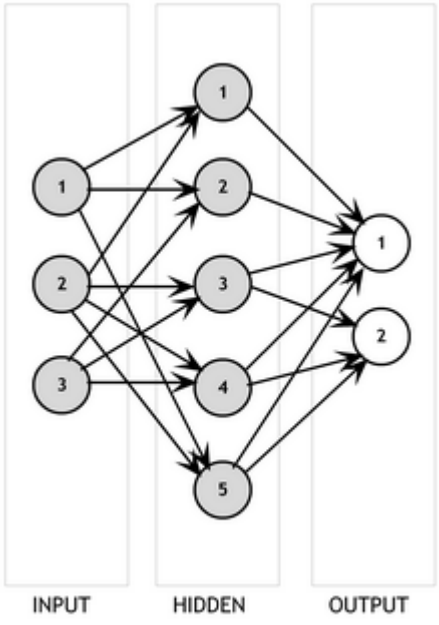
Introduction to artificial neural networks

Biological neurons, or nerve cells, receive multiple input stimuli, combine and modify the inputs in some way, and then transmit the result to other neurons. Most advanced animals (i.e. most life forms more complex than simple worms) have a very large number of such neurons, which in turn have vast numbers of interconnections with other neurons. This complex network structure is a biological neural network (BNN). A BNN is often divided into sub-structures that perform similar functions and the network as a whole is capable of performing very complex information processing.

The principal class of ANNs that we discuss are so-called *feedforward networks*. Of these, perhaps the most widely used is the *multi-level perceptron* (MLP) model (see below for more details). Other models we discuss below include *radial basis function neural networks* (RBFNNs, Section 8.3.2, Radial basis function networks, but also compare this with our earlier discussion of RBFs in Section 6.6.4, Radial basis and spline functions), and *self-organizing networks* (notably SOMs, see further Section 8.3.3, Self organizing networks).

A typical feedforward ANN consists of three or more inter-connected layers of nodes — an input layer, one or more hidden intermediate layers (often just 1), and an output layer (Figure 8-15). The arrows indicate the direction of information flow, feeding information forward from input to output. Note that there may be any number of nodes at each level of the network, and not all nodes need to be connected to every node in the next layer. For example, in Figure 8-15 hidden node H2 only receives input from input nodes I1 and I3, and only provides output to node O1. This arrangement can be seen as a directed graph, or as a rather complex-looking function mapping.

Figure 8-15 Simple 3-5-2 feedforward artificial neural network

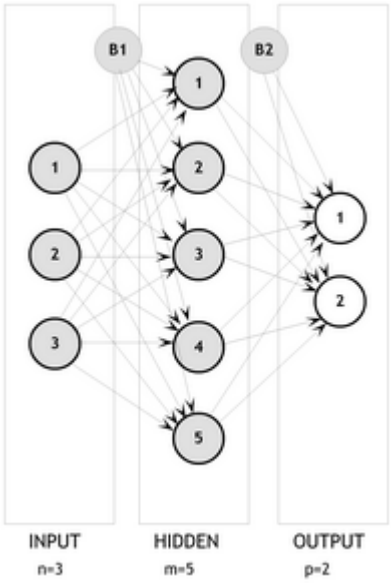


The connections between the input layer and hidden layer can be described using a weight matrix, **W**, where the row/column entries w_{ij} are positive or negative real-valued weights, or 0 if no connection exists. Likewise, the connections between the hidden layer and the output layer can also be viewed as a weight matrix, **Z** say, again consisting of a set of weights, z_{jk}. Positive weights in each case imply a reinforcement process associated with the source node or input, whilst negative weights correspond to inhibition.

Multi-level perceptrons (MLP)

When each node in the network is connected to every node in the next layer by a feedforward link, it is commonly referred to as a *multi-level perceptron* (MLP, Figure 8-16). The perceptron part of this description is a reference to its early development in a simpler form as a model of neuron triggering in the retina of an eye.

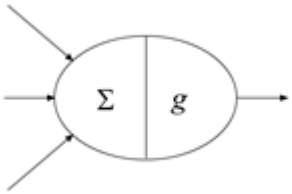
Figure 8-16 MLP 3-5-2 with bias nodes



By convention all nodes in the hidden layer and the output layer are also connected to a bias node. This feeds a constant input value, 1, into the set of weights, and acts in a similar manner to the constant term in a regression model. Bias node weights are treated in the same manner as per other node weights. In the MLP, with 1 bias node associated with each feedforward layer, *n* input nodes, one hidden layer with *m* hidden nodes and *p* output nodes, the **W** and **Z** weight matrices have dimensions (*n*+1)x(*m*) and (*m*+1)x(*p*). The MLP then has the architecture shown in Figure 8-16. The effective number of parameters, λ, for such a network is the sum of the number of feedforward connections (including the bias connections), so in this example λ=20+12=32.

