Training Product Unit Neural Networks with Genetic Algorithms

D. J. Janson and J. F. Frenzel
Department of Electrical Engineering
email: jff@k2.ee.uidaho.edu
email: djanson@bashful.ee.uidaho.edu
and D. C. Thelen
Microelectronics Research Center
email: dthelen@granite.mrc.uidaho.edu
University of Idaho, Moscow, Idaho 83843

Abstract- **This paper discusses the training of product neural networks using genetic algorithms. Two unusual neural network techniques are combined; product units are employed instead of the traditional summing units and genetic algorithms train the network rather than backpropagation. As an example, a neural network is trained to calculate the optimum width of transistors in a CMOS switch. It is shown how local minima affect the performance of a genetic algorithm, and one method of overcoming this is presented.**

1 Introduction

Neural networks have been applied successfully to many problems in recent years. Traditionally these networks are composed of multiple layers of summation units. These simple units sum their inputs, each input multiplied by a variable weight. This summation is usually then squashed by a non-linear equation such as the logistic function. Several researchers have shown that networks composed of these units can calculate any function to any arbitrary degree of accuracy given enough summation units. [1] However, there are many functions that are complicated enough that the number of summation units it takes to duplicate them are prohibitive. One very commonly found task is that of higher order combinations of the inputs such as either $X * X$ or $X * Y$.

One proposed solution is a new unit called the "sigma-pi unit" [3]. This unit not only applies a weight to each input, but also applies a weight to the second and possibly higher order products of the inputs. While much more powerful than the traditional summation unit, the number of weights increase very rapidly with the number of inputs, and soon become unmanageable when applied to solving large problems. Since most problems only need one, or at most a few, of these terms, the sigma-pi unit is overkill.

1.1 Product Units

A suitable alternative was introduced by Durbin and Rumelhart [2]. The "product unit" computes the product of its inputs, each raised to a variable power. This is shown in

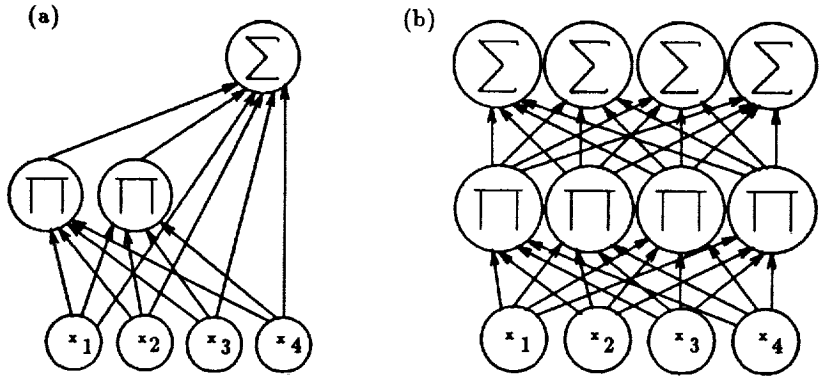

Figure 1: Recommended product network configurations [4]

Equation 1.

$$y = \prod_{i=1}^{N} X(i)^{p(i)} \tag{1}$$

The $p(i)$ term is treated in the same way as the variable weights for summation units. Using the modified version of backpropagation presented by Durbin and Rumelhart, these product units can provide much more generality than sigma-pi units. While a sigma-pi unit is constrained to using just polynomial terms, the product units can use fractional and even negative terms. As Durbin and Rumelhart point out, product units can actually be considered a superset of sigma-pi units; for if several of the product units are used, and they are constrained to only integer values, they would have the same results.

There are many ways that product units can be used in a network. However, the overhead required to raise an arbitrary base to an arbitrary power makes it unlikely that they will replace summation units. Durbin and Rumelhart propose that the primary use of the product units will be to supplement the power of the summation units. Two proposed architectures are shown in Figure 1. The term product neural networks (or product networks) will be used to refer to networks containing both product and summation units.

While product units increase the capability of a neural network, they also add complications. Not only is backpropagation harder to accomplish, but the solution space becomes more convoluted. As Durbin and Rumelhart pointed out, there are often local minima that trap the network. As a possible solution to this problem, this paper investigates the use of genetic algorithms to train product networks.

