Journal of Engineering and Applied Sciences 2 (1): 17-29, 2007
© Medwell Journals, 2007

Implementation of Artificial Neural Network with Hidden Markov Model for Analysing the Genetic Code

[1]C. Vijayalakshmi and [2]k. Senthamarai Kannan
[1]Department of Mathematics, Sathyabama Institute of Science and Technology Jeppiar Nagar, Chennai 600 119
[2]Department of statistics, Manonmainam Sundaranar University, Tirunelveli, Tamil Nadu, India

Abstract: This study mainly deals with a general framework for Hidden Markov Models and Neural Networks by using back propagation algorithm. In the training phase an efficient way of updating the weights of neurons based on the relative entopy function is introduced, so that the network converges in a rapid manner. Exponential gradient descent algorithm has been used for updating the weights of neurons in each iteration of the training phase. Here the component of the gradient term appears in the exponent of a factor that is used in updating the weight vector multiplicatively. A scaling factor is derived in which the hidden layer output is linear which represents the total weight on hidden layer nodes. The numbers of hidden layer units were also varied for the various learning rates and the performance were marked. The efficiency and accuracy of learning process greatly depends on the training algorithm used. An algorithm is designed in such a way that a new weight update rule has been introduced. Exponentiated gradient descent weight update rate that substitutes the existing gradient descent update rate of back propagation algorithm in which the results are faster and it is an efficient Markov learning network. The Hidden Neural Network can be viewed as an undirected probabilistic model in which the study of the structure of the standard genetic code is analyzed.

Key words: Genetic code, scaling factor, DNA, RNA, hidden neurons, neural network, pattern recognition

INTRODUCTION

In all areas of biological research, the role of Hidden Markov Models have proved useful like protein modeling and gene findings by Richard Durbin[1]. With the advance in technology of bio equipments like DNA sequences, enormous amount of date is being generated. A fundamental feature of chain molecules, which are responsible for the functioning ad evolution of the organisms, is that they can be cast in the form of digital symbol sequences. The nucleotide and amino acid monomers in DNA, RNA and protein are distinct and can be represented as a set of symbols. Machine learning techniques are excellent for the task of discarding and compacting redundant sequence information Pierre Baldi[2]. The training procedure superimposes sequences upon one another in a way that transforms a complex topology in the input sequence space into a simpler representation.

Neural network approach does not require any modeling between process parameters and the outputs are process state variables. The network maps the input domains with the output domains. The inputs are process parameters and the outputs are process state variables. Each process parameter or process state variable is called feature. The combination of input and output constitutes a pattern. Neural computing concepts were discussed Morton. H[3], Simon Haykin[4], Philip D. Wasserman[5] which forms the basis in order to have a general framework.

Hidden markov model: Although the Hidden Markov model is good at capturing the temporal nature of process, it has very limited capacity for recognizing complex patterns involving more than first order dependence in the observed date. Multi-layer Perceptrons are of opposite in this, we cannot model temporal phenomena, but complex patterns can be recognized. Maximum likelihood estimation is a discriminative training algorithm that aims at maximizing the ability of the model to discriminate between different classes. In this paper let us analyze the sequence of encoding for Multi-layer Feed Forward network as it consists of hidden neurons by using "CLASS HIDDEN MARKOV MODEL" to incorporate neural network in a valid probabilistic way Fig. 1.

Corresponding Author: C. Vijayalakshmi, Department of Mathematics, Sathyabama Institute of Science and Technology Jeppiar Nagar, Chennai 600 119, India

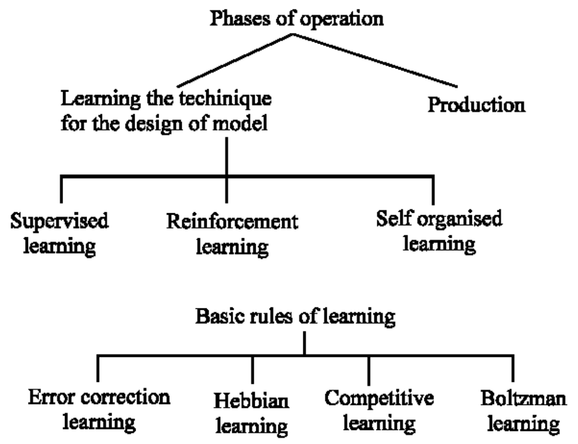


Fig. 1: Hidden markov model design framework

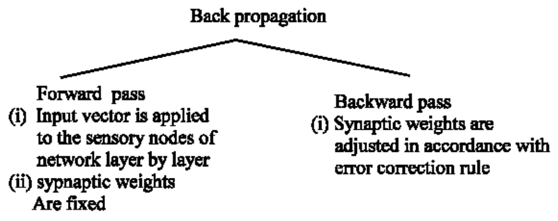
Sequence encoding and output interpretation: A new machine learning technique is used because of their ability to cope with non-linearities and to find more complex correlation in sequence space. In a Multi layer Perceptons the last hidden layer preceding the output units should represent the transformed linear function in a representable form. In many sequences analysis problems the input is often associated with a window of size w, covering the relevant sequence segment. The sequence analysis was discussed David W.Mount[1] in accordance with Genome Biological Analysis . For each position in a window there are |A| different possible monomers.

The space encoding scheme requires an input layer of size |A| * w. The neural networks technique has the potential to correlate the different input values to each other. The co-operativity of the weights that result from training is supposed to mirror the relevant correlations between the monomers in the input which are again correlated to the prediction task carried out by the network. An artificial neural network is defined as a massively parallel distributed process or that has a natural propensity for storing experimental knowledge and make it available for use.

This is a typical Markov process in which knowledge is acquired by modeling the temporal structure of date. Inter neuron connection strengths known as synaptic weights or interconnection weights are used to store the knowledge.

The main application in which in neural network approach computer programs are trained to be able to recognize amino acid patterns that are located in known secondary structures use of back propagation method to predict protein structures Qian[6] and Michael[7,8] which resulted in 64% accuracy. Programming Techniques were designed James A.Freeman[9] for neural network. Two more successful methods are PHD and INPREDICT for class Hidden Neural network. In prediction problems there are two methods in which GRAIL II predicts encoded protein sequences, constructs gene models, GENE PASER predicts the most likely combination of exons and introns in a genomic sequence by dynamic programming approach.

Back propagation network: Multi-layer Feed Forward network consists of many interconnected single layer perceptrons and the flow of signal is allowed only in the forward direction. An error back propagation algorithm is based on the error correction learning rule.


Back propagation
Forward pass
(i) Input vector is applied to the sensory nodes of network layer by layer
(ii) sypnaptic weights Are fixed
Backward pass
(i) Synaptic weights are adjusted in accordance with error correction rule

The back propagation algorithm trains a neural network using a gradient descent algorithm in which the mean square error between the network's output and the desired output is minimized by Hill[10]. Once the error is decreased, the network has converged and it is a Markov Model training is achieved in each layer.

