Machine Learning - Project Assignment
Training ANNs using distributed PSO

Martynas Pumputis, Gytis Bacevičius, Uwe Bauknecht
Sourcecode: http://user.it.uu.se/~mapu0695/ml/

June 2, 2009

1 Abstract

Particle Swarm Optimization (PSO) algorithms have been successfully used to train Artificial Neural Networks (ANN) for various purposes in the past. The PSO is used to find an optimal configuration by presenting it with a high-dimensional space the dimensions of which correspond to the parameters of the ANN.

One of the major performance factors with this approach is the fitness-function which assesses the quality of the present position a particle of the swarm is located in. This paper describes an efficient concurrent implementation of a PSO with a feedforward ANN in Erlang [1] which aims at improving on this problem by means of distributed execution. We will further present simluations and compare the results to related work.

Contents

2 Introduction

2.1 Particle Swarm Optimization

2.1.1 Basic Principle

Particle **S**warm **O**ptimization (PSO) is an algorithm which was developed to mimic the bahvior of bird flocks which are able to maintain precise formations while flying in groups. PSO uses a number of particles where each of which corresponds to a possible solution to the optimization problem. These particles then "fly" through a hyperdimensional search space towards an optimum using a combination of knowledge about the optimality of their own position so far (cognitive component) and that of its neighboring particles (social component).
The position $\vec{x}_i \in \mathbb{R}^j$of a given particle i at a certain point in time t is modeled as

$$\vec{x}_i(t+1) = \vec{x}_i(t) + \vec{v}_i(t+1) \tag{1}$$

with $\vec{v}$ being a velocity towards the presently known optima. The velocity in dimension j is defined as

$$v_{ij}(t+1) = v_{ij}(t) + c_1 r_{1j}(t)[y_{ij}(t) - x_{ij}(t)] + c_2 r_{2j}(t)[\hat{y}_j(t) - x_{ij}(t)] \tag{2}$$

Here $c_1, c_2 \in \mathbb{R}_0^+$ are positive constants which regulate the influence of the cognitive and social components on acceleration. The variables $r_{1j}(t)$ and $r_{2j}(t)$ are random values between 0 and 1 which add an element of nondeterminism.
The best position the individual particle has had so far is given as $y_{ij}(t)$ and the best position of the social component is $\hat{y}_j$. The worth of a certain position in search space is determined using a fitness-function $f : \mathbb{R}^{n_x} \to \mathbb{R}$ which has to be defined according to the problem at hand.

2.1.2 Variations

There are several variations of the basic algorithm which usually relate to how the social component is determined. Eg. in the *gbest* variant we would take the best $y_{ij}(t)$ among all particles, while in the *lbest* variant we use a more local approach, by selecting the $y_{ij}(t)$ among the particles of a certain neighborhood, defined by the distance of the indices. This tends to exploit local knowledge better, while still allows all particles to find a global optimum, as every particle is a member of several neighborhoods.

The structure of these neighborhoods is very important, as high connectivity and clustering will increase convergence speed, but at the same time