2 Genetic Algorithms

2.1 Introduction

A genetic algorithm (GA) is an exploratory procedure that is able to locate near-optimal solutions to complex problems. To do this, it maintains a set (called a population) of trial solutions (called chromosomes). Through a repeated four-step process, these chromosomes

evolve until an acceptable solution is found. These steps are evaluation, reproduction, breeding, and mutation. A representation for possible solutions must first be developed. Then, with an initial random population, the GA is able to solve the problem almost without regard to the interpretation of the chromosome. Each generation, the chromosomes produced, through survival-of-the-fittest and exploitation of old knowledge in the gene pool, should have an improved ability to solve the problem.

There were two primary reasons why GAs were applied to training product networks. First was that the addition of product nodes made the solution space more complicated. Because backpropagation is a gradient decent algorithm, it is very likely to get caught in a local minimum. The second reason was that backpropagation tends to be slow with complicated problems. It often takes many iterations to train a complicated network. It is hoped that the use of a GA will find the best answer much faster than backpropagation.

2.2 Representation

Before applying a genetic algorithm to any task, a representation for possible solutions must be found. The most common method for representing these possible solutions is with a bit string. Higher order strings (such as character strings) or trees (such as binary trees) have also been used. Since the architecture of the product networks to be trained will be known, a binary string representation with a fixed number of bits per weight can be constructed. Thus, each weight in the network has a certain number of bits associated with it. This representation permits each chromosome to be decoded easily, while still allowing each weight a large degree of freedom. The typical generation used had between 30 to 100 members in a population, with 16 bits representing a weight.

2.3 Evaluation

The first step in any generation is the evaluation of the current chromosomes. This is the only step where the interpretation of the chromosome is used. Each chromosome in the population is decoded, and the result is used to solve the original problem. This solution is then graded on how well it solved the problem. The method used to grade product networks is to calculate the sum of squared error (SSE) for the training set. The fitness of the chromosome is equal to $1/(1 + \text{SSE})$. This means that the better a network performs, the higher its fitness, with a perfect network having a fitness of 1.

2.4 Reproduction

The next step in a generation is to create a new population based upon the evaluation of the previous one. Every chromosome generates a specific number of copies of itself, based on how well it solved the problem. Thus the chromosomes that performed better will produce several copies of themselves, while the worst chromosomes won't produce any copies. This is the step that allows GAs to take advantage of a survival-of-the-fittest strategy.

There are several methods to calculate the number of offspring that each chromosome will have. One of the more prevalent methods is called ratioing. With ratioing, each chromosome produces a number of offspring directly related to its fitness, with the only restriction being that the total number of chromosomes per generation remains constant. Thus, if one chromosome has a fitness that is twice that of another, then the superior chromosome would produce twice as many offspring. However, there are two major problems with this method. First, if all the chromosomes have similar fitness, each member in the population will produce one offspring. This results in little pressure toward improving the solution. The second problem, although from a different source, has the same effect. If any one chromosome should happen to have a fitness much larger than any of the others, then that chromosome would create most, if not all of the new offspring. This discriminates against the remaining information of the gene pool in favor of this super-chromosome, loosing the information in the gene-pool. This particular type of stagnation has been labeled premature convergence.

The method the author used to train the product networks is ranking [5]. In ranking, the whole population is sorted by fitness. The number of offspring each chromosome will generate is then determined by where it falls in the population. The ranking algorithm used was that the top 30% of the population generated two offspring each, the bottom 30% of the population generated no offspring, and the rest of the population each generated one offspring. In this way, no one chromosome can overpower the population in a single generation, and no matter how close the actual fitness values are, there is always constant pressure to improve. While the problem of premature convergence still exists, it is greatly reduced by allowing other chromosomes a chance to mix information with high fitness chromosomes. The disadvantage of using ranking is speed. In not allowing better chromosomes to guide the population easily, good answers are slower to develop.

2.5 Breeding

The previous step, reproduction, created a population whose members currently best solve the problem. However, many of the chromosomes are identical and none are different than those in the previous generation. Breeding combines chromosomes from the population and produces new chromosomes that, while they did not exist in the previous generation, maintain the same gene pool. In natural evolution, breeding and reproduction are the same step, but in GAs they have been separated to allow different methods for each to be experimented with and independently evaluated. It is in this step where GAs can exploit knowledge of the gene pool by allowing good chromosomes to combine with chromosomes that aren't as good. This is based on the assumption that each individual, no matter how good it is, doesn't contain the answer to the problem. The answer is contained in the population as a whole, and only by combining chromosomes will the correct answer be found.

