Perceptrons with Polynomial Post-Processing

Louis Sanzogni[1], **Richard F. Bonner, Ringo Chan**
School of Information Systems and Management Science,
Griffith University, Nathan 4111, Australia, and,
John A. Vaccaro, Physics Department, The Open University,
Walton Hall, Milton Keynes MK76AA, U. K.

Abstract

We introduce tensor product neural networks, composed of a layer of univariate neurons followed by a net of polynomial post-processing. We look at the general approximation problem by these networks observing in particular their relationship to the Stone-Weierstrass theorem for uniform function algebras. The implementation of the post-processing as a two-layer network with logarithmic and exponential neurons leads to potentially important 'generalised' product networks, which however require a complex approximation theory of Müntz-Szasz-Ehrenpreis type. A back-propagation algorithm for product networks is presented and used in three computational experiments. In particular, approximation by a sigmoid product network is compared to that of a single layer radial basis network, and a multiple layer sigmoid network.

Keywords *neural network approximation, polynomial post-processing, product unit*

Introduction The general idea of multiplication in neural networks seems to go back to Durbin and Rumelhart [6], and has since been successfully employed on occasion, for example, in the context of engineering implementation [3,5] or robotics [4]. Note, however, that these applications use multiplication for local *pre-processing* (the 'sigma-pi' units) in the perceptron. Presently, on the other hand, we consider multiplication in the perceptron for *post-processing*. The present note and its expanded version [7] seem to be the first to consider general approximation properties of neural networks with multiplication. In what follows, all neural networks are standard feed-forward with no a priori restriction on weights or neurons. Furthermore, networks are identified with the classes of functions they implement as the weights range over the reals, and the notions 'network' and 'function space' are used interchangeably when there is no risk of confusion.

Tensor product: polynomial post-processing Let S be a finite set of univariate neurons, each identified with its activation function, let N be a positive integer, and denote by $P(S)=P_N(S)$ the standard N-variate (infinite) perceptron on S. By definition, $P(S)$ is the linear hull of the set $S \circ Aff_N$ of affine forms on R^N composed with functions in S. It is only natural to also consider the tensor algebra $A(S)$ of $P(S)$, which is the smallest function algebra containing $S \circ Aff_N$. A network implementing $A(S)$ will be called the (*tensor*) *product network* on S. It is clear that $A(S)$ is in effect the polynomial algebra in the continuum of variables $S \circ Aff_N$, so one may think of the product network as the extension of the perceptron by *polynomial post-processing.* Recall that multiplication can be implemented in a neural network, cf [1], via the exponential isomorphism $x \rightarrow \exp(x)$ of the additive and the multiplicative structures on the real line. Algebraic monomials in n variables are then realised as 'product units', each consisting of a network of n logarithmic and one exponential neurons, the partial powers in the monomial corresponding to the weights of the intermittent network connections. Recall also, however, that weights with values other than positive integers then yield 'generalised' monomials and, if not all function in S are Strictly positive, care must be taken that the logarithm is well-defined. We then talk about generalised polynomial post-processing and generalised tensor product network.

Approximation Let X be a linear topological function space on R^N containing the set $S \circ Aff_N$. Much of the known approximation theory for the perceptron is about approximating elements of X by nested exhausting subsets P_k *of* $P(S)$, $k=1, 2, \ldots$, typically defined by vanishing weight conditions. If $P(S)$ is dense in X, one concludes the 'universal approximation property' of $P(S)$ in X: "any function in X may be approximated with any accuracy by a sufficiently large perceptron". Questions of density of linear subspaces are most classical in linear functional analysis [8], so 'universal approximation

