当为 mean 时，返回 $\frac{1}{n}\sum_{i=1}^{n}\left|\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i\right|$；

当为 sum 时，返回 $\sum_{i=1}^{n}\left|\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i\right|$，默认为 mean。

示例：
```
loss = nn.L1Loss()
input = torch.randn(3, 4, requires_grad=True)
target = torch.randn(3, 4)
output = loss(input, target)
```

3.4.3 平滑L_1损失

- 平滑L_1损失(Smooth L1 Loss), 也称为Huber损失，提供了一种在L_1损失与平方L_2损失之间的平滑过渡
  - 当预测值与真实值之间的差异较小时，退化为平方L_2损失
  - 当差异较大时，退化为L_1损失

$$\ell_i = \begin{cases} |y_i - \hat{y}_i| - \frac{\beta}{2}, & if |y_i - \hat{y}_i| \geq \beta \\ \frac{1}{2\beta}(y_i - \hat{y}_i)^2, & if |y_i - \hat{y}_i| < \beta \end{cases}$$

$$\text{Smooth } L_1 \text{ loss} = \frac{1}{n}\sum_{i=1}^{n} \ell_i$$

让分段函数连续

设计思想：
误差较小时退化为MSE损失，避免了MSE对误差较大的异常值较为敏感的不足；
误差较大时退化为MAE损失，避免了MAE在0点处不可微的不足

  - 平滑L_1损失可能对参数β 较为敏感，使用时需要必要的调参
  - 在一些深度学习应用中，简单的MSE或MAE可能就够了，不需要复杂的平滑L_1损失

3.4.3 平滑L_1损失

- PyTorch API

API: torch.nn.SmoothL1Loss(size_average=None, reduce=None, reduction='mean', beta=1.0)

参数： size_average、reduce：一般使用默认设置；

reduction：当为 none 时，返回向量（或张量）$\begin{bmatrix} \ell_1 \\ \vdots \\ \ell_n \end{bmatrix}$；当为 mean 时，返回 $\frac{1}{n}\sum_{i=1}^{n}\ell_i$；

当为 sum 时，返回 $\sum_{i=1}^{n}\ell_i$，默认为 mean；

beta：式中的 β，默认值为 1.

示例：
```
loss = nn.SmoothL1Loss()
input = torch.randn(3, 4, requires_grad=True)
target = torch.randn(3, 4)
output = loss(input, target)
```

3.4.4 交叉熵损失

- 对于回归问题，L_1损失、平方L_2损失与平滑L_1损失函数较为常用
- 对于分类问题，交叉熵损失函数更为常见
  - 熵使用对数函数来量化数据中的信息内容，熵越高，表示传输的信息越多，熵越少，表示传输的信息越少
  - 交叉熵损失函数通过计算模型预测概率分布与真实概率分布之间的差异来衡量误差

3.4.4 交叉熵损失

- 当有多个类别时，每个样本 y_i使用独热编码（one-hot coding），即 $y_i = (y_{i,1}, y_{i,2}, \cdots, y_{i,d}) = (0, 0, 1, 0, \cdots, 0)$，只有第 r个位置为1（表示样本属于第 r类），其余位置都为0时

$$\text{Cross Entropy loss} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{d} y_{i,j}\log \hat{y}_{i,j}$$

i表示第i个样本，j表示第j个类别，n为样本数，d为类别数

$$= -\frac{1}{n}\sum_{i=1}^{n}\log \hat{y}_{i,r}$$

$y_{i,r}$为1，其余$y_{i,j}$为0

  - 交叉熵越小，预测分布越接近真实分布，模型性能越好
    - $y_{i,r} = 1$，如果 $\hat{y}_{i,r} = 1$，表示预测正确，此时 $-\log \hat{y}_{i,r} = 0$
    - 如果 $\hat{y}_{i,r} = 0.7$，表示预测该样本属于第 r 类的概率为0.7，此时 $-\log \hat{y}_{i,r} = 0.3567$
    - 如果 $\hat{y}_{i,r} = 0$，表示预测错误，此时 $-\log \hat{y}_{i,r} = \infty$
  - 对第i个样本，最小化交叉熵即为最大化 $\hat{y}_{i,r}$，即最大化模型输出的第 r类概率
    - 直观解释：如果看到了一个第 r类的样本，那么就调整模型参数，使得它判定这个样本属于第 r类的概率最大，这就是最大似然法

3.4.4 交叉熵损失

- 当有多个类别时，每个样本 y_i使用独热编码（one-hot coding），即 $y_i = (y_{i,1}, y_{i,2}, \cdots, y_{i,d}) = (0, 0, 1, 0, \cdots, 0)$，只有第 r个位置为1（表示样本属于第 r类），其余位置都为0时

$$\text{Cross Entropy loss} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{d} y_{i,j}\log \hat{y}_{i,j}$$

i表示第i个样本，j表示第j个类别，n为样本数，d为类别数

$$= -\frac{1}{n}\sum_{i=1}^{n}\log \hat{y}_{i,r}$$

$y_{i,r}$为1，其余$y_{i,j}$为0

- 负对数似然损失

$$\text{Negative Log-Likelihood Loss} = -\frac{1}{n}\sum_{i=1}^{n}\log \hat{y}_{i,r}$$

其中 r为样本所属类别

当使用one-hot coding时，交叉熵损失与负对数似然损失是等价的

3.4.4 交叉熵损失

- 当有多个类别时，每个样本 y_i使用独热编码（one-hot coding），即 $y_i = (y_{i,1}, y_{i,2}, \cdots, y_{i,d}) = (0, 0, 1, 0, \cdots, 0)$，只有第 r个位置为1（表示样本属于第 r类），其余位置都为0时

$$\text{Cross Entropy loss} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{d} y_{i,j}\log \hat{y}_{i,j}$$

i表示第i个样本，j表示第j个类别，n为样本数，d为类别数

$$= -\frac{1}{n}\sum_{i=1}^{n}\log \hat{y}_{i,r}$$

$y_{i,r}$为1，其余$y_{i,j}$为0

- 当只有两个类别时，也可以使用二类交叉熵

$$\text{Binary Cross Entropy loss} = -\frac{1}{n}\sum_{i=1}^{n}(y_i \log \hat{y}_i + (1 - y_i)\log(1 - \hat{y}_i))$$

