R. Reed and R.J. Marks II, "Genetic Algorithms and Neural Networks: An Introduction", Northcon/92 Conference Record,
(Western Periodicals Co., Ventura, CA), Seattle WA, October 19-21, 1992, pp.293-301

Northcon®/92

CONFERENCE RECORD

SEATTLE, WASHINGTON
OCTOBER 19-21, 1992

EDITED BY:
Hardev Juj
Tacoma City Light
Technical Conference Chairman

Alvin Todd Moser
Seattle University
Technical Conference Vice-Chairman

SPONSORED BY:

Seattle and Oregon Sections,
Institute of Electrical and Electronics Engineers (IEEE)

the Cascade Chapter
Electronics Representatives Association (ERA)

and the Electronics Manufacturers Association

Genetic Algorithms and Neural Networks: An Introduction

Russell Reed, Robert J. Marks II
Dept. of Electrical Engineering, FT-10
University of Washington, Seattle, WA, 98195

Abstract

The Genetic Algorithm (GA) is a general optimization/search method that has been successfully applied to many problems —including neural network design. Unlike some other methods, it works well on large dimensional, nonlinear, and noisy problems. This paper reviews the basic algorithm and discusses its application to neural networks.

One of the basic tasks in neural network design is to choose an appropriate architecture and weights in order to solve a particular problem. The Genetic Algorithm (GA) is a general optimization/search method that has been successfully applied to many problems —including neural network training. It is appropriate for neural networks since it scales well to large nonlinear problems.

As in the theory of natural evolution of species, a population of many candidate solutions compete for resources; the most successful survive and reproduce, the least successful reproduce less often and eventually become extinct. Since successful units pass on their characteristics to the next generation at a higher rate than less successful units, the average fitness of the population tends to increase over many generations and approach an optimum.

The principle advantage of the algorithm is that very little problem–specific information is needed. The algorithm itself simply operates on bit strings containing the genetic code. To apply it to a specific problem, all that is needed is a function to evaluate the solutions encoded in the bit strings and return a score indicating the quality of the solution. In particular, it doesn't need gradient information and so may be used on discontinuous functions and functions which are described empirically rather than analytically. It can also be used for temporal learning problems in which reinforcement comes at the end of a long sequence of actions with no intermediate target values. Since it isn't a simple hill–climbing method, it isn't particularly bothered by local maxima. It will also tolerate a certain amount of noise in the evaluation function.

The algorithm has some of the flavor of simulated annealing in that many possible solutions are examined and the search has an element of randomness which helps to avoid the problem of local maxima. It differs in that many candidate solutions are maintained rather than just one, and elements of the better solutions are combined to generate new candidates. Like simulated annealing, it is a general optimization/search method and not limited just to neural network problems.

The principle disadvantage of the method is the amount of processing needed to evaluate and store a large population of candidate solutions and converge to an optimum.

1 The Algorithm

The basic operations are (1) selection based on fitness, (2) recombination of genetic material by crossover, and (3) mutation. The algorithm operates on a population of many units. Each unit has a bit string —its genetic code —that encodes its solution to the given problem. A problem–specific function decodes the bit string, evaluates the solution, and returns a score which is translated into a fitness score. Units are selected for mating in proportion to their fitness scores and pass on their characteristics to the next generation. Since the offspring of successful parents tend to displace less successful individuals and also tend to be successful in turn, the average fitness of the population tends to increase over many generations and approach an optimum.

The algorithm starts with an initial population of N units with random parameters encoded in a binary bit string. Larger population sizes generally make the algorithm more likely to find a good

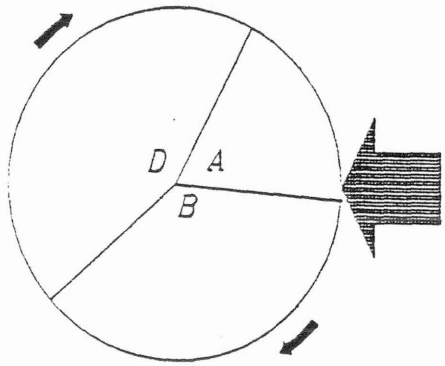

Figure 1: Selection. Units are selected for reproduction with probability proportional to their fitness. Units with higher fitness (corrsponding to more slots on the roulette wheel) are more likely to be chosen than units with lower fitness, but all units have some chance of being chosen.

solution, but, of course, require more processing time. The following steps are repeated until a solution is found.

