Cooperative Learning in Neural Networks using Particle Swarm Optimizers

[a]F. van den Bergh, [a]A.P. Engelbrecht

[a]*Department of Computer Science, University of Pretoria, South Africa,* `fvdbergh@cs.up.ac.za`, `engel@driesie.cs.up.ac.za`

Abstract

This paper presents a method to employ particle swarms optimizers in a cooperative configuration. This is achieved by splitting the input vector into several sub-vectors, each which is optimized cooperatively in its own swarm. The application of this technique to neural network training is investigated, with promising results.

Keywords: *Particle swarms, cooperative learning, optimization*
Computing Review Categories: *G.1.6, I.2.6*

1 Introduction

Particle Swarm Optimizers (PSOs) have previously been used to train neural networks [6, 10] and generally met with success. The advantage of the PSO over many of the other optimization algorithms is its relative simplicity.

This paper aims to improve the performance of the basic PSO by partitioning the input vector into several sub-vectors. Each sub-vector is then allocated its own swarm.

In Section 2, a brief overview of PSOs is presented, followed by a discussion of how cooperative behavior can be implemented through a splitting technique in Section 3. Implementations, applied to both neural network training and function minimization, are described in Section 4, followed by results obtained from various case studies in Section 5.

This paper is concluded by some ideas that can serve as future research directions.

2 Particle Swarms

Particle swarms, as first described by Eberhart and Kennedy [3, 5], are best classified under the heading of evolutionary computing. The basic operational principle of the particle swarm is reminiscent of the behaviour of a flock of birds or the sociological behaviour of a group of people.

Various attempts have been made to improve the performance of the baseline Particle Swarm Optimizer. Work done by Eberhart and Shi [9, 4] focus on optimizing the update equations for the particles. Their improvements generally leads to better performance on a very large class of problems. These improvements can be (and have been) applied to the technique presented here.

Angeline [1] used a selection mechanism (as used in genetic algorithms) in an attempt to improve the general quality of the particles in a swarm. This approach lead to improved performance on some problems, but somewhat worse performance on others.

Kennedy [7] used cluster analysis to modify the update equations so that particles attempt to conform to the center of their clusters rather than attempting to conform to a global best. This approach resulted in improved performance on some problems, but slightly worse performance on others.

2.1 Inside the Particle Swarm

A swarm consists of several particles, where each particle keeps track of its own attributes. The most important attribute of each particle is its current position, as given by an *n*-dimensional vector, corresponding to a potential solution of the function to be minimized. Along with the position the particle has a current velocity, keeping track of the speed and direction in which the particle is currently traveling. Each particle also has a current fitness value (analogous to population members in a GA), which is obtained by evaluating the error function at the particle's current position. For a neural network implementation, this value corresponds to a forward propagation through the network, assuming the position vector corresponds to the weight vector of the network.

Additionally, each particle also remembers its own personal best position (yielding the lowest error) so that potentially good solutions can be used to guide the construction of new solutions. At swarm level, the best overall position amongst all particles is also recorded; upon termination of the algorithm this will serve as the answer.

During each epoch every particle is accelerated towards its own personal best as well as in the direction of the global best position. This is achieved by calculating a new velocity term for each particle based on the distance from its personal best, as well as its distance from the global best position. These two components ('personal' and 'global' velocities) are then randomly weighted to produce the new velocity value for this particle, which will in turn affect the next position of the particle during the next epoch.

2.2 Particle Swarm Parameters

A number of factors will affect the performance of the PSO. Firstly, the number of particles in the swarm affects the run-time significantly, thus a balance between variety (more particles) and speed (fewer particles) must be sought. Another important factor in the convergence speed of the algorithm is the *maximum velocity* parameter. This parameter limits the maximum jump that a particle can make in one step, thus a too large value for this parameter will result in oscillation. A small value, on the other hand, can cause the particle to become trapped in local minima. Current research indicates that using a *constriction coefficient* leads to improved performance [4].

For the implementation used in this paper, the *maximum velocity* parameter is set to the default value of 2 for neural network training sessions (the value 2 was suggested in the earlier works in PSOs [5]), but increased appropriately for the function minimization tests (typically half the range of the function [4]).

