ESANN'1993 proceedings - European Symposium on Artificial Neural Networks
Brussels (Belgium), 7-8-9 April 1993, D-Facto public., ISBN 2-9600049-0-6, pp. 57-63

A learning and pruning algorithm for genetic boolean neural networks

Frédéric GRUAU

CENG/DRFMC/SP2M/PSC, BP85X 38041 Grenoble and
LIP-IMAG, URA CNRS, ENS-Lyon 46 Allee d'Italie 69007 Lyon

Abstract. The Genetic Algorithm (GA)can be used to find neural networks. In this paper, we present a learning algorithm designed to train boolean neural networks produced by the GA, using a cellular encoding of the neural network. The algorithm looks for weights in the range $\{-1, 0, 1\}$. The aim of learning is to speed-up the GA. Learning is fast but not 100% successful. Experiments report learning of the parity and the symmetry boolean functions.

1 Introduction

Genetic Algorithms can be applied to the problem of finding neural network architectures, or, in other words, deciding how many hidden units a network should have and how these units should be connected. In this approach, the central problem is how to encode a neural architecture on codes that can be manipulated by the GA. Typical encodings assume that there is a bound on the number of hidden units; the genetic algorithm can then be used to determine a neural network architecture within a finite range, that yield improved computational behavior. These directly coded network architectures have usually been trained using back-propagation. These ideas are implemented in [7, 9, 5] . Still, the problem of directly optimizing a network architecture is the highest cost of each architecture evaluation. If we must run a back-propagation algorithm for each evaluation, the time consumed for number of evaluations typically needed to find improved network architectures quickly becomes computationally prohibitive. The computation cost is typically so high as to make genetic algorithms impractical except for optimizing small topologies.

There are new directions being explored for optimizing neural network architectures such as grammar based architecture descriptions [6, 8] that may display better scaling properties. In our approach the genetic algorithm is used to develop both the architecture and the boolean weights for neural networks that learn boolean functions. Instead of coding these architectures directly, a *grammar tree* is used which is both more compact and more flexible than direct codings or previously developed grammatical representations. This representation is called *cellular encoding*. In [3], we used a genetic algorithm to recombine and mutate grammar trees and showed that neural networks for the parity problem and symmetry problem could be found. Furthermore, the grammar trees are recursive encodings that allow for whole families of related networks that compute parity or symmetry. In this way, once small parity or symmetry problems are solved, the grammars can generate solutions to parity and symmetry problems of arbitrary size. Gruau and Whitley [4] showed that it is

ESANN'1993 proceedings - European Symposium on Artificial Neural Networks
Brussels (Belgium), 7-8-9 April 1993, D-Facto public., ISBN 2-9600049-0-6, pp. 57-63

possible to get a speed up by combining the GA with a neural network based learning. Their study focused on how to use the learned information, rather than on the learning itself. They used a hebbian based learning for only one epoch, that flips no more than 2 or 3 weights. In [2], we describe a learning that fits networks generated by the genetic algorithm using cellular encoding. This learning algorithm looks for boolean weights (± 1). Choosing weights within $\{-1, 1\}$ is very restrictive. In this paper, we extend the learning described in [2] so as to choose weights in $\{-1, 0, 1\}$. Setting a weight to 0 amounts to prune the link. Finally, we report experiments with the target boolean functions parity and symmetry.

2 Specification of the learning algorithm

The proposed learning algorithm is designed to train networks generated by the GA using the cellular encoding. Therefore it has some specific requirements. The GA produces networks with integer biases and boolean weights. We do not learn the bias because the cellular encoding of a bias is compact. There is no need to specify a link number, which is the case if we want to encode a weight-change. Hence the GA needs less effort to encode the right biases than the right weights. The neuron's sigmoïd is the step function. A neuron n computes its net input which is the weighted sum of the neighbour's activities. If the net input is lower or equal to zero, the neuron's activity is set to 0, else it is set to 1. The GA generates thousands of networks, so learning needs not be 100% successful on each network. On the other hand learning must be fast, because the time of a GA run will be the learning time taken for one neural net multiplied by thousands. The purpose of the learning is to refine a neural net produced by the GA. It should change correctly a few weights in order to increase the performance of the network. It may find the global optimum, if the neural net before training is close to it. The networks generated by the GA using cellular encoding have some particularities. The neurons are sparsely connected, usually the fan-in is 2 or 3. The number of layers is high, so a standard back-propagation would fail. Most of the weights are 1, only a few weights are -1. This is because weights are set to 1 by default, and only negative weight values must be encoded. The learning algorithm should take advantage of these particularities.

