# The Adaptive Critic Learning Agent (ACLA) Algorithm: Towards Problem Independent Neural Network based Optimizers

Udhay Ravishankar
Electrical and Computer Engineering Dept.
University of Idaho
Idaho Falls, USA
ravi4736@vandals.uidaho.edu

Milos Manic
Computer Science Dept.
University of Idaho
Idaho Falls, USA
misko@ieee.org

***Abstract*—This paper presents the development of a new neural network based optimizer called the Adaptive Critic Learning Agent (ACLA) algorithm. The ACLA algorithm is based on the traditional Adaptive Critic Design (ACD) algorithm and hence its name. Conventional neural network based optimizers use the principle of Hopfield/Tank Neural Networks (HTNN) to solve unimodal optimization problems. These neural networks require tailored structures for the specific optimization problem. The ACLA algorithm presented in this paper uses a general randomly initialized neural network to solve any unimodal optimization problem. This is achieved by extending the principles of the traditional ACD algorithm for the ACLA algorithm. Other attributes of the ACLA algorithm are related to the issues with swarm based optimizers such as Particle Swarm Optimization (PSO) and Genetic Algorithms (GA). These issues are: 1) large memory requirements and 2) multiple parameters required to tune the algorithm's convergence performance. The ACLA algorithm resolves these issues by: 1) using only one neuron to reduce memory requirements and 2) using only a single learning coefficient parameter to tune the algorithm's convergence performance. The ACLA algorithm was tested and compared with three swarm based optimizers on two unimodal benchmark problems typically used for PSO and GA algorithms. Test results proved the ACLA algorithm to converge to solutions 7 orders greater than the swarm based algorithms. The ACLA algorithm was further tested on two multimodal benchmark problems to demonstrate its capability to converge to nearest local minima.**

***Keywords-Adaptive Critic Design, Neural Networks, Swarm Intelligence.***

## I. INTRODUCTION

Optimization is an important aspect in many fields of science, engineering and business. Hence there is always a lot of room to develop new optimization techniques. Optimization algorithms in the field of computational intelligence are either swarm based or neural network based. Particle Swarm Optimization (PSO) and Genetic Algorithms (GA) are examples of swarm based optimizers. These algorithms typically require large memory to enact swarm behaviors for optimization. Hopfield/Tank Neural Networks (HTNN), on the other hand, are examples of neural network based optimizers. These optimizers typically require neural networks that are tailored for the specific optimization problem.

The PSO algorithm was introduced by Kennedy and Eberhart in 1995 [1] and GAs were initially suggested by Fraser in [2], Fraser and Burnell in [3], Crosby in [4] and popularized by Holland in [5]. Although many variants of PSO and GA have been developed over the recent years, e.g. in [6] – [10], only a couple of variants have sought to minimize the memory requirements of the PSO and GA, such as the Small Population PSO (SPPSO) algorithm in [11] and the Mean Variance Optimization (MVO) in [12].

The HTNN algorithm was introduced by Hopfield and Tank in 1985 [13]. This work inspired many improvements, later on, on the development of new neural network structures for optimization. Examples of such work can be found in [14] – [19].

Adaptive Critic Design (ACD) was introduced by Werbos in 1974 [20]. The algorithm was designed for optimal neural control problems and has successfully been implemented in work such as [21] – [25]. The algorithm is Error Backpropagation (EBP) based, but on a large network of neural networks. The EBP algorithm is a very simple optimization algorithm and its principles are extended towards the development of the Adaptive Critic Learning Agent (ACLA) algorithm. The ACLA algorithm comprises of a randomly initialized neural network that seeks to solve the optimization problem using the principles of EBP. Further, to ensure minimal memory requirements, the ACLA neural network uses only one neuron for finding the optimal solution. By using this setup, the ACLA algorithm did prove to find the optimal solution.

The rest of this paper proceeds as follows: Section II will cover a brief overview on the current neural network approaches to optimization, i.e. using the principle of the Hopfield/Tank Neural Network. Adaptive Critic Design is introduced in Section III followed by the development of the ACLA algorithm in Section IV. Section V.A introduces the unimodal benchmark function in the experimental setup made to test the ACLA algorithm. Section V.B shows the trajectory of the ACLA algorithm on 2D versions of the benchmark

problems followed by comparative results of the ACLA with the PSO, SPPSO and MVO, in Section V.C, on higher dimensional versions of the same benchmark problems. Section V.D introduces two multimodal benchmark problems to demonstrate the capability of the ACLA algorithm to find the nearest local minima. The paper then concludes with future work in section VI.

## II. Hopfield/Tank Neural Network Optimizer

In this section the current approaches to neural network based optimizers is discussed, i.e. using the principle of Hopfield/Tank Neural Networks.

