USING GENETIC ALGORITHMS TO DESIGN NEURAL REINFORCEMENT CONTROLLERS FOR SIMULATED PLANTS

P.H.M. Spronck and E.J.H. Kerckhoffs
Delft University of Technology
Faculty of Technical Mathematics and Informatics
Zuidplantsoen 4, 2628 BZ Delft, The Netherlands
E-mail: p.spronck@inter.nl.net (Spronck)
eugene@kgs.twi.tudelft.nl (Kerckhoffs)

KEYWORDS

Control systems, genetic algorithms, neural networks, reinforcement control, dedicated computer software.

ABSTRACT

In production industry often highly non-linear processes (plants) are encountered which need to be controlled. In this paper, attention is focused on neural reinforcement control of simulated plants. Reinforcement control uses an evaluation of the performance of the controller in a practical situation to adapt the controller to work better. The design of a neural controller in a reinforcement situation is not a trivial task. A method is discussed to design such neural controllers using genetic algorithms. To test the viability of the approach a specific software environment has been built, which enables performing experiments with many different genetic algorithm configurations. As an illustrative example, the design of a controller for a bioreactor simulation is discussed.

INTRODUCTION

In production industry and modern business frequently there are processes, often called *plants*, which need to be controlled. Often the controller function is taken by a human being or a mechanical controller. However, mechanical controllers are only suitable for simple situations, and human beings are expensive and cannot always react quickly enough in situations where large numbers of parameters or fast-changing processes are concerned. In modern control, plants are computer controlled. The control software can be based on numerical procedures, traditional artificial intelligence techniques (reasoning systems), neural networks and fuzzy logic techniques.

In this paper we consider neural control, that is, control in which neural networks are involved. In neural control, it is tried to exploit the learning capabilities of neural networks. The design and training of neural networks is normally not an easy task. If a good conventional or neural model of the plant to be controlled is available, then there are ways to construct a good neural controller for the plant concerned (Jarmulak et al. 1995b). However, such a model can not always be built easily, and sometimes not at all. In that case an option that remains is to use reinforcement control, that is, to develop the neural controller according to the success, or the lack of success, of the performance of this controller in a practical situation. No plant model is needed in this case.

Genetic algorithms are search algorithms based on the principles of natural selection and natural genetics. They might be particularly suited to develop neural reinforcement controllers; this is studied in the research reported in this paper. Up to now, there has not been much research in the application of genetic algorithms for the design of neural controllers, and the question of the viability of this approach is still open. The approach is promising, though, so more research in this direction is certainly encouraged. For this kind of research many experiments need to be done. A dedicated software environment has been developed to perform, in a flexible way, a large number of different experiments in this respect.

The paper first gives a short introduction to genetic algorithms. This is followed by a discussion of neural control and how genetic algorithms can be applied to the design of neural reinforcement controllers. Then a software environment especially developed to perform the necessary experiments is presented, and as an illustrative application example the development of a neural reinforcement controller for a bioreactor simulation is discussed.

GENETIC ALGORITHMS

Genetic algorithms are search algorithms based on the principles of natural selection and natural genetics. They have been invented by John Holland, who in 1975 published his book *Adaptation in Natural and Artificial Systems* (Holland 1992). It took years before genetic algorithms became a major interest in the artificial intelligence community, but since the early '90s they have got much attention and have become widely accepted as a powerful tool to handle complex optimisation problems.

While most optimisation techniques start with one potential solution to a problem, and adapt that solution to ultimately get it to an optimum, genetic algorithms work with a set of potential solutions to the problem at hand. This set is called a *population*, and the potential solutions in the set are called *individuals*. The individuals are not simply stored in the population; they are *encoded*. For instance, an individual may be translated by some method to a binary string. Such a coded individual is also called a *chromosome*. One character of a chromosome is called a *gene*, and a gene value is called an *allele*. Each individual in the population has been given a *fitness measure*, which indicates how well this individual performs in solving the problem, in relation to the other individuals in the population.

