第3章 BP网络算法分析与工程应用

3.1 数值优化的BP网络训练算法

基于一阶梯度的方法用于简单问题时往往可以很快地收敛到期望值，然而，当用于较复杂的实际问题时，除了弹性BP算法以外，其余算法在收敛速度上都存在着一定的问题。

BP网络的训练实质是一个非线性目标函数的优化问题，人们对非线性优化问题的研究已有数百年的历史，而且不少传统数值优化的方法收敛也较快，因而人们自然想到采用数值优化方法的算法对BP网络的权值进行训练。与梯度下降法不同，基于数值优化的算法除了利用了目标函数的一阶导数信息，往往还利用目标函数的二阶导数信息。这类算法包括拟牛顿法、Levenberg-Marquardt法和共轭梯度法，它们可统一描述为

$$f(\boldsymbol{X}^{(k+1)}) = \min f(\boldsymbol{X}^{(k)} + \eta^{(k)} S(\boldsymbol{X}^{(k)}))$$

$$\boldsymbol{X}^{(k+1)} = \boldsymbol{X}^{(k)^{\eta}} + \eta^{(k)} S(\boldsymbol{X}^{(k)})$$

其中，$\boldsymbol{X}^{(k)}$为网络所有的权值和偏置值组成的向量；$S(\boldsymbol{X}^{(k)})$为由X的各分量组成的向量空间中的搜索方向；$\eta^{(k)}$为在$S(\boldsymbol{X}^{(k)})$的方向上，使$f(\boldsymbol{X}^{(k+1)})$达到极小的步长。这样，网络权值的寻优分为两步：首先确定当前迭代的最佳搜索方向$S(\boldsymbol{X}^{(k)})$，而后在此方向上寻求最优迭代步长。关于最优搜索步长$\eta^{(k)}$的选取，是一个一维搜索(线搜索)问题。对于这一问题有许多方法可供选择，如黄金分割法、二分法、多项式插值等。以下所讨论的三种方法正是在最佳搜索方向$S(\boldsymbol{X}^{(k)})$的选择上有所不同。

3.1.1 拟牛顿法

牛顿法是一种常见的快速优化方法，它利用了一阶和二阶导数信息，其基本形式为：第一次迭代的搜索方向确定为负梯度方向，即搜索方向$S(\boldsymbol{X}^{(0)}) = -\nabla f(\boldsymbol{X}^{(0)})$，以后各次迭代的搜索方向由下式确定

$$S(X^{(k)}) = -(\boldsymbol{H}^{(k)})^{-1}\nabla f(\boldsymbol{X}^{(k)}) \tag{3-1}$$

即

$$\boldsymbol{X}^{(k+1)} = \boldsymbol{X}^{(k)} - \eta^{(k)}S(\boldsymbol{X}^{(k)}) = \boldsymbol{X}^{(k)} - \eta^{(k)}(\boldsymbol{H}^{(k)})^{-1}\nabla f(\boldsymbol{X}^{(k)}) \tag{3-2}$$

式中，$\boldsymbol{H}^{(k)}$为海森(Hessian)矩阵(二阶导数矩阵)。牛顿法的收敛速率比一阶梯度法快，不过由于神经网络中参数数目庞大，导致计算海森矩阵的复杂性增加。

因此，人们在牛顿法的基础上，提出了一类无需计算二阶导数矩阵及其求逆运算的方法。这类方法一般是利用梯度信息或一个近似矩阵去逼近 $\boldsymbol{H}^{(k)}$，不同的构造 $\boldsymbol{H}^{(k)}$的方法就产生了不同的拟牛顿法。显然，拟牛顿法是为了克服梯度下降法收敛慢以及牛顿法计算复杂而提出的一种算法。BFGS 拟牛顿法及正割拟牛顿法是两种典型的拟牛顿法。

1. BFGS 拟牛顿法

除了第一次迭代外，对应(3-1)、(3-2)两式，BFGS 拟牛顿法在每一次迭代中采用下式来逼近海森矩阵

$$\boldsymbol{H}^{(k)} = \boldsymbol{H}^{(k-1)} + \frac{\nabla f(\boldsymbol{X}^{(k-1)}) * \nabla f(\boldsymbol{X}^{(k-1)})^{\mathrm{T}}}{\nabla f(\boldsymbol{X}^{(k-1)})^{\mathrm{T}} * S(\boldsymbol{X}^{(k-1)})} + \frac{\mathrm{d}g\boldsymbol{X} * \mathrm{d}g\boldsymbol{X}^{\mathrm{T}}}{\mathrm{d}g\boldsymbol{X}^{\mathrm{T}} * \eta^{(k-1)}S(\boldsymbol{X}^{(k-1)})} \tag{3-3}$$

式中，$\mathrm{d}g\boldsymbol{X} = \nabla f(\boldsymbol{X}^{(k)}) - \nabla f(\boldsymbol{X}^{(k-1)})$。

BFGS 拟牛顿法在每次迭代中都要存储近似的海森矩阵，海森矩阵是一个 $n \times n$ 的矩阵，n 是网络中所有的权值和偏置的总数，因此，当网络参数很多时，要求极大的存储量，计算也较为复杂。

【例 3-1】 用 BFGS 拟牛顿法训练 BP 网络。

```
>> clear all;
P = [ - 1  - 1 2 2;0 5 0 5];
T = [ - 1  - 1 1 1];
net = newff(minmax(P),[3,1],{'tansig','purelin'},'trainbfg');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e - 5;
[net,tr] = train(net,P,T);
% 对网络进行仿真
a = sim(net,P)
```

运行程序，输出如下，效果如图 3-1 所示。

```
a =
    - 0.9992    - 1.0013     1.0012     1.0017
```

因为每训练一次都需要存储近似的 n 维 Hessian(海森)阵，虽然它收敛速度快，但是这样就需要占更多的内存，因此在大的网络训练中建议采用 Rprop 或者某种共轭梯度法，而对于小些的网络，采用 trainbfg 训练函数效果更好。

2. 正割拟牛顿法

正割拟牛顿法不需要存储完整的海森矩阵，除了第一次迭代外，以后各次迭代的搜索方向由下式确定

$$S(\boldsymbol{X}^{(k)}) = -\nabla f(\boldsymbol{X}^{(k)}) + A_{\mathrm{c}}\eta^{(k-1)}S(\boldsymbol{X}^{(k-1)}) + B_{\mathrm{c}}\mathrm{d}g\boldsymbol{X} \tag{3-4}$$

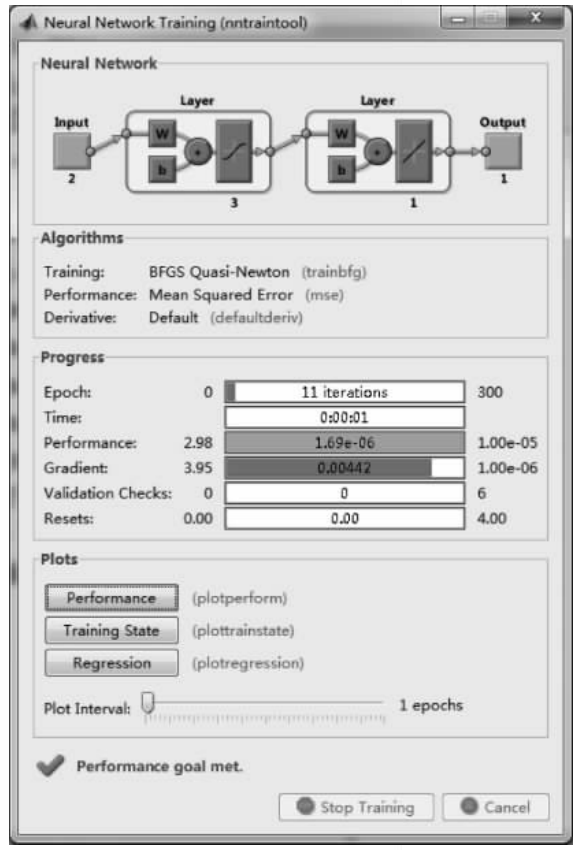

图 3-1　用 BFGS 拟牛顿法训练 BP 网络效果

式中

$$\begin{cases} \mathrm{d}g\boldsymbol{X} = \nabla f(\boldsymbol{X}^{(k)}) - \nabla f(\boldsymbol{X}^{(k-1)}) \\ A_c = -\left(1 + \dfrac{\mathrm{d}g\boldsymbol{X}^{\mathrm{T}} \times \mathrm{d}g\boldsymbol{X}}{S(\boldsymbol{X}^{(k-1)})^{\mathrm{T}}\mathrm{d}g\boldsymbol{X}}\right) \times B_c + \dfrac{\mathrm{d}g\boldsymbol{X}^{\mathrm{T}} \times \nabla f(\boldsymbol{X}^{(k)})}{S(\boldsymbol{X}^{(k-1)})^{\mathrm{T}}\mathrm{d}g\boldsymbol{X}} \\ B_c = \dfrac{S(\boldsymbol{X}^{(k-1)}) \times \nabla f(\boldsymbol{X}^{(k)})}{S(\boldsymbol{X}^{(k-1)})^{\mathrm{T}}\mathrm{d}g\boldsymbol{X}} \end{cases}$$