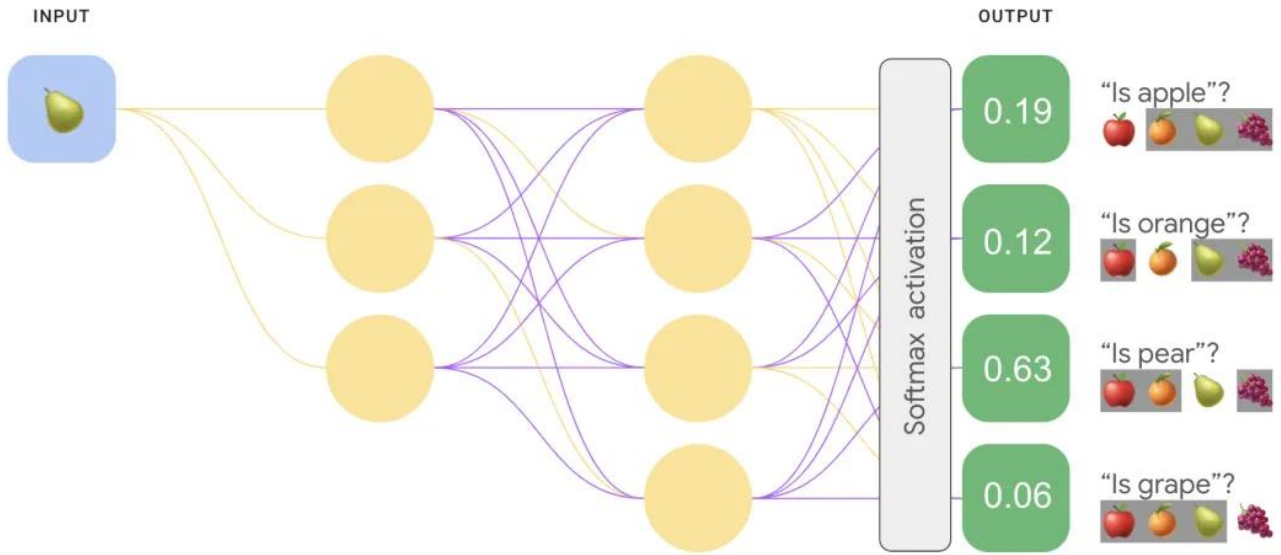
其中 $y_i = 1$或 $y_i = 0$

二类交叉熵是多类交叉熵的一个特例

3.4.4 交叉熵损失

- 深度学习中，一般将交叉熵与softmax函数结合使用
  - 神经网络最后一层的输出 h 一般为一组连续的数值，难以直观判断其实际意义
  - 为了实现分类任务，需要将输出转化为一组能够进行分类判断的数值
  - 具体地，我们希望得到每种类别的概率，同时要求概率之和为1，从而方便比较每个类别的概率大小
- softmax函数通常用于将一组任意范围的实数转换为近似的概率分布
  - 归一化函数，利用指数函数将一组实数值映射到零到正无穷之间，然后进行归一化处理，得到近似的概率分布

