Proceedings of the World Congress on Neural Networks, pp. 1091-5, 1996

A General Evolutionary/Neural Hybrid Approach to Learning Optimization Problems

Dan Ventura
Tony R. Martinez

Computer Science Department, Brigham Young University, Provo, Utah 84602
e-mail: dan@axon.cs.byu.edu, martinez@cs.byu.edu

A method combining the parallel search capabilities of Evolutionary Computation (EC) with the generalization of Neural Networks (NN) for solving learning optimization problems is presented. Assuming a fitness function for potential solutions can be found, EC can be used to explore the solution space, and the survivors of the evolution can be used as a training set for the NN which then generalizes over the entire space. Because the training set is generated by EC using a fitness function, this hybrid approach allows explicit control of training set quality.

1 INTRODUCTION

Efforts to develop mechanisms that can be considered in some way intelligent have resulted in many problem solving techniques from various fields including Machine Learning, Neural Networks, Evolutionary Computation, Fuzzy Logic, and Symbolic Artificial Intelligence. Approaches from these different fields each have their strengths and their weaknesses. Recent research suggests that further progression may depend upon the synergistic combination of several complementary approaches from these different fields [4][6][11]. This new synergistic approach to machine intelligence is sometimes referred to as Hybrid Systems. One such possibility is the combination of Evolutionary Computation (EC) [3][10] with Neural Networks (NN) [9][13]. For some examples of such combinations, the reader is referred to [1][5][7][8].

This paper extends work on one such hybrid method, first presented in [12], that combines the broad, parallel search capabilities of EC with the generalization and execution speed of NN. Given a function to be learned, under certain reasonable assumptions the EC can be used to solve the problem at selected points in the input space; the survivors of the evolution become the instances in a training set for the NN, which then generalizes the optimization for the entire input space.

Section two of the paper presents a generalized formal description of the problem to be solved. Section three then discusses the combination of NN with EC as a general approach to solving the problem. As a proof-of-concept for the general approach, section four describes a specific learning optimization problem and presents empirical results from simulations run on that problem. Finally, section five presents conclusions and directions for ongoing research.

2 PROBLEM DESCRIPTION

Given a system, Θ, the state of Θ may be described at time t by a vector of status variables, $\boldsymbol{s}^t$. Control of the system is effected by Γ which applies a control vector, $\boldsymbol{c}$, to Θ. That is, given a system Θ at time t described by vector $\boldsymbol{s}^t$, the setting of the values of the vector $\boldsymbol{c}$ will result in a different system Θ' at time $t+\delta$ described by the vector $\boldsymbol{s}^{t+\delta}$. The problem is how should Γ be constructed so that given a status vector, $\boldsymbol{s}^t$, Γ outputs a control vector $\boldsymbol{c}$ such that $\boldsymbol{s}^{t+\delta}$ describes a better system, if possible, than $\boldsymbol{s}^t$? We assume the existence of some evaluation function, f, that will determine whether or not one system is better than another. The operation of Θ may be either continuous or discrete and Γ knows nothing about Θ except for the value of $\boldsymbol{s}^t$. Given $\boldsymbol{s}^t$, Γ is expected to output values for $\boldsymbol{c}$, the goal being to maximize f for any given instance (status) of Θ. The problem may be a single-step optimization; alternatively, though this is not pursued here, the problem may include feedback with the process becoming iterative (see Figure 1).

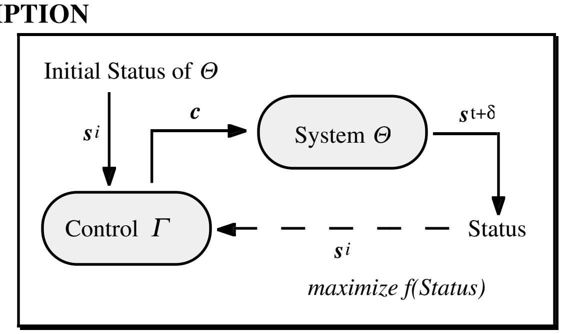

Figure 1. Problem Description

3 COMBINING EVOLUTIONARY COMPUTATION WITH NEURAL COMPUTATION

Assuming that the status and control variables are defined over even a modest range, it is obvious that the input (status) and output (control) spaces will be extremely large. Evolutionary computation lends itself well to the exploration of large spaces. However, such evolutionary exploration is often slow. If we assume that the mapping $\boldsymbol{s} \rightarrow \boldsymbol{c}$ is non random and in some sense generelizeable, then a (hopefully) representative set of points may be discovered via EC, and those points then used to train a NN which may then generalize over the rest of the function. The goal of the EC/NN synergism is to obtain the accuracy of evolutionary search and the speed of neural execution.

