Adaptive Particle Swarm Optimisation for High-Dimensional Highly Convex Search Spaces

Dean Tsou and Cara MacNish
School of Computer Science & Software Engineering
The University of Western Australia
35 Stirling Hwy, Crawley WA 6009
tsoud01@csse.uwa.edu.au, cara@csse.uwa.edu.au

Abstract- The Particle Swarm Optimisation (PSO) algorithm has been established as a useful global optimisation algorithm for multi-dimensional search spaces. A practical example is its success in training feed-forward neural networks. Such successes, however, must be judged relative to the complexity of the search space.

In this paper we show that the effectiveness of the PSO algorithm breaks down when extended to high-dimensional "highly convex" search spaces, such as those found in training recurrent neural networks. A comparative study of backpropagation methods reveals the importance of an adaptive learning rate to their success. We briefly review the physics of the particle swarm optimiser, and use this view to introduce an analogous adaptive time step. Finally we demonstrate that the new adaptive algorithm shows improved performance on the recurrent network training problem.

Keywords: Particle Swarm Optimisation, Recurrent Neural Network, Regular Languages, Adaptive Learning Rate.

1 Introduction

The Particle Swarm Optimisation (PSO) algorithm has been established as a useful global optimisation algorithm for multi-dimensional search spaces, comparing well with other approaches such genetic algorithms. A practical example is its success in training feed-forward neural networks.

Neural network training has traditionally been carried out using the backpropagation (BP) gradient descent algorithm (Rumelhart, Williams & Hinton 1986). This technique has proved to be effective for training feed-forward neural networks (FFN) which use summation unit functions and continuously differentiable transfer functions. However its application is limited in more complex networks.

Recently, global optimisation techniques have been used to train neural networks as they are applicable to a wider variety of networks such as Product Unit Networks (PUNN) (Ismail & Engelbrecht 2000), networks with arbitrary transfer functions, and recurrent neural networks (RNN) (Elman 1990). The Particle Swarm Optimisation (PSO) algorithm, for example, has been widely studied for training FFNs (El-Gallad, El-Hawary, Sallam & Kalas 2001, Mendes, Cortez, Rocha & Neves 2002, van den Bergh & Engelbrecht 2000, Kennedy & Eberhard 2001) and has been successfully applied to training product unit networks (Ismail & Engelbrecht 2000).

In this paper we investigate the particle swarm optimiser's ability to train RNNs. We show that the complex search space generated by RNNs raises difficulties that render the traditional PSO algorithm ineffective, and present a refined version of the particle swarm optimiser that succeeds where the traditional algorithm fails.

In Section 2 we briefly review the particle swarm model. Section 3 introduces the application domain — recurrent neural networks and the regular grammar learning problem — and outlines the test cases used in subsequent experiments. Section 4 presents the results of training RNNs using the traditional PSO algorithm, and makes some conjectures about the nature of the search space and why the PSO algorithm fails to find suitable solutions. Section 5 revisits backpropagation algorithms and presents empirical evidence showing that an adaptive learning rate is critical to their success in this space. Section 6 describes our new version of the PSO, APSO, which incorporates an analogous adaptive component. The improved results from applying this algorithm are presented in Section 7, and Section 8 concludes the paper.

2 The Particle Swarm Model

The PSO algorithm was originally inspired by animal group behaviour, particularly bird flocking and fish schooling. Kennedy & Eberhart (1995) suggested that "social sharing of information among conspeciates offers an evolutionary advantage". For example, when one member of a flock of birds lands at a source of food, other members will modify their flight paths and tend to converge around the successful individual, increasing their own chances of success.

The PSO algorithm seeks to utilise this principle to find good solutions to global optimisation problems. A population of potential solutions "flies" through a multi-dimensional solution space, with each member of the population continually adjusting its position and velocity according to its own experience (or *fitness*) and the experience of other members of the population. The result is the appearance that a population of particles seem to move together, yet each particle has a randomness associated with its behaviour (Ismail & Engelbrecht 2000). (Some versions of the PSO algorithm use velocity capping to ensure convergence. The result is that the population appears more like a swarm of insects than a flock of birds, hence the name PSO.)

Ideally the population will converge on an optimal (or at least "good") solution. However this is not always the case. We will show that for the high-dimensional problem of RNN training the traditional PSO algorithm fails to find acceptable solutions.

