SVM 的数学推导和 Python 实现

赵新锋

2021 年 7 月 10 日

摘要

支持向量机（support vector machines, SVM）是一种分类模型，该模型在特征空间中求解间隔最大的分类超平面。

关键词： 支持向量机; SVM; SMO; 矩阵运算; 矩阵求导;numpy;sklearn.

目录

1 数学推导与 python 实现

1.1 线性可分

三层网络简单说明如下:

- 输入层，不做处理，只是输入数据
- 隐藏层，单元数为 n_h 我们就以 RELU 为激活函数，另外一个常用的是 sigmoid，我们把激活函数封装成一个类，可以随时替换
- 输出层为一个单元，由于是二分类，我们就用 sigmoid 拟合对应类 1 的概率

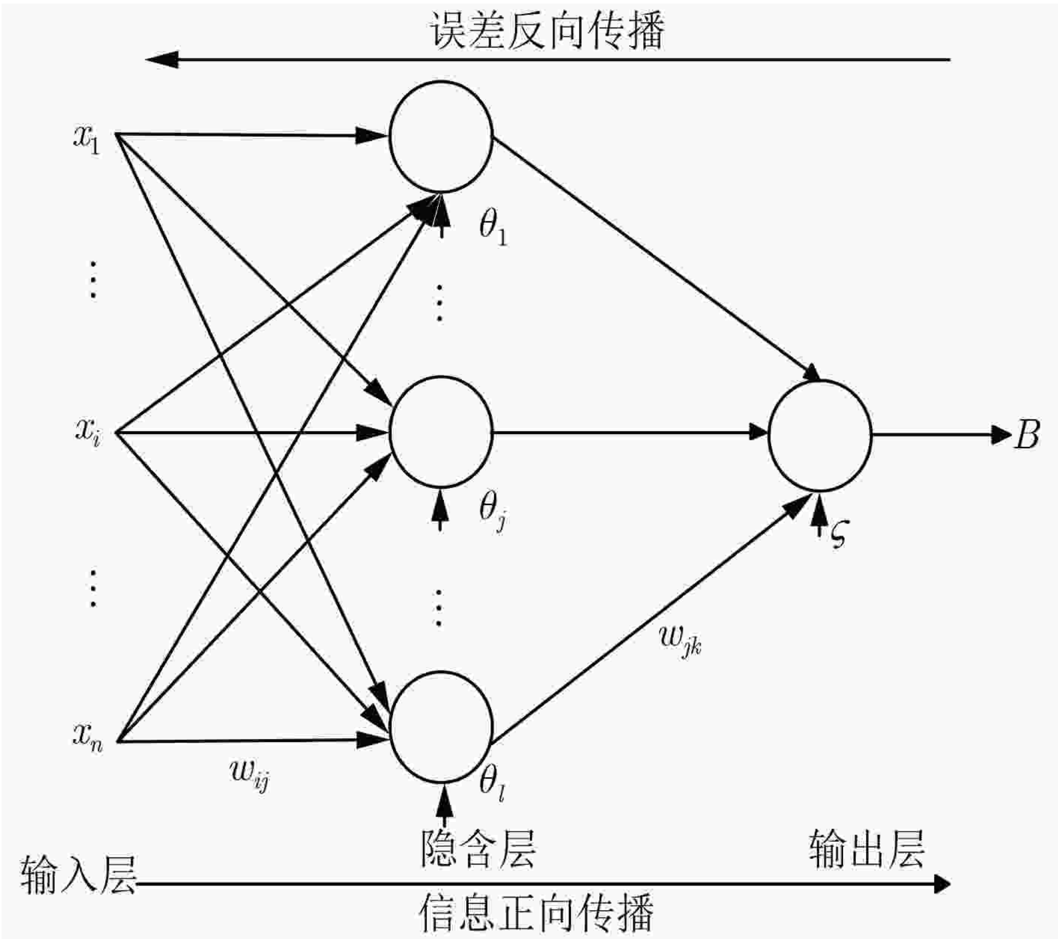

图 1: 神经网络示意图

- X 表示输入样本向量矩阵，Y 表示样本集的期望输出标记 (有可能是 0/1 标记的向量，有可能是 onehot 的矩阵)，本文示例由于是二分类，那么 Y 的每个元素要么是 0 要么是 1，Y 是 m 个元素的向量, 但是在运算中为了通用将其转换为 $m \times 1$ 的矩阵。
- m 为样本数量,n 为样本向量的维度。

