神经网络反向传播算法

张觉非
All IS ONE

> 本文内容都包含在拙作《深入理解神经网络》中

《深入理解神经网络 从逻辑回归到CNN》(张觉非)【摘要 书评 试读】- 京东图书
item.jd.com

> 一、符号与表示

本文介绍全连接人工神经网络的训练算法——反向传播算法（*关于人工神经网络的简单介绍，可参考“卷积神经网络简介”第二节*）。反向传播算法本质上是梯度下降法（*参考“上篇”*）。人工神经网络的参数多，梯度计算比较复杂。在人工神经网络模型提出几十年后才有研究者提出了反向传播算法来解决深层参数的训练问题。本文将详细讲解该算法的原理及实现。

首先把文中用来表示神经网络的各种符号描述清楚。文中向量用粗体小写字母表示，矩阵用粗体大写字母表示，非粗体都是标量。向量或矩阵的转置用上标 T 表示。上标括号中的数字表示神经网络的层序号。下标的含义结合上下文自明。请看图 1.1 。

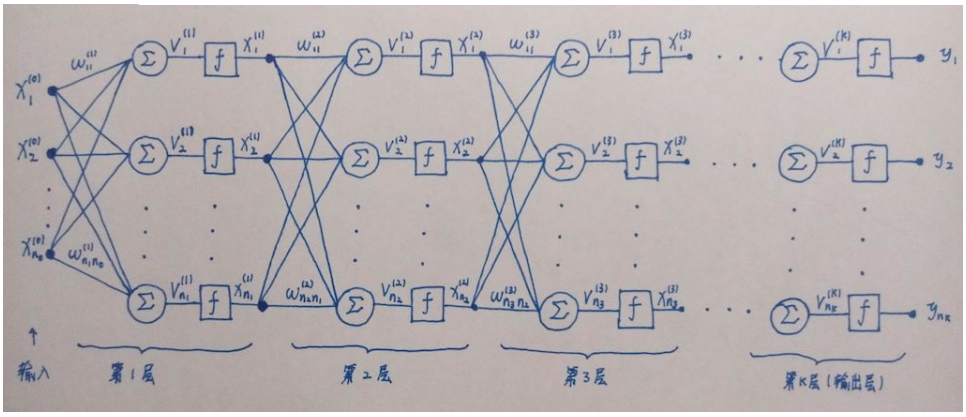


图 1.1 多层全连接神经网络

图 1.1 描绘了一个多层全连接神经网络。该网络共有 K（*大写*）层。第 k（*小写*）层包含 n_k 个神经元。最后一层——第 K 层是输出层。第 K 层的输出 $\mathbf{y}=(y_1,y_2,\cdots,y_{n_K})^T$ 是神经网络的输出向量。该神经网络接受 n_0 个输入 $\mathbf{x}=\left(x_1^{(0)},x_2^{(0)},\cdots,x_{n_0}^{(0)}\right)^T$。输入向量视作网络的第 0 层。网络的第 k 层（$0<k<K$）是隐藏层。网络第 k 层的第 j 个神经元以及它的前后连接如图 1.2 所示。

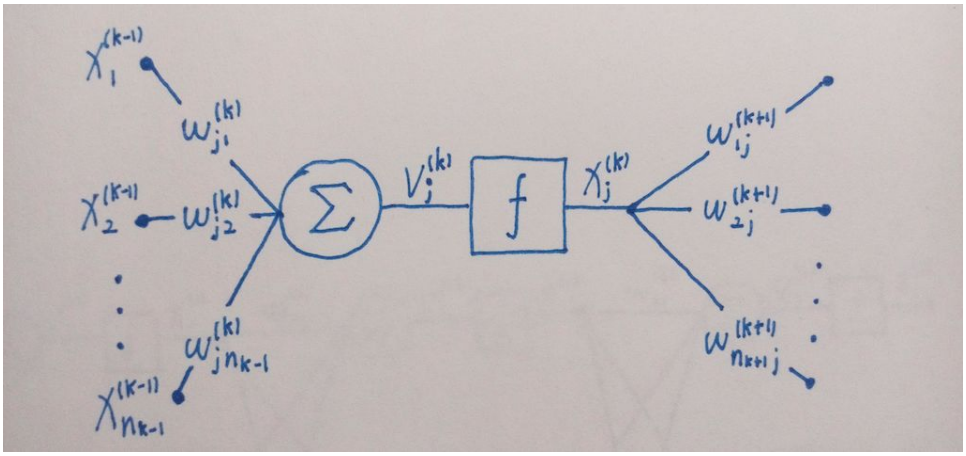


图 1.2 神经元

圆圈 Σ 是线性加和单元。它连接到第 k-1 层的 n_{k-1} 个神经元的输出 $x_i^{(k-1)}\ (1\le i\le n_{k-1})$。$w_{ji}^{(k)}$ 是连接的权重。线性加和单元的计算结果 $v_j^{(k)}$ 称作该神经元的“激活水平”，由 [1.1] 计算得到。

$$v_j^{(k)}=\sum_{i=1}^{n_{k-1}}w_{ji}^{(k)}x_i^{(k-1)}=\left(w_{j1}^{(k)},w_{j2}^{(k)},\ldots,w_{jn_{k-1}}^{(k)}\right)\begin{pmatrix}x_1^{(k-1)}\\x_2^{(k-1)}\\\vdots\\x_{n_{k-1}}^{(k-1)}\end{pmatrix}\quad[1.1]$$

