PROCEEDINGS, Thirty-Fourth Workshop on Geothermal Reservoir Engineering
Stanford University, Stanford, California, February 9-11, 2009
SGP-TR-187

AN ARTIFICIAL NEURAL NETWORK MODEL FOR NA/K GEOTHERMOMETER

+Genco Serpen, Yildiray Palabiyik* and Umran Serpen*

*ITU, Petroleum and Natural Gas Eng. Dept.
Maslak
Istanbul, 34469, Turkey
+Koza Evleri 3/35, 4. Levent/Istanbul, Turkey.
e-mail: serpen@itu.edu.tr

ABSTRACT

In this study, a brief explanation is first given on solute Na/K geothermometers developed until now, and a new Na/K geothermometer model is presented after using world geothermal database (n=212) to the ANN as a training set and another database (n=112) as a test set. In this model Na and K values are treated as input values and geothermometer temperatures as output values. A multilayer feed-forward neural network is trained using a genetic algorithm for optimizing hidden layer neuron weights and linear regression for optimizing output neuron weights. The model is successfully evaluated and compared with actual deep temperature measurements to avoid training bias.

INTRODUCTION

Artificial neural networks have lately been popular because of their applicability and ability to learn non-linear models, and simple implementation. New artificial neural network software is developed for modeling geothermal energy related problems. The code is first used for modeling Na/K geothermometer. Several versions of Na/K geothermometer have previously been studied using ANN by Can (2002), Bayram (2001) and Diaz-Gonzalez et al. (2008). In all these studies multilayer networks are trained by back-propagation algorithms. The new ANN software utilizes a genetic algorithm for optimizing neuron weights instead of back-propagation of errors. The use of a genetic algorithm is expected to reduce the probability of convergence to local minima of the network's error function occurring in back-propagation algorithms.

ANN SOFTWARE

An artificial neural network is an information processing system that shares characteristics with biological neural networks. Artificial neural networks have been developed as generalizations of mathematical models of human cognition and neural biology. Neural nets can be applied to a wide variety of problems, such as storing or recalling data or patterns, classifying patterns, noise reduction, function approximation, performing general mappings from input patterns to output patterns, finding solutions to constrained optimization problems, noise reduction, function approximation and time series prediction.

A neural net consists of a large number of simple processing elements called *neurons* or *nodes*. Each neuron is connected to other neurons by means of directed links, each with an associated weight that multiplies the signal transmitted. Each neuron is characterized by an *activation function* to its net input (sum of its weighted input signals) which determines its output signal, called *activation* or *activity level*.

A neural network is characterized by its pattern of connection between the neurons (called its *architecture*) and its method of determining the weights on the connections (called its *training* or *learning algorithm*). This text will focus on *genetic* and *error backpropagation* algorithms for training *multilayer feedforward network* architectures.

Multilayer Feedforward Networks

A multilayer feedforward network consists of a set of neurons that are logically arranged into two or more layers. There is an input layer and an output layer, each containing at least one neuron. Neurons in the input layer are hypothetical in that they do not themselves have any input, and they do no processing. Their activation is defined by the network input. There are usually one or more hidden layers sandwiched between the input and output layers. The term "feedforward" means that information flows in one direction only. The inputs to the neurons in each layer come exclusively from the outputs of neurons in previous layers and outputs from these neurons pass exclusively to neurons in following layers (Masters, 1993). The output units and the hidden units may have biases. These bias terms act like weights on connections from units whose output is always 1.

A single layer net is severely limited in the mappings it can learn, a multi-layer net (with one or more hidden layers) can learn any continuous mapping to an arbitrary accuracy (Fausett, 1994). More than one hidden layer may be beneficial (at the expense of more difficult training due to the dramatic increase of local minima of the function that is being optimized) for some applications such as learning a function having discontinuities. A multilayer network with two hidden layers (the *Z* and *ZZ* units) is shown in Figure.

The activation function of a neuron is usually a nonlinear function that, when applied to the net input of a neuron, determines the output of that neuron. The activation function is generally expected to be continuous, differentiable, have an unlimited domain and approach a finite maximum and minimum asymptotically. Usually the activation function's range is limited between (0, 1) and in some cases (-1, 1). For training with backpropagation of errors it is desirable for computational efficiency that the function's derivative is easy to compute and that the value of the derivative can be expressed in terms of the value of the function (Fausett, 1994). One of the most commonly used activation functions (also used in the Neuro-Gene application) is the binary sigmoid function, or *logistic* function which is defined as

$$f(x) = 1 / (1 + e^{-x}),$$

with

$$f'(x) = f(x)[1 - f(x)].$$

Sometimes using nonlinear activation functions for all neurons may be detrimental. The squashing function used in the output layer may cause compression of extreme values. This may be avoided by using any linear function, such as the identity function $f(x) = x$, for the output layer neurons. The biggest advantage of using a linear output layer is that using a regression technique for the output layer will produce optimal output weights. One potentially serious drawback to linear activation functions concerns noise immunity. Although the squashing functions in the hidden layer provide a fair degree of buffering, the extra amount provided at the output layer can sometimes be valuable (Masters, 1993).

Choosing an appropriate number of hidden neurons is extremely important. Using too few will starve the network of the resources it needs to solve the problem. Using too many will increase the training time. Also an excessive number of hidden neurons may cause a problem called *overfitting*. The network will have so much information processing capability that it will learn insignificant aspects of the training set, aspects that are irrelevant to that of the general population. The purpose of training the neural net is to achieve a balance between the ability to respond correctly to the input patterns that are used fot training (memorization) and the ability to give reasonably good responses to input that is similar, but not identical, to that used in training (generalization). A network with an excessive number of neurons may lose its ability to generalize and can perform poorly when called upon to work the general population even though it achieved excellent results with training sample data. Thus, it is imperative that the absolute minimum numbers of hidden neurons which will perform adequately are used (Masters, 1993).

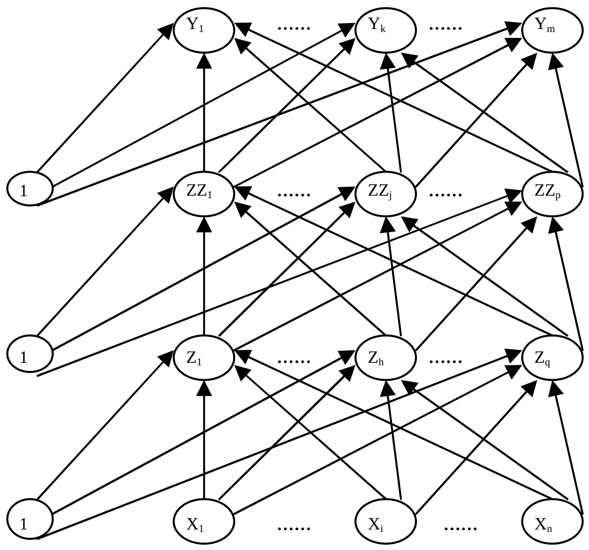


Figure 1: Multilayer Feedforward Neural Network with Two Hidden Layers.