Evaluation. Evaluate each unit and assign it a non-negative score (higher=better). Normalize by dividing by the sum of all scores to obtain fitness scores f_i in the range 0–1. If any unit satisfies the goal criteria, discard the other units and stop.

Reproduction. On N trials, select an individual with probability f_i and copy it to the mating population. Units can be selected with probability f_i in the following way. Assign to each unit a segment of the interval 0–1 proportional to its fitness. If, for example, there are three units with normalized fitness scores $f_1 = 0.1$, $f_2 = 0.7$, and $f_3 = 0.3$, then assign f_1 the interval 0–0.1, f_2 the interval 0.1–0.7, and f_2 the interval 0.7–1.0. Then choose a random number between 0 and 1; if it falls in the ith interval, then select unit i. Fig. 1 illustrates this by analogy to a roulette wheel where each unit has a number of slots proportional to its fitness.

Because of the element of chance, the number of times a unit reproduces will not be exactly proportional to its fitness, but, on average, if unit i has twice the fitness of unit j, then it will usually have about twice the offspring. Units with very low fitness ratings will rarely reproduce and face extinction.

Crossover. Divide the mating population into pairs and mix their genetic codes by crossing the bit strings at one or two random points. Fig. 1 illustrates the operation. If units A and B have parameter strings 110100111001 and 011011100101, for example , then they would produce offspring 011100111001 and 110011100101 if the crossing point is after position 3. The probability that crossover will occur is set by p_c; for $p_c < 1$, there is some chance that the parents simply survive in the next generation unaltered by crossover. This helps to preserve good solutions since, if a particular solution is good, some copies of it are likely to survive unchanged. Typical values are $p_c = 0.6$ to 0.9.

Mutation. For each unit in the new set, flip each bit with some small probability; e.g. $p_m \leq 0.001$. The number of mutations should be small; otherwise the algorithm deteriorates into random search. Its main purpose is to maintain diversity in the population. In general, p_m should be chosen so that mutations occur in only a very small percentage of the population —in only one or two units for moderately sized populations.

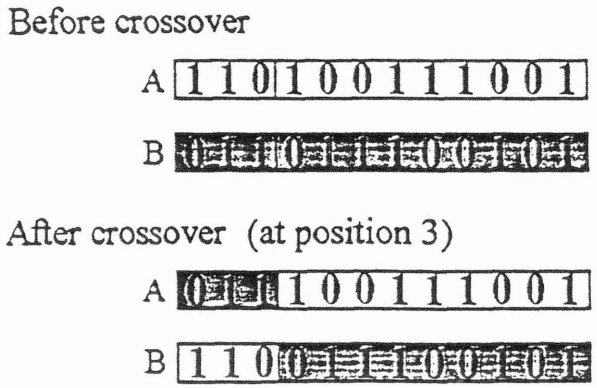

Figure 2: Crossover. Crossover mixes genetic codes inherited from two parents by crossing the bit strings at one or two random points. The bit strings encode characteristics of the parents so the offspring receives traits of both parents, but is not identical to either. The crossing point is random, so the mix of characteristics transferred varies with each mating.

1.1 Variations

Many variations of the algorithm have been proposed. In the basic algorithm, all units reproduce and large portions of the parameter strings are exchanged during reproduction so it is possible for good solutions to be lost. One remedy is to allow only the most successful fraction of the population to mate, with their offspring replacing the less successful part of the population. Since the offspring do not replace their parents, this helps to preserve good solutions.

Other variations extend the natural evolution analogy by incorporating features such as paired chromosomes (diploidy), dominance, inversion, niche specialization, etc. Some versions are Lamarkian, allowing adaptations made in the lifetime of a parent to be passed on to the offspring. Some vary the number of units that reproduce at each cycle. Some allow the population size to flucuate and some maintain several subpopulations with only limited mixing. Goldberg [12] reviews many of these cases.

1.2 Schemata

The core idea of the algorithm is that if a string contains a combination of bits, say $011****01***$, that are strongly correlated with good solutions, the string is likely to be reproduced in the next generation. A particular template of 1's, 0's and *'s (don't cares) in the bit string, e.g., $011****01***$, is called a *schema*.