There are many methods used for breeding; with the most common being crossover. Crossover typically takes two chromosomes and swaps parts of each to create two new chromosomes. Many variations on crossover have been used, but no results have shown

before	after
001100	000010
110011	111101
↑ ↑	↑ ↑

Figure 2: Example of two-point crossover (crossover points indicated by arrows)

which is decisively better. The crossover the author used to train the product networks was a simple two-point crossover. Two random points are chosen in the chromosome, and the bitstring between the two points is swapped between the two chromosomes. An example is shown in Figure 2.

2.6 Mutation

The last step in creating a new generation is based on the assumption that while each generation is better than the previous, the individuals that die may have some information that is essential to the solution. It is also possible that the initial population didn't have all the necessary information. The reinjection of information into the population is called mutation. Again, there are many ways to implement mutation, but essentially all choose and change members of the population randomly.

The method the author used was to simply inject a constant number of mutations every generation. The number of mutations used was approximately 0.25% of the total number of bits in the entire population. These mutations where then randomly distributed among all the bits, with each bit having the same chance of mutating. A mutation involved a 50/50 chance of setting the bit to a 1 or 0, in effect giving the mutated bit a 50/50 chance of changing. This means that any specific chromosome may or may not mutate, with a small chance that it could severely mutate.

2.7 An Application

A product network was trained that calculates the optimum width of the transistors in a CMOS switch given temperature, power supply voltage, and minimum conductance as inputs. While there are many excellent analysis tools available, such as circuit simulators, there are almost no software packages available that transform performance specifications into a circuit schematic. This network is designed as an aid to CMOS circuit designers, and was first proposed by Thelen in [4].

The data used to train the network was extracted from several SPICE simulations with differing transistor dimensions, temperatures, and power supply voltages. In the training set created from this data, the voltages ranged from 3 to 12 volts, the temperature from 303 to 403 °K, and the transistor width from 2 to 20 micrometers. Using these inputs, the conductance could range from approximately 1 to 500 micro-mhos. Two hundred data points were collected and a sample from these points is shown in Table 1.

Voltage	Temperature (°K)	Conductance	Desired Width
3	303	1.026E-6	2
3	303	3.806E-6	3
3	303	6.593E-6	4
3	303	1.204E-5	6
3	303	1.752E-5	8
3	303	2.851E-5	12
3	303	3.951E-5	16
3	303	6.152E-5	24

Table 1: Sample from the data points used to train the network

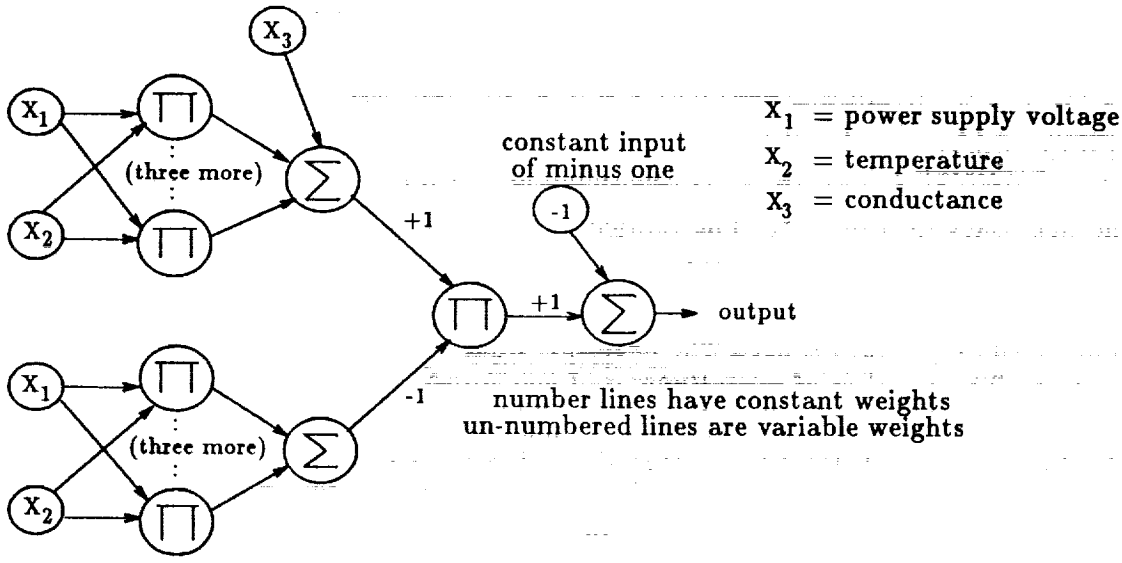

Figure 3: The product neural network trained to select the width of a CMOS switch

The configuration of the product network was designed by Thelen using *a priori* information about the equations to model a CMOS switch. The layout of the network is shown in Figure 3.