From the space defined by $\boldsymbol{s}$ that describes Θ we choose a representative set of system states by choosing n initial status vectors. We denote these $\boldsymbol{s}_i^{t=0}$, $0<i\leq n$ and refer to the system state described by $\boldsymbol{s}_i^{t=0}$ as $\Theta_i^{t=0}$, $0<i\leq n$. These choices could of course be biased by any a priori heuristics as to what constitutes a realistic system (see below and section 5). The goal is to know, given one of these $\boldsymbol{s}_i^{t=0}$, what a "good" $\boldsymbol{c}$ vector would be. For each of the $\boldsymbol{s}_i^{t=0}$, EC is used to discover such a $\boldsymbol{c}$ in the following way.

evolve()
    generate $\boldsymbol{s}_i$
    for each $\boldsymbol{s}_i^{t=0}$
        initialize population $\boldsymbol{c}_k$, $0<k\leq m$.
        evaluate($\boldsymbol{c}_k$,Θ_i)
        until(done)
            select parents from $\boldsymbol{c}_k$
            apply genetic operators to parents
            evaluate(children,Θ_i)
            $\boldsymbol{c}_k \leftarrow$ choose m survivors from $\boldsymbol{c}_k$ and children

evaluate($\boldsymbol{c}$,Θ_i)
    run $\Theta_i^{t=0}$ for δ time steps with control vector $\boldsymbol{c}$
    return $f(\boldsymbol{s}_i^{t=\delta})$

Figure 2. Algorithm for evolving training set

Assume a fitness function f that takes as input a status vector $\boldsymbol{s}$ and returns a real-valued fitness measure. Now for each $\boldsymbol{s}_i^{t=0}$, randomly initialize a population of m control vectors, denoted $\boldsymbol{c}_k$, $0<k\leq m$. Evaluate the initial population by simulating the workings of $\Theta_i^{t=0}$ for δ time steps (where δ time steps are sufficient for Θ_i to stabilize) for each $\boldsymbol{c}_k$, and then applying fitness function f to $\boldsymbol{s}_i^{t=\delta}$. Next, until some stopping criterion is reached (a maximum number of generations, for example) choose parents and use genetic operators (crossover, mutation, etc.) to produce m offspring, evaluate the children and select m survivors from amongst the parents and children. The algorithm is sketched in figure 2.

Finally, for each of the n populations, choose the individual, $\boldsymbol{c}_{max_i}$, (the individual with the highest fitness) and build a set of n training examples of the form $\boldsymbol{s}_i^{t=0} \rightarrow \boldsymbol{c}_{max_i}$.

The EC has now found "good" approximate solutions for n points from the input (status) space but can say nothing about any other points, many of which we are likely to encounter during normal execution of Θ. Obviously, one solution to the problem defined in section 2 would be to employ the evolutionary scheme discussed above as the control Γ. However, this would likely be unacceptable in terms of execution speed. Therefore, the NN is employed to generalize over the entire space defined by $\boldsymbol{s}$ using the relatively small set of approximate solutions as a training set. While the initial training of the network maybe somewhat time consuming, depending on the network and training algorithm employed, the generalization during execution will be extremely fast. The synergistic combination of EC and NN is then employed as the control Γ as in figure 3.

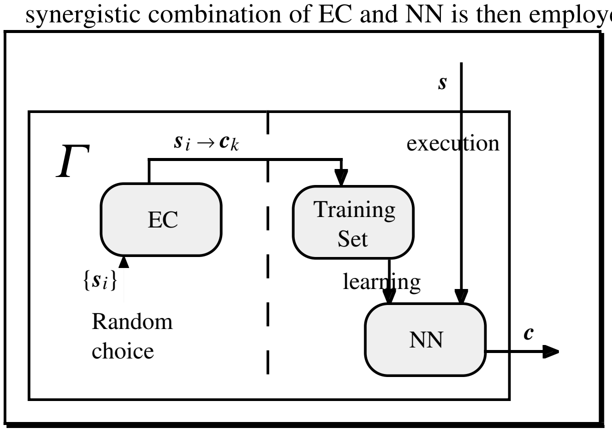

Figure 3. The control Γ

In experiments performed to date, the original set $\boldsymbol{s}_i^{t=0}$ is simply chosen randomly. However, it is possible that techniques such as active learning [2] can be employed to choose the set $\boldsymbol{s}_i^{t=0}$, leading to smaller training and/or better NN generalization.

