

# **Genetic Algorithms and Neural Networks: A Comparison Based on the Repeated Prisoner's Dilemma**

Robert E. Marks,<sup>1</sup>  
AGSM, UNSW, Sydney, NSW 2052  
bobm@agsm.unsw.edu.au

Hermann Schnabl,  
University of Stuttgart and AGSM, UNSW  
schnabl@sofo.uni-stuttgart.de

Revised: May 15, 1998.

**ABSTRACT** Genetic Algorithms (GAs) and Neural Networks (NNs) in a wide sense both belong to the class of *evolutionary computing algorithms* that try to mimic natural evolution or information handling with respect to everyday problems such as forecasting the stock market, firms' turnovers, or the identification of credit bonus classes for banks. Both methods have gained more ground in recent years, especially with respect to micro-economic questions. But owing to the dynamics inherent in their evolution, they belong to somewhat disjunct development communities that interact seldom. Hence comparisons between the different methods are rare. Despite their obvious design differences, they also have several features in common that are sufficiently interesting for the innovation-oriented to follow up and so to understand these commonalities and differences. This paper is an introductory demonstration of how these two methodologies tackle the well-known Repeated Prisoner's Dilemma.

---

1. The first author wishes to acknowledge the support of the Australian Research Council.

## 1. Introduction

The relationship between biology and economics has been long: it is said that 150 years ago both Wallace and Darwin were influenced by Malthus' writings on the rising pressures on people in a world in which human population numbers grew geometrically while food production only grew arithmetically. In the early 1950s there was an awakening that the processes of market competition in a sense mimicked those of natural selection. This thread has followed through to the present, in an "evolutionary" approach to industrial organisation. Recently, however, computer scientists and economists have begun to apply principles borrowed from biology to a variety of complex problems in optimisation and in modelling the adaptation and change that occurs in real-world markets. There are three main techniques: Artificial Neural Networks, Evolutionary Algorithms (EAs), and Artificial Economies, related to Artificial Life. Neural nets are described below. We discuss Evolutionary Algorithms<sup>2</sup> in general, and Genetic Algorithms in particular, below.<sup>3</sup>

Techniques of artificial economies borrow from the emerging discipline of artificial life (Langton et al. 1992) to examine through computer simulation conditions sufficient for specific economic macro-phenomena to emerge from the interaction of micro units, that is, economic agents. Indeed, Evolutionary Algorithms may be seen as falling into the general area of the study of *complexity*. In our view, the *emergence* of such objects of interest as characteristics, attributes, or behaviour distinguishes complexity studies from traditional deduction or induction. Macro phenomena emerge in sometimes unexpected fashion from aggregation of interaction of micro units, whose individual behaviour is well understood.

Genetic Algorithms (GAs) and Neural Networks (NNs) in a wide sense both belong to the class of *evolutionary computing algorithms* that try to mimic natural evolution or information handling with respect to everyday problems such as forecasting the stock market, firms' turnovers, or the identification of credit bonus classes for banks.<sup>4</sup> Both methods have gained

- 
2. Examples of EAs are: Genetic Algorithms (GAs), Genetic Programming (GP), Evolutionary Strategies (ESs), and Evolutionary Programming (EP). GP (Koza 1992) was developed from GAs, and EP (Sebald & Fogel 1994) from ESs, but GAs and ESs developed independently, a case of convergent evolution of ideas. ESs were devised by German researchers in the 1960s (Rechenberg 1973, Schwefel & Männer 1991), and GAs were suggested by John Holland in 1975. Earlier, Larry Fogel and others had developed an algorithm which relied more on mutation than on the sharing of information between good trial solutions, analogous to "crossover" of genetic material, thus giving rise to so-called EP.
  3. In his Chapter 4, Networks and artificial intelligence, Sargent (1993) outlines Neural Networks, Genetic Algorithms, and Classifier Systems. Of interest, he remarks on the parallels between neural networks and econometrics in terms of problems and methods, following White (1992).
  4. Commonalities between both methods are: mimicking events/behaviour as information flows within a learned or adaptive structure. Differences among EAs include: their methods of solution representation, the sequence of their operations, their selection schemes, how crossover/mutation are used, and the determination of their strategy parameters

more ground in recent years, especially with respect to micro-economic questions. Despite their obvious design differences, they also have several features in common that are sufficiently interesting for the innovation-oriented to follow up and so to understand these commonalities and differences.<sup>5</sup> Owing to the dynamics inherent in the evolution of both methodologies, they belong to somewhat disjunct scientific communities that interact seldom. Therefore, comparisons between the different methods are rare.

Genetic Algorithms (GAs) essentially started with the work of Holland (1975), who in effect tried to use Nature's genetically based evolutionary process to effectively investigate unknown search spaces for optimal solutions. The DNA of the biological genotype is mimicked by a bit string. Each bit string is seeded randomly for the starting phase and then goes through various (and also varying) procedures of mutation, mating after different rules, crossover, sometimes also inversion and other "changing devices" very similar to its biological origins in Genetics. A fitness function works as a selection engine that supervises the whole simulated evolution within the population of these genes, and thus drives the process towards some optimum, measured by the fitness function. Introductory Section 2 gives more details on GAs and their potential in solving economic problems.

Neural Networks (NNs) are based on early work of McCulloch & Pitts (1943), who built a first crude model of a biological neuron with the aim to simulate essential traits of biological information handling. In principle, a neuron gathers information about the environment via synaptic inputs on its so-called dendrites (= input channels) — which may stem from sensory cells such as in the ear or eye) or from other neurons — compares this input to a given threshold value, and, under certain conditions, "fires" its axon (= output channel), which leads to an output to a muscle cell or to another neuron that takes it as information input again. Thus, a neuron can be viewed as a complex "IF...THEN...[ELSE]..." switch. Translated into an economic context, the neural mechanism mimics decision making rules such as "IF the interest rate falls AND the inflation rate is still low THEN engage in new buys on the stock market," where the capital letters reflect the switching mechanism inherent in the neural mechanism. Still more realistic state-of-the-art modelling replaces the implicit assumption of weights of unity (for the straight "IF" resp. the "AND") by real-valued weights. Thus, the above IF..THEN example turns into more fuzzy information handling, viz.: "IF (with probability 0.7) the interest rate is lowered AND (with probability 0.8) the inflation rate is still low THEN engage in new buys on the stock market". This fuzzier approach also reflects different strengths of synaptic impact in

---

5. This paper aims to be an introductory demonstration of how these two methodologies tackle the well-known Repeated Prisoner's Dilemma. A further paper may concentrate on both methods' basic approaches in handling the given information in achieving their goals.

the biological paradigm.

While a single neuron is already a pretty complex “switch”, it is clear that a system of many neurons, a neural network, comprising certain combinations of units, mostly in patterns of hierarchical layers, will be able to handle even more complex tasks.

Development of the NN approach to solving everyday problems made good progress until the late 'sixties, but withered after an annihilating critique by Minsky & Papert (1967). This attack was aimed at the so-called “linear separability problem” inherent in 2-layer models, but was taken as being true for all NN models. The problem inherent in the criticised models was that a Multi-Layer-Perceptron (MLP), which could handle non-linear problems, lacked an efficient learning algorithm and thus was not applicable in practice. The cure came with the publication of Rumelhart & McClelland (1986) on the solution of the so-called (Error-) *Backpropagation Algorithm* that made nonlinear NNs workable.