To get new and hopefully fitter individuals, the genetic algorithm applies *genetic operators* to individuals which are selected from the population. A genetic operator takes one or more *parent* individuals, and performs some action on them to produce one or more new *child* individuals. For example, the *mutation* operator takes one individual as parent and makes a few changes in it to produce a child, while the (one-point) *crossover* operator takes two parents, cuts them both in two parts, and exchanges the tails to produce two children (see figure 1).

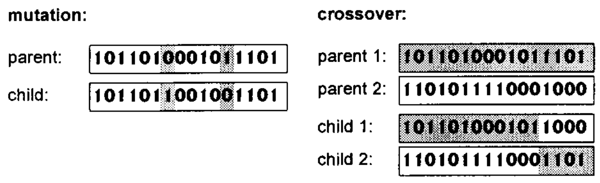

Figure 1: Examples of two genetic operators.

The selection of the parent chromosomes normally goes according to the fitness rates: the fittest individuals have a greater chance of being selected to procreate than the less fit individuals. Newly generated individuals either get inserted into the population, replacing other individuals, or are placed in a new population which will eventually replace the old population. The production of new individuals, called the *evolution* process, continues until some predefined goal is reached, most commonly until a maximum predetermined number of new individuals has been produced. At that point, the fittest individual in the population is considered to be the sought solution to the problem.

At first glance, genetic algorithms seem to be a completely random process, but in practice they work quite well provided the right configuration (population size, genetic operators, replacement policy, etc.) is chosen. Strong points of genetic algorithms are that they are robust, i.e. they work well in many different environments and on many different problems; that they search for a global optimum where most other techniques only lead to a local optimum; and that they need no more than a fitness measure to work. They are relatively slow, but since they are inherently parallel it is easy to speed them up if more processors are available. A weakness of genetic algorithms is that they are not guaranteed to lead to an acceptable solution, not even to a mediocre one. Experience in the design of the genetic algorithm and a sound analysis of the problem domain are certainly needed to insure that the genetic algorithm will indeed do what it is expected to do.

Those interested in more details of genetic algorithms are referred to the excellent introduction to the field by David Goldberg, *Genetic Algorithms in Search, Optimization & Machine Learning* (Goldberg 1989).

NEURAL CONTROL

In this paper we focus on the control of (simulated) plants with the use of neural networks (in the following with "plant" is frequently meant "simulated plant"). A *plant* is a process which maps an input to an output. The plant may have internal states, i.e. the plant's output does not only depend on the current input but also on the input's and plant output's history. The input to the plant is presented by a *controller*, which should get the plant to produce a specific predetermined output. Such a target output is called a *setpoint*. The controller receives the plant output to guide the determination of the plant input needed to reach the setpoint (see figure 2).

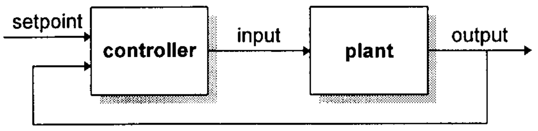

Figure 2: Plant control.

In the last decades, often straightforward computer programs have been employed as controllers. The more complex the plant becomes, the more difficult it is to design a good computer controller. Furthermore, really good computer controllers have mostly been developed to control linear systems. In practice, however, plants are rarely linear. Of the AI techniques which have been used in control design, neural networks have shown some promise. Neural networks have the ability to learn a non-linear function, and can therefore be used to control non-linear plants. There are basically two ways in which neural networks can be employed in control systems. First, the neural network can be trained to be the plant controller itself. Second, the neural network can be trained to be a model of the plant (called a neural *identifier*), and this is then properly incorporated in the control system.

The identifier-based approach has been studied by many researchers, a.o. by Jarmulak using his NeuroControl Workbench (Jarmulak 1994). He considers two kinds of models to create an identifier: forward and inverse. Inverse models are presented with a plant output, and react with the plant input needed to get that output. Since that is exactly what we wish a controller to do, an inverse model is theoretically equal to a controller. However, a good inverse model of a plant is far more difficult to build than a good forward model. Moreover, a direct inverse controller is not stable in practice and therefore useless. Forward models can be used in several configurations, of which the most successful is the so-called model-based predictive control, in which a plant identifier is used to improve plant input and a neural controller is trained on-line with these improvements.