Input data are weighted according to their w_{ij} values and combined (typically summed) in the hidden layer. This weighted sum is then modified by what is known as an *activation function*. This two-step process of summation of inputs and then modification of this sum by an activation function, ***g***, to create the output value can be illustrated at the node level as shown in Figure 8-17.

Figure 8-17 ANN hidden node structure



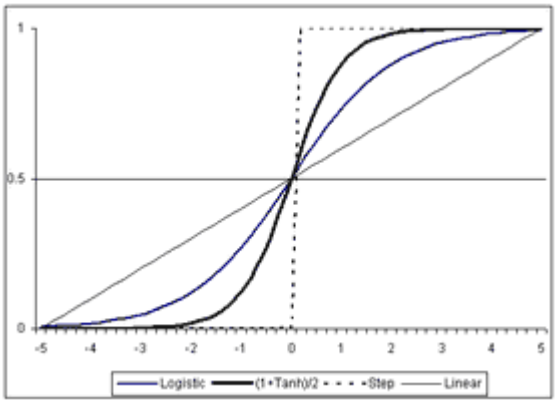
Various activation functions can be used, but the most commonly used in spatial analysis is the *logistic* or *sigmoid* function:

$$g(x) = \frac{1}{1+e^{-\beta x}}$$

where β is a slope parameter, typically set to β=1. With this function if *x*=0 the value output is 0.5. With *x* large and negative the output approximates 0, whilst for large positive *x* it approaches 1 (see Figure 8-18). For large positive β the slope of this function increases until it becomes essentially a step function from 0 to 1. The logistic function provides a model that approximates the actual response behavior of biological neurons to stimuli, but is primarily used owing to its convenient mathematical properties, as explained further below. Other commonly cited activation functions include simple step or threshold functions (of limited use in spatial analysis problems), the tanh() function (widely used) and simple linear activation (sometimes this name is used for the identity function, which involves no change). The basic tanh() function has a range [-1,1] but may be adjusted to provide a [0,1] range if required by taking (tanh()+1)/2 (which, of course, is exactly the same as the logistic function with β=2, so the two functions can be seen to be closely related). Experience suggests that for some problems the tanh() function, which produces

positive and negative values, can produce faster learning that logistic functions. Graphs of each of these functions are shown in Figure 8-18. Radial basis activation functions are described separately in Section 8.3.2, Radial basis function networks.

Figure 8-18 Sample activation functions



If we denote the set of inputs as a data matrix, **X**, and the set of outputs as a data matrix, **Y**, an artificial neural network is simply a mathematical operation that maps **X** to **Y**, i.e. f:**X**→**Y**. Some authorities describe this operation as a *network function*. Typically this mapping is a form of non-linear weighted sum — indeed non-linearity is an essential feature of ANNs applied to most problems. As noted above, the input data $\{x_i\}$ are modified twice, first by weighted summation and then by use of the activation function.

The same procedure applies to the next layer, where the hidden layer output values, h_j, are summed and optionally modified as per the input layer to produce the final results in the output layer, y_k:

$$h_j^* = \sum_i \left(x_i w_{ij} \right), \text{ and } h_j \leftarrow g_1\left(h_j^*\right), \text{ then}$$

$$y_k^* = \sum_j h_j z_{jk}, \text{ and finally } y_k \leftarrow g_2\left(y_k^*\right), \text{ hence}$$

$$y_k \leftarrow g_2\left(\sum_j g_1\left(\sum_i x_i w_{ij} \right) z_{jk} \right)$$

Although the same activation function, g(), may used for all nodes and layers, some implementations apply different activation functions at the output layer from those used for the hidden layers (e.g. linear or identity functions).

Learning and back-propagation for MLPs

Creating an artificial neural network, such as an MLP, does not of itself provide a direct means of solving a particular problem or class of problems. ANNs need to learn from sample datasets or subsets of a given large dataset before they can usefully be applied. Learning in the MLP case is a process of systematically adjusting the entries in the two weight matrices, **W** and **Z**. If you have one or more sets of control data, i.e. given a set of inputs, **i**, and you know what the set or vector of target outputs, **t**, should be, you can use this information to 'train' the network. This learning process is described as *supervised* because it implies we have prior knowledge about the nature of the solution and are supervising the training. Note that the control data may be divided into two main groups: *training data*, used to train the neural network, and *test data*, which the neural network has not previously seen, which can be used to assess the performance of the network. It is generally recommended that the control data be comprised of a substantial number of cases, in order to reasonably represent the underlying population data and to satisfactorily train and evaluate the ANN. It may be necessary to ensure that the range of values of training data variables includes the full range that may subsequently be used for validation and testing. By contrast, *unsupervised learning* has no vector of target outputs, **t**, only a vector of input data, **i**. In this situation training involves identifying patterns in the input vectors that generate consistent output vectors. This process is sometimes

described as *self organizing*. Both supervised and unsupervised training approaches have been applied in geospatial analysis.

