Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms, pp. 468-71, 1995

USING EVOLUTIONARY COMPUTATION TO GENERATE TRAINING SET DATA FOR NEURAL NETWORKS†

Dan Ventura
Tim Andersen
Tony R. Martinez

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

Most neural networks require a set of training examples in order to attempt to approximate a problem function. For many real-world problems, however, such a set of examples is unavailable. Such a problem involving feedback optimization of a computer network routing system has motivated a general method of generating artificial training sets using evolutionary computation. This paper describes the method and demonstrates its utility by presenting promising results from applying it to an artificial problem similar to a real-world network routing optimization problem.

Introduction

Many inductive learning algorithms based on neural networks, machine learning, and other approaches have been developed and have been shown to perform reasonably well on a variety of problems [2][4]. Typically, neural networks (NN) perform inductive learning through the presentation of preclassified examples; however, one of the largest obstacles faced in applying these algorithms to real-world problems is the lack of such a set of training examples. Many times collecting data for a training set is the most difficult facet of a problem.

This paper presents such a real-world problem -- one for which no training data exists and for which gathering such data is at best extremely expensive both in time and in resources. To remedy the lack of training set data, a method using evolutionary computation (EC) [3][8] is described in which the survivors of the evolution become the training examples for a neural network. The synthesis of EC with NN provides both initial unsupervised random exploration of the solution space as well as supervised generalization on those initial solutions. Work involving a combination of EC and NN is becoming more prevalent; the reader is referred to [1][5][6][7] for examples.

† This research was funded in part by a grant from Novell, Inc.

Problem Description

Although we describe here a specific problem, the approach described in the next section is general enough to apply to a large class of optimization/feedback problems. We are specifically concerned, however, with a computer networking application and the constant setting/adjusting of certain control variables depending on the values of certain status variables. More formally, given a network, Θ, the state of Θ may be described at time t by a vector of status variables, $\boldsymbol{s}^t$. Control of the network is effected by the setting of variables in a control vector, $\boldsymbol{c}$. That is, given a network 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 network Θ' at time $t+\delta$ described by the vector $\boldsymbol{s}^{t+\delta}$. The problem is, given a status vector, $\boldsymbol{s}^t$, what modifications should be made to the control vector $\boldsymbol{c}$ such that $\boldsymbol{s}^{t+\delta}$ describes a better network, if possible, than $\boldsymbol{s}^t$? Here, the best network is defined as the network that maximizes some function of its status variables, $f(\boldsymbol{s})$. The operation of Θ is continuous and feedback from Θ in the form of $\boldsymbol{s}$ is continuous as well. A neural network is expected to monitor the values of $\boldsymbol{s}$ either continuously or periodically and to update the values of $\boldsymbol{c}$, the goal being to maximize the performance of Θ over time. In this particular problem 21 status variables (e.g. AvgRndTripTime, TotalPacketsSent, TotalPacketsRcvd, etc.) and 11 control variables (e.g. SendWinSize, RetransTimeInt, MaxRetransCnt, etc.) have been identified.

Generating the Training Set

Assuming that the status variables are defined over even a modest range, it is obvious that Θ may be described by any of an extremely large number of unique state vectors. In other words, the search space defined by the number of unique networks is enormous. Even if we limit ourselves for a moment to considering a single network by "freezing" the values of $\boldsymbol{s}$, we are still faced with a similar problem in choosing values for $\boldsymbol{c}$. With the intractable search spaces involved, evolutionary computation suggests itself for the exploration of those spaces. Actually, we limit ourselves to exploring, via evolutionary computation, the solution space for $\boldsymbol{c}$ only.

From the space defined by $\boldsymbol{s}$ that describes Θ we choose a representative set of network states by choosing n initial status vectors. We denote these $\boldsymbol{s}_i$, $0<i\leq n$ and refer to the network state described by $\boldsymbol{s}_i$ as Θ_i, $0<i\leq n$. These choices could of course be biased by any a priori heuristics as to what constitutes a realistic network. In choosing this set of status vectors, $\boldsymbol{s}_i$, $0<i\leq n$, we have chosen the left hand sides of our training instances. We now use evolutionary computation to discover “good” right hand sides yielding

training instances of the form $\boldsymbol{s}_i^{t=0} \rightarrow \boldsymbol{c}_k$.

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 each $\boldsymbol{c}_k$ for δ time steps, where δ time steps are sufficient for Θ_i to stabilize, and then applying fitness function f to $\boldsymbol{s}_i^{t=\delta}$. Next choose parents and use genetic operators to produce m offspring. Now evaluate the children and select m survivors from amongst the parents and children. The algorithm is sketched in figure 1.

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

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

Figure 1. Sketch of algorithm for evolving training set

Finally, choose j individuals from each of the n populations and build a set of jn training examples of the form $\boldsymbol{s}_i^{t=0} \rightarrow \boldsymbol{c}_k$. (To avoid ambiguity in the training set we could set $j=1$.)