MULTILAYER FEED FORWARD NETWORK

From the above Fig. 2, the input and output values along with the different types of layers, weighted matrices are shown in which the multiplayer feed forward has three distinctive characteristics:

- The model of each neuron in the network includes a non-linearity at the output end. The non-linearity is smooth (i.e., differential everywhere), a. A commonly used form of non-linearity that satisfies this requirement is a sigmoidal activation function defined by:

$$y_j = \frac{1}{1+\exp(-v_j)} \tag{2.1}$$

where v_j is the net internal activity level of neuron j and y_j is the output of the neuron. The presence of non-linearity is important because, otherwise, the input-output relation of the network is reduced to that of a single-layer perceptron. Moreover, the use of the logistic function is biologically is motivated, since it attempts to account for the refractory phase of neural networks.

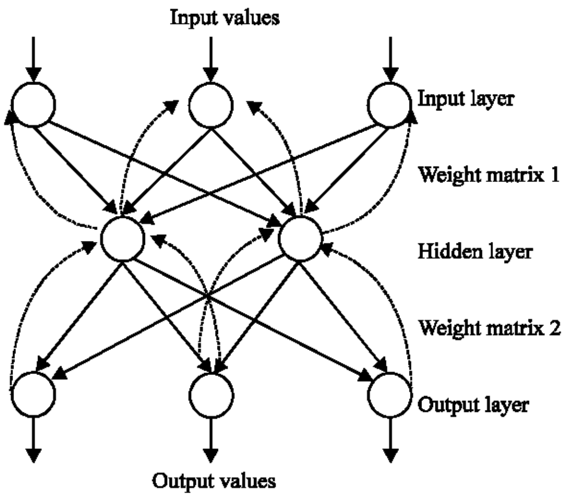


Fig. 2: Multilayer feed forward network

- The network layers consist of one or more layers of hidden neutrons that are not part of the input or output of the network. These hidden neurons enable the network to learn complex tasks by extracting progressively more meaningful features from the input patterns (vectors).
- The network exhibits a high degree of connectivity, determined by the synapses of the network. A change in the connectivity of the network requires a change in the population of synaptic connections or their weights.

In this network, two kinds of signals are identified as follows

Function signals: A function signal is an input signal (stimulus) that comes in at the input end of the network, propagates forward (neuron-by-neuron) through the network and emerges at the output end of the network as an output signal.

Error signals: An error signal originates at an output neuron of the network and propagates backward (layer by layer) through the network. We refer to it as an error signal because its computation by every neuron of the network involves an error dependent function in one form or another.

Hidden layer: The output neurons constitute the output layer and the input neurons constitute the input layer of the network. The remaining neurons constitute hidden layers of the network. Thus the hidden units are not part of the output or input of the network-hence their designation as hidden. The first hidden layer is fed from the input layer made up of sensory units (source nodes); the resulting outputs of the first hidden layer are in turn applied to the next hidden layer; and so on for the rest of the network. Each hidden or output neuron of a multiplayer feed forward network is designed to perform two computations:

- The computation of the function signal appearing at the output of a neuron, which is expressed as a continuous nonlinear function of the input signals and synaptic, associated with that neuron.
- The computation of an instantaneous estimate of the gradient vector (i.e., the gradients of the error surface with respect to the weights connected to the inputs of a neuron), which is needed for the backward pass through the network.

Rate of learning: The back propagation algorithm provides an approximation to the trajectory in weight space computed by the method of steepest descent. The smaller we make the learning rate parameter η the smaller will the changes to the synaptic weights in the network be from one iteration to the next and the smoother will be the trajectory in weight space. This improvement however is attained at the cost of slower rate of learning. If, on the other hand, we make the learning-rate parameter η to large to speed up the rate of learning, the resulting large changes in the synaptic weights assume such a form that a network may become unstable (i.e., oscillatory). In deriving the back-propagation algorithm, it is assumed that the learning parameter is a constant denoted by η.

Convergence: The back propagation algorithm cannot, in general, be shown to converge, nor there are well-defined criteria for stopping their operation. Rather there are some reasonable criteria. To formulate such a criteria, let us assume the weight vector w^* denote a minimum. A necessary condition for w^* to be the minimum is that the gradient vector $g(w)$ of the error surface with respect to the weight vector w be zero at $w=w^*$. Accordingly we may formulate a sensible convergence criterion for back propagation learning as the back propagation algorithm is considered to have converged when the Euclidean norm of the gradient vector reaches a sufficiently small gradient threshold.

Summary of the back propagation algorithm
Initialization
Presentation of training examples
Forward computation: Let a training example in the epoch be denoted by $[x(n)d(n)]$, with the input vector $x(n)$ applied to the input layer of sensory nodes and the

desired response vector d(n) presented to the output layer of computation nodes. Compute the activation potentials and function signals of the network by proceeding forward through the network, layer by layer. The net internal activity level $v_j^{(l)}$ (n) for neuron j in layer l is given by

$$v_j^{(l)}(n) = \sum_{i=0}^{p} w_{ji}^{(l)}(n)\, y_i^{(l-1)}(n) \qquad (2.2)$$

where $y_i^{(l-1)}$ is the functional signal of the neuron i in the previous layer (l-1) at iteration n, $w_{ji}^{(l)}$ is the synaptic weight of the neuron j in layer l fed from neuron i in layer (l-1) and p number of neurons in the previous layer. For i=0, we have $y_0^{(l-1)}$ and $w_{j0}^{(l)}$ (n) = $\theta_j^{(l)}$ (n) where $\theta_j^{(l)}$ is the threshold applied to the neuron j in layer l. Assuming the use of a logistic function for sigmoidal non-linearity, the function (output) signal of neuron j in layer l is given by

$$y_j^{(l)}(n) = \frac{1}{1+\exp(-v_j^{(l)}(n))} \qquad (2.3)$$

The error signal is given by

$$e_j(n) = d_j(n) - o_j(n) \qquad (2.4)$$

where $d_j(n)$ is the jth element of the desired response vector d(n) and

Backward computation: Compute the δ 's (the local gradient) of the network by proceeding backward layer by layer:
for neuron j in output layer L

$$\delta_j^{(L)}(n) = e_j^{(L)}(n) o_j(n)[1 - o_j(n)] \qquad (2.5)$$

for neuron j in hidden layer l

$$\delta_j^{(l)}(n) = y_j^{(l)}(n)[1 - y_j^{(l)}(n)] \sum_k \delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n) \qquad (2.6)$$

Feed forward phase computation: Let $(x_1, y_1), (x_2, y_2), \ldots, (x_p, y_p)$ represent the p vector-pairs used to train the network where $x_i \in R^N$, $y_i \in R^M$. Assume a Back Propagation Feed Forward Network with an input layer, output layer and only one hidden layer. Consider an input vector $x_p = (x_{p1},\ldots, x_{pn})$ be applied to the input layer of the network. The input units distribute the values to the hidden layer units.

