

3. 人工神经网络



提 纲

1. 深度学习

2. 前馈神经网络

① 准备数据

② 定义模型

③ 训练模型

④ 评估模型

- 加载数据集
- 数据预处理
- 数据集划分
- PyTorch张量

PyTorch

激活函数、损失函数、优化方法

BP算法、梯度消失、梯度爆炸

准确率、精确率、召回率、F1值

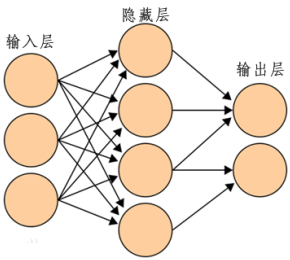
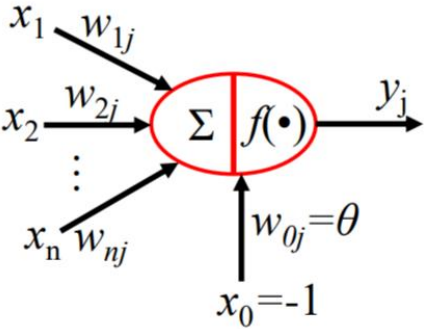
1

深度学习

Deep Learning

- 受生物学、神经科学启发，模拟生物神经网络的**数学模型**。如，感知器
- 一种**连接主义**模型：设计各种**网络结构**，使用不同的**学习方法**。
 - **连接主义**：认为AI起源于人脑，侧重**神经网络模型**、**连接机制**、**学习算法**。
 - **符号主义**：认为人类的思维过程可以用某种**符号来模拟、描述**，利用计算机进行知识的符号表征和逻辑推演，如 自动定理证明、专家系统。
 - **行为主义**：认为智能是系统**与环境的交互行为**，是对外界环境的一种适应。
目标是预见和控制行为，如 控制论，机器人，强化学习。
- 一种**高度非线性**模型，基本单元：**具有非线性激活函数的神经元**。
- 各个神经元之间连接的**权重**就是需要学习的参数，参数用各种优化算法求解。
- 早期强调模型的生物学合理性，现更关注对特定**认知能力**的模拟，如，物体识别。

连接主义 发展史



重新设计
BP算法，
解决学习
问题；
但存在严
重梯度消
失现象。

MLP
的万能
逼近定理
Robert Hecht-Nielsen

解决
梯度消失
问题
深度学习

M-P神经元
神经生理学家
McCulloch
数学家 Pitts

感知器
Rosenblatt

感知器
无法异或、
计算机能
力不足。

Minsky
1969年
(落)

用**BP算法**
建立
多层感知器
Paul Werbos

1981年

Hinton
1986年
(起)

支持
向量机
Vapnik
1969年
(落)

CNN
初始模型
LeNet
Yann LeCunn

1989年

LSTM
模型
Hochreiter

1997年

Hinton
2006年
(起)

深度学习

V.S.

传统机器学习

源于人工神经网络，含有**多个**隐藏层；
通过组合**低层特征**形成更加抽象**高层特征**，
对数据的特征进行多次**变换**；
用于分类、预测、推理 ...

本质：进行**特征**的**深层变换**。

关键：强大的**特征学习能力**。

- **表达能力强** 理论上可以模拟任意函数。
- **预测能力强** 明显高于浅层机器学习。
- **自动设计特征** 若有足够数据，学习效果好。
- **迁移性好** 精调下游任务的预测性能。

或者 不对特征进行变换，
或者 不生成新特征，如，决策树...
或者 对特征的变换层次较浅，如，支持向量机...

为什么要 深度学习？

1) AI 已经进入 DL 时代

- 人脸识别的应用
- Transformer、GPT在人类语言的应用：机器翻译、人机对话、AI写诗...
- 2020年DeepMind的AlphaFold2算法在蛋白质结构预测中达到人类水平

2) 促进 AI 与其他领域的交叉、融合

- 医学领域：医学图像自动分析
- 交通领域：汽车无人自动驾驶
- 环境领域：卫星、无人机航拍图像分析
- 能源领域、影视创作、天气预测

深度学习 发展史



解决
梯度消失
问题
深度学习

Geoffrey
Hinton

2006年

AlexNet
在
ImageNet
夺冠

Hinton

2012年
(爆发)

DeepFace
人脸识别
准确率97%

Facebook
(Meta)

2014年

残差网络
达150层
何恺明
MIT教授

证明损失
局部极值
可忽略

Hinton

2015年

AlphaGo
围棋击败
李世石

Google

2016年

Tranformer
Ashish Vaswani
Google

AlphaGoZero
击败
AlphaGo

Google

2017年

GPT-2

OpenAI

2018年

GPT-3

OpenAI

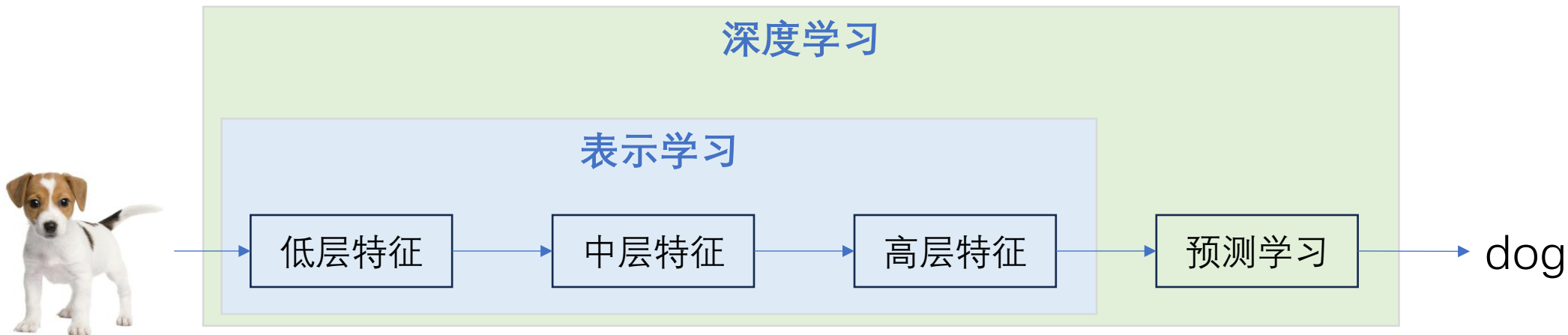
2020年

GPT-4

OpenAI

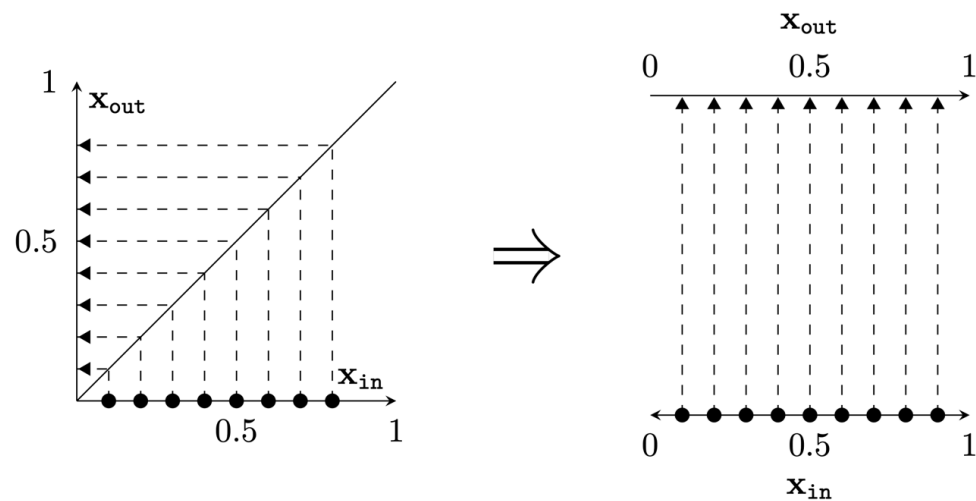
2023年

表示学习



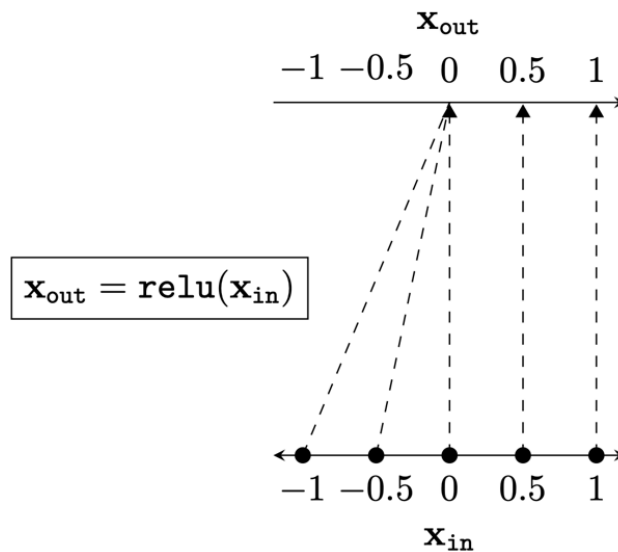
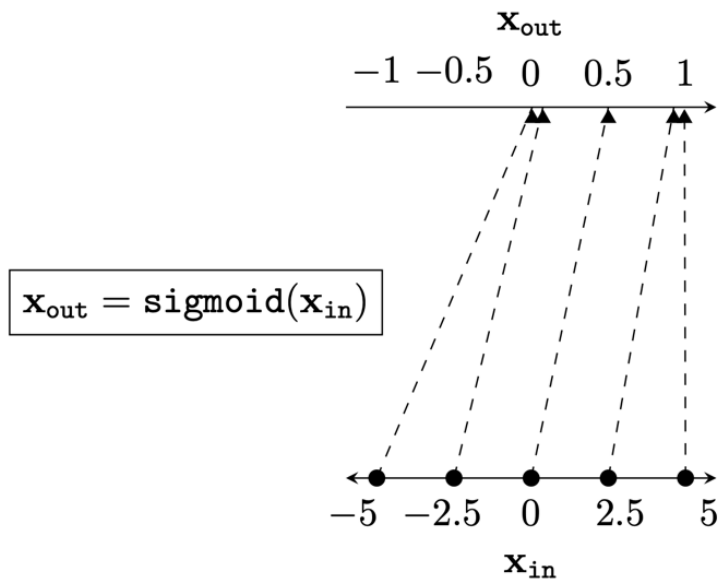
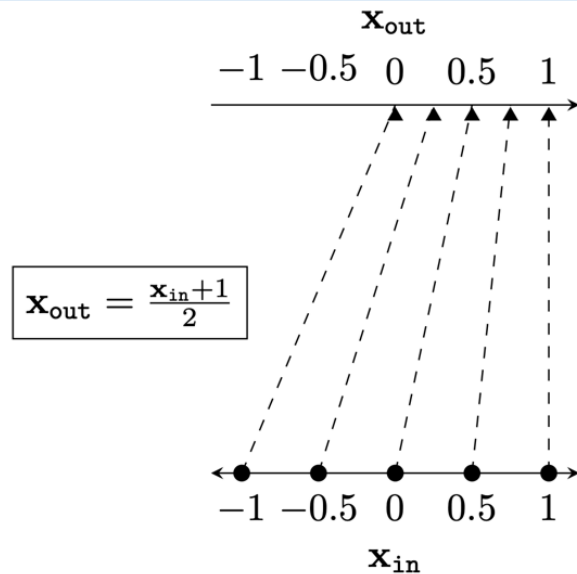
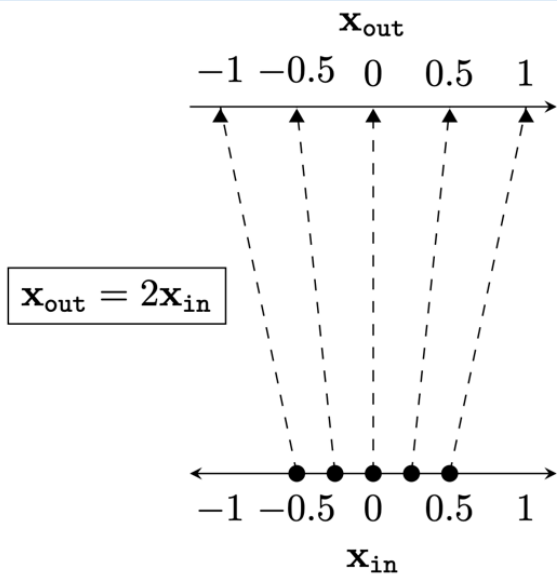
表示学习：如何自动从数据中学习好的表示。

数据表示：机器学习的核心问题。



同一函数的不同表达

数据表示 示例

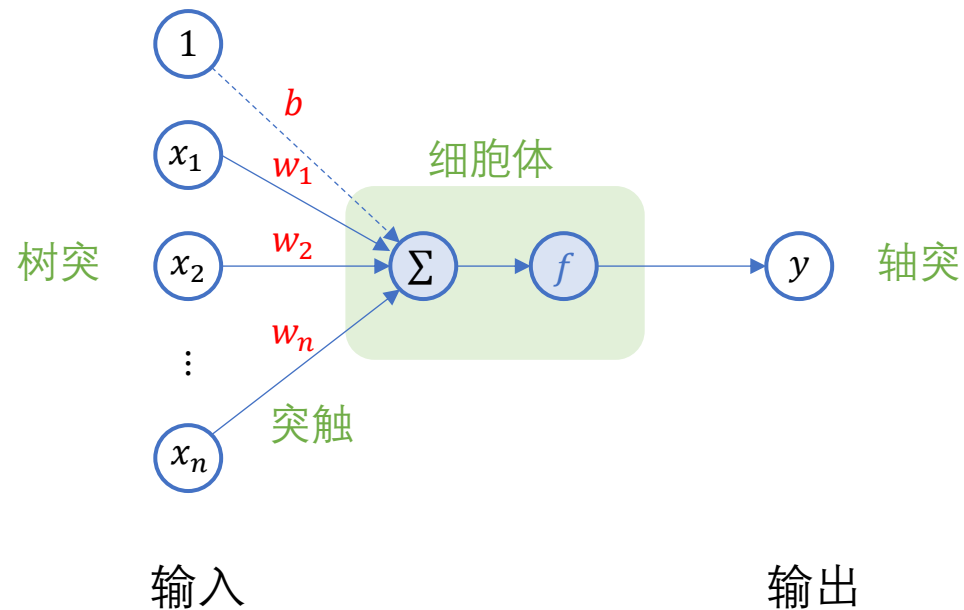
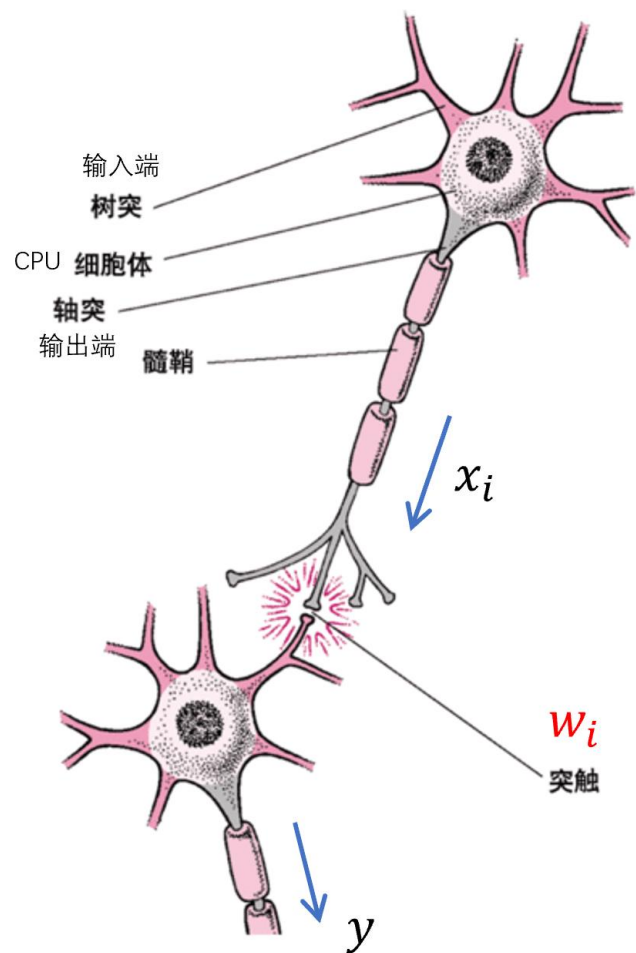