$$\mathrm{Softmax}(h)_i = \frac{\exp(h_i)}{\sum_j \exp(h_j)}$$

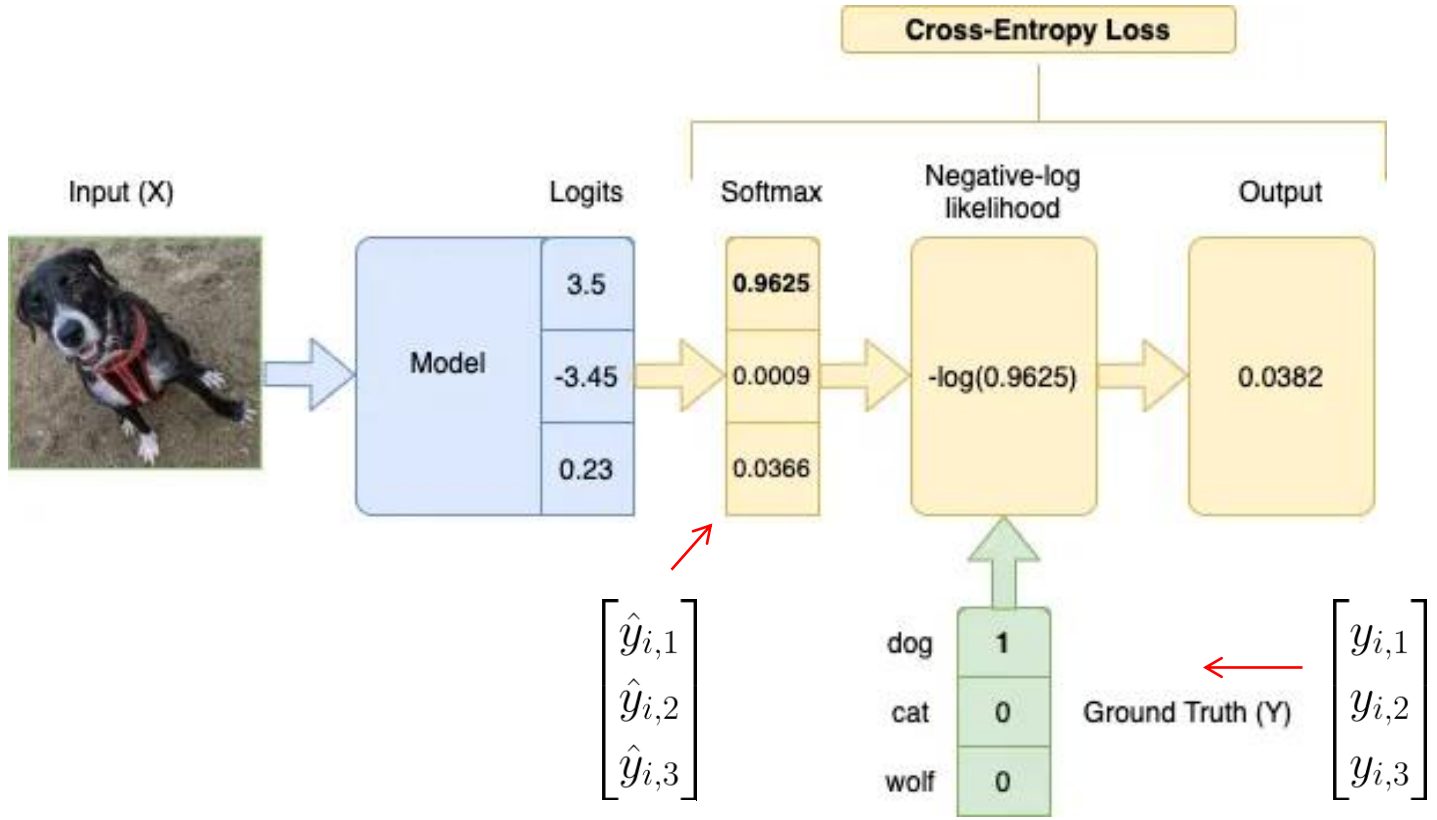


3.4.4 交叉熵损失

- 交叉熵和softmax函数是神经网络中最常用的组件之一
  - 通过softmax函数将网络的输出h转化为近似的概率分布$\hat{y}$，再使用交叉熵作为损失函数衡量$\hat{y}$和真实概率分布y之间的差异，交叉熵函数值越小，模型分类的准确程度就越高

Cross Entropy loss of sample i

$$
\begin{aligned}
&= -\sum_{j=1}^{d} y_{i,j} \log \hat{y}_{i,j} \\
&= -\left(1 \times \log(0.9625) + 0 \times \log(0.0009) + 0 \times \log(0.0366)\right) \\
&= -\log(0.9625) \\
&= 0.0382
\end{aligned}
$$



3.4.4 交叉熵损失

- PyTorch API

API：torch.nn.Softmax(dim=None)
参数：dim：指定哪一维和为 1，当输入为矩阵时，dim=0 表示每一列和为 1，
dim=1 表示每一行和为 1。

API: torch.nn.CrossEntropyLoss(weight=None, size_average=None, ignore_index=-100,
reduce=None, reduction='mean', label_smoothing=0.0)

参数：weight、size_average、ignore_index、reduce：一般使用默认设置；

reduction：输入 $n \times c$ 矩阵 $\boldsymbol{Y}$ 和 $\hat{\boldsymbol{Y}}$，n 表示批量大小或样本数，c 表示类别数

$\boldsymbol{Y}$ 每一行的和为 1，计算 $\ell_i = -\sum_{j=1}^{c} \boldsymbol{Y}_{i,j} \log \frac{\exp(\hat{\boldsymbol{Y}}_{i,j})}{\sum_{r=1}^{c} \exp(\hat{\boldsymbol{Y}}_{i,r})}$

当为 none 时，返回向量（或张量）$\begin{bmatrix} \ell_1 \\ \vdots \\ \ell_n \end{bmatrix}$；当为 mean 时，返回 $\frac{1}{n}\sum_{i=1}^{n} \ell_i$；

当为 sum 时，返回 $\sum_{i=1}^{n} \ell_i$，默认为 mean；

label_smoothing：0 到 1 之间的平滑值，默认为 0。

API: torch.nn.BCELoss(weight=None, size_average=None, reduce=None, reduction='mean')

参数：同上，二类交叉熵。

示例：
```
loss = nn.CrossEntropyLoss()
input = torch.randn(3, 5, requires_grad=True)
target = torch.randn(3, 5).softmax(dim=1)        # 保证 target 每一行的和为 1
output = loss(input, target)
```

- PyTorch中的CrossEntropyLoss实现了对 $\hat{Y}$ 的softmax操作，不需要再对 $\hat{Y}$ 调用Softmax函数
- 不需要对input（即 $\hat{Y}$）调用Softmax，但需要对target（即 Y）调用softmax
- input对应网络输出，target对应样本标签

3.4.5 KL散度

- KL散度损失（Kullback-Leibler Divergence Loss）用于衡量不同的分布之间的信息损失，也称为相对熵损失，广泛应用于聚类分析与参数估计等任务中

$$\text{KL Loss} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{d}\left[y_{i,j}\log\hat{y}_{i,j} - y_{i,j}\log y_{i,j}\right]$$

i表示第i个样本，j表示第j个类别，n为样本数，d为类别数

  - 回顾交叉熵损失：

$$\text{Cross Entropy loss} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{d} y_{i,j}\log\hat{y}_{i,j}$$

  - 当用作深度学习损失函数时，由于$y_{i,j}\log y_{i,j}$不随网络参数变化而变化（$y_{i,j}$为样本真实标签），因此对KL散度求导和对交叉熵求导的结果是一样的，使用交叉熵作为损失函数和使用KL散度作为损失函数在指导训练时是等价的

3.4.5 KL散度

- PyTorch API

```
API: torch.nn.KLDivLoss(size_average=None, reduce=None, reduction='mean',
                        log_target=False)
参数：size_average、reduce、log_target：一般使用默认设置；
      reduction：可取 none、mean、batchmean、sum，默认为 mean。
函数说明：为了防止下溢问题，假设 input 属于 log 空间；
          当 log_target 为 true 时，target 也属于 log 空间。
          因此函数执行如下操作：
          if not log_target:                                  # default
              loss_pointwise = target * (target.log() - input)
          else:
              loss_pointwise = target.exp() * (target - input)
          if reduction == "mean":                             # default
              loss = loss_pointwise.mean()
          elif reduction == "batchmean":
              loss = loss_pointwise.sum() / input.size(0)
          elif reduction == "sum":
              loss = loss_pointwise.sum()
          else:                                               # reduction == "none"
              loss = loss_pointwise
示例：import torch.nn.functional as F
      kl_loss = nn.KLDivLoss(reduction="mean")
      input = F.log_softmax(torch.randn(1, 5, requires_grad=True), dim=1)   # 输入属于 log 空间
      target = F.softmax(torch.rand(1, 5), dim=1)
      output = kl_loss(input, target)
```

3.4.6 其他损失函数

- PyTorch也实现了nn.CTCLoss、nn.NLLLoss、nn.PoissonNLLLoss、nn.GaussianNLLLoss、nn.BCEWithLogitsLoss、nn.MarginRankingLoss、nn.HingeEmbeddingLoss、nn.MultiLabelMarginLoss、nn.HuberLoss、nn.SoftMarginLoss、nn.MultiLabelSoftMarginLoss、nn.CosineEmbeddingLoss、nn.MultiMarginLoss、nn.TripletMarginLoss、nn.TripletMarginWithDistanceLoss等损失函数
- PyTorch官方文档https://pytorch.org/docs/stable/nn.html中的Loss Functions部分
- 深度学习中最常用的损失函数是均方误差损失和交叉熵损失（或负对数似然损失）