Hopfield and Tank in [26] showed that a Recurrent Neural Network (RNN) was capable of solving linear programming problems. For a given set of equations which can be described by a matrix, $W$, the energy function associated with $W$ is given by the Lyapunov Energy Function given as,

$$E = -\frac{1}{2} y^T W y - t^T y$$
$$\forall \quad Wx = t \quad x \in \Re^n \tag{1}$$

Note that $y$ is the output of the function $Wx - t$, i.e. $y=Wx - t$, and that $y \rightarrow x^*$ (optimal $x$) as $E \rightarrow 0$. The energy gradient, $\nabla E$ , is given by,

$$\nabla E = -Wy - t^T \tag{2}$$

Eq. (2) is simply the negative of the function $y=Wx - t$, and therefore the recurrent function naturally commutates to the minimum of the energy function given in Eq. (1). The RNN used to accomplish this is shown in Fig. 1.

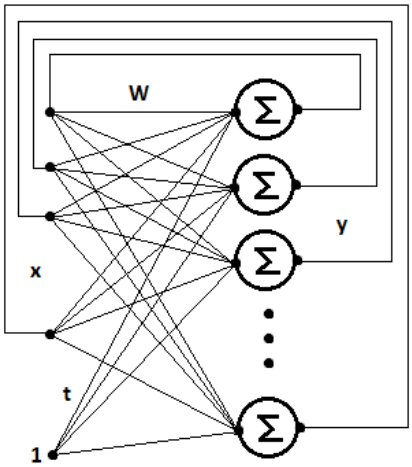


Fig. 1. The Hopfield/Tank Recurrent Neural Network used for linear programming.

## III. Adaptive Critic Design

This section introduces Adaptive Critic Design (ACD) from which the ACLA algorithm was developed.

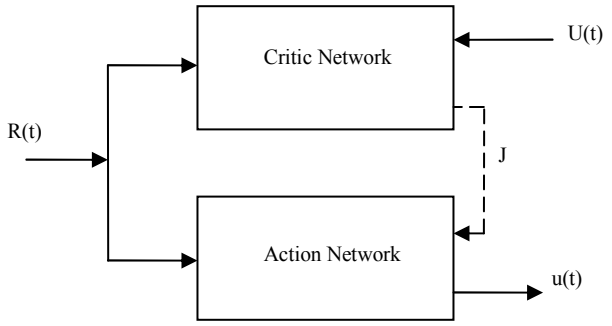


Fig. 2. The Classical Critic Architecture [28].

ACD, introduced by Werbos [20], belongs to the class of neural Markov Decision Processes (MDP) [27]. MDPs are typically used in computing the optimal control policy for industrial and commercial processes. While MDPs use state transition matrices to compute the optimal control policy, ACD uses a user defined utility function. The user defined utility function $U(t)$ is the optimization search space used through the Bellman Equation given in Eq. (3) [29].

$$J(t) = \sum_{k=0}^{\infty} \gamma^k U(t+k) \tag{3}$$

The $\gamma$ in Eq. (3) is the discount factor having a value between zero and one. The ACD algorithm classically comprises of two neural networks, namely the Critic Network and the Action Network shown in Fig. 2. In Fig. 2, the *R(t)* is a vector of system states at time $t$ and *u(t)* is the control output of the Action Network at time $t$. The Action Network is the desired optimal neural controller that will control a specified system once designed. The neural network setup shown in Fig. 2 is the basic setup that provides the foundation of the ACD principle. Typical ACD neural network setups include a Model Network that is pre-trained to model the system dynamics. This Model Network is then placed between the Critic Network and the Action Network to optimize the Action Network, i.e. the neural controller.

During the initialization stage, the Action Network is trained to model the conventional controller of the system under consideration. The critic values of a specific control sequence are evaluated using Eq. (3) in time-series form and then modeled by the Critic Network. Once the Critic Network is trained, the optimization of the Action Network begins by the backpropagation of a one at the output of the Critic Network for each time $t$. By the backpropagating a one at the output of the Critic Network, the gradient $\partial J/\partial A$ is obtained (Here, $J$ is the output of the Critic Network and $A$ is output of the Action Network assumed to be presented as an input to the Critic Network). To understand the meaning of the gradient $\partial J/\partial A$, one has to place the action inputs $A$ as the optimization parameters in the optimization search space and $J$ as the fitness value of input $A$. The gradient $\partial J/\partial A$ then becomes the gradient of the fitness function $J$ at location $A$. Once the gradient $\partial J/\partial A$ is calculated, it can be presented to the Action Network weight updation equation as the error in the Error Backpropagation

(EBP) equation. Replacing the error in the EBP equation with the gradient $\partial J/\partial A$ gives Eq. (4).

$$\Delta W_A = -\eta_A \frac{\partial J}{\partial A} \frac{\partial}{\partial W_A} \left( \frac{\partial J}{\partial A} \right) \tag{4}$$

In Eq. (4), $\eta_A$ is the learning coefficient of the Action Network.

## IV. The Adaptive Critic Learning Agent

This section discusses the development of the Adaptive Critic Learning Agent (ACLA) algorithm.

In the previous section, it was noted that the EBP equation can optimize any function as long as the function gradient is presented to the EBP equation. The ACLA algorithm extends this concept towards finding the optimal parameters in a given optimization search space. This, however, is not a straightforward procedure and it is mainly due to the EBP equation itself.