- 激活函数用 f 表示，隐藏层激活函数 f_h 使用 RELU 函数:$f_h(x) = max(0, x)$，输出层 f_o 为 sigmoid 函数，输出为 1 类的概率 $f_o(x) = \frac{1}{1+exp(-x)}$
- 隐藏层输出为 Y_h, 输出层的输出为 Y_o
- 隐藏层单元的参数为 w_h，N_h 个隐藏层单元的参数矩阵为 W_h, 输出层单元的参数为 w_o，N_o 个输出层单元的参数矩阵为 W_o。
- 虽然本文使用二分类即输出单元为 1 个，但是仍然使用矩阵 Yo 和 W_o 表示，从而使得算法是通用的，改成 onehot+softmax 方式的多分类，也是适用的。
- 从输入到输出的公式：$Y_o = f_o(f_h(X \cdot W_h) \cdot Wo)$
- 注：这里的线性变换 $X \cdot W_h$ 并没有使用偏置，通过在 X 增加一个全 1 的列，达到同样的效果。

1.2 公式推导

单个样本的运算过程表示如下：

$$\begin{aligned}
\arg\min_{\theta} d &= \frac{y_0 \cdot (x_0^T \cdot w_0 + b_0)}{\|w_0\|_2} \\
\text{s.t.} \quad & \frac{y_i \cdot (x_i^T \cdot w_0 + b_0)}{\|w_0\|_2} \geq d \\
w &= \frac{w_0}{y_0 \cdot (x_0^T \cdot w_0 + b_0)} \\
b &= \frac{b_0}{y_0 \cdot (x_0^T \cdot w_0 + b_0)} \\
d &= \frac{1}{\|w\|_2} \\
\text{s.t.} \quad & y_i \cdot (x_i^T \cdot w + b) \geq 1
\end{aligned}$$

批量样本利用矩阵表示运算过程如下:

$$\begin{aligned}
X_h &= X \cdot W_h \\
Y_h &= f_h(X_h) \\
X_o &= Y_h \cdot W_o \\
Y_o &= f_o(X_o) \\
&= f_o(Y_h \cdot W_o) \\
&= f_o(f_h(X_h) \cdot W_o) \\
&= f_o(f_h(X \cdot W_h) \cdot Wo) \qquad (1)
\end{aligned}$$

其中 (1) 式即为信息正向传导的向量化公式相应的 python 代码实现：

```
#forward
Lh = np.dot(X, Wh)
Yh = funcActivation.cal(Lh) #隐藏层输出
Yh = np.insert(Yh, np.shape(Yh)[1], values=np.ones(m), axis=1)
Lo = np.dot(Yh, Wo)
Yo = funcOut.cal(Lo)
```

其中 funcActivation 为隐藏层激活函数的封装，这里使用的 Relu，funcOut 为输出层激活函数的封装，使用的是 sigmoid。

```
class ReLU:
    def cal(self, z):
        return np.clip(z, 0, np.inf)
    def grad(self, x):
        return (x > 0).astype(int)
class Sigmoid:
    def cal(self, z):
        return 1 / (1 + np.exp(-z))
    def grad(self, x):
        z = self.cal(x)
        return z*(1-z)
```

1.3 损失函数

最常用的有 Mean Square Error 均方差损失函数，用于分类的 Cross Entropy 等。本文以经典的 MSE 为例。如下 (2) 式是 MSE 损失函数的向量化表示。

$$\begin{aligned} e &= \frac{1}{2m}\sum_{i=1}^{m}(y_{oi}-y_i)^2 \\ &= \frac{1}{2m}(Y_o-Y)^T\cdot(Y_o-Y) \end{aligned} \tag{2}$$

