Global Optimization Algorithms for Training Product Unit Neural Networks

A Ismail† and AP Engelbrecht‡
†Department of Computer Science, University of Western Cape, South Africa,
aismail@uwc.ac.za
‡Department of Computer Science, University of Pretoria, South Africa,
engel@driesie.cs.up.ac.za

Abstract

Product units in the hidden layer of multilayer neural networks provide a powerful mechanism for neural networks to efficiently learn higher-order combinations of inputs. Training product unit networks using local optimization algorithms is difficult due to an increased number of local minima and increased chances of network paralysis. This paper discusses the problems with using gradient descent to train product unit neural networks, and shows that particle swarm optimization, genetic algorithms and LeapFrog are efficient alternatives to successfully train product unit neural networks.

I. Introduction

Standard multilayer neural networks (NN) use summation units in the hidden and output layers to compute the net input signal to units. For summation units the net input signal is a weighted sum of the inputs connected to that unit. Research has shown that these summation unit neural networks (SUNN) can approximate any continuous function to an arbitrary degree of accuracy, provided that the hidden layers contain a sufficient number of hidden units [3], [7]. Product units (PU) present an alternative to compute the net input signal, with the advantages of increased information capacity and the ability to form higher-order combinations of inputs. Consequently, the network architecture can be reduced, and the error in approximation decreased.

Several neural network architectures have been developed to incorporate higher-order terms. These architectures include second-order NNs [12], higher-order NNs [4], [13], sigma-pi NNs [6], functional link NNs [8] and product unit NNs [1], [10], [11]. This paper concentrates on the training of product unit neural networks (PUNN).

Gradient descent (GD) is possibly the most popular optimization algorithm to train multilayer NNs. While GD has shown to be successful in training SUNNs, GD fails to train PUNNs in general. PUs introduce more local minima, deep ravines and large valleys in the search space that trap or paralyze GD. These deficiencies of GD are discussed with reference to a specific function approximation problem. Global optimization algorithms are then investigated as an alternative approach to train PUNNs.

Section II presents the training rule for PUNNs using gradient descent. The search space of PUNNs and problems with GD are illustrated in section III. A summary of the global optimization algorithms used in this study is given in section III. Results are presented and discussed in section IV.

II. Product Unit Training Rule

Product unit neural networks were introduced by Durbin and Rumelhart [1], and further explored by Janson and Frenzel [10] and Leerink *et al* [11]. Instead of using the usual summation units where the net input signal is computed as

$$net_{y_j,p} = \sum_{i=1}^{I} z_{i,p} v_{ji} \tag{1}$$

product units are used, where

$$net_{y_j,p} = \prod_{i=1}^{I} z_{i,p}^{v_{ji}} \tag{2}$$

In equations (1) and (2) $net_{y_j,p}$ is the net input signal to unit y_j for pattern p, $z_{i,p}$ is the activation value of unit z_i for pattern p, v_{ji} is the weight between units y_j and z_i, and I is the total number of units in the previous layer (including a bias unit).

While Durbin and Rumelhart suggested two types of networks incorporating PUs [1], i.e. (1) each SU is directly connected to the input units, and also connected to a group of dedicated PUs, and (2) alternating layers of product and summation units are used, terminating the network with a summation unit, this paper assumes a three layer NN with PUs only in the hidden layer and linear activation functions in all layers. Using this architecture, and assuming that the imaginary part can be removed (see [1] for a motivation of the removal of the imaginary part),

$$y_{j,p} = net_{y_j,p} = \prod_{i=1}^{I} z_{i,p}^{v_{ji}} = e^{\rho_{j,p}} \cos(\pi\phi_{j,p}) \tag{3}$$

where

$$\rho_{j,p} = \sum_{i=1}^{I} v_{ji} \ln|z_{i,p}| \tag{4}$$

$$\phi_{j,p} = \sum_{i=1}^{I} v_{ji}\mathcal{I}_i \tag{5}$$

with $z_{i,p} \neq 0$ and

$$\mathcal{I}_i = \begin{cases} 0 & \text{if } z_{i,p} > 0 \\ 1 & \text{if } z_{i,p} < 0 \end{cases} \tag{6}$$

If the mean squared error (MSE) is used as objective function, the error in approximation is

$$E = \frac{1}{2}\frac{\sum_{p=1}^{P}\sum_{k=1}^{K}(t_{k,p} - o_{k,p})^2}{PK} \tag{7}$$

where P is the total number of training patterns, K is the number of outputs, $t_{k,p}$ is the desired output for the k-th output unit for a specific pattern p, and $o_{k,p}$ is the actual output of the NN. If GD is used, the change in hidden-to-output weights is as for standard SUs, i.e.