Suppose we consider a linear neural network with one input layer, one hidden layer of linear neurons and one output layer. Let the weights of the neural network between the input layer and hidden layer be denoted as $W_{ih}$ and weights between the hidden layer and the output layer be denoted as $W_{ho}$. Further, let the inputs to the neural network be denoted as a vector $X$, the intermediate outputs from the hidden layer neurons be denoted as a vector $D$ and the actual output of the neural network be denoted as a vector $Y$. From the EBP equation, if $F$ denotes the fitness function, the weight updation equations for $W_{ho}$ and $W_{ih}$ are given by,

$$\Delta W_{ho} = -\eta \left( \frac{\partial F}{\partial Y} \right) D^T, \quad \Delta W_{ih} = -\eta W_{ho}^T \left( \frac{\partial F}{\partial Y} \right) X^T \tag{5}$$

From Eq. (5) it can be noted that weight updation depends on the input vector $X$ presented to the neural network. If such a neural network is to be used for optimization, then the question one can ask is what would be the inputs to this neural network, if the search parameters are its output? A possible immediate answer could be the optimization search parameters itself hence making it a Recurrent Neural Network (RNN) as the case with the Hopfield/Tank Neural Network (HTNN). It turned out later that this adoption was not the best solution although it performed the optimization. The final adoption will be discussed later in this section after highlighting the problems with the HTNN based adoption.

There are three factors that can bring the weight updations of Eq. (5) to instability. They are 1) large inputs, due to the initialization at locations with large values, 2) large gradients, due to large steeps in the fitness function, and 3) large weights, due to large initializations of the weights.

In optimization problems, the algorithms typically initialize optimization search parameters at random locations. In the case of our neural network optimizer, the initialization not only occurs with the optimization search parameter location but also with the weights of the neural network. If the randomly initialized optimization search parameter is presented as the input to the randomly initialized neural network, the output of the neural network will be another random optimization search parameter making the initial initialization worthless. Also since large weight initialization can cause instability in the EBP equation, initialization can be forced with smaller weights. Smaller initial weights in fact tend to drive the initial output of the neural network to small values and this poses a lack of controllability on the initialization, especially if the optimization problem is a constrained optimization problem.

One possible approach to overcome this problem is to make the randomly initialized optimization search parameter the output of the neural network and find the corresponding input based on the randomly initialized weights. This, however, can be a problem if the weights are initialized to small values. The problem of singularity occurs. Therefore, the possible best approach is to initially train the neural network to equalize the output with the randomly initialized optimization search parameter presented as the input of the neural network.

This, however, does not solve the problem of large values in the optimization search parameter even during the training for equalization. Therefore normalization of the initial optimization search parameter is essential to ensure stability of the equalization training process. Then the outputs of the neural network can be denormalized based on the initial optimization search parameter before being inputted to the fitness function. The normalization strategy minimizes the requirement for an equalization procedure each time the ACLA algorithm is invoked. An equalized neural network can be pre-trained beforehand, for any change in problem dimensionality, and be hardwired to the ACLA algorithm. To accommodate the normalization, the EBP equation given in Eq. (5) will need to be modified. If $X_0$ is the initial optimization search parameter, then the denormalization weights $G$ equals $X_0$. Since these are weights following the output of the neural network, the modified EBP equation should contain the vector $G$. This is given in Eq. (6).

$$\Delta W_{ho} = -\eta G \left( \frac{\partial F}{\partial Y} \right) D^T, \quad \Delta W_{ih} = -\eta G W_{ho}^T \left( \frac{\partial F}{\partial Y} \right) X^T \tag{6}$$

In Eq. (6), if $X_0$ was initialized to large values, it can also cause instability and so a factor $k$ is introduced to abate the effect of large $G$ only for the EBP equation.

$$\Delta W_{ho} = -\eta k G \left( \frac{\partial F}{\partial Y} \right) D^T, \quad \Delta W_{ih} = -\eta k G W_{ho}^T \left( \frac{\partial F}{\partial Y} \right) X^T$$
$$k = \begin{cases} 1/\max(|X_0|) & \forall \max(|X_0|) > \varepsilon_k \\ 1 & otherwise \end{cases} \tag{7}$$

A similar effect can be done for the large gradients $\partial F/\partial Y$. This is given in Eq. (8).

$$\frac{\partial F}{\partial Y} = \begin{cases} \left( \frac{\partial F}{\partial Y} \right) \Big/ \max\left( \left| \frac{\partial F}{\partial Y} \right| \right) & \forall \max\left( \left| \frac{\partial F}{\partial Y} \right| \right) > \varepsilon_f \\ \frac{\partial F}{\partial Y} & otherwise \end{cases} \tag{8}$$

Assuming the above equations are used with the HTNN adoption, i.e. making the neural network a RNN along with EBP updation, the neural network was able to converge the output to the optimal solution. However, there was a problem when the optimization search parameters neared all zeros in a fitness function with non-zero optimal parameters. The ACLA algorithm updated its output very slowly even when a bias was presented at the input side. Therefore to overcome this problem it was decided that a vector of all ones be persistently applied to the input of the neural network throughout the optimization run, hence eliminating the RNN adoption. The pseudo-code of the ACLA algorithm is given in Fig. 3 and the neural network is shown in Fig. 4. Notice that only one neuron is used in the ACLA neural network. This is done to ensure minimal memory requirements for the algorithm.

