Chapter 6 Nonlinear Regression – Neural Network

6.1 Generic mapping

The generic empirical retrieval problem

$$Y = f(X) \qquad (1)$$

is essentially a mapping from X to Y. This empirical mapping can be performed using conventional tools (linear and nonlinear regression).

Linear regression is an appropriate tool for developing many empirical algorithms. It is simple to apply and has a well-developed theoretical basis. In the case of linear regression, a linear model is constructed for transfer function (TF) *f,*

$$y_i = \sum_j a_j x_j \qquad (2)$$

This model is linear with respect to both a and X, thus it provides a linear approximation of the TF with respect to X. The most important limitation of such a linear approximation is that it works well over a broad range of variability of the arguments only if the function which it represents (TF in our case) is linear. If the TF, f, is nonlinear, linear regression can only provide a local approximation; when applied globally, the approximation becomes inaccurate.

Because, TFs are generally nonlinear functions of their arguments X, linear regression and a nonlinear approximation with respect to X is often better suited for modeling TFs. In this case, f can be introduced as a linear expansion using a basis of nonlinear functions $\{\varphi_j\}$:

$$y_i = \sum_j a_j \quad \varphi_j(X) \qquad (3)$$

Finally, nonlinear regression may be applied. For example, f in (1) can be specified as a complicated nonlinear function, f_{NR}:

$$y_i = f_{NR}(X, a) \qquad (4)$$

The expression (3) is nonlinear with respect to its argument *X* but linear with respect to the parameters *a*. The nonlinear regression (4) is nonlinear both with respect to its argument, *X*, and with respect to the vector of regression coefficients, *a.* However, in either case, we must specify in advance a particular type of nonlinear function f_{NR}, or φ_j. Thus, we are forced to implement a particular type of nonlinearity a priori. This may not always be possible, because we may not know in advance what kind of nonlinear behavior a particular TF demonstrates, or this nonlinear behavior may be different in different regions of the TF's domain. If an inappropriate nonlinear regression function is chosen, it may represent a nonlinear TF with less accuracy than with its linear counterpart.

In the situation described above, where the TF is nonlinear and the form of nonlinearity is not known, we need a more flexible, self-adjusting approach that can accommodate various types of nonlinear behavior representing a broad class of nonlinear mappings. Neural networks (NNs) are well-suited for a very broad class of nonlinear approximations and mappings.

6.2 A feed-forward neural network

A feed-forward neural network (NN) is a non-parametric statistical model for extracting nonlinear relations in the data. A common NN model configuration is to place between the input and output variables (also called `neurons'), a layer of `hidden neurons' (Fig.1). The value of the jth hidden neuron is

$$y_j = \tanh\left(\sum_i w_{ij} x_i + b_j\right), \qquad (5)$$

where x_i is the *i*th input, w_{ij} the weight parameters and b_j the bias parameters.

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

The output neuron is given by

$$z = \sum_j \tilde{w}_j y_j + \tilde{b}. \tag{6}$$

A cost function

$$J = \langle (z - z_{obs})^2 \rangle \tag{7}$$

measures the mean square error between the model output z and the observed values z_{obs}. The parameters w_{ij}, $\tilde{w}_j$, b_j and $\tilde{b}$ are adjusted as the cost function is minimized. The procedure, known as network training, yields the optimal parameters for the network. As in standard optimization procedure, steepest descent with momentum and adaptive learning rates was used during the optimization.