Supervised learning is applied extensively in the fields of pattern recognition/classification and complex function approximation (non-linear regression). For example, suppose that the entries in the two weight matrices are initially set to random uniform [0,1] values. A set or vector of input (or training) data, **i**, will be transformed as the values propagate through the network to the output, giving a set or vector of output values, **o**, that will (almost certainly) not match the known correct results, **t**. Individual differences or 'errors' or *error signals* at the k^{th} output node are positive or negative values of the form:

$$e_k = (t_k - o_k)$$

Ideally we seek to minimize such errors by choosing an appropriate set of weights. To achieve this a convenient measure is to minimize the sum of squared errors across all output nodes, an approach which mirrors that of least squares regression. Minimization is achieved by differentiation of the sum of squared errors expression, equating this to zero, and using the result to determine the required adjustment of the initial weights. The *average* difference or error between the generated output results and the target results can be computed and tracked (e.g. graphed) using a measure that is appropriate for the problem in question. With function estimation problems it is common to compute the root mean sum of squared errors (RMSE):

$$RMSE = \sqrt{\frac{1}{N}\sum_{k=1}^{N}(t_k - o_k)^2}$$

This provides a standardized measure of the 'cost' of this solution, and the aim is to minimize this cost function, ideally reducing its value to 0 by adjusting the weights as noted above. In fact the *universal approximation theorem* states that every mapping of the form f:**X**→**Y** where **X** and **Y** are finite real-valued sets, can be approximated as closely as one wishes using a 3 layer MLP of the kind described above. The process of learning (adjusting the entries in the weight matrices) is typically achieved through a procedure called *back propagation* with *gradient descent*. The back propagation part of this refers to the process of applying a learning rule backwards through the network, step by step — so the first step is to change the **Z** matrix entries and then to change the **W** matrix entries. The learning rule essentially adjusts the values such that at time (or *epoch*) t+1:

$$\mathbf{Z}^{t+1} = \mathbf{Z}^t + (\Delta\mathbf{Z})^t$$

and then

$$\mathbf{W}^{t+1} = \mathbf{W}^t + (\Delta\mathbf{W})^t$$

The positive or negative increments to the entries in the matrices are based on their relative contributions to the positive or negative errors in the outputs. Thus if a particular weight z_{jk} is large in relation to other weights that contribute to output node k, it may need to be increased or decreased in proportion to the size and sign of the generated 'error' and the node value, h_j.

A simple learning rule might be of the general form:

$$z_{jk}^{t+1} = z_{jk}^t + \eta(t_k - o_k)h_j, \text{ or}$$
$$z_{jk}^{t+1} = z_{jk}^t + \eta e_k h_j$$

where η is a small positive constant (e.g. η<0.1) known as the *learning rate*. The initial learning rate value is often reduced during the course of iteration so that the learning rate decreases over time. This corresponds to an initial broad step learning process that makes large adjustments to the weights and later makes smaller adjustments in order to fine-tune the results (sometimes referred to as *stochastic approximation*). In practice the learning rule provided above is not appropriate for MLPs, because it has not taken into account the use of the *activation function*. In order to incorporate this component the error

signal needs to be weighted by the derivative of the activation function. A particularly nice feature of the logistic activation function:

$$g(x) = \frac{1}{1+e^{-x}}$$

is that its derivative is simply:

$$g'(x) = g(x)[1-g(x)]$$

and instead of using e_k in the learning rule above, $g'e_k$ is used, i.e.:

$$z_{jk}^{t+1} = z_{jk}^{t} + \eta o_j \left(1-o_j\right) e_k h_j$$

The $g(x)$=tanh(x) activation function also has a very convenient derivative:

$$g'(x) = 1-g^2(x)$$

Essentially the same procedure is applied when computing the adjustments to the weights in the **W** matrix. However, the adjustment is slightly more complex because the error signal is now not based on the difference between output values and expected output values, but between hidden values and the back-propagated (adjusted) hidden values. These back-propagation steps correspond to a gradient descent approach to finding a local minimum of the mean squared error — see Abdi (1994) for a more detailed description and derivation of these formulas.

Having adjusted the weights for both levels, the forward propagation process takes place once more and the cost function is computed — this should now be less than or equal to the cost function for the previous time period. The process continues until reduction in the cost function ceases and/or a preset maximum number of iterations (e.g. typically more than 100) have been reached. Other variants of stopping criteria may also be provided, and separate criteria will apply to validation data (e.g. all data to be within x% of their true values). A number of additional observations should be made at this point:

(i) **local versus global minimization**: global minimization of the cost function is not guaranteed — for this reason some authors and software packages apply global minimization techniques such as simulated annealing and genetic algorithms rather than/as well as simple gradient descent in order to attempt to improve overall solutions. Faster convergence may also be achieved using more sophisticated local gradient search procedures, such as conjugate gradient methods (see Walsh, 1975, for a brief discussion of conjugate gradients).