1

单层、多层感知器

前馈神经网络

神经元



$$y = f\left(\underbrace{\sum w_i x_i + b}_{\text{线性变换}}\right)$$

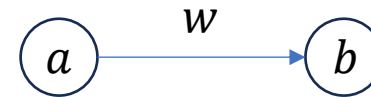
非线性变换

感知器

Perceptron

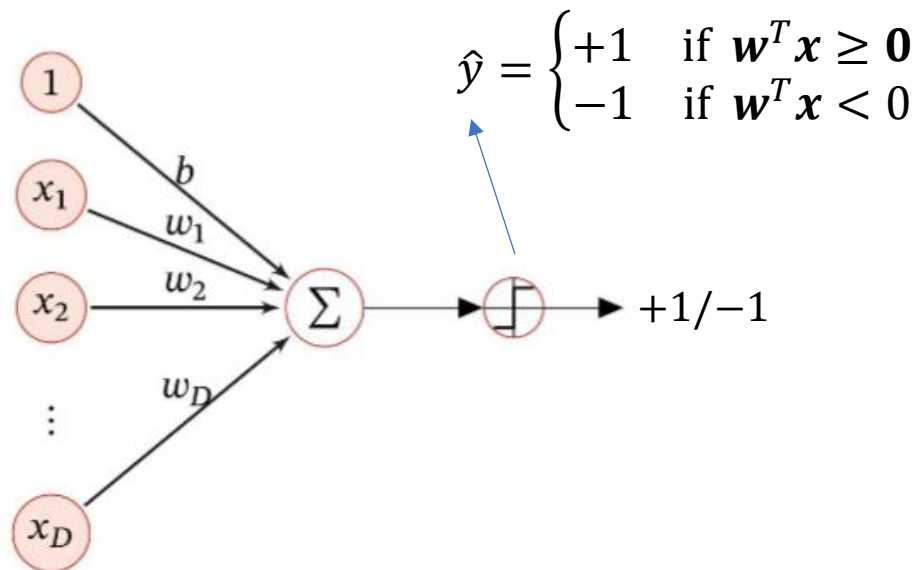
- 1943年，心理学家W. McCulloch、逻辑学家W. Pitts 提出人工神经元数学模型。
- 1949年，心理学家Donald Hebb 《行为的组织》提出人工神经元学习规则。

$$\Delta w = \eta \cdot a \cdot b$$

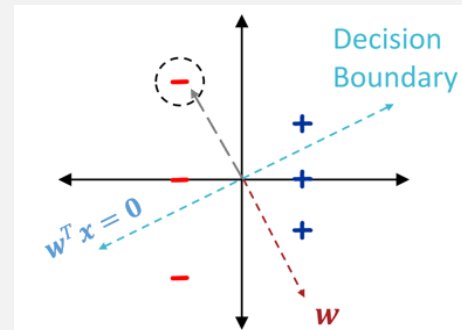


若一条突触两侧的两个神经元同时被激活，那么突触的强度将会增大。

- 1958年，神经学家 Frank Rosenblatt 提出模拟人类感知能力的机器：‘感知器’



错误驱动的在线学习算法

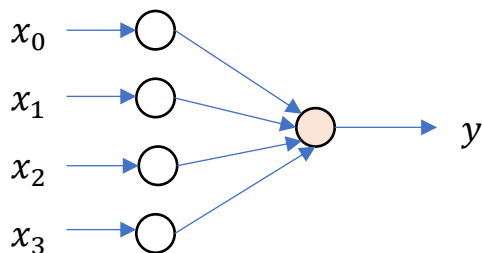


更新权重: $\mathbf{w} \leftarrow \mathbf{w} - \mathbf{x}^{(1)}$

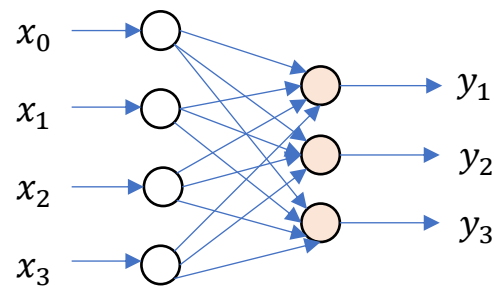
$$\mathbf{w} \leftarrow \mathbf{w} + y^{(1)} \mathbf{x}^{(1)}$$

感知器：单层神经网络

一种简单的神经网络，只拥有一层神经元。只能解决线性问题。

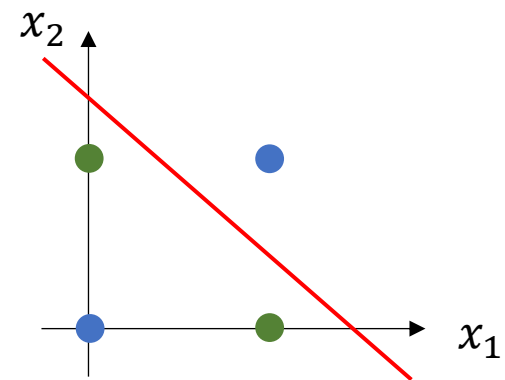


输入层 输出层



输入层 输出层

1969年, Marvin Minsky, 《Perceptrons》
不能解决异或等线性不可分问题。

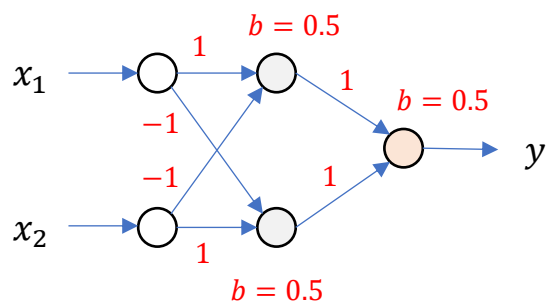


统计网络层数时，忽略输入层，因其没有计算。

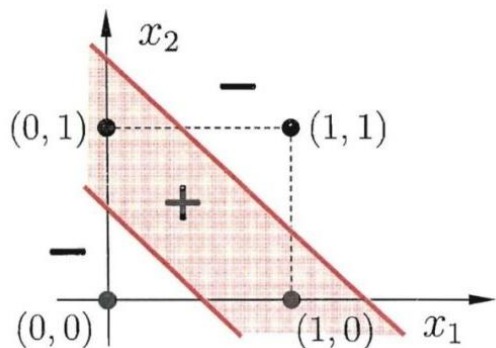
- 权重决定神经活动,
- 神经活动决定网络的输出,
- 网络的输出决定网络的误差。

两层感知器

- 解决非线性问题需用多层神经元。
- 两层感知器就可以解决异或问题。



输入层 隐藏层 输出层



```
from sklearn.neural_network import MLPClassifier

# 准备XOR数据, 共有4个点, 即4个样本
X = [[0, 0], [0, 1], [1, 0], [1, 1]] # 输入层2个特征2个神经元
y = [ 0,    1,    1,    0 ]         # 输出层1个神经元

# 创建MLP模型
model = MLPClassifier(
    hidden_layer_sizes=(2,), # 隐藏层有2个神经元
    activation='logistic',   # 激活函数
    solver='lbfgs',          # 优化器
    max_iter=10000           # 最大迭代次数

# 训练模型
model.fit(X, y)

# 测试模型
predictions = model.predict(X)

# 输出预测结果
print("预测结果:", predictions)

# 输出准确率
accuracy = model.score(X, y)
print(f"准确率: {accuracy:.0%}")
```

03-异或问题.ipynb

预测结果: [0 1 1 0]
准确率: 100%

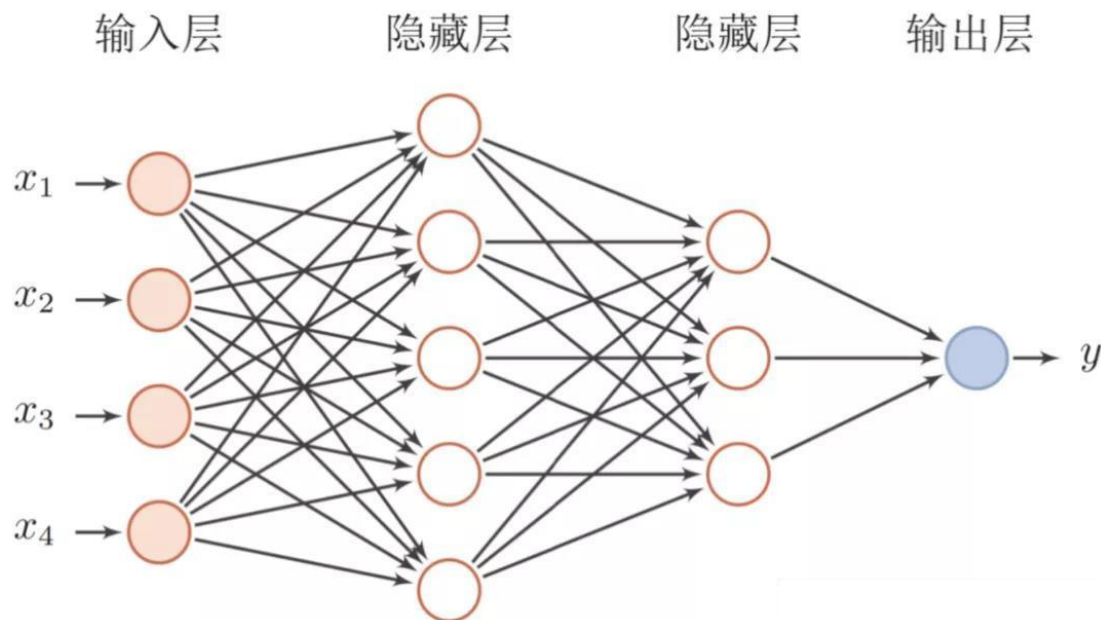
前馈神经网络

Feedforward Neural Network

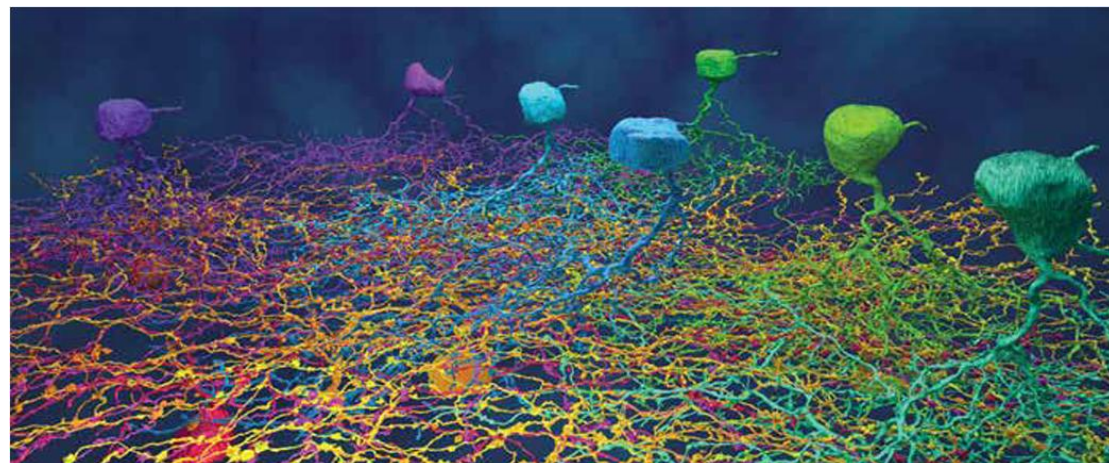
一种大规模的并行分布式处理器，天然具有**存储**并使用**经验知识**的能力。从两个方面模拟大脑：

- 1) 通过**学习**获取**知识**；
- 2) 内部神经元的**连接强度**，即突触权重，**存储知识**。

西蒙·赫金 (Simon Haykin) 1994



$$y = f^3(f^2(f^1(x)))$$



万能逼近定理

Universal Approximation Theorem

令 $\phi(\cdot)$ 是一个非常数、有界、单调递增的连续函数， J_D 是一个 D 维的单位超立方体 $[0, 1]^D$ ， $C(J_D)$ 是定义在 J_D 上的连续函数集合。对于任何一个函数 $f(x) \in C(J_D)$ ，存在一个整数 M ，和一组实数 v_l , $b_l \in \mathbf{R}$ 以及实数向量 $w_l \in \mathbf{R}^D$, $l = 1, 2, \dots, L$ ，以至于可以定义函数

$$F(x) = \sum_{l=1}^L v_l \phi(w_l^T x + b_l)$$

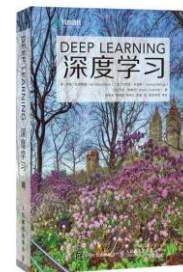
作为函数 $f(x)$ 的近似实现，即

$$|F(x) - f(x)| < \varepsilon, \quad \forall x \in J_D \quad \text{其中, } \varepsilon > 0 \text{ 是一个很小的正数。}$$

只说明了当隐含层神经元的数量足够多，神经网络能以任意精度去逼近一个给定的连续函数，但没有说明如何找到这样的网络，以及该网络是否最优。

“仅含有一层的前馈网络，的确足以有效地表示任何函数，但是，这样的网络结构可能会格外庞大，进而无法正确地学习和泛化。”

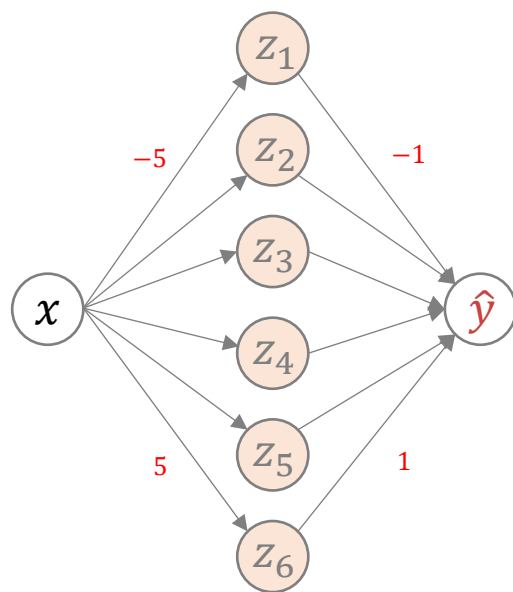
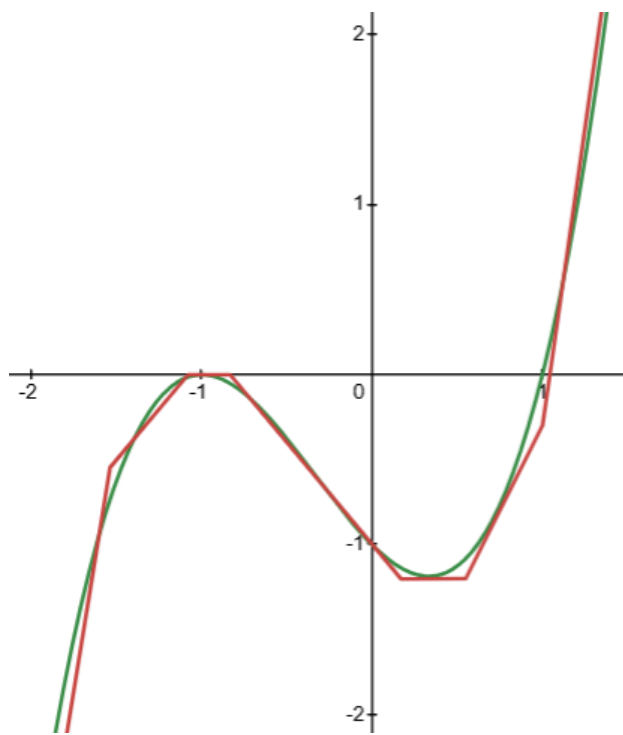
Ian Goodfellow, GAN网络



神经网络如何拟合一个函数 $f(x)$?

$$\hat{y} = -z_1(x) - z_2(x) - z_3(x) + z_4(x) + z_5(x) + z_6(x)$$

$$f(x) = x^3 + x^2 - x - 1$$



输入层

隐藏层

输出层

$$z_1(x) = \max(0, -5x - 7.7)$$

$$z_2(x) = \max(0, -1.2x - 1.3)$$

$$z_3(x) = \max(0, 1.2x + 1)$$

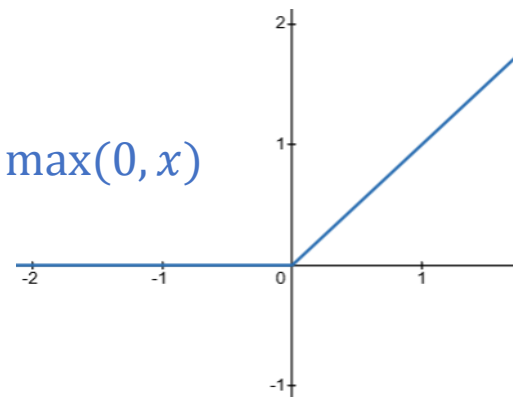
$$z_4(x) = \max(0, 1.2x - 0.2)$$

$$z_5(x) = \max(0, 2x - 1.1)$$

$$z_6(x) = \max(0, 5x - 5)$$

激活函数

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



03-前馈网络-PyTorch.ipynb

① 准备数据

代码：前馈神经网络 <1,2>

<1> 加载数据集

```
from sklearn import datasets
```

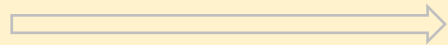
加载 yuān鸢尾花(iris)数据集

```
iris = datasets.load_iris()
```