从 [1.1] 可看出，神经元的“激活水平”是其权值向量与输入向量的内积。f 是神经元的激活函数。激活函数的输入是激活水平，输出是神经元的输出：

$$x_j^{(k)}=f(v_j^{(k)})\quad[1.2]$$

$x_j^{(k)}$ 提供给下一层（*第 k+1 层*）的 n_{k+1} 个神经元作为输入之一。例如 $x_j^{(k)}$ 乘上权重 $w_{qj}^{(k+1)}$ 送给第 k+1 层第 q 个神经元的线性加和单元。

神经网络的计算过程就是将输入向量提供给网络第 1 层各神经元，经过加权求和得到激活水平，之后对激活水平施加激活函数得到结果。将这些结果输送给下一层神经元。依此类推，直到最后一层（*输出层*）计算出结果，就是神经网络的输出向量。

二、训练过程

训练集中的样本形如：$[\mathbf{x}, \bar{\mathbf{y}}] = \left[(x_1, x_2, \ldots, x_{n_0})^T, (\bar{y}_1, \bar{y}_2, \ldots, \bar{y}_{n_K})^T\right]$。输入包含 n_0 个值，目标值包含 n_K 个值，分别对应神经网络的输入 / 输出维度。训练这样进行：将训练集中的样本一个接一个提交给神经网络。神经网络对样本输入 $\mathbf{x}$ 计算输出 $\mathbf{y}$，然后计算样本目标值与输出的平方和误差：

$$E = \frac{1}{2}\sum_{i=1}^{n_K}(\bar{y}_i - y_i)^2 = \frac{1}{2}(\bar{\mathbf{y}} - \mathbf{y})^T(\bar{\mathbf{y}} - \mathbf{y}) \quad [2.1]$$

视输入 $\mathbf{x}$ 为固定值，把 E 当作全体权值 $\mathbf{W} = \left\{w_{ji}^{(k)}\right\}$ 的函数。求 E 的梯度 ∇E，然后用下式更新全体权值：

$$\mathbf{W}(s+1) = \mathbf{W}(s) - \eta\nabla E \quad [2.2]$$

[2.2] 是梯度下降法的更新式。其中 η 是步长，s 是迭代次数。梯度矩阵 ∇E 由 E 对每一个权重 $w_{ji}^{(k)}$ 的偏导数 $\frac{\partial E}{\partial w_{ji}^{(k)}}$ 构成。式 [2.2] 等价于对每一个权重进行更新：

$$w_{ji}^{(k)}(s+1) = w_{ji}^{(k)}(s) - \eta\frac{\partial E}{\partial w_{ji}^{(k)}} \quad [2.3]$$

对每一个提交给神经网络的样本用式 [2.3] 对全体权值进行一次更新，直到所有样本的误差值都小于一个预设的阈值，此时训练完成。看到这里或有疑问：不是应该用所有训练样本的误差的模平方的平均值（*均方误差*）来作 E 么？如果把样本误差的模平方看作一个随机变量，那么所有样本的平均误差模平方是该随机变量的一个无偏估计。而一个样本的误差模平方也是该随机变量的无偏估计，只不过估计得比较粗糙（*大数定律*）。但是用一次一个样本的误差模平方进行训练可节省计算量，且支持在线学习（*样本随来随训练*）。

训练算法还可以有很多变体。例如动态步长、冲量等（*参考“上篇”*）。也可以将一批样本在同样的权值 $\mathbf{W}$ 下计算 ∇E，然后根据这一批 ∇E 的平均值更新 $\mathbf{W}$。这称为批量更新。训练的关键问题是如何计算 ∇E，即如何计算每一个 $\frac{\partial E}{\partial w_{ji}^{(k)}}$。

三、反向传播

回顾图 1.1 和

$$-\delta_j^{(k)} = \frac{\partial E}{\partial v_j^{(k)}} \quad [3.1]$$

将 $\delta_j^{(k)}$ 定义为 E 对第 k 层第 j 个神经元的激活水平 $v_j^{(k)}$ 的偏导数的相反数。根据求导链式法则，有：

$$\frac{\partial E}{\partial w_{ji}^{(k)}} = \frac{\partial E}{\partial v_j^{(k)}} \frac{\partial v_j^{(k)}}{\partial w_{ji}^{(k)}} \quad [3.2]$$

将 [3.2] 等号右侧的第二项展开：

$$\frac{\partial v_j^{(k)}}{\partial w_{ji}^{(k)}} = \frac{\partial}{\partial w_{ji}^{(k)}} \left(\sum_{s=1}^{n_{k-1}} w_{js}^{(k)} x_s^{(k-1)} \right) = x_i^{(k-1)} \quad [3.3]$$

结合定义 [3.1]，有：

$$\frac{\partial E}{\partial w_{ji}^{(k)}} = -\delta_j^{(k)} x_i^{(k-1)} \quad [3.4]$$