相对于 BFGS 拟牛顿法，正割拟牛顿法减小了存储量与计算量。实际上，正割拟牛顿法通常是需要近似计算海森矩阵的拟牛顿法与后面介绍的共轭梯度法的一种折中，它的形式与共轭梯度法相似。

【例 3-2】　用正割拟牛顿法对 BP 网络进行训练。

```
>> clear all;
P = [ - 1  - 1 2 2;0 5 0 5];
T = [ - 1  - 1 1 1];
net = newff(minmax(P),[3,1],{'tansig','purelin'},'trainoss');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e - 5;
[net,tr] = train(net,P,T);
% 对网络进行仿真
a = sim(net,P)
```

运行程序，输出如下，效果如图 3-2 所示。

```
a =
    -0.9998    -0.9993     1.0000     0.9976
```

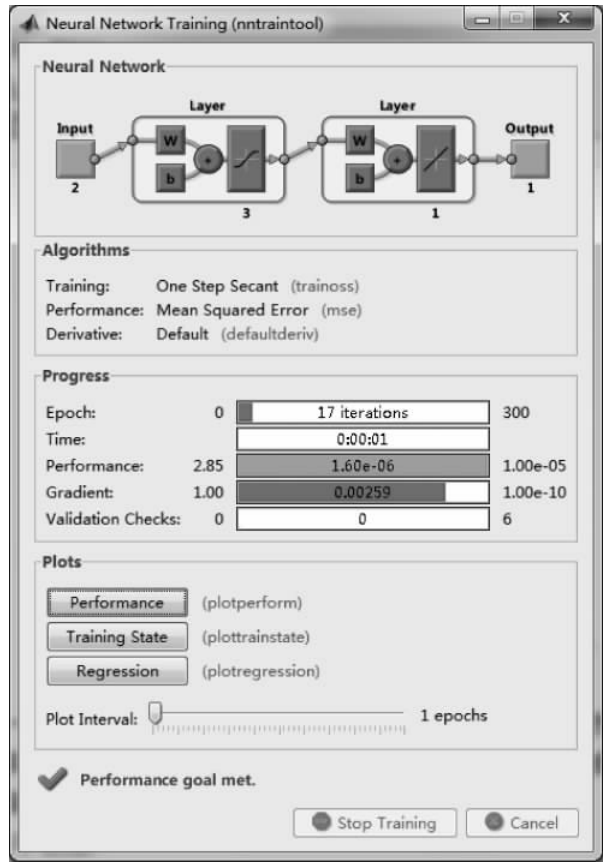

图 3-2　用正割拟牛顿法训练 BP 网络效果

由以上结果可知，网络输出值与期望值很接近，证明正割拟牛顿法训练后的 BP 网络是可行的。

3.1.2　共轭梯度法

梯度下降法收敛速度慢，而拟牛顿法计算较复杂，共轭梯度法则力图避免两种的缺点。共轭梯度法的第一步是沿负梯度方向进行搜索，然后沿当前搜索方向的共轭方向进行搜索，可以迅速达到最优值。其过程描述如下：

第一次迭代的搜索方向确定为负梯度方向，即搜索方向 $S(\boldsymbol{X}^{(0)})=-\nabla f(\boldsymbol{X}^{(0)})$，以后各次迭代的搜索方向由下式确定

$$\begin{cases} S(\boldsymbol{X}^{(k)}) = -\nabla f(\boldsymbol{X}^{(k)}) + \beta^{(k)} S(\boldsymbol{X}^{(k-1)}) \\ \boldsymbol{X}^{(k)} = \boldsymbol{X}^{(k)} + \eta^{(k)} S(\boldsymbol{X}^{(k)}) \end{cases} \tag{3-5}$$

根据 $\beta^{(k)}$ 所取形式的不同，可构成不同的共轭梯度法。常用的两种形式如下所示。

1. Fletcher-Reeves 共轭梯度

Fletcher-Reeves 共轭梯度的形式为

$$\beta^{(k)}=\frac{\boldsymbol{g}_k^{\mathrm{T}}\boldsymbol{g}_k}{\boldsymbol{g}_{k-1}^{\mathrm{T}}\boldsymbol{g}_{k-1}}$$

2. Polak-Ribiere 共轭梯度法

Polak-Ribiere 共轭梯度法的形式为

$$\beta^{(k)}=\frac{\Delta\boldsymbol{g}_{k-1}^{\mathrm{T}}\boldsymbol{g}_k}{\boldsymbol{g}_{k-1}^{\mathrm{T}}\boldsymbol{g}_{k-1}}$$

式中，$\boldsymbol{g}_k=\nabla f(\boldsymbol{X}^{(k)})$。

通常，搜索方向 $S(\boldsymbol{X}^{(k)})$ 在迭代过程中以一定的周期复位到负梯度方向，周期一般为 n（网络中所有的权值和偏差的总数）。

共轭梯度法比绝大多数常规的梯度下降法收敛都要快，而且只需增加很少的存储量及计算量，因而，对于权值很多的网络采用共轭梯度法不失为一个较好的选择。

【例 3-3】 用 Fletcher-Reeves 共轭梯度法对 BP 网络进行训练。

```
>> clear all;
P = [ - 1  - 1 2 2;0 5 0 5];
T = [ - 1  - 1 1 1];
net = newff(minmax(P),[3,1],{'tansig','purelin'},'traincgf');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e - 5;
[net,tr] = train(net,P,T);
% 对网络进行仿真
a = sim(net,P)
```

运行程序，输出如下，效果如图 3-3 所示。由于网络输出值与期望值很接近，证明训练后的 BP 网络是可行的。

```
a =
    - 1.0037    - 0.9996     0.9971     1.0031
```

【例 3-4】 用 Polak-Ribiere 共轭梯度法对 BP 网络进行训练。

```
>> clear all;
P = [ - 1  - 1 2 2;0 5 0 5];
T = [ - 1  - 1 1 1];
net = newff(minmax(P),[3,1],{'tansig','purelin'},'traincgp');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e - 5;
[net,tr] = train(net,P,T);
% 对网络进行仿真
a = sim(net,P)
```