4个特征：花萼长度、花萼宽度、花瓣长度、花瓣宽度

```
X = iris.data
```

```
print(X[0:5])
```

 # 输出前5行数据

```
[[5.1 3.5 1.4 0.2]
 [4.9 3.0 1.4 0.2]
 [4.7 3.2 1.3 0.2]
 [4.6 3.1 1.5 0.2]
 [5.0 3.6 1.4 0.2]]
```

花分类[0-setosa山鸢尾, 1-versicolor杂色鸢尾, 2-virginica维吉尼亚鸢尾]

```
y = iris.target
```

<2> 数据预处理


```
from sklearn.preprocessing import StandardScaler
```

使数据服从标准正态分布，均值为0，方差为1

```
scaler = StandardScaler()
```

```
X = scaler.fit_transform(X)
```

```
print(X[0:5])
```



```
[[-0.90068117  1.01900435 -1.34022653 -1.3154443 ]
 [-1.14301691 -0.13197948 -1.34022653 -1.3154443 ]
 [-1.38535265  0.32841405 -1.39706395 -1.3154443 ]
 [-1.50652052  0.09821729 -1.28338910 -1.3154443 ]
 [-1.02184904  1.24920112 -1.34022653 -1.3154443 ]]
```

StandardScaler()作用

1. 加速梯度下降算法的收敛速度。
2. 防止计算过程中数值不稳定。
3. 提高模型性能，尤其是基于距离的算法（如KNN）和基于梯度的优化算法（如ANN）。
4. 统一尺度，
每个特征对模型的贡献更加均衡。

常见的数据处理方法

1. 归一化：将数据缩放到[0, 1]范围内。

```
from sklearn.preprocessing import MinMaxScaler
```

2. 二值化：将数据转换为二进制值。

```
from sklearn.preprocessing import Binarizer
```

3. 正则化：将数据缩放到单位范数。

```
from sklearn.preprocessing import Normalizer
```

4. 独热编码：将分类变量转换为独热编码。

```
from sklearn.preprocessing import OneHotEncoder
```

5. 标签编码：将分类标签转换为整数编码。

```
from sklearn.preprocessing import LabelEncoder
```

6. 缺失值处理：填补缺失值 (如 strategy='mean')。

```
from sklearn.impute import SimpleImputer
```

代码：前馈神经网络 <3>

<3> 数据集划分

```
from sklearn.model_selection import train_test_split
```

```
# 将数据集分为：训练集、测试集
```

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
```

```
# 测试集占20%(30个)，隐含训练集占80%(120个)
```

```
test_size = 0.2,
```

```
# 默认为None。设置固定值可以保证多次运行得到相同的分割结果。
```

```
random_state = 42,
```

```
# 默认为True，分割前对数据集进行洗牌
```

```
shuffle = True,
```

```
# 默认为None。如果不是None，数据将按照这个参数进行分层采样。
```

```
# 如根据标签y进行分层抽样，确保训练集和测试集中各类别样本的比例
```

```
# 与原始数据集中相同。
```

```
stratify = y )
```

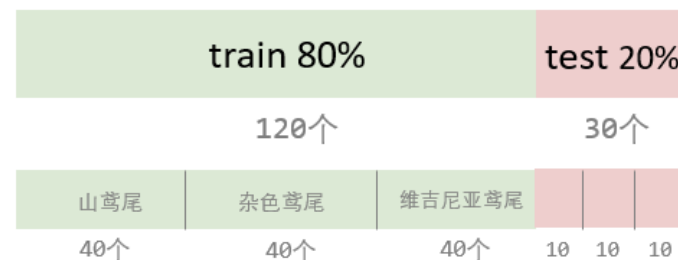
```
print(X_train[0:5]) # 训练集输入前5行数据
```

```
print(y_train[0:5]) # 训练集标签前5行数据
```

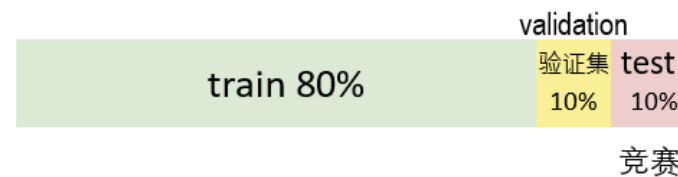
```
[[ -1.74885626 -0.36217625 -1.34022653 -1.3154443 ]
 [ -1.14301691 -1.28296331  0.42173371  0.65903847]
 [  1.15917263 -0.59237301  0.59224599  0.26414192]
 [ -1.14301691  0.09821729 -1.2833891  -1.44707648]
 [ -0.41600969 -1.28296331  0.13754657  0.13250973]]
[0 2 1 0 1]
```



留出法 (Hold-out)



模型容易偏向于训练集中出现次数多的样本

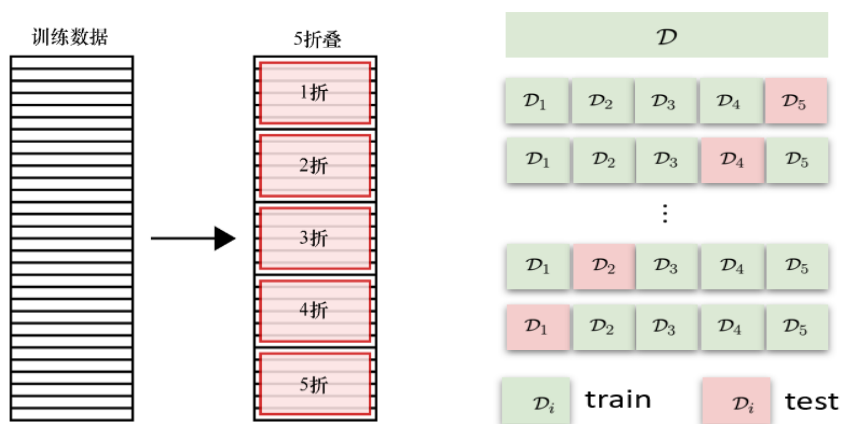


不足：只用80%的数据进行训练，而不是用全部训练。

K-fold 交叉验证法

k-fold cross-validation

- 将数据集**随机**分成 K 个大小相近的**子集**(fold)。
- 每次选择其中**1个子集**作为**测试集**，其余 $K - 1$ 个**子集**作为**训练集**。
- 重复上述过程 K 次，每个子集都会作为一次测试集。
- 将 K 次评估结果平均，得到最终模型性能的估计。



优点

- ✓ 充分利用有限的数据，每个样本都会被用作训练和测试。
- ✓ 模型性能估计稳定。避免单一训练/测试集带来的偏差。

```
import numpy as np
```

03-KFold.ipynb

```
# Create a sample dataset
```

```
X = np.array([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10],  
             [11, 12], [13, 14], [15, 16], [17, 18], [19, 20]])
```

```
y = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

```
from sklearn.model_selection import KFold
```

```
# Create a 10-fold cross validation
```

```
kf = KFold(n_splits=10,          # 10-fold  
           shuffle=True,  
           random_state=42)
```

```
# Split the dataset
```

```
for train_index, test_index in kf.split(X):  
    print("TRAIN:", train_index, "TEST:", test_index)  
    # X and y  
    X_train, X_test = X[train_index], X[test_index]  
    y_train, y_test = y[train_index], y[test_index]
```

输出结果:

```
TRAIN: [0 1 2 3 4 5 6 7 9] TEST: [8]  
TRAIN: [0 2 3 4 5 6 7 8 9] TEST: [1]  
TRAIN: [0 1 2 3 4 6 7 8 9] TEST: [5]  
TRAIN: [1 2 3 4 5 6 7 8 9] TEST: [0]  
TRAIN: [0 1 2 3 4 5 6 8 9] TEST: [7]  
TRAIN: [0 1 3 4 5 6 7 8 9] TEST: [2]  
TRAIN: [0 1 2 3 4 5 6 7 8] TEST: [9]  
TRAIN: [0 1 2 3 5 6 7 8 9] TEST: [4]  
TRAIN: [0 1 2 4 5 6 7 8 9] TEST: [3]  
TRAIN: [0 1 2 3 4 5 7 8 9] TEST: [6]
```

03-留一法.ipynb

- 将数据集分成 n (样本数) 个子集。
- 每次用 $n - 1$ 个样本为训练集，剩下1个样本为测试集。
- 重复这个过程 n 次，每个样本都会被用作一次测试集。
- 计算 n 次验证的平均误差作为模型性能的估计。

优点

- ✓ 充分利用数据，每个样本都被用于训练和测试。
- ✓ 对小数据集特别有用。
- ✓ 不需要多次随机划分数据集。

缺点

- 计算成本高，需要训练 n 次模型。
- 对于大数据集可能会非常耗时。

```
from sklearn.model_selection import LeaveOneOut
import numpy as np
```

```
# 假设我们有一个数据集X和对应的标签y
X = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
y = np.array([1, 2, 3, 4])
```

```
# 创建LeaveOneOut对象
loo = LeaveOneOut()
```

```
# 使用LeaveOneOut分割数据
for train_index, test_index in loo.split(X):
    print("TRAIN:", train_index, "TEST:", test_index)
    # X and y
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
```

```
TRAIN: [1 2 3] TEST: [0]
TRAIN: [0 2 3] TEST: [1]
TRAIN: [0 1 3] TEST: [2]
TRAIN: [0 1 2] TEST: [3]
```


通过从原始数据集中**有放回地**、随机、**重复**抽取样本，生成与原始数据集大小相同的新样本。

原数据集中部分样本重复在训练集中出现，
而部分样本从未在训练集中出现。

03-自助法.ipynb

优点

- ✓ 适用于小样本数据。
- ✓ 不需要对数据分布做严格假设。

缺点

- 计算成本高，需要大量重复采样。
- 对于时间序列或空间数据可能不适用。

```
import numpy as np
from sklearn.utils import resample

# 原始数据
data = np.random.normal(loc=30, scale=5, size=100)

# 自助法函数
def bootstrap(data, num_samples, statistic):
    stats = []
    for _ in range(num_samples):
        sample = resample(data)
        stats.append(statistic(sample))
    return stats

# 执行自助法
bootstrap_means = bootstrap(data, 1000, np.mean)
```

代码 前馈神经网络 <4>

```
# <1> 加载数据集  
# <2> 数据预处理  
# <3> 数据集划分  
# <4> 转换为PyTorch张量
```

```
import torch
```

```
print(X_train[0:5])  
print(y_train[0:5])
```

```
X_train = torch.FloatTensor(X_train)  
y_train = torch.LongTensor(y_train)  
X_test  = torch.FloatTensor(X_test)  
y_test  = torch.LongTensor(y_test)
```

```
print(X_train[0:5])  
print(y_train[0:5])
```

```
[[ -1.74885626 -0.36217625 -1.34022653 -1.3154443 ]  
 [ -1.14301691 -1.28296331  0.42173371  0.65903847]  
 [  1.15917263 -0.59237301  0.59224599  0.26414192]  
 [ -1.14301691  0.09821729 -1.2833891  -1.44707648]  
 [ -0.41600969 -1.28296331  0.13754657  0.13250973]]  
[0 2 1 0 1]
```

```
tensor([[ -1.7489, -0.3622, -1.3402, -1.3154],  
        [ -1.1430, -1.2830,  0.4217,  0.6590],  
        [  1.1592, -0.5924,  0.5922,  0.2641],  
        [ -1.1430,  0.0982, -1.2834, -1.4471],  
        [ -0.4160, -1.2830,  0.1375,  0.1325]])  
tensor([0, 2, 1, 0, 1])
```

PyTorch

常见的深度学习框架

为DL而开发的开源工具平台，包含常用DL模型开发包、训练模型等资源。

- **PyTorch**: 2016年, Facebook (学术) Python/C++

```
> pip install torch
```

- 核心代码开源
- 简洁、灵活、易用
- [文档全面, 示例多](#)
- 具备可扩展性



- **TensorFlow**: 2015年, Google (工业) Python/C++/Java



- **PaddlePaddle**: (飞桨) 2016年, 百度, 支持 Python/C++



- **MindSpore**: (昇思) 2020年, 华为, 支持 Python



第三方库管理工具：pip

Python官方提供的在线第三方库下载、安装、卸载、查找等管理工具。

【注】安装时需要上网。不要在IDLE下运行pip，在Win的命令提示符里运行pip/pip3、Mac终端里运行pip3。

pip install <拟安装库名>

```
命令提示符
Microsoft Windows [版本 10.0.19045.3930]
(c) Microsoft Corporation。保留所有权利。

C:\Users\Sam2023>pip install pypinyin
Collecting pypinyin
  Downloading pypinyin-0.50.0-py2.py3-none-any.whl.metadata (12 kB)
  Downloading pypinyin-0.50.0-py2.py3-none-any.whl (1.4 MB)
----- 1.4/1.4 MB 91.3 kB/s eta 0:00:00
Installing collected packages: pypinyin
Successfully installed pypinyin-0.50.0

C:\Users\Sam2023>_
```

pip uninstall <拟卸载库名>

```
命令提示符
Microsoft Windows [版本 10.0.19045.4046]
(c) Microsoft Corporation。保留所有权利。

C:\Users\Sam2023>pip uninstall pypinyin
Found existing installation: pypinyin 0.50.0
Uninstalling pypinyin-0.50.0:
  Would remove:
    c:\users\sam2023\appdata\local\programs\python\python312\lib\site-packages\pypinyin-0.50.0.dist-info\*
    c:\users\sam2023\appdata\local\programs\python\python312\lib\site-packages\pypinyin\*
    c:\users\sam2023\appdata\local\programs\python\python312\scripts\pypinyin.exe
Proceed (Y/n)? y
Successfully uninstalled pypinyin-0.50.0

C:\Users\Sam2023>_
```

pip -h 列出pip常用子命令

```
C:\Users\Sam2023>pip -h
```

Usage:

pip <command> [options]

Commands:

install
download
uninstall
freeze
inspect
list
show
check
config
search
cache
index
wheel
hash
completion
debug
help

Install packages.

Download packages. 下载第三方库的安装包

Uninstall packages.

Output installed packages in requirements format.

Inspect the python environment.

List installed packages. 列出当前已安装的库名

Show information about installed packages.

Verify installed packages have compatible dependencies.

Manage local and global configuration.

Search PyPI for packages.

Inspect and manage pip's wheel cache.

Inspect information available from package indexes.

Build wheels from your requirements.

Compute hashes of package archives.

A helper command used for command completion.

Show information useful for debugging.

Show help for commands.

国内清华镜像 pip

- 设为默认:

升级 pip 到最新的版本 ($\geq 10.0.0$)

```
>python -m pip install --upgrade pip
```

或

```
>python -m pip install -i https://pypi.tuna.tsinghua.edu.cn/simple --upgrade pip
```

配置到国内清华镜像

```
C:\Users\aa>pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple  
Writing to C:\Users\aa\AppData\Roaming\pip\pip.ini
```

安装 *torch* 库

```
>pip install torch
```

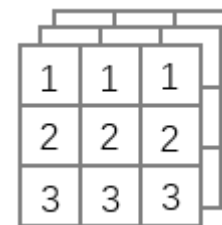
- 临时使用:

```
>pip install -i https://pypi.tuna.tsinghua.edu.cn/simple torch
```

PyTorch 核心模块

- **torch**: 常用的常量、函数 (加法操作 `torch.add`、随机函数`torch.randn`、激活函数`torch.relu...`)
- **torch.Tensor**: 定义了不同数值类型 (整数、单精度、双精度浮点数) 的张量。
- **torch.nn**: 构建**神经网络**的**核心**模块。已将 `torch.autograd` (自动求导) 重新封装。
 - **torch.nn.functional**: 神经网络常用函数。如, 激活函数、卷积函数、池化函数...
 - **torch.nn.init**: 模型初始化常见策略。如, 均匀分布初始化、正态分布初始化...
- **torch.optim**: 优化器 (`torch.optim.SGD`, `torch.optim.Adam...`)、学习率调整算法。
- 可视化: Tensorboard、Visdom、**wandb**。
- **torch.utils**: 辅助训练、测试、优化。
 - **torch.utils.data**: 数据处理, 如, 数据载入**DataLoader**、数据变换Transforms、数据分割...

数据表达



.....