In the basic version of the PSO algorithm each particle in the population manipulated according to the following assignment statements:

$$x_{id}^{t} = x_{id}^{t-1} + v_{id}^{t-1} \quad (1)$$

$$v_{id}^{t} = v_{id}^{t-1} + c_1 * rand() * (p_{id}^{t} - x_{id}^{t}) + c_2 * Rand() * (p_{gd}^{t} - x_{id}^{t}) \quad (2)$$

Here x_{id}^{t} and v_{id}^{t} are the dth dimensional component of the position and velocity of the i^{th} particle at time step t. p_{id}^{t} is the d^{th} component of the best (fittest) position the i^{th} particle has accomplished by time step t, and p_{gd}^{t} is the d^{th} component of the global best position achieved in the population by time step t.

The constants c_1 and c_2 are known as the "cognition" and "social" factors respectively as they control the relative strengths of individualistic and collective behaviour of each particle. Finally, $rand()$ and $Rand()$ are two different random numbers in the range of 0 to 1 and are used to enhance the exploratory nature of the PSO.

3 The Problem Domain

3.1 Recurrent Neural Networks

A neural network is a network of nodes (neurons) connected via unidirectional links, each with an associated weight. In a feed-forward network, neurons are arranged into layers, with neurons from layer *l* feeding signals forward typically to neurons in layer *l+1*. Eventually the signal propagates to the neurons in the output layer. Individual neurons take the values on their input links, process them according to a unit and transfer function, and send the results to their output links. The unit function combines input signals into one signal, and the transfer function uses the combined input signal to determine the neuron's output value. We will use networks with summation unit functions and logarithmic sigmoid transfer functions (Haykin 1994).

A feed-forward network implements a non-linear function from an input vector to an output vector. By contrast, recurrent neural networks typically process sequences of input vectors, producing a corresponding sequence of output vectors. The output vector at any given step[1] in the sequence is a function of the current input and previous inputs.

In a recurrent neural network the output values from each node of a layer *l* at step *i* are stored and fed back into the network as input into nodes of layer *l* or lower, at step *i+1*. These links are called recurrent (or feedback) links, and the data that they provide for the next step can be viewed as memory of the network's state.

Sometimes, a context layer is introduced to assist in visualisation, as shown in Figure 1. The output from each node of a hidden layer *l* is fed into a corresponding context layer neuron. The context layer neurons then feed this value back into hidden layer *l* or lower at the next step. If the recurrent network is visualised in this manner, it is important to note that a link from the hidden layer to the context layer has a constant weight of 1 and is not altered in the training process. The recurrent links are the links from the context layer to the hidden layer. Recurrent links, like feed-forward links, have variable weights associated with them.

[1]We will use the term *step* for each successive input to an RNN from a sequence, to avoid confusion with *iterations* in the PSO, each of which evaluates an entire sequence.

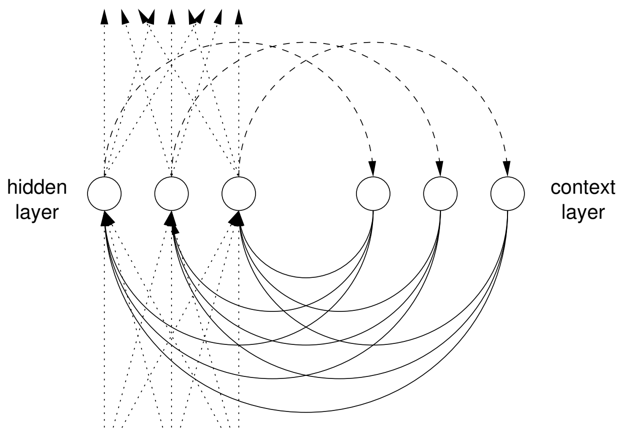

Figure 1: A portion of an RNN. The circles represent neurons. The dotted lines indicated connections from the previous layer, and to the subsequent layer. The dashed lines indicate storage from the hidden layer to the context layer (and have a weight of 1.) The solid lines are recurrent links.

