人工神经网络——反向传播算法（BP）以及Python实现

1821011728 张双桂

人工神经网络是模拟生物神经系统的一种计算机程序。神经元之间是通过轴突、树突互相连接的，神经元收到刺激时。神经脉冲在神经元之间传播，同时反复的脉冲刺激，使得神经元之间的联系加强。受此启发，人工神经网络中神经元之间的联系（权值）也是通过反复的数据信息“刺激”而得到调整的。而反向传播算法就是用来调整权值的。

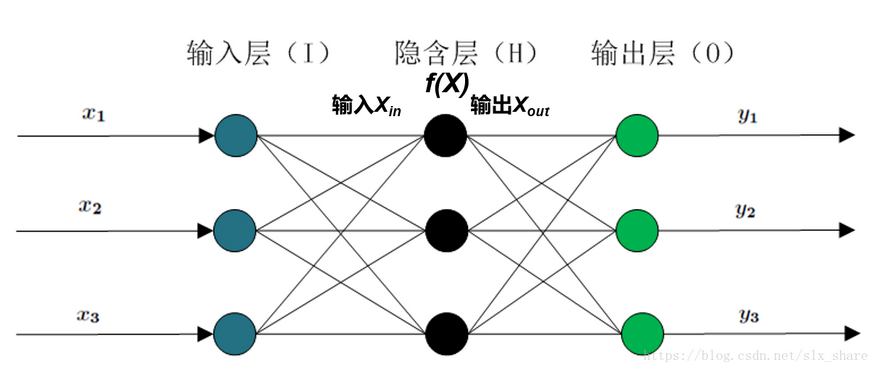
**核心思想**

训练误差逐层反向传播，每层神经元与下层神经元间权重通过误差最速梯度下降的方法调整。

**算法简介**

**人工神经网络简介**

通常人工神经网络分为输入层、隐藏层和输出层。隐藏层和输出层都有相应的激励函数（模拟神经脉冲的激励方式）。隐藏层可以为多层，但实际不超过三层为好。如图



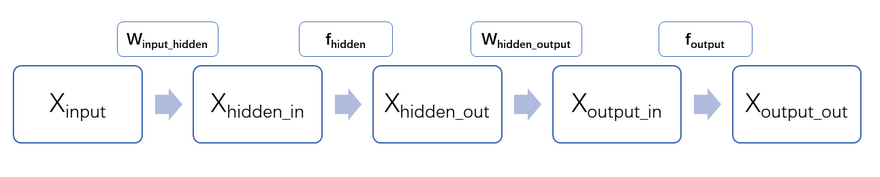
常用的激励函数有sigmoid和linear函数。如果使用神经网络来做分类的工作，那么输出层激励函数通常采用sigmoid函数，若做回归工作，就采用linear函数。输入层和输出层神经元数目由训练数据集而定，而隐藏层神经元数目的设定带有很大的经验性，对经验公式（为输入层、输出层神经元数目）向上取整的数值可作为隐藏层神经元数目。

**人工神经网络的前向传播**

训练数据包含的信息逐层从输入层传播到输出层的方式是以感知机为基础的。即

**误差反向传播**

与前向传播正好相反，训练误差都是从输出层开始反向传播到每个隐藏层。有了每一层的误差就好办了，训练的目的就是减小误差，即最小化期望风险，通常采用平方误差损失函数，采用最速梯度下降法更新权值。根据链式法则计算每层权值调整梯度：



**输出层梯度：**

即输出层的误差。

**隐藏层到输出层权值梯度：**

其中**，，**

这里**，**相当于反向传播到隐藏层的误差**。**

**隐藏层到输出层权值更新：**

**输入层到隐藏层权值更新：**

注：\*表示对应元素相乘，其他为矩阵乘法，f为激励函数，为动量项前系数。为学习率

**算法流程**

* Input:训练数据集X\_data和Y\_data，神经网络必要参数，阈值epsilon,最大迭代次数maxstep
* Output:神经网络权值W
* Step1:初始化权值矩阵W，阈值b
* Step2:迭代训练直到误差小于阈值epsilon或达到最大迭代次数maxstep.训练步骤包括前向传播和误差反向传播。