3.5 正则化

- 泛化性：需要深度学习模型在未观测到的输入上表现良好
- 正则化是指让学习算法降低泛化误差，而不仅仅是训练误差

3.5.1 权重衰减 好了解

- l_2 正则化
  - 使用最广泛的正则化技术之一，偏向于没有极端大权重的模型，对特征的微小扰动更鲁棒
    - 线性回归：$f(w,b) = w_1x_1 + w_2x_2 + \cdots + w_nx_n + b$，若 w_i 很大，则 x_i 的微小扰动对输出影响很大
    - 具体实现：$\min_{w,b} \text{loss}(f(w,b);y) + \frac{\lambda}{2}\sum_{l=1}^{n} w_i^2$

最小化损失函数，降低训练误差

l_2 正则化，希望权重不要太大，提高泛化能力

  - 深度学习训练过程中的目标函数：损失函数 + 正则化项

$$f(W,b) = \frac{1}{n}\sum_{i=1}^{n}\ell(y_i,\hat{y}_i) + \frac{\lambda}{2}\sum_{l=1}^{L}\|W_l\|_F^2$$

其中 $\|W\|_F^2 = \sum_i\sum_j W_{i,j}^2$

矩阵的F范数的平方是指矩阵所有元素的平方和

3.5.1 权重衰减

- l_2 正则化
  - 深度学习训练过程中的目标函数：交叉熵损失 + 正则化

$$f(W, b) = \frac{1}{n}\sum_{i=1}^{n} \ell(y(i), \hat{y}(i)) + \frac{\lambda}{2}\sum_{l=1}^{L} \| W_l \|_F^2$$

- 权重衰减
  - 当使用梯度下降最小化上述目标函数时

$$\begin{aligned} W^{k+1} &= W^k - \eta \nabla_W f(W^k, b^k) \\ &= W^k - \eta \left(\frac{1}{n}\sum_{i=1}^{n} \nabla_W \ell(y_i, \hat{y}_i) + \lambda W^k \right) \\ &= (1 - \eta\lambda) W^k - \frac{\eta}{n}\sum_{i=1}^{n} \nabla_W \ell(y_i, \hat{y}_i) \end{aligned}$$

$$\| W \|_F^2 = \sum_i \sum_j W_{i,j}^2$$

$$\frac{\partial \| W \|_F^2}{\partial W} = W$$

  - W^k 前面的 $1 - \eta\lambda$ 起到权重衰减的作用
    - W^k 乘以一个小于1的因子，表示对 W^k 衰减
    - 如果梯度项始终为0，那么 $W^{k+1} = (1 - \lambda\eta)^k W^1 \approx 0$，即权重衰减为0

3.5.2 Dropout

根先念 计算 作用

- 在深度网络的训练过程中，通过随机丢弃一部分神经元（同时丢弃其对应的连接边）来避免过拟合的做法，称为 dropout
  - 实现方式：设置一个固定的概率 p，对于每一个神经元都以一个概率 p 来判断是否丢弃
    - 靠近输入层的地方一般设置较低的 dropout 概率
  - 在反向传播时，与被丢弃神经元相关的权重的梯度为0
- 在测试阶段，一般不丢弃神经元，所有神经元都激活