3 The Switch learning Algorithm

We now describe the learning algorithm. We call it the "Switch Algorithm". At each epoch, all the patterns to be learned are forwarded through the network. After each pattern is passed through, we update two positive variables for each neuron n: r_n and w_n and six positive variables for each link l: $r_l[v]$ and $w_l[v]$, $v \in \{-1, 0, 1\}$ In fact, we do not consider $r_l[v_0]$ and $w_l[v_0]$, where v_0 is the value of the current weight. The letter r stands for right and w stands for wrong. Let o be an output unit, if the activity of o is the desired output, r_o is set to 1 and w_o is set to 0, otherwise, r_o is set to 0 and w_o is set to 1. For a hidden unit h the variables r_h and w_h are computed so as to give a hint about the correctness of h's computed activity. If r_h is high, the activity of h is right, a modification of h's activity is likely to decrease the performance of the net on the currently processed pattern. If w_h is high, the

ESANN'1993 proceedings - European Symposium on Artificial Neural Networks
Brussels (Belgium), 7-8-9 April 1993, D-Facto public., ISBN 2-9600049-0-6, pp. 57-63

```
Let c be the neuron currently processed;
oni:=the old net input of c;
For each neuron n that fans into c through link l
     If (l's weight is not null)
          Try to modify n's activity;
     nni = the new net input of c; d=Abs(nni-oni)/2;
     Switch(Comp(nni,oni))
     Case 0 : if(Small(oni)) r_n += r_c;
              else r_n += r_c * p1
              w_n += w_n + w_c;
     Case + : r_n += w_c * p2;
     Case - : w_n += w_c * p3 * d; r_n += r_c * p4 * d;
 If (n is an input unit) P5=1; Else P5=1/p5;
 If (n's activity is not null)
   For all v in {-1,0,1}, v not equal to l's initial weight
     Try to set l's weight to v;
     nni = the new net input of c; d=Abs(nni-oni)/2;
     Switch(Comp(nni,oni))
     Case 0 : if (Small(oni)) r_l[v] += r_c * P5;
              else r_l[v] += r_c * p1 * P5
              w_l[v] += w_c;
     Case + : r_l[v] += w_c * p2 * P5;
     Case - : w_l[v] += w_c * p3 * d * P5; r_l[v] += r_c * p4 * d * P5;
 Elseif n is not input unit
   For all v in {-1,0,1}, v not equal to l'sinitial weight
     Try to both set l's weight to v and modifyn's activity;
     nni = the new net input of c; d=Abs(nni-oni)/2;
     Switch(Comp(nni,oni))
     Case 0 : if(Small(oni)) r_n += r_c * p6; r_l[v] += r_c * P5 * p6;
          else r_n += r_c * p1 * p6; r_l[v] += r_c * p1 * P5 * p6;
          w_n += w_c * p6; w_l[v] += w_c * p6;
     Case + : r_n += w_c * p2 * p6; r_l[v] += w_c * p2 * P5 * p6;
     Case - : w_n += w_c * p3 * d * p6; w_l[v] += w_c * p3 * d * P5 * p6;
              r_n += r_c * p4 * d * p6; r_l[v] += r_c * p4 * d * P5 * p6;
```

Fig. 1. The Learning Algorithm. Function Comp returns 0 if its arguments are of oposite sign else it return the sign of the difference beetween the absolute values ot its first and second argument. Function Small returns 1 if its argument is 0 or 1, and 0 otherwise. The operator += increments the variable of the left side by the quantity in the right side.

activity of h could be wrong, a modification of h's activity is likely to increase the performance of the net. Similarly, for each link l, if $r_l[v]$ (resp. $w_l[v]$) is high, setting l's weight to v is likely to decrease (resp. increase) the performance of the net on the currently processed pattern.