可见有了 $\delta_j^{(k)}$ 就能计算 E 对任一权重 $w_{ji}^{(k)}$ 的偏导数。接下来的问题就是如何计算 $\delta_j^{(k)}$ 。采用一种类似数学归纳法的方法。首先计算第 K 层（*输出层*）第 j 个神经元的 $\delta_j^{(K)}$ 。

$$-\delta_j^{(K)} = \frac{\partial E}{\partial v_j^{(K)}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial v_j^{(K)}} = \frac{\partial E}{\partial y_j} f'\left(v_j^{(K)}\right) = \frac{\partial}{\partial y_j} \left(\frac{1}{2} \sum_{i=1}^{n_K} (\bar{y}_i - y_i)^2 \right) f'\left(v_j^{(K)}\right) = -\left(\bar{y}_j - y_j\right) f'\left(v_j^{(K)}\right) \quad [3.5]$$

f' 表示 f 的导函数。[3.5] 展示了推导过程，其结论是：对于输出层（*第 K 层*）第 j 个神经元来说：

$$\delta_j^{(K)} = \left(\bar{y}_j - y_j\right) f'\left(v_j^{(K)}\right) \quad [3.6]$$

$\delta_j^{(K)}$ 等于目标值 $\bar{y}_j$ 与输出 y_j 之差乘上 f 在 $v_j^{(K)}$ 的导数。可以将 $\delta_j^{(K)}$ 看成一个经过缩放的误差。这个观点在后面讨论反向传播的意义时有用。

现在推导某个隐藏层——第 k 层第 j 个神经元的 $\delta_j^{(k)}$ $(k < K)$。将第 k+1 层的全体 $v_j^{(k+1)}$ 值视作一个向量：

$$\mathbf{v}^{(k+1)} = \left(v_1^{(k+1)}, v_2^{(k+1)}, \cdots, v_{n_{k+1}}^{(k+1)}\right)^T \quad [3.7]$$

再次回顾图 1.1 和图 1.2 。 $v_j^{(k)}$ 被施加激活函数 f 得到 $x_j^{(k)}$ 。 $x_j^{(k)}$ 乘上第 k+1 层各神经元对 $x_j^{(k)}$ 的各个权值，再与第 k 层其他神经元的输出加权求和，得到第 k+1 层各神经元的激活水平 $\mathbf{v}^{(k+1)}$ 。 $\mathbf{v}^{(k+1)}$ 再经过后面的网络得到网络输出， 最终计算出 E 。将整个过程视作一个三个函数 f, g , h 的复合。图 1.

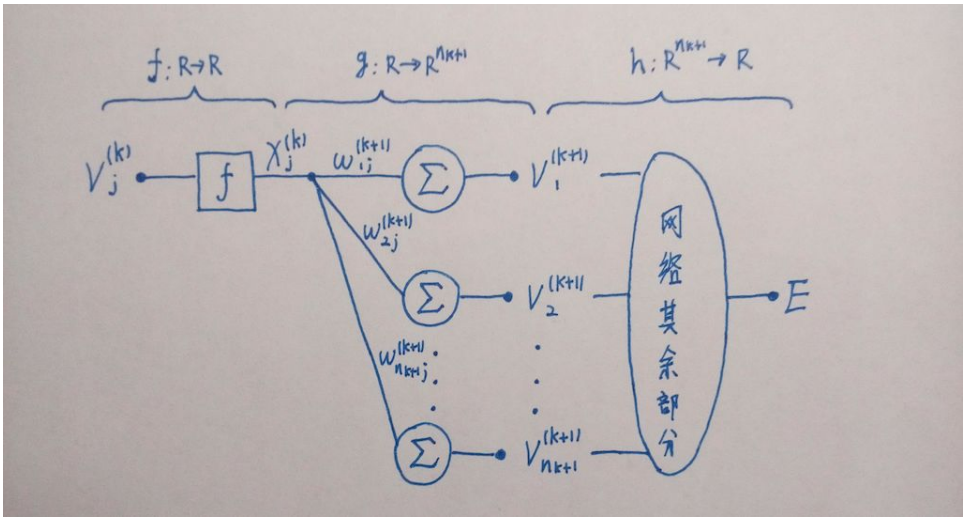


图 1.3 神经网络各个值影响损失函数的方式

连续使用链式法则，有：

$$-\delta_j^{(k)} = \frac{\partial E}{\partial v_j^{(k)}} = h' g' f' = \frac{\partial E}{\partial \mathbf{v}^{(k+1)}} \frac{\partial \mathbf{v}^{(k+1)}}{\partial x_j^{(k)}} \frac{\partial x_j^{(k)}}{\partial v_j^{(k)}} \quad [3.8]$$

等号右侧第一项是一个 $R^{n_{k+1}} \to R$ 函数的导数。它是 $1 \times n_{k+1}$ 元向量。它的第 i 个元素是：

$$\frac{\partial E}{\partial v_i^{(k+1)}} = -\delta_i^{(k+1)} \quad [3.9]$$

第二项是一个 $R \to R^{n_{k+1}}$ 函数的导数。它是 $n_{k+1} \times 1$ 元向量。它的第 i 个元素为：

$$\frac{\partial v_i^{(k+1)}}{\partial x_j^{(k)}} = \frac{\partial}{x_j^{(k)}} \left(\sum_{s=1}^{n_k} w_{is}^{(k+1)} x_s^{(k)} \right) = w_{ij}^{(k+1)} \quad [3.10]$$

最后一项是激活函数 f 在 $v_j^{(k)}$ 的偏导数。结合 [3.8]、[3.9] 和 [3.10] 得到：

$$-\delta_j^{(k)} = \frac{\partial E}{\partial v_j^{(k)}} = \left(-\delta_1^{(k+1)}, -\delta_2^{(k+1)}, \cdots, -\delta_{n_{k+1}}^{(k+1)} \right) \begin{pmatrix} w_{1j}^{(k+1)} \\ w_{2j}^{(k+1)} \\ \vdots \\ w_{n_{k+1}j}^{(k+1)} \end{pmatrix} f'(v_j^{(k)}) = -\left(\sum_{s=1}^{n_{k+1}} \delta_s^{(k+1)} w_{sj}^{(k+1)} \right) f'(v_j^{(k)}) \quad [3.11]$$