To aid the convergence of the algorithm, an inertia term (introduced in [9]) is used to attenuate the magnitude of the velocity updates over time. The attenuation factor is a linear function of the current epoch number.

2.3 Swarm Behavior

Each particle will "fly" in the direction of a better solution, weighted by some random factor, possibly causing it to overshoot its target, possibly finding a new personal or even global minimum. The interaction of the particles in the swarm creates a very good balance between straying off-course and staying close to the optimal solution.

This type of behavior seems to be ideal when exploring large error surfaces, especially with a relatively large *maximum velocity* parameter. Some particles can explore far beyond the current minimum, while the population still remembers the global best solution. This solves one of the problems of gradient-based optimization techniques, namely their poor performance in regions with very flat gradients. It also appears that the particle swarm optimizer performs well in regions where the error surface is highly perturbed, as is the case with product unit networks.

It is important to note that the PSO cannot guarantee a good solution, partly due to its stochastic nature. When compared to a second order optimization algorithm, *e.g.* scaled conjugate gradient descent, the PSO will typically have both the lower minimum error and a larger maximum error, when a number of experiments are performed.

3 Cooperative Learning

The driving force in a population of solutions in a genetic algorithm is the competition between the different members of the population. During crossover, two solutions cooperate to produce new solutions, but this serves only to create new members who will compete for the top spot.

In a cooperative learning framework, several populations will be considered simultaneously [8]. The solution vector is then split amongst the different populations according to some rule; the simplest of schemes does not allow any overlap between the space covered by different populations. Thus, to find a solution to the original problem, representatives from all the populations are combined to form the potential solution vector, which is passed on to the error function. A new dimension is now added to the survival game: cooperation between different populations.

The same concept can easily be applied to PSOs. There's already some form of cooperation between the elements in the swarm: they all share the global best position. By using multiple swarms and splitting the vector across these swarms inter-swarm cooperative behavior emerges.

Two new problems are introduced with this approach, however:

Selection: The solution vector is split into N parts, each part being optimized by a swarm with M particles, leaving $N \times M$ possible vectors to choose from. The simplest approach is to select the best particle from each swarm (how to calculate which particle is best will be discussed later). Note that this might not be the optimal choice, it is analogous to pairing highly fit members of a GA population only with other highly fit members, which is known to have a detrimental effect on training performance.

Credit assignment: This is analogous to the real-world problem of handing out group projects to students: Who gets the credit for the final product? In terms of swarms, how much credit should each swarm be awarded when the combined vector (built from all the swarms) results in a better solution? A simple solution is to give all swarms an equal amount of credit.

The next section will present an argument to help explain why splitting the vector results in improved performance.

3.1 Two Steps Forward, One Step Back

Consider an n-dimensional vector $\mathbf{v}$, constrained to the error surface by the function to be minimized. Somewhere in this vector three components, v_a, v_b, v_c, $1 \le a < b < c \le n$, will now be examined. Assume that the optimal solution to the problem requires that all three components must have a value of say 20.

Now consider a particle swarm containing, in particular, two vectors $\mathbf{v}_j$ and $\mathbf{v}_k$. Let's assume that $\mathbf{v}_j$ is currently the global best solution in the swarm, so that $\mathbf{v}_k$ will be drawn towards it in the next epoch. Assume that the three components $\mathbf{v}_{j,a}, \mathbf{v}_{j,b}, \mathbf{v}_{j,c}$ have the values (17, 2, 17), respectively. The equivalent components in vector $\mathbf{v}_k$, namely $\mathbf{v}_{k,a}, \mathbf{v}_{k,b}, \mathbf{v}_{k,c}$, have the values (5, 20, 5), respectively. In the next epoch, the vector $\mathbf{v}_k$ might be updated so that it now contains the sub-vector (15, 5, 15) at positions $\mathbf{v}_{k,a}, \mathbf{v}_{k,b}, \mathbf{v}_{k,c}$. If it is assumed that the function at the new $\mathbf{v}_k$ yields a smaller error value, this will be considered an improvement. However, the valuable information contained in the middle component ($v_{k,b}$) has been lost, as the optimal solution requires a sub-vector of (20,20,20).

The reason for this behavior is that the error function is computed only after all the components in the vector have been updated to their new values. This means an improvement in two components (two steps forward) will overrule a potentially good value for a single component (one step back).