In the aforementioned NeuroControl Workbench neural models are trained with backpropagation. Backpropagation works with a training set that consists of a number of plant inputs with the corresponding plant outputs. The neural network is then trained to produce the outputs when presented with the inputs. As stated before, the problem is that a plant will in general not react in one unique way to some input, while a training set can only attach one output to an input. To deal with this problem, the controller input consists of not only the desired plant output, but also of the current plant output and of some of the plant input's and output's history. In this way a static backpropagation network can learn a dynamic input-output relation. This strengthens the approach considerably, but it is of course not guaranteed that an acceptable plant model can always be designed.

An alternative control method is to use a neural network as the controller itself, and to train it with *reinforcement learning*. Reinforcement learning by definition does not make use of a plant model or a training set. Instead, observation of the performance of the controller in practical situations is used to adapt the controller to perform better. This is called *reinforcement control*. The design of a neural network as a controller is actually an optimisation problem. In reinforcement control the only information we have to adapt the neural controller concerns evaluations of the performance of the controller. Since genetic algorithms are used for optimisation, and since they need no more than a fitness evaluation to do the job, they seem to provide a viable approach to the design of neural controllers in a reinforcement situation.

Note that genetic algorithms could also be used to design a neural identifier which is then used in the controller. However, there are several good conventional techniques for this purpose, so genetic algorithms are not a logical first choice in this respect. Also note that genetic algorithms are only suitable for off-line training and are therefore dependent on a good plant simulation.

GENETIC ALGORITHMS AND NEURAL CONTROLLERS

A lot of research has been done concerning the use of genetic algorithms to optimise neural networks. This certainly does not mean that researchers are unanimous about what makes a genetic algorithm suitable for this task. Virtually every aspect of these genetic algorithms is subject for debate. For instance, concerning the chromosome encoding, some researchers prefer the use of real values for the connection weight encoding, while others defend the use of binary encoded weights. A lot of the points of discussion stem from the problem of so-called *competing conventions* (more aptly named the *structural/functional mapping problem*). The problem of competing conventions concerns the fact that one particular mapping from input to output can be encoded in several different ways. For instance, in a layered feedforward neural network, the nodes in one layer (including their connections) can be exchanged, leaving the neural network functionally the same, but structurally different. For an example, see figure 3.

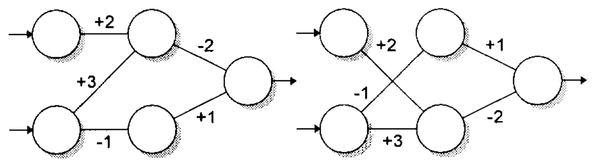

Figure 3: Competing conventions: two structurally different but functionally equivalent neural networks.

The possibility of the occurrence of competing conventions leads to a vast increase of the size of the solution space, which may lead to an increase of the time needed for the evolution process. Moreover, competing conventions also virtually nullify the beneficial effect of the crossover operator. This is because for the crossover operator to be executed in a useful manner on two chromosomes functionally equivalent neural nodes should be encoded in the same location on those chromosomes. For example, suppose we have devised an encoding mechanism in which every neural node is encoded as one gene of a chromosome, and that the optimal neural network is encoded as *ABCDEF*. Suppose there are two quite fit neural networks in the population, encoded as *ABCDEX* and *XBCDEF*. Performing the crossover operator on these two chromosomes has a great chance of resulting in the optimal chromosome. However, if competing conventions are allowed in the population, the second neural network could, for instance, be encoded as *FEDCBX*. In this case, use of the crossover operator would not result in the optimal chromosome, but instead would very probably produce children which are less fit than their parents. To solve the problem of competing conventions, different researchers use different approaches. Some just ignore the problem, some advocate the use of a small population combined with a high mutation rate, some use special genetic operators, and some rearrange the structure of one of the chromosomes before the crossover is executed.