The name “Neural Networks” already describes what they try to do, i.e. to handle information like biological neurons do, and thus using the accumulated experience of nature or evolution in developing those obviously effective and viable tools for real-time prediction tasks. (One thinks of playing tennis and the necessary on-line forecast of where the ball will be so the player can return it.) There is also a growing list of successful economic applications, including forecasting stock markets and options and credit bonus assignments, as well as more technically oriented tasks with economic implications, such as detecting bombs in luggage, detecting approaching airplanes including their brand type by classifying radar signals, or diagnosing diseases from their symptoms. Section 2 is an introduction to GAs. Section 3 is an introduction to NNs. Sections 4 and 5 then show how the Repeated Prisoner’s Dilemma (RPD) is tackled by each method.

## **2. The Genetic Algorithm Approach — A Short Introduction**

Genetic algorithms are a specific form of EA. The standard GA can be characterised by: operating on a population of bit-strings (0 or 1). where each string represents a solution, and GA individual strings are characterised by a duality of: the structure of the bit-string (the genotype), and the performance of the bit-string (the phenotype).<sup>6</sup> In general the phenotype (the string’s performance) *emerges* from the genotype.

---

6. We characterise the phenotype as the performance or behaviour when following the now-standard approach of using GAs in studying game-playing, in particular playing the Repeated Prisoner’s Dilemma, since the only characteristic of concern is the individual’s performance in the repeated game. Such performance is entirely determined by the string’s structure, or genotype, as well, of course, by the rival’s previous behaviour.

## 2.1 GA: The evolutionary elements:

There are four evolutionary elements:<sup>7</sup>

1. *Selection* of parent individuals or strings can be achieved by a “wheel of fortune” process, where a string’s probability of selection is proportional to its performance against the total performance of all strings.
2. *Crossover* takes pairs of mating partners, and exchanges segments between the two, based on a randomly chosen common crossover point along both strings.
3. *Mutation*: with a small probability each bit is flipped. This eliminates premature convergence on a sub-optimal solution by introducing new bit values into a population of solutions.
4. *Encoding* is the way in which the artificial agent’s contingent behaviour is mapped from the individual’s structure to its behaviour. As well as the decision of whether to use binary or decimal digits, or perhaps floating-point numbers, there is also the way in which the model is encoded.

## 2.2 Detailed description of a GA<sup>8</sup>

How can we use the GA to code for the behaviour of the artificial adaptive agents? How can strategies (sets of rules) for playing repeated games of the Repeated Prisoner’s Dilemma (RPD) be represented as bit strings of zeroes and ones, each locus or substring (or gene) along the string mapping uniquely from a contingent state — defined by all players’ moves in the previous round or rounds of the repeated game — to a move in the next round, or a means of determining this next move? This coding problem is discussed in more detail below.

We describe these behaviour-encoding strings as “chromosomes” because, in order to generate new sets of strings (a new generation of “offspring”) from the previous set of strings, GAs use selection and recombinant operators — crossover and mutation — derived by analogy from population genetics. Brady (1985) notes that “during the course of evolution, slowly evolving genes would have been overtaken by genes with better evolutionary strategies,” although there is some dispute about the extent to which such outcomes are optimal (Dupré 1987). The GA can be thought of as an optimization method which overcomes the problem of local fitness optima, to obtain optima which are almost always close to global (Bethke 1981).

- 
7. There are variants for each of these. Selection: linear dynamic scaling, linear ranking, stochastic universal sampling, survival of élites. Crossover:  $n$ -point crossover, uniform crossover. Coding: Gray code — small genotype changes → small phenotype changes; decimal strings; real numbers.
  8. For an introduction to GAs, see Goldberg (1989). See also Davis (1991), Michalewicz (1994), Nissen & Biethahn (1995), and Mitchell (1996).

Moreover, following biological evolution, it treats many candidate solutions (individual genotypes) in parallel, searching along many paths of similar genotypes at once, with a higher density of paths in regions (of the space of all possible solutions) where fitness is improving: the “best” individual improves in fitness and so does the average fitness of the set of candidates (the population).

Hereditary models in population genetics define individuals solely in terms of their genetic information: the genetic structure of an individual — or genotype — is represented as strands of chromosomes consisting of genes, which interact with each other to determine the ultimately observable characteristics — or phenotype — of the individual. A population of individuals can be viewed as a pool of genetic information. If all individuals in the population have equal probability of mating and producing offspring, and if the selection of mates is random, then the information in the gene pool will not change from generation to generation. But environmental factors affect the fitness of phenotypes of individuals, and hence affect the future influence of the corresponding genotypes in determining the characteristics of the gene pool — the principle of natural selection, which results in a changing gene pool as fitter genotypes are exploited. Natural selection can be viewed as a search for coadapted sets of substrings which, in combination, result in better performance of the corresponding phenotype (the individual’s behaviour) in its environment.

Schaffer & Grefenstette (1988) argue that the theory of GAs derived by Holland (1975) predicts that substrings associated with high performance will spread through the new populations of bit strings. Paraphrasing Holland (1984), a GA can be looked upon as a sampling procedure that draws samples from a potential set  $T$ . With each sample is associated a value, the fitness (or score) of the corresponding genotype (or fundamental hereditary factors). Then the population of individuals at any time is a set of samples drawn from  $T$ . The GA uses the fitness (scores) of the individuals in the population at each generation to “breed” and test a new generation of individuals, which may include the best individuals from the previous generation. The new generation is “bred” from the old using genetic operators: selection of parents according to their fitness, crossover of genetic material from both parents, and random mutation of bits. This process progressively biases the sampling procedure towards the use of combinations of substrings associated with above-average fitness in earlier generations (that is, sample individuals characterized by higher scores because their behaviours are “better”), so the mean score of successive generations rises owing to selective pressures. A GA is all but immune to some of the difficulties that commonly attend complex problems: local maxima, discontinuities, and high dimensionality.

Although realizations of the GA differ in their methods of survival selection, of mate selection, and of determining which structures will disappear, and differ in their size of population and their rates of application of the different genetic operators, all exhibit the characteristic known as *implicit parallelism*. Any structure or string can be looked at as a collection of substring components or schemata which together account for the good or bad performance of the individual structure. Then Holland’s Schema

Sampling Theorem (Holland 1975, Mitchell 1996) demonstrates that schemata represented in the population will be sampled in future generations in relation to their observed average fitness, if we can assume that the average fitness of a schema may be estimated by observing some of its members. (Note that many more schemata are being sampled than are individual structures of the population being evaluated.) Genetic algorithms gain their power by exploring the space of all schemata and by quickly identifying and exploiting the combinations which are associated with high performance.