Pseudo Code:

Let us call the optimization search parameter vector a particle. Assume that an equalized neural network is already trained and denote it as $X_n = N(1)$ with $X_n$ denoting the normalized output.

1)Initialize ACLA particle position $X_0$.
2)Set $G = X_0$.
3)Evaluate the gradient $\partial F/\partial X_0 = (F(X_0+hI) - F(X_0))/h$, where $h$ is some small value s.t. $h \to 0$ and $I$ is the identity matrix.
4)Apply Eq.s (7) – (8) to update the weights of the neural network.
5)Calculate $X_n = N(1)$ and $X = GX_n$.
6)Repeat steps 3 to 5 until termination criterion.

Fig. 3. Pseudo Code for ACLA algorithm.

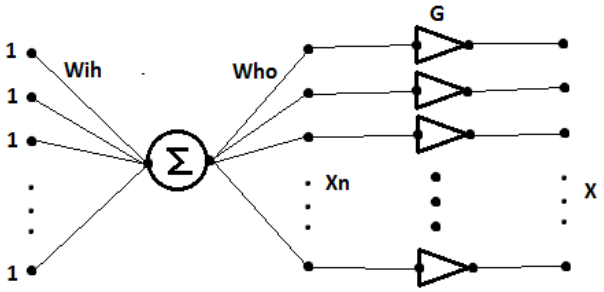


Fig. 4. The ACLA Neural Network.

## V. Experimental Setup

This section discusses the experimental setup made to test the ACLA algorithm.

### A. Benchmark Functions

Two unimodal benchmark functions typically used for evaluating the PSO and GA algorithms were used to test the ACLA algorithm. They are the Sphere and Rosenbrock functions given in Eq.s (9) and (10) and shown in Fig. 5 and Fig. 6. These benchmark functions and their respective ranges were used by Venayagamoorthy in [11] and so we used the same.

$$Sphere\ (x) = \sum_{i=1}^{n} x_i^2 \qquad -5.12 \le x_i \le 5.12 \tag{9}$$

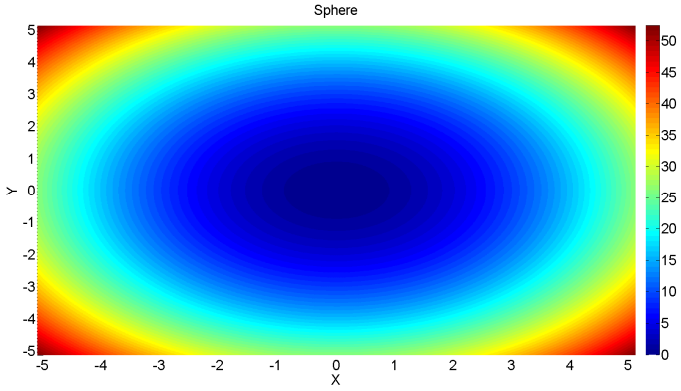


Fig. 5. The Sphere Function.

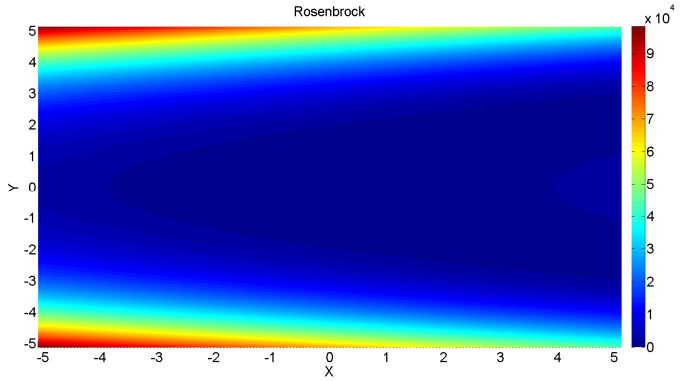


Fig. 6. The Rosenbrock Function.

$$Rosenbrock\ (x) = \sum_{i=1}^{n-1} (100 \cdot (x_{i+1} - x_i^2) + (1 - x_i)^2)$$
$$-2.05 \le x_i \le 2.05 \tag{10}$$

### B. ACLA Particle Trajectory on 2D Space

In this subsection, the trajectory of the ACLA particle on 2D versions of the benchmark problems will be shown.

Fig.s 7 and 8 show the ACLA particle's trajectory on 2D versions of the benchmark functions. The ACLA parameters, $\varepsilon_k$ and $\varepsilon_f$ of Eq.s (7) and (8), used were 10 and 100 respectively, and will be used throughout this paper. The learning coefficients $\eta$ used were different for the different functions.