In order to compute the value of all these variables we process the neurons one by one, backwards, starting from the output-units, as in back-propagation. The algorithm sketch in figure 1 shows how to process one neuron. Let c be the currently processed neuron. There are three possible trials to modify c's activity: We can either change a weight on a link l that fans into c (trial t_1) or modify the activity

ESANN'1993 proceedings - European Symposium on Artificial Neural Networks
Brussels (Belgium), 7-8-9 April 1993, D-Facto public., ISBN 2-9600049-0-6, pp. 57-63

of a neighbour n that fans into c (trial t_2) or else do trial t_1 and t_2 at the same time (trial t_3). For each neighbour n that fans into c through link l, the learning algorithm successively tries t_1, and one among t_2 or t_3 depending on the activity of n. For each trial t_i it analyses the effect of the modification on the neuron c. If the effect is desirable, it increases w_n if t_i has modified n'activity and $w_l[v]$ if t_i has set l's weight to v. If the effect is bad, it increases r_n if t_i has modified n's activity and $r_l[v]$ if t_i has set l's weight to v. There are three kind of effects, following the change of the net input of c which is the weighted sum of the neuron's activity that fans into c, minus c's threshold.

- e_1: The net input of c changes its sign (0 has the sign -). Thus c's activity changes. If $w_c > 0$ we wanted c's activity change with a strength w_c. We register the fact that the trial t_i is worth with a strength w_c. We add w_c to w_n or $w_l[v]$ depending on t_i. If $r_c > 0$ we did not want c's activity to change with a strength r_c. We register the fact that t_i is bad with a strength r_c. We add r_c to r_n or to $r_l[v]$ depending on t_i.
- e_2: The net input increases in absolute value. If we wanted c's activity to change, this effect is bad because the change will be harder. Thus we add w_c to r_n or $r_l[v]$, depending on t_i.
- e_3: The net input decreases in absolute value. If we wanted c's activity change, the effect is desirable since the change will be easier. We add w_c to w_n or $w_l[v]$ depending on t_i. If we did not want c's activity to change the effect is bad because c's activity is less stable and will be more easily changed. We add r_c to r_n or to $r_l[v]$ depending on t_i.

Until now, the Switch Algorithm looks like a standard back-propagation. However there is a difference. The quantities r_c and w_c that are back propagated, are multiplied by a coefficient which is used to rank the relative importance of the corresponding change in a weight value or an activity value. These coefficient are produced using 6 parameters p_i, $i = 1, \ldots, 6; 0 < p_i < 1$. Each of these parameter, corresponds to a particular heuristic. Parameter p_1 promote the change of the activity of a neuron having a small stability. This kind of strategy has already been used in [1]. It has theoretical and experimental foundations. Parameters p_2, p_3 and p_4 rank the importance of the effect e_1, e_2, and the two cases of e_3. We prefer to try to learn new patterns rather than consolidate old patterns. This heuristic can be implemented by choosing $p_3 >> p_4$. An input unit's activity is fixed. Therefore we can better evaluate the effect of a weight change on a link that fans out an input unit. Parameter p_5 is used to register this fact. The trial t_3 allows to register an information that is relevant only if we make at least two weight changes, one to change the weight of the link l that fans in c, another changes a weight upstream the neuron n connected by l in order to modify n's activity. On the other hand, the information computed using t_1 or t_2 is relevant for one weight change. Since the information computed during trial t_3 is less precise, we multiply the back propagated quantities by p_6 in t_3. The effect e_3 is more important if the change in the net input is 2, than if it is 1. Therefore, we multiply the backpropagated quantities by $\delta/2$ where δ is the absolute value of the difference between the net input before the trial and the net input after the trial. The algorithm in fig 1 can be implemented with 8

ESANN'1993 proceedings - European Symposium on Artificial Neural Networks
Brussels (Belgium), 7-8-9 April 1993, D-Facto public., ISBN 2-9600049-0-6, pp. 57-63

multiplications, 7 additions and 4 conditional branching for processing one link, by computing the coefficients in advance, for each possible c's net input, l's weight and n's activity.