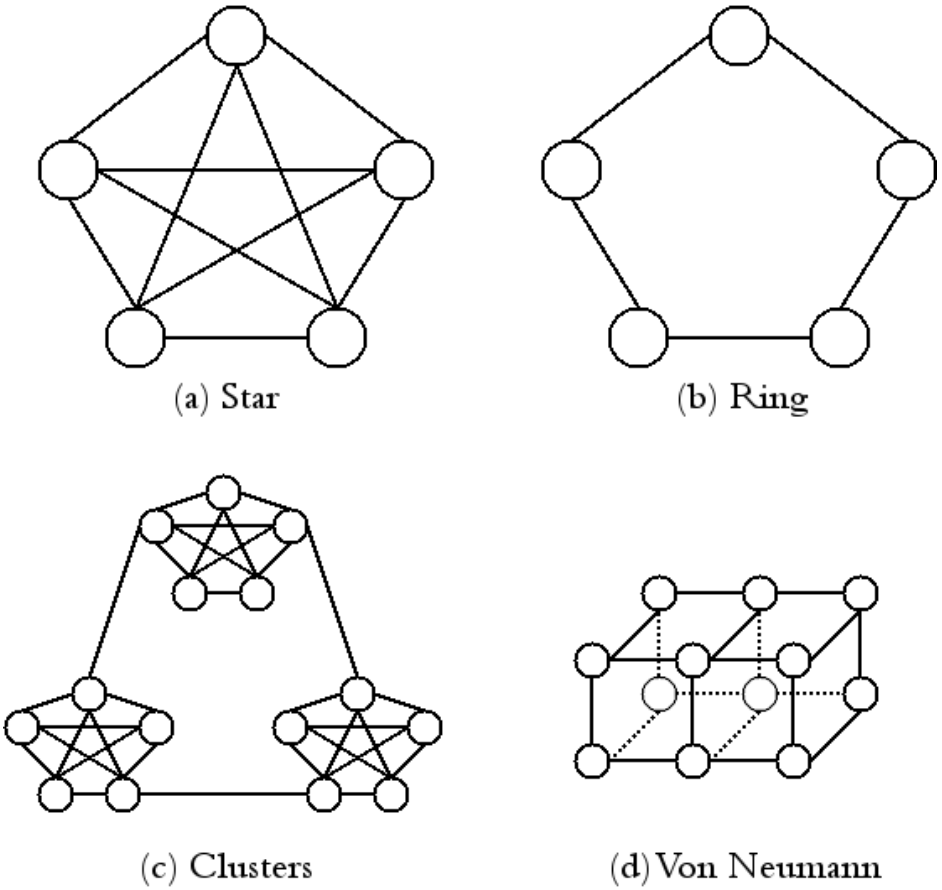


Figure 1: Example topologies for PSOs.

increase the risk of becoming trapped in a local minimum. Nets with low connectivity and clustering on the other hand might cover the search space better and therefore have a better chance of finding the global minimum. Between *lbest*, which topologically forms a ring, and *gbest*, which forms a fully connected star structure, there are verious hybrid forms to balance convergence speed against susceptibility to local minima. See figure 1 for examples.

PSO can use several termination criteria: We might consider a maximum number of iterations, give a bound for an acceptable distance of the particles to the optimum, watch whether we still improve after a given number of iterations, the swarm's radius or the number of particles located in a cluster around the optimum.

2.1.3 Extensions

Some modifications to this algorithm have been suggested in order to improve convergence properties: One recurrent problem is that particles far from any optimal position reach very high velocities and tend to leave the search space. In order to prevent this, one can set an upper bound on the velocity (so-called *velocity clamping*) for every dimension seperately according to the maximum spatial extension of relevant solutions. However, this maximal velocity has to be selected carefully to prevent overshooting (too high) and getting stuck in local minima (too low).

There is another problem that is inherent with this proposal: Setting velocity limits individually for each dimension means that the "flight trajectory" is not straight anymore and if we reach the maximum velocities of all dimensions simultaneously, then we are basically running along the sides of a hypercube and might therefore not find any minima within [6]. The suggested solution for this problem is to introduce a decay-factor which reduces the velocity again, either constantly after reaching the maximum or dynamically, whenever the algorithm runs through several iterations without any change in the global optimum.

Another approach to mitigate the problem of reaching the maximum velocities is to introduce a so-called *inertia weight*. It is used in equation 2 and multiplied to $v_{ij}(t)$ yielding

$$v_{ij}(t+1) = wv_{ij}(t) + c_1 r_{1j}(t)[y_{ij}(t) - x_{ij}(t)] + c_2 r_{2j}(t)[\hat{y}_j(t) - x_{ij}(t)] \quad (3)$$

However, this parameter has to be chosen very carefully, as it might lead to cyclic or divergent behavior, when not maintining the following relation [5]

$$w > \frac{1}{2}(c_1 + c_2) - 1 \quad (4)$$

There are various ways of dealing with this. In short we can select w randomly within certain bounds, use linear or nonlinear functions that decrease w with time and so on[1].