数 学

$x = 1$

$\mathbf{x} = [1, 2, 3]$

$$X = \begin{bmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{m1} & \cdots & x_{mn} \end{bmatrix}$$

NumPy

标量
Vector

向量
Vector

矩阵
Matrix

三维数组

N 维数组

PyTorch

零维张量

一维张量

二维张量
Tensor

三维张量

N 维张量

torch.Tensor 张量

张量：PyTorch基本运算单位，类似于Numpy中的ndarray，GPU加速、自动微分。

```
import numpy as np

# 创建一个2x3的张量
A = np.array([[1, 2, 3], [4, 5, 6]])
print(f"A={A}")

# 张量加法
B = A + A

print(f"B={B}")
```

```
A=[[1 2 3]
    [4 5 6]]
B=[[ 2  4  6]
    [ 8 10 12]]
```

```
import torch

# 创建一个2x3的张量
A = torch.tensor([[1, 2, 3], [4, 5, 6]])
print(f"A={A}")

# 张量加法
B = A + A

print(f"B={B}")
```

```
A=tensor([[1, 2, 3],
          [4, 5, 6]])
B=tensor([[ 2,  4,  6],
          [ 8, 10, 12]])
```

PyTorch 常用nn模块

- **线性层**
 - **nn.Linear**: 对输入数据进行线性变换
- **卷积层** **nn.Conv1d**, **nn.Conv2d**, **nn.Conv3d**: 对输入信号1D、2D或3D卷积
- **池化层**
 - **nn.MaxPool1d**, **nn.MaxPool2d**, **nn.MaxPool3d**: 对输入信号应用最大池化
 - **nn.AvgPool1d**, **nn.AvgPool2d**, **nn.AvgPool3d** : 对输入信号应用平均池化
 - **nn.AdaptiveMaxPool1d**, **nn.AdaptiveMaxPool2d**, **nn.AdaptiveMaxPool3d**: 自适应最大池化
 - **nn.AdaptiveAvgPool1d**, **nn.AdaptiveAvgPool2d**, **nn.AdaptiveAvgPool3d**: 自适应平均池化
- **循环层**
 - **nn.RNN**: 多层 Elman RNN
 - **nn.LSTM**: 多层长短期记忆 (LSTM) RNN
 - **nn.GRU**: 多层门控循环单元(GRU) RNN
- **归一化层**
 - **nn.BatchNorm1d**, **nn.BatchNorm2d**, **nn.BatchNorm3d**: 批量归一化
 - **nn.LayerNorm**: 层归一化
 - **nn.InstanceNorm1d**, **nn.InstanceNorm2d**, **nn.InstanceNorm3d**: 实例归一化
- **Dropout层** **nn.Dropout**, **nn.Dropout2d**, **nn.Dropout3d**
- **容器模块**
 - **nn.Sequential**: 顺序的模块容器
 - **nn.ModuleList**: 在列表中保存子模块
 - **nn.ModuleDict**: 在字典中保存子模块
- **激活函数**
 - **nn.Sigmoid**: sigmoid函数
 - **nn.Softmax**: softmax函数
 - **nn.ReLU**: 修正线性单元函数
 - **nn.LeakyReLU**: 带泄漏的ReLU
 - **nn.Tanh**: 双曲正切函数
 - **nn.ELU**: 指数线性单元函数
- **损失函数**
 - **nn.MSELoss**: 均方误差
 - **nn.CrossEntropyLoss**: 交叉熵
 - **nn.BCELoss**: 二元交叉熵

PyTorch 常用的层

- **Linear Layers**
- Convolution Layers
- Pooling layers
- Padding Layers
- Dropout Layers
- Vision Layers
- Recurrent Layers
- Transformer Layers

03-线性层.ipynb

```
import torch

x = torch.Tensor([1, 2])
model = torch.nn.Linear(2, 1)
y = model(x)      # (输入数, 输出数)
print(y)
```

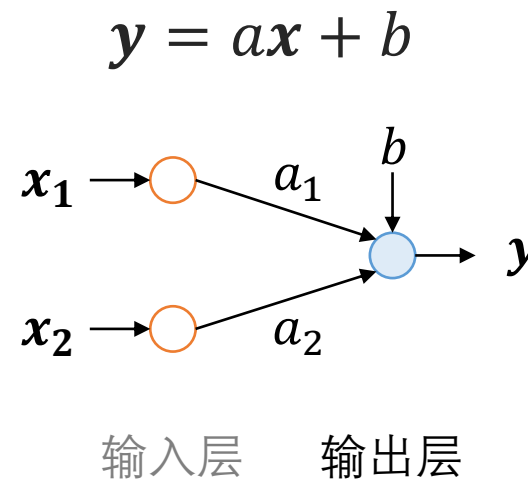
```
tensor([-0.9521], grad_fn=<ViewBackward0>)
```

模型参数

```
for param in model.parameters():
    print(param)
```

```
Parameter containing: tensor([[0.0417, -0.5226]], requires_grad=True)
Parameter containing: tensor([0.0514], requires_grad=True)
```

$$y = [1, 2] * [0.0417, -0.5226]^T + 0.0514 = -0.9521$$



多样本 批处理

```
import torch
```

03-线性层.ipynb

```
# 一次输入3个样本
```

```
x = torch.Tensor([[1, 2], # 样本1  
                  [3, 4], # 样本2  
                  [5, 6]]) # 样本3
```

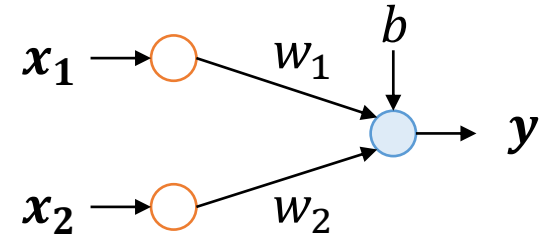
```
model = torch.nn.Linear(in_features=2,  
                        out_features=1,  
                        bias=True)
```

```
y = model(x)
```

```
print(y)
```

```
tensor([[ -0.2523], # 输出y1  
        [ -0.4731], # 输出y2  
        [ -0.6939]], # 输出y3  

```



输入层 输出层

$$y = xw^T + b$$

$$\begin{bmatrix} -0.2523 \\ -0.4731 \\ -0.6939 \end{bmatrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} + \begin{bmatrix} b \\ b \\ b \end{bmatrix}$$

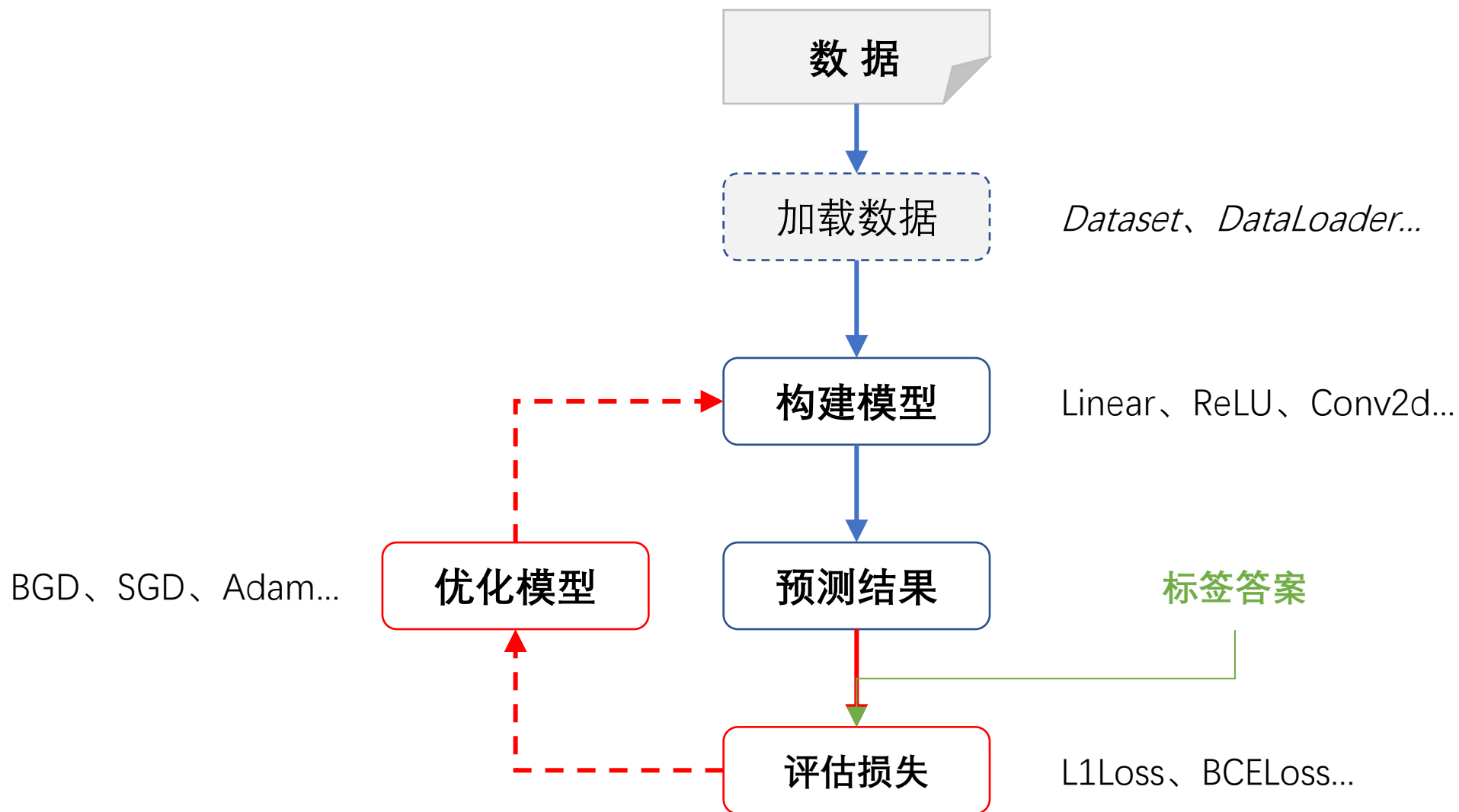
一次处理多个样本（批处理）

- 利用矩阵运算的优势，提高计算效率；
- 训练时，可以更稳定地更新模型参数；
- 模型训练时看到更多的数据，提高泛化能力。

03-前馈网络-PyTorch.ipynb

② 定义模型

监督学习 流程



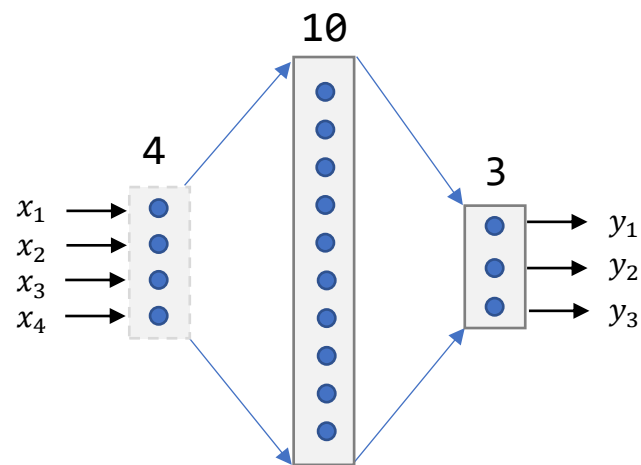
代码：前馈神经网络 <5-1>

<5> 定义模型

`import torch.nn as nn`

<5-1> 网络模型

```
model = nn.Sequential(  
    nn.Linear(4, 10),    # 隐藏层，输入维度为4，输出维度为10  
    nn.ReLU(),           # 激活函数（下页）  
    nn.Linear(10, 3) )  # 输出层，输入维度为10，输出维度为3
```

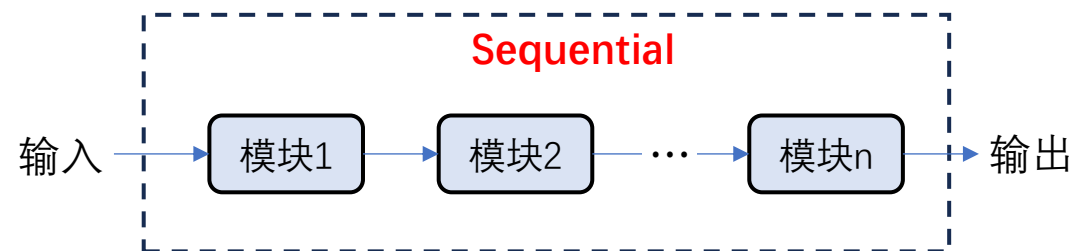


输入层 隐藏层 输出层

序列容器，用于搭建神经网络模型。

按照传入顺序添加到容器中，

模型前向传播时调用forward()方法。



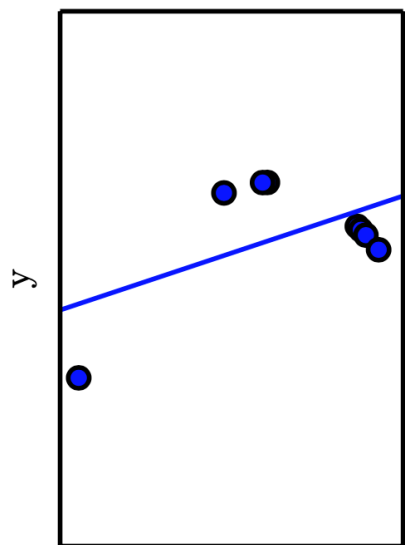
利用索引查询模型中的第1层、第2层

`print(model[0])``print(model[1])`

```
Linear(in_features=4, out_features=10, bias=True)  
ReLU()
```

模型的选择

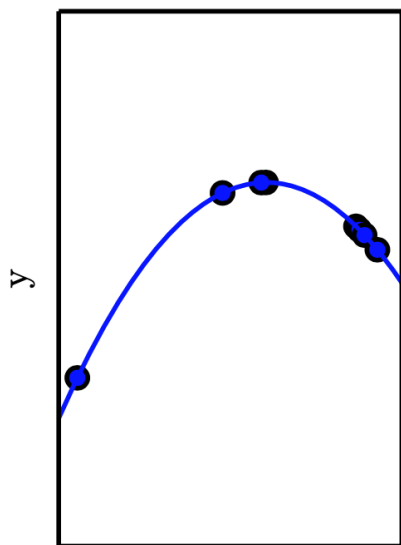
欠拟合



x_0

$$y = a_0 + a_1x$$

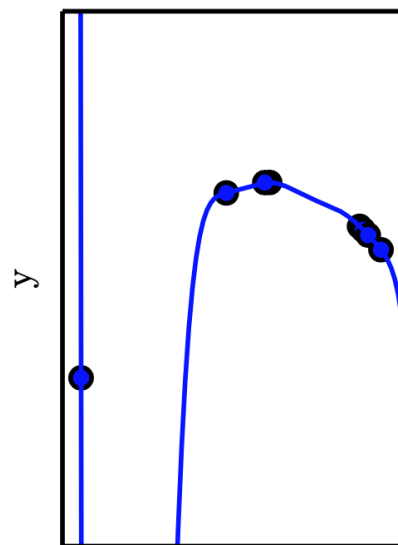
刚好拟合



x_0

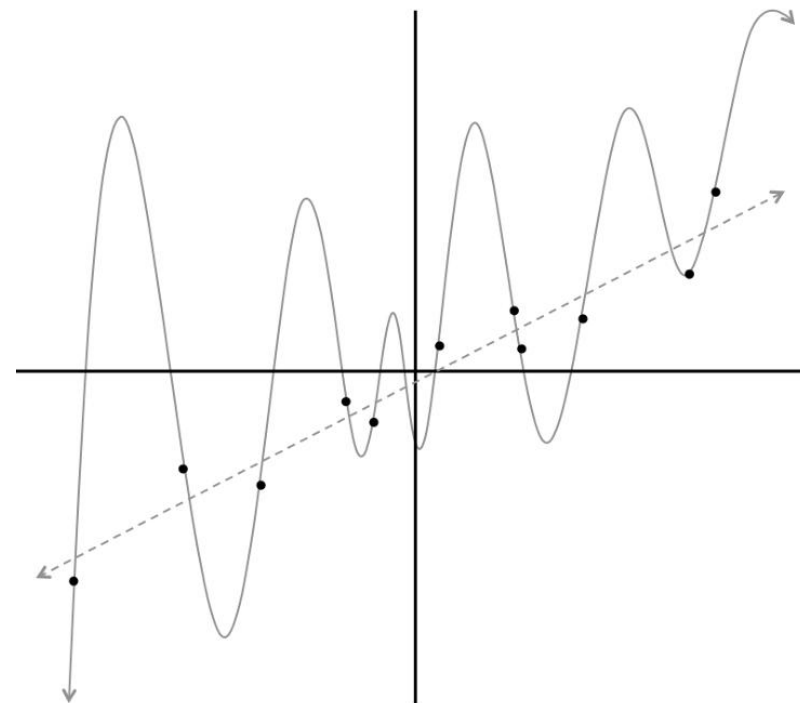
$$y = a_0 + a_1x + a_2x^2$$

过拟合



x_0

$$y = \sum_{i=0}^9 a_i x^i$$

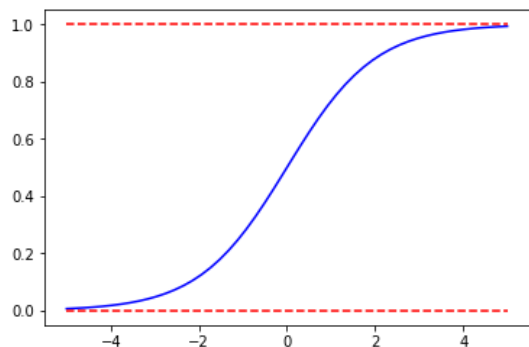


过拟合曲线对新数据拟合效果差

常用激活函数

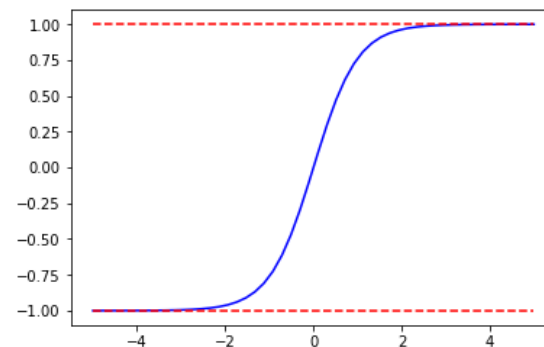
作用：引入**非线性**，让网络能逼近任意复杂的函数，解决线性模型无法解决的问题。
将输出值**归一化**，有助于训练过程的**稳定**。