One way to overcome this behavior is to evaluate the error function more frequently, for example, once for every time a component in the vector has been updated. A problem still remains with this approach: evaluation of the error function is only possible using a complete n-dimensional vector. Thus, after updating component $\mathbf{v}_{k,b}$, $n-1$ values for the other components of the vector still have to be chosen. A method for doing just this is presented below.

3.2 N Steps Forward

At the one extreme a single swarm consisted of a number of n-dimensional vectors. The other extreme would thus be to have n different swarms, each representing a single component of an n-dimensional vector. As mentioned above, the error function can only be evaluated using an n-dimensional vector, so representatives from each swarm must be combined to form such a vector so that the function can be evaluated.

The simplest way to construct such a vector is to use the best particle from each swarm for a particular component, as outlined in Figure 1.

```
Create and initialize $n$ 1-dimensional PSOs $S_1$–$S_n$.
**Repeat:**
    Select best particle $b_1$–$b_n$ from each of $S_1$–$S_n$ resp.
    **For** $k \in [1..M]$, $i \in [1..n]$ **:**
        Select $p = k^{\text{th}}$ particle from swarm $S_i$
        Construct a vector $\mathbf{v} = (b_1, b_2, \ldots, p, \ldots, b_n)$
        $E(\mathbf{v})$ = error function at $\mathbf{v}$
        Set fitness of particle $k$ in swarm $S_i$ to $E(\mathbf{v})$
        Update best fitness of $b_1$–$b_n$ if necessary
    Perform normal PSO updates on $S_1$–$S_n$
**Until** stopping criterion is met
```

Figure 1: Pseudo code for split swarm approach, assuming M particles per swarm

Note that the algorithm presented in Figure 1 now keeps track of the optimal value for each component of the n-dimensional vector in a separate swarm. This implies that the error with respect to each component will be a non-increasing function of the epoch number, so a good value for a specific component is never forgotten.

Care must be taken to keep track of the global minimum error. Whenever a combined vector ($\mathbf{v}$) with a lower error is found, the fitness values of the best particle in *all* the swarms are updated to this value, as all these particles participated in the creation of this improved vector. This is one solution to the credit assignment problem (mentioned above), and the one chosen for the implementation used to obtain the results presented here.

This algorithm optimizes each variable separately *within each epoch*, but cooperates with other variables between epochs.

4 Implementation

The optimization method described in Section 3 involves a trade-off: increasing the number of swarms leads to a directly proportional increase in the number of error function evaluations, as one function evaluation is required for each particle in each swarm during each epoch.

The total number of error function evaluations can thus be summarized as:

$$N_f = N_s \times N_p \times E, \tag{1}$$

where N_f is the number of error function evaluations, N_s the number of swarms used, N_p the number of particles per swarm and E the number of epochs allowed for training.

An extreme example illustrates this point: Consider that a single swarm using n-dimensional vectors is allowed to train for $2n$ epochs. To keep the number of error function evaluations constant (thus the execution time), a system using n 1-dimensional swarms will only be allowed to train for 2 epochs. Clearly the swarm optimizer will require more than 2 epochs to find a good solution (assuming that the single-swarm set-up was able to find a good solution in $2n$ epochs).

One way to increase the number of allowed epochs is to reduce the splitting factor. In the above example, splitting the n-dimensional vector across only two swarms will allow n epochs for optimization, which is more likely to succeed than only two epochs!

4.1 Neural Network Architectures

The neural network is trained by minimizing an error function in $W = (D+1) \times M + (M+1) \times C$-dimensional space (See Figure 2). PSOs generate possible solutions in this W-dimensional space using a forward propagation through the neural network to obtain the value of the error function for each particle in the swarm. This error value is used directly as the particle's fitness, so that constructing a particle with a lower error value is synonymous with learning in the network. A forward propagation through the network is a computationally expensive task, so the aim is to find the best possible solution using a limited number of forward propagations through the network.

To test the effectiveness of cooperative swarm configurations, a number of tests were performed on 2-layer feed-forward neural networks. A network as depicted in Figure 2 was used, with D input units, M sigmoidal hidden units and C linear output units.