[3.11] 是推导过程，它的结论是：

$$\delta_j^{(k)} = \left(\sum_{s=1}^{n_{k+1}} \delta_s^{(k+1)} \right) f'(v_j^{(k)}) \quad [3.12]$$

注意 [3.8] 至 [3.10] 的推导过程运用了**多元函数**的求导链式法则。一个 $R^n \to R^m$ 函数的导数是一个 $m \times n$ 的矩阵。多元复合函数的求导链式法则是将导矩阵相乘。具体证明请可参考书目 [1] 附录部分或其他微积分教材。至此所有要素齐备。综合 [2.3]、[3.4]、[3.6] 和 [3.12] 可得反向传播算法如下：

- -

- ***反向传播阶段：***

$$\begin{cases} \delta_j^{(K)} = (\bar{y}_j - y_j) f'\left(v_j^{(K)}\right) \\ \delta_j^{(k)} = \left(\sum_{s=1}^{n_{k+1}} \delta_s^{(k+1)} w_{sj}^{(k+1)}\right) f'(v_j^{(k)}),\ k < K \end{cases} \quad [3.13\ a]$$

- ***权值更新阶段：***

$$w_{ji}^{(k)}(s+1) = w_{ji}^{(k)}(s) - \eta \frac{\partial E}{\partial w_{ji}^{(k)}} = w_{ji}^{(k)}(s) + \eta \delta_j^{(k)} x_i^{(k-1)} \quad [3.13\ b]$$

- -

可以用更紧凑的矩阵形式表示反向传播算法。由 [3.11] 可以得到：

$$\begin{aligned} \Delta^{(k)} &= \left(\delta_1^{(k)}, \delta_2^{(k)}, \cdots, \delta_{n_k}^{(k)}\right) = \left(-\frac{\partial E}{\partial v_1^{(k)}}, -\frac{\partial E}{\partial v_2^{(k)}}, \cdots, -\frac{\partial E}{\partial v_{n_k}^{(k)}}\right) \\ &= \left(\delta_1^{(k+1)}, \delta_2^{(k+1)}, \cdots, \delta_{n_{k+1}}^{(k+1)}\right) \begin{pmatrix} w_{11}^{(k+1)} & w_{12}^{(k+1)} & \cdots & w_{1n_k}^{(k+1)} \\ w_{21}^{(k+1)} & w_{22}^{(k+1)} & \cdots & w_{2n_k}^{(k+1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_{k+1}1}^{(k+1)} & w_{n_{k+1}2}^{(k+1)} & \cdots & w_{n_{k+1}n_k}^{(k+1)} \end{pmatrix} \begin{pmatrix} f'(v_1^{(k)}) & 0 & \cdots & 0 \\ 0 & f'(v_2^{(k)}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'(v_{n_k}^{(k)}) \end{pmatrix} \\ &= \Delta^{(k+1)} \mathbf{W}^{(k+1)} \mathbf{F}^{(k)} \end{aligned} \quad [3.14]$$

如 [3.14] 所示，第 k 层（ $k<K$ ）全体 $\delta_j^{(k)}$ 值组成的向量 $\Delta^{(k)}$ 可由本层的激活函数导数对角阵 $\mathbf{F}^{(k)}$ 、第 k+1 层的 $\Delta^{(k+1)}$ 和权值矩阵 $\mathbf{W}^{(k+1)}$ 计算得到。输出层（*第 K 层*）的 $\Delta^{(K)}$ 这么计算：

$$\Delta^{(K)} = \left(\bar{y}_1 - y_1, \bar{y}_2 - y_2, \cdots, \bar{y}_{n_K} - y_{n_K}\right) \begin{pmatrix} f'(v_1^{(K)}) & 0 & \cdots & 0 \\ 0 & f'(v_2^{(K)}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'(v_{n_K}^{(K)}) \end{pmatrix} = (\bar{\mathbf{y}} - \mathbf{y})^T \mathbf{F}^{(K)} \quad [3.15]$$

将矩阵形式的反向传播与权值更新算法总结如下：

- -

- ***反向传播阶***

$$\begin{cases} \Delta^{(K)} = (\bar{\mathbf{y}} - \mathbf{y})^T \mathbf{F}^{(K)} \\ \Delta^{(k)} = \Delta^{(k+1)} \mathbf{W}^{(k+1)} \mathbf{F}^{(k)}, \ k < K \end{cases} \quad [3.16\ a]$$

- ***权值更新阶段：***

$$\mathbf{W}^{(k)}(s+1) = \mathbf{W}^{(k)}(s) + \eta \left(\mathbf{x}^{(k-1)} \Delta^{(k)} \right)^T \quad [3.16\ b]$$

- -

[3.16 b] 中 $\mathbf{x}^{(k-1)}$ 是第 k-1 层输出向量。可以看到隐藏层 $\Delta^{(k)}$ 的计算利用了下一层的 $\Delta^{(k+1)}$ 。一个训练样本 $\mathbf{x}$ “正向”通过网络计算输出 $\mathbf{y}$ 。之后“反向”逐层计算 $\Delta^{(k)}$ 更新权值，并将 $\Delta^{(k)}$ 向前一层传播。所谓“反向”传播就是 $\Delta^{(k)}$ 的传播。以上推导没有包括神经元的偏置。把偏置看成一个连接到常量 1 的连接上的权值即可。