(ii) **initialization and selection order**: initialization of weights is usually performed by random selection from a uniform distribution over a finite range. The particular initial choice of weights may affect the final outcome, depending on the nature of the problem and dataset, hence multiple initializations (e.g. 5) may be advisable in order to examine and manage this issue. Furthermore, the order in which individual weights are selected for updating can affect the process and it is common practice to select weights for updating at random until all have been updated in a given layer. Likewise, during training, it is common practice to select input vectors from the training set at random, since the selection order can affect the final trained set of weights.

(iii) **data normalization and coding**: the input dataset is generally transformed and/or pre-normalized such that all input data lie in the range [0,1]. With positive real-valued data a common procedure is to divide all input values by the maximum in the sample. Note that no distributional assumptions are made about the input data. The output dataset, if it involves distinct categories, e.g. A, B, C, D… is typically coded such that each category (output node) is assigned a binary 1 with the remainder being assigned binary 0. On completion the input and output data can be converted back to more familiar units/descriptions.

(iv) **momentum**: the number of iterations required and the quality of solutions can often be improved by amending the learning rule through the use of a *momentum factor*. This is a small multiple of the

adjustment used in the preceding iteration, time t, effectively prolonging this adjustment, i.e. giving it *momentum* — the adjustments are thus of the form:

$$\Delta \mathbf{Z}^{t+1} \leftarrow \Delta \mathbf{Z}^{t+1} + \alpha \Delta \mathbf{Z}^{t} \text{ and } \Delta \mathbf{W}^{t+1} \leftarrow \Delta \mathbf{W}^{t+1} + \alpha \Delta \mathbf{W}^{t}$$

(v) **model design, over-fitting and over-training:** the number of nodes in the input and output layers are determined by the problem and how it is modeled. In most ANNs nodes are fully connected between the layers, so connection choice is not normally a requirement. The number of hidden layer units, m, can be specified manually (e.g. initially set to the same as the number of input nodes, n, or some combination of n and the number of output nodes, p). This number, m, can then be systematically altered until results of the best 'quality' have been obtained. Quality is usually determined by some performance measure or multiple measures applied initially to the training data, and then examined for other datasets (see further, below). Quality is also measured in terms of model simplicity (number of parameters implied in the model structure). Some software packages attempt to compute suitable structures automatically. For example, a standard rule may be applied for selecting the number of hidden nodes, such as:

$$m = \text{int}\left(\sqrt{np}\right)$$

where int() is the integer part function. Hence with 3 inputs and 2 outputs, this rule would give m=2. This is the default used within Idrisi, for example. Other rules of thumb suggested by various authors include m=(n+p)/2, m<2n and m=2(n+p)/3. For problems involving simple function approximation (see the worked examples in Section 8.3.1.3) the number of hidden layer units is typically set to between 3 and 10 where there is a single input and single output node.

There is a risk that neural network models perform extremely well on training data but may not perform as well on unseen data as simpler ANN models. This problem, which falls into the general category of *over-training*, often occurs when over-complex systems have been specified. There are a number of procedures for attempting to minimize over-training. One approach is to divide the control data set into three parts rather than two (a form of split-sample validation): a training set; a validation set; and a test set. With this approach the validation set is used to identify when training should be stopped, based on the leveling out of performance measures for the validation set, having first identified the neural network model which best fits the data with the minimum number of parameters — see, for example, Fischer and Gopal (1994) for a discussion of this issue in connection with spatial interaction modeling, a paper that has been republished in Fischer (2006) together with several others that address the same problem and dataset. In their model they selected an n-m-p model with n=3, p=1 (problem defined) and values of m from 5 to 50 in steps of 5, with m=30 being their preferred solution. This equates to a model with 151 parameters as compared with 3 parameters for the conventional spatial interaction model, suggesting that there has to be a marked improvement achieved by such models and/or other benefits (e.g. robustness, dynamic adjustment) if they are to be used in preference to conventional alternatives. An alternative approach is to apply s-fold cross-validation. In this case the data are divided into s approximately equal-sized subsets. The NN is then trained s times, on each occasion leaving out one of the subsets which is then used to compute the validation statistic of interest. This approach is reportedly an improvement on split-sample methods for smaller datasets, but is more computationally intensive. A third approach involves the use of a technique known as bootstrapping, in which random samples are taken (with replacement) rather than pre-defined or random subsets — see Efron (1982) and Efron and Tibshirani (1997) for more details.

(vi) **forecasting**: the methods described above can be applied to time-series forecasting — indeed, this is an application area that has attracted considerable interest, notably in the financial sector. There are now a large number of software products, from Excel add-ins to multi-function integrated packages, which are specifically designed to provide 'intelligent' forecasting. For such problems it is not always necessary to have multiple inputs — for example, river flow levels can often be quite accurately predicted using existing records for daily or weekly flows over several years. However, multiple inputs, such as the inclusion of relevant correlated information (where available) such as rainfall data, catchment information, flows at

related measuring points etc., may all serve to improve the model robustness and its ability to be more widely applied.