Another suggestion to overcome this problem is the introduction of a so-called *constriction coefficient* χ, which changes equation 3 the following way:

$$v_{ij}(t+1) = \chi(v_{ij}(t) + c_1 r_{1j}(t)[y_{ij}(t) - x_{ij}(t)] + c_2 r_{2j}(t)[\hat{y}_j(t) - x_{ij}(t)]) \quad (5)$$

This coefficient χ is chosen as follows:

$$\chi = \frac{2\kappa}{|2 - (c_1 r_1 + c_2 r_2) - \sqrt{(c_1 r_1 + c_2 r_2) \cdot (c_1 r_1 + c_2 r_2 - 4)}|} \quad (6)$$

[1]For more details see [6].

With $(c_1 r_1 + c_2 r_2) \geq 4$ and $\kappa \in [0, 1]$. This approach was developed by Clerc using eigenvalue analysis of swarm dynamics in order to have a solution which converges to a stable point without using velocity clamping. In fact, if the assumptions above hold, convergence is guaranteed [4]. For small values of κ we will have slow convergence, but good exploitation. For large values we have the opposite case.

2.2 Training of Artificial Neural Networks using PSO

PSOs can be used to train **A**rtificial **N**eural **N**etworks (ANN). One advantage of this method in comparison to regular training methods like backpropagation is, that we can choose a non-differentiable error function [3]. If we use the ANN as a classifier, instead of feeding back the difference between desired and actual value of the net, we can choose a measure of misclassifications of the whole network as error function.

When using a PSO to train an ANN we would set up the PSO such, that every dimension of the search space of the PSO corresponds to one weight-parameter of the net, including the bias-weights. Therefore every particle represents one possible ANN the fitness of which we will assess using the sum of squared error of misclassifications or, in the case of an approximation task, the error function of the net. This entails a high computational effort, as we have to evaluate every network in every iteration, but the increased convergence speed can still make it significantly more efficient, than backpropagation [7].

3 Implementation

3.1 General Setup

We implemented the PSO and ANN using Erlang [1], which is a functional programming language especially designed for concurrent programming. The PSO is split into 3 parts: A server, one or more slaves and the particles themselves.

The server acts as a central coordinator to which all the information required for the execution of the given task can be passed via the interface. The server will then proceed to order the slave nodes to setup and execute the PSO algorithm according to these parameters and will also stop the process when a given precision has been reached. The slave nodes serve as distributed controllers for the individual particles and will be executed on different nodes of the distributed environment, like individual computers in

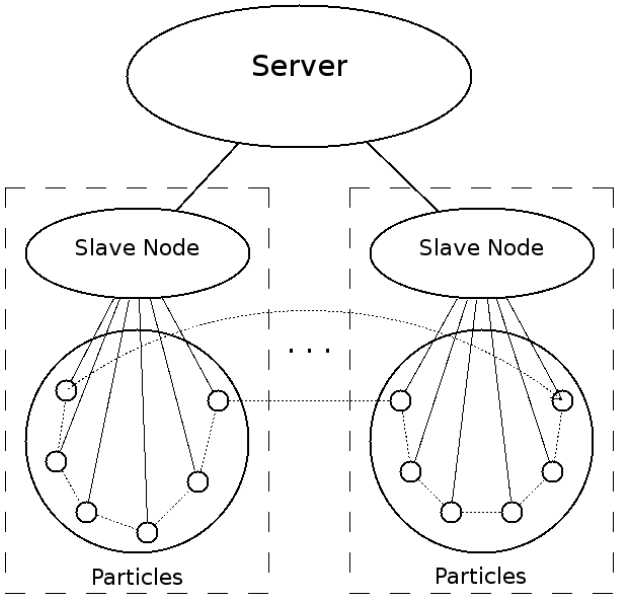


Figure 2: Layout of the PSO implementation.

a cluster. The particles finally contain all the algorithms regarding the PSO scheme itself. Every particle can directly communicate with its individual neighbors to exchange position information. See figure 2 for an illustration of this layout.