- sigmoid() 用于二分类问题



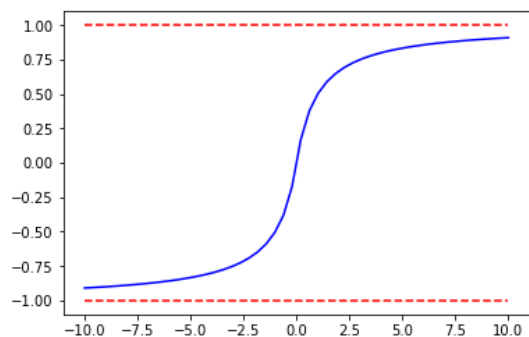
$$f(x) = \frac{1}{1 + e^{-x}}$$

- tanh()



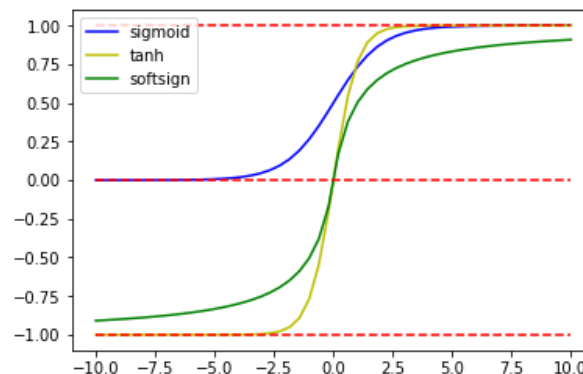
$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- softsign()



$$f(x) = \frac{x}{1 + |x|}$$

对比



代码：前馈神经网络 <5-2>

```
# <5> 定义模型
import torch.nn as nn

# <5-1> 网络模型
model = nn.Sequential(
    nn.Linear(4, 10),    # 输入层，输入维度为4，输出维度为10
    nn.ReLU(),           # 激活函数
    nn.Linear(10, 3)    ) # 输出层，输入维度为10，输出维度为3

# <5-2> 损失函数
Loss = nn.CrossEntropyLoss() # 交叉熵
```

Loss Function

损失函数 / 误差函数：

计算标签值(答案)和预测值差异的函数。

常用的损失函数

回归损失函数：

- nn.L1Loss: 平均绝对误差损失
- nn.MSELoss: 均方误差损失 **M**ean **S**quared **E**rror
- nn.SmoothL1Loss: 平滑L1损失，结合MSE和L1

分类损失函数：

- nn.CrossEntropyLoss: 交叉熵损失，多分类问题
- nn.BCELoss: 二元交叉熵损失，二分类问题
- nn.BCEWithLogitsLoss: 结合了Sigmoid和BCELoss

其他损失函数：

- nn.HingeEmbeddingLoss: 用于半监督学习
- nn.NLLLoss: 负对数似然损失
- nn.KLDivLoss: KL散度损失

L1Loss()、BCELoss()

平均绝对误差损失

二元交叉熵损失

```
import torch
```

```
# L1Loss() 损失函数：预测值和真实值绝对误差的平均数。
```

```
target = torch.tensor([[1., 1.], [1., 1.]])
```

```
output = torch.tensor([[0., 1.], [2., 3.]])
```

```
criterion = torch.nn.L1Loss()
```

```
loss = criterion(output, target)
```

```
print(loss)
```

```
# 返回: tensor(1.)
```

03-损失函数.ipynb

```
# BCELoss() 二元交叉熵损失
```

```
target = torch.tensor([[0.0, 1.0], [1.0, 0.0]])
```

```
output = torch.tensor([[0.2, 0.9], [0.6, 0.9]])
```

```
criterion = torch.nn.BCELoss()
```

```
loss = criterion(output, target)
```

```
print(loss)
```

```
# 返回: tensor(0.7855)
```

$$L1 = \frac{1}{N} \sum_{i=1}^N |y_i - \hat{y}_i|$$

答案 预测

$$BCE = \sum_{i=1}^N (-y_i \log \hat{y}_i - (1 - y_i) \log(1 - \hat{y}_i))$$

代码：前馈神经网络 <5-3>

```
# <5> 定义模型
import torch.nn as nn

# <5-1> 网络模型
model = nn.Sequential(
    nn.Linear(4, 10),    # 输入层，输入维度为4，输出维度为10
    nn.ReLU(),           # 激活函数
    nn.Linear(10, 3) )  # 输出层，输入维度为10，输出维度为3

# <5-2> 损失函数
Loss = nn.CrossEntropyLoss() # 交叉熵损失函数

# <5-3> 优化器
optimizer = torch.optim.Adam(
    model.parameters(), # 待优化参数
    lr=0.01)            # 学习率  $\eta$ 
```

根据梯度更新模型参数，最小化损失。

```
# SGD优化器
optimizer = torch.optim.SGD(
    model.parameters(), # 待优化参数
    lr=0.01 )          # 学习率  $\eta$ 
```

常用的优化器

- torch.optim.SGD：随机梯度下降

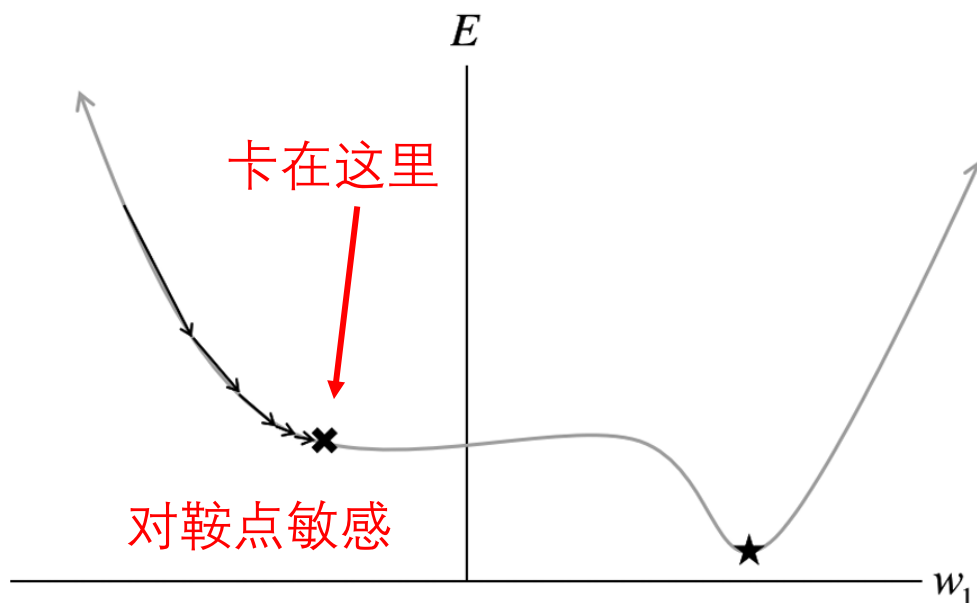
自适应学习率优化器

- torch.optim.Adam
- torch.optim.AdamW
- torch.optim.Adamax
- torch.optim.ASGD (平均随机梯度下降)
- torch.optim.Adagrad
- torch.optim.Adadelta
- torch.optim.RMSprop

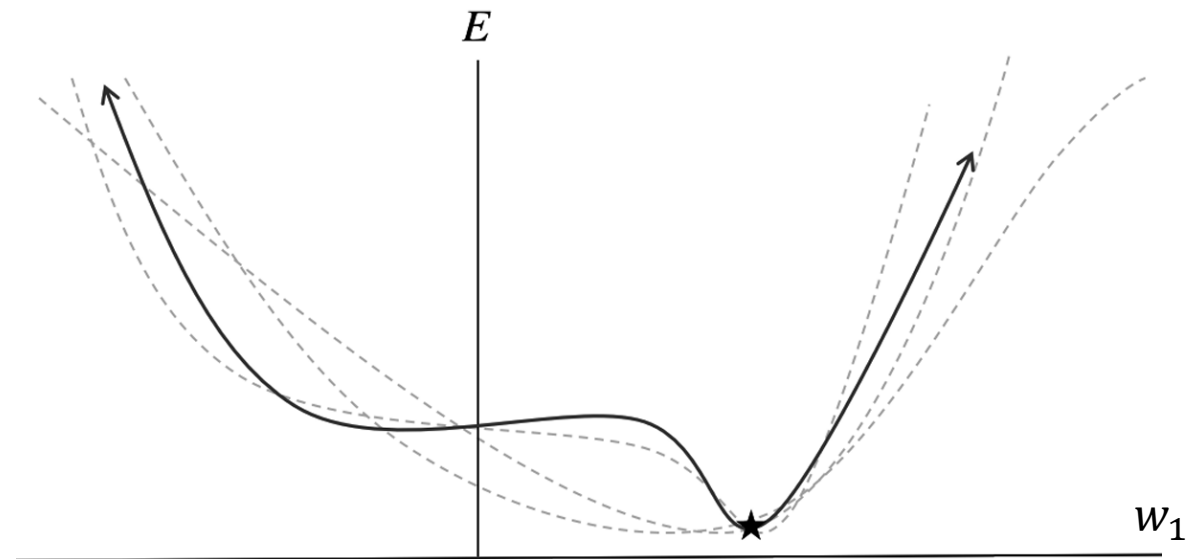
批量梯度下降、随机梯度下降

每次迭代使用全部训练数据计算梯度。

每次迭代随机选取一个样本
(或一个小批量样本) 计算梯度。



假设：仅有一个权重 w_1



每次迭代，误差曲面只针对一个样例进行评估。
误差曲面是动态的。

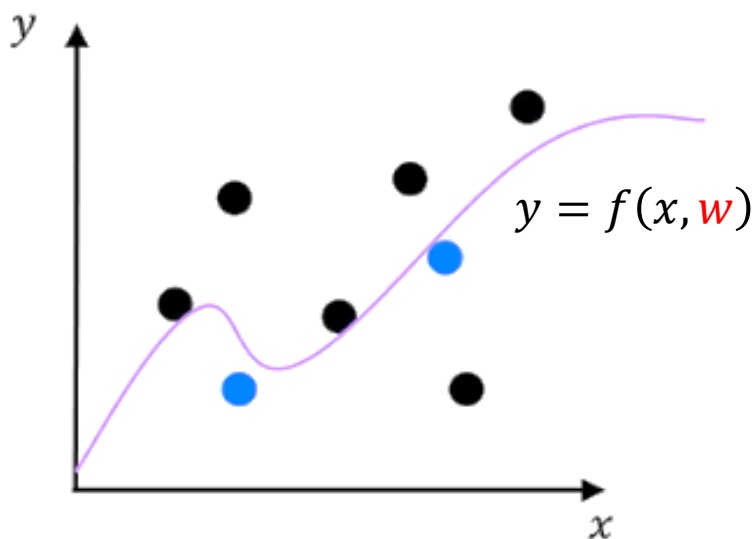
小批量梯度下降

随机梯度下降法

Stochastic Gradient Descent

基本思想：通过逐个样本或小批量样本来更新模型参数，而不用整个数据集。
提高了计算效率（大规模数据集）。

```
optimizer = torch.optim.SGD(model.parameters(), # 待优化参数
                              lr=0.01 )         # 学习率
```



【注】在学习过程中，权重 w 是函数中的变量，而不是输入 x 。

$$\nabla_w \mathcal{L}(x_1)$$

$$\nabla_w \mathcal{L}(x_2)$$

$$\nabla_w \mathcal{L}(x_3)$$

$$\nabla_w \mathcal{L}(x_4)$$

$$\nabla_w \mathcal{L}(x_5)$$

$$\nabla_w \mathcal{L}(x_6)$$

$$\nabla_w \mathcal{L}(x_7)$$

$$\nabla_w \mathcal{L}(x_8)$$

$$\mathcal{L}(f(x_i, w), y_i)$$

损失函数

$$\nabla_w \mathcal{L}(f(x_i, w), y_i)$$

损失函数梯度

$$g = \frac{1}{n} \nabla_w \sum_{i=1}^n \mathcal{L}(f(x_i, w), y_i)$$

平均梯度

$$w \leftarrow w - \eta \cdot g$$

更新权值

```
optimizer.step()
```

03-前馈网络-PyTorch.ipynb

③ 训练模型

代码：前馈神经网络 <6>

<6> 自定义计算准确率函数

```
def test(model,          # 模型
        X_test,         # 测试集输入, 共有30个test样本
        y_test,         # 测试集标签
        quiet=False,    # 是否打印输出
        ):

    # <6-1> 进入评估模式
    model.eval()

    # <6-2> 关闭梯度下计算输出
    with torch.no_grad():
        # 由测试集 X_test, 计算预测值 predicted
        outputs = model(X_test)

    # <6-3> 计算准确率
    # 按行(=1)取最大值(预测结果), 返回最大值、最大值的索引(鸢尾花种类)
    _, predicted = torch.max(outputs.data, 1)

    # 预测值predicted 与真实值y_test 比较后, 再求和 / 测试集总数
    accuracy = (predicted == y_test).sum().item() / y_test.size(0)

    # 返回: 准确率、预测值
    return accuracy, predicted
```

- 评估模式 (Evaluation Mode)
不进行反向传播、也不更新权重。
- Dropout层不工作。
- BatchNorm层: 直接使用训练阶段已经学出的值。
- 训练模式 (Training Mode)

```
输出值(1-5):
tensor([[ 4.5026, -0.9787, -6.6397],
        [-3.3380,  1.2933,  2.1879],
        [-0.8905,  3.0658, -1.7915],
        [-1.3904,  3.5429, -1.4668],
        [ 4.9271, -1.5089, -6.8071]])
```

```
print('输出值(1-5): ', outputs[:5], sep='\n')
```

```
最大值(1-5):
tensor([4.5026, 2.1879, 3.0658, 3.5429, 4.9271])
最大值序号(1-5): tensor([0, 2, 1, 1, 0])
```

```
print('最大值(1-5): ', _[:5], sep='\n')
print('最大值序号(1-5): ', predicted[:5])
```

```
print('正确总数: ', (predicted == y_test).sum())
print('正确总数: ', (predicted == y_test).sum().item())
print('test总数: ', y_test.size(0))
```

```
正确总数: tensor(29)
正确总数: 29
test总数: 30
```


代码：前馈神经网络 <7>

<7> 训练模型

`from tqdm import tqdm` # 进度条库

初始化训练过程中的指标：训练精度、测试精度、损失，（用于绘图）

`train_acc_history, test_acc_history, loss_history = [], [], []``num_epochs = 100` # 迭代次数，重复训练100次`model.train()` # 训练模式（默认）`for epoch in tqdm(range(num_epochs)):` 100%|██████████| 100/100 [00:00<00:00, 1959.26it/s]

