TRAINING PRODUCT UNITS IN FEEDFORWARD NEURAL NETWORKS USING PARTICLE SWARM OPTIMIZATION

A Ismail[a] and AP Engelbrecht[b]
[a]Department of Computer Science,
University of Western Cape
aismail@uwc.ac.za
[b] Department of Computer Science,
University of Pretoria
engel@driesie.cs.up.ac.za

ABSTRACT

Product unit (PU) neural networks are powerful because of their ability to handle higher order combinations of inputs. Training of PUs by backpropagation is however difficult, because of the introduction of more local minima. This paper compares training of a product unit neural network using particle swarm optimization with training of a PU using gradient descent.

INTRODUCTION

Traditional neural networks are constructed using multiple layers of summation units units. In these networks each input is multiplied by a weight and then summed. A nonlinear activation function, such as the logistic function, is often used to squash the sum. Research has shown that these networks can approximate any function to any arbitrary degree of accuracy [1, 5]. However, these networks require a large number of summation units when approximating complex functions that involve higher order combinations of its inputs. When approximating polynomials, higher order combinations of its inputs, such as x^3y^7, are often required. Networks that utilize higher combinations of its inputs will greatly reduce the number of processing units required to represent these complex functions.

Several neural network (NN) models have been developed to gain an advantage in using higher-order terms. These techniques include second-order NNs [10], higher-order NNs [6, 11], sigma-pi NNs [4] and functional link NNs [7]. This paper concentrates on another alternative, referred to as product unit neural networks (PUNN).

Description of a Typical Product Unit Neural Network

PUNNs were introduced by Durbin and Rumelhart [2] and further explored by Jansen and Frenzel [8] and Leerink *et al* [9]. In PUNNs the hidden layer summation units are replaced by product units (PU) to compute the weighted product of inputs:

$$net_{y_j} = \prod_{i=1}^{I} z_i^{v_{ji}} \tag{1}$$

instead of

$$net_{y_j} = \sum_{i=1}^{I} z_i v_{ji} \tag{2}$$

where net_{y_j} is the net input to hidden unit y_j, z_i is an input unit, v_{ji} is the weight between hidden unit y_j and input z_i, and I is the total number of input units (including a bias unit to the hidden layer).

Durbin and Rumelhart suggested two types of networks incorporating product units[2]. In the one network type each summation unit is directly connected to the input units, and also connected to a group of dedicated product units. In the other, the network consists of alternating layers of product and summation units, terminating the network with a summation unit. Durbin and Rumelhart tested the alternating summation-product unit network on several problems where the output of all the summation units was squashed using the standard logistic function and a linear function was applied to the output from the product units. The same approach was adopted in the following tests in training a PU neural network, except that linear functions were applied to the output of both summation and product units.

Training Rule for Product Units

Equation (1) can be rewritten in terms of logarithms and exponentials as follows,

$$\begin{aligned} y_j &= \prod_{i=1}^{I} z_i^{v_{ji}} \\ &= e^{\sum_{i=1}^{I} v_{ji} \ln |z_i|} \times \\ &\quad (\cos \pi \sum_{i=1}^{I} v_{ji}\mathcal{I}_i + i \sin \pi \sum_{i=1}^{I} v_{ji}\mathcal{I}_\rangle) \end{aligned} \tag{3}$$

where

$$\mathcal{I}_i = \begin{cases} 0 & \text{if } z_i > 0 \\ 1 & \text{if } z_i < 0 \end{cases} \tag{4}$$

and $z_i \neq 0$. From equation (3), the change in input-to-hidden weights are

$$\Delta v_{ji} = -\eta \delta_{y_j} D_{ji} \tag{5}$$

where, assuming gradient descent (GD) optimization, η is the learning rate, δ_{y_j} is the back-propagated error for hidden unit y_j, and

$$\begin{aligned} D_{ji} &= \ln|z_i| e^{\sum_{i=1}^{I} v_{ji} \ln|z_i|} \cos(\pi \sum_{i=1}^{I} v_{ji} \mathcal{I}_i) \\ &\quad - \mathcal{I}_i e^{\sum_{i=1}^{I} v_{ji}} \pi \sin(\pi \sum_{i=1}^{I} v_{ji} \mathcal{I}_i) \qquad (6) \end{aligned}$$

A severe drawback of PUs is that they introduce more local minima, and they induce abrupt, large changes in weights due to the exponential term in the input-to-hidden weight updates Δv_{ji}.