We now return to the infinite space defined by $\boldsymbol{s}$. Since we have only chosen a finite number of seed points from this space, our evolutionary computation has found approximate solutions for only these n points in the space and can say nothing about any other points, many of which we are likely to encounter during normal execution of Θ. Therefore it becomes necessary to generalize on this relatively small set of approximate solutions. Using this set of approximate solutions as training examples, an NN model can be trained to develop a general hypothesis over the entire space defined by $\boldsymbol{s}$.

Simulation

In order to establish proof of concept, simulations using artificially generated problems were run. The simulation process includes the following steps:

1. Generate a problem definition
2. Create a training set using the algorithm
3. Train an NN with the training set
4. Create a test set
5. Test the NN network on the training set

An artificial problem generation/ simulation program 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) the quality of the EC-generated training set.

Generating a problem definition

In the networking problem, $\boldsymbol{s}$ is a vector of p status variables and $\boldsymbol{c}$ is a vector of q control variables. At any time t, $\boldsymbol{s}^t = g(\boldsymbol{s}^{t-\delta}, \boldsymbol{c}^{t-\delta})$, where g represents the operation of the network for time δ. Unfortunately, because of the network's asynchronous nature, it is impossible to determine the length of δ necessary for the network to stabilize or to accurately define g.

Therefore, we develop an artificial problem that is a simple analog of the network routing problem that preserves the relationship among $\boldsymbol{s}^t$, $\boldsymbol{s}^{t-1}$, and $\boldsymbol{c}^{t-1}$ but that operates in definite time steps and employs a simpler function H. Given a vector $\boldsymbol{s}$ of p integer variables (this vector is the analog of the status vector so we abuse notation and call it $\boldsymbol{s}$) and a vector $\boldsymbol{c}$ of q integer variables (the analog of the control vector), define the relationship $\boldsymbol{s}^t = H(\boldsymbol{s}^{t-1}, \boldsymbol{c}^{t-1})$ where H is in turn defined as a vector of functions $h_1, h_2, ..., h_p$. Now, to generate a problem, generate a $p\times q$ matrix G whose elements are randomly selected from {+,-}. The function h_i calculates the value of s_i^t from s_i^{t-1} and $\boldsymbol{c}^{t-1}$ as follows

$$s_i^t = h_i\left(s_i^{t-1}, \boldsymbol{c}^{t-1}\right) = \sum_{k=1}^{q} G_{ik}\left(s_i^{t-1} - c_k^{t-1}\right).$$

Thus, a problem is completely defined by H and G, which together represent a simple analog for the operation of a network. Once H and G have been defined, the combination of EC (to create a training set) and NN (to generalize on that training set) can attempt to learn the problem. To generate a new and different problem, randomly regenerate the matrix G; this allows for the generation of $\binom{2^q}{p}$ unique problems. Note that $\boldsymbol{s}_i^t$ ($\boldsymbol{c}_i^t$) has been used to indicate the *ith* vector at time t, while s_i^t (c_i^t) is now used to indicate the *ith element* of a vector $\boldsymbol{s}$ ($\boldsymbol{c}$) at time t, a slight abuse of notation.

As an example, suppose p=3 and q=3, that is, there are 3 "status" variables and 3 "control" variables. In order to generate a problem, randomly generate a 3×3 matrix G. One possibility is

$$G = \begin{bmatrix} + & - & - \\ + & + & + \\ - & + & - \end{bmatrix}$$

The system of equations that define the problem is then:

$$\begin{aligned} s_1^t &= +\left(s_1^{t-1} - c_1^{t-1}\right) - \left(s_1^{t-1} - c_2^{t-1}\right) - \left(s_1^{t-1} - c_3^{t-1}\right) \\ s_2^t &= +\left(s_2^{t-1} - c_1^{t-1}\right) + \left(s_2^{t-1} - c_2^{t-1}\right) + \left(s_2^{t-1} - c_3^{t-1}\right) \\ s_3^t &= -\left(s_3^{t-1} - c_1^{t-1}\right) + \left(s_3^{t-1} - c_2^{t-1}\right) - \left(s_3^{t-1} - c_3^{t-1}\right). \end{aligned}$$

Creating a training set

Once the problem has been defined, the evolutionary algorithm is employed to generate a training set. The function H and matrix G are substituted for the step "run $\Theta_i^{t=0}$ for δ time steps with control vector $\boldsymbol{c}$" in the *evaluate* procedure. The fitness function may be arbitrary, and the one used in these simulations was simply:

$$f(\boldsymbol{s}) = \sum_{i=0}^{p} s_i$$

All simulations used training sets of size 1000.