The net input to jth hidden unit is

$$net_{pj}^{h} = \sum_i w_{ji}^{h} x_{p_i} + \theta_j^{h} \qquad (2.7)$$

where 'h' superscript refers to the quantities on the hidden layer, w_{ji}^h is the weight on connection from ith input unit to the jth hidden unit and θ_j^h is the bias term. Assume that the activation of this node is equal to the net input, then the output of this node is given by

$$i_{pj} = f_j^h(net_{pj}^h) \qquad (2.8)$$

The net input and output for the kth output node are

$$net_{pk}^{o} = \sum_j w_{kj}^{o} i_{pj} + \theta_k^{o} \qquad (2.9)$$

$$o_{pk}^{o} = f_k^{o}(net_{pk}^{o}) \qquad (2.10)$$

where the superscript 'o' refers to the quantities in the output layer and w_{kj}^o is the weight on the connection between the jth hidden unit and the kth output unit. The loss at a single output unit is $(y_{pk}-o_{pk})$ where y_{pk} is the desired output value and o_{pk} is the actual output value from the kth unit, for the pth input. Let the loss minimized by the exponentiated gradient algorithm be the sum of the square of losses for all the output units.

$$\text{Total Loss}: L_p = 1/2 \sum_k (y_{pk} - o_{pk})^2 \qquad (2.11)$$

The weight changes are proportional to the exponent of the gradient of L_p in this algorithm. Calculate the gradient of L_p with respect to output layer weights:

Using Eq. 2.11 and chain rule for partial derivatives we have

$$\frac{\partial L_p}{\partial w_{kn}^{o}} = -(y_{pk} - o_{pk}) \frac{\partial o_{pk}}{\partial net_{pk}^{o}} \frac{\partial net_{pk}^{o}}{\partial w_{kj}^{0}} \qquad (2.12)$$

but

$$\frac{\partial net_{pk}^{o}}{\partial w_{kj}^{o}} = i_{pj}$$

Also simply writing

$$\frac{\partial o_{pk}}{\partial net_{pk}^{o}}$$

as $f^{o'}_k$ in (2.12), we get

$$\frac{\partial L_p}{\partial w^o_{kj}} = -(y_{pk} - o_{pk}) f^{o'}_k (net^o_{pk}) i_{pj} \qquad (2.13)$$

If the output function is linear, i.e., $f^{o'}_k$ is linear

$$f^{o'}_k = 1 \qquad (2.14)$$

If the output function is sigmoid, i.e.,
$f^o_k(net^o_{pk}) = (1 + e^{-net^o_{pk}})^{-1}$, then

$$f^{o'}_k(net^o_{pk}) = f^o_k(1 - f^o_k) = o_{pk}(1 - o_{pk}) \qquad (2.15)$$

If the output function $f^{o'}_k$ is linear, then

$$\frac{\partial L_p}{\partial w^o_{kj}} = -(y_{pk} - o_{pk}) i_{pj} \qquad (2.16)$$

If the output function f^O_k is a sigmoid function, then

$$\frac{\partial L_p}{\partial w^o_{kj}} = -(y_{pk} - o_{pk}) o_{pk}(1 - o_{pk}) i_{pj} \qquad (2.17)$$

Similarly calculate the gradient of E_p with respect to hidden layer weights:

$$\frac{\partial L_p}{\partial w^h_{ji}} = 1/2 \frac{\partial[\sum_k (y_{pk} - o_{pk})^2]}{\partial w^h_{ji}} = -\sum_k (y_{pk} - o_{pk}) \frac{\partial o_{pk}}{\partial w^h_{ji}}$$

We see that,

o_{pk} depends on net^o_{pk} from (2.10),
net^o_{pk} depends on i_{pj} from (2.9),
i_{pj} depends on net^h_{pk} pj from (2.8),
net^h_{pk} depends on w^h_j from (2.7).

Hence,

$$\frac{\partial L_p}{\partial w^h_{ji}} = -\sum_k (y_{pk} - o_{pk}) \frac{\partial o_{pk}}{\partial net^o_{pk}} \frac{\partial net^o_{pk}}{\partial i_{pj}} \frac{\partial i_{pj}}{\partial net^h_{pj}} \frac{\partial net^h_{pj}}{\partial w^h_{ji}} \qquad (2.18)$$

From Eq. 2.10,

$$\frac{\partial o_{pk}}{\partial net^o_{pk}} = f^{o'}_k(net^o_{pk})$$

From Eq. 2.9,

$$\frac{\partial net^o_{pk}}{\partial i_{pj}} = w^o_{kj}$$

From Eq. 2.8,

$$\frac{\partial i_{pj}}{\partial net^h_{pj}} = f^{h'}_j(net^h_{pj})$$

From Eq. 2.7,

$$\frac{\partial net^h_{pj}}{\partial w^h_{ji}} = x_{p_i}$$

Substituting the above values in Eq. (2.18), we get

$$\frac{\partial L_p}{\partial w^h_{ji}} = -\sum_k [(y_{pk} - o_{pk}) f^{o'}_j(net^o_{pk}) w^o_{kj} f^{h'}_j(net^h_{pj}) x_{p_i}] \qquad (2.19)$$

In the hidden layer, if f^h_j (net^h_{pj}) linear, then f^h_j $(net^h_{pj}) = 1$
If f^h_j (net^h_{pj}) is a sigmoid function then

$$f^{h'}_j(net^h_{pj}) = i_{pj}(1 - i_{pj})$$

Similarly in the output layer,
If f^o_k (net^O_{pk}) is linear, then $f^{O'}_k$ $(net^O_{pk}) = 1$
If f^O_k (net^O_{pk}) is a sigmoid function then

$$f^{o'}_k(net^o_{pk}) = o_{pk}(1 - o_{pk})$$

If both the output functions are linear, then Eq. 2.19 becomes

$$\frac{\partial L_p}{\partial w^h_{ji}} = -\sum_k [(y_{pk} - o_{pk}) w^o_{kj} x_{p_i}] \qquad (2.20)$$

If both the output functions are sigmoid functions, then Eq. 2.19 becomes

$$\frac{\partial L_p}{\partial w^h_{ji}} = -\sum_k [(y_{pk} - o_{pk}) o_{pk}(1 - o_{pk}) w^o_{kj} i_{pj}(1 - i_{pj}) x_{p_i}] \qquad (2.21)$$

If hidden layer output function is linear and output layer output function is a sigmoid function, then Eq. 2.19 becomes

$$\frac{\partial L_p}{\partial w^h_{ji}} = -\sum_k [(y_{pk} - o_{pk}) o_{pk}(1 - o_{pk}) w^o_{kj} x_{p_i}] \qquad (2.22)$$

Gradient descent algorithm: The convergence of Gradient Descent Algorithm for Linear predictors have been already discussed by Kivinen[12,13] and to obtain such a compromise is to minimize a function

$$U(w) = d(w_{t+1}, w_t) + \eta\, L(y, w_{t+1}.x) \qquad (3.1)$$

where the coefficient $\eta > 0$ is the importance given to correctiveness relative to conservativeness.