Once all the patterns have been processed, the $w_l[v]$ and $r_l[v]$ corresponding to each pattern are summed. A fitness $f_l[v]$ is computed for each link l, $v \in \{-1, 0, 1\} \setminus \{v_0\}$, where v_0 refers to the current weight. We use the formula: $f_l[v] = w * w_l[v] - r * r_l[v] - d * d_l$. The positive coefficients r, w and d parametrize the learning. The fitness $f_l[v]$ measures how interesting would be to set l's weight to the value v. The variable d_l refers to the depth of the link l which is the length of the maximum path from this link to an output unit. The fitness of a link f_l is the maximun of $f_l[v]$, The two links in the whole network that have the lower fitness are selected. One of these two links is chosen for a weight change. The one with the highest fitness is chosen with a probability p higher than 0.5. The weight of the selected link is set to v where $f_l[v]$ is the maximum fitness. After the weight change, the performance of the neural net with the modified weight is computed. This performance is the total number of correct outputs divided by the number of patterns divided by the number of output units. If this performance increases, we accept the weight change, if not, we still accept it with a probability $\exp(-\Delta/T)$. T is a temperature, Δ is the decrease in the performance. From one epoch to the other T is decreased, in order to do simulated annealing. If the weight change is accepted, then the weight of the link that has been selected for change is frozen for e epochs where $e = e_1 + e_2 * e_3^{d_l}$, e_1, e_2 e_3 are positive parameters. Due to the term $e_2 * e_3^{d_l}$ in this expression, the links near the output units will be frozen a longer time. This counter balanced the fact that these links are more likely to be selected, due to the term $-d * d_l$ that appears in the computation of the fitness f_l. Thus, the algorithm starts by modifying weights of units near the output units, which is better because the information is more relevant there. It then carries on by modifying weights of deeper units.

4 Experiments

We now describe three sets of experiments during which we try to learn 2 boolean functions: the parity and the symmetry. For each target function, we consider a genetic code found by hand that produces a family of neural networks such that some links must be pruned by learning in order to find a solution. We tested the Switch algorithm on the first three neural networks of the family. Networks developed with cellular encoding are a relevant benchmark, since the learning algorithm is designed to be combined with cellular encoding. We represent the third network of the family, with a set of correct weights found by the learning in fig 2. We use two modes for the weight initialization: In the first mode we set all the initial weights to 1, in the other mode, we set the weights with a random value ± 1. For the symmetry we set the parameters to $p_1 = 0.3$, $p_2 = 0.5$, $p_3 = 0.04$, $p_4 = 0.006$, $p_5 = 0.17$, $p_6 == 0.02$, $r = 5$, $w = 7$, $p = 0.8$, $d = 0.5$, $e_1 = 2$, $e_2 = 1$, $e_3 = 0.7$. We keep the same parameters for the parity exept for $w = 6$, $d = 2$, $p_1 = 0.7$. The initial temperature is set to 1 and we divide it by 1.01 after each epoch. We run 32 trials for each experiment, and compute the average and standard deviation of the number of epochs needed to reach the solution. In order to obtain an estimation of the standard deviation

ESANN'1993 proceedings - European Symposium on Artificial Neural Networks
Brussels (Belgium), 7-8-9 April 1993, D-Facto public., ISBN 2-9600049-0-6, pp. 57-63

of the average, the standard deviation of the result must be divided by $\sqrt{32}$. We want learning to be fast and we do not require 100% of success. We therefore allow only 100 epochs. In the following table the rows report in this order, the target application, the mode of weight initialization, the network number which is L, the number of successful trials, the average number of epochs of the successful trials and the standard deviation of the number of epochs.

target	parity						symmetry					
weights	one			random			one			random		
L	1	2	3	1	2	3	1	2	3	1	2	3
success	32	32	32	32	21	7	32	32	31	32	15	8
epoch	3	6	17	8	22	24	3	7	16	16	41	34
SD	1	1	7	6	24	22	0	4	12	14	22	26