For neural controller evolution in a reinforcement control situation, another aspect is added to the genetic algorithm design, namely how the test run, executed to determine the fitness of a neural controller, should be performed. Very little research has been done in this area. Whitley is one of the few researchers who has examined the subject of neural controller evolution in a reinforcement environment (Whitley et al. 1993). As a test plant he uses a *pole balancing system*. This is a system which consists of a cart on a rail, on which a pole rests. The pole can fall to the left and right. The cart is controlled by applying a force, and it is the objective of the controller to keep the pole from falling (see figure 4). The reinforcement aspects of the pole balancing controller evolution are easily determined. The plant fails if the pole falls, therefore the fitness is determined by the length of time the controller can keep the plant from failing. This kind of fitness is called *time-until-failure* based fitness. A maximum test run length needs to be set, and the controller is considered to be perfect if the pole remains balanced for that amount of time. The only remaining reinforcement aspect is the initial plant state used in the test run. Whitley starts by placing the cart at one end of the rail, and the pole leaning over at some specific angle to the other side of the rail.

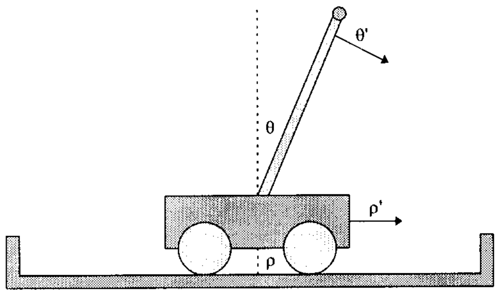

Figure 4: The pole balancing system. The state of the system is described by the pole angle θ, the angular velocity θ', the cart position ρ, and the cart velocity ρ'.

There are many plant types for which the controller cannot be evolved with time-until-failure-based fitness. For instance, a *trolley* is a plant which consists simply of a cart on a rail, and the controller has to direct the cart to specific positions on that rail. The plant fails if the cart drives off the rail, but just keeping the plant from failing is not sufficient for the controller to be any good. The controller should also minimise the distance between the cart position and the setpoint position. A straightforward way of implementing fitness determination for controllers for plants like the trolley is to use the mean square error (MSE) over the test run, wherein the error is defined as the difference between the current plant output and the setpoint for this output. This is called *MSE-based* fitness determination, and in this case the inverse fitness is used, meaning that the controller which has the lowest MSE is considered to be the most fit. If the plant does not fail during the test run (that is, if the cart does not drive off the rail), the inverse fitness is simply the sum of the squares of the error for each time step, divided by the number of time steps. If the plant does fail, this situation can be handled in several ways. We can simply reset the plant and continue the run; or we can award the maximum error for all the remaining time steps; or we can award the MSE for the time steps until the failure for the rest of the run. In the latter case, it would be wise to also set some penalty on premature failure.

Very little research has been done in the subject of genetic algorithm based evolution of neural controllers in a reinforcement environment, and most of it is concentrated on controllers with time-until-failure based fitness. The handful of papers published about the subject are, however, optimistic about the possibilities and results. Because this research is mainly of experimental nature (there is little theory about what makes a genetic algorithm work well), it would benefit from the availability of a dedicated software environment in which in a flexible way large numbers of genetic algorithm configurations can be tested on several different plant types. Such a software environment has been developed at Delft University of Technology (Spronck 1996). It is described in the next section.

THE SOFTWARE ENVIRONMENT "ELEGANCE"

"Elegance", which is an acronym for Engineering Laboratory for Experiments with Genetic Algorithms for Neural Controller Evolution, is a home-made software environment constructed to experiment with genetic algorithm configurations for the design of neural controllers, particularly in a reinforcement control situation. Elegance's view on the neural controller evolution process is shown in figure 5. The controller directs the plant with a control signal. This results in a plant output, which is sent back to the controller and to a history entity. The history entity can provide the controller with the last *n* plant outputs through so-called tapped-delay lines (TDLs). A setpoint generator indicates to the controller which desired plant output should be produced. The structure consisting of the plant, controller, setpoint generator and history entity is called the *control loop.*

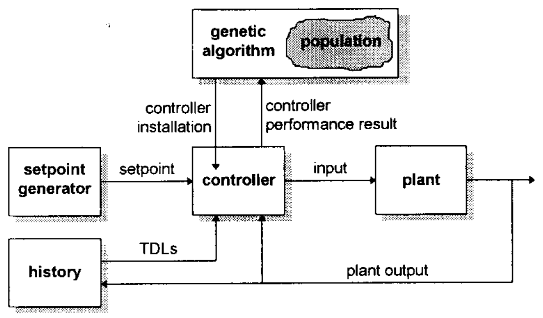

Figure 5: The basic Elegance control structure.