3.2 Applying the PSO to RNNs

The most common method of training neural networks involves adjusting link weights. In such training scenarios, we can view a network at any given time as a point (or vector) in a network solution space. The solution space contains a mapping from the values of a network's weights to the value of the network's fitness. In supervised training, the fitness is usually an error value calculated as some function of the network's output vector and a vector of desired output values. The goal of training is to locate an optimal (or close to optimal) point in the solution space, minimising the error.

Using the PSO algorithm, each "particle" consists of a vector representing a weighted neural network. Thus one can think of the algorithm as "flying" over the multi-dimensional error surface looking for good network solutions.

The PSO has shown potential in optimising many applications for FFNs. It has the advantage of not becoming easily trapped in local minima, being exploratory, and it is simple to implement. Furthermore, the PSO does not impose a restraint on the type of transfer functions, unlike gradient descent methods such as backpropagation. The PSO has also been demonstrated to work effectively in optimising PUNNs, despite the solution space being extremely convoluted (Ismail & Engelbrecht 2000).

3.3 Learning Regular Language Rules

Regular Language prediction is a common benchmarking application for assessing the effectiveness of a training method for RNNs and will be used as the application do-

	C	Vl	I	H	B	V
b	1	0	1	0	0	1
d	1	0	1	1	0	1
g	1	0	1	0	1	1
a	0	1	0	0	1	1
i	0	1	0	1	0	1
u	0	1	0	1	1	1

Table 1: This table defines the alphabet of our regular language as binary strings. For the interpretation of each column refer to Elman (1990).

main in this paper. Regular languages consist of words or sequences that can be constructed from an alphabet of symbols according to a set of rules known as a regular grammar.

We will demonstrate our results using a simple regular language defined by Elman (Elman 1990) and used by subsequent authors. The alphabet of the language consists of the letters b, d, g, a, i, u, each represented by a 6 bit vector shown in Table 1. (Each bit has a phonetic significance. Refer to Elman (1990) for further details.)

These 6 letters are combined according to the following set of rules:

- *b* must be followed by an *a*;
- *d* must be followed by two *i*'s;
- *g* must be followed by three *u*'s.

In the prediction task, at each step successive vectors (letters) from a sequence conforming to these rules are applied to the inputs. The outputs should represent the next vector (letter) in the sequence. The task of the PSO is to train (or, more accurately, find) a network that succeeds at the prediction task.

Elman trained an RNN to learn these language rules using conventional gradient descent techniques (Elman 1990). RNNs are thus known to be capable of performing such a task.

3.4 Test Cases

To give an accurate assessment of the effectiveness of the PSO in training RNNs we require a test set with several grades of difficulty. In the sequence prediction task, this can be achieved by varying the amount of "history" that the network must take into account in making a correct prediction. Intuitively this can be thought of as the amount that it must "remember".

In the experiments that follow we use the following test sequences, listed as input/output pairs in order of increasing complexity.

- (*diib*,*iiba*): This is the simplest case we examine. The RNN has to learn that *d* is followed by an *i* and *b* is followed by an *a*. In addition to learning these simple rules (which require no "memory" of the sequence), it must also learn that *i* is followed by an *i* when it is preceded by a *d*, but it is followed by a *b* when it is preceded by another *i*. This requires that the RNN uses information from the previous time step as input for the current time step.
- (*diibaguuub*,*iibaguuuba*): Here the RNN must learn the rules associated with *diib*, and it must also learn two additional complex rules. Firstly, if *u* is only preceded by a single *u*, then another *u* must succeed it. Secondly, if *u* is preceded by two *u*s, then *b* should succeed it. In this instance, the RNN must learn to "look back" two time steps.
- (*diibaguuubadiidiiguu*,*iibaguuubadiidiiguuu*): This final sequence embodies the three rules of our regular language example. For the RNN to learn this sequence, not only must the RNN learn the rules associated with *diibaguuub*, it must also learn that *i* is not necessarily followed by *b* if it is preceded by another *i*. Similarly, *u* is not necessarily followed by a *b* if it was preceded by two other *u*s. Also, *a* is not necessarily followed by a *g*.

4 Case Study I: Training RNNs with the PSO

To assess the ability of the PSO as a training algorithm for RNNs, we trained the RNN on each sequence of our regular language as outlined below.