The most important recombination operator is *crossover*. Under the crossover operator, two structures in the mating pool exchange portions of their binary representation. This can be implemented by choosing a point on the structure at random — the crossover point — and exchanging the segments to the right of this point. For example, let two “parent” structures be

$$\begin{aligned}x_1 &= 100:01010, \text{ and} \\x_2 &= 010:10100.\end{aligned}$$

and suppose that the crossover point has been chosen as indicated. The resulting “offspring” structures would be

$$\begin{aligned}y_1 &= 100:10100, \text{ and} \\y_2 &= 010:01010.\end{aligned}$$

Crossover serves two complementary search functions. First, it provides new strings for further testing within the structures already present in the population. In the above example, both  $x_1$  and  $y_1$  are representatives of the structure or schema 100#####, where the # means “don’t care, because the value at this position is irrelevant.” (If 1001 is a point, then 100# is a line, and 10## is a plane, and 1### is a hyperplane.) Thus, by evaluating  $y_1$ , the GA gathers further information about this structure. Second, crossover introduces representatives of new structures into the population. In the above example,  $y_2$  is a representative of the structure #1001###, which is not represented by either “parent.” If this structure represents a high-performance area of the search space, the evaluation of  $y_2$  will lead to further exploration in this part of the search space. The GENESIS package (Grefenstette 1987), which we use, implements two crossover points per mating.

A second operator is *mutation*: each bit in the structure has a chance of undergoing mutation, based on an interarrival interval between mutations. If mutation does occur, a random value is chosen from {0,1} for that bit. Mutation provides a mechanism for searching regions of the allele space not generated by selection and crossover, thus reducing the likelihood of local optima over time, but mutation is capable only of providing a random walk through the space of possible structures.

The GAs do not require well-behaved, convex objective functions — indeed, they do not require closed objective functions at all — which provides an opportunity for an exhaustive study of the solution to repeated games. This is possible because to use the GA to search for better solutions it is

sufficient that each individual solution can be scored for its “evolutionary fitness:” in our case the aggregate score of a repeated game provides that measure, but in general any value that depends on the particular pattern of each individual chromosome will do.

### 2.3 Applications

For a comprehensive survey of the use of evolutionary algorithms, and GAs in particular, in management applications, see Nissen (1995). In Industry: production planning, operations scheduling, personnel scheduling, line balancing, grouping orders, sequencing, and siting. In financial services: risks assessment and management, developing dealing rules, modelling trading behaviour, portfolio selection and optimisation, credit scoring, and time series analysis.

## 3. The Neural Network Approach — A Short Introduction

Neural Nets<sup>9</sup> can be classified in a systematic way as systems or models composed of “nodes” and “arcs”, where the nodes are artificial neurons or *units* (in order to distinguish them from their biological counterparts, which they mimic only with respect to the most basic features). Usually, within a specific NN all units are the same. The arcs, or connections between the units, simultaneously mimic the biological *axons* and the *dendrites* (in biology, the fan-in or input-gathering devices) including the *synapses* (i.e. the information interface between the firing axon and the information-taking dendrite). Their artificial counterpart is just a “weight” (given by a real-valued number) that reflects the strength of a given “synaptic” connection. The type of connectivity, however, is the basis for huge diversity in NN architectures, which accompanies great diversity in their behaviour. Figure 1 shows the described relationships between the biological neuron and its artificial counterpart, the unit.

### 3.1 Units and Neurons

There are two functions governing the behaviour of a unit, which normally are the same for all units within the whole NN, i.e.

- the input function, and
- the output function.

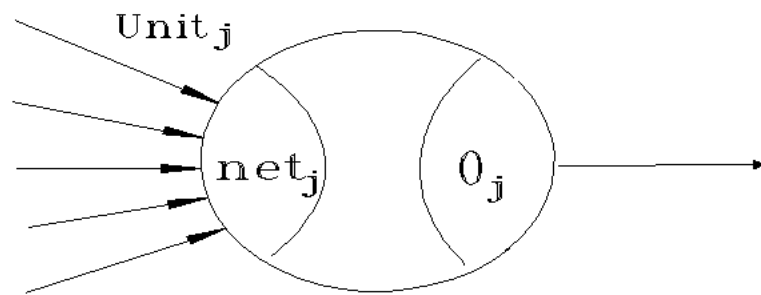
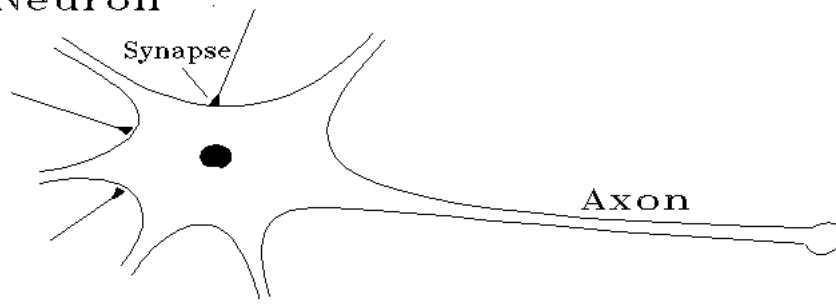
The input function is normally given by equation (1). The unit under consideration, unit  $j$ , integrates or sums up the numerous inputs:

$$net_j = \sum_i w_{ij} x_i, \quad (1)$$

where  $net_j$  describes the result of the net inputs  $x_i$  (weighted by the weights  $w_{ij}$ ) impacting on unit  $j$  (cf. the arrows in Figure 1, lower graph). These inputs can stem from two sources. First, if unit  $j$  belongs to the hierarchically



Neuron



### Figure 1. Neuron and Unit

lowest layer of the NN — the input layer — then they are caused by the environment. Second, if unit  $j$  belongs to a hierarchically higher layer of the NN, such as the hidden layer or the output layer, then the inputs come from units below unit  $j$  in the hierarchy. (In Figures 3 and 4, below, this means from units on the left.) Thus, in vector notation, the input function (1) sums the inputs  $x_i$  of its input vector  $\mathbf{x}$  according to their “strengths”, using the appropriate weights of a weight matrix  $\mathbf{W} = \{w_{ij}\}$ .

The output function exhibits a great variety, and has the biggest impact on behaviour and performance of the NN. The main task of the output function is to map the outlying values of the obtained neural input back to a bounded interval such as  $[0,1]$  or  $[-1,+1]$ .<sup>10</sup> Figure 2 shows some of the most frequently used output functions,  $o_j = f(\text{net}_j) = f(\sum_i w_{ij} x_i)$ .

