Neighbor Annealing for Neural Network Training

V. Scott Gordon

Abstract—**An extremely simple technique for training the weights of a feedforward multilayer neural network is described and tested. The method, dubbed "neighbor annealing" is a simple random walk through weight space with a gradually decreasing step size. The approach is compared against backpropagation and particle swarm optimization on a variety of training tasks. Neighbor annealing is shown to perform as well or better on the test suite, and is also shown to have pragmatic advantages.**

1. Introduction

TRAINING a standard feedforward neural network in a supervised manner requires finding a set of weights that allows the network to replicate the input/output pairs in a training set. Backpropagation is a commonly-used and method, and there are others.

Backpropagation works by computing the output error, and then reducing that error by gradient descent [1]. It has certain pragmatic and performance limitations, fueling interest in alternative methods. Researchers have tested a variety of search techniques in place of backpropagation, such as genetic algorithms and particle swarm optimization (PSO). These methods can sometimes have advantages over backpropagation with respect to flexibility and/or performance.

This paper describes an extremely simple technique for training the weights of a neural network, dubbed *neighbor annealing*. It is a sort of simplified version of simulated annealing with variable step size [2]. First results are shown on a variety of problems, and its efficacy is compared with backpropagation and PSO. Other advantages are discussed.

II. Neighbor Annealing

Neighbor annealing is a simple random walk through weight space with a gradually decreasing step size. It is, in a sense, a variant of simulated annealing. Both share an annealing schedule, and both utilize a single point in space from which a neighbor is randomly selected. However, they differ with respect to where their annealing is applied. In simulated annealing, a neighbor is chosen in a fixed manner, and then replaces the current point if it is a better solution, or probabilistically if the temperature is sufficiently high.

Manuscript received November 30, 2007.

V. Scott Gordon is with the Computer Science Department at California State University, Sacramento, CA 95819 USA (email: gordonvs@ecs.csus.edu).

By contrast, neighbor annealing applies an annealing schedule to the neighborhood size. At the early stages, the neighborhood size is large enough to encompass the entire problem domain. At each iteration, neighborhood size is decreased, effectively limiting how far of a random jump is allowed. Eventually, when the neighborhood size reaches some predetermined minimum value (varies depending on the range of domain values), the search process stops. Also unlike simulated annealing, a jump to a lower fitness is never made. The basic steps are shown below in Algorithm 1:

Algorithm 1 – Neighbor Annealing (basic)

1. select a random point P in *domain(F)*.
2. repeat:
   (a) select a random neighbor R in S, where S is the set of points in $domain(F) \cap (P-T \ldots P+T)$.
   (b) if $fitness(R) > fitness(P)$, replace P with R.
   (c) decrease T according to annealing schedule.

where:
F is the function being optimized,
P and R are points in *domain(F)*,
fitness(x) is a measure of the quality of $F(x)$, and the vector T is neighborhood size expressed as temperature.

Like simulated annealing, the process reduces to a sort of hill-climbing at the later stages.

A. Neighbor Annealing for Neural Networks

Neighbor annealing can be applied to a variety of optimization tasks. This paper focuses on its application for tuning neural network weights.

Adapting Algorithm 1 to training neural network weights is straightforward. P and R represent floating-point weight vectors. T is a floating point scalar, initialized to a relatively large value for weight changes (typically 1.0). Since the early random steps are large, it is sufficient to initialize the weights in P to *zero*. *Fitness(x)* is calculated by first assigning the values of the weight vector (R) into a neural network, then running a standard forward pass for the entire training set in batch mode, and finally computing some measure of the resulting success or failure, such as accumulating a sum-squared error. A simple annealing schedule could be to simply multiply T by a value close to (but less than) 1.0,

such as 0.9999. There is no backward pass. The entire training process is shown below in Algorithm 2:

ALGORITHM 2 – NN TRAINING W/ NEIGHBOR ANNEALING

1. Initialize *P* to *[0.0, 0.0, ... 0.0]*
   Initialize *Sched < 1* (for example, *Sched=0.9999*)
   Initialize *T to 1.0*
   Initialize *best_fitness = infinity*
   Rand = uniform random.
   Training data consists of pairs of inputs *I* and associated desired outputs *D*.
2. repeat
   (a) initialize *error* to 0
   (b) *R = P + [Rand(-T…T), rand(-T…T)…]*
   (c) assign weight vector *R* to a neural network
   (d) for each training pair (I_x, D_x):
      (i) apply I_x to inputs
      (ii) calculate outputs O_x (forward pass)
      (iii) *error* += $(D_x - O_x)^2$
   (e) if *error < best_fitness*, replace *P* with *R*
   (f) *T = T * Sched*
   until *T < .0001*

Upon completion, *P* contains the trained weights.