The complex part of the equation has been omitted in training the PUNN, since Durbin and Rumelhart have discovered in their experiments that apart for the added complexity of working in the complex domain, i.e. the doubling of equations and weight variables, no substantial improvements in the results were gained. PUNNs provide the advantage of increased information capacity compared to summation units [2, 9]. Durbin and Rumelhart showed empirically that the information capacity of these units (as measured by their capacity for learning random boolean patterns) is approximately $3N$, for a single product unit, compared to $2N$ for a single summation unit, where N is the number of inputs to the unit [2]. The larger capacity means that functions implemented by summation units, if implemented using product units will contain fewer processing units. Another advantage of PUs is their ability to use fractional and negative powers; they can also form simple product expressions by constraining weights to integer values. The overhead required to raise an arbitrary base to an arbitrary power in product units, makes it more likely that they will supplement rather than replace summation units [2].

In this paper two methods for training PUNNs are compared, i.e using gradient descent (GD) and particle swarm optimization (PSO). The following section briefly discusses the problems associated with the training of PUNNs. This will be followed by a brief overview of particle swarm optimization. Optimal parameter settings are determined for GD and PSO for each experiment. Performance comparisons of these two training methods are based on the optimal parameter settings. PSO is compared to GD in three case studies involving function approximation. This paper reports results to illustrate the efficiency of the PSO in training PUNNs. Results are reported to illustrate the training accuracy and convergence characteristics in comparison with the gradient descent approach.

Training problems with Product Units

While product units (PUs) increase a neural network's capability, they also add complications. Traditional networks are usually trained using GD. GD optimization works best when the solution space is relatively smooth, with a few local minima or plateaus. Unfortunately, the solution space for product networks can be extremely convoluted, with numerous local minima that trap GD [2, 8, 9]. Leerink *et al* have concluded that the parity-6 problem could not be trained by using product units and standard backpropagation [9]. They have found two main reasons for this: (a) weight initialization and (b) the presence of local minima. In the backpropagation procedure the first step is to initialize all weights to small random values. Leerink *et al* pointed out that small initial random weights is the worst possible choice for PU's. They suggested that larger initial weights be used instead. In our experience, GD only managed to train PUNNs when the weights are initialized in close range of the optimal weight values - the optimal weight values are, however, usually not available.

A global optimization method is needed instead of GD, which is a local optimizer, to allow searching of larger parts of the search space. Simulated annealing [9], random search [9] and genetic algorithms [8] have been applied successfully to train PUNNs. This paper applies another optimization approach, particle swarm optimization (PSO) [3], to the training of PUNNs.

Overview of Particle Swarm Optimization

PSO as an optimization tool provides a population-based search procedure, modelled after the social behaviour of flocks of birds and schools of fish. A swarm consists of individuals, called particles, which change their position with time. Each particle represents a potential solution to the problem. In a PSO system, particles fly around in a multidimensional search space. During flight each particle adjusts its position according to its own experience, and according to the experience of neighbouring particles, making use of the best position encountered by itself and its neighbours. The effect is that particles move towards the global best solution, while still searching a wide area around the best solution. The performance of each particle is measured according to a pre-defined fitness function, which is related to the problem being solved. For the purposes of this study, the fitness function is the sum squared error (SSE). The interested reader is also referred to [12] for more information on using PSO to train NNs with summation units.

Criteria for optimization

Three performance criteria for determining an optimal GD and PSO are considered:

(a) the average of the sum of squared errors (SSE) on the training and test sets after 4000 training epochs, over a number of simulations.

(b) the number of epochs required to reach a generalization level of $SSE < 0.001$.

(c) the number of simulations that converged to an $SSE < 0.001$ within the 4000 epochs.