3.2 PSO in detail

Our implementation is highly flexible, not only in terms of concurrent execution, but also regarding variations of the PSO's structure. Arbitrary topologies can be defined by setting the number of particles and their individual connectivity. We also make use of a wide variety of the extensions proposed in section 2.1.3: We implemented range limits, velocity clamping, the constriction coefficient and for experimental reasons the "selfless model" as proposed by Kennedy in [9], where the neighborhood's best value is determined excluding the updating particle itself in order to enhance social interaction. All these modes can be activated and their respective parameters set via the interface, as described in section A.

Another feature we considered worthwhile implementing, yet could not find the time, is based on a suggestion by Carlisle and Dozier to reduce the velocity of particles impacting given $x_{j,max}$ and $x_{j,min}$ values to zero [2]. The idea behind this is that otherwise particles hitting the faces of the hypercube spanned by those limits will exhibit a tendency to be stuck to them for an

extended period of time and will therefore not contribute to finding a solution until their velocity vector nondeterministically decreases.

After supplying the necessary parameters to the server and launching the slave nodes, the initialization process commences. The slave nodes will register with the server (figure 3 (a)) and the server will distribute the number of requested particles evenly among the available slaves (figure 3 (b)). In the next step the particles relay their IDs and locations to the server via the slave nodes (figure 3 (c)). The Server will use this information to determine which particle is neighbor to which and will send this information directly to the particles (figure 3 (d)). Finally the particles will notify the server once they are ready and the server will start the execution of the algorithm, as soon as all particles are registered as ready (figure 3 (e) and (f)).

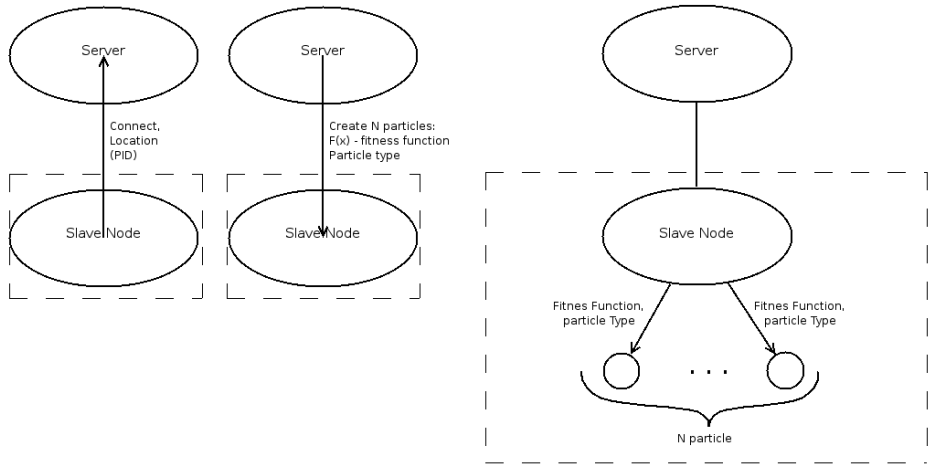
Most of the computation regarding the PSO itself is carried out by the particles. They will calculate their local optima and communicate them directly to their assigned neighbors, wait for their answer, calculate the fitness values and update their own position according to the algorithm, which is then in turn sent to its neighbors and the slave node, which will further relay this information back to the server in order to measure convergence. See figure 4.

For the choice of parameters we use a combination of values which have been known to work well in past cases and our own empirical results. Initially a number of 20 to 30 particles is considered to be sufficient to achieve a good tradeoff between the quality of approximation and computational complexity and likewise a global neighborhood with $c_1 = 2.8$ and $c_1 = 1.3$ or $c_1 = c_2 = 4.1$ when using the constriction coefficient generally offers good results [2].