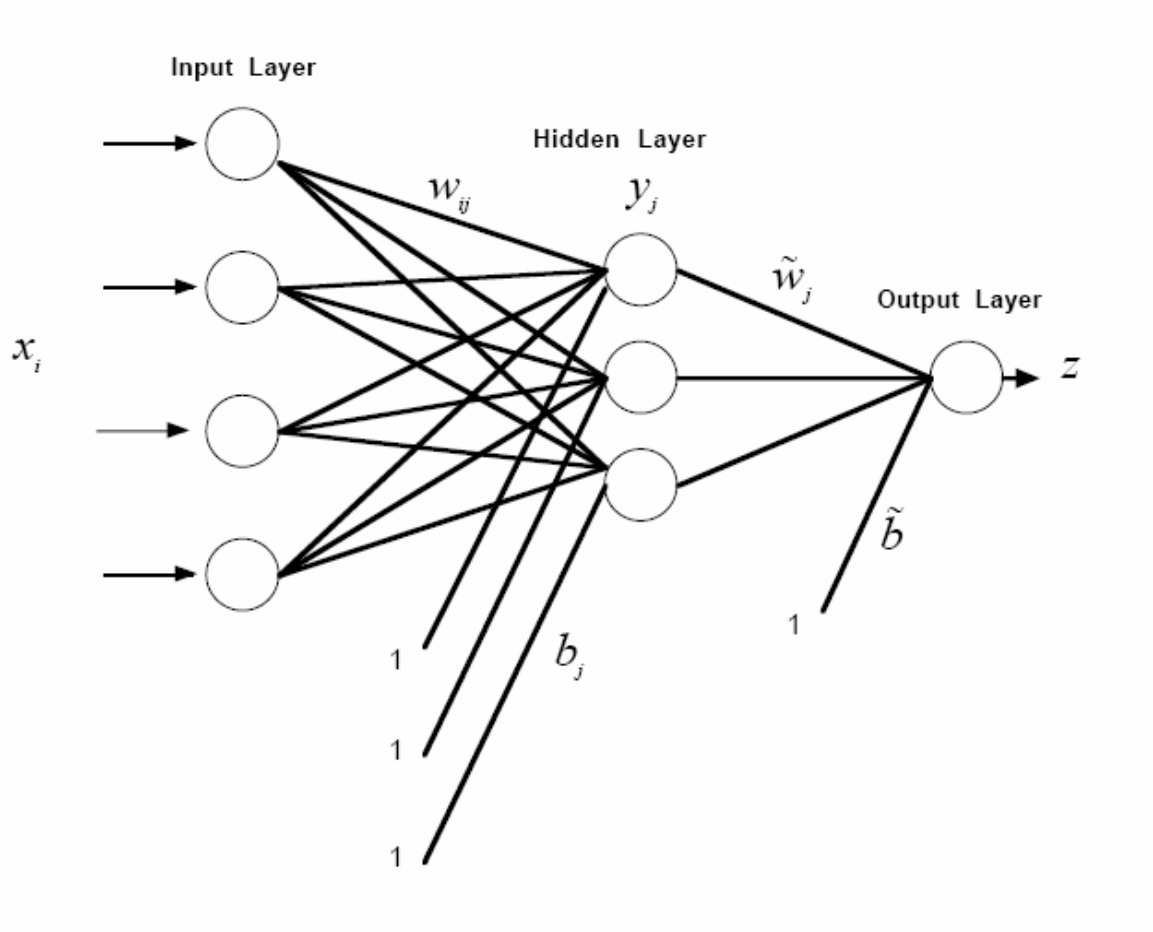


Fig.1 An example of a neural network model, where there are four neurons in the input layer, three in the hidden layer, and one in the output layer. The parameters w_{ij} and $\tilde{w}_j$ are the weights, and b_j and $\tilde{b}$ are the biases. The parameters b_j and $\tilde{b}$ can also be regarded as the weights for constant inputs of value 1.

6.3 Optimization

6.3.1 Newton's method

Considering the relation

$$y = w_0 + \sum_{l=1}^{L} w_l f_l \,, \tag{8}$$

where $f_l = f_l(x_1, \ldots, x_m)$, and the polynomial fit is a special case. Although the response variable y is nonlinearly related to the predictor variables $x_1, \ldots, x_m$ (as f_l is in general a nonlinear function), y is a linear function of the parameters $\{w_l\}$. It follows that the objective function

$$J = \sum (y - y_{\rm d})^2 \,, \tag{9}$$

(with $y_{\rm d}$ the target data and the summation over all samples) is a quadratic function of the $\{w_l\}$, which means that the objective function $J(w_0, \ldots, w_L)$ is a parabolic surface, which has a single minimum, the global minimum.

In contrast, when y is a nonlinear function of $\{w_l\}$, the objective function surface is in general filled with numerous hills and valleys, i.e. there are usually many local minima besides the global minimum. (If there are symmetries among the parameters, there can even be multiple global minima). Thus nonlinear optimization involves finding a global minimum among many local minima. The difficulty faced by the optimization algorithm is similar to that encountered by a robot rover sent to explore the rugged surface of a planet. The rover can easily fall into a hole or a valley and be unable to escape from it, thereby never reaching its final objective, the global minimum. Thus nonlinear optimization is vastly more tricky than linear optimization, with no guarantee that the algorithm actually finds the global minimum, as it may become trapped by a local minimum.