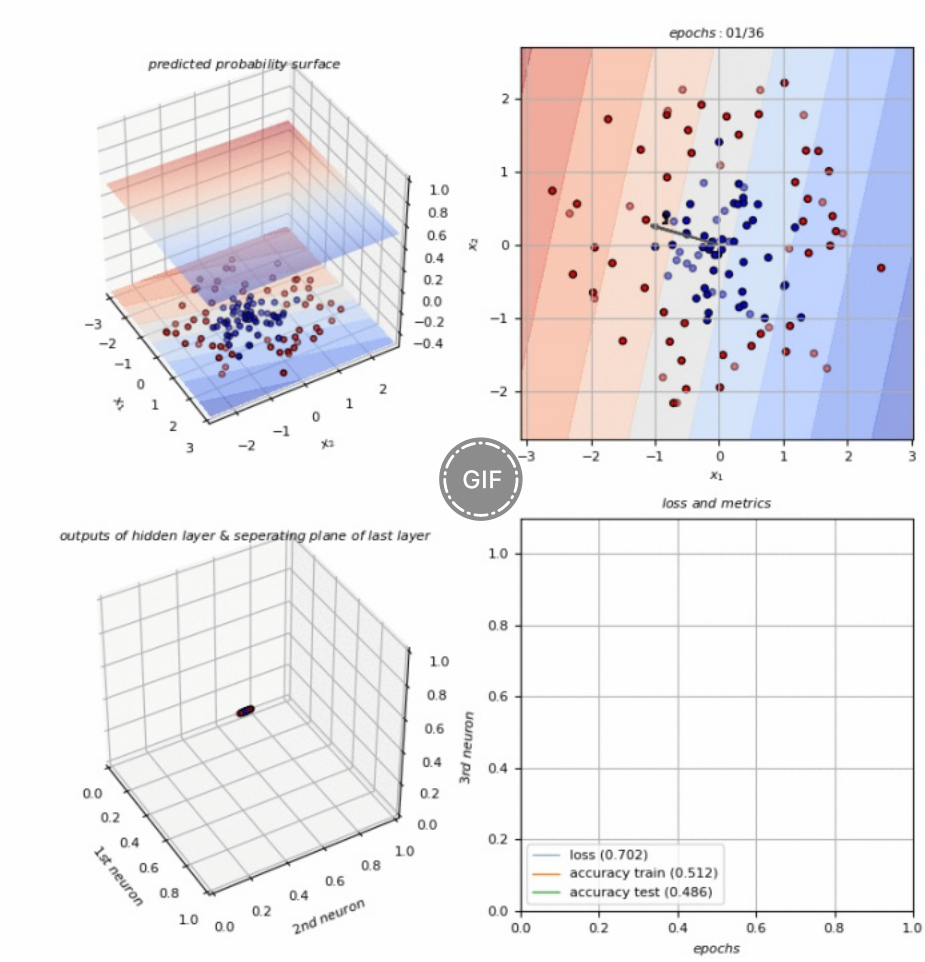
从计算式来看 $\delta_j^{(k)}$ 是 E 对第 k 层第 j 个神经元的激活水平 $v_j^{(k)}$ 的偏导数的相反数。还存在另一个视角。上文已经谈到，输出层的 $\delta_j^{(K)}$ 是经过缩放的误差。隐藏层的 $\delta_j^{(k)}$ 以连接权值加权组合了下一层各神经元的 $\delta_j^{(k+1)}$ 。可以把 $\delta_j^{(k)}$ 定义为某种“局部误差”。于是反向传播算法就是反向传播局部误差——把总误差分摊到各个神经元头上，让它们调整自己。

回顾一下图 1.3 。 第 k 层第 j 个神经元的激活水平 $v_j^{(k)}$ 通过 f 影响输出 $x_j^{(k)}$ 。输出 $x_j^{(k)}$ 通过加权求和影响第 k+1 层各神经元的激活水平。第 k+1 层的各激活水平通过网络其余部分最终影响误差 E 。于是第 k+1 层各神经元的局部误差按照权值分配到第 k 层第 j 个（以及其他）神经元的输出上。再经过 f 的导函数的调整，得到分配在第 k 层第 j 个神经元上的局部误差。反向传播本质是梯度下降，而局部误差视角为理解算法提供了一种洞见。

