Training Product Unit Networks using Cooperative Particle Swarm Optimisers

F. van den Bergh, A.P. Engelbrecht
Department of Computer Science
University of Pretoria
fvdbergh@cs.up.ac.za, engel@driesie.cs.up.ac.za

Abstract

The Cooperative Particle Swarm Optimiser (CPSO) is a variant of the Particle Swarm Optimiser (PSO) that splits the problem vector, for example a neural network weight vector, across several swarms. This paper investigates the influence that the number of swarms used (also called the split factor) has on the training performance of a Product Unit Neural Network. Results are presented, comparing the training performance of the two algorithms, PSO and CPSO, as applied to the task of training the weight vector of a Product Unit Neural Network.

1 Introduction

The particle swarm optimiser is a semi-global optimisation algorithm, first introduced by Eberhart and Kennedy [1]. It has been applied successfully to applications involving neural network training [2, 3, 4] and function minimisation [5, 6].

The Cooperative Particle Swarm Optimiser (CPSO, or split swarm) is a recent modification to the original PSO algorithm leading to a significant reduction in training time [3, 7]. Each vector to be optimised by the CPSO is split across multiple swarms, with each swarm optimising a disjoint part of the vector with the help of the other swarms. The cooperative approach increased the number of adjustable parameters in the PSO algorithm significantly, one parameter being the number of swarms used, which can also be interpreted as the number of parts that each particle is split into, henceforth called the *split factor*. The effect that the split factor has on the CPSO algorithm is studied here.

This paper applies the CPSO, as well as the original PSO, to the problem of finding the optimal weights of a Product Unit Neural Network (PUNN). Particle swarms have been used to train PUNNs with promising results [2].

Section 2 briefly describes the CPSO algorithm, followed by a brief review of the Product Unit Neural Network in Section 3. A description of the experimental set-up is described in Section 4, followed by some results in Section 5. A summary of the findings of this paper is presented in Section 6.

2 Cooperative Particle Swarms

The PSO, like a Genetic Algorithm, is a population based optimisation technique, but the population is now called a *swarm*.

Each particle j has the following attributes: A current position in search space, $\mathbf{x}_j$, a current velocity, $\mathbf{v}_j$, and a personal best position in search space, $\mathbf{y}_j$. During each iteration each particle in the swarm is updated using (1) and (2). Assuming that the function f is to be minimised, that the swarm consists of n particles, and $r_1 \sim U(0,1)$, $r_2 \sim U(0,1)$ are elements from two uniform random sequences in the range $(0,1)$, then:

$$\mathbf{v}_{j,i} := w\mathbf{v}_{j,i} + c_1 r_1(\mathbf{y}_{j,i} - \mathbf{x}_{j,i}) + c_2 r_2(\hat{\mathbf{y}}_i - \mathbf{x}_{j,i}) \quad (1)$$

for all $i \in 1..W$, where W is the dimension of the function being optimised.

$$\mathbf{x}_j := \mathbf{x}_j + \mathbf{v}_j \quad (2)$$

$$\mathbf{y}_j := \begin{cases} \mathbf{y}_j & \text{if } f(\mathbf{x}_j) \geq f(\mathbf{y}_j) \\ \mathbf{x}_j & \text{if } f(\mathbf{x}_j) < f(\mathbf{y}_j) \end{cases} \quad (3)$$

$$\hat{\mathbf{y}} \in \{\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_n\} \mid f(\hat{\mathbf{y}}) = \min(f(\mathbf{y}_0), f(\mathbf{y}_1), \ldots, f(\mathbf{y}_n)) \quad (4)$$

Note that $\hat{\mathbf{y}}$ is therefore the global best position amongst all the particles. The value of each of the $\mathbf{v}_{j,i}$ are clamped to the range $[-v_{max}, v_{max}]$ to prevent the PSO from 'exploding' — thus leaving the search space. The value of v_{max} is usually

chosen to be $k \times x_{max}$, with $0.1 \leq k \leq 1.0$ [8]. Note that this does not restrict the values of $\mathbf{x}_j$ to the range $[-v_{max}, v_{max}]$; it only limits the maximum distance that a particle will move during one iteration.

The variable w in (1) is called the *inertia weight*; this value is typically set up to vary linearly from 1 to near zero during the course of a training run. Larger values for w result in smoother, more gradual changes in direction through search space.

Acceleration coefficients c_1 and c_2 also control how far a particle will move in a single iteration. Typically these are both set to a value of 2 [8], although assigning different values to c_1 and c_2 sometimes leads to improved performance [9].