The output  $o_j$  for a *digital* (or *Heaviside*) function would, for example, be given by equation (2) (Figure 2a):

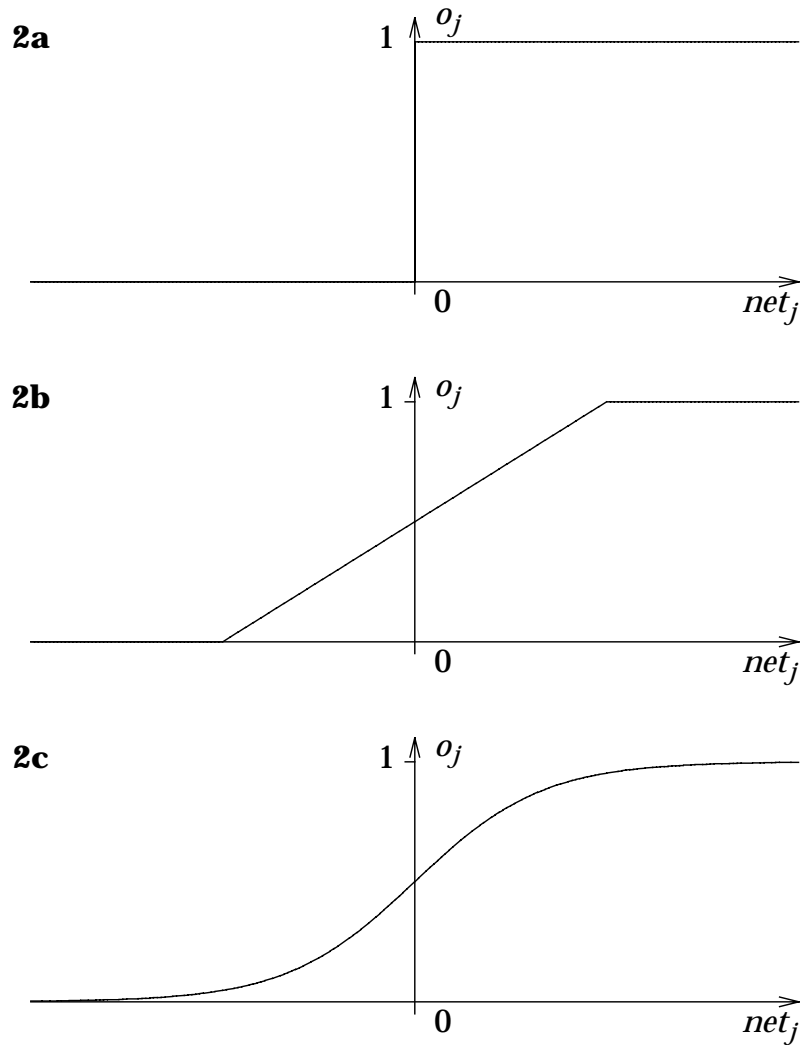
$$o_j = D(\text{net}_j - T) = D(\sum_i w_{ij} x_i - T), \quad (2)$$

where the symbol  $D$  stands for the Dirichlet operator, which gives a “step function”. Given, for example, a threshold  $T = 0.5$ , then output  $o_j = 1$  if  $\text{net}_j > 0.5$ , and  $o_j = 0$  otherwise. (See also Figure 2a, but for  $T = 0$ .) Other functions are the *semi-linear function* (see Figure 2b), and the so-called *Fermi* or *sigmoid functions* (Figure 2c and equation (3)):

---

9. For a rigorous introduction, see White (1992) or Bertsekas & Tsitsiklis (1996).

10. For this reason, the output function is sometimes known as the “squasher” function (Sargent 1993, p.54.)



**Figure 2.** Output Functions

$$o_j = \frac{1}{1 + e^{-\gamma net_j}}, \quad (3)$$

where  $\gamma$  is the “gain” of the function, i.e. the slope at  $net_j = 0$ . This function has some advantages, due to its differentiability within the context of finding a steepest descent gradient for the backpropagation method and moreover maps a wide domain of values into the interval  $[0,1]$ .

### 3.2 Net architecture

Two basic types of architecture can be distinguished: *feed-forward* and *recurrent* nets. In principle, each unit could be connected to all others (recurrent nets), while the feed-forward types propagate information strictly only forward from the input to the output and are organized in layers (one input layer, one output layer, and at least one so-called hidden layer, lying between the input and output layers, as in Figure 3 below).

The Hopfield net, the BAM (Bidirectional Associative Memory), and the so-called *Boltzmann machine* are recurrent networks. They handle

nonlinearities very well, but have a very limited “memory”, i.e. they can generalize only for less differentiated cases, compared to the feed-forward type. It turns out that the MLP (Multi-Layer Perceptron), a standard feed-forward type of NN, is best suited to use in the economic context, because its ability to take into account complex situations is much higher than those of the recurrent nets. We therefore focus on this type of NN architecture here.

### 3.3 Learning strategies

In addition to the output function used and the net architecture, the way they learn is a third criterion defining the structure and performance of NNs. We distinguish, first, *supervised learning* (where the “trainer” of the net knows the correct result and gives this information to the net with each learning step) and, second, *unsupervised learning* (where the net itself has to learn what is correct and what is not correct, mainly by using measures of similarity with events encountered earlier in the “learning history”). Moreover, the way a NN learns also depends on the structure of the NN and cannot be examined separately from its design. Therefore, we again focus on the most common type of learning, developed for the MLP, which is *Error Backpropagation*, although we also consider its predecessors.

There is a history of learning rules which starts with the so-called Hebb rule, formulated following Donald Hebb’s observation in neurology that the synaptic connection between two neurons is enhanced if they are active at the same time (Hebb 1949). As *learning* in the NN is simulated by adapting the (information-forwarding) weights between the different layers, this leads to equation (4) for an appropriate weight adaptation:

$$\Delta w_{ij} = \eta o_i o_j, \quad (4)$$

where  $\eta$  is an appropriate learning rate (mostly  $0 < \eta < 1$ ) and  $w_{ij}$  stands for the weight connecting, for example, input unit  $i$  and a unit  $j$ , located in the hidden layer.

Further development yielded the so-called *Delta rule*, which is a kind of “goal-deviation correction” (Widrow & Hoff 1960). Its formula is given by equation (5):

$$\Delta w_{ij} = \eta (z_j - o_j) o_i, \quad (5)$$

where  $\eta$  again gives the learning rate, and where  $z_j$  is the  $j$ th element of the goal vector. Thus,  $\delta = \mathbf{z} - \mathbf{o}$  describes the vector of deviations between the output propagated by the actual weight structure of the NN and the desired values of the target vector. The so-called *generalized Delta rule* at the heart of the Backpropagation Algorithm is given by equation (6) in a general form, for a weight matrix between any two layers, and must be further specified, when applied, with respect to the layers to which the weight matrix then is contingent:

$$\Delta w_{ij} = \eta \delta_j o_i. \quad (6)$$

This specification is done in defining the error term  $\delta_j$  in (6), since unit  $j$  is a member of the output layer or any *hidden layer*. For an output unit  $j$ , the Fermi output function (see Figure 2c and equation (3)) is mostly used, so that