The Gradient descent algorithm minimizes the function

$$U(w) = d(w_{t+1}, w_t) + \eta\, L(y, w_{t+1}) \qquad (3.2)$$

where $d(w_{i+1}, w_t) = ½\ \|w_{i+1}, w_t\|^2_2$ the squared Eucledian distance, which is defined by $1/2(w_{t+1}-w_t)^2$ for all the components of the weight vectors.

Therefore, we have $dU(w)/dw = 0$. Substituting R.H.S. of Eq. 3.2 in the above Eq. we have

$$d[1/2\|w_{t+1} - w_t\|_2^2 + \eta L(y, w_{t+1})] = 0 \qquad (3.3)$$

Substituting for the squared Euclidean distance in the above Eq., we have

$$d[1/2(w_{t+1} - w_t)^2 + \eta L(y, w_{t+1})] = 0 \qquad (3.4)$$

Differentiating the above equation with respect to "t" w_{t+1}, we have

$$w_{t+1} - w_t + \eta L(y, w_{t+1}) = 0 \qquad (3.5)$$

Rearranging the above equation, we have

$$w_{t+1} = w_t - \eta L'(y, w_{t+1}) \qquad (3.6)$$

Algorithm (GD$_L$)
Parameters:
L : a Loss function from R×R to [0, ∝),
s : a start vector in R^N and
η : a learning rate in [0, ∝).

Initialization: Before the first trial, set $w_1 = s$.

Prediction: Upon receiving the tth instance x_t, give the prediction $o_t = w_t \cdot x_t$.

Update: Upon receiving the tth outcome y_t, update the weights according to the rule

$$w_{t+1} = w_t - \eta L'_{yt}(o_t)$$

To determine the direction in which to change the weights, we calculate the negative of the gradient of L_{yt} with respect to the weights, $w_{t,i}$. Then, we can adjust the values of the weights such that the total loss is reduced. Thus, the gradient descent algorithm updates the weight vector by subtracting from it the gradient $L'_{yt}(o_t)$ multiplied by the scalar η. The GD algorithm can therefore be seen as a straightforward application of the usual gradient descent minimization method to the on-line prediction problem.

Updating weights in the output layer: The weights on the output layer nodes are updated using,

$$w^o_{kj}(t+1) = w^o_{kj}(t) - \eta \frac{\partial L_p}{\partial w^o_{kj}} \qquad (3.7)$$

i.e.,

$$w^o_{kj}(t+1) = w^o_{kj}(t) - \eta(y_{pk} - o_{pk}) f^{o'}_k(net^o_{pk}) i_{pj} \qquad (3.8)$$

For the linear output, we have

$$w^o_{kj}(t+1) = w^o_{kj}(t) - \eta(y_{pk} - o_{pk}) i_{pj} \qquad (3.9)$$

For the sigmoidal output, we have

$$w^o_{kj}(t+1) = w^o_{kj}(t) - \eta(y_{pk} - o_{pk}) o_{pk}(1 - o_{pk}) i_{pj} \qquad (3.10)$$

Updating weights in the hidden layer: The weights on the hidden layer nodes are updated using,

$$w^h_{ji}(t+1) = w^h_{ji}(t) - \eta \frac{\partial L_p}{\partial w^h_{ji}} \qquad (3.11)$$

i.e.,

$$w^h_{ji}(t+1) = w^h_{ji}(t) + \eta \sum_k \begin{matrix}[(y_{pk} - o_{pk}) f^{o'}_j(net^o_{pk}) \\ w^o_{kj} f^{h'}_j(net^h_{pj}) x_{p_i}]\end{matrix} \qquad (3.12)$$

If both the output functions are linear, then Eq. 3.12 becomes

$$w^h_{ji}(t+1) = w^h_{ji}(t) + \eta \sum_k [(y_{pk} - o_{pk}) w^o_{kj} x_{p_i}] \qquad (3.13)$$

If both the output functions are sigmoidal, then Eq. 3.12 becomes

$$w_{ji}^{h}(t+1)=w_{ji}^{h}(t)+\eta\sum_{k}\begin{matrix}[(y_{pk}-o_{pk})o_{pk}(1-o_{pk})\\ w_{kj}^{o}i_{pj}(1-i_{pj})x_{p_i}]\end{matrix} \tag{3.14}$$

If the hidden layer output function is linear and output layer output function is sigmoidal, then equation Eq. 3.12 becomes

$$w_{ji}^{h}(t+1)=w_{ji}^{h}(t)+\eta\sum_{k}\begin{matrix}[(y_{pk}-o_{pk})o_{pk}\\ (1-o_{pk})w_{kj}^{o}x_{p_i}]\end{matrix} \tag{3.15}$$

EG algorithm: The EG algorithm results from using for d the relative entropy, also known as Kullback-Leibler divergence,

$$d(w,s)=\sum_{i=1}^{N}w_i\ln\frac{w_i}{s_i} \tag{3.16}$$

This assumes that all the components s_i and w_i are positive and the constraints $\sum_i s_i = \sum_i w_i = 1$ are maintained. The use of the relative entropy as a distance measure is motivated by the Maximum Entropy Principle of Jaynes and the more general Minimum Relative Entropy Principle of Kullback. These fundamental principles have many applications in Information Theory, Physics and Economics. In particular with some distance measure, we wish to guarantee the addition property $\sum^{n}_{i=1} w_i = 1$. This is done by the usual method of introducing a Lagrangian multiplier γ. Hence, instead of minimizing U we minimize $\tilde{U}$ defined by

$$\begin{aligned}\tilde{U}(w,\gamma)&=d(w,s)+\eta(L(y,s.x)+l_{y}'\\&(s.x)x(w-s))+\gamma((\sum_{i=1}^{N}w_i)-1)\end{aligned} \tag{3.17}$$

Setting the N+1 partial derivatives of $\tilde{U}$ to zero gives us the Eq. 3.45

$$\frac{\partial(w,s)}{w_i}+\eta L_{y}'(s.x)x_i+\gamma=0 \tag{3.18}$$

for i=1, …, N and the additional equation

$$\sum_{i=1}^{N}w_i=1. \tag{3.19}$$

Thus when the additional constraint $\sum^{n}_{i=1} w_i = 1$ is needed, we solve for i=1, …, N the Eq. 4.18 and then apply in Eq. 4.19 to obtain the value for γ.

Consider now the relative entropy as the distance measure, which requires the constraint $\sum_i s_i = \sum_i w_i = 1$. For this case, the Eq. 4.18 becomes

$$\ln\frac{w_i}{s_i}+1+\eta L_{y}'(s.x)x_i+\gamma=0 \tag{3.20}$$

from which we obtain

$$w_i=s_i\exp(-\eta L_{y}'(s.x)x_i-1-\gamma) \tag{3.21}$$

Hence, $w_i = s_i r_i \exp(-\gamma-1)$ where

$$r_i=\exp(-\eta L_{y}'(s.x)x_i) \tag{3.22}$$

Applying Eq. 4.19, we obtain

$$\exp(-\gamma-1)=(\sum_{j}=1^{N}s_jr_j)^{-1} \tag{3.23}$$

Hence, the update rule is

$$w_i=\frac{s_ir_i}{\sum_{j=1}^{N}s_jr_j} \tag{3.24}$$

Note that the update rule keeps the weights w_i positive if the weights s_i are positive.