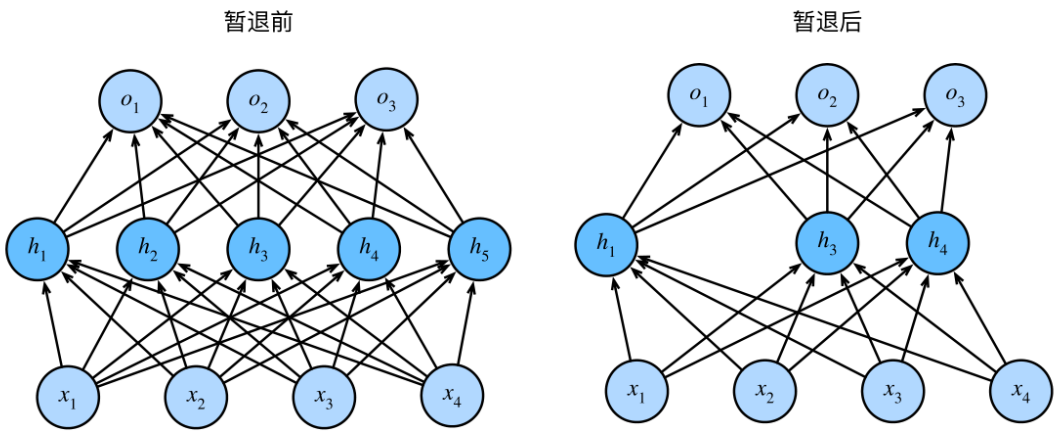


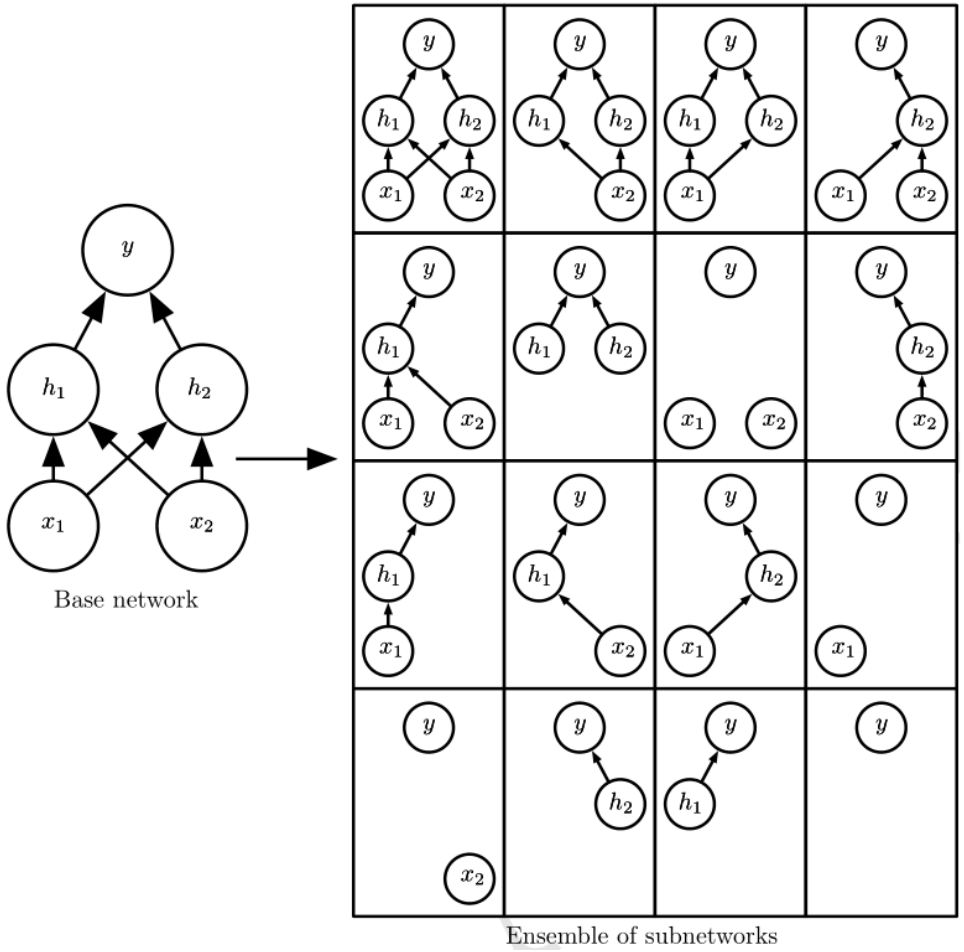
图4.6.1: dropout前后的多层感知机

3.5.2 Dropout

- 直观解释1：在训练中每个隐藏层神经元都有可能被以 p 的概率丢弃，这样就减少了神经元之间的依赖性，输出层的计算也无法过度依赖任何一个神经元，从而减少过拟合
  - 网络中任一神经元在每次训练中均有可能不起作用，从而迫使神经网络学会在不依赖某些特定神经元的情况下仍能稳定地工作

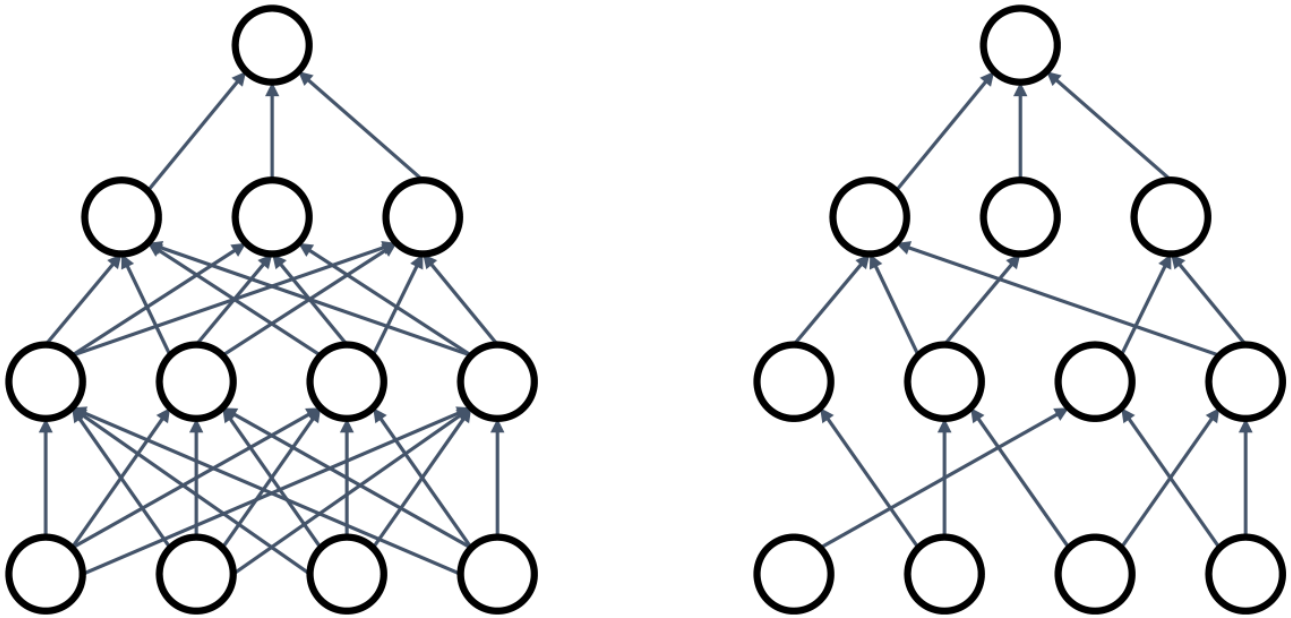
3.5.2 Dropout

- 直观解释2：集成学习角度
- 集成学习：分别训练几个不同的模型，然后让所有模型投票表决样本的输出
- Dropout训练的集成包括从原始网络移除部分神经元形成的所有可能的子网络
- 在训练过程中每做一次dropout，就相当于从原始网络中采样一个不同的子网络进行训练
    - 通过dropout，相当于在结构多样性的多个网络模型上进行训练，最终的网络可以看做是不同结构的网络的集成模型