1.3 The Effect of Crossover

Crossover is responsible for most of the adaptive power of the algorithm. Random crossover during mating mixes bit strings from both parents and and produces offspring that have characteristics of both parents, but which are not identical to either. Hopefully the offspring will inherit useful bit combinations from both parents and be better than either.

The *defining length* of a schema is distance between its most separated defining bits. The distance between the leading 0 and the final 1 of $011****01***$, for example, is 8 bits. If a schema has a long defining length (if it contains significant bits on both ends of the string, for example), it is likely to be broken during crossover. Thus, schemata with short defining lengths are more likely to survive than

longer ones. This tends to make the algorithm favor low order, less complex, solutions over high order ones —usually a desirable feature for a learning algorithm.

1.4 The Effect of Mutation

Mutation plays a rather small part in the algorithm. If the mutation rate is too large, the algorithm tends to degenerate into an inefficient random search. When all the units are very similar, however, as in the final stages of convergence, crossover creates few new solutions and mutation becomes important.

Since all defining bits of a schema must survive mutation for the schema to survive, schemata with fewer defining bits are more likely to survive than those with many defining bits. This also favors robust solutions. In a physical system, for example, small amounts of noise or parameter variation would be less likely to disturb a low order schema.

The combination of fitness selection, crossover, and mutation favors schemata with above average fitness, short defining length, and low order.

1.5 Fitness Scaling

Since the grading function can be arbitrarily chosen, it is useful to scale the raw scores to obtain the fitness scores. If all units receive scores in the range from 1000 to 1005, for example, the best solutions would have very little advantage over the worst units and the search will be essentially random. This might occur in late stages of the algorithm when many units are clustered around a good solution. Likewise, in the early stages, most of the units might have low scores; if some unit makes a significant (but not decisive) improvement and gets a much higher score, it would dominate the next generation, resulting in a premature loss of genetic diversity. This effect is especially important when populations are small.

An appropriate scaling can help avoid these problems. A linear transformation is often used to map the raw scores f to fitness values f'

$$f' = af + b.$$

In choosing a and b, it is desirable that $f_{avg} = f'_{avg}$ so that one expects each average unit to produce one offspring. The number of offspring for the best unit is controlled by ensuring $f'_{max} = C_{mult} f_{avg}$, where C_{mult} is the desired number of offspring for the best unit. For small populations ($n = 50$ to 100), values of $C_{mult} = 1.2$ to 2 are suggested [12]. If this scaling results in negative scores, set them to 0.

Other methods of fitness scaling are discussed in [12].

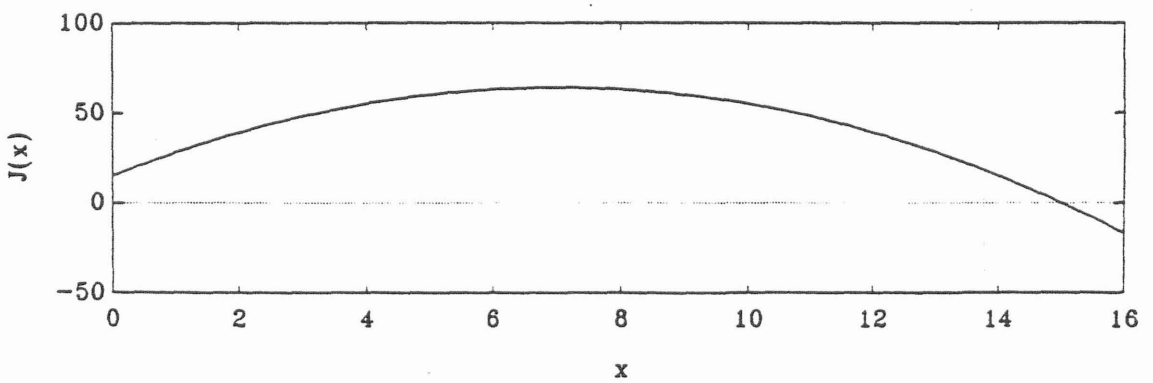

Figure 3: The function considered in the example.