One rough guideline for choosing the number of hidden neurons in many problems is the *geometric pyramid* rule. It states that, for many practical networks, the number of neurons follows a pyramid shape, with the number decreasing from the input towards the output. This guideline may underestimate the number of neurons required in cases where there are very few inputs and outputs and the problem is very complex. A more rigorous approach is to start training and testing with a small number of neurons and increase the number until the error is acceptably small or there is no significant improvement (Masters, 1993).

<u>Training by Backpropagation of Errors</u>

Training a network by backpropagation involves three stages: the feedforward of the input training pattern, the backpropagation of the associated error and the adjustment of the weights.

During feedforward , an input pattern is presented to the network and the response of the network is obtained by computing the activation of every neuron at the first hidden layer and broadcasting that signal to successive layers. For a network with *k* outputs, the output neuron activation y_k is compared against the training sample data t_k to determine the associated error for that pattern. Based on this error, the factor δ_k is calculated as

$$\delta_k = (t_k - y_k)\, f\,‘(y\_input_k) ,$$

where y_input_k is the net weighted sum of the input signals to output neuron Y_k. This value is used to distribute information on the error at output unit Y_k back to all units in the next lower layer. It also stored for later update of output neuron weights (Fausett, 1994). For a hidden layer with j neurons, the factor δ_j is computed similarly as

$$\delta_j = f\,‘(zz\_input_j) \sum_k \delta_k w_{j\,k} ,$$

where zz_input_j is the net weighted sum of the input signals to hidden neuron ZZ_j and w_{jk} represents the weight values associated with links between hidden neuron ZZ_j and all output neurons (Fausett, 1994). This value is then used to distribute the information on the error back to all units in the previous hidden layer, if there are any. It also stored for later update of the final hidden layer neuron weights. If there are multiple hidden layers, the factor δ_h for hidden neuron Z_h of those layers can be computed in a similar fashion by using the δ values of the next upper hidden layer neurons and weight values associated with links between neuron Z_h and all neurons of the next upper hidden layer.

The weights can be updated after each training pattern is presented but in a more popular variation of the training algorithm the weights are updated after one cycle through the entire set of training vectors (an *epoch*). For each pattern the weight updates are accumulated in a weight correction term. The delta weight correction terms for the output units are calculated as

$\Delta w_{j\,k} = \alpha\, \delta_k\, zz_j$ (for weights on links to output layer neuron k),
$\Delta w_{0\,k} = \alpha\, \delta_k$ (bias correction term for output layer neuron k),

where α is a user defined constant *learning rate* and zz_j is the activation of the neurons in the next lower hidden layer (Fausett, 1994). The weight correction terms are accumulated over an entire epoch and the new weights are calculated by adding the weight correction terms to the old weights at the end of the epoch. The delta weight correction terms for hidden layer units can be similarly calculated as

$\Delta v_{h\,j} = \alpha\, \delta_j\, z_h$ (for weights on links to hidden layer neuron j),
$\Delta v_{0\,j} = \alpha\, \delta_j$ (bias correction term for hidden layer neuron j),

where z_h is the activation of the neurons in the next lower hidden layer (Fausett, 1994). If there is no lower hidden layer, then the input pattern x_i should be used as activations from the previous layer in order to obtain the weight correction term $\Delta u_{ih} = \alpha\, \delta_h\, x_i$ for weights on links to the first hidden layer neuron h.

Drawbacks of Training By Backpropagation of Errors

The mathematical basis for the backpropagation algorithm is the optimization technique known as *gradient descent*. The gradient of a function gives the direction in which the function increases more rapidly, the negative of the gradient gives the direction in which the function decreases more rapidly. For backpropagation, the function is the network’s error for the training set and the optimized variables are the weights of the network. The exact distance to step in the negative gradient, often called the *learning rate*, can be critical. If the distance is too small, convergence will be excessively slow. If it is too large, the function will jump wildly and never converge.

There are two very serious flaws in the above method. First is the fact that the gradient is an extremely local pointer to optimal function change. Even a tiny distance away the gradient may point in a dramatically different direction. This can dramatically increase the search time. The second problem is that it is difficult to know in advance how far to step in the negative gradient direction (Masters, 1993).

Some of these problems have been addressed in variations of the algorithm, but they fail to address the problem of escaping false minima. It is surpisingly easy for gradient algorithms to get stuck in local minima when learning feedforward network network weights. Even tiny problems can sport local minima far inferior to global minima. Network error functions have broad expanses of plains that are nearly flat, but do definitely slope downward to a distant minimum. When a gradient descent algorithm finds itself in such an area, it will have trouble if it assumes that it is at a minimum because the gradient is very small (Masters, 1993).

A genetic algorithm has been chosen as the network training method for the NeuroGene application as it facilitates a much wider search to the global minimum and offers a fair degree of robustness.

<u>Genetic Algorithms</u>

Genetic algorithms are adaptive methods which may be used to solve search and optimization problems. They are based on the genetic processes of biological organisms. In nature evolution is driven by survival of the fittest. Weak individuals die before reproducing, while stronger ones live longer and bear more offspring, who often inherit the qualities that enabled their parents to survive. Artificial genetic optimization operates in a similar manner. The basic principles of genetic algorithms were first laid down rigorously by Holland (1975).

Genetic algorithms work with a *population of individuals*, each representing a possible solution to a given problem. The parameters of the function to be optimized are encoded as *genes* in a *chromosome*. Each individual is assigned a *fitness score* according to how good a solution to the problem it is. The highly-fit individuals are given opportunities to *reproduce*, by *cross breeding* with other individuals in the population. This produces new individuals as *offspring*, which share some features taken from each *parent*. The least fit members of the population are less likely to get selected for reproduction, and so *die out*.

A whole new population of possible solutions is thus produced by selecting the best individuals from the current "generation", and mating them to produce a new set of individuals. This new generation contains a higher proportion of the characteristics possessed by the good members of the previous generation. In this way, over many generations, good characteristics are spread throughout the population. By favoring the mating of the more fit individuals, the most promising areas of the search space are explored. If the genetic algorithm has been designed well, the population will *converge* to an optimal solution to the problem.

A genetic algorithm belongs to the class of methods known as *weak methods* because it makes relatively few assumptions about the problem that is being solved. Genetic algorithms are often described as a global search method that does not use gradient information. Thus, non-differentiable functions as well as functions with multiple local optima represent classes of problems to which genetic algorithms might be applied. Genetic algorithms, as a weak method, are robust but very general. They are not guaranteed to find the global optimum solution to a problem, but they are generally good at finding "acceptably good" solutions to problems "acceptably quickly". Where specialized techniques exist for solving particular problems, they are likely to outperform genetic algorithms in both speed and accuracy of the final result. The basic mechanism of a genetic algorithm is so robust that, within fairly wide margins, parameter settings are not critical.