B. Flexibility of Neighbor Annealing

Algorithm 2 (above) outlines a typical example of neighbor annealing for neural network weight training. But there is some flexibility for adapting training to the goals of the network. There are many parameter options:

1) Annealing schedule

Although our first experiments utilized a simple multiplication annealing schedule, other schedules (such as linear or Boltzmann) could of course be utilized.

2) Initial Step Size

While the value shown (1.0) seems to work for many problems, for some problems larger values appear to be effective, sometimes even as high as 20.0 or 30.0. Similarly, the terminating criteria can sometimes be set to occur sooner, such as at 0.1 or even 0.2.

3) Fitness computation and Error term

The method shown above corresponds to that used in backpropagation – namely, it computes the sum of the squares of the output errors (desired minus actual). However, there are times when it is useful to base the fitness on other factors. For example, one could instead count the number of training cases that were output correctly. In another study, the author utilizes a self-splitting network [3], and in that setting an appropriate fitness measure would be to count the largest chunk of contiguous solutions found. Unlike backpropagation, neighbor annealing (like genetic algorithms and PSO) does not require any gradient information – for that matter, error information plays no role whatsoever in assigning the next set of weights to be considered. Therefore, any desired fitness measure can be used.

4) Artificial neurons

This study utilized networks built using standard artificial neurons. That is, they compute a weighted sum, and pass the sum through the standard logistic function. Other units could be used instead, such as perceptrons, because backpropagation's reliance on a differentiable squashing function does not apply to neighbor annealing.

III. EXPERIMENTAL METHOD

The shear simplicity of neighbor annealing for neural network weight training is appealing, and the pragmatic advantages outlined in 2.2 have been described. However, for it to be practically useful it would of course need to show promise in solving problems competitively with other training methods.

For this first study, three weight training methods were implemented and compared: (1) backpropagation, (2) particle swarm optimization, and (3) neighbor annealing. The settings of each are now described.

A. Backpropagation settings

Except where noted, we used the backpropagation settings shown in Fig. 1:

- α *(learning rate) = 0.3*
- μ *(momentum rate) = 0.8*
- *Training / test data normalized to (0.1…0.9)*
- *Threshold units (output=1) used at every layer*

Fig. 1. backpropagation settings

B. Particle Swarm Optimization settings

Particle Swarm Optimization (PSO), introduced by Kennedy and Eberhart [4], has gained attention lately as an effective network training technique. PSO is based loosely on the concept of the clustering of species, such as the flocking of birds or the swarming of bees. This study utilized the basic PSO formulae for the changes in position and velocity shown in Fig. 2.

PSO can be used to train a neural network by assigning each particle to a separate neural network, specifically by including in each particle a complete set of weights for a possible neural network solution, where each weight is a dimension of the particle. PSO neural

network training, like neighbor annealing, utilizes a fitness function. As described earlier, for this study a particle's fitness is calculated by computing the output of the network for every point in the training set, and computing the sum of squares of the resulting errors.

$$\Delta V_i = wV_i + c_1 r_1 \bullet (max_i - P_i) + c_2 r_2 \bullet (max_g - P_i)$$
$$P_i \text{ (new)} = P_i \text{ (old)} + V_i$$

where:
Vi = velocity vector for particle i
Pi = position vector for particle i
r_1 r_2 = uniform random vectors, in (0..1)
c_1 c_2 = guiding constants (set to 2.0 in this study)
w = inertial constant (set to 0.5 in this study)
max_i = best position of particle i, so far
max_g = best position of any particle, so far

Fig. 2. PSO Training Algorithm [4]

PSO has a few other settings that must be chosen, such as the maximum allowable values for position and velocity, and the population size. These vary from problem to problem, and are listed later.

C. Normalizing Performance Measures

For each test problem, an attempt is made to compare the algorithms fairly, by *normalizing* the amount of work they each do in order to reasonably assess their relative efficiency and effectiveness. To this end, a normalized interpretation of a ***work epoch*** is utilized. An epoch is defined as a single set of passes through the entire training data set. The following adjustments are made to ensure that the three algorithms are assessed for a roughly equivalent amount of work:

1) for neighbor annealing

Each batch pass to evaluate a weight vector is considered a work epoch.

