第 6 章补充材料

● BP 算法的推导

多层感知器学习的目的是要调整网络中每个神经元的权值矢量和偏置，使得对于训练样本集合 D 的识别误差最小。同线性的两层感知网络一样，多层感知器的权值学习也要转化为对误差平方和的优化问题求解。

令训练样本集包含 n 个样本 $D=\{\mathbf{x}_1,\cdots,\mathbf{x}_n\}$，根据每个样本的类别属性设定相应的期望输出矢量 $\{\mathbf{t}_1,\cdots,\mathbf{t}_n\}$。如果将网络所有神经元的权值和偏置表示为一个统一的参数矢量 $\mathbf{w}$，对应所有训练样本的网络实际输出为 $\{\mathbf{z}_1,\cdots,\mathbf{z}_n\}$，需要优化的平方误差函数为：

$$\min_{\mathbf{w}} J(\mathbf{w})=\sum_{i=1}^{n}\mathcal{E}_i(\mathbf{w})=\frac{1}{2}\sum_{i=1}^{n}\|\mathbf{t}_i-\mathbf{z}_i\|^2 \qquad (1)$$

在线性网络的学习中，我们优化的是同样的平方误差函数。由于线性网络的实际输出 $\mathbf{z}_i$ 可以很容易地表示为网络参数 $\mathbf{w}$ 的简洁表达式，通过微分运算就能够得到优化函数的极值点；而在多层感知器网络中，由于隐含层神经元的存在，实际输出 $\mathbf{z}_i$ 与参数矢量 $\mathbf{w}$ 之间的关系比较复杂，很难由简单的微分求解（1）式优化问题的最小值点，需要采用梯度法进行优化，由初始的参数矢量 $\mathbf{w}(0)$ 开始，迭代优化直到收敛：

$$\mathbf{w}(m)=\mathbf{w}(m-1)-\eta\left.\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}\right|_{\mathbf{w}=\mathbf{w}(m-1)} \qquad (2)$$

考虑（1）式，由于：

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}=\sum_{i=1}^{n}\frac{\partial \mathcal{E}_i(\mathbf{w})}{\partial \mathbf{w}}$$

因此算法的关键是要计算输入训练样本 $\mathbf{x}_i$ 时的平方误差 $\mathcal{E}_i(\mathbf{w})=\frac{1}{2}\|\mathbf{t}_i-\mathbf{z}_i\|^2$ 关于每个网络参数的偏导数。下面我们以包含一个隐含层的三层感知器网络为例，推导各个网络参数的迭代公式，包含多个隐含层的情况可以依此类推。

I. 输出层

首先，考虑包含 n_H 个神经元的隐含层第 j 个神经元与包含 L 个神经元的输出层第 k 个神经元之间的连接权值 w_{kj} 以及偏置 b_k（图 1）。令隐含层神经元的输出为 $\{y_1,\cdots,y_{n_H}\}$，输出层神经元的净输入为 $\{net_1,\cdots,net_L\}$，输出层和隐含层神经元的激活函数分别为 $f_o(u)$ 和 $f_h(u)$，应用隐函数求导法则有（$\mathcal{E}$ 为输入 $\mathbf{x}_i$ 的平方误差，为简洁起见以下符号均省略下标 i）：

$$\frac{\partial \mathcal{E}}{\partial w_{kj}}=\frac{\partial \mathcal{E}}{\partial z_k}\frac{\partial z_k}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}}$$

$$\frac{\partial \mathcal{E}}{\partial b_k}=\frac{\partial \mathcal{E}}{\partial z_k}\frac{\partial z_k}{\partial net_k}\frac{\partial net_k}{\partial b_k} \tag{3}$$