Both genetic algorithms and neural nets are adaptive, learn, can deal with highly nonlinear models and noisy data and are robust, "weak" random search methods. They do not need gradient information or smooth functions. For practical purposes they appear to work best in combination: neural nets can be used as the prime modeling tool, with a genetic algorithm used to optimize the network parameters.

Coding

Before a genetic algorithm can be run, a suitable coding (or representation) for the problem must be devised. It is assumed that a potential solution to a problem may be represented as a set of parameters, such as the weight parameters that optimize a neural network. These parameters (known as genes) are joined together to form a string of values often referred to as a chromosome. For example, in order to maximize a function of three variables, each variable may be represented by a 10-bit binary number. The chromosome would therefore contain three genes, and consist of 30 binary digits. The explicit genetic structure represented by a particular chromosome is referred to as a genotype. The genotype contains the information required to construct an organism which is referred to as the phenotype. The phenotype is the physical expression of the genotype.

For phenotypes that express numerical values, binary encoding will produce poor results. For example, the 8 bit binary encoding for the number 127 is 01111111, while 128 is encoded as 10000000. A unit change in the number required all eight bits to change. Binary encoding is unsuitable for genetic expression because small changes in numerical values require large changes in the genotype. To alleviate this problem, a new coding system, called Gray code after its inventor, was devised. In this system, a unit change in the number causes exactly one bit to change (Masters, 1993).

Evaluation

The *evaluation function*, or *objective function*, provides a measure of performance with respect to a particular set of parameters. The *fitness function* transforms that measure of performance into an allocation of reproductive opportunities. The evaluation of a string representing a set of parameters is independent of the evaluation of any other string. The fitness of that string, however, is always defined with respect to other members of the current population. In the genetic algorithm, fitness is defined by: f_i / f_A where f_i is the evaluation associated with string i and f_A is the average evaluation of all the strings in the population (Whitley, 1993).

For neural network weight optimization the objective function value can be expressed as the network error for the entire training set. The next step is to convert the objective function's value to a raw fitness. Since the goal is to minimize the objective function, smaller function values should produce larger fitness values. Also, later calculations will be simplified if the fitness is never negative. The best conversion function can be somewhat problem dependent. However, the exponential function: $f(v) = e^{-Kv}$ generally has been found to be useful where the network error v ranges from 0-1 (Masters, 1993).

The final evaluation step is converting the raw fitness values to a scaled fitness. If the raw fitness values were used to determine parent-selection possibilities, two problems could arise. One is that in the first few generations, one or a very few extremely superior individuals usually appear. Their fitness values are so high that they would be selected as parents too many times and their genetic material would quickly dominate the gene pool. Population diversity, which is crucial to genetic optimization, would be lost early on. The second problem is just the opposite. After many generations, clearly inferior individuals will have been weeded out. The population will consist of individuals who have relatively high raw fitness. The maximum fitness will usually be only slightly greater than the average. As a result, the fittest individuals will not be selected as parents in the high proportions necessary for continued rapid development (Masters, 1993).

A popular fitness scaling method involves applying a linear transform to the raw fitness values such that the average scaled fitness remains unchanged, but the maximum scaled fitness becomes a fixed multiple of the average. However, the presence of just one super-fit individual (with a fitness ten times greater than any other, for example), can lead to *over-compression*. If the fitness scale is compressed so that the ratio of maximum to average is 2:1, then the rest of the population will have fitness values clustered closely about 1. Although premature convergence has been prevented, it has been at the expense of effectively flattening out the fitness function. As mentioned above, if the fitness function is too flat, genetic drift will become a problem, so over-compression may lead not just to slower performance, but also to drift away from the maximum.

Fitness ranking is another commonly employed method, which overcomes the reliance on an extreme individual (Baker, 1985). Individuals are sorted in order of raw fitness, and then reproductive fitness values are assigned according to rank. This may be done linearly or exponentially. This gives a similar result to fitness scaling, in that the ratio of the maximum to average fitness is normalized to a particular value. However, it also ensures that the remapped fitness values of intermediate individuals are regularly spread out. Because of this, the effect of one or two extreme individuals will be negligible and over-compression ceases to be a problem. Several experiments have shown ranking to be superior to fitness scaling.

Parent Selection

It is helpful to view the execution of the genetic algorithm as a two stage process. It starts with the *current population*. Selection is applied to the current population to create an *intermediate population*. Then recombination and mutation operators are applied to the intermediate population to create the *next population*. The process of going from the current population to the next population constitutes one generation in the execution of a genetic algorithm. Goldberg (1989) refers to this basic implementation as a *Simple Genetic Algorithm*.

In the first generation the current population is also the initial population. After calculating f_i / f_A for all the strings in the current population, selection is carried out. The probability that strings in the current population are copied (i.e. duplicated) and placed in the intermediate generation is in proportion to their fitness.

There are a number of ways to do selection. The population might be viewed as mapping onto a roulette wheel, where each individual is represented by a space that proportionally corresponds to its fitness. By repeatedly spinning the roulette wheel, individuals are chosen using *stochastic sampling with replacement* to fill the intermediate population.

A selection process that will more closely match the expected fitness values is *remainder stochastic sampling*. For each string i where f_i / f_A is greater than 1.0, the integer portion of this number indicates how many copies of that string are directly placed in the intermediate population. All strings (including those with f_i / f_A less than 1.0) then place additional copies in the intermediate population with a probability corresponding to the fractional portion of f_i / f_A. For example, a string with $f_i / f_A = 1.36$ places 1 copy in the intermediate population, and then receives a 0.36 chance of placing a second copy. A string with a fitness of $f_i / f_A = 0.54$ has a 0.54 chance of placing one string in the intermediate population (Whitley, 1993).

Remainder stochastic sampling is most efficiently implemented using a method known as *stochastic universal sampling*. In this method it can be assumed that the population is laid out in random order as in a pie graph, where each individual is assigned space on the pie graph in proportion to fitness. An outer roulette wheel is placed around the pie with N equally-spaced pointers. A single spin of the roulette wheel will simultaneously pick all N members of the intermediate population. The resulting selection is also unbiased (Baker, 1987).
Implicit fitness remapping methods fill the mating pool without passing through the intermediate stage

of remapping the fitness. In *binary tournament selection*, pairs of individuals are picked at random from the population. Whichever has the higher fitness is copied into a mating pool (and then both are replaced in the original population). This is repeated until the mating pool is full (Goldberg, 1990). Larger tournaments may also be used, where the best of *n* randomly chosen individuals is copied into the mating pool. Using larger tournaments has the effect of increasing the selection pressure, since below-average individuals are less likely to win a tournament and vice-versa.