> 四、实现

插播广告。后来笔者写了一个机器学习库。那里的 ANN 实现对本文例程做了改进：

zhangjuefei/mentat
github.com



本节以 Python 语言实现了一个 mini-batch 随机梯度下降反向传播神经网络，带冲量和学习率衰减功能。代码如下：

```
import numpy as np


class DNN:
    def __init__(self, input_shape, shape, activations, eta=0.1, threshold=1e-5, softm
                 regularization=0.001, minibatch_size=5, momentum=0.9, decay_power=0.5

        if not len(shape) == len(activations):
            raise Exception("activations must equal to number od layers.")

        self.depth = len(shape)
        self.activity_levels = [np.mat([0])] * self.depth
        self.outputs = [np.mat(np.mat([0]))] * (self.depth + 1)
        self.deltas = [np.mat(np.mat([0]))] * self.depth
        self.eta = float(eta)
        self.effective_eta = self.eta
        self.threshold = float(threshold)
        self.max_epochs = int(max_epochs)
        self.regularization = float(regularization)
        self.is_softmax = bool(softmax)
        self.verbose = bool(verbose)
        self.minibatch_size = int(minibatch_size)
        self.momentum = float(momentum)
        self.decay_power = float(decay_power)
        self.iterations = 0
        sel
```

```
        self.activations = activations
        self.activation_func = []
        self.activation_func_diff = []
        for f in activations:

            if f == "sigmoid":
                self.activation_func.append(np.vectorize(self.sigmoid))
                self.activation_func_diff.append(np.vectorize(self.sigmoid_diff))
            elif f == "identity":
                self.activation_func.append(np.vectorize(self.identity))
                self.activation_func_diff.append(np.vectorize(self.identity_diff))
            elif f == "relu":
                self.activation_func.append(np.vectorize(self.relu))
                self.activation_func_diff.append(np.vectorize(self.relu_diff))
            else:
                raise Exception("activation function {:s}".format(f))

        self.weights = [np.mat(np.mat([0]))] * self.depth
        self.biases = [np.mat(np.mat([0]))] * self.depth
        self.acc_weights_delta = [np.mat(np.mat([0]))] * self.depth
        self.acc_biases_delta = [np.mat(np.mat([0]))] * self.depth

        self.weights[0] = np.mat(np.random.random((shape[0], input_shape)) / 100)
        self.biases[0] = np.mat(np.random.random((shape[0], 1)) / 100)
        for idx in np.arange(1, len(shape)):
            self.weights[idx] = np.mat(np.random.random((shape[idx], shape[idx - 1])))
            self.biases[idx] = np.mat(np.random.random((shape[idx], 1)) / 100)

    def compute(self, x):
        result = x
        for idx in np.arange(0, self.depth):
            self.outputs[idx] = result
            al = self.weights[idx] * result + self.biases[idx]
            self.activity_levels[idx] = al
            result = self.activation_func[idx](al)

        self.outputs[self.depth] = result
        return self.softmax(result) if self.is_softmax else result

    def predict(self, x):
        return self.compute(np.mat(x).T).T.A

    def bp(self, d):
        tmp = d.T

        for idx in np.arange(0, self.depth)[::-1]:
            delta = np.multiply(tmp, self.activation_func_diff[idx](self.activity_leve
            self.deltas[idx] = delta
            tmp = delta * self.weights[idx]

    def update(self):

        self.effective_eta = self.eta / np.power(self.iterations, self.decay_power)

        for idx in np.arange(0, self.depth):
            # current gradient
            weights_grad = -self.deltas[idx].T * self.outputs[idx].T / self.deltas[idx
                           self.regularization * self.weights[idx]
            biases_grad = -np.mean(self.deltas[idx].T, axis=1) + self.regularization *

            # accumulated delta
            self.acc_weights_delta[idx] = self.acc_weights_delta[
                                          idx] * self.momentum - self.effective_et
```

```
            self.weights[idx] = self.weights[idx] + self.acc_weights_delta[idx]
            self.biases[idx] = self.biases[idx] + self.acc_biases_delta[idx]

    def fit(self, x, y):
        x = np.mat(x)
        y = np.mat(y)
        loss = []
        self.iterations = 0
        self.epochs = 0
        start = 0
        train_set_size = x.shape[0]

        while True:

            end = start + self.minibatch_size
            minibatch_x = x[start:end].T
            minibatch_y = y[start:end].T

            yp = self.compute(minibatch_x)
            d = minibatch_y - yp

            if self.is_softmax:
                loss.append(np.mean(-np.sum(np.multiply(minibatch_y, np.log(yp + 1e-10
            else:
                loss.append(np.mean(np.sqrt(np.sum(np.power(d, 2), axis=0))))

            self.iterations += 1
            start = (start + self.minibatch_size) % train_set_size

            if self.iterations % train_set_size == 0:
                self.epochs += 1
                mean_e = np.mean(loss)
                loss = []

                if self.verbose:
                    print("epoch: {:d}. mean loss: {:.6f}. learning rate: {:.8f}".form


                if self.epochs >= self.max_epochs or mean_e < self.threshold:
                    break

            self.bp(d)
            self.update()

    @staticmethod
    def sigmoid(x):
        return 1.0 / (1.0 + np.power(np.e, min(-x, 1e2)))

    @staticmethod
    def sigmoid_diff(x):
        return np.power(np.e, min(-x, 1e2)) / (1.0 + np.power(np.e, min(-x, 1e2))) **

    @staticmethod
    def relu(x):
        return x if x > 0 else 0.0

    @staticmethod
    def relu_diff(x):
        return 1.0 if x > 0 else 0.0

    @staticmethod
    def identity(x):
        return x

    @static
```

```
    def identity_diff(x):
        return 1.0

    @staticmethod
    def softmax(x):
        x[x > 1e2] = 1e2
        ep = np.power(np.e, x)
        return ep / np.sum(ep, axis=0)
```

测试一下神经网络的拟合能力如何。用网络拟合以下三个函数：

$$f_1(x_1, x_2) = x_1^2 + x_2^2$$
$$f_2(x_1, x_2) = x_1^2 - x_2^2$$
$$f_3(x_1, x_2) = cos(1.2x_1)cos(1.2x_2) \quad [4.1]$$