3.3 ANN in detail

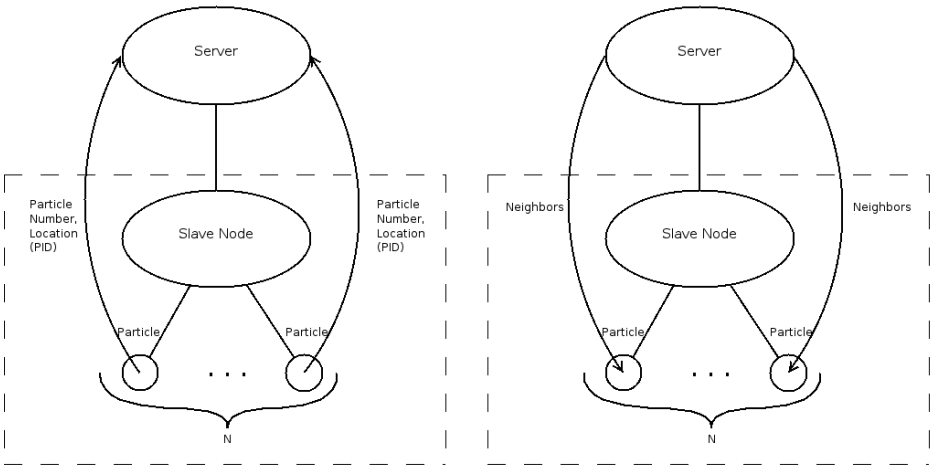
Our ANN is a simple feedforward neural network. It is implemented using nested lists to represent layers and weight vectors. Our set of provided activation functions include the standard linear, step and logistic functions, but is readily extensible to arbitrary functions. However our logistic function is slightly modified to overcome the problem that Erlang does not have big floating point numbers. We found empirically that the maximum value Erlang can compute for e^x is for $x = 709$ and therefore we cap the sigmoid-function and approximate with 2^x.

We can use any number of layers and nodes. Activation functions can be set either for each perceptron individually, per layer or for the whole net.



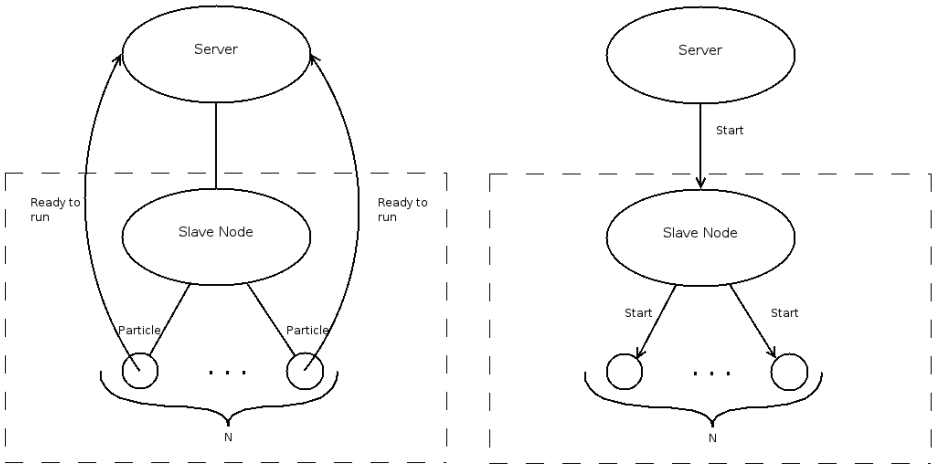
(a) Slaves are initialized

(b) Slaves create particles



(c) Particle information is relayed to server

(d) Server assigns neighborhoods to particles



(e) Particles notify server, once they are ready

(f) If all particles are ready, server starts algorithm

Figure 3: Initialization of the PSO.

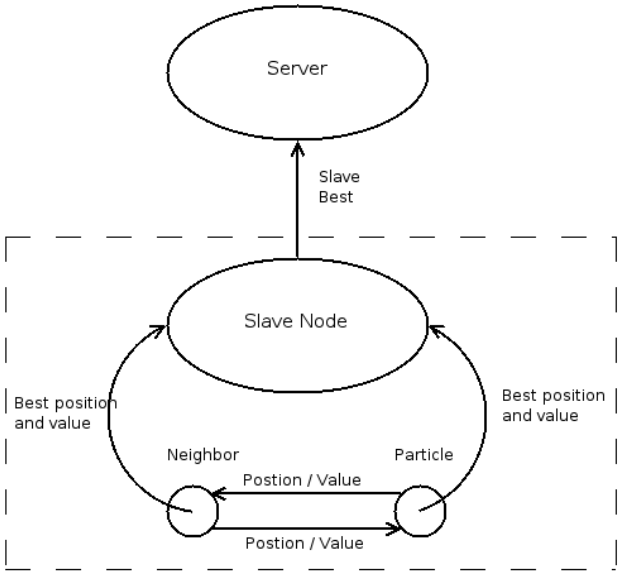


Figure 4: Information flow of best values.