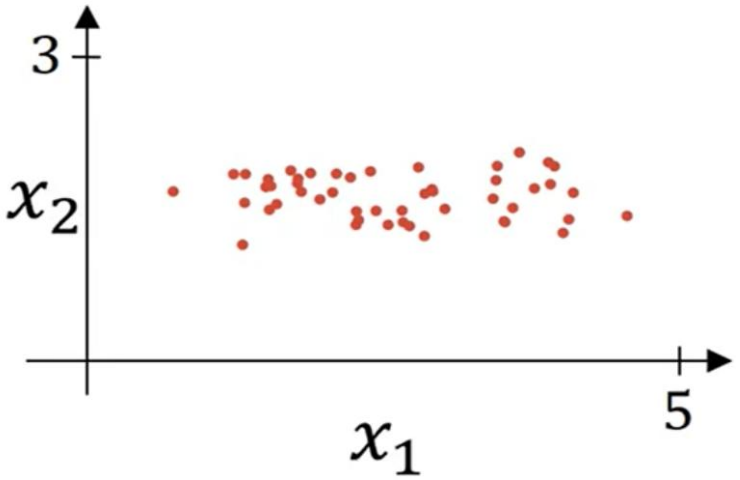
3.5.2 DropConnect

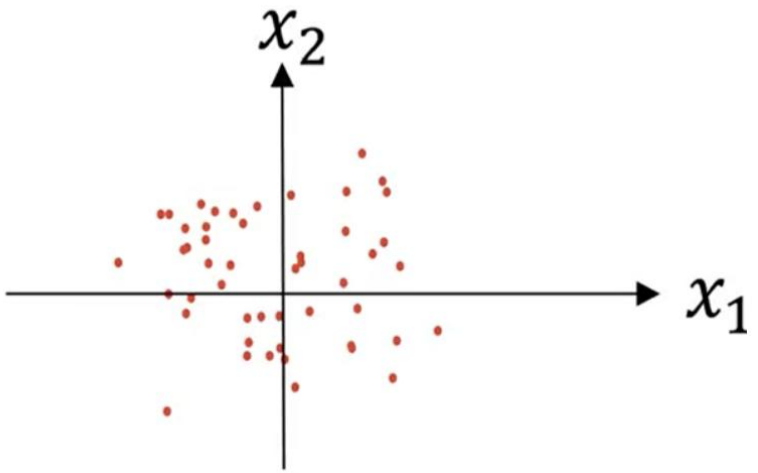
- DropConnect：训练阶段随机丢弃神经元之间的链接
- 测试阶段不丢弃，使用所有原始链接
- 可视为dropout的类比扩展



3.5.3 BatchNorm ☆去年考了

- 动机：数据预处理时，一般会进行标准化处理，使其平均值为0，方差为1
- 当训练深度网络时，进过若干层的传播，中间层的变量可能具有很大的变化范围，是否可以引入数据预处理时的标准化操作？
- 批量规范化（batch normalization，batchNorm，BN）：每次训练迭代中，对每层神经元的输入进行规范化，即减去其均值并除以其标准差，将输入值越来越偏的分布拉回到标准正态分布
  - 将数据的数值范围缩放到一个特定的尺度，消除不同特征之间的量纲差异





3.5.3 BatchNorm

- 记小批量样本B中第r个样本的输入为h^r，第i个特征（或神经元的值）为h_i^r， BatchNorm执行如下操作：

$$\mu_i = \frac{1}{|B|}\sum_{r\in B} h_i^r, \forall i$$

$$\sigma_i = \sqrt{\frac{1}{|B|}\sum_{r\in B}(h_i^r - \mu_i)^2, \forall i}$$

计算小批量样本上每个特征的均值和标准差

$$\hat{h}_i^r = \gamma_i \frac{h_i^r - \mu_i}{\sigma_i} + \beta_i, \forall i$$

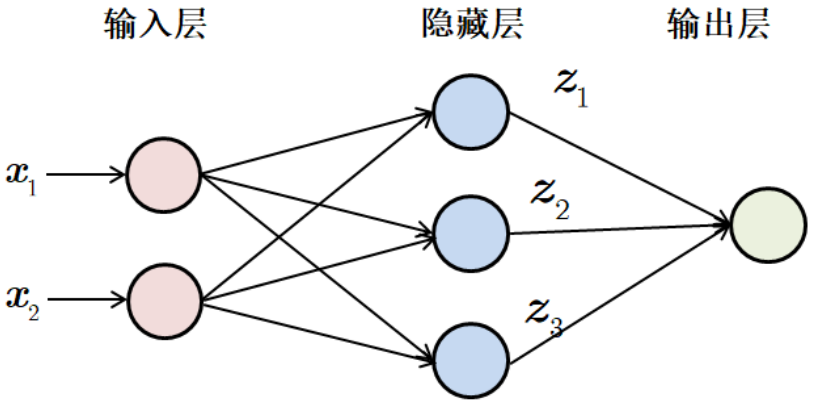
将每一个样本的每个特征归一化
乘以伸缩因子γ_i，加上偏移因子β_i

$$BN(h) = \hat{h}$$

  - 0均值1方差是一个主观的选择，因此需要伸缩因子和偏移因子
    - 二者是需要学习的参数，而不是超参数
    - 从0均值1方差的分布变为β均值γ方差的分布

3.5.3 BatchNorm

- BatchNorm示例



5个样本

x_1	x_2	z_1	z_2	z_3
2	5	2	5	3
1	3	3	4	6
3	5	2	1	6
4	2	1	3	4
3	6	3	6	7
μ		2.2	3.8	5.2
σ		0.8	1.9	1.6

按列求均值
和均方差

$$\tilde{z} = \frac{z - \mu}{\sigma}$$

-0.2	0.6	-1.3
0.9	0.1	0.5
-0.2	-1.4	0.5
-1.4	-0.4	-0.7
0.9	1.1	1.1

$$\gamma\tilde{z} + \beta$$

BN
输
出

3.5.3 BatchNorm

- 测试阶段，μ和σ可以使用训练阶段所有小批量的μ和σ的均值
  - 可以使用指数加权平均，前期迭代权重低，后期迭代权重高
- 在全连接层， BatchNorm一般置于线性变换和激活函数之间

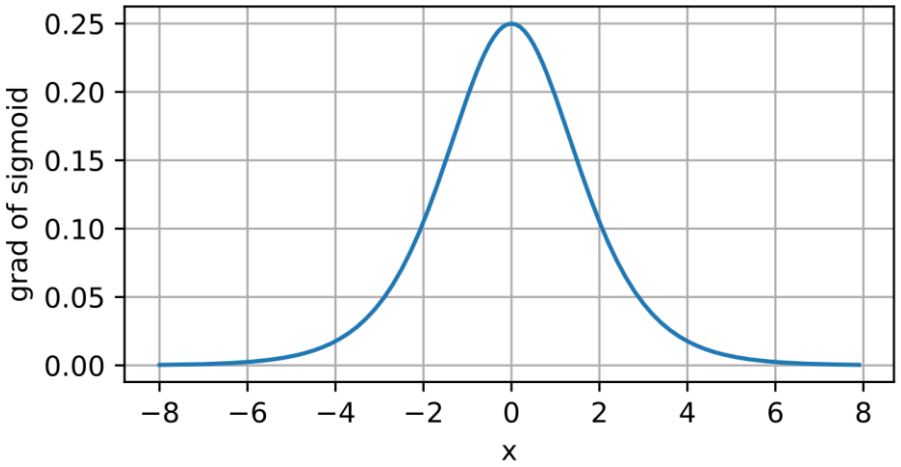
$$h = \sigma\left(\mathrm{BN}\left(Wx + b\right)\right)$$