Reproduction

After selection has been carried out the construction of the intermediate population is complete and recombination can occur by applying crossover to randomly paired strings. This operation can be viewed as creating the next population from the intermediate population. Recombination operators are applied in order to generate new samples in the search space. Crossover is not usually applied to all pairs of individuals selected for mating. A random choice is made, where the probability of crossover being applied is typically between 0.6 and 1.0. If crossover is not applied, offspring are produced simply by duplicating the parents. This gives each individual a chance of passing on its genes without the disruption of crossover (Whitley, 1993).

A binary string encoding would represent a possible solution to some parameter optimization problem. New sample points in the space are generated by recombining two parent strings. If the string 1101001100101101 and another binary string, yxyyxyxxyyyxyxxy, in which the values 0 and 1 are denoted by x and y, are recombined using a single randomly-chosen recombination point, 1-point crossover occurs as follows:

```
11010 \/ 01100101101
yxyyx /\ yxxyyyxyxxy
```

Swapping the fragments between the two parents produces the following offspring:

11010yxxyyyxyxxy and yxyyx01100101101

The problem with adding additional crossover points is that building blocks (hyper-plane partitions within search space that contain significant genetic information) are more likely to be disrupted. However, an advantage of having more crossover points is that the problem space may be searched more thoroughly. In *2-point crossover*, chromosomes are regarded as loops formed by joining the ends together. To exchange a segment from one loop with that from another loop requires the selection of two cut points. 1-point crossover can be seen as 2-point crossover with one of the cut points fixed at the start of the string. Hence 2-point crossover performs the same task as 1-point crossover (i.e. exchanging a single segment), but is more general (Whitley, 1993).

Uniform crossover is radically different to 1-point crossover. Each gene in the offspring is created by copying the corresponding gene from one or the other parent, chosen according to a randomly generated *crossover mask.* Where there is a 1 in the crossover mask, the gene is copied from the first parent, and where there is a 0 in the mask, the gene is copied from the second parent (Syswerda, 1989). The process is repeated with the parents exchanged to produce the second offspring. A new crossover mask is randomly generated for each pair of parents. Offspring therefore contain a mixture of genes from each parent. The number of effective crossing points is not fixed, but will average *L/2* (where *L* is the chromosome length).

Despite analytical results showing uniform crossover is in every case more disruptive than 2-point crossover for order-3 schemata (hyper-plane partitions represented by 3 bit substrings) for all defining string lengths, several researchers have suggested that uniform crossover is a better recombination operator. Spears and DeJong (1991) speculate that, "With small populations, more disruptive crossover operators such as uniform or *n-point* (*n* >> 2) may yield better results because they help overcome the limited information capacity of smaller populations and the tendency for more homogeneity." Uniform crossover appears to be more robust. Where two chromosomes are similar, the segments exchanged by 2-point crossover are likely to be identical, leading to offspring which are identical to their parents. This is less likely to happen with uniform crossover.

Mutation

The *mutation operator* is applied to each offspring after crossover. For each bit in the new population, a mutation can occur with some low probability p_m. It is typical for the mutation rate to be within 0.1%-1% probability. Mutation is applied by flipping the bit value (Whitley, 1993).

A genetic algorithm will always be subject to stochastic errors. One such problem is that of *genetic drift*. Even in the absence of any selection pressure, members of the population will still converge to some point in the solution space. If, by chance, a gene becomes predominant in the population, then it is just as likely to become more predominant in the next generation as it is to become less predominant. If an increase in predominance is sustained over several successive generations, and the population is finite, then a gene can spread to all members of the population. Once a gene has converged in this way, crossover cannot introduce new gene values. The rate of genetic drift can be reduced by increasing the mutation rate. However, if the mutation rate is too high, the search becomes effectively random.

Mutation is traditionally seen as a "background" operator, responsible for introducing *alleles* or inadvertently lost gene values, preventing genetic drift and providing a small element of random search in the vicinity of the population when it has largely converged (Whitley, 1993). However, mutation becomes more productive, and crossover less productive, as the population converges. Despite its generally low probability of use, mutation is a very important operator.

Neurogene Application

The NeuroGene application is a program that runs a neural network and a genetic algorithm in conjunction. The neural network is used as the prime modeling tool and the genetic algorithm is used to optimize network parameters. The application permits the user to create any single or double hidden layer feedforward network architecture and load sample data for training the network. The network weights and architecture can be saved at any time along with input/output data scaling parameters. A new network may be created by loading this file at any time. If the network is created from previously saved weights, the application will only allow execution of input patterns. Training will not be allowed in execution mode but a set of input vectors may be loaded from a file and presented to the network's inputs. The application will display the network's response to each input vector.

Create New Network Dialog

This dialog permits the user to set sample data scaling parameters and choose among multilayer network architectures. The number of neurons at each hidden layer and the type activation function (linear or logistic) for the output layer neurons can be set. The training and test sample data is loaded when a new network is created. The user can type in the path for the data file. Test data can be randomly extracted from the sample data or loaded explicitly. Testing data is never presented to the network during training but it is very important for the validation of the network. Without test samples, the user will have no measure of the network's performance when presented with general real-world data. Test data validation is the only way to detect over-fitting issues that can occur during training.

If a variable is used to train output neurons and the output neurons have an activation function with bounded range, target activations must certainly be limited to values that can comfortably be learned. That is why scaling is very important. Another reason for uniform scaling is to initially equalize the importance of variables. If one variable has an order of magnitude of 1,000,000 while another is about 0.000001, it asking a lot of the learning algorithm to traverse such a range. The network's life can be made a lot easier by giving it data scaled in such a way that all weights remain in small, predictable ranges.

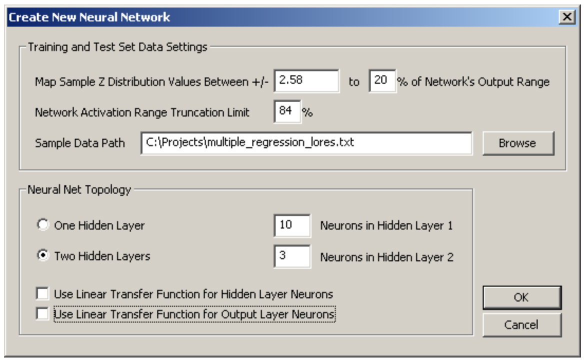
NeuroGene employs normalization based on the population's mean and distribution values. The input data is standardized to a *Z-score* by subtracting its mean and dividing by standard deviation:

$$Z = (x - \mu) / \sigma$$

This removes all effects of offset and measurement scale. Simply scaling to a Z-score is not generally sufficient for output variables, as the scaled values would still exceed the activations limits implicit in the network's model. NeuroGene will map from Z-score to neuron activation:

$$A = r\,[(x - \mu) / \sigma - Z_{min}] + A_{min},$$
$$r = (A_{max} - A_{min}) / (Z_{max} - Z_{min})$$

The practical limits for this mapping can be set in the network creation dialog. When the network's practical output range is set to 20% (of activation range 0-1) normalized variable practical limits Z_{max} and Z_{min} will be mapped to 0.6 (A_{max}) and 0.4 (A_{min}) respectively. Occasional outlier data falling outside those boundaries may be clipped at the network's truncation limits which can also be set within the dialog.