2) for backpropagation

Since processing is done for each training pair separately, the total number of iterations is divided by the training set size. Then, the result is multiplied by 2, since there is an additional backward pass for each training pair. The resulting value is the number of work epochs.

3) for particle swarm

Since each pass is applied to a swarm of weight vectors, the total number of iterations is multipled by the swarm size. The result is the number of work epochs.

In addition, other steps are taken to ensure fair comparison: (1) training methods are always compared on equivalent network topologies, (2) criteria settings are equivalent for each algorithm on any given problem, and (3) the same training and testing data files are used for each algorithm on any given problem.

In some cases it was difficult to ensure that the exact same number of epochs occurred in each algorithm. This is because the annealing schedule as defined does not incorporate a setting for the total number of iterations. However, it was possible to adjust the annealing parameters to result in reasonably similar values.

IV. Test Problems

We used five problems of varying complexity, including two relatively simple problems, one problem of moderate difficulty, and two relatively hard problems. Three of the problems were obtained from a variety of sources, and the others were hand-constructed:

Problem	Source	# inputs / # outputs	# training cases	# testing cases
XOR	n/a	2 - 1	4	-
BEAM	[5]	4 - 1	10	6
XOR regions	n/a	2 - 1	100	16
Snowplow	[6]	5 - 1	1024	-
2-Spiral	[8]	2 - 1	1058	1056

Fig. 3. Test Problem Details

The *XOR* problem is a simple truth table, but the outputs are in floating point space. The *XOR regions* problem extends the XOR problem to four quadrants of floating-point input/output in 2d space. The *BEAM* problem was taken from Adeli and Hung [5]. The *Snowplow Driving* problem was generated for the Olsen-Darter study [6] using a complex set of nonlinear computations described by Gabibulayev et. al [7], and consists of training data only. The Two-Spiral problem was generated using CMU's benchmark data set generation software [8].

For each test problem, we chose *training criteria* (permissible errors for outputs during training), *testing criteria* (permissible errors for outputs during testing), and *network topology* (number of nodes in each network layer, not including threshold units). The values we chose are shown below, in Fig. 4.

Descriptions of each problem are shown in Fig. 5. They are each taken (or paraphrased) from descriptions given in their source material, if any.

Problem	Topology	Training Criteria	Testing Criteria	other settings
XOR	2-2-1	0.4	-	Sched = 0.999 initial temp = 1.0 Swarm size = 20 pMin/Max = +/-5
BEAM	4-4-1	0.02	0.04	Sched = 0.999 initial temp = 1.0 Swarm size = 10 pMin/Max=+/-5
XOR regions	2-4-1	0.3	0.3	Sched = 0.99999 initial temp = 0.5 Swarm size = 50 pMin/Max=+/-50 $\mu = 0$
Snow Plow	5-5-1	2.0	-	Sched = 0.9999 initial temp = 1.0 Swarm size = 20 pMin/Max=+/-20
2-Spiral	2-5-1	0.4	0.4	Sched=0.9999 initial temp = 0.5 Swarm size = 20 pMin/Max=+/-50 $\mu = 0.8$

Fig. 4. Test Problem Details

XOR – Trains the network to correctly compute the exclusive-OR of its two inputs.

BEAM – Trains the network to compute the minimum weight steel beam for a given loading condition [5].

XOR regions – Trains the network to learn:

```
F(x,y) =
0 when x in [0..1] and y in [0..1],
1 when x in (1..2] and y in [0..1],
1 when x in [0..1] and y in (1..2],
0 when x in (1..2] and y in (1..2]
```

Snowplow Driving – Trains the network to predict the location of a vehicle 3 seconds into the future, given a set of sensor values [6].

Two Spirals – Trains the network to distinguish on which of two intertwined spirals a point lays [8].

Fig. 5. Test Problem Descriptions

V. Results

All of the neural networks were able to successfully learn the training sets of *XOR, XOR regions,* and *BEAM*. Those results are shown in Fig. 6, which compares how quickly each algorithm was able to learn the training data, and also how well each generalized on the test data.

None of the algorithms were able to completely learn either the *Snowplow* or the *2-Spiral* problems. Those results are shown in Fig. 7, which compares to what degree each algorithm was able to solve each problem, for roughly equivalent numbers of epochs.

In each table, *generalization* is the percentage of (untrained) testing cases that produced outputs within the testing criteria, after training was complete.