2 Example

Fig. 3 illustrates the function

$$J(x) = 64 - (x - 7)^2.$$

for $0 \leq x < 16$. Let the population consist of four units A, B, C, and D with solutions x encoded in 4–bit strings.

The units are initialized to random values for the first generation.

Generation 1				
unit	x		J	f_i
A	(1)	0 0 0 1	28	0.1854
B	(9)	1 0 0 1	60	0.3973
C	(15)	1 1 1 1	0	0
D	(6)	0 1 1 0	63	0.4742

After random selection weighted by fitness the population is A, B, D, D. Unit C with fitness 0 has died and unit D, with the highest fitness, is selected twice.

New Population	
unit	
A	0 0 0 1
B	1 0 0 1
D	0 1 1 0
D	0 1 1 0

A mates to D and B mates to D, both with crossover after the 3rd bit

Mating results	
unit	
a	0 0 0 0
b	1 0 0 0
c	0 1 1 1
d	0 1 1 1

Mutation flips the 3rd bit in a.

Mutation results	
unit	
a	0 0 1 0
b	1 0 0 0
c	0 1 1 1
d	0 1 1 1

The resulting population after one generation is

Generation 2				
unit	x		J	f_i
a	(2)	0 0 1 0	39	0.1696
b	(8)	1 0 0 0	63	0.2739
c	(7)	0 1 1 1	64	0.2782
d	(7)	0 1 1 1	64	0.2782

The average score has gone from 37.75 to 57.75.

3 Application to Neural Network Design

The Genetic Algorithm is not specific to neural networks; it is a general optimization/search method that can be applied to neural networks in a number of ways —from simply determining a few weights in a predetermined network to choosing the entire architecture: the number of layers and nodes, connections, weight values and node functions. Since it doesn't need gradient information, it can be used on networks with binary units and/or quantized weights.

Several things should be considered in applying the algorithm to neural networks. One of the more important is the representation —how the problem parameters are encoded in the bit string. Since neural networks often have many weights, the string can be long. Since crossover breaks the bit string, it is helpful to put related parameters close together in the bit string as much possible.

A similar problem is that the weights tend to be closely related, with the values of weights in the final layers depending on the values of many weights in preceeding layers. This means that the useful schemata often have high order and are easily broken by mutation and crossover. This interdependence leads some [9] to eliminate the crossover operation; this is not typical however.

The following paragraphs describe some recent applications of the algorithm to neural network design.

LISP Representation

Koza and Rice [19] describe an interesting use of the algorithm to determine both the weights and connection archictecture of a neural net. The network architecture is encoded in a LISP expression as a tree structure describing the network and the crossover operation exchanges subtrees between two parents.

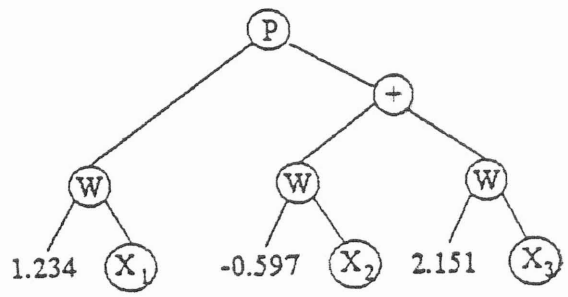

Figure 4: Representation of a neural network as a LISP expression. 'W' represents a multiplication operation by an input weight and 'P' represents a summation and the node nonlinearity.

Training and Evaluation

Keesing and Stork [16] include a small amount of training in the fitness evaluation function. Although one set of weights, A, could be worse that another set B, the set A might be much better than B after a small amount of training because of differences in the local terrain of the error space. In this case, unit A is closer to the solution than B even though it's initial performance is worse. Without training, the algorithm learns only the fitness of the initial point in the space. With training, each unit attempts to find the best spot reachable from its initial position and its fitness reflects the quality of a small area around the initial point so the algorithm searches more of the parameter space.

As in Darwinian evolution, the fitness of each unit is evaluated based on its performance after adaptation, but the original parameters are passed on during reproduction, rather than the adapted parameters. Since the performance of a unit (in a sense) reflects the nearness of good solutions, and

since reproduction will tend to produce more offspring near good units, the algorithm will tend to converge to good solutions.