The genetic algorithm contains a population of controllers. The fitness of these controllers is determined by placing them, one-by-one, into the control loop, making the control loop run for a certain length of time, and by examining the results of this test run. The genetic algorithm generates new controllers by applying genetic operators on parent controllers selected from the population, determining the fitness of these new controllers, and inserting them into the population, replacing existing controllers. Since the purpose of Elegance is to test many different genetic algorithm configurations, there is offered a great flexibility in the design of the genetic algorithm. In the following we give a non-exhaustive list of Elegance's functionalities:

- Elegance supports both weight determination and the combination of architecture design and weight determination.
- Elegance supports both real valued chromosomes (Whitley et al. 1993) and binary valued chromosomes of fixed and variable length (Maniezzo 1993).
- Elegance supports both time-until-failure-based fitness (Whitley et al. 1993) and MSE-based fitness determination. MSE-based fitness can be configured to use a basic run length, which gets increased until the change in MSE falls below a certain threshold.
- Elegance supports several fitness scaling techniques, like ranking (Goldberg 1989).
- Elegance supports a complexity penalty and a premature failure penalty.
- Elegance supports a wide range of genetic operator types, among which several kinds of weight mutation, several kinds of connection presence mutation, granularity mutation, several kinds of crossover operators and the GA-simplex operator (most genetic operators come from Montana and Davis (Montana and Davis 1989), GA-simplex is designed by Maniezzo (Maniezzo 1993), and some operators have been designed specifically for Elegance).
- Elegance supports Whitley's adaptive mutation technique (Whitley et al. 1993).
- Elegance supports Thierens' treatment of competing conventions (Thierens et al. 1993).
- Elegance supports several kinds of replacement policies, like crowding (meaning that the individual to be replaced is selected deterministically from a randomly selected subset of the population) (Goldberg 1989).
- Elegance supports feedforward neural networks, layered feedforward neural networks and recurrent neural networks as neural controllers. For comparison, it also supports conventional PID controllers.
- Elegance supports several different plant simulations.
- It is easy to add new plants and new genetic operators to Elegance.

For completeness, some of Elegance's limitations should also be mentioned:

- Elegance only supports direct encoding (meaning that a neural network is always encoded to all its details).
- The chromosome structure design is fixed.
- As yet, there is no combination of genetic algorithms with a local optimisation technique supported. Such a combination would make a useful addition.

The object-oriented design of Elegance has been inspired by the design of Jarmulak's NeuroControl Workbench (Jarmulak 1994). There are six basic objects in the program. The object "project" contains the five other objects: the plant, the controller, the setpoint generator, the control loop display and the genetic algorithm (the history entity is implemented as part of the plant). These six objects can be defined and maintained separately. After they have been defined, an evolution run can be started, and when this run is finished, the results can be examined by running the best controller found.

CONTROLLING A BIOREACTOR SIMULATION

Preliminary experiments done with Elegance included the design of a neural controller for a pole balancing system simulation and the design of a neural controller for a trolley simulation. The pole balancing system, however, was found to be so very simple to control, that randomly generating neural controllers with five hidden nodes would produce a good pole balancing neural controller within a few hundred trials. This means that almost any genetic algorithm configuration would produce good results, and therefore that the pole balancing system is not a very good test case to decide if the application of genetic algorithms in neural controller design is a promising technique. The trolley was found to be easiest to control with a neural controller with no hidden nodes at all, or at most one hidden node. Although the evolution techniques proved their worth by reducing neural networks with more hidden nodes to neural networks with at most one hidden node, this indicates that the trolley is also not a very suitable system to test these genetic algorithm based techniques.