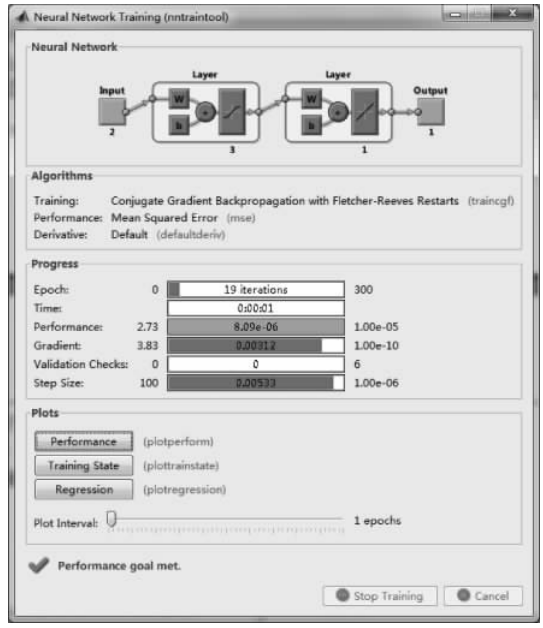

图 3-3　用 Fletcher-Reeves 共轭梯度法训练 BP 网络效果

运行程序，输出如下，效果如图 3-4 所示。

```
a =
    -0.9965    -1.0040     1.0015     0.9982
```

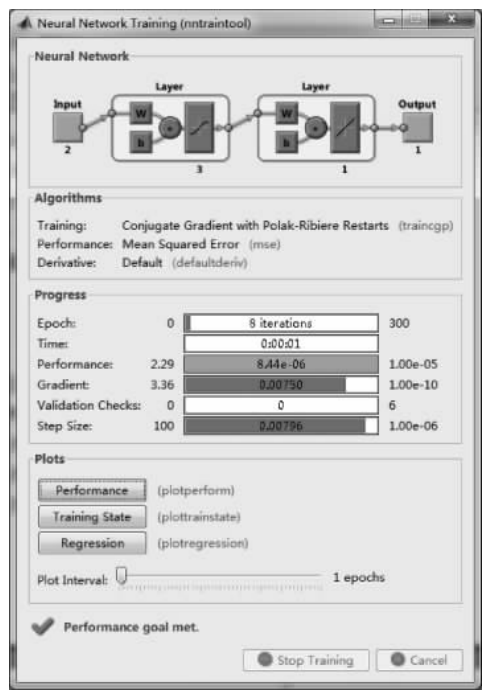

图 3-4　用 Polak-Ribiere 共轭梯度法训练 BP 网络效果

3.1.3 Levenberg-Marquardt 法

Levenberg-Marquardt 法实际上是梯度下降法和牛顿法的结合。我们知道，梯度下降法在开始几步下降较快，但随着接近最佳值，由于梯度趋于零，使得目标函数下降缓慢；而牛顿法可以在最佳值附近产生一个理想的搜索方向。Levenberg-Marquardt 法的搜索方向定为

$$S(\boldsymbol{X}^{(k)}) = -(\boldsymbol{H}^{(k)} + \lambda^{(k)}\boldsymbol{I})^{-1} \nabla f(\boldsymbol{X}^{(k)}) \tag{3-6}$$

令 $\lambda^{(k)}=1$，则 $\boldsymbol{X}^{(k+1)}=\boldsymbol{X}^{(k)}+S(\boldsymbol{X}^{(k)})$。

起始时，λ 取一个很大的数(如 1e+4)，此时相当于步长很小的梯度下降法；随着最优点的接近，λ 减小到零，则 $S(\boldsymbol{X}^{(k)})$ 从负梯度方向转向牛顿法的方向。通常，当 $f(\boldsymbol{X}^{(k+1)})<f(\boldsymbol{X}^{(k)})$ 时，减小 λ(如 $\lambda^{(k+1)}=0.5\lambda^{(k)}$)；否则增大 λ(如 $\lambda^{(k+1)}=2\lambda^{(k)}$)。

从式(3-6)中可以注意到该方法仍然需要求海森矩阵。不过由于在训练 BP 网络时目标函数常常具有平方和的形式(这也是该算法最初所解决的问题)，则海森矩阵可通过雅可比(Jacobian)矩阵进行近似计算：$\boldsymbol{H}=\boldsymbol{J}^{\mathrm{T}}\boldsymbol{J}$。雅可比矩阵包含网络误差对权值及偏置值的一阶导数，通过标准的反向传播技术计算雅可比矩阵要比计算海森矩阵容易得多。

Levenberg-Marquardt 法需要的存储量很大，因为雅可比矩阵使用一个 $Q\times n$ 矩阵，Q 为训练样本个数，n 为网络中所有的权值和偏差的总数目。为此，也产生了其他一些用以降低内存的 Levenberg-Marquardt 法。

【例 3-5】 用 Levenberg-Marquardt 法对 BP 网络进行训练。

```
>> clear all;
P = [ - 1  - 1 2 2;0 5 0 5];
T = [ - 1  - 1 1 1];
net = newff(minmax(P),[3,1],{'tansig','purelin'},'trainlm');
net.trainParam.show = 5;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e - 5;
[net,tr] = train(net,P,T);
% 对网络进行仿真
a = sim(net,P)
```

运行程序，输出如下，效果如图 3-5 所示。

```
a =
   - 1.0000    - 0.9972     1.0000     1.0000
```

Levenberg-Marquardt 法的长处是在网络权值数目较少时收敛非常迅速。

综上所述，显然，Levenberg-Marquardt 法和拟牛顿法因为要近似计算海森矩阵，需要较大的存储量，不过通常这两类方法的收敛速度较快。其中，Levenberg-Marquardt 法汇合了梯度下降法和牛顿法的优点，性能更加优良一些。共轭梯度法所需存储量较小，但收敛速度相对前两种方法慢。所以，考虑到网络的数目(即网络中所有的权值和偏差的总数目)，在选择算法对网络进行训练时，可以遵照以下原则：

- 在网络参数很少时，可以使用牛顿法或 Levenberg-Marquardt 法。

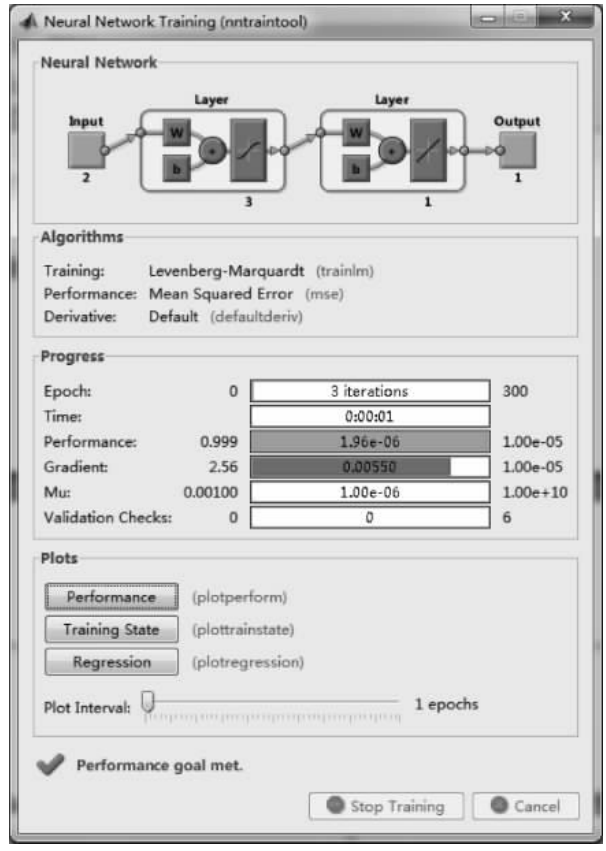

图 3-5　用 Levenberg-Marquardt 法训练 BP 网络效果

- 在网络参数适中时，可以使用拟牛顿法。
- 在网络参数很多，需要考虑到存储容量问题时，不妨选择共轭梯度法。

其实，对于不同的问题，很难比较算法的优劣。而对于特定的问题，一种通常较好的方法有时却可能不易获得良好的训练效果，甚至可能出现难以收敛到预定目标的情况。因此，在解决实际问题时，应当尝试采用多种不同类型的训练算法，以期获得满意的结果。在大多数情况下，可以使用 Levenberg-Marquardt 法和拟牛顿法。此外，弹性 BP 算法也是一种简单有效的方法。

3.2　BP 网络的工程应用

BP 网络包含了神经网络理论中最精华的部分，由于其结构简单、可塑性强，得到了广泛的应用，它的数学意义明确、步骤分明的学习算法更使其具有广泛的应用背景，其在分类、函数逼近、预测等方面都有非常好的优势，适合于解决复杂的非线性问题。

3.2.1　BP 网络在分类中的应用

下面以用于分类与模式识别的 BP 网络实例来说明 BP 神经网络在分类中的应用。

【例 3-6】 用 BP 神经网络来实现两类模式的分类，如图 3-6 所示。其模式确定的训练样本为：

P = [1 2; −1 1; −2 1; −4 0];

T = [0.2 0.8 0.8 0.2]