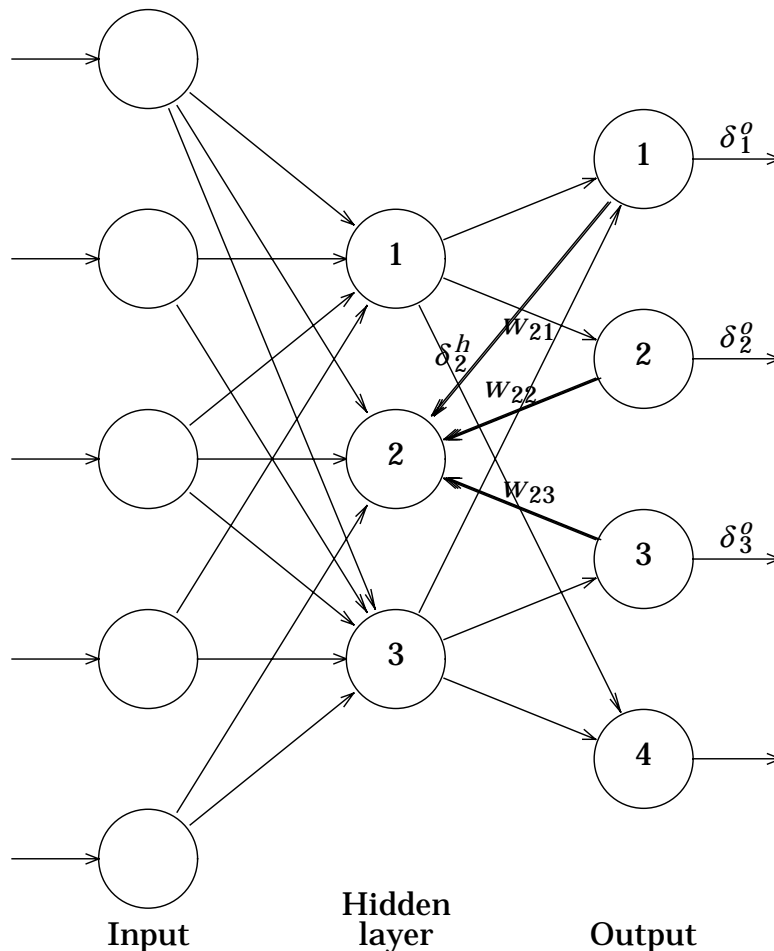
the derivative is easy to calculate:

$$\delta_j = o_j(1 - o_j)(z_j - o_j). \quad (7)$$

For the weight matrices lying to the left of the hidden layer (cf. Figure 3), however, the errors are not observable and therefore must be imputed. This imputation is performed according to equation (8):

$$\delta_i = o_i(1 - o_i) \sum_j w_{ij} \delta_j. \quad (8)$$

The basic structure of this MLP type of NN, as well as the backward propagation of errors  $\delta_j$  occurring at the output unit  $j$ , can be seen in Figure 3. While the deviations  $\delta_j^o$  in the output layer can be observed directly, the corresponding  $\delta_i^h$  are not known and cannot be observed. They must be inferred from their contribution in causing an error in the propagated output.



**Figure 3.** Error Backpropagation of  $j$

In Figure 3 this is shown for the hidden layer unit 2. The weights connecting this unit to the output units 1 through 3 —  $w_{21}$ ,  $w_{22}$ , and  $w_{23}$  — may be wrong to a certain extent and thus would cause errors in the output of units

1, 2 and 3. Correcting them in an appropriate manner, which takes into account their contribution in causing the errors, is called Error Backpropagation, since the errors  $\delta_j$  are imputed in a backwards direction and the weights concerned are adjusted in a manner such that a quadratic error function of the difference between target and NN output is minimized.

For example, the error of the hidden layer unit 2 is imputed by the contribution of its forward-directed information to the errors in the output layer, which were produced by these signals. The forward-directed signals (from each unit — here, unit 2 — of the hidden layer to several units of the output layer — here, units 1, 2, and 3) contribute to the output errors (here  $\delta_1^o$ ,  $\delta_2^o$ , and  $\delta_3^o$ , the errors in the signals from units 1, 2, and 3 in the output layer, respectively). The greater the error ( $\delta_2^h$ ) at the hidden unit 2, the greater the errors  $\delta_1^o$ ,  $\delta_2^o$ , and  $\delta_3^o$  at the output units 1, 2, and 3, respectively. So, in order to calculate the error ( $\delta_2^h$ ) at hidden unit 2, we must sum the errors observed from all output-layer units to which the hidden unit 2 contributes, suitable adjusted by the three weights of the signals from hidden unit 2 to the output layer units  $w_{ij}$  (shown in Figure 3 as linking output-layer unit  $j$  and hidden-layer unit  $i$ ; see the big-headed arrows). Therefore it is important in using a steepest-descent mechanism that the alterations of the weights can be found by differentiating the various output functions of the appropriate units (cf. Rumelhart & McClelland 1986).

Besides the Backpropagation Algorithm, which because of its gradient method approach may suffer the problem of converging to local minima, there are other methods for “learning”, i.e. adapting the weights, in the basket of evolutionary computing including GAs and the Evolutionary Strategies, all of which use a fitness function as an implicit error function. But these more evolutionary algorithms of learning do not use backpropagation.

Using NNs for everyday problems shows that there can be a tendency for a NN to “overfit” by learning the noise contained in the data too well, which reduces the potential for generalizing or the possibility of forecasting. This has led to different approaches to avoid overfitting of the weights to noise. One approach is to split the data set about 70%, 20% and 10%, using the first set for training and the second for validation and to end learning if the reported value of the error function increases again after a longer phase of reduction in the first part of the learning. Another approach takes into account that the architecture or final structure of a NN is highly dependent on the data. It then “prunes” the least important units and/or links of the NN before it continues learning data noise, and restarts learning. With the last data set covering 10% of the data, one can then test the “true” forecasting potential of the net, as these data are still unknown to the NN.

#### **4. The GA Solution to the Repeated Prisoner’s Dilemma**

To apply the GA to the solution of the Repeated Prisoner’s Dilemma (RPD), each individual string can be thought of as a mapping from the previous state of the game to an action (coöperate C or defect D) in the next round. That is, the players are modelled as stimulus-response automata, and the GA in effect searches for automata which score well in a RPD.<sup>11</sup> The RPD can pit each

individual in each population, or it can pit each individual against an environment of unchanging automata. The first method results in bootstrapping or *coevolution* of individuals, since each generation changes and so provides a changing niche for other players. The second method was used by Axelrod and Forrest (Axelrod 1987) in the first use of the GA to simulate the RPD — their niche of rivals was obtained by using the algorithms — some stochastic — that had been submitted to Axelrod’s now-famous computer tournaments (Axelrod 1984). Bootstrapping was first used by Marks (1992a), and it is this we describe here.

Choice of the environment is determined by the issue one is examining. For the RPD there is then the issue of how to model each artificial agent.<sup>12</sup> The state of the game is defined by the realisation of each player’s actions over the past three moves. With two players, each with two possible actions, there are  $2 \times 2 = 4$  combinations of actions possible in any one-shot game (see the corresponding payoffs in Table 1).

<b>Player A</b>	<b>Player B</b>	
	Coöperate	Defect
Coöperate	(3,3)	(0,5)
Defect	(5,0)	(1,1)

**TABLE 1.** The Prisoner’s Dilemma Payoffs

We model players as responding to the states (or combinations of actions) over the past three moves. This implies  $4 \times 4 \times 4 = 64$  possible states or combinations of actions. We calculate the state  $s(r)$  at round  $r$  as:

$$s(r) = 32 S(r-3) + 16 O(r-3) + 8 S(r-2) + 4 O(r-2) + 2 S(r-1) + O(r-1), \quad (9)$$

where  $S(z)$  and  $O(z)$  are respectively the moves of the player and his opponent in round  $z$ , either 0 for C or 1 for D.

How to map from state  $s(r)$  to action  $S(r)$ ? We use a string of 64 actions, one per possible state. With only two possible actions in the Prisoner’s Dilemma, each position on the string need only be a single bit — 0 maps to C, 1 to D — so we need 64 bits. Then, calculate the state  $s(r)$  from the actions of both players during the past three rounds (or games), using equation (9), and look at the  $s(r)$ th position on the string for the action  $S(r)$  to be undertaken next round.