Elegance does, however, contain more complex plants in its plant base, from which the bioreactor simulation was selected to perform the first really challenging experiments. The bioreactor is a tank reactor containing a biological cell mass in water, which feeds on nutrient added to the tank, while water is drained from the tank at a rate equal to the nutrient input flow. The controller can adjust the flow rate. The objective of the controller is to stabilise the cell mass and the amount of nutrient in the tank at certain setpoints. The following formulas describe the bioreactor:

$$\begin{cases} \dfrac{dC}{dt} = -Cw + C(1-N)e^{\frac{N}{\gamma}} \\ \dfrac{dN}{dt} = -Nw + C(1-N)e^{\frac{N}{\gamma}} \dfrac{1+\beta}{1+\beta-N} \end{cases}$$

where C is the cell mass, N the amount of nutrient, w the flow rate, β the cell growth rate and γ the nutrient consumption rate. The bioreactor is highly non-linear and chaotic, and there are few stable setpoints (which are setpoints for which the flow rate can remain constant).

If our goal is to evolve a general neural controller for a bioreactor, at first it seems logical to use a random setpoint generator. However, it was decided to use a setpoint generator that would switch between two stable setpoints. The major reason not to use a random setpoint generator is that it would introduce a large amount of chance in the fitness determination process, especially with such a chaotic plant, which might obstruct the evolution process. This could mean that the resulting neural controller can *only* be used for those two setpoints. However, this is not necessarily the case. If a general solution to the control problem is easier to implement than a solution which can only be used for the two selected setpoints, it is likely that this general solution will be the one found by the genetic algorithm. The bioreactor constants were set at β = 0.02 and γ = 0.48. The two setpoints chosen were (C, N) = (0.1207, 0.8801), which is stable with a flow rate w = 0.75, and (C, N) = (0.2107, 0.7226), which is stable with a flow rate w = 1.2.

Elegance works with a predefined maximum number of nodes in the neural network configuration, from which it can remove hidden nodes, but to which it cannot add more nodes. The needed initial network configuration largely depends on the complexity of the problem. The choice of the maximum number of nodes should be made carefully. If it is too big, the evolution process will take much longer than necessary. If it is too small, the evolution process won't succeed. For the bioreactor, it was decided to choose a configuration for which Jarmulak had already found it would evolve into a good bioreactor identifier (Jarmulak 1995a). The reasoning behind this decision was that for a model-based controller the plant model is often by far the most complex part of the controller, so the needed identifier complexity could probably be about the same as the needed controller complexity for Elegance. The neural controller configuration was therefore set to:

- 8 input nodes, consisting of two setpoint nodes (the desired cell mass and the desired amount of nutrient), two plant output nodes (again, the cell mass and the amount of nutrient), and two TDLs (totalling four nodes: the previous plant outputs, and the plant outputs before that).
- two hidden layers of 20 nodes each.
- 1 output node, the flow rate.

As activation function an arctangent was chosen with a range of [-1,+1]. This configuration has a total of 580 possible connections, which makes the solution space quite large indeed. The genetic algorithm configuration was designed as follows:

- Encoding was done with real values for the weights.
- Weight initialisation was done in a range of [-5,+5], with a connection presence chance of 0.5 (meaning that about 50 percent of all possible connections would be initially activated in a neural network).
- Fitness determination was MSE-based, with ranking as scaling technique. The basic run length was set to 500 time steps (one time step equalling 0.1 simulated seconds), which would be continuously increased with 100 time steps until the change in MSE would have dropped below 0.02. Premature failure is, at least in theory, not possible with the bioreactor.
- Population size was set to 100.
- Elitism was applied (meaning that the best individual in the population would never be selected for replacement).