分析以上问题，因为处理的问题简单，所以采用最速下降 BP 算法来训练该网络。其实现的 MATLAB 代码为：

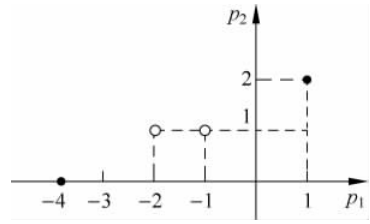

图 3-6 两类模式的分类

```
>> clear all;
% 定义输入向量及目标向量
P = [1 2; - 1 1; - 2 1; - 4 0]';
T = [0.2 0.8 0.8 0.2];
% 创建 BP 网络和定义训练函数及参数
net = newff([ - 1 1; - 1 1],[5 1],{'logsig','logsig'},'traingd');
net.trainParam.goal = 0.001;
net.trainParam.epochs = 5000;
[net,tr] = train(net,P,T);                    % 网络训练
disp('网络训练后的第一层权值为: ')
iw1 = net.iw{1}
disp('网络训练后的第一层阈值: ')
b1 = net.b{1}
disp('网络训练后的第二层权值为: ')
iw2 = net.Lw{2}
disp('网络训练后的第二层阈值: ')
b2 = net.b{2}
save li3_27 net;
% 通过测试样本对网络进行仿真
load li3_27 net;                              % 载入训练后的 BP 网络
p1 = [1 2; - 1 1; - 2 1; - 4 0]';             % 测试输入向量
a2 = sim(net,p1);                             % 仿真输出结果
disp('输出分类结果为: ')
a2 = a2 > 0.5                                 % 根据判决门限,输出分类结果
```

运行程序，输出如下，效果如图 3-7 所示。

```
网络训练后的第一层权值为:
iw1 =
   - 4.7114    - 4.1185
   - 6.3031     0.7245
    4.4480    - 4.4001
    3.1063    - 5.4349
    6.1978    - 2.5337
网络训练后的第一层阈值:
b1 =
    6.2576
    3.0154
    0.0066
    3.1312
    5.9266
```

```
网络训练后的第二层权值为:
iw2 =
    -3.6347    2.3032    -1.7934    0.2730    -3.3708
网络训练后的第二层阈值:
b2 =
    1.8086
输出分类结果为:
a2 =
     0     1     1     1
```

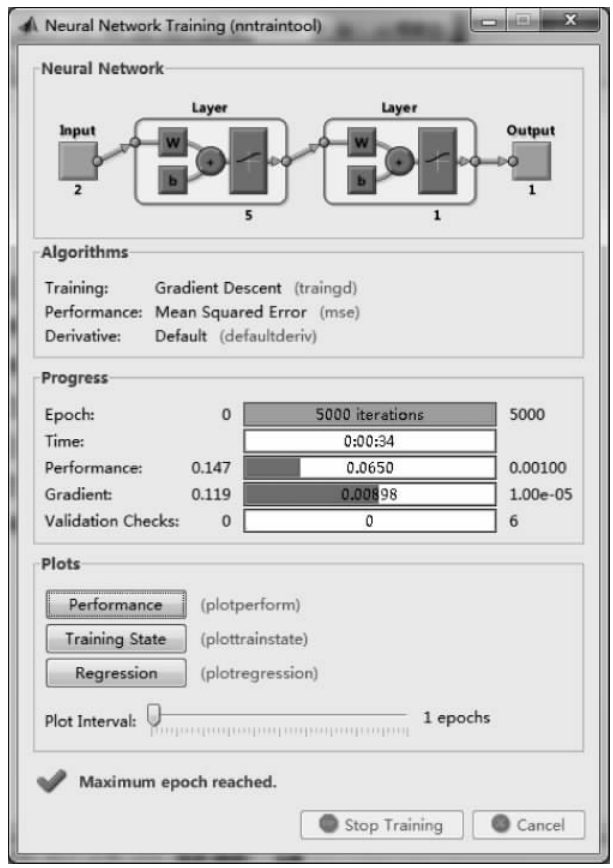

图 3-7　网络的训练过程效果图

由以上分类输出结果可知,上述为两类模式的分类。

3.2.2　函数逼近

下面通过设计一个简单的 BP 网络,实现对非线性函数的逼近。

【例 3-7】 BP 网络实现函数的逼近效果。

```
>> clear all;
% 将要逼近的非线性函数设为正弦函数,其频率参数可以调节
k = 1;
```

```
p = [ - 1:.05:1];
t = sin(k * pi * p);
% 假设频率参数为 1,绘制此函数的曲线
plot(p,t,':')                                    % 效果如图 3 - 8 所示
xlabel('时间');
ylabel('非线性函数');
>> % 建立网络
n = 10;
net = newff(minmax(p),[n,1],{'tansig','purelin'},'trainlm');
y1 = sim(net,p);                                 % 网络仿真
figure;
plot(p,t,'r:',p,y1,' - ')                        % 效果如图 3 - 9 所示
xlabel('时间');
ylabel('仿真输出 --   原函数 - ')
legend('原函数曲线','BP 网络输出曲线')
>> % 网络训练参数设置及训练
net.trainParam.epochs = 50;
net.trainParam.goal = 0.01;
net = train(net,p,t);                            % 效果如图 3 - 10 所示
```

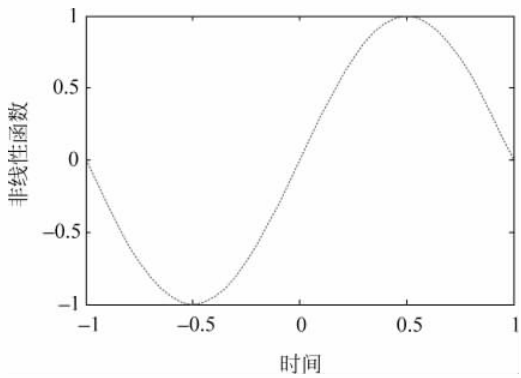

图 3-8　非线性函数曲线

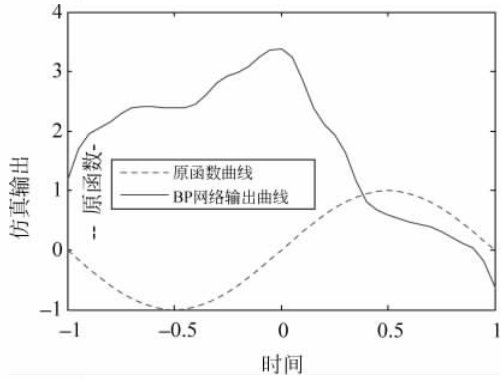

图 3-9　未训练网络的输出效果

从图 3-10 可看出,网络训练速度非常快,经过 2 次循环迭代过程就达到了要求的精度。

```
>> % 网络测试
y2 = sim(net,p);
figure;
plot(p,t,' - ',p,y1,' -- ',p,y2,' + ')           % 效果如图 3 - 11 所示
xlabel('时间');
ylabel('仿真输出')
legend('原曲线','BP 网络输出曲线','训练后曲线');
```