Only a small amount of training is allowed, typically 5–10% of what would be required to train to completion. If all units were trained to convergence, they could converge to the same or equivalent solutions and all units would have the same reproductive fitness regardless of the quality of their genes. Keesing and Stork also found that evolution is fastest when each network undergoes a different number of training cycles. Over a number of generations, this effectively measures the sensitivity of the gene to learning and rewards units which improve quickly with a small amounts of additional learning.

Subpopulations

Isolation of subpopulations is one factor that contributes to the development of new species in nature. Whitley and Starkweather [27] find that using many subpopulations with limited mixing allows aggressive optimization within each subpopulation while preserving diversity in the total population. This is said to be a more effective in preserving diversity than simply increasing the population size. One–at–a–time reproduction is used with fitness based on rank. The offspring don't replace their parents; they replace the worst units so good solutions are certain to survive.

GA Pruning of Neural Networks

Whitley and Bogart [26] use the Genetic Algorithm to prune neural networks. The bit string contains a bit for each weight; the bit is 1 if the connection is retained and 0 if it is pruned. The result is networks that are smaller, learn faster, have a low variance in training time, and may generalize better.

GA to Select Bit String Representation

Since crossover tends to break the string near the middle (on the average), it is useful to put related parameters close together rather than at opposite ends of the string, for example. It is not always obvious, however, which representation is best. Marti [20, 21] considers the influence of the representation on a recurrent neural network and uses a two–level GA to optimize the representation. Populations of networks using different representations compete on the second level while, within each population, GA is used to find the optimum weights for the given representation.

GA for Selection of Features

The algorithm does not do away with the need for good data sets. In particular, if the training set is small, good generalization cannot be expected simply because a network has been found that does well on the training set. Chang and Lippmann [7] uses the algorithm to select features for (non–neural) classification systems. They also use the algorithm to reduce the number of reference patterns for a K–nearest–neighbor classifier without significantly affecting performance.

NN vs. GA: Incremental Change

Wieland [28] compares gradient descent and Genetic Algorithm methods to create recurrent networks for pole–balancing systems. Both methods are successful in finding solutions, but when small changes are made to the control task, such as increasing the length of the pole by a small amount, the gradient method adapts quickly while the Genetic Algorithm takes much longer; i.e., one dozen generations to adapt to a 1% change.

NN vs. GA: Scaling

Spears and De Jong [24] suggest that neural networks trained by gradient descent may be better (learn faster) for small problems while Genetic Algorithms may scale better for large ones.

4 Discussion

The Genetic Algorithm is general stochastic optimization/search method that has been used successfully in a number of ways for neural network design. Its main advantages are that it requires very little problem–specific information and it can escape from local maxima (minima). The algorithm itself is

very easy to apply and can be computed efficiently. Most of the time is usually spent in the problem–specific evaluation function.

Unlike some other stochastic search techniques, it doesn't require detailed problem–specific knowledge in order to generate new search candidates.

Its main disadvantage is the amount of processing required to evaluate and store a large number of different network configurations. It is worth noting, however, that the candidate solutions can be evaluated independently so that N parallel processors should give close to a factor of N reduction in computation time.

References

[1] D. H. Ackley. *A Connectionist Machine for Genetic Hillclimbing.* Kluwer Academic Publishers, 1987.

[2] D. H. Ackley and M. S. Littman. Learning from natural selection in an artificial environment. In *Proceedings of the International Joint Conference on Neural Networks*, page 189, Washington, DC, 1990. vol. I.

[3] L. Booker. Improving search in genetic algorithms. In L. Davis, editor, *Genetic Algorithms and Simulated Annealing*, Morgan Kaufmann, 1990.

[4] L. Booker. Using classifier systems to implement distributed representations. In *Proceedings of the International Joint Conference on Neural Networks*, page 39, Washington, DC, 1990. vol. I.

[5] L.B. Booker, D.E. Goldberg, and J.H. Holland. Classifier systems and genetic algorithms. In J. Carbonell, editor, *Machine Learning: Paradigms and Methods*, MIT Press, 1990.

[6] M. Caudill. Evolutionary neural networks. *AI Expert*, March 1991.

[7] E. I. Chang and R. P. Lippmann. Using genetic algorithms to improve pattern classification performance. In R.P. Lippmann, J.E. Moody, and D.S Touretzky, editors, *Advances in Neural Information Processing (3)*, pages 797–803, 1991. (Denver 1990).