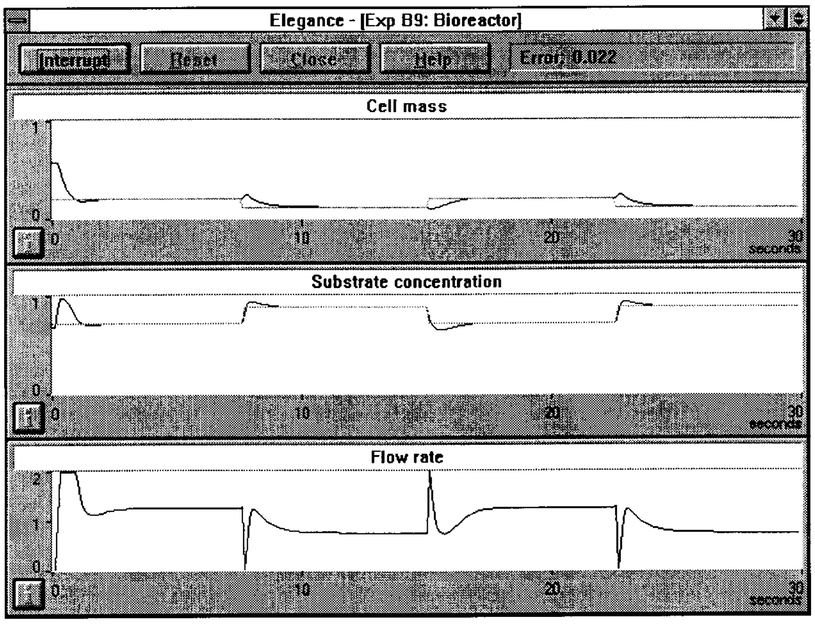

Figure 6: A control loop of a bioreactor neural controller evolved with Elegance. This controller reaches an MSE of about 0.019. It consists of a feedforward neural network with only seven hidden nodes. In the first two graphs, the 'square' lines indicate the setpoints, while the curves indicate the actual values.

- Duplicate checking was performed (meaning that the genetic algorithm would not allow duplicates in the population).
- Viability checking was applied (meaning that the genetic algorithm would not allow neural networks in the population which have output nodes which are not connected via some path to at least one of the input nodes).
- A treatment for competing conventions was applied.
- As replacement policy crowding was applied.
- Incest prevention was applied with three alleles (meaning that to be allowed to be used as parents, two chromosomes should differ by at least three alleles).

The following genetic operators were used:

- Biased weight mutation, which changes 10 percent of the weights of a parent chromosome within a small range (Montana and Davis 1989).
- Node existence mutation, which either removes a node completely or activates all connections to and from a node (Spronck 1996).
- Connectivity mutation, which removes some connections and adds others (Maniezzo 1993).
- One-point crossover, shown in figure 1.
- Nodes crossover, which creates a child chromosome by copying repeatedly from a random parent a node with all its incoming connections (Montana and Davis 1989).
- Randomisation, which in effect generates almost randomly new chromosomes, slightly based on an existing chromosome. This operator is equal to Whitley's mutation operator (Whitley et al. 1993).

The experiment was performed on a Pentium 90 PC. The evolution process needed a bit less than half-a-minute to generate one controller, including the fitness determination. This means that in a period of 24 hours about 3000 controllers could be generated. After the generation of 2000 controllers, the best controller generated had an MSE of 0.025. The evolution was interrupted at that point, and the controller was tested in the control loop. The result was found to be a good bioreactor neural controller, which could efficiently direct the bioreactor to the defined setpoints, and keep it there with a constant flow rate. The resulting neural controller had a configuration with two hidden layers, with 19 nodes in the first and 10 nodes in the second layer.

This was the result of the first experiment. More experiments were performed. Although the genetic algorithm used in the first experiment was found to be a good choice (small changes in the algorithm almost always harmed the evolution run), the initial neural network configuration was found to be less than optimal. In later experiments, a simple feedforward neural network (without the concept of layers) with a maximum of ten hidden nodes and no TDLs was used as initial neural network configuration. Surprisingly, in about 2 hours and the generation of 1500 controllers this configuration could be evolved as a good bioreactor neural controller with no more than seven hidden nodes. The genetic algorithm used was the same as for the first experiment, except that the weight initialisation was in a range of [-10,+10] and no treatment of competing conventions was used. The control results are shown in figure 6.

The experiment was repeated several times using the same configuration, with comparable results. Sometimes the resulting neural controller would have even fewer hidden nodes, with five hidden nodes as a minimum. However, when using an initial neural network configuration with a *maximum* of five (instead of ten) hidden nodes, which should, in principle, be sufficient, the evolution process would often fail to produce a good controller. This indicates that the process needs some elbow room to play with.