Since at the first round there are no previous actions to remember, following Axelrod and Forrest we code in a “phantom memory”, which each

---

11. For a comprehensive survey of the use of finite automata as models of boundedly rational players in repeated games, see Marks (1992b).  
12. Here we describe the Axelrod/Forrest linear-mapping representation also used by Marks, but Miller (1996) describes a different representation, in which each agent is explicitly modelled as a finite automaton.

agent uses during the first three rounds in order to have a state from which to map the next action. We model this with an additional 6 bits — 2 bits per phantom round — which are used in equation (9) to establish the states and hence moves in the first three rounds; for succeeding rounds in the RPD, the actual moves are remembered and used in equation (9). For each repeated game, the history of play will be path-dependent, so by encoding the phantom memory as a segment of the bit string to be evolved by the GA over successive generations, we have effectively endogenised the initial conditions of the RPD.

Each player is thus modelled as a 70-bit string: 64 bits for the state-to-action mappings, plus 6 bits to provide the phantom memory of the three previous rounds' moves at the first round. This string remains unchanged during the RPD, and is only altered when a new population of 50 artificial agents is generated (see Step 4 below) by the GA, which uses the “genetic” operations of selection, crossover, and mutation. The first generation of strings are chosen randomly, which means that the mappings from state to action are random too.

The process of artificial evolution proceeds as follows:

1. In order to determine how well it performs in playing the RPD (in evolutionary terms its “fitness”), each of the population of 50 strings is pair-wise matched against all other strings. This implies 2,500 matchings, but symmetry of the payoff matrix means that only 1,275 matchings are unique.<sup>13</sup>
2. Each pair-wise match consists of 22 rounds of repeated interactions, with the Prisoner's Dilemma payoffs (see Table 1) for each interaction and each unchanging 70-bit string.<sup>14</sup>
3. Each string's fitness is the mean of its scoring in the 1,275 22-round encounters.
4. After all matches have occurred, a new population is generated by the GA, in which strings with a high score or fitness are more likely to be parents and so pass on some of their “genes” or fragments of their string structures to their offspring.
5. After several generations, the selective pressure towards those strings that score better means that individual strings emerge with much higher scores and that the population's average performance also rises.<sup>15</sup>

---

13. The diagonal elements of an  $n \times n$  matrix, plus half the off-diagonal elements (the upper or lower half) number  $n(n+1)/2$ .

14. As discussed in Marks (1992a), a game length of 22 corresponds to a discount factor of 0.67% per round. Note that the strings do not engage in counting (beyond three rounds) or in end-game behaviour.

15. Since there is coevolution of one's competing players, this improvement may not be as marked as the improvements seen when playing against an unchanging environment of players, as in Axelrod (1987). A recent book by Gould (1996) discusses this issue.



6. The evolutionary process ends after convergence of the genotype (as measured by the GA) or convergence of the phenotype (as seen by the pattern of play in the RPD).

#### 4.1 Results of the GA Approach

As mentioned above, Axelrod and Forrest (Axelrod 1987) were the first to use the GA in simulating the RPD. Axelrod (1984) had earlier invited submissions of algorithms for playing the RPD in two computer tournaments. Rapoport's simple Tit for Tat emerged as an extremely robust algorithm in both tournaments. One can consider Axelrod's use of the GA as a way of searching for new algorithms, and indeed this was explicitly done by Fujiki & Dickinson (1987), but using an early form of Genetic Programming, not a Genetic Algorithm. Axelrod and Forrest bred their mapping strings against a fixed niche of strategies, a weighted combination of algorithms submitted to the earlier tournament.

We describe results first presented at the annual ASSA meetings in New York in 1988 under the auspices of the Econometric Society, and later published (Marks 1992a), in which the niche — comprised of all other individuals in each generation — evolves as a consequence of the improvements of the individual mappings, generation from generation. This is bootstrapping, or *coevolution*, and was also pioneered by Miller (1996).

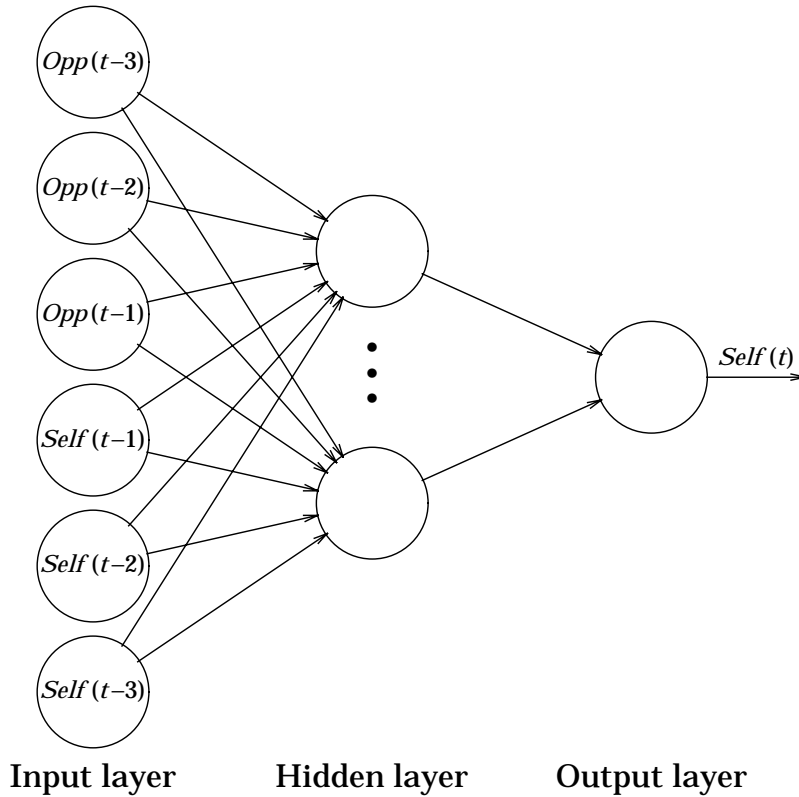
With coevolution, the outcome of interest is emergence of converging phenotypic characteristics, not the emergence of common genotypes. In our example, this means the emergence of *behaviour* in the RPD, not the emergence of common mappings. The main reason is that, given the selective pressures towards mutual coöperation (CC), as reflected in the payoff matrix of Table 1, there is selective pressure against other behaviour (phenotypes), and hence against positions on the string (genes) which correspond to one or more defections in the past three rounds.

As one would expect, mutual coöperation (CC) soon emerges as the outcome of coevolution of mapping strings, although high rates of mutation may occasionally disrupt this for some time.

### 5. The NN Solution to the Repeated Prisoner's Dilemma

The RPD can also be tackled by a NN. As with the GA solution, we assume three rounds back of "memory", i.e. the players take into account their own last three moves as well as the last three moves of their opponent in order to reach their own decision. As the decision encompasses only two possible moves — to coöperate or to defect — we can translate it to +1 (coöperate) or -1 (defect) as the only output of the NN. Thus, the input layer and the output layer are fixed, due to the specificity of the task, and only the hidden layer (besides the weights) offers a chance of adaptation towards an optimum.