Training the NN

The PDP group's software implementation of the backpropagation algorithm[4] was used for all simulation results. For the simulations we defined 5 integer "control" variables and 10 integer "status" variables on the range [0, 100). Note that even though we limit the problem to integer values, the search spaces are extremely large: 100^5 for "control" vectors and 100^{10} for "status" vectors. The inputs to the backprop (the "status" variables) were binary encoded, while the outputs were simple localist nodes, one for each control variable. Fifteen hidden units were used so that the entire network consisted of 70 input nodes, 15 hidden units, and 5 output nodes. The output nodes produce values in the range [0, 1) and these are simply multiplied by 100 to produce a "control" vector in the desired range. All simulations were run with the default settings and training was only allowed to proceed for 500 epochs.

Creating a test set

The main advantage of having an explicitly defined function H representing our problem is that given a status vector $\boldsymbol{s}^t$, the optimal control vector $\boldsymbol{c}$ (the one that optimizes $\boldsymbol{s}^{t+1}$) can be calculated. The simplest method is by brute force, though this can be significantly improved in the case of this particular function H. To create a test set, randomly generate a vector $\boldsymbol{s}$ and then determine the optimal vector $\boldsymbol{c}$. Add the instance $\boldsymbol{s} \rightarrow \boldsymbol{c}$ to the training set and repeat for as many instances as desired. All simulations used a test set of size 50.

Testing the NN network

Typically, when evaluating how well the NN performs on a given training set we determine the number of correctly classified instances from the test set. However, because this is an optimization problem and because the output nodes are localist and expected to produce outputs in the range [0, 1) (rather that just being active or not), the NN is not expected to produce the optimum vector $\boldsymbol{c}_{opt}$ but rather a good approximation of it, $\boldsymbol{c}_{approx}$. Thus, given a status vector $\boldsymbol{s}$ the NN generates $\boldsymbol{c}_{approx}$, while $\boldsymbol{c}_{opt}$ and $\boldsymbol{c}_{worst}$ (needed for normalization) may be found by brute force (or some more efficient method). A measure of the correctness (normalized % of optimum) of $\boldsymbol{c}_{approx}$ is

$$\frac{f(s_{approx}) - f(s_{worst})}{f(s_{opt}) - f(s_{worst})}$$

where

$s_{approx}=H(\boldsymbol{s},\boldsymbol{c}_{approx})$,
$s_{worst}=H(\boldsymbol{s},\boldsymbol{c}_{worst})$, and
$s_{opt}=H(\boldsymbol{s},\boldsymbol{c}_{opt})$.

Empirical Results

The simulation process as described in the previous section was performed for genetic population sizes (the upper bound m in the *evolve* algorithm) of 20, 100, and 200. Ten different simulations were run for each value of m and the results are reported in table 1 The % of optimum is the correctness measure discussed above where $\boldsymbol{c}_{approx}$ is produced by either the GA (training set) or the NN (test set). The average pss is the average sum-squared error per pattern.

Table 1. Simulation results for different population sizes

	Population Size (m)					
	20		100		200	
	mean	sd	mean	sd	mean	sd
% opt. (train)	73.8	8.1	87.6	5.5	89.9	4.6
% opt. (test)	70.8	10.3	80.9	6.8	85.1	8.6
avg pss (train)	.237	.037	.128	.023	.103	.023
avg pss (test)	.667	.145	.364	.102	.327	.102

A number of interesting observations may be made from the table. First, the values for percent of optimum for the training set indicate that the GA is finding artificial instances that seem reasonable in the sense that they are a significant percentage of optimum. Second, the values for percent of optimum for the test set indicate that the training set produced by the GA enabled the NN to learn the problem fairly well, since it performs almost as well as the GA. Even more significant is the fact that as we increase the population size, both the GA and the NN perform closer to optimum with smaller standard deviations. Also, as the population size is increased, the sum-squared error on both the training set and test set decreases indicating that the training set more closely represents the true function.

Conclusion

Empirical simulation results suggest that at least for some problems, evolutionary computation is indeed capable of generating a training set that closely represents the actual underlying function. 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 the networking problem *f* would attempt to maximize throughput while minimizing resource usage). This paper has developed proof-of-concept on a relatively simple problem. Current research involves application of this method to the real network optimization problem and other difficult real-world applications. Unfortunately, ascertaining what percentage of optimum the approximate solutions achieve will not be possible (since the function *g* will be unknown). A measure of performance will have to be obtained by comparing this approach with current methods for solving these problems. As a final note, although generating the training set and training the NN is potentially time consuming, this need be done only once (or very infrequently), and it is conceivable that further training could be ongoing in the background using results obtained during execution. The actual execution would be extremely fast.

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] Falhman, S. E. and Lebiere, F., "The Cascade-Correlation Learning Architecture", Advances in Neural Information Processing 2, D. S. Touretzky (ed.), Morgan Kaufman, 1990.

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

[4] McClelland, James L. and Rumelhart, David E., Explorations in Parallel Distributed Processing, MIT Press, Cambridge, Massachusetts, 1988.

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

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

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

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