The power of this EC/NN hybrid approach is its combination of the thoroughness of evolutionary search with the speed of neural generalization as well as its general applicability to any learning problem for which a fitness function can be found and for which "blackbox" access to the system to be learned (or a reasonable simulation thereof) is possible. In order to provide proof-of-concept, the next section presents a specific example of a problem and the results of applying the EC/NN hybrid to solve it.

4 SIMULATION

In the simulations, the EC/NN combination attempts to learn to use a simple gun to hit a target moving in 2-d space. An artificial problem was used for two reasons. First, it is much easier to work with in terms of analysis, reproduction of results, etc. Second, it is possible to create a test set which can be used to show how well the NN is performing in relation to optimum, and thus to establish (to some extent) both the quality of the training set generated as well as the quality of the NN's generalization.

The problem is difficult in several respects. Most notably in the fact that there are many optimal bullets (in the sense that they hit the target) for each target, and no attempt is made to constrain which of these is genetically chosen. However, it is unclear how a neural network trained with these two instances would generalize.

A second difficulty arises from the fact that only integer values are allowed for velocity values. This makes the optimization much more difficult than if real values were allowed, analogous to the difference in difficulty between the linear and integer programming problems.

4.1 The system Θ

The vector $\boldsymbol{s}$ describes the *x* and *y* components of the target's velocity and is therefore comprised of two elements: s_1 and s_2. The vector $\boldsymbol{c}$ describes the *x* and *y* components of the bullet's velocity and so consists of two elements: c_1 and c_2. The target vector is subject to a gravitational constant, but the bullet vector is not (it is assumed negligible). The system Θ describes the trajectories of both bullet and target as in figure 4.

Notice that the function *trajectories*() returns the minimum distance between bullet and target in the course of their trajectories. It therefore acts as the system simulator and the fitness function. Optimizing this system entails minimizing the Euclidean distance between target and bullet at their nearest point. The fitness function therefore is simply

$$fitness(\boldsymbol{s}) = 1 - \text{mindistance}(\boldsymbol{s},\boldsymbol{c})$$

which is to be maximized. The default fitness is the distance between the initial bullet position and the initial target position. Obviously, the ideal fitness is 0, and thus *fitness*() is defined over the range [0,default fitness].

```
trajectories(targetx, targety, bulletx, bullety)
    targetx = targetx + targetxvelocity
    targety = targety + targetyvelocity
    bulletx = bulletx + bulletxvelocity
    bullety = bullety + bulletyvelocity
    if(mindistance > ((targetx-bulletx)+(targety-bullety)^2)^(1/2))
        mindistance = ((targetx-bulletx)+(targety-bullety)^2)^(1/2)
    targetyvelocity = targetyvelocity - gravity
return(mindistance)
```

Figure 4. The target/bullet system

4.2 Creating a training set

Training sets were comprised of 250 instances obtained using the evolutionary method described above. For each randomly generated target (left hand side), a population of bullets was allowed to evolve and the bullet that came closest to hitting the target was selected as the right hand side. The function *trajectories* is substituted for the step "run $\Theta_i^{t=0}$ for δ time steps with control vector $\boldsymbol{c}$" in the *evaluate* procedure, and

$$fitness(\text{bullet}) = \text{mindistance}(\text{bullet, target}).$$

4.3 Training the neural network

The PDP group's software implementation of the backpropagation algorithm [9] was used for all simulation results. For the simulations, 2 integer "status" variables and 2 integer "control" variables were defined that describe the *x* and *y* velocities of the target and the bullet respectively. The inputs to the backpropagation NN (the "status" variables) were binary encoded, while the outputs were simple localist nodes, one for each control variable. Ten hidden units were used so that the entire network consisted of 12 input nodes, 10 hidden units (number of hidden nodes was determined empirically), and 2 output nodes. The output nodes produce values in the range (0, 1) and these are scaled to produce an integer "control" vector in the desired range. All NN simulations were run with the default settings and training was allowed to proceed for 1000 epochs.

4.4 Creating a test set

Creating a test set for this problem is extremely simple. Obviously, the optimum fitness for any bullet is 0. Therefore, to create a test set, simply randomly generate targets (left hand sides) that have not been seen by the NN before. Right hand sides are not necessary as explained below. All test sets generated contained 20 instances.