**代码**

**"""**

**BP神经网络**

**"""**

**import numpy as np**

**import math**

**# 激励函数及相应导数，后续可添加**

**def sigmoid(x):**

**return 1.0 / (1 + np.exp(-x))**

**def diff\_sigmoid(x):**

**fval = sigmoid(x)**

**return fval \* (1 - fval)**

**def linear(x):**

**return x**

**def diff\_linear(x):**

**return np.ones\_like(x)**

**class BP:**

**def \_\_init\_\_(self, n\_hidden=None, f\_hidden='sigmoid', f\_output='sigmoid',**

**epsilon=1e-3, maxstep=1000, eta=0.1, alpha=0.0):**

**self.n\_input = None # 输入层神经元数目**

**self.n\_hidden = n\_hidden # 隐藏层神经元数目**

**self.n\_output = None**

**self.f\_hidden = f\_hidden**

**self.f\_output = f\_output**

**self.epsilon = epsilon**

**self.maxstep = maxstep**

**self.eta = eta # 学习率**

**self.alpha = alpha # 动量因子**

**self.wih = None # 输入层到隐藏层权值矩阵**

**self.who = None # 隐藏层到输出层权值矩阵**

**self.bih = None # 输入层到隐藏层阈值**

**self.bho = None # 隐藏层到输出层阈值**

**self.N = None**

**def init\_param(self, X\_data, y\_data):**

**# 初始化**

**if len(X\_data.shape) == 1: # 若输入数据为一维数组，则进行转置为n维数组**

**X\_data = np.transpose([X\_data])**

**self.N = X\_data.shape[0]**

**# normalizer = np.linalg.norm(X\_data, axis=0)**

**# X\_data = X\_data / normalizer**

**if len(y\_data.shape) == 1:**

**y\_data = np.transpose([y\_data])**

**self.n\_input = X\_data.shape[1]**

**self.n\_output = y\_data.shape[1]**

**if self.n\_hidden is None:**

**self.n\_hidden = int(math.ceil(math.sqrt(self.n\_input + self.n\_output)) + 2)**

**self.wih = np.random.rand(self.n\_input, self.n\_hidden) # i\*h**

**self.who = np.random.rand(self.n\_hidden, self.n\_output) # h\*o**

**self.bih = np.random.rand(self.n\_hidden) # h**

**self.bho = np.random.rand(self.n\_output) # o**

**return X\_data, y\_data**

**def inspirit(self, name):**

**# 获取相应的激励函数**

**if name == 'sigmoid':**

**return sigmoid**

**elif name == 'linear':**

**return linear**

**else:**

**raise ValueError('the function is not supported now')**

**def diff\_inspirit(self, name):**

**# 获取相应的激励函数的导数**

**if name == 'sigmoid':**

**return diff\_sigmoid**

**elif name == 'linear':**

**return diff\_linear**

**else:**

**raise ValueError('the function is not supported now')**

**def forward(self, X\_data):**

**# 前向传播**

**x\_hidden\_in = X\_data @ self.wih + self.bih # n\*h**

**x\_hidden\_out = self.inspirit(self.f\_hidden)(x\_hidden\_in) # n\*h**

**x\_output\_in = x\_hidden\_out @ self.who + self.bho # n\*o**

**x\_output\_out = self.inspirit(self.f\_output)(x\_output\_in) # n\*o**

**return x\_output\_out, x\_output\_in, x\_hidden\_out, x\_hidden\_in**

**def fit(self, X\_data, y\_data):**

**# 训练主函数**

**X\_data, y\_data = self.init\_param(X\_data, y\_data)**

**step = 0**

**# 初始化动量项**

**delta\_wih = np.zeros\_like(self.wih)**

**delta\_who = np.zeros\_like(self.who)**

**delta\_bih = np.zeros\_like(self.bih)**

**delta\_bho = np.zeros\_like(self.bho)**

**while step < self.maxstep:**

**step += 1**

**# 向前传播**

**x\_output\_out, x\_output\_in, x\_hidden\_out, x\_hidden\_in = self.forward(X\_data)**

**if np.sum(abs(x\_output\_out - y\_data)) < self.epsilon:**

**break**

**# 误差反向传播，依据权值逐层计算当层误差**

**err\_output = y\_data - x\_output\_out # n\*o， 输出层上，每个神经元上的误差**

**delta\_ho = -err\_output \* self.diff\_inspirit(self.f\_output)(x\_output\_in) # n\*o**

**err\_hidden = delta\_ho @ self.who.T # n\*h， 隐藏层（相当于输入层的输出），每个神经元上的误差**

**# 隐藏层到输出层权值及阈值更新**

**delta\_bho = np.sum(self.eta \* delta\_ho + self.alpha \* delta\_bho, axis=0) / self.N**

**self.bho -= delta\_bho**

**delta\_who = self.eta \* x\_hidden\_out.T @ delta\_ho + self.alpha \* delta\_who**

**self.who -= delta\_who**

**# 输入层到隐藏层权值及阈值的更新**

**delta\_ih = err\_hidden \* self.diff\_inspirit(self.f\_hidden)(x\_hidden\_in) # n\*h**

**delta\_bih = np.sum(self.eta \* delta\_ih + self.alpha \* delta\_bih, axis=0) / self.N**

**self.bih -= delta\_bih**

**delta\_wih = self.eta \* X\_data.T @ delta\_ih + self.alpha \* delta\_wih**

**self.wih -= delta\_wih**

**return**

**def predict(self, X):**

**# 预测**

**res = self.forward(X)**

**return res[0]**

**if \_\_name\_\_ == '\_\_main\_\_':**

**import matplotlib.pyplot as plt**

**N = 100**

**X\_data = np.linspace(-1, 1, N)**

**X\_data = np.transpose([X\_data])**

**y\_data = np.exp(-X\_data) \* np.sin(2 \* X\_data)**

**bp = BP(f\_output='linear', maxstep=2000, eta=0.01, alpha=0.1) # 注意学习率若过大，将导致不能收敛**

**bp.fit(X\_data, y\_data)**

**plt.plot(X\_data, y\_data)**

**pred = bp.predict(X\_data)**

**plt.scatter(X\_data, pred, color='r')**

**plt.show()**