Algorithm (EGL(s,η))
Parameters:
L : a Loss function from R × R to [0, ∝),
s : a start vector, with $\sum^{n}_{i=1} s_i = 1$ and $s_i >= 0$ for all i and
η : a learning rate in [0, ∝).

Initialization: Before the first trial, set $w_i = s$.

Prediction: Upon receiving the t^th instance x_t, give the prediction $o_t = w_t.x_t$.

Update: Upon receiving the t^th outcome y_t, update the weights according to the rule

$$w_{t+1,i}=\frac{w_{t,i}r_{t,i}}{\sum_{j=1}^{N}w_{t,j}r_{t,j}} \tag{3.25}$$

where

$$w_{t+1,i} = \frac{w_{t,i} r_{t,i}}{\sum_{j=1}^{N} w_{t,j} r_{t,j}} \qquad (3.26)$$

As the GD algorithm, the EG algorithm has a loss function, start vector and a learning rate as its parameters. In the update of EG, each weight is multiplied by a factor $r_{t,i}$ that according to (4.26) is obtained by exponentiating the ith component

$$\frac{\partial L(y_t, w_t.x_t)}{\partial w_{t,i}} = L'_{yt}(w_t.x_t)x_i \qquad (3.27)$$

of the gradient of $L(y_t, w_t.x_t)$. After the multiplication, the weights are normalized, as shown in Eq. 4.25, so that they sum is equal to 1. The weights clearly never change sign. Hence the weight vector w_t of EG is always a probability vector, i.e., it satisfies $\sum_i w_{t,i} = 1$ and $w_{t,i} >= 0$ for all i. Therefore, the prediction $w_t.x_t$ is a weighted average of the input variables $x_{t,i}$ and w_t gives the relative weights of the components in this weighted average. This in contrast to the GD algorithm, where also the total weight $||w_t||_1$ can change. The fact that the weight vector is always a probability vector clearly restricts the abilities of EG to learn more general linear relationships. We shall soon see how these restrictions can be avoided by a simple reduction.

EG± algorithm

Algorithm $(EG^{\pm}_{L}(U,(s^+,s^-),\eta))$

Parameters:

L : a Loss function from R × R to [0, ∝),

U : the total weight of weight vectors, s^+ and

s^- : a pair of start vectors in $[0, 1]^N$, with $\sum^{n}_{i=1} (s^+_i + s^-_i)$ and

η : a learning rate in [0, ∝).

Initialization: Before the first trial, set $w^+_1 = Us^+$ and $w^-_1 = Us^-$.

Prediction: Upon receiving the tth instance xt, give the prediction $o_t = w_t.x_t$.

Update: Upon receiving the tth outcome y_t, update the weights according to the rules

$$w^+_{t+1,i} = U\frac{w^+_{t+1,i} r^+_{t,i}}{\sum_{j=1}^{N} w^+_{t,j} r^+_{t,j} + w^-_{t,j} r^-_{t,j}} \qquad (3.28)$$

$$w^-_{t+1,i} = U\frac{w^-_{t+1,i} r^-_{t,i}}{\sum_{j=1}^{N} w^+_{t,j} r^+_{t,j} + w^-_{t,j} r^-_{t,j}} \qquad (3.29)$$

where

$$r^+_{t,i} = \exp(-\eta U L'_{yt}(o_t)) \qquad (3.30)$$

$$r^-_{t,i} = \exp(-\eta U L'_{yt}(o_t)) = \frac{1}{r^+_{t,i}} \qquad (3.31)$$

The EG±algorithm can best be understood as a way to generalize the EG algorithm for more general weight vectors by using a reduction. Given a trial sequence S, let S' be a modified trial sequence obtained from S by replacing each instance x_t by x_t'=(Ux_1, …, Ux_N, -Ux_1, …, -Ux_N). Hence, the number of dimensions is doubled. For a start vector pair (s^+, s^-) for EG±, let s = (s^+_1,..., s^+_N, s^-_1,..., s^-_N). Consider using EG± (U,(s^+,s^-),η) on a trial sequence S and using EG(s,η) on the trial sequence S'. If we let W'_t be the tth weight vector of EG (s,η) on the trial sequence S', so it is easy to see that Uw'_t = ($w^+_{t,1}$,..., $w^+_{t,N}$, $w^-_{t,1}$,..., $w^-_{t,N}$) holds for all t and, therefore, $W'_t.X'_y$ = (W^+_t - W^-_t).X_t. Hence, the predictions of EG± on S and EG on S' are identical, so EG±is a result of applying a simple transformation to EG. This transformation leads to an algorithm that in effect uses a weight vector W^+_t-W^-_t, which can contain negative components. Further by using the scaling factor U, we can make the weight vector W^+_t-W^-_t range over all vectors W ∈ R for which $||W||_1$ <= U. Although $\|W^+_t\|$ + $\|W^-_t\|$, is always exactly U, vectors W^+_t-W^-_t with $\|W^+_t W^-_t\|$ <Usimply result from having both $w^+_{t,i}$>0 and $w^-_{t,i}$>0 for some i. The parameters of EG± are a loss function L, a scaling factor U, a pair (s^+, s^-) of start vectors in $[0, 1]^N$ with $\sum^{N}_{i=1}$ (s^+_i + s^-_i) and a learning rate η.

Updating weights in the output layer: The weights on the output layer nodes are update using,

$$w^o_{kj}(t+1) = w^{+o}_{kj}(t+1) - w^{+o}_{kj}(t+1) \qquad (3.32)$$

where

$$w^{+o}_{kj}(t+1) = U\frac{r^{+o}_{kj}(t) w^{+o}_{kj}(t)}{\sum_j [r^{+o}_{kj}(t) w^{+o}_{kj}(t) + r^{-o}_{kj}(t) w^{-o}_{kj}(r)]} \qquad (3.33)$$

$$r^{+o}_{kj}(t) = e^{U\eta \frac{\partial L_p}{\partial w^o_{kj}}} \qquad (3.34)$$

$$r_{kj}^{+o}(t) = e^{-U\eta\sum_{k}[(y_{pk}-o_{pk})f_k^{o'}(net_{pk}^{o})i_{pj}]} \qquad (3.35)$$

Similarly,

$$r_{kj}^{-o}(t) = e^{-U\eta\frac{\partial L_p}{\partial w_{kj}^{o}}} = \frac{1}{r_{kj}^{+0}(t)} \qquad (3.36)$$

i.e.,

$$r_{kj}^{-o}(t) = e^{U\eta\sum_{k}[(y_{pk}-o_{pk})f_k^{o'}(net_{pk}^{o})i_{pj}]} \qquad (3.37)$$

If the output function is linear, then from (3.14) we have

$$r_{kj}^{+o}(t) = e^{-U\eta\sum_{k}[(y_{pk}-o_{pk})i_{pj}]} \qquad (3.38)$$

If the output function is sigmoidal, then from (3.15) we have

$$r_{kj}^{+o}(t) = e^{-U\eta\sum_{k}[(y_{pk}-o_{pk})o_{pk}(1-o_{pk})i_{pj}]} \qquad (3.39)$$

$$w_{kj}^{-o}(t+1) = U\frac{r_{kj}^{-o}(t)w_{kj}^{-o}(t)}{\sum_j[r_{kj}^{+o}(t)w_{kj}^{+o}(t)+r_{kj}^{-o}(t)w_{kj}^{-o}(t)]} \qquad (3.40)$$

In the Eq. 3.33 and 3.40, U is the scaling factor representing the total weight on output layer nodes. In the weight updating formulas above, we can note that the component of gradient appears in the exponent of the factor that multiplies the weight values.