4.5 Testing the neural network

For this problem, the NN is expected to produce $\boldsymbol{bullet}_{approx}$, which represents the *x* and *y* velocities of a bullet that should come close to (hopefully hit) the target. Thus, given a target vector ***target*** the NN generates the bullet vector $\boldsymbol{bullet}_{approx}$. *Trajectories(***target,** $\boldsymbol{bullet}_{approx}$*)* is then used to determine how close $\boldsymbol{bullet}_{approx}$ comes to the target. Even though both $\boldsymbol{bullet}_{opt}$ and $\boldsymbol{bullet}_{worst}$ are unknown, *trajectories(****target,*** $\boldsymbol{bullet}_{opt}$*)* = 0 and *trajectories(****target,*** $\boldsymbol{bullet}_{worst}$*)* = default fitness (the default minimum distance between the target origin point and the bullet origin point in the simulation). A measure of the correctness (normalized % of optimum) of $\boldsymbol{bullet}_{approx}$ is therefore

$$\frac{trajectories(target,\ bullet_{worst}) - trajectories(target,\ bullet_{approx})}{trajectories(target,\ bullet_{worst}) - trajectories(target,\ bullet_{opt})} = 1 - \frac{trajectories(target,\ bullet_{approx})}{\text{default fitness}}.$$

4.6 Empirical Results

All simulations were run 15 times and the results averaged. Initially, the evolutionary computation was able in every case to generate a perfect training set in the sense that all the examples had the maximum fitness (all were direct hits). This seems to indicate that the EC has generated a training set that faithfully represents the problem and the results of the NN's generalization accuracy bear this out.

Next, varying levels of noise were introduced into the training sets. The majority of instances in each training set were still direct hits, but each training set also contained a few instances with very poor fitness (the bullet did not come anywhere near the target). One of the most difficult aspects of learning with NN is obtaining a good training set. In contrast, the EC/NN hybrid approach described here allows for explicit control of training set quality because of the fact that a fitness function exists. Therefore, the fitness or quality of each individual example as well as the overall fitness of the training set (taken as the average) is known. Any examples whose fitness is below a certain threshold can simply be thrown out.

Table 1 presents the results of training a backpropagation NN with 10 hidden nodes on the genetically generated training sets and then testing it on test sets that it had not seen before. The first column of results is from simulations run with no noise in which the EC was always able to find training sets comprised entirely of examples with maximum fitness (direct hits). The remaining three columns are from simulations run with varying degrees of noise injected into the training sets, with each column representing the results from a different noise level as labeled. Each is further divided in half with the left half representing the results of training the NN on the noisy training set and the right half the results of training the NN on the smaller training set obtained by eliminating the noisy example using the fitness function.

Noise level	0%	15%		25%		35%	
Average training set fitness (EC)	1.00	.958	1.00	.915	1.00	.866	1.00
Average NN generalization	.943	.876	.920	.804	.834	.775	.910
Best NN generalization	.991	.944	.996	.925	.895	.887	.999
Worst NN generalization	.851	.757	.842	.632	.741	.653	.809
Average number of direct hits	10.8	5.5	10.2	4.0	4.4	2.7	11.3
Most direct hits (of 20)	16	9	16	7	6	5	19
Least direct hits (of 20)	3	2	3	2	0	1	4

Table 1. Results of simulation

The first row indicates the average training set fitness over all 15 runs. Recall, this is *not* the performance of the NN on the training set, but rather a measure of how good the training set (generated by the EC) is compared to the theoretical optimum of all instances being direct hits. As mentioned earlier, in the case of 0% noise, the EC was always able to create a training set with maximum fitness. This is an indication that the training set is a good representation of the problem. However, this was not the case with the noisy simulations. Still, the majority of the 250 instances in all the training sets were direct hits; however, in each of the three scenarios, a few instances of each training set in each run were always very poor. The second row indicates how well, on average, the NN generalized on a test set after being trained on one of the training sets. Since each of the 15 runs themselves represent averages over 250 training instances and 20 test instances, the first two rows of the table are really averages of averages.

The third row shows the best NN generalization over any of the test sets and the fourth row shows the corresponding worst generalization. The fifth row shows the average number of times (out of 20 test instances) that the NN actually hit the test target (a hit entails the minimum distance between bullet and target being 0). The sixth row shows the most hits in a single test set, and the last row shows the least hits in a single test set. The results are encouraging because they show that the EC can generate a good training set that allows the NN to generalize well over the entire problem space. From another perspective, the EC/NN hybrid has potential for effectively approximating a system.

For example, the best NN generalization in six of the seven scenarios (the right half of the 25% noise level column being the exception) is very close to the training set fitness. As for the average NN generalization under noisy conditions, the better the training set fitness (quality), the better the NN generalization. Obviously, the noisier the training set, the poorer the generalization (see the left halves of the three noisy columns). This is certainly no surprise. However, as discussed earlier, the EC/NN hybrid can be used to explicitly control training set quality by detecting and eliminating any training example whose fitness falls below an arbitrary threshold. This ability allows the hybrid to perform well even under noisy conditions (see the right halves of the three noisy columns). Also, averages of 10+ direct hits out of 20 test instances is respectable (again, the exception being the 25% column), given the EC/NN hybrid's complete lack of *a priori* knowledge for the problem, and the maximum number of direct hits for the 35% noise column (19) is excellent. (The one miss was within .00041 of being a hit.)