Create New Neural Network

Training and Test Set Data Settings

Map Sample Z Distribution Values Between +/- 2.58 to 20 % of Network's Output Range

Network Activation Range Truncation Limit 84 %

Sample Data Path C:\Projects\multiple_regression_lores.txt

Neural Net Topology

One Hidden Layer [✓ Two Hidden Layers]

10 Neurons in Hidden Layer 1

3 Neurons in Hidden Layer 2

☐ Use Linear Transfer Function for Hidden Layer Neurons

☐ Use Linear Transfer Function for Output Layer Neurons

Figure 2: Create New Neural Network Dialog

Main Application Screen

The main application screen allows the user to set various training and genetic algorithm parameters, and provides several mating selection scheme choices. After a network has been created, training can be initiated and terminated by pressing the "*Start/Stop Net Training*" button. Training may also automatically end when the mean square error of the output dips below "*Target Error*" or the genetic algorithm has been running for more than "*Max Generations*". During training, the current generation number, minimum network error and current population mean error values are presented in real-time at the display screen. At the end of a training session, all input patterns in the sample test set are presented to the network and the outputs are compared to sample target outputs for calculating the root mean square error in the output variable's original un-scaled range. The root mean square error is a good measure of the network's real-world performance.

The "*crossover rate*" parameter defines the percentage of individuals chosen among the intermediate population as parents. The "*uniform crossover %p*" parameter defines the percentage of genes the first offspring will receive from one of its parents and the second offspring will receive from the other parent. This parameter is 50% for standard uniform crossover which can cause too much disruption of valuable genetic information. It must be noted that a low %p parameter should be accompanied with a low crossover rate to preserve the gene pool.

The "*Bits per Gene*" parameter specifies the number of bits used to represent each network weight. A large value will dramatically increase the search space, which in turn will lead to longer training times or increased difficulty in convergence to a global minimum. A low value might not provide enough resolution for sampling the search space and cause the algorithm to miss narrow valleys in the objective function. It is a good idea to use a larger population size for a larger search space. The "*Phenotype Range*" parameter defines the maximum and minimum network weight values. The gray encoded genes are decoded and scaled to that range during execution.

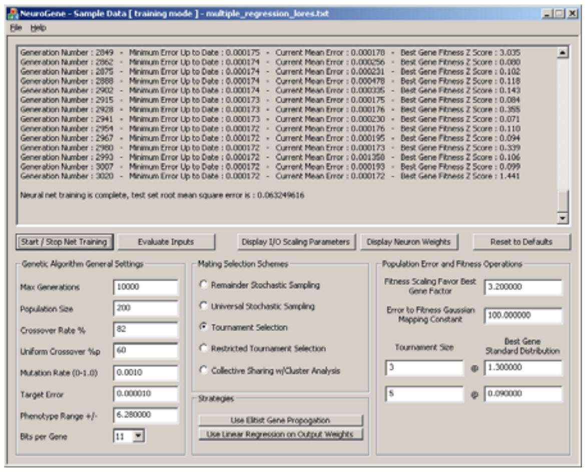
When remainder or universal stochastic sampling is chosen as the parent selection method, NeuroGene uses *sigma truncation* for scaling fitness values:

$$F' = F - (F_{avg} - c\,\sigma)$$

F_{avg} is the mean fitness value and σ is the standard deviation. c is the sigma scaling factor and is set internally based on the fitness distribution. The user can also set a mapping constant K, which is used in error to fitness Gaussian mapping $f(v) = e^{-Kv}$.

When tournament selection is selected as the parent selection method, NeuroGene uses an adaptive tournament size based on the difference between the population's mean fitness value and best fitness value. The difference is expressed in terms of multiples of the population's current fitness standard deviation. Two fitness distribution values, corresponding to the minimum and maximum tournament sizes, can be set by the user. NeuroGene decreases selection pressure by reducing tournament size when the population contains super-fit individuals and increases selection pressure when the population displays a flatter fitness distribution.

If the "Use *Elitist Gene Propagation*" checkbox is selected, the fittest individual in the current population will be guaranteed to pass on its genes to the next generation. Applying this strategy may lead to premature convergence of the genetic algorithm.


NeuroGene - Sample Data [training mode] - multiple_regression_lores.txt
File
Help
Start / Stop Net Training
Evaluate Inputs
Display I/O Scaling Parameters
Display Neuron Weights
Reset to Defaults
Use Elitist Gene Propogation
Use Linear Regression on Output Weights

```
Generation Number : 2849  -  Minimum Error Up to Date : 0.000175  -  Current Mean Error : 0.000178  -  Best Gene Fitness Z Score : 3.035
Generation Number : 2862  -  Minimum Error Up to Date : 0.000174  -  Current Mean Error : 0.000256  -  Best Gene Fitness Z Score : 0.080
Generation Number : 2875  -  Minimum Error Up to Date : 0.000174  -  Current Mean Error : 0.000231  -  Best Gene Fitness Z Score : 0.102
Generation Number : 2888  -  Minimum Error Up to Date : 0.000174  -  Current Mean Error : 0.000478  -  Best Gene Fitness Z Score : 0.118
Generation Number : 2902  -  Minimum Error Up to Date : 0.000174  -  Current Mean Error : 0.000335  -  Best Gene Fitness Z Score : 0.143
Generation Number : 2915  -  Minimum Error Up to Date : 0.000173  -  Current Mean Error : 0.000175  -  Best Gene Fitness Z Score : 0.084
Generation Number : 2928  -  Minimum Error Up to Date : 0.000173  -  Current Mean Error : 0.000176  -  Best Gene Fitness Z Score : 0.355
Generation Number : 2941  -  Minimum Error Up to Date : 0.000173  -  Current Mean Error : 0.000230  -  Best Gene Fitness Z Score : 0.071
Generation Number : 2954  -  Minimum Error Up to Date : 0.000172  -  Current Mean Error : 0.000176  -  Best Gene Fitness Z Score : 0.110
Generation Number : 2967  -  Minimum Error Up to Date : 0.000172  -  Current Mean Error : 0.000195  -  Best Gene Fitness Z Score : 0.094
Generation Number : 2980  -  Minimum Error Up to Date : 0.000172  -  Current Mean Error : 0.000173  -  Best Gene Fitness Z Score : 0.339
Generation Number : 2993  -  Minimum Error Up to Date : 0.000172  -  Current Mean Error : 0.001350  -  Best Gene Fitness Z Score : 0.106
Generation Number : 3007  -  Minimum Error Up to Date : 0.000172  -  Current Mean Error : 0.000193  -  Best Gene Fitness Z Score : 0.099
Generation Number : 3020  -  Minimum Error Up to Date : 0.000172  -  Current Mean Error : 0.000172  -  Best Gene Fitness Z Score : 1.441

Neural net training is complete, test set root mean square error is : 0.063249616
```