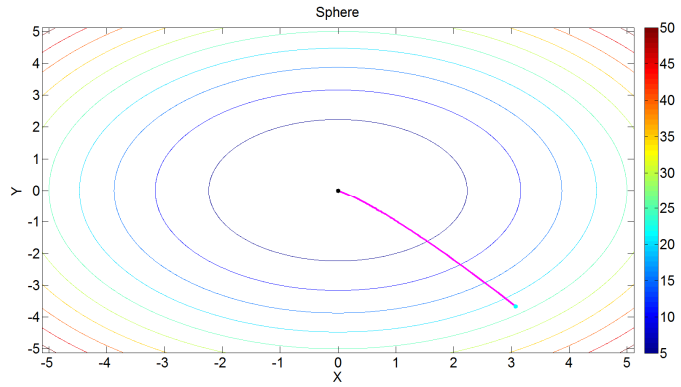


Fig. 7. Example ACLA particle trajectory on the Sphere Function.

TABLE I. PERFORMANCE OF ACLA ON 10 DIMESIONAL BENCHMARK FUNCTIONS

| Function | No. of Iterations | ACLA | | PSO | | SPPSO | | MVO | |
|---|---|---|---|---|---|---|---|---|---|
| | | *Learning Coeff.* | *Sample Result* | *Parameter [w c1 c2 Vmax]* | *Sample Result* | *Parameter [w c1 c2 Vmax refresh]* | *Sample Result* | *Parameter [AF fs m]* | *Sample Result* |
| Sphere | 30000 | 0.001 | 5.2659e-14 | [0.8 2 2 8.02] | 5.2187e-3 | [0.8 2 2 4.23 500] | 7.1327e-45 | [1 1 1] | 7.1234e-43 |
| | 30000 | 0.01 | 3.7209e-25 | [0.7 2.5 2 9.52] | 1.9028e-3 | [0.7 2.5 2 1.74 500] | 6.5542e-38 | [1 1 3] | 7.4471e-10 |
| | 30000 | 0.005 | 2.2443e-15 | [0.9 2 2.5 1.79] | 5.1114e-3 | [0.9 2 2.5 3.88 500] | 1.9579e-10 | [1 1 5] | 1.8481e-29 |
| Rosenbrock | 100000 | 1e-3 – 5e-4 | 1.1395e-8 | [0.8 2 2 4.32] | 1.9261 | [0.8 2 2 9.17 500] | 5.2839 | [1 1 1] | 0.5472 |
| | 100000 | 5e-4 – 1e-5 | 5.9484e-4 | [0.7 2.5 2 5.72] | 0.1235 | [0.7 2.5 2 9.6 500] | 3.4882 | [1 1 3] | 8.4954 |
| | 100000 | 1e-4 – 1e-5 | 2.2483e-3 | [0.9 2 2.5 5.85] | 3.9624 | [0.9 2 2.5 5.39 500] | 8.9522 | [1 1 5] | 0.5514 |

TABLE II. PERFORMANCE OF ACLA ON 30 DIMESIONAL BENCHMARK FUNCTIONS

| Function | No. of Iterations | ACLA | | PSO | | SPPSO | | MVO | |
|---|---|---|---|---|---|---|---|---|---|
| | | *Learning Coeff.* | *Sample Result* | *Parameter [w c1 c2 Vmax]* | *Sample Result* | *Parameter [w c1 c2 Vmax refresh]* | *Sample Result* | *Parameter [AF fs m]* | *Sample Result* |
| Sphere | 30000 | 0.001 | 1.9641e-16 | [0.8 2 2 4.92] | 0.3143 | [0.8 2 2 3.17 500] | 1.3351e-16 | [1 1 1] | 1.5141e-29 |
| | 30000 | 0.002 | 1.1006e-15 | [0.7 2.5 2 1.97] | 5.4594e-2 | [0.7 2.5 2 9.89 500] | 5.0041e-4 | [1 1 3] | 4.0563e-31 |
| | 30000 | 0.0005 | 1.0978e-13 | [0.9 2 2.5 4.37] | 0.3239 | [0.9 2 2.5 1.15 500] | 2.0048e-4 | [1 1 5] | 1.0967e-23 |
| Rosenbrock | 200000 | 5e-4 – 1e-5 | 5.2745e-4 | [0.8 2 2 2.65] | 46.5901 | [0.8 2 2 6.67 500] | 26.7892 | [1 1 1] | 7.8927 |
| | 200000 | 2e-4 – 1e-5 | 2.8575e-7 | [0.7 2.5 2 4.76] | 1.0441e+3 | [0.7 2.5 2 6.46 500] | 82.6185 | [1 1 3] | 5.6712 |
| | 200000 | 1e-4 – 1e-5 | 2.7488e-5 | [0.9 2 2.5 3.75] | 1.6608e+2 | [0.9 2 2.5 4.61 500] | 50.8158 | [1 1 5] | 9.9696 |

For the Sphere Function, $\eta$ of 0.01 was used and for the Rosenbrock Function, $\eta$ of 0.0001 was used. The learning coefficient plays a vital role in the speed and stability with which the ACLA particle converges to the solution. It is problem dependent and the best learning coefficient can be found by trial and error. For example, the 2D Sphere Function can tolerate learning coefficients as large as 0.5 but the Rosenbrock Function can only tolerate learning coefficients as large as 0.0001. For higher dimensional versions of the functions, these may vary.