对每个函数生成 100 个随机选择的数据点。神经网络的输入为 2 维，输出为 1 维。为每个函数训练 3 个神经网络。这些网络有 1 个隐藏层 1 个输出层，隐藏层神经元数量分别为：3、5 和 8 。隐藏层激活函数是 *sigmoid* 。输出层的激活函数是恒等函数 $f(x) = x$ 。初始学习率 0.4 ，衰减指数 0.2 。冲量惯性 0.6 。迭代 200 个 epoch 。mini batch 样本数为 40 。L2 正则，正则强度 0.0001 。拟合效果见下图：

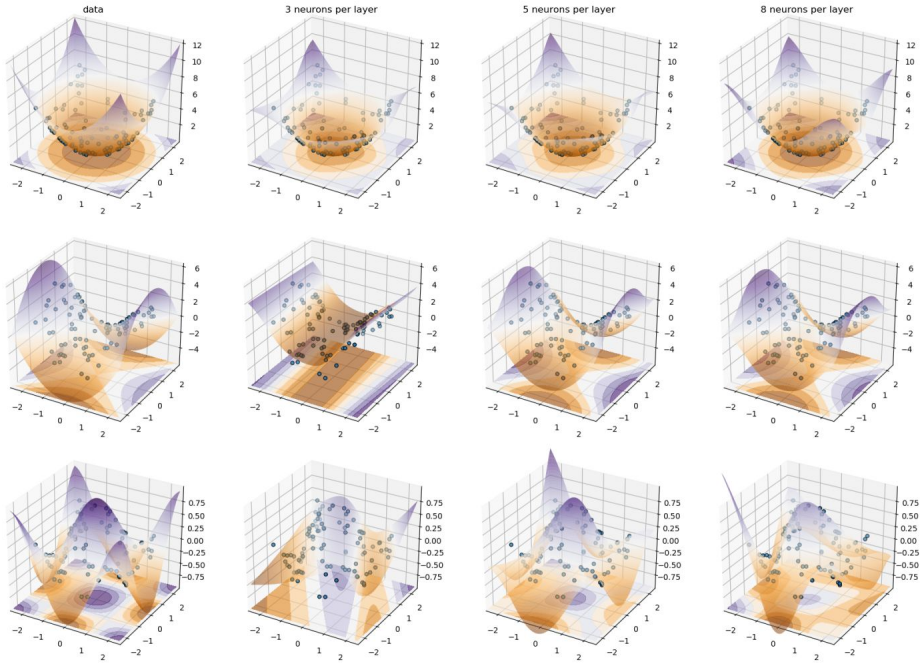


图 4.1 对三种函数进行拟合

测试代码如下：

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from mentat.classification_model import DNN
from mpl_toolkits.mplot3d import Axes3D

np.random.seed(42)

hidden_laye
```

```
hidden_layers = 1  # 隐藏层数量
hidden_layer_activation_func = "sigmoid"  # 隐藏层激活函数
learning_rate = 0.4  # 学习率
max_epochs = 200  # 训练 epoch 数量
regularization_strength = 0.0001  # 正则化强度
minibatch_size = 40  # mini batch 样本数
momentum = 0.6  # 冲量惯性
decay_power = 0.2  # 学习率衰减指数


def f1(x):
    return (x[:, 0] ** 2 + x[:, 1] ** 2).reshape((len(x), 1))


def f2(x):
    return (x[:, 0] ** 2 - x[:, 1] ** 2).reshape((len(x), 1))


def f3(x):
    return (np.cos(1.2 * x[:, 0]) * np.cos(1.2 * x[:, 1])).reshape((len(x), 1))


funcs = [f1, f2, f3]

X = np.random.uniform(low=-2.0, high=2.0, size=(100, 2))
x_min, x_max = X[:, 0].min() - .5, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - .5, X[:, 1].max() + .5
xx, yy = np.meshgrid(np.arange(x_min, x_max, .02), np.arange(y_min, y_max, .02))

# 模型
names = ["{:d} neurons per layer".format(hs) for hs
         in hidden_layer_size]

classifiers = [
    DNN(input_shape=2, shape=[hs] * hidden_layers + [1],
        activations=[hidden_layer_activation_func] * hidden_layers + ["identity"], eta
        softmax=False, max_epochs=max_epochs, regularization=regularization_strength,
        minibatch_size=minibatch_size, momentum=momentum, decay_power=decay_power) for
    hidden_layer_size
]

figure = plt.figure(figsize=(5 * len(classifiers) + 2, 4 * len(funcs)))
cm = plt.cm.PuOr
cm_bright = ListedColormap(["#DB9019", "#00343F"])
i = 1

for cnt, f in enumerate(funcs):

    zz = f(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)
    z = f(X)

    ax = figure.add_subplot(len(funcs), len(classifiers) + 1, i, projection="3d")

    if cnt == 0:
        ax.set_title("data")

    ax.plot_surface(xx, yy, zz, rstride=1, cstride=1, alpha=0.6, cmap=cm)
    ax.contourf(xx, yy, zz, zdir='z', offset=zz.min(), alpha=0.6, cmap=cm)
    ax.scatter(X[:, 0], X[:, 1], z.ravel(), cmap=cm_bright, edgecolors='k')
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_zlim(zz.min(), zz.max())

    i += 1
```

```
    for name, clf in zip(names, classifiers):

        print("model: {:s} training.".format(name))

        ax = plt.subplot(len(funcs), len(classifiers) + 1, i)
        clf.fit(X, z)
        predict = clf.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)

        ax = figure.add_subplot(len(funcs), len(classifiers) + 1, i, projection="3d")

        if cnt == 0:
            ax.set_title(name)

