Artificial Neural Networks

Andrew Cortis

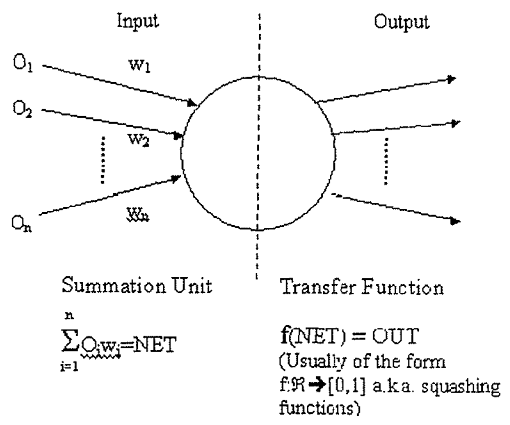
What is an *Artificial Neural Network*?

It is a Parallel Computational Network made up of interconnected neurons. They are biologically inspired, i.e. they are composed of elements that work analogously to the most elementary functions of the biological neuron. Despite the similarities, the actual *"intelligence"* exhibited by the most sophisticated artificial neural networks, is still very limited. Each Neuron performs a function on its input. Each neuron passes its output on to another neuron to allow it to perform its work. Artificial Neural Networks can modify their behavior in response to their environment: given a set of inputs (perhaps along with the desired outputs), they can self-adjust to produce consistent responses.

An Artificial Neuron

This is a simple processing element. It is the basic processing unit of an artificial neural network.

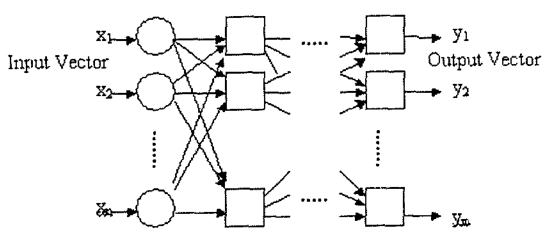
The Neuron will pass all the inputs through the Summation Unit Producing



the NET value and then apply the transfer function to the NET, to produce the output.

An Artificial Neural Network

Input Vector $\underline{x} = \{x_1, x_2, \ldots, x_n\}$



Output Vector $\underline{y} = \{y_1, y_2, \cdots, y_m\}$
Logical Layout of an interconnected feed-forward Artificial Neural Network

Training an Artificial Neural Network

The objective of training an artificial neural network is so to produce the desired(or a consistent) set of outputs after application of a set of particular inputs produces the desired (or a consistent) set of outputs. Like the brain structure these networks mimic, they keep a certain degree of unpredictability and unless every input is tried separately one might not be certain of the precise output. It might be impractical and costly to try an exhaustive search on a large network.

Supervised Training

By giving the Artificial Neural Network input and expected output pairs, it can be trained to 'learn' a particular problem. **Error-Back Propagation** uses a supervised training algorithm. When in training mode, each input vector is supplied with a corresponding desired output vector (the set being called a training pair). The output of the network is worked out and the error is fed back through the network and the weights are adjusted accordingly. Before the training begins, the weights are assigned to small random numbers.

The following steps are taken to train the network:

1. The next training pair is chosen and the input vector is applied to the network and the output is worked out.
2. The error is worked out.
3. The weights are adjusted so as to minimize the error.
4. These steps are repeated for several other training pairs, for several times until all the training pairs have a suitably low error ($< \epsilon$, where ϵ is the parameter called the error rate)

Calculating the Error Vector:
A simple way to calculate the error vector is the following:
Error Vector $\underline{e} = \{e_1, e_2, \ldots, e_n\}$ and
Target Vector $\underline{t} = \{t_1, t_2, \ldots, t_n\}$
$\underline{e} = \underline{t} - \underline{y}$
Other methods use the square of the difference as error.

Adjusting the weights:

Adjusting is accomplished layer by layer, from the output layer inwards. A learning rate, μ is a value, $0 \leq \mu \leq 1$, used as a parameter for the artificial neural network.

Output Layer:

The output layer is adjusted first. For each neuron p with an output to a neuron q in the output layer k, work out δ, where:

$$\delta_{pk} = \frac{df(y_p).e_p}{dy_p}$$

Then the change for the weight,

$$\Delta w_{pq,k} = \mu\delta_{qk}.y_{p,(k-1)}$$

Where $y_{p,(k-1)}$ is the output of the $p^(th)$ neuron in the $(k-1)^(th)$ layer. This represents the modification to be applied in order to minimize the error.

$$w_{pq,k}(n+1) = w_{pq,k}(n) + \Delta w_{pq,k}$$

$w_{pq,k}(n)$ is the value of a weight from a neuron p to a neuron q in the output layer k, in the $n^(th)$ step of learning.