由图 3-11 可看出,得到的曲线和原始的非线性函数曲线很接近,这说明经过训练后,BP 网络对非线性函数的逼近效果相当好。

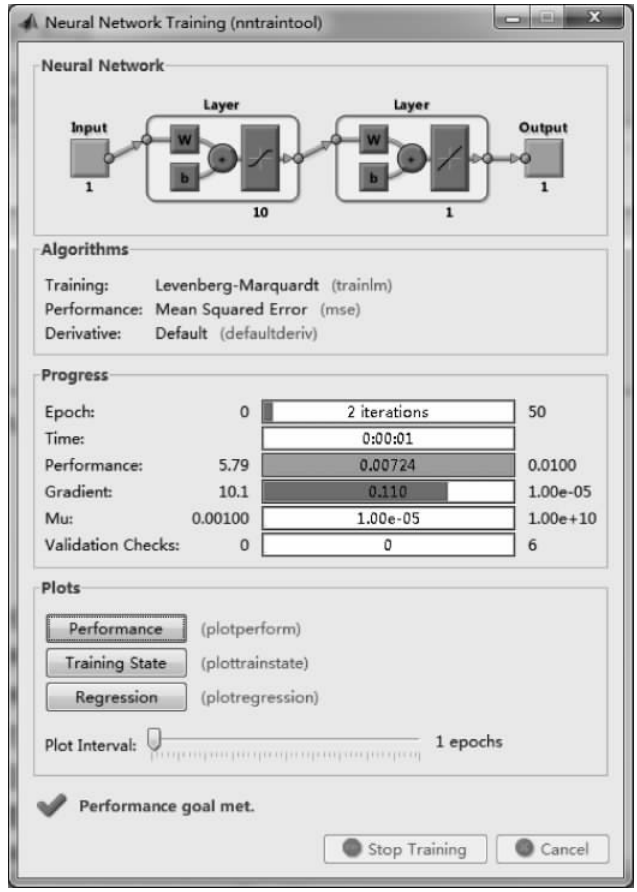

图 3-10　BP 网络训练效果图

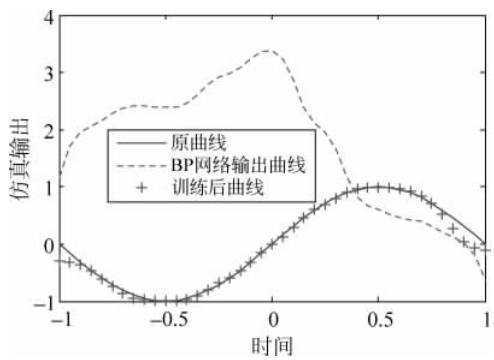

图 3-11　训练后网络的输出效果图

3.2.3　BP 网络用于胆固醇含量的估计

下面是一个神经网络在医学中的应用的例子。

【例 3-8】 拟设计一台仪器，通过对血液样本进行光谱分析来测试血清中的胆固醇水平。共采集了 264 位病人的血液样本，对其光谱分析共发现 21 种光谱波长，对这些病人，通过血清分离，同样也测量了 hdl、ldl、vldl 胆固醇水平。

其实现的 MATLAB 代码为：

```
>> clear all;
load choles_all;
[pn,meanp,stdp,tn,meant,stdt] = prestd(p,t);
% 删除一些数据，只保留占 99.9% 的主要成分数据
[ptrans,transMat] = prepca(pn,0.001);
[R,Q] = size(ptrans)                              % 检查转换后的数据矩阵大小
R =      4
Q =    264
```

从以上结果可看出，输入数据由 21 组减少到 4 组，由此可看出原数据有着很大的冗余。

```
% 将数据分成几部分，1/2 用于训练、1/4 用于验证及 1/4 用于测试
>> iitst = 2:4:Q;
iival = 4:4:Q;
iitr = [1:4:Q 3:4:Q];
val.P = ptrans(:,iival);
val.T = tn(:,iival);
test.P = ptrans(:,iitst);
test.T = tn(:,iitst);
ptr = ptrans(:,iitr);
ttr = tn(:,iitr);
% 创建网络，隐层神经元初步设计为 5 个，因为需要得到 3 个目标
% 网络输出层设计为 3 个神经元
net = newff(minmax(ptr),[5,3],{'tansig','purelin'},'trainlm');
% 采用 Levenberg - Marquardt 算法对 BP 进行网络训练
% 创建网络，隐层神经元初步设计为 5 个，因为需要得到 3 个目标，网络输出层设计为 3 个神经元
net = newff(minmax(ptr),[5,3],{'tansig','purelin'},'trainlm');     % 效果如图 3 - 12 所示
% 绘制相应误差曲线图
plot(tr.epoch,tr.perf,'r:',tr.epoch,tr.vperf,tr.epoch,tr.tperf,'-.');
                                                                   % 效果如图 3 - 13 所示
legend('训练误差曲线','验证误差曲线','测试误差曲线');
ylabel('平方差');xlabel('时间');
```

最后，对网络响应进行分析，将所有的数据都放在整个数据集中，包括训练数据、验证数据和测试数据，然后网络输出和相应的期望输出向量进行线性回归分析。线性回归之前需要对网络输出进行反规范化转换，因为有 3 组输出值，所以应进行 3 次线性回归，代码为：

```
an = sim(net,ptrans);
a = poststd(an,meant,stdt);
for i = 1:3
```

```
    figure(i);
    [m(i),b(i),r(i)] = postreg(a(i,:),t(i,:));          % 效果如图 3－14～图 3－16 所示
end
```