This network structure is shown in Figure 4 and was taken from Fogel & Harrald (1994), whose experiments with this type of NN we follow here.<sup>16</sup>



**Figure 4.** Neural Net Architecture to Reflect RPD Behavior  
(After Fogel & Harrald 1994)

The logical structure of this NN, due to the task outlined here, is a kind of dual to the normal NN: while in the normal case the net gets inputs from the environment of data and tries to forecast future behaviour, here it “makes” the data by creating behaviour of the actual move (i.e. cooperate or defect). The data input of the net then is the history of one’s own and one’s opponent’s moves. Due to this somewhat unusual reversal of the significance of the propagation step of the NN, *the learning method* also belongs to a class which is — as described in Section 3.3 — not frequently used. Fogel & Harrald used an Evolutionary Strategy (ES) to adjust the weights, i.e. adapting the randomly initialized weights ( $w_{ij} \in [-0.5, +0.5]$ ). The weights were “mutated” by adding a small number taken from Gaussian distribution (not specified by the authors). Then the weights’ fitness was tested according to a fitness function reflecting the payoff of the behavioural output as a result

---

16. Cho (1995) models the Prisoner’s Dilemma and other two-person games played by a pair of perceptrons (or neural networks). In an infinitely repeated (undiscounted) Prisoner’s Dilemma, he shows that any individually rational payoff vector can be supported as an equilibrium by a pair of single-layer perceptrons (with no hidden layer) — the Folk Theorem. When mutual cooperation is not Pareto efficient, at least one player’s perceptron must include a hidden layer in order to encode all subgame-perfect equilibrium strategies.

of the simulation. This fitness function was given by equation (10):

$$f(a, b) = -0.75a + 1.75b + 2.25, \quad (10)$$

where  $a$  and  $b$ :  $a, b \in [-1, +1]$ , reflect players A's and B's behaviour, respectively.

This fitness function had to approximate the payoff matrix used by Axelrod (1987), as seen in Table 2.

Player A	Player B	
	Coöperate	Defect
Coöperate	(3,3) <i>3.25</i>	(0,5) <i>0.25</i>
Defect	(5,0) <i>4.75</i>	(1,1) <i>1.25</i>

**TABLE 2.** Approximations to the PD Payoffs

The original payoffs are given in parenthesis while Player A's approximated payoffs used in the NN approach are given in italics.

Fogel & Harrald proceeded as follows:

A population of fixed size (the simulations were done with five population sizes: 10, 20,  $\dots$ , 50 "parent" nets) was "seeded" with uniformly distributed random weights in  $[-0.5, 0.5]$ .

Then:

1. "Mutation" (as described above) of a parent NN produced a single offspring.
2. All networks played against each other for 151 rounds, in order to test the fitness, which was defined as the average payoff per round from equation (10).
3. The top-ranking half of the population of NNs were selected as parents to produce the next generation of offspring.
4. Until the 500th generation was reached, the process was repeated from Step 1.

As seen from the above process, only the weights were adapted in the evolutionary procedure. The number of units in the hidden layer was held constant for each simulation and given in advance. Variation of the number of hidden-layer units thus was exogenous; their number  $n$  was varied from  $n = 2$  (the minimal possible) to  $n = 20$  (the maximum admitted). Besides the insight into the structure and mechanics of the NN in solving the RPD (which is the main goal of this paper), the results are very interesting.

### *5.1 Results of the NN Approach*

Fogel & Harrald varied two essential determinants of the experiment: the number of "parent NNs" (from 10 to 50 in steps of 10) and the number of hidden-layer units (between 2 and 20), with the expectation that there should

be enough units in the hidden layer to enable sufficient behavioural complexity. Thus, it hardly could be expected that a 6–2–1 NN (a short-hand way of describing a NN with 6 input units, 2 hidden units and 1 output unit) would develop stable coöperative behaviour, and in fact it did not. On the other hand, a 6–20–1 NN most of the time showed — in Fogel & Harrald’s words — “fairly” persistent coöperative behaviour, and thus, to a certain extent, met the expectations, but could never establish a stable regime of coöperation like Axelrod’s paradigmatic results. Although delivering the best performance of all tested architectures, the level of coöperation as measured by the average payoffs was below what could be expected if all had always coöperated, and was worse than the Axelrod simulations.

There also seemed to exist an increasing probability of stabilizing coöperation with population size, but it was never stable, and could instead produce a sudden breakdown of coöperation, even after generation 1,200, from which it mostly did not recover.

### *5.2 Critique and Conclusions to the NN Approach*

The results described above are surprising, and invite further investigation. A standard critique of the NN approach could be of the simulation design, which makes the evolutionary adaptations of weights and the network complexity (given here only by the number  $n$  of hidden units) disjunct parts of the trials. If we extrapolate “normal” NN experiences, then there is an inherent dependence of weights and NN structure with respect to an optimal adaptation. In this sense only *coadaptation* would make sense, but this was not implemented in the experiment of Fogel & Harrald’s.

The results contrast with the GA results, but give also rise to speculation about the different approaches of both methods: GAs work by a strict zero–one mapping within the genome string, while the above NN design admits of real values varying between  $-1$  or  $+1$ , thus weakening the boundedness or “degree of determinism” within the model, and possibly allowing a more “fuzzy” behavioural context between the opponents.

Unfortunately, the authors of this paper do not possess an appropriate NN simulator which would enable us to rerun the above experiments and to add elements or alter the design with respect to the above speculation, which could be tested by the so-called gain  $\gamma$  (i.e. the “steepness” of the sigmoid transfer function, cf equation (3)) of a unit and thus changing the behaviour gradually towards a more discrete (digital) one. Thus, these remarks can only be tentative and remain speculative.

## **6. Comparison of the Two Approaches and Concluding Remarks**

As the results may suggest, a close interpretation may be that the RPD is more the domain of the GA approach than that of the NN. In Section 3.3 we mention that the NN architecture must be specified with respect to the data it processes. Indeed, it is one of the wisdoms of NN expertise that the data structure will require — and, if allowed by pruning, will form — its special architectural form. In the above NN example, there was only limited opportunity for achieve this — only the number of hidden units.

Moreover, the NN tried to approximate the overt zero–one type of RPD problem — each player has only two choices, either Coöperate (C  $\equiv$  0) or Defect (D  $\equiv$  1) — at two points by real-valued functions:

- in simulating the integer-valued payoffs by a “best” equation, which provides for a much more linear approach (Table 2), as does the original (Table 1) used by the GA approach;
- in approximating the zero–one actions by real-valued numbers.

So the NN formulation is not as close to the problem as is the GA structure, which uses a zero–one approach and thus operates much more closely to the focus of the RPD problem. It may well be that the contrast of zero–one encoding of the GA solution against the more fuzzy, real-valued encoding of the NN are sufficient to explain the lower stability of performance of the NN compared to the GA, since it is readily imagined that the basin of attraction for the final solution of a typical zero–one problem such as the RPD is much more clear-cut and thus much more stable than it is for a smooth “landscape” explored by a NN. This conclusion may well be reversed for a problem which is formulated in real-valued solutions, such as the forecast of a stock price. It is certainly the case, at least when using the binary-string representation of solutions with the GA, that the number of significant digits is in general a prior decision: the length of the bit string places an upper limit on the precision of the solutions.