From Fig.s 7 and 8, the ACLA particle was able to converge to the optimal solutions (0,0) and (1,1) for the Sphere Function and Rosenbrock Function respectively.

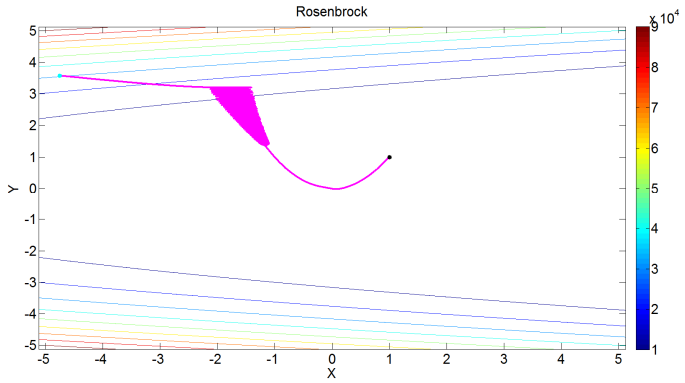


Fig. 8. Example ACLA particle trajectory on the Rosenbrock Function.

### C. *Test Results of ACLA Algorithm on Higher Dimensional versions of the Benchmark Functions*

In this subsection, the ACLA algorithm was tested on higher dimensional versions of the Sphere and Rosenbrock functions. The results obtained by the ACLA algorithm were compared with the results obtained by the traditional PSO, the SPPSO and the MVO algorithms. Table I and Table II show the test results for dimension of 10 and for dimension of 30 respectively. For all the swarm based algorithms only two particles were used to match the ACLA algorithm's memory requirements. From Fig. 4, it can be noted that the ACLA neural network needs memory equivalent to two particles. One for the $W_{ih}$ and one for the $W_{ho}$. The rest of the parameters of the PSO, SPPSO and MVO are explained in the references [1], [11] and [12] respectively.

From Tables I and II, it can be noted that the ACLA algorithm has superior performance for the Rosenbrock Function and comparatively not a bad performance for the Sphere Function. For the Rosenbrock Function, the learning coefficient was manually varied in the specified ranges in Tables I and II to enhance the convergence rate of the ACLA particle. This is the significant advantage of the ACLA algorithm in that only a single parameter is required to tune the convergence rate on the run. This is in contrast to the swarm based algorithms that have three or more of such parameters. Tables I and II show proof that the ACLA algorithm is in fact capable of converging to the optimal solutions.

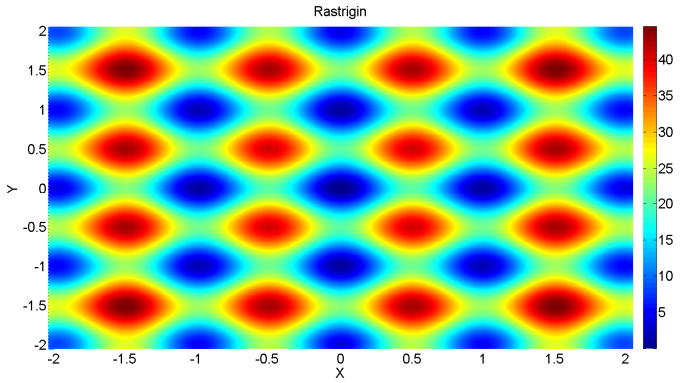


Fig. 9. The Rastrigin Function.

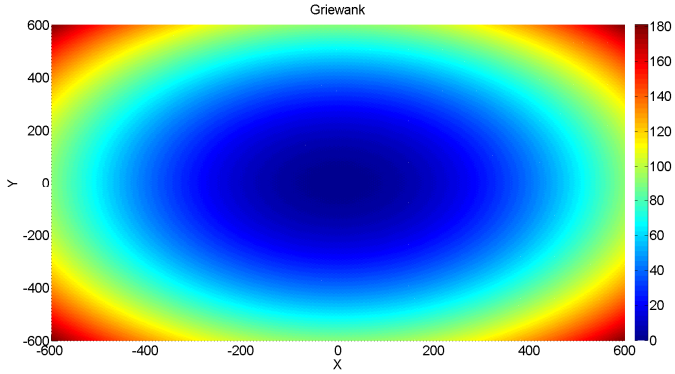


Fig. 10. The Griewank Function.

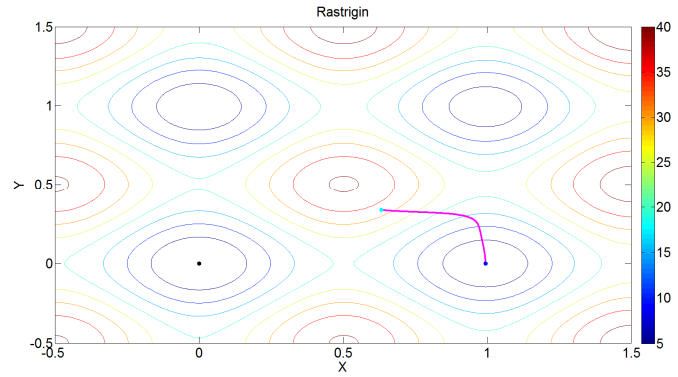


Fig. 11. Example ACLA particle trajectory on the Rastrigin Function.