2.1 Original PSO

The pseudo code for the original PSO algorithm, with M particles, is listed in Figure 1. Note that each W-dimensional vector is a complete potential solution vector.

```
Create and initialise an n-dimensional PSO : S
repeat:
   for each particle j ∈ [1..M] :
      if f(S.x_j) < f(S.y_j)
         then S.y_j = S.x_j
      if f(S.y_j) < f(S.ŷ)
         then S.ŷ = S.y_j
   endfor
   Perform updates on S using eqns. (1–2)
until stopping criterion is met
```

Figure 1: Pseudo Code for the Plain Swarm Algorithm

2.2 Cooperative PSO

Figure 2 lists the pseudo code for the Cooperative PSO with a variable split factor K, and M particles per swarm. This means that an input vector with W dimensions will be split across K swarms, where $W \bmod K$ swarms have $\lceil W/K \rceil$-dimensional vectors, and $K - (W \bmod K)$ swarms have $\lfloor W/K \rfloor$-dimensional vectors.

Note that the neural network requires a W-dimensional weight vector to perform a forward propagation. This means that each swarm in the cooperative algorithm must use a vector from each of the other swarms to build a full W-dimensional vector. The function **b** defined in Figure 2 does exactly this: it takes the best vector (particle) from each of the other swarms, concatenates them, splicing in the current vector (particle) from the current swarm j in the appropriate position. This vector is then used to determine the Mean Sum Squared Error (MSSE) of the network by performing a

```
define
   b(j, z) ≡ (S_1.ŷ, ..., S_{j-1}.ŷ, z, S_{j+1}.ŷ, ..., S_K.ŷ)
K_1 = W mod K
K_2 = K − (W mod K)
Initialise K_1 ⌈W/K⌉-dim. PSOs : S_i, i ∈ [1..K_1]
Initialise K_2 ⌊W/K⌋-dim. PSOs : S_i, i ∈ [(K_1 + 1)..K]
repeat:
   for each swarm i ∈ [1..K] :
      for each particle j ∈ [1..M] :
         if f(b(i, S_i.x_j)) < f(b(i, S_i.y_j))
            then S_i.y_j = S_i.x_j
         if f(b(i, S_i.y_j)) < f(b(i, S_i.ŷ))
            then S_i.ŷ = S_i.y_j
      endfor
      Perform updates on S_i using eqns. (1–2)
   endfor
until stopping criterion is met
```

Figure 2: Pseudo Code for the Split Swarm Algorithm

forward propagation through the network using **b** as weight vector.

It is important to realise that increasing the number of swarms used involves a trade-off with the number of iterations that the algorithm can execute before its allocated number of function evaluations have been used. For example, a plain PSO with 10 particles can use 1000 iterations on a budget of 10000 function evaluations. A CPSO with 5 swarms of 10 particles each effectively uses 5×10 function evaluations per iteration, thus it can only train for $10000/50 = 200$ iterations. It is believed that there will be an optimal number of swarms to use for a specific problem, which will be investigated in the experiments below.

3 Product Unit Neural Networks

The product unit network was first introduced by Durbin and Rumelhart [10], and can be used in more or less any situation where the better known summation unit back-propagation networks have been used.

A network with D inputs, M hidden units and C output units is shown in Figure 3, assuming that only product units are used in the hidden layer, followed by summation units in the output layer, with linear activation functions throughout. The value of an output unit y_k for pattern p is calculated using

$$y_k = \sum_{j=0}^{M} w_{kj} \prod_{i=1}^{D} x_{i,p}^{w_{ji}}, \tag{5}$$

where w_{kj} is a weight from output unit y_k to hidden unit z_j, w_{ji} is a weight from hidden unit z_j to input unit x_i and

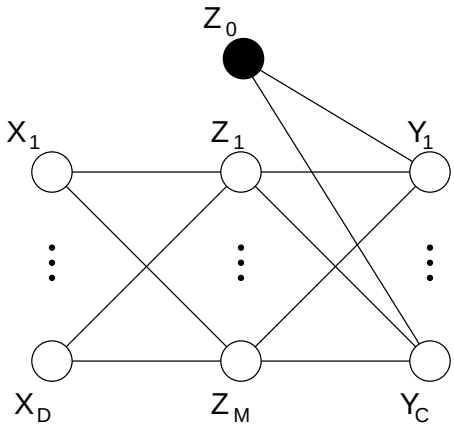

Figure 3: Neural network architecture

$\prod_{i=1}^{D} x_{i,p}^{w_{0i}} \equiv 1.0$ (the bias unit).

Note that quadratic functions of the form $ax^2 + c$ can be represented by a network with only one input unit and one hidden unit, thus a 1-1-1 PUNN can be used, compared to a summation unit network requiring at least 2 hidden units [2]. This is an indication of the increased information storage capacity of the product unit neural network [10].