The following architectures were selected for testing:

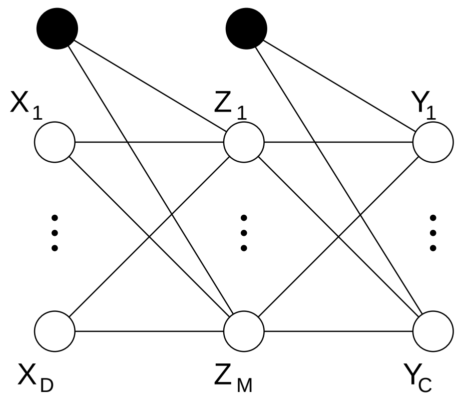

Figure 2: Neural network architecture

4.1.1 Plain

The plain architecture takes the weights from the different layers and serializes them into a $(D+1)\times M+(M+1)\times C$-dimensional vector, which is optimized using a single particle swarm. This architecture is used as the baseline for the benchmarks.

4.1.2 Lsplit

The Lsplit, or 'Layer'-split architecture splits the network weights between two swarms. The first swarm contains the $(D+1)\times M$ weights from the first layer, while the second swarm optimizes the $(M+1)\times C$ second-layer weights. This architecture was chosen on the assumption that the relative scales of the weights in these two layers differ. The number of allowed forward propagations, N_f, is kept constant by halving E, the number of allowed epochs during training.

4.1.3 Esplit

The Esplit, or 'Even'-split architecture takes the serialized $(D+1)\times M+(M+1)\times C$-dimensional vector (as used in the *plain* architecture) and splits it in half. The assumption here was that splitting the vector into a 90%/10% configuration (as sometimes happens with Lsplit) does not leave the 90% part with sufficient freedom to maneuver, thus an even split would result in the optimal 2-way split. The number of epochs are halved again to keep the number of error function evaluations constant.

4.1.4 Nsplit

The Nsplit architecture attempts to build on the Lsplit approach by splitting the weights according to function. For each hidden and output unit, a swarm is created containing all the weights entering that unit. This results in M $(D+1)$-dimensional vectors plus C $(M+1)$-dimensional vectors, for a total of $M+C$ swarms. By applying (1) it becomes clear that a large number of hidden or output units would put this architecture to a severe disadvantage, as the number of allowed epochs will be significantly reduced. Thus, if the plain architecture was allowed to train for E epochs, the Nsplit architecture was allowed only $E/(M+C)$ epochs.

4.2 Function Minimization Architectures

The following two architectures were used to test the vector splitting approach in a function minimization environment:

4.2.1 Plain

As was the case with the neural network experiments, the plain architecture makes use of only a single swarm consisting of n-dimensional vectors. This again serves as the baseline for comparison.

4.2.2 Dsplit

The Dsplit architecture is the extreme opposite of the plain architecture, splitting the n-dimensional vector across n 1-dimensional swarms. It is called Dsplit (in lieu of Nsplit) to avoid confusion with the definition of Nsplit given above.

5 Case Studies

5.1 Neural Network Experiments

The data sets used in this section were obtained from the UCI machine learning repository [2]. All experiments were performed using 500 runs per architecture.

The tables appearing in this section show the classification error, expressed as a percentage, followed by the 95% confidence interval width. Each table lists the total error (train + test) as well as the training error. Note that the Training error is the more significant of the two, as no attempt was made to prevent over-fitting.

5.1.1 Iris

The Iris classification problem consists of 150 patterns falling into three classes. This is considered to be a simple classification example, as the three classes are (almost) linearly separable. A 4-input, 3-hidden and 3-output network architecture was used.

From Table 1 it is clear that the 2-way split architectures (Lsplit and Esplit) perform significantly better than the plain architecture. Early on during training the difference is in the order of 267%, settling to 241% later on. Figure 3 presents the training errors graphically. It is interesting to note that the slope (rate of improvement caused by more iterations) of the split architectures seem to be less steep than the plain approach. The Nsplit architecture is leading by a small amount.