1) 向前传播计算结果

`outputs = model(X_train)`

2) 计算损失

`loss = Loss(outputs, y_train)`

3) 反向传播，计算梯度

`loss.backward()`

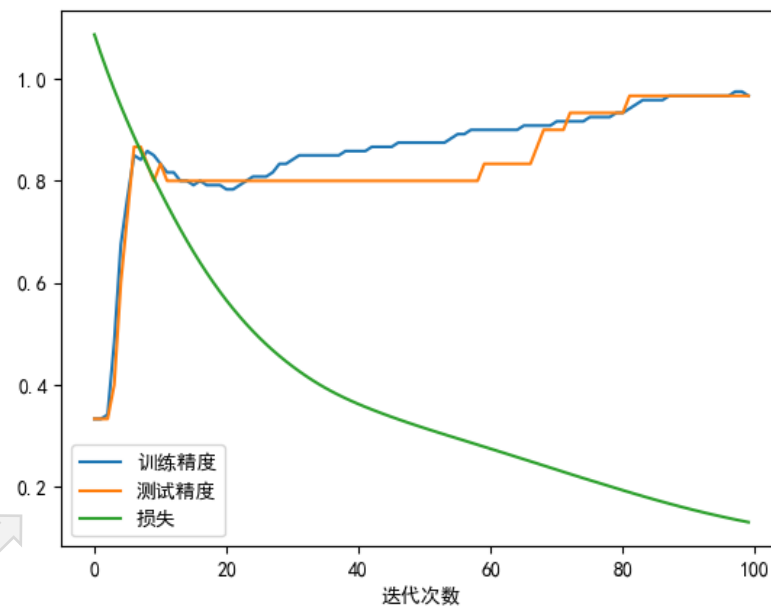
4) 更新权重

`optimizer.step()``optimizer.zero_grad()` # 梯度清零，准备下一次迭代

更新绘图指标

`train_acc_history.append(test(model, X_train, y_train, quiet=True)[0])``test_acc_history.append(test(model, X_test, y_test, quiet=True)[0])``loss_history.append(loss.item())`

打印训练信息

`if (epoch + 1) % 10 == 0:``print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')`

Epoch [10/100], Loss: 0.8420

Epoch [20/100], Loss: 0.5854

Epoch [30/100], Loss: 0.4359

Epoch [40/100], Loss: 0.3251

Epoch [50/100], Loss: 0.2360

Epoch [60/100], Loss: 0.1699

Epoch [70/100], Loss: 0.1264

Epoch [80/100], Loss: 0.0997

Epoch [90/100], Loss: 0.0832

Epoch [100/100], Loss: 0.0727

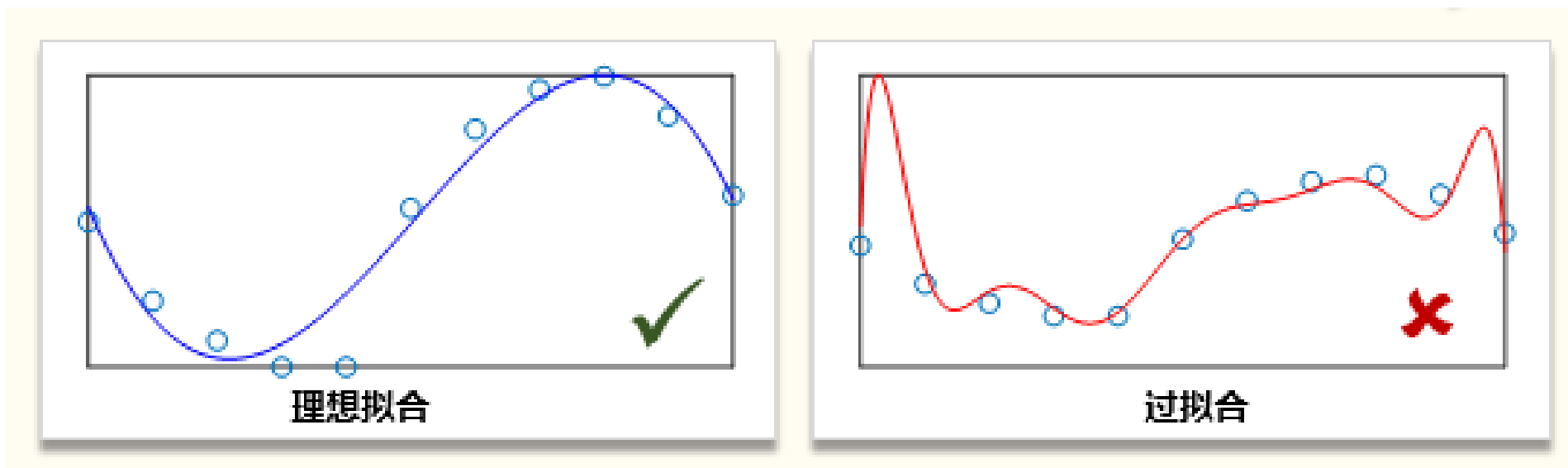
迭代学习

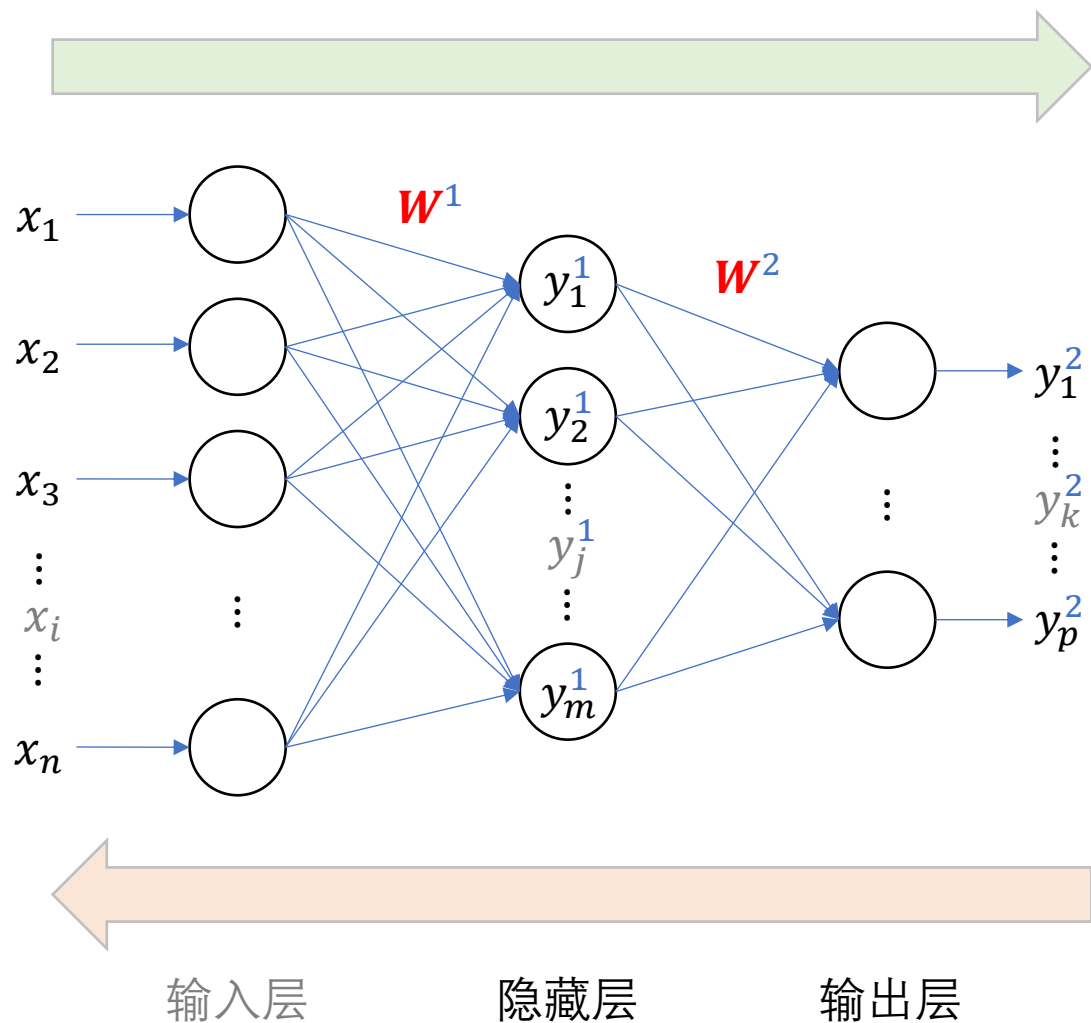
为了提高模型的精度，可让模型对同一训练数据进行**反复多次(epoch 轮次)**学习。

增加 epoch 数并不意味着模型的精度也会持续提升，

模型的准确率到一定程度就会停止提升，

不断重复的学习可能导致损失函数被最小化而引起**过拟合**问题。





学习过程

- 1) **正向传播**: 把样本的特征从输入层输入, 信号经过各个隐藏层逐层处理后, 从输出层传出。
- 2) **误差反向传播**: 把误差信号从最后一层逐层反传, 修正各个层神经元的权值。

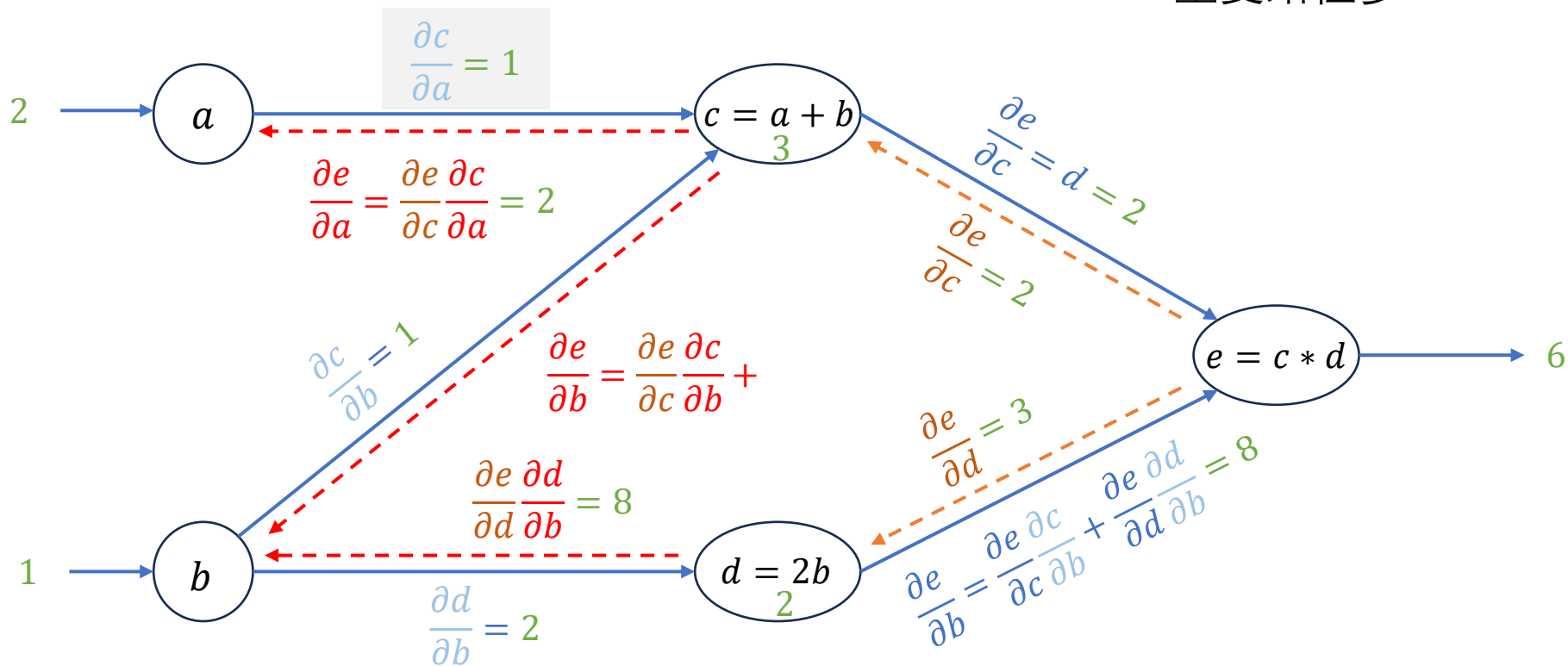
导数常用运算

- 相加函数的导数:
$$\frac{\partial(f(x) + g(x))}{\partial x} = \frac{\partial f(x)}{\partial x} + \frac{\partial g(x)}{\partial x}$$
- 相乘函数的导数:
$$\frac{\partial(f(x) \cdot g(x))}{\partial x} = \frac{\partial f(x)}{\partial x} g(x) + f(x) \frac{\partial g(x)}{\partial x}$$
- 嵌套函数的导数:
(链式求导法则)
$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \cdot \frac{\partial g(x)}{\partial x}$$

例： $e = c * d = (a + b) * 2b$

正向求导： $\frac{\partial}{\partial x}$

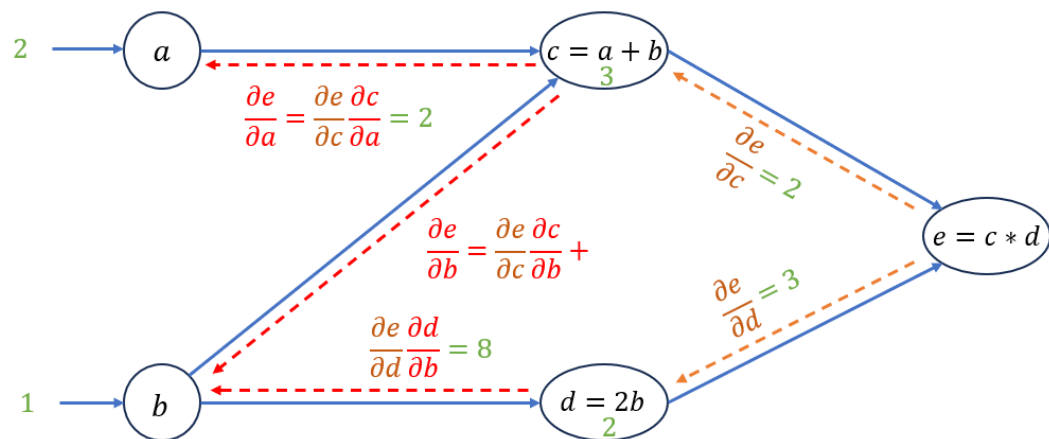
从头到尾一遍才得到一个输入 x 的导数！
重复路径多



反向求导： $\frac{\partial Z}{\partial}$

从尾到头一遍得到所有输入变量的导数！
当输入数大于输出数有优势

代码: $e = (a + b) * 2b$



反向求导: $\frac{\partial z}{\partial}$

从尾到头一遍得到**所有输入变量**的导数!

```
import torch
```

03-BP反向求导.ipynb

```
# 定义输入变量, requires_grad=True: 计算梯度
```

```
a = torch.tensor(2.0, requires_grad=True)
```

```
b = torch.tensor(1.0, requires_grad=True)
```

```
# 定义函数
```

```
e = (a + b) * 2 * b
```

```
# 计算e对所有输入的偏导数
```

```
e.backward()
```

```
# 所有输入的偏导数
```

```
print(f"∂e/∂a= {a.grad}")
```

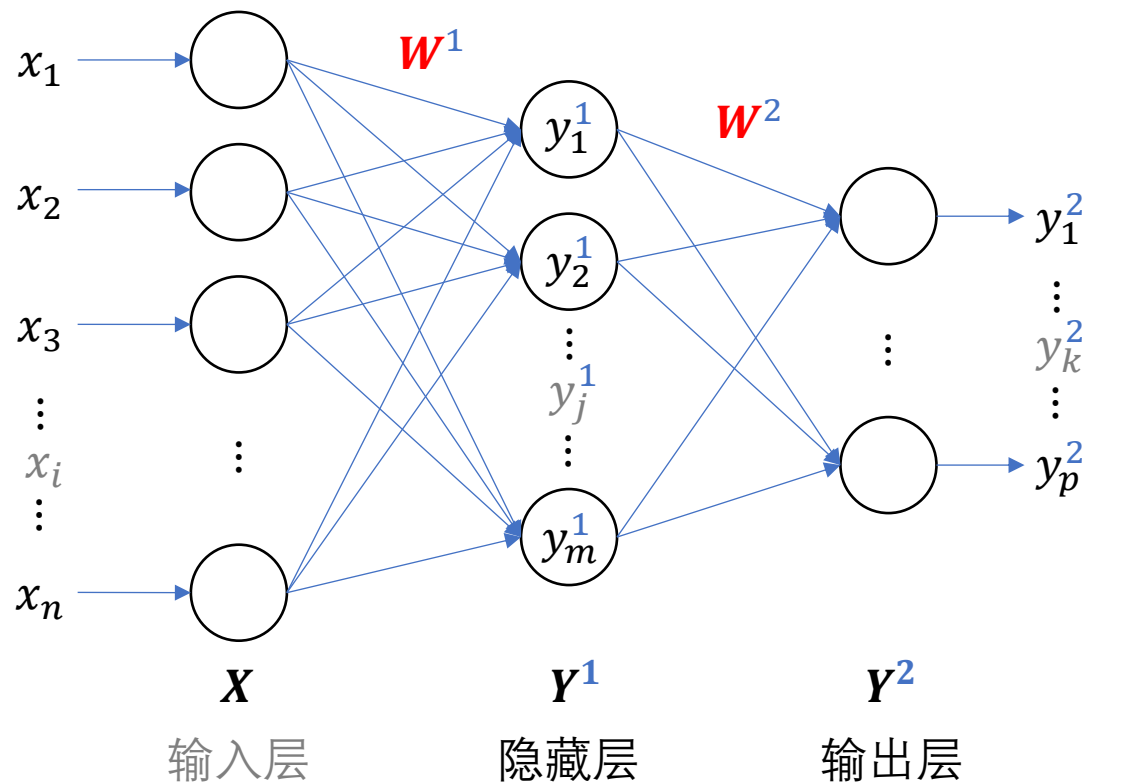
```
print(f"∂e/∂b= {b.grad}")
```

反向求导

$\partial e / \partial a = 2.0$

$\partial e / \partial b = 8.0$

BP 网络模型



$$net_j^1 = \sum_{i=1}^n w_{ij}^1 x_i \quad net_k^2 = \sum_{j=1}^m w_{jk}^2 y_j^1$$

$$y_j^1 = f(net_j^1) \quad y_k^2 = f(net_k^2)$$

输出层均方差: $E = \frac{1}{2} (\mathbf{Y}^* - \mathbf{Y}^2)^2 = \frac{1}{2} \sum_{k=1}^p (y_k^* - y_k^2)^2$