$$\Delta w_{kj} = -\eta\frac{\partial E}{\partial w_{kj}} \tag{8}$$

For the input-to-hidden weights, the update equations change due to the PUs, with

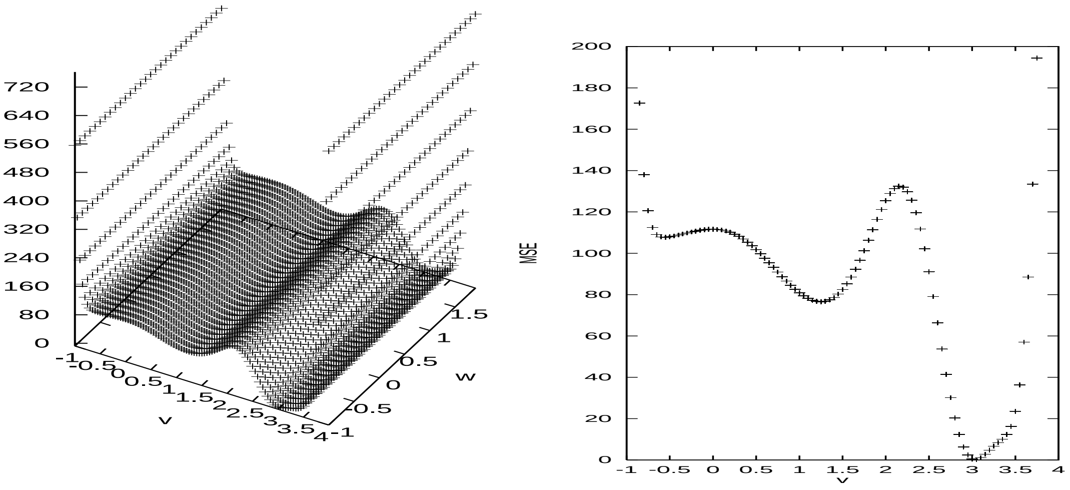
$$\Delta v_{ji} = -\eta\frac{\partial E}{\partial v_{ji}} = \eta\delta_{y_{j,p}} D_{ji,p} \tag{9}$$

where $\delta_{y_{j,p}}$ is the usual back-propagated error signal. In equation (9),

$$D_{ji,p} = \frac{\partial net_{y_{j,p}}}{\partial v_{ji}} = e^{\rho_{j,p}}[\ln|z_{i,p}|\cos(\pi\phi_{j,p}) - \mathcal{I}_i\pi\sin(\pi\phi_{j,p})] \tag{10}$$

III. Search Space for Product Unit Neural Networks

A major advantage of product units is an increased information capacity compared to summation units [1], [11]. Durbin and Rumelhart showed that the information capacity of a single PU (as measured by its capacity for learning random boolean patterns) is approximately $3N$, compared to $2N$ for a single SU (N is the number of inputs to the unit) [1]. The larger capacity means that functions approximated using PUs will require less processing elements than required if SUs were used.



(a) (b)

Fig. 1. Illustration of PU search space for $f(z) = z^3$

While PUNNs provide an advantage in having smaller network architectures, a major drawback of PUs is an increased number of local minima, deep ravines and valleys. The search space for PUs is usually extremely convoluted. Gradient descent, which works best when the search space is relatively smooth, therefore frequently gets trapped in local minima or becomes paralyzed (which occurs when the gradient of the error with respect to the current weight is close to zero). Leerink *et al* illustrated that the 6-bit parity problem could not be trained using GD and PUs [11]. Two reasons were identified to explain why GD failed: (1) weight initialization and (2) the presence of local minima. The initial weights of a network is usually computed as small random numbers. Leerink *et al* argued that this is the worst possible choice of initial weights, and suggested that larger initial weights be used instead. In our experience, GD only manages to train PUNNs when the weights are initialized in close proximity of the optimal weight values [9] - the optimal weight values are, however, usually not available.