CONCLUSIONS

Comparing the results of Elegance (Spronck 1996) with the results of the NeuroControl Workbench (Jarmulak et al. 1995a) on a bioreactor simulation, we find that they need similar training times to reach similar (good) results on neural networks with equivalent configurations. This shows that the application of genetic algorithms for neural controller design is at least competitive with conventional techniques. Elegance also showed that in some cases the model-based approach might work less well than the genetic reinforcement approach, since a neural controller might need a simpler configuration than a neural model of the plant. Besides that, there are a number of advantages of using genetic algorithms:

- Genetic algorithms can be used to design neural controllers with any neural network configuration, while conventional techniques are often limited to specific configurations, for instance to feedforward networks.
- Genetic algorithms need nothing more than a controller performance evaluation to work. In principle, such an evaluation is always available. Conventional techniques are often dependent on extra information, like the derivative of the error function.
- Genetic algorithms can be used to not only determine the neural network connection weights, but also the architecture. Most conventional techniques are limited to weight determination.

Still, although it has been shown that the technique is viable, it isn't at all clear which genetic algorithm configuration features lead to an efficient, successful evolution run, particularly in the design of neural controllers in a reinforcement situation. Elegance can be used to run a large number of experiments in this respect, and may therefore be a useful tool in this subject of research.

REFERENCES

Goldberg, D.E. 1989. *Genetic Algorithms in Search, Optimization & Machine Learning.* Addison-Wesley Publishing Company, Inc.

Holland, J.H. 1992. *Adaptation in Natural and Artificial Systems*, 2nd edition. MIT Press/Bradford Book Edition, Cambridge, Massachusetts (first edition 1975).

Jarmulak, J. 1994. *NeuroControl Workbench.* MSc thesis, Delft University of Technology, Faculty of Technical Mathematics and Informatics, Delft.

Jarmulak, J.; E.J.H. Kerckhoffs; and L.J.M. Rothkrantz. 1995a. "Universal Approach to Neural Process Control Illustrated on a Biomass Growth Model." In *Proceedings of the 2nd IFAC/IFIP/EurAgEng Workshop on Artificial Intelligence in Agriculture*, Wageningen, the Netherlands.

Jarmulak, J.; E.J.H. Kerckhoffs; and L.J.M. Rothkrantz. 1995b. "A Software Environment for Neural Control of Simulated Plants." In *Neural Network World*, Volume 6, 873-892.

Maniezzo, V. 1993. "Searching among Search Spaces: hastening the genetic evolution of feedforward neural networks." In *Artificial Neural Nets and Genetic Algorithms*, R.F. Albrecht, C.R. Reeves & N.C. Steel, eds. Springer-Verlag, Wien, New York, 635-642.

Montana, D.J. and L. Davis. 1989. "Training Feedforward Neural Networks Using Genetic Algorithms." In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, San Mateo, California, 762-767.

Spronck, P.H.M. 1996. *Elegance: Genetic Algorithms in Neural Reinforcement Control.* MSc thesis, Delft University of Technology, Faculty of Technical Mathematics and Informatics, Delft.

Thierens, D.; J. Suykens; J. Vandewalle; and B. de Moor. 1993. "Genetic Weight Optimization of a Feedforward Neural Network Controller." In *Artificial Neural Nets and Genetic Algorithms*, R.F. Albrecht, C.R. Reeves & N.C. Steel, eds. Springer-Verlag, Wien, New York, 658-663.

Whitley, D.; S. Dominic; R. Das; and C.W. Anderson. 1993. "Genetic Reinforcement Learning for Neurocontrol Problems." In *Machine Learning*, Kluwer Academy Publishers, Boston, Volume 13, 103-128.

AVAILABILITY OF SOFTWARE

The software presented in this paper can be downloaded via the following WWW-pages:

http://ford.twi.tudelft.nl/~jacek/ncwb.html
(NeuroControl Workbench, for MS-DOS).

http://web.inter.nl.net/users/p.spronck/e_eleg.htm
(Elegance, for MS-Windows 3.1, 95 and NT).