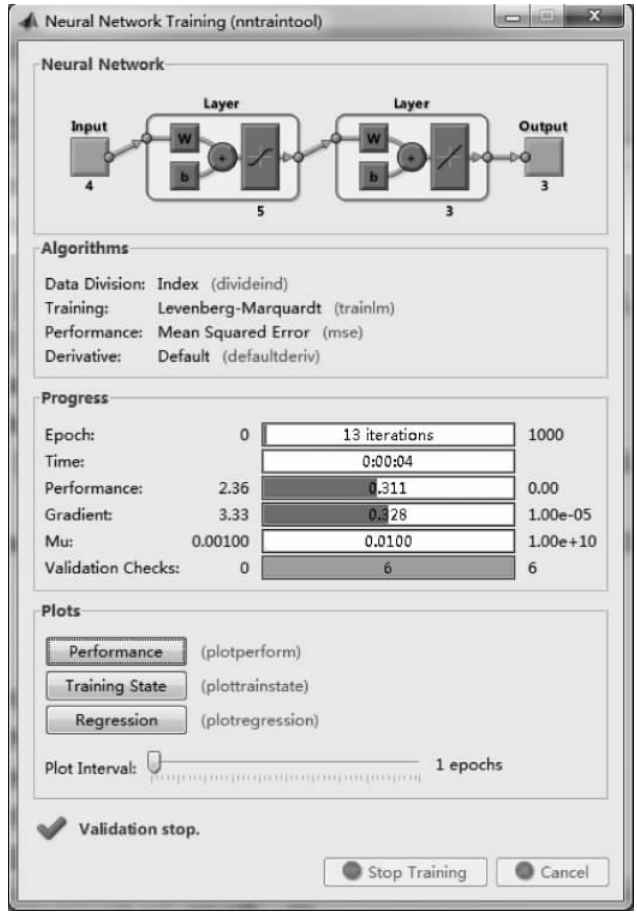

图 3-12　Levenberg-Marquardt 算法训练 BP 网络效果

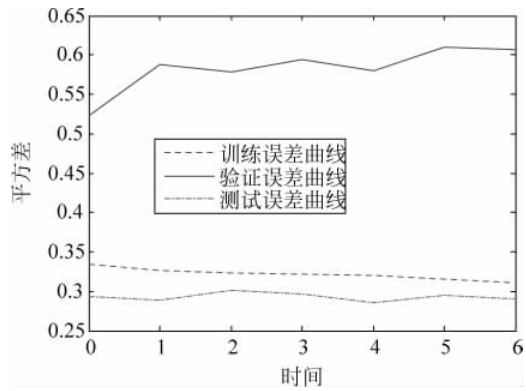

图 3-13　训练误差、验证误差、测试误差曲线

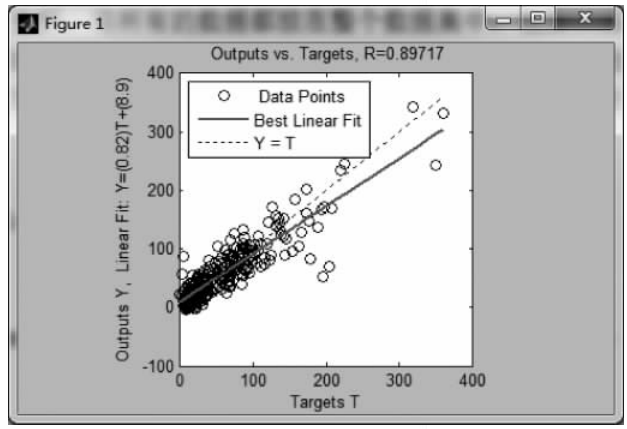

图 3-14　hdl 线性回归

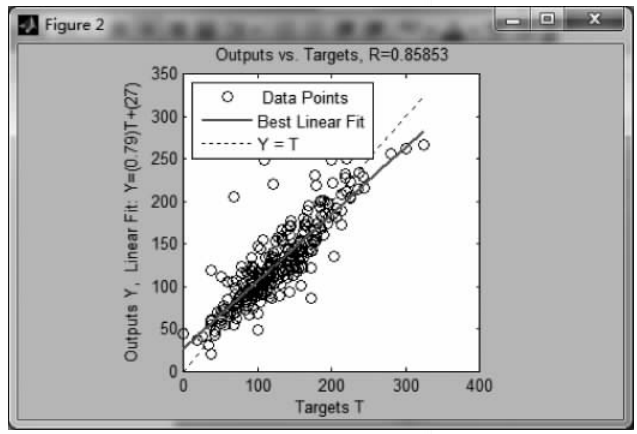

图 3-15　ldl 线性回归

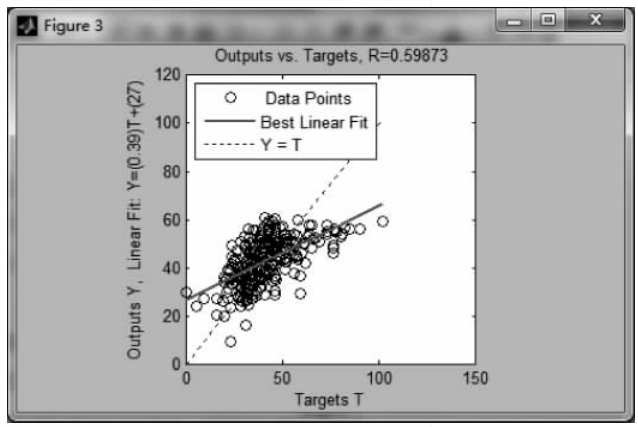

图 3-16　vhdl 线性回归

可以看出，前面两个输出对期望的跟踪较好，相应的 R 值几乎达到了 0.9，而第三个输出则吻合得不是很理想。如果需要进一步研究，可以尝试其他网络结构（如增加隐

层神经元数目),或在训练中不使用提早停止训练的方式,而采用 Bayesian 规范化方法尝试对 BP 网络模型进行改进,如将隐层神经元数目增加为 15 个,其实现代码为:

```
net = newff(minmax(ptr),[20,3],{'tansig','purelin'},'trainlm');
[net,tr] = train(net,ptr,ttr,[],[],val,test);          %效果如图 3-17 所示
plot(tr.epoch,tr.perf,':',tr.epoch,tr.vperf,'-.',tr.epoch,tr.tperf);
%效果如图 3-18 所示
legend('训练后误差曲线','验证误差曲线','测试误差曲线');
ylabel('平方差');xlabel('时间');
```

绘制改变隐层神经元个数后的 3 种线性回归,代码为:

```
an = sim(net,ptrans);
a = poststd(an,meant,stdt);
for i = 1:3
    figure(i);
    [m(i),b(i),r(i)] = postreg(a(i,:),t(i,:));          %效果如图 3-19～图 3-21 所示
end
[m(1),b(1),r(1)] = postreg(a(1,:),t(1,:));
```