In conclusion we make some more general observations. Evolutionary algorithms are well suited for high-dimensional, complex search spaces. With EAs there are no restrictive requirements on the objective function (if one exists), such as continuity, smoothness, or differentiability; indeed, there may be no explicit objective function at all. The basic EA forms are broadly applicable across many diverse domains, and, with flexible customising, it is possible to incorporate more knowledge of the domain, although such domain knowledge is not required. They have been found to be reliable, and are easily combined with other techniques, to form so-called hybrid techniques. They make efficient use of parallel-processing computer hardware.

Evolutionary algorithms are heuristic in nature (with no guarantee of reaching the global optimum in a specific time); indeed, finding good settings for strategy parameters (population size and structure, crossover rate, mutation rate in the GA) can require some experience. They are often ineffective in fine-tuning the final solution. The theory of EAs is still being developed. They have comparatively high CPU requirements, although with Moore’s law in operation, this is less and less a problem.

## 7. References

- Adeli, H., & Hung S.-L. (1995) *Machine Learning: Neural Networks, Genetic Algorithms, and Fuzzy Systems*, NY: Wiley.
- Axelrod, R. (1984) *The Evolution of Coöperation*, New York: Basic Books.
- Axelrod, R. (1987) The evolution of strategies in the iterated Prisoner’s Dilemma, in: *Genetic Algorithms and Simulated Annealing*, L. Davis (ed.)

- (Pitman, London) pp.32–41.
- Bertsekas, D.P., Tsitsiklis, J.N. (1996), *Neuro-Dynamic Programming*, Belmont, Mass.: Athena Scientific.
- Bethke, A.D. (1981) Genetic algorithms as function optimizers. (Doctoral dissertation, University of Michigan). *Dissertation Abstracts International* **41**(9): 3,503B. (University Microfilms No. 81–06,101)
- Brady, R.M. (1985) Optimization strategies gleaned from biological evolution. *Nature* **317**: 804–806.
- Cho, I.-K. (1995) Perceptrons play the repeated Prisoner's Dilemma, *Journal of Economic Theory*, **67**: 266–284.
- Davis, L. (1991) A genetic algorithms tutorial. In: Davis L. (ed.) *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold.
- Dupré, J. (ed.) (1987) *The Latest on the Best: Essays on Evolution and Optimality*. Cambridge: MIT Press.
- Fogel, D.B., Harrald, P.G. (1994) Evolving continuous behaviors in the iterated Prisoner's Dilemma in: Sebald, A., Fogel, L. (eds.) *The Third Annual Conference on Evolutionary Programming*, Singapore: World Scientific, pp.119–130.
- Fujiki, C., Dickinson, J. (1987) Using the genetic algorithm to generate Lisp source code to solve the Prisoner's Dilemma. In: Grefenstette J.J. (ed.) *Genetic Algorithms and their Applications, Proceedings of the 2nd International Conference on Genetic Algorithms*. Hillsdale, N.J.: Lawrence Erlbaum.
- Goldberg, D.E. (1989) *Genetic Algorithms in Search, Optimization and Machine Learning*, Reading, Mass.: Addison-Wesley
- Gould S.J. 1996, *Full House: The Spread of Excellence from Plato to Darwin*, New York: Harmony Books. (Also, for some reason, published in London as *Life's Grandeur*.)
- Grefenstette, J.J. (1987) A User's Guide to GENESIS. Navy Center for Application Research in Artificial Intelligence, Naval Research Laboratories, mimeo., Washington D.C.
- Hebb, D. (1949) *The Organization of Behavior*, New York: Wiley.
- Holland, J.H. (1975) *Adaptation in Natural and Artificial Systems*. Ann Arbor: Univ. of Michigan Press. (A second edition was published in 1992: Cambridge: MIT Press.)
- Holland, J.H. (1984) Genetic algorithms and adaptation. In: Selfridge O., Rissland E., & Arbib M.A. (eds.) *Adaptive Control of Ill-Defined Systems*. New York: Plenum.
- Koza, J.R. (1992) *Genetic Programming*, Cambridge: MIT Press.

- Langton, C.G., Taylor, C., Farmer, J.D., Rasmussen, S. (ed.) (1992) *Artificial Life II*, Reading: Addison-Wesley.
- Marks, R.E. (1989) Niche strategies: the Prisoner's Dilemma computer tournaments revisited. *AGSM Working Paper 89-009*.  
<<http://www.agsm.unsw.edu.au/~bobm/papers/niche.pdf>>
- Marks, R.E. (1992a) Breeding optimal strategies: optimal behaviour for oligopolists, *Journal of Evolutionary Economics*, **2**: 17–38.
- Marks, R.E. (1992b) Repeated games and finite automata in: Creedy, J., Borland, J., Eichberger, J. (eds.) *Recent Developments in Game Theory*. Aldershot: Edward Elgar.
- McCulloch, W.S., Pitts, W. (1943) A logical calculus of the ideas immanent in nervous activity, *Bulletin of Mathematical Biophysics* **5**.
- Michalewicz, Z. (1994) *Genetic Algorithms+Data Structures = Evolutionary Programs*, Berlin: Springer Verlag, 2nd ed.
- Miller, J.H. (1996) The coevolution of automata in the repeated Prisoner's Dilemma. *Journal of Economic Behavior and Organization*, **29**: 87–112.
- Minsky, M., Papert, S. (1969) *Perceptrons*, Cambridge: MIT Press.
- Mitchell, M. (1996) *An Introduction to Genetic Algorithms*, Cambridge: MIT Press.
- Nissen, V. (1995) An overview of evolutionary algorithms in management applications, in *Evolutionary Algorithms in Management Applications*, ed. by J. Biethahn & V. Nissen, Berlin: Springer-Verlag, pp.44–97.
- Nissen, V., Biethahn, J. (1995) An introduction to evolutionary algorithms, in *Evolutionary Algorithms in Management Applications*, ed. by J. Biethahn & V. Nissen, Berlin: Springer-Verlag, pp.3–43.
- Rechenberg, I. (1973) *Evolutionsstrategie. Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*, Stuttgart: Frommann-Holtzboog.
- Rumelhart, D.E., McClelland, J.L. (1986) *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Vol. 1: Foundation, 2. Aufl., Cambridge: MIT Press.
- Sargent, T.J. (1993) *Bounded Rationality in Macroeconomics*, Oxford: O.U.P.
- Schaffer, J.D., Grefenstette, J.J. (1988) A critical review of genetic algorithms. Mimeo.
- Schwefel, H.P., Männer, R., (1991) *Parallel Problem Solving from Nature*, (Lecture Notes in Computer Science 496). Berlin: Springer-Verlag.
- Sebald, A., Fogel, L. (eds.) (1994) *The Third Annual Conference on Evolutionary Programming*, Singapore: World Scientific.

White, H. (1992) *Artificial Neural Networks: Approximation and Learning*, Oxford: Basil Blackwell.

Widrow, B., Hoff, M.E. (1960) Adaptive switching circuits, In: Institute of Radio Engineers, Western Electronic Show and Convention, Convention Record, Part 4, pp.96–104.