Updating the weights in the hidden layer: The weights on the hidden layer nodes are updated using,

$$w_{ji}^{h}(t+1) = w_{ji}^{+h}(t+1) - w_{ji}^{-h}(t+1) \qquad (3.41)$$

where

$$w_{ji}^{+h}(t+1) = U\frac{r_{ji}^{+h}(t)w_{ji}^{+h}(t)}{\sum_i[r_{ji}^{+h}(t)w_{ji}^{+h}(t)+r_{ji}^{-h}(t)w_{ji}^{-h}(t)]} \qquad (3.42)$$

$$r_{ji}^{+h}(t) = e^{U\eta\frac{\partial L_p}{\partial w_{ji}^{h}}} \qquad (3.43)$$

$$r_{ji}^{+h}(t) = e^{-u\eta\sum_k[(y_{pk}-o_{pk})f_k^{o'}(net_{pk}^{o})w_{kj}^{o}f_j^{h'}(net_{pj}^{h})x_{p_i}]} \qquad (3.44)$$

Similarly,

$$r_{ji}^{-h}(t) = e^{-U\eta\frac{\partial L_p}{\partial w_{ji}^{h}}} = \frac{1}{r_{ji}^{+h}(t)} \qquad (3.45)$$

i.e.,

$$r_{ji}^{-h}(t) = e^{u\eta\sum_k[(y_{pk}-o_{pk})f_k^{o'}(net_{pk}^{o})w_{kj}^{o}f_j^{h'}(net_{pj}^{h})x_{p_i}]} \qquad (3.46)$$

If both the output functions are linear, then from Eq. 3.14 we have

$$r_{ji}^{+h}(t) = e^{-u\eta\sum_k[(y_{pk}-o_{pk})w_{kj}^{o}x_{p_i}]} \qquad (3.47)$$

If both the output functions are sigmoidal, then from Eq. 3.15 we have

$$r_{ji}^{+h}(t) = e^{-u\eta\sum_k[(y_{pk}-o_{pk})o_{pk}(1-o_{pk})w_{kj}^{o}i_{pj}(1-i_{pj})x_{p_i}]} \qquad (3.48)$$

If the hidden layer output is linear and the output layer output is sigmoidal, then from Eq. 3.16 we have

$$r_{ji}^{+h}(t) = e^{-u\eta\sum_k[(y_{pk}-o_{pk})o_{pk}(1-o_{pk})w_{kj}^{o}x_{p_i}]} \qquad (3.49)$$

$$w_{ji}^{+h}(t+1) = U\frac{r_{ji}^{-h}(t)w_{ji}^{-h}(t)}{\sum_i[r_{ji}^{+h}(t)w_{ji}^{+h}(t)+r_{ji}^{-h}(t)w_{ji}^{-h}(t)]} \qquad (3.50)$$

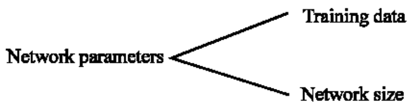
In the Eq. 3.42 and 3.50, U is the scaling factor representing the total weight on hidden layer nodes.

Experimental verification

BPN simulator: A customized Back Propagation Network (BPN) simulator was newly coded in C to test and benchmark the Gradient Descent and Exponentiated Gradient Descent versions of the algorithms. A three layer BPN was constructed, constituting the input layer, hidden layer and the output layer. In this network model, the input units are fan-out processors only. That is, the units in the input layer perform no data conversion on the network input pattern. They simply act to hold the components of the input vector within the network structure. Thus, the training process begins when an externally provided input pattern is applied to the input layer of units. Forward signal propagation then occurs. Once an output has been calculated for every unit in the network, the values computed for the units in the output layer are compared to the desired output pattern, element by element. At each output unit, an error value is calculated. These error terms are then fed back to all other units in the network structure and their connection weights are changed by using the respective weight updation formulas. Several assumptions have been incorporated into the design of this simulator. First, the output function on all hidden-layer and output-layer units is assumed to be the sigmoidal function. This assumption imply the need to store weight updates at one iteration, for use on the next iteration. Second, the bias values have not been included in the calculations. In a BPN, signals flow bi-directionally, but in only one direction at a time. During training, there are two types of signals present in the network: during the first half-cycle, modulated output signals flow from output layer to input layer. In the production mode, only the feed forward, modulated output signal is utilized. The program has been written in such a way to run either in the learning (training) phase or in the production (testing) phase.

Data structures for GD and EGD:

```
record BPN =
      InputLayer : LAYER* { locate input layer units }
      OutputLayer : LAYER* {locate output layer units }
      HiddenLayer : LAYER* {locate hidden layer units }
      Eta : float { learing rate }
end record;
record LAYEREGD =
      Units : integer { number of units in the layer }
      NetInput : float* { locate input array }
      Output : float* { locate output array }
      Weight : float** { locate connection weights }
end LAYER;
record LAYERGD =
      Units : integer { number of units in the layer }
      NetInput : float* { locate input array}
      Output : float* { locate output array }
      Weight : float** { locate connection weights }
      Wplus : float** { store W+ vector values }
      Wminus : float** { store W- vector values }
      U : float { total weight of weight vectors }
end LAYER;
```



Weights and learning parameters: Weights should be initialized to small, random values-say between ±0.5-as should the bias terms, θ_i that appear in the equations for the net input to a unit. It is common practice to treat this bias value as another weight, which is connected to a fictious unit that always has an output of 1.

$$net^o_{pk} = \sum_{j=1}^{L} w^o_{kj} i_{pj} + \theta^o_k$$

By making the definitions, $\theta \equiv w^0_{k(L+1)}$ and $i_{p(L+1)} \equiv 1$, we can write

$$net^o_{pk} = \sum_{j=1}^{L+1} w^o_{kj} i_{pj}$$

is treated just like a weight and it participates in the learning process as a weight. Another possibility is to remove the bias terms altogether and their use is optional.

Selection of a value for the learning rate parameter, η, has a significant effect on the network performance. Usually, η must be a small number-on the order of 0.05 to 0.25-to ensure that the network will settle to a solution. A small value of η means that the network has to make a large number of iterations. It is often possible to increase the size of η as learning proceeds. Increasing η as the network error decreases will often help to speed convergence by increasing the step size as the error reaches a minimum, but the network may bounce around too far from the actual minimum value if η gets too large.