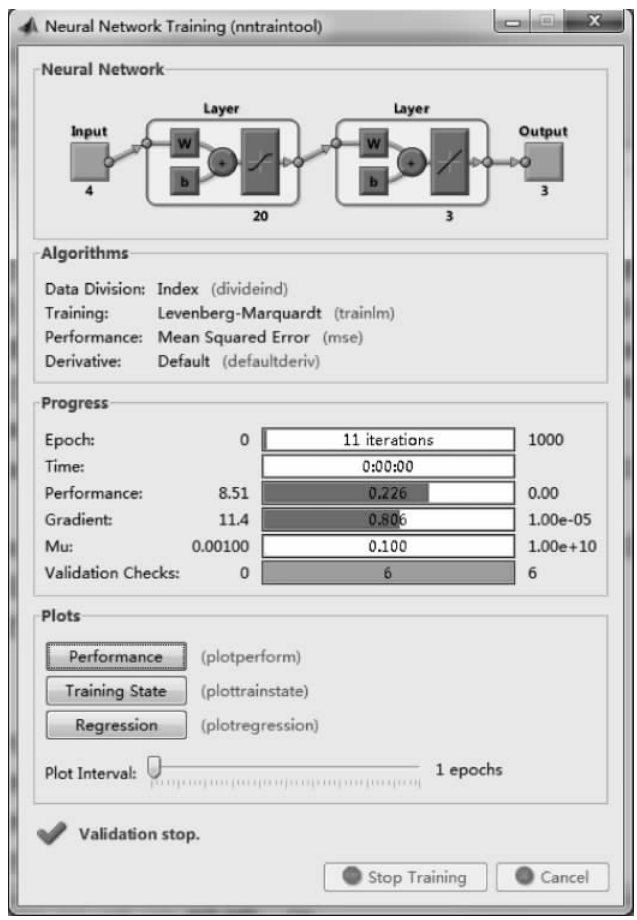

图 3-17　增加隐层神经元个数的网络训练过程

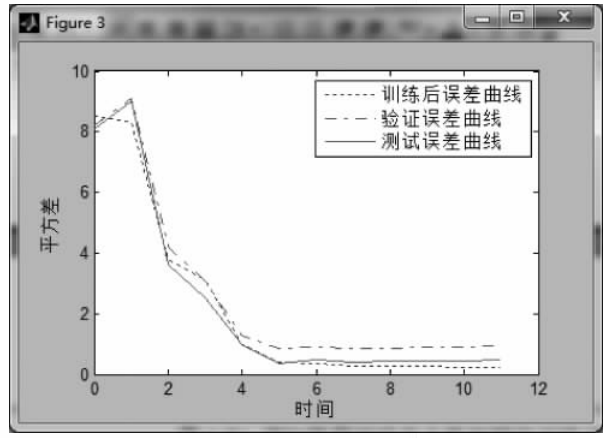

图 3-18　增加隐层神经元个数的三种误差曲线

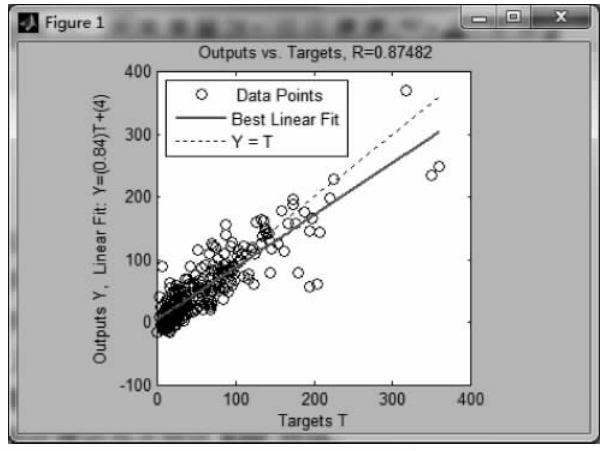

图 3-19　增加隐层神经元个数后的 hdl 线性回归

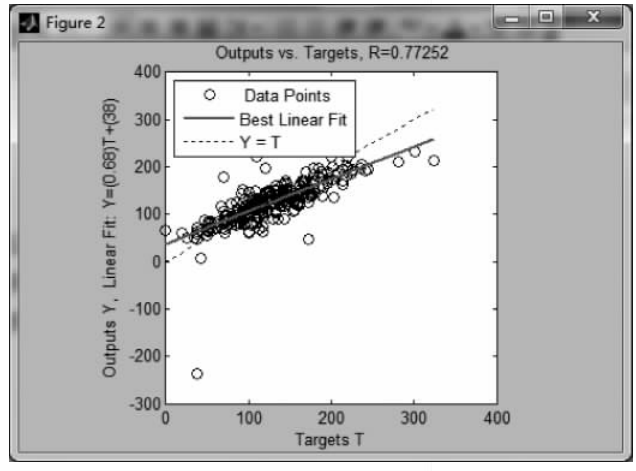

图 3-20　增加隐层神经元个数后的 ldl 线性回归

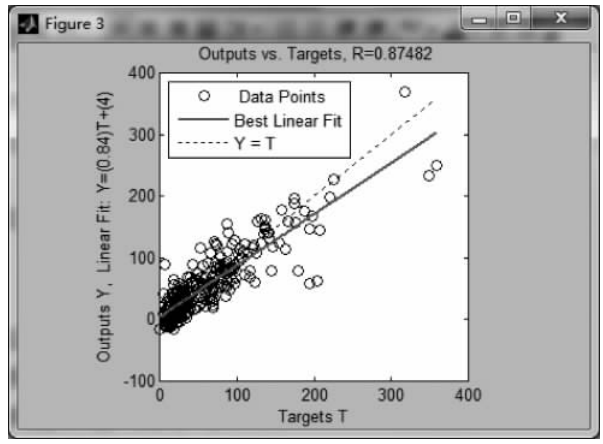

图 3-21　增加隐层神经元个数后的 vhdl 线性回归

3.2.4　模式识别

计算机模式识别是近年来发展起来的技术。如果利用机器来识别银行的签字，那么它就能在相同的时间里做更多的工作，既节省了时间，又节约了人力物力资源。

【例 3-9】　演示 BP 网络在模式识别中的应用。

设计一个网络并训练它来识别字母表中的 26 个字母，数字成像系统对每个字母进行分析将其变成数字信号。图 3-22 显示的就是字母 A 的网络图。

但是，图 3-22 只是理想图像系统得到的结果。实际情况总会存在一些噪声干扰，或者存在一些非线性因素，实际得到的字母网格图如图 3-23 所示。

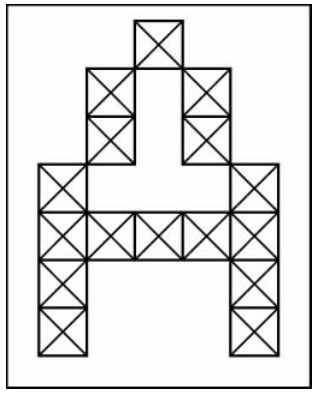

图 3-22　字母 A 的网格图

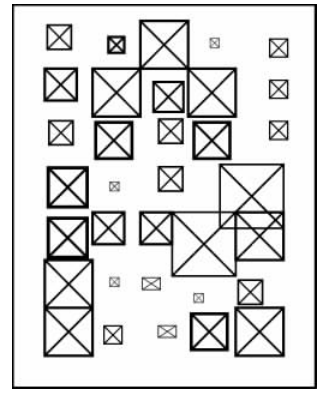

图 3-23　带有噪声的字母 A 的网格图

要求设计网络，不仅能够对理想输入向量进行很好的分类，也能够准确识别含有误差的输入向量。

本例中，26 个字母均被定义成输入向量，每个代表字母的输入向量均有 35 个元素，这样就组成一个输入向量矩阵 alphabet。期望输出向量被定义成一个变量 targets，期望

输出向量含有 26 个元素，字母在字母表中所占位置片元素为 1，其他位置为 0。例如，因为字母 A 在字母表中是第一个字母，所以其期望输出向量为(1,0,…,0)。

根据上面提出的设计要求，输入向量具有 35 个元素，网络输出就是反映字母所在位置的具有 26 个元素的输出向量。如果网络正确，那么输入一个字母，网络就能输出一个向量，它对应位置的元素值，其他位置的元素值为 0。

另外，网络还必须具有容错能力。因为实际情况下，网络不可能接收到一个理想的布尔向量作为输入。当噪声均值为 0，标准差小于等于 0.2 时，系统应该能够做到正确的识别输入向量，这就是网络的容错能力。

对于辨识字母的要求，神经网络被设计成两层 BP 网络，具有 35 个输入端，输出层有 26 个神经元，如图 3-24 所示。隐层和输出层的神经元传递函数均为 logsigmoid，这是因为 logsigmoid 函数输出量在(0,1)区间内，恰好满足输出布尔值的要求。

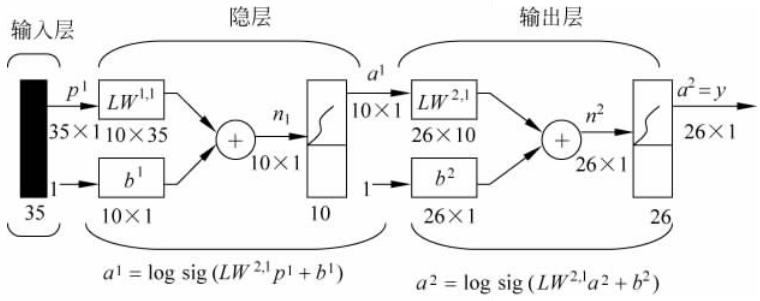

图 3-24　网络结构

隐层含有 10 个神经元，神经元数目的选择是依据经验和猜测而定。在实际训练中，如果训练过程不理想，可以适当增加隐层神经元数目。

训练网络就是要使其输出向量正确代表字母向量。然而，由于噪声信号的引入，网络可能会产生不精确的输出，通过竞争传递函数 compet 训练后，就能够保证正确识别带有噪声的字母向量。

其实现的 MATLAB 代码为：

```
>> clear all;
[alphabet,targets] = prprob;
[R,Q] = size(alphabet);
[S2,Q] = size(targets);
S1 = 10;
[R,Q] = size(alphabet);
[S2,Q] = size(targets);
P = alphabet;
net = newff(minmax(P),[S1,S2],{'logsig','logsig'},'traingdx');   % 构建 BP 网络
net.LW{2,1} = net.LW{2,1} * 0.01;
net.b{2} = net.b{2} + 0.01;
```

为了使产生的网络对输入向量有一定的容错能力，最好的办法是既使用理想的信号，又使用带有噪声的信号对网络进行训练。具体过程如下：

(1) 应用理想的输入信号对网络进行训练，直到其均方差达到精度为止。

(2) 应用10组理想信号和带有噪声的信号对网络进行训练。在进行训练时,同时对两组相同的无噪声字母信号样本进行训练,目的是确保网络能够正确分辨理想信号。

进行了上述的训练之后,网络对无噪声信号进行辨识的时候可能也会采用有噪声信号的方法,这样就会产生不必要的资源浪费。这时可以再次训练网络,这一次就只需要应用理想信号进行训练,从而保证在输入理想字母信号时,网络具备良好的辨识能力。

```
>>% 无噪声训练
T = targets;
net.performFcn = 'sse';
net.trainParam.goal = 0.1;
net.trainParam.show = 20;
net.trainParam.epochs = 5000;
net.trainParam.mc = 0.95;
[net,tr] = train(net,P,T);                    % 效果如图 3-25 所示
```

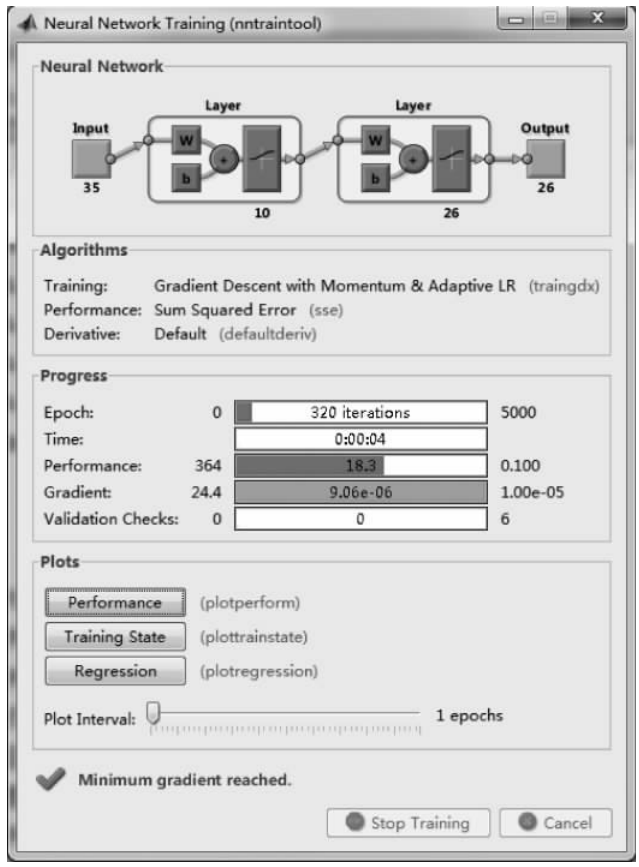

图 3-25 无噪声网络的训练过程

为了保证设计的网络对于噪声信号有一定的容错性,可以应用两套理想字母向量和两套有噪声字母向量作为训练样本,这样期望输出信号就包括4组重复字母向量信号,加入噪声采用均值为0.1和0.2两种信号,并且在循环中重复循环训练10次。这样做的目的,就是保证神经元正确分辨有噪声字母的同时,也能对理想字母向量做到正确的