展开至隐藏层: $E = \frac{1}{2} \sum_{k=1}^p (y_k^* - f(net_k^2))^2$

$$= \frac{1}{2} \sum_{k=1}^p \left[y_k^* - f \left(\sum_{j=1}^m w_{jk}^2 y_j^1 \right) \right]^2$$

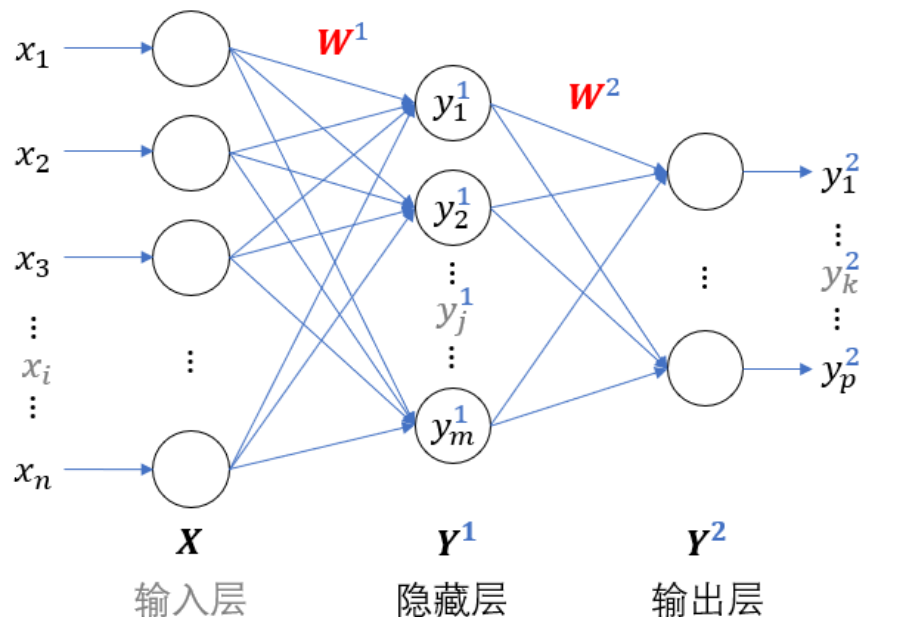
展开至输入层: $E = \frac{1}{2} \sum_{k=1}^p \left[y_k^* - f \left(\sum_{j=1}^m w_{jk}^2 f(net_j^1) \right) \right]^2$

$$= \frac{1}{2} \sum_{k=1}^p \left[y_k^* - f \left(\sum_{j=1}^m w_{jk}^2 f \left(\sum_{i=1}^n w_{ij}^1 x_i \right) \right) \right]^2$$

sigmoid 函数导数: $f'(x) = f(x)[1 - f(x)]$

BP算法推导：输出层

$$E = \frac{1}{2} \sum_{k=1}^p \left(y_k^* - f(\text{net}_k^2) \right)^2$$



$$\text{net}_k^2 = \sum_{j=1}^m w_{jk}^2 y_j^1$$

$$y_k^2 = f(\text{net}_k^2)$$

sigmoid函数导数: $f'(x) = f(x)[1 - f(x)]$

链式法则 $\Delta w_{jk}^2 = -\eta \frac{\partial E}{\partial w_{jk}^2} = -\eta \frac{\partial E}{\partial \text{net}_k^2} \frac{\partial \text{net}_k^2}{\partial w_{jk}^2} = \eta \delta_k^2 y_j^1$

$$\begin{aligned} \delta_k^2 &= -\frac{\partial E}{\partial y_k^2} \frac{\partial y_k^2}{\partial \text{net}_k^2} = -\frac{\partial E}{\partial y_k^2} f'(\text{net}_k^2) \\ &= (y_k^* - y_k^2) y_k^2 (1 - y_k^2) \end{aligned}$$

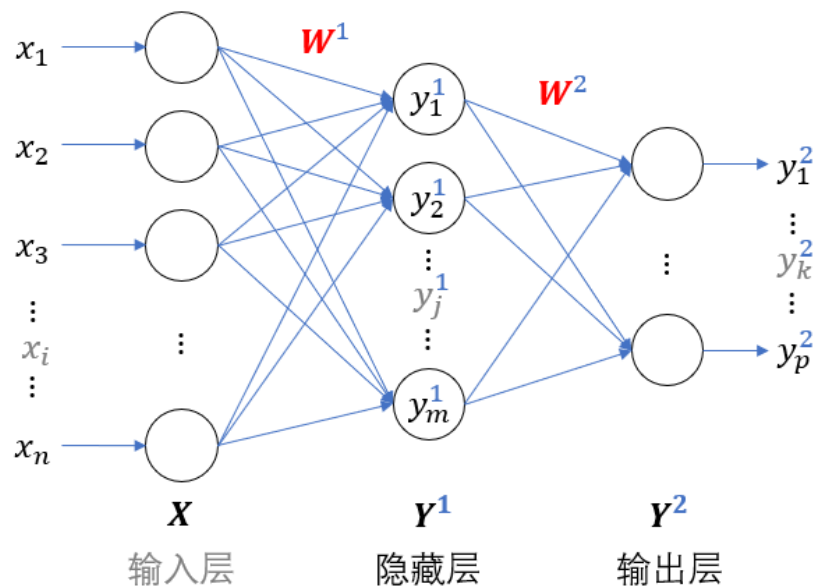
$$\Delta w_{jk}^2 = \eta (y_k^* - y_k^2) y_k^2 (1 - y_k^2) y_j^1$$

隐藏层h: $\Delta w_{jk}^{h+1} = \eta (y_k^* - y_k^{h+1}) y_k^{h+1} (1 - y_k^{h+1}) y_j^h$

矩阵形式: $\Delta W^{h+1} = \eta (Y^h)^T (Y^* - Y^{h+1}) \circ f'(Y^h W^{h+1})$

BP算法推导：隐藏层

$$E = \frac{1}{2} \sum_{k=1}^p \left[y_k^* - f \left(\sum_{j=1}^m w_{jk}^2 f(net_j^1) \right) \right]^2$$



$$net_j^1 = \sum_{i=1}^n w_{ij}^1 x_i$$

$$y_j^1 = f(net_j^1)$$

链式法则 $\Delta w_{ij}^1 = -\eta \frac{\partial E}{\partial w_{ij}^1} = -\eta \frac{\partial E}{\partial net_j^1} \frac{\partial net_j^1}{\partial w_{ij}^1} = \eta \delta_j^1 x_i$

$$\delta_j^1 = -\frac{\partial E}{\partial y_j^1} \frac{\partial y_j^1}{\partial net_j^1} = -\frac{\partial E}{\partial y_j^1} f'(net_j^1)$$

$$= \sum_{k=1}^q \left((y_k^* - y_k^2) f'(net_k^2) w_{jk}^2 \right) y_j^1 (1 - y_j^1)$$

$$\Delta w_{ij}^1 = \eta \left(\sum_{k=1}^p \delta_k^2 w_{jk}^2 \right) y_j^1 (1 - y_j^1) x_i$$

隐藏层h: $\Delta w_{ij}^h = \eta \left(\sum_{k=1}^p \delta_k^{h+1} w_{jk}^{h+1} \right) y_j^h (1 - y_j^h) x_i^{h-1}$

矩阵形式: $\Delta W^h = \eta (Y^{h-1})^T \delta^{h+1} (W^{h+1})^T \circ f'(Y^{h-1} W^h)$

BP算法 结论

h 层权值调整公式:

$$\Delta \mathbf{W}^h = \eta (\mathbf{Y}^{h-1})^T \delta^h$$

输出层: $\delta^{h+1} = (\mathbf{y}^* - \mathbf{Y}^{h+1}) \circ f'(\mathbf{Y}^h \mathbf{W}^{h+1})$ (已知)

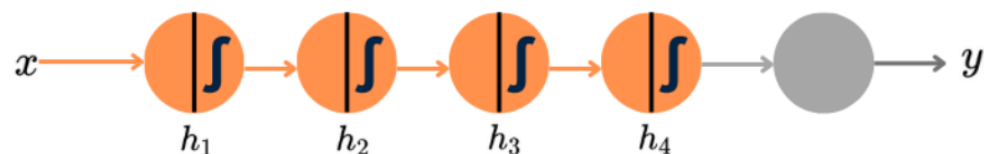
其余层: $\delta^h = \delta^{h+1} (\mathbf{W}^{h+1})^T \circ f'(\mathbf{Y}^{h-1} \mathbf{W}^h)$

δ^h : h 层学习信号

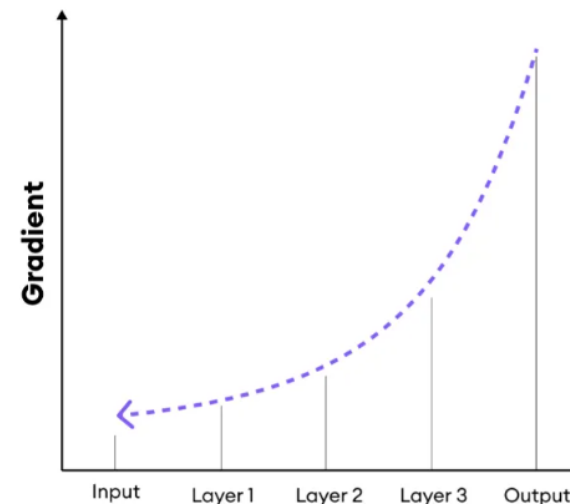
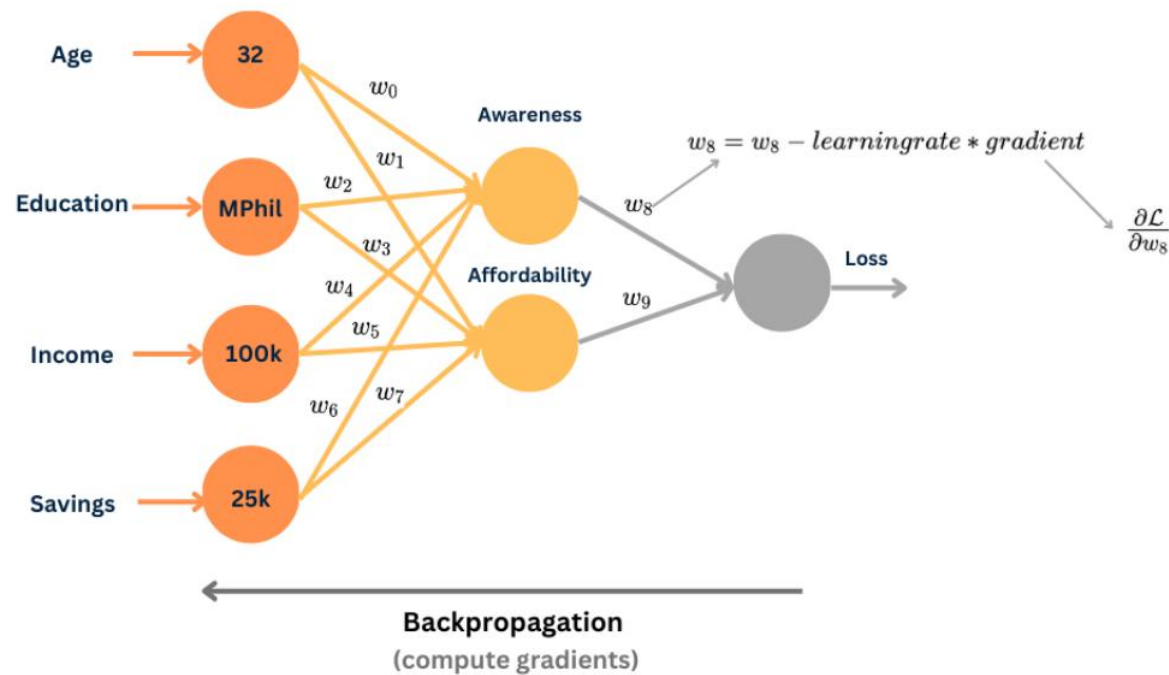
梯度消失

Vanishing Gradient

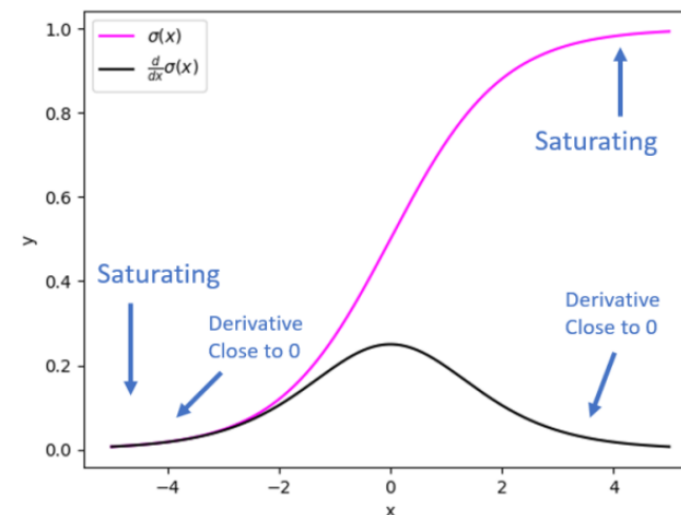
激活函数导数值越小， ΔW 的值也就越小



$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial h_4} \frac{\partial h_4}{\partial h_3} \frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial h_1} \frac{\partial h_1}{\partial w}$$



Layer



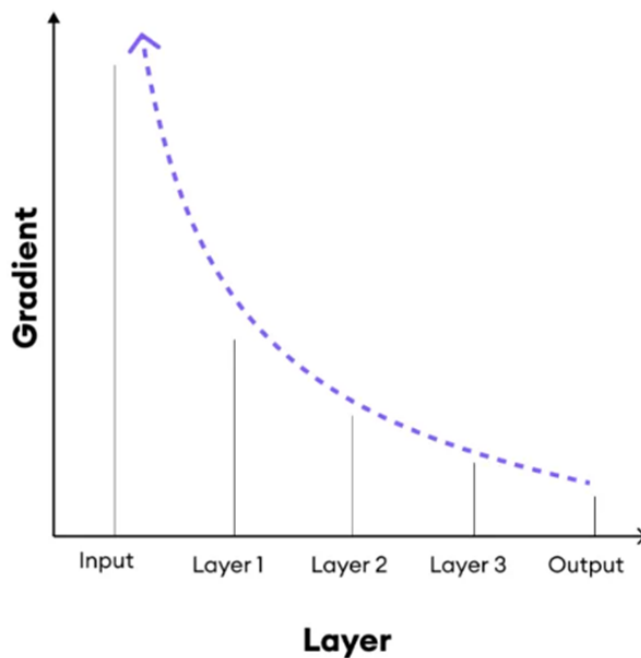
Sigmoid 函数及其导数

梯度爆炸

如果学习信号 δ 乘以一个大于 1 的数，那么 δ 就会变大。

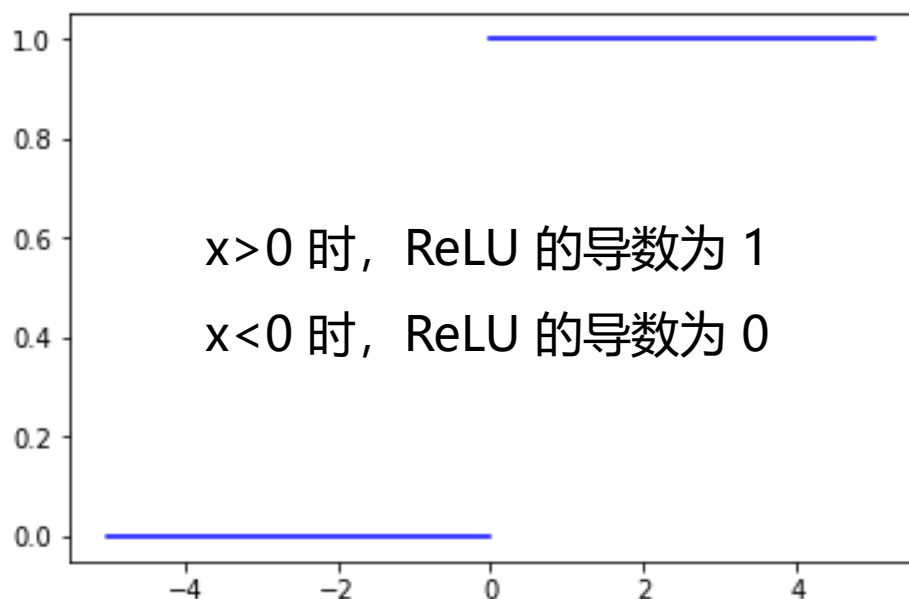
学习信号从输出层一层一层向前反向传播，每传播一层学习信号就会变大一点，经过多层传播后，学习信号就会接近于无穷大，使得权值 Δw 调整接近于无穷大。意味着该层的参数处于一种极不稳定的状态，网络不能正常工作。