由于：

$$\mathcal{E}=\frac{1}{2}\|\mathbf{t}-\mathbf{z}\|^2=\frac{1}{2}\sum_{l=1}^{L}(t_l-z_l)^2 \qquad \frac{\partial \mathcal{E}}{\partial z_k}=-(t_k-z_k)$$

$$z_k=f_o(net_k) \qquad \frac{\partial z_k}{\partial net_k}=f_o'(net_k)$$

$$net_k=\sum_{l=1}^{n_H}w_{kl}y_l+b_k \qquad \frac{\partial net_k}{\partial w_{kj}}=y_j,\frac{\partial net_k}{\partial b_k}=1$$

代入（3）式，有：

$$\frac{\partial \mathcal{E}}{\partial w_{kj}}=-(t_k-z_k)f_o'(net_k)y_j$$

$$\frac{\partial \mathcal{E}}{\partial b_k}=-(t_k-z_k)f_o'(net_k) \tag{4}$$

定义 $\delta_k=(t_k-z_k)f_o'(net_k)$，则可以得到平方误差关于输出层神经元参数 w_{kj} 和 b_k 的梯度：

$$\frac{\partial \mathcal{E}}{\partial w_{kj}}=-\delta_k y_j,\quad \frac{\partial \mathcal{E}}{\partial b_k}=-\delta_k \tag{5}$$

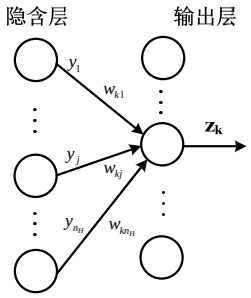

图 1 输出层神经元权值的学习

II. 隐含层

现在考虑隐含层第 j 个节点与输入层第 m 个节点之间的连接权值 w_{jm} 以及偏置 b_j（图 2）。与输出层类似，应用隐函数求导规则：

$$\frac{\partial \mathcal{E}}{\partial w_{jm}}=\frac{\partial \mathcal{E}}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{jm}}$$

$$\frac{\partial \mathcal{E}}{\partial b_j}=\frac{\partial \mathcal{E}}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial b_j} \tag{6}$$

其中：

$$y_j=f_h(net_j) \qquad \frac{\partial y_j}{\partial net_j}=f_h'(net_j)$$

$$net_j=\sum_{k=1}^{d}w_{jk}x_k+b_j \qquad \frac{\partial net_j}{\partial w_{jm}}=x_m,\frac{\partial net_j}{\partial b_j}=1 \tag{7}$$

$\mathcal{E}$ 不能直接表示成 y_j 的函数，但可以表示成输出层 $\{z_1,\cdots,z_L\}$ 的函数，而每一个输出 z_k 都是 y_j 的函数，$\partial \mathcal{E}/\partial y_j$ 的计算相对于输出层要复杂一些：

$$\mathcal{E}=\frac{1}{2}\sum_{k=1}^{L}\left(t_k-z_k\right)^2,\quad z_k=f_o\left(net_k\right),\quad net_k=\sum_{l=1}^{n_H}w_{kl}y_l+b_k$$

$$\begin{aligned}\frac{\partial\mathcal{E}}{\partial y_j}&=-\sum_{k=1}^{L}\left(t_k-z_k\right)\frac{\partial z_k}{\partial y_j}\\&=-\sum_{k=1}^{L}\left(t_k-z_k\right)\frac{\partial z_k}{\partial net_k}\frac{\partial net_k}{\partial y_j}\\&=-\sum_{k=1}^{L}\left(t_k-z_k\right)f_o'\left(net_k\right)w_{kj}=-\sum_{k=1}^{L}\delta_k w_{kj}\end{aligned}\tag{8}$$

将（7）和（8）代入（6）：

$$\frac{\partial\mathcal{E}}{\partial w_{jm}}=-\left[\sum_{k=1}^{L}\delta_k w_{kj}\right]f_h'\left(net_j\right)x_m$$

$$\frac{\partial\mathcal{E}}{\partial b_j}=-\left[\sum_{k=1}^{L}\delta_k w_{kj}\right]f_h'\left(net_j\right)$$

