Efficient Neural Net α-β-Evaluators

Alois P. Heinz*
Institut für Informatik, Universität Freiburg
D-79104 Freiburg, Germany

1 Introduction

We describe a new artificial neural network for the conceptual realization of α-β-evaluators which are extensively used as the central part of many game playing programs. Then we present a very efficient implementation of this network that provides a speed-up of recall time in the order of magnitudes on conventional computers as compared to sequential implementations of traditional neural network architectures.

The α-β-evaluator is a function of three arguments, a game position p and the boundaries α and β of a real interval $[\alpha,\beta]$, also called the *α-β-window*. It returns a real value $v(p,\alpha,\beta) \in [\alpha,\beta]$ that may be considered as a measure of the real strength of the position adjusted to the nearest point of the α-β-window. During the evaluation inquiries on a set of *features* of p can be made for the cost of additional computation time. A strong demand on the shape of any α-β-evaluation function is that it should be smooth in all parameters to prevent the so-called *blemish effect* which deteriorates the quality of the overall decision making process heavily (Berliner, 1980).

2 Setting up the network

Conceptually the α-β-evaluator is realized by a feedforward artificial neural network. Finding the right network architecture is an example of a *supervised learning* problem. The set of training feature vectors $X_c = \{x_{c,1},\ldots,x_{c,m}\}^t$ with assignments of values v_c may be generated by an automatic procedure (Heinz et al., 1993). The topology of the network and the settings of its parameters are developed in a sequence of four steps.

The first step consists of generating from the training set a binary decision tree that is able to approximate the desired function. Each inner node k of the tree is labeled with an index k_i and a constant k_c and each leaf L is labeled with a value $v(L)$. Informally, when a new vector X is to be evaluated by the tree, a path from the root to a leaf is traced. At an inner node k the path is extended to the left, if $x_{k_i} - k_c \leq 0$, and to the right in the other case. At the uniquely defined leaf L the value $v(L)$ is returned. Any of the acknowledged tree building procedures (Mingers, 1989a, 1989b) that are based on methods from information theory and statistics may be used in this step.

In the second step the tree is transformed into a network N. The input layer of N consists of one neuron for each of the feature values and for the values of α and β. The *decision layer* consists of the decision neurons k, which correspond to the inner nodes of the tree. Each neuron k receives input only from the

*This research was partly supported by the Academy of Finland.

feature neuron for x_{k_i} and is equipped with a threshold k_c. It computes the *sharp decision function*

$$d_k(X) := \begin{cases} 0 & \text{if } x_{k_i} - k_c \leq 0, \text{ and} \\ 1 & \text{else.} \end{cases}$$

The neurons of the *AND layer* correspond to the leaf nodes L of the tree. They compute the product of the input values which they receive from all neurons in the decision layer that correspond to ancestor nodes of the corresponding leaf of the tree. Any connection between these two layers computes for a pair (k, L) the *decision contribution* $D_{k,L}(X)$ that is defined to be $d_k(X)$ if L belongs to the right subtree of k and $(1 - d_k(X))$ if L belongs to the left subtree. Each neuron L of the AND layer feeds its output via a connection with weight $v(L)$ into a OR neuron that computes the sum of all its inputs. The maximum of its output value and the value of α is computed by a MAX neuron. Then a MIN neuron computes the minimum of this value and the value of β as the output of the network.

In the third step the network that up to now only mimics the behavior of the decision tree is changed by replacing the sharp decision functions by softer versions. These are based on a real function f(x), that is constant 0 for $x \in (-\infty, -1)$, constant 1 for $x \in (1, \infty)$ and f(x) $= 1/2 + x - x\,|x|\,/2$ for $x \in [-1, 1]$. The *soft decision function* of neuron k is defined to be

$$d_k^*(X) := \mathrm{f}\left(\frac{x_{k_i} - k_c}{R_k}\right).$$

The strictly positive parameter R_k controls the *radius of softness* of neuron k. Larger values of R_k cause softer decisions. For $R_k = 4$ the decision function of k approximates very closely to the logistic function $\mathrm{t}(x) = \frac{1}{1+e^{-x}}$ with a maximal aberration of less than 3 %. The evaluation of the network N for an input vector X and α and β can be described by the formula

$$v_N(X, \alpha, \beta) = \min\left(\beta, \max\left(\alpha, \sum_{L \in \text{AND-layer}(N)} v(L) \prod_{k \in \text{pred}(L)} D_{k,L}^*(X)\right)\right),$$

where the set pred(L) contains all predecessor decision neurons of neuron L. The functions $D_{k,L}^*(X)$ are defined according to $D_{k,L}(X)$ but with the soft decision functions instead of the sharp ones.

The network's topology is now fully determined and with some choice of small R_k values its evaluations are close to the already good results of the tree. But in the fourth step the net is retrained on its free parameters, namely the c_k, R_k, and $v(L)$ values to improve the evaluation. Only a few steps of incremental adaption are needed as compared to randomly initialized nets. As training method the quickprop algorithm (Fahlman, 1988) is used because it shows faster convergence than back-propagation (Rumelhart et al., 1986) and does not have problems with the derivative of function f that is non-zero only in the interval $(-1, 1)$.

3 Implementing the network

The fastest implementation of our network would make use of extensive parallel hardware. But this is not always a feasible way, for several reasons. We

therefore present a sequential implementation of the α-β-evaluation network on a binary decision tree. We only give the details of the data structure and the evaluation algorithm. From this description it should become clear that the incremental learning algorithm can be carried out on the tree structure as well.