Table 2 shows the results of a convergence test. Here, a training session is said to converge when it fails to improve for 200 consecutive epochs. The second column in this table lists the average number of error function evaluations (forward propagations in the network) used during training, until the network converged. The 2-way split

Type	Epochs	Error (Total)	95% dev.	Error (Train)	95% dev.
plain	500	6.00	± 0.43	5.52	± 0.41
Lsplit	250	2.81	± 0.23	2.32	± 0.23
Esplit	250	2.59	± 0.25	2.06	± 0.24
Nsplit	83	2.39	± 0.24	1.73	± 0.22
plain	1000	3.99	± 0.34	3.69	± 0.32
Lsplit	500	2.02	± 0.17	1.55	± 0.17
Esplit	500	2.07	± 0.23	1.53	± 0.22
Nsplit	167	1.88	± 0.23	1.49	± 0.23

Table 1: Iris plain *vs* split architectures

Type	Fwd. Props.	Error (Total)	95% dev.	Error (Train)	95% dev.
plain	27468	5.03	± 0.56	4.50	± 0.57
Lsplit	64165	1.24	± 0.16	0.72	± 0.16
Esplit	62573	1.10	± 0.07	0.59	± 0.06
Nsplit	300000	1.24	± 0.08	0.64	± 0.07

Table 2: Iris convergence tests

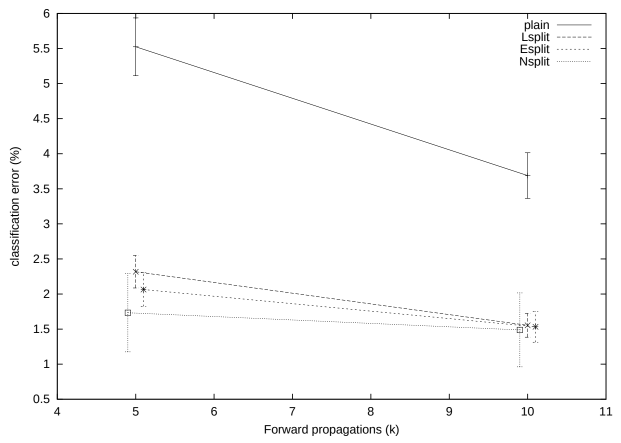

Figure 3: Iris classification error (train)

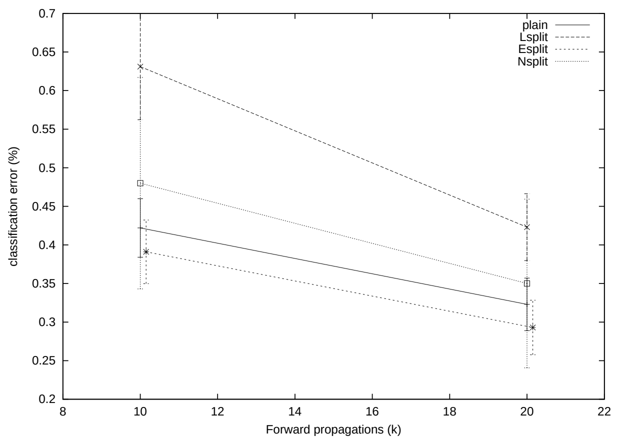

Figure 4: Iono classification error (train)

architectures used roughly twice as many function evaluations as the plain method, but much lower classification errors result. The Nsplit architecture completely failed to converge, which probably means that the threshold used in the convergence test is too sensitive. Besides the failure to converge, the error after 300000 epochs still exceeds the Esplit approach.

5.1.2 Ionosphere

The ionosphere problem has a much higher input dimension than the Iris problem, but only two output classes. This results in a 34-input, 5-hidden and 2-output network architecture.

Due to the high input dimension (versus the output dimension), the Lsplit architecture becomes highly unbalanced at a 175:12 ratio, or a 94%/6% split. Having to train a 175-dimensional vector using 500 epochs (Lsplit) *vs* 187-dimensional vector using 1000 epochs (plain) explains why the Lsplit architecture performs worse than the plain method. The Esplit method regains this ground by (narrowly) beating the plain architecture. Figure 4 graphically illustrates this fact.

The Nsplit architecture falls somewhere between the Lsplit and the plain architectures. This is probably caused by the higher interdependency between the variables, *i.e.* some weights in the network may be correlated.

5.1.3 Glass

The glass problem is difficult to solve, possibly because of its severely skewed class distribution. An 8-input, 6-hidden, 6-output unit network was used to perform the experiments.