1.4 误差反向传播

我们目的是最小化损失函数，通常使用梯度下降法，逐渐逼近最小极值点。需要逐渐求解参数主要是两个：W_h 和 W_o，下面分别计算对应损失函数的偏导数。

$$
\begin{aligned}
\mathrm{d}e &= \frac{1}{2m} tr((\mathrm{d}Y_o)^T \cdot (Y_o - Y) + (Y_o - Y)^T \cdot \mathrm{d}Y_o) \\
&= \frac{1}{2m} tr((\mathrm{d}Y_o)^T \cdot (Y_o - Y)) + \frac{1}{2m} tr((Y_o - Y)^T \cdot \mathrm{d}Y_o) \\
&= \frac{1}{2m} tr(\mathrm{d}Y_o \cdot (Y_o - Y)^T) + \frac{1}{2m} tr((Y_o - Y)^T \cdot \mathrm{d}Y_o) \\
&= \frac{1}{m} tr((Y_o - Y)^T \cdot \mathrm{d}Y_o) \qquad (3) \\
&= \frac{1}{m} tr((Y_o - Y)^T \cdot (f_o'(X_o) \odot \mathrm{d}X_o)) \qquad (4) \\
&= \frac{1}{m} tr(((Y_o - Y) \odot f_o'(X_o))^T \cdot \mathrm{d}X_o) \qquad (5) \\
&= \frac{1}{m} tr(((Y_o - Y) \odot f_o'(X_o))^T \cdot (Y_h \cdot \mathrm{d}W_o + \mathrm{d}Y_h \cdot W_o)) \qquad (6)
\end{aligned}
$$

其中 (3) 式利用迹 $tr(A^T) = tr(A)$ 的性质，所以前后两项一样 (4) 式到 (5) 式利用了迹 $tr(A^T \cdot (B \odot C)) = tr((A \odot B)^T \cdot C)$ 的性质，其中 $\odot$ 表示矩阵各个元素相乘，对应 numpy 的 multiply。神经网络梯度下降求解参数的时候，是从输出层到隐藏层逆着计算的，所以称之为“反向传播”，因此首先求对 W_o 的偏导，此时 Y_h 相当于常数，故 (6) 式的 $\mathrm{d}Y_h$ 忽略，继续推导：

$$
\begin{aligned}
\mathrm{d}e &= \frac{1}{m} tr(((Y_o - Y) \odot f_o'(X_o))^T \cdot (Y_h \cdot \mathrm{d}W_o)) \\
&= \frac{1}{m} tr((Y_h^T \cdot ((Y_o - Y) \odot f_o'(X_o)))^T \cdot \mathrm{d}W_o) \qquad (7) \\
\frac{\partial e}{\partial W_o} &= \frac{1}{m} Y_h^T \cdot ((Y_o - Y) \odot f_o'(X_o)) \qquad (8)
\end{aligned}
$$

这里 (7) 式到 (8) 式利用了标量对向量或矩阵微分 $\mathrm{d}f = tr((\frac{\partial f}{\partial x})^T \mathrm{d}x)$ 的性质。对应 python 实现：

```
delta = np.multiply(self.lossFunc.grad(Yo, Y) , funcOut.grad(Lo))
dO  = np.dot(Xo.T, delta)
```

同理继续推导 $\frac{\partial e}{\partial W_h}$ ：

$$
\begin{aligned}
\mathrm{d}e &= \frac{1}{m}tr(((Y_o - Y) \odot f_o^{'}(X_o))^T \cdot (\mathrm{d}Y_h \cdot W_o)) && (9)\\
&= \frac{1}{m}tr(W_o \cdot ((Y_o - Y) \odot f_o^{'}(X_o))^T \cdot \mathrm{d}Y_h)\\
&= \frac{1}{m}tr(((Y_o - Y) \odot f_o^{'}(X_o) \cdot W_o^T)^T \cdot \mathrm{d}Y_h)\\
&= \frac{1}{m}tr(((Y_o - Y) \odot f_o^{'}(X_o) \cdot W_o^T)^T \cdot (f_h^{'}(X_h) \odot \mathrm{d}X_h))\\
&= \frac{1}{m}tr((((Y_o - Y) \odot f_o^{'}(X_o) \cdot W_o^T) \odot f_h^{'}(X_h))^T \cdot \mathrm{d}(X_h)) && (10)\\
&= \frac{1}{m}tr((((Y_o - Y) \odot f_o^{'}(X_o) \cdot W_o^T) \odot f_h^{'}(X_h))^T \cdot \mathrm{d}(X \cdot W_h))\\
&= \frac{1}{m}tr((X^T \cdot (((Y_o - Y) \odot f_o^{'}(X_o) \cdot W_o^T) \odot f_h^{'}(X_h)))^T \cdot \mathrm{d}W_h)\\
\frac{\partial e}{\partial W_h} &= \frac{1}{m}X^T \cdot (((Y_o - Y) \odot f_o^{'}(X_o) \cdot W_o^T) \odot f_h^{'}(X_h)) && (11)
\end{aligned}
$$

对应的 python 的实现：

```
delta = np.multiply(self.lossFunc.grad(Yo, Y) , funcOut.grad(Lo))
dH = np.dot(X.T, np.multiply(np.dot(delta, Wo.T[:,:-1]) , funcActivation.
    grad(Lh)))
```