Hidden Layers:

Since the hidden layers have no target vector, the adjustment algorithm for the output vector cannot be used. The following one can be used instead:
Calculate δ, s.t.

$$\delta_{p,(k-1)} = \frac{df(y_{p,(k-1)})}{dy_{p,(k-1)}}.(\Sigma_q \delta_{p,k} w_{pq,k})$$

Since

$$\Delta w_{pq,k} = \mu\delta_{qk}.y_{p,(k-1)}$$

$$w_{pq,k}(n+1) = w_{pq,k}(n) + \Delta w_{pq,k}$$

Results

With further studies, by using improvements on the back-propagation algorithm it can be made to run "quite fast on practical applications. So claims that Back-Propagation is slow should be carefully analyzed with these ideas in mind" - Françoise Fogelman Soulie

Furthermore "Neural Network architectures and algorithms have progressively evolved from simple techniques to ... more complex architecture" where they can be trained very fast and used at their best with only a limited number of training tokens.

Sometimes, finding optimal weights for a network is "intractable",according to Edoardo Amaldi: "Since the learning of the problem is at least as hard as its decision version, the problem of the training perceptions ... is also NP-complete."

Therefore if P $\neq$ NP, there is no algorithm that can find the optimal weights in polynomial time.

This result is the reason why no known algorithm can yield an optimal weight setting in polynomial time and compels us to develop efficient heuristic methods with good average behavior.

"Back-propagation suffers from the drawback of the computational burden of training the network." - E. Monte, J. Arcusa, J.B. Harino, E.Lleida.

Many authors of recent papers have tried to devise their methods for accelerating the convergence rate of the algorithm and presented ideas to help improve the performance of the networks.

Devising improvements for an Artificial Neural Network can be deceptively simple, as was shown in 1987 by Stornetta and Hubermann: The conventional 0-1 dynamic range of inputs is not at its most advantageous. Because the weight adjustment $\Delta w_{pq,k}$ is proportional to the output level of the neuron, many inputs will be 0 and consequently will not train. A solution is to change the input range to $\pm\frac{1}{2}$ and add a bias to the squashing function.

A back-propagation network learns by making changes in its weights in a direction to minimize the errors between its result and the training data. The minimization is done using the steepest descent algorithm. In spite of how appealing such a solution is, there is no guarantee that the network can be trained in a finite amount of time. Also it is not certain that the network will converge to the best solution: local minima may trap the algorithm in an inferior solution.

The long uncertain training process in a complex problem might require days to train, and the effort might prove for lack of ... convergence at all. Long training time can be the result of a non-optimal step size while training failures arise from one of 2 sources: network paralysis and local minima.

Network Paralysis:

As the network trains, the weights can become adjusted to very large values. This can force all or most of the neurons to operate at large values of OUT, in a region where the derivative of the squashing function is very small. Since the adjustment on the weights is proportional to the derivative, the training process may come to a virtual standstill. Network Paralysis can be avoided by using a small step size μ, although this leads to an increase in training time.

Local Minima:

Back-propagation employs a type of gradient descent: i.e. it follows the slope of the error surface downwards, constantly adjusting the weights towards a minimum. The network can get trapped in a local minimum, impeding the network from converging. Wassermann has proposed a method that combines statistical methods with gradient descent of back-propagation to produce a system that finds global minima while retaining the higher training rate of the back-propagation.

Step Size:

The convergence proof of Rumelhart, Hinton and Williams shows that infinitesimally small weight adjustments are assumed. This is impractical as it implies infinite training time. It is necessary to select a finite step size. If the step size is too small, convergence will be too slow; if on the other hand it is too large, paralysis or continuous instability can result. Wassermann describes an adaptive step size algorithm intended to adjust step size automatically as training proceeds.

Temporal Instability:

If a network is learning from a training pair, it is not considered good if in doing so it forgets the previous one. The process should teach the network the entire set. Rumelhart's convergence accomplishes this but it requires showing the network all vectors in the training set before adjusting the weights.

Applications

Back-Propagation (amongst other types of network learning algorithms) is being used in a variety of applications. Improvements in the current systems such as the 'second order back-propagation' by Parker improve the convergence speed by using the second derivative to produce a more accurate estimate of the correct weight change. (Morever he has also shown that higher derivates are redundant) Also, more advanced techniques are being developed. Artificial Neural Networks have been applied in a variety of research applications such as the NEC's optical-character-recognition system, Sejnowski and Rosenberg's conversion of printed English into highly intelligible speech and Cottrell Munro and Zipser's successful image compression application.

References

[1] Philip D.Wassermann, *Neural Computing, Theory & Practice*, ANZA Research, Inc.

[2] Daniel Graupe, *Principles of Artificial Neural Networks*, World Scientific

[3] T. Kohonen, K.Makisara, O.Simula, J.Kangas, *Artificial Neural Networks, Volume 1 & 2*, North-Holland

[4] Stuart Russell, Peter Norvig, *Artificial Intelligence, A Modern Approach*, Prentice Hall Series in Artificial Intelligence

[5] P.G.J. Lisboa, *Neural Networks, current applications*

[6] S.Y. Kung, *Digital Neural Networks*