The results of the 25% noise column are a little less encouraging. In comparison to the other two noisy columns, detection and removal of noisy instances results in much less improvement in NN performance (still, *some* improvement is gained). This is probably due to the EC choosing somewhat unrepresentative points in the input space. Due to the nature of EC, this situation cannot always be avoided but increasing initial population size (if feasible) can minimize these effects.

5 CONCLUSION

Empirical simulation results suggest that for a broad class of problems (any problem for which a fitness function can be found to evaluate possible solutions), a combination of evolutionary and neural computation is indeed capable of efficiently learning to control a system by combining the thorough search capabilities of EC with the generalization and processing speed of NN. This is possible because the EC is able to generate a training set that closely represents the actual underlying function and the NN is then able to generalize from the training set. Further, the hybrid is capable of explicitly controlling the quality of the training set. The key to this process is the assumption that an appropriate fitness function *f* can be defined. We conjecture that in many cases this will indeed be the case (e.g. in a network control problem *f* would attempt to maximize throughput while minimizing resource usage). Although it is very difficult to solve optimization-type problems, determining the relative worth of one solution over another (via some fitness measure) appears, in general, to be a much simpler endeavor. This paper has developed proof-of-concept on a relatively simple problem.

There are many fruitful areas for future research including active learning of the training set, testing the hybrid on feedback problems, using other search techniques (other evolutionary approaches, simulated annealing, etc.), using other function approximation techniques (machine learning, fuzzy logic, etc.), the trade-off between a training set with few optimal examples vs. one with many sub-optimal examples, and determining better stopping criteria for the evolution and for the NN training. Also, application to real-world problems is an important next step. Unfortunately, ascertaining what percentage of optimum the approximate solutions achieve will not be possible (since the underlying function describing the system to be optimized will be unknown); however, a measure of performance can be obtained by comparing this approach with current methods for solving these problems.

REFERENCES

[1] Caudell, T. P. and Dolan, C. P., “Parametric Connectivity: Training of Constrained Networks using Genetic Algorithms”, *Proceedings of the Third International Conference on Genetic Algorithms*, 1989.

[2] Cohn, David, Atlas, Les and Ladner, Richard, “Improving generalization with active learning” *Machine Learning*, 15:201-21, May 1994.

[3] Goldberg, D. E., *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Publishing, 1989.

[4] Goonatilake, Suran and Khebbal, Sukhdev (eds.), *Intelligent Hybrid Systems*, John Wiley & Sons, 1995.

[5] Harp, S. A., Samad, T., and Guha, A., “Designing Application-Specific Neural Networks Using the Genetic Algorithm”, *NIPS-89 Proceedings*, 1990.

[6] Honavar, Vasant, and Uhr, Leonard (eds.), *Artificial Intelligence and Neural Networks: Steps Toward Principled Integration*, Academic Press, Inc., San Diego, California, 1994.

[7] Montana, D. J. and Davis, L., “Training Feedforward Neural Networks Using Genetic Algorithms”, *Proceedings of the Third International Conference on Genetic Algorithms*, 1989.

[8] Romaniuk, Steve G., “Evolutionary Growth Perceptrons”, *Genetic Algorithms: 5th International Conference (ICGA-93)*, S. Forrest (ed.), Morgan Kaufman, 1993.

[9] Rumelhart, David E., McClelland, James L. and the PDP Research Group, *Parallel Distributed Processing*, MIT Press, Massachusetts, 1988.

[10] Spears, W. M., Dejong, K. A., Baeck, T., Fogel, D., and de Garis, H., “An Overview of Evolutionary Computation”, *European Conference on Machine Learning (ECML-93)*, 1993.

[11] Sun, Ron and Bookman, Lawrence (eds.), *Computational Architectures Integrating Neural and Symbolic Processes: A Perspective on the State of the Art*, Kluwer Academic Publishers, Massachusetts, 1995.

[12] Ventura, Dan, Andersen, Tim, and Martinez, Tony R., "Using Evolutionary Computation to Generate Training Set Data for Neural Networks", *Proceedings of the International Conference on Neural Networks and Genetic Algorithms*, pp. 468-471, 1995.

[13] Wasserman, Philip D., *Advanced Methods in Neural Computing*, Van Nostrand Reinhold, New York, 1993.