As an example to illustrate the complexity of the search space for PUs, consider the approximation of the function $f(z) = z^3$, with $z \in [-1, 1]$. Only one PU is needed, resulting in a 1-1-1 NN architecture (that is, one input, one hidden and one output unit). In this case the optimal weight values are $v = 3$ (the input-to-hidden weight) and $w = 1$ (the hidden-to-output weight). Figure 1 visualizes the search space for $v \in [-1, 4]$ and $w \in [-1, 1.5]$. The error is computed as the mean squared error over 500 randomly generated patterns. Figure 1 clearly illustrates 3 minima, with the global minimum at $v = 3, w = 1$. Initial small random weights will cause the network to be trapped in one of the local minima (having very large MSE). Large initial weights may also be a bad choice. Assume an initial weight $v \geq 4$. The derivative of the error with respect to v is extremely large due to the steep gradient of the error surface. Consequently, a large weight update will be made which may cause jumping over the global minimum. The neural network either becomes trapped in a local minimum, or oscillates between the extreme points of the error surface.

A global optimization algorithm is rather needed to allow searching of larger parts of the search space. Simulated annealing [11], random search [11] and genetic algorithms have already been used successfully to train PUNNs [10]. In this paper we show that PSO and LeapFrog are other efficient candidates to train PUNNs. The paper also shows that random search is an inefficient approach to train PUNNs.

A short summary of the global optimization algorithms used for this study is given below:

- **Random Search:** At each epoch randomly perturb a weight vector with uniform noise until the specified number of epochs or the MSE threshold has been reached. The solution is the weight vector that corresponds to the lowest MSE on the test set.

- **Particle Swarm Optimization (PSO):** PSO is a population based search procedure where the individuals, referred to as particles, are grouped into a swarm [2]. Each particle in the swarm represents a candidate solution to the optimization problem. Each particle is "flown" through the multidimensional search space, adjusting its position in search space according to own experience and that of neighboring particles. A particle therefore makes use of the best position encountered by itself and the best position of neighbors to position itself towards the global minimum. The performance of each particle is measured as the MSE on the training set. For the purposes of this study, a particle represents the weight vector of a NN, including biases.
- **Genetic Algorithms (GA):** GAs are based on the principle of natural evolution where principles such as survival of the fittest, natural selection, reproduction and mutation are used to produce a "best" individual [5]. In a GA, a population of individuals compete to survive. Each individual represents one candidate solution, which is, in our case, a weight vector (including biases) of a NN. The survival strength, or fitness, of an individual is measured as a function of the MSE on the training set, i.e. $f(\vec{w}) = 1/(1 + MSE(\vec{w}))$. Each weight is mapped into a 30 bit binary number, $(2^{30} - 1)\frac{w - w_{min}}{w_{max} - w_{min}}$. The top 20% of each population is culled, and two-point crossover is used with random selection. However, 40% of the new generation is created through crossover where at least one of the parents is in the top 20% of the current population. Random mutation is used.
- **LeapFrog Optimization (LFOP):** LeapFrog is an optimization approach based on the physical problem of the motion of a particle of unit mass in an n-dimensional conservative force field [14]. The potential energy of the particle in the force field is represented by the function to be minimized - in the case of NNs, the potential energy is the MSE. The objective is to conserve the total energy of the particle within the force field, where the total energy consists of the particle's potential and kinetic energy. The optimization method simulates the motion of the particle, and by monitoring the kinetic energy, an interfering strategy is adapted to appropriately reduce the potential energy.

IV. Experimental Results

This section applies the global optimization algorithms listed above to the following functions:

- The quadratic function $f(z) = z^2$, with $z \sim U(-1, 1)$. The training and test sets consisted of 50 distinct randomly selected patterns. A 1-1-1 PUNN was used. The following optimal parameter values were used for PSO: an acceleration constant of 1.0, maximum velocity of 1.5, initial weights in the range $[-0.925, 0.925]$, and 50 particles. The optimal parameter values used for the GA were a 0.01 probability of mutation, a 0.7 probability of crossover, an a population of 50 individuals.
- The cubic function $f(z) = z^3 - 0.04z$, with $z \sim U(-1, 1)$. The training and test sets consisted of 50 distinct randomly selected patterns. A 1-2-1 PUNN was used. The optimal parameter values used for PSO were an acceleration of 1.0, maximum velocity of 1.0, initial weights in the range $[-0.95, 0.95]$ and 50 particles. For the GA, the optimal parameter values were a 0.01 probability of mutation, a 0.7 probability of crossover, and a population of 50 individuals.
- The henon-map $z_t = 1 + 0.3z_{t-2} - 1.4z_{t-1}^2$, with $z_1, z_2 \sim U(-1, 1)$. The training and test sets consisted of 200 distinct randomly selected patterns. A 1-5-1 PUNN architecture was used. PSO used the following optimal parameter values: an acceleration of 0.1, maximum velocity of 0.1, initial weights in the range $[-0.875, 0.875]$ and 50 particles. For the GA a 0.005 probability of mutation, a 0.7 probability of crossover and a population of 100 individuals.