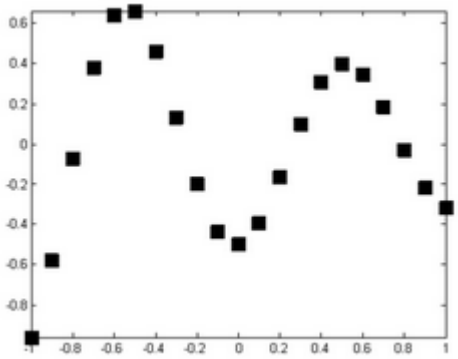
The preceding subsections have provided a description of MLPs, but their operation is not particular obvious from this information — indeed, such networks are often described as black boxes, where the inner workings are hidden from the observer, who merely provides inputs and views outputs. For this reason we now provide a small number of examples of their application.

MLP Example 1: Function approximation

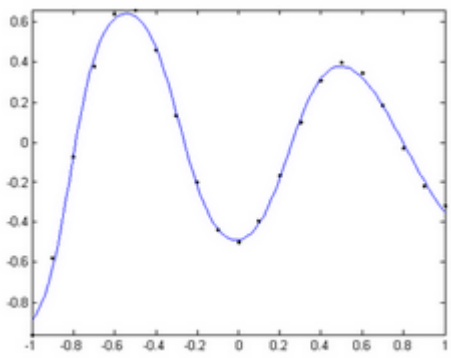
In this example we suppose that we have a total of 21 measured values of a response (or dependent) variable, *y*, for a set of equally spaced values of an independent variable, *x* (this example is based on sample MATLAB code produced by Roger Jang, Computer Science Department, Tsing Hua University, Taiwan). The data suggest a highly non-linear relationship (Figure 8-19A). This dataset provides the training information in the form of (*x*,*y*) pairs for the model, for which there is one input node and one output node, plus a bias node for each layer. The activation function used in this case was the tanh() function, with standard gradient descent back-propagation. A learning rate of 0.02 was used with a momentum factor of 0.8 — using a higher learning rate, for example, produced far less stable RMSE behavior (Figure 8-20A and B). The stopping criteria were set to 1000 epochs (iterations) or RMSE<0.01. Weights were all initialized with a random uniform value in the range [-0.5,0.5]. Choice of the number of hidden nodes was made by varying the values between 3 and 10 and examining the fit of the model and the RMSE curve. The fitted solution shown in Figure 8-19B was based on 4 hidden nodes, this being the smallest value that provided a satisfactory fit to the sample data.

Figure 8-19A MLP: Test data and fitted model

A. Test response data



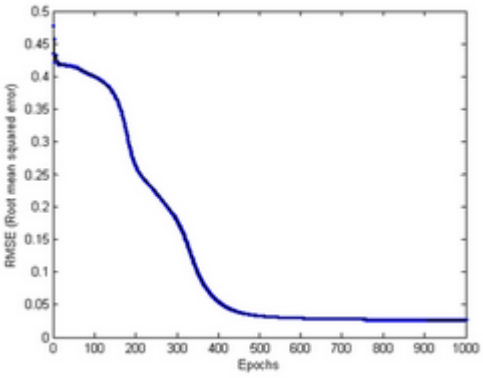
b. Fitted solution curve



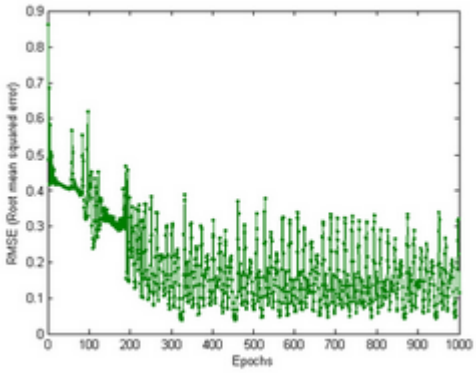
As might be expected, running the same dataset with a different MLP algorithm requires different settings to achieve the same or similar results. For example, the problem described above was also tested using software (Netlab) that uses a different (generally faster) training procedure, known as *scaled conjugate gradients*. In this model a tanh() activation function was again used for the hidden nodes whilst a linear function was used for the output nodes. Weight matrices were initialized with scaled random values based on a Normal distribution and a learning rate of 0.02 was again used, but this program variant requires no momentum parameter. It yielded slightly improved results to those above in around 150 iterations.

Figure 8-20 MLP: RMSE curves

A. RMSE vs. epochs, learning rate=0.02



B. RMSE for learning rate=0.05



It is immediately apparent from this discussion that this ANN modeling process is effectively a form of nonlinear regression, similar to iterative least squares regression (ILSR), previously mentioned in connection with Geographically Weighted Regression in Section 5.6.3. It is also clear that the process requires a level of interaction and experimentation with the software tools, in order to determine the ideal parameter and node settings for a given function estimation problem.