Unfortunately the usual optimisation algorithms like gradient descent cannot train the PUNN with the same efficiency that they exhibit on summation unit networks, due to the more turbulent error surface created by the product term in (5). Global-like optimisation algorithms like the Particle Swarm Optimiser, the Leapfrog algorithm and Genetic Algorithms are better suited to the task of training PUNNs [2].

4 Experimental Set-up

Three classification problems were selected as case studies, ranging from a small network (Iris problem) to a large network (Glass problem). The network configurations are listed in Table 1. The data sets for these classification problems can be obtained from the UCI repository [11].

Table 1: Network configuration

Problem	Architecture	#weights
iris	4-3-3	24
ionosphere	34-2-2	74
glass	9-6-6	96

The aim of the experiments was to determine whether the CPSO converges on the (local) minimum faster than the original PSO, at the same time trying to find the optimal number of swarms to use in the CPSO. This is done by keeping the number of function evaluations, or forward propagations through the network, fixed. The final classification error, as well as the graph of the Mean Sum Squared Error (MSSE), can then be used to compare the two architectures.

All runs trained the network for 2×10^4 function evaluations. The following parameter settings were used for all swarms:

Maximum velocity v_{max}: Set to a value of 5.0. This value is relatively large with respect to the final network weights, which were usually smaller than 1.0;

Acceleration coefficients c_1, c_2: Both set to a value of 1.4961798. This value corresponds to the value used by Eberhart and Shi [12];

Inertia weight w: A w value of 0.729844 (see [12]) was used.

The CPSO, implementing the algorithm listed in Figure 2, can be configured with different 'split factors', indicating the number of swarms used, and thus also the number of dimensions handled per particle in each of the swarms. This value was varied in the experiments below.

All results reported below are the averages computed over 500 runs for each configuration.

5 Results

Table 2: Training classification error for the Iris classification problem.

Type	#swarms	Error (%)
Plain	1	1.61 ± 0.22
Split	2	1.71 ± 0.27
	3	1.53 ± 0.21
	4	1.85 ± 0.24
	5	1.47 ± 0.21
	10	2.15 ± 0.25

Tables 2,3 and 4 conform to the following format: The first column indicates the type of PSO used while the second is the split factor (or number of swarms). The last column lists the training classification error, expressed as a percentage, followed by the 95% confidence interval width.

The Iris problem produced some interesting results, indicating that certain split factors produce better results than configurations with larger split factors. For example, the

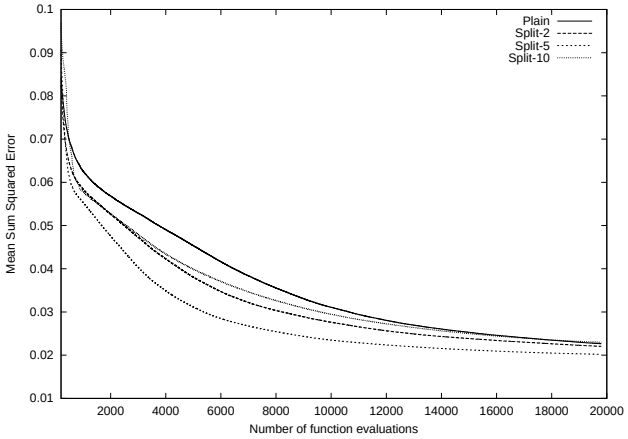

Figure 4: MSSE plot for the Iris classification problem

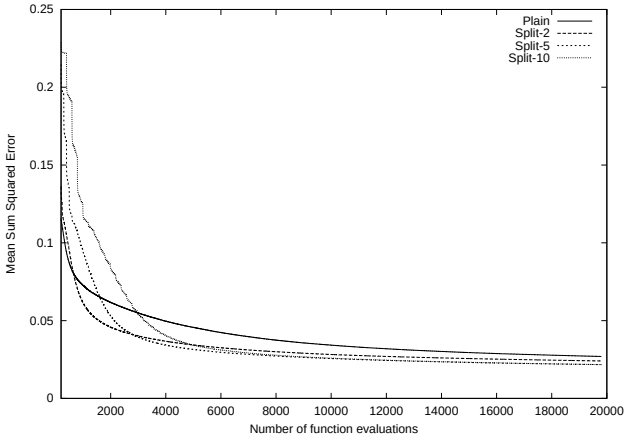

Figure 5: MSSE plot for the Ionosphere classification problem

Table 3: Training classification error for the Ionosphere classification problem.