定义$\delta_j=f_h'\left(net_j\right)\sum_{k=1}^{L}\delta_k w_{kj}$，则可以得到平方误差关于隐含层神经元参数$w_{jm}$和$b_j$的梯度：

$$\frac{\partial\mathcal{E}}{\partial w_{jm}}=-\delta_j x_m,\quad\frac{\partial\mathcal{E}}{\partial b_j}=-\delta_j\tag{9}$$

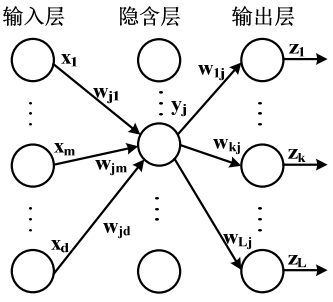

图 2 隐含层神经元权值的学习

由公式（5）和（9），我们得到了平方误差函数$\mathcal{E}$关于输出层和隐含层参数的梯度。注意到输出层需要计算的主要是每个节点的$\delta_k=\left(t_k-z_k\right)f_o'\left(net_k\right)$，某种程度上这可以看作是输出节点$k$上的误差；而隐含层每个节点计算$\delta_j=f_h'\left(net_j\right)\sum_{k=1}^{L}\delta_k w_{kj}$时需要用到所有输出层节点的误差$\delta_k$，这也可以看作是隐含层节点$j$的误差。由于隐含层节点的误差需要由输出层节点的误差反向计算得到，因此多层感知器网络参数的学习算法也被称为误差反向传播算法（Backpropagation Alogrithm，BP 算法）。

● Levenberg-Marquardt 算法（LM 算法）

LM 算法是一种专门适合于平方误差优化函数的方法。首先，我们需要将全部 n 个训练样本在 L 个网络输出神经元上的期望输出和实际输出写成矢量形式：

$$\mathbf{t}=\left(t_{11},\cdots,t_{1L},t_{21},\cdots,t_{nL}\right)^t,\quad \mathbf{z}=\left(z_{11},\cdots,z_{1L},z_{21},\cdots,z_{nL}\right)^t$$

其中 t_{ij} 和 z_{ij} 分别为第 i 个训练样本在第 j 个输出神经元上的期望输出和实际输出。定义 nL 维矢量：$\mathbf{v}(\mathbf{w})=\mathbf{t}-\mathbf{z}$，对比公式（1）可以看出，平方误差优化函数的优化问题可以表示为：

$$\min_{\mathbf{w}} J(\mathbf{w})=\frac{1}{2}\mathbf{v}^t(\mathbf{w})\mathbf{v}(\mathbf{w})$$

优化函数的梯度：

$$\nabla J(\mathbf{w})=\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}}=\left(\frac{\partial \mathbf{v}(\mathbf{w})}{\partial \mathbf{w}}\right)^t\mathbf{v}(\mathbf{w})=\tilde{\mathbf{J}}^t(\mathbf{w})\mathbf{v}(\mathbf{w}) \qquad (10)$$

其中 $\tilde{\mathbf{J}}(\mathbf{w})$ 为矢量 $\mathbf{v}$ 关于 $\mathbf{w}$ 的 Jacobian 矩阵，N 为网络权值数：

$$\tilde{\mathbf{J}}(\mathbf{w})=\frac{\partial \mathbf{v}(\mathbf{w})}{\partial \mathbf{w}}=\begin{bmatrix}\frac{\partial v_1(\mathbf{w})}{\partial w_1} & \frac{\partial v_1(\mathbf{w})}{\partial w_2} & \cdots & \frac{\partial v_1(\mathbf{w})}{\partial w_N}\\ \frac{\partial v_2(\mathbf{w})}{\partial w_1} & \frac{\partial v_2(\mathbf{w})}{\partial w_2} & \cdots & \frac{\partial v_2(\mathbf{w})}{\partial w_N}\\ \vdots & \vdots & \ddots & \vdots\\ \frac{\partial v_{nL}(\mathbf{w})}{\partial w_1} & \frac{\partial v_{nL}(\mathbf{w})}{\partial w_2} & \cdots & \frac{\partial v_{nL}(\mathbf{w})}{\partial w_N}\end{bmatrix}$$