3 Results

The first attempts at training the product network had very consistent, but wrong, results. Through many runs of the GA, every solution represented a network that gave outputs of approximately ten for the transistor width, with no regard for the input.

The first success came when the population was seeded with an approximation to the solution. This approximation was derived by a curve- fitting program using the training data. When seeded, the GA was able to quickly improve the approximation and find a

network that gave the desired output. While seeding did indeed allow an answer to be found, it was desired that the GA could find an answer using an initial random population.

Better success was found using a penalty function. Penalty functions decrease the fitness of a chromosome by adding constrictions to the solution. The penalty subtracted from the fitness of a chromosome dependent upon how close the output of two consecutive data points were. The closer the two outputs for the two points were, the larger the penalty. With the addition of this penalty function, the GA was able to find a solution for the network given an initial random solution.

4 Discussion

The initial results were very surprising. The inability of the GA to find an appropriate solution meant that either the network could not solve the problem, or that the real solution to the problem was extremely difficult to find. Previous work by Thelan showed that indeed a solution to this problem did exist. This meant that the real solution must be difficult for the GA to find. In fact, when seeding the GA with approximate solutions, an answer was found.

There are three ways to make a problem difficult for a GA to solve. Either the solution space is extremely convoluted, the best solution occupies a very small portion of the solution space, or the solution space is misleading to a GA. Since proving whether a GA is being mislead is very difficult, the other two possibilities where considered. Comparing the solutions found by the GA showed that they converged to the same answer each time. Thus, the solution space was not too convoluted for the GA to search.

The correct solution was not found with an initial random population. However, it was found with the insertion of the penalty function. (The effect of the penalty function was to place a pole in the middle of the unwanted solution, thus allowing the GA to continue searching the space, and find the correct solution.) This leads the authors to believe that the right answer occupied a very small portion of the solution space, allowing the GA to more easily find the undesired answer.

This example points out one common problem with GAs. In using GAs, often the solution space is not very well known, and suboptimal answers can often dominate the solution space. Indeed, if the problem to be solved is incorrectly or incompletely represented, the GA will take advantage of these mistakes, and produce wrong answers.

5 Conclusion

It has been shown that product networks can be successfully trained with Genetic Algorithms. A product network has been trained to give the width of CMOS switch, given power supply voltage, temperature and minimum conductance specifications for the switch. Further research will be done to compare the use of GAs to backpropagation in product networks. Also, the capabilities of product networks will be compared to traditional neural networks. While product units have been shown to have superior capabilities over

traditional summation units, almost no studies to compare different networks have been done.

References

[1] G. Cybenko. Continuous valued neural networks with two hidden layers are sufficient. Technical report, Department of Computer Science, Tufts University, Medford, MA, 1989.

[2] R. Durbin and D. Rummelhart. Product units: A computationally powerful and biologically plausible extension to backpropagation networks. *Neural Computation, Vol 1*, pages 133 – 142, 1989.

[3] D.E. Rumelhart, G.E. Hinton, and J.L. McClelland. *Parallel Distributed Processing 1*, Chapter 8: Learning Internal Representations by Error Propagation, pages 318 – 362. Cambridge, MA, and London: MIT Press, 1986.

[4] D. Thelen. A neural network for designing CMOS switches: an application for product units. In *JWSUUIISCNC '91: Proceedings of the 1991 Joint WSU/UI Interstate Student Conference on Neural Computation*, pages 100 – 109. Jack Medor, EE Dept, Washington State University, April 1991.

[5] D. Whitley. Selective pressure and ranked based allocation. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 116 – 123. Lawrence Erlbaum Associates, Inc., Hillsdale NJ, 1989.

This research was supported by NASA under Space Engineering Research Center Grant NAGW-1406