Table 4 shows that the Esplit method again outperformed all the other architectures. The Lsplit architecture is relatively well balanced, at a 54:42 ratio, so that it performs similarly to its sibling, Esplit. Nsplit seems to perform well, beating the other architectures at low epoch counts and drawing even at high counts. Figure 5 shows how dramatically the plain network improved by going from 500 epochs to 1000 epochs.

Type	Epochs	Error (Total)	95% dev.	Error (Train)	95% dev.
plain	1000	0.91	± 0.05	0.42	± 0.04
Lsplit	500	1.17	± 0.08	0.63	± 0.07
Esplit	500	1.09	± 0.06	0.39	± 0.04
Nsplit	143	1.07	± 0.06	0.48	± 0.04
plain	2000	0.86	± 0.04	0.32	± 0.03
Lsplit	1000	1.06	± 0.06	0.42	± 0.04
Esplit	1000	1.01	± 0.06	0.29	± 0.04
Nsplit	286	1.03	± 0.08	0.35	± 0.04

Table 3: Iono plain *vs* split architectures

Type	Epochs	Error (Total)	95% dev.	Error (Train)	95% dev.
plain	500	13.79	± 0.74	13.78	± 0.74
Lsplit	250	9.87	± 0.39	9.86	± 0.39
Esplit	250	9.41	± 0.36	9.40	± 0.36
Nsplit	42	8.57	± 0.35	8.58	± 0.35
plain	1000	11.08	± 0.53	11.08	± 0.53
Lsplit	500	9.33	± 0.26	9.31	± 0.26
Esplit	500	8.90	± 0.25	8.90	± 0.25
Nsplit	83	8.94	± 0.22	8.93	± 0.23
plain	2000	9.83	± 0.36	9.83	± 0.36
Lsplit	1000	8.87	± 0.21	8.87	± 0.21
Esplit	1000	8.74	± 0.23	8.72	± 0.23
Nsplit	167	8.70	± 0.24	8.72	± 0.24

Table 4: Glass plain *vs* split architectures

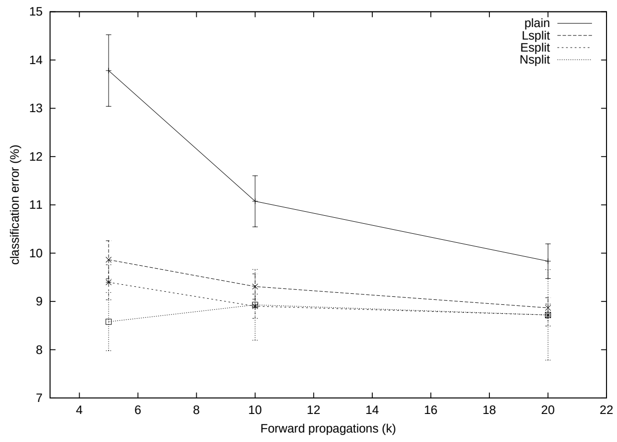

Figure 5: Glass classification error (train)

Type	Epochs	Error (Total)	95% dev.
plain	500	8.46e+00	± 2.26e-01
Dsplit	25	5.73e-05	± 4.23e-06
plain	1000	7.43e+00	± 2.02e-01
Dsplit	50	1.75e-10	± 3.32e-11

Table 5: Rastrigin function plain *vs* split architectures

5.2 Function Minimization Experiments

By including two pure function minimization problems it is possible to investigate the properties of split swarms in more detail, as more information is available on the nature of the error function as opposed to the error function of a neural network classification problem.

5.2.1 Rastrigin Function

The Rastrigin function is quite easy to optimize, as it is basically an n-dimensional parabola with some cosine "bumps" on its surface. Formally, it is given by

$$f(\mathbf{x}) = 3.0n + \sum_{i=1}^{n} \left[x_i^2 - 3.0\cos(2\pi x_i)\right], \qquad (2)$$

where $n = 20$ and $-5.12 \leq x_i \leq 5.12$, with its global minimum at $\mathbf{0}$.

Since there are no cross-component products, it is possible to minimize this function component-wise.