Parameters that influence the performance of PSO are optimized. These parameters include (1) inertia weight, (2) maximum velocity, (3) acceleration constant and (4) the size of the the population (number of particles of the swarm). A large inertia weight facilitates global exploration, while a small weight facilitates local exploration. For large weights the PSO always tries to explore new areas of the search space. The function of the inertia weight is to influence the trade-off between global and local exploration of the search space. A good value for the weight is thus critical to the performance of PSO. Maximum velocity determines the fineness with which a region between the current position and the target position will be searched [3]. A maximum velocity that is too high might cause the particles to fly past good solutions. A large maximum value will result in a greater area of the search space being explored. A small maximum velocity limits the maximum exploration ability, causing PSO to favour a local search.

A number of factors influence the performance of backpropagation by gradient descent. These include (1) the interval for initial weight selection (2) the learning rate and (3) momentum.

Determining Optimal Parameter Values

The aim of this paper is to compare the performance of an optimal PSO with an optimal GD in training a product unit neural network to approximate functions. Three PUNNs were used to approximate the functions $f(z) = z^2$, $f(z) = z^3 - 0.04z^2$ for the interval $-1 < z < 1$ and the Henon curve, a time series generated by the formula $z_t = 1 + 0.3z_{t-2} - 1.4z_{t-1}^2$. Two configurations, PSO and GD, were used for training each function.

Optimal PSO

A range of values for the inertia weight, maximum velocity, acceleration constant and number of particles have been tested to find the best combination of parameter values for each experiment.

The PUNN for the quadratic and cubic functions consisted of a single input unit, two hidden product units and a single output unit. The PUNNs were trained for a fixed number of epochs (4000). Each training and test set consisted of 50 randomly generated patterns. To determine the optimal value for each parameter for PSO, eight training sessions, each consisting of 30 simulations, were conducted. Training started with a swarm of 50 particles and parameters *maximum velocity* and *acceleration constant*, both initialized to 1.0. The best value for a parameter was determined by fixing the remaining parameters during a series of training sessions, while the parameter under consideration was gradually adjusted between training sessions. After each series of training sessions the parameter value with the smallest average SSE over the 30 simulations and the highest number of simulations where SSE < 0.001, was selected as the best value. This parameter was then fixed at its best value, while training continued in search of the best values for the other parameters. For the Henon curve the PUNN consisted of 2 input units, 5 hidden product units and 1 output unit, i.e. a 2:5:1 configuration. The optimal values for each of the parameters for PSO appear in the table 1. The size of the population was chosen as 50, since the marginal improvement gained in performance for swarms of sizes 70, 100, 150 and 200 did not justify the corresponding increase in processing time.

Function	Weight	Max velocity	Accel const	Particles
quadratic	0.925	1.5	1.0	50
cubic	0.95	1.0	1.0	50
Henon	0.875	1.0	1.0	50

Table 1: Best parameter values for PSO

Optimal GD

The same procedure for determining the optimal parameters in PSO, was applied to GD, using various initial weight initializations, learning rate and momentum values. However, for GD, the number of simulations were increased to 70 to compensate for the high number of simulations that resulted in overflow errors. Almost two thirds of the training sessions resulted in overflow errors. Training started with the momentum initialized to 0.9 and the learning rate to 0.05. During weight selection it was ensured that approximately 50% of the weights were negative.

The best values for each of the parameters are listed in table 2.

Parameter values for the Henon curve could not be

Function	Weight interval	Momentum	Learning rate
quadratic	[1, 2]	0.5	0.0275
cubic	[1,3]	0.5	0.0275

Table 2: Best parameter values for GD

	GD	PSO
No of simulations	70	30
no of overflows	40	0
no of simulations with generalization less than 0.001	12	30
Ave SSE	0.0161 ± 0.02222	0.000086 ± 0.000074
epochs to reach generalization less than 0.001	263.4 ± 135.4	338.3 ± 73.4

Table 3: Results for $f(z) = z^2$

established, since none of the simulations returned a result, other than overflows. A possible explanation for this behaviour is that the weights selected from the initial interval is too far from the optimal weights, causing too large jumps in weight space.