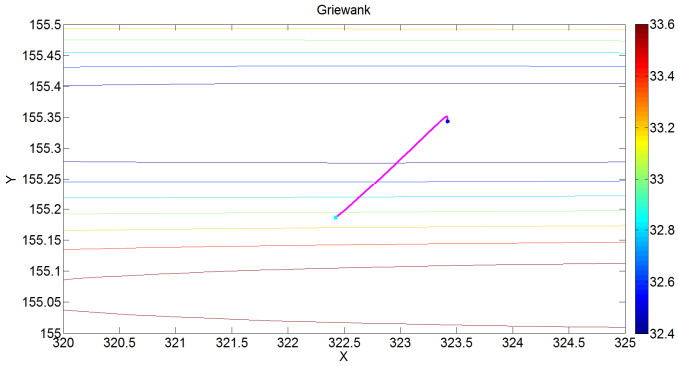


Fig. 12. Example ACLA particle trajectory on the Griewank Function.

### *D. Convergence of the ACLA Algorithm towards Local Minima on Multimodal Benchmark Functions*

In this subsection, the ACLA algorithm's capability to converge to the local minima on multimodal benchmark functions will be shown.

The multimodal benchmark functions used in this setup were the Rastrigin and Griewank functions given in Eq.s (11) and (12) and shown in Fig.s 9 and 10.

$$Rastrigin\ (x) = 10 \cdot n + \sum_{i=1}^{n} (x_i^2 - 10 \cdot \cos(2\pi \cdot x_i)) \qquad -5.12 \le x_i \le 5.12 \tag{11}$$

$$Griewank\ (x) = \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1 \qquad -600 \le x_i \le 600 \tag{12}$$

The Griewank function of Fig. 10 seems to appear like a unimodal function as it resembles the Sphere Function. Actually, the Griewank Funcation is a coarse Sphere function with multiple subtle peaks and troughs.

Fig.s 11 and 12 show the trajectory of the ACLA particle on a section of the Rastrigin and Griewank Functions. It can be noted that the ACLA particle successfully converged to point where the gradient is zero.

## VI. Conclusion and Future Work

The ACLA algorithm presented in this paper was developed based on the principles of Adaptive Critic Design (ACD). The algorithm proved to converge to the optimal solutions on unimodal benchmark functions. For the multimodal benchmark functions, the ACLA algorithm proved to converge to the local minima. The convergence was successful for both lower and higher dimensional versions of the benchmark function. To summarize the achievement of the work presented in this paper, 1) the ACLA algorithm uses a general neural network, unlike the Hopfield/Tank Neural Network, to find the optimal solution, 2) the ACLA algorithm uses the simple Error Backpropagation (EBP) algorithm to converge the neural network to the optimal solution, 3) the ACLA algorithm requires minimal memory requirements and 4) the ACLA algorithm uses only a single parameter to tune its convergence rate. For future work, we are looking to develop the ACLA algorithm further to tackle multimodal problems.

## VII. Acknowledgement

The authors acknowledge support for this work from Idaho National Laboratory through the U.S. Department of Energy Office of Electrical Delivery and Energy Reliability under DOE Idaho Operations Office Contract DE-AC07-05ID14517.

## REFERENCES


[1] J. Kennedy and R. Eberhart, "Particle Swarm Optimization," *Proc. IEEE Intl. Conf. on Neural Networks*, 1995, pp. 1942–1948.

[2] A. Fraser, "Simulation of Genetic Systems by Automatic Digital Computers," Australian Journal of Biological Sciences, vo1. 10, pp. 484 – 491, 1957.

[3] A. Fraser and D. Burnell, *Computer Models in Genetics*. McGraw-Hill, 1970.

[4] J. Crosby, *Computer Simulation in Genetics*. John Wiley and Sons, 1973.

[5] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Univ. of Michigan Press, Ann Arbor, 1975.

[6] J. J. Liang, A. K. Qin, P. N. Suganthan and S. Baskar, "Comprehensive Learning Particle Swarm Optimizer for Global Optimization of Multimodal Functions," *IEEE Trans. on Evolutionary Computation*, vol. 10, no. 3, pp. 281 – 295, Jun. 2006.

[7] A. Moraglio, C. D. Chio, J. Togelius and R. Poli, "Geometric Particle Swarm Optimization," *Journal of Artificial Evolution and Applications*, pp. 1 – 14, 2008.

[8] S. Yang and C. Li, "A Clustering Particle Swarm Optimizer for Location and Tracking Multiple Optima in Dynamic Environments," *IEEE Trans. on Evolutionary Computation*, vol. 14, no. 6, pp. 959 – 974, Dec. 2010.

[9] H. Babaee and A. Khosravi, "An improve PSO based Hybrid Algorithms," *Proc. Intl. Conf. on Management and Service Science*, 2011, pp. 1 – 5.