4 Simulation and Results

4.1 Test cases

Our tests are performed on the XOR function problem, as it is easier to handle than highly complex functions.

We began testing on a wide range of values for c_1 and c_2. Theory tells us that selecting the social component to be the more decisive will improve performance in highly multimodal tasks [9], such as ANN-training.

In our initial test we used a ring topology and velocity clamping at $v_{max} = 8$ on all dimensions. As convergence criteria we used both, an error measure and a maximum of iterations. Once the error drops below 0.02, we consider the PSO to have successfully converged. If, however, we do not reach this value after 4000 iterations, we consider the PSO to be stuck in a local optimum. The table below contains the mean number of iterations out of 20 runs until the error criterion was met.

c_1 \ c_2	0	1	5	10
0	-	179.58	29.18	35
1	-	224.41	32.93	37.55
5	-	82.2	149.28	52.86
10	-	75.38	57.18	95.41

The following table shows the percentages of particles that successfully

converged.

c_1 \ c_2	0	1	5	10
0	-	95%	90%	75%
1	0%	85%	80%	90%
5	0%	95%	80%	75%
10	0%	95%	95%	85%

Indeed, higher values for social components tend to increase the convergence speed significantly. From these tables we take the 3 fastest converging values and try to analyize the effects of velocity clamping. The results can be seen in the next tables.

v_{max} \ $c_1;c_2$	0;5	10;10	10;5
0.5	132	94	-
1	69.4	46.5	225.66
4	172.16	49.26	134.14
8	29.18	95.41	57.18
10	45.68	61.35	84.05

The following table shows the percentages of particles that successfully converged.

v_{max} \ $c_1;c_2$	0;5	10;10	10;5
0.5	5%	5%	0%
1	25%	10%	85%
4	65%	65%	65%
8	90%	85%	95%
10	80%	75%	95%

We can see that if we lower the velocity limit, our chances of converging decrease significantly. We seem to achieve lightning fast convergence in some cases of low speeds, but this is merely coincidental if the swarm happened to be initialized close to the optimum.
Next, we analyzed the effects of the selfless model based on the parameters we found so far, leaving out the velocity limits which did rarely converge.

v_{max} \ $c_1;c_2$	0;5	10;10	10;5
4	44.64	169.15	151.53
8	91.05	84.63	76.87
10	96.84	121.15	87.26

Again, the following table shows the percentages of particles that successfully converged.

v_{max} \ $c_1;c_2$	0;5	10;10	10;5
4	85%	65%	75%
8	90%	95%	98%
10	95%	95%	75%

Interestingly enough we find that using the selfless model increases the chances of converging, but has an adverse effect on convergence speed. The reduction in convergence speed is consistent with other research [9], but the influence on the probability of convergence hasn't been considered. We maintain the focus on convergence speed and therefore discard the selfless model at this point.

For the next test series we changed the topology of the nodes from the ring structure to the fully interconnected star structure.

v_{max} \ $c_1;c_2$	0;5	10;10	10;5
4	35.5	28.17	26.66
8	278.33	20.8	189.59
10	45.5	156.83	103.6

Yet again, the following table shows the percentages of particles that successfully converged.

v_{max} \ $c_1;c_2$	0;5	10;10	10;5
4	20%	17.5%	10%
8	50%	25%	30%
10	40%	70%	50%

The fully interconnected model turns out to be extremely susceptible to local minima and therefore unsuitable for our purposes of training an ANN. When looking at the tables we seem to have some very good entries, but again, these are just purely accidential when the particles just happened to be initialized next to a good optimum[2].