[1] Email: cadsanzo@griffin.itc.gu.edu.au Fax: + 617 3875 7750

0-8186-7686-8/96 $05.00 © 1996 IEEE

theorems' for the perceptron are readily fabricated. The situation is analogous if X is a topological function algebra on R^N containing $S\circ Aff_N$, which is the setting for a Stone-Weierstrass type approximation theory. For example, by the classical (real) Stone-Weierstrass theorem for uniform algebras [8], given a compact K in R^N, the restriction of $A(S)$ to K is dense in the uniform algebra $C(K)$ of all continuous functions on K if and only if $S\circ Aff_N$ separates points in K. Consequently, *the tensor product network enjoys the 'universal approximation property' for continuous functions if and only if at least one of its neurons is non-constant.* We only make three observations. First, obviously, nothing is lost in the Stone-Weierstrass theorem if the weights are *a priori* restricted yielding a finite K-separating subset of $S\circ Aff_N$. This has interesting geometric interpretations. Second, results of the Bernstein-Jackson type in polynomial approximation should give bounds on the size of the approximating networks. Last, we note that the situation for the generalised tensor product networks is harder, calling for an approximation theory of the Müntz-Szasz-Ehrenpreis kind. Though complex, this case is interesting for neurocomputing by analogy with known interpretations of the trigonometric perceptron as a generalisation of the classical Fourier decomposition in time series analysis. See also the computational experiments reported in [1].

Learning Back-propagation in the product network is easy. Assume, for simplicity, that the network consists of I inputs linked to L processing units, in turn linked to O outputs. Every processing unit contains a finite number neurons, each with the activation function f, and implements their tensor product. The processing units and the 'hidden' neurons in the network are indexed by integers, say, l and j, respectively, whereby j ranges between some values $P(l)$ and $P(l+1)-1$ for the neurons in the l'th processing unit. The output of the network is

$$\overline{net}_o = \sum_{l=1}^{L} \overline{W}_{ol} \prod_{j=P(l)}^{P(l+1)-1} f\Big(\sum_{i=1}^{I} W_{ji} R_i + B_j\Big) + \overline{B}_o$$

as expressed in terms of its input vector R, with the index o ranging from one to O, and the W's and B's generically denoting the weight and shift parameters, respectively. The training set consists of pairs $[R^k, T^k]$ where R^k is the k'th input vector to the network, T^k is the corresponding target output, and k ranges from one to some K. The global error E is defined as

$$E = \frac{1}{K}\sum_{k=1}^{K}\frac{1}{O}\sum_{o=1}^{O}(\overline{net_o^k} - T_o^k)^2 .$$

The original back-propagation algorithm [6] implements the method of gradient descent: the change ΔV to the vector V of the network parameters $W, \overline{W}, B, \overline{B}$ in each iteration is proportional to the gradient of E with respect to V. In the present case, this computes as follows:

$$\Delta W_{ji} = -\varepsilon\sum_{k=1}^{K}\delta_j^k R_j^k,, \quad \Delta\overline{W}_{ol} = -\varepsilon\sum_{k=1}^{K}\overline{\delta}_o^k \prod_{j=P(l)}^{P(l+1)-1} f(net_j^k),$$

$$\Delta B_j = -\varepsilon\sum_{k=1}^{K}\delta_j^k \qquad \Delta\overline{B}_o = -\varepsilon\sum_{k=1}^{K}\overline{\delta}_o^k, \qquad \text{where}$$

$$net_{jl}^k = \sum_{i=1}^{I} W_{jli} R_i^k + B_j,$$

$$\delta_j^k = \sum_{o=1}^{O}[\overline{W}_{ol'}\,\overline{\delta}_o^k] \prod_{\substack{j'=P(l')\\ j'\neq j}}^{P(l'+1)-1} f(net_{j'}^k) f'(net_j^k),$$

$\overline{\delta}_o^k = 2(\overline{net_o^k} - T_o^k)$, and l' is such that $P(l') \le j \le P(l'+1)$. The 'step size' ε is iteratively adjusted.

Numerical examples The above was implemented in C on a SUN4/630, and tested in applications, see [7] for references. We illustrate the observed learning behaviour. For easy comparison with the radial basis network and the multilayer perceptron, target functions in the first two examples below are as in [2], where the latter two networks are compared in local learning model. All hidden neurons in the product network have the sigmoid activation function $f(x) = \dfrac{1}{1+e^{-2x}}$.

Example 1. Target function $g(x,y) = ae^{-b(x^2+y^2)}$ in the unit square in the plane, two inputs and one output, three neurons in the single product unit. The target function is sampled on a 61x61 grid. After 1000 iterations, the global error is of the order of 0.02%. *The moral*: Whatever a gaussian network can learn, a product network can learn too.