Genetic Algorithm General Settings

Max Generations	10000
Population Size	200
Crossover Rate %	82
Uniform Crossover %p	60
Mutation Rate (0-1.0)	0.0010
Target Error	0.000010
Phenotype Range +/-	6.280000
Bits per Gene	11

Mating Selection Schemes

- ○ Remainder Stochastic Sampling
- ○ Universal Stochastic Sampling
- ● Tournament Selection
- ○ Restricted Tournament Selection
- ○ Collective Sharing w/Cluster Analysis

Population Error and Fitness Operations

Fitness Scaling Favor Best Gene Factor	3.200000
Error to Fitness Gaussian Mapping Constant	100.000000

Tournament Size		Best Gene Standard Distribution
3	@	1.300000
5	@	0.090000

Figure 3: Neuro-Gene Application Main Screen

When the "*Use Linear Regression on Output Weights*" checkbox is selected, output layer neuron weights will not be represented in the chromosome bit string. The genetic algorithm will only optimize hidden layer neuron weights. Neural networks are explicitly nonlinear. However, the operation of passing the output activations of one layer to the input of the next layer is linear. The input applied to an output layer neuron is a linear combination of the activations of the neurons in the previous layer. If that hidden layer's activations are treated as independent variables, and if the known desired input to an output neuron is treated as a dependent variable, the problem becomes a linear regression problem. The output neuron weight vector will be optimal in that it minimizes the mean square error of the input to the output neuron. To solve with linear regression, the inverse transfer function of the neuron's desired output must be computed for each training sample. Using linear regression in conjunction with a genetic algorithm is computationally very intensive but the algorithm is likely to converge in less generations. Linear regression interferes with the natural evolution of the population by producing dominant super-fit individuals early on, therefore in some cases it may lead to premature convergence.

BASICS OF NA/K GEOTHERMOMETERS

Chemical geothermometers are important tools that are used for prediction of equilibrium temperatures of geothermal systems. These geothermometers are analytical equations founded in empirical form, on the basis of data created by measured temperatures and chemical composition of fluids sampled in hot springs and wells.

Na/K ratio of geothermal fluids (spring and well waters) is likely to provide a significant indication of subsurface temperatures. The Na/K ration in natural hot water is controlled by a reversible temperature dependent rock-water equilibrium involving potash-mica, potash-feldspar and albite. The reversible relationship appears only at temperatures above 200^{o}C. The Na/K equilibrium adjusts after a temperature change relatively slowly, which enables useful information on conditions in the deep aquifer to be obtained from the values of Na/K in spring waters.

Na/K geothermometers based on this phenomenon has evolved in the past 40 years and several experimentally derived equations have been proposed by Ellis and Mahon (1967), Ellis (1970), Truesdell (1976), Fournier, (1979), Tonani, (1980), Arnorsson, (1983) and Gigenbach, (1988). Those equations work well for reservoirs with temperatures in the 180-350^{o}C range, but break down at lower temperatures, notably at less than 120^{o}C. At these temperatures Na and K concentrations are influenced by other minerals, such as clays, and are not controlled only by feldspar ion-exchange reaction (Nicholson, 1993).

Citing critics of Santoyo and Verma, (1993) and Verma and Santoyo, (1997) on validity and reliability of conventional Na/K geothermometers, Diaz et al, (2008) concluded that the effects of primary error sources for conventional geothermometers might be as follows: (1) analytical errors in chemical analysis, (2) errors in regression coefficients of developed equations, (3) errors derived from incorrect use of solute concentration units for geothermometers, (4) correct geochemical conditions, temperature range and concentration of same equations, (5) lack of well temperatures and rock-fluid interaction experimental temperatures for low and intermediate temperature ranges, (6) limited number of data collected and outliers among them. Therefore, Santoyo and Verma, (1993) and Verma and Santoyo, (1997) examined Na/K geothermometer through statistical theory of error propagation, and on this basis proposed new Na/K geothermometers.

APPLICATIONS OF NEW ANN MODEL FOR NA/K GEOTHERMOMETERS

The ANN computational technique has been perceived as an effective tool; (1) for relatively simple solutions for complex numerical problems, (2) for substituting experimental works that are difficult to realize.

Bayram (2001) and Can (2002) proposed new geothermometer equations of Na/K through implementation of ANN. Bayram (2001) proposed a simple ANN model using an non linear logistic activation function, which was trained with 6 known geothermometer equations. The architecture of ANN was trained with synthetic data of Na-K as input layer neurons and reservoir temperatures as output layer neurons that were inferred from the results of 6 known geothermometers. On the other hand, Can (2002) empirically developed a new Na/K geothermometer on the basis of 39 data set collected from various geothermal fields around the world. The architecture of ANN also took into account of a simple input layer of one neuron, a hidden layer and an output layer of one neuron. Training was conducted with back-propagation algorithm using again a non-linear logistic activation function.

Recently, Diaz et al., (2008) proposed 3 new Na/K geothermometers developed using ANN and ordinary linear regression. The obtained results appear to show that new geothermometers systematically provide better and reliable estimations of the deep equilibrium temperatures than the equations previously reported in the geothermal literature. They applied ANN method in obtaining two geothermometers and linear regression approach in the third one. In the first geothermometer the architecture of ANN was trained with real data of reservoir temperatures collected from different fields around the world as input layer neuron, a hidden layer and as output layer neuron of Na-K. Training was performed with back-propagation algorithm using a linear activation function taking advantage of linear nature of Na/K geothermometer [(1/T) = A log(Na/K) + B]. In the second geothermometer proposed by Diaz-Gonzalez et al., (2008), training was also conducted by back-propagation method using this time tangent hyperbolic activation function.

Considering limitations and uncertainties of conventional Na/K geothermometers, in this study several models for this geothermometer have been developed through training ANN models with data created from 324 data of measured temperatures and chemical compositions collected from geothermal wells all around the world. Data collected and examined in Diaz et al., (2008) study was composed of a training data set of 212 and a test data set of 112. In our study we have also used these two data sets as training and test data sets, and on the other hand, we have also utilized another data set of 39 collected by Can, (2002) as test data for our models.

Using our ANN software the following models are created in two groups, and in this first group of models the input is log(Na/K) and the output is 1/T:

- Model#1: Hidden layer & Output Layers => Logistic (with 7-8 neurons and percentage of No. of output range: 20).
- Model#2: Hidden Layer => Logistic; Output Layer => Linear (with 3 neurons and percentage of No. of output range: 20-40).
- Model 2a: Hidden Layer => Logistic; Output Layer => Linear, applied linear regression for output weights (with 3 neurons and percentage of No. of output range: 20).
- Model#3: Hidden & Output Layers => Linear (with 3 neurons and percentage of No. of output range: 20).

In the second group of following models inverse solution is used, and here input is temperature (1/T) and the output is log(Na/K).