For the next set of testcases we will use the Von Neumann structure.

v_{max} \ $c_1;c_2$	0;5	10;10	10;5
4	12.57	46.55	35.16
8	229.58	73.93	188.55
10	25.61	162.57	50.18

Finally, the following table shows the percentages of particles that successfully converged.

[2]There are actually PSO modifications which use this effect as a principle by discontinuously moving particles to Gauss distributions around good values [10].

v_{max} \ $c_1;c_2$	0;5	10;10	10;5
4	35%	55%	30%
8	60%	25%	65%
10	90%	70%	85%

The conclusion here is that Von Neumann is not the best choice for this problem, as it is also quite susceptible to local minima, albeit by far not as much, as the fully connected structure is. Von Neumann's topology is probably good in other cases, but here a ring structure with velocity clamping seems to be the most efficient choice.

4.2 Performance vs. SNNS

We measured the performance of our implementation against the results obtained by training an identical neural network using SNNS, a simulator for neural networks developed by the university of Stuttgart [13]. Using standard backprop SNNS achieves an error of less than 0.02 after about 950 iterations. This is more than 10 times slower as compared to the solution found by even the basic PSO.

Using backprop with momentum, we can cut the number of iterations down to about 450, which is still well above the values achieved with the PSO. RProp finally is able to match the PSO's speed in convergence, but the chances for convergence to occur are lower.

5 Conclusion

In this paper we have explained the theory behind using PSOs to train ANNs and presented a flexible, distributed implementation in Erlang, which achieves outstanding results. Convergence is vastly superior to the classical backprop algorithm.

Although our results seem quite promising, there still remains a lot of work to be done. The main advantage of the concurrent implementation is of course the possibility of running it in an actual distributed environment such as a cluster, which we did not have access to and therefore the real performance of our implementation can hardly be assessed. The software itself can also be vastly improved. Especially a graphic user interface offering more configuration options, like range limitations would significantly boost testing. Another addition that might be worthwhile would be an interface to SNNS in order to combine the versatility of both the PSO as a training tool and SNNS' variety of networks and tools.

In addition to that there should also be more and more diverse test cases to determine performance with different optimization tasks. Self-organizing feature maps and especially product unit networks are viable objectives for PSO-training [6][8]. The high number of parameters to set up a PSO makes it especially difficult to find the optimal setup for a given problem and further research should try and investigate good parameter sets which can be used as rules of thumb for specific optimization tasks.

The generally good performance should not hide the fact that there are more advanced and complex algorithms which seem to offer better results. Extensions of the standard backpropagation algorithm like *RProp* or *Quick-Prop* can achieve convergence faster on some specific problems [11]. On the other hand development on the PSO algorithm has yielded novel approaches further improving performance. For example models introducing additional stochasticity like SPSO [3] or quantum mechanics-based QPSOs [12]. There are also promising approaches of how to further improve ANNs by including the ANN's structure itself as part of an optimization task, as suggested in [15] and [14].

A Interface

The interface to use both, the ANN and the PSO, is rather rudimental as timing contraints did not allow to further delve into the intricacies of GUI programming. All parameters are entered directly into 3 configuration files.

A.1 ANN Parameters

The parameters of the ANN are specified in the file network.cfg.

- Architecture we can specify the number of layers and nodes to be used. The input takes the form of list of nodes per layer. The first item in the list are the number of nodes in the input layer and the last element is the number of nodes in the output layer, therefore the list needs to have at least two elements. All other elements inbetween will become hidden layers with the indicated number of nodes.

- Activation function can presently be one of three values. This selected function will be applied to all nodes of the network. One can either select the logistic function and specify the lambda-value to adjust the slope, a regular linear or the step function.

- Under inputs we have to specify the data set we would like to train the network on. The data has to be presented in the form of a list of tuples which correspond to input/output pairs. The first element of the tupel is a list of values which should hold the input vectot, the second element a list for the output vector.

- In the weights-section we can specify bounds for the values with which the weights of the links will be initialized.

A.2 PSO Parameters

In the file params.cfg we can set the values to configure the PSO.

- In the first section we can specify the c_1 and c_2-values for both the cognitive (local) and the social (global) component respectively.