        ax.plot_surface(xx, yy, predict, rstride=1, cstride=1, alpha=0.6, cmap=cm)
        ax.contourf(xx, yy, predict, zdir='z', offset=zz.min(), alpha=0.6, cmap=cm)
        ax.scatter(X[:, 0], X[:, 1], z.ravel(), cmap=cm_bright, edgecolors='k')
        ax.set_xlim(xx.min(), xx.max())
        ax.set_ylim(yy.min(), yy.max())
        ax.set_zlim(zz.min(), zz.max())

        i += 1
        print("model: {:s} train finished.".format(name))

plt.tight_layout()
plt.savefig(
    "pic/dnn_fitting_{:d}_{:.6f}_{:d}_{:.6f}_{:.3f}_{:3f}.png".format(hidden_layers, l
                                                                      regularization_s
```

五、参考书目

最优化导论 (豆瓣)
book.douban.com

神经网络设计 (豆瓣)
book.douban.com

机器学习 (豆瓣)
book.douban.com

神经计算原理 (豆瓣)
book.douban.com

编辑于 2019-12-09

「真诚赞赏，手留余香」

赞赏

2 人已赞赏

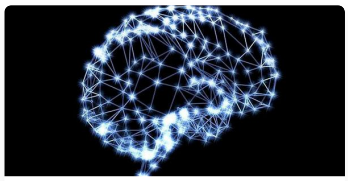
深度学习（Deep Learning） 神经网络 人工智能

文章被以下专栏收录

计算主义
“我不能建造者，我则没有真正理解”（费恩曼）

关注专栏

推荐阅读



关于神经网络：你需要知道这些

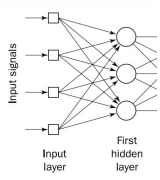
机器之心 发表于机器之心

[翻译] 神经网络与深度学习 第一章 利用神经网络识别手写...

Xovee

全连接神经网络中反向传播算法数学推导

南柯一梦宁沉沦



神经网络浅讲：学习

JimmyChen

23 条评论 ⇌ 切换为时间排序

写下你的评论...

精选评论（3）

张觉非 (作者) 回复 蔡超前 2017-03-09
是么，看的书还不够多啊。我改一下原文。
赞 查看回复

张觉非 (作者) 回复 walkeryoung 2017-03-10
这俩词在这儿是个人这么用，私货，没有遵照这俩词公认的定义。代码里实现的不是批更新，mse 也不是真正的 mse ，为实现简单。您是指这个问题还是还有其他问题？请指教：）
赞 查看回复

张觉非 (作者) 回复 walkeryoung 2017-03-10
恩，加了个说明。
赞 查看回复

评论（23）

蔡超前 2017-03-09
delta_k与delta_k+1的递推关系其实在Sergios的Pattern Recognition的4.6章节已经有详细推导。
赞

张觉非 (作者) 回复 蔡超前 2017-03-09

是么，看的书还不够多啊。我改一下原文。

赞

OMNI 2017-03-09

cover 好评，小波导引封面

赞

罗俊华 2017-06-21

E对V（k+1）的偏导为什么是一个行向量呀?

赞

张觉非 (作者) 回复 罗俊华 2018-01-02

一个 n->1 的函数，它的线性化是一个 n->1 的线性变换。用矩阵乘法表示线性变换，那就是用一个 1 x n 的行向量去乘一个 n x 1 的列向量，得到一个标量。所以其实 n->1 函数的导数是 1 x n 矩阵，或者说 n 元的行向量。我们平时说 n x 1 的梯度向量其实是导数的转置。

1

巨人脚下的蚂蚁 回复 罗俊华 06-08

其实博主在这里，，，，怎么说呢....我也要去查一下一些概念才行，但是你的问题其实很好解答，你去百度一下标量对向量求导就可以达到答案了

1

Cerulean 2018-04-27

请问这个例程无法在iris数据集上无法收敛，可能原因是什么呢?

赞

张觉非 (作者) 回复 Cerulean 2018-04-27

把 feature 标准化一下试试？再改改超参什么的。也不太敢说例程没 bug 。有一个改进的实现（速度更快，测试更多）在这里，您如果有兴趣可以看看 github.com/zhangjuefei/...

赞

Cerulean 回复 张觉非 (作者) 2018-04-27

谢谢，是我的参数初始化有问题，已经可以收敛啦，谢谢。

赞

塞巴斯万隆 2018-08-15

数学推导过程看得很清晰舒服，谢谢您的分享

赞

ChenJ 2019-07-22

写的很详细 图话的尤其好看 治愈强迫症的一切毛病😂

赞

张觉非 (作者) 回复 ChenJ 2019-07-22

在即将出版的《深入理解神经网络》中，图画得更好看，敬请期待[大笑]

赞

ChenJ 回复 张觉非 (作者) 2019-07-22

给力！是个狼人！！！

赞同 149 23 条评论 分享 收藏 ···

ChenJ 2019-07-22
写的很详细 图话的尤其好看 治愈强迫症的一切毛病😂
赞

高飞 2019-12-22
好评
赞

asd123www 06-02
收获很大，谢谢分享
赞

巨人脚下的蚂蚁 06-07
该评论已删除

张觉非 (作者) 回复 巨人脚下的蚂蚁 06-07
这里是将 W 固定，认为随机性来源于随机的训练样本。
赞

巨人脚下的蚂蚁 06-11
最好说明一下矩阵求导是分母布局
赞

3 条评论被折叠（为什么？）