- Model#4: Hidden & Output Layers => Logistic (with 3 neurons and percentage of No. of output range: 40).
- Model#5: Hidden Layer => Logistic; Output Layer => Linear (with 3 neurons and percentage of No. of output range: 60).
- Model#6: Hidden & Output Layers => Linear (with 3 neurons and percentage of No. of output range: 80).

As seen in the above models, logistic and linear activation functions, different percentage of number of output ranges and inverse models are used.

RMSE values were calculated by comparing training results with test data set. Minimum RMSE values obtained for these models are given in Table 1.

Table 1. RMSE Values for Different ANN Models.

Models	RMSE Values
Model#1	0.000106
Model#2	0.000098
Model#3	0.000096
Model#4	0.100533
Model#5	0.096819
Model#6	0.086503

The results of first model are shown in Fig. 1. As seen from Fig. 4, Diaz-Gonzalez et al, (2008) and Verma and Santoyo (1997) geothermometers represent well lower and upper limits of scattered data (n=112), respectively. Our 3 models remain in between and seem to represent the whole data. Diaz et al., (2008)'s third geothermometer which was obtained by regression analysis is very close to our models.

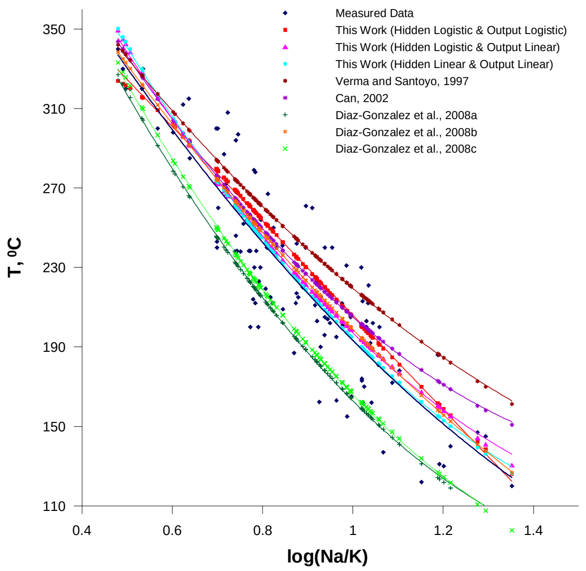


Figure 4: Comparative results of our first three ANN models with Diaz-Gonzalez et al, (2008) data set.

Similar results have been obtained with inverse models as seen in Fig. 5. These models also seem to represent the whole scattered data better.

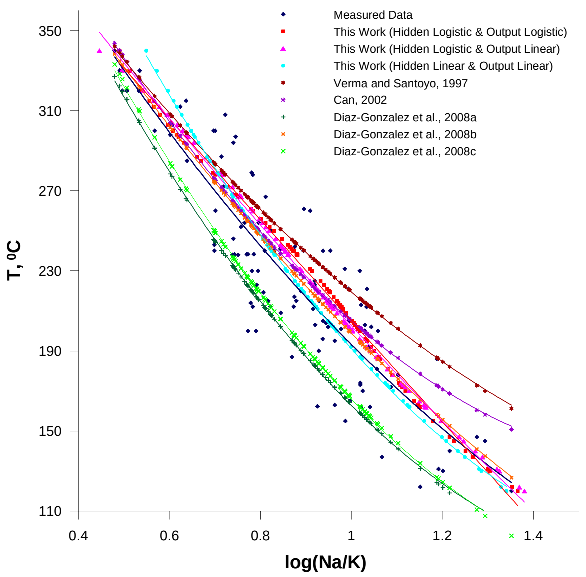


Figure 5: Comparative results of 3 inverse ANN models with Diaz-Gonzalez et al, (2008) data set.

As seen in Fig. 6 all results of our ANN models and some others for test data set (n=39) using from Can (2002) are pretty close each other. The best geothermometer which is close to measured temperatures is the one of Verma and Santoyo (1997). It appears that all geothermometers based on ANN models match better the measured temperatures in 180°-350°C range. In the lower range <160°C, Verma and Santoyo (1997) geothermometer performs better.

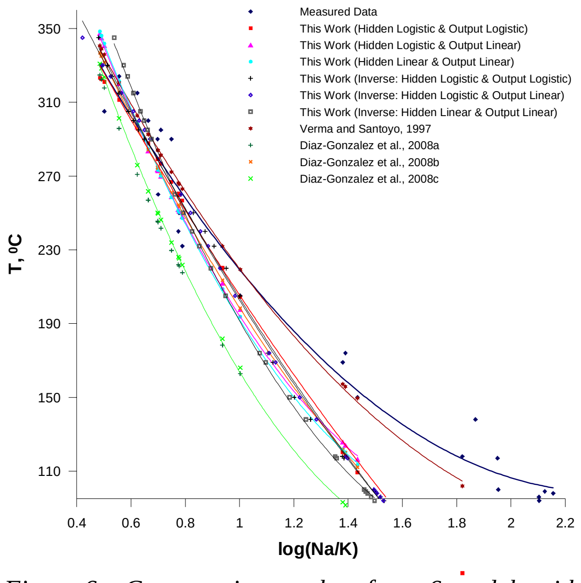


Figure 6: Comparative results of our 6 models with the others models with Can, (2002) data set.

In order to evaluate the results, a comparative study is conducted on resulting model solutions. This study is based on the calculation of percentage of deviation equation proposed by Verma and Santoyo (1997) as follows:

$$\% \text{ DEV} = [(t_c - t_m)/t_m]*100$$

Where, t_c and t_m are calculated by geothermometers and measured temperatures in wells, respectively, and %DEV is percentage of deviation.

%DEV is used as statistical parameter to evaluate the exactness of calculated geothermometer temperatures, assuming that the measured temperatures are real downhole temperatures of wells. Results obtained in calculation of %DEV for the first 3 models are shown in Fig. 4. As it can be seen in Fig. 4 %DEV is around within 10% like other geothermometers. On the other hand, %DEV is unusually high below 160°C. In their work, Diaz et al., (2008) have pointed out the same problem. They have attributed this behavior to insufficient data in that range. We have also observed the same behavior for our models, as seen in Fig 7.

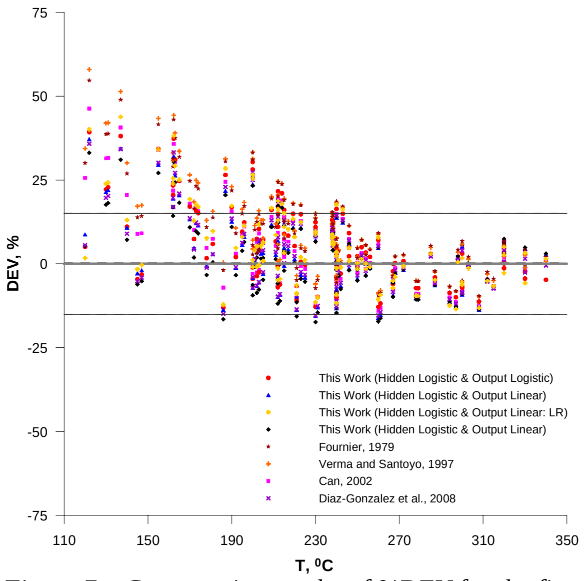


Figure 7: Comparative results of %DEV for the first 4 ANN models with Diaz-Gonzalez et al, (2008) data set.