MLP Example 2: Landcover change modeling (LCM)

In this example we use the LCM functionality included within the Idrisi GIS software (and also available for ArcGIS). This incorporates an MLP component for modeling the drivers behind landcover transitions between 1986 and 1994, as an input to specific predictions of change. The study area covers roughly 15km x 15km in part of Bolivia known as Chiquitania. It lies in the border between the Amazonian rain forest and deciduous dryland tropical forest. Existing data includes landcover in 1986 (Figure 8-21), landcover in 1994 and various vector layers (e.g. roads, rivers etc.). Fuller details of this example are provided in the Idrisi documentation.

All image/raster files used in this model are co-registered 987x927 pixels or cells, where one pixel=150mx150m. Four of the seven input rasters are generated from vector data using a simple distance transform — i.e. each pixel provides the distance to the closest object in the underlying vector set. These include urban areas, roads, streams and areas subject to anthropogenic disturbance (Figure 8-22). The underlying dataset used in this example is from the Conservation International's Center for Applied Biodiversity Science, Museo Noel Kempff Mercado, Bolivia, and supplied with the Idrisi software installation. Two of the input rasters provided the elevation and slope data for the study area, and the last of the 7 inputs is another computed raster, this time providing a measure of the likelihood of individual raster cells changing landcover based on the frequency patterns found for those cells that were subject to change in the period 1986-94.Our principal interest is in the application of the MLP modeling and classification process (Figure 8-23) within LCM. In this instance the MLP network topology is of the form 7-7-8 (plus bias), i.e. 7 input nodes (in this instance, Bands 1, 2, 3…), all of which are real-valued raster files, 1 hidden layer with 7 nodes, and 8 output nodes. The output nodes are the 4 main land-use transitions (classes) that are taking place in the region and 4 'persistence classes' — the latter represent raster cells that might have changed between 1986 and 1994, but did not. The authors state that the inclusion of these additional output nodes assists the MLP training process. Each of the 4 main transitions observed were to the class 'anthropogenic disturbance' (class 8) and transferred from the identified classes: (i) woodland savannah; (ii) Amazonian rain forest; (iii) savannah; or (iv) deciduous forests. The MLP training site specification (Figure 8-23, center left) automatically picks a random set of cells (pixels) — i.e. training sites — to be used for training the model and a similar number of sites for testing the model. The number is chosen by examining the smallest transition in the group of four that are the principal subject of the modeling process.

As can be seen from Figure 8-23, key training parameters for the MLP modeling process are specified via this form. These include: using a dynamic learning rate, which commences with a value of 0.005 and reduces to 0.0001 — this reduction takes place over the 5000 iterations or epochs of the model, but in this example most of the training was achieved within the first 1000 epochs (see the RMSE graph in Figure 8-23); applying a momentum factor — a value of 0.5 was specified in this example; and specifying the slope parameter — the activation function used within Idrisi is the logistic or sigmoidal function, and here was applied with slope parameter (constant *a* in Figure 8-23) of 1.

Normalization of the input raster files is not specified within the Idrisi documentation, but is assumed to be carried out by default. Three stopping criteria can be specified, although here the training stops after the maximum number of iterations has been reached. Notice that an accuracy rate of 81.5% of transitions has been obtained with this model (Figure 8-23, running statistics panel). The authors suggest that 75% is good and 80%+ is a preferable target. This figure represents the number of correctly predicted transitions for the test pixels.