1.5 梯度下降

神经网络的学习过程，即最小化损失函数的过程，是通过梯度下降法迭代求解参数的。其中下式的 η 为学习速率。太大可能出现震荡或不收敛, 太小可能收敛的太慢。

$$
\begin{aligned}
\frac{\partial e}{\partial W_o} &= \frac{1}{m}Y_h^T \cdot ((Y_o - Y) \odot f_o^{'}(X_o))\\
\frac{\partial e}{\partial W_h} &= \frac{1}{m}X^T \cdot (((Y_o - Y) \odot f_o^{'}(X_o) \cdot W_o^T) \odot f_h^{'}(X_h))\\
W_o &= W_o - \eta\frac{\partial e}{\partial W_o}\\
W_h &= W_h - \eta\frac{\partial e}{\partial W_h}
\end{aligned}
$$

2 使用偏置 b 的数学推导

以上是使用通过 X 增加全 1 的列的方式，省略偏置 b, 下面再推导一下使用偏置的方式。

2.1 forward 公式推导

单个样本的运算过程表示如下：

$$\begin{aligned}
x_{hj} &= x_i \cdot w_{hj} + b_{hj} \\
y_{hj} &= f_h(x_{hj}) \\
x_o &= y_{hj} \cdot w_o + b_{oj} \\
y_o &= f_o(x_o)
\end{aligned}$$

批量样本利用矩阵表示运算过程如下:

$$\begin{aligned}
X_h &= X \cdot W_h + B_h \\
Y_h &= f_h(X_h) \\
X_o &= Y_h \cdot W_o + B_o \\
Y_o &= f_o(X_o) \\
&= f_o(Y_h \cdot W_o + B_o) \\
&= f_o(f_h(X_h + B_h) \cdot W_o + B_o) \\
&= f_o(f_h(X \cdot W_h + B_h) \cdot Wo + B_o)
\end{aligned} \tag{12}$$

2.2 偏导公式推导

使用 MSE 的损失函数，矩阵运算形式的微分运算如下：

$$\begin{aligned}
de &= \frac{1}{m} tr(((Y_o - Y) \odot f_o^{'}(X_o))^T \cdot \mathrm{d}X_o) \\
&= \frac{1}{m} tr(((Y_o - Y) \odot f_o^{'}(X_o))^T \cdot \mathrm{d}(Y_h \cdot W_o + B_o)) \\
&= \frac{1}{m} tr(((Y_o - Y) \odot f_o^{'}(X_o))^T \cdot (\mathrm{d}(Y_h \cdot W_o) + \mathrm{d}B_o))
\end{aligned}$$

其中 $\frac{\partial e}{\partial W_o}$ 跟 (8) 式的结果相同，继续推导 $\frac{\partial e}{\partial B_o}$ ：

$$\begin{aligned}
de &= \frac{1}{m} tr(((Y_o - Y) \odot f_o^{'}(X_o))^T \cdot \mathrm{d}B_o) \\
\frac{\partial e}{\partial W_o} &= \frac{1}{m} Y_h^T \cdot ((Y_o - Y) \odot f_o^{'}(X_o)) \\
\frac{\partial e}{\partial B_o} &= \frac{1}{m} (Y_o - Y) \odot f_o^{'}(X_o)
\end{aligned}$$

下面推导 $\frac{\partial e}{\partial W_o}$ 和 $\frac{\partial e}{\partial B_h}$：

$$
\begin{aligned}
de &= \frac{1}{m}tr((((Y_o - Y) \odot f_o'(X_o) \cdot W_o^T) \odot f_h'(X_h))^T \cdot \mathrm{d}(X_h)) \\
&= \frac{1}{m}tr((((Y_o - Y) \odot f_o'(X_o) \cdot W_o^T) \odot f_h'(X_h))^T \cdot \mathrm{d}(X \cdot W_h + B_h))
\end{aligned}
$$

显然 $\frac{\partial e}{\partial W_h}$ 跟 (11) 式的结果相同，继续推导 $\frac{\partial e}{\partial B_o}$：

$$
\begin{aligned}
de &= \frac{1}{m}tr((((Y_o - Y) \odot f_o'(X_o) \cdot W_o^T) \odot f_h'(X_h))^T \cdot \mathrm{d}(X_h)) \\
&= \frac{1}{m}tr((((Y_o - Y) \odot f_o'(X_o) \cdot W_o^T) \odot f_h'(X_h))^T \cdot \mathrm{d}B_h) \\
\frac{\partial e}{\partial W_h} &= \frac{1}{m}X^T \cdot (((Y_o - Y) \odot f_o'(X_o) \cdot W_o^T) \odot f_h'(X_h)) \\
\frac{\partial e}{\partial B_h} &= \frac{1}{m}((Y_o - Y) \odot f_o'(X_o) \cdot W_o^T) \odot f_h'(X_h)
\end{aligned}
$$