[8] L. Davis. Mapping classifier systems into neural networks. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems (1)*, pages 49–56, 1989.

[9] H. de Garis. Genetic programming: modular neural evolution for Darwin machines. In *Proceedings of the International Joint Conference on Neural Networks*, page 194, Washington, DC, 1990. vol. I.

[10] N. Dodd. Optimisation of network structure using genetic techniques. In *Proceedings of the International Joint Conference on Neural Networks*, pages 965–970, San Diego, 1990. vol. I.

[11] S. Dominic, R. Das, D. Whitley, and C. Anderson. Genetic reinforcement learning for neural networks. In *Proceedings of the International Joint Conference on Neural Networks*, pages 71–76, 1991. vol. II, Seattle, Wa.

[12] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, & Machine Learning.* Addision–Wesley, Reading, MA, 1989.

[13] S. A. Harp and T. Samad. Genetic optimization of self–organizing feature maps. In *Proceedings of the International Joint Conference on Neural Networks*, pages 341–346, 1991. vol I, Seattle, Wa.

[14] S. A. Harp, T. Samad, and A. Guha. Designing application–specific neural networks using the genetic algorithm. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems (2)*, pages 447–454, 1989.

[15] J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI, 1975.

[16] R. Keesing and D. G. Stork. Evolution and learning in neural networks: The number and distribution of learning trials affect the rate of evolution. In R.P. Lippmann, J.E. Moody, and D.S Touretzky, editors, *Advances in Neural Information Processing (3)*, pages 804–810, 1991. (Denver 1990).

[17] J. R. Koza. A genetic approach to the truck backer upper problem and the inter–twined spiral problem. In *Proceedings of the International Joint Conference on Neural Networks*, pages 310–318, 1992. vol. IV, (Baltimore, Ma.).

[18] J. R. Koza and M. A. Keane. Cart centering and broom balancing by genetically breeding populations of control strategy programs. In *Proceedings of the International Joint Conference on Neural Networks*, page 198, Washington, DC, 1990. vol. I.

[19] J. R. Koza and J. P. Rice. Genetic generation of both the weights and architecture for a neural network. In *Proceedings of the International Joint Conference on Neural Networks*, page 397, 1991. vol. II, Seattle, Wa.

[20] L. Marti. Genetically generated neural networks I: representational effects. In *Proceedings of the International Joint Conference on Neural Networks*, pages 537–542, 1992. vol. IV, (Baltimore, Ma.).

[21] L. Marti. Genetically generated neural networks II: searching for an optimal representation. In *Proceedings of the International Joint Conference on Neural Networks*, pages 221–226, 1992. vol. II, (Baltimore, Ma.).

[22] I. Rechenberg. Artificial evolution and artificial intelligence. In R. Forsyth, editor, *Machine Learning, Principles and Techniques*, Chapman and Hall Computing, 1989.

[23] D. Rogers. Predicting weather using a genetic memory: a combination of Kanerva's sparse distributed memory with Holland's genetic algorithms. In D.S. Touretzky, editor, *Advances in Neural Information Processing Systems (2)*, pages 455–464, 1989.

[24] W. M. Spears and K. A. De Jong. Using neural networks and genetic algorithms as heuristics for NP–complete problems. In *Proceedings of the International Joint Conference on Neural Networks*, page 118, Washington, DC, 1990. vol. I.

[25] D. Whitley. The GENITOR algorithm and selection pressure: Why rank–based allocation of reproductive trials is best. In *Proceedings Third International Conference on Genetic Algorithms*, June 1990. Washington DC.

[26] D. Whitley and C. Bogart. The evolution of connectivity: Pruning neural networks using genetic algorithms. In *Proceedings of the International Joint Conference on Neural Networks*, page 134, Washington, DC, 1990. vol. I.

[27] D. Whitley and T. Starkweather. Optimizing small neural networks using a distributed genetic algorithm. In *Proceedings of the International Joint Conference on Neural Networks*, page 206, Washington, DC, 1990. vol. I.

[28] A. P. Wieland. Evolving controls for unstable systems. In *Connectionist Models: Proceedings of the 1990 Summer School*, page 81, Morgan Kaufmann, 1990.