4.1 Method

4.1.1 Parameters

For comparability with Elman (1990) we used the same size networks containing 20 hidden nodes. Note that the number of weights, and therefore the dimension of the solution space and length of a solution vector, increases quadratically with the number of hidden nodes.

We used a population size of 50 and a maximum number of iterations of 5000. Each experiment was repeated over 4 trials (which was limited by computing resources).

Both the cognitive factor and social factor were set at the standard value of 2 (Kennedy & Eberhart 1995). Throughout this particular experiment, velocity capping (Clerc & Kennedy 2002) was used to restrict each individual velocity component such that the magnitude of the overall velocity does not exceed 5.

4.1.2 Fitness Function

As our fitness (minimisation) function we used the average mean squared error (MSE), calculated by finding the MSE of each letter in the sequence and averaging the result. We defined a good solution as achieving an average MSE of less than 0.001.

4.2 Results

For each sequence described in Section 3.3, data were collected on success or failure, the number of *epochs* (iterations) used before a good solution was found or no significant improvement was observed, and best error achieved. Failure in training occurs if training runs over the maximum

Table 2: Summary of results from 4 training runs using the original PSO on the string *diib*.

Run	Result	Best Error	Iterations
1	Success	9.380×10^{-4}	102
2	Success	9.009×10^{-4}	95
3	Success	8.923×10^{-4}	86
4	Success	7.934×10^{-4}	76

Table 3: Summary of results from 4 training runs using the original PSO on the string *diibaguuub*.

Run	Result	Best Error	Iterations
1	Success	8.767×10^{-4}	369
2	Failure	4.059×10^{-2}	527
3	Failure	3.033×10^{-3}	382
4	Failure	8.469×10^{-3}	326

allowable epoch, 5000, without reaching an error rate of less than 0.001. Summaries of results are shown in Tables 2, 3, and 4.

From the results it can be seen that whilst the PSO efficiently solved the learning problem on shorter strings with simple rules such as *diib*, it performed poorly and in the majority of cases failed to find an acceptable solution on the strings with more complex rules.

4.3 Discussion

It should be noted that in cases of failure the algorithm did not make any significant improvement after a relatively small number of iterations. Thus increasing the allowable number of epochs is unlikely to improve performance.

One obvious hypothesis for the poor performance is that the population is simply converging on a local minimum that is "out of reach" of a better solution. However this does not appear to be the case for a number of reasons. First, increasing the "exploratory" nature of the algorithm (for example by giving high random weightings to component terms) did not produce an improvement in performance. Secondly, and perhaps more importantly, gradient descent methods such as those used by Elman have been able to find satisfactory solutions in this space (from random starting positions). This suggests that sufficient solutions are available in the space, but the PSO algorithm is failing to converge on them.

It is not possible to directly visualise the error function surface within the solution space due to its high dimensionality — in this case over 500 dimensions. However, slices of the function can give us some clues as to its behaviour.

Table 4: Summary of results from 4 training runs using the original PSO on the string *diibaguuubadiidiiguu*.

Run	Result	Best Error	Iterations
1	Failure	8.008×10^{-2}	301
2	Failure	3.896×10^{-2}	213
3	Failure	3.690×10^{-2}	415
4	Failure	4.196×10^{-2}	312

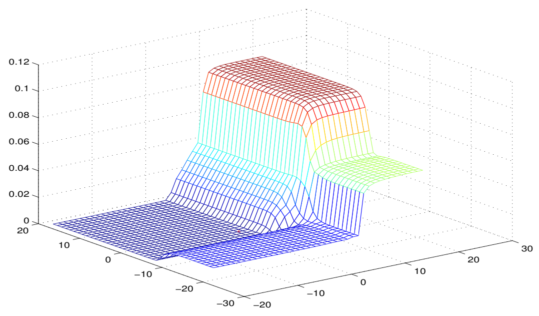

Figure 2: An error surface plot of two recurrent links' (the recurrent links connecting the first and second hidden nodes to themselves) weights around a solution minimum.