2.3 梯度下降

相比不使用偏置的梯度下降，多了两个 B 参数的梯度下降。

$$
\begin{aligned}
\frac{\partial e}{\partial W_o} &= \frac{1}{m}Y_h^T \cdot ((Y_o - Y) \odot f_o'(X_o)) \\
\frac{\partial e}{\partial B_o} &= \frac{1}{m}(Y_o - Y) \odot f_o'(X_o) \\
\frac{\partial e}{\partial W_h} &= \frac{1}{m}X^T \cdot (((Y_o - Y) \odot f_o'(X_o) \cdot W_o^T) \odot f_h'(X_h)) \\
\frac{\partial e}{\partial B_h} &= \frac{1}{m}((Y_o - Y) \odot f_o'(X_o) \cdot W_o^T) \odot f_h'(X_h) \\
W_o &= W_o - \eta\frac{\partial e}{\partial W_o} \\
W_h &= W_h - \eta\frac{\partial e}{\partial W_h} \\
B_o &= B_o - \eta\frac{\partial e}{\partial B_o} \\
B_h &= B_h - \eta\frac{\partial e}{\partial B_h}
\end{aligned}
$$

3 总结

3.1 多层网络

本文实现的是基于输入层、隐藏层、输出层经典的三层神经网络架构，在此基础上可以实现更多隐藏层的神经网络。我们观察隐藏层的参数基于损失函数的偏导 (11) 式与输出层的 (8) 式区别:

$$\frac{\partial e}{\partial W_h} = X^T \cdot (\overbrace{\frac{1}{m}((Y_o - Y) \odot f_o'(X_o) \cdot W_o^T)}^{\delta_o} \odot f_h'(X_h))$$

$$\frac{\partial e}{\partial W_o} = Y_h^T \cdot (\overbrace{\frac{1}{m}(Y_o - Y)}^{\delta} \odot f_o'(X_o))$$

我们观察到规律，$\frac{\partial e}{\partial W_h}$ 可以分成 3 项, X^T 项是自己这一层的输入，δ_o 为上一层的反馈，第 3 项为本层激活函数的导数，同理，$\frac{\partial e}{\partial W_h}$ 也可以看出三项，其中 δ 为损失函数的导数，我们可以把他看成输出层的上一层的反馈值，所以每层的参数矩阵的偏导可以统一公式为:

$$\frac{\partial e}{\partial W_i} = X_i^T \cdot (\delta_{i+1} \odot f_i'(X_i \cdot W_i))$$

3.2 Linear regression

如果把隐藏层去掉，输出层的激活函数也去掉，那么只有一层的神经网络就是多元线性回归。

$$Y = X \cdot w + b$$

3.3 Logistics regression

如果把隐藏层去掉，输出层的激活函数使用 sigmoid，那么只有一层的神经网络就是就变成了逻辑回归。

$$Y = \frac{1}{1 + exp(X \cdot w + b)}$$

3.4 关于隐藏层数量的选择

有一些启发式的公式选择隐藏层数量，但是主要还是通过实验选择最恰当的参数，这里可以使用交叉验证的方式。如将训练样本 3/7 分割为验证数据集和训练数据集，然后先从少的隐藏层数量开始试，选择在训练数据集上拟合效果好，同时在验证数据集上同样具有接近准确度的参数作为最终参数结果。

参考文献

[1] Ian Goodfellow / Yoshua Bengio / Aaron Courville . *Deep Learning*[M]. 北京: 清华大学出版社,2017-08-01.

[2] 张贤达. 矩阵分析与应用 [M]. 北京: 人民邮电出版社,2013-11-01.

[3] 李航. 统计学习方法 [M]. 北京: 清华大学出版社,2012-03.

[4] David C. Lay / Steven R. Lay / Judi J. McDonald . 线性代数及其应用 [M]. 北京: 机械工业出版社,2018-07.

[5] 史蒂文 · J. 米勒. 普林斯顿概率论读本 [M]. 北京: 人民邮电出版社,2020-08.