Type	#swarms	Error (%)
Plain	1	1.21 ± 0.07
Split	2	0.98 ± 0.06
	3	0.82 ± 0.05
	4	0.74 ± 0.05
	5	0.75 ± 0.05
	10	0.79 ± 0.05
	15	0.74 ± 0.05
	20	0.74 ± 0.05

Table 4: Training classification error for the Glass classification problem.

Type	#swarms	Error (%)
Plain	1	11.72 ± 0.34
Split	2	11.72 ± 0.30
	5	11.06 ± 0.23
	10	9.72 ± 0.21
	15	9.86 ± 0.20
	20	9.21 ± 0.20

4-swarm configuration performed worse than the 3-swarm one. It appears that an odd number of swarms is a better choice for this specific problem/network configuration combination, however, the significance of this, given the small total number of weights, is questionable.

Figure 4 shows that the MSSE curve for the CPSO algorithm is significantly below that of the plain PSO most of the time for the 2 and 5-swarm configurations. The 10-swarm configuration, as can also be seen in Table 2, performed slightly worse toward the end of the training run.

The Ionosphere problem had fewer surprises, with only the 10-swarm configuration producing anomalous results in Table 3. Overall, the CPSO seems to perform significantly better than the plain PSO on this problem.

The curves in Figure 5 follow the expected pattern, with the CPSO performing well in the early stages (between 1000 and 6000 function evaluations), and levelling off toward the end. The 'start-up delay' of the CPSO can clearly be seen for the 10-swarm case — note the large MSSE during the first 1000 function evaluations. This is a known problem of the CPSO [7]: the larger the number of swarms, the greater the start-up delay. The Ionosphere problem seems to accentuate this problem somewhat. Also note that once the CPSO caught up to the plain PSO (clearly visible for the 10-swarm case), the rate of improvement was significantly better than that of the standard PSO. The CPSO appears to perform better with increasing split factors on the Ionosphere problem.

The Glass problem exhibits the now-expected improvement for the CPSO as the number of swarms is increased. From Table 4 it is clear that the CPSO performed significantly better than the original PSO when a larger number of swarms was used. There was a slight deviation from this pattern for the 15-swarm case.

Figure 6 shows a strong case for the CPSO, with performance improving consistently with larger split factors. Even to the end of the simulation run the rate of decrease in the error for the 20-swarm case seems to be greater than that of the 5-swarm case, the opposite of what was observed on the Iris problem. The glitch in the 20-swarm curve during the first 2000 function evaluations is caused by the fact that only 500 runs were used per architecture, resulting in insufficient smoothing of the curve. This implies that the 20-swarm case has a larger variance in error during the early

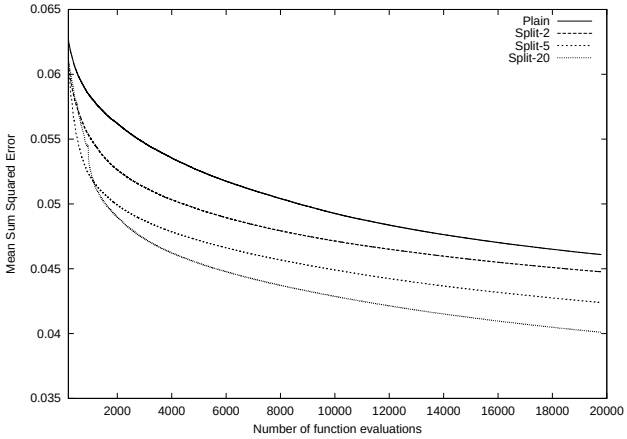

Figure 6: MSSE plot for the Glass classification problem

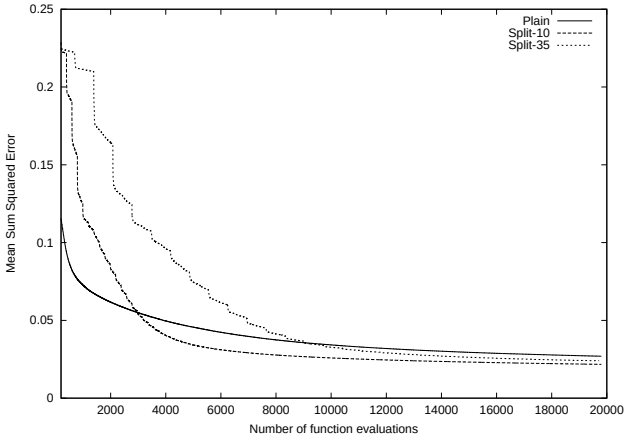

Figure 7: MSSE plot for the Ionosphere classification problem—large number of swarms

training stages, something that can also be observed for the 10-swarm case on the Ionosphere problem in Figure 5.