In essence, with NN models, one needs to minimize the objective function J with respect to $\mathbf{w}$ (which includes all the weight and offset/bias parameters), i.e. find the optimal parameters which will minimize J. It is common to solve the minimization problem using an iterative procedure. Suppose the current approximation of the solution is $\mathbf{w}_0$. A Taylor series expansion of $J(\mathbf{w})$ around $\mathbf{w}_0$ yields

$$J(\mathbf{w}) = J(\mathbf{w}_0) + (\mathbf{w}-\mathbf{w}_0)^{\mathrm{T}} \nabla J(\mathbf{w}_0) + \frac{1}{2}(\mathbf{w}-\mathbf{w}_0)^{\mathrm{T}} \mathbf{H} (\mathbf{w}-\mathbf{w}_0) + \cdots , \tag{10}$$

where ∇J has components $\partial J/\partial w_i$, and $\mathbf{H}$ is the *Hessian matrix*, with elements

$$(\mathbf{H})_{ij} \equiv \left. \frac{\partial^2 J}{\partial w_i \partial w_j} \right|_{\mathbf{w}_0} . \tag{11}$$

Applying the gradient operator to (10), we obtain

$$\nabla J(\mathbf{w}) = \nabla J(\mathbf{w}_0) + \mathbf{H} (\mathbf{w}-\mathbf{w}_0) + \cdots \tag{12}$$

Next, let us derive an iterative scheme for finding the optimal w. At the optimal w, $\nabla \boldsymbol{J}(\boldsymbol{w}) = 0$, and (12), with higher order terms ignored, yields

$$\mathbf{H} (\mathbf{w}-\mathbf{w}_0) = -\nabla J(\mathbf{w}_0) , \quad \text{i.e.} \quad \mathbf{w} = \mathbf{w}_0 - \mathbf{H}^{-1} \nabla J(\mathbf{w}_0) . \tag{13}$$

This suggests the following iterative scheme for proceding from step k to step $k+1$:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \mathbf{H}_k^{-1} \nabla J(\mathbf{w}_k) . \tag{14}$$

This is known as Newton's method. In the 1-dimensional case, (14) reduces to

$$w_{k+1} = w_k - \frac{J'(w_k)}{J''(w_k)} , \tag{15}$$

for finding a root of $J'(w) = 0$, where the prime and double prime denote respectively the first and second derivatives.

In the multi-dimensional case, if $\mathbf{w}$ is of dimension L, then the Hessian matrix $\mathbf{H}_k$ is of dimension $L \times L$. Computing $\mathbf{H}_k^{-1}$, the inverse of an $L \times L$ matrix, may be computational too costly. Simplification is needed, resulting in quasi-Newton methods.

6.3.2 Gradient descent method

A major simplification of Newton's method (14) is to use a parameter $\boldsymbol{\eta}$ to replace $\boldsymbol{H}_k^{-1}$, i.e.,

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \eta \nabla J(\mathbf{w}_k)\,, \tag{16}$$

η is called learning rate, and can be either a fixed constant, or calculated by a line minimization algorithm. In the former case, one simply takes a step of fixed size along the direction of the negative gradient of J. In the later, one proceeds along the negative gradient of J until one reaches the minimum of J along that direction (Fig. 6.1). More precisely. Suppose at step *k*, we have estimated parameters $\mathbf{w}_k$. We then descend along the negative gradient of the objective function, i.e. travel along the direction

$$\mathbf{d}_k = -\nabla J(\mathbf{w}_k)\,. \tag{17}$$

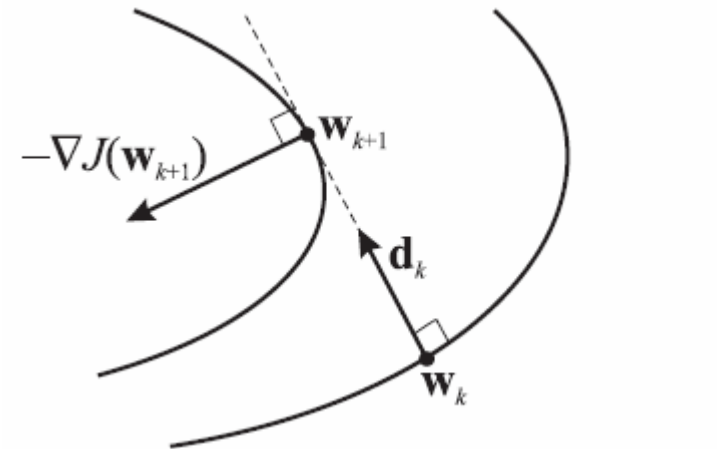


Fig6.1 The gradient descent approach starts from the parameters w_k estimated at step k of an iterative optimization process. The descent path $\mathbf{d}_k$ is chosen along the negative gradient of the objective function J, which is the steepest descent direction. Note that $\mathbf{d}_k$ is perpendicular to the J contour where $\mathbf{w}_k$ lies. The descent along $\mathbf{d}_k$ proceeds until it is tangential to a second contour at $\mathbf{w}_{k+1}$, where the direction of steepest descent is given by $-\nabla J(\mathbf{w}_{k+1})$. The process is iterated.