进一步计算优化函数 $J(\mathbf{w})$ 的 Hessian 矩阵：

$$\mathbf{H}=\frac{\partial\left(\nabla J(\mathbf{w})\right)}{\partial \mathbf{w}}=\tilde{\mathbf{J}}^t(\mathbf{w})\frac{\partial \mathbf{v}(\mathbf{w})}{\partial \mathbf{w}}+\mathbf{S}(\mathbf{w})=\tilde{\mathbf{J}}^t(\mathbf{w})\tilde{\mathbf{J}}(\mathbf{w})+\mathbf{S}(\mathbf{w})$$

其中 $\mathbf{S}(\mathbf{w})$ 由 Jacobian 矩阵关于 $\mathbf{w}$ 的微分和 $\mathbf{v}(\mathbf{w})$ 计算，如果假设 $\mathbf{S}(\mathbf{w})$ 很小，则可以得到 Hessian 矩阵的近似：

$$\mathbf{H}\approx\tilde{\mathbf{J}}^t(\mathbf{w})\tilde{\mathbf{J}}(\mathbf{w}) \qquad (11)$$

由此可以看出，Hessian 矩阵可以由 Jacobian 矩阵近似计算。将（10）和（11）式带入到牛顿法的权值增量计算公式：

$$\Delta\mathbf{w}=-\mathbf{H}^{-1}\left(\frac{\partial J}{\partial \mathbf{w}}\right)=-\left(\tilde{\mathbf{J}}^t(\mathbf{w})\tilde{\mathbf{J}}(\mathbf{w})\right)^{-1}\tilde{\mathbf{J}}^t(\mathbf{w})\mathbf{v}(\mathbf{w}) \qquad (12)$$

按照这种方式计算的优点是无需计算二阶导数矩阵 $\mathbf{H}$，但可以近似得到二阶优化的效果，这种方法一般称为高斯-牛顿法。

在实际应用中矩阵 $\tilde{\mathbf{J}}^t(\mathbf{w})\tilde{\mathbf{J}}(\mathbf{w})$ 可能是不可逆的，所以一般采用如下方式计算权值矢量的增量：

$$\Delta\mathbf{w}=-\left(\tilde{\mathbf{J}}^t(\mathbf{w})\tilde{\mathbf{J}}(\mathbf{w})+\mu\mathbf{I}\right)^{-1}\tilde{\mathbf{J}}^t(\mathbf{w})\mathbf{v}(\mathbf{w})$$

其中 $\mathbf{I}$ 是单位矩阵，$\mu>0$。算法的每一轮迭代中尝试不同 μ，在保证矩阵可逆的条件下使得 μ 尽量小，这种方法一般称为 Levenberg-Marquardt 算法。

Jacobian 矩阵 $\tilde{\mathbf{J}}(\mathbf{w})$ 的维数为 $nL\times N$，当样本很多时维数会很大。但在算法实现时可以直接计算 $N\times N$ 维矩阵 $\tilde{\mathbf{J}}^{t}(\mathbf{w})\tilde{\mathbf{J}}(\mathbf{w})$ 和 N 维矢量 $\tilde{\mathbf{J}}^{t}(\mathbf{w})\mathbf{v}(\mathbf{w})$，因此存储复杂度只是 $O(N^2)$。

Jacobian 矩阵中每个元素的计算类似于 BP 算法对 $\mathbf{w}$ 中每个元素导数的计算，也需要一个由输出层到各个隐含层的反馈过程，由于只是一次项，因此还要简单一些。

LM 算法避免了直接计算 Hessian 矩阵，但仍然需要计算 $N\times N$ 维矩阵的逆阵，每一轮迭代的计算复杂度比 BP 算法要高得多。但由于近似实现了二阶优化技术，迭代次数会远远少于 BP 算法，所以学习过程的收敛速度一般明显快于梯度下降的 BP 算法。LM 算法的存储复杂度和计算复杂度都与 N 相关，因此一般适用于网络规模不大，参数数量适中的问题。