Comparing the two optimal architectures

The optimal PSO and GD architectures were trained on the same sets for 4000 epochs. Results of these experiments are reported in the tables below. The error results reported are for the average SSE over all converged simulations, obtained by summing the SSE for the training and test sets. The tables include results for 95% confidence intervals obtained from the t-distribution.

Table 3 summarizes the results for the function $f(z) = x^2$. This table shows that PSO is by far superior to GD in terms of accuracy and convergence. PSO achieved a much lower error, with small variance, than GD. Also, PSO has a 100% convergence to a generalization of 0.001 after 4000 epochs, compared to GD's 17%. The average number of epochs needed to converge to a generalization SSE of 0.001 were calculated using only those simulations that did converge. While GD did require less epochs to converge, there is a very large variance compared to PSO. Observe that the average for GD includes 12 out of 70 simulations. The exceptionally better error results obtained by PSO clearly outway the increased training time.

The results for $f(z) = z^3 - 0.04z^2$ are summarized in table 4. Again, PSO performed exceptionally well, having a very small error and variance in error. Convergence results of PSO were better than GD, having 93% converged simulations, compared to GD's 10%. For this experiment, PSO also required more training epochs to reach a 0.001 generalization, but keep in mind that the average for GD is only over 7 simulations.

	GD	PSO
No of simulations	70	30
no of overflows	32	0
no of simulations with generalization less than 0.001	7	28
Ave SSE	0.0597 ± 0.021	0.00062 ± 0.000134
epochs to reach generalization less than 0.001	987.3 ± 629.2	1545.9 ± 408.1

Table 4: Results for $f(z) = z^3 - 0.04z^2$

For the henon map, the training and test sets consisted of 200 patterns each. Training continued for 1000 epochs only. None of the 70 GD simulations produced results, since they all ended in overflow. The PSO achieved an average SSE over 30 simulations of 0.077 ± 0.064, with a corresponding mean squared error (MSE) of 0.0004. This compares well with a MSE of 0.0848 using GD on summation units.

CONCLUSION AND FUTURE WORK

This paper presented PSO as an alternative global optimization algorithm to train PUNNs. The experiments and results presented showed PSO to be extremely successful in training PUNNs. Future work will include testing on more functions, and comparison between PSO for PUs and PSO for summation units to establish whether anything is gained in using PUs instead of summation units. PSO will also be compared to simulated annealing, genetic algorithms and LeapFrog optimization.

References

[1] G Cybenko, *Continuous-Valued Neural Networks with Two Hidden Layers Are Sufficient*, Technical Report, Department of Computer Science, Tufts University, Medford, Massachusetts, 1989.

[2] 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.

[3] RC Eberhart, RW Dobbins and P Simpson, *Computational Intelligence PC Tools*, Academic Press, 1996.

[4] KN Gurney, *Training Nets of Hardware Realizable Sigma-Pi Units*, Neural Networks, Vol. 5, pp 289-303, 1992.

[5] K Hornik, *Multilayer Feedforward Networks are Universal Approximators*, Neural Networks, Vol 2, 1989, pp 359-366.

[6] C Lee Giles, *Learning, invariance, and generalization in higher-order neural networks*, Applied Optics, 26(23), pp 4972-4978, 1987.

[7] A Hussain, JJ Soraghan, TS Durbani, *A New Neural Network for Nonlinear Time-Series Modelling*, NeuroVest Journal, Jan 1997, pp 16-26.

[8] DJ Janson and JF Frenzel, *Training Product Unit neural networks with Genetic Algorithms*, IEEE Expert Magazine, pp.26-33, October 1993.

[9] LR Leerink, C Lee Giles, BG Horne and MA Jabri, *Learning with Product Units*, Advances in Neural Information Processing Systems, Vol 7, p. 537, M.I.T Press, 1995.

[10] 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.

[11] 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.

[12] F van den Bergh, *Particle Swarm Weight Initialization in Multi-layer Perceptron Artificial Neural Networks*, accepted for ICAI, Durban, South Africa, 1999.