[10] S. Huisheng and Z. Yanmin, "An Improved Cooperative PSO Algorithm," *Proc. Intl. Conf. on Mechatronic Science, Electric Engg, and Computer*, 2011, pp. 1040 – 1042.

[11] P. Mitra and G. K. Venayagamoorthy, "Empirical study of a hybrid algorithm based on Clonal Selection and Small Population based PSO," *Proc. IEEE Symposium on Swarm Intelligence*, 2008, pp. 1 – 7.

[12] I. Erlich, G. K. Venayagamoorthy and N. Worawat, "A Mean-Variance Optimization Algorithm," *Proc. IEEE Congress on Evolutionary Computation*, 2010, pp. 1 – 6.

[13] J. J. Hopfield and D. W. Tank, "'Neural' Computation of Decisions in Optimization Problems," *Biological Cybern.*, vol. 52, pp. 141 – 152, 1985.

[14] P. W. Protzel, D. L. Palumbo and M. K. Arras, "Performance and Fault-Tolerance of Neural Networks for Optimization," *IEEE Trans. on Neural Networks*, vol. 4, no. 4, pp. 600 – 614, Jul. 1993.

[15] Y. Xia and J. Wang, "A General Projection Neural Network for Solving Optimization and Related Problems," *Proc. Intl. Joint Conf. on Neural Networks*, vol. 3, 2003, pp. 2334 – 2339.

[16] W. Bian and X. Xue, "Subgradient-based Neural Networks for Nonsmooth Nonconvex Optimization Problems," *IEEE Trans. on Neural Networks*, vol. 20, no. 6, 1024 – 1038, Jun. 2009.

[17] Q. Liu, C. Dang and J. Cao, "A Novel Recurrent Neural Network with One Neuron and Finite-Time Convergence for *k*-Winners-Take-All Operation," *IEEE Trans. on Neural Networks*, vol. 21, no. 7, 1140 – 1148, Jul. 2010.

[18] Q. Liu and J. Wang, "Finite-Time Convergent Recurrent Neural Network with a Hard-Limiting Activation Function for Optimization with Piecewise-Linear Objective Functions", *IEEE Trans. on Neural Networks*, vol. 22, no. 4, pp. 601 – 613, Apr. 2011.

[19] Q. Liu and J. Wang, "A One-Layer Recurrent Neural Network for Constrained Nonsmooth Optimization," *IEEE Trans. on Systems, Man and Cybern. – Part B: Cybernetics*, vol. 41, no. 5, 1323 – 1333, Oct. 2011.

[20] P. Werbos, "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences," Ph.D. dissertation, Committee on Applied Mathematics, Harvard Univ., Cambridge, MA, 1974.

[21] R. L. Welch and G. K. Venayagamoorthy, "HDP based Optimal Control of a Grid Independent PV System," *IEEE Power Engg. Society Genral Meeting*, 2006, pp. 1 – 6.

[22] S. Mohagheghi, G. K. Venayagamoorthy and R. G. Harley, "Adaptive Critic Design based Neuro-Fuzzy Controller for a Static Compensator in a Multimachine Power System," *IEEE Trans. on Power Systems*, vol. 21, no. 4, pp. 1744 – 1754, Nov. 2006.

[23] R. L. Welch and G. K. Venayagamoorthy, "Comparison of two Optimal Control Strategies for a Grid Independet Photovoltaic System," *Proc. Intl. Conf. on Industry Applications*, 2006, vol. 3, pp. 1120 – 1127.

[24] S. Ray, G. K. Venayagamoorthy, B. Chaudhuri and R. Majumder, "Comparison of Adaptive Critic based and Classical Wide-Area Controller for Power Systems," *IEEE Trans. on Systems, Man and Cybernetics – Part B: Cybernetics*, vol. 38, no.4, pp. 1002 – 1007, Aug. 2008.

[25] J-W Park, G. K. Venayagamoorthy and R. G. Harley, "Adaptive Critic Designs and their Implementations on Different Neural Network Architectures," *Proc. Intl. Joint Conf. on Neural Networks*, 2003, vol. 3, pp. 1879 – 1884.

[26] D. W. Tank and J. J. Hopfield, "Simple 'Neural' Optimization Networks: An A/D Converter, Signal Decision Circuit, and a Linear Programming Circuit," *IEEE Trans. on Circuits and Systems*, vol. cas-33, no. 5, pp. 533 – 541, May 1986.

[27] J. Si, A. G. Barto, W. B. Powell and D. Wunsch, *Handbook of Learning and Approximate Dynamic Programming*. New York: Wiley, Jul. 2004.

[28] P. Werbos, "Backpropagation and neurocontrol: A review and prospectus," *Proc. Intl. Joint Conf. on Neural Networks*, 1989, vol. 1, pp. 209 – 216.

[29] D. V. Prokhorov and D. C.Wunsch, II, "Adaptive critic designs," *IEEE Trans. Neural Networks*, vol. 8, no. 5, pp. 997 – 1007, Sep. 1997.