A 附录使用偏置 b 的 python 源码

使用偏置 b 的 python 源码实现，只使用 sklearn 库：

```
import numpy as np
from sklearn import datasets
from sklearn.datasets import load_breast_cancer
from sklearn import preprocessing
import matplotlib.pyplot as plt
from sklearn.metrics import r2_score

dataset = None
DT_FLAG = 1 #1测试预测，2测试回归拟合

if DT_FLAG:
    dataset = load_breast_cancer()
else:
    dataset = datasets.load_boston()

class ReLU:
    def __init__(self):
        return
    def cal(self, z):
        return np.clip(z, 0, np.inf)
    def grad(self, x):
        return (x > 0).astype(int)
class Sigmoid:
    def __init__(self):
        return
    def cal(self, z):
        return 1 / (1 + np.exp(-z))
    def grad(self, x):
        z = self.cal(x)
        return z*(1-z)

class LossMSE:
    def __init__(self):
        return
    def loss(self, y_pred, y):
        return np.sum(np.multiply((y_pred - y) , (y_pred - y))) / 2 /int(y.
            shape[0])
    def grad(self, y_pred, y):
        return (y_pred - y)/y.shape[0]
class LossCrossEntropy:
    def loss(self, y_pred, y):
        eps = np.finfo(float).eps
        cross_entropy = -np.sum(y * np.log(y_pred + eps))
        return cross_entropy
    def grad(self, y_pred, y):
```

```
        grad = y_pred - y
        return grad

class LinearOut:#如果用于回归，只是线性输出，不做任何转换，为了可以有统一的
    结构
    def cal(self, z):
        return z
    def grad(self, x):
        self.gradCache = np.ones(np.shape(x))
        return self.gradCache

class NN:
    def __init__(self):
        self.Wh = None #隐藏层参数w
        self.Wo = None #输出层参数w
        self.numHideUnit = 5
        self.funcActivation = ReLU() #隐藏层激活函数
        if DT_FLAG:
            self.lossFunc = LossMSE()
            self.funcOut = Sigmoid()
        else:
            self.lossFunc = LossMSE()
            self.funcOut = LinearOut()
        return
    def fit(self, X, Y):
        scalerX = preprocessing.StandardScaler().fit(X)#StandardScaler
        Y = Y.reshape(-1, 1)
        X = scalerX.transform(X)
        if not DT_FLAG:
            scalerY = preprocessing.StandardScaler().fit(Y)#MinMaxScaler
            Y = scalerY.transform(Y)

        m, n = np.shape(X)

        funcActivation = self.funcActivation
        funcOut = self.funcOut
        # 10 1000 0.01 for breat cancer
        numLayerHide = 5
        maxIterTimes = 2000
        eta  = 0.01
        if DT_FLAG:
            eta  = 10
        else:
            #maxIterTimes = 100000
            eta  = 0.5#eta  = 0.00001
            pass

        outputNodeNum = 1
```

```
        Wh = np.random.rand(n, numLayerHide) * 0.01
        Bh = np.random.rand(numLayerHide) * 0.01
        Wo = np.random.rand(numLayerHide, outputNodeNum) * 0.01 #Why +1?
            for reserver b
        Bo = np.random.rand(outputNodeNum) * 0.01

        errLog = []
        dO_Old = 0
        dH_Old = 0
        for i in range(maxIterTimes):
            #forward
            Lh = np.dot(X, Wh) + Bh
            Yh = funcActivation.cal(Lh) #隐藏层输出

            Lo = np.dot(Yh, Wo) + Bo
            Yo = funcOut.cal(Lo)

            loss = self.lossFunc.loss(Yo, Y)
            errLog.append(loss)
            if loss < 0.001:
                break

            delta = np.multiply(self.lossFunc.grad(Yo, Y) , funcOut.grad(Lo
                ))

            dBo  = delta
            dWo  = np.dot(Yh.T, dBo)

            dBh = np.multiply(np.dot(delta, Wo.T) , funcActivation.grad(Lh)
                )
            dWh = np.dot(X.T, dBh)

            Wo = Wo - eta * dWo
            Wh = Wh - eta * dWh
            Bo = Bo - eta * dBo
            Bh = Bh - eta * dBh
        score = 0
        if DT_FLAG == 0:
            score = r2_score(Yo, Y)
        else:
            score = r2_score((Yo > 0.5).astype(int), Y)
        print('score', score)
        return
if __name__ == '__main__':
    nn = NN()
    print('data', dataset.data.shape)
    nn.fit(dataset.data, dataset.target)
```