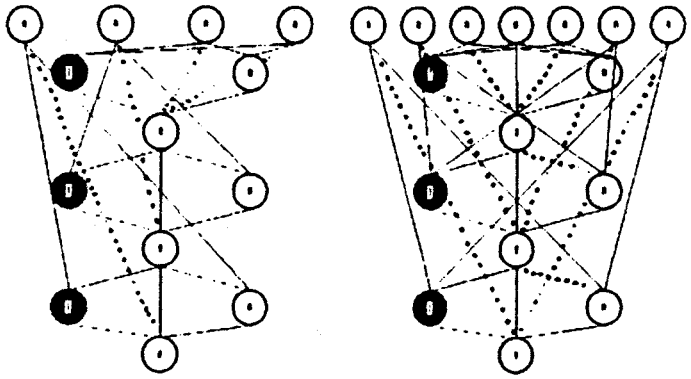

Fig. 2. The third learned network for the parity and the symmetry. Circles represent neurons. If the threshold is 0, the circle is empty, if it is one the circle is filled with black. A continuous line indicates a weight 1, a dashed line a weight -1 and a dotted line a null weight.

In the first set of experiments the target is the parity function. The parity of $l+1$ binary inputs returns the sum of its input modulo 2. We used a code found by hand that develops a network family (N_l) such that (N_l) has $l+1$ input unit, $3*l$ hidden units, $8*l$ connections, $2*l$ layers, In the next set of experiments we try to learn the symmetry function. The symmetry of $2l+1$ binary inputs returns 1 if and only if the input is symmetric and the middle bit is 1. We choose a code that develops a network family (N_l) with the following features: Network (N_l) has $2l+1$ input units, $3*l$ hidden units, $11*l$ connections, $2*l$ layers.

The experiments show that starting with initial weights of 1 helps the Switch Algorithm. When there are a lot of -1 weights, as with random initialization, the neuron's activity have a greater probability to be zero. When too many activities are 0, the Switch Algorithm is not able to retrieve useful information.

ESANN'1993 proceedings - European Symposium on Artificial Neural Networks
Brussels (Belgium), 7-8-9 April 1993, D-Facto public., ISBN 2-9600049-0-6, pp. 57-63

5 Conclusion

In this paper, we present a learning algorithm called "Switch Algorithm" that has been designed to refine neural networks produced by the Genetic Algorithm, with the so-called cellular encoding. During one epoch, the Switch Algorithm guesses the best weight to change, and also the best value into which to change it. This value can be $-1, 0$ or 1. This learning suits to the particular class of neural nets that are generated by the GA. These neural nets are made of units having a small fan in, and therefore, the number of connections is also small. They have a great number of weights 1, and many layers (up to 6 in our experiments). Moreover, the Switch Algorithm fulfills some extra requirements imposed by the GA. It is quick but not 100% successful. Experiment shows that a success rate above 20% is achieved with an average number of epochs under 40, on networks having up to 8 hidden units. For a complete validation, we must conduct a test with the GA and measure the speed up brought by learning with the Switch Algorithm.

References

1. M. B. Gordon, P. Peretto, and D. Berchier. Learning algorithms for perceptrons from statistical physics. *Journal de Physique*, january 1993.
2. F. Gruau. A learning algorithm for genetic neural networks. submitted to IWANN93.
3. F. Gruau. Genetic synthesis of boolean neural networks with a cell rewriting developmental process. In *Combination of Genetic Algorithms and Neural Networks*, 1992.
4. F. Gruau and D. Whitley. Adding learning to the the cellular developmental process: a comparative study. Submitted to Evolutionnary Computation, 1992.
5. S. Harp, T. Samad, and A. Guha. Toward the genetic synthesis of neural networks. In D. J. Schaffer, editor, *3rd Intern. Conf. on Genetic Algorithms*, pages 360–369, 1989.
6. H. Kitano. Designing neural network using genetic algorithm with graph generation system. *Complex Systems*, 4:461–476, 1992.
7. G. Miller, P. Todd, and S. Hedge. Designing neural networks using genetic algorithm. In D. J. Schaffer, editor, *3rd Intern. Conf. on Genetic Algorithms*, pages 379–384, 1989.
8. Eric Mjolness, David Sharp, and Bradley Alpert. Scaling, machine learning and genetic neural nets. La-ur-88-142, Los Alamos National Laboratory, 1988.
9. D. Whitley, T. Stakweather, and C. Bogart. Genetic algorithms and neural networks, optimizing connection and connectivity. *Parallel Computing*, 14:347–361, 1990.