We then travel along $\mathbf{d}_k$, with our path described by $\mathbf{w}_k + \eta \mathbf{d}_k$, until we reach the minimum of J along this direction. Going further along this direction would mean we would actually be acending rather than descending, so we should stop at this minimum of J along $\mathbf{d}_k$, which occurs at

$$\frac{\partial}{\partial \eta} J(\mathbf{w}_k + \eta \mathbf{d}_k) = 0\,, \tag{18}$$

thereby yielding the optimal step size η. The differentiation by η gives

$$\mathbf{d}_k^{\mathrm{T}} \nabla J(\mathbf{w}_k + \eta \mathbf{d}_k) = 0\,.$$

with

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \eta \mathbf{d}_k\,, \tag{19}$$

we can rewrite the above equation as

$$\mathbf{d}_k^{\mathrm{T}} \nabla J(\mathbf{w}_{k+1}) = 0\,, \quad \text{i.e. } \mathbf{d}_k \perp \nabla J(\mathbf{w}_{k+1}) \tag{20}$$

But since $\mathbf{d}_{k+1} = -\nabla J(\mathbf{w}_{k+1})$, we have

$$\mathbf{d}_k^{\mathrm{T}} \mathbf{d}_{k+1} = 0\,, \quad \text{i.e. } \mathbf{d}_k \perp \mathbf{d}_{k+1}\,. \tag{21}$$

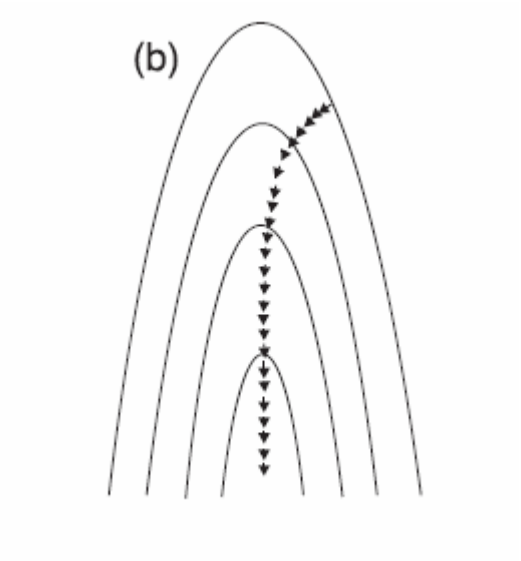
with the learning rate η replacing $\mathbf{H}^{-1}$, the inverse of the Hessian ,

We can reach the optimal w by descending along the negative gradient of J in (16) , hence the name gradient descent or steepest descent, as the negative gradient gives the direction of steepest descent.

But since $\mathbf{d}_{k+1} = -\nabla J(\mathbf{w}_{k+1})$, we have

$$\mathbf{d}_k^{\mathrm{T}} \mathbf{d}_{k+1} = 0\,, \quad \text{i.e. } \mathbf{d}_k \perp \mathbf{d}_{k+1}\,.$$ (22)

As the new direction $\mathbf{d}_{k+1}$ is orthogonal to the previous direction $\mathbf{d}_k$, this results in an inefficient zigzag path of descent (Fig.6.2)



One way to reduce the zigzag in the gradient descent scheme is to add '*momentum*' to the descent direction, so

$$\mathbf{d}_k = -\nabla J(\mathbf{w}_k) + \mu \mathbf{d}_{k-1} \,, \tag{23}$$

with μ the momentum parameter. Here the momentum or memory of $\mathbf{d}_{k-1}$ prevents the new direction $\mathbf{d}_k$ to be orthogonal to $\mathbf{d}_{k-1}$, thereby reducing the zigzag. The next estimate for the parameters in the momentum method is also given by (19).

6.4 Practical coding of a NN model in Matlab

```
% train model

net=init(net);
% if (16) is used, creating a network
net= newff(minmax(xtrain), [nhide, L],{'tansig' 'purelin'},'trainlm');
If (23) is applied
net= newff(minmax(xtrain),[nhide,L],{'tansig' 'purelin'},'trainbr');
net.trainParam.epochs = 100;  % maximum number of iterations
net.trainParam.goal   = 1E-4;  %  min cost function value
[net,tr]=train(net,xtrain,ytrain);
```

```
ytrain_nn = sim(net,xtrain);
ytest_nn =sim(net,xtest);
w1=net.iw{1,1};
b1=net.b{1};
w2=net.lw{2,1};
b2=net.b{2};
```

Note:

xtrain: [m,n], m is he # of input, n is the # of time points
ytrain: [L,n], L is the # of output.
xtest: [m, nnew], test period
nhide: number of hidden neurons

The trained model is save in variable 'net'. Function 'sim' is used to simulate/predict predictant using built NN network. 'net' is a structure, and contains lots of things, including W and bias parameters.