Figure 4 shows that a large split factor results in a slight decrease in performance (toward the end) for the Iris problem. To investigate this effect for the other two problems, large split factors of 35 and 48 were selected for the Ionosphere and Glass problems, respectively. These split factors result in about two weights per swarm, corresponding to the same number as a split factor of 10 in the Iris problem.

Figures 7 and 8 contain plots for these large split factor cases. Firstly, note that the MSSE variance at each iteration has increased, resulting a noticeably less-smooth curve. The most important property, however, is that the 35 and 48 split factor cases perform worse than the 10 and 20 split factor cases (respective to the problem). This shows that larger split factors do not necessarily lead to improved performance.

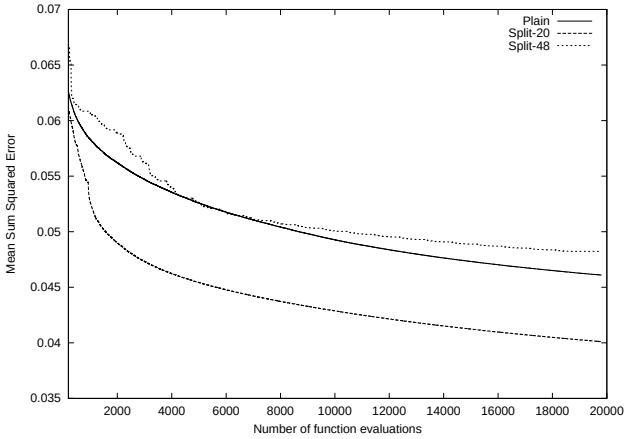

Figure 8: MSSE plot for the Glass classification problem—large number of swarms

6 Conclusions

When looking at the Mean Sum Squared Error, the CPSO outperforms the standard PSO consistently. The results indicate that generally the performance of the CPSO improves as the split factor is increased until a critical ratio (#weights/#swarms) is reached.

For the problems examined here it appears that around 5 dimensions per swarm results in good training performance, which translates to split factors of 5, 15 and 20 for the three problems, respectively. More classification problems will have to be investigated, but it would appear that the optimal split factor is thus $\lceil W/5 \rceil$ for product unit network training.

References

[1] R. C. Eberhart and J. Kennedy, "A New Optimizer using Particle Swarm Theory," in *Proceedings of the Sixth International Symposium on Micro Machine and Human Science*, (Nagoya, Japan), pp. 39–43, IEEE Service Center, 1995.

[2] A. P. Engelbrecht and A. Ismail, "Training product unit neural networks," *Stability and Control: Theory and Applications*, vol. 2, no. 1–2, pp. 59–74, 1999.

[3] F. van den Bergh and A. P. Engelbrecht, "Cooperative Learning in Neural Networks using Particle Swarm Optimizers," *South African Computer Journal*, pp. 84–90, Nov. 2000.

[4] R. C. Eberhart and X. Hu, "Human Tremor Analysis Using Particle Swarm Optimization," in *Proceedings of the Congress on Evolutionary Computation*, (Washington D.C, USA), pp. 1927–1930, July 1999.

[5] Y. Shi and R. C. Eberhart, "Empirical Study of Parti-

cle Swarm Optimization," in *Proceedings of the Congress on Evolutionary Computation*, (Washington D.C, USA), pp. 1945–1949, July 1999.

[6] Y. Shi and R. C. Eberhart, "A Modified Particle Swarm Optimizer," in *IEEE International Conference of Evolutionary Computation*, (Anchorage, Alaska), May 1998.

[7] F. van den Bergh and A. P. Engelbrecht, "A Cooperative Approach to Particle Swarm Optimisation," *IEEE Transactions on Evolutionary Computing*, 2001. Currently under review.

[8] R. C. Eberhart, P. Simpson, and R. Dobbins, *Computational Intelligence PC Tools*, chapter 6, pp. 212–226. Academic Press Professional, 1996.

[9] P. N. Suganthan, "Particle Swarm Optimizer with Neighbourhood Operator," in *Proceedings of the Congress on Evolutionary Computation*, (Washington D.C, USA), pp. 1958–1961, July 1999.

[10] 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.

[11] C. Blake, E. Keogh, and C. Merz, "UCI repository of machine learning databases," 1998. www.ics.uci.edu/∼mlearn/MLRepository.html.

[12] R. C. Eberhart and Y. Shi, "Comparing Inertia Weights and Constriction Factors in Particle Swarm Optimization," in *Proceedings of the Congress on Evolutionary Computing*, pp. 84–89, 2000.