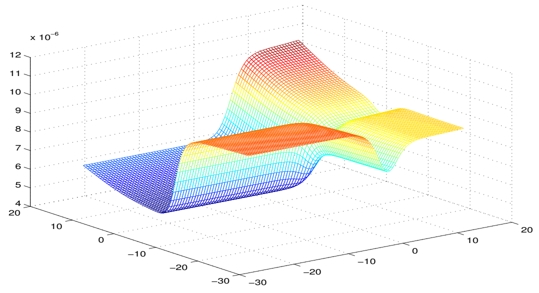

Figure 3: An error surface plot of two recurrent links' (the recurrent links connecting the sixteenth and seventeenth hidden nodes to the first hidden node) weights around a solution minimum.

To examine the surface we first found solutions for the complex strings using gradient descent (backpropagation) methods. We were then able to fix all but two weights. Figures 2 and 3 show illustrative three dimensional slices (two weights plus error function value) from of the error surface around a solution minimum. They are plotted by sampling 20 points, with a step size of 1 between each point, on either side of the solution minimum in both dimensions.

Both figures show broad flat planar regions and steep steps. This supports a conjecture, proposed by Magoulas, Vrahatis & Androulakis (1999), that the solution space consists of vast areas of flat regions joined with narrow steep ones. Intuitively this would make sense, since the unit functions used in our RNNs are sigmoid functions which have characteristics close to a step function. Therefore, by combining so many neurons with sigmoid transfer functions, there are likely to be regions where the steps effectively overlap (recalling that the particles vary in all dimensions in each iteration), causing sharp steepness.

We will adopt the terminology *highly convex* for such functions[2] in contrast, for example, to concave functions typified by De Jong's "bowl" function F1 (De Jong 1975) commonly used to compare genetic algorithms.

A steep region around a solution minimum suggests that

[2]One can imagine a marble rolling over a (relatively flat) convex hull before descending steeply into a minimum.

Table 5: This table summarises the results from 50 training runs using each type of backpropagation method to train an RNN on the string *diibaguuubadiidiigu*.

Training Method	Successes
BPM	0
BPALR	50
BPMALR	50

the particles may be "flying" directly over these positions. This area, although the most important, may get relatively little exploration by the swarm. If the valleys are very steep, the probability of a particle stumbling into it in the high dimensional solution space may be very low. (It should be borne in mind that the population size is small compared to the number of dimensions. Also, since the fitness function combines fitness over all dimensions, a particle has no way of "knowing" whether its performance is improving in a certain dimension.)

5 The Importance of Adaptive Learning Rates

This conjecture about the difficulty of descending the error function caused us to look more systematically at the apparent success of backpropagation methods. In doing so we found that different variants of the backpropagation algorithm had markedly different success rates. As the standard backpropagation algorithm had poor performance, we experimented with three enhanced algorithms: Backpropagation with Momentum (BPM), Backpropagation with Adaptive Learning Rate (BPALR), and Backpropagation with Momentum and Adaptive Learning Rate (BPMALR).

We used each of these training methods to train the RNN on the regular language string *diibaguuubadiidiigu*. Training with each method took place 50 times, each time recording its success or failure as shown in Table 5

Contrary to our initial predictions, BPM (when not combined with adaptive learning) performed extremely poorly, failing at every training run. In fact, this experiment produced a surprising result — that the adaptive learning rate apparently is the sole critical factor contributing to a successful training of RNNs on regular language strings. This is consistent with Magoulas et al. (1999)'s findings which showed that the effectiveness of the BP algorithm can be improved via the use of an adaptive learning rate when training applications with a large number of variables.

This result also lent support to the conjecture that the error space around a solution minimum is a very steep region. At a region of steepness, the learning rate can be adjusted so that the new position does not overshoot the solution minimum. If indeed this conjecture is correct, it should be possible to similarly improve the PSO by introducing the concept of an adaptive learning rate into the algorithm.

6 Improving the Particle Swarm Optimiser

Incorporating the notion of learning rate into the PSO algorithm requires some refinement of its original structure. From the perspective of physics, equations (1) and (2) can be regarded as a discretised version of Newton's laws of motion, with a unit time step, and accelerating forces proportional to the distances of the particle from its own and the group's best positions.