Figure 8-21 Land cover, 1986, Chiquitania

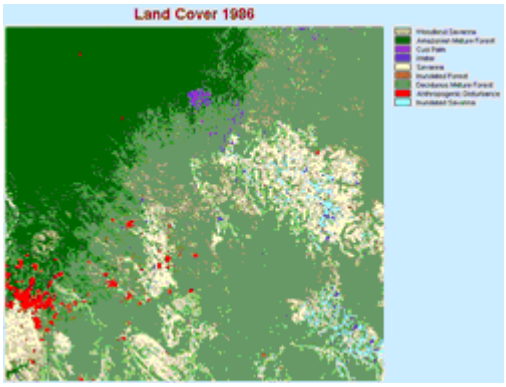


Figure 8-22 Distance raster (meters), anthropogenic disturbance, Chiquitania

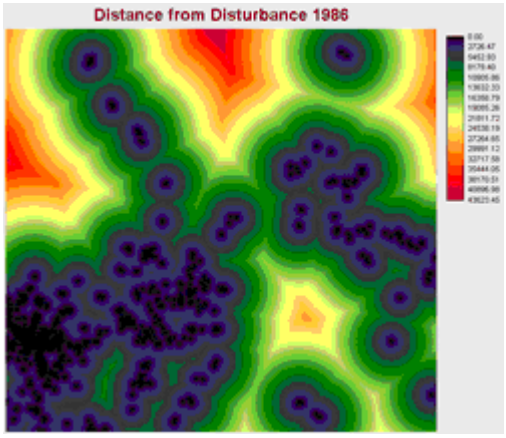
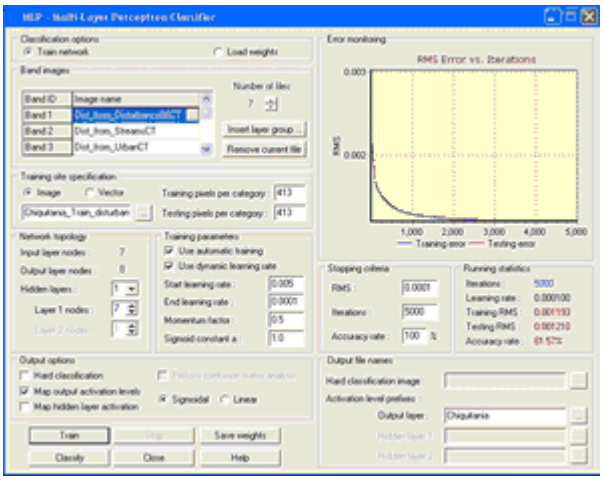


Figure 8-23 MLP Classifier — Idrisi



A set of four transition potential maps can then be generated using this information. One such map is shown in Figure 8-24. The next stage in the process is to use the transition potentials map, in conjunction with

Markov Chain analysis (see further, CATMOG 1), to predict landcover in the year 2000, for which actual landcover information is known. This enables the quality of the MLP modeling and subsequent predictive process to be evaluated (see Idrisi documentation for more details of these steps in the modeling process).

At this point the MLP has created two weight matrices, **W** and **Z**, which may be saved for reference and optionally re-loaded for future use (see example, Table 8-6). These weights provide what the authors regard as an acceptable level of prediction of 'transition potential'. There are (7+1)x7 input weights to the 7 hidden layer nodes (**W** matrix) plus (7+1)x8 hidden layer to output nodes weights (**Z** matrix), i.e. 120 parameters for this model.

Figure 8-24 Transition potential map

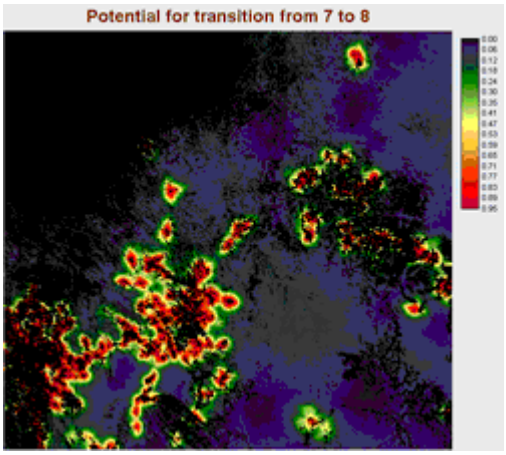


Table 8-6 W weights matrix, Chiquitania MLP model

-1.1017	-0.5180	-0.1645	-3.0337	-1.6198	-0.1066	-2.1134	0.2422
-6.5289	1.4412	-3.6655	-3.6952	-0.3428	1.7637	-2.9166	-0.2034
1.7461	-0.1886	1.6998	-2.0887	3.0947	0.0202	-7.1687	0.1928
-1.0112	-0.4277	1.3041	-0.7028	2.1360	0.4187	5.6591	0.1146
15.3872	0.1111	-0.4215	14.3066	0.0078	0.0681	-0.0071	-0.1195
2.9165	-0.3409	2.9754	5.5850	4.1987	0.0772	-26.8881	-0.2493
1.6520	0.0149	-0.2517	0.2964	-1.0419	-0.0475	6.5108	-0.1082

MLP Example 3: Spatial interaction modeling

In this example we consider a situation in which forecasts of shopping trips in region B are sought, based on data collected in region A (possibly a subset of B), or where regions A and B are the same but studied at different points in time. The data available for region A relates to attributes of source zones or origins (*O*), destination zones (*D*), a measure of the distance between each paired origin (*i*) and destination (*j*) zone, and the recorded average number of shopping trips (T_{ij}) per trading day. Here we assume a generalized trip model of the form:

$$T_{ij}=f(O_i,D_j,d_{ij})$$

where *f*() is an unknown function of the attribute data and the separation measure. For example, we may have data for origin zones providing the population numbers (POPUL) and the average household incomes (HOUSINC) which are demand driver or 'push' measures; and for destination zones we may have the retail floor space in thousands of square meters (RETAILSP) and the number of parking spaces in thousands (PARKINGSP) which are attraction or 'pull' measures. In addition, we can compute a measure of zone separation (DISTANCE), for example as Euclidean distance, network distance, estimated travel time, or some other separation measure. In total these data provide five explanatory variables (input variables) with which we seek to explain one observed variable (output), actual shopping trip levels. As mentioned previously, Fischer (2006) has studied this class of problem over many years, and has shown that ANN models can provide an excellent fit to observed data and provide good quality generalizations from such data.