Standard genetic code: The neural network model of the Genetic code is strongly correlated to GES scale of Amino Acid Transfer Free Energies Tolstrup[14]. In the neural network approach to studying the structure of the standard genetic code, the analysis is new and special in that it is unbiased and completely data driven. The neural network infers the structure directly from the mapping between the codons and amino acids as it is given in the standard genetic code. In the network that learns the genetic code, the input layer receives a nucleotide triplet and the output corresponds to the amino acid. Thus the 61 different triplets are possible as input and 20 different amino acid are possible as output. Here the triplets for the start and stop codons are ignored, hence 61 and not 64. The network has 12 input units, two or more hidden units and 20 output units.

The input layer encoded the nucleotide triplets as binary string comprising 3 blocks of 4 bits, with adenine as 0001, cytosine as 0010, guanine as 0100 and uracil as 1000. The output layer encoded the amino acid as a binary string of 20 bits. During the training the criterion for successful learning was that the activity of the corresponding output unit should be larger than the others. In each training epoch the codons were presented to the network in the random order Fig. 3.

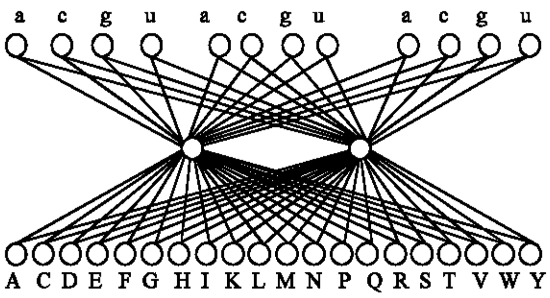


Fig. 3: Neural network trained to learn standard genetic code

A powerful way to have a low classification error was to have an adaptive training set, where the training examples are included and excluded after determining whether they are classified correctly by the current network. Such a scheme may introduce more noise in the learning process, which helps to avoid the local minima. Introducing the noise in the training is usually done by updating the network after each example rather than each epoch.

A network with two hidden units was successfully trained. The learning was varied from 0.1 to 0.01 during different trials. The value of the network parameter U was heuristically assumed to be 70 for the hidden units and 40 for the output units. The Net Loss or Error was assumed to be a constant 0.001. There were 12 units in the input layer and 20 units in the output layer. All the data were feed in the binary format and the output too was encoded in the binary format making it a sparse dataset, making it highly suitable for the Exponentiated Gradient Descent weight update rule.

RESULTS AND DISCUSSION

The above mentioned problem was tested against the Gradient Descent weight update rule and the Exponentiated Gradient Descent weight update. It can be observed from Table 1 and Table 2 that the experimental data were used and the data mapping was done by using the Standard Genetic code by giving the inputs as listed in the column and the corresponding output is obtained.

Table 1: Standard genetic code-data mapping

Triplet	Input	AA	AA (sym)	AA (val)	Output
UUU	100010001000	Phe	F	5	00001000000000000000
UUC	100010000010	Phe	F	5	00001000000000000000
UUA	100010000001	Leu	L	10	00000000010000000000
UUG	100010000100	Leu	L	10	00000000010000000000
UCU	100000101000	Ser	S	16	00000000000000010000
UCC	100000100010	Ser	S	16	00000000000000010000
UCA	100000100001	Ser	S	16	00000000000000010000
UCG	100000100100	Ser	S	16	00000000000000010000
UAU	100000011000	Tyr	Y	20	00000000000000000001
UAC	100000010010	Tyr	Y	20	00000000000000000001
UAA	100000010001	Sto	NA	NA	NA
UAG	100000010100	Sto	NA	NA	NA
UGU	100001001000	Cys	C	2	01000000000000000000
UGC	100001000010	Cys	C	2	01000000000000000000
UGA	100001000001	Sto	NA	NA	NA
UGG	100001000100	Trp	W	19	00000000000000000010
CUU	001010001000	Leu	L	10	00000000010000000000
CUC	001010000010	Leu	L	10	00000000010000000000
CUA	001010000001	Leu	L	10	00000000010000000000
CUG	001010000100	Leu	L	10	00000000010000000000
CCU	001000101000	Pro	P	13	00000000000010000000
CCC	001000100010	Pro	P	13	00000000000010000000
CCA	001000100001	Pro	P	13	00000000000010000000
CCG	001000100100	Pro	P	13	00000000000010000000
CAU	001000011000	His	H	7	00000010000000000000
CAC	001000010010	His	H	7	00000010000000000000
CAA	001000010001	Gln	Q	14	00000000000001000000
CAG	001000010100	Gln	Q	14	00000000000001000000
CGU	001001001000	Arg	R	15	00000000000000100000
CGC	001001000010	Arg	R	15	00000000000000100000
CGA	001001000001	Arg	R	15	00000000000000100000
CGG	001001000100	Arg	R	15	00000000000000100000
AUU	000110001000	Ile	I	8	00000001000000000000
AUC	000110000010	Ile	I	8	00000001000000000000
AUA	000110000001	Ile	I	8	00000001000000000000
AUG	000110000100	Sta	NA	NA	NA
ACU	000100101000	Thr	T	17	00000000000000001000
ACC	000100100010	Thr	T	17	00000000000000001000

Table 2: Standard genetic code-data mapping contd

Triplet	Input	AA	AA (sym)	AA (val)	Output
ACA	000100100001	Thr	T	17	00000000000000001000
ACG	000100100100	Thr	T	17	00000000000000001000
AAU	000100011000	Asn	N	12	00000000000100000000
AAC	000100010010	Asn	N	12	00000000000100000000
AAA	000100010001	Lys	K	9	00000000100000000000
AAG	000100010100	Lys	K	9	00000000100000000000
AGU	000101001000	Ser	S	16	00000000000000010000
AGC	000101000010	Ser	S	16	00000000000000010000
AGA	000101000001	Arg	R	15	00000000000000100000
AGG	000101000100	Arg	R	15	00000000000000100000
GUU	010010001000	Val	V	18	00000000000000000100
GUC	010010000010	Val	V	18	00000000000000000100
GUA	010010000001	Val	V	18	00000000000000000100
GUG	010010000100	Val	V	18	00000000000000000100
GCU	010000101000	Ala	A	1	10000000000000000000
GCC	010000100010	Ala	A	1	10000000000000000000
GCA	010000100001	Ala	A	1	10000000000000000000
GCG	010000100100	Ala	A	1	10000000000000000000
GAU	010000011000	Asp	D	3	00100000000000000000
GAC	010000010010	Asp	D	3	00100000000000000000
GAA	010000010001	Glu	E	4	00010000000000000000
GAG	010000010100	Glu	E	4	00010000000000000000
GGU	010001001000	Gly	G	6	00000100000000000000
GGC	010001000010	Gly	G	6	00000100000000000000
GGA	010001000001	Gly	G	6	00000100000000000000
GGG	010001000100	Gly	G	6	00000100000000000000

The dataset used was a sparse target vector (a input vector mostly represented by zeros). The learning rate parameter was changed constantly in increments of 0.01 and the performance of the network was monitored. The number of hidden layer units were also varied for the various learning rates and the performance of the network was monitored. Finally the value of the parameter U of the Exponentiated Gradient Descent weight update rule was varied to obtain an optimal results. The numerical values thus obtained are listed in Appendix A.