3.5.3 BatchNorm

- BatchNorm通过对每一层做规范化，避免了极端大小的输出出现，即使是对于sigmoid，也可以缓解梯度消失

$$h = \sigma\left(\mathrm{BN}\left(Wx + b\right)\right)$$

- BatchNorm减小了不同层之间参数的依赖关系，使模型更鲁棒
  - 没有BatchNorm层的话，当前面层的输出范围发生变化时，后面层也会受到很大影响。 BatchNorm通过对每一层输出做标准化，从而避免了该情况发生
- 第4、7讲将从训练的角度继续讲解BatchNorm的作用



3.5.3 LayerNorm 一样重要的

- 层规范化（Layer normalization，LayerNorm，LN）
  - BatchNorm使用小批量样本对每一个维度的特征进行归一化
  - LayerNorm对单个样本以层为单位，对每一层的所有维度的特征进行归一化
  - 记第l层的第i个特征（或神经元的值）为h_i^l，第l层的特征总数为n_l， LayerNorm执行如下操作：

$$\mu^l = \frac{1}{n_l}\sum_{i=1}^{n_l} h_i^l, \forall l$$

$$\sigma^l = \sqrt{\frac{1}{n_l}\sum_{i=1}^{n_l}(h_i^l - \mu^l)^2}, \forall l$$

计算第l层的特征的均值和方差

$$\hat{h}_i^l = \gamma_i \frac{h_i^l - \mu^l}{\sigma^l} + \beta_i, \forall l$$

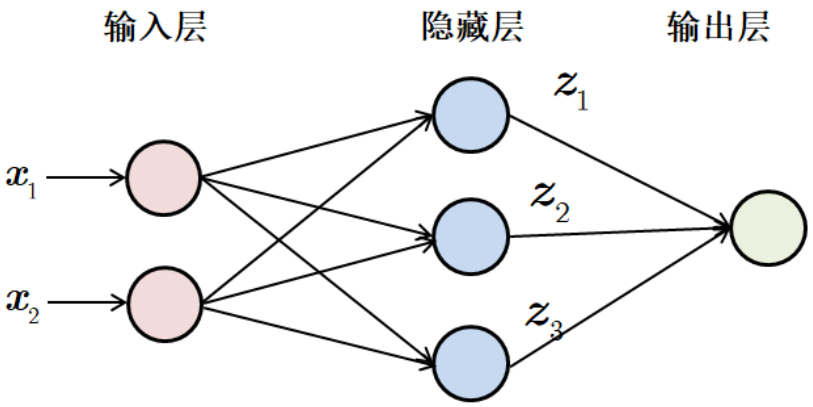
将每个特征归一化
乘以伸缩因子γ_i，加上偏移因子β_i

$$LN(h) = \hat{h}$$

  - LayerNorm测试/推理阶段和训练阶段的计算方式相同，不需要像BatchNorm那样保留训练阶段的移动平均用于测试阶段

3.5.3 LayerNorm

- LN示例



5个样本

x_1	x_2	z_1	z_2	z_3
2	5	2	5	3
1	3	3	4	6
3	5	2	1	6
4	2	1	3	4
3	6	3	6	7

μ	σ
3.3	1.5
4.3	1.5
3	2.6
2.7	1.5
5.3	2.1

按行求均值
和均方差

$$\tilde{z} = \frac{z - \mu}{\sigma}$$

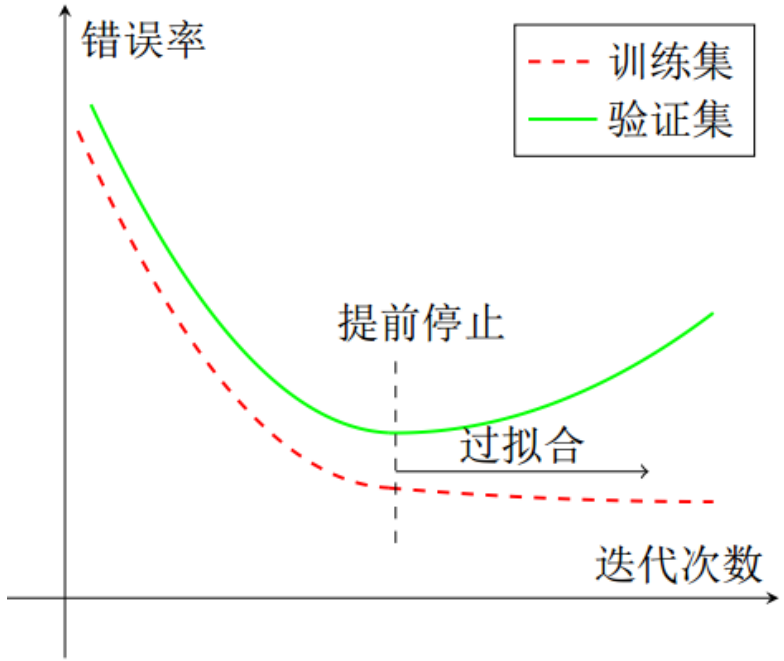
-0.9	1.1	-0.2
-0.9	-0.2	1.1
-0.4	-0.8	1.1
-1.1	0.2	0.9
-1.1	0.3	0.8