A simple ANN model for the problem we have described is a 5-*m*-1 MLP, which we train using the observed data (or more generally a subset of this) and validate and/or test using the remainder of the observed data. Because the data can be seen as a set of rows, with columns corresponding to the 5 input variables and 1 output variable, they are ideally suited to simple text file, spreadsheet or database storage. Indeed, such data are similar to those found in financial and commodity trading, where a range of index, price and volume data by time of day or across days is input and closing prices, for example, are the output. ANNs have found considerable favor in this field and have resulted in the availability of a large number of desktop-based software packages and add-ins (e.g. Excel utilities) targeted specifically at this community of users. For illustrative purposes we used one such package to import a spreadsheet of simulated origin, destination and distance data (inputs) and trip levels (output) that we have generated using the specific variables described above. We have then requested the software to model the data (Figure 8-25). In this diagram thicker lines identify stronger weights, green lines are positive weights, and red lines negative weights. Bias nodes are not separately illustrated but are including in the modeling. This particular software package uses logistic activation and gradient-based back-propagation (as described earlier) to train the network. Data are automatically normalized by the software for processing and then converted back when generating predictions. The software also estimates the preferred number of hidden layer nodes and can automatically increment these from an initial minimum value in order to obtain the best possible result. Between 1 and 3 hidden layers are catered for, but most models of this type can be accommodated with the default one hidden-layer model.

Rows from the input spreadsheet can be assigned explicitly as being for training or validation, or random samples of those provided may be used for validation. Other, non-commercial packages could equally well be used. The latter includes a range of ANN facilities and a simple user interface and data file structure. Assuming the training process yields a high quality network (i.e. excellent in terms of its fit to the validation data and any additional test data provided) its use for prediction may then be considered and compared with alternative models (e.g. gravity-based models with calibrated parameters) applied to the same datasets. However, this initial MLP modeling process did not yield good results based on fit to validation data. The process was therefore re-examined, this time log transforming the data (input and output) before commencing, adjusting the learning rate and momentum, and re-running the network modeling. The network on this occasion provided an improved fit to validation data (60% within 10% of known values — as applied to the log transformed data) using 8 hidden nodes rather than 5, as shown in Figure 8-26.

Figure 8-25 MLP trip distribution model 1

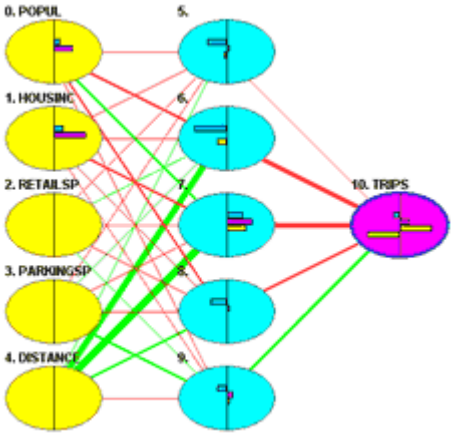
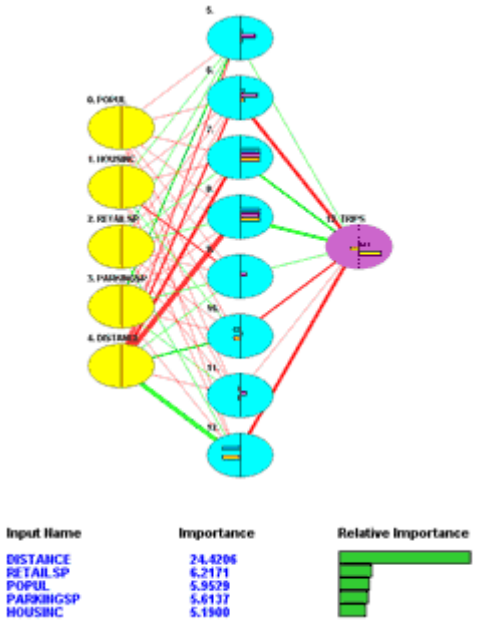


Figure 8-26 MLP trip distribution model 2



The bars in the node ovals indicate the positive and negative component levels, from top to bottom: net input; activation; bias; error

The effective number of model parameters therefore increased, which explains part of the observed improvement. In this example we have also shown the relative importance computed by the model of the input variables. As can be seen, distance is the dominant factor, with the remaining four variables (two push and two pull factors) being of approximately equal significance.

This rather simplistic analysis highlights several issues. First, it is clear that ANN modeling is a powerful and valuable tool for modeling a wide range of data types and problems. Second, each application area presents different challenges, and application-specific modeling (e.g. selection of training, validation and test data sets; transformation and normalization of input data; network design and control parameter settings) is an important aspect of successful modeling. Finally, ANN modeling may reveal structures in the data that point the way to simpler and more general models.