It is observed from Table 3 that the number of iterations taken by the network with Exponentiated Gradient Descent weight update rule, to converge for the given sparse dataset is far less when compared to the network with Gradient Descent weight update rule. From Table 4 it can be noted that the execution time is reduced and the predictability is constant invariant of the number of iterations. It is found from Table 5 that the proportionally the execution time taken for the network is also reduced. But one notable feature is that as additional calculations are required in Exponentiated Gradient Descent weight update rule, the time reduction is only by a factor. The time taken for one epoch in Exponentiated Gradient weight update rule is not equal to the time taken for one epoch in Gradient Descent weight update rule. The accuracy of the network or the predictability of the network also remains a constant for the various learning rates. The net loss or the net error value also drastically

Table 3: NET loss for GD and EGD

Iteration	GD	EGD	Iteration	GD	EGD
10	0.8149	0.2347	170	0.3664	0.0053
20	0.6199	0.1934	180	0.3647	0.0045
30	0.5134	0.1670	190	0.3630	0.0039
40	0.4574	0.1280	200	0.3613	0.0034
50	0.4269	0.0973	210	0.3597	0.0030
60	0.4091	0.0896	220	0.3581	0.0027
70	0.3982	0.0840	230	0.3565	0.0025
80	0.3909	0.0746	240	0.3549	0.0023
90	0.3858	0.0642	250	0.3533	0.0021
100	0.3819	0.0553	260	0.3518	0.0019
110	0.3788	0.0466	270	0.3502	0.0018
120	0.3762	0.0332	280	0.3486	0.0017
130	0.3739	0.0186	290	0.3471	0.0016
140	0.3719	0.0116	300	0.3455	0.0015
150	0.3699	0.0084	310	0.3440	0.0015
160	0.3681	0.0065	320	0.3425	0.0014

Table 4: Gradient descent algorithm-results

Learning rate	Execution time	No. of iterations	Predictability
0.01	180.05	100602	1
0.015	120.10	67149	1
0.02	90.27	50422	1
0.025	72.30	40387	1
0.03	60.16	33695	1
0.035	51.64	28914	1
0.04	45.27	25330	1
0.045	40.21	22539	1
0.05	36.30	20308	1
0.055	32.96	18481	1
0.06	30.21	16957	1
0.065	27.96	15668	1
0.07	25.98	14562	1
0.075	24.23	13608	1
0.08	22.74	12769	1
0.085	21.42	12024	1
0.09	20.21	11369	1
0.095	19.23	10778	1
0.10	18.18	10248	1

Table 5 Exponentiated Gradient Descent Algorithm-Results

Learning rate	Execution time	No. of iterations	Predictability
0.01	3.02	320	1
0.015	1.97	200	1
0.02	1.37	137	1
0.025	1.31	137	1
0.03	1.09	123	0.9
0.035	0.54	59	0.8
0.04	0.54	55	0.8
0.045	0.60	58	0.8
0.05	0.32	36	0.8
0.055	0.38	41	0.8
0.06	0.32	38	0.8
0.065	0.76	90	0.9
0.07	0.65	66	0.6
0.075	0.82	98	0.9
0.08	0.65	68	0.8
0.085	0.82	101	0.5
0.10	1.48	175	0.5

reduces in the Exponentiated Gradient Descent weight update rule, which is the major factor in reducing the execution time for the training of any network.

CONCLUSION

Back propagation algorithm has been a major driving force in the field of bioinformatics in the past two decades. Although many new algorithms like Recurrent feed forward networks, Hopfield Networks, Boltzman networks, Genetic Algorithms and Support Vector machines are introduced, backpropagation algorithm has displayed its robustness over the years. Most of the secondary structure prediction and gene prediction still problems rely on backpropagation algorithm. A new machine learning method has been proposed which would have a wider impact on the huge range of problems in bioinformatics that use backpropagation algorithm. In the proposed algorithm a new weight update rule has been introduced Exponentiated Gradient Descent weight update rule that substitutes the existing Gradient Descent weight update rule of the backpropagation algorithm.

All the problems in the field of Bio-Informatics deals with diverse set of data, these data most often have a sparse or binary representation as binary sequences for a easy encoding and decoding scheme. The proposed algorithm has been proved perform well on any given sparse dataset. The effectiveness of the new weight update rule on the backpropagation algorithm has been demonstrated by its performance in pattern recognition problem. Moreover the rate of convergence of the network has been greatly reduced, resulting in a faster and an efficient learning network.

Some of the open issues that may be dealt in the future are

- Applying the new Exponentiated Gradient Descent weight update rule on a wider range of problems.
- Using the new rule for large-scale genome analysis, as this consumes more time.
- A statistical analysis of the prediction by the new algorithm has to be carried out.
- Improving the predictability of the algorithm for non-sparse target vectors i.e., when data is feed in decimal form and not in binary form.

REFERENCES

1. Richard Durbin *et al.*, 1988. Biological sequence analysis, Cambridge University Press.
2. Pierre Baldi, Soren Brunak and Bioinformatics, 2001. The machine learning approach, MIT Press.
3. Aleksander, I. and H. Morton, 1990. An Introduction to Neural Computing, Chapman and Hall, London.
4. Simon Haykin, 1994. Neural Networks: A Comprehensive Foundation, Macmillan Publishing Company, New Jersey.
5. David W. Mount, 2001. Bioinformatics: Sequence and Genome Analysis, Cold Spring Harbor Laboratory Press.
6. Qian, N. and T.J. Sejnowski, 1988. Predicting the secondary structure of globular proteins using neural network models, J. Mol. Biol., 202: 865-884.
7. Michale S. Waterman, 1995. Introduction to computational Biology, Chapman and Hall.
8. Michael J.E. Sternberg, 1996. Protien Structure Prediction, Oxford University Press.
9. James A. Freeman and David M. Skapura, 1991. Neural Networks: Algorithms, Applications and Programming Techniques, Addison Wesley.
10. Hill, S.I. and R.C. Williamson, 2001. Convergence of Exponentiated Gradient Algorithms, IEEE Trans. Signal Processing, 49: 1208-1215.
11. Kivinen, J. and M.K. Warmuth, 1997. Additive versus Exponentiated Gradient Descent for Linear Predicorts, Inform. Comput., 132: 1-64.
12. Kivinen, J. and M.K. Warmuth, 1994. Exponentiated Gradient Versus Gradient Updates for Linear Predictions, Technical Report UCSC-CRL-94-16, University of California, USA.
13. Tolstrup, N., J. Toftgard, J. Englebrecht and S. Brunak, 1994. Neural Network Model of the Genetic Code is Strongly Correlated to the GES Scale of Amino Acid Transfer Free Energies. J. Mol. Biol., 243: 816-820.