Rewriting with an explicit representation of the acceleration, a_{id}^t, and time step, *dt*, we have:

$$\begin{aligned} a_{id}^t &= c_1 * rand() * (p_{id}^t - x_{id}^t) \\ &\quad + c_2 * Rand() * (p_{gd}^t - x_{id}^t) && (3) \\ v_{id}^{t+dt} &= v_{id}^t + a_{id}^t * dt && (4) \\ x_{id}^{t+dt} &= x_{id}^t + v_{id}^t * dt + \frac{a_{id}^t * dt^2}{2} && (5) \end{aligned}$$

Thus the acceleration at each time step is a function of the particle's present position, its own best position, and the global best position achieved so far. The velocity and projected position equations are essentially kinematic equations of constant acceleration.

The motivation for this representation of the PSO is that the introduction of an explicit time step, *dt*, is analogous to a learning rate as it controls how much change is to occur with each iteration.

If this variation were implemented without velocity capping, the swarm would eventually lose cohesion in much the same way as the original PSO. However, there is a more elegant method of keeping cohesion than by the use of velocity capping. In keeping with the philosophy of viewing the PSO from a physical point of view, we can introduce the idea of friction into the swarm world. This can act as a dampening factor to help keep the swarm in cohesion. Since the particles in the swarm essentially move with spring like behaviour, we introduce a spring retarding force which is proportional to its velocity, but acts in the opposite direction (Serway 1990). That is, $F \propto v$, and since $F \propto a$ by Newton's Law, we can conclude that $a = k * v$, where k is a negative constant coefficient of friction (dampening factor). Taking into account all modifications, our final version of the PSO is represented by equations 6, 7, and 8:

$$\begin{aligned} a_{id}^t &= c_1 * rand() * (p_{id}^t - x_{id}^t) \\ &\quad + c_2 * Rand() * (p_{gd}^t - x_{id}^t) + k * v_{id}^t && (6) \\ v_{id}^{t+dt} &= v_{id}^t + a_{id}^t * dt && (7) \\ x_{id}^{t+dt} &= x_{id}^t + v_{id}^t * dt + \frac{a_{id}^t * dt^2}{2} && (8) \end{aligned}$$

We can now introduce adaptive "learning" rates by modifying *dt*. The approach we take follows that of Magoulas et al. (1999). Each particle in the population keeps its own learning rate (or time-step), which is adjusted according to the algorithm presented in Figure 4.

In this algorithm, like backpropagation with adaptive learning rate, at each step a new position and its error are calculated. The new position will only be accepted if the error of the new position is at most *max_err_inc* times the error of the old position. If the new position is rejected, the step size is decremented by a factor of *step_dec*. Note that we also decrement the velocity by the same factor un-

```
Initialise population
Determine best particle in population
While (exit criteria not met) {
  For each particle x {
    Calculate acceleration at t
    Calculate velocity at t+dt
    Calculate position at t+dt
    If (error(position at t+dt)
        < error(position)) {
      position = position at t+dt
      velocity = velocity at t+dt
      step(x) = step_inc * step(x)
    } Else {
      If (error(position at t+dt)/
          error(position)
          < max_err_inc) {
        position = position at t+dt
        velocity = velocity at t+dt
      } Else {
        step(x) = step_dec * step(x)
        velocity = step_dec * velocity
      }
    }
  }
}
```

Figure 4: Algorithm for PSO with adaptive time step.

Table 6: Summary of the results from 4 training runs using the APSO on the string *diib*.

Run	Result	Best Error	Iterations
1	Success	9.722×10^{-4}	338
2	Success	9.975×10^{-4}	167
3	Failure	4.167×10^{-2}	175
4	Success	9.900×10^{-4}	81

der these conditions to prevent the old velocity value dominating the contribution to the new velocity value. For the remainder of this paper, we will refer to this algorithm as the Adaptive Particle Swarm Optimiser (APSO).

7 Case Study II: Training RNNs with the APSO

The regular language problem as described in Section 3.3 is used again here to compare the effectiveness of the adaptive PSO with the PSO. The experiments were run with the same parameterisation and fitness function, described in Section 4.

The results are shown in Tables 6, 7, and 8.

7.1 Discussion

While this is a small-scale empirical study, the results are promising, showing a substantial improvement over the conventional PSO. The APSO succeeded in training on the string *diibaguuubadiidiiguu* in 50% of cases, while the PSO failed every time. The string *diibaguuub* was successfully trained on all occasions while the PSO only managed one success of its four training runs. Although the APSO showed significant improvement over the PSO on the complex strings, it actually performed worse on the simple string *diib*, failing once while the PSO did not have any failures. The reason for this is not known, and requires further investigation.