识别。

```
%有噪声训练
netn = net;
netn.trainParam.goal = 0.6;
netn.trainParam.epochs = 300;
T = [targets targets targets targets];
for pass = 1:10
    P = [alphabet,alphabet,(alphabet + randn(R,Q) * 0.1),(alphabet + randn(R,Q) * 0.2)];
    [netn,tr] = train(net,P,T);                          %效果如图 3-26 所示
end
>>%再次无噪声训练,这次训练是为了使网络进行理想信号识别时,节省资源
netn.trainParam.goal = 0.1;
netn.trainParam.epochs = 500;
netn.trainParam.show = 5;
P = alphabet;
T = targets;
```

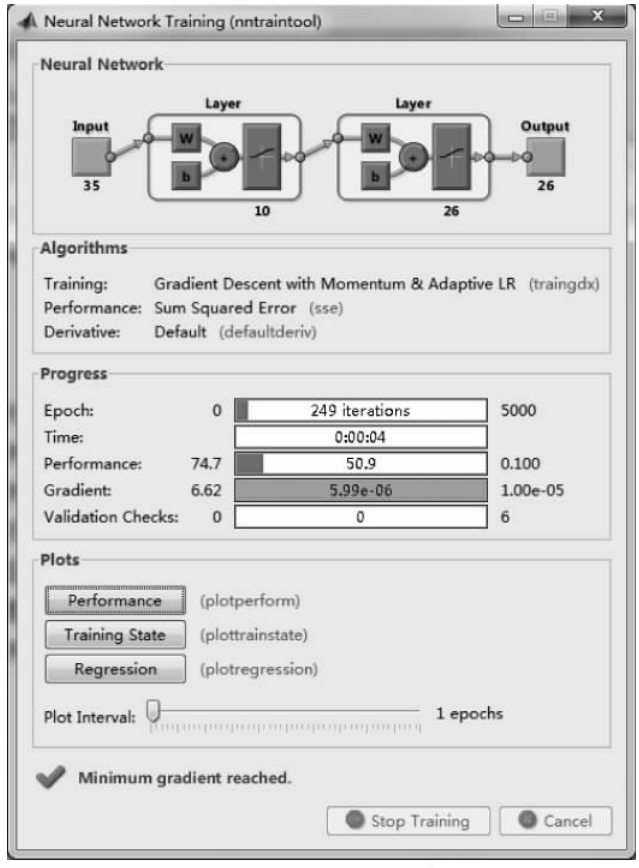

图 3-26　有噪声网络训练过程

为了测量所设计的神经网络模式识别系统的可靠性,用上百个输入向量加入了不同的噪声信号进行了测试,并绘制网络识别错误与噪声信号的比较曲线。

```
%系统性能
noise_range = 0:.05:.5;
max_test = 100;
network1 = [ ];
network2 = [ ];
T = targets;
for noiselevel = noise_range
    errors1 = 0;
    errors2 = 0;
    for i = 1:max_test
        P = alphabet + randn(35,26) * noiselevel;
        A = sim(net,P);
        AA = compet(A);
        errors1 = errors1 + sum(sum(abs(AA - T)))/2;
        An = sim(netn,P);
        AAn = compet(An);
        errors2 = errors2 + sum(sum(abs(AAn - T)))/2;
    end
network1 = [network1 errors1/26/100];
    network2 = [network2 errors2/26/100];
end
plot(noise_range,network1 * 100,'-- ',noise_range,network2 * 100);   %效果如图 3-27 所示
xlabel('噪声指标');
ylabel('无噪声训练网络 --  有噪声训练网络 --- ');
legend('无噪声训练网络','有噪声训练网络');
```

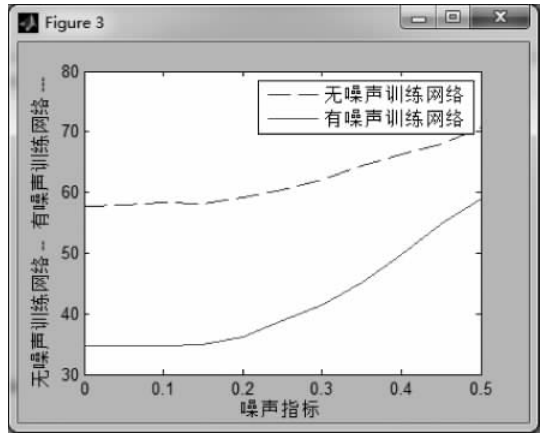

图 3-27　识别误差与噪声的关系

如果要达到更高的精度要求，可以增加网络训练时间或者增加网络隐层神经元数目。当然，可以将输入的字母得到由原来的 5×7 网格加大为 10×14 的网格形式。如果网络要求对于噪声信号有更高的容错性，还可以在训练时增加噪声信号的数量。

下面随意选取几个字母，并附加一定的噪声信号作为输入信号，用训练好的网络进行仿真。其实现 MATLAB 代码为：

```
noisyJ = alphabet(:,1) + randn(35,1) * 0.2;
```

```
figure;plotchar(noisyJ);                                    % 效果如图 3-28 所示
A2 = sim(net,noisyJ);
A2 = compet(A2);
answer = find(compet(A2) == 1);
figure;plotchar(alphabet(:,answer));                        % 效果如图 3-29 所示
noisyJ = alphabet(:,10) + randn(35,1) * 0.2;
figure;plotchar(noisyJ);                                    % 效果如图 3-30 所示
A2 = sim(net,noisyJ);
A2 = compet(A2);
answer = find(compet(A2) == 1);
figure;plotchar(alphabet(:,answer));                        % 效果如图 3-31 所示
noisyJ = alphabet(:,23) + randn(35,1) * 0.2;
figure;plotchar(noisyJ);                                    % 效果如图 3-32 所示
A2 = sim(net,noisyJ);
A2 = compet(A2);
answer = find(compet(A2) == 1);
figure;plotchar(alphabet(:,answer));                        % 效果如图 3-33 所示
```

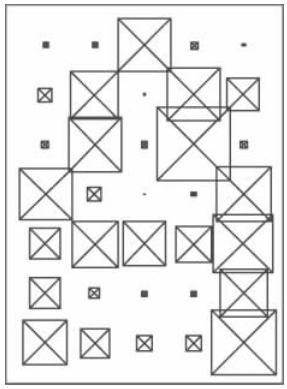

图 3-28　含噪声的输入字母 A

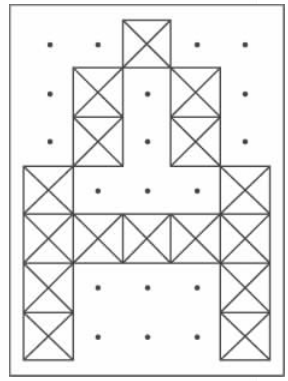

图 3-29　字母 A 的识别结果

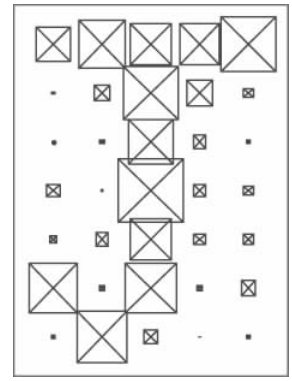

图 3-30　含噪声的输入字母 J

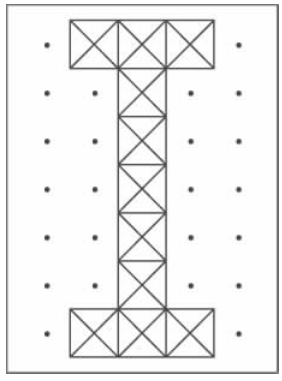

图 3-31　字母 J 的识别结果

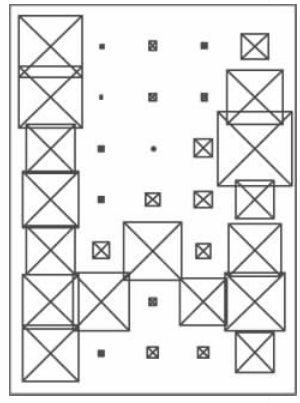

图 3-32　含噪声的输入字母 W

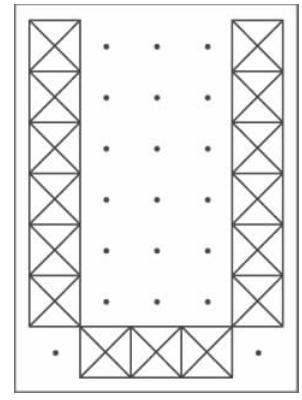

图 3-33　字母 W 的识别结果