Thirty simulations were performed for each of the global optimization algorithms using PUs. Different training sets and initial weights were used for each of the simulations. Each simulation was executed for 500 epochs, where an epoch is one training pass through the training set. Results reported are averages over the 30 simulations and 95% confidence intervals as obtained from the t-distribution.

Table I summarizes the average mean squared error for the training and test sets after 500 epochs for each problem and optimization algorithm. In addition, figure 2 illustrates the learning profiles for each optimization method for the test set (as measure of generalization performance). Table II lists the average number of epochs to reach specified generalization levels.

Random selection with PUs showed to be inefficient over the 500 epochs. PSO and GA gave substantially better training errors and generalization for the quadratic and cubic functions. For the henon-map PSO and GA achieved very good results. However, LeapFrog perform the best. Figure 2 illustrates that

		Average Mean Squared Error	
		Training set	**Test set**
PSO	quadratic	0.000117 ± 0.000088	0.000116 ± 0.000079
	cubic	0.000075 ± 0.000056	0.000270 ± 0.000417
	henon-map	0.005738 ± 0.004482	0.007176 ± 0.005345
GA	quadratic	0.000272 ± 0.000286	0.000407 ± 0.000437
	cubic	0.000214 ± 0.000169	0.000487 ± 0.000637
	henon-map	0.007233 ± 0.005299	0.010576 ± 0.009243
LFOP	quadratic	0.001583 ± 0.000725	0.001930 ± 0.000899
	cubic	0.001554 ± 0.001135	0.002203 ± 0.001737
	henon-map	0.000947 ± 0.001284	0.000901 ± 0.001223
RS	quadratic	0.035612 ± 0.015555	0.038345 ± 0.014542
	cubic	0.100760 ± 0.022660	0.096286 ± 0.019692
	henon-map	2.932112 ± 0.081377	3.129933 ± 0.074526

TABLE I

Mean squared error results

		Generalization Levels			
		0.1	**0.01**	**0.001**	**0.0001**
PSO	quadratic	2.47 ± 0.70	31.37 ± 9.38	113.27 ± 29.37	326.07 ± 88.10
	cubic	2.13 ± 0.73	97.03 ± 30.87	315.30 ± 109.38	1207.97 ± 340.77
	henon-map	69.33 ± 15.40	284.30 ± 97.72	485.53 ± 105.63	906.60 ± 74.32
GA	quadratic	3.50 ± 0.94	19.00 ± 3.77	87.60 ± 49.10	172.30 ± 64.86
	cubic	2.00 ± 0.62	73.70 ± 16.59	392.50 ± 190.24	1193.50 ± 447.69
	henon-map	36.30 ± 5.41	149.90 ± 45.79	459.80 ± 37.49	489.90 ± 19.80
LFOP	quadratic	63.10 ± 12.47	69.50 ± 9.55	149.90 ± 82.34	165.90 ± 70.76
	cubic	49.80 ± 7.47	72.50 ± 8.20	145.40 ± 90.75	237.50 ± 91.97
	henon-map	53.20 ± 10.46	85.0 ± 12.56	123.20 ± 58.00	152.50 ± 68.35
RS	quadratic	356.21 ± 3.54	500.00 ± 22.36	-	-
	cubic	477.92 ± 6.74	500.00 ± 22.36	-	-
	henon-map	-	-	-	-

TABLE II

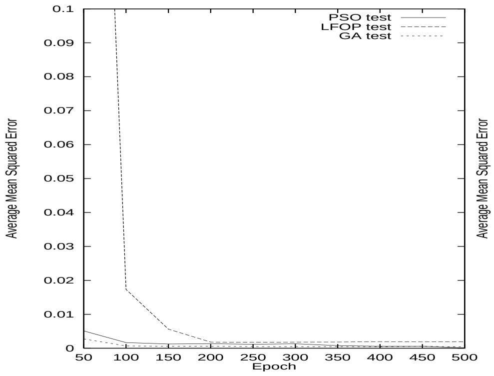
Convergence to generalization levels (considering only those simulations that did converge)

PSO and GA have larger reductions in error early in training, reaching low errors using substantially less training epochs (also refer to table II). To reach MSE levels less than 0.001, LFOP showed to use much less epochs than do PSO and GA. These results suggest that evolutionary approaches such as PSO and GAs be used for an initial training step, and that LFOP be used to refine the weights to lower errors.