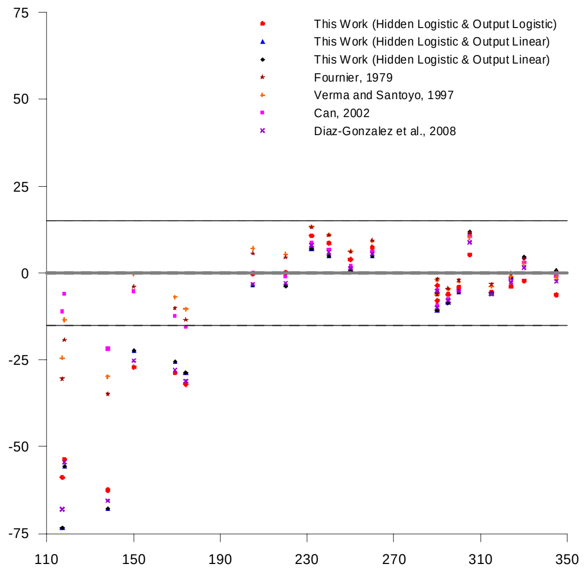


Figure 8: Comparative results of %DEV for the first 3 ANN models with Can, (2002) data set.

On the other hand, as seen in Fig. 8 all ANN based geothermometers seem to underestimate bottomhole temperatures in that range (<160°C) when Can (2002) data set is used. There is a remarkable difference between two data sets below 160°C. For this range, while log(Na/K) values range from 0.8 to 1.4 in Diaz et al., (2008) data set, they are found between 1.4 and 2.2 in Can (2002) data set. This may have been born from the fact that very few data really exist in Can (2002) data set below 160°C. On the other hand, there might be a difference between chemical compositional characteristics of samples collected.

Moreover, there may be outliers within the data set of Can (2002) data set.

CONCLUDING REMARKS

- A new ANN software using genetic algorithm is made
- Several ANN models that represent well the Na/K geothermometer are created.
- It seems more reliable data are needed to have a Na/K geothermometer representing better resources below 160^oC.

ACKNOWLEDGEMENT

We would like to extend our gratitude to Dr. Surendra Pal Verma and Lorena Diaz-Gonzalez and her coauthors for providing us data sets that have enabled us to realize this study.

REFERENCES

Arnorsson, S., 1983. “Chemical Equilibria in Icelandic Geothermal Systems; Implications for Chemical Geothermometry Investigations,” *Geothermics*, 12, 119-128.

Baker, J., 1985. “Adaptive selection methods for genetic algorithms.”, Proc. International Conf. on Genetic Algorithms and Their Applications. J. Grefenstette, ed. Lawrence Erlbaum.

Baker, J., 1987. “Reducing Bias and Inefficiency in the Selection Algorithm”, Genetic Algorithms and Their Applications: Proc. Second International Conf. J. Grefenstette, ed. Lawrence Erlbaum.

Bayram, A.F., 2001. “Application of An Artificial Neural Network Model to A Na-K Geothermometer,” *Journal of Volcanology and Geothermal Research* 112 (2001) 75-81.

Can, I., 2002. “A New Improved NA/K Geothermometer by Artificial Neural Networks,” *Geothermics* 31(2002) 751-760.

Diaz, G.L., Santoyo, E., y Reyes, J.R., 2008. “Desarollo de Nuevos Geotermometros Mejorados de Na/K Usando Redes Neoronales Artificiales Estadisticas: Aplicacion a la Prediccion de Temperaturas de Sistemas Geotermicos,” *Revista Mexicana de Ciencias Geologicas*, 2008, 25(3), 465-482.

Ellis A. J. and Mahon W.A.J., 1967. “Natural Hydrothermal Systems and Experimental Hot-Water/Rock Interactions (Part II),” *Geochim. Cosmochim. Acta*, 31, 519-538.

Ellis, A.J., 1970. “Quantitative Interpretation of Chemical Characteristics of Hydrothermal Systems,” *Geothermics*, 25, 219-226.

Fausett, L., 1994. “Fundamentals of Neural Networks”, Prentice-Hall, Inc.

Fournier, R.O., 1979. “A Revised Equation for Na/K Geothermometer,” Geoth. Res. Council Trans., 3, 221-224.

Gigenbach, W.F., 1988. “Geothermal Solute Equilibria. Derivation of Na-K-Mg-Ca Geoindicatores,” *Geochim. Cosmochim.Acta*, 52, 2749-2765.

Goldberg, D., 1989. “Genetic Algorithms in Search, Optimization and Machine Learning”, Reading, MA: Addison-Wesley.

Goldberg, D., 1990. “A Note on Boltzmann Tournament Selection for Genetic Algorithms and Population-oriented Simulated Annealing”, TCGA 90003, Engineering Mechanics, Univ. Alabama.

Holland, J.H., 1975. "Adaptation in Natural and Artificial Systems", MIT Press.

Masters T., 1993. “Practical Neural Network Recipes in C++”, Morgan-Kaufmann.

Nicholson, K., 1993. “*Geothermal Fluids. Chemistry and Exploration Techniques*”, Springer-Verlag, Berlin Heidelberg, 72-73.

Santoyo E. and Verma, S.P., 1993. “Evalucion de Errores en el Uso de los Geotermometros de SiO_2 y Na/K para la Determinacion de Temperaturas en Sistemas Geotermicos,” *Geofisica Internacional*, 32, 287-298.

Spears, W. and DeJong, K., 1991. “An Analysis of Multi-Point Crossover”, Foundations of Genetic Algorithms, G. Rawlins, ed. Morgan-Kaufmann.

Syswerda, G., 1989. “Uniform Crossover in Genetic Algorithms”, Proc 3rd International Conf on Genetic Algorithms, Morgan-Kaufmann, pp 2-9.

Tonani, F., 1980. “Some Remarks on the Application of Geochemical Techniques in Geothermal Exploration,” In: Proc. Adv. Eur. Geoth. Res., Second Symposium, Strasburg, 428-443.

Truesdell, A.H., 1976. “Summary of Section III. Geochemical Techniques in Exploration,” Proceedings 2nd UN Symposium on the development and use of geothermal resources. San Fransisco, 1, Iiii-Ixxix.

Verma S.P. and Santoyo, E., 1997. “New Improved Equations for Na/K, Na/Li and SiO_2 Geothermometers by Outlier Detection and Rejection,” *Journal of Volcanology and Geothermal Research*, 79(1-2), 9-24.

Whitley, D., 1993. "A Genetic Algorithm Tutorial", Technical Report CS-93-103, Colorado State University.