Example 2. Target function g(rcost,rsint) = sin(r)/r • sin(t-π/2), cf [2]. Trial and error shows that a good initialisation for this type of function is the alternating sum of characteristic functions of concentric rings regularly spaced around the origin. Ten such rings are used. Radial error weighting is applied to compensate for the singularity at the origin. The network trains for 10,000 epochs, but reasonable approximation is obtained already after 20 epochs. The learning rate is compared with that reported in [2] in Figure 1. The error of the product network goes down steadily beyond where the constrained 2-layer perceptron stabilises, with roughly half the error throughout learning. *The moral*: The product network can effectively learn singular functions.

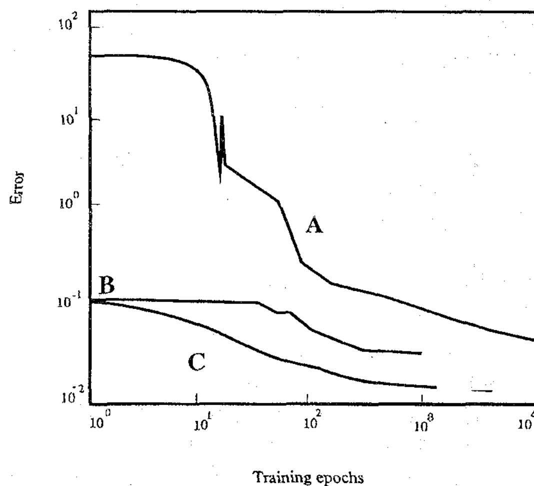

Figure 1: Comparison of networks.

Example 3. The target function is the characteristic function of a regular octahedron in the plane. A product network with thirteen 6-product units is initiated with a uniform 'honeycomb'. After a very large (!) number of iterations, the network learns the target function essentially exactly. *The moral*: The product network learns a function in its function space exactly starting from a uniform lattice-like initial state. Problems: (1) how generic is this phenomenon? (2) speed-up the convergence!

Closing comments The product network is a natural generalisation of the perceptron, and holds promise as a general purpose approximation tool. Its systematic study should preferably be conducted in a framework of some learning theory (Valiant's, for example) as it is easy to see that the VC dimension of a full product network *A(S)* is in general infinite. The generalised polynomial case seems, furthermore, to have both computational and mathematical interest. Computational experimentation is advocated, in particular in geometric applications such as image processing; observe, for example, the piece-wise linear geometry underlying the product extension of the binary sigmoid perceptron, and the potential for localised learning. Finally, we mention the possibility of parallel algorithmic implementation: the product network operates much like the single-layer perceptron, in contrast with the multi-layer perceptron where the processing is nested.

Acknowledgment The authors are grateful to an anonymous referee for pointing out the recent work [3-5] involving sigma-pi units.

REFERENCES

[1] R. Durbin and D. E. Rumelhart, *Product units with trainable exponents and multi-layer networks*, pp. 15-26 in: Fogelman Soulie, F. and J. Herault, eds., *Neurocomputing: Algorithms, Architectures and Applications*, Springer-Verlag, 1990.

[2] S. Geva and J. Sitte, *Constrained gradient descent*, pp. 76-79 in: *Proc. ACNN'94.*

[3] K.N.Gurney, *Training nets of hardware realizable sigma-pi units*, Neural Networks 5 (1992) 289-303.

[4] B. W. Mel, *MURPHY: a robot that learns by doing*, NIPS 2, American Inst. Physics, 1988.

[5] R. Neville and T.J. Stonham, *Generalization in sigma-pi networks*, Connection Science 7 (1995) 29-59.

[6] D.E.Rumelhart, J.C.McClelland, eds., *Parallel Distributed Processing,* I, MIT Press, 1986.

[7] L. Sanzogni, R.F.Bonner, R.Chan and J.A.Vaccaro, *Tensor product neural networks: perceptrons with polynomial post-processing,* Report ISMS-R12-96, Griffith University, Australia, 1996.

[8] K. Yosida, *Functional Analysis*, Springer-Verlag, 5th ed., 1978.