Table 7: Summary of the results from 4 training runs using the APSO on the string *diibaguuub*.

Run	Result	Best Error	Iterations
1	Success	9.963×10^{-4}	1092
2	Success	9.952×10^{-4}	1089
3	Success	9.407×10^{-4}	1287
4	Success	9.890×10^{-4}	872

Table 8: Summary of results from 4 training runs using the APSO on the string *diibaguuubadiidiiguu*.

Run	Result	Best Error	Iterations
1	Failure	1.213×10^{-3}	3544
2	Success	9.987×10^{-4}	3630
3	Success	9.909×10^{-4}	2733
4	Failure	6.089×10^{-2}	5000

As we can see, training with the APSO presents a higher success rate than the PSO, especially on more complex strings. It can be seen that the APSO tends to require more iterations to converge on solutions than the PSO, however it should be noted that the PSO tended to fail prematurely so the number of iterations is not directly comparable.

8 Conclusion

While the PSO may be suitable for many optimisation problems, we have shown that for the task of training RNNs to learn regular language strings the standard PSO performs inadequately. We have conjectured that the algorithm is not well suited to high-dimensional highly convex solution spaces. Since success in these spaces has been achieved previously using backpropagation methods, we examined these approaches to see what features made them successful. The results indicated that an adaptive learning rate was the critical factor.

We have presented a new version of the particle swarm algorithm, adaptive PSO, that contains an analogous mechanism to an adaptive learning rate. In this algorithm the duration of the time step (and hence distance) is varied according to performance. This algorithm shows substantially improved performance when training RNNs to learn more complex regular language strings.

As we believe the major contributing factor to the improved performance of the APSO is the reduced step size of each particle when it is around a minimum, it would be interesting to compare its performance against the PSO with constriction factor (Clerc & Kennedy 2002).

Bibliography

Clerc, M. & Kennedy, J. 2002, 'The particle swarm—explosion, stability, and convergence in a multidimensional complex space', *IEEE Transactions on Evolutionary Computation* **6**(1).

De Jong 1975, Analysis of the behaviour of a class of genetic adaptive systems, PhD thesis, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor, MI.

El-Gallad, A. I., El-Hawary, M., Sallam, A. A. & Kalas, A. 2001, Swarm-intelligently trained neural network for power transformer protection, *in* 'Canadian Conference on Electrical and Computer Engineering, 2001', Vol. 1, pp. 265–269.

Elman, J. L. 1990, Finding structure in time, *in* 'Cognitive Science', Vol. 14, pp. 179–211.

Haykin, S. 1994, *Neural Networks*, Prentice Hall Inc.

Ismail, A. & Engelbrecht, A. 2000, Global optimization algorithms for training product unit neural networks, *in* 'International Joint Conference on Neural Networks IJCNN'2000', Vol. 1, IEEE Computer Society, Los Alamitos, CA, pp. 132–137.

Kennedy, J. & Eberhard, R. 2001, *Swarm Intellegence*, Morgan Kaufmann Publishers.

Kennedy, J. & Eberhart, R. 1995, Particle swarm optimization, *in* '1995 Proceedings, IEEE International Conference on Neural Networks', Vol. 4, IEEE, pp. 1942–1948.

Magoulas, G., Vrahatis, M. & Androulakis, G. 1999, 'Improving the convergence of the backpropagation algorithm using learning rate adaptation methods', *Neural Computation* **11**, 1769–1796.

Mendes, R., Cortez, P., Rocha, M. & Neves, J. 2002, Particle swarm for feedforward neural network training, *in* 'Proceedings of the 2002 International Joint Conference on Neural Networks', Vol. 2, pp. 1895–1899.

Rumelhart, D. E., Williams, R. J. & Hinton, G. E. 1986, 'Learning internal representations by error propagation', *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* **1**, 318–362.

Serway, R. A. 1990, *PHYSICS For Scientists & Engineers with Modern Physics Third Edition*, Saunders College Publishing, chapter 13, p. 341.

van den Bergh, F. & Engelbrecht, A. 2000, 'Cooperative learning in neural networks using particle swarm optimizers', *South African Computer Journal* **26**, 84–90.