- Under SpaceClamp we can set a limit l, such that $\forall j : x_{j,min} = -x_{j,max} = -l$. The first parameter is either true or false and the second one gives a value for l.

- In the section SpeedClamp we can activate velocity clamping. Again by setting the first value to true and choosing a value k, such that $\forall i, j : v_{ij,max} = k$.
- The next entry manages the usage of the constriction coefficient, which can either be activated or deactivated by setting the value to true or false.
- The selfless model can be activated by entering "selfless" as value or deactivated by inputting "selfish".
- The section called Tmax is one of two convergence criteria that can be set in the config files. The server will stop the execution of the algorithm as soon as the number of iterations specified here is reached. If we do not want to use this criterion, we can put the value "nondefined" here.
- The second convergence criterion is a minimal error we would like to achieve. The real value entered here is the desired maximum sum of squared error the net will produce. Entering "nondefined" deactivates this limit.

A.3 PSO Topology

Finally the file topology.cfg is used to specify the topology of the PSO.

- The number entered under NodesNum indicates the number of slaves nodes which will connect to the server.
- Under Topology we have to specify the topology of the network as a list of tuples which contain the nodes' numbers as first element and a list of their neighbors as a second argument. It is imperative that node numbers increase monotonicly from 1 to the maximum number without any missing numbers inbetween.

References

[1] Joe Armstrong. *Programming Erlang, Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[2] A. Carlisle and G. Dozier. An off-the-shelf pso. In *In Proceedings of the Particle Swarm Optimization Workshop*, pages 1–6, 2001.

[3] Xin Chen and Yangmin Li. Neural network training using stochastic pso. In *ICONIP (2)*, pages 1051–1060, 2006.

[4] M. Clerc and J. Kennedy. The particle swarm - explosion, stability, and convergence in a multidimensional complex space. *Evolutionary Computation, IEEE Transactions on*, 6(1):58–73, 2002.

[5] F. Van den Bergh and Andries Petrus Engelbrecht. A study of particle swarm optimization particle trajectories. *Inf. Sci.*, 176(8):937–971, 2006.

[6] Andries Engelbrecht. *Computational Intelligence: An Introduction, Second Edition*. John Wiley & Sons Ltd., The Atrium, Southern Gate, Chichester, West Sussex, England, 2007.

[7] V. G. Gudise and G. K. Venayagamoorthy. Comparison of particle swarm optimization and backpropagation as training algorithms for neural networks. In *Swarm Intelligence Symposium, 2003. SIS '03. Proceedings of the 2003 IEEE*, pages 110–117, 2003.

[8] A. Ismail and A.P. Engelbrecht. Global optimization algorithms for training product unit neural networks. volume 1, pages 132–137 vol.1, 2000.

[9] J. Kennedy. The particle swarm: social adaptation of knowledge. pages 303–308, Apr 1997.

[10] J. Kennedy. Bare bones particle swarms. pages 80–87, April 2003.

[11] R. Mendes, P. Cortez, M. Rocha, and J. Neves. Particle swarms for feedforward neural network training. In *Neural Networks, 2002. IJCNN '02. Proceedings of the 2002 International Joint Conference on*, volume 2, pages 1895–1899, 2002.

[12] Millie Pant, Radha Thangaraj, and Ajith Abraham. A new quantum behaved particle swarm optimization. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 87–94, New York, NY, USA, 2008. ACM.

[13] Ed Petron. Stuttgart neural network simulator: Exploring connectionism and machine learning with snns. *Linux J.*, July:2, 1999.

[14] Jianbo Yu, Shijin Wang, and Lifeng Xi. Evolving artificial neural networks using an improved pso and dpso. *Neurocomputing*, In Press, Corrected Proof, 2008.

[15] John Paul T. Yusiong and Prospero C. Naval Jr. Training neural networks using multiobjective particle swarm optimization. In Licheng Jiao, Lipo Wang, Xinbo Gao, Jing Liu, and Feng Wu, editors, *ICNC (1)*, volume 4221 of *Lecture Notes in Computer Science*, pages 879–888. Springer, 2006.