既然激活函数的导数不能小于 1 也不能大于 1，
能不能使用线性函数 $y = x$ ，这个函数的导数是1。
既不会梯度消失，也不会梯度爆炸。
但是，线性函数不能处理非线性问题。



ReLU() 解决 梯度消失、梯度爆炸

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



ReLU 函数的导数

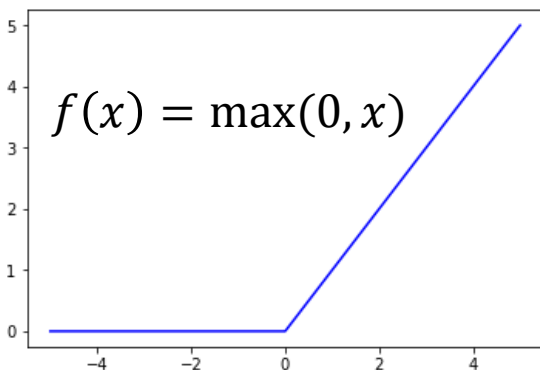
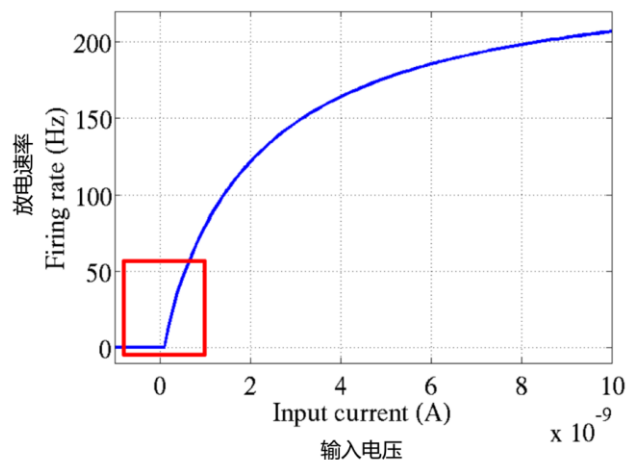
- ✓ 导数=1 不会使学习信号 δ 越来越小, 也不会让 δ 越来越大, 可以让学习信号比较稳定地从后向前传播, 解决了梯度消失和梯度爆炸的问题。
- ✓ 计算简单, 加速网络的训练。
- ✓ 非线性的激活函数, 可用来处理非线性问题。

问题

- x < 0 时ReLU输出=0, 导数也=0, 丢失一些信号, 神经网络中信号是冗余的, 影响不大。
- x = 0 时不可导。

- 2011年,《Deep Sparse Rectifier Neural Networks》
- 模拟生物神经元的激活函数设计出来的。

生物神经元放电曲线



- ✓ 计算更加高效: 只需加、乘、比较操作。
- ✓ 单侧抑制、宽兴奋边界(兴奋程度可以非常高)
- ✓ 有稀疏性, 适合生物的少活跃状态特点
- ✓ $x > 0$ 时导数=1, 缓解梯度消失问题。

缺点

- 给后一层引入偏置偏移;
- 易“死亡”, 若某个神经元在所有训练数据上不能被激活, 则其自身参数梯度永远=0, 以后也不能被激活。

03-前馈网络-PyTorch.ipynb

④ 评估模型

代码：前馈神经网络 <8>

<8> 绘制训练过程中的指标

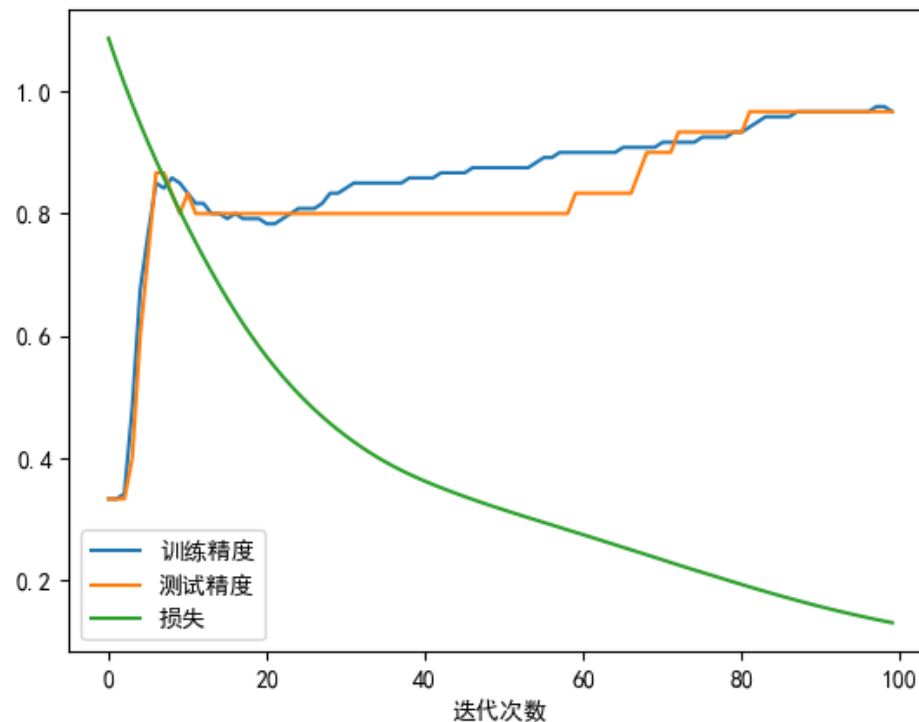
```
import matplotlib.pyplot as plt
from matplotlib import rcParams # 设置全局参数, 为了设置中文字体

# 定义绘制函数
def draw_plot(train_acc_history, test_acc_history, loss_history):
    rcParams['font.family'] = 'simHei' # 设置中文黑体字
    plt.figure()
    plt.plot(train_acc_history, label="训练精度")
    plt.plot(test_acc_history, label="测试精度")
    plt.plot(loss_history, label="损失")
    plt.legend() # 显示图例
    plt.xlabel("迭代次数")
    plt.show()

draw_plot(train_acc_history, test_acc_history, loss_history)
print(f"最终训练精度: {train_acc_history[-1]:.0%}")
print(f"最终测试精度: {test_acc_history[-1]:.0%}")
```

最终训练精度: 97%

最终测试精度: 97%



代码：前馈神经网络 <9>

```
# <9> 模型评估
# 计算测试集的结果
accuracy, predicted = test(model, X_test, y_test)
```

```
# 计算混淆矩阵(Confusion Matrix)
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, predicted)
print(cm)
```

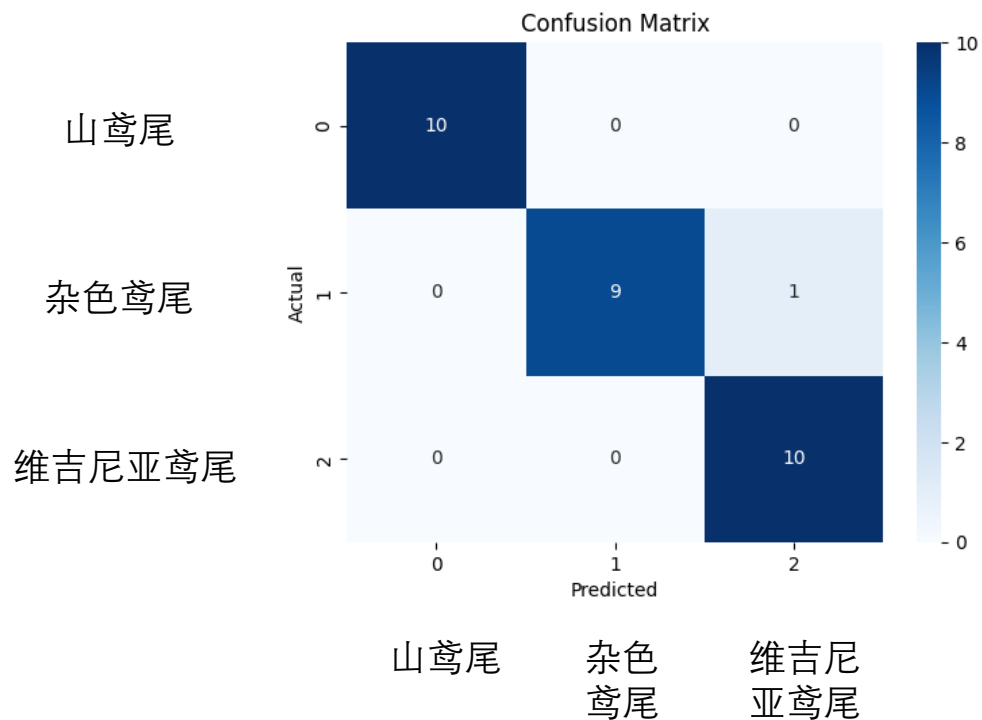
```
[[10  0  0]
 [ 0  9  1]
 [ 0  0 10]]
```

```
import seaborn # > pip install seaborn
# 绘制热力图
seaborn.heatmap(cm, # 混淆矩阵
                 annot=True, # 显示数字
                 cmap='Blues') # 颜色
```

```
import matplotlib.pyplot as plt
# 绘图
plt.xlabel('Predicted') # x轴标签, 预测值
plt.ylabel('Actual') # y轴标签, 真实值
plt.title('Confusion Matrix')
plt.show()
```

混淆矩阵 (Confusion Matrix)

评估分类模型性能的工具，以矩阵形式展示了模型预测结果与实际结果的对比。



准确率

accuracy

所有预测正确的样本占总体样本的比例。 适用：类别均衡的二分类问题。

$$\text{准确率} = \frac{\text{TruePos} + \text{TrueNeg}}{\text{TruePos} + \text{TrueNeg} + \text{FalsePos} + \text{FalseNeg}}$$

数据集：1000张图片，其中：

- 猫（正类）：500张（50%）
- 狗（负类）：500张（50%）

一个模型预测结果：

- 正确识别480只猫(TP)、460只狗(TN)。
- 错误将20只猫预测为狗(FN)、
将40只狗预测为猫(FP)。

$$\text{准确率} = \frac{\text{TP} + \text{TN}}{\text{总样本}} = \frac{480 + 460}{1000} = 94\%$$

TruePos：正确预测为正类的样本数

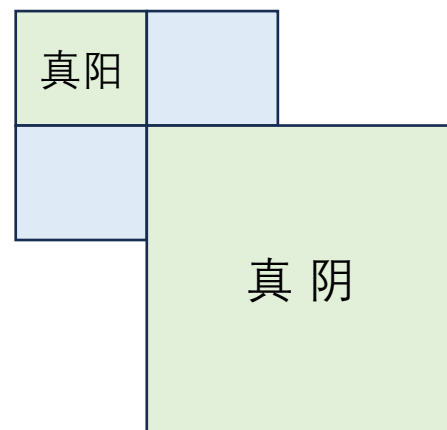
TrueNeg：正确预测为负类的样本数

FalsePos：负类被错误预测为正类的样本数(误报)

FalseNeg：正类被错误预测为负类的样本数(漏报)

设美国全年平均有8亿人次的乘客，2000~2017年共发现了20名恐怖分子。一个模型正确识别出其中10名恐怖分子，准确率：
(10+8亿-20)/8亿

= 99.999999%



精确率

precision

预测为正类的样本中，真实为正类的比例（查准率）。

$$\text{精确率} = \frac{\text{TruePos}}{\text{TruePos} + \text{FalsePos}}$$

TruePos：正确预测为正类的样本数

FalsePos：负类被错误预测为正类的样本数(误报)

- **意义**：关注预测的“准确性”，避免误报 (FP)。
- **适用**：重视减少误报的场景，如垃圾邮件检测 (避免将正常邮件误判为垃圾邮件)。

罕见病检测（患病率为1%）

数据集：1000个样本，其中：

- 正类（患病）： 10人（1%）
- 负类（健康）： 990人（99%）

某模型正确识别8名患者，但误判20名健康人为患者。

$$\text{准确度} = (8 + 970) / 1000 = 97.8\%$$

$$\text{精确度} = 8 / (8 + 20) = 28.6\%$$

		预 测	
		患病 (正类)	健康 (负类)
真 实	患病 (10)	TP 真阳 (8)	FN 假阴 (10-8=2)
	健康 (990)	FP 假阳 (20)	TN 真阴 (970)

召回率

Recall

真实为正类的样本中，被正确预测为正类的比例（查全率）。

$$\text{召回率} = \frac{TP}{TP + FN}$$

TruePos：正确预测为正类的样本数

FalseNeg：正类被错误预测为负类的样本数(漏报)

- **意义**：关注捕捉全部正类的能力，避免漏报 (FN)。
- **适用**：重视减少漏报的场景，如疾病诊断 (避免漏诊患者)。

罕见病检测（患病率为1%）

数据集：1000个样本，其中：

- 正类（患病）： 10人（1%）
- 负类（健康）： 990人（99%）

某模型正确识别8名患者，但误判20名健康人为患者。

$$\text{准确度} = (8 + 970) / 1000 = 97.8\%$$

$$\text{精确度} = 8 / (8 + 20) = 28.6\%$$

$$\text{召回率} = 8 / (8 + 2) = 80\%$$

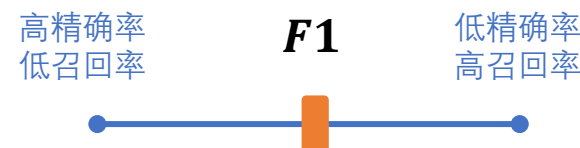
		预 测	
		患病	健康
真 实	患病(正类) (10)	TP 真阳 (8)	FN 假阴 (10-8=2)
	健康(正类) (990)	FP 假阳 (20)	TN 真阴 (970)

F1 值

F1 Score

- **精确率**高意味着较少的误报，即较少的非相关实例被错误地识别为相关。
- **召回率**高意味着较少的漏报，即更多的相关实例被正确地识别出来。

F1值：精确度和召回率的调和平均数，综合两者表现。



$$F_1 = \frac{2}{\frac{1}{\text{精度}} + \frac{1}{\text{召回率}}} = \frac{2TP}{TP + FP + FN}$$

TruePos：正确预测为正类的样本数

FalsePos：负类被错误预测为正类的样本数(误报)

FalseNeg：正类被错误预测为负类的样本数(漏报)

意义：平衡精确度和召回率，适用于两者需同时优化的场景（如医学检测、欺诈识别）。

罕见病检测（患病率为1%）

数据集：1000个样本，其中：

- 正类（患病）： 10人（1%）
- 负类（健康）： 990人（99%）

某模型正确识别8名患者，但误判20名健康人为患者。

$$\text{准确度} = (8 + 970) / 1000 = 97.8\%$$

$$\text{精确度} = 8 / (8 + 20) = 28.6\%$$

$$\text{召回率} = 8 / (8 + 2) = 80\%$$

$$\text{F1值} = 2 * (0.286 * 0.8) / (0.286 + 0.8) \approx \mathbf{42.1\%}$$



1. 数字图像

2. 全连接网络的问题

- 手写数字识别 sklearn digits
- 手写数字识别 MNIST数据集

3. 卷积神经网络 CNN

- 卷积、填充、池化、GPU
- LeNet-5