$\gamma\tilde{z} + \beta$

LN输出

3.5.4 提前终止

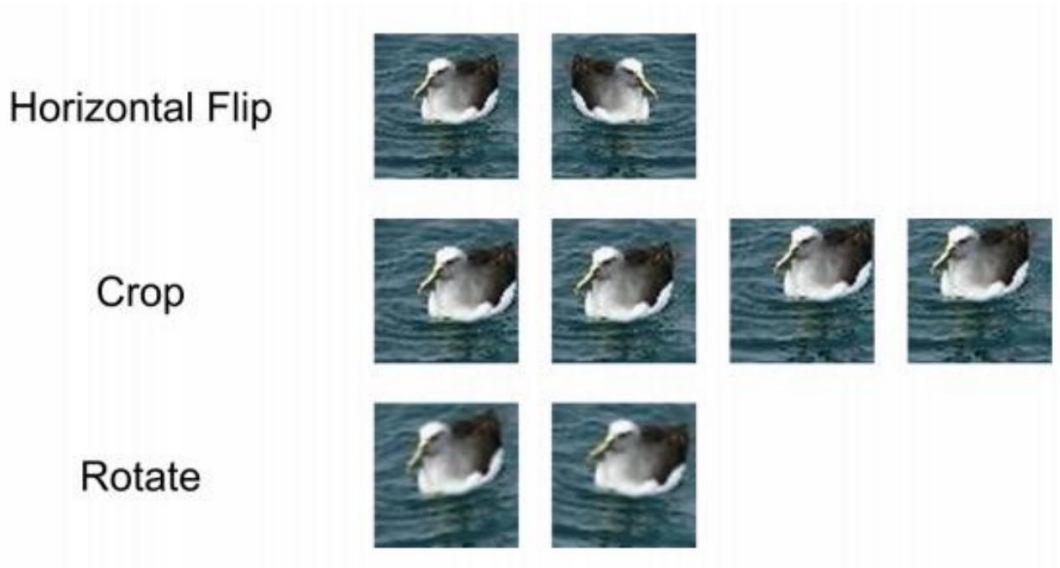
- 当模型有足够强的表示能力甚至会过拟合时，经常观察到训练误差会随着时间的推移逐渐降低，但验证集的误差会再次上升
  - 如果我们返回使验证集误差最低的参数设置，就可以获得更好的模型（因此，有希望获得更好的测试误差）



- 提前终止（early stop）
  - 超参数选择：训练步数是唯一超参数
  - 一个显著的代价是训练期间要定期评估验证集
  - 提前终止是非常不显眼的正则化形式，它几乎不需要对基本训练过程、目标函数或其他参数值进行改变

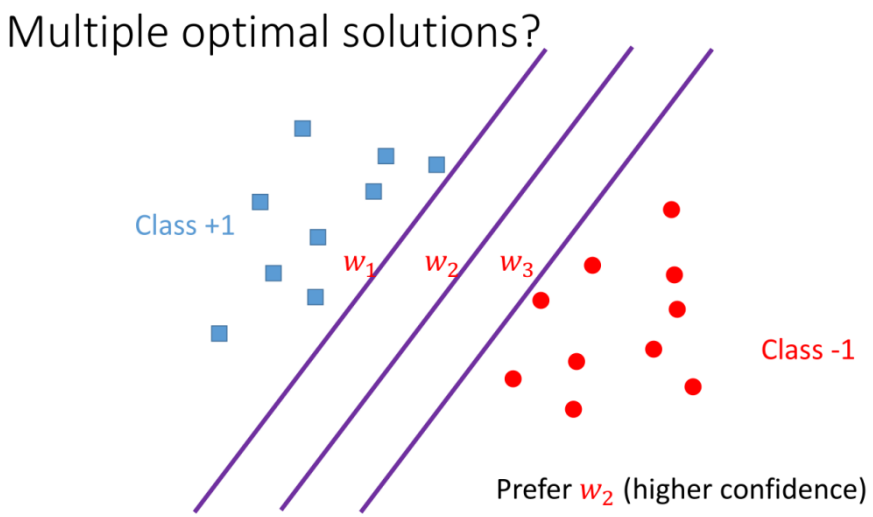
3.5.4 数据增强

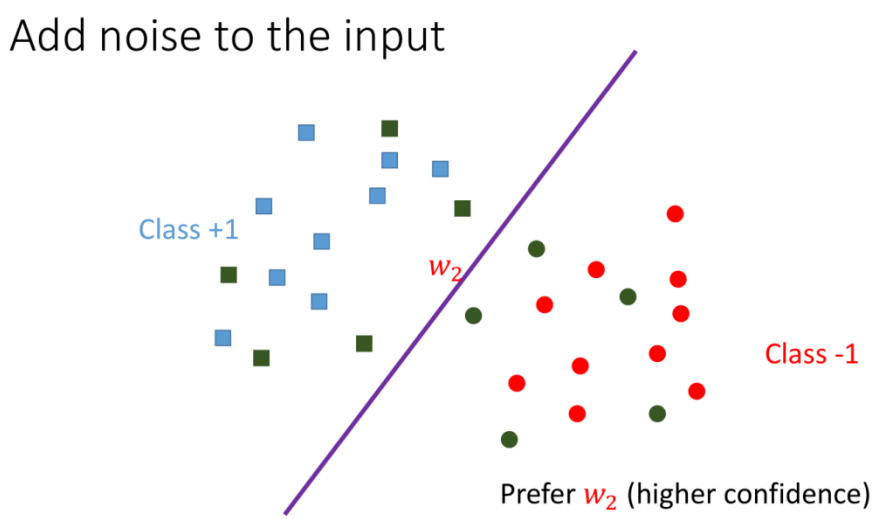
- 让机器学习模型泛化更好的最好办法是使用更多的数据进行训练
  - 在实践中，我们拥有数据量是有限的
- 解决这个问题的一种方法是创建“假”数据并把它添加到训练集
  - 将一张图像的一部分截出来放大
  - 很多机器学习任务要求对各种各样的变换保持不变
    - 图像翻转、平移、旋转、缩放
      - 翻转一般指左右翻转，很少把图像上下翻转
      - 必须要小心，不能使用改变正确类别的转换。例如，手写体识别任务需要识别 “b’’ 和 “d’’ 以及 “6’’ 和 “9’’，所以对这些任务来说，旋转 180 度就不是适当的数据集增强方式
  - 随机改变图像的亮度、对比度、色调，添加噪声



3.5.4 数据增强

- 在神经网络的输入层注入少量噪声也可以看作是数据增强的一种形式，即随机生成一些假样本
  - 模型应当对输入噪声鲁棒





- 向隐藏层施加噪声也是可行的，这可以被看作在抽象层上进行的数据集增强

总结

- 介绍了深度前馈网络的基本形式
- 介绍了常用的激活函数，包括ReLU、Leaky ReLU、softplus、maxout、sigmoid、tanh
- 两层非线性网络学习亦或问题
- 损失函数：L_2损失、L_1损失、交叉熵损失、KL散度
- 正则化：权重衰减、dropout、BatchNorm/LayerNorm、提前终止、数据增强