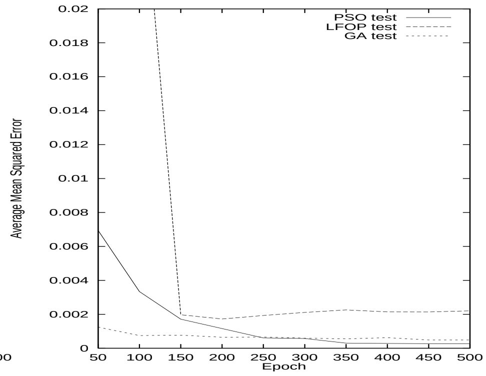
V. Conclusions

This paper discussed the deficiencies of local optimization algorithms, such as gradient descent, to train product unit neural networks. The problems associated with the search space of PUNNs were discussed. Experimental results have shown that particle swarm optimization, genetic algorithms and LeapFrog optimization are efficient in training PUNNs. Excellent approximation accuracies were obtained within the 500 epoch limit. PSO and GAs have shown to reach low errors very early in training.

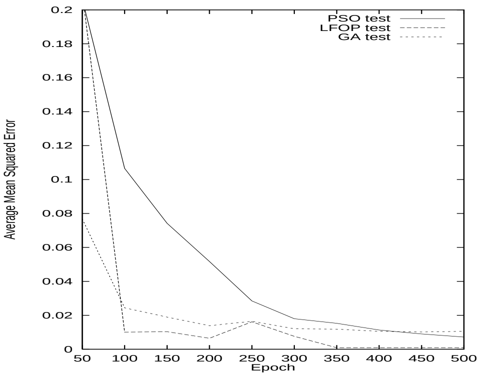
Future work on PUs will include a comparison of PUNNs with SUNNs to determine what is gained by using PUs. This investigation will test the hypotheses that PUNNs are more accurate, faster and use smaller architectures.



(a) $f(z) = z^2$



(b) $f(z) = z^3 - 0.04z$



(c) Henon Map

Fig. 2. PSO, LFOP and GA generalization

References

[1] R Durbin and D Rumelhart, *Product Units: A Computationally Powerful and Biologically Plausible Extension to Backpropagation Networks*, Neural Computation, Vol 1, pp 133-142, 1989.
[2] RC Eberhart, RW Dobbins and P Simpson, *Computational Intelligence PC Tools*, Academic Press, 1996.
[3] K-I Funahashi, *On the Approximate Realization of Continuous Mappings by Neural Networks*, Neural Networks, Vol 2, pp 183-192, 1989.
[4] C Lee Giles, *Learning, Invariance, and Generalization in Higher-Order Neural Networks*, Applied Optics, 26(23), pp 4972-4978, 1987.
[5] DE Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.
[6] KN Gurney, *Training Nets of Hardware Realizable Sigma-Pi Units*, Neural Networks, Vol 5, pp 289-303, 1992.
[7] K Hornik, M Stinchcombe, H White, *Universal Approximation of an Unknown Mapping and Its Derivatives using Multilayer Feedforward Networks*, Neural Networks, Vol 3, pp 551-560, 1990.
[8] A Hussain, JJ Soraghan, TS Durbani, *A New Neural Network for Nonlinear Time-Series Modelling*, NeuroVest Journal, pp 16-26, Jan 1997.
[9] A Ismail and AP Engelbrecht, *Training Product Units in Feedforward Neural Networks using Particle Swarm Optimization*, In: Development and Practice of Artificial Intelligence Techniques, VB Bajić, D Sha (eds), Proceedings of the International Conference on Artificial Intelligence, Durban, South Africa, pp 36-40, 1999.
[10] DJ Janson and JF Frenzel, *Training Product Unit Neural Networks with Genetic Algorithms*, IEEE Expert Magazine, pp 26-33, October 1993.
[11] LR Leerink, C Lee Giles, BG Horne and MA Jabri, *Learning with Product Units*, Advances in Neural Information Processing Systems, Vol 7, pp 537, 1995.
[12] S Milenković, Z Obradović and V Litovski, *Annealing Based Dynamic Learning in Second-Order Neural Networks*, Technical Report, Department of Electronic Engineering, University of Nis, Yugoslavia, 1996.
[13] NJ Redding, A Kowalczyk and T Downs, *Constructive Higher-Order Network Algorithm that is Polynomial in Time*, Neural Networks, Vol 6, pp 997-1010, 1993.
[14] JA Snyman, *An Improved Version of the Original LeapFrog Dynamic Method for Unconstrained Minimization: LFOP1(b)*, Applied Mathematical Modelling, Vol 7, pp 216-218, June 1983.