Problem	Backprop • Train epochs • Residual error • Generalization	PSO • Train epochs • Residual error • Generalization	Neighbor Annealing • Training epochs • Residual error • Generalization
XOR	484 0.43 ---	720 0.26 ---	299 0.33 ---
BEAM	2630 0.005 83%	580 .005 66%	788 .035 100%
XOR regions	358,756 1.95 100%	250,000 1.4 100%	318,983 1.87 100%

Fig. 6. Results - 1

Problem	Backprop • Train epochs • Residual Error • % Trained • Generalization	PSO • Train epochs • Residual Error • % Trained • Generalization	Neighbor Annealing • Training epochs • Residual Error • % Trained • Generalization
Snow Plow	97,656 0.28 96% ---	100,000 .85 84% ---	92,099 0.50 90% ---
Two Spiral	189,035 201.8 6% 6%	100,000 200.0 26% 26%	85,000 202.0 27% 27%

Fig. 7. Results - 2

First results for neighbor annealing on this test suite look very promising. It is certainly competitive, actually outperforming both backpropagation and PSO in most cases. Generalization is as good or better than backpropagation and PSO in every case. In one problem (XOR regions), PSO displays the fastest learning.

Some additional ad-hoc experimenting was done using neighbor annealing, substituting ***number of cases solved*** for the fitness function (instead of sum-squared error). This produced interesting results on the harder functions, shown below in Fig. 8. In these examples, the network topology was expanded to 2 hidden layers.

Problem	Neighbor Annealing 2-4-4-1 • Training epochs • % Trained	
XOR regions	5000 84%	103,287 100%
Snow Plow	1000 34%	
2 Spiral	1000 56%	10,000 61%

Fig. 8. Additional Results

The solution for XOR regions is notably better with these settings, even better than the previous PSO results. And, while the 2-spiral problem is still far from being solved, a much larger portion of it has been trained. In all three cases, significant chunks of the problem have been solved very quickly. Although this is of marginal interest in isoloation, it could be very useful if a divide-and-conquer strategy were being employed [3].

VI. Conclusions

A simple optimization technique called neighbor annealing was described, and then shown to be applicable to neural network weight training. Its characteristics were outlined, and shown to have benefits in terms of flexibility and adaptability to a variety of neural network settings. Its performance was compared against backpropagation and particle swarm optimization on a suite of five problems, and it was shown to be competitive with those existing methods on the suite.

Neighbor Annealing is so simple that it could be characterized as trivial, or even primitive. That it should work rather well without tuning or modification, leads us to believe that it merits further study.

VII. Future Work

Further experiments are needed on a wider variety of difficult training sets. Also, more testing is needed using alternative fitness measures and annealing schedules. It would also be natural to consider more sophisticated methods for adjusting the neighborhood size, such as that used in the variable step size GSA (VSGSA) developed by Sutter and Kalivas [2].

We have begun a study combining the neighbor annealing presented here with the self-splitting network described in [3]. The flexibility of neighbor annealing, specifically for encouraging the identification of solved chunks, seems well-suited for this framework.

References

[1] S. Haykin, *Neural Networks, a Comprehensive Foundation.* Prentice-Hall 1994.

[2] J. M. Sutter and J. H. Kalivas, *Convergence of generalized simulated annealing with Variable Step Size with Application Toward Parameter Estimations of Linear and Nonlinear Models*, Analytical Chemistry, 63 (1991) 2383-2386.

[3] V. Gordon and J. Crouson, *Self-Splitting Modular Network – Domain Partitioning at Boundaries of Trained Regions*, to appear in the International Joint Conference on Neural Networks – IJCNN 2008.

[4] J. Kennedy and R. Eberhart, *Swarm Intelligence.* Morgan-Kaufmann 2001.

[5] H. Adeli and S. Hung, *Machine Learning – Neural Networks, Genetic Algorithms, and Fuzzy Systems*, John Wiley & Sons, 1995

[6] M. Olsen-Darter and V. Gordon, *Vehicle Steering Control Using Modular Neural Networks*, IEEE International Conference on Information Reuse and Integration (IRI), pp 374-379, 2005.

[7] M. Gabibulayev, B. Ravani, and T. Lasky, *Stochastic Modeling for Lateral Control in Snowplow Operations*, 9th World Congress on Intelligent Transport Systems, Chicago, IL, October 2002.

[8] M. White, *Two-Spirals Benchmark Data Set Generator*, 26 April 2006:http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/areas/neural/bench/cmu/bench.tgz