As can be seen in Table 5, the Dsplit architecture is decidedly superior, yielding errors that are several orders of magnitude smaller than the plain architecture. Another

Type	Epochs	Error (Total)	95% dev.
plain	2500	9.06e-02	± 2.44e-03
Dsplit	250	4.68e-02	± 1.15e-03
plain	5000	8.08e-02	± 3.17e-03
Dsplit	500	4.43e-02	± 1.23e-03

Table 6: Griewangk function plain *vs* split architectures

benefit of the Dsplit approach is the reduced execution time. Note that although the number of error function evaluations were the same for both approaches, the overhead between error function evaluations is reduced in the Dsplit architecture as it deals with smaller vectors. This results in an effective speed-up factor of 2.5 for the same number of error function evaluations.

5.2.2 Griewangk Function

The Griewangk function cannot easily be minimized component-wise, as it contains a product term:

$$f(\mathbf{x}) = 1 + \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right), \qquad (3)$$

where $n = 10$ and $-600 \leq x_i \leq 600$, with a global minimum at $\mathbf{0}$.

Table 6 shows that the Griewangk function, having more inter-dependencies between its component variables, benefits less from the Dsplit architecture, although the reduction is still by a factor of two.

This example serves to characterize the split-swarm approach to optimization: Problems that have a low inter-dependency between vector components lend themselves well to split-swarm techniques, while problems with high inter-dependency tend to show very little gain.

6 Conclusion

Splitting a vector to be optimized across multiple swarms, be it for neural network training or function minimization, appears to improve performance on all the problems tested so far. One possible explanation for this phenomenon is that the splitting leads to finer-grained credit assignment, reducing the chance of neglecting a possibly good solution for a specific component of the vector.

Although the Esplit method ousted the plain architecture on all the neural network tests, it was not convincing in its victory on the Ionosphere problem. This is most likely due to a high degree of interdependency between the weights in the network, as the function minimization experiments shows that interdependent variables tend to reduce the effectiveness of the split approach.

7 Future Work

It has been observed in the results above that the split architectures are sensitive to the degree of inter-dependency between the variables, compared to the plain architecture. By using both an unsplit and a group of split swarms in conjunction, it might be possible to gain the best of both worlds. The split swarm would continue as usual, but during each epoch the vector formed by combining the best of all the split swarms could be injected into the unsplit swarm, serving as a new ‘attractor’ — especially if it has a lower error value than the previous best particle in the unsplit swarm.

Another direction might be to find the critical splitting factor (the number of disjoint swarms) where the reduction in the allowed number of epochs (for a fixed error function evaluation budget) outweighs the benefit of having more independent sub-vectors. It would be interesting to see if there is a universal constant governing this behavior.

Lastly, different methods of addressing the credit assignment or the selection problems will be investigated.

References

[1] P.J. Angeline. Using Selection to Improve Particle Swarm Optimization. In *Proceedings of IJCNN’99*, pages 84–89, Washington, USA, July 1999.

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

[3] Russ C. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Proc. Sixth Intl. Symposium on Micro Machine and Human Science*, pages 39–43, Nagoya, Japan, 1995. IEEE Service Center.

[4] Russ C. Eberhart and Y. Shi. Comparing inertia weights and constriction factors in particle swarm optimization. In *Proceedings of the 2000 Congress on Evolutionary Computing*, pages 84–89, 2000.

[5] Russ C. Eberhart, Pat Simpson, and Roy Dobbins. *Computational Intelligence PC Tools*, chapter 6, pages 212–226. AP Professional, 1996.

[6] AP Engelbrecht and A Ismail. Training product unit neural networks. *Stability and Control: Theory and Applications*, 2(1–2):59–74, 1999.

[7] J. Kennedy. Stereotyping: Improving Particle Swarm Performance With Cluster Analysis. In *Proceedings of the 2000 Congress on Evolutionary Computing*, pages 1507–1512, 2000.

[8] Mitchell A. Potter and Kenneth A. De Jong. A Cooperative Coevolutionary Approach to Function Optimization. In *The Third Parallel Problem Solving from Nature*, pages 249–257, Jerusalem, Israel, 1994. Springer-Verlag.

[9] Y. Shi and Russ C. Eberhart. A Modified Particle Swarm Optimizer. In *Proceedings of IJCNN’99*, pages 69–73, Washington, USA, July 1999.

[10] F. van den Bergh. Particle Swarm Weight Initialization in Multi-layer Perceptron Artificial Neural Networks. In *Development and Practice of Artificial Intelligence Techniques*, pages 41–45, Durban, South Africa, September 1999.