Using the same correspondence between network and tree as in the last section we only need to describe what information from the net is stored where in the tree and then state the evaluation algorithm. Each inner node k is labeled with the index k_i of the feature that is used in the decision, with the threshold k_c, and with the radius of softness R_k. And k has pointers to its left and right descendants k_l and k_r. Each leaf L of the tree is labeled with its value $v(L)$. Additionally, any node q is labeled with two values, q_α and q_β which are the minimum and maximum of all $v(L)$ values in the subtree of the tree with root q, respectively. The complete recursive evaluation algorithm is given as follows:

```
function evaluate (p: position; α, β: real; k: node): real;
begin
    if k_β ≤ α then return (α);
    if k_α ≥ β then return (β);
    if leaf (k) then return (v(k));
    if unknown [k_i] then begin
        x[k_i] := get_feature (p, k_i);
        unknown [k_i] := false
    end;
    d* := f ((x[k_i] − k_c)/R_k);
    if d* = 0 then return (evaluate (p, α, β, k_l));
    if d* = 1 then return (evaluate (p, α, β, k_r));
    α_l := max ((α − k_rβ · d*)/(1 − d*), −1);
    β_l := min ((β − k_rα · d*)/(1 − d*), +1);
    v_l := evaluate (p, α_l, β_l, k_l) · (1 − d*);
    α_r := max ((α − v_l)/d*, −1);
    β_r := min ((β − v_l)/d*, +1);
    v_r := evaluate (p, α_r, β_r, k_r) · d*;
    return (v_l + v_r)
end;
```

Initially this function is called with the root of the tree as last argument and the global vector of unknown-flags for the features completely set to true. This is to assure that each feature will be computed only if and only the first time it is needed. After the first reference it is stored in the vector x for repeated later reuse.

One idea of the algorithm is to use the fact that the $D^*_{k,L}(X)$ terms are hierarchically ordered. It's therefore sufficient to trace only those paths from the root to the leafs that proceed along branches with a non-zero decision contribution. The other idea exploits the available knowledge about the range of values that are contained in a given subtree with root k. If the intersection of this range $[k_\alpha, k_\beta]$ with the α-β-window is empty then further computation is needless and the nearest value of this window is returned. If in fact k is a leaf, its stored value $v(k)$ can be retuned. Otherwise the soft decision function $d^*_k(X)$ is computed. If it is 0 then the decision contribution of k and the leafs of the right subtree is 0. So only the evaluation of the left subtree has to be returned. The case of $d^*_k(X) = 1$ is handled symmetrically.

The interesting case is when $d_k^*(X)$ is somewhere between 0 and 1. Then both branches may contribute to the evaluation, the left one with a factor of $(1 - d_k^*(X))$ and the right one with a factor of $d_k^*(X)$. The α-β-windows of the two branches have to be adapted according to these scaling factors and because some pre-knowledge is available. In this case the sum of the evaluations of left and right subtrees is returned.

To estimate the algorithm's average case runtime we count the average number of branches $\ell_{ac}(n,p)$ traversed during an evaluation without α-β-cut-offs occurring as a function of the number of leafs n and p, which is assumed to be the probability that an inner node operates within its radius of softness. By imposing a balancing constraint during the tree building procedure it can be guaranteed that the height of the tree is limited by $c \log_2 n$ for some $c \geq 1$. Thence $\ell_{ac}(n,p)$ can be derived as

$$\ell_{ac}(n,p) = (1+p)\left\{ c \log_2 n + \binom{c \log_2 n}{2} p + \binom{c \log_2 n}{3} p^2 + \cdots \right\},$$

which grows slowly with n for small p. At any rate, $\ell_{ac}(n,p)$ is much smaller than the $O(n^2)$ steps required by a sequential implementation of the fully connected network with n neurons in the AND layer and the $O(nc \log_2 n)$ steps required for the sparsely connected network derived from a decision tree. A further speed-up is caused by α-β-cut-offs that even may occur at the root.

4 Conclusion

We have presented a new artificial neural network with a very efficient sequential implementation. The efficiency is achieved by mapping the network onto a tree and its algorithms that exploit the sparse connection structure, hierarchical connection ordering, vanishing decision contributions in the exterior of the radius of softness, and α-β-cut-offs. This strategy results in a system that combines the advantages of the neural net, namely smoothness, adaptability, and simplicity with the profits of the tree, namely efficiency.

References

Berliner, H. J. (1980). Backgammon computer program beats world champion — performance note. *Artificial Intelligence,* **14,** 205–220.

Fahlman, S. E. (1988). *An empirical study of learning speed in back-propagation networks* (CMU-CS-88-162). Pittsburgh, PA: Carnegie-Mellon University.

Heinz, A. P., & Hense, C. (1993). Bootstrap learning of α-β-evaluation functions. *Proceedings of the Fifth International Conference on Computing and Information (ICCI'93),* Sudbury, Canada, 1993.

Mingers, J. (1989a). An empirical comparison of selection measures for decision-tree induction. *Machine Learning,* **3,** 319–342.

Mingers, J. (1989b). An empirical comparison of pruning methods for decision-tree induction. *Machine Learning,* **4,** 221–243.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. In D. E. Rumelhart, J. L. McClelland and the PDP Research Group (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition, Vol. 1* (pp. 318-362). Cambridge, MA: MIT Press.